Public Articles

June 2002

- Even more string grids part II
- Extending your applications with COM Automation Part 1
- Using Web Services
- Source Code

May 2002

- Creating a Web server using TServerSocket
- Display-optimal DIB section bitmaps
- Fast bitmap zooming and scrolling
- Source Code

April 2002

- Using Windows file mapping for inter-process communications Part 1
- Even more string grids part I
- What's in a (form) name
- Source Code

March 2002

- Quick and easy toolbar customization
- Saving and loading components
- The basics of multi-tier applications
- Source Code

February 2002

- An HTML calendar generator
- An introduction to image color management
- Menu templates
- Setting an Open dialog's current directory
- Source Code

January 2002

- Abstract data types in C++Builder
- Custom Open dialogs part II
- Separating IDE-generated code
- Source Code

December 2001

- Buttons buttons and buttons
- Custom open dialogs part I
- IDE Tip- Selecting the form
- Writing aggregate components
- Source Code

November 2001

- A fancy expandable dialog box
- A simple expandable dialog box
- Custom buttons part III
- Persistent image lists
- Source Code

October 2001

- Custom buttons part II
- Events and callback functions
- Sharing code between database projects- a look at queries
- Source Code

Sepetember 2001

- Components- build beg or buy
- Custom buttons part I
- Drawing transparent images
- Under construction
- Source Code

August 2001

- A scrolling marquee component
- Creating a multiselect tree-view
- Managing to-do lists
- Using Delphi code in C++Builder
- Source Code

July 2001

- Get the message out
- Including hot links in TRichEdit
- Preprocessor macros
- Quick and easy re-sizeable controls
- Source Code

June 2001

- An enhanced type safe TList part 2
- Smart list views
- Using custom tooltips
- Source Code

May 2001

- A Custom Exception Handler
- A home cooked Outlook bar
- An enhanced type safe TList
- Custom popup controls
- TScreening room
- Source Code

April 2001

- Custom drawing the trackbar control
- Folder change notification
- Printing Bitmaps III- Wrapping Things Up

- Source Code

March 2001

- Printing bitmaps II- sending a DIB to the printer
- Setting the system time
- Status bar stuff
- Transparent 2000
- Windows hooks
- Source Code

February 2001

- C++Builder file types
- Dealing with network connections
- Finding files part 4
- Opening and closing the coffee holder
- Resource animations
- Where is the cursor
- Source Code

January 2001

- A UTC TDateTime function
- Finding files part 3
- How to determine the drive type
- Printing bitmaps part I
- TStringList as a template array class
- Source Code

December 2000

- Creating a flat combo box
- Getting aliases and table names
- Getting file version information
- Using Visual C++ DLLs with C++Builder
- Adding a background image to a list

- Source Code

November 2000

- Advanced string grid techniques
- Finding files part 2
- Using timers
- Validating URLs
- Source Code

October 2000

- Active Server Objects
- Creating a no-VCL Windows application
- More string grids
- Taking a screen shot
- Working with Windows messages
- Source Code

Sepetember 2000

- Building stand-alone EXEs
- Finding files
- Simulating keystrokes
- Working with string grids
- Source Code

August 2000

- Adding controls to a DBGrid
- Command line builds with make
- Hidden treasures of Sysutils Part 4
- IDE command-line options
- Loading tree views the easy way
- Understanding AnsiString's c str() function
- Source Code

July 2000

- Customized form navigation
- Exceptions and databases
- Hidden treasures of Sysutils part 3
- Understanding function pointers
- Using the ADO access components

June 2000

- Adding items to the shell context menu
- Error and exception handling strategies
- Hidden treasures of Sysutils Part 2
- Using the shell context menu
- Source Code

May 2000

- Hidden treasures of Sysutils Part 1
- Improving build times with pre-compiled headers
- Pre-compiled header tips
- Stepping into the VCL source
- Using the VCL source to track down bugs
- Source Code

April 2000

- Changing window styles on the fly
- Form designer tips
- Making ActiveX controls persistent
- Using sockets
- When to use fastcall
- Source Code

March 2000

- Designing with data modules
- Dragging images with TImageList

- Exploiting data module form inheritance
- Making marvelous message dialogs
- Storing binary data in the registry
- Source Code

February 2000

- Customized radio groups
- Getting shell item information
- Image shaped forms
- Overlays for all occasions
- To auto-create or not to auto-create
- Source Code

January 2000

- Changing common dialog button captions
- Enumerating the shell namespace
- Fast updates with virtual list views
- Using Perl compatible regular expressions in VCL applications
- Source Code

December 1999

- ActiveForms part III - MIDAS
- Building a no-VCL Automation Server
- Changing common dialog button captions
- Extending the common dialogs
- Programming the recycle bin
- Source Code

November 1999

- ActiveForms part II - deployment
- Dynamic C++ arrays using STL
- Shell and common control versions
- Single-instance applications

- Using file associations
- Source Code

October 1999

- ActiveForms Part I
- Faster rich edit syntax highlighting
- Using STL containers
- Who is using your program
- Source Code

Sepetember 1999

- Components- build beg or buy
- Creating better data implementations
- Custom buttons part I
- Determining the OS version
- Do you need a TDatabase
- Drawing a selection rectangle
- Source Code

August 1999

- Determining compiler version
- Hosting forms within a main form
- RAS update
- Registering AnsiString property editors
- Understanding VCL ownership and parentage
- Verifying data using CRC
- Hosting forms in a main form
- Source Code

July 1999

- Drawing transparent images
- Owner-drawn list boxes
- Ownership vs

- Under construction
- Using persistent fields

June 1999

- Drawing with metafiles
- Writing a Performance Monitor
- Drawing with metafiles
- Pinging a server
- Compiler speed
- Handling differences in integer representation

May 1999

- Sending mail
- Transferring form data
- A simple fly-over label component

April 1999

- Using RAS part 2
- Put a plasma in your form
- Using properties in C++ classes
- Enumerating components
- Dealing with sets
- Transferring form data

March 1999

- Spawning external applications
- Waiting on a spawned process
- How Windows locates files
- Using RAS part 2
- Reappearing forms
- Using RAS part 1

February 1999

- Using callbacks in DLLs
- Constructing constructors
- Constructor initializer lists
- Concrete data types
- Word 97 OLE Automation

January 1999

- Implementing a Help cursor
- A component for reading version information from a program file
- Working with version information
- TStrings and comma-separated text

December 1998

- Mastering the TTreeView component part 2
- Creating a URL label
- WriteLn for C++Builder
- What's in a name(space)
- Horizontal scrollbars for list boxes

November 1998

- Mastering TTreeView component part 1
- Add a handle
- Displaying multiple main forms
- Trouble on the border
- Transferring TRichEdit text
- Common controls
- Setting the TRichEdit font

October 1998

- Sharing data and methods between forms

- Packing database tables
- Low-level wave audio part 3
- RIFF update from Part 1

Sepetember 1998

- Using a VCL form in a DLL
- MDI child forms in a DLL
- Using TStringGrid with graphics
- Debugging with diagnostic macros
- How can I use BLOBs with C++Builder
- TRichEdit and RTF
- Horizontal scrollbars for list boxes

August 1998

- Low-level wave audio part 2
- Wave-out messages
- Playing wave resources
- Using OWL TDialogs in a VCL application

July 1998

- Message-handling for non-visual components
- Low-level wave audio Part I
- Automating table activation
- Sorting a TList
- Displaying your bitmaps quickly

June 1998

- Detecting disk errors
- File operations
- Formatting a floppy from within an application

- Getting error message text
- Using Panel components to simplify resizing
- Exploring the Menu Designer

May 1998

- Using packages in C++Builder 3
- Owner-drawn list boxes
- MDI 101
- MDI made easy
- Capturing a form to disk

April 1998

- What's new in C++Builder 3
- April 1998
- Building graphic applications
- Manipulating memory with TMemoryStream
- Are you a good tipper

March 1998

- File I O
- Today's buzzword- Streams
- Synchronizing two list boxes
- Quick ZDtips

February 1998

- Replacing variables in HTML documents
- February 1998
- Creating a font toolbar
- TQuery with parameters

January 1998

- Persistence pays off
- Cheating with TUpdateSQL
- Rotated fonts
- Drawing text 101
- Use a mutex to achieve synchronization

December 1997

- DB dilemmas
- The right list for the right job
- Starting your application minimized
- Reconnoitering resources
- Learning the ropes of TStringGrid

November 1997

- Building components part 3
- Putting the Registry to work
- Implementing a recent-files list
- Bitmaps on demand
- Quick C++Builder tips

October 1997

- Using the Find and Replace dialog boxes
- Building components part 2
- Property macros
- Owner-drawn status bars
- Structuring your code to avoid leaks

Sepetember 1997

- Sprucing up your status bars
- Simple or complex status bars

- Building components part 1
- What a drag!
- A drop in the bucket
- Command-prompt tips

August 1997

- Using OWL classes in C++Builder
- The magic of namespaces
- Undocumented Winsys
- Get a Grep!
- An AnsiString class reference
- All set

July 1997

- Strings strings strings!
- Using forms in console applications
- What's a console application
- Using Delphi components in C++Builder
- Finding a console window

June 1997

- Incorporate custom message-handling in your applications
- C++Builder extensions to C++
- Controlling the cursor
- Display forms without captions
- Getting IN to SQL
- SpeedButton tricks
- Editing string lists in the code-editing window

Even more string grids, part II

by Damon Chandler

In the April 2002 issue I showed you how to add controls to a string grid's cells. This month, I'll focus on the grid's in-place editor. I'll review the string grid's editor-related events, show how to access the existing editor in a standard `TStringGrid` instance, and explain how to create your own in-place editor class.

The TInplaceEdit class

As most of you know, when the `Options` property of a `TStringGrid` object contains the `goEditing` enumerator, the string grid permits in-place editing of a cell's contents via a small edit control that's displayed over the cell. This edit control, which is created and managed by the `TCustomGrid` class, is an instance of a `TCustomMaskEdit` descendant class called `TInplaceEdit`. The `TCustomGrid` class provides access to its `TInplaceEdit` instance via the `InplaceEditor` property.

From an application end user's perspective, the grid's in-place editor is a standard edit control. From the perspective of a developer using the `TStringGrid` class, the in-place editor is essentially a built-in `TCustomMaskEdit`.

I say the editor is "built in" because there's no formal means of accessing the grid's `TInplaceEdit` instance—the `TCustomGrid::InplaceEditor` property is protected, and is thus accessible only in descendant classes. However, the in-place editor forwards certain events to its associated grid. This provides users of the `TStringGrid` class *indirect* control over what the end user can input into the grid's cells. In most cases, this limited control suffices; in other cases, it doesn't.

Later I'll show you how to gain direct access to the grid's `TInplaceEdit` instance. For now, let's focus on the standard, indirect means of controlling the in-place editor.

TStringGrid's editor-related events

Before the user can edit a cell, the `TCustomGrid` class has to position the in-place editor over that cell. The grid does this via its internal `UpdateEditor()` method, which, in turn, calls the `TInplaceEdit::UpdateContents()` method. Here's a condensed C++ translation of the `UpdateContents()` method:

```
void __fastcall
```

```
  TInplaceEdit::UpdateContents()
{
  Text = "";
  EditMask = Grid->GetEditMask(
    Grid->Col, Grid->Row);
  Text = Grid->GetEditText(
    Grid->Col, Grid->Row);
}
```

As you can see from this code, the editor first clears its current text, then calls the grid's `GetEditMask()` method (assigning the return value to its `EditMask` property), and then calls the grid's `GetEditText()` method (assigning the return value to its `Text` property).

The `TCustomGrid::GetEditMask()` and `GetEditText()` methods correspond, respectively, to the grid's `OnGetEditMask` and `OnGetEditText` events. Both of these events have the same signature, which is of the following form:

```
typedef void __fastcall
  (__closure *TGetEditEvent)(
    System::TObject* Sender,
    long Col, long Row,
    AnsiString& Value);
```

The `Sender` parameter refers to the grid itself. The `Col` and `Row` parameters specify the indices of the "to-be-edited" cell. You use the `Value` parameter to specify either the editor's mask or its text (depending on which event you're working with).

The OnGetEditMask event

As I mentioned, the `TInplaceEdit` class is a `TCustomMaskEdit` descendant--this design provides the in-place editor with inherent text-filtering capabilities. If you were working with a `TMaskEdit` object, you'd normally specify its text mask by using the `TCustomMaskEdit::EditMask` property. In the string grid however, you don't have access to the in-place editor; rather, you specify the text mask by using the `TStringGrid::OnGetEditMask` event. For example, if the third (non-fixed) column of your grid represents dates, you'd use the `OnGetEditMask` event like so:

```
void __fastcall TForm1::
 StringGrid1GetEditMask(TObject *Sender,
  int Col, int Row, AnsiString& Value)
{
  if (Col == StringGrid1->FixedRows + 2)
```

```
  {
    Value = "!99/99/00;1;_";
  }
}
```

The OnGetEditText event

Recall from the definition of the `TInplaceEdit::UpdateContents()` method that the editor replaces its current text with the return value of the grid's `GetEditText()` method. The `TStringGrid` class defines the `GetEditText()` method to simply return the text of the cell over which the editor is placed. Before returning though, the string grid will call its `OnGetEditText` event handler (if one is assigned), giving you an opportunity to change the text assigned to the editor. For example, if you want the editor to display the cell's current text surrounded by parentheses, you'd use the `OnGetEditText` event as follows:

```
void __fastcall TForm1::
 StringGrid1GetEditText(TObject *Sender,
  int Col, int Row, AnsiString& Value)
{
  Value = "(" + Value + ")";
}
```

The OnSetEditText event

Whenever the user changes the editor's contents, the `TCustomGrid` class calls its private `UpdateText()` method, which, in turn, calls the grid's virtual `SetEditText()` method. This latter method is passed the editor's current text. Within its definition of the `SetEditText()` method, the `TStringGrid` class first assigns the specified text (i.e., the editor's text) to the corresponding cell and then calls its `OnSetEditText` event handler, if one is assigned. Here's what the `OnSetEditText` event looks like:

```
typedef void __fastcall
  (__closure *TSetEditEvent)(
    System::TObject* Sender,
    long Col, long Row,
    const AnsiString Value);
```

Because the string grid updates the cell's text *before* firing its `OnSetEditText` event, the `Value` parameter will always be the same as the text held in the cell located at `Col` and `Row`. Thus, the `OnSetEditText` event notifies you that both the editor's text *and* the cell's text have changed (both to the same value).

The OnKey* events

The `TInplaceEdit` class augments the virtual `TWinControl::KeyDown()`, `KeyPress()`, and `KeyUp()` methods to forward keyboard-related events to its associated grid. This allows you to use the string grid's `OnKeyDown`, `OnKeyPress`, and `OnKeyUp` events to monitor (and perhaps filter) keystrokes while a cell is being edited. For example, if you want to convert all characters to uppercase, you'd use the `OnKeyPress` event like so:

```
void __fastcall TForm1::
  StringGrid1KeyPress(TObject *Sender,
    char& Key)
{
  Key = std::toupper(Key);
}
```

Similarly, suppose you want to allow the user to press the escape key to restore the cell's text to its value prior to editing. In this case, you'd use a combination of the `OnGetEditText` and `OnKeyDown` events as follows:

```
void __fastcall TForm1::
 StringGrid1GetEditText(TObject *Sender,
  int Col, int Row, AnsiString& Value)
{
  // NOTE: old_cell_text_ is an
  // AnsiString-type member of TForm1
  old_cell_text_ = Value;
}

void __fastcall TForm1::
  StringGrid1KeyDown(TObject* Sender,
    WORD& Key, TShiftState Shift)
{
  if (Key == VK_ESCAPE &&
      StringGrid1->EditorMode)
  {
    const int col = StringGrid1->Col;
    const int row = StringGrid1->Row;
    StringGrid1->Cells[col][row] =
      old_cell_text_;
    StringGrid1->EditorMode = false;
  }
```

```
}
```

Notice the use of the `EditorMode` property in this code. `EditorMode` is a Boolean property (first introduced in the `TCustomGrid` class) that simply indicates whether or not a cell is being edited. (Note that you can also set `EditorMode` to `true` to initiate the cell-editing process.)

Input validation

Here's an example that demonstrates how to use the string grid's `EditorMode` property along with the `OnKeyPress`, `OnSelectCell`, and `OnExit` events to determine when a user has finished editing a cell's text. This scheme allows you to validate the user's input. Here's the code:

```
// trivial validation function
bool TForm1::IsValidEntry()
{
  const int col = StringGrid1->Col;
  const int row = StringGrid1->Row;
  const AnsiString text(
    StringGrid1->Cells[col][row]
    );
  if (text != "some_valid_entry")
  {
    Application->MessageBox(
      "Please input a valid value.",
      "Input Error", MB_OK);
    return false;
  }
  return true;
}

// OnKeyPress event handler
void __fastcall TForm1::
  StringGrid1KeyPress(TObject *Sender,
    char& Key)
{
  if (
    Key == '\r' &&
    StringGrid1->EditorMode &&
    !IsValidEntry()
    )
  {
    Key = 0;
```

```
  }
}

// OnSelectCell event handler
void __fastcall TForm1::
  StringGrid1SelectCell(TObject *Sender,
    int Col, int Row, bool& CanSelect)
{
  if (StringGrid1->EditorMode)
  {
    CanSelect = IsValidEntry();
    if (!CanSelect)
    {
      Editor_->SelectAll();
    }
  }
}

// OnExit event handler
void __fastcall TForm1::
  StringGrid1Exit(TObject *Sender)
{
  if (!IsValidEntry())
  {
    ActiveControl = StringGrid1;
    PostMessage(StringGrid1->Handle,
      WM_KEYDOWN, VK_F2, 0);
  }
}
```

Accessing the existing TInplaceEdit control

Earlier I mentioned that the `TCustomGrid` class positions its in-place editor from within the `TCustomGrid::UpdateEditor()` method. Here's a C++ translation of that method:

```
void __fastcall
  TCustomGrid::UpdateEdit()
{
  if (CanEditShow())
  {
    if (FInplaceEdit == NULL)
    {
```

```
      FInplaceEdit = CreateEditor();
      FInplaceEdit->SetGrid(this);
      FInplaceEdit->Parent = this;
      UpdateEditor();
    }
    else if (Col != FInplaceCol ||
      Row != FInplaceRow)
    {
      HideEdit();
      UpdateEditor();
    }
    // position and show the editor...
  }
}
```

The `TCustomGrid` class stores a pointer to the in-place editor in its private `FInplaceEdit` member. If this member is `NULL`, a call is made to the virtual `CreateEditor()` method to create the editor. And, after the editor is created, the grid's assigns itself (`this`) as the editor's `Parent`—this assignment allows you to access the in-place editor from outside the scope of the `TStringGrid` class. Namely, although you can't access the string grid's `InplaceEditor` property (because it's protected), you *can* access the string grid's `Controls` property. Because the grid sets itself as the in-place editor's `Parent`, and because the grid contains no other controls (unless you explicitly put them there yourself), `StringGrid->Controls[0]` will always to refer to the in-place editor (if it has been created).

When can you safely access the in-place editor via the `Controls` property? Well, recall that the grid's `GetEditMask()` method is called from within the `TInplaceEdit::UpdateContents()` method. Because the in-place editor is deleted only within the grid's destructor, the editor is guaranteed to exist anytime after the string grid's `OnGetEditText` event handler is first called. So an easy way to grab a pointer to the editor is to use the `OnGetEditText` event, like so:

```
#include <cassert>
void __fastcall TForm1::
 StringGrid1GetEditMask(TObject *Sender,
  int Col, int Row, AnsiString& Value)
{
  if (Editor_ == NULL)
  {
    assert(StringGrid1->
      ControlCount == 1);
    Editor_ =static_cast<TInplaceEdit*>(
      StringGrid1->Controls[0]);
  }
}
```

Here, `Editor_` is a class member of type `TInplaceEdit*` that's initialized to `NULL` in the form's constructor.

With a pointer to the in-place editor, you can use properties and methods of the `TInplaceEdit` class that you wouldn't otherwise have access to. For example, if you want to give the editor a thin-line border and a new background color, you'd do the following:

```
const LONG style = GetWindowLong(
  Editor_->Handle, GWL_STYLE);
SetWindowLong(Editor_->Handle,
  GWL_STYLE, style | WS_BORDER);
Editor_->Brush->Color = clInfoBk;
Editor_->Invalidate();
```

Similarly, if you'd rather have newly-entered text appended to the end of the current text (rather than replace it), you could use the `TInplaceEdit::Modified` property and `Deselect()` method as follows:

```
void __fastcall TForm1::
  StringGrid1KeyPress(TObject *Sender,
    char &Key)
{
  if (Editor_ && !Editor_->Modified)
  {
    Editor_->Deselect();
  }
}
```

Unfortunately, having access to the string grid's in-place editor isn't as promising as it might sound. For one, the `TInplaceEdit` class fails to expose several key properties (e.g., `CharCase`, `Font`). Second, and more importantly, many of the changes that you'll likely want to make require knowledge of when the editor is positioned and shown. In the previous code snippet, for example, the `Modified` property is queried each time a character-key is pressed. It would make more sense to deselect the text only after the editor is first shown.

Using a custom TInplaceEdit control

I just mentioned that the `TInplaceEdit` class is somewhat restrictive because it lacks and/or fails to expose several key properties and events. Fortunately, the `TCustomGrid` class provides a way to overcome these limitations; namely via the `CreateEditor()` method.

Recall from the definition of the `TCustomGrid::UpdateEdit()` method that if the grid's `FInplaceEdit` member is `NULL`, the grid creates the in-place editor via a call to the virtual `TCustomGrid::CreateEditor()` method. Here's how that method is declared:

```
virtual TInplaceEdit*
  __fastcall CreateEditor();
```

Because the `TCustomGrid` class assigns the return value of `CreateEditor()` method to its `FInplaceEdit` member, a descendant class can override the `CreateEditor()` method to specify a custom in-place editor.

In short, if your application requires functionality beyond that provided by the `TInplaceEdit` class, you can effectively instruct the grid to use your own `TInplaceEdit` descendant class by overriding the `CreateEditor()` method. **Listing A** provides an example of a `TInplaceEdit` descendant class, which I've named `TInplaceEditEx`. This class does nothing more than expose the `BorderStyle`, `CharCase`, `Color`, and `Font` properties.

A simple TStringGrid descendant class

As I just mentioned, in order to have the string grid use an instance of the `TInplaceEditEx` class instead of the default `TInplaceEdit` object, you need to override the `TCustomGrid::CreateEditor()` method. To demonstrate this, I've created a barebones `TStringGrid` descendant class, `TStringGridEx`, which is declared in **Listing B**.

Whereas the `TCustomGrid` class creates the in-place editor from within its `UpdateText()` method, the `TStringGridEx` class creates an instance of the `TInplaceEditEx` class from within the `TStringGridEx` constructor, like so:

```
__fastcall TStringGridEx::
  TStringGridEx(TComponent* Owner) :
    TStringGrid(Owner),
    Editor_(new TInplaceEditEx(this))
{
  Editor_->Parent = this;
}
```

Notice that a pointer to the `TInplaceEditEx` instance is stored in the private `Editor_` member. Accordingly, from within the (re)definition of the `CreateEditor()` method, the `TStringGridEx` class simply return `Editor_`'s value, like so:

```
TInplaceEdit* __fastcall TStringGridEx::
```

```
  CreateEditor()
{
  return Editor_;
}
```

That's all there is to it—the string grid will now use the specified `TInplaceEditEx` object instead of its default editor.

As you can see from **Listing B**, I've redefined (and exposed) the `InplaceEditor` property to provide external access to `Editor_`. This eliminates the hassle of typecasting the return value of the `Controls` property. Moreover, because the editor is created in the class constructor, you don't have to worry about when it's safe to access the editor. (Recall that with a standard `TStringGrid` object, we had to rely on the `OnGetEditMask` event to determine when it was safe to access `Controls[0]`.) For example, if `StringGridEx1` is an instance of the `TStringGridEx` class (created at design time), you could set its `InplaceEditor`'s properties from within its parent form's constructor like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  StringGridEx1->InplaceEditor->
    BorderStyle = bsSingle;
  StringGridEx1->InplaceEditor->
    CharCase = ecUpperCase;
  StringGridEx1->InplaceEditor->
    Color = clInfoBk;
  StringGridEx1->InplaceEditor->
    Font->Color = clRed;
}
```

Conclusion

In this article, I've presented three ways to access a string grid's in-place editor: (1) indirect access by using the string grid's `OnGetEditMask`, `OnGetEditText`, and `OnSetEditText` events; (2) direct access by using the `Controls` property; and (3) direct access by creating `TInplaceEdit` and `TStringGrid` descendant classes.

The third approach offers the most flexibility because you can tailor your `TInplaceEdit` descendant class to suit the specific needs of your application. See the sample code that accompanies this article (available at **www.bridgespublishing.com**) for some more examples of working with the in-place editor.

Listing A: ***Declaration of the TInplaceEditEx class***

```
#include <cassert>
class TInplaceEditEx : public TInplaceEdit
{
public:
  __fastcall TInplaceEditEx(TComponent* Owner)
    : TInplaceEdit(Owner) {}

  __property BorderStyle;
  __property CharCase;
  __property Color;
  __property Font;
  // publish other properties as needed...
};
```

Listing B: ***Declaration of the TStringGridEx class***

```
#include <grids.hpp>
class TStringGridEx : public TStringGrid
{
public:
  __fastcall TStringGridEx(TComponent* Owner);
  __property TInplaceEditEx* InplaceEditor =
    {read = Editor_};

protected:
  TInplaceEdit* __fastcall CreateEditor();

private:
  TInplaceEditEx* Editor_;
};
```

Extending your applications with COM Automation, Part 1

by David Bridges

Borland’s VCL is key to the power of Delphi and C++Builder. It allows developers to use a vast and powerful set of software components without knowing very much about how they are implemented. Using components, useful applications can be built with very little work—the developer only needs to fill in the details once the building blocks are in place.

Using the VCL and other C++ classes works just fine as long as you stay within C++Builder. But suppose that you want to add extensibility and customization to your application, so that other programs can control it. Maybe you want to implement a "macro" language to automate certain tasks of the program. The best way to do this is to add Component Object Model (COM) interfaces to your program.

Most programmers today are familiar with COM under its latest incarnation—ActiveX. Actually, ActiveX is a very specific type of COM object that exposes interfaces that allow it to be easily integrated into development environments such as C++Builder, and can also be embedded and scripted in an HTML page. But you don’t need to create ActiveX components in order to expose your application objects to the outside world—a simple COM Automation interface will do.

A COM primer

If you’re not familiar with COM, it is the basic underlying technology behind most software components designed for Microsoft Windows. VCL components are very similar to COM components—they both have similar ways of exposing their properties and methods. An explanation of how COM works is much too involved to be covered in this article, but a good book on the subject is *Inside COM* by Dale Rogerson (Microsoft Press). Luckily, C++Builder provides excellent tools for developing COM components, so only a basic understanding of the details is required.

COM uses a client-server model. A COM server exposes interfaces that a client can use. The client never accesses the actual objects, only the objects’ interfaces. This differs from C++ objects, including VCL objects. When you use C++ objects, your code makes calls directly into the code of the object. But with COM, you retrieve an interface to an object, and then use the interface. The power of COM comes from the fact that the server objects are completely hidden from the client. They may be implemented in a different language—they may even be on a different machine. The interface handles all the details of communication between server and client. You can add these interfaces to your C++Builder projects so that your applications can be controlled from Visual Basic, MS Word, Delphi, and even VBScript or JavaScript.

The dispatch interface

COM allows a class to have multiple interfaces. This is similar to a C++ class that uses multiple inheritances. For example, suppose there is a C++ class C that is derived from both A and B. Class C inherits all of the properties and methods of classes A and B. COM doesn't allow one class to inherit any code from another class, but it does allow two classes to implement the same interface. Since the client using the objects only knows about the interface, the actual object being used (the implementation) is transparent to the client. This is how COM provides *polymorphism*, one of the most important requirements of object-oriented development.

A COM interface is like an abstract base class in C++. All COM objects must implement one required interface, `IUnknown`. The primary purpose of the `IUnknown` interface is to expose other interfaces. `IUnknown` contains a method called `QueryInterface()` that can be used to "ask" an object if it has a particular interface. The next most important COM interface is `IDispatch`. The `IDispatch` interface exposes properties and methods. Its primary methods are `GetIDsOfNames()` and `Invoke()`. The `GetIDsOfNames()` method allows a client to see what properties and methods an interface has, without knowing anything at all about the interface. It's like asking the interface, "Tell me what you can do". The `Invoke()` method is used to access the properties and methods of the interface. Each property or method has an ID number, which is used as one of the parameters to the `Invoke()` function. A component which implements the `IDispatch` interface is called an *automation server*. A COM client that accesses the server through its `IDispatch` interface is called an *automation controller*. Microsoft Word is an example of an automation server, and VBScript is an example of an automation controller.

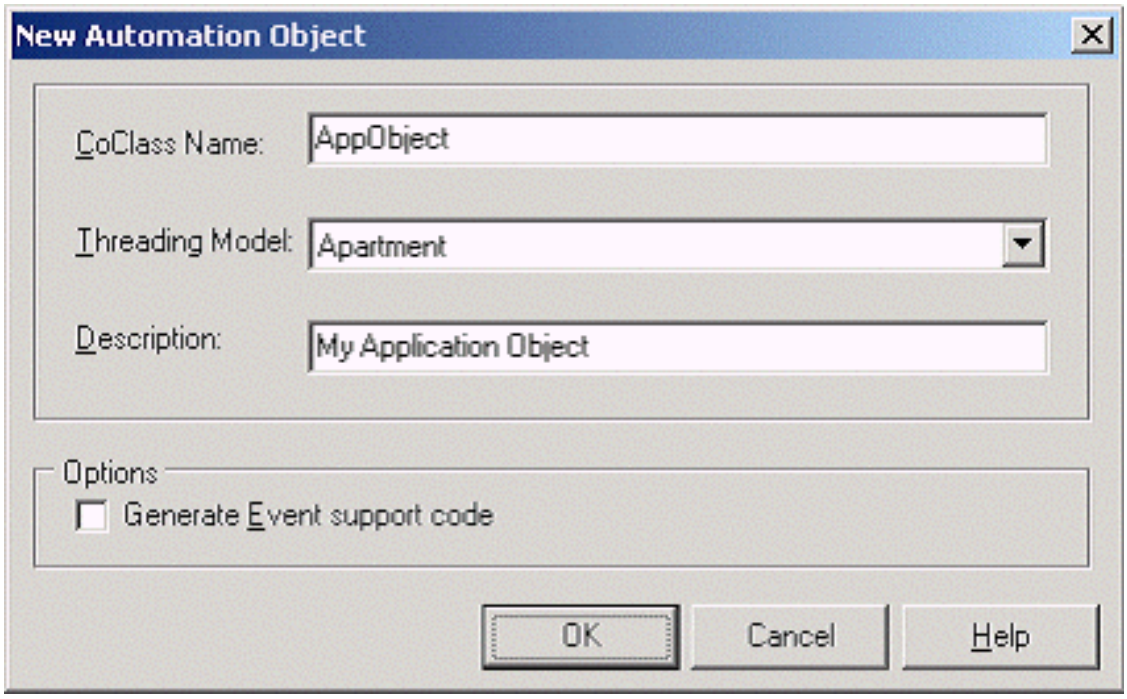
Now that we've brushed the surface of COM and interfaces, we can proceed to the next topic—designing interfaces in C++Builder. As I mentioned before, COM only allows its clients to access the server's interfaces, never the objects directly. This means that you don't have to worry too much about implementing your application objects to work with COM. As long as you create classes that have the properties and methods you want to expose, you can add the interfaces later. COM doesn't enforce or even specify how classes are implemented. For this reason, I usually create my application classes—the real code—outside of COM, and create COM interfaces to access them.

The Type Library Editor

COM interfaces can be added to a `.DLL` or `.EXE` project. For `.EXE` projects, COM interfaces can be added to any existing C++Builder application. However, `.DLL` projects must be initially created as an "ActiveX Library" in order to contain COM interfaces (the `.DLL` doesn't necessarily have to contain ActiveX objects, even though the project template is called "ActiveX Library").

When you're ready to add an interface, select File|New to add a new unit to your project, then select Automation Object from the ActiveX tab. The dialog box shown in **Figure A** will be displayed. The Automation Object template will create a COM class that implements the `IDispatch` interface.

Figure A



The New Automation Object dialog box is used to create a new COM object

The CoClass Name field is the name of the class that will be created. It will also be used to name the source file for the code, which the IDE will automatically add to your project. The Apartment threading model should be used unless you're sure that the component will be thread-safe. After clicking OK, the IDE will display the Type Library Editor.

If you've never developed COM components before, the Type Library Editor might seem a little complex at first. But you should feel lucky—in the early days of COM development, creating new components and interfaces was an extremely tedious chore, and one that nobody misses! I could spend a lot of time discussing how to use the Type Library Editor, but instead I'll refer you to the C++Builder documentation, which provides a good overview of all of its features.

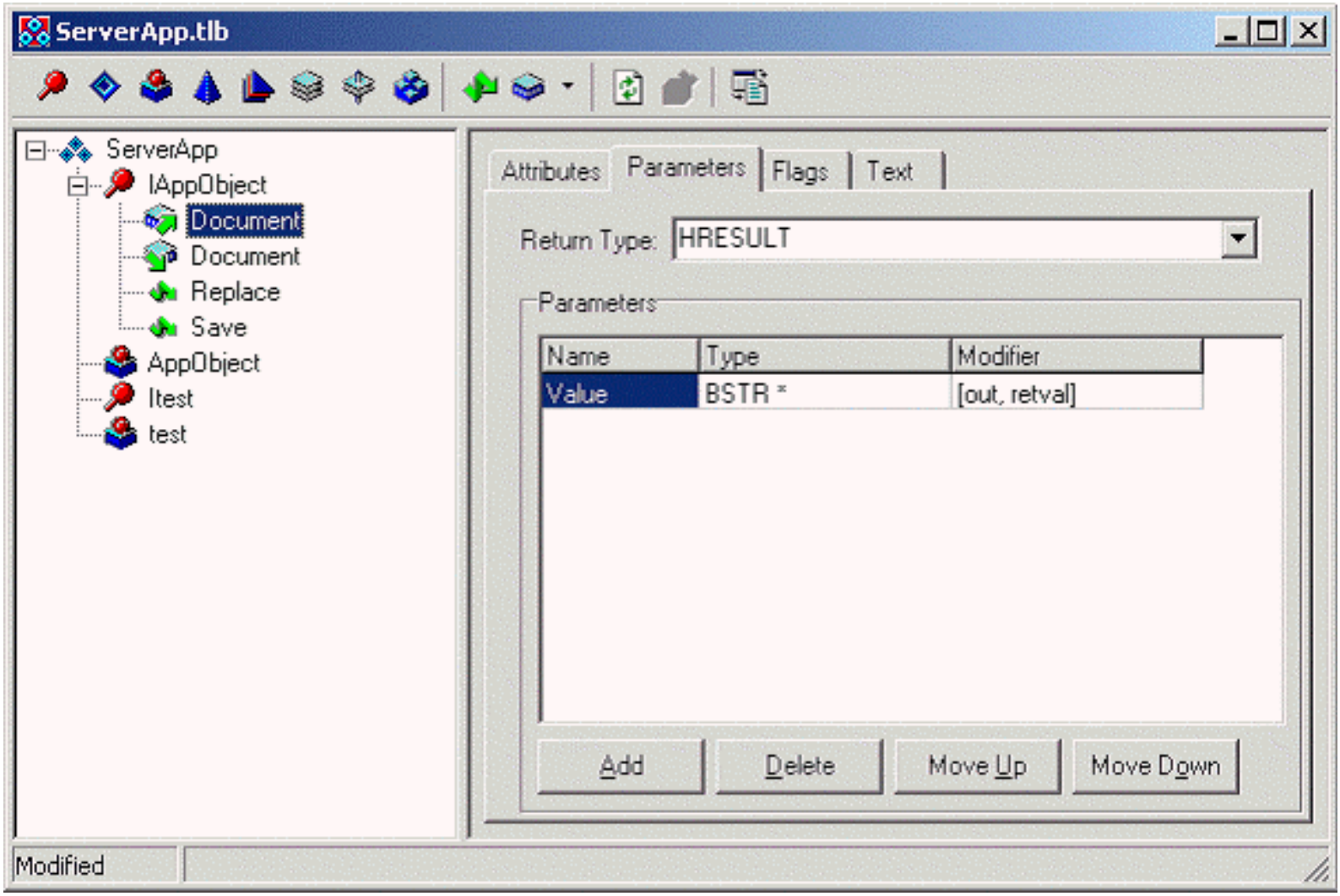
In the example shown, I've created a COM class called `AppObject` and an interface for the class called `IAppObject`. A COM class isn't much more than a container for interfaces—it's the interfaces that are really important. The `IAppObject` interface has one property, `Document`, and two methods, `Replace()` and `Save()`. The `AppObject` component is part of an application called `ServerApp.exe`, which is a simple text file editor. It has the ability to load a text file, search and replace text, and save the text to a file. The application has its own objects that do the real work, and the COM interfaces provide wrappers so that any COM client can control the program.

I'll briefly discuss how the interface properties and methods are defined in the Type Library Editor. Unlike C++, COM interface methods almost always return the same type, `HRESULT`. This return value is used by COM to indicate whether the method executed successfully. The "return value" that an automation

controller receives is actually specified as a parameter. In the example shown, the `Document` property is implemented as two methods—a "get" method and a "set" method. The "get" method has the [out] modifier added to its `Value` parameter to indicate that this parameter is to be used as the "return value" of the method. The `Replace()` method takes two parameters but has no return value, and the `Save()` method takes no parameters and has no return value. **Figure B** shows the `AppObject` object in the Type Library Editor.

In Visual Basic and VBScript, COM interface methods that have no return value are treated as subroutines, and interface methods that have a return value are treated as functions.

Figure B



The Type Library Editor is used to add properties and methods to a COM object

Once you've designed all of the properties and methods for the interface, including all of their parameters and return values, click Refresh Implementation (the button with the two green arrows). I've always thought that this should be done automatically upon closing the Type Library Editor, but it isn't. You should always click the refresh button before closing the editor.

After clicking the refresh button, you'll see a bunch of files added to the project (in this case, `ServerApp.bpr`). Here is a list of the different files that are added, and what they are for:

File	Purpose
`ServerApp.tlb`	This is the COM Type Library, a binary file that stores all of the details about the COM components.
`ServerApp_ATL.CPP` `ServerApp_ATL.H`	These files include the Microsoft ATL classes, which hide much of the complexity of developing COM components.
`AppObjectImpl.CPP` `AppObjectImpl.H`	This is the "implementation class" which will contain the application code or call other application objects.

The implementation class

Chances are that you'll not want to touch any of these files except for the implementation class source files. The implementation class is a C++ class that contains the "skeleton" of the interface. This class is written automatically by the IDE, and uses the ATL library to shield you from the drudgery of implementing all of the C++ code needed for creating COM interfaces. It has methods to access all properties and methods of the interface, but it doesn't contain any useful code yet. It's now up to you to write the code.

When adding code to the implementation class, be careful not to change the function names or parameter types of the pre-written methods. If you discover that you need to change a name or parameter type, close the `.cpp` and `.h` file, then invoke the Type Library Editor by double-clicking on the `.tlb` project file, and change it there. Then click the refresh button again, and the change will be made into the existing implementation file. If you do make changes to the function names or parameters without using the Type Library Editor, chances are it will lose its mind the next time you try to refresh.

Alas, you're not completely constrained to the Type Library Editor when developing your implementation classes. You're free to add additional functions and member variables, even a destructor (the IDE provides only a constructor by default). The Type Library Editor doesn't affect additional functions that you add—it will ignore them when it refreshes its changes to your implementation class. **Listing A** shows the implementation class for `IAppObject`. The code for the entire ServerApp project can be downloaded from **www.bridgespublishing.com**.

You can see from the implementation class code that it serves as a wrapper for the main VCL application objects. It's important to trap any exceptions that may occur in interface methods, because the automation

controller using the interface will not necessarily be able to handle them. In most cases, it's sufficient to pass the message text of the exception back to the client through the built-in `Error()` method. This will cause the automation controller to properly report the error condition. For example, VB Script will display a message box containing the exception message, the class and method that caused it, and the line number on which it occurred.

The BSTR data type

In the VCL environment, we have the luxury of using the versatile `AnsiString` class to pass strings back and forth between objects. Since COM is language (and theoretically, platform) independent, it must allocate and manage its own string objects. A `BSTR` is a pointer to a UNICODE character string that has a set length. This is the string datatype in COM. There is a host of API functions to manage these, such as `SysAllocString()`, `SysFreeString()`, `SysReallocString()`, and `SysStringLen()` (to name but a few). If you're interested, you can refer to the Windows API for descriptions of how these work. Luckily, the VCL class `WideString` provides a simple alternative to the API.

In COM, whenever a string is passed back from a function, it must be allocated by the server and then freed by the client. Luckily for you the component developer, you don't have to worry about what the client needs to do—just make sure that all return values you pass back as strings are newly allocated. The `WideString::Copy()` function does this for you. The easiest way to allocate a `BSTR` is to take a `char*` or `AnsiString` and convert it into a `WideString`, then call its `Copy()` function. This will allocate a new `BSTR` string that can be safely passed back to the client, which then has the responsibility of freeing it.

As an example, refer to the `AppObjectImpl::get_Document()` method. It takes a VCL `AnsiString` property and converts and copies it, then assigns it to the return value (the "out" parameter). The `AppObjectImpl::get_Document()` method takes a `WideString` input parameter and assigns it directly to a VCL `AnsiString` property. `AnsiString` and `WideString` can be assigned to each other, and will automatically convert.

ATL options

Projects that contain COM components have an additional tab on the project options dialog. Here you can specify the threading model of the server, and the instancing option. With *multiple use* instancing (the default), all clients share the same copy of the server; if multiple processes are accessing the server's components, only one copy of the server will load. With *single use* instancing, a separate instance of the server is created for each client. It's usually best to use multiple use instancing.

Registering the component

COM keeps track of the location of components via the registry. Components in an `.exe` file created with C++Builder will automatically register themselves the first time the program is run. After the program has been run at least once, its components will be accessible to any automation controller.

Using the component

Now we're finally ready to run `ServerApp.exe` from a client. The following example shows a VBScript file that uses ServerApp to load all of the files in a directory and perform a search and replace on all of them.

```
TestServerApp.vbs

‘ This script uses ServerApp.exe to
‘ process all files in a directory and
‘ append <BR> to each line.

Set fs = CreateObject(
  "Scripting.FileSystemObject")
Set Editor = CreateObject(
  "ServerApp.AppObject")
DocDir = "c:\articles\text\"
SearchStr = Chr(13) & Chr(10)
ReplaceStr = "<BR>" & SearchStr

Set FileDir = fs.GetFolder(DocDir)
For Each File in FileDir.Files
FileName = DocDir + File.Name
Editor.Document = FileName
Editor.Replace SearchStr, ReplaceStr
Editor.Save
Next
```

The `ServerApp.exe` program automatically loads when the `ServerApp.AppObject` object is created. Then the script controls it until the script is done with it. At that point, no more clients are referencing the server, so the program unloads.

Conclusion

Hopefully I've shed some light on the large topic of adding automation interfaces to your C++Builder programs. In Part 2 of this article, I'll explore some other aspects of COM development, including some of the utility functions and classes that are useful in developing more advanced interfaces.

Listing A: AppObjectImpl.h

```
// APPOBJECTIMPL.H : Declaration of TAppObjectImpl
#ifndef AppObjectImplH
#define AppObjectImplH

#include "ServerApp_TLB.H"

//////////////////////////////////////////////////
// TAppObjectImpl Implements IAppObject,
//   default interface of AppObject
// ThreadingModel : Apartment
// Dual Interface : TRUE
// Event Support  : FALSE
// Default ProgID : ServerApp.AppObject
// Description    : ServerApp Application Object
//////////////////////////////////////////////////
class ATL_NO_VTABLE TAppObjectImpl :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<
  TAppObjectImpl, &CLSID_AppObject>,
public IDispatchImpl<IAppObject, &IID_IAppObject,
  &LIBID_ServerApp>
{
public:
TAppObjectImpl()
{
}

// Data used when registering Object
DECLARE_THREADING_MODEL(otApartment);
DECLARE_PROGID("ServerApp.AppObject");
DECLARE_DESCRIPTION(
  "ServerApp Application Object");

// Function invoked to (un)register object
static HRESULT WINAPI
  UpdateRegistry(BOOL bRegister)
{
TTypedComServerRegistrarT<TAppObjectImpl>
regObj(GetObjectCLSID(),
  GetProgID(), GetDescription());
return regObj.UpdateRegistry(bRegister);
}
```

```
BEGIN_COM_MAP(TAppObjectImpl)
COM_INTERFACE_ENTRY(IAppObject)
COM_INTERFACE_ENTRY2(IDispatch, IAppObject)
END_COM_MAP()
// IAppObject
public:

STDMETHOD(get_Document(BSTR* Value));
STDMETHOD(set_Document(BSTR Value));
STDMETHOD(Replace(BSTR From, BSTR To));
STDMETHOD(Save());
};

#endif //AppObjectImplH
```

Listing A: AppObjectImpl.cpp

```
// APPOBJECTIMPL: Implementation of TappObjectImpl
// (CoClass: AppObject, Interface: IAppObject)
#include <vcl.h>
#pragma hdrstop

#include "APPOBJECTIMPL.H"
#include "ServerAppMain.h"

////////////////////////////////////////////////
// TAppObjectImpl
STDMETHODIMP TAppObjectImpl::get_Document(
  BSTR* Value)
{
WideString wsDoc =ServerAppMainForm->DocumentName;
*Value = wsDoc.Copy();
return S_OK;
};


STDMETHODIMP TAppObjectImpl::set_Document(
  BSTR Value)
{
try
  {
ServerAppMainForm->OpenTextFile(Value);
  }
```

```
catch(Exception &e)
  {
return Error(e.Message.c_str(), IID_IAppObject);
  }
return S_OK;
};


STDMETHODIMP TAppObjectImpl::Replace(
  BSTR From, BSTR To)
{
try
  {
ServerAppMainForm->ReplaceText(From, To);
  }
catch(Exception &e)
  {
return Error(e.Message.c_str(), IID_IAppObject);
  }
return S_OK;
}

STDMETHODIMP TAppObjectImpl::Save()
{
try
  {
ServerAppMainForm->SaveText();
  }
catch(Exception &e)
  {
return Error(e.Message.c_str(), IID_IAppObject);
  }
return S_OK;
}
```

Using Web Services

by Bob Swart

Web Services are hot, and just about everywhere. From Java to .NET, Web Services will be an important technology to reckon with. And C++Builder contains support for Web Services. With C++Builder 6 Professional we can import (consume) existing Web Services, and with C++Builder 6 Enterprise we can even make our own Web Services (the engines, as I'll explain in a future article).

This article will explain how to consume existing Web Services using C++Builder 6 (Professional or Enterprise). In this article, I will use a Web Service that I wrote last year using Delphi 6 Enterprise, but in theory we could use any Web Service. In the next article, I will be creating a new Web Service using C++Builder 6; a process which is only slightly more complex than using one.

Consuming Web Services

Delphi 6 and C++Builder 6 are only two of the Borland tools that can create and import Web Services. Others include Kylix 2 (running on Linux) and JBuilder 6. The upcoming C++ version of Kylix will most likely contain support for Web Services as well.

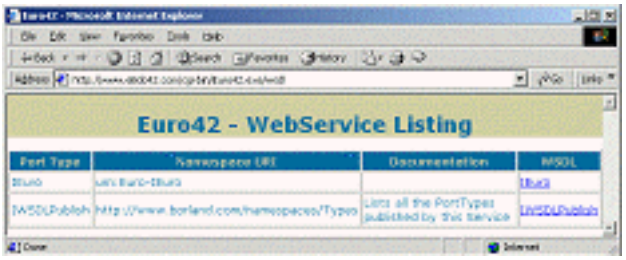
The main reason why I've decided to use C++Builder 6 to consume a Web Service written in Delphi 6, is to illustrate the cross-language nature of Web Services right from the start. In fact, with hardly any trouble you could consume Web Services written in Kylix on Linux or JBuilder (on Windows, Linux, Solaris or even the Mac).

IEuro

The example Web Service that we will be using in this article is the Euro Converter Web Service, also known by its interface name IEuro. This Web Service should be available on the internet as http://www.eBob42.com/cgi-bin/Euro42.exe/wsdl (you can also use the Linux version at http://www.drbob42.co.uk/cgi-bin/Euro42/wsdl).

Using a browser such as Internet Explorer, we can view the WSDL (Web Services Description Language) of our web service, which shows no less than two SOAP objects in our web service application: `IEuro` and `IWSDLPublish`. The latter is a "present" from Borland, and will be included in any Web Service that you write using Delphi 6, C++Builder 6 or Kylix 2. The main purpose of this SOAP Object is to publish the WSDL definition for the other SOAP Objects inside the Web Service application. This is illustrated in the documentation section of **Figure A**.

Figure A



The WebService Listing of Euro42.exe

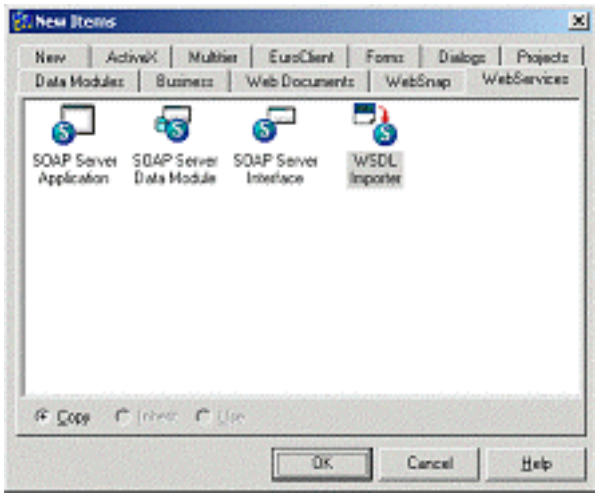
For the individual WSDL definition of the IEuro Web Service, we need to click on the IEuro link, or just specify http://www.eBob42.com/cgi-bin/Euro42.exe/wsdl/IEuro. Once you view the WSDL for `IEuro`, you should either save the URL (in the clipboard) or the WSDL contents itself (in a local file), because we'll need them in a moment when we create an import unit with the C++ definition mapping for the IEuro Web Service.

C++Builder 6 Enterprise

Start C++Builder 6 Enterprise, and save the new default project (or create a new one). Save the main form in file CLIENTFORM.CPP and the project itself in EUROCLIENT.BPR. Before we can use the IEuro web service, we first need to create a C++ import unit (based on the WSDL definition as seen in **Figure 1**). This can be done using the WSDL Importer.

Click File | New | Other, and go to the WebServices tab of the Object Repository. Here, you'll find four SOAP specific wizards as shown in **Figure B**. C++Builder 6 professional users will only find the WSDL Importer wizard, but can still work along with this article, because that's the only wizard we'll be using this time.

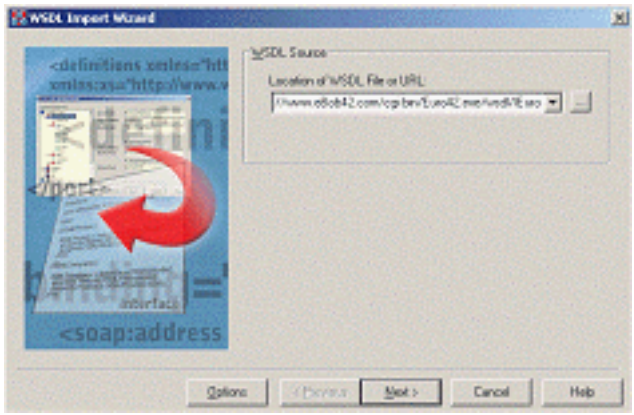
Figure B



The WebServices tab of the Object Repository

When you start the WSDL Importer, you'll get a wizard as shown in **Figure C**. The wizard starts by asking the location of the WSDL (the Web Service Description Language). Note that this can either be a local file that you saved earlier, or a live URL to the web service on the website (http://www.eBob42.com/cgi-bin/Euro42.exe/wsdl/IEuro in this case). If you do not want to be connected to the internet while developing the EuroClient application, use the local file containing the WSDL.

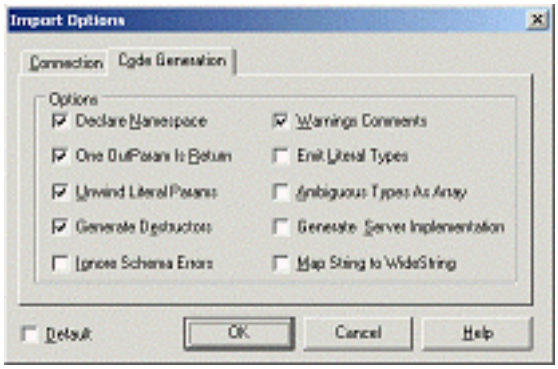
Figure C



The WSDL Import Wizard

The Options button is available on this page and the next one, and will enable you to specify a number of Connection (through a Proxy) and Code Generation options that you may want to set. I personally use the default settings, but you may want to be familiar with the various options. The Import Options dialog box is shown in **Figure D**.

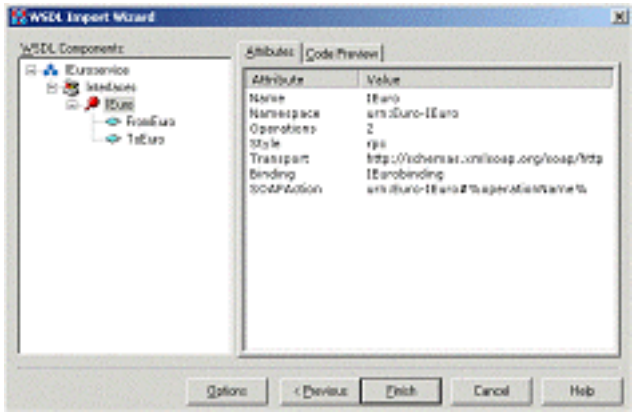
Figure D



The WSDL Import Options dialog box

When you close down the Import Options dialog box and click Next, you get to the second (and last) page of the WSDL Import Wizard. In this page, you are presented with an overview of the WSDL components, including the interfaces (in this case only `IEuro`) and the methods that are available. The last page of the wizard is shown in **Figure E**.

Figure E



The WSDL Import Wizard preview page

For closer examination of the generated source code, just click on the Finish button. This will generate the new import unit for the WSDL file or location specified. The header file IEURO_.H contains as most important part the definition of the `IEuro` interface, derived from `IInvokable`, and containing the two abstract virtual methods `FromEuro()` and `ToEuro()`.

The corresponding implementation inside file IEURO_.CPP contains a function called `GetIEuro()`. This function makes the usage of the generated `IEuro` interface inside the generated files very simple: we only need to call the `GetIEuro()` function (or in general any `Get` followed by the name of our interface) to get an interface back to the SOAP server that we want to use.

In our case, the call to `GetIEuro()` returns the `IEuro` interface—implemented by a component

behind the scenes—ready to be used. Since the `IEuro` interface contains only two methods, we can simply select one, for example the `FromEuro()` method, which takes an `AnsiString` and `double` as argument and returns a `double`.

GetIEuro()

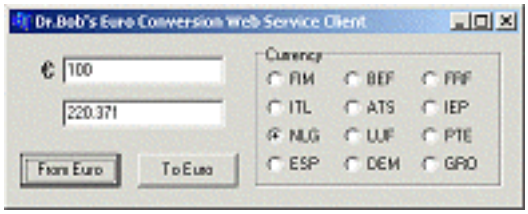
Now return to the `ClientForm` unit, and add the generated IEURO_.H header file to this unit (with File | Use Unit), so we can access the `GetIEuro()` to return the `IEuro` interface and its methods. Before doing that, let's first finish the Euro Client Form. For this, we need two `TEdit` components (call them `edtEuro` and `edtCurr`, and clear their `Text` property), two `TButtons` (call them `btnFromEuro` and `btnToEuro`), a `TRadioGroup` (call it `Currency` and set the `Columns` property to 3, and the `Items` property to FIM, ITL, NLG, ESP, BEF, ATS, LUF, DEM, FRF, IEP, PTE, and GRO), and finally a `TLabel` with the `Caption` property set to the Euro character € to finish it off (see **Figure G** to see how the client form looks).

Obviously, the `btnFromEuro` button will be used to convert the value inside `edtEuro` to a value for `edtCurr` for the selected currency inside the `RadioGroup`. For the `OnClick` event handler of the `btnFromEuro` button, we first need to convert the contents of the `edtEuro->Text` property from a string to a floating point value (a `double`) and then use this as argument to the `FromEuro()` method from the `IEuro` interface. Finally we convert the result back from a floating point value (a `double`) to an `AnsiString` so we can put it inside the `edtCurr->Text` property. The current item text of the `RadioGroup` can be used as an `AnsiString` argument to the `FromEuro()` method. In code, this is all done as follows:

```
void __fastcall TForm1::
  btnFromEuroClick(TObject *Sender)
{
 edtCurr->Text = FloatToStr(
  GetIEuro()->FromEuro(
   Currency->Items[
    Currency->ItemIndex],
     StrToFloatDef(edtEuro->Text,0)));
}
```

Using this event handler, we can convert 100 Euro to 220.371 Dutch Guilders, as you can see in **Figure F**.

Figure F



The Euro Web Service Client running

The HTTPRIO way

Calling the `GetIEuro()` function is really convenient to get a handle to the `IEuro` interface. But what exactly is happening behind the scenes? In order to explain that, we'll write the manual code inside the `OnClick` event handler of the `btnToEuro` button so you can compare the two methods, and also learn what the `GetIEuro()` does.

In order to retrieve the `IEuro` interface "manually", we need a `HTTPRIO` component (from the `WebServices` tab of the component palette). We need to set three properties of this component, starting with the `WSDLLocation` property, which should get the value "http://www.eBob42.com/cgi-bin/Euro42.exe/wsdl/Ieuro". After we've set the `WSDLLocation`, we then need to select a possible value for the `Service` property. If the `WSDLLocation` property contains a valid value (and if the web service can be accessed), then the combo box for the Service property value will be opened and show a single choice, namely `IEuroservice`. If you don't see anything in the combo box, then either the `WSDLLocation` contains a typo, or you cannot access the Web Service at this time. After you've set the `Service` property, you need to repeat the last step for the `Port` property, which should again only offer you a single choice, namely `IEuroPort`.

With these three properties set, the `HTTPRIO` component can act as a Remote Invokable Object over HTTP (hence its name), connecting your client form to the `IEuro` SOAP object inside the Euro42.exe Web Service application.

The `OnClick` event handler is now implemented as follows (note that we have to use `QueryInterface` to manually cast the `HTTPRIO` component to the interface of type `_di_IEuro`):

```
void __fastcall TForm1::
  btnToEuroClick(TObject *Sender)
{
  _di_IEuro service;
  HTTPRIO1->QueryInterface(service);
  if (service) {
    AnsiString S = Currency->Items->
      Strings[Currency->ItemIndex];
    edtEuro->Text = FloatToStr(
```

```
      service->ToEuro(S,StrToFloatDef(
        edtCurr->Text,0)));
  }
}
```

Apart from the fact that we now have to manually use a `HTTPRIO` component and extract the `_di_IEuro` interface from it, the remaining code is very similar to what we've written before. **Figure G** shows the Client form at design time again (this time with the `HTTPRIO` component on it as well).

Figure G

The Euro Web Service Client form at design time

HTTPRIO and URL

If you're interested, you can look inside the `GetIEuro()` function of file IEURO_.CPP, and notice that it does the same thing. Or does it? The argument `useWSDL` is set to `false` by default, meaning that it doesn't do the same thing, but rather uses the `URL` property of the `HTTPRIO` component (instead of the `WSDLLocation` property).

The `URL` property can be used if and only if both the Web Service interface definition and the generated import file are registered using the same namespace (this is done using the `InvRegistry()->RegisterInterface` call using the so-called Invokable Registry). As the on-line help explains, this can be done in a number of ways, and will be at least be the case automatically if the server was written in Delphi, Kylix or C++Builder itself. In order to explicitly use the `URL` property with the `HTTPRIO` component, we have to clear the `Port`, `Service` and `WSDLLocation` properties, and set the `URL` property to http://www.eBob42.com/cgi-bin/Euro42.exe/soap/IEuro (note that we're now using the path "soap" instead of "wsdl" as we did before so this isn't a straight copy of the URL that we used to import the WSDL definition). Only a single property is used now—compared to three in the more general usage—and the actual C++ code can remain untouched.

Personally, I always use the `WSDLLocation` property in combination with `Service` and `Port`, because this is a more general approach, that will also be able to use the Web Service when the server and client interface are not defined using the same namespace. But at least you now know the difference, and can make your own choices depending on your own preferences and needs.

Conclusion

In this article, I've tried to explain how to consume Web Services with C++Builder 6. We've seen how to generate the import files based on a WSDL definition of the Web Service, and how to actually call the methods from the Web Service (using the generated `GetInterface()` function, or using the `HTTPRIO` component). For real-world Web Services written in different environments, you'll find it useful to watch Borland's progress in the area of interoperability of Web Services at http://soap-server.borland.com. The full source for this article's example program can be downloaded from our Web site at **www.bridgespublishing.com**.

Next time, we'll continue the coverage of Web Services when we build one using C++Builder 6 Enterprise so stay tuned.

Bob Swart (aka Dr.Bob - **www.drbob42.com**) is an author, trainer and consultant who just started his own one-man company called "eBob42" in Helmond, The Netherlands. Bob, who writes his own "Delphi Clinic" training material, has spoken at Delphi and Borland Developer Conferences since 1993.

Bob is co-author of the Revolutionary Guide to Delphi 2, Delphi 4 Unleashed, C++Builder 4 Unleashed, C++Builder 5 Developer's Guide, Kylix Developer's Guide, Delphi 6 Developer's Guide and the upcoming C++Builder 6 Developer's Guide.

Creating a Web server using TServerSocket

By David Bridges

C++Builder comes with a variety of useful Internet components used for sending and receiving data over TCP/IP networks. They are the building blocks for n-tier application frameworks such as Borland's MIDAS, and can also be used to implement any other application requiring communication over TCP/IP (such as peer-to-peer applications). The `TServerSocket` component can be used to listen on a TCP/IP port, and respond to messages received on that port.

Why a Web server?

It's easy to use `TServerSocket` to create a Web server. Why write a Web server when there are many free Web servers out there for the platform of your choice? If the only function of your Web server were to serve up static Web pages, then it wouldn't make much sense. But having a Web server embedded in an application turns out to be a powerful thing. You can instantly add a Web interface to your application without having to write a middle tier to handle communication between the application and the Web server. In my case, I have an XML application server that runs on a machine locked away in a computer room. I needed the ability to remotely manage this server, so I dropped a `TServerSocket` into the application, added some functionality to make it use the HTTP protocol, and viola—my application server now has a Web interface!

HTTP basics

The HTTP protocol was invented back in 1990 by Tim Berners-Lee, a British researcher at the CERN particle physics lab in Switzerland. HTTP defines the format of messages sent over the Web. There are two types of HTTP messages—the request and the response. Both the request and response consist of two parts, the message header and the message body. For example, when you request the URL

```
http://www.bridgespublishing.com/default.htm
```

in your browser, it sends the following message to the server:

```
GET /default.htm HTTP/1.1
Accept: /
Accept-Language: en-us
User-Agent: Mozilla/4.0 (
```

```
  compatible; MSIE 5.5; Windows NT 5.0)
```

Most browsers also include additional information in the header. For more information on everything that can appear in a HTTP header, refer to www.w3.org.

For the purposes of this article, we're only concerned with the first line of the header message, the GET. The GET line has the following format:

```
GET {resource identifier} HTTP/{version}
```

The resource identifier is typically a file located on the Web server, such as "index.html". It may also be something that tells the server to generate a dynamic page, such as "mypage.asp?param=abc". The version tells the server what version of HTTP is being used.

If the server returns HTML that contains references to other files on the server, the client must send a separate HTTP GET for each one. For example, if an HTML page is returned that has three images on it, a Web browser will then send three HTTP GET messages to the server, one for each image.

A Web server's primary job is to take the resource identifier passed in the GET message, generate some HTML (or image data), and send the data back as a response. Here is the format of an HTTP response:

```
HTTP/1.0 {result code} {result message}
MIME-version: 1.0
{message body}
```

HTTP headers are always terminated with two line breaks. That's why there's an extra line before the message body. Some of the common result codes and messages are shown in **Table A**.

Table A: *Common HTTP Result Codes*

Result Code	Message	Description
200	OK	Normal, successful completion
400	BAD REQUEST	Bad HTTP request message
404	NOT FOUND	File/resource not found

The HTTP message shown below is a typical response from a Web server:

```
HTTP/1.0 200 OK
MIME-version: 1.0
Content-type: text/html

<HTML><HEAD><TITLE>Hello</TITLE></HEAD>
<BODY>Hello from the web server</BODY>
```

The content-type identifier tells the client (the Web browser) what kind of data is in the message body. The three most common types are "text", "image" and "application". The content type is followed by an additional identifier that describes the file type. For example, "text/html" indicates that the message contains text that should be interpreted as HTML. The content type "image/jpeg" means that the message will contain binary data representing a jpeg image. The "application" type specifies that the message body contains data formatted for an external application, such as .pdf or .doc files.

The TWebServer class

Now we know everything we need to about HTTP to implement a simple Web server. Luckily, all of the hard work involving sending data over TCP/IP sockets is handled by `TServerSocket`. We only need to add the code to tell it how to interpret GET messages, and send back the proper results.

There are two properties of `TWebServer` that are used to configure its operation. `WebDir` specifies the Web server's "root" directory. This is the top-level directory where it will look for files. `Port` specifies the TCP/IP port that the Web server will communicate on. The standard port for HTTP is 80, although you can use a different port number. If you use a port other than 80, then the port number must be specified in the browser URL, such as http://www.server.com:1234 for port 1234.

Using `TServerSocket` is as simple as setting its port number and then setting the `Active` property to `true`. At that point, your computer is a server. The `OnClientRead` event gets fired whenever a client sends data to the TCP/IP port on the server. The event names can be slightly confusing—`OnClientRead` happens when a client sends data to the server, and `OnClientWrite` happens when the server writes data to the client. In this application, only the `OnClientRead` event is used.

The `TWebServer` class responds to the `OnClientRead` event with its `ProcessRequest()` method. First, it parses the message header to see if it is a HTTP GET message. If it is, it tries to find the file specified by the resource identifier. If the file is found, it responds back with the file's contents. If the file doesn't exist, it responds back with a "404 NOT FOUND" message. Finally, if the message is not a HTTP GET message, it responds back with a "400 BAD REQUEST" message.

Note that the `GenerateHTML()` method is a virtual function, so it can be implemented differently in classes derived from `TWebServer`. This way, additional functionality can be implemented, such as the ability to parse parameters out of request strings.

Running the Web server

Listing A shows how to create an instance of `TWebServer`. To test it out, start the server and then point your browser to http://localhost. If you use a port other than 80, you must use http://localhost:port.

Extending TWebServer

By itself, the `TWebServer` class doesn't provide much utility, except as an extremely lightweight Web server. But classes derived from `TWebServer` can implement their own version of `GenerateHTML()` to dynamically create Web pages by using a file as a template, parsing parameters from the request string, and so on.

Now you've seen how `TServerSocket` can be used to add a Web interface to any application just by adding some code to make it use the HTTP protocol. The code for `TWebServer` and `TWebServerApp` is shown in **Listings B** and **C** and is also available at **www.bridgespublishing.com**.

Listing A: ***WebServerMain.cpp***

```
#include <vcl.h>
#pragma hdrstop

#include "WebServerMain.h"
#include "WebServer.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TWebServerMainForm *WebServerMainForm;

__fastcall TWebServerMainForm::TWebServerMainForm(
  TComponent* Owner) : TForm(Owner)
{
  WebServer = new TWebServer(this);
}

void __fastcall TWebServerMainForm::btnStartClick(
  TObject *Sender)
{
```

```
  if (btnStart->Caption == "&Start") {
    WebServer->WebDir = edtWebdir->Text;
    WebServer->Port = edtPort->Text.ToInt();
    WebServer->Start();
    btnStart->Caption = "&Stop";
    edtStatus->Text = "Running";
  }
  else {
    WebServer->Stop();
    btnStart->Caption = "&Start";
    edtStatus->Text = "Stopped";
  }
}
```

Listing B: ***WebServer.h***

```
#ifndef WebServerH
#define WebServerH

#include <ScktComp.hpp>

class TWebServer : public TComponent
{
  protected:

  TServerSocket* ServerSocket;

  AnsiString GetContentType(AnsiString sFile);

  virtual bool GenerateHTML(
    AnsiString sResource, TMemoryStream* Output);

  void __fastcall ProcessRequest(
    TObject *Sender, TCustomWinSocket *Socket);

  public:

  __fastcall TWebServer(TComponent* AOwner);

  AnsiString     WebDir;
  long           Port;

  bool Start();
  void Stop();
};
```

```
#endif
```

Listing C: *WebServer.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "WebServer.h"

#pragma package(smart_init)


// This is the set of known graphic file types
// for the web server.
bool IsImageType(AnsiString sFileExt)
{
  return
    (!sFileExt.AnsiCompareIC("jpg")   ||
     !sFileExt.AnsiCompareIC("jpeg")  ||
     !sFileExt.AnsiCompareIC("gif")   ||
     !sFileExt.AnsiCompareIC("bmp")   ||
     !sFileExt.AnsiCompareIC("png"));
}

__fastcall TWebServer::TWebServer(
  TComponent* AOwner) : TComponent(AOwner)
{
  ServerSocket = new TServerSocket(this);
  ServerSocket->Active = false;
  ServerSocket->Name = "ServerSocket";
  ServerSocket->ServerType = stNonBlocking;
  ServerSocket->OnClientRead = ProcessRequest;
  Port = 80;
}

bool TWebServer::Start()
{
  if (ServerSocket->Active) {
    return false;
  }
  try {
    // Assign TCP/IP Port to use
    ServerSocket->Port = Port;
```

```
    // Start the server
    ServerSocket->Active = true;
  }
  catch (Exception& e) {
    MessageBox(NULL, e.Message.c_str(),
      "Error Starting Server",MB_ICONEXCLAMATION);
    ServerSocket->Active = false;
  }
  return ServerSocket->Active;
}

void TWebServer::Stop()
{
  // Stop the server
  ServerSocket->Active = false;
}

AnsiString TWebServer::GetContentType(
  AnsiString sFileExt)
{
  // Default to text content type
  AnsiString sType = String(
    "Content-type: text/") + sFileExt;

  if (IsImageType(sFileExt)) {
    // File is an image
    sType =
      String("Content-type: image/") + sFileExt;
  }

  if (!sFileExt.AnsiCompareIC("zip")   ||
      !sFileExt.AnsiCompareIC("doc")  ||
      !sFileExt.AnsiCompareIC("pdf"))
  {
    // File is an application document
    sType = String("Content-type: application/") +
      sFileExt;
  }

  return sType;
}

// This function contains the core functionality
// of the web server. It takes a request and
```

```
// generates output to a stream.
bool TWebServer::GenerateHTML(
  AnsiString sResource, TMemoryStream* Output)
{
  bool bResult = false;
  if (FileExists(sResource)) {
    // Send the specified file to output stream
    Output->LoadFromFile(sResource);
    bResult = true;
  }
  return bResult;
}

void __fastcall TWebServer::ProcessRequest(
  TObject *Sender, TCustomWinSocket *Socket)
{
  // Read incoming data from the client.
  AnsiString sInput = Socket->ReceiveText();

  // Put the request into a TStringList, to
  // separate the request header lines.
  TStringList* Request = new TStringList();
  Request->CommaText = sInput;

  // Create another TStringList to hold response
  TStringList* Response = new TStringList();

  // Make sure the request is a HTTP GET. The
  // server doesn't support other request types.
  if (Request->Strings[0] == "GET") {

    // The second item in the request is the
    // file/resource to retrieve.
    AnsiString sResource = Request->Strings[1];

    // Build complete filename for requested file.
    AnsiString sRequestedFile = WebDir+sResource;

    // If the request ends in a slash, append
    // the default filename.
    char cEnd =
      sRequestedFile[sRequestedFile.Length()];
    if (cEnd == '\\' || cEnd == '/') {
      sRequestedFile += "index.html";
```

```
}

// Get file extension and omit the '.'
AnsiString sFileExt = ExtractFileExt(
  sRequestedFile).SubString(2,10);

// Get content type description
AnsiString sContentType =
  GetContentType(sFileExt);

// Create an output stream for HTML
TMemoryStream* HTMLData = new TMemoryStream();
AnsiString sHTTPStatus;

// Generate output.
if (GenerateHTML(sRequestedFile, HTMLData)) {
  sHTTPStatus = "HTTP/1.0 200 OK";
}
else {
  sHTTPStatus = "HTTP/1.0 404 NOT FOUND";
}

// Create response header
Response->Add(sHTTPStatus);
Response->Add("MIME-version: 1.0");
Response->Add(sContentType);
Response->Add("");

// Send response header
Socket->SendText(Response->Text);

// Send HTML
if (HTMLData->Size > 0) {
  Socket->SendBuf(
    HTMLData->Memory, HTMLData->Size);
}
else {
  // If the requested file is not an image,
  // return an HTML page indicating bad
  // filename. Otherwise, just return header.
  if (!IsImageType(sFileExt)) {
    Response->Add("<HTML><HEAD><TITLE>File "
      "Not Found</TITLE></HEAD>");
    Response->Add(
```

```
          "<BODY><H1>File Not Found</H1>");
        Response->Add(
          "The document you requested: ") ;
        Response->Add(Request->Strings[1]);
        Response->Add(
          "<BR>was not found.</BODY></HTML>");
        Socket->SendText(Response->Text);
      }
    }
    delete HTMLData;
  }
  else {
    // Invalid request type.
    Response->Add("HTTP/1.0 400 BAD REQUEST");
    Response->Add("MIME-version: 1.0");
    Response->Add("Content-type: text/html\n\n");
    Response->Add("<HTML><HEAD><TITLE>Invalid "
      "Request Type</TITLE></HEAD>\n");
    Response->Add(
      "<BODY><H1>Invalid Request Type.</H1>\n");
    Response->Add("Request:<BR><PRE>");
    Response->Add(sInput);
    Response->Add("</PRE></BODY></HTML>");
    Socket->SendText(Response->Text);
  }

  delete Request;
  delete Response;

  // Done sending
  Socket->Close();
}
```

Display-optimal DIB section bitmaps

by Damon Chandler

In the article "Fast bitmap zooming and scrolling," I discussed the advantages of drawing only the required portions of a bitmap. That approach is half of the equation necessary to drawing bitmaps as fast as possible. In this article, I'll discuss the other half.

Prevent format conversions

Recall that the `BitBlt()` GDI function is used to render a bitmap to a display device. Yet, contrary to its name, `BitBlt()` does not always perform a simple Bit-Block transfer of memory. If the color format of the display device is not the same as the color format of the bitmap, the `BitBlt()` function (and its close cousin `StretchBlt()`) will have to perform a potentially costly color format conversion.

Suppose, for example, that you use `BitBlt()` to display a 24 bits-per-pixel bitmap on a system whose display also uses 24 bits per pixel. In this case, because the color format of the bitmap is the same as that of the display device, the `BitBlt()` function will effectively perform a bit-for-bit copy from the memory associated with the bitmap to the memory associated with the screen. On the other hand, if the display device uses, say, 15 bits-per-pixel, `BitBlt()` will have to convert from 24 bpp to 15 bpp by discarding the three least-significant bits from each pixel. Depending on the size of the bitmap, this conversion can severely reduce the performance of the `BitBlt()` function.

Of course, there's a tradeoff between the cost of the conversion and the cost of copying so many bytes. In some cases, it's faster to convert a smaller block of memory than it is to just copy a larger block. In general, however, if you want to display a bitmap as fast as possible, you need to make sure a format conversion isn't required.

Creating a device-optimal bitmap

As I mentioned in the previous articles on printing bitmaps, there are three main types of bitmaps: device-dependent bitmaps (DDBs), device-independent bitmaps (DIBs) and DIB section bitmaps. Because the device itself maintains device-dependent bitmaps, this variety of bitmap should theoretically be the fastest to render. Unfortunately, device-dependent bitmaps are severely limited in size, and they don't provide direct access to their underlying pixels.

On the other hand, Windows maintains DIB section bitmaps. Because of this fact, not only can you access the pixels directly, you can also specify the color format of the DIB section bitmap. However, in order to display a DIB section bitmap as fast as possible, you must make sure that this format matches

that of the display device. Here's a function to do that:

```
unsigned char MakeBitmapOptimal(
   Graphics::TBitmap& Bitmap)
{
  // will hold a handle to the display-
  // optimal DIB section bitmap
  HBITMAP hDIBSection = NULL;
  // will hold the optimal color depth
  unsigned char opt_bpp = 0;

  // grab the dimensions of Bitmap and a
  // handle to its associated memory DC
  const SIZE SBmp =
    {Bitmap.Width, Bitmap.Height};
  const HDC hBmpDC =
    Bitmap.Canvas->Handle;

  // allocate memory for a BITMAPINFO
  // (this structure is a DIB w/o its
  // pixels; this structure will be
  // initialized with the optimal format)
  const std::size_t dib_size =
    sizeof(BITMAPINFOHEADER) +
    256 * sizeof(RGBQUAD);
  BITMAPINFO* pDIB =
    reinterpret_cast<BITMAPINFO*>(
      new unsigned char[dib_size]);

  // zero-out all fields and then
  // specify the header size
  memset(pDIB, 0, dib_size);
  pDIB->bmiHeader.biSize =
    sizeof(BITMAPINFOHEADER);

  // grab a handle to the screen's DC
  const HDC hScnDC = GetDC(NULL);

  // create a 1x1 DDB which we'll use to
  // determine the format of the device's
  // internal memory surface
  const HBITMAP hDDB =
    CreateCompatibleBitmap(hScnDC, 1, 1);
```

```
// use the DDB and the GetDIBits()
// function with a NULL lpvBits
// parameter to query the format of
// the device-managed surface
bool got_ok = GetDIBits(hScnDC, hDDB,
  0, 1, NULL, pDIB, DIB_RGB_COLORS);
if (got_ok)
{
  // get the optimal bit-field info
  // (for 15/16- or 32-bpp bitmaps)
  got_ok = GetDIBits(hScnDC, hDDB, 0,
    1, NULL, pDIB, DIB_RGB_COLORS);
}
// delete the DDB
DeleteObject(hDDB);

if (got_ok)
{
  // store the optimal color depth
  opt_bpp = pDIB->bmiHeader.biBitCount;

  // set the proper dimensions; create
  // a top-down DIB section by setting
  // the biHeight data member negative
  pDIB->bmiHeader.biWidth = SBmp.cx;
  pDIB->bmiHeader.biHeight = -SBmp.cy;

  // create the optimal DIB section
  // bitmap based on the information
  // in pDIB (which the GetDIBits()
  // calls should have filled in)
  hDIBSection =
    CreateDIBSection(hScnDC, pDIB,
      DIB_RGB_COLORS, NULL, NULL, 0);
}
// release the screen's DC
ReleaseDC(NULL, hScnDC);

// free the BITMAPINFO
delete [] reinterpret_cast
  <unsigned char*>(pDIB);

// copy (and convert) the pixels of
// Bitmap to the optimal format DIB
```

```
// section bitmap...
if (hDIBSection != NULL)
{
  HDC hMemDC = NULL;
  HBITMAP hOldBmp = NULL;
  bool blt_ok = false;
  try {
    // associate DIB section bitmap
    // with a memory DC (for drawing)
    hMemDC = CreateCompatibleDC(NULL);
    hOldBmp = static_cast<HBITMAP>(
      SelectObject(hMemDC,hDIBSection)
      );

    // sync. the color tables
    if (opt_bpp <= 8)
    {
      // get Bitmap's color table
      RGBQUAD rgbQ[256];
      UINT num_entries =
        GetDIBColorTable(
          hBmpDC, 0, 256, rgbQ);

      // if Bitmap has no color table,
      // let's use its Palette instead
      if (num_entries == 0)
      {
        // grab a handle to Bitmap's
        // logical palette
        const HPALETTE hPal =
          Bitmap.Palette;

        // get the number of Bitmap's
        // palette entries
        num_entries =
          GetPaletteEntries(
            hPal, 0, 0, NULL);
        // create a LOGPALETTE buffer
        const std::size_t pal_size =
          sizeof(LOGPALETTE) +
          (num_entries - 1) *
          sizeof(PALETTEENTRY);
        LOGPALETTE* pLogPal =
          reinterpret_cast
```

```
        <LOGPALETTE*>(
        new unsigned char[pal_size]
        );
    // copy the palette data
    const UINT num_got =
      GetPaletteEntries(
        hPal, 0, num_entries,
        pLogPal->palPalEntry);
    for (UINT i=0; i<num_got; ++i)
    {
      rgbQ[i].rgbRed = pLogPal->
        palPalEntry[i].peRed;
      rgbQ[i].rgbGreen = pLogPal->
        palPalEntry[i].peGreen;
      rgbQ[i].rgbBlue = pLogPal->
        palPalEntry[i].peBlue;
    }

    // free the LOGPALETTE buffer
    delete [] reinterpret_cast
      <unsigned char*>(pLogPal);
  }
  if (num_entries > 0)
  {
    // copy the colors in rgbQ to
    // hDIBSection's color table
    SetDIBColorTable(hMemDC, 0,
      num_entries, rgbQ);
  }
}

// copy and convert the pixels of
// Bitmap to DIB section bitmap
blt_ok = BitBlt(
  hMemDC, 0, 0, SBmp.cx, SBmp.cy,
  Bitmap.Canvas->Handle, 0, 0,
  SRCCOPY);

// assign the optimal DIB section
// bitmap to Bitmap (the TBitmap
// object will take ownership)
if (blt_ok)
{
  Bitmap.Handle = hDIBSection;
```

```
      }
    }
    __finally {
      // clean up
      SelectObject(hMemDC, hOldBmp);
      DeleteDC(hMemDC);
      if (!blt_ok)
      {
        DeleteObject(hDIBSection);
      }
    }
  }
  // return the optimal color depth
  return opt_bpp;
}
```

This function is fairly well commented, so I won't discuss its specific steps. The basic idea, however, is to use the `GetDIBits()` function with a display DDB to query the format of the display device. The `GetDIBits()` function will copy this information into a `BITMAPINFO` structure, which is then used with the `CreateDIBSection()` function to create the display-optimal DIB section bitmap.

Conclusion

There are a few things you should keep in mind when using the `MakeBitmapOptimal()` function. First, if the display device uses fewer bits per pixel than your bitmap does, there will be a loss of color information. There's just no way to avoid this. The second point is a bit obvious, but I need to mention it anyway: If the `MakeBitmapOptimal()` function changes the color format of your bitmap, and you later need to access and work with the bitmap's pixels, you'll need to modify your code to work with the new format.

You can download the `MakeBitmapOptimal()` function along with a sample project that demonstrates its use from **www.bridgespublishing.com**.

Fast bitmap zooming and scrolling

by Damon Chandler

The VCL makes displaying a bitmap trivial—you simply drop a `TImage` control on your form and then load it with the desired bitmap. If you need to zoom in on the bitmap, set the `TImage::Stretch` property to `true` and then resize the `TImage` accordingly. If the bitmap is too large to display all at once, place the `TImage` within a `TScrollBox`. Unfortunately, this combination of controls (a `TImage` and a `TScrollBox`) is one of the least optimal ways to display an image with zooming and scrolling capabilities.

In this article, I'll present an efficient technique for displaying a bitmap in a zoomed fashion, and I'll show you how to scroll the bitmap.

A quick review of the WM_PAINT message

As you all know, you can't just draw something to your form and expect it to stay there. Rather, because the screen is a shared surface, you have to redraw your form's contents whenever needed (e.g., when your form is minimized and then restored). Windows sends a special message, called `WM_PAINT`, to inform a window that some part of its client area needs to be drawn. Here's how a typical `WM_PAINT` message is handled:

```
void __fastcall
  TForm1::WndProc(TMessage& Msg)
{
  if (Msg.Msg == WM_PAINT)
  {
    PAINTSTRUCT ps;
    const HWND hwd = WindowHandle;
    const HDC hDC = BeginPaint(hwd,&ps);
    // draw to hDC...
    EndPaint(hwd, &ps);
  }
  else TForm::WndProc(Msg);
}
```

The first step is to use the `BeginPaint()` function to retrieve a handle to the form's device context (DC). Once you have a handle to your form's DC, you can direct your drawing code to this DC. Finally, you inform Windows that you're done drawing—and that you no longer need the DC—by using the `EndPaint()` function.

Notice that the `BeginPaint()` function takes two parameters: a handle to the window of the DC in which you're interested, and a pointer to a `PAINTSTRUCT` structure. The `BeginPaint()` function will fill the `PAINTSTRUCT::rcPaint` data member (of type `RECT`) with the coordinates of the tightest bounding rectangle that defines the area that needs to be drawn.

Although, most of us would never handle the `WM_PAINT` message directly (we'd simply use the form's `OnPaint` event) the point I want to make is that the `PAINTSTRUCT::rcPaint` data member is crucial to efficient drawing. Namely, because `rcPaint` defines the area that needs to be redrawn, there's no need to call code that draws outside of this area. The `TImage` class is a primary example of VCL drawing code that fails to exploit the `rcPaint` data member.

The downside of using TImage

The `TImage` class internally maintains a `TPicture` object that can contain standard pictographic objects such bitmaps and metafiles. When the `TPicture` is loaded with a bitmap, the bitmap is drawn to the screen by using the `TBitmap::Draw()` method. In turn, the `Draw()` method uses the `StretchBlt()` GDI function to render the bitmap to the window on which the `TImage` is placed. Here's a condensed C++ translation of the `TBitmap::Draw()` method:

```
void __fastcall TBitmap::Draw(
  TCanvas* DstCanvas,const TRect& ARect)
{
  // initialization and palette stuff...
  // render the bitmap
  StretchBlt(
    DstCanvas->Handle,
    ARect.Left, ARect.Top,
    ARect.Width(), ARect.Height(),
    BitmapCanvas->Handle,
    0, 0, BitmapWidth, BitmapHeight,
    DstCanvas->CopyMode);
}
```

As you can see from this code, the `StretchBlt()` function takes 11 parameters. The first parameter is a handle to the DC to which the bitmap will be rendered. The next four parameters define the "destination rectangle"—the rectangle to which the bitmap will be scaled to fit. The sixth parameter is a handle to the (memory) DC with which the bitmap is associated. The seventh through tenth parameters define the "source rectangle"—the portion of the bitmap that's to be drawn. The last parameter specifies how the bitmap's colors should be combined with those of the target DC.

The `TImage` class calls (directly or indirectly) the `TBitmap::Draw()` method with an `ARect`

parameter (i.e., the destination rectangle) that's set to either the *full* dimensions of the `TImage` (if its `Stretch` property is `true`) or the *full* dimensions of the `TImage`'s bitmap (if the `Stretch` property is `false`). Further, notice that the `TBitmap::Draw()` method passes the *full* dimensions of the bitmap as the source rectangle. In light of what I discussed about the `PAINTSTRUCT::rcPaint` data member, you can probably see the problem with this approach. Regardless of what area needs to be redrawn, the `TImage` class will always draw the *entire* bitmap. This is clearly inefficient.

Drawing only what's needed

Let's work through an example of using the `PAINTSTRUCT::rcPaint` data member to draw only the required portion of a bitmap. Specifically, we'll draw a bitmap to the client area of a form. The declaration of the `TForm1` class for this example is provided in **Listing A**.

The `TBitmap` object for this example (`Bitmap_`) is created and loaded in the form's constructor, like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner),
    Bitmap_(new Graphics::TBitmap())
{
 Bitmap_->LoadFromFile("my_bitmap.bmp");
}
```

We'll render this bitmap to the client area of our form in response to the `WM_PAINT` message. You can see from **Listing A** that the `WM_PAINT` message is mapped to the `WMPaint()` method. Here's how the `WMPaint()` method is defined:

```
void TForm1::WMPaint(TMessage& Msg)
{
  static PAINTSTRUCT ps;
  const HWND hWnd = WindowHandle;
  hDstDC_ = BeginPaint(hWnd, &ps);
  try
  {
    PaintBitmap(ps.rcPaint);
  }
  __finally
  {
    EndPaint(hWnd, &ps);
  }
}
```

I've stored the handle to our form's device context—that is, the return value of the `BeginPaint()` function—in the private `hDstDC_` member. This member defines the "destination" device context. We'll draw to this DC from within the `PaintBitmap()` method, which is called with its single `RECT`-type parameter set to the `PAINTSTRUCT::rcPaint` data member.

Drawing an un-scaled bitmap

Let's first consider the case in which the bitmap should be drawn in its original size. In this case, the `PaintBitmap()` method can be defined as follows:

```
void TForm1::PaintBitmap(RECT& RUpdate)
{
  // destination rectangle
  const RECT RDst = RUpdate;

  // source rectangle
  const RECT RSrc = RUpdate;

  StretchBlt(
    // destination DC
    hDstDC_,
    // destination coordinates
    RDst.left, RDst.top,
    RDst.right - RDst.left,
    RDst.bottom - RDst.top,
    // source DC
    Bitmap_->Canvas->Handle,
    // source coordinates
    RSrc.left, RSrc.top,
    RSrc.right - RSrc.left,
    RSrc.bottom - RSrc.top,
    // ROP3 code
    SRCCOPY);
}
```

Here, we simply pass the coordinates of the update rectangle to the `StretchBlt()` function. Because there's no stretching involved, we can specify the same destination and source rectangles. In contrast to `TBitmap::Draw()` method however, here we draw only the portion of the bitmap that corresponds to the portion of the window that requires updating. Note that because no stretching is involved here, we could have used the `BitBlt()` function instead of `StretchBlt()`.

Drawing a zoomed bitmap

Now suppose that we want to zoom in on the bitmap by a factor of two. In this case, in order for `StretchBlt()` to perform the scaling, the width of the destination rectangle needs to be two times the width of the source rectangle and the height of the destination rectangle needs to be two times the height of the source rectangle.

We can achieve this 2:1 ratio in dimensions by scaling the source rectangle by a factor of one-half, both horizontally and vertically. Here's an example:

```
void TForm1::PaintBitmap(RECT& RUpdate)
{
  // scaling factor
  const float zoom = 2.0;

  // destination rectangle
  const RECT RDst = RUpdate;

  // source rectangle
  const RECT RSrc = {
    0.5 + RUpdate.left / zoom,
    0.5 + RUpdate.top / zoom,
    0.5 + RUpdate.right / zoom,
    0.5 + RUpdate.bottom / zoom };

  StretchBlt(
    // destination DC
    hDstDC_,
    // destination coordinates
    RDst.left, RDst.top,
    RDst.right - RDst.left,
    RDst.bottom - RDst.top,
    // source DC
    Bitmap_->Canvas->Handle,
    // source coordinates
    RSrc.left, RSrc.top,
    RSrc.right - RSrc.left,
    RSrc.bottom - RSrc.top,
    // ROP3 code
    SRCCOPY);
}
```

Unfortunately, there's a problem with this approach; namely, this code will work only if the coordinates of the update rectangle are all integer multiples of the zooming factor. Otherwise, we'll get visible rounding artifacts.

So, before we compute the coordinates of the source rectangle, we need to adjust `RUpdate` so that its `left`, `top`, `right`, and `bottom` data members are all integer multiples of `zoom`. If the zooming factor is an integer—such as two, 10, or, say, 16—then we can simply use the `fmod()` function (or the `%` operator) to compute the amount by which to adjust the update rectangle. Here's an example function that does that adjustment:

```
void AdjustUpdateRect(
  RECT& RUpdate, float zoom)
{
  const int dleft =
    std::fmod(RUpdate.left, zoom);
  const int dtop =
    std::fmod(RUpdate.top, zoom);
  const int dright =
    std::fmod(RUpdate.right, zoom);
  const int dbottom =
    std::fmod(RUpdate.bottom, zoom);

  RUpdate.left -= dleft;
  RUpdate.top -= dtop;
  RUpdate.right += (dright == 0) ?
    0 : zoom - dright;
  RUpdate.bottom += (dbottom == 0) ?
    0 : zoom - dbottom;
}
```

Unfortunately, this approach won't work when the zooming factor is a non-integer, such as, say, 2.56. Specifically, when the zooming factor is a non-integer, the `fmod()` function might also return a non-integer. In short, we'll end up with the same rounding artifacts.

If we assume however, that the zooming factor will contain at most two non-zero digits after the decimal point (e.g., 2.5, 2.56, but not 2.567), we can use the following approach:

```
#include <cmath>
void IncCoord(
   long& val, float zoom, int N)
{
  if (N == 0) // integer zooming factor
  {
```

```
    const int dval =
      0.5 + std::fmod(val, zoom);
    val += (dval == 0) ? 0 : zoom - dval;
    return;
  }

  int i = 0.5 +
    static_cast<int>(val / zoom);
  i = 0.5 + N * static_cast<int>(
    static_cast<float>(i + N) / N);
  val = 0.5 + (i * zoom);
}

void DecCoord(
   long& val, float zoom, int N)
{
  if (val == 0) return;

  if (N == 0) // integer zooming factor
  {
    val -= std::fmod(val, zoom);
    return;
  }

  int i = 0.5 +
    static_cast<int>(val / zoom);
  const float num =
    static_cast<float>(i - N);
  if (num < 0)
  {
    i = -0.5 +
      std::ceil(-0.001 + num / N) * N;
    val = -0.5 + (i * zoom);
  }
  else
  {
    i = 0.5 +
      std::ceil(0.001 + num / N) * N;
    val = 0.5 + (i * zoom);
  }
}

void AdjustUpdateRect(
  RECT& RUpdate, float zoom, int N)
```

```
{
  DecCoord(RUpdate.left, zoom, N);
  DecCoord(RUpdate.top, zoom, N);
  IncCoord(RUpdate.right, zoom, N);
  IncCoord(RUpdate.bottom, zoom, N);
}
```

The `IncCoord()` and `DecCoord()` functions will increase or decrease the specified value, `val`, such that when `val` is divided by `zoom`, the result is an integer. The parameter `N` denotes the precision of the zooming factor. When `N` is set to zero, this specifies that `zoom` is an integer (e.g., 2.0); when `N` is set to 10, this specifies that `zoom` contains one non-zero digit in the tenths decimal place (e.g., 2.5); and, when `N` is set to 100, this specifies that `zoom` also contains a non-zero digit in the hundredths decimal place (e.g., 2.56).

Using the `AdjustUpdateRect()` function, we can now modify the `PaintBitmap()` method as follows (here with a zooming factor of 2.56):

```
void TForm1::PaintBitmap(RECT& RUpdate)
{
  // scaling factor
  const float zoom = 2.56;

  // ensure that the coordinates of
  // RUpdate are an integer multiple
  // of the zooming factor
  int N = 100;
  AdjustUpdateRect(RUpdate, zoom, N);

  // destination rectangle
  const RECT RDst = RUpdate;

  // source rectangle
  const RECT RSrc = {
    0.5 + RUpdate.left / zoom,
    0.5 + RUpdate.top / zoom,
    0.5 + RUpdate.right / zoom,
    0.5 + RUpdate.bottom / zoom };

  StretchBlt(
    // destination DC
    hDstDC_,
    // destination coordinates
    RDst.left, RDst.top,
```

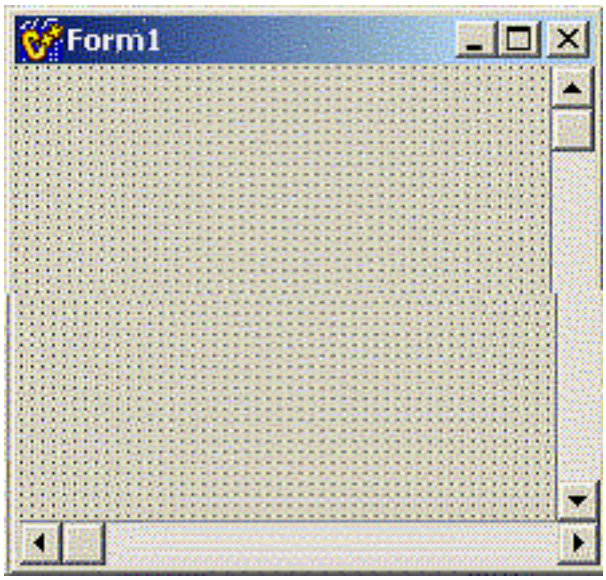
```
    RDst.right - RDst.left,
    RDst.bottom - RDst.top,
    // source DC
    Bitmap_->Canvas->Handle,
    // source coordinates
    RSrc.left, RSrc.top,
    RSrc.right - RSrc.left,
    RSrc.bottom - RSrc.top,
    // ROP3 code
    SRCCOPY);
}
```

By adjusting the update rectangle before dividing by `zoom`, we ensure that the coordinates of the source rectangle are precise integer values. In turn, this guarantees that the `StretchBlt()` function will perform the scaling consistently, regardless of which portion of the bitmap needs to be stretched.

Adding scrolling support

I've shown you how to draw only the required portion of a bitmap. Let's now work through an example of how to scroll the bitmap. We'll use the same setup as in the previous example, except the form in this case now contains two `TScrollBar`s (named `HorzSB` and `VertSB`) as depicted in **Figure A**. (I've decided to use `TScrollBar` controls instead of the form's built-in `TControlScrollBar`s to demonstrate the scroll bar-related code and the `ScrollWindowEx()` function.) The `TForm1` class declaration for this example is provided in **Listing B**.

Figure A



A form with two `TScrollBar` controls.

Setting up the scroll bars

The first thing we need to do is set the proper ranges and page sizes for the scroll bars. Because the required range of each scroll bar depends only on the dimensions of the bitmap and the zooming factor, we can set these ranges immediately after loading the image, like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner),
    Bitmap_(new Graphics::TBitmap()),
    zoom_(3.6)
{
  Bitmap_->LoadFromFile("my_bitmap.bmp");

  // set the scroll bars' ranges
  HorzSB->Max = zoom_ * Bitmap_->Width;
  VertSB->Max = zoom_ * Bitmap_->Height;
}
```

For this example, we'll assume a fixed zooming factor of 3.6 (held in the private `TForm1::zoom_` member.)

Whereas the ranges of the scroll bars define the total, "virtual" scrollable area (in our case, the dimensions of the zoomed image), the page sizes of the scroll bars define the area that's visible at any given time. Thus, we'll need to set the page size of each scroll bar from within the form's `OnResize` event handler:

```
void __fastcall
  TForm1::FormResize(TObject *Sender)
{
  // set the page sizes
  HorzSB->PageSize = std::min(
    ClientWidth - VertSB->Width,
    HorzSB->Max);
  VertSB->PageSize = std::min(
    ClientHeight - HorzSB->Height,
    VertSB->Max);

  // clip the scrolling positions
  HorzSB->Position = std::min(
    HorzSB->Max - HorzSB->PageSize,
    HorzSB->Position);
```

```
  VertSB->Position = std::min(
    VertSB->Max - VertSB->PageSize,
    VertSB->Position);

  // redraw the bitmap
  Invalidate();
}
```

By specifying a page size for each scroll bar, Windows will adjust the size of the thumb tabs in proportion to the ratio between the visible area (page size) and the total scrollable area (range). Also, notice that I've limited the scrolling position of each scroll bar to prevent scrolling past the last "page."

Scrolling the bitmap

Now that the scroll bars are set up, how do we actually scroll the bitmap? Well, the basic idea is to draw the bitmap at the new "location" defined by the `Position` of each scroll bar. Here's the definition of the `PaintBitmap()` method to do that:

```
void TForm1::PaintBitmap(RECT& RUpdate)
{
  // grab the scroll bars' Positions
  const int sX = HorzSB->Position;
  const int sY = VertSB->Position;

  // ensure that the coordinates of
  // RUpdate are an integer multiple
  // of the zooming factor
  int N = 10; // because zoom_ = 3.6
  AdjustUpdateRect(
    RUpdate, zoom_, N, sX, sY);

  // destination rectangle
  const RECT RDst = RUpdate;

  // source rectangle
  const RECT RSrc = {
    0.5 + (RUpdate.left + sX) / zoom_,
    0.5 + (RUpdate.top + sY) / zoom_,
    0.5 + (RUpdate.right + sX) / zoom_,
    0.5 + (RUpdate.bottom + sY) / zoom_
  };

  StretchBlt(
```

```
    // destination DC
    hDstDC_,
    // destination coordinates
    RDst.left, RDst.top,
    RDst.right - RDst.left,
    RDst.bottom - RDst.top,
    // source DC
    Bitmap_->Canvas->Handle,
    // source coordinates
    RSrc.left, RSrc.top,
    RSrc.right - RSrc.left,
    RSrc.bottom - RSrc.top,
    // ROP3 code
    SRCCOPY);
}
```

This code is similar to that of the previous `PaintBitmap()` definition. The main difference here is that the `Position`s of the scroll bars (`sX` and `sY`) are taken into account when defining the source rectangle. Remember, the source rectangle specifies which chunk of the bitmap to draw to the destination rectangle. By offsetting this chunk based on the `Position`s of the scroll bars, this code will map a different portion of the bitmap to the destination rectangle, effectively "scrolling" the bitmap.

Notice that when defining the source rectangle, we add `sX` and `sY` to the coordinates of the update rectangle *prior* to dividing by `zoom_`. Because of this fact, it is not necessary to ensure that the coordinates of the update rectangle are integer multiples of `zoom_`, but rather to insure that the *sums* of these coordinates and the scroll bars' `Position`s are integer multiples of `zoom_`. Accordingly, I've modified the `AdjustUpdateRect()` function to accept two additional parameters—`sX` and `sY`. Here's how the `AdjustUpdateRect()` function is now defined:

```
void AdjustUpdateRect(RECT& RUpdate,
  float zoom, int N, int sX, int sY)
{
  long x1 = RUpdate.left + sX;
  long y1 = RUpdate.top + sY;
  long x2 = RUpdate.right + sX;
  long y2 = RUpdate.bottom + sY;

  DecCoord(x1, zoom, N);
  DecCoord(y1, zoom, N);
  IncCoord(x2, zoom, N);
  IncCoord(y2, zoom, N);

  RUpdate.left = x1 - sX;
```

```
  RUpdate.top = y1 - sY;
  RUpdate.right = x2 - sX;
  RUpdate.bottom = y2 - sY;
}
```

Again, this function serves only to ensure that the coordinates of the update rectangle, when offset by the scroll bar's `Position`s, are all integer multiples of the zooming factor.

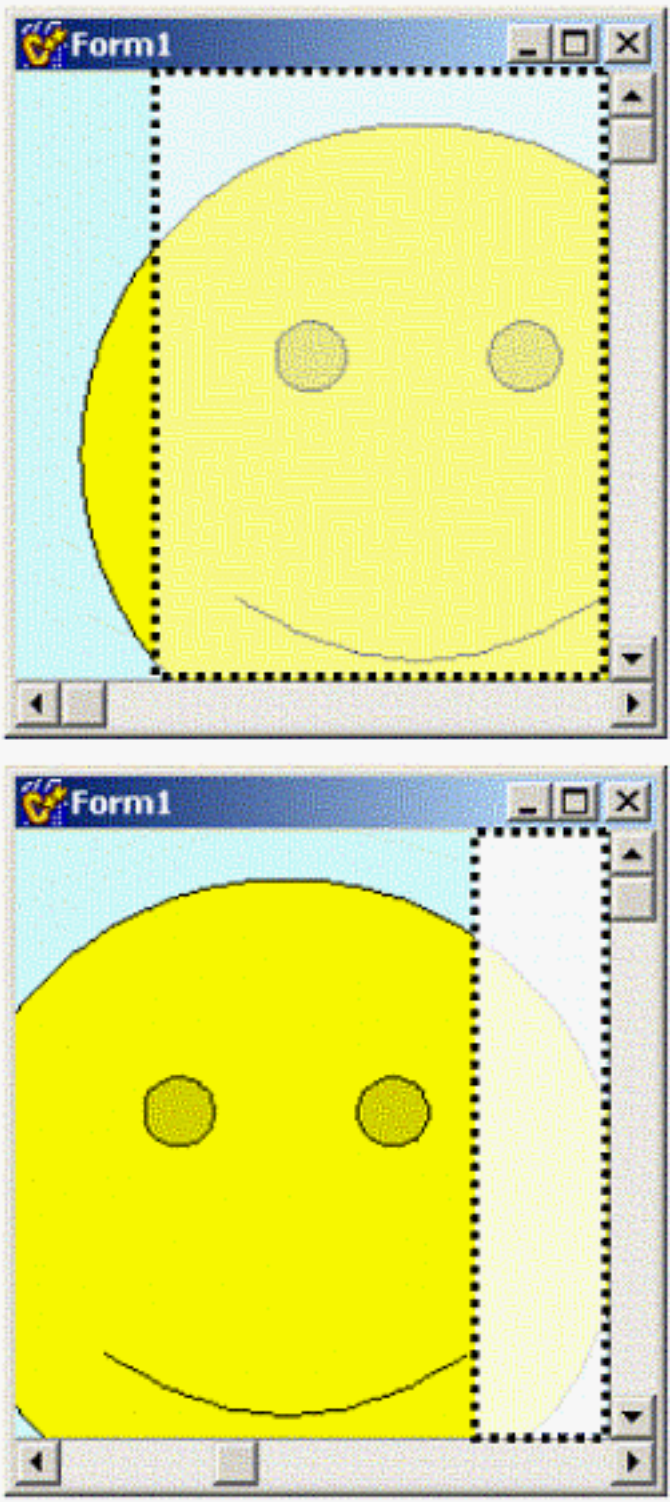
I've now shown you how to draw the bitmap at a new "location" based on the `Position` each scroll bar. All we need to do now is redraw the bitmap whenever the scroll bars are scrolled. One way to do this is as follows (using a shared `OnScroll` event handler for both scroll bars):

```
void __fastcall TForm1::HorzVertSBScroll(
  TObject *Sender, TScrollCode Code,
  int& ScrollPos)
{
  TScrollBar* SB =
    static_cast<TScrollBar*>(Sender);

  // clip the scrolling position
  ScrollPos = std::min(
    SB->Max - SB->PageSize,
    std::max(0, ScrollPos));

  // redraw the bitmap
  // (at the new position)
  InvalidateRect(
    WindowHandle, NULL, FALSE);
}
```

Here I've simply called the `InvalidateRect()` function to redraw the entire form (and thus the bitmap) upon a scroll. This approach works because our `PaintBitmap()` method takes into account the `Position` of each scroll bar. Unfortunately, this approach also obviates our efficient use of the update rectangle (because we're redrawing the entire area of the form).

A better approach is to compute the amount by which the user scrolled, move the bitmap that's already on the screen by that amount, and then redraw only the new "exposed" strip. **Figure B** illustrates this idea: the dotted rectangle in the top form denotes the area that needs to be moved to the left; the dotted rectangle in the bottom form denotes the "exposed" strip—only this area need be updated.

Figure B



The form before (top) and after (bottom) a horizontal scroll operation.

How do we move the bitmap that's already on the screen? Windows provides a function, called `ScrollWindowEx()`, to do just that. `ScrollWindowEx()` will offset a window's currently displayed contents by a specified amount. Here's what the function looks like:

```
int ScrollWindowEx(
  HWND hWnd,
```

```
  int dx,
  int dy,
  CONST RECT *prcScroll,
  CONST RECT *prcClip,
  HRGN hrgnUpdate,
  LPRECT prcUpdate,
  UINT flags);
```

The `hWnd` parameter specifies a handle to the window whose contents are to be scrolled. The `dx` and `dy` parameters specify by how much to scroll in the horizontal and vertical directions, respectively.

The `prcScroll` parameter specifies which area of the window to scroll. The `prcClip` parameter specifies which area of the window can be scrolled over. In other words, the area of the window that's defined by `prcScroll` can be moved *only* within the area defined by `prcClip`. This allows you to protect a portion of the window (`prcClip`) from being obscured by the chunk that's being moved (`prcScroll`). You can set both of these parameters to `NULL` if you want to scroll the window's entire client area.

The `hrgnUpdate` and `prcUpdate` parameters will receive, respectively, the region and rectangle of the new "exposed" area (i.e., the area to be redrawn). The `flags` parameter specifies if the `ScrollWindowEx()` function should invalidate this "exposed" area (so that the window will receive a `WM_PAINT` message with `PAINTSTRUCT::rcPaint` set to `prcUpdate`).

Using the `ScrollWindowEx()` function, the `OnScroll` event handler can be redefined as follows:

```
void __fastcall TForm1::HorzVertSBScroll(
  TObject *Sender, TScrollCode Code,
  int& ScrollPos)
{
  TScrollBar* SB =
    static_cast<TScrollBar*>(Sender);

  // clip the scrolling position
  ScrollPos = std::min(
    SB->Max - SB->PageSize,
    std::max(0, ScrollPos));

  // comute the amount to scroll by
  const int delta =
    SB->Position - ScrollPos;
  if (delta != 0)
  {
    if (SB == HorzSB)
```

```
    {
      // compute the area to be moved
      RECT RScroll = ClientRect;
      if (delta > 0) // move chunk right
      {
        RScroll.right -= delta;
      }
      else // move chunk left
      {
        RScroll.left -= delta;
      }

      // scroll the bitmap horizontally
      ScrollWindowEx(
       WindowHandle, delta, 0, &RScroll,
       NULL, NULL, NULL, SW_INVALIDATE);
    }
    else // SB == VertSB
    {
      // compute the area to be moved
      RECT RScroll = ClientRect;
      if (delta > 0) // move chunk down
      {
        RScroll.bottom -= delta;
      }
      else // move chunk up
      {
        RScroll.top -= delta;
      }

      // scroll the bitmap vertically
      ScrollWindowEx(
       WindowHandle, 0, delta, &RScroll,
       NULL, NULL, NULL, SW_INVALIDATE);
    }
  }
}
```

By using the `ScrollWindowEx()` function instead of redrawing the entire bitmap at its new location, the `PaintBitmap()` function performs a small stretch instead of a large one. Namely, the bitmap that’s currently displayed on the screen is already in video memory, so there’s no need to re-stretch and redraw that portion. Instead, `ScrollWindowEx()` will simply shift that portion over, eliminating the costly stretching. All that is then needed is to redraw the "exposed" strip. Note that the

`PaintBitmap()` method will be called automatically because `SW_INVALIDATE` is passed as the `flags` parameter to the `ScrollWindowEx()` function.

Conclusion

By drawing only the portion that's needed, the example presented here is significantly faster than what you'd achieve by using a `TImage` and a `TScrollBox`. In fact, because the source rectangle (passed to the `StretchBlt()` function) gets smaller as the zooming factor gets larger, you'll notice increased performance for higher zooms. (See also the article "Display-optimal DIB section bitmaps.")

There are a couple of caveats I should point out. First, I've neglected palette-related code throughout this article. If your application is running on a system with a palette-based display, you'll need to add support for palettes (see the section "Color Palettes" in the Windows SDK help files).

Second, the `StretchBlt()` function is limited on Windows 9x-based systems. On Win9x you can zoom up to a maximum of approximately 1100% before the function will fail. For higher zooms, you'll need to segment your bitmap into smaller bitmaps and then stretch and tile each bitmap segment.

Listing A: ***Declaration of the TForm1 class for the first example (w/o scroll bars)***

```
#include <memory>
class TForm1 : public TForm
{
__published:

private:
  HDC hDstDC_;
  std::auto_ptr<Graphics::TBitmap> Bitmap_;
  MESSAGE void WMPaint(TMessage& Msg);
  void PaintBitmap(RECT& RPaint);

public:
  __fastcall TForm1(TComponent* Owner);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TMessage, WMPaint)
END_MESSAGE_MAP(TForm)
};
```

Listing B: ***Declaration of the TForm1 class for the second example (with scroll bars)***

```
#include <memory>
```

```
class TForm1 : public TForm
{
__published:
  TScrollBar *HorzSB;
  TScrollBar *VertSB;
  void __fastcall FormResize(TObject *Sender);
  void __fastcall HorzVertSBScroll(TObject *Sender,
    TScrollCode Code, int &ScrollPos);

private:
  HDC hDstDC_;
  float zoom_;
  std::auto_ptr<Graphics::TBitmap> Bitmap_;
  MESSAGE void WMPaint(TMessage& Msg);
  void PaintBitmap(RECT& RPaint);

public:
  __fastcall TForm1(TComponent* Owner);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TMessage, WMPaint)
END_MESSAGE_MAP(TForm)
};
```

Using Windows file mapping for inter-process communications, Part 1

by Mark G. Wiseman

There are times when you want two or more processes running on the same computer to carry on a simple conversation with each other. There are several ways to do this under Windows.

You could use Dynamic Data Exchange (DDE). This is an older methodology that has been in Windows for some time. It is a little clunky and can be difficult to use.

You could also use the Component Object Model (COM). This is more modern that DDE, but it is even more difficult to use.

Writing and reading data to and from a shared file would normally be too slow for most applications and might put excessive stress on the hard drive.

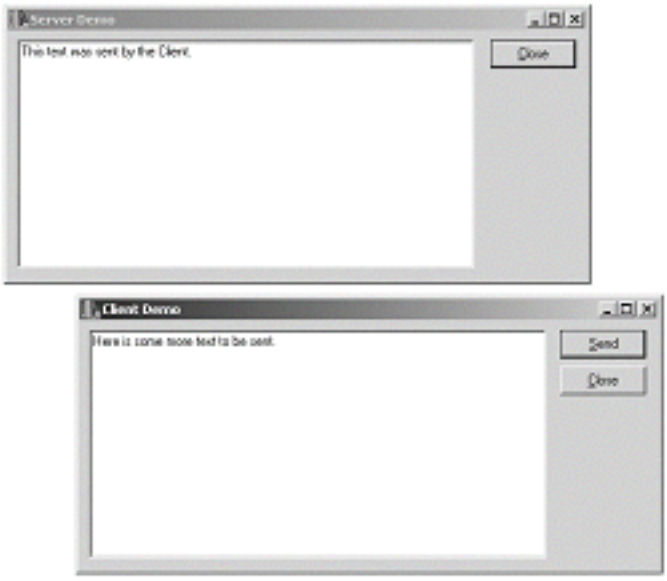
A shared space in memory would be ideal for simple inter-process communications. In Windows, shared memory is implemented through file mapping. In this article, I'm going to show you how to use file mapping to set up communication between two processes. In a follow-up article (Part 2) I'll discuss how to move the communication into its own thread and how to make the entire process more robust.

Demo programs

There are two demo programs that accompany this article. The source code for both of these programs can be found on the Bridges Publishing web site. The programs are a simple client and server.

The server program, **Listings A** and **B**, sets up a shared memory space using file mapping and waits for text to be placed into the memory by the client program, **Listings C** and **D**. Once the server finds some text in the shared memory it displays it. **Figure A** shows these programs in use.

Figure A



The client and server demo programs.

File mapping

In Windows, all memory allocated by Windows API functions such as `GlobalAlloc()` and `LocalAlloc()` can only be accessed by the process or application that allocated the memory. And, since C++ uses these underlying Windows API functions to implement the `new` operator, we can't allocate shared memory using `new` either.

The Windows API does have functions that support file mapping. Using file mapping, your application (process) can treat a file as if it were a block of memory in the address space of the process. This way, you can use simple pointer operations to read and write to the file. The Windows API also allows two or more processes to share a file mapping. Each process has a pointer to a memory in its own address space; but the pointers are all actually maps to a file.

The file itself can be a named file on a disk or it can be part of the Windows paging file. We will use the Windows paging file for programs.

Creating a file mapping

To create a file mapping, you use the Window function `CreateFileMapping()`. The prototype is:

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   LPSECURITY_ATTRIBUTES
      lpFileMappingAttributes,
```

```
DWORD flProtect,
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
LPCTSTR lpName
);
```

I will only discuss a two of the functions arguments. You can look up this, and the other functions, in the Windows online help provided with C++Builder.

The `hFile` argument is a handle to a file on disk. Since we are going to use the Windows page file, we use a special value of `0xFFFFFFFF` for `hFile`. If you wanted to use a named file, `hFile` would be the file handle you got when you created the name file with the Windows API function `CreateFile()`.

The `lpName` argument is a pointer to the name of the file mapping. This is different from any file name you might have used to create a file handle. This is an important name, because we are going to use this same name in both the client and the server demo programs so we can share the same file mapping. The name should be unique to these programs, but the same in both. I use the name "ReisdorphDemoFileMap" in the demo programs and actually create the file mapping in the server program.

When a file mapping has been created, you can open it in another process using the Windows API function `OpenFileMapping()` as I've done in the client program.

Both `CreateFileMapping()` and `OpenFileMapping()` return handles to the file mapping if they are successful. To convert these handles to a memory pointer, both programs use the `MapViewOfFile()` function, which takes as one of its arguments the file handle and returns a `void` pointer to a memory location within the application's memory space. I cast this `void` pointer to a `char` and use like any other pointer to `char`.

For these simple programs to work, the server application must be run first.

Communicating

I tried to make the communication between the two applications as simple as possible. The server program receives text messages from the client program. The server program uses a timer to periodically check the first byte of in the file mapping. If the byte is zero, nothing is done. If the byte is non-zero, the server application assumes that there is a text message sitting in the file mapping and it copies that text into a `TMemo` component for display. Finally, the server sets the first byte to zero, in effect resetting the file mapping.

The client program sends a text message to the server program by taking the text in a `TMemo` component

and copying it to the memory address of the file mapping.

Cleaning up

When the client and server programs are finished, they have to do a little clean up. First the view of the file mapping is released using the `UnmapViewOfFile()` function and then the file-mapping handle is released using the `CloseHandle()` function.

Windows is smart enough to not actually release a file mapping until every program using it has released it. So, the order in which two demo programs are closed is not important.

Conclusion

The demonstration programs discussed in this article represent the simplest example of inter-process communication I could think of using file mapping. Although, they give you a good idea of how file mapping works, there is a lot missing that you need to incorporate to make file mapping truly robust and useful.

In particular, the communication should take place in a separate thread, and the communication should be synchronized to prevent clashes.

I'll discuss this in Part 2 and I'll also show you how to deal with a missing server or client.

Listing A: *ServerMain.h*

```
class TServerForm : public TForm
{
  __published:
    TMemo *Memo1;
    TButton *CloseBtn;
    TTimer *Timer;

    void __fastcall OnTimer(TObject *Sender);
    void __fastcall Exit(TObject *Sender);

  public:
    __fastcall TServerForm(TComponent* Owner);
    __fastcall ~TServerForm();

  private:
    char *memmap;
```

```
    HANDLE fmHandle;
};
```

Listing B: *ServerMain.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "ServerMain.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TServerForm *ServerForm;

__fastcall TServerForm::TServerForm(
  TComponent* Owner) : TForm(Owner)
{
  Caption = Application->Title;

  fmHandle = CreateFileMapping(
    (HANDLE)0xFFFFFFFF, 0,
    PAGE_READWRITE, 0, 1024,
    "ReisdorphDemoFileMap");
  if (fmHandle == 0)
  {
    ShowMessage("Unable to create File Mapping.");
    Application->Terminate();
  }

  memmap =
    (char *)MapViewOfFile(fmHandle,
      FILE_MAP_ALL_ACCESS, 0, 0, 0);

  if (memmap == 0)
  {
    ShowMessage("Error!");
    Application->Terminate();
  }

  *memmap = 0;
}

__fastcall TServerForm::~TServerForm()
```

```
{
  if (memmap != 0)
    UnmapViewOfFile(memmap);

  if (fmHandle != 0)
    CloseHandle(fmHandle);
}

void __fastcall TServerForm::OnTimer(
  TObject *Sender)
{
  if (*memmap != 0)
  {
    Memo1->Lines->Text = memmap;
    *memmap = 0;
  }
}

void __fastcall TServerForm::Exit(TObject *Sender)
{
  Close();
}
```

Listing C: *ClientMain.h*

```
class TClientForm : public TForm
{
  __published:
    TMemo *Memo1;
    TButton *SendBtn;
    TButton *CloseBtn;
    void __fastcall Exit(TObject *Sender);
    void __fastcall Send(TObject *Sender);

  public:
    __fastcall TClientForm(TComponent* Owner);
    __fastcall ~TClientForm();

  protected:
    void __fastcall OnIdle(
      TObject *Sender, bool &Done);

  private:
    HANDLE fmHandle;
```

```
    char *memmap;
};
```

Listing D: *ClientMain.cpp*

```
__fastcall TClientForm::TClientForm(
   TComponent* Owner) : TForm(Owner)
{
  Caption = Application->Title;

  fmHandle =
    OpenFileMapping(FILE_MAP_ALL_ACCESS,
      false, "ReisdorphDemoFileMap");
  if (fmHandle == 0)
  {
    ShowMessage("Unable to create File Mapping.");
    Application->Terminate();
  }

  memmap =
    (char *)MapViewOfFile(fmHandle,
       FILE_MAP_ALL_ACCESS, 0, 0, 0);
  if (memmap == 0)
  {
    ShowMessage("Unable to map view of file.");
    Application->Terminate();
  }

  Application->OnIdle = OnIdle;
}

__fastcall TClientForm::~TClientForm()
{
  if (memmap != 0)
    UnmapViewOfFile(memmap);
  if (fmHandle != 0)
    CloseHandle(fmHandle);
}

void __fastcall TClientForm::Exit(TObject *Sender)
{
   Close();
}
```

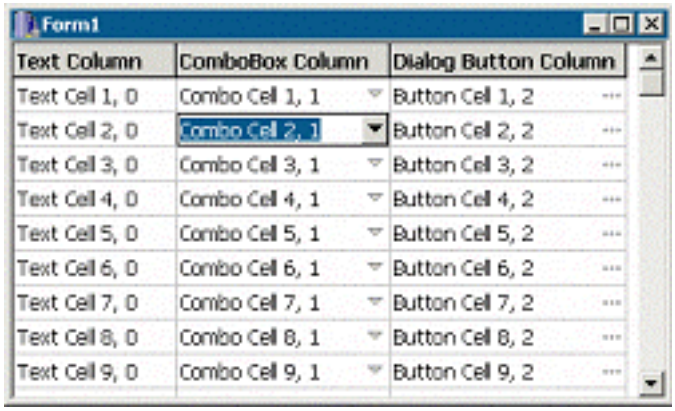
```
void __fastcall TClientForm::Send(TObject *Sender)
{
  lstrcpy(memmap, Memo1->Text.c_str());
  Memo1->Clear();
}

void __fastcall TClientForm::OnIdle(
  TObject *Sender, bool &Done)
{
  SendBtn->Enabled =
    Memo1->Lines->Text.IsEmpty() == false;
}
```

Even more string grids, part I

by Damon Chandler

Several months ago, I presented a technique for creating custom popup controls. And, several months prior to that, I demonstrated how to customize the appearance of a string grid. In this series of articles, I'll show you how to use these two techniques to add controls—such as a combo box—to the cells of a string grid. **Figure A** depicts an example.

Figure A



A `TStringGrid` object with customized cells.

TForm1::StringGrid1

This month, we'll work through an example of creating of the string grid depicted in **Figure A**. The first column contains "regular" cells—that is, cells that display just text. The second column contains cells that display a combo box. The third column contains cells that display a button that launches a dialog box.

The `TForm1` class for this example is declared in **Listing A**. `Form1` contains only a single control (`StringGrid1`), and one private member (`BtnDown_`) whose role I'll discuss later. The project also contains two other forms: a `TPopupForm` descendant (`PopupListBox`; see **Listing D**), and a generic dialog box form (`DialogForm`). (See the article "Custom popup controls" for the definition of the `TPopupForm` class.)

TObjects in each cell

For this example, we have three types of cells: cells that contain text, cells that contain a combo box, and

cells that contain a button. Accordingly, we'll need to associate data with the cells in order to store (and later retrieve) what control (if any) each cell contains. For this, we'll use the `TStringGrid::Objects` property, which is defined like so:

```
__property TObject*
  Objects[int ACol][int ARow] =
  {read=GetObjects, write=SetObjects};
```

Notice that the `Objects` property is used just like the `Cells` property; but, whereas the `Cells` property holds an `AnsiString`, the `Objects` property holds a pointer to a `TObject`. **Listing B** contains the definition of the `TObject` descendant class—`TCellObject`—that we'll use with the `Objects` property.

The `TCellObject` class is a bare-bones implementation of a class that can identify and interact with the information contained in each cell for this example. The `TCellObject::Type` property specifies the type of cell: `ctText` for a normal cell, `ctCombo` for a cell that contains a combo box, and `ctDialog` for cell that contains a button. I've also added properties called `Text`, `Col`, and `Row`, which are self-explanatory.

Before we associate a `TCellObject`–type object with each cell, let's define a couple of utility functions to retrieve the object held in each cell. These functions will eliminate the hassle of constantly casting the return value of the `Objects` property from `TObject` to `TCellObject`. Here's the code:

```
TCellObject* CellData(
  TStringGrid& Grid, int col, int row)
{
  if (col == -1 || row == -1 ||
      col >= Grid.ColCount ||
      row >= Grid.RowCount)
  {
    return NULL;
  }
  return static_cast<TCellObject*>(
    Grid.Objects[col][row]);
}

TCellObject* CellData(
  TStringGrid& Grid, TPoint P)
{
  int col, row;
  Grid.MouseToCell(P.x, P.y, col, row);
  if (col == -1 || row == -1 ||
      col >= Grid.ColCount ||
```

```
      row >= Grid.RowCount)
  {
    return NULL;
  }
  return static_cast<TCellObject*>(
    Grid.Objects[col][row]);
}
```

Both versions of the `CellData()` function return either a pointer to a `TCellObject`–type object or `NULL`, depending on whether or not the target cell is valid. In the first version of the `CellData()` function, the target cell is specified via column and row indices (`col` and `row`). In the second version of `CellData()`, the target cell is specified implicitly; the `TPoint`-type parameter (`P`) specifies the location of the cell in pixel-based coordinates (relative to the client area of the string grid).

Initializing the cells' objects

Now that the basic groundwork is set, let's associate a `TCellObject`-type object with each cell. We'll do this in the `TForm1` constructor, like so:

```
__fastcall TForm1::
  TForm1(TComponent* Owner)
    : TForm(Owner), BtnDown_(false)
{
  const int r_min =
    StringGrid1->FixedRows;
  const int r_max =
    StringGrid1->RowCount;

  StringGrid1->Cells[0][0] =
    "Text Column";
  StringGrid1->Cells[1][0] =
    "ComboBox Column";
  StringGrid1->Cells[2][0] =
    "Dialog Button Column";

  for (int r = r_min; r < r_max; ++r)
  {
    // first column
    TCellObject* pData =
      new TCellObject(StringGrid1,0,r);
    pData->Type = TCellObject::ctText;
    pData->Text = "Text Cell (" +
```

```
      IntToStr(r) + ", 0)";

    // second column
    pData =
      new TCellObject(StringGrid1,1,r);
    pData->Type = TCellObject::ctCombo;
    pData->Text = "Combo Cell (" +
      IntToStr(r) + ", 1)";

    // third column
    pData =
      new TCellObject(StringGrid1,2,r);
    pData->Type = TCellObject::ctDialog;
    pData->Text = "Button Cell (" +
      IntToStr(r) + ", 2)";
  }
}
```

Notice from **Listing B** that the `TCellObject` class will automatically associate itself with the cell that's specified via the `Grid`, `Col`, and `Row` parameters of the `TCellObject` constructor. As a result, we need only create a `TCellObject`-type object and pass its constructor the appropriate parameters; there's no need to access the string grid's `Objects` property directly.

Again, cells of the first column will contain just text; accordingly, so we set the `TCellObject::Type` property to `ctText` for these cells. Likewise, we set the `Type` property to `ctCombo` and `ctDialog` for cells of the second and third columns, respectively.

Deleting the cells' objects

Unfortunately, the string grid won't automatically delete the objects specified in its `Objects` property. Therefore, we'll delete each object manually in the `TForm1` destructor:

```
__fastcall TForm1::~TForm1()
{
  const int r_min =
    StringGrid1->FixedRows;
  const int r_max =
    StringGrid1->RowCount;

  for (int r = r_min; r < r_max; ++r)
  {
    delete CellData(*StringGrid1, 2, r);
```

```
    delete CellData(*StringGrid1, 1, r);
    delete CellData(*StringGrid1, 0, r);
  }
}
```

Drawing the cells' controls

Now that each cell has an object associated with it, we can use the `TCellObject::Type` property to determine what type of control (if any) the cell should contain. Namely, if `Type` is `ctCombo`, then the cell should contain a combo box; and if `Type` is `ctDialog`, then the cell should contain a button.

Of course, merely associating a `TCellObject`-type object with each cell won't automatically place the appropriate control in each cell; this task we'll need to do manually.

Unfortunately, several factors preclude placing a `TComboBox` object or a `TButton` object within each cell. One problem is that the string grid won't automatically move these controls when the grid is scrolled. Another problem is that—depending on the number of cells—a combo box or a button in each cell will consume a huge number of window handles (and thus system resources). Furthermore, adding so many controls to the string grid will likely make the grid appear too cluttered—notice from **Figure A** that only the focused cell has the control drawn in 3-D; the other cells display only an indicator glyph.

In the previous string grid article (see "More string grids" in the October 2000 issue), I showed you how to use the `DrawFrameControl()` function with the `TStringGrid::OnDrawCell` event to render a check box in each cell. We can use a similar approach here. Specifically, instead of placing a combo box or a button in each cell, we can render these controls manually from within `StringGrid1`'s `OnDrawCell` event handler. Here's the code for that:

```
void __fastcall TForm1::
  StringGrid1DrawCell(TObject *Sender,
    int ACol, int ARow, TRect &ARect,
    TGridDrawState State)
{
  TStringGrid& Grid =
    static_cast<TStringGrid&>(*Sender);
  TCanvas& SGCanvas = *Grid.Canvas;
  SGCanvas.Font = Grid.Font;
  RECT RText = ARect;

  // if the cell is fixed (header)
  if (State.Contains(gdFixed))
  {
    SGCanvas.Brush->Color =
```

```
    Grid.FixedColor;
  SGCanvas.FillRect(ARect);
  Frame3D(&SGCanvas, ARect,
    clBtnHighlight, clBtnShadow, 1);

  SGCanvas.Font->Style =
    SGCanvas.Font->Style << fsBold;
}
// if the cell is not a fixed cell
else
{
  SGCanvas.Brush->Color = Grid.Color;
  SGCanvas.FillRect(ARect);

  const TCellObject* pData =
    CellData(Grid, ACol, ARow);
  if (pData->Type !=
    TCellObject::ctText)
  {
    // make room for the button
    RText.right -= 18;

    // draw the combo-box or button
    DrawCellControl(SGCanvas, ARect,
      pData->Type, State, BtnDown_);
  }

  // if the cell is focused
  if (State.Contains(gdFocused))
  {
    //
    // render a black outline
    // (or use DrawFocusRect() if you
    // prefer a classic focus rect.)
    //
    SGCanvas.Pen->Color = clBlack;
    SGCanvas.Brush->Style = bsClear;
    SGCanvas.Rectangle(ARect);
    SGCanvas.Brush->Style = bsSolid;
  }

  bool selected =
    State.Contains(gdSelected);
  if (Grid.Options.
```

```
      Contains(goDrawFocusSelected))
    {
      selected = selected &&
        !State.Contains(gdFocused);
    }
    // if the cell is selected
    if (selected)
    {
      SGCanvas.Font->Color =
        clHighlightText;
      SGCanvas.Brush->Color =
        clHighlight;
    }
  }

  // draw the text
  RText.left += 2;
  RText.right -= 2;
  DrawText(
    SGCanvas.Handle,
    Grid.Cells[ACol][ARow].c_str(),
    -1, &RText, DT_LEFT |
    DT_SINGLELINE | DT_VCENTER
    );
}
```

This code is similar to the `OnDrawCell` event handler that I provided in the previous string grid article. The main difference here is that I've added code to draw a control within each cell if that cell's `TCellObject`-type object says to do so (i.e., if `Type` is `ctCombo` or `ctDialog`). This task is done by first calling the `CellData()` function to retrieve a pointer to the `TCellObject`-type object that's associated with the current ("to-be-drawn" cell). Then, the `TCellObject::Type` property's value is passed to the `DrawCellControl()` function to render the appropriate control.

The `DrawCellControl()` function renders either a combo-box scroll button (i.e., the drop-down button) or a button with an ellipsis (indicating that a dialog will be shown). Here's how the function is defined:

```
void DrawCellControl(
  TCanvas& Canvas, TRect& RCell,
  TCellObject::TCellType Type,
  TGridDrawState State, bool BtnDown)
{
  const TColor old_bsh_color =
```

```
  Canvas.Brush->Color;
const TColor old_pen_color =
  Canvas.Pen->Color;
try {
  Canvas.Brush->Color = clBtnFace;
  if (State.Contains(gdFocused))
  {
    TRect RBtn = RCell;
    RBtn.Left = RBtn.Right - 18;
    Canvas.FillRect(RBtn);

    if (BtnDown)
    {
      Frame3D(&Canvas,
        RBtn, clBtnShadow,
        clBtnHighlight, 1);
    }
    else
    {
      Frame3D(&Canvas,
        RBtn, clBtnHighlight,
        clBtnShadow, 2);
    }
    Canvas.Pen->Color = clBlack;
    Canvas.Brush->Color = clBlack;
  }
  else Canvas.Pen->Color =clBtnShadow;

  //
  // draw the combo box scroll arrow
  // or the dialog button ellipsis...
  //
  TRect RBtn = RCell;
  RBtn.Left = RBtn.Right - 18;
  if (Type == TCellObject::ctCombo)
  {
    TPoint Ps[3];
    Ps[0].x = RBtn.left - 4 +
      0.5 * (RBtn.right - RBtn.left);
    Ps[0].y = RBtn.top - 3 +
      0.5 * (RBtn.bottom - RBtn.top);
    if (BtnDown &&
      State.Contains(gdFocused))
```

```
      {
        Ps[0].y += 1;
      }

      Ps[1].x = Ps[0].x + 8;
      Ps[1].y = Ps[0].y;
      Ps[2].x = Ps[0].x + 4;
      Ps[2].y = Ps[0].y + 4;

      // draw the arrow
      Canvas.Polygon(Ps, 2);
    }
    else if (Type ==
      TCellObject::ctDialog)
    {
      TPoint P = Point(
        RBtn.left + 2, RBtn.top - 1 +
        0.5 * (RBtn.bottom - RBtn.top));
      if (BtnDown &&
        State.Contains(gdFocused))
      {
        P.y += 1;
      }

      // draw the ellipsis
      Canvas.Ellipse(P.x + 2, P.y,
        P.x + 5, P.y + 3);
      Canvas.Ellipse(P.x + 6, P.y,
        P.x + 9, P.y + 3);
      Canvas.Ellipse(P.x + 10, P.y,
        P.x + 13, P.y + 3);
    }
  }
  __finally {
    Canvas.Pen->Color = old_pen_color;
    Canvas.Brush->Color = old_bsh_color;
  }
}
```

The `Canvas` parameter specifies to which canvas the control should be drawn. The `RCell` parameter specifies the target cell's bounding rectangle. The `Type` parameter specifies the type of control to be drawn (`ctCombo` or `ctDialog`). The `BtnDown` parameter specifies whether the combo-box button or dialog button should be drawn in a pushed state. And, the `State` parameter specifies the state of the

target cell. If the cell is focused, the control is drawn in 3-D; otherwise, only the arrow or ellipsis is rendered (see **Figure A**).

Interacting with the cells' controls

At this point, we've done much of the work toward creating the string grid of **Figure A**. We have a string grid that will display either a combo box or an ellipsis button within each cell. Now it's time to provide functionality for these controls.

I just mentioned that the `BtnDown` parameter of the `DrawCellControl()` function specifies whether or not the button should be drawn as pushed. Furthermore, notice from the `StringGrid1DrawCell()` member function that the private `BtnDown_` member is passed as `DrawCellControl()`'s `BtnDown` parameter. Consequently, in order to make each button "clickable," we'll need to toggle the `BtnDown_` member whenever the user clicks the cell's button. For this, we'll use the string grid's `OnMouseDown` event handler, like so:

```
void __fastcall TForm1::
  StringGrid1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState,
    int X, int Y)
{
  if (Button != mbLeft) return;

  TStringGrid& Grid =
    static_cast<TStringGrid&>(*Sender);
  const TCellObject* pData =
    CellData(Grid, Point(X, Y));
  if (!pData) return;

  BtnDown_ = PtInBtn(Grid, X, Y);
  if (BtnDown_)
  {
    // draw the button as pushed
    RedrawBtn(
      Grid, pData->Col, pData->Row);
  }

  if (pData->Type ==
    TCellObject::ctCombo)
  {
    // show the popup window
    RECT RCell;
```

```
    CellRectFromPt(
      *StringGrid1, X, Y, RCell);

    PopupListBox->Left = RCell.left;
    PopupListBox->Top = RCell.bottom;
    PopupListBox->Width = max(
      RCell.right - RCell.left, 50L
      );
    PopupListBox->Grid = StringGrid1;
    //
    // add/remove items from
    // PopupListBox->ListBox
    // as needed...
    //
    PopupListBox->Show();
  }
  else if (pData->Type ==
    TCellObject::ctDialog)
  {
    // defer launching the dialog until
    // the mouse button is released
  }
}
```

Here, we first grab a pointer to the "clicked" cell's `TCellObject`-type object by using the version of the `CellData()` function that takes pixel-based coordinates (`X` and `Y`). We then use the `PtInBtn()` utility function (provided in **Listing C**) to determine whether or not the "clicked" point is within a cell's combo-box button or dialog button. If so, the cell's button is redrawn in the pushed state via the `RedrawBtn()` utility function (see **Listing C**).

Notice from this code that if the "clicked" cell is of type `ctCombo`, we determine the screen coordinates of the cell via the `CellRectFromPt()` utility function (see **Listing C**), and then we position and show the popup list box (i.e., the drop-down list of the combo box). I won't discuss the specifics of the popup list box, but I've provided its definition in **Listing D**. (See the article "Custom popup controls" in the May 2001 issue for more information on popup controls.)

If the "clicked" cell is of type `ctDialog`, we'll launch the dialog when and if the user releases the mouse button within the bounds of the cell's button. We can test for this condition from within the string grid's `OnMouseUp` event handler, like so:

```
void __fastcall TForm1::
  StringGrid1MouseUp(TObject *Sender,
    TMouseButton Btn, TShiftState Shift,
```

```
    int X, int Y)
{
  if (Btn != mbLeft || !BtnDown_)
    return;

  TStringGrid& Grid =
    static_cast<TStringGrid&>(*Sender);

  // draw the button as unpushed
  BtnDown_ = false;
  RedrawBtn(Grid, Grid.Col, Grid.Row);

  if (PtInBtn(Grid, X, Y))
  {
    const TCellObject* pData =
      CellData(Grid, Point(X, Y));

    if (pData->Type ==
      TCellObject::ctDialog)
    {
      // show the dialog
      DialogForm->ShowModal();

      //
      // adjust the cell's text
      // accordingly...
      //
    }
  }
}
```

Observe that two main tasks are performed within the `OnMouseUp` event handler: (1) the focused cell's button is redrawn in the un-pushed state; and (2) the dialog (`DialogForm`) is launched only if the mouse button is released while the cursor is over the button.

After the combo box or dialog is closed, how you adjust the cell is up to you. Notice from the `TPopupListBox:: ListBoxMouseUp()` member function of **Listing D** that the `TPopupListBox` class will automatically change the cell's text (via the `TStringGrid::Cells` property) according to which list box item the user selected. Depending on what you design `DialogForm` to do, you can use a similar approach when the dialog is closed.

Conclusion

In this article, I've demonstrated the basics of adding a combo box and a button to the cells of a string grid; and, I've shown you how to interact with these controls. (Actually, I've demonstrated only mouse-based interaction; I'll leave the keyboard-related code up to you). You can download the full code for this example at **www.bridgespublishing.com**.

Next month, I'll provide a more object-oriented approach to customization. I'll demonstrate how to create a `TStringGrid` descendant class with generic control-rendering functionality (e.g., an `OnDrawControl` event), and I'll show you how to customize the grid's in-place editor.

Listing A: ***Declaration of the TForm1 class***

```
class TForm1 : public TForm
{
__published:
  TStringGrid* StringGrid1;
  void __fastcall StringGrid1DrawCell(
    TObject* Sender, int ACol, int ARow,
    TRect &Rect, TGridDrawState State);
  void __fastcall StringGrid1MouseDown(
    TObject* Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  void __fastcall StringGrid1MouseUp(
    TObject* Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);

private:
  bool BtnDown_;

public:
   __fastcall TForm1(TComponent* Owner);
   __fastcall ~TForm1();
};
```

Listing B: ***Definition of the TCellObject class***

```
#include <cassert>
#include <Grids.hpp>
class TCellObject : public TObject
{
public:
  enum TCellType {ctText, ctCombo, ctDialog};

  __property TCellType Type =
```

```
    {read = Type_, write = DoSetType};
  __property AnsiString Text =
    {read = DoGetText, write = DoSetText};
  __property int Col = {read = Col_};
  __property int Row = {read = Row_};

public:
  __fastcall TCellObject(TStringGrid* Grid,
     int Col, int Row) : TObject(), Grid_(Grid),
     Row_(Row), Col_(Col), Type_(ctText)
    {
      assert(Grid_ != NULL);
      Grid_->Objects[Col_][Row_] = this;
    }

protected:
  virtual void __fastcall DoSetType(
    TCellType NewType)
    {
      if (Type_ != NewType)
      {
        Grid_->Objects[Col_][Row_] = this;
        Type_ = NewType;
      }
    }
  virtual AnsiString __fastcall DoGetText()
    {
      return Grid_->Cells[Col_][Row_];
    }
  virtual void __fastcall DoSetText(
    AnsiString NewText)
    {
      Grid_->Cells[Col_][Row_] = NewText;
    }

private:
  TStringGrid* Grid_;
  TCellType Type_;
  int Col_, Row_;
};
```

Listing C: ***Various string-grid-related utility functions***

```
void RedrawBtn(
  TStringGrid& Grid, int col, int row)
{
  RECT RBtn = Grid.CellRect(col, row);
  RBtn.left = RBtn.right - 18;

  RedrawWindow(
    Grid.Handle, &RBtn, NULL,
    RDW_INVALIDATE | RDW_UPDATENOW
    );
}

bool PtInBtn(TStringGrid& Grid, int X, int Y)
{
  int col, row;
  Grid.MouseToCell(X, Y, col, row);
  if (col != -1 && row != -1)
  {
    const POINT PMouse = {X, Y};
    RECT RBtn = Grid.CellRect(col, row);
    RBtn.left = RBtn.right - 18;

    return PtInRect(&RBtn, PMouse);
  }
  return false;
}

void CellRectFromPt(
  TStringGrid& Grid, int X, int Y, RECT& RCell)
{
  int col, row;
  Grid.MouseToCell(X, Y, col, row);
  if (col != -1 && row != -1)
  {
    RCell = Grid.CellRect(col, row);
    MapWindowPoints(
      Grid.Handle, HWND_DESKTOP,
      reinterpret_cast<POINT*>(&RCell), 2);
  }
  else SetRectEmpty(&RCell);
}
```

Listing D: ***The TPopupListBox class***

```
#include "POPUPFORMUNIT.h"
class TPopupListBox : public TPopupForm
{
__published:
  TListBox *ListBox;
  void __fastcall ListBoxMouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y);
  void __fastcall ListBoxMouseUp(TObject *Sender,
    TMouseButton Button, TShiftState Shift,
    int X, int Y);

private:
  TStringGrid* Grid_;

public:
  __fastcall TPopupListBox(TComponent* Owner)
    : TPopupForm(Owner) {}
  __property TStringGrid* Grid =
    {read = Grid_, write = Grid_};
};

void __fastcall TPopupListBox::
  ListBoxMouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
  if (Shift.Contains(ssLeft))
  {
    if (Grid_ != NULL)
    {
      POINT PMouse = {X, Y};
      MapWindowPoints(ListBox->Handle,
        Grid_->Handle, &PMouse, 1);

      RECT RBtn =
        Grid_->CellRect(Grid_->Col, Grid_->Row);
      RBtn.left = RBtn.right - 18;
      if (!PtInRect(&RBtn, PMouse))
      {
        PostMessage(
          Grid_->Handle, WM_LBUTTONUP, 0, 0);
      }
    }
  }
```

```
  const int hit_index =
    ListBox->ItemAtPos(Point(X, Y), false);
  if (hit_index != -1)
  {
    ListBox->ItemIndex = hit_index;
  }
}

void __fastcall TPopupListBox::
  ListBoxMouseUp(TObject *Sender,
    TMouseButton Btn, TShiftState Shift,
    int X, int Y)
{
  if (Grid_ != NULL)
  {
    PostMessage(
      Grid_->Handle, WM_LBUTTONUP, 0, 0);
  }

  const int hit_index =
    ListBox->ItemAtPos(Point(X, Y), true);
  if (hit_index != -1)
  {
    Close();
    if (Grid_ != NULL)
    {
      Grid_->Cells[Grid_->Col][Grid_->Row] =
        ListBox->Items->Strings[hit_index];
    }
  }
}
```

What's in a (form) name?

by Mark G. Wiseman

To paraphrase Shakespeare, "What's in a form name? That which we call TForm1 by any other name would display as well." As it turns out there is a lot to consider when naming a form, particularly the main form in a program.

I'll leave Romeo to Juliet and continue by paraphrasing T. S. Eliott:

The naming of forms is a difficult matter,

It isn't just one of your programmer games;

You may think that I am as mad as a hatter

When I tell you, a form must have THREE DIFFERENT NAMES.

The form class

First of all, there's the name that programmers use daily, Such as TForm1, TMainForm or TOpenDialog,

When you create a new Application in the IDE, a new class is created for you. This class is named `TForm1` by default. (The `Name` property in the Object Inspector.) This is a class derived publicly from `TForm`. `TForm` is the VCL encapsulation of a typical Windows application window or form.

You could leave this class named `TForm1`; but there is good reason to change it. `TForm1` gives no clue as to the function of the form. You could change it to `TMainForm`. In simple applications, I sometimes use `TMainForm`, especially if I will have other forms in the application.

However, when the form becomes complicated—when it has a lot of components, properties, data members and methods, I may want to be able to reuse that form. The IDE makes form reuse easy with the Object Repository. If you're not familiar with the Object Repository, check the IDE online help.

Naming a form's class `TMainForm` may be descriptive enough within an application, but it is not descriptive enough for the Object Repository.

Let's assume that we're writing a text editor program that mimics the Notepad program that comes with

Windows. It uses only one form, but it is a form that we might want to use again in another application. What if we named that form `TTextEditorForm` or `TNotepadForm`? Either of these would do nicely.

We can make it even better though. Below is the prototype for the Windows API function, `FindWindow()`:

```
HWND FindWindow(
   LPCTSTR lpClassName,
   LPCTSTR lpWindowName
   );
```

This function retrieves the handle to the top-level window whose class name and window name match the specified string arguments. The argument `lpClassName` is the form's class name, e.g. `TTextEditorForm`. The `lpWindowName` argument can be null or it can be the caption of the window. I'll talk more about window captions later.

If we wanted to find a running instance of our text editor application in Windows, we could make the following call to `FindWindow()`:

```
HWND h =
  FindWindow("TTextEditorForm", 0);
```

This would work fine, if there were no other program running that has a form with the same class name. But, since we can't control what other programmers may name their form classes, let's make our name more unique. Let's take the initials of Bridges Publishing and prefix our class name with them in lowercase. We'll use lowercase to make things more readable. Now we have `TrpTextEditorForm`. You can never be sure that this class name will be 100% unique, but it is unlikely you run into a conflict.

The form caption

But I tell you, a form needs a name that's particular, a name that's peculiar, and more dignified

The form's caption is the name that is displayed in the form's title bar. The `Caption` property of the form contains this text. This is the name that the user sees. It should be descriptive of the function of the form and may include additional information. Microsoft recommends that it contain the document name followed by a hyphen and the application name. In our text editor example, the Caption might be "Cats.txt – RP Text Editor". This tells the user that a document named "Cats.txt" is being edited by an application named "RP Text Editor".

To learn more about title bar captions, you can access *Microsoft's Official Guidelines for User Interface*

Developers and Designers online at

```
http://msdn.microsoft.com/library/
  default.asp?url=/library/en-us/
    dnwue/html/welcome.asp
```

Here is a function we might use in our editor application to set the form's caption:

```
void __fastcall TrpTextEditorForm
  ::SetCaption(String filename)
{
  Caption = filename + " – " +
    Application->Title;
}
```

You could call this function with "Untitled" as the `filename` argument when the application starts and call the function again whenever a file is loaded or saved.

The Application title

I'd also like to talk a little bit about the `Title` property of `TApplication`. You'll notice that I used it to build the forms caption in the example function above.

Normally, you set the application's title using the Project Options dialog box, under the `Application` tab. If you don't set this property, it will default to the name of the application's executable file.

Here's something you may not know. VCL applications create a hidden form to handle Windows messages and other tasks. This form's class name is "TApplication" and its caption is the value in the application's `Title` property. This can be useful information when you are using the `FindWindow()` function I mentioned above.

Conclusion

Hopefully, I've convinced you to give some thought to how you name your forms. If you were reading carefully, you'll notice that I talked about two names for your forms, the class name and the caption; but that I hinted there were three names.

But above and beyond there's still one name left over, and that is a name that you never will guess; the name that no programmer can discover—But Windows knows and will never confess.

Actually, Windows will confess if you named your form class well. It is the handle returned by the `FindWindows()` function. A *deep and inscrutable singular name*.

Quick and easy toolbar customization

by Damon Chandler

Many applications allow you to specify which buttons appear on its toolbars. In order to provide such a scheme in your own applications, you'll need a mechanism for dragging and dropping tool buttons. Fortunately, the VCL—namely, the `TControl` class—provides built-in drag-and-drop support. In this article, I'll provide a brief review of drag-and-drop, and then I'll show you how to use this technology to easily add customization support to your application's toolbars.

The DragMode property

All `TControl` descendants provide a `DragMode` property, which can take on one of two values: `dmManual` or `dmAutomatic`. When the `DragMode` property is set to `dmManual` (the default), the control behaves normally; specifically, the control does no special processing of mouse messages. In contrast, when `DragMode` is set to `dmAutomatic`, the control initiates the dragging process automatically by calling the `BeginDrag()` method in response to left mouse-button messages (`WM_LBUTTONDOWN` and `WM_LBUTTONDBLCLK`).

The OnStartDrag event

The `BeginDrag()` function puts the control into "dragging mode," at which point the control's `OnStartDrag` event is fired. Here's the signature of the `OnStartDrag` event:

```
typedef void __fastcall
  (__closure *TStartDragEvent)(
    System::TObject* Sender,
    TDragObject* &DragObject);
```

The `Sender` parameter specifies a pointer to the control that's being dragged (i.e., the control whose `BeingDrag()` method was called). The `DragObject` parameter is a reference to a `TDragObject` instance that will be used during the dragging process. As I'll discuss shortly, you can use the `DragObject` parameter to specify custom drag-and-drop cursors. If you don't assign a value to the `DragObject` parameter, the `TControl` class will create its own `TDragObject`-type object (specifically, of type `TDragControlObject`).

The OnDragOver event

After the `OnStartDrag` event handler is called, all subsequent mouse messages are sent to `DragObject`'s internal window. The `TDragObject` class uses these messages to determine how the dragging process should proceed. For example, as the control (i.e., the control that's being dragged) is dragged over other controls, `DragObject` receives the `WM_MOUSEMOVE` message, in response to which, it determines which VCL control is under the mouse cursor and then calls that control's `OnDragOver` event handler:

```
typedef void __fastcall
  (__closure *TDragOverEvent)(
    System::TObject* Sender,
    System::TObject* Source,
    int X, int Y,
    TDragState State,
    bool &Accept);
```

The `Sender` parameter specifies a pointer to the control that's being dragged over (i.e., the control under the mouse cursor). The `Source` parameter specifies either a pointer to the `TDragObject`-type instance that you assigned to the `DragObject` parameter in the `OnStartDrag` event handler, or a pointer to the control that's being dragged (i.e., the control whose `BeingDrag()` method was called) if you left the `DragObject` parameter at its default value of `NULL`. The `X` and `Y` parameters specify the location of the mouse cursor in coordinates relative to the client area of the control that's being dragged over. The `State` parameter specifies whether the mouse cursor has just entered (`dsDragEnter`), is leaving (`dsDragLeave`), or is moving over (`dsDragMove`) the control's client area. The `Accept` parameter is your way of indicating whether or not to accept the drop. Namely, if the control that's being dragged can be dropped on the control that's being dragged over, you set the `Accept` parameter to `true`; otherwise, you set `Accept` to `false`. `DragObject` will change the mouse cursor accordingly to provide visual feedback to the user.

The OnDragDrop event

When the user drops the control that's being dragged onto the control that's being dragged over (and the latter control accepts the drop by setting the `Accept` parameter to `true` in its `OnDragOver` event handler), the latter control's `OnDragDrop` event is fired. Here's what that event looks like:

```
typedef void __fastcall
  (__closure *TDragDropEvent)(
    System::TObject* Sender,
    System::TObject* Source,
    int X, int Y);
```

The `Sender` parameter specifies a pointer to the control that accepted the drop (i.e., the control that was being dragged over). The `Source` parameter specifies either a pointer to the `TDragObject`-type

instance that you assigned to the DragObject parameter in the OnStartDrag event handler, or a pointer to the control that was dropped (i.e., the control that was being dragged) if you left the DragObject parameter at its default value of NULL. The X and Y parameters specify the location of the mouse cursor (in coordinates relative to the client area of the control that's being dragged over).

Keep in mind that, although the TControl class will *initiate* the drag-and-drop process for you, it's up to you to handle the drop (e.g., by changing a control's location).

The OnEndDrag event

The OnEndDrag event is fired after the user drops the control (regardless of whether or not the drop was accepted). In contrast to OnDragOver and OnDragDrop, the OnEndDrag event handler is called for the control that was being dragged:

```
typedef void __fastcall
  (__closure *TEndDragEvent)(
    System::TObject* Sender,
    System::TObject* Target,
    int X, int Y);
```

The Sender parameter specifies a pointer to the control that was being dragged. The Target parameter specifies a pointer to the control on which Sender was dropped. If no control accepted the drop, or if Sender was dropped on a non-VCL control, the Target parameter will indicate NULL. If Target is non-NULL, then the X and Y parameters specify the location of the mouse cursor in coordinates relative to the client area of the control that's being dragged over. If Target is NULL and Sender was dropped within the application, then the X and Y parameters will both be zero. If Target is NULL and Sender was dropped outside of the application (e.g., on the desktop), then the X and Y parameters specify the location of the mouse cursor in screen coordinates.

Using a TDragObject descendant

As I mentioned earlier, you don't have to rely on the TControl class's internal TDragObject-type instance to handle the dragging and dropping. Instead, you can assign your own TDragObject-type instance to the DragObject parameter in the OnStartDrag event. Why would you want to do this? Well, the TDragObject class provides two very useful virtual methods: GetDragImages() and GetDragCursor(). By overriding these methods, you have complete control over the cursors that will be used during the drag-and-drop process. In addition, by using your own TDragObject descendant class, you can easily check whether or not to accept a drop. Recall that if you assign your own TDragObject-type instance to the DragObject property in the OnStartDrag event, the Source parameter of the OnDragOver event will specify a pointer to your TDragObject-type instance. This way, you don't have to compare the Source parameter with a specific control; rather,

you simply check whether or not `Source` is indeed your `TDragObject`-type descendant.

Listing A contains the definition of a `TDragControlObject` descendant class, `TDragTBObject`, which we'll use later for toolbar customization. Specifically, this class overrides the `GetDragCursor()` method in order to display custom, toolbar-specific drag-and-drop cursors. Also, the `GetDragImages()` method is coded to always return `NULL`; this guarantees that the custom cursors will always be used, regardless of the control that's being dragged. (List views and tree-views, for example, display their own drag-and-drop cursors; returning `NULL` in `GetDragImages()` will prevent these cursors from being generated.)

Drag-and-drop customization for toolbars

To keep things as clear as possible, I'll demonstrate toolbar customization by using utility functions and methods of the `TForm1` and `TCustomizeDlg` classes, whose declarations are provided in **Listings B** and **C**. The application's main form, `Form1`, and the customization dialog, `CustomizeDlg`, are depicted in **Figure A**. (For now, ignore the fact that the toolbars contain no buttons. In the article "**Saving and loading components**," I'll show you how to stream the toolbars to and from a file.)

Recall that there are two main customization features that require drag-and-drop support: (1) the user needs to be able to drag-and-drop new buttons onto the toolbar; and (2) the user needs to be able to rearrange the toolbar's existing buttons. `Form1` contains one `TCoolBar` (`CoolBar1`) that contains four child `TToolBar` controls: `FileToolBar`, `EditToolBar`, `FrmtToolBar`, and `ViewToolBar`. The customization dialog includes a list view (`CommandListView`), which lists the available commands; this list view will serve as the source of new buttons.

Figure A

The application's main form (`Form1`), which contains four toolbars, the customization dialog (`CustomizeDlg`), and the toolbar popup menu (`TBPopupMenu`).

Initiating the customization process

Notice from **Figure A** that `Form1` contains a popup menu with a menu item titled "Customize…" We'll initiate the customization process when this menu item is clicked:

```
void __fastcall TForm1::
  CustomizeMenuItemClick(TObject *Sender)
{
```

```
  CustomizeDlg->Show();
  CustomizeMode(true);
}

void __fastcall TForm1::
  CustomizeMode(bool customize)
{
  CustomizeToolBar(
    *FileToolBar, customize);
  CustomizeToolBar(
    *EditToolBar, customize);
  CustomizeToolBar(
    *FrmtToolBar, customize);
  CustomizeToolBar(
    *ViewToolBar, customize);

  CustomizeMenuItem->
    Enabled = !customize;
}
```

As I just mentioned, there are two drag sources: the toolbar's existing tool buttons and the customization dialog's `CommandListView`. Because the customization dialog is displayed only during customization, we can set the `CommandListView`'s `DragMode` property to `dmAutomatic` at design time. In contrast, we'll need to toggle the tool buttons' `DragMode` properties (between `dmManual` and `dmAutomatic`) at run time so that the tool buttons can be dragged and dropped only during customization. We'll do this from within the `CustomizeToolBar()` method, which is defined as follows:

```
void __fastcall TForm1::CustomizeToolBar(
   TToolBar& ToolBar, bool customize)
{
  const int count = ToolBar.ButtonCount;
  for (int idx = 0; idx < count; ++idx)
  {
    TToolButton& Btn =
      *ToolBar.Buttons[idx];
    Btn.DragMode = customize ?
      dmAutomatic : dmManual;
    Btn.OnStartDrag = ToolBtnsStartDrag;
    Btn.OnDragOver = ToolBtnsDragOver;
    Btn.OnDragDrop = ToolBtnsDragDrop;
    Btn.OnEndDrag = ToolBtnsEndDrag;
  }
```

```
}
```

Unfortunately, setting the `DragMode` property to `dmAutomatic` will work only for "regular" tool buttons. Because separators don't receive the `WM_LBUTTONDOWN` message, the `BeginDrag()` method is never called for these types of tool buttons. So, in order to drag-and-drop a separator, we'll need to forward the `WM_LBUTTONDOWN` message to the separators from within the toolbar's `OnMouseDown` event handler:

```
void __fastcall TForm1::
  ToolBarsMouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
{
  if (Button == mbLeft)
  {
    // grab a reference to the toolbar on
    // which the mouse button was pressed
    TToolBar& ToolBar =
      static_cast<TToolBar&>(*Sender);

    // if the cursor is over a separator
    const int sep_index =
      PtOnSeparator(ToolBar, X, Y);
    if (sep_index >= 0)
    {
      ToolBar.Buttons[sep_index]->
        Perform(WM_LBUTTONDOWN, 0, 0);
    }
  }
}
```

The `ToolBarsMouseDown()` method is assigned to all four toolbars at design time. This method uses the `X` and `Y` parameters with the `PtOnSeparator()` utility function to determine if the mouse cursor is over a separator. Here's how the `PtOnSeparator()` function is defined:

```
int PtOnSeparator(
   TToolBar& ToolBar, int X, int Y)
{
  RECT RBtn;
  const POINT PMouse = {X, Y};
  const HWND hToolBar = ToolBar.Handle;
  const int count = ToolBar.ButtonCount;
```

```
  for (int idx = 0; idx < count; ++idx)
  {
    TToolButton& Btn =
      *ToolBar.Buttons[idx];
    if (Btn.Style == tbsDivider ||
        Btn.Style == tbsSeparator)
    {
      SNDMSG(
        hToolBar, TB_GETITEMRECT, idx,
        reinterpret_cast<LPARAM>(&RBtn)
        );
      if (PtInRect(&RBtn, PMouse))
      {
        return idx;
      }
    }
  }
  return -1;
}
```

This function simply iterates over all of `Toolbar`'s buttons, checking to see whether the point specified by the `X` and `Y` parameters lies inside the separator's bounding rectangle. `PtOnSeparator()` returns either the index of the "hit" separator or –1.

OnStartDrag event handlers

As you can see from the `CustomizeToolBar()` method, all tool buttons will share the same `OnStartDrag` event handler, `ToolBtnsStartDrag()`. `CommandListView` is assigned it's own `OnStartDrag` event handler, `CommandListViewStartDrag()`, at design time. Here's how these handlers are defined:

```
void __fastcall TForm1::
  ToolBtnsStartDrag(TObject *Sender,
    TDragObject *&DragObject)
{
  TToolButton& Btn =
    static_cast<TToolButton&>(*Sender);
  Btn.Marked = true;
  DragTBObject_->Control = &Btn;
  DragObject = DragTBObject_.get();
}
```

```
void __fastcall TCustomizeDlg::
  CommandListViewStartDrag(
    TObject *Sender,
    TDragObject *&DragObject)
{
  DragTBObject_->Control =
    CommandListView;
  DragObject = DragTBObject_.get();
}
```

Recall that the `OnStartDrag` event handler is called immediately after the user begins to drag a control. For the tool buttons, we grab a reference to the `TToolButton` that's being dragged (passed via the `Sender` parameter) and then set the button's `Marked` property to `true` so that the user knows which button he or she is dragging. For `CommandListView`, the selected item is automatically highlighted. In both cases, we assign the `DragObject` parameter a pointer to each class's private `DragTBObject_` member (see **Listings A** and **B**), which is created in the class's constructor.

Providing visual feedback

While the user drags a `TToolButton` or list-view item over the toolbars, we'll need provide two forms of visual feedback: (1) an indication whether or not the drop will be accepted, and (2) an indication of where on the toolbar the button will be inserted when dropped.

We can provide visual feedback of the acceptance status by adjusting the current mouse cursor. As I mentioned earlier, we'll do this by using the `TDragTBObject` class of **Listing A**. To indicate where on the toolbar the button will be inserted, we'll use the `TB_SETINSERTMARK` message, as demonstrated in the following utility functions:

```
void ShowInsertMark(TToolBar& ToolBar,
  TToolButton& Btn, bool after)
{
  TBINSERTMARK tbim;
  tbim.iButton = Btn.Index;
  tbim.dwFlags = after;

  SNDMSG(
    ToolBar.Handle, TB_SETINSERTMARK, 0,
    reinterpret_cast<LPARAM>(&tbim));
}

void HideInsertMark(TToolBar& ToolBar)
{
```

```
  const TBINSERTMARK tbim = {-1};
  SNDMSG(
    ToolBar.Handle, TB_SETINSERTMARK, 0,
    reinterpret_cast<LPARAM>(&tbim));
}
```

The `TB_SETINSERTMARK` message is sent with an `lParam` value that's assigned a pointer to `TBINSERTMARK` structure. This structure contains two data members: `iButton`, which specifies the index of the `TToolButton` before or after which the insertion mark should be displayed; and `dwFlags`, which specifies whether the insertion mark should be displayed before (0) or after (1) the button indicated by `iButton`. As you can see from the definition of the `HideInsertMark()` utility function, when the `iButton` data member is set to –1, the insertion mark is removed.

OnDragOver event handlers

Recall that the `OnDragOver` event is called for the control that's being dragged over (not for the control that's being dragged). For this reason, we don't need an `OnDragOver` event handler for `CommandListView`. We do however, still need two `OnDragOver` event handlers: one for the toolbar's tool buttons, and one for the toolbar itself.

As the user drags a `TToolButton` (or list-view item) over the toolbar, one of two events will be fired depending on which control the mouse cursor is over. If the mouse cursor moves over another "regular" `TToolButton`, then that tool button's `OnDragOver` event handler will be called. If, however, the mouse cursor moves over a part of the toolbar that contains either a separator or no child control, then the toolbar's `OnDragOver` event handler will be called.

Notice from the declaration of the `CustomizeToolBar()` method that we've assigned the same `OnDragOver` event handler, `ToolBtnsDragOver`, to all of the tool buttons; here's how that handler is defined:

```
void __fastcall TForm1::ToolBtnsDragOver(
   TObject *Sender, TObject *Source,
   int X, int Y, TDragState State,
   bool &Accept)
{
  // determine whether to accept the drop
  Accept =
    dynamic_cast<TDragTBObject*>(Source);
  if (!Accept) return;

  // grab a reference to the
  // button that's being dragged over
```

```
  TToolButton& Btn =
    static_cast<TToolButton&>(*Sender);

  // grab a reference to the
  // button's toolbar
  TToolBar& ToolBar =
    static_cast<TToolBar&>(*Btn.Parent);

  // if the cursor is leaving the button
  if (State == dsDragLeave)
  {
    // hide the insertion mark
    HideInsertMark(ToolBar);
  }
  // otherwise
  else
  {
    // display an insertion mark
    // before or after the button
    const bool after = X > Btn.Width / 2;
    ShowInsertMark(ToolBar, Btn, after);
  }
}
```

Because we assigned a value to the `DragObject` parameter in the `OnStartDrag` event, the `Source` parameter of our `ToolBtnsDragOver()` member function will specify a pointer to a `TDragTBObject`-type object if the control that's being dragged is indeed a `TToolButton` or one of `CommandListView`'s items. We can therefore set the `Accept` parameter depending on whether or not `Source` points to a `TDragTBObject`-type object.

The `ToolBtnsDragOver()` event handler will handle the case in which the user drags a button (or list-view item) over a "regular" `TToolButton`. As I mentioned earlier however, if the user moves the mouse cursor over an empty toolbar or over a separator, the toolbar's `OnDragOver` event handler will be called. For this reason, we assign all of `Form1`'s toolbars the following `OnDragOver` event handler:

```
void __fastcall TForm1::ToolBarsDragOver(
   TObject *Sender, TObject *Source,
   int X, int Y, TDragState State,
   bool &Accept)
{
  // determine whether to accept the drop
  Accept =
    dynamic_cast<TDragTBObject*>(Source);
```

```
if (!Accept) return;

// grab a reference to the
// toolbar that's being dragged over
TToolBar& ToolBar =
  static_cast<TToolBar&>(*Sender);

// if the cursor is leaving the toolbar
if (State == dsDragLeave)
{
  // hide the insertion mark
  HideInsertMark(ToolBar);
}
// otherwise, if the
// toolbar contains buttons
else if (ToolBar.ButtonCount > 0)
{
  const int index =
    PtOnSeparator(ToolBar, X, Y);
  if (index >= 0)
  {
    // show an insertion mark before
    // or after the separator
    TToolButton& SepBtn =
      *ToolBar.Buttons[index];
    const bool after = X - SepBtn.Left
      > SepBtn.Width / 2;
    ShowInsertMark(
      ToolBar, SepBtn, after);
  }
  else
  {
    // show an insertion mark
    // after the last button
    TToolButton& LastBtn = *ToolBar.
      Buttons[ToolBar.ButtonCount - 1];
    ShowInsertMark(
      ToolBar, LastBtn, true);
  }
}
// otherwise
else
{
```

```
    // ToolBar contains no buttons so
    // accept drops anywhere on it
  }
}
```

This method is similar to the `ToolBtnsDragOver()` method except the `Sender` parameter, this time, specifies a pointer to the toolbar. That takes care of the `OnDragOver` event handlers. Let's now tackle the `OnDragDrop` event handlers.

OnDragDrop event handlers

All tool buttons will share the same `OnDragDrop` event handler, `ToolBtnsDragDrop()`; all four toolbars will share the handler `ToolBarsDragDrop()`. Here's how these two methods are defined:

```
void __fastcall TForm1::ToolBtnsDragDrop(
   TObject *Sender, TObject *Source,
   int X, int Y)
{
  // grab a reference to the
  // control that's being dragged
  TControl& SrcCtl =
    *static_cast<TDragTBObject*>
      (Source)->Control;
  // grab a reference to the button
  // that's being dragged over
  TToolButton& Btn =
    static_cast<TToolButton&>(*Sender);
  // grab a reference to the
  // button's parent (toolbar)
  TToolBar& ToolBar =
    static_cast<TToolBar&>(*Btn.Parent);

  // move/insert the button
  const bool after = X > 0.5 * Btn.Width;
  DoDrop(&ToolBar, &Btn, &SrcCtl, after);
}

void __fastcall TForm1::ToolBarsDragDrop(
   TObject *Sender, TObject *Source,
   int X, int Y)
{
  // grab a reference to the
```

```
// control that's being dragged
TControl& SrcCtl =
  *static_cast<TDragTBObject*>
    (Source)->Control;
// grab a reference to the toolbar
// that's being dragged over
TToolBar& ToolBar =
  static_cast<TToolBar&>(*Sender);

// if the toolbar contains buttons
if (ToolBar.ButtonCount > 0)
{
  // find the index of the separator
  // that's being dragged over
  const int index =
    PtOnSeparator(ToolBar, X, Y);
  if (index >= 0)
  {
    // grab a reference to the button
    // that's being dragged over
    TToolButton& SepBtn =
      *ToolBar.Buttons[index];

    // move/insert the source button
    const bool after = X - SepBtn.Left
      > 0.5 * SepBtn.Width;
    DoDrop(&ToolBar, &SepBtn,
      &SrcCtl, after);
  }
  else
  {
    // grab a reference to the
    // last button on the toolbar
    TToolButton& LastBtn = *ToolBar.
      Buttons[ToolBar.ButtonCount - 1];

    // move/insert the source button
    DoDrop(&ToolBar, &LastBtn,
      &SrcCtl, true);
  }
}
// otherwise
else
{
```

```
    // accept drops anywhere
    DoDrop(&ToolBar, NULL,
      &SrcCtl, false);
  }
}
```

These methods are similar to the corresponding `OnDragOver` event handlers, except that in this case, we actually move or insert the button instead of showing or hiding an insertion mark. This task is handled in the `DoDrop()` utility method, which is defined as follows:

```
void __fastcall TForm1::DoDrop(
   TToolBar& ToolBar, TToolButton* Btn,
   TControl* SrcCtl, bool after)
{
  // if SrcCtl is a toolbutton
  if (dynamic_cast<TToolButton*>(SrcCtl))
  {
    SrcCtl->Parent = &ToolBar;
    if (Btn)
    {
      SrcCtl->Left = after ?
        Btn->Left + Btn->Width :
        Btn->Left;
    }
  }
  // otherwise, we're adding a new button
  else
  {
    // grab a pointer to
    // the dropped list item
    TListView& SrcListView =
      static_cast<TListView&>(*SrcCtl);
    TListItem* DragItem =
      SrcListView.Selected;

    // create a corresponding toolbutton
    TToolButton* NewBtn =
      new TToolButton(this);
    NewBtn->Parent = &ToolBar;
    NewBtn->DragMode = dmAutomatic;
    NewBtn->OnStartDrag =
      ToolBtnsStartDrag;
    NewBtn->OnEndDrag =
```

```
      ToolBtnsEndDrag;
    NewBtn->OnDragOver =
      ToolBtnsDragOver;
    NewBtn->OnDragDrop =
      ToolBtnsDragDrop;
    // (separator is the last image)
    if (DragItem->ImageIndex ==
        TBImageList->Count - 1)
    {
      NewBtn->Style = tbsSeparator;
      NewBtn->Width = 8;
    }
    else
    {
      NewBtn->Caption =
        DragItem->Caption;
      NewBtn->ImageIndex =
        DragItem->ImageIndex;
    }
    // position the new button
    if (Btn)
    {
      NewBtn->Left = after ?
        Btn->Left + Btn->Width :
        Btn->Left;
    }
  }
  // fix the command IDs
  ResetCommandIDs(ToolBar);
}
```

The `DoDrop()` method is designed to either move an existing button or add a new button, depending on the `SrcCtl` parameter. The `SrcCtl` specifies a pointer to the control that was dropped. If this control is a `TToolButton`, then we simply move it to location at which it was dropped—namely, to the right-hand side of the `TToolButton` that's specified by the `Btn` parameter). If `SrcCtl` is not a `TToolButton` (i.e., if it's `CommandListView`) then we add a new `TToolButton` to the toolbar, again to the right-hand side of the `TToolButton` specified by the `Btn` parameter. Note from the definition of the `ToolBarsDragDrop()` method that `DoDrop()` might be called with a `NULL` `Btn` parameter. In this case, we simply add or move the new or existing button to (0,0).

Notice that a call to another utility function, `ResetCommandIDs()`, is made at the end of the `DoDrop()` method. This function compensates for a bug in the `TToolButton` class, which scrambles the index-to-command-identifier assignments. Normally, all `TToolButtons` are assigned a command

identifier that corresponds to its ordinal location within the toolbar (i.e., the same value as it's `Index` property). This correspondence is needed for the `Marked` and `Indeterminate` properties. The `ResetCommandIDs()` function simply updates the command ID of each `TToolButton` to reflect its `Index`:

```
void ResetCommandIDs(TToolBar& ToolBar)
{
  const HWND hToolBar = ToolBar.Handle;
  const int count = ToolBar.ButtonCount;
  for (int idx = 0; idx < count; ++idx)
  {
    SNDMSG(
      hToolBar, TB_SETCMDID, idx, idx);
  }
}
```

Conclusion

In this article, I've shown you how to add drag-and-drop customization to toolbars. Although the presented code is designed only for horizontal toolbars, it shouldn't be difficult to modify for vertical toolbars. Also, I've neglected to mention how to handle the tool button's event handlers. Users of newer versions of C++Builder can use a `TActionList` object; otherwise, I recommend using a single, shared handler for each event, querying, for example, the tool button's (`Sender`'s) `ImageIndex` property to determine what action to perform.

The sample project depicted in **Figure A** is available for download from **www.residorph.com**. In addition to the code presented here, this project demonstrates how to code the specifics of the customization dialog and the toolbar popup menu.

Listing A: ***Declaration of the TDragTBObject class***

```
static const TCursor crTBDropAdd = 1;
static const TCursor crTBDropMove = 2;
static const TCursor crTBDropRemove = 3;

class TDragTBObject : public TDragControlObject
{
public:
  __fastcall TDragTBObject(TControl* Control)
    : TDragControlObject(Control) {}

protected:
```

```
  virtual TCursor __fastcall
    GetDragCursor(bool Accepted, int X, int Y)
    {
      if (Accepted)
      {
        return dynamic_cast<TToolButton*>(Control) ?
          crTBDropMove : crTBDropAdd;
      }
      return crTBDropRemove;
    }
  virtual TDragImageList* __fastcall GetDragImages()
    {
      return NULL;
    }
};
```

Listing B: ***Declaration of the TForm1 class***

```
#include <memory>
class TForm1 : public TForm
{
__published:
  TCoolBar *CoolBar1;
  TToolBar *FileToolBar;
  TToolBar *EditToolBar;
  TToolBar *FrmtToolBar;
  TToolBar *ViewToolBar;
  TImageList *TBImageList;

  TPopupMenu *TBPopupMenu;
  TMenuItem *FileMenuItem;
  TMenuItem *EditMenuItem;
  TMenuItem *ViewMenuItem;
  TMenuItem *FrmtMenuItem;
  TMenuItem *CustomizeMenuItem;

  // drag-and-drop support for regular toolbuttons
  void __fastcall ToolBtnsStartDrag(
    TObject *Sender, TDragObject *&DragObject);
  void __fastcall ToolBtnsEndDrag(
    TObject *Sender, TObject *Target, int X, int Y);
  void __fastcall ToolBtnsDragOver(
    TObject *Sender, TObject *Source,
```

```
    int X, int Y, TDragState State, bool &Accept);
  void __fastcall ToolBtnsDragDrop(
    TObject *Sender, TObject *Source, int X, int Y);

  // drag-and-drop support for separator toolbuttons
  void __fastcall ToolBarsMouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  void __fastcall ToolBarsDragOver(
    TObject *Sender, TObject *Source, int X, int Y,
    TDragState State, bool &Accept);
  void __fastcall ToolBarsDragDrop(
    TObject *Sender, TObject *Source, int X, int Y);

  // TBPopupMenu event handlers
  void __fastcall CustomizeMenuItemClick(
    TObject *Sender);

private:
  std::auto_ptr<TDragTBObject>
    DragTBObject_;

  // toolbar utility methods
  void __fastcall CustomizeToolBar(
    TToolBar& ToolBar, bool customize);
  void __fastcall DoDrop(
    TToolBar& ToolBar, TToolButton* Btn,
    TControl* SrcCtl, bool after);

public:
  __fastcall TForm1(TComponent* Owner);
  void __fastcall CustomizeMode(bool customize);
};
```

Listing C: ***Declaration of the TCustomizeDlg class***

```
#include <memory>
class TCustomizeDlg : public TForm
{
__published:
  TListView *CommandListView;
private:
  std::auto_ptr<TDragTBObject>
    DragTBObject_;
```

```
public:
  virtual __fastcall TCustomizeDlg(
    TComponent* AOwner);
};
```

Saving and loading components

by Damon Chandler

As you know, the C++Builder IDE stores and loads a project's design-time forms and components to and from a `.DFM` file. The core of this work is delegated to two special component-streaming classes: `TWriter` and `TReader`. In this article, I'll show you how to use these classes to easily save and load components to and from a file.

Saving components to a file

The `TWriter` class contains numerous methods for writing a component's properties to a stream. Here, I'll demonstrate the `TWriter::WriteComponent()` method, which can be used to save all of a component's (and its sub-components') streamable properties, including event-type properties. Here's a utility function that uses the `WriteComponent()` method to save a component to a file:

```
#include <memory>
#include <cassert>
int __fastcall SaveComponent(
   AnsiString filename,
   TComponent* Component
  )
{
  assert(Component != NULL);
  assert(Component->Owner != NULL);

  std::auto_ptr<TFileStream> fs(
    new TFileStream(filename, fmCreate)
    );
  std::auto_ptr<TWriter> Writer(
    new TWriter(fs.get(), 4096)
    );

  // specify the Root component
  Writer->Root = Component->Owner;
  // write the component
  Writer->WriteComponent(Component);

  // return the number of bytes wrote
  return Writer->Position;
}
```

The `TWriter` class is designed for use with a `TStream`-type object. Here, I've used a `TFileStream` object whose pointer is passed as the first parameter of the `TWriter` constructor. The second parameter of the `TWriter` constructor is an integer that specifies how many bytes the `TWriter` object should use for its internal memory buffer. The `TWriter` class will write first to this buffer, and then to the stream when the buffer is full.

After the `TWriter` object is created, all that's left is to set its `Root` property and then call the `WriteComponent()` method. The `Root` property specifies the owner of the component that's to be saved. The `TWriter` class will stream a component's event-type properties only if the assigned event handlers are within `Root`'s `__published` section. Thus, to stream a component, it's important that: (1) the component and all of its sub-components share the same `Owner`; and (2) all of the component's (and sub-components') event handlers are within `Root`'s `__published` section. For example, if you stream a `TTabControl` object that contains child controls, you must make sure that all of these child controls have the same `Owner` as the tab control; and, you must make sure that all of the controls' events handlers are within this `Owner`'s `__published` section. The first restriction isn't much of a problem if all of the controls are created at design time; recall that the parent form owns all design-time created controls. Keep this restriction in mind, though, when creating controls dynamically.

Loading components from a file

The `TWriter` class has a counterpart, `TReader`, that's designed to load components from a stream. As you might have guessed, the specific method for doing this is called `ReadComponent()`. Here's a utility function that demonstrates how to use the `TReader::ReadComponent()` method to load a component (including its sub-components) from a file:

```
int __fastcall LoadComponent(
   AnsiString filename,
   TComponent*& Component
  )
{
  assert(Component != NULL);
  assert(Component->Owner != NULL);

  std::auto_ptr<TFileStream> fs(
    new TFileStream(filename, fmOpenRead)
    );
  std::auto_ptr<TReader> Reader(
    new TReader(fs.get(), 4096)
    );

  // set Root, Owner, and Parent
```

```
  Reader->Root = Component->Owner;
  TControl* Control =
    dynamic_cast<TControl*>(Component);
  if (Control) {
    Reader->Parent = Control->Parent;
  }

  // remove the existing component
  delete Component; Component = NULL;

  // load the stored component
  Reader->BeginReferences();
  try {
    Component =
      Reader->ReadComponent(NULL);
  }
  __finally {
    Reader->FixupReferences();
    Reader->EndReferences();
  }

  // return the number of bytes read
  return Reader->Position;
}
```

Because the `TWriter` and `TReader` classes descend from the same parent class (`TFiler`), both classes share a common interface. The parameters of the `TReader` constructor, for example, are the same as those of the `TWriter` constructor. Similarly, the `TReader` class has a `Root` property that serves the same role as the aforementioned `TWriter::Root` property. Notice from this code however, that the `TReader` class has an additional property, `Parent`, that needs to be set if the component to be loaded is a `TControl`-type descendant. This specification is needed because the `TControl::Parent` property isn't streamed.

The `LoadComponent()` utility function is designed for use in replacing an existing component with its stored counterpart. You specify the existing component via the `Component` parameter, whose value will point to the newly loaded component. Thus, after the `TReader::Root` and `Parent` properties are set, the existing component (`Component`) is deleted, and then its stored counterpart is loaded by using the `ReadComponent()` method. The call to the `ReadComponent()` method is preceded by a call to the `BeginReferences()` method and followed by calls to `FixupReferences()` and `EndReferences()`. These latter three methods ensure that all components in the stream are created before any component references are assigned.

Conclusion

There’s one caveat that you need to be aware of: The classes of all components loaded from a file must be registered with the streaming system before you call the `LoadComponent()` function. The code that accompanies this article (available at **www.bridgespublishing.com**) demonstrates how to do this, and it provides examples of how to use the `SaveComponent()` and `LoadComponent()` functions.

The basics of multi-tier applications

by Mark Cashman

Multi-tier applications offer a variety of advantages over single and two-tier applications. Before I get into the advantages of multi-tier applications, let me explain the various options available to you.

Single, two, or multi-tiered?

In a single tier, the application, the DBMS and the database all reside on the same system, or the database is a file whose directory is mounted as a network drive. While single-tier applications are fine for individuals, when there is a need to share data across a group, they often run into problems with locking, database consistency and performance. In addition, any changes to the application need to be propagated to all of the workstations where the application is running.

In a two-tier or client/server application, we most commonly see an executable on a workstation accessing a database on a server through a server process that runs on the server and mediates access to the database. This increases the safety of database operations, because only a single process is really changing the content of the database. Most often, as with SQL Server and Oracle, these server processes are provided by the DBMS vendor. One can also create non-database client/server applications, by writing a server process of any kind and using techniques such as sockets, named pipes, or CORBA to communicate between the client and the server. Examples of this sort of server might be a time server that provides an accurate shared timestamp, or a server to mediate access to a variety of sensors in a manufacturing application, or even a computation server that offers extremely powerful CPUs to execute demanding computations such as ray tracing of 3D objects for rendering photo realistic virtual images. However, in this case, the client code still usually embodies all of the rules of the application, and changes to the application need to still be propagated to all of the workstations where the client may run.

Advantages of multi-tiered databases

In a multi-tier application, at least one other machine lives (conceptually) between the client and the server. For many multi-tier applications (usually three-tier applications) the central tier embodies various rules which format or process information from one or more database servers. In this case, the middle-tier is the client of the server tier, and is the server for the client.

This offers both the advantage of control over database access as provided by client-server designs, and also reduces the problem of needing to propagate updates to the application. This is because much of the application logic can be embodied in the middle tier, leaving only the user interface on the client. And, of course, this means separate teams can work on the middle and client tier, only needing to agree on interfaces.

There can, of course, be more than three tiers. For instance, a middle tier may draw some of its data from another tier (usually referred to as an application server, but this is not quite the same term as it is in the Java world), merging that data (which had been drawn in turn from a database server) with its own data from its own database server, and then sending the combined data to the client. Another example might come from the world of web applications, where the client is the browser, its server is the web server, the web server runs a page provider CGI which in turn contacts a "middle-tier" data access application, which, in turn, contacts a database server (such as a machine running MS SQL Server or Oracle).

Another potential advantage for a multi-tier application is performance. When the client only has to run the user interface, the user interface will, in theory, be faster. The same holds true for each element of the chain. However, there are caveats to this.

The end-to-end throughput of a multi-tier system will be slightly lower than the equivalent single or two-tier system, due to the additional time needed to marshal (assemble/translate) and transmit data across the network between tiers. This is always true if the transactions performed in the multi-tier system (read or update) are synchronous—that is, if all operations must complete before the client can continue. However, this can be outweighed in a situation where the client workstations are relatively lightweight and the servers are beefy.

When requests from multiple clients are passing through a tier simultaneously there is no free lunch just because the middle and back tiers are multithreaded. Indeed, compared to a single tier system, an insufficiency of processing power on any tier (i.e. any tier is not *n* times more powerful than the client workstation, where *n* is the number of simultaneous requests) can choke the performance of the multi-tier system. Threads, except where the tier processor has multiple CPUs, actually introduce additional overhead as the operating system switches between them. On the other hand, if one request is waiting for a disk access to complete, another request’s thread may be able to simultaneously set up for its own disk access while the wait is continuing. This is where the raw performance leverage of the multi-tier system comes into play.

Another advantage of the multi-tier approach is scalability. If multiple machines can be used to support the middle tier application servers, then that improves throughput—potentially in proportion to the amount of hardware you are willing to throw at the problem.

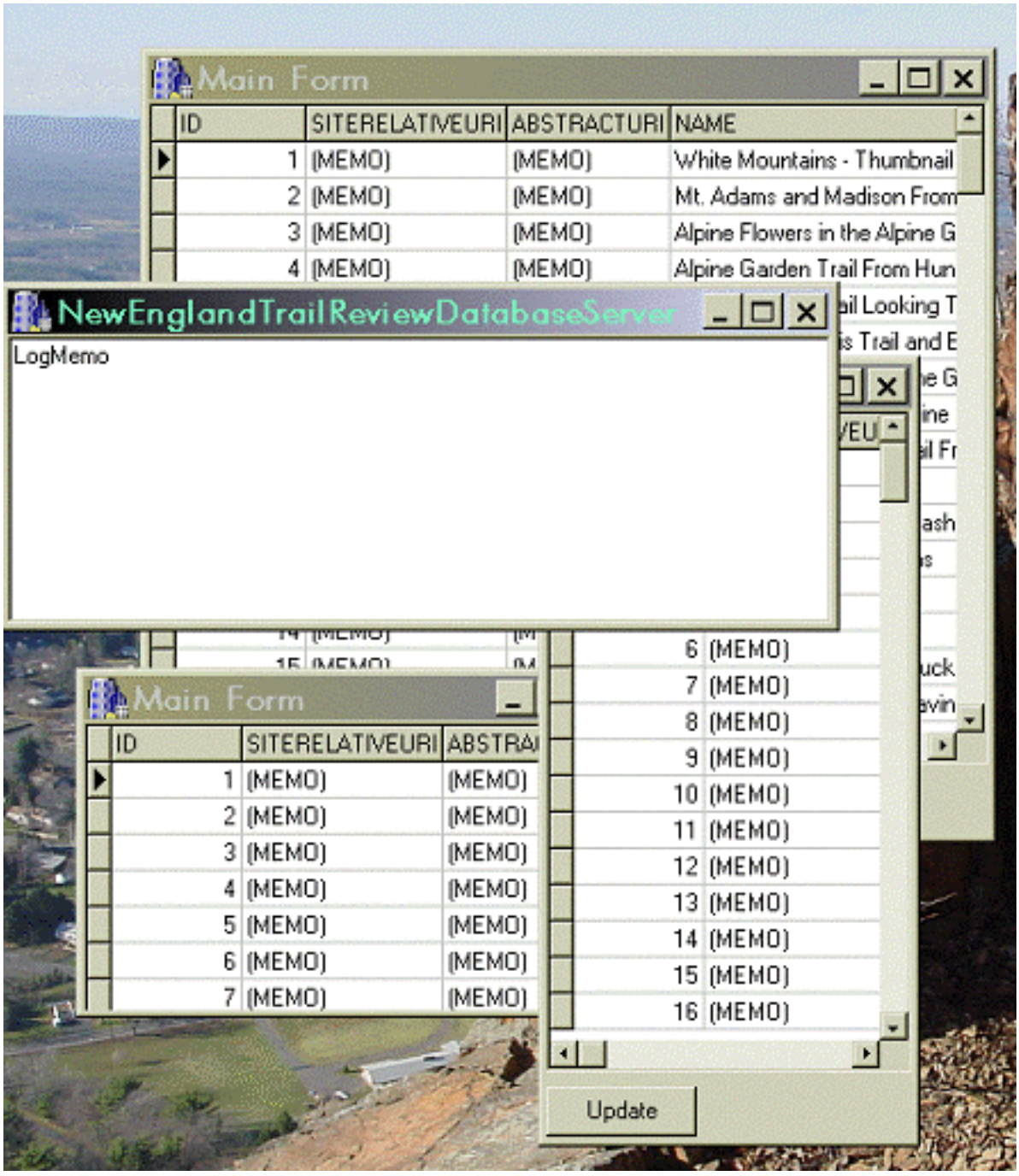
Multi-tiered applications in C++Builder

One of the most powerful ways to create a multi-tier application in C++Builder is to use MIDAS (now called DataSnap) to connect a client application to a middle-tier remote data module (RDM). The RDM is essentially identical to a conventional data module, but it is wrapped in a COM object to allow communication with a client. It can connect to a local or remote database. It also offers a data provider, which manages the transfer of data packets from the RDM to the client.

The client of the remote data module is a separate `.exe`, which has a client data set for each data set it wants to use from the RDM. The client data set caches the rows from the RDM's data set, so that the client basically runs almost as fast as if the data were stored locally. But of course, to be useful and scalable, the RDM will generally not be located on the same machine as the client. The typical exception is during development, when you will normally want the RDM running on the same machine as the client.

In **Figure A**, you can see the RDM and three clients running locally.

Figure A



An example of the RDM and three clients running locally.

There are two major ways to run a RDM:

- One process per request—multiple copies of the remote data module on the same CPU.

- One process for all requests—a single copy of the remote data module on a single CPU handles all requests.

Figure A shows the latter. The latter option can be implemented in two different styles:

- Single threaded—when multiple requests are made, they each wait until the preceding request has been satisfied.

- Multi-threaded—when multiple requests are made, each gets a thread of its own. This has potentially higher performance, since it can take advantage of disk access delays or other conditions in prior requests, to service requests (overall) more quickly.

Note that the Multi-threaded style of server does not guarantee the completion of requests in a first come, first complete order, even if all requests pass through the same processing instructions. This is because requests can block on disk access, and the disk location of particular data (and thus the latency) cannot be predicted. In addition, the application server developer must be very careful to guard all potentially shared objects that do not themselves have thread safety features, often through the use of critical sections or semaphores.

Creating the server

It is fairly straightforward to create the server. Create a new application project in BCB. While you can eliminate the form from the project, it is wise instead to leave that as an application server monitor. You can put a `TMemo` on it, for instance, and add messages to the memo as well as sending them (for instance) to a log.

Next, use File | New | Multi-tier to create a Remote Data Module. Select a name for the class and enter that as the "CoClass Name". For instance, you may want to call it "SomethingServer".

You can select any threading model. Apartment threading is recommended for BDE datasets but it does not allow more than one request to pass through the application server at the same time. Free threading can be used, and it will allow more than one request to pass through the application server at the same time, but it is not recommended for use with BDE datasets. Instead, the use of ADO datasets is recommended in this case. Note that if you ignore this advice, you may not encounter repeatable errors,

because errors will be the result of a combination of requests that cannot be repeated.

When the remote data module is created, it gets a lot of files:

- `SomethingServerDataImpl.h` and `.cpp`—this is the data module.

- `SomethingServer.cpp`—this is the project process, which instantiates the data module and the application server main form (if any).

- `SomethingServer.tlb`—this is the type library for the server's COM interface, which allows it to be invoked by MIDAS. This contains one other important thing—the GUID used to identify the server to local COM and to Distributed COM (DCOM).

- `SomethingServer_ATL.cpp` and `SomethingServer_TLB.cpp`—these provide the interfaces to the Active Template Library and TLB representations of the COM interface to the server. These are generated and you should generally leave them alone.

At this point you have an empty but usable server.

Populating the remote data module

At this point you need to decide whether to use the BDE or ADO components. In part, this decision will be driven by your threading model, which is driven by your performance requirements. In this instance, we will discuss the use of the BDE components.

A remote data module should have a `TSession`, a `TDatabase` and any needed `TDataSet` descendants. For each data set, it also needs a `TDataSetProvider`, which will manage the movement of data from the database to the client. The `TSession` needs to have `AutoSessionName` set to `true`, because it will require that for thread safety.

Making sure only one instance of the server is created

In the unit `SomethingServerDataImpl`, you need to change the line

```
regObj.Singleton = false
```

to

```
regObj.Singleton = true.
```

Completing the server for the development machine

The server now has all of its data sets and providers, and is ready to run. Compile and link the server. All is not yet complete, however. At an MSDOS window's prompt, change the directory to the development directory for the server, and run the server with the `/regserver` parameter (for instance "`SomethingServer /regserver`"). This ensures the proper registry entries are made. This same action will have to be performed to make the server available on whatever machine may be its ultimate production destination as well.

Creating the client

A client can be created with File | New Application. It is also recommended that you add a data module to correspond to your RDM.

A client needs to know how to contact the server. For this, you need to use the MIDAS connection components on the MIDAS tab of the component palette. For this server, use the DCOM Connection component. You can (and this is recommended), add it to the data module for the application. Select the server from the component's `ServerName` property drop down. You will find your registered server in the list because you have registered it. Select the appropriate name. Note that this will stimulate the server to run and to respond to design-time requests, such as the need to populate a client `TDBGrid`. The server will also run any time you reopen this project in the future.

Now, from the MIDAS tab, add `TClientDataSet` components for each data set in the RDM that you want the client to access. Note that you do not need one for every data set in the RDM—you only need one for each data set you intend to use. Then you should add `TDataSource` components for each one of the client data sets that you intend to have represented in the user interface.

The client data set components then need to be connected to the MIDAS connection object by selecting the connection object from the `RemoteServer` property drop down. Then pick the ProviderName from the ProviderName combo, which will list those from the server you selected.

If you open the client data sets, they will establish connection to the server just as they would establish connection to a database if they were conventional data sets.

You can at any time connect conventional data aware controls to the data sources of the client data sets—controls like `TDBGrid`, or third party data aware components, like the Woll2Woll or TurboPower database grids. All of these work normally, without any need to be aware of the fact that they are not

accessing a client server or desktop database.

Testing locally

You can now run the client, which will automatically run the server. You can use standard debugging in the client. If you would like to debug the server, you need to run it before running the client, from the IDE. Then set your breakpoints there. This will work, even if the client is remote. You may also be able to do this with remote debugging from the client machine or from a third machine, but that is beyond the scope of this article.

Special things to remember

Note that some of the standard event handlers on a `TTable` do not seem to get called. For instance before and after post events are not called when you apply updates from a client unless you set `ResolveToDataSet` on the `Provider` component for that data set.

Deploying Remotely

Once you are satisfied, you need to deploy the application server to `/Remote Data Module.exe` or to the production machine (or install it on the customer machine). The deployment is fairly straightforward, in the sense that the same actions you performed on the development machine to register the server must be performed.

On the client machine, however, things are a little different, because you have to establish a DCOM entry for the server on the client.

If you are running under Windows 9x, you need to set your network security (Control Panel | Networks) to User Level Access Control. You will need to have a WindowsNT server offering the domain user authentication for the network or you cannot run DCOM remotely on a network of Windows 9x machines.

Assuming your network meets these criteria, on the server machine run a DOS Window and cd to `c:\windows\system`. Enter the command "dcomcnfg" and press Enter.

Select your server from the Application tab and click Properties. You can change the security settings (for instance by making the server only able to be accessed by members of a specific group of users), and even the location where the server will actually be run. Note that if the server is being run on a Windows 9x machine, you cannot have the server be automatically started—you must start it manually or programmatically from its own computer so that it is ready to be called.

From an administrative perspective, the securing of the server to a specific group is the best alternative, since that reduces the situations under which one might need to come back and re-administer the server.

Conclusion

Creating a multi-tier system is not so hard, at least not when using Borland C++Builder. The biggest problems are those of deployment and administration.

In regard to scalability, you must generally step outside the C++Builder framework to obtain that. For instance, you will need a broker or a hardware load balancer to ensure that requests are balanced across the middle-tier machines available. You can, if you have to, of course, write an application server for an intermediate tier to select among several machines on a per request or per client basis.

Still, it is certainly gratifying when you get your first distributed application up and running. Enjoy it!

An HTML calendar generator

by Mark G. Wiseman

If you have read many of my articles, you'll know that I am still surprised and pleased at how easy C++Builder makes coding many projects. You'll also know that I like creating components for reuse. In this article, I'm going to show you a component that I created to generate monthly calendar pages in HTML. And guess what? It was very easy to create!

I used the IDE's Component Wizard to write the skeleton code for a new component, `THTMLCalendarGenerator`, derived from `TComponent`. Then I added only about 150 lines of code and ended up with a very useful component.

The history

I was asked by a client, a medical clinic, to help them convert data from their scheduling system into a monthly calendar they could display on their Web site. The calendar would show which days doctors were scheduled to work at each of their offices.

I wrote the code to extract the data I needed for each month, and then wrote some code to format that data into HTML code that would display a monthly calendar.

I noticed that the code to generate the HTML calendar was fairly generic. That is, the code was not specific to this one project. It was also encapsulated in a single C++ class. This made a little bell go off in my head that rang "component". So, I created the component presented in this article. A component I am sure I can reuse in other projects.

The code

The code for `THTMLCalendarGenerator` is in **Listings A and B**. `THTMLCalendarGenerator` is a non-visual component, derived from `TComponent`.

There are three published properties, `Month`, `Year` and `Title`. The `Month` and `Year` properties specify the month the calendar will depict and the `Title` property holds text that will be displayed at the top of the calendar.

There is one event, `OnGetEvent`, which allows the programmer using `THTMLCalendarGenerator` to add text to specific days of the calendar. I'll explain how that works later when I discuss the demo program. For now, don't be confused by the *Event* part of `OnGetEvent`. This means a calendar event,

something that occurs on a specific date. It doesn't mean an event in the C++Builder sense of the word.

Once the HTML code has been generated, it can be saved to a stream or file using the `SaveToStream()` and `SaveToFile()` methods. Or, the code can be retrieved as an `AnsiString` through the public property, `HTML`.

The actual generation of the code occurs when the protected method `Generate()` is called. `THTMLCalendarGenerator` generates the HTML code in a *lazy* manner. This means that the code is not generated until it is needed. This is a technique that can be very efficient. You can see how this works by looking at the methods `SetMonth()` and `SaveToFile()`.

The `SetMonth()` method is called when the `Month` property is assigned a value. The private data element `month` is assigned the value and the Boolean `needsGenerating` is set to `true`. However, the HTML code is not generated at this point.

When the `SaveToFile()` method is called, it first checks the value `needsGenerating`. If this value is `true`, then the `Generate()` method is called. If the value is `false`, the HTML code is up-to-date and does not need to be generated. Finally, `SaveToFile()` saves the HTML code to a file.

As you can see, the `Generate()` method sets `needsGenerating` to `false`. So, if nothing changes, a second call to `SaveToFile()` will not regenerate the HTML code.

Lets take a look at how the `Generate()` method works. This method uses the VCL component, `TStringList`. `TStringList`, an incredibly useful component, will provide the functionality for storing the HTML code and then writing that code out to a stream or file.

The calendar is simply an HTML table. The HTML code for this table and the rest of the HTML page is added to the `TStringList` on line at a time. Each time a new cell is added for a day of the month, the `OnGetEvent` event is called and any text returned by the event is added to the cell for that day.

The demo

The code for the `HTMLCalendarGenerator` component and the demonstration program can be found on the Bridges Publishing Web site.

Let's take a look at how the demo program uses the `HTMLCalendarGenerator` component.

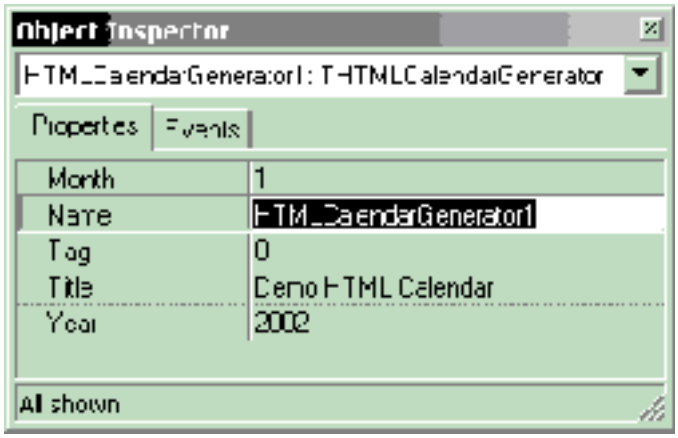
Figure A shows the demo program in action. This program allows you to choose a month and year for the calendar and then a click of the *Generate…* button lets you pick a file name and saves the HTML code to that file.

Figure A

The demonstration program.

In the IDE, I added an `HTMLCalendarGenerator` component to the form and filled in the properties for the component as seen in **Figure B**.

Figure B



The properties for THTMLCalendarGenerator.

I filled in the form's month and year combo boxes and set the initial values in the combo boxes in the form's constructor with this code:

```
// Fill in month combo box
for (int i = 0; i < 12; i++)
  MonthCombo->Items->Add(
    LongMonthNames[i]);

// Fill in year combo box
for (int i = 0; i < NUM_YEARS; i++)
  YearCombo->Items->Add(START_YEAR + i);

MonthCombo->ItemIndex =
  HTMLCalendarGenerator1->Month - 1;
YearCombo->ItemIndex =
```

```
  HTMLCalendarGenerator1->Year –
    START_YEAR;
```

Notice that I used the VCL array `LongMonthNames[]`, so that the names of the months will be locale specific. The variable `START_YEAR` is a `const int` that I set to `1999`.

I tied the `Month` and `Year` properties of the `HTMLCalendarGenerator` component to the form's combo boxes with this code:

```
void __fastcall TMainForm
  ::MonthComboChange(TObject *Sender)
{
  HTMLCalendarGenerator1->Month
    = MonthCombo->ItemIndex + 1;
}

void __fastcall TMainForm
  ::YearComboChange(TObject *Sender)
{
  HTMLCalendarGenerator1->Year
    = YearCombo->ItemIndex + START_YEAR;
}
```

Finally, I assigned the following function to the `HTMLCalendarGenerator` component's `OnGetEvent` event:

```
void __fastcall TMainForm
  ::HTMLCalendarGenerator1GetEvent(
  TObject *Sender, TDateTime date,
  AnsiString &desc)
{
  unsigned short y, m, d;
  date.DecodeDate(&y, &m, &d);

  if (d == 1)
    desc = "The first day of the month";
  else if (d == 5)
    desc = "To subscribe <a href="
    "'http://www.bridgespublishing.com'>"
    "click here.</a>";
  else if (d == 13)
    desc =
      "This event<br>has two lines";
```

```
  else if (d == 22)
    desc = "Some <b>bold</b> text";
  else if (d == 31)
    desc = "This month has 31 days.";
}
```

This event is fired each time a day is added to the calendar. The day's date and a reference to an `AnsiString`, `desc`, are passed to the event's method. I test the day of the month and for certain days and I assign some text to `desc` for those days. Notice that some of the text I assign to `desc`, contains embedded HTML. You can use this to really spice up the calendar. In fact, you can even add graphics to the calendar this way. To add a picture of a flag to calendar for the Fourth of July you add this code to the functions above:

```
else if (date == TDateTime(2002, 7, 4))
  desc = "Happy Fourth of July"
  + "<img src='../Pics/Flag.jpg'>";
```

The assignment

I find the `HTMLCalendarGenerator` component to be very useful as it's written. There are a few modifications you might want to consider, however. You might want to add properties to control the colors and the fonts used in the calendar. You might also want the component to generate the HTML table only and not the rest of the code for a complete HTML page.

If you are really ambitious, you could modify the component to generate calendars that cover ranges other than a single month. In any event, you have to admit that this component is both easy and useful.

Listing A: *Definition of THTMLCalendarGenerator.*

```
#ifndef HTMLCalendarGeneratorH
#define HTMLCalendarGeneratorH

#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>

typedef void __fastcall
  (__closure TGetCalendarEvent) (TObject* Sendr,
    TDateTime date, String &desc);

class PACKAGE THTMLCalendarGenerator
```

```
  : public TComponent
{
  __published:
    __property int Month =
      {read = month, write = SetMonth};
    __property String Title =
      {read = title, write = SetTitle};
    __property int Year =
      {read = year, write = SetYear};

    __property TGetCalendarEvent OnGetEvent =
      {read = onGetEvent, write = onGetEvent};

  public:
    __fastcall THTMLCalendarGenerator(
      TComponent* Owner);
    virtual __fastcall ~THTMLCalendarGenerator();

    virtual void __fastcall SaveToFile(
      const String fileName);
      virtual void __fastcall
    SaveToStream(TStream *stream);

    __property String HTML = {read = GetHTML};

  protected:
    virtual void __fastcall Generate();

  private:
    String __fastcall GetHTML();
    void __fastcall SetMonth(int m);
    void __fastcall SetTitle(String t);
    void __fastcall SetYear(int y);

    int month, year;
    TGetCalendarEvent onGetEvent;
    TStringList *output;
    bool needsGenerating;
    String title;
};


#endif   // HTMLCalendarGeneratorH
```

Listing B: *Implementation of THTMLCalendarGenerator.*

```
#include <vcl.h>
#pragma hdrstop

#include "HTMLCalendarGenerator.h"
#pragma package(smart_init)


__fastcall THTMLCalendarGenerator
  ::THTMLCalendarGenerator(TComponent*
  Owner) : TComponent(Owner)
{
  // Initially set date to next month
  unsigned short y, m, d;
  TDateTime date = IncMonth(Date(), 1);
  date.DecodeDate(&y, &m, &d);
  month = m;
  year = y;
  needsGenerating = true;
  onGetEvent = 0;
  output = new TStringList;
}

__fastcall THTMLCalendarGenerator
  ::~THTMLCalendarGenerator()
{
  delete output;
}

void __fastcall THTMLCalendarGenerator
  ::SaveToFile(const String fileName)
{
  if (needsGenerating) Generate();
    output->SaveToFile(fileName);
}

void __fastcall THTMLCalendarGenerator
  ::SaveToStream(TStream *stream)
{
  if (needsGenerating) Generate();

  output->SaveToStream(stream);
}
```

```
void __fastcall THTMLCalendarGenerator
  ::Generate()
{
  output->Clear();

  TDateTime startDate(year, month, 1);
  TDateTime endDate = IncMonth(startDate, 1) - 1;

  int days = endDate - startDate + 1;
  int startCol = startDate.DayOfWeek() - 1;
  int rowCount = (startCol + days + 6) / 7;
  int cellCount = rowCount * 7;

  // Begin HTML -------------------------

  output->Add("<html>");

  // HTML Header ------------------------

  output->Add("<head>");
  output->Add("<meta http-equiv="
    "'Content-Type' content="
    "'text/html'></meta>");
  output->Add("<meta name='GENERATOR' "
    "content='HTML Calendar Generator'>"
    "</meta>");
  output->Add("<title>" + title
    + FormatDateTime(" - mmmm yyyy",
        startDate) + "</title>");
  output->Add("</head>");

  // HTML Body --------------------------

  output->Add("<body>");

  // Calendar Title ---------------------

  output->Add("<table border='1' "
    "width='100%' cellspacing='0' "
    "cellpadding='5'>");
  output->Add("<tr>");
  output->Add("<td height='80' "
    "colspan='7' valign='bottom' "
    "align='center'><h2>"
```

```
  + FormatDateTime("mmmm yyyy",
  startDate) + "</h2>");
output->Add("<h3>" + title
  + "</h3></td>");
output->Add("</tr>");

// Day of Week Titles -----------------

output->Add("<tr>");
for (int i = 0; i < 7; i++)
  output->Add("<td width='14%' "
   "align='center' valign='middle'>"
   "<font size='2'>"
   + LongDayNames[i]
   + "</font></td>");
output->Add("</tr>");

// Fill in Calendar -------------------

TDateTime date = startDate;
for (int i = 0; i < cellCount; i++)
{
  if (i % 7 == 0) output->Add("<tr>");

  if (i >= startCol && i < startCol + days)
  {
    output->Add("<td valign='top' align='left'>");
    output->Add("<font size='2'>"
      + FormatDateTime("d", date)
      + "</font><p>"
       "<font size='1'>");

    String event = " ";
    if (onGetEvent)
      onGetEvent(this, date, event);
    output->Add(event);

    output->Add("</font></td>");

    date++;
  }
  else
    output->Add("<td valign='top' "
      "align='left' "
```

```
        "bgcolor='#EEEEEE'>"
        "<font size='1'> </font>"
        "</td>");

    if ((i + 1) % 7 == 0)
      output->Add("</tr>");
  }

  // End Calendar Table -----------------

  output->Add("</table>");

  // End HTML Body ----------------------

  output->Add("</body>");

  // End HTML ---------------------------

  output->Add("</html>");

  needsGenerating = false;
}

String __fastcall
THTMLCalendarGenerator::GetHTML()
{
  if (needsGenerating)
    Generate();
  return(output->Text);
}

void __fastcall THTMLCalendarGenerator
  ::SetMonth(int m)
{
  if (m == month) return;

  if (m < 1 || m > 12)
  {
    String msg =
      String(m) + " is not a valid month.";
   throw EConvertError(msg);
  }

  month = m;
  needsGenerating = true;
```

```
}

void __fastcall THTMLCalendarGenerator
  ::SetTitle(String t)
{
  if (title == t) return;

  title = t;
  needsGenerating = true;
}

void __fastcall THTMLCalendarGenerator
  ::SetYear(int y)
{
  if (y == year)
     return;

  if (y < 1900 || y > 3000)
  {
    String msg =
      String(y) + " is not a valid year.";
    throw EConvertError(msg);
  }

  year = y;
  needsGenerating = true;
}


// Component stuff ------------------------

namespace Htmlcalendargenerator
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] =
    {__classid(THTMLCalendarGenerator)};
    RegisterComponents("BCBJournal", classes, 0);
  }
}
```

An introduction to image color management

by Damon Chandler

Displaying an image on the screen is a no-brainer; you simply drop a `TImage` component on your form, and then load the desired image. Displaying an image on the screen in its true colors however, is an entirely different story, and perhaps an altogether impossible task. An “optimal” rendition lies somewhere between these two extremes. In this article, I’ll show you how to use Microsoft’s Image Color Management (ICM) 2.0 technology to color-match a bitmap for optimal display on a monitor. ICM 2.0 is available in Windows 98, Me, 2000, and XP.

Color management systems

Colors can be specified using an arbitrarily large number of bits. But because color is a perceived phenomenon, there’s a psychophysical limit as to just how many bits are really needed. Despite this limitation of our visual system, producing accurate colors on a computer monitor has proved a difficult task.

The problem with RGB monitors is that there’s no standard definition of red, green, and blue. One monitor might represent blue using light in the 470-nm range, whereas another monitor might use light in the 460-nm range. It’s inconsistencies like this (and lack of calibration) that lead to the vast majority of color variations between monitors. A *color management system* (CMS) is designed to compensate for these types of variations. And, indeed, most operating systems provide an integrated color management system: Macs use a CMS called ColorSync; Solaris machines use a CMS called Kodak Color Management System (KCMS); Windows machines use a CMS called Image Color Management (ICM).

Image Color Management 2.0

In its early days, ICM was called Image Color Matching. Microsoft later changed this to Image Color Management, perhaps to emphasize ICM’s role as an all-inclusive color management solution. (The latest version of ICM, version 2.0, was introduced in Windows 98). Regardless of its name, most of ICM’s value lies in its ability to color-match an image. (I’ll focus exclusively on color matching in this article.)

Color matching an image

What exactly does it mean to color-match an image? Well, in a nutshell, it means to adjust the colors of

an image so that when the image is displayed on an end-user's monitor, it looks the same (color-wise) as it did on the designer's monitor. Obviously, the success of this task is limited by the capabilities of end-user's display—a full-color image on a grayscale monitor will never look the same as if it were displayed on a color monitor. Still, within the limits of the display device, ICM does a pretty good job of matching the colors.

In order to color-match an image, ICM needs to know three things: (1) the color characteristics of the target device (i.e., the device on which the image will be displayed); (2) the color characteristics of the source device (i.e., the device on which the image was created or the device with which the image was captured); and (3) a rendering intent (more on this term shortly).

The color characteristics of the display device are specified via an International Color Consortium (ICC) device profile, which is often supplied as an .ICM file with a monitor's driver. The color characteristics of the source device are often embedded within the image's file; otherwise, the image is assumed to have been created or captured on a Standard RGB device (sRGB). See **www.srgb.com** for more information.

The rendering intent simply tells ICM what it should do about those colors that fall outside the color gamut of the target device (i.e., those colors that the target device can't reproduce). You can choose one of four rendering intents: *perceptual*, *absolute colorimetric*, *relative colorimetric*, or *saturation*. ICM will adjust its gamut-mapping algorithm, based on which rendering intent you choose. For natural images, the perceptual rendering intent produces the best-looking results. The others were designed for proofing (relative and absolute colorimetric) and conveying textual or graphical information (saturation).

Using ICM 2.0

As it turns out, using ICM to display a color-matched bitmap is fairly simple. Here's a list of the steps that are required:

1.Create a color profile object.

2.Initialize a `LOGCOLORSPACE` structure.

3.Create a color transform object.

4.Color-match, then display the bitmap.

5.Clean up.

Recall that ICM needs to know three main things in order to color-match a bitmap: the color characteristics of the target device, the color characteristics of the source device, and the rendering intent. The color characteristics of the target device are specified by using a color profile object (Step 1). The

color characteristics of the source device (and the rendering intent) are specified by using a `LOGCOLORSPACE` structure (Step 2). These two ingredients are used in Step 3 to create a color transform object, which, in turn, is used in Step 4 to color-match the bitmap. After the bitmap is color-matched, you simply display it as you would normally (e.g., via `TCanvas::Draw()`).

Step 1: Create a color profile object

A color profile object is simply a block of memory that holds the target device's ICC profile; ICM uses this profile to determine the color characteristics of the device on which the bitmap will be displayed. To create a color profile object, you use the `OpenColorProfile()` function, which is declared as follows:

```
HPROFILE OpenColorProfile(
  IN PPROFILE pProfile,
  IN DWORD dwDesiredAccess,
  IN DWORD dwShareMode,
  IN DWORD dwCreationMode
);
```

The `pProfile` parameter specifies a pointer to a `PROFILE` structure, which I'll discuss next. The other three parameters specify how the device profile file should be accessed. These are equivalent to `dwDesiredAccess`, `dwShareMode`, and `dwCreationDistribution` parameters of the `CreateFile()` API function (see **http://msdn.microsoft.com/library/en-us/fileio/filesio_7wmd.asp**).

As I just mentioned, the first parameter to the `OpenColorProfile()` function is a pointer to a `PROFILE` structure. This structure describes the location of the target profile (on disk or in memory). Here's how it's defined:

```
typedef struct tagPROFILE {
  DWORD dwType;
  PVOID pProfileData;
  DWORD cbDataSize;
} PROFILE;
```

The `dwType` data member indicates whether the profile data is located on disk (`PROFILE_FILENAME`) or in memory (`PROFILE_MEMBUFFER`). When `dwType` is set to `PROFILE_FILENAME`, the `pProfileData` data member specifies the name of the profile's file. When `dwType` is set to `PROFILE_MEMBUFFER`, the `pProfileData` data member specifies a pointer to the memory buffer that contains the profile data. Accordingly, the `cbDataSize` parameter specifies the length (in bytes) of either the file-name or the memory buffer.

The following snippet demonstrates how to create a color profile object for the primary monitor's current device profile:

```
// First, get the file-name of the
// monitor's default profile:
//
const HDC hScreenDC = GetDC(0);
DWORD num_chars = MAX_PATH;
TCHAR monitor_profilename[MAX_PATH];
const bool got_profile =
  GetICMProfile(
    hScreenDC, &num_chars,
    monitor_profilename);
ReleaseDC(0, hScreenDC);
if (!got_profile)
{
  throw EWin32Error("no profile");
}

// Next, initialize a PROFILE structure:
//
PROFILE monitor_profile;
monitor_profile.dwType =
  PROFILE_FILENAME;
monitor_profile.pProfileData =
  static_cast<void*>
    (monitor_profilename);
monitor_profile.cbDataSize =
  // "+ 1" for the NULL-termination
  (lstrlen(monitor_profilename) + 1) *
  sizeof(TCHAR);

// Then, create the color profile object:
//
const HPROFILE hMonitorProfile =
  OpenColorProfile(
    &monitor_profile, PROFILE_READ,
    FILE_SHARE_READ, OPEN_EXISTING);
if (hMonitorProfile == 0)
{
  throw EWin32Error(
    "OpenColorProfile() failed");
}
```

Notice that this code uses the `GetICMProfile()` function. This function simply retrieves the fully qualified file-name of the monitor's device profile (specified in Windows via the "Color Management" property page in the "Display Properties" dialog).

The `OpenColorProfile()` function will return either a handle to the color profile object or zero (if the function fails). Once you have a handle to the color profile object, you need to verify that the profile is valid. You do this by using the `IsColorProfileValid()` function, like so:

```
// Validate the contents of the profile
BOOL is_monitor_valid = FALSE;
IsColorProfileValid(
  hMonitorProfile, &is_monitor_valid);
if (!is_monitor_valid)
{
  throw EWin32Error("bad profile");
}
```

Step 2: Initialize a LOGCOLORSPACE structure

After the color profile object is created, the next step is to declare and initialize a `LOGCOLORSPACE`-type variable. This structure is used to specify the desired rendering intent and the color characteristics of the device on which the image was created (or with which the image was captured). Here's how the `LOGCOLORSPACE` structure is defined:

```
typedef struct tagLOGCOLORSPACE {
  DWORD lcsSignature;
  DWORD lcsVersion;
  DWORD lcsSize;
  LCSCSTYPE lcsCSType;
  LCSGAMUTMATCH lcsIntent;
  CIEXYZTRIPLE lcsEndpoints;
  DWORD lcsGammaRed;
  DWORD lcsGammaGreen;
  DWORD lcsGammaBlue;
  CHAR lcsFilename[MAX_PATH];
} LOGCOLORSPACE, *LPLOGCOLORSPACE;
```

The first three data members specify the structure's signature, version, and size—set them to `LCS_SIGNATURE`, `0x400`, and `sizeof(LOGCOLORSPACE)`, respectively. The `lcsCSType` data member specifies the source color space—set it to `LCS_sRGB` (in this article, we'll assume that the source image was created in the sRGB color space). The `lcsIntent` data member specifies rendering intent—set this data member to `LCS_GM_IMAGES` (to specify the perceptual rendering intent). The

remaining data member can all be zeroed-out. Here's an example:

```
// Initialize a LOGCOLORSPACE structure
// for the source image using the sRGB
// color space and the perceptual
// rendering intent (usually this
// information will be stored in the
// image file, but that's beyond the
// scope of this article; ICM info is
// common in PNG files but seldom used
// with bitmaps; sRGB is the best guess):
//
LOGCOLORSPACE lcs = {
  LCS_SIGNATURE, 0x400,
  sizeof(LOGCOLORSPACE)
  };
// sRGB color space
lcs.lcsCSType = LCS_sRGB;
// perceptual rendering intent
lcs.lcsIntent = LCS_GM_IMAGES;
```

Step 3: Create a color transform object

The next step is to create a color transform object by using the color profile object (from Step 1) and the `LOGCOLORSPACE` structure (from Step 2). The `CreateColorTransform()` function creates the color transform object:

```
HTRANSFORM CreateColorTransform(
  LPLOGCOLORSPACE pLogColorSpace,
  HPROFILE hDestProfile,
  HPROFILE hTargetProfile,
  DWORD dwFlags
);
```

The `pLogColorSpace` parameter specifies a pointer to the source device's corresponding `LOGCOLORSPACE` structure (which was initialized in Step 2). The `hDestProfile` parameter specifies a handle to the target device's corresponding color profile object (created in Step 1). The `hTargetProfile` parameter is used only for proofing (e.g., viewing the printer's output on the monitor)—set this parameter to `NULL`. The `dwFlags` parameter specifies the accuracy of the gamut-mapping algorithm—pass `NORMAL_MODE+ENABLE_GAMUT_CHECKING` as this parameter. Here's an example:

```
// Create a color transform object:
//
const HTRANSFORM hColorTransform =
  CreateColorTransform(&lcs,
    hMonitorProfile, NULL, NORMAL_MODE +
    ENABLE_GAMUT_CHECKING);
if (hColorTransform == 0)
{
  throw EWin32Error(
    "CreateColorTransform() failed");
}
```

If it's successful, the `CreateColorTransform()` function will return a handle to the newly created color transform object; otherwise, the function returns zero.

Step 4: Color-match the bitmap

Now that the color transform object is created, it's time to do the actual color matching; this step is performed by using the `TranslateBitmapBits()` function:

```
BOOL WINAPI TranslateBitmapBits(
  HTRANSFORM hColorTransform,
  PVOID pSrcBits,
  BMFORMAT bmInput,
  DWORD dwWidth,
  DWORD dwHeight,
  DWORD dwInputStride,
  PVOID pDestBits,
  BMFORMAT bmOutput,
  DWORD dwOutputStride,
  PBMCALLBACKFN pfnCallback,
  ULONG ulCallbackData
);
```

The first parameter is easy enough; it specifies a handle to a color transform object (which was created in Step 3).

The `pSrcBits` and `pDestBits` parameters specify, respectively, pointers to the source ("to be color-matched") and destination (color-matched) pixel arrays. In most cases, you'll set both of these parameters to the same value (e.g., the address of the first scan line of your bitmap). This instructs the `TranslateBitmapBits()` function to perform in-place color matching.

The `bmInput` and `bmOuput` parameters specify the format of the pixels (e.g., color-channel ordering and color-depth). You specify `BM_x555RGB` if your source image is a 16-bit bitmap, `BM_RGBTRIPLETS` if your source image is a 24-bit bitmap, or `BM_xRGBQUADS` if your source image is a 32-bit bitmap. (If you're color-matching an 8-bit bitmap, you also specify `BM_xRGBQUADS`, but instead of color-matching the bitmap's pixels, you color-match its color table.)

The `dwWidth` and `dwHeight` parameters specify the horizontal and vertical dimensions of the bitmap. In addition, because the scan lines of Windows bitmaps are usually aligned on 32-bit boundaries, the `dwInputStride` and `dwOutputStride` parameters specify the number of bytes per scan line in the source and destination pixel arrays.

The `pfnCallback` and `ulCallbackData` parameters can be used to specify a progress-indicating callback function. You can set both of these parameters to zero.

Here's an example of color-matching a bitmap that's contained in a `TImage` object (`Image1`):

```
// Color-match the bitmap...
//
// (1) grab a reference to the TBitmap
Graphics::TBitmap& Bitmap =
  *Image1->Picture->Bitmap;
Bitmap.PixelFormat = pf24bit;

// (2) grab a pointer to the pixels
// (specified in bmp.bmBits)
BITMAP bmp;
GetObject(Bitmap.Handle,
  sizeof(BITMAP), &bmp);

// (3) color-match the bitmap
const bool match_ok =
  TranslateBitmapBits(
    hColorTransform,
    bmp.bmBits, BM_RGBTRIPLETS,
    bmp.bmWidth, bmp.bmHeight, NULL,
    bmp.bmBits, BM_RGBTRIPLETS, NULL,
    NULL, NULL);
if (match_ok)
{
  Image1->Refresh();
}
```

At this point, within the capabilities of monitor, the bitmap will be displayed in its "true" colors. All

that’s left to do now is destroy the color profile and color transform objects.

Clean up

To destroy the color transform object (from Step 3), you use the `DeleteColorTransform()` function; this function takes a single parameter—a handle to the color transform object:

```
// destroy the color transform object
DeleteColorTransform(hColorTransform);
```

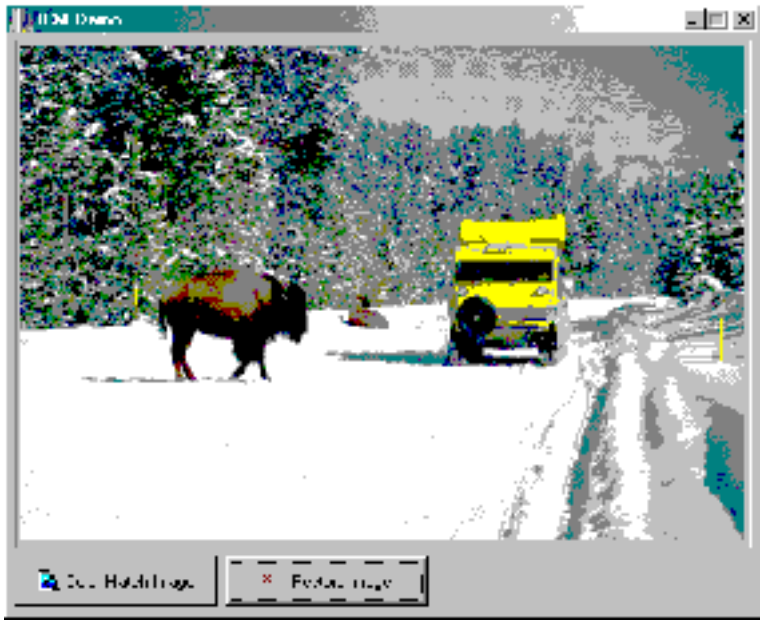
The color profile object of Step 1 is destroyed in a similar fashion by using the `CloseColorProfile()` function:

```
// destroy the color profile object
CloseColorProfile(hMonitorProfile);
```

A simple ICM-aware image viewer

Let’s work through an example of adding basic ICM support to a VCL application. Specifically, let’s create the application depicted in **Figure A**. The declaration of the `TForm1` class for this example is provided in **Listing A**.

Figure A



A bare bones ICM-aware VCL application.

Before you can use the various ICM functions, you need to include the header file `ICM.H`; you need to link with the static library `MSCMS.LIB`; and you need to make a few constant redefinitions. Thus, here's the code that goes at the top of the source file:

```
#include <vcl.h>
#pragma hdrstop

#if defined(__BORLANDC__)
  #undef LCS_SIGNATURE
  #undef LCS_sRGB
  #define LCS_SIGNATURE 'COSP'
  #define LCS_sRGB      'BGRs'
#endif

#include <icm.h>
#pragma link "mscms.lib"
```

And, here's the code that goes in the class constructor:

```
__fastcall TForm1::TForm1(
  TComponent* Owner): TForm(Owner),
  OrigBitmap_(new Graphics::TBitmap())
{
  OrigBitmap_->Assign(
    Image1->Picture->Bitmap);
}
```

`OrigBitmap_` holds a copy of the original, un-color-matched bitmap; it's used when `RestoreButton` is clicked:

```
void __fastcall TForm1::
  RestoreButtonClick(TObject *Sender)
{
  Image1->Picture->Bitmap->Assign(
    OrigBitmap_.get());
  ShowMessage("Bitmap restored.");
}
```

The meat of the application is the `ColorMatchButtonClick()` method, which demonstrates the five core steps that I presented earlier:

```
void __fastcall TForm1::
  ColorMatchButtonClick(TObject *Sender)
```

```
{
  // get the file-name of the
  // monitor's default profile
  TCHAR monitor_profilename[MAX_PATH];
  const HDC hScreenDC = GetDC(NULL);
  DWORD num_chars = MAX_PATH;
  const bool got_profile =
    GetICMProfile(
      hScreenDC, &num_chars,
      monitor_profilename);
  ReleaseDC(NULL, hScreenDC);
  if (!got_profile)
  {
    throw EWin32Error("no profile");
  }

  // initialize a PROFILE structure
  PROFILE monitor_profile;
  monitor_profile.dwType =
    PROFILE_FILENAME;
  monitor_profile.pProfileData =
    static_cast<void*>
      (monitor_profilename);
  monitor_profile.cbDataSize =
    (lstrlen(monitor_profilename) + 1) *
    sizeof(TCHAR);

  // create the color profile object
  HPROFILE hMonitorProfile =
    OpenColorProfile(
      &monitor_profile, PROFILE_READ,
      FILE_SHARE_READ, OPEN_EXISTING);
  if (hMonitorProfile == 0)
  {
    throw EWin32Error(
      "OpenColorProfile() failed");
  }
  try
  {
    // validate the profile's contents
    BOOL is_monitor_valid = FALSE;
    IsColorProfileValid(hMonitorProfile,
      &is_monitor_valid);
```

```
if (!is_monitor_valid)
{
  throw EWin32Error("bad profile");
}

// initialize a LOGCOLORSPACE struct
LOGCOLORSPACE lcs = {
  LCS_SIGNATURE, 0x400,
  sizeof(LOGCOLORSPACE)
  };
// sRGB color space
lcs.lcsCSType = LCS_sRGB;
// perceptual rendering intent
lcs.lcsIntent = LCS_GM_IMAGES;

// create a color transform object
const HTRANSFORM hColorTransform =
  CreateColorTransform(
   &lcs, hMonitorProfile, NULL,
   NORMAL_MODE+ENABLE_GAMUT_CHECKING
   );
if (hColorTransform == 0)
{
  throw EWin32Error(
    "CreateColorTransform failed");
}
try
{
  // color-match the bitmap...

  // grab a reference to
  // the TBitmap object
  Graphics::TBitmap& Bitmap =
    *Image1->Picture->Bitmap;
  Bitmap.PixelFormat = pf24bit;

  // grab a pointer to the pixels
  BITMAP bmp;
  GetObject(Bitmap.Handle,
    sizeof(BITMAP), &bmp);

  // color-match the bitmap's pixels
  const bool match_ok =
    TranslateBitmapBits(
```

```
          hColorTransform,
          bmp.bmBits, BM_RGBTRIPLETS,
          bmp.bmWidth, bmp.bmHeight,
          NULL, bmp.bmBits,
          BM_RGBTRIPLETS, NULL, NULL, 0
          );
      if (match_ok)
      {
        // repaint
        Image1->Refresh();
        ShowMessage(
          "Bitmap color-matched.");
      }
      else
      {
        throw EWin32Error(
          "TranslateBitmapBits failed");
      }
    }
    catch (...)
    {
      // destroy the color
      // transform object
      DeleteColorTransform(
        hColorTransform);
      throw;
    }
  }
  catch (...)
  {
     // destroy the color profile object
     CloseColorProfile(hMonitorProfile);
     throw;
  }
  // destroy the color profile object
  CloseColorProfile(hMonitorProfile);
}
```

That's all there is to displaying a color-matched bitmap. Although this example is relatively bare bones, my aim was to emphasize the simplicity of adding ICM support to a VCL application. Indeed, Steps 1, 2, and 3 (from the previous list) are key to any ICM application. (Note that this example won't work with C++Builder version 1.0 because it assumes a DIB-section-type `TBitmap` object; if you're using C++Builder 1.0, see the Platform SDK help files for the `GetDIBits()` and `SetDIBits()`

functions.)

Conclusion

In this article, I've presented a somewhat practical introduction to ICM 2.0. For more information on ICM, sRGB, and general color management, see ISBN 1576108767 and **http://www.microsoft.com/hwdev/color**. The source code for the example presented in this article can be downloaded at **www.bridgespublishing.com**.

Listing A: ***Declaration of the TForm1 class for a simple ICM-aware image viewer***

```
#include <memory>
class TForm1 : public TForm
{
__published:
  TImage *Image1;
  TBitBtn *ColorMatchButton;
  TBitBtn *RestoreButton;
  void __fastcall
    ColorMatchButtonClick(TObject *Sender);
  void __fastcall
    RestoreButtonClick(TObject *Sender);
private:
   std::auto_ptr<Graphics::TBitmap> OrigBitmap_;
public:
   __fastcall TForm1(TComponent* Owner);
};
```

Menu templates

by Mark G. Wiseman

There's a useful feature in the C++Builder IDE that you might not be aware of—menu templates. Menu templates are pre-designed menus that are included with C++Builder. With menu templates and a couple of mouse clicks, you can insert an entire menu structure into your application. You can use the templates as is, customize them, or even create your own templates.

To use the menu template features of the IDE, just bring up the Menu Designer and right-click on the designer form. (You can bring up the Menu Designer by double-clicking on a `TMainMenu` or `TPopupMenu` component.)

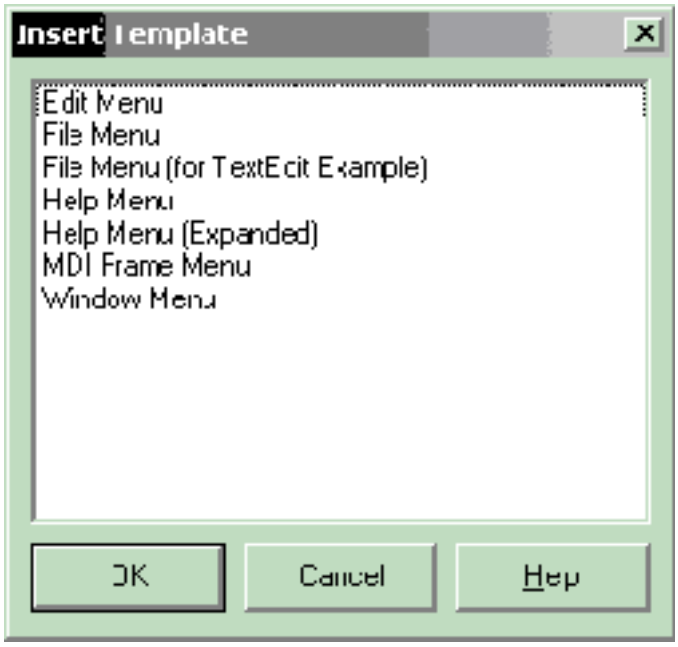
The context menu displayed when you right-click on the designer form contains menu items for saving a menu as a template, inserting a menu from a template and deleting templates. **Figure A** shows the context menu.

Figure A

The Menu Designer context menu.

If you select Insert From Template… in the menu an Insert Template dialog, similar to the one in **Figure B** will appear.

Figure B



The Insert Template dialog.

The best way to learn how the templates work is to experiment with them. Create a new application and drop a `TMainMenu` component on the form and start inserting menu templates to see what happens.

Also, clear all the items from the main menu, design a menu of your own, and save it as a template. If you have a certain menu design that you use regularly, saving that design as a template can save a lot of time in future projects.

You'll still need to write the code that executes when a menu item is clicked, but you don't have to design standard menus every time you create a new application.

Setting an Open dialog's current directory

by Damon Chandler

Recently, I demonstrated a method for adding VCL controls to an Open dialog box. In this article, I'll show you how to change the dialog's currently displayed folder. Together, these techniques allow you to leverage the standard Open dialog's existing navigation capabilities (see **Figure A**).

Figure A

An Open dialog box that provides a custom navigation bar.

Navigating to a specific folder

The Windows API provides no straightforward way of specifying the directory that's displayed in an Open dialog's explorer-pane list-view control. You might recall however, that if you type a path (either relative or absolute) into the dialog's "File name" edit control and then click Open, the dialog will navigate to the specified folder. You can perform this procedure programmatically via the following seven steps:

1.Grab a handle to the Open dialog.

2.Grab a handle to the "File name" edit control.

3.Retrieve a copy of the "File name" edit control's current text.

4.Change the "File name" edit control's text to the desired path.

5.Grab a handle to the “Open” button.

6.Send the `BM_CLICK` message to the “Open” button to incite a click.

7.Restore the “File name” edit control’s text.

The first step is simple—just pass the value of the `TOpenDialog::Handle` property to the `GetParent()` API function:

```
const HWND hOpenDlg =
  GetParent(OpenDialog->Handle);
```

Once you have a handle to the Open dialog, you can grab a handle to the “File name” edit control (Step 2), by using the `GetDlgItem()` API function like so:

```
const HWND hFileEdit =
  GetDlgItem(hOpenDlg, edt1);
```

At this point, you need to make a copy of the “File name” edit control’s current text (Step 3); you do this by using the `GetWindowText()` API function:

```
TCHAR old_text[MAX_PATH];
GetWindowText(
  hFileEdit, old_text, MAX_PATH);
```

Now it’s time to change the “File name” edit control’s text to desired path (Step 4); this is done via the `SetWindowText()` API function:

```
SetWindowText(
  hFileEdit, new_path_string);
```

To perform Steps 5 and 6 (i.e., to click the “Open” button), you use a combination of the `GetDlgItem()` API function (specifying `IDOK` as the second parameter) and the `SendMessage()` API function:

```
const HWND hOKButton =
  GetDlgItem(hOpenDlg, IDOK);
SNDMSG(hOKButton, BM_CLICK, 0, 0);
```

The last step is to restore the contents of the “File name” edit control; again, use the `SetWindowText()` API function:

```
SetWindowText(hFileEdit, old_text);
```

Although these seven steps are fairly painless to implement, there are a couple of problems: First, the observant user might notice the (albeit quick) change in the “File name” edit control’s text. Second, if there’s an item selected in the dialog’s explorer-pane list-view control, clicking Open will dismiss the dialog, regardless the text that’s specified in the “File name” edit control.

To work around the first problem, you can send the `WM_SETREDRAW` message (with a `false` `wParam` value) to the “File name” edit control just before you set its new text. This will prevent the edit control from redrawing itself after the new text is specified. To work around the second problem, you can use the `LVM_SETITEMSTATE` message to deselect all items in the explorer-pane list-view control before clicking Open.

The following function demonstrates how to implement all of the aforementioned steps and the two workarounds:

```
void NavigateToFolder(
  TOpenDialog& OpenDialog,
  AnsiString folder_name,
  int folder_csidl = 0)
{
  if (folder_csidl != 0)
  {
    // retrieve the path of the
    // specified special folder
    TCHAR specialfolder_name[MAX_PATH];
    SHGetSpecialFolderPath(
      OpenDialog.Handle,
      specialfolder_name,
      folder_csidl, false);
    folder_name = specialfolder_name;
  }

  if (folder_name.Length() != 0)
  {
    // grab a handle to the Open dialog
    const HWND hOpenDlg =
      GetParent(OpenDialog.Handle);

    // grab handle to the explorer-pane
    // list view's parent window
    const HWND hShellView =
      FindWindowEx(hOpenDlg, 0,
```

```
  "SHELLDLL_DefView", "");

// grab a handle to the
// explorer-pane list view
const HWND hListView = FindWindowEx(
   hShellView, 0, WC_LISTVIEW, "");
if (hListView)
{
  // deselect all items
  LV_ITEM lvi = {LVIF_STATE};
  lvi.stateMask = LVIS_SELECTED;
  // NOTE: because lvi.state = 0,
  // this code will deselect
  SNDMSG(
    hListView, LVM_SETITEMSTATE,
    -1, // all items
    reinterpret_cast<LPARAM>(&lvi));

  // grab a handle to the
  // "File name" edit control
  const HWND hFileEdit =
    GetDlgItem(hOpenDlg, edt1);

  // retrieve a copy of the "File
  // name" edit's current text
  TCHAR old_text[MAX_PATH];
  GetWindowText(
    hFileEdit, old_text, MAX_PATH);

  // change the "File name" edit's
  // text to the desired path
  SNDMSG(hFileEdit,
    WM_SETREDRAW, false, 0);
  SetWindowText(
    hFileEdit, folder_name.c_str());
  SNDMSG(hFileEdit,
    WM_SETREDRAW, true, 0);

  // click the "Open" button to
  // change to the target folder
  const HWND hOKButton =
    GetDlgItem(hOpenDlg, IDOK);
  SNDMSG(hOKButton, BM_CLICK, 0, 0);
```

```
      // restore the "File name"
      // edit's text
      SetWindowText(hFileEdit, old_text);
    }
  }
}
```

Conclusion

There’s one limitation of this approach that needs mentioning—namely, you can’t set the Open dialog’s current directory to a virtual folder such as “My Computer”; users will have to navigate to these types of folders manually.

I’ve put together a sample project that demonstrates how to use the `NavigateToFolder()` function. This project—available at **www.bridgespublishing.com**—uses the `TOpenDialogEx` class (presented in last month’s issue) and the `NavigateToFolder()` function to provide an interface similar to that depicted in **Figure A**.

Abstract data types in C++Builder

by Cris Gallegos

Many applications provide objects that their users can create, manipulate, store, and retrieve at run time. As the objects themselves become more complex, managing and utilizing them can become more difficult. For example, during a recent project I had to provide objects that contained, among other things, twenty-eight different Boolean variables. The users were allowed to manipulate these variables whenever they needed to.

The Boolean values were needed to import, export, interpret, format, sort, display, and output the data that was manipulated by the program. They had to be used constantly from within many long and time critical loops. They had to be referenced and manipulated from many different areas of the program in many different combinations. At first it seemed as if using them was going to be, to say the least, awkward.

One solution

Then I began experimenting with abstracting the format of the data in different ways. The advantages that this approach brought to that application became immediately apparent. As I developed the abstraction techniques further, it became obvious that I was really creating an Abstract Data Type (ADT). I found that my understanding of just what an ADT is was a little fuzzy. The phrase itself intrigued me so did a little research.

What I found was many different definitions for, supposedly, the same thing. One definition said that the phrase “abstract data type” is simply another name for any user-defined data type. The most agreed upon definition that I could find was expressed in many different ways by many different sources. It says, for the most part, that an ADT is any software construct that internalizes information and provides ways of accessing and/or changing that information. This definition seems to be the most historically correct. It will, therefore, be the one used for the rest of this article.

The benefits provided by the ADT concept are many. They include:

- The encapsulation (i.e. hiding) of data. This helps protect the validity of the encapsulated data.

- A public user interface for accessing and manipulating the encapsulated data. The interface allows for more correct usage of the data. It allows the exact nature of the encapsulated data to be isolated and hidden from the rest of the program. It also provides a way to make the data usage more self-documenting.

•The chance to move the use and interpretation of the encapsulated data out of the computer science world and back into the real world.

•The ability to eliminate the passage of raw data around inside a program. This makes it easier to transfer more complete, robust, and useful data around within a program.

Sounds familiar, doesn't it? The ADT concept predates most, if not all, of the modern object oriented programming languages. Given that, the similarities between the ADT concept and the object oriented class concept are obvious. While every C++ class is not an actual ADT, the ADT concept can be capably implemented, and indeed further modularized, from within a C++ class.

One of the more powerful services that an ADT can provide is made possible by its public user interface. Through this interface, the exact nature of data can be easily and transparently transformed. Data that is readily used by the external program can be sent to or returned from the ADT. The ADT can then, in turn, store and retrieve that data using whatever format is most appropriate given its operating requirements.

What does all this mean in the context of a working C++Builder application? For my application I was able to combine a set of bit masks, the C++ bitwise operators, and the `__property` keyword to create an ADT that made coping with the "Bountiful Bools" much easier. In general, the ADT framework provided for greater efficiency and flexibility, both in coding and execution, throughout the program. I was also able to create more powerful objects that were smaller in size.

The implementation

Once I began abstracting the data format of the Boolean variables, I started doing it with other types of data. Consider the `TGroceryList` ADT presented in **Listing A**. The code within this listing is available as `TGrocList.h` and is available at **www.bridgespublishing.com.** The `TGroceryList` object internalizes just two basic variables (an unsigned integer and a character array), yet provides the storage for and access to five different Boolean values, one small number, and an `AnsiString`. However, only one quarter of the integer is actually used!

Notice the way that the internal data structures are initialized in the default constructor. This simplicity is possible due to the fixed size of the object. The fixed size is also exploited by the equally simple `Assign()` method. This method is needed because the C++Builder compiler won't allow objects containing properties to be copied via direct assignment.

If having a direct assignment operator is important, you could eliminate the properties, demote the private member methods to public, and then use the default equality operator provided by the compiler. This should work just as well as using the `Assign()` method. I have found, however, that using properties can be more intuitive and elegant than using straight function calls.

Boolean variables

The storage of Boolean values is accomplished by using some of the C++ bitwise operators and some constant integers to flip certain bits within the `FBitStash` field. The same constants are used to detect the current value of the very same bits. The `NeedMilk` property, and its associated read and write methods, show how easily a single bit within the `FBitStash` field can be utilized. Changing or querying the status of this particular bit is as easy as writing the following code:

```
TGroceryList * pGrocList =
  new TGroceryList;
pGrocList->NeedMilk = true;
CheckBox1=>Checked =
  pGrocList->NeedMilk;
```

The `GetItems()` and `GotItems()` methods allow for more flexible and powerful access to the `FBitStash` bits. These methods make it easy to query or change random, multiple bits in a single operation. The `NeedAnyGroceries()` method shows how a single query can determine whether any of the "Boolean" `FBitStash` bits are "true". Changing the status of multiple bits can be accomplished with a single statement:

```
PGrocList->GetItems(
  glBread | glFruit | glPBJ, true);
```

This statement changes the first, third, and fifth Least Significant Bits (LSBs) of `FBitStash` to ones, thereby making them "true". Making the same statement with the last parameter set equal to false will clear the bits, making them "false".

Number variables

Small numbers can be stored within the `FBitStash` field by using some of the other bitwise operators. To illustrate this point I've created the `NumOfLoaves` property. Its read and write methods use the shift-left and shift-right operators to store and retrieve an integer's value to or from the twenty-eighth, twenty-ninth, and thirtieth Most Significant Bits (MSBs) of `FBitStash`.

Notice that the write method, `SetNumOfLoaves()`, will accept any integer value but ensures that only valid numbers, ones with values between zero and eight, are stored. It then clears the appropriate "bit slots" before storing the new value into them. The read method, `GetNumOfLoaves()` simply reads the combinational value of the stored bits. It does this by assigning the value of `FBitStash` (after its bits have been shifted twice to the left) to the temporary `LoavesInt` variable. It then right shifts the bits of interest into the first, second, and third LSB positions and casts the result into an integer.

The proper operation of `GetNumOfLoaves()` requires that `FBitStash` be declared as an unsigned integer. When right shifting bits within a signed integer, the actual value of the original MSB is carried through every time the bits are shifted to the right. For example, if a signed integer starts with a one in the MSB position (1000b), then gets shifted to the right three times, it will wind up with all four MSBs set to one (1111b). Declaring `FBitStash` as unsigned ensures that any bits added to the MSB position will have a value of zero.

Overall, this technique is similar to using C++ bit fields; it packs more variables into a smaller space. There are a few important differences however. The compiler doesn't have to generate any extra code to manipulate the named fields. While this saves on compiler and run-time overhead, it means that only numbers with a smaller size (fewer number of bits) can be stored within a larger container. In general, this alleviates the need to worry about padding and data alignment issues; any unused bits within the containing field are all the padding that's needed. Also, the default data alignment of the C++Builder compiler is not guaranteed to remain constant between versions. The technique presented here should work flawlessly with all future versions of the compiler (assuming that the size of an integer doesn't decrease); code using C++ bit fields may or may not.

AnsiString variables

The last technique that I'll present deals with handling strings of characters. It involves storing and retrieving the strings using C++ character arrays but converting them to and from `AnsiStrings` as needed. If you look at the code for the `StoreName` property, you can see that its read and write methods are really pretty simple. `GetStoreName()` uses one of the standard `AnsiString` constructors to create and then return a new string containing a copy of the `FStoreName` character array. `SetStoreName()` uses `strncpy()` to safely store any new store name information within `FStoreName`.

If you're willing to assign arbitrary limits to the length of your object's strings and are willing to tolerate a small bit of wasted space, this technique takes the fixed length advantage of a C++ character array and couples it to the already powerful `AnsiString` class. It has become a favorite technique of mine for dealing with strings that need to be limited in length or permanently stored.

Storage

Speaking of storage, how do you store and retrieve the information contained within a `TGroceryList` instance? Storing it to a disk file can be as simple as the following example:

```
TFileStream* pFS = new TfileStream(...);
pFS->Write(
  pGrocList, sizeof(TGroceryList));
```

```
pFS->Read(
  pGrocList, sizeof(TGroceryList));
```

If you need to store the information to the Windows System Registry, the `TRegistry::WriteBinaryData()` and `TRegistry::ReadBinaryData()` methods could be used just as easily.

Conclusion

To accommodate a given situation, any fixed length, non-dynamic data type can be added to your ADT. You can add or subtract as many fixed size character arrays as needed while developing it. As long as the compiler can accurately determine the exact size of all of the data stored within your ADT when it compiles the code the default constructor, the `Assign()` method, and the storage code should work without further modification.

When you need to store and retrieve lots of `AnsiStrings`, it just doesn't get much easier than this.

Listing A: *The TGroceryList ADT code.*

```
const int STORE_NAME_LENGTH = 25;
const int glBread = 1, glMilk = 2,
  glFruit = 4, glCereal = 8, glPBJ = 16;

class TGroceryList
{
  private:
    unsigned int FBitStash;
    char FstoreName [STORE_NAME_LENGTH + 1];

    bool GotMilk()
    {
      return (FBitStash & glMilk) > 0;
    }

    void GetMilk(bool NewValue)
    {
      NewValue ? FBitStash |= glMilk :
        FBitStash &= ~glMilk;
    }

    int GetNumOfLoaves()
    {
```

```
    unsigned int LoavesInt = (FBitStash << 2);
    return static_cast<int>(LoavesInt >> 29);
  }

  void SetNumOfLoaves(int NewValue)
  {
    if (NewValue >= 0 && NewValue <= 7) {
      FBitStash &= 0xC7FFFFFF;
      //Clear the placeholders.
      FBitStash |= (NewValue << 27);
      //Set the new value.
    }
    else
      ShowMessage("Get a life dude.");
  }

  String GetStoreName()
  {
    return ::AnsiString(FStoreName);
  }

  void SetStoreName(String NewName)
  {
    strncpy(FStoreName,
      NewName.c_str(), STORE_NAME_LENGTH);
  }

public:
  TGroceryList()
  {
    memset(this, 0, sizeof(TGroceryList));
  }

  void Assign(TGroceryList * pGL)
  {
    if (pGL == NULL)
      ShowMessage("Get a valid pointer.");
    else if(this == pGL)
      ShowMessage("Get a life.");
    else
      memcpy(this, pGL, sizeof(TGroceryList));
  }

  bool __property NeedMilk =
```

```
    {read=GotMilk, write=GetMilk};
  bool GotItems(int BitIndex)
    {return (FBitStash & BitIndex) > 0;}

  void GetItems(int BitIndex, bool NewValue) {
    NewValue ? FBitStash |= BitIndex :
      FBitStash &= ~BitIndex;}

  bool NeedAnyGroceries()
  {
    return  GotItems(glBread | glMilk |
      glFruit | glCereal | glPBJ);
  }

  int __property NumOfLoaves =
    {read=GetNumOfLoaves, write=SetNumOfLoaves};
  String __property StoreName =
    {read=GetStoreName, write=SetStoreName};
};
```

Custom Open dialogs, part II

by Damon Chandler

Last month, I showed you how to place a `TWinControl`-type child window to the right side of an Open dialog's standard controls. This month, I'll demonstrate how to use an in-memory dialog box template to move the standard controls. This way you can place multiple `TWinControl`-type child windows anywhere within the dialog.

Using a dialog template with the standard Open dialog

Recall from last month's article that an Open dialog box contains a child dialog designed to host additional, user-defined controls. This child dialog has a default width of zero, which was preferable last time because we didn't use it (i.e., we placed the additional controls directly on the Open dialog box itself, not on its child dialog). That technique was fine for adding controls below or to the right of the Open dialog's standard controls, but it won't work if you need to place additional controls above or to the left of the standard controls. Why? Well, last month, when we placed our `TWinControl`-type window to the right of the Open dialog's standard controls, we didn't have to worry about the location of these standard controls—they weren't in the way. Before you can place additional controls above or to the left of the standard controls however, you'll need to move the standard controls to make room for the new controls.

There are two ways to move the Open dialog's standard controls:

- Use a dialog box template.

- Grab a handle to each control, and then use the `MoveWindow()` or `SetWindowPos()` API function.

As it turns out, the second approach doesn't work very well, particularly because the Open dialog's list view control (i.e., the view that displays the files and folders) is re-created each time the user makes a folder change.

Specifying the dimensions of the Open dialog box

As I mentioned last time, you can use a resource template to specify the size of the Open dialog's child dialog. The Open dialog will expand its own height by whatever value you specify as the child dialog's height. For example, here's a resource script that will expand the Open dialog's height by 100 vertical dialog units:

```
OPENDLGTEMPLATE DIALOG 0, 0, 80, 100
STYLE WS_CHILD | WS_CLIPSIBLINGS |
      DS_CONTROL
BEGIN
END
```

Here, I've specified 100 as the child dialog's height, which implicitly instructs the Open dialog to increase its own height by 100 vertical dialog units. In contrast, although I've specified 80 as the child dialog's width, the Open dialog box won't adjust its own width by this (or any) amount. This (somewhat odd) behavior is by design; namely, the Open dialog adjusts only its height because, by default, all additional user-defined controls are to be placed below the Open dialog's standard controls.

Note that, because this resource script defines the template for the child dialog (not the Open dialog), I've specified the `WS_CHILD` and `DS_CONTROL` styles. In addition, I've specified the `WS_CLIPSIBLINGS` style, which ensures that the child dialog won't occlude the Open dialog's standard controls.

To use this resource template, you simply save it to an RC file (e.g., MYOPENDIALOG.RC), add the file to your project, and then assign the child dialog's identifier (`OPENDLGTEMPLATE` in this example) to the `TOpenDialog::Template` property (in a descendant class), like so:

```
bool __fastcall TMyOpenDialog::Execute()
{
  Template = "OPENDLGTEMPLATE";
  return TOpenDialog::Execute();
}
```

Figure A depicts the resulting Open dialog box when this template is used.

Figure A

By using a dialog box template to define the Open dialog's child dialog, you can indirectly specify the height (but not the width) of the Open dialog box.

Specifying the location of the Open dialog's standard controls

Again, by using a resource template to specify the styles and the height of the child dialog, you can implicitly define the height of the Open dialog box. In order to adjust the width of the Open dialog box—and also move its standard controls—you'll need to place a special child window on the Open dialog's child dialog; specifically, you need to add a static control with the identifier `stc32` (defined as 1119 in DGLS.H). For example:

```
#include <dlgs.h>
OPENDLGTEMPLATE DIALOG 0, 0, 80, 100
STYLE WS_CHILD | WS_CLIPSIBLINGS |
      DS_CONTROL
BEGIN
  LTEXT "", stc32, 52, 48, 0, 0,
    NOT WS_VISIBLE | NOT WS_GROUP
END
```

The `LTEXT` statement creates a static control whose text is aligned to the left. Here, this static control is given an identifier of `stc32`, a horizontal position of 52, a vertical position of 48, and a width and height of zero. As depicted in **Figure B**, the Open dialog will interpret the horizontal and vertical position of this special static control as the desired location of the standard controls. The "stc32" static control is still created, but it's not positioned at the location specified in the template (52, 48 here).

Rather, the “stc32” static control is automatically positioned at the lower-right corner of the Open dialog’s standard controls. You can see this in **Figure C**, which depicts the resulting Open dialog box when the following resource script is used:

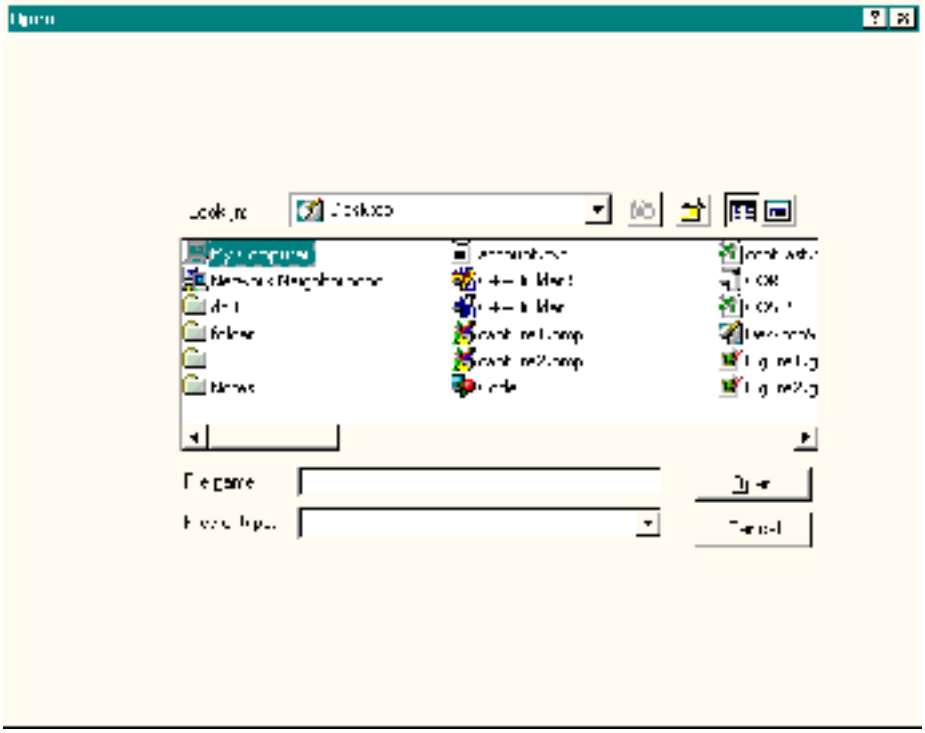
```
#include <dlgs.h>
OPENDLGTEMPLATE DIALOG 0, 0, 80, 100
STYLE WS_CHILD | WS_CLIPSIBLINGS |
      DS_CONTROL
BEGIN
  LTEXT "STC32", stc32, 52, 48, 0, 0
END
```

Notice from **Figures B** and **C** that creating the “stc32” static control not only allows you to specify the location of the Open dialog’s standard controls, it also instructs the Open dialog to increase its own width by the specified width of the child dialog (80 horizontal dialog units in these examples).

Keep in mind that the coordinates of the “stc32” static control are relative to the client area of the Open dialog’s child dialog, and that they are specified in dialog units (not pixels). To convert from pixels to dialog units, you can use the `GetDialogBaseUnits()` API function like so:

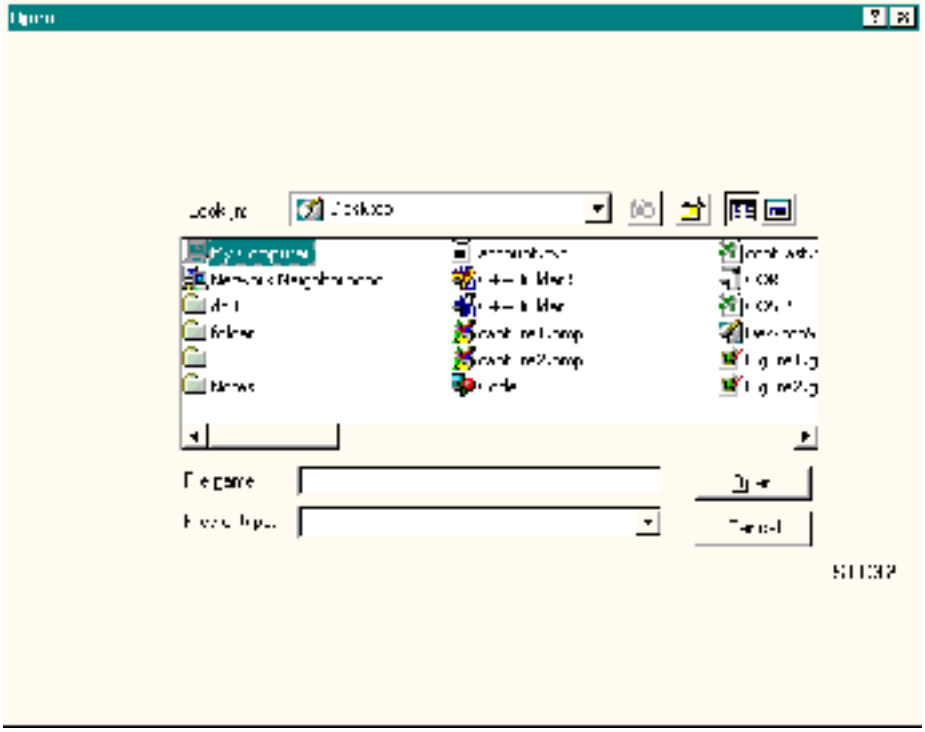
```
short Pix2DlgUnitsX(short x)
{
  return (x * 4.0) /
    LOWORD(GetDialogBaseUnits());
}

short Pix2DlgUnitsY(short y)
{
  return (y * 8.0) /
    HIWORD(GetDialogBaseUnits());
}
```

Figure B



By placing a static control with the identifier stc32 on the Open dialog's child dialog, you can specify the location of the Open dialog's standard controls.

Figure C



The "stc32" static control isn't positioned at the coordinates specified in the resource script; instead, it's placed at the bottom-left corner of the standard controls.

I've just shown you how to use a resource script to define the template that's used for the Open dialog's child dialog. Although creating the script isn't too hard, defining the correct size of the child dialog, and the correct location of the "stc32" static control, can require a bit of trial and error. Remember, the whole point of resizing the Open dialog box and moving its standard controls is to make room for the `TWinControl`-type VCL controls that will be added to the Open dialog. If you later change the size of one or more of these VCL controls, you'll have to edit the resource script to adjust the location of the static control and the size of the child dialog—more trial and error. As I'll discuss next, however, there's a way around this hassle.

Creating an in-memory dialog template

Instead of using a resource script to define the template that's used to create the Open dialog's child dialog, there is an alternative technique: you can use a template that's stored in global memory. The following function demonstrates how this is done; it creates an in-memory dialog box template with the dimensions specified by the `dlg_w` and `dlg_h` parameters and with an "stc32" static control at the location specified by the `stc_x` and `stc_y` parameters. Here's the code:

```
HGLOBAL CreateChildDlgTemplate(
   short dlg_w, short dlg_h,
   short stc_x, short stc_y,
   DWORD styles = 0,
   DWORD ex_styles = 0)
{
  // create a global memory object
  const HGLOBAL hTemplate =
    GlobalAlloc(GMEM_ZEROINIT, 384);

  // grab a pointer to the memory
  DLGTEMPLATE* pTemplate =
    static_cast<DLGTEMPLATE*>
      (GlobalLock(hTemplate));
  if (!pTemplate) return 0;

  // initialize the child dialog
  pTemplate->style =
    styles | WS_CHILD |
    WS_CLIPSIBLINGS | DS_CONTROL;
  pTemplate->dwExtendedStyle = ex_styles;
  pTemplate->cdit = 1; // 1 child control
  pTemplate->cx = Pix2DlgUnitsX(dlg_w);
  pTemplate->cy = Pix2DlgUnitsY(dlg_h);

  // grab a pointer to the data for
```

```
  // the dialog's menu, class, and
  // title properties
  WORD* pData = reinterpret_cast<WORD*>
    (pTemplate + 1);

  // set no menu, use default class,
  // and no title
  pData += 3;

  // grab a pointer to the data for
  // the "stc32" control properties
  DLGITEMTEMPLATE* pItem =
    reinterpret_cast<DLGITEMTEMPLATE*>
      (pData);
  pItem->style = WS_CHILD | SS_LEFT;
  pItem->id = stc32;
  pItem->x = Pix2DlgUnitsX(stc_x);
  pItem->y = Pix2DlgUnitsY(stc_y);

  // grab a ponter to the data for the
  // static's class and title properties
  pData =
    reinterpret_cast<WORD*>(pItem + 1);
  // set the STATIC class and no title
  pData[0] = 0xFFFF;
  pData[1] = 0x0082;
  pData += 3;

  // clean up
  GlobalUnlock(hTemplate);

  // return a handle to the memory
  return hTemplate;
}
```

This code first uses the `GlobalAlloc()` API function to create a global memory object of 384 bytes (large enough to hold our template data). It then grabs a pointer to the underlying memory (by using the `GlobalLock()` function) and fills it with the data of the template; these data include the following:

1.A `DLGTEMPLATE` structure that specifies the attributes of the dialog.

2.A variable-length array of identifiers for the dialog’s menu, class, and title.

3.One or more DLGITEMTEMPLATE structures that specify the attributes of the dialog's child controls; each DLGITEMTEMPLATE data is followed by a variable-length array of identifiers for the child control's class, title, and creation data.

This is a just brief overview of the layout of an in-memory template. You can read a full description in the section titled "Templates in Memory" in the Platform SDK help files or on MSDN online. (Pay particular attention to the DWORD-alignment requirements of the data structures.)

The dlg_w, dlg_h, stc_x, and stc_y parameters are specified in pixels. The CreateChildDlgTemplate() function uses the previously defined Pix2DlgUnitsX() and Pix2DlgUnitsY() functions to convert these coordinates to dialog units. The styles and ex_styles parameters allow you to specify additional styles and extended styles, respectively, for the dialog.

Using an in-memory dialog template

If successful, the CreateChildDlgTemplate() function will return a handle to the global memory object that holds the template. How do you instruct the Open dialog box to use this in-memory template? Well, the standard approach is to set the OFN_ENABLETEMPLATEHANDLE flag in the OPENFILENAME::Flags data member, and then assign the handle to the in-memory template's global memory object to the OPENFILENAME::hInstance data member. For example:

```
void __fastcall TForm1::
  OpenButtonClick(TObject *Sender)
{
  TCHAR filename[MAX_PATH];
  memset(&filename, 0, MAX_PATH);

  // create the in-memory template
  const HGLOBAL hTemplate =
    CreateChildDlgTemplate(
      160, 200, 104, 96);

  OPENFILENAME ofn = {
    OPENFILENAME_SIZE_VERSION_400};
  ofn.hwndOwner = Handle;
  ofn.nMaxFile = MAX_PATH;
  ofn.lpstrFile = filename;

  // assign the handle to the in-memory
  // template to the hInstance member
  ofn.hInstance = hTemplate;
```

```
  // add the OFN_ENABLETEMPLATEHANDLE
  // flag to the Flags member
  ofn.Flags =
    OFN_EXPLORER |
    OFN_ENABLETEMPLATEHANDLE;

  // display the open dialog box.
  if (GetOpenFileName(&ofn))
  {
    // ...
  }

  // clean up
  GlobalFree(hTemplate);
}
```

Although the `TOpenDialog` class provides the `Template` property (which adds the `OFN_ENABLETEMPLATE` flag to the `OPENFILENAME::Flags` data member, and which sets the specified resource identifier to the `OPENFILENAME::lpTemplateName` data member), there is no `TOpenDialog::TemplateHandle` property that would make things easy here. Fortunately, the `TOpenDialog` class does grant access to its internal `OPENFILENAME` structure, which you can modify in a fashion similar to that shown in this code snippet. Later, I'll show you how to perform this modification.

By using the `CreateChildDlgTemplate()` function, you don't have to worry about creating a resource script; and, you don't have to hand code the dimensions of the dialog and/or the location of the "stc32" static control. Instead (as I'll demonstrate shortly), you can compute these values at run time (by querying the size(s) of the VCL control(s) that you're going to add to the Open dialog) and then simply pass them to the `CreateChildDlgTemplate()` function.

Extending the `TOpenDialogEx` class

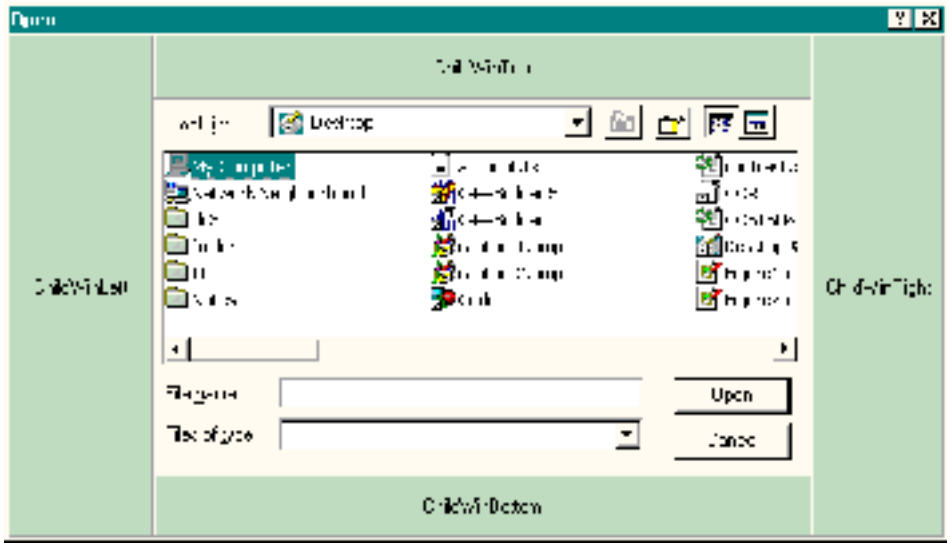
Last month, we created a `TOpenDialog` descendant class (`TOpenDialogEx`) that allowed you place a `TWinControl`-type VCL window to the right of the Open dialog's standard controls. Now that we know how to move these standard controls, let's extend this class to accept four `TWinControl`-type child windows. As you can see from **Listing A**, I've named these four child windows `ChildWinLeft`, `ChildWinTop`, `ChildWinRight`, and `ChildWinBottom`; **Figure D** depicts their layout within the Open dialog box.

Pointers to the four child windows are stored in the `ChildWinLeft_`, `ChildWinTop_`, `ChildWinRight_`, and `ChildWinBottom_` members. These members are initialized in the class constructor, like so:

```
__fastcall TOpenDialogEx::
  TOpenDialogEx(TComponent* Owner)
  : TOpenDialog(Owner),
    ChildWinLeft_(NULL),
    ChildWinTop_(NULL),
    ChildWinRight_(NULL),
    ChildWinBottom_(NULL),
    OldDlgWP_(NULL), NewDlgWP_(NULL)
{
}
```

Recall from last month that the `OldDlgWP_` and `NewDlgWP_` members are used to subclass the Open dialog box to support resizing. I'll return to this issue shortly.

Figure D



The layout of the TOpenDialogEx::ChildWinLeft, ChildWinTop, ChildWinRight, and ChildWinBottom windows.

Making room for the child windows

Last time, we adjusted the width of the Open dialog manually by using the `SetWindowPos()` API function. This time, we'll use an in-memory template to adjust the size of the Open dialog and move its standard controls. It's the job of the `TOpenDialogEx::DoCreateTemplate()` method to create the in-memory template. Here's how the method is defined:

```
HGLOBAL __fastcall TOpenDialogEx::
  DoCreateTemplate()
```

```
{
  // compute the desired location of the
  // "stc32" static control and the size
  // of the child dialog
  const short stc_x = ChildWinLeft_ ?
    ChildWinLeft_->Width : 0;
  const short stc_y = ChildWinTop_ ?
    ChildWinTop_->Height : 0;
  const short dlg_w =
    stc_x + (ChildWinRight_ ?
    ChildWinRight_->Width : 0);
  const short dlg_h =
    stc_y + (ChildWinBottom_ ?
    ChildWinBottom_->Height : 0);

  // create the child dialog template
  return CreateChildDlgTemplate(
    dlg_w, dlg_h, stc_x, stc_y);
}
```

The `DoCreateTemplate()` method determines which child windows are specified, and then computes the appropriate dimensions of the Open dialog's child dialog and the appropriate location of the "stc32" static control. These values are then passed to the previously defined `CreateChildDlgTemplate()` function, which returns a handle to the global memory object that holds the child dialog's template.

In order to instruct the `TOpenDialog` class to use our in-memory template, we'll need to augment the `TOpenDialog::DoTaskModal()` method. It's from within this method that we're granted access to the internal `OPENFILENAME` structure. Here's the code:

```
BOOL __fastcall
TOpenDialogEx::TaskModalDialog(
  void* DialogFunc, void* DialogData)
{
  // punt, if there's a resource template
  // specified or if there are no child
  // windows to add
  if (Template || !HasChild())
  {
    // show the dialog
    return TOpenDialog::TaskModalDialog(
      DialogFunc, DialogData);
  }
```

```
  // create an in-memory template
  const HGLOBAL hTemplate =
    DoCreateTemplate();
  try
  {
    // grab a pointer to the OPENFILENAME
    TOpenFilename* pofn =
      static_cast<TOpenFilename*>
        (DialogData);

    // remove the resource template flag
    pofn->Flags &=
      ~OFN_ENABLETEMPLATE;
    // add the in-memory template flag
    pofn->Flags |=
      OFN_ENABLETEMPLATEHANDLE;
    // set the in-memory template handle
    pofn->hInstance = hTemplate;

    // show the dialog
    const BOOL ok =
      TOpenDialog::TaskModalDialog(
        DialogFunc, DialogData);

    // clean up
    GlobalFree(hTemplate);
    return ok;
  }
  catch (...)
  {
    // clean up
    GlobalFree(hTemplate);
    throw;
  }
}
```

Adding the child windows to the dialog

After the Open dialog is resized, it’s time to add the child windows (`ChildWinLeft`, `ChildWinTop`, `ChildWinRight`, and `ChildWinBottom`). Whereas last time, we placed the child window (`ChildWin`) directly on the Open dialog, this time we need to place the child windows on the Open dialog’s child dialog. Again, this step is straightforward—you just use the `ParentWindow` property

like so:

```
void __fastcall TOpenDialogEx::
  DoShowChildWins()
{
  RECT RDlg;
  const HWND hChildDlg = Handle;
  if (GetClientRect(hChildDlg, &RDlg))
  {
    if (ChildWinLeft_)
    {
      // parent the child to the dialog
      ChildWinLeft_->
        ParentWindow = hChildDlg;

      // position/size the child window
      ChildWinLeft_->Left = 0;
      ChildWinLeft_->Top = 0;
      ChildWinLeft_->Height =
        RDlg.bottom;

      // show the child window
      ChildWinLeft_->Visible = true;
    }
    if (ChildWinTop_)
    {
      // parent the child to the dialog
      ChildWinTop_->
        ParentWindow = hChildDlg;

      // position/size the child window
      ChildWinTop_->Left =
        ChildWinLeft_ ?
        ChildWinLeft_->Width : 0;
      ChildWinTop_->Top = 0;
      ChildWinTop_->Width =
        RDlg.right -
        ChildWinTop_->Left -
        (ChildWinRight_ ?
         ChildWinRight_->Width : 0);

      // show the child window
      ChildWinTop_->Visible = true;
    }
```

```
    if (ChildWinRight_)
    {
      // parent the child to the dialog
      ChildWinRight_->
        ParentWindow = hChildDlg;

      // position/size the child window
      ChildWinRight_->Left =
        RDlg.right -
        ChildWinRight_->Width;
      ChildWinRight_->Top = 0;
      ChildWinRight_->Height =
        RDlg.bottom;

      // show the child window
      ChildWinRight_->Visible = true;
    }
    if (ChildWinBottom_)
    {
      // parent the child to the dialog
      ChildWinBottom_->
        ParentWindow = hChildDlg;

      // position/size the child window
      ChildWinBottom_->Left =
         ChildWinLeft_ ?
         ChildWinLeft_->Width : 0;
      ChildWinBottom_->Top =
        RDlg.bottom -
        ChildWinBottom_->Height;
      ChildWinBottom_->Width =
        RDlg.right -
        ChildWinBottom_->Left -
        (ChildWinRight_ ?
         ChildWinRight_->Width : 0);

      // show the child window
      ChildWinBottom_->Visible = true;
    }
  }
}
```

Recall that the `DoShowChildWins()` method (which was named `DoShowChildWin()` last month) is called from within the `TOpenDialogEx::DoShow()` method (i.e., after the dialog has initialized

its standard controls). Here's how the `DoShow()` method is defined:

```
void __fastcall TOpenDialogEx::DoShow()
{
  if (
    HasChild() &&
    !Options.Contains(ofOldStyleDialog))
  {
    // show the child windows
    DoShowChildWins();

    // subclass the dialog
    DoSubclassDialog(true);
  }
  TOpenDialog::DoShow();
}
```

Before the Open dialog closes, we need to remove the child windows, like so:

```
void __fastcall TOpenDialogEx::DoClose()
{
   if (ChildWinLeft_)
   {
      // clean up
      ChildWinLeft_->Hide();
      ChildWinLeft_->ParentWindow = 0;
   }
   if (ChildWinTop_)
   {
      // clean up
      ChildWinTop_->Hide();
      ChildWinTop_->ParentWindow = 0;
   }
   if (ChildWinRight_)
   {
      // clean up
      ChildWinRight_->Hide();
      ChildWinRight_->ParentWindow = 0;
   }
   if (ChildWinBottom_)
   {
      // clean up
      ChildWinBottom_->Hide();
```

```
      ChildWinBottom_->ParentWindow = 0;
    }
    TOpenDialog::DoClose();
}
```

Finishing up

Recall from last month that we subclassed the Open dialog in order to resize the child windows whenever the user resizes the Open dialog box. Now that we have four child windows instead of one, the subclass procedure needs to be modified to resize all four child windows. I won't repeat the code that does the actual subclassing, but here's the modified subclass procedure:

```
void __fastcall TOpenDialogEx::
  DialogWndProc(TMessage& Msg)
{
  // call the default window procedure
  const HWND hDlg = GetParent(Handle);
  Msg.Result = CallWindowProc(
    OldDlgWP_, hDlg,
    Msg.Msg, Msg.WParam, Msg.LParam);

  switch (Msg.Msg)
  {
    case WM_SIZE:
    {
      RECT RDlg;
      if (::GetClientRect(hDlg, &RDlg))
      {
        // resize the child windows
        if (ChildWinLeft_)
        {
          ChildWinLeft_->Height =
            RDlg.bottom;
        }
        if (ChildWinRight_)
        {
          ChildWinRight_->Height =
            RDlg.bottom;
        }
        if (ChildWinTop_)
        {
          ChildWinTop_->Width =
            RDlg.right -
```

```
            ChildWinTop_->Left -
            (ChildWinRight_ ?
             ChildWinRight_->Width : 0);
        }
        if (ChildWinBottom_)
        {
          ChildWinBottom_->Width =
            RDlg.right -
            ChildWinBottom_->Left -
            (ChildWinRight_ ?
             ChildWinRight_->Width : 0);
        }
      }
      break;
    }
    case WM_DESTROY:
    {
      // remove the subclass
      DoSubclassDialog(false);
      break;
    }
  }
}
```

Conclusion

How do you use the new `TOpenDialogEx` class? Because I provided an example in last month's article, and because the interface to the class hasn't changed much, I'll let the sample code (which is available at **www.bridgespublishing.com**) do the talking here. If you have further questions on the customizing common dialogs, please fell free to e-mail me at **dmc27@cornell.edu**.

Listing A: ***Declaration of the modified TOpenDialogEx class***

```
#include <dlgs.h>
class PACKAGE TOpenDialogEx : public TOpenDialog
{
public:
  // default constructor
  __fastcall TOpenDialogEx(TComponent* Owner);

  // introduced properties
  __property TWinControl* ChildWinLeft =
```

```
    {read = ChildWinLeft_,
     write = SetChildWinLeft};
  __property TWinControl* ChildWinTop =
    {read = ChildWinTop_,
     write = SetChildWinTop};
  __property TWinControl* ChildWinRight =
    {read = ChildWinRight_,
     write = SetChildWinRight};
  __property TWinControl* ChildWinBottom =
    {read = ChildWinBottom_,
     write = SetChildWinBottom};

protected:
  // inherited member functions
  DYNAMIC void __fastcall DoShow();
  DYNAMIC void __fastcall DoClose();
  virtual BOOL __fastcall TaskModalDialog
    (void* DialogFunc, void* DialogData);

  // introduced member functions
  virtual void __fastcall DoShowChildWins();
  virtual void __fastcall DoSubclassDialog
    (bool subclass);
  virtual void __fastcall DialogWndProc
    (TMessage& Msg);
  virtual HGLOBAL __fastcall DoCreateTemplate();

private:
  FARPROC OldDlgWP_;
  FARPROC NewDlgWP_;

  TWinControl* ChildWinLeft_;
  TWinControl* ChildWinTop_;
  TWinControl* ChildWinRight_;
  TWinControl* ChildWinBottom_;

  bool __fastcall HasChild()
    {
      return (
        ChildWinLeft_ || ChildWinTop_ ||
        ChildWinRight_ || ChildWinBottom_
        );
    }
  void __fastcall SetChildWinLeft
```

```
    (TWinControl* Value)
    {
      if (ChildWinLeft_ != Value)
      {
        ChildWinLeft_ = Value;
        ChildWinLeft_->Parent = NULL;
      }
    }
  void __fastcall SetChildWinTop
    (TWinControl* Value)
    {
      if (ChildWinTop_ != Value)
      {
        ChildWinTop_ = Value;
        ChildWinTop_->Parent = NULL;
      }
    }
  void __fastcall SetChildWinRight
    (TWinControl* Value)
    {
      if (ChildWinRight_ != Value)
      {
        ChildWinRight_ = Value;
        ChildWinRight_->Parent = NULL;
      }
    }
  void __fastcall SetChildWinBottom
    (TWinControl* Value)
    {
      if (ChildWinBottom_ != Value)
      {
        ChildWinBottom_ = Value;
        ChildWinBottom_->Parent = NULL;
      }
    }
};
```

Separating IDE-generated code

by Mark G. Wiseman

Lets say you're going write a text editor— an improved Notepad. You start up the C++Builder IDE and create a new application. Then you drop a `TMemo`, `TMainMenu`, `TToolBar`, `TActionList` and a `TStatusBar` onto the form. You also add a `TImageList` to hold the bitmaps you'll use for the `TActionList` and the `TToolBar`.

Now, you need to add `TAction` components to the `TActionList`. You'll need methods to open and save files, methods such as copy and paste for editing the text, and some methods for accessing online help for your editor.

Since C++Builder really is a visual development environment. You create each of the `TAction` components with the visual Action List Editor. You start by creating an action named `FileOpenAction`. You give it a Caption and everything else a good file open action would need. Next you double-click on the action name in the Action List Editor. This generates the skeleton code for the action's `OnExecute` event:

```
void __fastcall
TFileModule::FileOpenActionExecute(
  TObject *Sender)
{

}
//---------------------------------------
```

You add the necessary code the `FileOpenActionExecute()` method.

You assign this action to a menu item by selecting the action's name from the drop-down list displayed when you click on the menu item's `Action` property. You also assign the action to a tool bar button.

It's surprising how quickly you can create a better Notepad using the VCL and C++Builder's other visual development features. So, you're quickly finished and you save your work, renaming the `Unit1.cpp` file to `Main.cpp`.

This is when you realize that all the code for your editor is in one file—`Main.cpp`—and that file is 2,000 lines long. The extremely easy and quick visual development has left you with a mess.

What you wanted, and what good programming practice requires, is a project with several smaller source files.

Okay, so what do you do? You could create source files for file methods, editing methods, and help methods. But, when you cut the code for these methods from `Main.cpp` and paste it into the other files, the IDE will become very angry with you; telling you that it can't find the implementation of the method.

Or, you could start over. You could write all the code for each method yourself, foregoing all the help the IDE gave you in the first place. But, why make your job harder than it needs to be?

So, what *do* you do? You use `TDataModule` components.

TDataModule to the rescue

When Borland created the `TDataModule` component they created a very useful component; but they gave it a very misleading name. The "Data" part of the name makes you think this component is to be used for databases. And, in fact, `TDataModule` is a great place for placing database components such as `TTable` and `TQuery`. However, Borland's online help says that `TDataModule` "centralizes the handling of non-visual components in an application."

This definition says nothing about databases. It does however refer to non-visual components. Guess what? `TActionList` and `TImageList` are non-visual components.

A rewrite with TDataModule

Let's rewrite the editor using `TDataModule` components. There's a demo program on the Bridges Publishing Web site at **www.bridgespublishing.com**.

Let's start by creating a new application as you did above, but instead of dropping a `TActionList` and a `TImageList` onto the form, let's create a `TDataModule` for each of our programs functions: file methods, editing methods and help methods. Let's name these modules `FileModule`, `EditModule`, and `HelpModule`.

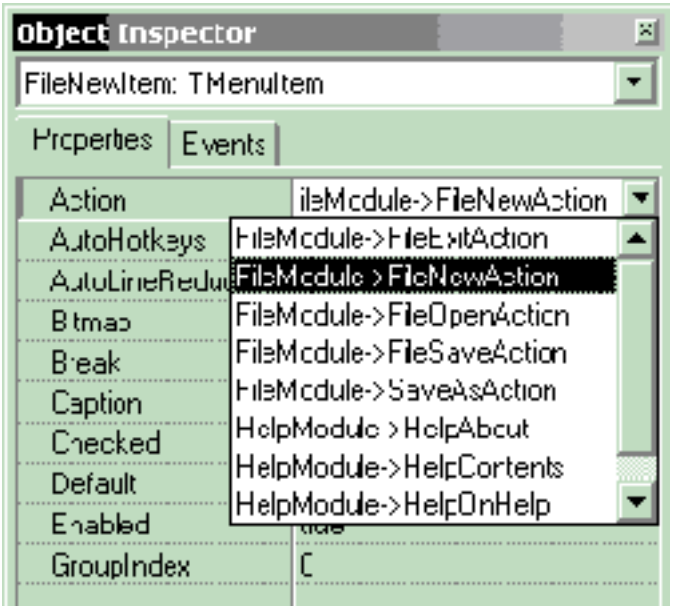
Now, let's drop a `TActionList` and a `TImageList` onto each of the modules. In the `FileModule`, we'll add `TAction` components to the `TActionList` for file operations just like you did above. We'll complete the other two modules by adding `TAction` components for editing and help.

Now we need to connect our modules to the form. This is really easy.

First, we need to include the header files for our three modules in the source file for the form, the file we named `Main.cpp`. We can do this by running the Include Unit Header Wizard. Click on the File menu of the IDE and select Include Unit Hdr… from the menu to run this wizard.

Now that the form is aware of the other modules, we can assign `TAction` components to the menu and tool bar just like you did above. The IDE is smart enough to realize that there are `TAction` components available in the three modules. **Figure A** shows the drop-down list for the `FileNewItem` menu item. Notice how the IDE show the items belonging to the `FileModule` and the `HelpModule`.

Figure A



TAction items in drop-down list.

A small problem

By using separate modules, the organization of the source is much cleaner and will be much easier to maintain. There is one small problem that you will have to deal with though.

Some of the methods assigned to the `OnExecute` event of the `TAction` components may need to refer to the form. For example, the `EditCutAction` needs to be able to access the `TMemo` component on the form.

If we include the header file for the form in the source for our modules, we will create a circular reference. And, although it might work in this instance, we don't want to do this.

So, lets add a property to the `EditModule` called `Memo`. This property will contain a pointer to the `TMemo` component on our form. **Listing A** contains the definition for `TEditModule` and the `Memo` property definition.

In the constructor for `TEditModule`, lets assign null to `Memo` just to be safe:

```
__fastcall TEditModule::TEditModule(
  TComponent* Owner): TDataModule(Owner)
{
  Memo = 0;
}
```

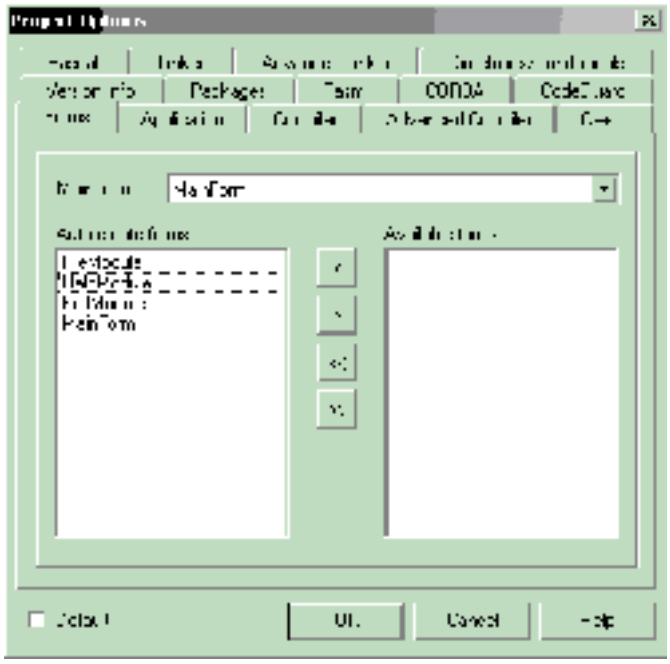
Then in the constructor for our form, lets assign the `TMemo` component to the edit module's `Memo` property:

```
__fastcall TMainForm::TMainForm(
  TComponent* Owner) : TForm(Owner)
{
  // More stuff…

  EditModule->Memo = Memo;
}
```

There's one more thing we have to do. When we created this new application, the IDE correctly assumed that the form, the only form, was our main form. So, the IDE wrote code to create this form before any others. This would be okay, except that the IDE also assumed we wanted to create the form before any data modules. So, in the function above, the edit module has not yet been created and `EditModule` does not point to a valid object.

We can easily fix this. Select the Options… item in the Project menu. Then select the Forms tab in the Project Options dialog and drag our form to the bottom of the list. **Figure B** shows the dialog after we've moved the form.

Figure B



Project Options dialog with form creation last

Now everything should work like we want. Here is the `EditCutExecute()` method we assigned to the `OnExecute` event of the `EditCutAction` component:

```
void __fastcall
TEditModule::EditCutExecute(
  TObject *Sender)
{
  if (Memo) Memo->CutToClipboard():
}
```

A big bonus

Separating our code into modules makes the project a lot easier to manage and maintain, but there is an even bigger bonus. Data modules can be saved to the Object Repository!

This means that once we create the `EditModule` we can save it to the Repository and then reuse it in every application we create that needs to edit a `TMemo` component. This is a huge timesaver. We can also save the `FileModule` and `HelpModule` to the Object Repository.

Conclusion

The `TDataModule` component of the VCL has more uses than its name implies. It great for organized database components; but, as we've seen in this article, it can also be used to separate code into more

meaningful and manageable modules.

Listing A: *Definition of TEditModule.*

```
class TEditModule : public TDataModule
{
  __published:
    TActionList *EditActionList;
    TEditCopy *EditCopy;
    TEditCut *EditCut;
    TEditDelete *EditDelete;
    TEditPaste *EditPaste;
    TEditSelectAll *EditSelectAll;
    TEditUndo *EditUndo;

    void __fastcall EditCopyExecute(
      TObject *Sender);
    void __fastcall EditCutExecute(
      TObject *Sender);
    void __fastcall EditDeleteExecute(
      TObject *Sender);
    void __fastcall EditPasteExecute(
      TObject *Sender);
    void __fastcall EditSelectAllExecute(
      TObject *Sender);
    void __fastcall EditUndoExecute(
      TObject *Sender);

  public:
    __fastcall TEditModule(TComponent* Owner);

    __property TMemo *Memo =
      {read = memo, write = memo};

  private:
    TMemo *memo;
};
```

Buttons, buttons and buttons

by Mark G. Wiseman

If you have worked with the Visual Component Library (VCL) for long, you will have noticed that Borland has provided you with more than one type of button. There are components named `TButton`, `TBitBtn` and `TSpeedButton`.

In this article I'll explain the differences between these components and give you some suggestions on when you might want to use one over the other. I'll also touch briefly on `TToolButton`, the button used with the `TToolBar` component.

There is a demonstration program on the Bridges Publishing web site that will give you a good idea of how these components look and behave.

Different, yet the same

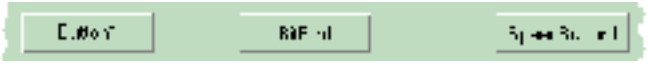
Before I talk about the difference between the VCL button components, I'd like to point out the similarities.

`TButton`, `TBitBtn` and `TSpeedButton` all have `TControl` as a common ancestor. This means that they share many properties and events. The `Action`, `Hint` and `Caption` properties are some of the more important properties shared by these components.

I'll come back to the `Action` property after I discuss the `OnClick` event below. The `Hint` property contains the text string that can appear when the user moves the mouse over the button.

The `Caption` property contains the text that is displayed on the button's face. **Figure A** shows a section of the demo program with a `TButton`, `TBitBtn` and `TSpeedButton`. Their respective captions are "Button1", "BitBtn1" and "SpeedButton1".

Figure A



TButton, TBitBtn and TSpeedButton components

The `OnClick` event is the most important of the shared events. Users click buttons to initiate events within a program. Most of the time you will use the `OnClick` event to respond to a button click.

OnClick works exactly the same for all three types of button components. Here is an example of a function assigned to the OnClick event:

```
void __fastcall TMainForm::ButtonClick(
  TObject *Sender)
{
  TButton *button =
    dynamic_cast<TButton *>(Sender);
  if (button == 0) return;

  ShowMessage(button->Caption
    + " was clicked.");
}
```

If you assign an action to the Action property of a button, the action's OnExecute event will be assigned to the button's OnClick event. Several of the action's other properties (e.g. Caption and Hint) will also be assigned to corresponding properties of the button.

TButton

TButton is the button component you will probably use the most. The TButton component is the VCL encapsulation of the standard Windows button and works just like a button that you would create using the Windows API. You place a TButton onto a form, give it a meaningful Caption and have it respond to user clicks.

As mentioned above, you could have the button respond to clicks by writing an OnClick event or it could respond through a TAction object.

TBitBtn

The TBitBtn component is derived directly from TButton.

The TBitBtn component adds the ability to display glyphs or small bitmaps on the button's face through several properties: Glyph, NumGlyphs, Layout, Margin, Spacing and Kind.

The Glyph property contains the glyphs to be displayed. If the Glyph property is empty, only the Caption text is displayed on the face.

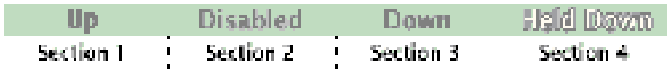
A TBitBtn can have three states: up, down, and disabled. You can have the TBitBtn display a different bitmap for each state. If you provide only one bitmap, TBitBtn will use it for the up and down

states and will alter it (gray it) for the disabled state.

You can only assign one bitmap to the `Glyph` property, but that bitmap can contain one to four sections. The fourth section is unused by `TBitBtn` but can be used by `TSpeedButton` (see below). The `NumGlyphs` property should be set to the number of sections in the `bitmap`.

Figure B shows the bitmap used in the demonstration program, and defines its four sections. My total lack of artistic skills forced me to use text in the bitmap, but you can still see how it's used.

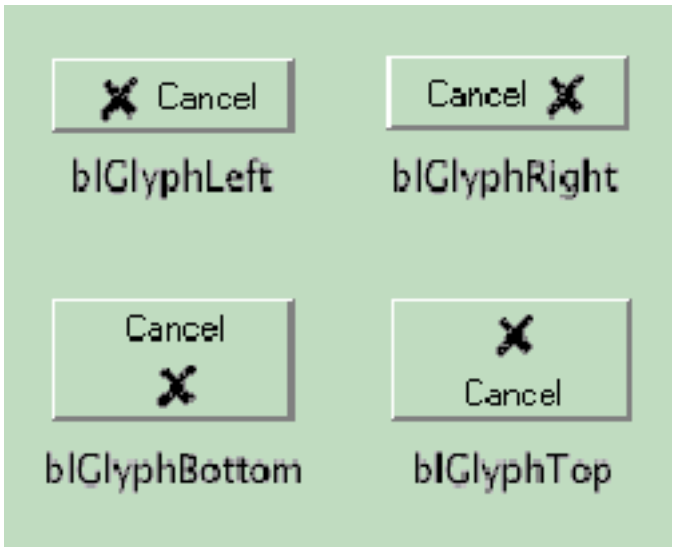
Figure B



A bitmap assigned to the Glyph property of TBitBtn and TSpeedButton

The `Layout` property determines whether the glyph is displayed on the left, right, top or bottom of the button's text. **Figure C** shows the four positions and corresponding value of `Layout`.

Figure C



Effects of the Layout property

The `Margin` property of `TBitBtn` determines the number of pixels between the glyph and the edge of the button. For instance, if the `Layout` property is set to `blGlyphLeft` and the `Margin` property is set to 10, then the glyph's left edge will be 10 pixels from the buttons left edge. If `Margin` is set to -1, the glyph and caption text will be centered on the button face.

The `Spacing` property is the number of pixels between the glyph and the caption text. By default, the

two are separated by 4 pixels.

The `Kind` property will save you a lot of work. Borland has incorporated ten commonly used glyphs and captions, such as those for the OK and Cancel buttons used in the demonstration program. Using the built in glyphs will relieve you from having to create them.

Setting the `Kind` property of a `TBitBtn` object to `bkOK` will cause the button to use an internal glyph of a checkmark and the caption of “OK”. It also sets the button’s `ModalResult` property to the corresponding value.

There is one more thing I should point out about `TBitBtn`. If you leave the button’s `Caption` property empty, then only the glyph will be displayed. This is also true of the next VCL button component I’ll discuss, `TSpeedButton`.

TSpeedButton

The `TSpeedButton` component was originally meant to be used when building tool bars from `TPanel` objects. Before Borland added the `TToolBar` component, using a `TPanel` object as a tool bar was a fairly common practice.

To save Windows resources, in particular window handles, Borland created the `TSpeedButton` component. Unlike `TButton` and `TBitBtn`, which are derived from `TWinControl`, `TSpeedButton` is derived from `TGraphicControl`. `TGraphicControl` does not use a window handle like `TWinControl` does. So, `TSpeedButton` is very resource efficient.

Unfortunately, this efficiency comes at a price. Since `TSpeedButton` has no window handle, it cannot receive the keyboard focus. This means you cannot tab to the button and press it by hitting Enter on the keyboard.

There are some neat features to `TSpeedButton` though. Just like `TBitBtn`, `TSpeedButton` can use glyphs. However, `TSpeedButton` adds a fourth state and thus can use four different glyphs; one more than `TBitBtn`.

The fourth state supported by `TSpeedButton` is the *keep down* state. The button is depressed when you click on it with the left mouse button and it stays depressed. Click it again with the left mouse button and the `TSpeedButton` goes back to the *up* state. Using this feature you can create `TSpeedButton` objects that act like check boxes and radio buttons.

Figure D shows a section of the demo program where three `TSpeedButton` objects are set to behave like radio buttons. In this example, the top button is depressed and the other two are not. If you click on the middle or bottom button, then that button would then remain depressed and the top button would

return to its up position.

Figure D

TspeedButtons used as radio buttons.

It's easy to make `TSpeedButton` objects act as radio buttons. In the example above, I put three `TSpeedButton` objects on a form and set the `GroupIndex` property to 1 for each of the objects. There is no significance to using the value of 1. The only thing that matters is that all of the buttons in a "radio group" have the same `GroupIndex` value. I also made sure that the `AllowAllUp` property of each of the objects was set to `false`. This tells the objects that one of them must always be depressed. At design time, you can choose the button that starts off depressed by setting its `Down` property to `true`.

To make a single `TSpeedButton` object act like a check box, set its `GroupIndex` to a value (other than zero) that is not used by any other `TSpeedButton`. Then set the `AllowAllUp` property to `true`.

Setting the `GroupIndex` property to zero tells the `TSpeedButton` to act like a normal button.

There is one more property of `TSpeedButton` I'd like to mention. The `Flat` property, when set to `true`, causes the button to be drawn flat, unless the mouse is over it. When the mouse does pass over the button, it rises up to take on the appearance of a normal button.

TToolButton

I wanted to briefly discuss `TToolButton` since in a way it replaces `TSpeedButton` in `TSpeedButton's` original function. `TToolButton` objects can only be created in `TToolBar` objects and the `TToolBar` object is really too complicated to explain in this article. Just be aware that if you want to create a tool bar with buttons that you will probably want to use a `TToolBar` and `TToolButtons`.

Third-party buttons

There are a lot of third-party button components available. Some of them give you shapes other than rectangular. You can have round buttons or even triangular buttons. Other button components have animated glyphs. Still others have multiple states—more states than only up or down.

If you have a need for a specialized button component, someone has probably already created one you can use.

Which button?

So, which button component should you use? Here are some guidelines you should consider.

First, if you want a standard Windows button, then use `TButton`. It *is* a standard Windows button.

If you need to add a glyph to a button, use `TBitBtn`. If you used `TSpeedButton`, you would force users to use the mouse since `TSpeedButton` cannot receive keyboard focus. Your users might not appreciate this.

There are a couple of times when you might consider using `TSpeedButton`. The first is when you need to create a toolbar that you can't create using the `TToolBar` component.

The second is when you need to have a lot of buttons in a single window. A good example of this would be a scientific calculator. Using `TSpeedButton` would save a lot of Windows resources and the need for keyboard focus could be met by assigning the numeric keypad to the numerals and arithmetic functions.

Punch out

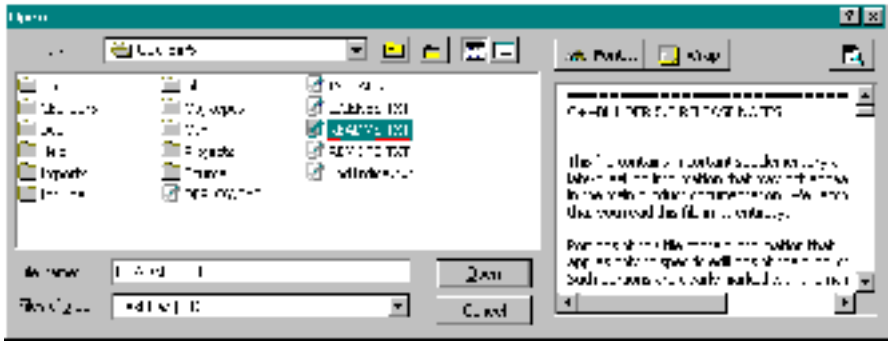
I hope that I have given you a better feel for the different types of button components included in the VCL and for when you might use each of them.

Custom open dialogs, part I

by Damon Chandler

In this two-part series, I'll show you how to customize the standard Open dialog box. In this first installment, I'll demonstrate how to create an extended `TOpenDialog` class that allows you to place any `TWinControl` descendant to the right of the Open dialog's standard controls. **Figure A** depicts an example.

Figure A



A customized Open dialog box that allows users to preview text files.

The standard Open dialog and the TOpenDialog class

The `TOpenDialog` is one of the easiest VCL components to work with. You simply drop one on your form at design time, set a few of its properties, call its `Execute()` method (to display the dialog), and then query its `FileName` (or `Files`) property to determine which file (or files) to open:

```
void __fastcall TForm1::
  OpenButtonClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    // open the file
    // OpenDialog1->FileName...
  }
}
```

In fact, even if the `TOpenDialog` class wasn't available, the standard API approach is almost as

simple. You first initialize an `OPENFILENAME` structure, set a few of its data members, pass its address to the `GetOpenFileName()` function (to display the dialog), and then read the `OPENFILENAME::lpstrFile` data member to determine which file (or files) to open:

```
void __fastcall TForm1::
  OpenButtonClick(TObject *Sender)
{
  TCHAR filename[MAX_PATH];
  memset(&filename, 0, MAX_PATH);

  OPENFILENAME ofn = {
    OPENFILENAME_SIZE_VERSION_400 };
  ofn.hwndOwner = Handle;
  ofn.nMaxFile = MAX_PATH;
  ofn.lpstrFile = filename;
  ofn.Flags = OFN_EXPLORER;

  // display the open dialog box
  if (GetOpenFileName(&ofn))
  {
    // open the file "filename"...
  }
}
```

The `TOpenDialog` class performs these steps for you from within its `DoExecute()` method.

Either of these approaches will work if you don't need to customize the Open dialog; if you do, the process is a bit more complicated. There are two standard ways to perform this customization:

- Use a custom resource script.

- Use a hook procedure.

The resource-based approach is a bit of a pain not only because C++Builder doesn't come with a useful resource editor, but also primarily because there's no simple way to use VCL controls within the resource script. By using a hook procedure, on the other hand, you can place any `TWinControl` descendant directly on the dialog. In fact, the `TOpenDialog` class (in C++Builder versions 3 and higher) uses a hook procedure to support many of its events (e.g., `OnShow`). I'll explain how this works in the next section.

Hooking the dialog

Installing a hook procedure for the Open dialog involves three main steps:

- Define the hook procedure.
- Assign a pointer to the hook procedure to the `OPENFILENAME::lpfnHook` data member.
- Add the `OFN_ENABLEHOOK` bit to the `OPENFILENAME::Flags` data member.

Here's an example of steps 2 and 3:

```
void __fastcall TForm1::
  OpenButtonClick(TObject *Sender)
{
  TCHAR filename[MAX_PATH];
  memset(&filename, 0, MAX_PATH);

  OPENFILENAME ofn = {
    OPENFILENAME_SIZE_VERSION_400 };
  ofn.hwndOwner = Handle;
  ofn.nMaxFile = MAX_PATH;
  ofn.lpstrFile = filename;
  ofn.lpfnHook = HookProc;
  ofn.Flags =
    OFN_EXPLORER | OFN_ENABLEHOOK;

  // display the open dialog box
  if (GetOpenFileName(&ofn))
  {
    // open the file "filename"...
  }
}
```

And, here's how the hook procedure (`HookProc`) is defined:

```
#include <cassert>
UINT __stdcall HookProc(
   HWND hChildDlg, UINT msg,
   WPARAM wParam, LPARAM lParam)
{
  if (msg != WM_NOTIFY) return 0;

  OFNOTIFY* pNotify =
```

```
    reinterpret_cast<OFNOTIFY*>(lParam);
  assert(pNotify != NULL);

  switch (pNotify->hdr.code)
  {
    case CDN_INITDONE:
      // dialog has been initialized
      // and is about to be displayed
      break;
    case CDN_SELCHANGE:
      // user has selected a new file
      break;
    case CDN_FOLDERCHANGE:
      // user had selected a new folder
      break;
    // etc...
  }
  return 0;
}
```

By using this code, the dialog box will call the hook procedure whenever certain events take place within the dialog's controls. These notifications are sent in the form of a `WM_NOTIFY` message, whose accompanying `lParam` value specifies a pointer to an `OFNOTIFY` structure. The `OFNOTIFY::hdr` data member is an `NMHDR` structure whose `code` data member specifies the type of notification. For example, when the `code` data member specifies `CDN_INITDONE`, the dialog box has finished initializing its controls and it is about to be shown; this is when the `TOpenDialog` class fires its `OnShow` event handler. Similarly, when the code data member specifies `CDN_SELCHANGE`, the user has selected a new file in the Explorer-style list view control; this is when the `TOpenDialog` class fires its `OnSelectionChange` event handler. Other notifications include:

- `CDN_FILEOK` (compare with `OnCanClose`)
- `CDN_FOLDERCHANGE` (compare with `OnFolderChange`)
- `CDN_TYPECHANGE` (compare with `OnTypeChange`)
- `CDN_INCLUDEITEM` (compare with `OnIncludeItem`)

Notice that the first parameter to the hook procedure is not a handle to the dialog itself. Rather, it's a handle to a child dialog. This child dialog is designed to serve as a container for additional controls (If you use a resource script, you can set the initial size of the child dialog). As I'll discuss in the sections that follow however, you don't have to use this child dialog; instead, you can place controls directly on

the Open dialog box.

Extending the TOpenDialog class

Listing A contains the declaration of a `TOpenDialog` descendant class called `TOpenDialogEx`. This class has three private members: `ChildWin_`, `OldDlgWP_`, and `NewDlgWP_`, which are initialized in the class constructor like so:

```
__fastcall TOpenDialogEx::
  TOpenDialogEx(TComponent* Owner)
  : TOpenDialog(Owner), ChildWin_(NULL),
  OldDlgWP_(NULL), NewDlgWP_(NULL)
{
}
```

The `OldDlgWP_` and `NewDlgWP_` members are used to subclass the Open dialog box (I'll discuss how and why this is done later). The `ChildWin_` data member is used with the `ChildWin` property, the latter of which is used to specify the `TWinControl`-type object that's to be placed on the Open dialog box.

Making room for the child window

Recall that an Open dialog box contains a child dialog that's designed to hold additional controls. As I mentioned previously, you can use a resource script to define the size of this child dialog. In this case the Open dialog box will automatically resize itself to accommodate the child dialog's new size. In our case, we're not using a resource script, so the child dialog's width is set to zero. This much is fine because we're not going to use the child dialog anyway. We will, however, need to increase the width of the Open dialog box to make room for our own `TWinControl`-type child window (`ChildWin`).

How do we change the Open dialog's width? Well, unfortunately, because the Open dialog box isn't a VCL window, adjusting its width is not as simple as setting a `Width` property. Instead, we'll have to use the `MoveWindow()` or `SetWindowPos()` API functions. I prefer the latter because it allows you to change a window's size without specifying its position. We'll call the `SetWindowPos()` function after the dialog has initialized its controls—i.e., in response to the `CDN_INITDONE` notification, or in VCL terms, from within an augmented `DoShow()` method:

```
void __fastcall TOpenDialogEx::DoShow()
{
  if (ChildWin_ &&
    !Options.Contains(ofOldStyleDialog))
  {
```

```
    RECT RDlg;
    const HWND hDlg = GetParent(Handle);
    if (GetWindowRect(hDlg, &RDlg))
    {
      // resize the dialog box
      SetWindowPos(
        hDlg, NULL, 0, 0,
        RDlg.right - RDlg.left +
        ChildWin_->Width + 6,
        RDlg.bottom - RDlg.top,
        SWP_NOMOVE | SWP_NOZORDER);

      // place and show the child window
      DoShowChildWin();

      // subclass the dialog
      DoSubclassDialog(true);
    }
  }
  TOpenDialog::DoShow();
}
```

You might be wondering why this code uses the `GetParent()` API function (along with the `TOpenDialog::Handle` property) to retrieve a handle to the Open dialog box. Well, as it turns out, the `Handle` property returns a handle to the *child* dialog (which in our case has a width of zero), not a handle to the Open dialog. Thus, in order to grab a handle to the Open dialog box (`hDlg`), we need to pass the value that's returned by the `Handle` property to the `GetParent()` function. This handle (`hDlg`) is then used with the `GetWindowRect()` and `SetWindowPos()` functions to query and change the Open dialog's width.

Adding the child window to the dialog

After the Open dialog's width has been adjusted, the next task is to place `ChildWin` on the Open dialog. This is the job of the `DoShowChildWin()` method, which is defined as follows:

```
void __fastcall TOpenDialogEx::
  DoShowChildWin()
{
  RECT RDlg;
  const HWND hDlg = GetParent(Handle);
  if (GetClientRect(hDlg, &RDlg))
  {
    // parent the child to the dialog
```

```
    ChildWin_->ParentWindow = hDlg;

    // position/size the child window
    ChildWin_->Top = 3;
    ChildWin_->Left = RDlg.right -
      ChildWin_->Width - 6;
    ChildWin_->Height = RDlg.bottom - 10;

    // show the child window
    ChildWin_->Visible = true;

    // bring the child window to the top
    ChildWin_->BringToFront();
  }
}
```

As before, we first use the `GetParent()` function to retrieve a handle to the Open dialog box because we want to place `ChildWin` on the Open dialog, not on the zero-width child dialog. Actually placing `ChildWin` on the Open dialog is trivial—we simply use the `ParentWindow` property.

Resizing the child window

At this point, we have code to adjust the Open dialog box's width and code to place the child window (`ChildWin`) on the dialog box. So far, so good. But if you're using a version of Windows that allows the user to resize the Open dialog box, `ChildWin` won't automatically conform to the Open dialog's new size.

Although we can easily resize `ChildWin` manually by changing its `Width` and `Height` properties, we'll still need to know *when* to adjust these properties. Ideally, we want to resize `ChildWin` immediately after the Open dialog box has been resized. Unfortunately, there is no notification—such as `CDN_RESIZED` or `OnResized`—that tells us when the user has resized the dialog box. So, in order to determine when to resize `ChildWin`, we'll need to subclass the Open dialog box and respond to the `WM_SIZE` message within the subclass procedure. This is where the `OldDlgWP_` and `NewDlgWP_` members and the `DoSubclassDialog()` and `DialogWndProc()` methods come into play.

Notice from the previous definition of the `DoShow()` method that after the Open dialog's width is adjusted (via the `SetWindowPos()` function) and after `ChildWin` has been placed on the dialog (via the `DoShowChildWin()` function), a call is made to the `DoSubclassDialog()` method. This method, which places an instance subclass on the Open dialog box, is defined like so:

```
void __fastcall TOpenDialogEx::
  DoSubclassDialog(bool subclass)
```

```
{
  if (OldDlgWP_)
  {
    // restore the old window procedure
    SetWindowLong(
      GetParent(Handle), GWL_WNDPROC,
      reinterpret_cast<LONG>(OldDlgWP_)
      );
    OldDlgWP_ = NULL;
  }
  if (NewDlgWP_)
  {
    // free the function map
    FreeObjectInstance(NewDlgWP_);
    NewDlgWP_ = NULL;
  }

  if (!subclass) return;

  // allocate the function map
  NewDlgWP_ = reinterpret_cast<FARPROC>(
    MakeObjectInstance(DialogWndProc));

  // perform the instance subclass
  const LONG res = SetWindowLong(
    GetParent(Handle), GWL_WNDPROC,
    reinterpret_cast<LONG>(NewDlgWP_));
  OldDlgWP_ =
    reinterpret_cast<FARPROC>(res);
}
```

This function uses the `SetWindowLong()` API function to replace or restore the dialog's window procedure, depending on the value of the `subclass` parameter. (The call to the `MakeObjectInstance()` VCL function is needed because our subclass procedure, `DialogWndProc()`, is a non-static class member function as opposed to a standard callback function.)

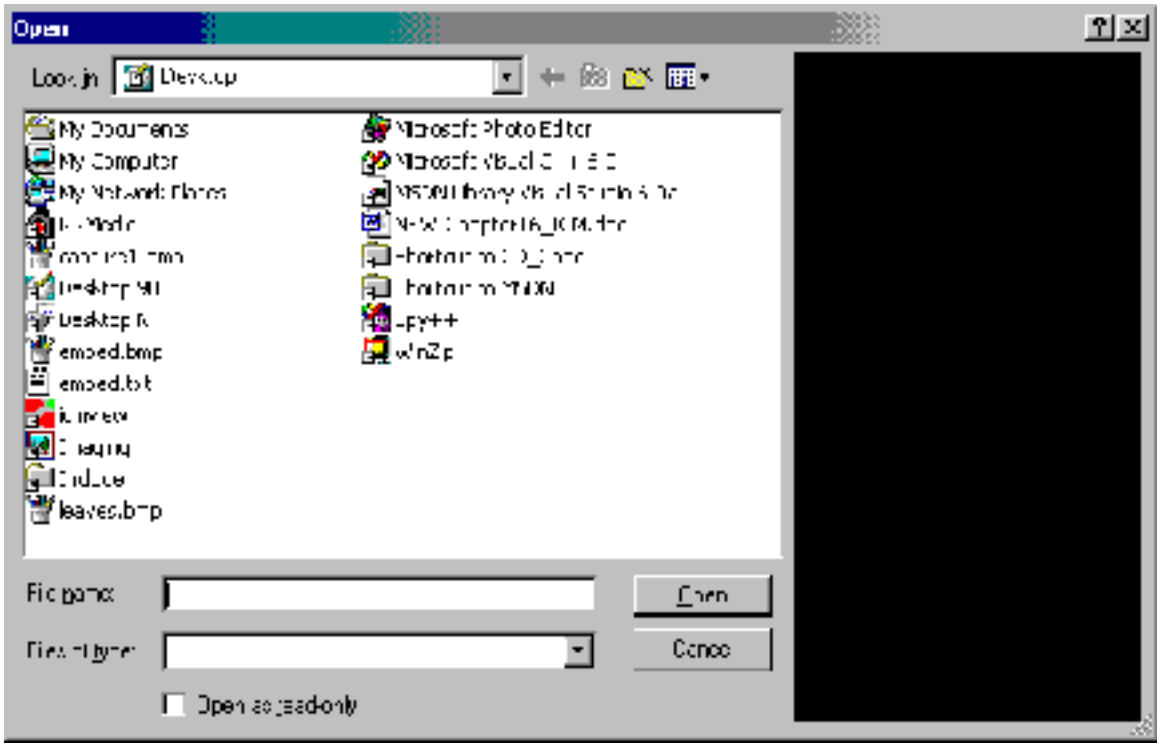
After the Open dialog box is subclassed, the `WM_SIZE` message will be sent to our subclass procedure (`DialogWndProc()`) immediately after the dialog has been resized. At this point, we can adjust the size of the `ChildWin`:

```
void __fastcall TOpenDialogEx::
  DialogWndProc(TMessage& Msg)
{
```

```
  // call the default window procedure
  Msg.Result = CallWindowProc(
    OldDlgWP_, GetParent(Handle),
    Msg.Msg, Msg.WParam, Msg.LParam
    );

  switch (Msg.Msg)
  {
    case WM_SIZE:
    {
      RECT RDlg;
      const HWND hDlg =
        GetParent(Handle);
      if (::GetClientRect(hDlg, &RDlg))
      {
        // resize the child window
        ChildWin_->Height =
          RDlg.bottom - 10;
      }
      break;
    }
    case WM_DESTROY:
    {
      // remove the subclass
      DoSubclassDialog(false);
      break;
    }
  }
}
```

Again, we subclass the Open dialog box so that we can resize the child window (`ChildWin`) whenever the dialog box is resized. **Figure B** depicts a `TOpenDialogEx`-type dialog whose child window is simply a black `TPanel` object. Notice that the panel is resized along with the dialog.

Figure B



A `TOpenDialogEx`-type dialog on Windows 2000; the child window is a black `TPanel` object that's resized whenever the user resizes the dialog.

Finishing up

Recall that we use the `ChildWin` object's `ParentWindow` property (from within the `DoShowChildWin()` method) to place the child window on the Open dialog box. Accordingly, before the dialog box is closed, we'll need to remove and hide the child window. We can do this by augmenting the `DoClose()` method as follows:

```
void __fastcall TMyOpenDialog::DoClose()
{
   if (ChildWin_)
   {
      // clean up
      ChildWin_->Hide();
      ChildWin_->ParentWindow = NULL;
   }
   TOpenDialog::DoClose();
}
```

And, in case you're wondering, here's the definition of the `SetChildWin()` method, which is called whenever the `ChildWin` property is set:

```
void __fastcall TOpenDialogEx::
  SetChildWin(TWinControl* Value)
{
  if (ChildWin_ != Value)
  {
    ChildWin_ = Value;
    ChildWin_->Parent = NULL;
  }
}
```

Notice that, in addition to updating the `ChildWin_` member, the `DoSetChildWin()` method sets the child window's `Parent` property to `NULL`. This is an important step because the `ParentWindow` assignment would have no effect if the `ChildWin`'s `Parent` property weren't `NULL`.

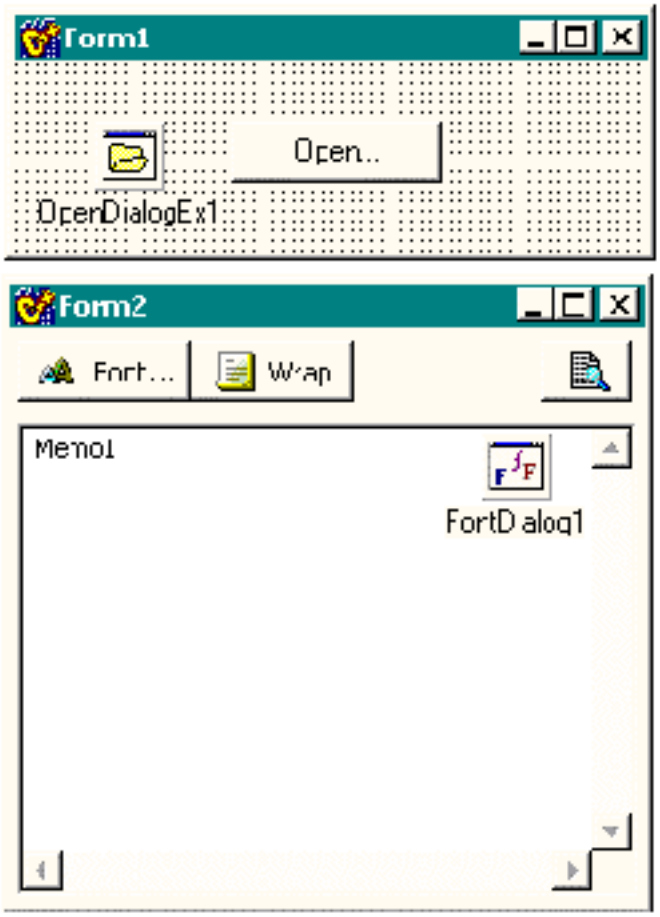
That's it for the `TOpenDialogEx` class. As you can see, its implementation is relatively short, primarily because the class doesn't handle any of the child window's functionality; that part is up to you.

Using TOpenDialogEx

Let's work through an example of using the `TOpenDialogEx` class; specifically, let's create the text-file-preview Open dialog box that's shown in **Figure A**.

As depicted in **Figure C**, the main form of this example (`Form1`) consists only of a single button and a `TOpenDialogEx` object (`OpenDialogEx1`). `Form2` is what we'll use as the Open dialog's child window.

Figure C



Design-time forms: `Form1` is the application's main form; `Form2` is to be used as the child window of `OpenDialogEx1`.

As provided in **Listing B**, the `TForm2` class consists of a `TMemo` control (`Memo1`), a couple of `TPanel` objects, a few `TSpeedButton`s, and a `TFontDialog` object. Together, these VCL controls will provide the text-file-preview functionality of the Open dialog box.

The `TForm2` class also contains a public method called `SelectionChange()`, which serves to show (and load) or hide `Memo1`, depending on whether the selected file is supported. Here's how the method is defined:

```
void __fastcall TForm2::
  SelectionChange(AnsiString filename)
{
  const bool file_ok =
    !(FileGetAttr(filename) &
      faDirectory) &&
     (ExtractFileExt(filename).
      AnsiCompareIC(".TXT") == 0);

  Memo1->Visible = file_ok;
```

```
  if (file_ok)
  {
    Memo1->Lines->LoadFromFile(filename);
  }
}
```

Now, all that's left is to code the `OnClick` event handlers of the various `TSpeedButton`s:

```
void __fastcall TForm2::
  PreviewButtonClick(TObject *Sender)
{
  ShellExecute(Handle, "open",
    Form1->OpenDialogEx1->
      FileName.c_str(),
    NULL, "", SW_SHOWNORMAL);
}

void __fastcall TForm2::
  WordWrapButtonClick(TObject *Sender)
{
  if (WordWrapButton->Down)
  {
    Memo1->WordWrap = true;
    Memo1->ScrollBars = ssVertical;
  }
  else
  {
    Memo1->WordWrap = false;
    Memo1->ScrollBars = ssBoth;
  }
}

void __fastcall TForm2::
  FontButtonClick(TObject *Sender)
{
  if (FontDialog1->Execute())
  {
    Memo1->Font = FontDialog1->Font;
  }
}
```

The implementation of the `TForm1` class is even simpler. Here's the code for `OnClick` event handler of `Form1`'s button:

```
void __fastcall TForm1::
  OpenButtonClick(TObject*)
{
  OpenDialogEx1->ChildWin = Form2;
  OpenDialogEx1->Execute();
}
```

And, here's the handler that's assigned to `OpenDialogEx1`'s `OnSelectionChange()` event:

```
void __fastcall TForm1::
  OpenDialogEx1SelectionChange(TObject*)
{
  Form2->SelectionChange(
    OpenDialogEx1->FileName);
}
```

Conclusion

I've shown you how to create a `TOpenDialog` descendant class and one example of how to use it. Although the example was rather dry, the main point that I wanted to make was that you can use the VCL to design and implement the Open dialog's extended functionality. This modular approach is not only significantly easier than the resource-based method (which requires that you hand-code all of the extra controls), it obviates the hassle of creating a new `TOpenDialog` descendant class for every new file type that you want to support. And, if you do need to reuse a previously customized dialog in another project, you can simply add the child window (`TForm2` in the previous example) to the Object Repository. (Note that the `TOpenDialogEx` class will not work in C++Builder version 1.)

Next month, I'll show you how to place additional controls on other parts of the dialog, and how to customize some of the dialog's standard controls.

Listing A: *Declaration of the TOpenDialogEx class*

```
class PACKAGE TOpenDialogEx : public TOpenDialog
{
public:
  // default constructor
  __fastcall TOpenDialogEx(TComponent* Owner);

  // introduced property
  __property TWinControl* ChildWin =
    {read = ChildWin_, write = SetChildWin};
```

```
protected:
  // inherited member functions
  DYNAMIC void __fastcall DoShow();
  DYNAMIC void __fastcall DoClose();

  // introduced member functions
  virtual void __fastcall DoShowChildWin();
  virtual void __fastcall
    DoSubclassDialog(bool subclass);
  virtual void __fastcall
    DialogWndProc(TMessage& Msg);

private:
  TWinControl* ChildWin_;
  void __fastcall SetChildWin(TWinControl* Value);

  FARPROC OldDlgWP_;
  FARPROC NewDlgWP_;
};
```

Listing B: *Declaration of the TForm2 class*

```
class TForm2 : public TForm
{
__published:
  TMemo *Memo1;
  TPanel *Panel1;
  TPanel *Panel2;
  TSpeedButton *PreviewButton;
  TSpeedButton *WordWrapButton;
  TSpeedButton *FontButton;
  TFontDialog *FontDialog1;

  void __fastcall FontButtonClick(
    TObject *Sender);
  void __fastcall WordWrapButtonClick(
    TObject *Sender);

public:
  __fastcall TForm2(TComponent* Owner);
  void __fastcall SelectionChange(
    AnsiString filename);
};
```

IDE Tip: Selecting the form

by Kent Reisdorph

It is not uncommon to have a form entirely covered by components. For example, you may have a tree view, list view, memo, panel or any other component aligned to the client area of the form. The result is that no part of the form's client area is uncovered. Setting a property or event of the form, obviously, requires you to first select the form. Selecting the form is not possible by clicking on it in this case. There are at least two ways to go about selecting a form covered by other components.

The first way to select the form is to simply use the component selector combo box at the top of the Object Inspector. This certainly works, but the form can be hard to find amongst the other listed components. This is particularly problematic when the form is a main form where, not only are the form's components listed, but also every menu item. Finding the main form in the list takes time.

The second way to select the form is much easier and often overlooked. Simply click on any component on the form and hit the Escape key. When you do, that component's parent is automatically selected. For example, if you have a simple form with a memo aligned to client, hitting ESC will select the form.

This method can be used to select any container component. Consider the example where you have a panel aligned to client and a memo on that panel, also aligned to client. Select the memo and hit ESC; the panel is selected. Hit ESC one more time and the form is selected. This technique allows you to quickly select any component that is otherwise impossible (or inconvenient) to select by clicking with the mouse.

You may find this tip simplistic if you are already familiar with this IDE feature. However, I find many people who are unaware of this feature, some of whom have been using C++Builder for years.

Writing aggregate components

by Kent Reisdorph

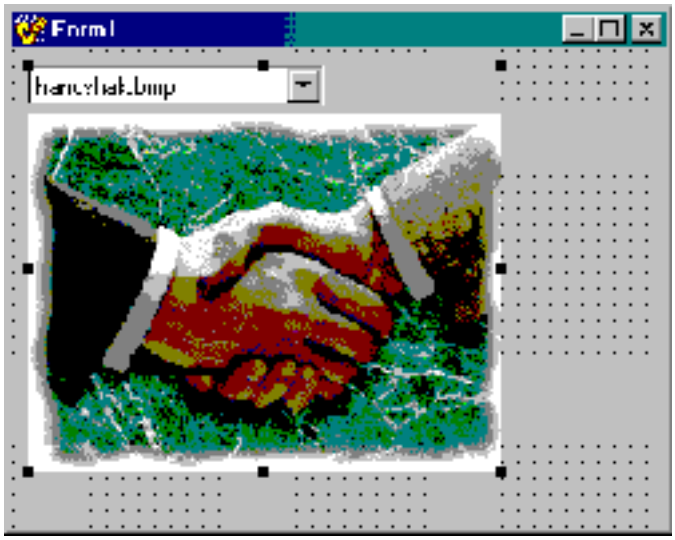
Most components you write are derived from one of the VCL components or classes. A button component, for example, might be derived from TCustomButton or TButton. This is the traditional way of writing components, and many component writers never venture beyond this type of component.

Sometimes, however, you need a more complex component. One such type of component uses aggregation to build a single component by implementing one or more existing components. This article will show the basics of creating aggregate components and will address some of the problems that can arise.

TPictureSelector

The example component for this article is called `TPictureSelector`. This component combines a `TComboBox` component and a `TImage` component into a single component. The combo box contains a list of bitmap files. When a bitmap file in the combo box is selected, it is shown in the image component. **Figure A** shows this component on a form at design time.

Figure A



The PictureSelector component combines the TComboBox and TImage components into a single component.

`TPictureSelector` is derived from `TWinControl`. `TWinControl` provides is the base class for all windowed controls. **Listings A** and **B** show the header and source for `TPictureSelector`.

This component doesn't contain a lot of code. The component is nowhere near what I would call finished, but it does what it is advertised to do and that's all that counts for now. In order for this component to be finished I should, at a minimum, resurface some of the `TComboBox` and `TImage` events. For example, I should have an `OnSelectionChanged` event that simply passes on the combo box's `OnClick` event.

Creating the child components

`TPictureSelector` declares two variables in its `private` section:

```
private:
  TComboBox* FComboBox;
  TImage* FImage;
```

In the component's constructor these two components are dynamically created, given some default values, and positioned. Here's the constructor of `TPictureSelector`:

```
__fastcall TPictureSelector::
  TPictureSelector(TComponent* Owner)
  : TWinControl(Owner)
{
  Width = 240;
  Height = 205;
  FComboBox = new TComboBox(this);
  FComboBox->Left = 0;
  FComboBox->Top = 0;
  FComboBox->Width = 150;
  FComboBox->Style = csDropDownList;
  FComboBox->OnClick = ComboBoxClick;
  FComboBox->Parent = this;
  FImage = new TImage(this);
  FImage->Left = 0;
  FImage->Top = FComboBox->Height + 1;
  FImage->Width = Width;
  FImage->Height =
    Height - FComboBox->Height + 1;
  FImage->Parent = this;
}
```

First the component itself is set to a default size of 240 pixels wide and 200 pixels high. These are somewhat arbitrary values. I knew I would be using the splash images installed with C++Builder and that

the images were 180 by 240 pixels in size. Every component has a default size and this code shows how to set the default size for a component.

Next I create the combo box dynamically and set some of its properties:

```
FComboBox = new TComboBox(this);
FComboBox->Left = 0;
FComboBox->Top = 0;
FComboBox->Width = 150;
FComboBox->Style = csDropDownList;
FComboBox->OnClick = ComboBoxClick;
FComboBox->Parent = this;
```

Notice that I set the `Top` and `Left` properties to 0. The `TPictureSelector` will be the parent of the combo box and, as such, the position 0,0 is the top left corner of the component. I set the `Width` property to an arbitrary value of 150 pixels. I thought the component looked better this way rather than having the combo box extend across the full width of the component.

Now turn your attention to the line of code that assigns the `OnClick` event:

```
FComboBox->OnClick = ComboBoxClick;
```

This code assigns the combo box's `OnClick` event to a function called `ComboBoxClick()`. Here is the `ComboBoxClick()` function:

```
void __fastcall UnlPictureSelector::
  ComboBoxClick(TObject* Sender)
{
  int index = FComboBox->
    Items->IndexOf(FComboBox->Text);
  FImage->Picture->
    LoadFromFile(Strings[index]);
}
```

When the combo box is clicked, the `ComboBoxClick()` function is called and the bitmap associated with the selected combo box item is loaded into the image component. Without providing an internal handler for the combo box's `OnClick` event I would have no way of knowing when the combo box selection changed.

It's important to understand that by using the combo box's `OnClick` event internally, I am making it impossible for the user to access this event in the finished component. It's sort of a moot point because I haven't made the combo box component public so the user has no way of getting to the event anyway.

The solution in this case would be to provide a new event that emulates the combo box's `OnClick` event.

Finally, notice the line of code that assigns the `TPictureSelector` as the combo box's parent:

```
FComboBox->Parent = this;
```

I've mentioned this in past articles, but without this line of code the combo box would never show itself. Forgetting this one line of code has left many new component writers scratching their heads wondering why the child component isn't visible.

The rest of the code in the constructor creates the image component and gives its properties default values. The bulk of the code is positioning and sizing code. I position the `TImage` component on the left edge, and just below the combo box. I size the image component so that it fills the remainder of the `TPictureSelector` component.

You may have noticed that the `TPictureSelector` component does not have a destructor. If you are new to the VCL you might think that I have violated the rules of object oriented programming by not deleting the objects I allocated with operator `new`. I don't need to delete the objects because the VCL will do it for me. When the `TPictureSelector` component is destroyed, the VCL will automatically destroy the `TComboBox` and `TImage` objects.

Window handles and child components

A situation often arises when using aggregate components. In this component I am filling the combo box with a list of strings contained in a `const` variable called `Strings`. Most component writers' first thought is to fill the combo box with strings in the component's constructor. The problem with this approach is that the combo box doesn't have a window handle at this point in the creation process. Attempting to call any methods of the combo box that require a window handle will result in an exception at runtime that says:

```
Control has no parent window.
```

This somewhat cryptic message is telling you that you are attempting to access some portion of the combo box that requires a window handle and that the window handle has not yet been created. The solution is to override the `CreateWnd()` method. When the base class `CreateWnd()` returns, all of the components have been created and the window handle is valid. Here's my overridden `CreateWnd()`:

```
void __fastcall
  TPictureSelector::CreateWnd()
```

```
{
  TWinControl::CreateWnd();
  for (int i=0;i<6;i++)
    FComboBox->Items->Add(Strings[i]);
  FComboBox->ItemIndex = 0;
  ComboBoxClick(0);
}
```

First I call the base class's `CreateWnd()` function. After the base class's `CreateWnd()` function returns I know that I have a valid window handle for the combo box and I can add the strings to it. I also set the `ItemIndex` property to 0 so that the combo box displays the first item when the program runs. Without this step the combo box selection would be blank on program startup. The last line of code calls the `ComboBoxClick()` function to display the bitmap in the Image component.

In the last line of code in the preceding code example I call the `ComboBoxClick` function like this:

```
ComboBoxClick(0);
```

I pass 0 for the `Sender` parameter because I'm not doing anything with `Sender` in the `ComboBoxClick` function. In other words, it doesn't matter one bit what I pass in the `Sender` parameter because `ComboBoxClick` doesn't use the parameter. Some folks would be more comfortable if I had called the function like this:

```
ComboBoxClick(this);
```

`Sender` is a `TObject` pointer so passing 0 is just as good as passing `this` so long as I know that I'm not using the parameter in the `ComboBoxClick` function.

I can just about guarantee that you'll get the "Component has no parent window" exception at some point. Remembering this one tidbit regarding `CreateWnd()` may save you hours of frustration when that time comes.

Exposing child component properties, methods, and events

When writing aggregate components, you have to decide which of the child components' properties, methods, and events you will surface in the aggregate component.

In some cases you won't surface any of the child components' properties, methods, and events. Instead, you would keep them hidden from the user.

In other cases you may surface some or all of the properties, methods, and events in order to allow the user to interact with the individual child components. In fact, you may choose to allow access to the child components in their entirety by making them public or published properties of the aggregate component.

The `TPictureSelector` component doesn't expose the child components, nor does it expose any of their properties, methods, or events. This is due to the fact that this component is a simple example and really doesn't do anything meaningful. The point is that you have to give careful consideration to how the component will be used and expose child component properties, methods, and events as needed.

Conclusion

Creating aggregate components raises issues you don't have to face when writing standard components.

Listing A: *Declaration of the TPictureSelector class*

```
#ifndef PictureSelectorH
#define PictureSelectorH

#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>

const String Strings[] = {
  “handshak.bmp”,
  “factory.bmp”,
  “chemical.bmp”,
  “shipping.bmp”,
  “finance.bmp”
};

class PACKAGE TPictureSelector :public TWinControl
{
private:
  TComboBox* FComboBox;
  TImage* FImage;
protected:
  void __fastcall ComboBoxClick(TObject* Sender);
  virtual void __fastcall CreateWnd();
public:
  __fastcall TPictureSelector(TComponent* Owner);
};
```

```
#endif
```

Listing B: *Definition of the TPictureSelector class*

```
#include <vcl.h>
#pragma hdrstop

#include “PictureSelector.h”
#pragma resource “PictureSelector.res”
#pragma package(smart_init)

__fastcall TPictureSelector::TPictureSelector(
  TComponent* Owner) : TWinControl(Owner)
{
  Width = 240;
  Height = 205;
  FComboBox = new TComboBox(this);
  FComboBox->Left = 0;
  FComboBox->Top = 0;
  FComboBox->Width = 150;
  FComboBox->Style = csDropDownList;
  FComboBox->OnClick = ComboBoxClick;
  FComboBox->Parent = this;
  FImage = new TImage(this);
  FImage->Left = 0;
  FImage->Top = FComboBox->Height;
  FImage->Width = Width;
  FImage->Height = Height - FComboBox->Height;
  FImage->Parent = this;
}

void __fastcall TPictureSelector::CreateWnd()
{
  TWinControl::CreateWnd();
  for (int i=0;i<6;i++)
    FComboBox->Items->Add(Strings[i]);
  FComboBox->ItemIndex = 0;
  ComboBoxClick(0);
}

void __fastcall TPictureSelector::ComboBoxClick(
  TObject* Sender)
{
```

```
  int index =
    FComboBox->Items->IndexOf(FComboBox->Text);
  FImage->Picture->LoadFromFile(Strings[index]);
}

namespace Pictureselector
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] =
      {__classid(TPictureSelector)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

A fancy expandable dialog box

by Mark G. Wiseman

Like some college courses, this article has a prerequisite. Before reading it and using the information contained it, you really need to read the article *A simple expandable dialog box*. This article is also in this issue.

What's so fancy?

Well, the fancy expandable dialog uses more resources than a simple expandable dialog and it's slower. But it's still simple to create and it looks really cool!

The simple expandable dialog just pops into and out of its expanded state. The fancy expandable dialog uses animation to expand and contract.

When its Expanded property is set to true, the dialog expands by first rolling down the bottom of the dialog and then two doors open to reveal the controls of the expanded section.

What's the design?

Let's start off in the IDE by creating a dialog box exactly the same as the simple expandable dialog. Make a note of the bottom panel's `Height` value. In my version, the `Height` is `253`.

Once that's done, drop two more panels on the bottom panel of the dialog. These will be our doors. Set the `Align` property of one of the panels to `alLeft` and the set the other to `alRight`. Now adjust the `Width` property of both panels until the interior edges of the panels just touch—like two closed doors. To make it easier to code, make sure that `Width` has the same value for both panels. In my design the `Width` is `245`.

Now make a note of the panels' `Width` value. Then set the `Width` property of both panels to zero. The panels will disappear (the doors are open).

That takes care of the modifications to the simple dialog.

What's the code?

Listings A and **B** are the source code for the fancy expandable dialog box. You'll notice that the only

significant difference from the code for the simple dialog is the `SetExpanded()` method.

This method first checks to see if it needs to expand or contract the dialog. If it needs to expand the dialog, it first rolls the bottom panel down by increasing its `Height` property one pixel at a time from `1` to `253`. Next it opens the doors by decreasing their `Width` property one pixel at a time from `245` to `0`. `SetExpanded()` contracts the dialog by reversing this process.

To get everything to work smoothly, I call the method, `Application->ProcessMessages()`, for each step of the expansion or contraction. This serves a dual purpose. First, it allows the dialog a chance to redraw itself and keep everything looking nice. Second, it slows the process down just enough to make the animation look good.

As I did in the simple dialog code, I use the trick of setting the dialog's `AutoSize` property to `false` and then to `true` to get it to resize to fit the changing bottom panel.

What's left?

That's all it takes to make a simple expanding dialog fancy. I hope I have expanded your horizons when it comes to creating dialog boxes.

There are a couple more things I'd like to mention.

First, it really isn't good programming practice to use *magic numbers*. These are the numbers `253` and `245` I used for the `Height` and `Width` properties of the panels. I did this to keep this code simple. If you want to make the fancy dialog more versatile, you could calculate these numbers when you first create the dialog in your program.

Second, I could have derived `TFancyExpandableDialog` from `TExpandableDialog`. Changing the `SetExpanded()` method to a virtual method would do the trick. But, then I would have had to add the door panels completely in code and that seemed like too much trouble.

Listing A: *FancyExpandDlg.h*

```
#ifndef FancyExpandDlgH
#define FancyExpandDlgH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
```

```
class TFancyExpandableDialog
  : public TForm
{
  __published:
    TPanel *Panel1;
    TButton *OKBtn;
    TButton *CancelBtn;
    TButton *DetailsBtn;
    TPanel *Panel2;
    TPanel *Panel3;
    TPanel *Panel4;
    void __fastcall DetailsBtnClick(
      TObject *Sender);

  public:
    __fastcall TFancyExpandableDialog(
       TComponent* Owner);

    __property bool Expanded =
      {read = expanded,
       write = SetExpanded};

  private:
    void SetExpanded(bool expand);

    bool expanded;
};


extern PACKAGE
  TFancyExpandableDialog *FancyExpandableDialog;


#endif   // FancyExpandDlgH
```

Listing B: *FancyExpandDlg.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "FancyExpandDlg.h"

#pragma package(smart_init)
```

```
#pragma resource "*.dfm"

TfancyExpandableDialog *FancyExpandableDialog;


__fastcall
TfancyExpandableDialog::TFancyExpandableDialog(
  TComponent* Owner) : TForm(Owner)
{
  expanded = true;
  Expanded = false;
}

void __fastcall
TFancyExpandableDialog::DetailsBtnClick(
  TObject *Sender)
{
  Expanded = !Expanded;
}

void TFancyExpandableDialog::SetExpanded(
  bool expand)
{
  if (expanded == expand) return;

  if (expanded)
  {
    for (int i = 1; i <= 245; i++)
    {
      Panel3->Width = i;
      Panel4->Width = i;
      Application->ProcessMessages();
    }
    for (int i = 253; i >= 0; i--)
    {
      Panel2->Height = i;
      Application->ProcessMessages();
    }
    AutoSize = false;
    AutoSize = true;
  }
  else
  {
    for (int i = 1; i <= 253; i++)
```

```
    {
      Panel2->Height = i;
      Application->ProcessMessages();
    }
    for (int i = 245; i >= 0; i--)
    {
      Panel3->Width = i;
      Panel4->Width = i;
      Application->ProcessMessages();
    }
  }

  expanded = expand;

  DetailsBtn->Caption = expanded
    ? "&Details <<" : "&Details >>";
}
```

A simple expandable dialog box

by Mark G. Wiseman

Expandable dialog boxes can be useful for selectively displaying information to users. In its normal state an expandable dialog box will display a limited amount of information or controls to the user. When the dialog box is expanded, more information or controls are displayed.

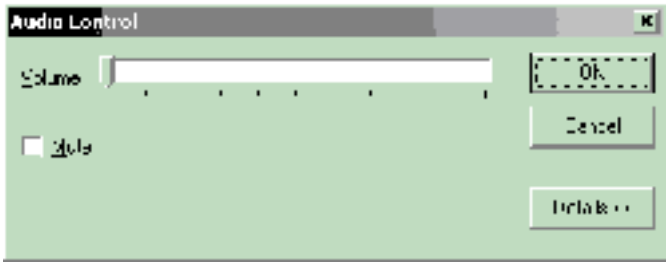
Define it

Microsoft uses the term *Modal Unfolding Secondary Window* to describe and expandable dialog box. This use of the word unfolding is meant to emphasize the fact that the dialog is not resizable in the normal Windows way. The expandable dialog only has two sizes normal (folded) and expanded (unfolded).

Normally, an *unfold* button is used to fold and unfold the dialog. Microsoft recommends appending two "greater than" characters (>>) to the button's label to indicate that the dialog can be unfolded. Two "lesser than" characters are appended to the button's label to show that the dialog has been expanded and that clicking in the button will fold the dialog back to its normal state.

Figure A shows an expandable dialog box in its normal state, with many of the dialog's controls hidden. The dialog can be expanded or unfolded by clicking on the "Details" button.

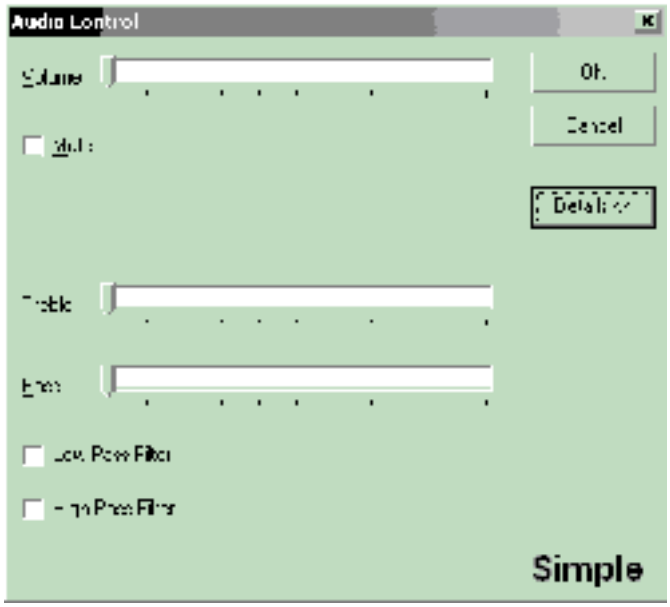
Figure A



Expandable dialog in its normal state

Figure B shows the same dialog box in its expanded state, with all of the dialog's controls available to the user. The dialog can be returned to its normal or folded state by clicking on the "Details" button.

Figure B



Expandable dialog in its expanded state

Build it

Using the VCL, building a simple expandable dialog turns out to be … well, simple. Most of the work can be done in the IDE by dropping a few components on a form and setting a few properties in the object inspector. We will only need to write a few lines of code.

Let's start by creating a new application. Change the `Name` property of the form to "ExpandableDialog". To turn the form into a dialog box, change the form's `BorderStyle` property to `bsDialog`.

Next set the `Height` and `Width` properties to `360` and `400`. This is the expanded size of the dialog.

Drop a `TPanel` component on the form and set its `Align` property to `alTop`. This panel defines the portion of the dialog that will be displayed in the normal or folded state. Clear the panel's `Caption` property and set the `BevelOuter` property to `bvNone`. This makes the panel appear "invisible". Finally, set the panels `Height` property to `167`.

Now drop a second panel onto the form below the first panel. This panel will be the section of the dialog that appears when the dialog is expanded.

We'll use a trick to place this panel. Set the panel's `Align` property to `alBottom`. This will place the panel at the bottom of the form. Again, clear the panel's `Caption` and set `BevelOuter` to `bvNone`. Now drag the top of this panel until it meets the bottom of the first panel.

We used the panel's `Align` property to place the panel at the bottom of the form, but we don't want the panel to always align with the form's bottom, so set the `Align` property back to `alNone`.

Again, the top panel will hold the controls that will always be displayed in the dialog. The bottom panel will hold the controls that will only be display when the dialog is expanded.

Let's finish up by placing three buttons vertically on the right side of the top panel. Change the `Name` of the top button to "OKBtn" and set its `Caption` property to "OK". Also set its `ModalResult` property to `mrOk`.

The `Name` property of the middle button should be "CancelBtn" and its `Caption` should be "Cancel". Set its `ModalResult` property to `mrCancel` and set its `Cancel` property to `true`.

The `ModalResult` property for the first two buttons does a couple of things. When this dialog is displayed using the `ShowModal()` method, these buttons will close the dialog and return the value of the `ModalResult` property of the button clicked. All this is done without having to write an `OnClick` handler for the buttons.

Setting the `Cancel` property to `true` for the middle button tells the VCL to act as if this button was clicked when the escape key is pressed on the keyboard.

The third button is the "unfold" button. We don't need to make any changes to this button's properties.

Code it

Now, let's write a few lines of code. **Listings A** and **B** are the source code for this simple expandable dialog box.

If you'll notice I've added a property called `Expanded` to the header file (**Listing A**). This does exactly what you think it does. Setting `Expanded` to `true` expands the dialog and setting it to `false` returns the dialog to its normal state.

This is accomplished in the `SetExpanded()` method. The trick this method uses is to make the lower panel, `Panel2`, visible or invisible and then forcing the dialog to resize by setting the dialog's `AutoSize` property first to `false` and then to `true`.

The `SetExpanded()` method also takes care of setting the correct `Caption` for the unfold button.

The `OnClick` event of the unfold button simply reverses the value of the dialog's `Expanded` property.

Finally, save the unit containing the dialog as "ExpandDlg.cpp".

As it stands, you can run this application and the dialog will expand and contract. But it won't do much else. So, how do we make this dialog useful?

Reposit it

To allow us to use the expandable dialog in any project, we are going to add it to the IDE's Object Repository. The easiest way to do this is to right click on the dialog and choose "Add to repository…" from the menu. A dialog similar to the one in **Figure C** will appear.

Figure C

The Add To Repository dialog box

Fill in the fields with the appropriate information and click the OK button. Now we can use this dialog in any of our projects.

By the way, I looked it up in my dictionary and *reposit* really is a word.

Use it

Once the dialog has been placed in the Object Repository it very easy to include the dialog in you projects.

While working in the IDE on a project, just click on the File | New… menu item. Select the Dialogs tab and double click on the Expandable Dialog icon. The code and form for the expandable dialog are now in your project. Add the controls you want in the IDE's designer, remembering to place them on the top or bottom panel, as your design requires.

There is a demonstration application on the Bridges Publishing web site that was written in just this way.

Conclude it

Creating a simple expandable dialog box was simple. But what if you want something more exciting and dynamic? What if you want a fancy expandable dialog box? If you do, then read my article, *A fancy expandable dialog box*, in this issue.

Listing A: *ExpandDlg.h*

```
#ifndef ExpandDlgH
#define ExpandDlgH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>


class TExpandableDialog : public TForm
{
  __published:
    TPanel *Panel1;
    TPanel *Panel2;
    TButton *OKBtn;
    TButton *CancelBtn;
    TButton *DetailsBtn;
    void __fastcall DetailsBtnClick(
      TObject *Sender);

  public:
    __fastcall
      TExpandableDialog(TComponent* Owner);

    __property bool Expanded = {
       read = expanded,
       write = SetExpanded};

  private:
    void __fastcall SetExpanded(
      bool expand);

    bool expanded;
};
```

```
extern PACKAGE
  TExpandableDialog *ExpandableDialog;


#endif   // ExpandDlgH
```

Listing B: *ExpandDlg.cpp*

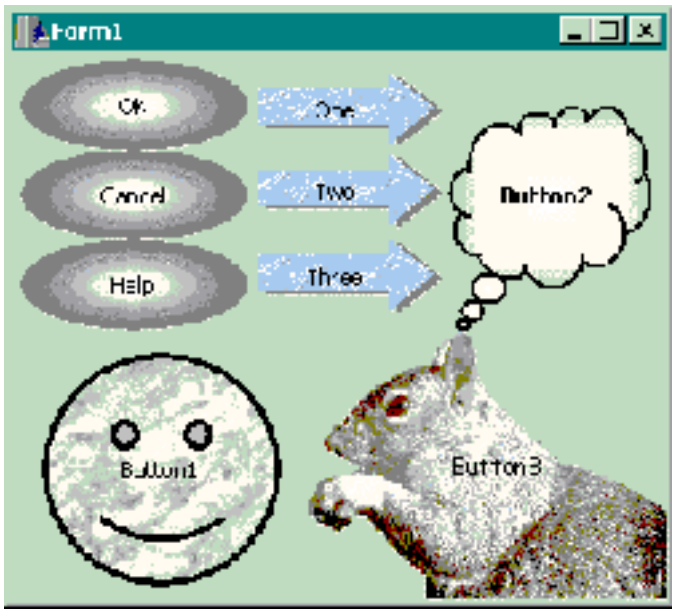
```
#include <vcl.h>
#pragma hdrstop

#include "ExpandDlg.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TExpandableDialog *ExpandableDialog;

__fastcall TExpandableDialog::TExpandableDialog(
  TComponent* Owner) : TForm(Owner)
{
  Expanded = false;
}

void __fastcall TExpandableDialog::
  DetailsBtnClick(TObject *Sender)
{
  Expanded = !Expanded;
}

void __fastcall TExpandableDialog::SetExpanded(
  bool expand)
{
  AutoSize = false;

  expanded = expand;
  Panel2->Visible = expanded;
  DetailsBtn->Caption = Expanded ?
    "&Details <<" : "&Details >>";

  AutoSize = true;
}
```

Custom buttons, part III

by Damon Chandler

In the previous articles of this series, I showed you how to create a button component that supports a color specification and a custom bitmap face. This month, I'll conclude the series by demonstrating how to use a GDI region object to create a button that conforms to the shape of an image. **Figure A** shows some examples.

Figure A



Some custom-shaped buttons.

The GDI region object

A *region* is a special GDI object that's designed to clip the output of a rendering operation. For example, when a window is in the background but only partially obscured, that window still has to paint its visible portions. To this end, the window will still receive the `WM_PAINT` message, but any rendering that it performs to its device context is clipped to the visible portions. Windows uses a region to perform this clipping.

In fact, regions are used to clip more than just drawings; they're also used to define the shape of a window (I'll show you how this is done later). In this way, regions can be used to clip both rendered output and mouse-related input.

There are two main types of regions: *simple* regions and *complex* regions. A simple region is a region with a rectangular shape. This rectangle (or square) can be of virtually any size, even as small as a single pixel. A complex region is a non-rectangular region that's composed of two or more simple regions.

Creating a region from an image

Windows provides many functions for creating simple and complex regions. For example, to create a simple region, you'd use the `CreateRectRgn()` or `CreateRectRgnIndirect()` functions. Alternatively, to create an elliptic (complex) region, you'd use `CreateEllipticRgn()` or `CreateEllipticRgnIndirect()`. Unfortunately, these functions are a bit restrictive because you can create only a limited number of shapes. Although you can use the `CreatePolygonRgn()` function to create any polygonal region, this approach requires that you specify the all of the polygon's vertices. Fortunately, Windows provides the `CombineRgn()` function, which can be used to combine two (simple or complex) regions into a third. With this function, you can build a region of any shape by successively combining smaller (and simpler) regions.

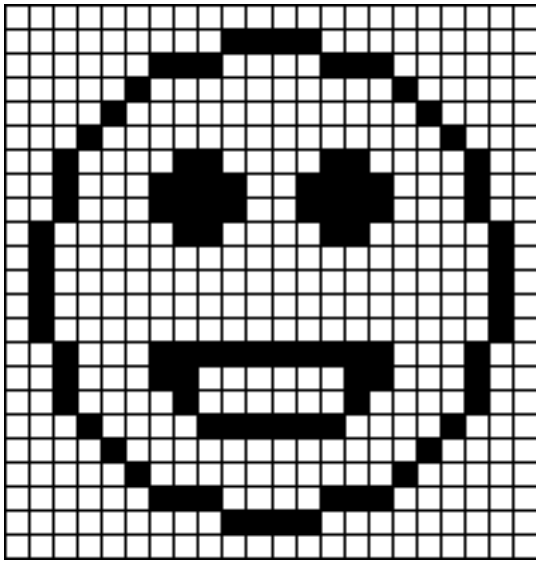
As I mentioned earlier, a region can be as small as a single pixel. This is actually an important requirement because it allows you to create a complex region from multiple pixel-sized regions. These regions can be created based on the pixels of a bitmap. For example, suppose you want to create a region that's shaped like the image in **Figure B**. Assuming that this bitmap is held in a `TImage` object called `Image1`, here's the code that you'd use:

```
Graphics::TBitmap& Bitmap =
  *Image1->Picture->Bitmap;
Bitmap.PixelFormat = pf24bit;
const SIZE SImage =
  {Bitmap.Width, Bitmap.Height};

HRGN hTotalRgn = NULL;

// scan the pixels of the bitmap...
for (int y = 0; y < SImage.cy; ++y)
{
  const RGBTRIPLE* pRow =
    static_cast<RGBTRIPLE*>(
      Bitmap.ScanLine[y]);
  for (int x = 0; x < SImage.cx; ++x)
  {
    // if the pixel is black...
    if (pRow[x].rgbtRed == 0 &&
        pRow[x].rgbtGreen == 0 &&
        pRow[x].rgbtBlue == 0)
    {
```

```
      // create a pixel-sized region
      const HRGN hPixelRgn =
        CreateRectRgn(x, y, x+1, y+1);

      if (hTotalRgn)
      {
        // combine the regions
        CombineRgn(
          hTotalRgn, hTotalRgn,
          hPixelRgn, RGN_OR);
        // free the pixel-sized region
        DeleteObject(hPixelRgn);
      }
      else hTotalRgn = hPixelRgn;
    }
  }
}
// use the region...

// later...
DeleteObject(hTotalRgn);
```

This code constructs an image-based region one pixel at a time. Specifically, it scans the bitmap (in `Image1`), looking for black-colored pixels. Each time a black-colored pixel is encountered, the `CreateRectRgn()` function creates a pixel-sized region (`hPixelRgn`). This region is then combined with the composite region (`hTotalRgn`) by using the `CombineRgn()` function.

Figure B



A simple black and white image whose pixels can be used to define the shape of a region.

As you can see from this code, the `CreateRectRgn()` function takes four parameters that, together, specify the corners of the rectangle. Upon success, this function returns a handle to the rectangular region (of type `HRGN`). The `CombineRgn()` function also takes four parameters. The second and third parameters specify the handles to the two regions that you want to combine. This composite region is copied to the (pre-existing) region whose handle is specified as the function's first parameter. The fourth parameter specifies how the two regions should be combined (e.g., `RGN_OR` = union; `RGN_AND` = intersection; `RGN_XOR` = non-overlapping union).

A faster alternative

I've just shown you how to create an image-shaped region by using the `CreateRectRgn()` and `CombineRgn()` functions. Unfortunately, depending on the number of pixels that you use to define the region, this approach can be unbearably slow. Combining two regions is a cheap operation, but the multiple calls to the `CreateRectRgn()` function impose a severe bottleneck.

Fortunately, there's a way to create a complex region via one function call; specifically, a call to the `ExtCreateRegion()` function. Here how this function is declared:

```
HRGN ExtCreateRegion(
  CONST XFORM* lpXform,
  DWORD nCount,
  CONST RGNDATA* lpRgnData);
```

The `lpRgnData` parameter specifies a pointer to a `RGNDATA` structure (discussed next) that conveys information about the rectangles that will define the region. Accordingly, the `nCount` parameter specifies the number of bytes in this `RGNDATA` structure. The `lpXform` data member can be used to specify a coordinate transformation (e.g., a scaling). In our case, we'll pass `NULL` as this parameter to indicate no transformation.

The `RGNDATA` structure is defined as follows:

```
typedef struct _RGNDATA {
  RGNDATAHEADER rdh;
  char          Buffer[1];
} RGNDATA, *PRGNDATA;
```

The `Buffer` parameter is a placeholder for a variable-length array of rectangles that define the region. The `rdh` data member is of type `RGNDATAHEADER` and is defined like so:

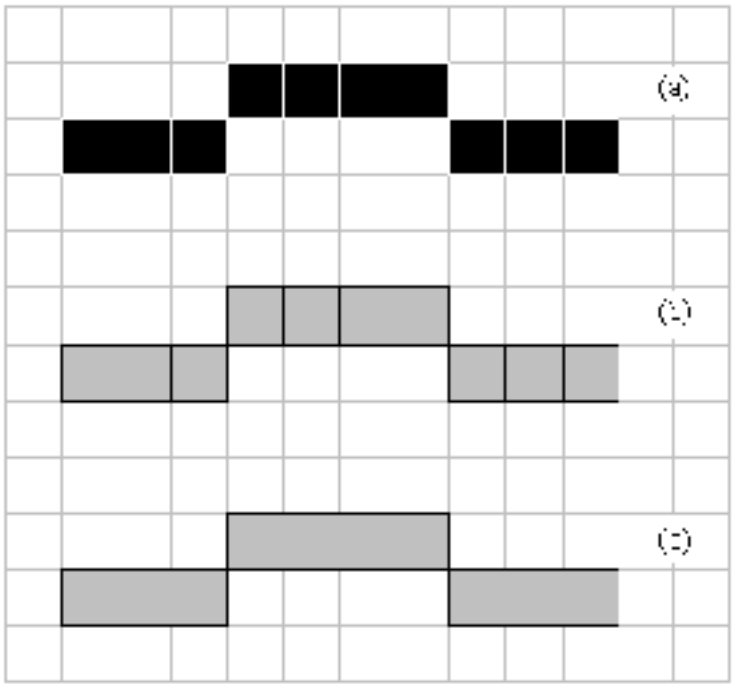
```
typedef struct _RGNDATAHEADER {
  DWORD dwSize;
  DWORD iType;
  DWORD nCount;
  DWORD nRgnSize;
  RECT  rcBound;
} RGNDATAHEADER, *PRGNDATAHEADER;
```

The `dwSize` parameter specifies the number of bytes that the structure uses—i.e., `sizeof(RGNDATAHEADER)`. The `iType` parameter specifies the shape of the primitives that are contained in the `RGNDATA::Buffer` member. Because current versions of Windows support only rectangles, the `iType` parameter must be set to `RDH_RECTANGLES`. Accordingly, the `nCount` data member specifies how many rectangles make up the region. The `nRgnSize` and `rcBound` data members are ignored by the `ExtCreateRgn()` function, so you can simply set both of these to zero.

Using ExtCreateRgn()

Now that we've gone over the specifics of the `ExtCreateRgn()` function's parameters, let's see how the function is actually used. The first step is to initialize the rectangles that'll be used to define the shape of the region. Unfortunately, this is somewhat of a cumbersome procedure because, on NT-based systems, the rectangles that define the region cannot share a vertical boundary—i.e., you can't have two horizontally adjacent rectangles (see **Figure C**). In order to accommodate this restriction, you must initialize your array of rectangles such that all horizontally adjacent rectangles are encompassed by one larger "parent" rectangle. This scheme is illustrated in **Figure C**, which shows a magnified section of the bitmap in **Figure B** along with the incorrect and correct way to define the rectangles.

Figure C



Defining the rectangles from the pixels of a bitmap: (a) two scan lines of a bitmap (10 total pixels); (b) the incorrect way to define the rectangles (this results in 10 rectangles); (c) the correct way to define the rectangles (this results in three rectangles).

Of course, before you can allocate memory for the rectangles, you'll need to know how many rectangles are required. The following function demonstrates how to count these rectangles by scanning the pixels (`pPixels`) of a 24-bpp bitmap for those whose color is not equal (within `tolerance`) to `transparent_color`. These are assumed to be opaque—i.e., the pixel defines part of the region. Here's the code:

```
#include <math.h>
unsigned int CountNumRects(
 const unsigned char* pPixels,
 const SIZE& img_size,
 COLORREF transparent_color,
 unsigned char tolerance)
{
  // NOTE: the pPixels buffer is assumed
  // to be aligned to a 32-bit boundary
  //
  // compute the width in bytes
  const unsigned int bytes_per_line =
     (((img_size.cx * 24) + 31)
      & ~31) >> 3;

  // scan the pixels to count the
```

```
  // number of required rectangles...
  unsigned int num_rects = 0;
  for (int y = 0; y < img_size.cy; ++y)
  {
    const RGBTRIPLE* pScanLine =
     reinterpret_cast<const RGBTRIPLE*>(
        pPixels + (y * bytes_per_line));

    bool start_new_rect = true;
    for (int x = 0; x < img_size.cx;++x)
    {
      const int dR =
        pScanLine[x].rgbtRed -
        GetRValue(transparent_color);
      const int dG =
        pScanLine[x].rgbtGreen -
        GetGValue(transparent_color);
      const int dB =
        pScanLine[x].rgbtBlue -
        GetBValue(transparent_color);

      const bool opaque =
        sqrt(dR*dR + dG*dG + dB*dB)
          > tolerance;

      // if we're on an opaque pixel
      if (opaque)
      {
        if (start_new_rect)
        {
          ++num_rects;
          start_new_rect = false;
        }
      }
      // we're on a transparent pixel
      else start_new_rect = true;
    }
  }

  // return the number of RECTs required
  return num_rects;
}
```

Now that you have a method for counting the number of rectangles that the region requires, the next step

is to allocate and initialize an array of RECT structures. The following function demonstrates how this is done:

```
PRECT InitRects(
  unsigned int num_rects,
  const unsigned char* pPixels,
  const SIZE& img_size,
  COLORREF transparent_color,
  unsigned char tolerance)
{
  // make room for the RECTs
  PRECT pRects = new RECT[num_rects];
  PRECT pCurrentRect = pRects - 1;

  // compute the width in bytes
  const unsigned int bytes_per_line =
    (((img_size.cx * 24) + 31)
     & ~31) >> 3;

  // scan the image...
  for (int y = 0; y < img_size.cy; ++y)
  {
    const RGBTRIPLE* pScanLine =
     reinterpret_cast<const RGBTRIPLE*>(
        pPixels + (img_size.cy - y - 1)
        * bytes_per_line);

    bool start_new_rect = true;
    for (int x = 0; x < img_size.cx;++x)
    {
      const int dR =
        pScanLine[x].rgbtRed -
        GetRValue(transparent_color);
      const int dG =
        pScanLine[x].rgbtGreen -
        GetGValue(transparent_color);
      const int dB =
        pScanLine[x].rgbtBlue -
        GetBValue(transparent_color);

      const bool opaque =
        sqrt(dR*dR + dG*dG + dB*dB)
        > tolerance;
```

```
      // if we're on an opaque pixel
      if (opaque)
      {
        if (start_new_rect)
        {
          start_new_rect = false;
          // grab a pointer to the next
          // rectangle in the array and
          // initialize it
          ++pCurrentRect;
          pCurrentRect->left = x;
          pCurrentRect->top = y;
          pCurrentRect->right = x + 1;
          pCurrentRect->bottom = y + 1;
        }
        else
        {
          // increase the width
          // of the current RECT
          ++(pCurrentRect->right);
        }
      }
      // we're on a transparent pixel
      else start_new_rect = true;
    }
  }
  // return a pointer to the RECTs
  return pRects;
}
```

Once the array of `RECT`s is allocated and initialized, the final step is to allocate a `RGNDATA` structure, initialize its header, copy a chunk of rectangles from the array of `RECT`s to the `RGNDATA::Buffer` data member, and then pass the initialized `RGNDATA` structure to the `ExtCreateRgn()` function. It's important that you copy only a limited number of rectangles to the `RGNDATA::Buffer` data member because the `ExtCreateRgn()` function is a bit flaky on Windows 9x. Specifically, the function will fail if you pass it too many rectangles (e.g., more than a few thousand). The following function demonstrates how to use the `CountNumRects()` and `InitRects()` utility functions, and how to use the `ExtCreateRgn()` function in an iterative fashion—i.e., passing the function a maximum batch-size of 1000 `RECT`s. Each batch-sized region is combined with the composite result by using the `CombineRgn()` function. Here's the code:

```
#include <cassert>
#include <stdexcept>
```

```
HRGN RegionFromDIBSection(
  HBITMAP hDIBSection,
  COLORREF transparent_color,
  unsigned char tolerance)
{
  // total region
  HRGN hTotalRgn = NULL;

  // get info about the DIB section
  BITMAP bmp;
  const size_t size = GetObject(
     hDIBSection, sizeof(BITMAP), &bmp);

  // verify the format of the DIB section
  assert(size == sizeof(BITMAP));
  assert(bmp.bmBitsPixel == 24);
  assert(bmp.bmBits != NULL);

  // ensure the orientation (assume
  // a bottom-up DIB section)
  bmp.bmHeight = abs(bmp.bmHeight);

  // grab a byte-pointer to the pixels
  const unsigned char* pPixels =
    static_cast<unsigned char*>
      (bmp.bmBits);

  // store the image dimensions
  const SIZE img_size = {
    bmp.bmWidth, bmp.bmHeight};

  // count the number of rectangles
  unsigned int num_rects =
    CountNumRects(
      pPixels, img_size,
      transparent_color, tolerance);

  // allocate and initialize the RECTs
  PRECT pRects = InitRects(
    num_rects, pPixels, img_size,
    transparent_color, tolerance);
  try
  {
    // now that the buffer of rectangles
```

```
// is initialized, let's create the
// region in batches of 1000 RECTs

// allocate memory for the batch
const int rects_per_batch =
  min(1000U, num_rects);
const std::size_t buffer_size =
   sizeof(RGNDATAHEADER) +
   rects_per_batch * sizeof(RECT);
unsigned char* pBuffer =
  new unsigned char[buffer_size];
try
{
  memset(pBuffer, 0, buffer_size);

  // intialize a RGNDATA structure
  LPRGNDATA pRgnData =
    reinterpret_cast<LPRGNDATA>
      (pBuffer);
  pRgnData->rdh.dwSize =
    sizeof(RGNDATAHEADER);
  pRgnData->rdh.iType =
    RDH_RECTANGLES;

  int iRect = 0;
  while (num_rects > 0)
  {
    pRgnData->rdh.nCount = min(
      static_cast<unsigned int>
        (rects_per_batch),
        num_rects);
    num_rects -=
      pRgnData->rdh.nCount;

    // copy the memory from pRects
    // to the RGNDATA's buffer
    // (i.e., fill the batch)
    memcpy(
      pRgnData->Buffer,
      reinterpret_cast<PBYTE>
        (pRects + iRect),
      pRgnData->rdh.nCount *
      sizeof(RECT));
    iRect += pRgnData->rdh.nCount;
```

```
        // create/combine the region(s)
        const HRGN hBatchRgn =
          ExtCreateRegion(
            NULL, buffer_size, pRgnData);
        if (!hBatchRgn)
        {
          throw std::runtime_error(
            "!hBatchRgn");
        }
        if (hTotalRgn)
        {
          CombineRgn(
            hTotalRgn, hTotalRgn,
            hBatchRgn, RGN_OR);
          DeleteObject(hBatchRgn);
        }
        else hTotalRgn = hBatchRgn;
      }
    }
    catch (...)
    {
       // clean up
       delete [] pBuffer;
       throw;
    }
    // clean up
    delete [] pBuffer;
  }
  catch (...)
  {
     // clean up
     delete [] pRects;
     throw;
  }
  // clean up
  delete [] pRects;

  // return the total region
  return hTotalRgn;
}
```

Although this code is a bit cryptic, the `RegionFromDIBSection()` function is nearly instantaneous,

even for moderately sized images. This is a significant speed improvement over the previous approach that used `CreateRectRgn()`. In fact, the size of the image isn't much of an issue for buttons, because you'll rarely need such a large button. Because most application use several dozen buttons however, the speed factor is crucial.

You might be wondering why I used a raw DIB section bitmap instead of using a `TBitmap` object. I did this for portability with all versions of C++Builder, but primarily because of C++Builder version 1. C++Builder version 1 supports only device-dependent bitmaps (DDBs), which provide no direct pixel access.

You now have the framework for creating an image-shaped region; and, believe me, this is the most cumbersome part of the process. In the sections that follow, I'll briefly discuss how to integrate this code with the `TBitmapButton` class, and I'll show you how to make the button take the shape of the region.

Extending TBitmapButton

Listing A contains the declaration of the `TBitmapButton` class, which I've modified from last month's version to support an image-based region. Specifically, I've added three properties: `AutoShape`, which is a Boolean that determines whether or not the button conforms to the shape of the bitmap (held in the first position of `Faces_`); `TransColor`, which implicitly defines the pixels of the bitmap that you don't want included the region; and `TransTolerance`, which specifies how close (in Cartesian space) each color must be to `TransColor` in order to be considered transparent. Accordingly, each of these properties has a write-access (i.e., "setter") method that updates not only the associated property, but the region as well. Here's how these methods are defined:

```
void __fastcall TBitmapButton::
  SetAutoShape(bool Value)
{
  if (AutoShape_ != Value)
  {
    if (Faces_->Empty)
    {
      AutoShape_ = false;
    }
    else
    {
      AutoShape_ = Value;
    }
    DoShapeButton();
    DoRedrawButton();
  }
}
```

```
void __fastcall TBitmapButton::
  SetTransColor(TColor Value)
{
  if (TransColor_ != Value)
  {
    TransColor_ = Value;
    DoShapeButton();
    DoRedrawButton();
  }
}

void __fastcall TBitmapButton::
  SetTransTolerance(short Value)
{
  if (TransTolerance_ != Value)
  {
    TransTolerance_ = Value;
    DoShapeButton();
    DoRedrawButton();
  }
}
```

As I just mentioned, each of these methods updates its corresponding property, then updates the region (if necessary) by using the `DoShapeButton()` method, and then updates the display by using the `DoRedrawButton()` method. I'll discuss the `DoShapeButton()` method next; the `DoRedrawButton()` method simply punts its work to the `RedrawWindow()` API function.

Setting the button's shape

There's one final (and very important) detail that needs to be addressed. Specifically, I've shown you how to create an image-based region, but I have yet to discuss how to make the button take the shape of this region. This task is fairly effortless—you simply call the `SetWindowRgn()` function. The `TBitmapButton` class calls this function from within its `DoShapeButton()` method, which is defined as follows:

```
void __fastcall TBitmapButton::
  DoShapeButton()
{
  if (AutoShape_ && !Faces_->Empty)
  {
    // size the button to the first face
    SetBounds(
```

```
    Left, Top,
    Faces_->Width / NumFaces_,
    Faces_->Height);
}
else
{
  // restore the original shape
  SetWindowRgn(Handle, NULL, false);
  return;
}

// don't shape at design time
if (ComponentState.
      Contains(csDesigning))
{
  return;
}

// create a 24-bpp DIB section
HBITMAP hDIBSection =
  DoCreateDIBSection();
if (!hDIBSection)
{
  throw EInvalidGraphicOperation(
    "Failed to create DIB section.");
}

// create a region from the bitmap
const HRGN hRgn =
  RegionFromDIBSection(
    hDIBSection,
    ColorToRGB(TransColor_),
    TransTolerance_);
if (!hRgn)
{
  throw EInvalidGraphicOperation(
    "Failed to create region.");
}
try
{
  // destroy the DIB section
  DeleteObject(hDIBSection);
  hDIBSection = NULL;
```

```
    // shape the button to the region
    if (!SetWindowRgn(
          Handle, hRgn, false))
    {
      // report the error
      throw EInvalidGraphicOperation(
        "Failed to set window region.");
    }
  }
  catch (...)
  {
    // clean up on error
    if (hDIBSection)
    {
      DeleteObject(hDIBSection);
    }
    DeleteObject(hRgn);
    throw;
  }
}
```

You can see from this code that the `SetWindowRgn()` function takes three parameters: a handle to the window whose shape you want to define, a handle to the region that specifies this shape, and a Boolean that indicates whether or not the window should be redrawn. Notice that I destroy the region (via a call to the `DeleteObject()` GDI function) only if the `SetWindowRgn()` function fails. This is an important step because the window takes ownership of the region if the `SetWindowRgn()` is successful—i.e., if it returns non-zero. Also notice that you can pass `NULL` as the `SetWindowRgn()` function's second parameter to restore the window's original shape.

Finally, notice that the `DoShapeButton()` method calls a method named `DoCreateDIBSection()`. I created this latter method to retrieve a handle to a DIB section by using either a `TBitmap` object (for C++Builder versions 3 and greater) or the `CreateDIBSection()` GDI function (for C++Builder version 1). Although I won't present this method here, the source code that accompanies this article contains its implementation.

Conclusion

In this series of articles, we examined several approaches to creating a customized button. Here, I've demonstrated how to define the button's shape by using a bitmap and a region. You can download the source code for the `TBitmapButton` class—along with a sample application—at **www.bridgespublishing.com**.

Listing A: *Declaration of the modified TBitmapButton class*

```
class PACKAGE TBitmapButton : public TColorButton
{
public:
  __fastcall TBitmapButton(TComponent* Owner);
  __fastcall ~TBitmapButton();

__published:
  __property Graphics::TBitmap* Faces =
    {read = Faces_, write = SetFaces};
  __property int NumFaces =
    {read = NumFaces_, write = SetNumFaces};
  __property bool NoFocusRect =
    {read = NoFocusRect_, write = SetNoFocusRect};
  __property bool AutoShape =
    {read = AutoShape_, write = SetAutoShape};
  __property TColor TransColor =
    {read = TransColor_, write = SetTransColor,
     default = clFuchsia};
  __property short TransTolerance =
    {read = TransTolerance_,
     write = SetTransTolerance};

protected:
  DYNAMIC HPALETTE __fastcall GetPalette();
  virtual void __fastcall DoDrawButtonFace(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonText(
    const TOwnerDrawState& state);
  virtual void __fastcall DoShapeButton();
  virtual HBITMAP __fastcall DoCreateDIBSection();
  virtual void __fastcall DoRedrawButton();

private:
  Graphics::TBitmap* Faces_;
  int NumFaces_;
  bool NoFocusRect_;
  bool AutoShape_;
  TColor TransColor_;
  short TransTolerance_;

  void __fastcall SetFaces(
    Graphics::TBitmap* Value);
```

```
  void __fastcall SetNumFaces(int Value);
  void __fastcall SetNoFocusRect(bool Value);
  void __fastcall SetAutoShape(bool Value);
  void __fastcall SetTransColor(TColor Value);
  void __fastcall SetTransTolerance(short Value);
};
```

Persistent image lists

by Damon Chandler

All components that descend from the `TPersistent` class have the ability to be saved to and loaded from a file. (This streaming ability is primarily what makes the `TPersistent` class unique from the `TObject` class.) In this article, I'll show you how to exploit this feature to easily save and load an image list to and from a file.

Saving an image list

The VCL provides a simple function that you can use to save any `TComponent` to a file: `WriteComponentResFile()`. Here's how it's declared:

```
void __fastcall WriteComponentResFile(
  const AnsiString FileName,
  TComponent* Instance);
```

The `WriteComponentResFile()` function takes two parameters: an `AnsiString` that specifies the name of the file to save the component to, and a pointer to the `TComponent` instance that you want to save. Here's an example that uses the `WriteComponentResFile()` function to save a `TImageList` to a file:

```
void __fastcall SaveImageList(
   AnsiString filename,
   TImageList* ImageList)
{
  // write ImageList to a file
   WriteComponentResFile(
    filename, ImageList);
}
```

Loading an image list

As you might have guessed, there is a complementary function that can be used to load a `TComponent` instance from a file: `ReadComponentResFile()`:

```
TComponent* __fastcall
  ReadComponentResFile(
```

```
    const AnsiString FileName,
    TComponent* Instance);
```

The `ReadComponentResFile()` function also accepts two parameters. The first parameter specifies the name of the file from which to load the component. The second parameter specifies a pointer to an instance of the component into which you want the stored data read. Here's an example that uses the `ReadComponentResFile()` function to load a `TImageList` object from a file:

```
#include <memory>
TImageList* __fastcall LoadImageList(
  TComponent* Owner,AnsiString filename)
{
  // create a new TImageList object
  std::auto_ptr<TImageList>
    ImageList(new TImageList(Owner));

  // read the data into ImageList
  ReadComponentResFile(
    filename, ImageList.get());

  // return a pointer to ImageList
  return ImageList.release();
}
```

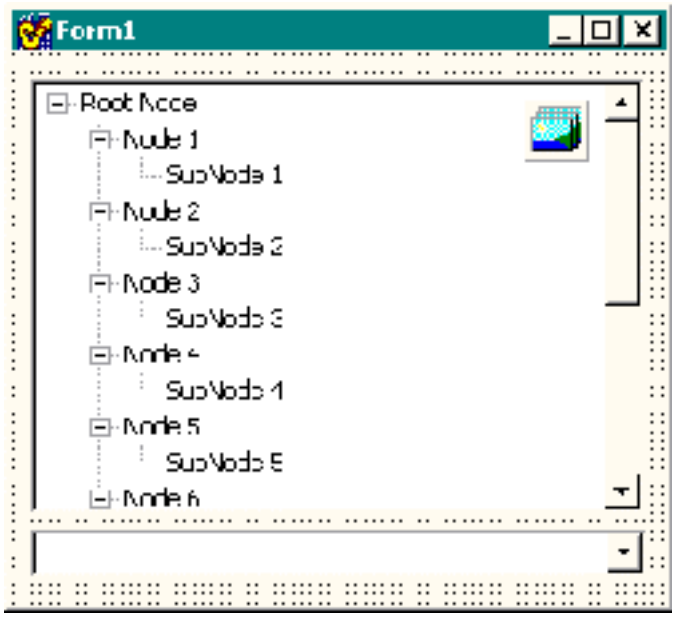
An example

Let's work through an example that uses the `LoadImageList()` function. **Figure A** depicts a form that contains a combo box (`ComboBox1`), a tree view (`TreeView1`), and an image list (`ImageList1`). The goal is to change the icons that the tree view uses (i.e., the icons held in `ImageList1`) depending on which item is selected in the combo box. The combo box simply contains the strings "Style 1", "Style 2", and "Style 3"). Assuming that you've already used the `SaveImageList()` function to save three `TImageList` objects to three files: "imglist_style1.dat", "imglist_style2.dat", and "imglist_style3.dat", here's the code that you'd place in `ComboBox1`'s `OnChange` event handler:

```
void __fastcall TForm1::
  ComboBox1Change(TObject *Sender)
{
  AnsiString filename;
  switch (ComboBox1->ItemIndex)
  {
    case 0:
```

```
      filename = "imglist_style1.dat";
      break;
    case 1:
      filename = "imglist_style2.dat";
      break;
    case 2:
      filename = "imglist_style3.dat";
      break;
  }

  // disassociate the image list
  TreeView1->Images = NULL;

  // delete the current image list
  delete ImageList1; ImageList1 = NULL;

  // load the new image list
  ImageList1 =
    LoadImageList(this, filename);

  // associate the image list
  TreeView1->Images = ImageList1;
}
```

Figure A



A form with an image list, a tree-view, and a combo box. The contents of the image list will change depending on which item is selected in the combo box.

Conclusion

The VCL streaming system makes saving and loading an image list fairly simple. There are, however, a couple of caveats that you should be aware of. First, because the `ReadComponentResFile()` function uses the `FindUniqueName()` function, the `Name` of the `TImageList` will change each time it's loaded. If you use code that relies on the `Name` of your `TImageList`, you'll have to restore the name manually. In addition, if you have an `OnChange` event handler assigned to your image list, you'll have to reassign it after calling the `LoadImageList()` function.

I should mention that Windows does provide its own mechanism for loading and saving an image list to and from a file; namely, you can use the `ImageList_Write()` and `ImageList_Read()` API functions. These functions aren't as user-friendly as `WriteComponentResFile()` and `ReadComponentResFile()` however, because they require the use of an OLE document. (Please feel free to email me if you need an example of using the `ImageList_Write()` and `ImageList_Read()` functions.)

Custom buttons, part II

by Damon Chandler

Last month, I showed you how to create an owner-drawn button; and we worked through a specific example of creating a colored-button component. In this month's article, I'll discuss two things: (1) how to add a glyph to the button; and (2) how to define the button's face by using a custom bitmap.

Adding a glyph

Listing A contains the declaration of the `TColorButton` class, which I've changed from last month's version by adding support for a glyph. Specifically, I added a property called `Glyphs` that specifies a pointer to a `TImageList` object, and a property called `GlyphIndex` that specifies the index of the button's glyph within the image list. (Also notice that I added a `Canvas` property so that descendant classes can access the private `Canvas_` member.) The `Glyphs` and `GlyphIndex` properties are supported via the private `Glyph_` and `GlyphIndex_` members, which are initialized in the class constructor like so:

```
__fastcall TColorButton::
  TColorButton(TComponent* Owner)
    : TButton(Owner),
    ColorLo_(clBtnShadow),
    ColorHi_(clBtnHighlight),
    Canvas_(new TCanvas()),
    draw_as_default_(false),
    GlyphIndex_(0), Glyphs_(NULL)
{
}
```

The `Glyph_` and `GlyphIndex_` members are used from within the `DoDrawButtonGlyph()` method, which—as you can guess—is used to draw the button's glyph. Here's the code for one version of that method:

```
void __fastcall TColorButton::
  DoDrawButtonGlyph(
    const TOwnerDrawState& state)
{
  if (Glyphs_ && GlyphIndex_ >= 0)
  {
    // compute the width of the
```

```
    // caption (in pixels)
    Canvas_->Font = Font;
    const int text_width =
      Canvas->TextWidth(Caption) + 2;

    // compute the correct position at
    // which to draw the glyph
    TPoint PGlyph = Point(
      (Width - text_width -
       Glyphs_->Width) / 2.0,
      (Height -
       Glyphs_->Height + 2) / 2.0
      );

    // offset the glyph if the
    // button is pushed
    if (state.Contains(odSelected))
    {
      PGlyph.x += 1;
      PGlyph.y += 1;
    }

    // render the glyph...
    Glyphs_->Draw(
      Canvas_.get(), PGlyph.x, PGlyph.y,
      GlyphIndex_, Enabled);
  }
}
```

Because `Glyphs_` is a pointer to a `TImageList` object, drawing the button's glyph is simple, even if the button is disabled. Recall that the last parameter of the `TImageList::Draw()` method is a Boolean that specifies whether the image should be rendered as enabled (`true`) or disabled (`false`). Unfortunately, however, the `Draw()` method will always use `clBtnShadow` and `clBtnHighlight` for the disabled glyph's shadows and highlights, respectively. This is fine if your button's color is `clBtnFace`, but it looks weird if you use most other colors. Moreover, in some versions of C++Builder, the `TImageList::Draw()` method accepts only four parameters. Because of these limitations, let's examine an alternative technique for drawing a disabled glyph. The following function demonstrates how to use masking to do so:

```
void __fastcall DrawDisabledGlyph(
    TCanvas& Canvas,
    const TPoint& PDraw,
    TImageList& ImageList,
```

```
    int index,
    TColor ColorShadow,
    TColor ColorHilite)
{
  static const int MASKPAT = 0x00E20746;

  // STEP 1: Create the mask bitmap
  const SIZE SImg = {
    ImageList.Width, ImageList.Height};
  std::auto_ptr<Graphics::TBitmap>
    MaskBitmap(new Graphics::TBitmap());
  MaskBitmap->Monochrome = true;
  MaskBitmap->Width = SImg.cx;
  MaskBitmap->Height = SImg.cy;

  const HDC hDCSrc =
    MaskBitmap->Canvas->Handle;
  PatBlt(
    hDCSrc, 0, 0, SImg.cx, SImg.cy,
    WHITENESS
    );
  ImageList.Draw(MaskBitmap->Canvas,
    0, 0, index, true);

  // STEP 2: Render the highlights
  Canvas.Brush->Color = ColorHilite;
  HDC hDCDst = Canvas.Handle;
  SetTextColor(
    hDCDst, ColorToRGB(clWhite));
  SetBkColor(
    hDCDst, ColorToRGB(clBlack));
  BitBlt(hDCDst, PDraw.x + 1,
    PDraw.y + 1, SImg.cx, SImg.cy,
    hDCSrc, 0, 0, MASKPAT);

  // STEP 3: Render the shadows
  Canvas.Brush->Color = ColorShadow;
  hDCDst = Canvas.Handle;
  SetBkColor(
    hDCDst, ColorToRGB(clBlack));
  BitBlt(hDCDst, PDraw.x, PDraw.y,
    SImg.cx, SImg.cy, hDCSrc, 0, 0,
    MASKPAT);
}
```

The `DrawDisabledGlyph()` function works by using a mask bitmap and the `BitBlt()` GDI function—specifying the `MASKPAT` ternary raster operation—to render the disabled glyph's highlights and shadows transparently (i.e., without disturbing the background).

With the `DrawDisabledGlyph()` function defined, we can now modify the `DoDrawButtonGlyph()` method accordingly:

```
void __fastcall TColorButton::
  DoDrawButtonGlyph(
    const TOwnerDrawState& state)
{
  if (Glyphs_ && GlyphIndex_ >= 0)
  {
    // compute the width of the
    // caption (in pixels)
    Canvas_->Font = Font;
    const int text_width =
      Canvas->TextWidth(Caption) + 2;

    // compute the correct position at
    // which to draw the glyph
    TPoint PGlyph = Point(
      (Width - text_width -
       Glyphs_->Width) / 2.0,
      (Height -
       Glyphs_->Height + 2) / 2.0);

    // offset the glyph if the
    // button is pushed
    if (state.Contains(odSelected))
    {
      PGlyph.x += 1;
      PGlyph.y += 1;
    }

    // render the glyph...
    if (!Enabled ||
        state.Contains(odDisabled))
    {
      DrawDisabledGlyph(
        *(Canvas_.get()),
        Point(PGlyph.x, PGlyph.y),
```

```
        *Glyphs_, GlyphIndex_,
        ColorLo, ColorHi);
    }
    else
    {
      Glyphs_->Draw(Canvas_.get(),
        PGlyph.x, PGlyph.y, GlyphIndex_,
        true);
    }
  }
}
```

The next task is to modify the `CNDrawItem()` method so that the `DoDrawButtonGlyph()` method will be called. Here's the code for that:

```
void __fastcall TColorButton::
  CNDrawItem(TMessage& Msg)
{
  // grab a pointer to the DRAWITEMSTRUCT
  const DRAWITEMSTRUCT* pDrawItem =
    reinterpret_cast<DRAWITEMSTRUCT*>
      (Msg.LParam);

  // store the current state of the DC
  SaveDC(pDrawItem->hDC);
  // bind Canvas_ to the target DC
  Canvas_->Handle = pDrawItem->hDC;
  try
  {
    // extract the state flags...
    TOwnerDrawState state;
    // if the button has keyboard focus
    if (pDrawItem->itemState & ODS_FOCUS)
    {
      state = state << odFocused;
    }
    // if the button is pushed
    if (pDrawItem->itemState &
        ODS_SELECTED)
    {
      state = state << odSelected;
    }
    // if the button is disabled
```

```
    if (pDrawItem->itemState &
        ODS_DISABLED)
    {
      state = state << odDisabled;
    }

    // draw the button's face
    DoDrawButtonFace(state);

    // draw the button's glyph
    DoDrawButtonGlyph(state);

    // draw the button's text
    DoDrawButtonText(state);
  }
  catch (...)
  {
    // clean up
    Canvas_->Handle = NULL;
    RestoreDC(pDrawItem->hDC, -1);
  }
  // clean up
  Canvas_->Handle = NULL;
  RestoreDC(pDrawItem->hDC, -1);

  // reply TRUE
  Msg.Result = TRUE;
}
```

Notice that this version of `CNDrawItem()` is nearly identical to that of last month's article. Here I've simply placed a call to `DoDrawButtonGlyph()` between the calls to `DoDrawButtonFace()` and `DoDrawButtonText()`. It's important that the `DoDrawButtonGlyph()` method is called before `DoDrawButtonText()` because it's from within the latter method that the selection rectangle is drawn.

There's one last modification that needs to be made. Namely, the `DoDrawButtonText()` method needs to be changed so that the button's caption and its glyph don't overlap:

```
void __fastcall TColorButton::
  DoDrawButtonText(
    const TOwnerDrawState& state)
{
  if (Caption.Length() == 0) return;
```

```
  RECT RText = {0, 0, Width, Height};
  SetBkMode(
    Canvas_->Handle, TRANSPARENT);

  // if the button has a glyph
  if (Glyphs_ && GlyphIndex_ >= 0)
  {
    RText.left += Glyphs_->Width + 6;
  }

  // other code from before...
}
```

And, in case you're wondering, here's the code for the `SetGlyphs()` and `SetGlyphIndex()` methods:

```
void __fastcall TColorButton::
  SetGlyphIndex(int Value)
{
  if (GlyphIndex_ != Value)
  {
    GlyphIndex_ = Value;
    InvalidateRect(Handle, NULL, FALSE);
  }
}

void __fastcall TColorButton::
  SetGlyphs(TImageList* Value)
{
  if (Glyphs_ != Value)
  {
    Glyphs_ = Value;
    InvalidateRect(Handle, NULL, FALSE);
  }
}
```

That takes care of adding support for a glyph. The rest of `TColorButton`'s methods are the same as before. **Figure A** depicts some `TColorButton` objects with glyphs.

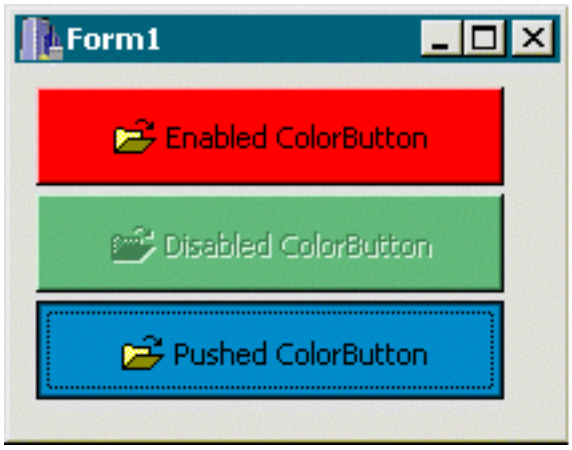


Figure A: *A few TColorButton objects with glyphs.*

Using a custom face

Recall from last time that the bulk of the code for the `TColorButton` class was for drawing the button's face in its various states (e.g., normal, pushed, disabled, etc.). That code was placed within the `DoDrawButtonFace()` method, which drew a standard push button in a user-specified color. For some user interfaces, however, a standard push button might look awkward, even if the button is in a matching color. Although you could create a `TColorButton` descendant class and override the `DoDrawButtonFace()` method (in which you draw the button to match your UI), this route suffers from two major limitations: First, it's a hassle, to say the least; and more importantly, depending on the complexity of your drawing code, it might be inefficient.

Suppose that you override the `DoDrawButtonFace()` method in a `TColorButton` descendant class; and within this method, you place some code to render a 3-D elliptic button that uses some fancy ray tracing. Although your code might be relatively fast, I argue that it's probably not well suited for an owner-drawn button. Remember, the `DoDrawButtonFace()` method is called whenever the button receives the `CN_DRAWITEM` message. This message is sent whenever some aspect of the button needs to be redrawn. This means that your drawing code will be invoked each time the button is uncovered by another window, each time the button is pushed, each time the button's `Enabled` property is changed, each time the button's font is changed, etc. Unless your drawing code is extremely efficient, you'll be using a fair number of CPU cycles, just to draw this *one* button.

Instead of calling your drawing code each time the button needs to be redrawn, it makes sense to call your code only once, storing the result in a bitmap object. Better still, there are a multitude of applications that can be used to create custom button images, why not use one of them? (ZPaint 1.4 is one such application; see **www.steffengerlach.de/freeware/**)

In the sections that follow, I'll show you how to create a `TColorButton` descendant class—which I've called `TBitmapButton`—that allows you to specify the bitmaps that you want the button to use for its normal, pushed, and disabled faces.

The TBitmapButton class

Listing B contains the declaration of the `TBitmapButton` class, which extends the `TColorButton` class by providing three new properties: `Faces`, `NumFaces`, and `NoFocusRect`. These properties are supported via the private `Faces_`, `NumFaces_`, and `NoFocusRect_` members, respectively. These members are initialized in the class constructor like so:

```
__fastcall TBitmapButton::
  TBitmapButton(TComponent* Owner)
    : TColorButton(Owner),
    NumFaces_(2), NoFocusRect_(false)
{
  Faces_ = new Graphics::TBitmap();
}
```

As I'll discuss next, the `Faces` and `NumFaces` properties work just like the `Glyph` and `NumGlyphs` properties of the `TSpeedButton` class. The `NoFocusRect_` member—which I'll get to later—specifies whether the button should display a selection rectangle when it has keyboard focus. Here are the definitions of the `SetFaces()`, `SetNumFaces()`, and `SetNoFocusRect()` methods:

```
void __fastcall TBitmapButton::
  SetFaces(Graphics::TBitmap* Value)
{
  if (Faces_ != Value)
  {
    Faces_->Assign(Value);
    InvalidateRect(Handle, NULL, TRUE);
  }
}

void __fastcall TBitmapButton::
  SetNumFaces(int Value)
{
  if (NumFaces_ != Value)
  {
    NumFaces_ = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}

void __fastcall TBitmapButton::
```

```
  SetNoFocusRect(bool Value)
{
  if (NoFocusRect_ != Value)
  {
    NoFocusRect_ = Value;
    InvalidateRect(Handle, NULL, FALSE);
  }
}
```

Drawing the faces

The `Faces` property is a pointer to a `TBitmap` object that holds a composite image of the button's faces in its normal, pushed, and disabled states (side-by-side, in that order; see **Figure B**). Accordingly, the `NumFaces` property specifies the number of faces that this bitmap contains. The `NumFaces` property is actually important because, when later you draw the button's face, you'll need to know which portion of `Faces_` to render to match the button's current state. Here's the definition of the `DoDrawButtonFace()` method, which demonstrates how this is done:

```
void __fastcall TBitmapButton::
  DoDrawButtonFace(
    const TOwnerDrawState& state)
{
  // if no custom images have been set...
  if (Faces_->Empty)
  {
    // draw a normal push button
    TColorButton::
      DoDrawButtonFace(state);
  }
  // otherwise...
  else
  {
    TRect RSource = Rect(0, 0, 0, 0);

    // if the button is disabled
    if (!Enabled ||
        state.Contains(odDisabled))
    {
      switch (NumFaces_)
      {
        case 1:
        {
```

```
      RSource = Rect(
        0, 0, Faces_->Width,
        Faces_->Height);
      break;
    }
    case 2:
    {
      RSource = Rect(
        0, 0, Faces_->Width / 2,
        Faces_->Height);
      break;
    }
    case 3:
    {
      RSource = Rect(
        2 * Faces_->Width / 3, 0,
        Faces_->Width, Faces_->Height
        );
      break;
    }
  }
}
// otherwise, if the button is pushed
else if (state.Contains(odSelected))
{
  switch (NumFaces_)
  {
    case 1:
    {
      RSource = Rect(
        0, 0, Faces_->Width,
        Faces_->Height);
      break;
    }
    case 2:
    {
      RSource = Rect(
        Faces_->Width / 2, 0,
        Faces_->Width,Faces_->Height
        );
      break;
    }
    case 3:
```

```
        {
          RSource = Rect(
            Faces_->Width / 3, 0,
            2 * Faces_->Width / 3,
            Faces_->Height);
          break;
        }
      }
    }
    // otherwise (if normal)
    else
    {
      RSource = Rect(
        0, 0, Faces_->Width / NumFaces_,
        Faces_->Height);
    }

    // preserve colors
    SetStretchBltMode(
      Canvas->Handle, COLORONCOLOR);
    // render the face
    Canvas->CopyRect(ClientRect,
      Faces_->Canvas, RSource);
  }
}
```

Notice that the bulk of this code serves to define the local `RSource` variable, which is later passed to the `TCanvas::CopyRect()` method to specify which portion of `Faces_` to draw. As I just mentioned, this portion depends on the number of face images (`NumFaces_`) that the composite bitmap (`Faces_`) contains.

Figure B depicts a few `TBitmapButton` objects along with the bitmap that defines each button's faces (placed in a `TImage` object next to each button). Because the face-image of the first button (`BitmapButton1`) contains only one image, `BitmapButton1`'s `NumGlyphs` property is set to 1. Likewise, `NumGlyphs` is set to 2 and 3, respectively, for `BitmapButton2` and `BitmapButton3`.

Figure B: *Three TBitmapButton objects with custom bitmap faces.*

Suppressing the focus rectangle

You might have noticed from **Figure B** that the second button (`BitmapButton2`) uses a face-image that conflicts with the standard selection rectangle. In cases like this, you'd probably want to prevent the selection rectangle from being drawn; this is where the `NoFocusRect` property comes into play.

Recall from the definition of the `TColorButton::DoDrawButtonText()` method that the selection rectangle is rendered (if the button has keyboard focus) after the button's caption is drawn. To suppress this selection rectangle, you simply augment the `DoDrawButtonText()` method and remove the `odFocused` enumerator, like so:

```
void __fastcall TBitmapButton::
  DoDrawButtonText(
    const TOwnerDrawState& state)
{
  // draw the caption...
  TOwnerDrawState new_state(state);
  TColorButton::DoDrawButtonText(
    NoFocusRect_ ?
    new_state >> odFocused : new_state);
}
```

Finishing up

There's one method of the `TBitmapButton` class that I haven't mentioned yet: `GetPalette()`. The `GetPalette()` method is first introduced in the `TControl` class. By overriding this method, a descendant class can punt the palette-related work to the `TControl` class, which will ensure that—on displays that support only 256 or fewer colors—the system palette will be modified appropriately when your application is displayed. Here's the code for the `GetPalette()` method, which simply returns a handle to the logical palette that the `Faces_` bitmap uses (if any):

```
HPALETTE __fastcall TBitmapButton::
  GetPalette()
{
  if (!Faces_->Empty)
  {
    return Faces_->Palette;
  }
```

```
  return TColorButton::GetPalette();
}
```

Further customization

There are a few things that you might want to add to the `TColorButton` and `TBitmapButton` classes. Specifically, if you need to place the button's glyph at a location other than the left side of the text, you'll need to modify the `TColorButton::DoDrawButtonGlyph()` and `DoDrawButtonText()` methods accordingly. Also, you might want to extend the `TBitmapButton::DoDrawButtonFace()` method to support a fourth face that would be used when the button's `Default` property is `true`. In addition, you'll likely want to use the `TCustomImageList::RegisterChanges()` method in order to receive a notification when the underlying image list of the button's `Glyphs_` property is modified. Lastly, I encourage you to add a few events—such as `OnDrawFace`, `OnDrawGlyph`, `OnDrawText`, and `OnPostDraw`—that can be used to further tailor the appearance of the button on a per-instance basis. You never know what type of customization may be required in the future; these events will save a great deal of work farther down the road.

Conclusion

I've shown you how to create a button component that supports a glyph and a custom face-image. In fact, by using a separate image to define the button's face, the `TBitmapButton` component will conform to most any user interface. You can even provide separate image-sets (i.e., skins) that your application can load at run time to allow your users to change the look of the UI. (The code that accompanies this article—available at **www.bridgespublishing.com**—demonstrates this process; **Figure C** depicts the sample application.) Next month, I'll show you how to use a GDI region object to give your button a custom shape.

Figure C: *Sample application in Windows, MacOS, and KDE modes.*

Listing A: ***Declaration of the modified TColorButton class***

```
#include <memory>
class PACKAGE TColorButton : public TButton
{
public:
  __fastcall TColorButton(TComponent* Owner);
  __property TCanvas* Canvas = {read = GetCanvas};

__published:
  __property TColor Color =
    {read = GetColor, write = SetColor};
  __property TColor ColorLo =
    {read = ColorLo_, write = SetColorLo,
     default = clBtnShadow};
  __property TColor ColorHi =
```

```
    {read = ColorHi_, write = SetColorHi,
     default = clBtnHighlight};
  __property int GlyphIndex =
    {read = GlyphIndex_, write = SetGlyphIndex};
  __property TImageList* Glyphs =
    {read = Glyphs_, write = SetGlyphs};

protected:
  // inherited methods
  virtual void __fastcall CreateParams(
    TCreateParams& Params);
  virtual void __fastcall SetButtonStyle(
    bool ADefault);

  // introduced methods
  virtual void __fastcall DoDrawButtonFace(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonText(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonGlyph(
    const TOwnerDrawState& state);

private:
  TColor ColorLo_;
  TColor ColorHi_;
  std::auto_ptr<TCanvas> Canvas_;
  bool draw_as_default_;
  int GlyphIndex_;
  TImageList* Glyphs_;

  TCanvas* __fastcall GetCanvas();
  TColor __fastcall GetColor();
  void __fastcall SetColor(TColor Value);
  void __fastcall SetColorLo(TColor Value);
  void __fastcall SetColorHi(TColor Value);
  void __fastcall SetGlyphIndex(int Value);
  void __fastcall SetGlyphs(TImageList* Value);

  MESSAGE void __fastcall CNDrawItem(
    TMessage& Msg);
  MESSAGE void __fastcall WMLButtonDblClk(
    TMessage& Msg);
  MESSAGE void __fastcall CMFontChanged(
    TMessage& Msg);
```

```
  MESSAGE void __fastcall CMEnabledChanged(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CN_DRAWITEM, TMessage, CNDrawItem)
  MESSAGE_HANDLER(
    WM_LBUTTONDBLCLK, TMessage, WMLButtonDblClk)
  MESSAGE_HANDLER(
    CM_FONTCHANGED, TMessage, CMFontChanged)
  MESSAGE_HANDLER(
    CM_ENABLEDCHANGED, TMessage, CMEnabledChanged)
END_MESSAGE_MAP(TButton)
};
```

Listing B: *Declaration of the TBitmapButton class*

```
class PACKAGE TBitmapButton : public TColorButton
{
public:
  __fastcall TBitmapButton(TComponent* Owner);
  __fastcall ~TBitmapButton();

__published:
  __property Graphics::TBitmap* Faces =
    {read = Faces_, write = SetFaces};
  __property int NumFaces =
    {read = NumFaces_, write = SetNumFaces};
  __property bool NoFocusRect =
    {read = NoFocusRect_, write = SetNoFocusRect};

protected:
  DYNAMIC HPALETTE __fastcall GetPalette();
  virtual void __fastcall DoDrawButtonFace(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonText(
    const TOwnerDrawState& state);

private:
  Graphics::TBitmap* Faces_;
  int NumFaces_;
  bool NoFocusRect_;
```

```
  void __fastcall SetFaces(
    Graphics::TBitmap* Value);
  void __fastcall SetNumFaces(int Value);
  void __fastcall SetNoFocusRect(bool Value);
};
```

Events and callback functions

By David Bridges

Any veteran Windows programmer should be familiar with callback functions. In this article, I will cover how to use callback functions and explain how VCL events are just a special kind of callback function. Once you understand how VCL events work, you'll be able to add events to your own classes for added functionality, even if your classes aren't derived from the VCL.

Using callbacks

The most common way to have access to a function in C++ is to either declare the function, or include a header file that declares the function. This tells the compiler that the function is defined somewhere, but doesn't say exactly where it is. The linker is responsible for finding the actual function's address and linking it into the final code. Generally, the code for the function will be in a .LIB file or in another source file in the project.

Using function pointers, you can call any function that is in the process's memory space, without having to declare it or having the source code or .LIB file. All you need to know is the address of the function, and what type of parameters it expects. Unlike other types, function pointers do not require the usual syntax for pointers. To take the address of a function, you don't have to use the & operator. Instead, just leave off the parentheses and use only the function name. When calling the function via its address, it does not have to be dereferenced. The following code shows an example:

```
typedef void (*MyFunctionType)();

void MyFunction()
{
  MessageBox(
    NULL, "Hello", "Example", MB_OK);
}

void CallFunction()
{
  // Create a function pointer and
  // assign the address of a function.
  MyFunctionType func = MyFunction;
  func();
}
```

Callback functions are generally functions that are written by a programmer with the sole purpose of being called from somewhere else—for example, the operating system. The way this is done is to supply the function's address to the outside program, which then calls it at some later time. This is what's referred to as the "callback".

One common use of callbacks in the Windows API is for enumeration functions. For example, suppose you want to load a combo box with the names of all of the fonts available on the machine. One possible solution would be to have a Windows API function that takes all of the available font names and loads them into a specified combo box. This would work fine, but what if we wanted to add the font names to some other kind of control, or put them into a user-defined memory structure? This is one situation where callback functions are particularly useful. The example below shows how to write a callback function that will perform a certain action for every font name that exists on the system.

```
extern "C" int CALLBACK AddFontToList(
  const LOGFONT* lf, const TEXTMETRIC* tm,
  unsigned long iType, long lData)
{
  TMyForm* form =
    reinterpret_cast<TMyForm*>(lData);
  form->edtFonts->Lines->Add(
    lf->lfFaceName);
  return 1;
}

void __fastcall TMyForm::FormCreate(
  TObject *Sender)
{
  EnumFontFamilies(Canvas->Handle,
    NULL, AddFontToList, (LPARAM)this);
}
```

There’s actually a much easier way to do this—you could just use the following line:

```
edtFonts->Lines->Assign(Screen->Fonts);
```

The example is meant to show that by the use of a callback function, Windows can transfer a set of information to another program without any knowledge of what the program will do with that information.

Caller ID

Many callback functions require a parameter that represents an object or context. This is commonly known as the "user data" parameter. In the `AddFontToList` function, this is the `lData` parameter. This parameter allows us to pass an application-defined value to the enumeration function along with the address of the callback function. Every time Windows calls the `AddFontToList` function, it passes back the same value for `lData` that was passed to `EnumFontFamilies`. The value of this parameter is totally up to the programmer. In this case, it contains the address of the form object.

The __closure keyword

In this example, it would be much easier if we could just specify a member function of the class as the callback function – that way, we wouldn't have to keep passing the `this` pointer around. Unfortunately, C++ does not allow callbacks into class member functions. Fortunately, Borland C++ Builder provides a way to do this using the `__closure` keyword extension. The example below shows how to use this keyword to enable callbacks into a class member function.

```
typedef void (__closure *AddStringEvent)
  (AnsiString sValue);

// This function supplies a list of
// strings by using a callback function
void ListEmployees(
  AddStringEvent AddString)
{
  AddString("Alex");
  AddString("Brian");
  AddString("David");
  // ...
}

void TMyForm::AddNextEmployeeName(
  AnsiString sName)
{
  edtEmployees->Lines->Add(sName);
}

void __fastcall TMyForm::FormCreate(
  TObject *Sender)
{
  ListEmployees(AddNextEmployeeName);
}
```

In standard C++, the reason you can't use a class member function as a callback function is because in C (and C++), a pointer is a pointer, and they are all the same size (32 bits in the Win32 world). In order to use a class member function, you must have two pointers: one that has the address of the object, and the other which contains the address of it's member function, or method. The `__closure` keyword passes the object's `this` pointer as a "hidden parameter". This address is actually stored along with the function address in the pointer. If you use the `sizeof()` operator on a closure function pointer, you will see that it's size is 64 bits—32 bits for the object's address and 32 bits for the function address.

Events

So far, we've seen how to use callback functions to transfer sets of information, and how to use the `__closure` keyword in order to use class member functions as callbacks. Now we'll see how to create and handle our own events.

In the VCL, events are exposed as properties. Events are no different from other properties, except that their type is a pointer to a function. Whenever the event occurs, the function whose address is contained in the property gets called. If the event is `NULL`, no function is called.

The simplest type of event in C++ Builder is `TNotifyEvent`. Here is the definition of `TNotifyEvent`:

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender);
```

For a function to be of the `TNotifyEvent` type, it must be a class member function that has the same return type and parameters, and must be declared as `__fastcall`. For example:

```
void __fastcall MyForm::edtNameChange(
  TObject* Sender)
{
  // ...
}
```

The IDE automatically writes functions such as this and assigns them to the events of controls through the Object Inspector. However, you can also assign these events programatically. Instead of double-clicking on the control's `OnChange` event, you could instead just write the member function above and assign it with this code:

```
void __fastcall TMyForm::FormCreate(
  TObject *Sender)
{
  edtName->OnChange = edtNameChange;
}
```

Conclusion

Listing A shows the code for a component that uses events. The `ValueChangeEvent` type is a function that has two arguments, `Sender` and `NewValue`. Most VCL events have `TObject* Sender` as their first parameter, and this new event type follows that convention. The `Sender` parameter is very useful because it tells the callback function what object generated the event.

The class `MyComponent` is a very simple one—it has one property and one event. Whenever the `Value` property is changed, the `OnChange` event is generated. This class is derived from `TComponent`, but it doesn't have to be. You can use properties and events in your own classes even if they aren't derived from VCL classes.

Listing A: *A component with an event.*

```
// Define the event type.
typedef void (__closure *ValueChangeEvent)
```

```
  (TObject* Sender, AnsiString NewValue);

class MyComponent : public TComponent
{
  private:
    AnsiString       FValue;
    ValueChangeEvent FOnChange;

  public:

    __fastcall MyComponent(TComponent* owner)
      : TComponent(owner)
    {
      // Assign a null handler by default
      OnChange = NULL;
    }

    void SetValue(AnsiString sNewValue)
    {
      FValue = sNewValue;

      // If a handler is assigned to
      // this event, call it.
      if (OnChange) {
        OnChange(this, FValue);
      }
    }

    // 'Value' property
    __property AnsiString Value =
      {read=FValue, write=SetValue};

    // 'OnChange' event
    __property ValueChangeEvent OnChange =
      {read=FOnChange, write=FOnChange};
};

// This is the OnChange event handler
void TMyForm::ComponentValueChanged(
  TObject* Sender, AnsiString sNewValue)
{
  MessageBox(Handle, sNewValue.c_str(),
    "Value Changed", MB_OK);
}

void __fastcall TMyForm::FormCreate(
  TObject *Sender)
{
```

```
  // Create the object
  MyComponent* c = new MyComponent(this);

  // Assign the handler for the 'OnChange' event.
  c->OnChange = ComponentValueChanged;

  // Setting the 'Value' property
  // will generate the 'OnChange' event.
  c->Value = "New Value";

  delete c;
}
```

Sharing code between database projects: a look at queries

by Willem Semmelink

Have you ever had to modify code because the project had to be ported from BDE objects to Interbase or ADO objects? Almost each new client-server data base I work on has a report engine, lookup table maintenance forms, and user validation code. Much of this code could have been shared between projects if I planned the usage of my queries a bit better. Developers using C++Builder 5 Professional can choose between BDE and IBExpress components, and C++Builder 5 Enterprise developers have the additional option to use ADO components. Peeping over the shoulder of a Delphi 6 developer, I foresee that new C++Builder releases may probably also include a `dbExpress` query as Delphi 6 does.

The problem is this: the three existing queries use different property names and even different data types for similar tasks. Since `TDataSet` is their first common parent, polymorphic operations on query-specific properties are difficult. `TDataSet` for example, lacks the SQL property. Therefore it is difficult to generalize abstract behavior for a query in a true O-O manner. When a source code file is ported from BDE to IBExpress, or to ADO (or dbExpress, or whatever else), we don't want to manually search-and-replace our source code to change each `TQuery` to `TADOQuery` or to `TIBQuery`, resolve property names and their data types, and so on.

By manually changing code, we run the risk of introducing errors. My first attempt at solving this was to define C-style `#define` macros and `typedef's` that would determine which of the three environments we are using. Thereafter, I will mention a few things about my most recent idea to solve the problem with an object-oriented solution.

For the C-style macros, I created the following definitions in a file that I called `DBDefsU.h`:

```
#ifdef USING_BDE    // BDE Definition
  #include <dbtables.hpp>
  //Query
  typedef TQuery TFischerQuery;
  #define DB_LINK_PROPERTY DatabaseName
  #define DB_LINK_VALUE
    DataModule1->Database1->DatabaseName
#elif USING_ADO    // ADO Definition
  #include <ADOdb.hpp>
  //Query
  typedef TADOQuery TFischerQuery;
  #define DB_LINK_PROPERTY Connection
```

```
  #define DB_LINK_VALUE
    DataModule1->ADOConnection1
#elif USING_INTERB // IB Definition
  #include <Ibquery.hpp>
  //Query
  typedef TIBQuery TFischerQuery;
  #define DB_LINK_PROPERTY Database
  #define DB_LINK_VALUE
    DataModule1->IBDatabase1
#else #error DBDefsU.h Missing USING_BDE
  or USING_ADO or USING_INTERB
#endif //end DB check
```

In a new project, I add a data module and to that, I add either a BDE `TDatabase` component, `TADOConnection` or an Interbase `TIBDatabase` component (remember also to set up the parameters to connect to the database). I assume that only one connection object is present per project, and that a project exclusively uses either ADO, IB, or BDE. In the header file of the data module, I specify either one of the following three definitions:

```
#define USING_BDE
#define USING_ADO
#define USING_INTERB
```

Thereafter follows an `#include` to add the definitions file:

```
#include "DBDefsU.h"
```

Wherever the data module is included, the definitions of DBDEFS.H will be available. Then, I instantiate a `TFischerQuery`, which is `typedef`ed in DBDEFS.H as a `TQuery`, a `TADOQuery`, or a `TIBQuery`. I called it `FischerQuery` since that is the name of the company I work for. I then set the database connection property, and the query can be used.

An implementation of the macros in a program is illustrated below:

```
TFischerQuery* qry =
  new TFischerQuery(NULL);
qry->DB_LINK_PROPERTY = DB_LINK_VALUE;
qry->SQL->Clear();
qry->SQL->Add("select * from customer");
qry->Open();
for (int i=0;i<=qry->RecordCount;i++) {
  ListBox1->Items->Add(
```

```
    Qry->FieldByName(
      "CUSTNO")->AsString + " " +
      Qry->FieldByName(
        "COMPANY")->AsString);
  qry->Next();
}
```

If we used the `USING_BDE` directive in the data module, the compiler will replace the above code to the following:

```
TQuery* qry = new TQuery (NULL);
qry->DatabaseName =
  DataModule1->Database1->DatabaseName;
// etc
```

Suppose we need to re-use the above code in an InterbaseExpress project, and then replace the `TDatabase` in the data module with a `TIBDatabase` and a `TIBTransaction` object. Set the `DefaultTransaction` property of the database object to point to the `IBTransaction`. Check that the `TIBDatabase` name is the same as the name the previous listing and change the lines in the data module of the new project to:

```
#define  USING_INTERB
#include "DBDefsU.h"
```

When we re-compile, the macros will be replaced as if we had written the following code:

```
TIBQuery* qry = new TIBQuery(NULL);
qry->Database = DataModule1->IBDatabase;
// etc
```

Developers that have the C++Builder Enterprise version may replace the `Database` object in the data module with a `TADOConnection` object. Ensure that the name of the connection object is the same as the definition in the earlier macro and change the data module to:

```
#define  USING_ADO
#include "DBDefsU.h"
```

The ADO macro definitions will then be replaced during compiling as:

```
TADOQuery * qry = new TADOQuery(NULL);
qry->Connection =
  DataModule1->ADOConnection1;
```

```
// etc
```

Transaction management

The macro definitions also provide for transaction management, so you can use code like the following:

```
q_SomeQuery->SQL->Add(
  String("UPDATE USER SET password = '") +
    Encrypt(strNewPassword) +
    "' WHERE userid = '" +
    strCurrentUser + "'");
try {
  DB_BEGIN_TRANSACTION;
  q_SomeQuery->ExecSQL();
  DB_COMMIT_TRANSACTION;
} catch (exception &E) {
  DB_ROLLBACK_TRANSACTION;
  ShowMessage(E.Message);
}
```

Parameters

The greatest challenge in solving the problem of porting lies in handling the `Parameters` property, because the ADO implementation differs significantly from the BDE and IBExpress interfaces.

For simple queries, it is sufficient to insert parameters in the query, preceded by colon. The difference in the parameter property name is handled by the following definition in the `DBDefsU` file:

```
#define DB_PARAM_PROPERTY params
```

For ADO, the property name is `Parameters`. It can then be used as follows:

```
qry ->SQL->Add("select Company from "
  "Customer where Custno = :cno")
qry->DB_PARM_PROPERTY->ParamByName(
  "cno")->Value = "15";
```

The parameter challenge comes in when we use stored procedures and we want to create parameters dynamically using the `CreateParam` method. For BDE or IBExpress, we require two lines of code:

```
pr->Params->CreateParam(
  ftInteger,"USER_ID",ptInput);
pr->ParamByName("USER_ID")->AsInteger =
  fUserID;
```

Porting to a `TADOStoredProc`, we find that the above is handled by one ADO method, and a change was made to the data type of the direction parameter:

```
Pr->Parameters->CreateParameter(
 "USER_ID",ftInteger,pdInput,0,fUserID);
```

Perhaps we can solve the above with a parameterized `#define`, but let's leave the macros here, and look at an object-oriented solution that I thought about after solving our commercial software problems with the preceding solution.

An object-oriented solution

I experimented using an adaptor class pattern. This pattern is described in *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley Longman Inc. ISBN 0-201-63361-2). It is also known as a wrapper class.

`TTargetQuery` is defined as a pure virtual class. There, we define all public properties and methods that will be needed. The implementation of the class is found in one of three classes that inherit from `TTargetQuery`. Each of these hides the VCL classes in a private data member, so that it is not visible from the outside. The member functions of the adapter classes map the function calls to the correct functions of their respective adaptee classes.

This configuration is complicated significantly by the fact that we will most probably inherit `TTargetQuery` itself from `TDataSet`, so that standard data-aware components can use the new query classes. Since many of `TDataSet`'s functions also need to be mapped to the adaptee classes, we will end up writing a hefty bunch of code before we can do anything useful with the class. This ended up in so much code that I abandoned the idea and implemented `TTargetQuery` without inheriting it from anything. I have not yet tested this in a real-life application, so I use my attached example code only to illustrate the concept. I briefly tested `TABDEQuery` on the `BCDEMOS` database. The other two adapter classes are not yet tested.

The code is still too much to discuss in detail in an article, but the following comments should be sufficient. The class hierarchy allows us to implement a common interface, and then we can swap between the three classes rather easily:

In a new project, I add a data module and to that, add either a BDE `TDatabase` component, or an ADO

TADOConnection or an Interbase TIBDatabase component. Remember to set up the connection to the database as well. In the header file of the data module, specify either one of the following three definitions (and comment out the other two):

```
typedef TABDEQuery TFischerQuery;
// typedef TAIBQuery  TFischerQuery;
// typedef TAADOQuery TFischerQuery;
```

Using the query is very simple:

```
TTargetQuery* qry =
  new TFischerQuery(this,0);
qry->SQL->Clear();
qry->SQL->Add("select * from customer");
qry->Open();
for (int i=0;i<=qry->RecordCount;I++) {
  ListBox1->Items->Add(
    qry->FieldByName("CUSTNO")->AsString
    + " " + qry->FieldByName("COMPANY")
      ->AsString);
  qry->Next();
}
```

I define all new types as TTargetQuery, so that I have a generic type, but I create them as TFischerQuery, so that the typedef can be used to switch all my code to another sub-type of TTargetQuery.

If we suppose that the BDE is used, the compiler will render code as if we typed:

```
TTargetQuery* qry =
  new TABDEQuery(this,0);
```

A simple change in the typedef will change the query to be compiled as either a TAIBQuery or a TAADOQuery. No further change in the source code should be necessary, besides creating the correct DB connection object, and setting the connection parameters for the DB.

Since we are on that topic, you will notice from the previous listing that the constructor takes two arguments, and that I removed the line that assigns the database connection. I implement the query's connection through the second parameter of the constructor. Each adaptor class has a static TList that contains connection pointers. Each class also has a static function to add new connections. We set the connection once only in the application (e.g. in the constructor of the data module):

```
TABDEQuery::AddConnection(Database1);
```

If we had an IB database or ADO, we would have used one of the following:

```
TAIBQuery::AddConnection(IBDatabase1);
TAADOQuery::
  AddConnection(ADOConnection1);
```

Since these connections are added to a static list, they are available to all instances of the class. To allow the user to choose which connection to implement, I pass an integer to the constructor of the query. This integer is the `ItemIndex` of the static list, and it uniquely identifies the connection to be used. The integer will not pose any type conversion problems when we change from one query type to another, as the connection property itself did.

Again, my greatest obstacle was the ADO `Parameters` property. I implemented a `TParams` to operate in parallel with the `Parameters` property; so all classes work with the same type. Each time the `Param` is written, I write the `Parameter`, and each time a `Param` is read, I get its value from the `Parameter` property. The performance of my ADO example can be improved, though.

Conclusion

The preceding article illustrates that it is possible to port code between projects that implement different database connection strategies, without having to change more than three or four lines of code in the data module. This is of course only valid if you can refrain from the temptation to use the visual designer to create queries—this article's info will be rather worthless if you have queries pasted on VCL forms.

C++Builder 5 users that wish to use ADO should download a patch called `BCB5ADOUpgrade1.exe` from Borland's web site to fix an error in ADO queries.

Willem Semmelink is a senior developer at Fischer Consulting in Pretoria, South Africa. The company specializes in transport-related software, but at present, Willem is involved with a project for the upcoming South African Census, using C++Builder and Oracle. Willem can be contacted at willem@fischerint.co.za

Components: build, beg, or buy?

by Kent Reisdorph

Part of the power of C++Builder is the VCL and its components. Need a split window? Drop a few components on your form. Need a file open dialog? Drop a component on your form. Components make your life easier, plain and simple. The more components you have, the more efficient you are.

The VCL, however, is far from complete. There are many programming needs that are not covered by the VCL. For those needs you will either have to write your own components (build), use shareware or freeware components (beg), or purchase commercial components (buy). The route you go depends on a number of factors. I will explain some of those factors in this article.

Build

There are a number of reasons to go the route of building components:

- You have a programming need that is not addressed by available components.
- You work for a company that insists on everything being in-house.
- You feel you cannot afford commercial components.
- You simply want to learn how to write components.

Some of these are certainly valid reasons. If you have a programming need that is not covered by any available components, you may have no choice but to write that component yourself. Component vendors typically write components that can appeal to a large audience. Anything that applies to a small audience is probably not going to be available in the third party component market.

Some companies insist that all code be produced in-house to reduce reliance on third parties. Custom components will have to be written in cases like this.

Probably the poorest reason to write your own components is because you feel you cannot afford to purchase commercial components. Granted, it depends on whether you program for a living or you are a hobbyist programmer. Except for hobbyists, it rarely pays to write a component if one is otherwise available.

I like to use TurboPower’s AsyncProfessional as an example. AsyncProfessional is a set of components

for serial communications. It includes components for basic serial communications, TAPI support (including voice modems), faxing via a fax modem and much more. At the time of this writing, the retail price of AsyncProfessional is $349.00 (US). It would easily take 10-12 man months to write the equivalent of AsyncProfessional. Do the math and you can see that it could cost from $50,000 to $100,000 to reproduce the features found in AsyncProfessional. Obviously, it does not pay to write components for full serial communications support when they can be purchased for a mere $349.00.

Writing components for the knowledge attained is certainly a viable reason to build rather than buy.

Beg

I used the word beg here simply because it fits with the alliteration used in this article's title. What I mean by this is to obtain freeware or shareware components. Most shareware and freeware components are available as Delphi components but they can probably be used in C++Builder without much additional effort.

Shareware and freeware components are certainly an option, but some care must be taken when using this type of component. There are many good freeware and shareware components available. There are many more that are not up to the quality you have come to expect from the VCL components. You may have to kiss many frogs before finding that prince.

A common problem with shareware and freeware components is documentation and support. You may find a component with online help, but many won't contain any documentation at all. Certainly it is rare to find a shareware component with printed documentation (if, in fact, one exists at all).

Installation is often a problem with shareware and freeware components. Some include pre-built packages but most are simply a collection of source files that you are expected to put into a package on your own.

Finally, consider that the hunt for a suitable freeware or shareware component may be more costly than purchasing a proven commercial component. Time is money. Consider the scenario where you spend several days finding, installing, and testing a component only to find that it doesn't work as advertised. It looked like a good deal to start with but now you have lost valuable time and must start again. I recommend that you only use shareware and freeware components that come highly recommended by your peers.

I want to stress that there are many good shareware and freeware components. I don't want to leave the impression that all components in this category are less than adequate.

Buy

Finally, you can buy commercial components. Buying commercial components has several advantages over building your own components and over shareware and freeware components. Some of those advantages are:

- Full documentation, often including printed documentation
- Technical support is almost always provided (free or fee-based)
- Professional component design and implementation
- Efficient cost vs. effort ratio
- Typically come with full source code
- Professional component installation

Most of these points are self-explanatory so I won't go over them in detail. Buying commercial components is a good solution if you can find components that fit your needs. It is efficient from a monetary standpoint (buying is much less expensive than developing). Typically commercial components come with some form of tech support. Documentation is also provided with commercial components.

One of the most compelling reasons to use commercial components is that they often come with full source code (at least those from reputable companies do). This means that you are never stuck with a set of components that becomes outdated in the unlikely event the company from which you originally purchased them is no longer available. You have the source so you can modify it as needed.

All in all, commercial components allow you to implement a particular bit of functionality with the least amount of time and money involved.

A word about Delphi components

Most of the commercial, shareware, and freeware components you find will be written in Delphi. There are a number of reasons for this. First, Delphi was available several years before C++Builder. The Delphi component vendors were already established by the time C++Builder came on the scene. As a result, the majority of the VCL third party component market has a Delphi code base. Second, Delphi components can be used in both Delphi and C++Builder. The converse is not true; components written in C++Builder cannot be used in Delphi. Obviously, it makes sense for component writers to maximize their profit potential by writing components in Delphi.

The fact that most components are written in Delphi is, for the most part, inconsequential. Usually a professional installation is implemented. The component packages are installed directly into the IDE and you can begin using the components immediately.

Earlier I said that most commercial components come with source code. In all but a few cases you will find that source code to be Delphi code. Looking at Delphi code may put you off at first, but it doesn't take long for the average C++ programmer to figure out what is happening in that code.

Conclusion

You may encounter situations where you must write your own components, and at that time you will have no choice but to write those components. Shareware and freeware components are an option, but one that comes with a degree of risk. Given the choice, I will opt to buy proven commercial components whenever possible.

Custom buttons, part I

by Damon Chandler

In this series of articles, I'll show you how to create a `TButton`-descendant component that will handle all of your button needs. In this first installment, I'll discuss how to create a generic owner-drawn button and how to use this technique to extend the `TButton` class to accept a color specification.

Owner-drawn buttons

Like many standard control classes, the Windows `BUTTON` control provides a so-called "owner-drawn" style, which allows you to customize the appearance of a button. This style is specified by adding `BS_OWNERDRAW` to the other styles that you'd normally pass as the `dwStyle` parameter to the `CreateWindowEx()` API function. Let's see how this is done by working through an example.

Figure A: *A form that contains an owner-drawn button.*

Creating an owner-drawn button from scratch

Figure A depicts a form that contains one child control: an owner-drawn button that's created directly via the `CreateWindowEx()` API function:

```
// in Form1's header...
private:
  HWND hODButton_;

// in Form1's source...
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  hODButton_ = CreateWindowEx(
    0, "BUTTON", "ODButton1",
```

```
    WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
    10, 10, 75, 25,
    Handle, NULL, HInstance, NULL
    );
}
```

By specifying the `BS_OWNERDRAW` style, you're effectively telling the button that you want to take over its rendering process. This is done—as it is with other owner-drawn controls—by handling the `WM_DRAWITEM` message that's sent by the button to its parent window whenever the button needs to be drawn. For our example, because `Form1` is the parent of the button, we need to handle the `WM_DRAWITEM` that's sent to `Form1`. We can do this by augmenting the `TForm::Dispatch()` method (via the message-mapping macros), like so:

```
// in Form1's header...
private:
  MESSAGE void __fastcall WMDrawItem(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    WM_DRAWITEM, TMessage, WMDrawItem)
END_MESSAGE_MAP(TForm)

// in Form1's source...
void __fastcall TForm1::
  WMDrawItem(TMessage& Msg)
{
  // grab a pointer to the DRAWITEMSTRUCT
  const DRAWITEMSTRUCT* pDrawItem =
    reinterpret_cast<DRAWITEMSTRUCT*>
      (Msg.LParam);

  // if the message is from our button...
  if (pDrawItem->hwndItem == hODButton_)
  {
    // grab a handle to the target DC
    const HDC hButtonDC = pDrawItem->hDC;

    // extract the target rectangle
    RECT RButton = pDrawItem->rcItem;

    // is the button pushed?
    const bool is_pushed =
```

```
  pDrawItem->itemState & ODS_SELECTED;
// does the button have focus?
const bool is_focused =
  pDrawItem->itemState & ODS_FOCUS;

// set the rendering flags to draw
// a regular push button, and adjust
// the rendering flags according to
// the button's current state
unsigned int flags = DFCS_BUTTONPUSH;
if (is_pushed) flags |= DFCS_PUSHED;

// render the push button by using
// the DrawFrameControl() function
DrawFrameControl(
  hButtonDC, &RButton,
  DFC_BUTTON, flags
  );

// render the button's text...
RECT RButtonText = RButton;
if (is_pushed)
{
  OffsetRect(&RButtonText, 1, 1);
}
DrawText(
  hButtonDC, "ODButton1\0", -1,
  &RButtonText, DT_CENTER |
  DT_VCENTER | DT_SINGLELINE
  );

// indicate keyboard focus (here via
// an etched edge rather than the
// traditional focus rectangle)...
if (is_focused)
{
  InflateRect(&RButton, -4, -4);
  DrawEdge(
    hButtonDC, &RButton,
    EDGE_ETCHED, BF_RECT
    );
}

// return TRUE for this message
```

```
    Msg.Result = TRUE;
  }
  // otherwise, pass the message along
  else TForm::Dispatch(&Msg);
}
```

You can see from this code that the first step is to grab a pointer to a `DRAWITEMSTRUCT` structure, which is sent with the `WM_DRAWITEM` message via the `LParam` data member. This structure holds all of the crucial drawing-related information such as the target device context (in the `hDC` data member), the target rectangle (in the `rcItem` data member), and the current state of the button (in the `itemState` data member). With this information, you'll know where to draw (`hDC`), what area to draw (`rcItem`), and in what state (`itemState`) you should draw the button. In fact, how you draw the button is up to you. As depicted in **Figure A**, this code will render a standard push-button with an etched selection rectangle. Note that when the button is pushed, the `itemState` data member will contain the `ODS_SELECTED` bit. Likewise, when the button has keyboard focus, `itemState` will contain the `ODS_FOCUS` bit. And, when the button is disabled, `itemState` will contain the `ODS_DISABLED` bit.

Unfortunately, this example isn't too useful because most of us never call the `CreateWindowEx()` function directly. We could create a `TWinControl` descendant class to handle this work, but because the `TButton` class is designed specifically for buttons, let's start there. We'll still need to specify the `BS_OWNERDRAW` style to create an owner-draw button, but we can do this from within the `CreateParams()` method of our `TButton` descendant class.

The TColorButton component

I've just shown you how to create a simple owner-drawn button. Let's now work on wrapping this functionality into a reusable VCL component, which I'll call `TColorButton`. The declaration of this class is provided in **Listing A**.

Notice that the `TColorButton` class introduces three published properties: `Color`, `ColorLo`, and `ColorHi`. As you can guess, the `Color` property will be used to specify the color of the button's face. This property is maintained by using the `Color` property of the button's `Brush`. The `ColorLo` and `ColorHi` properties will be used to specify the colors of the button's shadow and highlight, respectively. These latter two properties are maintained via the private `ColorLo_` and `ColorHi_` members, which we can initialize in the class constructor, like so:

```
__fastcall TColorButton::
  TColorButton(TComponent* Owner)
    : TButton(Owner),
    ColorLo_(clBtnShadow),
    ColorHi_(clBtnHighlight),
    Canvas_(new TCanvas(),
```

```
    draw_as_default_(false)
{
}
```

Note that the private `Canvas_` member is also initialized in the constructor. Later, I'll show you how to use this `TCanvas` object to ease the drawing process. The constructor is also used to initialize the `draw_as_default_` member, whose role I'll discuss shortly.

Specifying the BS_OWNERDRAW style

As I mentioned earlier, we'll use the `CreateParams()` method to specify the `BS_OWNERDRAW` style. Here's the code for that method:

```
void __fastcall TColorButton::
  CreateParams(TCreateParams& Params)
{
  TButton::CreateParams(Params);
  Params.Style |= BS_OWNERDRAW;
}
```

By adding `BS_OWNERDRAW` to the `TCreateParams::Style` data member, this code effectively instructs the `TButton` class (and thus the `TWinControl` class) to pass `BS_OWNERDRAW` to the `CreateWindowEx()` function, which is later called from within the `TWinControl::CreateWnd()` method. Notice, though (from **Listing A**), that the `TColorButton` class also overrides the `TButton::SetButtonStyle()` method. This step is needed because the `TButton` class uses `SetButtonStyle()` to change the button's style to `BS_PUSHBUTTON` or `BS_DEFPUSHBUTTON` depending on the value of the `ADefault` parameter. In our case, we want to preserve the `BS_OWNERDRAW` specification, so we need to override the `SetButtonStyle()` method:

```
void __fastcall TColorButton::
  SetButtonStyle(bool ADefault)
{
  if (draw_as_default_ != ADefault)
  {
    draw_as_default_ = ADefault;
    InvalidateRect(Handle, NULL, FALSE);
  }
}
```

This code updates the value of the private `draw_as_default_` member according to the `ADefault` parameter. We'll later use this member to determine if the button should be drawn as the default button

(i.e., with a thick black border).

That takes care of creating an owner-drawn button, but where's the actual drawing code? Well, this is where the `TColorButton::CNDrawItem()` method comes into play.

Handling the CN_DRAWITEM message

Recall that a normal owner-drawn button sends its parent window the `WM_DRAWITEM` message, which simply instructs the parent window that the button needs to be drawn or redrawn. This means that if you place a `TColorButton` object directly on a form, you can handle the `WM_DRAWITEM` message by tapping into the form's window procedure (just as we did before). This isn't too hard for a form—you simply augment the `TForm::Dispatch()` member function (by using the message-mapping macros). What do you do, though, if you want to place your `TColorButton` object on, say, a panel? Do you subclass the panel? Or, do you create a new `TPanel` descendant class and augment's *its* `Dispatch()` member function? Either of these options results in too much work.

Fortunately, when any `TWinControl` descendant receives the `WM_DRAWITEM` message, it will forward a copy of the message—called `CN_DRAWITEM`—back to the button itself. This way, you can handle the `CN_DRAWITEM` message from within the button's window procedure without worrying about which control the button is placed on. From the declaration of the `TColorButton` class, you can see that this message is mapped to the `CNDrawItem()` method. It's from within this method that you can draw the button in a new, customized fashion. Here, we'll use the `Color`, `ColorLo`, and `ColorHi` specifications—along with our `TCanvas` object—to render a colored button:

```
void __fastcall TColorButton::
  CNDrawItem(TMessage& Msg)
{
  // grab pointer to the DRAWITEMSTRUCT
  const DRAWITEMSTRUCT* pDrawItem =
    reinterpret_cast<DRAWITEMSTRUCT*>
      (Msg.LParam);

  // store the current state of
  // the target DC
  SaveDC(pDrawItem->hDC);
  // bind Canvas_ to the target DC
  Canvas_->Handle = pDrawItem->hDC;
  try
  {
    // extract the state flags...
    TOwnerDrawState state;
    // if the button has keyboard focus
```

```
    if (pDrawItem->itemState & ODS_FOCUS)
    {
      state = state << odFocused;
    }
    // if the button is pushed
    if (pDrawItem->itemState &
        ODS_SELECTED)
    {
      state = state << odSelected;
    }
    // if the button is disabled
    if (pDrawItem->itemState &
        ODS_DISABLED)
    {
      state = state << odDisabled;
    }

    // draw the button's face
    DoDrawButtonFace(state);

    // draw the button's text
    DoDrawButtonText(state);
  }
  catch (...)
  {
    // clean up
    Canvas_->Handle = NULL;
    RestoreDC(pDrawItem->hDC, -1);
  }
  // clean up
  Canvas_->Handle = NULL;
  RestoreDC(pDrawItem->hDC, -1);

  // reply TRUE
  Msg.Result = TRUE;
}
```

You'll notice that this definition of the `TColorButton::CNDrawItem()` method is somewhat similar to the previous definition of the `TForm1::WMDrawItem()` method. Here, instead of actually drawing the button from within the `CNDrawItem()` method, the `TColorButton` class fills a `TOwnerDrawState`-type variable, and then punts the work to its `DoDrawButtonFace()` and `DoDrawButtonText()` methods.

Drawing the button’s face

We’ll render the button’s face from within the `TColorButton::DoDrawButtonFace()` method. Here’s the code for that method:

```
void __fastcall TColorButton::
  DoDrawButtonFace(
    const TOwnerDrawState& state
    )
{
  // draw a colored button...
  Canvas_->Brush = Brush;
  TRect RClient = ClientRect;

  // if the button is the default button
  // or has keyboard focus...
  if (draw_as_default_ ||
      state.Contains(odFocused))
  {
    Canvas_->Pen->Color = clWindowFrame;
    Canvas_->Rectangle(
       RClient.Left, RClient.Top,
       RClient.Right, RClient.Bottom
       );
    InflateRect(
       reinterpret_cast<PRECT>(&RClient),
       -1, -1
       );
  }

  // if the button is pushed...
  if (state.Contains(odSelected))
  {
    Canvas_->Pen->Color = ColorLo_;
    Canvas_->Rectangle(
       RClient.Left, RClient.Top,
       RClient.Right, RClient.Bottom
       );
  }

  // if the button isn't pushed...
  else
  {
```

```
    Canvas_->FillRect(RClient);
    Frame3D(
      Canvas_.get(), RClient,
      ColorHi_, clWindowFrame, 1
      );

    POINT P[] = {
      {1, RClient.Bottom - 1},
      {RClient.Right - 1,
       RClient.Bottom - 1},
      {RClient.Right - 1,
       RClient.Top - 1}
      };
    Canvas_->Pen->Color = ColorLo_;
    Canvas_->Polyline(
       reinterpret_cast<TPoint*>(P), 2
       );
  }
}
```

There’s nothing special about this code—I simply took some screenshots of a button in its various states, examined its appearance, and then worked through the necessary `TCanvas` methods. You might be wondering why I didn’t use the `DrawFrameControl()` API function. Unfortunately, that function will always render a button in its default color (`clBtnFace`).

Drawing the button’s caption

The next task is to render the button’s caption. We’ll do this from within the `TColorButton::DoDrawButtonText()` member function, like so:

```
void __fastcall TColorButton::
  DoDrawButtonText(
    const TOwnerDrawState& state)
{
  if (Caption.Length() == 0) return;

  RECT RText = {0, 0, Width, Height};
  Canvas_->Font = Font;
  Canvas_->Brush = Brush;
  SetBkMode(
    Canvas_->Handle, TRANSPARENT
    );
```

```
  // if the button is pushed...
  if (state.Contains(odSelected))
  {
     // offset the caption
     OffsetRect(&RText, 1, 1);
  }
  // if the button is disabled...
  if (!Enabled ||
      state.Contains(odDisabled))
  {

    // render the caption
    // in a disabled fashion
    OffsetRect(&RText, 1, 1);
    Canvas_->Font->Color = ColorHi_;
    DrawText(
      Canvas_->Handle, Caption.c_str(),
      -1, &RText, DT_CENTER |
      DT_VCENTER | DT_SINGLELINE
      );
    OffsetRect(&RText, -1, -1);
    Canvas_->Font->Color = ColorLo_;
  }

  // render the caption
  DrawText(
    Canvas_->Handle, Caption.c_str(), -1,
    &RText, DT_CENTER | DT_VCENTER |
    DT_SINGLELINE
    );

  // if the button has keyboard focus...
  if (state.Contains(odFocused))
  {
    // render the selection rectangle
    TRect RFocus = ClientRect;
    InflateRect(
      reinterpret_cast<PRECT>(&RFocus),
      -4, -4
      );
    Canvas_->DrawFocusRect(RFocus);
  }
}
```

The bulk of the work of drawing the button's caption is handled by the `DrawText()` API function. This function is particularly handy because it will center the text—both vertically and horizontally—within the rectangle that you specify via the fourth (`lpRect`) parameter. Notice that when the button is disabled, the `DrawText()` function is called twice to achieve the chiseled effect. Also, when the button has keyboard focus, we use the `TCanvas::DrawFocusRect()` method to render the selected rectangle.

That takes care of drawing the button's face, text, and selection rectangle. Let's now focus on the rest of `TColorButton`'s methods

Finishing up

As you might have guessed, the `GetColor()`, `SetColor()`, `SetColorLo()`, and `SetColorHi()` methods provide access to the `Color`, `ColorLo`, and `ColorHi` properties. Here's how these methods are defined:

```
TColor __fastcall TColorButton::
  GetColor()
{
  return Brush->Color;
}

void __fastcall TColorButton::
  SetColor(TColor Value)
{
  if (Brush->Color != Value)
  {
    Brush->Color = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}

void __fastcall TColorButton::
  SetColorLo(TColor Value)
{
  if (ColorLo_ != Value)
  {
    ColorLo_ = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}
```

```
void __fastcall TColorButton::
  SetColorHi(TColor Value)
{
  if (ColorHi_ != Value)
  {
    ColorHi_ = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}
```

The `CMFontChanged()` and `CMEnabledChanged()` methods are called when the button receives the `CM_FONTCHANGED` and `CM_ENABLEDCHANDED` messages, respectively. These are VCL-specific messages that are sent to the button when its `Font` or `Enabled` properties have changed (either at design time or at run time). When this happens, we need to instruct the button to repaint itself:

```
void __fastcall TColorButton::
  CMFontChanged(TMessage& Msg)
{
  TButton::Dispatch(&Msg);
  InvalidateRect(Handle, NULL, TRUE);
}

void __fastcall TColorButton::
  CMEnabledChanged(TMessage& Msg)
{
  TButton::Dispatch(&Msg);
  InvalidateRect(Handle, NULL, TRUE);
}
```

There's one last method: `WMLButtonDblClk()`. You can see from **Listing A** that this method is called whenever the button receives the `WM_LBUTTONDBLCLK` message; this message is sent whenever the button is double-clicked. Because a normal button (i.e., a non-owner-drawn push-button) processes this message as if it were a `WM_LBUTTONDOWN` message, we'll need to do the same (otherwise, the button will react very slowly when it's clicked rapidly). Here's the definition of the `WMLButtonDblClk()` method:

```
void __fastcall TColorButton::
  WMLButtonDblClk(TMessage& Msg)
{
  SNDMSG(
    Handle, WM_LBUTTONDOWN,
    Msg.WParam, Msg.LParam
```

```
    );
}
```

Figure B shows a form containing several `TColorButtons`.

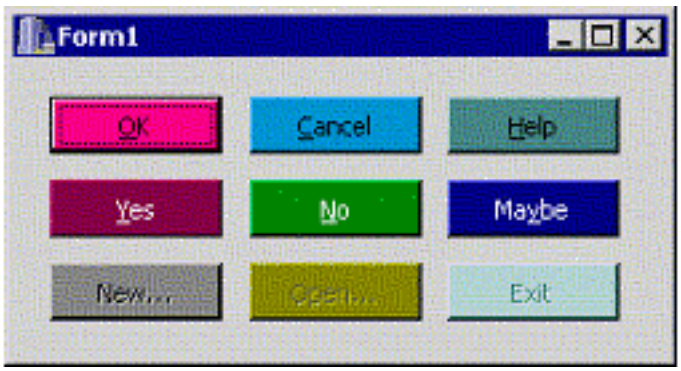


Figure B: *Some TColorButton objects.*

Conclusion

I've shown you how to create an owner-drawn button and how to use this style to create a colored-button component. Next month, I'll show you how to extend this technique even further. I'll discuss how to add a glyph to the button, and how to define the button's face by using a custom bitmap (which is the first step toward creating an application that supports skins). For now, experiment with the code for the `TColorButton` component; it's available for download from **www.residorph.com.**

Listing A: ***Declaration of the TColorButton class***

```
#include <memory>
class TColorButton : public TButton
{
public:
  __fastcall TColorButton(TComponent* Owner);

__published:
  __property TColor Color =
    {read = GetColor, write = SetColor};
  __property TColor ColorLo =
    {read = ColorLo_, write = SetColorLo,
     default = clBtnShadow};
  __property TColor ColorHi =
    {read = ColorHi_, write = SetColorHi,
     default = clBtnHighlight};
```

```
protected:
  // inherited member functions
  virtual void __fastcall CreateParams(
    TCreateParams& Params);
  virtual void __fastcall SetButtonStyle(
    bool ADefault);

  // introduced member functions
  virtual void __fastcall DoDrawButtonFace(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonText(
    const TOwnerDrawState& state);

private:
  TColor ColorLo_;
  TColor ColorHi_;
  std::auto_ptr<TCanvas> Canvas_;
  bool draw_as_default_;

  TColor __fastcall GetColor();
  void __fastcall SetColor(TColor Value);
  void __fastcall SetColorLo(TColor Value);
  void __fastcall SetColorHi(TColor Value);

  MESSAGE void __fastcall CNDrawItem(
    TMessage& Msg);
  MESSAGE void __fastcall WMLButtonDblClk(
    TMessage& Msg);
  MESSAGE void __fastcall CMFontChanged(
    TMessage& Msg);
  MESSAGE void __fastcall CMEnabledChanged(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CN_DRAWITEM, TMessage, CNDrawItem)
  MESSAGE_HANDLER(
    WM_LBUTTONDBLCLK, TMessage, WMLButtonDblClk)
  MESSAGE_HANDLER(
    CM_FONTCHANGED, TMessage, CMFontChanged)
  MESSAGE_HANDLER(
    CM_ENABLEDCHANGED, TMessage, CMEnabledChanged)
```

```
END_MESSAGE_MAP(TButton)
};
```

Drawing transparent images

By Mark G. Wiseman

If you have used the VCL object, `TBitmap`, you know that you can use it to display bitmaps where one color is transparent. If you have ever looked at the source code for `TBitmap`, you find it takes quite a few lines of code to accomplish this. If you have tried to draw semi-transparent images in early versions of 32-bit Windows, you probably found it incredibly difficult.

There are two Windows API functions, `TransparentBlt()` and `AlphaBlend()`, that you can use to draw transparent images. The `TransparentBlt()` function will draw bitmaps with a single transparent color, just like the `TBitmap`. However, you can do this using only a few lines of code.

The `AlphaBlend()` function will draw images where every pixel in the image can have a different level of transparency—from totally opaque to completely transparent.

These two functions may sound too good to be true. Well, they are true if you're using Windows 98 or greater or Windows 2000 or greater. Unfortunately, they will not work with Windows 95 or Windows NT 4.

The online help included with C++Builder does not include help for `TransparentBlt()` or `AlphaBlend()`. You can, however, find help for these two functions on the Microsoft Web site at http://msdn.microsoft.com. Do a search on the function names.

In this article, I'll show you how to use these two functions, and since we'll need a way to see the transparency effects for testing, I'm also going to show you a neat little function for drawing checkerboard patterns.

I've written a demonstration program that lets you to see everything in action. You can download this program from the Bridges Publishing Web site.

Crown me

The `DrawCheckerboard()` function from this article's example program draws a checkerboard pattern. Here is the declaration for `DrawCheckerboard()`:

```
void DrawCheckerboard(
  TCanvas *canvas,
  int boardWidth, int boardHeight,
  TColor color1 = clBlack,
```

```
  TColor color2 = clWhite,
  int checkerWidth = 8,
  int checkerHeight = 8);
```

Listing A shows a snippet of source code from the demo program that shows how it uses the `DrawCheckerboard()` function. This pattern is used in the demo program as a background for the transparent images it draws.

The checkerboard pattern is drawn onto a `TCanvas`, the first argument to the function. The next two arguments are the height and width of the checkerboard. The rest of the arguments to `DrawCheckerboard()` have default values that produce a black-and-white checkerboard pattern where each square in the pattern has dimensions of 8 x 8 pixels. You can vary the colors of the alternating squares using the `color1` and `color2` arguments and the size of the squares using the `checkerWidth` `checkerHeight` arguments.

Actually, using the word *squares* is a little misleading. The pattern can consist of squares, rectangles, stripes or a solid color. For example, calling `DrawCheckerboard()` with `checkerWidth = 16` and `checkerHeight = 4`, will produce a pattern of rectangles. Calling the function with `checkerWidth = 0`, will produce horizontal stripes. Calling the function with `checkerHeight = 0`, will produce vertical stripes. And, if both `checkerWidth` and `checkerHeight` are equal to zero, `DrawCheckerboard()` will draw a solid background using `color1`.

In the demo program, I draw the checkerboard pattern onto a `TBitmap` object. The program allows you to change the colors and sizes of the squares and the transparency and visibility of the `TBitmap`. By setting the `Transparent` property of the `TBitmap` object to true, the color used in the top-left square will become transparent throughout the entire bitmap. This is a function of the `TBitmap` object, not `DrawCheckerboard()`.

TransparentBlt

Now, let's take a look at the `TransparentBlt()` function. **Figure A** shows the demo program with an image displayed by this function.

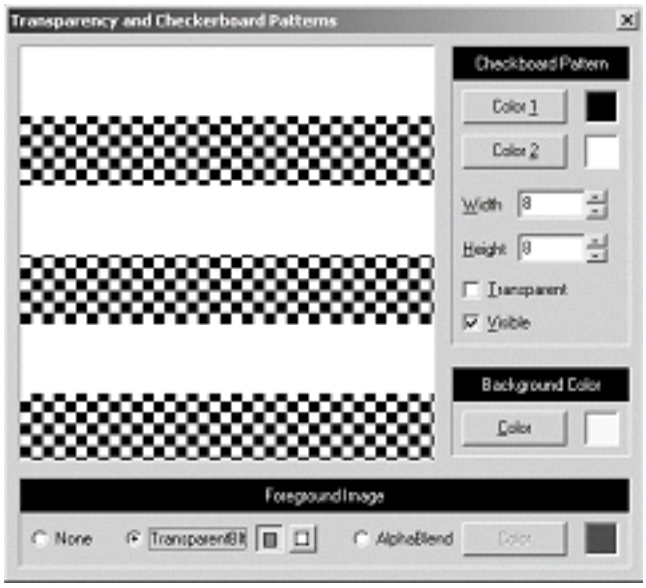


Figure A: *Image displayed with TransparentBlt() function.*

The image consists of horizontal red and white stripes. The `TransparentBlt()` function has been told to draw the color red transparently. Here is the declaration for `TransparentBlt()`:

```
BOOL TransparentBlt(
  // handle to destination DC
  HDC hdcDest,
  // x-coord of destination
  // upper-left corner
  int nXOriginDest,
  // y-coord of destination
  // upper-left corner
  int nYOriginDest,
  // width of destination rectangle
  int nWidthDest,
  // height of destination rectangle
  int hHeightDest,
  // handle to source DC
  HDC hdcSrc,
  // x-coord of source
  // upper-left corner
  int nXOriginSrc,
  // y-coord of source
  // upper-left corner
  int nYOriginSrc,
  // width of source rectangle
  int nWidthSrc,
  // height of source rectangle
```

```
  int nHeightSrc,
  // color to make transparent
  UINT crTransparent
);
```

This function works just like the `StretchBlt()` function, except the last argument to this function is the color that should be drawn transparently. In the demo program, I use a `TPaintBox` object to draw the example bitmaps. I do the drawing in a function I assign to the `OnPaint` event of the `TPaintBox`. This function is shown in **Listing B**.

I've created a simple bitmap image in the demo program that consists of red and white horizontal stripes. The handle to the image is `transHandle`. The demo program allows you to tell `TransparentBlt()` which of the two colors, red or white, should be made transparent.

AlphaBlend

The `AlphaBlend()` function is a little more complicated than `TransparentBlt()`, but it is very powerful. **Figure B** shows the demo program with an image displayed using this function.

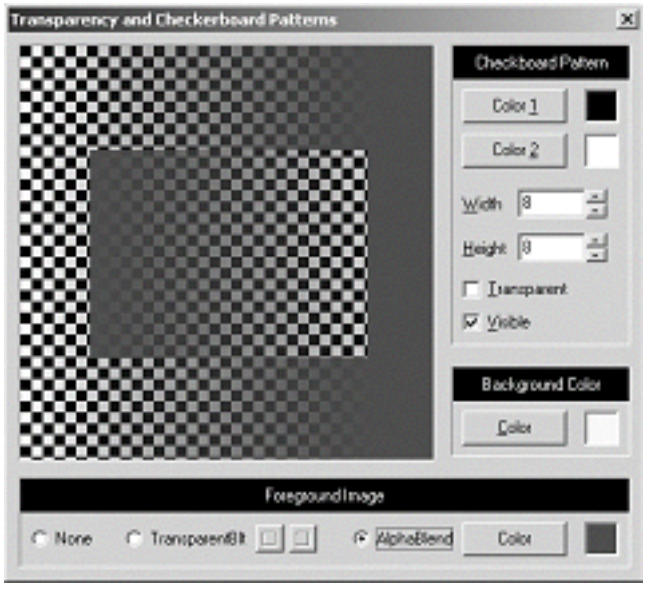


Figure B: *Image displayed with AlphaBlend() function.*

Here is the declaration for `AlphaBlend()`:

```
BOOL AlphaBlend(
  // handle to destination DC
  HDC hdcDest,
  // x-coord of upper-left corner
```

```
  int nXOriginDest,
  // y-coord of upper-left corner
  int nYOriginDest,
  // destination width
  int nWidthDest,
  // destination height
  int nHeightDest,
  // handle to source DC
  HDC hdcSrc,
  // x-coord of upper-left corner
  int nXOriginSrc,
  // y-coord of upper-left corner
  int nYOriginSrc,
  // source width
  int nWidthSrc,
  // source height
  int nHeightSrc,
  // alpha-blending function
  BLENDFUNCTION blendFunction
);
```

This function uses a 32-bit image consisting of four 8-bit channels for red, green, blue and *alpha*. The alpha channel value is the amount of transparency. A value of 0 is totally transparent and a value of 255 is totally opaque. Values in between are semi-transparent.

The Windows API declares the structure, `RGBQUAD`, which will hold this 32-bit value. The `rgbReserved` member of `RGBQUAD` holds the alpha channel value.

I want to be clear about how powerful this function is. Every pixel in the image can have one of over 16-million colors *and* 256 different levels of transparency. This means, unlike `TransparentBlt()`, that you can have the same color of red that is transparent in some parts of the image and opaque or semi-transparent in other parts.

So, how do we use `AlphaBlend()`? The actual function call is very similar to the standard `StretchBlt()` function; but we'll have to do some preparatory work before calling it.

First, we have to create an image using pixels in the `RGBQUAD` format. **Listing C** is the function I use in the demo program to create such an image. I used a Windows `DIBSECTION` to make things easy. I just drew two squares, one inside the other, consisting of one color, but varying the transparency across each of the squares horizontally and in opposite directions. The demo program will allow you to select the color to use.

There is one tricky part to creating an image to pass to `AlphaBlend()`. The red, green and blue channels of each pixel must be pre-multiplied by a factor equal to the alpha channel value divided by 255. This requirement is a limitation of the `BLENDFUNCTION` structure used as the last argument when calling `AlphaBlend()`. I'll talk more about `BLENDFUNCTION` shortly.

Let's take a look at how to do the pre-multiplication. Every pixel in the image must have the pre-multiplication performed on it. Remember, each pixel is represented by an `RGBQUAD` structure; here's some example code to perform the pre-multiplication:

```
RGBQUAD oldPixel, newPixel;

newPixel.rgbBlue = (oldPixel.rgbBlue *
  oldPixel.rgbReserved) / 255;
newPixel.rgbGreen = (oldPixel.rgbGreen *
  oldPixel.rgbReserved) / 255;
newPixel.rgbRed = (oldPixel.rgbRed *
  oldPixel.rgbReserved) / 255;
newPixel.rgbReserved =
  oldPixel.rgbReserved;
```

The last argument to the `AlphaBlend()` function is a `BLENDFUNCTION` structure. This structure tells `AlphaBlend()` how to blend the source and destination bitmaps. You should look at the Microsoft Web site for a complete description of how `BLENDFUNCTION` works.

Fading away

Using `TransparentBlt()` and `AlphaBlend()` makes drawing transparent images in Windows relatively easy. Remember that you can't use these functions with all versions of 32-bit Windows.

Depending on your alpha value, I may see you later.

Listing A: *The DrawCheckerboard() function.*

```
void TMainForm::DrawCheckerboard(
  TCanvas *canvas,
  int boardWidth, int boardHeight,
  TColor color1, TColor color2,
  int checkerWidth, int checkerHeight)
{
  TColor startColor = color1;
  for (int row = 0; row < boardHeight;row++) {
```

```
    if (height != 0 && row % height == 0) {
      startColor = startColor == color1
        ? color2 : color1;
    }

    TColor color = startColor;
    for (int col = 0; col < boardWidth;col++) {
      if (width != 0 && col % width == 0) {
        color = color == color1
          ? color2 : color1;
      }

      canvas->Pixels[col][row] = color;
    }
  }
}
```

Listing B: *Function assigned to the OnPaint event of TPaintBox.*

```
void __fastcall TMainForm::PaintBoxPaint(
 TObject *Sender)
{
  int cx = BkgdPanel->ClientWidth;
  int cy = BkgdPanel->ClientHeight;

  HDC hDC = CreateCompatibleDC(
    PaintBox->Canvas->Handle);
  if (hDC == 0)
    throw Exception("Whoops!");

  if (TransRadio->Checked) {
    if (SelectObject(hDC, transHandle) == 0)
      throw Exception("Whoops!");

    TransparentBlt(PaintBox->Canvas->Handle,
      0, 0, cx, cy, hDC, 0, 0, cx, cy, transColor);
  }
  else {
    if (SelectObject(hDC, alphaHandle) == 0)
      throw Exception("Whoops!");

    BLENDFUNCTION blend =
      {AC_SRC_OVER, 0, 255, AC_SRC_ALPHA};
    AlphaBlend(PaintBox->Canvas->Handle,
```

```
      0, 0, cx, cy, hDC, 0, 0, cx, cy, blend);
  }

  if (hDC) DeleteDC(hDC);
}
```

Listing C: *Creating a bitmap for the AlphaBlend() function.*

```
void TMainForm::CreateAlphaImage()
{
  int cx = BkgdPanel->ClientWidth;
  int cy = BkgdPanel->ClientHeight;

  if (alphaHandle)
    DeleteObject(alphaHandle);
  alphaHandle =
    GetDIBSection(cx, cy, &alphaImage);

  RGBQUAD of, nf;
  of.rgbBlue = (alphaColor & 0x00FF0000) >> 16;
  of.rgbGreen = (alphaColor & 0x0000FF00) >> 8;
  of.rgbRed = (alphaColor & 0x000000FF);
  of.rgbReserved = 0;

  for (int r = 0; r < cy; r++) {
    for (int c = 0; c < cx; c++) {
      int x = min(c, 255);
      nf.rgbBlue = (of.rgbBlue * x) / 255;
      nf.rgbGreen = (of.rgbGreen * x) / 255;
      nf.rgbRed = (of.rgbRed * x) / 255;
      nf.rgbReserved = x;
      alphaImage[r * cx + c] = nf;
    }
  }

  for (int r = 75; r < cy - 75; r++) {
    for (int c = 50; c < cx - 50; c++) {
      int x = min(cx - c, 255);
      nf.rgbBlue = (of.rgbBlue * x) / 255;
      nf.rgbGreen = (of.rgbGreen * x) / 255;
      nf.rgbRed = (of.rgbRed * x) / 255;
      nf.rgbReserved = x;
      alphaImage[r * cx + c] = nf;
    }
```

```
    }
}
```

Under construction

By Mark G. Wiseman

I am going to show you two functions that I use in nearly every program I write. Two functions that are never included in any program I distribute. Well, actually, the two functions are really different, over-loaded version of one function, `UnderConstruction()`.

These two functions will display a message box telling you that something in your program—usually event code—is incomplete. They will also make an attempt to tell you what object called them.

Breaking ground

When you add a component such as a `TButton`, `TMenuItem` or `TAction` to a VCL project, you need to provide code for an event of that component before it becomes useful. Normally, you would write code for the `OnClick()` event of `TButton` and `TMenuItem` and for the `OnExecute()` event of `TAction`.

I find that when I start to design and lay out a form or menu, I already know what buttons and menu items I want to use and I know the names of the event functions I want to associate with these. However, I don't necessarily want to stop and write all the code required to make each of these event functions operational.

I could just leave the event code out and go back and fill it in later. However, there are two problems with this approach.

First, when testing a program that is under construction, menu items, buttons etc. that have no code attached will just do nothing when selected. They may also appear to do nothing even if they have events coded. How do you tell the difference? Using the `UnderConstruction()` functions, you know instantly that this event is not working, because the code has not been completed.

Second, when using a control associated with a `TAction` item, (e.g. a `TmenuItem`), the control will be disabled if there is no event code for the `OnExecute()` event of `TAction`. Again, it is difficult to tell if a control is disabled because there is no event code or for some other reason. The `UnderConstruction()` function enables the item and lets you know that the code is incomplete.

The blueprint

Listings A and B contain the source code for the `UnderConstruction()` functions. The first

version of `UnderConstruction()` takes a pointer to a `TObject` as a required parameter and an optional argument of an `AnsiString`. It tries to identify the object by casting it to a `TControl` and reading the `Name` property of the `TControl`. It tries to further identify any `TAction`, `TButton`, or `TMenuItem` by reading that controls `Caption` property. If the optional `AnsiString` parameter was used, `UnderConstruction()` appends this string to the control's identification string and calls the second version of `UnderConstruction()` with the identification string as its argument.

The second version of `UnderConstruction()` displays a message box with its `AnsiString` string argument as the bulk of the text in the message box.

Let's say we're designing a form, `TMainForm`, for a new project. We add a menu item named "ConvertItem" with the caption "Convert File". If we generate the `OnClick()` event code for the item, we will get the empty shell for a function named `ConvertItemClick()`. However, some one else is writing the code for this event. So, let's use the `UnderConstruction()` functions as a placeholder. Here's what the event code will look like:

```
void __fastcall TMainForm::
  ConvertItemClick(TObject *Sender)
{
  UnderConstruction(Sender);
}
```

When this menu item is selected, the message box shows in **Figure A** will be displayed.

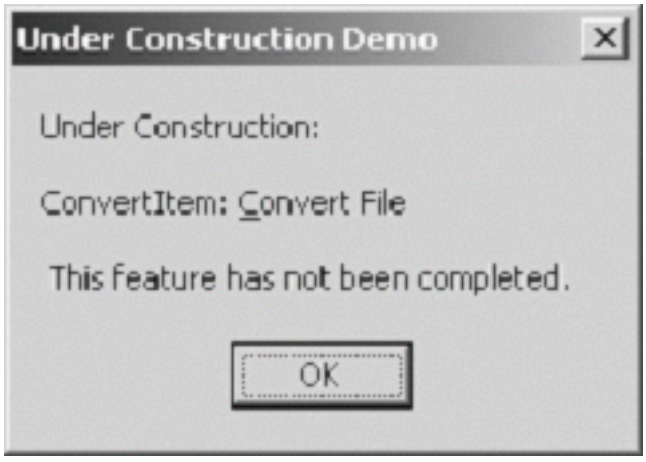


Figure A: *Under Construction message box*

Let's use the optional string parameter of `UnderConstruction()` to provide even more information:

```
void __fastcall TMainForm::
  ConvertItemClick(TObject *Sender)
{
```

```
  UnderConstruction(Sender,
    "Bill is working on this code");
}
```

This code will produce the message box shown in **Figure B** when the menu item is selected.

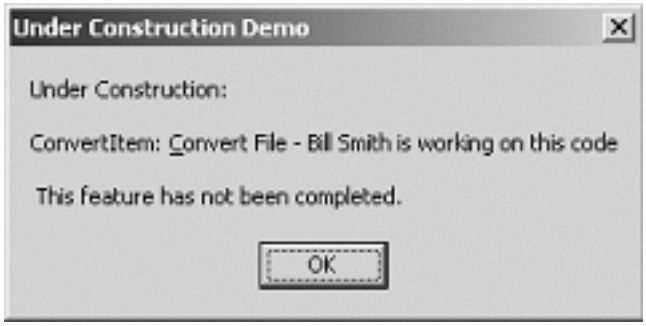


Figure B: *Under Construction message box using additional AnsiString parameter*

Project cleanup

Obviously, the `UnderConstruction()` functions should only be used during debugging and testing—while your program is under construction. That's why I say that I use them in every program I write but they are not included in any program I distribute (except, of course, for the demo program for this article). The program can be found on the Bridges Publishing Web site.

For these functions to work, you must first define the macro `_UNDERCONSTRUCTION`. The best place to do this is in the Conditional Defines setting of the Directories/Conditionals tab of the Project Options dialog. When this macro is defined, the `UnderConstruction()` functions work as described in this article. Before you do the final build of your project, remove the `_UNDERCONSTRUCTION` macro definition. If you have accidentally left a call to `UnderConstruction()` in your code, you will get a compiler error.

Construction complete

You could add code to the first version of `UnderConstruction()` to provide information for more types of VCL objects. For instance, it might be useful to include code to provide the name of the `TDataSet` in events such as `BeforePost()`. However, I've found `UnderConstruction()` to be a very useful function just as it's written.

Listing A: *UnderConstruction.h*

```
#ifndef UnderConstructionH
```

```
#define UnderConstructionH

#ifdef _UNDERCONSTRUCTION

void __fastcall UnderConstruction(
  TObject *Sender, const String &msg = "");

void __fastcall UnderConstruction(
  const String &feature = "");

#endif   // _UNDERCONSTRUCTION

#endif   // UnderConstructionH
```

Listing B: *UnderConstruction.cpp*

```
#ifdef _UNDERCONSTRUCTION

#include <vcl.h>
#pragma hdrstop

#include "UnderConstruction.h"

void __fastcall UnderConstruction(
  TObject *Sender, const String &msg)
{
  String feature;

  TComponent *component =
    dynamic_cast<TComponent *>(Sender);
  if (component != 0)
    feature += component->Name;

  TAction *action =
    dynamic_cast<TAction *>(Sender);
  if (action)
    feature += ": " + action->Caption;

  TMenuItem *item =
    dynamic_cast<TMenuItem *>(Sender);
  if (item)
    feature += ": " + item->Caption;

  TButton *button =
    dynamic_cast<TButton *>(Sender);
```

```
  if (button)
    feature += ": " + button->Caption;

  if (msg.IsEmpty() == false)
    feature += " - " + msg;

  UnderConstruction(feature);
}

void __fastcall UnderConstruction(
   const String &feature)
{
  String msg = "Under Construction:";
  if (feature.IsEmpty() == false)
    msg += "\r\n\r\n" + feature;
  msg += "\r\n\r\n This feature has "
    "not been completed.";

  ShowMessage(msg);
}

#pragma package(smart_init)

#endif   // _UNDERCONSTRUCTION
```

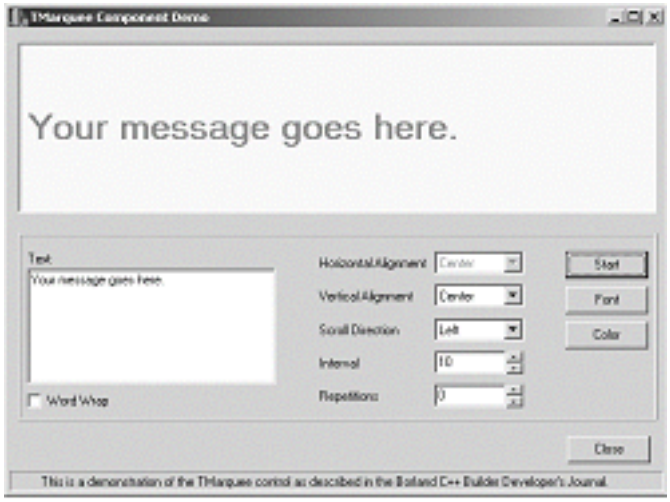
A scrolling marquee component

by Mark G. Wiseman

Scrolling marquees. You may either love them or hate them. But how do you create one? In this article, I'll show you how I created a scrolling marquee component which I named `TMarquee`. I'll discuss creating components in general, drawing text in Windows and a little bit about the graphics technique of double buffering.

Figure A shows a sample application that I've written to experiment with all the important properties of `TMarquee`. Using this application, you can instantly see the effect of changing a property in the component.

Figure A



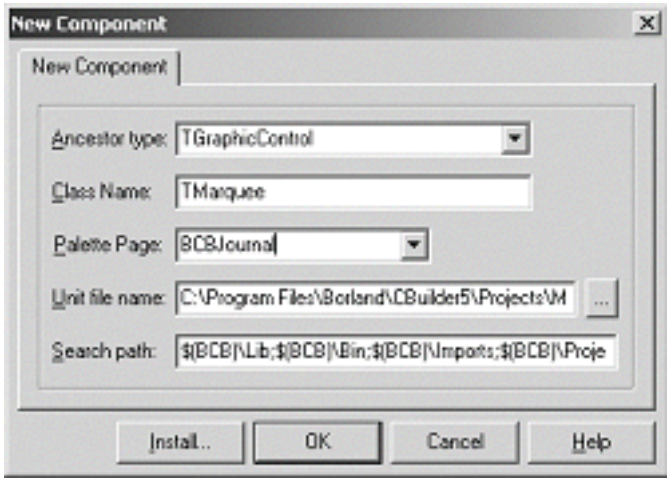
Demo application using a TMarquee component.

Creating components

Some programmers seem to believe that creating components in C++Builder is difficult. They're wrong. Creating a component is easy. The `TMarquee` component in this article consists of about 380 lines of code. Less than 20 of those lines were necessary to turn the *class* `TMarquee` into the *component* `TMarquee`. And guess what? C++Builder wrote all of those lines of code for me when I used the Component Wizard in the IDE!

Figure B shows the Component Wizard that is displayed when you select Component | New Component… from the C++Builder main menu.

Figure B



The C++Builder New Component wizard.

Using the wizard, you enter the name of your component, the name of the palette page where your component should be placed, and the *Ancestor Type* for your component.

The ancestor type is the name of the component from which you will derive your component. In C++ terms, the ancestor type is your component's parent class.

Ancestor type

Selecting the correct ancestor type is very important. You can save yourself a lot of work by picking an ancestor type that implements most of the properties, methods and events that your component needs.

I selected the `TGraphicControl` as the ancestor type for `TMarquee`. `TGraphicControl` provides a `Canvas` property that I can use to paint the marquee's background and text and it provides a virtual `Paint` method that I can override to respond to `WM_PAINT` messages.

You also want to be careful that you don't select an ancestor type that provides more functionality than you need. I could have derived `TMarquee` from `TCustomControl`. `TCustomControl` provides everything that `TGraphicControl` does. But, it is also a windowed control. This means that `TCustomControl` has a Window handle and can receive input focus. `TMarquee` doesn't need input focus and creating a window to display text on screen is wasteful of Windows' resources.

The __published keyword

For a component's property to appear in the Object Inspector of the IDE, it must be declared as a `__published` member of the component's class. The `__published` keyword is a Borland specific

extension to the C++ language. It is similar to the C++ keyword `public`. Class members that are `__published` are, in fact, `public` members that will also appear in the Object Inspector.

If you look at the header file for `TGraphicControl`, you'll notice that some of the properties I use in `TMarquee`, such as `Visible` and `OnClick`, are declared as either `protected` or `public`. I re-declared the properties as `__published` in the header file for `TMarquee` so that they would appear in the Object Inspector. Here is a snippet from the header file showing how this is done:

```
class PACKAGE TMarquee :
  public TGraphicControl
{
  __published:

    __property Visible;
    __property OnClick;

  public:

};
```

TMarquee specifics

Now lets take a look at the design of `TMarquee` and how that design is implemented. I wanted a scrolling marquee—a component that scrolled text across the screen.

I wanted the text to move in any of four directions: left, right, up and down. I wanted to set the speed at which the text moved. And, of course, I wanted to be able to specify the appearance of the text (typeface, size, color, etc).

I also wanted to specify the background color behind the text. Finally, for all those users who hate scrolling marquees, I wanted to be able to limit the number of times the text scrolled by their irritated eyes.

Properties

I created several properties to control the scrolling text. **Table A** lists these properties. The best way to determine exactly how these properties work is to study the source code and then play with the demo application.

Here is the really cool part about this component: it works during design time! While designing a form in the IDE, drop a `TMarquee` component onto the form and set the `Active` property to `true`. The text

will actually start scrolling right there in the IDE.

Table A: Properties specific to TMarquee

Property	Description
`Active`	Set `Active` to `true` to start the text scrolling. Setting it to `false` will stop the scrolling.
`Alignment`	The horizontal alignment of the text within the marquee. This property is only used when the text is scrolling vertically.
`Color`	The background color of the marquee.
`Direction`	The direction the text will scroll within the marquee.
`Font`	The font properties of the marquee text.
`Interval`	The interval in milliseconds between each move of the text.
`Layout`	The vertical alignment of the text within the marquee. The property is only used when the text is scrolling horizontally.
`Lines`	This is the text displayed in the marquee. Internally, `TMarquee` stores its text in a `TStringList`.
`Repetitions`	How may times the text should scroll across the marquee before stopping. Setting this property to `0` will cause the text to scroll forever.
`WordWrap`	If the text is too wide to fit inside the marquee, the text will be wrapped when this property is `true`.

Methods

Most of the code is simple and straightforward. There are three methods that I do want to discuss, `Paint()`, `Move()`, and `MeasureText()`.

The `Paint()` method contains the code to actually draw the marquee on the screen. It is inherited from `TGraphicControl` and is called whenever `TMarquee` receives a `WM_PAINT` message from Windows.

`Paint()` uses a graphics technique called *double buffering*. The marquee is first constructed or drawn off screen, in memory, and then is moved to the screen all at once. This prevents a lot of unwanted

flashing that would appear if each part of the marquee were drawn to the screen individually. Here is code for the `Paint()` method:

```
void __fastcall TMarquee::Paint(void)
{
  work->Width = Width;
  work->Height = Height;

  UINT format = wordWrap ?
    DT_WORDBREAK : 0;

  TRect r(0, 0, Width, 0);
  if (direction == mqLeft
    || direction == mqRight)
  {
    r.Left = xpos;

    r.Top = 0;
    if (layout == tlCenter)
      r.Top =
        (Height - textHeight) / 2;
    else if (layout == tlBottom)
      r.Top = Height - textHeight;

    r.Right = r.Left + textWidth;
  }
  else
  {
    r.Left = 0;
    if (alignment == taCenter)
      format |= DT_CENTER;
    else if (alignment==taRightJustify)
      format |= DT_RIGHT;

    r.Top = ypos;

    r.Right = Width;
  }

  r.Bottom = r.Top + textHeight;

  TRect s(0, 0,
    work->Width, work->Height);
  work->Canvas->FillRect(s);
```

```
  DrawText(work->Canvas->Handle,
    lines->Text.c_str(), -1, &r,format);

  Canvas->CopyRect(s, work->Canvas, s);
}
```

I use a `TBitmap` object, named `work`, to hold the off-screen image of the marquee. I draw the background of the marquee first, by calling the `FillRect()` method of the `TBitmap's Canvas`. Then I draw the text in the correct location within the marquee using the Windows API function, `DrawText()`. I then copy this completed image of the marquee to the screen by calling the `CopyRect()` method of the marquee's `Canvas`.

The location of the text within the marquee is calculated by the `Move()` method. This method is called by a `TTimer` component at the interval set in the `Interval` property of `TMarquee`. The `Move()` method moves the text one pixel at a time. Here is that code:

```
void __fastcall TMarquee::Move()
{
  switch(direction)
  {
    case mqLeft:
      if (xpos < -textWidth)
      {
        xpos = Width;
        if (repetitions > 0)
          repCount++;
      }
      else
        xpos--;
      break;

  case mqRight:
    if (xpos > Width)
    {
      xpos = -textWidth;
      if (repetitions > 0)
        repCount++;
    }
    else
      xpos++;
    break;
```

```
  case mqUp:
    if (ypos < -textHeight)
    {
      ypos = Height;
      if (repetitions > 0)
        repCount++;
    }
    else
      ypos--;
    break;

  case mqDown:
    if (ypos > Height)
    {
      ypos = -textHeight;
      if (repetitions > 0)
        repCount++;
    }
    else
      ypos++;
    break;
  }

  if (repetitions > 0
    && repCount == repetitions)
      Active = false;
  else if (Visible)
    Paint();
}
```

The size of the text within the marquee is calculated by the `MeasureText()` method. This method uses the Windows API function `DrawText()` when the `WordWrap` property is `true`. Look up this function in the Windows API online help to see how this works. In particular, look at the flag `DT_CALCRECT`. The `MeasureText()` method is as follows:

```
void __fastcall TMarquee::MeasureText()
{
  int lineHeight =
    work->Canvas->TextHeight("Wj");

  TRect r(0, 0, Width, 0);
  if (wordWrap)
  {
```

```
    textHeight =
      DrawText(work->Canvas->Handle,
        lines->Text.c_str(), -1, &r,
        DT_CALCRECT | DT_WORDBREAK);
    textHeight -= lineHeight;
    textWidth = r.Right;
  }
  else
  {
    textWidth = textHeight = 0;
    for (int i= 0;i<lines->Count;i++)
    {
      String s = lines->Strings[i];
      textWidth = max(textWidth,
      work->Canvas->TextWidth(s));
    }
    textHeight =
      lineHeight * lines->Count;
  }
}
```

Scrolling off

That’s really all there is to it. As I mentioned at the beginning of this article, some users won’t like scrolling marquees. They find them too distracting. `TMarquee` was designed to look and act just like a `TLabel` control when the `Active` property is `false`. So, when using `TMarquee` in your programs, you might want to give the user the option of turning off the scrolling. Once the scrolling is stopped, you can center the text in the marquee by calling the `Center()` method.

Creating a multiselect tree-view

by Damon Chandler

Tree-views are great for displaying data in a compact, hierarchical format. Unfortunately, the standard tree-view control—and thus, the VCL `TTreeView` component—allows only a single node to be selected at any given time. In this article, I'll show you how to overcome this limitation. I'll first give a brief overview of the Custom Draw service and it's applicability to tree-views; and then I'll explain how to use this technology to easily implement basic multiple-selection functionality.

Custom drawn tree-views

A few months back, I explained how to use the Custom Draw service with the `TTrackBar` control. In that article, I mentioned that many common controls provide Custom Draw notification messages—the tree-view control is no exception.

As it turns out, in newer version of C++Builder, the `TTreeView` class exposes Custom-Draw-specific events (e.g., `OnCustomDrawItem`). I won't cover these events for two reasons: (1) they're not available in all versions of C++Builder and (2) they're a bit buggy. Instead, I'll show you how to handle the `NM_CUSTOMDRAW` notification message directly. Recall that this notification message is sent to the tree-view's parent window in the form of a `WM_NOTIFY` message. Next I'll explain how to handle this latter message.

Handling the WM_NOTIFY message

Imagine a form that contains only one child control: a `TTreeView` object. Because the tree-view is a direct child of the form, the tree-view will send all of its notification messages to the form (i.e., to its parent window). So in this ideal setup, to handle the `WM_NOTIFY` message, all you need to do is augment the form's `Dispatch()` method by using the message-mapping macros:

```
class TForm1 : public TForm
{
__published:
  TTreeView *TreeView1;
  TImageList *ImageList1;
private:
  MESSAGE void __fastcall WMNotify(
    TMessage& Msg);
public:
  __fastcall TForm1(TComponent* Owner);
```

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    WM_NOTIFY, TMessage, WMNotify)
END_MESSAGE_MAP(TForm)
};
```

Let me remind you of what this code is doing. The `WM_NOTIFY` message, as its name suggests, is the tree-view's way of notifying its parent window of specific events. In this case, because `TreeView1`'s parent window is `Form1`, I handle the `WM_NOTIFY` message by tapping into `Form1`'s window procedure via the `Dispatch()` method. The `WMNotify()` method can now be defined as follows:

```
void __fastcall TForm1::
  WMNotify(TMessage& Msg)
{
  // grab a pointer to the tree-view's
  // notification structure
  const NM_TREEVIEW* pnmtv =
    reinterpret_cast<const NM_TREEVIEW*>
      (Msg.LParam);

  // if the notification is NM_CUSTOMDRAW
  // and its from the tree-view
  if (pnmtv->hdr.code == NM_CUSTOMDRAW &&
      pnmtv->hdr.hwndFrom ==
      TreeView1->Handle)
  {
    // Custom Draw stuff...
  }

  // pass everything else on
  TForm::Dispatch(&Msg);
}
```

Notice from this code that I look specifically for the `NM_CUSTOMDRAW` notification code. This step is necessary because I'm interested only in those events that correspond to the tree-view's drawing cycle. Also, notice that the `NMHDR::hwndFrom` data member is tested for equality with the tree-view's handle. This step ensures that only the notifications that are sent from `TreeView1` are handled. (This test isn't really needed in this ideal case because the form doesn't contain any other child controls.)

So far, so good, but, where's the code to handle the "Custom Draw stuff"? Well, I didn't provide that code because you'll rarely handle the `WM_NOTIFY` message in this way. Remember, this implementation assumes—and will work only if—`TreeView1` is a direct child of `Form1`. If you later decide that your

tree-view will look better if it's placed on, say, a `TPanel` object, then this code won't work. Instead, you'd have to handle the `WM_NOTIFY` message that's sent to the panel (i.e., to the tree-view's new parent window). To avoid this hassle the VCL provides the `CN_NOTIFY` message, which is a reflected version of `WM_NOTIFY` that's sent back to the tree-view itself. Because you handle this reflected message within the tree-view's window procedure, you don't have to worry about which window the tree-view is placed on.

Handling the CN_NOTIFY message

I just mentioned that the `CN_NOTIFY` message is a reflected version of the `WM_NOTIFY` message. Because of this fact, you handle the `CN_NOTIFY` message in the exact same way as you'd handle the `WM_NOTIFY` message. The only difference is where you handle the message—i.e., in which window procedure. In this case it is done within a descendant class, like so:

```
class TMyTreeView : public TTreeView
{
public:
  __fastcall TMyTreeView(
    TComponent* Owner);

private:
  MESSAGE void __fastcall CNNotify(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CN_NOTIFY, TMessage, CNNotify)
END_MESSAGE_MAP(TTreeView)
};
```

And, here's the definition of the `TMyTreeView::CNNotify()` method:

```
void __fastcall TMyTreeView::
  CNNotify(TMessage& Msg)
{
  // grab a pointer to the
  // tree-view's notification structure
  const NM_TREEVIEW* pnmtv =
    reinterpret_cast<const NM_TREEVIEW*>
      (Msg.LParam);
```

```
// if the notification is NM_CUSTOMDRAW
if (pnmtv->hdr.code == NM_CUSTOMDRAW)
{
  // grab a pointer to the tree-view's
  // Custom Draw structure
  NMTVCUSTOMDRAW* ptvcd =
    reinterpret_cast<NMTVCUSTOMDRAW*>
      (Msg.LParam);

  switch(ptvcd->nmcd.dwDrawStage)
  {
    // prior to painting...
    case CDDS_PREPAINT:
    {
      // tell the tree-view we want
      // individual notification of
      // each item being drawn
      Msg.Result = CDRF_NOTIFYITEMDRAW;
      break;
    }

    // upon notification of
    // each item being drawn...
    case CDDS_ITEMPREPAINT:
    {
      // extract the handle to the node
      const HTREEITEM hItem =
        reinterpret_cast<
          const HTREEITEM>(
            ptvcd->nmcd.dwItemSpec
            );

      // grab a pointer to the node
      TTreeNode* Node =
        Items->GetNode(hItem);

      // extract the state of the node
      const UINT state =
        ptvcd->nmcd.uItemState;

      // if the node is selected
      // or at an odd ordinal position
      if ((state & CDIS_SELECTED) ||
          (Node->AbsoluteIndex % 2))
```

```
        {
          // change the background color
          ptvcd->clrTextBk =
            ColorToRGB(clHighlight);

          // change the text color
          ptvcd->clrText =
            ColorToRGB(clHighlightText);

          // tell the tree-view that
          // we changed an attribute
          Msg.Result = CDRF_NEWFONT;
        }
        // otherwise...
        else
        {
          // punt to the tree-view
          Msg.Result = CDRF_DODEFAULT;
        }
        break;
      }

      // let the tree-view
      // draw everything else
      default:
      {
        Msg.Result = CDRF_DODEFAULT;
        break;
      }
    }
  }
  else TTreeView::Dispatch(&Msg);
}
```

This code will render every other node in a selected state, regardless of whether the node is actually selected. You can begin to see here how multiselect functionality can be added to a standard tree-view control. At this point the code to render multiple nodes in a selected state is done; what is needed now is to add code to store and manage the selected state of each node.

Adding multiselect functionality

Listing A provides the definition of a `TTreeView` descendant class, which I've called `TMultiTreeView`. In the sections that follow, I'll show you how to implement each of its methods.

Getting and setting the selected state

As I mentioned earlier, the selected state of each node must be stored; the `TTreeNode::Data` property can be used for this purpose. Accordingly, I will define a couple of methods to get and set the value of the `Data` property:

```
inline bool __fastcall
  TMultiTreeView::IsNodeSelected(
    TTreeNode* Node
  )
{
  if (Node) return Node->Data;
  return false;
}

inline void __fastcall
  TMultiTreeView::SelectNode(
    TTreeNode* Node, bool select,
    bool redraw
  )
{
  if (Node && Node->Data !=
      reinterpret_cast<void*>(select))
  {
    Node->Data =
      reinterpret_cast<void*>(select);
    if (redraw) RedrawNode(Node);
  }
}
```

The definition of the `IsNodeSelected()` method is fairly simple: it just returns the value that's held in the specified node's `Data` property (implicitly casting the result to a Boolean value). The `SelectNode()` method serves a similar role: instead of reading the `Data` property, it assigns the value that's specified by the `select` parameter to the node's `Data` property. Notice that the `Data` property is changed only if the `select` parameter indicates a different value than what's currently indicated in the node's `Data` property. Also notice that the `redraw` parameter governs whether the `RedrawNode()` method—which I will explain next—is called.

Redrawing a node

After you change the selected state of a node, you'll need a way to incite the tree-view to redraw that node. Specifically, you'll want the tree-view to send its parent (and thus itself) a Custom Draw notification so that the `TMultiTreeView::CNNotify()` method—which I'll define shortly—can redraw the node to reflect its new state.

The `RedrawWindow()` API function can be used to redraw a specific rectangular-based area of a window. In this case, to redraw a specific node, you simply pass the `RedrawWindow()` function the target node's bounding rectangle, which you can retrieve by using the `TTreeNode::DisplayRect()` method. Here's the code:

```
inline void __fastcall
  TMultiTreeView::RedrawNode(
    TTreeNode* Node
  )
{
  if (Node)
  {
    const RECT R =
      Node->DisplayRect(true);
    RedrawWindow(
      Handle, &R, NULL,
      RDW_INVALIDATE | RDW_UPDATENOW
      );
  }
}
```

Note the use of the `RDW_UPDATENOW` flag in the call to `RedrawWindow()`. This specification ensures that the `CNNotify()` method will be called before `RedrawWindow()` returns. Now I'll explain how the `CNNotify()` method is defined.

Drawing the selected nodes

I've already explained the mechanism to draw multiple nodes in a selected state. Here's the definition of the `TMultiTreeView::CNNotify()` method:

```
void __fastcall TMultiTreeView::
  CNNotify(TMessage& Msg)
{
  // grab a pointer to the tree-view's
  // notification structure
  const NM_TREEVIEW* pnmtv =
    reinterpret_cast<const NM_TREEVIEW*>
```

```
    (Msg.LParam);

// if the notification is NM_CUSTOMDRAW
if (pnmtv->hdr.code == NM_CUSTOMDRAW)
{
  // grab a pointer to the tree-view's
  // Custom Draw structure
  NMTVCUSTOMDRAW* ptvcd =
    reinterpret_cast<NMTVCUSTOMDRAW*>
      (Msg.LParam);

  switch(ptvcd->nmcd.dwDrawStage)
  {
    // prior to painting...
    case CDDS_PREPAINT:
    {
      // tell the tree-view we want
      // individual notification of
      // each item being drawn
      Msg.Result = CDRF_NOTIFYITEMDRAW;
      break;
    }

    // upon notification of
    // each item being drawn...
    case CDDS_ITEMPREPAINT:
    {
      // extract the handle to the node
      const HTREEITEM hItem =
         reinterpret_cast<
          const HTREEITEM>(
            ptvcd->nmcd.dwItemSpec
            );

      // grab a pointer to the node
      TTreeNode* Node =
        Items->GetNode(hItem);

      // extract the state of the node
      const UINT state =
        ptvcd->nmcd.uItemState;

      // if the node is selected
      // or pseudo-selected
```

```
        if ((state & CDIS_SELECTED) ||
            IsNodeSelected(Node))
        {
          // change the background color
          ptvcd->clrTextBk =
            ColorToRGB(clHighlight);

          // change the text color
          ptvcd->clrText =
            ColorToRGB(clHighlightText);

          // tell the tree-view that
          // we changed an attribute
          Msg.Result = CDRF_NEWFONT;
        }
        // otherwise...
        else
        {
          // punt to the tree-view
          Msg.Result = CDRF_DODEFAULT;
        }
        break;
      }

      // let the tree-view
      // draw everything else
      default:
      {
        Msg.Result = CDRF_DODEFAULT;
        break;
      }
    }
  }
  else TTreeView::Dispatch(&Msg);
}
```

Notice that this method is nearly identical to the previous `CNNotify()` method, which was defined for the `TMyTreeView` class. Here, instead of just drawing every other node selected, I use the `IsNodeSelected()` method to decide which nodes to highlight.

Although I've now shown you how to toggle the selected state of each node, you'll still need to know *when* to call the `SelectNode()` method. Because there are two ways in which a user can select nodes—via the mouse and via the keyboard—you need to provide code to handle both of these situations.

Handling multiple selection via the mouse

Notice from **Listing A** that I've mapped the `WM_LBUTTONDOWN` message to the `WMLButtonDown()` method. This method will be called immediately after the user presses the left mouse button while the cursor is within the tree-view. You can use this opportunity to strategically call the `SelectNode()` method. Here's the code:

```
void __fastcall TMultiTreeView::
  WMLButtonDown(TMessage& Msg)
{
  // find the node that's hit
  const POINT PHit =
    {Msg.LParamLo, Msg.LParamHi};
  TTreeNode* HitNode =
    GetNodeAt(PHit.x, PHit.y);

  // if a node is hit
  if (HitNode)
  {
    // perform a hit test
    const TV_HITTESTINFO
      hit_info = {PHit.x, PHit.y};
    TreeView_HitTest(Handle, &hit_info);

    // if the node's label or icon is hit
    if (hit_info.flags & TVHT_ONITEM)
    {
      // extract the key-state flags
      const WPARAM keys = Msg.WParam;

      // if the control key is pressed
      if (keys & MK_CONTROL)
      {
        // if the node is selected
        if (HitNode->Selected ||
            IsNodeSelected(HitNode))
        {
          // deselect the node
          HitNode->Selected = false;
          SelectNode(HitNode, false);
          return;
        }
```

```
        // otherwise, select the
        // currently-selected node
        SelectNode(Selected, true);
      }

      // if the shift key is pressed
      else if (keys & MK_SHIFT)
      {
        // deselect all but from
        // HitNode to PreviousNode_
        SelectAllExcept(
          HitNode, PreviousNode_, false);

        // select all nodes from
        // HitNode to PreviousNode_
        SelectNodes(
          HitNode, PreviousNode_, true);
      }

      // otherwise, deselect all nodes
      else SelectAll(false);
    }

    // otherwise, if the user clicks
    // to the right of the nodes...
    else if (
      hit_info.flags & TVHT_ONITEMRIGHT)
    {
      // if the tree-view has focus
      if (Handle == GetFocus())
      {
        // deselect all nodes
        SelectAll(false);
        Selected = NULL;
      }
    }
  }
  // pass the message on
  TTreeView::Dispatch(&Msg);
}
```

The logic behind this code is as follows: First use the `TTreeView::GetNodeAt()` method to identify the node that's closest to the location that was clicked. Next, perform a hit test—by using the `TreeView_HitTest()` macro—to make sure that the node's label or icon was hit. If this condition is

satisfied, use the `WParam` value (that accompanies the `WM_LBUTTONDOWN` message) to determine if the control and/or shift keys were held down during the click.

If the control key is held down, simply toggle `HitNode`'s selected status. Note that when you deselect `HitNode`, you'll have to `return` immediately so that the tree-view doesn't make the node editable.

If the shift key is held down, select a range of nodes. Specifically, select the range from `HitNode` to the last node that was selected without using the shift key—i.e., to `PreviousNode_`. (I'll show you how to set the `PreviousNode_` member shortly.)

If neither the control key nor the shift key is held down, deselect all nodes and then let the tree-view highlight its real selected node as usual.

Finally, notice that—in addition to the `SelectNode()` method—the `WMLButtonDown()` method makes use of the `SelectNodes()`, `SelectAllExcept()`, and `SelectAll()` methods. I added these latter three methods to help keep things sane; they're defined in **Listing B**.

Handling multiple selection via the keyboard

To handle the case in which the user selects nodes via the keyboard, you use the `WM_KEYDOWN` message. From **Listing A**, you can see that this message is mapped to the `WMKeyDown()` method, which is defined as follows:

```
void __fastcall TMultiTreeView::
  WMKeyDown(TMessage& Msg)
{
  switch (Msg.WParam)
  {
    // if the up-arrow key is pressed
    case VK_UP:
    {
      // if the shift key is down
      if (GetKeyState(VK_SHIFT) < 0)
      {
        // get the previous visible node
        TTreeNode* PrevNode =
          Selected->GetPrevVisible();

        // toggle the currently selected
        // node based on the selected
        // state of the previous node
        SelectNode(
```

```
          Selected,
          !IsNodeSelected(PrevNode)
          );
      }
      // otherwise, deselect all nodes
      else SelectAll(false);
      break;
    }

    // if the down-arrow key is pressed
    case VK_DOWN:
    {
      // if the shift key is down
      if (GetKeyState(VK_SHIFT) < 0)
      {
        // get the next visible node
        TTreeNode* NextNode =
          Selected->GetNextVisible();

        // toggle the currently selected
        // node based on the selected
        // state of the next node
        SelectNode(
          Selected,
          !IsNodeSelected(NextNode)
          );
      }
      // otherwise, deselect all nodes
      else SelectAll(false);
      break;
    }
  }
  // pass the messages on
  TTreeView::Dispatch(&Msg);
}
```

The logic behind this code is as follows: If the user holds down the shift key while navigating down (i.e., pressing the down arrow), grab a pointer to the next visible node, and then toggle the status of the currently-selected node based on the status of this next node.

Why do you need to toggle the status of the currently selected node? Well, when the user holds down the shift key, this suggests that he or she intends to select a continuous range of nodes. If the next node isn't selected, this means that the user has pressed the down arrow to *add* the next node to his or her current

range. The tree-view will, by default, select the next node. But because the tree-view can have only one truly selected node, before it selects the next node, it will deselect the currently selected node. Therefore, when the user wants to add the next node to his or her range, you need to select the currently selected node (i.e., set the node's `Data` property) because the tree-view will deselect it. On the other hand, if the user holds down the shift key while navigating down, but the next node isn't selected, this means that the user wants to *remove* the currently selected from his or her range. In this case, you need to deselect the currently selected node. A similar argument can be made for the case in which the user has pressed the up arrow.

Finally, notice that there's no code to handle the case in which the user holds down the control key while pressing the arrows. For tree-views, the control key is used for scrolling via the keyboard.

Keeping things in sync

There's one final step that needs to be addressed. You'll want the `Data` property of tree-view's truly selected node to always indicate `true`. This way, you can retrieve the selected nodes by using only the `IsNodeSelected()` method (within a `while` loop). Otherwise, you'd also have to test the tree-view's `Selected` property. To keep the truly-selected node's `Data` property updated, you can use the `TTreeView::Change()` method like so:

```
void __fastcall TMultiTreeView::
  Change(TTreeNode* Node)
{
  // store the previous node that
  // was selected without using
  // the shift key
  if (GetKeyState(VK_SHIFT) >= 0)
  {
    PreviousNode_ = Node;
  }

  // select the to-be-truly-selected node
  SelectNode(Node, true, false);
  TTreeView::Change(Node);
}
```

Note that in addition to setting the `Data` property of the truly selected node, this is where the `PreviousNode_` member is set (when the shift key isn't pressed).

Conclusion

I've shown how to add basic multiple-selection support to a standard tree-view control. You can download the source code for the `TMultiTreeView` class from **www.bridgespublishing.com**. I must warn you, however, that this code isn't well suited for tree-views that contain numerous items (say, greater than 1000). The reason is that the `SelectNodes()` and `SelectAllExcept()` methods are relatively expensive because they use the `AbsoluteIndex` property. There are ways to speed things up (e.g., by using a `TList` object and/or comparing each node's bounding rectangle instead of using its `AbsoluteIndex` property), but I'll save that for another day. If you do have a large number of items, consider using a virtual-mode `TListView` object instead.

Listing A: *Definition of the TMultiTreeView class*

```
class TMultiTreeView : public TTreeView
{
public:
  __fastcall TMultiTreeView(TComponent* Owner);

  virtual bool __fastcall IsNodeSelected(
    TTreeNode* Node);
  virtual void __fastcall RedrawNode(
    TTreeNode* Node);
  virtual void __fastcall SelectNode(
    TTreeNode* Node, bool select,
    bool redraw = true);
  virtual void __fastcall SelectNodes(
    TTreeNode* NodeA, TTreeNode* NodeB,
    bool select);
  virtual void __fastcall SelectAll(
    bool select);
  virtual void __fastcall SelectAllExcept(
    TTreeNode* NodeA, TTreeNode* NodeB,
    bool select);

protected:
  DYNAMIC void __fastcall Change(
    TTreeNode* Node);

private:
  TTreeNode* PreviousNode_;
  MESSAGE void __fastcall CNNotify(
    TMessage& Msg);
  MESSAGE void __fastcall WMKeyDown(
    TMessage& Msg);
  MESSAGE void __fastcall WMLButtonDown(
    TMessage& Msg);
```

```
public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CN_NOTIFY, TMessage, CNNotify)
  MESSAGE_HANDLER(
    WM_KEYDOWN, TMessage, WMKeyDown)
  MESSAGE_HANDLER(
    WM_LBUTTONDOWN, TMessage, WMLButtonDown)
END_MESSAGE_MAP(TTreeView)
};
```

Listing B: *Definition of the TMultiTreeView::SelectNodes(), SelectAll(), and SelectAllExcept() methods*

```
//-----------------------------------
// Selects all nodes that lie in
// the range from NodeA to NodeB
// (or from NodeB to NodeA).
//-----------------------------------
void __fastcall
  TMultiTreeView::SelectNodes(
    TTreeNode* NodeA,
    TTreeNode* NodeB,
    bool select
   )
{
  TTreeNode* FirstNode = NULL;
  TTreeNode* LastNode = NULL;

  if (NodeA && NodeB)
  {
    if (NodeA->AbsoluteIndex <
        NodeB->AbsoluteIndex)
    {
      FirstNode = NodeA;
      LastNode = NodeB;
    }
    else
    {
      FirstNode = NodeB;
      LastNode = NodeA;
    }
  }
```

```
  else if (NodeA && !NodeB)
  {
    FirstNode = NodeA;
  }

  SelectNode(FirstNode, select);
  while (FirstNode != LastNode)
  {
    FirstNode = FirstNode->GetNext();
    SelectNode(FirstNode, select);
  }
}

//------------------------------------
// Selects all nodes.
//------------------------------------
void __fastcall
  TMultiTreeView::SelectAll(bool select)
{
  SelectNodes(
    Items->GetFirstNode(), NULL, select);
}

//------------------------------------
// Selects all nodes expect those in
// the range from NodeA to NodeB.
//------------------------------------
void __fastcall
  TMultiTreeView::SelectAllExcept(
    TTreeNode* NodeA,
    TTreeNode* NodeB,
    bool select
   )
{
  if (!NodeA || !NodeB) return;

  if (NodeA->AbsoluteIndex <
      NodeB->AbsoluteIndex)
  {
    if (NodeA->AbsoluteIndex > 0)
    {
      SelectNodes(
        Items->GetFirstNode(),
        NodeA->GetPrev(), select);
```

```
    }
    if (NodeB->AbsoluteIndex <
        Items->Count - 1)
    {
      SelectNodes(
        NodeB->GetNext(), NULL, select);
    }
  }
  else
  {
    if (NodeB->AbsoluteIndex > 0)
    {
      SelectNodes(
        Items->GetFirstNode(),
        NodeB->GetPrev(), select);
    }
    if (NodeA->AbsoluteIndex <
        Items->Count - 1)
    {
      SelectNodes(
        NodeA->GetNext(), NULL, select);
    }
  }
}
```

Managing to-do lists

by David Bridges

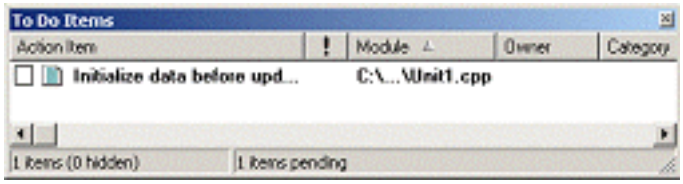
I have always liked Borland's IDE, ever since I started using Borland C++ back in the early 90's. At the time, it had as many features as some other full-blown third party code editors. With the introduction of Delphi and C++ Builder, the IDE has grown in complexity, with added features such as auto complete and Code Insight.

I recently stumbled upon one IDE feature that I really like—the To Do List (which Borland randomly calls "To Do List" or "To-Do List"). Each project maintains its own To Do List. To view it, select View|To-Do List from the menu. A window pops up which is similar to the "Tasks" list in Microsoft Outlook. This window allows you to add new items to the list. But what really makes the To Do list powerful it its ability to pull comments out of the source files in the project.

Any project file containing the word TODO: in a comment will have that comment listed in its To Do list. For example, the following code is reflected in **Figure A**:

```
// TODO: Initialize data before updating
```

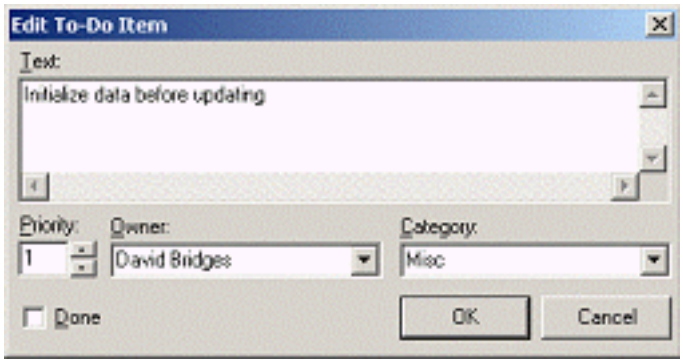
Figure A



The To Do Items window lists all of the to do items.

You can right-click on any To Do item in the window, and a dialog box will appear showing its properties as shown in **Figure B**:

Figure B



Edit To-Do Item dialog box

From here, you can assign the item to someone, categorize it, and set the priority. It then updates this information directly into your source code comment. After entering the information shown in the dialog box, the source code comment will look like this (the line is broken here for formatting reasons):

```
// TODO 1 -cMisc -oDavid Bridges:
  Initialize data before updating
```

Double-clicking on an item in the window will take you directly to the source code comment.

I work on many development projects, and some have very long To Do lists. My team uses additional software to handle big projects, some of which contain hundreds of detailed items. But for code-related notes, you can’t beat the To Do List window—especially when it’s late at night, you’re racing to get out of the office, and you want to jot down a note to get you started again next time.

Using Delphi code in C++Builder

by Kent Reisdorph

As you probably know, C++Builder was created from Delphi. Much of what C++Builder contains comes directly from Delphi. Sometimes this can be frustrating, but there are some benefits. There is a wealth of Delphi example code available (on the Internet and from other sources) that can be a great aid in developing C++Builder applications. In some cases this code can be used directly. I other case the code can be converted for use in C++Builder. Further, there are many example components available for Delphi for which there is no C++Builder example.

C++Builder has a built-in Pascal compiler. The Pascal compiler allows you to use Delphi code in C++Builder. It can also aid in converting code from Delphi to C++Builder. The Pascal compiler is available both from within the C++Builder IDE (transparently) or via a command line compiler.

Using Delphi units directly

Often you will find Delphi example projects that contain a unit that you want to use in your applications. The simplest way of using a Delphi unit is to simply add it to your project. Here are the steps to take to add a Delphi unit to a C++Builder project:

1. Create your C++Builder project.
2. Select Add to Project from the C++Builder toolbar or menu.
3. Select "Pascal unit" from the Files of Type combo box on the Open dialog box.
4. Select a Delphi unit to add to your project and click OK.
5. Build the application before writing any code that references the Delphi unit. This builds the Delphi unit and creates a header you can include in your application.
6. Choose File | Include Unit Hdr… from the C++Builder main menu and add the Delphi form to your application.
7. Write code that references the Delphi unit.

When you build the application, C++Builder uses its built-in Pascal compiler to create an OBJ that the application can use. The Pascal compiler also generates a header from the Pascal source. Using Delphi units this way is very easy.

Converting code

As you can see, adding a Delphi unit to your project is quite simple. However, you may not want to use a

Delphi unit in this way. You may, for example, require that all of your code be in C++. In this case you will have to convert the code from Pascal to C++.

There isn't any practical way for me to explain every detail of converting Delphi code to C++. What I can do, however, is show you how to easily convert some of the tricky Pascal declarations to C++.

Let's say, for example, that you have a Delphi unit that looks like this (obviously simplified for this example):

```
unit TestUnit;

interface

type
  MyEnum = (meOne, meTwo, meThree);
  function DoSomething(Value : MyEnum) :
    string;
var
  I : Integer;
  Buffer : array [0..255] of Char;

implementation

  function DoSomething(Value : MyEnum) :
    string;
  begin
    case Value of
      meOne : Result := 'One';
      meTwo : Result := 'Two';
      meThree : Result := 'Three';
    end;
  end;

end.
```

Even without knowing Pascal you can probably manage to convert this unit manually. However, you can get a head start by using C++Builder's Pascal compiler to create a header for this unit. You could add the unit to your C++Builder application and compile, but you can also use the command line compiler. Here are the steps:

1.Open a command prompt box and navigate to the folder containing the Delphi unit.

2.At the command prompt, type the following:

```
dcc32 -jphn TestUnit.pas
```

DCC32.EXE is the Pascal compiler. The `-jphn` switch tells the Pascal compiler to create a header and an OBJ file compatible with C++Builder. When this command executes, the file is compiled and a header and OBJ are created (the OBJ is not really significant in this case since you won't be using it anyway). The header generated for the test unit looks like this (with comment lines removed for clarity):

```
#ifndef TestUnitHPP
#define TestUnitHPP

#pragma delphiheader begin
#pragma option push -w-
#pragma option push -Vx
#include <SysInit.hpp>    // Pascal unit
#include <System.hpp>    // Pascal unit

namespace Testunit
{

#pragma option push -b-
enum MyEnum { meOne, meTwo, meThree };
#pragma option pop

extern PACKAGE int I;
extern PACKAGE char Buffer[256];
extern PACKAGE AnsiString __fastcall
  DoSomething(MyEnum Value);
}    /* namespace Testunit */

#if !defined(NO_IMPLICIT_NAMESPACE_USE)
using namespace Testunit;
#endif
#pragma option pop    // -w-
#pragma option pop    // -Vx

#pragma delphiheader end.

#endif    // TestUnit
```

This is still a bit muddied by various compiler options, but here is the relevant part:

```
enum MyEnum { meOne, meTwo, meThree };
int I;
```

```
char Buffer[256];
AnsiString __fastcall
  DoSomething(MyEnum Value);
```

Notice how the declarations are conveniently created for you. You still have to convert the actual code in the unit, but at least the declarations are done for you.

Here's another example, only slightly more complex:

```
const
  MaxSize = MaxLongInt;
type
  TDoubleArray = array[0..
    (MaxSize div SizeOf(Double))-1]
    of Double;
  PDoubleArray = ^TDoubleArray;
  TIntArray = array[0..
    (MaxSize div SizeOf(Integer))-1]
    of Integer;
  PIntArray = ^TIntArray;
```

The generated declaration looks like this:

```
typedef double TDoubleArray[268435455];
typedef double *PDoubleArray;
typedef int TIntArray[536870911];
typedef int *PIntArray;
```

The previous examples are fairly simple. Some Pascal declarations, however, can leave you scratching your head wondering how to convert them to C++. Here's an example:

```
TMyCallback = function(const S : string;
  Size : Integer) : Integer;
```

This is a declaration for a callback function. When you compile this unit with the Pascal compiler you get a header that contains this declaration:

```
typedef int __fastcall (*TMyCallback)
  (const AnsiString S, int Size);
```

Perhaps you would have easily figured out how to convert the Pascal code to this declaration, but it is unlikely unless you have expertise with both Pascal and C++. The point is, of course, that the header

generation feature of the Pascal compiler makes it simple to convert any Pascal declaration to C++.

I could provide even more complex examples, but I think you get the point.

Using Delphi components

There are many example, shareware, and freeware components available for Delphi. In most cases, the author does not include a C++Builder equivalent. Components including Delphi source code can typically be used with little or no modification. The steps to use a Delphi component are:

1.Create a new package for the component. Typically you will create a package that is both a run-time and a design package.

2.Add the Delphi source code for the component to the package.

3.Build the package and install it.

Granted, this process is simple, but many C++Builder programmers don’t realize that Delphi components can be used in this way.

Conclusion

There is a large amount of Delphi code available on the Internet. The ability to use this code in your applications is certainly a benefit. Knowing that you can use this code, and exactly how to use it is the key.

Get the message out

by Mark G. Wiseman

At some point in all but the simplest program you will need to display a message box to the user. You may want to inform the user of something that has occurred in the program or you may want to get a simple yes-or-no answer from the user.

There are a variety of ways you can display message boxes in your programs. In this article I'm going to discuss several of these ways and I'm going to show you three simple inline functions that will make using message boxes easier.

Let's assume that you are writing a VCL-based program and that you want to display a message box inside a `TMyForm`, which is derived from `TForm`. You want the caption of the message box to be the title of the application and the message box to display a message contained in an `AnsiString` object. The message box should make a sound when first displayed and then wait until the user acknowledges the message by clicking OK. How can it be done?

Windows API

Using the Windows API to create and display a message box is easy. The most common method is to use the `MessageBox()` function. Here is a simple code fragment:

```
AnsiString msg = "This is my message.";
MessageBox(Handle, msg.c_str(),
  Application->Title.c_str(),
  MB_OK | MB_ICONINFORMATION);
```

There are two lines of code. The first argument to the `MessageBox()` function is the window handle of `TMyForm`. The second is the message to display and the third is the application's title. The fourth argument consists of flags that tell the message box to display an OK button and an icon indicating that the message box is providing information to the user.

Although using this Windows API function is not hard, it could be easier. You have to pass a window handle. You cannot pass an `AnsiString` for the message, you have to pass a `char *` by calling the `AnsiString` function `c_str()`. You have to explicitly pass the Application's title, again using `c_str()`. And, finally, you have to explicitly pass in the flags for the OK button and the icon.

There are two more Windows API functions that create message boxes. They are `MessageBoxEx()` and `MessageBoxIndirect()`. I won't discuss them here; but you can look them up in the online

help for the Windows API.

VCL message boxes

As it turns out, the VCL offers several functions for creating message boxes and they are somewhat more convenient to use than the Windows API.

The first is a method contained in the `TApplication` class. It is also named `MessageBox()`. Here is an example:

```
AnsiString msg = "This is my message.";
Application->MessageBox(msg.c_str(),
  Application->Title.c_str(),
  MB_OK | MB_ICONINFORMATION);
```

As you can see, this method is nearly identical to the Windows API `MessageBox()`. The only difference is that `TApplication::MessageBox()` eliminates the window handle as an argument. In fact, I think this method is too much like the Windows API version. It is a VCL function and it won't accept an `AnsiString` for either the message or the caption arguments!

Another VCL function is `ShowMessage()`. This function takes only one argument, an `AnsiString` containing the message to be displayed. Here is an example of code using `ShowMessage()`:

```
AnsiString msg = "This is my message.";
ShowMessage(msg);
```

This couldn't be any be easier; but it turns out to be too good to be true. `ShowMessage()` does not display an icon and it does not make any sounds when displayed.

There are some other VCL functions that will also display a message box. They are `ShowMessageFmt()`, `MessageDlg()` and `MessageDlgPos()`. Each of these three functions suffers from its Delphi heritage though. `ShowMessageFmt()` allows you to format the message using multiple parameters. However, it does involve using `TVarRec` which is very awkward to use in C++. `MessageDlg()` and `MessageDlgPos()` take arguments that are `Sets` from Delphi and are also awkward in C++.

Write your own

The answer to the inconveniences of the VCL functions is to write your own message box functions. These can take advantage of C++ features to make programming message boxes even easier. Here are

three functions you can use:

```
inline int MessageBox(
  char *msg, unsigned short flags =
    MB_OK | MB_ICONINFORMATION)
{
  return(Application->MessageBox(msg,
    Application->Title.c_str(), flags));
}

inline int MessageBox(
  String msg, unsigned short flags =
    MB_OK | MB_ICONINFORMATION)
{
  return(Application->MessageBox(
    msg.c_str(),
    Application->Title.c_str(), flags));
}

inline int MessageBox(int strID,
  unsigned short flags =
    MB_OK | MB_ICONINFORMATION)
{
  return(Application->MessageBox(
    LoadStr(strID).c_str(),
    Application->Title.c_str(), flags));
}
```

All three functions are inline, so they are very efficient. They all use the `TApplication::MessageBox()` function internally, so you don't need to pass in a window handle. They also fill in the caption for the message box using the application's title.

Finally, they all use C++ default arguments to pass in flags for an OK button and the information icon. In case you are wondering, using any of the `MB_ICONxxx` flags also causes Windows to play a sound associated with the icon, so these functions all play a sound. Here's the revised code using the new `MessageBox()` function:

```
AnsiString msg = "This is my message.";
MessageBox(msg);
```

This uses the first of the three inline functions. It is just as easy as the VCL function, `ShowMessage()`; but it displays an icon and makes a sound.

The other two inline functions work in a similar fashion. The second takes a `char *` instead of `AnsiString` and the third takes an integer representing a string resource.

Final message

There is a demonstration program on the Bridges Publishing Web site for this article. The code includes a header file, `MessageBox.h` with the three inline functions. Include this file in your programs and get the message out.

Including hot links in TRichEdit

by Mark G. Wiseman

The `TRichEdit` control included in the VCL does a nice job of encapsulating the Windows rich edit control. However, it leaves out a lot of the functionality found in more recent versions of this control.

In this article I am going to show you how to display a working or *hot link* URL in a `TRichEdit` component. If you left-click this URL, your Web browser will be launched and the link will be loaded into the browser.

I've written a sample application that demonstrates this. **Figure A** shows this application with a link to the URL www.bridgespublishing.com, the Reisdoprh Publishing Web site. By clicking on this underlined link, your browser should launch and display the home page for this site.

Let's cook up some hot links.

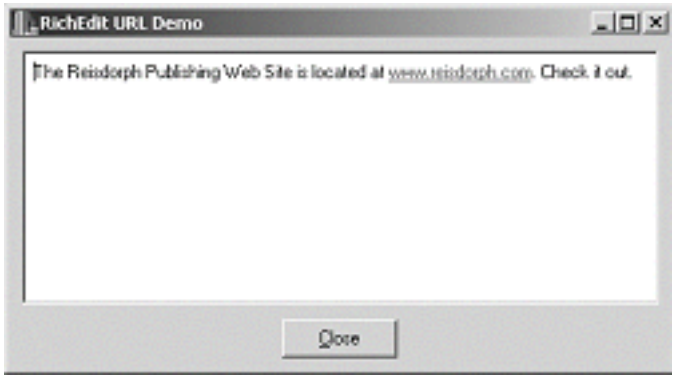


Figure A: *Sample application showing a hot URL link in a rich edit control*

Spicy or mild?

For the code described in this article to display a hot link, you will need to have version 2.0 or greater of the Windows rich edit control. If you use this code on a computer that has version 1.0, `TRichEdit` will still display the link as ordinary text.

You could test for the version of the rich edit control, but it really isn't necessary since the code is designed to run on any version of the rich edit control. You will just lose the hot link functionality on version before 2.0.

Ingredients

The relevant code from the sample application is shown in **Listing A**. There is a `TRichEdit` component, named `RichEdit1` on the `TMainForm` form. In the form's constructor, I get the event mask for `RichEdit1` by sending an `EM_GETEVENTMASK` message to the component. I then add the `ENM_LINK` event to the value of the mask and set the new mask in `RichEdit1` by sending it to `RichEdit1` with an `EM_SETEVENTMASK` message.

The event mask tells the rich edit control which messages it should send when things happen in the control. Once `ENM_LINK` is included in the mask, the `EN_LINK` message is sent when a link (URL) is clicked with the mouse.

In the form's constructor, I also send `RichEdit1` an `EM_AUTOURLDETECT` message. This tells the control that it should examine the text that is entered into it and automatically determine if the text is a link or URL.

Finally, I assign some text to `RichEdit1` that contains a link, www.bridgespublishing.com.

Simmer until done

All that's left to do is catch any `EN_LINK` messages that are sent to the application and act on those messages. I do that by overriding the virtual function `WinProc()` found in `TForm`.

In this function, I look for any `WM_NOTIFY` messages that contain the code `EN_LINK`. If such a message is received, the `lParam` member of the message will be a pointer to an `ENLINK struct`. Here is the definition for this `struct`:

```
typedef struct _enlink
{
  NMHDR nmhdr;
  UINT msg;
  WPARAM wParam;
  LPARAM lParam;
  CHARRANGE chrg;
} ENLINK;
```

I examine the `msg` member of the `ENLINK struct` and if it is equal to `WM_LBUTTONDOWN`, then a link has been clicked with the left button of the mouse.

The `chrg` member of the `ENLINK struct` contains the range of characters that make up the link. I select the characters by sending an `EM_EXSETSEL` message to `RichEdit1` with this range.

Now the text that makes up the link is accessible through the `SelText` property of `RichEdit1`. I use this text as input to the `ShellExecute()` function of the Windows API. The `ShellExecute()` function will take care invoking the browser.

All other messages to `WinProc()` are passed on to `TForm::WinProc()`.

Secret recipe?

If you try to look up the `structs` and messages from the Windows API in the online help included with C++Builder, you won't find them. They are not secret though. Look on the Microsoft Developer Network Web site, www.msdn.Microsoft.com and do a search for them there.

Listing A: ***Code to display a hot link in a TRichEdit component***

```
__fastcall TMainForm::TMainForm(
  TComponent* Owner) : TForm(Owner)
{
  unsigned mask = SendMessage(RichEdit1->Handle,
    EM_GETEVENTMASK, 0, 0);
  SendMessage(RichEdit1->Handle, EM_SETEVENTMASK,
    0, mask | ENM_LINK);
  SendMessage(RichEdit1->Handle, EM_AUTOURLDETECT,
    true, 0);

  RichEdit1->Text = "The Bridges Publishing Web"
    " Site is located at www.bridgespublishing.com. Check"
    " it out.";
}

void __fastcall TMainForm::WndProc(
  Messages::TMessage &Message)
{
  if (Message.Msg == WM_NOTIFY)
  {
    if (((LPNMHDR)Message.LParam)->code ==EN_LINK)
    {
      ENLINK* p = (ENLINK *)Message.LParam;
      if (p->msg == WM_LBUTTONDOWN)
      {
        SendMessage(RichEdit1->Handle,
          EM_EXSETSEL, 0, (LPARAM)&(p->chrg));
        ShellExecute(Handle, "open",
```

```
            RichEdit1->SelText.c_str(), 0, 0,
            SW_SHOWNORMAL);
        }
      }
    }

    TForm::WndProc(Message);
}
```

Preprocessor macros

by David Bridges

Preprocessor macros are used in all C++ Builder programs, and are one of the most basic building blocks of any C++ project. When a source file is compiled, it is first processed by the "preprocessor". Some commonly used preprocessor symbols are the familiar `#include` and `#define` directives.

The Borland preprocessor

The preprocessor is responsible for creating the final C++ code, which is given to the compiler. This includes bringing in all header files and stripping out comments. To see for yourself how this works, you can create a small program such as the "hello" listing shown below. To see the output created by the preprocessor, go to the command prompt and type `bcc32 hello.cpp.` This will create an output file called `hello.i` containing the output from the preprocessor. As you can see, lots of stuff was added before the first line of code. If you want to see a *huge* amount of code returned by the preprocessor, try including `<windows.h>` at the top—in this case, you will want to use `bcc32 -P- hello.cpp`. The `P' option prevents the addition of source file location comments. Without it, the `hello.i` file will be approximately 20 MB! The difference between `#include <filename>` and `#include "filename"` is the way in which the preprocessor searches for the file. With `<filename>`, only the directories listed under the project "Include Path" will be searched. With "`filename`", the preprocessor first looks in the same directory as the source file that includes it, then the current directory, and then the project include path.

```
// hello.cpp
#include <stdio.h>
void main()
{
  printf("hello");
}
```

Simple macros

In addition to including files, the preprocessor also processes macros. A macro is anything that begins with `#define`. This can be as simple as defining logical names for constants, such as the following line:

```
#define ERROR_VALUE –1
```

The `#define` directive can also be used for more interesting things. For example, the following macro mimics the `max()` function:

```
#define MAX(x,y) (x > y ? x : y)
```

The macro is named `MAX` to differentiate it from the run-time library `max()` function. The difference between a macro "function" and a real C++ function is that the macro doesn't really execute the function—it just performs text replacement for the specified arguments. Because of this, you must use caution when using macros to perform arithmetic functions. For example, suppose you use the `MAX` macro function defined above like this:

```
z = MAX(++x, y);
```

The precompiler would expand it to:

```
z = (++x > y ? ++x : y);
```

Not the intended result!

Conditional processing

Preprocessor directives can be used to conditionally compile parts of a source file. The `#ifdef` and `#ifndef` directives evaluate whether a certain macro is or is not defined. One common use of these conditionals is in DLL projects, where a function should be declared with the `_export` keyword in the DLL project, and with the `_import` keyword for applications, which use the DLL. In this case, the code would look something like this:

```
#ifdef DLL
  void _export f();
#else
  void _import f();
#endif
```

These are also commonly used in header files to prevent the header from being compiled more than once for each source file. C++ Builder automatically inserts a `#ifndef` block around the header for all new modules that it adds to a project. For example, if you add a new unit to a project, the unit header file will look like this:

```
#ifndef Unit1H
#define Unit1H
```

```
// This code will be compiled only
// once, no matter how many times
// the file is included.
// ...
#endif
```

Another common use of `#ifdef` is to make code portable across different platforms. For example, if you're writing code, which will be compiled on both Windows and Macintosh platforms, it might look like the following:

```
#ifdef MACINTOSH
  sFilename = GetMacFilename();
#else
  sFilename = GetWindowsFilename();
#endif
```

Another helpful preprocessor directive is `#error`. This can be useful in tracking down where the precompiler is when unknown errors are occurring. For example:

```
#ifdef FOO
  // ...
  #error "FOO is defined!"
  // ...
#endif
```

This will display a compiler error such as "Error in file.cpp line xxx: error FOO is defined!"

Keystroke-saving macros

I like to make the preprocessor do mundane typing jobs for me. I usually define a string copy macro such as the following when copying lots of character arrays:

```
#define NCOPY(x,y) \
  lstrcpyn(x, y, sizeof(x));
```

This is especially useful if `x` is something like "mystruct.somefieldname".

Special operators

There are some lesser-known precompiler operators, which can be really useful:

Operator	Use
\	This is the line continuation character. It lets you define a precompiler macro on more than one line.
#	When placed before a macro argument, it turns the macro argument into a string.
##	When placed between two symbols, it concatenates the symbols together.

Here is an example that uses all three of these to add colors to a combo box. The color names are added as the combo box strings, and the color values are added as the associated objects.

```
#define ADD_COLOR(color) \
  cmbColors->Items->AddObject(#color, (TObject*)cl##color);

  ADD_COLOR(Aqua)
  ADD_COLOR(Black)
  ADD_COLOR(DkGray)
  ADD_COLOR(Fuchsia)
  ADD_COLOR(Gray)
  ADD_COLOR(Green)
  ADD_COLOR(Lime)
  ADD_COLOR(LtGray)
  ADD_COLOR(Maroon)
  ADD_COLOR(Navy)
  ADD_COLOR(Olive)
  ADD_COLOR(Purple)
  ADD_COLOR(Red)
  ADD_COLOR(Silver)
  ADD_COLOR(Teal)
  ADD_COLOR(White)
  ADD_COLOR(Yellow)
```

Here’s another example that displays application error messages:

```
void DisplayError(int iError)
{
  #define SHOW_ERR(err)           \
  if (iError == err) {            \
```

```
    MessageBox(                    \
     NULL, #err, "Error", MB_OK);  \
  }

  SHOW_ERR(ERROR_NOT_ENOUGH_MEMORY)
  SHOW_ERR(ERROR_BAD_ENVIRONMENT)
  SHOW_ERR(ERROR_INVALID_ACCESS)
  SHOW_ERR(ERROR_INVALID_DATA)
  SHOW_ERR(ERROR_OUTOFMEMORY)
  SHOW_ERR(ERROR_INVALID_DRIVE)
  // Others...
}
```

Conclusion

I hope you've learned some useful things about preprocessor macros in this article. They can be used to save repetative typing, reduce coding errors, and make your code more portable.

Quick and easy re-sizeable controls

by Damon Chandler

In this article, I'll show you how to make any `TWinControl` descendant re-sizeable and movable at run time.

Moving and resizing a form

We all know that standard top-level forms can be moved and resized at run time. Specifically, a `TForm` object whose `BorderStyle` property is `bsDialog`, `bsSingle`, `bsToolWindow`, `bsSizeable`, or `bsSizeToolWin` can be moved at run time; and a `TForm` object whose `BorderStyle` property is `bsSizeable` or `bsSizeToolWin` can also be resized at run time.

How exactly does the `BorderStyle` property work? Well, it turns out that this property does little more than adjust the window styles that are passed to the `CreateWindowEx()` API function. You can see this by examining the following excerpt from the `TCustomForm::CreateParams()` method:

```
void __fastcall TCustomForm::
  CreateParams(TCreateParams& Params)
{
  TScrollingWinControl::
    CreateParams(Params);

  // ...
  switch (BorderStyle)
  {
    case bsNone:
    {
      Params.Style |= WS_POPUP;
      break;
    }
    case bsSingle:
    case bsToolWindow:
    {
      Params.Style |=
        WS_CAPTION | WS_BORDER;
      break;
    }
    case bsSizeable:
    case bsSizeToolWin:
```

```
    {
      Params.Style |=
        WS_CAPTION | WS_THICKFRAME;
      break;
    }
    case bsDialog:
    {
      Params.Style |=
        WS_CAPTION | WS_POPUP;
      Params.ExStyle |=
        WS_EX_DLGMODALFRAME |
        WS_EX_WINDOWEDGE;
     }
  }
  // ...
}
```

Recall that the primary role of the `CreateParams()` method is to initialize a `TCreateParams`-type variable whose data members will subsequently be passed to the `CreateWindowEx()` API function. Accordingly, there are two main things that you should notice from this condensed definition of the `TCustomForm::CreateParams()` method. First, observe that the `WS_CAPTION` window style is specified for all `BorderStyle` enumerators that allow the form to be moved—i.e., `bsDialog`, `bsSingle`, `bsToolWindow`, `bsSizeable`, and `bsSizeToolWin`. This makes sense because the standard way to move a form is to click on its caption. Second, notice that the `WS_SIZEBOX` window style is specified for the enumerators that allow the form to be resized (`bsSizeable` and `bsSizeToolWin`). This latter style creates a window with the familiar sizing border.

In short, when you adjust the `BorderStyle` property, you're indirectly specifying whether the `WS_CAPTION` style (for moving) and/or the `WS_SIZEBOX` style (for resizing) are used. When these styles are specified, Windows provides built-in support for run time moving and resizing. Next I'll explain how to use this functionality in a more general sense.

Moving and resizing any TWinControl

Unfortunately, not every `TWinControl` descendant class provides a `BorderStyle` property. But now that you know exactly what happens when this property is manipulated, you really don't need the `BorderStyle` property; instead, you can directly manipulate the window styles by using a combination of the `GetWindowLong()` and `SetWindowLong()` API functions. Here are the declarations for these functions:

```
LONG GetWindowLong(
 HWND hWnd, int nIndex);
```

```
LONG SetWindowLong(
 HWND hWnd, int nIndex, LONG dwNewLong);
```

Both of these functions require a handle to a window as the first parameter (`hWnd`) and an identifier as the second parameter (`nIndex`). To retrieve or specify a particular window style, you pass `GWL_STYLE` constant (`-16`) as the `nIndex` parameter. The new style will be returned by each function. To change the window's current style, you use the `SetWindowLong()` function, specifying the new styles as the `dwNewLong` parameter. In this case the function will return the previous styles. (Note that users of newer versions of C++Builder might wish to use the `GetWindowLongPtr()` and `SetWindowLongPtr()` functions instead.) Let's look at an example.

Toggling the `WS_SIZEBOX` style

Figure A depicts a form (`Form1`) that contains two child controls: a `TPanel` and a `TCheckBox` component.

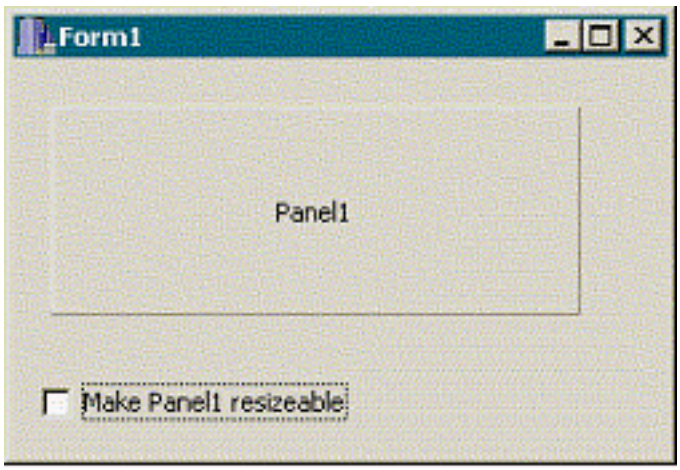


Figure A: *A form with a panel and a check box.*

As you can probably guess from its caption, we'll use the check box's `OnClick` event handler to toggle the resizeability of the panel. Remember, to make the panel re-sizeable, you add the `WS_SIZEBOX` style by using the `GetWindowLong()` and `SetWindowLong()` functions. Similarly, to make the panel not sizeable, you use these functions to remove the `WS_SIZEBOX` style. Here's the code:

```
#include <cassert>
void __fastcall TForm1::
  CheckBox1Click(TObject *Sender)
{
  // grab a handle to the panel
  const HWND hPanel = Panel1->Handle;

  // get the current styles
```

```
  const LONG current_style =
    GetWindowLong(hPanel, GWL_STYLE);
  assert(current_style);

  if (CheckBox1->Checked)
  {
    // add the WS_SIZEBOX style
    const BOOL ok = SetWindowLong(
      hPanel, GWL_STYLE,
      current_style | WS_SIZEBOX
      );
    assert(ok);
  }
  else
  {
    // remove the WS_SIZEBOX style
    const BOOL ok = SetWindowLong(
      hPanel, GWL_STYLE,
      current_style & ~WS_SIZEBOX
      );
    assert(ok);
  }

  // have the panel redraw its border
  const bool ok = SetWindowPos(
    hPanel, 0, 0, 0, 0, 0,
    SWP_FRAMECHANGED | SWP_NOMOVE |
    SWP_NOSIZE | SWP_NOZORDER
    );
  assert(ok);
}
```

Note the use of the `SetWindowPos()` API function in this code snippet. This call is needed to instruct the panel to redraw its border (to reflect the new style). **Figure B** depicts the results of this code when the check box is checked.

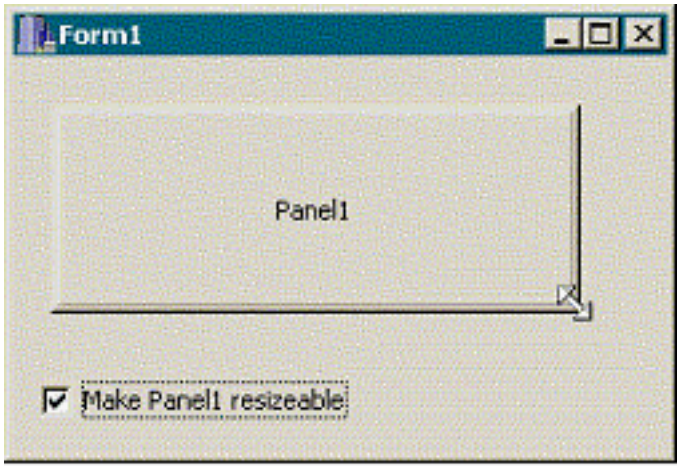


Figure B: *A panel that has been made re-sizeable by adding the `WS_SIZEBOX` style.*

Implementing moveability

As we all know, the standard way to move a window is to click its caption and then drag it. And as I mentioned earlier, the `WS_CAPTION` style determines whether or not a form has a caption. For a child control—such as our panel—however, it might look weird to display a caption just so the control can be moved at run time. As it turns out, the `WS_CAPTION` style isn't really needed for run time moveability.

During a window's lifetime, the window communicates with Windows via a series of messages. One of these messages, `WM_NCLBUTTONDOWN`, is sent (by Windows) to a window to instruct it that the user has pressed the left mouse button while the cursor was located within the window's non-client area. When this non-client area is the window's caption, the standard default window procedure handles the `WM_NCLBUTTONDOWN` message by initiating and managing a move operation.

How is the `WM_NCLBUTTONDOWN` message useful for moving child controls? Well, it turns out that the standard default window procedure will initiate and manage the window's move operation *regardless* of whether or not the window has a caption. All you need to do is fool the window into thinking that the user has clicked its (non-existent) caption. To do this, you use the `SendMessage()` API function—specifying `WM_NCLBUTTONDOWN` as the `Msg` parameter and `HTCAPTION` as the `wParam` parameter—from within the window's `OnMouseDown` event handler, like so:

```
void __fastcall TForm1::Panel1MouseDown(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  // release current mouse capture
  ReleaseCapture();

  // fake a caption hit
```

```
  SendMessage(Panel1->Handle,
    WM_NCLBUTTONDOWN, HTCAPTION, 0);
}
```

Using the `WM_NCLBUTTONDOWN` message and the `WS_SIZEBOX` style for run time moveability and resizeabilty has one main advantage: the code is extremely simple. Namely, you don't have to worry about managing the sizing cursors and/or clipping the cursor to the bounds of the parent window; and you don't have to worry about adjusting the size or the position of the window after the resize or move operation. These nitty-gritty details are handled by Windows (specifically, the standard default window procedure). Moreover, because Windows is handling the moving and resizing, the current system display settings determines whether or not the window's contents are shown during the move/resize operation.

Of course, simplicity does have its limitations. Namely, by adding the `WS_SIZEBOX` style, the child window will always display the resizing border (as in **Figure B**); this might not be practical for some situations or you might simply dislike the look. In addition, you don't have much control over the moving/sizing operation. If you want to display the window's current position and dimensions in a tooltip, for example, it'll take a bit more work.

Controlling the moving and sizing operations

Because Windows does most of the moving/resizing work, it might seem like a major chore to determine when a move/resize operation is occurring and then control the operation. Remember though, these are the same move and resize operations that occur when the user moves and resizes a regular form. To get notification of (and control over) these operations for a regular form, you'd handle the `WM_MOVING` and `WM_SIZING` messages. Well, the same approach can be used for all `TWinControl` descendants. In this case, you simply handle the `WM_MOVING` and `WM_SIZING` messages that are sent to the child window. The following sections illustrate this through an example.

Creating a movable and re-sizeable TScrollBox descendant class

Listing A contains the declaration of a `TScrollBox` descendant class (`TScrollBoxEx`), which I'll use to demonstrate two things:

1.How to incorporate the moving and resizing functionality into a class.

2.How to handle the `WM_MOVING` and `WM_SIZING` messages.

The `TScrollBoxEx` class has two private members: `MoveResize_` and `OldCursor_`. The `MoveResize_` member is a Boolean that's used with the `MoveResize` property to allow you to easily

toggle the WS_SIZEBOX style. Assigning a value to the MoveResize property invokes the DoSetMoveResize() method, which is defined as follows:

```
void __fastcall TScrollBoxEx::
  DoSetMoveResize(bool value)
{
  if (MoveResize_ != value)
  {
    MoveResize_ = value;
    const HWND hWndThis = Handle;

    // get the current styles
    const LONG current_style =
      GetWindowLong(hWndThis, GWL_STYLE);
    assert(current_style);

    if (MoveResize_)
    {
      // change the cursor to crSizeAll
      OldCursor_ = Cursor;
      Cursor = crSizeAll;

      // add the WS_SIZEBOX style
      const BOOL ok = SetWindowLong(
        hWndThis, GWL_STYLE,
        current_style | WS_SIZEBOX
        );
      assert(ok);
    }
    else
    {
      // restore the cursor
      Cursor = OldCursor_;

      // remove the WS_SIZEBOX style
      const BOOL ok = SetWindowLong(
        hWndThis, GWL_STYLE,
        current_style & ~WS_SIZEBOX
        );
      assert(ok);
    }

    // have the panel redraw its border
    const bool ok = SetWindowPos(
```

```
      hWndThis, 0, 0, 0, 0, 0,
      SWP_FRAMECHANGED | SWP_NOMOVE |
      SWP_NOSIZE | SWP_NOZORDER
      );
    assert(ok == true);
  }
}
```

Notice that this code is nearly identical to the previous definition of the `TForm1::CheckBox1Click()` method. Here I've simply added code to change the cursor to `crSizeAll` when `MoveResize_` is `true`, and to restore the cursor (by using the `OldCursor_` member) when `MoveResize_` is `false`. There's no code for the sizing-related cursors because, remember, Windows handles this part automatically. **Figure C** depicts a `TScrollBoxEx` object whose `MoveResize` property is set to `true`.

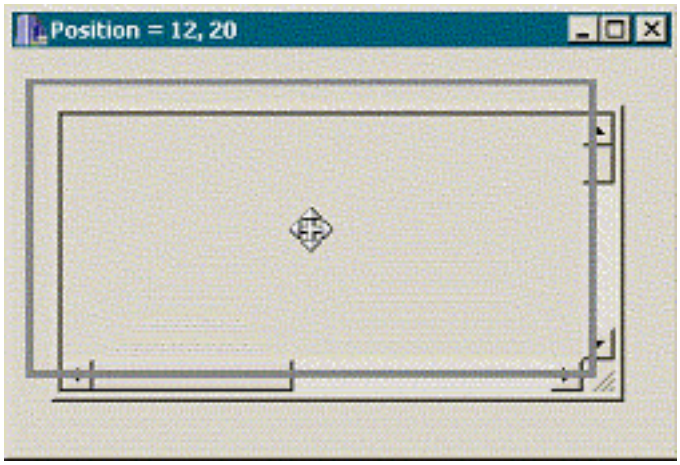


Figure C: *A movable and re-sizeable TScrollBox descendant class.*

Notice that **Listing A** contains two other protected methods: `CreateParams()` and `MouseDown()`. The `CreateParams()` method is used to restore the `WS_SIZEBOX` style in the event that the scroll box's underlying window is recreated. Here's the code for that function:

```
void __fastcall TScrollBoxEx::
  CreateParams(TCreateParams& Params)
{
  TScrollBox::CreateParams(Params);
  if (MoveResize_)
  {
    Params.Style |= WS_SIZEBOX;
  }
}
```

The MouseDown() method is used to invoke the move operation. Remember, to do this, you simply send the scroll box the WM_NCLBUTTONDOWN message, like so:

```
void __fastcall TScrollBoxEx::MouseDown(
  TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  TScrollBox::MouseDown(
    Button, Shift, X, Y);

  if (MoveResize_)
  {
    // release current mouse capture
    ReleaseCapture();

    // fake a caption hit
    SendMessage(Handle,
      WM_NCLBUTTONDOWN, HTCAPTION, 0);
  }
}
```

Handling the WM_SIZING and WM_MOVING messages

Now let's look at the WMSizing and WMMoving methods. As you can see from **Listing A**, these methods are called when the scroll box receives the WM_SIZING and WM_MOVING messages, respectively. The lParam value that accompanies these messages specifies a pointer to a RECT structure that describes the current position and dimensions (in screen coordinates) of the drag rectangle or of the window itself when the display properties are set to "Show window contents while dragging." You can use this rectangle to control the move/resize operation. In addition, for the WM_SIZING message, the wParam value specifies which edge or corner of the window is being dragged; this value will be one of the following: WMSZ_BOTTOM, WMSZ_LEFT, WMSZ_RIGHT, WMSZ_TOP, WMSZ_TOPLEFT, WMSZ_TOPRIGHT, WMSZ_BOTTOMLEFT, WMSZ_BOTTOMRIGHT.

For example, the following definition of the WMSizing() method demonstrates how to limit the maximum re-sizeable width:

```
void __fastcall TScrollBoxEx::WMSizing(
  TMessage& Msg)
{
  // grab a pointer to the drag rect
  RECT* pRect =
    reinterpret_cast<RECT*>(Msg.LParam);
```

```
  // limit the width to 400
  if (pRect->right - pRect->left > 400)
  {
    switch (Msg.WParam)
    {
      case WMSZ_LEFT:
      case WMSZ_TOPLEFT:
      case WMSZ_BOTTOMLEFT:
      {
        pRect->left = pRect->right-400;
        break;
      }
      case WMSZ_RIGHT:
      case WMSZ_TOPRIGHT:
      case WMSZ_BOTTOMRIGHT:
      {
        pRect->right = pRect->left+400;
        break;
      }
    }
  }

  // pass the message on
  TScrollBox::Dispatch(&Msg);
}
```

Similarly, the following definition of the `WMMoving()` method demonstrates how to display the current position of the drag rectangle (as depicted in **Figure C**):

```
void __fastcall TScrollBoxEx::WMMoving(
  TMessage& Msg)
{
  // grab a pointer to the drag rect
  const RECT* pRect =
    reinterpret_cast<const RECT*>(
      Msg.LParam);

  // translate to client coordinates
  const TPoint pos =
    Parent->ScreenToClient(
      Point(pRect->left, pRect->top));

  // display the current position
```

```
  Application->MainForm->Caption =
    "Position = " + IntToStr(pos.x) +
    ", " + IntToStr(pos.y);

  // pass the message on
  TScrollBox::Dispatch(&Msg);
}
```

Note: If you want to know when a move or resize operation has begun or ended, you can use the `WM_ENTERSIZEMOVE` or `WM_EXITSIZEMOVE` message, respectively.

Resizing without the funky border

As depicted in **Figures B** and **C**, when you use the `WS_SIZEBOX` style, the window displays a sizing border. However, some users might prefer the more common sizing grips, such as those depicted in **Figure D**.

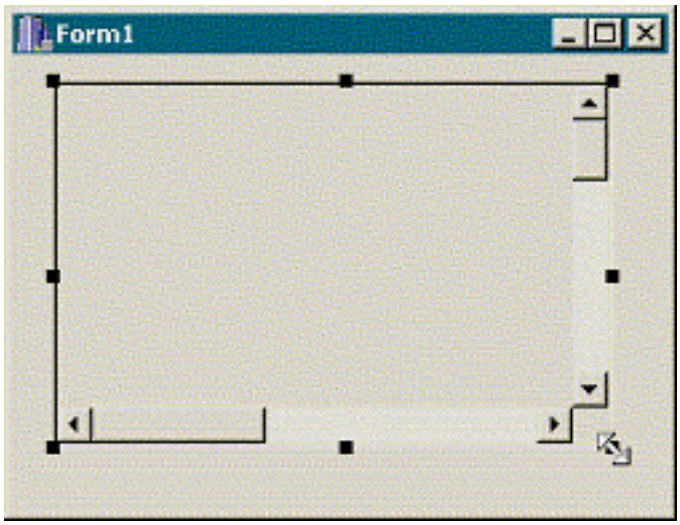


Figure D: *A movable and re-sizeable TScrollBox descendant class with sizing grips.*

Setting up the sizing grips

Each sizing grip in **Figure D** is actually just a black `TPanel` object; there are eight of these for the scroll box. **Listing B** contains a modified declaration of the `TScrollBoxEx` class that uses a `vector` to hold these eight panels (`Panels_`).

The first task is to create and initialize the panels, like so:

```
__fastcall TScrollBoxEx::TScrollBoxEx(
```

```
  TComponent* Owner) :
  TScrollBox(Owner), MoveResize_(false),
  OldCursor_(crDefault)
{
  for (int indx = 0; indx < 8; ++indx)
  {
    TPanel* Panel = new TPanel(this);
    Panel->Visible = false;
    Panel->BevelOuter = bvNone;
    Panel->Color = clBlack;
    Panel->OnMouseDown=PanelsMouseDown;
    Panel->Tag = indx;

    switch (indx)
    {
      case 0:
      case 4:
      {
        Panel->Cursor = crSizeNWSE;
        break;
      }
      case 1:
      case 5:
      {
        Panel->Cursor = crSizeNS;
        break;
      }
      case 2:
      case 6:
      {
        Panel->Cursor = crSizeNESW;
        break;
      }
      case 3:
      case 7:
      {
        Panel->Cursor = crSizeWE;
        break;
      }
    }
    Panels_.push_back(Panel);
  }
}
```

Note that as opposed to the case in which we used the `WM_SIZEBOX` style, this time I had to manually assign the correct sizing cursor to each panel. Also notice that all eight panels share the same `OnMouseDown` event handler (`PanelsMouseDown`). This event handler will later be used to invoke the resizing. Before I get to that code, however, let's first see how to position and display the panels; this is the job of the `DoShowGrips()` method:

```
void __fastcall
TScrollBoxEx::DoShowGrips(bool show)
{
  if (show)
  {
    const int grip_size = 6;
    const int half_grip_size =
      0.5 + 0.5 * grip_size;

    const TRect BR = BoundsRect;
    Panels_[0]->SetBounds(
      BR.Left - half_grip_size,
      BR.Top - half_grip_size,
      grip_size, grip_size
      );
    Panels_[1]->SetBounds(
      BR.Left +
        0.5 * Width - half_grip_size,
      BR.Top - half_grip_size,
      grip_size, grip_size
      );
    Panels_[2]->SetBounds(
      BR.Right - half_grip_size,
      BR.Top - half_grip_size,
      grip_size, grip_size
      );
    Panels_[3]->SetBounds(
      BR.Right - half_grip_size,
      BR.Top +
        0.5 * Height - half_grip_size,
      grip_size, grip_size
      );
    Panels_[4]->SetBounds(
      BR.Right - half_grip_size,
      BR.Bottom - half_grip_size,
      grip_size, grip_size
      );
```

```
    Panels_[5]->SetBounds(
      BR.Left +
        0.5 * Width - half_grip_size,
      BR.Bottom - half_grip_size,
      grip_size, grip_size
      );
    Panels_[6]->SetBounds(
      BR.Left - half_grip_size,
      BR.Bottom - half_grip_size,
      grip_size, grip_size
      );
    Panels_[7]->SetBounds(
      BR.Left - half_grip_size,
      BR.Top +
        0.5 * Height - half_grip_size,
      grip_size, grip_size
      );

    for (int indx = 0; indx < 8; ++indx)
    {
      Panels_[indx]->Parent = Parent;
      Panels_[indx]->Visible = true;
      Panels_[indx]->BringToFront();
    }
  }
  else
  {
    for (int indx = 0; indx < 8; ++indx)
    {
      Panels_[indx]->Visible = false;
    }
  }
}
```

The `DoShowGrips()` method will either show or hide the panels, depending on the value of the `show` parameter. Because these panels should be displayed when the `MoveResize` property is set to `true`, we can call this method from within the `DoSetMoveResize()` method, which is now defined like so:

```
void __fastcall TScrollBoxEx::
  DoSetMoveResize(bool value)
{
  if (MoveResize_ != value)
  {
```

```
    MoveResize_ = value;
    if (MoveResize_)
    {
      // change the cursor to crSizeAll
      OldCursor_ = Cursor;
      Cursor = crSizeAll;
    }
    else
    {
      // restore the cursor
      Cursor = OldCursor_;
    }
    // display the resizing grips
    DoShowGrips(value);
  }
}
```

Note that I removed the previous calls to `GetWindowLong()` and `SetWindowLong()` because the `WM_SIZEBOX` style is no longer needed.

We'll also want to hide the panels when the user begins a move/resize operation; and we'll want to reshow (and reposition) the panels after the move/resize operation is complete. As I mentioned earlier, you can use the `WM_ENTERSIZEMOVE` and `WM_EXITSIZEMOVE` messages for notification of these two events:

```
void __fastcall TScrollBoxEx::
  WMEnterSizeMove(TMessage& Msg)
{
  // hide the panels
  DoShowGrips(false);

  // pass the message on
  TScrollBox::Dispatch(&Msg);
}

void __fastcall TScrollBoxEx::
  WMExitSizeMove(TMessage& Msg)
{
  // reshow and reposition the panels
  DoShowGrips(true);

  // pass the message on
  TScrollBox::Dispatch(&Msg);
```

```
}
```

Handling the resizing

Because the `WM_SIZEBOX` style isn't used, you might be thinking that you'll have to implement the sizing-related code manually. As it turns out, you still can punt most of the work to Windows. The key is to use the `WM_NCLBUTTONDOWN` message just as we did earlier to initiate a move operation. This time, instead of sending the message with `HTCAPTION` as the `wParam` value, you set `wParam` to one of the following: `HTLEFT`, `HTTOP`, `HTRIGHT`, `HTBOTTOM`, `HTTOPLEFT`, `HTTOPRIGHT`, `HTBOTTOMLEFT`, `HTBOTTOMRIGHT`. The `WM_NCLBUTTONDOWN` message should be sent when any of the panels are clicked—i.e., from within the `PanelsMouseDown()` method:

```
void __fastcall
TScrollBoxEx::PanelsMouseDown(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  TComponent& Panel =
    static_cast<TComponent&>(*Sender);

  int code = HTNOWHERE;
  switch (Panel.Tag)
  {
    case 0: code = HTTOPLEFT; break;
    case 1: code = HTTOP; break;
    case 2: code = HTTOPRIGHT; break;
    case 3: code = HTRIGHT; break;
    case 4: code = HTBOTTOMRIGHT; break;
    case 5: code = HTBOTTOM; break;
    case 6: code = HTBOTTOMLEFT; break;
    case 7: code = HTLEFT; break;
  }

  // release current mouse capture
  ReleaseCapture();

  // fake a sizing border hit
  SendMessage(
    Handle, WM_NCLBUTTONDOWN, code, 0);
}
```

That's it for the resizing code. As promised, Windows handles most of the work.

Conclusion

I've demonstrated how you can add run time moving and resizing functionality to any `TWinControl` descendant. You might be wondering if a similar technique also be used for `TGraphicControl` descendants. Unfortunately, it cannot. Remember, a `TGraphicControl` doesn't have an underlying window, so you can't use the `SendMessage()` function. What you can do, though, is place the `TGraphicControl` on a `TWinControl`, and then make the `TWinControl` moveable and sizeable. For example, **Figure E** depicts a moveable and re-sizeable `TImage` object; in this case, the `TImage` is simply placed on a `TScrollBoxEx`. You can download the source code that accompanies this article from **www.bridgespublishing.com**.

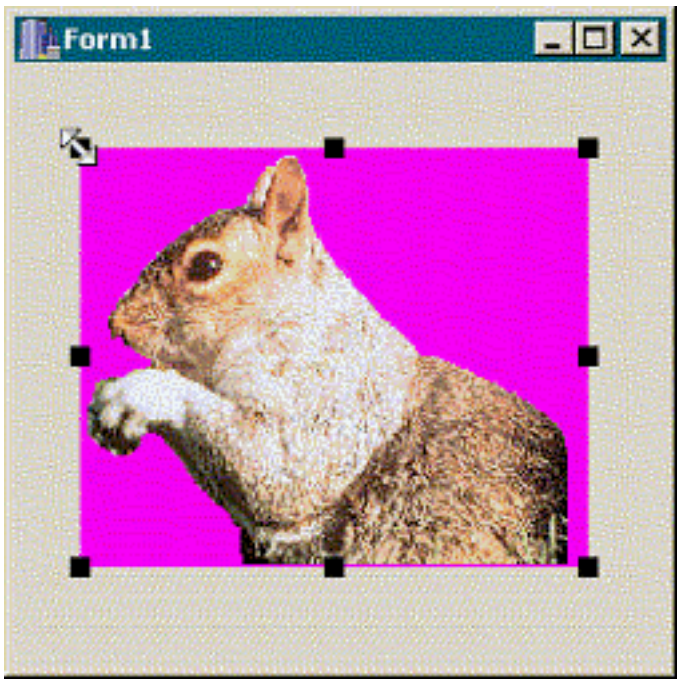


Figure E: *A movable and re-sizeable TImage.*

Listing A: ***The declaration of the TScrollBoxEx class***

```
#include <cassert>
class TScrollBoxEx : public TScrollBox
{
public:
  __fastcall TScrollBoxEx(TComponent* Owner) :
    TScrollBox(Owner), MoveResize_(false),
    OldCursor_(crDefault) {}

  __property bool MoveResize =
    {read = MoveResize_, write = DoSetMoveResize};
```

```
protected:
  // inherited methods
  virtual void __fastcall CreateParams(
    TCreateParams& Params);
  DYNAMIC void __fastcall MouseDown(
    TMouseButton Button,
    TShiftState Shift, int X, int Y);

  // introduced method
  virtual void __fastcall DoSetMoveResize(
    bool value);

private:
  bool MoveResize_;
  TCursor OldCursor_;

  MESSAGE void __fastcall WMSizing(TMessage& Msg);
  MESSAGE void __fastcall WMMoving(TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_SIZING, TMessage, WMSizing)
  MESSAGE_HANDLER(WM_MOVING, TMessage, WMMoving)
END_MESSAGE_MAP(TScrollBox)
};
```

Listing B: ***The declaration of the TScrollBoxEx class with the eight panels***

```
#include <vector>
class TScrollBoxEx : public TScrollBox
{
typedef std::vector<TPanel*> TPanels;
public:
  __fastcall TScrollBoxEx(TComponent* Owner);

  __property bool MoveResize =
    {read = MoveResize_, write = DoSetMoveResize};

protected:
  // inherited method
  DYNAMIC void __fastcall MouseDown(
    TMouseButton Button,
    TShiftState Shift, int X, int Y);
```

```
  // introduced method
  virtual void __fastcall DoSetMoveResize(
    bool value);
  virtual void __fastcall DoShowGrips(bool show);

private:
  bool MoveResize_;
  TCursor OldCursor_;
  TPanels Panels_;

  void __fastcall PanelsMouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);

  MESSAGE void __fastcall WMSizing(TMessage& Msg);
  MESSAGE void __fastcall WMMoving(TMessage& Msg);
  MESSAGE void __fastcall WMEnterSizeMove(
    TMessage& Msg);
  MESSAGE void __fastcall WMExitSizeMove(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_SIZING, TMessage, WMSizing)
  MESSAGE_HANDLER(WM_MOVING, TMessage, WMMoving)
  MESSAGE_HANDLER(WM_ENTERSIZEMOVE,
    TMessage, WMEnterSizeMove)
  MESSAGE_HANDLER(WM_EXITSIZEMOVE,
    TMessage, WMExitSizeMove)
END_MESSAGE_MAP(TScrollBox)
};
```

An enhanced type safe TList, part 2

By Cris Gallegos

In Part 1 of this article, I presented the basic code for the `TssTempList` class. This class wraps a VCL `TList` into a C++ template. The template functionality allows the C++Builder compiler to provide basic type checking and type casting services that makes the encapsulated `TList` easier, cleaner, and safer to use. The encapsulation of a `TList` also provides the framework for a VCL-style public interface that should be familiar to most C++Builder users.

In this article, I will extend the basic idea to allow for more intuitive sorting of the objects contained within the list. I will also show how the class can be used to instantiate objects automatically. I will then finish the article by showing how basic object information can easily be stored to, and read from, persistent storage.

This month's code for `TssTempList` is presented as part of **Listing A**. It can be added to the code given in the first half of the article. Better still, the complete source code, with comments, for all three of the units mentioned in this article can be found on the Bridges Publishing Web site. Some refinements have been made to `TssTempList`, `TssDiskList`, and `TssRegList` during the writing of this article. Please download the latest versions if they are of interest to you.

Object Sorting

The sorting of objects is really not difficult even when using a basic `TList`. The call to `TList::Sort()` must contain the address of a callback comparison function that has the `TlistSortCompare` signature. Once called, the `TList` performs a quick sort on the pointers contained in the list by repeatedly calling the comparison function. Each time the comparison function is called, two of the pointers contained within the list are passed into it. The function's return value indicates which pointer should come first in the list.

One of the problems with this mechanism is that `TList` can only provide `void` pointers to the comparison function. This means that the pointers must be cast into something useful before they can be used. The required casting can be made safer by using `TssTempList`. Even so, the programmer must still perform them explicitly in code. Another problem is that the `TList` sorting mechanism requires that the comparison function have a 32-bit address; it must either be a stand-alone function, or else a static member function of some class.

Once `TList` has been wrapped inside a template, the compiler can do the necessary typecasting as the sorting of object pointers is occurring. This allows the type casting to be resolved safely and effortlessly at compile time (remember, only one type of object pointer per template instance and the compiler tracks

the details).

What `TssTempList` does is act like a relay station in the sorting operation. Its version of `Sort()` doesn't require any parameters. It simply calls `TList::Sort()` and provides it with the static member function, `SortingFunc()`. This protected method has the `TlistSortCompare` signature. As the underlying `TList` performs its quick sort, `SortingFunc()` typecasts the `void` pointers passed into it automatically.

What `SortingFunc()` does then is call upon a user supplied comparison function. This sorting function can now be a normal member function; thereby allowing the use of non-static member data and/or methods. The comparison function doesn't even have to be a standard "function" as such. In the code for `TMeterCfg`, I chose to implement the sorting functionality via an overloaded sorting operator (the `^` operator). Note that the pointers in `SortingFunc()`, once cast, have to be de-referenced. This insures that the `TMeterCfg` sorting operator is called. De-referencing would not be necessary if a comparison function were used instead.

The point I'm trying to make is that `SortingFunc()` should have the flexibility to accommodate just about any sorting methodology a situation might require. I must inject one word of caution however: If `F_ALLOW_NULL` is `true`, the user supplied comparison function must be able to handle `NULL` pointers correctly.

One of the nice, although sometimes frustrating, things about templates is that if a property or method is implemented within the template but is not specifically referenced by other code (either by being called upon it or by having its address taken) then it is not instantiated by the compiler. This has been the default behavior of the C++Builder compiler since version 3. This means that if `TssTempList::Sort()` is never called, `SortingFunc()` can be left in the header file yet the compiler won't require that the objects implement a comparison function or operator.

Object creation

Last month I mentioned that the first parameter to the `TssTempList` constructor is used to initialize the protected `F_ALLOW_CREATE` constant. This flag, when `true`, allows the class to create object instances when called upon to do so. Objects can be created from within a second version of both the `Add()` and `Insert()` methods. Notice that the methods each have one less parameter than the comparable first versions do. Both can return a shiny new instance of `T`. If `F_ALLOW_CREATE` is `false`, the programmer is warned of the error. If a memory allocation problem occurs, the end user is warned. In either case, a `NULL` pointer is returned to the calling function.

Multiple objects can be instantiated or deleted in a single operation by assigning a value to the `Count` property. When assigned to do so, `Count` calls upon the `SetListCount()` method. If more objects are needed and `F_ALLOW_CREATE` is `true`, `SetListCount()` calls the

AllocateNewObjects() method. This allocates an array of T onto the heap. NULL pointers are then added to the underlying TList by assigning the change to its Count property. The new list pointers are then indexed into the array using a simple for loop. The pointer to the actual array then falls out of scope, leaving the new objects accessible via the pointers in the list. As unnatural as it may seem, delete[] should not be called upon the array.

If F_ALLOW_CREATE is false and F_ALLOW_NULL is true, the default behavior of TList is preserved and only NULL pointers are added to the underlying list. If the new count is less than the old count, pointers are deleted from the underlying list and the objects are individually deleted from memory (assuming that F_ALLOW_DELETE is true). Please note that if both F_ALLOW_CREATE and F_ALLOW_NULL are false, direct manipulation of the Count property is not allowed.

Before I continue on, I need to touch upon the subject of object constructors. By necessity, TssTempList can only create objects by utilizing their default (zero argument) constructors. This means that any objects tracked by TssTempList will require some form of default constructor if the second forms of Add() or Insert() or, more likely, the Count property (which instantiates SetListCount()) are referenced in any way. Objects with multiple constructors are not a problem as long as the object implements a default constructor; the compiler will not create one for you in this instance. F_ALLOW_CREATE can be set to false if object creation by the default constructor is not appropriate.

Object storage

If storage and retrieval of basic object data is required, TssDiskList or TssRegList can be used instead of TssTempList. Both inherit from TssTempList. TssDiskList reads and writes object information to and from a disk file. TssRegList does the same thing only it uses the system registry. The TssDiskList source file is presented as **Listing B**. The listing for TssRegList is similar so it will not be printed here in order to conserve space. As mentioned earlier, it is available on the Bridges Publishing Web site. TssRegList's operation will still be explained.

Both classes were designed to work reliably with objects whose underlying data is contained in fixed length fields (for example character arrays instead of AnsiString arrays). Both classes operate in a similar fashion. Their constructors require the same parameters as TssTempList (they are simply passed onto the base class for its initialization). Both support the reading and writing of information containing NULL pointers (assuming, of course, that F_ALLOW_NULL is set to true).

When reading in object information, both classes always create the correct number of actual objects needed in their ReadFromXXX() methods (by directly calling AllocateNewObjects()). NULL pointers are then added to the list as needed and in the correct locations. Note that a default object constructor will always be required regardless of the value of F_ALLOW_CREATE. Even a compiler-generated constructor should be sufficient as each new object is overwritten by the, hopefully valid, stored object information.

Both `ReadFromXXX()` methods provide a quick and dirty validity check of the information being read by testing the size of the current `T` against the size of `T` written to the file. Both require that a location be specified before any reading or writing of information can occur.

`TssDiskList::WriteToFile()` tags each of the object records that it writes as either a valid object, or a `NULL` pointer. `TssDiskList::ReadFromFile()` uses these tags to insure that the proper operations are performed during a disk read. The last 96 bits of the disk file are used to store the size of the objects being stored, the total number of records and the number of valid object records, respectively. `SetFileName()` provides the opportunity to verify filenames, drives, and so on. If the file being opened by `ReadFromFile()` does not exist, it throws an exception and then returns `false`.

`TssRegList's` constructor initializes three extra string constants. These constants are used to write data to the registry in a consistent manner. If a value is not assigned to the `RootKey` property, `ReadFromRegistry()` and `WriteToRegistry()` default to using the `HKEY_CURRENT_USER` key. When writing to the registry, Values are created that store the size of the objects being stored, the total number of records, and the number of valid object records.

Information is read from and written to the registry in the form of binary data. Theoretically, this allows any size of objects to be stored and retrieved, but the limitations of the registry itself should be kept in mind. If large items (such as graphical objects) need to be stored, system performance dictates the use of `TssDiskList` instead of `TssRegList`.

The following code snippet shows how trivial it can be using `TssDiskList` to manipulate user-configurable data. Please see **Listing A** for the definition of the `TMeterCfg` class.

```
//At program start-up.
TssDiskList<TMeterCfg*> * MeterList
  = new TssDiskList<TMeterCfg*>(
    true, true, false);
//Allows object creation/deletion.
// No NULL objects.
MeterList->FileName = "MeterCfg.dat";
If(MeterList->ReadFromFile())
{
  //Take the necessary steps.
}

//Saving user modified information.
MeterList->Items[9]->MeterName =
  Edit9->Text;
```

```
MeterList->WriteToFile();
```

Conclusion

I do want to mention a few things before closing. I have chosen to implement an `Extract()` method. It is modeled after like methods found in the C++Builder 5 versions of `TList`, `TComponentList`, and `TObjectList` (C++Builder 5 only). `Extract()` will remove a pointer from the list, but will not delete the object from memory, even if `F_ALLOW_DELETE` is `true`. Incorrect use of `Extract()` is guaranteed to leak memory. So far I have run across only one situation where this option was useful, but in that instance it was very useful. I leave it to you to weigh the benefit against the risk.

I also need to mention that I am using `std:bad_alloc` to trap resource allocation failures resulting from calls to new. The `std::bad_alloc` exception does not seem to be thrown reliably in all conditions, under all flavors of Windows, and on all possible system configurations. If you have any information relating to this problem, or have any other questions or tips concerning the code presented, please contact me directly.

In closing, I would like to thank all of the people that helped with the creation of this article (there were a few). I hope you have enjoyed seeing your ideas and tips in print. I would also like to thank the people who create and maintain those wonderful C++Builder Web sites. From all of us who have learned so much from their content, I thank you.

Listing A: *The declaration of the TssTempList class*

```
#ifndef TempListH
#define TempListH

#include <new.h>//Needed to catch std::bad_alloc.
#include <stdio.h>//Needed for strncpy().

class TMeterCfg
{
  private:
    char FMeterID[SITE_NAME_LENGTH + 1];

    String GetMeterID()
    {
      return ::AnsiString(FMeterID);
    }

    void SetMeterID(String NewVal)
    {
```

```
      strncpy(FMeterID, NewVal.c_str(),
        SITE_NAME_LENGTH + 1);
    }

  public:
    TMeterCfg()
    {
      ::ZeroMemory(
        &FMeterID, SITE_NAME_LENGTH + 1);
    }
    __property String MeterName =
      {read=GetMeterID, write=SetMeterID};

   //The sorting operator.
   // Used by TssTemplateList->Sort().
   int operator ^(const TMeterCfg& rhs)const
   {
     return MeterName.AnsiCompare(rhs.MeterName);
   }
};

template <class T>
  class TssTempList< T * > : TObject
{
  protected:
    void AllocateNewObjects(int NewCount);

    void SetListCount(int NewCount);

    static int __fastcall SortingFunc(
      void * Item1, void * Item2)
    {
      return *reinterpret_cast< T * >(
        Item1) ^ *reinterpret_cast< T * >(Item2);
    }

  public:
    inline T * Add();//Second form.

    void Extract(int Index)
    {
      FList->Delete(Index);
    }
```

```
    inline T * Insert(int Index);//Second form.

    void Sort()
    {
      FList->Sort(SortingFunc);
    }
};

template <class T> void TssTempList< T * >
  ::AllocateNewObjects(int NewCount)
{
  try
  {
    int OldCount = FList->Count;
    //Allocate objects
    T * ArrayOfT = new T[NewCount - OldCount];
    //Allocate space in the list.
    FList->Count = NewCount;

    //OldCount now indexes first
    // new pointer in FList.

    //Add objects to list.
    for(int i = OldCount; i < FList->Count; i++)
      FList->Items[i] = &ArrayOfT[i - OldCount];
  }
  catch(std::bad_alloc &E)
  {
    //Warn the end user.
  }
}

//Second form.
template <class T> T * TssTempList< T * >::Add()
{
  T * pT = NULL;
  try
  {
    if(F_ALLOW_CREATE)
    {
      pT = new T;
      FList->Add(pT);
    }
    else
```

```
    {
      //Warn the programmer.
    }
  }
  catch(std::bad_alloc &e)
  {
    //Warn the end user.
    pT = NULL;//Ensure return value.
  }
  return pT;
}

//Second form.
template <class T> T * TssTempList< T * >::Insert(
  int Index)
{
  T * pT = NULL;
  try
  {
    if(F_ALLOW_CREATE)
    {
      pT = new T;
      FList->Insert(Index, pT);
    }
    else
    {
      //Warn the programmer.
    }
  }
  catch(std::bad_alloc &e)
  {
    //Warn the end user.
    pT = NULL;//Ensure the return value.
  }
  return pT;
}

template <class T> void TssTempList< T * >::
  SetListCount(int NewCount)
{
  if(!(F_ALLOW_CREATE || F_ALLOW_NULL))
  {
    //Warn programmer. Direct manipulation of
```

```
    //Count not allowed.
    return;
  }
  if(NewCount < 0 ||
    ((Count + NewCount) > MaxListSize))
    //MaxListSize is defined in Classes.pas.
  {
    //Warn the programmer.
    return;
  }
  if(NewCount > FList->Count)
  {
    if(F_ALLOW_CREATE)
    {
      AllocateNewObjects(NewCount - Count);
    }
    else
      FList->Count = NewCount;
  }
  else if(NewCount < FList->Count)
  {
    while(NewCount < FList->Count)
      Delete(NewCount);
  }
}
#endif
```

Listing B: *The declaration of the TssDiskList class*

```
#ifndef DiskListH
#define DiskListH
#include "TempList.h"//Contains TssTempList.

//First version. See first half of article.
template <class T> class TssDiskList
  : System::TObject
{
private:
  TssDiskList();//Constructor not defined.
  //Copy Constructor not defined.
  TssDiskList(const TssDiskList&);
};
```

```
//Second version. See first half of article.
template <class T> class TssDiskList< T * >
  : public TssTempList< T * >
{
  private:
    String FFileName;

    void SetFileName(const String NewValue);

  public:
    TssDiskList(bool AllowCreation,
      bool AllowDeletion, bool AllowNULL) :
      TssTempList< T * >(
        AllowCreation, AllowDeletion, AllowNULL)
    {
    }

  //No need for destructor.

  __property String FileName =
    {read=FFileName, write=SetFileName};

  bool ReadFromFile();
  bool WriteToFile();
};

template <class T> void TssDiskList< T * >::
  SetFileName(const String NewNameStr)
{
  try
  {
    if(NewNameStr.IsEmpty())
      throw EInvalidOperation("Empty String. ”);

    //Test NewNameStr validity. Throw
    //InvalidOps as needed.
    FFileName = NewNameStr;
  }
  catch(EInvalidOperation &E)
  {
    //Warn the programmer.
  }
}
```

```
template <class T> bool TssDiskList< T * >::
  ReadFromFile()
{
  bool Result = false, NULLRecordFlag;
  int ObjectSize, TotalCount, RecordCount;
  TFileStream * pFS = NULL;
  try
  {
    if(FFileName.IsEmpty())
      throw EInvalidOperation(
        “Warn the programmer.”);
    pFS = new TFileStream(FileName,
      fmOpenRead | fmShareExclusive);
    pFS->Position = pFS->Size - (3 * sizeof(int));
    pFS->Read(&ObjectSize, sizeof(int));
    pFS->Read(&TotalCount, sizeof(int));
    pFS->Read(&RecordCount, sizeof(int));

    //Do error checking.
    if(TotalCount < 0 || TotalCount > MaxListSize
      || RecordCount < TotalCount
      || RecordCount > TotalCount)
    //MaxListSize is defined in Classes.pas.
      throw EInvalidOperation(
      "Warn the programmer");
    if(ObjectSize != sizeof(T))
      throw EInvalidOperation("Warn who ever.");
    if(!F_ALLOW_NULL && (TotalCount >RecordCount))
      throw EInvalidOperation(
        "Warn the programmer.");

    //Continue on.
    pFS->Position = 0;//Reset.
    Clear();//Empty list.
    AllocateNewObjects(RecordCount);
    //Allocate correct number of actual objects.
    for(int i = 0; i < TotalCount; i++)
    {
      pFS->Read(&NULLRecordFlag, sizeof(bool));
      if(NULLRecordFlag)
        Insert(i, NULL);//Insert NULLs as needed
      else
        pFS->Read(FList->Items[i], sizeof(T));
```

```
    }
  Result = true;
  }
  catch(EInvalidOperation &E)
  {
    ShowMessage(E.Message);
  }
  catch(...){}//Trap filestream problems.
  if(pFS != NULL)
    delete pFS;
  return Result;
}

template <class T> bool TssDiskList< T * >::
  WriteToFile()
{
  bool Result = false;
  TFileStream * pFS = NULL;
  if(FFileName.IsEmpty())
  {
    Beep(1000, 1000);
    ShowMessage("FileName is empty.");
  }
  else
  {
    const int ObjectSize = sizeof(T);
    const int TotalRecords = FList->Count;
    int ValidRecords = 0;
    T * pT = NULL;
    bool NULLRecordFlag;
    pFS = new TFileStream(
      FFileName, fmCreate | fmShareExclusive);

    try
    {
      for(int i = 0; i < FList->Count; i++)
      {
        pT = Get(i);
        if(pT == NULL)
        {
          NULLRecordFlag = true;
          pFS->Write(
            &NULLRecordFlag, sizeof(bool));
```

```
        }
        else
        {
          NULLRecordFlag = false;
          pFS->Write(
            &NULLRecordFlag, sizeof(bool));
          pFS->Write(pT, sizeof(T));
          ++ValidRecords;
        }
      }
      pFS->Write(&ObjectSize, sizeof(int));
      pFS->Write(&TotalRecords, sizeof(int));
      pFS->Write(&ValidRecords, sizeof(int));
      Result = true;
    }
    catch(...)//Catch problems with file stream.
    {
      if(pFS != NULL)
      {
        ValidRecords = 0;//Set to safe value.
        pFS->Write(&ValidRecords, sizeof(int));
        pFS->Write(&ValidRecords, sizeof(int));
        pFS->Write(&ValidRecords, sizeof(int));
      }
    }
  }
  if(pFS != NULL)
    delete pFS;
  return Result;
}
#endif
```

Smart list views

By David Bridges

The list view common control has been around since Windows 95. It was created as a successor to the list box control, and offers numerous enhancements. Among these is the ability to display data in resizable columns, sort the items, and display icons. The VCL `TListView` class provides a powerful, easy to use interface for list view controls. However, it still takes a fair amount of code to make a list view control do really useful things. If you have many different list view controls in your application, writing code to display data in them turns into a large and repetitive task.

In this article, I will introduce the `ListViewManager` class, which takes care of most of the dirty work in implementing a useful list view control. **Listing A** shows this class. `ListViewManager` is a template class that provides a high-level interface to the list view control. Specifically, it does the following:

- Add items to the listview and display their data in columns
- Sort the data by column when a column header is clicked
- Display the icon for an item.

The item data class

In order to use the `ListViewManager` class, you must first create an "item data" class. This class contains the data to be displayed as an item in the list view control. It can be implemented any way you like, as long as it contains two functions: `GetColumnString()`, which returns the data for a particular column, and `GetImageIndex()`, which returns the item's image index.

As an example, here is an item data class called `Contact` which holds names and addresses:

```
class Contact
{
  protected:
  AnsiString sName;
  AnsiString sContact;
  AnsiString sAddress;
  AnsiString sState;
  AnsiString sZip;

  public:

  // Constructor
  Contact(AnsiString name,
    AnsiString contact, AnsiString address,
    AnsiString state, AnsiString zip);

  AnsiString GetColumnString(int iColumn);
  int GetImageIndex();
};
```

```
#define ICON_PERSON  0
#define ICON_CONTACT 1

Contact::Contact(AnsiString name,
  AnsiString contact, AnsiString address,
  AnsiString state, AnsiString zip)
{
  sName = name;
  sContact = contact;
  sAddress = address;
  sState = state;
  sZip = zip;
}

// This function is called to get the
// string data to be displayed in the
// listview columns.
AnsiString Contact::GetColumnString(
  int iColumn)
{
  switch (iColumn) {
    case 0: return sName;
    case 1: return sContact;
    case 2: return sAddress;
    case 3: return sState;
    case 4: return sZip;
  }
  return "";
}

// This function called to get the icon
int Contact::GetImageIndex()
{
  if (sContact.Length() > 0) {
    return ICON_CONTACT;
  }
  return ICON_PERSON;
}
```

Notice that in this class there is no code having anything to do with the list view control—it is only concerned with storing the data and returning information about it.

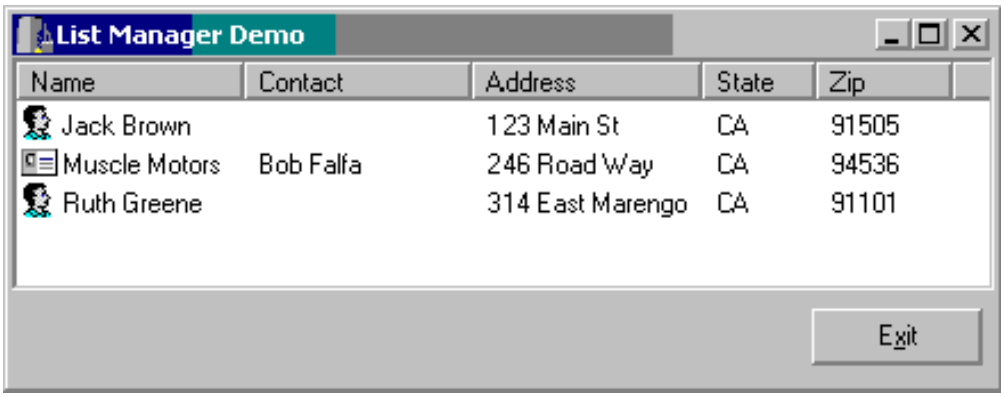
Now that the item data class is defined, all it takes to start adding items to the list view control is the following code:

```
ListViewManager<Contact> *mgr =
  new ListViewManager<Contact>(
    lvContacts);
mgr->AddObject(new Contact("Jack Brown",
  "", "123 Main St", "CA", "91505"));
mgr->AddObject(new Contact(
  "Muscle Motors","Bob Falfa",
```

```
  "246 Road Way", "CA", "94536"));
mgr->AddObject(new Contact("Ruth Greene",
  "", "314 East Marengo", "CA", "91101"));
```

Figure A shows a list view using the `vsReport` view style.

Figure A: *A list view in vsReport style*



The data is displayed in its appropriate column, the icons are shown, and the data will re-sort when the header for a column is clicked. The following sections explain how this is done.

Displaying the data

First I will explain how to display data in columns using only the `TListView` properties. To display data in the first column, the `Caption` property is used. To set data in additional columns, the `SubItems` property is used. `SubItems` is a `TStrings` object. For example, the second column displays the `SubItems->Strings[0]` property of each item, and so on.

The `ListViewManager` class abstracts all of this and treats everything as just columns of strings. The item data class only needs to return the string values for each column, via the `GetColumnString()` function. In order for the data to show up in the "correct" columns, the listview must be created (or set programmatically, at run time) with the correct number of columns and with the proper headings.

Displaying the icons

In order to display icons in a `TListView`, it must be associated with a `TImageList` object. Setting the ListView control's `SmallImages` property to the name of the `ImageList` makes this association. The icon shown for each `TListItem` in the list view control depends on its `ImageIndex` property.

The `ListViewMananger` class calls the `GetImageIndex()` function of the item data class to get its image index. This means that the item data class must have knowledge of the bitmaps used in the `ImageList` object. This is a somewhat unavoidable problem, and the solution in this example is to define the constants `ICON_PERSON` and `ICON_CONTACT` to represent the image indexes.

Sorting the data

To implement sorting, the `ListViewManager` class uses the `AnsiString` method's `AnsiCompareIC()` function to order the items based on a specified column. When the `ListViewManager` is created, it assigns its own event handlers to the list view's `OnColumnClick` and `OnCompare` events. In case the form has it's own handlers attached to these events, the `ListViewManager` class stores the previous handler assigned to these events and calls them after performing its own handlers.

Cleaning up

The `ListViewManager` class also acts as a container for the item data objects that it holds. When the ListView is deleted, all of the item data objects are also deleted. This is done via the list view control's `OnDeletion` event.

Conclusion

By using the `ListViewManager` class, you can save time implementing the display and sorting logic for list view controls. The item data class is only responsible for holding application data and returning information about it. In addition, the item data class can be used with other common controls as well (such as `TtreeView`) by writing a "manager" class for that type of control.

Listing A: *The ListViewManager template class*

```
template <class T> class ListViewManager :
  public TComponent
{
  protected:
  TLVDeletedEvent     OldOnDelete;
  TLVCompareEvent     OldOnCompare;
  TLVColumnClickEvent OldOnColumnClick;
  int iCurrentSort;

  public:

  TListView* lvList;

  __fastcall ListViewManager(TListView* pList);

  TListItem* AddObject(T* pObj);
  void RefreshObject(T* pObj);
  void RefreshItem(TListItem* item);

  void __fastcall OnListViewDeletion(
    TObject *Sender, TListItem *Item);

  void __fastcall OnListViewColumnClick(
    TObject *Sender, TListColumn *Column);

  void __fastcall OnListViewCompare(
    TObject *Sender, TListItem *Item1,
    TListItem *Item2, int Data, int &Compare);
};
```

```
template <class T>
  __fastcall ListViewManager<T>::ListViewManager(
  TListView* pList) : TComponent(pList)
{
  lvList = pList;
  iCurrentSort = -1;

  // Save previous event handlers defined in DFM
  OldOnDelete = lvList->OnDeletion;
  OldOnCompare = lvList->OnCompare;
  OldOnColumnClick = lvList->OnColumnClick;

  // Assign new event handlers
  lvList->OnDeletion    = OnListViewDeletion;
  lvList->OnCompare     = OnListViewCompare;
  lvList->OnColumnClick = OnListViewColumnClick;
}

// This function updates the column text
// of the specified object
template<class T>
  void ListViewManager<T>::RefreshObject(T* pObj)
{
  TListItem* pItem = FindObject(pObj);
  if (pItem) {
    RefreshItem(pItem);
  }
}

// Function RefreshItem() updates the columns of
// text and the icon for an object in the listview
template<class T>
void ListViewManager<T>::RefreshItem(
  TListItem* item)
{
  if (item) {
    T* pObject = (T*)(item->Data);
    if (pObject) {
      item->SubItems->Clear();
      for (int i=0;i<lvList->Columns->Count;i++) {
        if (i == 0) {
          item->Caption =
            pObject->GetColumnString(i);
        }
        else {
          item->SubItems->Add(
            pObject->GetColumnString(i));
        }
      }
      item->ImageIndex = pObject->GetImageIndex();
    }
  }
```

```
}

// AddItem() adds an object to the list view and automatically sets the text of all
the columns.
template <class T>
TListItem* ListViewManager<T>::AddObject(
  T* pObject)
{
  TListItem* item = lvList->Items->Add();
  item->Data = (TObject*)pObject;
  RefreshItem(item);
  return item;
}

// This event ensures that the objects will get
// deleted when the listview is deleted.
template <class T>
void __fastcall
  ListViewManager<T>::OnListViewDeletion(
    TObject *Sender, TListItem *Item)
{
  T* pObject = (T*)(Item->Data);
  if (pObject) {
    delete pObject;
    Item->Data = NULL;
  }
}

// This event handles automatic sorting.
template <class T>
void __fastcall ListViewManager<T>::OnListViewColumnClick(
  TObject *Sender, TListColumn *Column)
{
  int iColumn = 0;
  for (int i=0;i<lvList->Columns->Count;i++) {
    if (lvList->Columns->Items[i] == Column) {
      iColumn = i;
      break;
    }
  }

  lvList->CustomSort(NULL, iColumn);

  if (iCurrentSort == iColumn) {
    iCurrentSort = -1;
  }
  else {
    iCurrentSort = iColumn;
  }

  if (OldOnColumnClick) {
    OldOnColumnClick(Sender, Column);
```

```
  }
}

// This event sorts columns via string comparison.
template <class T>
void __fastcall ListViewManager<T>::
  OnListViewCompare(
    TObject *Sender, TListItem *Item1,
    TListItem *Item2, int Data, int &Compare)
{
  String sValue1;
  String sValue2;
  if (Data == 0) {
    sValue1 = Item1->Caption;
    sValue2 = Item2->Caption;
  }
  else {
    sValue1 = Item1->SubItems->Strings[Data - 1];
    sValue2 = Item2->SubItems->Strings[Data - 1];
  }

  Compare = sValue1.AnsiCompareIC(sValue2);

  if (iCurrentSort == Data) {
    Compare = -Compare;
  }
}
```

Using custom tooltips

by Damon Chandler

A tooltip control is a small popup window that, as its name suggests, allows you to provide explanatory text (i.e., a tip) about your application's tools. Tooltips are supported in C++Builder via the `THintWindow` class. In a VCL application, the `TApplication` object creates and maintains a `THintWindow` object, which it uses for all controls whose `ShowHint` property is `true`.

In this article I'll show you how to create and use a `THintWindow` object to display your own tooltips (as shown in **Figure A**). I'll also explain how to create a `THintWindow` descendant class, and how to use this class to replace the default tooltips that are used by your application.

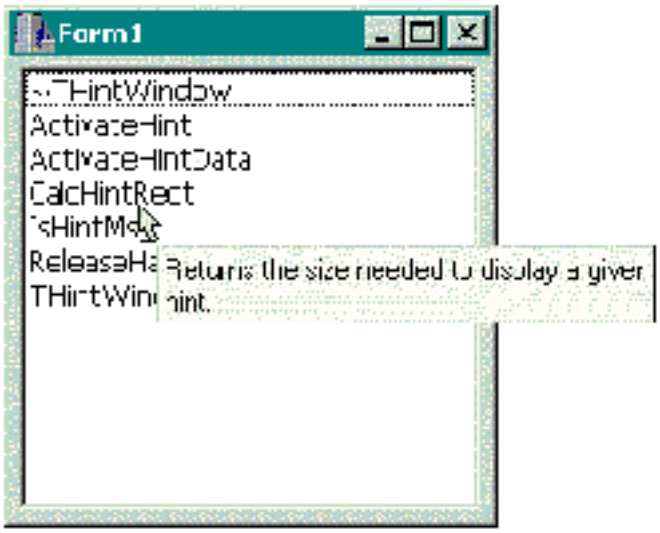


Figure A: *A list box with per-item tooltips.*

The THintWindow class

The `THintWindow` class is a direct descendant of the `TCustomControl` class. `TCustomControl` is simply an extended `TWinControl` that provides a `Canvas` property and an associated `Paint()` method. Because the `THintWindow` class descends from the `TWinControl` class, all `THintWindow` objects contain an underlying window handle. This is actually an important requirement because tooltips can appear beyond the bounds of a specific form. For example, in **Figure A** you can see that the tooltip's right edge indeed exceeds the form's right edge. As I mentioned last month, there are actually two ways to accomplish this effect. One way is to use a combination of the `WS_CHILD` and `WS_EX_TOOLWINDOW` styles and then make the popup window a child of the desktop window. The other method—which is useful only when the popup window doesn't have to capture the mouse—is to use a combination of the `WS_POPUP` and `WS_EX_TOOLWINDOW` styles. The `THintWindow` class

takes this latter approach, which you can see from the following C++ translation of the `THintWindow::CreateParams()` method:

```
void __fastcall THintWindow::
  CreateParams(TCreateParams& Prms)
{
  TCustomControl::CreateParams(Prms);

  Prms.Style = WS_POPUP | WS_BORDER;
  Prms.WindowClass.style |= CS_SAVEBITS;
  if (NewStyleControls)
    Prms.ExStyle = WS_EX_TOOLWINDOW;
  // ...
}
```

Displaying a tooltip

What might not be obvious from looking at **Figure A** is that the form doesn't lose activation when the tooltip is displayed. This is accomplished by specifying the `SWP_NOACTIVATE` flag in the call to the `SetWindowPos()` API function, which the `THintWindow` class makes from within its `ActivateHint()` method. As you might have guessed, the `ActivateHint()` method is used to display a tooltip. Here's the declaration of `ActivateHint()`:

```
virtual void __fastcall ActivateHint(
  const TRect &Rect,
  const AnsiString AHint);
```

The `Rect` parameter specifies where you want the tooltip to appear (relative to the screen's top-left corner), and its size. The `AHint` parameter specifies the caption that the tooltip should display.

As it turns out, the `ActivateHint()` method is rarely used alone. In most cases, a call to `ActivateHint()` is preceded by a call to the `THintWindow::CalcHintRect()` method. This latter method is used to calculate the width and the height of the tooltip for a given font and caption. In other words, `CalcHintRect()` is used to determine the width and the height of the rectangle that's passed as `ActivateHint()`'s `Rect` parameter. Here is that declaration:

```
virtual TRect __fastcall CalcHintRect(
  int MaxWidth, const AnsiString AHint,
  void* AData);
```

The `MaxWidth` parameter specifies the width (in logical units) beyond which the tooltip's caption is wrapped to the next line. The caption is specified by the `AHint` parameter. If you want to pass user-

defined data to the `THintWindow` object, you can use the `AData` parameter.

Within a VCL application, the `TApplication` object calls the `CalcHintRect()` and `ActivateHint()` methods from within its own `ActivateHint()` method—i.e., from within the `TApplication::ActivateHint()` method. I won't discuss this latter method because it's not accessible in all versions of C++Builder.

Hiding a tooltip

As with any window, there are several ways to hide a tooltip. Unfortunately, because the `ActivateHint()` method doesn't update the tooltip's `Visible` property, the `Hide()` method won't work. Instead, you can use the `THintWindow::ReleaseHandle()` method, which will hide the tooltip by destroying its underlying window. A more efficient approach, however, is to simply call the `ShowWindow()` API function, specifying the `SW_HIDE` flag. This will hide the tooltip without actually destroying its window. In fact, the `TApplication` class takes this latter approach from within its `HideHint()` method.

Unlike the `TApplication::ActivateHint()` method, the `HideHint()` method is accessible in all versions of C++Builder. Unfortunately, `HideHint()` isn't useful for hiding custom tooltips unless you've previously called the `TApplication::ActivateHint()` method to display the tooltip.

Using THintWindow

In this section I'll provide an example of using the `THintWindow` class by implementing a per-item tooltip for a list box as depicted in **Figure A**. The header for this example is provided in **Listing A**.

Notice from **Listing A** that there are four private members in the `TForm1` class: `last_index_`, `tic_`, `HintWindow_`, and `HintList_`. I'll explain the `last_index_` member shortly. The `tic_` member is used with a `TTimer` object (`Timer1`), which allows the hint window to be hidden after a specified delay. The `HintWindow_` member is the `THintWindow` object, and `HintList_` is simply a `TStrings` object that will hold the list of tips (one for each item in the list box). The members are initialized in the form's constructor like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner),
  last_index_(-1), tic_(0),
  HintList_(new TStringList())
{
  HintWindow_ = new THintWindow(this);
  HintWindow_->Brush->Color = clInfoBk;
```

```
  HintList_->Add( /* ~THintWindow */
    "Frees the memory associated with "
    "the THintWindow object.");
  HintList_->Add( /* ActivateHint */
    "Displays the hint window.");
  HintList_->Add( /* ActivateHintData */
    "Displays the hint window using a "
    "provided data value.");
  HintList_->Add( /* CalcHintRect */
    "Returns the size needed to display"
    " a given hint.");
  HintList_->Add( /* IsHintMsg */
    "Determines whether an application "
    "message requires hiding the hint "
    "window.");
  HintList_->Add( /* ReleaseHandle */
    "Destroys the window displayed by "
    "THintWindow.");
  HintList_->Add( /* THintWindow */
    "Creates and initializes an "
    "instance of THintWindow.");
}
```

Note that although I manually typed in the text for each tip in this example, you'll likely want to use a more maintainable approach in practice. For example, you could load the tips from a separate text file, a string table resource, or even a database table.

Showing and hiding tooltips

Next I'll implement a few methods to show and hide the tooltips. Remember, to display a tooltip, you use the `THintWindow::ActivateHint()` method. This method is called from within the `TForm1::DoActivateHint()` method:

```
void __fastcall TForm1::DoActivateHint(
  const TPoint& P, AnsiString TipText)
{
  // compute the tooltip's rect
  const int max_width = 200;
  TRect HintRect =
    HintWindow_->CalcHintRect(
      max_width, TipText, NULL
      );
```

```
  // position the tooltip's rect
  OffsetRect(
    static_cast<PRECT>(&HintRect),
    P.x + 10, P.y + 10
    );

  // display the tooltip
  HintWindow_->ActivateHint(
    HintRect, TipText
    );

  // reset the time-out counter
  tic_ = 0;

  // enable the time-out timer
  Timer1->Enabled = true;
}
```

To hide a tooltip, use the `ShowWindow()` API function. This function is called from within the `TForm1::DoHideHint()` method:

```
void __fastcall TForm1::DoHideHint()
{
  const HWND hHint =HintWindow_->Handle;

  // is the tooltip displayed?
  const bool showing =
    IsWindowVisible(hHint);

  if (showing)
  {
    // hide the tooltip
    ShowWindow(hHint, SW_HIDE);

    // disable the time-out timer
    Timer1->Enabled = false;
  }
}
```

Handling mouse messages

Although you now have methods to easily show and hide tooltips, you still need to know *when* to call

these functions. Well, because you want to show a tooltip for each of the list box's items, you can use the list box's `OnMouseMove` event to: (1) determine which item, if any, the mouse cursor is over, and then use that information to decide which tip to display; and (2) determine the coordinates at which to display the tooltip. Here's the code for that:

```
void __fastcall TForm1::
  ListBox1MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
  // find the item that's hit, if any
  TPoint PMouse = Point(X, Y);
  const int current_index =
    ListBox1->ItemAtPos(PMouse, true);

  // if an item is and it's not the
  // last item that was hit...
  if (current_index != last_index_ &&
      current_index != -1)
  {
    // display the tooltip
    DoActivateHint(
      ListBox1->ClientToScreen(PMouse),
      HintList_->Strings[current_index]
      );
  }
  else if (current_index == -1)
  {
    // dismiss the tooltip
    DoHideHint();
  }

  // update the last index
  last_index_ = current_index;
}
```

Notice that I used the private `last_index_` member to prevent the same tooltip from displaying multiple times when the mouse cursor moves over the current (same) item. Also notice that the `DoHideHint()` method is called when no item is hit (i.e., `current_item == -1`). You will also want to call this method when the user clicks a mouse button. For this latter task, use the list box's `OnMouseDown` event, like so:

```
void __fastcall TForm1::
  ListBox1MouseDown(TObject *Sender,
```

```
    TMouseButton Button,
    TShiftState Shift, int X, int Y)
{
  // dismiss the tooltip
  DoHideHint();
}
```

Implementing a time-out period

For most situations, you'll want to dismiss your tooltip after a given amount of time as passed. For this example, I used a `TTimer` object (and the `tic_` member). Here's the code for `Timer1`'s `OnTimer` event handler:

```
void __fastcall TForm1::
  Timer1Timer(TObject *Sender)
{
  // time-out counting...
  if (++tic_ > MAXTIC)
  {
    DoHideHint();
  }
  // cursor position checking...
  else
  {
    // extract position of the cursor
    POINT PMouse = {0};
    GetCursorPos(&PMouse);

    // grab the screen-relative position
    // of the target control
    RECT RListBox = {0};
    GetWindowRect(
      ListBox1->Handle, &RListBox
      );

    // if the mouse cursor is beyond
    // the bounds of the list box...
    if (!PtInRect(&RListBox, PMouse))
    {
      // clear the last index
      last_index_ = -1;

      // dismiss the tooltip
```

```
      DoHideHint();
    }
  }
}
```

Notice that the `Timer1Timer`() method actually serves two purposes. First, it hides the tooltip after a specified interval; this interval depends on `MAXTIC` and the value of `Timer1`'s `Interval` property. Second, the function queries the current position of the mouse cursor and then hides the tooltip if the cursor goes beyond the bounds of the list box. (You can also use the `CM_MOUSELEAVE` message for this purpose, but it's a good idea to back this message up with a timer.)

Extending THintWindow

In addition to using the `THintWindow` class as-is, you can also create a `THintWindow` descendant class and then use this class as a separate object (as described in the previous example). For instance, suppose that you have a tree view that contains items that correspond to different animals. As depicted in **Figure B**, you can use a `THintWindow` descendant class to display a tooltip for each node—specifically, a tooltip that contains an image of the corresponding animal.

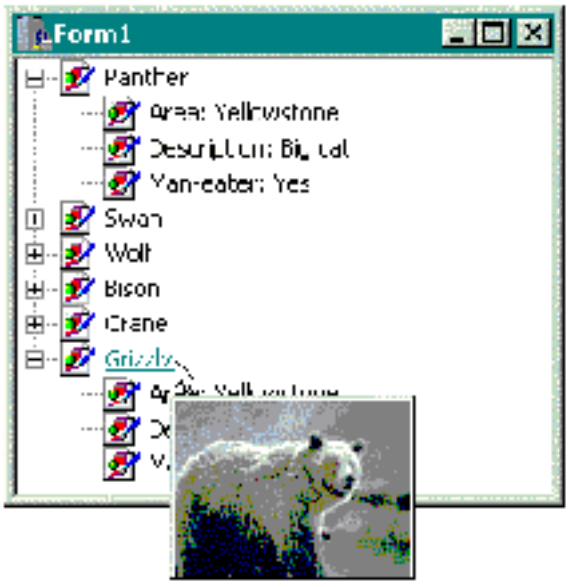


Figure B: *A tree view with per-item image-based tooltips.*

Creating a tooltip that contains an image is actually much simpler than you might think. Remember, the `THintWindow` class is a `TWinControl` descendant; this means that you can parent other child controls onto the tooltip. In this case, all you need to do is create a `THintWindow` descendant class that contains a `TImage` object. The following example creates such a class, which I'll call `TImageHintWindow`. The declaration of this class is provided in **Listing B**.

Defining TImageHintWindow

Notice from **Listing B** that the `TImageHintWindow` class contains one private member: `Image_`. This member—which is exposed via the `TImageHintWindow::Image` property—is simply the `TImage` object that the tooltip will contain. All that is required is to create the `Image` and set a few of its properties. This can be done in the class constructor:

```
__fastcall TImageHintWindow::
  TImageHintWindow(TComponent* Owner)
   : THintWindow(Owner)
{
  Image_ = new TImage(this);
  Image_->Parent = this;
  Image_->AutoSize = true;
  Image_->Left = 1;
  Image_->Top = 1;

  // prevent the Image from
  // processing mouse messages
  Image_->Enabled = false;
}
```

The next task is to override the `CalcHintRect()` method. Remember, this function is used to query the size of the tooltip for a given caption (and font). But, because the tooltip doesn't display a caption, you can simply implement `CalcHintRect()` to return the size of the image, like so:

```
TRect __fastcall TImageHintWindow::
  CalcHintRect(int MaxWidth,
    const AnsiString AHint, void* AData)
{
  TRect HintRect = Image_->ClientRect;
  HintRect.Right += 4;
  // HintRect.Bottom is increased by
  // 4 pixels in the THintWindow class
  return HintRect;
}
```

That's it for the `TImageHintWindow` class. Let's now put it to good use.

Using TImageHintWindow

The code to manage a `TImageHintWindow` object (for the tree view) is similar to that of the list box-

related example. As before an instance of the tooltip's class is created from within `Form1`'s constructor, like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner),
  last_node_(NULL)
{
  HintWindow_ =
    new TImageHintWindow(this);
  HintWindow_->Brush->Color = clWindow;
}
```

Note that this time `HintWindow_` is declared as a `TImageHintWindow*` (as opposed to the previous example). And, whereas in the previous example I used a `TStrings*`-type member (`HintList_`) for the list of tips, this time I use a `TImageList` component (`HintImageList`) that contains the list of images. **Listing C** contains the declaration of the `TForm1` class for this example.

Because you don't have to fuss with the tooltip's caption, the `DoActivateHint()` method actually becomes slightly simpler:

```
void __fastcall TForm1::DoActivateHint(
  const TPoint& P)
{
  // compute the tooltip's rect
  TRect HintRect =
    HintWindow_->CalcHintRect(
      0, NULL, NULL
      );

  // position the tooltip's rect
  OffsetRect(
     static_cast<PRECT>(&HintRect),
     P.x + 10, P.y + 10
     );

  // display the tooltip
  HintWindow_->ActivateHint(
    HintRect, NULL
    );

  // reset the time-out counter
  tic_ = 0;
```

```
  // enable the time-out timer
  Timer1->Enabled = true;
}
```

Now simply use the tree view's mouse-related events to incite the tooltips. This time I use the `TTreeView::GetNodeAt()` method to query the "hit" node, and the `TImageHintWindow::Image` property to set the tooltip's image (which is extracted from the image list via the `TImageList::GetBitmap()` method). Here's the code:

```
void __fastcall TForm1::
  TreeView1MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
  // find the node that's hit, if any
  TTreeNode* current_node =
    TreeView1->GetNodeAt(X, Y);

  // if a node is hit and it's not the
  // last node that was hit and it's
  // not a child node...
  if (current_node &&
      current_node != last_node_ &&
      current_node->Level == 0)
  {
    // set the image to display
    HintImageList->GetBitmap(
     current_node->Index,
     HintWindow_->Image->Picture->Bitmap
    );

    // display the tooltip
    DoActivateHint(TreeView1->
      ClientToScreen(Point(X, Y))
      );
  }
  else if (!current_node ||
           current_node->Level > 0)
  {
    // dismiss the tooltip
    DoHideHint();
  }

  // update the last node
```

```
  last_node_ = current_node;
}

void __fastcall TForm1::
  TreeView1MouseDown(TObject *Sender,
    TMouseButton Button,
    TShiftState Shift, int X, int Y)
{
  // dismiss the tooltip
  DoHideHint();
}
```

The timer-related code is virtually identical to that of the previous example; you simply change the `Timer1Timer()` method to use the tree view's bounding rectangle instead of the list box's, and you clear `last_node_` instead of `last_index_`.

Using the HintWindowClass variable

I've just shown you an example of creating and using a `THintWindow` descendant class that displays an image. You might have noticed from this example that creating the descendant class was the easy part; most of the work involved implementing the code to display the tooltips. (In newer versions of C++Builder you can avoid much of this work by using the `TApplication::ActivateHint()` method.)

Suppose you create a `THintWindow` descendant class that you want displayed for every control within your application. You certainly don't want to deal with manually displaying a tooltip for every control. As it turns out, the VCL provides a simple solution: the global `HintWindowClass` variable.

By using the `HintWindowClass` variable, you can instruct the `TApplication` class to use your `THintWindow` descendant class for all tooltips that your application will display. For example, if you create a `THintWindow` descendant class called `TMyHintWindow`, you'd use the `HintWindowClass` variable like so:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  HintWindowClass =
    __classid(TMyHintWindow);
}
```

This code instructs the `TApplication` class to create and maintain an instance of the `TMyHintWindow` class instead of the default `THintWindow` class. This provides an easy way to

replace all tooltips in your application.

Conclusion

I've demonstrated how to use the `THintWindow` class to display your own tooltips for a list box and a tree view. Similar techniques can be used to display tooltips for other controls. You can download the source code for both of these examples from **www.bridgespublishing.com**.

Listing A: *The declaration of the TForm1 class for the list-box-related example*

```
#include <memory>
class TForm1 : public TForm
{
__published:
  TListBox *ListBox1;
  TTimer *Timer1;
  void __fastcall ListBox1MouseMove(
    TObject *Sender, TShiftState Shift,
    int X, int Y);
  void __fastcall ListBox1MouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  void __fastcall Timer1Timer(
    TObject *Sender);

private:
  int last_index_;
  unsigned char tic_;
  THintWindow* HintWindow_;
  std::auto_ptr<TStrings> HintList_;

protected:
  virtual void __fastcall DoActivateHint(
    const TPoint& P, AnsiString TipText);
  virtual void __fastcall DoHideHint();

public:
  __fastcall TForm1(TComponent* Owner);
  const static unsigned char MAXTIC = 10;
};
```

Listing B: *The declaration of the TImageHintWindow class*

```
class TImageHintWindow : public THintWindow
{
__published:
  __property TImage* Image =
    {read = Image_, write = Image_};

private:
  TImage* Image_;

public:
  __fastcall TImageHintWindow(
    TComponent* Owner);

  // we'll override this method
  virtual TRect __fastcall CalcHintRect(
    int MaxWidth, const AnsiString AHint,
    void* AData);
};
```

Listing C: *The declaration of the TForm1 class for the tree view-related example*

```
#include "ImageHintWindow.h"
class TForm1 : public TForm
{
__published:
  TTreeView *TreeView1;
  TImageList *ImageList1;
  TImageList *HintImageList;
  TTimer *Timer1;
  void __fastcall TreeView1MouseMove(
    TObject *Sender, TShiftState Shift,
    int X, int Y);
  void __fastcall TreeView1MouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  void __fastcall Timer1Timer(
    TObject *Sender);

private:
  TTreeNode* last_node_;
  unsigned char tic_;
  TImageHintWindow* HintWindow_;
```

```
protected:
  virtual void __fastcall DoActivateHint(
    const TPoint& P);
  virtual void __fastcall DoHideHint();

public:
  __fastcall TForm1(TComponent* Owner);
  const static unsigned char MAXTIC = 10;
};
```

A Custom exception handler

by Mark G. Wiseman

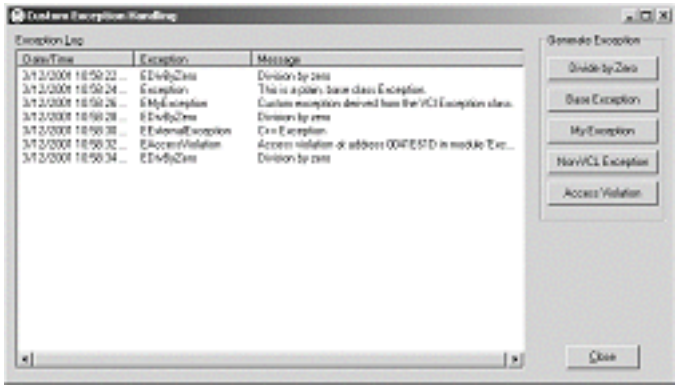
The VCL has a very nice built-in exception handler. If an exception occurs in a VCL program, the exception will be caught and a message box displays a message explaining the exception. There are times, however, when you may want to handle exceptions yourself. You may want to display a custom dialog box when a exceptions occur, log exceptions to a file for later review or handle exceptions internally, not showing anything to the user.

In this article I show you how to write a custom exception handler. As with lot of programming tasks, the VCL makes creating and using a custom exception handler very easy.

I've written a sample application that creates a custom exception handler. **Figure A** shows this application after a few exceptions have been generated by clicking the buttons on the right side of the form. The custom exception handler logs all the exceptions to a `TListView` control.

A custom message box is displayed when a divide-by-zero exception is thrown. All other exceptions are passed on to the normal VCL exception handler. You can find this sample application on the Bridges Publishing Web site.

Figure A:



Sample application using a custom exception handler

The OnException event

The `TApplication` class has an event named `OnException`. To create a custom exception handler, all you need to do is write an exception handling function and assign that function to the `OnException` event.

Here is the VCL `typedef` for a function you can assign to `OnException`:

```
typedef void __fastcall
  (__closure *TExceptionEvent)
  (System::TObject* Sender,
  Sysutils::Exception* E);
```

In the sample application, I created a function in the `TMainForm` class that conforms to this `typedef`:

```
void __fastcall TMainForm::OnException(
  TObject* sender, Exception* exception)
{
  Log(exception);

  if (String(exception->ClassName())
        == "EDivByZero")
  {
    exception->Message += "\r\n\r\nYou"
      " shouldn't divide by zero!";
    Application->MessageBox(
      exception->Message.c_str(),
      "Whoa!", MB_OK | MB_ICONSTOP);
  }
  else
    Application->
      ShowException(exception);
}
```

I named this function `OnException()` and I assign it to the `OnException` event of `TApplication` in the form's constructor with this line of code:

```
Application->OnException = OnException;
```

As you can see, the `OnException()` function is not complicated. `TApplication` passes in a pointer to an `Exception` object. First, I log the exception by calling a function, `Log()`, with this pointer. `Log()` is another simple function that adds the name of the exception class, the text message associated with the exception, and the date and time to a `TListView`.

Next, I test the exception to determine if its class name is `EDivByZero`. If it is, I display a custom message box. If the exception in not of type `EDIvByZero`, I invoke the default VCL exception handler by calling the `ShowException()` function in `TApplication`.

Exceptions, exceptions and exceptions

There are three different types of exceptions that might occur in a Windows program: a VCL exception, a C++ exception, or a Windows structured exception. The VCL converts all of these to VCL exceptions. VCL exceptions are all derived from the VCL `Exception` class. In the sample application, I have written code to throw all three types.

The sample application will throw two different VCL exceptions: one using the base class `Exception` and one using an exception class I created, `EMyException`. Here is the code that causes these two exceptions:

```
throw Exception("This is a plain, "
  "base class Exception.");
```

and

```
throw EMyException();
```

The `EMyException` class is derived from the `Exception` class:

```
class EMyException : public Exception
{
  public:
    EMyException() : Exception(
      "Custom exception derived from "
      "the VCl Exception class.") {}
};
```

Two Windows structured exceptions are generated: attempting to divide by zero, and attempting to use a `NULL` pointer to call a function. The VCL converts these into the VCL exception `EDivByZero` and `EAccessViolation` respectively. Here is the code I use to cause the divide by zero exception:

```
int a = 0, b = 3;
int c = b / a;
b = c;
```

The code I used to cause the invalid pointer exception is as follows:

```
TForm *badPtr = 0;
badPtr->Close();
```

Both of these classes are derived from `EExternal` (which is derived from `Exception`). `EExternal` is the VCL class that captures Windows structured exceptions.

The sample application also throws an `int` to demonstrate how the VCL handles a plain C++ exception:

```
throw 12;
```

The VCL converts this exception into an `EExternalException`. The `EExternalException` class is derived from `EExternal` and represents an exception that the VCL does not recognize. The VCL does not preserve the value of the `int`, in this case `12,` when it converts the exception to `EExternalException`.

A handy new component

Version 5 of C++Builder has a handy new component named `TApplicationEvents`. You can find this component on the Additional tab of the Component Palette. This component makes creating handlers for events in `TApplication`, such as `OnException`, even easier.

Just drop a `TApplicationEvents` component on your form, click on the Events tab of the Object Inspector and then double-click on the `OnExceptions` event. The IDE will create the skeleton of an exception handler function for you.

A couple of exceptions

A custom exception handler assigned to the `OnException` event of `TApplication` will handle all of the exceptions thrown in application with a couple of exceptions (pun intended). For `TApplication` to catch an exception and fire the `OnException` event, the exception must occur within the scope of the `Run()` method in `TApplication`. If an exception is thrown before `Run()` is called or after `Run()` has returned, the `OnException` event will not occur.

Listing A: *Main unit for the Exception example program.*

```
#include <vcl.h>
#pragma hdrstop

#include <except.h>

#include "Main.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
```

```
TMainForm *MainForm;


__fastcall TMainForm::TMainForm(
  TComponent* Owner) : TForm(Owner)
{
  Caption = Application->Title;

  sortType = 0;
  sortAscending = true;

  Application->OnException = OnException;
}

void __fastcall TMainForm::CloseBtnClick(
  TObject *Sender)
{
  Close();
}

void __fastcall TMainForm::ZeroBtnClick(
  TObject *Sender)
{
  int a = 0, b = 3;
  int c = b / a;
  b = c;
}

void __fastcall TMainForm::OnException(
  TObject* sender, Exception* exception)
{
  Log(exception);

  if (String(exception->ClassName())
        == "EDivByZero")
  {
    exception->Message += "\r\n\r\nYou really "
      "shouldn't divide by zero!";
    Application->MessageBox(
      exception->Message.c_str(), "Whoa!",
      MB_OK | MB_ICONSTOP);
  }
  else
    Application->ShowException(exception);
```

```
}

void __fastcall TMainForm::Log(
  Exception *exception)
{
  TListItem *item = LogListView->Items->Add();
  item->Caption = Now().DateTimeString();

  item->SubItems->Add(exception->ClassName());
  item->SubItems->Add(exception->Message);
}

void __fastcall TMainForm::BaseBtnClick(
  TObject *Sender)
{
  throw Exception(
    "This is a plain, base class Exception.");
}

void __fastcall TMainForm::LogListViewColumnClick(
  TObject *Sender, TListColumn *Column)
{
  if (sortType == Column->Tag)
    sortAscending = !sortAscending;
  else
  {
    sortAscending = true;
    sortType = Column->Tag;
  }

  LogListView->AlphaSort();
}

void __fastcall TMainForm::LogListViewCompare(
  TObject *Sender, TListItem *Item1,
  TListItem *Item2, int Data, int &Compare)
{
  int sortDir = sortAscending ? 1 : -1;

  TDateTime d1(Item1->Caption);
  TDateTime d2(Item2->Caption);
  int cd = d1 < d2 ? - 1 : d1 > d2 ? 2 : 0;

  switch (sortType)
```

```
  {
    case 0:
      Compare = cd;
      break;
    case 1:
    {
      int c = CompareText(
        Item1->SubItems->Strings[0],
        Item2->SubItems->Strings[0]);
      Compare = c == 0 ? cd : c;
      break;
    }
    case 2:
    {
      int c = CompareText(
        Item1->SubItems->Strings[1],
        Item2->SubItems->Strings[1]);
      Compare = c == 0 ? cd : c;
      break;
    }
  }

  Compare *= sortDir;
}

void __fastcall TMainForm::MyBtnClick(
  TObject *Sender)
{
  throw EMyException();
}

void __fastcall TMainForm::NonVCLBtnClick(
  TObject *Sender)
{
  throw 12;
}

void __fastcall TMainForm::AccessBtnClick(
  TObject *Sender)
{
  TForm *badPtr = 0;
  badPtr->Close();
}
```

A home cooked Outlook bar

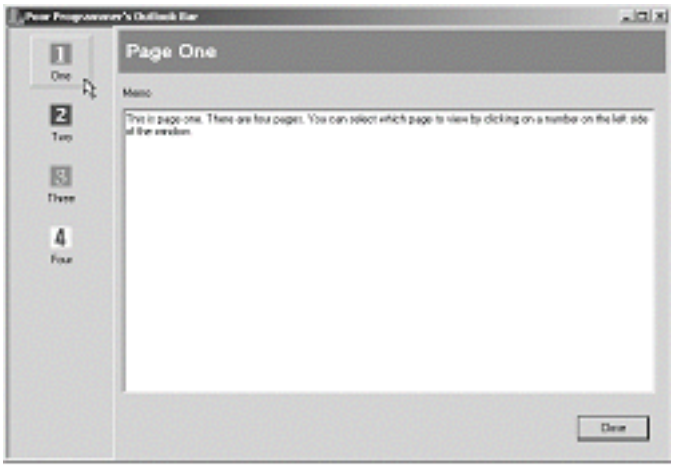
by Mark G. Wiseman

So, you really like the way the “Outlook bar” looks and works in Microsoft products; products like, well, Outlook. You’d like to use an Outlook bar in your program, but you’ve noticed that Borland didn’t include one with the VCL. You only wanted to use the Outlook bar in a small utility program, so you really don’t want to spend the big bucks for a third-party component. How do you satisfy your hunger?

Well, you can cook up a form that contains a reasonable facsimile of an Outlook bar and you can do this using ingredients from standard VCL kitchen. It won’t have all of the functionality of the real McCoy, but it will still taste great. In this article, I’ll give you the recipe.

You can find a sample application on the Bridges Publishing Web site. **Figure A** shows what the Outlook bar on the sample application looks like.

Figure A: *Sample application with Outlook bar*



The recipe

The Outlook bar consists of `TSpeedButton` components placed on a `TScrollBox` component. Next to the `TScrollBox` is a `TPageControl` component that contains one page for each `TSpeedButton` in the `TScrollBox`. The title of the active page is displayed in a `TPanel` component above the `TPageControl`.

A cup of panels

Start by creating a new application in the C++Builder IDE. Drop a `TScrollBox` component onto the

blank form. The TScrollBox component is on the Additional tab of the IDE's Component Palette. Set the Align property of the TScrollBox to alLeft and the Width property to 112.

Now drop a TPanel onto the form, to the right of the TScrollBox. Set the TPanel's Align property to alClient, the BevelOuter property to bvNone and BorderWidth to 4. Also clear the text out of the Caption property. I'll refer to this TPanel as the Background Panel.

Next drop another TPanel onto the Background Panel. Set Align to alTop, Alignment to taLeftJustify, BevelOuter to bvNone, Color to clBtnShadow, and BorderWidth to 8. You will also need to change some of the sub-properties of the TPanel's Font property. In particular set Color to clCaptionText, Height to -19 and Style to fsBold. I will refer to this TPanel as the Caption Panel.

A pinch of pages

At this point you will add multiple pages to the form. This is accomplished by dropping a TPageControl onto the Background Panel beneath the Caption Panel. Set the Align property to alClient and set Style to tsFlatButtons.

Now add four pages to the TPageControl. Add a page by right-clicking on the TPageControl and selecting New Page from the menu. A new TTabSheet is added.

Set the Caption property of the TTabSheet to "Page One" and the TabVisible property to false. Create three more TTabSheets and set their TabVisble properties to false as well. Set their Caption properties to "Page Two", "Page Three" and "Page Four".

A dash of buttons

Finally, drop four TSpeedButton components on to the TScrollBox on the left side of the form. Place them vertically in the TScrollBox and center them horizontally in the TScrollBox. Refer to **Figure A** if this isn't clear.

You can change some of the properties of the TSpeedButtons in the TScrollBox by selecting all four of them (click on each while holding the shift key down). Set the Flat property to true, Height to 54, Layout to blGlyphTop and Width to 64.

While all four of the TSpeedButtons are still selected, select the OnClick event in the Object Inspector and type in "OBClick" as the name of the event and hit the enter key. The IDE will generate the skeleton for the OBClick() function. Complete the code for this function so that it looks like this:

```
void __fastcall TForm1::OBClick(
```

```
  TObject *Sender)
{
  TComponent *component =
    dynamic_cast<TComponent *>(Sender);
  if (component == 0) return;
  SetPage(component->Tag);
}
```

The `OBClick()` function casts the `Sender` parameter to a `TComponent` pointer and checks to make sure the cast worked. If the cast failed `OBClick()` returns. If the cast succeeds, `OBClick()` calls the `SetPage()` function passing the `Tag` value of the `TComponent` as the `page` parameter. Here is the code for the `SetPage()` function:

```
void __fastcall TForm1::SetPage(
  int page)
{
  if (page < 0 || page >
      PageControl1->PageCount - 1)
    return;
  PageControl1->ActivePageIndex = page;
  Panel2->Caption =
    PageControl1->ActivePage->Caption;
}
```

This function first checks the value of `page` to be sure it's valid then activates the page by setting the `ActivePageIndex` property of `PageControl1` to `page`. Finally, `SetPage()` changes the page name displayed in the Caption Panel to the `Caption` of the active page.

At this point you are nearly finished building your Outlook bar. Select only the top `TSpeedButton` and change its `Caption` property to `"One"`. Add a 24 x 24 pixel bitmap to the `Glyph` property. The `Tag` property is `0`, so you can leave that unchanged. Moving down, individually select each of the other three `TSpeedButtons`. Set the `Caption` properties to "Two", "Three", and "Four" respectively, and add some hand-made bitmaps to the `Glyph` property. Set the respective `Tag` properties to `1`, `2`, and `3`.

In case you haven't figured it out, the `Tag` property of each `TSpeedButton` is equal to the zero-based index of the `TTabSheet` pages in the `TPageControl`.

Stick a fork in it

There's one last thing to do. To make sure everything is synchronized when the program starts, add one line of code to the constructor of the `TForm`:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  SetPage(0);
}
```

It’s done, and by my count, you wrote only seven lines of code. Delicious and low cal!

An enhanced type safe TList

by Cris Gallegos

Quite often, when writing an application, one comes across the need to manage, store, and retrieve varying amounts of homogenous information. For example, in a recent project I had to provide my users with a way to create, delete, modify, store, and retrieve an unrestricted, number of flow meter configuration profiles. This type of information is often contained within multiple instances of a custom written class. In the past, when faced with this type of need, I've chosen to employ a derivative of the humble `TList` as a container for managing the custom class instances.

For anyone who doesn't know, `TList` is part of the Visual Component Library. It is declared and defined within the CLASSES.PAS file. It's an Object Pascal construct that maintains a list of `void` (i.e. generic or untyped) pointers to other objects. `TList` has properties and methods that allow one to add, delete, rearrange, sort, locate, and access the pointers. It's inherent flexibility, efficiency, and simplicity makes it a powerful tool.

That very same flexibility and simplicity also leads to `TList's` greatest problems. For instance, `TList` doesn't provide any measure of type safety. It can't assume any ownership of the objects that are represented by the pointers that it manages. Any of the `TList` methods or properties that return a pointer, can only return a `void` pointer. Also, using `TList's` built in sorting functionality can be somewhat less than intuitive.

Until recently, I've solved these problems by creating different classes that inherited from `TList`. They were written with strongly typed replacements for the standard `TList` properties and methods. They also encompassed the means to store and retrieve object information. These classes were implemented via boilerplate code but, by necessity, became application specific as I used them. While this worked well enough in the short term, it didn't provide the flexibility and re-usability I was looking for.

A Better Way

The solution was to develop a C++ class template that both encapsulated and enhanced `TList`. An introduction to the elegance and power of C++ templates is beyond the scope of this article. Many C++ programming books provide information on C++ templates. I encourage anyone who doesn't already know about templates to learn about them. As the following example will show, they can be very useful.

I wound up splitting my solution into three parts. The first part, `TssTempList`, provides the actual encapsulation of `TList`. A partial code listing is shown in **Listing A**. `TssTempList` has no provisions for permanent storage and retrieval of object information. If needed, application specific code can be custom written to store and retrieve object information in whatever fashion is required. The other two,

TssDiskList and TssRegList, inherit from TssTempList and provide the ability to read and write basic object information to or from a disk file or the system registry. TssDiskList and TssRegList, along with some of the more advanced aspects of TssTempList, will be presented in part two of this article. The full source code to all three is available on the Bridges Publishing web site.

Implementation

The first thing you might notice is that the TssTempList class is actually declared twice. The first declaration insures that TssTempList can't be instantiated by specifying a fundamental data type (int, bool, float, etc.). The second declaration insures, via partial specialization, that TssTempList can only be instantiated by specifying a pointer type (notice the unique template signature).

The second thing you might notice is that I chose to inherit from TObject instead of directly from TList. A TList object is then internalized as a private data member. This prevents direct access to potentially troublesome things (like it's List property). Descending from TObject also helps ensure that the only constructor that can be successfully called, is the one explicitly defined for TssTempList. It also helps provide a more consistent public interface over time (the Extract() method was added to TList in C++Builder 5).

The parameter and initialization lists for the second TssTempList's constructor look a little cumbersome at first. All they really do is initialize the three constant data members: F_ALLOW_CREATE, F_ALLOW_DELETE, and F_ALLOW_NULL.

F_ALLOW_CREATE allows TssTempList to provide factory type services. Its use will be covered in detail in the second part of this article. F_ALLOW_DELETE is used in the Clear(), Delete(), Put(), and Remove() methods. If true, the above methods will automatically delete any objects referenced by the pointers as they are removed from the underlying list. F_ALLOW_NULL is used in the Add(), Insert(), and Put() methods. If true, NULL pointers cannot be added to the list.

The Add() and Insert() methods and the Items property (when it calls Put()) will only accept pointers to type "T", or NULL pointers (assuming F_ALLOW_NULL is true). This is where the template itself provides strict type checking services. Only one type of pointer per TssTempList instance will be accepted for storage. This guarantees that the typcasting services provided by methods such as First() and Last() and the Items property (when it calls Get()) will be safe. The worst that can ever be returned, assuming that F_ALLOW_NULL is true, is a NULL pointer.

Usage

Now look at an example of how to use TssTempList:

```
class TMeterCfg
{
public:
  String MeterName;
};

TMeterCfg * pMC = NULL;//Used later.
TssTempList<float> * MeterList;
  //Compiler error! Must be initialized
  //with a pointer to object.
TssTempList<TMeterCfg*> * MeterList;
  //Could be declared in header file.
MeterList =
  new TssTempList<TMeterCfg*>(
    false, true, false);
  //Frees memory used by objects.
  //Won't accept NULL pointers.
for(int i = 0; i < 5; i++)
{
  MeterList->Add(new TMeterCfg);
  pMC = MeterList->Items[i];
    //Allows direct assignment.
  MeterList->Items[i]->MeterName =
    "Meter #" + IntToStr(i);
    //Direct access to data and methods.
}
MeterList->IndexOf(this);
  //Compiler error!
  //Not a TMeterCfg pointer!
MeterList->Insert(3, NULL);
  //TssTempList error!
  //No NULL pointers allowed!
MeterList->Delete(3);
  //Only four objects left in memory.
delete MeterList;
  //All remaining objects
  //are deleted from memory.
```

I must provide you with one word of caution. While the compiler doesn't allow `TssTempList` to be instantiated by directly specifying a fundamental data type, it will allow instantiation by specifying a pointer to one. This can lead to unreliable behavior if the size of the data type is 32 bits or less. I've traced this behavior back to `TList` itself.

Conclusion

As you can see, `TssTempList` greatly simplifies the use of `TList`. It's type checking services help eliminate type mismatch errors while it's typecasting services simplify the setting and getting of object information. Due to the inlining of it's methods, the extra operational overhead imposed by using `TssTempList` is minimal.

In the next installment, I will explain how the `F_ALLOW_CREATE` member is used. I will also show you the workings of the `TssTempList` sorting mechanism. I will extend the functionality of `TssTempList` to provide for the storage and retrieval of basic object information by deriving the `TssDiskList` and `TssRegList` templates. See you then.

Listing A: *The declaration of the TssTempList class*

```
/*********************************************
  Copyright (c) 2000 - 2001
  by Cristobal J. Gallegos
*********************************************/
#ifndef TempListH
#define TempListH

//First version. See article.
template <class T>
class TssTempList : System::TObject
{
private:
  TssTempList();//Constructor not defined.
  TssTempList(const TssTempList&);
    //Copy Constructor not defined.
};

//Second version. See article.
template <class T>
class TssTempList< T * > : TObject
{
protected:
  const bool
    F_ALLOW_CREATE, F_ALLOW_DELETE, F_ALLOW_NULL;

  TList * FList;

  T * Get(int Index)
  {
```

```
    return reinterpret_cast
      < T * >(FList->Items[Index]);
  }

  int GetListCount(){ return FList->Count; }

  inline void Put(int Index, T * Item);

public:
  TssTempList(
    bool AllowCreation,
    bool AllowDeletion,
    bool AllowNULL) :
      F_ALLOW_CREATE(AllowCreation),
      F_ALLOW_DELETE(AllowDeletion),
      F_ALLOW_NULL(AllowNULL)
  {
    FList = new TList;
  }

  virtual __fastcall ~TssTempList()
  {
    Clear();//Have to call this manually.
    delete FList;
  }

  inline int Add(T * Item);

  inline void Clear();

  int __property Count
    = {read=GetListCount, write=SetListCount};

  inline void Delete(int Index);

  void Exchange(int Index1, int Index2)
  {
    FList->Exchange(Index1, Index2);
  }

  T * First(void)
  {
    return reinterpret_cast
      < T * >(FList->First());
  }
```

```
  int IndexOf(T * Item)
  {
    return FList->IndexOf(Item);
  }

  inline void Insert(int Index, T * Item);

  __property T *
    Items[int Index] = {read=Get, write=Put};

  T * Last(void)
  {
    return reinterpret_cast< T* >(FList->Last());
  }

  void Move(int CurIndex, int NewIndex)
  {
    FList->Move(int CurIndex, int NewIndex);
  }

  void Pack(void)
  {
    FList->Pack(void);
  }

  inline int Remove(T * Item);
};

template <class T>
int TssTempList< T * >::Add(T * Item)
{
  int Result = -1;
  try
  {
    if(Item == NULL && !F_ALLOW_NULL)
      throw EInvalidOperation("");
    Result = FList->Add(Item);
  }
  catch(EInvalidOperation &E)
  {
    //Warn the user
  }
  return Result;
```

```
}

template <class T>
void TssTempList< T * >::Clear()
{
  if(F_ALLOW_DELETE)
  {
    for(int i = 0; i < FList->Count; i++)
      delete Get(i);//Delete the object.
  }
  FList->Clear();//Delete pointers.
}

template <class T>
void TssTempList< T * >::Delete(int Index)
{
  if(F_ALLOW_DELETE)
    delete Get(Index);//Delete the object.
  FList->Delete(Index);//Delete pointer.
}

template <class T> void
TssTempList< T * >::Insert(int Index, T * Item)
{
  try
  {
    if(Item == NULL && !F_ALLOW_NULL)
      throw EInvalidOperation("");
    FList->Insert(Index, Item);
  }
  catch(EInvalidOperation &E)
  {
    //Warn the user.
  }
}

template <class T> void
TssTempList< T * >::Put(int Index, T * Item)
{
  try
  {
    if(Item == NULL && !F_ALLOW_NULL)
      throw EInvalidOperation("");
    if(F_ALLOW_DELETE)
```

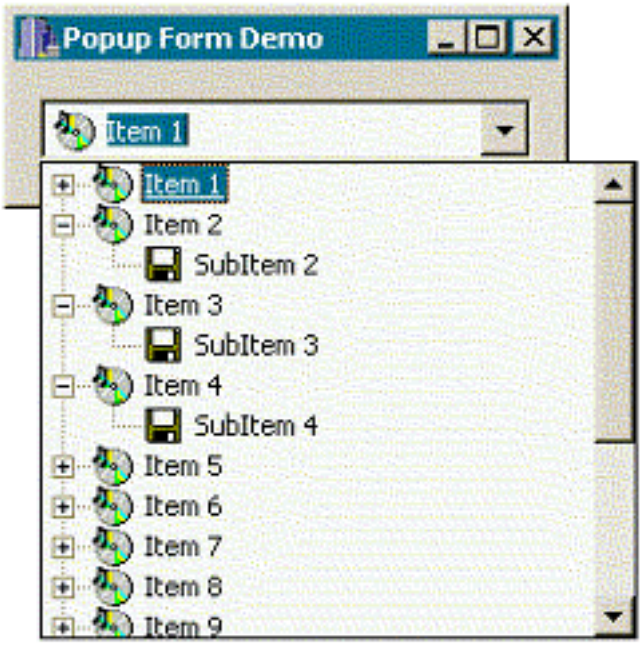
```
      delete Get(Index);//Delete the object.
    FList->Items[Index] = Item;
  }
  catch(EInvalidOperation &E)
  {
    //Warn the user.
  }
}

template <class T>
int TssTempList< T * >::Remove(T * Item)
{
  int Result = FList->Remove(Item);
  if(F_ALLOW_DELETE && Result >= 0)
    delete Item;
  return Result;
}

#endif
```

Custom popup controls

by Damon Chandler

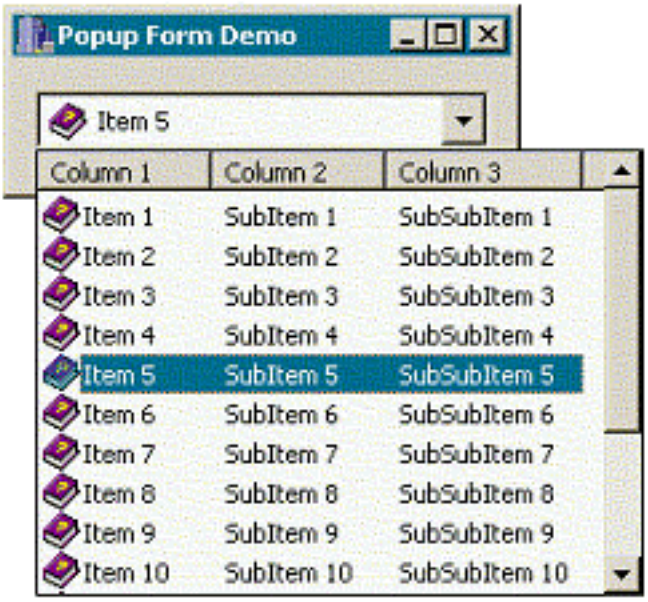
In this article, I'll show you how to create a generic popup form, which can serve as the drop-down portion of a combo box or some other control. **Figures A**, **B**, and **C** depict some examples.

Figure A



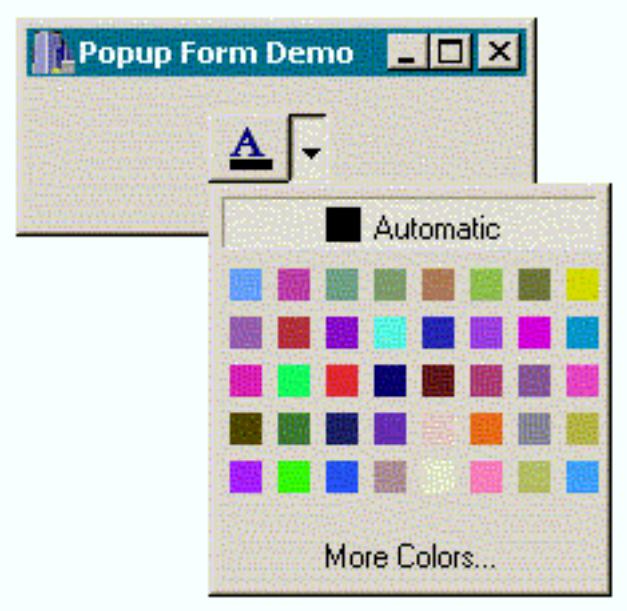
A combo tree-view control.

Figure B



A combo list view control.

Figure C



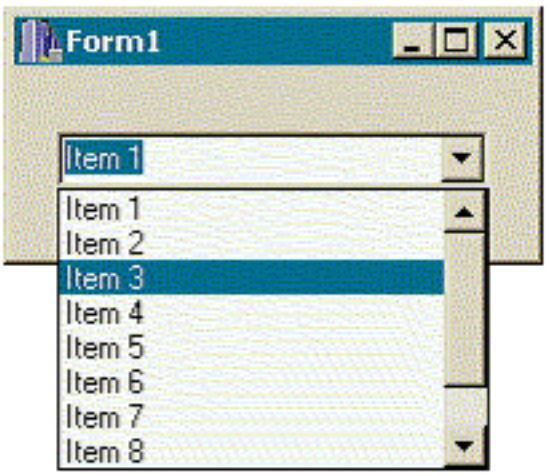
A combo color-picker control.

The `ComboLBox` class

The `ComboLBox` class is a native standard control class whose windows serve as the list box, or "drop-down", portion of a standard style combo box. `ComboLBox` windows are similar to both child and top-level windows, but they are unique in two respects: First, they are children of the desktop window, which means that when a `ComboLBox` window is shown, its contents are not limited to the client area of a

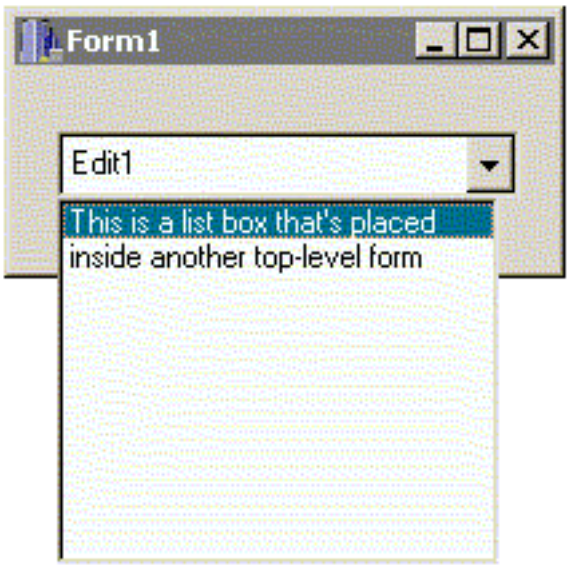
specific top-level window. If you've ever placed a combo box at the bottom of your form, you'll recall that the drop-down portion can indeed exceed the bounds of the form (see **Figure D**). Second, unlike true top-level windows, a `ComboLBox` window can be displayed without forcing the active window to lose its active status. For example, instead of using a true combo box, you might be tempted to use an edit control, a button, and a small borderless form (for the drop-down portion). As depicted in **Figure E**, however, when you display another top-level window—borderless or not—the currently active window will lose activation.

Figure D



A ComboLBox window can exceed the bounds of its associated form.

Figure E



Mimicking a ComboLBox by displaying another top-level window will cause the main form to lose

activation.

So, how does a `ComboLBox` window do it? How does it exceed the bounds of the form to which the combo box itself is parented without causing that form to lose activation? Well, as mentioned earlier, a `ComboLBox` window is a child of the desktop window. Because it's a child of the desktop window, a `ComboLBox` window is not limited to the client area of a specific form. In a sense, all top-level windows are children of the desktop, but a `ComboLBox` window is a *true* child window—that is, it uses the `WS_CHILD` window style. It's this window style that prevents the active window from losing activation when the `ComboLBox` window is displayed.

The `TPopupForm` class

The goal is to create a `TForm` descendant class that mimics the `ComboLBox` class. To do this, create a `TForm` descendant instead of a `TListBox` descendant. This is done for two reasons: first, a `TForm` descendant can be manipulated at design time; and second, because a `TForm` object is a container control (`csAcceptsControls`), other child controls can be placed within the form. This way, the drop-down window won't be limited to just a list box. The declaration of this descendant class—which I'll call `TPopupForm`—is provided in **Listing A**. I will provide a definition for each of its member functions in the following sections.

Specifying the window styles

The first task is to ensure that each `TPopupForm` window is created with the correct window styles; specifically those that are used by the `ComboLBox` class. To this end, you can override the `CreateParams()` member function and provide the following definition:

```
void __fastcall TPopupForm::
  CreateParams(TCreateParams& Params)
{
  TForm::CreateParams(Params);
  Params.Style &=
    ~(WS_CAPTION | WS_SIZEBOX |
      WS_POPUP);
  Params.Style |= WS_CHILD | WS_BORDER;
  Params.ExStyle |= WS_EX_PALETTEWINDOW;
}
```

You may recall that the `CreateParams()` member function is an ideal place to manipulate the styles and extended styles that are later passed to the `CreateWindowEx()` API function. Because a drop-down window should not possess a title bar or a resizable border, the first step is to remove the `WS_CAPTION` and `WS_SIZEBOX` styles. The `WS_POPUP` style is also removed because this style cannot be used in combination with the `WS_CHILD` style. The `WS_CHILD` style is specified to make the

form a true child window. The WS_BORDER style is set in order to achieve the thin-line border that's inherent to ComboLBox windows. For the extended styles, the WS_EX_PALETTEWINDOW style is set. This serves two purposes. First, it makes the form top-most, which ensures that other windows won't obscure the form. Second, it makes the form a tool window, which ensures that it won't appear in the task list that is displayed when the Alt+Tab keystroke combination is pressed.

Specifying the parent window

As I mentioned earlier, the drop-down form needs to be a child of the desktop window so that the currently active form doesn't lose its active status when the drop-down form is displayed. The WS_CHILD style has been specified, so now the parent window needs to be set. The SetParent() API function is used for this purpose. Call SetParent() and pass a handle to the desktop window (NULL) as the function's second parameter. As its first parameter, the SetParent() function requires a handle to the window whose parent is to be changed (i.e., a handle to the drop-down form). The SetParent() function must be called after the TPopupForm window has been created. To this end, call the function from within an overridden CreateWnd() function, like so:

```
void __fastcall TPopupForm::CreateWnd()
{
  TForm::CreateWnd();
  ::SetParent(
    Handle, GetDesktopWindow());
}
```

At this point, the drop-down form will appear as desired when it's displayed, but it won't *behave* like a true ComboLBox window. Specifically, steps must be taken to ensure that the drop-down form closes when the user clicks within its client area, and that it closes when the user dismisses it by clicking somewhere else. I will address the latter task first.

Handling the mouse messages

The task of closing the TPopupForm window is trivial—simply call the Close() function. The question is, *when* should this function be called (in response to what event)? Well, you could use the TPopupForm's OnMouseDown event and then test the X and Y parameters to see if the user clicked outside of its client area:

```
void __fastcall
TPopupForm::FormMouseDown(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
```

```
  // if the specified point is beyond
  // the bounds of the form...
  if (X < 0 || X >= Width ||
      Y < 0 || Y >= Height)
  {
     // close the form
     Close();
  }
}
```

The problem, however, is that the `OnMouseDown` event is typically fired only when the user has pressed a mouse button within the window's client area. You need the `OnMouseDown` event to fire when the user presses a mouse button *outside* of the window's client area. Fortunately, there's an exception to the "within client area" rule. Specifically, the `OnMouseDown` event will be called—regardless of the position of the mouse cursor—if the popup window has captured the mouse. So, in order to ensure that the `OnMouseDown` event is invoked when the user presses the mouse button beyond the bounds of the drop-down form, use the `SetCapture()` API function to give the form mouse capture. This function must be called whenever the form is displayed (i.e., within the `VisibleChanging()` function), and whenever the mouse cursor leaves the bounds of the popup form (i.e., in response to the `CM_MOUSELEAVE` VCL message). Here's the code:

```
void __fastcall
TPopupForm::VisibleChanging()
{
  TForm::VisibleChanging();

  // if the form is being hidden
  if (Visible)
  {
    ReleaseCapture();
  }
  // if the form is being shown
  else
  {
    if (ActiveControl)
    {
      SetCapture(ActiveControl->Handle);
    }
    else SetCapture(Handle);
  }
}

void __fastcall TPopupForm::
```

```
  CMMouseLeave(TMessage& AMsg)
{
  // when the cursor goes beyond the
  // bounds of our window, capture
  // the mouse...
  if (Visible)
  {
    if (ActiveControl)
    {
      SetCapture(
        ActiveControl->Handle);
    }
    else SetCapture(Handle);
  }
}
```

Notice the use of the `ActiveControl` property in both of these member functions; I'll explain the logic behind this shortly. For now, focus on the call to the `ReleaseCapture()` API function from within the `VisibleChanging()` member function. The `ReleaseCapture()` function is called to restore the mouse capture whenever the form is being hidden. This function must also be called when the mouse cursor enters the bounds of the form so that child windows (of the drop-down form) can process mouse messages normally. This latter task is accomplished by using the `CM_MOUSEENTER` VCL message, as shown here:

```
void __fastcall
TPopupForm::CMMouseEnter(TMessage& AMsg)
{
  // when the cursor is within the
  // bounds of our window, release
  // the mouse capture
  ReleaseCapture();
}
```

Handling the `WM_ACTIVATEAPP` message

There's one final message that must be handled: `WM_ACTIVATEAPP`. This message is sent with a `wParam` set to `false` whenever the user activates another application. This provides a perfect opportunity to hide the drop-down form because it shouldn't be lingering on top of the screen when the host application is not in the foreground. Here's the code that goes within the `WM_ACTIVATEAPP` message handler:

```
void __fastcall
```

```
TPopupForm::WMActivateApp(TMessage& AMsg)
{
  // if deactivating...
  if (!AMsg.WParam)
  {
    // close the form
    Close();
  }
}
```

That's all there is to the `TPopupForm` class. Next I'll show you how to put it to good use.

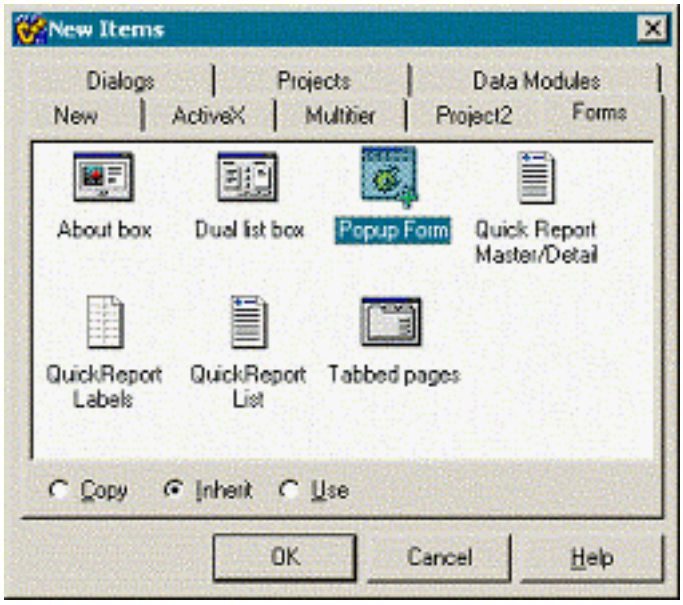
Using the TPopupForm class

After you've implemented the code for the `TPopupForm` class, the first thing you should do is add it to the Object Repository. This way, you can easily create descendant classes that you can manipulate at design time. The following sections take you through and example of creating such a descendant class, which I'll call `TTreeViewPopupForm`.

Creating the TTreeViewPopupForm class

Because the `TPopupForm` class is in the Object Repository, creating a descendant class is trivial. Just choose "File | New..." from the main menu, select the parent class, choose the "Inherit" radio button, and then confirm the dialog (see **Figure F**).

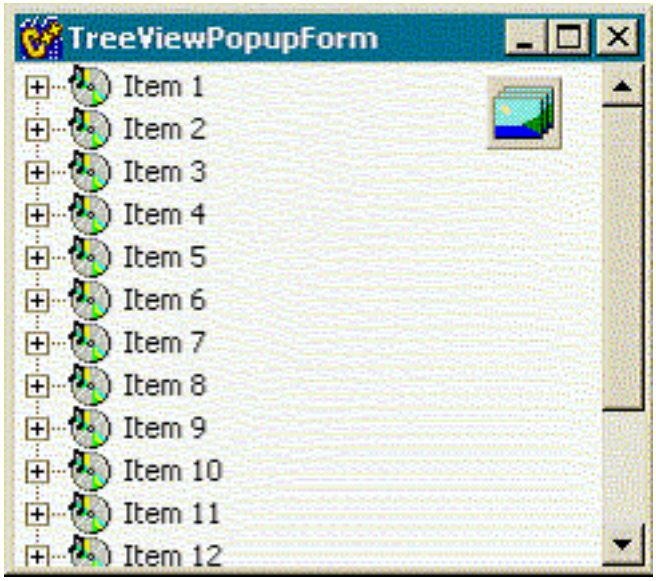
Figure F



Creating a TPopupForm descendant class by using the Object Repository.

At this point, you'll be presented with a blank `TPopupForm` descendant window, which you can easily customize. (Remember, this design-time functionality was one of the main reasons the popup form is a `TForm` descendant. C++Builder 5 users may choose to use frames, if desired.) For this example, change the `Name` of the form to "TreeViewPopupForm", and then add a `TTreeView` and a `TImageList` to it as depicted in **Figure G**.

Figure G



A TPopupForm descendant class with a TTreeView object and a TImageList object.

There are three things that must be done to the `TTreeViewPopupForm` class before it's useable. First, assign the tree-view's `OnMouseDown` event handler to `TPopupForm::FormMouseDown`. Why is this assignment required? Well, recall from the `TPopupForm::VisibleChanging()` and `CMMouseLeave()` member functions that the `SetCapture()` function is called with `ActiveControl->Handle` if the form's `ActiveControl` property is set. This way, mouse capture is given to `ActiveControl` (`TreeView1` in the present example) instead of the form so that the active control can process mouse messages as it normally would. If mouse capture were given to the form (`TreeViewPopupForm`) instead of to the child window (`TreeView1`), the child window wouldn't receive any mouse messages. In this example, this would prevent the tree-view's nodes from being expanded. Because mouse capture is given to the active control instead of the drop-down form, it is necessary to ensure that the active control's `OnMouseDown` event maps to the form's `OnMouseDown` event. To accomplish this, the `TPopupForm::FormMouseDown` event is assigned to the `OnMouseDown` event of `TreeView1`. In conjunction with that, the next step is to set the `ActiveControl` property of the form to `TreeView1`. The third step is to provide a handler for the tree-view's `OnMouseUp` event. This way, the drop-down form can be closed when the user selects a node. Here's the code for that:

```
void __fastcall
TTreeViewPopupForm::TreeView1MouseUp(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  // BCB4/5 users can alternatively use
  // the GetHitTestInfoAt function
  TV_HITTESTINFO tvhti = {X, Y};
  SNDMSG(TreeView1->Handle, TVM_HITTEST,
    0,reinterpret_cast<LPARAM>(&tvhti));

  // if an item was hit...
  if (tvhti.flags & TVHT_ONITEM)
  {
    // close the form
    Close();
  }
}
```

Displaying the TreeViewPopupForm

To display the drop-down form, set its `Top` and `Left` properties (which are relative to the screen) and then simply call the `Show()` function:

```
void __fastcall
TForm1::TreeViewButtonClick(
  TObject *Sender)
{
  RECT R;
  if (GetWindowRect(Edit1->Handle, &R))
  {
    TreeViewPopupForm->Left = R.left;
    TreeViewPopupForm->Top = R.bottom;
    TreeViewPopupForm->Show();
  }
}
```

You can show the drop-down window as part of a fake combo box (by using an edit/static control and a `TSpeedButton`), or perhaps in response to a click of the right mouse button. You can even use the drop-down form in conjunction with a toolbar to create a single-level “menu” such as the color box depicted in **Figure C**.

Conclusion

In the October 2000 issue, I showed you how to use the `OnDrawCell` event of the `TStringGrid` class to customize the appearance of a string grid. Well, now that you know how to display a generic drop-down form, you can use this method to efficiently display a combo box in each cell of a string grid. All you need to do is draw a drop-down button in each cell (by using the `DrawFrameControl()` API function), and then show the drop-down form (which contains, say, a `TListBox`) when this button is clicked.

There are a few limitations to using a drop-down form instead of a full-blown dialog. Namely, a drop-down form cannot receive keyboard focus without causing the main form to lose activation. For this reason, you should avoid using child windows that display a caret within your drop-down form. Just because a drop-down form cannot receive keyboard focus, doesn't mean it can't process keyboard messages. If you want to add keyboard navigation to the tree-view, for example, you can simply forward the `WM_KEYDOWN`/`WM_KEYUP` messages to the tree-view by using the `SendMessage()` API function. You can download a sample project that demonstrates this technique—along with the source code for the `TPopupForm` class—from **www.residorph.com**.

Listing A: *The declaration of the TPopupForm class*

```
class TPopupForm : public TForm
{
__published:
  void __fastcall FormMouseDown(
    TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);

protected:
  virtual void __fastcall CreateWnd();
  virtual void __fastcall CreateParams(
    TCreateParams& AParams);
  DYNAMIC void __fastcall VisibleChanging();

private:
  MESSAGE void __fastcall CMMouseEnter(
    TMessage& Msg);
  MESSAGE void __fastcall CMMouseLeave(
    TMessage& Msg);
  MESSAGE void __fastcall WMActivateApp(
    TMessage& AMsg);

public:
```

```
  __fastcall TPopupForm(TComponent* Owner);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CM_MOUSEENTER, TMessage, CMMouseEnter)
  MESSAGE_HANDLER(
    CM_MOUSELEAVE, TMessage, CMMouseLeave)
  MESSAGE_HANDLER(
    WM_ACTIVATEAPP, TMessage, WMActivateApp)
END_MESSAGE_MAP(TForm)
};
```

TScreening room

by Mark G. Wiseman

I recently saw some source code to log the activity of a program written to use the Borland's Visual Component Library (VCL). I was very surprised to see a couple of hundred lines of code to determine which form and control were active and to log that information to a file. It used very complicated calls to the Windows API and code searches through lists of `TComponent` objects.

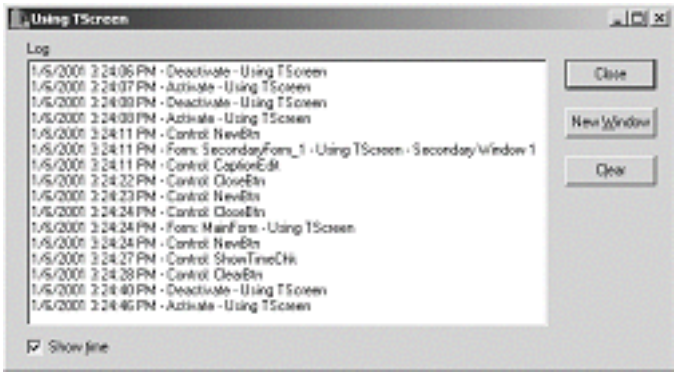
While the code was very well written, there was absolutely no reason for a programmer to expend that much effort. Borland has done nearly all the work for you. Using the `TScreen` class you can instantly tell which form and control are active. Also, you can be notified, using events in `TScreen`, when the active form or control changes.

If you add in a couple of events from the `TApplication` class, you can also tell when the application is activated or deactivated.

Silver TScreen

The code in **Listing A** is taken from a demonstration program I wrote. You can find this program and its complete source code on the Bridges Publishing Web site. **Figure A** shows the program's main form with the events that were logged.

Figure A



The demonstration program with events logged.

Let me explain how this code works. The program has a main form with a few controls; including a button that will create child forms that also have a few controls. Whenever the active control or form changes, the program writes an entry into a `TMemo` object that I have named `LogMemo`. If the program is activated or deactivated, a line will be written to `LogMemo` indicating which of those events happened.

Writing to LogMemo is accomplished with the simple function Log(). This function could be easily changed to write information to a file for a more permanent record of events.

Whenever you build a program using the VCL, the VCL creates a TApplication object and assigns a pointer to that object to the global variable named Application. You probably knew this. What you might not know is that a TScreen object is also created and a pointer to this object is assigned to the global variable Screen. I'll use events in TApplication and TScreen and properties in TScreen to get all the information I need to log events.

Take 1, TApplication

Lets take a look at TApplication first. TApplication has two events, OnActivate and OnDeactivate. These events do exactly what you would expect. They fire when the application is either activated or deactivated. I wrote two functions, ActivateApplication() and DeactivateApplication(), to be assigned to these events in the main form's constructor. These two functions merely call Log() with the text that indicates what happened.

Take 2, TScreen

Using TScreen is also very simple. TScreen contains two events that I used, OnActiveFormChange and OnActiveControlChange. The OnActiveFormChange event fires whenever the active form in the application changes. Similarly, OnActiveControlChange fires when the active control changes. I assigned the functions FormChange() and ControlChange() to these two events in the main form's constructor.

In the FormChange() function, I use the ActiveForm property of TScreen to determine which form is active and call the Log() function with a string containing the Caption property of that form.

The ControlChange() function uses the TScreen property, ActiveControl, to report a control change. I use the Name property of the active control to identify it in a call to the Log() function.

Cleanup

Notice what I do in the main form's destructor. I assign 0 to the TApplication and TScreen events. If I didn't, the program would crash. Both TApplication and TScreen are destroyed after the main form. So, when the main form is destroyed events that are connected to changes in controls and forms will fire. Since the code that these events try to use is contained in the main form, which no longer exists, bad things can happen.

That’s a wrap

I am sure you have run into `TApplication` in the past. You may not be as familiar with `TScreen`. I have just touched on one small use for `TScreen`. There is a lot of good information and useful functionality in this VCL class. Take a look at the online help to see what else you can do with `TScreen`.

Listing A: *Code for logging application, form and control events.*

```
__fastcall TMainForm::TMainForm(
  TComponent* Owner) : TForm(Owner)
{
  Application->OnActivate = ActivateApplication;
  Application->OnDeactivate =
    DeactivateApplication;

  Screen->OnActiveFormChange = FormChange;
  Screen->OnActiveControlChange = ControlChange;
}

__fastcall TMainForm::~TMainForm()
{
  Screen->OnActiveControlChange = 0;
  Screen->OnActiveFormChange = 0;

  Application->OnActivate = 0;
  Application->OnDeactivate = 0;
}

void __fastcall TMainForm::Log(String text)
{
  if (ShowTimeChk->Checked)
    text.Insert(Now().DateTimeString() +
      " - ", 1);
   LogMemo->Lines->Add(text);
}

void __fastcall TMainForm::FormChange(
  TObject *Sender)
{
  Log("Form: " + Screen->ActiveForm->Name + " - "
    + Screen->ActiveForm->Caption);
}
```

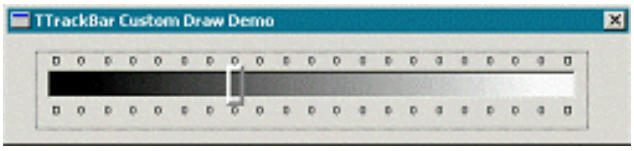
```
void __fastcall TMainForm::ControlChange(
  TObject *Sender)
{
  Log("Control: " + Screen->ActiveControl->Name);
}

void __fastcall TMainForm::ActivateApplication(
  TObject *Sender)
{
  Log("Activate - " + Application->Title);
}

void __fastcall TMainForm::DeactivateApplication(
  TObject *Sender)
{
  Log("Deactivate - " + Application->Title);
}
```

Custom drawing the trackbar control

by Damon Chandler

Without a doubt, the appearance of a stock trackbar control leaves much to be desired. In this article, I'll show you how to use the Custom Draw service to tailor the appearance of a trackbar control. **Figure A** depicts the trackbar that we'll create.

Figure A



A custom-drawn TTrackBar object.

An overview of the Custom Draw service

The Custom Draw service was first introduced with version 4.70 of Comctl32.dll as a simplified and modular way to customize the appearance of a common control. Back before the Custom Draw service was available, if you wanted to change the font or the color of, say, a tree-view control (on a per-item basis), you'd pretty much have to draw all of the items yourself. Fortunately, this hassle is no longer necessary. By using the Custom Draw service, you can tap into your control's drawing routine; this way, you can change the appearance of a certain element without the need to draw the entire control.
The Custom Draw service is available for header controls, list views, rebars, toolbars, tree-views, tooltips, and trackbars. In fact, if you have Borland C++Builder 4.0 and later, many of the common control-related components provide Custom Draw-specific events (e.g., `OnCustomDrawItem`). Unfortunately, the `TTrackBar` class isn't one of these.

Custom Draw for trackbars

I just mentioned that the Custom Draw service allows you to tap into a control's drawing routine. How exactly is this done? Well, like most everything in Windows, the Custom Draw service is presented via a series of messages; namely the `NM_CUSTOMDRAW` notification message. This notification is sent to the parent of the common control in the form of a `WM_NOTIFY` message. The first step, then, is to handle the `WM_NOTIFY` message. If your trackbar is parented directly to your main form, you can handle this message like so:

```
// in header...
class TForm1 : public TForm
{
__published:
  TTrackBar *TrackBar;
private:
  MESSAGE void __fastcall WMNotify(TMessage& Msg);
public:
  __fastcall TForm1(TComponent* Owner);
```

```
  BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
                WM_NOTIFY, TMessage, WMNotify)
  END_MESSAGE_MAP(TForm)
};

// in source...
void __fastcall TForm1::WMNotify(TMessage& Msg)
{
  // grab a pointer to NMHDR struct
  LPNMHDR pnmh = reinterpret_cast
                <LPNMHDR>(Msg.LParam);

  // if the notification message is
  // NM_CUSTOMDRAW and from our trackbar
  if (pnmh->code == NM_CUSTOMDRAW && pnmh->hwndFrom==TrackBar->Handle)
  {
    //
    // proceed with drawing...
    //
    return;
  }

  // pass on all other messages
  TForm::Dispatch(&Msg);
}
```

Within the block marked "proceed with drawing", the first thing you need to do is grab a pointer to a special structure called `NMCUSTOMDRAW`. The data members of this structure will provide you with vital information such as an identifier of the drawing stage (`dwDrawStage`); a handle to the device context to draw to (`hdc`); an identifier of the element (i.e., channel, thumb track, or tick marks) that's to be drawn (`dwItemSpec`); and the bounding rectangle of this element (`rc`). There are a few more data members, but these are the main ones.

The `NMCUSTOMDRAW::dwDrawStage` data member specifies the current drawing stage. This data member will be set to one of the following identifiers: `CDDS_PREPAINT`, `CDDS_ITEMPREPAINT`, `CDDS_POSTPAINT`, `CDDS_PREERASE`, `CDDS_POSTERASE`, `CDDS_ITEM`, `CDDS_ITEMPOSTPAINT`, `CDDS_ITEMPREERASE`, `CDDS_ITEMPOSTERASE`, `CDDS_SUBITEM`. The first two identifiers are our main interest.

When the `dwDrawStage` data member specifies `CDDS_PREPAINT`, this is your cue that the trackbar is about to draw itself. How you respond to this cue (i.e., what you set `Msg.Result` to) will govern whether or not you'll receive further Custom Draw-related notifications. In most cases, you'll want to reply with `CDRF_NOTIFYITEMDRAW`, which tells the trackbar that it should notify you before it draws each of its elements (channel, thumb track, or tick marks). This elemental notification is sent with the `dwDrawStage` data member set to `CDDS_ITEMPREPAINT`.

So, within the `WMNotify` member function, here's what we have so far:

```
// other code from before...
if (pnmh->code == NM_CUSTOMDRAW && pnmh->hwndFrom == TrackBar->Handle)
{
```

```
  // grab a pointer to NMCUSTOMDRAW
  LPNMCUSTOMDRAW pDraw = reinterpret_cast<LPNMCUSTOMDRAW> (Msg.LParam);

  // test the drawing stage...
  switch (pDraw->dwDrawStage)
  {
  // before anything is drawn...
  case CDDS_PREPAINT:
    {
      //
      // tell the trackbar to notify us
      // before it draws its elements
      //
      Msg.Result = CDRF_NOTIFYITEMDRAW;
      return;
    }

    // before an element is drawn...
  case CDDS_ITEMPREPAINT:
    {
      //
      // draw each element...
      //
      return;
    }
  }
}

// other code from before...
```

When the `dwDrawStage` data member is set to `CDDS_ITEMPREPAINT`, this indicates that the trackbar is about to draw each of its elements. In this case, you need to do one of two things. One choice is to draw the element manually and then respond by setting `Msg.Result` to `CDRF_SKIPDEFAULT`, which instructs the trackbar to skip its default drawing for that element. The other choice is that you not draw anything and simply respond with `CDRF_DODEFAULT`, which tells the trackbar to draw the element as it normally would. You can see here how the modular design of the Custom Draw service simplifies things a great deal: you can selectively choose which elements you want to draw, punting the work to the trackbar control if you don’t need to customize a certain aspect.
Again, the `CDDS_ITEMPREPAINT` is your cue that an element is about to be drawn. How do you determine which element this cue refers to? This is where the `NMCUSTOMDRAW::dwItemSpec` data member comes into play. This data member will be set to one of the following: `TBCD_CHANNEL` (for the channel), `TBCD_THUMB` (for the thumb track), or `TBCD_TICS` (for the tick marks).

Drawing the channel

When the NMCUSTOMDRAW::dwDrawStage data member indicates CDDS_ITEMPREPAINT and the dwItemSpec data member indicates TBCD_CHANNEL, it's time to draw the channel. Remember, if you don't need to customize the channel, you can always respond with CDRF_DODEFAULT to let the trackbar handle this task. Here's a simple example of drawing a background image to the channel:

```
// before an element is drawn...
case CDDS_ITEMPREPAINT:
{
  // if we're drawing the channel
  if (pDraw->dwItemSpec==TBCD_CHANNEL)
  {
    // #include <memory> for auto_ptr
    std::auto_ptr<TCanvas>
    tbCanvas(new TCanvas());
    TRect rect = pDraw->rc;

    // render the background image
    tbCanvas->Handle = pDraw->hdc;
    tbCanvas->StretchDraw(rect,
    Image1->Picture->Graphic);
    tbCanvas->Handle = NULL;

    // draw the channel's edge
    DrawEdge(pDraw->hdc, &pDraw->rc,
    EDGE_SUNKEN, BF_RECT);

   // tell the trackbar we
   // drew the channel manually
   Msg.Result = CDRF_SKIPDEFAULT;
   return;
  }
}
```

Drawing the thumb track

When the NMCUSTOMDRAW::dwDrawStage data member indicates CDDS_ITEMPREPAINT and the dwItemSpec data member indicates TBCD_THUMB, it's time to draw the thumb track (or punt with CDRF_DODEFAULT). Here we'll use the Frame3D() VCL function:

```
// before an element is drawn...
case CDDS_ITEMPREPAINT:
{
  // channel code from before...

  // if we're drawing the thumb
```

```
  if (pDraw->dwItemSpec == TBCD_THUMB)
  {
  // #include <memory> for auto_ptr
  std::auto_ptr<TCanvas>
  tbCanvas(new TCanvas());
  TRect rect = pDraw->rc;

  // render the background image
  tbCanvas->Handle = pDraw->hdc;
  Frame3D(tbCanvas.get(),
  rect, clWhite, clGray, 3);
  tbCanvas->Handle = NULL;

  // tell the trackbar we
  // drew the thumb manually
  Msg.Result = CDRF_SKIPDEFAULT;
  return;
}
```

Drawing the tick marks

When the `NMCUSTOMDRAW::dwDrawStage` data member indicates `CDDS_ITEMPREPAINT` and the `dwItemSpec` data member indicates `TBCD_TICS`, it's time to draw the tick marks (or punt). Here's we'll use a series of ellipses and rectangles. (Note that this code works only for horizontal trackbars; you'll have to swap the horizontal and vertical coordinates for vertical trackbars.):

```
// if we're drawing the ticks
if (pDraw->dwItemSpec == TBCD_TICS)
{
  // determine the thumb dimensions
  RECT RThumb = {0};
  SNDMSG(TrackBar->Handle,
         TBM_GETTHUMBRECT, 0,
         reinterpret_cast<LPARAM>(&RThumb));

  // determine the number of ticks
  const int num_ticks =
  SNDMSG(TrackBar->Handle,
         TBM_GETNUMTICS, 0, 0) - 2;

  // draw the middle ticks
  for (int iTick = 0; iTick < num_ticks;
      ++iTick)
  {
```

```
    const int x_pos =
    SNDMSG(TrackBar->Handle,
           TBM_GETTICPOS, iTick, 0);

    Ellipse(pDraw->hdc,
            x_pos - 2, RThumb.top - 6,
            x_pos + 2, RThumb.top);
    Ellipse(pDraw->hdc,
            x_pos - 2, RThumb.bottom,
            x_pos + 2, RThumb.bottom + 6);
  }

  // draw the first and last ticks
  RECT RChanl = {0};
  SNDMSG(TrackBar->Handle,
         TBM_GETCHANNELRECT, 0,
         reinterpret_cast<LPARAM>(&RChanl));
  InflateRect(&RChanl, -4, 0);
  Rectangle(pDraw->hdc,
            RChanl.left, RThumb.top - 6,
            RChanl.left + 4, RThumb.top);
  Rectangle(pDraw->hdc,
            RChanl.left, RThumb.bottom,
            RChanl.left + 4, RThumb.bottom + 6);
  Rectangle(pDraw->hdc,
            RChanl.right - 4, RThumb.top - 6,
            RChanl.right, RThumb.top);
  Rectangle(pDraw->hdc,
            RChanl.right - 4, RThumb.bottom,
            RChanl.right, RThumb.bottom + 6);

  // tell the trackbar we
  // drew the ticks manually
  Msg.Result = CDRF_SKIPDEFAULT;
  return;
}
```

Handling the CN_NOTIFY message

Earlier, I mentioned that a control always sends the `WM_NOTIFY` message to its parent window. This actually poses a bit of a problem because our `WMNotify` member function will work only if the `TTrackBar` object is placed directly on `Form1`. If you place your trackbar on a `TPanel` object, for example, the trackbar's parent is now the panel. This means that all of your trackbar's `WM_NOTIFY` messages won't be sent to your form, but instead to the panel.

You could work around this problem by subclassing the panel, and then handling the `WM_NOTIFY` message from within the subclass procedure. A better approach, however, is to create a `TTrackBar` descendant class and handle the `CN_NOTIFY` VCL message. This message is a reflected version of the `WM_NOTIFY` message that's sent back to the trackbar itself. (See the sample project at **www.residorph.com** for an example of handling this message.)

Conclusion

Perhaps the `TTrackBar` class in the next version of C++Builder will have built-in support for the Custom Draw service. For now, we have to resort to handling the notification messages manually. I've shown you how to do this with the trackbar control; now go out and try this with the other common controls that you need to customize. (For more information on the Custom Draw service, see: **http://msdn.microsoft.com/library/psdk/shellcc/commctls/CustDraw/CustDraw.htm**.)

Folder change notification

by Mark G. Wiseman

I'm going to show you how to do something with C++Builder that sounds very, very complicated, but turns out to be really easy. I'm going to show you how to create a VCL component that uses multithreading to monitor changes to the files and subfolders of folder on a hard drive. The component itself requires less than two hundred lines of code and the C++Builder IDE will even generate some of that for you.

Listings A and B contain the source code for the component that I named `TChangeNotification`. This source code and the source for a demonstration program can be found on the Bridges Publishing Web site at www.bridgespublishing.com.

Monitoring folder changes

The Windows API contains a set of functions that allow a programmer to monitor the file system for changes. These functions are `FindFirstChangeNotification()`, `FindNextChangeNotification()` and `FindCloseChangeNotification()`.

Let's take a look at `FindFirstChangeNotification()` first. Here is the prototype:

```
HANDLE FindFirstChangeNotification
(
  LPCTSTR lpPathName,
  BOOL bWatchSubtree,
  DWORD dwNotifyFilter
);
```

The first argument to this function, `lpPathName`, is the path of the folder you want to monitor for changes, e.g. `"c:\\WinNT\\System"`.

If the second parameter, `bWatchSubtree`, is `true`, not only will the folder be monitored, but all its subfolders will also be monitored.

The third parameter, `dwNotifyFilter`, is a flag that tells Windows what kinds of changes you want to monitor. You can monitor changes to file names, changes to file size and attributes, last-write and last-access times and changes to file security settings. Take a look at the Windows API online help to see all the values you can use for this flag. I used the `FILE_NOTIFY_CHANGE_FILE_NAME` and `FILE_NOTIFY_CHANGE_DIR_NAME` flags for the component. Using these flags, the component will be notified whenever a file or folder is created, deleted, or renamed.

If the call to `FindFirstChangeNotification()` is successful, it returns a `HANDLE` to a find-change-notification object. The easiest way to monitor this object is by using one of the Windows API "Wait Functions". I used the `WaitForSingleObject()` function.

The `WaitForSingleObject()` function simply stops everything and waits for the find-change-notification object to signal a change. The easiest way to use this function is to wrap it in a separate thread. I'll discuss how to do this a little later. This function takes two parameters, the handle returned by `FindFirstChangeNotification()` and a time-out interval. Since I don't want the component to time out, I set the interval value to `INFINITE`.

When the find-change-notification object signals a change, `WaitForSingleObject()` returns with a `DWORD` return value equal to `WAIT_OBJECT_0`.

Once `WaitForSingleObject()` returns, I call `FindNextChangeNotification()`, passing in the handle to the find-change-notification object as its only parameter. If this function call is successful, I again call `WaitForSingleObject()` to continue monitoring.

When I want to stop the monitoring, I need to release the find-change-notification object by calling `FindCloseChangeNotification()` and passing in the handle to the object.

Using TCNThread

Since the `WaitForSingleObject()` function stops and waits for a signal from the find-change-notification object, you can't call it in your program's main thread. If you did, the program would freeze until there was a change in the folder being monitored.
This is why I used a separate thread, `TCNThread`, derived from the `TThread` class in the VCL. Since `TCNThread` has only one very specific function, I have embedded its class definition inside the class definition for `TChangeNotification`. `TCNThread` gets all the data it needs to work from the `TChangeNotification` component that owns it.
There really isn't a lot of coding involved in creating `TCNThread`. As required for all classes derived from `TThread`, I have written an `Execute()` function. Let's take a look at this function.
The `Execute()` function performs the tasks I described in the section above. It creates a find-change-notification object and then enters a `while` loop in which it waits for that object to signal a change.
When a change occurs, `Execute()` notifies the `TChangeNotification` component that owns its `TCNThread` by calling the `Notify()` function. `Execute()` then calls `FindNextChangeNotification()` and cycles back through the `while` loop to wait for another signal.
The `while` loop will exit when the `TCNThread` is terminated. At this point `Execute()` calls `FindCloseChangeNotification()` to free the find-change-notification object.
If there should be an error, `Execute()` notifies its parent component by calling the `Error()` function and then terminating the `TCNThread`. I use the `Error()` function instead of a C++ exception, because throwing an exception out of a thread will crash a program.
If you've looked at the `Execute()` function in `TCNThread` source code, then you might have noticed that the calls to the `Notify()` and `Error()` functions are wrapped in a call to `Synchronize()`. It is usually not safe to access VCL components from a thread other than the main VCL thread. The `Synchronize()` function forces `Notify()` and `Error()` to execute in the main VCL thread.

The TChangeNotification component

The `TChangeNotification` class encapsulates everything in an easy to use component.
I started by letting the C++Builder IDE generate the skeletal code for `TChangeNotification`. I did this by choosing File | New… from the IDE menu and then selecting Component from the New tab of the resulting dialog box. I filled in the blanks of the new component wizard and clicked the OK button.
`TchangeNotification`, which is derived from `TComponent`, is a very simple component. In addition to a constructor and destructor, the public interface for this component has two functions, `Start()` and `Stop()`, and four properties, `OnChange`, `OnError`, `Folder` and `WatchSubFolders`.
Let's take a look at how the component works, starting with the `Start()` and `Stop()` functions. The `Start()` function creates a new `TCNThread` if one doesn't already exist and the `Stop()` function deletes the `TCNThread` and zeroes out the pointer to it.
All the data that the `TCNThread` needs to run is obtained from data members of the component. These data members are set and read through the four published properties of the component.
The `Folder` property corresponds to the private data member `folder`, an `AnsiString` that contains the path of the folder to monitor. The `WatchSubFolders` property is a `bool` that determines whether the folders subfolders are also monitored.
These two properties, `Folder` and `WatchSubFolders` use functions to set the values of the underlying data members, `folder` and `watchSubFolders`. These functions are `SetFolder()` and `SetWatchSubFolders()`. The important thing to notice about these two functions is how they handle a property change while the `TCNThread` is running. Since the values from these two properties are used in the call to `FindFirstChangeNotification()`, the running thread must be terminated and deleted, and then a new thread must be created. This is accomplished by wrapping the change in the data members with calls to `Stop()` and `Start()`.
The properties, `OnNotify` and `OnError`, are for programmer defined events that occur when a change in the folder is signaled or when an error occurs within the `TCNThread`. These two properties also use functions to set the values of

the underlying data members, `onNotify` and `onError`.
It is not necessary to delete a running thread and then create a new one when changing the values of `OnNotify` and `OnError`. However, you don't want `TCNThread` calling one of these events while it is being change. To avoid this, I have wrapped the change in data members with calls to the `Suspend()` and `Resume()` functions of `TCNThread`.
That's really all there is to creating the `TChangeNotification component`. Now let's take a quick look at how to use it.

Using TChangeNotification

Once you have created the `TChangeNotification` component you will need to install the component into your IDE's component palette. Using the component is very easy.
Just drop the `TChangeNotification` component on a form and assign a path to the folder property in the Object Inspector of the IDE. Next, switch to the Events tab of the Object Inspector and double-click on the `OnChange` event. Fill in the code you want to execute when a folder change is signaled and you're ready to go.
You will need to call the `Start()` function of the component to actually begin monitoring folder changes.
Although it is not necessary, I would recommend that you also assign some error handling code to the `OnError` event. If you don't deal with an error, the component stops its thread and will not report folder changes. The most common error you are likely to run into is passing in a non-existent folder to be monitored.

Conclusion

Look at the demo program to see how I used `TChangeNotificaiton`. The program allows the user to set a folder to monitor and to specify whether its subfolders should also be monitored. It allows the user to start and stop the monitoring and logs any folder changes to a `TEdit` control.
Unfortunately, the find-change-notification object only tells you that a change occurred in the folder. It does not tell you what that change was. If you need to know more, you will have to do a before and after analysis of the folder.
Finally, I think this component is fairly complete, but there are two improvements you might want to consider.
First, you could add a custom editor to the `Folder` property to let the programmer browse for the folder to be monitored. This may or may not be that useful depending on how much you know about the directory structure of the computer running the program.
Second, you might want to consider adding a property to let the programmer select the flags for the value of parameter, `dwNotifyFilter,` which is passed into the call to `FindFirstChangeNotification()`. This way you could use the component to monitor other changes in the file system, for example changes to file size.

Listing A: *Header file for theTChangeNotificion component.*

```
#ifndef ChangeNotificationH
#define ChangeNotificationH

#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>


class PACKAGE TChangeNotification :
public TComponent
{
  __published:
```

```
    __property TNotifyEvent OnChange = {read = onChange, write = SetOnChange};
    __property TNotifyEvent OnError = {read = onError, write = SetOnError};
    __property bool WatchSubFolders = {read = watchSubFolders, write =
SetWatchSubFolders};
    __property String Folder = {read = folder, write = folder};

  public:
    __fastcall TChangeNotification(TComponent *owner);
    __fastcall ~TChangeNotification();

    void __fastcall Start();
    void __fastcall Stop();

  private:
    void __fastcall SetOnChange(TNotifyEvent event);
    void __fastcall SetOnError(TNotifyEvent event);
    void __fastcall SetWatchSubFolders(bool watch);
    void __fastcall SetFolder(String aFolder);

    TNotifyEvent onChange;
    TNotifyEvent onError;
    bool watchSubFolders;
    String folder;

  class TCNThread : public TThread
  {
    friend TChangeNotification;

  public:
    __fastcall TCNThread(TChangeNotification *owner);
    __fastcall ~TCNThread();

    void __fastcall Execute();
    void __fastcall Notify();
    void __fastcall Error();

  private:
    TChangeNotification *owner;
    HANDLE handle;
  } *thread;
};

#endif // ChangeNotificationH
```

Listing B: *Source file for theTChangeNotificion component.*

```
#include <vcl.h>
#pragma hdrstop

#include "ChangeNotification.h"
```

```
__fastcall
TChangeNotification::TChangeNotification(TComponent *owner) : TComponent(owner)
{
  onChange = 0;
  onError = 0;
  thread = 0;
  watchSubFolders = false;
}

__fastcall
TChangeNotification::~TChangeNotification()
{
  delete thread;
}

void __fastcall TChangeNotification::Start()
{
  if (thread == 0) thread = new TCNThread(this);
}

void __fastcall TChangeNotification::Stop()
{
  delete thread;
  thread = 0;
}

void __fastcall TChangeNotification::SetFolder(String aFolder)
{
  if (thread != 0) {
    Stop();
    folder = aFolder;
    Start();
  } else
    folder = aFolder;
}

void __fastcall TChangeNotification::SetOnChange(TNotifyEvent event)
{
  if (thread != 0) {
    thread->Suspend();
    onChange = event;
    thread->Resume();
  } else
    onChange = event;
}

void __fastcall TChangeNotification::SetOnError(TNotifyEvent event)
{
  if (thread != 0) {
    thread->Suspend();
```

```
    onError = event;
    thread->Resume();
  } else
    onError = event;
}

void __fastcall
TChangeNotification::SetWatchSubFolders(bool watch)
{
  if (thread != 0) {
    Stop();
    watchSubFolders = watch;
    Start();
  } else
    watchSubFolders = watch;
}


// Thread ----------------------------------

__fastcall TChangeNotification::TCNThread::
TCNThread(TChangeNotification *owner)
: TThread(false), owner(owner)
{
  handle = 0;
}

__fastcall
TChangeNotification::TCNThread::~TCNThread()
{
  Terminate();

  if (handle != 0) {
    FindCloseChangeNotification(handle);
    handle = 0;
  }
}

void __fastcall
TChangeNotification::TCNThread::Execute()
{
  handle = FindFirstChangeNotification(owner->Folder.c_str(),
                                       owner->WatchSubFolders,
                                       FILE_NOTIFY_CHANGE_FILE_NAME);

  if (handle == INVALID_HANDLE_VALUE) {
    Synchronize(Error);
    Terminate();
  }
```

```
  while (!Terminated) {
    DWORD status =
    WaitForSingleObject(handle, INFINITE);

    if (status == WAIT_OBJECT_0
        && Terminated == false) {
      Synchronize(Notify);
      if (FindNextChangeNotification(handle) == 0) {
        Synchronize(Error);
        Terminate();
      }
    } else
      Terminate();
  }
}

void __fastcall
TChangeNotification::TCNThread::Notify(void)
{
  if (owner->OnChange) owner->OnChange(owner);
}

void __fastcall
TChangeNotification::TCNThread::Error(void)
{
  if (owner->OnError) owner->OnError(owner);
}


// -----------------------------------------

static inline void ValidCtrCheck(TChangeNotification *)
{
  new TChangeNotification(NULL);
}


#pragma package(smart_init)
```

Printing Bitmaps III: Wrapping Things Up

by Damon Chandler

In the first article of this series, I presented a conceptual overview of three types of bitmaps: DDBs, DIBs, and DIB section bitmaps. Last month, we examined how to use the `StretchDIBits()` function to send a DIB to the printer. This time, we'll wrap all of the printing code into a reusable class, and I'll discuss a technique called "banding" which can enhance the reliability of the `StretchDIBits()` function.

The TBitmapPrinter class

In the past two articles, we've covered a fair bit of conceptual information and code. Now it's time to consolidate these ideas and procedures into a class; we'll call this class `TBitmapPrinter`. Before we get our hands dirty with the implementation, however, let me present a short code snippet that demonstrates what this class should be able to do once it's complete:

```
#include <printers.hpp>
#include <memory>

// create a TBitmapPrinter object
std::auto_ptr<TBitmapPrinter> BitmapPrinter(new TBitmapPrinter());

// (1) set the Bitmap
BitmapPrinter->Bitmap = Image1->Picture->Bitmap;

// (2) set the target rectangle
BitmapPrinter->PrintRect = Rect(0, 0,
                                Printer()->PageWidth,
                                Printer()->PageHeight);

// (3) print the image
Printer()->BeginDoc();
BitmapPrinter->PrintBitmap(Printer()->Canvas->Handle);
Printer()->EndDoc();
```

There are three main steps in this code snippet: (1) specifying the bitmap to be printed; (2) specifying the location and dimensions on the page where this bitmap should be printed; and (3) specifying the device context of the device with which this bitmap should be printed.

We've actually discussed all of the code necessary to accomplishing these three steps, so we can move through the implementation fairly quickly. **Listing A** shows the declaration of the `TBitmapPrinter` class. Here we'll focus on implementing the `TBitmapPrinter::DoSetBitmap()` and `DoInternalPrintBitmap()` member functions.

Setting the Bitmap

From **Listing A**, you can see that assigning a `TBitmap` object to the `TBitmapPrinter::Bitmap` property invokes the `DoSetBitmap()` member function. The `DoSetBitmap()` member function will do two things:

it will "create" a DIB from the specified `TBitmap` object, storing the results in the private `pDIB_` and `pBits_` members; and, it will create a copy of the `Bitmap's` palette, storing the result in the private `hPal_` member. Here's the code:

```
void __fastcall
TBitmapPrinter::DoSetBitmap(const Graphics::TBitmap* Bitmap)
{
  DeleteObject(hPal_);
  hPal_ = NULL;

  delete [] pBits_;
  pBits_ = NULL;

  delete [] reinterpret_cast<unsigned char*>(pDIB_);
  pDIB_ = NULL;

  if (Bitmap)
  {
    pDIB_ =
    DoCreateDIB(*Bitmap, pBits_);
    hPal_ = CopyPalette(const_cast<Graphics::TBitmap*>
                        (Bitmap)->Palette);
  }
}
```

Simple enough, right? Actually, most of the work is delegated to the `DoCreateDIB()` member function, which is identical to the `CreateDIB()` function that was presented in the last article. (I won't repeat this function here, but you can see the implementation in the source code. If you don't have last month's issue, see **www.bridgespublishing.com**.) The `CopyPalette()` function is a VCL utility function that's declared in graphics.hpp. As its name suggest, this function simply creates a copy of a logical palette.

Printing the Bitmap

Now that we have a DIB (`pDIB_` and `pBits_`), let's implement the code to print it. From the first code snippet, you can see that this uses the `TBitmapPrinter::PrintBitmap()` member function. This member function simply calls the `DoPrintBitmap()` member function, which is defined as follows:

```
void __fastcall TBitmapPrinter::
DoPrintBitmap(HDC hPrnDC)
{
  if (!pDIB_)
  {
    throw EBitmapPrinterError("No bitmap defined");
  }
  if (!hPrnDC)
  {
    throw EBitmapPrinterError("Invalid printer DC");
```

```
  }

  // compute the rectangle to print to
  DoCalculatePrintRect(hPrnDC);

  // print the bitmap
  DoInternalPrintBitmap(hPrnDC);
}
```

As with the `DoSetBitmap()` member function, the `DoPrintBitmap()` member function punts most of its work elsewhere. The `DoCalculatePrintRect()` member function simply computes the dimensions of the rectangle to print to—by using the `GetDeviceCaps()` function—and then stores the result in the `RPrint_` member. The `DoInternalPrintBitmap()` member function is the workhorse of the class. It's from within this member function that the DIB is printed via the `StretchDIBits()` function.

The DoInternalPrintBitmap function, version 1

We discussed the specifics of the `StretchDIBits()` function last time, so you should already have an idea of how to define the `DoInternalPrintBitmap()` member function. Here's what one version of the function might look like:

```
void __fastcall TBitmapPrinter::
DoInternalPrintBitmap(HDC hPrnDC)
{
  if (IsRectEmpty(&RPrint_))
  {
    throw EBitmapPrinterError("Invalid target rectangle");
  }

  if (pDIB_ && pBits_)
  {
    //
    // extract the width and the
    // height from the DIB's header
    //
    const int dib_width =
    pDIB_->bmiHeader.biWidth;
    const int dib_height =
    pDIB_->bmiHeader.biHeight;

    // support palettized printers
    bool palettized = GetDeviceCaps(hPrnDC, RASTERCAPS) & RC_PALETTE;
    if (palettized)
    {
      hPal_ = SelectPalette(hPrnDC, hPal_, FALSE);
      RealizePalette(hPrnDC);
```

```
  }

  //
  // set the stretching mode
  // to preserve colors
  //
  SetStretchBltMode(hPrnDC, COLORONCOLOR);

  // render the DIB
  bool result = GDI_ERROR !=
                StretchDIBits(
                              // draw to the printer's DC
                              hPrnDC,
                              // dest rect
                              RPrint_.left, RPrint_.top,
                              RPrint_.right - RPrint_.left,
                              RPrint_.bottom - RPrint_.top,
                              // source rect (entire DIB)
                              0, 0, dib_width, dib_height,
                              // source bits
                              pBits_,
                              // source DIB
                              pDIB_,
                              //color table contains RGBQUADs
                              DIB_RGB_COLORS,
                              // copy source
                              SRCCOPY
                              );

  // restore the original palette
  if (palettized)
  {
    hPal_ = SelectPalette(hPrnDC, hPal_, TRUE);
  }

  // report errors
  if (!result)
  {
    throw EBitmapPrinterError("Error printing DIB");
  }
}
// report errors
else
{
  throw EBitmapPrinterError("Error getting DIB info");
}
```

```
}
```

Notice that this implementation is virtually identical to the code related to `StretchDIBits()` that I presented last time. And, as I mentioned last time, this approach will work on most printers. However, due to memory limitations on Windows-9x/Me-based systems, there is a chance that the `StretchDIBits()` function will fail. For example, if the target device is, say, a poster printer, the dimensions of `RPrint_` may be extremely large. In this case, you'd be asking the `StretchDIBits()` function to affect a huge stretch, which can be a recipe for disaster. Let's see how we can work around this problem by using banding.

The DoInternalPrintBitmap function, version 2

Banding is a technique of dividing an image into a series of strips or "bands", and then rendering each strip separately to draw the entire image. Banding is especially useful for drawing large images because you can limit the amount of stretching (and thus, memory) that's required for each call to the `StretchDIBits()` function.

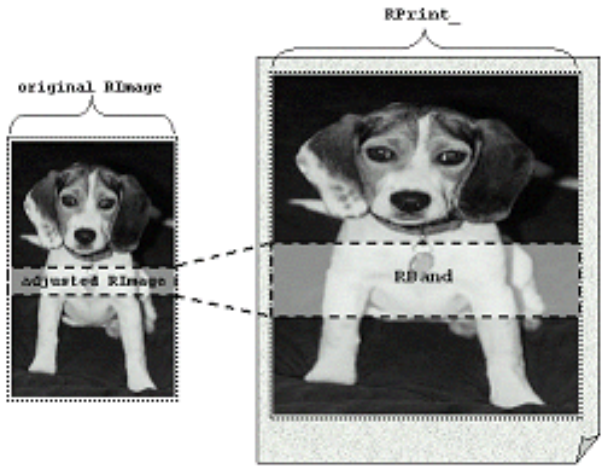
The trickiest part of banding is computing which strip of the DIB to render. This chore is delegated to the `TBitmapPrinter::DoCalculateBand()` member function, which is defined as follows:

```
bool __fastcall TBitmapPrinter::
DoCalculateBand(RECT& RBand, RECT& RImage)
{
  // compute the printable band area
  if (IntersectRect(&RBand, &RBand, &RPrint_))
  {
    //
    // compute the ratio of the print
    // width to the image width
    //
    const double ratioX =
    static_cast<double>
    (RPrint_.right - RPrint_.left) /
    static_cast<double>
    (RImage.right - RImage.left);

    //
    // compute the ratio of the print
    // height to the image height
    //
    const double ratioY =
    static_cast<double>
    (RPrint_.bottom - RPrint_.top) /
    static_cast<double>
    (RImage.bottom - RImage.top);

    //
    // compute the scaled image band
```

```
    // (this will tell us where in the
    // image to blit from)...
    //
    RImage.left = static_cast<int>(
                  0.5 + (RBand.left - RPrint_.left)/ ratioX);

    RImage.top = static_cast<int>(
                  0.5 + (RBand.top - RPrint_.top) /  ratioY);

                RImage.right = RImage.left + static_cast<int>(
                  0.5 + (RBand.right - RBand.left) / ratioX);

    RImage.bottom = RImage.top + static_cast<int>(
                  0.5 + (RBand.bottom - RBand.top) / ratioY);

    return true;
  }
  return false;
}
```

The `DoComputeBand()` member function takes two `RECT`-type parameters: `RBand` and `RImage`. The `RBand` parameter specifies a strip of the page to which a corresponding strip of the image should be printed. The `RImage` parameter specifies the dimensions of the image. These dimensions are used with the `RPrint_` member to compute horizontal and vertical scaling factors. These scaling factors are then used with `RBand` and `RPrint_` to adjust `RImage` so that it specifies the strip of the image that should be printed (to `RBand`). This scheme is illustrated in **Figure A**.

Figure A



The TBitmapPrinter::DoComputeBand() member function clips RBand to RPrint_, and it adjusts RImage to specify the strip of the image that corresponds to RBand.

So, by using the `DoComputeBand()` member function we'll know which part of the DIB to render. But, how do we know which part of the page to render this strip to? That is, how do we compute `RBand`? Well, one technique is to ask the printer driver for this information by using the `NEXTBAND` escape sequence, like so:

```
RECT RBand = {0};
//
// ask the printer driver
// to initialize RBand...
//
while (Escape(hPrnDC, NEXTBAND, 0,
              NULL, &RBand))
{
  RECT RImage =
  {0, 0, dib_width, dib_height};
  DoComputeBand(RBand, RImage);

  //
  // StretchDIBits() to hPrnDC...
  //
}
```

unfortunately, the vast majority of printer drivers will initialize `RBand` with the dimensions of the entire page, which obviates the role of our banding code. In other words, if `RBand` specifies the entire page, then `RImage` will indicate the dimensions of the entire image even after the call to `DoComputeBand()`.

To work around this potential problem, let's help the printer driver out a bit. Specifically, we can use the `NEXTBAND` escape sequence to ask the printer driver to initialize `RBand`, and then we can divide this band (which might indicate the entire page) into a series of sub-bands. This way, we can ensure that our banding code will be used, regardless of how the printer driver initializes `RBand`. This scheme is demonstrated in the following definition of the `DoInternalPrintBitmap()` member function:

```
void __fastcall TBitmapPrinter::
DoInternalPrintBitmap(HDC hPrnDC)
{
  if (IsRectEmpty(&RPrint_))
  {
    throw EBitmapPrinterError("Invalid target rectangle");
  }

  if (pDIB_ && pBits_)
  {
    //
    // extract the width and the
    // height from the DIB's header
    //
    const int dib_width =
    pDIB_->bmiHeader.biWidth;
    const int dib_height =
    pDIB_->bmiHeader.biHeight;
```

```
//
// query the printer for NEXTBAND
// capability (not really needed,
// better to be safe than sorry)
//
const int query_cap = NEXTBAND;
int do_bands =
Escape(
      hPrnDC, QUERYESCSUPPORT,
      sizeof(int), reinterpret_cast
      <LPCSTR>(&query_cap),
      NULL
      );

bool result = true;
if (do_bands)
{
  RECT RBand = {0};
  //
  // ask the printer driver
  // to initialize RBand...
  //
  while (Escape(hPrnDC, NEXTBAND, 0,
                NULL, &RBand))
  {
    if (IsRectEmpty(&RBand)) break;

    // limit the band height to 75
    const int band_height = min(75L, RBand.bottom - RBand.top);

    // compute the number of bands
    const int num_bands = 0.5 +
                          (RBand.bottom - RBand.top) /
                          static_cast<double> (band_height);

    // for each band...
    for (int iBand = 0;
        iBand < num_bands; ++iBand)
    {
      RBand.top =
      iBand * band_height;
      RBand.bottom =
      RBand.top + band_height;

      RECT RImage = {
        0,0,dib_width, dib_height};
```

```
if (DoCalculateBand(RBand, RImage))
{
  //
  // set the stretching mode
  // to preserve colors
  //
  SetStretchBltMode(hPrnDC, COLORONCOLOR);

  // for palettized printers
  bool palettized =
  GetDeviceCaps(hPrnDC, RASTERCAPS) & RC_PALETTE;
  if (palettized)
  {
    hPal_ = SelectPalette(hPrnDC, hPal_, FALSE);
    RealizePalette(hPrnDC);
  }

  // extract the dimensions
  const int dst_width =
  RBand.right - RBand.left;
  const int dst_height =
  RBand.bottom - RBand.top;
  const int src_width =
  RImage.right -RImage.left;
  const int src_height =
  RImage.bottom -RImage.top;

  // render the DIB
  result &= GDI_ERROR !=
            StretchDIBits(
                        // printer's DC
                        hPrnDC,
                        // dest rect
                        RBand.left, RBand.top,
                        dst_width, dst_height,
                        // source rect
                        RImage.left, dib_height-
                        RImage.bottom,
                        src_width, src_height,
                        // source bits
                        pBits_,
                        // source DIB
                        pDIB_,
                        //c.t. contains RGBQUADs
                        DIB_RGB_COLORS,
```

```
                                    // copy source
                                    SRCCOPY
                                    );

            // restore the old palette
            if (palettized)
            {
              hPal_ = SelectPalette(hPrnDC, hPal_, TRUE);
            }
          }
        }
      }
    } else
    { // no banding
      //
      // other code from
      // DoInternalPrintBitmap()
      // version 1...
      //
    }

    // report errors
    if (!result)
    {
      throw EBitmapPrinterError("Error printing DIB");
    }
  }
  // report errors
  else
  {
    throw EBitmapPrinterError("Error getting DIB info");
  }
}
```

There are a couple important things to note about this code snippet. First, the limit on the height of each band (which I fixed to 75 device units) is only a hypothetical number. You should examine both the height and the width of `RBand` to gauge the amount of stretching that's required; you can then decide whether or not you want to divide the band into sub-bands. Second, avoid using the sub-band division scheme on Windows NT/2000-based systems. I've found that the memory requirements of the spooler process increases in direct proportion to the number of sub-bands that you use. You can use the `GetVersionEx()` function (see **http://msdn.microsoft.com/library/psdk/sysmgmt/sysinfo_49iw.htm**) to determine the version of Windows on which your application is running.

Limitations of the TBitmapPrinter class

One of the reasons I declared most of the `TBitmapPrinter` class's member functions as virtual was to work

around its limitations. One limitation is that the class won't split the output across multiple pages. You'll need to implement this functionality manually by either dividing your `TBitmap` object beforehand (i.e., before assigning it to the `TBitmapPrinter::Bitmap` property), or by modifying the `DoPrintBitmap()` and/or `DoInternalPrintBitmap()` member functions.

Another limitation involves the use of 8-bpp bitmaps and the `GetDIBits()` GDI function. Specifically, on palettized displays, the `GetDIBits()` function—which is called from within the `GetDIB()` VCL function—incorrectly initializes the color table of the resultant DIB with entries of the default system palette (instead of with entries from the bitmap's palette). If you're using Borland C++Builder 3.0 or later, you can work around this bug by setting the `TBitmap::PixelFormat` property to `pf15bit` before assigning your `TBitmap` object to the `TBitmapPrinter::Bitmap` property. Alternatively (or if you're using Borland C++Builder 1.0), you can manually initialize the DIB's color table entries (recall the `BITMAPINFO::bmiColors` data member) with the entries of the bitmap's palette. To do this, you'll need to use the `GetPaletteEntries()` GDI function (see **http://msdn.microsoft.com/library/psdk/gdi/colors_1l4j.htm** and **http://www.deja.com/threadmsg_ct.xp?AN=498742760**). This latter method is actually preferable to adjusting the `PixelFormat` property because it avoids the overhead of creating a DIB of a greater color depth than is necessary.

Finally, in the rare case that the printer is palette-based, you need to ensure that you have a suitable palette. If your original bitmap uses eight (or less) bits per pixel, then the `TBitmap::Palette` property will indeed refer to a suitable logical palette; namely, the logical palette that's created from the color table of the BMP file. Recall, however, that a color table is optional for bitmaps of greater color depths. If you load, say, a 24-bpp bitmap from a file, chances are good that your `Bitmap's` palette will contain entries that correspond to the default system palette (a mere 20 colors). In this case, you need to use a color quantification strategy. See the following references for more information on color quantification: **www.acm.org/tog/GraphicsGems/** **or** **ftp://ftp.ledalite.com/pub/cquant97.bib**.

Conclusion

Although printing a bitmap is far from straightforward, there's certainly no black magic to it. Just keep the following in mind as you implement your printing code: (1) always use a DIB; (2) always use the `StretchDIBits()` function; and, (3) use banding when required. If your bitmap fails to print (and an exception wasn't thrown), the very first place you should look is the return value of the `StretchDIBits()` function.

You can download the source code for the `TBitmapPrinter` class along with a sample project from **www.bridgespublishing.com**. And, as always, if you have any questions, feel free to contact me at dmc27@ee.cornell.edu.

Listing A: ***The declaration of the TBitmapPrinter class***

```
//---------------------------------------
#ifndef BITMAP_PRINTER_H
#define BITMAP_PRINTER_H
//---------------------------------------
#include <vcl.h>
#include <stdexcept>
```

```
//---------------------------------------

typedef Exception EBitmapPrinterError;
enum TBitmapPrinterScale {bpsHorz, bpsVert, bpsPage, bpsCustom};

class TBitmapPrinter : public TObject
{
  public:
    __fastcall TBitmapPrinter();
    __fastcall ~TBitmapPrinter();

    // main printing member function
    void __fastcall PrintBitmap(HDC hPrinterDC)
    {DoPrintBitmap(hPrnDC);}

    // sets the TBitmap to print
    __property const Graphics::TBitmap*
    Bitmap = {write = DoSetBitmap};

    // gets/sets the target printing rect.
    __property RECT PrintRect = {read = RPrint_, write = DoSetRect};

    // gets/sets the scaling options
    __property TBitmapPrinterScale
    PrintScale = {read = scale_, write = DoSetScale};

  protected:
    // main printing member functions
    virtual void __fastcall DoPrintBitmap(HDC hPrnDC);
    virtual void __fastcall DoInternalPrintBitmap(HDC hPrnDC);

    // gets/sets class properties
    virtual void __fastcall DoSetBitmap(const Graphics::TBitmap* Bitmap);
    virtual void __fastcall DoSetPrintRect(RECT RPrint);
    virtual void __fastcall DoSetScale(TBitmapPrinterScale scale);

    // internal utility member functions
    virtual LPBITMAPINFO __fastcall DoCreateDIB(const Graphics::TBitmap&
      Bitmap, unsigned char*& pBits);
    virtual void __fastcall DoCalculatePrintRect(HDC hPrnDC);
    virtual bool __fastcall DoCalculateBand(RECT& RBand, RECT& RImage);

  private:
    // pointer to the DIB (header and CT)
    LPBITMAPINFO pDIB_;
    // pointer to the DIB's pixels
    unsigned char* pBits_;
```

```
    // handle to the bitmap's palette
    HPALETTE hPal_;
    // target printing rectangle
    RECT RPrint_;
    // scaling options
    TBitmapPrinterScale scale_;
};
//---------------------------------------
#endif // BITMAP_PRINTER_H
//---------------------------------------
```

Printing bitmaps II: sending a DIB to the printer

by Damon Chandler

Last time, we discussed how to generate a device-independent representation of a bitmap object—that is, how to create a device-independent bitmap (DIB). We also talked a little bit about the format of a DIB and some advantages and disadvantages of a DIB's device independence. Now that the conceptual groundwork is set, let's actually do some printing.

Transferring a DIB—the StretchDIBits() function

As I mentioned last time, the main advantage of a DIB's device independence is that the format of a DIB is well known. Why is this useful? Because the format is universal, you can safely transfer a DIB among different devices. The greatest disadvantage of using a DIB, however, is that you can't select a DIB into a memory device context, and so you can't use the `BitBlt()` or `StretchBlt()` functions to draw to a DIB, nor can you use these functions to draw a DIB to a device. However, there is an alternative method you can use to render a DIB to a device: the `StretchDIBits()` function can transfer a DIB to a graphics device. This function is similar to the `StretchBlt()` function, except whereas the latter accepts a handle to a (memory) device context for the source image, the former accepts a pointer to a `BITMAPINFO` structure. The following is the declaration of the `StretchDIBits()` function:

```
int StretchDIBits(
  // handle to the target DC
  HDC hdc,
  // destination rectangle
  int XDest,
  int YDest,
  int nDestWidth,
  int nDestHeight,
  // source rectangle
  int XSrc,
  int YSrc,
  int nSrcWidth,
  int nSrcHeight,
  // pointer to the DIB's pixels
```

```
 CONST VOID* lpBits,
 // pointer to the DIB's BITMAPINFO
 CONST BITMAPINFO* lpBitsInfo,
 // color-table format identifier
 UINT iUsage,
 // ternary ROP code
 DWORD dwRop
);
```

The `hdc` parameter specifies the destination device context—you use this parameter to tell the `StretchDIBits()` function which device you want the DIB transferred to.

The `XDest`, `YDest`, `nDestWidth`, and `nDestHeight` parameters specify, in logical coordinates, a bounding rectangle to which the DIB should stretch to fit. Likewise, the `XSrc`, `YSrc`, `nSrcWidth`, and `nSrcHeight` parameters specify, in pixels, the rectangular portion of the DIB that you want rendered.

The `lpBits` parameter specifies a pointer to the DIB's pixels and the `lpBitsInfo` parameter specifies the address of the DIB's corresponding `BITMAPINFO` structure.

The `iUsage` parameter specifies the type of color table that the DIB contains. Recall from last time that a DIB's color table can consist of either `RGBQUAD`-type entries or WORD-type entries. If the DIB's color table contains `RGBQUAD`s, you specify `iUsage` as `DIB_RGB_COLORS`. If the DIB's color table contains `WORD`s, you specify `iUsage` as `DIB_PAL_COLORS`. And, if the DIB has no color table, you can specify either identifier.

The `dwRop` parameter specifies an ROP3 code. In most cases, you'll want to set `dwRop` to `SRCCOPY`; this tells the `StretchDIBits()` function to simply copy the DIB to the target device, ignoring the colors of the destination and brush. (Note that many printers support only the `SRCCOPY` mode.)

If the `StretchDIBits()` function is successful, it returns a value that indicates the number of scan lines that were transferred to the device. If the function fails, it returns `GDI_ERROR`.

Using the StretchDIBits() function

Before we actually use the `StretchDIBits()` function to transfer a DIB to a printer, let's look at an example that demonstrates how to draw a DIB to the screen. The first step, of course, is to actually create the DIB. I'll repeat the code from last time for convenience, here, wrapped in a function called `CreateDIB()`:

```
LPBITMAPINFO CreateDIB(
   IN const Graphics::TBitmap& Bitmap,
```

```
   OUT unsigned char*& pBits
  )
{
  //
  // use the GetDIBSizes() function to
  // gauge the size of the DIB's parts
  //
  int header_ct_size = 0;
  int img_size = 0;
  GetDIBSizes(
    const_cast<Graphics::TBitmap&>
      (Bitmap).Handle,
    header_ct_size, img_size
    );

  // compute the total size
  const size_t total_size =
    header_ct_size + img_size;

  if (total_size > 0)
  {
    //
    // memory for the header
    // and the color table
    //
    unsigned char* pHeaderAndCT =
      new unsigned char[header_ct_size];

    // memory for the pixels
    pBits = new unsigned char[img_size];

    try
    {
      //
      // use the GetDIB() function
      // to get the header, color table
      // and a copy of Bitmap's pixels
      //
      bool got_dib =
        GetDIB(
          const_cast<Graphics::TBitmap&>
```

```
            (Bitmap).Handle,
          const_cast<Graphics::TBitmap&>
            (Bitmap).Palette,
          pHeaderAndCT, pBits
          );
      if (got_dib)
      {
        return reinterpret_cast
          <LPBITMAPINFO>(pHeaderAndCT);
      }
    }
    catch (...)
    {
      // clean up
      delete [] pHeaderAndCT;
      delete [] pBits;
      throw;
    }
  }
  return NULL;
}
```

Drawing a DIB to a display device context

Contrary to its name, the `StretchDIBits()` function isn't limited to rendering a DIB in only a stretched fashion. If you specify a destination rectangle that's identical to the source rectangle (and the target DC has a 1:1 mapping mode), then the DIB will be rendered in its original size. Let's look at a bare-bones example. Here, we'll create and draw a DIB in response to the click of a button. Assume that `Image1` contains the bitmap that's to be drawn. Here's the code:

```
void __fastcall TForm1::Button1Click(
  TObject *Sender)
{
  Graphics::TBitmap& Bitmap =
    *Image1->Picture->Bitmap;

  unsigned char* pBits = NULL;
  const LPBITMAPINFO pDIB =
    CreateDIB(Bitmap, pBits);
```

```
if (pDIB && pBits)
{
  try
  {
    const int num_lines =
      StretchDIBits(
        // destination DC
        Canvas->Handle,
        // destination rectangle
        0, 0,
        pDIB->bmiHeader.biWidth,
        abs(pDIB->bmiHeader.biHeight),
        // source rectangle
        0, 0,
        pDIB->bmiHeader.biWidth,
        abs(pDIB->bmiHeader.biHeight),
        // source image
        pBits, pDIB,
        // color table type
        DIB_RGB_COLORS,
        // ROP3 code
        SRCCOPY
        );

    if (num_lines !=
        abs(pDIB->bmiHeader.biHeight))
    {
      throw
        Exception("!StretchDIBits");
    }
  }
  catch (...)
  {
    delete [] pDIB;
    delete [] pBits;
    throw;
  }
  delete [] pDIB;
  delete [] pBits;
}
```

```
}
```

Note that we passed `DIB_RGB_COLORS` as the `iUsage` parameter to the `StretchDIBits()` function. This specification is indeed valid in our case because the `GetDIB()` function (which we called from within the `CreateDIB()` function) always creates a DIB with an `RGBQUAD`-type color table entry.

If you've used the `TCanvas::CopyRect()` method before, you've probably noticed that the `StretchDIBits()` function isn't that much different. In contrast to the `StretchDIBits()` function, however, the `CopyRect()` method takes care of the palette-related issues when needed (i.e., on palette-based displays). So, before we move on to printing, let's modify this example to account for palettized displays. We do this by using a combination of the `GetDeviceCaps()`, `SelectPalette()`, `RealizePalette()` functions, and by using the `TBitmap::Palette` property, like so:

```
void __fastcall TForm1::Button1Click(
  TObject *Sender)
{
  Graphics::TBitmap& Bitmap =
    *Image1->Picture->Bitmap;

  unsigned char* pBits = NULL;
  const LPBITMAPINFO pDIB =
    CreateDIB(Bitmap, pBits);

  if (pDIB && pBits)
  {
    try
    {
      // is the display palette-based?
      bool uses_palette =
        GetDeviceCaps(
          Canvas->Handle, RASTERCAPS
          ) & RC_PALETTE;

      HPALETTE hOldPal;
      if (uses_palette)
      {
        //
        // select the bitmap's palette
        // into the target DC
```

```
  //
  hOldPal =
    static_cast<HPALETTE>(
      SelectPalette(
        Canvas->Handle,
        Bitmap.Palette,
        FALSE
        )
      );
  // update the system palette
  RealizePalette(Canvas->Handle);
}

const int num_lines =
  StretchDIBits(
    // destination DC
    Canvas->Handle,
    // destination rectangle
    0, 0,
    pDIB->bmiHeader.biWidth,
    abs(pDIB->bmiHeader.biHeight),
    // source rectangle
    0, 0,
    pDIB->bmiHeader.biWidth,
    abs(pDIB->bmiHeader.biHeight),
    // source image
    pBits, pDIB,
    // color table type
    DIB_RGB_COLORS,
    // ROP3 code
    SRCCOPY
    );

if (uses_palette)
{
  // clean up
  SelectPalette(
    Canvas->Handle, hOldPal, TRUE
    );
}
```

```
      if (num_lines !=
          abs(pDIB->bmiHeader.biHeight))
      {
        throw
          Exception("!StretchDIBits");
      }
    }
    catch (...)
    {
      delete [] pDIB;
      delete [] pBits;
      throw;
    }
    delete [] pDIB;
    delete [] pBits;
  }
}
```

The `GetDeviceCaps()` function is used to query various capabilities of a device. Here we specify the `RASTERCAPS` identifier and look for the `RC_PALETTE` bit. The presence of this bit indicates that the specified device is palette-based, in which case we use a combination of the `SelectPalette()` and `RealizePalette()` functions to update the display's system palette. Note that these latter two functions are usually (and more appropriately) used in response to the `WM_QUERYNEWPALETTE` message. As we'll examine next, in the case that a printer is palette-based, we'll use the `SelectPalette()` and `RealizePalette()` functions in a similar fashion.

Drawing a DIB to a printer device context

Okay, so we've seen how to use the `StretchDIBits()` function to render a DIB to a display DC. How is this function used to print a DIB? Well, it's the same deal, just a different target DC. This time, instead of passing a handle to our form's device context as the `StretchDIBits()` function's `hdc` parameter, we pass a handle to a device context that refers to a specific printer. For example, by using the `TPrinter` class, this can be done like so:

```
#include <printers.hpp>
void __fastcall TForm1::Button1Click(
  TObject *Sender)
{
  Graphics::TBitmap& Bitmap =
```

```
  *Image1->Picture->Bitmap;

unsigned char* pBits = NULL;
const LPBITMAPINFO pDIB =
  CreateDIB(Bitmap, pBits);

if (pDIB && pBits)
{
  // start the print job
  Printer()->BeginDoc();
  try
  {
    // is the printer palette-based?
    bool uses_palette =
      GetDeviceCaps(
        Printer()->Canvas->Handle,
        RASTERCAPS
        ) & RC_PALETTE;

    HPALETTE hOldPal;
    if (uses_palette)
    {
      //
      // select the bitmap's palette
      // into the target DC
      //
      hOldPal =
        static_cast<HPALETTE>(
          SelectPalette(
            Printer()->Canvas->Handle,
            Bitmap.Palette,
            FALSE
            )
          );
      // update the printer's palette
      RealizePalette(Canvas->Handle);
    }

    const int num_lines =
      StretchDIBits(
        // destination DC
```

```
      Printer()->Canvas->Handle,
      // destination rectangle
      0, 0,
      pDIB->bmiHeader.biWidth,
      abs(pDIB->bmiHeader.biHeight),
      // source rectangle
      0, 0,
      pDIB->bmiHeader.biWidth,
      abs(pDIB->bmiHeader.biHeight),
      // source image
      pBits, pDIB,
      // color table type
      DIB_RGB_COLORS,
      // ROP3 code
      SRCCOPY
      );

  if (uses_palette)
  {
    // clean up
    SelectPalette(
      Printer()->Canvas->Handle,
      hOldPal, TRUE
      );
  }

  // end the print job
  Printer()->EndDoc();

  if (num_lines !=
      abs(pDIB->bmiHeader.biHeight))
  {
    throw
      Exception("!StretchDIBits");
  }
}
catch (...)
{
  Printer()->EndDoc();
  delete [] pDIB;
  delete [] pBits;
```

```
      throw;
    }
    delete [] pDIB;
    delete [] pBits;
  }
}
```

Note that this code is virtually identical to that of the previous code snippets. The only difference is that here we use the `TPrinter::BeginDoc()` and `EndDoc()` methods, and we specify `Printer()->Canvas->Handle` instead of `Canvas->Handle` (i.e., `Form1->Canvas->Handle`). In fact, that's pretty much all there is to printing a bitmap. This approach will succeed on most printers. If you do execute this code, however, you'll likely get a very small print depending on the resolution of your printer and the dimensions of your bitmap. The reason is that we specified the destination rectangle (i.e., the `XDest`, `YDest`, `nDestWidth`, and `nDestHeight` parameters) in terms of the bitmap's actual dimensions. Usually, you'll want the printed bitmap to be of a certain number of inches or to optimally fit within a page.

To determine the correct scaling factor for the destination rectangle, you can use the `TPrinter::PageWidth` and `PageHeight` properties; or you can use the `GetDeviceCaps()` function. Or you can use a combination of these methods. For example, to stretch the bitmap to the size of the paper (i.e., to occupy the entire printable area), you use the `PageWidth` and `PageHeight` properties as follows:

```
const SIZE SPrint = {
   Printer()->PageWidth,
   Printer()->PageHeight
   };

const int num_lines =
  StretchDIBits()(
    // destination DC
    Printer()->Canvas->Handle,
    // destination rectangle
    0, 0,
    SPrint.cx, SPrint.cy,
    // source rectangle
    0, 0,
    pDIB->bmiHeader.biWidth,
    abs(pDIB->bmiHeader.biHeight),
    // source image
```

```
    pBits, pDIB,
    // color table type
    DIB_RGB_COLORS,
    // ROP3 code
    SRCCOPY
    );
```

Similarly, if you want the printed bitmap as large as possible (on a single page) while preserving the bitmap's aspect ratio, you use a combination of `PageWidth`, `PageHeight`, and `GetDeviceCaps()`, like so:

```
// pixels per inch horizontally
const int pix_inchX =
  GetDeviceCaps(
    Printer()->Canvas->Handle,
    LOGPIXELSX
    );
// pixels per inch vertically
const int pix_inchY =
  GetDeviceCaps(
    Printer()->Canvas->Handle,
    LOGPIXELSY
    );

SIZE SPrint;
SPrint.cx = Printer()->PageWidth;
SPrint.cy =
   SPrint.cx * pix_inchY *
   abs(pDIB->bmiHeader.biHeight) /
   (pix_inchX *pDIB->bmiHeader.biWidth);

const int num_lines =
  StretchDIBits()(
    // destination DC
    Printer()->Canvas->Handle,
    // destination rectangle
    0, 0,
    SPrint.cx, SPrint.cy,
```

```
// source rectangle
0, 0,
pDIB->bmiHeader.biWidth,
abs(pDIB->bmiHeader.biHeight),
// source image
pBits, pDIB,
// color table type
DIB_RGB_COLORS,
// ROP3 code
SRCCOPY
);
```

Paperless proofing

Unless you're extremely lucky or you have a knack for getting things right the first time, it'll probably take a few tries to get your printing code fine-tuned. Although you can proof some aspects by simply sending the output (of `StretchDIBits()`) to your form's DC instead of to your printer's DC, it's a bit difficult to verify positioning and sizing related code without seeing a printed page. After wasting a few stacks of paper, I finally realized that there's a simple way to gauge (at least approximately) what the printed output will look like—both in terms of position and dimensions—without actually producing a hard copy.

If you go to this URL: **http://www.cs.wisc.edu/~ghost/**, you can download Ghostscript and GSview. Together, these programs allow you to view Postscript files on screen. This way, you can test your printing code by first displaying a Print common dialog box, and then checking the "Print to file" option. When you confirm the Print dialog box, you'll be asked to specify a destination file name; your "printed" output will then be directed to the file that you specify. After you've printed to the file, you can open it in GSView to see what the output looks like. Of course, for this to work, you'll need a Postscript printer driver. If you already have a Postscript printer then you're good to go. Otherwise, pop in your Windows CD-ROM and install a Postscript printer driver (you don't need an actual printer for this to work). I've found the Tektronix Phaser 540 driver to work particularly well.

Conclusion

Is the `StretchDIBits()` function the only way to print a bitmap? Actually, no. I pointed out last time that there are two functions for transferring a DIB to a device: `StretchDIBits()` and `SetDIBitsToDevice()`. The `SetDIBitsToDevice()` function is similar to `StretchDIBits()` except the former doesn't allow stretching. Moreover, the `SetDIBitsToDevice()` function is supported by fewer devices, so in many cases, the work is

simply punted to the `StretchDIBits()` function.

Next time, I'll discuss how to make the `StretchDIBits`-based approach a bit more robust by adding support for a technique called banding. I'll also show you how to use ICM (Image Color Management) technology to ensure the fidelity of a printed image's colors.

Setting the system time

by Kent Reisdorph

Some applications expect the system time to be synchronized with a known time source. Other applications depend on several machines on a network to have the same date and time. You can set the system time for a particular machine by calling the `SetLocalTime()` or `SetSystemTime()` functions. These functions operate in the same way except that the specified time is local time for the `SetLocalTime()` function and UTC time for the `SetSystemTime()` function. These functions are used to set both the date and the time, despite what their names would indicate. This article will focus on the `SetLocalTime()` function.

Before calling `SetLocalTime()` you must declare an instance of the `SYSTEMTIME` structure and initialize it's members to the new date and time. `SYSTEMTIME` is declared as follows:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME;
```

As you can see, this structure's members correspond to the separate parts of dates and times. The `dwDayOfWeek` member is not used when setting the system date and time (it is, however, used when retrieving the date and time using the `GetLocalTime()` and `GetSystemTime()` functions).

After you have set the members of this structure you call `SetLocalTime()` to update the system date and time. Here is the declaration for `SetLocalTime()`:

```
BOOL SetLocalTime(
  CONST SYSTEMTIME *lpSystemTime);
```

`SetLocalTime()` returns a non-zero value if the call succeeded, or zero if the call failed. Call `GetLastError()` to determine the cause of the failure if zero is returned. Reasons for failure could include a bad value (specifying a minute value greater than 59 for example) or because

the user didn’t have proper access to change the system time.

The following code adds one minute to the current time. It first declares an instance of `TDateTime` to get the current date and time. It then decodes the date and time in order to get the individual date and time values. These values are then assigned to their corresponding member in the `SYSTEMTIME` structure (with the exception that the minute is incremented by one). Here is the example code:

```
unsigned short year, month, day;
unsigned short h, m, s, ms;
TDateTime dt = Now();
dt.DecodeDate(&year, &month, &day);
dt.DecodeTime(&h, &m, &s, &ms);
SYSTEMTIME st;
st.wYear = year;
st.wMonth = month;
st.wDay = day;
st.wHour = h;
st.wMinute = m + 1;
st.wSecond = s;
st.wMilliseconds = ms;
SetLocalTime(&st);
```

Note that this code doesn’t take into account the fact that the current minute may be 59. In that case both the hour and the minute values should be changed accordingly. Obviously, many values would have to be manually adjusted if the current date and time were December 31, 2001 at 23:59:00.

Changing the system time is easy but care must be taken to use proper values if you are simply making a time adjustment.

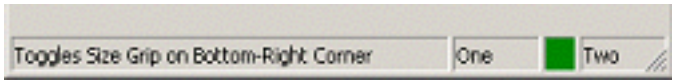
Status bar stuff

by Mark G. Wiseman

Most of the programs I write use status bars. I normally just use the `TStatusBar` component in the VCL. `TStatusBar` does just about everything I want. The one irritation I have with `TStatusBar` is the fact that it aligns the panels within the status bar to the left (**Figure A**). I don't like this. I want the panels aligned to the right (**Figure B**).

Figure A: *Status bar panels aligned to left. Bad!*

Figure B: *Status bar panels aligned to right. Good!*



I am going to show you how I align the panels to the right and how I keep them aligned that way when the form that contains the status bar is resized. While I'm at it, I'm going to show you a few other things you can do with `TStatusBar`.

There is an example program that accompanies this article. You can find it at the Bridges Publishing Web site.

Right-aligned panels

I placed a `TStatusBar` component on a form named `TStatusBarForm`. `TStatusBar` has an event named `OnResize` that will fire whenever the `TStatusBar` is resized. I assigned the following code to the `OnResize` event:

```
int width = 0;

for (int i = 1; i <
     StatusBar->Panels->Count; i++)
  width += StatusBar->Panels->Items[i]->Width;
```

```
StatusBar->Panels->Items[0]->Width =
  StatusBar->ClientWidth - width;
```

This code will align the panes to the right in the status bar. It does this by assigning, to the left-most panel, a width that will push the rest of the panels all the way to the right.

`TStatusBar` contains a property, `Panels`. `Panels` contains a list of the panels in the status bar. The list is contained in the `Items` property of `Panels` as a zero-based array. The first or left-most panel is `Item[0]`. The code sums the width of all the panels except the first panel. It then subtracts this sum from the `ClientWidth` of the `TStatusBar` and assigns the result to the first panel.

It is important to use the `ClientWidth` property of `TStatusBar` and not the `Width` property. The `Width` property includes the border width of `TStatusBar` and would make the first panel too wide.

The `OnResize` event of `TStatusBar` is fired every time the status bar is resized, including when the status bar is first created.

Owner-drawn panels

If you want to put anything other than text into a `TStatusBar` panel, you will need to use owner-drawn panels. This is actually very easy to do.

In the Object Inspector for `TStatusBar`, click on the button for the `Panels` property to bring up the panels editor. Click on one of the panels to show its properties in the Object Inspector. Next change the `Style` property of the panel from `psText` to `psOwnerDraw`.

Now, go back to the Object Inspector for `TStatusBar`. Under the Events tab, double click on the `OnDrawPanel` event to insert the skeleton code for the event into your form. Now all you have to do is add the code to draw your panel.

In the example program, I draw a colored rectangle in one panel. The rectangle is either red or green based on the state of a `TCheckBox` component. Here is the code to do that:

```
void __fastcall
TStatusBarForm::StatusBarDrawPanel(
  TStatusBar *StatusBar,
  TStatusPanel *Panel,
  const TRect &Rect)
{
  StatusBar->Canvas->Brush->Color =
    WarningChk->Checked ?
```

```
      clRed : clGreen;
  StatusBar->Canvas->FillRect(Rect);
}
```

As you can see, I only wrote two lines of code; the rest was generated by the IDE.

Other stuff

There are a few more things I wanted to tell you about `TStatusBar`. If you set the `AutoHint` property to true, hints will be automatically displayed in the first panel of the status bar. You don't need to write any code to display hints!

If you set the `SimplePanel` property to true, the status bar will display only the first panel. If you have created other panels, they will be hidden. This might be useful if you want to display a very long hint. When `SimplePanel` is true, you can set the text displayed in the Panel by assigning a string to the `SimpleText` property.

Finally, the `SizeGrip` property of `TStatusBar` controls the display of the size grip at the bottom right corner of the status bar. Setting this property to `false` will cause the grip to be hidden.

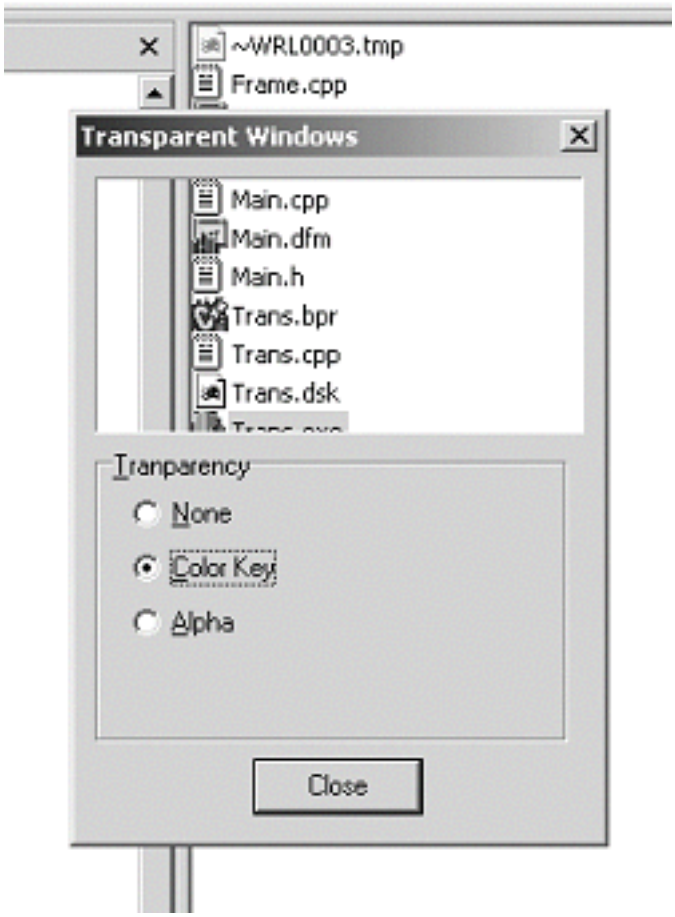
Transparent 2000

by Mark G. Wiseman

If you are writing a program to be run on Windows 2000 only and you need to have a transparent window, I've got just the API call for you. If you need transparency and are writing for other versions of Windows, you can still use this function when running on 2000.

Starting with Windows 2000, Microsoft introduced the function, `SetLayeredWindowAttributes()` in the Windows API. I've written a small demonstration program that you can download from the Bridges Publishing Web site. **Figures A and B** show the program in two different transparency states. You can see part of *Windows Explorer* underneath the program window.

Figure A: *Demo Program using Color Key Transparency.*



Perfectly transparent

So, how do you create transparent windows in Windows 2000? The first thing you have to do is tell Windows that your form is a *layered window*. You can do this by setting the extended

window style bit, WS_EX_LAYERED. Here is the code for that:

```
SetWindowLong(Handle, GWL_EXSTYLE,
  GetWindowLong(Handle, GWL_EXSTYLE)
  & ~WS_EX_LAYERED);
```

Now all you need to do is call SetLayeredWindowAttributes(). Here is the prototype for this function:

```
BOOL SetLayeredWindowAttributes(
  HWND hwnd, COLORREF crKey,
  BYTE bAlpha, DWORD dwFlags);
```

The hwnd parameter is a handle to your form. Either the crKey or bAlpha parameter is used based on the value of the dwFlags parameter. There are two flags you can use with the dwFlags parameter: LWA_COLORKEY and LWA_ALPHA.

Using this dwFlags parameter to SetLayeredWindowAttributes(), you can create two types of transparency. The first type, color key transparency, makes a specific color in the window transparent. This is what you see in **Figure A**. The transparent part of the form is just a TPanel with its color set to clRed. The second type of transparency is alpha-blending. With alpha-blending the entire form becomes transparent. **Figure B** shows this.

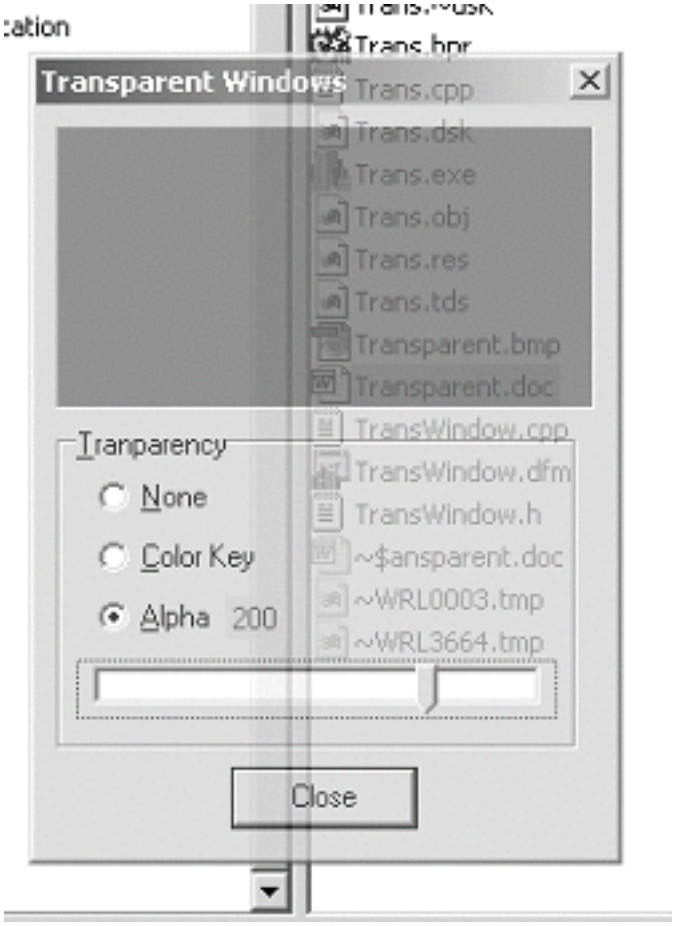
When you use the LWA_COLORKEY flag, you have to set the crKey parameter to the color you want to become transparent. Any part of your form that has this color will become transparent, so you need to be careful. In the demo program, I chose a color, clRed that was only used as the color of a TPanel that I wanted to become transparent.

```
SetLayeredWindowAttributes(
  Handle, clRed, 0, LWA_COLORKEY);
```

When dwFlags is set to LWA_ALPHA, alpha-blending is used. As I mentioned, the entire form or window becomes transparent with alpha-blending. You can set the amount of transparency using the bAlpa parameter. When bAlpha is set to 255, the form is opaque and looks just like a normal window. If you set bAlpha to 0, the form becomes completely transparent – invisible! In **Figure B**, I have set the bAlpha value to 200 using the slider control.

```
SetLayeredWindowAttributes(
  Handle, 0, 200, LWA_ALPHA);
```

Figure B: *Demo Program using Alpha-Blending.*



Conclusion

Using layered windows is a really easy way to program transparency. Unfortunately `SetLayeredWindowAttributes()` only works with Windows 2000. There are ways to make forms transparent in Windows 9x and NT, but they are much more difficult and beyond the scope of this article.

Also, you need to remember to be careful if you are using alpha-blending. Make sure you give the users of your program a way to find the form if they make it completely invisible.

Windows hooks

by Kent Reisdorph

There are times when you need to know what's going on within Windows at the system level. You might need to know when a particular window (application) is closed, or when a particular keystroke combination is typed on the keyboard. The Windows API provides a means for spying on the system to obtain this kind of information: Windows hooks.

This article will explain hooks and the use of the Win32 API functions `SetWindowsHookEx()`, `UnhookWindowsHookEx()`, and `CallNextHookEx()`. It will also point out some of the common pitfalls to avoid when implementing a hook.

What's a hook?

A Windows hook is a way of spying on Windows at the system level. Ultimately a Windows hook is a function you write that Windows calls whenever a hook event occurs. A hook allows you to examine Windows messages before they are processed by the system. You can simply monitor messages or you can modify certain messages if you choose. You provide a hook function, register the hook function with Windows, and from that point on Windows will call your hook function each time a particular type of system event occurs.

There are several types of Windows hooks. Most hooks can be installed to monitor an event in a specific thread or at the system level. A keyboard hook, for example, could capture the keystrokes for just one application or for all applications on the system. Some hooks can only be implemented at the system level. **Table A** lists the constants representing the hook types, the scope for that hook, and a description of each hook type.

Table A: *Windows hook types.*

Hook	Scope	Description
`WH_CALLWNDPROC`	Thread or system	Monitors Windows messages before they are sent to a particular window's window procedure.
`WH_CALLWNDPROCRET`	Thread or system	Monitors Windows messages after they have been processed by a particular window's window procedure.
`WH_CBT`	Thread or system	Used for computer-based training applications.
`WH_DEBUG`	Thread or system	Used when debugging other hook functions.
`WH_FOREGROUNDIDLE`	Thread or system	The hook procedure will be called when an application's foreground window goes idle.
`WH_GETMESSAGE`	Thread or system	Monitors messages posted to a message queue.
`WH_JOURNALPLAYBACK`	System only	Plays back a series of keystrokes and mouse movements previously recorded with a journal record hook (a macro). Normal mouse and keyboard input is disabled while the macro is playing.
`WH_KEYBOARD`	Thread or system	The hook procedure will be called when a `WM_KEYDOWN` or `WM_KEYUP` message is about to be processed.
`WH_JOURNALRECORD`	System only	The hook procedure will be called for all keyboard and mouse messages. Useful for recording macros.
`WH_KEYBOARD_LL`	Thread or system	A low-level keyboard hook (Windows NT only).
`WH_MOUSE`	Thread or system	The hook procedure will be called when a mouse message is about to be processed.
`WH_MOUSE_LL`	Thread or system	A low-level mouse hook (Windows NT only).
`WH_MSGFILTER`	Thread or system	Monitors messages as a result of an action in a menu, dialog box, or scroll bar created by a specific application.
`WH_SHELL`	Thread or system	Monitors messages affecting the Windows shell.
`WH_SYSMSGFILTER`	System only	Monitors messages as a result of an action in a menu, dialog box, or scroll bar created by any application.

The first step in implementing a Windows hook is to decide on the type of hook you need. A quick glance at **Table A** will reveal that making that determination might be difficult. You may have to spend some time experimenting before you settle on the best type of hook to use. Once you have determined the type of hook you need, you can get on with the work of writing the hook itself.

Windows hook functions

The Windows API has three primary hook functions. The `SetWindowsHookEx()` function is used to install a hook, the `UnhookWindowsHookEx()` function is used to remove a hook, and the `CallNextHookEx()` function is used to call the next hook in the hook chain (I will explain hook chaining a little later). The following sections describe these functions and explain their use.

SetWindowsHookEx()

As its name implies, `SetWindowsHookEx()` is used to install a hook. Here's the declaration:

```
HHOOK SetWindowsHookEx(
  int idHook,
  HOOKPROC lpfn, procedure
  HINSTANCE hMod,
  DWORD dwThreadId
);
```

The first parameter is used to specify the type of hook to install (see the constants in **Table A**). The second parameter is used to pass the address of a hook procedure that will be called when Windows activates the hook (I will explain the hook procedure later). The third parameter is the instance handle of the module where the hook procedure resides (the executable or a DLL). The final parameter is used to specify the thread ID of the thread to which the hook is being attached. For system-wide hooks this parameter must be 0.

If the hook was successfully installed, `SetWindowsHookEx()` returns a handle to the hook (an `HHOOK`). This handle is used later when uninstalling the hook and when calling the next hook in the hook chain. If the call to `SetWindowsHookEx()` fails the return value is 0. The following example installs a keyboard hook for the current process:

```
HHOOK KbHook = SetWindowsHookEx(
  WH_KEYBOARD, (HOOKPROC)MyKBHook,
  HInstance, GetCurrentThreadId());
```

In this example, the name of the hook procedure is `MyKBHook()`. Notice that the Windows API function `GetCurrentThreadId()` is used to specify the thread ID of the process for which the hook is attached. Notice also that the return value is saved in a variable called `HKbHook`. You should always check the return value to insure that the hook was successfully installed.

UnhookWindowsHookEx()

`UnhookWindowsHookEx()` removes a hook from the system. This function returns a non-zero value if the hook was successfully removed, or zero if an error occurred removing the hook. The most common reason for an error to occur while removing a hook is that the hook as already been removed and the hook handle is invalid. Removing a hook is trivial as this example shows:

```
Res = UnhookWindowsHookEx(HKbHook);
```

As you can see, there's not much to `UnhookWindowsHookEx()`. Just be sure that you check the return value and take appropriate action if removing the hook fails.

CallNextHookEx()

Windows uses a chaining mechanism for hooks. This allows more than one process to implement a particular type of hook. Let's say, for example, that you install a keyboard hook. At that point you don't know whether your keyboard hook is the only installed hook or not. One or more other applications might also have installed a keyboard hook. Windows calls your keyboard hook procedure and then expects you to keep the chain intact by calling the next hook in the chain. Calling the next hook in the chain is accomplished using the `CallNextHookEx()` function. You will see an example of `CallNextHookEx()` in the next section.

The hook procedure

The key element to a Windows hook is the hook procedure. The hook procedure is called each time an event occurs within Windows pertinent to the type of hook installed. The hook procedure for a keyboard hook, for example, will be called each time a key is pressed or released. Windows defines a hook procedure for each hook type. The declaration for a keyboard hook procedure, for example, looks like this:

```
LRESULT CALLBACK MyKBHook(int Code,
  WPARAM wParam, LPARAM lParam);
```

Actually, all of the hook procedures have this same signature. The difference between the different hook procedures is in the values of the `Code`, `wParam`, and `lParam` parameters. These values of these parameters vary with the type of hook installed. It is not practical to attempt to cover each and every hook procedure in this article. See the Win32 API help under the topics `SetWindowsHookEx()` and "Hook Functions" for a description of the various hook procedures and their parameters.

Any code in the hook procedure should be designed as efficiently as possible. A mouse hook, for example, is called every time a mouse message is generated. This could result in the mouse hook procedure being called thousands of times per second. Obviously you want any code in a hook procedure to be very efficient.

Ultimately the hook procedure needs to call `CallNextHookEx()` before exiting. Here's an example of a keyboard hook procedure that performs no action other than to call the next hook in the hook chain:

```
LRESULT CALLBACK MyKBHook(int Code,
  WPARAM wParam, LPARAM lParam)
{
  return CallNextHookEx(
    HKbHook, Code, wParam, lParam);
}
```

As you can see, the values of the handle to the current hook (obtained in the initial call to `SetWindowsHookEx`) and the values of the `Code`, `wParam`, and `lParam` parameters are passed on to `CallGetNextHookEx()`. This insures that hook chain remains intact. Failure to call `CallNextHookEx()` will have unpredictable and almost certainly undesirable effects. Notice that the return value of `CallNextHookEx()` is returned as the hook procedure's function result.

Where does the hook procedure live?

An important aspect of a Windows hook is the question of where the hook procedure resides. If the hook is an application (thread) hook, the hook procedure can reside in either the application itself or in a DLL. If the hook is a system-wide hook then the hook procedure *must* reside in a DLL.

Implementing a system-wide hook in a DLL can be frustrating on the first attempt. It's important to realize that each application that accesses the hook will load the hook DLL. For example, let's say you have installed a system-wide keyboard hook. Remember that you need to call the `CallNextHookEx()` function from within your hook procedure, and that you must pass the handle of the current hook when you do so. But where do you store the hook's handle? You would normally use a global variable, but remember that each process that attaches to a 32-bit DLL gets its own copy of any global variables in the DLL. That means that the variable that holds the hook handle will contain a valid value only for the process that installed the hook. You must find a way to share the hook handle among all instances of the DLL.

The best way to share data among instances of a DLL is with shared memory or a memory mapped file. Memory mapped files are beyond the scope of this article, so the example DLL presented later in this article uses VCL file streams to store hook handles. This is not necessarily a recommended approach, but is used in our example for the sake of simplicity. Needless to say, implementing a system-wide hook in a DLL takes careful consideration and planning.

Thread or system hook?

As indicated earlier, a hook can be installed for a specific thread or for the entire system (with the exception of system-only hooks). System-wide hooks can affect performance of the entire system so you should only use system-wide hooks when absolutely necessary. To install a hook for a specific thread, pass the thread ID in the final parameter to `SetWindowsHookEx()`. For example:

```
DWORD ID = GetCurrentThreadId();
HHOOK HKbHook = SetWindowsHookEx(
```

```
  WH_KEYBOARD, (HOOKPROC)MyKBHook,
  HInstance, ID);
```

This code installs a keyboard hook for the current process only. To install a system-wide keyboard hook use code like this:

```
HHOOK HKbHook = SetWindowsHookEx(
  WH_KEYBOARD, (HOOKPROC)MyKBHook,
  HInstance, 0);
```

This hook is a system-wide hook because 0 is passed for the `dwThreadId` parameter.

Regardless of whether you use a thread hook or a system-wide hook, you should install the hook only when needed, and uninstall it as soon as possible to avoid a drain on the system.

Note: Many programmers have attempted the use of a system-wide keyboard hook to disable the Ctrl-Alt-Delete key combination. This key combination is protected by Windows and even a keyboard hook will not allow you to intercept Ctrl-Alt-Delete.

Conclusion

Listings A and **B** contain the header and source code for a DLL that implements two hooks. The first hook is a keyboard hook. This hook can be either a thread hook or a system-wide hook (as determined by the thread ID passed to the DLL when installing the hook). The keyboard hook saves keystrokes in a buffer and then displays the keystrokes in a message box when the buffer is full. The second hook in the example program is a shell hook. This hook monitors the system and notifies you when Windows Notepad starts and also when Notepad shuts down. The code for this article also includes a calling application that installs the hooks. I don't show the source for the calling application here, but the source code for both the calling application and the DLL can be downloaded from the Bridges Publishing Web site.

There is no question that the use of windows hooks is rare. When you need a hook of a particular type, however, nothing else will do the job as effectively. Be judicious in your use of hooks, but keep the power of this technique in your arsenal.

Listing A: *MyHook.h*

```
#ifndef _MYHOOK_H
#define _MYHOOK_H

#ifdef __DLL__
  #define DLL_EXP __declspec(dllexport)
#else
  #define DLL_EXP __declspec(dllimport)
#endif

extern "C" {
  bool DLL_EXP HookKb(DWORD ThreadID);
  bool DLL_EXP HookShell();
  bool DLL_EXP Unhook();
}

#endif
```

Listing B: *MyHook.cpp*

```
#include <vcl.h>
#include <windows.h>
#pragma hdrstop

#pragma argsused
```

```
#include "myhook.h"

const int BuffSize = 50;
const String TempHookFile = "c:\\hook.dat";
HHOOK HKbHook;
HHOOK HShellHook;
char Buffer[BuffSize];
int Index;
TFileStream* FS;

int WINAPI DllEntryPoint(HINSTANCE hinst,
  unsigned long reason, void* lpReserved)
{
  if (reason == DLL_PROCESS_ATTACH) {
    HKbHook = 0;
    HShellHook = 0;
    Index = 0;
  }
  return 1;
}

// NOTE: This example uses a file stream to
// save and restore the DLL's global data. It
// would be better to use a memory mapped file
// for this purpose but use of a memory mapped
// file is beyond the scope of this article.

// The keyboard hook procedure.
LRESULT CALLBACK MyKBHook(int Code,
  WPARAM wParam, LPARAM lParam)
{
  // if (HKbHook is 0 it means that a new
  // process has attached to the hook DLL.
  // We must load the HKbHook value from
  // the temp file so that we can pass it
  // in CallNextHookEx.
  if (HKbHook == 0) {
    FS = new TFileStream(
      TempHookFile, fmOpenRead);
    FS->Read(HKbHook, sizeof(HKbHook));
    FS->Read(HShellHook, sizeof(HShellHook));
    delete FS;
  }
  // if (Code < 0, call the next hook and exit.
  if (Code >= 0)
    // Key up messages only
    if ((lParam & 0x80000000) == 0x80000000)
    {
      // Save the keystroke in the buffer and
      // increment the buffer index variable.
      Buffer[Index] = char(wParam);
      Index++;
      if (Index >= BuffSize) {
        // Buffer is full. Reset the buffer
        // index variable and show the buffer.
        Index = 0;
        MessageBox(0, Buffer, "Hook Message", 0);
      }
    }
```

```
  // Call the next hook in the chain.
  return CallNextHookEx(
    HKbHook, Code, wParam, lParam);
}

// The shell hook procedure.
int __stdcall MyShellHook(int Code,
  WPARAM wParam, LPARAM lParam)
{
  // if (HShellHook is 0 it means that a new
  // process has attached to the hook DLL.
  // We must load the HKbHook value from
  // the temp file so that we can pass it
  // in CallNextHookEx.
  if (HShellHook == 0) {
    FS = new TFileStream(
      TempHookFile, fmOpenRead);
    FS->Read(HKbHook, sizeof(HKbHook));
    FS->Read(HShellHook, sizeof(HShellHook));
    delete FS;
  }
  // Only interested in those case where a
  // window is created or destroyed.
  char buff[255];
  if ((Code == HSHELL_WINDOWCREATED) ||
     (Code == HSHELL_WINDOWDESTROYED)) {
    // Get the class name of the window.
    GetClassName((
      HWND)wParam, buff, sizeof(buff));
    // if (class name is Notepad then let
    // the show a message box.
    String S;
    if (!strcmp(buff, "Notepad")) {
      if (Code == HSHELL_WINDOWCREATED)
        S = "Notepad Starting!";
      else
        S = "Notepad Shutting Down!";
      MessageBox(0, S.c_str(), "Hook Message",0);
    }
  }
  // Call the next hook in the chain.
  return CallNextHookEx(
    HShellHook, Code, wParam, lParam);
}

// Exported function that is used to install
// the keyboard hook.
bool DLL_EXP HookKb(DWORD ThreadID)
{
  HKbHook = SetWindowsHookEx(WH_KEYBOARD,
    (HOOKPROC)MyKBHook, HInstance, ThreadID);
  bool Result = Boolean(HKbHook);
  // Save the hook handles to a temporary file
  // so that other processes can have access
  // to those values.
  FS = new TFileStream(
    TempHookFile, fmCreate);
  FS->Write(HKbHook, sizeof(HKbHook));
  FS->Write(HShellHook, sizeof(HShellHook));
```

```
  delete FS;
  return Result;
}

// Exported function that is used to install
// the shell hook.
bool DLL_EXP HookShell()
{
  HShellHook = SetWindowsHookEx(WH_SHELL,
    (HOOKPROC)MyShellHook, HInstance, 0);
  bool Result = Boolean(HShellHook);
  // Save the hook handles to a temporary file
  // so that other processes can have access
  // to those values.
  FS = new TFileStream(
    TempHookFile, fmCreate);
  FS->Write(HKbHook, sizeof(HKbHook));
  FS->Write(HShellHook, sizeof(HShellHook));
  delete FS;
  return Result;
}

// Exported function that uninstalls all hooks.
bool DLL_EXP Unhook()
{
  bool Result = false;
  if (HKbHook != 0)
    Result = UnhookWindowsHookEx(HKbHook);
  if (HShellHook != 0)
    Result = UnhookWindowsHookEx(HShellHook);
  HKbHook = 0;
  HShellHook = 0;
  // Delete the temporary file.
  DeleteFile(TempHookFile);
  return Result;
}
```

C++Builder file types

by Mark G. Wiseman

This issue of the Journal also contains part 4 of my series of articles on finding files. In these articles I develop a utility to delete unnecessary project files created by C++Builder. These files can take up a lot of hard disk space and I wanted a program that would easily remove them from my computer.

But, which files should the utility delete? This article presents a table of some of the file types associated with C++Builder and a brief discussion of each files purpose. I will indicate which files I think can safely be deleted. For all of the file types discussed in this article, the file's extension is assumed to indicate its type. **Table A** gives a quick reference to these files and whether they can be deleted. Further explanation of some of the file types follows.

Table A: C++Builder File Types

File Type	Description	Delete?
*.~???	Backup File	M
*.tds	External Debugger File	Y
*.il?	Incremental Linker File	Y
*.#nn	Precompiled Header File	M
*.csm	Precompiled Header File	Y
*.dci	Code Insight File	N
*.dct	Component Template File	N
*.dsk	Desktop File	M
*.dmt	Menu Template File	N
*.dro	Repository File	N

*.todo	To-do List	N
*.res	Compiled Resource File	N
*.idl	CORBA File	N
*.dfm	Form File	N
*.lib	Library File	N
*.bpi	Package File	N
*.bpl	Package File	N
*.dll	Windows DLL File	N
*.bpf	Project File	N
*.bpg	Project File	N
*.bpr	Project File	N
*.bpk	Project File	N
*.rc	Resource Script	N
*.c	Source Code	N
*.cpp	Source Code	N
*.h	Source Code	N
*.hpp	Source Code	N
*.pas	Source Code	N
*.tlb	Type Library File	N

Delete? Y = Yes, N = No, M = Maybe

***.~???** – Files with a tilde (~) as the first character in their extensions are backup files created by the IDE. For example MYCODE.~CPP is the most recent backup of the file MYCODE. CPP. Backup files are created for several types including source code, project files, and desktop files. If you are sure the current file is correct, you can delete the backup file.

***.tds** – Files with this extension are external debugger files. They contain information that the IDE uses to step through your project in the debugger. These files can easily

grow to several megabytes in size. One you have finished a project, you can safely delete these files. In fact, you can safely delete these files before a project is finished. If you do, you won't be able to debug you project until this file is recreated. The IDE will automatically recreate this file when you rebuild your project.

***.il?** – These are files used by the incremental linker to speed up linking. You can safely delete these files at any time. These files will be rebuilt the next time you link your project. This will cause that link to take longer, but your project will not be harmed in any way.

***.#nn** – (nn is a two-digit number, e.g. 01) These are precompiled header files for the VCL and are normally stored in the Program Files\Borland\CBuilder5\Lib folder. I have version 5 of C++Builder on my computer and currently I have five of these files named VCL50.#*nn*, where *nn* ranges from 00 to 04. You should be able to safely delete all of these files, however, I would recommend leaving the most recent. The IDE will rebuild these as needed, but there is not a good reason to force it to rebuild the most recent file.

***.csm** – Precompiled header files specific to each project begin with this extension. These are used to speed up project compilation. You can safely delete these files at any time. The files will be rebuilt when needed. The rebuilding will, of course, slow down your compile; but it will not harm your project.

***.dci**, ***.dct**, ***.dmt**, and ***.dro** – Information used for Code Insight, component templates, menu templates and the repository are stored in these files. This includes any changes you have made to these features of the IDE. These files should not be deleted. In fact, you may want to save these files before installing a new version of C++Builder.

***.dsk** – These file contains the desktop settings for the IDE. If you delete this file, your project will not be harmed, but you will lose all the window positions, file bookmarks, etc. you have set in the IDE.

***.todo** – This file stores the to-do items associated with a project but not associated with a specific source file. Do not delete this file.

***.res** – These are compiled resource files. It could be a file compiled form a resource script or a file created by the IDE to hold the project icon, etc. Don't delete these files unless you are very sure of what your doing.

***.idl** – This is a file type used for the CORBA Interface Definition Language (IDL). Don't delete these files.

***.dfm** – These files store definitions for VCL forms and should not be deleted. You should consider these files the same as source code.

***.lib**, ***.bpi**, ***.bpl**, ***.dll**, and ***.tlb** – These files are library files containing precompiled code (*.lib), package code (*.bpi and *.bpl), Windows DLLs (*.dll) and type library code (*.tlb). These files should not be deleted.

***.bpf**, ***.bpg**, ***.bpr** and ***.bpk** – Are project definition files and should not be deleted.

***.rc**, ***.cpp**, ***.c**, ***.hpp**, ***.h**, ***.pas** – Are all source code files. You may have written them, they may have been generated by the IDE, or they may have come from some other place. Normally you will not want to delete these files.

Conclusion

I have covered a lot of the file types associated with C++Builder projects. There are also a lot I have not covered. For instance, I did not cover many of the types associated with databases, Internet applications, and help files.

I hope you have a better idea of which files you might want to delete. In particular, after a project is complete, deleting only the files with the *.tds and *.csm extensions can free up megabytes of drive space.

Remember, be careful when deleting files and always consider making a backup first.

Dealing with network connections

by Kent Reisdorph

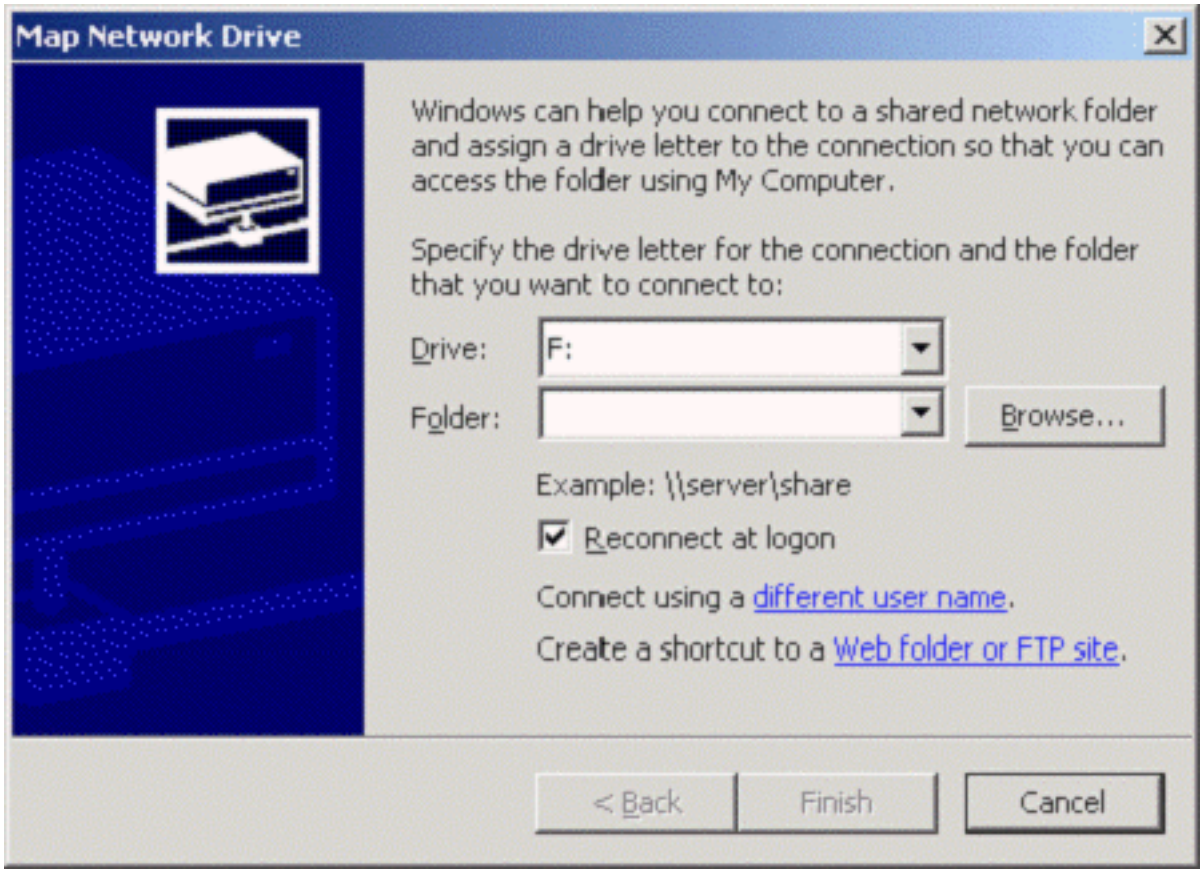
Network and Internet programming is becoming more and more common. An application may have to connect to a remote machine, transfer a file, and then disconnect again. Simple file transfers can be performed by connecting to a particular drive on a remote machine and copying a file in the usual ways.

The `WNet` family of Windows API functions is used to establish network connections, terminate connections, and enumerate the current connections. Mapping a drive on a remote machine can be performed in two different ways: using the built-in Windows dialog boxes, or entirely in code. I will explain each of these methods in this article.

Networking dialog boxes

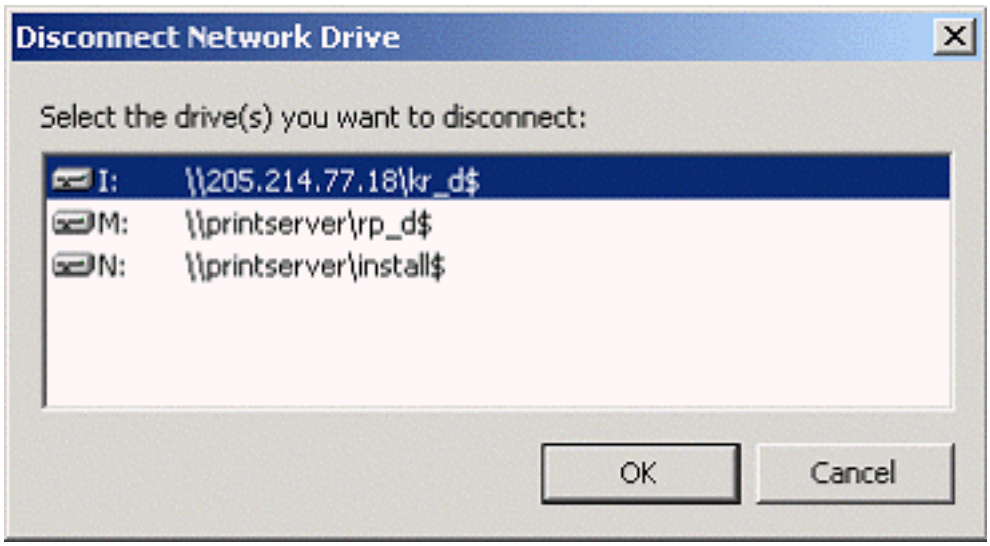
In some applications you will want to publicly announce your intention to map a network drive and rely on the user to provide the connection parameters. For those applications you can simply use the built-in Windows dialog boxes to map a network drive. Two dialog boxes are available for your use: the Map Network Drive dialog box and the Disconnect Network Drive dialog box. **Figure A** shows the Windows 2000 Map Network Drive dialog box. **Figure B** shows the Windows 2000 Disconnect Network Drive dialog box.

Figure A:



The Map Network Drive dialog box.

Figure B:



The Disconnect Network Drive dialog box.

The Windows NT and Windows 95/98 dialog boxes look slightly different than their Windows 2000 counterparts but perform the same tasks.

Putting these dialog boxes to work in your programs is really quite simple, as you'll find out in the following sections.

Mapping a network drive

The Map Network Drive dialog box is invoked with a call to `WNetConnectionDialog()`. Just one line of code is required:

```
WNetConnectionDialog(
  Handle, RESOURCETYPE_DISK);
```

It's as simple as that. The return value of `WNetConnectionDialog()` is 0 if the connection succeeds or `0xFFFFFFFF` if the connection fails. If the connection fails, you should call `GetLastError()` to determine the reason for the failure. Reasons for failure to connect include a bad remote machine name, a bad share name, an invalid password, or no network available. If a username and password are required to make the connection, Windows will display a second dialog box asking for that information.

Disconnecting a network drive

Disconnecting a network drive with the Disconnect Network Drive dialog box is just as easy. The `WNetDisconnectDialog()` function displays this dialog. Here's the code:

```
WNetDisconnectDialog(
  Handle, RESOURCETYPE_DISK);
```

The return value of `WNetDisconnectDialog()` is the same as described for `WNetConnectionDialog()`. `WNetDisconnectDialog()` may fail if the network is unavailable, but otherwise should succeed since you are choosing from a Windows-defined set of network connections. By the way, you can also remove a network printer using this function. Pass `RESOURCETYPE_PRINT` in the second parameter instead of `RESOURCETYPE_DISK`.

If your program is the type that can allow the user to choose a connection, you should certainly use the network dialog boxes as provided by Windows.

Connecting via code

Some applications need to map to a network drive entirely via code. For example, you might want to map a drive programmatically so the user is unaware that the connection is taking place. (Not all users need to know everything about your network.) Mapping a drive via code allows you to enforce a certain amount of security and still allow your program to do what it needs to do.

A connection can be made either with or without a local drive mapping. For example, you can connect to a shared drive on a remote machine and then perform operations against that drive using the network share's UNC name. For example, let's say you have successfully connected to a network share called `\\server\src$`. In that case you could copy a file from that drive like this:

```
CopyFile("\\\\\server\\src$\\data.txt",
  "c:\data\data.txt", false);
```

You don't need a drive letter at all in this case. In other cases it is advantageous to map a network share to a local drive letter. If so, you can map a network share to any available local drive letter. You'll have to find a drive letter that is not currently in use (I'll explain how to do that later in the article).

Connecting a network share via code requires using one of the `WNetAddConnection()` functions. There are three such functions, called `WNetAddConnection()`, `WNetAddConnection2()`, and `WNetAddConnection3()`. The `WNetAddConnection()` function is provided for backwards compatibility with earlier versions of Windows and should not be used. The difference between `WNetAddConnection2()` and `WNetAddConnection3()` is only that the latter provides an extra parameter for specifying a window handle. This handle is used as the owner window for any dialogs that Windows displays during the connection process. This example uses `WNetAddConnection2()` to map a drive:

```
void __fastcall TForm1::MapBtnClick(TObject *Sender)
{
  TNetResource NR;
  ZeroMemory(&NR, sizeof(NR));
  NR.dwType = RESOURCETYPE_DISK;
  NR.lpLocalName = "N:";
  NR.lpRemoteName =
    "\\\\machine\\share";
  DWORD Res = WNetAddConnection2(
    &NR, "Password", "Username", 0);
}
```

The first two lines declare an instance of the `TNetResource` structure and initialize all its members to 0. Next, the `dwType` field is set to `RESOURCETYPE_DISK`, indicating that you are connecting to a disk drive. After that the `lpLocalName` field is assigned to the string "N:". This specifies that the connection will be mapped to the local drive N. Note that you must provide the colon following the drive letter or the function call will fail. The next line in this example sets the `lpLocalName` field to the computer name and share to which you want to connect. Finally, the `WNetAddConnection2()` is called to make the connection.

The first parameter to `WNetAddConnection2()` is a pointer to a `TNetResource` structure. The

second and third parameters are used to specify the password and username used to make the connection. The final parameter determines whether the connection should be persistent. A persistent connection will be remembered when the user shuts down and restarts Windows. Pass 0 for this parameter to create a temporary connection, or `CONNECT_UPDATE_PROFILE` to establish a persistent connection.

The `WNetAddConnection2()` function returns 0 if it succeeds, or an error code if the function fails . A connection could fail for any number of reasons including a bad username or password, access denied, network busy or no network detected, and so on. You should check the return value from `WNetAddConnection2()` before attempting to use the connection. If the call to `WNetAddConnection2()` is successful you can use the mapped drive letter just like a local drive.

To establish a connection without mapping to a local drive letter, simply pass an empty string to the `lpLocalName()` field of the `TNetResource` structure. You can then access files and folders on the network share by using the UNC path rather than a local drive letter.

Terminating a network connection

Once you have connected to a network share and have done your business, you should disconnect. Disconnecting an existing connection is accomplished by calling the `WNetCancelConnection2()` function. (There is a `WNetCancelConnection()` function but it is provided for backwards compatibility and should not be used.) Here's how a call to `WNetCancelConnection2()` looks:

```
WNetCancelConnection2("N:", 0, false);
```

The first parameter to `WNetCancelConnection2()` is the name of the connection to cancel. This example disconnects the connection mapped to local drive N. To disconnect a share that is not mapped to a local drive letter, provide the computer and share name when you call `WNetCancelConnection2()`. For example:

```
WNetCancelConnection2(
  "\\\\computer\\share", 0, false);
```

The second parameter of `WNetCancelConnection2()` is used to determine connection persistence. If the connection was initially connected as persistent, passing 0 for this parameter will cancel the connection but will leave the connection in the list of remembered connections. If you pass `CONNECT_UPDATE_PROFILE` for this parameter then Windows will remove the connection from the remembered connections list.

The final parameter of `WNetCancelConnection2()` determines whether the connection is unconditionally terminated. If you pass `false` for this parameter, the call to `WNetCancelConnection2()` will fail if a user has a file open on the share. If you pass `true,` the

connection will be broken even if the user has files open on the share.

Enumerating connections

Sometimes you have to enumerate existing connections to determine the state of one or more connections. I can't explain every conceivable use for enumerating connections so I'll only show you how to enumerate active and remembered connections. Enumerating connections starts with a call to `WNetOpenEnum()`. It looks like this:

```
HANDLE EnumHandle;
DWORD Res = WNetOpenEnum(
  RESOURCE_CONNECTED, RESOURCETYPE_DISK,
  0, 0, &EnumHandle);
```

The first parameter is used to determine the type of the enumeration. Pass `RESOURCE_CONNECTED` for this parameter to enumerate only the active connections. If you want to enumerate remembered (persistent) connections, pass `RESOURCE_REMEMBERED` for this parameter.

The second parameter is used to specify the resource type to enumerate. Here I pass `RESOURCETYPE_DISK` to enumerate the active disk connections. I pass 0 for the third parameter to tell Windows to enumerate all resources. The fourth parameter is used to tell Windows what resource to enumerate. In this case we just pass 0 so Windows will enumerate the root of the network. (You could pass an instance of a `TNetResource` structure to enumerate a specific network container, but that subject is beyond the scope of this article.) Finally, we pass the address of the `EnumHandle` variable in the last parameter. If `WNetOpenEnum()` returns successfully, this variable will contain a handle we can use for the next step.

If `WNetOpenEnum()` returns successfully you can then call `WNetEnumResource()` to enumerate the connections. You pass the handle obtained in the call to `WNetOpenEnum()`, a pointer to a buffer (an array of `TNetResource` structures), variables for the number of resources you want to enumerate, and the required size of the buffer that Windows needs in order to carry out the enumeration. For example:

```
TNetResource NR[30];
DWORD Count = 0xFFFFFFFF;
DWORD Size = sizeof(TNetResource) * 30;
Res = WNetEnumResource(
  EnumHandle, &Count, &NR, &Size);
```

First I declare an array of 30 `TNetResource` structures. This code is cheating somewhat in that it assumes no more than 30 connections will exist. Note that I set the value of the `Count` variable to

`0xFFFFFFFF` before calling `WNetEnumResource()`. This value tells Windows to enumerate all available connections. I also set the `Size` variable to the size of the `TNetResource` array. Finally, I call `WNetEnumResource()` to tell Windows to enumerate the connections.

`WNetEnumResource()` will return 0 if the enumeration succeeded. A return value other than 0 indicates an error. (The code shown up to this point hasn't had any error-checking code, but you should always check the return value from the various `WNet` functions and take appropriate action if an error occurs.)

If `WNetEnumResource()` is successful, the `Count` variable will contain the number of connections found. You can then use a simple `for` loop to enumerate the connections:

```
for (DWORD i=0;i<Count;i++)
  Memo1->Lines->Add(
    NR[i].lpRemoteName);
```

This code adds the remote name of each connection to a memo component.

After enumerating connections you must call `WNetCloseEnum()` to free the memory Windows allocated for the enumeration:

```
WNetCloseEnum(EnumHandle);
```

Enumerating connections can be a bit more complex than the simple examples presented here. Much of the time, however, you may not even need to enumerate connections.

Getting a local drive letter

Before you can map a local drive letter to a network share, you must determine what drive letters are available. You cannot map a network share to a drive letter that is already in use. This includes removable drives (floppy, CD-ROM, or Zip drives), fixed drives (hard disks), and currently mapped network drives. Fortunately you can easily determine the next available drive letter with a simple `for` loop. Here's how it looks:

```
for (char i='D';i<'Z';i++) {
  String S = String(i) + ":";
  int DriveType =
    GetDriveType(S.c_str());
  DWORD Res = WNetGetConnection(
    S.c_str(), Buff, &Size);
  if (Res == 0)
```

```
    // No error. Drive is already connected.
  if (Res == ERROR_NOT_CONNECTED)
    // Drive not connected, so it's available.
}
```

This code checks the drive type by calling the Windows API function `GetDriveType()`. If the drive is not assigned to a local device, the `WNetGetConncetion()` function is called to get the status of the connection. If `WNetGetConnection()` returns `ERROR_NOT_CONNECTED` then the drive letter can be used to map a network share. See the `DrivesBtnClick()` method in the listing at the end of this article for the actual code used to determine if a drive letter is available.

Conclusion

With the information presented in this article you'll be able to easily map network drives from within your applications. **Listing A** contains the source code for our example program's main form. The example program allows you to map network drives and disconnect those drives again. It shows how to enumerate connections, and how to display the connection information for all drives on your system. You can download the source code for the example program from our Web site: www.bridgespublishing.com.

Listing A: *The example program's main unit.*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall TForm1::AddDlgBtnClick(
  TObject *Sender)
{
  // Show the Map Network Drive dialog.
  WNetConnectionDialog(
    Handle, RESOURCETYPE_DISK);
```

```
  // Update the display.
  ShowConnections();
}

void __fastcall TForm1::RmvDlgBtnClick(
  TObject *Sender)
{
  // Show the Disconnect Network Drive dialog.
  WNetDisconnectDialog(
    Handle, RESOURCETYPE_DISK);
  ShowConnections();
}

void __fastcall TForm1::EnumBtnClick(
  TObject *Sender)
{
  ShowConnections();
}

void __fastcall TForm1::DrivesBtnClick(
  TObject *Sender)
{
  Memo1->Lines->Clear();
  char Buff[256];
  DWORD Size = sizeof(Buff);
  // Enumerate the drives starting with D:
  for (char i='D';i<'Z';i++) {
    String S = String(i) + ":";
    // Find out the drive type. if (it"s a
    // hardware drive) continue looking.
    int DriveType = GetDriveType(S.c_str());
    if ((DriveType == DRIVE_FIXED) ||
       (DriveType == DRIVE_CDROM) ||
       (DriveType == DRIVE_RAMDISK)) {
      Memo1->Lines->Add(S +
        "\tFixed, removable, or CD-ROM drive");
      continue;
    }
    // Must be a network drive. Get the
    // connection status.
    DWORD Res = WNetGetConnection(
      S.c_str(), Buff, &Size);
    // Check the status.
    String S2;
```

```
    if (Res == 0)
      // No error. Drive is already connected.
      S2 = "Connected to " + String(Buff);
    if (Res == ERROR_NOT_CONNECTED)
      // Drive not connected, so it"s available.
      S2 = "Available";
    if (Res == ERROR_CONNECTION_UNAVAIL)
      // Remembered but not connected, can't use
      S2 = "Remembered but not connected ("
        + String(Buff) + ")";
    // Show the connection in the memo.
    Memo1->Lines->Add(S + "\t" + S2);
  }
}

void __fastcall TForm1::MapBtnClick(
  TObject *Sender)
{
  TNetResource NR;
  // Zero the memory.
  ZeroMemory(&NR, sizeof(NR));
  // Mapping a disk.
  NR.dwType = RESOURCETYPE_DISK;
  // Set the drive letter and path based on
  // the contents of the corresponding edits.

  char Drive[3];
  char Path[256];
  strcpy(Drive, DriveEdit->Text.c_str());
  strcpy(Path, PathEdit->Text.c_str());
  NR.lpLocalName = Drive;
  NR.lpRemoteName = Path;
  // Make the connection, passing the username
  // and password in the corresponding edits.
  DWORD Res = WNetAddConnection2(&NR,
    PasswordEdit->Text.c_str(),
    UsernameEdit->Text.c_str(), 0);
  // Report the connection results.
  if (Res == 0)
    ShowMessage("Connected");
  else
    ShowMessage(
      "Unable to connect. Error code "
```

```
      + IntToStr(Res) + ".");
  // Update the display.
  ShowConnections();
}

void __fastcall TForm1::UnMapBtnClick(
  TObject *Sender)
{
  // Unmap the drive or connection. if (the
  // drive letter edit is empty) assume
  // the path edit contains the name of the
  // connection to terminate.
  char Buff[256];
  if (DriveEdit->Text != "")
    strcpy(Buff, DriveEdit->Text.c_str());
  else
    strcpy(Buff, PathEdit->Text.c_str());
  DWORD Res =
    WNetCancelConnection2(Buff, 0, False);
  // Show the results.
  if (Res == 0)
    ShowMessage("Drive " + String(Buff) + " disonnected.");
  else
    ShowMessage("Unable to disconnect. "
      " Error code " + String(Res) + ".");
  // Update the display.
  ShowConnections();
}

void TForm1::ShowConnections()
{
  TNetResource NR[30];
  Memo1->Lines->Clear();
  // First enumerate the active conncections.
  Memo1->Lines->Add("Active Connections:");
  HANDLE EnumHandle;
  DWORD Res = WNetOpenEnum(RESOURCE_CONNECTED,
    RESOURCETYPE_DISK, 0, 0, &EnumHandle);
  if ((Res != 0)) {
    ShowMessage("Error");
    return;
  }
  DWORD Count = 0xFFFFFFFF;
```

```
DWORD Size = sizeof(TNetResource) * 30;
Res = WNetEnumResource(
  EnumHandle, &Count, &NR, &Size);
if ((Res != 0)) {
  if (Res == ERROR_NO_MORE_ITEMS)
    Memo1->Lines->Add("No active connections");
  else {
    ShowMessage("Error");
    return;
  }
}
// Iterate through the connections, and
// display the connection name in the memo.
String S;
for (DWORD i=0;i<Count;i++) {
  if (NR[i].lpLocalName == "")
    S = "(none)";
  else
    S = NR[i].lpLocalName;
  Memo1->Lines->Add(
    S + "\t" + NR[i].lpRemoteName);
}
// Close the enumeration.
WNetCloseEnum(EnumHandle);
Memo1->Lines->Add("");

// Now enumerate the remembered connections.
Memo1->Lines->Add("Remembered Connections:");
Res = WNetOpenEnum(RESOURCE_REMEMBERED,
  RESOURCETYPE_DISK, 0, 0, &EnumHandle);
if ((Res != 0)) {
  ShowMessage("Error");
  return;
}
Count = 0xFFFFFFFF;
Size = sizeof(TNetResource) * 30;
Res = WNetEnumResource(
  EnumHandle, &Count, &NR, &Size);
if ((Res != 0)) {
  if (Res == ERROR_NO_MORE_ITEMS)
    Memo1->Lines->Add(
      "\tNo remembered connections");
  else {
```

```
      ShowMessage("Error");
      return;
    }
  }

  if (Res == 0) {
    char Buff[512];
    Size = sizeof(Buff);
    for (DWORD i=0;i<Count;i++) {
      Res = WNetGetConnection(
        NR[i].lpLocalName, Buff, &Size);
      S = String(NR[i].lpLocalName) +
        "\t" + NR[i].lpRemoteName;
      if ((Res == ERROR_CONNECTION_UNAVAIL))
        S = S + "\t" + " (Not connected)";
      if (Res == 0)
        S = S + "\t" + " (Connected)";
      Memo1->Lines->Add(S);
    }
  }
  // Close the enumeration.
  WNetCloseEnum(EnumHandle);
}
```

Finding files, part 4

by Mark G. Wiseman

Actually, this article should be entitled “Deleting the files we found”. In the first three parts of this series, I showed you the code I wrote to find specific files on a computer. I wanted to write a utility program to find and delete the unnecessary files created by C++Builder.

Once you’ve found the files, deleting them is actually very easy. In this article I’ll present only the code I wrote to delete the files. If you’d like to see the whole thing tied together, you can download the example program from the Bridges Publishing Web site.

Choosing an approach

There are a couple of different ways to approach deleting files in Windows. The VCL has a function `DeleteFile()` that just calls the Windows API function of the same name.

The other approach is to use the Windows shell to delete the files. I chose this approach because the shell gives me the option of deleting the files outright or sending them to the Windows Recycle Bin. Also, if used correctly, the shell will prompt the user for confirmation before deleting files and show a progress dialog while deleting files.

I created a class called `TDeleteFiles` to do the work. The code for `TDeleteFiles` is in **Listings A** and **B**. Let’s see what the code does.

Going, going, gone…

The `TDeleteFiles` class encapsulates the Windows shell function `SHFileOperation()` and the structure `SHFILEOPSTRUCT`. This is done completely within the `Delete()` function of `TDeleteFiles`.

The `SHFileOperation()` relies on information contained within `SHFILEOPSTRUCT` to copy, move, rename or delete files. The `wFunc` member of `SHFILEOPSTRUCT` takes a value that tells `SHFileOperation()` which operation to perform. We will set `wFunc` to `FO_DELETE` to delete files.

Although I could use `SHFileOperation()` to delete each file as it is found, this function really shines when it is passed a list of files in the `pFrom` member of

SHFILEOPSTRUCT. When given a list of files and with the proper flags set, SHFileOperation() will prompt the user for confirmation before deleting files and will show a progress dialog while performing the deletion. If you have used the Windows Explorer, you have seen the dialog boxes shown in **Figures 1** and **2**. I'll discuss how to build a list of files later in the article.

The fFlags member of SHFILEOPSTRUCT accepts several flags that control the behavior of SHFileOperation(). The three flags we need to use are FOF_ALLOWUNDO, FOF_NOCONFIRMATION and FOF_SILENT.

Figure A: *The Windows 2000 Delete Files confirmation dialog box.*

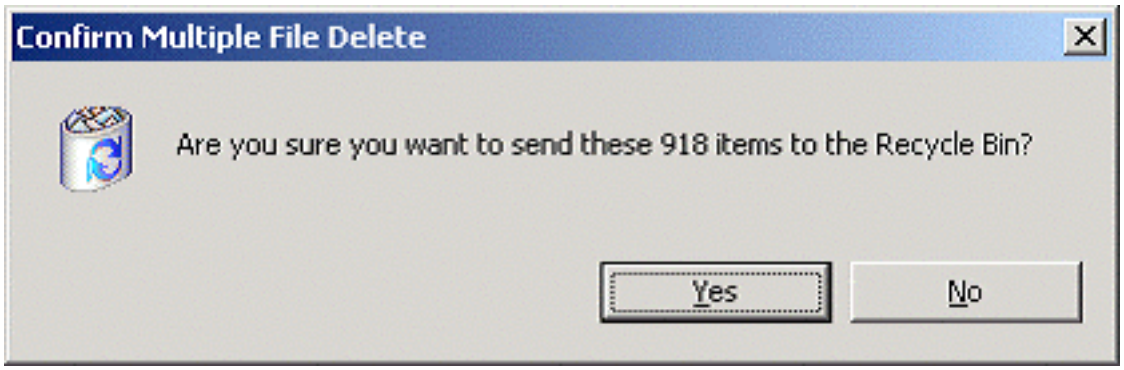
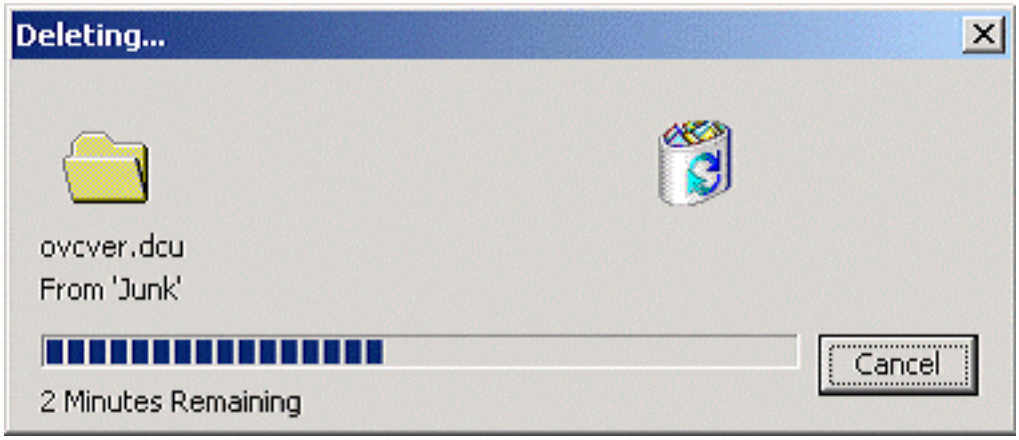


Figure B: *The Windows 2000 Delete Files progress dialog box.*



When the FOF_ALLOWUNDO flag is set, files are sent to the Recycle Bin instead of being permanently deleted. TDeleteFiles sets this flag by default and provides a property, AllowUndo that can be used to set or read this flag.

The FOF_NOCONFIRMATION flag, when set, will cause SHFileOperation() to not show the Confirmation dialog before deleting files. To be safe, TDeleteFiles does not set this flag by default. The property that controls this flag is NoConfirm.

If the FOF_SILENT flag is set, SHFileOperation() will not show the progress dialog while deleting files. SHFIleOperation() can take several seconds to delete a large number of files, so this flag is not set by default. The corresponding property in TDeleteFiles is Silent.

Once all of the members of SHFILEOPSTRUCT are zeroed or are filled with meaningful values, SHFileOperation() is called with a pointer to the SHFILEOPSTRUCT as its only argument. If there is an error, SHFileOperation() returns a nonzero value. The Delete() function reflects the return value of SHFileOperation() by returning true if the operation was successful and false if SHFileOperation() reported an error.

If the user should cancel the operation before SHFileOperation() has completed, the fAnyOperationsAborted member of SHFILEOPSTRUCT is set to true. This information can be accessed through the read-only property UserCancelled in TDeleteFiles.

The file list

As I mentioned, the pFrom member of SHFILEOPSTRUCT will accept a list of files. The list is an array of characters. Each file in the list must be null terminated (“\0”) and the entire list must have a double null termination (“\0\0”).

AnsiString will allow embedded null characters, so I used an AnsiString variable, named fileList, to store the list of files within TDeleteFiles. The Add() function takes a file name as an AnsiString argument and appends it to the end of fileList along with a null character. When the Delete() function is called, it adds the extra null character to the end of fileList before assigning fileList to pFrom using c_str().

The Clear() function empties fileList by setting it to an empty string.

Conclusion and warning

In this series of articles I have shown you how to build a utility that will search for unnecessary project files on your computer and delete them. The code in the utility, both for deleting files and for finding files should be useful for other applications as well.

If you download the example program, you will notice that it is set, by default, to “Preview Only”. Think about it. If you are building an application that has the potential to delete all the files on your computer, you need to be very careful. So, my advice to you is to be very careful. And, have fun.

Listing A: DelFiles.h.

```
#ifndef DelFilesH
#define DelFilesH

class TDeleteFiles
{
  public:
    TDeleteFiles();

    void Add(const String &file);
    void Clear();

    bool Delete();

    __property bool AllowUndo =
      {read = allowUndo, write = allowUndo};
    __property bool NoConfirm =
      {read = noConfirm, write = noConfirm};
    __property bool Silent =
      {read = silent, write = silent};
    __property bool UserCancelled =
      {read = userCancelled};

  private:
    String fileList;

    bool allowUndo, noConfirm;
    bool silent, userCancelled;
};

#endif   // DelFilesH
```

Listing B: DelFiles.cpp.

```
#include <vcl.h>
#pragma hdrstop

#include "DelFiles.h"
```

```
TDeleteFiles::TDeleteFiles()
{
  allowUndo = true;
  noConfirm = false;
  silent = false;
  userCancelled = false;
}

void TDeleteFiles::Add(const String &file)
{
  fileList += file + String('\0');
}

void TDeleteFiles::Clear()
{
  fileList = "";
}

bool TDeleteFiles::Delete()
{
  if (fileList.IsEmpty()) return(true);

  fileList += String('\0');

  SHFILEOPSTRUCT op;
  ZeroMemory(&op, sizeof(SHFILEOPSTRUCT));
  op.hwnd = Application->MainForm->Handle;
  op.wFunc = FO_DELETE;
  if (allowUndo)
    op.fFlags |= FOF_ALLOWUNDO;
  if (noConfirm)
    op.fFlags |= FOF_NOCONFIRMATION;
  if (silent)
    op.fFlags |= FOF_SILENT;
  op.pFrom = fileList.c_str();

  bool error = SHFileOperation(&op) != 0;

  userCancelled = op.fAnyOperationsAborted;

  Clear();
```

```
  return(error);
}

#pragma package(smart_init)
```

Opening and closing the coffee holder

by Mark G. Wiseman

You may have heard the story about a computer user who calls tech support and says that the holder for his coffee mug has broken. When the dismayed technician questions this, the user explains that he just pushes the button on the front of his computer and the coffee mug holder slides out. When he's finished, pushing the button causes the holder to retract back into the computer.

While I wouldn't recommend using your CD-ROM drawer has a coffee mug holder, I will tell you how to open and close the CD-ROM drive through software.

Open sesame

Unlike Ali Baba, you don't need to know magic words to control the CD-ROM door. All you need to know is one of the functions in the Windows Media Control Interface (MCI). This function, `mciSendString()`, is declared in the `mmsystem.h` header file.

The simplest way to open the CD-ROM door is to use `mciSendString()` like this:

```
mciSendString("set cdaudio door open", 0, 0, 0);
```

The first argument to this function, a command string, is the only argument you need to worry about. A minor change to the command string will close the door:

```
mciSendString("set cdaudio door closed", 0, 0, 0);
```

If your computer has only one CD-ROM drive, these two function calls are all you need to know.

However, my computer has both a DVD drive and a CD-RW drive. The first of these two drives will open and close with the function calls above, but the second will not.

Open sesame, please

With just a little more effort, you can open and close any CD-ROM-type drive on a computer.

Using the MCI, you open an interface to the CD-ROM drive in question and give it an

alias of “cd”. Then you send the command string to open or close the door. The final step is to close the interface to the drive. The code below will open the door on the F drive:

```
mciSendString("open f: type cdaudio alias cd", 0, 0, 0);
mciSendString("set cd door open", 0, 0, 0);
mciSendString("close cd", 0, 0, 0);
```

Use this code to close the door on the F drive:

```
mciSendString("open f: type cdaudio alias cd", 0, 0, 0);
mciSendString("set cd door closed", 0, 0, 0);
mciSendString("close cd", 0, 0, 0);
```

Which drives?

If you are wondering how to determine if Windows recognizes a drive as a CD-ROM drive, the code below will populate a combo box, named `DriveCombo`, with a list of CD-ROM drives:

```
DWORD ld = GetLogicalDrives();
for (int i = 0; i < 26; i++)
{
  if ((ld & (1 << i)) != 0)
  {
    String drv =
      String(char('A' + i)) + ":";
    if (GetDriveType(drv.c_str())
        == DRIVE_CDROM)
      DriveCombo->Items->Add(drv);
  }
}
```

Conclusion

Don’t drink too much coffee, don’t set your mug on the CD-ROM door, and don’t forget

to check the example program on the Bridges Publishing Web site. Close sesame.

Resource animations

by Kent Reisdorph

The VCL's `TAnimate` component provides an easy way of adding animations (AVIs) to your applications. Animations are normally played when some process is taking place in your application. With `TAnimate` it's very easy to play an animation from a file on disk. You may, however, want to store your animations in your EXE file rather than lug around a number of animation files. This article will explain how to create and implement an AVI file as a resource.

Creating and playing an animation contained as a resource requires the following steps:

- Create a resource script (RC) file
- Add the RC file to your project
- Write code to play the AVI

I will explain these steps in turn.

Create a resource script file

The first step is to create a resource script file. A resource script file is nothing more than a text file with an extension of .RC. You can create the RC file with any text editor. The C++Builder text editor works fine for this purpose. Simply choose File|New from the C++Builder main menu to invoke the Object Repository and double-click on the Text icon. C++Builder will create a new text file in the Code Editor.

Now that you have a new text file, you need to enter the command that will take an AVI file on disk and turn it into a resource. This part is remarkably simple. Simply type a line similar to this in the blank text file:

```
MyAvi AVI "some.avi"
```

The first parameter on this line is the name or numeric identifier of the resource. If you type a textual name, the parameter will be interpreted as a resource name. If you enter a numeric value, the parameter will be interpreted as a numeric identifier for the

resource. For example:

```
100 AVI "some.avi"
```

You will need to know whether the resource is identified by a resource name or a resource ID when you play the AVI (more on that later).

The second parameter of the resource statement is the type of resource. C++Builder's built in resource compiler understands the type `AVI` as an AVI resource so that's what you type for the second parameter.

The final parameter is the filename that contains the AVI itself. You must have an AVI file on disk before you can convert the AVI into a resource. You can enter the filename with or without double quotes.

That's all there is to creating the resource script file. If you have more than one AVI resource, you should add a line for each AVI you want linked to the application. When you are done, save the file with an extension of .RC. Be sure you change the Save dialog's file filter from "Text file" to "Any file" or you will end up with a filename like MYFILE.RC.TXT.

Note: You have several choices when it comes to creating animations. Corel Draw has an animation editor, as do other commercial drawing packages. Search the Web for other animation editing programs. Note that AVIs created for use with the `TAnimate` component must not contain audio and must be either uncompressed, or RLE compressed only. Any other format will cause the `TAnimate` component to raise an exception at runtime when you attempt to play the animation.

Add the RC file to the project

After you have created and saved the resource script file, you can add it to your project using Project | Add to Project on the main menu. The RC file will be compiled the next time you make or build the project.

The resource compiler will report any errors that occur during the compile. If you encounter errors go back and check your RC file for errors. If the resource compiler successfully compiles the resource script, you will end up with a binary resource file. The binary resource file has the same name as the resource script file but with an extension of .RES. This file will automatically be bound to the executable during the link phase of a make or build.

Writing code to play the AVI

Next you need to write code to play the AVI. When you play an AVI from a file, you simply set the `FileName` property to the name of an AVI file, and then set `Active` to `True`. When you play an animation contained as a resource, however, you need to use the `ResHandle`, `ResName`, and `ResId` properties.

The `ResHandle` property is used to specify the handle of the module that contains the AVI resource. If you have bound the AVI resource to your executable file, you should set `ResHandle` to 0. (In theory you should be able to set the `ResHandle` property to the global `HInstance` variable. For some inexplicable reason, this results in an exception at run time when you attempt to load the AVI resource.) If your animations were contained in a DLL, you would set `ResHandle` to the instance handle of the DLL (obtained with the Windows API function, `LoadLibrary()`).

Next, set either the `ResName` or `ResId` property to the name or resource ID of the AVI resource you wish to play. Use `ResName` if your AVI resource is specified as a text resource name. Use `ResId` if your resource is specified with a numeric value. By the way, the VCL help incorrectly lists this property as `ResID` where it is actually `ResId` (note that the upper case “D” is incorrect).

To play the resource, set the `Active` property to `true`. Alternatively, you can use the `Play()` method to play the animation. The `Play()` method gives you more control over animation playback although it’s slightly more complicated to use.

The following code snippet illustrates playing an AVI resource created with a resource name and bound the application’s executable file:

```
Animate->ResHandle = 0;
Animate->ResName = "TurboGuy";
Animate->Active = true;
```

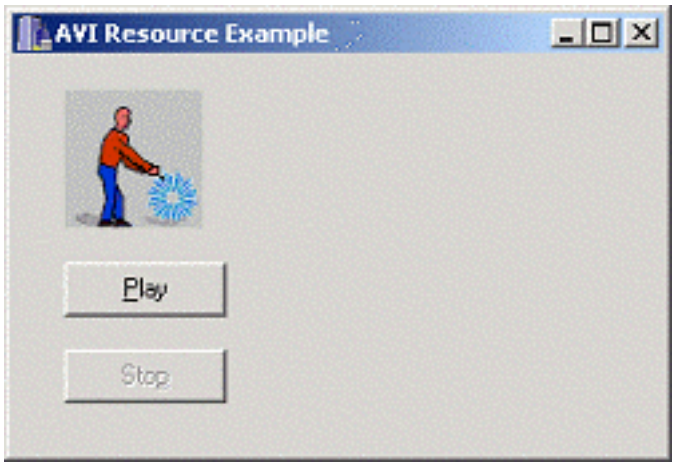
To stop the animation, call the `Stop()` method.

Prove it!

Listing A contains the main unit of a program that plays an AVI video contained as a resource. **Listing B** contains the resource script file used to create the binary resource used in the example program. The program’s main form contains a `TAnimate` component, a Play button, and a Stop button. **Figure A** shows the application playing

the TurboPower “TurboMan” AVI (TurboMan is the AVI that plays when TurboPower’s Memory Sleuth starts running an application). The example program can be downloaded from our Web site.

Figure A



The example program playing a resource AVI.

Listing A: *AviRes.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall TForm1::PlayBtnClick(
  TObject *Sender)
{
  Animate->ResHandle = 0;
  Animate->ResName = "TurboGuy";
  Animate->Active = true;
  PlayBtn->Enabled = false;
  StopBtn->Enabled = true;
}

void __fastcall TForm1::StopBtnClick(
  TObject *Sender)
{
  Animate->Stop();
  PlayBtn->Enabled = true;
  StopBtn->Enabled = false;
}
```

Listing B: *AviResrc.rc*

```
TurboGuy AVI "Start5c.avi"
```

Where is the cursor?

By Kent Reisdorph

Sometimes you need to know where the cursor is on the screen. For an application, the VCL makes this fairly easy by exposing the cursor X and Y coordinates in the `OnMouseMove` event handler. However, there are some cases where the `OnMouseMove` event handler won't be available to you (when writing components or when writing mouse hook callbacks, for example). In those cases you will need to go to the Windows API in order to determine the position of the cursor.

Getting the cursor position

The `GetCursorPos()` Windows API function will tell you the current location of the cursor relative to the desktop. This method takes a pointer to a `POINT` structure as its only parameter. The `x` and `y` members of this structure will contain the X and Y coordinates of the cursor position when `GetCursorPos()` returns. You can substitute a VCL `TPoint` structure for a `POINT` structure if you prefer. The following code gets the cursor position and displays the coordinates in a label:

```
TPoint pt;
GetCursorPos(&pt);
Label3->Caption =
  String(pt.x) + ", " + String(pt.y);
```

The return value of `GetCursorPos()` is nonzero if the call succeeds, or zero if the call fails. I doubt most programmers check the return value of `GetCursorPos()` as the chances of failure are quite low.

Window-relative cursor positions

As I said earlier, `GetCursorPos()` gets the position of the cursor relative to the desktop. It is often necessary to get the cursor position relative to a particular window. If you were writing a visual component, for example, you may need to determine the cursor position relative to the component rather than the desktop.

You can convert screen (desktop) coordinates to window coordinates by calling the `ScreenToClient()` function. Both the VCL and the Windows API have functions by this name, and they differ in their use. The Windows version is declared as follows:

```
BOOL ScreenToClient(
  HWND hWnd, LPPOINT lpPoint);
```

The first parameter is the handle of the window for which the coordinates will be converted. The second parameter is a point to a `POINT` (or `TPoint`) structure. You fill in the `x` and `y` members of this structure prior to calling `ScreenToClient()`, and after the function reuturns, the `POINT` structure will contain the converted values.

The VCL version of `ScreenToClient()` is different two ways. First, it does not have a window handle parameter. It automatically uses the window handle of the window (form or component) of the object. Second, it takes a reference to a `TPoint` structure as a parameter, and returns a new `TPoint` containing the converted coordinates. If you are writing a VCL application the compiler will automatically assume the VCL version of this function because it is in the most immediate scope.

`ScreenToClient()` is most often used in conjunction with `GetCursorPos()`. The following example uses the VCL version of `ScreenToClient()` to obtain the current cursor position relative to the form (assuming this code were called from within a `TForm` descendant):

```
TPoint pt;
GetCursorPos(&pt);
pt = ScreenToClient(pt);
```

Finally, let me add that there is also a `ClientToScreen()` function (both API and VCL). This function can be used in conjunction with `ScreenToClient()` to convert the cursor position of one window to coordinates relative to a second window.

A UTC TDateTime function

By Mark G. Wiseman

If you read part 1 of my series of articles *Hidden treasures of Sysutils*, you know that both the `TDateTime` class and the `Sysutils` namespace in the VCL contain a rich selection of time and date functions.

However, there is one function I recently needed that was not in the VCL. I needed a function that returned a `TDateTime` object with the current system time. In the 32-bit Windows operating systems, the system time is Coordinated Universal Time, not the local time.

What is Coordinated Universal Time and why would I want to use it? Coordinated Universal Time (which for some reason I don't understand is abbreviated UTC) was called Greenwich Mean Time (GMT) until 1972. It is still referred to as "Zulu Time", in some military organizations. UTC is the international standard for representing time.

UTC makes comparing times much easier. In the application where I use UTC, I am logging events in a client/server program. These events are occurring on clients that can be spread across several time zones. In order to understand the correct chronological order of the events all the times are logged as UTC.

As it turns out, returning the UTC time as a `TDateTime` object was fairly easy after I did some investigative work. I found that the Windows API function, `GetSystemTime()`, fills a `SYSTEMTIME struct` with the system date and time. Unfortunately there is no VCL equivalent that returns this time in a `TDateTime` object. However, the VCL function, `SystemTimeToDateTime()`, converts a `SYSTEMTIME struct` to a `TDateTime` object. This was all I needed to create the `SystemDateTime()` function. Here's how that function looks:

```
TDateTime SystemDateTime()
{
  SYSTEMTIME systemTime;
  GetSystemTime(&systemTime);
  return (SystemTimeToDateTime(systemTime));
}
```

This simple function allows me to deal with UTC times in a familiar `TDateTime` object.

If you have a further interest in UTC and date and time standards, here are two Web sites you should look at:

www.cl.cam.ac.uk/~mgk25/iso-time.html

www.iso.ch/markete/8601.pdf

A Summary of the International Standard Date and Time Notation

by Markus Kuhn

International Standard ISO 8601 specifies numeric representations of date and time. This standard notation helps to avoid confusion in international communication caused by the many different national notations and increases the portability of computer user interfaces. In addition, these formats have several important advantages for computer usage compared to other traditional date and time notations. The time notation described here is already the de-facto standard in almost all countries and the date notation is becoming increasingly popular.

Especially authors of Web pages and software engineers who design user interfaces, file formats, and communication protocols should be familiar with ISO 8601.

Contents:

NEWS: The second edition ISO 8601:2000 has been published!

Date

The international standard date notation is

YYYY-MM-DD

where YYYY is the year in the usual Gregorian calendar, MM is the month of the year between 01 (January) and 12 (December), and DD is the day of the month between 01 and 31.

For example, the fourth day of February in the year 1995 is written in the standard notation as

1995-02-04

Other commonly used notations are e.g. 2/4/95, 4/2/95, 95/2/4, 4.2.1995, 04-FEB-1995, 4-February-1995, and many more. Especially the first two examples are dangerous, because as both are used quite often in the U.S. and in Great Britain and both can not be distinguished, it is unclear whether 2/4/95 means 1995-04-02 or 1995-02-04. The date notation 2/4/5 has at least six reasonable interpretations (assuming that only the twentieth and twenty-first century are reasonable candidates in our life time).

Advantages of the ISO 8601 standard date notation compared to other commonly used variants:

- easily readable and writeable by software (no 'JAN', 'FEB', ... table necessary)
- easily comparable and sortable with a trivial string comparison
- language independent
- can not be confused with other popular date notations
- consistency with the common 24h time notation system, where the larger units (hours) are also written in front of the smaller ones (minutes and seconds)
- strings containing a date followed by a time are also easily comparable and sortable (e.g. write "1995-02-04 22:45:00")
- the notation is short and has constant length, which makes both keyboard data entry and table layout easier
- identical to the Chinese date notation, so the largest cultural group (>25%) on this planet is already familiar with it :-)
- date notations with the order "year, month, day" are in addition already widely used e.g. in Japan, Korea, Hungary, Sweden, Finland, Denmark, and a few other countries and people in the U.S. are already used to at least the "month, day" order
- a 4-digit year representation avoids overflow problems after 2099-12-31

As dates will look a little bit strange anyway starting with 2000-01-01 (e.g. like 1/1/0), it has been suggested that the year 2000 is an excellent opportunity to change to the standard date notation.

ISO 8601 is only specifying numeric notations and does not cover dates and times where words are used in the representation. It is not intended as a replacement for language-dependent worded date notations such as "24. Dezember 2001" (German) or "February 4, 1995" (US English). ISO 8601 should however be used to replace notations such as "2/4/95" and "9.30 p.m.".

Apart from the recommended primary standard notation **YYYY-MM-DD**, ISO 8601 also specifies a number of alternative formats for use in applications with special requirements. All of these alternatives can easily and automatically be distinguished from each other:

The hyphens can be omitted if compactness of the representation is more important than human readability, for example as in

19950204

For situations where information about the century is really not required, a 2-digit year representation is available:

95-02-04 or **950204**

If only the month or even only the year is of interest:

1995-02 or **1995**

In commercial and industrial applications (delivery times, production plans, etc.), especially in Europe, it is often required to refer to a week of a year. Week 01 of a year is per definition the first week that has the Thursday in this year, which is equivalent to the week that contains the fourth day of January. In other words, the first week of a new year is the week that has the majority of its days in the new year. Week 01 might also contain days from the previous year and the week before week 01 of a year is the last week (52 or 53) of the previous year even if it contains days from the new year. A week starts with Monday (day 1) and ends with Sunday (day 7). For example, the first week of the year 1997 lasts from 1996-12-30 to 1997-01-05 and can be written in standard notation as

1997-W01 or **1997W01**

The week notation can also be extended by a number indicating the day of the week. For example, the day 1996-12-31, which is the Tuesday (day 2) of the first week of 1997, can also be written as

1997-W01-2 or **1997W012**

for applications like industrial planning where many things like shift rotations are organized per week and knowing the week number and the day of the week is more handy than knowing the day of the month.

An abbreviated version of the year and week number like

95W05

is sometimes useful as a compact code printed on a product that indicates when it has been manufactured.

The ISO standard avoids explicitly stating the possible range of week numbers, but this can easily be deduced from the definition:

> **Theorem:** Possible ISO week numbers are in the range 01 to 53. A year always has a week 52. (There is one historic exception: the year in which the Gregorian calendar was introduced had less than 365 days and less than 52 weeks.)
>
> **Proof:** Per definition, the first week of a year is W01 and consequently days before week W01 belong to the previous year and so there is no week with lower numbers. Considering the highest possible week number, the worst case is a leap year like 1976 that starts with a Thursday, because this keeps the highest possible number of days of W01 in the previous year, i.e. 3 days. In this case, the Sunday of W52 of the worst case year is day number 4+51*7=361 and 361-366=5 days of W53 belong still to this year, which guarantees that in

> the worst case year day 4 (Thursday) of W53 is not yet in the next year, so a week number 53 is possible. For example, the 53 weeks of the worst case year 1976 started with 1975-12-29 = 1976-W01-1 and ended with 1977-01-02 = 1976-W53-7. On the other hand, considering the lowest number of the last week of a year, the worst case is a non-leap year like 1999 that starts with a Friday, which ensures that the first three days of the year belong to the last week of the previous year. In this case, the Sunday of week 52 would be day number 3+52*7=367, i.e. only the last 367-365=2 days of the W52 reach into the next year and consequently, even a worst case year like 1999 has a week W52 including the days 1999-12-27 to 2000-01-02. q.e.d.

[The new 1999 version of the C programming language standard (ISO 9899) added in the `strftime()` function means to generate the ISO 8601 week notation. The author of this text developed a further proposal for a modernised clock and calendar API for C, which provides full proper treatment of leap seconds and timezones and fixes numerous other problems in the current C timing library functions. It also serves as a model for those who want to design clock library functions for other programming languages.]

Both day and year are useful units of structuring time, because the position of the sun on the sky, which influences our lives, is described by them. However the 12 months of a year are of some obscure mystic origin and have no real purpose today except that people are used to having them (they do not even describe the current position of the moon). In some applications, a date notation is preferred that uses only the year and the day of the year between 001 and 365 (366 in leap years). The standard notation for this variant representing the day 1995-02-04 (that is day 035 of the year 1995) is

> **1995-035** or **1995035**

Leap years are years with an additional day YYYY-02-29, where the year number is a multiple of four with the following exception: If a year is a multiple of 100, then it is only a leap year if it is also a multiple of 400. For example, 1900 was not a leap year, but 2000 is one.

Time of Day

The international standard notation for the time of day is

> **hh:mm:ss**

where hh is the number of complete hours that have passed since midnight (00-24), mm is the number of complete minutes that have passed since the start of the hour (00-59), and ss is the number of complete seconds since the start of the minute (00-60). If the hour value is 24, then the minute and second values must be zero. [The value 60 for ss might sometimes be needed during an inserted leap second in an atomic time scale like Coordinated Universal Time (UTC). A single leap second 23:59:60 is inserted into

the UTC time scale every few years as announced by the International Earth Rotation Service in Paris to keep UTC from wandering away more than 0.9 s from the less constant astronomical time scale UT1 that is defined by the actual rotation of the earth.]

An example time is

23:59:59

which represents the time one second before midnight.

As with the date notation, the separating colons can also be omitted as in

235959

and the precision can be reduced by omitting the seconds or both the seconds and minutes as in

23:59, **2359**, or **23**

It is also possible to add fractions of a second after a decimal dot or comma, for instance the time 5.8 ms before midnight can be written as

23:59:59.9942 or **235959.9942**

As every day both starts and ends with midnight, the two notations **00:00** and **24:00** are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time:

1995-02-04 24:00 = **1995-02-05 00:00**

In case an unambiguous representation of time is required, 00:00 is usually the preferred notation for midnight and not 24:00. Digital clocks display 00:00 and not 24:00.

ISO 8601 does not specify, whether its notations specify a point in time or a time period. This means for example that ISO 8601 does not define whether 09:00 refers to the exact end of the ninth hour of the day or the period from 09:00 to 09:01 or anything else. The users of the standard must somehow agree on the exact interpretation of the time notation if this should be of any concern.

If a date and a time are displayed on the same line, then always write the date in front of the time. If a date and a time value are stored together in a single data field, then ISO 8601 suggests that they should be separated by a latin capital letter T, as in **19951231T235959**.

A remark for readers from the U.S.:

The 24h time notation specified here has already been the de-facto standard all over the world in written language for decades. The only exception are a few English speaking countries, where still notations with hours between 1 and 12 and additions like "a.m." and "p.m." are in wide use. The common 24h international standard notation is widely used now even in England (e.g. at airports, cinemas, bus/train timetables, etc.). Most other languages don't even have abbreviations like "a.m." and "p.m." and the 12h notation is certainly hardly ever used on Continental Europe to write or display a time. Even in the U.S., the military and computer programmers have been using the 24h notation for a long time.

The old English 12h notation has many disadvantages like:

- It is longer than the normal 24h notation.
- It takes somewhat more time for humans to compare two times in 12h notation.
- It is not clear, how 00:00, 12:00 and 24:00 are represented. Even encyclopedias and style manuals contain contradicting descriptions and a common quick fix seems to be to avoid "12:00 a.m./p.m." altogether and write "noon", "midnight", or "12:01 a.m./p.m." instead, although the word "midnight" still does not distinguish between 00:00 and 24:00 (midnight at the start or end of a given day).
- It makes people often believe that the next day starts at the overflow from "12:59 a.m." to "1:00 a.m.", which is a common problem not only when people try to program the timer of VCRs shortly after midnight.
- It is not easily comparable with a string compare operation.
- It is not immediately clear for the unaware, whether the time between "12:00 a.m./p.m." and "1:00 a.m./p.m." starts at 00:00 or at 12:00, i.e. the English 12h notation is more difficult to understand.

Please consider the 12h time to be a relic from the dark ages when Roman numerals were used, the number zero had not yet been invented and analog clocks were the only known form of displaying a time. Please avoid using it today, especially in technical applications! Even in the U.S., the widely respected *Chicago Manual of Style* now recommends using the international standard time notation in publications.

A remark for readers from German speaking countries:

The German standard DIN 5008, which specifies typographical rules for German texts written on typewriters, was updated in 1996-05. The old German numeric date notations DD.MM.YYYY and DD.MM.YY have been replaced by the ISO date notations YYYY-MM-DD and YY-MM-DD. Similarly, the old German time notations hh.mm and hh.mm.ss have been replaced by the ISO notations hh:mm and hh:mm:ss. Those new notations are

now also mentioned in the latest edition of the *Duden*. The German alphanumeric date notation continues to be for example "3. August 1994" or "3. Aug. 1994". The corresponding Austrian standard has already used the ISO 8601 date and time notations before.

ISO 8601 has been adopted as European Standard EN 28601 and is therefore now a valid standard in all EU countries and all conflicting national standards have been changed accordingly.

Time Zone

Without any further additions, a date and time as written above is assumed to be in some local time zone. In order to indicate that a time is measured in Universal Time (UTC), you can append a capital letter **Z** to a time as in

23:59:59Z or **2359Z**

[The Z stands for the "zero meridian", which goes through Greenwich in London, and it is also commonly used in radio communication where it is pronounced "Zulu" (the word for Z in the international radio alphabet). Universal Time (sometimes also called "Zulu Time") was called Greenwich Mean Time (GMT) before 1972, however this term should no longer be used. Since the introduction of an international atomic time scale, almost all existing civil time zones are now related to UTC, which is slightly different from the old and now unused GMT.]

The strings

+hh:mm, **+hhmm**, or **+hh**

can be added to the time to indicate that the used local time zone is hh hours and mm minutes ahead of UTC. For time zones west of the zero meridian, which are behind UTC, the notation

-hh:mm, **-hhmm**, or **-hh**

is used instead. For example, Central European Time (CET) is +0100 and U.S./Canadian Eastern Standard Time (EST) is -0500. The following strings all indicate the same point of time:

12:00Z = **13:00+01:00** = **0700-0500**

There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. In addition, politicians enjoy modifying the rules for civil time zones, especially for daylight saving times, every few

years, so the only really reliable way of describing a local time zone is to specify numerically the difference of local time to UTC. Better use directly UTC as your only time zone where this is possible and then you do not have to worry about time zones and daylight saving time changes at all.

More Information about Time Zones

Arthur David Olson and others maintain a database of all current and many historic time zone changes and daylight saving time algorithms. It is available via ftp from elsie.nci.nih.gov in the `tzcode*` and `tzdata*` files. Most Unix time zone handling implementations are based on this package. If you want to join the `tz` mailing list, which is dedicated to discussions about time zones and this software, please send a request for subscription to tz-request at elsie.nci.nih.gov. You can read previous discussion there in the tz archive.

Other Links about Date, Time, and Calendars

Some other interesting sources of information about date and time on the Internet are for example the Glossary of Frequency and Timing Terms and the FAQ provided by NIST, the Yahoo Science:Measurements and Units:Time link collection, the U.S. Naval Observatory Server, the International Earth Rotation Service (IERS) (for time gurus only!), the University of Delaware NTP Time Server, the time and calendar section of the USENET sci.astro FAQ, and the Calendar FAQ.

This was a brief overview of the ISO 8601 standard, which covers only the most useful notations and includes some additional related information. The full standard defines in addition a number of more exotic notations including some for periods of time. The ISO 8601:2000 standard itself can only be ordered on paper or as a PDF file on CD-ROM either via ISO's web site online or from

> International Organization for Standardization
> Case postale 56
> 1, rue de Varembé
> CH-1211 Genève 20
> Switzerland
>
> phone: +41 22 749 01 11
> fax: +41 22 733 34 30
> email: sales at isocs.iso.ch

A more detailed online summary of ISO 8601 than this one is the text *ISO 8601:1988 Date/Time Representations* available from ftp.informatik.uni-erlangen.de/pub/doc/ISO/ISO8601.ps.Z (PostScript,

16 kb, 5 pages) written by Gary Houston, now also available in HTML. Ian Galpin (G1SMD) proposes to use ISO 8601 as a Common Date-Time Standard for Amateur Radio. Steve Adams has written another web page about the ISO date format that is partially based on this text. Another summary of ISO 8601 is Jukka Korpela's page and there are further related pages listed in the Open Directory.

The committee in charge of ISO 8601 is ISO TC 154 and the editor of the second edition ISO 8601:2000 was Louis Visser.

I wish to thank Edward M. Reingold for developing the fine GNU Emacs calendar functions, as well as Rich Wales, Mark Brader, Paul Eggert, and others in the comp.std.internat, comp.protocols.time.ntp, and sci.astro USENET discussion groups for valuable comments about this text. Further comments and hyperlinks to this page are very welcome.

Some journalists recently got interested in the international date and time format and reported about it. Examples include:

- An article by Jon G. Auerbach in the 1999-06-01 issue of the Wall Street Journal, page A1.

If you are a journalist and need information on this or related topics, please feel free to contact me.

You might also be interested in the International Standard Paper Sizes Web page.

Markus Kuhn

created 1995 -- last modified 2001-11-10 -- http://www.cl.cam.ac.uk/~mgk25/iso-time.html

Finding files, part 3

By Mark G. Wiseman

If you've read parts 1 and 2 in this series of articles, you know I am building a program that will delete all the unnecessary files created by C++Builder during the lifetime of a project.

I created a class, `TFindFile`, and some support classes that I was going to use to search for these files on my hard drives. In this article, I had planned to discuss how I was going to use `TFindFile` to actually find and delete files. However, I made some discoveries along the way that caused me to alter the original design.

Best laid plans…

In my original design, I was going to create another class, `TMultiFindFile`, which used `TFindFile` to match multiple file patterns in one pass through a hard drive. The `TFindFile` class can only match one file pattern per pass, so I had planned to use `TFindFile` to match all files using the *.* pattern and then pick the correct files to delete with `TMultiFindFile`. I designed these classes for the greatest flexibility. `TFindFile` would be very fast but not versatile and `TMultiFindFile` would be versatile but not very fast. I thought I might need the speed of `TFindFile` in a future project, so it made sense to have two classes.

However, when I tested `TMultiFindFile`, I was surprised to discover that it, like `TFindFile`, was very fast. What I had was a level of complexity that I no longer needed. So, I have rewritten `TFindFile` to include the abilities of `TMultiFileFind` and to avoid confusion, I have named this new hybrid class `TFileFinder`.

Powerful and fast

If you've looked at the code for `TFileFind`, most of `TFileFinder` will look familiar. **Listings A and B** contain the source for `TFileFinder`. I still use the `TFFStack` class to avoid using recursion while searching subfolders. And I still use an event to report when a file is found that matches a specified pattern.

So what has changed? Well, `TFileFinder` will accept a list of patterns to match and a list of patterns to exclude. It will also accept a list of paths to search and a list of paths to skip or not search.

If you compare the public interface of the old `TFindFile` and the new `TFileFinder`, you will see only a few changes. I have added five properties and deleted one property. Also, the `Find()`

method no longer takes any arguments.

Paths

Four of the properties I added reflect the power of `TFileFinder`. These properties are `Paths`, `SkipPaths`, `Include`, and `Exclude`. `Paths` and `SkipPaths` are of type `TFFPathList`. I derived `TFFPathList` from `TStringList`. The source for `TFFPathList` can be found in **Listing C**. The sole purpose of `TFFPathList` is to store a list of path strings with a `bool`, `SearchSubfolders`, attached that indicates whether the path's subfolders should be searched.

Using `Paths` and `SkipPaths` is easy. All of my C++Builder projects can be found in subfolders of two folders on my C drive, C:\PROJECTS and C:\ARTICLES. I want to search these folders for files to delete; but I don't want to search the folder C:\PROJECTS\CURRENT or any of its subfolders. The following code will set up `TFileFinder` to do this:

```
Paths->Add("c:\\Projects", true);
Paths->Add("c:\\Articles", true);
SkipPaths->Add("c:\\Projects\\Current", false);
```

I call `Add()` to add the C:\PROJECTS path to Paths and also pass in `true` to indicate that the subfolders of C:\PROJECTS should also be searched. I add C:\PROJECTS\CURRENT to the list of paths that `TFileFinder` should skip while searching. In this case, I pass in `false` to indicate that the subfolders of C:\PROJECTS\CURRENT should not be searched. Had I passed in `true`, C:\PROJECTS\CURRENT would not be searched, but all of its subfolders would be searched.

I dropped the property `SearchSubfolders` used in the original `TFindFile` class, since each path now carries a `bool` to indicate the search method.

Patterns

The `Include` and `Exclude` properties of `TFileFinder` are of type `TStringList` and are lists of file patterns to either include or exclude when matching files.

If I want to match files that have the extensions .TDS and .CSM but only if those files don't start with the letter 'f', I could use the following code:

```
Include->Add("*.tds");
Include->Add("*.csm");
Exclude->Add("f*.*");
```

When the `FindFiles()` method of `TFileFinder` does its work, it first checks to see if a file

matches any of the patterns in the `Include` list. If the file does match, `FindFiles()` checks to see if it also matches any of the patterns in the `Exclude` list and if it does the file is not reported as a match.
I use the VCL function, `MatchesMask()`, to compare file names and patterns. This function uses a subset of regular expressions, so you can create some sophisticated patterns to match file names against. Check the VCL online help for `MatchesMask()` to learn more.

Matching folders
Normally, when searching a hard drive for files, you won't want `TFileFinder` to report matches on folder names in addition to file names. But, you might; so I have included the property `MatchFolders`. If this property, a `bool`, is true, the `OnFileFound` event will also report any folder names that match.

Finding files
Finally, the last change I made when creating `TFileFinder` was to drop the pattern argument from the `Find()` function originally use in `TFileFind`. The `Paths`, `SkipPaths`, `Include` and `Exclude` properties of `TFileFinder` eliminate the need for an argument to `Find()`.

Conclusion
There is a test program for `TFileFinder` on the Bridges Publishing Web site at **www.bridgespublishing.com**. You can use it to see how the `Paths`, `SkipPaths`, `Include` and `Exclude` properties work.

In part 2, I told you that this article would finally delete some files. Well, it doesn't. I do think, though, that the very powerful `TFileFinder` class is worth making you wait until part 4, where I really will delete some files.

Listing A: *FileFinder.h*

```
#ifndef FileFinderH
#define FileFinderH

#include "FFData.h"
#include "FFPathList.h"

class TFFStack;
class TFileFinder;


typedef void __fastcall (__closure TFileFound)
```

```
  (TFileFinder *Sender, String fileName, String foldername, TFFData
data);
typedef void __fastcall (__closure *TSearchFolder)
  (TFileFinder *Sender, String folderName, bool &skip);


class TFileFinder
{
  public:
    TFileFinder();
    ~TFileFinder();

    void Find();
    void Stop();
    bool IsRunning();

    __property TStringList *Include = {read = include, write =
SetInclude};
    __property TStringList *Exclude = {read = exclude, write =
SetExclude};
    __property TFFPathList *Paths = {read = paths, write = SetPaths};
    __property TFFPathList *SkipPaths = {read = skipPaths, write =
SetSkipPaths};

    __property bool MatchFolders = {read = matchFolders, write =
matchFolders};
    __property TFileFound OnFileFound = {read = onFileFound, write =
onFileFound};
    __property TSearchFolder OnSearchFolder = {read=onSearchFolder,
write=onSearchFolder};


  private:
    void FindFiles(String filePattern);
    void FindDirs(String baseDir, TFFStack &stack);
    bool SkipDir(const String dir);

    void __fastcall SetInclude(TStringList *val);
    void __fastcall SetExclude(TStringList *val);
    void __fastcall SetPaths(TFFPathList *val);
    void __fastcall SetSkipPaths(TFFPathList *val);

    TFileFound onFileFound;
    TSearchFolder onSearchFolder;
```

```
    TFFPathList *paths, *skipPaths;
    TStringList *include, *exclude;

    bool stop;
    bool matchFolders;
};


inline void TFileFinder::Stop()
{
  stop = true;
}

inline bool TFileFinder::IsRunning()
{
  return(stop == false);
}


#endif   // FileFinderH
```

Listing B: *FileFinder.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include <Masks.hpp>

#include "FileFinder.h"
#include "FFStack.h"

TFileFinder::TFileFinder()
{
  onFileFound = 0;
  onSearchFolder = 0;

  include = exclude = 0;
  paths = skipPaths = 0;

  try {
```

```
    include = new TStringList;
    exclude = new TStringList;
    paths = new TFFPathList;
    skipPaths = new TFFPathList;
  }
  catch(...) {
    delete include;
    delete exclude;
    delete paths;
    delete skipPaths;

    throw;
  }

  stop = true;
  matchFolders = false;
}

TFileFinder::~TFileFinder()
{
  delete include;
  delete exclude;
  delete paths;
  delete skipPaths;
}

bool TFileFinder::SkipDir(const String dir)
{
  bool skip = false;

  for (int i=0;i<skipPaths->Count && skip == false;i++)
  {
    skip = dir.AnsiCompareIC(
      IncludeTrailingBackslash(skipPaths->Strings[i])) == 0 ||
        (dir.Pos(skipPaths->Strings[i]) == 1 &&
        skipPaths->SearchSubfolders[i] == false);
  }

  if (onSearchFolder)
    onSearchFolder(this, dir, skip);

  return(skip);
```

```
}

void TFileFinder::Find()
{
  stop = false;

  TFFStack dirStack;

  for (int i=0;i<paths->Count && stop == false;i++)
  {
    String curDir = IncludeTrailingBackslash(ExpandFileName(paths-
>Strings[i]));

    if (onFileFound && SkipDir(curDir) == false)
      FindFiles(curDir);

    if (paths->SearchSubfolders[i]) {
      while (stop == false) {
        Application->ProcessMessages();

        FindDirs(curDir, dirStack);
        if (dirStack.IsEmpty())
          break;

        curDir = dirStack.Pop();

        if (SkipDir(curDir))
          continue;

        if (onFileFound)
          FindFiles(curDir);
      }
    }
  }

  stop = true;
}

void TFileFinder::FindFiles(String dir)
{
  TFFData ffData;
```

```
  String filePattern = dir + "*.*";

  HANDLE handle = FindFirstFile(filePattern.c_str(),
&ffData.data);
  if (handle != INVALID_HANDLE_VALUE) {
    bool found = true;

    while (found == true && stop == false) {
      Application->ProcessMessages();

      if (ffData.IsFolder() == false || matchFolders == true) {
        String name = ffData.GetName();

        bool match = false;
        for (int i=0;i<include->Count && match == false;i++)
          match = MatchesMask(name, include->Strings[i]);

        for (int i=0;i<exclude->Count && match == true;i++)
          match = !MatchesMask(name, exclude->Strings[i]);

        if (match == true && name != "." && name != "..")
          onFileFound(this, name, dir, ffData);
      }

      found = (FindNextFile(handle, &ffData.data) != 0);
    }

    FindClose(handle);
  }
}

void TFileFinder::FindDirs(String baseDir, TFFStack &stack)
{
  TFFData ffData;

  String dirPattern = baseDir + "*.*";

  HANDLE handle = FindFirstFile(dirPattern.c_str(), &ffData.data);
  if (handle != INVALID_HANDLE_VALUE) {
    bool found = true;

    while (found == true && stop == false) {
```

```
      Application->ProcessMessages();

      if (ffData.IsFolder() && ffData.GetName() != "." &&
ffData.GetName() != "..")
        stack.Push(baseDir + ffData.GetName() + "\\");

      found = (FindNextFile(handle, &ffData.data) != 0);
    }

    FindClose(handle);
  }
}

void __fastcall TFileFinder::SetInclude(TStringList *val)
{
  include->Assign(val);
}

void __fastcall TFileFinder::SetExclude(TStringList *val)
{
  exclude->Assign(val);
}

void __fastcall TFileFinder::SetPaths(TFFPathList *val)
{
  paths->Assign(val);
}

void __fastcall TFileFinder::SetSkipPaths(TFFPathList *val)
{
  skipPaths->Assign(val);
}
```

Listing C: FFPathList.h.

```
#pragma package(smart_init)

#ifndef FFPathListH
#define FFPathListH

#include <vcl.h>
#pragma hdrstop
```

```
class TFFPathList : public TStringList
{
  public:
    virtual int __fastcall Add(const String path);
    virtual int __fastcall Add(const String path, bool
searchSubfolders);

    __property bool SearchSubfolders[int index] =
      {read = GetSearchSubfolders, write = SetSearchSubfolders};

  private:
    bool __fastcall GetSearchSubfolders(int index);
    void __fastcall SetSearchSubfolders(int index, bool search);

    __property System::TObject* Objects[int Index]= {read=GetObject,
write=PutObject};
    virtual int __fastcall AddObject(const AnsiString
S,System::TObject* AObject);
};

inline int __fastcall TFFPathList::Add(const String path)
{
  return(TStringList::Add(path));
}

inline int __fastcall TFFPathList::Add(const String path, bool
searchSubfolders)
{
  return(AddObject(path, (TObject *)searchSubfolders));
}

inline bool __fastcall TFFPathList::GetSearchSubfolders(int index)
{
  return((bool)Objects[index]);
}

inline void __fastcall TFFPathList::SetSearchSubfolders(int index,
bool search)
{
  Objects[index] = (TObject *)search;
}

inline int __fastcall TFFPathList::AddObject(
  const AnsiString S, System::TObject* AObject)
```

```
{
  return(TStringList::AddObject(S, AObject));
}


#endif // FFPathListH
```

How to determine the drive type

By Kent Reisdorph

It is often necessary to determine the types of the drives on a user's machine. You may, for example, need to know which drive letter is associated with the CD-ROM drive, or how many CD-ROM drives are installed on a particular machine. Another scenario is that you may need to determine which of the drives on a system are hard drives, and which are drives containing removable media (a floppy drive, for example). Determining the drive type is fairly easy, as this article will show.

The GetDriveType() function

The Windows API includes a function, `GetDriveType()`, that tells you the type of each drive on a particular machine. The declaration for `GetDriveType()` is as follows:

```
UINT GetDriveType(
LPCTSTR lpRootPathName);
```

This method is fairly straightforward. You pass the drive letter of the drive to check, and `GetDriveType()` returns the type of the drive. The string passed to `GetDriveType()` is the drive letter with a trailing colon. You can pass the trailing backslash as well, but it isn't necessary. The following two lines of code, then, are functionally equivalent:

```
int type;
type = GetDriveType("A:");
type = GetDriveType("A:\\");
```

It is important to realize that any combination of characters other than the above will fail. For example, the following two lines of code will each return `1`, indicating that the drive does not exist:

```
int type;
type = GetDriveType("C");
type = GetDriveType("C:\\Windows");
// the value of type will be 1
```

You can pass `NULL` to `GetDriveType()`, in which case the drive type of the current drive will be returned.

Obviously, calling `GetDriveType()` is trivial. The rest of the work comes in interpreting the return value.

Return values

`GetDriveType()` returns an integer. This integer is a code that tells you the drive type. **Table A** shows the possible values.

Table A: Return values from GetDriveType()

Value	Description
`0`	Drive type cannot be determined.
`1`	Drive does not exist.
`DRIVE_REMOVABLE`	Drive has removable media.
`DRIVE_FIXED`	Drive is a hard drive.
`DRIVE_REMOTE`	Drive is a network drive.
`DRIVE_CDROM`	Drive is a CD-ROM drive.
`DRIVE_RAMDISK`	Drive is a RAM disk.

A return value of `0` indicates that the drive type cannot be determined. I have yet to see this value returned, so I can't say with any certainty what set of circumstances will lead to this return value. If the drive is not mapped (i.e. doesn't exist), `1` is returned.

A return value of `DRIVE_REMOVABLE` means that the drive has removable media. This can be a bit misleading. A CD-ROM drive, for example, obviously has removable media. However, a CD-ROM drive will return `DRIVE_CDROM` and not `DRIVE_REMOVABLE`. Other removable drive types include some large-capacity removable drives, such as ZIP and Jazz drives. These drive types will generally return `DRIVE_REMOVABLE`.

If `GetDiskType()` returns `DRIVE_REMOTE`, the drive is a mapped network drive and if it returns `DRIVE_RAMDISK` the drive is a RAM disk.

A simple example

The following example enumerates all drives (from A: to Z:). If the drive exists, the drive letter and type are added to a memo on the form. Here is that code:

```
void __fastcall TForm1::Button1Click(
  TObject *Sender)
{
  for (char c='A';c<='Z';c++) {
    String S = String(c) + ":";
    int type = GetDriveType(S.c_str());
    if (type > 1) {
      String TypeStr;
      switch (type) {
        case DRIVE_REMOVABLE :
          TypeStr = " is removable";
          break;
        case DRIVE_FIXED :
          TypeStr = " is fixed";
          break;
        case DRIVE_REMOTE :
          TypeStr = " is network drive";
          break;
        case DRIVE_CDROM :
          TypeStr = " is CD-ROM";
          break;
        case DRIVE_RAMDISK :
          TypeStr = " is RAM disk";
          break;
      }
      Memo1->Lines->Add(S + TypeStr);
    }
  }
}
```

Conclusion

Getting the drive type is simple once you know about the `GetDriveType()` function. A related question is obviously, “Is there media in the removable media drive.” The answer to that question will have to wait for a later article.

Printing bitmaps, part I

By Damon Chandler

“How do a I print a bitmap?” This question is almost as ubiquitous in the Borland forums as “How do I assign an event handler to a dynamically-created control?” Unfortunately, while the answer to the latter is simple, printing a bitmap is tricky business.

Originally, I had planned to cover the topic of printing bitmaps in a single article. The problem is that this subject requires knowledge of the different bitmap types supported by Windows. As such, I’ve decided to spread the topic of printing bitmaps over a series of articles. In this first installment, I’ll get you up to speed with the different bitmaps types you are likely to encounter when printing. This article will not actually get as far as printing bitmaps, but will lay the necessary foundation. Later in this series I’ll cover topics such as printing, banding, proofing, and Image Color Management (ICM).

Device-dependent bitmaps

In Windows there are three types of bitmaps: device-dependent bitmaps (DDBs), device-independent bitmaps (DIBs), and DIB section bitmaps. Let’s first examine the simplest case, the DDB.

A DDB is a true GDI graphic object. You can select a DDB into a (memory) device context, you can use the `GetObject()` function with a DDB, and there’s a dedicated type, `HBITMAP`, which is used to store a handle to a DDB. For this reason, you’ll often see the term “bitmap object” used to refer to a DDB.

DDBs are, of course, device dependent. Internally, a DDB is stored in a format known only to the device driver. (Note that this is not the case with monochrome DDBs.) Because the format of a (color) DDB is specific to a particular device, DDBs are also commonly referred to as “compatible bitmaps”.
To create a DDB, you use the `CreateCompatibleBitmap()` function. It is declared as follows:

```
HBITMAP CreateCompatibleBitmap(hdc, nWidth, nHeight);
```

The `nWidth` and `hHeight` parameters specify the width and the height of the DDB, respectively. The `hdc` parameter is a handle to a device context; you use this handle to tell the `CreateCompatibleBitmap()` function the device with which you want the DDB to be compatible. In turn, the `CreateCompatibleBitmap()` function will instruct the corresponding device driver to create the DDB. Again, there’s no dogma as to how the device driver should

create and store this DDB.

There is an advantage to a DDB's device-dependence: because there's no conversion involved, drawing a DDB to its compatible device is extremely fast. This is, however, the only advantage. Because a DDB's format is proprietary, you can't access a DDB's pixels, nor can you reliably blit (copy) a DDB to another device. This is the main reason printing bitmaps is so problematic—before you can print, you need to convert your DDB into a universal format that all device drivers can understand. This universal format is known as a DIB, a device-independent bitmap that's understood by any device context.

Device-independent bitmaps

Before we get into the specifics of DIBs, there are a few things that you should keep in mind. The first thing to note is that a DIB is not a true GDI graphic object. You can't select a DIB into a device context, nor are there any functions for creating a DIB from scratch. The second thing to note is that there are no functions for drawing to a DIB. Remember, you can't select a DIB into a device context, so you can't use any GDI drawing function to draw anything to a DIB. There are, however, a couple of functions that you can use to render a DIB to a device. Namely, you can use the `StretchDIBits()` or the `SetDIBitsToDevice()` function to transfer a DIB to a device.

A universal format

I have established that a DIB is not a GDI graphic object. What, then, is a DIB? Simply put, it's a composite entity that's composed of three parts: a header, an optional color table, and an array of pixels. While this format this not truly universal, it is well known. This layout is depicted in **Figure A**.

Figure A*: A header, a color table, and an array of pixels define a DIB.*

Note from **Figure A** that the image appears upside-down. As it turns out, the pixels of a DIB are often stored in this fashion. This variety, which is known as a "bottom-up" DIB, is a remnant from the OS/2 days. Also, the pixels of a DIB don't always immediately follow the DIB's color table. The pixels can actually be stored in a separate block of memory. This is not the case with the header and the color table; the color table, if present, must always come right after the header.

The BITMAPINFO structure

As indicated in **Figure A**, a DIB's header and color table are stored together in a special structure called `BITMAPINFO`. Here's the declaration of this structure:

```
typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD bmiColors[1];
} BITMAPINFO, FAR *LPBITMAPINFO, *PBITMAPINFO;
```

The `BITMAPINFO` structure contains two data members: a `BITMAPINFOHEADER` structure, and a variable-length array of `RGBQUAD`s. As you've probably guessed, the first data member holds the DIB's header, and the array of `RGBQUAD`s holds the DIB's color table.

The header

The header contains vital information about a DIB, such as it's width, height, and color-depth.

Indeed, without this information, there would be no way to decode the DIB's pixels. There are actually three structures that can be used to store a DIB's header: `BITMAPINFOHEADER`, `BITMAPV4HEADER`, and `BITMAPV5HEADER`. Because the latter two are used only with ICM, in this article I'll focus strictly on the `BITMAPINFOHEADER` structure. Here's what it looks like:

```
typedef struct tagBITMAPINFOHEADER {
  DWORD biSize;
  LONG biWidth;
  LONG biHeight;
  WORD biPlanes;
  WORD biBitCount;
  DWORD biCompression;
  DWORD biSizeImage;
  LONG biXPelsPerMeter;
  LONG biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
} BITMAPINFOHEADER, FAR *LPBITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Don't be intimidated by the sheer number of data members in this structure. It turns out that most of these are usually set to zero.

The `biSize` data member specifies the size of the header, and is usually set to `sizeof(BITMAPINFOHEADER)`. Sometimes, however, additional information is stored immediately following the header, in which case, the `biSize` data member is increased accordingly. (In fact, this is the notion behind the `BITMAPV4HEADER`, and `BITMAPV5HEADER` structures—they store additional information at the end of the stock `BITMAPINFOHEADER`-type header.)

The `biWidth` and `biHeight` data members specify the width and height of the DIB (in pixels), respectively. The `biPlanes` data member specifies the number of color planes that the DIB uses; this data member must always be set to 1. The `biBitCount` data member specifies the number of bits that are used for each pixel: 1, 4, 8, 16, 24, or 32.

The `biCompression` data member specifies the type of compression that's used, if any. Usually, this data member is set to `BI_RGB` (0), indicating that the DIB is uncompressed. Alternatively, you can set `biCompression` to `BI_BITFIELDS`, indicating that the DIB is uncompressed and the first three entries of its data members contain `DWORD`-type values that are used as color masks. This identifier applies only to 16- or 32-bit DIBs. You can also `biCompression` to `BI_RLE4` or `BI_RLE8` to specify a run-length encoded 4-bit or 8-bit DIB, respectively. Or, you can specify this data member as `BI_JPEG` or `BI_PNG` if the DIB is compressed in the same fashion as a JPEG or

PNG image, respectively.
The `biSizeImage` data member specifies the total number of bytes that are required for the DIB's pixels. For uncompressed DIBs, this data member is usually set to zero.

The `biXPelsPerMeter` and `biYPelsPerMeter` data members specify the number of pixels per meter in the horizontal and vertical directions, respectively. These data members are usually set to zero.

The `biClrUsed` data member specifies the number of entries that the DIB's color table contains. Typically, this data member is set to zero, indicating that the color table contains the maximum number of colors for the DIB's color depth. For example, the color table of an 8-bit DIB can contain up to 256 entries. Likewise, the color table of a 4-bit DIB can contain up to 16 entries. As we'll discuss shortly, a color table is required for 1-, 4-, or 8-bit DIBs, but is optional for 16-, 24-, and 32-bit DIBs.

The `biClrImportant` data member specifies the number of colors in the DIB's color table that are "important". Usually, this data member is set to zero, indicating that all of the DIB's color table entries are important.

The color table

DIBs that represent each pixel using 1, 4, or 8 bits, must have a corresponding color table. For these types of DIBs, each pixel refers not to a literal RGB value, but instead to a color-table index value:

```
pixel_value = pixels[x][y]
color = color_table[index]
```

There are actually two color table varieties—those that are made up of an array of `RGBQUAD`s, and those that are composed an array of `WORD`s. In the latter case, each color table entry (i.e., `WORD`) refers to an index into a logical palette. Naturally, this type of color table is rare because it obviates the DIB's device independence. The `RGBQUAD` structure, on the other hand, conveys a literal RGB value. Here is that declaration:

```
typedef struct tagRGBQUAD {
  BYTE rgbBlue;
  BYTE rgbGreen;
  BYTE rgbRed;
  BYTE rgbReserved;
} RGBQUAD, FAR* LPRGBQUAD;
```

The `rgbBlue`, `rgbGreen`, and `rgbRed` data members specify the blue, green, and red intensities, respectively. (This odd ordering is another hangover from OS/2.) The `rgbReserved` data member is usually set to zero, but if you want to store extra information here (a translucency percentage, for example), you're free to do so.

As I pointed out earlier, not all DIBs have a color table. A color table is required for 1-, 4-, and 8-bit DIBs, but is optional for DIB of greater color-depths. You can tell whether a 16-, 24-, or 32-bit DIB uses a color table by examining the `biClrUsed` data member of the DIB's header. If this data member is zero, the DIB doesn't have a color table. This isn't case for 1-, 4-, and 8-bit DIBs; when the `biClrUsed` data member zero for these types, you should assume that the color table contains the maximum amount of entries allowed for the color-depth. You can compute this value like so (`bmi` is the DIB's header):

```
const short num_entries = 1 << bmi.biClrUsed;
```

The pixels

A DIB's pixels are used to define the actual image. How these pixels are stored depends on the number of bits that are used to represent each pixel (i.e., the DIB's color-depth). For example, a 1-bit DIB represents each pixel using only a single bit. A 4-bit DIB uses four bits to represent each pixel.

An 8-bit DIB represents each pixel using a full byte. This type of DIB is one of the easiest varieties to work with because you don't need to use bit-shifting or hexadecimal masks to decode each pixel. Just remember, each pixel conveys an index into the DIB's color table, not an actual color.

16-bit DIBs are often cumbersome to work with because each pixel value is packed into a `WORD`. To decode the RGB value from each `WORD` you need to use three hexadecimal-masks: `0xF800` for red, `0x07E0` for green, and `0x001F` for blue. These "bit field" masks tell you how many bits are allocated for each color channel (5 bits; the most significant bit is ignored) and how these bits are packed (blue, green, and then red). There is, however, a catch: if the `biCompression` data member of the DIB's header indicates `BI_BITFIELDS`, you need to grab the first three entries from the color table and use them as the masks.

The 24-bit variety is perhaps the easiest variety to work with because you can treat each pixel as an `RGBTRIPLE`. This structure is declared like this:

```
typedef struct tagRGBTRIPLE {
  BYTE rgbtBlue;
```

```
  BYTE rgbtGreen;
  BYTE rgbtRed;
} RGBTRIPLE;
```

In this case, because each pixel refers to an actual RGB value, there's no need for a color table. Also note that the bits for each color channel are packed as indicated in the `RGBTRIPLE` structure: eight for blue, then eight for green, and then eight for red (in that order).

For the 32-bit case, each pixel is often stored as an `RGBQUAD`, making this variety appealing as well. However, as with the 16-bit case, there's a catch. If the `biCompression` data member of the DIB's header indicates `BI_BITFIELDS`, you should use the first three entries of the DIB's color table to determine how many bits are allocated to each color channel.

DIB section bitmaps

A DIB section bitmap (often simply referred to as a "DIB section") is a hybrid between a DDB and a DIB. DIB sections contain both DDB and DIB components, as indicated by its corresponding structure, `DIBSECTION`. Here is it's declaration:

```
typedef struct tagDIBSECTION {
  BITMAP dsBm;
  BITMAPINFOHEADER dsBmih;
  DWORD dsBitfields[3];
  HANDLE dshSection;
  DWORD dsOffset;
} DIBSECTION, FAR *LPDIBSECTION, *PDIBSECTION;
```

The `dsBm` data member is a `BITMAP` structure that describes the DIB section's DDB component. The `dsBmih` data member is a `BITMAPINFOHEADER` structure that describes the DIB section's DIB component. For 16-bit and 32-bit DIB sections, there's no need to extract the `DWORD`-type masks from the color table—this information is conveniently presented via the `dsBitfields` data member. The `dshSection` data member specifies a handle to an optional memory-mapped file that can be used to store the DIB section's pixels. Accordingly, the `dsOffset` data member specifies the location of these pixels within the memory-mapped file.

From its DDB component, a DIB section bitmap inherits the properties of a true GDI graphic object—you can select a DIB section into a memory DC and then use your favorite GDI function to draw to it (or to draw the DIB section to another DC).

From its DIB component, a DIB section bitmap inherits a bit of device-independence. That is, the

color-depth of a DIB section bitmap is not limited to that of a specific device. Moreover, as with DIBs, DIB sections offer direct pixel access. This is the reason why the `TBitmap` class from C++Builder 3 and later offers the `ScanLine` property. Internally, `TBitmap` relies on DIB sections. In contrast, the `TBitmap` class from C++Builder 1 is based only on DDBs, and thus, there's no `ScanLine` property in this version.

Creating a DIB from a DDB or DIB section

As it turns out, the VCL (namely, the graphics.pas unit) simplifies the process of creating a DIB from a DDB or DIB section bitmap. Specifically, you use the `GetDIBSizes()` and `GetDIB()` functions. Here's what the former looks like:

```
void __fastcall GetDIBSizes(HBITMAP Bitmap, int
&InfoHeaderSize,int &ImageSize);
```

The `Bitmap` parameter, of type `HBITMAP`, specifies a handle to the DDB or DIB section bitmap whose required sizes (when converted to a DIB) you're interested in retrieving. This information is returned via the `InfoHeaderSize` and `ImageSize` parameters. The `InfoHeaderSize` parameter refers to an `int`-type variable that receives the size (in bytes) of the DIB's header and color table. The `ImageSize` parameter refers to another `int`-type variable that receives the size (in bytes) required to hold the DIB's pixels. So, you can use the `GetDIBSizes()` function to determine how much memory to allocate for the DIB.

Once you've called the `GetDIBSizes()` function and allocated a sufficiently large chunk of memory, you can then use the `GetDIB()` function to actually obtain the DIB. Here is the declaration for `GetDIB()`:

```
bool __fastcall GetDIB(HBITMAP Bitmap, HPALETTE Palette, void
*BitmapInfo, void *Bits);
```

As with that of the `GetDIBSizes()` function, the `Bitmap` parameter specifies a handle to the DDB or DIB section from which you're interested in creating a DIB. The `Palette` parameter specifies a handle to the DDB's or the DIB section's logical palette, if required. The `BitmapInfo` parameter is a pointer to a previously allocated block of memory that receives the DIB's header and color table. Likewise, the `Bits` parameter is a pointer to a previously allocated block of memory that receives a copy of the DDB's or the DIB section's pixels. Again, to determine how much memory to allocate for these data, you use the `GetDIBSizes()` function. Here's a simple example of how to use these functions:

```
// use the GetDIBSizes() function to gauge the size of the DIB's
parts

size_t header_ct_size = 0;
size_t image_size = 0;
GetDIBSizes(Bitmap->Handle, header_ct_size, image_size
  );

// compute the total size
const size_t total_size = header_ct_size + image_size;

if (total_size > 0)
{
  // allocate memory for the header
  unsigned char* pHeaderAndCT = new unsigned
char[header_ct_size];

  // allocate memory for the pixels
  unsigned char* pBits = new unsigned char[image_size];

  try
  {
    // use the GetDIB() function to get the header, color table,
    // and a copy of Bitmap’s pixels

    bool got_dib = GetDIB(Bitmap->Handle, Bitmap->Palette,
pHeaderAndCT, pBits);
    if (got_dib)
    {
      // grab a pointer to the BITMAPINFO structure

      LPBITMAPINFO lpbi =
reinterpret_cast<LPBITMAPINFO>(pHeaderAndCT);

      // grab a pointer to the color table

      LPRGBQUAD pColorTable = lpbi->bmiColors;

      // work with the DIB...
    }
  }
  catch (...)
```

```
  {
     // clean up
     delete [] pHeaderAndCT;
     delete [] pBits;
     throw;
  }
  // clean up
  delete [] pHeaderAndCT;
  delete [] pBits
}
```

Remember, a DIB is not a GDI graphic object, so the process of “creation” is actually a simple initialization. That is, the `GetDIB()` function calls the GDI `GetDIBits()` function, which in turn, instructs the device driver to convert the DDB from its internal format to the universal format (i.e., a DIB).

Conclusion

Well we didn’t do any printing yet, but we got the preliminaries out of the way. The take-home message is that the format of a DIB is well known, and so you can safely transfer a DIB from one device to another. From the previous code snippet, you now know how to create a DIB from a DDB or a DIB section bitmap. Next time, I’ll show you how to send this DIB to the printer.

TStringList as a template array class

By David Bridges

There are many container and array classes available for use with C++ Builder, such as the ones in the Standard Template Library (STL) that ships with all versions. Often, however, I like a “quick and easy” alternative to using these classes, which frequently have a high overhead of coding, and impose requirements on the type of objects that you put into an array.

C++ Builder’s Visual Component Library contains a few types of simple array classes, and among them is `TStringList`. `TStringList` is ideal for storing an array of strings. It has built-in methods for adding, deleting, sorting, finding, etc. One important feature that it also has is the ability to store an additional user-defined value along with each string. This value is accessed via the `Objects` property, which works just like the `Strings` property except that it returns objects of type `TObject*`.

The `Objects` property greatly extends the capabilities of `TStringList`. Because you can store pointers to any type of object, `TStringList` can be used as a container class for any type of object you can think of. However, there are a few drawbacks to using `TStringList` this way. First, in order to store and then retrieve your object pointers using the `Objects` property, you must always cast the pointers back and forth from between your type and `TObject*`. Second, there is no ownership defined for the objects in a `TStringList`. This means that when the `TStringList` object is deleted, any associated pointers contained in `Objects` are not deleted. You must manually delete all objects yourself in order to avoid a memory leak. For example:

```
class MyClass
{
  public:
  AnsiString sValue;
  MyClass(AnsiString sInValue) { sValue = sInValue; }
};

// Create the list
TStringList* mylist = new TStringList();

// Add objects to the list
mylist->AddObject("1", (TObject*) new MyClass("First"));
mylist->AddObject("2", (TObject*) new MyClass("Second"));

// Read objects from the list
for (int i=0;i< mylist->Count;i++) {
```

```
  MyClass* nextobj = (MyClass*)(mylist->Objects[i]);
  MessageBox(NULL, nextobj->sValue.c_str(), "Next Object:", MB_OK);
}

// Delete the objects
for (int i=0;i< mylist->Count;i++) {
  MyClass* nextobj = (MyClass*)(mylist->Objects[i]);
  delete nextobj;
}

// Free the list
delete mylist;
```

As you can see, the code isn't very pretty. We continually have to cast back and forth between `TObject` and the actual class we're using for the objects. Also, we must manually delete all of the objects when we're done with them.

But we do get some good benefits from using the `TStringList` class: we can sort the list based on the string values ("1" and "2", in this case), we can iterate through the list, and use all of the other methods and properties of `TStringList`.

The ObjectList class

The `ObjectList` class solves the casting problem and the deleting problem. One of my requirements in designing this class is that it imposes no restrictions upon the class of objects that it holds. The class doesn't have to be derived from `TObject`, doesn't have to have any operators defined, etc. In fact, it can be absolutely any type you want. Also, it should have a simple way to handle ownership of its objects. The listing for this class is shown in **Listing A**.

Before explaining any further, let's revisit the first example, but using the `ObjectList` class instead:

```
typedef ObjectList<MyClass> MyObjectList;

MyObjectList* myobjects = new MyObjectList(true);

// Add objects to the list
myobjects->AddItem("1", new MyClass("First"));
myobjects->AddItem("2", new MyClass("Second"));

// Read objects from the list
for (int i=0;i<myobjects->Count;i++) {
```

```
  MyClass* nextobj = myobjects->Items[i];
  MessageBox(NULL, nextobj->sValue.c_str(), "Next Object:", MB_OK);
  }

// Free the list, which will delete all objects.
delete myobjects;
```

The `AddItem()` method replaces the `AddObject()` method of `TStringList`. It takes a string and a pointer to an object of the class used in the template definition (in this case, `MyClass`). There is also a method called `AddNewItem()`. This works exactly the same as `AddItem()`, but it adds a new copy of the original object to the list, instead of the object itself. This sometimes saves writing code to allocate a new object each time. The only restriction is that the object should have a copy constructor.

The `Items` property replaces the `Objects` property of `TStringList`. It returns an object of the desired class, instead of a `TObject` pointer.

As you can see from the class constructor, the default behavior of the class is to sort the list and accept duplicates. It will sort based on whatever you have supplied as the string parameter in the `AddItem()` method. This gives a lot of flexibility since you can generate any kind of sort key you want for an object (or have the object create its own).

The `ObjectList` class has the ability to take ownership if the objects it contains. The class constructor takes one parameter, which is a flag indicating whether the list owns its objects. If it owns the objects, they will be automatically deleted when the list is deleted. If not, then you are responsible for deleting them. You can always override this ownership flag however, by using the `ClearItems()` method. If you pass `true` to `ClearItems()`, the list will be emptied and each of it's objects will be deleted. Passing `false` to `ClearItems()` will empty the list without deleting the objects.

One final thing to mention is that this provides a good example of creating properties in a class—in this case, the `Items` property. This is a very typical example of a read/write property and its associated read and write methods. This happens to be an indexed property so the read method looks like this:

```
T* GetItem(int index)
```

The write method is declared like this:

```
void SetItem(int index, T* ptr)
```

If it were not an indexed property, the read and write methods would be identical but lacking the index parameters. The `__property` keyword ties these together and specifies that these functions actually get called when you read or write the `Items` property.

Conclusion

By using the built-in power of the `TStringList` class and extending it by creating a template class that can handle any data type, we have created a very capable array class that can take advantage of all of the common array functions.

```
template <class T> class ObjectList :
  public TStringList
{
  public:

  // Set OwnsObjects to true if the objects should automatically be
  // deleted when the list is destroyed.
  bool OwnsObjects;

  __fastcall ObjectList(bool bOwnsObjects = false) : TStringList()
  {
    Duplicates = dupAccept;
    Sorted = true;
    OwnsObjects = bOwnsObjects;
  }

  __fastcall ~ObjectList()
  {
    ClearItems(OwnsObjects);
  }

  int AddItem(AnsiString sValue, T* ptr)
  {
    int index = Add(sValue);
    Objects[index] = (TObject*)ptr;
    return index;
  }

  int AddNewItem(AnsiString sValue, T* ptr)
  {
    T* newptr = new T;
    *newptr = *ptr;
```

```
    return AddItem(sValue, newptr);
  }

  T* GetItem(int index)
  {
    return (T*)Objects[index];
  }

  void SetItem(int index, T* ptr)
  {
    Objects[index] = (TObject*)ptr;
  }

  void ClearItems(bool Delete)
  {
    if (Delete) {
      for (int i=0;i<Count;i++) {
        T* ptr = GetItem(i);
        if (ptr) {
          delete ptr;
          SetItem(i, NULL);
        }
      }
    }
    Clear();
  }

  __property T*  Items[int Index] = {read=GetItem,write=SetItem};

};
```

Creating a flat combo box

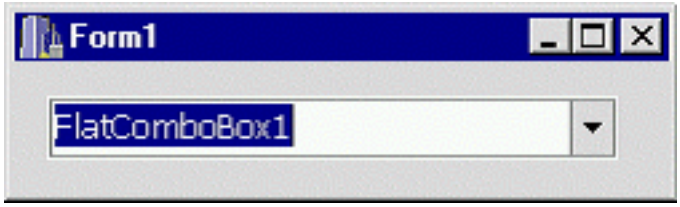
by Damon Chandler

You've seen them in Microsoft Word, you've seen them in Excel, and you've probably even seen them in some VCL applications. No, I don't mean illegal operations; I'm talking about flat combo boxes.

What is it with these new styles anyway? In Windows 3.1, most everything was flat—combo boxes, buttons, even scrollbars. In Windows 95, everyone was amazed by the new three-dimensional look of things. Presently, most controls seem to be a hybrid of these paradigms—flat at first, but three dimensional when touched by the mouse cursor. What's next, controls that morph into the shape of Bill Gates?

As it turns out, creating a flat combo box such as the one depicted in **Figure A** is relatively simple. In fact, the entire implementation boils down to one helper function and a slight modification of a standard combo box's window procedure. In this article, I'll guide you through the process of creating such a control.

Figure A: *A flat combo box*



The TFlatComboBox class

The key to making a flat combo box lies in redefining the way a standard combo box draws itself to the screen. The approach we'll take is this: if the mouse cursor is within the bounds of the combo box, we'll simply let the combo box draw itself normally. If the mouse cursor is beyond the bounds of the combo box, in which case the combo box should appear flat, we'll call additional code to draw the combo box in a flat fashion.

For this approach to succeed, we'll need notification of a few things. First, we'll need to know when the combo box draws itself to the screen. In addition, we'll need to know when the mouse cursor enters and leaves the bounds of the combo box. And, although it's probably not apparent at this point, we'll need to know when the combo box is clicked and when its drop-down list is closed.

When you read the word “notification”, you should immediately think of messages. As it turns out, combo boxes perform their entire drawing routine in response to the `WM_PAINT` message. Moreover, the VCL `CM_MOUSEENTER` and `CM_MOUSELEAVE` messages can be used to determine when the mouse cursor enters and leaves the bounds of the combo box, respectively. To get at these message, there are a few options, including performing an instance subclass on an existing `TComboBox` object, or by augmenting the virtual `Dispatch()` method (directly or via the message-mapping macros), or by augmenting the virtual `WndProc()` method. Since we’re going to handle multiple messages, let’s take the latter approach. To do this, we’ll need to create a new `TComboBox` descendant class, `TFlatComboBox`.

Defining a new window procedure

Let’s start with the declaration of the `TFlatComboBox` class, which is listed below. We’ll discuss each of its members along the way so don’t worry if they don’t’ make sense at this point. Also, note that the call to the `PACKAGE` macro is not needed in C++Builder version 1.

```
#include <StdCtrls.hpp>
#include <memory>

class PACKAGE TFlatComboBox :
  public TComboBox
{
private:
  bool pushed_;
  bool mouse_in_control_;

protected:
  virtual void __fastcall
    WndProc(TMessage& AMsg);
  virtual void __fastcall
    FlattenComboBox(TCanvas& CBCanvas);

public:
  __fastcall TFlatComboBox(
    TComponent* AOwner
    );
};
```

Now, our goal is to provide an implementation for the `WndProc()` method. This is our "tap", so to speak, into the combo box's message stream. Let's tackle the simplest part first, determining when the mouse cursor enters and leaves the bounds of the combo box.

Handling the CM_MOUSEENTER and CM_MOUSELEAVE messages

The `CM_MOUSEENTER` message is sent to a VCL control (i.e., a `TControl` descendant) when the mouse cursor enters the control's client area. Likewise, the `CM_MOUSELEAVE` message is sent to a VCL control when the mouse cursor goes exits the control's client area. We can handle these messages from within our definition of the `WndProc()` method to determine whether the combo box should be painted in a normal or a flat fashion. For example:

```
void __fastcall
  TFlatComboBox::WndProc(TMessage& AMsg)
{
  // only handle mouse- and action-
  // related messages at run-time
  if (!ComponentState.Contains(
    csDesigning))
  {
    switch (AMsg.Msg)
    {
      // when the mouse cursor
      // enters the combo box
      case CM_MOUSEENTER:
      {
        mouse_in_control_ = true;
        if (!DroppedDown) Invalidate();
        break;
      }
      // when the mouse cursor
      // leaves the combo box
      case CM_MOUSELEAVE:
      {
        mouse_in_control_ = false;
        if (!DroppedDown) Invalidate();
        break;
      }
```

```
    }
  }
  // pass the messages on to the
  // default window procedure
  TComboBox::WndProc(AMsg);
}
```

Notice that in response to the `CM_MOUSEENTER` and `CM_MOUSELEAVE` messages, we appropriately set a flag, `mouse_in_control_`, and then invoke a repaint via the `Invalidate()` method. In this way, when it's time to draw the combo box, we can test the `mouse_in_control_` member to determine whether the combo box should be drawn as flat or normal. Let's see how this is done.

Handling the WM_PAINT message

As I pointed out earlier, a combo box performs all of its drawing in response to the `WM_PAINT` message. This means that we don't have to worry about other commonly used painting-related messages such as `WM_ERASEBKGND` or `WM_NCPAINT`. In other words, inside our `WndProc()` method, we only need to handle the `WM_PAINT` message, which we can do like so:

```
void __fastcall
  TFlatComboBox::WndProc(TMessage& AMsg)
{
  // only handle mouse- and action-
  // related messages at run-time
  if (!ComponentState.Contains(
    csDesigning))
  {
    switch (AMsg.Msg)
    {
      // other code from before...
    }
  }
  // pass the messages on to the
  // default window procedure
  TComboBox::WndProc(AMsg);

  // after the combo box
  // has drawn itself and if
```

```
  // it should be made flat...
  if (AMsg.Msg == WM_PAINT &&
      !mouse_in_control_)
  {
    // render the flat combo box
    std::auto_ptr<TControlCanvas>
      CBCanvas(new TControlCanvas());
    CBCanvas->Control = this;
    FlattenComboBox(*CBCanvas);
  }
}
```

Notice that we handle the `WM_PAINT` message only after calling the `WndProc()` method of the parent class. In this way, we're effectively passing the `WM_PAINT` message to the default window procedure first, in response to which, the combo box will render itself as usual. After the combo box has rendered itself, we test the `mouse_in_control_` flag to see if we need to call additional drawing code to make the combo box appear flat. This task is accomplished by using a `TControlCanvas` object and via the application-defined `FlattenComboBox` method (which we'll implement shortly). If you've never used the `TControlCanvas` class before, consider the following lines of code:

```
std::auto_ptr<TControlCanvas>
  CBCanvas(new TControlCanvas());
CBCanvas->Control = this;
// ...
```

The lines in the preceding example are equivalent to the following `TCanvas` based approach:

```
std::auto_ptr<TCanvas>
  CBCanvas(new TCanvas());
CBCanvas->Handle = GetDC(Handle);
// ...
ReleaseDC(Handle, CBCanvas->Handle);
CBCanvas->Handle = NULL;
```

Finishing the WndProc() definition

Before we delve into coding the `FlattenComboBox` method, there are a few more

messages that we need to handle. Namely, when we later draw the combo box's scroll (drop-down) button, we'll need to know if this button should be rendered in a depressed (pushed) or normal fashion. To this end, we can use the `MouseDown` and `MouseUp` methods, but since we've already implemented much of the `WndProc()` method, we might as well handle the `WM_LBUTTONDOWN` and `WM_LBUTTONUP` messages directly. Remember, we'll want to repaint the scroll button in accordance with the status of the drop-down list. For notification of when the scroll button is in the non-depressed state, we can exploit the `CBN_CLOSEUP` notification message. Here's the code:

```
void __fastcall
  TFlatComboBox::WndProc(TMessage& AMsg)
{
  // only handle mouse- and action-
  // related messages at run-time
  if (!ComponentState.Contains(
    csDesigning))
  {
    switch (AMsg.Msg)
    {
      // when the left mouse
      // button is pressed
      case WM_LBUTTONDOWN:
      {
        pushed_ = true;
        break;
      }
      // when the left mouse
      // button is released
      case WM_LBUTTONUP:
      {
        pushed_ = false;
        if (!DroppedDown) Invalidate();
        break;
      }
      // on receipt of a reflected
      // WM_COMMAND message
      case CN_COMMAND:
      {
        // when the combo box's
```

```
        // drop-down list is closed
        if (AMsg.WParamHi == CBN_CLOSEUP)
        {
          pushed_ = false;
          Invalidate();
        }
        break;
      }
    }
  }
  // other code from before...
}
```

If you haven't already guessed, the `pushed_` member simply serves as a flag indicating if the scroll button is depressed. We'll test this flag from within the `FlattenComboBox()` method before drawing the scroll button. Let's now implement this method.

The FlattenComboBox() method

The `FlattenComboBox()` method is the meat and potatoes of the `TFlatComboBox` class. The role of this function is simple: alter the appearance of the combo box so that it appears as if it's flat. And, as it turns out, the implementation of this function is simple as well. Here's the code:

```
void __fastcall
  TFlatComboBox::FlattenComboBox(
    TCanvas& CBCanvas
    )
{
  RECT RBox =
    static_cast<RECT>(ClientRect);

  // use NULL_BRUSH so as not to
  // disturb edit control portion
  CBCanvas.Brush->Style = bsClear;

  // draw the new flat border
  CBCanvas.Pen->Width = 1;
  CBCanvas.Pen->Color = Color;
```

```
  CBCanvas.Rectangle(
    RBox.left, RBox.top,
    RBox.right, RBox.bottom
    );
  InflateRect(&RBox, -1, -1);
  CBCanvas.Pen->Width = 1;
  CBCanvas.Pen->Color = clBtnShadow;
  CBCanvas.Rectangle(
    RBox.left, RBox.top,
    RBox.right, RBox.bottom
    );

  // get the width of the scroll button
  int button_width =
     GetSystemMetrics(SM_CXVSCROLL);
  RECT RButton = {
    RBox.right - button_width - 1,
    RBox.top, RBox.right, RBox.bottom
    };

  // determine the style in
  // which to draw the button
  UINT style = DFCS_SCROLLCOMBOBOX;
  if (!Enabled) style |= DFCS_INACTIVE;
  if (pushed_) style |= DFCS_PUSHED;
  else style |= DFCS_FLAT;

  // draw the new scroll button
  DrawFrameControl(
    CBCanvas.Handle, &RButton,
    DFC_SCROLL, style
    );
}
```

Notice that we made two calls to the `TCanvas::Rectangle()` method to draw the new "flat" border. If you're not too worried about custom colors, you can replace these calls with a single call to the `DrawEdge()` API function. We delegate the hard part—drawing the scroll button—to the `DrawFrameControl()` API function. If you want to customize the appearance of this button, you can use the `Frame3D()` VCL function (declared in EXTCTRLS.HPP), or you could draw the scroll button manually using one or more methods from the `TCanvas` class.

Conclusion

Will a flat combo box enhance the functionality of your application? Well, no. What it might do, however, is enhance the aesthetic quality of your application's user interface. This aspect is something that can't be taken too lightly these days. Plus, the implementation is clean-cut, and you can even download the source code for the `TFlatComboBox` component from **www.bridgespublishing.com**. I encourage you to try a similar approach with other controls, especially those that the `DrawFrameControl()` function can draw for you (e.g., buttons, check boxes, and radio buttons).

Getting aliases and table names

by Kent Reisdorph

Every VCL database application has a global variable called `Session`. This variable is an instance of the `TSession` class. (Although there is a `TSession` component on the component palette, it is rarely necessarily to use this component in your applications.) Among other things, the `TSession` class can be used to get a list of alias names, tables within a particular database, and even to get a list of stored procedures.

To get a list of aliases that are currently registered with the BDE Administrator, call the `GetAliasNames()` method:

```
TStringList* aliases = new TStringList;
Session->GetAliasNames(aliases);
```

The single parameter passed to `GetAliasNames()` is an instance of `TStrings`. This means that you can easily add all alias names to any component that uses a `TStrings` instance (a list box, for example). This code does that:

```
Session->GetAliasNames(ListBox1->Items);
```

Once you have an alias you can easily get a list of tables contained in the database represented by that alias. This is done with the `GetTableNames()` method. For example:

```
TStringList* tables = new TStringList;
Session->GetTableNames(
  "BCDEMOS", "", true, false, tables);
```

The first parameter is the name of the BDE alias for which you want a list of tables. The second parameter is used to specify a search pattern. You can use wildcard characters to limit the list of tables returned. To retrieve all tables, simply pass an empty string for this parameter. The third parameter is used to specify whether the returned table names include the file extension. This parameter only applies to Paradox and dBase aliases. It is ignore for other database types. The fourth parameter is used to specify whether you want system tables returned as well as user tables. Usually you will pass `false` for this parameter, since you are not usually interested in system tables. The final parameter is used to specify the `TStrings` instance (a `TStringList`, for example) that will contain the table names when the call

to `GetTableNames()` returns.

The `GetStoredProcNames()` method works similar to `GetTableNames()`. As its name implies, it is used to get a list of stored procedures contained on a database server.

`TSession` has a number of other useful utility functions. See the VCL help for `TSession` for more information.

Getting file version information

by Bret Knigge

Under the Project Options dialog box in the C++Builder IDE, you will find the Version Info page. This is where you specify the file version information, including the product name, product version number, and file description to name just a few. This information is then displayed when a user right clicks on the application in Windows Explorer and chooses Properties from the context menu. This is all well and good, but C++Builder does not come with a component to actually retrieve this version information from within your program. This article will explain how to extract that information. The resulting information can then be displayed in an About dialog box.

There are three API functions that need to be called to obtain the version information for a file. The `GetFileVersionInfoSize()` and `GetFileVersionInfo()` function are used to extract version information about a specified file (application or DLL). The `VerQueryValue()` function returns specific data related to the version information. The first step is to get the size of the version information block within the file. That is accomplished with this code:

```
DWORD dwSize;
DWORD dwReserved;
AnsiString FileName =
Application->ExeName;
dwSize = GetFileVersionInfoSize(
FileName.c_str(), &dwReserved);
```

The second parameter in the function call is required by Windows and is used to set the value to zero. It has no meaningful purpose beyond that.

Now that you have the size of the version information block, you can get version information itself using `GetFileVersionInfo()`. Here's the code that does that:

```
LPVOID lpBuffer = new(dwSize);
GetFileVersionInfo(
FileName.c_str(), 0, dwSize, lpBuffer);
```

`lpBuffer` is an untyped block of memory (a `void*`) and will contain the version information when `GetFileVersionInfo()` returns. In this case I used operator `new` to allocate memory for `lpBuffer`. I could also have used the Windows API, function `HeapAlloc()`, to allocate the memory but that is a bit old fashioned in C++.

`lpBuffer` now holds the file's version information. However you still need to use `VerQueryValue()` to extract the individual data related to that information. The following code shows how this is done:

```
LPVOID lpStr;
UINT wLength;
VerQueryValue(lpBuffer,
"\\StringFileInfo\\040904E4\\”
“ProductName", &lpStr, &dwLength);
AnsiString Info =
reinterpret_cast<char *>(lpStr);
```

Using the `VerQueryValue()` function is fairly straightforward, with the result being written to the third argument (`lpStr`), which in then cast to a more meaningful type (an `AnsiString`).

When you've finished accessing the version information, you must free the memory allocated for `lpBuffer`. If the memory was allocated using the Windows API function `HeapAlloc()`, then you must call `HeapFree()` to free the memory. If, on the other hand you used operator `new` to allocate the memory, you would need to use operator `delete` to free the memory. Here's how that code looks:

```
if (lpBuffer)
delete(lpBuffer);
```

Adding version information to an About dialog box in an application can aid in giving your program a professional touch. For those that are interested, the second argument in the `VerQueryValue()` call can be obtained by examining the resource file of the application using a resource editor such as that found in Borland C++ 5.02.

Using Visual C++ DLLs with C++Builder

by Harold Howe

It is possible that one day your boss will ask you if you can create an application with C++Builder that interfaces to an existing DLL compiled with Microsoft Visual C++. Often, the original DLL source code won't be available to you; either because the DLL comes from a third party vendor, or because your new intern just deleted the \DLL\SRC directory from the network. This article shows you how to call a DLL built with Visual C++ from your C++Builder project given only a DLL and a header file.

Calling DLL functions from a C++Builder project

Calling a DLL that was created with Visual C++ presents some unique challenges. Before I explain how to use DLLs generated by Visual C++, it may be beneficial to review how you call functions in a DLL that was created with C++Builder. A DLL that was created with C++Builder presents fewer roadblocks than one that was built with Visual C++.

You need to gather three ingredients in order to call a DLL function from your C++Builder program: the DLL itself, a header file with function prototypes, and an import library. (You could load the library at run time instead of using an import library, but we will stick to the import library method for simplicity.) To call a function in a DLL, you first add the import library to your C++Builder project by selecting the Project | Add To Project from the main menu. The import library is created for you automatically when you build a DLL project with C++Builder. Next, insert a `#include` statement for the DLL header file in the C++ source file that needs to call one of the DLL functions. Finally, add the code that calls the DLL function.

Listings A and **B** contain source code for a simple DLL that that you can use for testing. Notice that the test code implements two different calling conventions: standard call (represented by the `__stdcall` keyword) and the C calling convention (`__cdecl`). I did this for a very good reason: when you try to call a DLL that was compiled with Visual C++, most of your headaches will result from problems that arise due to mismatched calling conventions. Also notice that one function does not explicitly list the calling convention that it uses. This unknown function will act as a measuring stick for DLL functions that don't specifically declare a calling convention.

To build the test DLL, first create a new DLL project from the Object Repository. (Alternatively, you can download the project from the Bridges Publishing Web site.) C++Builder now shows the source for the DLL. At this point, the source contains just a DLL entry point function and some include statements. Create a new unit and save it as DLL.CPP. Enter the code from **Listing A** and insert it into the DLL.H header. Next switch to DLL.CPP file and enter the code from **Listing B**. Make sure that the `#define` for `_BUILD_DLL_` is placed above the include statement for DLL.H.

Save the project as BCBDLL.BPR. Next, compile the project and take a look at the files produced. C++Builder generates both a DLL and an import library with a .LIB extension.

At this point, you have the three ingredients needed to call functions in the DLL from a C++Builder project. Next, you need to create a C++Builder project that will try to call the DLL functions. Create a new project in C++Builder and save it to your hard drive. Copy the DLL, the import library, and the DLL.H header file from the DLL project's folder to the folder containing the test EXE project. Select Project | Add To Project from the C++Builder main menu and add BDCBDLL.LIB to the project. Next, add a `#include` statement in the main unit that includes DLL.H. Finally, add code that calls the DLL functions. **Listing C** shows code that calls each of the DLL's functions. You should find that all is well and that the test application can call functions in the DLL without problems.

The problem with Visual C++ DLLs

In an ideal world, calling a DLL created with Visual C++ would be no more difficult than calling a DLL built with C++Builder. Unfortunately, Borland and Microsoft disagree on three primary technical points when it comes to DLLs. First, Borland and Microsoft use different object file formats (this affects objs and import library files). Visual C++ uses the COFF library format while Borland uses OMF. This means that you can't add a Microsoft generated import library to a C++Builder project. Thanks to the Borland IMPLIB utility, the file format differences are easily remedied.

Second, the two products disagree on linker naming conventions. This turns out to be the primary hurdle when trying to call a Visual C++ DLL from C++Builder. Every function in a DLL or OBJ has a linker name. The linker uses the linker name to resolve functions that were prototyped at compile time. The linker will generate an unresolved external error if it can't find a function with a linker name that it thinks is needed by the program.

With regard to linker function names, Borland and Microsoft disagree on these points:

- Visual C++ sometimes decorates exported `__stdcall` functions
- Borland C++Builder expects imported `__cdecl` functions to be decorated

So why is this such a big deal? Take the first point above. Let's say you have a DLL created with Visual C++ and that the DLL contains a `__stdcall` function called `MyFunction()`. Visual C++ will give the function a linker name of `_MyFunction@4`. When the Borland linker tries to resolve calls made to this function, however, it expects to find a function with the name `MyFunction`. Since the import library for the Visual C++ DLL doesn't contain a function called `MyFunction`, the Borland linker generates an "unresolved external" linker error. Basically, the linker couldn't find the function in the import library so it generates the error.

Your strategy for overcoming these problems will depend on how the Visual C++ DLL was compiled. I have broken the process into four steps.

Step 1: Identify the calling conventions

In order to combat the naming convention entanglements, you must first determine what calling conventions are used by functions in the DLL. You can do this by investigating the header file for the DLL. The function prototypes in the DLL header should look something like this:

```
__declspec(dllimport) void <convention>
  MyFunction(int nArg);
```

The placeholder `convention` will usually be `__stdcall` or `__cdecl` (see **Listing A** for concrete examples). In many cases, the calling convention won't be specified, in which case it defaults to `__cdecl`.

Step 2: Examine the linker names in the DLL

If Step 1 reveals that the DLL utilizes the `__stdcall` calling convention, you will need to examine the DLL to determine the naming convention used when the DLL was built. Visual C++ decorates `__stdcall` functions by default, but the DLL programmer can prohibit name decorations if he or she adds a DEF file to the project. Your work will be slightly more tedious if the DLL supplier did not use a DEF file.

The TDUMP command line utility allows you to examine the linker names of functions exported by the DLL. The following command invokes TDUMP on a DLL.

```
TDUMP -ee MYDLL.DLL > MYDLL.TXT
```

TDUMP can report a ton of information about the DLL. We're only interested in functions exported by the DLL, so the `-ee` switch is used to instruct TDUMP to list only export information. If the DLL is large, you may want to redirect the output of TDUMP to a text file as shown in the previous example.

The TDUMP output for the test DLL in **Listings A** and **B** looks like this:

```
Turbo Dump  Version 5.0.16.4 Copyright
(c) 1988, 1998 Borland International

Display of File DLL.DLL

EXPORT ord:0000='CdeclFunction'
EXPORT ord:0002='UnknownFunction'
EXPORT ord:0001='_StdCallFunction@4'
```

Notice the leading underscore and the trailing `@4` on the `__stdcall` function. The `__cdecl` and the unknown function don't contain any decorations. If the Visual C++ DLL had been compiled with a DEF file, the decorations on the `__stdcall` function would not be present.

Step 3: Generate an import library for the Visual C++ DLL

Now we get to the hard part. Due to the library file format differences between C++Builder and Visual C++, you cannot add an import library created with Visual C++ to your C++Builder project. You must create an OMF import library using the command line tools that come with C++Builder. Depending out what you found in the first two steps, this step will either go smoothly, or it could take some time.

As stated earlier, C++Builder and Visual C++ don't agree on how functions should be named in a DLL. Due to naming convention differences, you will need to create an aliased import library if the DLL implements calling conventions in which C++Builder and Visual C++ disagree. **Table A** lists the areas of disagreement.

Table A: *Visual C++ and C++Builder naming conventions*

Calling convention	VC++ name	VC++ (DEF used)	C++Builder Name
__stdcall	_MyFunction@4	MyFunction	MyFunction
__cdecl	MyFunction	MyFunction	_MyFunction

The C++Builder column lists function names that the Borland linker expects to see. The first Visual C++ column lists the linker names that Visual C++ generates when the Visual C++ project does not utilize a DEF file. The second Visual C++ column contains linker names that Visual C++ creates when a DEF file is used. For things to go smoothly, the C++Builder name should agree with the Visual C++ name. Notice that the two products agree in only one place: `__stdcall` functions where the Visual C++ project contained a DEF file. For the remaining scenarios, you will need to create an import library that aliases the Visual C++ name to a C++Builder compatible name.

Table A shows that there are several combinations that you may need to deal with when creating the import library. I have separated the combinations into two cases.

Case 1: The DLL contains only __stdcall functions and the DLL vendor utilized a DEF file.

Table A reveals that Visual C++ and C++Builder agree only when the DLL uses `__stdcall` functions. Furthermore, the DLL must be compiled with a DEF file to prevent Visual C++ from decorating the linker names. The header file will tell you if the `__stdcall` calling convention was used (Step 1), and TDUMP will reveal whether or not the functions are decorated (Step 2). If the DLL contains `__stdcall` functions that are not decorated, then Visual C++ and C++Builder agree on how the functions should be named. If that is the case, you can create an import library by simply running IMPLIB on the DLL. No aliases are needed. IMPLIB works like this:

```
IMPLIB (output lib name) (source dll)
```

For example:

```
IMPLIB mydll.lib mydll.dll
```

Once you have created the import library you can move on to step 4.

Case 2: The DLL contains __cdecl functions or decorated __stdcall functions.

If your DLL vendor is adamant about creating DLLs that are compiler independent, then you have a good chance of falling into the Case 1 category. Unfortunately, odds are you won't fall into the Case 1 group for several reasons. For one, the calling convention defaults to `__cdecl` if a calling convention was not specified when the DLL was built. Secondly, even if your vendor has utilized the `__stdcall` calling convention, they probably neglected to utilize a DEF file to strip the Visual C++ name decorations.

However you got here, welcome to Case 2. You're stuck with a DLL whose function names don't agree with C++Builder. Your only way out of this mess is to create an import library that aliases the Visual C++ function names into a format compatible with C++Builder. Fortunately, the C++Builder command line tools allow you to create an aliased import library.

The first step is to create a DEF file from the Visual C++ DLL by using the IMPDEF utility that comes with C++Builder. IMPDEF creates a DEF file that lists all of the functions exported by the DLL. You invoke IMPDEF like this:

```
IMPDEF (output DEF file) (source DLL)
```

For example:

```
IMPDEF mydll.def mydll.dll
```

After running IMPDEF, open the resulting DEF file using the editor of your choice. When the DLL shown in **Listings A** and **B** is compiled with Visual C++, the DEF file created by IMPDEF looks like this:

```
LIBRARY     DLL.DLL

EXPORTS
 CdeclFunction @1
 UnknownFunction @3
 _StdCallFunction@4 =_StdCallFunction @2
```

The next step is to alter the DEF file so it aliases the DLL functions into names that C++Builder will like. You can alias a function by listing a C++Builder compatible name followed by the original Visual C++ linker name. For the test DLL the aliased DEF looks like this:

```
EXPORTS
```

```
 ; use this type of aliasing
 ; (Borland name) =
    (Name exported by Visual C++)
 _CdeclFunction   = CdeclFunction
 _UnknownFunction = UnknownFunction
 StdCallFunction  = _StdCallFunction@4
```

Notice that the function names on the left match the Borland compatible names from **Table A**. The function names on the right are the actual linker names of the functions in the Visual C++ DLL.

The final step is to create an aliased import library from the aliased DEF file. Once again, you rely on the IMPLIB utility, except that this time, you pass IMPLIB the aliased DEF file as its source file instead of the original DLL. The format is:

```
IMPLIB (dest lib file) (source def file)
```

For example:

```
IMPLIB mydll.lib mydll.def
```

Before going on to Step 4 you may want to examine the import library first to ensure that each DLL function appears in a naming format that C++Builder agrees with. You can use the TLIB utility to inspect the import library.

```
TLIB mydll.lib, mydll.lst
```

The list file for the test DLL looks like this:

```
Publics by module

StdCallFunction size = 0
        StdCallFunction

_CdeclFunction  size = 0
        _CdeclFunction

_UnknownFunction size = 0
        _UnknownFunction
```

Step 4: Add the import library to your project

Once you have created an import library for the Visual C++ DLL, you can add the import library to your C++Builder project. You use the import library without regard to whether the import library contains aliases or not. After adding the import library to your project, build the project and see if you can successfully link.

Conclusion

You may have noticed that this article only discusses how to call C style functions in a DLL. No attempt is made to call methods of an object where the code for the class resides in a Visual C++ DLL. DLLs containing C++ classes present an even greater array of problems because linker names for member functions are mangled. The compiler employs a name-mangling scheme in order to support function overloading. Unfortunately, the C++ standard does not specify how a compiler should mangle class method names. Without a strict standard in place, Borland and Microsoft have each developed their own techniques for name mangling, and the two conventions are not compatible. In theory, you could use the same aliasing technique to call member functions of a class that resides in a DLL. However, you may want to consider creating a COM object instead. COM introduces many of its own problems, but it does enforce a standard way of calling methods of an object. A COM object created by Visual C++ can be called from any development environment, including both Delphi and C++Builder.

C++Builder 3.0 introduced a new command line utility called COFFToOMF.EXE. This utility can convert a Visual C++ import library to a C++Builder import library. Furthermore, the program automatically aliases `__cdecl` functions from the Visual C++ format to the C++Builder format. The automatic aliasing can simplify Step 3 if the DLL uses the `__cdecl` calling convention exclusively.

Listing A: *DLL.H*

```
#ifdef __cplusplus
extern "C" {
#endif

#ifdef _BUILD_DLL_
#define FUNCTION __declspec(dllexport)
#else
#define FUNCTION __declspec(dllimport)
#endif
```

```
FUNCTION int __stdcall StdCallFunction(int Value);
FUNCTION int __cdecl CdeclFunction  (int Value);
FUNCTION int UnknownFunction(int Value);

#ifdef __cplusplus
}
#endif
```

Listing B: *DLL.C*

```
#define _BUILD_DLL_
#include "dll.h"

FUNCTION int __stdcall StdCallFunction(int Value)
{
  return Value + 1;
}

FUNCTION int __cdecl   CdeclFunction(int Value)
{
  return Value + 2;
}

FUNCTION int UnknownFunction(int Value)
{
  return Value;
}
```

Listing C: *MAINFORM.CPP - DLLTest program*

```
#include <vcl\vcl.h>
#pragma hdrstop

#include "MAINFORM.h"
#include "dll.h"

#pragma resource "*.dfm"
TForm1 *Form1;
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
  TForm1::Button1Click(TObject *Sender)
{
  int Value = StrToInt(Edit1->Text);
  int Result= StdCallFunction(Value);
  ResultLabel->Caption = IntToStr(Result);
}

void __fastcall
  TForm1::Button2Click(TObject *Sender)
{
  int Value = StrToInt(Edit1->Text);
  int Result= CdeclFunction(Value);
  ResultLabel->Caption = IntToStr(Result);
}

void __fastcall
  TForm1::Button3Click(TObject *Sender)
{
  int Value = StrToInt(Edit1->Text);
  int Result= UnknownFunction(Value);
  ResultLabel->Caption = IntToStr(Result);
}
```

Adding a background image to a list view

by Damon Chandler

By and large, I'm impressed with the Windows API. It offers a set of well documented, albeit sometimes intimidating, functions that can be used to performed most everything that we see in Windows itself. What I don't like is the fact that we're often presented with a limited set of features. The list view control is a prime example. Even back in Windows 95 we were able to specify a background image for the desktop (which is also a list view). Only recently has this feature been added to standard list views.

There are actually a few ways to add a background image to a list view. The technique I'll discuss in this article involves the `LVM_SETBKIMAGE` message. Note that this message requires that at least version 4.71 (IE 4.0+) of the common control `DLL` be installed on the target. If this is not the case, then next best approach is to use the Custom Draw service, which requires Comctl32.dll version 4.70+ (IE 3.0+) of the. A third approach is to render the image manually in response to the `WM_ERASEBKGND` message.

Setting the background image

Use of the `LVM_SETBKIMAGE` message is the ideal approach to display a background image in a list view because it offers support for a variety of image formats. However, because this support is tied to COM (the Component Object Model), you must first initialize the COM `DLL`s. Let's see how this is done.

Initializing the COM libraries

To initialize the COM `DLL`s, you use the `CoInitialize()` function. For 32-bit applications, this function's single parameter must be `NULL`. You can call this function from within your form's constructor, like so:

```
__fastcall
  TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  if (FAILED(CoInitialize(NULL)))
  {
```

```
    throw EWin32Error(
      "Failed to initialize "
      "the COM libraries!"
      );
  }
}
```

The `CoInitialize()` function returns a value of type `HRESULT`, which the `FAILED()` macro tests for negativity. Later, I'll show you how to un-initialize the COM libraries, but for now, let's look at the `LVM_SETBKIMAGE` message.

The LVM_SETBKIMAGE message

As its name suggests, the `LVM_SETBKIMAGE` message is used to set a list view's background image. This image does not have to be a bitmap—it can alternatively be an icon, a metafile, an enhanced metafile, a JPEG, or even a GIF. You indicate the particulars of this image via the `lParam` parameter, which is specified as the address of a `LVBKIMAGE` structure. The `wParam` parameter is ignored.

The `LVBKIMAGE` structure is designed to hold information about a list view's background image. It's defined, with the rest of the common-controls-related structures, in commctrl.h:

```
typedef struct tagLVBKIMAGE
{
  ULONG ulFlags;
  HBITMAP hbm;
  LPSTR pszImage;
  UINT cchImageMax;
  int xOffsetPercent;
  int yOffsetPercent;
} LVBKIMAGE, FAR *LPLVBKIMAGE;
```

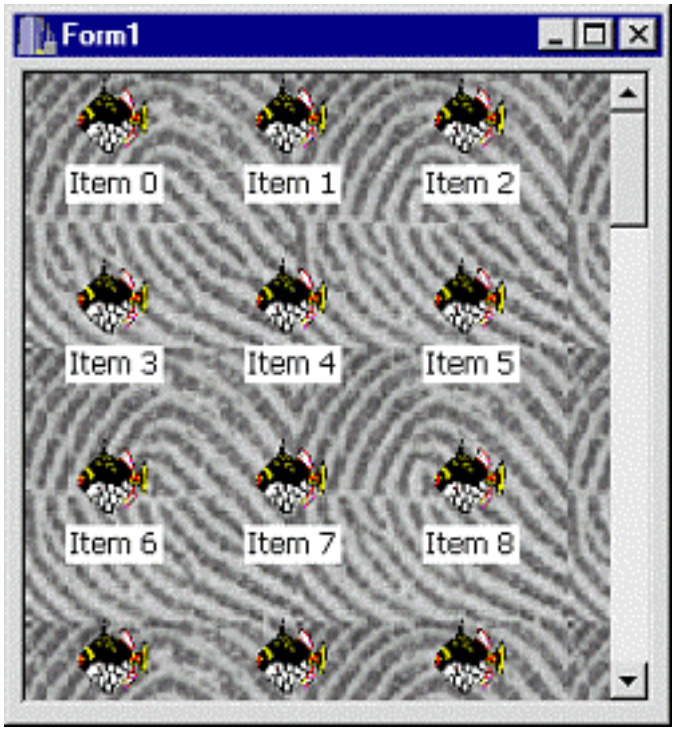
The `ulFlags` data member is used to specify whether you're adding (`LVBKIF_SOURCE_URL`) or removing (`LVBKIF_SOURCE_NONE`) a background image, and whether this image is to be displayed in a normal (`LVBKIF_STYLE_NORMAL`) or a tiled (`LVBKIF_STYLE_TILE`) fashion. The `hbm` data member is currently ignored, but guessing from its name, eventually this data member will allow the specification of a handle to a bitmap object. Until that time, you have to use the `pszImage` data member, which you can assign the absolute path to an existing image file (use the `TGraphic::SaveToFile()` method if you need to create a file from an existing object),

or a URL to an image file. The `cchImageMax` data member is ignored unless the `LVBKIMAGE` structure is used with the `LVM_GETBKIMAGE` message (which is designed to retrieve information about a list view's background image). Finally, the `xOffsetPercent` and `yOffsetPercent` data members are used to specify the position of the background image—in percentage of the list view's client area—when the image is not tiled.

Once you've initialized an `LVBKIMAGE` structure, you're ready to send the list view the `LVM_SETBKIMAGE` message. You can do this by means of the `SendMessage()` function or the `TControl::Perform()` method. For example, to display a background JPEG image in a tiled fashion, as depicted in **Figure A**, you'd use the `SendMessage()` function, like so:

```
LVBKIMAGE lvbki = {0};
lvbki.ulFlags =
  LVBKIF_SOURCE_URL |
  LVBKIF_STYLE_TILE;
lvbki.pszImage = TEXT("C:\\FBI.JPG\0");

const bool success = SendMessage(
  ListView1->Handle, LVM_SETBKIMAGE, 0,
  reinterpret_cast<LPARAM>(&lvbki)
  );
if (!success)
{
  throw EWin32Error(
    "LVM_SETBKIMAGE failed!"
    );
}
```

Figure A: *A list view with a tiled background image*



If you use the `TControl::Perform()` method, the technique is similar, except you need to ensure that the underlying list view has indeed been created. You can do this by using the `TWinControl::HandleNeeded()` method. For example, to display a background image that's centered in the list view's client area, where the image itself is located remotely, you'd use the `Perform()` method, like so:

```
LVBKIMAGE lvbki = {0};
lvbki.ulFlags =
  LVBKIF_SOURCE_URL |
  LVBKIF_STYLE_NORMAL;
lvbki.pszImage = TEXT(
  "http://www.pinkfloyd.com/"
  "pics/pulse.jpg\0"
  );
lvbki.xOffsetPercent = 50; // 50%
lvbki.yOffsetPercent = 50; // 50%

ListView1->HandleNeeded();
const bool success =
  ListView1->Perform(
    LVM_SETBKIMAGE, 0,
    reinterpret_cast<LPARAM>(&lvbki)
    );
```

```
if (!success)
{
  throw EWin32Error(
    "LVM_SETBKIMAGE failed!"
    );
}
ListView1->Perform(
  LVM_SETTEXTBKCOLOR, 0, CLR_NONE
  );
```

Note the use the `LVM_SETTEXTBKCOLOR` message in this code snippet. Here, the message is sent with the `LParam` argument set to `CLR_NONE` (`WParam` is ignored) so that the text of each item is drawn with a transparent background, as depicted in **Figure B**. In general, you can use this message to specify the background color of the list items.

Figure B: *A list view with a centered background image*

Closing the COM libraries

Every successful call to the `CoInitialize()` function should have a corresponding call to the `CoUninitialize()` function, which simply serves to close the COM libraries. You can call this function from within your form's destructor, but you must

ensure that the list view's background image is removed beforehand. You can do this by using the `LVM_SETBKIMAGE` again, this time using the `LVBKIF_SOURCE_NONE` identifier, like so:

```
__fastcall TForm1::~TForm1()
{
  LVBKIMAGE lvbki =
    {LVBKIF_SOURCE_NONE};
  ListView1->Perform(
    LVM_SETBKIMAGE, NULL,
    reinterpret_cast<LPARAM>(&lvbki)
    );
  CoUninitialize();
}
```

An alternative approach is to simply destroy the list view:

```
__fastcall TForm1::~TForm1()
{
  delete ListView1;
  CoUninitialize();
}
```

Fixing the TListView class

As it turns out, the `LVM_SETBKIMAGE` message won't work with a `TListView` object. The message will work when sent to a standard list view control, and it will even indicate success when sent to a `TListView` object. The problem is, in the latter case, no background image is displayed. To understand why, we need to examine how the background image is rendered and what the `TListView` class does to prevent this image from appearing

How the background image is displayed

When you send the `LVM_SETBKIMAGE` message to a list view, you're instructing the control to display an image in its background. An ideal time to draw this image is in response to the `WM_ERASEBKGND` message, and this is exactly what the list view does. The support for all the various file formats is provided by the `IPicture` COM interface, which the list view uses—this explains the need for the initialization of the COM libraries. Things become more challenging, however, when we mix in the VCL.

What the TCustomListView class does wrong

There are actually two problems working against us here, one of which is caused by the `TCustomListView` class and the other by the `TWinControl` class. For any window, there are two ways to adjust the color of its background. The most common approach is to specify the handle to a colored brush object to the `WNDCLASS::hbrBackground` data member; in which case the window's default window procedure will use this brush to paint the background in response to the `WM_ERASEBKGND` message. The second approach is to handle the `WM_ERASEBKGND` message directly, in response to which the background is usually painted manually via the `FillRect()` function. The `TWinControl` class takes this second approach, using the `FillRect()` function, along with the `TWinControl::Brush` property, to paint the control's background. In this way, you can change the background color by simply changing the `Color` property of the control's `TBrush` object. Moreover, since the `TWinControl` class paints the background manually, it traps the `WM_ERASEBKGND` message since there's no need for the default window procedure to receive it. Indeed, this makes sense since the background has already been painted.

Well, there's a problem here. If the underlying list view paints its background image in response to the `WM_ERASEBKGND` message but the `TWinControl` class traps this message, then it's not surprising that a `TListView` object doesn't display the background image. But, this isn't the actual problem. As it turns out, the `TCustomListView` class traps the `WM_ERASEBKGND` message before the `TWinControl` class ever sees it. The logic here follows from the fact that standard list views provide a custom means of setting the background color; namely, via the `LVM_SETBKCOLOR` message. For this reason, the `TCustomListView` class traps the `WM_ERASEBKGND` message and redefines the `TControl::SetColor` method to use the `LVM_SETBKCOLOR` message. Again, since the `WM_ERASEBKGND` message is trapped, no background image is displayed.

The solution

If you haven't already guessed, the solution is to intercept the `WM_ERASEBKGND` message before the `TCustomListView` class gets a chance to trap it. In this way, you can manually pass the message on to the default window procedure, in response to which, the underlying list view will indeed render the background image. For example:

```
class TMyListView : public TListView
{
protected:
```

```
  virtual void __fastcall
    WndProc(TMessage& AMsg)
    {
      if (AMsg.Msg == WM_ERASEBKGND)
      {
        DefaultHandler(&AMsg);
      }
      else TListView::WndProc(AMsg);
    }
public:
  __fastcall TMyListView(
    TComponent* AOwner
    ) : TListView(AOwner) {}
};
```

Conclusion

Of all the common controls, list views are, without a doubt, the most versatile variety. In fact, because list views are so ubiquitous in Windows itself, we'll likely see several more advances in the near future. In this article, I've demonstrated the use of the `LVM_SETBKIMAGE` message, which, with a bit of tweaking, works quite well with the `TListView` class. The source for the `TFixedListView` class, a `TListView` descendant class with the aforementioned `WM_ERASEBKGND`-related modifications already in place, is available for download at **www.bridgespublishing.com**.

Advanced string grid techniques

by Damon Chandler

Last month, I focused mainly on visual aspects of the `TStringGrid` class, and the month before, on functionality. This month, I'll discuss some techniques for enhancing a string grid's accessibility—I'll show you how to copy a group of cells to and from the clipboard, and how to support drag and drop operations from other applications.

Clipboard transfers

If you've never transferred data to or from the clipboard, don't worry—the `TClipboard` class significantly simplifies things. In fact, since we're only transferring text, the process is entirely straightforward, even without help from the VCL. As it turns out, the most cumbersome part of copying a group of cells to the clipboard is actually copying the text of each cell into a buffer, and then formatting the buffer into a form that other grid-based applications, such as Microsoft Excel, can accept.

When you select a group of cells in Excel, then copy the selection to the clipboard, the clipboard will contain a text buffer, where a tab character delimits each column, and the end of each row is denoted by a carriage return/linefeed combination. For example, if you copy cells A1:C3 (where the text of each cell denotes its location), the clipboard will contain the following (minus the white space):

```
A1 \t A2 \t A3 \r\n
B1 \t B2 \t B3 \r\n
C1 \t C2 \t C3 \r\n
```

When you paste text from the clipboard to Excel, the text buffer must have this format to align properly into each cell. This means that before your application copies text to the clipboard, you must ensure that the text is arranged in the above format. Likewise, when you copy text from Excel to the clipboard, and then paste this text into your application, you'll receive a buffer that's formatted as above, which you'll need to parse into each cell. Let's tackle the former case first.

Copying cells to the clipboard

Before you can copy cells to the clipboard, you need to extract the text from each cell, and then arrange the collected text in the aforementioned format. The following application-defined function, `CellToText()`, demonstrates this process:

```
AnsiString __fastcall CellsToText(
  TStringGrid& AGrid,
```

```
  const TGridRect& ACells)
{
  //
  // compute the total size of the
  // buffer required to hold the text
  // of each cell plus the tab and
  // CR/LF delimiters
  //
  int text_len = 0;
  for (int row = ACells.Top;
       row <= ACells.Bottom;
       ++row)
  {
    for (int col = ACells.Left;
         col <= ACells.Right;
         ++col)
    {
      text_len +=
        AGrid.Cells[col][row].Length();
      if (col < ACells.Right)
        ++text_len;
    }
    if (row < ACells.Bottom)
      text_len += 2;
  }

  //
  // fill the AnsiString with the
  // text of each cell in a tab-
  // and CR/LF-delimited format
  //
  AnsiString text;
  text.SetLength(text_len);
  text = "";
  for (int row = ACells.Top;
       row <= ACells.Bottom;
       ++row)
  {
    for (int col = ACells.Left;
         col <= ACells.Right;
```

```
        ++col)
    {
      text += AGrid.Cells[col][row];
      if (col < ACells.Right)
        text += '\t';
    }
    if (row < ACells.Bottom)
      text += "\r\n";
  }
  return text;
}
```

To actually copy the text to the clipboard, you simply use the `AsText` property of `TClipboard` like so:

```
void __fastcall CopyCellsToClipboard(
  TStringGrid& AGrid,
  const TGridRect& ACells)
{
  // grab the formatted text
  AnsiString text(
    CellsToText(AGrid, ACells)
    );
  // copy the text to the clipboard
  Clipboard()->AsText = text;
}
```

The `ACells` parameter of both of these functions simply defines which cells to copy. Most likely, you'll want to copy the selected cells, in which case, the string grid's `Selection` property can be used as in the following code:

```
// OnKeyDown event handler
void __fastcall TForm1::
  StringGrid1KeyDown(TObject *Sender,
  WORD &Key, TShiftState Shift)
{
  if (Key == 'C' &&
      Shift.Contains(ssCtrl))
  {
    CopyCellsToClipboard(
```

```
      *StringGrid1,
      StringGrid1->Selection
      );
  }
}
```

Pasting cells from the clipboard

To retrieve text from the clipboard, you read the `AsText` property of `TClipboard` instead of writing to it. For example:

```
void __fastcall PasteCellsFromClipboard(
  TStringGrid& AGrid,
  const TGridCoord& AFirstCell)
{
  // extract the text
  AnsiString text(Clipboard()->AsText);
  if (!text.IsEmpty())
  {
    // prevent grid updates
    AGrid.Rows[0]->BeginUpdate();
    try
    {
      // fill the cells with the text
      TextToCells(
        AGrid, AFirstCell, text
        );
    }
    catch (...)
    {
       // restore grid updates
       AGrid.Rows[0]->EndUpdate();
       throw;
    }
    // restore grid updates
    AGrid.Rows[0]->EndUpdate();
  }
}
```

Note the use of the `TextToCells()` function in the above code. This application-defined function simply parses the text buffer, copying the appropriate portion into the appropriate cell:

```
void __fastcall TextToCells(
  TStringGrid& AGrid,
  const TGridCoord& AFirstCell,
  const AnsiString AText)
{
  TGridCoord cell = AFirstCell;

  // if the text has no tab-delimiters,
  // copy it as a whole string
  if (!AText.Pos('\t'))
  {
    AGrid.Cells[cell.X][cell.Y] = AText;
    return;
  }

  //
  // parse the text by scanning its
  // contents, character by character,
  // looking for tab and CR delimiters
  //
  int word_start = 1;
  const int text_len = AText.Length();
  for (int index = 1;
       index < text_len;
       ++index)
  {
    char current_char = AText[index];
    // if it's a new column
    if (current_char == '\t')
    {
      AGrid.Cells[cell.X++][cell.Y] =
         AText.SubString(
            word_start,
            index - word_start
            );
      // skip the tab character
      word_start = index + 1;
    }
    // if it's a new row
    else if (current_char == '\r')
```

```
    {
      AGrid.Cells[cell.X][cell.Y++] =
         AText.SubString(
            word_start,
            index - word_start
            );
      cell.X = AFirstCell.X;
      // skip the CR/LF characters
      word_start = index + 2;
    }
  }
}
```

Both the `TextToCells()` and the `PasteCellsFromClipboard()` functions accept an `AFirstCell` parameter, which is used to specify the top-left corner of the paste destination. For this parameter, you'll likely want to pass the `TopLeft` coordinate of the `TStringGrid::Selection` property:

```
// OnKeyDown event handler
void __fastcall TForm1::
  StringGrid1KeyDown(TObject *Sender,
  WORD &Key, TShiftState Shift)
{
  if (Key == 'V' &&
      Shift.Contains(ssCtrl))
  {
    PasteCellsFromClipboard(
      *StringGrid1,
      StringGrid1->Selection.TopLeft
      );
  }
}
```

Drag and drop from other applications

Perhaps one of the most convenient means of entering data into Excel is by simply using the mouse. For example, you can select a word in WordPad or Internet Explorer and simply drag the text into a specific cell. (This feature is especially useful during the morning hours, when most of us have one hand occupied by a coffee cup). Unfortunately, while the `TClipboard` class can be

used to simplify the process of clipboard transfers, it has no drag and drop counterpart. Moreover, the VCL drag and drop framework won't work with other applications.
For inter-application drag-drop transfers, Excel and most other applications use OLE. Indeed, to enhance your string grid to accept text that's dragged from another application, you too will need to use OLE. But, don't let this paradigm intimidate you. Remember, we're only working with text here.

An IDropTarget descendant class

Accepting an OLE drag/drop transfer is accomplished via the `IDropTarget` interface. The Windows shell uses this interface to communicate with the window that's to receive the text (i.e., the "drop target"). To this end, let's create the simplest possible `IDropTarget` descendant class, `TSGDropTarget`. Here's the class declaration:

```
#include <ole2.h>
static const int
  WM_OLEDRAGOVER = WM_USER + 101;
static const int
  WM_OLEDRAGDROP = WM_USER + 102;

class TSGDropTarget : public IDropTarget
{
private:
  unsigned long num_refs_;
  TWinControl* TargetWnd_;
  TPoint PMouse_;
  bool format_ok_;

protected:
  // IUnknown member functions
  STDMETHOD(QueryInterface)(REFIID riid,
    void** ppvObj);
  STDMETHOD_(ULONG, AddRef)();
  STDMETHOD_(ULONG, Release)();

  // IDropTarget member functions
  STDMETHOD(DragEnter)(
    LPDATAOBJECT pDataObj,
    DWORD grfKeyState, POINTL pt,
    LPDWORD pdwEffect);
  STDMETHOD(DragOver)(DWORD grfKeyState,
    POINTL pt, LPDWORD pdwEffect);
```

```
  STDMETHOD(Drop)(LPDATAOBJECT pDataObj,
    DWORD grfKeyState, POINTL pt,
    LPDWORD pdwEffect);
  STDMETHOD(DragLeave)()
    { return E_NOTIMPL; };

public:
  TSGDropTarget(
    TWinControl* ATargetWnd_);
  ~TSGDropTarget();
};
```

The `WM_OLEDRAGOVER` and `WM_OLEDRAGDROP` messages serve as a means of communication between the `TSGDropTarget` class and the rest of the application. Specifically, these messages are sent to the `TWinControl` descendant that's indicated by the `TargetWnd_` member; this member is initialized via the constructor.
Note that we've haven't declared any new methods. The `QueryInterface()`, `AddRef()`, and `Release()` methods are standard to all `IUnknown` descendants. The `DragEnter()`, `DragOver()`, `Drop()`, and `DragLeave()` methods come from the `IDropTarget` class. As you'll soon see, these latter functions are not unlike the `TControl`'s `OnDragOver` and `OnDragDrop` events.

The DragEnter() method

The `DragEnter()` method is called when the user initially drags an object within the bounds of a window that has been previously registered as a drop target (which we'll do later with our string grid). It's the job of `DragEnter()` to decide whether a drop operation can be supported:

```
STDMETHODIMP TSGDropTarget::DragEnter(
  LPDATAOBJECT pDataObj,
  DWORD grfKeyState, POINTL pt,
  LPDWORD pdwEffect)
{
  FORMATETC fmtetc;
  fmtetc.cfFormat = CF_TEXT;
  fmtetc.ptd = NULL;
  fmtetc.dwAspect = DVASPECT_CONTENT;
  fmtetc.lindex = -1;
  fmtetc.tymed = TYMED_HGLOBAL;

  // does the source provide CF_TEXT?
```

```
  HRESULT hRes =
    pDataObj->QueryGetData(&fmtetc);
  if (SUCCEEDED(hRes))
  {
    format_ok_ = true;
    *pdwEffect = DROPEFFECT_COPY;
  }
  else
  {
    format_ok_ = false;
    *pdwEffect = DROPEFFECT_NONE;
  }
  return NOERROR;
}
```

How do we know whether to accept the drop? Well, since we're only supporting text, we simply fill a `FORMATETC` structure, specifying `CF_TEXT` as the format (i.e., CR/LF-delimited text), and `TYMED_HGLOBAL` as the storage medium (i.e., global memory). Next, we pass this information to the `IDataObject`'s `QueryGetData()` method, via the supplied `pDataObj` pointer, to determine whether the underlying data is indeed text. Finally, we communicate our decision back to the source via the `pdwEffect` parameter, in response to which, the source will (usually) change the mouse cursor. This is similar to the `Accept` parameter presented by the `OnDragOver` event of `TControl`.

The DragOver() method

As its name implies, the `DragOver()` method is repeatedly called while the user drags an object over the drop target. We'll implement this method to simply send the `WM_OLEDRAGOVER` message, along with the mouse cursor location (`PMouse_`), to the target window:

```
STDMETHODIMP TSGDropTarget::DragOver(
  DWORD grfKeyState, POINTL pt,
  LPDWORD pdwEffect)
{
  if (format_ok_)
  {
    // store the mouse cursor point
    PMouse_ = Point(pt.x, pt.y);
    // indicate the copy effect
    *pdwEffect = DROPEFFECT_COPY;
```

```
    //
    // notify the target window of
    // the drag over event
    //
    LPARAM lParam =
      reinterpret_cast<LPARAM>(
        &PMouse_
        );
    TargetWnd_->Perform(
      WM_OLEDRAGOVER, NULL, lParam
      );
  }
  // otherwise, indicate no effect
  else *pdwEffect = DROPEFFECT_NONE;
  return NOERROR;
}
```

The Drop() method

The `Drop()` method is called when the user drops the drag object within the bounds of the drop target. We'll implement this method to send the `WM_OLEDRAGDROP` message to the target window:

```
STDMETHODIMP TSGDropTarget::Drop(
  LPDATAOBJECT pDataObj,
  DWORD grfKeyState, POINTL pt,
  LPDWORD pdwEffect)
{
  if (format_ok_)
  {
    // initialize a FORMATETC structure
    FORMATETC fmtetc;
    fmtetc.cfFormat = CF_TEXT;
    fmtetc.ptd = NULL;
    fmtetc.dwAspect = DVASPECT_CONTENT;
    fmtetc.lindex = -1;
    fmtetc.tymed = TYMED_HGLOBAL;

    //
    // user has dropped on us--get
    // the text from drag source
    //
```

```
    STGMEDIUM smed;
    HRESULT hResult =
      pDataObj->GetData(&fmtetc, &smed);
    if (SUCCEEDED(hResult))
    {
      //
      // notify the target window
      // of the drop event
      //
      LPARAM lParam =
        reinterpret_cast<WPARAM>(
          smed.hGlobal
          );
      TargetWnd_->Perform(
        WM_OLEDRAGDROP, NULL, lParam
        );

      // free the associated memory
      ReleaseStgMedium(&smed);
    }
    else return hResult;
  }
  return NOERROR;
}
```

Since we've already verified the format of the data in the `DragEnter()` method, here we use the `GetData()` method to actually get the text. Specifically, this function will use the information contained in a `FORMATETC` structure and appropriately fill a `STGMEDIUM` structure. Since, we've specified `CF_TEXT` and `TYMED_HGLOBAL`, the `hGlobal` data member of `STGMEDUIM` will indicate a handle to a global memory object that contains the text. We pass this handle along with the `WM_OLEDEGDROP` message to the target window, then use the `ReleaseStgMedium()` function to free the global memory (remember, `Perform()` won't return until the target window has replied to the message).

Using the TSGDropTarget class

Now that we've created the `TSGDropTarget` class, let's put it to good use. First, in our form's header file, we declare an `IDropTarget` member pointer and map the `WM_OLEDRAGOVER` and `WM_OLEDRAGDROP` messages:

```
class TForm1 : public TForm
```

```
{
//...
private:
  IDropTarget* pDropTarget_;
  MESSAGE void __fastcall WMOleDragDrop(
    TMessage& AMsg);
  MESSAGE void __fastcall WMOleDragOver(
    TMessage& AMsg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_OLEDRAGDROP,
    TMessage, WMOleDragDrop)
  MESSAGE_HANDLER(WM_OLEDRAGOVER,
    TMessage, WMOleDragOver)
END_MESSAGE_MAP(TForm)
//...
};
```

Next, in our form's constructor, we create an instance of the `TSGDropTarget` class, passing a pointer to our form (`this`) as the `ATargetWnd_` parameter (remember, we want the `WM_OLE*` messages sent to our form, not the string grid). We also lock the `IDropTarget` object in memory, and then register the string grid as a drop target:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  OleInitialize(NULL);

  pDropTarget_ =
     reinterpret_cast<IDropTarget*>(
        new TSGDropTarget(this)
        );
  CoLockObjectExternal(
   pDropTarget_, true, true
   );
  RegisterDragDrop(
   StringGrid1->Handle, pDropTarget_
   );
}
```

Likewise, in our form's destructor, we reverse these operations: we un-register the string grid as a drop target, unlock the `IDropTarget` pointer, and then free the `IDropTarget` instance:

```
__fastcall TForm1::~TForm1()
{
  RevokeDragDrop(StringGrid1->Handle);
  CoLockObjectExternal(
   pDropTarget_, false, true
   );
  pDropTarget_->Release();

  OleUninitialize();
}
```

Our next task is to implement the message handlers. For the `WM_OLEDRAGOVER` message, we extract the coordinates of the mouse cursor, and then determine which cell lies at these coordinates, if any, via the `MouseToCell()` method of `TStringGrid`. This way, we can provide visual feedback by tracking the mouse cursor with the focused cell via the `Col` and `Row` properties:

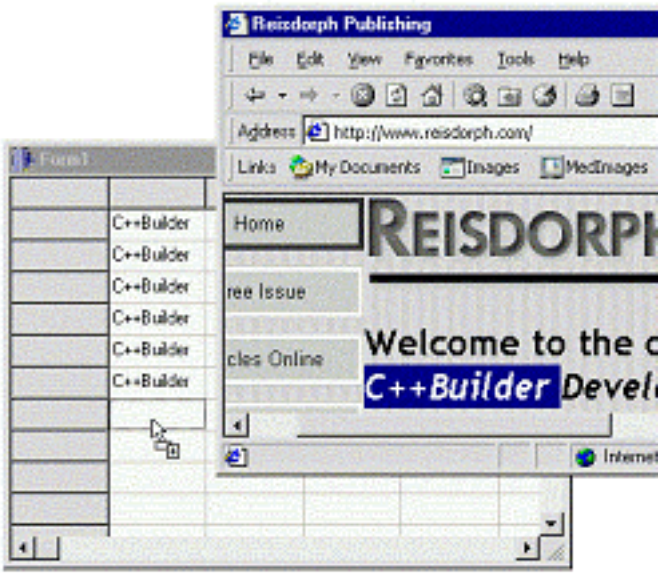
```
void __fastcall TForm1::WMOleDragOver(
  TMessage& AMsg)
{
  TPoint* pPMouse =
    reinterpret_cast<TPoint*>(
      AMsg.LParam
      );
  *pPMouse =
    StringGrid1->ScreenToClient(
      *pPMouse
      );

  int col, row;
  StringGrid1->MouseToCell(
    pPMouse->x, pPMouse->y, col, row
    );
  if (col >= StringGrid1->FixedCols &&
      row >= StringGrid1->FixedRows)
  {
    StringGrid1->Col = col;
```

```
    StringGrid1->Row = row;
  }
}
```

Finally, to handle the `WM_OLEDRAGDROP` message, we simply extract the handle to the global memory object from the `LParam` data member, grab a pointer to the text via the `GlobalLock()` function, and then copy this text to the target cell. The result is depicted in **Figure A**. Here's the code:

```
void __fastcall TForm1::WMOleDragDrop(
  TMessage& AMsg)
{
  HGLOBAL hData =
    reinterpret_cast<HGLOBAL>(
      AMsg.LParam
      );
  char* pText =
    reinterpret_cast<char*>(
      GlobalLock(hData)
      );
  try
  {
    AnsiString text(pText);
    GlobalUnlock(hData); hData = NULL;

    const int col = StringGrid1->Col;
    const int row = StringGrid1->Row;
    StringGrid1->Cells[col][row] = text;
  }
  catch (...)
  {
    if (hData) GlobalUnlock(hData);
    throw;
  }
}
```

Figure A



Copying text from Internet Explorer to a string grid via OLE-based drag and drop.

Conclusion

Enhancing a string grid's user-friendliness is more an art than a science. You actually need to work with the control yourself before guessing what the end-user may or may not want. Here, I've demonstrated the basics—how to support clipboard transfers and one-way drag and drop. Indeed the topic of dragging text *to* other applications is enough to fill another article in itself.

I encourage you to download the source code to this article (available from **www.bridgespublishing.com**) and examine the specifics of the `TSGDropTarget` class. And, if you have any further questions on string grids, feel free to drop me a line. My e-mail address is: **dmc27@ee.cornell.edu**.

Finding files, part 2

by Mark G. Wiseman

In part 1 of this series of articles, I introduced two classes, `TFFData` and `TFindFile`, that can be used to find files on a hard drive. Ultimately I will use these classes to develop a utility program that will delete the unnecessary files left by a C++Builder project. I would like to continue my discussion of the main class, `TFindFile`, in this article. I will also talk a little about other uses for `TFindFile` and another approach you might consider for finding files.

An updated TFindFile

Since writing part 1, I made a small change to the source code for `TFindFile`. I added a simple inline function, `IsRunning()`. This function simply returns `true` if `TFindFile` is currently searching for files and `false` if it is not. The updated code for `TFindFile` is in **Listings A and B**.
As you may recall, I suggested that `TFindFiles` might be rewritten to run in a separate thread. Although, I haven't done this, I do call the `ProcessMessages()` function of `TApplication` several places in the code for `TFindFile`. `ProcessMessages()` could be viewed as a simple form of multithreading. In either case I found it useful to test if `TFindFile` was actually running. What if a user of our utility program clicked on the program window's close button? The code segment below shows how we might use `IsRunning()`.

```
// findfile is an instance of TFindFile

void __fastcall TMainForm::FormCloseQuery(
  TObject *Sender, bool &CanClose)
{
  if (findFile.IsRunning() == false) return;

  if (Application->MessageBox(
    "Stop search and exit?",
    "Find Files", MB_YESNO) == IDYES)
      findFile.Stop();
  else
    CanClose = false;
}
```

Searching subfolders

The `Find()` function in the `TFindFile` class searches for a file pattern that is passed to it in a `String` argument. If the `SearchSubfolders` property is set to `true`, `Find()` will also search all subfolders.
The `SearchSubfolders` property is the public interface for the private `bool` variable

searchSubfolders. After searching the starting folder for files, Find() tests searchSubfolders and if it is true, Find() will continue searching into any subfolders of the starting folder.
The first thing Find() does when searching subfolders, is to create a TFFStack object named dirStack. You will recall from part 1 that using TFFStack allows us to search through subfolders without using recursion.
Find() then enters a while loop that will run as long as stop is false. Within the while loop, Find() calls the private function FindDirs(), passing in the current folder and a reference to dirStack. The FindDirs() function uses the Windows API to find all folders in the current folder and pushes each one of these folders onto the dirStack.
Find() then tests dirStack to see if it is empty. If dirStack is empty, then there are no more subfolders to search and Find() will break from the while loop, set stop equal to true and return. If dirStack is not empty, Find() pops the next subfolder off of dirStack, sets the current folder to that subfolder and then calls FindFiles() to continue the search.

Skipping folders

I have included a property, OnSearchFolder in TFindFile. This property hold a closure, a pointer to a function of the typedef TSearchFolder. Here is that typedef:

```
typedef void __fastcall
  (__closure  *TSearchFolder)
  (TFindFile *Sender, String folderName,
  bool &skip);
```

If a function has been assigned to OnSearchFolder, the Find() function will call it before it searches a folder. When Find() calls the function, it sets the skip variable equal to false. When the function returns, Find() checks skip and if skip is true, Find() will not search the folder.
The ability to skip a folder can be very useful. Remember, we are going to end up with a program that deletes files from C++Builder project folders. On my system, most C++Builder projects are stored in subfolders of “c:\\Projects”. If my current project is very large, I may not want to delete the files for that project each night when I run the utility. If the current project resides in a subfolder named “Current”, I could use this code to skip that subfolder.

```
void __fastcall
TMainForm::OnSearchFolder(
  TFindFile *Sender,
  String folderName, bool &skip)
{
  if (folderName ==
      "c:\\Projects\\Current\\")
    skip = true;
}
```

I can also use the `OnSearchFolder` closure function to display the name of the current folder being searched.

```
void __fastcall
TMainForm::OnSearchFolder(
  TFindFile *Sender,
  String folderName, bool &skip)
{
  StatusBar->Panels->Items[0]->Text =
    folderName;
}
```

Finding files

So, what happens when the `Find()` function of `TFindFile` finds a file? I have included another property, `OnFileFound`, that holds a closure for a function of the `typedef` `TFileFound`. Here is that `typedef`:

```
typedef void __fastcall
  (__closure *TFileFound)
  (TFindFile *Sender, String fileName,
  String foldername, TFFData data);
```

This is where the rubber meets the road. When the `Find()` function finds a file that matches the pattern we have specified, it will call the function held in the `OnFileFound` property, passing the file's name and the `TFFData` object associated with the file. We can have this function do anything we want with this file. For instance, lets delete each file that is found.

```
void __fastcall TMainForm::OnFileFound(
  TFindFile *Sender, String fileName,
  String folderName, TFFData data)
{
  if (data.IsFolder()) return;

  String filePath =
    folderName + fileName;
  DeleteFile(filePath);
}
```

Notice two things about how `TFindFile` works: first, the file that is found may actually be a folder that matches the file pattern specified in the call to `Find()`. You need to test for this.
Second, a copy of the `TFFData` associated with the files is passed in as an argument. This can be very

useful. For example, using information contained in `TFFData`, we could easily modify our function to delete only those files of size 0 bytes:

```
void __fastcall TMainForm::OnFileFound(
  TFindFile *Sender, String fileName,
  String folderName, TFFData data)
{
  if (data.IsFolder()) return;

  String filePath =
    folderName + fileName;
  if (data.GetSize() == 0)
    DeleteFile(filePath);
}
```

In every program where you use `TFindFile`, you will almost certainly have to write a function of `typdef TFileFound` and assign it to the `OnFileFound` property of `TFindFile`. Otherwise, there is not really a good reason to include `TFindFile`.

Other uses

In this series of articles, I will use `TFindFile` for one specific purpose – deleting certain C++Builder project files. However, I designed `TFindFile` to be used in any kind of program that needs to search a drive for files.
As hinted at above you could use `TFindFile` to scan an entire hard drive to find all files of size zero, list the files for the user and allow them to decide which files should be deleted.
Along with some file compression code, you could use `TFindFile` to write a custom backup program. What I am trying to say, is there are lots of ways you can use `TFindFile`. Don't just think of it as part of our utility program.

Possible changes

There are modifications you might want to make to `TFindFile`. As I've already mentioned, you might want to rewrite it to use a separate thread.
Also, instead of using the Windows API functions for finding files, you could use the Windows Shell functions. This approach might offer faster searching and you might be able to search for things other than files (e.g. computers and printers).
There is one modification that I have seriously consider making to `TfindFile`. I would like for `TFindFile` to accept wild cards for drives, both on the local machine and on the network. So, instead of calling `Find()` for each drive on a computer or network, you would only have to call `Find()` once. For example:

```
// Current implementation of TFindFile
```

```
findFile.Find("c:\\*.*");
findFile.Find("d:\\*.*");
findFile.Find("e:\\*.*");

// Wouldn't this be nice!
// Search c, d and e drives
FindFile.Find("?:\\*.*);
```

Making a modification to allow drive wild cards on a single computer should be easy. Wild cards for network drives would be harder, but still do-able.

Conclusion

This concludes my discussion of `TFindFile`. In my next article, I am going to show you the code for actually deleting C++Builder project files. The approach I use for finding the files, using `TFindFile` of course, may be a little surprising. As you will see, I only call the `Find()` functions with this pattern, "*.*". Read my next article to find out why.

Listing A: *`FindFile.h`*

```
#ifndef FindFileH
#define FindFileH

#include "FFData.h"

class TFFStack;

typedef void __fastcall (__closure *TFileFound)
  (TFindFile *Sender, String fileName,
   String foldername, TFFData data);
typedef void __fastcall (__closure *TSearchFolder)(TFindFile *Sender,
  String folderName, bool &skip);

class TFindFile
{
  public:
    TFindFile();

    void Find(String filePattern);
    void Stop();
    bool IsRunning();

    __property bool SearchSubfolders =
```

```
     {read = searchSubfolders,
       write = searchSubfolders};
    __property TFileFound OnFileFound =
     {read = FOnFileFound, write = FOnFileFound};
    __property TSearchFolder OnSearchFolder =
     {read = FOnSearchFolder,
       write = FOnSearchFolder};

  private:
    void FindFiles(String filePattern);
    void FindDirs(String baseDir, TFFStack &stack);

    TFileFound FOnFileFound;
    TSearchFolder FOnSearchFolder;

    bool stop;
    bool searchSubfolders;
};

inline TFindFile::TFindFile()
{
  FOnFileFound = 0;
  FOnSearchFolder = 0;

  stop = true;
  searchSubfolders = false;
}

inline void TFindFile::Stop()
{
  stop = true;
}

inline bool TFindFile::IsRunning()
{
  return(stop == false);
}


#endif   // FindFilesH
```

Listing B: *FindFile.cpp.*

```
#include <vcl.h>
#pragma hdrstop

#include "FindFile.h"


class TFFStack
{
  public:
    TFFStack();
    ~TFFStack();

    void Push(AnsiString name);
    AnsiString Pop();

    void Empty();

    bool Contains(String name);
    bool IsEmpty();

  private:
    TStringList *list;
};

inline TFFStack::TFFStack()
{
  list = new TStringList;
  list->Sorted = true;
  list->Duplicates = dupIgnore;
}

inline TFFStack::~TFFStack()
{
  delete list;
}

inline void TFFStack::Push(AnsiString name)
{
  list->Add(name);
}

inline AnsiString TFFStack::Pop()
{
```

```
  AnsiString temp = list->Strings[0];
  list->Delete(0);
  return(temp);
}

inline void TFFStack::Empty()
{
  list->Clear();
}

inline bool TFFStack::Contains(String name)
{
  int index;
  return(list->Find(name, index));
}

inline bool TFFStack::IsEmpty()
{
  return(list->Count == 0);
}


// ----------------------------------------

void TFindFile::Find(String filePattern)
{
  stop = false;

  String curDir = ExtractFilePath(
    ExpandFileName(filePattern));
  String fileName = ExtractFileName(filePattern);

  bool skip = false;
  if (FOnSearchFolder)
    FOnSearchFolder(this, curDir, skip);
  if (FOnFileFound && skip == false)
    FindFiles(curDir + fileName);

  if (searchSubfolders)
  {
    TFFStack dirStack;

    while (stop == false)
    {
```

```
      Application->ProcessMessages();

      FindDirs(curDir, dirStack);
      if (dirStack.IsEmpty()) break;

      curDir = dirStack.Pop();

      skip = false;
      if (FOnSearchFolder)
        FOnSearchFolder(this, curDir, skip);
      if (skip) continue;

      if (FOnFileFound) FindFiles(curDir + fileName);
    }
  }

  stop = true;
}

void TFindFile::FindDirs(
  String baseDir, TFFStack &stack)
{
  TFFData ffData;

  String dirPattern = baseDir + "*.*";

  HANDLE handle = FindFirstFile(
    dirPattern.c_str(), &ffData.data);
  if (handle != INVALID_HANDLE_VALUE)
  {
    bool found = true;

    while (found == true && stop == false)
    {
      Application->ProcessMessages();

      if (ffData.IsFolder() &&
          ffData.GetName() != "." &&
          ffData.GetName() != "..")
        stack.Push(
          baseDir + ffData.GetName() + "\\");

      found = (FindNextFile(handle, &ffData.data) != 0);
    }
```

```
    FindClose(handle);
  }
}

void TFindFile::FindFiles(String filePattern)
{
  TFFData ffData;

  HANDLE handle = FindFirstFile(
    filePattern.c_str(), &ffData.data);
  if (handle != INVALID_HANDLE_VALUE)
  {
    bool found = true;

    while (found == true && stop == false)
    {
       Application->ProcessMessages();

       if (FOnFileFound &&
           ffData.GetName() != "." &&
           ffData.GetName() != "..")
         FOnFileFound(this, ffData.GetName(),
           ExtractFilePath(filePattern), ffData);
       found = (FindNextFile(
         handle, &ffData.data) != 0);
    }

    FindClose(handle);
  }
}


#pragma package(smart_init)
```

Using timers

by Kent Reisdorph

Some applications require code to be executed at a specific time interval. Timers are used for this purpose. Windows is, obviously, an event-drive operating system. For this reason you must deal with the timers provided by Windows. Put another way, you can't get directly to hardware interrupts for timing purposes. The exception to this is if you write a device driver for a specific piece of hardware, but that is beyond the scope of this article.

Windows comes with two basic types of timers: the system timer and the multimedia timer. I'll explain the differences between these timer types, and the implications of using both.

One of the problems with an article on timers is that it is difficult to come up with a timer example that depicts a real-world programming situation. Most articles on timers create a clock and update its display every second. Creating a clock is a convenient way to illustrate the use of timers, but it isn't particularly useful. I won't create a clock in this article. What I will do is show you the basics of using timers and leave it to you to decide when and where to use them.

How timers work

The concept behind timers is simple. You set an interval for the timer and activate it. The timer will fire each time the interval period elapses. You specify a callback function when you create the timer. The callback function will be called whenever the timer fires. In the case of the Windows system timer, you can opt to have a `WM_TIMER` message sent to your application's Window procedure each time the timer fires rather than using a callback function.

In the days of Windows 3.1 you had to be careful using timers. There were only so many timers available, and you never knew for sure whether you would get a timer handle from the system when you requested one. With 32-bit Windows, however, there is virtually no limit to the number of timers available. Further, today's hardware and operating systems handle dozens or hundreds of timers simultaneously without difficulty.

Using the system timer

Of the two timers provided by Windows, the system timer is by far the easiest to use. It is also the lease accurate.

Limitations of the system timer

There are two important things that you should know about the system timer. First, this timer uses the system clock to perform timing. As such, the minimum resolution (the lowest interval you can effectively use) is about 55 milliseconds. You can certainly set the timer interval to a value less than 55, but the results will be unpredictable. This is particularly true on Windows 95 and 98. Windows NT and 2000 are less susceptible to this, but it is still not advisable to depend on the

system timer for critical applications.
The second thing you need to be aware of when using the system timer is that it uses the `WM_TIMER` message to notify you each time the interval elapses. Of all the messages in Windows, `WM_TIMER` is one of the lowest priority messages. If the system is busy, `WM_TIMER` messages may be delayed. In some cases `WM_TIMER` messages may be removed from the message queue altogether. This means that a given timer event may not occur at all if the system is busy. Considering these two factors, the system timer is potentially both inaccurate and unreliable. However, that is not to say that the system timer is unusable. Many tasks for which you will use a timer don't require a high degree of accuracy or reliability. Let's say you have an application that performs some service at 1:00 AM each day (a backup operation, for example). How do you know when it's 1:00 AM? You could use a timer that fires once per second and checks the current time. If the time is determined to be 1:00, then some processing is performed. In this example it doesn't much matter that the system timer is a bit inaccurate, nor does it matter that a timer message may be lost by Windows.

Putting the system timer to use

The most efficient way to access the system timer is to simply use the VCL `TTimer` component. `TTimer` is very simple to use. Just drop one on the form and set its `Interval` property to the timer interval you wish to use. The `Interval` property is specified in milliseconds. To create a timer that fires each second, for example, set the `Interval` property to `1000`. Next create an event handler for the `OnTimer` event. Place code in the event handler that you wish to execute each time the timer fires. The timer can be enabled or disabled by setting the `Active` property to either `true` or `false`.
The other way to create and use a system timer is using the Win32 API. The `SetTimer()` function is used to start a timer and to set the timer interval. The `KillTimer()` function is used to stop the timer and free resources associated with the timer. There's really no need to use the API to manage timers when you have the `TTimer` component. `TTimer` is simply a wrapper around the Win32 API functions.

Using the multimedia timer

As I have said, the system timer is both inaccurate and potentially unreliable. Fortunately a solution is available in the multimedia timer provided by Windows. Don't let the name fool you; the multimedia timer can be used for any purpose at all. It is not limited to multimedia processes. The multimedia timer has a resolution of 1 millisecond. Further, it is not subject to lost timer messages as the system timer is. This makes the multimedia timer a good candidate for timing needs in all but the most demanding of situations.

Starting and stopping the timer

The multimedia timer functions are found in the MMSYSTEM.H header. You may or may not have

to explicitly include this header, depending on the version of C++Builder you have. To use the multimedia timer you first call `timeBeginPeriod()` to set the minimum timer resolution you need. For example:

```
timeBeginPeriod(1);
```

Next you call the `timeSetEvent()` as shown here:

```
MMRESULT timerID;
...
timerID = timeSetEvent(
  1, 0, TimeProc, 0, TIME_PERIODIC);
```

The first parameter is used to specify the timer interval. In this case I have specified an interval of one millisecond. The second parameter is used to set the delay. You should pass zero for this parameter, indicating that you want the greatest accuracy possible. The third parameter is used to pass the address of the callback function that will be called when the timer fires. I will explain the callback function in the next section. The third parameter is used to pass user-specified data to the callback function. You can pass anything you want for this parameter, including a pointer to an object. The last parameter is used to specify the type of timer. The possible values are `TIME_ONESHOT` and `TIME_PERIODIC`. Passing `TIME_ONESHOT` for this parameter indicates that the timer is a one-shot timer; the timer fires at the end of the specified interval and never fires again. The `TIME_PERIODIC` value is used when creating a timer that will fire repeatedly, once for each time the specified timer interval passes. `timeSetEvent()` returns zero if an error occurred, or a timer handle if the function succeeded. You will use this handle later to stop the timer.
To stop the timer, call `timeKillEvent()`, passing the handle of the timer you received when you called `timeSetEvent()`. For example:

```
timeKillEvent(timerID);
```

You must also call `timeEndPeriod()`, passing the same value you passed to `timeBeginPeriod()` as shown here:

```
timeEndPeriod(1);
```

Each call to `timeBeginPeriod()` must have a matching call to `timeEndPeriod()`.

The callback function

As I said earlier, the callback function is called once for each timer event. The callback function has the following signature:

```
void CALLBACK TimeProc(
  UINT uID, UINT uMsg, DWORD dwUser,
  DWORD dw1, DWORD dw2);
```

The first parameter is the timer ID of the timer that generated the event. This allows you to use multiple timers with a single callback function (not that you would want to). The second parameter is reserved by Windows and is not used. The third parameter is used to pass user-defined data to the callback. The data you passed in the call to `timeSetEvent()` will be passed to the callback function in this parameter. The final two parameters are also reserved and should not be used.
The callback function must be a stand-alone function; it cannot be a class member function. The Windows documentation for the callback function says that you should call only the `PostMessage()`, `timeGetSystemTime()`, `timeGetTime()`, `timeSetEvent()`, `timeKillEvent()`, `midiOutShortMsg()`, `midiOutLongMsg()`, and `OutputDebugString()` functions from within the callback itself. At first glance this of approved functions looks very restrictive. However, the key is the `PostMessage()` function. What you typically do is create a user-defined message. Then you create a handler for that message in your application's main form class. From within the callback you post yourself a message by calling `PostMessage()`. This way the handler for the user-defined message does all the work. You can call any functions you like from the message handler for the user-defined message.

Lean and mean code

Regardless of the timer used, care must be taken to ensure that the code that executes as the result of a timer event is short and concise. This is not typically necessarily if your timer has a long interval. If the timer interval is short, however, you need to be sure your code his time to execute before the next timer event comes along.
With today's machines it is surprising how much code you can execute in one millisecond. Still, if you are using a timer with a one millisecond interval you should know how much time your code takes to execute. A good profiler is indispensable in determining this. I used TurboPower's Sleuth QA Suite 2 when writing this article. Sleuth QA Suite 2 has both method and line profilers, a memory-checking tool, a coverage analyzer, and a macro recorder for automating testing tasks. You can find out more about Sleuth QA Suite 2 at **www.turbopower.com**.

Conclusion

Timers can be used for a variety of purposes. For most applications the system timer, accessed via `TTimer`, works just fine. For applications that require more accuracy and dependability, the multimedia timers are the way to go.

Validating URLs

by Mark G. Wiseman

Writing programs which utilize the Internet has become very popular. There are numerous freeware, shareware and commercial libraries and components that can be used to make programming for the Internet easier. However, did you know you may already have a good library of Internet functions?

The Windows' Internet API

If you have Microsoft Internet Explorer on your computer, you have the Windows' Internet API. This API is housed in the WININET.DLL file and is defined in the WININET.H header file. These are the same Internet access functions that Internet Explorer uses.

As far as I could tell there is no documentation on the API included with C++Builder. However, you can find everything you need to know on Microsoft's MSDN web site. Here is a good link to start with: msdn.microsoft.com/workshop/networking/wininet/reference/functions/general.asp.

You can use the Windows' Internet API in your code by including the WININET.H file and linking in the WININET.LIB file. Both of these files are included with C++Builder. As I mentioned before, you will need Internet Explorer for the WININET.DLL file. It is part of Internet Explorer 4.0 or later.

Using the API

Let's try out the API, by writing a simple function to validate a URL. The `ValidURL()` functions takes an `AnsiString` argument containing a URL to be tested and returns `true` if the URL is a valid URL or `false` if it is not.

The code for the `ValidURL()` function can be found in **Listings A and B**. As you can see, it is really very simple.

Listing A: *ValidURL.h.*

```
#ifndef ValidURLH
#define ValidURLH


bool ValidURL(String url);


#endif   // ValidURLH
```

Listing B: *ValidURL.cpp.*

```
#include <vcl.h>
#pragma hdrstop

#include "ValidURL.h"

#include <wininet.h>
```

```
bool ValidURL(String url)
   {
   bool result = false;

   HINTERNET hSession = InternetOpen("ValidURL", INTERNET_OPEN_TYPE_PRECONFIG, 0, 0,
0);
   if (hSession != 0)
      {
      HINTERNET hFile = InternetOpenUrl(hSession, url.c_str(), 0, 0,
INTERNET_FLAG_RELOAD, 0);
      if (hFile != 0)
         {
         int code = 0;
         DWORD codeLen = sizeof(int);
         HttpQueryInfo(hFile, HTTP_QUERY_STATUS_CODE | HTTP_QUERY_FLAG_NUMBER, &code,
&codeLen, 0);

         result = code == HTTP_STATUS_OK || code == HTTP_STATUS_REDIRECT;

         InternetCloseHandle(hFile);
         }

      InternetCloseHandle(hSession);
      }

   return(result);
   }

#pragma package(smart_init)
```

The `ValidURL()` function makes four calls into the Windows' Internet API, `InternetOpen()`, `InternetOpenUrl()`, `HttpQueryInfo()` and `InternetCloseHandle()`. Be sure to look at the documentation for these four functions on the MSDN website previously mentioned.

First, `ValidURL()` calls the `InternetOpen()` function. This function initializes the Internet API for use by your application. It takes several arguments and returns an `HSESSION`, a handle to an Internet session. If this return value is zero, then `InternetOpen()` has failed for some reason. If a valid session handle is returned, you must remember to close it when you are finished with it. You do this by calling `InternetCloseHandle()`.

The `ValidURL()` function tests for a good session handle. If there was an error and the session handle is equal to zero, `ValidURL()` reports an invalid URL. This is technically not correct, since the actual URL was not tested. If you would like to make this function more robust, you could throw an exception in `ValidURL()` instead of returning false.

If `InternetOpen()` returns a valid session handle, `ValidURL()` then calls `InternetOpenUrl()`. The `InternetOpenUrl()` function can open resources on URLs for FTP, Gopher or HTTP. Like `InternetOpen()`, the `InternetOpenUrl()` function returns a non-zero handle that must be closed when no longer needed.

If the value returned by `InternetOpenUrl()` is zero, an error has occurred and `ValidURL()` will report an invalid URL. If the return value is a valid handle, `ValidURL()` then calls `HttpQueryInfo()`.

The `HttpQueryInfo()` function specifically tests the URL. One of the arguments to `HttpQueryInfo()` is a pointer to

an `int` variable named `code`. The value placed in this variable by `HttpQueryInfo()` should be either `HTTP_STATUS_OK` or `HTTP_STATUS_REDIRECT` if the URL is valid.

Conclusion

The Bridges Publishing web site has a small application that uses the `ValidURL()` function. Admittedly, the `ValidURL()` function in this article is probably the simplest Internet program you can write. However, it is a useful function and a great way to get your feet wet in Internet programming.

Active Server Objects

by Bob Swart

One of the new features of C++Builder 5 is support for Active Server Pages, or more specifically, Active Server Objects that can be used inside Active Server Pages. In this article I will show you just how far this support goes, and what's missing (or what could be desired in future versions of C++Builder).

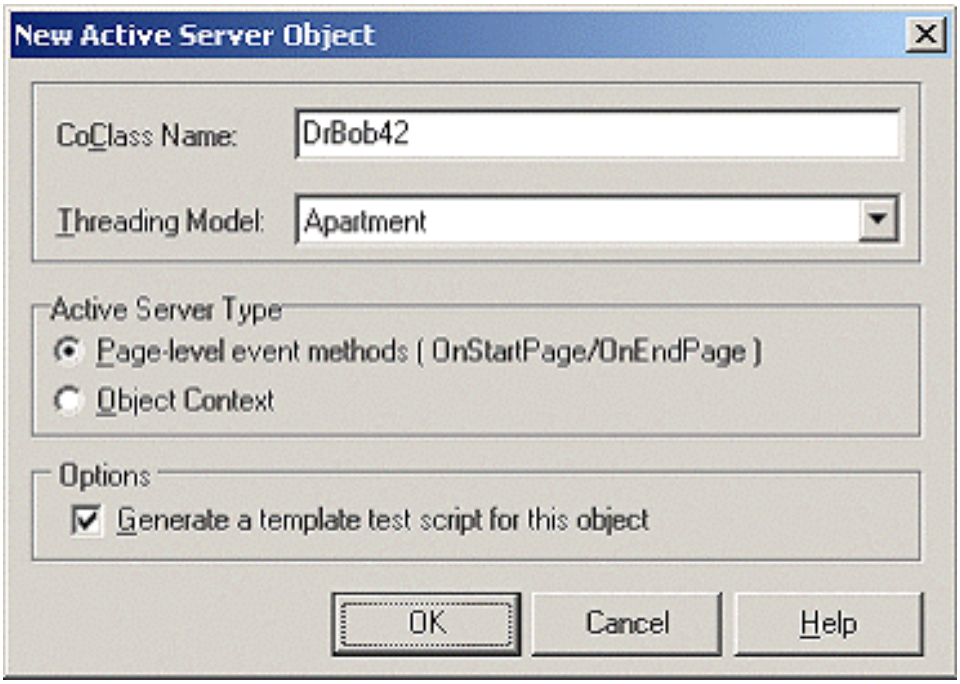
Active Server Pages

Active Server Pages (ASP) is a Microsoft solution to provide a server-side scripting language (compared with JavaScript on the client side). An Active Server Page is just a simple HTML Web page with an .asp extension and ASP scripting statements between <% and %> tags. Numerous books and articles are available on the ASP scripting language, which has little to do with C++Builder. The only overlap between the two is that fact that the ASP scripting languages can create and access Active Server Objects; COM objects that you can create using C++Builder 5.

Active Server Objects

To facilitate the use of Active Server Objects, C++Builder 5 includes an Active Server Object Wizard. Before employing the wizard, you must first create a new ActiveX Library project (using File|New, and selecting the ActiveX Library from the ActiveX tab). Make sure to save your project, for example as BCB5DJ.BPR, before you call the Active Server Object Wizard.
Once you've saved your new empty ActiveX Library, you can start the Active Server Object Wizard from the same ActiveX tab of the Object Repository. A combo box will prompt you to provide specific information, as shown in **Figure A**.

Figure A



The Active Server Object Wizard is where you set the object's properties.

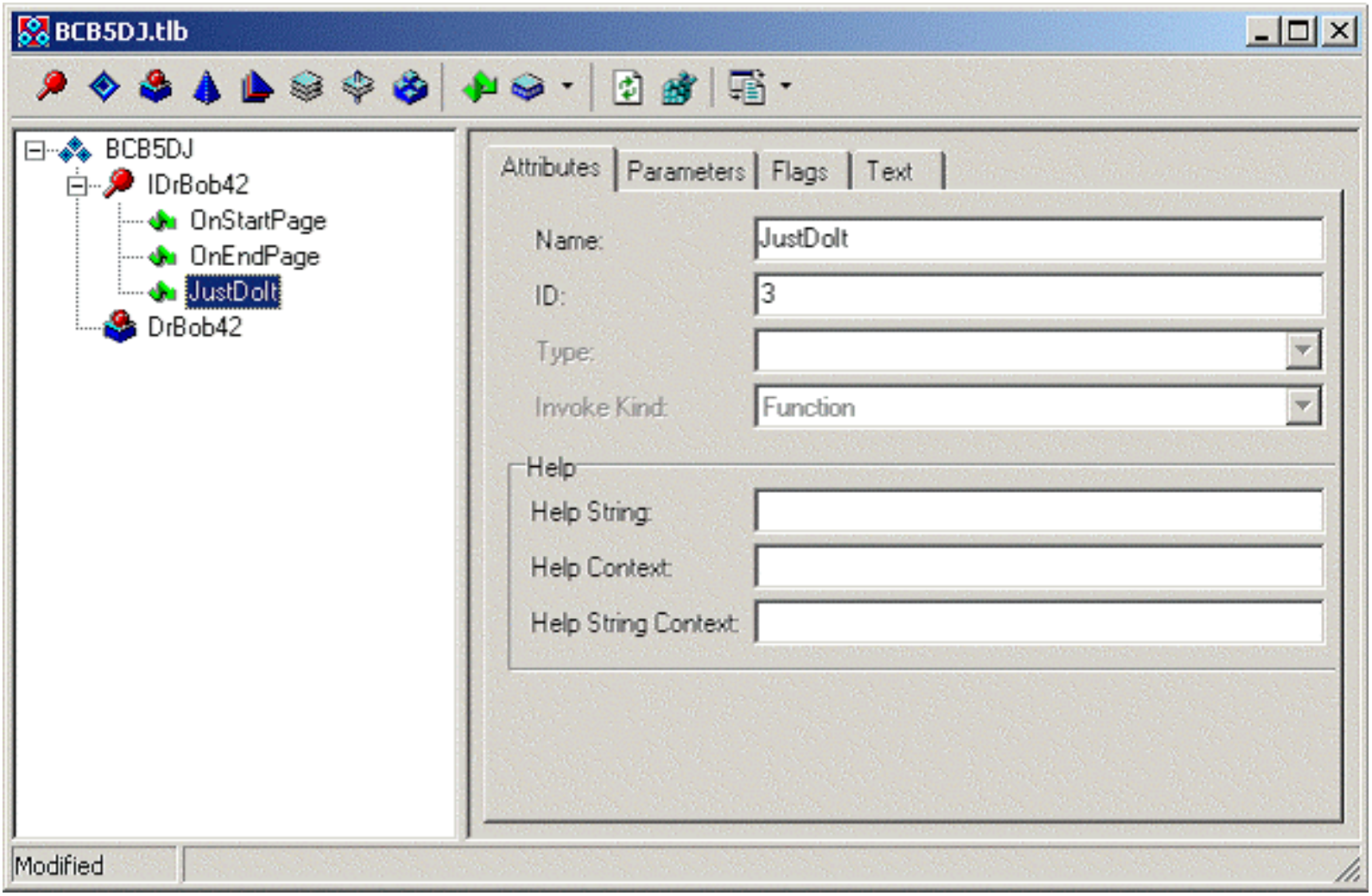
The "CoClass Name" field is used to specify the internal name of the new Active Server Object that this dialog will create for you. I've named it DrBob42 here, but you can use anything you like. The Active Server Type option gives you two choices: The default "Page-level event methods" type, or the "Object Context" type. The option you choose depends on the version of ASP that your Web server supports. Internet Information Server (IIS) 5 supports ASP 3.0, which can work with both options. However, IIS 4 supports ASP 2.0, which only supports the page-level event methods. To avoid any compatibility issues, I'll just stick with the default choice of page-level event methods. Note that the dialog already mentions the `OnStartPage` and `OnEndPage` events; those are indeed what we'll be using to work with our Active Server Object.

The final option, marked by default as well, helps to generate an example .asp file with a two-line script that shows you how to create your object and call a certain method from it. This ASP file will have the name of your project (provided through the wizard) with an .asp extension. This is why I recommended that you save your project before you started this dialog.

When you click OK, the skeleton for your Active Server Object is created (along with the sample .asp file if you left that option marked). The first thing you'll see now is the Type Library Editor. The Active Server Object is still "empty" and you must now define the ways the Active Server Object can interact with the outside world. Specifically, you must define how the Active Server Object is used by ASP scripting statements. If you expand the `IDrBob42` interface in the Type Library Editor, you'll notice the two methods that are already available: `OnStartPage` and `OnEndPage`.

At this point we'll create a single method in the Type Library that will do all the work for us. In this case, I have called the method `JustDoIt()`. The Type Library Editor should now display three methods as shown in **Figure B**.

Figure B



The Type Library Editor shows the newly-added JustDoIt() method.

Click the Refresh button on the Type Library Editor to update your C++ source code, and then close the Type Library Editor. This is a good time to save your entire project again using File|Save All. You can accept the default filenames C++Builder suggests. One of the files you just saved contains the implementation of the Active Server Object. In my case, it's called DRBOB42IMPL.CPP. This file contains the implementation of the `OnStartPage` and `OnEndPage` methods, as well as the implementation of the `JustDoIt()` method that we now need to implement. Before we can (or know how to) do that, you first need to know a little bit more about the available Active Server "support" objects that are available to you. The two objects I will explain here are `Request` and `Response`.

The Response object

The `Response` object is contained within the Active Server Object. We can use `Response` and its properties/methods to produce dynamic output. A straightforward way is to call the `Response->Write()` method that takes a `Variant` as argument. Using `Response`, the `JustDoIt()` method can be implemented like this:

```
STDMETHODIMP TDrBob42Impl::JustDoIt()
{
```

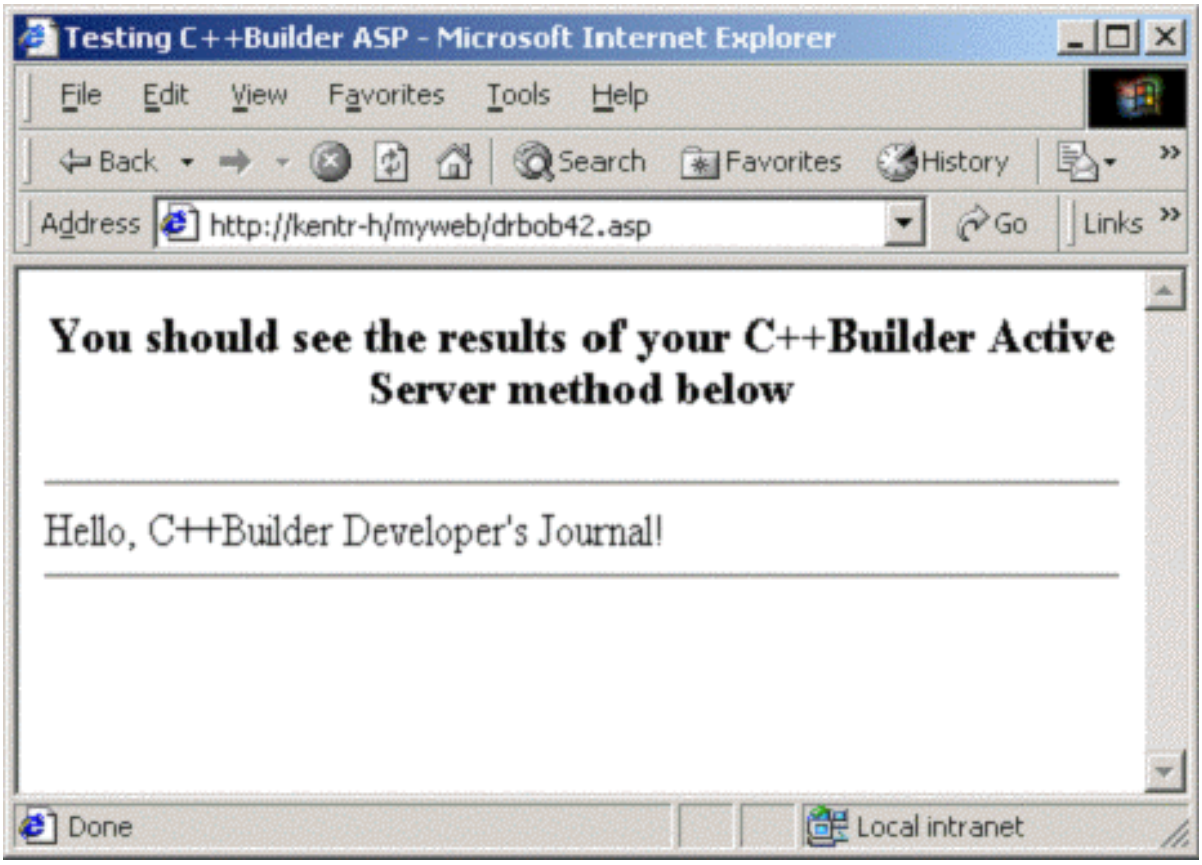
```
  return Response->Write(Variant(
    "Hello, C++Builder "
    "Developer's Journal!"));
}
```

After you compile the project, you need to register the Active Server Object. This is done by choosing Run|Register ActiveX Server from the C++Builder main menu. (You can unregister the object using Run|Unregister ActiveX Server.) After choosing this menu item you will receive notification that the ActiveX Server has been successfully registered.
In order to test this example, you need to take a look at the DRBOB42.ASP file that creates the `BCB5DJ.DrBob42` Active Server Object and contains an example to call a method. In our case, the method is `JustDoIt()`, of course, so the modified DRBOB42.ASP looks as follows:

```
<HTML>
<BODY>
<TITLE> Testing C++Builder ASP </TITLE>
<CENTER>
<H3> You should see the results of your
  C++Builder Active Server method below
</H3>
</CENTER>
<HR>
<% Set CBuilderASPObj = Server.CreateObject("BCB5DJ.DrBob42")
   CBuilderASPObj.JustDoIt
%>
<HR>
</BODY>
</HTML>
```

After you've made this modification, there's one important step you have to take: The DRBOB42.ASP file must be placed in a (virtual) directory of your Web server that has scripting rights assigned to it. This means that when you request the document from within a browser, the Web server itself will provide the page to you, executing the ASP scripting code along the way. An Active Server Page will not be "executed" if you load it as a local file inside your browser; the Web server must be involved. On my machine, the CGI-BIN directory has scripting rights, so I can copy DRBOB42.ASP to that directory and request http://localhost/cgi-bin/DrBob42.asp inside my browser. Doing so results in the Web page shown in **Figure C**.

Figure C



The ASP object serves up this Web page.

At this time, you've completed a full cycle of coding, compiling and using the resulting Active Server Object in a Web browser. Of course, in real-life you'll want to return something more entertaining than just the simple "hello" message. Before you go back to the coding editor, however, you should know that once loaded, an ASP 2.0 object is really hard to unload. And with the ASP object loaded (or more specifically, with the BCB5DJ.DLL loaded), you cannot recompile your project again because the BCB5DJ.DLL is in use and hence locked. In order to unload the BCB5DJ.DLL, you have to shut down the IIS Admin Service, and then restart it again (and the World Wide Web Service as well). To make things easier, I always use a batch file that unloads all Active Server Objects and reloads the Web server again.
Note, by the way, that you will not have this problem when using ASP 3.0 or when wrapping the Active Server Object into a Microsoft Transaction Server (MTS) object. However I'm not using MTS, so I can only show you the "hard way" here.

The Request object

Once you've unloaded BCB5DJ.DLL, it's time to return to C++Builder and start working with another important Active Server supporting object: `Request`. While `Response` can be used to send output back to the client, `Request` is used to obtain input information from the client. There are three main properties of the `Request` object: `Form` (the fields sent using the `POST()` method), `QueryString` (the fields sent using the `GET()` method) and `Cookies`.
Unfortunately, using these properties is not straightforward, as you first have to extract these items before you can use them. For example, to get the form from the `Request` object, and obtain the value

of the Choice form field, you'd have to write the following code:

```
STDMETHODIMP TDrBob42Impl::JustDoIt()
{
  HRESULT hr = E_FAIL;
  try
  {
    Response->Write(Variant(
      "Your choice is: "));
    Variant Choice;
    IRequestDictionary* Form;
    Request->get_Form(&Form);
    Form->get_Item(
      Variant("Choice"),Choice);
    hr = Response->Write(Choice);
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IDrBob42);
  }
  return hr;
}
```

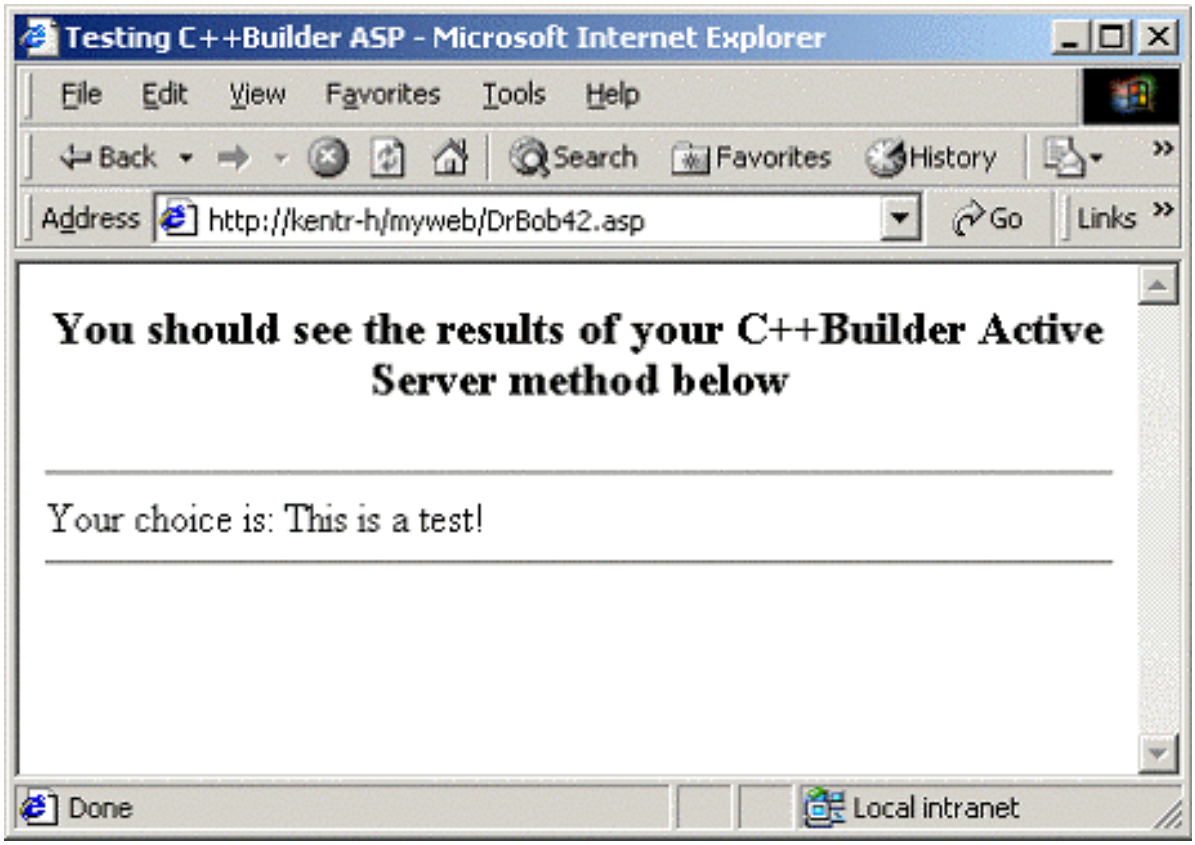
You have access to QueryFields (using the get_QueryFields() method) and Cookies (via the get_Cookies() method) in the same way. Once you know how, it works just fine. Note that we now also use a try/catch block to make sure no exception is raised (and not caught by the Active Server Object).
When deploying the above solution, you need some way to actually put data inside the Choice form variable. We can do this using a "calling" HTML file in which we define the correct action (our Active Server Page), the POST() method, and an input field named Choice. I've written the following HTML Web page for that purpose. It is placed in the same directory as the Active Server Page:

```
<HTML>
<BODY>
<H1>WebBroker HTML Form</H1>
<HR>
<FORM ACTION="DrBob42.asp" METHOD=POST>
<P>
Choice: <INPUT TYPE=text NAME=Choice>
<P>
<INPUT TYPE=SUBMIT VALUE=Submit>
```

```
</FORM>
</BODY>
</HTML>
```

When you load the above Web page, you can fill in some choice (any text will do) and click on the Submit button. When you do, you should see a page similar to the one shown in **Figure D**. The value of `Choice` is obtained from the form variables.

Figure D



The results of the ASP object using the Request object

Of course, the `Request` and `Response` objects have more useful properties and methods, but this article has shown the most useful in order to get you started.

Further reading

For more information on writing Active Server Object in C++Builder 5, especially combining Active Server Objects with the WebBroker (HTML producing) Technology, I recommend the *C++Builder 5 Developer's Guide*, by SAMS Publishing (ISBN 0672319721). I've written chapters in this book on ActiveX (ActiveForms, Active Server Pages), WebBroker, InternetExpress and MIDAS 3.

Creating a no-VCL Windows application

by Kent Reisdorph

In the article “Working with Windows messages”, I mentioned the fact that writing Windows applications without the benefit of a class library is tedious. This is certainly true. You may, however, need to write a straight C or C++ Windows application using C++Builder. One reason may be that you need the smallest possible executable size. Or possibly you are forced to write straight C/C++ Windows programs by mandate of some higher power (your boss, for example). The good news is that you can easily create a Windows project that doesn’t require the use of the VCL.

The code in **Listing A** is a straight C “Hello World” Windows application. This application compiles to about 50K. Contrast this with a do-nothing VCL application, which compiles to about 350K.

C++Builder 1

Creating a straight C++ Windows application in C++Builder 1 requires a bit more work than for later versions of the compiler. These are the steps you must perform:

1. Create a new application.
2. Remove the main form using the “Remove File From Project” button on the toolbar.
3. Select View | Project Source from the main menu.
4. Remove the include for VCL.H and replace with WINDOWS.H.
5. Remove the USERES line.
6. Choose View | Project Makefile from the main menu. In the project makefile on the ALLLIB line, remove vcl.lib and replace cp32mt.lib with cw32mt.lib. This prevents the linker from pulling in unnecessary code and reduces the final executable by about 70K.
7. Write your code.

The C++Builder 1 IDE doesn’t allow you many compiler options so you will have to accept a C++ application with the overhead that comes with it. Put another way, C++Builder doesn’t allow you to create a Windows GUI application in straight C.

C++Builder 3 and 4

With C++Builder 3 and 4, creating a standard Windows GUI application gets easier. The steps are relatively few:

1. Choose File | New from the main menu. The Object Repository is displayed.
2. Double click the Console Wizard icon.
3. In the Console Wizard dialog box, select Window (GUI) from the Window Type section. Be sure the option to build with the VCL is not checked and click the Finish button.
4. Write your code.

Here again, you don't have a lot of control over project options, but creating the initial project is fairly simple.

C++Builder 5

C++Builder 5 gives you the most flexibility in how you create your Windows GUI application. The steps are the basically the same as for C++Builder 4, but the Console Wizard gives you a couple of extra options. Most notably, you can elect to create an application that uses C only, resulting in a slightly smaller executable. You can also choose whether or not you want to use multi-threading. The C++Builder 5 Console Wizard dialog box is slightly different than in previous versions, so you must uncheck the Console Application option to create a GUI application.

Conclusion

Given the choice, you will likely opt to write Windows programs using the VCL. However, for those times when you need to write a Windows application the old fashioned way, it is good to know that you can do so with C++Builder.

Listing A: *Code for a C "Hello World" Application*

```
#include <windows.h>
#pragma hdrstop

LRESULT FAR PASCAL _export
  WndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(
  HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpszCmd, int nCmdShow)
```

```
{
  static char AppName[] = "HelloWorld";
  HWND        hwnd;
  MSG         msg;
  WNDCLASS    wndclass;
  if (!hPrevInstance)
  {
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = (WNDPROC)WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon =
      LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground =
      (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = 0;
    wndclass.lpszClassName = AppName;

    RegisterClass(&wndclass);
  }

  hwnd = CreateWindow(AppName,
    "Hello World",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL);

  ShowWindow(hwnd, SW_NORMAL);

  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
```

```
  }
  return msg.wParam;
}

LRESULT FAR PASCAL _export
WndProc(HWND hwnd,
  UINT message, WPARAM wParam, LPARAM lParam)
{
  switch(message)
  {
    case WM_PAINT :
    {
      char text[] = "Hello World!!";
      PAINTSTRUCT ps;
      BeginPaint(hwnd, &ps);
      TextOut(ps.hdc, 20, 20, text, 13);
      EndPaint(hwnd, &ps);
      break;
    }
    case WM_DESTROY : {
      PostQuitMessage(0);
      return 0;
    }
    default:
      return DefWindowProc(
        hwnd, message, wParam, lParam);
  }
  return 0;
}
```

More string grids

by Damon Chandler

While string grids are great for displaying large amounts of data, their stock appearance leaves much to be desired. In fact, compared to the grid controls of other applications, such as Microsoft Excel, string grids look downright amateur. And if you've ever tried to force a string grid to blend in with the rest of your user interface, you've likely discovered that there's a lot that a string grid just can't do. For example, you can't change the text or background color on a per-cell basis, you can't italicize a particular cell's font, and you can't even change the color used to highlight the selected cells.

Last month, I introduced the basics of the `TStringGrid` class, how to use many of its key properties, and even some techniques for adding extended functionality. This month, I'll show you how to customize a string grid's appearance, and even how to place a check box in each cell.

Using the OnDrawCell event

One of the techniques that I discussed last month was how to provide support for per-cell text justification. We accomplished this feat by providing a handler for the `OnDrawCell` event, in which we simply rendered the text via the `DrawText()` function.

In fact, the `OnDrawCell` event can be used for much more than manipulating the alignment of each cell's text. You can use this event to further tailor the appearance of the grid on a per-cell basis.

Our previous `OnDrawCell` event handler was relatively simple because we left the `DefaultDrawing` property at its default value of `true`. As such, most of the rendering was handled by the `TCustomGrid` class itself. However, if you do use this event to perform a more advanced rendering operation, you should set the `DefaultDrawing` property to `false`. To understand why, let's examine how the `TStringGrid` and `TCustomGrid` classes handle the drawing of each cell.

A look inside the rendering routine

Like all `TCustomControl` descendants, the `TCustomGrid` class performs its core drawing from within its definition of the virtual `Paint()` method. Within this function, are two nested functions: `DrawLines()` and `DrawCells()`. As you may have guessed, the former is used to render the grid lines, while the latter is used to render the actual cells. It is from within the definition of the `DrawCells()` function that the pure virtual `DrawCell()` method is called. If the `DefaultDrawing` property is `true`, this method is called after the `Font` and `Brush` properties of the grid's `Canvas` have been properly initialized, and after the cell's background is filled via the canvas' `FillRect()` method.

The `TDrawGrid` class defines the `DrawCell()` method to expose the `OnDrawCell` event. In fact,

except for a little shuffling to support the `UseRightToLeftAlignment` property, the `DrawCell()` method simply calls the associated `OnDrawCell` event handler, if one is assigned. The `TStringGrid` class augments the `DrawCell()` method to provide support for textual display. Specifically, the text of each cell is drawn via the canvas' `TextRect()` method. Let's look at the definition of the `DrawCell()` method (translated to C++):

```
void __fastcall TStringGrid::DrawCell(
  int ACol, int ARow, const TRect& ARect,
  TGridDrawState AState)
{
  if (DefaultDrawing) {
    Canvas->TextRect(ARect,
      ARect.Left + 2, ARect.Top + 2,
      Cells[ACol][ARow]);
  }
  TDrawGrid::DrawCell(
    ACol, ARow, ARect, Astate);
}
```

You can see from this implementation that the text for each cell is drawn only if the `DefaultDrawing` property is set to `true`. Also, notice that the `TStringGrid` class calls the `DrawCell()` method of its parent class so that the `OnDrawCell` event handler is still supported. Moreover, this event handler is called *after* the string grid renders the cell's text.
What does all this mean? Simply put, if you provide a handler for the `OnDrawCell` event and render something to the string grid's `Canvas`, all while the `DefaultDrawing` property is set to `true`, you're actually drawing over what the `TCustomGrid` and `TStringGrid` classes have already drawn. While this may be fine if you're only drawing something simple, or only a limited number of cells, it certainly isn't efficient.

Taking control of the rendering process

One of the problems with setting the `DefaultDrawing` property to `false` is that you're forced to manually implement much of the what the `TCustomGrid` class draws by default (i.e., when the `DefaultDrawing` property is `true`). The advantage, however, is that you have complete control over the appearance of each cell. If you're worried about how you're going to draw the fixed cells, don't be. The `Frame3D()` VCL utility function makes this task trivial (it's declared in EXTCTRLS.HPP). All you need to do is base your rendering process on the information contained in the `State` parameter. The following `OnDrawCell` event handler demonstrates how to make a string grid look "normal" when the `DefaultDrawing` property is `false`:

```
#include <cassert>
```

```
void __fastcall
TForm1::StringGrid1DrawCell(
  TObject *Sender, int ACol, int ARow,
  TRect &Rect, TGridDrawState State)
{
  TStringGrid* StringGrid =
    static_cast<TStringGrid*>(Sender);
  assert(StringGrid != NULL);

  TCanvas* SGCanvas =StringGrid->Canvas;
  SGCanvas->Font = StringGrid->Font;

  RECT RText = static_cast<RECT>(Rect);
  const AnsiString text(
    StringGrid->Cells[ACol][ARow]);

  const bool fixed =
    State.Contains(gdFixed);
  const bool focused =
    State.Contains(gdFocused);
  bool selected =
    State.Contains(gdSelected);
  if (!StringGrid1->Options.Contains(
    goDrawFocusSelected)) {
    selected = selected && !focused;
  }
  // if the cell is fixed (headers)
  if (fixed) {
    SGCanvas->Brush->Color =
      StringGrid1->FixedColor;
    SGCanvas->Font->Color = clBtnText;
    SGCanvas->FillRect(Rect);
    Frame3D(SGCanvas, Rect,
      clBtnHighlight, clBtnShadow, 1);
  }
  // if the cell is selected
  else if (selected) {
    SGCanvas->Brush->Color =clHighlight;
    SGCanvas->Font->Color =
      clHighlightText;
    SGCanvas->FillRect(Rect);
```

```
  }
  // if the cell is normal
  else {
    SGCanvas->Brush->Color =
      StringGrid1->Color;
    SGCanvas->Font->Color =
      StringGrid1->Font->Color;
    SGCanvas->FillRect(Rect);
  }
  // if the cell is focused
  if (focused) {
    DrawFocusRect(
      SGCanvas->Handle, &RText);
  }

  // draw the text
  RText.left += 2; RText.top += 2;
  DrawText(SGCanvas->Handle,
    text.c_str(), text.Length(), &RText,
    DT_LEFT |DT_VCENTER |DT_SINGLELINE);
}
```

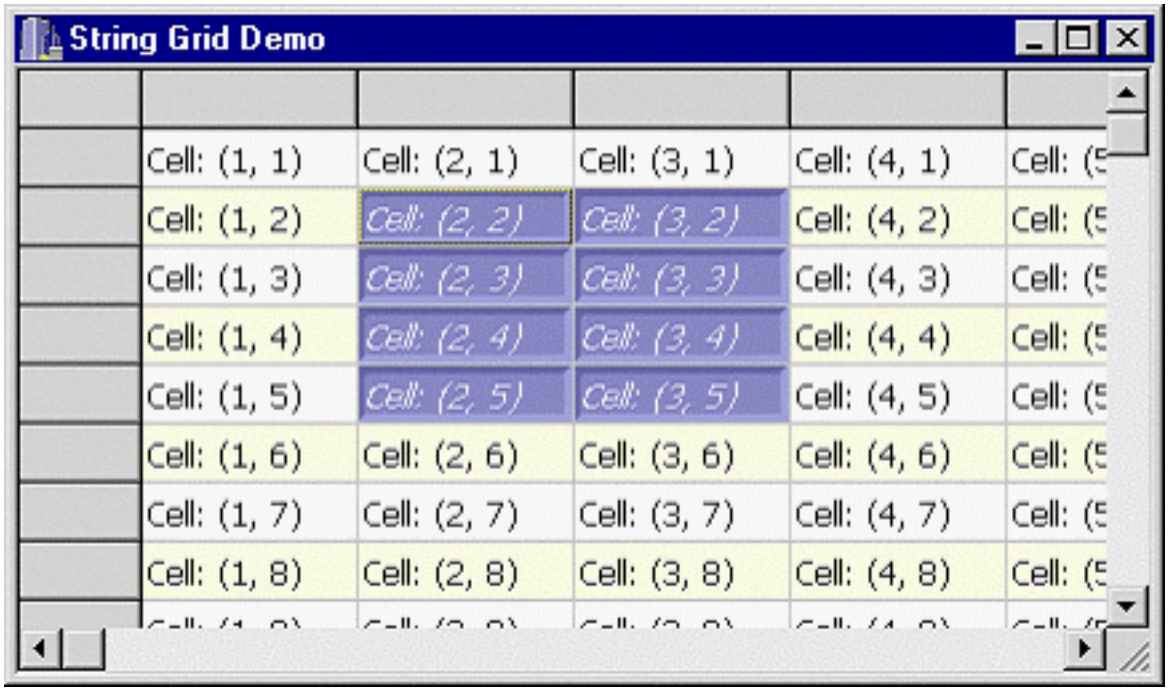
There are, in fact, a few more conditions to test when the `Options` property contains the `goRowSelect` flag, but you get the idea. Still, we've written 40 lines of code just to make the string grid look normal. What's the point? Since the `DefaultDrawing` property is indeed `false`, you can be sure that the `TCustomGrid` and `TStringGrid` classes aren't drawing anything that you're going to draw over. And as an added bonus, you have total control over the appearance of the focus rectangle. The main advantage, however, is efficiency.

Using the `OnDrawCell` event, you can change the way each cell is rendered by implementing your rendering code based on the `ACol`, `ARow`, and `State` parameters. For example, you can use the following approach to change the appearance of the selected cells, and the background color of every other non-selected row. The result is depicted in **Figure A**:

```
void __fastcall TForm1::
  StringGrid1DrawCell(TObject *Sender,
  int ACol, int ARow, TRect &Rect,
  TGridDrawState State)
{
  // other code from before...
```

```
  // if the cell is selected
  else if (selected) {
    unsigned char step = 30;
    unsigned char r = 0x80,
      g = 0x80, b = 0xC0;
    TColor clr_highlight =
      static_cast<TColor>(PALETTERGB(
        r + step, g + step, b + step));
    TColor clr_shadow =
      static_cast<TColor>(PALETTERGB(
        r - step, g - step, b - step));

    SGCanvas->Brush->Color =
      static_cast<TColor>(
        PALETTERGB(r, g, b));
    SGCanvas->Font->Color = clWhite;
    SGCanvas->Font->Style =
      SGCanvas->Font->Style << fsItalic;

    SGCanvas->FillRect(Rect);
    Frame3D(SGCanvas, Rect,
      clr_shadow, clr_highlight, 3);
  }
  // if the cell is normal
  else {
    if (ARow % 2 == 0)
      SGCanvas->Brush->Color = clInfoBk;
    else
      SGCanvas->Brush->Color =
        StringGrid1->Color;

    SGCanvas->Font->Color =
      StringGrid1->Font->Color;
    SGCanvas->FillRect(Rect);
  }
  // other code from before...
}
```

Figure A



A string grid with every other non-fixed row rendered in clInfoBk and the selected cells drawn with a sunken look.

When you use the `OnDrawCell` event in conjunction with the `TStringGrid`'s `Objects` property, you have a solid framework for customizing the appearance of each cell. In fact, this is identically the approach we'll take when we render a check box to each cell; we'll use the `Objects` property to hold the state of each check box.

Adding a check box to each cell

Most grids contain hundreds, even thousands, of cells. If you place, for example, a `TCheckBox` control in each cell, you're looking at consuming a large number of window handles, and thus, system resources. For simple controls such as check boxes or buttons, there is an alternative approach—the `DrawFrameControl()` function. As it name suggests, `DrawFrameControl()` can be used to draw a wide variety of controls, including push buttons, check boxes, popup menu items, and even title bar buttons. Here, we'll use this function to draw a check box to each cell.

Storing the state of each check box

Again, the main advantage of rendering each check box manually is that we avoid the overhead of numerous `TCheckBox` objects. The main disadvantage is that we have to manually implement the check box's functionality. Still, as mentioned previously, we can store the state of each check box in the `Objects` property. To this end, we'll need a few helper functions:

```
bool __fastcall GetCheckState(
  TStringGrid& AGrid, int ACol,int ARow)
{
```

```
  return HIWORD(reinterpret_cast<long>(
    AGrid.Objects[ACol][ARow]));
}

void __fastcall SetCheckState(
  TStringGrid& AGrid,
  int ACol, int ARow, bool AChecked)
{
  long data = reinterpret_cast<long>(
    AGrid.Objects[ACol][ARow]);
  AGrid.Objects[ACol][ARow] =
    reinterpret_cast<TObject*>(
      MAKELONG(LOWORD(data), AChecked));
}
```

These two functions simply provide an easy means by which to set and get the checked state of each cell, to and from the `Objects` property, respectively (we store this state in the high-order word). There's no need to invalidate the cell (i.e., incite a repaint) once the state is changed, since the `TStringGrid` performs this when the `Objects` property itself is changed. In fact, if you ever do need to repaint a particular cell, you can simply assign the `Objects` property of that cell to the value that it already contains:

```
void __fastcall InvalidateCell(
 TStringGrid& AGrid, int ACol, int ARow)
{
  AGrid.Objects[ACol][ARow] =
    AGrid.Objects[ACol][ARow];
}
```

In addition to storing the state of each check box, we'll need a function that can be used to determine if a cell indeed contains a check box, and another function that can be used to add a check box to each cell. We'll store this "has_check_box" flag in the low-order word of the `Objects` property:

```
bool __fastcall GetCheckBox(
 TStringGrid& AGrid, int ACol, int ARow)
{
  return LOWORD(reinterpret_cast<long>(
    AGrid.Objects[ACol][ARow]));
```

```
}

void __fastcall SetCheckBox(
 TStringGrid& AGrid, int ACol, int ARow,
 bool AShow, bool AChecked)
{
  AGrid.Objects[ACol][ARow] =
    reinterpret_cast<TObject*>(
      MAKELONG(AShow, false));
  SetCheckState(
    AGrid, ACol, ARow, AChecked);
}
```

Rendering the check boxes

Putting our functions to good use, we can define the `OnDrawCell` event handler as listed below. Again, to actually render each check box, we'll use the `DrawFrameControl()` function:

```
#include <cassert>
void __fastcall
TForm1::StringGrid1DrawCell(
  TObject *Sender, int ACol, int ARow,
  TRect &Rect, TGridDrawState State)
{
  // other code from before...

  // if this cell contains a checkbox
  if (GetCheckBox(
        *StringGrid, ACol, ARow)) {
    // set the flags for rendering
    // checked/unchecked
    unsigned int state =
      DFCS_BUTTONCHECK;
    if (GetCheckState(
          *StringGrid, ACol, ARow)) {
        state = state | DFCS_CHECKED;
    }

    // size the checkbox
    RECT RCell =static_cast<RECT>(Rect);
```

```
    OffsetRect(&RCell, 2,
      0.5 * (RCell.bottom - RCell.top));
    RCell.right = RCell.left +
      GetSystemMetrics(SM_CXMENUCHECK);
    RCell.bottom = RCell.top +
      GetSystemMetrics(SM_CYMENUCHECK);
    RCell.top -= 0.5 *
      (RCell.bottom - RCell.top) + 2;

    // draw the checkbox
    DrawFrameControl(
      StringGrid1->Canvas->Handle,
      &RCell, DFC_BUTTON, state);

    // move the text over
    RText.left = RCell.right;
  }

  // draw the text
  RText.left += 2; RText.top += 2;
  DrawText(SGCanvas->Handle,
    text.c_str(), text.Length(), &RText,
    DT_LEFT |DT_VCENTER |DT_SINGLELINE);
}
```

Interacting with the check boxes

In addition to displaying each check box, we'll need to change its state when it's clicked or when the spacebar is pressed while the corresponding cell is focused. For the former, we can perform a hit test via a combination of `TStringGrid`'s `CellRect()` method and `the PtInRect()` function. For the latter, we can simply use the string grid's `OnKeyPress` event:

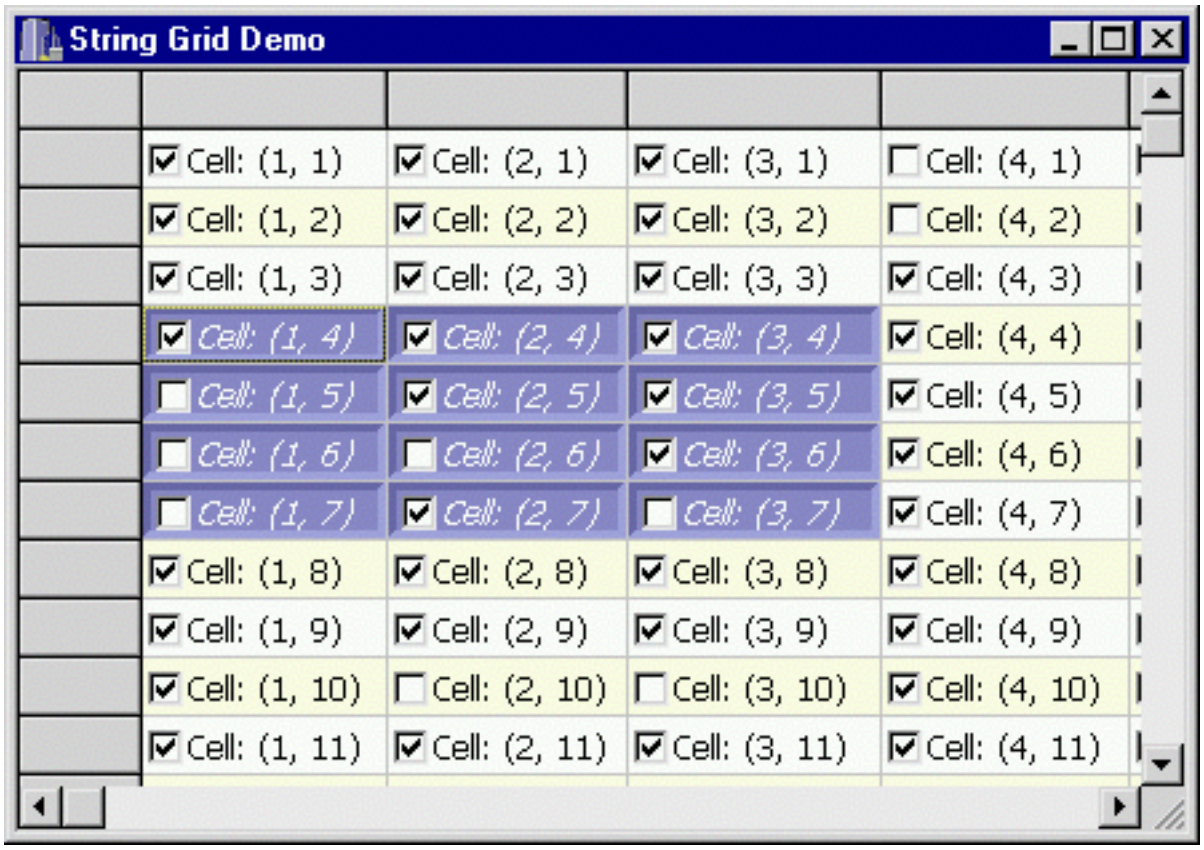
```
bool __fastcall PtInCheckBox(
  TStringGrid& AGrid, int AX, int AY,
  int &ACol, int &ARow)
{
  AGrid.MouseToCell(AX, AY, ACol, ARow);
  RECT RCell = static_cast<RECT>(
    AGrid.CellRect(ACol, ARow));

  OffsetRect(&RCell, 2,
```

```
    0.5 * (RCell.bottom - RCell.top));
  RCell.right = RCell.left +
    GetSystemMetrics(SM_CXMENUCHECK);
  RCell.bottom = RCell.top +
    GetSystemMetrics(SM_CYMENUCHECK);
  RCell.top -= 0.5 *
    (RCell.bottom - RCell.top) + 2;

  return
    PtInRect(&RCell, Point(AX, AY));
}

// OnMouseDown event handler:
void __fastcall TForm1::
  StringGrid1MouseDown(TObject *Sender,
  TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  TStringGrid* StringGrid =
    static_cast<TStringGrid*>(Sender);
  assert(StringGrid != NULL);

  int Col, Row;
  if (PtInCheckBox(
       *StringGrid, X, Y, Col, Row)) {
    if (GetCheckBox(
         *StringGrid, Col, Row)) {
      bool is_checked = GetCheckState(
        *StringGrid, Col, Row);
      SetCheckState(*StringGrid,
        Col, Row, !is_checked);
    }
  }
}

// OnKeyPress event handler:
void __fastcall
TForm1::StringGrid1KeyPress(
  TObject *Sender, char &Key)
{
  TStringGrid* StringGrid =
```

```
    static_cast<TStringGrid*>(Sender);
  assert(StringGrid != NULL);

  if (Key == VK_SPACE) {
    const int col = StringGrid->Col;
    const int row = StringGrid->Row;

    if (GetCheckBox(
          *StringGrid, col, row)) {
      SetCheckState(*StringGrid, col,
        row, !GetCheckState(
          *StringGrid, col, row));
    }
  }
}
```

With the above implementation in place, and through use of our `SetCheckBox()` function to add a check box to each non-fixed cell, we get the result depicted in **Figure B**.

Figure B



A customized string grid with a check box in each non-fixed cell.

Conclusion

While string grids may, at first, look rather bland, you should be convinced at this point that there is much room for enhancement. In fact, once you've implemented a robust `OnDrawCell` event handler, tailoring the appearance of a string grid is fairly straightforward. You can even use the code presented in this article, which can be downloaded from **www.bridgespublishing.com**, as a starting point.

I've demonstrated how to add a check box to each cell, but you're certainly not limited to this type of control. As mentioned, the `DrawFrameControl()` function can be used to render a wide variety of controls. And if you need something more complex in each cell, such as a tree view, you can use the `DrawFrameControl()` function to draw the drop-down button of a combo box. In this way, you can display a small popup window that contains any type of control.

Taking a screen shot

by Kent Reisdorph

Last month in the article, “Simulating keystrokes”, I explained how to take a screen shot using the `keybd_event()` function. That method, however, places the bitmap image in the clipboard. You may prefer to take a screen shot and save that screen shot to a file on disk. This is accomplished fairly easily using a `TCanvas` and a `TBitmap`.

When it comes down to it, the Windows desktop is nothing more than one big bitmap. It may appear that you have several different Windows layered upon one another, but that is purely perception. The basic concept behind taking a screen shot, then, is to copy a portion of the desktop to a `TBitmap` and then save it using the `SaveToFile()` method.

The first step is to create an instance of `TCanvas` that represents the desktop’s device context (DC). The Windows API function `GetDC()` will return a device context handle for any window. Normally you pass the window handle of the window for which you want a device context to `GetDC()`. To get a handle to the desktop DC, though, you simply pass `0` to `GetDC()` as shown here:

```
TCanvas* dtCanvas = new TCanvas;
dtCanvas->Handle = GetDC(0);
```

The next step is to create a `TBitmap` instance and set its width and height as needed. If you are capturing your application’s main form you can use code like this:

```
Graphics::TBitmap*
  bitmap = new Graphics::TBitmap;
bitmap->Width = Width;
bitmap->Height = Height;
```

Once you have a `TCanvas` for the desktop DC and the `TBitmap`, you need to copy the exact portion of the desktop that your application occupies to the `TBitmap`. To begin with you determine the position and size of your application’s window. The VCL makes this easy by providing the `BoundsRect()` method of `TForm`. Next you create a second `TRect` instance representing the location within the `TBitmap` where the screen bits will be copied. The code for these two steps looks like this:

```
TRect src = BoundsRect;
TRect dest = Rect(0, 0, Width, Height);
```

The final step is to copy the pixels contained in the source rectangle from the `TCanvas` to the

`TBitmap` and save the bitmap to disk:

```
bitmap->Canvas->CopyRect(
  dest, dtCanvas, src);
bitmap->SaveToFile("form.bmp");
```

Naturally you need to free the memory allocated for these objects before you are done:

```
delete bitmap;
delete dtCanvas;
```

The resulting BMP file's color depth will be determined by the system's current settings (256 color, 16-bit color, 32-bit color, and so on). If you want to change the color depth of the BMP file then you will need to do a bit more work, but `TBitmap` has the functionality you need.
I should add that this code simply copies a portion of the screen to the `TBitmap`. If part of your application is obscured by another window, then that window will also show in the final BMP image. Also, be aware that there is no way to copy any part of a window that is off the screen or otherwise hidden. Still, this technique is easy to implement and is something you may find useful.

Working with Windows messages

by Kent Reisdorph

In the early days of Windows programming, the programmer had a lot of work to do to create even the simplest of programs. Much of the work entailed handling dozens of Windows messages. This was accomplished by implementing huge `switch` statements in the application's window procedure. Because of this, it was vital for the early Windows programmer to have intimate familiarity with Windows messages. This knowledge included understanding of vast numbers of Windows messages and also a good working knowledge of how the Windows message system worked.

With the advent of Windows framework libraries (such as the VCL), the programmer was relieved from having to know the minute details of the Windows message system. As a VCL programmer you, at least for the most part, don't ever have to worry about your application's window procedure—everything you need to know is passed on to you as VCL events.

This ease of use can be problematic in that new programmers never learn the basics of the Windows messaging system. There are times when you need to handle messages not surfaced as VCL events. There are other times when you need to send a Windows message in order to accomplish some specific task not covered by the VCL methods.

The Windows message system

Before I get into the specifics of working with Windows messages, I need to explain the basics of the Windows message system. The following sections explain windows messages, how Windows generates those messages, and how an application processes messages.

Windows is event driven

The Windows operating system is driven by messages. When an application is hidden by another window and then brought to the front, for example, Windows sends that application a `WM_PAINT` message. The `WM_PAINT` message instructs the application to redraw its main window. Likewise, Windows sends the application a `WM_MOUSEMOVE` message every time the mouse moves over the application. From this you can deduce that there may be thousands of Windows messages flying around at any given moment in time. By the way, I'm using the term "application" loosely here; Windows doesn't actually send the application a message, but rather sends the application's main window a message.

There are three basic types of Windows messages. The first type is used to instruct a window to perform a specific action. The `WM_PAINT` message mentioned earlier falls into this category.

The second message type is used to notify a window that some event has occurred. As an example, consider an edit control on a main form. When the user types in the edit control, Windows sends an `EN_CHANGE` message to the edit control's parent window to notify the parent

that the contents of the edit control have changed. Similarly, when a button is clicked, Windows sends the parent window a `BN_CLICKED` notification message. Many of the events defined for VCL components are generated in response to notification events.
The third type of message is the user-defined message. This is a message that the programmer defines for use within the application.

Message queues

To manage all of these messages, Windows maintains message queues. A message queue is a FIFO (first in, first out) list of messages. Messages are placed in the queue and processed in order they were received. Windows will process the messages in the queue when it has time. Windows maintains a global system message queue and separate message queues for each GUI thread. For simplicity sake, you can consider a thread queue as specific to a particular application.
Not all messages go through the message queue. Some messages are sent directly to the application's window procedure (explained in the next section). Messages can be removed from the message queue using the `GetMessage()` function. Messages in the queue can be examined using the `PeekMessage()` function. `PeekMessage()` allows you to view a message but not remove it, or to remove the message from the queue and process that message manually. Processing the message manually is accomplished by calling `TranslateMessage()` and then `DispatchMessage()`. I won't explain usage of these functions in this article as they are rarely needed in VCL programming.

Window procedures

When a window class is created by an application, its window procedure is registered with the operating system. Each time a message is generated, Windows will call the window procedure associated with that window.
A window procedure is a callback function. The declaration for a window procedure might look like this:

```
LRESULT CALLBACK WndProc(
  HWND hwnd, UINT uMsg,
  WPARAM wParam, LPARAM lParam);
```

The `hwnd` parameter identifies the window for which the message is intended. Several windows can use a single window procedure and the `hwnd` parameter is how the application determines the window to which the message is being sent. The `uMsg` parameter contains the ID of the message being sent. The Windows headers define a macro for each message. The `WM_PAINT` message, for example, has a value of `0x000F` hexadecimal. The `wParam` and `lParam` parameters contain information specific to the type of message being sent. A particular message may utilize neither of these parameters, just the `wParam`, or both `wParam` and the `lParam`. For example, the `WM_PAINT`

message passes the device context handle of the window to be painted in the `wParam`, and nothing (zero) in the `lParam`.
You have no doubt seen an application that, for one reason or another, isn't repainting its main window. This can be due to many reasons, but the bottom line is that Windows is trying to send messages to the window procedure but the window is unable to process those messages.
A straight C/C++ Windows application has to register and maintain a window procedure. Fortunately, the VCL takes care of this for you.

VCL message handling

The VCL automatically registers a window procedure, handles messages, and passes those messages on to you in the form of VCL events. The VCL obviously includes a large number of events. Most of these events are generated in response to a Windows message (some are generated via other means but those events aren't relevant to this article). The `OnKeyDown` event, for example, is fired when Windows sends the application a `WM_KEYDOWN` message.
Not all windows messages are surfaced as VCL events. There are times when you need to handle a message for which there is no corresponding VCL event. For those messages you will have to make use of a message map. I have covered message maps in past articles, so I won't explain them here. See the C++Builder help on `MESSAGE_MAP` for more information on message maps.
VCL classes representing a window all have a `WndProc()` method. You can override this method to intercept or respond to messages before the VCL gets a chance to handle them. This is not typically necessary when writing applications. Component writers, however, may need to override `WndProc()` in order to perform specialized message handling.

Sending messages

In addition to specialize message handling, you may need to send Windows messages from your application. A good example is the `WM_SETREDRAW` message. This message is used to temporarily disable painting of a particular control and later to re-enable painting. For example, let's say you needed to add hundreds or thousands of nodes to a `TTreeView`. Adding a large number of nodes to a tree view can take a long time because the tree view repaints itself as each node is added. By disabling painting of the tree view, items are added much more quickly.
There are three ways to send messages to a VCL application. They are discussed in the next three sections.

The SendMessage() function

The API's `SendMessage()` function is used to send a message directly to a window's window procedure. `SendMessage()` is a synchronous call; the call to `SendMessage()` will not return until the window procedure has processed the message. `SendMessage()` bypasses the application's message queue. Take the `WM_SETREDRAW` message mentioned earlier, for example. The code to disable painting of a `TTreeView`, add items, and re-enable painting might look like

this:

```
SendMessage(
  TreeView->Handle, WM_SETREDRAW, 0, 0);
// Add items to the tree view
SendMessage(
  TreeView->Handle, WM_SETREDRAW, 1, 0);
```

Note that I pass the handle of the tree view control in the first parameter. This is the window (a control in this case) for which the message is destined. In the case of `WM_SETREDRAW`, you pass `0` for the `wParam` to disable painting, and `1` to enable painting again. The `lParam` is not used so I simply pass `0` for this parameter.
Some messages return a value. The purpose of the return value varies widely depending on the message being sent. If you are sending a message and want to examine the return value from that message, you must use `SendMessage()`.

The PostMessage() function

The `PostMessage()` function is used to place a message in a window's message queue. Unlike `SendMessage()`, `PostMessage()` is an asynchronous call. That is, `PostMessage()` returns immediately after posting the message to the message queue. The application continues on its way and the message will eventually make its way to the application's window procedure. `PostMessage()` can be used in situations where you don't need a message's return value. It is also used in situations where you wish to pass a message to windows but need to interject a slight delay before that message is handled.

The VCL's Perform() method

VCL classes that correspond to a window (`TForm` and all windowed components) have a method called `Perform()`. This method sends a message directly to the control's window procedure, bypassing the Windows message system altogether. `Perform()` works much like the API function `SendMessage()`. The earlier code presented for the `WM_SETREDRAW` message could be rewritten to use `Perform()` as follows:

```
TreeView->Perform(WM_SETREDRAW, 0, 0);
// Add items to the tree view
TreeView->Perform(WM_SETREDRAW, 1, 0);
```

Ultimately, calling `Peform()` is the fastest way of sending a message to a VCL windowed component. In most cases it doesn't matter whether you use `Perform()` or `SendMessage()`; the results are practically the same.

TApplication's ProcessMessages() method

No explanation of Windows messages would be complete without at least brief coverage of the VCL `Application` object's `ProcessMessages()` method. This method queries the application's message queue for waiting messages and processes them.
Any time you have a lengthy process in your application (such as a long `for` loop), you should call `Application->ProcessMessages()` during that process. If you do not do this, your application will appear to be locked up. That is, its main form will not repaint, it cannot be moved, maximized, minimized, and so on. Here's an example of `ProcessMessages()`:

```
for (int i=0;i<count;i++) {
  // processing code here
  Application->ProcessMessages();
}
```

Naturally, calling `ProcessMessages()` takes time. For long loops you may want to call `ProcessMessages()` only periodically rather than on each iteration. For example:

```
for (int i=0;i<count;i++) {
  // processing code here
  if ((i % 1000) == 0)
    Application->ProcessMessages();
}
```

In this case `ProcessMessages()` will only be called once for every `1000` iterations of the loop.

Conclusion

The VCL takes much of the pain of message handling out of the programmer's hands. Still, there are times when you need to handle Windows messages yourself or send messages to a particular window. In those cases, it is important that you understand how the Windows message system works.

Building stand-alone EXEs

by Kent Reisdorph

A stand-alone EXE is one that doesn't require external DLLs or packages in order to run. Probably most of you know how to configure a project to build a stand-alone EXE. However, you may not know exactly what happens when you set the project options this way. Further, you may have encountered situations where it was not possible to create an application that is not dependent on external files. Finally, you may have wondered how you can configure C++Builder to create a stand-alone EXE by default. All of these topics will be explained in this article.

Static vs. dynamic linking

A project can be built in one of two ways: using static linking or dynamic linking. When dynamic linking is used (the default for C++Builder projects), the executable uses code from the C++Builder DLLs and runtime packages. This makes the executable very small. However, the small size is misleading because you must ship the required DLLs and runtime packages with your application. A simple do-nothing application, for example, compiles to about 23 KB. However, the application requires the runtime library (RTL) DLL and at least one VCL package. The combined size of the required files is roughly 3 MB at a minimum. In reality, the initial executable size of 23 KB is meaningless. Windows will display an error message if a required DLL or package cannot be located when the application starts. Once that happens, the game is up. Your application cannot run without the required files.

When static linking is used, the linker will pull the code needed to run the application from the RTL and package library files. The code is then linked directly into the application. This results in a larger executable size, but eliminates the requirement of shipping the DLLs and runtime packages. When built using static linking, a do-nothing C++Builder application compiles to about 350 KB. Although the initial size of the executable is larger, the runtime library DLL and runtime packages are not needed. The end result is that the entire distribution is much smaller.

In most cases you will want to use static linking. Use dynamic linking if you have a large application suite that contains many applications and/or DLLs. All of the EXEs and DLLs will share the code in the DLLs and runtime packages. Because of this, the overall size of the entire suite may very well be smaller than when static linking is used.

Creating a stand-alone EXE

Creating a stand-alone EXE requires modifying the project options to use static linking. There are two project options you must modify to accomplish this. Those options are explained in the following sections. After you have changed the project options and rebuild the project, the EXE will not require external files in order to execute.

The dynamic RTL

The first setting you must modify is the project's use of the C++ runtime library DLL. This DLL contains the code for the C++ library. All C++ applications need the RTL code in order to operate. By default, a new C++Builder project is configured to use the RTL DLL. The RTL DLL for C++Builder 5 is named CC3250MT.DLL. For other versions of C++Builder, it has a slightly different name. For C++Builder 4, for example, the DLL is named CP3245MT.DLL.
To eliminate the need for the runtime library DLL, open the Project Options dialog, click on the Linker tab, and uncheck the "Use dynamic RTL" option.

Runtime packages

The second setting you must modify is the setting that controls how packages are used by the application. The default project settings require the use of runtime packages.
All VCL components (including those written by third parties) must reside in a package. Packages are nearly identical to DLLs in the way they are used by an application (the technical differences are not pertinent here). When you build an application using runtime packages, the application must be able to load the runtime package when it starts. When the use of runtime packages is turned off, the code for all components used in the project is linked into the executable.
To turn off the use of runtime packages, select the Packages page on the Project Options dialog and uncheck the "Build with runtime packages" option.

Exceptions to the rule

There are a few situations where you cannot build a stand-alone application. The first is the case of database applications. All C++Builder applications that use the VCL database components require the use of the Borland Database Engine (BDE). The BDE is a set of drivers and DLLs. The BDE cannot be statically linked to your application. You can still build your application using static linking (thereby eliminating the need for the dynamic RTL and runtime packages), but you will still need to ship the BDE with your application.
The second case where you cannot build a stand-alone application is when the application uses ActiveX controls. By their very definition, the code for ActiveX controls cannot be linked into an application. If your application uses ActiveX controls, you must ship the corresponding DLL or OCX file with your application. This applies to COM objects as well as ActiveX controls.
The third case is when an application uses third party libraries that are contained in DLLs. If the third party vendor supplies a static library, then you may use static linking. If only a DLL is supplied, though, you must ship that DLL with your application.
There are probably other situations where you cannot create a stand-alone EXE but these are the most common.

Setting default project options

For whatever reason, a default C++Builder project uses the dynamic RTL and runtime packages. I would venture that 95% of C++Builder users don't build their projects with these setting.

Fortunately, you can easily change the default settings for new projects. There are two ways to go about this.

The first way is to open the Project Options dialog and change the project options as explained earlier. Check the Default check box at the bottom of the dialog before you close the dialog. When you click the OK button, the default project file will be modified. Any new projects will use the project options you just set.

The second way is nearly identical to the first. In this case, though, you begin with no project open in the IDE. When you open the Project Options dialog with no project open, any changes made to the project options will automatically become the new defaults when you click the OK button.

Conclusion

You now know what you must do to build using either static or dynamic linking. Understanding not only *how* to accomplish this, but *why* you might use either option is important to any C++Builder developer.

Finding files

by Mark G. Wiseman

After you have been using C++Builder for a while, the project folders on your hard drive can become really cluttered with unnecessary files. Backup files, debug files, pre-compiled header files, and others can take up a lot of space. To help manage this situation, I'd like to share with you a utility program I've written to clean up my hard drive.
This will be the first of many discussions covering several sections of the program's code in a series of articles. Future articles will look at the user interface code, the code for storing and retrieving settings, and the code for deleting files from the hard drive. I will also talk about design issues when working with C++Builder and planning for program enhancements.

Project file organization

Before we can delete those pesky files, we have to find them. So, how do we do it? Some of it will depend on the way the files are organized, so I'll tell you a little about how I organize my C++Builder project files. (I'm not going to talk about version control software other than to say any project involving more than one developer should always use version control software.)
In the past, I have always kept all of my project files on a separate hard drive. My latest computer has only one very large hard drive, though, so now I keep all of my projects in one folder named PROJECTS. Each project is then put in a separate folder under the PROJECTS folder.
Organizing projects this way makes backups very easy, just backup everything in the PROJECTS folder and its subfolders.
Or do I really mean everything? What about all those files that clutter up folders and take up space? Do I really need to waste time and backup media to backup files that I don't need? What I really need is a utility program that will run before my automated backup each night and delete all those unnecessary files. (You do backup your files every night, don't you?)
There are a few projects that don't reside in the PROJECTS folder. Those reside in subfolders of a folder named ARTICLES. There is, for instance, a subfolder named FINDFILES.
So, what I need is code that will search through different folders and their subfolders to find all the files that match certain name patterns.

Finding the best approach

The Windows API provides a structure, `WIN32_FIND_DATA`, and three functions, `FindFirstFile()`, `FindNextFile()` and `FindClose()` that we will need to search for files. `FindFirstFile()` searches for the first file that matches a specific file name pattern and, if found, will fill in a `WIN32_FIND_DATA` structure. `FindNextFile()` will find the next file that matches the pattern and `FindClose()` will close the handle returned by `FindFirstFile()`. These functions are well documented in the Windows API online help.

We could just use these functions and start writing code, but this isn't very object oriented and there are things we could do to make finding files easier. What about wrapping this structure and these functions in a class or two?
If you read Part 4 of my series of articles, "Hidden Treasures of Sysutils" (Vol. 3, Number 8; August 2000), you know that Borland has written some wrapper code for these functions. You will also know that it is flawed. Borland's code uses an `int` to store file size and this can be a problem with very large files. The Windows API stores file size in two `DWORD` variables. And, as you'll see, with very little effort we can develop some wrapper code that will be much more useful and easier to use.

Wrapping WIN32_FIND_DATA

First let's wrap the Windows API structure `WIN32_FIND_DATA.` Here is the definition for `WIN32_FIND_DATA`:

```
typedef struct _WIN32_FIND_DATA
   {
   DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD    nFileSizeHigh;
   DWORD    nFileSizeLow;
   DWORD    dwReserved0;
   DWORD    dwReserved1;
   TCHAR    cFileName[ MAX_PATH ];
   TCHAR    cAlternateFileName[ 14 ];
   } WIN32_FIND_DATA;
```

This structure contains a `DWORD` that holds the file's attributes (such as read-only or hidden). It also contains three different dates stored in another Windows API structure `FILETIME` and the file size variables, `nFileSizeHigh` and `nFileSizeLow`, that I mentioned earlier. Next, are two reserved `DWORD` variables that we don't need to worry about and, finally, two forms of the file name: the long file name stored in `cFileName` and the old DOS 8.3 file name stored in `cAlternateFileName`.
Since we're programming in C++Builder, there are two obvious things we can do to make this structure easier to use. We can return the three dates in the VCL `TDateTime` format and we can return the two file names in `AnsiString` format. And, while we're at it let's combine those two size variables and return them as a single `__int64`.
Let's take a look at the class I have named `TFFData` (for "Find Files Data") and you'll see why I say that the code will *return* the above values. **Listings A and B** contain the code for `TFFData`.

As you can see from these listings, I did not derive `TFFData` from `WIN32_FIND_DATA`. Instead, I made the `WIN32_FIND_DATA` structure a private member of `TFFData` named `data`. I did this for two reasons.

First, users of this class should not be changing the values in `data`. Making `data` private prevents this. I could have achieved this goal by inheriting privately from `WIN32_FIND_DATA`, but then this code would not be very portable which is the second reason I made `WIN32_FIND_DATA` a member of `TFFData`.

I realize that I'm using the VCL specific `TDateTime` and `AnsiString`; and that this will also limit portability. However, this may not be a problem if I am trying to port my code to the upcoming version of C++Builder for Linux. I have my fingers crossed anyway.

All but one of the functions in `TFFData` is implemented as an inline function for efficiency. The one function I did not implement as an inline is `FileTimeToDateTime()`, which is a private function used to convert a `FILETIME` structure to a `TDateTime`.

You might be wondering why this function and the three functions that use it, `GetCreationDateTime()`, `GetLastAccessDateTime()` and `GetLastWriteDateTime()`, return a `bool` instead of a `TDateTime`. As it turns out, not all versions of Windows keep track of all three dates for all files. As a result, these functions return `true` if the date exists and `false` if it does not. The converted `FILETIME` is then returned to a `TDateTime` that was passed into the functions by reference.

I want to point out a few more things in `TFFData`. The constructor for this class fills the memory occupied by the class with zeroes. This is a safety measure. I could have tested within every member function to be sure that each instance of the class had been initialized with data for a real file, but I thought this was overkill for a simple utility class. Filling the data with zeroes should be good enough.

The functions `GetName()`, `GetAlternateName()`, `GetAttributes()` and `GetSize()` perform exactly what their names imply. Of course, `GetSize()` returns the proper `__int64` and `GetName()` and `GetAlternateName()` return the more useful `AnsiString`.

In addition to `GetAttributes()`, there are three additional functions that deal with file attributes. I added these after I found I was constantly using the `GetAttributes()` function to get the attributes for the file and then testing to see if the file was actually a file or if it was a folder. The functions `IsFile()` and `IsFolder()` now do those tests for me. The function `HasAttribute()` is a little more generic and can be used to test if the file in question has a specific attribute. You can find a list of file attributes in the Windows API online help for the `WIN32_FIND_DATA` structure.

I also included the function `GetRawData()` just in case I need access to the raw `WIN32_FIND_DATA` structure.

Finally, you will notice that I have declared the class `TFindFile` as a `friend` to `TFFData`. I will explain this class in the next section.

Wrapping the Windows API functions

The `TFindFile` class not only wraps the Windows API functions `FindFirstFile()`,

`FindNextFile()` and `FindClose()`, it also adds the capability to search for files in subfolders. The Windows API functions do not have this capability. **Listings C and D** contain the definition and implementation for `TFindFile`.
The constructor for `TFindFile` initializes a few data members and is declared `inline`.
The `Find()` function is the workhorse function in this class. `Find()` takes an `AnsiString` argument that represents a file name pattern. This pattern can contain a file name or pattern and optionally a path to the starting folder to search. If no path is specified, `Find()` will begin its search in the current directory. Let's take a closer look at `Find()`.

The pitfalls of recursion

If you've written code to search through subdirectories or subfolders, you have probably used recursion. Using this approach on a large disk with a lot of nested subfolders, could quickly lead to stack overload because of the number of times a function might have to call itself. To avoid such a situation, I've created a small helper class, local to `TFindFiles`, named `TFFStack`. `TFFStack` uses a `TStringList` to store folder names. I could have used the STL `stack` template, but I found this implementation of `TFFStack` to be a little easier.
The `Find()` function uses the `TFFStack` and two private methods, `FindFiles()` and `FindDirs()` to search through nested folders without using recursion and therefore avoids any stack problems. It also has an added bonus of presenting folders in alphabetical order.

How do we stop?

Let's say we call `Find()` like this:

```
Find("c:\\*.*");
```

On my system, the C drive contains about 45,000 files occupying more than 13 gigabytes of disk space. Obviously, `Find()` may run a very long time on my machine. We need a way to stop it. I've chosen a very simple way.
`TFindFiles` has a private data member, a `bool` named `stop`. When `stop` is set to `true`, `Find`() and its helper functions `FindFiles()` and `FindDirs()` will stop. They do this by checking the value of `stop` at various locations in the code. These functions also call the `ProcessMessages()` method of `TApplication` to allow for the value of `stop` to be changed. The inline `Stop()` function will set the private `stop` variable to `true`.
A more complicated way to allow `Find()` to stop would be to use threads. I'll leave this exercise to you, or possibly a future article.

How do we know Find() has found a file?

`Find()` uses the C++Builder language extension of closures to report when it has found a file. A closure is similar to a callback function, only much better.

You can assign a closure function that you write to the `OnFileFound` property of `TFindFiles`. (The combination of this property and the closure function, make `OnFileFound` an *event* in C++Builder programming lingo.) Every time a file is found, `Find()` calls the `OnFileFound` function, passing it a pointer to the instance of `TFindFile`, the name of the file, the path to the folder containing the file and a copy of the `TFFData`.

Conclusion

The main topic of my next article will be what you do with the information given to you by the `OnFileFound` event. I will also finish discussing the `TFindFile` class, with particular emphasis on the `SearchSubfolders` and `OnSearchFolder` properties.

Listing A: *FFData.h*

```
#ifndef FFDataH
#define FFDataH


class TFFData {
  public:
    TFFData();

    String GetName();
    String GetAlternateName();

    unsigned long GetAttributes();
    bool HasAttribute(DWORD attribute);
    bool IsFolder();
    bool IsFile();

    __int64 GetSize();

    bool GetCreationDateTime(TDateTime &dateTime);
    bool GetLastAccessDateTime(TDateTime &dateTime);
    bool GetLastWriteDateTime(TDateTime &dateTime);

    WIN32_FIND_DATA GetRawData();

  private:
    bool FileTimeToDateTime(
      const FILETIME fileTime,
      TDateTime &dateTime);
```

```
    WIN32_FIND_DATA data;

  friend class TFindFile;
};


// Inline Functions --------------------

inline TFFData::TFFData() {
  ZeroMemory(&data, sizeof(data));
}

inline String TFFData::GetName() {
  return(String(data.cFileName));
}

inline String TFFData::GetAlternateName() {
  return(String(data.cAlternateFileName));
}

inline unsigned long TFFData::GetAttributes() {
  return(data.dwFileAttributes);
}

inline bool TFFData::HasAttribute(
  unsigned long attribute)
{
  return(data.dwFileAttributes & attribute ?
    true : false);
}

inline bool TFFData::IsFolder() {
  return(HasAttribute(FILE_ATTRIBUTE_DIRECTORY));
}

inline bool TFFData::IsFile() {
  return(!IsFolder());
}

inline __int64 TFFData::GetSize() {
  return(data.nFileSizeHigh *
```

```
    MAXDWORD + data.nFileSizeLow);
}

inline bool TFFData::GetCreationDateTime(
  TDateTime &dateTime)
{
  return(FileTimeToDateTime(
    data.ftCreationTime, dateTime));
}

inline bool TFFData::GetLastAccessDateTime(
  TDateTime &dateTime)
{
  return(FileTimeToDateTime(
    data.ftLastAccessTime, dateTime));
}

inline bool TFFData::GetLastWriteDateTime(
  TDateTime &dateTime)
{
  return(FileTimeToDateTime(
    data.ftLastWriteTime, dateTime));
}


#endif   // FFDataH
```

Listing B: *FFData.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "FFData.h"


bool TFFData::FileTimeToDateTime(
  const FILETIME fileTime, TDateTime &dateTime)
{
  if (fileTime.dwLowDateTime == 0 &&
      fileTime.dwHighDateTime == 0)
```

```
    return(false);

  FILETIME localTime;
  if (FileTimeToLocalFileTime(
      &fileTime, &localTime) == 0)
    return(false);

  SYSTEMTIME systemTime;
  if (FileTimeToSystemTime(
      &localTime, &systemTime) == 0)
    return(false);

  dateTime = SystemTimeToDateTime(systemTime);
  return(true);
}


#pragma package(smart_init)
```

Listing C: *FindFile.h*

```
#ifndef FindFileH
#define FindFileH

#include "FFData.h"

class TFFStack;

typedef void __fastcall (__closure *TFileFound)
  (TFindFile *Sender, String fileName,
   String foldername, TFFData data);
typedef void __fastcall (__closure *TSearchFolder)
  (TFindFile *Sender,
   String folderName, bool &skip);

class TFindFile {
  public:
    TFindFile();

    void Find(String filePattern);
    void Stop();
```

```
    __property bool SearchSubfolders = {
      read = searchSubfolders,
      write = searchSubfolders};
    __property TFileFound OnFileFound = {
      read = FOnFileFound, write = FOnFileFound};
    __property TSearchFolder OnSearchFolder = {
      read = FOnSearchFolder,
      write = FOnSearchFolder};

  private:
    void FindFiles(String filePattern);
    void FindDirs(String baseDir, TFFStack &stack);

    TFileFound FOnFileFound;
    TSearchFolder FOnSearchFolder;

    bool stop;
    bool searchSubfolders;
};

inline TFindFile::TFindFile() {
  FOnFileFound = 0;
  FOnSearchFolder = 0;

  stop = true;
  searchSubfolders = false;
}

inline void TFindFile::Stop() {
  stop = true;
}


#endif   // FindFilesH
```

Listing D: *FindFile.cpp*

```
#include <vcl.h>
#pragma hdrstop
```

```
#include "FindFile.h"

class TFFStack {
  public:
    TFFStack();
    ~TFFStack();

    void Push(AnsiString name);
    AnsiString Pop();

    void Empty();

    bool Contains(String name);
    bool IsEmpty();

  private:
    TStringList *list;
};

inline TFFStack::TFFStack() {
  list = new TStringList;
  list->Sorted = true;
  list->Duplicates = dupIgnore;
}

inline TFFStack::~TFFStack() {
  delete list;
}

inline void TFFStack::Push(AnsiString name) {
  list->Add(name);
}

inline AnsiString TFFStack::Pop() {
  AnsiString temp = list->Strings[0];
  list->Delete(0);
  return(temp);
}

inline void TFFStack::Empty() {
  list->Clear();
```

```
}

inline bool TFFStack::Contains(String name) {
  int index;
  return(list->Find(name, index));
}

inline bool TFFStack::IsEmpty() {
  return(list->Count == 0);
}

// --------------------------

void TFindFile::Find(String filePattern)
{
  stop = false;

  String curDir = ExtractFilePath(
    ExpandFileName(filePattern));
  String fileName = ExtractFileName(filePattern);

  bool skip = false;
  if (FOnSearchFolder)
    FOnSearchFolder(curDir, skip);
  if (FOnFileFound && skip == false)
    FindFiles(curDir + fileName);

  if (searchSubfolders) {
    TFFStack dirStack;

    while (stop == false) {
      Application->ProcessMessages();

      FindDirs(curDir, dirStack);
      if (dirStack.IsEmpty())
        break;

      curDir = dirStack.Pop();

      skip = false;
      if (FOnSearchFolder)
        FOnSearchFolder(curDir, skip);
```

```
      if (skip)
        continue;

      if (FOnFileFound)
        FindFiles(curDir + fileName);
    }
  }

  stop = true;
}

void TFindFile::FindDirs(
  String baseDir, TFFStack &stack)
{
  TFFData ffData;

  String dirPattern = baseDir + "*.*";

  HANDLE handle = FindFirstFile(
    dirPattern.c_str(), &ffData.data);
  if (handle != INVALID_HANDLE_VALUE) {
    bool found = true;

    while (found == true && stop == false) {
      Application->ProcessMessages();

      if (ffData.IsFolder() &&
          ffData.GetName() != "." &&
          ffData.GetName() != "..")
        stack.Push(baseDir + ffData.GetName() + "\\");

      found = (FindNextFile(handle, &ffData.data) != 0);
    }

    FindClose(handle);
  }
}

void TFindFile::FindFiles(String filePattern)
{
  TFFData ffData;
```

```
  HANDLE handle = FindFirstFile(
    filePattern.c_str(), &ffData.data);
  if (handle != INVALID_HANDLE_VALUE) {
    bool found = true;

    while (found == true && stop == false) {
      Application->ProcessMessages();

      if (FOnFileFound &&
          ffData.GetName() != "." &&
          ffData.GetName() != "..")
        FOnFileFound(ffData.GetName(),
          ExtractFilePath(filePattern),
          ffData, stop);
      found = (FindNextFile(handle, &ffData.data) != 0);
    }

    FindClose(handle);
  }
}


#pragma package(smart_init)
```

Simulating keystrokes

by Kent Reisdorph

Sometimes an application needs to simulate keystrokes that are normally performed by the user. There could be a variety of reasons to simulate keystrokes. For example, you may need to dismiss a menu the user has open or take a screenshot of the application for technical support reasons. Or maybe you are creating an automated tutorial that guides the user through a particular sequence. These actions can be accomplished by simulating keystrokes.

The keybd_event() function

There are several ways to go about simulating keystrokes, but the simplest and most effective is to use the Win32 API `keybd_event()` function. This function is declared as follows:

```
VOID keybd_event(BYTE bVk, BYTE bScan,
  DWORD dwFlags, DWORD dwExtraInfo);
```

The first parameter, `bVk`, is used to specify the virtual key code of the key you wish to simulate. (I will explain this parameter in more detail in the next section). The `bScan` parameter is used to pass the hardware scan code for the key. You should pass `0` for this parameter. The `dwFlags` parameter is used to specify whether the key is being pressed or whether it is being released. Pass `0` for this parameter if you are simulating a key press, or `KEYEVENTF_KEYUP` if you are simulating the release of a key. The `dwExtraInfo` parameter is used to pass any application-specific data associated with the key press. You won't typically use this parameter in a VCL application so you can simply pass `0`.

Specifying the key code

As I have said, the first parameter of `keybd_event()` is used to pass the value of the key for which you are simulating a key press. You can pass virtual key codes to `keybd_event()` in one of two ways: using one of Windows' virtual key codes, or the ASCII value corresponding to an alphanumeric key. Let's say, for example, that you wanted to simulate the press of the Enter key. In that case you would use this code:

```
keybd_event(VK_RETURN, 0, 0, 0);
keybd_event(
  VK_RETURN, 0, KEYEVENTF_KEYUP, 0);
```

In this example, the virtual key constant passed is `VK_RETURN`, the value that corresponds to the Enter key. The Windows virtual key constants can be found by looking at the topic, "Virtual-Key

Codes” in the Win32 help file. Be aware, though, that this help topic is not entirely accurate. For example, it states that the virtual key codes for alpha keys are `VK_A` through `VK_Z`. In reality, these virtual key codes no longer exist in the Windows headers. For the ultimate source of information on virtual key codes, see the `WINUSER.H` header that ships with C++Builder.
Note that the preceding code calls `keybd_event()` once for the key press, and once for the key release. If you don’t call `keybd_event()` for the key release, your application may exhibit strange behavior. For example, if you call `keybd_event()` to simulate an Alt key press and do not call `keybd_event()` a second time to simulate release of the Alt key, the application will behave as if the Alt key were stuck down.
To simulate the press of an alpha key, pass the ASCII value of the upper case letter for that key:

```
keybd_event('M', 0, 0, 0);
keybd_event(
  'M', 0, KEYEVENTF_KEYUP, 0);
```

You must pass the letter in upper case or the function will fail. Even though the ASCII value for an upper case m is passed, the actual key code sent to Windows will depend on what other keys are down. If the Shift key is down when this code executes, an upper case m will be sent to Windows. An upper case m will also be sent to Windows if Caps Lock is on when this code executes.

Uses of keybd_event()

You might need to simulate a keystroke in any number of situations. The specific reasons for simulating a keystroke are application-specific. However, I will illustrate the use of `keybd_event()` with a few examples.

Simulating menu selections

One case where you may want to simulate key presses is to emulate programmatically what the user may do manually. For example, assume that you have an application with a standard File menu. The keystrokes for choosing File|New are Alt-F-N (assuming the default accelerator keys). The code to simulate this key combination is:

```
// simulate Alt key press
keybd_event(VK_MENU, 0, 0, 0);
// simulate F key press
keybd_event('F', 0, 0, 0);
// release F key
keybd_event('F', 0, KEYEVENTF_KEYUP, 0);
// release Alt key
```

```
keybd_event(
  VK_MENU, 0, KEYEVENTF_KEYUP, 0);
// press and release N key
keybd_event('N', 0, 0, 0);
keybd_event('N', 0, KEYEVENTF_KEYUP, 0);
```

You may have noticed from this code that the virtual key code for the Alt key is `VK_MENU`. There is no `VK_ALT` virtual key code as you might expect. Note that each key press is followed by code that simulates a key release.

Dismissing a menu

Users are known for doing things with your application that you don't expect. For example, a user may drop down a menu and go for a cup of coffee. If your application needs to do processing on a periodic basis, the fact that the menu is dropped down may interfere with that processing. You can easily dismiss a menu that is showing by sending an Esc key press to the application. The code for doing this is simple:

```
keybd_event(VK_ESCAPE, 0, 0, 0);
keybd_event(
  VK_ESCAPE, 0, KEYEVENTF_KEYUP, 0);
```

You may want to follow this code with a second Esc key press in case the top-level menu item is still selected.

Grabbing a screen shot

You can easily take a screen shot of the desktop by simulating a Print Screen key press, or a screen shot of the active window by simulating an Alt-Print Screen key press. Here's how the code looks for getting a screen shot of the active window:

```
keybd_event(VK_MENU, 0, 0, 0);
keybd_event(VK_SNAPSHOT, 0, 0, 0);
keybd_event(
  VK_SNAPSHOT, 0, KEYEVENTF_KEYUP, 0);
keybd_event(
  VK_MENU, 0, KEYEVENTF_KEYUP, 0);
```

Note that this code first sends the Alt keystroke to the application and then the Print Screen keystroke (the `VK_SNAPSHOT` constant is used for the Print Screen key). The documentation for `keybd_event()` explains another way to accomplish this. It says that you can pass `0` for the

second parameter to get a screen shot of the entire desktop, or `1` to get a screen shot of the active window.
Once you have the screen shot image in the clipboard you can easily save it to a file on disk using this code:

```
Sleep(1000);
Graphics::TBitmap* bm =
  new Graphics::TBitmap;
bm->Assign(Clipboard());
bm->SaveToFile("d:\\test.bmp");
delete bm;
```

Note that I have introduced a slight delay to ensure that the entire image is in the clipboard prior to saving the image. This is a low-tech way to go about this. You will probably want to implement a more robust method in your own applications.

Demo animations

Another possible use of `keybd_event()` is in creating demo animations for your users. You can move the mouse cursor using a combination of `GetCursorPos()` and `SetCursorPos()`. You can then use `keybd_event()` to simulate button or menu clicks. I won't go into further detail on this subject, but the example program for this article shows a simple demo animation effect.

Conclusion

Simulating keystrokes can be valuable for certain types of applications. The `keybd_event()` function provides a simple and effective way of simulating keystrokes. In addition to `keybd_event()`, Windows also has a `mouse_event()` function that can be used to simulate mouse events.
The example application for this article shows several uses of `keybd_event()`. First, it automatically dismisses the main menu if it is left open for three seconds. Second, allows the user to take a screen shot of the application's main window. Third, it clicks a button on the form called "Test" using that button's accelerator key. Finally, it provides a demo animation that "clicks" the File | New menu item. Download and examine the example program to see how these effects are implemented.

Working with string grids

by Damon Chandler

String grids are ideal for displaying large amounts of data in a compact, tabular format. Conceptually, a string grid is little more than a graphical representation of a two-dimensional array of strings. Despite this conceptual simplicity, working with the `TStringGrid` class can sometimes be intimidating.

In this article, I'll get you up to speed on using the `TStringGrid` class effectively. I'll start with a brief overview of where the class lies in the VCL hierarchy, and then discuss a few of its most commonly used properties. Next, I'll explore some of the more practical aspects of the `TStringGrid` class: I'll demonstrate how to render the text of each cell in a customized format, how to insert and remove a column or row, and how to save and load the cells to and from a file.

The ancestor classes

The `TStringGrid` class is a direct descendant of the `TDrawGrid` class, which is itself a descendant of the `TCustomGrid` class—the base class for all standard grid controls. The `TCustomGrid` class is a `TCustomControl` descendant that provides the basic functionality of a stock grid-type control.

The `TDrawGrid` class adds little to its parent class, introducing only the `CellRect()` and `MouseToCell()` methods. Also introduced are several events, whose handlers are simply invoked in response to corresponding inherited methods. For example, the `OnDrawCell` event handler is called from within the `DrawCell()` method. As the `TDrawGrid` class acquires nearly all of its functionality from the `TCustomGrid` class, it's no surprise that the implementation of the former contains less than one hundred lines of code. The implementation of its parent class, on the other hand, contains nearly thirty times that amount. In short, a `TDrawGrid` object is essentially an "exposed" version of a `TCustomGrid` object.

The `TStringGrid` class extends the `TDrawGrid` class by providing support for per-cell `AnsiString` and application-defined data association. The strings are exposed via the `Cells`, `Cols`, and `Rows` properties, while the data is coupled via the `Objects` property.

The Cells, Cols, and Rows properties

As mentioned, each cell in a string grid can be assigned an `AnsiString` value. This functionality is provided through the `Cells`, `Cols`, and `Rows` properties.

The `Cells` property affords the most common and intuitive means of manipulating the contents of a particular cell. This two-dimensional array-type property is declared as follows:

```
__property AnsiString Cells
  [int ACol][int ARow] = {
```

```
    read=GetCells, write=SetCells};
```

The `ACol` parameter indicates the zero-based index of the column to which the target cell belongs, while the `ARow` parameter indicates the zero-based index of the target cell's row. For example, to assign a value to the first (top-leftmost), non-fixed cell, you use the `Cells` property as such:

```
const int target_col =
  StringGrid1->FixedCols;
const int target_row =
  StringGrid1->FixedRows;
StringGrid1->Cells[target_col]
  [target_row] = "New String";
```

The `FixedCols` and `FixedRows` properties indicate the number of fixed columns and rows, respectively.
To extract the value of a particular cell, you read the `Cells` property in a similar fashion. For example, the following code displays the contents of the last (bottom-rightmost) cell:

```
const int target_col =
  StringGrid1->ColCount - 1;
const int target_row =
  StringGrid1->RowCount - 1;
ShowMessage(StringGrid1->Cells
  [target_col][target_row]);
```

The `ColCount` and `RowCount` properties correspond to the number of columns and rows (including the fixed cells), respectively.

The `Cols` and `Rows` properties provide access to an entire column or row of strings, respectively. Each of these properties is published as an array of `TStrings*`:

```
__property Classes::TStrings*
  Cols[int Index] = {
    read=GetCols, write=SetCols};
__property Classes::TStrings*
  Rows[int Index] = {
    read=GetRows, write=SetRows};
```

To access the contents of a particular cell from the return value of the `Cols` or `Rows` property, you use the `Strings` property, specifying the target row or column. For example, to access the

contents of fourth cell in third column using the `Cols` property:

```
const int target_col = 2;
const int target_row = 3;
ShowMessage(StringGrid1->Cols
  [target_col]->Strings[target_row]);
```

And, to access the contents of the same cell using the `Rows` property:

```
const int target_col = 2;
const int target_row = 3;
ShowMessage(StringGrid1->Rows
  [target_row]->Strings[target_col]);
```

The number of entries contained in the `TStrings` object returned via the `Cols(Rows)` property is identically `RowCount(ColCount)`. So, in our first example using the `Cols` property, `target_row` can range from zero to `RowCount`. Similarly, for our second example using the `Rows` property, `target_col` can range from zero to `ColCount`.

The Objects property

The `Objects` property is the `TObject*` equivalent of the `Cells` property. That is, instead of representing a two-dimensional array of `AnsiStrings`, the `Objects` property stores a pointer to any application-defined data. In this way, you can associate extra information with each cell. The property is declared as follows:

```
__property System::TObject*
  Objects[int ACol][int ARow] = {
    read=GetObjects, write=SetObjects};
```

You access the `Objects` property in the same way as you would the `Cells` property. For example, to associate an integral value with the first (top-leftmost), non-fixed cell, you use the `Objects` property as such:

```
const int target_col =
  StringGrid1->FixedCols;
const int target_row =
  StringGrid1->FixedRows;
const int associated_data = 1001;
StringGrid1->Objects
```

```
  [target_col][target_row] =
    reinterpret_cast
      <TObject*>(associated_data);
```

Also, note that the `TStringGrid` class does not automatically free the memory associated with any data assigned to its `Objects` property. Therefore, if you indeed assign the `Objects` property a pointer to an instantiated object, be sure to handle the memory de-allocation appropriately.

Controlling the text alignment

Not only does the `TStringGrid` class handle the storage of each cell's string and data, it also provides automatic rendering of the text. This task is accomplished from within the redefinition of the inherited `DrawCell()` method, where the `TextRect()` method is employed. However, you may want to have the contents of each cell formatted in a personalized fashion. For example, if your string grid contains a column that holds monetary values, you might prefer the text of these cells to be aligned to the right.

While the `TStringGrid` class does not provide a direct means of controlling the text alignment of its cells, it does inherit the `OnDrawCell` event from the `TDrawGrid` class. By providing a handler for this event, you can render the cell's contents justified to the left, center, or right side of the cell.

The OnDrawCell event

As mentioned, the `OnDrawCell` event is inherited from the `TDrawGrid` class where it is declared as follows:

```
__property TDrawCellEvent OnDrawCell =
  {read=FOnDrawCell, write=FOnDrawCell};
```

`TDrawCellEvent` is a custom event type designed specifically for rendering grid-type controls:

```
typedef void __fastcall
  (__closure *TDrawCellEvent)
  (System::TObject* Sender, long ACol,
   long ARow, Windows::TRect Rect,
   TGridDrawState State);
```

The `Sender` parameter is a pointer to the string grid itself. The `ACol` and `ARow` parameters indicate the current column and row of the cell that requires rendering, respectively, while the `Rect` parameter identifies the bounding rectangle of this cell (in client coordinates). The `State` parameter is a set that indicates the state of the current cell. You decode the members of the

State parameter to determine how the cell should be rendered. This parameter can contain a combination of the following self-explanatory values: gdSelected, gdFocused, and gdFixed.

Rendering the text

In addition to the OnDrawCell event, the TStringGrid class inherits the DefaultDrawing property, which is first introduced in the TCustomGrid class. When this property is set to true (the default), the TCustomGrid class handles all of the "default" rendering. That is, each cell is automatically rendered according to the members of the State parameter. In this case, within a handler for the OnDrawCell event, you are only required to render the formatted text. For example, to render the contents of each cell in a right-justified fashion:

```
#include <cassert>
void __fastcall
TForm1::StringGrid1DrawCell(
  TObject *Sender, int ACol, int ARow,
  TRect &Rect, TGridDrawState State)
{
  TStringGrid* StringGrid =
    static_cast<TStringGrid*>(Sender);
  assert(StringGrid != NULL);
  StringGrid->Canvas->FillRect(Rect);

  AnsiString text(
    StringGrid->Cells[ACol][ARow]);
  RECT RText = static_cast<RECT>(Rect);
  InflateRect(&RText, -3, -3);

  DrawText(StringGrid->Canvas->Handle,
    text.c_str(), text.Length(), &RText,
    DT_RIGHT | DT_SINGLELINE |
      DT_VCENTER
  );
}
```

The DrawText() API function is ideal for rendering justified text. As in the preceding example, the last parameter to this function can be specified as a combination of several text-formatting values. For left-justification, you simply replace the DT_RIGHT flag with DT_LEFT. Likewise, for center justification, use the DT_CENTER flag.
In fact, you can use this technique to adjust the color or style of the text as well. In such a case,

you simply manipulate the `Font` property of the string grid's `Canvas`.

Inserting or removing a row or column

While the `ColCount` and `RowCount` properties can be used to adjust the number of columns and rows, respectively, there is no built-in technique to insert a column or row at a specific location. For example, if you need to append a row to the end of the string grid, you simply increment the `RowCount` property by one. However, if you need to insert a row at, for instance, the *middle* of the grid, the `RowCount` property alone will not help.

As you may have guessed, inserting a column or row actually requires two steps: (1) Append a column or row to the end of the grid, then (2) shift the contents of each trailing (i.e., located after the insertion location) column or row. For example, if your grid contains five rows and you need to insert a new row between the first and second rows, you must first add a new row to the end of the grid, then shift the contents of the second through fifth rows down by one. This process can be accomplished via the following approach:

```
void __fastcall InsertRow(
  TStringGrid* StringGrid,int AfterIndex)
{
  SendMessage(StringGrid->Handle,
    WM_SETREDRAW, false, 0);
  try {
    const int row_count =
      StringGrid->RowCount;

    // (1) append a new row to the end
    StringGrid->RowCount = row_count +1;

    // (2) shift the contents
    // of the trailing rows
    for (int row = row_count;
         row > AfterIndex + 1; --row) {
      StringGrid->Rows[row] =
        StringGrid->Rows[row - 1];
    }
    StringGrid->Rows[
      AfterIndex + 1]->Clear();
  }
  catch (...) {
    SendMessage(StringGrid->Handle,
      WM_SETREDRAW, true, 0);
```

```
  }
  SendMessage(StringGrid->Handle,
    WM_SETREDRAW, true, 0);

  // update (repaint) the shifted rows
  RECT R =
    StringGrid->CellRect(0, AfterIndex);
  InflateRect(&R, StringGrid->Width,
    StringGrid->Height);
  InvalidateRect(
    StringGrid->Handle, &R, false);
}
```

The `WM_SETREDRAW` message is used to temporarily prevent the string grid from repainting itself during the manipulation of its contents. After the new row has been inserted, the `InvalidateRect()` API function is used to force the string grid to repaint the shifted cells. A similar approach can be employed to insert a new column.
Removing a particular column or row also requires a two-step process, but in the reverse order. This time, you first shift the contents of the trailing columns or rows, then remove the last column or row. For example:

```
void __fastcall RemoveCol(
  TStringGrid* StringGrid, int Index)
{
  SendMessage(StringGrid->Handle,
    WM_SETREDRAW, false, 0);
  try {
    const int col_count =
      StringGrid->ColCount;

    // (1) shift the contents of
    // the trailing columns
    for (int col = Index;
         col < col_count - 1; ++col) {
      StringGrid->Cols[col] =
        StringGrid->Cols[col + 1];
    }

    // (2) remove the last column
    StringGrid->ColCount = col_count -1;
  }
```

```
  catch (...) {
    SendMessage(StringGrid->Handle,
      WM_SETREDRAW, true, 0);
  }
  SendMessage(StringGrid->Handle,
    WM_SETREDRAW, true, 0);

  // update (repaint) the shifted cols
  RECT R =
    StringGrid->CellRect(0, Index);
  InflateRect(&R, StringGrid->Width,
    StringGrid->Height);
  InvalidateRect(StringGrid->Handle,
    &R, false);
}
```

As it turns out, the `TCustomGrid` class contains the protected virtual `DeleteColumn()` and `DeleteRow()` methods. These methods can be used to remove a particular column or row, respectively. Therefore, as an alternative to the manual approach, you could create your own `TStringGrid` descendant class that provides public access to these protected methods. For example:

```
class TMyStringGrid : public TStringGrid
{
public:
  __fastcall TMyStringGrid(
    TComponent* AOwner) :
      TStringGrid(AOwner) {}

  void __fastcall RemoveCol(int AIndex) {
     TStringGrid::DeleteColumn(AIndex);
  }
  void __fastcall RemoveRow(int AIndex) {
     TStringGrid::DeleteRow(AIndex);
  }
};
```

Persistent cells

Rarely is a string grid used for the display of interim data. In most cases, you'll want to provide your consumers with the ability to save and move their work to and from a file. While the

TStringGrid class does not provide an automated means of accomplishing this task, it is not difficult to implement manually. In fact, since the Cols and Rows properties are both published as type TStrings*, you can delegate most of the work to this latter class.

Writing the cells to a file

The TStrings class provides the SaveToFile() method that can be used to write its contained strings to a file. To use this method with a string grid, first create a temporary TStringList object that will serve as a container of all the cells. Next, iterate over the Cols (or Rows) property, adding the contents of each column (or row) to the temporary container via the AddStrings() method. Finally, call the SaveToFile() method to write the contents of the temporary container to a file. For example:

```
#include <memory>
void __fastcall SaveCells(
  TStringGrid* StringGrid,
  const AnsiString& FileName)
{
  std::auto_ptr<TStrings> SaveStrings(
    new TStringList());
  const int col_count =
    StringGrid->ColCount;
  for (int index = 0;
       index < col_count; ++index) {
    SaveStrings->AddStrings(
      StringGrid->Cols[index]);
  }
  SaveStrings->SaveToFile(FileName);
}
```

Loading the cells from a file

To load a string grid with text from a file created via the above SaveCells() function, you perform a similar operation. Again, make use of the TStrings class; this time, the LoadFromFile() method. First, create a temporary TStringList object that will hold the contents of the file. Next, call the LoadFromFile() method to fill this container, then iterate over the TStringGrid's Cells property, assigning each cell its corresponding entry from the container. For example:

```
#include <memory>
void __fastcall LoadCells(
  TStringGrid* StringGrid,
  const AnsiString& FileName)
{
  std::auto_ptr<TStrings> LoadStrings(
    new TStringList());
  LoadStrings->LoadFromFile(FileName);

  int index = 0;
  const int col_count =
    StringGrid->ColCount;
  const int row_count =
    StringGrid->RowCount;
  for (int col=0;col<col_count;++col)
  {
     for (int row=0;row<row_count;++row)
     {
        StringGrid->Cells[col][row] =
          LoadStrings->Strings[index++];
     }
  }
}
```

As an exercise, modify the above `LoadCells()` function to make use of the `WM_SETREDRAW` message in order to prevent repainting during loading.

Conclusion

String grids provide a great opportunity for representing data in a tabular format. Perhaps this demonstration of these techniques for accomplishing the most fundamental string grid-related tasks has convinced you that, even though the class lacks certain basic features, it is flexible enough to allow for easy enhancement. From there, the possibilities are practically endless.

Adding controls to a DBGrid

by Steve Ketcham

The standard `TDBGrid` does its job, but sometimes the default behavior is not exactly what you want. Boolean fields, for example, show either “True” or “False” depending on the value of the data field. In this case, it is much more visually appealing to be able to you use a check box rather than the standard display. Put another way, it would be better if the grid could show a `TDBCheckBox` rather than “True” or “False”. Or maybe you want to be able to show a customized combo box or other control on a grid. You can add these controls to a `TDBGrid`. You only need to handle three events and write three worker functions.

Defining the process

When adding controls to `DBGrid` there are a couple of questions to be answered and several requirements:

1. If the control is for visual effect, as in the case of a `TDBCheckBox,` then the grid should draw the appropriate image in the correct cell after the control is no longer visible. The drawing event should also allow for the user to change column widths. If the control is not being used for visual reasons these are not required .
2. If there is more than one added control, there must be a way to select the correct control.
3. When the grid’s `SelectedField` is the same as the corresponding control’s `DataField,` the control should receive focus and become visible. The opposite is also true. When the grid’s `SelectedField` is not the same as the control’s `DataField,` the control should lose focus and become invisible. While some controls are easy to add, others need to be doctored so they look right. For example, a `TDBCheckBox` does not have `taCenterJustify` value in its `Alignment` property, only `taLeftJustify` and `taRightJustify`. This makes it is hard to center in a grid cell.
4. Is this going to be a component, a new class or just cut and paste code for occasional use? This really is not “everyday use” coding: how often would you use it? Some people might use it a lot, most probably would not. If you intend to use this code often, you should consider writing a component to do the work.

Rather than writing a component, this is going to be cut and paste code since the example described is not very detailed. The idea is to show how to integrate controls. However, this code could easily be made into a class or component, thereby making it reusable.
Now that I have outlined what needs to be done, I’ll explain the process of adding a

TDBCheckBox to a grid.

Step one: add controls

Add a TDBGrid to a form and connect the grid to the appropriate TDataSource and TTable. Make sure the TTable is connected to a database that contains a Boolean field (the VENDORS.DB table that ships with C++Builder is a good choice if you don't have a table of your own to use). Next place a TDBCheckbox on the form and set its Visible property to false.
Set the Name property of the TDBCheckbox to DBCheckBoxGrid. Change the color of DBCheckBoxGrid to clWindow so it blends in with the grid and remove any captions. If your grid is not the default color, change the control appropriately. One last thing: make sure the TDBCheckBox is connected to the same TDatasource as the grid and to the correct field.

Step two: create a function to draw the checked or unchecked state

This is the first worker function. A TDBCheckBox is normally connected to a Boolean field. The TDbCheckBox you placed on the form is going to be added to the DBGrid for visual effect. This means you need at least two images for drawing: one for the checked state, and one for the unchecked state. The easiest way to accomplish this is to use the Windows API DrawFrameControl() function to draw directly on the grid's canvas(that's how Windows does it!). Declare a function in the private section of your main form with this declaration:

```
void SetCheck(
const TRect &Rect, bool IsChecked);
```

This function needs to know two things: whether it should draw the mark as checked or unchecked, and where to draw the mark. The OnDrawColumnCell event handler for the grid (which we haven't written yet) will call this function (see **Listing A**).
Inside the SetCheck() function is another function called NewDimensions(). This function is declared as follows:

```
int NewDimensions(
int Rect1, int Rect2, int Dimension);
```

This is the second worker function. NewDimensions() will provide the value needed to center DBCheckBoxGrid if it has focus or to center the check mark correctly if required. NewDimensions() also takes care of automatically centering the check mark (see **Listing F**).

Step three: find the correct control

This is a little out of order because I still have not finished addressing the grid's cell drawing code but I will do that shortly. Create another function in the main form with this declaration:

```
TWinControl* FindGridControl(
AnsiString FieldName, bool IsVisible);
```

This function takes a field name and returns a `TWinControl` pointer to the correct control. It also sets the `Visible` property of the control. Inside the function the `Parent` property of the control is changed to match the grid's `Parent` property. This gives the control and the grid a common reference for positioning the control. This is explained a little later.
The `FindGridControl()` method performs two different functions. It determines whether the grid needs to paint check marks for a particular column, and it allows the grid to give focus to the control and hide the control when it is not needed. **Listing D** shows this function. This code is only necessary if you use more than one control on a grid or if you want to reuse `DBCheckBoxGrid` for another Boolean field. Generally speaking, this will be the only function that will ever need to be changed after the original code is written.
Note: There are two extra dummy controls added to the function for demonstration purposes. In your own code these will need to be changed or removed.

Step four: give the control focus or the draw cell

Add an `OnDrawColumnCell` event handler to the grid (do not use `OnDrawDataCell`; see **Listing C**). Here is where the grid starts doing its work and it is probably the most confusing part. This function has four parameters and we will use three of them. The first parameter of note is called `State`. The grid is either going to give the appropriate control focus or is going to redraw the cell depending on the value of the `State` parameter. The `OnDrawColumnCell` event handler performs two functions and the `FindGridControl()` function is used in both cases. The following sections will explain each of these functions.
Handling control focus

If the `State` parameter is set to `gdFocused` or `gdSelected` and `FindGridControl()` does not return `NULL`, the database control will have its `Visible` property set to `true` and receive focus. If the control found by `FindGridControl()` is not `DBCheckBoxGrid` then the grid will resize the control to fit the current cell by using the values in the `Rect` parameter.
A `TDBCheckBox` must be centered manually so it's width and height will stay the same and the grid's canvas will be erased in the immediate area behind `DBCheckBoxGrid`. The `TDBCheckBox` is centered by using values returned by the function `NewDimension()`. This will give the illusion that the `TDBCheckBox` is part of the grid.
There are a couple of things to pay attention to here:

- `FindGridControl()` uses `Column->Fieldname` as its parameter to find the correct control. In the third requirement above I said that it was necessary to determine certain things by using a grid's `SelectedField`. The `Column->Fieldname` replaces

`SelectedField` in this case. Use the VCL help fields to learn more about `TColumn`.

- The `Rect` variable only gives its cell coordinates relative to the grid. To get the correct left and top values for a control, the `Left` and `Top` properties of the grid must be added to the values in `Rect` (assuming the grid and the control have the same `Parent`). This is the reason for changing the `Parent` property in the `FindGridControl()` function. Changing the control's `Parent` property to the grid will not work (it may but I have not investigated it enough to find the trick).
- Finally, the controls need to be slightly offset and smaller than the actual cell to prevent overlap.

Drawing the cell

The second part of this function is to draw the correct image in the correct cell. This only occurs if the `State` variable is not set to `gdFocused` or `gdSelected` and `FindGridControl()` does not return `NULL`. If all is well, the `SetCheck()` function is called to draw the check mark in the cell. The value of the Boolean field is determined so the grid knows whether to paint a checked or an unchecked mark. The `Rect` parameter is passed to the `SetCheck()` function so that the mark is drawn in the correct location. Since there is no other use for the control at this point, the `FindGridControl()` function sets its `Visible` property to `false`.

Step five: handling loss of focus

Add an `OnColExit` event handler for the grid. The code for the event handler is shown in **Listing E**. As you can see, the event handler calls `FindGridControl()` function. It passes the grid's `SelectedField->FieldName` value for the first parameter, and `false` for the second parameter. If `FindGridControl()` returns a non-`NULL` value it means there is a control that needs to be hidden. Nothing is done with the return value so it is not used.

Step six: handling key press events

This is the last step. Create an `OnKeyPress` event handler for the grid and enter the code shown in **Listing F**. This event handler also calls the `FindGridControl()` function. If `FindGridControl()` returns a non-`NULL` values, the resulting pointer's `Visible` property is set to `true`. The remaining code forces the grid to repaint and then gives the appropriate control focus. As a last step, all keystrokes are sent to the control using the `SendMessage()` function.

Conclusion

If used judiciously, these relatively simple steps of adding controls to a grid can benefit the end user by making the application more user-friendly. There are some issues that are not addressed here, however. For example, there is nothing to keep the user from clicking "behind" the checkbox (the normal "True" or "False" will appear). Also, the `FindGridControl()` function would be much nicer if the controls had pointers stored in a `TList` instead of a static

`TWinControl` array. Still, you now have the basics for displaying other controls on a `TDBGrid`.

Acknowledgment: This article was based on an article and code originally written by Alec Bergamini (O&A Productions **www.o2a.com**) for Delphi 1.0 taken from a CD called Delphi Developers Kit. Used by permission

Listing A

```
void TForm1::SetCheck(
  const TRect &Rect, bool IsChecked)
{
  // Code to create checked
  // or unchecked mark
  DBGrid1->Canvas->FillRect(Rect);
  int SquareSize = 13;
  // Location of check mark
  int NewTop = NewDimensions(
    Rect.Top, Rect.Bottom, SquareSize);
  int NewLeft = NewDimensions(
    Rect.Left, Rect.Right, SquareSize);
  TRect ARect(NewLeft, NewTop,
    NewLeft + SquareSize, NewTop + SquareSize);
  UINT tmpState = DFCS_BUTTONCHECK |
    (IsChecked == true) ? DFCS_CHECKED : 0;
  DrawFrameControl(DBGrid1->Canvas->Handle,
    &ARect, DFC_BUTTON, tmpState);
}
```

Listing B

```
TWinControl* TForm1::FindGridControl(
  AnsiString FieldName, bool IsVisible)
{
  TWinControl *tmpWC = NULL;
  TWinControl *tmpWCDBControl[] = {DBCheckBoxGrid,
    FirstDBControl, SecondDBControl, NULL};
  AnsiString tmpWCFieldName[3];

  tmpWCFieldName[0] = DBCheckBoxGrid->DataField;
  tmpWCFieldName[1] = FirstDBControl->DataField;
  tmpWCFieldName[2] = SecondDBControl->DataField;
```

```
  int n = 0;
  while (tmpWCDBControl[n] != NULL)
  {
    if (FieldName == tmpWCFieldName[n])
    {
      tmpWC = tmpWCDBControl[n];
      tmpWC->Visible = IsVisible;
      if (tmpWC->Parent != DBGrid1->Parent)
        tmpWC->Parent = DBGrid1->Parent;
      break;
    }
  ++n;
  }
  return tmpWC;
}
```

Listing C

```
void __fastcall TForm1::DBGrid1DrawColumnCell(
  TObject *Sender, const TRect &Rect, int DataCol,
  TColumn *Column, TGridDrawState State)
{
  if (State.Contains(gdFocused) ||
      State.Contains(gdSelected))
  {
    TWinControl *tmpWC =
      FindGridControl(Column->FieldName, true);
    if (tmpWC != NULL)
    {
      if (tmpWC == DBCheckBoxGrid)
      {
        int BorderMargin =
          DBGrid1->BorderStyle == bsSingle ? 2 : 0;

          DBGrid1->Canvas->FillRect(Rect);
          //in case it is reused for multiple fields
          DBCheckBoxGrid->DataField =
            Column->FieldName;
          int NewLeft = NewDimensions(Rect.Left,
            Rect.Right, DBCheckBoxGrid->Width);
```

```
          tmpWC->Left = NewLeft +
            DBGrid1->Left + BorderMargin;
          int NewTop = NewDimensions(Rect.Top,
            Rect.Bottom, DBCheckBoxGrid->Height);
          tmpWC->Top = NewTop +
            DBGrid1->Top + BorderMargin;
      }
      else
      {
        int LineMargin = 1;
        tmpWC->Left = Rect.Left + LineMargin;
        tmpWC->Width =
          Rect.Right - Rect.Left - LineMargin;
        tmpWC->Top =
          Rect.Top + LineMargin + DBGrid1->Top;
        tmpWC->Height =
          Rect.Bottom - Rect.Top - LineMargin;
      }
    }
  }
  else
  {
    if (FindGridControl(Column->FieldName,
          false) == DBCheckBoxGrid)
    {
      bool IsChecked = Table1->FieldByName(
        Column->FieldName)->AsBoolean;
      SetCheck(Rect, IsChecked);
    }
  }
}
```

Listing D

```
void __fastcall TForm1::DBGrid1ColExit(
  TObject *Sender)
{
  FindGridControl(
    DBGrid1->SelectedField->FieldName, false);
}
```

Listing E

```
void __fastcall TForm1::DBGrid1KeyPress(
  TObject *Sender, char &Key)
{
  TWinControl *tmpWC = FindGridControl(
    DBGrid1->SelectedField->FieldName, true);
  if (tmpWC != NULL)
  {
    DBGrid1->Repaint();
    tmpWC->SetFocus();
    SendMessage(tmpWC->Handle, WM_CHAR, Key, 0);
  }
}
```

Listing F

```
int TForm1::NewDimensions(
  int Rect1, int Rect2, int Dimension)
{
  int NewDm = Rect2 - Rect1 - Dimension;
  NewDm = Rect1 + ((NewDm > 0) ? NewDm / 2 : 0);
  return NewDm;
}
```

Command line builds with make

by Kent Reisdorph

In the article “IDE command-line options” I explained the C++Builder IDE’s command-line options and how to use them. In all cases, those options simply control how the IDE starts.
In some cases it is beneficial to be able to build or make a project entirely from the command line (without starting the IDE at all). The make utility that ships with C++Builder allows you to do that. This article will explain how you can use make to build projects from the command line.

Make targets

The file specified as the input file for make is considered the target. One of the nice things about make is that it can be used to build not only projects, but also to build project groups and packages (C++Builder 3 and later). This makes it possible to write scripts (batch files or perl scripts, for example) to automate your build process. At TurboPower, we use make to build our C++Builder packages for a particular product.
The file passed to make must be a makefile. A makefile is a text file that contains a list of the files to compile, rules for how those files will be compiled, the locations of headers or library files that the project requires, instructions for the linker, and so on. Basically, a makefile contains all of the information that the compiler and linker need to build a given project.

Make for C++Builder 1-4

Fortunately, it is not important that you are able to read and fully understand makefiles to be able to use make. You may or may not be aware of this fact, but C++Builder 1-4 projects are actually makefiles. This makes it possible to pass a project file (.MAK for C++Builder 1, or .BPR for C++Builder 3 and 4) directly to make for processing. I’ll explain the makefile command-line options later in the article, but the most important option is the `f` option. It is used to specify the file that make will operate on. Given that, you can build a project from the command line like this:

```
c:\Borland\bcb4\bin\make -fproject1.bpr
```

When make runs, you will see a number of messages on the screen as make calls the compiler and linker. If either the compiler or linker encounters an error, an error message is displayed and make stops building the project.
Not only are project files makefiles, but so are project group files (.BPG) and package source files (.BPK). This means that you can build an entire group of projects with just one call from the command line.

Make for C++Builder 5

The make utility that ships with C++Builder 5 is no different than the make that shipped with previous versions of C++Builder. However, project files in C++Builder 5 are no longer makefiles. Instead, a project file in C++Builder 5 is XML-based. This makes it impossible to pass a C++Builder 5 project file to make as you could with previous versions.
The Borland designers provided a new menu item that exports the project file as a makefile. Choose Proejct | Export Makefile from the C++Builder 5 main menu. When you choose this menu item, the Save As dialog will be displayed. The Save As dialog will have a default name of PROJECT.MAK, where PROJECT is the name of the current project. All you have to do is hit the Enter key to create a makefile for the project. You can then pass this makefile to the make utility. Alternatively, you can use BPR2MAK.EXE (found in the C++Builder 5 BIN directory) to create makefiles for C++Builder 5 projects.
Both project files and package source files are in XML format so you will have to export the makefile for these project types. Project groups, however, still use the old format and can be used directly with make.
It is important to understand that the makefile generated by the IDE is static. If you change the project (by changing project options or adding units to the project, for example) you will have to regenerate the makefile to ensure that it is up to date. If you are building your projects or packages from a batch file, you can simply run BPR2MAK.EXE on the project file before calling make. Doing so will ensure that your makefiles are always up to date.

Make options

The make utility has a number of command-line options. The most often used of those options are listed in **Table A**. Some of the more esoteric options are not listed in this table. To see a complete list of the command-line options available for make, run make from the command line with the `h` option:

```
make -h
```

This will list all of the make command-line options.

Table A: make command-line options

`-B` Performs a full build of the target. If this option is not specified, a make of the target is performed rather than a build.
`-D` Defines a compiler symbol. This can be a simple symbol (`-DDEBUG`, for example) or can be a symbol with an associated string (as in `-DDEBUG=ON`).
`-I` Specifies an include directory where headers are located.
`-K` Temporary files created by make are preserved when make is finished. When this option is off (the default), temporary files are deleted when make is finished.
`-U` Undefines a compiler symbol

`-f` Specifies the target filename (a project file, a makefile, a project group, or a package).
`-a` Performs auto-dependency checks for included files.
`-c` Caches auto-dependency information for included files.
`-i` Ignores errors returned by commands. When this option is off (the default), make will terminate when a compiler or linker error is encountered.
`-l` Enables use of long command lines (on by default).
`-m` Displays the date and time stamp of each file compiled.
`-n` Outputs commands to the console but does not perform them. This can be used to preview a make before actually performing the make.
`-p` Displays all macro definitions and implicit rules. This is essentially a textual description of the symbols and rules that will be used when the target is built.
`-q` Returns zero if target is up-to-date and nonzero if it is not (for use in batch files).
`-s` Silent mode, does not print commands before doing them.
`-h` Displays the full list of command-line options for make.
`-?` Same as `-h`

Using make is fairly simple in most cases. The command-line options you are most likely to use are the `f` option, and the `B` option. To perform a complete build of your project, for example, you would use:

```
make -fmyproject.bpr -B
```

Make will take the project file and build the project. It's as simple as that.

Conclusion

In some cases, building from the command line using the make utility will result in faster builds. The primary reason to use make, though, is when using build scripts for complex projects. Complex projects often require a number of steps when built from the IDE (opening several project groups, packages, or projects and building each individually). By using make in a batch file (or other scripting language) you can reduce the number of steps required to build complex projects.

Hidden treasures of Sysutils, Part 4

by Mark G. Wiseman

In Parts 1 and 2 of “Hidden treasures of Sysutils”, I talked about variables and functions that worked with times, dates, numbers, and currency. In Part 3, I introduced you to functions that work with file names. In this article, I will discuss the functions in `Sysutils` that actually manage files and directories.
So, if these treasures are hidden in `Sysutils`, how did I find them? I simply looked at the source code. The files SYSUTILS.HPP and SYSUTILS.PAS provided the initial clues (function and variable names) and using these clues I was able to look up most of these treasures in the online help.

Directory functions

There are four functions in `Sysutils` that you can use to work with directories:

```
AnsiString GetCurrentDir();
bool SetCurrentDir(const AnsiString Dir);
bool CreateDir(const AnsiString Dir);
bool RemoveDir(const AnsiString Dir);
```

The `GetCurrentDir()` function returns an `AnsiString` containing the fully qualified path name of the current directory. Its counterpart, `SetCurrentDir()` takes an `AnsiString` specifying a directory and attempts to make that directory current. If `SetDurrentDir()` is successful, it returns `true`. If the call fails, `SetCurrentDir()` returns `false`. The argument to `SetCurrentDir()` may be a fully qualified path name, or a path name relative to the current directory: For example:

```
// fully-qualified path
bool success =
  SetCurrentDir("c:\\windows\\system");
// relative to the current directory
bool success = SetCurrentDir("system");
```

The `CreateDir()` function takes an `AnsiString` argument specifying a directory path and tries to create that directory. A return value of `true` indicates success. The argument to `CreateDir()` will only work if all the subdirectories in a directory path exist except for the lowest subdirectory. In the following line of code, `CreateDir()` will only succeed if the directory “c:\testing\part4” already exists.

```
bool success = CreateDir(
  "c:\\testing\\part4\\subtest");
```

The final directory function I want to talk about is `RemoveDir()`. This function takes an `AnsiString` specifying an empty directory and attempts to remove it. If it cannot remove the directory, `RemoveDir()` returns `false`. Note that the directory must be empty or the function will fail.

Disk information functions

There are two functions in `Sysutils` that help you find out about the size of and free space on a disk.

```
int64 DiskFree(Byte Drive);
__int64 DiskSize(Byte Drive);
```

Both of these functions accept a `Byte` argument that represents the drive to query for information. If this argument is 0, the functions query the current drive. Argument values of 1–26 correspond to drives A–Z.
The return value for both functions is an `__int64`, which is large enough to work with today's large hard drives. If either function should encounter an error—a non-existent drive, for example—the value `-1` is returned.

Directory search functions

There are four functions in `Sysutils` that allow you to search a directory for specific files. They are:

```
int FindFirst(const AnsiString Path,
  int Attr, TSearchRec &F);
int FindNext(TSearchRec &F);
void FindClose(TSearchRec &F);
AnsiString FileSearch(
  const AnsiString Name,
  const AnsiString DirList);
```

The first three of these functions are light wrappers around the Windows API functions `FindFirstFile()`, `FindNextFile()` and `FindClose()`. They are used to search through a specific directory for files matching a given file name pattern and attributes.
These three functions use the `TSearchRec` structure to hold data about the files they find.

Unfortunately, the `TSearchRec` structure introduces a serious flaw that makes the functions less useful than you might think. The `Size` member of this structure holds the file size. `Size` is declared as an `int`, not an `__int64`. This means that for very large files (greater than 4 GB), the size of the file may be reported incorrectly.
For this reason, I recommend that you use the Windows API functions. They are only slightly more difficult to use and they will report very large file sizes correctly. If, however, you are still interested in using the functions in `Sysutils`, there is an example in the program that accompanies this article and also a good example in the VCL help for `FindFirst()`.
The fourth function, `FileSearch()`, is a very useful function. This functions searches through a list of directories for a file name. It takes two `AnsiString` arguments: a file name and a semicolon-delimited list of directories. If `FileSearch()` finds the file, it returns an `AnsiString` with the fully qualified path to the file. If there is no match for the file name in any of the directories, `FileSearch()` returns an empty `AnsiString`. For example:

```
AnsiString fileName = "notepad.exe";
AnsiString dirList =
  "c:\\;c:\\Windows;c:\\WinNT;"
  "c:\\Program Files\\Accessories";
AnsiString path =
  FileSearch(filename, dirList);
```

This code searches for NOTEPAD.EXE in the four directories specified in the `dirList` variable. On my system, a value of “c:\Windows\notepad.exe” is returned in the variable `path`.

File date functions

There are several functions in `Sysutils` that you can use to work with file dates. Some of them are:

```
int FileGetDate(int Handle);
int FileSetDate(int Handle, int Age);
int FileAge(const AnsiString FileName);
System::TDateTime
  FileDateToDateTime(int FileDate);
int DateTimeToFileDate(
  System::TDateTime DateTime);
```

The `FileGetDate()` and `FileSetDate()` functions work with a file handle on an open file to get and set the date of the file.
`FileGetDate()` takes a file handle as an argument and returns an `int` containing a DOS date-

time value. If an error is encountered, `FileGetDate()` returns `-1`.
`FileSetDate()` takes an `int` argument for the file handle, and a second `int` argument containing a DOS date-time value. It attempts to set the file's date using the value passed in the second parameter. If unsuccessful, `FileSetDate()` returns `-1`.
As you probably know, Windows maintains three dates for a file: the creation date, the date the file was last accessed, and the date the file was last modified. You might be wondering which of these three file dates, the preceding functions work with. The Borland online help says it's the DOS file date which, when translated to Windows, means the last modified date.
The `FileAge()` function, like `FileGetDate()`, returns the last modified date of file. Unlike `FileGetDate()`, `FileAge()` works on closed files. It takes an `AnsiString` argument containing the file name and returns a DOS date-time value.
The `FileDateToDateTime()` and `DateTimeToFileDate()` functions can be used to convert a DOS date-time value to a `TDateTime` and vice versa.

File attribute functions

There are two functions that will retrieve and set the attributes of a file:

```
int FileGetAttr(
  const AnsiString FileName);
int FileSetAttr(
  const AnsiString FileName, int Attr);
```

`FileGetAttr()` takes an `AnsiString` argument containing a file name, and returns an `int`. The return value can be a combination of the values `faReadOnly`, `faHidden`, `faSysFile`, `faVolumeID`, `faDirectory`, and `faArchive`.
`FileSetAttr()` also takes an `AnsiString` representing a file name, and an `int` containing a combination of attributes. If `FileSetAttr()` is successful, it returns `0`. Yes, that's right a zero on success.

File rename and delete functions

There are two files in `Sysutils` that you can use to rename and delete files. These functions are (you guessed it!):

```
bool DeleteFile(
  const AnsiString FileName);
bool RenameFile(const AnsiString OldName,
  const AnsiString NewName);
```

Both functions return true if they're successful and false if they're not.

File read, write, and seek functions

The last group of functions I want to tell you about are functions you can use to create, open, read, write, and seek into files. All of these functions are thin wrappers around Windows API functions.

```
int FileCreate(
  const AnsiString FileName);
int FileOpen(const AnsiString FileName,
  unsigned Mode);
int FileRead(int Handle,
  void *Buffer, unsigned Count);
int FileWrite(int Handle,
  const void *Buffer, unsigned Count);
void FileClose(int Handle);
int FileSeek(int Handle,
  int Offset, int Origin);
__int64 FileSeek(int Handle,
  const __int64 Offset, int Origin);
```

The `FileCreate()` function accepts a file name as an `AnsiString` argument and returns a file handle (an `int`) to a file that is open and ready for reading and writing. A return value of `-1` indicates an error.

`FileOpen()` takes two arguments: an `AnsiString` containing a file name and an `unsigned int` representing the mode for opening the file. The mode can be one of several values including `fmOpenRead` and `fmOpenReadWrite`. For a complete list of modes, see the online help.

The `FileRead()` function takes as arguments a file handle, a pointer to a buffer, and an `unsigned int` that represents the number of bytes to be read into the buffer. `FileRead()` returns an `int` that indicates the number of bytes actually read into the buffer. This function reads from the current position in the file buffer.

The `FileWrite()` function takes a file handle, a `void` pointer to raw data, and an `int` specifying how many bytes of the raw data should be written into the file at the current buffer position. `FileWrite()` returns an `int` that is the number of bytes written, or `-1` if there was an error.

The `FileSeek()` function is an overloaded function. One version uses arguments and return values of `int`. The other uses `__int64` for very large files. Both functions take a file handle as their first argument, an offset value as the second argument, and an `int` representing the origin of the offset as the third. Using these arguments, `FileSeek()` moves the file pointer. The file pointer is the spot in a file where the next byte will be written or the next byte will be read.

The offset argument is either an `int` or an `__int64`. The functions return either an `int` or an `__int64`, which will contain the new position of the file pointer or `-1` if an error occurred.

The origin argument, an `int` in both versions, should be `0` to offset the file pointer from the

beginning of the file, 1 to offset the pointer from its current position, or 2 to offset the pointer from the end of the file.
The following code creates a file named TESTFILE.TXT, writes a line of text to the file, and then closes the file. Next it opens the file for reading only, seeks to a point in the file where the words "some text" are located, and reads those words into a buffer.

```
String fileName = "TestFile.txt";

int handle = FileCreate(fileName);
if (handle == -1) return;

char text[] =
  "This is some text for our test file.";
int error =
  FileWrite(handle, text, strlen(text));
if (error == -1)
{
  FileClose(handle);
  return;
}

FileClose(handle);

handle = FileOpen(fileName, fmOpenRead);
if (handle == -1)
  return;

// Seek in 8 bytes from
// the beginning of the file
int pos = FileSeek(handle, 8, 0);
if (pos == -1)
{
  FileClose(handle);
  return;
}

// Read 9 bytes into buffer
count = FileRead(handle, buffer, 9);
buffer[count] = 0;
```

Conclusion

Well, I guess it's about time to close the file on `Sysutils`. I have just a couple of suggestions for you before I do.

First, there are even more useful functions for dealing with files and file names. You can find these in the FILECTRL.HPP header file. In particular, look at the `MinimizeName()`, `DirectoryExists()` and `ForceDirectories()` functions.

Second, explore the VCL source code that Borland has included with C++Builder. You will find useful classes and functions that you didn't know were there. These are indeed hidden treasures.

IDE command-line options

by Kent Reisdorph

The C++Builder IDE has a number of command-line options that can be used to control how the IDE starts. For the most part, these command-line options are undocumented (they are documented in C++Builder 5 but not in earlier versions). You can use the C++Builder IDE much more effectively by better understanding the IDE's command-line options.

Using the command-line options

Not all of the command-line options described here are available on every version of C++Builder. As far as I know, only C++Builder 5 documents these options. To view the help page that describes the command-line options for C++Builder 5, see the topic "command-line options, IDE (BCB.EXE)" in the help file. Alternatively you can start C++Builder 5 with the `-?` option. When C++Builder 5 is started with this option, the help page describing the command-line options will be displayed.

The command-line options can be specified using either a forward slash or a dash. If, for example, you wanted to start the IDE with the `ns` option (explained in the next section) you would use either of these commands:

```
bcb.exe /ns
bcb.exe -ns
```

The command-line options are not case sensitive so `ns` and `NS` result in the same behavior.

The easiest way to implement the command-line options is to simply add them to the shortcut for the IDE (the IDE executable is BCB.EXE and is in your BIN directory).

The following sections explain the command-line options, broken down into relevant categories.

Startup options

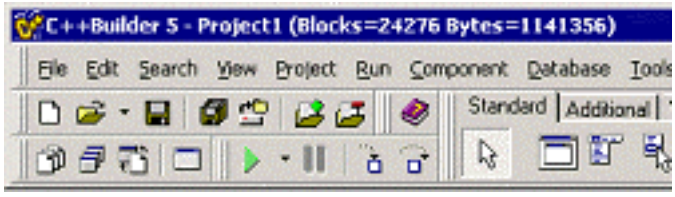
The startup options are `ns` and `np`. The `ns` option (which is short for "no splash") suppresses the splash screen that is displayed when C++Builder is loading. On some systems, bad video drivers can cause the IDE to generate an error when the IDE starts up or, in some cases, shuts down. By inhibiting the splash screen, these errors are usually eliminated.

The `np` option ("no project") is only available in C++Builder 5. This option starts the IDE but does not create a default blank project. This is particularly useful for those users who dislike the initial blank project that is displayed by default. I don't mind the blank default project, but I know that many C++Builder users have complained about it. Now those users have a way of eliminating the blank project on startup (C++Builder 5 users anyway).

Heap options

The heap command-line options are `hm` and `hv`. The `hm` option ("heap monitor") causes the IDE to display heap statistics in the IDE's title bar. **Figure A** shows the C++Builder 5 IDE's title bar when the IDE is started with this option. I suspect that this option is used primarily as an IDE debugging tool for the Borland developers. However, it is sometimes interesting to watch the IDE's memory usage during a build. It can also be used if you notice odd IDE behavior and wish to report that behavior to Borland.

Figure A



The hm option displays IDE heap usage in the title bar.

The `hv` ("heap verification") option is used to track memory errors that occur in the IDE. If any heap errors are detected, they are report in the IDE's title bar.

Project options

Project options are used to load a specific project when the IDE loads; to build or make a project when the IDE loads; and to specify an output file for compiler errors or warnings.
To specify the project that should be loaded when the IDE starts, simply add the project name to the command line:

```
bcb.exe c:\myfiles\myproject.bpr
```

This allows you to have several shortcuts to C++Builder, each specifying its own project. The command-line option is available in all versions of C++Builder.
The `b` option ("build") tells the IDE to build the specified project when the IDE loads. For example:

```
bcb.exe c:\myfiles\myproject.bpr -b
```

When the `b` option is used, the IDE will start; load the specified project; build it; and then close.
The `m` option ("make") works just like the `b` option but makes the project rather than building it.
The `o` option ("output") allows you to specify an output file for compiler errors and warnings. This option must be used with either the `m` or `b` options. The resulting error file will be placed in the project directory.

The `m`, `b`, and `o` options are only available in C++Builder 5.

Debugger options

The debugger command-line options are more for power users. As such, I'll only give them light treatment here.

The `d` option ("debug") is used to open the IDE and debug an existing project. Obviously, the project must have been built using debug information (the default for new projects). Simply specify the EXE name following the `d` option:

```
bcb.exe -dc:\myfiles\myproject.exe
```

When this command executes, the IDE will load; load the specified executable; and start the debugger. The debugger will pause in the application's startup code (in `WinMain()` for a GUI application). This option is available on C++Builder 4 and 5.

The `attach` option allows you to attach a running process to the debugger. It is the same as choosing Debug | Attach to Process from the IDE's main menu.

The `td` option ("TurboDebugger") is used with the `d` option. It controls how the IDE behaves when the debugger runs. For one thing, it causes the CPU and FPU windows to stay open when the process terminates.

The `sd` option ("source directories") allows you to specify the directories where the source code is loaded. It is used with the `d` option.

The `h` option ("hostname") is used to specify the remote host name of a remote machine when performing remote debugging. It is used with the `d` option.

The `l` option (a lowercase L) tells the debugger to start debugging but to skip the startup code. I'm not sure how useful this is, as the debugger stops with the CPU window initially displayed.

Conclusion

Most programmers will never use the bulk of the command-line options available in C++Builder. However, the `np` and `ns` options alone can be invaluable. Knowing exactly what your tools are capable of is part of the key to success.

Loading tree views the easy way

by Kent Reisdorph

Tree views offer a convenient way of displaying hierarchical information. One of the problems with tree views, however, is that loading the tree view can be tedious. It is tedious because you need to first create a node and then create one or more sub-nodes under that node. Also, you may need to add individual items to specific nodes.

Fortunately, TTreeView has a LoadFromFile() method that makes it very easy to populate a tree view. You can use LoadFromFile() any time you have a tree view that contains static information. (If your tree views display dynamically determined information then you will need to load the tree the hard way.)

As you might expect, the LoadFromFile() method takes the name of a file as a parameter. The file is a text file in a specific format that TTreeView expects. The format is very simple: each level of the tree view is indented one space. Take the following text, for example:

Dogs

Big dogs

Rottweiler

Doberman Pinscher

Great Dane

Small dogs

Jack Russell Terrier

Yorkshire Terrier

Toy Poodle

Cats

Good cats

(none)

Other cats

Mexican Hairless

Siamese

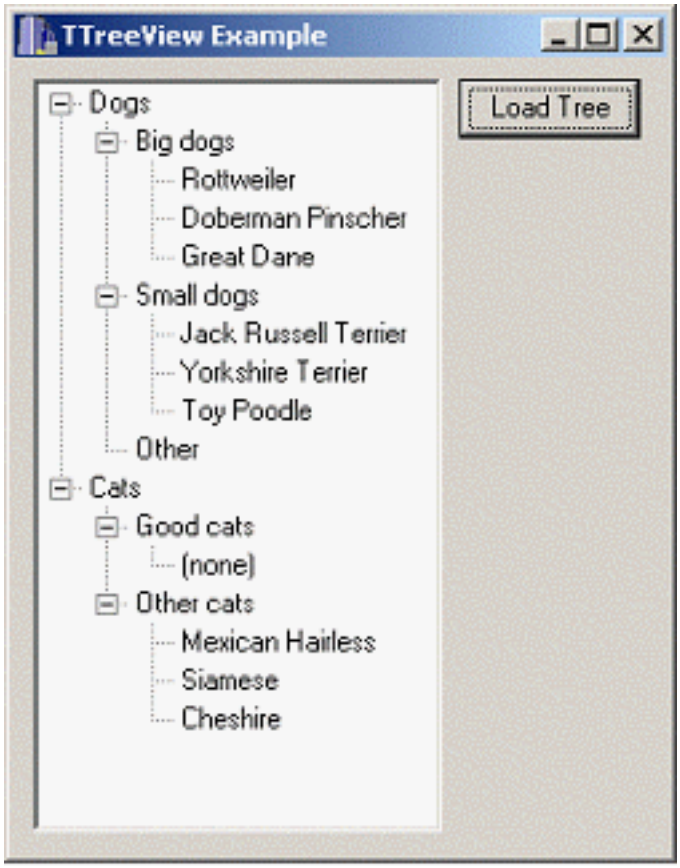
Cheshire

Once you have created the text file, the code to load it into the tree view is simple:

```
TreeView1->LoadFromFile("c:\\pets.txt");
```

Figure A shows the tree view as it looks after loading this text using the LoadFromFile() method. As you can see, the topics "Dogs" and "Cats" are top-level nodes in the tree view. Under the "Dogs" node are nodes called "Big dogs" and "Small dogs". Each of these nodes has individual items. The "Cats" node is structured the same way.

Figure A



Populating a TTreeView is easy with the LoadFromStream() method.

In addition to LoadFromFile(), TTreeView has a LoadFromStream() method. This method allows you to populate a tree view from any TStream descendant. Loading a tree view from a stream presents many opportunities, including loading the tree view from a database field or from a resource bound to the EXE.

The LoadFromFile() and LoadFromStream() methods are invaluable any time you need to load a tree view from static text.

Understanding AnsiString's c_str() function

by Kent Reisdorph

As you probably know, the VCL is written in Object Pascal. Object Pascal contains a dynamic string data type called `AnsiString`. The Object Pascal `AnsiString` type is dynamic in that memory is allocated and deallocated by the compiler as needed. The VCL makes heavy use of the `AnsiString` data type.
Unlike Object Pascal, C++ does not have a native data type for strings. Instead, character arrays are used to hold string data (often called null-terminated strings). C++Builder's `AnsiString` class was designed to interface with Object Pascal's `AnsiString` type.
The `AnsiString` class has a wide variety of methods that make string manipulation easy. Perhaps the most widely misunderstood (and abused) `AnsiString` function is the `c_str()` function. It can helpful to know when to use `c_str()` and the pitfalls to avoid when using it.

The advantages of a string class

A string class makes working with strings much easier than is possible using character arrays and the C runtime library functions. As an example, consider the case where you want to concatenate two strings. The C code to concatenate two strings looks like this:

```
char buff[20];
strcpy(buff, "Hello");
strcat(buff, " World!");
Label1->Caption = buff;
```

This is fairly straightforward, but careful attention must be paid to ensure that you don't overwrite the end of the character array. Using `AnsiString`, you can accomplish the same thing with this code:

```
String S = "Hello";
S += " World!";
Label1->Caption = S;
```

Obviously, this code is more readable and more intuitive than the C code presented in the first example. Concatenation of strings is a very simple example. Deleting part of a character array, for example, is much more work using the C functions than it is using the `AnsiString` class.

Ultimately, the string data for any C++ string class is still stored within the class as a null-terminated string. In the case of `AnsiString`, a private data member called `Data` holds the string data. `Data` is declared in DSTRING.H as follows:

```
private:
  char *Data;
```

The memory needed to hold the string data is dynamically allocated as needed. For example, concatenating two strings together will result in the memory associated with the `Data` variable being deleted and then reallocated to account for the new size.

The c_str() function

The `c_str()` function returns a pointer to `AnsiString`'s private `Data` variable. It is an inline function whose declaration is ridiculously simple:

```
char* __fastcall c_str() const
  { return (Data)? Data: "";}
```

If `Data` is non-null, `c_str()` returns a pointer to the data. If `Data` is null, `c_str()` returns an empty string.
When do you use the `c_str()` function? Basically, you use `c_str()` when passing the string represented by an `AnsiString` to any function requiring a `char*`. The `TCanvas` class, for example, does not have a `DrawText()` method. In order to use the Windows API function `DrawText()` you need to pass a `char*` for the second parameter. For example:

```
String S = "Hello World!";
TRect R(20, 20, 100, 40);
DrawText(Canvas->Handle,
  S.c_str(), -1, &R, DT_SINGLELINE);
```

The most common use of `c_str()` is when calling API functions, or when passing data from an `AnsiString` to other string classes (such as the STL's `basic_string` class).

Common misuses of c_str()

The `c_str()` function is necessary but it is often misused by C++Builder programmers. This section will address common misuses of `c_str()`.

Using c_str() out of scope

It is important to understand that the pointer returned by `c_str()` is temporary. That is, it is not

guaranteed to exist past the line of code in which it is used. For example, the following code, while syntactically correct, will likely fail:

```
char* buffer = S.c_str();
strcat(buffer, " more text");
```

In some cases this code may work, but if it does it is more luck than science. The code may appear to work, but often the `buffer` variable will point to some random data in the second line of this example.
If you need to store the string data contained in an `AnsiString`, you need to allocate memory and copy the data pointed to by `c_str()` as shown here:

```
char* buffer = new char[reqLen];
strcpy(buffer, S.c_str());
strcat(buffer, " more text");
delete[] buffer;
```

This is the only way to ensure that you are working with data that is, in fact, valid.

Writing to c_str()

Simply put, you should never attempt to write directly to `c_str()`. For example, the `GetWindowsDirectory()` API function is used to obtain the directory where Windows is installed. The first parameter of `GetWIndowsDirectory()` is used to specify a `char*` that will contain the Windows path when the function returns. You might be tempted to use code like this:

```
String S;
GetWindowsDirectory(S.c_str(),MAX_PATH);
Label1->Caption = S;
```

This code compiles and runs without error, but will produce erroneous results. In this example, `AnsiString` has not allocated any storage for the `Data` variable. If you look back and examine the code for `c_str()` you will see that the code above evaluates to something like this:

```
GetWindowsDirectory("", MAX_PATH);
```

Remember that `c_str()` will return an empty string if no memory has been allocated for the string data. For this reason, the previous code won't produce the expected results, but it also won't generate any errors. In some cases, though, writing to `c_str()` may result in an access violation. Here's an example:

```
String S = "hello";
GetWindowsDirectory(S.c_str(),MAX_PATH);
Label1->Caption = S;
```

In this case `AnsiString` allocates six bytes of storage to hold the assigned string. When `GetWindowsDirectory()` executes, the internal `AnsiString` buffer is overwritten by several bytes (the exact number depends on where Windows is installed on your system). The result is an access violation, either immediately or sometime later in the program's execution.
You can force `AnsiString` to allocate memory by calling the `SetLength()` method. For example:

```
String S;
S.SetLength(MAX_PATH);
GetWindowsDirectory(S.c_str(),MAX_PATH);
Label1->Caption = S;
```

This code produces the expected result, but it is still best to avoid writing to the `c_str()` function at all. Doing so is not good practice and there is always an alternative. The proper way to write the code in preceding examples is like this:

```
char buff[MAX_PATH];
GetWindowsDirectory(buff, MAX_PATH);
String S = buff;
Label1->Caption = S;
```

`AnsiString` will make a copy of the data contained in the `buff` variable and will allocate the proper amount of storage.

Conclusion

The `AnsiString` class is a powerful way to manipulate string data. Its `c_str()` function is a necessary and valuable tool, but must be used with care. Understanding how to properly use `c_str()` is vital to writing error-free applications.

Customized form navigation

by Kent Reisdorph

Microsoft has defined a set of rules for how a Windows program should look and act. This includes rules for keyboard navigation of windows. This is true whether a particular window is a main window or a dialog box. The most basic keyboard handling rules are fairly simple to state:

- The Tab key moves focus forward in the tab order
- The Shift-Tab key combination moves focus backwards in the tab order
- The Enter key invokes the default button on a dialog if one is designated (usually the OK button)
- The Esc key closes a dialog (the same as clicking the Cancel button)

Further, some controls have built-in keyboard handling. Pressing the space bar when a check box has focus, for example, will mark or clear the check box. Another obvious example is the case of a multi-line edit control where the control can be navigated using the arrow keys, the Enter key, and, if enabled, the Tab key.
For the most part I highly recommend that you follow Microsoft's rules for keyboard navigation of forms. However, there are some specialty applications that require special window navigation features. This article will explain how you can implement customized keystroke handling for your applications.

Keystroke handling events

The VCL defines two events for keystroke handling: `OnKeyDown` and `OnKeyPress`. The `OnKeyPress` event corresponds to the Windows `WM_CHAR` message. In simple terms, this message is generated whenever a displayable key is pressed (such as one of the letter or number keys). It is also generated when the Enter key is pressed.
The `OnKeyDown` event, on the other hand, is fired both when displayable characters are pressed and when special keys are pressed. Special keys include function keys, arrow keys, page movement keys (Insert, Home, Delete, End, Page Up, and Page Down) and so on. The `OnKeyDown` event will be fired for just about every key on the keyboard. Exceptions are special Windows key combinations such as Ctrl-Alt-Delete and Alt-Tab. Windows grabs these key combinations and doesn't pass them on to applications.
It is important to understand that pressing an alphanumeric key will result in both an `OnKeyDown` and an `OnKeyPress` event being fired. This article will focus on non-alphanumeric keys and will thus be using the `OnKeyDown` event for examining keystrokes.

Both of the keystroke events contain a parameter called `Key`. This parameter can be used to determine the key that was pressed. Windows defines a number of constants that map to keys on the keyboard. These virtual key constants all begin with `VK_` and are defined in WINUSER.H. For example, the virtual key constant that represents the Enter key is `VK_RETURN`, the constant that represents the Shift key is `VK_SHIFT`, the constants that represent the arrow keys are `VK_LEFT`, `VK_RIGHT`, `VK_UP`, and `VK_DOWN`, and so on. You can compare the value of the `Key` parameter to one of these constants to determine which key was pressed.

The KeyPreview property

Custom keyboard navigation requires that you grab keystrokes as Windows and the VCL process those keystrokes. There are several ways to go about intercepting keystrokes. One way is to catch the `OnKeyDown` and/or `OnKeyPress` events for each component. Another way is to have the form receive and act on all keyboard events. This can be accomplished by setting the form's `KeyPreview` property to `true`.

When `KeyPreview` is `true`, the form's `OnKeyDown` and `OnKeyPress` events will be fired when a key is pressed and a component has focus. The event for the form will be fired before the corresponding event in the component is fired (hence the "preview").

All of this assumes that a particular component is capable of catching keystrokes. Group boxes, for example, handle keystrokes internally and, as such, cannot pass keystrokes on to the form.

Using the `KeyPreview` has one obvious advantage: keystroke handling for all components on the form can be centralized. Rather than having handlers for each component's keystroke events, you can have one handler that previews all keystrokes.

One disadvantage to using `KeyPreview` is that the form's `OnKeyDown` and `OnKeyPress` event handlers can be complex. Whether you use `KeyPreview` for your forms depends on the specific needs of that form. The remainder of this article will assume a form with `KeyPreview` set to `true`.

Enter as tab

One of the more common needs is to have the Enter key act as the Tab key when an edit control has focus. This may be advantageous for forms that have a lot of data entry fields. Users can quickly enter data and simply hit the Enter key to move to the next field. This is particularly appealing in applications that require a large amount of numerical data entry. For example, an experienced 10-key operator is accustomed to using the Enter key on the numeric keypad after entering data. By making the Enter key act as the Tab key, users of this type of application can be much more efficient at data entry.

Changing focus when the Enter key is pressed is a relatively simple process. First, make sure that all components have their `TabStop` property set to `true` and that the tab order for the form has been properly set. After you have done that, the event handler for the form's `OnKeyDown` event might look like this:

```
void __fastcall TForm1::FormKeyDown(
  TObject *Sender, WORD &Key,
  TShiftState Shift)
{
  if (Key == VK_RETURN) {
    TWinControl* ctrl = FindNextControl(
      ActiveControl, true, true, false);
    if (ctrl)
      ctrl->SetFocus();
  }
}
```

This code first examines the `Key` parameter to see if the Enter key was pressed. If so, the `FindNextControl()` method of `TForm` is called to obtain the control that comes next in the tab order. After obtaining a pointer to the next control in the tab order, the `SetFocus()` method is called for that control.
This code is simplified in that it doesn't check to see what type of control currently has focus. You will implement some mechanism to determine whether the focused control is an edit control before switching the focus to the next control in the tab order. You can do this in one of two ways. One way is to use `dynamic_cast` to ensure that the focused control is an edit control before changing focus. For example:

```
if (dynamic_cast<TEdit*>(ActiveControl))
{
  // it's a special navigation component
}
```

Another way is to use the `Tag` property to designate the components that allow the Enter key to act as the Tab key. For example, you could set the `Tag` property to `1` for these controls. In that case the code might look like this:

```
if (ActiveControl->Tag == 1)
{
  // it's a TEdit
}
```

This method is a bit more flexible in that it allows you to specify exactly the components that allow Enter as Tab.

Using the arrow keys

For some applications you may wish to allow the user to press the arrow keys to change focus from one control to the next. The code to implement this is a variation on the code in the previous section. Here is an example:

```
TWinControl* ctrl;
switch (Key) {
  case VK_UP : {
    ctrl = FindNextControl(
      ActiveControl,false, true, false);
    if (ctrl)
      ctrl->SetFocus();
    break;
  }
  case VK_DOWN : {
    ctrl = FindNextControl(
      ActiveControl, true, true, false);
    if (ctrl)
      ctrl->SetFocus();
    break;
  }
}
```

The key element to this code is the second parameter passed to the `FindNextControl()` method. When `true` is passed for this parameter, a pointer to the next component in the tab order is returned. When `false` is passed, a pointer to the previous component in the tab order is returned. The result is that focus moves back in the tab order when the up arrow key is pressed, and forward in the tab order when the down arrow key is pressed.

This is fine when all of the controls on a form are edit controls. However, if a ListBox, ComboBox, Memo, or RichEdit component has focus, then that component's default arrow key handling will be interrupted. You can easily write code around this possibility, but you must be aware of the types of components your form contains.

Naturally, you can add handling for the left and right arrow keys as well. However, you need to again carefully consider the components that are on your form. The user expects the right and left arrow keys to move the editing caret in an edit control from character to character. You don't want to replace such obvious editing behavior unless you have a very good reason to do so.

Overall, getting notification that an arrow key was pressed is easy. From there on you will likely have to write code to determine the type of component that has focus and determine how to handle the arrow keys for each component type as needed.

Finally, let me say that you cannot necessarily handle arrow keys for every component type. The

RadioGroup and GroupBox components, for example, handle the arrow keys internally. They do not surface those keystrokes to the form.

Conclusion

As I said at the beginning of this article, you should follow Microsoft's guidelines regarding keyboard navigation whenever possible. If, however, you have an application that needs custom keyboard handling then you can use the techniques described in this article to implement that kind of support.

Exceptions and databases

by Mark Cashman

While exceptions in normal code are fairly easy to understand and control, the handling of database exceptions can be more confusing. Part of the confusion comes from the nature of database operations in C++Builder, where many normal database events occur without direct control from code.
The proper management of exceptions falls into a few categories, some of which are related to the architecture of your application, and some of which are more incidental.
This article will explain some of the database exceptions that can occur and how to handle them properly.

Databases and C++Builder

As you probably know, C++Builder database programs can be written without a single line of code. (See my article at **http://www.temporaldoorway.com/programming/cbuilder/basics/simpledbprogram.htm**.) This presents a number of problems that can be formulated as questions typically asked by the developer:

- How can I handle exceptions that occur when I open a database, table, or query, especially if that open is implicit in creating a form or data module?
- How can I handle exceptions that occur during data entry when using data aware controls (for instance as a result of validation attempts on input)?
- How can I handle exceptions that occur when a data aware control posts a record to a table or query (for instance as a result of a DBMS problem or an invalid field in a row being posted)?

The following sections will address these questions.

Architectural issues

As always, the structure of your program has a great influence on the ease with which these questions can be answered. In addition, the VCL offers specialized features that can help you develop solutions to exception handling problems, or to avoid having to handle exceptions entirely.
The project source contains the catch-all (no pun intended) exception handler for the entire application. To see the project source, choose Project|View Source from the C++Builder main menu. The project source looks like this:

```
WINAPI WinMain(
  HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    Application->Initialize();
    Application->CreateForm(
      __classid(TForm1), &Form1);
    Application->Run();
  }
  catch (Exception &exception)
  {
    Application->
      ShowException(&exception);
  }
  return 0;
}
```

If an exception occurs at application startup, the exception will be reported and the program will terminate. You can place code in the `catch` block of the project source, but about all you can do is gracefully report the error and allow the application to terminate. In theory, you might even restart the application, for instance by correcting the error condition and issuing another `Application->Run()`. In practice, this is rarely useful. However, this is the place to deal with errors that occur as a result of the creation of forms and data modules. Types of errors that might occur in a database application include:

- A mismatch between the type of a persistent field and the underlying database field.
- An error in the activity of a form or data module constructor. For example, an attempt to open an improperly specified table or query as a result of the `Active` property having been set to `true` at design time (tables and queries with `Active` set to `true` at design time are opened by the form or data module constructor).

For more graceful error management, including the possibility of recovery, close all datasets at design time, and open the dataset within a `try`/`catch` block. You can do this either in the constructor (with appropriate recovery actions) or in a method to be called by another form. Here’s how the code might look:

```
for (int i=0;i<ComponentCount;i++)
{
  try
  {
    TQuery* Qry = dynamic_cast<TQuery*>
      (Components[i]);
    if (Qry)
    {
      if (Qry->SQL->Text.Length() > 0)
      {
        Qry->Open();
      };
    }

    TTable* Tbl = dynamic_cast<TTable*>
      (Components[i]);
    if (Tbl)
    {
      if (Tbl->DatabaseName.Length() > 0
          && Tbl->TableName.Length() > 0)
        Tbl->Open();
    }
  }
  catch (Exception &E)
  {
    Application->ShowException(&E);
    // Or other code to handle exception
  };
}
```

This code attempts to open tables or queries when the application starts. If an exception is thrown, the application attempts to continue. In addition, this code avoids attempting to open components that have not been properly initialized (due to a bad table name, missing SQL statement, and so on). You can also arrange for the exception handling to terminate the application, either by directly calling `Application->Terminate()`, or if this code is in a constructor, by re-throwing the exception. In that case, it is best to indicate the lack of recovery by placing the `try`/`catch` outside the loop.

Once the application has started, you can use the `TApplication` class to catch exceptions at a global level. This is done by providing an event handler for the `OnException` event. In C++Builder 5, this handler can be easily implemented via the `TApplicationEvents`

component. This component allows you to assign event handlers to `TApplication` events using the Object Inspector just as you do with other components.
In previous versions of C++Builder, it is necessary to implement the event handler and then to assign its address to the `OnException` property of the `Application` object. Here's an example event handler for the `OnException` event:

```
void __fastcall
TExceptionForm::HandleOnException(
  System::TObject* Sender,
  Sysutils::Exception* E)
{
  ReportException(E.Message);
};
```

After defining the event handler, simply assign its address to the `OnException` event:

```
Application->OnException =
  ExceptionForm->HandleOnException;
```

Obviously, the `OnException` handler can be used to deal with a variety of database errors that might occur once the program has started. However, it is better to first consider some architectural alternatives.

Reporting database errors during normal operation

A properly structured C++Builder program separates the database operations from the user interface, primarily through the use of data modules. This allows the data to be linked to a variety of user interfaces. However, for this scheme to work, the data module must not be dependent on any aspect of a particular user interface. In particular, it must not use any controls from a specific form to report errors.
So how do you report errors that occur during the database operations of the data module? Exceptions, of course, are the obvious answer.
If the database error occurs during a request from a form to a data module, that request can be placed within a `try`/`catch` block. Take, for example, this code that accesses a data module called `Account`:

```
Account->Database->StartTransaction();
try
{
  Account->Table->Open();
```

```
  Account->Table->Append();
  Account->TableID->AsString =
    AccountEdit->Text;
  Account->Table->Post();
  Account->Database->Commit();
}
catch (EDatabaseError &DatabaseError)
{
  Account->Database->Rollback();
  ReportError(DatabaseError.Message);
};
```

In this example, one or more `TTable` methods may throw an exception. Further the `Commit()` method may throw an exception if the changes cannot be written to the database. The `catch` block handles this possibility by rolling back the transaction and reporting the error to the user. But what about an error that occurs as a result of an operation from a data-aware control, an equivalent post from a `TDBGrid` for instance? The only place those can be caught will be in the `TApplication` class's `OnException` handler. This means that the data module needs to provide sufficient information for the application to direct the error to the appropriate form, or to allow the application to present the message directly from the `OnException` handler.

Validation and error messages

Many developers who learned programming in languages without events—and without the specialized database support provided by C++Builder—tend to bind database operations to the visual control. This includes database field entry validation. However, data modules provide a variety of facilities to avoid this dependence of validation on visual presentation.

When it comes to validating entry to a database field, you need first to construct persistent fields for all of your datasets. For those fields which will be presented in a user interface, and for which validation must therefore be performed, you need to use the `OnValidate` event handler provided by all `TField` descendants. Logic in this event handler can check the data against a mask, against other data in the row, or against literal data encoded in the event handler. If the data is in error, the field can ensure that the record is not posted by either throwing an exception to be caught by the `OnException` event handler, or by producing a message itself and calling the special, silent exception function `Abort()` (note that the first letter must be capitalized or you will end up calling the `abort()` function in the RTL). `Abort()` throws a silent exception that bypasses all handlers, including `TApplication`'s `OnException` event handler.

Naturally, throwing an exception to the application is preferable to presenting a message. Remember that the data module, and even the validation facilities, may be reused in non-interactive contexts. It may even be used as part of a distributed system and may run on a machine without a user interface or a screen. If you can avoid having the data module present errors, you

have more possibilities for reuse.

Conclusion

Exceptions are just as powerful for handling and providing information on errors arising from database operations as they are in handling and providing information on other errors in processing. Properly combined with the other features of C++Builder, they can allow you to decouple your database operations and error management from the user interface, thus improving the degree to which you can reuse your database objects.

Hidden treasures of Sysutils, part 3

by Mark G. Wiseman

What's in a file name? Well, actually there are a lot of parts to what is generally referred to as a file name. A file name may contain a drive letter, a directory and subdirectories, the filename itself, and an extension. The `Sysutils` namespace of the VCL provides several useful functions for working with filenames. In Parts 1 and 2 of this series, I discussed variables and functions that worked with times, dates, numbers, and currency. In this article, I will talk about functions that work with file names.

As I mentioned in Parts 1 and 2, the treasures in Sysutils are not really hidden. Most of them are in the online help, although many of them can be hard to find if you don't know exactly what you are looking for.

A useless typedef

Before I get into the heart of the article, I want to tell you about a `typedef` in the VCL. That `typedef` is defined as follows:

```
typedef AnsiString TFileName;
```

In the online help, Borland says, "Use `TFileName` to represent strings that are only used for file names." On the surface, this `typedef` makes a lot of sense. But, as you will see in this article, the folks at Borland did not follow their own advice. All of the following functions use `AnsiString` instead of `TFileName`!

File name parts

Earlier I asked, "What's in a file name?" You can use these functions to find out:

```
AnsiString ExtractFilePath(
  const AnsiString FileName);
AnsiString ExtractFileDir(
  const AnsiString FileName);
AnsiString ExtractFileDrive(
  const AnsiString FileName);
AnsiString ExtractFileName(
  const AnsiString FileName);
AnsiString ExtractFileExt(
  const AnsiString FileName);
```

If you have version 5 of C++Builder, there are two more handy functions that you can use:

```
AnsiString IncludeTrailingBackslash(
  const AnsiString S);
AnsiString ExcludeTrailingBackslash(
  const AnsiString S);
```

The names of these functions are fairly self-explanatory. The following examples will show what these functions do using the following file and path:

```
c:\windows\notepad.exe
```

`ExtractFilePath()` will return an `AnsiString` containing "c:\windows\".
`ExtractFileDir()`, on the other hand, returns "c:\windows". Notice that the difference between `ExtractFilePath()` and `ExtractFileDir()` is that `ExtractFilePath()` returns the trailing backslash in addition to the path.
`ExtractFileDrive()` returns an `AnsiString` containing "c:". Note that the drive separator (the colon) is considered part of the drive name.
Calling `ExtractFileName()` for the preceding file name will return the string "notepad.exe". `ExtractFileName()` considers the file extension part of the name.
The `ExtractFileExt()` function returns an `AnsiString` containing the string ".exe". The extension separator (the period) is returned as part of the extension.
If the file name you are working with is actually a directory name, the functions `IncludeTrailingBackslash()` and `ExcludeTrailingBackslash()` can be use to ensure that the name either has or does not have a trailing backslash. These two functions are new in C++Builder 5.

Alternate paths

There are two functions in `Sysutils` that allow you to use alternate forms of a file name's path.

```
AnsiString ExtractRelativePath(
  const AnsiString BaseName,
  const AnsiString DestName);
AnsiString ExtractShortPathName(
  const AnsiString FileName);
```

The `ExtractRelativePath()` function takes a file name in the `DestName` argument and a path name in the `BaseName` argument and returns an `AnsiString` that represents a path to `DestName` relative to `BaseName`.

Lets take a look at a couple of examples.

```
String filename = "c:\\My Documents\\"
  "Reports\\1999 Annual Report.doc";

String base1 =
  "c:\\My Documents\\Reports\\";
String relpath1 =
  ExtractRelativePath(base1, filename);

String base2 = "c:\\Windows\\System\\";
String relpath2 =
  ExtractRelativePath(base2, filename);
```

When this code executes, the `relpath1` variable will contain “1999 Annual Report.doc” because the path to `filename` and the path specified in `base1` are the same. The path is relative to itself. The value of the `relpath2` variable will be “..\..\My Documents\Reports\1999 Annual Report.doc”. To get from `base2` to `filename`, you must move up two directories and then down through the path of `filename`.
For `ExtractFilePath()` to work correctly, you must include the trailing path delimiter in the `BaseName` argument.
The `ExtractShortPathName()` function takes a file name and returns the DOS 8.3 version of that name. For this function to work correctly, the file name must actually exist on the system. Take this code, for example:

```
String filename = "c:\\My Documents\\"
  "Reports\\1999 Annual Report.doc";
String shortpath =
  ExtractShortPathName(filename);
```

On my system this code returns:

```
c:\MYDOCU~1\REPORTS\1999AN~1.DOC
```

Expanding paths

There are two functions you can use to expand paths:

```
AnsiString ExpandFileName(
  const AnsiString FileName);
AnsiString ExpandUNCFileName(
```

```
  const AnsiString FileName);
```

The `ExpandFileName()` functions takes a file name as an argument and returns an `AnsiString` containing the file name with the path equal to that of the current drive and directory on your system. For example:

```
ExpandFileName("dummy.txt");
```

Executing this code on my system resulted in:

```
C:\Writing\Reisdorph\SysUtils\dummy.txt
```

The file name used in the call to `ExpandFileName()` does not have to exist for this function to work.
The `ExpandUNCFileName()` function works like `ExpandFileName()` except that `ExpandUNCFileName()` will expand the drive name to its UNC version, if the drive is mapped. The return value will be dependent on the specific system. Take the following code:

```
ExpandUNCFileName("k:\\on_net.txt);
```

On my system `ExpandUNCFileName()` returns:

```
\\SERVER1\D\on_net.txt
```

File name case and comparison

There are two functions in `Sysutils` that allow you to change the case of a file name and one function that compares files in a case-insensitive manner:

```
AnsiString AnsiLowerCaseFileName(
  const AnsiString S);
AnsiString AnsiUpperCaseFileName(
  const AnsiString S);
int AnsiCompareFileName(
  const AnsiString S1,
  const AnsiString S2);
```

Here is an example of their use:

```
String filename =
```

```
  "c:\\My Documents\\mixed CASE.doc";

String lower =
  AnsiLowerCaseFileName(filename);
String upper =
  AnsiUpperCaseFileName(filename);

int compare1 =
  AnsiCompareFileName(filename, lower);
int compare2 =
  AnsiCompareFileName(filename, upper);
int compare3 = AnsiCompareFileName(
  filename, "d:\\junk.txt");
int compare4 =
  AnsiCompareFileName(filename,
  "c:\\My Documents\\Stuff.txt");
```

In the above example, `AnsiLowerCaseFileName()` and `AnsiUpperCaseFileName()` take `filename` as an argument and these functions do exactly what you think they do. They return `AnsiStrings`, `lower` and `upper`, that contain the following strings, respectively:

```
c:\\My Documents\\mixed CASE.doc
C:\\MY DOCUMENTS\\MIXED CASE.DOC
```

In Windows, file names are case insensitive. The function, `AnsiCompareFileName()`, uses this fact to compare two file names and return `0` if they are equal. If the first file name argument is "less than" the second argument, `AnsiCompareFileName()` returns `-1`. If the first argument is "greater", `AnsiCompareFileName()` returns `1`. `AnsiCompareFileName()` is smart enough to work correctly with Asian locales and with multi-byte character sets.

Miscellaneous file name functions

There are three more functions I want to tell you about. Those functions are:

```
AnsiString ChangeFileExt(
  const AnsiString FileName,
  const AnsiString Extension);
bool IsPathDelimiter(
  const AnsiString S, int Index);
bool FileExists(
  const AnsiString FileName
```

The `ChangeFileExt()` function takes two arguments: a file name and a file extension. It will return an `AnsiString` that contains the path and file name with the extension contained in the second argument. For example:

```
String filename =
  "c:\\My Documents\\1999 Report.doc";
String newext =
  ChangeFileExt(filename, ".rpt");
```

When this code executes, the `newext` variable contains:

```
c:\My Documents\1999 Report.rpt
```

Notice that the extension argument to `ChangeFileExt()` must contain the extension separator character.
The `IsPathDelimiter()` functions takes a file name and an index into that file name and returns `true` if the character at that index is a path delimiter character.
Using the `filename` value shown previously, `IsPathDelimiter(filename, 7)` returns `false` and `IsPathDelimiter(filename, 16)` returns `true`.
Also, be aware that this function manifests its Pascal heritage by being 1-based and not 0-based. This means that the first character in the file name is considered to be at index 1.
The last function I wanted to mention, `FileExists()` takes a file name as an argument and returns `true` if the file exists on the system and `false` if it does not. This function is convenient when you want to check for the existence of a file before attempting to open it.

Conclusion

You may be wondering how I knew that there were two new file name functions in C++Builder 5. I simply looked at the source code in SYSUTILS.PAS to see what was new. I encourage you to browse the VCL source to see what you can find. It is often easier (and more interesting) to look at the source rather than hunting around in the VCL help file.
In writing this article, I chose to include the `FileExists()` function with those dealing with file names. It could also be grouped with file control functions, which will be the topic of Part 4 of this series.

Understanding function pointers

by Brent Knigge

As you probably know, a pointer is a variable containing a memory address. Most programmers use pointers to store the address of simple data types (such as a `char`, `int`, or `long`), or to access class objects. What isn't well known is that functions reside in memory as well, and therefore it is possible to store the address of a function in a pointer.
This concept gives the programmer incredible flexibility with the way applications are constructed. Pointers to functions are used extensively throughout the Windows API and VCL. Each framework utilizes pointers to functions in different ways. This article will explain how each technology implements them.

Simple function pointers

To start with, I'll explain basic function pointers. Basic function pointers have existed since the inception of the C language. Let's say that you had a simple function declared like this:

```
int func(float arg);
```

To declare a pointer to this function you would use this syntax:

```
int (*ptrfunc)(float arg);
```

Once you have declared a function there are two things you can do with it: invoke the function, or take its address. When a function name is used in code without parentheses, the address of the function is returned. For example:

```
int func(float arg)
{
  // function body
}

ptrfunc = func;
```

The `ptrfunc` variable now points to the address of the `func()` function. `func()` can now be called in the following ways.

```
// call function normally
func(3.2);
```

```
// call function via pointer
ptrfunc(3.2);
// call function via pointer
(*ptrfunc)(3.2);
```

The last two examples achieve the same result; calling the function through a pointer. Using the syntax provided from the last example helps to better document the code so that a programmer can see that the function is in fact being invoked through a pointer.

Function pointers in the VCL

Pointers to functions are used in the VCL to provide the flexibility of calling a function that will be determined at runtime. Imagine with the previous example there was another function declared, `funcB()`, and that this function had the same argument list and return type as the `func()` function. Then `ptrfunc` could point to either function, as shown here:

```
if (choice)  // some boolean variable
  ptrfunc = func;
else
  ptrfunc = funcB;

ptrfunc(3.2);
```

Now when `ptrfunc` is invoked, it calls either `func()` or `funcB()` depending on the value of the `choice` variable. This method of programming was used in C before the advent of C++ and virtual functions. The VCL uses a similar methodology for events, allowing several pointers to point to the same function.
Programmatically this can be done in the following manner:

```
Button1->OnClick = Button1Click;
Button2->OnClick = Button1Click;
```

Notice that these two events will invoke the same function when either button is clicked. This is possible because the VCL invokes events through the use of a pointer.
Event variables are not just ordinary function pointers, though. Declaring a pointer to an event in a VCL class differs slightly from standard C++ function pointers. Events require the use of a `typedef` and the `__closure` keyword. The following example illustrates this:

```
// usually declared private
typedef int
```

```
  (__closure *ptrdef)(float arg);
// local to a function or declared public
ptrdef ptrfunc;
```

Now `ptrfunc` is a pointer to a function that can be utilized the same way as shown in the previous section.
The `__closure` keyword is used to declare a special kind of function pointer. A regular function pointer holds a 32-bit value. It simply points to a memory location. A closure, however, holds a 64-bit value. It contains the address of the function highest 32 bits, and the address of the class where the function resides in the lowest 32 bits. Closures are not part of standard C++. Borland added the `__closure` keyword to C++Builder specifically to handle the event architecture.
The most likely reason for a programmer to use a pointer to a function in C++Builder would be to create events for custom components. This is a fairly simple process but there are a few things that you should consider when declaring an event:

- The first parameter of the event should always be a pointer to the object that invoked the event (`TObject* Sender`).
- Events should have a return type of `void`.

The following code shows how a programmer might declare an event in the header file of a custom component:

```
private:
// the event type
typedef void __fastcall
  (__closure *ptrEvent)(
  TObject *Sender, float arg);
// declare a variable to hold
// the function pointer
ptrEvent FOnPtrfunc;

published:
// declare the event
__property FOnPtrfunc OnPtrfunc =
  {read=FOnPtrfunc, write=FOnPtrfunc};
```

It is imperative to check that the event pointer actually contains a valid address before attempting to call the event handler. Events are typically fired with code like this:

```
// Check for valid address
if(FOnPtrfunc)
  // if valid, invoke the function
  FOnPtrfunc(this, 3.2);
```

You won't typically use closures outside of the VCL event model but you certainly may if you wish. You must understand that you can use regular function pointers in a C++Builder program. In other words, the fact that Borland created closures for C++Builder doesn't mean you can't use regular function pointers in your applications.

Function pointers and the Windows API

The Windows API uses pointers to functions for different purposes. For example, the API often uses function pointers as parameters to other function calls.
There are two ways to declare a function when one of the arguments is a pointer to a function. One way is to use the `typedef` keyword to declare a new type and then use that type in the function declaration. The other way is to simply redefine the function pointer in the function that uses the pointer. When declaring C++ functions, argument names do not need to be specified. For example:

```
void funcD(int, ptrdef);
void funcD(int, int(__closure *)(float));
```

The following code shows several ways in which the address of a function called `func()` can be passed to the function called `funcD()`.

```
ptrdef ptr;
ptr = func;

funcD(2, ptr);
//address of funcB passed directly
funcD(2, func);
//no difference if the ampersand is used
funcD(2, &func);
```

In the last two examples, the actual name of the function is passed. Because the function name was passed without parentheses, the expression evaluates to the address of the function. Even though a function address is being passed as a parameter, the responsibility still resides with the programmer to call the function that the pointer points to. An example of how this might work is shown below.

```
void FuncD(int Size, ptrdef SomeFunc)
{
  if(Size > 1000)
    Size = SomeFunc(3.2);
}
```

The Windows API contains many functions that make use of a callback function. When defining the callback function you must be sure that the function pointer you are passing exactly matches the declaration that Windows has defined for the callback function. Consider the API function `EnumFontFamilies()`. This function enumerates the fonts in a specified font family. One of the parameters of this function is the address of a callback function. The declaration for `EnumFontFamilies()` is as follows:

```
int EnumFontFamilies(
  HDC hdc, LPCTSTR lpszFamily,
  FONTENUMPROC lpEnumFontFamProc,
  LPARAM lParam);
```

As you can see, the third parameter is used to pass the address of a callback function. The declaration of the callback function must match the specifications defined for `FONTENUMPROC`. The `FONTENUMPROC` type is declared like this:

```
int CALLBACK EnumFontFamProc(
  ENUMLOGFONT FAR *lpelf,
  NEWTEXTMETRIC FAR *lpntm,
  int FontType, LPARAM lParam);
```

The `CALLBACK` macro gives the programmer a clue that this function is used as an argument for another API function. `CALLBACK` is simply a macro that expands to `__stdcall` (Windows callback functions are always declared with this calling convention). The actual callback function could be defined as follows.

```
int CALLBACK MyEnumFont(
  ENUMLOGFONTEX *lpelf,
  NEWTEXTMETRICEX *lpntm,
  int FontType, LPARAM lParam)
{
  //Extract font information if desired
  ShowMessage(
```

```
    lpelf->elfLogFont.lfFaceName);
  return 0;
}
```

Notice how this function is a near mirror image of the `FONTENUMPROC` type. Windows API callback functions must be stand-alone functions (they cannot be class member functions). If you attempt to pass a class member function where Windows expects a callback function you will receive compiler errors.
This particular callback function will be called every time the `EnumFontFamilies()` function comes across a font with the name specified in the `LPCTSTR` parameter. An example call to `EnumFontFamilies()` might look like this:

```
int result = EnumFontFamilies(
  Canvas->Handle, "Times New Roman",
  (FONTENUMPROC)MyEnumFont, NULL);
```

As with all Windows API callback functions, the address of the callback function is cast to the specific type Windows is expecting. In this case the function pointer is cast to the `FONTENUMPROC` type.

Conclusion

Pointers to functions are often not documented in C++ books. As such, programmers often find themselves using function pointers in their code without fully understanding the concepts. This is particularly true of VCL events as they use a function pointer type not found in standard C++. The material presented in this article should remove any misconceptions that you may have had and prepare you for effectively using pointers to functions in your code.

Using the ADO access components

by Kent Reisdorph

C++Builder makes database programming fairly easy, at least compared to other C++ programming environments. One of the problems with C++Builder database programs is the fact that all database applications must use the Borland Database Engine (BDE). The BDE is a very robust way of connecting your program to its data. However, installing the BDE on your customers' machines can be frustrating. Further, the BDE files must be shipped with your program and that means more installation overhead.

Fortunately, C++Builder 5 includes a set of components that allow you to connect directly to any ActiveX Data Object (ADO) data source. Borland calls these components ADOExpress. ADOExpress ships with C++Builder Enterprise edition, but Professional edition owner can purchase ADOExpress separately (for a reasonable amount of money, I might add). An application built using the ADO components does not use the BDE. As such, you do not have to ship the BDE when deploying these applications.

This article will explain the ADO components in C++Builder 5 and how to use them to connect to ADO data sources.

Note: C++Builder 5 also comes with a set of similar components for InterBase database access, called InterBase Express. The InterBase Express components ship with the Professional and Enterprise versions of C++Builder.

What is ADO?

ADO is a Microsoft technology that adds object-oriented access to a variety of data sources. ADO is essentially a wrapper around Microsoft's OLE DB technology. ADO can be used to access FoxPro or Access databases, Microsoft SQL Server, Oracle databases, dBASE files, and so on. Basically, if you have an ODBC connection for a particular data source you can use ADO to access that data. One implication is that you can write an application that uses dBASE files without the need for the BDE.

The ADO components

The ADO components that come with C++Builder 5 allow access to any ADO data set. Those components include:

- `TADOConnection`
- `TADOCommand`
- `TADOTable`
- `TADOQuery`

- TADOStoredProc
- TADODataSet
- TRDSConnection

With the exception of TRDSConnection, each of these components has a ConnectionString property. This property is used to specify the connection parameters for a connection. Connection parameters include provider name, driver name, user name, password, and other connection-specific parameters. I will explain how to set the ConnectionString property in the next section.

Most of the ADO components also have a Connection property. When a form or data module includes a TADOConnection, component, this property can be used to hook up one of the ADO data access components (TADOQuery, TADOTable, or TADOStoredProc) to the TADOConnection. This statement requires a bit of explanation.

The ADO components can be used in one of two ways. One way is to set the ConnectionString property of a particular data access component as needed. The other way is to use a TADOConnection component and hook the data access component's Connection property to the TADOConnection. You will probably use the first method when you have a single TADOQuery, TADOTable, or TADOStoredProc on a form. You will likely use the second method if you have a number of data access components in your application and you want all of those components to share a single connection to the database.

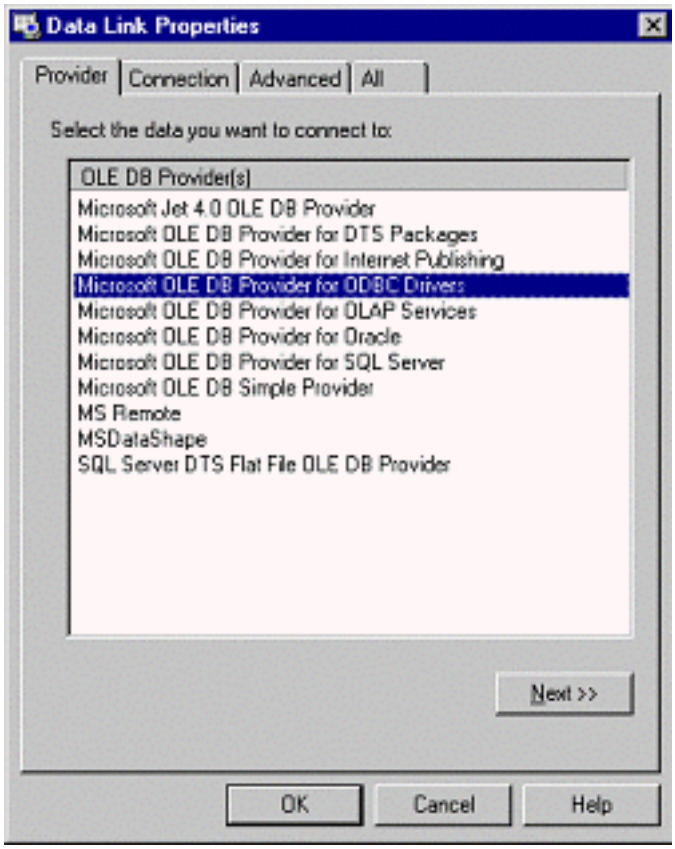
When you use this second method you set the ConnectionString property of the TADOConnection as needed, and then hook other data access components to the TADOConnection using the Connection property. In this way, you only need to set the ConnectionString property for the TADOConnection rather than for each individual data access component. The TADOConnection property works somewhat like the traditional TDatabase component. That is, it provides a central connection point that other components can use to access data.

Setting the ConnectionString property

The ConnectionString property contains all of the information that ADO needs to connect to a particular data source. Looking at the VCL help for the ConnectionString property makes it appear that setting the connection string is tedious. In reality, you can simply use the property editor provided for this property.

To invoke the property editor, double-click next to the ConnectionString property in the Object Inspector. When you do, the Data Link Properties dialog is displayed. This dialog has four pages. The first page, Provider, is used to specify the type of ADO connection you want to use (see **Figure A**). For an Access database, choose the Jet OLE DB Provider. For a dBASE database, choose the provider for ODBC drivers. For other data sources (such as Microsoft SQL Server) choose the provider for that specific data source.

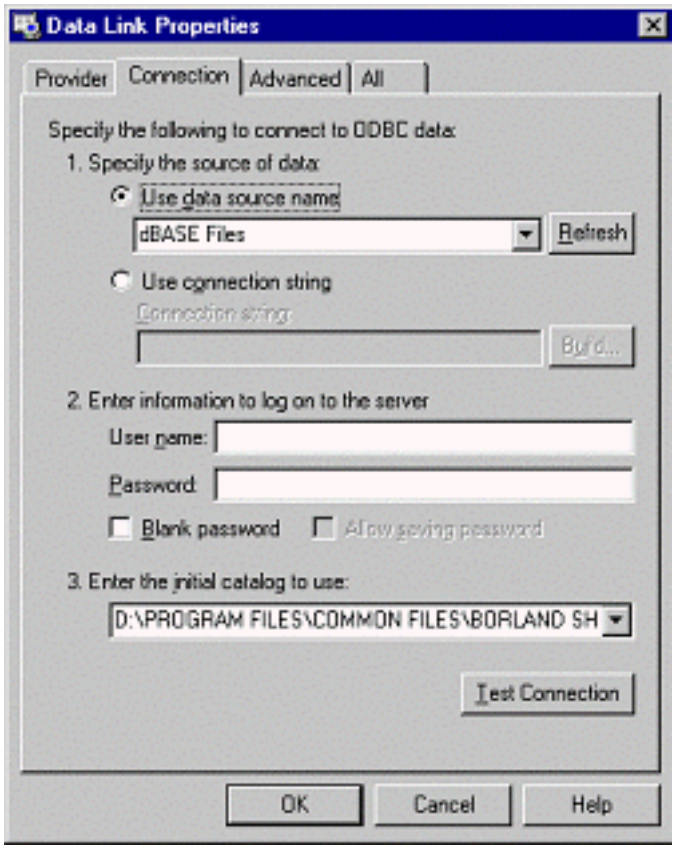
Figure A



The Provider page of the Data Link Properties dialog is where you specify the data source type.

After selecting the provider, click the Next button or click on the Connection tab. The Connection page varies a bit depending on the provider type you selected on the Provider page. Basically, the connection page allows you to select the location of the data (by server, file name, or directory), and the user name and password (if required). This page also has a button called Test Connection that you can use to verify that the parameters you specified are correct and that the ADO component can connect to the database. The connection dialog is shown in **Figure B**.

Figure B



The Connection page is where you set the connection parameters, those for a dBASE database in this case.

The Advanced page is where you can set the connection timeout and access permissions. In some cases the database server controls the access permissions. For those types of connections that section of the Advanced page will be disabled.
The All page allows you to see each connection string and to modify the value of the string if necessary. Some of the connection string values are taken from the selections made on the first two pages of the dialog. The actual connection strings shown depend on the type of provider being used.
All in all, the `ConnectionString` property editor provides a relatively easy way to set the connection strings for an ADO connection.

Using the ADO components

Once you have established a connection (with or without a `TADOConnection` component) many of the ADO components behave just like their BDE counterparts. The `TADOTable` component, for example, has a `TableName` property that works just like the `TTable` property of the same name. The `TADOQuery` and `TADOStoredProc` components also behave just like `TQuery` and `TStoredProc`. Most importantly, you can hook the ADO components to a `TDataSource` and proceed just like you do with your BDE programming.
For the most part, the `TDataSet` properties, methods, and events that you use now can also be used with the ADO components. For example, the `First()`, `Last()`, `Next()`, `Prior()`, `FieldByName()`, `ParamByName()`, etc. methods are still available for your use. Some of the

events are different for the ADO components, but most of the `TDataSet` events are still there.
The `TADODataSet` component is the most basic of the ADO components. It allows you to retrieve data using a SQL statement, but you can only perform SELECTs. To be honest, I'm not sure why you would use this component when you could just use `TADOQuery` instead.
The `TADOCommand` component is used to issue ADO commands to a data source. It is a wrapper around ADO's Command object. `TADOCommand` can return a record set. However, since it is not descended from `TDataSet` it cannot be hooked to a `TDataSource`. What you do, then, is assign the return value of the `Execute()` method to a `TADODataSet`'s `Recordset` property. For example, let's say that you had a `TADOCommand` with a `CommandText` property set to the following SQL statement:

```
select * from Cusomter
```

The code to execute the command might look like this:

```
ADODataSet1->Recordset =
  ADOCommand1->Execute();
```

The `TADODataSet` could then be hooked up to a `TDataSource` in order to display the results in data-aware components.
You can use `TADOCommand` if you have to issue commands directly to ADO, otherwise use `TADOQuery` or `TADOStoredProc`.

Conclusion

The new ADO components in C++Builder 5 offer a simple and effective way of accessing ODBC data sources without having to install the BDE when you deploy your applications. Delphi and C++Builder users have been asking for a way to connect to ODBC databases without using the BDE and ADOExpress is Borland's answer to those requests.

Adding items to the shell context menu

by Kent Reisdorph

In the article, “Using the shell context menu”, I explained how to use the shell context menu in your applications. This article will explain how to add your own items to the shell context menu. This article builds on the previous article so be sure to read that article first.

After obtaining the `IContextMenu` for a particular file, you call `QueryContextMenu()` to obtain the actual context menu. You can add items to this popup menu to provide added features to your users. There are two basic ways to go about adding to the context menu:

- Use the Windows API
- Use a VCL `TPopupMenu`

Using the Windows API entails creating new menu items and inserting them into the popup menu. Alternatively, you can build the popup menu first, and then call `QueryContextMenu()` to merge the shell’s context menu with the menu you have created. Either way you will need a `WndProc()` to handle the menu items selected.

Most of you will probably want to take advantage of the VCL’s event mechanism so that is the technique I will describe in this article.

The first step is to add a `TPopupMenu` to your form and add menu items to the popup menu. After you have added the popup menu items, create event handlers for each item’s `OnClick` event.

Next you will merge the `TPopupMenu` items with the shell’s context menu. This step requires a bit of trickery. Here’s the code:

```
MENUITEMINFO mi;
char buff[80];
for (int i=PopupMenu1->Items->Count - 1;
      i>-1;i--) {
  ZeroMemory(&mi, sizeof(mi));
  mi.dwTypeData = buff;
  mi.cch = sizeof(buff);
  mi.cbSize = 44;
  mi.fMask =
    MIIM_TYPE | MIIM_ID | MIIM_DATA;
  DWORD result = GetMenuItemInfo(
    PopupMenu1->Handle, i, true, &mi);
```

```
    if (result) {
      mi.wID = PopupMenu1->Items->
       Items[i]->Command + 100;
      InsertMenuItem(
        hMenu, 0, true, &mi);
    }
}
```

First I declare an instance of the `MENUITEMINFO` structure. Next I create a buffer to hold the menu item text. The `for` loop starts at the bottom of the menu and works backwards. I start at the bottom because each menu item in the `TPopupMenu` is inserted at the top of the shell's context menu. In the `for` loop I fill in the required values for the `MENUITEMINFO` structure. Take special note of this line:

```
mi.cbSize = 44;
```

Normally you would use `sizeof(mi)` to assign the size of the structure to the `cbSize` member. For whatever reason, this causes the call to `GetMenuItemInfo()` to fail on some operating systems. By hard-coding the size to 44 bytes, the call will work on all operating systems. By the way, the VCL uses this technique as well. Normally I am reluctant to hard-code a value like this but it's the only way to get the call to work in this case.

After obtaining the `MENUITEMINFO` for a particular menu item, I use this code:

```
mi.wID = PopupMenu1->Items->
  Items[i]->Command + 100;
```

The VCL automatically assigns menu IDs to the items in a `TPopupMenu`. The shell context menu commands have values less than `100`. By adding `100` to the menu ID generated by the VCL, I can determine whether a shell context item was clicked, or one of the items from the `TPopupMenu`. Finally, I insert each menu item into the context menu:

```
InsertMenuItem(hMenu, 0, true, &mi);
```

When `TrackPopupMenu()` returns (see the previous article), I check the return value and take appropriate action based on whether one of my menu items was clicked, or whether a shell item was clicked. The code looks like this:

```
int Cmd = TrackPopupMenu(hMenu,
  TPM_LEFTALIGN | TPM_LEFTBUTTON |
  TPM_RIGHTBUTTON | TPM_RETURNCMD,
  pt.x, pt.y, 0, Handle, 0);
```

```
if (Cmd < 100 && Cmd != 0) {
  CI.lpVerb = MAKEINTRESOURCE(Cmd - 1);
  CI.lpParameters = "";
  CI.lpDirectory = "";
  CI.nShow = SW_SHOWNORMAL;
  CM->InvokeCommand(&CI);
}
else
  for (int i=0;i<PopupMenu1->Items->Count;i++) {
    TMenuItem* menu =
      PopupMenu1->Items->Items[i];
    // Call its OnClick handler.
    if (menu->Command == Cmd - 100)
      menu->OnClick(this);
  }

DestroyMenu(hMenu);
```

If the item clicked is one of the shell's items I call `InvokeCommand()` to execute the item. If the item clicked is one of my `TPopupMenu` items, I find the item in the `TPopupMenu` and then call that item's `OnClick` event handler. This allows me to use the VCL's event handling mechanism to respond to the items I place on the context menu. This is much easier than using the API to add menu items, and fits nicely with the VCL model. See **Listing A** in the previous article to see the code in context.

Note that this allows you to add menu items to the shell context menu in your own programs. It does not, however, allow you to add items to the context menu shown by Explorer. Adding items to Explorer's context menu involves writing a shell extension, and that is beyond the scope of this article.

Error and exception handling strategies

by Mark Cashman

Programmers don't like errors. We'd like to write our code without worrying about all of the terrible things that can happen: bad parameters, uninitialized variables and pointers, division by zero, resource limits, and so on. These are things that can go wrong and can quickly dominate our code, overwhelming the clear expression of the problem and complicating both development and maintenance.

Nevertheless, errors can happen, and what goes wrong at run-time is going to crash your system, cripple your program, damage your database, or, at the very least, raise your support costs.

For most of us, handling errors is a matter of prevention put in place after realizing an error can occur. A large percentage of the deeply nested `if` statements in the code are dedicated to preventing errors from occurring. Some are placed in advance, and some are placed after testing reveals a weak spot. But there are also `if` statements which represent true alternative paths through the code, required by the logic and design. Distinguishing between that type of code and that which protects against errors can be difficult, and can make maintenance more difficult. For instance:

```
if (myIndexDef != NULL)
{
  for (int I=0;I<myIndexDef->Count;I++)
  {
    delete (TIndexDef *)
      myIndexDef->Items[I];
  };
  myIndexDef->Clear();
};
```

Especially troublesome in well-structured programs is dealing with errors that originate in one class or method that can have an effect on another. Programmers have evolved a variety of strategies including status variables and return codes. But just like the `if` statements, these obscure a program's structure. Further, the require every participant in calls down through layers of methods to understand and participate in error handling, even if they are unaffected by the error. For instance:

```
if (DoSomething(Parameter) < 0)
  return -1;
// Pass on error from DoSomething
```

Inevitably some error information is lost, if only information about where the error happened, and what it meant when it happened there. For instance, in the above example, `DoSomething()` may return a

variety of negative error codes, but the routine which detects the error converts them all to `-1`, which gets passed to the next level up, and which, consequently, knows less about the error than may be necessary to provide a useful error report.

Exceptions and errors: differences and similarities

Ideally, errors and decisions could be separated into completely different types of code, each with their own characteristics, each grouped together. To some extent, that is the purpose of C++ exception handling. Observe the following conversion of our first example to an exception-based framework:

```
try
{
  for (int I=0;I<myIndexDef->Count;I++)
  {
    delete (TIndexDef *)
      myIndexDef->Items[I];
  };
  myIndexDef->Clear();
}
// This exception occurs when
// a pointer is invalid
catch (EAccessViolation &AccessViolation)
{
  // Don't do any of the above if an
  // exception occurs because myIndexDef
  // is NULL
};
```

The `try` keyword introduces a block of code that is subject to exception handling. The closing `catch` identifies the class of exception that is to be caught. Note that exceptions are rooted in the `Exception` class. There are many built-in classes of exception and you can also create your own. The `catch` clause will `catch` every exception of the stated class, and any descendants of that class, so when you are catching multiple, related exception classes, make sure the most specific one is first. For instance, when catching database errors, `EDBEngineError` is a descendant of `EDatabaseError`. Therefore, if you want different handling for each of those exception classes, you must specify `EDBEngineError` first, then `EDatabaseError`. For example:

```
try
{
  Something();
}
catch (EDBEngineError &DBEngineError)
```

```
{
  HandleEngineError(DBEngineError);
}
catch (EDatabaseError &OtherDbError)
{
  HandleOtherDatabaseError(OtherDbError);
};
```

As you can see, the `try`/`catch` localizes the handling of exceptions to the end of the code concerned. A special type of statement, easily identifiable, is used to handle exceptions. All in all, this is an improvement over the use of `if` statements to prevent errors, at least in terms of providing clear code. I should also point out that providing three dots in a `catch` statement would catch all exceptions, regardless of the type. For example:

```
try
{
  Something();
}
catch (...)
{
  // all exception are caught
}
```

An exception is, well... an exceptional event. Generally speaking, exceptions are used to handle operating system and API level errors. But they can be useful for much more.

Nevertheless, there is still a place for error detection before the fact. Indeed, it can work with exception handling to produce a much more robust and comprehensive error handling system.

Local error handling: exceptions and memory management

One of the reasons for catching exceptions in C++ is to manage memory, mostly to make sure memory is released in the event of an error. Fortunately, C++ Builder offers an extension to the exception handling system, which makes this even easier.

Derived from Delphi, the `__finally` statement allows you to create an exception handling block where the logic in the `__finally` clause is executed whether or not an exception occurs, and regardless of where in the `try` block the exception happens. A typical pattern might be as follows:

```
TStringList  *StringList = new TStringList;
```

```
try
{
  DoSomething();
  DoSomethingElse();
}
__finally
{
  delete StringList;
};
```

Regardless of what happens in the `try` block, the `__finally` block is always executed. The result in this case is that the string list is always deleted and its memory is always reclaimed.

One apparent disadvantage of the `try`/`__finally` combination is that `__finally` cannot be mixed with `catch`. However, this is not actually a disadvantage, since it requires separation of memory management in the face of an error from handling the error. Extending the above example to handle errors as well as release memory is simple:

```
try
{
  try
  {
    DoSomething();
    DoSomethingElse();
  }
  __finally
  {
    delete StringList;
  };
}
catch (EDBEngineError &DBEngineError)
{
  HandleEngineError(DBEngineError);
}
catch (EDatabaseError &OtherDbError)
{
  HandleOtherDatabaseError(OtherDbError);
};
```

In the event of an exception, the `__finally` clause performs its logic and then passes the exception on. In this way the exception is capable of being caught by the outer `try` block, and the memory is still freed.

Module and system-level error handling

Most programmers who use exception handling use it at the local level; preventing errors from propagating out of a method or class, or preventing the premature termination of a loop when one iteration's operation fails.

But every program needs a more cohesive approach to errors than simple local error trapping. The typical program consists of several levels, each of which has its own needs for error handing and for passing error information to higher layers. Those layers are:

1. Hardware and operating system
2. VCL and third party components
3. Generic components developed at your installation
4. Application-specific components developed at your installation
5. Application data modules
6. Application user interfaces

Of these levels, only level 6 can safely and reliably decide how to report errors and exceptions to the users. Therefore, the key focus of error management must be to promote errors up to level 6, or to at least up to where they can be logged to databases or files at level 5. However, it is best to let level 6 decide to use the resources of level 5 for logging as needed.

From a support perspective, errors must be managed so that when a user reports an error to support, the support personnel can quickly determine the cause, location, and context of the error. This can be managed by a simple strategy.

It is important to remember that the meaning of an error at level 3, the first level over which you have any control, is quite different from the meaning of the error at level 6. At level 3, the error may be `EListError`, but at level 4 it is "Attempt to insert a service type in the service type list failed", at level 5 it is "Update of Package record failed", and at level 6 it is "Unable to ship package." To the user, the level 6 perspective is the most relevant, and it is usually the message which is displayed. The problem is that the support person needs to know more, and the programmer needs to know even more to fix the problem.

The solution to this problem is to cascade the exception up through the levels, with each level adding its context to the front of the message. For instance, if all the levels were in one method, they would look something like this:

```
try // Level 6
{
```

```
  try // Level 5
  {
    try // Level 4
    {
      try // Level 3
      {
        DoSomethingWithList();
      }
      catch (Exception &VCLException)
      {
        throw new Exception("Unable to "
          DoSomethingWithList due to: "+
          VCLException.Message);
      };
    }
    catch (Exception &VCLException)
    {
      throw new Exception("Attempt to "
        "insert a service type in the "
        "service type list failed due "
        "to: " + VCLException.Message);
    };
  }
  catch (Exception &VCLException)
  {
    throw new Exception("Update of "
      "Package record failed due to: "+
      VCLException.Message);
  };
}
catch (Exception &VCLException)
{
  Display("Unable to ship package due "
    "to: " + VCLException.Message);
};
```

Note that the final level actually displays the message. The resulting message looks something like:

Unable to ship package due to: Update of Package record failed due to: Attempt to insert a service type in the service type list failed due to: Unable to DoSomethingWithList due to: EListOutOfBounds (-1)

The first part of the message is what is most important to the user, but all of the information needed by support and the programmer is present as well.

Your own exception classes: do you need them?

The base class `Exception` and the specialized VCL classes derived from `Exception` are the real workhorses of exception management. You may not need anything more if your only intent is to pass messages up to level 6 for display and logging.

On the other hand, if you need to pass specialized information in your exception, or if your intent is to offer higher levels an opportunity to recover from specific exceptions, you may wish to create your own exception classes descending from `Exception`. Keep in mind the following factors when considering this:

You cannot predict the nature of higher levels or their exception handling policy. Your exceptions may not reach the level for which they are intended without being changed into a generic exception, in which case your special type and message information may be lost.

Levels above the one that calls the method where the exception originates often will not want to know about the peculiarities of your specific exception class. Make sure your special exception class can be informatively handled as an `Exception` type.

Exceptions from threads

As with data modules and components, threads should not directly report their own error or exception conditions, because of their ignorance of the context of the error. Only the main thread is likely to know what the error means, whether or not and how it should be displayed. Therefore, while threads should catch their exceptions, they should only do so in order to report them to the main thread.

Basically, the following is what the thread should do:

```
try
{
  Something();
}
catch (Exception &VCLException)
{
  ReportToMainThread(Exception);
};
```

The report function should act as follows:

```
void __fastcall
ThreadClass::ReportToMainThread(
  Exception &theException)
{
  // A thread variable
  ExceptionReference = theException;
  Synchronize(
    ProvideMainThreadWithException);
};
```

The `ProvideMainThreadWithException()` function might throw an `Exception` reference, which should now be safe because it is in the context of the main thread.

Conclusion

Exceptions offer an excellent opportunity to structure error handling in a way useful to users, support, and programmers. In addition, they make it possible to shift error handling out of mainline code and into less obtrusive areas of your program. They require some forethought and consistency, but the rewards are obvious.

Hidden treasures of Sysutils, Part 2

by Mark G. Wiseman

The `Sysutils` namespace of the VCL provides several functions and global variables that you may find useful when working with times, dates, numbers, currencies, files, file names, and strings. In Part 1 of this series, I introduced you to variables and functions that worked with dates and times. In this article, you will learn about the variables and functions that can help you when dealing with numbers and currency.

As I mentioned in Part 1, these variables and functions are not really hidden. Most of them can be found in the online help—if you know where to look.

Locales

With the exception of `HexDisplayPrefix` (explained later), the global variables discussed in this article are initialized with locale settings obtained from Windows. The VCL function, `GetFormatSettings()` initializes these variables at application startup.

The VCL also calls `GetFormatSettings()` whenever your program receives a `WM_WININICHANGE` message, so be careful when working with the global variables because they can change at any time.

The code examples in this article and the companion program are based on the United States locale settings, the settings specific to my system.

Numbers and currency

I am sure you know that the VCL was originally written in Object Pascal for Delphi. The `Sysutils` functions and variables that work with numbers and currency were originally written to work with Delphi data types. With the exception of two data types, the Delphi data types all correspond with C++ data types and are represented in C++ with the use of `typedef`. For example, the `LongInt` data type from Delphi is defined in the VCL header file `sysmac.h` as

```
typedef signed long Longint;
```

The two exceptions to the use of `typdef` are the Delphi data types `Currency` and `Comp`. These two data types are represented in C++ as classes with the names, as I am sure you've guessed, of `Currency` and `Comp`.

`Currency` is obviously a class that represents currency values. The actual currency value is stored within the class as an `__int64`. Like `AnsiString`, `Currency` is a full-featured class that you should seriously consider using.

The `Comp` class represents a 64-bit integer. The `Comp` class is included in the VCL only for backward compatibility and is of no use to C++ programmers. I will not discuss this class or the functions in `Sysutils` that work with it.

Numbers

In `Sysutils`, there are functions and variables for working with decimal and hexadecimal numbers.

Number variables

There are two variables that are used when formatting numbers. Here are their declarations:

```
char ThousandSeparator;
char DecimalSeparator;
```

These variables are set by the VCL function `GetFormatSettings()`.

On my system, `ThousandSeparator` contains ‘,’ and `DecimalSeparator` contains ‘.’.

Number functions

There are several functions in `Sysutils` that deal with numbers. Some of those functions are:

```
AnsiString IntToStr(int Value);
AnsiString IntToStr(__int64 Value);
AnsiString FloatToStr(Extended Value);
AnsiString FloatToStrF(
  Extended Value, TFloatFormat Format,
  int Precision, int Digits);
AnsiString FormatFloat(
  const AnsiString Format,
  Extended Value);
int StrToInt(const AnsiString S);
__int64 StrToInt64(const AnsiString S);
int StrToIntDef(
  const AnsiString S, int Default);
```

```
__int64 StrToInt64Def(
  const AnsiString S, __int64 Default);
Extended StrToFloat(const AnsiString S);
```

The `IntToStr()` function is overloaded to accept either an `int` or an `__int64` and returns an unformatted `AnsiString`. For example:

```
int i = 1234;
AnsiString s = IntToStr(i);
__int64 i64 = 1234567890;
AnsiString s64 = IntToStr(i64);
```

The `FloatToStr()` function takes an `Extended` data type as an argument. The `Extended` data type is a native Delphi type that corresponds to a `long double` in C++:

```
typedef long double Extended;
```

Since `Extended` is really just a `long double`, you can pass a `float`, a `double` or a `long double` into `FloatToStr()`. Here are some examples:

```
float f = 1234.5;
String sf = FloatToStr(f);
double d = 1234.5678;
String sd = FloatToStr(d);
long double ld = 1234567890.12345;
String sld = FloatToStr(ld);
```

The only formatting done by `FloatToStr()` is to place the character contained in the `DecimalSeparator` variable in the correct location.

The `FloatToStrF()` function is similar to `FloatToStr()`, but it gives you more control over the formatting of the returned `AnsiString`. In addition to taking an `Extended` as an argument, `FloatToStrF()` also takes as arguments a `TFloatFormat`, an `int` specifying precision, and an `int` that specifies the number of digits in the returned `AnsiString`. `TFloatFormat` is an `enum` that tells `FloatToStrF()` how to format the number. The acceptable values for `TFloatFormat` are `ffGeneral`, `ffExponent`, `ffFixed`, `ffNumber`, and `ffCurrency`. The online help for the VCL contains a detailed explanation of each of these `enums`. Just look under “TFloatFormat” in the help index.

If you need even more control over formatting than that provided by `FloatToStrF()`, then you need to use the `FormatFloat()` function. This function accepts an `AnsiString` that specifies the formatting used when converting an `Extended` value to an `AnsiString`. You can specify different

formats for positive numbers, negative numbers and numbers that are equal to zero. Here is an example:

```
String fs = "#,##0.00;(#,##0.00);'-zero-'";
double n = 12736.987;
sn = FormatFloat(fs, n);
// sn = 12,736.99
n = 0;
sn = FormatFloat(fs, n);
// sn = -zero-
n = -27456;
sn = FormatFloat(fs, n);
// sn = (27,456.00)
```

Look in the online help for `FormatFloat()` to learn about all the different format specifications you can use with this powerful function.

The two functions `StrToInt()` and `StrToInt64()` both take an `AnsiString` as an argument and return an `int` or `__int64` respectively. If the `AnsiString` doesn't represent a valid number, an exception will be thrown. Here are some examples:

```
String s = "87456";
int i = StrToInt(s);
__int64 i64 = StrToInt64(s);
```

You may find `StrToIntDef()` and `StrToInt64Def()` more useful. These two functions also take an `AnsiString` as an argument and return either an `int` or an `__int64` just as the `StrToInt()` and `StrToInt64()` functions do. However, they also take a second argument that specifies a default value to be returned if the `AnsiString` does not contain an acceptable value. No exception is thrown if the string value is invalid. The following examples both return the "default":

```
String s = "Not a number";
int i = StrToIntDef(s, 5213);
__int64 i64 = StrToInt64Def(s, 789426);
// i = 5213
// i64 = 789426
```

The `StrToFloat()` function takes an `AnsiString` and returns an `Extended`. (Remember, you can use a `long double` in place of `Extended`.) If the `AnsiString` argument contains a decimal separator, it must match the separator stored in the `DecemalSeparator` variable. Unfortunately, the `AnsiString` may not contain any other non-numeric characters, not even the character in the `ThousandSeparator` variable. If the `AnsiString` does not meet these requirements, an exception will be thrown.

Hexadecimal variable

There is one global variable related to hexadecimal numbers:

```
AnsiString HexDisplayPrefix;
```

Unlike the other number-related variables in `Sysutils`, `HexDisplayPrefix` is not set in the `GetFormatSettings()` function and is not dependent on the Windows locale. The VCL sets `HexDisplayPrefix` to "0x" at application startup.

Hexadecimal functions

There is one overloaded function that works with hexadecimal numbers:

```
AnsiString IntToHex(
  int Value, int Digits);
AnsiString IntToHex(
  __int64 Value, int Digits);
```

The `IntToHex()` function is overloaded to accept either an `int` or an `__int64` and returns an `AnsiString` formatted as a hexadecimal number. Surprisingly, `IntToHex` does not attach the prefix specified in the `HexDisplayPrefix` variable. If you want to use `HexDisplayPrefix`, you have to write additional code. For example:

```
int i = 1024;
String s = IntToHex(i, 8);
// s = 00000400 Huh?
int i64 = 987654321;
String s64 = IntToHex(i64, 8);
// s64 = 3ADE68B1
String s =
  HexDisplayPrefix + IntToHex(99, 4);
// s = 0x0063 Ahh!
```

Currency

As I mentioned, the Delphi data type `Currency` is represented in C++Builder as a class named `Currency`. The `Currency` class defines a complete set of arithmetic and comparison operators. The class also has stream insertion and extraction operators and conversion operators for `AnsiString`, `double`, and `int`. The class is defined in the `syscurr.h` header file.

Internally, `Currency` stores the actual value as an `__int64`, so you can use it to represent a very large amount of money.

Currency variables

There are four currency related variables:

```
AnsiString CurrencyString;
Byte CurrencyFormat;
Byte NegCurrFormat;
Byte CurrencyDecimals;
```

All of these variables are set by the `GetFormatSettings()` function using locale information provided by Windows.

The `CurrencyString` variable contains the currency symbol used when formatting currency. On my system, `CurrencyString` is equal to "$". The `CurrencyFormat` and `NegCurrFormat` variables contain numbers that represents a currency format. Look at the VCL online help for an explanation of these numbers.

The number of digits to the right of the decimal point is contained in the `CurrencyDecimals` variable.

Currency functions

The functions `Sysutils` provides for working with `Currency` are:

```
AnsiString CurrToStr(Currency Value);
AnsiString CurrToStrF(Currency Value,
  TFloatFormat Format, int Digits);
AnsiString FormatCurr(
  const AnsiString Format,
  Currency Value);
Currency StrToCurr(const AnsiString S);
```

The `CurrToStr()` function takes a `Currency` value and returns an unformatted `AnsiString`. Since the `Currency` class provides conversion operators for `int` and `double`, you can also use these data types as input to `CurrToStr()`. Here are some examples:

```
Currency c = 15128.62;
```

```
String s = CurrToStr(c);
String s = CurrToStr(15128.62);
String s = CurrToStr(15128);
```

The `CurrToStrF()` returns a formatted `AnsiString`. In addition to the `Currency` argument, `CurrToStrF()` also takes a `TFloatFormat` argument and an `int` argument that specifies the number of digits used to format the `AnsiString`. I don't really understand why Borland decided to require the `TFloatFormat` argument. You need to set this argument to `ffCurrency` to get the format you would expect. For example:

```
Currency c = 15128.6;
String s = CurrToStrF(c, ffCurrency, 0);
// s = $15,129
s = CurrToStrF(c, ffCurrency, 2);
// s = $15,128.60
```

The `FormatCurr()` function is nearly identical to the `FormatFloat()` function except that it takes a `Currency` argument instead of an `Extended` argument:

```
Currency c = 15128.6;
String s = FormatCurr(
  "#,##0.00;(#,##0.00)", c);
// s = 15,128.60
s = FormatCurr(
  "$#,##0.00;($#,##0.00)", c);
// s = $15,128.60
```

Notice that `FormatCurr()` will not automatically use the value from `CurrencyString`.

The `CurrToStr()` function takes an `AnsiString` argument and returns a `Currency` object. The `AnsiString` cannot contain a currency symbol or any thousand separators and the decimal separator must match the value in the `DecimalSeparator` variable. If the `AnsiString` is invalid, `CurrToStr()` throws an exception.

Conclusion

In Part 1 of this series, I explained the functions and variables that `Sysutils` provides for working with dates and times. In this article, I have discussed numbers and currency. In Part 3, I will tell you about functions that work with file names.

In the meantime, I would encourage you to examine the header file `sysutils.hpp` and the Pascal

source file `sysutils.pas` to see what hidden treasures you can find.

Using the shell context menu

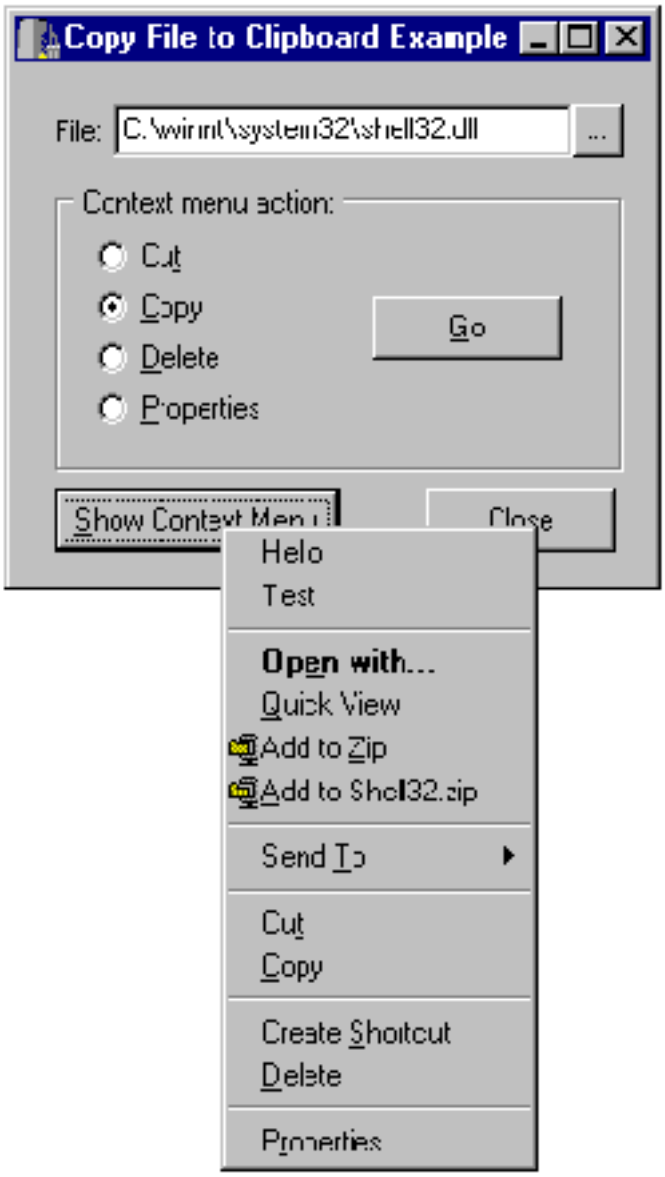
by Kent Reisdorph

As you know, when you right-click on an item in Windows Explorer, the shell context menu is displayed. You can implement the shell context menu in your own applications with a minimum amount of effort. This has implications beyond just displaying the context menu for a file, though. Through the shell context menu you can:

- Display the context menu and have the shell carry out the appropriate action for the item clicked.
- Silently cut or copy files to the clipboard.
- Allow the user to delete a file (with confirmation) without showing the context menu.
- Add your own menu items to the shell context menu.

This article will explain how to accomplish these tasks. (Adding items to the context menu is explained in the article, “Adding items to the shell context menu.”) **Figure A** shows the example program for this article displaying the shell context menu for Windows’ SHELL32.DLL. Note that two additional items appear at the top of the shell context menu.

Figure A



You can display the shell context menu in your applications and even add your own menu items.

The key to these tasks is the `IContextMenu` interface. The code presented in this article assumes that you are working with a single file in the file system. The concepts presented, however, can be adapted to allow access to the shell context menu for folders or other shell items. This article builds on past articles regarding the shell namespace, but it is not necessary for you to have read those articles to understand the concepts presented here. You must include SHELLAPI.H in order to use the objects explained in this article.

Note to C++Builder 5 Users: You must define `NO_WIN32_LEAN_AND_MEAN` if you include SHELLAPI.H. You will normally do this in your main form's unit before the include for VCL.H (see **Listing A**).

IContextMenu

The `IContextMenu` interface (and the `IContextMenu2` extension to this interface) represents the shell's context menu. `IContextMenu` is a fairly simple interface as it only contains two methods of interest. Those methods are:

```
QueryContextMenu()
InvokeCommand()
```

The `QueryContextMenu()` method obtains the shell context menu for a particular item in the shell namespace. In order to obtain the context menu for a particular shell item you must have an `IShellFolder` interface for the item's parent folder, and a pidl (pointer to an item ID list) for the item itself. I will explain how to get these items in the next section.

The `InvokeCommand()` method invokes a particular shell context menu command. If the Copy item on the shell context menu is clicked, for example, the item will be copied to the clipboard. The copied item can then be pasted in Explorer or any other application that supports pasting of shell items.

One other method, `GetCommandString()`, allows you to get the language-independent verb that corresponds to a particular shell menu item. It is not typically necessary to use `GetCommandString()`, though, so I will not explain that method in this article.

Obtaining an IContextMenu interface

Obtaining the `IContextMenu` interface for a particular shell item requires that you first obtain an `IShellFolder` interface for the item's parent. Once you have the `IShellFolder` for the parent object you must obtain a pidl for the item itself. I have explained pidls in previous articles, but the short version is that a pidl is a binary object that represents an item in the shell namespace, whether that item be a file, a folder, an item in the Control Panel, and so on.

Let's say that you have a path to a file. The first step is to separate the file and its path into separate parts:

```
String FName = "c:\\Data\\Stuff.dat";
String FilePath =
  ExtractFilePath(FName);
String FileName =
  ExtractFileName(FName);
```

Now you can get the `IShellFolder` for the folder in which the file is located. This code may be somewhat intimidating, so I'll break it into separate parts. (I have removed error-checking code to simplify the code shown. Naturally you should perform error checking at the various steps to ensure that the pointers you get from the shell are valid before attempting to use them.) First you must obtain a pointer to the Desktop's `IShellFolder` interface. The Desktop is the root of the shell namespace, and

everything begins with the Desktop:

```
LPSHELLFOLDER DesktopFolder;
SHGetDesktopFolder(&DesktopFolder);
```

Now that you have a pointer to the Desktop you can use it to obtain a pidl for the path that contains the file:

```
wchar_t Path[MAX_PATH];
LPITEMIDLIST ParentPidl;
DWORD Eaten;
StringToWideChar(
  FilePath, Path, MAX_PATH);
DesktopFolder->ParseDisplayName(Handle,
  0, Path, &Eaten, &ParentPidl, 0);
```

At this point the `ParentPidl` variable contains a pidl for the folder. Next you get the `IShellFolder` that represents the folder that contains the file, using the pidl obtained in the previous step:

```
LPSHELLFOLDER ParentFolder;
DesktopFolder->BindToObject(
  ParentPidl, 0, IID_IShellFolder,
  (void**)&ParentFolder);
```

Now you have an `IShellFolder` for the folder that contains the file, but there are a few steps yet to go. The next step is to obtain a pidl for the file itself. This is done using the file name and the parent folder's `IShellFolder`:

```
LPITEMIDLIST Pidl;
StringToWideChar(
  FileName, Path, MAX_PATH);
ParentFolder->ParseDisplayName(
  Handle, 0, Path, &Eaten, &Pidl, 0);
```

Finally, you get to the important part: obtaining an `IContextMenu` interface for the file. This is done by calling the `GetUIObjectOf()` method of `IShellFolder`. You pass the pidl for the file and a pointer that will receive the `IContextMenu` interface:

```
LPCONTEXTMENU CM;
ParentFolder->GetUIObjectOf(
  Handle, 1, (LPCITEMIDLIST*)&Pidl,
```

```
  IID_IContextMenu, 0, (void**)&CM);
```

If the call to `GetUIObjectOf()` succeeds, the `CM` variable will contain a pointer to the `IContextMenu` for the file.

The above steps may seem complex, but you can simply use this code in any of your applications that need access to the shell's context menu. As I have said, the code presented here assumes you are obtaining the context menu for a file in the file system. However, you can obtain the context menu for any item in the shell namespace provided that you have an `IShellFolder` for the item's parent, and a pidl representing the item.

I will explain the different things you can do with the `IContextMenu` interface shortly. Before I do, though, I need to explain a critical element of using the shell context menu.

OleInitialize

On some operating systems it is necessary to call `OleInitialize()` to ensure that the context menu actions will work properly. My tests indicate that calling `OleInitialize()` is not always necessary on Windows 98, but is necessary on Windows 95, Windows NT, and Windows 2000. You don't need to be concerned about which of the operating systems have this requirement as calling `OleInitialize()` is safe in any case.

In writing this article I noticed something interesting (and frustrating, too, I might add). In my test application I called `OleInitialize()` just prior to the code shown in the previous section. The Cut and Copy commands for the context menu simply were not working. I examined my `IShellFolder()`, my pidls, and everything else I could think of but to no avail. Windows didn't report any error, but the clipboard operations just did not work. Having tried everything else I could think of, I finally moved the call to `OleInitialize()` to the form's `OnCreate` event handler. For whatever reason, this worked, and the Cut and Copy menu items began working. I can't explain why, but I thought you should be aware of the problems I encountered with `OleInitialize()`.

Every call to `OleInitialize()` must have a matching call to `OleUninitialize()`. Typically, you will place this call in your form's `OnDestroy` event handler, or in the form's destructor.

Silently copying a file to the clipboard

You may simply want to copy a file (or any shell item) to the clipboard so that your users can then paste that file in Explorer or any other application that supports pasting of shell items. `IContextMenu` can do this silently, without ever displaying the context menu. I will discuss the steps required to copy a file to the clipboard, but you will later see that this same technique will allow you to do other things with the

file (displaying the Properties dialog for the file, for example).

The first step in this process is to declare an instance of the `CMINVOKECOMMANDINFO` structure and set a few of that structure's members:

```
CMINVOKECOMMANDINFO CI;
ZeroMemory(&CI, sizeof(CI));
CI.cbSize = sizeof(CMINVOKECOMMANDINFO);
CI.hwnd = Handle;
```

Next comes the important part. Each item on the shell context menu has an associated verb. This verb is language-independent. The verb used to copy a file to the clipboard, for example, is "open". You set the verb by assigning a value to the `lpVerb` member of the structure:

```
CI.lpVerb = "copy";
```

At this point the structure is set up and you are ready to invoke the command. This is done with the `InvokeCommand()` method:

```
CM->InvokeCommand(&CI);
```

If `InvokeCommand()` is successful, the return value will be `0`. In this example the file will be copied to the clipboard and can be pasted in Explorer.

As I said earlier, you can do more than just copy a file using the context menu. **Table A** shows a list of verbs used with `IContextMenu`, and their associated menu text. Note that not all shell context menu items have an associated verb. Note also that the items listed in **Table A** do not appear for all shell items and, in some cases, do not appear on some operating systems. I will explain this in more detail in a later section.

Table A: Common verbs used with IContextMenu.

Verb	**Menu Item Text**
openas	Open
cut	Cut
copy	Copy

paste	Paste
link	Create Shortcut
delete	Delete
properties	Properties
Explore	Explore
find	Find
COMPRESS	Compress
UNCOMPRESS	Uncompress

Using these verbs you can invoke the commands associated with a particular verb. The most common use will be to delete, open, cut, copy, or paste files, and to show the Properties dialog for a file.

Displaying the context menu

In some cases you may want to display the context menu for a shell item in order to allow the user to carry out an action. Once you have the `IContextMenu` interface for a file, you can easily show the shell context menu. The following sections explain how to obtain the menu, how to display it, and how to take action when a menu item is clicked.

Creating the menu

The first step in this process is to call `QueryContextMenu()` to associate the shell's context menu with a menu handle:

```
HMENU hMenu = CreatePopupMenu();
DWORD Flags = CMF_EXPLORE;
CM->QueryContextMenu(
  hMenu, 0, 1, 0x7FFF, Flags);
```

First, I create a popup menu by calling the `CreatePopupMenu()` function. Next I set the flags I want for the context menu. I will explain the flags in just a bit. You may have noticed that one line in the previous code snippet is commented out. I will get to that shortly.

Next I call `QueryContextMenu()` to get the context menu. I pass the menu handle of the popup menu in the first parameter. In this case I am creating a new empty popup menu. I could, however, pass the handle for an already-existing menu. In that case, Windows will merge the shell context menu with the items in the already-existing menu. This would allow you to add items to the shell context menu (see the next article for details). I will explain that in a later section. I pass `0` for the second parameter and `0x7FF` for the third parameter. These two parameters indicate the range of allowed menu ID values for the menu items. In most cases you can just pass the values shown here and not give it another thought.

The final parameter is used to specify the flags that should be used. In this case I am only using the `CMF_EXPLORE` flag. This flag tells Windows to return the same context menu that Explorer shows. For a complete list of the flags available, see `IContextMenu` in the Win32 API help.

If `QueryContextMenu()` is successful, the menu represented by the `hMenu` variable will contain the shell context menu.

Showing the menu

Showing the context menu is trivial. Here is the code:

```
TPoint pt;
GetCursorPos(&pt);
int Cmd = TrackPopupMenu(hMenu,
  TPM_LEFTALIGN | TPM_LEFTBUTTON |
  TPM_RIGHTBUTTON | TPM_RETURNCMD,
  pt.x, pt.y, 0, Handle, 0);
```

First I get the location of the cursor by calling the `GetCursorPos()` function. Next I call `TrackPopupMenu()` to display the menu. I won't go over every parameter to `TrackPopupMenu()`, but you can see the Win32 API help for further explanation.

Invoking a command

`TrackPopupMenu()` is a synchronous call. That is, it will not return until the popup menu has been dismissed. The Win32 API help for `TrackPopupMenu()` says that it returns a `BOOL`, and that the return value will be `0` if the call fails, or a non-zero value if the call succeeds. What the help does not say, though, is that the return value is the command ID of the menu item that was clicked. You can use the return value of `TrackPopupMenu()` to invoke a shell command. Here is the code:

```
if (Cmd) {
  CI.lpVerb = MAKEINTRESOURCE(Cmd - 1);
  CI.lpParameters = "";
```

```
  CI.lpDirectory = "";
  CI.nShow = SW_SHOWNORMAL;
  CM->InvokeCommand(&CI);
}
```

The key lines are the line that set the `lpVerb` member of the `CMINVOKECOMMANDINFO` structure and the line that calls `InvokeCommand()`. When you call `InvokeCommand()` the selected command is executed.

If you are copying or cutting a file to the clipboard, you can switch to Explorer to confirm that the item was copied to the clipboard. If you select the Properties item on the menu you will see the Properties dialog displayed, and so on.

Conclusion

The shell's context menu is a powerful tool. You may want to show the context menu to your users (such as when displaying shell items in a list view or tree view) or you may simply want to use the context menu to carry out some other action. Either way, the context menu is available to you and relatively easy to use. The code for the main form of this article's example program is shown in **Listing A**. In addition to the techniques explained in this article, the example shows how to add items to the context menu.

Listing A: *MAINU.CPP.*

```
#define NO_WIN32_LEAN_AND_MEAN
#include <vcl.h>
#pragma hdrstop

#include <shellapi.h>
#include <shlobj.h>

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::FormCreate(TObject *Sender)
```

```
{
  // Initialize OLE. This is required to get
  // the cut and copy actions to work.
  if (OleInitialize(0) != S_OK) {
    ShowMessage("Unable to initialize OLE.");
    return;
  }
  FileNameEdit->Text = Application->ExeName;
}

void __fastcall
TForm1::FormDestroy(TObject *Sender)
{
  // Unitialize OLE
  OleUninitialize();
}

void __fastcall
TForm1::GoBtnClick(TObject *Sender)
{
  // Get an IShellFolder for the desktop.
  LPSHELLFOLDER DesktopFolder;
  SHGetDesktopFolder(&DesktopFolder);
  if (!DesktopFolder) {
    ShowMessage(
      "Failed to get Desktop folder.");
    return;
  }

  // Separate the file from the folder.
  String FilePath = ExtractFilePath(
    FileNameEdit->Text);
  String FileName = ExtractFileName(
    FileNameEdit->Text);

  // Get a pidl for the folder the file
  // is located in.
  wchar_t Path[MAX_PATH];
  LPITEMIDLIST ParentPidl;
  DWORD Eaten;
  StringToWideChar(FilePath, Path, MAX_PATH);
  DWORD Result =
    DesktopFolder->ParseDisplayName(
      Handle, 0, Path, &Eaten, &ParentPidl, 0);
```

```
if (Result != NOERROR) {
  ShowMessage("Invalid file name.");
  return;
}

// Get an IShellFolder for the folder
// the file is located in.
LPSHELLFOLDER ParentFolder;
Result = DesktopFolder->BindToObject(ParentPidl,
  0, IID_IShellFolder, (void**)&ParentFolder);
if (!ParentFolder) {
  ShowMessage("Invalid file name.");
  return;
}

// Get a pidl for the file itself.
LPITEMIDLIST Pidl;
StringToWideChar(
  FileName, Path, MAX_PATH);
ParentFolder->ParseDisplayName(
  Handle, 0, Path, &Eaten, &Pidl, 0);

// Get the IContextMenu for the file.
LPCONTEXTMENU CM;
ParentFolder->GetUIObjectOf(
  Handle, 1, (LPCITEMIDLIST*)&Pidl,
  IID_IContextMenu, 0, (void**)&CM);

if (!CM) {
  ShowMessage(
    "Unable to get context menu interface.");
  return;
}

// Set up a CMINVOKECOMMANDINFO structure.
CMINVOKECOMMANDINFO CI;
ZeroMemory(&CI, sizeof(CI));
CI.cbSize = sizeof(CMINVOKECOMMANDINFO);
CI.hwnd = Handle;

if (Sender == GoBtn) {
  // Verbs that can be used are cut, paste,
  // properties, delete, and so on.
  String Action;
```

```
  if (CutRb->Checked)
    Action = "cut";
  else if (CopyRb->Checked)
    Action = "copy";
  else if (DeleteRb->Checked)
    Action = "delete";
  else if (PropertiesRb->Checked)
    Action = "properties";

  CI.lpVerb = Action.c_str();
  Result = CM->InvokeCommand(&CI);
  if (Result)
    ShowMessage(
      "Error copying file to clipboard.");

  // Clean up.
  CM->Release();
  ParentFolder->Release();
  DesktopFolder->Release();
} else {
  HMENU hMenu = CreatePopupMenu();
  DWORD Flags = CMF_EXPLORE;
  // Optionally the shell will show the extended
  // context menu on some operating systems when
  // the shift key is held down at the time the
  // context menu is invoked. The following is
  // commented out but you can uncommnent this
  // line to show the extended context menu.
  // Flags |= 0x00000080;
  CM->QueryContextMenu(
    hMenu, 0, 1, 0x7FFF, Flags);

  // Merge the form's popup menu with the shell
  // menu.
  MENUITEMINFO mi;
  char buff[80];
  // Work backwards, adding each item to the
  // top of the shell context menu.
  for (int i=PopupMenu1->Items->Count-1;
         i>-1;i--) {
    ZeroMemory(&mi, sizeof(mi));
    mi.dwTypeData = buff;
    mi.cch = sizeof(buff);
    mi.cbSize = 44;
```

```
  mi.fMask = MIIM_TYPE | MIIM_ID | MIIM_DATA;
  // Get the menu item.
  DWORD result = GetMenuItemInfo(
    PopupMenu1->Handle, i, true, &mi);
    if (result) {
      // Modify its ID by adding 100 to the
      // Command property. This ensures that
      // there are no conflicts between the
      // shell command IDs and the popup items.
      mi.wID = PopupMenu1->Items->
        Items[i]->Command + 100;
      // Add the item to the shell menu.
      InsertMenuItem(hMenu, 0, true, &mi);
    }
}
// Show the menu.
TPoint pt;
GetCursorPos(&pt);
int Cmd = TrackPopupMenu(hMenu,
  TPM_LEFTALIGN | TPM_LEFTBUTTON |
  TPM_RIGHTBUTTON | TPM_RETURNCMD,
  pt.x, pt.y, 0, Handle, 0);
// Handle the command. If the return value
// from TrackPopupMenu is less than 100 then
// a shell item was clicked.
if (Cmd < 100 && Cmd != 0) {
  CI.lpVerb = MAKEINTRESOURCE(Cmd - 1);
  CI.lpParameters = "";
  CI.lpDirectory = "";
  CI.nShow = SW_SHOWNORMAL;
  CM->InvokeCommand(&CI);
}
// If Cmd is > 100 then it's one of our
// inserted menu items.
else
  // Find the menu item.
  for (int i=0;
       i<PopupMenu1->Items->Count;i++) {
    TMenuItem* menu =
      PopupMenu1->Items->Items[i];
    // Call its OnClick handler.
    if (menu->Command == Cmd - 100)
      menu->OnClick(this);
```

```
      }
    // Release the memory allocated for the menu.
    DestroyMenu(hMenu);
  }
}

void __fastcall
TForm1::Button2Click(TObject *Sender)
{
  if (OpenDialog->Execute())
    FileNameEdit->Text = OpenDialog->FileName;
}

void __fastcall
TForm1::CloseBtnClick(TObject *Sender)
{
  Close();
}

// The OnClick handlers for the poup menu.
void __fastcall
TForm1::Hello1Click(TObject *Sender)
{
  ShowMessage("Hello!");
}

void __fastcall
TForm1::Test1Click(TObject *Sender)
{
  ShowMessage("This is a test.");
}
```

Hidden treasures of Sysutils, Part 1

by Mark G. Wiseman

As I am sure you know, the Visual Component Library (VCL) in C++Builder contains a wealth of visual and non-visual components and classes. You might not know that there are a lot of very useful stand-alone functions and global variables in the VCL.

Many of these functions and variables are hidden in the `Sysutils` namespace. I say hidden, because although most of these functions and variables are actually documented in the online help, they are not easy to find. Naturally, the header that prototypes these variables and functions is SYSUTILS.HPP. This header is automatically included by VCL.H so it is not necessary to specifically include it in a default VCL application.

The SYSUTILS unit contains functions and variables you can use to work with time, dates, numbers, currency, files, file names, and strings. There is also a small hodgepodge of miscellaneous system functions and variables.

In this article I will explain some of the variables and functions you can use when working with times and dates. In future articles I will explain other hidden treasures found in the SYSUTILS unit.

Times and dates

The functions and variables you can use to work with times and dates are based on the `TDateTime` class. As you might expect, `TDateTime` is a C++ class that implements dates and times. It was designed to interface with the Delphi `TDateTime` data type, which is nothing more than a `double`. Several of the functions in `Sysutils` mirror methods in the C++Builder `TDateTime` class.

Many of the time and date variables are set by the `GetFormatSettings()` function in `Sysutils`. `GetFormatSettings()` uses the locale information provided by Windows and is called when the VCL is initialized at application startup. The examples you will see in this article are specific to my system, which uses the United States locale settings.

`GetFormatSettings()` is also called by the `TApplication` object any time your program receives a `WM_WININICHANGE` message. The good news is that your programs should automatically update the display formats of times and dates if the user changes time or date settings. The bad news is that you must be careful when using the time and date variables. It is important to understand that these variables can change while your program is running and you should write your code to account for possible changes.

In the sample program that accompanies this article, I actually assign values to some of the variables to demonstrate how the variables are used by some of the time and date functions. In general, this is not a good practice since the system may change their values at any time.

Dealing with times

One set of variables and functions deals specifically with times. I will discuss those variables and functions in the following sections.

Time variables

Two time variables contain constant values and are self-explanatory. Here are their declarations:

```
int SecsPerDay = 86400;

int MSecsPerDay = 8640000;
```

Other variables contain values that are set at runtime using the `GetFormatSettings()` function. Those variables include:

```
char TimeSeparator;
AnsiString TimeAMString;
AnsiString TimePMString;
AnsiString ShortTimeFormat;
AnsiString LongTimeFormat;
```

On my system the `TimeSeparator` variable contains the ‘:’ character. The VCL uses this value for formatting output of time strings, and for parsing input strings into times. Look at the “FormatDateTime” topic in the C++Builder online help for an explanation of time and date formatting options.

Time functions

In addition to the time variables explained in the previous section, `Sysutils` also contains several time functions. Those functions include:

```
TDateTime EncodeTime(Word Hour,
  Word Min, Word Sec, Word MSec);void DecodeTime(
  TDateTime Time, Word &Hour, Word &Min,
  Word &Sec, Word &MSec);
TDateTime Time(void);
```

```
AnsiString TimeToStr(TDateTime Time);
TDateTime StrToTime(const AnsiString S);
```

The `EncodeTime()` function takes four parameters representing hours, minutes, seconds, and milliseconds, and returns a `TDateTime`. Here is an example:

```
TDateTime t = EncodeTime(21, 10, 30, 0);
```

When `EncodeTime()` returns, the `TDateTime` variable will contain the time as specified by the parameters passed to the function.

The `DecodeTime()` function performs the reverse:

```
Word hours, minutes,
  seconds, milliseconds;
DecodeTime(t, hours,
  minutes, seconds,
milliseconds);
```

The `Time()` function returns the current system time and `TimeToStr()` converts a `TDateTime` to an `AnsiString` using the format specified in the `LongTimeFormat` variable. For example:

```
TDateTime t = Time();
String s = TimeToStr(t);
```

The `StrToTime()` function takes an `AnsiString` and returns a `TDateTime`. Note that the character used to separate the time portions in the `String` (hours, minutes, and seconds) must match `TimeSeparator` for this function to work. Here is an example:

```
TDateTime t = StrToTime("8:27");
```

Working with dates

Now that I have covered the time variables and functions found in `Sysutils`, I will explain the date variables and functions.

Date variables

The VCL sets the following variables using the `GetFormatSettings()` function:

```
char DateSeparator;
AnsiString ShortDateFormat;
AnsiString LongDateFormat;
AnsiString ShortMonthNames[12];
AnsiString LongMonthNames[12];
AnsiString ShortDayNames[7];
AnsiString LongDayNames[7];
```

`ShortMonthNames` contains abbreviated names for the months of the year (“Jan”, “Feb”, “Mar”, and so on). As you can see from the declaration, `ShortMonthNames` is an array. To extract the short month name for the month of February, for example, you could use code like this:

```
String Feb = ShortMonthNames[1];
```

`LongMonthNames` contains the full name of the months of the year (“January”, “February”, “March”, etc.). The variables for day names work similarly.

There are two more date variables that are set by the VCL, but they are not connected with the Windows locale information.

```
Word TwoDigitYearCenturyWindow;
Word MonthDays[2][12];
```

`MonthDays` is very useful for the quick lookup of the number of days in a month. Here is a simple function that uses `MonthDays`:

```
int DaysInMonth(int year, int month)
{
  int leap = IsLeapYear(year) ? 1 : 0;
  return(MonthDays[leap][month - 1]);
}
```

Borland included the `woDigitYearCenturyWindow` variable so users with legacy code could address Y2K issues related to 2-digit years. If you’d like more information on this variable, look in the online help under the topic “Y2K Issues.”

Date functions

The `Sysutils` namespace declares the following date functions:

```
TDateTime EncodeDate(
```

```
  Word Year, Word Month, Word Day);
void DecodeDate(TDateTime Date,
  Word &Year, Word &Month, Word &Day);
bool IsLeapYear(Word Year);
int DayOfWeek(TDateTime Date);
TDateTime Date(void);
AnsiString DateToStr(TDateTime Date);
TDateTime IncMonth(const TDateTime Date,
  int NumberOfMonths);
TDateTime StrToDate(const AnsiString S);
```

The `EncodeDate()` function takes year, month, and day parameters and returns a `TDateTime` object. The `DecodeDate()` function performs the reverse by taking a `TDateTime` and returning the year, month, and day in variables passed to the function by reference.

`IsLeapYear()` returns `true` if a year is a leap year and `false` otherwise. `IsLeapYear()` does work correctly for year 2000.

`DayOfWeek()` takes a `TDateTime` as input and returns an `int` representing the day of the week. This is a one-based value, so to use it with the `Sysutils` array variables (such as `LongDayNames` or `LongMonthNames`) you will have to subtract one. Here is an example:

```
TDateTime id(2000, 7, 4);
int dw = DayOfWeek(id);
String dayName = LongDayNames[dw - 1];
```

The `Date()` function returns the current system date. `DateToStr()` takes a `TDateTime` and returns an `AnsiString` representing the date using the format specified in `ShortDateFormat`.

`IncMonth()` is a very useful function. You can use it to add or subtract months from a date. It takes a `TDateTime` value argument and an `int` argument that represents the number of months to increment or decrement. Positive numbers add months and negative numbers subtract months. Here are some examples:

```
TDateTime d = TDateTime(2000, 7, 31);
TDateTime nextmonth = IncMonth(d, 1);
TDateTime lastmonth = IncMonth(d, -1);
TDateTime nextyear = IncMonth(d, 12);
TDateTime lastyear = IncMonth(d, -12);
```

There is one potential “gotcha” with `IncMonth()`. In the previous example, `d` is set to `7/31/2000`. When `d` is decremented by one month, the result, `lastmonth`, is `6/30/2000`. This is because June

has only 30 days. When you think about it, though, this is the correct behavior.

`StrToDate()` takes an `AnsiString` and parses it into a date. This function is not as useful as it may sound, because it is very particular about the format of the string. The separator character in the date string must match the separator contained in the `DateSeparator` variable or an exception is thrown. `StrToDate()` will accept dates of the form "`8/27`" and will assume the missing year is the current year. `StrToDate()` will also accept two-digit years, but it may not translate as you would expect unless you have set the `TwoDigitYearCenturyWindow` variable correctly.

Date/time functions

There are also functions in `Sysutils` that work on both the date and time portions of a `TDateTime` object. Those functions are:

```
TDateTime Now(void);
AnsiString DateTimeToStr(
  TDateTime DateTime);
AnsiString FormatDateTime(
  const AnsiString Format,
  TDateTime DateTime);
void DateTimeToString(
  AnsiString &Result,
  const AnsiString Format,
  TDateTime DateTime);
TDateTime StrToDateTime(
  const AnsiString S);
void DateTimeToSystemTime(
  TDateTime DateTime,
  SYSTEMTIME &SystemTime);
TDateTime SystemTimeToDateTime(
  const SYSTEMTIME &SystemTime);
```

The `Now()` function returns a `TDateTime` that contains the current system time and date. `DateTimeToStr()` takes a `TDateTime` and returns an `AnsiString` with the date formatted using `ShortDateFormat` and the time formatted using `LongTimeFormat`.

`FormatDateTime()` is a very useful function. `FormatDateTime()` will return an `AnsiString` containing the date, the time, or the date and time in nearly any format you want. For example:

```
TDateTime n = Now();
String s = FormatDateTime(
```

```
  "hh:nn 'on' mmmm, d, yyyy);
// Example: s = 10:27 on March 15, 2000
```

Calling `FormatDateTime(LongTimeFormat, t)` is the same as calling `TimeToStr(t)`, and `FormatDateTime(ShortDateFormat, d)` is the same as `DateToStr(d)`. In fact, `TimeToStr()`, `DateToStr()`, and `FormatDateTime()` all call the `DateTimeToString()` function internally.

`DateTimeToString()` is nearly identical to `FormatDateTime()` in function. Instead of returning an `AnsiString`, though, it takes an additional argument, a reference to an `AnsiString` that receives the formatted output.

`StrToDateTime()` parses an `AnsiString` and returns a `TDateTime`. Like `StrToDate()` and `StrToTime()`, you may find it of limited use.

`DateTimeToSystemTime()` and `SystemTimeToDateTime()` can be used to translate `TDateTime` values into the `SYSTEMTIME` structure used by Windows and vice versa.

Conclusion

As I mentioned, you can find most of these functions and variables in the C++Builder online help. But unless you know their exact names, it is difficult to know what to look for. A great source for learning about all the variables and functions in the `Sysutils` namespace is the source itself. You can look at the header file, SYSUTILS.HPP, but you will find much more information in the Pascal source, SYSUTILS.PAS, which is very well documented.

I am sure you will find some of the time and date variables and functions found in `Sysutils` to be useful. In Part 2 of this series I will examine number and currency routines.

Improving build times with pre-compiled headers

by Harold Howe

C++Builder is one of the fastest C++ compilers around, and probably the fastest Win32 C++ compiler in terms of compilation speed. Despite the speed advantage that C++Builder holds over other C++ compilers, the C++ language is complex and requires a lot of processing time to compile. This article explains why C++ compilers are inherently slow, and demonstrates techniques to boost compile times in C++Builder. The focus of this article is pre-compiled headers and how to use them properly in your projects.

Why C++ compilers are slow

In C++, you cannot call a function from a source file unless that function has been previously defined or declared. So what does this mean? Consider a simple example where function `A()` calls function `B()`. `A()` cannot call `B()` unless a prototype for `B()`, or the function body for `B()`, resides somewhere above the function body for `A()`. The code below illustrates this point.:

```
// declaration or prototype for B
void B();

void A()
{
  B();
}

// definition, or function body of B
void B()
{
  cout << "hello";
}
```

The code will not compile without the prototype for `B()`, unless the function body for `B()` is moved up above `A()`.

Function prototypes serve a crucial role for the compiler. Every time you execute a routine, the compiler must insert proper code to call the routine. The compiler must know how many parameters to pass to the function, and whether the function expects its parameters on the stack or in registers. In short, the compiler needs to know how to generate the correct code to call the function, and it can only do this if it

has seen a previous declaration or definition for the function that is being called.

To simplify the prototyping of functions and classes, C++ supports the `#include` directive. The `#include` directive allows a source file to read function prototypes from a header file prior to the location in code where the prototyped functions are called. The `#include` directive plays an important role in Win32 C++ development. Function prototypes for C runtime library (RTL) functions are provided in a standard set of header files, the Win32 API is prototyped in a set of header files provided by Microsoft, and the VCL classes and functions are prototyped in the VCL headers. You can't create a useful Windows program without including header files provided by Microsoft and Borland.

Header files help implement C++ type checking in a manner that is easy to manage for the programmer. However, this benefit comes at a cost. When the compiler runs across an `#include` directive, it literally opens the included file and inserts it into the file it is compiling. The compiler then parses the included file as part of the file currently compiling. So what happens if the first header file includes yet another file? The compiler will suck in that file and start parsing it as well. Imagine what happens when 10, 20, or even 100 files are included? While this number of include files may sound large, it isn't unrealistic when you start adding up the Windows SDK header files and the VCL header files.

To demonstrate how the compiler branches off and translates included files, I created a simple console mode program. Here is that program:

```
// include some standard header files
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>

#pragma hdrstop
#include <condefs.h>

int main()
{
  printf("Hello from printf.\n");
  cout << "Hello from cout" << endl;
  MessageBeep(0);
  return 0;
}
```

To prove a point, I turned off pre-compiled headers in the Project Options dialog. When I built this project, the build progress dialog reported that the project contains 130,000 lines of code. 130 *thousand* lines! How can that be? The source file contains only four lines of actual code. The 130,000 lines were contained in STDIO.H, STRING.H, IOSTREAM.H, WINDOWS.H, and all of the other header files that are included by these four header files. In this example, the compiler spent the vast majority of its time

processing header files.

Next I added a new unit with a single simple function, and added code to the console application that calls this function:

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>
// prototype for A() is in unit1.h
#include "Unit1.h"
#pragma hdrstop

void A()
{
  printf("Hello from function A.\n");
}
```

The console application now looked like this:

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>
#include "Unit1.h"

#pragma hdrstop
#include <condefs.h>

USEUNIT("Unit1.cpp");

int main()
{
  printf("Hello from printf.\n");
  cout << "Hello from cout" << endl;
  A();
  MessageBeep(0);
  return 0;
}
```

When I built the project again, the compiler progress dialog reported 260,000 lines of code compiled. This is because the compiler had to translate the same set of header files twice, once for each of the CPP files in the project. Imagine how this line count would grow in a large project. The burden of processing

the same group of header files over and over greatly increases compile times.

How C++Builder uses pre-compiled headers

The engineers at Borland realized that they could decrease build times by designing a compiler that did not process the same header files over and over during a build. To achieve this goal, Borland C++ 3 (Borland C++, not C++Builder) introduced the concept of pre-compiled headers.

The idea behind pre-compiled headers is relatively simple. When the compiler processes a set of header files for a particular source file, it saves the compiled image of the header files on the hard drive. When that set of header files is required by another source file, the compiler loads the compiled image instead of compiling the header files a second time.

Next, I turned pre-compiled headers on for the console mode program and entered PCH.CSM for the pre-compiled header filename (in the Project Options dialog). When I built the project again, the compiler processed 130,000 lines of code when it compiled PROJECT1.CPP, but only 20 lines of code when it compiled UNIT1.CPP. The compiler generated a pre-compiled image when it parsed the first source file, and that pre-compiled image was loaded and reused for the second source file. Imagine the performance boost that you would attain if the project contained 50 source files instead of only two.

Pre-compiled headers in a VCL GUI project

The use of pre-compiled headers in the previous example reduced the build time of the project by almost 50%. But that was a simple console mode program that didn't do much. You probably want to know how you can take advantage of pre-compiled headers in a full-blown VCL GUI program. By default, C++Builder automatically turns on pre-compiled headers for you. However, C++Builder does not pre-compile every header file that is used by your program. It only pre-compiles the file VCL.H. Inspect the top of any form's source file and you will find these lines:

```
#include <vcl.h>
#pragma hdrstop
```

The `#pragma hdrstop` directive tells the compiler to stop generating the pre-compiled image. Any `#include` statement located before the `hdrstop` directive will be pre-compiled, while any `#include` below the directive will not be pre-compiled.

So how many header files are pre-compiled when VCL.H is compiled? If you look at VCL.H, you will see that it includes another file called VCL0.H. If you don't alter the default settings of C++Builder, VCL0.H will include a small set of VCL header files. For C++Builder 4 and 5, those files are:

```
#include <system.hpp>
```

```
#include <windows.hpp>
#include <messages.hpp>
#include <sysutils.hpp>
#include <classes.hpp>
#include <graphics.hpp>
#include <controls.hpp>
#include <forms.hpp>
#include <dialogs.hpp>
#include <stdctrls.hpp>
#include <extctrls.hpp>
```

This is a small cross section of header files, and it probably represents only a subset of the header files that are used in a moderate to large sized project. VCL0.H does allow you to pre-compile more header files through the use of conditional defines. For example, you can define a variable called `INC_VCLDB_HEADERS` to pre-compile the VCL database header files, or `INC_VCLEXT_HEADERS` to pre-compile header files for the extra controls that come with C++Builder. If you define a variable called `INC_OLE_HEADERS`, C++Builder will pre-compile some of the Win32 COM header files. If you use these defines, place them before the `#include` statement for VCL.H:

```
#define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS
#include <VCL.H>
#pragma hdrstop
```

If you decide to try this technique, make sure you add to the two defines to every CPP file in your project, even if they don't use database classes or extra controls. The reasoning for this will be explained shortly.

Optimizing pre-compiled headers

The default pre-compiled header settings do reduce the time it takes to build a project. You can prove this fact by timing a full build of a large project when use of pre-compiled headers are enabled, and by timing the build when pre-compiled headers are disabled. The goal of this article is to tweak the way C++Builder pre-compiles files in order to reduce build times even more. In the next section I will outline two techniques for improving build times.

Before we discuss those techniques, it is important to realize how C++Builder determines whether it can use an existing pre-compiled image when compiling a source file. C++Builder generates a unique pre-compiled image for every source file in your project. These pre-compiled images are saved in a file on your hard drive. The compiler will reuse an existing pre-compiled image when two source files require the same pre-compiled image. This is an important distinction. Two source files will require the same pre-compiled image if they include exactly the same files. Furthermore, they must include the files in the same order. Simply put, the source files must be identical up to the `#pragma hdrstop` directive.

Table A shows a few examples of pre-compiled images that don't match, and **Table B** shows examples of pre-compiled images that do match.

Table A: Mismatched header includes.

Example	UNIT1.CPP	UNIT2.CPP
1	`#include <stdio.h>` `#pragma hdrstop`	`#include <iostream.h>` `#pragma hdrstop`

2	`#include <stdio.h>`
	`#include <iostream.h>`
	`#pragma hdrstop`
		`#include <stdio.h>`
		`#pragma hdrstop`

3	`#include <stdio.h>` `#pragma hdrstop`	`#pragma hdrstop` `#include <stdio.h>`

Table B: Properly matched header includes.

Example	UNIT1.CPP	UNIT2.CPP

1	`#include <stdio.h>`
	`#include <string.h>`
	`#include <iostream.h>`
	`#include <windows.h>`
		`#include <stdio.h>`
		`#include <string.h>`
		`#include <iostream.h>`
		`#include <windows.h>`

	`#include “unit1.h”`	`#include “unit1.h”`
	`#pragma hdrstop`	`#pragma hdrstop`
2	`#define INC_VCLDB_HEADERS`	`#define INC_VCLDB_HEADERS`
	`#define INC_VCLEXT_HEADERS`	`#define INC_VCLEXT_HEADERS`
	`#include <vcl.h>`	`#include <vcl.h>`
	`#pragma hdrstop`	`#pragma hdrstop`
	`#include “unit1.h”`	`#include “unit2.h”`

When the compiler processes a source file with a pre-compiled image that does not match an existing image, it will produce a completely new image from scratch. Look at Example 2 in **Table A**. Even though STDIO.H is compiled along with UNIT1.CPP, the compiler will translate STDIO.H again when it compiles UNIT2.CPP because the second unit’s include list does not match that of the first unit. Pre-compiled headers reduce compile times only when the compiler can reuse an existing pre-compiled image across multiple source files.

This is the foundation for the techniques that I explain in the next section. Pre-compile as many header files as you can, and make sure that you use the same pre-compiled image in every source file.

Pre-compiled header optimization techniques

I have discovered two different techniques for optimizing pre-compiled headers. I will explain those techniques in the following sections.

Technique One

The first technique involves boosting the number of files that VCL.H includes. This is accomplished by adding two conditional defines to every source file. To implement this technique, open every CPP file in your project (including the project source file) and modify the first two lines so they look like this:

```
#define  INC_VCLDB_HEADERS
```

```
#define  INC_VCLEXT_HEADERS
#include <VCL.H>
#pragma hdrstop
```

If you don't like the idea of adding these defines to every source file, you can accomplish the same thing by adding `INC_VCLDB_HEADERS` and `INC_VCLEXT_HEADERS` to the *Conditional defines* field of the Project Options dialog (found on the Directories/Conditional page).

You might want to throw in some of the RTL header files that you commonly use, along with WINDOWS.H. Make sure you add the lines before the `#pragma hdrstop` line, and make sure that you list them in the same order in every C++ source file:

```
#define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS
#include <VCL.H>
#include <windows.h>
#include <stdio.h>
#pragma hdrstop
```

This technique works fairly well, but it isn't very flexible. If you decide to add a new header file to the list of files that get pre-compiled, you need to modify every C++ source file in your project. Furthermore, this technique is prone to error. If you mess up the order of your includes, you can actually make your compile times worse, not better.

Technique Two

The second technique addresses some of the downfalls described for the first technique. The strategy for this technique is to create one huge header file that includes every header file that is required by your project. This single file will include the VCL headers, the Windows SDK headers, and the RTL headers. It can also include all of the header files for forms and units that you have created, but you don't want to pre-compile header files that are likely to change (see the companion article "Pre-compiled header tips" for further explanation).

Here is an example of how the common header file (which I have named PCH.H) might look:

```
// PCH.H: Common header file
#ifndef PCH_H
#define PCH_H

// include every VCL header that we use
// could include VCL.H instead
#include <Buttons.hpp>
```

```
#include <Classes.hpp>
#include <ComCtrls.hpp>
#include <Controls.hpp>
#include <ExtCtrls.hpp>
#include <Forms.hpp>
#include <Graphics.hpp>
#include <ToolWin.hpp>

// include the C RTL headers that we use
#include <string.h>
#include <iostream.h>
#include <stdio.h>

// headers for the 3rd party controls
// TurboPower System
#include "StBase.hpp"
#include "StVInfo.hpp"

// Our custom controls
#include "DBDatePicker.h"
#include "DBRuleCombo.h"
#include "DBPhonePanel.h"

// project headers are pre-compiled
// only if PRECOMPILE_ALL is defined
#ifdef PRECOMPILE_ALL
#include "About.h"
#include "mainform.h"
// remainder of user-created header files
#endif

#endif
```

Note the section that checks to see if `PRECOMPILE_ALL` has been defined. This is a symbol that I created to indicate whether the individual headers for my project's units should be pre-compiled. See the following article for more information regarding the effects of pre-compiling your own headers.

Once you have created the gigantic common header file, you must modify every source file so it includes only this file. For example:

```
#include <VCL.H>
#include "pch.h"
#pragma hdrstop
```

I have chosen to leave the original include statement for VCL.H intact, but you might want to move VCL.H to the common header file.

After you have added the include for PCH.H to every C++ source file, don't insert any more include files prior to the `#pragma hdrstop`. Doing so will cause those C++ files to require a pre-compiled image that does not match the pre-compiled image from other files.

Results

I am currently employing Technique 2 without defining `PRECOMPILE_ALL` on my current project. The project is a medium sized client/server database program that consists of 113 C++ source files, most of which are forms or data modules. Using this technique, a full build of the project takes only 195 seconds. Of those 195 seconds, 40 seconds are spent generating the pre-compiled header, and about 40 seconds are spent linking. In the remaining time, the compiler translates 113 C++ source files. That's an average of one file per second. By way of comparison, the project takes more than 30 minutes to build when no pre-compiled headers are used, and the project takes 18 minutes to build when pre-compiled headers are used but this technique is not utilized.

Incremental makes with Technique 2 are lightning fast when no header files have changed. The compiler does not bother to regenerate the pre-compiled image on disk if no header files have changed. When this condition is met, an incremental make takes only one or two seconds, because only those C++ source files that have changed need to be compiled. The compiler spends all of its time compiling those files, instead of compiling system header files. When a header file does change, the speed of an incremental make depends on whether or not the `PRECOMPILE_ALL` flag was defined.

Conclusion

Proper use of pre-compiled headers can make a huge difference in the amount of time required to compile a project. The first step to properly using pre-compiled headers is in understanding how they work. The second step is putting that knowledge to work. By implementing the techniques explained in this article you can drastically reduce your project build times.

Pre-compiled header tips

by Harold Howe

There are several things to keep in mind when using pre-compiled headers. This article will explain what to do—and what not to do—when using pre-compiled headers. The tips in this article apply primarily to the second technique described in the article, “Improving build times with pre-compiled headers.”

Don’t pre-compile constant variables

The compiler cannot pre-compile a header file if it contains a constant variable that is assigned a value. For example, placing the following line in a header file can interfere with the creation of a pre-compiled header image:

```
const AnsiString strError = "Error!!";
```

If you want to place `const` variables in a header file, create a separate header that contains the constants, and don’t pre-compile that file. Try to reduce the burden on the compiler by not allowing this header file to include other files. Similarly, don’t include this file from other header files if you can help it. If this seems like a difficult task, you can use the `extern` keyword. Create a header file that contains `extern` prototypes for your constants. Then create a CPP file that defines the constants (i.e. gives them a value).

Note that the problem of `const` variables only occurs when you pre-compile your own source file headers. If you don’t pre-compile a header file, then you can add constants to it without any problems. Also, `#defines` do not present a problem, only `const` variables (although I don’t recommend that you use `#defines` simply to eliminate this problem).

Don’t pre-compile template headers

This suggestion is based on empirical evidence. I have a template class with several inline functions. The entire template class resides in a header file. The compiler was able to pre-compile this header file, but I noticed that the pre-compiled image was always re-generated during an incremental make. This apparently has to do with the way some templates are handled by the compiler. Some template headers pre-compile just fine, while others do not. I suggest that you go ahead and try to compile template headers, but pay close attention to the compiler progress dialog. You may need to stop pre-compiling certain template headers if they cause problems.

Keep an eye on the compiler progress dialog

The compiler progress dialog tells you how well your pre-compiled headers are working. After implementing pre-compiled headers, you should see that the compiler takes a long time to compile the first C++ source file in your project, and that subsequent files compile much faster. The compiler generates the pre-compiled header image during compilation of the first file in the project. When the first file is compiled, you will see the line count on the compiler progress dialog reach a huge number (100,000-500,000 lines). The line count for other files in the project should only be between 20 and 1,000 lines if you have defined the `PRECOMPILE_ALL` symbol. If you don't define this symbol, though, the line count should still stay under 15,000 lines or so. Once the compiler finishes translating the first file in the project, subsequent files should only take a second or two to compile.

If the compiler gets bogged down on one C++ file for more than a few seconds, you probably have a source file whose pre-compiled image doesn't match the image created by the common header file. The line count is another indicator. If you see the line count sail up above 50,000 lines for a particular source file, it's a good indication that the compiler was unable to apply the existing pre-compiled image to that source file.

Don't pre-compile header files that change frequently

When using pre-compiled headers, realize that any small change to a header file will force the compiler to regenerate the pre-compiled image. Based on my tests, this could take from 20 seconds to a minute. During the early stages of development, your project header files may change frequently. In that case you may want to pre-compile only system and VCL header files. This is the purpose of the `PRECOMPILE_ALL` symbol. It allows you to easily include or remove your header files from the pre-compiled image. For example:

```
// PCH.H: Common header file
#ifndef PCH_H
#define PCH_H

// include every VCL header that we use
#include <Buttons.hpp>
#include <Classes.hpp>

// include the C RTL headers that we use
#include <string.h>
#include <iostream.h>
#include <stdio.h>

// project headers are pre-compiled
```

```
// only if PRECOMPILE_ALL is defined
#ifdef PRECOMPILE_ALL
#include "About.h"
#include "mainform.h"
// remainder of user-created header files
#endif

#endif
```

To add your own header files to the pre-compiled header, add `PRECOMPILE_ALL` to the *Conditional defines* field in the Project Options dialog (Directories/Conditional page). If your header files change frequently, then don't add this conditional define to the project. When you don't pre-compile your own header files, a full build of your project will take a little longer. However, when you make a change to one of your header files, an incremental make will be faster because the compiler won't waste 20-60 seconds rebuilding the pre-compiled image.

I do not define `PRECOMPILE_ALL` for a project during the early stages of development because I am changing my header files frequently. I found that incremental makes of my project were taking more than 2 minutes. I the amount of time a full build took when the `PRECOMPILE_ALL` symbol was defined. Then I made a small change to the header file for my main form and performed an incremental make. Next, I repeated this process with the `PRECOMPILE_ALL` value not defined. **Table A** shows are the results.

Table A: Effects of defining PRECOMPILE_ALL

Setting	Full Build	Incremental Make
Not defined	195 seconds, 408,887 lines	28 seconds, 27,059 lines
Defined	179 seconds, 255,689 lines	179 seconds, 255,689 lines

Notice that a full build is 16 seconds faster when I pre-compile my own header files, but look what happens when I do an incremental make after changing a header file. The incremental make takes just as long as a full build. When you pre-compile your own header files, the compiler rebuilds the pre-compiled image every time you change a header file. Additionally, when the pre-compiled image changes, every file that depends on that image will be re-compiled as well. When you alter a header file, the entire project essentially gets rebuilt. When I do not pre-compile my own header files, the pre-compiled image never gets rebuilt. This keeps the incremental make time down.

Since you probably perform an incremental make much more often than you do a full build, it seems wise to keep the incremental make time down, even if it means that a full build will be 10% slower.

Don't remove existing include statements

Creating a common header file does not mean that you should remove from your header files the includes that the IDE automatically adds. Leave those include statements where they are. There are several reasons why you should leave existing include statements as they are. First, if you remove include statements from your header files, C++Builder will simply add them back again. Second, you may want to stop pre-compiling certain files, which would force to you add the include statements back into your source files. Lastly, by leaving include statements intact, you preserve the necessary inclusion order between header files. If you remove include statements, you will need to worry about the order of the include statements in your common header.

The include files have guards to prevent multiple inclusion, so there's no need to worry about including the same file twice. The point is that even though you may use a common include file, you don't need to worry about removing the include statements that C++Builder generates. When I create a new source file, I add include statements so that file compiles without relying on the common header. Once the new source file compiles, I insert an `#include` statement for the common header to keep compile times down.

Observe case sensitivity for include statements

It is important to understand that the compiler observes case sensitivity when matching pre-compiled images. If you include the common header with varying case in different units, the compiler will regenerate the pre-compiled image for each one. Take this code, for example.

```
// MAINFORM.CPP
#include <VCL.H>
#include "pch.h"
#pragma hdrstop

// SPLASH.CPP
#include <VCL.H>
#include "PCH.H" // mismatched case
#pragma hdrstop
```

In this case the compiler will generate and use a pre-compiled image for MAINFORM.CPP, but SPLASH.CPP will generate its own pre-compiled image. This will, obviously, slow down the compile time. The rule of thumb is this: every include file listed above the `#pragma hdrstop` directive should use the same case that other files use. Include statements below the `#pragma hdrstop` directive don't have to match case, because they are not pre-compiled.

Consider adding VCL.H to the common header

The common header shown in the previous article does not include the file VCL.H. Each CPP source file includes both VCL.H and PCH.H, like this:

```
#include <VCL.H>
#include "pch.h"
#pragma hdrstop
```

You may prefer to include VCL.H in the common header. If you do, then each CPP file can simply include the common header.

```
#include "pch.h"
#pragma hdrstop
```

This is cleaner, and less prone to error because you don’t have to worry about which file should be listed first. However, it violates the previous tip that suggested that you don’t remove existing `#include` statements. If you ever need to yank out the common header file, you will need to add VCL.H back into every CPP file in your project.

Stepping into the VCL source

by Kent Reisdorph

In the article, “Using the VCL source to track down bugs,” Brent Knigge explained a situation where the VCL was not acting as he expected. He said that he examined the VCL source to find the cause of the problem. What many C++Builder programmers do not realize is that they can step into the VCL source while debugging. I’ll explain how to do that in this article.

Preparing to step into the VCL

Obviously, stepping through the VCL is only possible if you have the VCL source to begin with. The Standard version of C++Builder does not come with the VCL source, so stepping through the VCL is not an option with that version of the product. If you have C++Builder Professional or Enterprise (formerly called the Client/Server version) you can step through the VCL source, provided that you elected to install the VCL source when you installed C++Builder, and provided you have properly set up your project options. The default project options don’t allow you to step into the source, but this is easy to remedy with a few mouse clicks.

The steps required to set up a project to allow stepping into the VCL differ with the various versions of C++Builder. For C++Builder 1, you only need to go to the Linker page of the Project Options dialog and check the *Link debug version of VCL* check box.

For C++Builder 3, 4, and 5, setting up a project to allow stepping into the VCL is a two-step process. First open the Project Options dialog. On the Linker page, check the *Use debug libraries* option. On the Packages page, uncheck the *Build with runtime packages* option. You must build the application without runtime packages because the VCL packages do not contain debug information. The debug information is only built into the debug library (LIB) files for the VCL.

After you have made the changes to the project options, rebuild your application.

A simple test

Once you have the project set up to use the debug libraries, stepping into the VCL source is easy. For a quick test of this theory, start a new application and set the project options as explained in the previous section. Place a button on the form and put this code in the button’s `OnClick` event handler:

```
Top = 20;
```

Set a breakpoint on this line of code and run the application. When the debugger stops at the breakpoint, press F7 to step into the VCL. You should find yourself at this function in CONTROLS.PAS:

```
procedure TControl.SetTop(Value: Integer);
begin
  SetBounds(FLeft, Value, FWidth, FHeight);
  Include(FScalingFlags, sfTop);
end;
```

From this point you can continue to step into the VCL methods to find out where a particular problem lies. The C++Builder debugger is smart enough to allow you to inspect variables and objects, even though the VCL source is Object Pascal.

In some cases you may find that you have stepped into assembly code. If you are familiar with assembly programming this probably won't concern you in the least. However, if you don't read assembly, you can usually continue stepping through the assembly code until you get to some source code that you understand.

Conclusion

Stepping through the VCL source can be extremely valuable when trying to track down a bug. It is unlikely (but not completely out of the question) that you will find a bug in the VCL. However, examining VCL variables and objects may help you spot a problem in your own code. This technique goes beyond just the VCL itself, since it applies to third party component libraries as well. That assumes, of course, that those third party libraries were built with debug information, and that you have the source for the library. In some cases you may need to rebuild third party component packages and LIB files to include debug information.

Using the VCL source to track down bugs

by Brent Knigge

You have probably encountered debugging situations where you lost all hope of ever finding the source of an error. When nothing appears to be wrong with your code, you may have to consider the VCL as a possible source of the problem.

Recently I derived a component from `TListBox` in order to add a property that mimics the `HideSelection` property found in `TTreeView`. I did this so the selection(s) of the list box would be hidden when the control loses focus. This helps eliminate confusion for the user when it appears that multiple controls have focus.

I decided to use the `Selected` property of `TListBox` to hide the selection when the list box loses focus. This property returns `true` if a particular item is selected, and `false` if no item is selected. You can also set the `Selected` property in order to select or deselect an item. `Selected` is an indexed property. That is, you ask for the selected state of a particular item by specifying the item index. For example:

```
if (ListBox1->Selected[0])
  // the item is selected
```

Remember that the first item has an index of `0` and the last item has an index of `Items->Count - 1`. I wanted to write code that iterates over the items in the list box, setting the `Selected` property of each item to `false`. My plan was to do this in the `OnExit` event handler so that the item states were cleared when the list box lost focus. My initial testing was done using a regular list box. The `OnExit` event handler looked like this:

```
void __fastcall
TForm1::ListBox1Exit(TObject *Sender)
{
  for(int i=0;
      i<ListBox1->Items->Count;i++)
    ListBox1->Selected[i] = false;
}
```

I added a few input controls to the form and ran the application. I selected a list box item and then tabbed to another control on the form. What happened at that point was a bit confusing. The VCL gave me an `EListError` exception saying that list index `0` was out of bounds.

Okay, so I might have made a mistake in the `for` loop. After examining the code in the loop, I couldn't see any reason for the list index out-of-bounds error. Further investigation revealed that setting the `MultiSelect` property of the list box to `true` did not result in an exception. That wasn't particularly helpful, though, because I needed a customized list box that could handle both single and multiple selection.

Still investigating the cause of the problem, I started hunting around in the VCL source. I found these lines in STDCTRLS.PAS:

```
procedure TCustomListBox.SetSelected(
  Index: Integer; Value: Boolean);
begin
  if SendMessage(Handle, LB_SETSEL,
    Longint(Value), Index) = LB_ERR then
    raise EListError.CreateFmt(
      SListIndexError, [Index]);
end;
```

Now that I knew where the exception was being raised, I could do some further investigation to find the cause of the problem.

Studying the documentation for the `LB_SETSEL` message, I found that an application sends this message to a list box in order to select an item in a multiple-selection list box. If an error occurs (such as when the list box only supports single-selection) the return value is `LB_ERR`, and the exception is raised.

Fortunately, I found that there is a way to do what I wanted for a single-selection list box using the Windows API. What I found was that I needed to send the list box an `LB_SETCURSEL` message. This message is the one that should be used for single-selection list boxes, instead of the `LB_SETSEL` message.

At this point I simply modified the code to check the value of the `MultiSelect` property, and take appropriate action based on the value of this property. The following code shows an updated version of my test code, which works whether `MultiSelect` is `true` or `false`:

```
void __fastcall
TForm1::ListBox1Exit(TObject *Sender)
{
  if(ListBox1->MultiSelect) {
    for(int i=0;
      i<ListBox1->Items->Count;i++)
      ListBox1->Selected[i] = false;
```

```
  }
  else // single selection only.
    SendMessage(ListBox1->Handle,
      LB_SETCURSEL, -1, 0);
}
```

This experience taught me that the initial error message that I received from the VCL did not accurately convey the actual problem. In fact, it may have hindered my investigation because I was focusing on the `for` loop as the source of the problem. As it turns out, there was nothing wrong with my code. The VCL simply was not behaving as I thought it should. Whether this is a bug in `TListBox` depends on your perspective. The point is that you should not be afraid to open up the VCL source files and examine them to understand what is going on in a given situation. Obviously the VCL source is in Object Pascal. Still, it is easy enough to read and understand, even if you don't have any Pascal experience.

When tracking down bugs, be sure to examine all possible sources. Look at your own code first, of course, but don't stop there if the problem persists. Probe deeper by looking into the VCL for possible problems. Once a problem is found and identified, you can usually find a workaround using the Windows API. In the end it may be the difference between understanding a problem and living with a puzzling mystery.

Changing window styles on the fly

by Kent Reisdorph

The VCL does a good job of insulating the C++Builder programmer from the Windows API. Sometimes, however, you need to use the API if you want to accomplish some goal that isn't provided for by the VCL. Take window styles, for example. Every window has a set of style bits that determine how the window should look. These style bits are or'd together to get the desired behavior for the window. Most VCL components surface window styles as properties. In some cases, though, the VCL does not provide properties for a particular style. Let's say, for example, that you want a progress bar that has no border. If you drop a `TProgressBar` component on a form you will find that there is no way to turn off the control's border. In that case you'll have to go to the Windows API in order to remove the progress bar's border.

Style types

Windows defines three style types. The first set of style types has to do with basic window styles. This includes styles that indicate whether the window has a title bar, whether it is a child window, and so on. The second group of styles consists of extended style types. You might consider these styles as "Win32 styles," since many of them refer to window styles that didn't exist in Windows 3.x. The third type of style consists of styles specific to a particular type of window. For example, the Win32 tree view control has a style called `TVS_DISABLEDRAGDROP`. This style indicates whether the tree view can accept dropped files.

You can find out about the basic window styles by looking at the `CreateWindow()` function in the Win32 help file. For extended styles, see the `CreateWindowEx()` function in the help file. For styles specific to a particular control, see the help for that control.

Changing styles at run time

Changing styles at runtime requires just a few lines of code. First you get the current window style. Next you add or remove style bits as needed. Finally, you reset the window style to the new style bits. Here's how the code looks:

```
long style = GetWindowLong(
  ProgressBar1->Handle, GWL_EXSTYLE);
style &= ~WS_EX_STATICEDGE;
SetWindowLong(ProgressBar1->Handle,
  GWL_EXSTYLE, style);
```

This code removes the `WS_EX_STATICEDGE` style from the progress bar. To add this style back you would use code like this:

```
long style = GetWindowLong(
  ProgressBar1->Handle, GWL_EXSTYLE);
style |= WS_EX_STATICEDGE;
SetWindowLong(ProgressBar1->Handle,
  GWL_EXSTYLE, style);
```

Typically you will place this code in your form's constructor or on `OnCreate` event handler.

The great debate?

Recently I had a rather spirited discussion on this topic with a fellow TeamB member. His position is that using `SetWindowLong()` is a hack and shouldn't be used in a VCL application. Instead, he said, you should create a new component and set the window style in an overridden `CreateParams()`. His reasoning was due to the fact that the VCL will, in some extreme cases, destroy and recreate a window. By setting the style bits in `CreateParams()` the component will always have the correct window styles regardless of what the VCL does.

My position on the subject is simple: Creating a component for such a simple task is extreme overkill. Imagine what your Component Palette would look like if you created a new component for every simple programming task! While the previous position is the absolute safest route to take, it simply isn't necessary in most cases. Ultimately you can decide which way to go.

Conclusion

Changing window styles at runtime (or through a component) is a simple and effective way of getting the correct window appearance and behavior for your components. This technique should be in your bag of Windows programming tricks.

Form designer tips

by Kent Reisdorph

Regardless of how long you have been using C++Builder there are no doubt features of the IDE that you have overlooked. This article will explain some lesser-known features of the IDE Form Designer.

Selecting the form

As you develop an application, you often need to select the form you are currently working on in order to change a property value or to generate an event handler. In some cases selecting the main form is as simple as clicking on the form. In many cases, though, the form may be completely obscured by the components that it contains.

One way to select a form that is covered with components is to use the Object Selector combo box at the top of the Object Inspector. This works, but it can be difficult to find the form class in the list of components, particularly if the form contains a large number of components.

Fortunately, the Form Designer offers a shortcut to selecting a form. Let's say, for example, that you have a Memo component aligned to the client area of the form. To select the form, simply click on the Memo and hit the Esc key on the keyboard. When you hit the Esc key, the Form Designer will select the form.

What actually happens is that the Form Designer changes the selection to the *parent* of the component that was originally selected. If you have a component on a panel, for example, you can click on the component and then hit Esc to select the panel. If the panel itself is located directly on the form, hitting Esc again will select the form.

Multiple selection of components on a container

Panels are often used as containers for other components. You may want to select a number of components on the panel in order to move them, or to change the value of a common property. You can, of course, simply use Shift-click to select the components. If you try to select components by dragging with the mouse, though, you will end up moving the panel. To select components on the panel by dragging, hold down the Ctrl key while dragging. Now you can drag to select components on the panel without moving the panel itself. This works for any component that acts as a container for other components (GroupBox, ScrollBox, ControlBar, and so on).

Fine tuning

Once you have placed a component on a form, you often need to move it slightly. You can move a component one pixel at a time by holding down the Ctrl key and pressing any of the arrow keys. Holding down both the Shift and the Ctrl keys will move the component to the next grid point when an arrow key is pressed.

To size a component one pixel at a time, hold down the Shift key and press any of the arrow keys. Note that the width of the component changes when you press either the right or left arrow keys, and that the height of the component changes when you press the up or down arrow keys.

Changing the properties of several components at once

When you select multiple components, the Object Inspector will display only those properties that the selected components have in common. Let's say, for example, that you have several Edit components on a form and that you want to make them all the same width. Just select all of the components and then change the `Width` property in the Object Inspector. The width of all selected components will change to reflect the new value. This is a convenient way of clearing the text from several Edit components at one time.

Use the Alignment dialog!

Often you need to make sure a group of components are aligned to one another in a specific way. For example, you may want an Edit component and its associated Label centered on one another. First select the Edit component, hold down the Shift key, and then select the Label component. Next, right-click on one of the selected components and choose Align from the context menu. The Alignment dialog comes up, allowing you to align the components in several ways. In this case you would select the Centers radio button in the Vertical radio group. When you click OK, the Edit component and Label will be centered on one another vertically.

There is one aspect of using the Align dialog that I should point out, and that is the concept of the *anchor component*. The first component of a group you select will act as the anchor component. That is, the anchor component will stay where it is, and all other components will move according to the alignment options you selected.

Finally, remember that the Alignment Palette does essentially the same thing as the Alignment dialog does. To use the Alignment Palette, choose View | Alignment Palette from the C++Builder main menu.

Conclusion

For some of you, this article may appear to be ridiculously introductory. However, I would be willing to bet that most of you have learned something about the Form Designer that you did not previously know. The RAD aspect of C++Builder is impressive, and this is particularly true once you learn how to use the IDE to its fullest extent.

Making ActiveX controls persistent

by Thomas Wieland

C++Builder makes it fairly easy to create your own ActiveX controls derived from a VCL component. However, if you add some new properties to your new ActiveX control, you might be astonished to find that your control does not behave as you expected. Specifically, when embedded in an application, all settings you make for these new properties in the Object Inspector are forgotten when you run the program. That is, they are set back to their default values. In this article I will show you how to make your control remember the settings. Or, as the computer scientist would say, how to make the properties of your control persistent.

Background

When you derive an ActiveX control from a VCL component, the complex structure of the resulting class ensures that all properties of the base component are saved and loaded automatically. The real work comes when you add new properties to the control. You'll need to do this in most cases, since you generally do not want to produce an ActiveX control that is identical to its underlying VCL component.

Unfortunately the problem of making your own properties persistent is almost completely undocumented. If you dig into the code generated by C++Builder, you will have to try hard to find out which operations carry out the saving and loading of the base component's properties. A class for an ActiveX control in C++Builder is usually derived from `TVclControlImpl`. This class alone has seventeen base classes!

One of these base classes is a class called `TVclComControl`. It comprises the basic functionality for creating an ActiveX control from a VCL component. `TVclComControl` has the methods we are looking for. In C++Builder 3 and 4 the methods are called `IPersistStreamInit_Load()` and `IPersistStreamInit_Save()`. In C++Builder 5, the methods were renamed to `IPersistStreamInit_LoadVCL()` and `IPersistStreamInit_SaveVCL()`. (Note that this article was written prior to the release of C++Builder 5. The techniques discussed in this article have not been tested with that version.) In principle, to make your new properties persistent, all you have to do is to overload these two functions. This is exactly what I will show you next.

Saving and loading properties

To illustrate, I created a sample control called `ClockLabelX`. This control is a simple text label that displays the current time. In the following paragraphs I refer to "the implementation class." In this case I am referring to the `TClockLabelXImpl` class. This class is generated by the IDE when you create a

new ActiveX control. The steps discussed in this section are an abstract of what is required to implement persistence for your ActiveX control properties. Later I will get into the specifics of the `ClockLabelX` ActiveX control.

Step 1: Declare the stream functions

First you have to declare the two functions you need to overload. Put the declarations in the public part of the implementation class of your control (note that these are the C++Builder 3 and 4 declarations):

```
HRESULT IPersistStreamInit_Load(
  LPSTREAM pStm,
  ATL_PROPMAP_ENTRY* pMap);
HRESULT IPersistStreamInit_Save(
  LPSTREAM pStm, BOOL fClearDirty,
  ATL_PROPMAP_ENTRY* pMap);
```

Step 2: Implement the stream functions

When implementing the load method you start with calling the respective operation of the base class `TVclComControl`. Since this is defined via templates, you need to give the name of your implementation class and of the base VCL component as parameters. For example:

```
HRESULT TClockLabelXImpl::
  IPersistStreamInit_Load(
    LPSTREAM pStm,
    ATL_PROPMAP_ENTRY* pMap)
{
  HRESULT hr = S_OK;
  hr = TVclComControl<
    TClockLabelXImpl, TStaticText>
    ::IPersistStreamInit_Load(pStm,pMap);
  if (FAILED(hr))
    return hr;
  ...
```

Step 3: Load the property values

Next you load the values of the properties you have added. You have to pick them out of the byte stream using the `pStm->Read()` function, which needs the address of the variable and its size. For a property of type `long`, the code might look like this:

```
long lInterval;
hr = pStm->Read(
  &lInterval, sizeof(long), NULL);
if (FAILED(hr))
  return hr;
set_UpdateInterval(lInterval);
```

Afterwards you can set the value internally.

Step 4: Save the property values

Saving the values of your properties works in much the same way. You first call the save function of the base class:

```
HRESULT TClockLabelXImpl
  ::IPersistStreamInit_Save(
    LPSTREAM pStm, BOOL fClearDirty,
    ATL_PROPMAP_ENTRY* pMap)
{
  HRESULT hr = S_OK;
  hr = TVclComControl<
    TClockLabelXImpl, TStaticText>
    ::IPersistStreamInit_Save(
      pStm, fClearDirty, pMap);
  if (FAILED(hr))
    return hr;
  ...
```

Next you retrieve the current value and write it to the stream:

```
long lInterval;
get_UpdateInterval(&lInterval);
hr = pStm->Write(
  &lInterval, sizeof(long), NULL);
if (FAILED(hr))
  return hr;
```

A clock label control

Now let’s take a look at a complete ActiveX control. As I have said, the ActiveX control I created for this article is called `ClockLabelX`. The control is a numeric clock that displays the current time,

updated at a given interval. Optionally, you can display the date in addition to the time.

I used `TStaticText` as the base class for the `ClockLabelX` control. I derived from `TStaticText` because C++Builder only allows you to create ActiveX controls from VCL components that have a window (and, by extension, a window handle). `TLabel` is a non-windowed control and cannot be used to create an ActiveX control. `TStaticText`, on the other hand, is a windowed label component and can be used as the base class for an ActiveX control.

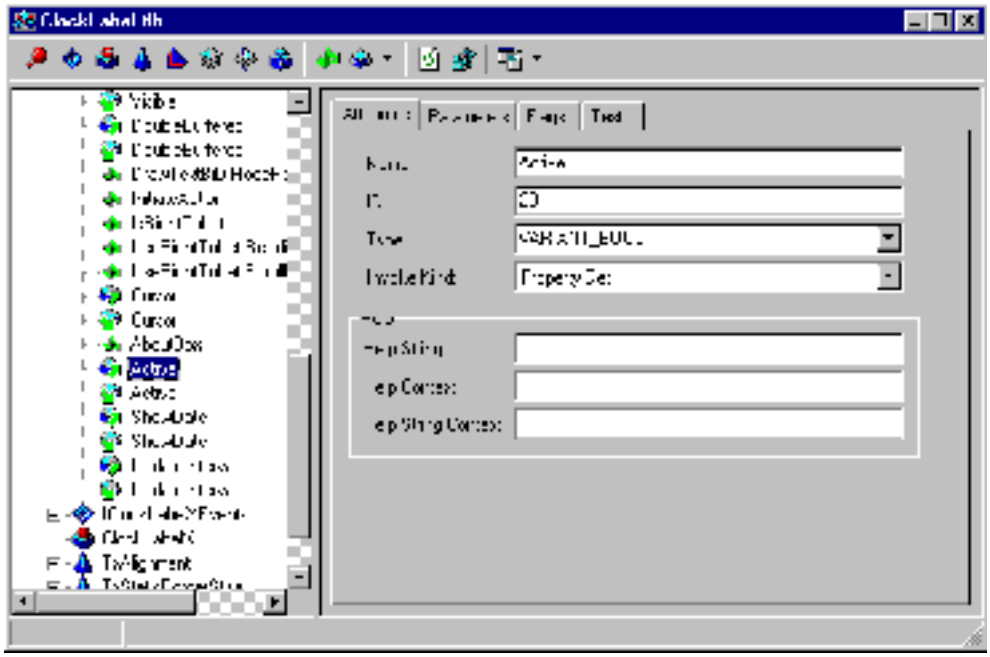
`ClockLabelX` adds three properties not found in the base class. Those properties are shown in **Table A**.

Table A: ClockLabelX Properties

Property	Type	Description
Active	VARIANT_BOOL	Specifies whether the label is updated at regular intervals.
ShowDate	VARIANT_BOOL	Specifies whether the date is displayed in the label in addition to the time.
UpdateInterval	long	Specifies the time interval (in milliseconds) at which the control will be updated.

After you have created the shell of the ActiveX control, you add properties via the Type Library Editor. Simply add a new property, give it the appropriate name, and set its data type. **Figure A** shows the Type Library Editor after adding the new properties.

Figure A



Add your own properties to the control using the Type Library Editor.

C++Builder will add the property declaration and the get and set functions to the implementation unit. You will need to fill out the get and set methods, of course. **Listings A** and **B** show the code I added to the implementation unit's header and to the unit itself. These listings don't show the code that was added automatically by C++Builder. I should add that the code shown in the listings is the code for C++Builder 4.

The values of all these properties should be persistent. To accomplish this, the implementation class contains code to save and load the property values as described in the previous section. It also includes code for the timer and painting of the control. That code is fairly straightforward so I won't explain it in this article.

When you use the control in an application, you will see that the new properties behave as you would expect. That is, they behave just as the VCL properties do.

Conclusion

Creating an ActiveX control with C++Builder is fairly easy, but the support does not (or cannot) go as far as it does for the underlying VCL control. The saving and loading of the values of additional properties is not guaranteed automatically. As you have seen, it is not very complicated to implement persistence. You only have to overload two methods in which you call the operations of the base class and then call the load or save functions accordingly.

ActiveX controls are an essential step on the way to component oriented software development. They allow you to transfer the component concept that you appreciate in C++Builder to numerous other

programming environments. With the support you get from C++Builder, it has become easy to make your own visual components enduring and reusable.

Listing A: *Code added to the implementation unit's header.*

```
private:
  TTimer*  FTimer;
  bool     FShowDate;
  void __fastcall OnTimer(TObject* Sender);
public:
  __fastcall TClockLabelXImpl();
  __fastcall ~TClockLabelXImpl();
  HRESULT IPersistStreamInit_Load(
    LPSTREAM pStm, ATL_PROPMAP_ENTRY* pMap);
  HRESULT IPersistStreamInit_Save(
    LPSTREAM pStm, BOOL fClearDirty,
    ATL_PROPMAP_ENTRY* pMap);
```

Listing B: *Code added to the implementation unit's source.*

```
__fastcall TClockLabelXImpl::TClockLabelXImpl()
{
  FTimer = new TTimer(m_VclCtl);
  FTimer->Enabled = true;
  FTimer->Interval = 1000;
  FTimer->OnTimer = OnTimer;
  FShowDate = false;

  ShortDateFormat = "mm'/'dd'/'yyyy";
  LongTimeFormat = "hh:mm:ss";
  OnTimer(NULL);
}

__fastcall TClockLabelXImpl::~TClockLabelXImpl()
{
  delete FTimer;
}

void __fastcall
TClockLabelXImpl::OnTimer(TObject* Sender)
{
  if (FShowDate)
    m_VclCtl->Caption = Date().DateString() +
```

```
    ", " + Time().TimeString();
  else
    m_VclCtl->Caption = Time().TimeString();
}

STDMETHODIMP
TClockLabelXImpl::get_Active(TOLEBOOL* Value)
{
  try
  {
    *Value = FTimer->Enabled;
  }
  catch(Exception &e)
  {
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
};

STDMETHODIMP
TClockLabelXImpl::get_ShowDate(TOLEBOOL* Value)
{
  try
  {
    *Value = FShowDate;
  }
  catch(Exception &e)
  {
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
};

STDMETHODIMP
TClockLabelXImpl::get_UpdateInterval(long* Value)
{
  try
  {
    *Value = FTimer->Interval;
  }
  catch(Exception &e)
  {
```

```
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
};

STDMETHODIMP
TClockLabelXImpl::set_Active(TOLEBOOL Value)
{
  try
  {
    const DISPID dispid = 23;
    if (FireOnRequestEdit(dispid) == S_FALSE)
      return S_FALSE;

    FTimer->Enabled = Value;
    FireOnChanged(dispid);
  }
  catch(Exception &e)
  {
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
};

STDMETHODIMP
TClockLabelXImpl::set_ShowDate(TOLEBOOL Value)
{
  try
  {
    const DISPID dispid = 24;
    if (FireOnRequestEdit(dispid) == S_FALSE)
      return S_FALSE;

    FShowDate = Value;
    FireOnChanged(dispid);
  }
  catch(Exception &e)
  {
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
```

```
};

STDMETHODIMP
TClockLabelXImpl::set_UpdateInterval(long Value)
{
  try
  {
    const DISPID dispid = 25;
    if (FireOnRequestEdit(dispid) == S_FALSE)
      return S_FALSE;

    FTimer->Interval = Value;
    FireOnChanged(dispid);
  }
  catch(Exception &e)
  {
    return Error(
      e.Message.c_str(), IID_IClockLabelX);
  }
  return S_OK;
};

HRESULT TClockLabelXImpl::IPersistStreamInit_Load(
  LPSTREAM pStm, ATL_PROPMAP_ENTRY* pMap)
{
  HRESULT hr = S_OK;
  hr = TVclComControl<TClockLabelXImpl,TStaticText>
    ::IPersistStreamInit_Load(pStm,pMap);
  if (FAILED(hr))
    return hr;

  long lInterval;
  hr = pStm->Read(&lInterval, sizeof(long), NULL);
  if (FAILED(hr))
    return hr;
  set_UpdateInterval(lInterval);

  TOLEBOOL bActive;
  hr = pStm->Read(
    &bActive, sizeof(TOLEBOOL), NULL);
  if (FAILED(hr))
    return hr;
  set_Active(bActive);
```

```
  TOLEBOOL bShowDate;
  hr = pStm->Read(
    &bShowDate, sizeof(TOLEBOOL), NULL);
  if (FAILED(hr))
    return hr;
  set_ShowDate(bShowDate);

  return hr;
}

HRESULT TClockLabelXImpl::IPersistStreamInit_Save(
  LPSTREAM pStm, BOOL fClearDirty,
  ATL_PROPMAP_ENTRY* pMap)
{
  HRESULT hr = S_OK;
  hr = TVclComControl<TClockLabelXImpl,TStaticText>
    ::IPersistStreamInit_Save(
      pStm, fClearDirty, pMap);
  if (FAILED(hr))
    return hr;

  long lInterval;
  get_UpdateInterval(&lInterval);
  hr = pStm->Write(&lInterval, sizeof(long), NULL);
  if (FAILED(hr))
    return hr;

  TOLEBOOL bActive;
  get_Active(&bActive);
  hr = pStm->Write(
    &bActive, sizeof(TOLEBOOL), NULL);
  if (FAILED(hr))
    return hr;

  TOLEBOOL bShowDate;
  get_ShowDate(&bShowDate);
  hr = pStm->Write(
    &bShowDate, sizeof(TOLEBOOL), NULL);

  return hr;
}
```

Using sockets

by Andrea Fasano

In the past several years we have seen incredible growth in networks in general, and the Internet in particular. Not surprisingly, the demand for Internet-based applications has increased as a result. Even if you're not currently working on an Internet application, you should take the time to learn a little about programming for the Internet The foundation of all network programming is the socket. In this article I will explain how the VCL facilitates the use of sockets.

What is a socket?

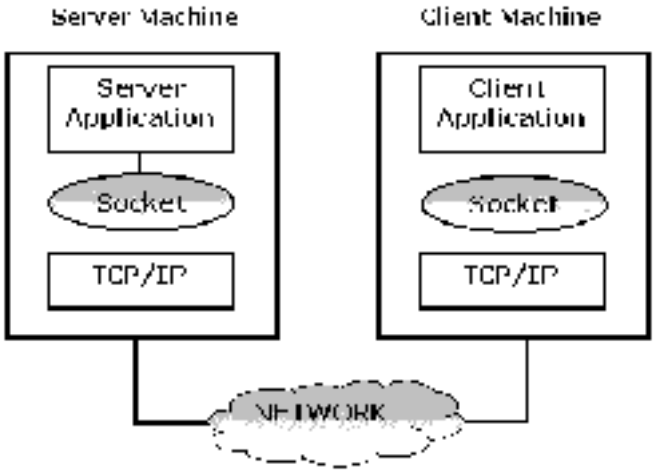
A socket is a very simple but powerful mechanism that allows communication between two or more applications. More precisely, sockets are the connection endpoints between the applications. One of the most interesting things about sockets is that you can safely ignore the boring details of the underlying network. It doesn't matter if two applications communicating with one another are running on the same machine, on two machines on the same local network, or on machines that are thousands of miles apart. It is only important to know that the applications are running on machines that are connected by some (hopefully functioning) hardware. For the most part you don't event need to know much about the protocol that allows the two machines to communicate (such as TCP/IP).

Socket terminology

Before going on, I need to explain some of the terms used to describe socket connections. **Figure A** illustrates a simple socket connection.

The client and server machines run a client and server application, respectively. Between them is the magical network that facilitates the transfer of data between the two machines. At the both ends of the connection is the TCP/IP protocol. The protocol translates data into a form the socket can understand. The individual applications then pump data through the socket. This particular type of connection is often defined as a *client/server* connection. The server offers its services to the client and manages incoming client requests. The client, on the other hand, makes its requests to the server, and expects an answer in reply.

Figure A



This diagram shows the simplest type of a socket connection between two applications.

But what exactly is a *service*? Generally speaking, a service is a set of operations the server is able to perform. Often this means that the server provides the client with some type of data. What that data is depends entirely on the server itself. Whether you know it or not, you use a lot of services when simply navigating the Internet. For example, the HTTP service is used when browsing Web pages and the FTP service is used when downloading files.

Known services are often associated with a particular *port*. A port is a numeric value that identifies a given service. Many common services have a standard port value. The POP3 service, for example, uses port 110, the FTP service uses port 21, and the SMTP service uses port 25. You can see the relationships between common services and ports by looking in the SERVICES file (search your WINDOWS or WINNT directory to locate this file).

Finally, we come to the terms *host names* and *IP address*. An IP address is a 32-bit value, typically displayed as a group of four numeric values, `165.212.210.41`, for example. The IP address identifies the network and a particular machine running on that network (whether client or server). Since an IP address is often difficult to remember, *host names* are used to describe a particular IP address. The host name is simply a string that refers to a particular IP address. For example, the host name `www.bridgespublishing.com` is mapped to the IP address `165.212.210.41`.

There is one other aspect of socket connections that you need to be aware of. A socket can establish a connection in one of two ways, *blocking* and *non-blocking*.

In a blocking connection, the receiver is blocked while waiting for the message sent by the sender. As such, it can't do anything else until the expected message arrives. In this way, blocking connections are synchronous operations.

Non-blocking connections, on the other hand, allow the receiver to go about its business while waiting for a message from the sender. When the message arrives, an event signals the receiver of the incoming

data. Non-blocking connections are asynchronous operations.

A blocking connection is preferred when using a precise protocol for data exchange. The synchronous nature of a blocking connection means that the normal I/O of an application is halted while the operation is being carried out. A non-blocking connection is more flexible and a bit simpler to use. For this reason, I will concentrate only on non-blocking connections in this article.

Step 1: Establishing a connection

Before the client begins to send data to or receive data from the server, it needs to establish a socket connection. Obviously, if the server isn't available, the client cannot connect to anything. As a first step, I'll explain how to create a server. Later I'll show how a client can connect to that server. Along the way I'll introduce you to the socket components that ship with the Client/Server and Enterprise versions of C++Builder.

Writing the server application

When writing a server application, you use the `TServerSocket` component (located on the Internet tab of the Component palette). This component has a number of properties and methods that allow you to activate and manage connections. The primary properties are:

- `Port`
- `Service`
- `Active`

The `Port` and `Service` properties correspond to the previous explanation of ports and services. It is necessary to set one of these properties before opening a server; since the server needs to know where to listen for the client requests. After setting either of these properties, you set the `Active` property to `true` to start the server listening. Alternatively, you can use the `Open()` and `Close()` methods to start and stop the server. For example, suppose you want to open a server using port number `5000`:

```
// Shut down the server before
// changing the port value.
ServerSocket1->Close();
ServerSocket1->Port = 5000;
ServerSocket1->Open();
```

If the host name of the service is known (for example "MyOwnService"), you can safely exchange the second line of the previous code with this one:

```
ServerSocket1->Service = "MyOwnService";
```

When you activate the server, the socket opens a *listening connection*. This is a particular type of connection (also called a *half connection*), because the server isn't really connected with something. Instead, the server is ready and listening for client requests. Only when the server receives and accepts a client request does it create a new socket, a *server socket*. This socket will manage the connection with the specific client. The server will continue to listen on the original socket for additional client connection requests.

Once the server is up and listening it handles client requests as needed. I'll explain this further in a later section.

Writing a client application

Any client can attempt to connect to a listening server. For client applications, C++Builder provides the `TClientSocket` component. `TClientSocket` has many properties and methods in common with `TServerSocket`, because both are derived from the base class `TAbstractSocket.` `TClientSocket` has these primary properties:

- `Address`
- `Host`
- `Port`
- `Service`
- `Active`

As you can see, there are two primary properties not found in `TServerSocket`, `Address` and `Host`. These properties let you specify the server location by its IP address or its host name. In order to open a connection with a server, you need to specify at least one of these properties. Also, it's necessary to set the `Port` or `Service` properties to match the server. For example, if you want to connect a client to the server shown in the previous code, you would write code like this:

```
//Disable client connection
ClientSocket1->Close();
ClientSocket1->Address = "123.45.6.78";
ClientSocket1->Port = 5000;
ClientSocket1->Open();
```

Of course, if you know both server host name and service, you can use the following code:

```
ClientSocket1->Host = "www.server.com";
ClientSocket1->Service = "MyOwnService";
```

If the client finds a server at the specified address, and if the server is listening on the specified port, the connection is established. But, how does a server know that a client is trying to connect? Also, how does the client know that the server accepted the connection request? This information is presented to you in the form of VCL events.

Connection events

During a connection, both client and server components generate events so you can proceed with specific actions. There are two primary sets of events concerning socket components:

- Events that can occur when a client is trying to connect with the server.
- Events that take place during the connection.

I'll concentrate on the first set of events, from both the client and server point of view.

On the client side, there are three events that take place one after the other:

- `OnLookup`
- `OnConnecting`
- `OnConnected`

The first event occurs just after you set the client's `Active` property to `true`. At this point, you cannot modify the `Host`, `Address`, `Service`, and `Port` properties.

After, the socket is initialized and the server located, the `OnConnecting` event is generated. At his point it is possible to begin collecting information about the connection. If the server accepts the client connection request, the connection is completed and the `OnConnect` event is generated. You can start reading and writing data over the connection within your `OnConnect` event handler. If something goes wrong, the client socket generates an `OnError` event.

On the server side, things are a bit more complicated because the server has to take more actions than the client. The primary events that are generated for `TServerSocket` are:

- `OnListen`
- `OnGetSocket`
- `OnClientConnect`
- `OnAccept`

The `OnListen` event occurs just before opening the server. After this event is processed, the listening socket is ready to receive client requests.

When a client tries to connect with the server, the listening socket creates a new server socket, and generates the `OnGetSocket` event. This event is followed by an `OnClientConnect` event. At this point, the server can safely begin to send data over the connection. Finally, the `OnAccept` event is generated after the client request has been accepted.

Step 2: Transmitting data

Once the connection between client and server has been established, they can begin to "talk" to one another. Both `TServerSocket` and `TClientSocket` encapsulate a Windows socket object. This object is represented by the `Socket` property. For a client, it represents a *client socket* while, for a server, it represents a *listening socket*.

The `Socket` object provides several helpful properties and methods to gather information about the socket, and for writing and reading over the connection.

There is a common misunderstanding regarding the `Socket` property that needs to be clarified. Both classes use the same property name (`Socket`) for Windows socket objects. The two objects are quite similar, but in reality these socket objects are different. For a client socket, the `Socket` property is an instance of `TClientWinSocket`. For the server, it's an instance of the `TServerWinSocket` class.

These two classes are derived from a common base class, `TCustomWinSocket`. As such, they share many properties and methods. As you can imagine, this fact hides some important differences. The major difference concerns the listening socket. As I have said, when a new client connects with the server, the server creates a new socket. This socket will be in charge of communications with the particular client that requests the connection. The server's listening socket provides two important properties that maintain information about the active connections. Those properties are `ActiveConnections`, and `Connections`.

The `ActiveConnections` property is an integer value specifying the number of currently open client connections. The `Connections` property is an array of pointers to all the server sockets created. When the first client connects to the server, a new server socket is created, and its pointer is stored in the array. You can access properties and methods of the first server socket connection using:

```
ServerSocket1->Socket->Connections[0]
```

If the server accepts another connection request from a different client, a new server socket is created, and a pointer is stored in the next free index of the array.

Luckily, you don't normally have to select a server socket by index in order to interact with a specific client. The events generated during the connection will give you the pointer for the appropriate socket.

Gathering information

After the client has connected to the server, you can collect information about the new connection using the socket object's properties. The primary properties are:

- `LocalHost`
- `LocalAddress`
- `LocalPort`
- `RemoteHost`
- `RemoteAddress`
- `RemotePort`

The properties that begin with "Local" provide you with information about the local socket (the one you are actually using). Those beginning with "Remote" refer to the socket at the other end of connection. For a client application, you will write code to access the socket inside the `OnConnect` event handler. For a server application you can use `OnClientConnect.` Both events pass you a `Socket` parameter to provide access to the socket. For example, if the server needs to show that a new client is connected, you can use code similar to this:

```
void __fastcall
TForm1::ServerSocket1ClientConnect(
  TObject *Sender,
  TCustomWinSocket *Socket)
{
  Memo1->Lines->Add
    ("New client connected!");
  Memo1->Lines->Add("Client name is "
    + Socket->RemoteHost);
  Memo1->Lines->Add("Client address is "
    + Socket->RemoteAddress);
  Memo1->Lines->Add("Client port is " +
    (String)Socket->Port);
}
```

As you can see, the event handler provides you two parameters, `Sender` and `Socket`. The `Sender` parameter is a pointer to the listening socket, and the `Socket` parameter is a pointer to the server socket connected with the client. In this example, all references to the `Remote` properties refer to the client.

Client transmission

If a client application desires to send data to the server, it can use methods of the `Socket` property. The

two primary methods are `SendBuf()`, and `SendText()`.

`SendBuf()` requires two parameters. The first parameter is a pointer to the buffer containing some data, and the second parameter is the number of bytes of the buffer that must be sent. For example:

```
char buffer[10] =
  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
//Send the entire buffer
ClientSocket1->
  Socket->SendBuf(buffer, 10);
```

For non-blocking connections, `SendBuf()` sends data via the Windows' WinSock DLL. If WinSock has no more space to queue the data, `SendBuf()` returns `-1` and no data is queued. In this case you will need to wait for a period of time and try to resend the data.

A simpler approach is to use the `SendText()` method. This method requires only one parameter, an `AnsiString`:

```
AnsiString msg = "Hi, there!";
ClientSocket1->Socket->SendText(msg);
```

When the client receives a message from the server, an `OnRead` event is generated. You can deploy an event handler to grab the message, using one of these methods:

- `ReceiveLength()`
- `ReceiveBuf()`
- `ReceiveText()`

If you want to use the `ReceiveBuf()` method, you must first call `ReceiveLength()` to determine how many bytes are in the incoming message. `ReceiveBuf()` takes two parameters, a pointer to a buffer for receiving data, and a number of bytes to read. This code reads a message coming from a server:

```
void __fastcall
TForm1::ClientSocket1Read(
  TObject *Sender,
  TCustomWinSocket *Socket)
{
  char *buffer;
  int  buffer_len;

  // get the message length
```

```
  buffer_len = Socket->ReceiveLength();
  // create a new buffer big
  // enough to contain the message
  buffer = new char[buffer_len];
  // get the message
  if (Socket->ReceiveBuf(
    buffer, buffer_len) == -1)
  {
    ShowMessage ("No message received!");
    // free the buffer
    delete[] buffer;
    return;
  }

  // code here that uses the data

  // free the buffer
  delete[] buffer;
}
```

As I said earlier, the `OnRead` event handler provides you a `Socket` parameter. For the client, this `Socket` parameter points to the same object as the component's `Socket` property. `ReceiveBuf()` returns the number of bytes read, or, if no bytes are read, it returns `-1`.

The `ReceiveText()` method is even easier to use, as it returns an `AnsiString` object:

```
void __fastcall
TForm1::ClientSocket1Read(
  TObject *Sender,
  TCustomWinSocket *Socket)
{
  String message = Socket->ReceiveText();

  // use the message
  ...
}
```

Server transmission

Server transmission of data is similar to that of the client. The principal difference is that the `Connections` property of the listening socket object is used when you want to write to a specific client. Suppose you want to send a message to the first client. In that case you would write code like this:

```
ServerSocket1->Socket->Connections[0]
  ->SendText ("Are you there?");
```

If you want to broadcast a message to all clients, you can iterate over all active connections using the `ActiveConnections` property:

```
for (int i=0;i<SocketServer1->Socket->
      ActiveConnections;i++)
  ServerSocket1->Socket->Connections[i]
    ->SendText("Server shuting down");
```

When a server receives a message, an `OnClientRead` event is generated. A pointer to the server socket is passed as a parameter, so the server can respond directly to the client:

```
void __fastcall
TForm1::ServerSocket1ClientConnect(
  TObject *Sender,
  TCustomWinSocket *Socket)
{
  String ClientRequest =
    Socket->ReceiveText();
  if (ClientRequest == "Send me file X")
    SendFileX(Socket);
}
```

In this case, the `Sender` parameter represents the listening socket, while the `Socket` parameter indicates the server socket connected to the client.

Closing a connection

Eventually, you will need to terminate the connection. To terminate the connection from the client side, you simply call the `Close()` method (or set `Active` to `false`). Regardless of who closes the connection, an `OnDisconnect` event is generated at the client.

Calling the `Close()` method for a server socket results in all active connections begin closed, and the server will quit listening. To terminate a specific client connection, call the `Close()` method of that client (using the `Connections` property). When the client socket terminates the connection, the server receives an `OnClientDisconnect` event.

Conclusion

In this article I have explained the basics of communication between two or more applications through the use of sockets. In particular I showed you how to establish a connection between a client and a server, how to pass data between the two applications, and how to close the connection. As you can see, using sockets is not difficult once you know these basics. The best way to further understand this topic is to create a simple server and client and spend some time experimenting.

The code for this article consists of simple server and client applications. You can download the code from **www.bridgespublishing.com** and examine it to see the client/server relationship in action.

When to use __fastcall

by Kent Reisdorph

A certain amount of confusion exists among C++Builder programmers regarding the `__fastcall` keyword in VCL applications. This article will explain when to use `__fastcall`, and, more importantly, when not to use it.

Calling conventions

First I should say that `__fastcall` is the keyword used to specify the Register calling convention. A calling convention tells the compiler how it should call functions; how the function name will be exported, how the function's parameters should be passed, and who is responsible for cleaning up the stack (called function or calling function). **Table A** lists the calling conventions supported by C++Builder and their associated keywords.

Table A: Calling conventions and their keywords.

Calling convention	Keyword
C	`__cdecl`
Standard call	`__stdcall`
Register	`__fastcall`
Pascal	`__pascal`

The default calling convention for a C++Builder project is the C calling convention. You can see this by looking at the Advanced Compiler tab of the Project Options dialog. (I won't attempt to explain the details of calling conventions in this article. If you wish to learn more about calling conventions, search for "calling conventions" in the C++Builder help. You will find several help topics dedicated to this subject.)

The important thing to understand is that the default calling convention for a C++Builder project is the C calling convention, and that the default for the VCL is the Register calling convention.

__fastcall and the VCL

All functions in the VCL are exported using the Register calling convention. This is most obvious when you generate an event handler. Take the `OnClick` event handler, for example:

```
void __fastcall
TForm1::FormCreate(TObject *Sender)
{
}
```

It's obvious from looking at this declaration that the Register calling convention is being used (the `__fastcall` keyword proceeds the function name). If for some strange reason you were tempted to remove the `__fastcall` keyword from an event handler you would find that the application wouldn't even compile. It's a fact that `__fastcall` is required when dealing with VCL event handlers, and there is nothing you can (or should want to) do about that.

When to use __fastcall

C++Builder programmers (regardless of their experience level) often ask when they should use the `__fastcall` keyword. My response is always the same: Use `__fastcall` when you must, and never use it when you don't have to. That rather broad statement requires a bit of explanation.

As I have said, the C++Builder IDE automatically adds the `__fastcall` keyword for event handlers that it generates. This is as it should be, and you don't have to give it much thought. Sometimes, however, you will have to assign an event handler to a VCL event via code. In that case you will need to declare the event handler yourself and assign it to the event you are interested in handling. In this case you must use the `__fastcall` keyword when you declare the function.

Some VCL classes, such as `TList`, allow you to specify a callback function (a sort routine in the case of `TList`). You will have to use the `__fastcall` keyword in this case, too, as the VCL expects it.

That sums up when you *must* use `__fastcall` but it doesn't address those situations where use of this keyword is optional. Obviously, the IDE takes care of generating event handlers for you. In most applications, though, you will certainly have to add functions of your own to your main unit's class, and in supporting units. When declaring these functions you can use any calling convention you like. You might be tempted to use `__fastcall` when you declare these functions. After all, that is what you see in IDE-generated functions and it is common to use those function declarations as a guide. However, it is not required that you use `__fastcall` for your internal functions.

I never use `__fastcall` for my internal functions and my advice is that you don't either. In short, just declare your internal functions with no calling convention at all and let the compiler use the project options to determine the calling convention. The primary reason for this advice is that using `__fastcall` is simply not necessary. There is another reason, however, that I will explain in the

following section.

Is __fastcall actually fast?

If you look up calling conventions in the C++Builder help you will find that the Register calling convention specifies that function parameters should be passed in the CPU registers. (The other calling conventions pass function parameters on the stack.) Obviously, there are only so many registers available. The compiler will use registers if they are available, but otherwise will use the stack if no registers are available for passing function parameters.

At first glance it would appear that passing function parameters in registers would be much faster than passing parameters on the stack. Not long ago I received an email from a reader regarding use of `__fastcall`. The reader said that his tests indicated that the Register calling convention was actually slower than the default calling convention. Curious, I decided to conduct a test. I created a simple application that had four functions that were variations on the following:

```
int TForm1::GetValue(int x,int y,int z)
{
  return x + y + z;
}
```

Naturally, I declared each of the functions to use one of the four available calling conventions.

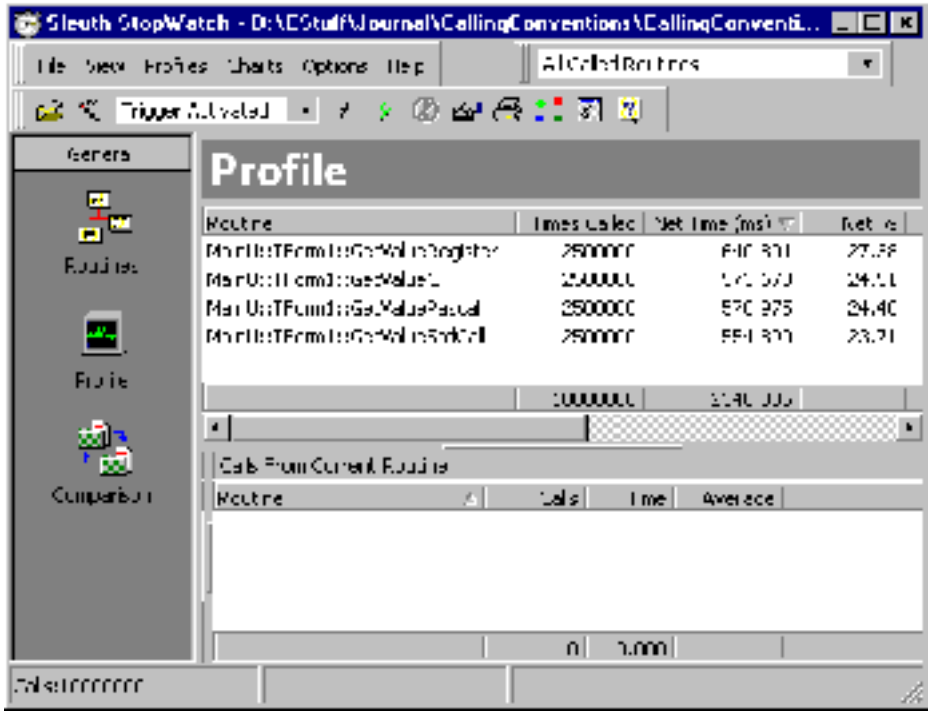
Next I added four buttons to the form, one for each of the calling conventions. The `OnClick` event handler for the button that calls the `__cdecl` version of the function looks like this:

```
void __fastcall
TForm1::CBtnClick(TObject *Sender)
{
  CBtn->Enabled = false;
  for (int i=0;i<500000;i++)
    int result = GetValueC(1, 2, 3);
  CBtn->Enabled = true;
}
```

As you can see, this code simply calls the `GetValueC()` function 500,000 times.

Finally, I ran the application under StopWatch, the profiler that ships with TurboPower's Sleuth QA Suite. (I could have used `GetTickCount()` to time the results, but the results would not have been nearly as accurate.) I clicked each button five times to get an average. **Figure A** shows a screen shot of StopWatch reporting the results.

Figure A



StopWatch shows that the Register calling convention is not particularly fast.

I found the results interesting to say the least. The C, Standard call, and Pascal calling conventions were very close in total execution time. Running the applications several times, I noticed slight differences in which of these three calling conventions produced the fastest results. Regardless of who came out the winner, they were all very close. What was most noticeable, however, was that the Register calling convention consistently produced the worst results. Roughly speaking, the Register calling convention proved to be about 10% slower than any other calling convention, at least when using the default project options.

In an attempt to be fair, I tried a variety of compiler options. I first changed the Optimization option to optimize for speed. This improved the performance of the Register calling convention somewhat, but not enough to matter (the Pascal calling convention was the loser in this particular test). Next I changed the Instruction set to Pentium and the Register variables option to Automatic. This made some difference as well, but again, not enough to matter.

The bottom line with my tests was that the Register calling convention was not any faster than the C calling convention and was, in most cases, slower. Granted, the differences in time are miniscule but I proved to myself that it would be difficult for proponents of `__fastcall` to claim that the Register calling convention is faster than any other.

Conclusion

To repeat what I have already said, I advise that you never use `__fastcall` unless it is specifically required by the VCL. Using `__fastcall` isn't necessary, it clutters up your code, and it appears to execute a bit slower than the default calling convention.

Designing with data modules

by Mark Cashman

Data modules are a specialized type of form descended from `TDataModule`. Normally, you place components that descend from `TDataSet` and `TDataSource` on a data module (`TTable` and `TQuery`, for example), but data modules can contain any non-visual component.

Data modules represent one of the most underutilized aspects of the VCL. This is not surprising when you consider that only a few lines are dedicated to them in the C++Builder Developer's Guide. In my first year of working with C++Builder, I completely ignored data modules, piling datasets and data sources on my forms as needed. I would guess that most developers initially take that same approach. In the last few years, however, I have discovered a powerful set of design patterns emerging from increased use of data modules. I will explain those design patterns in this article.

Basic data modules

A basic data module is a container for datasource components such as `TTable` and `TQuery`, and for their associated `TDataSources`. It provides a convenient place to put datasets that would otherwise clutter a user interface form at design time.

The basic data module, though, has a more important purpose. It allows you to decouple your data from the view of the data provided by the user interface. This means that the same data module can be used by multiple user interfaces, or to perform processing without the involvement of a user interface (for instance in an NT server or a non-visual DCOM, Web, or CORBA server).

Within the basic data module, event handlers associated with datasets and data sources can be used to enforce consistency (preventing deletion of data from a table when the ID of the row to be deleted is referenced from a row in another table), to constrain the availability of data to the user interface (by applying a filter to one dataset based on some field in the current row of another dataset), or to trigger changes to data in other datasets (to update a summary dataset after a change to a detail dataset, for example). And, of course, custom datasets can be developed to provide dataset-like access to non-dataset sources of data (real time data monitors or processes communicating with sockets from remote systems, for example).

Once you make the transition to the basic data module, you need to follow certain rules to gain the greatest benefits from the approach.

First, never access elements of any user interface from a data module. Doing so couples the data module to the specific form and prevents it from being used with some other form or for use without a form. You can, and should, access elements of related data modules as needed, but be aware that this couples those data

modules and requires that they be used together in any project

Second, try to connect user interfaces to the data module mostly through data aware controls attached to data sources. If need be, create new data aware controls for your user interface. For instance, if you need a record count label, create a data-aware descendant of `TLabel` that interrogates the dataset of its data source for the record count when a data change event occurs. Don't have the form programmatically interrogate a data module dataset and definitely don't have a data source in the data module reference a label on a form to update it with the current record count. If you need user interface related processing associated with a `TDataSource` event (such as `OnDataChange`), either create a data aware non-visual control for the form and associate it with the data source, or add a data source to the form and attach it to the dataset in the data module.

Third, keep "business rules" and processing logic in the data module. Only methods highly specific to a user interface should be placed in a form. This maximizes the reusability of the data module and keeps processing close to the data, where anyone maintaining your code will expect to find it.

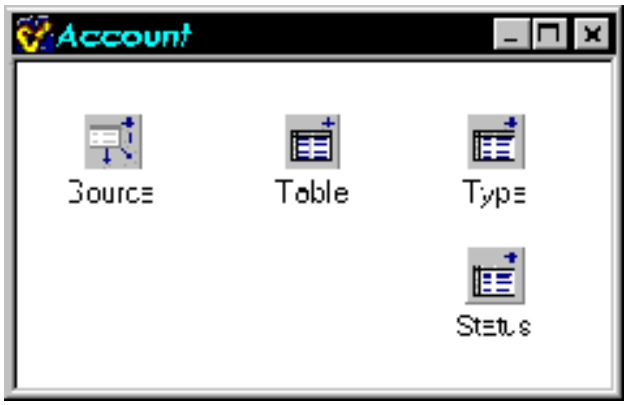
Finally, name your data module, data sources, and datasets carefully and consistently. This helps ensure that event handlers and user interface logic are easily readable and traceable to specific data modules and datasets. Use persistent fields on your datasets to minimize the need for the user interface to know about the internal structure of the data module.

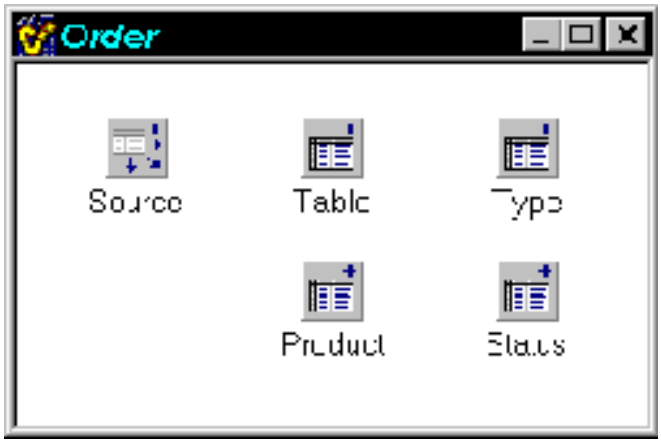
Using basic data modules to partition data

A data module should be more than a pile of datasets and data sources needed by your current application. Properly composed, a basic data module is a cohesive unit of related datasets that reflects your database design. This makes it easier for others to understand how your system relates to the underlying database schema should they need to maintain your code.

Take a typical account/order relationship in a database, for example. The two data modules shown in **Figure A** illustrate how you might implement this relationship.

Figure A





An account/order database might be implemented with data modules like these.

Notice how the data module controls the scope of names so that you can reference both `Order->Table` and `Account->Table`. This kind of standard naming makes it easier for a programmer maintaining your code to understand a data module and its logic.

Here again, there are some rules that should be followed to make this work. The first rule is that `Table` and `Source` are the dataset and data source for the primary table and are used in any user interface that edits the table.

The second rule is that the domain tables should have an associated data source if they can make an appearance in the user interface (for editing).

There are few data modules that do not need to reference data from some other data module. For instance, an order references its account for information like account name and address. Often this information is accessed via persistent lookup fields (like `Order->TableAccountName`).

Such a lookup field can reference the other data module's data sets. The advantage to this is that the other data module retains responsibility for its data. However, the disadvantage is the coupling between the two data modules, which means that the Order data module cannot be used in a project without also including the Account data module.

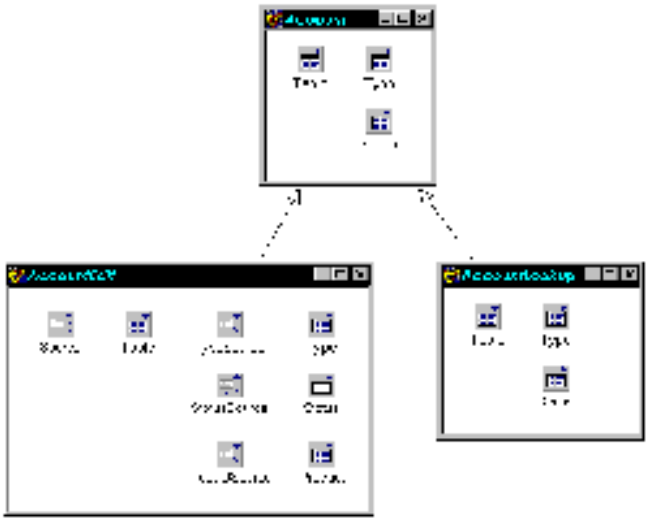
You may be tempted to include a data set in the Order data module to represent the account table for the lookup fields. Doing so, however, eliminates the coupling at the cost of preventing the application of appropriate business rules to the data set, since only the Account data module should contain code for those business rules. In some cases that may not be a problem, but when it is, there is another way to approach the issue; form inheritance.

Form inheritance and data modules

Form inheritance allows you to create a descendant of your data module and to augment the ancestor class

with enhanced event handlers, additional components, or changes to the properties of inherited components. To use form inheritance, first create the base data module. Next, right click on the data module and select "Add To Repository" from the context menu. Give the data module a name and put it on a new page in the Object Repository (I use the data module name itself for the page name). Now choose File|New from the main menu and inherit a new data module from the data module you just added to the repository. Add components or make other changes in the inherited data module as needed. If you create an event handler that overrides a base class event handler, don't forget to call that inherited event handler where appropriate. This can be used to deal with the lookup problem in an especially elegant way. **Figure B** shows an inheritance tree of data modules that deals with the lookup problem.

Figure B



Form inheritance can be used to solve programming problems.

Note that the base class does not have any data sources. The data sources are only introduced in the "Edit" leaf of the inheritance tree because the lookup leg will generally not be used in a user interface. That may not always be true, though. For instance, `TDBLookupCombos` should be linked to data sources in the lookup data module. This may cause you to decide to place data sources in the base class as the need arises to have them available for both edit and lookup, rather than introducing them in the descendant classes.

At any rate, the answer to the question posed in the previous section is that the Order data module lookup fields references the datasets in the lookup data module. Further, any `Locate()` operations should also use the lookup data module.

What about iterating through a table? Generally, you should be able to use the lookup data module for this purpose, unless you intend to edit the data as part of the iteration. In that case, the edit data module should be used to ensure that integrity checks and business rules are maintained. In a few cases, you may need to derive a third leaf class to be used for iteration.

Note that the leaf classes are the actual instances used in a project. By the nature of form inheritance, only

one instance of a non-leaf class can be present in a project at design time.

Some developers may be tempted to instantiate multiple copies of what should be a base class data module at runtime to provide a lookup table. However, the advantage of the scheme described here is precisely that it allows design time inspection and modification of the various data modules, and a thoughtful partitioning of function between them. For instance, the base data module implements tables, fields, and handlers common to both editing and lookup. The edit data module adds components and fields, and overrides handlers to produce edit specific behavior. If a different behavior is needed for the lookup module, that can be implemented separately.

One last note: Multiple cursors into the same physical table require a refresh of all cursors when an insert or update is performed on one. For `TTable` descendants a call to `Refresh()` is sufficient. For `TQuery` descendants, the query must be closed and then opened again. Usually, I define a `TQuery` descendant which implements `Refresh()` in that way, since the `Refresh()` method has no effect for `TQuery`. Effectively supporting refreshes at runtime across multiple cursors generally requires you to find which tables and queries in the `Sessions->Databases->Database->DataSets` list reference the physical table just updated. The method for doing so is beyond the scope of this article, but is discussed at **http://www.temporaldoorway.com/programming/cbuilder/databaseandbde/scanningtorefresh.htm**

Avoiding commitment to a dataset type

You may be designing a system where you are not sure what dataset type may be used to connect to the database, or where one version of the system may use a `TTable` attachment to a desktop data base, another version may use a `TQuery` to work with a client server data base, and a third version may use client datasets to work with a multi-tier system. You can exploit form inheritance to solve this particular problem.

The base class data module should contain only data sources. The user interface should connect to the base class data sources. Programmatic references to the datasets from other data modules or the user interface should be of the form:

```
DataModule->DataSource->DataSet
```

Persistent fields for the datasets that will be added should be included in the class definition of the base class data module. This allows the user interface to reference the base class instances of those fields.

Each descendant of the data module then adds the actual dataset components, associates the data sources with the datasets, and creates the persistent field instances from the actual datasets programmatically. The descendent data module also initializes the base class global variable for the data module with `this`. This can also be useful when some descendants may use non-BDE dataset components, as well as with different types of BDE dataset components, as shown in **Figure C**.

Figure C

Inheritance can be used to create a data module that can obtain data from varied data sources.

Note that a descendant of the base class can even make the decision about which type of dataset to establish at runtime, and can dynamically create and link datasets of the desired type based on user input, a registry setting, or some other indicator.

Note also that this can lead to “uneven form inheritance” which requires a special fix in your program source, as documented in
http://www.temporaldoorway.com/programming/cbuilder/advancedissue/unevenforminheritance.htm

Adding design time properties to data modules

Most C++ Builder developers have heard that adding properties to data modules is difficult. Indeed, it seems to require producing a Wizard and a Form Designer, something that is not directly productive for working developers looking to complete a project. Fortunately, there is a much simpler and better way—components.

Generally speaking, components encapsulate both data and behavior, offering design time properties that affect behavior. However, you can use a non-visual component to add design time properties to data modules.

To illustrate, I have created a component called `SamplePropertyComponent`. The header for this component is shown in **Listing A**. As you can see, this component is quite simple. It also allows use of the component’s properties from within the data module with this code (assuming I gave the component a name of `Property`):

```
TDataSet *CurrentTable =
  Property->PrimaryTable;
```

The component can be referenced from outside the data module like this:

```
TDataSet *CurrentTable =
  Account->Property->PrimaryTable;
```

Either of these notations in the right context can be as clear as directly specifying access to a data module property:

```
if (AccessorCount ==
    Account->Property->
      NumberOfAccessorsAllowed)
```

Other non-visual components in data modules

There's no reason to limit data modules to contain only dataset-oriented components. You can have data modules that contain nothing but other non-visual components.

Such components can include any combination of the following:

- Data module property components as described in the previous section.
- Components for access to hardware such as serial ports, parallel ports, or more specialized hardware.
- Components for access to non-visual resources like non-database files.
- Components for access to the registry. Registry-oriented components can be constructed so that each one offers direct access to a specific registry key through a property which, when read, gets the value from the registry and when written writes the value back to the registry.
- Components whose purpose is to allow the developer to select another component of interest at design time. Program code can access the other component indirectly through this "redirection" component.
- Components that encapsulate threads.
- Components that provide processing services via event handlers that are established and augmented by form inheritance (explained in the accompanying article, "Exploiting data module form inheritance").

Conclusion

One of the virtues of RAD in general, and the C++ Builder component approach in particular, is to shift system structure from implicit expression in code to explicit expression in the arrangement and linking of components. The addition of event handlers and form inheritance makes C++Builder an exceptional platform for advanced system development. The resulting software can be very powerful, abstract, and self-documenting, while offering outstanding potential for reuse.

Listing A: *The header for SamplePropertyComponent*

```
#ifndef SamplePropertyComponentH
#define SamplePropertyComponentH
```

```
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>

class PACKAGE SamplePropertyComponent
  : public TComponent
{
  private:
    TDataSet *myPrimaryTable;
    int myNumberOfAccessorsAllowed;
    String myComment;
  public:
    __fastcall SamplePropertyComponent(
      TComponent* Owner);
   __published:
    __property TDataSet *PrimaryTable =
      {read=myPrimaryTable,write=myPrimaryTable};
    __property int NumberOfAccessorsAllowed =
      {read=myNumberOfAccessorsAllowed,
       write=myNumberOfAccessorsAllowed};
    __property String Comment =
      {read=myComment,write=myComment};
};

#endif
```

Dragging images with TImageList

by Kent Reisdorph

As its name implies, the `TImageList` component allows you to manage a list of images. What many people don't know is that `TImageList` also has a built in mechanism for dragging images on the screen. You can see this in action if you run any program that has a list view and which allows dragging of the list view items. It is not the list view that draws the drag image, but rather the list view's internal image list.

This article will explain how to use `TImageList`'s built-in dragging mechanism to move images about on a form. When I speak of dragging I am, of course, referring to images dragged with the mouse. However, dragging goes beyond use of the mouse. You can programmatically "drag" images on the screen to provide animation, with no user intervention required.

Dragging images

`TImageList` has four methods that facilitate image list dragging. Those methods are `SetDragImage()`, `BeginDrag()`, `DragMove()`, and `EndDrag()`.

Before you can drag an image, you must set that image as the image list's drag image. This is done with a call to `SetDragImage()`. Here is an example:

```
ImageList1->SetDragImage(Index, 0, 0);
```

The first parameter is used to specify the index number of the image in the image list that will be dragged. Images are zero based, so the index of the first image is `0`, the index of the second image is `1`, and so on. The second and third parameters are used to specify the X and Y coordinates of the hotspot. The hotspot is the location within the image where subsequent dragging operations will be based.

Once you have set the drag image, you initiate a drag sequence by calling the `BeginDrag()` method:

```
ImageList1->BeginDrag(Handle, X, Y);
```

The first parameter is the window handle of the window that will host the drag operation (contrary to what the VCL help for `BeginDrag()` says). The drag image will only be displayed within that window, and will be constrained to that window. You can pass any valid window handle for this parameter, including your form's `Handle` property. The second and third parameters are the X and Y coordinates of the drag position. You can think of these coordinates as the origin of the drag image. If your program supports dragging an image using the mouse, then you would call `BeginDrag()` in your

`OnMouseDown` event handler.

Now you can call `DragMove()` to move the drag image around on the screen. (More accurately, `DragMove()` will cause the drag image to be displayed on the window whose handle you passed when you called `BeginDrag()`.) Calling `DragMove()` is trivial:

```
ImageList1->DragMove(X, Y);
```

Obviously the parameters passed to `DragMove()` are the X and Y coordinates of the new drag position.

You can call `DragMove()` in response to mouse movements (such as in the `OnMouseMove` event handler) or you can provide some routine to programmatically move the drag image in order to provide some sort of animation.

Once the drag operation is complete, call `EndDrag()` to terminate the drag operation:

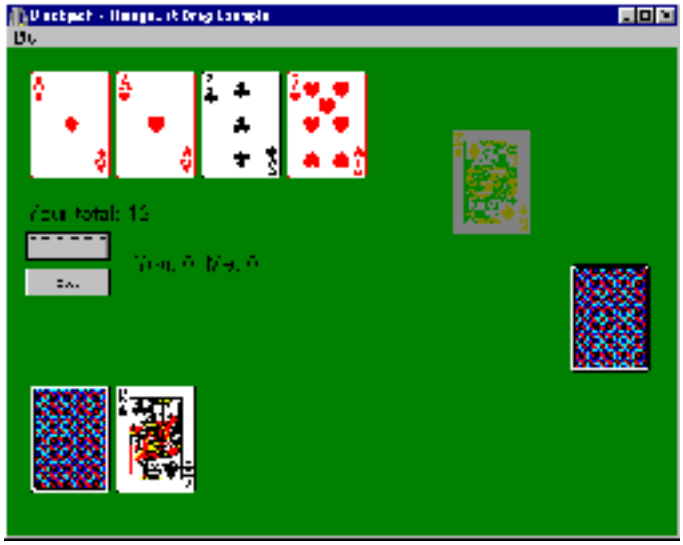
```
ImageList1->EndDrag();
```

As you can see, calling `EndDrag()` is simple and doesn't require further explanation.

The article's example program

The example program for this article is a simple implementation of the popular card game, Black Jack. As the cards are dealt, they are animated using the techniques discussed in this article. In addition, the player can drag his or her cards around on the screen. We don't provide the code listing for the example program here because so much of the code is unrelated to this article's topic, but you can download the example from our Web site at **www.bridgespublishing.com**. **Figure A** shows the example program running.

Figure A



Dragging images is easy with TImageList..

I should mention that the example program uses a modified version of the CARDS.DLL that ships with Windows. The version that ships with Windows 95 and 98 is a 16-bit DLL and cannot be used with C++Builder. The modified version, CARDS2.DLL, is a 32-bit version is provided for testing purposes only. The images in CARDS2.DLL are copyrighted by Microsoft and, as such cannot be distributed with your applications.

Conclusion

Dragging images, whether programmatically or using the mouse, is fairly simple using the functionality provided by `TImageList`. You may be able to benefit from the ability to drag image list items, regardless of whether your program is a game or a business application.

Exploiting data module form inheritance

by Mark Cashman

I the article “Designing with data modules,” I explained how form inheritance can be used to separate edit and lookup access to a database, and to defer or allow inheritance structures that implement different decisions about the type of datasets to be used.

Inheritance can also be used as a more comprehensive system-structuring technique. Such a technique is based on the ability to override and augment event handlers in descendant forms.

As you may know, descendant forms can contain components not present in the base form class. In addition, property values can be changed. However, it is not possible to replace a component with a descendant. So component inheritance cannot be used to augment the data module, except insofar as it produces additional components to be added to the descendant data module.

This problem can be overcome through the use of event handlers. If your base class component is designed as a framework to hold event handling logic, then descendant data modules can augment those capabilities. Indeed, the base data module can simply contain the components without any event handling logic, much as an abstract class contains abstract member functions to be implemented by descendants.

Let's use a concrete example to see how this might work. Imagine a data module that contains an incoming and outgoing communication component and a set of processing components that operate on a standard data structure. **Figure A** shows a data module that illustrates this type of arrangement.

Figure A

This data module is used for processing communications.

The data module uses two instances of a custom component class, `TCommunicationComponent`, that has an assignable data property, and an `OnAssign` event that is triggered by the setter function when data is assigned to that property. It also has an instance of a custom component class called `TProcessor`, which also exposes an assignable data property. It has a method called `Perform()` that does nothing but invoke an event handler for the `ToProcess` event.

The base class data module has logic in the `IncomingCommunication` component's `OnAssign` event that passes the assigned data along to the `Process` component and calls the `Perform()` method. The `ToProcess` event of the base class data module then does nothing but assign the data to the `OutgoingCommunication` component.

This framework can be extended in a number of descendant data modules. For example:

- A descendant data module that includes a `TServerSocket` which sets the `IncomingCommunication` from the incoming data on the socket, and implements the `OutgoingCommunication` `OnAssign` handler to send the processed data back to the originator via the same socket. Obviously, specific processing logic can be added to the `ToProcess` event handler of the `Process` component.
- A descendant that works with a form that assigns to the `IncomingCommunication` based on user input. The `OutgoingCommunication` `OnAssign` handler then updates the form directly with the results.
- A descendant which uses an `NMHTTP` component to get and send communications and which translates the http data to/from the standard data structure with the help of a table. The `OnAssign` for `OutgoingCommunication` calls upon the `NMHTTP` component to send the data back to the requestor.

This technique has the advantage of making a framework explicit at design time, of offering customization points (sometimes called *hinges*) in the form of event handlers, and of offering opportunities to support specialized event handlers with other components introduced in descendant classes.

Obviously, you can start with more or less specialized frameworks in the base class, and expand them to be as complex and elaborate both horizontally and vertically as needed.

Making marvelous message dialogs

by Stephen Posey

The Windows API provides a basic means of obtaining information from the user in the form of the `MessageBox()` API call. Calling `MessageBox()` produces a simple dialog box containing a text message and from one to four buttons which the user can select. Generally the calling program's message loop stops while the message box is displayed (the dialog is modal).

`MessageBox()` has its uses, but it suffers from a somewhat archaic look (the `MessageBox()` API call hails all the way back to Windows 1.0!) and has a very limited number of options regarding the buttons shown and the overall appearance of the resulting dialog. So, what's a poor programmer to do if the options provided by `MessageBox()` are inadequate or inappropriate for his/her information gathering requirements? This article will explain some of the built-in VCL message-display functions and shows how to create your own unique message boxes.

VCL message box functions

Of course, when using a RAD tool like C++Builder, it's always possible (and relatively painless) to design a custom form from scratch or from a template to produce the desired effect. Even with RAD capabilities, though, it can still be a lot of work to create individual forms if you have a large variety of message dialogs. Coming up with a generic mechanism also involves a fair bit of work, especially when you find out you'd be re-inventing the wheel (as you will soon see).

Step into the VCL and its `MessageDlg()` family of functions. The Dialogs unit of the VCL exports these functions. If you have not yet looked at them you owe it to yourself to do so as they can make your information gathering chores remarkably easier. (Obviously, the `MessageDlg()` source is written in Pascal, but examining how the dialog functions are implemented is a great tutorial in dynamic form and component creation, not to mention potentially giving further ideas for customization as shown by this article.)

One of the functions available is `MessageDlg()`. `MessageDlg()` is quite similar to `MessageBox()` in its overall operation, but the "look" is somewhat different. In addition, you have more options regarding the buttons to be placed on the dialog. There are also variants that allow positioning of the dialog on the screen (`MessageDlgPos()`), and one that includes help context information (`MessageDlgPosHelp()`). Using `MessageDlg()` is pretty straightforward:

```
MessageDlg("Hello World!",
  mtError, TMsgDlgButtons() << mbOK, 0);
```

The first parameter is the message text. The second parameter is an enumeration value that indicates the icon that appears on the dialog. The third parameter is a set of `TMsgDlgButtons()` that determines the buttons that appear on the dialog. The final parameter is for an optional help context ID (generally set to 0). `MessageDlg()` will return the ID of the button that was selected as one of the `mrXXX` constants (see the `ModalResult` topic in the VCL help for a list of these constants).

If you're only in need of getting a quick and dirty text message to the user (yourself when you're debugging, for example), this same family includes the incredibly simple `ShowMessage()` function. `ShowMessage()` takes just one parameter—the text of the message to show. (The `ShowMessagePos()` function provides a variant that allows you to position the dialog.)

The Dialogs unit also exports two functions that provide simple data entry forms. `InputBox()` and `InputQuery()` return strings entered in edit boxes on the forms. Since these are generated in a different way from the others they aren't of direct interest here (though certainly useful in their own rights). The following code shows an example using a combination of `InputBox()` and `ShowMessage()`:

```
String S = InputBox("Name Entry Form",
 "Please enter your name", "");
if (S != "")
  ShowMessage("Hello " +
    S + ", how are you today?");
```

A better message box

Having access to these fancier replacements for `MessageBox()` is all well and good, but sometimes it's still necessary or desirable to show a message dialog, some part of which doesn't conform to one of the pre-defined solutions provided by `MessageDialog()` and its cousins.

So, then, are we back to creating Forms from scratch again? Fortunately not, because Borland in its (remarkable in this case) foresight and wisdom, has provided access to the matriarch of the `MessageDialog` clan, the `CreateMessageDialog()` function.

All of the VCL dialog functions rely on `CreateMessageDialog()` for instantiating the basic model of dialog that they all share. You can do the same.

The `CreateMessageDialog()` function returns an instance of the `TForm` class (the same class that is ancestor of all VCL forms). The instance is the form as built using (and thus appearing) essentially the same as the forms produced by the standard functions. With the form instance in hand you can perform all kinds of magic on the dialog before it is displayed.

Here is the prototype for `CreateMessageDialog()` (I've taken the liberty of removing the namespace qualifiers for clarity):

```
TForm* CreateMessageDialog(
  const AnsiString Msg,
  TMsgDlgType DlgType,
  TMsgDlgButtons Buttons);
```

The `Msg` parameter is the message text that will be displayed on the dialog. `DlgType` is an enumeration that determines which of the several types of standard dialogs is shown (see the on-line docs if you need more details on this). `Buttons` is a set of the desired buttons that will appear on the form.

To create custom effects in a message dialog, the basic process goes like this:

- Provide an instance variable of type `TForm*`
- Call `CreateMessageDialog()` and assign its return value to the `TForm*` variable
- Access the form's properties or individual components on the form

Once you have the `TForm*` for the dialog, you can locate and modify the components owned by the form to fit the needs at hand. You can even change characteristics of the form itself! After modifying the form or its components, you show the dialog by calling the `ShowModal()` method. When `ShowModal()` returns, free the `TForm` instance and proceed normally.

As a simple first example, **Listing A** shows how you can use this method to create a message dialog with custom button captions. It also shows how you might respond according to the return value of the message dialog. For this simple example the intermediate `TButton` instance variables I used are extraneous, but using them like this is a nice segue into more complex examples.

Getting at the form's components

The method for accessing the individual buttons requires some explanation. When the form instance is created, it adds as many buttons as requested. It assigns a value to each button's `Name` property equivalent to the associated `TMsgDlgButton` constant, without the "mb" in front. If you specified `mbYes` for the dialog buttons, for example, then the Yes button's `Name` property will be "Yes."

One caveat with regard to the buttons; they are sized according to the length of the original captions. If you give the buttons new captions, it's easy to make the captions overrun the available space. Once you have a reference to a button you can, of course, move or resize it, but that takes a bit of work to get right.

Similarly, the message text is a `TLabel` with a `Name` of "Message", and the dialog's icon is a `TImage` with the name "Image."

You can get a pointer to a particular component on the form using the `FindComponent()` method. Once you have the pointer, you can access and manipulate the component using any of its properties and methods. For example, you can do things like change the fonts of individual buttons or of the message, or change the image and location of the dialog's icon. At the form level you can modify the dialog's title bar caption, change the color of the background, position the form, or even add a bitmap background to the form.

Listing B goes a bit overboard in showing some of the possibilities. As I mentioned earlier, it's easy to change the captions to something that no longer fits (like the "All Please" button in **Listing B**). You'll have to be careful not to exceed the original button width or you'll have to write code to move and resize the buttons. I use `ExtractIcon()` to get the icon for Windows Notepad and display that icon on the dialog. I've hard coded the path to NOTEPAD.EXE so you'll have to change the path if you run this code.

Listing C is a set of several new `MessageDlg()` style functions. The first simply adds the ability to change the dialog's caption. Note that I've set it up so that passing an empty string results in the default caption being displayd.

The second function is an enhancement of the simple `ShowMessage()` function, but allows you to set the caption and to show more than just the OK Button provided by default (since this is a `void` function and doesn't return the modal result, picking a button other than OK may be of limited usefulness).

The third function lets you specify an icon (via a `TIcon` instance) to replace the icon on the standard dialog. Passing `0` for the `TIcon` parameter causes the function to use the standard icon, much as passing an empty string does for the message dialog's caption.

Of course you can mix and match these features in your own versions to suit the needs of your applications. I don't include examples of calling these new functions, but you can download the source for the example program from the Bridges Publishing Web site.

Conclusion

Using the techniques presented in this article, you can easily create customizable informational dialogs. Your display and information gathering chores will surely be a lot easier.

Listing A: *A simple CreateMessageDialog() example*

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
```

```
  TForm* Dlg = CreateMessageDialog(
    "Purge Warp Core?", mtConfirmation,
    TMsgDlgButtons() << mbYes << mbNo);
  TButton* yb = dynamic_cast<TButton *>
    (Dlg->FindComponent("Yes"));
  if (yb)
    yb->Caption = "Affirmative";
  TButton* nb = dynamic_cast<TButton *>
    (Dlg->FindComponent("No"));
  if (nb)
    nb->Caption = "Negative";
  int Rslt = Dlg->ShowModal();
  switch (Rslt) {
    case mrYes: ;// do "Yes" stuff
    case mrNo:  ;// do "No" stuff
  }
}
```

Listing B: *This example modifies several components on the message dialog*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TForm* Dlg = CreateMessageDialog(
    "My Message", mtConfirmation,
    TMsgDlgButtons() << mbYes << mbNo << mbAll);

  // change the caption
  Dlg->Caption = "Please Confirm";

  // shift the position
  Dlg->Top =+ 300;
  Dlg->Left =+ 300;

  // Get the button instances
  TButton* yb = dynamic_cast
    <TButton *>(Dlg->FindComponent("Yes"));
  TButton* nb = dynamic_cast
    <TButton *>(Dlg->FindComponent("No"));
  TButton* ab = dynamic_cast
    <TButton *>(Dlg->FindComponent("All"));

  // change the button texts
  yb->Caption = "Indeed!";
  nb->Caption = "Surely Not!";
```

```
  ab->Caption = "All Please!";

  // change the button fonts
  yb->Font->Name = "Times New Roman";
  nb->Font->Name = "Arial";
  ab->Font->Name = "Courier New";

  // change the Message's appearance
  TLabel* lb = dynamic_cast
    <TLabel *>(Dlg->FindComponent("Message"));
  lb->Font->Name = "Courier New";
  lb->Font->Size = 16;
  lb->Font->Color = clRed;

  // supply a custom icon
  TImage* img = dynamic_cast
    <TImage *>(Dlg->FindComponent("Image"));
  img->Picture->Icon->Handle = ExtractIcon(
    HInstance,"C:\\WIN95\\NOTEPAD.EXE", 0);

  // change its position
  img->Left = 200;
  img->Top = 15;

  int Rslt = Dlg->ShowModal();

  switch (Rslt) {
    case mrYes: ;// do "Yes" stuff
    case mrNo:  ;// do "No" stuff
    case mrAll: ;// do "No to All" stuff
  }
}
```

Listing C: *Customized message dialog functions*

```
int MessageDlgCaption(
  const AnsiString Caption, const AnsiString Msg,
  TMsgDlgType DlgType, TMsgDlgButtons Buttons)
{
  TForm* dlg =
    CreateMessageDialog(Msg, DlgType, Buttons);
  if (Caption != "")
    dlg->Caption = Caption;
  return dlg->ShowModal();
```

```
}

void ShowMessageCaptionBtn(const AnsiString Caption,
  const AnsiString Msg, TMsgDlgBtn Btn)
{
  TForm* dlg = CreateMessageDialog(
    Msg, mtCustom, TMsgDlgButtons() << Btn);
  if (Caption != "")
    dlg->Caption = Caption;
  dlg->ShowModal();
}

int MessageDlgIcon(const AnsiString Caption,
  const AnsiString Msg, TIcon* Icon,
  TMsgDlgType DlgType, TMsgDlgButtons Buttons)
{
  TForm* dlg =
    CreateMessageDialog(Msg, DlgType, Buttons);
  if (Caption != "")
    dlg->Caption = Caption;
  if (Icon) {
    TImage* image = dynamic_cast<TImage *>
      (dlg->FindComponent("Image"));
    if (image)
      image->Picture->Icon = Icon;
  }
  return dlg->ShowModal();
}
```

Storing binary data in the registry

by Kent Reisdorph

Saving user settings in the registry is common fare for most Windows programs today. Just take a look at the C++Builder entries in the registry and you'll see what I mean. You may already be using the registry to store user configuration data. It's fairly easy to store integer values or strings in registry keys. Storing binary data is just as easy, but is something that is often overlooked by C++Builder programmers.

This article will explain how to store binary data in the registry. To illustrate, I will show how to store the font attributes of a `TFont` object to the registry and read it back again.

Writing binary data to the registry

Writing binary data to the registry is accomplished by calling `TRegistry`'s `WriteBinaryData()` method. Let's take the case of `TFont` for example. Users on the Borland newsgroups frequently ask how to save a `TFont` to a file or to the registry. The key to saving and retrieving any binary data is to start with a fixed-length structure. If you have a fixed-length structure you can write exactly that many bytes to the binary stream and later, when you want to retrieve that data, you read exactly that number of bytes again. Windows provides such a structure for font data in the `LOGFONT` structure (this structure also has a corresponding VCL version called `TLogFont`). I should mention that the `LOGFONT` structure doesn't know anything about a font's `Color` property. If you want to store a font's color in addition to it's other attributes (size, typeface name, style, etc.) then you'll have to write the font color to the registry in a separate step.

The first step is to get the font data from a `TFont` object and convert it to a `LOGFONT` structure. This is achieved by calling the API function `GetObject()`, as shown here:

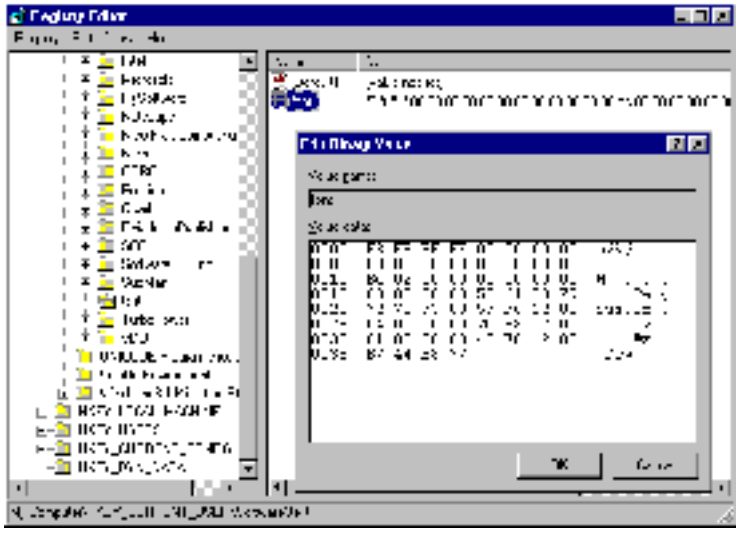
```
LOGFONT lf;
GetObject(Font->Handle, sizeof(lf), &lf);
```

`GetObject()` takes the font specified by the `TFont` object's `Handle` property and returns its font data in a `LOGFONT` structure. Now that you have the font data in a fixed-length structure, writing it to the registry is a trivial matter:

```
TRegistry* reg = new TRegistry;
reg->OpenKey("\\software\\test", true);
reg->WriteBinaryData('font', &lf, sizeof(lf));
delete reg;
```

That's all there is to it! Windows and the `TRegistry` class do all the hard work behind the scenes so you don't have to worry about it. **Figure A** shows the Windows Registry Editor displaying the binary data created by the preceding code.

Figure A



The Registry Editor shows the binary font data created by the example program.

Reading binary data from the registry

OK, so you've stored the font's data in the registry. Naturally you have to retrieve that data from the registry in order for it to be useful. Reading binary data from the registry is also fairly trivial. Simply call the `ReadBinaryData()` method of `TRegistry` as this code illustrates:

```
LOGFONT lf;
TRegistry* reg = new TRegistry;
reg->OpenKey("\\software\\test", false);
reg->ReadBinaryData('font', &lf, sizeof(lf));
delete reg;
```

Since I stored a specific number of bytes to the registry key (`sizeof(LOGFONT)`) I simply read exactly that number of bytes again. The second parameter of `ReadBinaryData()` takes a `void*` so I simply pass the address of the `LOGFONT` variable in the second parameter, and the size of a `LOGFONT` structure in the third parameter. `TRegistry` loads the binary data into the memory location of the `LOGFONT` variable.

All that remains is to assign the just-retrieved `LOGFONT` data to the `Font` property of a given component (or the form itself). Here again, a little API code does the trick:

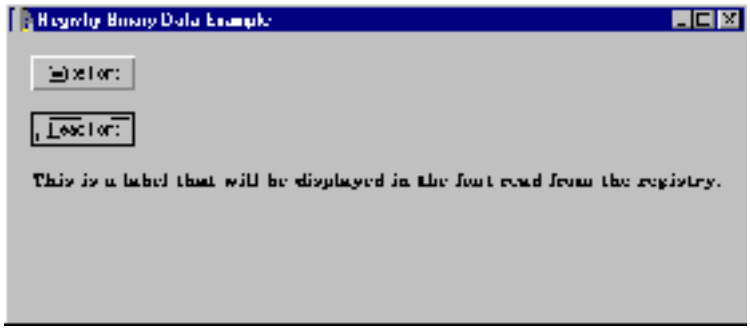
```
Font->Handle = CreateFontIndirect(&lf);
```

`CreateFontIndirect()` takes the data in a `LOGFONT` structure, creates a font from that data, and returns a font handle (and `HFONT`). You can assign the result of `CreateFontIndirect()` directly to a `TFont` object's `Handle` property. The VCL disposes of the old font handle, applies the new font handle, and updates the label with the new font.

Conclusion

You can store any type of binary data to the registry using `WriteBinaryData()` and read it back again using `ReadBinaryData()`. The key is to read exactly the number of bytes that you originally wrote. Saving user-configuration data to the registry in binary form is simple once you know how. **Listings A** and **B** show the header and main unit of a program that illustrates the technique discussed in this article. The program displays the font selection dialog on a button click and saves the selected font's data to the registry. A second button's `OnClick` event handler reads the font data from the registry and shows the results by applying the font to a label on the form. **Figure B** shows the example program running. You can download the example program from our web site at **www.bridgespublishing.com**.

Figure B



The example program writes a font to the registry, reads it back again, and applies the font to a label.

Listing A: *The main unit's header for the example program*

```
#ifndef MainUH
#define MainUH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Dialogs.hpp>
```

```
class TMainForm : public TForm
{
__published:    // IDE-managed Components
  TButton *WriteBtn;
  TButton *ReadBtn;
  TLabel *Label1;
  TFontDialog *FontDialog1;
  void __fastcall WriteBtnClick(TObject *Sender);
  void __fastcall ReadBtnClick(TObject *Sender);
private:    // User declarations
public:        // User declarations
  __fastcall TMainForm(TComponent* Owner);
};

extern PACKAGE TMainForm *MainForm;

#endif
```

Listing B: *The main unit for the example program*

```
#include <vcl.h>
#pragma hdrstop

#include <Registry.hpp>
#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;

__fastcall TMainForm::TMainForm(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
TMainForm::WriteBtnClick(TObject *Sender)
{
  // Show the Font dialog.
  if (FontDialog1->Execute()) {
    LOGFONT lf;
    // Create a TRegistry object.
    TRegistry* reg = new TRegistry;
    try {
```

```
      // Open a key in the registry. Create
      // the key if it doesn't yet exist.
      reg->OpenKey("\\software\\test", true);
      // Put the font data in a LOGFONT structure.
      GetObject(FontDialog1->Font->Handle,
        sizeof(lf), &lf);
      // Write the data to the registry.
      reg->WriteBinaryData(
        "font", &lf, sizeof(lf));
      ShowMessage("Registry key created.");
    }
    __finally {
      // Clean up.
      delete reg;
    }
  }
}

void __fastcall
TMainForm::ReadBtnClick(TObject *Sender)
{
  LOGFONT lf;
  // Create a TRegistry object.
  TRegistry* reg = new TRegistry;
  try {
    // Open the key created earlier.
    reg->OpenKey("\\software\\test", false);
    // Read the data directly into the
    // LOGFONT variable, lf.
    reg->ReadBinaryData("font", &lf, sizeof(lf));
    // Create a font from the LOGFONT data and
    // assign it to the label's Handle property.
    Label1->Font->Handle = CreateFontIndirect(&lf);
  }
  __finally {
    // Clean up.
    delete reg;
  }
}
```

Customized radio groups

by Stephen Posey

If you have ever had to create a group of radio buttons using old-style dialog resource techniques, you know how convenient C++Builder and the VCL make this task with the `TRadioGroup` component. `TRadioGroup` conveniently encapsulates a group box containing a programmer-specified number of radio buttons.

While the VCL encapsulation removes much of the drudgery associated with creating radio button groups, it also conceals the fact that the individual radio buttons are themselves members of a collection of `TRadioButton` objects maintained by the `TRadioGroup`. This article will explain how to access the individual items in a `TRadioGroup` in order to create radio groups that can be customized to provide additional features.

Better radio groups

Sometimes it's desirable to create a group of radio buttons that has more variety in its appearance or behavior than is provided by the standard `TRadioGroup` component. One can, of course, simply put individual `TRadioButtons` into a basic `TGroupBox`, but that requires more effort to keep track of radio buttons that are added, removed, or changed at runtime.

Is it possible to access the individual `TRadioButtons` inside a `TRadioGroup`? As you might have guessed, I wouldn't be writing this article if it weren't! As a `TWinControl` descendant, `TCustomGroupBox` (and, hence, `TRadioGroup`) inherits the ability to "parent" other components (in this context, *parent* refers to the Windows API notion of a "parent window," and not the OOP notion of inheritance). This is manifested in the VCL by the `Controls` property that `TWinControl` and descendants possess.

Thus, the radio buttons maintained by `TRadioGroup` are accessible as `TRadioButtons` through the radio group's `Controls` property. The list of `TRadioButton` components exposed by the `Components` property of `TRadioGroup` is populated in conjunction with the addition of elements to the `Items` property. Therefore, the standard `TRadioGroup` is the parent to *only* enough `TRadioButtons` to account for its contained radio buttons. The radio buttons contained in the `Controls` property are in the order they were added to the `Items` property, and as they appear visually.

Accessing the radio buttons

So, what can you do with this knowledge? Since you now have access to the individual radio buttons, and since they're effectively identical to `TRadioButton` components (at runtime, at least), you can treat them in much the same way you do stand-alone `TRadioButton` components.

For example, `TRadioGroup` provides a single `Hint` property for its entire collection of radio buttons. With the code shown in **Listing A**, however, you can assign separate hints to each radio button individually. This code also assigns an `OnClick` event handler for each radio button (I will explain how to use the `OnClick` event in just a moment). **Listing B** provides a couple of small functions that simplify accessing the separate radio buttons. One function locates a radio button by its index, and the other locates a radio button by the text in the `Caption` property. **Listing C** shows how you could add emphasis to the selected radio button by changing its `Font` property.

Listing C also shows what originally prompted me to pursue this line of investigation. Recently on one of the Usenet programming groups, I ran across a question concerning `TRadioGroup`'s odd behavior regarding tab stops. If you set the standard `TRadioGroup`'s `TabStop` property to `false` and no item is selected, the radio group behaves as expected. That is, tabbing through the form does *not* stop on the radio group. However, if the `ItemIndex` property is set to something other than `-1` ("no item selected") or an item in the group is selected at runtime, that item receives focus in the tab order regardless of the state of the radio group's overall `TabStop` property. By setting a radio button's `TabStop` property to match that of the group when the radio button is selected, the radio group appears to exhibit a more intuitive behavior. I show how to do this in the radio group's `OnClick` event handler, but it is also possible to do this (or anything else for that matter) by assigning event handlers to the events of the individual radio buttons. **Listing 4** shows some of the possibilities. (Refer to **Listing 1** for the code that assigns a method to the `OnCreate` event for each radio button.)

If you are adding items to the radio group at runtime, you'll need to assign the event handler for the new items sometime after you create the radio button.

Other properties of `TRadioButton` that you might care to tinker with include `Color`, `Enabled`, `HelpContext`, `PopupMenu`, and `Visible`. One "gotcha" I ran across regarding changing properties of radio group items is that the items appear to be created with their widths set to the width of the group box. This means that the radio group might look odd if you change the `Alignment` property of a radio button. Similarly, if you change the background color, the change extends the entire width of the group box. You can change the `Width` property in code to alleviate some of this, but you have to be careful not to obscure the caption.

Conclusion

C++Builder and the VCL give us great flexibility and options for creating Windows programs easily. Of course, sometimes a little extra digging can make a good thing even better.

Listing A: *The OnCreate event handler*

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TRadioButton* rb;
  for (int i=0;i<RadioGroup1->ControlCount;i++) {
    rb = dynamic_cast<TRadioButton*>
      (RadioGroup1->Controls[i]);
    if (rb) {
      rb->ShowHint = true;
      rb->Hint = "Something about " + rb->Caption;
      rb->OnClick = RBOnClick;
    }
  }
```

Listing B: *Radio button helper functions*

```
TRadioButton* RadioButtonByCaption(
  TCustomRadioGroup* rg, String Caption)
{
  for (int i=0;i<rg->ComponentCount-1;i++) {
    TRadioButton* rb = dynamic_cast
      <TRadioButton*>(rg->Controls[i]);
    if (rb)
      if (rb->Caption == Caption) {
      return rb;
    }
  }
  return NULL;
}

TRadioButton* RadioButtonByIndex(
  TCustomRadioGroup* rg, int Index)
{
  return dynamic_cast
    <TRadioButton *>(rg->Controls[Index]));
}
```

Listing C: *The radio group's OnClick event handler modifies the radio buttons.*

```
void __fastcall
TForm1::RadioGroup1Click(TObject *Sender)
{
```

```
  // reset the color back to normal
  for (int i=0;i<RadioGroup1->Items->Count;i++) {
    RadioButtonByIndex(RadioGroup1, i)->Font =
     RadioGroup1->Font;
  }
  RadioButtonByIndex(RadioGroup1,
    RadioGroup1->ItemIndex)->TabStop =
    RadioGroup1->TabStop;
  RadioButtonByIndex(RadioGroup1,
    RadioGroup1->ItemIndex)->Font->Color = clRed;
}
```

Listing D: *The individual radio buttons' OnClick event handler.*

```
void __fastcall TForm1::RBOnClick(TObject *Sender)
{
  TRadioButton* rb =
    dynamic_cast<TRadioButton*>(Sender);
  TRadioGroup* rg =
    dynamic_cast<TRadioGroup*>(rb->Parent);
  for (int i=0;i<rb->Parent->ControlCount;i++) {
    TRadioButton* rb2 = dynamic_cast
      <TRadioButton*>(rg->Controls[i]);
    if (rb2)
      rb2->Font = rg->Font;
  }
  rb->Font->Color = clBlue;
  rb->Font->Style = rb->Font->Style << fsItalic;
  rb->TabStop = rb->Parent->TabStop;
}
```

Getting shell item information

by Kent Reisdorph

Last month in the article, “Enumerating the shell namespace” I showed you how to iterate items in the shell namespace. In that article you learned about pidls (a pointer to an item ID list). A pidl is used to describe an item in the shell namespace. In this article you will learn how to use a pidl to obtain information about a particular shell item. Specifically, you will learn how to use the `SHGetFileInfo()` function to obtain the following:

1. Large and small icons for shell items
2. The overlay icon for shared items
3. The overlay icon for shortcuts
4. The display name of shell items
5. The type description of shell items

The functions and constants discussed in this article are defined in SHELLAPI.H and SHLOBJ.H so you will need to include those headers in any application that uses the techniques presented here.

The shell is a moving target

Before I get into the explanations of how to go about obtaining the details of shell items I need to warn you of some issues that you may encounter. Specifically, you need to be aware that the Windows shell will vary widely from one machine to another.

The shell version is largely determined by the version of Microsoft Internet Explorer installed on a particular system. As you might assume, later versions of the shell have features not found in earlier versions of the shell. For the most part this is determined by the version of SHELL32.DLL. Further complicating the problem is the fact that the various flavors of Windows have different shells. For example, shell operations that work on Windows 95 may not work on Windows 98 or Windows NT. In fact, the shell can even vary between different versions of the same operating system. For example, a machine running Windows NT with Service Pack 1 installed may behave differently than an NT machine with a different Service Pack installed. Considering all of this, you can figure out that there are dozens of combinations of operating system and shell version. The best you can do is to write your shell code and know that it may not work properly on all systems. This situation is not quite as dire as it might sound, at least for the topics I will discuss in this article, but let me give you an example.

It is fairly obvious that Windows Explorer allows you to share drives or individual folders. A shared drive or folder is depicted in Explorer with a hand icon. This icon is an overlay icon drawn over the regular icon for the shared folder. The same is true for shortcuts; the shortcut overlay icon is drawn on

top of the regular icon for a particular item. This article will explain how to handle overlay icons. My tests show that the overlays work fine on Win9x systems that I have tested on, and on my NT Workstation machine. However, the overlay icons don't work properly on my NT Server machine with the IE Desktop Update installed. MSDN has an article on this subject and that article says that the problem is "as designed." See the sidebar entitled, "Overlays for all occasions" for information on how to work around this feature of the shell.

The point of all of this is that you need to be aware that what we think of as "the shell" will vary widely on different machines.

Getting item icons

Perhaps one of the more interesting details you can obtain for a shell item is the icon associated with that item. The icons for shell items are obtained from the system image list. Before I explain how to get the icons let me explain how the system image list works.

The system image list and TImageList

The system image list is a list of icons maintained by the operating system. To obtain an item in the system image list, you first associate a `TImageList` with the system image list. I'll explain how to do that later in this section. The interaction between the system image list and the `TImageList` varies between operating systems. Under Win9x, the entire system image list is copied to the `TImageList`. Under NT, however, each process gets a copy of the system image list. Icons are not added to the `TImageList` until they are requested by the application. For example, when you first connect the system image list to a `TImageList`, the `TImageList` will contain from one to five icons (again, depending on the OS and shell version). One image that will always be in the list is the default folder icon. Other images will include the icon for a folder in the open state, the standard Windows icon for a document and executable file, and the overlay icons for shared folders and shortcuts. As you enumerate the shell and request icons for shell items, any new icons that are needed are copied to the `TImageList`. You can witness this on an NT machine by running this article's example program and clicking the Show Images button. You should see new images added to the image list each time you enumerate a new folder.

It is not vital that you understand how the connection between the system image list and a `TImageList` works, but I've included this explanation in case you examine the `TImageList` and notice the surprising lack of icons on NT.

Hooking into the system image list

Hooking into the system image list is trivial. Once you have an instance of `TImageList` you can easily associate it with the system image list. The example program for this article displays shell items in a list

view. First I set up the list view's image lists:

```
ListView1->SmallImages =
  new TImageList(this);
ListView1->LargeImages =
  new TImageList(this);
ListView1->
  SmallImages->ShareImages = true;
ListView1->
  LargeImages->ShareImages = true;
```

I first allocate memory for the `SmallImages` and `LargeImages` properties of the list view. Next I set the `ShareImages` property of each image list to `true`. This step is important because it keeps the system image list intact. When `ShareImages` is `true`, the handle of the underlying image list will not be deleted when the `TImageList` is destroyed. The list view's image lists will be "borrowing" the system image list handle and, as such, should not destroy that handle when the `TImageList` is deleted. Failure to set `ShareImages` to `true` could corrupt the system image list on Win9x platforms.

Next I hook the list view's `SmallImages` and `LargeImages` properties to the system image list. This is done by calling the `SHGetFileInfo()` function:

```
SHFILEINFO fi;
ListView1->SmallImages->Handle =
  SHGetFileInfo("", 0, &fi, sizeof(fi),
  SHGFI_SYSICONINDEX | SHGFI_SMALLICON);
ListView1->LargeImages->Handle =
  SHGetFileInfo("", 0, &fi, sizeof(fi),
  SHGFI_SYSICONINDEX | SHGFI_LARGEICON);
```

`SHGetFileInfo()` takes five parameters. The first parameter is the path and filename for the file or folder for which you wish to obtain shell information. In this case I pass an empty string because I am only getting the image list handle. The second parameter is used to specify the file attributes you are requesting. I don't need the attributes so I pass `0` for this parameter. The third and fourth parameters are a pointer to a `SHFILEINFO` structure and the size of the structure. This structure is filled out with the requested information. In this case, I'm not interested in the information returned in the structure but I still have to pass the pointer and size.

Finally, we get to the final parameter. This parameter is used to specify a set of flags that determine the type of information requested. In the first call to `SHGetFileInfo()` I pass the `SHGFI_SYSICONINDEX` and `SHGFI_SMALLICON` flags. When `SHGetFileInfo()` is called with these flags, it will return a handle to the system image list that contains the shell's small icons. I assign the return value to the `Handle` property of the list view's `SmallIcons` property. This associates the

system image list with the list view's small icon `TImageList`. The next call to `SHGetFileInfo()` passes the `SHGFI_LARGEICON` flag to obtain the large icons.

At this point the list view can use the icons from the system image list.

Getting the icon for an item

Finally we get to the point where we are ready to retrieve the icon for a particular item in the shell. This is done by calling `SHGetFileInfo()`. There are two ways you can go about getting the icon. One way is to pass the path and filename for the item to `SHGetFileInfo()`:

```
DWORD Flags =
  SHGFI_SYSICONINDEX | SHGFI_ATTRIBUTES;
SHFILEINFO fi;
DWORD Result = SHGetFileInfo(
  "c:\\test.txt", 0, &fi,
  sizeof(fi), Flags);
if (Result != 0) // success
```

The second way is to pass a pidl in the first parameter. Since the first parameter takes a `char*` you will need to cast the pidl:

```
DWORD Flags = SHGFI_PIDL |
  SHGFI_SYSICONINDEX | SHGFI_ATTRIBUTES;
SHFILEINFO fi;
DWORD Result = SHGetFileInfo(
  (char*)pidl, 0, &fi,
  sizeof(fi), Flags);
if (Result != 0) // success
```

The first method certainly seems easier, but remember that not all shell items are part of the file system. You must use a pidl, for example, to obtain the icons for items in the Control Panel, in the Network Neighborhood, and in other special shell folders. Also, if you are enumerating the shell you already have a pidl so the hard work is already done by the time you get around to calling `SHGetFileInfo()`. It is important to remember that the pidl passed to `SHGetFileInfo()` must be a fully qualified pidl (relative to the Desktop folder).

Note the flags I set for each of the methods presented. The `SHGFI_ATTRIBUTES` flag is not used in this code but it is required to get the overlay icon as explained in the next section.

When `SHGetFileInfo()` returns, the `iIcon` member of the `SHFILEINFO` structure will contain the index of the item's icon in the system image list. Using the icon in a list view is simple:

```
TListItem* item =
  ListView1->Items->Add();
item->ImageIndex = fi.iIcon;
```

I simply assign the icon index contained in the `iIcon` member to the `ImageIndex` property of the list view.You could, of course, access the icon in the image list directly. If, for example, you were creating an owner-drawn combo box that displays shell items, you would have a dedicated `TImageList` to hold the shell icons. You would then use the methods of `TImageList` to extract and draw the icon.

Setting the overlay icons

Setting the overlay icon for a shell item is relatively easy. The index for the overlay icon that indicates a shared folder is `0`, and the index for the shortcut overlay icon is `1`. Given that, you only need to check to see whether an item is shared or a shortcut and assign the appropriate image accordingly. The `dwAttributes` member of the `SHFILEINFO` structure contains the shell item's attributes. After you call `SHGetFileInfo()` you can set the overlay icon with this code:

```
if ((fi.dwAttributes & SFGAO_SHARE)
     == SFGAO_SHARE)
  item->OverlayIndex = 0;
else if ((fi.dwAttributes & SFGAO_LINK)
     == SFGAO_LINK)
  item->OverlayIndex = 1;
else
  item->OverlayIndex = -1;
```

This code builds on the code you saw in the previous section. I simply set the `TListItem's OverlayIndex` property to `0` if the item is shared, to `1` if the item is a shortcut, or to `–1` for all other cases. The list view will do the rest by painting the overlay icon over the regular icon for the shell item.

There is one aspect of `TListView` and overlay icons that you need to be aware of. For some reason, the `OverlayIndex` property is ignored when the list view is a virtual list view. I suspect this is a bug in the `TListView` code. If you are using a virtual list view and want overlay icons, you will have to catch the `WM_PAINT` message and drawing each item yourself.

Getting the display name and type

Getting the display name and type description of a shell item can be combined into a single operation. Here is the code:

```
DWORD Flags = SHGFI_PIDL |
  SHGFI_DISPLAYNAME | SHGFI_TYPENAME;
SHFILEINFO fi;
DWORD Result = SHGetFileInfo(
  (char*)pidl, 0, &fi, sizeof(fi), Flags);
String typeDescription;
if (Result != 0) {
  item->Caption = fi.szDisplayName;
  typeDescription = fi.szTypeName
}
```

When the `SHGFI_DISPLAYNAME` flag is set, the `szDisplayName` member of `SHFILEINFO` will contain the display name of the item. The value contained in `szDisplayName` can be used for the caption of the list view item. Likewise, when the `SHGFI_TYPENAME` flag is set, the `szTypeName` member will contain the type description of the item. You could save this description and use it to display the file's type if the list view is in report mode.

Earlier I showed you how to get the icon for a shell item. There I set flags to obtain just the icon. In reality, I would combine the code presented in that section with the code presented here and make just one call to `SHGetFileInfo()`.

There is one aspect of `SHGetFileInfo()` that you should be aware of. Specifically, `SHGetFileInfo()` takes a relatively long time to execute. If you are enumerating a folder containing thousands of items, the call to `SHGetFileInfo()` will be the most time consuming part of the enumeration. I have worked around this in my own code by using a virtual list view and by calling `SHGetFileInfo()` only when an item needs to be displayed (virtual list views were discussed last month in the article, "Faster Updates with Virtual List Views").

Conclusion

In this article I showed you how to obtain information about a shell item using the `SHGetFileInfo()` function. The example program for this article allows you to enumerate a folder in the file system. The results are shown in a list view, complete with the proper icons. The example program also has a second form that can be used to view the system image list. The source for the program's main form is shown in **Listing A**. I don't show the main form's header or the code for secondary form to save space. I should point out that the example uses an undocumented shell function called `ILCombine()`. As its name implies, this function combines two pidls and returns a new pidl containing the result. A pointer to this function is obtained using `GetProcAddress()` because the C++Builder linker does not support importing functions by ordinal. For more information on undocumented shell functions I suggest you visit the Web site at **www.geocities.com/siliconvalley/4942**.

Listing A: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include <shellapi.h>
#include <shlobj.h>
#include "MainU.h"
#include "ViewImagesU.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;
HINSTANCE Shell32Inst;
typedef LPITEMIDLIST (__stdcall *pILCombine)
  (LPITEMIDLIST, LPITEMIDLIST);
pILCombine ILCombine;
LPSHELLFOLDER DesktopFolder;

int __stdcall ListViewCompareFunc(LPARAM lParam1,
  LPARAM lParam2, LPARAM lParam)
{
  TListItem* Item1 = (TListItem*)lParam1;
  TListItem* Item2 = (TListItem*)lParam2;
  // Use CompareIDs to sort the list view.
  return (short)DesktopFolder->CompareIDs(0,
    (LPITEMIDLIST)Item1->Data,
    (LPITEMIDLIST)Item2->Data);
}

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  // Set up the image lists.
  ListView1->SmallImages = new TImageList(this);
  ListView1->LargeImages = new TImageList(this);
  ListView1->SmallImages->ShareImages = true;
  ListView1->LargeImages->ShareImages = true;
  SHFILEINFO fi;
  ListView1->SmallImages->Handle =
```

```
    SHGetFileInfo("", 0, &fi, sizeof(fi),
      SHGFI_SYSICONINDEX | SHGFI_SMALLICON);
  ListView1->LargeImages->Handle =
    SHGetFileInfo("", 0, &fi, sizeof(fi),
      SHGFI_SYSICONINDEX | SHGFI_LARGEICON);
  // Load SHELL32.DLL and get a pointer to the
  // undocumented ILCombine function.
  Shell32Inst = LoadLibrary("shell32.dll");
  if (Shell32Inst)
    ILCombine = (pILCombine)GetProcAddress(
      Shell32Inst, (const char*)25);
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  FreeLibrary(Shell32Inst);
}

void __fastcall TForm1::EnumBtnClick(TObject *Sender)
{
  ListView1->Cursor = crHourGlass;
  ListView1->Perform(WM_SETREDRAW, 0, 0);
  FreePidls();
  ListView1->Items->Clear();

  LPSHELLFOLDER ParentFolder;
  LPITEMIDLIST ParentPidl;
  LPITEMIDLIST ChildPidl;
  LPITEMIDLIST fqPidl;
  DWORD Eaten;
  wchar_t Path[MAX_PATH];
  // Get an IShellFolder for the path in the edit.
  StringToWideChar(
    StartPathEdit->Text, Path, MAX_PATH);
  DWORD Result = DesktopFolder->ParseDisplayName(
    Handle, 0, Path, &Eaten, &ParentPidl, 0);
  Result = DesktopFolder->BindToObject(ParentPidl,
    0, IID_IShellFolder, (void**)&ParentFolder);
  if (Result != NOERROR)
    return;
  LPENUMIDLIST Enum;
  // Set enumeration flags to get all items.
  Result = ParentFolder->EnumObjects(Handle,
    SHCONTF_FOLDERS | SHCONTF_NONFOLDERS |
```

```
    SHCONTF_INCLUDEHIDDEN, &Enum);
  if (Result != NOERROR)
    return;
  // Get the pidl for the first item in the folder.
  LPMALLOC Malloc;
  SHGetMalloc(&Malloc);
  Result = Enum->Next(1, &ChildPidl, 0);
  while (Result != S_FALSE) {
    // If result is something other than
    // NOERROR then some error occured.
    if (Result != NOERROR)
      break;
    TListItem* Item = ListView1->Items->Add();
    fqPidl = ILCombine(ParentPidl, ChildPidl);
    SetItemAttributes(Item, fqPidl);
    // Free the memory for the pidl returned
    // by the Enum->Next() function.
    Malloc->Free(ChildPidl);
    // Get a pidl for the next item in the folder.
    Result = Enum->Next(1, &ChildPidl, 0);
  }
  // Sort the list view and enable drawing.
  ListView1->CustomSort(ListViewCompareFunc, 0);
  ListView1->Perform(WM_SETREDRAW, 1, 0);
  // Clean up.
  Malloc->Free(ParentPidl);
  Enum->Release();
  DesktopFolder->Release();
  ParentFolder->Release();
  Malloc->Release();
  ListView1->Cursor = crDefault;
}

void TForm1::SetItemAttributes(TListItem* item,
  LPITEMIDLIST pidl)
{
  DWORD Flags = SHGFI_PIDL |
    SHGFI_SYSICONINDEX | SHGFI_DISPLAYNAME |
    SHGFI_ATTRIBUTES;
  SHFILEINFO fi;
  DWORD Result = SHGetFileInfo(
    (char*)pidl, 0, &fi, sizeof(fi), Flags);
  // Error getting the info.
```

```
  if (Result == 0)
    return;
  // Set the image index with the value
  // returned by Windows.
  item->ImageIndex = fi.iIcon;
  // Save the pidl in the Data property of the
  // item. We'll need it to sort the list view.
  item->Data = pidl;
  // Set the Cut property based on the
  // ghosted attribute.
  item->Cut = (fi.dwAttributes & SFGAO_GHOSTED)
       == SFGAO_GHOSTED;
  // Set the caption of the item.
  item->Caption = fi.szDisplayName;
  // Set the overlay icon index.
  if ((fi.dwAttributes & SFGAO_SHARE)
       == SFGAO_SHARE)
    item->OverlayIndex = 0;
  else if ((fi.dwAttributes & SFGAO_LINK)
       == SFGAO_LINK)
    item->OverlayIndex = 1;
  else
    item->OverlayIndex = -1;
}

void TForm1::FreePidls()
{
  LPMALLOC Malloc;
  SHGetMalloc(&Malloc);
  for (int i=0;i<ListView1->Items->Count;i++)
    Malloc->Free(ListView1->Items->Item[i]->Data);
  Malloc->Release();
}

void __fastcall TForm1::ShowIconsBtnClick(TObject *Sender)
{
  TViewImagesForm* form = new TViewImagesForm(this);
  form->SmallImages = ListView1->SmallImages;
  form->LargeImages = ListView1->LargeImages;
  form->ShowModal();
  delete form;
}
```

Image shaped forms

by Andy Walker

No doubt you have seen applications that contain dialogs or main windows that have unusual shapes and even have transparent areas that don't appear to be part of the form. This article will take you step-by-step through the process of creating a component that can be placed on a form and can be used to shape and define transparent areas of any visual component. That includes forms, buttons, list boxes, and, in fact, anything that has a window handle.

The component created in this article assumes that you are using C++Builder 4. The techniques, however, are generic and can be coded using any development tool. (**Note:** The code presented in this article uses the `ScanLine` property of `TBitmap` which first appeared in C++Builder 3).

Creating the component

To begin, choose Component|New Component from the C++Builder main menu. Select `TComponent` for the ancestor type and enter `TTransparentWindow` in the Class Name field. Leave the other settings as they are and click the OK button. Save the file as TRANSPARENTWINDOW.CPP.

Next you will need to add some properties and methods. In the header for your new component, add the following definitions to the `private` section:

```
AnsiString asTransparentImage;
HWND lWindowHandle;
HRGN rgnOpaque;
Graphics::TBitmap *bmTransparentImage;
void __fastcall SetTransparentImage
  (System::AnsiString value);
bool __fastcall SetTransparentRegion
  (Graphics::TBitmap *bmBitmap);
```

The first variable is the data member for a property called `TransparentImage`. I'll explain this property later in this section. The next three variables are used internally by the component. The `lWindowHandle` variable will be used to store the window handle for the selected component. The `rgnOpaque` variable will hold the region handle for the opaque region of the window. The `bmTansparentRegion` variable will hold the bitmap that defines the region. The `SetTransparentImage()` function is the write method for the `TransparentImage` property. This function will create the transparency image and load it into memory. The `SetTransparentBitmap()` function creates the opaque region of the form and removes the

transparent parts.

The `public` section of the component already defines a constructor, but you need to add definitions for a destructor and two public methods. Here are those declarations:

```
__fastcall ~TTransparentWindow();
bool __fastcall Activate();
void __fastcall SetWindowHandle
  (HWND Handle) {lWindowHandle=Handle;}
```

The `Activate()` method is used to activate the component, and the `SetWindowHandle()` method is used to tell the component which windows handle to work with. Note that the `SetWindowHandle()` method is an inline function.

To finish off the header you need to add a property to the `__published` section:

```
__property AnsiString TransparentImage =
  { read = asTransparentImage,
    write = SetTransparentImage };
```

This property is used to specify the name of the required transparency image.

An overview of Windows regions

Before I get to the code, I need to take a moment to explain how Windows regions work. Any object in Windows that can be filled, painted, inverted, framed, or hit tested (testing the location of the cursor) is a region. The region can be any shape, including rectangles, ellipses, and polygons. Regions can be combined or compared with any other region. By combining multiple sources, it is possible to build up a region that is irregular in shape and contains gaps that do not form part of the original area. Regions can be combined using the API call `CombineRgn()`. `CombineRgn()` uses the following Boolean operators:

```
RGN_AND
RGN_OR
RGN_XOR
RGN_COPY
RGN_DIFF
```

Once a region has been created, it can be selected into a device context. At that point the region can be filled, inverted, painted, framed, or it can be used to replace an existing region. In this article, `CombineRgn()` is used to replace the window region for a given window handle. This is done by

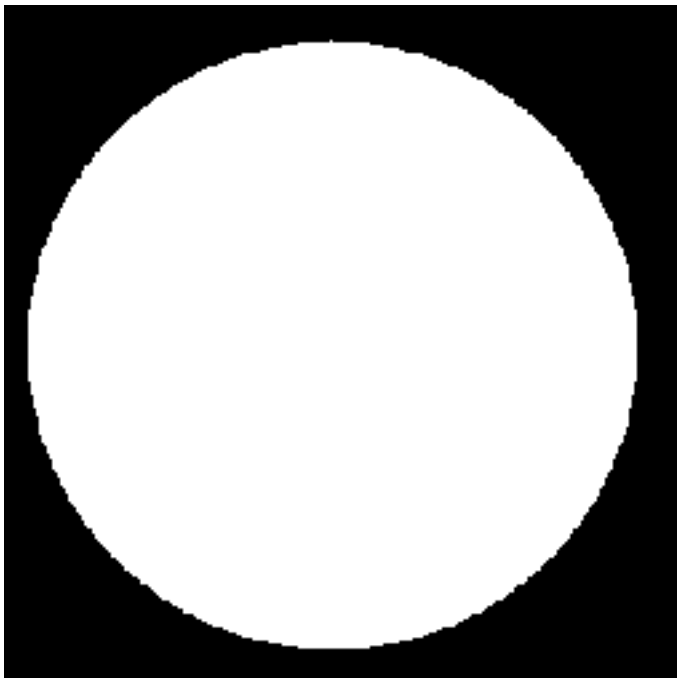
calling the Windows API `SetWindowRgn()` function.

I can't explain every possible use of regions in this article, but you can find out more about regions by starting with the "Regions" topic in the Win32 SDK Reference help file.

Setting the transparent region

Now that the header is finished, you can move on to the actual coding of the component. The most important method in the entire project is `SetTransparentRegion()`. This method creates an irregular shaped region from a given transparency bitmap. It creates a region from the bitmap by processing the transparency image line by line and creating the region from the non-black parts of the bitmap. If, for example, the bitmap contains a black background with a non-black circle, the routine will create a circular window region (see **Figure A**). It achieves this by creating regions for the non-black area on each line of the bitmap and then combining them all together to create one irregular region. Depending on the complexity of the image, the resulting region might consist of single opaque pixels. Of course, all of this takes processing time so the simpler the image the faster the processing will go.

Figure A



This bitmap will create a circular window.

The `SetTransparentRegion()` routine can be split into five distinct functional steps. I will discuss each of these steps in the following sections.

Step 1: Create a region

The first step is to create a region that is the same size as the original bitmap:

```
HRGN rgnOpaque;
rgnOpaque = CreateRectRgn(0, 0,
  bmTransparentBitmap->Width,
  bmTransparentBitmap->Height);
```

This code creates an opaque region that starts at co-ordinates (0,0) and extends to the width and height of the bitmap.

Step 2: Process each line of the bitmap

Once you have a region to work with, you can begin the process of scanning each line of the bitmap and manipulating the region to create transparent parts. Here's the code for this step:

```
for (Y=0;
  Y<bmTransparentBitmap->Height;
  Y++)
{
  Line = (Byte*)
    bmTransparentBitmap->ScanLine[Y];
  // step 3 and 4 code here
}
```

You now have a variable called `Line` that will point to the next line of the bitmap each time through the loop.

Step 3: Process non-transparent region

Now you can use the `Line` variable to test the color of each pixel on the current scan line. Remember that black is the transparent color so I loop through the pixels until I either reach a black pixel (represented by a `0`) or the final pixel on that line:

```
while (X < bmTransparentBitmap->Width
  && Line[X] != 0)
{
  //loop until a black pixel is found
  X++;
}
```

Step 4: Process transparent region

At this point you have either reached the end of the scan line, or reached a black pixel. The next step is to make a note of the position of the first black pixel on the line and continue processing the rest of the line:

```
XTransparent=X;
while (X<bmTransparentBitmap->Width
    && Line[X] == 0)
{
  //loop until a non-black pixel is found
  X++;
}
```

When this loop finishes, you will have the transparent region start position in the `XTransparent` variable, and the end position in the variable `X`. The end position could be the end of the line, or if another non-black pixel was encountered, could be partway through the line.

Step 5: Remove the transparent part

The final step is to check whether a transparent section exists and, if so, remove it from the original opaque region. (The original region was created in Step 1.) This is achieved by creating a temporary region that is one pixel high and `XTransparent` to `X` pixels wide. This region could potentially be zero pixels, one pixel, a whole line of pixels the width of the bitmap, or any number in between. Here is the code that creates the region:

```
rgnTransparent =
  CreateRectRgn(XTransparent, Y, X, Y+1);
```

Once you have a new region of the correct size, you can remove it from the original. This is done by combining the original region and the newly created region using the `CombineRgn()` function. `CombineRgn()` takes four parameters. The first parameter is the handle of the region that will contain the result. The second and third parameters are the handles to the two regions to be combined. The last parameter is used to specify the region combine mode. In this case you will perform an exclusive `OR` region combine by passing the `RGN_XOR` constant for the final parameter:

```
CombineRgn(rgnOpaque,
  rgnOpaque, rgnTransparent, RGN_XOR);
```

The result is the original opaque region with the transparent region cut out.

Now imagine that you have a bitmap whose first line consists of the pixels shown in **Figure B**. The code

detailed in Steps 3, 4, and 5 would process the first set of white pixels (as non-transparent), then process the black pixels (as transparent), create a temporary region for the transparent area, and then `XOR` that region with the original.

Figure B

An example scan line from a transparent image bitmap.

In order to get as far as the final set of white pixels, it is necessary to enclose the entire process within another loop that ensures each scan line is processed completely:

```
for(X=0;X<bmTransparentBitmap->Width;X++)
{
  // step 3 code
  // step 4 code
  // step 5 code
}
```

The source file for the example component is contained in **Listing A**. Examine the `SetTransparentRegion()` function in this listing to see the entire set of steps in context.

Conclusion

After entering the code in **Listing A** (or downloading it if you prefer) you can compile and install the component. After the component is installed, place it on a form and set the `TransparentImage` property to a valid bitmap. In the form's `OnCreate` event handler, call the `SetWindowHandle()` method, passing the `Handle` property of the form. Finally, call the `Activate()` method to activate the component. The result is a form that is shaped like the non-black pixels of the transparency bitmap.

The `CreateRectRgn()` and `CombineRgn()` functions make it possible to create windows that take on irregular and unusual shapes. The techniques covered here can be applied to any component and could be used to create shaped buttons, list boxes, toolbars, and so on.

All of the code detailed in this article can be downloaded from the Bridges Publishing Web site, or from my site at **http://www.iola.co.uk**. Just follow the link for FreeStuff and download the zip file.

Listing A: *TTransparentWindow.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "TransparentImage.h"
#pragma package(smart_init)

static inline void
ValidCtrCheck(TTransparentWindow *)
{
  new TTransparentWindow(NULL);
}

__fastcall TTransparentWindow::TTransparentWindow(
  TComponent* Owner) : TComponent(Owner)
{
  //<AW>Make Sure Pointers Are NULL
  bmTransparentImage = NULL;
}

bool __fastcall
TTransparentWindow::Activate()
{
  bool bSuccess;

  /*<AW>If the image has been set and the
  Windows handle has been specified then
  we can continue and return success*/
  if (bmTransparentImage &&
  lWindowHandle!=0)
  {
    //<AW>Create an opaque region from a bitmap

    bSuccess=SetTransparentRegion
      (bmTransparentImage);
    if (bSuccess)
    //<AW>Set region for the supplied Windows handle
      SetWindowRgn (lWindowHandle, rgnOpaque, true);
  }
  else
  {
  //<AW>If something is missing then report
  // what and return failure
  if (bmTransparentImage==NULL)
    Application->MessageBox ("No Bitmap Selected",
```

```
      "Activate Cancelled", MB_OK);
  else if (lWindowHandle == 0)
    Application->MessageBox (
      "No Window Handle Selected",
      "Activate Cancelled", MB_OK);
  bSuccess=false;
  }
  return bSuccess;
}

void __fastcall
TTransparentWindow::SetTransparentImage(
  System::AnsiString value)
{
  //<AW>Set The Window Image
  try
  {
    //<AW>If The Bitmap Exists Then Delete It
    if (bmTransparentImage)
    {
      delete bmTransparentImage;
      bmTransparentImage = NULL;
    }
    //<AW>Load The Image
    bmTransparentImage = new Graphics::TBitmap();
    asTransparentImage = value;
    bmTransparentImage->
      LoadFromFile(asTransparentImage);
    bmTransparentImage->PixelFormat = pf8bit;
  }
  catch (Exception &e)
  {
    //<AW>Handle Any Errors And Delete The Pointer
    Application->MessageBox (e.Message.c_str(),
      "Couldn't Load Transparent Image", MB_OK);
    if (bmTransparentImage)
    {
      delete bmTransparentImage;
      bmTransparentImage = NULL;
      asTransparentImage = "";
    }
  }
}
```

```
__fastcall TTransparentWindow::~TTransparentWindow()
{
  //<AW>Clean Up And Delete The Bitmap
  if (bmTransparentImage)
    delete bmTransparentImage;
}

bool __fastcall
TTransparentWindow::SetTransparentRegion(
  Graphics::TBitmap *bmTransparentBitmap)
{
  //<AW>Set The Active Region Of The Window
  long X,Y,XTransparent;
  Byte *Line;
  HRGN rgnTransparent;
  bool bSuccess;
  try
  {
    bSuccess = true;
    //<AW>Loop Through Each ScanLine and Each Pixel
    rgnOpaque = CreateRectRgn (0, 0,
     bmTransparentBitmap->Width,
     bmTransparentBitmap->Height);
    for (Y=0;Y<bmTransparentBitmap->Height;Y++)
    {
      Line=(Byte*)bmTransparentBitmap->ScanLine[Y];
      for(X=0;X<bmTransparentBitmap->Width;X++)
      {
        while (X<bmTransparentBitmap->Width
          && Line[X]!=0)
        {
          //<AW>Process The Non-Transparent Region
          X++;
         }
        //<AW>Transparent Start
        XTransparent=X;

        while (X<bmTransparentBitmap->Width
          && Line[X]==0)
        {
          //<AW>Process The Transparent Region
          X++;
        }
```

```
        //<AW>If There Was A Transparent Region
        if (XTransparent<X)
        {
          //<AW>Create The Region
          //(Remove The Transparent Bit)
          rgnTransparent = CreateRectRgn
            (XTransparent, Y, X, Y+1);
          //<AW>If It's A Null Region, Return False
          if (CombineRgn (rgnOpaque,
            rgnOpaque, rgnTransparent,
            RGN_XOR) == NULLREGION)
          bSuccess=false;
         DeleteObject(rgnTransparent);
        }
      }
    }
  }
  catch (Exception &e)
  {
    //<AW>Catch The Errors
    Application->MessageBox(e.Message.c_str(),
      "Error Creating Transparency", MB_OK);
  }
  return bSuccess;
}

namespace Transparentimage
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] =
      {__classid(TTransparentWindow)};
    RegisterComponents("Sample", classes, 0);
  }
}
```

Overlays for all occasions

by Kent Reisdorph

In the article, “Getting shell item information,” I explained how to get the icon for shell items and, if applicable, the overlay icon. As it turns out, the overlay icons are not added to the application’s image list on Windows NT with the Internet Explorer 4 (or later) Desktop Update installed. The overlay icons *are* added to the image list properly when the Desktop Update is not installed.

This article will explain how to work around this problem (which Microsoft has categorized, “as designed”).

The key to adding the overlay icons to the image list is the `IShellIconOverlay` interface. The following steps are required to add the overlay icons:

1. Create a temporary shortcut file
2. Get an `IShellFolder` interface to the folder where the temporary file was created
3. Get an `IShellIconOverlay` interface for the folder
4. Get a pidl for the temporary file
5. Call the `GetOverlayIndex()` method of `IShellIconOverlay` to add the overlays to the image list
6. Get the index of NTDETECT.COM to use for DOS executables

Following is the code that performs these steps. For the most part the code is explained by the comments in the code.

```
LPSHELLFOLDER DesktopFolder;
SHGetDesktopFolder(&DesktopFolder);
if (Win32Platform ==
      VER_PLATFORM_WIN32_NT) {
  LPMALLOC Malloc;
  SHGetMalloc(&Malloc);
  LPSHELLFOLDER Folder;
  LPITEMIDLIST Pidl;
  LPITEMIDLIST ParentPidl;
  IShellIconOverlay *IconOverlay;
  DWORD Eaten;
  // Get the Windows temp directory.
  char tempDir[MAX_PATH - 1];
  GetTempPath(sizeof(tempDir), tempDir);
```

```
// Build a path and filename to the
// temporary file. Note that the file
// has an extension of .lnk.
char tempFile[MAX_PATH - 1];
strcpy(tempFile, tempDir);
strcat(tempFile, "temp.lnk");
// Create the file.
CreateFile(tempFile, GENERIC_WRITE,
  0, 0, CREATE_NEW,
  FILE_ATTRIBUTE_NORMAL, 0);
// Get a pidl for the temp folder.
wchar_t Path[MAX_PATH];
StringToWideChar(
  tempDir, Path, MAX_PATH);
DesktopFolder->ParseDisplayName(
  Handle, 0, Path,
  &Eaten, &ParentPidl, 0);
// Get an IShellFolder for
// the temp folder.
DesktopFolder->BindToObject(
  ParentPidl, 0, IID_IShellFolder,
  (void**)&Folder);
// Get an IShellIconOverlay interface
// for the folder.
DWORD Result = Folder->QueryInterface(
  IID_IShellIconOverlay,
  (void**)&IconOverlay);
// If we failed then this version of
// the shell does not have this
// interface. That's OK, though,
// because in that case we already
// have the overlay icons.
if (Result == 0) {
  StringToWideChar(
    "temp.lnk", Path, MAX_PATH);
  // Get a pidl for the temp file.
  Folder->ParseDisplayName(
    Handle, 0, Path, &Eaten, &Pidl, 0);
  int Index;
  // Get the overlay icons.
  IconOverlay->GetOverlayIndex(
    Pidl, &Index);
  IconOverlay->Release();
```

```
  }
  // Delete the temporary file.
  DeleteFile(tempFile);
  // Clean up.
  Malloc->Free(Pidl);
  Malloc->Free(ParentPidl);
  Folder->Release();
  Malloc->Release();
  // Get the icon index of ntdetect.com. This
  // index will be used for dos executables
  SHFILEINFO fi;
  SHGetFileInfo("c:\\ntdetect.com", 0,
    &fi, sizeof(fi), SHGFI_SYSICONINDEX);
  DosExeIndex = fi.iIcon;
}
```

I call `GetOverlayIndex()` for a shortcut because a shortcut, naturally, has an associated overlay icon. It is better to create a temporary shortcut for this purpose then to try to use a shortcut that may or may not be present on a given system. Calling `GetOverlayIndex()` adds all overlays to the image list, not just the shortcut overlay.

The call to `QueryInterface()` obtains the `IShellIconOverlay` interface from the `IShellFolder`. This call will fail if the `IShellIconOverlay` interface is not available on a particular system. If the interface cannot be obtained, the overlay icons are already in the image list so `IShellIconOverlay` is not required.

Note the last three lines of code in the preceding listing. This code gets the icon index for a file called NTDETECT.COM and saves the index in a class member variable. This is necessary because `SHGetFileInfo()` will return `2` for the icon index of DOS executables, or for executables that don't define an icon. For whatever reason, this icon index is incorrect. The result is that the icon shown for this type of file will be wrong (usually the shortcut overlay icon is shown). I use NTDETECT.COM because it has the same icon as DOS executables. It is a hidden system file and should be available on all NT systems. I save the icon index for this file and then later apply it if I determine that the icon index is `2`. The following code snippet was taken from **Listing A** of the preceding article and modified to account for the DOS icon index problem.

```
if (DosExeIndex != -1 && fi.iIcon == 2)
  item->ImageIndex = DosExeIndex;
else
  item->ImageIndex = fi.iIcon;
```

If `DosExeIndex` is `–1` it means that the shell has the correct icons (no Desktop Update installed). If

`DosExeIndex` is not `–1`, and if the `iIcon` member of `SHFILEINFO` is `2`, then I assign the value of `DosExeIndex` to the `ImageIndex` of the list view rather than the icon index returned by `SHGetFileInfo()`. I'll admit that this technique feels like a bit of a hack, but I have yet to find any information on how to properly work around this problem.

This is just one example of how certain combinations of operating system and shell version can cause you headaches. Your application may be working perfectly until a user installs Internet Explorer with the Desktop Update. Understanding the quirks of the Windows shell can help you write programs that work on all system configurations.

The complete example program for the article, "Getting shell item information," includes the code presented in this article. The code was removed from that article's listing because it is shown here. Download the example from our Web site to see the code in context.

To auto-create or not to auto-create

by Kent Reisdorph

Obviously a C++Builder GUI program consists of one or more forms. C++Builder gives you two options for runtime form creation; you can allow the VCL to auto-create your forms or you can take care of form creation yourself. By default, all new forms are auto-created.

Many C++Builder programmers have ignored the option of creating forms at runtime, blindly allowing the VCL to auto-create all forms. Why? Because auto-creation of forms is the IDE default and the alternative of runtime form creation never enters their minds. I should point out that when I talk about form creation options I am talking about all forms *except* the main form. The main form should always be auto-created.

So, of the two form creation options, which is better? To a degree, the answer to that question depends on the specific form being created. Auto-creation has one primary advantage—it's easy. Very little thought is required to write an application in which all forms are auto-created. You simply build the form and access the form when needed:

```
AboutBox->ShowModal();
```

Auto-creation has two primary disadvantages. First, the application takes longer to load if there are several forms that need to be created. When a VCL application starts, it goes through the process of creating every form in the auto-create list. This takes time, especially if there are a lot of forms to create. The second disadvantage of auto-creation is that your application will use more memory than is required. If, for example, your application contains 20 forms, memory is allocated for every form, even when that form is not being displayed. If any of those forms allocate large blocks of memory in the constructor or `OnCreate` event handler, even more memory is being wasted.

Runtime creation of forms also has its advantages and disadvantages. For the most part, the advantages of runtime form creation are the inverse of the auto-creation disadvantages. That is, faster application loading and less memory used. One of the disadvantages of runtime creation is that a little more work is required by the programmer:

```
TAboutBox* about = new TAboutBox(this);
about->ShowModal();
delete about;
```

This is a simple example. The situation gets more complicated if you want your forms to maintain state information between invocations. Another disadvantage of runtime creation is there may be a slight delay while the form is being created. For simple forms this delay won't be noticed, but for complex forms

there may be a noticeable delay as the form is being created.

All things considered, I rarely use form auto-creation. To remove a form from the auto-create list, go to the Forms page of the Project Options dialog and move the form from the Auto-create list box to the Available forms list box. Don’t forget that C++Builder creates a global variable for all forms and that the variable will be `NULL` when the application starts if you are not using auto-creation. You can simply delete the global variable if it you are creating your forms at runtime.

Changing common dialog button captions

by Kent Reisdorph

Last month we showed you how to extend the common dialogs using dialog templates. Sometimes, however you only want to change the caption of an existing button on a common dialog. If, for example, you want to use the open dialog to allow your users to delete a file then you would probably want the Open button to show “Delete” rather than “Open.” This is accomplished rather easily in the dialog’s `OnShow` event handler. In your event handler you can simply do this:

```
HWND hWndDlg =
  GetParent(OpenDialog1->Handle);
HWND hWndBtn = GetDlgItem(hWndDlg, 1);
SetWindowText(hWndBtn, "&Delete");
```

I first obtain the dialog’s window handle by calling `GetParent()` passing the `Handle` property as a parameter. It is important to understand that the dialog’s `Handle` property is the window handle of a child dialog created by Windows, not the handle to the file open dialog itself. Calling `GetParent()` gives us a window handle to the file open dialog itself. Next I call `GetDlgItem()` to get the window handle of a control on the dialog. In this case I pass the handle to the dialog and a resource ID of 1, the resource ID of the dialog’s Open button. I used the resource editor from Borland C++ 5.02 to determine the control ID of the open button. I simply loaded COMDLG32.DLL in the resource editor and spied on the file open dialog resource.

Changing the caption of a button on a common dialog is easy provided you know how the common dialogs are structured.

Enumerating the shell namespace

by Kent Reisdorph

The Windows API provides several functions for accessing the Windows shell. The functions include functions for displaying the Windows shell browser, for adding to the recent documents folder, for accessing the recycle bin, for adding icons to the system tray, for spawning applications, and so on. We have covered some of these functions in previous articles and most of them are fairly easy to implement.

One aspect of the Windows shell that is not quite as straightforward is the task of enumerating the shell namespace. You may want to enumerate the shell namespace in order to provide a custom view into the shell for your applications. For example, you may wish to provide a tree view that shows specific folders and files beyond what the shell browser provides. Or maybe you need a tree view that can be placed on a form rather than using the browser dialog as provided by Windows.

This article will explain how to enumerate the Windows shell. First I will explain the basics of enumerating the shell using the `IShellFolder` and `IEnumIDList` COM interfaces. After that I will show how to obtain the display name and attributes for items in the shell. The functions and constants discussed in this article are defined in SHELLAPI.H and SHLOBJ.H so you will need to include these headers in any application that enumerates the shell.

Folders and Pidls

Manipulating the Windows shell requires an understanding of folders and item identifier lists. I'll discuss these two elements in the following sections.

Folders and the IShellFolder interface

For the most part, shell folders are not difficult to comprehend. If you run Windows Explorer you will see folders representing My Computer, the Recycle Bin, and, if you are on a network, the Network Neighborhood. Opening the My Computer folder results in Explorer displaying subfolders that contain your file system, the Control Panel, the Printers folder, and so on.

Folders are represented by an instance of the `IShellFolder` interface. This interface is the key to manipulating shell folders. Once you have a pointer to an `IShellFolder` interface for a particular folder you can get the details for an item in the folder (display name, icon, file type, etc.), enumerate the folder, rename an item, or get the shell context menu for an item. **Table A** lists the most often used `IShellFolder` functions and gives a description of each. Later you will see examples of how to use many of these functions.

Table A: Commonly-used IShellFolder functions.

Function	Description
BindToObject()	Retrieves an IShellFolder interface for a subfolder.
EnumObjects()	Retrieves a pointer to an IEnumIDList interface in order to enumerate the folder.
GetAttributes()	Retrieves the attributes of a particular item in the folder.
GetDisplayName()	Retrieves the display name of an item in the folder. The string returned can be the normal display name (as shown in Explorer) or a display name that can be used for parsing (the fu the item).
GetUIObjectOf()	Retrieves a pointer to a supporting COM interface. Typically used to get a pointer to an IC IShellDetails interface.
ParseDisplayName()	Retrieves a pidl from a file system path.
SetNameOf()	Renames an item in the folder.

Understanding pidls

Most `IShellFolder` functions take a pidl as a parameter. A pidl is a pointer to an array of item IDs. Each item ID in the list represents a level in the shell namespace. Take this path, for example:

```
c:\Windows\System\shell32.dll
```

In this case, the pidl for this path and filename will contain five item IDs, one for each level in the shell hierarchy:

```
My Computer
  c:\
    Windows
      System
        shell32.dll
```

A pidl by itself is practically useless. Put another way, you cannot inspect a pidl with the C++Builder debugger and hope to find any meaningful information. Instead, you must call the appropriate Win32 API shell function or one of the `IShellFolder` functions to obtain information about the pidl.

Pidls are without question the most confusing aspect of dealing with the shell. This is true for several reasons. For one, an item ID list is a variable-length structure. This makes it difficult to determine how many items are in the pidl, how to reference a particular item in the pidl, or how to find one item ID relative to another (to find an item's parent item ID, for example).

Another reason pidls are confusing is that they are created by the shell relative to the parent folder. In the previous example, I said that the pidl would contain five item IDs. This assumes that the parent folder used to obtain the pidl was the desktop folder (I'll explain the desktop folder a little later). If, however, the parent folder were C:\WINDOWS\SYSTEM, then the pidl would only contain one item, the item ID of the SHELL32.DLL file. With that in mind, **Table B** lists the types of pidls you may encounter and a description

of each.

Table B: Types of pidls returned by the shell.

Pidl TypeDescription

simpleContains a single item ID relative to the parent folder.

complexContains multiple item IDs relative to the parent folder.

fully-qualifiedContains one or more item IDs, but always relative to the desktop folder.

Some shell operations require a fully-qualified pidl in order to work properly. Other shell operations require a simple pidl. You will see examples later on in the article and you will learn more about pidls at that time.

It all starts at the desktop

The first step in enumerating the Windows shell is to obtain an `IShellFolder` interface for the desktop folder. The desktop folder is the root of all shell folders and, by extension, all items within those folders. You obtain an `IShellFolder` interface for the desktop folder by calling the `SHGetDesktopFolder()` function. Here's how the code looks:

```
LPSHELLFOLDER DesktopFolder;
SHGetDesktopFolder(&DesktopFolder);
```

Obviously, this step is simple. Once you have the folder object for the desktop folder you can move on to enumerating the folders and items contained within the shell.

Enumerating folders

After obtaining a pointer to the desktop folder you must call `EnumObjects()` to obtain a pointer to an `IEnumIDList` interface for the folder. The call to `EnumObjects()` looks like this:

```
LPENUMIDLIST Enum;
DWORD EnumFlags = SHCONTF_FOLDERS;
DesktopFolder->EnumObjects(
  Handle, EnumFlags, &Enum);
```

The `EnumObjects()` function takes three parameters. The first parameter is the window handle that will serve as the parent for any dialogs that Windows displays during the enumeration. For example, if you attempt to enumerate the folder representing a floppy drive and there is no diskette in the drive, Windows will

automatically display a message indicating that the drive is not ready.

The second parameter is a set of flags that determine how the enumeration operates. In this example, only the SHCONTF_FOLDERS constant is used. This will cause the shell to only enumerate objects that are themselves folders. Other constants for the flags parameter include SHCONTF_NONFOLDERS (for items that are not folders) and SHCONTF_INCLUDEHIDDEN (for enumerating hidden files and folders). If you wanted to enumerate all objects in a folder, then, you would declare the EnumFlags variable like this:

```
DWORD EnumFlags =
  SHCONTF_FOLDERS | SHCONTF_NONFOLDERS |
  SHCONTF_INCLUDEHIDDEN;
```

The final parameter for EnumObjects() is the address of a variable that will contain a pointer to the IEnumIDList interface if the call succeeds. You must check the return value of EnumObjects() to see if the call succeeded. If you do not, you will get an access violation when you attempt to use the pointer if EnumObjects() fails.

Once you have a pointer to an IEnumIDList interface you can enumerate all of the objects in the parent folder (the desktop in this case). This is done by calling the Next() function of IEnumIDList. Here's an example:

```
DWORD Result = Enum->Next(1, &Pidl, 0);
```

The first parameter of this function is the number of items you wish to retrieve. In theory, you should be able to specify any number of items to retrieve. In reality, though, I have never seen an example that specifies anything other than 1 for this parameter. Further, passing a value greater than 1 appears to have no effect. The second parameter is a pointer to an item ID list. If the call to Next() is successful, this parameter will contain the pidl for an item in the folder. I'll explain what to do with this pidl in a later section. The third parameter is a pointer to a DWORD that, on success, will return the number of items enumerated. If you are not interested in the number of items enumerated you can set this parameter to NULL. Next() will return NOERROR (0) on success, S_FAIL (1) if there are no more items to enumerate, or an error code if an error occurs.

Typically you will enumerate objects using a while loop:

```
LPITEMIDLIST Pidl;
DWORD Result = Enum->Next(1, &Pidl, 0);
while (Result != S_FALSE) {
  if (Result != NOERROR)
    break;
  // do something with Pidl
  SHGetMalloc(&Malloc);
  Malloc->Free(Pidl);
  Malloc->Release();
```

```
  Result = Enum->Next(1, &Pidl, 0);
}
Enum->Release();
```

The first line of code in the `while` loop checks the value of `Result` to see if some error occurred during enumeration. For example, an error may occur if a particular folder is no longer valid. If an error occurs, the loop is terminated. Naturally, you must do something with the pidl returned by the `Next()` function. I will address that in the next section. After the pidl is processed, the shell's memory allocator is used to free the pidl returned by the `Next()` function. If you do not perform this step, your program will leak memory. When the loop terminates, the memory allocated for the `IEnumIDList` interface is freed using the `Release()` function.

Putting the pidl to good use

Once you have a pidl you can use it to obtain information about the folder item that the pidl represents. The information you can obtain from a pidl includes:

- The display name
- The full path (for file system objects)
- The large and small icon
- The file type
- The item's attributes

There are two ways to obtain this information. One way is to use the functions of `IShellFolder`. The other way is to use the shell's `SHGetFileDetails()` function. In this section I will explain how to get the display name and path using `IShellFolder`'s `GetDisplayNameOf()` and `GetAttributesOf()` functions.

Getting the display name

The display name of the item is obtained using the `GetDisplayNameOf()` function. `GetDisplayNameOf()` takes three parameters. The first parameter is the pidl of the item for which you want to obtain the display name. The second parameter is used to specify flags that determine the way the display name will be returned. The third parameter is a pointer to a `STRRET` structure. `STRRET` will contain the display name if the function returns successfully.

The flags parameter is specified by type. The `SHGDN_NORMAL` value is used to indicate that the pidl should be evaluated relative to the desktop. The `SHGDN_INFOLDER` value is used when you want the display name relative to the parent folder, and not relative to the desktop. In addition to these values, you can also specify modifier values. The only modifier of significance is `SHGDN_FORPARSING`. When this value is present in the flags parameter, the shell will return the path (and filename if applicable) rather than the display name. When the `SHGDN_NORMAL` flag is used alone, the shell will return the display name of an item as you see it in Explorer.

Here's how a call to `GetDisplayNameOf()` might look:

```
STRRET StrRet;
StrRet.uType = STRRET_CSTR;
ParentFolder->GetDisplayNameOf(
    Pidl, SHGDN_NORMAL, &StrRet);
```

This looks easy enough, but the real work comes in extracting the display name from the `STRRET` structure. Here is the declaration for `STRRET`:

```
typedef struct _STRRET
{
  UINT uType;
  union
  {
    LPWSTR pOleStr;
    LPSTR  pStr;
    UINT   uOffset;
    char   cStr[MAX_PATH];
  } DUMMYUNIONNAME;
} STRRET, *LPSTRRET;
```

A display name can be returned in one of three forms. **Table C** lists the constants that represent the different forms and a description of each. You can specify the way that you want the string returned by setting the `uType` member to one of the values in **Table C** prior to calling `GetDisplayNameOf()`. However, this is often an exercise in futility, because Windows will determine how the string is returned. After the call to `GetDisplayNameOf()` you must check the value of the `uType` member to see how Windows returned the string. For example:

```
String DisplayName;
switch (StrRet.uType) {
  case STRRET_CSTR :
    DisplayName = StrRet.cStr;
    break;
  case STRRET_WSTR :
    DisplayName = WideCharToString(StrRet.pOleStr);
    break;
  case STRRET_OFFSET :
    DisplayName = ((char*)Pidl) + StrRet.uOffset;
}
```

It has been my experience that Windows most often returns the string as an offset into the pidl.

Table C: STRRET display type constants.

Type ConstantDescription

`STRRET_WSTR`The string is returned in `pOleStr` as a wide string.

`STRRET_OFFSET`The string is embedded in the item ID list. The `uOffset` member contains the offset value.

`STRRET_CSTR`The string is returned in `cStr` as a null-terminated string.

The `SHGetFileInfo()` function provides an easier way of getting the display name for an item. However, it is not as flexible as `GetDisplayNameOf()`. For example, `SHGetFileInfo()` will give you only the display name as displayed in Explorer. It cannot give you the path of a particular item. `SHGetFileInfo()` does, however, give you other important information about a pidl. I will discuss `SHGetFileInfo()` in more detail next month when I explain how to get display icons for shell items.

Getting the attributes

When I speak of getting the attributes, I am not talking about the file attributes. Rather, I am talking shell item attributes. The list of possible attributes are listed under the `IShellFolder::GetAttributesOf()` topic on the Microsoft Developer Network (MSDN) CD. If you do not have MSDN you can find the attribute constants in SHLOBJ.H. The entire list of attributes is too long to list here, but **Table D** shows the most commonly used attributes.

Table D: The most commonly used shell item attributes.

Attribute ConstantDescription

`SFGAO_CANCOPY`The item can be copied.

`SFGAO_CANMOVE`The item can be moved.

`SFGAO_CANRENAME`The item can be renamed.

`SFGAO_CANDELETE`The item can be deleted.

`SFGAO_LINK`The item is a shortcut.

`SFGAO_SHARE`The item is shared (for shared drives, printers, and folders).

SFGAO_FOLDERThe item is a folder that may contain other items.

SFGAO_FILESYSTEMThe item is part of the file subsystem.

SFGAO_HASSUBFOLDERThe item has subfolders.

SFGAO_REMOVABLEThe item has removable media.

SFGAO_COMPRESSEDThe item is in a compressed folder.

You retrieve item attributes using the IShellFolder::GetAttributesOf() function. GetAttributesOf() takes three parameters. The first parameter is used to specify the number of items for which you want to obtain the attributes. This parameter is usually set to 1. The second parameter is the address of the pidl representing the item within the folder. The final parameter is the address of a DWORD variable that will contain the attributes of the item if GetAttributesOf() is successful. You request specific attributes by setting the attributes variable prior to calling GetAttributesOf(). For example:

```
DWORD Attr = SFGAO_FOLDER;
ParentFolder->GetAttributesOf(
  1, (LPCITEMIDLIST*)&Pidl, &Attr);
if ((Attr & SFGAO_FOLDER)==SFGAO_FOLDER)
  // it's a folder
```

This code shows how to determine if a particular item in a folder is itself a folder. You can request as many attributes as you want for a single call to GetAttributesOf(). For example, let's say you wanted to know if the given item was a folder in the file system and whether or not the folder was compressed. In that case you would set up the Attr variable like this:

```
DWORD Attr = SFGAO_FOLDER |
  SFGAO_FILESYSTEM | SFGAO_COMPRESSED;
```

After calling GetAttributesOf() you can check the value of Attr to see if a specific attribute exists.

There is one important aspect of attributes that you need to be aware of. You may be tempted to simply set the Attr variable to all available attribute constants. Doing so, however, will dramatically slow down the call to GetAttributesOf(). If you are enumerating the entire file system this will mean that your enumeration will be very slow. It is best to set the Attr variable to only those attributes you need for a given type of enumeration.

Getting an IShellFolder from a pidl

Another thing you may want to do with a pidl is to obtain an IShellFolder interface for the folder that the pidl represents. To do this, you call the BindToObject() function of IShellFolder. Given a pidl and

a parent folder you obtain an `IShellFolder` interface for the pidl like this:

```
ParentFolder->BindToObject(Pidl,
  0, IID_IShellFolder, (void**)&Folder);
```

If the call to `BindToObject()` is successful, the return value will be `NOERROR` and the `Folder` variable will contain the address of the interface. `BindToObject()` will fail if the shell item represented by the pidl is not a folder object, or if the pidl is not a direct child of the parent folder.

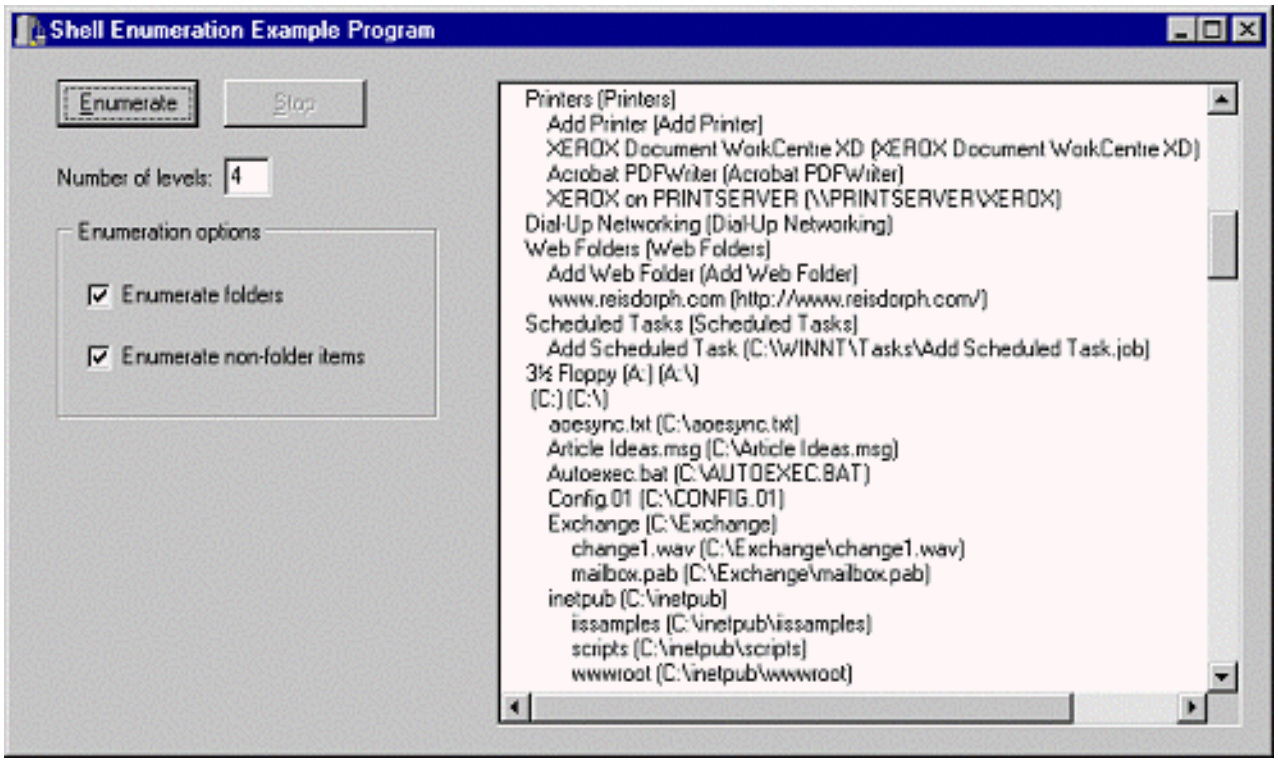
The example program

The example program for this article can be found in ENUMSHELL.BPR. The main unit's header and source code are shown in **Listings A** and **B**. This program enumerates the shell using parameters set on the main form. The parameters include whether you want to enumerate folders or non-folder items, and how many levels to enumerate.

The results are displayed in a memo on the main form. The display name is shown in the memo, followed by the full path.

The `Enumerate()` method of the main form uses recursion to enumerate additional levels, obtaining an `IShellFolder` interface for each new folder as it goes. Bear in mind that if you are on a network, the example program may appear to hang when enumerating network drives more than two levels deep, and if you are enumerating non-folder items. **Figure A** shows the main form when the application is run.

Figure A



The example program enumerates the shell given the specified number of levels and the enumeration options.

In order to keep the code as basic as possible, the example program does not contain features that you may need in your applications. For example, before enumerating a particular drive you should check the drive to see if has removable media and whether the media is present. In the case of network drives, you should check to see if the drive is connected before attempting to enumerate that drive. These are not required features, as the shell will display the appropriate dialog when it encounters these situations. Still, you may want to handle these situations yourself rather than depending on the shell.

Conclusion

Enumerating the Windows shell is not trivial, especially if you are approaching it without the benefit of this article. This article explained how to use `IShellFolder` and `IEnumIDList` to enumerate the shell. By using the techniques in this article you can add shell enumeration features to your applications.

Next month I will explain how to get display icons for shell items, and how to display the context menu for a shell item.

Listing A: *MAINU.H*

```
#ifndef MainUH
#define MainUH

#include <Classes.hpp>
#include <Controls.hpp>
```

```
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>

class TForm1 : public TForm
{
__published:
  TMemo *Memo1;
  TGroupBox *GroupBox1;
  TCheckBox *EnumFoldersCb;
  TCheckBox *EnumNonFoldersCb;
  TButton *EnumBtn;
  TEdit *NumLevelsEdit;
  TLabel *Label1;
  TButton *StopBtn;
  void __fastcall EnumBtnClick(TObject *Sender);
  void __fastcall StopBtnClick(TObject *Sender);
private:
  LPMALLOC Malloc;
  int NumLevels;
  bool Abort;
  void Enumerate(IShellFolder* ParentFolder,
    bool EnumNonFolders, int levels);
  String GetDisplayName(LPITEMIDLIST Pidl,
    IShellFolder* ParentFolder,
    DWORD type = SHGDN_NORMAL);
public:
  __fastcall TForm1(TComponent* Owner);
  __fastcall ~TForm1();
};

extern PACKAGE TForm1 *Form1;

#endif
```

Listing B: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include <shellapi.h>
#include <shlobj.h>
#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
```

```
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  // Get the shell's IMalloc interface.
  SHGetMalloc(&Malloc);
}

__fastcall TForm1::~TForm1()
{
  // Release the IMalloc interface.
  Malloc->Release();
}

void TForm1::Enumerate(IShellFolder* ParentFolder,
  bool EnumNonFolders, int Levels)
{
  LPITEMIDLIST Pidl;
  LPENUMIDLIST Enum;
  // Set up the enumeration flags based on the
  // main form's enum options check boxes.
  DWORD EnumFlags = SHCONTF_FOLDERS;
  if (EnumNonFolders)
    EnumFlags |= SHCONTF_NONFOLDERS;
  DWORD Result = ParentFolder->EnumObjects(
        Handle, EnumFlags, &Enum);
  // Error - return.
  if (Result != NOERROR)
    return;
  String DisplayName;
  // Get the pidl for the first item in the folder.
  Result = Enum->Next(1, &Pidl, 0);
  while (Result != S_FALSE) {
    int CurrentLevel = Levels;
    // If result is something other than
    // NOERROR then some error occured.
    if (Result != NOERROR)
      break;
    // Get the "normal" display name.
    DisplayName =
      GetDisplayName(Pidl, ParentFolder);
    // Add spaces to the string so each level
    // is indented.
    String PadStr = String().StringOfChar(
      ' ', (NumLevels - CurrentLevel) * 4);
```

```
    PadStr = PadStr + DisplayName;
    // Now get the full path using the
    // SHGDN_FORPARSING constant.
    DisplayName = GetDisplayName(
      Pidl, ParentFolder, SHGDN_FORPARSING);
    // Add it to the end of the string and
    // display the full string in the memo.
    Memo1->Lines->Add(
      PadStr + " (" + DisplayName + ")");
    CurrentLevel--;
    // See if this shell item is a folder.
    DWORD Attr = SFGAO_FOLDER;
    ParentFolder->GetAttributesOf(
      1, (LPCITEMIDLIST*)&Pidl, &Attr);
    if ((CurrentLevel > 0)
        && (Attr & SFGAO_FOLDER) == SFGAO_FOLDER) {
      LPSHELLFOLDER Folder;
      // Get the IShellFolder for the pidl.
      int res = ParentFolder->BindToObject(Pidl,
        0, IID_IShellFolder, (void**)&Folder);
      if (res == NOERROR) {
        // Recurse by calling Enumerate again.
        Enumerate(
          Folder, EnumNonFolders, CurrentLevel);
        // Free the memory for the new folder.
        Folder->Release();
      }
    }
    // Free the memory for the pidl returned
    // by the Enum->Next() function.
    Malloc->Free(Pidl);
    // If the user pressed the Stop button then
    // stop enumerating.
    Application->ProcessMessages();
    if (Abort)
      break;
    // Get a pidl for the next item in the folder.
    Result = Enum->Next(1, &Pidl, 0);
  }
  // Free the IEnumIDList interface.
  Enum->Release();
}

void __fastcall
TForm1::EnumBtnClick(TObject *Sender)
```

```
{
  StopBtn->Enabled = true;
  Abort = false;
  Memo1->Cursor = crHourGlass;
  Memo1->Lines->Clear();
  // Start with the desktop folder.
  LPSHELLFOLDER DesktopFolder;
  SHGetDesktopFolder(&DesktopFolder);
  // Set the number of levels.
  NumLevels = NumLevelsEdit->Text.ToIntDef(2);
  // Start enumerating. Enumerate is a recursive
  // function so we only call it once from here.
  Enumerate(DesktopFolder,
    EnumNonFoldersCb->Checked, NumLevels);
  // Free the IShellFolder for the desktop folder.
  DesktopFolder->Release();
  StopBtn->Enabled = false;
  Memo1->Cursor = crDefault;
}

String TForm1::GetDisplayName(LPITEMIDLIST Pidl,
  IShellFolder* ParentFolder, DWORD type)
{
  String DisplayName;
  STRRET StrRet;
  // Request the string as a char* although
  // Windows will likely ignore the request.
  StrRet.uType = STRRET_CSTR;
  // Call GetDisplayNameOf() to fill in the
  // STRRET structure.
  ParentFolder->GetDisplayNameOf(
    Pidl, type, &StrRet);
  // Extract the string based on the value
  // of the uType member of STRRET.
  switch (StrRet.uType) {
    case STRRET_CSTR :
      DisplayName = StrRet.cStr;
      break;
    case STRRET_WSTR :
      DisplayName =WideCharToString(StrRet.pOleStr);
      break;
    case STRRET_OFFSET :
      DisplayName = ((char*)Pidl) + StrRet.uOffset;
  }
  // Return the result.
```

```
  return DisplayName;
}

void __fastcall TForm1::StopBtnClick(TObject *Sender)
{
  // User clicked the Stop button so set the
  // Abort flag.
  Abort = true;
  StopBtn->Enabled = false;
  Memo1->Cursor = crDefault;
}
```

Fast updates with virtual list views

by Kent Reisdorph

The `TListView` component is a powerful component. It is useful for situations where you want to present a list of items to the user, to allow the user to view those items in a number of ways, and to display icons next to each item. `TListView` provides three view styles; report (typically known as the details view), large icon, small icon, and list. `TListView` has one major shortcoming, though. If you have a lot of items to display then `TListView` can be very slow.

This article will explain how to use `TListView` as a virtual list view. I will introduce you to the `OwnerData` property, and the `OnData` and `OnDataFind` events. I will also show you how to store your data separate from the list view. Virtual list view support appeared in the VCL in C++Builder 4. It is not available in C++Builder 1 and 3.

Normal vs. virtual list view

A virtual list view is much faster than a normal list view. It is faster because the list view does not actually store any data. To illustrate, consider the case where you have 1,000 items to present in a list view. (At first this might seem extreme, but it is actually quite common.) Let's say, for example, that you are displaying the contents of a file folder in a list view. In this case you could easily have 1,000 items to display. Using a normal list view, you would have to add all of the items to the list view. Further, if you wanted to display the contents of a different directory, you would have to remove all of the existing items before adding the new items. This all takes a considerable amount of time. You can speed up the process by disabling screen updates while you are removing and adding items, but it still takes a long time.

A virtual list view, on the other hand, doesn't contain any items at all. You store your data in some external storage, such as a `TList` or STL `vector`. When the list view needs to paint items, it requests the information for each item from you through the `OnData` event. Depending on screen resolution, a list view can only display between 100 and 200 items on the screen at one time. This means that the list view is only requesting data for that number of items, rather than for the entire list. The disadvantage of a virtual list view is that you must maintain the list of items separate from the list view. Still, this is an acceptable tradeoff for those times where you need the speed of a virtual list view.

So how much faster is a virtual list view? The example program for this article fills the list view with the contents of a directory. On my system, the WINNT\SYSTEM32 directory contains 1680 files and folders. Using a regular list view, adding all items takes about 550 milliseconds. Using a virtual list view, it only takes about 40ms. That's an impressive difference, but it gets even better. Removing all items from the regular list view and adding them back in again takes over 1,000ms. With the virtual list view it takes about 50ms. That's a speed increase of over 20 fold.

Storing your data

The first thing you need to do is to decide how you are going to store your data. For this article's example program, I chose to use a `TList` to store the item data and created a structure called `TFileItem` for this purpose. Here's the declaration:

```
struct TFileItem {
  String FileName;
  String FullPath;
  int IconIndex;
  bool IsFolder;
};
```

This structure contains the basic information I will need in order to display a list view item, as well as some additional information that I will use to sort the items. I used a typical `FindFirst()`/`FindNext()` loop to iterate through the items in the folder. Here's an abbreviated version of the code:

```
TList* FileList = new TList;
ListView1->OwnerData = true;
TSearchRec SR;
FindFirst(PathEdit->Text +
  "\\*.*", faAnyFile, SR);
while (FindNext(SR) == 0) {
  TFileItem* fi = new TFileItem;
  fi->FileName = SR.FileName;
  fi->IsFolder =
    (SR.Attr & faDirectory) != 0;
  FileList->Add(fi);
}
FindClose(SR);
ListView1->Items->Count =FileList->Count;
```

First I create a `TList` object to hold the items. Next, I set the `OwnerData` property to `true`. I could have set this property in the Object Inspector but I put it in the code to illustrate that `OwnerData` must be set to `true` for a virtual list view. Next I start the file search by calling `FindFirst()`. `PathEdit` is a `TEdit` on my main form that is used to specify the starting path. You may have noticed that I don't do anything with the original file found by `FindFirst()`. This is because the first item returned by `FindFirst()` is a single period representing the current directory.

In the `while` loop I create an instance of the `TFileItem` structure, set a few of its members, and then

add it to the `TList`. Note that I do not delete the pointer contained in the `fi` variable. I'll do this later when I clear the `TList`.

When the `while` loop ends, I call `FindClose()` to free the memory allocated for the file search operation. Finally, I set `ListView1->Items->Count` to the number of items in the file list. This is an important step because it forces the list view to updated. Before this line executes, the list view has no knowledge that it has items to display.

I said earlier that this is an abbreviated version of the code. It is abbreviated because in the example program's code I assign values to the remaining structure members. The example program includes an option to use the list view as a normal list view or as a virtual list view. I removed the code that adds the items to the normal list view because it is not pertinent to this part of the article. Naturally, I will need to run through the item list at some point and free the memory for the `TFileItem` objects in the list. Doing so is standard `TList` fare so I won't elaborate further.

Handling the OnData event

The VCL will fire the `OnData` event whenever Windows needs to draw an item in the list view. `OnData` gives you a pointer to the list item that is about to be displayed. You use that pointer (a `TListItem*`) to assign the item's text, its icon, and so on. Here is the example program's `OnData` event handler:

```
void __fastcall TForm1::ListView1Data(
  TObject *Sender, TListItem *Item)
{
  TFileItem* fi = (TFileItem*)
    FileList->Items[Item->Index];
  Item->Caption = fi->FileName;
  Item->ImageIndex = fi->IconIndex;
}
```

The first line of code extracts the `TFileItem` object corresponding to the item that is about to be displayed. The `Index` property of the item corresponds to an item in the file list. Next I set the item's `Caption` property to the `FileName` member of the `TFileItem` object, and the `ImageIndex` property of the item to the `IconIndex` member.

In the previous code I didn't show how I set the icon index so I'll explain that now. The example program contains a `TImageList` component that has three images. The first image is an icon representing a file folder, the second image is the standard Windows file icon, and the third is the default Windows icon for an executable file. In the file search loop, I set the `IconIndex` member of the `TFileItem` structure to `0`, `1`, or `2` depending on the file item attributes.

That's all there is to handling the `OnData` event. The VCL gives you the `TListItem` pointer representing the item about to be displayed, and you set any of the item's properties as needed.

More virtual list view events

The `OnData` event is the only event that you are required to respond to if you are using a virtual list view. Other events that you can optionally respond to are `OnDataFind`, `OnDataHint`, and `OnDataStateChange`.

Create an event handler for the `OnDataFind` event if you plan to search for items in the list view using the `FindCaption()` or `FindData()` methods. The example program, for example, allows you to search for a file or folder by name. The code that finds the item looks like this:

```
TListItem* item =
  ListView1->FindCaption(0,
   SearchEdit->Text, false, true, false);
```

When this code executes, the `OnDataFind` event will fire. Here is the pertinent code in the example program's `OnDataFind` handler:

```
for (int i=0;i<FileList->Count;i++) {
  TFileItem* item =
    (TFileItem*)FileList->Items[i];
  if (UpperCase(item->FileName) ==
      UpperCase(FindString)) {
    Index = i;
    break;
  }
}
```

In this code, `FindString` is the search string passed in the event handler. `Index` is a parameter passed by reference that is used to return the index of the matching item. I simply run through the file list and, if a matching item is found, I set the `Index` parameter to the index of the item in the file list. See **Listing B** for the complete `OnDataFind` event handler.

`OnDataHint` is used in special cases where you need even faster list view updates. The `OnDataStateChange` event is fired when the state of a range of items changes. The VCL help is a bit spotty on these two event handlers so some experimentation will be required in order to implement them. The good news is that neither event is specifically required for virtual list views.

Sorting the virtual list view

Sorting the items in a virtual list view is a matter of sorting the underlying list itself. After I fill the list of file items, I sort the list:

```
FileList->Sort(ListSortFunc);
```

Obviously, this code doesn't tell you much. The real work is done in the `ListSortFunc()` function:

```
int __fastcall
ListSortFunc(void* Item1, void* Item2)
{
 TFileItem* item1 = (TFileItem*)Item1;
 TFileItem* item2 = (TFileItem*)Item2;
 if (item1->IsFolder && !item2->IsFolder)
   return -1;
 if (item2->IsFolder && !item1->IsFolder)
   return 1;
 return CompareText(
   item1->FileName, item2->FileName);
}
```

This is a basic `TList` sort function, where you return `-1` from the function if item one is less than item two, return `1` if item one is greater than item two, or `0` if the two items are the same. In this case I return `–1` if the first item is a directory and the second item is not. Conversely, I return `1` if the second item is a folder and the first is not. Finally, if neither of these conditions is met, I return the result of a string compare between the two file (or folder) names. The result is that the list is sorted with the folders at the top and the individual files after the folders. Both the folders and the files are sorted by name.

Sorting the list can be much more complex than the simple sort shown here. However complex your sort, remember to keep the code in the sort function as efficient as possible. This will drastically reduce the time required to perform the sort.

Finally, consider using a profiler to improve the performance of your sort functions. I used TurboPower's Sleuth QA Suite to profile my list sort function. I found out that to sort the 1680 items in my SYSTEM32 directory, the sort function was called 20,948 times. The total time spent in the function, however, was only 11.16 milliseconds. Certainly I can live with a sort that only takes a fraction of a second.

Conclusion

Virtual list views should be used any time you need fast updates for your list view controls. Managing the list of items takes a bit of work but it is well worth it in the long run. **Listings A** and **B** are the header and main unit for the example program. Examine them to get the full picture of how to use virtual list views. As an added bonus, the example program shows you how to sort a regular list view as well.

Listing A: *MAINU.H*

```
#ifndef MainUH
#define MainUH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include <ImgList.hpp>

// The structure that holds file items.
struct TFileItem {
  String FileName;
  String FullPath;
  int IconIndex;
  bool IsFolder;
};

class TForm1 : public TForm
{
__published:    // IDE-managed Components
  TPanel *Panel1;
  TListView *ListView1;
  TLabel *Label1;
  TEdit *PathEdit;
  TButton *FillViewBtn;
  TImageList *LargeImages;
  TRadioButton *VirtualRb;
  TRadioButton *NormalRb;
  TLabel *Label2;
  TLabel *Label3;
  TButton *SearchBtn;
  TLabel *Label4;
  TEdit *SearchEdit;
  void __fastcall FillViewBtnClick(TObject *Sender);
```

```
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall ListView1Data(
    TObject *Sender, TListItem *Item);
  void __fastcall ListView1Click(TObject *Sender);
  void __fastcall SearchBtnClick(TObject *Sender);
  void __fastcall ListView1DataFind(TObject *Sender,
    TItemFind Find, const AnsiString FindString,
    const TPoint &FindPosition, Pointer FindData,
    int StartIndex, TSearchDirection Direction,
    bool Wrap, int &Index);
private:    // User declarations
  TList* FileList;
  void ClearList();
public:        // User declarations
  __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;

#endif
```

Listing B: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

int __fastcall
ListSortFunc(void* Item1, void* Item2)
{
  TFileItem* item1 = (TFileItem*)Item1;
  TFileItem* item2 = (TFileItem*)Item2;
  // Folder items come first so sort by folder.
  if (item1->IsFolder && !item2->IsFolder)
    return -1;
  if (item2->IsFolder && !item1->IsFolder)
    return 1;
  // Now sort by the display name of each item.
  return CompareText(
```

```
    item1->FileName, item2->FileName);
}

int __stdcall ListViewSortFunc(
  LPARAM Item1, LPARAM Item2, LPARAM Data)
{
  // This is the sort function for the list view
  // in normal mode. It is essentially the same
  // as the sort function for the TList.
  TListItem* item1 = (TListItem*)Item1;
  TListItem* item2 = (TListItem*)Item2;
  if (!item1->ImageIndex && item2->ImageIndex)
    return -1;
  if (item1->ImageIndex && !item2->ImageIndex)
    return 1;
  return
    CompareText(item1->Caption, item2->Caption);
}

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  FileList = new TList;
  // Assign the form's TImageList to the
  // list view's LargeImages property.
  ListView1->LargeImages = LargeImages;
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  ClearList();
  delete FileList;
}

void __fastcall
TForm1::FillViewBtnClick(TObject *Sender)
{
  int startTime;
  if (VirtualRb->Checked) {
    //OwnerData must be true for virtual list views
    ListView1->OwnerData = true;
    startTime = GetTickCount();
    // Clear out the items and the TList.
    ListView1->Items->Clear();
```

```
  ClearList();
}
else {
  // Disable updates for the normal list view.
  ListView1->Perform(WM_SETREDRAW, 0, 0);
  ListView1->OwnerData = false;
  startTime = GetTickCount();
  ListView1->Items->Clear();
}
TSearchRec SR;
FindFirst(
  PathEdit->Text + "\\*.*", faAnyFile, SR);
while (FindNext(SR) == 0) {
  if (SR.Name == "..")
    continue;
  // Get the next file in the folder.
  String FileName = ExtractFileName(SR.Name);
  int IconIndex;
  // Set the IconIndex property based on type.
  if ((SR.Attr & faDirectory) != 0)
    IconIndex = 0;
  else if (ExtractFileExt(SR.Name) == ".exe")
    IconIndex = 2;
  else
    IconIndex = 1;
  if (VirtualRb->Checked) {
    // Create a new TFileItem, set its properties
    TFileItem* fi = new TFileItem;
    fi->FileName = FileName;
    fi->FullPath =
      PathEdit->Text + "\\" + SR.Name;
    fi->IconIndex = IconIndex;
    fi->IsFolder = (SR.Attr & faDirectory) != 0;
    // Add the item to the list.
    FileList->Add(fi);
  }
  else {
    // Normal list view stuff.
    TListItem* item = ListView1->Items->Add();
    item->Caption = FileName;
    item->ImageIndex = IconIndex;
  }
}
```

```
  FindClose(SR);
  if (VirtualRb->Checked) {
    // Sort the list.
    FileList->Sort(ListSortFunc);
    // This is the part that causes the virtual
    // list view to begin updating.
    ListView1->Items->Count = FileList->Count;
  }
  else {
    // Sort the normal list view.
    ListView1->CustomSort(ListViewSortFunc, 0);
    // Enable updates again.
    ListView1->Perform(WM_SETREDRAW, 1, 0);
  }
  Label2->Caption = "Elapsed time: " +
    String(GetTickCount() - startTime) + "ms for "+
    String(ListView1->Items->Count) + " items";
}

void TForm1::ClearList()
{
  // Clear out the file list each time.
  for (int i=0;i<FileList->Count;i++)
    delete ((TFileItem*)FileList->Items[i]);
  FileList->Clear();
}

void __fastcall TForm1::ListView1Data(
  TObject *Sender, TListItem *Item)
{
  // Get a TFileItem pointer from the file list.
  TFileItem* fi =
    (TFileItem*)FileList->Items[Item->Index];
  // Set the Caption and ImageIndex properties.
  Item->Caption = fi->FileName;
  Item->ImageIndex = fi->IconIndex;
}

void __fastcall
TForm1::ListView1Click(TObject *Sender)
{
  // A list view item may have been selected. If
  // so, update a label on the form with the path
  // of the item that was selected.
```

```
  if (VirtualRb->Checked && ListView1->Selected) {
    TFileItem* fi = (TFileItem*)
      FileList->Items[ListView1->Selected->Index];
    Label3->Caption = fi->FullPath;
  }
}

void __fastcall
TForm1::SearchBtnClick(TObject *Sender)
{
  // Finds an item and, if found, selects it.
  TListItem* item = ListView1->FindCaption(
    0, SearchEdit->Text, false, true, false);
  ListView1->Selected = item;
  ListView1->SetFocus();
  // Windows macro to make sure the found item
  // is visible.
  ListView_EnsureVisible(
    ListView1->Handle, item->Index, false);
}

void __fastcall TForm1::ListView1DataFind(
  TObject *Sender, TItemFind Find,
  const AnsiString FindString,
  const TPoint &FindPosition, Pointer FindData,
  int StartIndex, TSearchDirection Direction,
  bool Wrap, int &Index)
{
  // This event handler is called when the
  // FindCaption method is called. Find the item.
  for (int i=0;i<FileList->Count;i++) {
    TFileItem* item =(TFileItem*)FileList->Items[i];
    if (UpperCase(item->FileName) ==
        UpperCase(FindString)) {
      Index = i;
      break;
    }
  }
}
```

Using Perl compatible regular expressions in VCL applications

by Jeffrey J. Peters

When I first started working for Borland, back in 1990, I didn't know anything about regular expressions. In fact, I don't think I'd even heard of GREP (a text search tool provided with Borland compilers) much less used it. Eventually I did discover GREP, but I only used it for simple plain text searching. Eventually someone introduced me to the use of `.*` and simple character classes like `[a-z0-9].` I was able to get by on that small subset of regular expressions with `GREP` for the next 7 years.

Then, only a few years ago, I discovered Perl (Perl is an acronym for Practical Extraction and Reporting Language). One of my co-workers, an avid Unix fanatic, solved a problem in the build process of one of our products by using a piece of Perl code. Later he taught an introductory class on Perl to the rest of us who were unfamiliar with it. Since Perl is an interpreted languag, I have to admit that I was quite skeptical of Perl's usefulness (although recent versions of Perl now compile to p-code). My background was in various flavors of assembly and C, followed by C++. I was not prepared to admit that a simple, interpreted "script" could be any more useful than a DOS batch file. As it turns out, I wrong in that assessment. I found Perl so powerful that, after I learned enough about it to be comfortable, I replaced numerous batch files and several custom C programs with Perl scripts. Those scripts are now vastly more powerful than their original versions ever were.

Perl's power stems from several key features, the most important of which are its support of regular expressions. Perl is a scripting language and the purpose of supporting regular expressions is to allow the script to interact with the data it can search for. Considering this, it makes sense that there would be interesting features built into Perl for doing this. For example, Perl allows you to search a string for a regular expression pattern (regexp) that contains groups. The technical term is called "capturing sub-expressions" and it refers to the groups of parentheses in the expression. Take this regexp, for example:

```
^My (cat|dog) has (fleas|paws)$
```

This expression will match all of the following four strings:

```
My cat has fleas
My dog has fleas
My cat has paws
My dog has paws
```

Now if you wanted the script to do something different depending on the various words that actually were matched, you can make use of Perl's grouping variables. In this case the group `$1` would contain

the matched string from the first group (either cat or dog) and `$2` from the second group (either fleas or paws). This way the script could do one complex regexp search and then continue on and use the variables that contain the grouping strings.

C++Builder's built-in support for regular expressions

C++Builder 4.0 added a new collection of routines to the RTL that implement Perl's dialect of regular expressions. These are the Perl Compatible Regular Expression (PCRE) routines. Their declarations and supporting constants can be found in PCRE.H. The functions were written by Phillip Hazel of Cambridge University and released into the public domain.

The functions in PCRE.H are `pcre_compile()`, `pcre_exec()`, `pcre_maketables()`, `pcre_info()`, `pcre_study()`, and `pcre_version()`. These C-based routines support a very high percentage of the Perl syntax for regular expressions. However, they are a bit tedious to work with directly. The pattern to be searched for must first be compiled into an efficient internal format, and then the target string or strings in which to search are processed with that compiled pattern. Successful matches result in all the captured sub-expressions being made available as two integer offsets into the target string for each match (the beginning offset and the ending offset of the sub-string). The address of an array of integers that the user allocates and the length of that array are parameters passed into the PCRE routine in order for the sub-expression offsets to be returned.

This may sound easy, but in actuality it's a bit more complex than that. In order for the PCRE routines to be thread safe, they don't use any global variables. And in the interest of speed, they do not dynamically allocate memory for use as temporary space during the matching process. As such, the user must allocate extra elements in the array of integers and that extra memory is used internally as temporary storage.

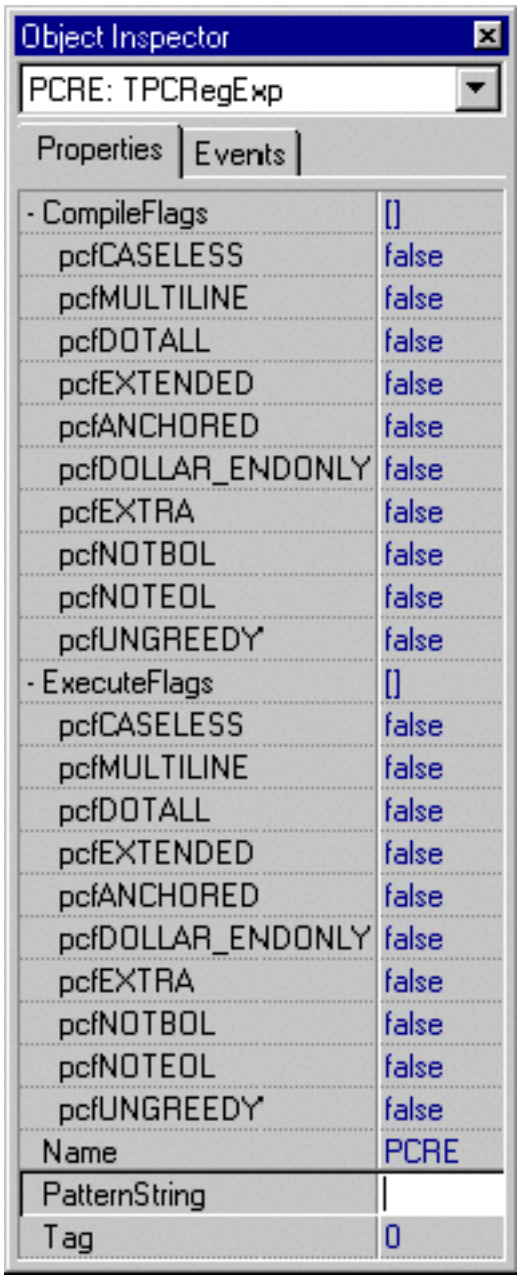
The formula for determining the proper size of the array is something like (x+1)*3, where x is the number of captured sub-expressions that you expect to use. The +1 part of the equation is to make room for a 0^{th} element that contains the entire string that was matched even if there are no parentheses around the entire pattern. The *3 part of the equation makes room for the two elements (start and end) that describe where that part of the sub-expression is in the target string, and the extra storage that the PCRE routines can use internally.

To complicate things even more, there are various flags that can be specified at either the pattern compile time or the pattern usage time to modify how the PCRE routines behave. For example, there is a flag that specifies whether the search should be case sensitive or not. The end result is that using these routines can be tedious and complicated.

The PCRE component

C++Builder's component architecture makes the PCRE routines quite easy to use. I've written a component called `TPCRegExp` that takes care of the tedious work involved in using regular expressions (see **Figure A**). The component publishes two sets called `CompileFlags` and `ExecuteFlags`. These sets are used to set the various PCRE option flags for the compile of the pattern and the execution of that pattern (searching for a match). There is also a string property called `PatternString`, which holds the regular expression text.

Figure A



The Object Inspector showing the published properties of the TPCRegExp component.

To use the component, simply set any of the flags as desired, put the regular expression into the

`PatternString` property, and call the `Compile()` function. The `Compile()` function will return `true` or `false` indicating success or failure with the pattern. If there were errors, the offending offset in the pattern and an error message string are returned as reference parameters. Here is the declaration for the `Compile()` function:

```
bool Compile(
  AnsiString &ErrorString,
  int &ErrorOffset);
```

Now the pattern is ready to be executed against a target string. This is done by calling the `Execute()` function, passing the target string and an optional maximum length of the string. The `Execute()` function is declared as follows:

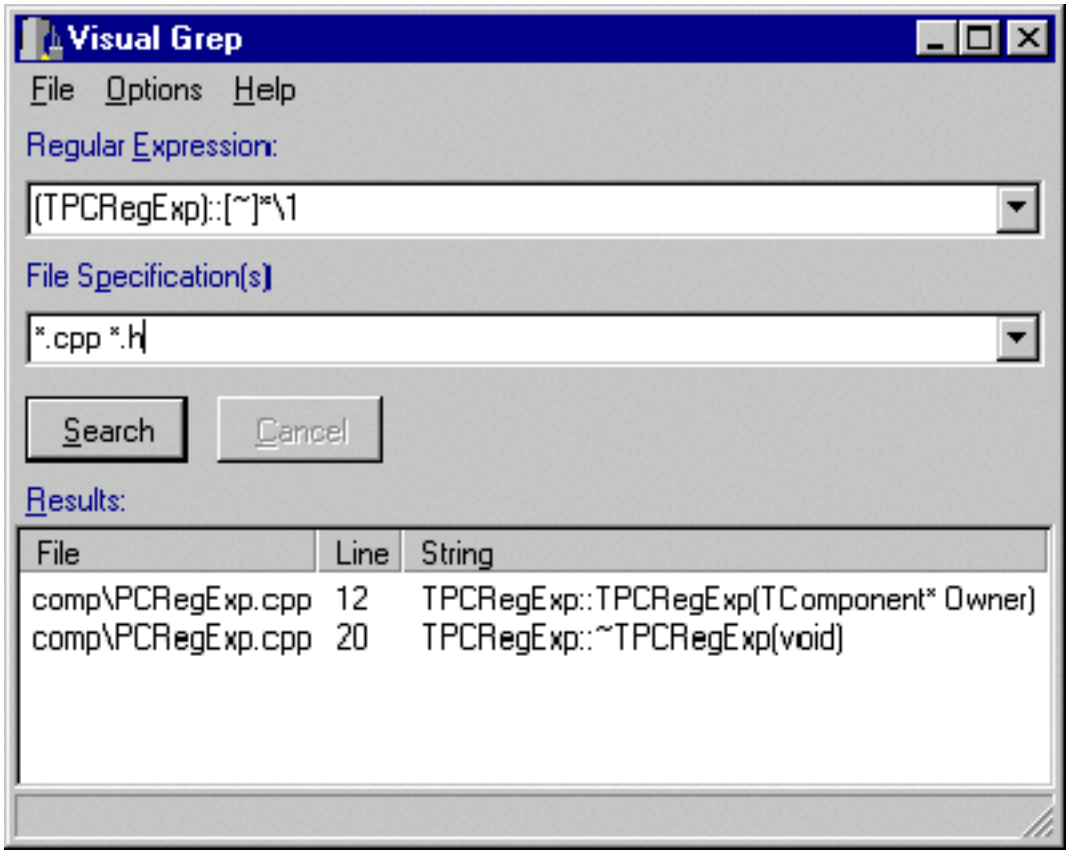
```
bool Execute(
  const char *subject, int len = 0);
```

As you can see, the `len` parameter has a default value of `0`. When `len` is `0`, the entire length of the target string is searched. The return value will indicate whether or not there were any successful matches. The entire match string can then be retrieved via the `EntireMatch` property. If sub-expressions are expected, then they will be converted into `AnsiStrings` and added into the `CaptureList` property, which is a `TStringList`.

The VGREP program

In order to show that my `TPCRegExp` component worked correctly, I wrote a small test program. Little by little I added things until it started getting to be more than just a test program. Eventually it seemed like this test program might make a useful utility, so I polished it up a bit and have included it here for you to use and/or modify. It's called VGREP.EXE and it is a visual GREP tool (see **Figure B**). In its present form it doesn't really have any advantages over the normal command line versions of GREP, but I do have some plans for it (possibly to be presented in future articles) that will make it much more useful.

Figure B



The Visual GREP program uses a regular expression pattern to find all constructors and destructors of the TPCRegExp class.

VGREP is pretty easy to use. Simply enter a regular expression and the list of file specifications (including wild cards) to search. Select any options from the Options Menu, such as case insensitivity or subdirectory search. Press the Search button and the results, if any, will appear in the “Results” ListView. The results are separated into the file name (and directory if necessary), line number, and the entire matching string.

Conclusion

In this article I introduced you to regular expressions and Perl. I also talked about the PCRE routines that come with C++Builder and how they are a bit cumbersome to use. Then I showed you my solution to this in the form of the `TPCRegExp` component. Finally, I explained the talked about the initial version of VGREP as an example of how to use the `TPCRegExp` component.

I have barely scratched the surface of what can be done with regular expressions. If you are interested in finding out more about Perl you can visit the major Perl web sites at **www.perl.com** and **www.perl.org**. Or, if you prefer printed material, I highly recommend the O’Reilly book “Mastering Regular Expressions,” by Jeffrey Friedl. Look for two owls on the front cover) .

The entire source code for VGREP and the `TPCRegExp` component can be downloaded from the

Bridges Publishing Web site at **www.bridgespublishing.com**.

ActiveForms, part III - MIDAS

by Bob Swart

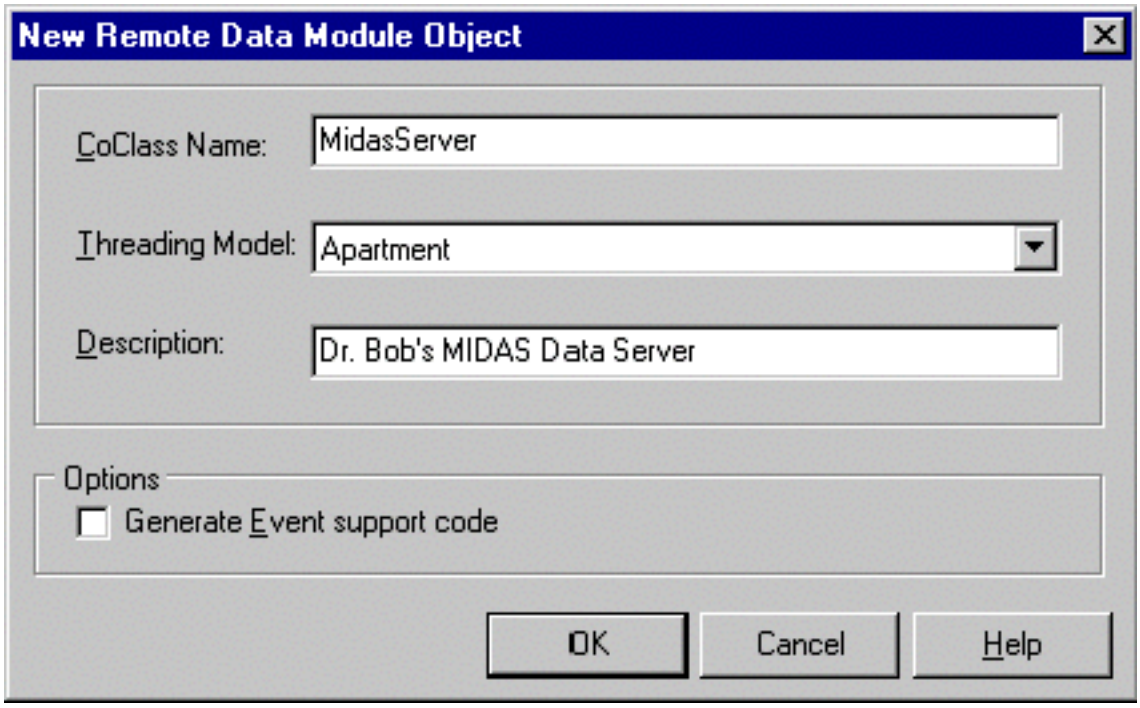
In the first two parts of this series you learned about ActiveForms and how to deploy your ActiveForms on a Web page. In the final part of this series we will show you how to build thin clients using Borland's MIDAS technology.

By using components from the MIDAS tab of the component palette, you can turn the "fat" ActiveForm created last month into a thin client. You can then connect it to a middleware remote data module. The first step is to create the middleware database server, followed by the process of turning the fat ActiveForm into a thin client.

Database server

To begin, create a new application. The middleware application can use the default form. The contents of the main form are not important, as the form will only be used to show that the server is currently running. Next, add a Remote Data Module to the project by selecting its icon from the Multitier tab of the Object Repository. The Remote Data Module is based on DCOM, which is what you'll be using to make the connection. When you create a new Remote Data Module you will see the dialog displayed in **Figure A**.

Figure A



The New Remote Data Module dialog is used to set the data module's properties.

The *CoClass Name* field is where you specify a name that you will use later when you connect the thin client to the database server. The option to include event support code is handy when you want to support events from the Remote Data Module sent to the thin client as notifications. We won't be using events in this example so you can leave this option unchecked.

The Remote Data Module is just like a regular Data Module in that you can only add non-visual components to it (Tables, Queries, DataSources, and so on). We will put the `TableCustomer` and `TableOrders` components on the Remote Data Module using the same components as we did in Part I of this series. In case you did not receive those issues, **Table A** shows the components needed and their properties.

Table A: The components needed to build the server and their properties.

Component	Property	Value
TTable	Name	TableCustomer
	DatabaseName	BCDEMOS
	TableName	CUSTOMER.DB
TDataSource	DataSet	TableCustomer
TTable	Name	TableOrders
	DatabaseName	BCDEMOS
	TableName	ORDERS.DB
	MasterSource	DataSource1
TDataSource	DataSet	TableOrders

You need to specify the master-detail relationship at this time. To define a master-detail relationship, select the `MasterFields` property in the Object Inspector and click on the ellipsis button to bring up the Field Link Designer. In the Field Link Designer, select the `CustNo` index from the *Available Indexes* combo box. Then select the `CustNo` field in both the *Detail Fields* and *Master Fields* list boxes. Now

you only need to click on the Add button to link the two tables in a master-detail relationship.

After you've set up the master-detail relationship, drop a `TProvider` component on the form from the Midas tab. Change the `Name` property to ProviderCustomer. You only need one `TProvider` because you will only export one table from the remote data module. If you export both the master and detail table from the remote data module, the clients could receive data that is out of sync (detail data that doesn't belong to the currently active master record at the client side, for example). Alternatively you could export both the master and the detail table, but the master-detail relationship at the client side means both tables are sent over the network. This is the case even if only a small portion of the detail table is needed. The best solution is to specify the master-detail relationship on the remote data module and export only the master table. This results in the detail records being included in a so-called nested table.

Set the Provider's `DataSet` property to TableCustomer. Right-click the Provider and select *Export ProviderCustomers from remote data module* from the context menu.

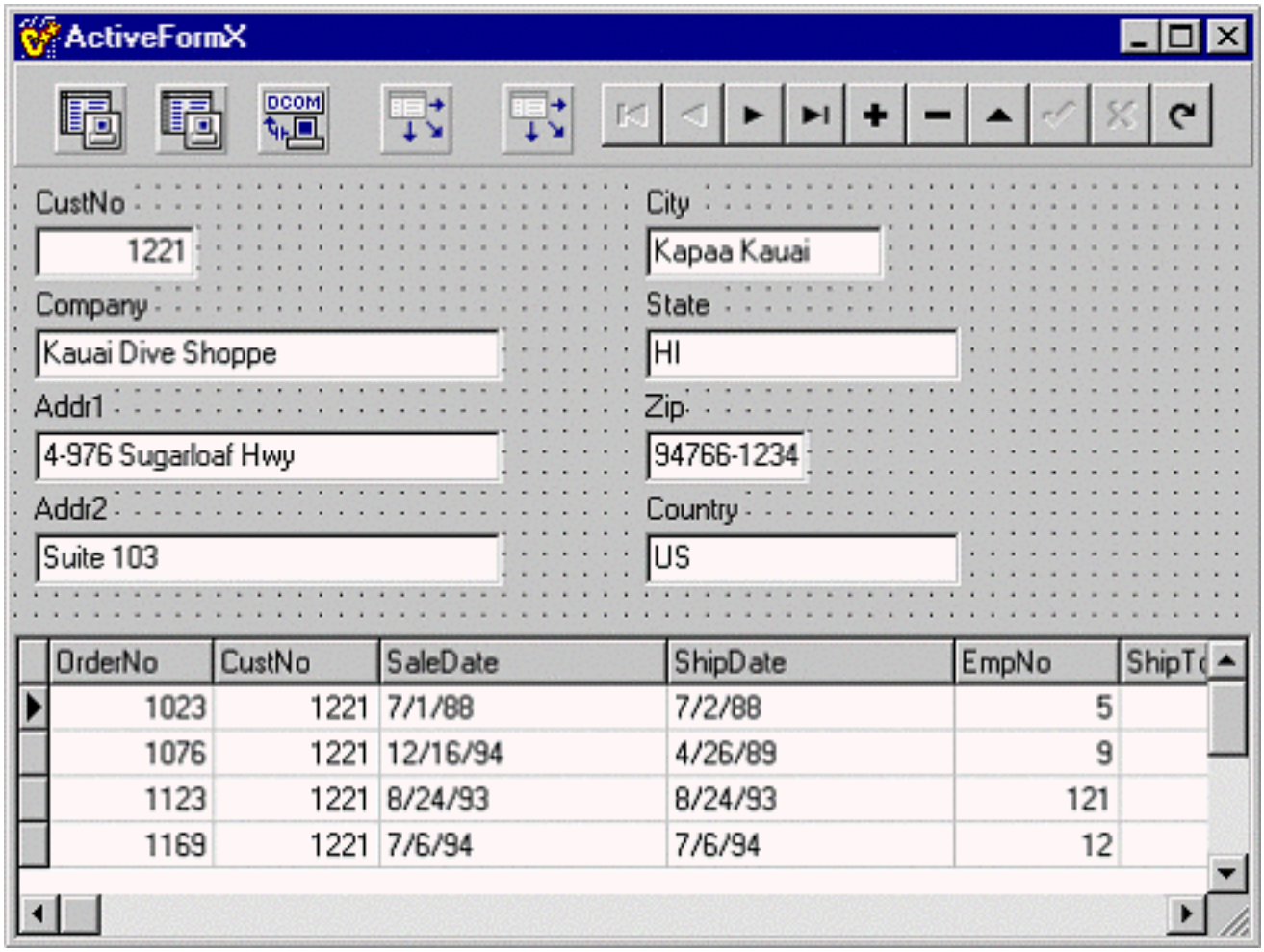
Your remote data module and middleware database server is now ready. You only need to compile and run it. Every time you run the middleware application, it's GUID and (new) location is registered. This means that if you move the server on your machine, you only need to run the server it to reregister it.

The DCOM connection

Now go back to the ActiveForm project from Part I of this series. Remove the two `TTable` components from the Form, and replace them with two `TClientDataSets` and one `TDCOMConnection` component. Select the DCOMConnection component and double-click next to the `ServerName` property in the Object Inspector. This will display a dialog showing all DCOM (remote data module) servers currently available on your machine. The server you just created should be shown in this list.

Set the `RemoteServer` property of both ClientDataSets to the DCOMConnection component. Select the first ClientDataSet component (the one associated with the Customers table) and open up the list for the available providers. This list is populated from the server. When you request the list, the server is automatically executed and the special default form will appear. You can now select the Customer provider, which connects to the remote data module. Invoke the Fields Editor and add all fields to the ClientDataSet. Note the last field, `TableOrders`. This is the nested dataset and contains the detail records for the master-detail relationship. Now select the second ClientDataSet, connect its DataSetField to the `ClientDataSet1->TableOrders` nested table. Reconnect the DataSource components to the ClientDataSets, and set the `Active` properties to `true` to see live data again as shown in **Figure B**.

Figure B



The modified ActiveForm obtains live data from the DCOMConnection at design time.

As a result of removing the `TTable` components and replacing them with `TClientDataSets`, the ActiveForm is now a thin client because the BDE is no longer needed. The database tables no longer need to be on the client machine. Instead, all clients now connect to a single middleware database server that is able to provide the data to multiple clients simultaneously. For bigger applications we would replace Paradox tables with an Oracle or SQL Server DBMS on the middle tier, and end up with a true N-tier application architecture.

Recommendations

Due to the security problems, package distribution/download issues and BDE deployment "fact of life", I can only recommend the use of ActiveForms in a controlled intranet environment. But in that particular case, they make an excellent choice for a thin client in an N-Tier distributed application.

Building a no-VCL Automation Server

by Alex Bakaev

Borland C++ Builder 4.0 provides excellent support for using and building COM/ActiveX components. The COM support in C++Builder 4.0 is a major improvement over version 3.0. Its ActiveX Wizard contains several options to create a COM-based component. Of course, C++Builder also gives you the ability to import ActiveX controls.

All this sounds good, and it is. But somewhere has to be a catch. After all, this ease of use cannot come for free. In this case, the price comes in the form of the size of the final DLL or OCX. The code generated by C++Builder drags in a lot of the VCL library—code that a lean COM object doesn't need.

Borland doesn't make a secret out of the fact that VCL-less COM was one of the design goals for C++Builder 4.0. For various reasons, though, this feature was dropped from the final product. But don't despair. A fair amount of this design goal has made it into the C++Builder 4.0 code base. In this article I will show how an Automation control can be tweaked to remove reliance on the VCL. To do this, you only need to make a small number of changes to the source files C++Builder generates when it creates a COM object.

A simple automation server

To begin, you will create a simple Automation Server. Select File|New from the main menu. Select the ActiveX tab in the Object Repository and double-click the ActiveX Library icon. C++Builder will create a project for you that contains two source files. I saved the project with the name VCLLESSCOM. Using that name results in the IDE generating files called VCLLESSCOM.CPP and VCLLESSCOM_ATL.CPP.

Next you need to give this object some functionality. This is accomplished by adding an automation object to the ActiveX library. Again select File|New from the main menu. This time double-click the Automation Object icon on the ActiveX page of the Object Repository. In the New Automation Object dialog, fill in the CoClass name. Predictably, I chose the name "VCLLess." I also opted to generate events support code.

When you close the New Automation Object dialog, C++Builder will add two more files to the project. The files are VCLLESSCOM.TLB (the type library) and VCLLESSIMPl.CPP (the implementation of the `IVCLLess` interface). The Type Library Editor is displayed, ready for you to add properties and methods to your automation object.

This article does not cover the subject of developing COM objects, so you will just define a simple

property. To do this, right click on the `IVCLLess` interface in the Type Library Editor, select New|Property and give it the name `SimpleProperty`. After you add the property, you should save the project. You will be prompted to select a name for the interface implementation file. You can save the implementation with the default name C++Builder provides.

Building the server

Before you build the server you need to set the project options. You don't want the COM server DLL to depend on any other DLLs or packages. Select Project|Options from the main menu and click on the Linker tab. Uncheck the *Use dynamic RTL* check box. You don't need an import library either so you can uncheck the *Generate import library* check box. Now select the Packages tab. Uncheck the *Build with runtime packages* option. Finally, go to the Directories/Conditionals page and set the intermediate directory to DEBUG. Close the Project Options dialog and save the project with the new project options.

Now build the project by selecting Project|Make VCLLessCom from the main menu. When the build is finished, start Windows Explorer and navigate to the directory where your newly born COM server resides. You should find that the DLL is a whopping 400K!

Dumping the VCL

400K for a COM object that does nothing is a bit extreme to say the least. Don't despair, though, because by the time we are done, over 300K of this bloat will be gone. What follows is a set of steps that you can use to remove the VCL from your COM objects.

Change project options

The first step requires adding a defined symbol to the project. Open the Project Options dialog and click on the Directories/Conditionals tab. Add the `_NO_VCL` define to the *Conditional defines* edit box. Close the Project Options dialog.

Modify the project source files

The project source files must be modified to eliminate dependence on the VCL. Switch to the main project file (VCLSLESSCOM.CPP). The first three lines look like this:

```
#include <vcl.h>
#pragma hdrstop
#include <atl\atlvcl.h>
```

Change these lines so that they read:

```
#include <windows.h>
#pragma hdrstop
#include <atl\atlmod.h>
#include <condefs.h>
```

The line that includes CONDEFS.H is required for C++Builder' s "USE" macros (`USERES`, `USEUNIT`, and so on). Normally CONDEFS.H is included by VCL.H. You are no longer including VCL.H so you must explicitly include CONDEFS.H.

In the VCLLESSCOM_ATL.CPP file, insert the following above the line that includes VCLLESSCOM_ATL.H:

```
#undef USING_ATLVCL
#define USING_ATL
```

The first few lines in VCLLESSCOM_ATL.CPP should now look like this:

```
#include <vcl.h>
#pragma hdrstop

#undef USING_ATLVCL
#define USING_ATL

#include "VCLLessCOM_ATL.h"
```

Now it's time to do some precision surgery. You will start with the project make file.

Change the project makefile

Select Project|View Makefile from the main menu. The VCLLESSCOM.BPR file will be displayed in the Code Editor. This is where most of the changes will take place. Do the following:

1. Remove `VCL40.LIB` from the `SPARELIBS` line.

2. Remove `sysinit.obj` from the `ALLOBJ` line.

3. Remove `$(LIBFILES)` from the `ALLLIB` line.

4. On the `ALLLIB` line, change `cp32mt.lib` to `cw32mt.lib`.

After making the above changes, save the project.

Modify ATLMOD.H

Next big change involves the `Unlock()` function of the `TATLModule` template in ATLMOD.H. You can open this file from VCLLESSCOM_ATL.CPP by right-clicking the mouse and selecting Open Source/Header file. You will be brought into the VCLLESSCOM_ATL.H file. While here, you should fix a rather nasty bug. Locate the line that reads:

```
extern CComModule _Module;
```

Change the line so that it reads:

```
extern CComModule &_Module;
```

The code generator "forgot" to add the reference operator to the declaration of `CComModule`.

Locate the line that includes ATLVCL.H. Click on the filename and press Ctrl-Enter on the keyboard, or right-click on the filename and select Open file at cursor. In the ATLVCL.H file, find the line that includes ATLMOD.H. Open ATLMOD.H and locate the `TATLModule<T>::Unlock()` method. Modify it so that it looks like this:

```
template <class T>
LONG TATLModule<T>::Unlock()
{
  LONG result = T::Unlock();
  if ((result == 0) && m_bExe)
  {
#ifndef _NO_VCL
    TSysCharSet DelimSet;
    DelimSet << '/' << '-';
    if (FindCmdLineSwitch(
        "AUTOMATION", DelimSet, true))
      ::PostThreadMessage(
        m_ThreadID, WM_QUIT, 0, 0);
#endif
  }
  return result;
}
```

What I've added is the `#ifndef _NO_VCL` and `#endif` lines. (The function body was formatted to fit the Journal's column width, so it will look slightly different here than in the original source.) The problem is that the code inside the `#ifndef/#endif` block is implemented in the VCL. I have no desire to rewrite the `FindCmdLineSwitch()` function in C. At any rate, this code is only needed for executables using the COM object. For those applications, the VCL is already used and the `_NO_VCL` macro will not be defined. Save ATLMOD.H after making the changes.

Modify ATLWIN.CPP

Next you will make a change to ATLWIN.CPP. The easiest way to find the line you will modify is to compile the project. When you compile, the compiler will produce an error message in ATLWIN.CPP on line 296. Here is that line:

```
for(i = 0; i < m_nEntries; i++)
```

The compiler error is generated because Microsoft does not adhere to the C++ standard regarding the scope of variables in `for` loops. In this case, a previous `for` loop declares the variable `i`. The second `for` loop attempts to reference that same variable. The C++Builder compiler correctly reports the second reference to `i` as being undefined because it is not in scope at that point. To fix this error, simply add `int` before the variable `i`:

```
for(int i = 0; i < m_nEntries; i++)
```

Save ATLWIN.CPP after making this change.

Change the property stub functions

The last thing to change is the stubs that the IDE generates for the property access functions in VCLLESSIMPL.CPP. The problem here is that the IDE generates code like this:

```
catch(Exception &e)
{
  return Error(
    e.Message.c_str(), IID_IVCLLess);
}
```

`Exception` is a VCL class and, obviously, we have removed reliance on the VCL. This is a problem in the `catch` statement itself, and in the `return` statement within the `catch` block. Change the `catch` blocks in the property access methods so that they look like this:

```
catch(...)
{
  return E_FAIL;
}
```

This code simply catches all exceptions and returns `E_FAIL` if an exception occurs.

Drum roll, please

At this point you are ready to build the DLL again and see the results of your work. Build the project again, making sure you do a full build and not just a make. You will see a few warning messages coming from the ATL files, but they are benign and nothing to worry about.

Now take a look at the file size of VCLLESSCOM.DLL. You will see that it is now only about 95K in size. Wow! That’s quite a savings!

Conclusion

Once you know what is required, it's rather easy to remove the VCL from a COM server. The best part is that all the features of RAD COM development are still at your disposal. The Type Library Editor will still work and the IDE will modify your source as you add properties and methods to the COM server. You may have noticed that there are still quite a few places where VCL.H is included. You can change those to include WINDOWS.H if you like. Look in the VCLLESSCOM_ATL.CPP, VCLLESSCOM_TLB.CPP and VCLLESSIMPL.CPP files for references to VCL.H. This change is not required to have a COM object stripped of the VCL, but will probably result in faster compile times. Also, by getting rid of VCL.H, you are guaranteed to hit compiler errors if by some chance you accidentally use a VCL function.

C++Builder is a complex and powerful piece of software. Though some corners had to be cut on the COM side of the product, enough hooks where put in place that a determined hacker can get rid of the fat that a default C++Builder COM object contains. Keep in mind that if you create an ActiveForm or create a new ActiveX control derived from a VCL component, you are still bound to the VCL. That's not necessarily a bad thing, because the VCL makes developing those kinds of objects a breeze. However, when you need a lean and mean COM server, the approach described in this article is the way to go.

Changing common dialog button captions

by Kent Reisdorph

A common question on the Borland C++Builder newsgroups is, "How do I change the caption of the file open dialog's Open button?" If, for example, you want to use the open dialog to allow your users to delete a file then you would probably want the Open button to show "Delete" rather than "Open." This is accomplished rather easily in the dialog's `OnShow` event handler. In your event handler you can simply do this:

```
HWND hWndDlg =
  GetParent(OpenDialog1->Handle);
HWND hWndBtn = GetDlgItem(hWndDlg, 1);
SetWindowText(hWndBtn, "&Delete");
```

I first obtain the dialog's window handle by calling `GetParent()` passing the `Handle` property as a parameter. It is important to understand that the dialog's `Handle` property is the window handle of a child dialog created by Windows, not the handle to the file open dialog itself. Calling `GetParent()` gives us a window handle to the file open dialog itself. Next I call `GetDlgItem()` to get the window handle of a control on the dialog. In this case I pass the handle to the dialog and a resource ID of 1, which is the resource ID of the dialog's Open button. (I used the resource editor from Borland C++ 5.02 to determine the control ID of the open button. I simply loaded COMDLG32.DLL in the resource editor and spied on the file open dialog resource.)

Changing the caption of a button on a common dialog is easy provided you know how the common dialogs are structured. The article, "Extending the common dialogs" explains how to extend the common dialogs to provide even more functionality.

Extending the common dialogs

by Kent Reisdorph

C++Builder provides components for Windows' common dialogs. These dialogs include `TOpenDialog`, `TSaveDialog`, `TFontDialog`, and so on. The actual dialogs for these components are not provided by the VCL, but rather are provided by Windows. The VCL common dialog components simply wrap the dialogs that already exist in Windows. This is probably common knowledge for most of you.

What may not be common knowledge, however, is that Windows provides a way of extending the common dialogs. By extending the common dialogs you can add additional controls to the common dialogs, thereby modifying their behavior to suit your needs.

In this article I will explain how to extend the Windows common dialogs. We'll be looking into areas virtually unknown to many C++Builder programmers. Specifically, I will show you how to create a dialog resource and how to apply that resource to the common dialogs. I will also show you how to respond to messages sent to controls you place on a common dialog. The techniques explained in this article will show how to extend Windows' file open dialog. The techniques presented, however, apply to all of the Windows common dialogs.

Dialog resources

When you extend the common dialogs, you provide a dialog resource that contains the extra controls you want shown on the common dialog. The concept of a dialog resource is foreign to many C++Builder programmers because all windows in a traditional C++Builder application (including dialogs) are implemented as forms. When I speak of dialog resources I am not talking about C++Builder forms. Instead, I am talking about true Windows dialogs, created from a dialog resource. Before going on, I need to spend a little time explaining standard Windows API programming and how it pertains to this article.

Windows programming the hard way

In the old days a dialog box was created entirely in text. The dialog's description was manually typed into a resource script file. A resource script file is simply a text file that, by tradition, has a filename extension of RC. Next, the resource script file was compiled into a binary resource file (RES file) using a resource compiler. The RES file was then linked to the application when the application was built. Here's how a simple dialog resource looks for a dialog that contains just an OK button and a static text control (a label).

```
IDD_DIALOG1 DIALOG 0, 0, 240, 120
STYLE DS_MODALFRAME | WS_POPUP |
  WS_VISIBLE | WS_CAPTION |  WS_SYSMENU
CAPTION "Test Dialog"
FONT 8, "MS Sans Serif"
{
  CONTROL "OK",1,"BUTTON",
    BS_PUSHBUTTON | BS_CENTER |
    WS_CHILD | WS_VISIBLE | WS_TABSTOP,
    92, 96, 50, 14
  CONTROL "This is a test.",100,"static",
```

```
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    88, 28, 56, 11
}
```

Notice that the header of the dialog resource describes the dialog's style, its size, font, and so on. The styles are `or`'d together just as in C++ code. The body of the dialog resource (the text within the curly braces) describes each control on the dialog. Here again, the control's caption, resource ID, class name, style, size, and position are described. A simple dialog resource is not terribly complex, but complexity increases as you add custom controls, ActiveX controls, etc.

To display a dialog contained as a resource, you must call a Windows API function such as `DialogBox()` (for modal dialogs) or `CreateDialog()` (for modeless dialogs). The programmer must provide a dialog procedure (`DialogProc()`) to handle messages sent to the dialog. Creating dialogs in this manner requires a great deal of time.

It wasn't long before the major compiler vendors were providing resource editors for their compilers. A resource editor allows the programmer to design dialog boxes visually, much like the C++Builder form designer. While the use of a resource editor greatly reduced the time required to design dialogs, it is still a tedious process compared to creating forms in C++Builder.

Dialog resources and common dialogs

In order to add controls to a common dialog, you must provide a dialog resource that contains additional controls that you want placed on the common dialog. The dialog resource you create should contain only the additional controls. Windows will merge the controls in your dialog resource with those on the common dialog. More accurately, Windows does the following:

- Expands the common dialog by the height and width of your dialog resource
- Creates your dialog resource as a child of the common dialog
- Executes the common dialog

At runtime, the new controls are placed below the existing controls on the common dialog.

In order for your dialog resource to merge with the common dialog it should have these styles:

`WS_CHILD`Required because the new dialog is a child of the common dialog.

`WS_CLIPSIBLINGS`Insures that the new controls don't overlay any of the existing controls.

`DS_3DLOOK`Insures that the new controls have a 3D look.

`DS_CONTROL`Allows the user to use the Tab key to navigate all controls on the dialog.

It's important that the dialog does not have a border (the `WS_BORDER` style is not present). If the dialog has a border, the border will show on the common dialog when it is displayed. As long as the dialog does not have a border, it is seamlessly merged with the common dialog and the user sees what appears to be a single dialog.

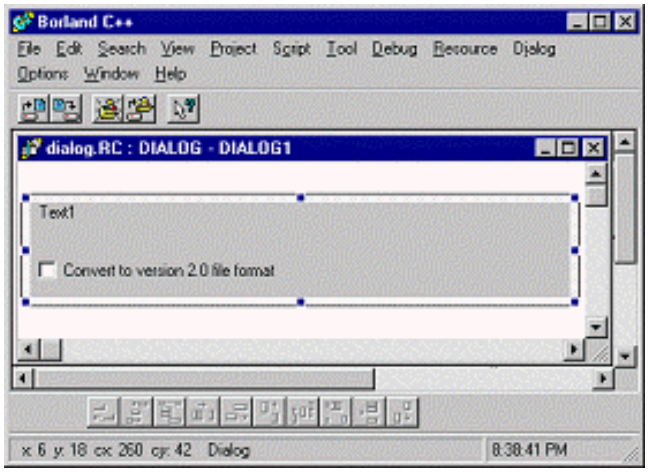
Creating the dialog template

Without question the easiest way to create the dialog resource template is with a dialog editor. Unfortunately C++Builder does not include a dialog editor. However, C++Builder ships with Borland C++ 5.02 so you can use the resource editor built into that product to create your dialog resources. C++Builder 4 also ships with Resource Workshop. (Resource Workshop is not installed by default so you will have to install it manually. Look in the WORKSHOP directory on the C++Builder CD.) If you use Resource Workshop to create your dialogs you may have to add the `DS_3DLOOK` style manually (the value of `DS_3DLOOK` is `0x0004`) as Resource Workshop doesn't include that style.

The dialog resource for this article's example program has a resource name of `DIALOG1`. It contains two controls that will be merged with the standard open dialog's controls. The first control is a static text control (a label). This control will be used to display explanatory text on the open dialog. The resource identifier for this control is 102 (an arbitrary number selected by the resource editor). The second control is a check box. This check box will be labeled, "Convert to version 2.0 file format." This assumes a fictitious program that is version 2.0. When this check box is checked the user program can update the file format from version 1.0 to the new 2.0 format. This control has a resource ID of 101.

The standard Windows file open dialog is 280 pixels wide (assuming the default Windows system font). I have made the dialog template 260 pixels wide to insure that it fits on the common dialog without expanding its size. If the dialog is wider than 260 pixels, the common dialog will be expanded to make room for the new dialog's width. **Figure A** shows the dialog resource in a BC5 resource editor window.

Figure A



The dialog as it appears in the Borland C++ 5.02 resource editor.

Listing A shows the resource script file (DIALOGS.RC) produced by the resource editor.

Once you have created the resource script file, you can simply add it to your project. C++Builder will compile the resource file and will produce an output file called DIALOGS.RES. The output file is then bound to the executable during the link phase of the build.

VCL and common dialog templates

A bit of work is needed to implement a custom common dialog in VCL. Here are the required tasks (assuming the dialog resource created earlier):

- Create a new class derived from `TOpenDialog`. This new class will have properties associated with the two new controls on the common dialog and a constructor that allows you to specify the dialog resource name.

- In the constructor, assign the resource name to `TOpenDialog's Template` property.

- Provide an overridden `WndProc()` for the dialog in order to respond to messages pertinent to the additional controls.

The following sections detail each of these steps.

Create a new dialog class

The first thing you need to do is create a new class derived from the common dialog you wish to extend. In this case we'll be extending the file open dialog so the new class will be derived from `TOpenDialog`. It is necessary to derive a class from `TOpenDialog` because you need access to a property called `Template`. This property is declared in the `protected` section of `TOpenDialog`, so you must derive a new class in order to gain access to it. `Template` is a `char*` property used to specify the name of the dialog resource template that contains the controls that will be added to the dialog. All of the VCL classes representing the common dialogs have a `Template` property.

Here's the declaration for the class, which I have named `TMyOpenDialog`:

```
class TMyOpenDialog : public TOpenDialog
{
  private:
    String FInstructions;
    bool FUpdate;
  protected:
    virtual void __fastcall WndProc(
      Messages::TMessage &Message);
  public:
    TMyOpenDialog(TComponent* AOwner,
      char* TemplateName);
    __property String Instructions = {
      read = FInstructions,
      write = FInstructions};
    __property bool Update = {
      read = FUpdate,
      write = FUpdate};
};
```

As you can see, this class only has two methods. The base class's `WndProc()` method is overridden so that you can respond to message sent to the controls that you place on the file open dialog (I'll discuss the `WndProc()` in a later section). The constructor takes the usual `AOwner` parameter as well as a new parameter called `TemplateName`. You will pass the name of the dialog template for this parameter (remember that the dialog's resource name is `DIALOG1`).

`TMyOpenDialog` has two properties. The `Instructions` property will be used to specify the instructions to display on the open dialog. The code will set the text of the static text control in your dialog resource to the value of the `Instructions` property. The `Update` property is associated with the check box control contained in the dialog

resource. You can examine the value of this property after the open dialog closes to see if the user checked the check box.

Writing the constructor

The `TMyOpenDialog` constructor sets the `Template` property to the value passed in the `TemplateName` parameter. It's as simple as this:

```
TMyOpenDialog::TMyOpenDialog(
  TComponent* AOwner, char* TemplateName)
  : TOpenDialog(AOwner)
{
  if (strlen(TemplateName) != 0)
    Template = TemplateName;
  else
    Template = NULL;
}
```

Notice that I assign the value of the `TemplateName` parameter to the `Template` property only if a valid string was passed in. If an empty string was passed in, I set `Template` to `NULL`. This allows the dialog class to be used either as a custom file open dialog (if a resource name was passed in) or as a standard file open dialog (if an empty string was passed in).

When the `Execute()` method is called, VCL passes the template name on to Windows and sets up the proper flags so that Windows knows that a custom file open dialog is being used.

Dealing with the WndProc()

When you add controls to the common dialogs you must deal with those controls the old fashioned way—at the API level. Dealing with controls on this level requires knowledge of Windows messages and the Windows API.

When the file open dialog is invoked, Windows sends messages to the dialog at various times during the creation process. Windows also sends messages to the controls on the dialog, both at creation time and while the dialog is visible. You must intercept some of those messages in order to interact with the controls you have placed on the dialog. Take the static text control on the dialog for example. You need to set the text of this control prior to the dialog being shown. To do that you must intercept the `WM_INITDIALOG` message. When your `WndProc()` receives this message, you can set the control's text using the `SetWindowText()` API function. Here's how that code might look:

```
TMyOpenDialog::WndProc(TMessage& Message)
{
  if (Message.Msg == WM_INITDIALOG)
    SetWindowText(GetDlgItem(Handle,102),
      FInstructions.c_str());
  TOpenDialog::WndProc(Message);
}
```

The pertinent piece of code is the line following the `if` statement. The `GetDlgItem()` function returns the window handle of a control on a dialog. I pass the window handle of the parent dialog (the `Handle` property in this case) and the resource identifier of the control whose handle I am requesting. If you recall, the resource ID for the static text control on

the dialog resource is 102. Once I have the static text control's window handle I can call the API function `SetWindowText()` to set the label's text. I could have sent the control a `WM_SETTEXT` message, but this method is easier both to implement and to understand. Notice that I am passing the value of the `FInstruction` field to `SetWindowText()` using the `c_str()` function. The static text control's text is now set to the value of the `Instructions` property.

The check box's text is already set so I don't need to worry about setting the control's text as I did for the static control. What I must do, however, is determine whether or not the check box is checked so I can update the value of the `Update` property. I respond to the `WM_COMMAND` message and examine the state of the check box at that time. First look at the full code for the `WndProc()` function:

```
void __fastcall
TMyOpenDialog::WndProc(TMessage& Message)
{
  switch (Message.Msg) {
    case WM_INITDIALOG : {
      SetWindowText(
        GetDlgItem(Handle,102),
        FInstructions.c_str());
      break;
    }
    case WM_COMMAND : {
      if (LOWORD(Message.WParam)==101 &&
         (HIWORD(Message.WParam) ==
           BN_CLICKED)) {
        FUpdate =
          IsDlgButtonChecked(Handle,101);
        Message.Result = 1;
      }
      break;
    }
  }
  TOpenDialog::WndProc(Message);
}
```

Notice that I have added a `switch` statement to handle the different Windows messages that I am interested in. In this particular case I don't necessarily need a `switch` since I am only responding to two messages. The `switch` statement, however, allows me to easily add additional message handlers should the need arise.

Now turn your attention to the code for the `WM_COMMAND` message. This message is sent to a window procedure when some action takes place for a control. In the case of buttons (a check box is a form of button) the `WM_COMMAND` message will be sent when the button is clicked. The low order word of the `WPARAM` contains the resource ID of the control that generated the message and the high order word contains the message that was generated. Here I am checking the to see if the low order word contains the value 101 (the resource ID of the check box) and if the high order word contains the `BN_CLICKED` notification message. If both of those conditions are met, I know the check box is checked. The next line calls `IsDlgButtonChecked()` and assigns the return value to the `FUpdate` field. When the open dialog closes, the `Update` property will contain the check box's checked state.

Writing a `WndProc()` is not terribly complicated if your open dialog contains simple controls. If your open dialog contains complex controls (such as a tree view or a list view) then additional code is required to interact with those

controls. Certainly some knowledge of the Windows API is required in order to implement a `WndProc()` for your custom common dialogs. Don't forget to call the base class `WndProc()` so that messages to the dialog flow normally.

Putting the dialog to work

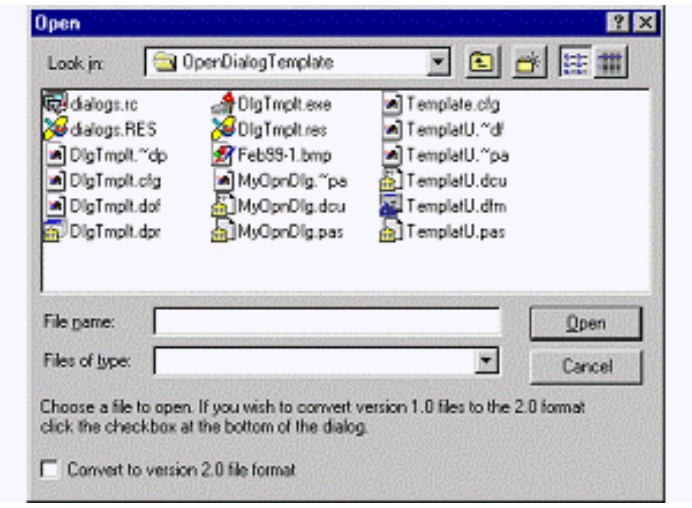
At this point the hard work is done. Putting your customized file open dialog to work is relatively simple:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  TMyOpenDialog* Dialog =
    new TMyOpenDialog(this, "DIALOG1");
  Dialog->Instructions="This is a test."
  if (Dialog->Execute()) {
    // Do something with FileName
    if (Dialog->Update)
      ShowMessage(
        "Updated to version 2.0.");
  }
  delete Dialog;
}
```

Here I simply create an instance of the `TMyOpenDialog` class, assign a text string to the `Instruction` property, and call the `Execute()` method (provided by the base class). When `Execute()` returns, I check the value of the `Update` property to determine whether or not the check box was checked.

Figure B shows the customized dialog at runtime. Note the instruction text and check box at the bottom of the dialog.

Figure B



The customized file open dialog at runtime.

Positioning controls

As I have said, new controls are automatically added below the common dialog's existing controls. Windows does, however, provide a way of specifying where new controls are placed on the common dialog.

Windows defines a constant called `stc32`. This constant has a value of `0x045f`. When a control with this ID exists on the dialog, any other controls on the dialog will be placed on the common dialog relative to that control. The second dialog resource in **Listing A** (the dialog resource name `DIALOG2`) shows a dialog resource that uses this control.

Note that I have used the value `0x045f` for this control's resource ID. I use the actual value `0x045f` because the resource compiler has no knowledge of the `stc32` constant. In your dialog editor, make this control hidden so that it is not displayed on the dialog at runtime.

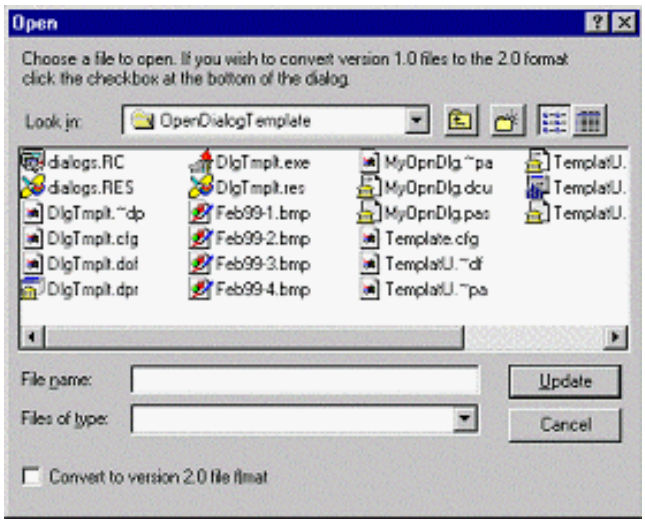
You may be wondering how Windows uses this special control. Any of the controls that appear on your custom dialog resource above this special control will appear *above* the existing controls on the common dialog. Any controls that appear below this control will appear *below* the common dialog's existing controls. Similarly, any controls placed to the right or left of the `stc32` control will be placed on the common dialog to the right or left of the existing controls.

Figure C shows the design-time image of a dialog resource that uses this control. Note that although the `stc32` control shows up at design time it is not visible at runtime. **Figure D** shows the file open dialog as it appears at runtime when applying the custom controls.

Figure C

This dialog resource uses the special stc32control.

Figure D



The new file open dialog at run time.

The `stc32` control gives you additional flexibility when customizing Windows' common dialogs. Some experimentation

will likely be necessary to get controls to appear just as you want them.

Conclusion

The listings at the end of this article make up a complete program that illustrates the concepts discussed in this article. **Listing A** shows the resource script file containing two custom dialog resources. **Listing B** is the declaration of the `TMyCustomDialog` class. **Listing C** contains the example program's main unit. The top portion of the listing shows the implementation of the `TMyCustomDialog` class's constructor and `WndProc()` functions. The main form has three buttons that display the file open dialog, customized in three different ways. You can download the example program from the Bridges Publishing Web site at **www.bridgespublishing.com**.

Customizing the Windows common dialogs is something that, from the outside, appears difficult to accomplish. While not trivial, customizing the common dialogs is not so daunting once you know the basic requirements. While dealing with dialog resources is not something that most C++Builder programmers have experience with, it is not difficult once you have done it a time or two. A good resource editor is the key to editing dialog resources.

Listing A: DIALOGS.RC

```
DIALOG1 DIALOG 0, 0, 260, 42
STYLE DS_3DLOOK | DS_CONTROL | DS_CONTEXTHELP |  WS_CHILD | WS_VISIBLE |
WS_CLIPSIBLINGS | WS_SYSMENU
CAPTION ""
FONT 8, "MS Sans Serif"
begin
 CONTROL "Convert to version 2.0 file format", 101, "button", BS_AUTOCHECKBOX |
WS_CHILD | WS_VISIBLE | WS_TABSTOP, 4, 24, 120, 12
 CONTROL "Text1", 102, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 0, 248, 20
end

DIALOG2 DIALOG 0, 0, 260, 58
STYLE DS_3DLOOK | DS_CONTROL | DS_CONTEXTHELP | WS_CHILD | WS_VISIBLE |
WS_CLIPSIBLINGS | WS_SYSMENU
CAPTION ""
FONT 8, "MS Sans Serif"
begin
 CONTROL "Convert to version 2.0 file format", 101, "button", BS_AUTOCHECKBOX |
WS_CHILD | WS_VISIBLE | WS_TABSTOP, 6, 36, 120, 12
 CONTROL "Text1", 102, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 6, 4, 254, 16
 CONTROL "This is the placeholder control", 0x045f, "static", SS_LEFT | WS_CHILD |
NOT WS_VISIBLE, 0, 24, 260, 9
end
```

Listing B: The TMyOpenDialog declaration.

```
class TMyOpenDialog : public TOpenDialog {
  private:
    String FInstructions;
    bool FUpdate;
  protected:
```

```
    virtual void __fastcall
      WndProc(Messages::TMessage &Message);
  public:
    TMyOpenDialog(TComponent* AOwner,
      char* TemplateName);
    __property String Instructions = {
      read = FInstructions,
      write = FInstructions};
    __property bool Update = {
      read = FUpdate,
      write = FUpdate};
};
```

Listing C: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

TMyOpenDialog::TMyOpenDialog(TComponent* AOwner,
  char* TemplateName) : TOpenDialog(AOwner)
{
  // Pass on the dialog resource if one was
  // supplied. If not, set Template to nil
  // so the standard File Open dialog is shown.
  if (strlen(TemplateName) != 0)
    Template = TemplateName;
  else
    Template = NULL;
}

// The window procedure. We need to provide message
// handling for the extra controls on the form.
void __fastcall
TMyOpenDialog::WndProc(TMessage& Message)
{
  switch (Message.Msg) {
    case WM_INITDIALOG : {
      // Set the static label's text based on the
      // value of the Instructions property. The
      // resource ID for the static label is 102.
      SetWindowText(GetDlgItem
        (Handle, 102), FInstructions.c_str());
      break;
    }
    case WM_COMMAND : {
```

```
      // The check box was checked. Set the
      // Update property according to
      // the check box's checked state.
      if (LOWORD(Message.WParam) == 101 &&
         HIWORD(Message.WParam) == BN_CLICKED) {
        FUpdate =
          IsDlgButtonChecked(Handle, 101);
        Message.Result = 1;
      }
      break;
    }
  }
  TOpenDialog::WndProc(Message);
}

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  // Create an instance of the class using the
  // DIALOG1 dialog template. Assign text to
  // the Instructions property and Execute the
  // dialog.
  TMyOpenDialog* Dialog =
    new TMyOpenDialog(this, "DIALOG1");
  Dialog->Instructions = "Choose a file to "
    "open. If you wish to convert version "
    "1.0 files to the 2.0 format click the "
    "check box at the bottom of the dialog.";
  if (Dialog->Execute()) {
    Label1->Caption = Dialog->FileName;
    if (Dialog->Update) {
      Label2->Caption = "Checked";
      ShowMessage(
        "File updated to version 2.0 format.");
    }
    else
      Label2->Caption = "Not Checked";
  }
  delete Dialog;
}

void __fastcall
TForm1::Button2Click(TObject *Sender)
{
  // Create an instance of the class using the
  // DIALOG2 dialog template.
```

```
  TMyOpenDialog* Dialog =
    new TMyOpenDialog(this, "DIALOG2");
  Dialog->Instructions = "Choose a file to "
    "open. If you wish to convert version "
    "1.0 files to the 2.0 format click the "
    "check box at the bottom of the dialog.";
  if (Dialog->Execute()) {
    Label1->Caption = Dialog->FileName;
    if (Dialog->Update) {
      Label2->Caption = "Checked";
      ShowMessage(
        "File updated to version 2.0 format.");
    }
    else
      Label2->Caption = "Not Checked";
  }
  delete Dialog;
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{
  OpenDialog1->Execute();
}

void __fastcall
TForm1::OpenDialog1Show(TObject *Sender)
{
  HWND hWndDlg =
    GetParent(OpenDialog1->Handle);
  HWND hWndBtn = GetDlgItem(hWndDlg, 1);
  SetWindowText(hWndBtn, "&Delete");

}
```

Programming the recycle bin

By Mark G. Wiseman

One nice feature in the Windows shell is the Recycle Bin. You can manipulate the Recycle Bin in your code by using functions contained in SHELL32.DLL. You can send deleted files to the Recycle Bin and, if you have the right version of SHELL32.DLL, you can empty the Recycle Bin as well.

Listing A shows the header for a unit that contains the functions presented in this article. **Listing B** shows the source code for three functions that you can use to manipulate the Recycle Bin.

Recycling files

The shell API contains a very powerful function called `SHFileOperation()`. Using this function, you can copy, move, rename, or delete a list of files and directories. `SHFileOperation()` can also ask for user confirmation and display animations for operations that take a long time. `SHFileOperation()` is supported by all versions of SHELL32.DLL.

I have wrapped `SHFileOperation()` with a function called `DeleteFiles()`. This function takes a `String` argument, `filespec`, which contains the name of a file. The `filespec` parameter can include wildcard characters to delete a group of files. `DeleteFiles()` also takes a `bool` argument, `allowUndo`. If `allowUndo` is `true`, the files are removed from their original location and sent to the recycle bin. If `allowUndo` is `false` (the default) the files are permanently deleted.

Emptying the recycle bin

Versions of SHELL32.DLL greater than or equal to 4.71 contain two functions called `SHQueryRecycleBin()` and `SHEmptyRecycleBin()`.(In the last issue, you learned how to determine the version of SHELL32.DLL using the `GetDLLVersion()` function.) You can use these two functions to empty the Recycle Bin in your programs.

`SHQueryRecycleBin()` returns information about the Recycle Bin, including how many files are in it. The `CanEmptyRecycleBin()` function is a simple wrapper for `SHQueryRecycleBin()`. If the Recycle Bin is empty or if `SHQueryRecycleBin()` is not available, `CanEmptyRecycleBin()` returns `false`, otherwise it returns `true`.

The `EmptyRecycleBin()` function wraps `SHEmptyRecycleBin()`. It takes a `DWORD` argument, `flags`. The values in **Table A** can be used with the `flags` parameter. The default is `SHERB_SILENT`.

Table A: Flags for the EmptyRecycleBin() function.

SHERB_NOCONFIRMATION	Do not display a confirmation dialog before emptying the Recycle Bin.
SHERB_NOPROGRESSUI	Do not display progress while emptying the Recycle Bin.
SHERB_NOSOUND	Do not play a sound when the Recycle Bin has been emptied.
SHERB_SILENT	A combination of all the flags above.

`EmptyRecycleBin()` returns `true` if `SHEmptyRecycleBin()` is available and the Recycle Bin is successfully emptied. Otherwise, `EmptyRecycleBin()` returns `false`.

The example program for this article creates three dummy files and then uses `DeleteFiles()` to delete the files to the Recycle Bin. The program uses `CanEmptyRecycleBin()` to enable or disable a button that, when enabled, uses `EmptyRecycleBin()` to empty the Recycle Bin. You can download the example from the Bridges Publishing Web site.

Listing A: *RecBin.h*

```
#ifndef RecycleH
#define RecycleH

#include <shellapi.h>

bool DeleteFiles(
  String filespec, bool allowUndo = false);

const DWORD SHERB_SILENT = SHERB_NOCONFIRMATION |
  SHERB_NOPROGRESSUI | SHERB_NOSOUND;

bool EmptyRecycleBin(DWORD flags = SHERB_SILENT);

bool CanEmptyRecycleBin();

#endif    // RecycleH
```

Listing B: *RecBin.cpp*

```
#include <vcl\vcl.h>
#pragma hdrstop

#include <shellapi.h>

#include "DLLVersion.h"
#include "RecBin.h"

bool DeleteFiles(String filespec, bool allowUndo)
{
  SHFILEOPSTRUCT shop;
  ZeroMemory(&shop, sizeof(shop));
  shop.wFunc = FO_DELETE;
  shop.pFrom = filespec.c_str();
  shop.fFlags =
    FOF_FILESONLY | FOF_NOCONFIRMATION | FOF_SILENT;
  if (allowUndo) shop.fFlags |= FOF_ALLOWUNDO;

  return (SHFileOperation(&shop) == 0);
}

bool EmptyRecycleBin(DWORD flags)
{
  if (GetDLLVersion("shell32.dll") <
      PackDLLVersion(4, 71))
    return(false);

  return (SHEmptyRecycleBin(
    Application->MainForm->Handle,
    0, flags) == S_OK);
}

bool CanEmptyRecycleBin()
{
  if (GetDLLVersion("shell32.dll") <
      PackDLLVersion(4, 71))
    return(false);

  SHQUERYRBINFO info;
  ZeroMemory(&info, sizeof(info));
  info.cbSize = sizeof(info);
  SHQueryRecycleBin(0, &info);
```

```
  return (info.i64NumItems > 0);
}

#pragma package(smart_init)
```

ActiveForms, part II - deployment

by Bob Swart

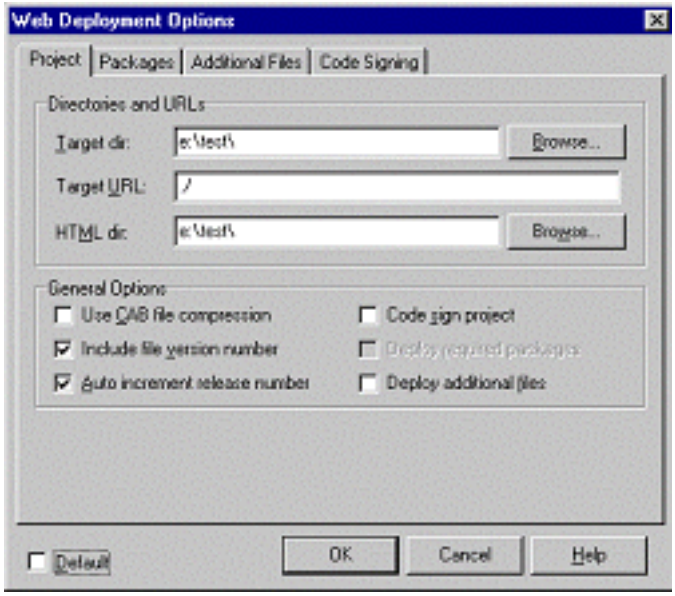
Last month we introduced you to ActiveForms. In this article we will discuss the steps necessary to deploy ActiveForms on a Web page.

Deployment basics

Once you have built an ActiveForm, you may want to deploy it for use on a Web page. To begin, choose Project|Web Deployment Options from the C++Builder main menu to display the Web Deployment Options dialog. This dialog allows you to control the way in which your ActiveForm is deployed.

You must specify both the target directory for the ActiveForm and the HTML directory for the accompanying HTML file. I always put the ActiveForm in the same directory as the HTML file so the project is easy to maintain. Given that, the first and third edit boxes both contain the same value. For local testing you can enter a directory on your hard drive, such as `c:\test`. The *Target URL* field must contain the relative or absolute URL used by the HTML file to refer to the ActiveX control. You can enter either a full URL here such as `http://www.drbob42.com/ActiveX/` or a relative path such as `./`. (Using a relative path is convenient for local testing.) **Figure A** shows the Web Deployment Options dialog for the ActiveForm test project.

Figure A



The Web Deployment Options dialog allows you to set the output directories and target URL for your ActiveForm.

Remember last month when we included version information for the ActiveForm? Make sure the *Include file version number* and *Auto increment release number* options are checked. This ensures that the release number part of the version information is increased every time a new version of the ActiveForm is deployed.

Keep in mind that we built the ActiveForm using neither runtime packages nor the dynamic RTL. As a result, we don't have to deploy any additional files. However, the stand-alone ActiveForm will be almost 1 MB in size. Of course, when specifying runtime packages and/or the use of the dynamic RTL, the ActiveForm will be much smaller. However, you will need to deploy the required packages, BC3245MT.DLL, and BORLNDMM.DLL (assuming C++Builder 4.0).

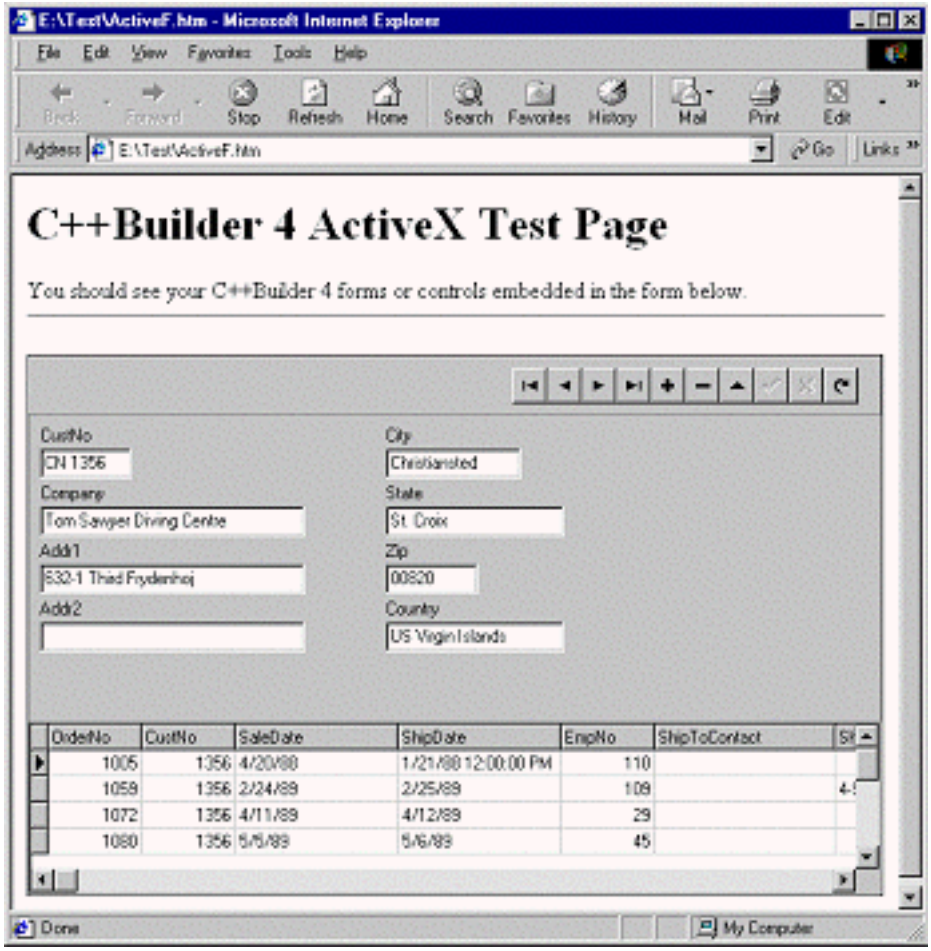
After closing the Web Deployment Options dialog, select Project|Web Deploy to deploy your ActiveForm. The Deploying Project dialog box will appear while the ActiveForm is deployed. If you are writing to a local file, the dialog might only be displayed for a second or so. After the dialog disappears, the ActiveForm and HTML files will have been written to the specified directories.

Whether or not you are using packages, you can always shrink down the file size of the ActiveForm by selecting the *Use CAB file compression* option. CAB compression was developed by Microsoft and can often compress files to 50% (or less) of their original size. The deployment process itself will take somewhat longer, but the result will be a faster download of the ActiveForm.

Microsoft Internet Explorer

When using ActiveForms on web pages, you must have Internet Explorer version 3 or higher. (Netscape browsers require a third-party plug-in by NCompass, and require some manual conversions of the generated HTML file.) To view the ActiveForm on a Web page, start Internet Explorer and enter the local web page where you deployed your ActiveForm (`file:///C:/test/ActiveF.htm`, for example). If Internet Explorer is your default browser you can just navigate to your deployment directory and double-click on ACTIVEF.HTM. **Figure B** shows the ActiveForm hosted in a Web page.

Figure B



Once deployed, the ActiveForm can be viewed on a Web page in Internet Explorer.

Last month, when we built the ActiveForm, we set the `Active` property of both tables to `true`. As a result, the ActiveForm shows live data as soon as it is loaded in the browser.

Additional considerations

There are still a few bits and pieces regarding ActiveForm deployment that you should be aware of. ActiveX controls are not confined to the web browser, but are, in effect, regular Win32 applications running on the user's machine. As such, they have all the power and abilities of a regular application including network permissions, access rights, and so on. In this regard ActiveX controls are like a Trojan horse, invited by you to run on your machine.

The only sure way to solve this potential security risk is to instruct Internet Explorer to never show any ActiveX content. However, that will render our ActiveForm useless. Fortunately, there's a compromise, called code signing. Code signing options are specified on the Code Signing tab of the Web Deployment Options dialog. Any ActiveX control can be code signed by the author, resulting in a control that can be verified for authenticity by any user. The author of the control obtains a special credentials file and key that he can use to add his digital signature to the ActiveX control. While this doesn't guarantee a well-behaved ActiveX control, at least you can be sure you "know" the author. Combined with the ability to

tell Internet Explorer to show only ActiveX controls that are code signed, you can ignore all non-signed (and potential dangerous) ActiveX controls.

Fat client

Apart from security issues, there's another problem that makes ActiveForms rather unsuitable for an open Internet environment (as opposed to a closed intranet environment), and that is code size. As I said earlier, an ActiveForm built without the dynamic RTL and runtime packages will be almost 1 MB in size. If you were to build the ActiveForm using the dynamic RTL and runtime packages you would see that it's only a small OCX file. However, when the ActiveForm is built this way, you must deploy DLLs and packages totaling nearly 3.5 MB in size.

In the end, it's a matter of initial size (and many small updates) vs. total size (and almost no updates).

Borland Database Engine

A final problem is the fact that the ActiveForm—with or without packages—is a so-called "fat client." This is partially because the ActiveForm uses the Borland Database Engine (BDE). As a result, the BDE must be installed on the client machine. BDE installation is not something that you can expect the average Internet user to handle. Worse, the BCDEMOS alias that the ActiveForm uses (including the CUSTOMER.DB and ORDERS.DB tables) must also be available on the local client machine. Of course, this is just a sample ActiveForm, but generally speaking an ActiveForm client set up like this, will require the BDE on each client machine. Plus, the only way to make sure every client is sharing tables (in the case of multi-user access) is by making sure that the database alias actually points to a shared resource such as a database server on a local network. Again, this is not something you might trust to the average Internet user, although it’s actually quite common in an intranet environment.

At my company, we have a local network where the network fileserver is also the database server. In a case like this you could place the CUSTOMER.DB and ORDERS.DB files on the shared server. Then every ActiveForm client could use and share those same tables. In my particular case, I can pretty much guarantee not only that every machine runs Internet Explorer 3 or higher, but also that every client machine has the BDE installed.

Clearly an ActiveForm that does not require the BDE would be easier to deploy. Next month, in Part III of this series, you will see how to turn the fat client ActiveForm into a thin client using Borland’s MIDAS technology.

Dynamic C++ arrays using STL

by John Miano

C++'s treatment of arrays is awkward in many ways. Most of the problems can be traced back to C's quirky interchangeability of arrays and pointers. This article will explain how to implement dynamic arrays using the Standard Template Library (STL) `vector` and `auto_ptr` classes.

Defining the problem

Unlike some programming languages, C++ does not support dynamic arrays. In many applications there is a need for arrays whose sizes are set by function parameters. Unfortunately, a declaration such as this one is illegal because array bounds must be a constant expression:

```
void SomeFunction(int size)
{
  int array[size]; // ILLEGAL
  // . . .
  return;
}
```

If you need to use a parameter to specify the size of an array, you can use `new` to allocate the array at the start of a function and use `delete` to free it at the end of the function:

```
void SomeFunction(int size)
{
  int *array = new int[size];
  // . . .
  delete[] array;
  return;
}
```

The problem with this approach is that it is not completely reliable when it comes to preventing memory leaks. The memory allocated by `new` will not get freed if an exception occurs before the call to `delete`.

To ensure that all allocated memory is freed, we need to create an exception handler within the function. Since exception handling in C++ does not include a finally construct we need to use two `delete` calls. One is called when an exception occurs, the other when there are no exceptions. For example:

```
void SomeFunction(int size)
```

```
{
  int *array = new int[size];
  try
  {
    // . . .
  }
  catch(...)
  {
    delete[] array;
    throw;
  }
  delete[] array;
  return;
}
```

The previous example works when you have one array, but what if you need more? Here we add a second array using the same error handling process:

```
void SomeFunction(int size)
{
  int *array1 = new int[size];
  double *array2 = new double[size];
  try
  {
    // . . .
  }
  catch(...)
  {
    delete[] array1;
    delete[] array2;
  }
  delete[] array1;
  delete[] array2;
  return;
}
```

Adding the second array introduces the potential for a new memory leak. If the second `new` call fails it will throw a `bad_alloc` exception. Since the `new` calls are outside of the exception handler, the memory allocated by the first `new` call will not be freed if the second call fails.

We could try to close the potential memory leak by moving the `new` calls within the try block:

```
void SomeFunction(int size)
```

```
{
  int *array1;
  double *array2;
  try
  {
    array1 = new int[size];
    array2 = new double[size];
    // . . .
  }
  catch(...)
  {
    delete[] array1;
    delete[] array2; // Wrong
  }
  delete[] array1;
  delete[] array2;
  return;
}
```

Unfortunately, this introduces a subtle error. If one of the `new` calls fails and throws a `bad_alloc` exception, at least one and possibly both of the pointers will not be initialized. The `delete` call using the initialized pointer in the exception handler will produce unpredictable and usually catastrophic results.

In order to ensure that the `delete` calls will work whether or not the `new` calls succeed, we need to initialize the pointers to null before the first call. This example allocates two arrays in a safe manner:

```
void SomeFunction(int size)
{
  int *array1 = 0;
  double *array2 = 0;
  try
  {
    array1 = new int[size];
    array2 = new double[size];
    // . . .
  }
  catch(...)
  {
    delete[] array1;
    delete[] array2;
    throw;
  }
  delete[] array1;
```

```
  delete[] array2;
  return;
}
```

The goal here is to create a dynamic array, but most of the code is for handling errors that may not even occur.

The obvious way to avoid all of this error handling code would be to create an array class with an overloaded `[]` operator. Since an auto object's destructor is always called when the function exits, even as the result of an exception, using an array class could ensure that that all of the dynamic memory would be freed. If we made the array class a template, we would even be able to use the same code to handle `int` and `double` arrays.

The vector class

The best solution to dynamic arrays is so obvious once you think about it that you might wonder if anyone had thought of it before. Not only did people think of it, they included it in the STL as defined by the ANSI C++ standard. The solution is the `vector` template class.

Using the `vector` class, the previous example looks like this:

```
#include <vector>
void SomeFunction(int size)
{
  std::vector<int> array1(size);
  std::vector<double> array2(size);
  // . . .
  return;
}
```

This example is even simpler than using `new` and `delete` with no error handling, and has the added benefit of being much more readable.

Initialization

Another advantage of the `vector` class over `new` and `delete` is that you can specify an initial value for all of the elements of the array. The second parameter in the constructor specifies the initial value for all elements in the array so this

```
void SomeFunction(int size)
{
```

```
  std::vector<int> array1(size, (int)-1);
  std::vector<double> array2(size, 0.0);
  // . . .
  return;
}
```

is equivalent to this

```
void SomeFunction(int size)
{
  std::vector<int> array1(size);
  for (int ii = 0; ii < size; ++ii)
    array1[ii] = -1;
  std::vector<double> array2(size, 0.0);
  for (int ii = 0; ii < size; ++ii)
    array2[ii] = 0;
  // . . .
  return;
}
```

Range Checking

As with normal arrays, the `vector` class's `[]` operator does not perform range checking. While this makes the `vector` class no less efficient than a C++ array, it makes applications more susceptible to errors.

The `at()` member function allows you to do range checking with a `vector` object. `at()` throws an `out_of_range` exception when the index is not within the bounds of the array. The `at()` function returns a reference so you can use it to make assignments:

```
std::vector<int> array(10);
array[10] = 42; // Bad but no exception
array.at(10) = 42; // Exception
```

You might want to use range checking while debugging and have it disabled at other times for performance. An easy way to do this is to create your own class derived from `vector`. The class shown in **Listing A** called, `Array`, allows you to enable range checking by defining the preprocessor symbol `ENABLECHECKS`. You can use the `Array` class as a substitute for the `vector` class.

The auto_ptr class

While arrays are the most common example of dynamic memory intended to last for a single function call, the problem of ensuring that memory allocated in a function is freed applies to other objects as well.

The `auto_ptr` template class in the standard C++ library provides a simple mechanism to ensure that dynamically allocated objects are deleted when a function exits. The `auto_ptr` class essentially makes objects allocated by `new` behave like an auto object.

The `auto_ptr` class maintains a pointer to an object allocated with `new`. The class defines overloaded `*` and `->` operators that give access to the dynamic object. An `auto_ptr` variable assumes ownership of its related object. When the variable goes out of scope, the `auto_ptr`'s destructor deletes the dynamic object.

The following example shows how to use an `auto_ptr` object to reference a dynamically allocated `ofstream` object. The `auto_ptr` object will delete the `ofstream` object even if an exception causes the function to exit. Here's the code:

```
#include <fstream>
#include <memory>
using namespace std;
void SomeFunction(const string &filename)
{
  auto_ptr<ofstream> strm(
    new ofstream(filename.c_str()));
  if (*strm)
  {
    *strm << "Anyone home" << endl;
    strm->close();
  }

  // . . .
}
```

The `auto_ptr` class defines a copy constructor and assignment operator so that you can have more than one `auto_ptr` variable reference the same object. In addition to creating an additional reference to the same object, the `auto_ptr` class transfers ownership of the object whenever one `auto_ptr` variable is assigned to another, either by the copy constructor or assignment operator.

In order to prevent multiple `auto_ptr` variables from deleting the same dynamic object, `auto_ptr` only deletes the associated dynamic object if it owns the object. Multiple `auto_ptr` variables can reference the same object but only one can own it.

In this example, both `strm1` and `strm2` can be used to access the same `ofstream` object, but only the owner of the object, `strm2`, will delete it when the function exits:

```
#include <memory>
#include <fstream>
using namespace std;
void SomeFunction(const string &filename)
{
  auto_ptr<ofstream> strm1(
    new ofstream(filename.c_str()));
  auto_ptr<ofstream> strm2;
  strm2 = strm1;
  // strm2 now owns the object.

  // . . .
}
```

You need to be careful when assigning one `auto_ptr` variable to another in a different scope. In the following example, `strm2` takes ownership of the `ofstream` object during the assignment. When `strm2` goes out of scope the `ofstream` object gets deleted, at which point `strm1` no longer references a valid object:

```
#include <memory>
#include <fstream>
using namespace std;
void SomeFunction(const string &filename)
{
  auto_ptr<ofstream> strm1(
    new ofstream(filename.c_str()));
  {
    auto_ptr<ofstream> strm2;
    strm2 = strm1;
  }
  *strm1 << 123 << endl; // ERROR

  // . . .
}
```

Just as deadly is associating an `auto`, `static`, or `extern` object with an `auto_ptr` variable. While doing so is legal syntactically, it will produce unpredictable results as the `auto_ptr` attempts to delete the object when it goes out of scope:

```
#include <memory>
#include <fstream>
using namespace std;
```

```
void SomeFunction(const string &filename)
{
  ofstream strm(filename.c_str());
  auto_ptr<ofstream> strm1(&strm);
  // . . .
}
```

Conclusion

The `vector` class in the standard template library is the best mechanism for creating dynamic arrays with one dimension. STL `vector` objects can be referenced exactly like an array allocated with `new`, but `vector` objects have the advantage that dynamic memory is managed for you.

The `auto_ptr` template class provides similar memory management capabilities for generic objects. Your code must allocate the dynamic object but `auto_ptr` handles the deletion.

!!Bob: I've left the margins on this listing wide as I expect this listing to appear on the back page where we have the entire page width to work with. - Kent

Listing A: *The Array class.*

```
template<class TYPE>
class Array : public std::vector<TYPE>
{
public:
  Array () {}
  Array (size_t size) : vector<TYPE>(size) {}
  Array (size_t size, const TYPE &value) : vector<TYPE>(size, value) {}
  Array (const Array &source) : vector<TYPE>(source) {}
  TYPE &operator[](size_t) ;
  const TYPE &operator[] (size_t) const ;
} ;

#if defined (ENABLECHECKS)

template<class TYPE>
inline TYPE &Array<TYPE>::operator[](size_t index)
{
  return vector<TYPE>::at (index) ;
}
template<class TYPE>
inline const TYPE &Array<TYPE>::operator[](size_t index) const
```

```
{
  return vector<TYPE>::at (index) ;
}

#else

template<class TYPE>
inline TYPE &Array<TYPE>::operator[](size_t index)
{
  return vector<TYPE>::operator[](index) ;
}
template<class TYPE>
inline const TYPE &Array<TYPE>::operator[](size_t index) const
{
  return vector<TYPE>::operator[](index) ;
}

#endif
```

Shell and common control versions

by Mark Wiseman

In previous issues I talked about how to detect the version of Windows your program is running on. You learned that you need to know the Windows version in order to use certain API functions and registry settings. But knowing the version of Windows is sometimes not enough.

There are three DLLs that contain the APIs for the Windows' shell and common controls. These DLLs are COMCTL32.DLL, SHELL32.DLL, and SHLWAPI.DLL.

As you might expect, the APIs contained in these DLLs vary based on the version of Windows. But they also vary depending on the installed version of Internet Explorer.

What version is it?

Listings A and **B** are the source code for two functions that you can use to determine the versions of these DLLs.

The `GetDLLVersion()` function takes the name of the DLL (COMCTL32.DLL, SHELL32.DLL, or SHLWAPI.DLL) and returns a `DWORD` that contains the DLL's version. The major version number is in the `HIWORD` and the minor version number is in the `LOWORD`.

We can easily convert the version information to a string using this code:

```
String version =
  String(HIWORD(version)) +
  "." + String(LOWORD(version));
```

The example program uses this technique to display the versions of the three DLLs in a dialog box.

The inline function `PackDLLVersion()` takes the major and minor version numbers and packs them into a `DWORD`. We can use `PackDLLVersion()` to make testing the DLLs' versions easier:

```
if (GetDLLVersion("Shell32.dll")
    >= PackDLLVersion(4,71))
  // Shell32.dll is atleast version 4.71
else
  // Shell32.dll is older
```

What’s next?

In the next issue, you will see how to manipulate the Windows’ Recycle Bin with code. To do so, you will need to know which version of the Shell32.dll is on the system.

Listing A: *DLLVersion.h*

```
#ifndef DLLVersionH
#define DLLVersionH

inline DWORD
PackDLLVersion(DWORD major, DWORD minor)
{
  return((major << 16) + minor);
}

DWORD GetDLLVersion(String dllName);

#endif    // DLLVersionH
```

Listing B: *DLLVersion.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "DLLVersion.h"

#include <shlwapi.h>

DWORD GetDLLVersion(String dllName)
{
  DWORD version = 0;

  HINSTANCE hInst = LoadLibrary(dllName.c_str());

  if (hInst)
  {
    DLLGETVERSIONPROC getVersion =
      (DLLGETVERSIONPROC)GetProcAddress(
      hInst, "DllGetVersion");

      if (getVersion)
      {
```

```
        DLLVERSIONINFO info;
        ZeroMemory(&info, sizeof(info));
        info.cbSize = sizeof(info);

        HRESULT result = (*getVersion)(&info);
        if (SUCCEEDED(result))
          version = PackDLLVersion(
            info.dwMajorVersion,
            info.dwMinorVersion);
      }

      FreeLibrary(hInst);
  }

  return(version);
}
```

Single-instance applications

by Kent Reisdorph

Some applications are built to allow users to run as many instances of the application as the user desires. Other applications allow only a single instance of the application to be run. The VCL model does not contain a pre-built method of allowing only a single instance of an application. This article will show you how to create applications that only allow a single instance.

This article also shows how to pass information from a second instance of an application to the first instance. Consider the case where your application is already running and the user double-clicks a file associated with your application in Explorer (file associations are discussed in the article, "Using file associations"). In that case you would want to prevent a second instance from starting, but yet load the file the user double-clicked in the original instance of the application. This article will explain how to handle that situation.

The example program for this article is the same as the example program for the article, "Using file associations." I will refer you to **Listing A** in that article at points in this article.

Preventing a second instance from running

An application that allows only a single instance requires you to go where you might never have been before—to the project source file. The project source contains a `WinMain()` function. `WinMain()` is the entry point for all Windows GUI applications. The `WinMain()` for a standard VCL GUI application contains code that initializes the `Application` object, creates any forms in the project's auto-create list, and calls `Application->Run()` to start the application. You can view the project source by choosing Project|View Source in C++Builder 4, or View|Project Source in C++Builder 1 and 3. In most VCL applications you never have to look at the project source. When preventing a second instance of an application from running, though, you need to execute code before the VCL gets a chance to initialize the `Application` object.

In the days of 16-bit Windows, detecting a previous instance was easy. The `WinMain()` function contains a parameter called `hPrevInstance`. You only had to examine `hPrevInstance` and see if it contained a valid instance handle (indicating a previously running instance of the program). If it was `0`, there was no previous instance. In 32-bit Windows, `hPrevInstance` is still a `WinMain()` parameter, but it is always `0`.

Preventing a second instance, then, requires you to use some global mechanism to detect that an instance of the application is already running. By global I mean that the mechanism must be available to any Windows application. You can detect an existing instance of an application in one of several ways. One

way is to use `FindWindow()` or `EnumWindows()` to locate a previous instance. Another, more reliable way, is to use a mutex.

Using a mutex

The term mutex comes from the words "mutually exclusive." A mutex is a synchronization object typically used to insure that two or more threads do not attempt to simultaneously access shared memory. Using a mutex is relatively straightforward. When used in this context, the mutex is used in the `WinMain()` function as follows:

?Attempt to read the mutex. If the mutex does not exist then this is the first instance of the application.

?Create the mutex if it does not already exist.

?Release the mutex after `Application->Run()` returns. This only happens when the application closes.

?If the mutex exists then this is a second instance of the application. Terminate the second instance by returning from `WinMain()`.

The following code is the simplest `WinMain()` that can be written given the above steps:

```
WINAPI WinMain(
  HINSTANCE, HINSTANCE, LPSTR, int)
{
  try {
    // Try to open the mutex.
    HANDLE hMutex = OpenMutex(
      MUTEX_ALL_ACCESS, 0, "MyApp1.0");

    if (!hMutex)
      // Mutex doesn’t exist. This is
      // the first instance so create
      // the mutex.
      hMutex =
        CreateMutex(0, 0, "MyApp1.0");
    else
      // The mutex exists so this is the
      // the second instance so return.
      return 0;

    Application->Initialize();
```

```
    Application->CreateForm(
      __classid(TForm1), &Form1);
    Application->Run();

    // The app is closing so release
    // the mutex.
    ReleaseMutex(hMutex);
  }
  catch (Exception &exception) {
    Application->
      ShowException(&exception);
  }
  return 0;
}
```

Note that the calls to `OpenMutex()` and `CreateMutex()` specify a mutex name in their final parameters. The mutex name must be unique or you may end up opening a mutex that belongs to someone else. It is up to you to decide what constitutes a unique name, but any reasonable combination of your application name and version should suffice.

Bringing the application to the front

As I said, the previous `WinMain()` shows the simplest code that will prevent a second instance of the application from running. In most cases, though, you will want to bring the running instance of the application to the front before terminating the second instance. This can be accomplished with only two additional lines of code:

```
if (!hMutex)
  hMutex = CreateMutex(0, 0, "MyApp1.0");
else {
  HWND hWnd = FindWindow(
    0, "File Association Example");
  SetForegroundWindow(hWnd);
  return 0;
}
```

First I use `FindWindow()` to obtain the window handle of the first instance of the application. Next, I call `SetForegroundWindow()` to bring the first instance to the top of all other applications. If your application’s title bar changes based on the file currently open, you may have to use `EnumWindows()` to get the window handle of the running instance.

Passing data to the initial instance

When writing Windows applications, you should always try to anticipate how your customers will use (or abuse) your application. If you have a file association for your application then your users may double-click a document file in Explorer to launch your application. If an instance of the application is already running when that happens, you should bring the application to the top and load the file the user double-clicked. This requires a bit of work to implement, as you must pass the path and file name of the file to the first instance of the application.

Passing data from one application to another in 32-bit Windows is not necessarily straightforward. This is because Windows prevents a process from accessing data owned by another process. In order to pass data from the second instance of the application to the first instance, you must implement some type of shared memory scheme. As with many tasks in Windows, this can be accomplished in many ways. You might use a memory mapped file, a named pipe, or a mailslot. You might even be tempted to write a file to disk that the initial instance can read (although I would consider that approach a hack). Another approach is to use the `WM_COPYDATA` message.

Using the WM_COPYDATA message

Perhaps the simplest way of getting data from the second instance to the first instance is by using the `WM_COPYDATA` message. This message is specifically designed to allow one application to send data to another application. When you send a `WM_COPYDATA` message, you pass the handle of the window sending the message in the `WPARAM`, and a pointer to a `COPYDATASTRUCT` in the `LPARAM`. `COPYDATASTRUCT` is a simple structure:

```
typedef struct tagCOPYDATASTRUCT {
  DWORD dwData;
  DWORD cbData;
  PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

The `dwData` member can be used if you are only passing 32 bits of data to the second instance. If you need to pass a block of memory to the second instance, you set the `cbData` member to the size of the memory block, and the `lpData` member to the address of the memory block.

Windows will guarantee that the data sent in the `COPYDATASTRUCT` will exist until after the `WM_COPYDATA` message has been carried out. As such, you must use `SendMessage()` to send a `WM_COPYDATA` message. You cannot use `PostMessage()`. Here is the code I use to pass the command line from the second instance of the example application to the first instance:

```
if (strlen(cmdLine) != 0) {
```

```
  COPYDATASTRUCT cds;
  cds.cbData = strlen(cmdLine) + 1;
  cds.lpData = cmdLine;
  SendMessage(hWnd,
    WM_COPYDATA, 0, (LPARAM)&cds);
}
```

In this code, `cmdLine` is the command line passed to the application by Windows. The command line is passed in the third parameter to `WinMain()`. Note that C++Builder does not assign variable names to the `WinMain()` parameters so you will have to add the variable name to the function header (see **Listing B**). I set the `cbData` member to the length of the command line text and the `lpData` member to the address of the command line (`cmdLine` is a `char*`). After that, I send the `WM_COPYDATA` message to the first instance's window handle. Remember, I had previously obtained the window handle to the first instance when I brought the application to the foreground. In this case I am not interested in the `WPARAM` so I set it to `0`. I send the address of the `COPYDATASTRUCT` instance in the `LPARAM` (the cast is necessary because `LPARAM` is an `int`). To see this code in its proper context, see the `WinMain()` function in **Listing B**.

Naturally, the application must have code to catch the `WM_COPYDATA` message and to take appropriate action when the message is received. Let's look at that now.

Handling the WM_COPYDATA message

The example program's `WmCopyData()` method is the message handler for the `WM_COPYDATA` message. The code in this method extracts the command line from the `COPYDATASTRUCT` data and either prints or opens a file. The `WmCopyData()` method is shown in **Listing A** of the previous article.

The `WmCopyData()` method has a `TWMCopyData` reference as its parameter. This makes it easy to extract the command line:

```
String S =
  (char*)Message.CopyDataStruct->lpData;
```

I simply cast the `lpData` member to a `char*` and assign the result to a `String` object. I now have the command line that was passed to the second instance of the application. At that point I parse the command line to see if I am printing or if I should open the file passed in the command line.

If you examine the `WmCopyData()` method you will see that I use a temporary `TRichEdit` object to print the contents of the file. I do this so that I can leave the text in the application's RichEdit control intact during printing.

Conclusion

Creating an application that only allows a single instance to run can be challenging at first. This is especially true if your application has a file association. Your users can run your application in many ways, and that always leads to complications. Properly handling a single-instance application is easy if you follow the guidelines in this article.

Listing A: *FileAssociation.cpp*

```
#include <vcl.h>
#pragma hdrstop
USERES("FileAssociation.res");
USEFORM("MainU.cpp", Form1);

WINAPI WinMain(
  HINSTANCE, HINSTANCE, LPSTR cmdLine, int)
{
  try {
    // Try to open the mutex.
    HANDLE hMutex = OpenMutex(
      MUTEX_ALL_ACCESS, 0, "MyApp1.0");

    // If hMutex is 0 then the mutex doesn't exist.
    if (!hMutex)
      hMutex = CreateMutex(0, 0, "MyApp1.0");
    else {
      // This is a second instance. Bring the
      // original instance to the top.
      HWND hWnd = FindWindow(
        0, "File Association Example");
      SetForegroundWindow(hWnd);

      // Command line is not empty. Send the
      // command line in a WM_COPYDATA message.
      if (strlen(cmdLine) != 0) {
        COPYDATASTRUCT cds;
        cds.cbData = strlen(cmdLine);
        cds.lpData = cmdLine;
        SendMessage(
          hWnd, WM_COPYDATA, 0, (LPARAM)&cds);
      }

      return 0;
```

```
    }

    Application->Initialize();
    Application->CreateForm(
      __classid(TForm1), &Form1);
    Application->Run();

    ReleaseMutex(hMutex);
  }
  catch (Exception &exception) {
    Application->ShowException(&exception);
  }
  return 0;
}
```

Using file associations

by Kent Reisdorph

Experienced Windows users know that Windows employs file associations to associate a particular file extension with an application. When you double-click a file with an association in Windows Explorer, Windows will execute the application associated with that file extension and will load the file.

This article will explain how to create a file association for your applications. It will also show how to load the associated file when the Windows shell executes your application.

The example program

Before going on, I want to take a moment to explain the example program for this article. The application is a simple text editor whose main form contains only a main menu, a rich edit component, a file open dialog, and a file save dialog. The program registers a file extension of ZZY (a ZZY file is nothing more than a text file). I chose this extension in the hopes that it would not conflict with other registered extensions. The Tools menu contains two menu items for creating and removing a file association. After creating the file association, you may have to restart Windows before the proper icon will show next to a ZZY file in Explorer.

The program also contains code to load a file when an associated file is double-clicked in Windows Explorer, and to print a file if the user chooses Print from the Explorer context menu.

The example program is not particularly friendly in that it will not prompt you to save a file before loading a new file. The example is also unfriendly in that it doesn't check to see whether the ZZY file association exists before creating the association. I wanted to keep the example simple and, as such, didn't add features that would be found in a real-world application.

One final point about the example program is that it contains some code not discussed in this article. This is because the example is also the example program for the article, "Single-instance applications." The remainder of the code is discussed in that article.

How associations work

When you double-click on a file in Windows explorer, the Windows shell looks up the extension of the file in the registry to see if the extension is registered. If the extension is not registered, Windows displays the Open With dialog box, allowing the user to choose an application to associate with the file type. If the extension is registered, Windows calls the `ShellExecute()` function with a command of

"open." It also passes the name of the file that was double-clicked as a command line parameter. (I discussed `ShellExecute()` in the March, 1999 issue, so I won't go over the specifics again here.)

Associations go further than simply opening a file, though. If you right-click on a text file (.TXT) in Explorer you will see two items at the top of the context menu. The first is named Open. Choosing this menu item is the same as double-clicking the file in Explorer. When you choose Open, NOTEPAD.EXE will be started with the selected file loaded (assuming a default Windows installation). The second menu item is called Print. Clicking this menu item will cause the file to be printed without displaying Notepad at all.

Other file types display even more items on Explorer's context menu. If you right-click on a Microsoft PowerPoint file, for example, you will see context menu items named Open, New, Print, and Show. The items shown on the context menu for a particular file type are obtained from the registry.

There are at least two ways to create a file association in Windows. One way is to right-click a file in Windows Explorer and choose *Open with…* from the context menu. When you do, Windows will display the Open With dialog. Naturally, this method requires user intervention. When you deploy your application you probably don't want to force your users to set up a file association manually.

A better way to create an association is by making various registry entries from your application. A good installation program will make the registry entries for you, but there are times when you need more control over the process.

Registering an association

Registering a file association requires creating two separate registry keys. Both keys are created in the `HKEY_CLASSES_ROOT` section of the registry.

The file extension key

The first key is the name of the file extension, preceded by a dot. As I said earlier, the example program for this article registers a file extension of ZZY. Given that, I created a key with the following path:

```
HKEY_CLASSES_ROOT\.zzy
```

In a production application, you should check the registry to be sure a key does not exist before you attempt to create a new key. If the key already exists, your application will need to either prompt the user to replace the file association, or be prepared to use a different file extension altogether.

The value of this key is linked to the second key you will create. In fact, it is the name of the second key.

For the example program, I gave this key a value of "Test App File." This value can be anything you choose, but, as with the first key, you must be sure the key does not already exist in the registry.

The application association key

The second key has the same name as the default value for the first key. In my case, I created a new key as follows:

```
HKEY_CLASSES_ROOT\Test App File
```

This key must have at least one subkey. Windows uses this subkey when it executes the application. The entire key is structured as follows:

```
HKEY_CLASSES_ROOT
  Test App File
    shell
      open
        command
```

The string given to the `command` key is the full path and file name of the application followed by `%1` (see **Figure A**). For example:

```
C:\MyApp\MyApp.exe %1
```

When Windows launches the application, it replaces the `%1` symbol with the path and file name of the file that was double-clicked in Windows explorer. This value is passed to your application as a command line parameter. I will show you how to access command line parameters later in the article.

Additional keys

There are other subkeys that you can create under the file association key. One such key is the `DefaultIcon` key. This key is used to specify the icon that the Windows shell will display next to files of the registered types. This key is not required if you only have one file type registered and if that file type should use the application icon. Here's how the value of the `DefaultIcon` key looks for an association that specifies the default application icon:

```
C:\MyApp\MyApp.exe,0
```

This specifies that the first icon found in the application's EXE file should be used as the file association's display icon. If your application has more than one file type, you can specify other icons by

changing the icon index that follows the comma. For example, C++Builder has icons for a project file, a form file, a source file, and so on. If you look in the registry under `HKEY_CLASSES_ROOT\BCBProject\DefaultIcon` you will see that the icon for a project file is icon index 4 (for C++Builder 4, at least).

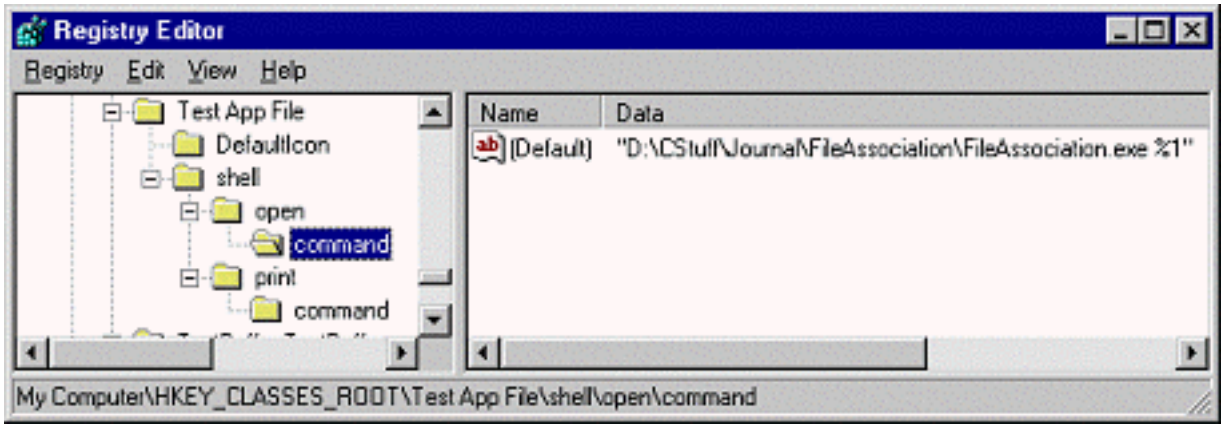
If you want to allow users to print a document you can add a `print` subkey in addition to the `open` subkey. The value of the `print` subkey is similar to that of the `open` subkey, with one exception:

```
C:\MyApp\MyApp.exe /p %1
```

Notice that this value has a command line switch of `/p` inserted between the application name and the `%1` symbol. Your application can watch for the `/p` switch and take appropriate action when the switch is detected.

You can add as many subkeys as you like for a particular file type. The name of each subkey will appear on the Explorer context menu. You only need to add a command line switch for each command type so that your application can identify the context menu item that was selected. If you provide a default value for the subkey, Windows will use that text for the context menu item text. If you do not supply a default value, Windows will use the key name itself for the menu item. **Figure A** shows the Windows registry Editor displaying the keys created by the example application.

Figure A



The example program creates a key called "Test App File" to implement file associations.

The code used to create these registry keys is basic `TRegistry` code so I won't explain it here. Instead, I'll refer you to the `CreateFileAssociation1Click()` method in **Listing A**.

Handling the code in your application

Once you have the file association registered, you must write code in your application to handle the file

name Windows passes to your program. This part is relatively easy. Let's say, for example, that you have written a simple text editor and that you want to simply open the file that Windows passes to your program. In that case you can place code like this in your form's `OnCreate` event handler (alternatively you can place this code in the `OnShow` event handler or in the form's constructor):

```
if (ParamCount() == 1)
  Memo->Lines->LoadFromFile(ParamStr(1));
```

The `ParamCount()` function returns the number of parameters passed to the application. The `ParamStr()` function returns a specific parameter by index. `ParamStr(0)` is always the path and file name to the application itself, so `ParamStr(1)` will return the first parameter passed by Windows.

If your application allows printing from Explorer or other special commands, then you will have to write code that looks for the command line switches and execute code based on the particular switch passed. The `FormCreate()` method in **Listing A** shows how to handle a command line switch for printing a text file.

Conclusion

The code for the example program's main form is shown in **Listing A**. When reviewing the code, keep in mind that some of the code is discussed in the next article.

There is no magical Windows API call for creating file associations, but the job is handled easily enough with just a few lines of `TRegistry` code. Handling a shell launch of one of your application's files only requires a few lines of code as well. Creating one or more file associations for your application adds a professional touch that today's users expect.

Listing A: *MainU.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  // If there are two params then they are the
  // /p switch followed by the file name.
  if (ParamCount() > 1) {
    // First param is the print command.
    if (ParamStr(1) == "/p") {
      Visible = false;
      // Load the file, print, and exit.
      RichEdit->Lines->LoadFromFile(ParamStr(2));
      Print1Click(0);
      Application->Terminate();
    }
  }
  // Not printing, just load the file.
  else if (ParamCount() == 1)
    RichEdit->Lines->LoadFromFile(ParamStr(1));
}

void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenDialog->Execute())
    RichEdit->Lines->LoadFromFile(
      OpenDialog->FileName);
}

void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (SaveDialog->Execute())
    RichEdit->Lines->SaveToFile(
      SaveDialog->FileName);
}

void __fastcall TForm1::Print1Click(TObject *Sender)
{
  RichEdit->Print("Test App Document");
}

void __fastcall TForm1::Exit1Click(TObject *Sender)
{
  Close();
}
```

```
void __fastcall
TForm1::CreateFileAssociation1Click(TObject *Sender)
{
  // Create the registry keys.
  TRegistry* reg = new TRegistry;
  try {
    reg->RootKey = HKEY_CLASSES_ROOT;
    // Create the file extension key.
    reg->OpenKey(".zzy", true);
    reg->WriteString("", "Test App File");

    // Create the file association key and
    // its open, print, and DefaultIcon keys.
    reg->OpenKey("\\Test App File\\"
      "shell\\open\\command", true);
    reg->WriteString("", ParamStr(0) + " %1");
    reg->OpenKey("\\Test App File\\"
      "shell\\print\\command", true);
    reg->WriteString("", ParamStr(0) + " /p %1");
    reg->OpenKey(
      "\\Test App File\\DefaultIcon", true);
    reg->WriteString("", ParamStr(0) + ",0");

    MessageDlg("File association created!\r\n"
      "You may have to restart Windows to see "
      "the proper icon for a ZZY file.",
  }
  __finally {
    delete reg;
  }
}

void __fastcall
TForm1::RemoveFileAssociation1Click(TObject *Sender)
{
  // Delete the two registry keys.
  TRegistry* reg = new TRegistry;
  try {
    reg->RootKey = HKEY_CLASSES_ROOT;
    reg->DeleteKey(".zzy");
    reg->DeleteKey("Test App File");

    MessageDlg("File association removed!",
      mtInformation, TMsgDlgButtons() << mbOK, 0);
```

```
  }
  __finally {
    delete reg;
  }
}

void __fastcall
TForm1::WmCopyData(TWMCopyData& Message)
{
  String S = (char*)Message.CopyDataStruct->lpData;
  int pos = S.Pos("/p");
  if (pos) {
    // Printing. Create a temp TRichEdit
    // to do the printing.
    S = S.Delete(1, pos + 2);
    TRichEdit* re = new TRichEdit(this);
    re->Visible = false;
    re->Parent = this;
    re->Lines->LoadFromFile(S);
    re->Print("Test App Document");
    delete re;
    return;
  } else {
    // Not printing, just load the file.
    RichEdit->Lines->LoadFromFile(S);
    OpenDialog->FileName = S;
    SaveDialog->FileName = S;
  }
}
```

ActiveForms, Part I

by Bob Swart

In this article, you will see how to build and deploy ActiveForms using C++Builder… all without writing a single line of code! In Part II of this series I will show you how to deploy your ActiveForms. In Part III, I will show you how to build thin client ActiveForms by incorporating Inprise MIDAS techniques.

Creating an ActiveForm

`TActiveForm` is the base class for a VCL form exposed as an ActiveX control. `TActiveForm` represents a form that is created via the Active Template Library (ATL) and used as an ActiveX control in a host application. This makes it very easy to embed visual and non-visual components on the ActiveForm. The result is a complex ActiveX control that can be used in C++Builder itself, on a Web page, or in other development environments such as Visual Basic or Delphi.

To create an ActiveForm in C++Builder, first select File | New from the main menu. Next click the ActiveX tab in the Object Repository, and double-click the ActiveForm icon. When you do, the ActiveForm Wizard will be displayed. In the ActiveForm Wizard, you need to specify the name of the ActiveForm (the default name is `ActiveFormX`), the name of the implementation unit, and the project name. When you change the ActiveX name, C++Builder will change the name of the implementation file and the project file. You can change the filenames, of course, to use names you prefer. The project name will also result in the final name of the ActiveX control. For this example I named the project `ActiveF` so the final output file will be named ACTIVEF.OCX. Starting with version 4 of C++Builder, the threading model is set to Apartment. This model is required to use the ActiveForm in Internet Explorer version 4 or higher.

You have the option to include an About box for your ActiveForm, a design-time license, and version information. Including a design-time license will result in an ActiveForm that can only be used in a run-time environment, or at design-time if the license file is present. More importantly, choosing this option will also result in an ActiveForm that won't work inside Internet Explorer because the ActiveX is designated as a design-time control. I seldom use this option, but it can certainly be useful for demo editions of commercial ActiveForms.

The option that you should always check is Include Version Information. You'll need this information when deploying new versions of your ActiveForms (updates, bug fixes, and so on). Obviously, without version information, there is no way the browser can detect that an update is available for download.

Easy COM, easy go

After you click on the OK button of the ActiveForm Wizard, C++Builder generates a number of source files. One file is the source for your new ActiveX project (ACTIVEF.CPP in this case). This file defines the four routines you need to export in order to turn the library into a real OCX control.

Another file is the source for the ActiveForm itself, ACTIVEFORMIMPL.CPP, which contains almost 650 lines of code. Much of this code consists of complex COM interface methods that you really don't need to be concerned with. This is especially true if you just want to design and deploy an ActiveForm inside a Web browser without learning everything there is to know about COM, type libraries, and DCOM.

If you take a look at the ActiveX project source file, you will see that the project uses two more files called ACTIVEF_ATL.CPP and ACTIVEF_TLB.CPP. These two files are the ATL source and the COM type library for the ActiveForm. The main thing you need to know about these files is that they are generated automatically whenever the type library changes. As such, you should never make any manual changes to these files, as they will always be overwritten when the type library is regenerated.

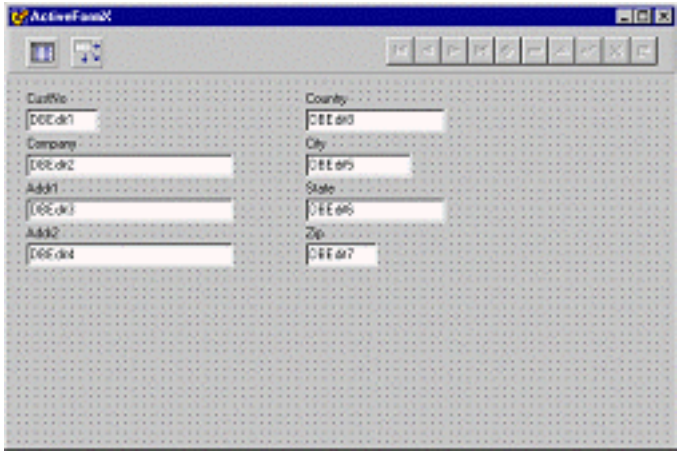
For the rest of this article you should consider the ActiveForm a regular visual form that merely needs to be deployed in a somewhat special manner. Before you continue, however, you must make sure to create a stand-alone ActiveForm, one that doesn't require additional files or packages. To do this, go to the Packages tab of the Project Options dialog and un-check the *Build with runtime packages* option. Next, select the Linker page and uncheck the *Use dynamic RTL* option. I'll explain the importance of creating a stand-alone ActiveForm in Part II of this series when I discuss how to deploy your ActiveForm.

An ActiveForm client

As an example, you will create an ActiveForm that shows a master-detail relationship between two tables. First, resize the form to 600 x 400. Next drop a `TPanel` component on the ActiveForm, clear its `Caption` property, and set its `Align` property to `alTop`. Now drop a `TTable` (name it `TableCustomer`), a `TDataSource`, and a `TDBNavigator` on the panel. Set the DataSource component's `DataSet` property to `TableCustomer`, and the DBNavigator's `DataSource` property to `DataSource1`. Set the Table's `DatabaseName` property to BCDEMOS, and its `TableName` property to CUSTOMER.DB.

Now double-click on the table component to invoke the Fields Editor. Right-click in the Fields Editor and choose *Add all fields* from the context menu. This will add all fields from the table into the fields list. Select the fields you want to display on the ActiveForm, drag them to the client area of the ActiveForm, and arrange them to your liking. I added the customer number and address information so I ended up with the form shown in **Figure A**.

Figure A



The ActiveForm looks like a regular form at design time.

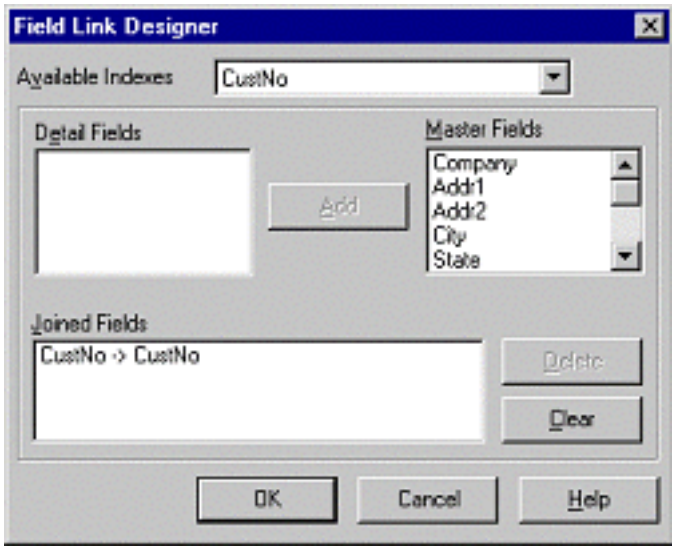
Before you continue, you can make sure the database navigator bar always stays at the upper-right corner of the ActiveForm. In C++Builder 4 you can do this easily by modifying the `DBNavigator`'s `Anchors` property. Set the `Anchor` property's `skLeft` value to `false`, and `akRight` to `true`. Now the `DBNavigator` will always be in the upper-right corner of the ActiveForm.

The master-detail relationship

At this point you have the master table finished and you need to set up the details table. Drop another `TTable` on the ActiveForm and name it `TableOrders`. Drop a `TDataSource` and a `TDGBrid` component on the form as well. Connect the DBGrid to the `TableOrders` table through `DataSource2.` Set the DBGrid's `Align` property to `alBottom`. Set the table's `DatabaseName` property to BCDEMOS and its `TableName` property to ORDERS.DB.

To define a master-detail relationship, select the `TableOrders` table and set its `MasterSource` property to `DataSource1`. Now select the `MasterFields` property in the Object Inspector and click on the ellipsis button to bring up the Field Link Designer. In the Field Link Designer, select the `CustNo` index from the Available Indexes combo box. Then select the `CustNo` field in both the Detail Fields and Master Fields list boxes. Now you only need to click on the Add button to link the two tables in a master-detail relationship as shown in **Figure B**. When you set the `Active` property of `TableCustomer` and `TableOrders` to `true`, and you get live data at design-time.

Figure B



Setting up a master-detail relationship is easy with the Field Link Designer.

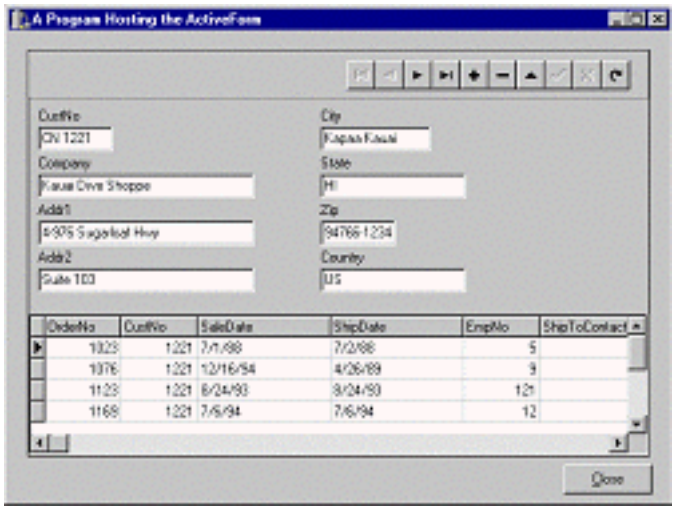
Using the ActiveForm

At this point your ActiveForm is completed. With a few exceptions, all components work just like on a regular form. The exceptions are menus (which will not appear) and the caption of the ActiveForm, which will disappear when the ActiveForm is shown at run-time. (If you want your ActiveForm to show a caption, drop a panel on the form, set its `Align` property to `alTop`, and give it the color normally used for the active caption.)

Now save the project and then compile it. After the project is built, choose, Run | Register ActiveX Server from the C++Builder main menu. This registers the ActiveX control with Windows. Next you need to install the ActiveForm into the C++Builder component palette. Close your project and then choose Component | Import ActiveX Control from the main menu. Find "ActiveF Library" in the list of installed controls, select it, and click the Install button. C++Builder will then ask you for the name of the package in which to install the ActiveX control. Choose the default user package, DCLUSR40.BPK, and click OK. C++Builder will then build the package. After the package is built your ActiveForm will be on the ActiveX page of the component palette.

Now create a new project. Select the ActiveFormX control from the ActiveX page and drop it on your form. Resize the control to your liking and run the program. It’s as easy as that! **Figure C** shows the ActiveForm hosted in a C++Builder program.

Figure C



Here the ActiveForm is hosted in a C++Buidler program.

Conclusion

ActiveForms are convenient for creating complex ActiveX controls. This is especially interesting when you consider that the ActiveForm can be used on a Web page. Next month, in Part II of this series, I will show you how to deploy an ActiveForm on a Web page.

Faster rich edit syntax highlighting

By Robert Dunn

Several weeks ago, I received an e-mail from a programmer who had written a syntax-highlighting editor using a Rich Edit control. It was inexplicably slow and he wanted to know how improve the performance of the syntax highlighting. Having received such queries before, I offered up my standard answers: Disable screen redraws during parsing and formatting, use Win32 API calls instead of `TRichEdit` properties and methods for fetching text from the control and setting character formats, and so on.

The programmer replied that he had done all of those things. I probably would not have given it any more thought had this programmer not mentioned that he had isolated the portion of the code that was consuming the bulk of the cycles. It was not in his syntax parser as I would have expected. Instead, the Win32 API `EM_SETCHARFORMAT` message alone seemed to be the culprit.

This puzzled me. Yes, I would expect applying color to the text to take a bit of time, but his experience was that this message was taking a surprisingly long time to execute. I did not give it much more thought until several weeks later when, as I was falling asleep, I thought I might have an answer.

A basic syntax-highlighting edit control

To test my nocturnal revelation, I cobbled together a very small, and very flawed, syntax-highlighting editor. I assumed that most programmers would design their editors something like this:

1. Parse and highlight the entire file immediately after loading it.
2. Parse the line again (or a portion of it) whenever the user typed some text by implementing an `OnChange` handler.
3. Track the insert/overstrike mode and display it on a status bar by implementing an `OnKeyDown` handler.
4. Track the current cursor-position line number and display it on a status bar by implementing an `OnSelectionChange` handler.

With these design expectations and goals in mind, I threw together a quick program. First I created a new project and then added a main menu, a status bar with a couple of panels, and a `TRichEdit`. Next I added an `OnKeyDown` handler to the `TRichEdit` to track the insert/overwrite state:

```
void __fastcall TForm1::RichEdit1KeyDown(
  TObject *Sender, WORD &Key,
  TShiftState Shift)
{
```

```
  // initialized to shift-state of none
  static TShiftState ss;
  static AnsiString Ins("INS");
  static AnsiString Ovr("OVR");
  if (Shift == ss && Key == VK_INSERT)
    StatusBar1->Panels->Items[1]->Text =
     (StatusBar1->Panels->Items[1]->Text
        == Ins) ? Ovr : Ins;
}
```

I then added an `OnSelectionChange` handler to the `TRichEdit` to display in the status bar the number of the line containing the cursor:

```
void __fastcall TForm1::
RichEdit1SelectionChange(TObject *Sender)
{
  StatusBar1->Panels->Items[0]->Text =
    AnsiString("Line ") +
    AnsiString(::SendMessage(
      RichEdit1->Handle, EM_LINEFROMCHAR,
      RichEdit1->SelStart, 0));
}
```

Next I added a Parse item to the main menu and an `OnClick` handler for the menu item. The `OnClick` handler kicks off a parse of the entire file by calling a function called `ParseAllText()`. I also added some code around this call to measure the effectiveness of optimizations.

The `ParseAllText()` function is excruciatingly simple. It walks through the text using the `EM_FINDWORDBREAK` message to break the text into tokens. It then calls a function called `GetTokenColor()` that looks up a token in a list and returns the associated color for the token. `ParseAllText()` then calls `SetTokenColor()` to set the text's character format to the selected color. Here's the `SetTokenColor()` function:

```
void SetTokenColor(TRichEdit* RichEdit1,
  TEXTRANGE& tr, TColor color)
{
  int selStart = RichEdit1->SelStart;
  int selLength = RichEdit1->SelLength;
  RichEdit1->SelStart = tr.chrg.cpMin;
  RichEdit1->SelLength =
    tr.chrg.cpMax - tr.chrg.cpMin;
  RichEdit1->SelAttributes->Color=color;
```

```
  RichEdit1->SelStart = selStart;
  RichEdit1->SelLength = selLength;
}
```

Finally, I added an `OnChange()` handler for the `TRichEdit` to reparse lines whenever the user changes the text. My version is as lame as the `ParseAllText()` function mentioned earlier. It simply moves the cursor back by a couple of words and calls `GetTokenColor()` and `SetTokenColor()` for each of the next few words.

Optimizing The Code

I timed several optimizations of my program, the results of which are presented in **Table A**. The baseline routine was, indeed, incredibly slow. Of course, the Rich Edit control is redrawing and scrolling every time the text or cursor position changed during the parsing/formatting pass so the lack of execution speed is not surprising.

Table A: Comparison of Optimization Routines

Optimization	Clock Ticks	Percent of Non-Optimized
None (baseline)	189,363	100.0
Optimization 1	87,373	46.1
Optimizations 1 & 2	57,881	30.6
Optimizations 1, 2, & 3	10,207	5.4
Optimizations 1, 2, & 4	8,187	4.3

Optimization 1

The first optimization is incredibly obvious. I simply disabled redrawing of the Rich Edit by wrapping the `ParseAllText()` call with `WM_SETREDRAW` messages:

```
::SendMessage(RichEdit1->Handle,
  WM_SETREDRAW, false, 0);
ParseAllText(RichEdit1);
::SendMessage(RichEdit1->Handle,
```

```
  WM_SETREDRAW, true, 0);
::InvalidateRect(
  RichEdit1->Handle, 0, true);
```

As expected, this made a significant difference and cut the parsing time by more than half. However, I did not feel much of a sense of pride since it was such an obvious optimization.

Optimization 2

The second optimization involves changing `SetTokenColor()` to use Win32 API calls instead of using the `TRichEdit` methods and properties to apply character formatting. After applying this code the function now looks like this:

```
void SetTokenColor(TRichEdit* RichEdit1,
  TEXTRANGE& tr, TColor color)
{
  CHARRANGE chrgSave;
  ::SendMessage(RichEdit1->Handle,
    EM_EXGETSEL, 0, (LPARAM) &chrgSave);
  ::SendMessage(RichEdit1->Handle,
    EM_EXSETSEL, 0, (LPARAM) &tr.chrg);
  CHARFORMAT cf;
  memset(&cf, 0, sizeof(cf));
  cf.cbSize = sizeof(cf);
  cf.dwMask = CFM_COLOR;
  cf.crTextColor = color;
  ::SendMessage(RichEdit1->Handle,
    EM_SETCHARFORMAT,
    SCF_SELECTION, (LPARAM) &cf);
  ::SendMessage(RichEdit1->Handle,
    EM_EXSETSEL, 0, (LPARAM) &chrgSave);
}
```

Again, this made a significant difference—about 1/3 faster than Optimization 1 alone. However, it is still a rather obvious optimization and one that I would have previously recommended.

Optimization 3

These first two optimizations were all I had to offer my correspondent. It was not until I thought about how one might write a syntax-highlighting editor that I realized that the problem might not be in the Rich Edit control itself. Instead, it might be in the `TRichEdit` VCL component and, perhaps, in the manner

that the programmer had structured his program.

Remember the `OnSelectionChange` and `OnChange` handlers that I mentioned earlier? These are called every time the parser moves the cursor or changes the character format. This happens regardless of whether we enable or disable screen redraws. Worse, my `OnChange` event handler is actually reparsing every time `ParseAllText()` changes the text format. Clearly, this was not a good design decision.

My next thought was to set these handlers to `NULL` before calling `ParseAllText()` and restore them upon return. Here's the code:

```
RichEdit1->OnChange = 0;
RichEdit1->OnSelectionChange = 0;
RichEdit1->OnKeyDown = 0;
::SendMessage(RichEdit1->Handle,
  WM_SETREDRAW, false, 0);
ParseAllText(RichEdit1);
::SendMessage(RichEdit1->Handle,
  WM_SETREDRAW, true, 0);
::InvalidateRect(
  RichEdit1->Handle, 0, true);
RichEdit1->OnChange = RichEdit1Change;
RichEdit1->OnSelectionChange =
  RichEdit1SelectionChange;
RichEdit1->OnKeyDown = RichEdit1KeyDown;
```

Wow! As you can see from **Table A**, Optimization 3 cut the execution time to about 6% of the time for Optimizations 1 and 2 alone. Granted, this optimization simply fixes a flaw in the original program design (calling the `OnChange` handler when it is not needed). Maybe it was an obvious optimization, maybe not. Still, I was starting to feel a little proud of my efforts thus far. However, I found that I could do even better.

Optimization 4

Understanding the last optimization requires a bit of insight into how Rich Edit controls provide notification when selection changes, text changes, and other events occur.

By default, Rich Edit controls do not provide any notification of such events. In order to get event notifications, you create an "event mask" specifying the events your program wants to be notified of. You then pass this mask to the Rich Edit control through the `EM_SETEVENTMASK` message. The `LPARAM` for this message is a mask with a bit set for each type of notification type that you wish to receive. For each enabled event, the Rich Edit control's parent window will receive an `EN_XXX`

message.

`TRichEdit` makes these events available to your program through the familiar `OnXXX` events. When a `TRichEdit` control is created (more accurately, when the underlying Rich Edit control's window is created), `TRichEdit` sets the event notification mask to notify the control's parent whenever the text is changed (the `OnChange` event), the selection is changed (the `OnSelectionChange` event), the window needs to be resized (the `OnResizeRequest` event), or whenever text is marked or unmarked as protected (the `OnProtectChange` event). For each of these events, the underlying Rich Edit control will send a notification message to the `TRichEdit`'s parent to tell it that the event occurred. The parent control will forward this message to the `TRichEdit`. The `TRichEdit` will then check to see if you have assigned a handler for the event and, if so, call it.

The key point here is that all of the above processing occurs whether or not your program has installed a handler for a specific event. The parent control is notified and it passes the message on to the `TRichEdit` control. The `TRichEdit` then checks for an installed handler. All of this takes time. In the previous optimization, we simply eliminated the custom event handlers from being called. Now, let's eliminate the notification messages entirely.

To do this, I wrapped the `ParseAllText()` call with code to save the Rich Edit control's event notification state, tell it not to generate notification messages, and restore the event notification state afterwards. I no longer need to disable the `OnXXX` handlers individually as I did in Optimization 3. The relevant code now looks like this:

```
int eventMask =
  ::SendMessage(RichEdit1->Handle,
  EM_SETEVENTMASK, 0, 0);
::SendMessage(RichEdit1->Handle,
  WM_SETREDRAW, false, 0);
ParseAllText(RichEdit1);
::SendMessage(RichEdit1->Handle,
  WM_SETREDRAW, true, 0);
::InvalidateRect(
  RichEdit1->Handle, 0, true);
::SendMessage(RichEdit1->Handle,
  EM_SETEVENTMASK, 0, eventMask);
```

Table A shows that, using Optimization 4, the code executes in a fraction of the time required with only the first two optimizations. Further, it is 20% faster than optimization 3. The results are impressive for just a few lines of code.

Conclusion

The VCL `TRichEdit` component greatly simplifies coding reasonably sophisticated applications. However, it is designed for convenience, not for speed.

Simple and obvious optimizations (Optimizations 1 & 2) may significantly improve your syntax-highlighting editor's performance. Careful attention to potential program design flaws (discovered in Optimization 3) is equally important. An understanding of the implementation of the `TRichEdit` component combined with careful study of the underlying Rich Edit control can improve your code even more as proven by Optimization 4.

For more information on Rich Edit controls and the `TRichEdit` component, see my Web site at **http://home.att.net/~robertdunn/Yacs.html**.

Using STL containers

by Bill Whitney

One of the challenges of coding in C++ is managing the data used by your programs. How, for example, do you store and manipulate a collection of objects that you intend to work with while your program is running?

The choice of an array as the underlying mechanism is not always the most flexible nor the most memory efficient approach. How, for instance, do you choose an initial size? How do you manage "holes" in the array caused by objects that are removed? How do you locate the objects you've stored in the array? What if you want to store and retrieve objects using a key? In all of these instances, you would have to write custom code around your array to accomplish all of these tasks.

This article will give you a head start to solving data management problems like these, using the Standard C++ Library that ships with C++ Builder. This library, which includes the containers and algorithms known as the Standard Template Library (STL), addresses these problems by employing C++ templates to implement containers that can store, manipulate, and retrieve objects. Although the examples in this article are extremely simple, they give you some idea of the power of the STL.

Background

There are some concepts that you should understand before we explore the STL examples. The following sub-sections provide a quick overview.

Iterators

An iterator is a pointer-like mechanism that allows you to reference items in a container. Iterators can be used to specify a particular item or range of items, and are used as parameters to some of the most powerful container management functions. Iterators will be used in the code throughout the article, and described in more detail with each example.

Containers & memory management

Containers can hold just about anything, including primitives (such as `int`, `float`, and `char`), user defined types (including objects), and pointers. There are some basic rules you should be aware of when working with them.

All values placed into a container are copied there using the assignment operator (=). The effects of this are clear when using primitive C++ types or pointers; the value of the primitive or the pointer is simply placed into the container.

If you are placing objects in a container, however, there are ramifications to the assignment operator. The objects are still copied, but how you code the class controls whether the copy is *deep* or *shallow*. For a detailed explanation of deep and shallow copies, you should consult a good C++ text.

In a nutshell, a deep copy is accomplished via a copy constructor, and should be used if your object contains pointers to anything (other objects, character buffers, etc). This insures that you get a distinct object that has allocated its own memory or instantiated its own version of any aggregate objects. In most cases this probably won't be an issue, as you will often be storing pointers in containers instead of actual objects.

Here are some tips to remember when *removing* primitives, pointers, or objects from a container:

- Removing a primitive or a pointer causes it to be discarded by the container.

- Removing an object causes it to be discarded by the container as well, but the object's destructor is called in the process.

- If you tell a container to remove a pointer (whether it's a character buffer or an object), that memory is lost (leaked) unless you've retained a local copy of that pointer with which you can later call `delete`. Containers do not automatically delete memory!

Here are some tips to remember when *retrieving* primitives, pointers, or objects from a container:

- Retrieving a primitive gives you a copy in your local variable.

- Retrieving an object will copy it to your local variable (using a deep or shallow copy depending on whether you've supplied a copy constructor).

- Retrieving a pointer gives you the value of the pointer so that you can reference the "contained" object (to invoke methods, for example).

Namespaces

In large systems, the danger of names clashing between classes and libraries increases. To mitigate that risk, namespaces were added to C++. They are a means of packaging classes and data so that they can be referenced without conflict. The STL, for example, lives in a namespace called `std`.

If you look just below the `#include` statements in **Listing A** you'll see this line:

```
using namespace std;
```

The purpose of this is to let the compiler know that we'll be using members of the `std` library. This saves you a bit of "resolution specification" typing later on. For example, look at the following vector definitions:

```
vector<int> x;
std::vector<int> y;
```

The declaration for vector `x` appears in a file containing the `using` directive. The declaration for `y` does not, and requires that you supply sufficient information for the compiler to locate the correct vector.

STL vectors

We'll start with the vector, as it is most like the C++ array. The major differences are the management of memory to hold the vector (dynamic), and the means of accessing the items therein. The sample program defines a vector of integers (see the `VectorButtonClick()` function in **Listing A**. It contains a vector declaration called `myInts`. You can create a vector of virtually any type by inserting that type within the `<>` symbols, such as:

```
vector<float> myFloats;
vector<TButton*> myButtons;
```

The declarations simply indicate the type of elements the container holds, and a name by which to reference the container.

Managing vectors

The `VectorButtonClick()` function creates a vector of integers, and shows you how to add and delete members. At first, manipulating the vector contents through pushes and pops may seem a little strange. It's the price you pay for the flexibility of the particular container you're using, and you get used to it with practice.

The code places the values 1 through 9 in the vector using the `push_back()` function. This appends the values to the existing vector. It then adds the value `10` to the end, and `0` at the front.

Adding the value `10` is easy because we already know how to use `push_back()`. To accomplish the insertion, though, we have to use an iterator. We call the `insert()` function and tell it which vector

member to *insert before* via the `begin()` function (`begin()` returns an iterator pointing to the first value in the vector).

Notice that vectors support the use of the array subscript operator (`[]`) to access values at an absolute location (just like an array). You should only use the array subscript operator when you know that the offset you are accessing actually exists in the vector.

Next, the value `10` is popped off of the back of the array, and replaced with `25`. After showing the values in `ListBox1` one last time, the vector is cleared (removing all members). The call to `clear()` isn't really necessary because the vector is about to go out of scope, but it's a function you should be aware of.

Because we've stored primitives in the vector, we aren't concerned with the container's contents. If the vector were holding pointers, then we would have walked through them and deleted the memory prior to calling `clear()` or letting the container go out of scope. The following section on lists shows how pointers are dealt with.

To demonstrate any of the examples in this article, run the program and click on the button labeled with the name of the container you want to see.

STL lists

A list stores a linear sequence of items. While it does share many of the characteristics of a vector, it doesn't support random access (via the `[]` operator). You can, however, add, delete, or retrieve values anywhere in a list using an iterator.

If there is a lot of variation in the size of your collection throughout its lifetime, a list might be a better choice than a vector. Vectors can incur performance penalties for growth, whereas the overhead associated with list operations remains relatively constant regardless of the size of the list.

Managing lists

The `ListButtonClick()` function shown in **Listing A** implements the list example. The example stores a list of words and then re-arranges them to form a sentence. You'll notice that the `list` template also supports some of the same functions as `vector` (such as the `push_back()` function).

In this example, we're storing pointers instead of primitives. This means that we're responsible for memory management. It also means that there's an extra step (some indirection) when retrieving list contents.

The example begins by creating a list that holds pointers to four objects of type `Wisdom` (each of which

manages a character buffer holding a word that we'll use later to create the sentence). As each `Wisdom` object is instantiated, `push_back()` is used to add the new object's address to the end of the list. We don't show the `Wisdom` class definition, but you can download the complete code example from our Web site at **www.bridgespublishing.com**.

Next, we'll use the `wisIt` iterator to step through the entire list (using the `begin()` and `end()` functions to signify the list's boundaries). Not only can iterators be used in `for` loops, but they also support the increment (`++`) and decrement (`--`) operators to move back and forth. Don't forget that the `wisIt` iterator is "pointing" to a pointer! The proper way to call the `say` method on the `Wisdom` object is:

```
(*wisIt)->say();
```

Because we placed the word "Have" at the end of the list, the `for` loop used to print out the sentence dutifully constructs "a nice day Have." This is obviously wrong, but can be fixed by removing "Have" from the end of the list and placing it at the beginning. I do this by pointing `wisIt` to the end of the list, and retrieving *Have*'s pointer. Once I have the pointer, I supply it as an argument in a call to `remove()` to take it out of the list.

Next, the `push_front()` method is called to insert the pointer (which we're still holding on to) at the beginning. Now when the sentence is constructed, the wording is correct.

One of the last things the list example does is call `delete` on all of the `Wisdom` objects after adding the words to a character array called `sentence`. This way no memory will be lost when `myWisdom` goes out of scope at the end of the `ListButtonClick()` function.

Maps

Maps are one of the most useful containers. You can store virtually anything in a map and retrieve it with a key (the key must be unique with the map container—if you need duplicate keys, check out the `multimap` template).

In the map example, the `string` type from the Standard C++ Library was used (note the lower case *s*). Don't get this class confused with the VCL's `String` class!

Managing maps

The example (found in the `MapButtonClick()` function in **Listing A**) stores some objects containing the full name of a president and a meaningless integer value. One of the most interesting features of maps is the ability to insert and retrieve objects using brackets (like array subscripting). For example, we added

and retrieved Lincoln like this:

```
anyBody["honest"] =
  Person("Abe Lincoln", 98);
Person who = anyPerson["honest"];
```

In this example `anyBody` is an STL `map` container, and `Person` is a class I created to manage people. Be aware, however, that searching for a person using a construct like `anyPerson["honest"]` will cause that person to be created in the map if he or she doesn't exist! For this reason, any object you plan to store in a map must have a default constructor (one that takes no arguments so it can be called to create a "vanilla" object). An alternative search method (and one without any undesirable side effects) is the `find()` function, which sets an iterator to point to the object if it exists (or the end of the container if it doesn't):

```
map<string, Person>::iterator persIter;
persIter = anyBody.find("honest");
if (persIter != anyBody.end())
  // then the record existed
else
  // it didn't!
```

There are many other useful functions that operate on the map container that you can learn from any good STL book.

Algorithms

Besides a number of containers and iterators, the STL also comes with some algorithms that you can apply to container contents. **Table A** describes some of the more interesting ones.

Table A: STL Algorithms

FunctionDescription

`for_each`Apply a function to each member of a container.

`copy`Copies a sequence of members from one container to another.

`fill`Fills a range in a container with a specified value.

`max_element`Finds the largest element within a range in a container.

`merge`Merges two sequences into a third.

`remove`Removes a specified range of elements from a container.

`reverse`Reverses a specified range.

`sort`Sorts a specified range.

Here's an example of using the `copy` algorithm on two integer vectors. It copies the entire contents of vector `a` to vector `b`:

```
copy(a.begin(), a.end(), b.begin());
```

Conclusion

The Standard C++ Library's containers are powerful features of the language, and the simple examples in this article are only the tip of the iceberg. If you plan to explore containers in more depth, it's a good idea to pick up some books about templates and the STL.

Listing A: Unit1.cpp

```
#include <vcl.h>
#include <stdlib>
#include <vector>
#include <list>
#include <map>
#include <string>
#pragma hdrstop

#include "Unit1.h"
#include "WrkObjs.h"

using namespace std;

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
```

```
void __fastcall
TForm1::QuitButtonClick(TObject *Sender)
{
  exit(0);
}

void __fastcall
TForm1::VectorButtonClick(TObject *Sender)
{
  vector<int> myInts;

  ListBox1->Clear();
  ListBox1->Items->Add("Vector Example");
  ListBox1->Items->Add("==============");
  // Add numbers 1-9 to myInts
  for( int y = 1; y < 10; y++ )
    myInts.push_back(y);

  // Add them to ListBox1 using array subscripting
  ListBox1->Items->Add(
    "After adding 1 through 9:");
  for(int y = 0; y < 9; y++)
    ListBox1->Items->Add(String(myInts[y]));

  // Add 10 to the end of the vector
  myInts.push_back(10);

  // Insert a 0 at the front of the list
  myInts.insert(myInts.begin(), 0);

  // Show the contents of the vector
  ListBox1->Items->Add("After adding 0 and 10:");
  for( int y = 0; y < 11; y++ )
    ListBox1->Items->Add(String(myInts[y]));

  // For practice, remove the 10 from the end
  // and place a 25 in the there.
  myInts.pop_back();
  myInts.push_back(25);

  // Show the contents of the vector again
  ListBox1->Items->
    Add("After deleting 10 and adding 25:");
```

```
  for( int y = 0; y < 11; y++ )
    ListBox1->Items->Add(String(myInts[y]));

  myInts.clear();
  ListBox1->Items->Add(
    "Vector example complete!");
}
//---------------------------------

void __fastcall
TForm1::ListButtonClick(TObject *Sender)
{
  list<Wisdom*> myWisdom;
  list<Wisdom*>::iterator wisIt;
  char sentence[50];
  char* myWords[] = {
    "a ", "nice ", "day ", "Have "};

  ListBox1->Clear();
  ListBox1->Items->Add("List Example");
  ListBox1->Items->Add("============");

  // Add all words to the list from the array.
  for( int x = 0; x < 4; x++ ) {
    char* wordMemory =
      new char[strlen(myWords[x]+1)];
    strcpy(wordMemory, myWords[x]);
    myWisdom.push_back(new Wisdom(wordMemory));
  }

  // Form a sentence
  ListBox1->Items->Add("My Words of wisdom:");
  *sentence = NULL;
  for(wisIt = myWisdom.begin();
      wisIt != myWisdom.end(); wisIt++)
    strcat(sentence, (*wisIt)->say());
  ListBox1->Items->Add(sentence);

  // Move "Have" to the beginning of the sentence
  wisIt = myWisdom.end();
  Wisdom* tmpWis = (*(--wisIt));
  myWisdom.remove(tmpWis);
  myWisdom.push_front(tmpWis);
  ListBox1->Items->
```

```
    Add("My Words of wisdom (corrected):");
  *sentence = NULL;

  // Rebuild the sentence again and delete the
  // Wisdom objects as we go.
  for(list<Wisdom*>::iterator li =
      myWisdom.begin();
      li != myWisdom.end(); li++) {
    strcat(sentence, (*li)->say());
    delete (*li);
  }
  ListBox1->Items->Add(sentence);
}

void __fastcall
TForm1::MapButtonClick(TObject *Sender)
{
  map<string, Person> anyBody;
  map<string, Person>::iterator persIter;

  ListBox1->Clear();
  ListBox1->Items->Add("Multimap Example");
  ListBox1->Items->Add("================");

  // Add some people to the map
  anyBody["honest"] = Person("Abe Lincoln", 98);
  anyBody["cherry tree"] =
    Person("George Washington", 100);
  anyBody["watergate"] =
    Person("Richard Nixon", 74);
  anyBody["pardon"] = Person("Gerald Ford", 76);
  anyBody["peanuts"] = Person("Jimmy Carter", 80);
  anyBody["scandal"] = Person("Bill Clinton", 92);

  // Find a couple of people and show them
  Person who = anyBody["honest"];
  ListBox1->Items->Add((who.getName()).c_str());
  who = anyBody["watergate"];
  ListBox1->Items->Add((who.getName()).c_str());

  // Delete Gerald Ford by calling erase function
  anyBody.erase("pardon");

  // The find function is a better way to look
```

```
  // for entries in the list.
  persIter = anyBody.find("pardon");
  if (persIter != anyBody.end())
    ListBox1->Items->Add((who.getName()).c_str());
  else
    ListBox1->Items->Add("No such president!");
}
```

Who is using your program?

By Mark G. Wiseman

In the last issue, you learned how to determine the version of Windows your program was running on. In this article you will apply that knowledge and learn some new techniques to help you find out who is using your program.

Fishing in the registry

Many times when you install new software, the installation program automatically fills in your name and the name of your organization in a dialog. Where does the installation program find this information? Although there is no official way to get the name and organization for the registered user of Windows, there are a couple of places in the Registry that you can look for user information.

As it turns out, the first place you can look is dependent on whether your program is running on Win9x or Windows NT. If you recall from last month you can use the VCL variable, `Win32Platform`, to determine the platform the user is running under.

The Registry may contain values under `RegisteredOwner` and `RegisteredOrganization`. Windows 9x will have these values under the key

```
HKEY_LOCAL_MACHINE\Software\
  Microsoft\Windows\CurrentVersion
```

Windows NT, on the other hand, uses the key

```
HKEY_LOCAL_MACHINE\Software\
  Microsoft\Windows NT\CurrentVersion
```

The second place to look is the same for both Windows 9x and NT. You can look for the values `DefName` and `DefCompany` under the key

```
HKEY_LOCAL_MACHINE\Software\
  Microsoft\MS Setup (ACME)\User Info
```

Your program can attempt to read these Registry values to determine the user name and company name. I say that your program can *attempt* to read these values because they may or may not exist in the Registry on a given user's machine.

Interrogating the network

If the computer is on a network your program can query the network for the user name and the computer name. This will sometimes work even if the computer is not connected to a network.

The Windows API functions `GetUserName()` and `GetComputerName()` will give you what you need. For example:

```
char buf[256];
DWORD bufSize = sizeof(buf);
GetUserName(buf, &bufSize);
String userName = buf;
bufSize = sizeof(buf);
GetComputerName(buf, &bufSize);
String computerName = buf;
```

This is a simplified example in that it does not check the return value of the `GetUserName()` and `GetComputerName()` functions. Refer to **Listing A** for more complete examples of the use of these functions.

Note that the network user name and computer name do not contain the same information as discussed in the previous section. Nevertheless, they can be used to determine who is using your program. Whether you are looking for a plain-text user name and company name or a network user name and computer name depends on the needs of your program.

Asking the user

If both of the above methods fail, you can, of course, have your program ask the user for his or her name, company, or other information you require. This could be done through a standard VCL form, and, since there is nothing special about using a VCL form to ask for user information I won't dwell on the subject further.

Conclusion

Listing A shows the source code for the example program's main form. The example demonstrates the first two methods of obtaining user information. The example's main form contains a memo that displays the user information, and the method used to obtain that information. We don't show the main form's header in order to save space.

Listing A: *Main.cpp*

```
#include <vcl.h>
#pragma hdrstop

#include <registry.hpp>

#include <memory>
using namespace std;

#include "Main.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TMainForm *MainForm;

__fastcall TMainForm::
TMainForm(TComponent* Owner) : TForm(Owner)
{
  MainMemo->Lines->Add("Owner/User Information:");
  MainMemo->Lines->Add("");

  GetRegisteredOwner(MainMemo->Lines);

  MainMemo->Lines->Add("");

  GetNetworkInfo(MainMemo->Lines);
}

void __fastcall
TMainForm::GetNetworkInfo(TStrings *lines)
{
  char buf[300];
  DWORD bufSize = sizeof(buf);

  bool success = ::GetUserName(buf, &bufSize);
  if (success)
    lines->Add(
      "Network user name = " + String(buf));
  else
    lines->Add(
     "Unable to get network user name.");

  bufSize = sizeof(buf);
```

```
  success = ::GetComputerName(buf, &bufSize);
  if (success)
    lines->Add(
      "Network computer name = " + String(buf));
  else
    lines->Add(
      "Unable to get network computer name.");
}

static char *MS1_NT = "\\Software\\"
  "Microsoft\\Windows NT\\CurrentVersion";
static char *MS1_9X = "\\Software\\"
  "Microsoft\\Windows\\CurrentVersion";
static char *MS1_NAME = "RegisteredOwner";
static char *MS1_ORG  = "RegisteredOrganization";

static char *MS2 = "\\Software\\"
  "Microsoft\\MS Setup (ACME)\\User Info";
static char *MS2_NAME = "DefName";
static char *MS2_ORG  = "DefCompany";

void __fastcall
TMainForm::GetRegisteredOwner(TStrings *lines)
{
  try {
    auto_ptr<TRegistry> registry(new TRegistry);
    registry->RootKey = HKEY_LOCAL_MACHINE;

    String key = (Win32Platform ==
      VER_PLATFORM_WIN32_NT) ? MS1_NT : MS1_9X;
    if (registry->OpenKey(key, false)) {
      String name, org;
      lines->Add(
        "Checking HKEY_LOCAL_MACHINE" + key);
      if (registry->ValueExists(MS1_NAME))
        name = registry->ReadString(MS1_NAME);
      if (registry->ValueExists(MS1_ORG))
        org = registry->ReadString(MS1_ORG);

      lines->Add(String(MS1_NAME) +" = " + name);
      lines->Add(String(MS1_ORG) + " = " + org);
    }
    else
      lines->Add("Unable to open "
```

```
        "HKEY_LOCAL_MACHINE" + key);

    lines->Add("");

    if (registry->OpenKey(MS2, false)) {
      String name, org;
      lines->Add("Checking HKEY_LOCAL_MACHINE" +
        String(MS2));

      if (registry->ValueExists(MS2_NAME))
        name = registry->ReadString(MS2_NAME);
      if (registry->ValueExists(MS2_ORG))
        org = registry->ReadString(MS2_ORG);

      lines->Add(String(MS2_NAME)+ " = " + name);
      lines->Add(String(MS2_ORG) + " = " + org);
    }
    else
      lines->Add("Unable to open "
        "HKEY_LOCAL_MACHINE" + String(MS2));
  }

  catch (...) {
    lines->Add("Stopped by an exception.");
    throw;
  }
}

void __fastcall
TMainForm::OKBtnClick(TObject *Sender)
{
  Close();
}
```

Components: build, beg, or buy?

by Kent Reisdorph

Part of the power of C++Builder is the VCL and its components. Need a split window? Drop a few components on your form. Need a file open dialog? Drop a component on your form. Components make your life easier, plain and simple. The more components you have, the more efficient you are.

The VCL, however, is far from complete. There are many programming needs that are not covered by the VCL. For those needs you will either have to write your own components (build), use shareware or freeware components (beg), or purchase commercial components (buy). The route you go depends on a number of factors. I will explain some of those factors in this article.

Build

There are a number of reasons to go the route of building components:

- You have a programming need that is not addressed by available components.
- You work for a company that insists on everything being in-house.
- You feel you cannot afford commercial components.
- You simply want to learn how to write components.

Some of these are certainly valid reasons. If you have a programming need that is not covered by any available components, you may have no choice but to write that component yourself. Component vendors typically write components that can appeal to a large audience. Anything that applies to a small audience is probably not going to be available in the third party component market.

Some companies insist that all code be produced in-house to reduce reliance on third parties. Custom components will have to be written in cases like this.

Probably the poorest reason to write your own components is because you feel you cannot afford to purchase commercial components. Granted, it depends on whether you program for a living or you are a hobbyist programmer. Except for hobbyists, it rarely pays to write a component if one is otherwise available.

I like to use TurboPower’s AsyncProfessional as an example. AsyncProfessional is a set of components

for serial communications. It includes components for basic serial communications, TAPI support (including voice modems), faxing via a fax modem and much more. At the time of this writing, the retail price of AsyncProfessional is $349.00 (US). It would easily take 10-12 man months to write the equivalent of AsyncProfessional. Do the math and you can see that it could cost from $50,000 to $100,000 to reproduce the features found in AsyncProfessional. Obviously, it does not pay to write components for full serial communications support when they can be purchased for a mere $349.00.

Writing components for the knowledge attained is certainly a viable reason to build rather than buy.

Beg

I used the word beg here simply because it fits with the alliteration used in this article's title. What I mean by this is to obtain freeware or shareware components. Most shareware and freeware components are available as Delphi components but they can probably be used in C++Builder without much additional effort.

Shareware and freeware components are certainly an option, but some care must be taken when using this type of component. There are many good freeware and shareware components available. There are many more that are not up to the quality you have come to expect from the VCL components. You may have to kiss many frogs before finding that prince.

A common problem with shareware and freeware components is documentation and support. You may find a component with online help, but many won't contain any documentation at all. Certainly it is rare to find a shareware component with printed documentation (if, in fact, one exists at all).

Installation is often a problem with shareware and freeware components. Some include pre-built packages but most are simply a collection of source files that you are expected to put into a package on your own.

Finally, consider that the hunt for a suitable freeware or shareware component may be more costly than purchasing a proven commercial component. Time is money. Consider the scenario where you spend several days finding, installing, and testing a component only to find that it doesn't work as advertised. It looked like a good deal to start with but now you have lost valuable time and must start again. I recommend that you only use shareware and freeware components that come highly recommended by your peers.

I want to stress that there are many good shareware and freeware components. I don't want to leave the impression that all components in this category are less than adequate.

Buy

Finally, you can buy commercial components. Buying commercial components has several advantages over building your own components and over shareware and freeware components. Some of those advantages are:

- Full documentation, often including printed documentation
- Technical support is almost always provided (free or fee-based)
- Professional component design and implementation
- Efficient cost vs. effort ratio
- Typically come with full source code
- Professional component installation

Most of these points are self-explanatory so I won't go over them in detail. Buying commercial components is a good solution if you can find components that fit your needs. It is efficient from a monetary standpoint (buying is much less expensive than developing). Typically commercial components come with some form of tech support. Documentation is also provided with commercial components.

One of the most compelling reasons to use commercial components is that they often come with full source code (at least those from reputable companies do). This means that you are never stuck with a set of components that becomes outdated in the unlikely event the company from which you originally purchased them is no longer available. You have the source so you can modify it as needed.

All in all, commercial components allow you to implement a particular bit of functionality with the least amount of time and money involved.

A word about Delphi components

Most of the commercial, shareware, and freeware components you find will be written in Delphi. There are a number of reasons for this. First, Delphi was available several years before C++Builder. The Delphi component vendors were already established by the time C++Builder came on the scene. As a result, the majority of the VCL third party component market has a Delphi code base. Second, Delphi components can be used in both Delphi and C++Builder. The converse is not true; components written in C++Builder cannot be used in Delphi. Obviously, it makes sense for component writers to maximize their profit potential by writing components in Delphi.

The fact that most components are written in Delphi is, for the most part, inconsequential. Usually a professional installation is implemented. The component packages are installed directly into the IDE and you can begin using the components immediately.

Earlier I said that most commercial components come with source code. In all but a few cases you will find that source code to be Delphi code. Looking at Delphi code may put you off at first, but it doesn't take long for the average C++ programmer to figure out what is happening in that code.

Conclusion

You may encounter situations where you must write your own components, and at that time you will have no choice but to write those components. Shareware and freeware components are an option, but one that comes with a degree of risk. Given the choice, I will opt to buy proven commercial components whenever possible.

Creating better data implementations

by Mark Cashman

C++Builder offers many tools for organizing database applications. Some of these are unique, and using them effectively requires an understanding that, unfortunately, usually takes several implementations to develop. A certain amount of trial and error is required to find what works and what doesn't. This process can leave your system littered with examples of poor component use, component naming, and programming logic that you typically don't take the time to clean up later. This article will help you jumpstart your database development by showing you how to apply good data implementation practices at the start.

To fully understand this article you should know C++Builder database basics, know about data modules in general terms, and be familiar with `TTable`, `TQuery`, `TDatabase` and `TField` objects.

Design goals

The design goals of a good data implementation include:

I. Keeping the implementation as similar as possible to the database design to make it easier to verify the implementation and to simplify development and maintenance.

II. Grouping related data components to help developers reuse the implementation.

III. Organizing the implementation for future extension and enhancement to allow adaptation to changing requirements.

IV. Making the implementation Database Management System (DBMS) independent to allow the application to scale between more and less powerful database systems.

V. Keeping the implementation independent of the user interface so other forms and applications can use it.

VI. Making component names clear and direct so program logic will be readable and comprehensible, thereby aiding maintenance and enhancement.

VII. Creating an efficient data implementation to ensure good performance at all scales.

C++Builder offers many tools and components to attain these goals. Those tools include:

I. **Data modules:** A specialized form that is the typical home of data components.

II. **Data components:** Provide DBMS-independent views into database tables.

III. **Data-aware user interface components**: Linked to a `TDataSource`, they are outside the data module and the data module is independent of them.

IV. **Event handlers:** The homes of programming logic to filter rows, set default values, enforce constraints, and propagate updates.

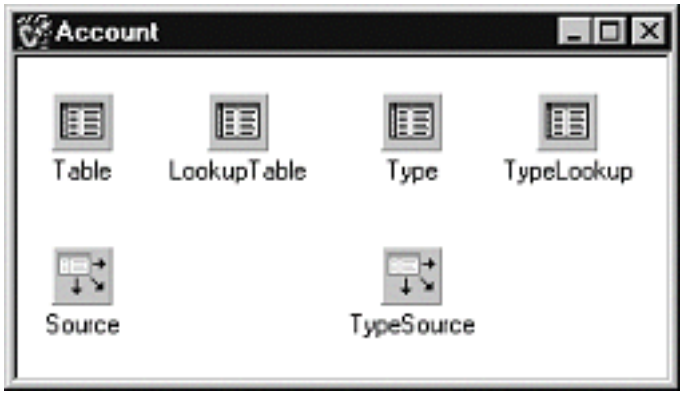
V. **Persistent fields:** Data fields, lookup fields, and calculated fields display and/or allow editing of physical and calculated data, and can also have event handling logic trigged by changes to their data values.

VI.BDE aliases: DBMS-independent references to databases.

Structuring the data implementation

Databases are often designed using Computer-Aided Software Engineering (CASE) tools. These tools support one or more object-oriented models such as Chen Entity-Relationship (ER) or Booch notation. Whether you use ER diagrams or Booch notation, the end result is a set of fundamental data objects and their interconnections. An ER diagram, for example, shows objects as entities connected by relationships. An entity represents a database table, and the connections between the entities are joins, lookup fields, or calculated fields based on foreign keys. In a C++Builder implementation, a data module represents an entity and is named according to the entity's purpose ("Account", for example). Each data module should have a primary table and a lookup table. The primary table, naturally, represents the core data for the entity. The lookup table is used for lookup fields and to perform other lookup needs from within or outside the data module. The data module also usually needs a table to explain and provide descriptive text for each coded field used by the entity. These tables, such as "Product" or "Status", are sometimes referred to as the *domain* from which the field values are drawn. A lookup table ("TypeLookup", "StatusLookup", etc.) may also be needed to support lookup fields in the primary table. **Figure A** shows a data module for an entity that represents a customer account.

Figure A

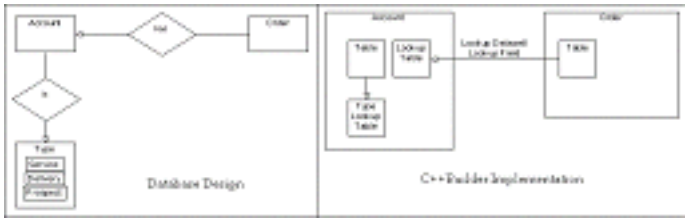


Data modules represent the database design in the implementation.

The primary and domain tables are typically linked to the user interface for editing. The tables are linked to the user interface through data sources. They usually have event handlers (`AfterPost`, `BeforeDelete`, and so on) to support referential integrity. Lookup tables may be used in the user interface, but for display or selection only.

Figure B shows how the database design and data module correspond.

Figure B



Database design and the data module directly correspond to one another.

The data module acts as a boundary for the implementation. It takes a special act to bring things across that boundary, which gives you the opportunity to ask, "Should this be brought in from another data module, or should it be in this data module?"

Remember that one of the goals of better database design is reuse. Considering that, you should never access the user interface (i.e. a control) from the data module itself. Referencing a user interface element from a data module essentially makes it impossible to reuse the data module with any other user interface. If you are tempted to do this, create your own data-aware control instead.

A well-designed structure with good clean component names aids in having a clean notation. For example, if you have implemented a logical component naming convention you can use code like this:

```
S = Account->Type->
```

```
FieldByName("DESCRIPTION")->AsString;
```

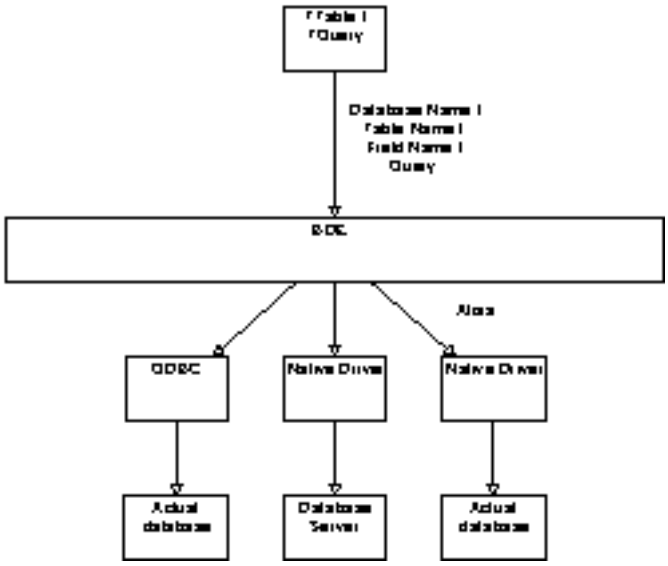
This helps keep you thinking of data components in an object-oriented fashion.

DBMS-independence and why you need it

In the bad old days, the application knew the physical layout of the data on disk. A little later, the first level of independence was born when the only thing developers needed to know was the layout of the file. Next came the database schema and the DBMS, which let the developer avoid knowing the file layout of the data, and which provided SQL and/or an API to access the database content.

With C++Builder, there is an additional layer between the developer and the data—the Borland Database Engine (BDE). The BDE uses aliases to provide access to a database. An alias describes the DBMS type (dBase, Paradox, SQL Server, and so on) and location (a database connection or a directory). The BDE is the only part of the system that knows the actual location and type of a database. The alias is then used by VCL data access components such as `TTable` and `TQuery`. In this way, the C++Builder program does not even need to know about the DBMS or even the specifics of the alias. It only needs to know the name of the `TTable` or `TQuery` component used to access the database. **Figure C** shows the relationship between the VCL data access components, the BDE, and the DBMS.

Figure C



The BDE is the link between the user program and the DBMS.

The BDE also offers an amazing capability—the heterogeneous join. A heterogeneous join allows a SQL `SELECT` statement to join tables across DBMS. For instance, a dBase table can be joined to a Sybase table simply by properly specifying the alias. This feature allows you to work with legacy applications or foreign databases as if they were a natural part of your system.

Why is all this abstraction good for the developer? If done right, a C++Builder program can scale from a simple desktop DBMS like dBase or Paradox all the way to large client/server databases like Sybase and Oracle without any programming changes.

Writing DBMS-independent applications

Attaining DBMS independence requires some simple discipline on your part. By following a few simple rules when you design your application you can attain DBMS-independence from the start:

I. Whatever DBMS you use, keep table and field names at the lowest common denominator. Use 8.3 uppercase table and database file names, and 15 character upper case field names without spaces, numbers, or special characters.

II. Establish separate BDE aliases for any tables that you may want to reside in separate locations (on the client vs. on the server, for example). This gives you flexibility in distributing your database.

III. Avoid references to database or table names in your code. Keep those names restricted to data components in the data module. If you must reference them in your code, use notation like `Table->DatabaseName` rather than "MyAlias." For maximum independence, place a `TDatabase` object in each data module for each alias, and instead of having tables refer to the alias, have them refer to the `TDatabase` object. A change to each of these `TDatabase` objects automatically alters the database used by the tables and queries.

IV. Use persistent fields. See the side bar entitled *Using persistent fields* for more information on persistent fields.

With this strategy, you have a very clear data implementation which is separated from the user interface and from other data modules, is highly reusable, is tightly related to the database design, is efficient, and which provides clean notation for your program code. What more could you want?

Conclusion

Designing a database with these features in mind has several advantages, not the lease of which is the ability to easily move your application to a different DBMS. To retarget your data implementation to a new DBMS, all you should need to do is:

I. Create your tables in the new DBMS, using the same table and field names.

II. Retarget the alias to the new DBMS driver and database location.

III. Run your program.

If, for some reason, you are forced to change table names, you only have to change them in the referencing objects in the data module (tables and queries). Field names only have to be changed in the persistent field objects and in SQL statements.

Finally, you can use form inheritance to create variations based on a core data module. To extend the base data module's functionality, you can add some fields, add tables, or override or supplement your `TTable`, `TQuery` or `TField` event handlers. The possibilities are endless, and largely unexplored. Try something new!

Custom buttons, part I

by Damon Chandler

In this series of articles, I'll show you how to create a `TButton`-descendant component that will handle all of your button needs. In this first installment, I'll discuss how to create a generic owner-drawn button and how to use this technique to extend the `TButton` class to accept a color specification.

Owner-drawn buttons

Like many standard control classes, the Windows `BUTTON` control provides a so-called "owner-drawn" style, which allows you to customize the appearance of a button. This style is specified by adding `BS_OWNERDRAW` to the other styles that you'd normally pass as the `dwStyle` parameter to the `CreateWindowEx()` API function. Let's see how this is done by working through an example.

Figure A

A form that contains an owner-drawn button.

Creating an owner-drawn button from scratch

Figure A depicts a form that contains one child control: an owner-drawn button that's created directly via the `CreateWindowEx()` API function:

```
// in Form1's header...
private:
  HWND hODButton_;

// in Form1's source...
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
```

```
  hODButton_ = CreateWindowEx(
    0, "BUTTON", "ODButton1",
    WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
    10, 10, 75, 25,
    Handle, NULL, HInstance, NULL
    );
}
```

By specifying the `BS_OWNERDRAW` style, you're effectively telling the button that you want to take over its rendering process. This is done—as it is with other owner-drawn controls—by handling the `WM_DRAWITEM` message that's sent by the button to its parent window whenever the button needs to be drawn. For our example, because `Form1` is the parent of the button, we need to handle the `WM_DRAWITEM` that's sent to `Form1`. We can do this by augmenting the `TForm::Dispatch()` method (via the message-mapping macros), like so:

```
// in Form1's header...
private:
  MESSAGE void __fastcall WMDrawItem(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    WM_DRAWITEM, TMessage, WMDrawItem)
END_MESSAGE_MAP(TForm)

// in Form1's source...
void __fastcall TForm1::
  WMDrawItem(TMessage& Msg)
{
  // grab a pointer to the DRAWITEMSTRUCT
  const DRAWITEMSTRUCT* pDrawItem =
    reinterpret_cast<DRAWITEMSTRUCT*>
      (Msg.LParam);

  // if the message is from our button...
  if (pDrawItem->hwndItem == hODButton_)
  {
    // grab a handle to the target DC
    const HDC hButtonDC = pDrawItem->hDC;

    // extract the target rectangle
    RECT RButton = pDrawItem->rcItem;
```

```
// is the button pushed?
const bool is_pushed =
  pDrawItem->itemState & ODS_SELECTED;
// does the button have focus?
const bool is_focused =
  pDrawItem->itemState & ODS_FOCUS;

// set the rendering flags to draw
// a regular push button, and adjust
// the rendering flags according to
// the button's current state
unsigned int flags = DFCS_BUTTONPUSH;
if (is_pushed) flags |= DFCS_PUSHED;

// render the push button by using
// the DrawFrameControl() function
DrawFrameControl(
  hButtonDC, &RButton,
  DFC_BUTTON, flags
  );

// render the button's text...
RECT RButtonText = RButton;
if (is_pushed)
{
  OffsetRect(&RButtonText, 1, 1);
}
DrawText(
  hButtonDC, "ODButton1\0", -1,
  &RButtonText, DT_CENTER |
  DT_VCENTER | DT_SINGLELINE
  );

// indicate keyboard focus (here via
// an etched edge rather than the
// traditional focus rectangle)...
if (is_focused)
{
  InflateRect(&RButton, -4, -4);
  DrawEdge(
    hButtonDC, &RButton,
    EDGE_ETCHED, BF_RECT
    );
}
```

```
    // return TRUE for this message
    Msg.Result = TRUE;
  }
  // otherwise, pass the message along
  else TForm::Dispatch(&Msg);
}
```

You can see from this code that the first step is to grab a pointer to a `DRAWITEMSTRUCT` structure, which is sent with the `WM_DRAWITEM` message via the `LParam` data member. This structure holds all of the crucial drawing-related information such as the target device context (in the `hDC` data member), the target rectangle (in the `rcItem` data member), and the current state of the button (in the `itemState` data member). With this information, you'll know where to draw (`hDC`), what area to draw (`rcItem`), and in what state (`itemState`) you should draw the button. In fact, how you draw the button is up to you. As depicted in **Figure A**, this code will render a standard push-button with an etched selection rectangle. Note that when the button is pushed, the `itemState` data member will contain the `ODS_SELECTED` bit. Likewise, when the button has keyboard focus, `itemState` will contain the `ODS_FOCUS` bit. And, when the button is disabled, `itemState` will contain the `ODS_DISABLED` bit.

Unfortunately, this example isn't too useful because most of us never call the `CreateWindowEx()` function directly. We could create a `TWinControl` descendant class to handle this work, but because the `TButton` class is designed specifically for buttons, let's start there. We'll still need to specify the `BS_OWNERDRAW` style to create an owner-draw button, but we can do this from within the `CreateParams()` method of our `TButton` descendant class.

The TColorButton component

I've just shown you how to create a simple owner-drawn button. Let's now work on wrapping this functionality into a reusable VCL component, which I'll call `TColorButton`. The declaration of this class is provided in **Listing A**.

Notice that the `TColorButton` class introduces three published properties: `Color`, `ColorLo`, and `ColorHi`. As you can guess, the `Color` property will be used to specify the color of the button's face. This property is maintained by using the `Color` property of the button's `Brush`. The `ColorLo` and `ColorHi` properties will be used to specify the colors of the button's shadow and highlight, respectively. These latter two properties are maintained via the private `ColorLo_` and `ColorHi_` members, which we can initialize in the class constructor, like so:

```
__fastcall TColorButton::
  TColorButton(TComponent* Owner)
    : TButton(Owner),
    ColorLo_(clBtnShadow),
```

```
    ColorHi_(clBtnHighlight),
    Canvas_(new TCanvas(),
    draw_as_default_(false)
{
}
```

Note that the private `Canvas_` member is also initialized in the constructor. Later, I'll show you how to use this `TCanvas` object to ease the drawing process. The constructor is also used to initialize the `draw_as_default_` member, whose role I'll discuss shortly.

Specifying the BS_OWNERDRAW style

As I mentioned earlier, we'll use the `CreateParams()` method to specify the `BS_OWNERDRAW` style. Here's the code for that method:

```
void __fastcall TColorButton::
  CreateParams(TCreateParams& Params)
{
  TButton::CreateParams(Params);
  Params.Style |= BS_OWNERDRAW;
}
```

By adding `BS_OWNERDRAW` to the `TCreateParams::Style` data member, this code effectively instructs the `TButton` class (and thus the `TWinControl` class) to pass `BS_OWNERDRAW` to the `CreateWindowEx()` function, which is later called from within the `TWinControl::CreateWnd()` method. Notice, though (from **Listing A**), that the `TColorButton` class also overrides the `TButton::SetButtonStyle()` method. This step is needed because the `TButton` class uses `SetButtonStyle()` to change the button's style to `BS_PUSHBUTTON` or `BS_DEFPUSHBUTTON` depending on the value of the `ADefault` parameter. In our case, we want to preserve the `BS_OWNERDRAW` specification, so we need to override the `SetButtonStyle()` method:

```
void __fastcall TColorButton::
  SetButtonStyle(bool ADefault)
{
  if (draw_as_default_ != ADefault)
  {
    draw_as_default_ = ADefault;
    InvalidateRect(Handle, NULL, FALSE);
  }
}
```

This code updates the value of the private `draw_as_default_` member according to the `ADefault` parameter. We'll later use this member to determine if the button should be drawn as the default button (i.e., with a thick black border).

That takes care of creating an owner-drawn button, but where's the actual drawing code? Well, this is where the `TColorButton::CNDrawItem()` method comes into play.

Handling the CN_DRAWITEM message

Recall that a normal owner-drawn button sends its parent window the `WM_DRAWITEM` message, which simply instructs the parent window that the button needs to be drawn or redrawn. This means that if you place a `TColorButton` object directly on a form, you can handle the `WM_DRAWITEM` message by tapping into the form's window procedure (just as we did before). This isn't too hard for a form—you simply augment the `TForm::Dispatch()` member function (by using the message-mapping macros). What do you do, though, if you want to place your `TColorButton` object on, say, a panel? Do you subclass the panel? Or, do you create a new `TPanel` descendant class and augment's *its* `Dispatch()` member function? Either of these options results in too much work.

Fortunately, when any `TWinControl` descendant receives the `WM_DRAWITEM` message, it will forward a copy of the message—called `CN_DRAWITEM`—back to the button itself. This way, you can handle the `CN_DRAWITEM` message from within the button's window procedure without worrying about which control the button is placed on. From the declaration of the `TColorButton` class, you can see that this message is mapped to the `CNDrawItem()` method. It's from within this method that you can draw the button in a new, customized fashion. Here, we'll use the `Color`, `ColorLo`, and `ColorHi` specifications—along with our `TCanvas` object—to render a colored button:

```
void __fastcall TColorButton::
  CNDrawItem(TMessage& Msg)
{
  // grab pointer to the DRAWITEMSTRUCT
  const DRAWITEMSTRUCT* pDrawItem =
    reinterpret_cast<DRAWITEMSTRUCT*>
      (Msg.LParam);

  // store the current state of
  // the target DC
  SaveDC(pDrawItem->hDC);
  // bind Canvas_ to the target DC
  Canvas_->Handle = pDrawItem->hDC;
  try
  {
    // extract the state flags...
```

```
    TOwnerDrawState state;
    // if the button has keyboard focus
    if (pDrawItem->itemState & ODS_FOCUS)
    {
      state = state << odFocused;
    }
    // if the button is pushed
    if (pDrawItem->itemState &
        ODS_SELECTED)
    {
      state = state << odSelected;
    }
    // if the button is disabled
    if (pDrawItem->itemState &
        ODS_DISABLED)
    {
      state = state << odDisabled;
    }

    // draw the button's face
    DoDrawButtonFace(state);

    // draw the button's text
    DoDrawButtonText(state);
  }
  catch (...)
  {
    // clean up
    Canvas_->Handle = NULL;
    RestoreDC(pDrawItem->hDC, -1);
  }
  // clean up
  Canvas_->Handle = NULL;
  RestoreDC(pDrawItem->hDC, -1);

  // reply TRUE
  Msg.Result = TRUE;
}
```

You'll notice that this definition of the `TColorButton::CNDrawItem()` method is somewhat similar to the previous definition of the `TForm1::WMDrawItem()` method. Here, instead of actually drawing the button from within the `CNDrawItem()` method, the `TColorButton` class fills a `TOwnerDrawState`-type variable, and then punts the work to its `DoDrawButtonFace()` and

`DoDrawButtonText()` methods.

Drawing the button’s face

We’ll render the button’s face from within the `TColorButton::DoDrawButtonFace()` method. Here’s the code for that method:

```
void __fastcall TColorButton::
  DoDrawButtonFace(
    const TOwnerDrawState& state
    )
{
  // draw a colored button...
  Canvas_->Brush = Brush;
  TRect RClient = ClientRect;

  // if the button is the default button
  // or has keyboard focus...
  if (draw_as_default_ ||
      state.Contains(odFocused))
  {
    Canvas_->Pen->Color = clWindowFrame;
    Canvas_->Rectangle(
       RClient.Left, RClient.Top,
       RClient.Right, RClient.Bottom
       );
    InflateRect(
       reinterpret_cast<PRECT>(&RClient),
       -1, -1
       );
  }

  // if the button is pushed...
  if (state.Contains(odSelected))
  {
    Canvas_->Pen->Color = ColorLo_;
    Canvas_->Rectangle(
       RClient.Left, RClient.Top,
       RClient.Right, RClient.Bottom
       );
  }

  // if the button isn't pushed...
```

```
  else
  {
    Canvas_->FillRect(RClient);
    Frame3D(
      Canvas_.get(), RClient,
      ColorHi_, clWindowFrame, 1
      );

    POINT P[] = {
      {1, RClient.Bottom - 1},
      {RClient.Right - 1,
       RClient.Bottom - 1},
      {RClient.Right - 1,
       RClient.Top - 1}
      };
    Canvas_->Pen->Color = ColorLo_;
    Canvas_->Polyline(
       reinterpret_cast<TPoint*>(P), 2
       );
  }
}
```

There's nothing special about this code—I simply took some screenshots of a button in its various states, examined its appearance, and then worked through the necessary `TCanvas` methods. You might be wondering why I didn't use the `DrawFrameControl()` API function. Unfortunately, that function will always render a button in its default color (`clBtnFace`).

Drawing the button's caption

The next task is to render the button's caption. We'll do this from within the `TColorButton::DoDrawButtonText()` member function, like so:

```
void __fastcall TColorButton::
  DoDrawButtonText(
    const TOwnerDrawState& state)
{
  if (Caption.Length() == 0) return;

  RECT RText = {0, 0, Width, Height};
  Canvas_->Font = Font;
  Canvas_->Brush = Brush;
  SetBkMode(
```

```
  Canvas_->Handle, TRANSPARENT
  );

// if the button is pushed...
if (state.Contains(odSelected))
{
   // offset the caption
   OffsetRect(&RText, 1, 1);
}
// if the button is disabled...
if (!Enabled ||
    state.Contains(odDisabled))
{

  // render the caption
  // in a disabled fashion
  OffsetRect(&RText, 1, 1);
  Canvas_->Font->Color = ColorHi_;
  DrawText(
    Canvas_->Handle, Caption.c_str(),
    -1, &RText, DT_CENTER |
    DT_VCENTER | DT_SINGLELINE
    );
  OffsetRect(&RText, -1, -1);
  Canvas_->Font->Color = ColorLo_;
}

// render the caption
DrawText(
  Canvas_->Handle, Caption.c_str(), -1,
  &RText, DT_CENTER | DT_VCENTER |
  DT_SINGLELINE
  );

// if the button has keyboard focus...
if (state.Contains(odFocused))
{
  // render the selection rectangle
  TRect RFocus = ClientRect;
  InflateRect(
    reinterpret_cast<PRECT>(&RFocus),
    -4, -4
    );
  Canvas_->DrawFocusRect(RFocus);
```

```
  }
}
```

The bulk of the work of drawing the button's caption is handled by the `DrawText()` API function. This function is particularly handy because it will center the text—both vertically and horizontally—within the rectangle that you specify via the fourth (`lpRect`) parameter. Notice that when the button is disabled, the `DrawText()` function is called twice to achieve the chiseled effect. Also, when the button has keyboard focus, we use the `TCanvas::DrawFocusRect()` method to render the selected rectangle.

That takes care of drawing the button's face, text, and selection rectangle. Let's now focus on the rest of `TColorButton`'s methods

Finishing up

As you might have guessed, the `GetColor()`, `SetColor()`, `SetColorLo()`, and `SetColorHi()` methods provide access to the `Color`, `ColorLo`, and `ColorHi` properties. Here's how these methods are defined:

```
TColor __fastcall TColorButton::
  GetColor()
{
  return Brush->Color;
}

void __fastcall TColorButton::
  SetColor(TColor Value)
{
  if (Brush->Color != Value)
  {
    Brush->Color = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}

void __fastcall TColorButton::
  SetColorLo(TColor Value)
{
  if (ColorLo_ != Value)
  {
    ColorLo_ = Value;
    InvalidateRect(Handle, NULL, TRUE);
```

```
  }
}

void __fastcall TColorButton::
  SetColorHi(TColor Value)
{
  if (ColorHi_ != Value)
  {
    ColorHi_ = Value;
    InvalidateRect(Handle, NULL, TRUE);
  }
}
```

The `CMFontChanged()` and `CMEnabledChanged()` methods are called when the button receives the `CM_FONTCHANGED` and `CM_ENABLEDCHANDED` messages, respectively. These are VCL-specific messages that are sent to the button when its `Font` or `Enabled` properties have changed (either at design time or at run time). When this happens, we need to instruct the button to repaint itself:

```
void __fastcall TColorButton::
  CMFontChanged(TMessage& Msg)
{
  TButton::Dispatch(&Msg);
  InvalidateRect(Handle, NULL, TRUE);
}

void __fastcall TColorButton::
  CMEnabledChanged(TMessage& Msg)
{
  TButton::Dispatch(&Msg);
  InvalidateRect(Handle, NULL, TRUE);
}
```

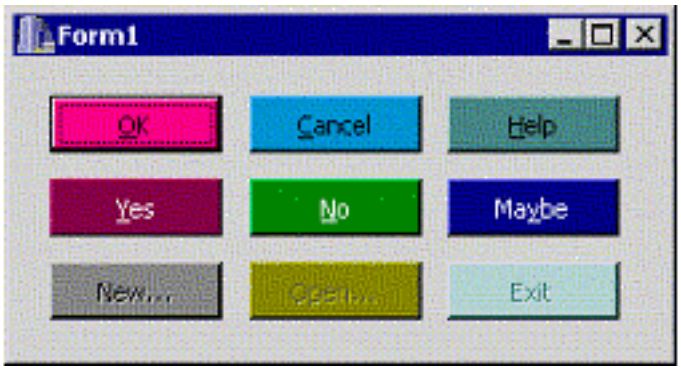
There's one last method: `WMLButtonDblClk()`. You can see from **Listing A** that this method is called whenever the button receives the `WM_LBUTTONDBLCLK` message; this message is sent whenever the button is double-clicked. Because a normal button (i.e., a non-owner-drawn push-button) processes this message as if it were a `WM_LBUTTONDOWN` message, we'll need to do the same (otherwise, the button will react very slowly when it's clicked rapidly). Here's the definition of the `WMLButtonDblClk()` method:

```
void __fastcall TColorButton::
  WMLButtonDblClk(TMessage& Msg)
{
  SNDMSG(
```

```
    Handle, WM_LBUTTONDOWN,
    Msg.WParam, Msg.LParam
    );
}
```

Figure B shows a form containing several `TColorButtons`.

Figure B



Some TColorButton objects.

Conclusion

I've shown you how to create an owner-drawn button and how to use this style to create a colored-button component. Next month, I'll show you how to extend this technique even further. I'll discuss how to add a glyph to the button, and how to define the button's face by using a custom bitmap (which is the first step toward creating an application that supports skins). For now, experiment with the code for the `TColorButton` component; it's available for download from **www.residorph.com.**

Listing A: ***Declaration of the TColorButton class***

```
#include <memory>
class TColorButton : public TButton
{
public:
  __fastcall TColorButton(TComponent* Owner);

__published:
  __property TColor Color =
    {read = GetColor, write = SetColor};
  __property TColor ColorLo =
    {read = ColorLo_, write = SetColorLo,
```

```
     default = clBtnShadow};
  __property TColor ColorHi =
    {read = ColorHi_, write = SetColorHi,
     default = clBtnHighlight};

protected:
  // inherited member functions
  virtual void __fastcall CreateParams(
    TCreateParams& Params);
  virtual void __fastcall SetButtonStyle(
    bool ADefault);

  // introduced member functions
  virtual void __fastcall DoDrawButtonFace(
    const TOwnerDrawState& state);
  virtual void __fastcall DoDrawButtonText(
    const TOwnerDrawState& state);

private:
  TColor ColorLo_;
  TColor ColorHi_;
  std::auto_ptr<TCanvas> Canvas_;
  bool draw_as_default_;

  TColor __fastcall GetColor();
  void __fastcall SetColor(TColor Value);
  void __fastcall SetColorLo(TColor Value);
  void __fastcall SetColorHi(TColor Value);

  MESSAGE void __fastcall CNDrawItem(
    TMessage& Msg);
  MESSAGE void __fastcall WMLButtonDblClk(
    TMessage& Msg);
  MESSAGE void __fastcall CMFontChanged(
    TMessage& Msg);
  MESSAGE void __fastcall CMEnabledChanged(
    TMessage& Msg);

public:
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    CN_DRAWITEM, TMessage, CNDrawItem)
  MESSAGE_HANDLER(
    WM_LBUTTONDBLCLK, TMessage, WMLButtonDblClk)
```

```
  MESSAGE_HANDLER(
    CM_FONTCHANGED, TMessage, CMFontChanged)
  MESSAGE_HANDLER(
    CM_ENABLEDCHANGED, TMessage, CMEnabledChanged)
END_MESSAGE_MAP(TButton)
};
```

Determining the OS version

By Mark G. Wiseman

If you are programming with C++Builder, then you are programming for Microsoft Windows. In fact, since C++Builder will only generate 32-bit code, you are programming for 32-bit Windows specifically. Is that all you need to know?

There are several versions of 32-bit Windows, including Windows NT and Windows 95/98. With each of these different versions there are similar but different APIs, registry entries and file structures. Sometimes it *is* important for us to be able to determine which 32-bit version of Windows our program is running on.

There are two methods can be used to get operating system version information; the Windows API or the VCL. I will discuss the both the API and the VCL methods of obtaining operating system information in the following sections.

The example program for this article is called OSVERSION. We don't show the code for the entire project here, but you can download it from our Web site at http://www.bridgespublishing.com. The program gets the operating system information and displays it in a Memo component.

Using the Windows API

The example program's `GetOSVersionAPI()` function uses the Windows API function `GetVersionInfoEx()` to obtain the version information. Here is the code for the `GetOSVersionAPI()` function:

```
void __fastcall TMainForm::GetOSVersionAPI(TStrings *lines)
{
  OSVERSIONINFO info;
  ZeroMemory(&info, sizeof(OSVERSIONINFO));
  info.dwOSVersionInfoSize =
    sizeof(OSVERSIONINFO);
  bool success = GetVersionEx(&info);
  if (!success)
  {
    lines->Add("Unable to get OS Version Info");
    return;
  }
```

```
  lines->Add("Major Version:  " +
    String(info.dwMajorVersion));
  lines->Add("Minor Version:  " +
    String(info.dwMinorVersion));

  DWORD buildNumber = info.dwBuildNumber;
  if (info.dwPlatformId ==
        VER_PLATFORM_WIN32_WINDOWS)
    buildNumber = LOWORD(buildNumber);
  lines->Add("Build Number:   " +
    String(buildNumber));

  String platform = "Unknown";
  if (info.dwPlatformId == VER_PLATFORM_WIN32s)
     platform = "Win32s";
  else if (info.dwPlatformId ==
             VER_PLATFORM_WIN32_WINDOWS)
  {
    if ((info.dwMajorVersion > 4) ||
         ((info.dwMajorVersion == 4) &&
         (info.dwMinorVersion > 0)))
      platform = "Windows 98";
    else
      platform = "windows 95";
  }
  else if (info.dwPlatformId ==
             VER_PLATFORM_WIN32_NT)
    platform = "Windows NT";
  lines->Add("Platform:       " + platform);

  String csdVersion = info.szCSDVersion;
  if (csdVersion.IsEmpty() &&
     info.dwPlatformId == VER_PLATFORM_WIN32_NT)
    csdVersion = "No Service Pack Installed";
  lines->Add("CSDVersion:     " + csdVersion);
}
```

As you can see, the first parameter of `GetVersionInfoEx()` is a pointer to an `OSVERSIONINFO` structure. The members of this structure are listed in **Table A**.

Table A: *OSVERSIONINFO members*

Member	Description	
dwOSVersionInfoSize	This size of the structure in bytes. You must set this member before calling GetVersionInfoEx().	
dwMajorVersion	The major version number of Windows operating system.	
dwMinorVersion	The minor version number of Windows operating system.	
dwBuildNumber	The build number of the operating system for Windows NT. For Windows 9x, the low order word of dwBuildNumber is the build number.	
dwPlatformId	A value representing the Windows platform.	
	Value	Platform
	VER_PLATFORM_WIN32s	Win32s on Windows 3.1.
	VER_PLATFORM_WIN32_WINDOWS	Windows 9x. If the dwMinorVersion is zero, then Windows 95, otherwise Windows 98.
	VER_PLATFORM_WIN32_NT	Windows NT.
szCSDVersion	A null-terminated string that indicates the latest installed Service Pack on Windows NT. If the string is empty, no Service Pack has been installed. May contain arbitrary information for Windows 9x.	

In the `GetOSVersionAPI()` function I create an instance of `OSVERSIONINFO`, zero the memory occupied by the structure and set the size of the structure in the `dwOSVersionInfoSize` member.

Next I call `GetVersionInfoEx()` passing it a pointer to the `OSVERSIONINFO` structure and the

size of that structure. `GetVersionInfoEx()` returns a non-zero value if it is successful. I test for this and if `GetVersionInfoEx()` fails, I display a message to the user and exit the function. If `GetVersionInfoEx()` succeeds, I analyze the values of the `OSVERSIONINFO` data members and post the findings to a `TMemo` component on the main form.

Using the VCL

If you are using the VCL in our program, then C++Builder has done some of the work for you. There are a series of variables, which are global in the `Sysutils` namespace and correspond directly to the data members of the `OSVERSIONINFO` structure. Those variables and the corresponding `OSVERSIONINFO` data member are listed in **Table B**.

Table B: *VCL variables*

VCL Variable	OSVERSIONINFO Member
Win32MajorVersion	dwMajorVersion
Win32MinorVersion	dwMinorVersion
Win32BuildNumber	dwBuildNumber
Win32Platform	dwPlatformId
Win32CSDVersion	dwCSDVersion

The example program's `GetOSVersionVCL()` function uses these variables instead of `OSVERSIONINFO` and `GetVersionEx()`, but otherwise is identical to `GetOSVersionAPI()`. Here is the code for the `GetOSVersionVCL()` function:

```
void __fastcall TMainForm::GetOSVersionVCL(TStrings *lines)
{
  lines->Add("Major Version:  " +
    String(Win32MajorVersion));
  lines->Add("Minor Version:  " +
    String(Win32MinorVersion));

  int buildNumber = Win32BuildNumber;
  if (Win32Platform ==
        VER_PLATFORM_WIN32_WINDOWS)
    buildNumber = LOWORD(buildNumber);
  lines->Add("Build Number:   " +
    String(buildNumber));

  String platform = "Unknown";
  if (Win32Platform == VER_PLATFORM_WIN32s)
    platform = "Win32s";
  else if (Win32Platform ==
             VER_PLATFORM_WIN32_WINDOWS)
  {
    if ((Win32MajorVersion > 4) ||
         ((Win32MajorVersion == 4) &&
         (Win32MinorVersion > 0)))
      platform = "Windows 98";
    else
      platform = "windows 95";
  }
  else if (Win32Platform ==
             VER_PLATFORM_WIN32_NT)
    platform = "Windows NT";
  lines->Add("Platform:       " + platform);

  if (Win32CSDVersion.IsEmpty() &&
        Win32Platform == VER_PLATFORM_WIN32_NT)
    lines->Add("CSDVersion:"
      "     No Service Pack Installed");
  else
    lines->Add("CSDVersion:     " +
      Win32CSDVersion);
}
```

Don’t underestimate how much coding these VCL variables can save you. If you need to call one function on Windows NT and another function on Windows 9x, simply use the `Win32Platform`

variable. Nothing could be easier.

Do you need a TDatabase?

by Kent Reisdorph

Database applications come in many shapes and sizes. Some are as simple as a mailing list utilizing just a single table. Others are monolithic client/server applications that contain dozens of tables, views, triggers, and stored procedures.

Most C++Builder database programmers are familiar with the data access components such as `TTable`, `TQuery`, and `TDataSource`. Many, however, have never used the `TDatabase` component in their applications. One reason many users don't use `TDatabase` is that database applications are not required to explicitly use a `TDatabase`. If no `TDatabase` is specified for the application, the VCL uses a temporary `TDatabase` with default values.

The `TDatabase` component provides these primary features:

I. The ability to quickly switch between databases during development

II. Database login control

III. Transaction control

IV. Database connection control

This article will explain the advantages of these `TDatabase` features and how to implement them in your applications.

Easy database switching

Before I explain how `TDatabase` can be used to switch between databases, let me explain now `TDatabase` is used with other data access components. First you drop a `TDatabase` on your form and set the `AliasName` property to an existing BDE alias. Next, you provide a name for the `DatabaseName` property. The name can be any name you choose. I typically use something like "MainDB." Next, you set the `DatabaseName` property of all your data access components (`TTable`, `TQuery`, `TStoredProc`, etc.) to the name you gave the `TDatabase`'s `DatabaseName` property. Now all your data access components go through the `TDatabase` rather than going directly to the BDE.

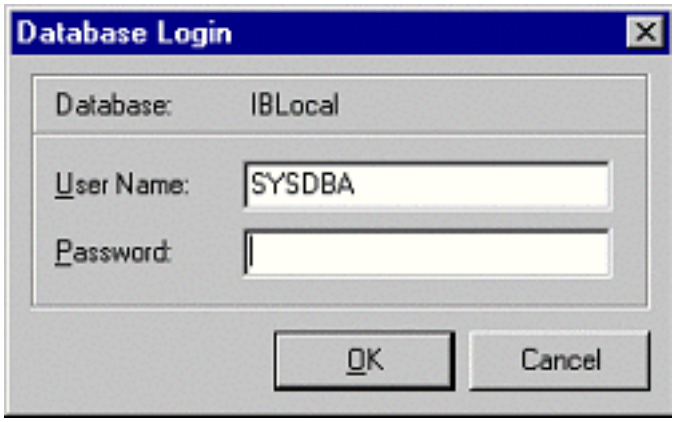
Now consider a situation where you have developed a database application for in-house use and have deployed that application. Development is, of course, a never-ending task. Therefore, you will probably continue to work on the application. You probably don't want to develop and test against the live

database. Instead, you would likely work with a "test" database. You would have one BDE alias for the live database and another alias for the test database. Provided that you have properly set up all your data access components to go through the `TDatabase` component, switching between the live database and the test database is as simple as changing the `AliasName` property of the `TDatabase`. This makes it very easy to safely continue development of your application without putting the live database at risk. The next time you deploy your application, you simply change the `TDatabase` alias name back to the alias for the live database and recompile.

Database login control

Flat-file databases—such as those using Paradox or dBase tables—don't require a user name or password in order to access the database. Full-scale client/server databases, however, generally require a user name and password in order to connect to the server. If your application uses the default `TDatabase` object, the BDE will pop up a dialog the first time you attempt to connect to the database, as shown in **Figure A**.

Figure A

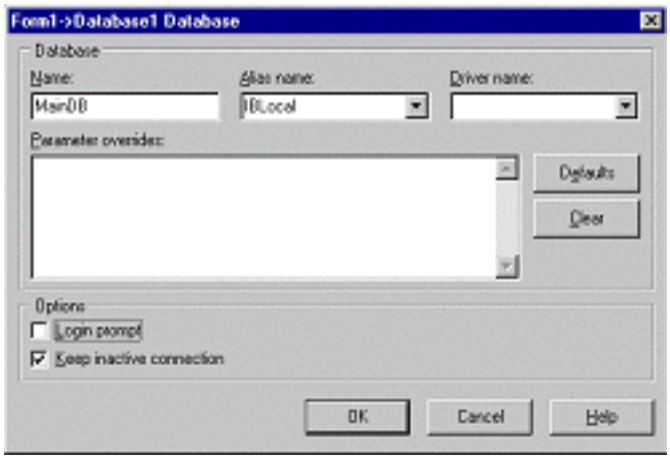


The default database login dialog is shown when you attempt to connect to a client/server database.

For some applications the default login dialog will suffice. Other applications require a specialized login dialog or may not wish to show a login dialog at all. For those applications, a `TDatabase` component is needed in order to control login.

Login control is facilitated in one of two ways. One way is via the `TDatabase` component editor. If you aren't concerned about security you can simply double-click the `TDatabase` component on your form to invoke the component editor. The component editor is shown in **Figure B**. Uncheck the *Login prompt* check box and you will never again see a login prompt. This is especially convenient during development, although I wouldn't necessarily recommend it for your release builds.

Figure B



The TDatabase component editor allows you to set login and connection parameters.

The other way to control login is via the `TDatabase`'s `OnLogin` event. The event handler for the `OnLogin` event includes a parameter called `LoginParams`. `LoginParams` is a `TStringList` that expects information structured as follows:

```
USERNAME=SYSDBA
PASSWORD=masterkey
```

Given that, the `OnLogin` event handler used to log into Local Interbase would look like this:

```
void __fastcall TForm1::Database1Login(
  TDatabase *Database,
  TStrings *LoginParams)
{
  LoginParams->Values["UserName"]
    = "SYSDBA";
  LoginParams->Values["Password"]
    = "masterkey";
}
```

The `OnLogin` event gives you the opportunity to create your own login dialog or to extract login information from some other source, such as the Registry.

Keep in mind that `OnLogin` will be fired every time your application needs to connect to the database server. Obviously this will happen the first time your application needs to access the database. It will also occur if the connection to the server was dropped for any reason. I'll address persistent connections in the section, "Database connection control." The point is that when using non-persistent connections, login can occur more frequently than you expect and you should be prepared to respond to the `OnLogin` event at any time.

Transaction control

The BDE provides support for database transactions. A transaction may consist of a single update to the database or may consist of multiple updates. The advantages to transactions are:

I. All actions against the database are applied at one time

II. Connection or hardware problems don't result in lost data

III. Control over database access in multi-user environments

IV. Transactions can be committed (applied) or rolled back (canceled)

V. Reduced network traffic

Transactions are initiated with the `StartTransaction()` method, followed by one or more database actions such as running a query or updating a table. To apply the actions, call the `Commit()` method. To cancel the actions, call the `Rollback()` method. For example:

```
Database1->StartTransaction();
try {
  Query1->Open();
}
catch (EDBEngineError&) {
  // handle the exception
  Database1->Rollback();
  return;
}
Database1->Commit();
```

Naturally, this is only a simple example. Transaction control is a subject that requires a full article to cover adequately. It is important to understand, though, that once you have started a transaction you must call either `Commit()` or `Rollback()` to end the transaction. If you fail to call one of these functions, an exception will be raised the next time you attempt to access the database.

As with much of the VCL, you aren't required to use transactions. If you do not explicitly create a transaction the VCL will create a temporary transaction on your behalf. Be aware, though, that the VCL will create a transaction for each and every database action, resulting in more network traffic and higher load on the database server.

Database connection control

The connection to the database server is controlled through `TDatabase`'s `KeepConnection` property. If `KeepConnection` is `true` then the database connection is maintained regardless of what happens within the application (barring catastrophic events, that is). If `KeepConnection` is `false`, the database connection state is handled by the BDE. The BDE will establish a connection when the application accesses the database and will close that connection when there are no active datasets. Keep in mind that the BDE must log onto the database server each time a connection is required. The process of logging onto the server takes time so if your application accesses the database frequently you will probably want to set `KeepConnection` to `true`. For full connection control, set `KeepConnected` to `false` and call the `Open()` and `Close()` methods as needed. Alternatively you can modify the `Connected` property to open and close the connection.

Conclusion

`TDatabase` is a very useful component and it pays to be familiar with its features. I particularly like the feature that allows me to switch between databases with just a few mouse clicks. Another major feature of `TDatabase` is transaction control. Any serious client/server application will likely use transactions. If you haven't yet experimented with `TDatabase` I encourage you to spend some time investigating this component.

Drawing a selection rectangle

by Kent Reisdorph

Many graphical applications allow the user to select one or more objects by dragging a bounding rectangle around the objects. For example, the C++Builder IDE's Form Designer uses this technique to allow you to select components on the form.

Drawing a bounding rectangle is fairly easy but there are a couple of subtleties that it pays to be aware of. This article will explain how to draw a selection rectangle and the various ways you can go about it.

Overview

In order to draw a bounding rectangle you must respond to the form's `OnMouseDown`, `OnMouseMove`, and `OnMouseUp` events. Drawing a selection rectangle requires the following steps:

I. Set a flag in the `OnMouseDown` event handler to indicate dragging is taking place and draw the first rectangle.

II. In the `OnMouseMove` event handler, erase the existing rectangle and draw a new rectangle at the current mouse point.

III. In the `OnMouseUp` event handler, clear the flag indicating dragging and erase the last rectangle drawn.

I'll discuss these steps in detail in the following sections.

Handling OnMouseDown

The action starts in the `OnMouseDown` event handler:

```
void __fastcall TMainForm::FormMouseDown(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Dragging = true;
  DrawRect.left = X;
  DrawRect.top = Y;
  DrawRect.right = X;
```

```
  DrawRect.bottom = Y;
  Canvas->DrawFocusRect(DrawRect);
}
```

I first set a flag indicating that dragging is taking place. After that I create a drawing rectangle using the `X` and `Y` parameters passed to the event handler. `DrawRect` is a `TRect` variable and is declared in the private section of the form's class. The `X` and `Y` parameters are, of course, the current position of the mouse. Finally, I draw a rectangle on the screen using the `DrawFocusRect()` method of the canvas (albeit a very small rectangle). I'm now going to detour for a moment and discuss `DrawFocusRect()` and how it works.

Drawing the rectangle

The selection rectangle itself can be drawn using any of several GDI drawing functions. For this article I chose the simplest of these functions, `DrawFocusRect()`. The `TCanvas` class has a `DrawFocusRect()` method that simply calls the Windows API function of the same name. `DrawFocusRect()` draws a rectangle on the canvas using a dotted line. This is the very same rectangle that Windows draws to indicate that a control has focus (hence, the name).

The key feature of `DrawFocusRect()` is that it is an `XOR` drawing operation. This means that the rectangle bits are `XOR`'d with the bits on the canvas. As a result, you can call `DrawFocusRect()` once to draw the rectangle, and call `DrawFocusRect()` a second time to erase the rectangle.

Now that you understand how `DrawFocusRect()` works we can get back to our discussion of drawing a selection rectangle.

Handling OnMouseMove

Keeping in mind how `DrawFocusRect()` works, take a look at the code for the `OnMouseMove` event handler:

```
void __fastcall TMainForm::FormMouseMove(
  TObject *Sender, TShiftState Shift,
  int X, int Y)
{
  if (Dragging) {
    Canvas->DrawFocusRect(DrawRect);
    DrawRect.right = X;
    DrawRect.bottom = Y;
    Canvas->DrawFocusRect(DrawRect);
  }
```

```
}
```

This code first checks to see if the `Dragging` variable is `true`. The `OnMouseMove` event is generated every time the mouse moves and I only want to execute the rectangle drawing code if dragging is taking place.

Next I erase the current rectangle on the screen by calling `DrawFocusRect()`. Remember that the `OnMouseDown` event handler drew the initial rectangle. There will always be a rectangle on the screen when `OnMouseDown` is generated so this call to `DrawFocusRect()` is needed to erase that rectangle.

Next I update the bottom-right point of the rectangle. This will be the size of the new rectangle drawn on the screen.

Finally, I call `DrawFocusRect()` again to display the focus rectangle in its new location.

Handling OnMouseUp

The drawing operation is, naturally, terminated in the `OnMouseUp` event handler. Here's how that event handler looks:

```
void __fastcall TMainForm::FormMouseUp(
  TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Dragging) {
    Dragging = false;
    Canvas->DrawFocusRect(DrawRect);
  }
}
```

This code is basically self-explanatory. It sets the `Dragging` variable to false and then calls `DrawFocusRect()` one last time to erase the focus rectangle for the last time.

Drawing on the desktop

The code presented in the previous sections is very basic in that it shows the steps required to draw a selection rectangle with the least amount of code. It may not, however, produce the results you want. In particular, the focus rectangle will be drawn *behind* any components on the form. In order to draw the focus rectangle on top of all components on the form you will need to draw not on the form's canvas, but on the desktop itself.

Like most aspects of Windows programming with the C++Builder, you can go about drawing on the desktop in more than one way. You might use the Windows API to get a handle to the desktop's device context (DC) and use the API to draw the focus rectangle. Another way is to use the `TCanvas` class to accomplish the same thing. I prefer `TCanvas` because it cleans up all GDI resources when the `TCanvas` instance is destroyed.

The first step in creating a `TCanvas` object that allows you to draw on the desktop is to create an instance of `TCanvas` and assign a handle to the desktop DC to the `TCanvas` object's `Handle` property:

```
DesktopCanvas = new TCanvas;
DesktopCanvas->Handle = GetDC(0);
```

The Windows API function `GetDC()` returns a handle to a device context for a particular window. When you pass 0 to `GetDC()` you get a handle to the desktop DC. After setting the `Handle` property, you have a `TCanvas` object that can be used to draw directly on the desktop. You can now use the code presented earlier, replacing the form's `Canvas` property with your `DesktopCanvas` variable.

You'll have to take steps to insure that the mouse cursor is confined to your application's main window and possibly some other housekeeping chores. Other than that it is relatively easy to draw on the desktop versus the form's canvas. This technique will produce the results you are likely to require in an application that uses selection rectangles.

The example program for this article allows you to draw either on the form's canvas or on the desktop. You can download the code from our Web site at **www.bridgespublishing.com**.

Conclusion

As you can see, drawing a selection rectangle is not difficult. The more difficult part comes in what to do after the selection rectangle is drawn (selecting objects, for example). That's a discussion we'll leave for another time.

August 1999

Determining compiler version

by Kent Reisdorph

If you are writing applications or components that must compile under different versions of C++Builder then you may need to determine the version of the compiler in order to conditionally compile sections of code. Each version of the compiler assigns a value to a macro called __BORLANDC__. You can test the version of the compiler by checking the value of this macro in your code. For example:

```
#if (__BORLANDC__ >= 0x530)
#pragma package(smart_init)
#endif
```

In this case, the #pragma package line is only compiled if the code is being compiled on C++Builder 3 or greater (C++Builder 1 does not use packages, of course). We use code like this to insure that the examples for the journal will compile under any version of C++Builder. You can easily find out the value of the __BORLANDC__ macro by running this code under various versions of C++Builder:

```
char buff[10];
sprintf(buff, "%x", __BORLANDC__);
Label1->Caption = buff;
```

Table A shows the values of the __BORLANDC__ macro for the current versions of C++Builder. Armed with this information you can easily write both applications and components that work on all current versions of C++Builder.

Table A: Values of __BORLANDC__ by compiler

Value	Compiler version
0x520	C++Builder 1
0x530	C++Builder 3
0x540	C++Builder 4

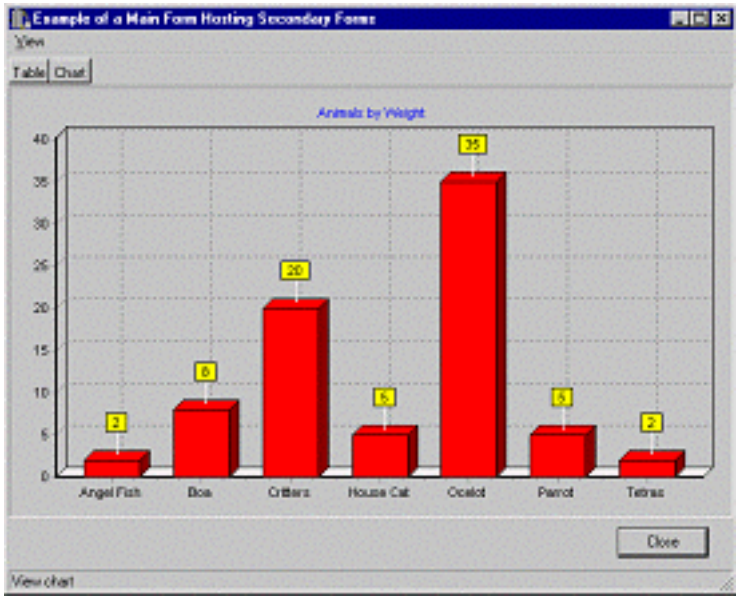
Hosting forms within a main form

by Kent Reisdorph

Almost every serious C++Builder application has secondary forms in addition to the main form. Sometimes these forms are presented to the user in the form of dialogs and other times in the form of modeless windows. The VCL model makes creating and displaying secondary forms easy.

The problem with this ease of use is that sometimes C++Builder programmers have trouble thinking outside the box. Not all applications can benefit from the modeless window model. Some applications need to display various windows—views, if you will—within the main form. This article will explain how to host a secondary form within your main form. The secondary form will appear to be part of the main form and the user won't even know that a second form is being shown. **Figure A** shows a main form with a second form in the client area.

Figure A



A main form hosting a secondary form in the client area.

Understanding the parent/child relationship

The basic idea of this type of application is to make all secondary forms children of the main form. This design is common in other frameworks (such as OWL or MFC), but is not something that you see often in a VCL application. The VCL doesn't allow you to simply set a property in order to make one form the child of the second form. You must do a little work in order to make one form a child of another. (See the sidebar, *Understanding VCL ownership and parentage* for more information on how the VCL model

deals with child windows.)

In order to have one form host another form, you must tell Windows that the secondary form is a child of the main form. In C++Builder programming we tend to think of forms as windows and components as child objects. The truth is, though, that from Windows' perspective all forms and components are simply windows. You can specify that any window (a form or a component) be the child of another window. You only need to step out of the VCL box for a moment.

The example program's design

Before I go on, I want to give you some background on the example program for this article. The example, called PARENTING, contains a main form that has a toolbar at the top, and a status bar at the bottom. In addition to the main form, the program has two child forms. One child form, called `TTableForm`, displays the ANIMALS.DBF table in a grid. The ANIMALS table is one of the sample database tables that ships with C++Builder. A second child form, `TChartForm` displays the ANIMALS table in a `TChart`. (My apologies to those of you who are using C++Builder 4 Standard as it does not ship with the database components.)

You can choose to view either the table form or the chart form by selecting an item from the main menu, or by clicking on one of the toolbar buttons. When you select a form to display, the active form, if any, will be destroyed and the selected form displayed. The child form will be displayed in the client area of the main form below the toolbar and above the status bar. As an added feature, the child form is resized to always fill the client area of the main form if the main form is resized.

With that description of the example program behind us, we can move on to a discussion of how to set up the child forms.

Setting up the child forms

One of the advantages to hosting child forms within the main form is that you can design your child form just as you would any other secondary form. That is, you create a new form, add components to it, and write the code for the form. This makes it easy to design the child form, and to keep all the code that drives the child form in one central place.

Overriding the `CreateParams()` method

As I have said, in order for the main form to host a secondary form you need to set the main form as the secondary form's parent. This is done by overriding the VCL's `CreateParams()` method. `CreateParams()` is called when the VCL creates the underlying window associated with a form. The declaration for `CreateParams()` looks like this:

```
void __fastcall
  CreateParams(TCreateParams& Params);
```

As you can see, `CreateParams()` has a reference to a `TCreateParams` structure as its only parameter. `TCreateParams` is defined in the VCL as follows:

```
struct TCreateParams
{
  char *Caption;
  unsigned Style;
  unsigned ExStyle;
  int X;
  int Y;
  int Width;
  int Height;
  HWND WndParent;
  void *Param;
  tagWNDCLASSA WindowClass;
  char WinClassName[64];
};
```

This structure contains all the information that Windows needs to create a window. (If you have done Windows programming using the API, you will recognize that the members of the `TCreateParams` structure map to a Windows `CREATESTRUCT` structure.) When you override `CreateParams()` you first call the base class's `CreateParams()` method. After that, you can modify the individual members of the `TCreateParams` structure. Here's how a basic overridden `CreateParams()` method might look:

```
void __fastcall
TChartForm::CreateParams(TCreateParams& Params)
{
  TForm::CreateParams(Params);
  Params.Style = WS_CHILD | WS_CLIPSIBLINGS;
  Params.WndParent = MainForm->Handle;
  Params.X = 0;
  Params.Y = 0;
  Params.Width = MainForm->ClientRect.Right;
  Params.Height = MainForm->ClientRect.Bottom;
}
```

The key points in the preceding code are the lines that set the `Style` and `WndParent` members of the `TCreateParams` structure. Note that the style is set to a value that includes the `WS_CHILD` and

WS_CLIPSIBLINGS window styles. The WS_CHILD window style specifies that this window is the child of another window. By definition, a child window has no title bar. At design time the child form will have a title bar, but the title bar will be removed when Windows creates the form at runtime. The WS_CLIPSIBLINGS style insures that the various child windows on the main form don't interfere with one another when the form is painted.

Obviously, a child window must have a parent. You specify the parent by assigning the window handle of the parent window to the TCreateParams structure's WndParent member. As you can see from the preceding code, the WndParent member is set to the Handle property of the main form. Assigning the parent is relatively straightforward, so I won't go into further detail on the subject.

Setting the child form's properties

In addition to the code you see in the CreateParams() method, you must also set some of the child form's properties. Most of the form's properties can be left at their default values. You should, however, set the AutoScroll property to false. This assumes, of course, that your form is designed in such a way that scrolling the form will not be necessary. You should also set the Position property to poDefault, since the size and position of the child window will be set in the CreateParams method. The Caption and BorderIcons properties will be ignored so you shouldn't have to worry about them. Be sure to leave the BorderStyle property set to bsSizeable, and the BorderWidth property set to 0. If you change these properties, the child form won't fit properly on the main form.

Accounting for other components on the form

In many cases, your main form will contain components besides the secondary form. For example, your main form may have a toolbar and a status bar. In that case, you need to account for the toolbar and status bar when you set the X, Y, Width, and Height members of the TCreateParams structure. The child form must fit between the toolbar at the top of the form, and the status bar at the bottom of the form. Given that, the code that sets the various members of the TCreateParams structure might look like this:

```
Params.X = 0;
Params.Y = MainForm->ToolBar->Height + 1;
Params.Width = MainForm->ClientRect.Right;
Params.Height =
    (MainForm->StatusBar->Top - 1) - Params.Y;
```

Note that the Y member is set to the bottom of the toolbar, plus one pixel. The width of the child form is set to the width of the main form's client area, and the height of the child form is calculated based on the top of the child window, and the top of the status bar. Basically, the height is set to that part of the main form's client area that falls between the bottom of the toolbar and the top of the status bar.

That is all that is required to make a secondary form the child of the main form. There are a few other features you may want to implement in the child form, but I'll save discussion of those features for a bit later.

Setting up the main form

The main form also needs to be set up to handle a secondary form hosted as a child. First, you must remove the child forms from the application's auto-create list. You will be creating the child forms when needed and don't want them auto-created. In fact, if you don't remove the child forms from the auto-create list they will automatically display when the application starts.

You'll need a variable that keeps track of which child form is currently active. I declared the variable in the main form's public section as follows:

```
TForm* ActiveChild;
```

The `ActiveChild` variable is public because the child forms need access to the variable. I'll show you how this variable is used in just a bit.

Now you can write the code that will display the child form. First, look at the code, and then I'll explain it:

```
void __fastcall
TMainForm::Chart1Click(TObject *Sender)
{
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}
```

This method is the `OnClick` handler for a menu item on the main form. As you might guess, the handler displays the `TChartForm` child. Note that I first check to see if the `ActiveChild` variable is non-zero. `ActiveChild` will be non-zero if a child window is active. If `ActiveChild` is not zero, I delete the pointer associated with the variable to destroy the active child form. If I don't first destroy the active child form, my program will continue to stack child after child on top of one another.

Next, I create an instance of the `TChartForm` class. I then assign the pointer returned from operator `new` to the `ActiveChild` variable. This way, the `ActiveChild` variable always contains a pointer to the current child form. Finally, I call the `Show()` method to display the child form. The last two lines of code insure that the menu displays a check mark next to the menu item representing either the table or chart view.

In order to complete the discussion of the `ActiveChild` variable, I have to take you back to the child form unit for a moment. Each of the child forms contains an event handler for the `OnClose` event that looks like this:

```
void __fastcall TChartForm::FormClose(
  TObject *Sender, TCloseAction &Action)
{
  MainForm->ActiveChild = 0;
  MainForm->Chart1->Checked = false;
  Action = caFree;
}
```

Note that when the form is destroyed, the main form's `ActiveChild` variable is set to 0. I also uncheck the menu item associated with the child form, and set the `Action` parameter to `caFree`. Setting `Action` to `caFree` tells the VCL to free the memory associated with the form.

You might be wondering why the `FormClose` handler contains those last two lines of code. After all, I just showed you code in the main form that performs these same actions. The answer is that each of the child forms contains a button called Close that can, naturally, be used to close the form. If the form is closed using the Close button, then the memory needs to be freed and the menu item unchecked.

A few extra features

There is at least one feature of the example program that I haven't discussed yet. That is, if the child form is larger than the current client area of the main form, the main form is resized to accommodate the child. That code is placed in the child form's `CreateParams()` method. Earlier, I showed you an example of a basic `CreateParams()` method. I left out the code that resizes the main form because I didn't want to introduce more complex code at that time. You can find the completed `CreateParams()` method for the `TChartForm` class in **Listing B**. The method only differs from that shown earlier in that it contains this code:

```
if (Width > MainForm->ClientWidth)
  MainForm->ClientWidth = Width;
if (Height > (MainForm->StatusBar->Top -
    MainForm->ToolBar->Height))
```

```
  MainForm->ClientHeight = Height +
  MainForm->ToolBar->Height +
  MainForm->StatusBar->Height;
```

This code checks to see if the child form's width is greater than main form's `ClientWidth` property. If it is, then the main form's `ClientWidth` property is set to the width of the child form. The next few lines, although a bit more complex, do the same thing for the main form's client height.

The result of this code is that the main form will always be resized to accommodate the child form being displayed.

The example program also accounts for the main form being resized. If the main form is resized, the child form must also be resized so that it continues to fill the client area of the main form. Here's the `OnResize` event handler for the main form:

```
void __fastcall
TMainForm::FormResize(TObject *Sender)
{
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
      ActiveChild->Top;
  }
}
```

This code is fairly straightforward, so I don't need to go over every line. Note, though, that I first check the value of the `ActiveChild` variable to be sure that it is non-zero (that is, that it points to a child form). Obviously I don't need to do anything in the `OnResize` event handler if no child form is currently active. The remaining code is a variation of the code you saw in the child form's `CreateParams()` method. It simply calculates the new size for the child window and sets the `Width` and `Height` properties accordingly.

Conclusion

Listing A contains the source code for the example program's main form. **Listing B** shows the source code for the `TChartForm` unit. I don't show the headers for these units because they don't contain any meaningful code. I also don't show the code for the `TTableForm` unit because it is identical to the code for the `TChartForm` unit. You can download the example program from our Web site at `http://www.bridgespublishing.com`.

Hosting child windows within the main form provides a clean alternative to using MDI, and also to an application that would otherwise display data to the user as modeless forms. Using child forms allows you to design your secondary windows using the form designer, and also helps you keep the code that drives the child form in one place.

Listing A: *MAINU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"
#include "ChartU.h"
#include "TableU.h"

#pragma resource "*.dfm"
TMainForm *MainForm;

__fastcall
TMainForm::TMainForm(TComponent* Owner)
  : TForm(Owner)
{
  // Zero out the ActiveChild variable or it
  // will contain random data.
  ActiveChild = 0;
  // Open the main form's Table component.
  Table->Active = true;
}

void __fastcall
TMainForm::Table1Click(TObject *Sender)
{
  // If this form is already being displayed
  // then return without doing anything.
  if (Table1->Checked)
    return;
  // Delete the active child if it exists.
  if (ActiveChild) {
    delete ActiveChild;
    ActiveChild = 0;
  }
  // Create an instance of TTableForm.
  TTableForm* form = new TTableForm(this);
  // Assign the DBGrid::DataSource property of
  // the TTableForm's DBGrid to the datasource
```

```
  // on the main form.
  form->DBGrid->DataSource = DataSource;
  // Keep track of the active child.
  ActiveChild = form;
  // Show the form.
  form->Show();
  // Update the check marks on the View menu.
  Table1->Checked = true;
  Chart1->Checked = false;
}

void __fastcall
TMainForm::Chart1Click(TObject *Sender)
{
  // Essentially the same code as described
  // for the Table1Click method above.
  if (Chart1->Checked)
    return;
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}

void __fastcall
TMainForm::FormResize(TObject *Sender)
{
  // If the main form is resized, resize the
  // active child to fill the client area of
  // the main form.
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
      ActiveChild->Top;
  }
}
```

Listing B: *CHARTU.CPP*

```
#include <vcl.h>
#pragma hdrstop

#include "ChartU.h"
#include "MainU.h"

#pragma resource "*.dfm"

TChartForm *ChartForm;

__fastcall
TChartForm::TChartForm(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
TChartForm::CreateParams(TCreateParams& Params)
{
  // Call the base class CreateParams method.
  TForm::CreateParams(Params);
  // Set the style to create a child window.
  Params.Style = WS_CHILD | WS_CLIPSIBLINGS;
  // Set the window's parent as the main form.
  Params.WndParent = MainForm->Handle;
  // The X position of the window is 0.
  Params.X = 0;
  // If the main form is too narrow or too short
  // to accomodate the child form, resize it.
  if (Width > MainForm->ClientWidth)
    MainForm->ClientWidth = Width;
  if (Height > (MainForm->StatusBar->Top -
      MainForm->ToolBar->Height))
    MainForm->ClientHeight = Height +
      MainForm->ToolBar->Height +
      MainForm->StatusBar->Height;
  // The Y position of the child form is just
  // below the main form's toolbar.
  Params.Y = MainForm->ToolBar->Height + 1;
  // The width of the child form is the same as
  // the main form's client width.
  Params.Width = MainForm->ClientRect.Right;
  // Calculate a height based on the bottom of
  // the toolbar, and the top of the status bar.
```

```
  Params.Height =
    (MainForm->StatusBar->Top - 1) - Params.Y;
}

void __fastcall TChartForm::FormClose(
  TObject *Sender, TCloseAction &Action)
{
  // Update the main form's ActiveChild property
  // to indicate no child is active.
  MainForm->ActiveChild = 0;
  // The child form might have been closed via
  // the Close button so update the main menu's
  // check marks, and tell the VLC to clean up
  // the memory for this form.
  MainForm->Chart1->Checked = false;
  Action = caFree;
}

void __fastcall
TChartForm::CloseBtnClick(TObject *Sender)
{
  Close();
}
```

August 1999

Ras update

by Kent Reisdorph

In our March and April issues we presented articles on using Microsoft's Remote Access Services (RAS) to establish dial-up connections. Those articles were written using C++Builder 3. With C++Builder 4, however, I noticed that my RAS applications no longer worked as I expected. One of the symptoms was a runtime error from some of the RAS functions. The return value from those functions was ERROR_BUFFER_TOO_SMALL. The textual description of this error is, "The caller's buffer is too small." I spent more than a few hours trying to track down this problem. I started digging around in RAS.H and found some entries like this:

```
RASCONNA
{
  DWORD dwSize;
  HRASCONN hrasconn;
  CHAR szEntryName[ RAS_MaxEntryName + 1 ];
  #if (WINVER >= 0x400)
  CHAR szDeviceType[ RAS_MaxDeviceType + 1];
  CHAR szDeviceName[ RAS_MaxDeviceName + 1];
  #endif
  #if (WINVER >= 0x401)
  CHAR szPhonebook [ MAX_PATH ];
  DWORD dwSubEntry;
  #endif
};
```

Hmm… that could certainly lead to a "buffer too small error" if the WINVER macro were defined as something other than 0x0400. Curious, I started hunting in the Windows headers. Here's what I found in WINDEF.H:

```
#ifndef WINVER
#define WINVER 0x0500
#endif /* WINVER */
```

Aha! Presumably Microsoft and Borland were building support for Windows 2000 (which includes many enhancements to RAS) into the Windows headers and those headers shipped with C++Builder 4. In C++Builder 3, WINVER was defined as 0x0400 so obviously something changed.
Once I figured out the problem, it did not take long to fix it. If you are building applications

that use RAS in C++Builder 4 you must use this construct when including the RAS headers:

```
#pragma warn -dup
#define WINVER 0x400
#include <ras.h>
#include <raserror.h>
#define WINVER 0x500
```

This code temporarily redefines the WINVER macro to 0x400 and then sets it back to 0x500 after including the RAS headers. The line that reads, #pragma warn -dup suppresses the warning the compiler will issue that says:

```
Redefinition of ‘WINVER’ is not identical.
```

It’s not only RAS.H that suffers from this problem. For a list of affected headers, search the Windows headers for these lines of text:

```
WINVER >= 0x401
WINVER >= 0x0500
WINVER >= 0x500
```

You will no doubt find several places where problems may arise with your existing code. As I have said, this problem caused me a few hours of aggravation before I figured out what was going wrong. I hope that this information will help you avoid the frustration I experienced before I discovered the solution.

August 1999

Registering AnsiString property editors

by

One of the most powerful features of C++Builder is the ability to create your own components. Certainly not all of you are writing your own components. Even fewer of you have written property editors for your components. On the other hand, if you have written a property editor for a property based on AnsiString, you almost certainly ran into serious problems trying to register the property editor.

Simply put, the VCL will not let you register a property editor for a property based on AnsiString. This article will explain how to register a property editor for an AnsiString property.

As part of this article I'll also show you how to register property editors for the C++ integral data types such as int, char, long, and so on.

Identifying the problem

You've just written a property editor and, of course, you're proud of your creation. All you have to do is register the property editor and try it out for the first time. Naturally, you try to register the property editor with the following code.

```
RegisterPropertyEditor(__typeinfo(String),
__classid(TMyComponent), "FileName",
__classid(TMyFileNameEditor));
```

You compile the package and are immediately greeted with an error from the compiler:

```
__classid requires VCL style class type.
```

At this point most of you would try to work out the problem and, no doubt, throw your hands up in frustration a few miserable hours later. This is the kind of error that is not easily understood, nor remedied.

The problem is that __typinfo used in RegisterPropertyEditor() requires a VCL-derived class or a Pascal data type. AnsiString, being a pure C++ helper class, is not derived from any VCL class and, as such, does not qualify for use with __typeinfo. This problem not only affects AnsiString property editors, but also property editors for properties based on C++ types such as int or char.

Now that you see what the problem is, I'll explain the solution.

The solution(s)

There are at least three solutions for the AnsiString property editor dilemma. One solution to the AnsiString property editor problem involved deriving a class from TPersistent and declaring an AnsiString as a class member. The class itself was registered as the type for the property and the AnsiString data member held the string data. This solution, while the only one available for a long time, is messy. I won't go into the details, but you can take my word for it.

The remaining two solutions are easier as you'll see in the following sections.

The Van Ditta solution

Several months ago I read a post from Mark Van Ditta in the Borland newsgroups. Mark described how he had solved the AnsiString property editor problem. The solution was both simple and ingenious. Mark was determined to solve the AnsiString property editor problem. He discovered that the __typeinfo keyword was emulating what the Object Pascal TypeInfo() function does—return a pointer to a TTypeInfo structure. (Later I realized that the VCL help for RegisterPropertyEditor() has a hint that pointed in this direction.) With that information Mark created a function that can be used with RegisterPropertyEditor(). The function looked something like this:

```
TTypeInfo* AnsiStringTypeInfo()
{
  TTypeInfo* typeInfo = new TTypeInfo;
  typeInfo->Name = "AnsiString";
  typeInfo->Kind = tkLString;
  return typeInfo;
}
```

Like I said, simple yet ingenious. This function simply creates a dynamic instance of the TTypeInfo structure, sets the Name and Kind members of the structure, and then returns a pointer to the structure. (You don't need to delete the memory associated with the structure because the VCL will take care of it for you.) The correct call to RegisterPropertyEditor() is shown below.

```
RegisterPropertyEditor(
  AnsiStringTypeInfo(),
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

So long as RegisterPropertyEditor() gets a pointer to a TTypeInfo structure, and as long as the Name and Kind members of the structure contain information that the VCL understands, everything works.

Once you understand the basic idea of the AnsiStringTypeInfo() function, it's easy to create additional functions to register property editors of the integral data types, such as int. The key is in understanding TTypeInfo::Kind. The Kind member of TTypeInfo is an enumeration of type TTypeKind. TTypeKind is defined as follows:

```
enum TTypeKind {
  tkUnknown, tkInteger, tkChar,
  tkEnumeration, tkFloat, tkString, tkSet,
  tkClass, tkMethod, tkWChar, tkLString,
  tkWString, tkVariant, tkArray, tkRecord,
  tkInterface, tkInt64, tkDynArray };
```

Obviously, you can create functions similar to AnsiStringTypeInfo() by using one of the values of TTypeKind along with a corresponding value for the Name member of TTypeInfo. Here's a type info function for a property of type int:

```
TTypeInfo* IntTypeInfo(void)
{
  TTypeInfo* typeInfo = new TTypeInfo;
  typeInfo->Name = "int";
  typeInfo->Kind = tkInteger;
  return typeInfo;
}
```

Note that I set the Name member to "int" and the Kind member to tkInteger. Some trial and error is required in order to get the Name and Kind members synchronized, but for the most part it's pretty obvious what to pass for the Name member, given a known type. If your property editors don't work then you may have to try different values for the Name member until the property editor works.

The Mitov solution

Another solution was recently presented to me by Boian Mitov. Boian's solution looks like this:

```
TPropInfo* PropInfo = ::GetPropInfo
  (__typeinfo(TMyComponent), "FileName" );
```

```
RegisterPropertyEditor(*PropInfo->PropType,
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

This solution is actually a bit cleaner than the first solution presented. In this case a pointer to a TPropInfo structure is used. The property information for a particular property is first obtained by a call to
GetPropInfo(). The result is a pointer to a TPropInfo structure. TPropInfo is defined in TYPEINFO.HPP:

```
struct TPropInfo
{
  PTypeInfo *PropType;
  void *GetProc;
  void *SetProc;
  void *StoredProc;
  int Index;
  int Default;
  short NameIndex;
  System::ShortString Name;
};
```

Notice that the PropType parameter is a pointer to a PTypeInfo type. Since PTypeInfo is itself a pointer, the PropType member is a pointer to a pointer. Since PropType is a pointer to a pointer it must be dereferenced before passing it to the RegisterPropertyEditor() function.

Here's how the call to RegisterPropertyEditor() looks:

```
RegisterPropertyEditor(*PropInfo->PropType,
  __classid(TMyComponent), "FileName",
  __classid(TMyFileNameEditor));
```

As I have said, this solution is a bit cleaner than the first solution presented, although each works equally well. In the end, choose the method that makes the most sense to you.

Conclusion

Registering AnsiString property editors is as vexing as any aspect of writing components in C++Builder. Since the first version of the compiler, component developers have looked to Borland to provide a reasonable solution to this problem. Unfortunately, each new version of C++Builder has failed to address the issue.

Thanks to the efforts of Mark Van Ditta and Boian Mitov, C++Builder programmers have not one, but two solutions to the problem.

Understanding VCL ownership and parentage

by Kent Reisdorph

The VCL has two elements that you need to understand; ownership and parentage. Ownership is the mechanism by which the VCL facilitates memory management. Ownership is determined by a form or component's `Owner` property. Consider the case of a typical application where all forms are auto-created at application startup. When the VCL auto-creates the forms, it assigns the `Application` object as the owner of all forms. When the application is closed, the `Application` object runs through its list of owned forms and deletes each form. Similarly, a form owns all the components placed on the form. As each form is deleted, it deletes all of the components on the form. Ownership is specific to the class library—the VCL in this case. Windows itself does not have support for the concept of ownership.

Parentage deals with visual components contained on a form (or within another component such as a panel). Parentage is a Windows concept, not a VCL concept. Parentage is controlled in the VCL via the `Parent` property. Every windowed control (visual component) must have a parent. The parent window is the window that hosts a control. When a button is placed on a form, the form becomes the button's parent. The `Top` and `Left` property of the button are specified relative to the parent's top left corner.

To summarize, ownership deals with memory management, and parentage deals with the parent/child relationship between windows. In a C++Builder application, the form is the parent and the components placed on the form are the children.

August 1999

Verifying data using CRC

by John Miano

Whenever a data message is transmitted it can be corrupted in ways that cannot be predicted. Even data sitting on a disk can be corrupted by hardware failures, software bugs, or malicious activity.
A key requirement of networking systems and file transfer programs is to ensure that data gets transferred reliably. When a message is received, the amount of data and contents must match what was transmitted exactly. If a message is modified during transmission, the system asks the sender to retransmit the corrupted message.

Simple Error Detection

The first problem in handling data corruption is how to determine if the data has been modified. An early method used in serial communications was to check the parity of the data bytes. Since ASCII characters only require 7-bits, one bit is left over in each byte and that bit can be used to ensure that each byte has either odd or even parity. In other words, the sender would set or clear the last bit to ensure that each data byte either had either an odd or even number of bits set, depending upon what the receiver was expecting. The problem with this system is that it only works reliably with errors of one bit. If more than one bit gets corrupted, there is only a 50/50 chance of detecting errors.

Checksum

Another simple method for error detection is to include a checksum value within each message. The sender calculates the checksum for the data before it is transmitted. The receiver also calculates the checksum on the data and, if the checksum values do not match, the receiver knows that the data has been corrupted in some way.

A simple checksum function is to take the sum of each byte in the message. Take a message consisting of the characters “ABCDEF”, for example. Before transmitting the message, an application can sum each byte and append the result (495) to the message.

While summing the data bytes allows errors to be detected in some situations, it is possible for many types of errors to slip by. For example, a message consisting of the characters “ACBDEF” differs from the previous string by only two bits, but the sum of the data bytes is the same. The data is obviously incorrect but a checksum won’t catch the error.

A Better Check Value

What is really needed for reliable error detection is a function that causes small changes in the input to produce large changes in the check value. In other words, we need a function that essentially produces a random value from a data block.

The most common method for generating a check value is known as the Cyclic Redundancy Check, or CRC. A CRC function can be used to calculate the CRC on a block of data. The CRC function works by treating a data block as a large binary number and dividing by a constant value. The CRC result is the division remainder.
In reality, the process of generating a CRC is not true division, but rather something that looks a lot like division. CRC calculations use binary arithmetic with no carries or borrows. Bit positions are XORed rather than subtracted.

Figure A show the process for generating the CRC value for the message “AB” (0100 0001 0100 0010 in binary) using the divisor 1 10001. Notice in Figure A how XOR operations are used in place of subtraction as in real division. Also notice that the number of bits in the CRC value is one less than the number of bits in the divisor.

Implementing the CRC

CRC is simple to implement in hardware but if the CRC were implemented in software by performing bit-by-bit pseudo-long division, it would be relatively computationally intensive. Fortunately long division is not required in a software implementation.

If you look at the individual XOR operations in Figure A you can see that at each step the one data bit is shifted into the remainder and XORed. The individual steps are trivial.

Software CRC implementations are invariably implemented with a lookup table, allowing byte-by-byte processing rather than by bit. The application maintains a register containing the CRC value and updates it as each byte is processed.

The following code fragment shows how the CRC is calculated for a buffer from a lookup table. BITCOUNT is the size of the CRC value in bits. I will explain how the lookup table is generated later.

```
for (unsigned int ii=0;ii<length;++ ii)
{
  int index = ((crc_register >> (BITCOUNT - 8)) ^ buffer[ii]);
  crc_register = crc_table [index & 0xFF] ^ (crc_register << 8);
}
```

The CRC Polynomial

In Figure A I made an arbitrary choice for the divisor in the CRC calculation. There are a number of CRC variants in use and one of the ways in which they vary is in the choice of the divisor. The choice of non-zero bits in the divisor determines the types of errors the CRC variant can detect.

Descriptions of the CRC frequently describe it in terms of polynomial division. Instead of representing the CRC divisor as 1 1000 0000 0000 0101 it is represented as x16 + x15 + x2 + 1. This is the reason you will usually see the divisor referred to as the CRC polynomial.

Since the highest order bit is always 1, when the polynomial is shown in hexadecimal the highest order bit is invariably left out. For example, the CRC polynomial shown above would be listed as the
2-byte value 0x8005 rather than 0x18005.

Most CRC functions produce either 16 or 32-bit values (17 or 33-bit polynomials). There are some 12-bit CRC functions, but these are not very common. The number of bits in the divisor limits the size of the block that can reliably detect errors. Divisors with more bits can reliably detect errors in larger data blocks.

Other CRC Variations

Besides the polynomial length and polynomial value, there are three other common variations among CRC functions.

Reversed CRC

From an implementation point of view, the most significant difference among CRC variants is the ordering of the bits in the data block. When performing division, start with the most significant bit and work to the least significant.

However, when data is transmitted, such as over a serial line, the least significant bit within a byte often comes first in the data stream. In a hardware implementation, it is easier to process the bits as they arrive.
As a result, there are CRC variants that process data bits both in the natural order and reversed. Since the CRC uses something that looks like division, but is not actually division, the order in which the bits are processed is irrelevant (as long as they are processed consistently in the same order).

This example shows the process for using a table lookup to calculate a reversed CRC. The two versions are almost identical except for way the CRC register is shifted.

```
for (unsigned int ii=0;ii < length; ++ii)
```

```
{
  crc_register = crc_table.values[(crc_register ^
    buffer [ii]) & 0xFF] ^ (crc_register >> 8);
}
```

Initialization

Refer back to the CRC process shown in Figure A. Suppose the message were corrupted with zero bits inserted at the start of the message. Since zero divided by the polynomial is zero, these extra bits do not change the CRC value. In order to detect this type of error, some CRC variants initialize the CRC register to a value other than zero.

Final Value

Some CRC variants modify the remainder value as well by taking the complement of the value (XOR with –1). The CRC variants that modify the remainder invariably initialize the CRC to a value other than 0. The converse is not true, however.

Classifying CRC Variants

In the previous sections I have identified the five ways common CRC variants differ: polynomial length, polynomial value, reversed or forward, initialization, and finalization. Table A shows how the major CRC variants fit into these classifications.

Table A: CRC Variants

Name	Polynomial Length	Polynomial 1	Reserved	Initialization	Final XOR
CRC-16	16	0x8005	Yes	0	0
X.25	16	0x1021	Yes	-1	-1
CCITT	16	0x1021	No	-1	0
XMODEM	16	0x1021	No	0	0
CRC-32	32	0x04D11DB7	Yes	-1	-1

Creating a generic CRC implementation

C++ templates provide a good mechanism for dealing with parameterized variants, such as the ones shown in Table A. Here we have a template class whose parameters are modeled on Table A.

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>

class Crc {
public:
  Crc();
  Crc(const Crc &);
  ~Crc() {}
  Crc &operator=(const Crc &);

  unsigned long value() const;
  void reset();
  void update(const char *buffer,
  unsigned int length);
private:
  struct CrcTable {
    enum { MAXBYTEVALUES = 256 };
    CrcTable();
    unsigned long values [MAXBYTEVALUES];
  };
  static const CrcTable crc_table;
  unsigned long crc_register;
};
```

Creating the Lookup Table

The following function creates the value for an entry in a lookup table for the forward CRC process. You can see that this function divides the input value using the process shown in Figure A. Here is the function:

```
unsigned int ForwardTableEntry(
  unsigned long polynomial,
  unsigned int entryindex,
  unsigned int bitcount)
{
  unsigned long result =
  entryindex << (bitcount - 8);
  for (unsigned int ii=0;
```

```
  ii<BITSPERBYTE; ++ii) {
    if ((result & bits [bitcount-1]) == 0)
      result <<= 1;
    else
      result = (result << 1) ^ polynomial;
  }
  unsigned long mask = ((1UL << (bitcount - 1)) - 1UL) |
    (1UL << (bitcount - 1));
  result &= mask;
  return result;
}
```

Entries for reversed CRC lookup tables are created in a similar manner, except with bit reversals.

```
const unsigned long bits [32] = {
  0x00000001UL, 0x00000002UL, 0x00000004UL,
  0x00000008UL, 0x00000010UL, 0x00000020UL,
  0x00000040UL, 0x00000080UL, 0x00000100UL,
  0x00000200UL, 0x00000400UL, 0x00000800UL,
  0x00001000UL, 0x00002000UL, 0x00004000UL,
  0x00008000UL, 0x00010000UL, 0x00020000UL,
  0x00040000UL, 0x00080000UL, 0x00100000UL,
  0x00200000UL, 0x00400000UL, 0x00800000UL,
  0x01000000UL, 0x02000000UL, 0x04000000UL,
  0x08000000UL, 0x10000000UL, 0x20000000UL,
  0x40000000UL, 0x80000000UL};

unsigned long Reverse(
  unsigned long value,
  unsigned int bitcount)
{
  unsigned long result = 0;
  for (unsigned int jj=0;jj<bitcount;++jj)
  {
    if ((value & bits [jj]) != 0)
      result |= bits [bitcount - jj - 1];
  }
  return result;
}

unsigned long ReverseTableEntry(
  unsigned int polynomial,
```

```
  unsigned int entryindex,
  unsigned int bitcount)
{
  unsigned long result = entryindex;
  for (unsigned int ii=0;ii < 8; ++ii)
  {
    if ((result & 1) == 0)
      result >>= 1;
    else
      result = (result >> 1) ^
    Reverse(polynomial, bitcount);
  }
  unsigned long mask =
  ((1UL << (bitcount - 1)) - 1) |
    (1UL << (bitcount - 1));
  result &= mask;
  return result;
}
```

Here is the class constructor and definition for the lookup table:

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>

const CrcTable Crc<
  BITCOUNT,
  POLYNOMIAL,REVERSE,INITIAL,
  FINALMASK>::crc_table;

template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>

Crc<BITCOUNT,POLYNOMIAL, REVERSE,
  INITIAL,FINALMASK>::CrcTable::CrcTable()
{
```

```
  if (REVERSE) {
    for(unsigned int ii=0;
      ii<MAXBYTEVALUES; ++ii) {
      values [ii] =
      ReverseTableEntry(
        POLYNOMIAL, ii, BITCOUNT);
    }
  } else {
    for(unsigned int ii=0;
      ii<MAXBYTEVALUES; ++ii) {
      values [ii] = ForwardTableEntry(
      POLYNOMIAL, ii, BITCOUNT);
    }
  }
}
```

The lookup table is a static member object so that all equivalent instantiations share the same table. The table defines a constructor that initializes it at startup.

The reset() function on the following page initializes the CRC using the value specified in the template parameter. Use this function when you want to use a CRC object to calculate the CRC value for more than one block of data.

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>

void Crc<BITCOUNT,POLYNOMIAL,
  REVERSE,INITIAL,FINALMASK>::reset()
{
  crc_register = INITIAL;
  return;
}
```

The update() function updates the CRC register using a data block:

```
template <
  unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
```

```
  unsigned long INITIAL,
  unsigned long FINALMASK>

void Crc<BITCOUNT,POLYNOMIAL,
  REVERSE,INITIAL,FINALMASK>
  ::update(const char *buffer,
  unsigned int length)
{
  // The process for updating depends upon
  // whether or not we are using the reversed
  // CRC form.
  if (REVERSE)
  {
    for (unsigned int ii=0;ii<length;++ii){
      crc_register = crc_table.values
        [(crc_register ^ buffer[ii]) & 0xFF]
        ^ (crc_register >> 8);
    }
  } else {
    for (unsigned int ii=0;ii<length;++ii) {
      unsigned long index = (
        (crc_register >> (BITCOUNT -
        BITSPERBYTE)) ^ buffer[ii]);
      crc_register =
        crc_table.values[index & 0xFF] ^
        (crc_register << BITSPERBYTE);
    }
  }
  return;
}
```

The value() function returns the contents of the CRC register XORed with the FINALMASK template parameter:

```
template <unsigned int BITCOUNT,
  unsigned long POLYNOMIAL,
  bool REVERSE,
  unsigned long INITIAL,
  unsigned long FINALMASK>

unsigned long Crc<BITCOUNT,POLYNOMIAL,
  REVERSE,INITIAL,FINALMASK>::value() const
{
```

```
  unsigned long result =
  crc_register ^ FINALMASK;
  // The initial value is
  // 1 << BITCOUNT - 1. The convolutions
  // prevent an integer overflow.
  static const unsigned long mask =
    ((1UL << (BITCOUNT - 1)) - 1UL) |
    (1UL << (BITCOUNT - 1));
    result &= mask;
  return result;
}
```

The example program for this article includes the copy constructor, default constructor, and assignment operator. You can download the example from our Web site at www.bridgespublishing.com.

Using the CRC Template Class

You need to create a header file and compilation for each CRC class you want to create from the template. To create a CRC-32 class you would create the header file CRC32.H that defines a type that uses the values in Table A as template parameters:

```
#include "crc.h"
typedef Crc <32, 0x04C11DB7,
true, 0xFFFFFFFF, 0xFFFFFFFF> Crc32;
```

To create the compilation unit CRC32.CPP, all you need are include directives.

```
#include "crc32.h"
#include "crc.cpp"
```

When you compile the CRC32.CPP file you must use the -Ja switch either at the command line or using

```
#pragma option -Ja
```

This ensures that the template members get created properly.

The following console application example shows how to use the Crc32 class to calculate the CRC-32 value for the string “abcdefgh12345678.”

```
#include <iostream>
```

```
#include <cstring>

using namespace std;
#include "crc32.h"

const char *msg1 = "abcdefgh";
const char *msg2 = "12345678";
main() {
  Crc32 crc;
  crc.update(msg1, strlen(msg1));
  crc.update(msg2, strlen(msg2));
  cout << hex << crc.value() << endl;
  return 0;
}
```

Don’t be concerned over efficiency because of the tests for reversed CRC in the update function. Since the value in the test is a constant, you can be assured that any compiler will optimize these tests out.

Conclusion

The CRC is an effective and efficient method for detecting corruption in data. A template implementation allows many CRC variants to be implemented with one set of code. The template CRC class shown here allows you easily incorporate data verification in your applications.

C++BUILDER

Tips & Techniques for Borland C++ Builder

Developer's Journal

Home

Free Issue

Articles Online

Subscriptions

About

Feedback

FAQ

Welcome to the online home of the C++Builder Developer's Journal

Are you getting the most from C++Builder? Each month, the *C++Builder Developer's Journal* delivers tips and techniques that will make you a better C++Builder programmer. Dollar for dollar, the Journal is the most cost-effective way to get the most from your investment in C++Builder. Why wait? Try a free trial copy today!

Start your subscription now for as little as $79 per year

Special CD Offer

Get over 3 years worth of back issues on CD! This new CD contains nearly 700 pages of articles, along with complete source code examples. This valuable resource is available for just $39.00!
For more information, click here.

For correspondence, please use the following addresses:

Editorial: editor@bridgespublishing.com
Subscriptions: subscriptions@bridgespublishing.com

Postal Address:

Bridges Publishing
P.O. Box 91064
Pasadena, CA 91109-1064
USA

Bridges Publishing is a member of The C++ Builder Programmer's Ring

[Previous 5 Sites | Skip Previous | Previous | Next | Skip Next | Next 5 Sites | Random Site | List Sites]

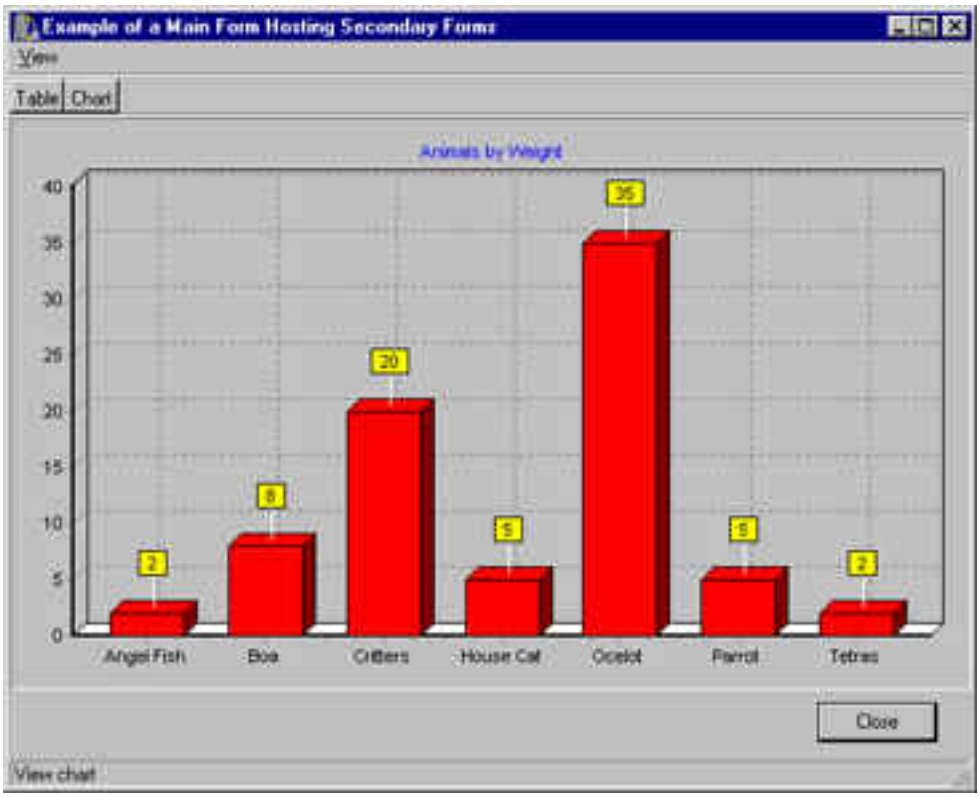
August 1999

Hosting forms in a main form

by Kent Reisdorph

Almost every serious C++Builder application has secondary forms in addition to the main form. Sometimes these forms are presented to the user as dialogs and other times as modeless windows. The VCL model makes creating and displaying secondary forms easy.

The problem with this ease of use is that sometimes C++Builder programmers have trouble thinking outside the box. Not all applications can benefit from the modeless window model. Some applications need to display various views within the main form. This article explains how to host a secondary form within your main form. The secondary form will appear to be part of the main form and the user won't even know that a second form is being shown. Figure A (shown below) displays a main form with a second form in the client area.



Understanding the parent/child relationship

The basic idea of this type of application is to make all secondary forms children of the main form. This design is common in other frameworks (such as OWL or MFC), but is not something that you see often in a VCL application. The VCL doesn't allow you to simply set a property in order to make one form the child of another. You must do a little work in order to make this happen.

To have one form host another, you must tell Microsoft Windows that the secondary form is a child of the main form. In C++Builder programming we tend to think of forms as windows and components as child objects. The truth is, though, that from Windows' perspective all forms and components are simply windows. You can specify that any window (a form or a component) be the child of another window. You only need to step out of the VCL box for a moment.

A better mousetrap

One of the advantages to hosting child forms within the main form is that you can design your child form just as you would any other secondary form. That is, you create a new form, add components to it, and write the code for the form. This makes it easy to design the child form, and to keep all the code that drives the child form in one central place.

The example program's design

Before I go on, I want to give you some background on the example program for this article. The example, called PARENTING, contains a main form that has a toolbar at the top and a status bar at the bottom. In addition to the main form, the program has two child forms. One child form, called TTableForm, displays the ANIMALS.DBF table in a grid. The ANIMALS table is one of the sample database tables that ships with C++Builder. A second child form, TChartForm displays the ANIMALS table in a TChart. (My apologies to those of you who are using C++Builder 4 Standard as it does not ship with the database components.)

You can choose to view either the table or the chart form by selecting an item from the main menu, or by clicking on one of the toolbar buttons. When you select a form to display, the active form, if any, will be destroyed and the selected form displayed. The child form will be displayed in the client area of the main form below the toolbar and above the status bar. In addition, the child form is resized to always fill the client area of the main form if the main form is resized.

Overriding CreateParams()

As I have said, in order for the main form to host a secondary form you need to set the main form as the secondary form's parent. This is done by overriding the VCL's CreateParams() method.

CreateParams() is called when the VCL creates the underlying window associated with a form. The declaration for CreateParams() looks like this:

```
void __fastcall CreateParams(TCreateParams& Params);
```

As you can see, CreateParams() has a reference to a TCreateParams structure as its only

parameter. TCreateParams is defined in the VCL as follows:

```
struct TCreateParams
{
  char *Caption;
  unsigned Style;
  unsigned ExStyle;
  int X;
  int Y;
  int Width;
  int Height;
  HWND WndParent;
  void *Param;
  tagWNDCLASSA WindowClass;
  char WinClassName[64];
};
```

This structure contains all the information that Windows needs to create a window. (If you have done Windows programming using the API, you will recognize that the members of the TCreateParams structure map to a Windows CREATESTRUCT structure.)

When you override CreateParams() you first call the base class's CreateParams() method. After that, you can modify the individual members of the TCreateParams structure. Here's how a basic overridden CreateParams() method might look:

```
void __fastcall TChartForm::CreateParams(
TCreateParams& Params)
{
  TForm::CreateParams(Params);
  Params.Style =
  WS_CHILD | WS_CLIPSIBLINGS;
  Params.WndParent = MainForm->Handle;
  Params.X = 0;
  Params.Y = 0;
  Params.Width =
  MainForm->ClientRect.Right;
  Params.Height =
  MainForm->ClientRect.Bottom;
}
```

The key points in this code are the lines that set the Style and WndParent members of the TCreateParams structure. Style is set to a value that includes the WS_CHILD and

WS_CLIPSIBLINGS window styles. WS_CHILD specifies that this window is the child of another window. By definition, a child window has no title bar. At design time the child form will have a title bar, but the title bar will be removed when Windows creates the form at runtime. The WS_CLIPSIBLINGS style insures that the various child windows on the main form don't interfere with one another when the form is painted.

Obviously, a child window must have a parent. You specify the parent by assigning the window handle of the parent window to the TCreateParams structure's WndParent member. As you can see from the preceding code, the WndParent member is set to the Handle property of the main form. Assigning the parent is relatively straightforward, so I won't go into further detail on the subject.

Setting the child form's properties

In addition to the code you see in the CreateParams() method, you must also set some of the child form's properties. Most of the form's properties can be left at their default values. You should, however, set the AutoScroll property to false. This assumes, of course, that your form is designed in such a way that scrolling the form will not be necessary. You should also set the Position property to poDefault, since the size and position of the child window will be set in the CreateParams() method. The Caption and BorderIcons properties will be ignored so you shouldn't have to worry about them. Be sure to leave the BorderStyle property set to bsSizeable, and the BorderWidth property set to 0. If you change these properties, the child form won't fit properly on the main form.

Other components on the form

In many cases, your main form will contain components besides the secondary form. For example, your main form may have a toolbar and a status bar. In that case, you need to account for the toolbar and status bar when you set the X, Y, Width, and Height members of the TCreateParams structure. The child form must fit between the toolbar at the top of the form, and the status bar at the bottom of the form.

Given that, the code that sets the various members of the TCreateParams structure might look like this:

```
Params.X = 0;
Params.Y = MainForm->ToolBar->Height + 1;
Params.Width = MainForm->ClientRect.Right;
Params.Height = (MainForm->StatusBar->Top-1) - Params.Y;
```

Note that the Y member is set to the bottom of the toolbar, plus one pixel. The width of the child form is set to the width of the main form's client area, and the height of the child form

is calculated based on the top of the child window and the top of the status bar. Basically, the height is set to that part of the main form's client area that falls between the bottom of the toolbar and the top of the status bar.
That is all that is required to make a secondary form the child of the main form. There are a few other features you may want to implement in the child form, but I'll save discussion of those features for later.

Setting up the main form

The main form also needs to be set up to handle a secondary form hosted as a child. First, you must remove the child forms from the application's auto-create list. You will be creating the child forms when needed and don't want them auto-created. In fact, if you don't remove the child forms from the auto-create list they will automatically display when the application starts.

You'll need a variable that keeps track of which child form is currently active. I declared the variable in the main form's public section as follows:

```
TForm* ActiveChild;
```

The ActiveChild variable is public because the child forms need access to the variable. I'll show you how this variable is used in just a bit.

Now you can write the code that will display the child form. First, look at the code, and then I'll explain it.

```
void __fastcall
TMainForm::Chart1Click(TObject *Sender)
{
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}
```

This method is the OnClick handler for a menu item on the main form. As you might guess, the handler displays the TChartForm child. I first check to see if the ActiveChild variable is non-zero.
ActiveChild will be non-zero if a child window is active. If ActiveChild is not zero, I delete

the pointer associated with the variable to destroy the active child form. If I don't first destroy the active child form, my program will continue to stack child after child on top of one another.

Next, I create an instance of the TChartForm class. I then assign the pointer returned from operator new to the
ActiveChild variable. This way, the ActiveChild variable always contains a pointer to the current child form. Finally, I call the Show() method to display the child form. The last two lines of code insure that the menu displays a check mark next to the menu item representing either the table or chart view.
In order to complete the discussion of the ActiveChild variable, I have to take you back to the child form unit for a moment. Each of the child forms contains an event handler for the OnClose event that looks like this:

```
void __fastcall TChartForm::FormClose(
TObject *Sender, TCloseAction &Action)
{
  MainForm->ActiveChild = 0;
  MainForm->Chart1->Checked = false;
  Action = caFree;
}
```

Note that when the form is destroyed, the main form's ActiveChild variable is set to 0. I also uncheck the menu item associated with the child form, and set the Action parameter to caFree. Setting Action to caFree tells the VCL to free the memory associated with the form.

You might be wondering why the FormClose handler contains those last two lines of code. After all, I just showed you code in the main form that performs these same actions. The answer is that each of the child forms contains a button called Close that can, naturally, be used to close the form. If the form is closed using the Close button, then the memory needs to be freed and the menu item unchecked.

A few extra features

There is at least one feature of the example program that I haven't discussed yet. That is, if the child form is larger than the current client area of the main form, the main form is resized to accommodate the child. That code is placed in the child form's CreateParams() method. Earlier, I showed you an example of a basic
CreateParams() method. I left out the code that resizes the main form because I didn't want to introduce more complex code at that time. You can find the completed CreateParams() method for the TChartForm class in Listing B. The method only differs from that shown earlier in that it contains this code:

```
if (Width > MainForm->ClientWidth)
  MainForm->ClientWidth = Width;
if (Height > (MainForm->StatusBar->Top - MainForm->ToolBar->Height))
  MainForm->ClientHeight = Height +
  MainForm->ToolBar->Height +
  MainForm->StatusBar->Height;
```

This code checks to see if the child form's width is greater than main form's ClientWidth property. If it is, then the main form's ClientWidth property is set to the width of the child form. The next few lines, although a bit more complex, do the same thing for the main form's client height.
The result of this code is that the main form will always be resized to accommodate the child form being displayed.

The example program also accounts for the main form being resized. If the main form is resized, the child form must also be resized so that it continues to fill the client area of the main form. The following code shows the OnResize event handler for the main form.

```
void __fastcall TMainForm::FormResize(TObject *Sender)
{
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
       ActiveChild->Top;
  }
}
```

This code is fairly straightforward, so I don't need to go over every line. Note, though, that I first check the value of the ActiveChild variable to be sure that it is non-zero (that is, that it points to a child form). Obviously I don't need to do anything in the OnResize event handler if no child form is currently active. The remaining code is a variation of the code you saw in the child form's CreateParams() method. It simply calculates the new size for the child window and sets the Width and Height properties accordingly.

Conclusion

Listing A contains the source code for the example program's main form. Listing B shows the source code for the TChartForm unit. I don't show the headers for these units because they don't contain any meaningful code. I also don't show the code for the TTableForm unit because it is identical to the code for the TChartForm unit.

Hosting child windows within the main form provides a clean alternative to using MDI, and also to an application that would otherwise display data to the user as modeless forms. Using child forms allows you to design your secondary windows using the form designer, and also helps you keep the code that drives the child form in one place.

```
#include <vcl.h>
#pragma hdrstop

#include “MainU.h”
#include “ChartU.h”
#include “TableU.h”

#pragma resource “*.dfm”
TMainForm *MainForm;

__fastcall TMainForm::TMainForm(TComponent* Owner) : TForm(Owner)
{
  // Zero out the ActiveChild variable or it
  // will contain random data.
  ActiveChild = 0;
  // Open the main form’s Table component.
  Table->Active = true;
}

void __fastcall TMainForm::Table1Click(TObject *Sender)
{
  // If this form is already being displayed
  // then return without doing anything.
  if (Table1->Checked)
    return;
  // Delete the active child if it exists.
  if (ActiveChild) {
    delete ActiveChild;
    ActiveChild = 0;
  }
  // Create an instance of TTableForm.
  TTableForm* form = new TTableForm(this);
  // Assign the DBGrid::DataSource property of the
  // TTableForm’s DBGrid to the datasource
  // on the main form.
  form->DBGrid->DataSource = DataSource;
  // Keep track of the active child.
  ActiveChild = form;
```

```
  // Show the form.
  form->Show();
  // Update the check marks on the View menu.
  Table1->Checked = true;
  Chart1->Checked = false;
}

void __fastcall TMainForm::Chart1Click(TObject *Sender)
{
  // Essentially the same code as described
  // for the Table1Click method above.
  if (Chart1->Checked)
    return;
  if (ActiveChild)
    delete ActiveChild;
  TChartForm* form = new TChartForm(this);
  ActiveChild = form;
  form->Show();
  Chart1->Checked = true;
  Table1->Checked = false;
}

void __fastcall
TMainForm::FormResize(TObject *Sender)
{
  // If the main form is resized, resize the
  // active child to fill the client area of
  // the main form.
  if (ActiveChild) {
    ActiveChild->Width = ClientRect.Right;
    ActiveChild->Height =
      (MainForm->StatusBar->Top - 1) -
      ActiveChild->Top;
    }
}

#include <vcl.h>
#pragma hdrstop

#include “ChartU.h”
#include “MainU.h”

#pragma resource “*.dfm”
```

```
TChartForm *ChartForm;

__fastcall
TChartForm::TChartForm(TComponent* Owner)
: TForm(Owner)
{
}

void __fastcall
TChartForm::CreateParams(TCreateParams& Params)
{
  // Call the base class CreateParams method.
  TForm::CreateParams(Params);
  // Set the style to create a child window.
  Params.Style = WS_CHILD | WS_CLIPSIBLINGS;
  // Set the window's parent as the main form.
  Params.WndParent = MainForm->Handle;
  // The X position of the window is 0.
  Params.X = 0;
  // If the main form is too narrow or too short
  // to accomodate the child form, resize it.
  if (Width > MainForm->ClientWidth)
    MainForm->ClientWidth = Width;
  if (Height > (MainForm->StatusBar->Top -
    MainForm->ToolBar->Height))
    MainForm->ClientHeight = Height +
    MainForm->ToolBar->Height +
    MainForm->StatusBar->Height;
  // The Y position of the child form is just
  // below the main form's toolbar.
  Params.Y = MainForm->ToolBar->Height + 1;
  // The width of the child form is the same as
  // the main form's client width.
  Params.Width = MainForm->ClientRect.Right;
  // Calculate a height based on the bottom of
  // the toolbar, and the top of the status bar.
  Params.Height =
    (MainForm->StatusBar->Top - 1) - Params.Y;
}

void __fastcall TChartForm::FormClose(
TObject *Sender, TCloseAction &Action)
{
  // Update the main form's ActiveChild property
```

```
  // to indicate no child is active.
  MainForm->ActiveChild = 0;
  // The child form might have been closed via
  // the Close button so update the main menu's
  // check marks, and tell the VLC to clean up
  // the memory for this form.
  MainForm->Chart1->Checked = false;
  Action = caFree;
}

void __fastcall
TChartForm::CloseBtnClick(TObject *Sender)
{
  Close();
}
```

Drawing transparent images

By Mark G. Wiseman

If you have used the VCL object, `TBitmap`, you know that you can use it to display bitmaps where one color is transparent. If you have ever looked at the source code for `TBitmap`, you find it takes quite a few lines of code to accomplish this. If you have tried to draw semi-transparent images in early versions of 32-bit Windows, you probably found it incredibly difficult.

There are two Windows API functions, `TransparentBlt()` and `AlphaBlend()`, that you can use to draw transparent images. The `TransparentBlt()` function will draw bitmaps with a single transparent color, just like the `TBitmap`. However, you can do this using only a few lines of code.

The `AlphaBlend()` function will draw images where every pixel in the image can have a different level of transparency—from totally opaque to completely transparent.

These two functions may sound too good to be true. Well, they are true if you're using Windows 98 or greater or Windows 2000 or greater. Unfortunately, they will not work with Windows 95 or Windows NT 4.

The online help included with C++Builder does not include help for `TransparentBlt()` or `AlphaBlend()`. You can, however, find help for these two functions on the Microsoft Web site at http://msdn.microsoft.com. Do a search on the function names.

In this article, I'll show you how to use these two functions, and since we'll need a way to see the transparency effects for testing, I'm also going to show you a neat little function for drawing checkerboard patterns.

I've written a demonstration program that lets you to see everything in action. You can download this program from the Bridges Publishing Web site.

Crown me

The `DrawCheckerboard()` function from this article's example program draws a checkerboard pattern. Here is the declaration for `DrawCheckerboard()`:

```
void DrawCheckerboard(
  TCanvas *canvas,
  int boardWidth, int boardHeight,
  TColor color1 = clBlack,
```

```
  TColor color2 = clWhite,
  int checkerWidth = 8,
  int checkerHeight = 8);
```

Listing A shows a snippet of source code from the demo program that shows how it uses the `DrawCheckerboard()` function. This pattern is used in the demo program as a background for the transparent images it draws.

The checkerboard pattern is drawn onto a `TCanvas`, the first argument to the function. The next two arguments are the height and width of the checkerboard. The rest of the arguments to `DrawCheckerboard()` have default values that produce a black-and-white checkerboard pattern where each square in the pattern has dimensions of 8 x 8 pixels. You can vary the colors of the alternating squares using the `color1` and `color2` arguments and the size of the squares using the `checkerWidth` `checkerHeight` arguments.

Actually, using the word *squares* is a little misleading. The pattern can consist of squares, rectangles, stripes or a solid color. For example, calling `DrawCheckerboard()` with `checkerWidth = 16` and `checkerHeight = 4`, will produce a pattern of rectangles. Calling the function with `checkerWidth = 0`, will produce horizontal stripes. Calling the function with `checkerHeight = 0`, will produce vertical stripes. And, if both `checkerWidth` and `checkerHeight` are equal to zero, `DrawCheckerboard()` will draw a solid background using `color1`.

In the demo program, I draw the checkerboard pattern onto a `TBitmap` object. The program allows you to change the colors and sizes of the squares and the transparency and visibility of the `TBitmap`. By setting the `Transparent` property of the `TBitmap` object to true, the color used in the top-left square will become transparent throughout the entire bitmap. This is a function of the `TBitmap` object, not `DrawCheckerboard()`.

TransparentBlt

Now, let's take a look at the `TransparentBlt()` function. **Figure A** shows the demo program with an image displayed by this function.

Figure A

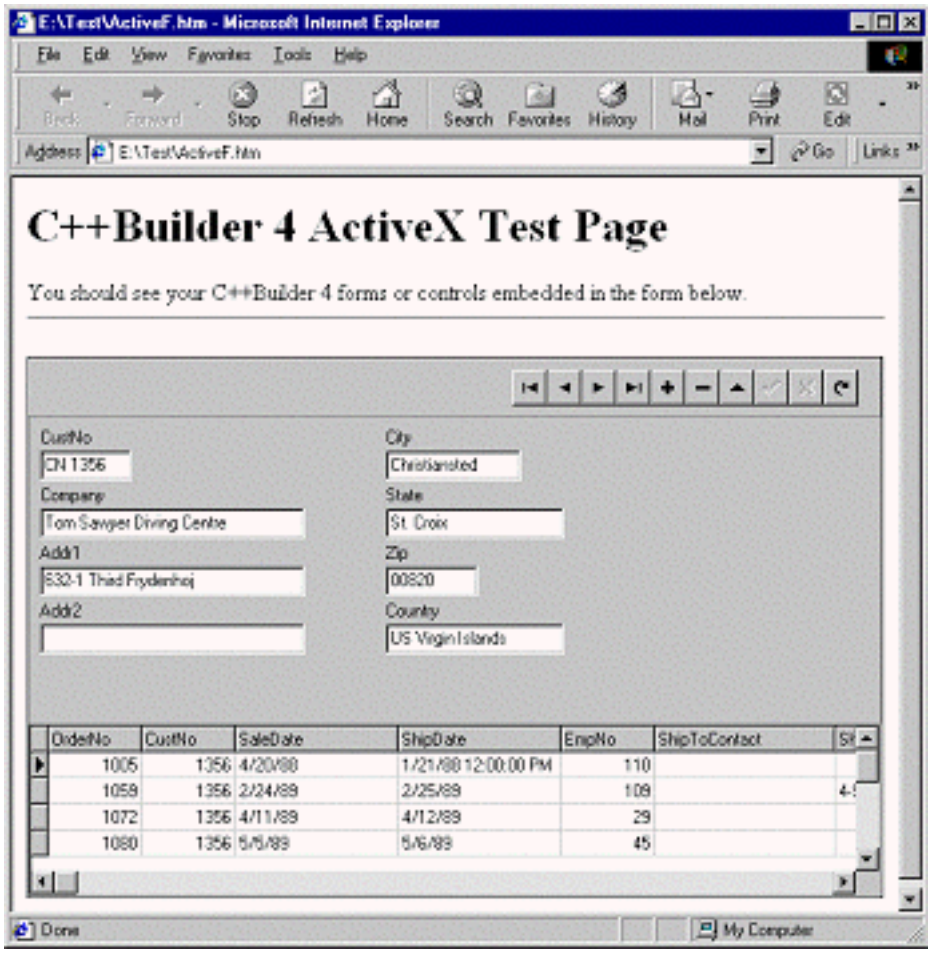


Image displayed with TransparentBlt() function.

The image consists of horizontal red and white stripes. The `TransparentBlt()` function has been told to draw the color red transparently. Here is the declaration for `TransparentBlt()`:

```
BOOL TransparentBlt(
  // handle to destination DC
  HDC hdcDest,
  // x-coord of destination
  // upper-left corner
  int nXOriginDest,
  // y-coord of destination
  // upper-left corner
  int nYOriginDest,
  // width of destination rectangle
  int nWidthDest,
  // height of destination rectangle
  int hHeightDest,
  // handle to source DC
  HDC hdcSrc,
```

```
  // x-coord of source
  // upper-left corner
  int nXOriginSrc,
  // y-coord of source
  // upper-left corner
  int nYOriginSrc,
  // width of source rectangle
  int nWidthSrc,
  // height of source rectangle
  int nHeightSrc,
  // color to make transparent
  UINT crTransparent
);
```

This function works just like the `StretchBlt()` function, except the last argument to this function is the color that should be drawn transparently. In the demo program, I use a `TPaintBox` object to draw the example bitmaps. I do the drawing in a function I assign to the `OnPaint` event of the `TPaintBox`. This function is shown in **Listing B**.

I've created a simple bitmap image in the demo program that consists of red and white horizontal stripes. The handle to the image is `transHandle`. The demo program allows you to tell `TransparentBlt()` which of the two colors, red or white, should be made transparent.

AlphaBlend

The `AlphaBlend()` function is a little more complicated than `TransparentBlt()`, but it is very powerful. **Figure B** shows the demo program with an image displayed using this function.

Figure B

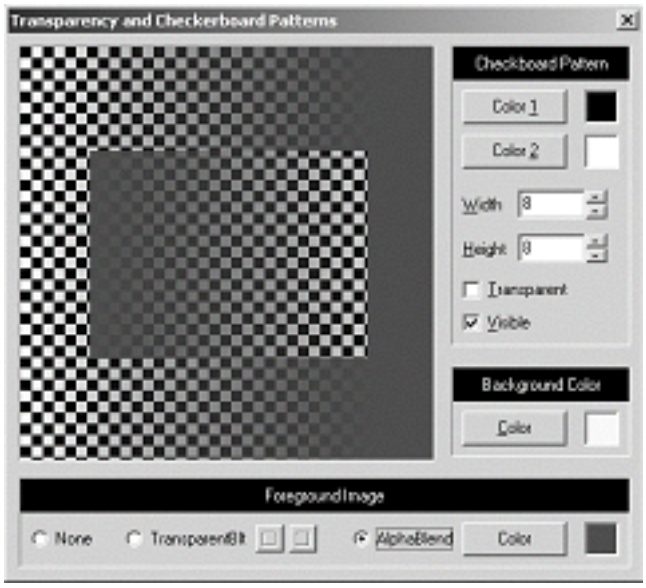


Image displayed with AlphaBlend() function.

Here is the declaration for `AlphaBlend()`:

```
BOOL AlphaBlend(
  // handle to destination DC
  HDC hdcDest,
  // x-coord of upper-left corner
  int nXOriginDest,
  // y-coord of upper-left corner
  int nYOriginDest,
  // destination width
  int nWidthDest,
  // destination height
  int nHeightDest,
  // handle to source DC
  HDC hdcSrc,
  // x-coord of upper-left corner
  int nXOriginSrc,
  // y-coord of upper-left corner
  int nYOriginSrc,
  // source width
  int nWidthSrc,
  // source height
  int nHeightSrc,
  // alpha-blending function
  BLENDFUNCTION blendFunction
);
```

This function uses a 32-bit image consisting of four 8-bit channels for red, green, blue and *alpha*. The alpha channel value is the amount of transparency. A value of 0 is totally transparent and a value of 255 is totally opaque. Values in between are semi-transparent.

The Windows API declares the structure, `RGBQUAD`, which will hold this 32-bit value. The `rgbReserved` member of `RGBQUAD` holds the alpha channel value.

I want to be clear about how powerful this function is. Every pixel in the image can have one of over 16-million colors *and* 256 different levels of transparency. This means, unlike `TransparentBlt()`, that you can have the same color of red that is transparent in some parts of the image and opaque or semi-transparent in other parts.

So, how do we use `AlphaBlend()`? The actual function call is very similar to the standard `StretchBlt()` function; but we'll have to do some preparatory work before calling it.

First, we have to create an image using pixels in the `RGBQUAD` format. **Listing C** is the function I use in the demo program to create such an image. I used a Windows `DIBSECTION` to make things easy. I just drew two squares, one inside the other, consisting of one color, but varying the transparency across each of the squares horizontally and in opposite directions. The demo program will allow you to select the color to use.

There is one tricky part to creating an image to pass to `AlphaBlend()`. The red, green and blue channels of each pixel must be pre-multiplied by a factor equal to the alpha channel value divided by 255. This requirement is a limitation of the `BLENDFUNCTION` structure used as the last argument when calling `AlphaBlend()`. I'll talk more about `BLENDFUNCTION` shortly.

Let's take a look at how to do the pre-multiplication. Every pixel in the image must have the pre-multiplication performed on it. Remember, each pixel is represented by an `RGBQUAD` structure; here's some example code to perform the pre-multiplication:

```
RGBQUAD oldPixel, newPixel;

newPixel.rgbBlue = (oldPixel.rgbBlue *
  oldPixel.rgbReserved) / 255;
newPixel.rgbGreen = (oldPixel.rgbGreen *
  oldPixel.rgbReserved) / 255;
newPixel.rgbRed = (oldPixel.rgbRed *
  oldPixel.rgbReserved) / 255;
newPixel.rgbReserved =
  oldPixel.rgbReserved;
```

The last argument to the `AlphaBlend()` function is a `BLENDFUNCTION` structure. This structure tells

`AlphaBlend()` how to blend the source and destination bitmaps. You should look at the Microsoft Web site for a complete description of how `BLENDFUNCTION` works.

Fading away

Using `TransparentBlt()` and `AlphaBlend()` makes drawing transparent images in Windows relatively easy. Remember that you can't use these functions with all versions of 32-bit Windows.

Depending on your alpha value, I may see you later.

Listing A: *The DrawCheckerboard() function.*

```
void TMainForm::DrawCheckerboard(
  TCanvas *canvas,
  int boardWidth, int boardHeight,
  TColor color1, TColor color2,
  int checkerWidth, int checkerHeight)
{
  TColor startColor = color1;
  for (int row = 0; row < boardHeight;row++) {
    if (height != 0 && row % height == 0) {
      startColor = startColor == color1
        ? color2 : color1;
    }

    TColor color = startColor;
    for (int col = 0; col < boardWidth;col++) {
      if (width != 0 && col % width == 0) {
        color = color == color1
          ? color2 : color1;
      }

      canvas->Pixels[col][row] = color;
    }
  }
}
```

Listing B: *Function assigned to the OnPaint event of TPaintBox.*

```
void __fastcall TMainForm::PaintBoxPaint(
 TObject *Sender)
{
```

```
  int cx = BkgdPanel->ClientWidth;
  int cy = BkgdPanel->ClientHeight;

  HDC hDC = CreateCompatibleDC(
    PaintBox->Canvas->Handle);
  if (hDC == 0)
    throw Exception("Whoops!");

  if (TransRadio->Checked) {
    if (SelectObject(hDC, transHandle) == 0)
      throw Exception("Whoops!");

    TransparentBlt(PaintBox->Canvas->Handle,
      0, 0, cx, cy, hDC, 0, 0, cx, cy, transColor);
  }
  else {
    if (SelectObject(hDC, alphaHandle) == 0)
      throw Exception("Whoops!");

    BLENDFUNCTION blend =
      {AC_SRC_OVER, 0, 255, AC_SRC_ALPHA};
    AlphaBlend(PaintBox->Canvas->Handle,
      0, 0, cx, cy, hDC, 0, 0, cx, cy, blend);
  }

  if (hDC) DeleteDC(hDC);
}
```

Listing C: *Creating a bitmap for the AlphaBlend() function.*

```
void TMainForm::CreateAlphaImage()
{
  int cx = BkgdPanel->ClientWidth;
  int cy = BkgdPanel->ClientHeight;

  if (alphaHandle)
    DeleteObject(alphaHandle);
  alphaHandle =
    GetDIBSection(cx, cy, &alphaImage);

  RGBQUAD of, nf;
  of.rgbBlue = (alphaColor & 0x00FF0000) >> 16;
  of.rgbGreen = (alphaColor & 0x0000FF00) >> 8;
  of.rgbRed = (alphaColor & 0x000000FF);
```

```
  of.rgbReserved = 0;

  for (int r = 0; r < cy; r++) {
    for (int c = 0; c < cx; c++) {
      int x = min(c, 255);
      nf.rgbBlue = (of.rgbBlue * x) / 255;
      nf.rgbGreen = (of.rgbGreen * x) / 255;
      nf.rgbRed = (of.rgbRed * x) / 255;
      nf.rgbReserved = x;
      alphaImage[r * cx + c] = nf;
    }
  }

  for (int r = 75; r < cy - 75; r++) {
    for (int c = 50; c < cx - 50; c++) {
      int x = min(cx - c, 255);
      nf.rgbBlue = (of.rgbBlue * x) / 255;
      nf.rgbGreen = (of.rgbGreen * x) / 255;
      nf.rgbRed = (of.rgbRed * x) / 255;
      nf.rgbReserved = x;
      alphaImage[r * cx + c] = nf;
    }
  }
}
```

Owner-drawn list boxes

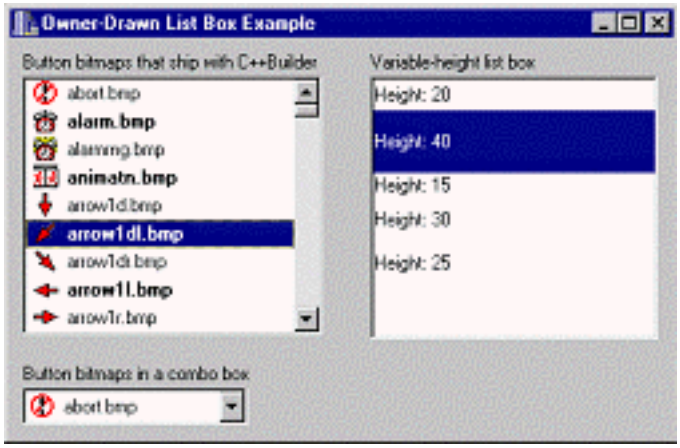
by Kent Reisdorph

I frequently see users of the Borland newsgroups asking, "How do I show certain items in a list box as disabled?" Or, "How do I make items in my list box different colors?" The answer to both of these questions is the same: Use an owner-drawn list box.

Fortunately the VCL makes it relatively easy to create owner-drawn list boxes and combo boxes. You need to understand the basics of how owner-drawn controls work, but once you do, the rest is easy. By owner-drawing list boxes and combo boxes you can display items in different colors, display bitmaps in the control, or even create highly customized list boxes for specialty applications such as games.

This article will explain how to create owner-drawn list boxes and combo boxes. I will focus on list boxes, but everything you read will apply to combo boxes as well. The example program for this article displays two owner-draw list boxes, and one owner-drawn combo box as shown in **Figure A**.

Figure A



The example program shows how to create owner-drawn list boxes and combo boxes.

The VCL owner-draw mechanism

As I have said, the VCL makes it easy to owner-draw list boxes and combo boxes. It is not necessary for you to write specialized components in order to have owner-drawn list boxes. You only need to set a property or two and respond a couple of VCL events. Creating an owner-drawn list box requires these steps:

I.Set the `Style` property to either `lbOwnerDrawFixed` or `lbOwnerDrawVariable`.

II.Create an event handler for the `OnDrawItem` event and write the drawing code in that event handler.

III.Optionally, create an event handler for the `OnMeasureItem` event.

The steps themselves are easy, but writing the code that draws the list box items can be a bit challenging at first. I will explain what is required for each of these steps in the following sections.

Fixed or variable height?

The items in a list box or combo box can all be the same height (owner-draw fixed) or each item can have a different height (owner-draw variable). The type of the list box is controlled through the `Style` property. For a fixed-height list box, set the `Style` property to `lbOwnerDrawFixed`. For a variable-height list box, set the `Style` to `lbOwnerDrawVariable`.

Fixed-height list boxes are much more common than variable-height list boxes. After you set the `Style` property to `lbOwnerDrawFixed`, you must provide an event handler for the `OnDrawItem` event. `OnDrawItem` will be fired for every item in the list box that requires painting. I'll discuss the `OnDrawItem` event in the next section. For fixed-height list boxes you must set the `ItemHeight` property to the desired height of the list box items. The `OnMeasureItem` event is not used for fixed-height list boxes and, in fact, is never generated for fixed-height list boxes.

Variable-height list boxes require only slightly more work than fixed-height list boxes. The additional work comes in the form of telling Windows the exact height of each list box item. This is done through the `OnMeasureItem` event. There are a number of ways in which you might determine the item height. That process is implementation-specific so I won't go into the possibilities here. Instead, I will show you an example of setting the item height by brute force using a `switch` statement. The `OnMeasureItem` event handler looks like this:

```
void __fastcall
TMainForm::ListBox2MeasureItem(
  TWinControl *Control,
  int Index, int &Height)
{
  switch (Index) {
    case 0 : Height = 20; break;
    case 1 : Height = 40; break;
    case 2 : Height = 15; break;
    case 3 : Height = 30; break;
    case 4 : Height = 25; break;
  }
}
```

The `Index` parameter tells you the item for which Windows is requesting the height. The `Height` parameter is used to pass the height of the item back to the VCL (which, in turn, passes the information on to Windows). The example program for this article contains a variable-height owner-drawn list box (refer back to **Figure A**). You can download and examine the code to see how to implement a variable-height list box.

Drawing the items

The challenging part of owner-drawn list boxes is in writing the code that draws each item. You are responsible for drawing each list box item in its entirety (the VCL handles drawing of the focus rectangle for selected items but you must do the rest of the drawing yourself). There are several aspects to consider when drawing the items. For example, the item needs to be drawn one way when the item is not selected, and drawn another way when the item is selected. First, take a look at an empty `OnDrawItem` event handler and then I'll explain further:

```
void __fastcall
TMainForm::ListBox1DrawItem(
  TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)
{
}
```

The `Control` parameter is a pointer to the control that generated the event. You can use the `Control` parameter if you want to use the same `OnDrawItem` event handler for multiple controls (list boxes or combo boxes). The `Index` parameter is the index number of the item that needs to be drawn. The `Rect` parameter is the clipping rectangle for the item. Any drawing you do must be within this rectangle. The `State` parameter tells you whether the item is selected or not selected, among other things. The `State` parameter is important in that it tells you whether you need to draw the item as selected or not selected.

Drawing the items is itself a multi-step process. I'll break down those steps into sections to make it easier to digest.

Selecting the colors

One of the first things you'll need to do is determine the colors needed to draw the background and the text of the item. This is essentially a two-step process. The first step is to determine if the item is to be drawn selected or normal. The second step is to ask Windows for the colors you'll need to drawn the items. Here's the code:

```
TColor backColor;
TColor textColor;
if (State.Contains(odSelected)) {
  backColor = (TColor)
    GetSysColor(COLOR_HIGHLIGHT);
  textColor = (TColor)
    GetSysColor(COLOR_HIGHLIGHTTEXT);
}
else {
  backColor = (TColor)
    GetSysColor(COLOR_WINDOW);
  textColor = (TColor)
    GetSysColor(COLOR_WINDOWTEXT);
}
```

This code first determines if the `odSelected` value is in the `State` parameter (the `State` parameter is a set). If so, the background color is set to Windows' `COLOR_HIGHLIGHT` value, and the text color is set to Windows' `COLOR_HIGHLIGHTTEXT` value. If `odSelected` is not in the `State` parameter then the Windows colors for the window background and window text are used.

It is important to get the color values from Windows. It is important because you want to be sure that you draw the list box according to the user's color preferences. If you simply hard code colors into your drawing routine, your application will look odd on systems where the user has changed the default color settings.

Drawing the background

Drawing the background is trivial, but still important. You must "erase" the background of the item prior to drawing any text or graphics you want displayed for the item. Erasing the background is really a matter of filling the item's drawing rectangle with the background color (as determined by the previous step). Here is the code:

```
ListBox1->Canvas->Brush->Color =
  backColor;
ListBox1->Canvas->FillRect(Rect);
```

Remember, the `Rect` parameter of the `OnDrawItem` event handler is the rectangle within which you can draw. This code simply sets the list box's brush color to the background color and draws a rectangle with that color.

Drawing the text

Almost every list box contains text. Naturally, you will have to draw the text for each list box item. The first step is to determine what text you want displayed. Depending on how you have stored the item information, you may display all of the text in the item, part of the text, or get the text from an external source. This is one of the advantages to owner-drawn list boxes. The data stored in each item may or may not be textual. For example, you may choose to store a class instance in the `Object` property of the list box items and obtain the display text from that class. Let me take a moment to explain.

Let's say that you were creating a list box that displayed bitmaps along with a description of the bitmap. You might create a class that handles the bitmap, contains the display text, and manages other details of the list box item. When you fill the list box, you would add pointers to each object to the `Objects` property of the list box items. You could then extract the description string from the object before drawing the list box item. There are a multitude of methods you could use to obtain the display text and this is just one example. The point is that with owner-drawn list boxes you are in control of what the display text is and where it comes from.

After you have determined the text you will draw, the actual drawing of the text can be accomplished in several ways. One way is to use `TCanvas`'s `TextOut()` or `TextRect()` methods. Another, more flexible, way is to use the Windows API function `DrawText()`. I recommend `DrawText()` because it gives you full control over how the text is drawn within the clipping rectangle. The text can be centered horizontally or vertically, can be aligned left or right, can be single line or multi-line, and can even be displayed with an ending ellipsis if the text is too wide for the list box. See the `DrawText()` item in the Win32 API help for a complete listing of the drawing options available. **Listing A** shows the `OnDrawItem` event handler for the example program's main list box. Examine the code to see how the item text is drawn.

Anything goes

Essentially, you can draw anything you want in your owner-drawn list boxes. You can use the list box's `Canvas` to do your drawing, use the Windows API, or any combination of the two. The example program for this article contains both a list box and a combo box that show all of the button bitmaps that ship with C++Builder and the file name of each bitmap (assuming you installed the bitmaps when you installed C++Builder). Refer to **Listing A** for the code that draws the bitmaps and the item text. Note that the text is displayed to the right of the bitmap, and centered within the list box item.

It is probably not clear from the code how I obtain the display text and the bitmap's filename. When I fill the list box I create item text that is structured like this:

```
abort.bmp=c:\images\buttons\abort.bmp
```

This format follows the `TStringList` class's `Name=Value` mechanism. The first part of the string

(the `Name`) is the display text for the list box item. The second part of the string (the `Value`) is the actual path and filename of the bitmap file. I extract these two portions of the string and store them in local variables for user later in the function. Once again, I refer you to **Listing A** to see how the item's display text is extracted from the item text stored in the list box.

Conclusion

Creating owner-drawn list boxes and combo boxes is not difficult once you understand the principles. The full source for this article's example program can be downloaded from our Web site at **www.bridgespublishing.com**. The example shows how to both draw fixed and variable-height list boxes. As an added bonus, it also shows how to enumerate all the files in a directory using the Windows API functions `FindFirstFile()` and `FindNextFile()`. Owner-drawn list boxes can be used to convey application-specific information to the user in one neat package.

Listing A: ***The Example Program's OnDrawItem Event Handler***

```
void __fastcall
TMainForm::ListBox1DrawItem(TWinControl *Control,
  int Index, TRect &Rect, TOwnerDrawState State)
{
  // Get the name of the file.
  // This will be used in the list box display.
  String name = ListBox1->Items->Names[Index];

  // Get the actual filename from the Values
  // property. We'll use this filename to load
  // the bitmap for display.
  String filename = ListBox1->Items->Values[name];

  TColor backColor;
  TColor textColor;

  // Set up the colors used to draw the background
  // and the text.If the item is selected then use
  // the Windows selection colors.
  if (State.Contains(odSelected)) {
    backColor =
      (TColor)GetSysColor(COLOR_HIGHLIGHT);
    textColor =
      (TColor)GetSysColor(COLOR_HIGHLIGHTTEXT);
  }
  // Item not selected so use the regular
```

```
// Windows colors.
else {
  backColor =
    (TColor)GetSysColor(COLOR_WINDOW);
  textColor =
    (TColor)GetSysColor(COLOR_WINDOWTEXT);
}
// Bold the odd-numbered items just for show.
if (Index % 2 == 1)
  ListBox1->Canvas->Font->Style =
    TFontStyles() << fsBold;
else
  ListBox1->Canvas->Font->Style = TFontStyles();

// Fill drawing rect with the background color.
ListBox1->Canvas->Brush->Color = backColor;
ListBox1->Canvas->FillRect(Rect);

// Get the image.
Graphics::TBitmap* bm = new Graphics::TBitmap;
bm->LoadFromFile(filename);

// The button images are actually 16x32 pixels.
// However, we only want to show the first part of
// the image (16x16) so we'll modify the source
// rectangle accordingly.
TRect src;
src.Left = 0;
src.Top = 0;
src.Right = bm->Height;
src.Bottom = bm->Height;

// Set up the destination rectangle.
TRect dst;
dst.Left = 5;
dst.Top = Rect.Top + 1;
dst.Right = dst.Left + bm->Height;
dst.Bottom = Rect.Top + bm->Height;

// Draw the image using BrushCopy(). We use
// rushCopy() because it allows us to copy part
// of an image, with transparency.
ListBox1->Canvas->BrushCopy(
  dst, bm, src, bm->TransparentColor);
```

```
  // Done with the bitmap so delete it.
  delete bm;

  // Set the color used to draw the text.
  ListBox1->Canvas->Pen->Color = textColor;

  // We'll draw the text 5 pixels to the right of
  // the image so adjust the drawing rectangle's
  // Left by the width of the bitmap plus 5 pixels.
  Rect.Left = dst.Right + 5;

  // Draw the text. We use DrawText so we can
  // vertically center the text in the list box item.
  DrawText(ListBox1->Canvas->Handle, name.c_str(),
    -1, &Rect, DT_SINGLELINE | DT_VCENTER);
  return;
}
```

Ownership vs. parentage

by Kent Reisdorph

There are two elements you should understand when writing VCL applications; ownership and parentage.

Ownership is the mechanism by which the VCL facilitates memory management. Ownership is determined by a form or component's `Owner` property. Consider the case of a typical application where all forms are auto-created at application startup. When the VCL auto-creates the forms, it assigns the `Application` object as the owner of all forms. When the application is closed, the `Application` object runs through its list of owned forms and deletes each form. Similarly, a form owns all the components placed on the form. As each form is deleted, it deletes all of the components on the form. Ownership is specific to the class library—the VCL in this case. Windows itself does not have support for the concept of ownership.

Parentage deals with visual components contained on a form (or within another component such as a panel). Parentage is a Windows concept, not a VCL concept. Parentage is controlled in the VCL via the `Parent` property. Every windowed control (visual component) must have a parent. The parent window is the window that hosts a control. When a button is placed on a form, the form becomes the button's parent. The `Top` and `Left` property of the button are specified relative to the parent's top left corner.

To summarize, ownership deals with memory management, and parentage deals with the parent/child relationship between windows. Understanding these two terms and their roles in the VCL will help you when writing applications and components.

Under construction

By Mark G. Wiseman

I am going to show you two functions that I use in nearly every program I write. Two functions that are never included in any program I distribute. Well, actually, the two functions are really different, over-loaded version of one function, `UnderConstruction()`.

These two functions will display a message box telling you that something in your program—usually event code—is incomplete. They will also make an attempt to tell you what object called them.

Breaking ground

When you add a component such as a `TButton`, `TMenuItem` or `TAction` to a VCL project, you need to provide code for an event of that component before it becomes useful. Normally, you would write code for the `OnClick()` event of `TButton` and `TMenuItem` and for the `OnExecute()` event of `TAction`.

I find that when I start to design and lay out a form or menu, I already know what buttons and menu items I want to use and I know the names of the event functions I want to associate with these. However, I don't necessarily want to stop and write all the code required to make each of these event functions operational.

I could just leave the event code out and go back and fill it in later. However, there are two problems with this approach.

First, when testing a program that is under construction, menu items, buttons etc. that have no code attached will just do nothing when selected. They may also appear to do nothing even if they have events coded. How do you tell the difference? Using the `UnderConstruction()` functions, you know instantly that this event is not working, because the code has not been completed.

Second, when using a control associated with a `TAction` item, (e.g. a `TmenuItem`), the control will be disabled if there is no event code for the `OnExecute()` event of `TAction`. Again, it is difficult to tell if a control is disabled because there is no event code or for some other reason. The `UnderConstruction()` function enables the item and lets you know that the code is incomplete.

The blueprint

Listings A and B contain the source code for the `UnderConstruction()` functions. The first

version of `UnderConstruction()` takes a pointer to a `TObject` as a required parameter and an optional argument of an `AnsiString`. It tries to identify the object by casting it to a `TControl` and reading the `Name` property of the `TControl`. It tries to further identify any `TAction`, `TButton`, or `TMenuItem` by reading that controls `Caption` property. If the optional `AnsiString` parameter was used, `UnderConstruction()` appends this string to the control's identification string and calls the second version of `UnderConstruction()` with the identification string as its argument.

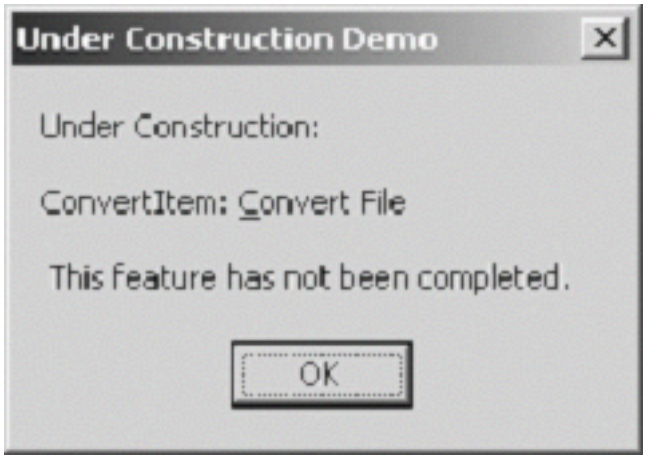
The second version of `UnderConstruction()` displays a message box with its `AnsiString` string argument as the bulk of the text in the message box.

Let's say we're designing a form, `TMainForm`, for a new project. We add a menu item named "ConvertItem" with the caption "Convert File". If we generate the `OnClick()` event code for the item, we will get the empty shell for a function named `ConvertItemClick()`. However, some one else is writing the code for this event. So, let's use the `UnderConstruction()` functions as a placeholder. Here's what the event code will look like:

```
void __fastcall TMainForm::
  ConvertItemClick(TObject *Sender)
{
  UnderConstruction(Sender);
}
```

When this menu item is selected, the message box shows in **Figure A** will be displayed.

Figure A



Under Construction message box

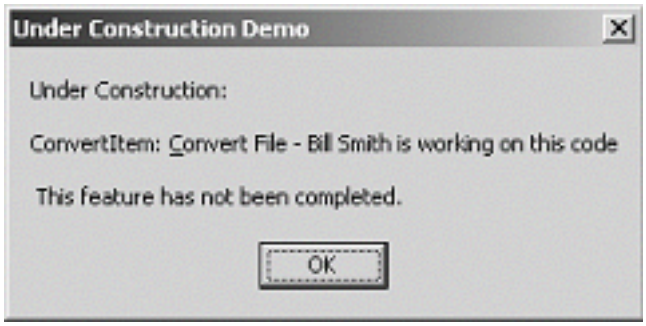
Let's use the optional string parameter of `UnderConstruction()` to provide even more information:

```
void __fastcall TMainForm::
```

```
  ConvertItemClick(TObject *Sender)
{
  UnderConstruction(Sender,
    "Bill is working on this code");
}
```

This code will produce the message box shown in **Figure B** when the menu item is selected.

Figure B



Under Construction message box using additional AnsiString parameter

Project cleanup

Obviously, the `UnderConstruction()` functions should only be used during debugging and testing—while your program is under construction. That's why I say that I use them in every program I write but they are not included in any program I distribute (except, of course, for the demo program for this article). The program can be found on the Bridges Publishing Web site.

For these functions to work, you must first define the macro `_UNDERCONSTRUCTION`. The best place to do this is in the Conditional Defines setting of the Directories/Conditionals tab of the Project Options dialog. When this macro is defined, the `UnderConstruction()` functions work as described in this article. Before you do the final build of your project, remove the `_UNDERCONSTRUCTION` macro definition. If you have accidentally left a call to `UnderConstruction()` in your code, you will get a compiler error.

Construction complete

You could add code to the first version of `UnderConstruction()` to provide information for more types of VCL objects. For instance, it might be useful to include code to provide the name of the `TDataSet` in events such as `BeforePost()`. However, I've found `UnderConstruction()` to be a very useful function just as it's written.

Listing A: *UnderConstruction.h*

```
#ifndef UnderConstructionH
#define UnderConstructionH

#ifdef _UNDERCONSTRUCTION

void __fastcall UnderConstruction(
  TObject *Sender, const String &msg = "");

void __fastcall UnderConstruction(
  const String &feature = "");

#endif   // _UNDERCONSTRUCTION

#endif   // UnderConstructionH
```

Listing B: *UnderConstruction.cpp*

```
#ifdef _UNDERCONSTRUCTION

#include <vcl.h>
#pragma hdrstop

#include "UnderConstruction.h"

void __fastcall UnderConstruction(
  TObject *Sender, const String &msg)
{
  String feature;

  TComponent *component =
    dynamic_cast<TComponent *>(Sender);
  if (component != 0)
    feature += component->Name;

  TAction *action =
    dynamic_cast<TAction *>(Sender);
  if (action)
    feature += ": " + action->Caption;

  TMenuItem *item =
    dynamic_cast<TMenuItem *>(Sender);
  if (item)
```

```
    feature += ": " + item->Caption;

  TButton *button =
    dynamic_cast<TButton *>(Sender);
  if (button)
    feature += ": " + button->Caption;

  if (msg.IsEmpty() == false)
    feature += " - " + msg;

  UnderConstruction(feature);
}

void __fastcall UnderConstruction(
   const String &feature)
{
  String msg = "Under Construction:";
  if (feature.IsEmpty() == false)
    msg += "\r\n\r\n" + feature;
  msg += "\r\n\r\n This feature has "
    "not been completed.";

  ShowMessage(msg);
}

#pragma package(smart_init)

#endif   // _UNDERCONSTRUCTION
```

Using persistent fields

by Mark Cashman

Persistent fields are, of course, useful for lookup and calculated fields, but you can also use them to avoid references such as:

```
S = Account->Status->
  FieldByName("DESCRIPTION")->AsString;
```

This type of reference might be vulnerable to a DBMS change and is best avoided. Using persistent fields allows you to instead to use code like this:

```
S = Account->Status->
  StatusDescription->AsString;
```

Note that even if you rename the field object from its default, it must have a name unique in the data module. Otherwise we could refer `to`

```
S = Account->Status->Description->AsString;
```

Adding the table name to the field name is probably the least objectionable way to deal with this problem.

There are other important factors in the use of persistent fields. First, use of persistent fields is independent of the sequence of fields in the table. As a result, reordering those fields has no effect on your use of the fields (though renaming and deleting does). Second, you can control the display of a table field in a grid or elsewhere by setting the `Visible` property of the field (which controls whether of not the field is shown in a grid), its `EditMask`, its `Alignment`, its `DisplayWidth`, and its `DisplayLabel`. Third, you can attach events to the field to deal with new or changed content (in the `OnSetText` and `OnValidate` event handlers, for example). Fourth, references through persistent fields are very efficient compared to `FieldByName`, since a lookup of the field name and determination of its status and position in the row does not need to be done on each reference to the field.

June 1999

Drawing with metafiles

by Thomas Wieland

All graphic control components in the VCL provide the Canvas property a drawing surface for objects that draw images of themselves. In contrast to the window controls, Windows doesn't repaint such controls automatically, when, for example, the window pops up from the background. If you want to keep the content of the canvas highly dynamic, your program has to do the necessary bookkeeping of the drawing commands itself. In this article, we'll explain how to solve this problem by the metafile mechanism that easily records the necessary commandsjust like a VCR such that you only have to replay the tape, so to speak, to (re)paint the graphics.

Drawing on a canvas

The simplest graphical control is the PaintBox. It consists of hardly anything other than the canvas. In a window containing this component, you can make any drawings you want using all the operations offered by the TCanvas class. But if your window is overlapped by another for a moment, your canvas will be empty afterwards, at least in the regions that were covered. Everything you've drawn there so far will disappear. Thus, when you work with a PaintBox, you must always provide a handler for the OnPaint event. This is an encapsulation of the Windows WM_PAINT message and occurs every time the box needs to be repainted. In the worst case, this can happen quite a number of times during the runtime of your application. If your image isn't static, but has to be constructed by some calculations or from the users input, for example, you should store the rules for the drawing. Then you can perform the repainting fast enough.

Windows metafiles

In general, to save some graphics (to disk or just to memory), there are two different ways. You can save them pixel by pixel and get a bitmap. However, if you only save the rules for the drawing, (that is, the shape, filling, coordinates, etc.) you'll have a vector graphic, also known as a metafile. In C++Builder, metafiles are represented by the TMetafile class, which we'll use here. The only question is how to get the drawing from a canvas into an object of this classand vice versa.

Drawing into metafiles

The trick is done by a second class called TMetafileCanvas. This builds the bridge between the canvas and the metafile. Instances of this class can paint the contents of a metafile on a real canvas, but are also able to store the commands that make the image. The usage consists of the following steps:

- Declare a variable of type TMetafile* in your form class. Create a metafile object at the beginning of your application, that is, in the constructor of the form:

```
aMetafile = new TMetafile;
```

- To draw something that should be saved in the metafile, you must create a metafile canvas. The simplest way to do this is with the command:

```
mfCanvas = new TMetafileCanvas(aMetafile, 0);
```

- Now you can draw objects on your mfCanvas just like on any other canvas. The differences are that the picture is only created in memory and not on the screen, and that all commands are recorded.

- After your drawing is complete, you should always add the command delete mfCanvas. The recording ends when the mfCanvas object is destroyed. This is also the moment when the information is transmitted to the metafile.

- Now you may replay the recorded operations easily. Every canvaslike the one in a PaintBoxhas a Draw method that takes the metafile and the coordinates of the upper-left corner and starts rendering. For example:

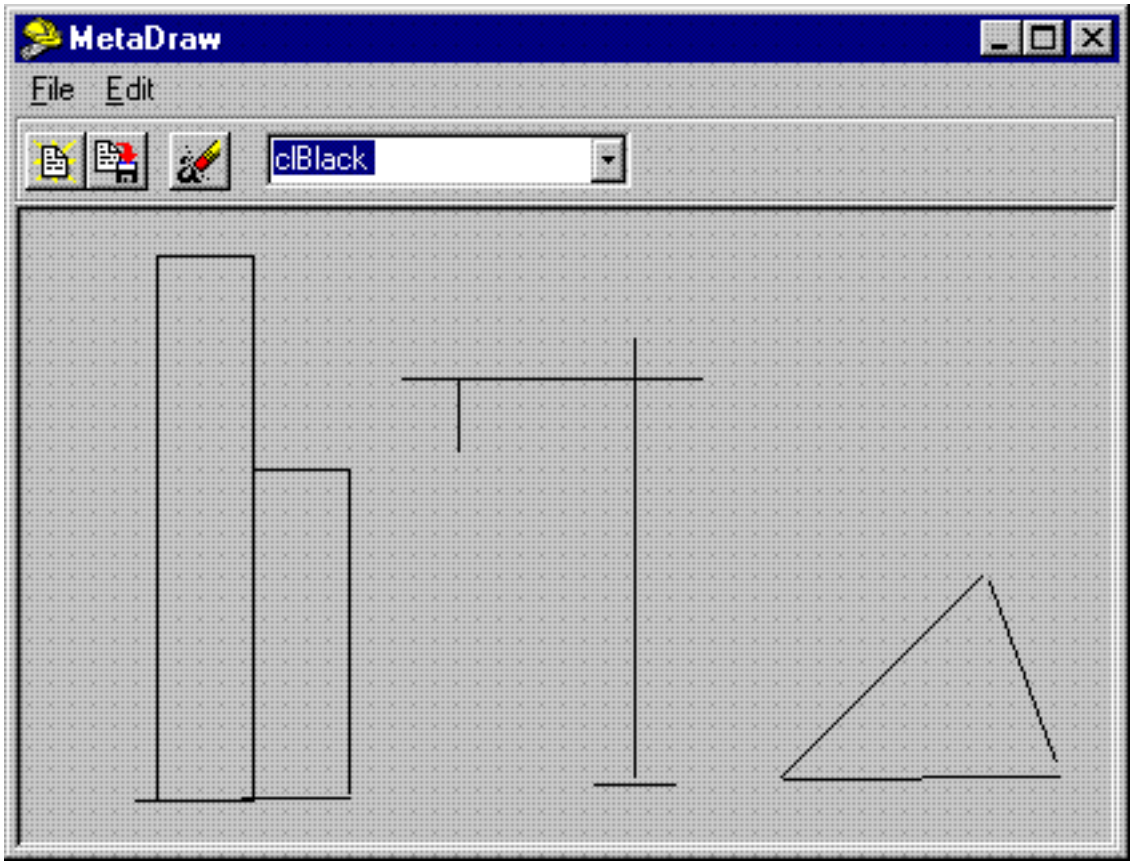
```
PaintBox->Canvas->Draw(0, 0, aMetafile);
```

When following these steps, you always begin with an empty canvas and, thus, ignore everything that might already be in the metafile. To add something to an existing content, you first draw the metafile onto the mfCanvas object before drawing anything else.

Freehand drawing in a PaintBox

Now let's take a look at a little application that makes use of this concept. The program, called MetaDraw, allows users to draw lines on a free space. They can select the color, as well as save, load, and clear the drawing.

Figure A: The MetaDraw application comprises just a PaintBox and a ToolBar.



We create the metafile object in the constructor of the main form, like we did in step 1. In addition, we initialize a couple of internal variables:

```
__fastcall TDrawForm::TDrawForm(
        TComponent* Owner) : TForm(Owner)
{
        x_start = 0;
        y_start = 0;
        mouse_down = false;
        cbColor->ItemIndex = 1;

        aMetafile = new TMetafile;
        aMetafile->Width = PaintBox->Width;
        aMetafile->Height = PaintBox->Height;
        Application->ShowHint = true;
}
```

The crucial events of the application are the pressing of the mouse button and the mouse movement. When the users press the button, we remember the position of the pointer and move the pen there. The following code illustrates this:

```
void __fastcall
```

```
        TDrawForm::PaintBoxMouseDown(
        TObject *Sender, TMouseButton Button,
        TShiftState Shift, int X, int Y)
{
        // Remember position.
        x_start = X;
        y_start = Y;
        x_prev = x_start;
        y_prev = y_start;

        // Move pen to this position.
        PaintBox->Canvas->MoveTo(X,Y);

        // Remember pressing of mouse button.
        mouse_down = true;
}
```

When the users move the mouse pointerwhen the OnMouseEvent occurswe delete the current line and draw a new one, so they can always see what kind of line they just drew. As this can only happen during direct user interaction, we can paint directly on the form.

The interesting part comes when the users release the mouse button. Then we regard the present act of drawing as complete and save it to our metafile. The entire routine is given in Listing A; it follows the procedure proposed above.

Listing A: OnMouseUp event

```
void __fastcall
TDrawForm::PaintBoxMouseUp
        (TObject *Sender, TMouseButton Button,
         TShiftState Shift, int X, int Y)
{
        mouse_down = false;

        TColor col;
        switch (cbColor->ItemIndex)
        {
                case 0: col = clAqua;
                                break;
                // ...
                // More colors
        }
```

```
        // Create a canvas to draw into the
        // metafile.
        mfCanvas = new TMetafileCanvas(
        aMetafile, 0);

        // Draw existing lines on the canvas.
        mfCanvas->Draw(0,0,aMetafile);

        // Draw the line.
        mfCanvas->Pen->Color = col;
        mfCanvas->MoveTo(x_start,y_start);
        mfCanvas->LineTo(X,Y);

        // Free the canvas.
        delete mfCanvas;

        // Update the window.
        PaintBox->Invalidate();
        PaintBoxPaint(Sender);
}
```

First, we create the metafile canvas and draw all previously saved lines on it. Then, we set the desired color of the line and add the line to the drawing. Finally, we free the canvas to transmit the additions into the metafile and update the screen by setting it to invalid.

This command forces a repainting of the box. To be precise, it causes an OnPaint event in our C++Builder program. In the handler of this event, all we have to do is replay the contents of the metafile:

```
void __fastcall TDrawForm::PaintBoxPaint(TObject *Sender)
{
        PaintBox->Canvas->Draw(0,0,aMetafile);
}
```

Conclusion

The TMetafile class has many convenient features. Certainly most important is the recording and replaying of drawing commands. As you could see from the example, you just have to draw on a metafile canvas instead of the canvas of your graphical component and you can almost forget about organizing and saving your drawing commands. The complete example application shows you even more features of TMetafile. It is, for instance, also very easy to save and load the contents of the canvas to and from disk, respectivelyas the file in the class name already suggests. Go ahead and explore the possibilities to make your programs a little

bit more colorful and unique.

<%@ Language=VBScript %> <% ProductProtected = 1 %>

June 1999

Writing a performance monitor

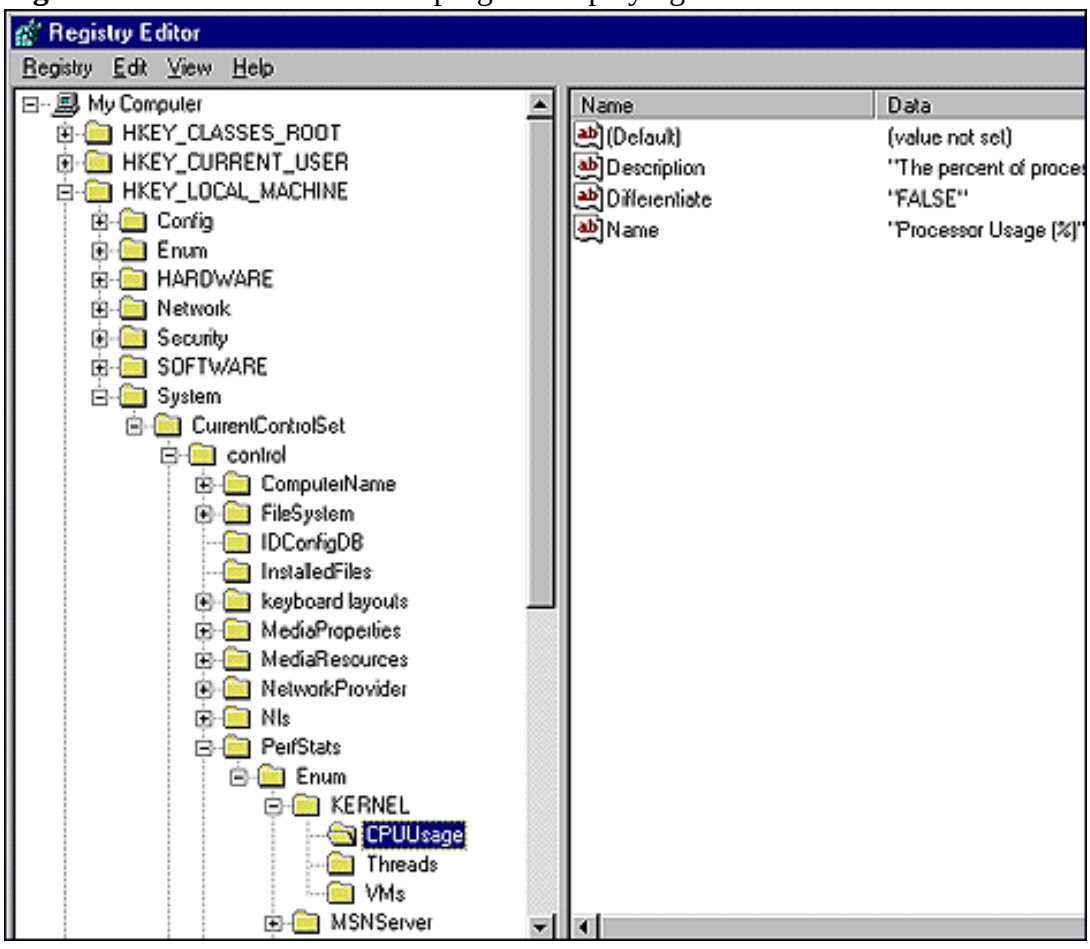
by John M. Miano

The System Monitor utility is well known to developers on Windows 95 and Windows 98. This utility can display various system performance statistics, such as page fault rate and CPU utilization. Applications such as Norton Utilities contain similar performance monitors of their own. So, how can you incorporate performance monitoring in your own applications or custom controls? All you need to do is read the registry. When a Windows device driver loads, it can define any number of statistics that are updated in the registry.

Finding out what statistics are available

Just as the device drivers in use vary from system to system, so do the available performance statistics. The first thing your performance monitoring application will need to do is find out what statistics are available. If you run the REGEDIT program and look under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\PerfStats\Enum, you'll find keys for each server that's generating statistics. The server keys have a value called Name associated with them that gives a translated name for the server key. The subkeys for each server represent the individual statistics. Figure A shows the REGEDIT program displaying the statistics available from KERNEL.

Figure A: This is the REGEDIT program displaying the statistics available from KERNEL.



Each statistic key has three valuesName, Description, and Differentiatethat contain the attributes of the statistic.

Name is a short name for the statistic and *Description* gives a longer description. The value of *Differentiate* tells an application if it needs to calculate a rate from the statistic value. When the value of Differentiate is the string TRUE, the application has to calculate a rate using the previous value of the statistic using

```
display_value = (current_value - last_value) / time_interval
```

The time_interval is the number of seconds between the times the statistics were read from the registry. For statistics where the value of Differentiate is FALSE, the application can simply display the value from the registry.

You can use the Windows API to read the registry. However, I find it easier to use the TRegistry class in VCL. If you use TRegistry, you must explicitly include the file registry.hpp, because it isn't automatically included in VCL applications.

Listing A contains simple command line application that displays the attributes for all performance statistics available on a system. It reads the server names from the HKEY_LOCAL_MACHINE\System\CurrentControlSet\control\PerfStats\Enum key, and the statistics properties from the subkeys.

Listing A: Application to display performance statistic attributes

```
#include <vcl.h>
#include <registry.hpp>
#include <condefs.h>
#include <iostream>
#pragma hdrstop
using namespace std ;

#pragma argsused
int main(int argc, char **argv)
{
  // Open the registry key containing the server/statistic information.
  const String root = "System\\CurrentControlSet\\control\\PerfStats\\Enum" ;
  TRegistry *registry = new TRegistry ;
  registry->RootKey = HKEY_LOCAL_MACHINE ;
  bool status = registry->OpenKey (root, false) ;

  if (! status)
  {
    cerr << "Can't read registry key '" << root.c_str () << "'
      " << endl ;
    return 1 ;
  }

  TStringList *servers = new TStringList ;
  TStringList *statistics = new TStringList ;

  // Read the names of the servers and process each one.
```

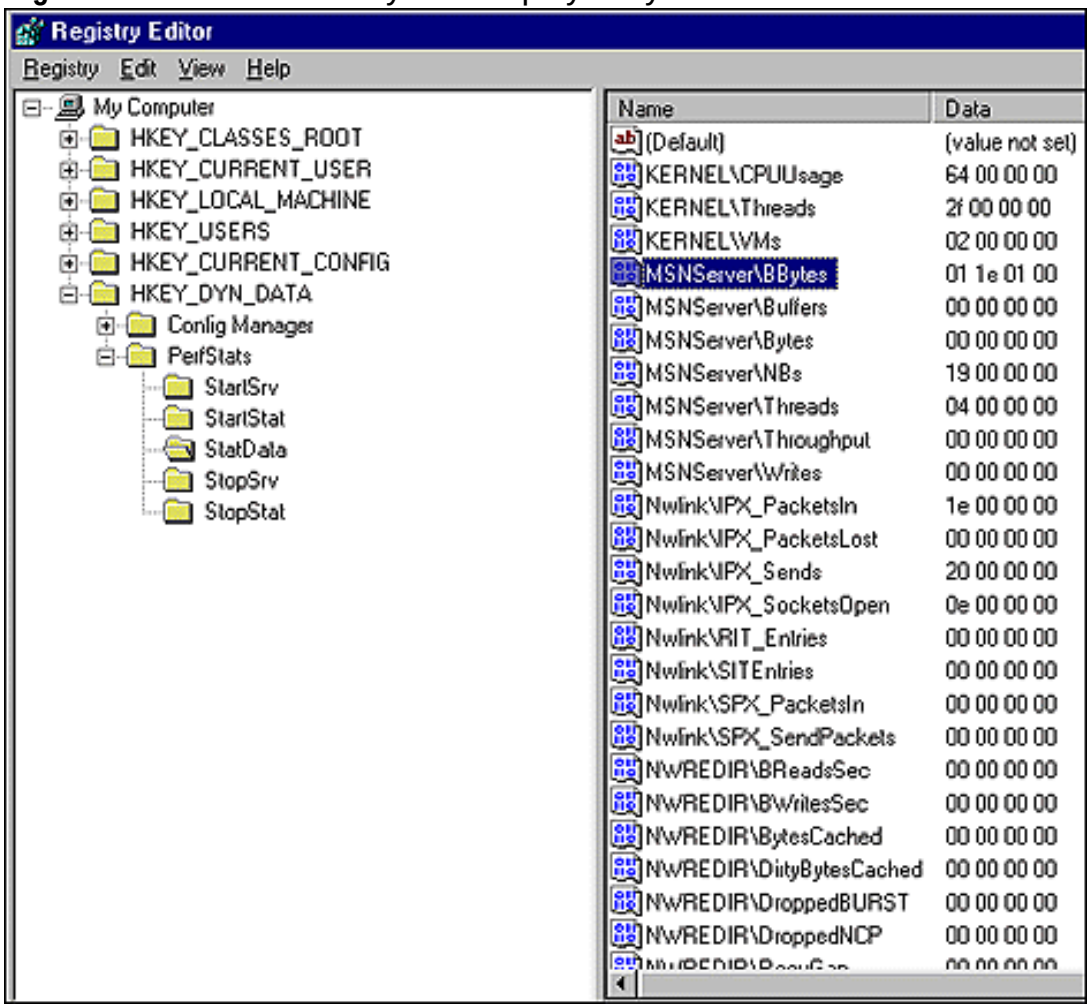
```
registry->GetKeyNames (servers) ;
registry->CloseKey () ;
for (int ii = 0 ; ii < servers->Count ; ++ ii)
{
  // Read the statistics for the for the server.
  String server = servers->Strings [ii] ;
  const String key = root + "\\" + server ;
  bool status = registry->OpenKey (key, false) ;
  if (! status)
  {
    cerr << "Can't read registry key '" << key.c_str () << "'
      " << endl ;
    continue ;
  }
  String name = registry->ReadString ("Name") ;
  registry->GetKeyNames (statistics) ;
  registry->CloseKey () ;

  // Print the server description.
  cout << "Server: " << server.c_str () << endl ;
  cout << "Name:   " << name.c_str () << endl ;

  // Print the descriptions for the server's statistics.
  for (int jj = 0 ; jj < statistics->Count ; ++ jj)
  {
    String statistic = statistics->Strings [jj] ;
    const String key = root + "\\" + server + "\\
      " + statistic ;
    bool status = registry->OpenKey (key, false) ;
    if (! status)
    {
      cerr << "Can't read registry key
        '" << key.c_str () << "' " << endl ;
      continue ;
    }
    String description = registry->ReadString ("Description");
    String name = registry->ReadString ("Name") ;
    String differentiate = registry->ReadString("Differentiate");
    registry->CloseKey () ;
    cout << " Statistic:  " << statistic.c_str () << endl ;
    cout << " Name:       " << name.c_str () << endl ;
    cout << " Description: " << description.c_str () << endl ;
    cout << " Differentiate: " << differentiate.c_str () << endl ;
    cout << endl ;
  }
  cout << endl ;
}
delete servers ;
delete statistics ;
delete registry ;
return 0 ;
```

```
}
```

Getting statistics values

The actual performance data resides under the registry key HKEY_DYN_DATA\PerfStats\StatData. The value names under this key have the form SERVERNAME\STATISTICNAME. For example, the CPUUsage statistic from the KERNAL server is KERNAL\CPUUsage. Figure B shows the statistics keys displayed by REGEDIT. By pressing [F5], you can update the statistics values in the registry, giving you a poor-man's system monitor.

Figure B: The statistics keys are displayed by REGEDIT.



You'll notice that some statistics don't change. KERNEL\CPUUsage just stays at 64 Hex (=100 %). Obviously the CPU usage isn't 100 percent all the time. The reason there's no change is that you have to tell the server to start reporting statistics.

Using REGEDIT, take a look at the values in the keys HKEY_DYN_DATA\PerfStats\StartStat and HKEY_DYN_DATA\PerfStats\StopStat. You'll notice that the values for these keys have the same names as the ones in HKEY_DYN_DATA\PerfStats\StatData. To start collecting a performance statistic, you simply read the value for the statistic you're interested in from the StartData key. Likewise, to turn off a statistic, you read the corresponding key value from the StopData key.

To see the starting and stopping of a performance statistic in operation, use REGEDIT to view the HKEY_DYN_DATA\PerfStats\StatData key. Try refreshing a few times. Notice that there's no change.

Next, start the System Monitor program and use it to view CPUUsage. Go back to REGEDIT while System Monitor is running and refresh. You should see the CPUUsage get updated. After closing the System Monitor, take another look at REGEDIT. The CPUUsage should stop updating shortly.

Once you've started collecting a statistic, you read the actual statistic value from the HKEY_DYN_DATA\PerfStats\StatData key. All of the statistics values are 4-byte integers. However, they're stored in the registry as binary values, so you can't use the TRegistry::ReadInteger method to access them. You must use TRegistry::ReadBinaryData instead.

An application for displaying CPU usage

To illustrate how to gather performance statistics, I've created a simple application that displays the CPU usage percentage found in the KERNEL\CPUUsage key value. This is a statistic that will be present on all systems. Listing B contains the class definition of a form that displays the current CPU usage. I've placed three controls on the form. A TPanel is aligned to the right, a TPerformanceGraph from the Samples page is aligned to the client, and there's a TTimer control, which isn't visible at runtime. To the class definition, I've added a pointer to a TRegistry object and included the file Registry.hpp.

Listing B: CPU.H

```
//---------------------------------------------------
#ifndef CPUH
#define CPUH
//---------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Registry.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include "perfgrap.h"
//---------------------------------------------------

class TForm1 : public TForm
{
  __published:  // IDE-managed Components
    TTimer *Timer;
    TPerformanceGraph *PerformanceGraph;
    TPanel *Panel;
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall FormDestroy(TObject *Sender);
    void __fastcall TimerTimer(TObject *Sender);

    void __fastcall FormResize(TObject *Sender);
  private:      // User declarations
    TRegistry *registry ;
  public:       // User declarations
    __fastcall TForm1(TComponent* Owner);
```

```
};
//---------------------------------------------------

extern PACKAGE TForm1 *Form1;
//---------------------------------------------------

#endif
```

Listing C contains the implementation for the form class. The form reads the registry to start the reporting of the KERNEL\CPUUsage statistic in the OnCreate event. The OnDestroy event reads the registry to stop the performance measurements when the form closes.

The TTimer control is used to update the CPU usage display. The OnTimer event for the TTimer object gets called every second. In this event, the form reads the current CPU usage and displays it.

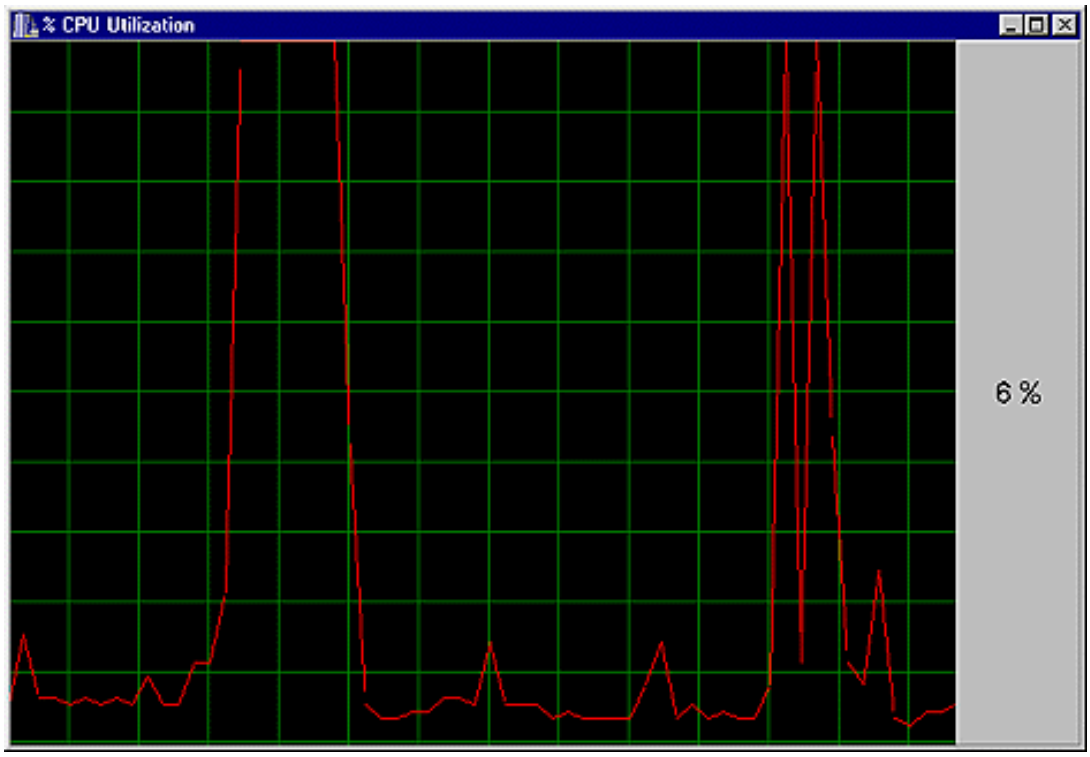
Listing C: CPU.cpp

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "CPU.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "perfgrap"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------

__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner)
{
}
//---------------------------------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  // Allocate and set up the TRegistry object used throughout.
  registry = new TRegistry ;
  registry->RootKey = HKEY_DYN_DATA ;

  // Read the key value that starts the collection of CPUUsage.
  registry->OpenKey ("PerfStats\\StartStat", false) ;
  unsigned int value ;
  registry->ReadBinaryData ("KERNEL\\CPUUsage", (void *) &value, sizeof (value)) ;
  registry->CloseKey () ;
  return ;
}
//---------------------------------------------------------------------

void __fastcall TForm1::FormDestroy(TObject *Sender)
```

```
{
  // Shut down the CPUUsage statistic.
  registry->OpenKey ("PerfStats\\StopStat", false) ;
  unsigned int value ;
  registry->ReadBinaryData ("KERNEL\\CPUUsage", (void *) &value, sizeof (value)) ;
  registry->CloseKey () ;
  delete registry ;
  registry = 0 ;
  return ;
}
//---------------------------------------------------------------
void __fastcall TForm1::TimerTimer(TObject *Sender)
{
  // Get the current CPU usage percentage.
  bool status = registry->OpenKey ("PerfStats\\StatData", false) ;
  unsigned int value ;
  registry->ReadBinaryData ("KERNEL\\CPUUsage", (void *) &value, sizeof (value)) ;
  registry->CloseKey () ;

  // Display the current value.
  PerformanceGraph->DataPoint (clRed, value) ;
  PerformanceGraph->Update () ;
  Panel->Caption = String (value) + " %" ;
  return ;
}
//---------------------------------------------------------------

void __fastcall TForm1::FormResize(TObject *Sender)
{
  // Ensure that the graph always has 10 blocks.
  PerformanceGraph->GridSize = ClientHeight / 10 ;
  return ;
}
//---------------------------------------------------------------
```

The CPU monitor application is shown in Figure C. The left side of the form contains a historical graph of CPU usage and the right side displays the current value.

Figure C: Here's the CPU monitor application.



Conclusion

In this article, we've shown you how to determine which statistics are available to a performance monitor on Windows 95 or 98, how to start and stop statistics gathering, and how to read statistics values. For simplicity, we've split these processes into two simple applications. How you put these pieces together would depend upon the type of application or component you're trying to create.

Version: 1.0, 3.0, 4.0

June 1999

Pinging a server

by Kent Reisdorph

Internet programming provides many challenges. One of those challenges is detecting whether or not your user is connected to the Internet before attempting to connect to a POP3, SMTP, FTP, or other server. More specifically, you might need to check that the particular remote machine to which you're trying to connect is actually online. This article will explain how to ping a remote machine. We'll explain how to obtain components that allow you to perform a ping. Then, we'll show you how to write code to ping a server using the scarcely documented ICMP.DLL.

What's ping?

First of all, PING is a program. One version or another of PING ships with most UNIX systems, as well as with Windows 95/98 and Windows NT. The term *ping* has since been adopted to refer to the process by which a remote machine is sent a packet of data to which it responds. It can be boiled down to, "Hey, machine! Are you there?" To which the remote machine responds, "Yes, I am here," or, if the machine is down or you're not connected to the Internet, doesn't respond at all. The original PING program was written by Mike Muuss more than 15 years ago. Contrary to some statements to the contrary, PING is *not* an acronym. Rather, PING is named after the process used by sonar; a signal is sent out and an echoing signal is received, or not received, in reply. If you want to know more about the PING story, see Mr. Muuss' *The Story of the PING Program* at

http://ftp.arl.mil/~mike/ping.html

To see how PING works, first connect to the Internet, if necessary. Next, open a command prompt box in Windows and type the following:

```
ping www.borland.com
```

If your ISP doesn't have a DNS server running, then you'll need to type the actual IP address of the server:

```
ping 207.105.83.51
```

The PING program will respond with something like the following:

```
Reply from 207.105.83.51: bytes=32 time=91ms TTL=245
Reply from 207.105.83.51: bytes=32 time=70ms TTL=245
Reply from 207.105.83.51: bytes=32 time=70ms TTL=245
Reply from 207.105.83.51: bytes=32 time=90ms TTL=245
```

The response from PING lists the remote machine's IP address, the number of bytes in the packet sent to the machine, the round trip time in milliseconds, and the time to live (TTL) value. (TTL isn't important for this discussion, but it basically specifies the amount of time this packet is considered to be good.) The PING program is great for testing your connection to the Internet and for checking if a particular machine is up. However, it doesn't help much if you want to perform that task from within a C++Builder program.

Ping, the easy way

Without question, the easiest way to implement ping services in your C++Builder applications is to get a component that does the work for you. One such component is Francois Piette's TPing component. TPing is part of Francois' Internet Component Suite (ICS). You can download ICS from the Francois Piette's Web site at

www.rtfm.be/fpiette/icsuk.htm

Once you've downloaded and installed ICS, you can simply drop a TPing component on a form and write code like this:

```
if (Ping1->Ping())
  // Ping found the remote machine.
  ShowMessage("Machine replied");
else
  // Machine did not respond.
  ShowMessage("No response");
```

There are other ping components available, but ICS has a good reputation and the components are freeware.

TNMPing

One other ping component I want to mention is the TNMPing component from NetMasters. The original NetMasters Internet components that ship with C++Builder Professional doesn't contain a ping component. NetMasters has made an update to their components available for download from their Web site. This update adds the TNMPing component to the FastNet component suite. You can download the update from the NetMasters Web site at

www.netmastersllc.com

Writing your own ping routine

If you need more control over the ping process, or if you simply want to delve into the mysteries of Windows, then you can write your own ping routine. There are essentially two ways to go about writing a ping routine:

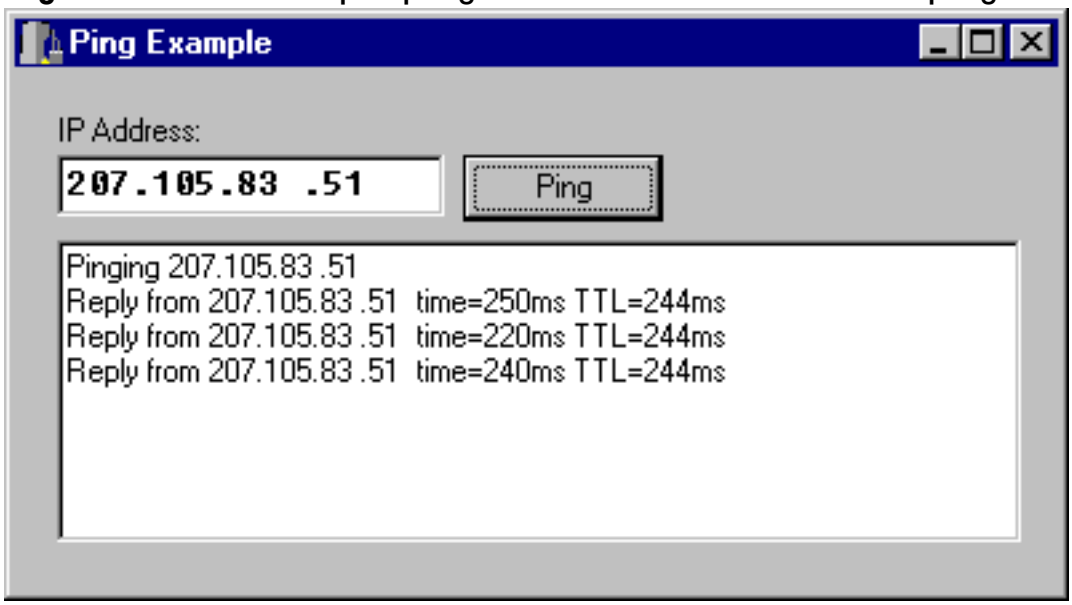
- Use the functions found in Microsoft's ICMP.DLL.
- Use Winsock 2 raw sockets (SOCK_RAW).

We'll focus on the first of these methods in this article. If you want to find out more about using raw sockets, you can find information on the Microsoft Developer's Network (MSDN) CD-ROM. There you'll find an example program that shows how to implement raw sockets. If you don't have MSDN, you can check the Microsoft Web site for information on this topic. In either case, search for the article entitled *Ping: SOCK_RAW in Winsock 2.0*.

Ping using ICMP.DLL

The remainder of this article will explain how to use the services found in ICMP.DLL to ping a remote machine. Figure A shows our sample program running.

Figure A: The example program shows the results of a ping.



This technique comes with a caveat. Microsoft documentation on ICMP.DLL (such as it is) includes this warning: *Notice that the functions in icmp.dll are not considered part of the Win32 API and will not be supported in future releases. Once we have a more complete solution in the operating system, this DLL, and the functions it exports, will be dropped.* Granted, this is a warning that shouldn't be discarded lightly. On the other hand, Windows 2000 will still include ICMP.DLL, so it's not going away anytime soon. In addition, the Windows CE API includes the ICMP functions as part of the standard API. While Windows CE doesn't apply to C++Builder, the fact that the ICMP functions are available in CE would seem to indicate that ICMP.DLL will be around for some time.

Implementing ping using ICMP.DLL requires these steps:

- Create an import library file for ICMP.DLL.
- Obtain the headers required to use the routines in ICMP.DLL.
- Call the IcmpSendEcho() function to perform the ping.

Of these steps, the second is the most formidable. We'll explain why in the following sections.

Creating an import library file

As with all Windows DLLs, you can choose to load ICMP.DLL either dynamically, using LoadLibrary(), or statically, using an import library. If you choose to load ICMP.DLL dynamically, you can skip this step. This article assumes static loading of the DLL. Creating an import library for ICMP.DLL is a trivial exercise. Use the C++Builder IMPLIB utility to create the import library file. Simply open a command prompt box and enter this line at the command prompt:

```
implib icmp.lib c:\winnt\system32\icmp.dll
```

Naturally, if you're using Windows 95 or 98, then you'll need to modify the command line to point to your Windows\System directory:

```
implib icmp.lib c:\windows\system\icmp.dll
```

When IMPLIB runs, you'll end up with a file called ICMP.LIB. You'll add this import library file to your C++Builder projects that use the ICMP routines. Adding the import library file to your C++Builder project is as simple as choosing Add to Project, and then choosing ICMP.LIB.

Locating the ICMP headers

As I said earlier, this is possibly the hardest step. It's a difficult step because the header files required for ICMP.DLL aren't readily available. The headers you need are the following:

- IPEXPORT.H
- ICMPAPI.H

The Microsoft Win32 SDK used to ship with ICMPAPI.H, so if you have an older version of the SDK, you can still obtain the headers from that source. The header could still be available on the latest SDK CD-ROM, so be sure to check there before embarking on a Web search for this header.

The more problematic header is IPEXPORT.H. This header contains structures required by ICMPAPI.H. Curiously enough, the Windows CE Platform SDK references these files so if you

have that SDK, you can check to see if you already have the required files.

The quickest way to obtain these files is simply by searching the Internet. At the time of this writing you can find IPEXPORT.H and ICMPAPLH at

http://callamer.com/~jscarlet/JimWare/VBicmp/Ipexport.txt

(There are probably other Web sites that have these files, but this is where I was able to find them.) Once you've obtained the ICMP headers, you can get on to the more serious business of writing your ping routine.

Writing the ping routine

Writing the ping routine itself requires less than a dozen lines of code. The first step is to obtain an ICMP handle by calling IcmpCreateFile(). This handle is needed later when you call IcmpSendEcho(). Here's the code for obtaining the file handle:

```
HANDLE hIcmp = IcmpCreateFile();
```

The next step requires making an int value out of an IP address. As you probably know, an Internet address is specified in the form 207.105.83.51. This value is translated into an integer value. The easy way to make an int from four byte values is by using Windows' MAKEWORD and MAKELONG macros. The following code illustrates this:

```
int addr = MAKELONG(
        MAKEWORD(207, 105),
        MAKEWORD(83, 51));
```

This assumes that you're starting with the IP address of the machine you want to ping, rather than a host name such as **www.bridgespublishing.com**.

At this point, you have an ICMP handle and an IP address in integer form. You can now proceed to call IcmpSendEcho() to perform the ping. IcmpSendEcho() is declared in ICMPAPI.H as follows:

```
IcmpSendEcho(
        HANDLE IcmpHandle,
        IPAddr DestinationAddress,
        LPVOID RequestData,
        WORD   RequestSize,
        PIP_OPTION_INFORMATION RequestOptions,
```

```
        LPVOID ReplyBuffer,
        DWORD  ReplySize,
        DWORD  Timeout
);
```

Granted, this declaration looks a bit intimidating. It's not quite as bad as it looks, though, because you can simply set most of the parameters to 0. Here's how the call to IcmpSendEcho() looks:

```
int size = sizeof(icmp_echo_reply) + 8;
char* buff = new char[size];
DWORD res = IcmpSendEcho(hIcmp, addr, 0, 0, 0, buff, size, 1500);
```

The first parameter of IcmpSendEcho() is the ICMP handle obtained in the call to IcmpCreateFile(). The second parameter is the IP address of the machine you're pinging. The third, fourth, and fifth parameters are used to specify request parameters. For simple ping operations you can set these parameters to 0. If you wanted to perform advanced ICMP operations, such as performing a trace route, you'd need to set these parameters to meaningful values.

The sixth parameter, ReplyBuffer, is a pointer to a buffer that will receive the reply information returned from the remote machine. The seventh parameter is used to specify the size of the return buffer. We used this code to create the buffer and set the size:

```
int size = sizeof(icmp_echo_reply) + 8;
char* buff = new char[size];
```

Note that we set the buffer size to the size of an icmp_echo_reply structure, plus eight bytes. This is a safe buffer size. Typically, this is a large enough buffer to contain the reply information, based on the way we're calling IcmpSendEcho(). I've seen example code that allocated an extra eight *kilobytes* of buffer space; that seems like gross overkill, especially for what we're doing.

The final parameter of IcmpSendEcho() is used to specify a timeout value. We used a timeout value of 1500 milliseconds in this example, but you can use any timeout value you like. If you use a longer timeout value, a failed call to IcmpSendEcho() will take longer to complete. If you use a short timeout value, the operation may timeout before the call to IcmpSendEcho() completes, even though the server you're pinging may indeed be up and running.

If the server replies, IcmpSendEcho() will return a non-zero value (usually 1). If the server doesn't reply during the timeout period, IcmpSendEcho() will return 0. Keep in mind that the call to IcmpSendEcho() will fail if the server isn't responding, if you've provided an invalid IP

address, or if you're not connected to the Internet. Further, understand that a "bad" IP address may contact a remote machine other than the one you intended to ping. In other words, you may get a reply from a server, but it may not be the server you're looking for!

After calling IcmpSendEcho() you need to free the ICMP handle you obtained in the call to IcmpCreateFile(). You do that by calling IcmpCloseHandle():

```
IcmpCloseHandle(hIcmp);
```

At this point the ping operation, whether successful or not, is complete.

Examining the reply

If IcmpSendEcho() succeeds, you may want additional information on the result of the ping. You can obtain the reply information by copying the first 24 bytes of the reply buffer to an instance of the icmp_echo_reply structure. You can then examine the members of the icmp_echo_reply structure for more information. icmp_echo_reply is declared as follows:

```
struct icmp_echo_reply {
        // Replying address
        IPAddr Address;
        // Reply IP_STATUS
        unsigned long Status;
        // RTT in milliseconds
        unsigned long RoundTripTime;
        // Reply data size in bytes
        unsigned short DataSize;
        // Reserved for system use
        unsigned short Reserved;
        // Pointer to the reply data
        void FAR *Data;
        // Reply options
        struct ip_option_information
        Options;
};
```

Here again, this structure looks like it will take a lot of work to parse. In reality, you're probably only interested in the Status and RoundTripTime data members. You can ignore the rest of the data members unless you have specific needs beyond a simple ping operation. The Options member is a structure that contains the time to live in its Ttl member, so you might be interested in that information, as well.

To begin, we copy the first part of the reply buffer to an instance of icmp_echo_reply:

```
icmp_echo_reply reply;
memcpy(&reply, buff, sizeof(reply));
```

Now we can build a string to report the results of the ping to the user. The string might be constructed like this:

```
String rtt = reply.RoundTripTime;
String ttl = reply.Options.Ttl;
String S = "Reply from " + IPEdit->Text +
        " time=" + rtt + "ms TTL=" + ttl + "ms";
```

This code assumes that you have an Edit component on a form, and that the Edit contains the IP address of the server you're pinging. When this code executes, the string will contain a value similar to this:

```
Reply from 207.105.83.51 time=220ms TTL=244ms
```

It's no coincidence that this string roughly resembles the text returned from the PING program itself. You may also want to examine the value of the Status member of the icmp_echo_reply structure to determine the cause of any ping failures. The Status member will be set to 0 if the ping operation succeeds, but will be a value of 11000 or greater if an error occurred. For example, IcmpSendEcho() might return 0, indicating the function was called successfully, but the status value may indicate some error condition. The IPEXPORT.H header defines the error values listed in Table A.

Table A: Error values

```
IP_BUF_TOO_SMALL
IP_DEST_NET_UNREACHABLE
IP_DEST_HOST_UNREACHABLE
IP_DEST_PROT_UNREACHABLE
IP_DEST_PORT_UNREACHABLE
IP_NO_RESOURCES
IP_BAD_OPTION
IP_HW_ERROR
IP_PACKET_TOO_BIG
IP_REQ_TIMED_OUT
IP_BAD_REQ
IP_BAD_ROUTE
IP_TTL_EXPIRED_TRANSIT
```

```
IP_TTL_EXPIRED_REASSEM
IP_PARAM_PROBLEM
IP_SOURCE_QUENCH
IP_OPTION_TOO_BIG
IP_BAD_DESTINATION
IP_ADDR_DELETED
IP_SPEC_MTU_CHANGE
IP_MTU_CHANGE
IP_UNLOAD
IP_ADDR_ADDED
```

If you've used the Windows' PING program, you may recognize some of these error constants as relating to messages PING returns in the case of a problem connecting to a remote machine. It's a simple matter to build an array of strings that map to these values (which fall in the range of 11001 to 11023). See Listing A for an example. Figure A shows the example program running. The program's main form contains an Edit component used to specify the IP address, a Button component that starts the ping operation, and a Memo component that displays the result of the ping. We haven't placed the code for this article on our Web site due to the fact that it requires IPEXPORT.H and ICMPAPI.H files, which we can't provide due to copyright considerations.

Notes

Listing A contains the code for the main unit of this article's example program.

Listing A: PingExU.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "PingExU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

// The ICMP.DLL headersyou'll have to obtain
// these from Microsoft or on the Internet.
extern "C" {
#include "ipexport.h""
#include "icmpapi.h"
}
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

// An array of error message strings.
String ErrorStrings[] = {
        "Error Base",
        "Buffer too small.",
        "Destination net unreachable.",
        "Destination host unreachable.",
        "Destination protocol unreachable.",
        "Destination port unreachable.",
        "Out of resources.",
        "Bad option.",
        "Hardware error.",
        "Packet too large.",
        "Request timed out.",
        "Bad request.",
        "Bad route.",
        "TTL expired in transit.",
        "TTL expired REASSEM.",
        "Param problem.",
        "Source quench.",
        "Option too large.",
        "Bad destination.",
        "Address deleted.",
        "Spec MNU change.",
        "MTU change.",
        "Unload"
};

void __fastcall
TForm1::PingBtnClick(TObject *Sender)
{
        // Obtain an ICMP handle.
        HANDLE hIcmp = IcmpCreateFile();
        if (!hIcmp) {
                ShowMessage("Error getting ICMP handle");
                return;
        }
        // Clear the Memo of any previous text.
        Memo1->Lines->Clear();
```

```
Memo1->Lines->Add("Pinging " + IPEdit->Text);
// Parse the IP address in the Edit.
String S = IPEdit->Text;
int addr1 = Trim(S.SubString(1, 3)).ToInt();
int addr2 = Trim(S.SubString(5, 3)).ToInt();
int addr3 = Trim(S.SubString(9, 3)).ToInt();
int addr4 = Trim(S.SubString(13, 3)).ToInt();
// Make an int out of the IP address.
int addr = MAKELONG(
        MAKEWORD(addr1, addr2),
        MAKEWORD(addr3, addr4));
// Allocate a buffer for the reply info.
int size = sizeof(icmp_echo_reply) + 8;
char* buff = new char[size];
// Show the user we'll be busy for a while.
Screen->Cursor = crHourGlass;
// Send the echo request three times to
// emulate what the PING program does.
for (int i=0;i<3;i++) {
        Application->ProcessMessages();
        // Call IcmpSendEcho().
        DWORD res = IcmpSendEcho(hIcmp,
                addr, 0, 0, 0, buff, size, 1500);
        if (!res) {
                Memo1->Lines->Add("Request timed out.");
                continue;
        }
        // Prepare to report the status.
        icmp_echo_reply reply;
        memcpy(&reply, buff, sizeof(reply));
        // If the status is non-zero then show the
        // corresponding error message from the
        // ErrorStrings array to the user.
        if (reply.Status > 0)
                Memo1->Lines->Add(
                        ErrorStrings[reply.Status - 11000]);
        else {
                // Build a string to report the results.
                String rtt = reply.RoundTripTime;
                String ttl = reply.Options.Ttl;
                String S = "Reply from " + IPEdit->Text +
                " time=" + rtt +"ms TTL=" + ttl + "ms";
                // Add it to the memo.
```

```
                        Memo1->Lines->Add(S);
                }
                // Pause a second and then loop.
                Sleep(1000);
        }
        // Close the ICMP handle.
        IcmpCloseHandle(hIcmp);
        // Restore the cursor.
        Screen->Cursor = crArrow;
}
```

Conclusion

Pinging a remote machine using ICMP.DLL is a good way to determine whether users of your program are currently connected to the Internet, or whether a particular server you want to access is available. This method allows you to quietly check for the availability of a server without worrying about exceptions that many Internet components throw in the event of a connect failure.

NetMasters

The Internet Developers' Developer

June 4, 2002
*Fast*Net Version 6.2.1 for all versions of Delphi and CBuilder available now

- POP3
  - POP3 Re-engineered to handle hierarchical attachments
  - 2 New events for greater control of parsing of each level
  - Variations in CRLF at end of line handled
  - OnDecodeStart Moved so attachment files with invalid characters can be renamed
  - Parse attachments made TRUE by default
- SMTP
  - Reply from server added to OnRecipientNotFound event
  - Non-English characters in Subject, FromAddress, Attachments, ToAddress etc encoded
  - Htm as well as html attachments goes as text/htlm by default
  - Extra ' =' character in QP encoded documents eliminated - This caused problems in sending .pdf
- FTP
  - Lists and NLists with no CRLF handled
  - Listings without group fields handled by parser
  - Lists and NLists made threaded
  - Multiple line replies of length less than 4 and zero CRLF replies handled properly
- GeneralServer
  - Each Server threads Timeout set to Main Timeout
  - RemoteAddress of threads set immediately after Accept
  - ClientContact Fired after accept

December 1, 2001
*Fast*Net Version 6.1.2 for all versions of Delphi and CBuilder released

The leading set of internet components keeps getting better.

For C++ Builder

- Version 6
- Version 5
- Version 4
- Version 3

For Delphi

- Version 7
- Version 6
- Version 5
- Version 4
- Version 3

New Features in Ver 6.1.2 include

SMTP

- Authentication on SMTP server
- 2 New Events to individually set the Content-Type of attachments and modify headers of attachments (Allows users to send html with graphics etc)
- OnAttachmentNotFound firing at correct time
- Extended reporting of OnRecipientNotFound
- Body can be sent as html with attachments present
- EHLO on logon to get capabilities of server
- Boundary string fixed for problems with non-ASCII characters in the Time String of non-English operating systems

FTP

- Threaded upload, download and lists that eliminate the possibility of timing
- errors and enable easier processing of other tasks while waiting for a download, upload or list.
- Cleaner aborts resulting from new architecture
- New parser for Unix and MSDOS type listings
- ReadExtraLines changed to work for users with different settings in logic handling

HTTP

- 256-character limitation in URLs removed
- NTLM Authentication added
- Wait for data before looking for HTTP at beginning
- RemoveHeader parsing improved

POP3

- Non-ASCII characters in Subject, From, To, Reply To fields and Attachment
- Filenames handled.
- OnDecodeStart moved to handle attachment filenames with invalid characters
- Date added to Summary
- Extra carriage return line feed removed to help QP decoder
- Boundary Parsing improved

UDP

- 255.255.255.255 valid destination
-

PSock

- Separate buffers for Read and Write to handle simultaneous reads and writes
- Sending and Receiving properties added
- CaptureString exit conditions extended
- Bind method added
- CloseAfterData method enhanced
- Buffer not cleared after a remote close to prevent losing data on a remote close
- DataAvailable checks buffer before start and closes socket if a remote close is detected.
- BytesTotal set in SendFile

Click here to see features of earlier versions including all changes from the free version included with Delphi and CBuilder

Click here to Purchase Version 6.1.2

- We are putting the finishing touches on our next release. Please feel free to send us your suggestions for new components and features for future releases. All Maintenance Release Customers will automatically receive the next release. So be sure that you become a maintenance customer and know that your investment will stay current for the next year.

Click here to send suggestions and feature requests

---

131 DW HWY #507 Nashua, NH 03060
(603) 889-1067

Site Last Updated
March 11, 2002

June 1999

Compiler speed

Borland's compiler is famous for the speed it compiles a source file. But I wonder why C++Builder's compiling speed is so slow. Would you mind telling me how to speed up a project's build time?

Huang Xiaoyang
via email

Believe it or not, the compiler in C++Builder 4 is probably the fastest Borland C++ compiler to date. It only *appears* to be slower than its predecessors. With each new C++ compiler that Borland produces, the Windows operating system grows more and more complex. The result is that more APIs are added to Windows at every turn. More APIs mean more and larger header files for those APIs.

The vast majority of C++Builder's compiling time is spent compiling hundreds of thousands of lines of Windows headers. In addition to the Windows headers, the compiler also has to deal with the VCL headersat least for a standard C++Builder application. Depending on how you have your projects set up, the compiler may be compiling these headers over and over again for each unit in your project.

In addition to the header file problem is the problem of operating system performance. Obviously, Windows 95, 98, and NT are more bloated than Windows 3.1, which itself was horribly bloated when compared to the lowly DOS. These operating systems require more and more memory, and more processing power, to do their thing. This means that you must have a faster processor and more memory when running under these operating systems than you needed back in the Windows 3.1 or DOS days.

The solution to the header file problem is proper use of pre-compiled headers. Once you have pre-compiled headers properly set up, your compile times will likely be drastically reduced (maybe by as much as 75 percent). I won't try to explain how to properly set up pre-compiled headers here. Instead, I'll point you to the Web site of my fellow TeamB member, Harold Howe, which contains a detailed article on pre-compiled headers. You can find the article at

www.bcbdev.com/articles/pch.htm

The solution to the problem with Windows itself, naturally, is to be sure you have plenty of RAM, and plenty of processing power in your development machine. I wouldn't consider running today's compilers on a machine with less than 64 MB of RAM. Most of my machines have at least 128 MB of RAM. RAM is probably more important than processor speed, so if you have to choose between getting more RAM and getting a faster processor, get more RAM.

As the programming world becomes more complex, C++ compilers will appear to be slower than their

predecessors, even though they're actually faster. The truth is, it's not the compiler's fault as much as it is the fault of the behemoth we call Windows.

Kent Reisdorph
TurboPower Software
KentR@TurboPower.com

Improving C++Builder Build Times With Pre-Compiled Headers.

C++Builder is one of the fastest C++ compilers around, and probably the fastest Win32 C++ compiler in terms of compilation speed. Despite the speed advantage that C++Builder holds over other C++ compilers, many Delphi programmers contort in agony while waiting for a C++Builder project to compile. Anyone that has seen Delphi knows that it is blazing fast in comparison to any C++ compiler. Delphi can compile small example projects in less than a second, and large projects can be built in less than five.

So why does Delphi hold such a speed advantage over C++Builder? Furthermore, is there anything that can be done to improve the compilation speed of C++Builder? This article explains why C++ compilers are inherently slow, and demonstrates a simple tactic to boost compile times in C++Builder.

Why C++ Compilers are Slow

In C++, you cannot call function from a source file unless that function has been previously defined or declared. So what does this mean? Consider a simple example where function `A()` calls function `B()`. `A()` cannot call `B()` unless a prototype for `B()`, or the function body for `B()`, resides somewhere above the function body for `A()`. The code below illustrates this point.:

```
// declaration or prototype for B
void B();

void A()
{
    B();
}

// definition, or function body of B
void B()
```

```
{
    cout << "hello";
}
```

The code will not compile without the prototype for `B()`, unless the function body for `B()` is moved up above `A()`.

Function prototypes serve a crucial role to the compiler. Every time you execute a routine, the compiler must insert proper code to call the routine. The compiler must know how many parameters to pass the function. The compiler must know if the function expects its parameters on the stack or in registers. In short, the compiler needs to know how to generate the correct code to call the function, and it can only do this if it has seen a previous declaration or definition for the function that is being called.

To simplify the prototyping of functions and classes, C++ supports a `#include` statement. The `#include` directive allows a source file to read function prototypes from a header file prior to the location in code where the prototyped functions are called. The `#include` directive plays an important role in Win32 C++ development. Function prototypes for C RTL functions are provided in a standard set of header files. The Win32 API is prototyped in a set of header files provided by Microsoft, and the classes and functions of the VCL are listed in header files that come with C++Builder. You can't create a very useful Windows program without including header files provided by Microsoft or Borland.

Header files help implement C++ type checking in a manner that is easy to manage for the programmer. However, this benefit comes at a huge cost. When the compiler runs across a `#include` directive, it literally opens the included file and inserts it into the current file. The compiler then parses the included file as if it was part of the file that it was already compiling. So what happens if the first header file includes yet another file? The compiler will suck in that file and start parsing it. Imagine what happens when 10, 20, or even 100 files are included? While this number of include files may sound large, it isn't unrealistic when you start adding up the Windows SDK header files and all of the VCL header files.

To demonstrate how the compiler branches off and translates included files, consider the following code example. This is a simple console mode program that I built using the Console Wizard from the Object Repository. In order to test this code, select Options-Project-Compiler and turn off pre-compiled headers.

```
// include some standard header files
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>

#pragma hdrstop
#include <condefs.h>

//---------------------------------------------
int main()
{
    printf("Hello from printf.\n");
```

```
    cout << "Hello from cout" << endl;
    MessageBeep(0);
    return 0;
}
```

When I build this project with C++Builder, the build progress dialog reports that the project contains 130,000 lines of code. 130 thousand lines! How can that be? The source file only contains about four lines of code. The 130,000 lines were contained in STDIO.H, STRING.H, IOSTREAM.H, WINDOWS.H and all of the other header files that are included by these four header files. In this example, the compiler spent the vast majority of its time processing header files.

Now let's investigate what happens when you have multiple CPP files in a project. Building off of the existing project, let's add a unit to the console program that we already have. Add a simple function to this second unit. Then alter the first CPP file so it will call the new function.

```
//---------------------------------------------------------
// UNIT1.CPP
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>
#include "Unit1.h"          // prototype A() in unit1.h

#pragma hdrstop

void A()
{
    printf("Hello from function A.\n");
}
//---------------------------------------------------------

//---------------------------------------------------------
// PROJECT1.cpp
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <windows.h>
#include "Unit1.h"

#pragma hdrstop
#include <condefs.h>

//---------------------------------------------------------
USEUNIT("Unit1.cpp");
//---------------------------------------------------------
int main()
{
```

```
    printf("Hello from printf.\n");
    cout << "Hello from cout" << endl;
    A();
    MessageBeep(0);
    return 0;
}
//---------------------------------------------------------
```

Now build the project. If you turned off pre-compiled headers before building the project, you will see that the compiler progress dialog reports 260,000 lines of code when the build finishes. Notice that the compiler had to translate the same set of header files in two different CPP files. We know from the previous example that these header files place a burden of 130,000 lines on the compiler. The second CPP file places the same 130,000 line burden on the compiler, for a grand total of 260,000 lines of header files. Imagine how this line count would rapidly grow in a large project. The burden of processing the same group of header files over and over can greatly increase compile times.

How C++Builder Uses Pre-Compiled Headers To Reduce Compile Times

The engineers at Borland realized that they could decrease build times by designing a compiler that did not process the same header files over and over during the build. To achieve this goal, Borland C++ 3.0 introduced the concept of pre-compiled headers. The idea behind pre-compiled headers is relatively simple. When the compiler processes a set of header files for one particular source file, it saves the compiled image of the header files on the hard drive. When that set of header files is required by another source file, the compiler loads the compiled image instead of processing the header files a second time.

Let's modify our console mode program to see how pre-compiled headers can impact build times. The code itself should already be fine. We just need to turn the project's pre-compiled headers option back on. Select Options-Project-Compiler and check the *Use pre-compiled headers option* or the *Cache pre-compiled headers option*. Enter PCH.CSM in the pre-compiled header filename box. Do a full rebuild of the project once you change the settings.

During the build, pay special attention to the compiler progress dialog. You should see that the compiler processes 130,000 lines of code when it compiles PROJECT1.CPP, but when it compiles UNIT1.CPP, it only process 20 lines of code. The compiler generates a pre-compiled image when it parses the first source file, and that pre-compiled image is used to speed up compilation of the second source file. Imagine the performance boost that you would attain if the project contain 50 source files instead of 2.

Explanation of Pre-Compiled Headers in a VCL GUI Project

The use of pre-compiled headers in the previous example reduced the build time of the project by almost 50%. But that was a simple console mode program that didn't do much. You probably want to know how you can take

advantage of pre-compiled headers in a full blown VCL GUI program. By default, C++Builder automatically turns on pre-compiled headers for you. However, C++Builder does not pre-compile every header file that is used by your program. It only pre-compiles the file `VCL.H`, which you can see by inspecting the top of any form's source file:

```
#include <vcl.h>
#pragma hdrstop
```

The `#pragma hdrstop` directive tells the compiler to stop generating the pre-compiled image. Any `#include` statement located before the `hdrstop` directive will be pre-compiled, while any `#include` below the directive will not be pre-compiled.

So how many header files get pre-compiled when `VCL.H` is pre-compiled? If you look at `VCL.H`, you will see that it includes another file called `VCL0.H` (assuming you have BCB3). If you don't alter the default settings of C++Builder, `VCL0.H` will include a small set of VCL header files. They are:

```
// Core (minimal) VCL headers
//
#include <sysdefs.h>
#include <system.hpp>
#include <windows.hpp>
#include <messages.hpp>
#include <sysutils.hpp>
#include <classes.hpp>
#include <graphics.hpp>
#include <controls.hpp>
#include <forms.hpp>
#include <dialogs.hpp >
#include <stdctrls.hpp>
#include <extctrls.hpp>
```

This is a small cross section of header files, and it probably represents only a subset of the header files that are used in a moderate to large sized project. `VCL0.H` does allow you to pre-compile more header files through the use of conditional defines. You can `#define` a variable called `INC_VCLDB_HEADERS` to pre-compile the VCL database header files. Likewise, you can define `INC_VCLEXT_HEADERS` to pre-compile header files for the extra controls that come with C++Builder. If you define a variable called `INC_OLE_HEADERS`, C++Builder will pre-compile some of the SDK COM header files. These defines should be placed before the `#include` statement for `VCL.H`.

```
#define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS
#include <vcl.h>
#pragma hdrstop
```

Note: If you decide to try this technique, make sure you add to the two defines to every CPP file, even if they don't use DB classes or extra controls. The reasoning for this will be explained shortly.

Optimizing C++Builder's Use of Pre-Compiled Headers

The default pre-compiled header settings do reduce the time it takes to build a project. You can prove this fact by timing a full build of a large project when pre-compiled headers are on and by timing the build when pre-compiled headers are off. The goal of this article is to tweak the way C++Builder pre-compiles files to reduce build times even more. In this section, I have outlined two techniques for improving build times.

Before we discuss the techniques, it important to realize how C++Builder decides that it can use an existing pre-compiled image when compiling a source file. C++Builder generates a unique pre-compiled image for every source file in your project. These pre-compiled images are saved in a file on your hard drive. The compiler will re-use an existing pre-compiled image when two source files require the same pre-compiled image. This is an important distinction. Two source files will require the same pre-compiled image if they include exactly the same files. Furthermore, they must include the files in the same order. Simply put, the source files must be identical up until the `#pragma hdrstop` directive. Here are some examples:

Example 1: Pre-compiled images don't match

```
//--------------------                  //--------------------
// UNIT1.CPP                            // UNIT2.CPP
#include <stdio.h>                      #include <iostream.h>
#pragma hdrstop                         #pragma hdrstop
```

Example 2: Pre-compiled images don't match

```
//--------------------                  //--------------------
// UNIT1.CPP                            // UNIT2.CPP
#include <stdio.h>                      #include <stdio.h>
#include <iostream.h>                   #pragma hdrstop
#pragma hdrstop
```

Example 3: Pre-compiled images don't match

```
//--------------------                  //--------------------
// UNIT1.CPP                            // UNIT2.CPP
#include <stdio.h>                      #pragma hdrstop
#pragma hdrstop                         #include <stdio.h>
```

Example 4: Pre-compiled images match

```
//--------------------                  //--------------------
// UNIT1.CPP                            // UNIT2.CPP
#include <stdio.h>                      #include <stdio.h>
#include <string.h>                     #include <string.h>
#include <iostream.h>                   #include <iostream.h>
#include <windows.h>                    #include <windows.h>
```

```
#include "unit1.h"                          #include "unit1.h"
#pragma hdrstop                             #pragma hdrstop
```

Example 5: Pre-compiled images match

```
//--------------------                      //--------------------
// UNIT1.CPP                                // UNIT2.CPP
#define  INC_VCLDB_HEADERS                  #define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS                 #define  INC_VCLEXT_HEADERS
#include <vcl.h>                            #include <vcl.h>
#pragma hdrstop                             #pragma hdrstop

#include "unit1.h"                          #include "unit2.h"
```

Example 6: Pre-compiled images don't match

```
//--------------------                      //--------------------
// UNIT1.CPP                                // UNIT2.CPP
#define  INC_VCLDB_HEADERS                  #include <vcl.h>
#define  INC_VCLEXT_HEADERS                 #pragma hdrstop
#include <vcl.h>
#pragma hdrstop
```

When the compiler processes a source file with a pre-compiled image that does not match an existing image, the compiler will produce a completely new image from scratch. Look at Example 2 above. Even though `STDIO.H` is compiled along with `UNIT1.CPP`, the compiler will translate `STDIO.H` again when it compiles `UNIT2.CPP`. Pre-compiled headers reduce compile times only when the compiler can re-use an existing pre-compiled image across multiple source files.

This is the foundation for both of the techniques that I list here. Pre-compile as many header files as you can, and make sure that you use the same pre-compiled image in every source file.

Technique 1:

The first technique is to simply boost the number of files that `VCL.H` includes by adding two conditional defines to every source file. Open every CPP file in the project, including the project source file, and change the first two lines of the file so they look like:

```
#define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS
#include <vcl.h>
#pragma hdrstop
```

If you don't like the idea of adding these defines to every source file, you can accomplish the same thing by adding `INC_VCLDB_HEADERS` and `INC_VCLEXT_HEADERS` to the conditional defines line under Project - Options - Directories/Conditional.

You might want to throw in some of the C RTL header files that you commonly use, along with `WINDOWS.H`.

Make sure that you add the lines before the `hdrstop` pragma, and make sure that you list them in the same order in every C++ source file.

```
#define  INC_VCLDB_HEADERS
#define  INC_VCLEXT_HEADERS
#include <vcl.h>
#include <windows.h>
#include <stdio.h>
#pragma hdrstop
```

Technique 2:

Technique 1 works fairly well, but it isn't very flexible. If you decide to add a new header file to the list of files that get pre-compiled, you need to modify every C++ source file in your project. Furthermore, Technique 1 is prone to error. If you mess up the order of your includes, you can actually make your compile times worse, not better.

Technique 2 addresses some of the downfalls of Technique 1. The strategy here is to create one huge header file that includes every header file that is used in your project. This single file will include the VCL files, windows SDK header files, and RTL header files. It can also include all of the header files for forms and units that you have created, but as we will see later on, you don't want to pre-compile header files that are likely to change (see the note entitled *Don't pre-compile header files that change*).

Here is an example of what the common header file will look like:

```
//---------------------------------------------------------
// PCH.H: Common header file
#ifndef PCH_H
#define PCH_H

// include every VCL header that we use
// could include vcl.h instead
#include <Buttons.hpp>
#include <Classes.hpp>
#include <ComCtrls.hpp>
#include <Controls.hpp>
#include <ExtCtrls.hpp>
#include <Forms.hpp>
#include <Graphics.hpp>
#include <ToolWin.hpp>

// include the C RTL headers that we use
#include <string.h>
#include <iostream.h>
#include <stdio.h>
```

```
// include headers for the 3rd party controls
// TurboPower System
#include "StBase.hpp"
#include "StVInfo.hpp"

// Our custom controls
#include "DBDatePicker.h"
#include "DBRuleCombo.h"
#include "DBPhonePanel.h"

// Object Repository header files
#include "BaseData.h"
#include "BASEDLG.h"

// project include files
// pre-compile these only if PRECOMPILE_ALL is defined
#ifdef PRECOMPILE_ALL
#include "About.h"
#include "mainform.h"
...
... // about 60 more files
...
#include "validate.h"
#endif

#endif
```

Once you have the gigantic common header file ready, change every source file so it includes this file. I have chosen to leave the original include statement for `VCL.H` intact. You might want to move `VCL.H` to the common header file.

```
//---------------------------------------------
#include <vcl.h>
#include "pch.h"
#pragma hdrstop
```

Note: After you add the include for `PCH.H` to every C++ source file, don't insert any more include files prior to the `#pragma hdrstop`. Doing so will cause those C++ files to require a pre-compiled image that does not match the pre-compiled image from other files.

Results:

I am currently employing Technique 2 without defining `PRECOMPILE_ALL` on my current project. The project is a medium sized client/server database program that consists of 113 C++ source files, most of which are forms or datamodules. Using Technique 2, a full build of the project takes only 195 seconds. Of that 195 seconds, 40 seconds are spent generating the pre-compiled header, and about 40 seconds are spent linking. In the remaining time, the compiler translates 113 C++ source files. That's an average of one file per second. By way of

comparison, the project takes more than 30 minutes to build when no pre-compiled headers are used, and the project takes 18 minutes to build when pre-compiled headers are used but Technique 2 is not utilized.

Incremental makes with Technique 2 are lightning fast when no header files have changed. The compiler does not bother to regenerate the pre-compiled image on disk if no header files have changed. When this condition is met, an incremental make takes only 1 or 2 seconds, because only the C++ source files that have changed need to be compiled. The compiler spends all of its time compiling those files, instead of wasting its time compiling system header files. When a header file does change, the speed of an incremental make depends on whether or not the `PRECOMPILE_ALL` flag was defined.

Notes:

Don't pre-compile constant variables: The compiler cannot pre-compile a header file if it contains a constant variable that is assigned a value. For example, placing the following line in a header file can interfere with the creation of a pre-compiled header image:

```
const AnsiString strError = "An Error Occurred!!!!!";
```

If you want to place const variables in a header file, create a separate header file to contain the constants and don't pre-compile that file. Try to reduce the burden on the compiler by not allowing this header file to include other files. Similarly, don't include this file from other header files if you can help it. If this seems like a difficult task, you can use the `extern` keyword. Create a header file that contains `extern` prototypes for your constants. Then create a `CPP` file that defines the constants (ie gives them a value).

Note that the problem of const variables only occurs when you do define the `PRECOMPILE_ALL` flag. When you do not define this flag, your own header files are not pre-compiled. If you don't pre-compile a header file, then you can add constants to it without any problems. Also, `#define`'s do not present a problem, just const variables (although I don't recommend that you switch back to `#define`'s).

Don't pre-compile template headers: This suggestion is based on empirical evidence. I have a template class with several inline functions. The entire template class resides in a header file. The compiler was able to pre-compile this header file, but I noticed that the pre-compiled image was always re-generated during an incremental make. I think this has to do with the way templates are handled by the compiler. I can pre-compile the STL headers without any problems, so I'm not sure what the problem is. I suggest that you go ahead and try to compile template headers, but pay close attention to the compiler progress dialog. You may need to stop pre-compiling template headers if they cause problems.

Keep an eye on the compiler progress dialog: The compiler progress dialog tells you how well your pre-compiled headers are working. When you employ Technique 2, you should see that the compiler takes a long time to compile the first C++ source file in your project. The compiler generates the pre-compiled header image during compilation of the first file in the project. During this time, you should see the line count on the compiler progress dialog reach a huge number (100,000-500,000). Once the compiler moves on to other C++ files, the line count should probably be between 20 and 1000 lines for each source file if you define the `PRECOMPILE_ALL` flag. If you don't define this flag, the line count should stay under 15000 or so. Once the compiler finishes translating the first file in the project, subsequent files should only take a second or two to compile.

If the compiler gets bogged down on one C++ file for more than 4 seconds, you probably have a source file whose pre-compiled image doesn't match the image created by the common header file. The line count is another indicator. If you see the line count sail up above 50,000 lines for one source file, it's a good indication that the compiler was unable to apply the existing pre-compiled image to that source file.

Don't pre-compile header files that change: When using Technique 2, realize that any small change to a header file will force the compiler to regenerate the pre-compiled image. Based on the test results, this could take from 20 seconds to a minute. If your header files change frequently, you may want pre-compile only system and VCL header files. This is the purpose of the `PRECOMPILE_ALL` flag. It allows you to easily include or remove your header files from the pre-compiled image.

```
//---------------------------------------------------------
// PCH.H: Common header file
#ifndef PCH_H
#define PCH_H

// include every VCL header that we use
#include <Buttons.hpp>
#include <Classes.hpp>

// include the C RTL headers that we use
#include <string.h>
#include <iostream.h>
#include <stdio.h>

// project include files
// pre-compile these only if PRECOMPILE_ALL is defined
#ifdef PRECOMPILE_ALL
#include "About.h"
#include "mainform.h"
...
... // about 60 more files
...
#include "validate.h"
#endif

#endif
```

To pre-compile your own header files, add `PRECOMPILE_ALL` to the conditional defines line under Project - Options - Directories/Conditional. If your header files change frequently, then don't add this conditional define. When you don't pre-compile your own header files, a full build of your project will take a little longer. However, when you make a change to one of your header files, an incremental make will be faster because the compiler won't waste 20-60 seconds rebuilding the pre-compiled image.

I do not define `PRECOMPILE_ALL` for the project that I described in the Results section because I found I was still changing my header files frequently, and incremental makes were taking more than 2 minutes. The table

below illustrates how the PRECOMPILE_ALL directive affects compile time. I timed how long a full build took when the PRECOMPILE_ALL was defined. Then I made a small change to the header file for my main form and performed an incremental make. Next, I repeated this process with the PRECOMPILE_ALL value not defined. Here are the results.

```
Not defined (do not pre-compile my headers)
-----------------------------------------------------------
Full Build: 195 sec 408887 lines compiled
Inc Make  :  28 sec 7 files affected: 27059 lines compiled

Defined     (pre-compile my headers)
-----------------------------------------------------------
Full Build: 179 sec 255689 lines
Inc Make  : 179 sec all files affected: 255689 lines
```

Notice that a full build is 16 seconds faster when I pre-compile my own header files, but look what happens when I do an incremental make after changing a header file. The incremental make takes just as long as a full build. When you pre-compile your own header files, the compiler rebuilds the pre-compiled image every time you change a header file. Additionally, when the pre-compiled image changes, every file that depends on that image will be re-compiled as well. So if you alter a header file, the entire project essentially gets rebuilt. When I do not pre-compile my own header files, the pre-compiled image never gets rebuilt. This keeps the incremental make time down, 28 seconds compared to 179 seconds.

Since you probably perform an incremental make 10 times more often than you do a full build, it seems wise to keep the incremental make time down, even if it means that a full build will be 10% slower. This is the approach that I take. I do not define the PRECOMPILE_ALL value in any of my projects.

Don't remove existing `#include` statements: Creating a common header file does not mean that you should remove include statements from your header files and C++ source files. Leave those include statements where they are. There are several reasons why you should leave existing include statements. First, if you remove include statements from your header files, C++Builder will simply add them back again. Second, you may want to stop pre-compiling certain files, which would force to you add the include statements back into your source files. Lastly, by leaving include statements intact, you preserve the necessary inclusion order between header files. If you remove include statements, you will need to worry about the order that you list include statements in your common header.

The include files prevent against multiple inclusion, so you don't need to worry about including the same file twice. The code below is taken from the mainform of the project from the Results section. It shows the include statements that remain in the source, even though the common header already includes them.

```
//---------------------------------------------------------------------
// MAINFORM.CPP
#include <vcl.h>
#include "pch.h"
#pragma hdrstop
```

```
#include <system.hpp>
#include "mainform.h"
#include "About.h"
#include "util.h"
#include "claim.h"
#include "expert.h"
#include "vendor.h"
#include "lawfirm.h"
#include "registry.h"
#include "exceptions.h"
...
...
//---------------------------------------------------------------------------
// MAINFORM.H
#ifndef mainformH
#define mainformH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#include <ComCtrls.hpp>
#include <ToolWin.hpp>
#include <ExtCtrls.hpp>
#include <Buttons.hpp>

#include "defs.h"

//---------------------------------------------------------------------------
class TMDIParent : public TForm
{
__published:
...
```

The point of showing this code is to demonstrate that even though I use a common include file called `PCH.H`, I don't remove existing include statements from my source files. When I create a new source file, I add include statements so that file compiles without relying on includes from the common header. Once the new source file compiles, I insert an include statement for the common header to keep compile times down.

Observe case sensitivity for include statements: Someone posted a comment on the C++Builder IDE group that the compiler observes case sensitity when matching pre-compiled images. If you include the common header with varying case in different units, the compiler will regenerate the pre-compiled image for each one. The next code example demonstrates what you should not do.

```
//---------------------------------------------------------------------------
// MAINFORM.CPP
```

```
#include <vcl.h>
#include "pch.h"
#pragma hdrstop
...

//---------------------------------------------------------------------
// ABOUT.CPP
#include <vcl.h>     // OK. same case as mainform.cpp
#include "pch.h"
#pragma hdrstop
...

//---------------------------------------------------------------------
// SPLASH.CPP
#include <vcl.h>
#include "PCH.H"     // WRONG. mismatched case
#pragma hdrstop
...

//---------------------------------------------------------------------
// LOGON.CPP
#include <Vcl.h>     // WRONG. mismatched case
#include "pch.h"
#pragma hdrstop
...
```

In this example, the compiler will generate and use the same pre-compiled image for `MAINFORM.CPP` and `ABOUT.CPP`, but `SPLASH.CPP` and `LOGON.CPP` will each generate their own pre-compiled image, which will slow down the compile time. The rule of thumb is this: every include file listed above the `#pragma hdrstop` directive should use the same case that other files use. Include statements below the `#pragma hdrstop` directive don't have to match case, because they are not pre-compiled.

Consider adding VCL.H to the common header: The common header that was used in the example code for technique 2 does not include the file `VCL.H`. Each CPP source file includes both `VCL.H` and `PCH.H`, like this.

```
//-----------------------------------------------
#include <vcl.h>
#include "pch.h"
#pragma hdrstop
```

You may prefer to include `VCL.H` from within the common header. If you do, then each CPP file can simply include the common header.

```
//-----------------------------------------------
#include "pch.h"
#pragma hdrstop
```

This is cleaner, and less prone to error because you don't have to worry about which file should be listed first. However, it violates the suggestion from the note *don't remove existing #include statements*. If you ever need to yank out the common header file, you will need to add VCL.H back into every CPP file in your project.

Use a separate CSM file for each project: By default, C++Builder creates a common pre-compiled header file called vcl50.csm in the $(BCB)\lib directory (the name of this file is different with each version of C++Builder). C++Builder will share this pre-compiled header file among all of your projects. In order to take advantage of the techniques in this article, you should configure your projects to create and use their own pre-compiled image file.

You can do tell your project to use its own pre-compiled header file by specifying a file name for the pre-compiled image. This option is on the Compiler tab of the Project-Options dialog box. Change this value from $(BCB)\lib\vcl50.csm to a filename that won't conflict with other projects, such as pch.csm.

There are several advantages to using a separate CSM file for each project. First, it allows you to create a customize your common header file for each project. Secondly, because the CSM file in the lib directory is shared, it tends to grow in size. It is not uncommon to have a shared CSM file that is larger than 30 MB. Lastly, some users have reported that sharing the same CSM across multiple projects is a source of phantom compiler errors. If you are getting strange compiler errors in algorithm.h, you might be able to solve the problem by not sharing your CSM files across multiple projects.

Count the number of #00 files in your project directory: When C++Builder generates a pre-compiled header image, it saves that image to a file with a .CSM extension. It will also save one or more files with an extension of .#?? (ie #00, #01, #02). The number of .#?? files depends on how well you have optimized your pre-compiled headers. If you optimize them perfectly, C++Builder will generate a single file with an extension of .#00. If you don't optimize your files correctly, you will see other files with similar file extenstions (.#01, .#02, etc). The presence of additional #00 files indicates that you have not optmized your headers correctly.

So what are these files? For each unique pre-compiled image in the .CSM file, C++Builder generates one file with a .#?? extension, starting with .#00. These files are usually between 1 and 2 MB's in size. When you have optimized your files perfectly, the .CSM file will contain one and only one pre-compiled image. As a result, you end up with one .CSM file and one file with a .#00 extension. When you don't optimize your projects, the .CSM file may contain many unique pre-compiled images. Each unique image generates one additional .#?? file.

Here are a couple of additional tips regarding these mysterious .#00 files. C++Builder generates these files as its needs them, but it never deletes them. Because these files are fairly large, you should delete them every so often. Also, because C++Builder never deletes them, you may see .#00 files even after you optimize your pre-compiled headers. Third, these files are created in the same directory that the CSM file resides. By default, this is the $(BCB)\lib directory. Lastly, you may want to take a look in your $(BCB)\lib and count how many of these files are lying around. In the previous note, we talked about how C++Builder generates one common .CSM file in the lib directory that all projects use. This shared .CSM file tends to be big, and it also tends to contain many different pre-compiled images. Since each unique image generates a separate .#00 file, you end up with tons of .#00 files in your lib directory. Currently, my BCB4 lib directory contains five of these files (vcl40.#00,vcl40.#01, vcl40.#02, vcl40.#03, and vcl40.#04).

June 1999

Handling differences in integer representation

by John M. Miano

Applications that exchange data across multiple systems must deal with differences in integer representation. The function WriteIntegers() reads three integers from cin and writes them in binary format to the file DATA.DAT. The function ReadIntegers() reads the file and prints the three integers to cout. Both functions are shown in Listing A.

Listing A: WriteIntegers() and ReadIntegers()

```
void WriteIntegers ()
{
        BYTE4 a, b, c ;
        cin >> a >> b >> c ;
        ofstream strm ("DATA.DAT", ios::binary) ;
        strm.write ((char*) &a, sizeof (a)) ;
        strm.write ((char*) &b, sizeof (b)) ;
        strm.write ((char*) &c, sizeof (c)) ;
        return ;
}
void ReadIntegers()
{
                BYTE4 a, b, c ;
                ifstream strm ("DATA.DAT", ios::binary) ;
                strm.read ((char *) &a, sizeof (a)) ;
                strm.read ((char *) &b, sizeof (b)) ;
                strm.read ((char *) &c, sizeof (c)) ;

                cout << "a = " << a
                        << " b = " << b
                        << " c = " << c << endl ;
                return ;
}
```

If you execute WriteIntegers() followed by ReadIntegers() on a PC, everything will work as expected. On the other hand, if you were to run WriteIntegers() on another system, such as a SPARC or a Mac, and then transfer the file to a PC and run ReadIntegers(), the output values wouldn't be the same as those written to the file.

The problem is that two different formats are used to represent multi-byte integers. The Intel 80x86 family, Alpha, and some MIPS processors store the least significant byte first in an integer variable. This format is known as Little Endian. Not surprisingly, systems that store the most significant byte of an integer first are known as Big Endian. The SPARC, 680x0, and some MIPS chips are Big Endian. Table A shows how the value 0x12345678 is stored on both types of systems.

Table A: Two different storage formats

```
Bit 31          Bit 0
|78|56|34|12|   Big Endian
|12|34|56|78|   Little Endian
```

When you create applications that transmit binary data over a network or exchange it in a file, you need to take the integer format into account. For example, graphics formats, such as GIF (Little Endian) and JPEG (Big Endian), specify the byte order to be used.

In a Little Endian system, a small value in an int can be correctly accessed with an overlaid char. The function WhatType() uses this relationship to display the type of integer format used by a processor:

```
void WhatType()
{
        int x = 1 ;
        if (*(char *) &x == 1)
                cout << "Little Endian" << endl ;
        else
                cout << "Big Endian" << endl ;
        return ;
}
```

Converting between Little Endian and Big Endian format is simply a matter of swapping bytes and the conversion is the same in both directions. The following example shows how an Endian conversion function could be implemented for multiple systems. It would be appropriate in applications that exchange data in Little Endian format. On a Big Endian system, the function EndianConversion() converts its parameter value to Little Endian format. On a Little Endian system, this function simply returns its input value. The macro BIGENDIAN would be defined on the command line:

```
typedef unsigned long UBYTE4 ;
#if defined (BIGENDIAN)
inline UBYTE4 EndianConversion (UBYTE4
```

```
        input)
{
        UBYTE4 result =
                ((input & 0x000000FFL) << 24)
        | ((input & 0x0000FF00L) << 16)
        | ((input & 0x00FF0000L) >> 16)
        | ((input & 0xFF000000L) >> 24) ;
        return result ;
}
#else
inline UBYTE4 EndianConversion (UBYTE4
        input)
{
        return input ;
}
#endif
```

If you've ever done any Internet socket programming, you've probably used the functions htonl() and ntohl(). Internet protocols use Big Endian format to represent integers. These functions return their argument on Big Endian systems and do a byte swap on Little Endian systems.

When you need to exchange integers among different systems, you need to pick an integer format to use. One format is just as good as another. You simply need to pick one and be consistent. Use functions to convert between the exchange format and the system format.

May 1999

Sending mail

by Kent Reisdorph

As many C++ programmers have found, sometimes you need to send email from your C++Builder program. You may also want your users to be able to send you mail regarding technical support, diagnostic information, or for any number of reasons.

Unfortunately, VCL doesn't provide built-in support for mail services. Instead, ActiveX controls are provided for this purpose. The ActiveX controls differ between C++Builder 1.0 and C++Builder 3.0. This article will show you how to send mail from your applications, whether you're using C++Builder 1.0 or 3.0.

SMTP basics

The Simple Mail Transfer Protocol (SMTP) is used to send Internet mail. This protocol, like most Internet protocols, was defined many years ago for the UNIX platform. It has evolved some over the years but the basic protocol remains the same. (If you want to learn more about SMTP, search the Internet for RFC 821; this document defines SMTP.) Sending mail via SMTP requires these steps:

- Drop an SMTP control on your form.
- Set the remote host and remote port.
- Set the properties to the address of an SMTP server.
- Connect to the SMTP server.
- Set up the message header.
- Send the message.

First, we'll discuss mail operations in general. Keep in mind that the information in the following sections is general since it applies to both C++Builder 1.0 and 3.0. We'll get to the specifics in later sections when we talk about sending email with either C++Builder 1.0 or 3.0.

Setting the remote host and port

Before you can connect to an SMTP server, you must specify a remote host name and a port. Most of the time, you can just specify mail for the remote host. This tells the SMTP control to connect the default SMTP server for your system. If you want, you can specify a specific mail server. For example, you might set the remote host property to *mail.mycompany.com*, or *mail.myisp.net*. Naturally, these are fictitious names and you should use the actual name of the server to which you're connecting. The port can usually be left set to the default value of 25. Port 25 is the default port for mail systems, so you should rarely need to change this property. If you're connecting to a mail system that's listening on a port other than 25, then set the port property accordingly.

Connecting to an SMTP server

To send an email message, you must first log on to a remote SMTP server. Most SMTP servers don't require a user name and password. This means that you can connect to a mail server without worrying about authentication. Most likely you'll connect to your company's mail server, or, if you're going through an Internet Service Provider, that ISP's mail server.

To connect to a server, you call the Connect method of the SMTP control. We'll talk more about connecting to an SMTP server a little bit later.

Setting up the message header

An email message contains a message header that has several fields. For the most part, the field names and their values are obvious. For example, it's readily apparent what the To, From, CC, and Subject fields contain. A couple of header fields, however, require further explanation. For example, the Date field usually indicates the time in either GMT (Greenwich Mean Time) or as an offset from GMT. The Date field for a typical email message might look like this:

```
Date: Fri, 02 Jan 1998 15:25:14 -0600
```

Notice the -0600 at the end of the line. This is the offset, in hours, from GMT. It tells you that the message was sent from a time zone that is GMT minus six hours (US Central Standard Time). Otherwise, you'll just see the time specified in GMT; for example:

```
Date: Fri, 02 Jan 1998 21:04:01 GMT
```

Whichever format you prefer, it's up to you to build the date and time string when you set the value of the Date field. Another field worth mentioning is the Content-Type field. This field usually looks like this:

```
Content-Type: text/plain; charset=US-ASCII
```

In the case of the SMTP control that comes with C++Builder 1.0, you'll have to set the value of this field. In the case of the SMTP control that comes with C++Builder 3.0, this field is set for you.

Finally, there's the Message-ID field. This field specifies a unique ID for each email message. There's no rule on what the message ID should contain, but it usually has a unique ID number followed by the host name from which the message originated. Here are some examples of Message-ID fields:

```
Message-ID: <000B3C9D.3371@mcp.com>
Message-ID: <34AD5B3A.7E9A@worldnet.att.net>
Message-ID: <349F0E4C.1DA10FD5@mtnfolk.com>
```

You should come up with some scheme for generating unique message ID numbers. Note that the SMTP control that comes with C++Builder 3.0 doesn't generate the Message-ID field.

Naturally, an email message contains a message body. It might also contain attachments. We'll discuss

these in more detail as we look at how to send mail in the following sections.

Sending mail with C++Builder 1.0

The Internet controls that come with C++Builder 1.0 are provided by a company called NetManage. There are several problems with these controls, the primary problem being a lack of any useable documentation. The second problem is that Borland didn't go to any particular effort to fully implement the interfaces to these controls. For those reasons, trying to send mail with the NetManage controls can be a very frustrating experience. The following section will describe the steps required to send mail with the TSMTP control. The TSMTP control, however, doesn't allow file attachments to email messages.

Getting started

To begin, you need to place a TSMTP control on your form. You can find this control on the Internet tab of the Component Palette. Once you have the control on your form, you can set the RemoteHost and RemotePort properties to the SMTP server to which you want to log on. As I said earlier, usually you can leave the RemoteHost property set to the default value of mail, and the RemotePort property set to 25.

Connecting to the SMTP server is simple. You just need to call the Connect method of TSMTP. There *is* one thing you need to be aware of when calling this method, though, which requires some explanation. The Connect method takes two parameters, which are instances of the Variant class. The first parameter is used to specify the remote host, and the second parameter is used to specify a remote port. These parameters are optional--you won't normally supply them because you'll use the RemoteHost and RemotePort properties instead. Since you aren't using the parameters, you need to provide a dummy value for them. You can't just use an empty string or NULL, because TSMTP is expecting a Variant. You have to create a minimal Variant object and pass that object to the Connect method. Here's how the code would look:

```
Variant dummy;
dummy.VType = varError;
dummy.VError = DISP_E_PARAMNOTFOUND;
SMTP1->Connect(dummy, dummy);
```

This will ensure that the call to Connect will succeed without error. Attempting to pass a NULL, a 0, or an empty string will result in an exception at runtime.

When the connection is established, you'll get an OnStateChanged event with a State parameter of prcConnected. Once you know you're connected, you can send the mail message. If you have a permanent Internet connection, then you should be connected right away. If you're using dial-up networking, then Windows should automatically dial when you call the Connect method.

Setting up the message header

Now you need to set up the mail message header. As discussed earlier, the message header contains

fields for the To address, the From address, the CC field, the Subject field, the Date field, and so on.

The TSMTP control has a rather obtuse method of setting these parameters. The DocInput property of TSMTP itself has a Variant property called Headers. (Both DocInput and Headers are OLE objects.) You add parameters to the Headers property using the Add method, and then pass it to the SendDoc method when you send a message. If we could call the Add method directly, it would look like this:

```
Headers.Add("From", "jimbo@redneck.com");
```

We can't do that, though, because Headers is a Variant and doesn't know anything about the Add method. The OleProcedure method of the Variant class allows us to call a method of an OLE object returned as a Variant. We can use OleProcedure, then, to call the Add method. That probably doesn't make much sense, so an example is in order. Here's the code required to set the From field of an email message:

```
Variant Headers = SMTP1->DocInput.OlePropertyGet("Headers");
Headers.OleProcedure("Add", "From", "jimbo@redneck.com");
```

This code first creates a Variant object for the Headers property of the DocInput object. After that, OleProcedure is called to call the Add method of the Headers property. The first parameter of the OleProcedure method is the name of the OLE procedure to call. In this case, the second and third parameters are the parameters to pass to the procedure (OleProcedure allows up to nine parameters).

We repeat this step for each of the fields we want to set. The To and CC fields can contain multiple email addresses. Just separate each address with a comma.

I should note that the message header fields aren't fixed. You can add anything you want to a message header. There are certain header fields that are customary, though, and you should provide those header fields. Examine email message headers to see what fields you should provide when constructing an email message header.

Sending the message

Once you've connected to the SMTP server and have filled in all of the header fields, you can send the message. You send a message with the SendDoc method of TSMTP. Here's how it looks:

```
Variant dummy;
dummy.VType = varError;
dummy.VError = DISP_E_PARAMNOTFOUND;
SMTP1->SendDoc(dummy, Headers, Memo->Text, "", "");
```

Notice that the first parameter is another dummy Variant instance. This parameter can be used to specify a URL that contains the document to send, but you'll rarely use this parameter, so the dummy argument is used. The second parameter is the Headers object you created and filled out earlier. The third parameter is used to specify the message text.

In this example, the contents of the message are contained in a memo control so we just pass Memo->Text for the message parameter. The fourth parameter can be used to specify a text file that contains the text to be sent. Note that this parameter isn't used to specify an attachment. The text file is simply read and the contents of the file are sent as the body of the message. In this case, we aren't using a text file so an empty string is sent for this parameter. The fourth parameter doesn't apply in this case, so it's also set to an empty string.

Oddly enough, the TSMTP control doesn't have an event that tells you a message was sent successfully. What you can do, though, is monitor the Busy property as follows:

```
SMTP1->SendDoc(dummy, Headers, Memo1->Text, "", "");
while (SMTP1->Busy)
        Application->ProcessMessages();
StatusBar->SimpleText = "Message Sent!";
```

While I don't really like this approach, it's the best we can do under the circumstances.

The TSMTP control has few useful events. The OnStateChanged event will tell you when you're connected to a mail server, and when you disconnect. The OnError event will tell you if a mail operation failed. These are probably the only events you'll use regularly.

Sending the message

Listing A and Listing B contain the header and source file for a C++Builder 1.0 program that uses the TSMTP control to send a mail message. The program has just a main form that contains the primary components in
Table A.

Table A: Main form's primary components

Component	Name
TSMTP	SMTP
TButton	ConnectBtn
TButton	DisconnectBtn
TButton	SendBtn
TEdit	ToEdit
TEdit	FromEdit
TEdit	SubjectEdit
TMemo	Message

Place these components on a form and then enter the code from Listing B.

Sending mail with C++Builder 3.0

Sending mail with C++Builder 3.0 is more straightforward than sending mail with C++Builder 1.0. This is because the SMTP control provided with C++Builder 3.0 is the TNMSMTP control that NetMasters provided. This control is much more intuitive to use than the NetManage control that comes with C++Builder 1.0. Connecting to an SMTP server with the TNMSMTP control is a trivial exercise:

```
SMTP->Host = "mail";
SMTP->Connect();
```

It's necessary to set the Host property because the TNMSMTP control doesn't have a default value for this property. The Port property defaults to 25 and you shouldn't have to change it. The OnConnect event will be generated when you've successfully connected to the server.

Setting up the message header is straightforward, as well. The PostMessage property of TNMSMTP is a class which itself contains properties representing the message header fields. Table B lists the properties of the PostMessage class.

Table B: Properties of PostMessage

Property	Use
FromAddress	The email address of the sender
FromName	The name of the sender in plain text
ToAddress	One or more email addresses to which you're sending the message
ToCarbonCopy	One or more email addresses that will receive carbon copies of the message
ToBlindCarbonCopy	One or more email addresses to receive blind carbon copies of the message
Body	The body of the message
Attachments	One or more files to attach
Subject	The message subject
LocalProgram	The name of the program sending the email

Filling out the header is a simple matter of providing values for the PostMessage properties:

```
SMTP->ClearParameters();
SMTP->PostMessage->FromAddress = FromEdit->Text;
SMTP->PostMessage->FromName =  FromEdit->Text;
SMTP->PostMessage->ToAddress->Add(ToEdit->Text);
SMTP->PostMessage->Subject = SubjectEdit->Text;
SMTP->PostMessage->LocalProgram = "My Mailer";
SMTP->PostMessage->Body->Assign(MsgMemo->Lines);
SMTP->PostMessage->Attachments->Add("test.txt");
```

Notice that the ClearParameters method is called first to clear all of the properties of the PostMessage class. In this example, the values of many of the fields are taken from Edit components on a form. The

Body field is filled from the lines of a Memo component.

Once you have the mail header set up, you can send the mail message. This is also a trivial exercise with the TNMSMTP control:

```
SMTP->SendMail();
```

You can respond to the OnSuccess or OnFailure events to determine whether or not the mail message was sent successfully. The TNMSMTP control has a rich set of events that you can use to monitor all aspects of the mail operation. (Our Web site contains a C++Builder 3.0 project, which sends a simple email message with TNMSMTP. Go to the www.zdjournals.com and click on the Source Code hyperlink.)

Deploying a mail application

As I said earlier, the SMTP controls provided with C++Builder are ActiveX controls. You need to take special steps to deploy an application using ActiveX controls. You need to be sure that you register the ActiveX controls when you install your program. This can be done with the REGSRVR.EXE application provided with C++Builder, or with a good installation program. We suggest using a commercial installation program. A commercial installation program takes most of the work out of installing ActiveX controls.

The most important thing about deploying an application using ActiveX controls is knowing which files to ship. Check the ActiveX documentation for details on which files to ship. If you want to avoid the ActiveX hassle, you should check out the various Internet sources for freeware, shareware, or commercial native VCL components.

Conclusion

Sending mail using the ActiveX controls that ship with C++Builder is fairly easy once you know how. Unfortunately, not all applications can benefit from mail support. For those that can, however, it can make the difference between your application and that of the competition.

Listing A: SMTPU1.H

```
//---------------------------------------------
#ifndef SMTPU1H
#define SMTPU1H
//---------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include <vcl\ISP.hpp>
#include <vcl\OleCtrls.hpp>
#include <vcl\ComCtrls.hpp>
//---------------------------------------------
```

```
class TForm1 : public TForm
{
  __published:  // IDE-managed Components
        TSMTP *SMTP1;
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TButton *ConnectBtn;
        TButton *DisconnectBtn;
        TButton *SendBtn;
        TEdit *ToEdit;
        TEdit *FromEdit;
        TEdit *SubjectEdit;
        TMemo *Message;
        TStatusBar *StatusBar;


        void __fastcall SMTP1StateChanged(
                TObject *Sender, short State);

        void __fastcall ConnectBtnClick(
                TObject *Sender);
        void __fastcall DisconnectBtnClick(
                TObject *Sender);
        void __fastcall SendBtnClick(TObject *Sender);

  private:  // User declarations
  public:   // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: SMTPU1.CPP

```
//---------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "SMTPU1.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------

void __fastcall TForm1::SMTP1StateChanged(TObject *Sender, short State)
{
  switch (State) {
    case prcConnected : {
      DisconnectBtn->Enabled = true;
      SendBtn->Enabled = true;
      ConnectBtn->Enabled = false;
      break;
    }
    case prcDisconnected : {
      DisconnectBtn->Enabled = false;
      SendBtn->Enabled = false;
      ConnectBtn->Enabled = true;
    }
  }
  StatusBar->SimpleText = SMTP1->StateString;
}
//---------------------------------------------

void __fastcall TForm1::ConnectBtnClick(TObject *Sender)
{
  Variant dummy;
  dummy.VType = varError;
  dummy.VError = DISP_E_PARAMNOTFOUND;
  SMTP1->Connect(dummy, dummy);
}
//---------------------------------------------

void __fastcall TForm1::DisconnectBtnClick(TObject *Sender)
{
  SMTP1->Quit();
}
//---------------------------------------------

void __fastcall TForm1::SendBtnClick(TObject *Sender)
{
  Variant dummy;
  dummy.VType = varError;
  dummy.VError = DISP_E_PARAMNOTFOUND;
  Variant Headers = SMTP1->DocInput.OlePropertyGet("Headers");
  Headers.OleProcedure("Clear");
  Headers.OleProcedure("Add", "To", ToEdit->Text);
```

```
  Headers.OleProcedure("Add", "From", FromEdit->Text);
  Headers.OleProcedure("Add", "Subject", SubjectEdit->Text);
  Headers.OleProcedure("Add", "Date", Now().DateTimeString());
  Headers.OleProcedure("Add", "Content-Type", "text/plain; charset=us-ascii");
  SMTP1->SendDoc(dummy, Headers, Message->Text, "", "");
  while (SMTP1->Busy)
    Application->ProcessMessages();
  StatusBar->SimpleText = "Message Sent!";
}
//---------------------------------------------
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

May 1999

Transferring form data

by Kent Reisdorph

Any serious Windows application will have one or more forms. Most forms are used to retrieve information from the user (aside from simple forms like an About form). Your program must have some way of extracting information from the form and, in some cases, sending information to the form. This article will discuss various ways of transferring information from the main form to a secondary form, and vice versa.

Simple forms: one-way data

In the case of a simple form, you don't really need to do much work to transfer data. Let's use a password form as an example. Invoking this type of form from the main form and transferring data might look something like this:

```
PWForm->ShowModal();
if (PWForm->PWEdit->Text == "bubba")
        // password failed
else
        // password succeeded
```

Here, we read the value of the password edit component directly through the password form's pointer. This is a one-way transfer of data because we're only reading the data on the form and aren't making any attempt to set data on the form.

This situation is only slightly more complicated when you take control of creating the form yourself (in cases where the form isn't auto-created at application startup). Here's how the code might look:

```
TPWForm* form = new TPWForm(this);
form->ShowModal();
if (form->PWEdit->Text == "bubba")
        // password failed
else
        // password succeeded
delete form;
```

The only item of significance here is that you must remember to extract any data from the form before you delete the form's pointer. Once you delete the form's pointer, any attempts to access the pointer will result in an access violation or, in extreme cases, a complete system crash.

By the way, I never let the VCL auto-create my forms in real-world applications. I always allocate the memory for the form just before I show the form and free that memory as soon as I'm done with the form.

Simple forms: two-way data

In the previous example, we only transferred data one way--from the form to the main form. In many circumstances, you'll need to assign values to controls on a form before the form is shown, and retrieve the values for those controls when the form is closed.

Let's say, for example, that you had an application with an Options form. The Options form, naturally, would allow the user to specify options that control how the application behaves. For the sake of argument, let's say you had an application that dials a remote machine in order to transfer some data via a modem. The user options for this type of program might include the phone number to dial, the name of the TAPI device used to make the connection, and the number of times to retry dialing if the number is busy. These settings would be stored in the Registry, in an INI file, or in a user-defined configuration file. The options would be read from the storage medium to variables in the main form's class at application startup. When the application closes, the options would be saved to persistent storage for retrieval the next time the application runs. Transferring the data to and from the Options form will look something like this:

```
TOptionsForm* form = new
        TOptionsForm(this);
form->PhoneNumber->Text = PhoneNumber;
form->TapiDevice->Text = TapiDevice;
form->DialRetries->Text = DialRetries;
if (form->ShowModal() == mrOk) {
  PhoneNumber = form->PhoneNumber->Text;
  TapiDevice = form->TapiDevice->Text;
  DialRetries =
  form->DialRetries->Text.ToInt();
}
delete form;
```

Here, we're only dealing with three controls on the form, and yet the code already looks messy. If you had a large number of components on a form, the code would be even more unwieldy. Plus, we aren't showing the code required to save and load the data from persistent storage. That's a lot of code in the main unit of your application that probably doesn't need to be there at all. Now, let's look at an alternative to this method of data transfer.

Encapsulation

Before describing a better method of data transfer, a few words about encapsulation are warranted. Encapsulation is one of those Object Oriented Programming (OOP) terms that we're all familiar with. *Encapsulation* means using an object to perform a specific task whenever possible. The object takes the code required to carry out that task and raises it to a level that's much easier to deal with.

In the context of this article, encapsulation means that you should, whenever possible, write your forms so that they're completely self-contained and don't need to have interaction with the main form at all. This is the ideal way to deal with forms in your applications. There are many types of forms, however, where this approach isn't practical because data needs to be transferred between the main form and the form.

Data transfer the OOP way

Let's go back to our previous example of an application that dials a remote machine. How can we use encapsulation to make the code cleaner? One way is to create a class that holds the options for the application. The class could have data members for each of the application's options. Further, the class could contain methods to save the options to the Registry or to a file and to read those options back again when needed.

An instance of the class can be created in the main form so that the main form has access to the options. A pointer to the class can be passed to the Options form so that it also has access to the options. Setting up this type of arrangement requires several steps. We'll examine each of these steps in the following sections.

Create a transfer class

The first step is to create a class that will serve as the transfer mechanism for your Options form. First, let's look at the declaration for the class:

```
class TAppOptions {
        public:
                String PhoneNumber;
                String TapiDevice;
                int DialRetries;
                TAppOptions();
                LoadFromRegistry();
                SaveToRegistry();
};
```

As you can see, we've created data members for each option on the Options form. We've also added methods to load from the Registry and save to the Registry. Note that this class violates OOP principles by making all data members public, but for simple examples, this is an acceptable practice.

An instance of this class will be created in the main form of the application. Typically, this would be done at application startup in either the main form's constructor or in its OnCreate event handler. At creation time, the previous options could be read from persistent storage. For example:

```
void __fastcall
TMainForm::FormCreate(TObject *Sender)
{
  Options = new TAppOptions;
  Options->LoadFromRegistry();
}
```

The application's options are now loaded into the class represented by the Options variable and are available for the main form's use. The Options variable is declared as a member of the main form's class.

Naturally, you'll want to destroy the TAppOptions instance before the application terminates. Either the main form's destructor or the OnDestroy event handler are logical places for this code. Before you destroy the object, however, you'll want to save the current options to persistent storage. Here's how the code would look when placed in an OnDestroy event handler:

```
void __fastcall
TMainForm::FormDestroy(TObject *Sender)
{
  Options->SaveToRegistry();
  delete Options;
}
```

Using this technique, the application's options are always loaded at application startup and saved again at application termination. As an alternative, you could place the code to load the options from persistent storage in the class constructor and the code to save the options in the class destructor.

Modify the Option form's class

In order to pass an instance of the TAppOptions class to your Options form, you'll need to modify the Options form's constructor. Simply add a pointer to the TAppOptions class to your constructor. Your Options form class must also contain a TAppOptions pointer so you have access to the instance throughout the class. The modified TOptionsForm class looks like this (the form's component declarations have been removed to save space):

```
class TOptionsForm : public TForm
{
__published:

  // IDE-managed Components
  // component declarations here
private:
  // User declarations
  TAppOptions* Options;
public:
  // User declarations
  __fastcall TOptionsForm(TComponent* Owner, TAppOptions* options);
};
```

Notice that we've added a TAppOptions pointer to the private section of the class. Notice, also, that the declaration for the constructor has been expanded to allow us to pass a TAppOptions pointer from the main form.

Let's skip ahead just a bit and explain how the Options form will be used in the main form. Using the modified TOptionsForm class requires that you create an instance of the Options form at runtime. Since the Options form now has a specialized constructor, you can't rely on the VCL's auto-creation feature for forms. Here's how the code in the main form would look:

```
void __fastcall
TMainForm::Options1Click(TObject *Sender)
{
  TOptionsForm* form = new TOptionsForm(this, Options);
  form->ShowModal();
  delete form;
}
```

This code assumes that the Options variable has already been instantiated, as discussed in the previous section.

Writing the Options form constructor

Next, we must write the body of the TOptionsForm constructor. First, we assign the TAppOptions pointer passed in the constructor to our internal Options variable. We save the pointer because we need access to it later when the form is closed. Then, we assign values contained in the Options class to the various controls on the Options form. Here's how the constructor looks:

```
__fastcall TOptionsForm::TOptionsForm(
        TComponent* Owner, TAppOptions*
        options) : TForm(Owner)
{
  Options = options;
  PhoneNumberEdit->Text = Options->PhoneNumber;
  TapiDeviceEdit->Text = Options->TapiDevice;
  DialRetriesEdit->Text = Options->DialRetries;
}
```

As you can see, we simply take each data member in the Options class and assign its value to a corresponding component on the Options form. This part of the operation is fairly simple.

Saving the new options

Now, we need to take some action when the form is closed. A form like an Options form can be closed in one of two ways--with the OK button or with the Cancel button (the form's close box is the same as clicking the Cancel button). Naturally, we need to account for both possibilities.

If the form is closed with the Cancel button, then no further action needs to be taken. The user can change any or all fields on the Options form, but if the Cancel button is clicked, all those changes are effectively abandoned. By simply not acting, we've already accounted for the user closing the form with the Cancel button or with the form's close box.

If the form is closed with the OK button, on the other hand, we need to act to ensure that the new options are implemented. We simply provide an OnClick event handler for the OK button and copy data from the form's controls to the Options class instance. This is essentially the opposite of the code we wrote for the

form's constructor. For example:

```
void __fastcall
TOptionsForm::OKBtnClick(TObject *Sender)
{
  Options->PhoneNumber = PhoneNumberEdit->Text;
  Options->TapiDevice = TapiDeviceEdit->Text;
  Options->DialRetries =  DialRetriesEdit->Text.ToIntDef(0);
  Close();
}
```

Since our local Options variable is simply a pointer to the instance created in the main class, this code updates the main class' instance directly. When the Options form closes, the main form's Options variable already contains the new options. The main form doesn't have to change at all to account for the new options. This is where encapsulating the application's options in the Options form has great benefit. The Options form does all the work of updating the options and the main form doesn't have to do any extra processing.

If there's a drawback to this mechanism, it's that you need to create a transfer class for each of your application's forms. Still, this method is preferred over those discussed earlier.

Example

Transferring data to and from your forms is something that's required in nearly every non-trivial C++Builder application. Our example application for this article transfers data to and from an options form using a transfer class. Listing A contains the header for the application's main form. Listing B contains the source unit for the main form. Listing C shows the header for the application's Options form and Listing D shows the source unit for the Options form. We've placed the entire implementation of the TAppOptions class in the header for the Options form to save space. Normally, however, you'd place this class in its own unit.

Listing A: MAINU.H

```
//---------------------------------------------
#ifndef MainUH
#define MainUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#include "OptionsU.h"
//---------------------------------------------
class TMainForm : public TForm
{
__published:    // IDE-managed Components
        TMainMenu *MainMenu1;
```

```
        TMenuItem *File1;
        TMenuItem *Exit1;
        TMenuItem *Tools1;
        TMenuItem *Options1;
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
        void __fastcall Exit1Click(TObject *Sender);
        void __fastcall
               Options1Click(TObject *Sender);
private:        // User declarations
        TAppOptions* Options;
public:         // User declarations
        __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------
#endif
```

Listing B: MAINU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;

//---------------------------------------------
__fastcall
TMainForm::TMainForm(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
  // Create an instance of TAppOptions and
  // load from the registry.
  Options = new TAppOptions;
  Options->LoadFromRegistry();
}
//---------------------------------------------
void __fastcall TMainForm::FormDestroy(TObject *Sender)
{
  // Save the options to the registry and
  // then delete the TAppOptions object.
  Options->SaveToRegistry();
```

```
  delete Options;
}
//---------------------------------------------
void __fastcall TMainForm::Exit1Click(TObject *Sender)
{
  Close();
}
//---------------------------------------------
void __fastcall
TMainForm::Options1Click(TObject *Sender)
{
  // Create an instance of the TOptionsForm
  // class, passing the Options variable as
  // a parameter. TOptionsForm does the rest.
  TOptionsForm* form = new TOptionsForm(this, Options);
  form->ShowModal();
  delete form;
}
//---------------------------------------------
```

Listing C: OPTIONSU.H

```
//---------------------------------------------
#ifndef OptionsUH
#define OptionsUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Registry.hpp>
//---------------------------------------------
// The registry key where we will save the data.
const String RegKey =
  "Software\\ZDJournals\\DialogTransferEx";

// The TAppOptions class for transferring data.
class TAppOptions {
        public:
           String PhoneNumber;
           String TapiDevice;
           int DialRetries;
           TAppOptions() {
              PhoneNumber = "";
              TapiDevice = "";
              DialRetries = 0;
           }
           LoadFromRegistry() {
              TRegistry* reg = new TRegistry;
              reg->OpenKey(RegKey, true);
```

```
            if (reg->ValueExists("PhoneNumber")) {
               PhoneNumber =
                  reg->ReadString("PhoneNumber");
                TapiDevice =
                   reg->ReadString("TapiDevice");
               DialRetries =
                   reg->ReadInteger("DialRetries");
             }
             delete reg;
         }
         SaveToRegistry() {
            TRegistry* reg = new TRegistry;
            reg->OpenKey(RegKey, true);
            reg->WriteString(
               "PhoneNumber", PhoneNumber);
            reg->WriteString(
               "TapiDevice", TapiDevice);
            reg->WriteInteger(
               "DialRetries", DialRetries);
            delete reg;
         }
};

class TOptionsForm : public TForm
{
__published:    // IDE-managed Components
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TEdit *PhoneNumberEdit;
        TEdit *TapiDeviceEdit;
        TEdit *DialRetriesEdit;
        TButton *OKBtn;
        TButton *CancelBtn;
        void __fastcall
                CancelBtnClick(TObject *Sender);
        void __fastcall OKBtnClick(TObject *Sender);
private:        // User declarations
        TAppOptions* Options;
public:         // User declarations
        __fastcall TOptionsForm(TComponent* Owner, TAppOptions* options);
};
//---------------------------------------------
extern PACKAGE TOptionsForm *OptionsForm;
//---------------------------------------------
#endif
```

Listing D: OPTIONSU.CPP

```
//---------------------------------------------
```

```
#include <vcl.h>
#pragma hdrstop

#include "OptionsU.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TOptionsForm *OptionsForm;
//---------------------------------------------
__fastcall TOptionsForm::TOptionsForm(TComponent* Owner, TAppOptions* options)  :
TForm(Owner)
{
        // Save the incoming TAppOptions pointer.
        Options = options;
        // Transfer data from the Option class
        // to the individual controls on the form.
        PhoneNumberEdit->Text =
                Options->PhoneNumber;
        TapiDeviceEdit->Text =
                Options->TapiDevice;
        DialRetriesEdit->Text =
                Options->DialRetries;
}
//---------------------------------------------
void __fastcall TOptionsForm::CancelBtnClick(TObject *Sender)
{
        Close();
}
//---------------------------------------------
void __fastcall TOptionsForm::OKBtnClick(TObject *Sender)
{
        // Transfer data from the individual controls
        // to the Options class.
        Options->PhoneNumber =
                PhoneNumberEdit->Text;
        Options->TapiDevice =
                TapiDeviceEdit->Text;
        Options->DialRetries =
                DialRetriesEdit->Text.ToIntDef(0);
        Close();
}
//---------------------------------------------
```

May 1999

A simple fly-over label component

by Mark G. Wiseman

If you've never written a component for C++Builder, the simple component we create in this article will be a great one to start with. If you're an experienced component writer, you'll be delighted with the usefulness and flexibility of this component.

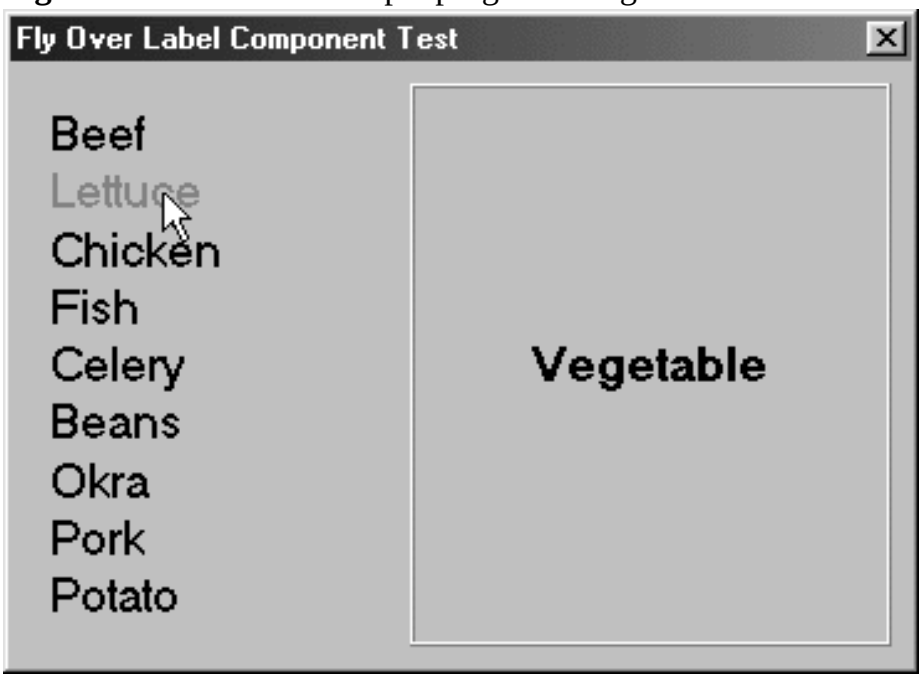
What's a fly-over label?

Originally, the design of this component called for a label that would change color when the mouse was over it. The final design turned out to be more powerful, allowing you to do just about anything when the mouse passed over the component. This is accomplished by adding two events OnMouseEnter, which fires when the mouse first moves over the label, and OnMouseLeave, which fires when the mouse moves off the label.

Component construction

We'll start by letting C++Builder write some of the code for us. Let's create a new component by selecting the File | New menu item in the IDE. Double-click on Component in the New tab. The New Component dialog box will appear. In the dialog box, select TCustomLabel from the dropdown list for Ancestor Type. Enter *TOLabel* for the name of our new component and then enter *ZDJ* for the name of the Palette Page. Click the OK button and C++Builder will write most of the code for TOLabel. The example program shown in Figure A uses several instances of TOLabel.

Figure A: This is the example program using TOLabel.



All we have to do to finish TOLabel is add the two mouse-aware events and publish those properties and events that we want users of TOLabel component to have access to.

So, how do we know when the mouse enters TOLabel's space and when it leaves that space? Inprise has again done most of the work for us. The VCL sends us two messages, CM_MOUSEENTER and CM_MOUSELEAVE. Can you guess what they do?

We need to catch these two messages. We can do this with a message map:

```
BEGIN_MESSAGE_MAP
        MESSAGE_HANDLER(CM_MOUSEENTER,
        TMessage, CMMouseEnter)
        MESSAGE_HANDLER(CM_MOUSELEAVE,
        TMessage, CMMouseLeave)
        END_MESSAGE_MAP(TCustomLabel)
```

When TOLabel receives the messages CM_MOUSEENTER and CM_MOUSELEAVE, the message map will call the functions CMMouseEnter() and CMMouseLeave(), respectively. These two functions simply call a user-supplied event, if one exists, passing the address of the TOLabel object that received the message:

```
void __fastcall TOLabel::CMMouseEnter(
        Messages::TMessage &Message)
        {
        if (FOnMouseEnter) FOnMouseEnter(
        this);
        }

void __fastcall TOLabel::CMMouseLeave(
        Messages::TMessage &Message)
        {
        if (FOnMouseLeave) FOnMouseLeave(
        this);
        }
```

FOnMouseEnter and FOnMouseLeave are data members of TOLabel that hold the user-assigned functions for the events. The user assigns the values to FOnMouseEnter and FOnMouseLeave through two properties published by TOLabel:

```
__property TNotifyEvent OnMouseEnter =
        {read = FOnMouseEnter, write =
        FOnMouseEnter};
__property TNotifyEvent OnMouseLeave =
        {read = FOnMouseLeave, write =
        FOnMouseLeave};
```

Just to be safe, we should set FOnMouseEnter and FOMouseLeave to null in the constructor of TOLabel:

```
__fastcall TOLabel::TOlabel(TComponent*
        Owner) : TCustomLabel(Owner)
        {
        FOnMouseEnter = 0;
        FOnMouseLeave = 0;
        }
```

Finally, we finish TOLabel by publishing the normal properties of a TLabel component. These are declared as protected in TCstomLabel; we need to declare them as published in TOLabel:

```
__published:
        __property Align;
        // etc. ...
```

All the code for TOLabel can be found in Listings A and B.

Listing A: TOLabel include

```
#ifndef OLabelH
#define OLabelH

#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>


class PACKAGE TOLabel : public TCustomLabel
        {
        private:
                TNotifyEvent FOnMouseEnter;
                TNotifyEvent FOnMouseLeave;

        protected:
                void __fastcall CMMouseEnter(Messages::TMessage
                        &Message);
                void __fastcall CMMouseLeave(Messages::TMessage
                        &Message);

        BEGIN_MESSAGE_MAP
                MESSAGE_HANDLER(CM_MOUSEENTER, TMessage,
                        CMMouseEnter)
                MESSAGE_HANDLER(CM_MOUSELEAVE, TMessage,
```

```
                CMMouseLeave)
        END_MESSAGE_MAP(TCustomLabel)

public:
        __fastcall TOLabel(TComponent* Owner);

__published:
        __property TNotifyEvent OnMouseEnter = {read =
                FOnMouseEnter, write = FOnMouseEnter};
        __property TNotifyEvent OnMouseLeave = {read =
                FOnMouseLeave, write = FOnMouseLeave};

        __property Align;
        __property Alignment;
        __property AutoSize;
        __property Caption;
        __property Color;
        __property DragCursor;
        __property DragMode;
        __property Enabled;
        __property FocusControl;
        __property Font;
        __property ParentColor;
        __property ParentFont;
        __property ParentShowHint;
        __property PopupMenu;
        __property ShowAccelChar;
        __property ShowHint;
        __property Transparent;
        __property Layout;
        __property Visible;
        __property WordWrap;
        __property OnClick;
        __property OnDblClick;
        __property OnDragDrop;
        __property OnDragOver;
        __property OnEndDrag;
        __property OnMouseDown;
        __property OnMouseMove;
        __property OnMouseUp;
        __property OnStartDrag;
};


#endif
```

Listing B: TOLabel source

```
#include <vcl.h>
#pragma hdrstop

#include "OLabel.h"
#pragma package(smart_init)

static inline void ValidCtrCheck(TOLabel *)
        {
        new TOLabel(NULL);
        }


// ---------------------------------------------------------------

__fastcall TOLabel::TOLabel(TComponent* Owner) :
        TCustomLabel(Owner)
        {
        FOnMouseEnter = 0;
        FOnMouseLeave = 0;
        }

void __fastcall TOLabel::CMMouseEnter(Messages::TMessage &Message)
        {
        if (FOnMouseEnter) FOnMouseEnter(this);
        }

void __fastcall TOLabel::CMMouseLeave(Messages::TMessage &Message)
        {
        if (FOnMouseLeave) FOnMouseLeave(this);
        }


// ---------------------------------------------------------------

namespace Olabel
        {
        void __fastcall PACKAGE Register()
                {
                TComponentClass classes[1] = {__classid(TOLabel)};
                RegisterComponents("ZDJ", classes, 0);
                }
}
```

Component installation

Let's install our new component into C++Builder so we can use it in a program. First, select Component | Install Component from the IDE menu. Then, select the New Package tab. The Unit file

name should be OLabel.cpp. The Package file name should be ZDJ. Enter *ZDJ Example Components* for the Package description. Then, click the OK button and allow C++Builder to build and install the package.

As a finishing touch, you can add a custom bitmap to represent TOLabel in the component palette of the IDE. You can use a bitmap of your own or use the one included with the source code for TOLabel.

Using TOLabel

To use TOLabel, just drag it from the component palette and drop it on a form. You can then assign functions to the OnMouseEnter and OnMouseLeave events of the TOLabel object. For example, let' refer to the program shown in Figure A.

Each item on the left, a vegetable or meat, is an instance of TOLabel. When the mouse cursor passes over an item, the item's color changes from black to red and the Category of the item, Vegetable or Meat, is displayed by a TPanel object on the right. When the mouse cursor leaves an item, the item's color is set back to black and the Category displayed on the right is cleared.

If you click on one of the items, a message box will announce your selection. You can see how this is done in Listings C and D for the example program.

Listing C: Example program header

```
#ifndef MainH
#define MainH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "OLabel.h"
#include <ExtCtrls.hpp>


class TMainForm : public TForm
        {
        __published:
                TPanel *Category;
                TOLabel *OLabel;

                void __fastcall OLabelMouseEnter(TObject *Sender);
                void __fastcall OLabelMouseLeave(TObject *Sender);
                void __fastcall OLabelClick(TObject *Sender);
```

```
        public:
                __fastcall TMainForm(TComponent* Owner);
        };


extern PACKAGE TMainForm *MainForm;


#endif
```

Listing D: Example program source

```
#include <vcl.h>
#pragma hdrstop

#include "Main.h"

#pragma package(smart_init)
#pragma link "OLabel"
#pragma resource "*.dfm"

TMainForm *MainForm;


__fastcall TMainForm::TMainForm(TComponent* Owner) : TForm(Owner)
        {
        }

void __fastcall TMainForm::OLabelMouseEnter(TObject *Sender)
        {
        TOLabel *label = dynamic_cast<TOLabel *>(Sender);
        if (!label) return;

        label->Font->Color = clRed;
        Category->Caption = label->Tag == 1 ? "Meat" : "Vegetable";
        }

void __fastcall TMainForm::OLabelMouseLeave(TObject *Sender)
        {
        TOLabel *label = dynamic_cast<TOLabel *>(Sender);
        if (!label) return;

        label->Font->Color = clBtnText;
        Category->Caption = "";
        }

void __fastcall TMainForm::OLabelClick(TObject *Sender)
        {
        TOLabel *label = dynamic_cast<TOLabel *>(Sender);
```

```
        if (!label) return;

        Application->MessageBox(label->Caption.c_str(), "You clicked
                on...", MB_OK);
        }
```

TMainForm, the form in our example program, has three functions that are all assigned to different events of our TOLabel objects. OLabelMouseEnter() is assigned to the OnMouseEnter event, OLabelMouseLeave() to OnMouseLeave, and OLabelClick() is assigned to--you guessed it--OnClick.

Notice that these three functions are used for all nine instances of TOLabel. This is much more efficient than writing a separate function for each instance. Let's look at how these functions work.

First, test all three functions to make sure that the Sender argument passed to them is indeed a pointer to a TOLabel object. We do this by testing the result of a dynamic cast:

```
TOLabel *label = dynamic_cast<TOLabel *>(Sender);
If (!label) return;
```

If, for some reason, Sender isn't a pointer to a TOLabel, the functions just return without doing anything. Next, OLabelMouseEnter() and OLabelMouseLeave() change the color of the label font:

```
label->Font->Color = clRed;
```

and

```
label->Font->Color = clBtnText;
```

OLabelMouseEnter() sets the Caption property of the Category TPanel to Vegetable or Meat based on the Tag property of the TOLabel object. During the design of the form, we set the value of Tag to 0 for vegetables and 1 for meats:

```
Category->Caption = label->Tag == 1 ? "Meat" : "Vegetable";
```

OLabelMouseLeave() simply clears the Caption property of the Category TPanel:

```
Category->Caption = "";
```

OLabelClick() displays a message box showing the Caption of the TOLabel that was clicked:

```
Application->MessageBox(label->Caption.c_str(), "You clicked on...", MB_OK);
```

Conclusion

If we count the number of lines of code we had to actually enter while creating TOLabel, we'll find it to be about fifty lines. And, most of those are just exposing the properties from TCustomLabel. The example program required us to enter less than a dozen lines of code!

April 1999

Version: 1.0 and 3.0

Using RAS, part 2

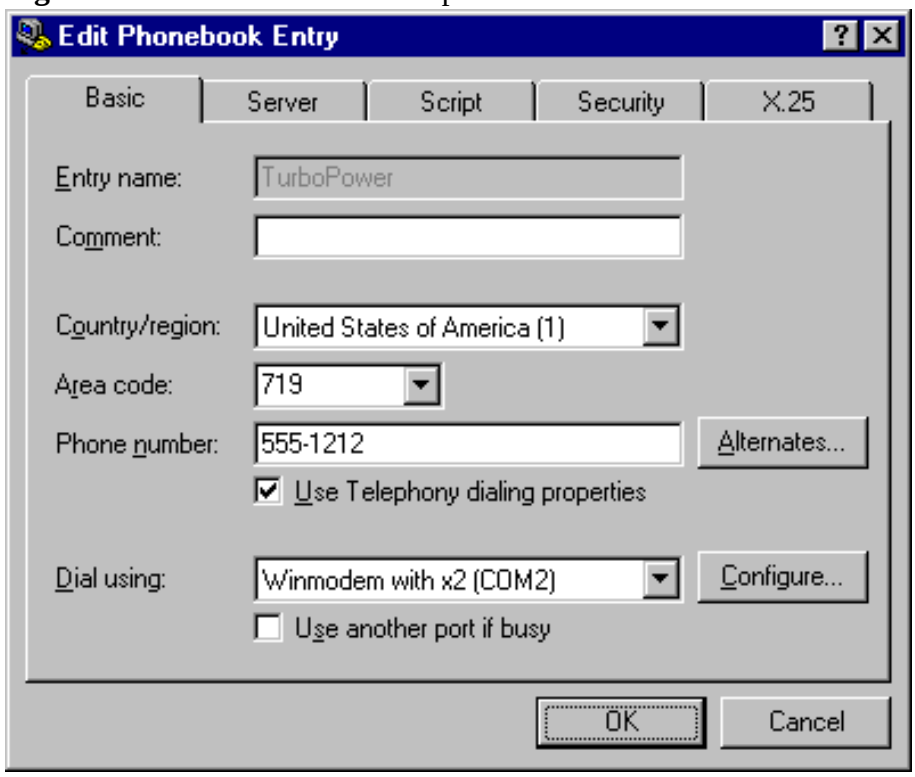
by Kent Reisdorph

Last month, we gave you an introduction to Microsoft's Remote Access Service (RAS) API. We showed you how to make a connection, detect an existing connection, and how to enumerate active connections. This month, we'll continue our discussion on RAS by covering phonebooks in more detail, including how to add, edit, and delete phonebook entries. We'll also introduce you to some of the dialogs that RAS provides for use in your applications.

Got a phonebook handy?

Windows stores dial-up networking information in phonebooks. A phonebook contains one or more phonebook entries. Each phonebook entry describes a connection to a remote machine. This connection information includes the modem used to dial, the phone number to dial, server types, allowed protocols, scripting, and so on. A user may have just one phonebook entry or may have several entries. For example, users may regularly connect to a server at their office (both local and 800 numbers), to an ISP, or to a number of online services, such as AOL, CompuServe, or MSN. Each of these connections requires a separate phonebook entry. Phonebooks are handled differently on Windows NT and Windows 95/98. Under Windows NT, the phonebook is a file on disk, as shown in Figure A.

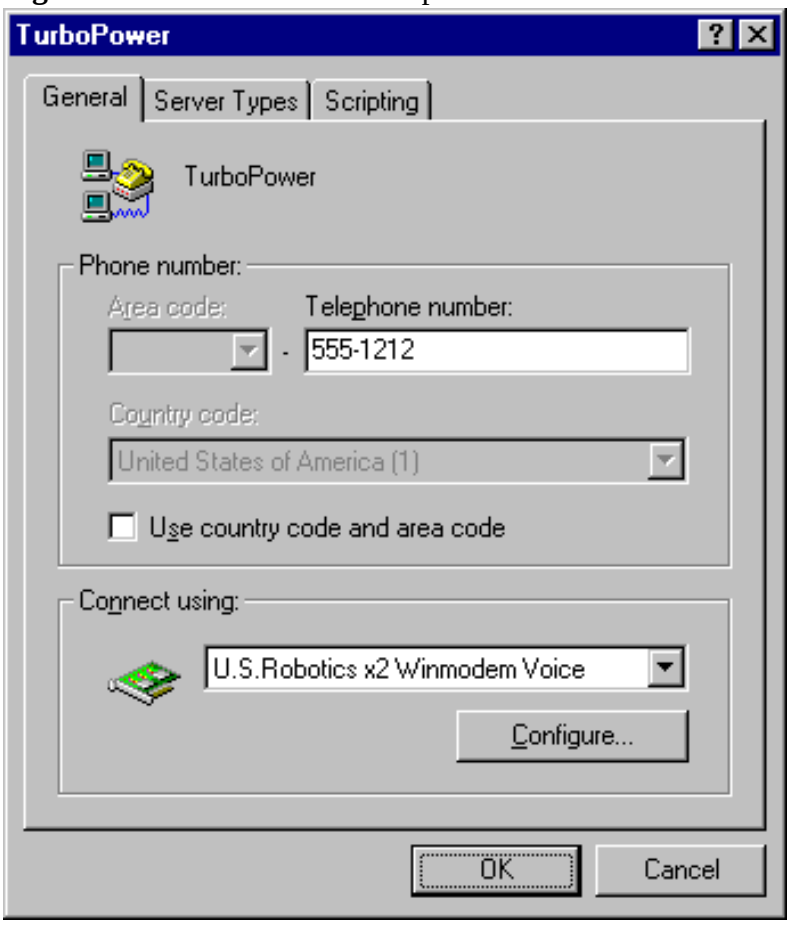
Figure A: This is a Windows NT phonebook.



Phonebook files have a .PBK extension. The phonebook file is a simple text file that describes each entry's properties. The phonebook file is laid out in standard Windows configuration (INI) file format. Multiple phonebooks are possible in Windows NT although, in reality, more than one phonebook is a rarity for most users. Windows NT has a default phonebook so a specific phonebook isn't required.

In Windows 95 and 98 only one phonebook exists. The phonebook isn't stored on disk, but rather is stored in the registry, as shown in Figure B.

Figure B: Here's a Windows 95 phonebook.



When writing applications that use RAS you must account for the fact that the user may have multiple phonebook entries. If only one entry exists in the default phonebook, you can use that entry to make a connection. If, however, more than one phonebook entry exists, you should provide a list of entries for your user to select from. Further, if you know your application will be deployed on NT machines, then you should allow the user to specify a phonebook file.

Dealing with phonebooks

The RAS API provides functions that allow you to create, edit, and delete phonebook entries. Here again, differences exist between Windows NT and Windows 95/98. The following sections describe each of these operations in detail.

Creating a phonebook entry

Creating a phonebook entry is relatively simple, particularly in Windows 95 and 98. The RasCreatePhonebookEntry function provides this service in Windows 95 and 98. In Windows NT, use the RasEntryDlg function.

Note: You can use RasCreatePhonebookEntry on Windows NT systems, even though the Win32 API Help suggests using RasEntryDlg (a call to RasCreatePhonebookEntry maps to RasEntryDlg on Windows NT anyway). Usually, it's wise to heed the recommendations in the Win32 API Help file. The problem in this case, however, is that use of the Windows NT dialog functions requires RASDLG.DLL.

If you build an application on NT that uses the NT functions, it may fail to load on a Windows 95 or 98 machine because RASDLG.DLL doesn't exist on these platforms. Unless your application is specifically targeting Windows NT, you should program for the lowest common denominator. In this case, that means using RasCreatePhonebookEntry instead of RasEntryDlg to create phonebook entries.

When you call RasCreatePhonebookEntry on a Windows 95 or Windows 98 machine, Windows invokes the Make New Connection Wizard. The wizard takes the user through the process of creating a new phonebook entry. This wizard is fairly basic and only asks for a connection name, modem, phone number, and country of origin. It doesn't take into account all of the possible phonebook entry options. Additional options can be set later by editing the phonebook entry (discussed in the next section).

When you call RasCreatePhonebookEntry or RasEntryDlg on a Windows NT machine, Windows will display the New Phonebook Entry dialog box. This dialog is a multi-page dialog and allows the user to specify all aspects of a dial-up connection. The same dialog is displayed regardless of whether you call RasCreatePhonebookEntry or RasEntryDlg.

When the user completes the new phonebook entry information and closes the wizard (Windows 95/98) or dialog (Windows NT), a new entry is created in the phonebook. Here's an example of how to use the RasCreatePhonebookEntry function:

```
RasCreatePhonebookEntry(Handle, 0);
```

As you can see, there's not much to it. The first parameter of this function is a handle to the dialog's owner. In this case, we're passing the form's Handle property for this parameter. The second parameter is used to specify the phonebook in which to create the new entry. In the case of Windows 95/98, this parameter is ignored and should be 0. In Windows NT, this parameter can be the path and filename of a phonebook file. If this parameter is 0 under NT, then the default system phonebook file is used. The following example shows how to use RasEntryDlg in an application targeted for Windows NT:

```
RASENTRYDLG r;
memset(&r, 0, sizeof(RASENTRYDLG));
r.dwSize = sizeof(RASENTRYDLG);
r.hwndOwner = Handle;
r.dwFlags = RASEDFLAG_NewEntry;
RasEntryDlg(0, 0, &r);
```

First, we create an instance of the RASENTRYDLG structure, zero out the memory, and set the dwSize member to the size of the structure. Next, we set the hwndOwner member to the form's Handle property. After that, we set the dwFlags member to RASEDFLAG_NewEntry to tell Windows we're creating a new phonebook entry. Finally, we call RasEntryDlg to invoke the New Phonebook Entry dialog. The first parameter of RasEntryDlg is used to specify the phonebook that the new entry will be added to. As with all of the RAS functions, if you pass 0 for this parameter, Windows will use the default phonebook file. The second parameter is unused when creating a phonebook entry so we pass 0 for that parameter. The final parameter is a pointer to the RASENTRYDLG structure.

Editing an existing phonebook entry

Editing an existing phonebook entry is no more complicated than creating a new phonebook entry. This is true because Windows does most of the work by providing dialogs for editing phonebook entries. Under Windows 95 and 98, you edit phonebook entries by calling the RasEditPhonebookEntry function. You simply pass a handle to the owning application, the phonebook to use, and the name of the phonebook entry to edit. Here's an example:

```
RasEditPhonebookEntry(Handle, 0, "My Connection");
```

The second parameter is used to specify the phonebook in which the entry can be found. As we said earlier, this parameter is ignored under Windows 95 and 98 and should be set to 0. When you call RasEditPhonebookEntry on a Windows 95/98 system, Windows will display a multi-page dialog containing all of the entry's configuration options. When the user clicks the OK button the entry is updated in the registry.

The RasEditPhonebookEntry function can also be used under Windows NT, but here, the Win32 API Help recommends that you use RasEntryDlg to edit phonebook entries on NT applications. A call to RasEntryDlg might look like this:

```
RASENTRYDLG r;
memset(&r, 0, sizeof(RASENTRYDLG));
r.dwSize = sizeof(RASENTRYDLG);
r.hwndOwner = Handle;
r.dwFlags = 0;
RasEntryDlg(0, "MyConnection", &r);
```

This is nearly identical to the code we used when we created a phonebook entry in the previous section. The only difference is the fact that the dwFlags parameter is set to 0 and the second parameter of RasEntryDlg specifies a phonebook entry to edit. Regardless of whether you use RasEditPhonebookEntry or RasEntryDlg on an NT system, Windows will display the Edit Phonebook Entry dialog box. This dialog is functionally identical to the New Phonebook Entry dialog box discussed in the previous section. The only difference is the dialog's title. When the user clicks the OK button, the phonebook entry is updated in the phonebook file.

Deleting phonebook entries

Deleting phonebook entries is accomplished with the RasDeleteEntry function. Using this function is ridiculously simple:

```
RasDeleteEntry(0, "MyConnection");
```

Here, we simply pass 0 for the phonebook name and "MyConnection" as the phonebook entry to delete. Windows doesn't prompt you before deleting the entry, so it's up to you to provide the proper warning dialog to your user before deleting an entry from the phonebook. If you look up RasDeleteEntry in the Win32 API Help, you'll find that Help says it's only available on Windows NT. This isn't accurate. RasDeleteEntry is available on Windows 98 and on Windows 95 with service release OSR2. Be aware, though, that earlier versions of Windows 95 may not have this function.

Windows NT RAS functions

If you're writing an application that will run only on Windows NT, then you have more RAS functions available to you. Only use the RAS functions specific to Windows NT if you're certain your application will only be deployed on NT systems. Applications using the NT functions may fail to load if run on a Windows 95 or Windows 98 machine. At best, they'll likely fail to operate correctly. The following RAS functions are only available on Windows NT:

```
RasConnectionNotification
RasDialDlg
RasEntryDlg
RasEnumAutodialAddresses
RasEnumDevices
RasGetAutodialAddress
RasGetAutodialEnable
RasGetAutodialParam
RasGetCountryInfo
RasGetCredentials
RasGetEntryProperties
RasGetSubEntryHandle
RasGetSubEntryProperties
RasMonitorDlg
RasPhonebookDlg
RasRenameEntry
RasSetAutodialAddress
RasSetAutodialEnable
RasSetAutodialParam
RasSetCredentials
RasSetEntryProperties
RasSetSubEntryProperties
RasValidateEntryName
```

Some of these functions are available only on Windows NT because they're intended to be used when writing a RAS server. As such, we won't attempt to cover all of these functions in this article. There are a few, however, that we'll discuss briefly.

RasPhonebookDlg

The RasPhonebookDlg function can be used to display NT's main dial-up networking dialog box. From this dialog box, users can choose a phonebook entry to dial, can edit, copy, or delete entries, and can initiate dialing. The following example illustrates how to call RasPhonebookDlg:

```
RASPBDLG r;
memset(&r, 0, sizeof(RASPBDLG));
r.dwSize = sizeof(RASPBDLG);
r.hwndOwner = Handle;
RasPhonebookDlg(0, 0, &r);
```

The first two parameters of RasPhonebookDlg are identical to those found in RasEntryDlg. The first parameter is used to specify the phonebook file, and the second is used to specify the phonebook entry that should be initially displayed on the dialog. The final parameter is a pointer to a RASPBDLG structure.

RasDialDlg

Another NT-only function of note is the RasDialDlg function. This function will dial the specified phonebook entry and will display a simple dialog showing the connection status. A call to RasDialDlg looks like this:

```
RASDIALDLG r;
memset(&r, 0, sizeof(RASPBDLG));
r.dwSize = sizeof(RASDIALDLG);
r.hwndOwner = Handle;
RasDialDlg(0, "Compuserve", 0, &r);
```

The first two parameters are identical to those of RasPhonebookDlg. The third parameter can be used to specify a phone number to dial if you wish to override the phonebook entry's phone number. The RASDIALDLG structure is used to specify additional options for the RasDialDlg function. These additional options include the X and Y position of the dialog or a phonebook subentry to dial. On return from RasDialDlg, the dwError member can be examined to determine if an error occurred during dialing.

RasMonitorDlg

The RasMonitorDlg function is used to display a dialog showing the status of a connection. This is the same dialog box you see if you right-click the Dial-Up Networking Monitor application in NT's system tray, and from the shortcut menu, choose Open Dial-Up Monitor. A call to RasMonitorDlg looks like this:

```
RASMONITORDLG r;
memset(&r, 0, sizeof(RASMONITORDLG));
r.dwSize = sizeof(RASMONITORDLG);
r.hwndOwner = Handle;
r.dwStartPage = 0;
RasMonitorDlg(0, &r);
```

The first parameter is the device the status dialog will display initially. If this parameter is 0, the status of the first device is displayed. The dwStartPage parameter of the RASMONITORDLG structure is used to specify the page of the Dial-Up Monitor dialog that's initially displayed.

Conclusion

Listing A shows the header for our test application's main unit. Listing B shows the unit itself.

Understanding the RAS API is important if you're writing applications that use dial-up networking. This is true of most Internet-based applications. Fortunately RAS isn't that difficult to use, once you have a basic understanding of how it works.

Listing A: RASPBKU.H

```
//---------------------------------------------
#ifndef RasPBkUH
#define RasPBkUH
//---------------------------------------------
#include <Classes.hpp>
```

```
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
  __published:  // IDE-managed Components
    TButton *EditBtn;
    TButton *CreateBtn;
    TComboBox *EntriesCB;
    TButton *DeleteBtn;
    TLabel *Label1;
    void __fastcall EditBtnClick(TObject *Sender);
    void __fastcall CreateBtnClick(TObject *Sender);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall DeleteBtnClick(TObject *Sender);

  private:      // User declarations
    void EnumEntries();
  public:               // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------

#if (__BORLANDC__ < 0x530)
  #define PACKAGE
#endif
extern PACKAGE TForm1 *Form1;
//---------------------------------------------

#endif
```

Listing B: RASPBKU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "RasPBkU.h"
#include <ras.h>
#include <raserror.h>
//---------------------------------------------
#if (__BORLANDC__ >= 0x530)
#pragma package(smart_init)
#endif
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
```

```
}
//---------------------------------------------

void __fastcall TForm1::EditBtnClick(TObject *Sender)
{
  // Edit a phone book entry. The entry to
  // edit is the currently selected item
  // in the entries combo box.
  RasEditPhonebookEntry(Handle, 0, EntriesCB->Text.c_str());
}
//---------------------------------------------

void __fastcall TForm1::CreateBtnClick(TObject *Sender)
{
  // Create a new phone book entry.
  RasCreatePhonebookEntry(Handle, 0);
  // Reset the contents of the entries combo.
  EnumEntries();
}
//---------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  // Populate the combo box with a list of
  // current phonebook entries.
  EnumEntries();
}
//---------------------------------------------

void TForm1::EnumEntries()
{
  // Enumerate the entries. This is the same
  // code used in the part 1 of the RAS
  // articles so it is not explained here.
  RASENTRYNAME* entries = new RASENTRYNAME[1];
  entries[0].dwSize = sizeof(RASENTRYNAME);
  DWORD numEntries;
  DWORD size = entries[0].dwSize;
  DWORD res = RasEnumEntries(0, 0, entries, &size, &numEntries);
  if (res == ERROR_BUFFER_TOO_SMALL) {
    // Allocate enough memory to get
    // all the phonebook entries.
    delete[] entries;
    entries = new RASENTRYNAME[numEntries];
    entries[0].dwSize = sizeof(RASENTRYNAME);
    res = RasEnumEntries(0, 0, entries, &size, &numEntries);
    if (res) {
      char buff[256];
      RasGetErrorString(res, buff, sizeof(buff));
      ShowMessage(buff);
    }
```

```
  }
  EntriesCB->Items->Clear();
  for (int i=0;i<(int)numEntries;i++)
    EntriesCB->Items->Add(entries[i].szEntryName);
    EntriesCB->ItemIndex = 0;
}

void __fastcall
TForm1::DeleteBtnClick(TObject *Sender)
{
  // About to delete an entry so display a Yes/
  // No dialog to the user for confirmation.
  char buff[256];
  wsprintf(buff, "Delete phonebook entry '%s'?\nWarning! "
    "Do not delete your regular dialup entry.",
  EntriesCB->Text.c_str());
  int res = MessageDlg(buff, mtConfirmation, TMsgDlgButtons() << mbYes << mbNo, 0);
  if (res == mrYes) {
    // OK to delete the entry.
    RasDeleteEntry(0, EntriesCB->Text.c_str());

    // Reset the list of entries in the combo.
    EnumEntries();
  }
}
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

April 1999

Put a plasma in your form

by Andrea Fasano

With the addition of some new features to BCB3, it's now possible to develop fast, graphic-based applications. In this article, you'll learn how to take advantage of these features, and customize your form with special animated effects. To do so, we'll examine the properties of a TBitmap object, and we'll produce the code that utilizes a simple but amazing graphic effect, the *plasma*.

Fooling the eyes

Our goal is to successfully trick the eyes into seeing movement. When the application starts, it flashes graphics on the screen at a very rapid rate. Each time, the graphics change just a little, so that the user receives the sensation of *motion*. The concept is similar to that of watching cartoons. You see a stream of *static* frames, but because they're flashed so fast on the screen, you see *animation*. The average film displays approximately 24 frames per second (FPS). So, if your application has a good frame rate, you'll be sure to have smooth animation.

How it works

The way to reach our objective is straightforward: we need a method that calculates and draws some graphics in a *back buffer*. When the buffer is filled, we'll copy it into the form's background. We'll repeat these operations to achieve a high FPS and, subsequently, good animation. Therefore, you can understand how crucial a factor *velocity* is, and, how much attention we must pay to speeding up the performance of the application.

Using a back buffer is the first step to optimization; in fact, it's faster to draw all the graphics in a buffer and copy them to the monitor, instead of directly accessing each pixel on the screen. Also, the use of a back buffer avoids the ugly flickering effect (called *double-buffering*).

Finally, we need only three things to draw a special effect in the form:

- A *back buffer* (to be more precise, a TBitmap object)
- A *timer*, used to invoke the drawing routine
- The *code* that produces some cool effect

Step 1: Inserting a back buffer

Inserting a back buffer into the application is easy. After you've started a new application, select the form and change its Name property to *PlasmaForm*. Now, switch to the header file (by pressing [Ctrl][F6] in the Code Editor) and insert this line of code in the *public* section of the form class:

```
...
public:
```

```
Graphics::TBitmap *backbuffer;
...
```

We must also allocate some memory, fill some fields for the back buffer when the application starts, and de-allocate the memory used when the program dies. So, switch to the Object Inspector and double-click OnCreate on the Events palette. Then, insert the following code:

```
void __fastcall
        TPlasmaForm::FormCreate(TObject *Sender)
{
        backbuffer = new Graphics::TBitmap;
        backbuffer->Height = ClientHeight;
        backbuffer->Width = ClientWidth;
        backbuffer->PixelFormat = pf32bit;
        backbuffer->IgnorePalette = true;
}
```

The first three lines of code initialize the back buffer and set its size equal to the form's client area. Then we set the PixelFormat property equal to pf32bit. In this way each pixel of the back buffer is formed by four bytes (one byte for the red component, one for the green, and one for the blue; the last one is reserved). We use 32 bits per pixel because the buffer benefits by the maximum number of colors allowed (about 16 million), and because the CPU manages faster 32 bits per pixel instead of 24 bits per pixel (given by the constant pf24bit).

Finally, the bitmap's property, IgnorePalette, is set to true to achieve a faster drawing. The only drawback is a lower picture quality on 256-color video modes. Now we need to de-allocate the memory used by the back buffer when the application ends. So, come back to the Object Inspector and double-click the OnDestroy event. In the method that appears inside the Code Editor, insert the instruction:

```
void __fastcall
        TPlasmaForm::FormDestroy(TObject
        *Sender)
{
        delete backbuffer;
}
```

And that's all about the back buffer. Now it's ready to be used.

Step 2: Using a timer

As I told you, our application will take advantage of a timer to execute the code that draws the back buffer and copies it inside the form. Of course, there are other ways to perform these actions; for instance, we could use the threads, but they're a lot slower, and a timer is generally simpler to use.

Take a look at the Component Palette and locate the System tab. Click on the watch icon (the first on

the left) and place it inside the form. Since it isn't a visual component, you can put it anywhere. Press [F11] and change the Name to *PlasmaTimer*. Next, insert the value *40* inside the Interval property. This last property determines how much time (in milliseconds) must pass before the OnTimer event occurs. For example, when the timer starts, it waits 40 milliseconds before raising the OnTimer event. Then, the timer waits another 40 milliseconds (ms) and raises the event for the second time and so on, until it gets stopped or the application ends. Setting 40 ms for the Interval property means that in one second, the OnTimer event is raised about 25 times (1000 / 40 = 25).

Next comes the most important part. In the Object Inspector, click the OnTimer event, insert the string DrawForm, and press [Enter]. The Code Editor will appear showing this method:

```
void __fastcall
        TPlasmaForm::DrawForm(TObject *Sender)
{

}
```

So, we'll put the code that implements the plasma effect inside this method! In fact, DrawForm will be called 25 times per second, and that's what we want to do initially.

Now, I want to fix a concept, going a little further: at this point in the article, we've just developed a *template* application. It doesn't matter if you add the code for the plasma effect, to the DrawForm method, or update a label, it only matters that this code is *fast* enough. If it isn't, the application will suffer a loss in performance.

Okay, but what kind of actions do we have to develop inside the DrawForm? Essentially, the answer is three specific actions:

1. Accommodate the back buffer dimensions (if the user resizes the form at execution time).
2. Draw the plasma inside the buffer.
3. Copy it in the form's background.

Next, we'll analyze these three topics.

Step 3: The tricky section

The plasma is a well-known but ever-beautiful effect. It works in a very simple way. First, we must calculate some values using a nice math function--like sin() or cosin()--and put them in a table. For convenience, we'll call it *PlasmaTable* (note that this operation is done only one time--at the start of the application). After that, we scan the entire bitmap and we fill each pixel with the values previously stored in PlasmaTable.

Now, the first tip is to animate the plasma, so we access the table using some indexes, and during the scan of the buffer we increment (or decrement) these indexes.

Also, each time the routine is called, we change the starting point of every index by storing the

starting point in a position variable (so, each index has its own position variable). Each position variable is incremented (or decremented) every time the routine is called.

If you're a little confused, don't worry. Read this section again carefully and take a look at Listing A. Now, focus your attention on the table; it must be declared somewhere. Of course, the best place is the *public* section of the TPlasmaForm class:

```
...
Byte PlasmaTable[256];
...
```

This table's type is the *unsigned char* or *byte*, because the values stored will be used as color intensities. Since we're dealing with RGB colors, and since each component is big as one byte, we have no choice about the table's type. But why does the array hold only 256 elements? That's another little trick. To make the things faster, if you use unsigned char variables as indexes, in order to access the PlasmaTables' values, you don't need to check for the upper and lower bounds of the array. In fact, the increment and the decrement of unsigned variables produces a wrapping around the maximum (or the minimum) value that a variable could hold.

Next, we must discuss the table initialization. Since this step must be done only at the start of the application, a right place is the method FormCreate. To fill the table, we can act in a manner like this one:

```
for (int x=0; x<256; x++)
    PlasmaTable[x] = 30. * (1. + sin(x * 2. * 3.1415 / 256.));
```

You can easily try to modify these values to obtain a different effect, or you can try to use other functions instead of the sin().

Step 4: Drawing, copying, and resizing the back buffer

Now that we understand the inner mechanisms of the plasma, we're finally ready to fill the back buffer. First of all, we'll resize the buffer:

```
buffer->Height = ClientHeight;
buffer->Width = ClientWidth;
```

Next, we need to find a way to write a single pixel. Thanks to a new TBitmap's feature, we can now access every row of the buffer. The property is called ScanLine, and it returns a void pointer to a given row of the bitmap. Since we know the pixel format used internally by the buffer, we can make a safe type casting. For instance, if you want to plot a white pixel at the coordinates (15, 34), you must produce this code:

```
unsigned int *LinePtr;
LinePtr = (unsigned int *) backbuffer->ScanLine[34];
LinePtr[15] = 0x00ffffff;
```

In our case, we use an unsigned int pointer because the back buffer has 32 bits per pixel. Casting the pointer returned by ScanLine to a different type doesn't ensure the right access to the desired pixel. Also, remember that in this pixel format, the first byte represents the *blue* component, the second byte the *green* component, while the third one represents the *red* component; the last byte is unused.

If you look at Listing A, you can see that the final color for each pixel is stored inside PlasmaColor, and is calculated by adding four different values of the table. You can notice the use of a switch statement, which reflects the user's choice during the execution of the application; you can decide the plasma's color by simply clicking a radio group. I put this component inside the form only to clarify the different ways to fill a pixel.

Okay, after we finished filling the buffer, we must copy it inside the form. The simplest (and most efficient) way is given by the use of the Draw method. You only need to specify the destination coordinates and what you want to copy

```
Canvas->Draw (0, 0, backbuffer);
```

That's all! Try to compile the source code and watch what happens. Here's one last trick. To speed up the execution of a method, you should avoid the use of many (and large) local variables. In particular, declare all variables as static: each time the method is called, it won't lose precious time in allocating memory for the local variables. At last, our method performs very few and simple tasks: it resizes the back buffer, scans the entire bitmap, fills each pixel with some color, and copies the buffer to the screen.

Conclusion

In this article, we showed you how to develop a simple graphic effect, and some tricks to improve the application's performance. You can use this knowledge to customize your form (or whatever has a Canvas!) in order to produce, for example, cool splash screens. It's a great exercise to modify the source code, and to draw new and astounding graphics. So, good luck!

Listing A: Code for achieving smooth animation

```
//---------------------------------------------------------------------------
 void __fastcall TPlasmaForm::DrawForm(TObject *Sender)
{
 /*** These variables holds the plasma direction and aspect. ***/
 static Byte PlasmaDir1,
                     PlasmaDir2,
                     PlasmaDir3,
                     PlasmaDir4;
 /*** These variables holds the old values of the plasma position. ***/
 static Byte PlasmaPos1 = 0,
                     PlasmaPos2 = 0,
```

```
                    PlasmaPos3 = 0,
                    PlasmaPos4 = 0;
static unsigned int PlasmaColor,   /*** Final plasma color ***/
                    *LinePtr;      /*** Pointer to a buffer's row of
                                         pixels ***/
static int          x,
                    y;

/*** If the user resizes the windows, we must accomodate the size of the
     back buffer. ***/
backbuffer->Height = ClientHeight;
backbuffer->Width = ClientWidth;

PlasmaDir1 = PlasmaPos1;
PlasmaDir2 = PlasmaPos2;

/*** We scan the entire bitmap, row by row ***/
for (y=0; y<backbuffer->Height; y++)
   {
     /*** We now obtain a pointer to the start of the current row ***/
     LinePtr = (unsigned int *) backbuffer->ScanLine[y];

     PlasmaDir3 = PlasmaPos3;
     PlasmaDir4 = PlasmaPos4;

     for(x=0; x<backbuffer->Width; x++)
     {
       /*** Using the Plasma Table, we obtain the color for the given
            pixel of the row. Try to change this line of code. ***/
       PlasmaColor = PlasmaTable[PlasmaDir1] + PlasmaTable[PlasmaDir2] +
                     PlasmaTable[PlasmaDir3] + PlasmaTable[PlasmaDir4];

       /*** Check for the color choosen in the radio group. ***/
       switch (rgPlasmaColor->ItemIndex)
        {
         case cRed:
               /*** Only the red component. (green = blue = 0) ***/
               LinePtr [x] = (PlasmaColor<<16);
               break;
         case cGreen:
               /*** Only the green component. (red = blue = 0) ***/
               LinePtr [x] = (PlasmaColor<<8);
               break;
         case cBlue:
               /*** Only the blue component. (red = green = 0) ***/
               LinePtr [x] = (PlasmaColor);
               break;
         case cMix:
```

```
         default:
               /*** This is a mix of colours. Try to changes the values,
                    and remember that each color component is big as 1
                    Byte. ***/
               LinePtr [x] = (((255-PlasmaColor)<<16) |  //Red
                             (PlasmaColor<<8)       |  //Green
                             (128+(PlasmaColor>>1)));  //Blue
               break;
         }

       PlasmaDir3 += (Byte) 1;
       PlasmaDir4 += (Byte) 2;
     }

     PlasmaDir1 += (Byte) 2;
     PlasmaDir2 += (Byte) 1;
   }

   /*** This section controls the plasma speed. Try to change these
        values as you like. ***/
   PlasmaPos1 += (Byte) 2;
   PlasmaPos2 -= (Byte) 4;
   PlasmaPos3 += (Byte) 6;
   PlasmaPos4 -= (Byte) 8;

   /*** Finally, the back buffer is copied into the Canvas's Form ***/
   Canvas->Draw (0, 0, backbuffer);
}
//---------------------------------------------------------------------------
```

April 1999

Version: 1.0, 3.0, and 4.0

Using properties in C++ classes

by Kent Reisdorph

It's common knowledge that the VCL uses properties to provide much of its functionality. Properties allow the VCL to do what appears at first to be black magic. For example, when you change the value of the Left property for a form, the form magically moves on the screen. This is because the Left property has a write method associated with it. When you assign a value to the Left property, the write method is called, and code is executed behind the scenes. In this case, the code moves the form on the screen. It's a given that properties work in C++Builder--you use them every time you write a VCL application. But did you know that you can use properties in your own C++ classes? Indeed, you can. This article will explain how.

Properties in C++

The property keyword is intrinsic to the Object Pascal language (the language in which the VCL is written). The C++ language, however, doesn't include the concept of properties. In order to facilitate use of the VCL in C++Builder, Inprise created extensions to the C++ language. One extension is manifest in the __property keyword. This new keyword allows C++ to use properties just as Delphi does. A quick glance at the VCL headers shows you hundreds of declarations with the __property keyword. Using properties in your own C++ classes is extremely easy and can provide an alternative to traditional data hiding in C++. Just about any book that covers OOP will tell you that you shouldn't make your class data members public. Instead, you should write *getters* and *setters*. Getters and setters are class member functions that get and set the value of the class' private data members. By using getters and setters, you can control access to your data members (data hiding). Properties provide an alternative, and much cleaner, way of implementing data hiding.

Property basics

Before we explain how to use properties, let's take a brief moment to review the characteristics of properties. First, properties are unable to store any data in and of themselves. Instead, an underlying data member of the class is used to store the property's value. The property is tied to the data member when the property is declared. Second, properties can be read/write, read-only, or, in extremely rare cases, write-only. Third, some properties use direct access to store and retrieve their data. When a property uses direct access, the property is directly tied to the underlying data member. Reading the property simply returns the value of the data member. Writing to the property simply assigns a new value to the underlying data member.

Finally, some properties use read and/or write methods. Read and write methods are methods that are called when the property is read or written to. Let's go back to the example of a form's Left property. The Left property has a write method called SetLeft. This method is called every time the Left property is assigned a value. The SetLeft method includes code to move the form on the screen (among other things). So it's not black magic at work at all, but simply properties in action.

Note: Direct access and read/write methods can be mixed in the same property. Many properties use direct access for reading the property value, but a write method for writing to the property.

To summarize, if a property includes a write method, then that method will be called every time the property is written to. If the property includes a read method, then that read method will be called every time the property's value is read.

A class that uses properties

Let's first look at a simple class that emulates a point (*x* and *y* coordinates). In straight C++, the class might look like this:

```
class TMyPoint {
        int X;
        int Y;
};
```

That's a pretty simple class, so let's take it a bit further. Let's say that your point class only allows a specific range of *x* and *y* coordinates. For the sake of argument, let's say that both coordinates must fall in the range of 0 to 100. Using straight C++, you'd write the class like this:

```
class TMyPoint {
        private:
                int X;
                int Y;
        public:
                void SetX(int x) {
                        if (x > 100 || x < 0)
                                throw ("Range error");
                        else
```

```
                X = x;
        }
        void SetY(int y) {
                if (y > 100 || y < 0)
                        throw ("Range error");
                else
                        Y = y;
        }
        int GetX() { return X; };
        int GetY() { return Y; };
};
```

As you can see, the class now has private data and uses getter and setter methods to access the data. Note that the setter methods throw an exception if the value passed in is less than 0 or greater than 100. This all works perfectly fine, but the code to make use of this class isn't exactly a thing of beauty:

```
TMyPoint point;
point.SetX(50);
point.SetY(100);
Label1->Caption = String(point.GetX())
        + ", " + String(point.GetY());
```

Now, let's consider this class when written using properties. The new class declaration looks like this:

```
class TMyPoint {
        private:
                int FX;
                int FY;
                void SetX(int x) {
                        if (x > 100 || x < 0)
                                throw ("Range error");
                        else
                                FX = x;
                }
                void SetY(int y) {
                        if (y > 100 || y < 0)
                                throw ("Range error");
                        else
                                FY = y;
                }
        public:
```

```
                __property int X = {read=FX,
                        write=SetX};
                __property int Y = {read=FY,
                        write=SetY};
};
```

There are several items of note in our new TMyPoint class. First, notice that the underlying data members are called FX and FY. The use of the letter *F* preceding the data members is traditional, but you can use any naming convention you like. The important thing to note here, though, is that the underlying data members must use a name other than X and Y because those are our property names.

Next, notice the SetX and SetY property write methods. These methods will be called each time the X and Y properties are assigned a value. The declaration of a write method must conform to a predetermined signature. Specifically, the write method must take a single parameter of the property's type and must return void.

Finally, notice the X and Y property declarations. Here's the X property's declaration again:

```
__property int X = {read=FX, write=SetX};
```

This property is declared of type int because the property's data type must exactly match the underlying data member's type. Notice that we're using direct access to read the property (read=FX), and a write method to write to the property (write=SetX). (Note that if you omit the write statement, the property becomes a read-only property.) The code to implement our newly written class now looks like this:

```
TMyPoint point;
point.X = 50;
point.Y = 100;
Label1->Caption =
        String(point.X) + ", " +
                String(point.Y);
```

This code is much easier to read and is much cleaner than the previous example using a straight C++ class.

Conclusion

Properties allow you to write cleaner code and to perform some specific action when the property is written to or read. In addition, using write methods for your properties allows you to throw exceptions if invalid values are assigned to your properties. One drawback to using properties in your C++ classes is that properties aren't portable to other platforms. If writing portable code is important to you, then you probably don't want to use properties in your C++ classes. For your VCL applications, however, using properties can enhance the usefulness of your C++ classes.

April 1999

Version: 1.0, 3.0, and 4.0

Enumerating components

by Kent Reisdorph

There are several questions that pop up over and over again on the Inprise newsgroups. One such question (in various incarnations) is "How do I clear the text of all edit controls on a form?" This type of question is a common query from Visual Basic programmers moving to C++Builder. Those programmers are accustomed to using VB component arrays and want to accomplish the same thing in C++Builder. This article will explain how to use the VCL's Components property to enumerate all components on a form. We'll also show you how to use the dynamic_cast operator to determine the type of each component on the form.

The components array

The TForm class has two properties that can be used to enumerate a form's components. The ComponentCount property is a read-only property that returns the number of components owned by the form. This includes components placed on the form at design time, and components (visual and non-visual) that are created at runtime. The Components property is an array property. It contains a pointer to every component owned by the form. If you look at the declaration for the Components property, you'll find that it contains a list of TComponent pointers. TComponent is, ultimately, the base class for all components. By storing a list of TComponent pointers, the pointers can be dealt with at their lowest common denominator. In your code you can cast the pointer to any type you want, in order to access the properties of a particular type of component. We'll explain exactly how to do that a bit later in the article.

The Components array is used internally by the VCL. Among other things, the array is used by the VCL when it deletes all owned objects for a form, usually when the form itself is deleted.

Enumerating a form's components

A form's components can easily be enumerated using a simple for loop. The following code enumerates the components on a form and places the component names in a memo:

```
Memo1->Lines->Clear();
for (int i=0;i<ComponentCount;i++) {
        String S = Components[i]->Name;
        Memo1->Lines->Add(S);
}
```

This code is mostly self-explanatory. The loop starts at 0 (the component array is 0-based) and stops at ComponentCount minus 1. Each component's Name property is read and is added to the memo. As is our practice, we've broken the code down to make it easier to understand. Most of the time you would write the code as follows:

```
for (int i=0;i<ComponentCount;i++)
        Memo1->Lines->Add(Components[i]->Name);
```

Keep in mind that dynamically created components will have a blank Name property, unless you've specifically provided a name when you created the component. For example:

```
TLabel* label = new TLabel(this);
label->Top = 20;
label->Left = 20;
label->Parent = this;
label->Caption = "Hello";
```

In this case, the label's Name property isn't assigned a value and will contain an empty string. You can use the pointer returned from the Components array directly if you only need to access properties of the component that are found in the TComponent class. For example, let's say you want to enumerate just those components that have their Tag property set to 1. In that case, the code would look like this:

```
for (int i=0;i<ComponentCount;i++)
        if (Components[i]->Tag == 1)
                Memo1->Lines->Add(Components[i]->Name);
```

Here, only those components with a Tag property equal to 1 are added to the memo. If you want to access the properties of a particular type of component (a TEdit, for example), then you'll need to cast the TComponent pointer. Let's look at that next.

Casting the TComponent pointer

In a real-world application, you'll likely need to cast the pointer returned from the Components array in order to access the individual properties of a particular type of component. Let's say, for example, that you want to clear the text in all edit controls on a form. In that situation, you'll have to cast the TComponent pointer to a TEdit pointer. There's a right way and a wrong way to perform this cast.

The wrong way

The wrong way is to use a C-style cast. This code illustrates the wrong way of casting the pointer returned from the Components array:

```
for (int i=0;i<ComponentCount;i++)
        ((TEdit*)Components[i])->Text = "";
```

This code will work if, and only if, all components on the form are TEdits. If a component other than a TEdit is encountered, an access violation will almost certainly occur when you attempt to access the Text property. Given that, you should never use a C-style cast in this case. (In fact, we would argue that you should never use C-style casts in a C++ program.)

The right way

The right way to perform the cast is with the dynamic_cast operator. The dynamic_cast operator will return a pointer to the object if the cast can be performed, or 0 if the cast can't be performed. The basic idea when enumerating the components on the form is to cast every component to the type you want and see if the cast fails or succeeds. If the cast fails, then the component isn't of the type you're looking for. If the cast succeeds, then you can use the pointer returned from dynamic_cast to carry out some action.

Clearing the text in all edit controls

Now let's look at how dynamic_cast can be used to clear the text in all edit controls on a form. Here's the code:

```
for (int i=0;i<ComponentCount;i++) {
        TEdit* edit;
        edit = dynamic_cast<TEdit*>(Components[i]);
        if (edit)
                edit->Text = "";
}
```

Each component in the Components array is cast to a TEdit. If the cast fails, then the loop continues without further processing. If the cast succeeds, then we use the pointer returned from dynamic_cast to set the Text property to an empty string. Let's look at another example. This code changes the font of all labels on the form to bold:

```
for (int i=0;i<ComponentCount;i++) {
        TLabel* label = dynamic_cast
```

```
                <TLabel*>(Components[i]);
        if (label)
                label->Font->Style =
                        TFontStyles() << fsBold;
}
```

This code also illustrates the proper way to set a font's style. See the article, "Dealing with sets," for more information on the correct way of adding values to a set. You can put any number of dynamic_cast calls within your enumeration loop. This code, for example, combines the two operations we've seen up to this point:

```
for (int i=0;i<ComponentCount;i++) {
        TEdit* edit = dynamic_cast
                <TEdit*>(Components[i]);
        if (edit)
                edit->Text = "";
        TLabel* label = dynamic_cast
                <TLabel*>(Components[i]);
        if (label)
                label->Font->Style =
                        TFontStyles() << fsBold;
}
```

If a TEdit is found, then its text is cleared. If a TLabel is found, its font style is changed to bold. All other components on the form are ignored.

Conclusion

Enumerating components on a form is a trivial task once you understand how the Components property works. Using dynamic_cast to cast the pointer returned from the Components array gives you full control over all components on your forms.

April 1999

Version: 1.0, 3.0, and 4.0

Dealing with sets

by Kent Reisdorph

We've covered sets in past issues of *C++Builder Developer's Journal*, but one particular aspect of sets bears repeating. That aspect deals specifically with adding elements to a set when a set property includes a write method. First, let's look at the syntax for adding elements to a set in C++Builder:

```
// Declare a set of characters from A to Z.
Set<char, 'A', 'Z'> mySet;
// Add A to the set.
mySet << 'A';
// Add B and C to the set.
mySet << 'B' << 'C';
// Remove B from the set.
mySet >> 'B';
```

All of the above code is valid and works just as the comments indicate. Now let's take the case of a property that's a set. The Font class' Style property provides a perfect example. To change the font style of a label to bold, you might, logically, write code like this:

```
Label1->Font->Style << fsBold;
```

The problem is that this code doesn't have the desired effect--the label's font doesn't change to bold. To understand why this code doesn't work, you need to understand how the VCL deals with properties. Some properties use direct access. When you write to the property, the property's underlying data variable is simply assigned a value. Other properties have write methods. If a write method exists for a property, that method is called whenever a value is assigned to the property. This is what causes a form to move when its Top property is assigned a value, an edit control to display a value when its Text property is modified, or a label's font style to change when its Font->Style property is written to.

The problem when writing to a set property is that the set insertion (<<) and extraction (>>) operators don't result in an assignment. Because there's no direct assignment, the property's write method is never called. Therefore, the property's value never changes.

Modifying a set property

The key when dealing with set properties, then, is to always make sure an assignment is made when modifying the property. There are two ways to accomplish this. One is to simply insert into the set and immediately assign:

```
Label1->Font->Style =
        Label1->Font->Style << fsBold;
```

Right-to-left code generation ensures that the fsBold element is added to the set and then the result is assigned to the Style property. This allows the write method for the Style property to be called and the label's style to be properly modified. Another way to perform an assignment to a set is to create a temporary set, add elements to the set, and then assign the temporary set to the property. For example:

```
TFontStyles style;
style << fsBold;
Label1->Font->Style = style;
```

This code can be condensed as follows:

```
Label1->Font->Style = TFontStyles() <<
        fsBold;
```

In both of these examples, a direct assignment is involved and the Style property is properly updated as a result. It's important to understand that the two methods shown for modifying a set property aren't identical. The first method takes an existing set, adds a value, and assigns the result back to the set. The second method creates a new empty set, adds one or more values, and assigns the new set to the property. The second method can easily be modified to emulate the first as follows:

```
TFontStyles style = Label1->Font->Style;
Label1->Font->Style = style << fsBold;
```

Regardless of which method you use, be sure that you always make an assignment when dealing with properties that have a set as the underlying data type.

April 1999

Version: 1.0, 3.0, and 4.0

Transferring form data

by Kent Reisdorph

Any serious Windows application will have one or more forms. Most forms are used to retrieve information from the user (aside from simple forms like an About form). Your program must have some way of extracting information from the form and, in some cases, sending information to the form. This article will discuss various ways of transferring information from the main form to a secondary form, and vice versa.

Simple forms: one-way data

In the case of a simple form, you don't really need to do much work to transfer data. Let's use a password form as an example. Invoking this type of form from the main form and transferring data might look something like this:

```
PWForm->ShowModal();
if (PWForm->PWEdit->Text == "bubba")
        // password failed
else
        // password succeeded
```

Here, we read the value of the password edit component directly through the password form's pointer. This is a one-way transfer of data because we're only reading the data on the form and aren't making any attempt to set data on the form. This situation is only slightly more complicated when you take control of creating the form yourself (in cases where the form isn't auto-created at application startup). Here's how the code might look:

```
TPWForm* form = new TPWForm(this);
form->ShowModal();
if (form->PWEdit->Text == "bubba")
        // password failed
else
        // password succeeded
delete form;
```

The only item of significance here is that you must remember to extract any data from the form before you delete the form's pointer. Once the form's pointer has been deleted, any attempts to access the pointer will result in an access violation or, in extreme cases, a complete system crash. By the way, I never let the VCL auto-create my forms in real-world applications. I always allocate the memory for the form just before I show the form and free that memory as soon as I'm done with the form.

Simple forms: two-way data

In the previous example, we only transferred data one way--from the form to the main form. In many circumstances, you'll need to assign values to controls on a form before the form is shown, and retrieve the values for those controls when the form is closed.

Let's say, for example, that you had an application with an Options form. The Options form, naturally, would allow the user to specify options that control how the application behaves. For the sake of argument, let's say you had an application that dials a remote machine in order to transfer some data via a modem. The user options for this type of program might include the phone number to dial, the name of the TAPI device used to make the connection, and the number of times to retry dialing if the number is busy. These settings would be stored in the Registry, in an INI file, or in a user-defined configuration file. The options would be read from the storage medium to variables in the main form's class at application startup. When the application closes, the options would be saved to persistent storage for retrieval the next time the application runs. Transferring the data to and from the Options form will look something like this:

```
TOptionsForm* form = new
        TOptionsForm(this);
form->PhoneNumber->Text = PhoneNumber;
form->TapiDevice->Text = TapiDevice;
form->DialRetries->Text = DialRetries;
if (form->ShowModal() == mrOk) {
        PhoneNumber = form->PhoneNumber->Text;
        TapiDevice = form->TapiDevice->Text;
        DialRetries =
                form->DialRetries->Text.ToInt();
}
delete form;
```

Here, we're only dealing with three controls on the form, and yet the code already looks

messy. If you had a large number of components on a form, the code would be even more unwieldy. Plus, we aren't showing the code required to save and load the data from persistent storage. That's a lot of code in the main unit of your application that probably doesn't need to be there at all. Now, let's look at an alternative to this method of data transfer.

Encapsulation

Before going on to describing a better method of data transfer, a few words about encapsulation are warranted. Encapsulation is one of those Object Oriented Programming (OOP) terms that we're all familiar with. *Encapsulation* basically means using an object to perform a specific task whenever possible. The object takes the code required to carry out that task and raises it to a level that's much easier to deal with.

In the context of this article, encapsulation means that you should, whenever possible, write your forms so that they're completely self-contained and don't need to have interaction with the main form at all. This is the ideal way to deal with forms in your applications. There are many types of forms, however, where this approach isn't practical because data needs to be transferred between the main form and the form.

Data transfer the OOP way

Let's go back to our previous example of an application that dials a remote machine. How can we use encapsulation to make the code cleaner? One way is to create a class that holds the options for the application. The class could have data members for each of the application's options. Further, the class could contain methods to save the options to the Registry or to a file and to read those options back again when needed. An instance of the class can be created in the main form so that the main form has access to the options. A pointer to the class can be passed to the Options form so that it also has access to the options. Setting up this type of arrangement requires several steps. We'll examine each of these steps in the following sections.

Create a transfer class

The first step is to create a class that will serve as the transfer mechanism for your Options form. First, let's look at the declaration for the class:

```
class TAppOptions {
        public:
```

```
                String PhoneNumber;
                String TapiDevice;
                int DialRetries;
                TAppOptions();
                LoadFromRegistry();
                SaveToRegistry();
};
```

As you can see, we've created data members for each option on the Options form. We've also added methods to load from the Registry and save to the Registry. Note that this class violates OOP principles by making all data members public, but for simple examples, this is an acceptable practice. An instance of this class will be created in the main form of the application. Typically, this would be done at application startup in either the main form's constructor or in its OnCreate event handler. At creation time, the previous options could be read from persistent storage. For example:

```
void __fastcall
TMainForm::FormCreate(TObject *Sender)
{
        Options = new TAppOptions;
        Options->LoadFromRegistry();
}
```

The application's options are now loaded into the class represented by the Options variable and are available for the main form's use. The Options variable is declared as a member of the main form's class. Naturally, you'll want to destroy the TAppOptions instance before the application terminates. Either the main form's destructor or the OnDestroy event handler are logical places for this code. Before you destroy the object, however, you'll want to save the current options to persistent storage. Here's how the code would look when placed in an OnDestroy event handler:

```
void __fastcall
TMainForm::FormDestroy(TObject *Sender)
{
        Options->SaveToRegistry();
        delete Options;
}
```

Using this technique, the application's options are always loaded at application startup and saved again at application termination. Alternatively, you could place the code to load the options from persistent storage in the class constructor and the code to save the options in the class destructor.

Modify the Option form's class

In order to pass an instance of the TAppOptions class to your Options form, you'll need to modify the Options form's constructor. Simply add a pointer to the TAppOptions class to your constructor. Your Options form class must also contain a TAppOptions pointer so you have access to the instance throughout the class. The modified TOptionsForm class looks like this (the form's component declarations have been removed to save space):

```
class TOptionsForm : public TForm
{
__published:    // IDE-managed Components
        // component declarations here
private:        // User declarations
        TAppOptions* Options;
public:         // User declarations
        __fastcall TOptionsForm(
                TComponent* Owner, TAppOptions*
                        options);
};
```

Notice that we've added a TAppOptions pointer to the private section of the class. Notice, also, that the declaration for the constructor has been expanded to allow us to pass a TAppOptions pointer from the main form. Let's skip ahead just a bit and explain how the Options form will be used in the main form. Using the modified TOptionsForm class requires that you create an instance of the Options form at runtime. Since the Options form now has a specialized constructor, you can't rely on the VCL's auto-creation feature for forms. Here's how the code in the main form would look:

```
void __fastcall
TMainForm::Options1Click(TObject *Sender)
{
        TOptionsForm* form =
                new TOptionsForm(this, Options);
        form->ShowModal();
        delete form;
```

```
}
```

This code assumes that the Options variable had already been instantiated as discussed in the previous section.

Writing the Options form constructor

Next, we must write the body of the TOptionsForm constructor. First, we assign the TAppOptions pointer passed in the constructor to our internal Options variable. We save the pointer because we need access to it later when the form is closed. Then, we assign values contained in the Options class to the various controls on the Options form. Here's how the constructor looks:

```
__fastcall TOptionsForm::TOptionsForm(
        TComponent* Owner, TAppOptions* options)
        : TForm(Owner)
{
        Options = options;
        PhoneNumberEdit->Text =
                Options->PhoneNumber;
        TapiDeviceEdit->Text =
                Options->TapiDevice;
        DialRetriesEdit->Text =
                Options->DialRetries;
}
```

As you can see, we simply take each data member in the Options class and assign its value to a corresponding component on the Options form. This part of the operation is fairly simple.

Saving the new options

Now, we need to take some action when the form is closed. A form like an Options form can be closed in one of two ways--with the OK button or with the Cancel button (the form's close box is the same as clicking the Cancel button). Naturally, we need to account for both possibilities. If the form is closed with the Cancel button, then no further action needs to be taken. The user can change any or all fields on the Options form, but if the Cancel button is pressed, all those changes are effectively abandoned. By simply not acting, we already have accounted for the user closing the form with the Cancel button or with the form's close box.

If the form is closed with the OK button, on the other hand, we need to act to ensure that the new options are implemented. We simply provide an OnClick event handler for the OK button and copy data from the form's controls to the Options class instance. This is

essentially the opposite of the code we wrote for the form's constructor. For example:

```
void __fastcall
TOptionsForm::OKBtnClick(TObject *Sender)
{
        Options->PhoneNumber =
                PhoneNumberEdit->Text;
        Options->TapiDevice =
                TapiDeviceEdit->Text;
        Options->DialRetries =
                DialRetriesEdit->Text.ToIntDef(0);
        Close();
}
```

Since our local Options variable is simply a pointer to the instance created in the main class, this code updates the main class' instance directly. When the Options form closes, the main form's Options variable already contains the new options. The main form doesn't have to change at all to account for the new options. This is where encapsulating the application's options in the Options form has great benefit. The Options form does all the work of updating the options and the main form doesn't have to do any extra processing. If there's a drawback to this mechanism, it's that you need to create a transfer class for each of your application's forms. Still, this method is preferred over those discussed earlier.

Example

Transferring data to and from your forms is something that's required in nearly every non-trivial C++Builder application. Our example application for this article transfers data to and from an options form using a transfer class. Listing A contains the header for the application's main form. Listing B contains the source unit for the main form. Listing C shows the header for the application's Options form and Listing D shows the source unit for the Options form. We've placed the entire implementation of the TAppOptions class in the header for the Options form to save space. Normally, however, you'd place this class in its own unit.

Listing A: MAINU.H

```
//---------------------------------------------
#ifndef MainUH
#define MainUH
//---------------------------------------------
#include <Classes.hpp>
```

```
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#include "OptionsU.h"
//---------------------------------------------
class TMainForm : public TForm
{
__published:    // IDE-managed Components
        TMainMenu *MainMenu1;
        TMenuItem *File1;
        TMenuItem *Exit1;
        TMenuItem *Tools1;
        TMenuItem *Options1;
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
        void __fastcall Exit1Click(TObject *Sender);
        void __fastcall
                Options1Click(TObject *Sender);
private:        // User declarations
        TAppOptions* Options;
public:         // User declarations
        __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------
#endif
```

Listing B: MAINU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "MainU.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------
__fastcall
TMainForm::TMainForm(TComponent* Owner)
```

```
        : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall
TMainForm::FormCreate(TObject *Sender)
{
        // Create an instance of TAppOptions and
        // load from the registry.
        Options = new TAppOptions;
        Options->LoadFromRegistry();
}
//---------------------------------------------
void __fastcall
TMainForm::FormDestroy(TObject *Sender)
{
        // Save the options to the registry and
        // then delete the TAppOptions object.
        Options->SaveToRegistry();
        delete Options;
}
//---------------------------------------------
void __fastcall
TMainForm::Exit1Click(TObject *Sender)
{
        Close();
}
//---------------------------------------------
void __fastcall
TMainForm::Options1Click(TObject *Sender)
{
        // Create an instance of the TOptionsForm
        // class, passing the Options variable as
        // a paramter. TOptionsForm does the rest.
        TOptionsForm* form =
                new TOptionsForm(this, Options);
        form->ShowModal();
        delete form;
}
//---------------------------------------------
```

Listing C: OPTIONSU.H

```
//---------------------------------------------
#ifndef OptionsUH
#define OptionsUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Registry.hpp>
//---------------------------------------------
// The registry key where we will save the data.
const String RegKey =
        "Software\\ZDJournals\\DialogTransferEx";

// The TAppOptions class for transferring data.
class TAppOptions {
        public:
                String PhoneNumber;
                String TapiDevice;
                int DialRetries;
                TAppOptions() {
                        PhoneNumber = "";
                        TapiDevice = "";
                        DialRetries = 0;
                }
                LoadFromRegistry() {
                        TRegistry* reg = new TRegistry;
                        reg->OpenKey(RegKey, true);
                        if (reg->ValueExists
                                        ("PhoneNumber")) {
                                PhoneNumber =
                                        reg->ReadString
                                        ("PhoneNumber");
                                TapiDevice =
                                        reg->ReadString
                                        ("TapiDevice");
                                DialRetries =
                                        reg->ReadInteger
                                        ("DialRetries");
                        }
                        delete reg;
                }
                SaveToRegistry() {
```

```
                        TRegistry* reg = new TRegistry;
                        reg->OpenKey(RegKey, true);
                        reg->WriteString(
                                "PhoneNumber", PhoneNumber);
                        reg->WriteString(
                                "TapiDevice", TapiDevice);
                        reg->WriteInteger(
                                "DialRetries", DialRetries);
                        delete reg;
                }
};

class TOptionsForm : public TForm
{
__published:    // IDE-managed Components
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TEdit *PhoneNumberEdit;
        TEdit *TapiDeviceEdit;
        TEdit *DialRetriesEdit;
        TButton *OKBtn;
        TButton *CancelBtn;
        void __fastcall
                CancelBtnClick(TObject *Sender);
        void __fastcall OKBtnClick(TObject *Sender);
private:        // User declarations
        TAppOptions* Options;
public:         // User declarations
        __fastcall TOptionsForm(
                TComponent* Owner, TAppOptions* options);
};
//---------------------------------------------
extern PACKAGE TOptionsForm *OptionsForm;
//---------------------------------------------
#endif
```

Listing D: OPTIONSU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#include "OptionsU.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TOptionsForm *OptionsForm;
//---------------------------------------------
__fastcall TOptionsForm::TOptionsForm(
        TComponent* Owner, TAppOptions* options)
        : TForm(Owner)
{
        // Save the incoming TAppOptions pointer.
        Options = options;
        // Transfer data from the Option class
        // to the individual controls on the form.
        PhoneNumberEdit->Text =
                Options->PhoneNumber;
        TapiDeviceEdit->Text =
                Options->TapiDevice;
        DialRetriesEdit->Text =
                Options->DialRetries;
}
//---------------------------------------------
void __fastcall
TOptionsForm::CancelBtnClick(TObject *Sender)
{
        Close();
}
//---------------------------------------------
void __fastcall
TOptionsForm::OKBtnClick(TObject *Sender)
{
        // Transfer data from the individual controls
        // to the Options class.
        Options->PhoneNumber =
                PhoneNumberEdit->Text;
        Options->TapiDevice =
                TapiDeviceEdit->Text;
        Options->DialRetries =
                DialRetriesEdit->Text.ToIntDef(0);
        Close();
}
//---------------------------------------------
```

March 1999

Version: 1.0, 3.0

Spawning external applications

by Kent Reisdorph

The Inprise newsgroups are a great source of information for C++Builder programmers. It seems that barely a day goes by without someone asking the question, "How do I execute an external application from my program?" This article will explain how to spawn an external application from within your program using the ShellExecute, WinExec, and CreateProcess Windows API functions. We'll also show you how to make your application wait until a spawned external program has terminated before continuing on.

Executing an external application

There are many reasons you may want to execute an application from within your program. You may, for example, want to display a text file to your users. Or you may have an application that your program must call in order to perform some specialized processing. Sometimes this type of program is a legacy 16-bit Windows or DOS program for which you don't have the source code. Running the application from your program is the only way to gain the external application's functionality. This situation is common in applications being ported from DOS to Windows. You essentially have three choices of Win32 API functions when executing an external application:

- The WinExec function
- The ShellExecute function
- The CreateProcess function

The following sections will explain each of these functions and how they are used. The functions are presented from easiest to use to most complex.

Note: We aren't including the spawn and exec family of functions from the C/C++ runtime library in this article. Those functions, while useable in a C++Builder application, weren't designed for the Win32 environment.

WinExec

The WinExec function provides the easiest way of spawning an external process. It's the simplest in that WinExec takes only two parameters. Here's the function declaration for WinExec (found in WINBASE.H):

```
UINT WinExec(
  LPCSTR lpCmdLine, UINT uCmdShow);
```

The first parameter, lpCmdLine, is the command line. The command line can include an executable filename, or the filename and additional parameters to pass to the application. The uCmdShow parameter is the window display command. uCmdShow is usually one of the values shown in Table A.

Table A: Common show commands

Command	Description
SW_HIDE	The application is run invisibly.

SW_MAXIMIZE	The application is run maximized.
SW_MINIMIZE	The application is run minimized.
SW_NORMAL	The application is run in the default size and position.

While the show commands in Table A are the most commonly used values for the uCmdShow parameter, they aren't the only possible values. For a complete list of possible values for this parameter, see the ShellExecute topic in the Win32 API Help.

Note: You should only run applications hidden if you know they don't require any user input. If a hidden application requires user input, it--and possibly your application--will hang waiting for input that will never come.

If the call to WinExec succeeds, the return value is an integer greater than 31. If the return value is less than 32, an error occurred and the return value represents an error code. Reasons for an error include the EXE wasn't found, the path wasn't found, the system is out of memory, or a bad executable format was detected. The bad format error will occur if the application being executed isn't a valid Win32 application or is corrupted.

The following example illustrates executing Windows Notepad using WinExec:

```
WinExec("notepad.exe", SW_NORMAL);
```

We simply pass the name of the application to execute and the show command value. If the filename doesn't include a path, Windows will attempt to locate the file (see "How Windows locates files"). Here's another example that executes Notepad and loads a file called TEST.TXT:

```
WinExec("notepad.exe test.txt", SW_NORMAL);
```

As you can see, using WinExec in an application is trivial. Although WinExec is easy to use, it suffers from some major drawbacks. First, the application can't exercise any control over the spawned application. Most notably, your application can't easily determine when the spawned application has terminated. WinExec starts the application specified in the lpCmdLine parameter and returns immediately. At that point, your application is on its way again, with no consideration given to the state of the spawned application.

Another drawback to WinExec is that it's on the Win32 "endangered species" list. Specifically, the Win32 API Help has this to say about WinExec: "Win32-based applications should use the CreateProcess function rather than this function. The WinExec function exists in Win32 to provide compatibility with earlier versions of Windows." This warning essentially says that WinExec may not be available in future versions of Windows. While it's unlikely that WinExec will be removed from Windows any time soon, you should be aware that applications that use WinExec might have to be modified and recompiled if this function is removed from Windows at some point in the future.

Finally, WinExec can't be used to execute a 16-bit program. If your application must execute a 16-bit program (either DOS or Windows) you need to use CreateProcess rather than WinExec.

ShellExecute

The ShellExecute function can be used to spawn an executable file, but it has a specific feature that makes it much more powerful than WinExec. The key to this feature is the function name itself. ShellExecute can be used to display any document for which the Windows shell has registered a file extension. Windows keeps a list of known file extensions and the application associated with that extension. For example, for a default Windows installation the TXT extension is associated with Windows Notepad, the BMP extension is associated with the Paint program, the WAV extension is associated with the Sound Recorder program, and so on. Other applications you install may also register their file extensions. Examples include Microsoft Word (DOC), Microsoft Excel (XLW), and C++Builder (CPP, BPR, RES, and others).

The Windows shell uses these associations in many ways. One way is in Windows Explorer. If you double-click on a filename in Explorer, Windows looks up the filename's extension in its list of file associations. If it finds an association for that file, the associated application is executed and the document that was double-clicked is loaded into that application. If, for example, you double-click on a file with a TXT extension, Windows will start Notepad and will load the text file (assuming the TXT extension hasn't been changed to be associated with a program other than Notepad).

Another way Windows uses file associations is when you call the ShellExecute function. If the filename passed to ShellExecute is a document file, Windows searches the association database just as it does when you double-click a file in Explorer. We'll talk more about that later, but first let's take a look at the ShellExecute declaration in SHELLAPI.H:

```
HINSTANCE ShellExecute(HWND hwnd,
  LPCSTR lpOperation, LPCSTR lpFile,
  LPCSTR lpParameters, LPCSTR lpDirectory,
  INT nShowCmd);
```

Obviously ShellExecute is a bit more complex than WinExec. In some cases, you'll only use a couple of the parameters and the rest can be 0. In other cases, you'll use all of the parameters. First, we'll describe the individual parameters and then we'll look at some examples. The hwnd parameter is the handle of the window that will act as the parent for the spawned application. The lpOperation parameter is used to specify the action that the spawned application should perform on the document when it's opened. Operations include open, print, and explore. If the operation is open, then Windows executes the associated application and opens the document specified in the lpFile parameter. If the operation is print, then Windows will print the document. This assumes, of course, that the application associated with the given filename is capable of printing. If operation is explore, then Windows will open the folder specified in lpFile in a shell browser window.

As you've probably surmised, the lpFile parameter is used to specify the document to open or print, an executable file to run, or a folder name. The lpParameters parameter is used to specify any additional command line arguments you wish to pass to the application. If no command line arguments are needed, set this parameter to 0. The lpDirectory parameter is used to specify the default directory for the application. If this parameter is 0, the current directory is used. Finally, the nShowCmd parameter is used to specify the show command. The possible values for this parameter are listed in
Table A.

If successful, the return value of ShellExecute will be greater than 31 and will be the instance handle of the application that was executed. If the return value is 31 or less, the return value is an error code. Possible errors include those listed earlier for the WinExec function, plus a few that are specific to ShellExecute (such as the *no such file association* error code). We won't list all of the possible error codes here, but you can find them listed under the ShellExecute topic in the Win32 API Help.

After that relatively dry explanation of ShellExecute's parameters, a few examples are in order. Let's take first the situation where you want to use ShellExecute to simply run an external application. In that case, the code might look like this:

```
ShellExecute(Handle, 0, "notepad.exe", 0, 0, SW_NORMAL);
```

Because we're simply using ShellExecute to spawn an external application, we can supply 0's for most of the parameters. Now, let's look at an example that opens a particular document file. Look at these two lines of code:

```
ShellExecute(Handle, "open", "test.txt", 0, 0, SW_NORMAL);
ShellExecute(Handle, 0, "notepad.exe", "test.txt", 0, SW_NORMAL);
```

In the first example, we simply pass a command of open and the filename to open. Windows uses file association to open the application associated with the TXT extension (Notepad in most cases). In the second example, we pass the name of the executable in the lpFile parameter and the name of the file to open in the lpParameters parameter. Use the latter method if you prefer not to rely on Windows file associations to run an application. These two lines result in exactly the same behavior *if* Windows Notepad is associated with the TXT extension (the default Windows setup). ShellExecute can be used as a poor man's text file printer. Using ShellExecute you can execute Notepad invisibly and print the contents of a text file. Here's an example:

```
ShellExecute(Handle, "print", "test.txt", 0, 0, SW_HIDE);
```

Note that here we are using a command of print and that the nShowCmd parameter is SW_HIDE. ShellExecute can be used in a wide variety of circumstances. It can be used to display a Web page in a Web browser, to invoke the user's default email client, or to execute batch files (BAT). The following examples show how you might use ShellExecute in these situations:

```
// Execute a batch file.
ShellExecute(Handle, 0, "go.bat", 0, 0, SW_NORMAL);

// Display a Web page.
ShellExecute(Handle, 0, "http://www.turbopower.com", 0, 0, SW_NORMAL);

// Begin an e-mail message.
ShellExecute(Handle, 0, "mailto:info@turbopower.com", 0, 0, SW_NORMAL);
```

As you can see, ShellExecute is a very versatile function. It does, however, suffer from one of the limitations that we mentioned in the section on WinExec. Namely, it's incapable of executing a 16-bit program. For that task you'll have to use CreateProcess. ShellExecute has a companion function called ShellExecuteEx. ShellExecuteEx provides everything that ShellExecute does, plus the ability to get a handle to the spawned application's process. This is important if you want your application to pause execution until the spawned application has finished. See the sidebar "Waiting on a spawned process," for a bit more discussion on this process.

CreateProcess

CreateProcess is the preferred way of executing external applications in a Win32 program. The biggest problem with CreateProcess is that it's a bit cumbersome to use. Looking at the Win32 API Help for

CreateProcess can easily leave you with the impression that it's just too difficult to use. The truth is that the whole of CreateProcess is horribly complex. The good news, however, is that you don't have to understand CreateProcess in its entirety, but only a small subset of it. Once you know how to use CreateProcess to execute an application, however, you'll find that it's not as difficult to implement as you had feared. We won't attempt to explain all of the functionality of CreateProcess, but rather will show you only as much as you need to know to run an external application. First, let's look at the declaration for CreateProcess (also in WINBASE.H):

```
BOOL CreateProcess(
  LPCSTR lpApplicationName,
  LPSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES
  lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCSTR lpCurrentDirectory,
  LPSTARTUPINFOA lpStartupInfo,
  LPPROCESS_INFORMATION
  lpProcessInformation );
```

Granted, this declaration looks a bit intimidating, but many of the parameters can be ignored. This is particularly true of the parameters dealing with security attributes. A basic call to CreateProcess looks like this:

```
STARTUPINFO si;
PROCESS_INFORMATION pi;
memset(&pi, 0, sizeof(pi));
memset(&si, 0, sizeof(si));
si.cb = sizeof(si);
int res = CreateProcess("c:\\windows\\notepad.exe", 0, 0, 0, 0, 0, 0, 0, &si, &pi);
```

First, we declare two structures and zero them out to insure they don't contain random data. Next, we set the cb member to the size of the STARTUPINFO structure. Finally, we call CreateProcess to spawn the application. As you can see, the only required parameters are the lpApplicationName, lpStartupInfo, and lpProcessInformation parameters. If CreateProcess succeeds the return value is non-zero. If CreateProcess returns 0, you should call GetLastError to get an error code indicating what went wrong.

Note: If the file to be executed isn't in the current directory, you must fully qualify the path and filename of the application. This is only true of filenames passed in the lpApplicationName parameter, not for filenames passed in the lpCommandLine parameter.

There's a quirk of CreateProcess that you should be aware of if you plan on running a 16-bit application from your program. Take this code for example:

```
CreateProcess("MyDosApp.exe", 0, 0, 0, 0, 0, 0, 0, &si, &pi);
```

Note that we passed the filename of the file to execute in the first parameter, lpApplicationName. Under Windows 95 and Windows 98 this code will fail if the filename passed in lpApplicationName is a 16-bit DOS application (it will work on Windows NT, however). In order to execute a 16-bit DOS app in Win95 and 98 you must pass the filename in the lpCommandLine parameter rather than in the lpApplicationName parameter. The following code shows the correct way of calling CreateProcess to spawn a 16-bit application:

```
CreateProcess(0, "MyDosApp.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
```

This code will work properly in all versions of Windows (32-bit) so you should make a habit of passing the filename to execute in the second parameter in your call to CreateProcess rather than in the first parameter. The STARTUPINFO structure passed to CreateProcess can contain additional startup information if desired. In the previous examples, we wanted to run the spawned application normally so we didn't provide any startup information. If you wanted to run the application hidden, you'd need to do this:

```
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = SW_HIDE;
int res = CreateProcess(0, "myapp.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
```

See the STARTUPINFO topic in the Win32 API Help for full details on control of the startup process.

Conclusion

Spawning external applications in Windows can be a frustrating exercise the first time you attempt it. Knowing the pitfalls, as explained in this article, may save you hours of time. With WinExec, ShellExecute, ShellExecuteEx, and CreateProcess, you have everything you need to effectively spawn external applications from your programs.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

TURBOPOWER®

Company of the Year

Delphi Informant Magazine Readers Choice Awards

Happy Holidays from TurboPower Software Company

The American Liberty Partnership

from the entire TurboPower staff...

Happy Holidays!

In celebration of the holidays our offices will be closed from December 21, 2002 to January 2, 2003 in order to give our staff the opportunity to spend some quality time with family and friends. All orders and correspondence received during this period will be held until our return in the New Year.

Borland® TECHNOLOGY PARTNER

2002 READERS CHOICE Delphi INFORMANT MAGAZINE COMPANY OF THE YEAR

- Visual PlanIt—Integrated PIM controls help your customers' business run smoothly
- SysTools 4—Build your best ideas fast with routines that are designed to fit

Recent headlines

ProActivate	ProActivate 1.10 update now available for download. This update adds Delphi 7 compatibility and is recommended for all users.	28 Oct
ShellShock	ShellShock 1.02 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of ShellShock. This update is recommended for all users.	18 Oct
FlashFiler	FlashFiler 2.12 update now available for download. This update corrects problems found in earlier versions of FlashFiler. This update is recommended for all users.	17 Oct
XMLPartner	XMLPartner 2.57 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of XMLPartner. This update is recommended for all users.	17 Oct
Abbrevia	Abbrevia 3.04 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Abbrevia. This update is recommended for all users.	15 Oct
OfficePartner	OfficePartner 1.64 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of OfficePartner. This update is recommended for all users.	15 Oct
Essentials	Essentials 1.11 update now available for download. This update adds Delphi 7 compatibility and is recommended for all users.	7 Oct
Internet Professional	Internet Professional 1.14 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Internet Professional. This update is recommended for all users.	4 Oct

OnGuard	OnGuard 1.13 update now available for download. This update adds Delphi 7 compatibility and is recommended for all users.	4 Oct
LockBox	LockBox 2.07 update now available for download. This update adds Delphi 7 compatibility and is recommended for all users.	4 Oct
SysTools	SysTools 4.02 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of SysTools. This update is recommended for all users.	26 Sep
Orpheus	Orpheus 4.05 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Orpheus. This update is recommended for all users.	20 Sep
Async Professional ActiveX	Async Professional ActiveX 1.12 update now available for download. This update includes enhancements and corrects problems found in earlier versions of APAX, and is recommended for all users.	19 Sep
Async Professional	Async Professional 4.05 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Async Professional. This update is recommended for all users.	17 Sep
Visual PlanIt	Visual PlanIt 1.02 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Visual PlanIt. This update is recommended for all users.	17 Sep
Sleuth QA Suite	Sleuth QA Suite 3.06 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Sleuth QA Suite. This update is recommended for all users.	13 Sep
Memory Sleuth	Memory Sleuth 3.01 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of Memory Sleuth. This update is recommended for all users.	13 Sep
FlashFiler	FlashFiler 2.11 update now available for download. This update adds Delphi 7 compatibility and corrects problems found in earlier versions of FlashFiler. This update is recommended for all users.	13 Sep
Internet Professional	Internet Professional 1.13 update now available for download. This update corrects problems found in earlier versions of Internet Professional and is recommended for all users.	9 Aug
ANNOUNCEMENT	TurboPower Software Company receives the honor of being named Borland's Technology Partner of the Year for Rapid Application Development at Borland's 13th Annual Borland Conference in Anaheim, CA.	20 May
ANNOUNCEMENT	Thanks for your support! Once again, TurboPower Software Company fares well in the *Delphi Informant* Readers Choice Awards, winning six awards including "Company of the Year."	25 Apr

More stories >>

Call us at 800/333.4160 or 719/471.3898
Secure online ordering Customer Service? E-Mail Us!

Published: Monday, March 04, 2002
Modified: Friday, December 20, 2002

March 1999

Waiting on a spawned process

by Kent Reisdorph

Sometimes you want to spawn an external process and wait until it's completed before continuing with your application. Both CreateProcess and ShellExecuteEx give you a handle to the thread Windows created to run the spawned application. You can use this thread handle, in conjunction with the WaitForSingleObject API function, to pause execution of your program until the spawned application has terminated. To do so, simply pass the thread handle of the spawned process to WaitForSingleObject and Windows does the rest. Here's how the code would look if you're using the CreateProcess function:

```
STARTUPINFO si;
PROCESS_INFORMATION pi;
memset(&pi, 0, sizeof(pi));
memset(&si, 0, sizeof(si));
si.cb = sizeof(si);
int res = CreateProcess(0, "notepad.exe",
  0, 0, 0, 0, 0, 0, &si, &pi);
if (res) {
  WaitForSingleObject(pi.hThread,
    INFINITE);
  SetFocus();
}
```

If CreateProcess is successful, the hThread member of the PROCESS_INFORMATION structure contains the thread handle for the new process. We pass that thread handle to WaitForSingleObject with a timeout value of INFINITE. The host application will now pause execution until the spawned application is closed. ShellExecuteEx can also be used to wait for a spawned process. Here's an example of displaying a text file in Windows Notepad and waiting for Notepad to close before allowing the application to continue:

```
SHELLEXECUTEINFO si;
memset(&si, 0, sizeof(si));
si.cbSize = sizeof(si);
si.hwnd = Handle;
si.lpVerb = "open";
si.lpFile = "test.txt";
si.nShow = SW_NORMAL;
si.fMask = SEE_MASK_NOCLOSEPROCESS;
bool res = ShellExecuteEx(&si);
```

```
if (res)
  WaitForSingleObject(si.hProcess,
    INFINITE);
```

Although we haven't talked about ShellExecuteEx in any detail, it's easy to see how this function works compared to ShellExecute. Rather than sending the parameters directly to the function, ShellExecuteEx requires you to fill out a structure with the required information. If ShellExecuteEx returns successfully, the hProcess member of the SHELLEXECUTEINFO structure contains the process handle of the spawned application. This member is only valid if the fMask member contains the SEE_MASK_NOCLOSEPROCESS flag. You can pass the process handle to WaitForSingleObject, as shown in the previous example.

March 1999

How Windows locates files

by Kent Reisdorph

In the article, "Spawning external applications," we passed filenames or document files to the WinExec, ShellExecute, and CreateProcess functions. These functions allow you to pass a fully qualified path and filename or just the filename itself. If just a filename is passed, Windows will attempt to locate the file. Windows searches for the file in the following order:

1. The directory where the application resides.
2. The current directory.
3. The Windows system directory (SYSTEM for Windows 95 and 98, and SYSTEM32 for Windows NT).
4. The Windows 16-bit system directory, usually called SYSTEM (NT only).
5. The Windows root directory.
6. The directories on the system path (listed in the PATH environment variable).

If the application being spawned doesn't reside in any of the directories listed above, you must fully qualify the path and filename to the application.

March 1999

Version: 1.0 3.0

Using RAS, part 2

by Kent Reisdorph

In Using RAS, part 1, we gave you an introduction to Microsoft's Remote Access Service (RAS) API. We showed you how to make a connection, detect an existing connection, and how to enumerate active connections. Now we'll continue our discussion on RAS by covering phonebooks in more detail, including how to add, edit, and delete phonebook entries. We'll also introduce you to some of the dialogs that RAS provides for use in your applications.

Got a phonebook handy?

Windows stores dial-up networking information in phonebooks. A phonebook contains one or more phonebook entries. Each phonebook entry describes a connection to a remote machine. This connection information includes the modem used to dial, the phone number to dial, server types, allowed protocols, scripting, and so on. A user may have just one phonebook entry or may have several entries. For example, a user may regularly connect to a server at his or her office (both local and 800 numbers), to an ISP, or to a number of online services, such as AOL, CompuServe, or MSN. Each of these connections requires a separate phonebook entry. Phonebooks are handled differently on Windows NT and Windows 95/98. Under Windows NT the phonebook is a file on disk. Phonebook files have a .PBK extension. The phonebook file is a simple text file that describes each entry's properties. The phonebook file is laid out in standard Windows configuration (INI) file format. Multiple phonebooks are possible in Windows NT although, in reality, more than one phonebook is a rarity for most users. Windows NT has a default phonebook so a specific phonebook isn't required.

In Windows 95 and 98 only one phonebook exists. The phonebook isn't stored on disk, but rather is stored in the registry.

When writing applications that use RAS you must account for the fact that the user may have multiple phonebook entries. If only one entry exists in the default phonebook, you can use that entry to make a connection. If, however, more than one phonebook entry exists, you should provide a list of entries for your user to select from. Further, if you know your application will be deployed on NT machines, then you should allow the user to specify a phonebook file.

Dealing with phonebooks

The RAS API provides functions that allow you to create, edit, and delete phonebook entries. Here again,

differences exist between Windows NT and Windows 95/98. The following sections describe each of these operations in detail.

Creating a phonebook entry

Creating a phonebook entry is relatively simple, particularly in Windows 95 and 98. The RasCreatePhonebookEntry function provides this service in Windows 95 and 98. In Windows NT, use the RasEntryDlg function.

Note: You can use RasCreatePhonebookEntry on Windows NT systems even though the Win32 API help suggests using RasEntryDlg (a call to RasCreatePhonebookEntry maps to RasEntryDlg on Windows NT anyway). Usually, it's wise to heed the recommendations in the Win32 API Help file. The problem in this case, however, is that use of the Windows NT dialog functions requires RASDLG.DLL.

If you build an application on NT that uses the NT functions, it may fail to load on a Windows 95 or 98 machine because RASDLG.DLL doesn't exist on these platforms. Unless your application is specifically targeting Windows NT, you should program for the lowest common denominator. In this case, that means using RasCreatePhonebookEntry instead of RasEntryDlg to create phonebook entries.

When you call RasCreatePhonebookEntry on a Windows 95 or Windows 98 machine, Windows invokes the Make New Connection Wizard. The wizard takes the user through the process of creating a new phonebook entry. This wizard is fairly basic and only asks for a connection name, modem, phone number, and country of origin. It doesn't take into account all of the possible phonebook entry options. Additional options can be set later by editing the phonebook entry (discussed in the next section).

When you call RasCreatePhonebookEntry or RasEntryDlg on a Windows NT machine, Windows will display the New Phonebook Entry dialog box. This dialog is a multi-page dialog and allows the user to specify all aspects of a dial-up connection. The same dialog is displayed regardless of whether you call RasCreatePhonebookEntry or RasEntryDlg.

When the user completes the new phonebook entry information and closes the wizard (Windows 95/98) or dialog (Windows NT) a new entry is created in the phonebook. Here's an example of how to use the RasCreatePhonebookEntry function:

```
RasCreatePhonebookEntry(Handle, 0);
```

As you can see, there's not much to it. The first parameter of this function is a handle to the dialog's owner. In this case, we're passing the form's Handle property for this parameter. The

second parameter is used to specify the phonebook in which to create the new entry. In the case, of Windows 95/98 this parameter is ignored and should be 0. In Windows NT this parameter can be the path and filename of a phonebook file. If this parameter is 0 under NT, then the default system phonebook file is used. The following example shows how to use RasEntryDlg in an application targeted for Windows NT:

```
RASENTRYDLG r;
memset(&r, 0, sizeof(RASENTRYDLG));
r.dwSize = sizeof(RASENTRYDLG);
r.hwndOwner = Handle;
r.dwFlags = RASEDFLAG_NewEntry;
RasEntryDlg(0, 0, &r);
```

First, we create an instance of the RASENTRYDLG structure, zero out the memory, and set the dwSize member to the size of the structure. Next, we set the hwndOwner member to the form's Handle property. After that, we set the dwFlags member to RASEDFLAG_NewEntry to tell Windows we're creating a new phonebook entry. Finally, we call RasEntryDlg to invoke the New Phonebook Entry dialog. The first parameter of RasEntryDlg is used to specify the phonebook that the new entry will be added to. As with all of the RAS functions, if you pass 0 for this parameter Windows will use the default phonebook file. The second parameter is unused when creating a phonebook entry so we pass 0 for that parameter. The final parameter is a pointer to the RASENTRYDLG structure.

Editing an existing phonebook entry

Editing an existing phonebook entry is no more complicated than creating a new phonebook entry. This is true because Windows does most of the work by providing dialogs for editing phonebook entries. Under Windows 95 and 98, you edit phonebook entries by calling the RasEditPhonebookEntry function. You simply pass a handle to the owning application, the phonebook to use, and the name of the phonebook entry to edit. Here's an example:

```
RasEditPhonebookEntry(
  Handle, 0, "My Connection");
```

The second parameter is used to specify the phonebook in which the entry can be found. As we said earlier, this parameter is ignored under Windows 95 and 98 and should be set to 0. When you call RasEditPhonebookEntry on a Windows 95/98 system, Windows will display a multi-page dialog containing all of the entry's configuration options. When the user clicks the OK button the entry is updated in the registry.

The RasEditPhonebookEntry function can also be used under Windows NT, but, here, the Win32 API Help recommends that you use RasEntryDlg to edit phonebook entries on NT

applications. A call to RasEntryDlg might look like this:

```
RASENTRYDLG r;
memset(&r, 0, sizeof(RASENTRYDLG));
r.dwSize = sizeof(RASENTRYDLG);
r.hwndOwner = Handle;
r.dwFlags = 0;
RasEntryDlg(0, "MyConnection", &r);
```

This is nearly identical to the code we used when we created a phonebook entry in the previous section. The only difference is the fact that the dwFlags parameter is set to 0 and the second parameter of RasEntryDlg specifies a phonebook entry to edit. Regardless of whether you use RasEditPhonebookEntry or RasEntryDlg on an NT system, Windows will display the Edit Phonebook Entry dialog box. This dialog is functionally identical to the New Phonebook Entry dialog box discussed in the previous section. The only difference is the dialog's title. When the user clicks the OK button, the phonebook entry is updated in the phonebook file.

Deleting phonebook entries

Deleting phonebook entries is accomplished with the RasDeleteEntry function. Using this function is ridiculously simple:

```
RasDeleteEntry(0, "MyConnection");
```

Here, we simply pass 0 for the phonebook name and "MyConnection" as the phonebook entry to delete. Windows doesn't prompt you before deleting the entry so it's up to you to provide the proper warning dialog to your user before deleting an entry from the phonebook. If you look up RasDeleteEntry in the Win32 API Help, you'll find that Help says it's only available on Windows NT. This isn't accurate. RasDeleteEntry is available on Windows 98 and on Windows 95 with service release OSR2. Be aware, though, that earlier versions of Windows 95 may not have this function.

Windows NT RAS functions

If you're writing an application that will run only on Windows NT, then you have more RAS functions available to you. Only use the RAS functions specific to Windows NT if you're certain your application will only be deployed on NT systems. Applications using the NT functions may fail to load if run on a Windows 95 or Windows 98 machine. At best they'll likely fail to operate correctly. The following RAS functions are only available on Windows NT:

```
RasConnectionNotification
RasDialDlg
RasEntryDlg
RasEnumAutodialAddresses
RasEnumDevices
RasGetAutodialAddress
RasGetAutodialEnable
RasGetAutodialParam
RasGetCountryInfo
RasGetCredentials
RasGetEntryProperties
RasGetSubEntryHandle
RasGetSubEntryProperties
RasMonitorDlg
RasPhonebookDlg
RasRenameEntry
RasSetAutodialAddress
RasSetAutodialEnable
RasSetAutodialParam
RasSetCredentials
RasSetEntryProperties
RasSetSubEntryProperties
RasValidateEntryName
```

Some of these functions are available only on Windows NT because they're intended to be used when writing a RAS server. As such, we won't attempt to cover all of these functions in this article. There are a few, however, that we'll discuss briefly.

RasPhonebookDlg

The RasPhonebookDlg function can be used to display NT's main dial-up networking dialog box. From this dialog box, users can choose a phonebook entry to dial, can edit, copy, or delete entries, and can initiate dialing. The following example illustrates how to call RasPhonebookDlg:

```
RASPBDLG r;
memset(&r, 0, sizeof(RASPBDLG));
r.dwSize = sizeof(RASPBDLG);
r.hwndOwner = Handle;
RasPhonebookDlg(0, 0, &r);
```

The first two parameters of RasPhonebookDlg are identical to those found in RasEntryDlg.

The first parameter is used to specify the phonebook file, and the second is used to specify the phonebook entry that should be initially displayed on the dialog. The final parameter is a pointer to a RASPBDLG structure.

RasDialDlg

Another NT-only function of note is the RasDialDlg function. This function will dial the specified phonebook entry and will display a simple dialog showing the connection status. A call to RasDialDlg looks like this:

```
RASDIALDLG r;
memset(&r, 0, sizeof(RASPBDLG));
r.dwSize = sizeof(RASDIALDLG);
r.hwndOwner = Handle;
RasDialDlg(0, "Compuserve", 0, &r);
```

The first two parameters are identical to those of RasPhonebookDlg. The third parameter can be used to specify a phone number to dial if you wish to override the phonebook entry's phone number. The RASDIALDLG structure is used to specify additional options for the RasDialDlg function. These additional options include the X and Y position of the dialog or a phonebook subentry to dial. On return from RasDialDlg, the dwError member can be examined to determine if an error occurred during dialing.

RasMonitorDlg

The RasMonitorDlg function is used to display a dialog showing the status of a connection. This is the same dialog box you see if you right-click the Dial-Up Networking Monitor application in NT's system tray and from the shortcut menu, choose Open Dial-Up Monitor. A call to RasMonitorDlg looks like this:

```
RASMONITORDLG r;
memset(&r, 0, sizeof(RASMONITORDLG));
r.dwSize = sizeof(RASMONITORDLG);
r.hwndOwner = Handle;
r.dwStartPage = 0;
RasMonitorDlg(0, &r);
```

The first parameter is the device the status dialog will display initially. If this parameter is 0, the status of the first device is displayed. The dwStartPage parameter of the RASMONITORDLG structure is used to specify the page of the Dial-Up Monitor dialog that's initially displayed.

Conclusion

Listing A shows the header for our test application's main unit. Listing B shows the unit itself. The test application contains a combo box that shows all of the current phonebook entries on the system. It also contains buttons that allow you to create, edit, and delete phonebook entries. You can download the code from our Web site at **ftp://ftp.zdjournals.com/cpb**; click on the Source Code hyperlink. Understanding the RAS API is important if you're writing applications that use dial-up networking. This is true of most Internet-based applications. Fortunately RAS isn't that difficult to use, once you have a basic understanding of how it works.

Listing A: RASPBKU.H

```
//---------------------------------------------
#ifndef RasPBkUH
#define RasPBkUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
        TButton *EditBtn;
        TButton *CreateBtn;
        TComboBox *EntriesCB;
        TButton *DeleteBtn;
        TLabel *Label1;
        void __fastcall EditBtnClick(TObject *Sender);
        void __fastcall
                CreateBtnClick(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall
                DeleteBtnClick(TObject *Sender);

private:        // User declarations
void EnumEntries();
public:         // User declarations
__fastcall TForm1(TComponent* Owner);
```

```
};
//---------------------------------------------
#if (__BORLANDC__ < 0x530)
#define PACKAGE
#endif
extern PACKAGE TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: RASPBKU.CPP

```


//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "RasPBkU.h"
#include <ras.h>
#include <raserror.h>
//---------------------------------------------
#if (__BORLANDC__ >= 0x530)
#pragma package(smart_init)
#endif
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//---------------------------------------------
void __fastcall
TForm1::EditBtnClick(TObject *Sender)
{
// Edit a phone book entry. The entry to
        // edit is the currently selected item
        // in the entries combo box.
        RasEditPhonebookEntry(
                Handle, 0, EntriesCB->Text.c_str());
}
//---------------------------------------------
void __fastcall
TForm1::CreateBtnClick(TObject *Sender)
```

```
{
        // Create a new phone book entry.
        RasCreatePhonebookEntry(Handle, 0);
        // Reset the contents of the entries combo.
        EnumEntries();
}
//---------------------------------------------
void __fastcall
TForm1::FormCreate(TObject *Sender)
{
        // Populate the combo box with a list of
        // current phonebook entries.
        EnumEntries();
}
//---------------------------------------------
void TForm1::EnumEntries()
{
        // Enumerate the entries. This is the same
        // code used in the part 1 of the RAS
        // articles so it is not explained here.
        RASENTRYNAME* entries = new RASENTRYNAME[1];
         entries[0].dwSize = sizeof(RASENTRYNAME);
        DWORD numEntries;
        DWORD size = entries[0].dwSize;
        DWORD res = RasEnumEntries(
                0, 0, entries, &size, &numEntries);
        if (res == ERROR_BUFFER_TOO_SMALL) {
                // Allocate enough memory to get
                // all the phonebook entries.
                delete[] entries;
                entries = new RASENTRYNAME[numEntries];
                entries[0].dwSize = sizeof(RASENTRYNAME);
                res = RasEnumEntries(
                        0, 0, entries, &size, &numEntries);
                if (res) {
                        char buff[256];
                        RasGetErrorString(
                                res, buff, sizeof(buff));
                        ShowMessage(buff);
                }
        }
        EntriesCB->Items->Clear();
        for (int i=0;i<(int)numEntries;i++)
```

```
                EntriesCB->
                        Items->Add(entries[i].szEntryName);
        EntriesCB->ItemIndex = 0;
}

void __fastcall
TForm1::DeleteBtnClick(TObject *Sender)
{
        // About to delete an entry so display a Yes/
        // No dialog to the user for confirmation.
        char buff[256];
        wsprintf(buff,
                "Delete phonebook entry '%s'?\nWarning! "
                "Do not delete your regular dialup entry.",
                EntriesCB->Text.c_str());
        int res = MessageDlg(buff, mtConfirmation,
                TMsgDlgButtons() << mbYes << mbNo, 0);
        if (res == mrYes) {
                // OK to delete the entry.
                RasDeleteEntry(0, EntriesCB->Text.c_str());
                // Reset the list of entries in the combo.
                EnumEntries();
        }
}
```

March 1999

Reappearing forms

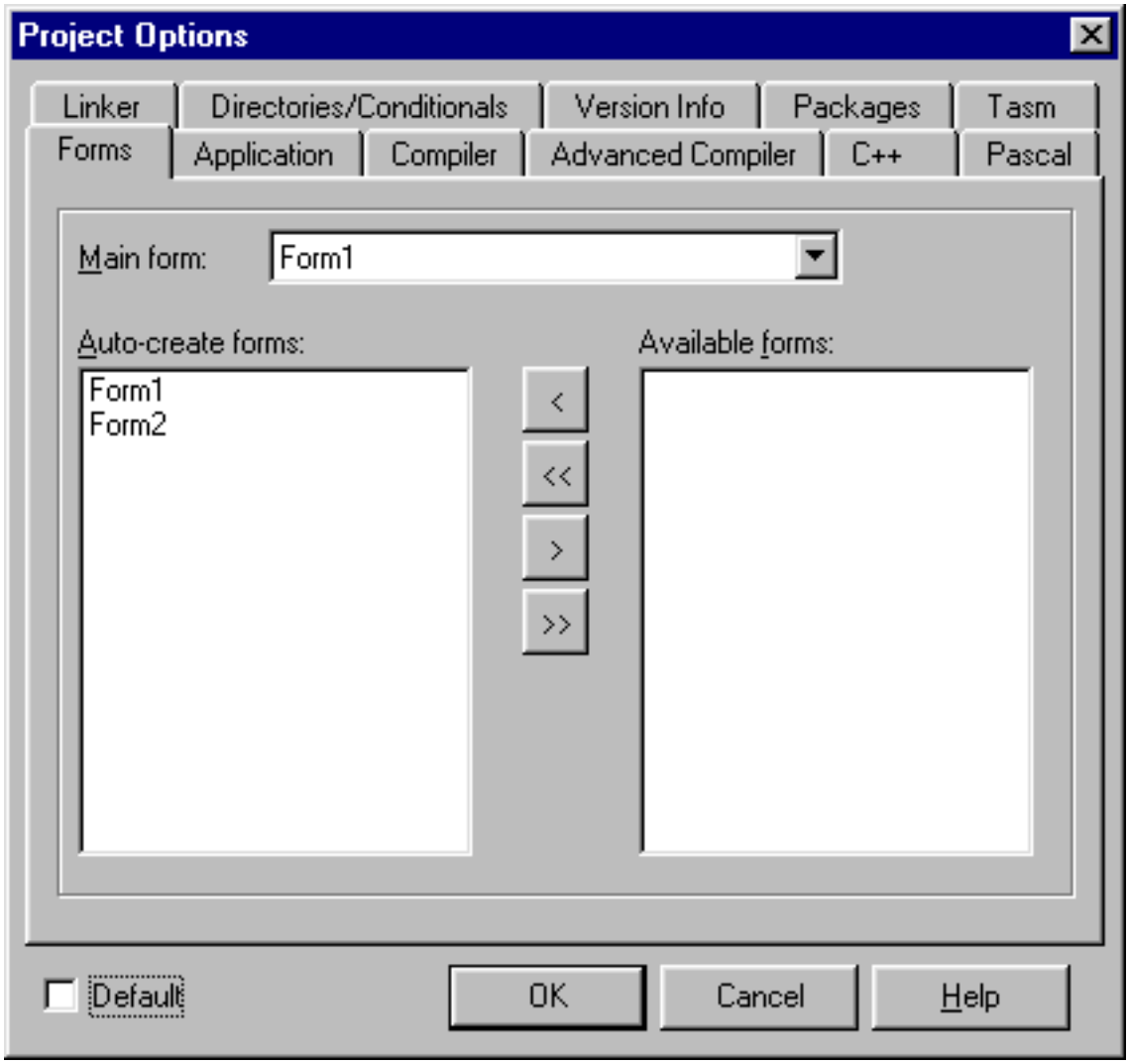
by Gerry Myers

What's the best way to hide forms until you need them? This seems like a simple question but there are really two answers. The first answer is simple: Auto-create all forms and use Show/Hide when needed. The second answer is a little more complicated, but generally gives better memory and performance results: Manually create forms and use Show/Destroy when needed. This article will explain some of the tradeoffs and help you choose the answer that's right for you, depending on your application.

Auto-create

If you're a C++Builder beginner, you may not be aware that all of your forms are automatically constructed when the program starts up. Only the forms that you set the Visible properties to true will be displayed, but all the forms are constructed nevertheless. Figure A shows the Forms tab of the Project Options dialog box.

Figure A: Here's the Forms tab of the Project Options dialog box.



Notice that, by default, all forms show up in the Auto-create list and the first one in the list is selected as the main form. All forms will be constructed in the order shown in that list. The Project Options dialog box gives you the ability to move forms up or down in the list or to select a different form as the main form. However, the one selected as the main form is always moved to the top of the list and constructed first.

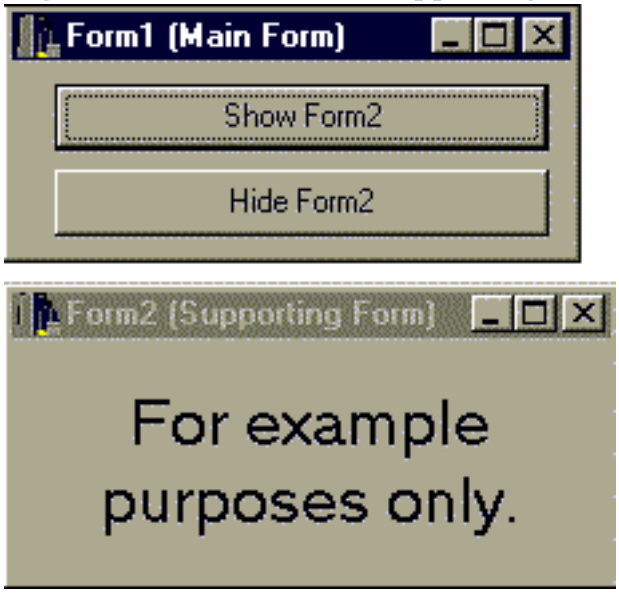
This approach by Borland made our programming jobs much easier. Since all forms are auto-created, they only need to be made visible by calling Show(). When a form is closed, it's really just hidden by the system and can easily be re-shown later.

The only exception to this is the main form. The form specified as the main form will be displayed at application startup, regardless of the value of its Visible property. When the main form is closed, instead of simply being hidden, it queues the application to terminate and all forms are destroyed. That's why the Project Options dialog box specifically names which form is the main form. The main form is constructed first, is displayed regardless of its Visible property, and, when closed, terminates the application.

Not having to reconstruct forms each time you need them is a great timesaver. All you need to do is call

Show() or Hide() to display or hide the forms. Figure B shows an example main form (Form1) and a supporting form (Form2).

Figure B: The main and supporting forms are examples for this technique.



Listing A shows the code for handling button events that show and hide the supporting form. The best part is that since the form is only hidden, when it's displayed again, it will be in the same position on the screen as before, and will retain all the entries and settings the user saw before.

Listing A: Simple Show/Hide code

```
//---------------------------------------
void __fastcall TForm1::ShowBtnClick(
                TObject *Sender)
{
        // Show the supporting form.
        Form2->Show();
}
//---------------------------------------
void __fastcall TForm1::HideBtnClick(
                TObject *Sender)
{
        // Hide the supporting form.
        Form2->Hide();
}
//---------------------------------------
```

Of course, with simplicity comes inefficiency. Since all forms are constructed at startup, they all take up memory. Even forms that you may only use once or perhaps never use (depending

on the selections made by the user) are constructed and chew up RAM. For small programs, this isn't much to worry about. However, for programs that have many and/or large form classes, the construction of these forms may severely cut into your available memory.

Another drawback is that your program may take longer to load during startup, since processing time must be used by the system to allocate memory and construct all of the forms. Again, for small applications, it's nothing to worry about. For larger projects, your user may end up staring at the hourglass cursor for 15-30 seconds while the program loads.

Manual create

The other way to hide your forms is not to create them until they're needed. This is done by opening the Project Options dialog box and moving selected forms from the Auto-create list to the Available list, as shown in Figure A. Only the forms listed in the Auto-create column will be constructed at program startup. The Available forms are just that. They're linked into the program and are *available* for your use. However, you must handle all construction, display, and destruction. Listing B shows the code required to handle the same button clicks as before, but this time treating the supporting form as just another dynamic class and handling its construction and destruction. Every form's class header file has a line at the bottom similar to *extern TForm2* Form2*. Every form's class implementation file has a line near the top, similar to *TForm2* Form2*. This pointer is used by the auto-create system but can also be set by you if the form is dynamically created.

Listing B: Dynamic code in main form

```
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
                : TForm(Owner)
{
        // Flag the form pointer as empty.
        Form2 = 0;
}
//---------------------------------------------------
void __fastcall TForm1::ShowBtnClick(
                TObject *Sender)
{
        // Construct the supporting form object.
        // if that hasn't been done yet.
        if ( !Form2 ) Form2 = new TForm2( this );

        // Show the supporting form.
        Form2->Show();
```

```
}
//---------------------------------------------------
void __fastcall TForm1::HideBtnClick(
                TObject *Sender)
{
        // Check to make sure the supporting form
        // object even exists.
        if ( Form2 )
        {
                // Release memory and clean up.
                delete Form2;

                // Flag the form pointer as empty.
                Form2 = 0;
        }
}
//---------------------------------------------------
void __fastcall TForm1::FormDestroy(
                TObject *Sender)
{
        // Make sure the supporting form is
        // cleaned up.
        if ( Form2 ) delete Form2;
}
//---------------------------------------------------
```

When the Show button is selected, the supporting form object must be created (using the new menu choice) and then shown. When the Hide button is selected, the form must be destroyed. To the user, this approach appears exactly the same as the Auto-create method. The supporting form appears and disappears when the buttons are selected.

Since we're handling the form object's construction and destruction ourselves, there are a couple of additional things we need to add to our code. In the main form's constructor, notice that we give an initial value of 0 to the supporting form's pointer. This serves as a flag, so that later we can tell whether an object has been created and assigned to this pointer.

Some code also had to be added to the form's destruction event handler. This is done so that the supporting form will be properly destroyed, in case the main form is closed by the user while the supporting form (Form2) is still open.

The main advantage of this approach is that memory isn't allocated until the user wants to see each supporting form, and is reclaimed when the user closes the form. Another

advantage is that, for programs with many supporting forms, the application will load and start up faster, since fewer objects need to be constructed at that time.

Things are not all rosy, though. This approach requires more coding and concentration on your part--as the developer. In addition, since the form object is re-created each time, its previous position and entries are lost unless you write even more code to save and restore them each time.

Conclusion

For most applications, it's probably fine to let all of your forms auto-create. However, when your program has many forms (and especially if some of those forms aren't used that often), it may be better to move some of your forms out of the Auto-create list and handle them yourself.

March 1999

Version: 1.0, 3.0

Using RAS, part 1

by Kent Reisdorph

Microsoft's Remote Access Server (RAS) can be used to establish a connection to a remote RAS server. Typically, this means dialing a modem to connect to a remote network, such as an Internet Service Provider (ISP). RAS isn't a terribly complex API, but there are significant differences in the RAS implementation between Windows NT and Windows 95. This article will explain how to use RAS to establish a connection to a remote machine. In part two of this series, we'll discuss some of the more advanced RAS functions and the built-in RAS dialogs.

How RAS works

RAS uses Windows' dial-up networking settings to make a connection. This assumes, of course, that RAS has been installed on the user's machine. RAS is installed when a user sets up dial-up networking on their machine. RAS isn't automatically installed for all Windows installations, though. If, for example, you have a permanent Internet connection, then you may never need to install dial-up networking. In that case, RAS will likely not be installed on your machine. Differences exist between Windows 95 and NT regarding how RAS operates. On Windows 95, dial-up networking automatically starts when an application attempts to connect to a remote machine. If, for example, you start a Web browser and type in a URL, Windows will automatically invoke RAS and will begin dialing. Under NT, however, this doesn't occur. It's up to the programmer to explicitly dial with RAS on NT machines. It's probably best to explicitly dial RAS in your applications so that you don't have to write code for a specific platform. In other words, always assume the lowest common denominator and write your programs accordingly.

Most of the time when dealing with RAS, you're dealing with a modem dialing out over a phone line. That isn't always true, however. Users using ISDN, for example, don't have a modem per se. Windows handles these details for you and you don't have to worry too much about the specific hardware on a particular user's machine.

RAS phonebooks

Key to RAS operations is Windows' concept of a phonebook. The phonebook contains the user's dial-up networking connection settings. This includes the device to use (usually a modem), the phone number to dial, the network connection settings, and so on. Most of the time, the phonebook will only contain one entry. In some cases, however, the phonebook will contain multiple entries. A user might have multiple entries if they typically connect to more than one remote machine. Let's say, for example, that a user has

phonebook entries for an ISP, a network at work, and one for CompuServe. The phonebook will then contain three entries. You must consider multiple entries when using RAS to dial a remote machine. We'll address retrieving the phonebook entries a bit later in the article. The phonebook is handled differently in NT and Windows 95. In Windows NT, phonebooks are contained in a phonebook file (a file with a .PBK extension). Complicating the issue is the fact that the user may have more than one phonebook file (although multiple phonebooks are rarely used). In Windows 95, there's only one phonebook and it's stored in the registry, as opposed to a separate file. Later, we'll look at how the phonebook comes into play when dialing with RAS.

Dialing with RAS

Before you can use RAS in your applications, you'll need to include the RAS.H and RASERROR.H headers to your source code. Once you've done that, you can get on with the business of dialing using RAS. Dialing is a multi-step process requiring these steps:

- Determine the phonebook entry that will be used to make the connection.
- Determine whether a connection already exists.
- Dial the modem (or other device).

We'll examine each of these steps in the following sections.

Getting the phonebook entry

As we said earlier, you're probably only dealing with one phonebook entry. However, you must account for the possibility of multiple phonebook entries. If only one entry exists in the phonebook, then you can simply use that entry to establish the connection. If multiple entries exist in the phonebook, then you should give the user the option of selecting the phonebook entry used to dial. Phonebook entries are obtained using the RasEnumEntries function. This function retrieves the entries contained in the phonebook. RasEnumEntries is one of those interesting API functions that can return a variable number of entries. It does this by filling a buffer with a variable number of RASENTRYNAME structures. Your code needs to take that fact into account. The RASENTRYNAME structure is very simple. Here's how it's defined in RAS.H (for single-byte character versions of Windows):

```
RASENTRYNAMEA
{
                DWORD dwSize;
                CHAR  szEntryName[RAS_MaxEntryName + 1];
};
```

As you can see, the szEntryName member is the only one of significance. When RasEnumEntries returns, this member will contain the name of a specific phonebook entry. At this point an example might help. Here's the first step in retrieving phonebook entries using RasEnumEntries:

```
RASENTRYNAME* entries = new RASENTRYNAME[1];
entries[0].dwSize = sizeof(RASENTRYNAME);
DWORD numEntries;
DWORD size = entries[0].dwSize;
```

The first line in this code snippet declares an array of RASENTRYNAME structures. We initially assume there's only one entry in the phonebook so the array size is 1. Following that, we set the dwSize member of the structure to the size of a RASENTRYNAME structure. This is a necessary step and is typical of many Windows API functions. Then we declare a variable called numEntries. This variable will contain the number of entries in the phonebook after RasEnumEntries returns. Finally, we declare a variable called size. This variable must initially be set to the size of the RASENTRYNAME structure. When RasEnumEntries returns, this variable will be set to the size of the buffer required to hold all of the entry names. Now we can actually call the RasEnumEntries function. Here's the code:

```
DWORD res = RasEnumEntries(
        0, 0, entries, &size, &numEntries);
```

The first parameter to RasEnumEntries is reserved and must be set to 0. The second parameter is used to specify the phonebook to enumerate. Under Windows 95, this parameter is ignored. In Windows NT you can set this parameter to the path and filename of a particular phonebook. If you specify 0 for this parameter under NT, then Windows will use the current default phonebook. Most of the time, this is sufficient. Some applications, however, need to take multiple phonebooks into account. If you're writing an application that must take multiple phonebooks into account, then you need to write code to handle multiple phonebooks. The third parameter in RasEnumEntries is the address of the buffer that will contain the list of RASENTRYNAME structures if RasEnumEntries returns successfully. The final two parameters are the buffer size and number of entries parameters. Notice that we pass the addresses of our size and numEntries variables for these parameters.

One of two scenarios will develop when you call RasEnumEntries. First, (and most likely) the function may return success on the first call (a return value of 0 indicates success). If this happens, then you know there was only one entry in the phonebook. The numEntries variable will contain the value 1 and the entries array will contain the entry. In this instance, you can simply use the value of entries[0].szEntryName to dial RAS.

In the second scenario, RasEnumEntries will return ERROR_BUFFER_TOO_SMALL. This return value indicates more than one entry in the phonebook. You need to re-allocate the buffer and call RasEnumEntries again.

You can re-allocate the buffer in one of two ways. The size variable will contain the required

size of the buffer. Using the size variable you can re-allocate the buffer like this:

```
delete[] entries;
entries = (RASENTRYNAME*)new char[size];
```

Or, if you prefer, you can use the numEntries to re-allocate the buffer like this:

```
delete[] entries;
entries = new RASENTRYNAME[numEntries];
```

Once you've re-allocated the buffer, you can call RasEnumEntries again. Before you do, however, you must set the dwSize member of the first structure in the array. For example:

```
entries = new RASENTRYNAME[numEntries];
entries[0].dwSize = sizeof(RASENTRYNAME);
res = RasEnumEntries(
        0, 0, entries, &size, &numEntries);
```

If RasEnumEntries returns 0 you can enumerate the array to retrieve the list of entry names. Typically, you'd place this list in a combo box to allow the user to select the entry he wants to use to dial. The code might look like this:

```
for (int i=0;i<(int)numEntries;i++)
        EntriesCb->Items->Add(entries[i].szEntryName);
```

Now you can move on to the second step, determining whether a connection currently exists.

Detecting an active connection

Sometimes a user might connect to the Internet using one application and then expect any other Internet-based applications to use the existing connection. Put another way, if the user is already connected to a remote machine, your application might as well use that connection. The RasEnumConnections function is used to detect active connections. Here again, there might be more than one active connection. While this situation is fairly uncommon, you need to understand that such a condition might exist.

Detecting an active connection is a two-step operation:

1. Check to see whether there are active connections.
2. Check the status of any active connections.

Here's the code for enumerating active connections:

```
RASCONN rc;
rc.dwSize = sizeof(rc);
DWORD numConns;
DWORD size;
DWORD res =
        RasEnumConnections(&rc, &size, &numConns);
```

This code should look familiar, as it's very similar to the code we used when we called RasEnumEntries. We're technically cheating by assuming that there will only be one active connection. If you need to account for more than one active connection, then you should use the allocate-check-re-allocate method described earlier for RasEnumEntries. If RasEnumConnections returns 0 (success), then you can check the value of the numConns variable. If numConns is 0, you know that there are no active connections. Then you can move directly to dialing RAS. If numConns is 1, then there's one active connection and you should check that connection's status. If numConns is greater than 1, you should enumerate the connections and look for the connection that matches the phonebook entry name obtained earlier.

The RASCONN structure has two members of note. The hrasconn member contains a handle to an active connection. The szEntryName member contains the phonebook entry used to make the connection.

Once you've detected an active connection, you need to check the status of that connection. This is accomplished using the RasGetConnectStatus function. Here's the code:

```
RASCONNSTATUS status;
status.dwSize = sizeof(status);
res = RasGetConnectStatus(rc.hrasconn, &status);
```

In this code snippet, we're calling RasGetConnectStatus passing the hrasconn parameter of the RASCONN structure obtained earlier with the call to RasEnumConnections and the address of a RASCONNSTATUS structure. If RasGetConnectStatus returns 0 (indicating success), you should check the rasconnstate member of the RASCONNSTATUS structure to determine the status. The rasconnstate member can contain either RASCS_Connected or RASCS_Disconnected. If rasconnstate is RASCS_Connected, then you can use the active connection. A value of RASCS_Disconnected indicates an active connection that, for whatever reason, is no longer valid. If the status is RASCS_Connected, you can save the hrascon parameter of the RASCONN structure to be used later. For example:

```
if (status.rasconnstate == RASCS_Connected)
```

```
        // Good connection handle, save it
        TheHandle = rc.hrasconn;
else
        // A connection was detected but its
        // status is RASCS_Disconnected.
        TheHandle = 0;
```

Now we can move on to actually dialing RAS.

Dialing RAS

Now that you've determined the phonebook entry to use and the existence of active connections, you can dial. Again, dialing is a multi-step operation.

Setting the RAS dialing parameters

The first step in dialing is filling out a RASDIALPARAMS structure. Consider this code:

```
RASDIALPARAMS params;
params.dwSize = sizeof(params);
strcpy(params.szEntryName, PBEntry.c_str());
strcpy(params.szPhoneNumber, "");
strcpy(params.szCallbackNumber, "");
strcpy(params.szUserName, "");
strcpy(params.szPassword, "");
strcpy(params.szDomain, "TURBOPOWER");
```

First, note how we assign the szEntryName member. In this code example, PBEntry is an AnsiString that contains the name of the phonebook entry to use. This value is obtained as described earlier in the section, "Getting the phonebook entry." Alternatively, you can specify an empty string for the szEntryName member. When you do that, Windows will establish a simple modem connection on the first available port. If you use this method, you must provide a phone number to dial in the szPhoneNumber member. Note that in this example, the szPhoneNumber member is set to a blank string. This causes Windows to use the phone number as determined by the phonebook entry. If you want to override the phonebook entry's phone number, you can specify the new number here. The szCallbackNumber is set to an empty string, as well. This member is used when the caller requests the remote machine to call back the local machine (Windows NT only).

The szUserName and szPassword members are used to log on to the remote machine. When these members contain an empty string, Windows behaves in two different ways depending on whether the operating system is Windows NT or Windows 95. With NT, Windows uses the

current user's logon credentials to log on to the remote server and no further user input is required. Windows 95, on the other hand, can't use the current logon credentials. Under Windows 95, the user will be presented with a log on dialog where they can specify their user name and password. In either situation, you can specifically set the szUserName and szPassword in code if you prefer.

The szDomain member of the RASDIALPARAMS structure can be set to the domain name of the machine to which you are connecting or to one of the following values:

"" The domain in which the remote access server is a member
"*" The domain specified in the phonebook entry settings

In most situations, you should set this member to an asterisk so the phonebook entry settings are used.

Synchronous vs. asynchronous dialing

The call to RASDial can be either synchronous or asynchronous. When using synchronous mode, the call to RASDial doesn't return until the connection operation has been carried out. If the connection was successful, RasDial will return 0. If an error occurred during dialing, RasDial will return one of the RAS error codes (defined in RASERROR.H). While synchronous mode is fairly simple to implement, it suffers from one major drawback: your application can't provide any status information to the user when in synchronous mode. When RasDial is called in asynchronous mode, RasDial begins dialing and then immediately returns control to the application. In asynchronous mode, a callback function is called at various stages of the dialing operation (the RAS callback function is described in the next section). The callback function can be used to provide feedback on the dialing operation to the user (connecting, logging on, authenticating, authenticated, etc.). In addition, you can provide error information to the user if an error occurs while connecting to the remote machine. Asynchronous mode isn't difficult to implement, and provides more control over the dialing operation. You should use asynchronous mode most of the time.

Synchronous or asynchronous operation is determined by the way RasDial is called. If the address of a callback function is provided, then RasDial operates asynchronously. If no callback function is provided, then RasDial operates synchronously.

Providing a RAS callback function

Much of the time, you'll execute RasDial asynchronously. When operating in asynchronous mode, RAS will notify your application when RAS events occur. RAS can use several methods of providing your application with status information. The RAS messaging system type is

determined by the value of the dwNotifierType parameter of RasDial. Table A shows the possible values for this parameter and the type of messaging system they represent.

Table A: Possible dwNotifierType values

0xFFFFFFFF	A window handle which receives WM_RASDIALEVENT messages
0	A RasDialFunc callback function
1	A RasDialFunc1 callback function
2	A RasDialFunc2 callback function

You'll probably use a RasDialFunc1 callback function. This type of callback provides both status and error messages (the RasDialFunc callback doesn't provide error messages). In situations where multilink connections are possible, you may elect to use the RasDialFunc2. This callback provides more information than RasDialFunc1 does. A typical RAS callback function might look like this:

```
VOID WINAPI RasCallback(HRASCONN hrasconn,
        UINT unMsg, RASCONNSTATE rascs,
        DWORD dwError, DWORD dwExtendedError)
{
        String S = "";
        if (dwError) {
                char buff[256];
                RasGetErrorString(
                        dwError, buff, sizeof(buff));
                // display error message
                return;
        }
        switch (rascs) {
                case RASCS_DeviceConnected :
                        S = "Connected..."; break;
                case RASCS_Authenticate :
                        S = "Logging on..."; break;
                case RASCS_Connected :
                        S = "Logon Complete"; break;
                // additional case statements
        }
        Form1->Memo1->Lines->Add(S);
}
```

This callback function provides error and status information. You can make your callback as

simple or as complex as you like, as determined by the type of application you're writing. A common mistake is to try to make the callback function a member of your main form's class. A callback must be a stand-alone function and cannot be a class member function.

Calling RasDial

Finally, you're ready to call RasDial. A typical call to RasDial looks like this:

```
DWORD res = RasDial(
                0, 0, &params, 1, RasCallback, &hRas);
```

The first parameter of RasDial is used to specify extended RAS features. The extended RAS features are only available under Windows NT. We aren't using the extended features here, so we set this parameter to 0.

The second parameter is used to specify the phonebook to be used for dialing. We set this parameter to 0 so Windows will use the default phonebook. Under Windows NT, you can specify a phonebook other than the default if multiple phonebooks are defined.

The third parameter is a pointer to a RASDIALPARAMS structure. We filled this structure out previously in step one of the dialing operation.

The fourth parameter is used to specify which RAS messaging mechanism to use. Now, we pass the value 1 to indicate that we're using a RasDialFunc1 callback type. The fourth parameter is the address of the callback function itself. In this example, the name of the callback function is RasCallback. The fourth parameter is also the address of a variable that will contain a RAS connection handle. This variable will contain the connection handle when RasDial returns. If RasDial returns 0, the call was successful. If a non-zero value is returned, an error occurred. If that happens, you can use RasGetErrorString to obtain an error message to display to the user.

Terminating the connection

If you've established a new connection, then you should terminate that connection before your application terminates. Terminating a RAS connection is achieved with the RasHangUp function. The following code illustrates the proper method of calling RasHangUp:

```
RasHangUp(hRas);
DWORD res = 0;
while (res != ERROR_INVALID_HANDLE) {
        RASCONNSTATUS status;
```

```
        status.dwSize = sizeof(status);
        res = RasGetConnectStatus(hRas, &status);
        Sleep(0);
}
```

Notice that we call RasHangUp, passing the handle to the RAS connection we're terminating. The while loop insures that RAS has completed the disconnect process before allowing the application to continue on its way. Listing A shows the code for the main unit of an example program that illustrates the concepts discussed in this article. This application establishes a RAS connection on a button click. It then downloads a file from TurboPower Software's FTP site on a second button click. Finally, it hangs up the connection on a third button click. Status messages are shown in a memo on the form. We don't show the main form's header or the code for a second unit, which allows the user to select a phonebook entry. You can download the complete example program from our Web site.

Conclusion

RAS is a fairly complex and somewhat intimidating API. Establishing a simple RAS connection is involved, certainly, but once you understand how RAS works, it isn't so daunting. Next month, we'll examine some of the more advanced aspects of RAS, including connection paused states and the built-in RAS dialogs.

Listing A: RASEXU.CPP

```
#include <vcl.h>
#pragma hdrstop

#include "RasExU.h"
#include "EntriesU.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

// The RAS callback function
VOID WINAPI RasCallback(HRASCONN hrasconn,
        UINT unMsg, RASCONNSTATE rascs,
        DWORD dwError, DWORD dwExtendedError)
{
        String S = "";
```

```
        if (dwError) {
                // Error occurred, show the error string.
                char buff[256];
                RasGetErrorString(
                        dwError, buff, sizeof(buff));
                Form1->Memo1->Lines->Add(buff);
                return;
        }
        switch (rascs) {
                // Build a status string based on the
                // status message.
                case RASCS_PortOpened :
                        S = "Port opened..."; break;
                case RASCS_DeviceConnected :
                        S = "Connected..."; break;
                case RASCS_Authenticate :
                        S = "Logging on..."; break;
                case RASCS_Authenticated :
                        S = "Authenticated"; break;
                case RASCS_Connected : {
                        S = "Logon Complete";
                        Form1->DownloadBtn->Enabled = true;
                        break;
                }
                case RASCS_Disconnected :
                        S = "Disconnected"; break;
        }
        // Show the status message in the memo.
        if (S != "")
                Form1->Memo1->Lines->Add(S);
}

void __fastcall
TForm1::ConnectBtnClick(TObject *Sender)
{
        // Get the phonebook entry to use to dial.
        String PBEntry = GetPhoneBookEntry();
        if (PBEntry == "") {
                ShowMessage("Operation Cancelled");
                return;
        }
        // Check for existing connections.
        hRas = CheckForConnections();
        if (hRas) {
                Memo1->Lines->Add(
                        "Using existing connection...");
                Memo1->Lines->Add("Click the "
                        "Download button to transfer files.");
                DownloadBtn->Enabled = true;
                hRas = 0;
                // No need to call RasDial for an
```

```
                // existing connection.
                return;
        }
        Memo1->Lines->Add(
                "No current connection, dialing...");
        // Set up the connection params.
        RASDIALPARAMS params;
        params.dwSize = sizeof(params);
        strcpy(params.szEntryName, PBEntry.c_str());
        strcpy(params.szPhoneNumber, "");
        strcpy(params.szCallbackNumber, "");
        strcpy(params.szUserName, "");
        strcpy(params.szPassword, "");
        strcpy(params.szDomain, "*");
        // Dial.
        DWORD res = RasDial(
                0, 0, &params, 1, RasCallback, &hRas);
        if (res) {
                char buff[256];
                RasGetErrorString(res, buff, sizeof(buff));
                Memo1->Lines->Add(buff);
        }
}

HRASCONN TForm1::CheckForConnections()
{
        char buff[256];
        RASCONN rc;
        rc.dwSize = sizeof(rc);
        DWORD numConns;
        DWORD size;
        // Enumerate the connections.
        DWORD res =
                RasEnumConnections(&rc, &size, &numConns);
        if (!res) {
                // No connections, return 0.
                if (numConns == 0)
                        return 0;
                // Multiple connections. Should add code to
                // handle multiple connections and decide
                // which one to use.
                if (numConns > 1)
                        Memo1->Lines->Add("Multiple connections");
        }
        if (res) {
                // Error. Report it.
                RasGetErrorString(res, buff, sizeof(buff));
                Memo1->Lines->Add(buff);
        } else {
                // Get the connection status.
```

```
                RASCONNSTATUS status;
                status.dwSize = sizeof(status);
                res = RasGetConnectStatus(
                        rc.hrasconn, &status);
                if (res) {
                        // Error. Report it.
                        RasGetErrorString(
                                res, buff, sizeof(buff));
                        Memo1->Lines->Add(buff);
                        return 0;
                } else {
                        // Found connection, show details.
                        if (status.rasconnstate
                                        == RASCS_Connected) {
                                Memo1->Lines->Add("Device type: " +
                                    String(status.szDeviceType));
                                Memo1->Lines->Add("Device name: " +
                                    String(status.szDeviceName));
                                Memo1->Lines->Add("Connected to: " +
                                    String(rc.szEntryName));
                                return rc.hrasconn;
                        } else {
                                // A connection was detected but its
                                // status is RASCS_Disconnected.
                                Memo1->Lines->Add
                                        ("Connection Error");
                                return 0;
                        }
                }
        }
        return 0;
}

String TForm1::GetPhoneBookEntry()
{
        RASENTRYNAME* entries = new RASENTRYNAME[1];
        entries[0].dwSize = sizeof(RASENTRYNAME);
        DWORD numEntries;
        DWORD size = entries[0].dwSize;
        DWORD res = RasEnumEntries(0, 0, entries, &size, &numEntries);
        if (numEntries == 1)
                return entries[0].szEntryName;
        if (res == ERROR_BUFFER_TOO_SMALL) {
                // allocate enough memory to get all the phonebook entries
                delete[] entries;
                entries = new RASENTRYNAME[numEntries];
                entries[0].dwSize = sizeof(RASENTRYNAME);
                res = RasEnumEntries(0, 0, entries, &size, &numEntries);
                if (res) {
                        char buff[256];
                        RasGetErrorString(res, buff, sizeof(buff));
```

```
                        ShowMessage(buff);
                }
        }
        TPBEntriesForm* form = new TPBEntriesForm(this);
        for (int i=0;i<(int)numEntries;i++)
                form->EntriesCb->Items->Add(entries[i].szEntryName);
        form->EntriesCb->ItemIndex = 0;
        String S;
        if (form->ShowModal() == mrCancel)
                S = "";
        else
                S = form->EntriesCb->Text;
        delete form;
        delete[] entries;
        return S;
}

void __fastcall
TForm1::HangUpBtnClick(TObject *Sender)
{
        if (!hRas) return;
        // Hang up.
        RasHangUp(hRas);
        // Be sure the RAS state machine has cleared.
        DWORD res = 0;
        while (res != ERROR_INVALID_HANDLE) {
                RASCONNSTATUS status;
                status.dwSize = sizeof(status);
                res = RasGetConnectStatus(hRas, &status);
                Sleep(0);
        }
        hRas = 0;
}

void __fastcall
TForm1::DownloadBtnClick(TObject *Sender)
{
        FTP->Connect();
}

void __fastcall
TForm1::FTPConnect(TObject *Sender)
{
        Memo1->Lines->Add(
                "Connected to TurboPower FTP Site");
        FTP->ChangeDir("pub");
}

void __fastcall
TForm1::FTPSuccess(TCmdType Trans_Type)
{
```

```
        switch (Trans_Type) {
                case cmdChangeDir : {
                        Memo1->Lines->Add("Changed directory");
                        FTP->Download(
                                "00index.txt", "00index.txt");
                        break;
                }
                case cmdDownload : {
                        Memo1->Lines->Add("File downloaded!");
                        break;
                }
        }
}

void __fastcall
TForm1::FormCreate(TObject *Sender)
{
        hRas = 0;
}

void __fastcall
TForm1::FormDestroy(TObject *Sender)
{
        // Terminate the connection if necessary
        if (hRas)
                HangUpBtnClick(this);
}
```

February 1999

Using callbacks in DLLs

by Kent Reisdorph

By far, the most common use of DLLs is where the calling program simply calls functions in the DLL. Sometimes, however, the calling application needs periodic information from the DLL while a process is taking place. For example, if the DLL is doing some lengthy calculations, it would be beneficial for the calling application to receive periodic reports of the calculation, such as the percentage complete. The calling application could use that information to display a progress bar or other indicator to the user. Callbacks allow you to do just that. This article will show you how to implement a callback function in a DLL.

What's a callback?

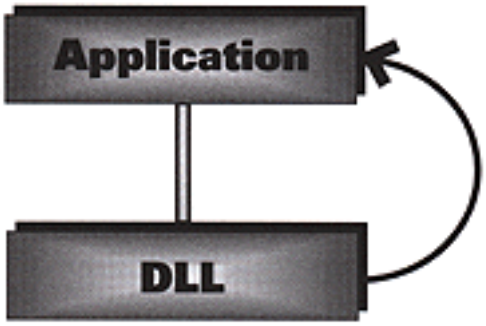
A *callback* is a function in an application that a DLL can call at suitable times. It's fairly obvious that an application can call functions in a DLL. This is the traditional relationship between a calling application and a DLL. Figure A illustrates the typical application/DLL relationship.

Figure A: Here's a typical application/DLL relationship.

This figure is a little misleading, however, because it indicates that the data flows in only one direction--from the calling application to the DLL. The DLL *can* return data to the calling application in the form of a function's return value, but we won't be addressing that here. Right now we're more concerned with the *dynamic* passing of data from the DLL to the calling application. So now we've established that it's typical for an application to call a function in a DLL. Consider, then, the case of a DLL calling a function in an application. In addition, suppose that the DLL has no knowledge of the calling application or its functions. In fact, the DLL doesn't even know that the caller is an application at all--it could be another DLL. As you've probably already surmised, it *is* possible for a DLL to call a function in an application, provided that it has a pointer to such a function. Figure B illustrates a DLL calling a callback function in

an application. Now that we've established what a callback is, we can move on to explaining how to implement a callback.

Figure B: A DLL calling a function in an application via a callback.



Writing the DLL

The first step in implementing a callback is writing a DLL that will make use of the callback. This part of the job can be broken into three pieces:

- Defining the callback function
- Declaring a type for the callback function
- Writing the code that uses the callback

We'll examine each of these steps in detail in the following sections.

Defining the callback function

The DLL determines the function signature of the callback function. The DLL knows what information it needs to pass to the application, and it describes that information in the signature of the callback function. Let's take the example of a callback function that provides percent-complete information to the calling application. The function signature for this type of callback could be relatively simple:

```
void CALLBACK MyCallback(int percent);
```

As you can see, this callback returns void and has only one parameter--an integer. This parameter will contain the percent complete value, expressed as an integer between 0 and 100. Note the use of the CALLBACK macro in the preceding declaration. The CALLBACK macro for Win32 is defined in WINDEFS.H. The declaration looks like this:

```
#define CALLBACK __stdcall
```

So CALLBACK, when used in this context, is simply a function declared to use the __stdcall calling convention. It's not required that you use CALLBACK but it's a good idea if you want to

enable developers using other development environments to use your DLL. Defining a function to use the CALLBACK modifier is something that other Windows programmers using your DLL will expect.

Declaring a type for the callback

Next, we must declare a new type for the callback function. This allows us to use the new type when we write the DLL function that will take a pointer to a callback function as a parameter. That probably doesn't make much sense right now, but we'll clarify it shortly. Given the prototype of the callback that we described earlier, the declaration of the new type would look like this:

```
typedef void CALLBACK(CallbackFunc)(int);
```

The typedef keyword associates the symbol name CallbackFunc with a pointer to a function. In this case the type has the signature described earlier. In other words, the CallbackFunc symbol is a new type that declares a pointer to a function, one that returns void and takes a single int for a parameter. You'll see how that type is used in the next section. Notice the use of the CALLBACK macro as described in the preceding section. This type definition should be placed in the DLL's header file.

Writing code that uses the callback

Now, we have an idea what the callback function should look like and a new type to describe that function. Next, we need to put the callback to use in the DLL. Here's an example of a function in the DLL, called CalcResults, that uses the callback:

```
void DLL_EXP CalcResults(CallbackFunc* Callback)
```

Should this be indented?

```
{
  for (int i=0;i<100;i++) {
    // do some processing here
    if (Callback)
      Callback(i);
  }
}
```

Let's take a moment to analyze this code. First, note that this function has a single parameter--a pointer to type CallbackFunc. It's easier to reference a function pointer by

name, which is only possible when you've declared a type for that function.

Notice also the use of the DLL_EXP symbol in this code example. The symbol tells the compiler whether the function is being exported or imported. This symbol is defined in the DLL's header like this:

```
#if defined(__DLL__)
  #define DLL_EXP __export
#else
  #define DLL_EXP __import
#endif
```

This code defines DLL_EXP as __export when building the DLL and __import when building the calling application. The CalcResults function contains a loop that performs some kind of processing. The important part of the loop, for our purposes, contains this code:

```
if (Callback)
  Callback(i);
```

This code first checks the Callback variable for a non-zero value. We must check the value of Callback because the user could pass 0 for this value when calling the CalcResults function. If Callback happens to be 0, then attempting to execute Callback will result in an access violation. Your DLLs should be written in such a way that your users can opt to use the callback or not, as they see fit. If Callback is 0, you just don't attempt to call the callback function. If Callback is non-zero, then the next statement executes the callback function, passing the variable i as a parameter. Don't be concerned if you find this line of code a bit odd. It is indeed strange syntax. The fact is that this is how you call a function when that function is represented by a function pointer. In short, this code calls the user's callback function each time through the loop, provided the user has defined a callback function. Now that we have the DLL part of the equation done, we can move onto the calling application.

Writing the calling application

The code in the calling application is relatively simple. First, declare a function to be used as the callback:

```
#include "MyDll.h"

void CALLBACK MyCallback(int percent)
{
  // code here to do something with
  //`percent'
```

```
}
```

There are two things to note about this function. First, note that the callback function is a stand-alone function and not a member of a class, such as the form's class. A callback function must be a regular function and can't be a class member function. (You could conceivably use a VCL event as a callback, but that would unnecessarily complicate the issue. In addition, the callback couldn't be used in non-VCL calling applications.) The second thing to note is that the function declaration for the callback exactly matches that of the CallbackFunc type declared earlier. Now that we've written the callback function, the only remaining item is to call the CalcResults function in the DLL. (This statement assumes that you've already added the import library file for the DLL to your calling application's project.) The code to call CalcResults is simple:

```
CalcResults(MyCallback);
```

Here, we call CalcResults, passing a pointer to the callback function, MyCallback, as a parameter. When this code executes, the CalcResults function will be called and, subsequently, the MyCallback function will be called at periodic intervals by the DLL.

Adding a return value

Many callbacks allow for controlling some aspect of the DLL's processing function via the callback's return value. For example, you might return true from the callback to continue processing, or false to stop processing. First, we need to modify the code in the DLL to account for the new callback signature. Here's the new declaration for the callback type:

```
typedef bool CALLBACK(CallbackFunc)(int);
```

Note that the return type is now bool instead of void as it was previously. Next, we need to modify the DLL's CalcResults function to utilize the return value from the callback. Here's how that function looks under the new mechanism:

```
void DLL_EXP CalcResults(CallbackFunc* Callback)
```

Indent?

```
{
  for (int i=0;i<100;i++) {
    // do some processing
    if (Callback)
      if (!Callback(i))
        break;
```

```
  }
}
```

This code checks the return value from the callback function and, if false, breaks out of the loop. It's as simple as that. Note that we used the traditional C++ shorthand for an if statement in this example. The long version would look like this:

```
if (Callback(i) == false)
  break;
```

Obviously some changes need to be made to the callback function in the calling application, as well. Going back to our example, let's say you wanted to stop processing when the percent completed reaches 60 percent. In that case, the callback function in the calling application might look like this:

```
bool CALLBACK MyCallback(int percent)
{
  // do some things
  if (percent < 60)
    return true;
  else
    return false;
}
```

This is the traditional notion of "return true to keep processing, or false to stop." Granted, not all processes need to be interruptible, but some do, and this is the method you should use for those cases.

Adding a user data parameter

There's one other technique that's commonly implemented in callback functions--the user data parameter. A user data parameter allows you to send any data you wish to the callback function. The user data parameter is almost invariably a DWORD. (For those of you not familiar with Windows API programming, a DWORD is an unsigned integer.) Why a DWORD? Because a DWORD is four bytes in size and can ultimately be used to pass any amount of data to the callback function. You can, for example, cast a pointer to a DWORD, thereby passing just about any amount of data to your callback function. Once again, let's modify the callback mechanism to add a user data parameter. First, let us show you the callback type definition:

```
typedef bool CALLBACK(CallbackFunc)(int, DWORD);
```

As you can see, a DWORD parameter has been added to the callback declaration. Now let's look at the modified CalcResults function in the DLL:

```
void DLL_EXP CalcResults(
  CallbackFunc* Callback, DWORD UserData)
{
  for (int i=0;i<100;i++) {
    // do some processing
    if (Callback)
      if (!Callback(i, UserData))
        break;
  }
}
```

Note that the CalcResults function now takes a DWORD as a parameter. Note, also, this line:

```
if (!Callback(i, UserData))
```

All we're doing here is passing the user data parameter of the CalcResults function straight through to the callback function. Our responsibility is to pass on the user data without modifying it in any way. In the calling application, we'll pass the form's pointer to the callback function in the user data parameter. Here's how the call to CalcResults looks with that modification:

```
CalcResults(MyCallback, (DWORD)this);
```

We're passing the pointer to the MyCallback function as we did before, but we're also passing the form's pointer. (Remember, the this keyword, when used within a class, is a pointer to the class itself--the form in this case.) As indicated earlier, the pointer must be cast to a DWORD because both the CalcResults function and the callback function take a DWORD for the user data parameter. Finally, let's look at the new implementation of the callback function in the calling application:

```
bool CALLBACK
MyCallback(int percent, DWORD userData)
{
  TForm1* form = (TForm1*)userData;
  form->ProgressBar1->Position = percent;
  return true;
}
```

Remember, we passed a pointer to the form when we called the CalcResults function. Now

we simply cast the userData parameter back to a TForm1* so we can access members of the form class. We then use this pointer to update the status bar on the form with the percent complete. Note that we're unconditionally returning true so that the processing runs to completion. If you always provide a user data parameter in your callbacks, your users will have all the functionality they need (in almost any circumstance). Adding a user data parameter is simple and makes your DLLs more useable and more professional.

Conclusion

Listing A shows the header for our test DLL, Listing B shows the source for the DLL, and Listing C shows the source for the calling application.

Listing A: CBDLL.H

```
#ifndef CBDLL_H
#define CBDLL_H

#if defined(__DLL__)
  #define DLL_EXP __export
#else
  #define DLL_EXP __import
#endif

// Type definition for the callback function.
typedef bool CALLBACK(CallbackFunc)(int, DWORD);

// Declaration for the CalcResults function.
extern "C" {
  void DLL_EXP CalcResults(
    CallbackFunc* Callback, DWORD UserData);
}

#endif
```

Listing B: CBDLL.CPP

```
#include <windows.h>
#pragma hdrstop

// Include the DLL's header.
#include "cbdll.h"

// The standard DLLEntryPoint function.
```

```
int WINAPI DllEntryPoint(
  HINSTANCE, unsigned long, void*)
{
  return 1;
}

// Our CalcResults function.
void DLL_EXP CalcResults(
  CallbackFunc* Callback, DWORD UserData)
{
  // A loop to simulate processing.
  for (int i=0;i<100;i++) {
    // Delay a bit to simulate some process.
    Sleep(50);
    // If a callback was provided, call it.
    if (Callback)
      // Pass i and userData. If the callback
      // returns false, stop processing.
      if (!Callback(i, UserData))
        break;
  }
}
```

Listing C: TESTAPP.CPP

```
#include <vcl.h>
#pragma hdrstop

#include "TestAppU.h"
// Include the DLL's header.
#include "cbdll.h"

#pragma resource "*.dfm"

TForm1 *Form1;

// Our callback function.
bool CALLBACK
MyCallback(int percent, DWORD userData)
{
  // Cast userData back to a TForm pointer.
  TForm1* form = (TForm1*)userData;
  // Update the status bar.
  form->ProgressBar1->Position = percent;
```

```
  // Return true to keep processing.
  return true;
}

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}

void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  // Call CalcResults, passing the address of
  // the callback function and a pointer to the
  // form as parameters. Must cast 'this' to a
  // DWORD to make the compiler happy.
  CalcResults(MyCallback, (DWORD)this);
  ProgressBar1->Position = 0;
  Label1->Caption = "Done!";
}
```

The calling application is simply a form with a button, a progress bar, and a label. When you click the button, the CalcResults function is called and the callback function updates the status bar. The CalcResults function doesn't actually do any processing (other than impose a short delay), but it does illustrate the concepts presented in this article.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

February 1999

Constructing constructors

by Kent Reisdorph and Jody Hagins

Writing a good C++ class involves more than just adding a few methods and data members. If you want your classes to be both flexible and powerful, then you should write several constructors for them. Many programmers don't realize that they should have several constructors for their classes. A C++ class may have one or more of the following constructors:

- Default constructor
- Copy constructor
- Overloaded constructors

In this article, we'll discuss the various types of C++ constructors, their benefits, and their usage.

Order of execution

Before explaining the various constructors, it's important that we understand what happens when a constructor, of any kind, is called. Before the code contained in a constructor is called, several things take place. First, if the class is derived from another class, a constructor is called to initialize the base class. Next, each member variable in the class is constructed. Finally, the constructor function itself is called. When you declare a constructor, you have control over how each phase is executed. If the constructor doesn't have an initializer list (see the article, "Constructor initializer lists"), then the compiler will generate code for each phase. If the compiler has to generate a call to a base class constructor, it will always generate a call to the base class' default constructor. This can result in subtle bugs, and is explained fully in the section on constructors in derived classes. If the compiler has to generate calls to member variable constructors, the compiler will always construct the member variable with its default constructor.

Thus, by the time your actual constructor code (the statements within the constructor code block) executes, you're insured that the base class and all members have been constructed. Note that built-in data types don't have constructors of any type, so their data must always be initialized before use.

The default constructor

The purpose of a constructor is to perform any initialization required by the class. The default constructor, however, takes no parameters, which means that it constructs every

instance in the same (default) way. Here's an example of a class that declares a default constructor:

```
 class MyClass {
  public:
    MyClass();
    int Multiply() { return x * y; }
    void SetXY(int _x, int _y) {
      x = _x;
      y = _y;
    }
  private:
    int x;
    int y;
};
```

The definition of the MyClass constructor might look like this:

```
MyClass::MyClass()
{
  x = 0;
  y = 0;
}
```

In this case, the x and y variables are initialized to 0.

Note: We're using inline functions for the Multiply and SetXY functions to shorten the listings. If you're unfamiliar with inline functions, see the C++Builder Help, under the topic "inline functions." We've made the constructors regular member functions to separate them from the class declaration.

The C++ language requires every class to have at least one constructor. Thus, if the class doesn't explicitly declare at least one constructor, then the compiler will generate a default constructor. The following code, for example, compiles and runs just fine (note that no constructor is defined for the class):

```
class MyClass {
  public:
    int Multiply() { return x * y; }
```

```
    void SetXY(int _x, int _y) {
      x = _x;
      y = _y;
    }
  private:
    int x;
    int y;
};

// later...
MyClass c;
Label1->Caption = c.Multiply();
```

The results, however, might not be what you expect, since the compiler-generated default constructor does nothing at all. In fact, it will look very much like this:

```
MyClass::MyClass() { }
```

The default constructor, generated by the compiler, simply calls the default constructor for each member variable in the class. However, built-in types don't have constructors, so the values for *x* and *y* will be uninitialized, and will contain whatever happened to be in that memory location before the class was instantiated. By providing a default constructor you control how your class is initialized. The compiler will generate a default constructor only if you haven't declared a constructor of any kind. Thus, it's possible to declare a class that has no default constructor. At times, it may be appropriate to declare a class that doesn't have a default constructor. However, such classes can't be created without explicit constructor arguments. Thus, you can't easily create arrays of the class, nor can you use the class in most template classes. In addition, virtual base classes that don't have default constructors are a nightmare to deal with.

The copy constructor

The purpose of the copy constructor is to construct an instance of the class with the data contained in an existing instance of the class. A copy constructor has this function signature:

```
MyClass(const MyClass& mc);
```

Note that the copy constructor takes a const reference to the class itself. A simple copy constructor would look like this:

```
MyClass::MyClass(const MyClass& mc)
{
```

```
  x = mc.x;
  y = mc.y;
}
```

Here we make a simple member-by-member copy of the class' data members. (Some C++ legalists argue that the copy constructor's parameter be named rhs, for *right hand side*. Whether or not you follow that convention is up to you.) As with the default constructor, the compiler will provide a copy constructor if you haven't defined one. Unlike the rules for generating a default constructor, the compiler will generate a copy constructor any time one isn't explicitly declared (and the compiler detects the need for one), regardless of how many other constructors are already declared. In fact, the previous example of a copy constructor isn't necessary at all because it exactly duplicates what the compiler-generated copy constructor would do for you, which is to do a member-by-member copy by calling the copy constructor of each member.

The compiler-generated copy constructor is fine for simple classes, but it won't work properly for more complex classes. If you have any dynamically allocated data, for example, you should provide your own copy constructor to insure that the data is properly copied. In this case, you'd probably want your own default constructor, assignment operator, and destructor, as well (see the article entitled, "Concrete data types"). Consider the following example:

```
class MyClass {
  public:
    MyClass();
    MyClass(const MyClass& mc);
    int Multiply() { return x * y; }
    void SetXY(int _x, int _y) {
      x = _x;
      y = _y;
    }
    char* GetData() { return data; }
    void SetData(char* _data) {
      free(data);
      data = strdup(_data);
    }
  private:
    int x;
    int y;
    char* data;
};

MyClass::MyClass(const MyClass& mc)
```

```
{
  x = mc.x;
  y = mc.y;
  data = strdup(mc.data);
}

MyClass::MyClass()
{
  x = 0;
  y = 0;
  data = strdup("");
}
```

Here we've introduced a data member called data. This member is a dynamically allocated character array that's intended to hold a string value. Examine the new copy constructor. Notice that it allocates memory for the string and copies the value of the incoming instance's data using the strdup function (a C++ runtime library function). In addition, you'll note the use of free to release the memory, since strdup is implemented with malloc, and mixing memory managers is a no-no (a topic for another time). Consider for a moment what would happen if we let the compiler generate the copy constructor. The compiler would generate something like this:

```
MyClass::MyClass(const MyClass& mc)
{
  x = mc.x;
  y = mc.y;
  data = mc.data;
}
```

In this case, we'd have this instance's data variable and the original instance's data variable both pointing to the same location in memory. The results would be undesirable, to say the least. Writing our own copy constructor insures that each instance of the class has its own memory for the data variable. When does the compiler use the copy constructor? Basically, in three cases. The first case is when an object instance is created from another object instance of the same type:

```
MyClass a;
a.SetData("This is a test.");
MyClass b(a);
Label1->Caption = b.GetData();
```

The third line in this example constructs a new instance, b, from the first instance, a, by passing a to the MyClass copy constructor. Note that the following code

```
MyClass b = a;
```

will use the copy constructor, as well. Some may think that the assignment operator would be called in this case; however, that's not true. The instance b doesn't exist, so nothing can be assigned to it. The compiler recognizes that b is being constructed and calls the copy constructor. Another case where the compiler uses the copy constructor is when an instance of the class is passed by value to a function. Here's an example:

```
void ShowData(MyClass c)
{
  MessageBox(0, c.GetData(), "Message", 0);
}

void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  MyClass a;
  a.SetData("This is a test.");
  ShowData(a);
}
```

When the ShowData function is called, the compiler invokes the copy constructor for the MyClass class. This only happens when a class is passed by value because the compiler makes a copy of the class instance before calling the function. When a class is passed by pointer or by reference this doesn't happen, because the compiler doesn't make a copy of the instance in those cases. The third case where the compiler invokes the copy constructor is when a function returns an instance of a class. For example:

```
MyClass GetClass()
{
  MyClass c;
  // do some things with 'c'
  return c;
}
```

Here, as with the previous case, the compiler makes a copy of c before returning it to the caller. Provide a copy constructor for your classes any time you need more control over the copy process. This is particularly true anytime dynamic data is involved.

Overloaded constructors

This is the most common type of constructor for most users. An *overloaded constructor* is simply a constructor that you define in order to facilitate initialization of the class. Consider our class thus far. It has data members called x and y and a method called SetXY to initialize those data members. It also has a data member called data and a method called SetData. We can create a constructor that will initialize all of these data members at one time. It looks like this:

```
MyClass::MyClass(int _x, int _y,
  char* _data)
{
  x = _x;
  y = _y;
  data = strdup(_data);
}
```

Now, we can create an instance of our class and initialize all of the data in one go:

```
MyClass a(10, 20, "Hello");
```

By adding default parameters, we can also make this constructor double as a default constructor. The new declaration would look like this:

```
MyClass(int _x=0, int _y=0,
  char* _data="");
```

Now, we can create an instance of the class using any of the following:

```
MyClass a;
MyClass b(10);
MyClass c(10, 20);
MyClass d(10, 20, "The text");
```

If parameters are passed to the constructor then those parameters are used. If any of the parameter is omitted the default value for that parameter will be used instead. We've written our new constructor to use default parameters, so it now serves as the default constructor for the class. Since that's the case, we need to remove our previous default constructor from the class. If we don't remove the previous default constructor, the following code will generate a compiler error:

```
MyClass a;
```

The compiler will complain about an ambiguity between the earlier default constructor and our new constructor (since either constructor can be called without parameters).

Constructors in derived classes

One of the most powerful features of C++ is the ability to reuse functionality through derivation. However, this language feature is also responsible for some critical misunderstandings, and thus, bugs. A derived class is the same as its base class, with some extra functionality and/or data. So, when a derived class is constructed, its base class constructor will be called first, followed by construction of the class data members. Finally, the statements in the constructor function block are executed. The big issue is which base class constructor is called when a derived class constructor is executed. Before we get into the special situations, let's dispense with the easy part. Let's assume that we don't define any constructors in the derived class. As we learned earlier, in this case the compiler will generate a default constructor and a copy constructor for the class. The compiler-generated default constructor will construct the base class by calling the base class' default constructor. Likewise, the compiler-generated copy constructor will construct the base class by calling the base class' copy constructor, passing the rhs argument. This behavior is what we would expect.

However, if you declare a constructor (of any kind) in the derived class, you must specifically call one of the base class constructors in the derived class' constructor initialization list. If you don't, then the compiler will automatically insert a call to the base class default constructor. This would be undesirable at best. Assume we want to derive a class from the MyClass class in the previous example. We might implement it like this:

```
class Derived : public MyClass {
  public:
    Derived() { i = 0; }
    Derived(const Derived& d) { i = d.i; }
  private:
    int i;
};
```

At first, this may seem perfectly fine. However, there's a subtle bug lurking about. Assume you have the following code:

```
Derived d1;
d1.SetData("Hello");
Derived d2(d1);
```

What's the value returned by d2.GetData()? It appears that it should be "Hello" since d2 was copy constructed from d1. However, the value will actually be an empty string. This is because the Derived copy constructor didn't explicitly call one of the base class constructors. Anytime a derived class constructor doesn't specifically call a base class constructor, the compiler will call the base class default constructor, which, in most cases, isn't what you want. Thus, for the above class, the copy constructor should be written as

```
Derived(const Derived& d)
  : MyClass(d) { i = d.i; }
```

Virtual constructors?

The C++ language doesn't support virtual constructors. Yet if you look at the TForm declaration in FORMS.HPP you'll see something like this (some superfluous code removed for clarity):

```
__fastcall virtual TForm(TComponent*
  AOwner);
```

If C++ doesn't support virtual constructors what's the virtual keyword doing in this declaration? The Visual Component Library is, of course, written in Object Pascal and Object Pascal does have virtual constructors. As a result, Inprise made changes to the C++ language to support virtual constructors. This only applies to VCL classes; however, and classes you derive from VCL classes. You can't declare constructors as virtual for your regular C++ classes (nor would you ever want to).

Conclusion

A well-written class allows full flexibility. By providing a default constructor, a copy constructor, and overloaded constructors, you control how your class functions. While the compiler does a fair job of providing default and copy constructors, you can maintain complete control over your class by writing your own constructors.

February 1999

Constructor initializer lists

by Kent Reisdorph

With C++ classes you have the option of initializing the class's data members in the constructor's initializer list. Let's say, for example, that you had a class, SomeClass, with integer data members called x, y, and z. You could initialize those data members in a default constructor like this:

```
SomeClass::SomeClass()
{
  x = 0;
  y = 0;
  z = 0;
}
```

This is something you commonly see in C++ programming. Now, look at the same default constructor using an initializer list:

```
SomeClass::SomeClass() :
  x(0), y(0), z(0)
{
}
```

If you haven't seen an initializer list used before, this might look odd to you. The initializer list follows the colon in a constructor's definition. The initializer list is actually a more efficient way of initializing a class's data members. When member variables are constructed in an initializer list, they're constructed with the given argument. However, if the initializer list is omitted, the member variables are first constructed with their default constructors, and then explicitly assigned in the code. Thus, while it doesn't matter much for built-in data types, like int, initializer lists are more efficient for any other data type. Here's another example. Let's say that SomeClass had another constructor defined as follows:

```
SomeClass(int _x, int _y, int _z);
```

The definition of this constructor would look like this when an initializer list was implemented:

```
SomeClass::SomeClass(int _x, int _y, int _z) :
  x(_x), y(_y), z(_y)
```

```
{
}
```

If your class is derived from another class, then the initializer list typically follows the call to the base class constructor. Note that if the base class constructor is omitted, then the compiler will insert a call to the base class' default constructor. If SomeClass were derived from a class called Base, then the default constructor would look like this:

```
SomeClass::SomeClass() :
  Base(), x(0), y(0), z(0)
{
}
```

Most of the time, use of the initializer list is optional. In some cases, however, you must use the initializer list. One such case is when you have a reference to a class as a data member. Take this class for example:

```
class MyRegistry {
  private:
    TRegistry& registry;
  public:
    MyRegistry();
};
```

This class has a reference to a VCL TRegistry object as a class member. You might think to initialize the registry member like this:

```
MyRegistry::MyRegistry()
{
  registry = *new TRegistry;
}
```

This code, however, will result in a compiler error. The only way to initialize the registry data member is in the initializer list:

```
MyRegistry::MyRegistry() :
  registry(*new TRegistry)
{
}
```

Because registry is a reference to a TRegistry object, you must initialize it in the initializer list. Another example is when you have a member variable (or base class) that doesn't have a

default constructor. In this case, you must construct the member (or the base class) in the initialization list by calling its appropriate constructor.

One subtlety that deserves mention is the order of initialization. While it seems logical that member variables are constructed in the order they appear in the initializer list, this is not the case. Instead, member variables are always constructed in the order in which they're declared in the class. Likewise, they're always destructed in the reverse order of construction. This can be very important, especially if the initialization of one member variable is dependent on the initialization of another.

Initializer lists are more than just syntactical sugar. In this article, we showed that they represent the preferred (and, in some cases, the only) way of initializing a class.

February 1999

Concrete data types

by Jody Hagins

A *concrete data type* (CDT) is a user-defined type that behaves just like a built-in type. In other words, a concrete data type can be created, deleted, assigned, copied, and passed as a parameter just like any C++ built-in type. This is a powerful feature that allows you to correctly make certain assumptions about how a specific class will behave. In this article, we'll learn how to build concrete data types, and, along the way, pick up on some subtleties of the C++ language itself.

A simple CDT

The simplest CDT you can create is

```
class Nada { };
```

Now, I admit that on the surface this class isn't very interesting. However, looks can be deceiving. This class may look like nothing, but it is, in fact, a complete and correct CDT. When the compiler is through with this class, it will look something like this:

```
class Nada
{
public:
  Nada() { }
  ~Nada() { }
  Nada(Nada const &) { }
  Nada & operator = (Nada const &)
    { return *this; }
  Nada * operator & () { return this; }
  Nada const * operator & () const
    { return this; }
};
```

For the purists among us, I should mention that the compiler generates only those methods that are actually called by the program. In other words, if you never create an instance of class Nada, the compiler won't generate any code. As another example, if your code doesn't call the copy constructor, the compiler won't generate one. A special note should also be made of the destructor. The compiler won't really generate a destructor for this class. In fact, all of the following conditions must be true for the compiler to generate a destructor

for any class:

- The class itself doesn't declare a destructor.
- The class is derived from another class.
- The base class has a destructor.

Also note that the compiler-generated destructor will be public, and will take on the virtual characteristic of the base class destructor. The Nada class may not be very interesting, but it gives us a unique look at the minimal requirements for a class to be a CDT. For a class to be considered a CDT, it must have correct public implementations of the following:

- Default constructor
- Copy constructor
- Assignment operator
- Destructor

For a given class, the compiler-generated code for any of the above may constitute a correct implementation. Then again, the class most likely will need to specifically define each of the above. We'll examine these requirements in more detail as we take the MyClass example from the "Constructing constructors" article, and make it into a CDT.

The constructors

The "Constructing constructors" article explains the ins and outs of constructors, so we won't dwell on them too much here. However, it's worth pointing out that a CDT copy constructor must construct the instance in such a way that it remains valid even if the object being copied is deleted. The MyClass constructors are fine the way they are, so they don't need to be modified in our CDT conversion process.

The destructor

We previously discovered that, in some cases, the compiler doesn't even create a constructor for a class. In these cases, how does the object clean up its member variables? When the compiler calls an object's destructor, the first thing it does is execute the code inside the destructor's code block. Then it automatically cleans up the member variables by destructing them--in the reverse order in which they were created. Next, it walks up the inheritance hierarchy, doing the same thing for each base class. If you don't declare a destructor, the compiler does the exact same thing, it just has no destructor code block to execute. Thus, you can rest safely, knowing that your member variables and base classes will be destructed, even if there is no explicit destructor. Symmetry is important in class design. In almost all cases, the destructor should perform the inverse operation of the constructor. Specifically, it should clean up any dynamic memory, close any open files, etc. So, in our attempt to convert MyClass into a CDT, we need to add the following code:

```
~MyClass() { free(data); }
```

The assignment operator

The assignment operator (also known as operator=) is often misunderstood. It's not the same as the copy constructor. Remember, the copy constructor actually constructs a brand new object instance, using another object instance to initialize itself. The assignment operator, however, assigns the value of one object instance to another, already existing instance. This is a subtle difference, but one which is very important.

As an example, consider a first attempt at writing a MyClass assignment operator:

```
void operator=(const MyClass& mc)
{
  x = mc.x;
  y = mc.y;
  data = strdup(mc.data);
}
```

This is a typical implementation, and it makes sense. In fact, it's identical to the copy constructor. After all, you're copying the data. However, the assignment operator only applies to existing objects. From our previous constructors, we can see that all existing objects have a data pointer that points to allocated memory. In this code, however, we're overwriting our data pointer, which causes a memory leak. What we really want to do is free this memory, and then assign it.

Our next attempt would look like this:

```
void operator=(const MyClass& mc)
{
  free(data);
  x = mc.x;
  y = mc.y;
  data = strdup(mc.data);
}
```

Now, that's more like it. We've eliminated the memory leak. Unfortunately, as is often the case, fixing one problem has now caused another. Consider the case where a MyClass object is assigned to itself. The data pointer for the instance being assigned is freed. However, this pointer is the same pointer as that for the instance that it's being copied from (since they're the same instance). Thus, the strdup function will now receive a pointer to memory that has already been freed. This is obviously not good news, and brings us to another attempt, in which we check to see if we're assigning an object to itself:

```
void operator=(const MyClass& mc)
{
  if (this != &mc)
  {
    free(data);
    x = mc.x;
    y = mc.y;
    data = strdup(mc.data);
  }
}
```

Ahhh. Now, surely, all is well with the world. We've eliminated all the bugs from the assignment operator. While this may be true, we still don't have a CDT. Remember the definition of a CDT? It must be able to be used just as any built-in data type. Consider the following code:

```
int i, j, k;
i = j = k;
MyClass x, y, z;
x = y = z;
```

The i = j = k line compiles (and rightly so). However, the x = y = z line doesn't, because the assignment operator returns a void. Thus, for a class to be a CDT, the assignment operator must return a reference (MyClass&) to the object being assigned. Why a reference? Why not an object, or a const reference? The answer to this question is hidden in another example of what's permitted for built-in types. Consider the following code:

```
int i = 1;
int j = 2;
int k = 3;
(i = j) = k;
```

First off, let me say that this won't even compile with a C compiler, but it's perfectly legal C++. What are the values of i, j, and k after the assignment? Let's break it down. First, the parenthesized expression is evaluated, which sets i to the value of j. So, i and j are both currently equal to 2. Now the return value of that expression is set equal to the value of k. However, what's the return value of the parenthesized expression? In C, the return value would be 2, which is not an l-value (thus the compiler error). In C++, the return value is i,

with i treated as an l-value. Thus, this line of code leaves i and k both equal to 3, and j equal to 2.

So, if the assignment operator of a class returns an object, or a constant reference to an object, you can't write code like this. Now, some would argue that you shouldn't write code like this in the first place, but that's beside the point. The fact of the matter is that if you want your classes to be CDTs, then you must return a reference to the assignee. The final version of the assignment operator is as follows:

```
MyClass& operator=(const MyClass& mc)
{
  if (this != &mc)
  {
    free(data);
    x = mc.x;
    y = mc.y;
    data = strdup(mc.data);
  }
  return *this;
}
```

Thus, in general (there are exceptions), the assignment operator first does the work of the destructor, and then does the work of the copy constructor.

Derivation and CDTs

If you want your derived classes to behave as CDTs, then you must ensure that the big four are implemented correctly. The implementation of the derived class is identical to what we learned about any CDT. However, when you derive a class from a base class, there are several issues you must remember. First, there are some gotchas related to writing constructors for derived classes. Those can be found in the article, "Constructing constructors". Next, we must concern ourselves with the destructor. Since the compiler takes care of calling the base class destructor, we really have nothing special to worry about. However, as a note, any class that's intended to be used as a base class should have a virtual destructor. Note that if the base class has a virtual destructor, and the derived class doesn't declare a destructor, the compiler will generate a virtual destructor for the derived class. Finally, for the assignment operator, we must address a similar issue as we saw for the copy constructor. If the derived class doesn't declare an assignment operator, the base class assignment operator will be called, followed by a member-by-member assignment of the members in the derived class.

However, if the derived class does declare an assignment operator, the base class assignment operator won't be called at all, unless the derived class assignment operator specifically calls it. Thus, a derived assignment operator would look like this:

```
Derived& operator=(const Derived & d)
{
  if (this != &d)
  {
    MyClass::operator=(d);
    i = d.i;
  }
  return *this;
}
```

Conclusion

Concrete data types are very powerful, in that, once built, they can be used just like built-in types, without special consideration to how and when they can be allocated, and in what manner they can be used. Granted, there are some special considerations to be taken, but once the implementation is sound, the user of the classes needn't be concerned with any hidden subtleties. We encourage you to create concrete data types out of all your classes.

February 1999

Word 97 OLE Automation

Object Linking and Embedding (OLE) provides us a simple way to share components between applications. It gives us the ability to use different object-based services in order to enable a full integration between components. OLE 2.0 introduced more powerful features such as OLE Containers and OLE Automation. In this article, we'll explain how to use OLE Automation to execute different Word 97 tasks from Borland C++Builder, like opening, closing, or printing files.

The Variant Class

In order to connect to Word 97, we'll use the Variant class. The Variant class can represent different dynamically changing values, such as strings, integers or OLE Automation objects. Now, we can open the Word 97 application (don't forget to include the <comobj.hpp> header required for using OLE Automation):

```
Variant wordApp;
if (wordApp.IsEmpty())
{
    wordApp=Variant::CreateObject("Word.Application");
}
else
{
    wordApp=GetActiveOleObject("Word.Application");
}
if (wordApp.IsEmpty())
{
    ShowMessage("Unable to find Word application.");
    return;
}
```

The CreateObject() method, executes Word if it's not already opened. If it is, it just creates a new window using GetActiveOleObject(). Once we're connected, the Variant class presents us three important methods: OleProcedure(), OlePropertyGet() and OlePropertySet(). For example, if we want to make Word 97 visible, we can just use the property Visible and set it to true.

```
    wordApp.OlePropertySet("Visible", (Variant) true);
```

Creating the application

First, we'll create a simple form. Choose File | New Application from the menu. Place five buttons on the form and call them Connect, Open, Save, Print, and Insert Table. Double-click the Connect button and insert the previous line of code into it. Compile the application and test it.

Opening / Saving a file

In order to open or save a file, we'll use the methods provided by the Documents object. Here's how to derive Documents from the wordApp:

```
Variant wordDocuments = wordApp.OlePropertyGet("Documents");
```

If you check the VBA help file that comes with Word 97, you'll see that the Documents object has five methods: Add(), Close(), Item(), Open(), and Save(). Here, we'll use the Open() method using the following syntax Open(*FileName*). For example:

```
// opens the "config.sys" file
wordDocuments.OleProcedure("Open", (Variant)
        "c:\\config.sys");
```

The Save() method is even easier:

```
wordDocuments.OleProcedure("Save");
```

Printing a file

As we saw, the Documents object doesn't give us a print method but the variant wordApp that we already created does. Now, using PrintOut() we'll print the current document:

```
wordApp.OleProcedure("PrintOut");
```

Insert a table

Let's suppose we want to open a new document and insert a table (3 rows and 5 columns). The AciveDocuments.Tables object contains the method Add() that we'll use in this example. Here's the syntax:

```
Add(Range As Range, NumRows As Long, NumColumns As Long)
```

Now you can easily insert the table as follows:

```
// creates a new document
wordDocuments.OleProcedure("Add");
Variant wordActiveDocument =
        wordApp.OlePropertyGet("ActiveDocument");
Variant wordTables =
        wordActiveDocument.OlePropertyGet("Tables");
Variant wordSelection =
        wrdApp.OlePropertyGet("Selection");
Variant Range = wordSelection.OlePropertyGet("Range");
// inserts the table
wordTables.OleProcedure("Add", Range, (Variant) 3,
        (Variant) 5);
```

Creating the table wasn't that difficult, but how can we now insert our own value in the 2nd row and the 3rd column for example. Again, we'll use the Tables object but this time Tables.Cell.Range.Text:

```
Variant wordTable1 = wordTables.OleFunction("Item",
        (Variant) 1);
Variant wordCell = wordTable1.OleFunction("Cell",
        (Variant) 2, (Variant) 3 );
Variant wordRange = wordCell.OlePropertyGet("Range");
wordRange.OlePropertySet("Text", (Variant) "We are at 2/3");
```

Conclusion

As we saw in this article, using OLE isn't difficult when we know the right object and method to use. The VBA help file contains the full list of all Word functions (you can find it in \Microsoft Office\Office\Vbawrd8.hlp). OLE gives us the flexibility to execute almost any Word task without even using Word. Plus, all other Microsoft Office applications like Excel, PowerPoint, or Access also include OLE Automation. The power is yours…

January 1999

Implementing a Help cursor

by Kent Reisdorph

You can download sample files from our Web site as part of the file jan99.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

Context-sensitive Help is a feature of Windows applications that users have come to expect. Most of the time, Help is invoked via the application's Help menu. Some Windows programs, however, go the extra mile: These programs provide a Help button on the toolbar for easy access to the program's context Help. When you click the Help button, the mouse pointer changes to display a Help cursor. Clicking on a menu, button, or edit control with the Help cursor displays a Help topic on that particular control. In this article, we'll show you how to implement a Help toolbar button on your main form. We'll also demonstrate how to implement the stock Windows Help button in a dialog box (the button that Windows puts on the title bar next to the close button).

Designing the Help system

The basic design of this type of Help system includes these elements:

- Help mode, which determines whether or not the user is seeking Help on a topic
- A Help button on the toolbar that toggles Help mode on and off
- A Help cursor indicating to the user that Help mode is on
- A Help file or other message system that displays Help information to users when they click on an item in Help mode

We'll examine each of these design points in the following sections.

Help mode

Help mode itself is the simplest part of the design. All you need is a Boolean variable called HelpMode. HelpMode can be declared in the form's class:

```
bool HelpMode;
```

This article's example program declares HelpMode in the private section of the form's class. You can declare the variable in the public, private, or protected section, as you see fit. When HelpMode is true, you'll display Help information to the user when a UI element (menu, button, or control) is clicked. When HelpMode is false, the program will operate normally. Take the File | Open menu item, for example. The **OnClick** handler for that menu item might look like this:

```
void __fastcall
TForm1::Open1Click(TObject *Sender)
{
  if (HelpMode)
    ShowMessage("Help for File|Open.");
  else
    if (OpenDialog->Execute())
    {
      Memo->Lines->
        LoadFromFile(OpenDialog->FileName);
    }
}
```

As you can see, if HelpMode is true, the code displays a message to the user. If HelpMode is false, the code shows the standard Open dialog box and loads the selected file into a memo. Note that you use ShowMessage to display simple text to the user. In a real Help system, you'd call the Application object's HelpContext method to display your Help file with the selected topic. You must consider one other aspect regarding Help mode. Should Help mode shut off immediately after the user chooses a menu item or control, or should the user be required to turn off Help mode by clicking the Help button a second time (a toggle)? The default Windows behavior is to shut off Help mode immediately after the user selects an item. When used in this way, Help mode could be considered a one-shot event. If you implement this method, you'll need to add a line of code in your **OnClick** events to shut off Help mode after displaying the Help topic:

```
HelpMode = false;
```

On the other hand, if your Help mode button acts as a toggle, then you may also need to modify the Help button on the toolbar (we'll address that in the next section). Regardless of

your Help mode operation, you should probably allow the user to click the Help button again to turn off Help mode, in case it was turned on accidentally.

Creating a Help button on the toolbar

How you implement a Help button on a toolbar depends on whether you're using C++Builder 1 or C++Builder 3. In C++Builder 1, you'll use a panel and speed buttons to build your toolbar. In C++Builder 3, you'll probably use a ToolBar component (although you can use a panel and speed buttons, if you desire). Regardless of the method you choose, the Help button should have these characteristics:

- A bitmap that's representative of the task the button performs
- Optionally, a Toggle mode that makes the button stay down until the user clicks the button again to turn off Help mode
- Code to toggle the Help mode on and off

For the toolbar's bitmap, you can use any bitmap that you feel represents Help mode. The example program available at the *C++Builder Developer's Journal* Web site contains a bitmap that you can use for the Help button, if you like.

As we discussed in the preceding section, you should decide whether you want your Help mode to be a toggle (on until it's shut off again) or a one-shot event. If your Help mode will be a toggle, you'll need to set properties for the toolbar button accordingly. In C++Builder 3, set the toolbar button's **AllowAllUp** property to true, and the **Style** property to tbsCheck. For C++Builder 1, see the TSpeedButton Help for instructions on how to make a speed button act as a toggle.

Even if your Help mode is a one-shot event, you may elect to have the Help mode toolbar button stay down while Help mode is in effect. If you go this route, you'll need to pop up the toolbar button programmatically when Help mode terminates. You can accomplish that with just one line of code:

```
HelpModeBtn->Down = false;
```

The toolbar button's **OnClick** event handler either turns on Help mode or toggles the Help mode. As we stated earlier, you should probably treat the toolbar button as a toggle, so the user can turn it off at any time. Given that, your **OnClick** event handler can contain just one line of code:

```
HelpMode = !HelpMode;
```

This code turns Help mode on if it's currently off, or off if it's currently on. It's as simple as that.

Displaying the Help cursor

Displaying the Help cursor for a main window requires you to handle the WM_SETCURSOR message in your application. The WM_SETCURSOR message is sent to an application every time the mouse pointer moves over the application. Obviously, that could result in your WM_SETCURSOR handler being called hundreds of times per second--so, any code in the handler needs to be short and efficient. To handle this message, you need to implement a VCL message map; see Listing B at the end of this article for an example. Once you have the message map set up, you'll provide a message handler for the WM_SETCURSOR message. The handler could be as simple as this:

```
void __fastcall
TForm1::WmSetCursor(TWMSetCursor& Message)
{
  if (HelpMode) {
    HCURSOR cursor = LoadCursor(NULL,
       IDC_HELP);
    ::SetCursor(cursor);
    Message.Result = true;
  }
  else TForm::Dispatch(&Message);
}
```

If the HelpMode variable is true, you call the Windows API function LoadCursor, passing NULL for the hInstance parameter and IDC_HELP for the final parameter. Doing so causes Windows to load its built-in Help cursor into the cursor variable. Now, the API function SetCursor is called to actually change the cursor.

Note: Global scope

In our example, double colon precedes the SetCursor function name (::). This is necessary because the TForm class has a protected method called SetCursor, as does the Windows API. The double-colon global scope operator tells the compiler to use the global scope SetCursor function, rather than the local scope SetCursor function. Use the global scope operator any time you need to call a Windows API function when the VCL has a function by the same name.

Most of the time, you'd write the calls to SetCursor and LoadCursor all on one line, like so:

```
::SetCursor(LoadCursor(NULL, IDC_HELP));
```

We broke the code into multiple lines in the preceding example to make it more readable. You might be wondering if all of this is completely necessary. After all, you can use the Screen object's **Cursor** property to control the application's cursor, right? That's true, but only up to a point. Specifically, changing the **Cursor** property will change the cursor when the mouse pointer is over the client area of the form--but *not* when the pointer is over the menu. You want the Help cursor to be displayed even when over the menu, and you must drop to the API level to accomplish that effect.

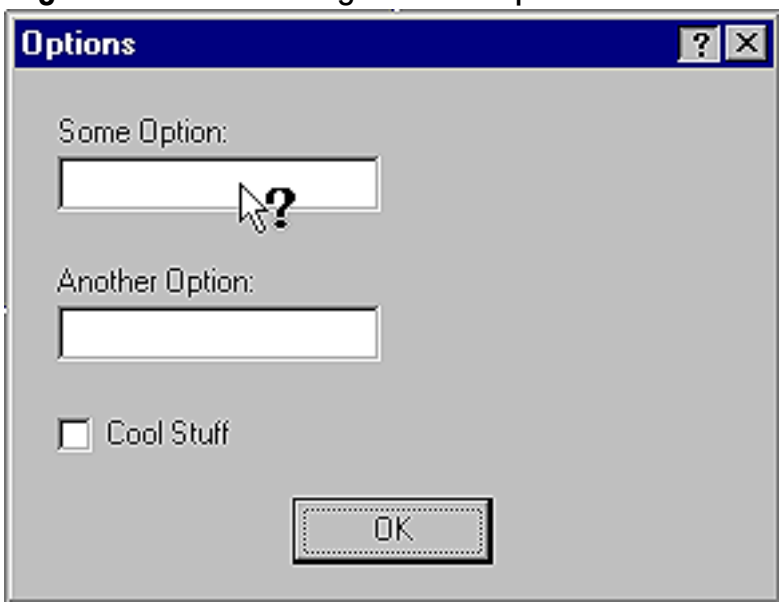
You can also change the application's cursor to the Help cursor by sending the form a WM_SYSCOMMAND message with a WPARAM of SC_CONTEXTHELP. When you click on a control on the form, the form will receive a WM_HELP message and the cursor will be automatically set back to the arrow cursor. Your application can then catch the WM_HELP message or assign an event handler for the Application object's **OnHelp** event. The handler could display any Help topic, based on the control clicked. This method, while attractive at first glance, suffers from the same limitation as setting the screen's **Cursor** property--it doesn't work when a menu item is clicked.

Using a dialog box's Help button

Windows has a built-in mechanism for context-sensitive Help in dialog boxes: You can add a button with a question mark to the dialog's title bar. When the user clicks the Help button,

the cursor changes to the Help cursor. Figure A shows a dialog box with the Help button and Help cursor in use.

Figure A: This dialog box's Help cursor is enabled.



Clicking on a control with the Help cursor will result in a WM_HELP message being sent to the form. This message is handled by VCL at the application level, and the Help file page associated with the control clicked will be displayed in a pop-up window. It's all more or less automatic. More or less--but it isn't totally automatic, because, in order to get this behavior, you must perform the following tasks:

- Set the form's **BorderStyle** property to bsDialog.
- Add the biHelp element to the **BorderIcons** property (a set).
- Assign a Help file to the application (either in the IDE or via the **HelpFile** property of the Application object).
- Assign context IDs for each topic in the Help itself.
- Set the **HelpContext** property for each component for which context Help is enabled.

As we stated earlier, clicking on a control with the Help cursor results in the form receiving a WM_HELP message. While the Help button on a dialog box is handled more or less automatically by the VCL, you can still catch this message yourself if you want to perform some special processing when the user clicks on a control with the Help cursor. Once again, you'll have to implement a VCL message map. Your message handler will receive a reference

to a TWMHelp structure. The TWMHelp structure contains a member called HelpInfo, which is a pointer to a Windows HELPINFO structure. You could use the information in the HELPINFO structure to determine what control was clicked. For example:

```
void __fastcall TForm2::WmHelp(TWMHelp&
    Message)
{
  if (Message.HelpInfo->hItemHandle
      == Edit1->Handle)
    ShowMessage(
      "Show Help for edit control #1.");
  // etc.
}
```

It may rarely be necessary for you to handle the WM_HELP message yourself, but you certainly have that flexibility, if needed. The listings at the end of this article don't show the source code for the example application's Options dialog box. You can, however, download the code for this article from our Web site and try the Help button in the Options dialog box. Click on any control in the dialog box, and a Help topic will be displayed.

Try it out

Listing A contains the header file for our example program; note the use of the message map. Listing B shows the main unit for the example program. It implements the Help cursor and Help mode to display context Help on demand. We've created a simple Help file to illustrate the concepts presented in this article.

Conclusion

Full context-sensitive Help support will certainly add value to your applications. Implementing a Help cursor is something that will enhance the program's value even further.

Listing A: HELPCURS.H

```
#ifndef HelpCurUH
#define HelpCurUH

#include <Classes.hpp>
```

```
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <Menus.hpp>
#include <ToolWin.hpp>
#include <Dialogs.hpp>

class TForm1 : public TForm
{
__published:  // IDE-managed Components
  TMainMenu *MainMenu1;
  TMenuItem *File1;
  TMenuItem *Exit1;
  TMenuItem *N2;
  TMenuItem *Save1;
  TMenuItem *Open1;
  TToolBar *ToolBar1;
  TToolButton *FileOpenBtn;
  TToolButton *FileSaveBtn;
  TToolButton *ToolButton3;
  TToolButton *HelpModeBtn;
  TImageList *ImageList1;
  TMemo *Memo;
  TOpenDialog *OpenDialog;
  TSaveDialog *SaveDialog;
  TMenuItem *Tools;
  TMenuItem *Options1;
  void __fastcall Open1Click(TObject *Sender);
  void __fastcall
    HelpModeBtnClick(TObject *Sender);
  void __fastcall Save1Click(TObject *Sender);
  void __fastcall Exit1Click(TObject *Sender);
  void __fastcall
    Options1Click(TObject *Sender);
private:        // User declarations
  bool HelpMode;
  void DoHelp(int ID);
  void __fastcall
    WmSetCursor(TWMSetCursor& Message);
public:    // User declarations
  __fastcall TForm1(TComponent* Owner);
  BEGIN_MESSAGE_MAP
```

```
    MESSAGE_HANDLER(
      WM_SETCURSOR, TWMSetCursor, WmSetCursor)
  END_MESSAGE_MAP(TForm)
};

extern PACKAGE TForm1 *Form1;

#endif
```

Listing B: HELPCURS.CPP

```
#include <vcl.h>
#pragma hdrstop

#include "HelpCurU.h"
#include "Options.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  // Set the Help mode to false initially.
  HelpMode = false;
}

void __fastcall
TForm1::Open1Click(TObject *Sender)
{
  // If Help mode is on, then call the
  // DoHelp function. If not, display the File.
  // Open dialog box. The event handlers for the
  // other menu items follow the same pattern.
  if (HelpMode)
    DoHelp(1);
  else
    if (OpenDialog->Execute())
    {
      Memo->Lines->
        LoadFromFile(OpenDialog->FileName);
    }
}
```

```
void __fastcall
TForm1::Save1Click(TObject *Sender)
{
  if (HelpMode)
    DoHelp(2);
  else
    if (SaveDialog->Execute())
    {
      Memo->Lines->
        SaveToFile(SaveDialog->FileName);
    }
}

void __fastcall
TForm1::Exit1Click(TObject *Sender)
{
  if (HelpMode)
    DoHelp(3);
  else
    Close();
}

void __fastcall
TForm1::Options1Click(TObject *Sender)
{
  if (HelpMode)
    DoHelp(4);
  else
    Form2->ShowModal();
}

void __fastcall
TForm1::HelpModeBtnClick(TObject *Sender)
{
  // Toggle the Help mode.
  HelpMode = !HelpMode;
}

void TForm1::DoHelp(int ID)
{
  // Show the Help file with the given ID.
  Application->HelpContext(ID);
  // Reset Help mode to off.
```

```
  HelpModeBtn->Down = false;
  HelpMode = false;
}

void __fastcall
TForm1::WmSetCursor(TWMSetCursor& Message)
{
  // If the Help mode is on, then we'll handle
  // the cursor changes ourselves.
  if (HelpMode) {
    // If the cursor is over the client window,
    // then display the 'NO' cursor.
    if (Message.CursorWnd == Memo->Handle)
      ::SetCursor(LoadCursor(NULL, IDC_NO));
    else
      // Otherwise display the Help cursor.
      ::SetCursor(LoadCursor(NULL, IDC_HELP));
    Message.Result = true;
  }
  // If Help mode is not on, then let VCL and
  // Windows handle the message.
  else TForm::Dispatch(&Message);
}
```

January 1999

A component for reading version information from a program file

by John M. Miano

You can download sample files from our Web site as part of the file jan99.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

Almost every program that goes into production will eventually exist in multiple versions. And, unless the application resides on a single computer, multiple versions will exist at the same time. Multiple versions of the same application create support problems--not the least of which is simply determining which version of the application is running on a particular computer. Suppose you're supporting insurance systems, and the users already have an auto-insurance application. Now you're rolling out a home-insurance application that shares some elements with the auto-insurance program. For example, the original system configuration program that shipped with the auto-insurance application has been upgraded so that it can configure both the auto- and home-insurance applications.

If someone re-installs the auto-insurance application, you don't want the old configuration program to be installed on top of the new one. (You'd be getting support calls about the home-insurance program not working.)

Windows lets you store program version information in an application's resource data. An installation procedure can call Windows API functions to extract the version of an existing file and compare this with the version of the new file.

Another version issue occurs when a user calls for support. The first thing a support person needs to know is the version of the application the caller is using. Traditionally, the application version is displayed on an About dialog box accessible through the Help menu.

If you're building a C++Builder application, you don't want to store the current version in both the application's resource data and in a label control on the About dialog box. If you have to keep two copies of the same information they'll invariably get out of synchronization. The solution for this problem is a VCL custom control that automatically updates a label control with information from the application's version resources. In this article, we'll show you how to create such a component. (For some background on accessing version information, see "Working with version information".)

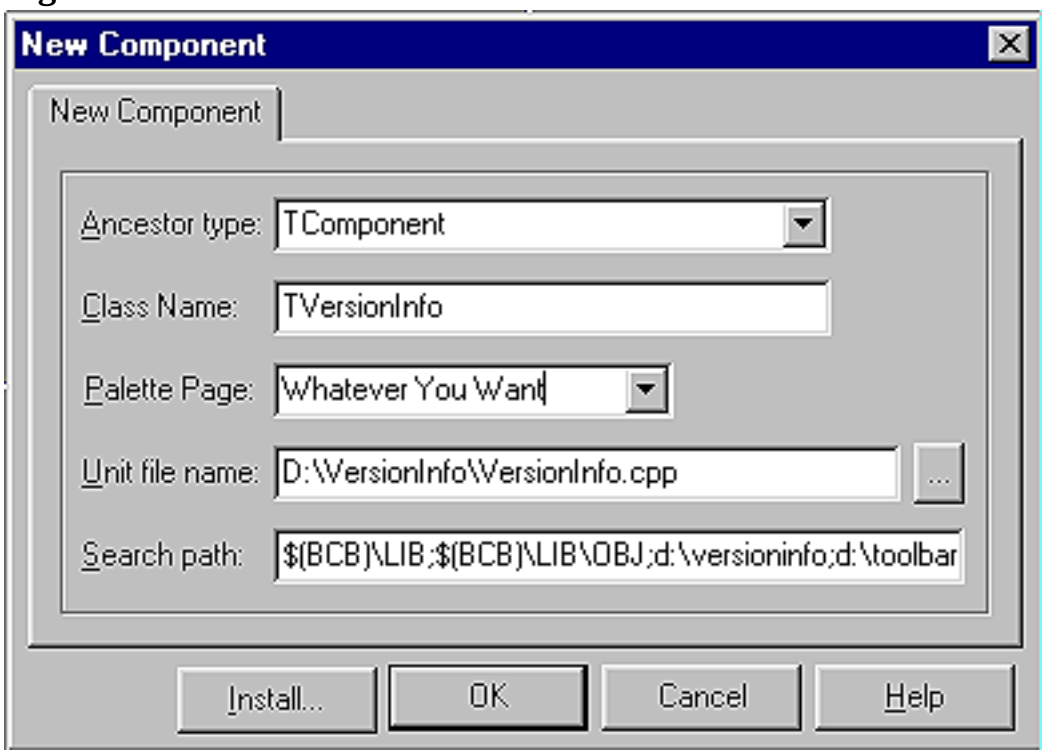
The TVersionInfo component

Our version-information component has two properties. **FileVersion** is a runtime, read-only property that returns the file version string; **FileVersionLabel** is a read/write property that specifies a TLabel control, which is automatically updated with the file version.

Creating the component

To create the component, start a new project and select Component | New Component from the C++Builder main menu. Fill in the resulting New Component dialog box, as shown in Figure A; then click OK. Because this will be a non-visual component, you use TComponent as the ancestor class.

Figure A: Create the TVersionInfo class.



The component class definition

The TVersionInfo class definition, containing two property definitions, is shown in Listing A. The **FileVersion** property is read-only and the value never changes while a program runs, so it simply references a member variable. **FileVersionLabel** is a design-time property that lets the user assign a label to automatically display the file-version string. This property uses the SetFileVersionLabel function to write its value.

Listing A: The TVersionInfo class definition, versioninfo.h

```
#ifndef VersionInfoH
```

```
#define VersionInfoH
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------
class PACKAGE TVersionInfo : public TComponent
{
private:
  String file_version ;
  TLabel *file_version_label ;

void __fastcall TVersionInfo::SetFileVersionLabel (
                                    TLabel *label)
  {
    if (ComponentState.Contains (csDesigning))
      label->Caption = "#File Version#" ;
    else
     label->Caption = file_version ;
    file_version_label = label ;
    return ;
  }
protected:
  virtual void __fastcall Notification(
                              TComponent *AComponent,
                              TOperation Operation) ;
  virtual void __fastcall Loaded () ;
public:
  __fastcall TVersionInfo(TComponent* Owner);

 __property String FileVersion = { read = file_version } ;
__published:
  __property TLabel *FileVersionLabel =
                             {
                               read = file_version_label,
                               write = SetFileVersionLabel
                             } ;
};
//---------------------------------------------------
#endif
```

The only reason for using a function to write to the **FileVersion** property is that the label's

caption will be updated when it's linked to the TVersionInfo control. At design time, the label is updated with the string *#File Version#*; at runtime it gets the actual version stored in the file. Using a constant string at design time avoids confusing the user. At design time, the control reads the version information for the C++Builder IDE rather than for the application.

The component implementation

The TVersionInfo class's CPP implementation file appears in Listing B. Because the file-version information doesn't change during the execution of a program, and because it's available throughout the program's execution, the TVersionInfo class extracts the version information in the class constructor. The class constructor uses the process described earlier to store the file-version string in the file_version member variable.

Listing B: The TVersionInfo class implementation, versioninfo.cpp

```
//---------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "VersionInfo.h"
#pragma package(smart_init)
//---------------------------------------------------
// ValidCtrCheck is used to assure that the components
// created do not have any pure virtual functions.
//

static inline void ValidCtrCheck(TVersionInfo *)
{
    new TVersionInfo(NULL);
}
//---------------------------------------------------
//
//  Description:
//
//    Class Constructor
//
__fastcall TVersionInfo::TVersionInfo(TComponent* Owner)
    : TComponent(Owner)
{
   file_version_label = NULL ;
```

```
 DWORD handle ; // Dummy, Windows does not use
                // this parameter.
 DWORD size = GetFileVersionInfoSize (
                 Application->ExeName.c_str (),
                 &handle) ;
 if (size == 0)
   return ; // No file information

 char *buffer = new char [size] ;

 bool status = GetFileVersionInfo (
                  Application->ExeName.c_str (),
                  0, // Unused Parameter
                  size,
                  buffer) ;
 if (! status)
 {
   delete [] buffer ;
   return ;
 }

 // Extract the language ID
 UINT datasize ;
 struct
 {
   unsigned short language ;
   unsigned short character_set ;
 } *translation ;
 status = VerQueryValue(
                    buffer,
                    "\\VarFileInfo\\Translation",
                    (void **) &translation,
                    &datasize) ;

String key
  = "\\StringFileInfo\\"
  + String::IntToHex (translation [0].language, 4)
  + String::IntToHex (translation [0].character_set, 4)
  + "\\FileVersion" ;
char *data ;
status = VerQueryValue(
                    buffer,
                    key.c_str (),
                    (void **) &data,
```

```
                        &datasize) ;
  if (status)
    file_version = data ;
  else
    file_version = "" ;
  delete [] buffer ;
  return ;
}

void __fastcall TVersionInfo::Notification(
                                TComponent *AComponent,
                                TOperation Operation)
{
  // We don't care about controls being added.
  if (Operation != opRemove)
    return ;

  if (AComponent == file_version_label)
    file_version_label = NULL ;
  return ;
}

void __fastcall TVersionInfo::Loaded ()
{
  if (file_version_label != NULL)
    file_version_label->Caption = file_version ;
  return ;
}

//---------------------------------------------------
namespace Versioninfo
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] =
          {__classid(TVersionInfo)};
        RegisterComponents(
          "Whatever You Want", classes, 0);
    }
}
```

Loaded() is a virtual function that VCL calls after all the components have been loaded. You use this function to update the TLabel control referenced by the **FileVersionLabel** property with the file version string, because you know that when Loaded() is called, the label will be

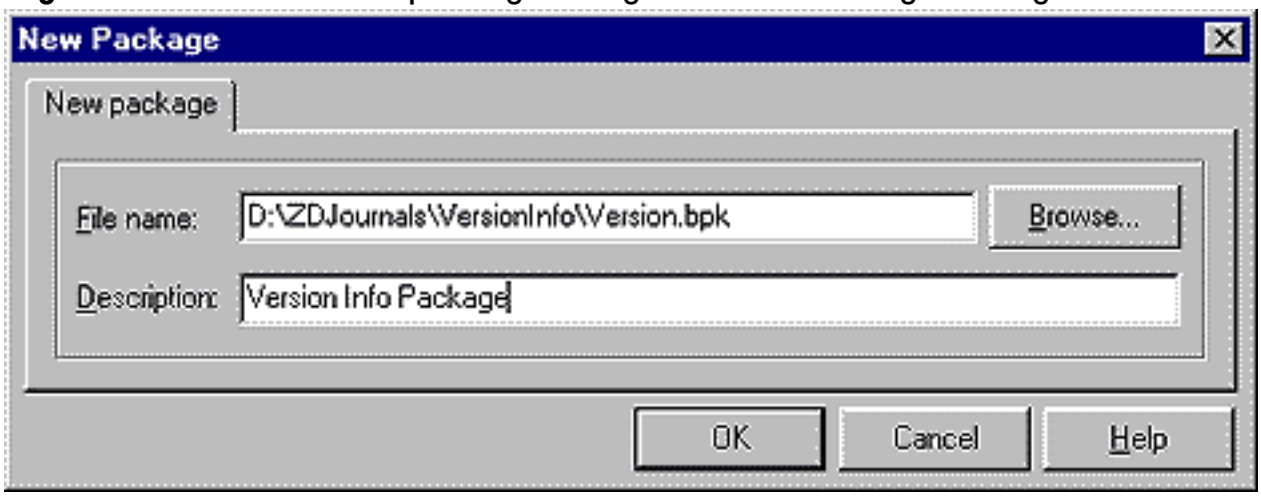
ready for updating.

The other virtual function used in this component is Notification(). VCL calls this function when a control is added to or removed from a form. Suppose you add a label control to a form and then assign the label a TVersionInfo control's **FileVersionLabel** property. If you then delete the label, the **FileVersionLabel** property would contain an invalid value. The Notification() function lets the component determine whether a component it's referencing is becoming invalid. If the TVersionInfo control references a TLabel control that's no longer valid, the Notification() function deletes the reference.

Installing the component

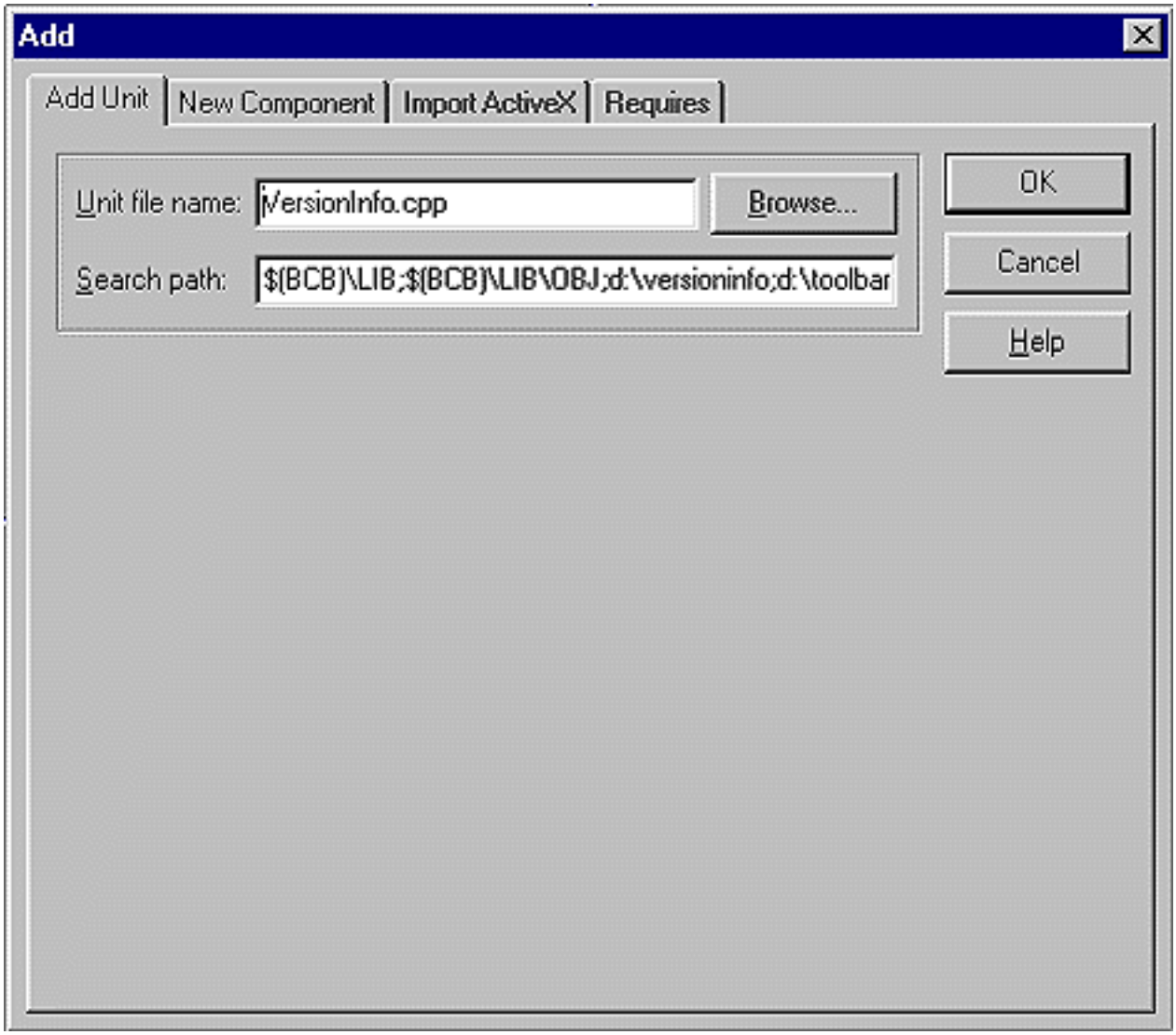
Having created the TVersionInfo component, you're ready to install it on the component palette in the C++Builder IDE. Select File | New from the main menu and then double-click on the Package icon. In the New Package dialog box, enter the name and description of the package, as shown in Figure B.

Figure B: Create a new package using the New Package dialog box.



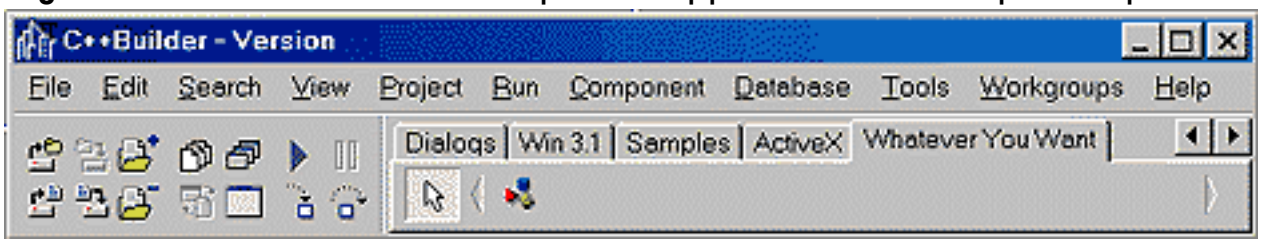
Next, you need to add the component to the package. Select Project | Add To Project and fill in the Add dialog box, as shown in Figure C.

Figure C: Add the component to the package.



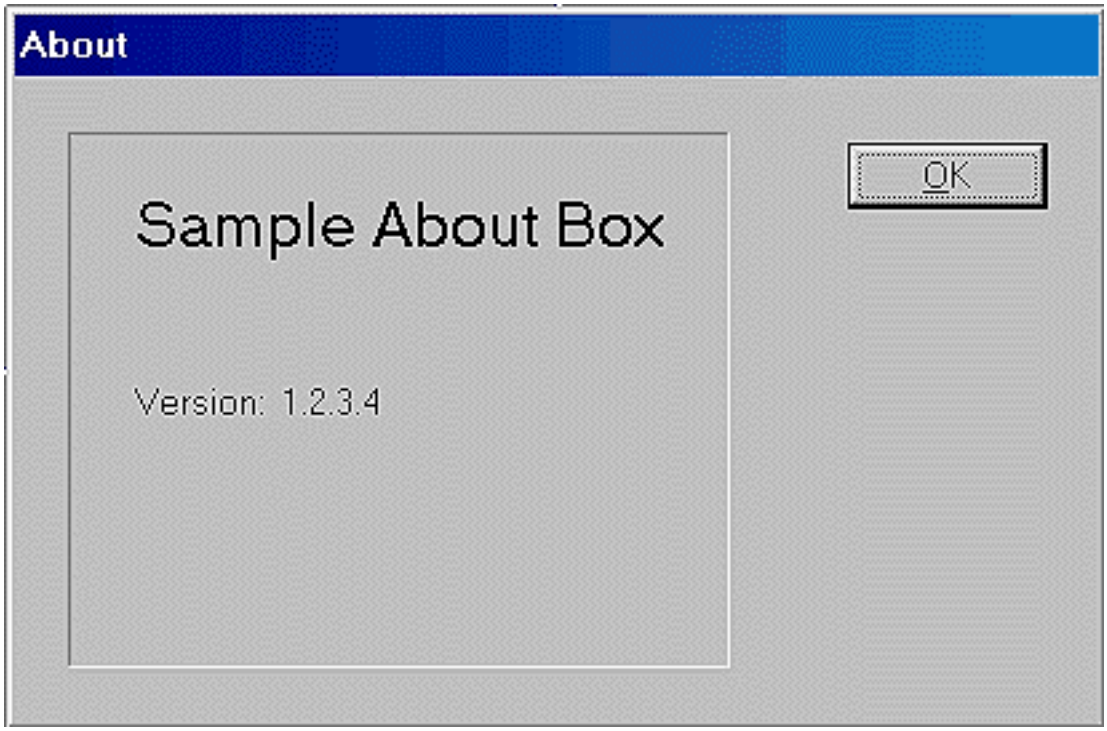
Now, rebuild your package. Then, select Component | Install Packages from the main menu and add your package to the list of installed packages. When you're finished, the TVersionInfo control should appear on the component palette, as shown in Figure D.

Figure D: The TVersionInfo component appears on the component palette.



Try creating a new project. Add a TVersionInfo control and a TLabel control to the main form. Go to the object inspector and assign the label control to the **FileVersionLabel** property and then run the program. The results appear in Figure E. Go back and select Project | Options and update the version information; then run the program again and see what happens.

Figure E: The project displays version information like this.



Final thoughts

In order to avoid duplication, this component implements properties for reading only one of the possible version-information values. If you decide to use the component in your projects, you'll probably want to expand it to read some of the other pieces of version information from the image file. Other keywords are read in exactly the same manner as **FileVersion**. Another possible extension is to support the use of multiple languages--you could add a property that would allow the user to select the language to display.

January 1999

Working with version information

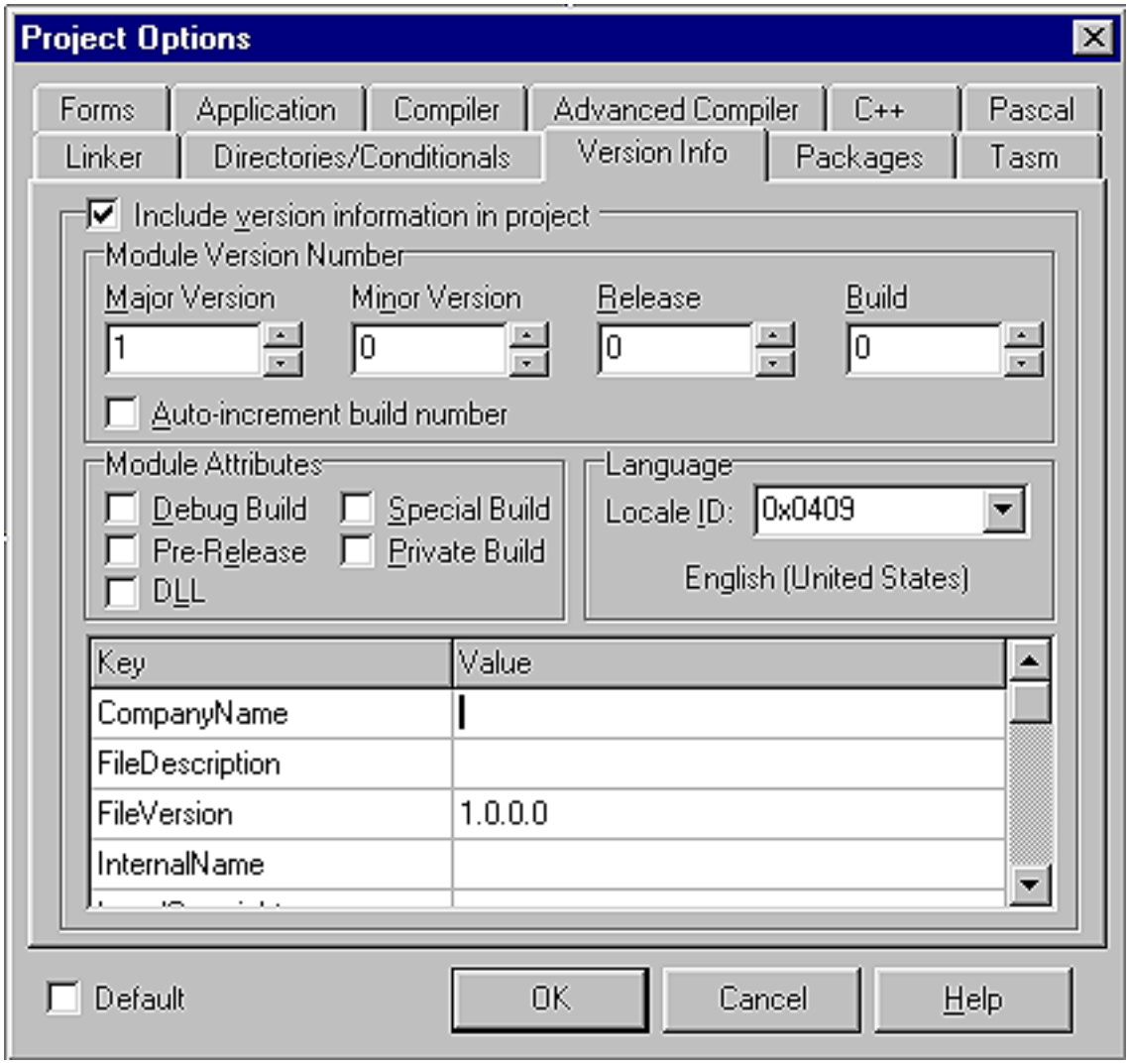
by John M. Miano

In order to understand the mechanics of our technique in the article, "A component for reading version information from a program file," you must be familiar with the way C++Builder works with component information. Let's quickly review version resources.

Creating version resources

In C++Builder, you set the version information for a program on the Version Info tab in the Project Options dialog box, as shown in Figure A. You must check the Include Version Information In Project check box in order to write the version resources to the executable file.

Figure A: Enter version information page in the Project Options dialog box.



At the bottom of the Version Info tab, you'll find a list of version keywords to which you can assign values. C++Builder automatically creates a set of predefined keywords--you should use these keywords if possible, but you can add your own keywords to this list, if you have the need.

Reading version information

The GetFileVersionInfoSize Windows API function returns the number of bytes of version information stored in the image:

```
DWORD unusedparam ;
DWORD versionsize = GetFileVersionInfoSize (
            Application->ExeName.c_str (),
            &unusedparam) ;
```

You also use this function to determine whether your application actually contains version information. The only important parameter to this function is the name of the file. The GetFileVersionInfo API function copies the version resource information to a buffer so your program can access it:

```
if (versionsize != 0)
{
  char *buffer = new char [versionsize] ;
  bool status = GetFileVersionInfo (
              Application>ExeName.c_str (),
              unusedparam,
              versionsize,
              (void *) buffer) ;
}
```

Getting the language information

To extract individual items from the version information, you use the VerQueryValue API function. A unique string key identifies each version-information item. The format of this key string is

```
\StringFileInfo\LLLLCCCC\KeyName
```

where *KeyName* is the keyword defined in the Project Options dialog box and *LLLL* and *CCCC* are hexadecimal representations of the language and character-set identifiers used. The language and character-set identifiers may seem like extra information, because C++Builder allows you to enter version information for only one language. However, a program can contain version information in multiple languages and character sets. If you have access to a full-featured Windows resource editor, you can create multi-language resource information.

The key value \VarFileInfo\Translation references a list of language translations contained in the version information. This fragment returns a pointer to an array containing information about each language contained in the version information:

```
UINT datasize ;
struct
{
```

```
  unsigned short language ;
  unsigned short character_set ;
} *translation ;
status = VerQueryValue(
          buffer,
          "\\VarFileInfo\\Translation",
          (void **) &translation,
          &datasize) ;
 int translationcount
           = datasize
           / sizeof (*translation)
```

The full key for the FileVersion keyword would be

```
String key
    = "\\StringFileInfo\\"
    + String::IntToHex (translation
      [0].language, 4)
    + String::IntToHex (translation
      [0].character_set, 4)
    + "FileVersion" ;
```

The file version is returned like this:

```
char *version ;
UNIT versionsize
status = VerQueryValue(
                  buffer,
                  key.c_str (),
                  (void **) &translation,
                  &versionsize) ;
```

January 1999

TStrings and comma-separated text

by Mark G. Wiseman

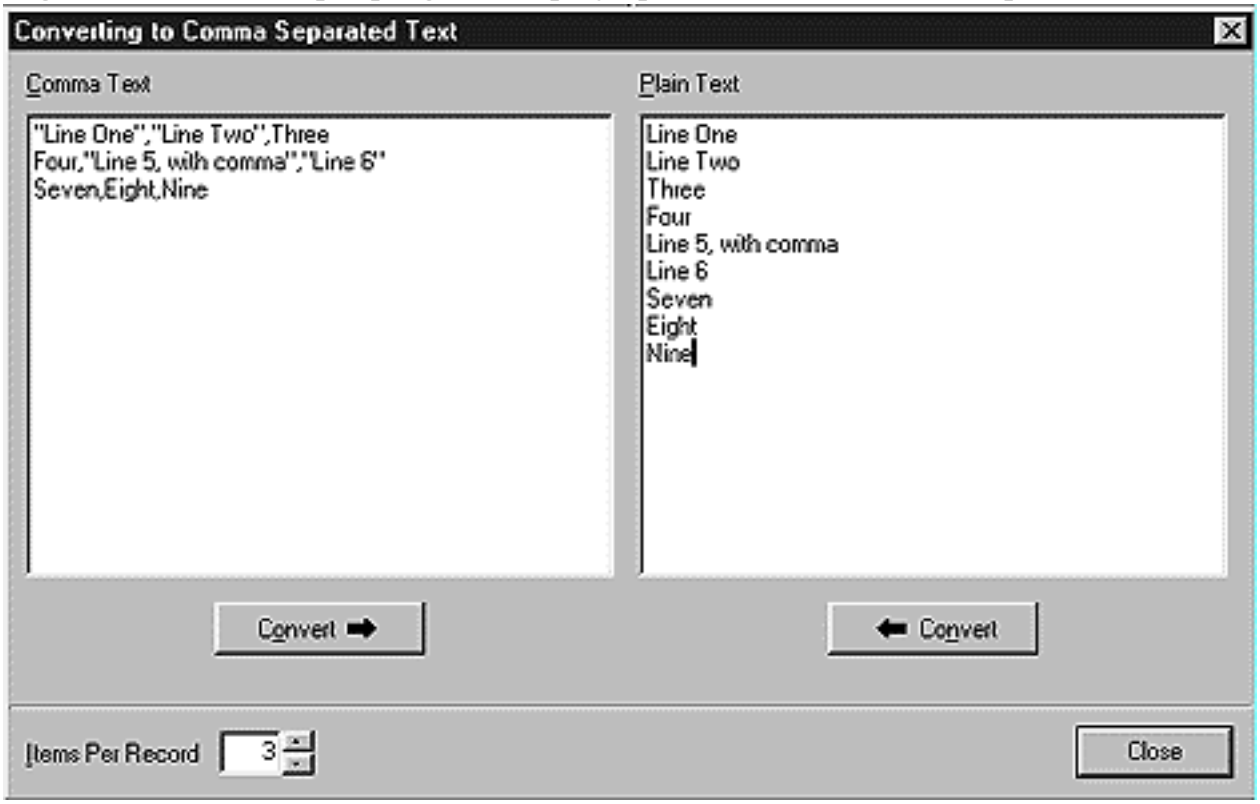
You can download sample files from our Web site as part of the file jan99.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

The VCL class TStrings is a powerful class with many features you may not be aware of. One such feature is the **CommaText** property. Using the **CommaText** property, you can convert the text in a TString object from plain text to comma-separated text and back again to plain text. In this article, we'll show you how.

Conversion functions

Figure A shows our example program. On the right is a TMemo control containing plain text. On the left is a TMemo control with the same text converted to comma-separated text. This entire program is driven by the two simple functions, ConvertToCommSep() and ConvertFromCommaSep(), as shown in Listing A and B.

Figure A: Our example program displays plain text and comma-separated text.



Listing A: Function to convert text to comma-separated text

```
void ConvertToCommaSep(TStrings *source,
   TStrings *dest, int itemsPerRecord)
   {
   if ((source->Count % itemsPerRecord) != 0)
      throw EConvertError(
             "Wrong number of items per record");

   int recordCount = source->Count /
      itemsPerRecord;

   TStringList *temp = new TStringList;
   for (int i = 0; i < recordCount; i++)
      {
      for (int j = 0; j < itemsPerRecord; j++)
      temp->Add(source->Strings[i *
      itemsPerRecord + j]);

      dest->Add(temp->CommaText);
      temp->Clear();
      }

   delete temp;
   }
```

Listing B: Function to convert back from comma-separated text

```
void ConvertFromCommaSep(Tstrings
   *source, TStrings *dest)
   {
   dest->CommaText = source->Text;
   }
```

Both functions take arguments for two pointers to objects derived from TStrings. These pointers point to a source and destination for the conversion. ConvertToCommSep() also takes an argument--itemsPerRecord--that specifies the number of items to put on each line (record) of comma-separated text.

ConvertFromCommaSep() really requires no explanation. It derives all its functionality from the **CommaText** property of TStrings.

ConvertToCommaSep() is only slightly more complicated. ConvertToCommaSep() first checks to see if the number of items to be converted is a multiple of itemsPerRecord. If it isn't, an exception is thrown. Next, ConvertToCommaSep() creates a temporary TStringList object to use while building records with the correct number of items.

ConvertToCommaSep() iterates through the source text, adding itemsPerRecord items to the temporary TStringList. Once a record contains the correct number of items, it's added to the destination TStrings object using the **CommaText** property of the temporary TStringList. This is the point at which the actual conversion takes place. Listing C shows how easily we use these two functions in our example application to perform the conversion in either direction.

Listing C: Using the conversion functions

```
void __fastcall TMainForm::ConvertFromButtonClick(
   TObject *Sender)
   {
   ConvertFromCommaSep(Memo1->Lines,
      Memo2->Lines);
   }

void __fastcall TMainForm::ConverToButtonClick(
   TObject *Sender)
   {
   Memo1->Lines->Clear();

   ConvertToCommaSep(Memo2->Lines, Memo1->Lines,
      IPREdit->Lines->Text.ToInt());
   }
```

As we comma-separate

You could write complicated parsing code to convert to and from comma-separate text. Or, you could use the Borland Database Engine. Either method would work, but either of these methods would cost you a lot of time and effort. The CommaText property of TStrings comes to the rescue with a minimum of fuss and muss. As usual, the VCL provides a simple, elegant solution to a common programming problem.

December 1998

Mastering the TTreeView component, part 2

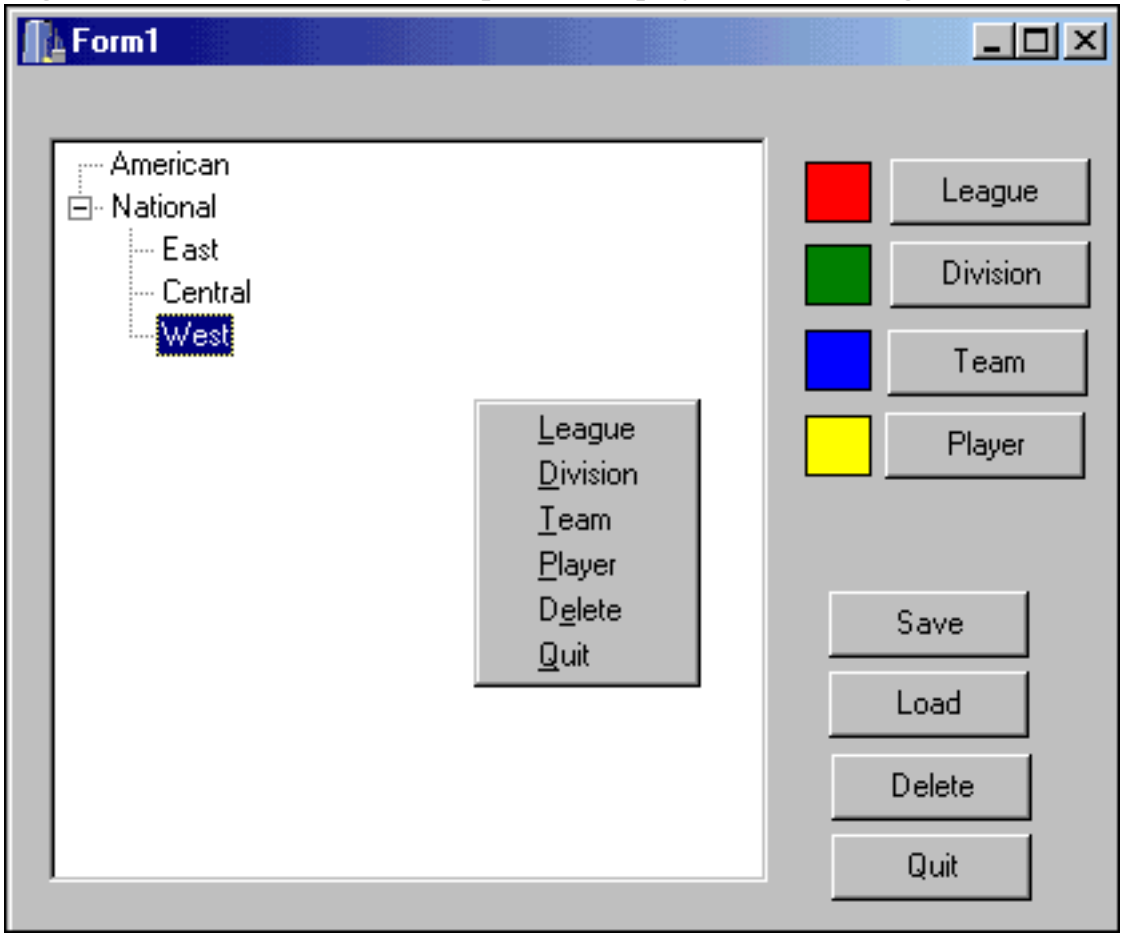
by Bill Whitney

In the November issue of *C++Builder Developer's Journal*, we showed you how to build a TTreeView component programmatically and gave you some tips for runtime navigation. This month, we'll create a similar application, but we'll start with an empty TTreeView and write the code to respond to user-generated events--including drag and drop. Finally, we'll demonstrate how you can save the information in a TTreeView so that it can be retrieved when your program runs again later.

The application

Figure A shows our sample application's main window; the complete code for the application appears in Listing A at the end of this article.

Figure A: Our TTreeView component displays baseball league information.



The TTreeView component can contain leagues, divisions, teams, and baseball players by category

(pitcher, infield, catcher, and outfield). As was true in last month's example, the root nodes represent leagues. The child of any league is a division, the child of any division is a team, the child of any team is a player category, and the child of any player category is a player.

The goal is to let the user create nodes and assign names to them. You'll implement code to build the TTreeView by responding to clicks, drag-and-drop operations, and pop-up menus.

In Figure A, you can see buttons for adding leagues, divisions, teams, and players. You'll be able to select an existing node in the TTreeView (by clicking on it with the mouse) and then click one of the buttons to add or delete an item.

The colored squares (TShape objects) to the left of the buttons will demonstrate drag-and-drop operations. Each square represents the drag-and-drop equivalent of its adjacent button; you can drag it into the TTreeView to create a new node. We'll show you not only how to tell where the square was dropped, but also how to determine whether the operation is valid within the context of the application.

Clicking buttons to add nodes

Adding leagues to the TTreeView is as simple as adding a root node. Since your application will offer several ways to add leagues (button, drag and drop, or pop-up menu), you'll place the code in a separate AddLeague method that they can all use. The League button event handler and AddLeague method appear in sections 6 and 8 of Listing A.

Within AddLeague, you should set the ActiveControl to TreeView1. If you don't, then the focus will remain on the button you just clicked (you can comment out this line to see what happens). The Add method adds a new root node (remember from part 1 of this article series that the NULL argument creates a root node). You then call the new node's EditText() method to highlight the new node's text and place the user in edit mode.

After the user edits the name and presses [Enter], the TTreeView's default behavior sets the topmost root node to the selected node. This doesn't really look right, nor is it convenient if you want to take further action on the new node. Fortunately, an Edited event is generated when any node name is updated. By adding a single line of code to this event, you can guarantee that the new node remains selected (see section 13 of Listing A.)

Adding divisions is equally simple--you just need to be sure the user has clicked on an existing league. The AddDivision and DivisionButtonClick methods are shown in sections 9 and 14 of Listing A.

Before you call AddDivision, you check for a selected node and then make sure the selected node is at the league level. Because you know the exact structure of the TTreeView, you know for sure which entries are allowed at each level. The ev_league parameter is defined in Unit1.h, as shown in section 1 of Listing A.

If your TTreeView isn't so rigidly structured, then you can't always rely on a node's level to determine its contents. In these cases, you can place a handle in the void* attribute of each node. As you'll recall from part 1, it's possible to point the void* to any object or structure. The nodeHandle structure in section 2 of Listing A contains the node's type (nodeType) and still lets the node point to any void* object (through nodeHandle's obj attribute).

To use node handles, you need to point each node's Data attribute to a valid nodeHandle structure with an appropriate nodeType value. (Part 1 provided an example of how to use handles.)

Once you're sure a league is selected--that is, when

```
TreeView1->Selected->Level == ev_league
```

you call AddDivision with a temporary new name and the selected league's node pointer. AddDivision then ensures that TreeView1 has focus and calls the AddChild method with the new division name and parent node (league) pointer. You should also call Expanded(true) on the league node to ensure that the new division is visible--otherwise, the node won't show up when the EditText call is made.

Next, you'll add a team. This is a bit more involved, because you must add the category nodes automatically when a new team is created. By now, you can guess the button event-handler code. The AddTeam code appears in section 10 of Listing A.

As usual, you set the active control and add the team node to the selected division. You then perform the expand prior to adding the categories, so that you end up with the team name preceded by a plus sign (+)--an indicator that the node has children (categories, in this case). The categories are "hidden" from view until a user clicks on the plus sign.

Adding players to the categories is as simple as selecting a category and clicking the Player button. You must first be certain that an item is selected at the category level--to see how to do so, check out the AddPlayer method in section 11
of Listing A.

Responding to the pop-up menu

In Figure A, you can see the pop-up menu at runtime. It contains the same options as the buttons on the main form. To avoid code duplication, the menu events simply call the button event handlers. Sections 7, 15, 16, and 17 of Listing A contain a few examples.

Adding drag and drop

When adding drag-and-drop capabilities, you can reuse the Add methods you created for the buttons. You must, however, determine where the additions should take place based on where the TShapes are dropped on the TTreeView. Before you write the code, be sure to set the *DragMode* property to dmAutomatic for each of the shapes--this setting will let you drag them. If you've used drag and drop in the past, then you've probably noticed that the item you're dragging changes its appearance when you drag it over a valid target. This is accomplished in the *OnDragOver* event, which determines whether the control's OnDragDrop method knows how to deal with the object you're trying to drop on it. You'll implement TTreeView's *OnDragOver* event to determine whether it's appropriate to accept the dragged TShape object. Look at the code in section 4 of Listing A.

The TreeView1DragOver method is provided with some useful arguments. You can determine the type of object being dragged by looking at the value of Source--in this code, you check to see whether it's a TShape object. If it is, then you set Accept to true; this indicates that dropping the object here is a valid action. Setting Accept to true also changes the object's appearance to let the user know it can be dropped.

You can use the X and Y values to figure out exactly where you are in the TTreeView and determine whether a particular drop operation is allowed. For example, you can drop leagues, divisions, teams, and players onto the TTreeView (because it can contain them all). However, you shouldn't be able to drop a player on a league, division, or team--players must go in a category. You can impose this behavior in either OnDragOver or OnDragDrop.

The code in sections 5a and 5b of Listing A determines whether a new division request was dropped in the correct location. The GetNodeAt method retrieves the node under the cursor at the time of the drop. From the node's level (or by identification information stored in the handle pointed to by the Data attribute's void*), you can discern whether the operation should be allowed. You do this by checking to see if the source is a DivisionShape. Since you know that the only place you can drop a division is on a league, you'll check to see if the target node's Level is ev_league. If the drop is valid, you call the AddDivision method you wrote earlier.

The example program checks the validity of the drop in the OnDragDrop method. Essentially, the OnDragOver method accepts any TShape you drag onto the TTreeView. For practice, you can try moving the GetNodeAt(X, Y) method call into the OnDragOver method. If the target node isn't valid for the operation (dropping a player on a league, for instance), you can simply set Accept to false.

Drag and drop within the TTreeView

You can also drag and drop objects within the TTreeView to relocate them. For example, you

might move a player from one team to another. Again, see section 4 of Listing A; the OnDragOver method accepts an item dragged from the TTreeView (be sure to change TTreeView's DragMode attribute to dmAutomatic). If you look at the OnDragDrop method for TreeView1 (called TreeView1DragDrop) in section 5 of Listing A, you'll see that the first Source check is for TShape. Toward the bottom of that method, however, you check for a TTreeView source. Once you determine that the source is a player and the target is a category, you call MovePlayer to relocate the player's node to the new team.

Deleting nodes

The application also allows you to delete a node. Again, using the tests for node level, you can determine whether you should follow through on the request. You wouldn't, for example, let a user delete the categories listed below a team--they're static members (children) of the team mode. Section 12 of Listing A contains the delete routine that allows the user to delete all nodes with the exception of player categories.

Saving and restoring a TTreeView

The TTreeView component comes with methods to save TTreeView nodes to a file (SaveToFile) and reload them again later (LoadFromFile). Each method takes an AnsiString identifying the name of the file. Here's how they would look in your program:

```
void __fastcall TForm1::SaveClick(
  TObject *Sender)
{
  TreeView1->SaveToFile("saved.ttv");
}

void __fastcall TForm1::LoadClick(
  TObject *Sender)
{
  TreeView1->LoadFromFile("saved.ttv");
}
```

The only problem with SavetoFile and LoadFromFile is that when the information is saved, nothing is preserved but the hierarchy and node names. If a node was pointing to an object, that information is lost when the nodes are reloaded. Unfortunately, there's no single solution to the problem--especially if a node points to an object instance. Writing and reading objects from a file requires special consideration and is beyond the scope of this article. If you must attach something to the void* of a TTreeNode, it's best to use structures, for the time being. If you need to use objects, copy the values of the object's attributes to structure members and save the structures to disk, instead.

Our example implementation works for this application. As you write the TTreeView to disk, you'll preserve information about each node, such as its level and name. If you wish to follow along, please refer to the SaveClick and LoadClick methods in sections 18 and 19 of Listing A.

The code captures each node's name, its level, and whether the node's data pointer points to something. You'll save this information in a structure called SaveNode, shown in section 3 of Listing A.

As you write nodes to disk, you walk the items in the TTreeView in their absolute order. You copy the name and level (stored in type) and set the hasStruct flag if you're using the Data attribute (void*). (Remember, the sample application is using the Data attribute only for player nodes--so, only those nodes will have this flag set.) You then write the structure to a file. If the node is a player, you write the structure the player is pointing to right behind the SaveNode structure. You continue this activity until there are no more nodes and then close the file.

Reading the nodes back in is a bit more complicated, because you must re-insert nodes into the correct levels in the TTreeView you're building. You must also check the hasStruct flag in SaveNode to be certain you read any player structures stored behind the player nodes. The while loop reads each saved node and figures out where it should be added; to do so, it uses an if/then structure adapted from the VCL Object Pascal source upon which LoadFromFile is based.

Listing A: Sample TTreeView application

```
//---------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include <Menus.hpp>


//---------------------------------
enum entryValue {ev_league = 0, ev_division,
ev_team, ev_categories, ev_player};
```

```
struct nodeHandle
{
  entryValue nodeType;
  void* obj;
};

struct PlayerStr
{
  int number;
  int age;
  float weight;
  int height;
};

struct SaveNode

{
  char name[64];
  int type;
  int hasStruct;
};

//---------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
  TTreeView *TreeView1;
  TImageList *ImageList1;
  TShape *DivisionShape;
  TShape *LeagueShape;
  TShape *TeamShape;
  TShape *PlayerShape;
  TButton *DeleteButton;
  TButton *QuitButton;
  TButton *LeagueButton;
  TButton *DivisionButton;
  TButton *TeamButton;
  TButton *PlayerButton;
  TPopupMenu *PopupMenu1;
  TMenuItem *NewLeague1;
  TMenuItem *Division1;
  TMenuItem *Team1;
  TMenuItem *Player1;
```

```
  TMenuItem *Delete1;
  TMenuItem *Quit1;
  TButton *Load;
  TButton *Save;
  void __fastcall QuitButtonClick(Tobject *Sender);
  void __fastcall TreeView1DragOver(TObject *Sender,
         TObject *Source, int X, int Y, TDragState State,
         bool &Accept);
  void __fastcall TreeView1DragDrop(TObject *Sender, TObject
         *Source, int X, int Y);
  void __fastcall LeagueButtonClick(TObject *Sender);
  void __fastcall NewLeague1Click(TObject *Sender);
  void __fastcall DeleteButtonClick(TObject *Sender);
  void __fastcall TreeView1Change(TObject *Sender,
          TTreeNode *Node);
  void __fastcall TreeView1Edited(TObject *Sender,
         TTreeNode *Node, AnsiString &S);
  void __fastcall DivisionButtonClick(TObject *Sender);
  void __fastcall TeamButtonClick(TObject *Sender);
  void __fastcall PlayerButtonClick(TObject *Sender);
  void __fastcall Division1Click(TObject *Sender);
  void __fastcall Team1Click(TObject *Sender);
  void __fastcall Player1Click(TObject *Sender);
  void __fastcall Delete1Click(TObject *Sender);
  void __fastcall Quit1Click(TObject *Sender);
  void __fastcall SaveClick(TObject *Sender);
  void __fastcall LoadClick(TObject *Sender);
private:  // User declarations
  void AddLeague(char* name);
  void AddDivision(char* name, TTreeNode* leagueNode);
  void AddTeam(char* name, TTreeNode* divisionNode);
  void AddPlayer(char* name, TTreeNode* categoryNode);
  void MovePlayer(TTreeNode* player,    TTreeNode* newTeam);
public:    // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------
#endif

//---------------------------------
#include <vcl.h>
```

```
#pragma hdrstop

#include <iostream>
#include <fstream>

#include "Unit1.h"
//---------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------

void __fastcall TForm1::QuitButtonClick(TObject *Sender)
{
    Close();
}
//---------------------------------

void __fastcall TForm1::TreeView1DragOver(TObject *Sender,
  TObject *Source, int X, int Y, TDragState State, bool &Accept)
{

  if ( (Source->InheritsFrom(__classid(TShape))) ||
       (Source->InheritsFrom(__classid(TTreeView))))
     Accept = true;
   else
     Accept = false;
}
//---------------------------------

void __fastcall TForm1::TreeView1DragDrop(TObject *Sender,

  TObject *Source, int X, int Y)
{
  if (Source->InheritsFrom(__classid(TShape)))
  {
    TTreeNode *target = TreeView1>GetNodeAt(X, Y);
    if ( Source == LeagueShape )
    {
```

```
    if ( target == NULL )
      AddLeague("(New League)");
    else
      Application->MessageBox(
      "Leagues must be dropped in dead space",
      "OOPS!", MB_OK);
  }
  else if ( Source == DivisionShape )
  {
    if ( (target != NULL) && (target->Level ==
       ev_league) )
      AddDivision("(New Division)", target);
    else
      Application->MessageBox(
      "Divisions must be dropped in Leagues",
       "OOPS!", MB_OK);
  }
  else if ( Source == TeamShape )
  {
    if ( (target != NULL) && (target->Level ==
      ev_division) )

      AddTeam("(New Team)", target);
    else
      Application->MessageBox(
      "Teams must be dropped in Divisions", "OOPS!", MB_OK);
  }
  else if ( Source == PlayerShape )
  {
    if ( (target != NULL) && (target->Level ==
      ev_categories) )
      AddPlayer("(New Player)", target);
    else
      Application->MessageBox(
      "Players must be dropped in categories", "OOPS!", MB_OK);
  }
}
else if (Source->InheritsFrom(__classid(TTreeView)))
{
  TTreeNode *target = TreeView1->GetNodeAt(X, Y);
  TTreeNode *source = (TTreeNode*) TreeView1->Selected;

  if ( source->Level != ev_player )
```

```
    {
      Application->MessageBox(
     "Only a player can be dragged and dropped",
        "OOPS!", MB_OK);
    }
    else
    {
      if ( target->Level == ev_categories )
      {
        if ( (target != NULL) )
          MovePlayer(source, target);
      }
      else
      {
        Application->MessageBox(
        "A player can only be dropped on a category",
        "OOPS!", MB_OK);
      }
    }
  }
}
//---------------------------------

void TForm1::MovePlayer(TTreeNode* player,
        TTreeNode* newTeam)
{
  TTreeNode* n = TreeView1->Items->AddChild
        (newTeam, "x");
  n->Assign(player);
  player->Delete();
}
//---------------------------------


void __fastcall TForm1::LeagueButtonClick
        (TObject *Sender)
{
  AddLeague("(New League)");
}
//---------------------------------


void __fastcall TForm1::NewLeague1Click
        (TObject *Sender)
```

```
{
  LeagueButtonClick(Sender);
}
//---------------------------------

void TForm1::AddLeague(char* name)

{
  ActiveControl = TreeView1;
  TTreeNode* n = TreeView1->Items->Add
        (NULL, name);
  n->EditText();
}
//---------------------------------

void TForm1::AddDivision(char* name, TTreeNode*
         leagueNode)
{
  ActiveControl = TreeView1;
  TTreeNode* n = TreeView1->Items->AddChild
        (leagueNode, name);

  leagueNode->Expand(true);
  n->EditText();
}
//---------------------------------

void TForm1::AddTeam(char* name, TTreeNode*
        divisionNode)
{
  ActiveControl = TreeView1;
  TTreeNode* n = TreeView1->Items->AddChild
        (divisionNode, name);
  divisionNode->Expand(true);

  TreeView1->Items->AddChild(n, "Pitchers");
  TreeView1->Items->AddChild(n, "Catchers");
  TreeView1->Items->AddChild(n, "Infielders");
  TreeView1->Items->AddChild(n, "Outfielders");
  n->EditText();
}
//---------------------------------

void TForm1::AddPlayer(char* name, TTreeNode* categoryNode)
```

```
{
  ActiveControl = TreeView1;
  TTreeNode *n = TreeView1->Items->AddChild
        (categoryNode, name);
  // Add a structure to the Data (void*)
  // attribute of this node
  PlayerStr *p = new PlayerStr;

  p->number = 20;
  p->age = 31;
  p->weight = 210.0;
  p->height = 74;
  n->Data = (void*) p;
  categoryNode->Expand(true);
  n->EditText();
}
//---------------------------------

void __fastcall TForm1::DeleteButtonClick(TObject *Sender)
{
  // Delete a selected node.
  ActiveControl = TreeView1;
  if ( TreeView1->Selected == NULL )
  {
    Application->MessageBox("You must select a node to delete",
                            "OOPS!", MB_OK);
  }
  else
  {
    switch ( TreeView1->Selected->Level )
    {
      case ev_categories :
        Application->MessageBox("You can't delete a category!",

                                "OOPS!", MB_OK);
        break;
      default :
        TreeView1->Items->Delete(TreeView1->Selected);
    }
  }
}
//---------------------------------
```

```
void __fastcall TForm1::TreeView1Edited(TObject *Sender,
  TTreeNode *Node, AnsiString &S)
{
  Node->Selected = true;
}
//---------------------------------

void __fastcall TForm1::DivisionButtonClick(TObject *Sender)
{

  if ((TreeView1->Selected != NULL) &&
      (TreeView1->Selected->Level == ev_league))
  {
    AddDivision("(New Division)", TreeView1->Selected);
  }
  else
  {
    Application->MessageBox("Divisions are added to Leagues!",
                            "OOPS!", MB_OK);
  }
}
//---------------------------------

void __fastcall TForm1::TeamButtonClick(TObject *Sender)
{
  if ( (TreeView1->Selected != NULL) &&
       (TreeView1->Selected->Level == ev_division) )
  {
    AddTeam("(New Team)", TreeView1->Selected);
  }
  else
  {
    Application->MessageBox("Teams are added to Divisions!",
                            "OOPS!", MB_OK);
  }
}
//---------------------------------

void __fastcall TForm1::PlayerButtonClick(TObject *Sender)
{
  if ( (TreeView1->Selected != NULL) &&
       (TreeView1->Selected->Level == ev_categories) )
  {
    AddPlayer("(Player Name)", TreeView1->Selected);
```

```
  }
  else
  {
    Application->MessageBox("Players are added to categories!",
                            "OOPS!", MB_OK);
  }
}
//---------------------------------


void __fastcall TForm1::Division1Click(TObject *Sender)
{
  DivisionButtonClick(Sender);
}
//---------------------------------


void __fastcall TForm1::Team1Click(TObject *Sender)
{
  TeamButtonClick(Sender);
}
//---------------------------------


void __fastcall TForm1::Player1Click(TObject *Sender)
{
  PlayerButtonClick(Sender);
}
//---------------------------------

void __fastcall TForm1::Delete1Click(TObject *Sender)
{
  DeleteButtonClick(Sender);
}
//---------------------------------

void __fastcall TForm1::Quit1Click(TObject *Sender)
{
  QuitButtonClick(Sender);
}
//---------------------------------

void __fastcall TForm1::SaveClick(TObject *Sender)
{
  ofstream   nodeFile;
```

```
  TTreeNode* curNode;
  SaveNode   savNode;
  PlayerStr  plyStruct;

  nodeFile.open("saved.ttv", ios::binary | ios::trunc);
  for ( int a = 0; a < TreeView1->Items->Count; a++ )
  {
    curNode = TreeView1->Items->Item[a];
    strcpy(savNode.name, curNode->Text.c_str());
    savNode.type = curNode->Level;
    if ( curNode->Level == ev_player )
      savNode.hasStruct = 1;
    else
      savNode.hasStruct = 0;
    nodeFile.write((unsigned char*)&savNode, sizeof
        (SaveNode));
    if ( curNode->Level == ev_player )
    {

      memcpy(&plyStruct, (PlayerStr*)curNode->Data,
             sizeof(PlayerStr));
      nodeFile.write((unsigned char*)
                      &plyStruct, sizeof(PlayerStr));
    }
  }
}
//---------------------------------

void __fastcall TForm1::LoadClick(TObject *Sender)
{
  ifstream   nodeFile;
  TTreeNode* curNode;
  TTreeNode* nxtNode;
  SaveNode   savNode;
  PlayerStr  plyStruct;
  PlayerStr* pStr;
  entryValue curLvl;

  nodeFile.open("saved.ttv", ios::binary);
  curNode = (TTreeNode*) NULL;
  while (nodeFile.read((unsigned char*)
         &savNode, sizeof(SaveNode)))
  {
    curLvl = savNode.type;
```

```
    if ( curNode == NULL )
    {
      curNode = TreeView1->Items->AddChild
        (NULL, savNode.name);
    }
    else if ( curNode->Level == curLvl )
    {
      curNode = TreeView1->Items->AddChild
        (curNode->Parent,
                savNode.name);
    }
    else if ( curNode->Level == (curLvl - 1) )
    {
      curNode = TreeView1->Items->AddChild
        (curNode,
                savNode.name);
    }
    else if ( curNode->Level > curLvl )
    {
      nxtNode = curNode->Parent;
      while ( nxtNode->Level > curLvl )
      {
        nxtNode = nxtNode->Parent;
      }
      curNode = TreeView1->Items->AddChild
        (nxtNode->Parent,
                savNode.name);

    }
    if ( savNode.type == ev_player )
    {
      nodeFile.read((unsigned char*)
                    &plyStruct, sizeof(PlayerStr));
      pStr = new PlayerStr;
      pStr->number = plyStruct.number;
      pStr->age = plyStruct.age;
      pStr->weight = plyStruct.weight;
      pStr->height = plyStruct.height;
      curNode->Data = (void*) pStr;
    }
  }
}
```

Conclusion

TTreeView is a powerful visual component. Unfortunately, it's often difficult to work with. In this article, we've given you some tools that will help you take advantage of some of the TTreeView's capabilities.

December 1998

Creating a URL label

C++Builder programming
version markers: 1.0, 3.0

by Kent Reisdorph

You can download sample files from our Web site as part of the file dec98.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

A Web site is a great way to promote your company. If you have a Web site, you want your users to be able to get to it with the minimum amount of effort. A URL label in your application's About dialog box allows your users to quickly and easily connect to your Web site from your application. In this article, we'll show you how you can create a URL label.

What's a URL label?

A URL label is nothing more than a regular Label component that, when clicked, starts a Web browser and connects to a Web site. A URL label has the following characteristics:

- The label's caption is a URL.
- The URL appears in blue text, to indicate that it can be clicked.
- Your cursor changes to a pointing hand when it moves over the label.
- When you click on the label, it spawns a Web browser.

For example, Figure A shows the URL label in our sample application's About dialog box.

Figure A: Users can click on a URL label to quickly go to a Web site.

Most of the steps required to fulfill this feature list are easy to perform--you certainly don't need any help to drop a label on a form, change the *Caption* property, and change the font's color. The rest, though, requires some explanation.

Creating and implementing the cursor

A URL label should make the cursor change to a pointing hand, so the user can tell that the label is more than just colored text. You can easily create a pointing hand cursor using the Image Editor (or any other resource editor you're familiar with).

Using Image Editor, create a resource file (RES) and then create a cursor resource within that resource file. Name the cursor resource something appropriate (*WEBPOINTER* would be a good name). Save the resource file and close Image Editor. Now, choose C++Builder's Project | Add To Project menu item and add the resource file to your project.

Next, you need to load the cursor resource so that it's available to the URL label. Put this code in your main form's OnCreate event handler:

```
Screen->Cursors[1] =
  LoadCursor(HInstance, "WEBPOINTER");
```

The cursor will be loaded when the application starts and will be ready for you to use. Notice that the code loads the cursor into array index 1 of the Cursors array. To assign the new cursor to the URL label, simply change the label's *Cursor* property to 1. You can do this either at design time or at runtime.

Starting the Web browser

At this point, you have a label that looks good but doesn't do anything. You need to write code so that the Web browser starts when you click on the URL label. If you think this step is going to be difficult, you might be surprised at how easy it actually is. The key is the Win32 API ShellExecute function. Before you can use ShellExecute, you need to include the SHELLAPI.H header in your application. Place this line near the top of the header for any units that call ShellExecute:

```
#include <ShellApi.h>
```

Now that you've included the header, you can call ShellExecute. Here's the code:

```
ShellExecute(0, "open",
  "http://www.turbopower.com", "", "",
  SW_SHOWNORMAL);
```

ShellExecute will open a document based on the document's extension. In order for this function to work, the filename extension must be registered with Windows. A browser document is a special case: Windows recognizes the document name as a URL, automatically executes the default Web browser, and loads the document. If ShellExecute fails, it returns a value of 32 or fewer. You should write your code to account for an error. For example:

```
int result =(int)ShellExecute(0, "open",
  "http://www.turbopower.com", "", "",
  SW_SHOWNORMAL);
if (result <= 32)
  ShowMessage("Unable to start web browser.");
```

All you have to do now is add this code to the *OnClick* event handler for the URL label. Double-click on the label, and C++Builder will create an event handler for the *OnClick* event. Enter code similar to the above code snippet, and you're all set.

But wait, there's more!

Not only can you start a Web browser with ShellExecute, you can also use it to let your users send you E-mail. It's just a matter of changing things a little:

```
int result =(int)ShellExecute(0, "open",
  "mailto:freddy@kreuger.com", "", "",
  SW_SHOWNORMAL);
if (result <= 32)
  ShowMessage(
    "Unable to start mail client.");
```

Notice that the document string in this example includes *mailto:* rather than a URL. When this code executes, Windows will start the mail client registered on the user's system and fill in the To field with the E-mail address passed in ShellExecute.

Conclusion

Allowing your users quick access to you via the Internet is a great way to increase your business. In this article, we've shown you how to let users easily access your Web site from an application's About dialog box.

December 1998

WriteLn for C++Builder

C++Builder programming
version markers: 1.0, 3.0

by Kent Reisdorph

You can download sample files from our Web site as part of the file dec98.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

Delphi programmers have used the WriteLn function as a debugging tool for a long time. Text passed to WriteLn is sent to a console window, thereby allowing you to use a console window as a real-time log file. By doing so, you can track your program in time-critical sections of code where a breakpoint won't work. C++Builder doesn't include a WriteLn function. It's easy enough, though, to create your own WriteLn function. In this article, we'll explain how to do that. As part of our explanation, we'll examine the AllocConsole, GetStdHandle, and WriteConsole Windows API functions.

WriteLn the Delphi way

Using the WriteLn function in Delphi requires two steps: You allocate a console window for the application and then call WriteLn. The code looks something like this:

```
AllocConsole();
WriteLn(`Entering Critical Section');
{ Some processing here }
WriteLn(`Exiting Critical Section');
```

AllocConsole is a Win32 API function; as its name implies, it allocates a console window for the application. Any application can have a console window--but it can have only one. This is true for GUI applications, as well as traditional console applications.

After the console window is allocated, the WriteLn function displays text in the window. The system is simple but extremely effective. In fact, in some debugging situations, nothing else will do.

Console output in C++

C++Builder has debugging tools similar to WriteLn. The diagnostic macros TRACE and WARN and the Windows API function OutputDebugString perform tasks that are roughly equivalent to what WriteLn does. These methods, however, suffer from one major drawback--their output goes to a log file that isn't updated until the application either stops at a breakpoint or is terminated. That limitation makes these debugging tools of limited value in some situations.

The good news is that Delphi's WriteLn function can be easily duplicated in C++Builder. In fact, you can go one better by including the AllocConsole call within the WriteLn function. You write text to the console window with the WriteConsole function. This function requires a handle to the output buffer for the console window; you can obtain that handle by calling the GetStdHandle function. The whole thing looks something like this:

```
AllocConsole();
HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);
WriteConsole(handle, "Test", 4, 0, 0);
```

Let's examine this code in more detail. Naturally, the AllocConsole line allocates the console window. In the second line, the GetStdHandle function retrieves a handle to the standard output buffer. You pass STD_OUTPUT_HANDLE to this function to tell Windows to give you the handle of the standard output buffer. Other possible values for this parameter are STD_INPUT_HANDLE and STD_ERROR_HANDLE, but you aren't concerned with those handles for this operation.

Once you have the handle to the standard output buffer, you can call the WriteConsole function to send the text to the console window. The first parameter of WriteConsole is the handle to the standard output buffer, the second parameter is the text to send to the console window, and the third parameter is the number of characters to display. (The fourth parameter returns the number of characters displayed and the fifth parameter is a reserved, unused parameter that's unused; you don't need to worry about either of them here.)

A WriteLn for C++Builder

Now that you know how to send text to the console window, you can go to work and create your WriteLn function. First, let's define the function prototype for the function. It's pretty basic:

```
void WriteLn(String text);
```

As you can see, the function takes an AnsiString as a parameter and returns void. Naturally, the text parameter will be used to pass the text you want sent to the console window. Now, let's look at the complete WriteLn function:

```
void WriteLn(String text)
{
  static HANDLE handle;
  if (!handle) {
    AllocConsole();
    handle = GetStdHandle(STD_OUTPUT_HANDLE);
  }
  text += "\n";
  WriteConsole(handle,
    text.c_str(), text.Length(), 0, 0);
}
```

First, you declare a static variable called handle; this variable will tell you if the console window has been allocated already. (Remember, a static variable has an initial value of 0 and retains its value between function calls.) If handle is 0, then the console window hasn't been created; so, you call AllocConsole. After that, you call GetStdHandle to obtain the handle to the standard output buffer. Next, you add a newline character to the end of the string passed in. Doing so ensures that each line written leaves the cursor in a position to start a new line. Finally, you call WriteConsole to write the text to the console window. Notice that you pass the length of the AnsiString as the third parameter of the WriteConsole function, so that all characters in the string are written to the console window.

An example

Listing A contains a program that uses the WriteLn function; this is the code for a main form containing just one button. When you click the button, a loop calls the WriteLn function to write 20 lines of text to a console window.

Listing A: WRITELNU.CPP

```
//---------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "WriteLnU.h"
//---------------------------------------
```

```
#pragma resource "*.dfm"

TForm1 *Form1;

// The WriteLn function.
void WriteLn(String text)
{
  static HANDLE handle;
  if (!handle) {
    AllocConsole();
    handle = GetStdHandle(STD_OUTPUT_HANDLE);
  }
  text += "\n";
  WriteConsole(handle,
    text.c_str(), text.Length(), 0, 0);
}

//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  for (int i=0;i<20;i++) {
    Application->ProcessMessages();
    WriteLn("Iteration: " + String(i));
    Sleep(100);
  }
}
```

Conclusion

The WriteLn function can be a great debugging tool. You may not use it often, but when you really need it, there's no substitute.

December 1998

What's in a name(space)?

by Gerry Myers

Imagine you've almost finished a program and you decide to add some nice graphics (plots and charts). No problem--you install a professional-looking graphing package and add some function calls to your program to execute the new package. Unfortunately, when you try to compile the application, you get errors that say multiple declarations were found for functions x, y, and z. After some investigation, you realize that both your code and the third-party package have x, y, and z functions. What do you do?

You have several choices (most of them bad). You can globally change all your function, variable, and class names to other, unique names. You can get the graphing package's source code and do the same to it. Or (drum roll, please), you can enclose your declarations within a *namespace*, which would effectively give all your identifiers unique names.

I programmed C and C++ for many years without using namespaces--I never had to. Only within the last couple years have I found myself putting them to work. As programming languages evolve, different constructs fall in and out of favor with the programming community. Lately, namespaces have seen a rise in popularity due in large part to the C++ Standard Template Library (STL), which uses them extensively. In this article, we'll look at how namespaces work, and explore when you should (or must!) put them to work.

What's a namespace?

Namespaces fill a niche that's hard to work around *when you encounter it*. You may never need them or you may always need them, depending on your choice of programming tools. Before we look at when and why namespaces are needed, let's briefly review what they are.

A namespace delineates a block of code in which identifiers are unique from the rest of the code. Code outside the namespace block can still reference the variables, functions, or classes inside the block, but their full identifier names must be used.

The syntax of a namespace is simple. You declare the namespace block using the namespace keyword, followed by its name and an interior block of code:

```
namespace MySpace
{
    int x;
    float SomeFnc()  { ... }
```

```
    class SomeClass { ... };
}// end of MySpace namespace block
```

There's nothing special about the code inside a namespace--it's just like any other code.

Explicit versus implicit

You may have never *explicitly* used a namespace like that shown in the previous example. However, without knowing it, you almost certainly *implicitly* use them in every program. If you declare a class or function, you're really declaring a namespace. Any identifiers declared within the class or function are local to that block. If you want to reference the identifiers from outside the class, you need to use the full identifier name, which is the class (namespace) name followed by the identifier.

Unlike class and function blocks, explicit namespaces are concerned only with global identifiers. You can't declare explicit namespaces within a class or function. However, you can nest them, as shown in Listing A, or extend/continue them, as shown in Listing B.

Listing A: Nested namespaces

```
namespace SpaceA
{
    int i;
    float f;

    namespace SpaceB
    {
        int i;
        float f;
    }  // end of SpaceB

}  // end of SpaceA
```

Listing B: Extended namespaces

```
namespace SpaceA
{
    int i;
    float f;
}  // end of SpaceA

int i;
```

```
float f;
double d;

namespace SpaceA
{
    double d;
}// end of SpaceA extension
```

Referencing namespace identifiers

You reference the declarations inside the namespace by placing the name of the namespace and the scope operator before the identifiers, like this:

```
float f = MySpace::SomeFnc() + MySpace::x;
```

If such code gets to be too tedious (if you have to make many references to identifiers in the namespace), you can declare your intent to use the namespace for all references until further notice. You do so by placing the command using namespace *xyz* before the part of your program in which you need to reference the namespace code:

```
using namespace MySpace;
. . .
float f = SomeFnc() + x;
```

This method saves you from repeatedly typing the name of the namespace. However, you may still need to use the full identifier name (including namespace) if it happens to be the same identifier used for a different purpose in another portion of the program.

Referencing only a specific identifier

If you don't want (or need) to issue a broad using namespace command, you can specify using namespace for a specific identifier. You might do this if you use only one function or class from the third-party package--but you use it throughout your code. You can issue the broad statement

```
using namespace MySpace;
```

or add MySpace:: to the front of the identifier each time you use it. But, the best method may be to place

```
using namespace MySpace::SomeFunc;
```

at the top of the module. This line will alert the compiler that any reference to SomeFunc() in that module should be mapped to MySpace::SomeFunc(). The rest of your code will be unaffected and will map to your identifiers.

Try on an alias

Another handy widget in the namespace toolbox is the *alias*. You can give an alias (nickname) to an existing namespace. For instance, you'll want to do this if a third-party package declares a namespace that's very descriptive and unique--but also very long. How would you like to write

```
XYZCOMPANY_FINACIAL_UTILITIES_NAMESPACE
```

before each reference to a package's identifiers? You can shorten such a namespace to one of your own liking by writing the following code:

```
namespace XYZ =
  XYZCOMPANY_FINACIAL_UTILITIES_NAMESPACE;
```

If the company's namespace doesn't depict the certain utilities you use from its package, give the namespace an alias that makes it easier for you to relate to. For example, if you use only a full-blown financial package for loan interest calculations, it may be easier to relate to the namespace if you alias it as follows:

```
 namespace INT_CALC = XYZCOMPANY_FINACIAL_UTILITIES_NAMESPACE;
```

You don't need namespaces if...

As we mentioned earlier, you may never need to explicitly use namespaces. (*Never* may be too strong a word--you're likely to see them crop up sooner or later.) If your projects involve

only your own code, you don't need to use namespaces. As long as you can control and track the naming of identifiers, namespaces may be more of a programmatic burden then they're worth. (This assumes that you don't purchase software packages from third-party vendors.) However, it wouldn't be a bad idea to get used to namespaces, because they can help compartmentalize your program.

You should use namespaces if...

If you purchase software utilities to be linked into and used by your project, you may (or may not) run into a problem of duplicate identifiers. Because you probably write literate code (descriptive names for classes, variables, and functions), and the people who wrote the new package you bought probably write literate code, you may end up conflicting on customary identifiers such as GetString(), SetFileName(), or even int i.

If you encounter a duplicate name, the simplest way out is to enclose all your header material within a namespace with a name that no one else would use (such as your initials or name). When the compiler goes through your code, it will prepend your namespace name to the front of all *your* identifiers when building its identifier list--for example,

```
int i;
```

will become

```
int SomeSpace::i;
```

Now, you'll have to reference the new identifier names a little differently. Simply place a using namespace command at the top of your implementation (CPP) file; all your identifier usages will be prepended with your namespace name. Presto, no more identifier conflicts!

To tell the compiler that you want to use a non-namespace identifier provided in the third-party package, you simply add the scope resolution operator (::) to the front of the third-party function call. Everything that isn't in a namespace is considered global and can be accessed this way.

Tip: Save your users the same trouble

You should add explicit namespaces to your programs and utilities if other people will be using them. If you're producing a third-party product, using namespaces will protect the unsuspecting user from the dilemma of duplicate identifiers.

You must use namespaces if...

There are times when you just can't get around using namespaces. If a third-party software vendor encloses its identifier declarations within a namespace, you have no choice but to use its namespace (or an alias of your choosing) to get to its functions and classes. You can do so using any of the methods we've mentioned.

The C++ STL is gaining popularity lately, and guess what? It's enclosed in a namespace called std. If you want to use any of the templates in the STL, you must become at least vaguely familiar with using namespace. The easiest way to reference the STL identifiers (vectors, lists, deques, and so on) is to place using namespace std; at the top of your module--a pretty painless task.

Conclusion

You may never have needed (or even heard about) namespaces before, but that's about to change. Increasingly, third-party software vendors are using namespaces, which means that you *must* reference their utilities through that namespace. The C++ Standard Template Library is fast becoming a tool of choice for many software developers, a fact that also forces them to learn about namespaces. Namespaces add a little complexity to your code, but they're worth it--take time to become familiar with this construct and begin putting it to work to improve your applications.

December 1998

Horizontal scrollbars for list boxes

C++Builder programming
version markers: 1.0, 3.0

by Kent Reisdorph

It isn't uncommon to have text that's too long to fit the width of a list box. In such a case, you need a horizontal scrollbar so your users can see the rest of the text. The vertical scrollbar appears automatically in a list box when the number of items in a list box exceeds the height of the list box. However, you have to do some work to display a horizontal scrollbar.

Adding a horizontal scrollbar to a list box requires two steps: determining the width of the text (in pixels) of the widest item in the list box and sending a Windows message to the list box to add the horizontal scrollbar. In this article, we'll show you how to add a horizontal scrollbar to a list box. In the process, you'll learn about the TextWidth method of the TCanvas class and Windows' LB_SETHORIZONTALEXTENT message.

How wide is wide?

When you create a horizontal scrollbar, you tell Windows how far to scroll. Before you can do that, you need to determine the width of the widest item in the list box. VCL makes this part easy by providing the TextWidth method of the TCanvas class. This method will give you the width, in pixels, of a string of text; the value is calculated using the current font for the canvas. Here's an example:

```
int x = ListBox1->Canvas->
  TextWidth("Hello!");
```

Now, it's a simple matter of checking all of the items in the list box to see which is the longest. Depending on how you fill your list box, you could check the length as you add items or later, after all the items have been added. To check the length of the items after the items have been added, you can use code like this:

```
int length = 0;
for (int i=0;i<ListBox1->Items->Count;i++)
  {
```

```
  String text = ListBox1->Items->
    Strings[i];
  int l = ListBox1->Canvas->
    TextWidth(text);
  if (l > length) length = l;
}
```

Now, you have the width, in pixels, of the longest item in the list box. Note that if your list box has a large number of items, this may not be the most efficient method of determining the longest item's length--it should be good enough for most cases, though.

Adding the scrollbar

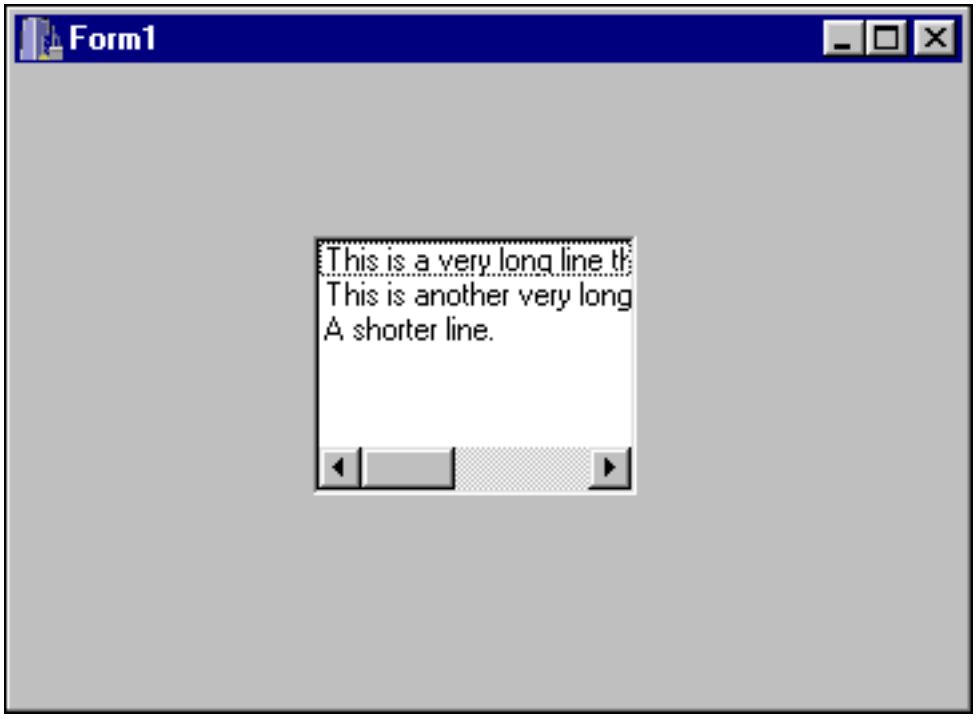
Now that you have the length of the longest item in the list box, you can tell Windows to add a horizontal scrollbar. You do so with the Windows message LB_SETHORIZONTALEXTENT. It's as simple as this:

```
SendMessage(ListBox1->Handle,
  LB_SETHORIZONTALEXTENT, length, 0);
```

That's all there is to it! When this message is sent, Windows will add a horizontal scrollbar to the list box. As you can see, the scrollbar's horizontal extent is passed in the wParam of the LB_SETHORIZONTALEXTENT message. If the list box's width is already wider than the value of wParam, then the scrollbar won't be added. However, if the list box's width is reduced after sending the LB_SETHORIZONTALEXTENT message, then the scrollbar will appear.

To test this theory, start a new project in C++Builder and drop a ListBox component on the form. Double-click on the form's background to generate an event handler for the *OnCreate* event. Now, enter the code from Listing A in the event handler you just created. When you run the program, the list box will display a horizontal scrollbar, as shown in Figure A.

Figure A: This list box displays a horizontal scrollbar.



Listing A: Adding a Horizontal Scrollbar

```
void __fastcall
TForm1::FormCreate(TObject *Sender)
{
  // Put some items in the list box.
  ListBox1->Items->Add("This is a very long "
    "line that is too wide for the list box.");
  ListBox1->Items->Add("This is another very long line "
    "that is too wide for the listbox.");
  ListBox1->Items->Add("A shorter line.");

  // Find the length of the longest item.
  int length = 0;
  for (int i=0;i<ListBox1->Items->Count;i++) {
    String text = ListBox1->Items->Strings[i];
    int l = ListBox1->Canvas->TextWidth(text);
    if (l > length) length = l;
  }

  // Add a little for a good visual effect.
  length += 10;

  // Tell Windows to create the scrollbar.
  SendMessage(ListBox1->Handle,
    LB_SETHORIZONTALEXTENT, length, 0);
```

```
}
```

November 1998

Mastering TTreeView component, part 1

by Bill Whitney

You can download our sample files as part of the file nov98.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

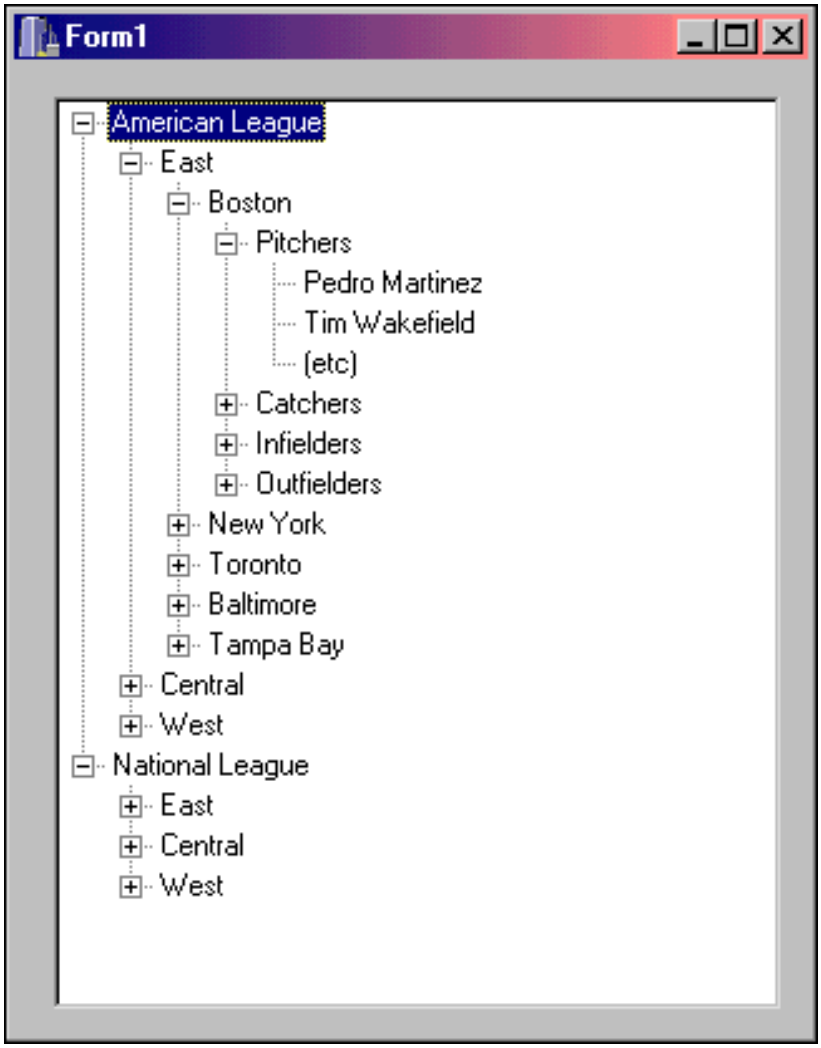
This article is the first in a two-part series covering the TTreeView component. If you aren't familiar with what this component can do for you, you can see its behavior by experimenting with the left pane in the Windows Explorer application. In this pane, the TTreeView represents system elements, such as disk drives, the Recycle Bin, the Printers folder, and the Control Panel.

While TTreeView is a powerful tool for visualizing hierarchical relationships, it's somewhat difficult to take advantage of as a data structure. We'll try to take the pain out of implementing TTreeView by looking at its basic elements and discussing their relationships. Then, we'll examine the methods and attributes used to build and maneuver within a TTreeView.

Defining TTreeView

TTreeView is a container, which means you can use it to store and manipulate other objects. The primary difference between TTreeView and most other containers is its hierarchical arrangement. Figure A shows a partially expanded TTreeView containing all the leagues, divisions, teams, and players in Major League Baseball. Throughout this article, we'll refer back to Figure A.

Figure A: Our sample TTreeView displays information about Major League Baseball.



Basic relationships

TTreeView wouldn't exist without TTreeNodes and TTreeNode. TTreeView maintains a visual representation of the individual TTreeNode objects and handles events affecting that representation. The *Items* property of TTreeView is of type TTreeNodes, and it maintains the list of individual TTreeNode objects contained in a TTreeView. You'll work directly with TTreeView to handle events, with TTreeNodes to insert and delete nodes, and with TTreeNode to delete nodes and determine relationships between individual nodes.

The heart of the matter: TTreeNode

If you've worked with data structures such as linked lists or balanced trees in the past, then TTreeNode shouldn't be a mystery. It serves as the basic element (node) in a TTreeView container. In Figure A, each league, division, team, position, and player is represented by its own individual TTreeNode. TTreeNode has two very useful properties that we'll cover in more detail as we go. The first is a Text attribute that identifies the node in a TTreeView (its name). The second is a void* that can point to any object you choose (a structure, an object, or even another character string).

TTreeNode relationships

Since TTreeView is hierarchical, an individual TTreeNode can have more than one type of relationship to other nodes. It can be a parent, a child, or a sibling (or, in many cases, all three). The parent/child relationship is best described by a look back at Figure A. East, Central, and West are children of the American League node--children are indented immediately below the parent. Since Boston is a child of East, East is the parent to Boston. Notice we say "the" parent--a node can have only one parent. When one or more children have the same parent, they are siblings.

Root nodes and node levels

Every TTreeView implementation will contain at least one *root node*. The Windows 98 Explorer, for example, shows Desktop as the solitary root node. Figure A contains two root nodes: American League and National League. The number of root nodes is entirely up to you, and should reflect the needs of your application. A node's *level* reflects its distance from the root node (where the root level is 0). For instance, you can find the names of the members of Boston's pitching staff on level 4.

Selected nodes

The *selected* node is the node in relation to which all adding and deleting will take place (it really makes sense only if you're looking at TTreeView). If you allow users to insert and delete based on the node selections they make (as you'll see later , you can use the TTreeView *Selected* property), then your job is a little easier. If you're building a TTreeView programmatically, you either must know where you are at all times or how to select the correct node when necessary. In the programmatic sense, *selected node* refers to the existing node you pass as an argument to insert and delete operations. Let's take a look at the methods you use to insert and delete nodes.

Programming TTreeView

Regardless of the eventual breadth and depth of your tree, you must start by inserting one or more root nodes. To do so, you can use one of two methods: Add or AddFirst. Both methods take a pointer to an existing TTreeNode and an AnsiString as arguments, and both return a pointer to the newly created TTreeNode. Here's a look at Add's signature:

```
TTreeNode* __fastcall Add(TTreeNode* Node,
  const System::AnsiString S);
```

Any time the Node parameter is NULL, you add the node to level 0 (a root node). The AnsiString argument is simply the text that will appear in the TTreeView representation of the new node (such as *Boston*). Add appends the new node to the end of the current group of siblings, whereas AddFirst inserts it at the beginning. Here's some code to insert the two root nodes shown in Figure A:

```
TreeView1->Items->Add(NULL, "American League");
TreeView1->Items->Add(NULL, "National League");
```

All the methods that add nodes to the TTreeView take the same arguments as Add--an existing

TTreeNode and an AnsiString. Later, we'll cover some exceptions (such as assigning something to the new TTreeNode object's void*).

Add and AddFirst operate on the current group of siblings--they take a TTreeNode pointer to any existing node, then adding a sibling to it. For example, you can add a new pitcher named Tom Gordon to the bottom of Boston's pitching roster by calling Add and passing in a pointer to an existing Boston pitcher:

```
TTreeNode* tomGordonNode =
  TreeView1->Items->Add(pedroMartinezNode,
                        "Tom Gordon");
```

The Insert method gives you a bit more precision when adding new nodes. Insert takes a pointer to an existing node and inserts a new node just prior to it. For example, the following code will place Tom Gordon between Martinez and Wakefield on the Boston pitching roster:

```
TTreeNode* tomGordonNode = TreeView1->Items->
  Insert(timWakefieldNode, "Tom Gordon");
```

Like Add and AddFirst, Insert operates on the current sibling group. Note that calling Insert with a NULL Node parameter will append the new node to the end of the list of root nodes--essentially the equivalent of calling Add with a NULL Node parameter.

Adding to the family

AddChild and AddChildFirst function like Add and AddFirst, but they operate on the children of an existing node rather than the siblings of an existing node. Here's a quick example that appends a new infielder to the Boston team:

```
TreeView1->Items->
  AddChild(bostonInfieldersNode, "Mo Vaughn");
```

(AddChildFirst would have placed Mo at the top). Now, let's see how to delete a TTreeNode. When we added Tom Gordon to the list of pitchers, we got back a pointer to the new node and stored it in tomGordonNode. To get rid of him, we can simply make the following call:

```
tomGordonNode->Delete();
```

Note: Be careful with Delete
Calling Delete on a node that has children associated with it will delete all the children, as well--this includes *all* nodes subsequent to the node being deleted (children, grandchildren, and so on).

You can just as easily remove all the children of any parent node without removing the parent itself, by calling the DeleteChildren method. The following code wipes out the entire Boston pitching staff, but leaves the Pitchers node:

```
bostonPitchers->DeleteChildren();
```

Adding and deleting nodes that point to something

When we talked earlier about TTreeNode, we mentioned that it could point to the object of your choice. You can set this object pointer when you create the new TTreeNode by using an alternate series of add and insert methods that take a pointer to the object as a void* parameter. Table A shows the methods you've already seen and the corresponding methods that accept an object.

Table A: Add and Insert methods that accept objects

Method	Method With Object
Add	AddObject
AddFirst	AddObjectFirst
AddChild	AddChildObject
AddChildFirst	AddChildObjectFirst
Insert	InsertObject

The signature changes slightly to include the void*:

```
TTreeNode* __fastcall AddObject(TTreeNode* Node,
  const System::AnsiString S, void * Ptr);
```

The add and insert methods invoke their Object versions behind the scenes. They simply provide NULL as the value for the void* parameter, indicating that no object is present. Besides supplying a pointer through the add and insert methods, you can also set and access the pointer through the node's *Data* property.

Suppose we have a Stats object containing a pitcher's current statistics. The following code creates a new Stats object and attaches it to a new TTreeNode:

```
Stats *newStats = new Stats();
tomGordonNode = TreeView1->Items->
  AddObject(pedroMartinezNode, "Tom Gordon",
  newStats);
```

You can also attach this object to a pitcher's TTreeNode later by setting the *Data* property:

```
tomGordonNode->Data = (void*) newStats;
```

And the following code gets the object back:

```
Stats* tomsStats = (Stats*) tomGordonNode->Data;
```

Being able to associate any object with a TTreeNode adds exciting possibilities to your code. However, you must treat the *Data* property with caution. For example, the TTreeNode Delete method doesn't delete the memory associated with the void*--it's up to your code to take care of that. Here's a simple way to do it:

```
Stats* tomsStats = (Stats*) tomGordonNode->Data;
delete tomsStats;
tomGordonNode->Delete();
```

If you're going to delete the object pointed to by *Data* but leave the TTreeNode, it's a good idea to set the Data pointer to NULL in case it's checked by subsequent operations in your program. You can do so as follows:

```
Stats* tomsStats = (Stats*) tomGordonNode->Data;
delete tomsStats;
tomGordonNode->Data = (void*) NULL;
```

You now have the tools to build a TTreeView. Next, we'll discuss some methods you can use to navigate a TTreeView from within your code.

Navigating TTreeView

You can use numerous methods to locate specific nodes in a TTreeView at runtime--some of them are positional, and others are absolute. Let's examine how both methods work and look at examples of each, starting with positional.

Note: Watch out for lowercase
Some of the methods in this section (getFirstChild, getNextSibling, and getPrevSibling) don't start with a capital letter, as do the other methods. This is a bug in C++Builder that has been reported.

Positional methods

You can find the first root node of any tree using the GetFirstNode method, like this:

```
TTreeNode* firstRoot = TreeView1->Items->GetFirstNode();
```

Since all root nodes are siblings of each other (part of the same sibling group), you can then determine the other root nodes using the getNextSibling method. This code adds the names of a tree's root nodes to a ListBox:

```
????
TTreeNode* rootNode =
  TreeView1->Items->GetFirstNode();
while ( rootNode )
{
  ListBox1->Items->Add(rootNode->Text);
  rootNode = rootNode->getNextSibling();
}
????
```

If we executed this code on the TTreeView in Figure A, ListBox1 would contain two items: American League and National League. The getNextSibling method's getPrevSibling counterpart returns the previous sibling in a sibling group. If there are no previous or next siblings, the method will return a NULL--your code should check for this before trying to perform some operation on the resulting TTreeNode pointer.

In addition to the sibling methods, four methods return information about the children of a TTreeNode: getFirstChild, GetNextChild, GetPrevChild, and GetLastChild. Here are two short examples--one that walks the teams in the American League East division from top to bottom, and one that goes from bottom to top (we've set alEast to the correct parent node):

```
TTreeNode* team = alEast->getFirstChild();
while ( team )
{
  ListBox1->Items->Add(team->Text);
  team = alEast->GetNextChild(team);
}
```

```
TTreeNode* team = alEast->GetLastChild();
while ( team )
{
  ListBox1->Items->Add(team->Text);
  team = alEast->GetPrevChild(team);
}
```

Here's another way to walk the children of alEast:

```
if ( alEast->HasChildren )
{
  ListBox1->Items->
    Add("Here are the teams in the AL East:");
  for ( int team = 0; team < alEast->Count;
    team++ )
    ListBox1->Items->
       Add("  " + alEast->Item[team]->Text);
}
else
  ListBox1->Items->
    Add("There are no teams in the AL East!");
```

This example exposes some additional methods and properties we haven't covered yet, but their purpose will be obvious. Again, we're dumping information into ListBox1.

Notice that you can address an individual child by its item index. For any TTreeNode with children, each child has a unique index in its sibling group and can be addressed by the parent via parentNode->Item[x]. A child can also tell you its position, as follows:

```
int x = childNode->Index;
```

You can find out any TTreeNode's parent using the *Parent* property (this will return a NULL for root nodes):

```
TTreeNode* parent = childNode->Parent;
```

Interestingly enough, C++Builder also provides a method to find out if one node is the parent of another. Here, node A checks to see if node B is its parent:

```
bool isParent = nodeA->HasAsParent(nodeB);
```

You can also test in the opposite direction, checking to see whether node A is a child of node B. You do this using the IndexOf method, which returns the position of the child node among its siblings (starting with zero):

```
int pos = nodeB->IndexOf(nodeA);
```

The method returns -1 if the child node isn't found. Finally, you can ask any TTreeNode its level by using the *Level* property:

```
int nodeLvl = anyNode->Level;
```

Absolute positioning methods

You've already seen an absolute positioning method that gives you direct access to a parent's children via the *Item* property. The other absolute method--AbsoluteIndex--is a unique index assigned to every TTreeNode in a TTreeView. AbsoluteIndex values depend on a node's position within the TTreeView, and will change with the addition and deletion of nodes. Values are assigned starting with 0 for the first root node. Note that children of a current node will contain subsequent AbsoluteIndex values, not the siblings. Table B shows the AbsoluteIndex values for some of the members
of Figure A.

Table B: AbsoluteIndex values from Figure A

TTreeNode	Index
American League	0
East	1
Boston	2
Pitchers	3
Pedro Martinez	4
Tim Wakefield	5

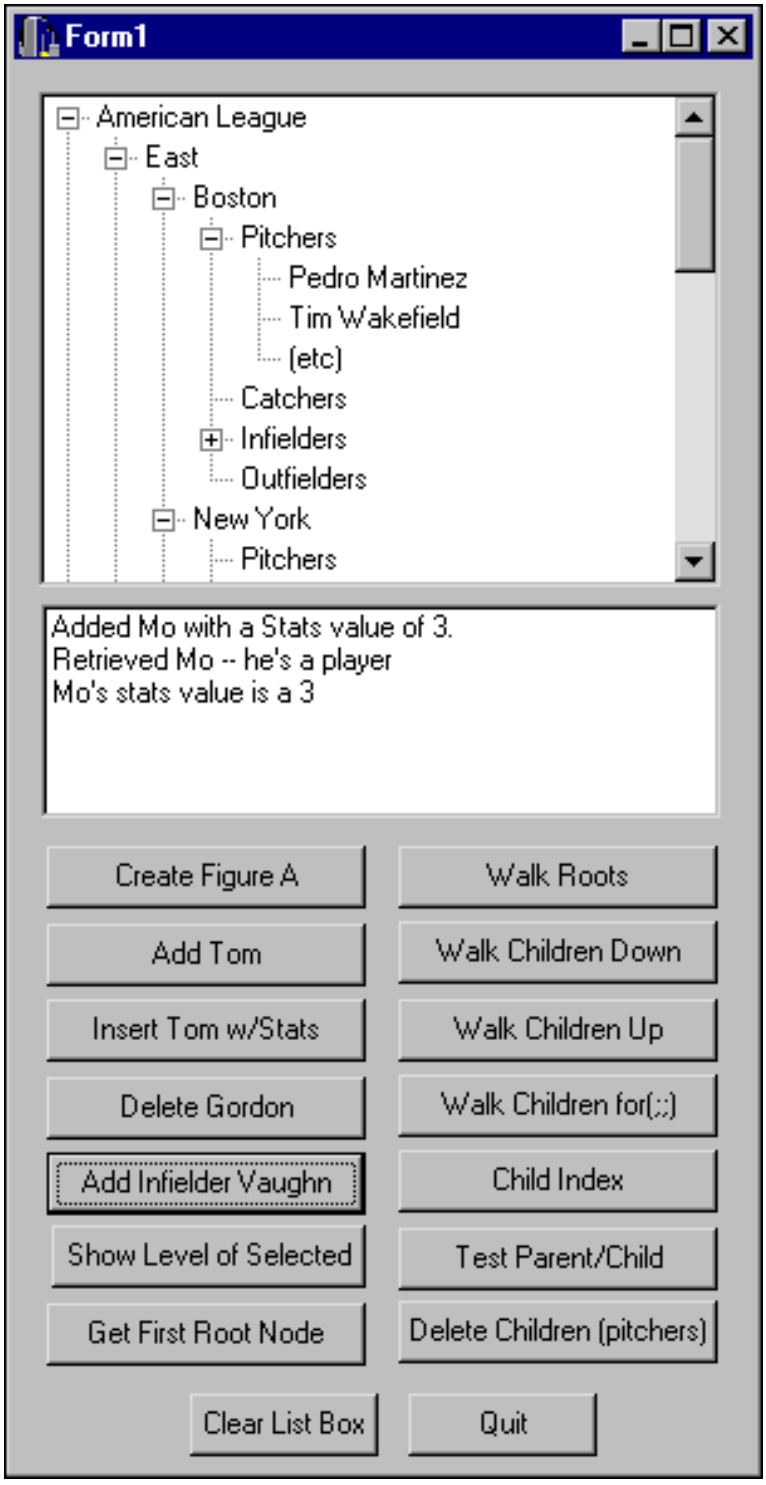
If the (etc) node didn't exist, then Catchers would be 6. We don't recommend relying heavily on AbsoluteIndex in your code, because the values change regularly.

Building a TTreeView

Now, let's write some code that reproduces the TTreeView shown in Figure A and illustrates many of the concepts we've discussed. The sample source code appears in Listing A (Unit1.cpp), and the header file is in Listing B (Unit1.h).

The application consists of one form containing a TTreeView, a list box to display method output, and several buttons that invoke the methods demonstrated in the article. Because the application demonstrates how to build a TTreeView, you should click the buttons in a logical order: first the left column from top to bottom, then the right column from top to bottom. Figure B shows the application's main window.

Figure B: Our sample application illustrates many of the concepts we've discussed.



Listing A: Unit1.cpp

```
//---------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
```

```
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
  tomGordonNodeAdd = NULL;
  tomGordonNodeInsert = NULL;
}
//---------------------------------------------------

void __fastcall TForm1::CreateFigAClick(TObject *Sender)
{

  // Add the root nodes
  al = TreeView1->Items->Add(NULL, "American League");
  nl = TreeView1->Items->Add(NULL, "National League");

  // Add the divisions in both leagues
  alEast    = TreeView1->Items->AddChild(al, "East");
  alCentral = TreeView1->Items->AddChild(al, "Central");
  alWest    = TreeView1->Items->AddChild(al, "West");
  nlEast    = TreeView1->Items->AddChild(nl, "East");
  nlCentral = TreeView1->Items->AddChild(nl, "Central");
  nlWest    = TreeView1->Items->AddChild(nl, "West");

  // Add teams only to American League East to save space
  bosNode = TreeView1->Items->AddChild(alEast, "Boston");
  TreeView1->Items->AddChild(alEast, "New York");
  TreeView1->Items->AddChild(alEast, "Toronto");
  TreeView1->Items->AddChild(alEast, "Baltimore");
  TreeView1->Items->AddChild(alEast, "Tampa Bay");

  // Insert player categories under each team
  for ( int i = 0; i < alEast->Count; i++ )
  {
    TreeView1->Items->AddChild(alEast->Item[i], "Pitchers");
    TreeView1->Items->AddChild(alEast->Item[i], "Catchers");
    TreeView1->Items->AddChild(alEast->Item[i], "Infielders");
    TreeView1->Items->AddChild(alEast->Item[i], "Outfielders");
  }

  // Add the Boston pitchers
  pitNode = bosNode->getFirstChild();
  pedroMartinezNode = TreeView1->Items->
    AddChild(pitNode, "Pedro Martinez");
  timWakefieldNode = TreeView1->Items->
    AddChild(pitNode, "Tim Wakefield");
  TreeView1->Items->
```

```
    AddChild(pitNode, "(etc)");

  // Find the infield for use later
  // pitchers
  bostonInfieldersNode = bosNode->getFirstChild();
  // Catchers
  bostonInfieldersNode = bostonInfieldersNode->getNextSibling();
  // Infielders!
  bostonInfieldersNode = bostonInfieldersNode->getNextSibling();

  // Show the entire tree from the top
  TreeView1->FullExpand();
  TreeView1->TopItem = TreeView1->Items->GetFirstNode();
}
//---------------------------------------------------
void __fastcall TForm1::QuitClick(TObject *Sender)
{
  Close();
}
//---------------------------------------------------

void __fastcall TForm1::AddTomGordonClick(TObject *Sender)
{
  // Add a Tom Gordon and put Stats in the Data property
  Stats* lclStats = new Stats(1);
  tomGordonNodeAdd = TreeView1->
    Items->Add(pedroMartinezNode, "Tom Gordon");
  tomGordonNodeAdd->Data = (void *) lclStats;
  ListBox1->Items->Add("Inserted Tom with a new stats object");
  ListBox1->Items->Add("containing the number 1");
}
//---------------------------------------------------

void __fastcall TForm1::InsertTomGordonClick(TObject *Sender)
{
  // Insert a Tom Gordon with a Stats object
  Stats* lclStats = new Stats(2);
  tomGordonNodeInsert = TreeView1->
    Items->InsertObject(timWakefieldNode,"Tom Gordon", lclStats);
  ListBox1->Items->Add("Inserted Tom with a new stats object");
  ListBox1->Items->Add("containing the number 2");
}
//---------------------------------------------------

void __fastcall TForm1::DeleteGordonClick(TObject *Sender)
{
  if ( tomGordonNodeAdd != NULL )
```

```
  {
    // Retrieve Stats object from Data property
    Stats* statsInstance = (Stats*) tomGordonNodeAdd->Data;
    String number = statsInstance->getStats();
    ListBox1->Items->Add("Stats contained a " + number);
    // Delete Stats and then the Tom Gordon node.
    delete statsInstance;
    tomGordonNodeAdd->Data = (void*) NULL;
    tomGordonNodeAdd->Delete();
    ListBox1->Items->Add("Deleted the added Tom Gordon node.")
    tomGordonNodeAdd = NULL;
  }

  if ( tomGordonNodeInsert != NULL )
  {
    // Retrieve Stats object from Data property
    Stats* statsInstance = (Stats*) tomGordonNodeInsert->Data;
    String number = statsInstance->getStats();
    ListBox1->Items->Add("Stats contained a " + number);
    // Delete Stats and then the Tom Gordon node.
    delete statsInstance;
    tomGordonNodeInsert->Data = (void*) NULL;
    tomGordonNodeInsert->Delete();
    ListBox1->Items->
      Add("Deleted the inserted Tom Gordon node.");
    tomGordonNodeInsert = NULL;
  }
}
//---------------------------------------------

void __fastcall TForm1::AddVaughnClick(TObject *Sender)
{
  // Add Vaughn as child to Boston's infielders. Test handle
  // object with Vaughn. Adapted from calander example.
  nodeIdHandle* moData = new nodeIdHandle;

  nodeIdHandle* moDataOut;
  nodeIdHandle* moDataIn;

  moData->nodeType = ev_player;
  moData->obj = NULL;
  moVaughn = TreeView1->Items->AddChild(bostonInfieldersNode, "Mo Vaughn");
  Stats* moStats = new Stats(3);
  moVaughn->Data = (void*) moData;

  moDataOut = (nodeIdHandle*) moVaughn->Data;
  moDataOut->obj = (void*) moStats;
```

```
  moStats = NULL;
  ListBox1->Items->Add("Added Mo with a Stats value of 3.");

  moDataIn = (nodeIdHandle*) moVaughn->Data;
  if ( moDataIn->nodeType == ev_player )
    ListBox1->Items->Add("Retrieved Mo -- he's a player");
  moStats = (Stats*) moDataIn->obj;
  String moVal = moStats->getStats();
  ListBox1->Items->Add("Mo's stats value is a " + moVal);
}
//---------------------------------------------------

void __fastcall TForm1::DeleteChildrenClick(TObject *Sender)
{
  // Delete all Boston pitchers
  pitNode->DeleteChildren();
}
//---------------------------------------------------

void __fastcall TForm1::GetFirstRootClick(TObject *Sender)
{
  // Show the name of the first root node
  ListBox1->Items->Add("First root node is " +
    TreeView1->Items->GetFirstNode()->Text);
}
//---------------------------------------------------

void __fastcall TForm1::ShowLevelClick(TObject *Sender)
{
  // Show the level of the selected node
  TTreeNode *fc = TreeView1->Selected;
  if ( fc != NULL )
    ListBox1->Items->Add("Node " + fc->Text + " is at level " +
      fc->Level);
  else
    ListBox1->Items->Add("Nothing selected!");
}
//---------------------------------------------------

void __fastcall TForm1::WalkRootsClick(TObject *Sender)
{
  // Get the first tree node (root)
  TTreeNode* rootNode = TreeView1->Items->GetFirstNode();
  while ( rootNode )
  {
    ListBox1->Items->Add(rootNode->Text);
```

```
    // Continue as long as there are siblings
    rootNode = rootNode->getNextSibling();
  }
}
//---------------------------------------------------

void __fastcall TForm1::WalkChildrenDownClick(TObject *Sender)
{
  // Get the first AL East child (team)
  TTreeNode* team = alEast->getFirstChild();
  while ( team )
  {
    ListBox1->Items->Add(team->Text);

    // Continue as long as there are children
    team = alEast->GetNextChild(team);
  }
}
//---------------------------------------------------

void __fastcall TForm1::WalkChildrenUpClick(TObject *Sender)
{
  // Get the last child node in the AL East (team)
  TTreeNode* team = alEast->GetLastChild();
  while ( team )
  {
    ListBox1->Items->Add(team->Text);
    // Walk backwards as long as there are previous children
    team = alEast->GetPrevChild(team);
  }
}
//---------------------------------------------------

void __fastcall TForm1::WalkChildrenForClick(TObject *Sender)
{
  // If AL East has children, add names to ListBox1
  if ( alEast->HasChildren )
  {
    ListBox1->Items->Add("Here are the teams in the AL East:");
    for ( int team = 0; team < alEast->Count; team++ )
      ListBox1->Items->Add("  " + alEast->Item[team]->Text);
  }
  else
    ListBox1->Items->Add("There are on AL East teams!");
}
//---------------------------------------------------
```

```
void __fastcall TForm1::ChildIndexClick(TObject *Sender)
{
  // Get the index of the selected node
  TTreeNode *fc = TreeView1->Selected;
  if ( fc != NULL )
    ListBox1->Items->Add("Child node index for " + fc->Text +
      " is " + fc->Index);
  else
    ListBox1->Items->Add("Nothing selected!");
}
//---------------------------------------------------

void __fastcall TForm1::TestParentChildClick(TObject *Sender)
{
  // Test out Parent, HasAsParent, and Index
  TTreeNode* parent = bosNode->Parent;
  bool isparent = bosNode->HasAsParent(alEast);
  int bosPos = alEast->IndexOf(bosNode);

  if ( isparent )
  {
    String bosLoc = bosPos;
    ListBox1->Items->
      Add("Checked to see if Boston's parent was");
    ListBox1->Items->Add("AL East...it was!  Its child index is "
      + bosLoc);
    ListBox1->Items->Add("Boston's parent name via the Parent");
    ListBox1->Items->Add("was " + parent->Text);
  }
}
//---------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  // Empty the ListBox
  ListBox1->Items->Clear();
}
//---------------------------------------------------
```

Listing B: Unit1.h

```
//---------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------
```

```
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
//---------------------------------------------------

class Stats
{
  public:
    Stats(int x) { s = x; }
    ~Stats() { }
    int getStats(void) { return s; }
  private:
    int s;
};
//---------------------------------------------------

enum entryValue {ev_league = 0, ev_division, ev_team, ev_player};

struct nodeIdHandle
{
  entryValue nodeType;
  void* obj;
};

//---------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
  TTreeView *TreeView1;
  TButton *CreateFigA;
  TButton *AddTomGordon;
  TButton *InsertTomGordon;
  TButton *AddVaughn;
  TButton *DeleteGordon;
  TButton *DeleteChildren;
  TButton *GetFirstRoot;
  TButton *WalkRoots;
  TButton *WalkChildrenDown;
  TButton *Quit;
  TListBox *ListBox1;
  TButton *WalkChildrenUp;
  TButton *WalkChildrenFor;
  TButton *ChildIndex;
  TButton *TestParentChild;
  TButton *ShowLevel;
```

```
  TButton *Button1;
  void __fastcall CreateFigAClick(TObject *Sender);
  void __fastcall QuitClick(TObject *Sender);
  void __fastcall AddTomGordonClick(TObject *Sender);
  void __fastcall InsertTomGordonClick(TObject *Sender);
  void __fastcall DeleteGordonClick(TObject *Sender);
  void __fastcall AddVaughnClick(TObject *Sender);
  void __fastcall DeleteChildrenClick(TObject *Sender);
  void __fastcall GetFirstRootClick(TObject *Sender);
  void __fastcall ShowLevelClick(TObject *Sender);
  void __fastcall WalkRootsClick(TObject *Sender);
  void __fastcall WalkChildrenDownClick(TObject *Sender);
  void __fastcall WalkChildrenUpClick(TObject *Sender);
  void __fastcall WalkChildrenForClick(TObject *Sender);
  void __fastcall ChildIndexClick(TObject *Sender);
  void __fastcall TestParentChildClick(TObject *Sender);
  void __fastcall Button1Click(TObject *Sender);
private:        // User declarations
  // Pointers will help find specific parts of TTreeView later.
  TTreeNode *al, *alEast, *alCentral, *alWest, *nl, *nlEast,
    *nlCentral, *nlWest, *bosNode, *pitNode, *pedroMartinezNode,
    *timWakefieldNode, *tomGordonNodeAdd, *tomGordonNodeInsert,
    *bostonInfieldersNode, *moVaughn;
  Stats* stats;
public:         // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------
#endif
```

Conclusion

In this article, we've tried to overcome the complexity of TTreeView by breaking it into manageable pieces. We've shown you how to create a TTreeView from scratch, and how to find your way around the TTreeView nodes. In the second part of this series, we'll discuss how to control the addition, deletion, and movement of TTreeNodes based on a user's interaction with TTreeView. We'll also work on controlling the appearance of TTreeView, as well as handling user events, persistence, and drag-and-drop operations.

November 1998

Add a handle

by Bill Whitney

You can add integrity to a TTreeView by attaching *handle* objects to the Data property of each TTreeNode (which you can later use to identify the node's purpose). Or, you can maintain a separate data structure that leads you to specific points in the TTreeView.

Let's consider the first option, which is useful for trees with many nodes. (We'll use the baseball tree from the accompanying article.) Assume we have a TTreeNode void* that can point to any object. We create an identifier structure and place one in each TTreeNode we create:

```
enum entryValue {ev_league = 0, ev_division,
                 ev_team, ev_player};

struct nodeIdHandle
{
  entryValue nodeType;
  void* obj;
};
```

Now, we just insert nodeIdHandles into the TTreeNode *Data* properties. Each nodeIdHandle can not only store a pointer to any object (just as TTreeNode->Data could), but tell you what type it is, as well. (If you're an advanced C++ programmer, you'll recognize other ways to do this, but our method will work.)

The code in Listing A adds a player node to our TTreeView. We're using the TTreeNode->Data property to store a handle that identifies the player and maintains a pointer to a Stats object.

Listing A: Handle example

```
nodeIdHandle* anyHandle = new nodeIdHandle;
nodeIdHandle* tmpHandle;

// Add a new player.
newPlayerNode = TreeView1->Items->
  AddChild(whereToAdd, "Babe Ruth");

// Set new handle node to player type.
```

```
        Point its object to NULL.
anyHandle->nodeType = ev_player;
anyHandle->obj = NULL;

// Allocate a new Stats object.
Stats* playerStats = new Stats(3);
//  Set the TTreeNode to point to the handle.
newPlayerNode->Data = (void*) anyHandle;

// Point our handle to Stats.
tmpHandle = (nodeIdHandle*) newPlayerNode->Data;
tmpHandle->obj = (void*) playerStats;
```

In the second option, we use a separate data structure to manage a TTreeView. It's as simple as creating one or more TTreeNode object pointers to help us locate things quickly in the TTreeView. For example, if we need frequent access to all of the teams in the East division in the American League, we might create a variable that points to the East division TTreeNode:

```
TTreeNode* alEast;
```

When the tree is built, the node pointer to the East node is stored here:

```
alEast = TreeView1->Add(americanLeagueNode,
  "East");
```

November 1998

Displaying multiple main forms

by Kent Reisdorph

You can download our sample files as part of the file nov98.zip. Visit www.zdjournals.com/cpb and click on the Source Code hyperlink.

Certain types of applications change the look of the main window, depending on the state of the application. In this article, we'll show you how to make your application behave as if it can change the main form on the fly. Our main focus will be on the PageControl and TabControl components, but we'll also take a quick peek at the Notebook component.

Changing the main form

At first glance, it would appear that the easiest way to change the main form of an application is to change the *MainForm* property of the Application object. However, this property is read-only. The *MainForm* property is set by VCL at runtime and can't be changed by the application. Clearly, we need a different solution.

You can approach this problem in several ways. The simplest and most effective way is to implement some sort of paging mechanism on the main form. The other methods are more difficult to implement then a paging solution, so we won't waste time discussing them.

A paging solution means using a page container component which, naturally, contains two or more child pages. These pages can contain other components such as edits, labels, memos, check boxes, and so on. A paging component can switch pages as needed at runtime, using one of two VCL components: PageControl or Notebook. Let's take a look at these components in more detail.

The PageControl component

The PageControl component, along with its helper, the TTabSheet class, is an encapsulation of the Win32 tab control. It has relatively few properties. The *ActivePage* property contains a pointer to the active page; you can read this property to determine the active page or set it to change the active page. For example:

```
PageControl->ActivePage = TabSheet1;
```

The *PageCount* property tells you how many pages the page control contains. The *Pages* property is an array of TTabSheet pointers; you can also use this property to set the active page by index:

```
// set page 1 active
PageControl->ActivePage = PageControl->Pages[0];
```

When used in the way we present in this article, the PageControl component doesn't have any methods of interest to us. The methods of TPageControl depend on the pages' tabs being visible, and in this case, we'll hide the tabs.

The PageControl component has two events of note, but unfortunately neither works when changing pages via code. The *OnChanging* event will notify you that a page is about to change, and the *OnChange* event will tell you when a page has already changed. In particular, the *OnChanging* event would be useful for validating a page's contents. Both events depend on a tab click, so don't waste your time trying to use them in this type of implementation.

To begin creating your multi-page main form, first drop a PageControl component on your main form. You'll probably want to change the *Align* property to alClient, so the page control fills the entire form.

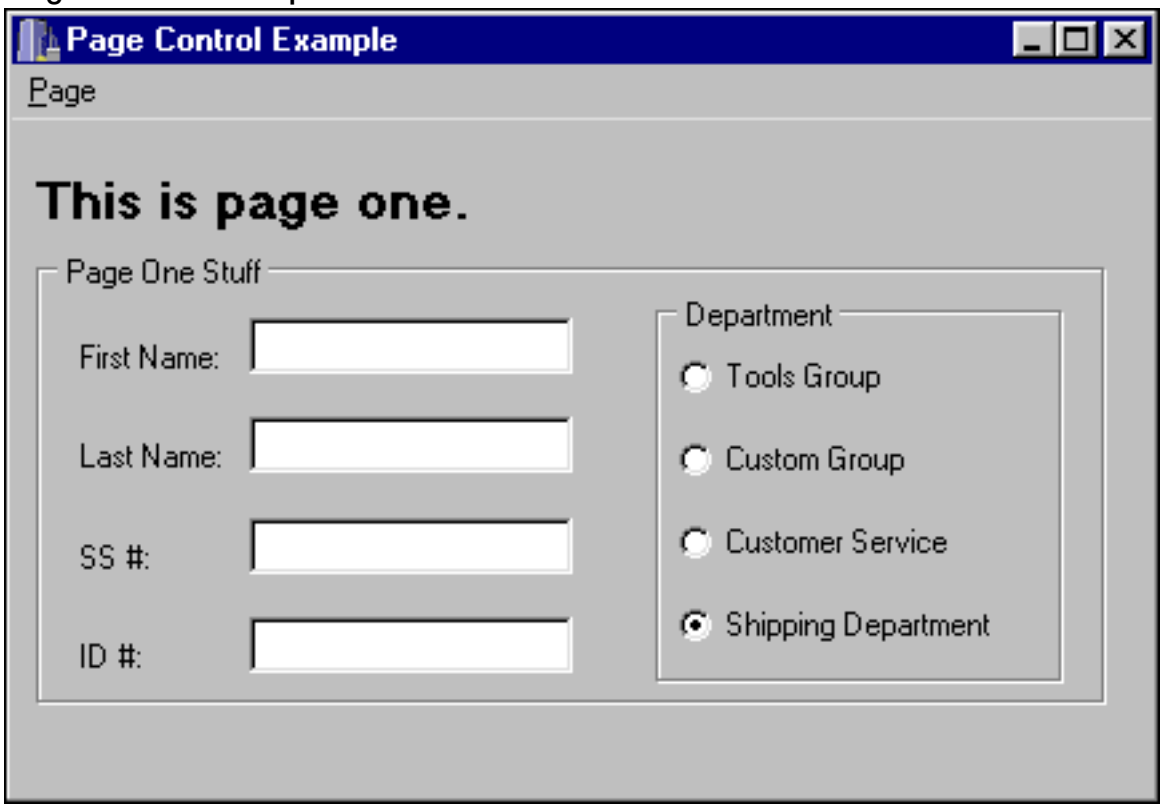
Naturally, the PageControl component isn't much good without pages--you need to create some. To create pages at design time, right-click on the page control and choose New Page from the speed menu. (Note the Next Page and Previous Page menu items as well--these will come in handy later when you're working with the individual pages.) C++Builder will create a new TTabSheet object and place it on the page control.

At this point, a tab appears at the top of the form with the name of the tab sheet in the caption. In this case, you don't want to see the tabs; set the *TabVisible* property of the tab sheet to False. When you do this, the tab sheet expands to occupy the full size of the page control.

Add as many pages as you need, placing any components you want on each page. Use the Next Page and Previous Page items on the speed menu to navigate between the pages. Since the tabs aren't visible, this is the most convenient way to switch between pages. (You can also use the component selector at the top of the Object Inspector.)

That's essentially all there is to using a PageControl component. Figure A shows the first page of a sample multi-page application available from our Web site.

Figure A: This application's Page menu provides easy access to several pages in the PageControl component.



You can change the pages at runtime as we indicated earlier, using the *ActivePage* property. As your users interact with your program, you can change pages as needed. The resulting effect is an application that changes views seamlessly.

The Notebook component

The Notebook component is also a paging control. You can find it on the Component Palette under the Win 3.1 tab.

Note: Don't ignore the Win 3.1 tab
Don't neglect a whole set of components simply because they appear on the Component Palette's Win 3.1 tab! Some very useful components reside in this group.

The Notebook component is every bit as useful and capable as the PageControl component for our purposes. In fact, it has some features that make it more desirable. The *PageIndex* property, for instance, can set the active page by page index--the PageControl component

doesn't have that ability except through the *Pages* property. The Notebook component also has an *ActivePage* property that can set the active page by name rather than by index.

The Notebook component differs from the PageControl component in that the notebook manages its pages internally by the notebook (as opposed to the PageControl's pages, which are individual instances of the TTabSheet class). The page list is simply a list of strings contained in the *Pages* property. Other than that, you can use the Notebook component just as we describe in the section on the PageControl component.

PageControl or Notebook?

Both the PageControl component and the Notebook component have strengths and weaknesses. In the end, it's up to you to decide which VCL paging component to use. This decision isn't a big deal--you'll be fine either way.

November 1998

Trouble on the border

by Kent Reisdorph

Both the PageControl component and the Notebook component we discuss in the accompanying article have problem borders. In fact, the notebook has no border at all. You can easily fix this, however, by placing the notebook on a panel and setting the border style of the panel as desired. (It's unlikely that you'll want to go to this effort, since you typically won't want a border on your paging control.)

The PageControl component has the opposite problem: You're forced to accept the default raised-edge border because the component has no *BorderStyle* property. (This is really a limitation of the underlying Win32 tab control, rather than the VCL PageControl component.) Fortunately, you can get around this border problem--but only by brute force.

To hide the raised border, first set the page control's *Align* property to alNone. Next, set the form's *AutoScroll* property to false. Then, create the following event handler for the *OnResize* event:

```
void __fastcall
TForm1::FormResize(TObject *Sender)
{
  PageControl1->SetBounds(-3, -3,
    ClientWidth + 5, ClientHeight + 5);
}
```

Manually position the page control so the painted border is hidden outside the form's displayable area. This approach has one drawback: The contents of the form may flicker as you resize the form (depending on your Windows settings). As an alternative to using the *OnResize* event, you can disallow sizing of the form. In this case, you'll want to position the page control in the form's *OnCreate* event handler. If you go this route, change the form's *BorderStyle* property to bsSingle and place the previous code snippet in the form's *OnCreate* event handler.

November 1998

Transferring TRichEdit text

by Gerry Myers

Sample files are available as part of the file nov98.zip from our Web site. Visit www.zdjournals.com/cpb and click the Source Code hyperlink.

If you've ever used a Rich Edit component, you've no doubt been impressed with its visual appeal. You can display characters in different fonts and paragraphs with varied indentations. And, most of this functionality is hidden, so you needn't know anything about the rich text format (RTF) layout to make good use of it.

That's an advantage if you're concerned only with displaying your text. However, what if your application requires that you gain access to the underlying formatted text--for instance, to copy formatted text to another component, or to send an RTF string out a socket or port? You'll find that this hidden functionality becomes a disadvantage.

In this article, we'll show you how to directly access the formatted text of a TRichEdit component. We'll also discuss the marriage between the TRichEdit VCL component and the underlying Rich Edit common control provided by Windows. (Throughout this article, we'll use the terms *component* and *common control* to differentiate between the TRichEdit *component* and the Windows Rich Edit *common control*.)

The Windows Rich Edit common control

The Write and WordPad applications that come with Windows are good examples of the Rich Edit common control at work. Note that the Rich Edit control doesn't provide mechanisms for the user to directly format the text that appears in buttons, menus, font list boxes, and so on. The application that uses the control must offer any such capabilities. However, once the user specifies certain formatting, the application can then call the functions of the Rich Edit control to set the characteristics of the text.

The TRichEdit component

As great as the Windows Rich Edit common control is, C++Builder makes it easier to use. The TRichEdit component is a wrapper class for the Rich Edit common control. That doesn't mean that TRichEdit mimics the operation of the common control--it simply provides an object-oriented interface to it. TRichEdit takes care of all the structs and pointers required when using the common control directly. For our discussion, we'll cover the mechanisms TRichEdit provides that allow you to work directly on the RTF text (as opposed to the plain text). If you look through the list of TRichEdit methods, you'll come across several that you'd think (by their titles) return the contents of the rich edit as a string

or character buffer. Let's look at one in particular: GetSelTextBuf(). According to the Help file, this method loads a user-provided character buffer with the selected (highlighted) text from the Rich Edit component. Sounds good. Unfortunately, it gives you only the text and none of the formatting (control) characters. So, if you have a nicely formatted string such as

This **is** some *formatted* text...

all you get back from a method like GetSelTextBuf() is:

This is some formatted text...

In fact, all the methods and properties of TRichEdit that have the word *text* in their titles operate the same way. You may then turn to TRichEdit's *Lines* property, which is a TStrings* type. Knowing that the text of the Rich Edit control is stored there, you may try code like

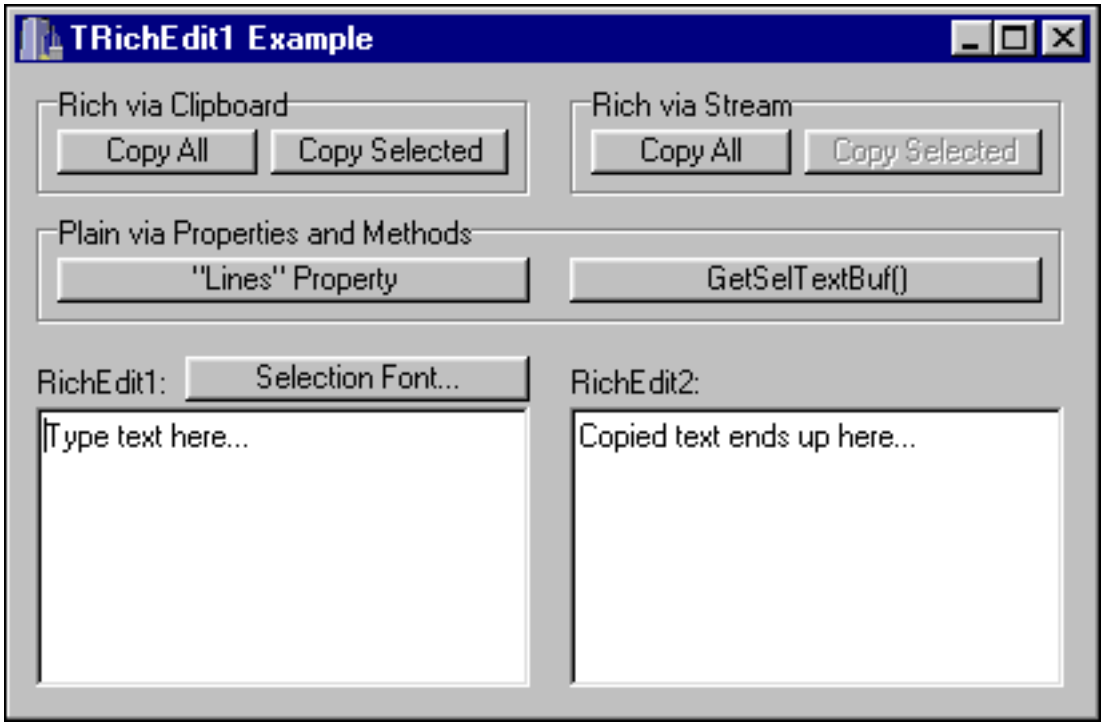
```
String s = RichEdit1->Lines->Text;
```

to pull out all the rich-edit text at one time. Again, you'll find that only the plain text is returned. So, what's the deal with hiding the RTF characters? Now, you've discovered the disadvantage of the Rich Edit common control provided by Windows.

An example

To explain the things that work and don't work when you're accessing the RTF contents of the Rich Edit control, we wrote the example program shown in Figure A.

Figure A: This example program demonstrates using the TRichEdit component.



We've included buttons that let you copy text from one TRichEdit component to another. Notice the three groups of buttons: one group copies RTF text using the Clipboard's functionality, one copies RTF text using streams, and the third copies plain text using the TRichEdit component's properties and methods.

We've already discussed the two buttons for transferring plain text. The other buttons represent the means to access the underlying RTF text in a Rich Edit common control via the Clipboard and via streams.

Using the Clipboard

Since the Rich Edit common control is integrated with Windows, that control and the Windows Clipboard are fully aware of each other. The Rich Edit common control has functions that pass RTF text to and from the Clipboard, and the Clipboard recognizes the rich text format and knows how to handle it.

Since the TRichEdit component is a wrapper around the Rich Edit common control, it contains methods that make direct use of the common control's Clipboard functions. It should be no surprise that the easiest way to transfer RTF text from one TRichEdit component to another is through the Clipboard. Listing A shows the simple code in the Rich Via Clipboard Copy All button's event handler. In just a few lines, RTF text is copied and pasted through the clipboard.

Listing A: Copying RTF text via the Clipboard

```
// The rich edit control registers the following
// clipboard formats: CF_RTF and
// CF_RETEXTOBJ (rich edit text and objects).
RichEdit1->SelectAll();
RichEdit1->CopyToClipboard();
RichEdit2->Clear();
RichEdit2->PasteFromClipboard();
```

The first line in Listing A is required to select all the text in RichEdit1, because the CopyToClipboard() method sends only the selected (highlighted) text to the Clipboard. The third line clears the destination component's contents for aesthetic reasons only.

As you may have guessed, the code for the Rich Via Clipboard Copy Selected button is even simpler, because only the selected text is copied. This code looks like Listing A, minus the line

```
RichEdit1->SelectAll();
```

Using the Clipboard, you can paste RTF text from your TRichEdit component into other applications that recognize RTF. For example, you could have a Copy button or Edit | Copy menu item that simply executes the single command

```
RichEdit1->CopyToClipboard()
```

Then, other applications like Write or WordPad can use their Edit | Paste menu item to insert your RTF text. The process can also go the other way--you can paste RTF text from other applications into your TRichEdit component.

Using streams

The Rich Edit common control (and, therefore, the TRichEdit component) provides built-in functions for reading and writing RTF files on disk--a very nice feature. Your application needn't understand anything about the RTF layout to be able to store and recall RTF data. As we mentioned earlier, the TRichEdit component has a *Lines* property. This property is a TStrings* and contains all of the text-processing properties and methods. Remember, RTF text *can't* be accessed by using the familiar *Lines* properties *Text* or *String[i]*. However, the *Lines* property can make use of the underlying Rich Edit common control's disk-streaming capability. Therefore, to save or load your RTF text, simply call

```
RichEdit1->Lines->SaveToFile( filename );

or

RichEdit1->Lines->LoadFromFile( filename );
```

Such code is great for disk access--but what about transferring RTF text to other components? There's hope. Just as the SaveToFile() and LoadFromFile() methods use streams to pass formatted data to disk, you can use the SaveToStream() and LoadFromStream() methods of *Lines* to pass RTF text to a stream (for instance, TMemoryStream). Listing B shows the code we placed in the event handler of our example's Rich Via Stream Copy All button.

Listing B: Copying via streams

```
// Create a memory stream.
TMemoryStream* strm = new TMemoryStream;

// Save the rich edit text to the stream.
RichEdit1->Lines->SaveToStream( strm );

// Reset to front of stream.
strm->Position = 0;

// Copy the stream to the other rich edit control.
RichEdit2->Lines->LoadFromStream( strm );

// Clean up.
delete strm;
```

Note: Flaw in using Text

The Help file says that SaveToStream() uses the *Text* property of *Lines* to get the RTF text from the TRichEdit component. However, if you use Lines->Text directly, only plain text is returned. Something is going on that isn't explained, but that's beyond the scope of this article.

Let's look at one last capability. So far, everything we've discussed about transferring RTF text has treated the text as a mysterious RTF object whose insides we didn't have to understand. But, what if you want to get a character buffer of the actual RTF text, so you can look at the special control characters and do some tweaking? Or, what if you want to

send the RTF text through a socket or port, but the text must first be in a character-string format?

To get an actual character string of the RTF text, you begin by using a stream, as in the previous example. The only difference is that once the RTF text is saved to the stream, you can call the stream's Read method to read the contents of the stream into a user-provided character buffer. Listing C shows the appropriate code snippet.

Listing C: RTF text-to-character buffer

```
// Create a memory stream.
TMemoryStream* strm = new TMemoryStream;

// Save the rich edit text to the stream.
RichEdit1->Lines->SaveToStream( strm );

// Reset to front of stream.
strm->Position = 0;

// Create a temporary character buffer.
int bufSize = strm->Size;
char* buf = new char [ bufSize + 1 ];

// Copy the stream to the other rich edit control.
strm->Read( buf, bufSize );

// Do stuff with the buffer now.
. . .

// Clean up.
delete strm;
delete buf;
```

Before we leave our discussion on streams, you'll notice that another button in the Rich Via Stream group--Copy Selected--is disabled. Its operation is left as an exercise for the reader to figure out how to copy only the selected text to (or from) the memory stream. You'll remember that the SaveToStream() method copies the entire TRichEdit contents regardless of what text is selected (highlighted). Sorry, I left you the harder one.

Conclusion

You can use the TRichEdit component as a simple means to display text in an attractive manner. But, when you want direct access, you can also make the component give up its RTF

text. In this article, we've shown you how.

November 1998

Common controls

by Gerry Myers

Every upgrade to Windows includes greater capabilities than previous versions (and rightly so). Windows 95 and Windows NT contain a whole set of new visual components called *common controls*. These controls include status bars, toolbars, trackbars, list and tree views, property sheets, wizards, common dialog boxes, rich edit text, and more.

Common controls offer two advantages. First, they simplify your application development job. Rather than writing custom items like status bars, tree views, and rich edit controls, you can make immediate use of the fully functioning controls that Windows provides. Second, common controls will give your applications the look and feel that people are familiar with in other Windows programs. Windows itself uses these common controls to make up its displays.

November 1998

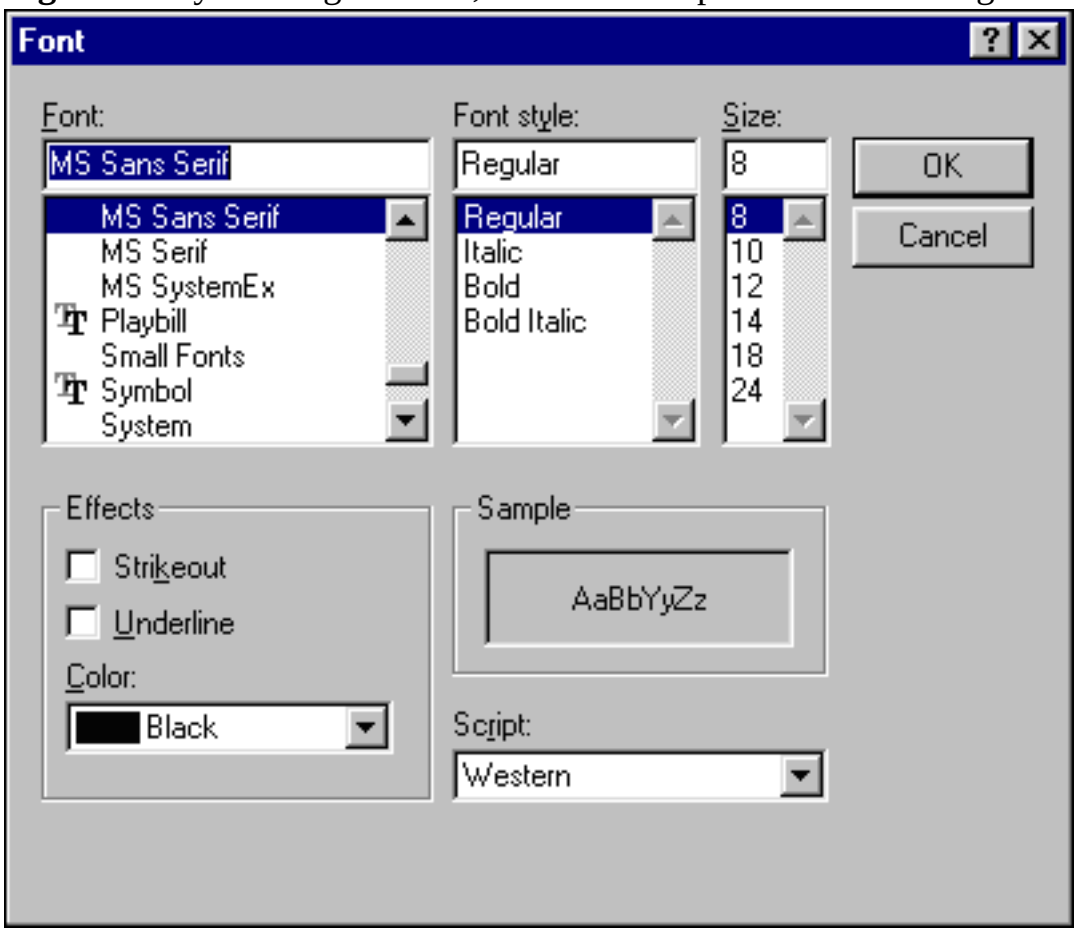
Setting the TRichEdit font

by Gerry Myers

Setting the font of a TRichEdit component isn't very difficult. To easily let the user select the font setting, you can drop a TFontDialog component onto your form and tie it to a menu item or button.

In the article "Transferring TRichEdit Text,", that's exactly what we do. The user clicks a button, and up pops the standard Font dialog box shown
in Figure A.

Figure A: By clicking a button, the user can open this Font dialog box and select text-formatting options.



The user can then specify the font name, style, and size. When the dialog box closes, our program accesses the TFontDialog *Font* property to get the individual font attributes.

Since both TRichEdit and TFontDialog have a *Font* property, your first inclination may be to simply make the assignment

```
RichEdit1->Font = FontDialog1->Font
```

However, this code will more than likely produce an unexpected result--the TRichEdit's *Font* property specifies the default font to use when no other font is specified for a text selection. When the user highlights some text and selects a specific font, you must use a different property, so only that text is set to the new font. You'll use the *SelAttribute* property, which has many of the same subproperties as the TFont class: *Color*, *Height*, *Pitch*, *Style*, and so on. Assigning most of the properties is straightforward, as you can see in Listing A.

Listing A: Assigning text properties

```
// Must set just the selected text.
Graphics::TFont* theFont = FontDialog1->Font;
RichEdit1->SelAttributes->Charset = theFont->Charset;
RichEdit1->SelAttributes->Color = theFont->Color;
RichEdit1->SelAttributes->Height = theFont->Height;
RichEdit1->SelAttributes->Name = theFont->Name;
RichEdit1->SelAttributes->Pitch = theFont->Pitch;
RichEdit1->SelAttributes->Size = theFont->Size;
```

The only fly in the ointment is the *Style* property, which is of type Set. Strangely, when you're loading characteristics into a Set, the set must appear as an L-value and as an R-value within the same statement. For example, to load the Bold characteristic, you must write your statement as follows:

```
RichEdit1->SelAttributes->Style =
    RichEdit1->SelAttributes->Style <<
    fsBold;
```

Notice that the *Style* property shows up on both sides of the assignment operator. Also, notice the double left-angle bracket (<<), which is the Set insertion operator. After looking at the TFont class, you may say, "Wait--TFont has a *Style* property and SelAttribute has a similar *Style* property. Can't I simply assign one to the other?" That's the other strange thing about the *Style* property. If you try to make a direct assignment to *Style*, as in

```
RichEdit1->SelAttributes->Style =
```

```
    theFont->Style;
```

or even as in

```
RichEdit1->SelAttributes->Style =
    RichEdit1->SelAttributes->Style +
    theFont->Style;
```

your code will compile and run without error--but your style selections (italics, bold, underline, and strikeout) won't show up in the TRichEdit component. Sorry, you'll have to load each of the four style characteristics separately. The easiest way we've found to do this is to clear the rich edit's *Style* property and then reload only the desired styles, as shown in Listing B.

Listing B: Clearing and reloading styles

```
RichEdit1->SelAttributes->Style =
    RichEdit1->SelAttributes->
Style.Clear();
if ( theFont->Style.Contains( fsBold ) )
    RichEdit1->SelAttributes->Style =
      RichEdit1->SelAttributes->Style << fsBold;
if ( theFont->Style.Contains( fsItalic ) )
    RichEdit1->SelAttributes->Style =
      RichEdit1->SelAttributes->Style << fsItalic;
if ( theFont->Style.Contains( fsUnderline ) )
    RichEdit1->SelAttributes->Style =
      RichEdit1->SelAttributes->Style << fsUnderline;
if ( theFont->Style.Contains( fsStrikeOut ) )
    RichEdit1->SelAttributes->Style =
      RichEdit1->SelAttributes->Style << fsStrikeOut;
```

October 1998

Sharing data and methods between forms

by Bill Whitney

Most of the C++Builder programs you write will contain a main form supported by one or more secondary forms. The way these forms communicate with each other and share information can not only add to the strength of your object-oriented design, but also improve the overall cohesiveness and efficiency of your application.

In this article, we'll focus on ways to open up communication between forms. We'll discuss some techniques you can use to centralize widely used objects, exchange data, and make calls to methods implemented in other forms within your application. Finally, we'll show you a short sample application that makes use of these techniques.

Hey, forms are objects too

All C++Builder forms are defined in a C++ class, meaning that they're objects as far as your application is concerned. Because they are objects, you have control over what attributes and methods they contain. This means that you can treat a C++Builder form as if it were any other C++ object instance, adding methods and attributes to suit your needs. Using the techniques we'll cover, you can increase your code's efficiency by centrally maintaining attributes and objects that will be used throughout your application. Next, let's look at where to store this information and some ways to get at it.

In touch with your main form

Every C++Builder application has a designated main form that's instantiated automatically when the program is executed. This form is an ideal place to store common information and services. You'll find it a useful repository for objects that you want to use throughout an application, but want to instantiate only once (because they're slow to construct or use a lot of memory). Before you can do anything useful with the main form, however, you need to learn how to access it from anywhere within your program. You'll do this using the Application global variable. Using Application, you can access a pointer to the application's main form through the MainForm property.

Note: About Application

For those of you new to C++Builder, Application is an instance of the TApplication object and an integral part of each C++Builder program. It encapsulates the methods and attributes prescribed by Windows to qualify as a Windows application, including the processing of Windows messages. Application is defined globally in all C++Builder programs and can provide you with many useful values (check out TApplication's properties in the online VCL help for more information).

Let's look at a short example that shows you how to declare and set a pointer to your main form. You can follow along by creating a new C++Builder application and adding a second blank form. We'll assume that the main form is called TForm1 and that it's declared and implemented in Unit1.cpp and Unit1.h (these are the defaults when you create a new C++Builder application, anyway). Assume also that you want to access the main form (TForm1) from a secondary form called TForm2 (implemented in Unit2.cpp and Unit2.h). To do this, you'll acquire a pointer to TForm1 from

```
Application->MainForm
```

Before you can declare a pointer to a TForm1 instance inside of TForm2, however, TForm2's header file (Unit2.h) must include TForm1's header file. You can do this by adding

```
#include "Unit1.h"
```

to the list of #include statements already present in Unit2.h, as follows:

```
//---------------------------------------
#ifndef Unit2H
#define Unit2H
//---------------------------------------
#include
#include
#include
#include
#include "Unit1.h" // Include Unit1 defs
```

You can now safely declare a pointer to TForm1 within your TForm2 class declaration:

```
class TForm2 : public TForm
{
__published:  // IDE-managed Components
    void __fastcall FormShow(
```

```
      TObject *Sender);
private:        // User declarations
// TForm1 pointer
TForm1* myAppsMainForm;
public:      // User declarations
    __fastcall TForm2(TComponent* Owner);
};
```

Next, you can set the TForm1 pointer to point to the application's main form when you need it. The code looks like this:

```
myAppsMainForm =
    (TForm1*) Application->MainForm;
```

You can now call any public method or access any public attributes defined in TForm1 from within TForm2 using the myAppsMainForm pointer. You might use the following code, for example, to call a method in the main form asking for identification of the application's user:

```
String userName =  myAppsMainForm->getUsrName();
```

To be able to use this call, you'd simply need to add to TForm1 a getUserName() public method that returns a String variable, like this:

```
class TForm1 : public TForm
{
  __published:  // IDE-managed Components
  private:      // User declarations
  public:       // User declarations
    __fastcall TForm1(TComponent* Owner);

  // Get the user name
    String getUserName(void) { return "Dilbert"; }
};
```

So far, you've seen how to generate a pointer to an application's main form. You've also used that pointer to call a method and retrieve a value located in the main form. In this case, the method you were calling was contained in the main form's object. Suppose for a moment, however, that the TForm1 object didn't have direct knowledge of the user name, but rather contained a pointer to another object that did. Let's invent this new object and call it UserIdent (you can place this code before TForm1's class definition inside Unit1.h for simplicity):

```
class UserIdent
{
public:
    UserIdent(String x) { name = x; }
    ~UserIdent() { }
    char* getUserName(void)
          { return name.c_str(); }
private:
    String name;
};
```

Now take a look at the changes to the TForm1 class definition:

```
class TForm1 : public TForm
{
__published:  // IDE-managed Components
private:        // User declarations
public:      // User declarations
     // Pointer to UserIdent object
     UserIdent* userInfo;
    __fastcall TForm1(TComponent* Owner);
};
```

We've removed the getUserName() method, adding in its place a new attribute called userInfo that points to an instance of UserIdent. Somewhere within TForm1, you need to instantiate UserIdent and set the userInfo pointer to the new object so you can call it later from TForm2. Here's an example that does just that, shown inside TForm1's constructor:

```
__fastcall TForm1::TForm1( TComponent* Owner) : TForm(Owner)
{
    userInfo = new UserIdent("Dilbert");
}
```

Now that you've hidden the getUserName() method inside the UserIdent class, you can no longer access it directly through myAppsMainForm as you did previously. Instead, you now have to reference the userInfo object inside of TForm1 to get the information. That call is as follows:

```
userName =
   myAppsMainForm->userInfo->getUserName()
```

Keep in mind when using this approach that userInfo is a public member of TForm1-meaning

that TForm2 can create chaos by changing where userInfo is pointing. A safer approach would be to privatize userInfo and add an accessor to TForm1 to return a pointer to userInfo. Here's a short piece of code showing the private userInfo and a public getUserInfo() method returning the pointer:

```
Class TForm1 : public TForm
{
__published:  // IDE-managed Components
private:        // User declarations
   // PRIVATE userInfo
   UserIdent* userInfo;
public: // User declarations
   __fastcall TForm1(TComponent* Owner);
    // Get the pointer to userInfo
   UserIdent* getUserInfo(void)
              { return userInfo; }
};
```

You now need to call the getUserInfo() method of myAppsMainForm to retrieve the pointer to the UserIdent object before calling getUserName(). A quick modification to TForm2 would have the getUserName() method call working again:

```
userName = myAppsMainForm->getUserInfo()->getUserName();
```

Switching roles

Let's look at this from another angle for a moment. Suppose TForm2 was the user information expert. Let's make it responsible for gathering the user's name and creating UserIdent. You'll then have TForm2 set the pointer in TForm1 to the valid instance of UserIdent created by TForm2. Check out the new TForm1 class declaration:

```
class TForm1 : public TForm
{
  __published:  // IDE-managed Components
  private:      // User declarations
    UserIdent* userInfo;// PRIVATE userInfo
  public:       // User declarations
    __fastcall TForm1(TComponent* Owner);

  // Return the pointer
    UserIdent* getUserInfo(void) { return userInfo; }
  // Set the pointer
```

```
    void setUserInfo(UserIdent* x) {  userInfo = x; }
};
```

Notice that we've added to TForm1's public area a method called setUserInfo(), which accepts a UserIdent pointer and stores it in userInfo. Once TForm2 has collected the user's name and created the UserIdent object, it must call setUserInfo() to send the UserIdent pointer back to TForm1. Here is a snippet from TForm2 that does just that:

```
void __fastcall TForm2::Button1Click(
  TObject *Sender)
{
    UserIdent *uid =
        new UserIdent(Edit1->Text);
    myAppsMainForm =
        (TForm1*) Application->MainForm;
    myAppsMainForm->setUserInfo(uid);
}
```

The code is an event handler for a button that retrieves the user's name from a text box, instantiates the UserIdent object, and sets TForm1's pointer. Note that you shouldn't delete the UserIdent pointer before you exit TForm2. That responsibility has been shifted to the main form (a task which you'd most likely do in TForm1's OnClose event handler).

Note: A word of caution about pointers
When declaring a pointer that you may have to delete, always set it to NULL somewhere during initialization (in an object's constructor, for example). Calling delete on a NULL pointer is safe...but calling delete on a pointer that you haven't initialized could be disastrous.

Putting it all together

So far, you've seen how you can modify an application's main form to act as a central location for storing data and objects. You've also learned how to get a pointer to the main form, then exploit that pointer to access data and methods stored there. And, we've explained how to create an object in a secondary form and set a pointer to that object in the main form, making that object's services available throughout your application. Now we're going to look at a sample application that uses all the techniques we've covered. The application consists of three forms: a main form called SampleAppForm and two secondary forms called IdentifyUserForm and DemoForm. During initialization, SampleAppForm creates an instance of an AppConfig class that contains some information about the program and its environment, including the fully qualified name of the executable, the path where the executable resides, and the names of a couple of other directories where the program might want to store some

information (called dat and backup). Listing A shows the SampleAppForm implementation in Unit1.h and Unit1.cpp. Notice that Unit1.h also contains the declarations for the UserIdent and AppConfig classes.

Listing A: SampleAppForm

```
// Unit1.h
//---------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------
#include
#include
#include
#include

//---------------------------------------
class UserIdent
{
public:
    UserIdent(String x) { name = x; }
    ~UserIdent() { }
    char* getUserName(void)
{ return name.c_str(); }
private:
    String name;
};

//---------------------------------------
class AppConfig
{
public:
    AppConfig()
    {
       exeName = Application->ExeName;
       exePath = ExtractFileDir(exeName);
       exeDrive = ExtractFileDrive(exeName);
       datPath = exePath + "\\dat";
       bkpPath = exePath + "\\backup";
    }
    ~AppConfig() { }
    char *getExeName(void)
{ return exeName.c_str(); }
```

```
    char *getExePath(void)
{ return exePath.c_str(); }
    char *getExeDrive(void)
{return exeDrive.c_str(); }
    char *getDatPath(void)
{ return datPath.c_str(); }
    char *getBkpPath(void)
{ return bkpPath.c_str(); }
private:
    String exeName, exePath, exeDrive, datPath, bkpPath;
};

//---------------------------------------
class TSampleAppForm : public TForm
{
__published:  // IDE-managed Components
    TButton *Demo;
    TButton *Quit;
    void __fastcall FormShow(TObject *Sender);
    void __fastcall DemoClick(TObject *Sender);
    void __fastcall QuitClick(TObject *Sender);
    void __fastcall FormClose(TObject *Sender,
      TCloseAction &Action);
private:        // User declarations
    // Pointer to UserIdent object to be
    // supplied by IdentifyUserForm
    UserIdent* userInfo;
    // Pointer to AppConfig object to be
    // created in SampleAppForm's constructor
    AppConfig* appInfo;
public:      // User declarations
    __fastcall TSampleAppForm(TComponent* Owner);
    // Get and set UserIdent pointers
    UserIdent* getUserInfo(void)
{ return userInfo; }
    void setUserInfo(UserIdent* x)
{ userInfo = x; }
    // Get the AppConfig pointer
    AppConfig* getAppConfig(void)
{ return appInfo; }
};
//---------------------------------------
extern PACKAGE TSampleAppForm *SampleAppForm;
```

```
//---------------------------------------
#endif

// Unit1.cpp
//---------------------------------------
#include
#pragma hdrstop

#include "Unit1.h"
#include "Unit2.h"
#include "Unit3.h"
//---------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TSampleAppForm *SampleAppForm;
//---------------------------------------
__fastcall TSampleAppForm::TSampleAppForm(TComponent* Owner)
    : TForm(Owner)
{
    appInfo = new AppConfig();

    // Next line sets userInfo to NULL. If we try to delete
    // appInfo without having set it, the app could fail.
    userInfo = NULL;   // Just for safety
}
//---------------------------------------
void __fastcall TSampleAppForm::FormShow(TObject *Sender)
{
    // Before we see the main form, collect user info and
    // create TForm1's UserIdent object
    IdentifyUserForm->ShowModal();
}
//---------------------------------------

void __fastcall TSampleAppForm::DemoClick(TObject *Sender)
{
    // Show off the objects we're accessing from the main form.
    DemoForm->ShowModal();
}
//---------------------------------------

void __fastcall TSampleAppForm::QuitClick(TObject *Sender)
{
    Close();
```

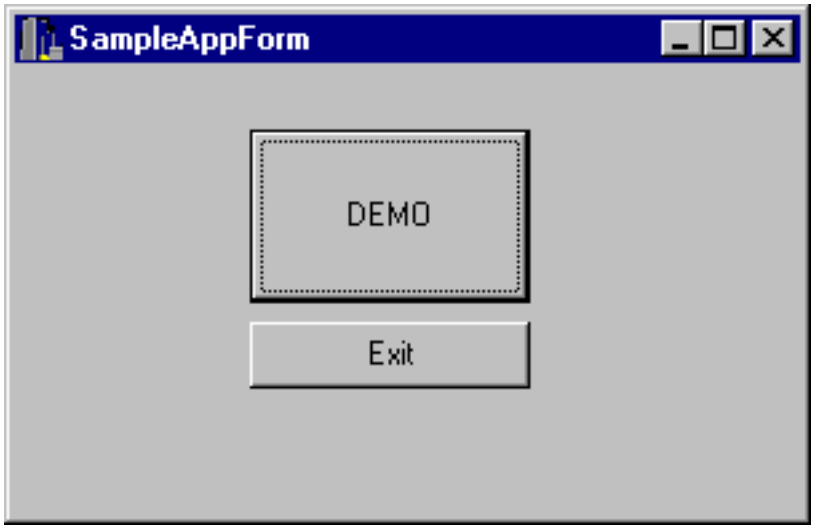
```
}
//---------------------------------------

void __fastcall TSampleAppForm::FormClose(TObject *Sender,
      TCloseAction &Action)
{
    // Delete the memory we used
    delete userInfo;
    delete appInfo;
}
//---------------------------------------
```

After SampleAppForm initializes, it opens an instance of IdentifyUserForm, as shown in Figure A.

Figure A: The main form opens the subform IdentifyUserForm.

Listing B contains the source code for IdentifyUserForm. It collects a name, creates a UserIdent object, and then sets SampleAppForm's UserIdent pointer to that object by calling setUserInfo(). Once IdentifyUserForm closes, the user can open DemoForm by clicking the Demo button located on SampleAppForm, as shown in Figure B.

Figure B: You can click the Demo button on the main form to open DemoForm.



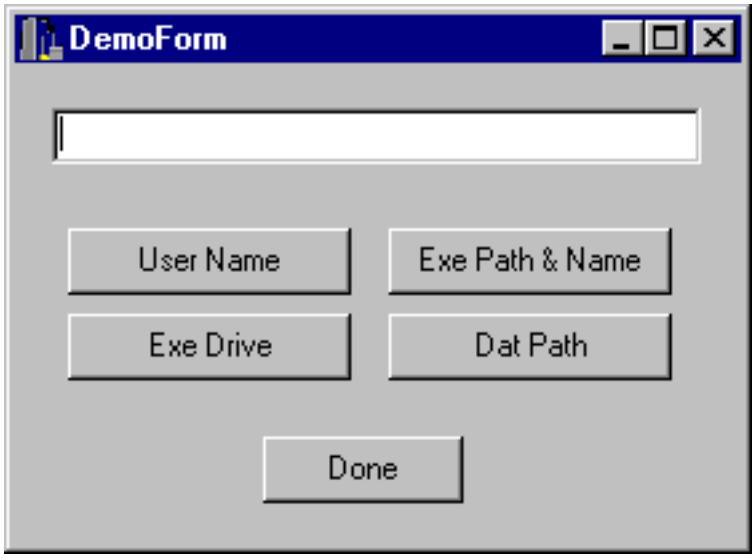
Listing B: IdentifyUserForm

```
// Unit2.h
//---------------------------------------
#ifndef Unit2H
#define Unit2H
//---------------------------------------
#include
#include
#include
#include
#include "Unit1.h"

//---------------------------------------
class TIdentifyUserForm : public TForm
{
  __published:  // IDE-managed Components
    TEdit *Edit1;
    TButton *Ok;
    TLabel *Label1;
    void __fastcall OkClick(TObject *Sender);

  private:      // User declarations
    // Local UserIdent pointer. Before this form closes, we'll
    // pass this pointer to the main form.
    UserIdent *uid;
    TSampleAppForm* sampleAppForm; // Main form
  public:    // User declarations
    __fastcall TIdentifyUserForm(TComponent* Owner);
};
```

```
//---------------------------------------

extern PACKAGE TIdentifyUserForm *IdentifyUserForm;
//---------------------------------------

#endif


// Unit2.cpp
//---------------------------------------
#include
#pragma hdrstop

#include "Unit2.h"
//---------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TIdentifyUserForm *IdentifyUserForm;
//---------------------------------------
__fastcall TIdentifyUserForm::TIdentifyUserForm(
  TComponent* Owner) : TForm(Owner)
{

}
//---------------------------------------

void __fastcall TIdentifyUserForm::OkClick(TObject *Sender)
{
    uid = new UserIdent(Edit1->Text);
    sampleAppForm = (TSampleAppForm*)Application->MainForm;
    sampleAppForm->setUserInfo(uid);
    Close();
}
//---------------------------------------
```

Listing C contains the source code for DemoForm. The form contains one text area and several buttons that retrieve information from one of the objects contained in the main form. As you can see in Figure C, each button's purpose is clearly labeled. The User Name button, for example, calls upon TForm1's UserIdent object to supply the user's name. The Exe Drive button calls the TForm1 AppConfig object's getExeDrive() method to retrieve the letter of the drive where the executable resides.

Figure C: DemoForm gets information from objects on the main form.



Listing C: DemoForm

```
// Unit3.h
//---------------------------------------
#ifndef Unit3H
#define Unit3H
//---------------------------------------

#include
#include
#include
#include
#include "Unit1.h"
//---------------------------------------

class TDemoForm : public TForm
{
  __published:  // IDE-managed Components
    TEdit *Edit1;
    TButton *UserName;
    TButton *ExePath;
    TButton *ExeDrive;
    TButton *DatPath;
    TButton *Done;
    void __fastcall FormShow(TObject *Sender);
    void __fastcall UserNameClick(TObject *Sender);
    void __fastcall ExePathClick(TObject *Sender);
    void __fastcall ExeDriveClick(TObject *Sender);
    void __fastcall DatPathClick(TObject *Sender);
```

```
    void __fastcall DoneClick(TObject *Sender);
  private:  // User declarations
    TSampleAppForm* sampleAppForm; // Main form
  public:   // User declarations
    __fastcall TDemoForm(TComponent* Owner);
};
//---------------------------------------

extern PACKAGE TDemoForm *DemoForm;
//---------------------------------------

#endif


// Unit3.cpp
//---------------------------------------
#include
#pragma hdrstop

#include "Unit3.h"
//---------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TDemoForm *DemoForm;
//---------------------------------------

__fastcall TDemoForm::TDemoForm(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------

void __fastcall TDemoForm::FormShow(TObject *Sender)
{
  // Set up the pointer to our main form
  sampleAppForm = (TSampleAppForm*) Application->MainForm;
}
//---------------------------------------

void __fastcall TDemoForm::UserNameClick(TObject *Sender)
{
  Edit1->Text = sampleAppForm->getUserInfo()->getUserName();
}
//---------------------------------------
```

```
void __fastcall TDemoForm::ExePathClick(TObject *Sender)
{
  Edit1->Text = sampleAppForm->getAppConfig()->getExeName();
}
//---------------------------------------

void __fastcall TDemoForm::ExeDriveClick(TObject *Sender)
{
  Edit1->Text = sampleAppForm->getAppConfig()->getExeDrive();
}
//---------------------------------------

void __fastcall TDemoForm::DatPathClick(TObject *Sender)
{
  Edit1->Text = sampleAppForm->getAppConfig()->getDatPath();
}
//---------------------------------------

void __fastcall TDemoForm::DoneClick(TObject *Sender)
{
  Close();
}
//---------------------------------------
```

Conclusion

You're now armed with techniques that give you the ability to share methods and attributes between forms, making it possible for you to create more efficient and tightly integrated applications. Don't be fooled by the simplicity of the objects we worked within this article- you can easily extend these techniques to handle much more complex applications.

October 1998

Packing database tables

by Mark G. Wiseman

You can download sample files from our Web site as part of the file oct98.zip. Visit www.zdjournals.com/cpb and click the Source Code hyperlink.

Packing database tables…. Do you have a mental picture of going on a trip and deciding to take your favorite database table? First, you stuff the table into your suitcase and then you sit on the suitcase's lid until you can get the latches fastened. Well, that is not exactly what we mean, but the image isn't too far off. In this article, we'll explain what it means to pack a table, then we'll describe how to accomplish table packing (without a suitcase).

Inside the black box

When you add a record to a database table, the database creates a slot for the record in the correct location, inserts the record, and updates any indices the table may have. When the database creates the slot for the record, it may have to enlarge the disk file that holds the table.

Later, when that record is deleted from the table, the database marks the record as deleted-but it doesn't reduce the size of the table's disk file. The two main reasons for this behavior both involve speed. Rather than deleting a record from a table and then shrinking the table's disk file, it's much faster for the database to just set a flag for the newly deleted record, indicating that the record has been deleted. Later, when a new record is added, it's again faster to reuse the slot of a deleted record than it is to expand the table's file on disk and then insert the new record.

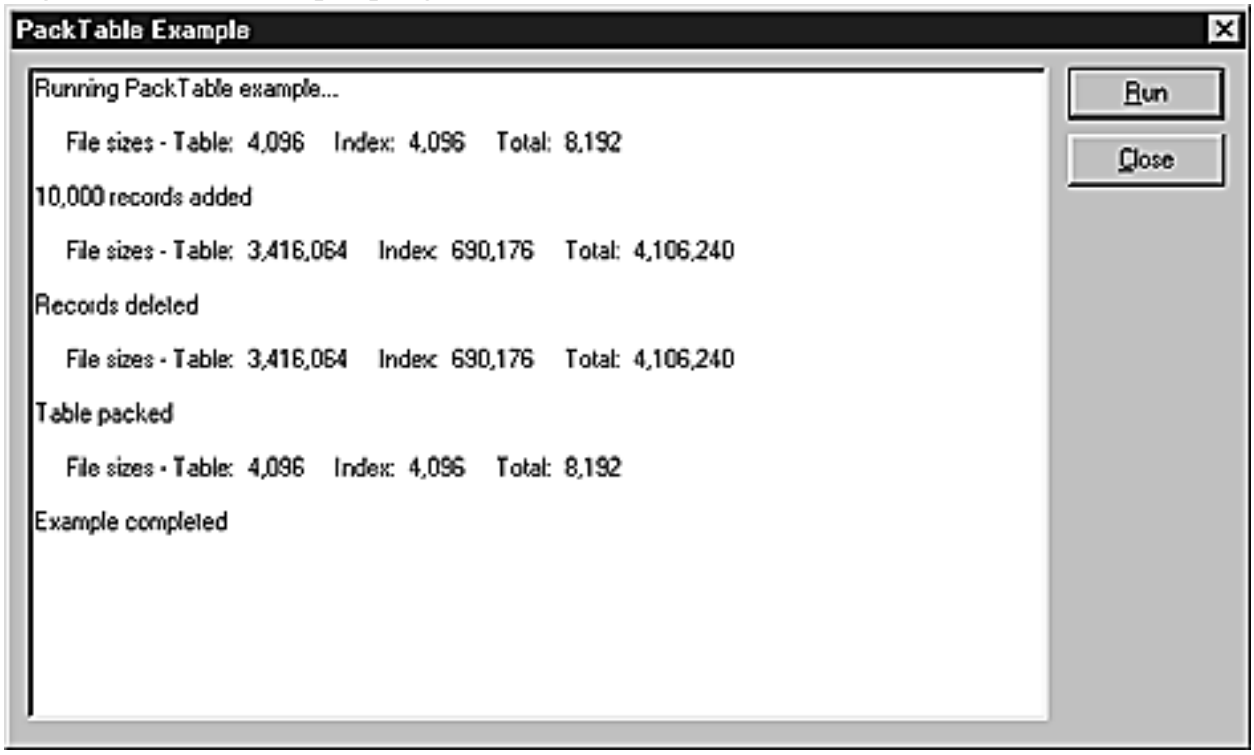
Marking records as deleted without physically deleting them from the file is very similar to the way DOS deletes files from a disk. The file system sets a flag in the file-allocation table indicating that the file has been deleted-but the file actually remains on the disk until it's overwritten by a new file. DOS also does this to improve the speed of the file system. When you pack a table, you force the database to remove the deleted slots from the table's file. As a result, you shrink the size of that file on disk.

So why pack a table?

If marking records as deleted instead of physically altering the table file is faster, why bother packing a table? The answer is simple: size. As a programmer, you aren't concerned only with the speed of your program: You must also keep the program and its associated files to a reasonable size. Doing so will often involve tradeoffs between speed and size in the program's design and implementation. If a program adds many records to a table and then deletes most of those records, the table can take up a lot of unnecessary

disk space. By packing the table, you may slow down the operation of the database, but you'll free disk space that another program may need. Let's consider an example that's similar to packing a table into a suitcase and taking it on a trip. We have a database table that we want to put into a ZIP file and send over the Internet. In this case, reducing the size of the table also increases its speed of transmission-no tradeoffs here. Can the disk-space savings be that dramatic? Yes, they can. Figure A shows our example program (available from our Web site).

Figure A: Our example program demonstrates the dramatic difference in table size after packing a table.



The table in the example program starts with a single record; the table and its index use 8,192 bytes of file space. The program then adds 10,000 records; the table and its index now use 4,106,240 bytes of disk space. Next, the program deletes all but one of the records. This brings the table back to having a single record-but the table and index still occupy 4,106,240 bytes! Finally, the example program packs the table and-whew!-the total size returns to 8,192. Listings A and B contain the code that adds and then deletes records.

Listing A: Code snippet to add 10,000 records

```
const int RECNUM = 10000;

void TMainForm::AddRecords()
   {
   Table->Active = true;

   for (int i = 0; i < RECNUM; i++)
      {
```

```
      Table->Insert();
      TableName->Value = "PackTable Example";
   Table->Post();
   }

   Table->Active = false;
   }
```

Listing B: Code snippet to delete all but one record

```
void TMainForm::DeleteRecords()
   {
   Table->Active = true;

   int delNum = Table->RecordCount - 1;

   Table->First();
   for (int i = 0; i < delNum; i++)
      Table->Delete();

   Table->Active = false;
   }
```

How do we do it?

OK, you're convinced that packing a database table is something you might want to do. You've looked in the online help for the VCL components TTable and TDatabase and found nothing about packing tables. So, how do you pack a table? Listings C and D contain the code for a PackTable() function that will pack Paradox or dBASE tables.

Listing C: The PackTable function header

```
#ifndef PackTableH
#define PackTableH

#include

bool PackTable(TTable *table);
#endif
```

Listing D: The PackTable function

```
#include
#pragma hdrstop

#include "PackTable.h"

bool PackTable(TTable *table)
   {
   bool active = table->Active;
   bool exclusive = table->Exclusive;
   bool retval = true;

   try
      {
      if (!exclusive)
         {
         table->Active = false;
         table->Exclusive = true;
         }
      table->Active = true;

      CURProps props;
      Check(DbiGetCursorProps(table->Handle, props));

      String tableType = props.szTableType;
      if (tableType == szPARADOX)
         {
         CRTblDesc tableDesc;
         memset(&tableDesc, 0, sizeof(tableDesc));
         lstrcpy(tableDesc.szTblName,
           table->TableName.c_str());
         lstrcpy(tableDesc.szTblType, szPARADOX);
         tableDesc.bPack = true;

         hDBIDb hDb = table->DBHandle;
         table->Close();
         Check(DbiDoRestructure(hDb, 1, &tableDesc,
           0, 0, 0, false));
         table->Open();
         }
      else if (tableType == szDBASE)
         Check(DbiPackTable(table->DBHandle,
           table->Handle, 0, szDBASE, true));
      else
         retval = false;
```

```
    }
  catch(...)
    {
    retval = false;
    }

  table->Active = false;
  table->Exclusive = exclusive;
  table->Active = active;

  return(retval);
  }
```

PackTable() makes some calls into the Borland Database Engine (BDE) API. The BDE API includes a function, DbiPackTable(), that will pack a dBASE table-unfortunately, it won't pack a Paradox table. Packing a Paradox table requires the use of another function: DbiDoRestructure(). Using DbiDoRestructure() is more complicated, but it's not too bad. First things first, though.

Tip: Look up BDE API

The online help reference for the BDE API is included with C++Builder; you can find it at \Program Files\Borland\BDE\bde32.hlp. While you're exploring the help file for the BDE API, be sure to look up the function DbiUndeleteRecord()-especially if you're using dBASE tables.

PackTable() takes a single argument, a pointer to a TTable object. If PackTable() successfully packs the table, it returns true. If it can't pack the table, it returns False. Both DbiPackTable() and DbiDoRestructure() require that the table to be packed be open in exclusive mode, meaning that no other programs can have the table open while you pack it. PackTable() first saves the current table settings for the Active and Exclusive properties. If the table isn't in exclusive mode, PackTable() tries to set the Exclusive property to True. If the table is in use by another program, attempting to set Exclusive to True will throw an exception. In this case, PackTable() will catch the exception and return False. Once the table is in exclusive mode, PackTable() calls the BDE API function DbiGetCursorProps(). PackTable() uses information returned by this function to determine the type of the table: Paradox, dBASE, or something else.

If it's a Paradox table, PackTable() sets up a CRTblDesc structure used in the call to DbiDoRestructure(). Most importantly for us, PackTable() sets the bPack member of

CRTblDesc to true. The table must be closed before you call DbiDoRestructure(), so PackTable() closes the table, calls DbiDoRestructure(), and then reopens the table. If it's a dBASE table, PackTable() simply calls DbiPackTable(). If the table is neither a Paradox or dBASE table, PackTable() will return False. Finally, PackTable() resets the Active and Exclusive properties of the table and returns.

As we're packing to go...

We have a few final thoughts. Packing a table will frequently slow down the operation of the table. Also, packing a table that has many deleted records may take several seconds. So, you'll probably want to use PackTable() sparingly. You could call PackTable() every few days or only when you're going to transfer a table by disk or online. Or, you could allow your users to pack the table when they think it's necessary. Don't overuse it, but do use it. As an added bonus, PackTable() also regenerates all the maintained indices of the table. It does so as a by-product of calling either DbiDoRestructure() or DbiPackTable(). If you're using a table format other than Paradox or dBASE, you may be able to enhance PackTable() to work with that format also. For example, it should be possible to pack Microsoft Access tables using COM.

Finally, we may be packing, but we'll be back. In our next article on this subject, we'll build a component that incorporates PackTable() and will pack all the tables in a database.

October 1998

Low-level wave audio, part 3

by Kent Reisdorph

Sample files are available from our Web site as part of the file oct98.zip. Visit www.zdjournals.com/cpb and click the Source Code hyperlink.

This month, we finish our series on low-level audio; the previous two articles appeared in the July and August 1998 issues of *C++Builder Developer's Journal*. In this installment, we'll show you how to record wave audio using the low-level audio functions. Those functions include waveInOpen, waveInPrepareHeader, waveInAddBuffer, waveInStart, waveInUnprepareHeader, and waveInClose. Recording audio is only half the story, of course. You'll ultimately have to save the recorded data to disk (at least, in most applications). Our example application shows you how to save data after you record it.

Recording wave data

Recording wave data with the low-level interface is only slightly more complicated than playing wave data, as we showed you in Part 2 of this series. Here are the steps required:

- Set the wave format.
- Open the wave input device.
- Allocate a buffer to hold the wave data.
- Prepare the wave header.
- Start recording.
- Close the device when recording finishes.

In the following sections, we'll explain each of these steps in detail.

Set the wave format

Before you open the device, you must set the recording parameters (wave format) for the wave data. You do so by filling out an instance of the WAVEFORMATEX structure. Wave data is in pulse code modulation (PCM) format. PCM data has three properties that determine the quality of the recorded sound: the number of bits per sample, the number of samples per second, and the number of channels. You can set the bits-per-sample element of the equation to either 8 or 16. The number of samples per second is usually 8,000; 11,025; 22,050; or 44,100. The number of channels will be either 1 (monaural) or 2 (stereo).

The size of the wave data stored on disk is directly proportional to the quality of the audio. Ten seconds of audio recorded at 8 kHz, mono, and 8 bits per sample will produce a WAV file

about 79KB in size. Ten seconds of audio recorded at 44.1 kHz, stereo, and 16 bits per sample, however, will result in a file that is 862KB in size. Obviously, you want the wave audio quality to be good--but be careful of going overboard. In general, a wave format of 22.05 kHz, 8 bits per sample, and mono provides a reasonable compromise between sound quality and wave data size (the Windows system sounds are recorded using that format).

After you've determined the wave format, you need to fill out the WAVEFORMATEX structure. Here's the code:

```
// class member variable
WAVEFORMATEX WaveFormat;

// later...
WaveFormat.wFormatTag      =
  WAVE_FORMAT_PCM;
WaveFormat.nChannels       = 1;
WaveFormat.nSamplesPerSec  = 22050;
WaveFormat.wBitsPerSample  = 8;
WaveFormat.nAvgBytesPerSec = 22050;
WaveFormat.nBlockAlign     = 1;
WaveFormat.cbSize          = 0;
```

Notice that the wFormatTag member is set to WAVE_FORMAT_PCM. This is the wave format for Windows WAV files (other wave formats are defined in MMREG.H, if you want to take a look). The nChannels, nSamplesPerSec, and nBitsPerSample members are set as described in the preceding paragraphs on PCM formats. The nAvgBytesPerSec member is set to the average number of bytes recorded (or *sampled*) per second. This value is determined by the following formula:

```
SamplesPerSecond * Channels
```

The nBlockAlign data member also requires explanation. You determine the value for the block alignment with this formula:

```
(Channels * BitsPerSample) / 8
```

The cbSize data member specifies the number of extra bytes of data stored with the wave

format header. This value isn't typically used when recording wave data.

Open the wave input device

Opening the wave input device is nearly identical to opening the wave output device in Part 2 of this series. Before you open the wave device, though, you should query the device to be sure it supports the format you've selected. You do so by calling waveInOpen() with the WAVE_FORMAT_QUERY flag. For example:

```
int Res = waveInOpen(&WaveHandle,
  WAVE_MAPPER, &WaveFormat, 0, 0,
  WAVE_FORMAT_QUERY);
if (Res == WAVERR_BADFORMAT) return;
```

If the specified wave format (passed in the WaveFormat structure in the third parameter) is compatible with the wave device, waveInOpen() will return 0. If the format is incompatible with the wave device, waveInOpen() will return WAVERR_BADFORMAT. When you query the device, Windows checks the device's capabilities, but doesn't open the device. Querying the device gives you a chance to abort a recording operation if the specified wave format is invalid. The rest of the parameters for waveInOpen() are identical to those of waveOutOpen(), as in Part 2.

Now that you know the requested format is valid, you can open the wave input device. Here's the code:

```
Res = waveInOpen(
  &WaveHandle, WAVE_MAPPER, &WaveFormat,
  MAKELONG(Handle, 0), 0, CALLBACK_WINDOW);
```

In this example, WaveHandle is a variable (of type HWAVEIN) that will receive a handle to the device if the device is opened successfully. The WAVE_MAPPER constant tells Windows to use the first available wave input device on the system that can support the specified format (usually the sound card). The fourth and sixth parameters of waveInOpen() tell Windows to send wave input messages to the form's window procedure. The fifth parameter passes any additional data to the application when Windows sends the wave-in messages. We aren't using any additional data, so we pass 0 for this parameter. If waveInOpen() returns 0, you can proceed with the next step: creating the wave input buffer.

Note: Checking return values

All of the code in this article assigns the return value from the various wave-input functions to a variable called Res. The code examples don't show our error-checking code, for brevity's sake--but you should always check the return values in your own code and take appropriate action if you encounter an error.

Allocate a buffer for the wave data

Next, you allocate a buffer to hold the wave data. First, you must know how much memory to allocate for the buffer. You can calculate the buffer size based on the number of seconds of data you want to record and the recording format, using this formula:

```
RecordSeconds * AvgerageBytesPerSecond
```

For example, if you want to record 10 seconds of data at 22.05 kHz, mono, 8-bit, you'll use this code to allocate the buffer:

```
// class member variables
char* WaveData;
int BufferSize;

BufferSize = 10 * 22050;
WaveData = new char[BufferSize];
```

Remember to keep track of your allocations and deallocations so that your program doesn't leak memory.

Prepare the wave header

Now you're ready to prepare the wave input header, a structure that contains the information Windows needs to carry out the recording operation. Specifically, it holds the size of the wave data buffer and a pointer to the buffer itself. (Don't confuse the wave input header with the wave format header.) The wave input header is an instance of a WAVEHDR structure--this structure is used when performing both playback and recording, so you can

ignore many of its data members when recording wave data. You need to set these data members: dwBufferLength, dwFlags, and lpBuffer. For example:

```
//class member variable
WAVEHDR WaveHeader;

WaveHeader.dwBufferLength = BufferSize;
WaveHeader.dwFlags        = 0;
WaveHeader.lpData         = WaveData;
```

Notice that the dwBufferLength and lpData members are assigned values obtained when we allocated the wave data buffer in the previous step. You must set the dwFlags parameter to 0 before recording. Now that the wave input header is set up, you can prepare it for use with the waveInPrepareHeader() function:

```
Res = waveInPrepareHeader(WaveHandle,
  &WaveHeader, sizeof(WAVEHDR));
```

If waveInPrepareHeader() returns 0, the wave header was successfully prepared and is ready for use. To implement the wave header, call the waveInAddBuffer() function, as follows:

```
Res = waveInAddBuffer(WaveHandle,
  &WaveHeader, sizeof(WAVEHDR));
```

This function adds the buffer specified in the wave header to the list of buffers that will be played. We're using only one buffer in this case, but this step is still required.

Start recording

You're ready to start recording. You do so by calling the waveInStart() function, passing the handle to the wave input device (obtained when we called waveInOpen()). This part is simple:

```
Res = waveInStart(WaveHandle);
```

The waveInStart() function starts recording and immediately returns. If waveInStart() returns 0, recording has started and control is immediately returned to the calling application. Put another way, the recording process happens asynchronously--recording starts and your application is free to go about its business while recording is taking place. This gives you the ability to let the user stop the recording in response to a button click or some other event.

Catching wave-in messages

In order to do something useful with the wave data, you need to know when recording has finished. To accomplish this, you'll catch the Windows messages that correspond to wave audio input.

Wave-in messages

In Part 2 of this series, we talked about the wave-out messages Windows sends when playing a wave file. Naturally, there are corresponding messages for wave recording. Table A lists those messages.

Table A: Windows wave-in messages

Message	Description
MM_WIM_OPEN	Device has been opened.
MM_WIM_CLOSE	Device has been closed.
MM_WIM_DATA	Recording has finished and the input buffer is being returned to the application.

Of these messages, you're most likely to be concerned with MM_WIM_DATA. In fact, you *must* respond to this message if you're going to do anything useful with the recorded wave data. MM_WIM_DATA is received when the wave buffer is being returned to the calling application. This can happen as the result of two primary events: Either the wave input buffer has filled up, or the recording operation was interrupted. In either case, this message notifies you that the wave recording operation has finished. At that point, you can save the wave data to a file.

Catching the MM_WIM_DATA message

You catch the MM_WIM_DATA message by implementing a C++Builder message map. The message map looks like this:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(
    MM_WIM_DATA, TMessage, OnWaveMessage)
END_MESSAGE_MAP(TForm)
```

The OnWaveMessage() message handler will be called when the MM_WIM_DATA message is received. At that point, you'll take appropriate action, such as saving the wave data to a file. Let's look at that next.

The MM_WIM_DATA message handler

Your message handler for the MM_WIM_DATA message needs to do three things:

- Close the wave input device.
- Save the wave data to disk.
- Free the memory allocated for the wave input buffer.

Here's how your OnWaveMessage method will look:

```
void TMainForm::OnWaveMessage(
  TMessage& msg)
{
  if (msg.Msg == MM_WIM_DATA) {
    // close the wave device
    waveInClose(WaveHandle);
    // save the wave data
    SaveWaveFile();
    // Free the memory for the wave buffer.
    WaveHeader.lpData = 0;
    delete[] WaveData;
    WaveData = 0;
  }
}
```

This code is straightforward. First, the waveInClose() function closes the wave input device using the wave handle obtained when the device was opened. Next, the SaveWaveFile() function saves the wave data to disk (we'll discuss that next). Finally, the memory allocated for the wave buffer is freed. The MM_WIM_DATA message handler is simple, but it's a vital part of the wave recording operation.

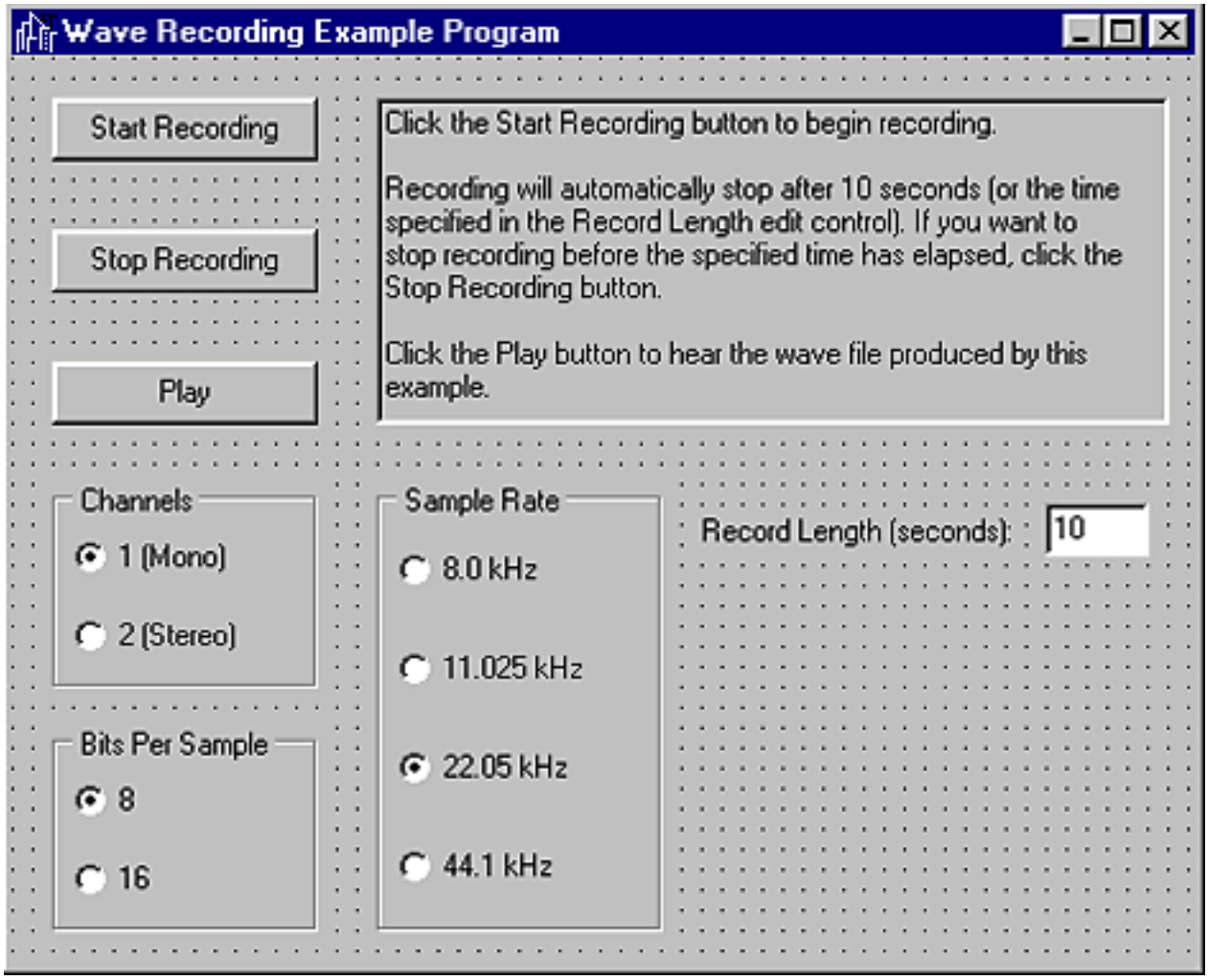
Saving the wave data to disk

We covered saving the wave data to disk in Part 1 of this series when we discussed RIFF files. You need to create a RIFF file, write out the wave format header, and then write the actual wave data to the file. We won't explain each of these steps again here--refer to that article, and to the SaveWaveFile() function in Listing B. Also see "RIFF Update from Part 1" in this issue.

Putting it all together

Listings A and B contain the header file and source code for an example program that records a wave file. Figure A shows the program window.

Figure A: Our sample program lets you record wave files.



The example lets you set the recording length and parameters (the wave format). To begin, click Start Recording. Recording will automatically stop after the specified number of seconds has elapsed (10 seconds by default). To stop recording before the specified time is up, click Stop Recording. After you finish recording, you can play the wave file by clicking

Play.

To test this program, you'll need a microphone attached to your sound card. It would be a good idea to record a wave file using Windows Sound Recorder, so you know the sound card and microphone are functioning properly before attempting to run our program.

That's it! Using the tools we've provided in this article series, you're ready and able to control the recording and playback of wave audio data in your applications.

Listing A: RecWaveU.H

```
//---------------------------------------------
#ifndef RecWaveUH
#define RecWaveUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <mmsystem.h>
#include <ExtCtrls.hpp>
//---------------------------------------------
class TMainForm : public TForm
{
__published:  // IDE-managed Components
  TButton *StartBtn;
  TButton *StopBtn;
  TButton *PlayBtn;
  TMemo *Memo1;
  TRadioGroup *ChannelsGroup;
  TRadioGroup *BitsGroup;
  TRadioGroup *SamplesGroup;
  TEdit *SecondsEdit;
  TLabel *Label1;
  void __fastcall PlayBtnClick(TObject *Sender);
  void __fastcall StartBtnClick(TObject*Sender);
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall StopBtnClick(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
private:  // User declarations
private:  // User declarations
  char* WaveData;
  HWAVEIN WaveHandle;
```

```
  int DataSize;
  void CheckMMIOError(DWORD code);
  void OnWaveMessage(TMessage& msg);
  void CheckWaveError(DWORD code);
  void SaveWaveFile();
  WAVEHDR WaveHeader;
  WAVEFORMATEX WaveFormat;
public:    // User declarations
  __fastcall TMainForm(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(MM_WIM_DATA, TMessage, OnWaveMessage)
  END_MESSAGE_MAP(TForm)
};
//---------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------
#endif
```

Listing B: RecWaveU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "RecWaveU.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------
__fastcall TMainForm::TMainForm
  (TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall
TMainForm::StartBtnClick(TObject *Sender)
{
  // Get recording parameters from radio group boxes on form.
  int samplesPerSec = (int)SamplesGroup->
    Items->Objects[SamplesGroup->ItemIndex];
  WORD channels = (WORD)(ChannelsGroup->ItemIndex + 1);
  WORD bitsPerSample = WORD(8 * (BitsGroup->ItemIndex + 1));
```

```
DWORD avgBytesPerSec = channels * samplesPerSec;

// Fill in the WAVEFORMATEX header.
WaveFormat.wFormatTag      = WAVE_FORMAT_PCM;
WaveFormat.nChannels       = channels;
WaveFormat.nSamplesPerSec  = samplesPerSec;
WaveFormat.nAvgBytesPerSec = avgBytesPerSec;
WaveFormat.nBlockAlign     =
  WORD((channels * bitsPerSample) / 8);
WaveFormat.wBitsPerSample  = bitsPerSample;
WaveFormat.cbSize          = 0;

// Query device to see if it supports the selected format.
int Res = waveInOpen(&WaveHandle, WAVE_MAPPER,
  &WaveFormat, 0, 0, WAVE_FORMAT_QUERY);
CheckWaveError(Res);
if (Res == WAVERR_BADFORMAT)
  return;

// Open device. Wave-in messages go to window proc of form.
Res = waveInOpen(&WaveHandle, WAVE_MAPPER, &WaveFormat,
  MAKELONG(Handle, 0), 0, CALLBACK_WINDOW);
CheckWaveError(Res);

// Allocate buffer for wave data large enough to hold data.
int seconds = SecondsEdit->Text.ToIntDef(10);
int bufferSize = seconds * avgBytesPerSec;
if (WaveData)
  delete[] WaveData;
WaveData = new char[bufferSize];

// Set up WaveHeader structure.
WaveHeader.dwBufferLength = bufferSize;
WaveHeader.dwFlags        = 0;
WaveHeader.lpData = WaveData;

// Prepare the header.
Res = waveInPrepareHeader(
  WaveHandle, &WaveHeader, sizeof(WAVEHDR));
CheckWaveError(Res);
Res = waveInAddBuffer(
  WaveHandle, &WaveHeader, sizeof(WAVEHDR));

// Error. Free the memory and exit.
```

```
  if (Res != 0) {
    waveInUnprepareHeader(
      WaveHandle, &WaveHeader, sizeof(WAVEHDR));
    delete[] WaveData;
    WaveData = 0;
    return;
  }

  // Start recording.
  StopBtn->Enabled = true;
  Res = waveInStart(WaveHandle);
  CheckWaveError(Res);
}
//---------------------------------------------
void __fastcall
TMainForm::StopBtnClick(TObject *Sender)
{
  // Call waveInReset() to stop recording. Forcec Windows to send
  // the MM_WIM_DATA message to the application.
  waveInReset(WaveHandle);
}
//---------------------------------------------
void __fastcall
TMainForm::PlayBtnClick(TObject *Sender)
{
  StartBtn->Enabled = false;
  StopBtn->Enabled = false;
  // Could have used low-level audio functions to play wave file
  // but PlaySound is much easier.
  PlaySound("test.wav", 0, SND_FILENAME);
  StartBtn->Enabled = true;
  StopBtn->Enabled = true;
}
//---------------------------------------------
void __fastcall
TMainForm::FormCreate(TObject *Sender)
{
  // Assign the sample rates that correspond to each radio button
  // in the Samples Per Second radio group.
  SamplesGroup->Items->Objects[0] = (TObject*)8000;
  SamplesGroup->Items->Objects[1] = (TObject*)11025;
  SamplesGroup->Items->Objects[2] = (TObject*)22050;
  SamplesGroup->Items->Objects[3] = (TObject*)44100;
```

```
  WaveData = 0;
}
//---------------------------------------------
void __fastcall
TMainForm::FormDestroy(TObject *Sender)
{
  // Just in case.
  if (WaveData)
    delete[] WaveData;
}
//---------------------------------------------
void TMainForm::OnWaveMessage(TMessage& msg)
{
  // Record buffer is full so free the memory allocated for the
  // buffer. After that write out the file.
  if (msg.Msg == MM_WIM_DATA) {
    waveInClose(WaveHandle);
    SaveWaveFile();
    WaveHeader.lpData = 0;
    if (WaveData) {
      delete[] WaveData;
      WaveData = 0;
    }
    StartBtn->Enabled = true;
    StopBtn->Enabled = false;
    PlayBtn->Enabled = true;
  }
}
//---------------------------------------------
void TMainForm::CheckWaveError(DWORD code)
{
  if (code == 0) return;
  char buff[256];
  // Report a wave out error, if one occurred.
  waveInGetErrorText(code, buff, sizeof(buff));
  MessageBox(Handle, buff, "Wave Error", MB_OK);
}

void TMainForm::CheckMMIOError(DWORD code)
{
  // Report an mmio error, if one occurred.
  if (code == 0) return;
  char buff[256];
```

```
  wsprintf(buff,
    "MMIO Error. Error Code: %d", code);
  Application->MessageBox(buff, "MMIO Error", 0);
}

void TMainForm::SaveWaveFile()
{
  // Declare the structures we'll need.
  MMCKINFO ChunkInfo;
  MMCKINFO FormatChunkInfo;
  MMCKINFO DataChunkInfo;

  // Open the file.
  HMMIO handle = mmioOpen(
    "test.wav", 0, MMIO_CREATE | MMIO_WRITE);
  if (!handle) {
    MessageBox(0, "Error creating file.", "Error Message", 0);
    return;
  }

  // Create RIFF chunk. First zero out ChunkInfo structure.
  memset(&ChunkInfo, 0, sizeof(MMCKINFO));
  ChunkInfo.fccType = mmioStringToFOURCC("WAVE", 0);
  DWORD Res = mmioCreateChunk(
    handle, &ChunkInfo, MMIO_CREATERIFF);
  CheckMMIOError(Res);

  // Create the format chunk.
  FormatChunkInfo.ckid = mmioStringToFOURCC("fmt ", 0);
  FormatChunkInfo.cksize = sizeof(WAVEFORMATEX);
  Res = mmioCreateChunk(handle, &FormatChunkInfo, 0);
  CheckMMIOError(Res);
  // Write the wave format data.
  mmioWrite(handle, (char*)&WaveFormat, sizeof(WaveFormat));

  // Create the data chunk.
  Res = mmioAscend(handle, &FormatChunkInfo, 0);
  CheckMMIOError(Res);
  DataChunkInfo.ckid = mmioStringToFOURCC("data", 0);
  DataSize = WaveHeader.dwBytesRecorded;
  DataChunkInfo.cksize = DataSize;
  Res = mmioCreateChunk(handle, &DataChunkInfo, 0);
  CheckMMIOError(Res);
  // Write the data.
```

```
  mmioWrite(handle, (char*)WaveHeader.lpData, DataSize);
  // Ascend out of the data chunk.
  mmioAscend(handle, &DataChunkInfo, 0);

  // Ascend out of the RIFF chunk (the main chunk). Failure to do
  // this will result in a file that is unreadable by Windows95
  // Sound Recorder.
  mmioAscend(handle, &ChunkInfo, 0);
  mmioClose(handle, 0);
}
```

October 1998

RIFF update from Part 1

by Kent Reisdorph

Part 1 of this article series contained some misleading information. That article said, in part:

Notice the line that writes the wave format header. It looks like this:

```
mmioWrite(handle, (char*)&waveFmt,
  sizeof(WAVEFORMATEX) - 2);
```

Notice that we subtract 2 from the size of a WAVEFORMATEX structure when we write the structure. If we don't do this then the wave file we created won't be able to be played by the Sound Recorder program that comes with Windows95.

That statement isn't correct. Wave-file corruption can occur, but the corruption doesn't stem from writing the incorrect number of bytes for the wave header--rather, it comes from not properly ascending out of the individual chunks of a RIFF file before closing the file. When you ascend out of a RIFF file chunk, Windows will automatically update the chunk info with the size of the data in the chunk. If you neglect to ascend out of the root chunk, the root chunk data size isn't updated with the total size of the file. When Windows95 encounters a file with a RIFF chunk data size of 0, it reports the file as corrupted (even though the wave data itself is intact). The SaveWaveFile() function in Listing B of the accompanying article shows the correct way to ascend out of the root chunk before closing the wave file.

September 1998

Using a VCL form in a DLL

by Kent Reisdorph

One important feature of Windows is the ability it gives you to put code in a dynamic link library, or DLL. DLLs let you compartmentalize your code, providing flexibility when you're performing updates. For example, if you place different aspects of your application in DLLs, then updating an aspect of your program is as easy as shipping a new DLL. In this article, we'll explain how to create DLLs that contain VCL forms. We'll show you how to write a DLL so that your forms can be used by C++Builder programs or even by programs written with tools such as Borland C++, Visual C++, or Delphi.

Import, export, and __declspec

Perhaps one of the most frustrating aspects of dealing with DLLs is the concept of importing and exporting functions. When you build a DLL, you need to declare all public functions with the __export keyword. (*Public functions* are called from your calling application. A DLL may also contain *non-public functions* used only within the DLL.) Conversely, when you build the application that uses functions in the DLL, you need to declare the functions with the __import keyword. This process sounds simple, but it's easy to get wrong. You must place the __import or __export symbol in exactly the right place in your function declaration, or your application won't compile. Fortunately, the __declspec keyword cures this ill. To export a function from the DLL, use either of the following:

```
int __declspec(dllexport)
        MyFunction();
__declspec(dllexport)
        int MyFunction();
```

Notice that you can place __declspec either before or after the function's return type (int in this example). The compiler straightens things out at compile time. To import a function, use either of the following:

```
int __declspec(dllimport)
         MyFunction();
__declspec(dllimport)
        int MyFunction();
```

Again, the placement of __declspec isn't critical, as long as it falls somewhere before the function name. To simplify importing or exporting functions, you'll generally use a symbol that expands to __import or __export depending on whether you're building the DLL or the calling

application. Let's look at an example before we explain further:

```
#ifdef BUILDDLL
  #define DLL_MODE
        __declspec(dllexport)
#else
  #ifdef BUILDAPP
    #define DLL_MODE
         __declspec(dllimport)
  #endif
#endif

extern "C" int
        DLL_MODE MyFunction();
```

Place this code in the header file for your DLL. In the source code for your DLL, you define BUILDDLL before you include the header:

```
#define BUILDDLL
#include "MyDll.h"
```

When you define BUILDLL, the DLL_MODE symbol is defined as __declspec(dllexport), thereby exporting the symbols. Naturally, when you build the calling application you'll write the following lines:

```
#define BUILDAPP
#include "MyDll.h"
```

Now, DLL_MODE will be defined as __declspec(dllimport) and the functions will be declared as imported. This technique greatly simplifies importing and exporting functions from DLLs. All you have to remember to do is define either BUILDDLL or BUILDAPP in the DLL code or the calling application code, respectively.

Calling conventions and name mangling

Another confusing aspect of imported and exported functions is the *calling convention*. The calling convention determines how the exported function names appear in DLL and how the parameters are passed on the stack. The different calling conventions and their keyword

modifier appear in Table A.

Table A: Function calling conventions

Convention	Keyword
C	__cdecl
Pascal	__pascal
Register	__fastcall
Standard call	__stdcall

By default, a DLL built with C++Builder will use the C calling convention. This means function names will be case sensitive and will have an underscore prefix. For example, the function in the previous example will show up in the DLL's export section as

```
_MyFunction
```

If you're creating DLLs for use only with C++Builder applications, you don't need to worry about this convention. If, however, you're creating DLLs for use with other development environments, you may wish to use the Standard Call convention to eliminate the leading underscore. The important thing is to use the same calling convention when you build the DLL and when you import the functions in the calling application. If you decide to use the Standard Call convention, you need to use the __stdcall keyword in both the function declaration and the function definition:

```
// declaration
extern "C"
int __decldpec(DLL_MODE)
        __stdcall
  MyFunction();

// function definition
int __stdcall MyFunction()
{
  // function body
}
```

You'll notice two things about this code. First, the __declspec modifier is used only in the function declaration and not in the function definition. Next, the code uses the extern "C" modifier, thereby preventing the function name from being mangled. C++ compiler vendors mangle function names to account for function overloading--you must use the extern "C" modifier if your DLL will be used from applications built with development tools other than

C++Builder. Without this modifier, other applications won't be able to "see" the functions in your DLL. One extra note concerning extern "C": If you do use this modifier, you won't be able to use overloaded functions in your DLL.

VCL forms in DLLs

With that critical bit of DLL groundwork behind us, we can get on with how to use VCL forms in a DLL. The most basic scenario is a VCL application calling a VCL form that resides in a DLL. This scenario is fairly easy to handle. The steps required are as follows:

- Create a new DLL project.
- Create a header file for the DLL.
- Create a form.
- Write an exported function that will show the form.
- Run IMPLIB.EXE on the DLL to create an import library file.
- Add the resulting LIB file to the calling application's project.
- Call the DLL function to show the form.

Let's look at these steps on more detail.

Create a DLL project and a header

To create a new DLL project, choose File | New from the main menu and double-click on the DLL icon in the Object Repository. Next, you need to create a header for your DLL that will contain the function declarations for all functions exported from your DLL. When you create a new DLL project, C++Builder doesn't automatically create a header file as it does when you create a source code unit. To create a header for your DLL, create a new text file from the Object Repository and save the file with a .H extension. Be sure your header includes a sentry and the import/export macros we presented in the section "Import, export, __declspec, and you."

A *sentry* is a mechanism that prevents your header from being included more than once in a project. A sentry looks like this:

```
#ifndef __MYDLL_H
#define __MYDLL_H

// header code

#endif
```

Normally, C++Builder does this for you when you create a new unit. But when you create a

header from scratch, you must add the sentry code yourself. Be sure to add a line near the top of your DLL's main unit to include your header, and remember to define the BUILDDLL symbol as we discussed earlier.

Create a form in the DLL

You create a new form for a DLL the same way you create a new form for an application. There is, however, one difference: The Include Unit Hdr option isn't available from the File menu in a DLL project. This means you'll have to manually type the code to include the units for any forms you create. Let's say you have a form in a unit saved as MYFORM.CPP. Let's further assume that your DLL is called MYDLL.CPP. The code to include the header for the DLL (created in the previous step) and the code to include the new unit would look like this:

```
#include <vcl\vcl.h>
#pragma hdrstop

#define BUILDDLL
#include "MyDll.h"
#include "MyForm.h"
```

In this example, C++Builder creates the first two lines, leaving you to type the remaining three lines. Remember to manually include the headers for any forms you create for the DLL. If you forget this step, the compiler will remind you by complaining about undefined symbols.

Write a function to show the form

Now you're ready to enter the code that will show the form. The following code shows both the function declaration (contained in the header) and the function definition (in the DLL's source unit):

```
// declaration in header
extern "C"
int DLL_MODE ShowDll
        FormModal(TForm* parent);

// definition in CPP file
int ShowDllFormModal
        (TForm* parent)
{
  TMyForm* form = new
        TMyForm(parent);
  int result = form->
```

```
        ShowModal();
  delete form;
  return result;
}
```

This code creates a new TMyForm object with the given parent, shows the form modally, and then deletes the form. The modal result of the ShowModal() call is returned to the calling application.

Create an import library file for the DLL and add it to your project

At this point, you can build the DLL. After you've done that, you'll have a DLL file in your DLL project's directory. Before you can do anything with the DLL in the calling application, you need to create an import library file for your DLL. To do so, go to a command prompt, move to your DLL project's directory, and type

```
implib mydll.lib mydll.dll
```

to create MYDLL.LIB in the current directory. The IMPLIB utility is located in your CBUILDER\BIN directory. That directory is already on your system's path, so you don't need to type the entire path to IMPLIB.EXE.

Create the calling application

Now that you have an import library file, you can create the calling application. First, add the import library file to your application's project using Add To Project. Doing so will cause the DLL to be loaded when the application starts. The only step remaining is to call the function in the DLL that will show the form. That code looks like the following:

```
if (ShowDllFormModal(this))
  // do something here
else
  // do something else
```

You could also write a separate function in the DLL to call form->Show() rather than form->ShowModal(). Doing so would allow you to show the form as a modeless form. That's all there is to it! You have a form in a DLL that you can call from any C++Builder application.

Note: Build without the incremental linker.

Be sure to turn off the incremental linker and debug information before you do a final build of your DLL. If you don't do a build without the incremental linker, your DLL will contain hard-coded path information that will make it nearly impossible to utilize on your users' machines.

Calling a VCL form from a non-VCL application

You've already done the work to show a VCL form from a non-VCL application. What remains is to describe some of the reasons behind what we've done so far. Remember how you declared the DLL function? It looked like this:

```
extern "C"
int DLL_MODE ShowDll
        FormModal(TForm* parent);
```

The extern "C" wasn't strictly necessary because the function was being called from a C++Builder application. But using extern "C" means you're ready to go if you want to use your DLL with non-C++Builder applications. Also, notice the function parameter. Passing a TForm pointer makes no sense for a non-VCL calling application (a Visual Basic application, for example, has no understanding of TForm). If your application will be used only from non-VCL applications, you should write the function with no parameters. In the function, you can pass 0 for the Owner parameter when you create the TForm object. For example:

```
int ShowVCLForm()
{
  TDLLForm* form = new TDLLForm(0);
  int result = form->ShowModal();
  delete form;
  return result;
 }
```

Specifying an Owner of 0 causes VCL to create the form without a parent object--in this case, VCL automatically parents the form to the process that created it. This feature is good news because you can call a form from a non-VCL application and the form will behave modally--but it's bad news because you can't show the form modelessly. You can try calling Show() rather than ShowModal(), but as long as the parent of the form is 0 the form will always behave modally.

The only other consideration, when creating a DLL that will be called from non-VCL applications, is the calling convention, as we described earlier. You may want to consider using the Standard Call convention to eliminate the problem with leading underscores in the function name. Doing so will make calling functions in your DLL more straightforward for

applications written with tools other than C++Builder.

An example is worth a thousand words

As always, an example will help to solidify the concepts presented in this article. Listings A and B display the code for a DLL that contains two forms: a regular form and an MDI child form (see the companion article, "MDI Child Forms in a DLL"). To save space, we didn't include the code for the forms, but any forms will do.

Listing A: MYFORMS.H

```
#ifndef __MYFORMS_H
#define __MYFORMS_H

#ifdef BUILDDLL
  #define DLL_MODE __declspec(dllexport)
#else
  #ifdef BUILDAPP
    #define DLL_MODE __declspec(dllimport)
  #endif
#endif

// These functions are exported only when
// building a BCB application.
#ifdef BCB
  int DLL_MODE ShowDllFormModal(TForm* parent);

  void DLL_MODE ShowDllForm(TForm* parent);

  void DLL_MODE ShowMDIChildForm (TApplication* app);

  void DLL_MODE ResetDllApplication();
#endif

// This function is exported regardless.
  extern "C" int DLL_MODE ShowVCLForm();

#endif
```

Listing B: MYFORMS.CPP

```
#include <vcl\vcl.h>
#pragma hdrstop
```

```
TApplication* DllApp = 0;

//---------------------------------------------
#define BCB
#define BUILDDLL
#include "MyForms.h"
#include "FormOne.h"
#include "MDIChild.h"
//---------------------------------------------
USEFORM("FormOne.cpp", DLLForm);
USEFORM("MDIChild.cpp", ChildForm);
//---------------------------------------------
int WINAPI DllEntryPoint(
  HINSTANCE hinst, unsigned
          long reason, void*)
{
  return 1;
}
//---------------------------------------------
int ShowDllFormModal(TForm* parent)
{
  // Create the form and show it modally.
  TDLLForm* form = new TDLLForm(parent);
  int result = form->ShowModal();
  delete form;
  return result;
}

void ShowDllForm(TForm* parent)
{
  // Create the form and show it modelessly.
  // The calling application will free the
  // memory used by the modeless form. Don't
  // call this function from a non-VCL app
  // or it will leak memory.
  TDLLForm* form = new TDLLForm(parent);
  form->Show();
}

int ShowVCLForm()
{
  // Create the form and show it modally.
  return ShowDllFormModal(0);
```

```
}

void ShowMDIChildForm
        (TApplication* mainApp)
{
  // If the Application object for the DLL has
  // not yet been saved then save it now and
  // assign the Application object for the
  // calling application to the DLL's
  // Application object.
  if (!DllApp) {
    DllApp = Application;
    Application = mainApp;
  }

  // Create and show the MDI child form.
  TChildForm* child = new TChildForm (Application->MainForm);
  child->Show();
}

void ResetDllApplication()
{
  // Reset the saved DLL's Application object.
  // You must do this before the calling
  // application closes or you will get an
  // access violation.
  if (DllApp) {
    Application = DllApp;
  }
}
```

The DLL project creates a DLL called MYFORMS.DLL. If you enter the code by hand, remember to run IMPLIB on the final DLL to create an import library file that the calling application will need.

Code included in the header in Listing A lets either a C++Builder application or a non-C++Builder application use the header. The extra code makes use of a symbol we created called BCB. If BCB isn't defined, the functions requiring VCL objects (like TForm and TApplication) aren't exported.

Listings C and D contain the code for the calling application. The main form of the calling application has the *FormStyle* property value fsMDIForm. The form contains three buttons on a panel: One button creates a MDI child window, the second button creates a modal form, and

the third creates a modeless form. Though you can't tell it from the code listings, the calling application's project includes the file MYFORMS.LIB. Doing so ensures that C++Builder loads the DLL when the application starts and resolves the functions referenced in the calling application at link time.

Listing C: DLLAPP.H

```
#ifndef DLLAppUH
#define DLLAppUH
//----------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include <vcl\ExtCtrls.hpp>
//----------------------------------------------

class TForm1 : public TForm
{
  __published:  // IDE-managed  Components
  TPanel *Panel1;
  TButton *Button1;
  TButton *Button2;
  TButton *Button3;
  void __fastcall Button1Click(TObject *Sender);
  void __fastcall FormCloseQuery(TObject *Sender, bool &CanClose);
  void __fastcall Button2Click(TObject *Sender);
  void __fastcall Button3Click(TObject *Sender);
  private:  // User declarations
  public:   // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//----------------------------------------------
extern TForm1 *Form1;
//----------------------------------------------

#endif
```

Listing D: DLLAPP.CPP

```
#include <vcl\vcl.h>
#pragma hdrstop
```

```
#define BUILDAPP
#include "Forms.h"

#include "DLLAppU.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  ShowMDIChildForm(Application);
}
//---------------------------------------------
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool &CanClose)
{
  ResetDllApplication();
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  ShowDllFormModal(this);
}
//---------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  ShowDllForm(this);
}
```

Conclusion

The ability to put forms in a DLL is a powerful feature of C++Builder. As always, use the tools available wisely--don't put forms in DLLs just because you can. But when you must put your forms in a DLL, you can do so with ease, using the techniques we've discussed in this article.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and

Teach Yourself C++Builder in 14 Days. You can contact Kent at **editor@bridgespublishing.com**.

September 1998

MDI child forms in a DLL

by Kent Reisdorph

In the article, "Using a VCL Form in a DLL," we discused how to implement a VCL form contained in a DLL. However, we didn't cover the special case of an MDI child form in a DLL. Let's say your C++Builder application's main form is an MDI form. If you try to use an MDI child form contained in a DLL, you'll get the following exception from VCL: *No MDI forms are currently active*. "What?" you say. "But I *have* an MDI form in my application!" Not according to VCL you don't. Let's see why this happens.

One object or two?

When you try to show your MDI child form, VCL checks to see if the Application object's *MainForm* property is valid. If *MainForm* isn't valid, an exception is thrown. Of course, your *MainForm* property is valid. But the DLL *also* contains an Application object--and VCL checks the DLL's *MainForm*, not that of the application. Since a DLL doesn't have a main form, this check will always fail. You can fix this problem by assigning the DLL's Application object to the Application object of the calling application. Naturally, this will work only if the calling application is a VCL application.

You must also set the DLL's Application object back to its original state before the DLL unloads. Doing so lets the VCL memory manager clean up any memory allocated for the DLL. You'll have to store the DLL's Application object pointer in a variable global to the DLL so it can be restored before the DLL unloads.

Steps to show the form

Let's review the steps required to show an MDI child form in a DLL:

- Create a global TApplication pointer.
- Save the DLL's Application object in the global TApplication pointer.
- Assign the calling application's Application object to the DLL's Application object.
- Create and show the MDI child form.
- Reset the DLL's Application object before the DLL unloads.

The first step is easy. Place the following code at the top of your DLL's source unit:

```
TApplication* DllApp = 0;
```

We set the pointer to 0 so that we can tell whether it's already been assigned. Next, create a function that will perform the TApplication switch and create the child form. The function will look something like this:

```
void ShowMDIChildForm
        (TApplication* mainApp)
{
  if (!DllApp) {
    DllApp = Application;
    Application = mainApp;
  }
  TChildForm* child =
    new TChildForm
        (Application->MainForm);
  child->Show();
}
```

When you call this function, you'll pass the Application object of the calling application. If the DllApp pointer hasn't yet been assigned, you assign the Application object of the DLL to the temporary pointer. You then assign the Application object of the calling application to the DLL's Application object. This check ensures that the Application object is set only once. Next, the code creates the MDI child form and passes the calling application's MainForm as the owner. Finally, the form is displayed. All that remains is to reset the DLL's Application pointer prior to unloading the DLL. You might think you'd perform this action in the *OnClose* event handler for the calling application. However, by the time *OnClose* fires, it's too late--the DLL has already been unloaded. The next best thing is to perform the action in the *OnQueryClose* event handler.

To do so, you'll need a function in the DLL that resets the Application pointer:

```
void ResetDllApplication()
{
  if (DllApp)
    Application = DllApp;
}
```

Remember, you saved the DLL's Application pointer earlier; and now you restore it. Finally, call this function from your application's *OnQueryClose* event handler, and the DLL will unload without incident.

September 1998

Using TStringGrid with graphics

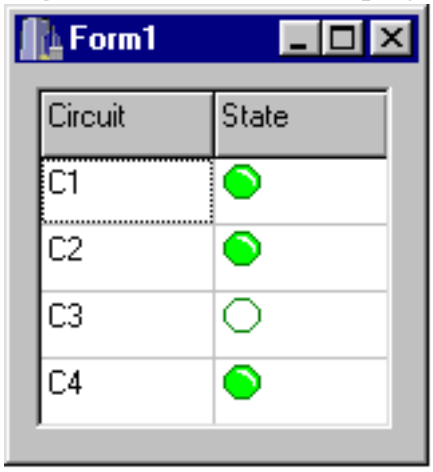
by Gerry Myers

TStringGrid is a good component. However, it lacks a direct property or method to handle one particularly useful capability: graphics. If you've used the TStringGrid component, you know that it works well for text (hence the *string* in *TStringGrid*). But what if you want to throw a bit of graphics into a cell--a custom button or indicator light, for instance? If one column in your grid represents a Boolean (on/off) value, how can you represent its state in a clean, professional way? In this article, we'll show you how to include simple graphics in your grid cells.

Visual indicators

Recently, I worked a project that contained a Boolean column like that I just described. The TStringGrid satisfied all my other display requirements, since the remaining data was text. At first I thought I'd show a 0 or 1 in the Boolean cells--but then I'd have to check every user input to be sure it wasn't something other than *0* or *1*. Instead, I decided to place an indicator-light graphic in those cells and let the user click on them. The graphic toggles between an on and off indication, as shown in Figure A (the indicator is actually green).

Figure A: Our form displays On and Off graphics.

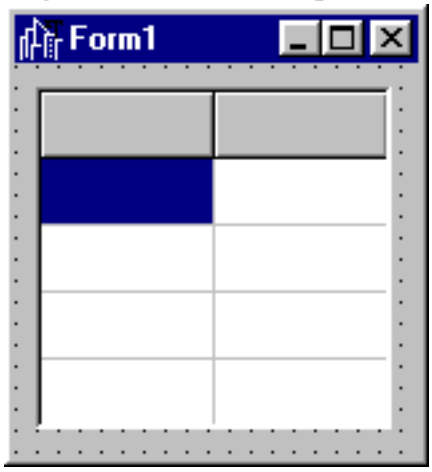


Though I didn't want to display 0 or 1 in the Boolean cell, those values are the best way to represent the state internally. When reading the grid data from a disk file, a 0 in that position meant *off* and a 1 meant *on*. When internally setting information in the cells, I also stored a 0 or 1--but when the grid was displayed, I intercepted the *draw* event for those cells and displayed a graphic rather than the underlying 0 or 1. Let's look at how you can duplicate my results.

OnDrawCell

To begin, open a new C++Builder project. Place a two-column TStringGrid component on the form, as shown in Figure B.

Figure B: Our example starts with this simple, single-component form.



You'll want C++Builder to draw the rest of the grid as usual but let you take over one of the columns. As with most of the VCL components, a built-in mechanism lets you do custom drawing.

Now, enter the code from Listing A in the TStringGrid's *OnDrawCell* event handler.

Listing A: OnDrawCell handler

```
void __fastcall TForm1::
        StringGrid1DrawCell
  (TObject *Sender, int Col, int Row,
  TRect &Rect,
        TGridDrawState State)
{
// Only worry about the State column
// (be sure to skip its header cell).
if ( Col != 1 || Row == 0 ) return;

// Draw or remove the green light to
// show circuit state.
if ( StringGrid1->Cells[ Col ]
  [ Row ].ToInt() != 0 )
  StringGrid1->Canvas->Draw(
    Rect.Left, Rect.Top,
    GreenLightBmp );
```

```
else
  StringGrid1->Canvas->Draw(
    Rect.Left, Rect.Top, BlankBmp );
}
```

This method is called immediately before each cell is drawn. Since you want to handle only one of the columns, the *OnDrawCell* code checks which cell is currently being drawn. If it isn't the Boolean column, you simply return and let C++Builder continue. However, if it's the Boolean column, you load the appropriate bitmap onto the grid's canvas at the cell's location.

Notice in Listing A that the Col and Row parameters you pass indicate the current cell being drawn. You first check whether the cell being drawn is a Boolean cell (column 1 is Boolean). Then, you check to be sure this isn't a header cell (row 0), since you don't want to mess up the State column header.

To determine which bitmap to display (On or Off), you check the actual text value stored in that cell. If it's 1, you load the On bitmap (GreenLightBmp). If the value is 0, you load the Off bitmap (BlankBmp). The graphics are placed on the grid canvas, since there's no such thing as a cell canvas. The location of the current cell being drawn is also passed into this event handler; you use this location to determine where on the grid canvas to place the graphic. You can create the bitmaps using any imaging package, then save them with a BMP extension. Listing B shows the form's constructor, which fills the grid and loads the bitmaps.

Listing B: Form constructor

```
__fastcall TForm1::TForm1(TComponent*
  Owner) : TForm(Owner)
{
// Load the grid cells.
StringGrid1->Rows[ 0 ]->CommaText =
  "Circuit,State";
StringGrid1->Rows[ 1 ]->CommaText =
  "C1,0";
StringGrid1->Rows[ 2 ]->CommaText =
  "C2,0";
StringGrid1->Rows[ 3 ]->CommaText =
  "C3,0";
StringGrid1->Rows[ 4 ]->CommaText =
  "C4,0";

// Load the green (on) light bitmap.
// GreenLightBmp is a class data member.
GreenLightBmp = new Graphics::TBitmap;
```

```
GreenLightBmp->LoadFromFile(
  "GreenLight.bmp" );

// Load the outline (off) light bitmap.
// BlankBmp is a class data member.
BlankBmp = new Graphics::TBitmap;
BlankBmp->LoadFromFile( "Blank.bmp" );
}
```

OnMouseUp

So far, you have a grid that will display different cell graphics depending on the underlying values in the cells. This is fine for programs that change the 0 and 1 internally. However, you'll also want to let users click on the indicator light and toggle the underlying value. You can do so fairly easily by overriding the *OnMouseUp* event handler. As Listing C shows, this time the column and row aren't passed in as parameters. Therefore, you find the position of the mouse, then use the MouseToCell() method to convert this location to the coordinates of the grid cell that the mouse is over.

Listing C: OnMouseUp event handler

```
void __fastcall TForm1::
        StringGrid1MouseUp
    (TObject *Sender,
        TMouseButton Button,
     TShiftState Shift, int X, int Y)
{
  // Get indices of cell
        under the mouse.
  int col, row;
  StringGrid1->MouseToCell( X, Y,
    col, row );

  // Don't worry about
        clicking on header row.
  if ( row == 0 ) return;

  // If left mouse button
        is active, user may
  // user may want to mark or unmark
  // State field.
  if ( Button == mbLeft )
  {
```

```
    // If State column selected,
    // to mark (or unmark) the cell.
    if ( col == 1 )
    {
      // If cell is off, turn on.
      if ( StringGrid1->Cells[ col ]
          [ row ].ToInt() == 0 )
        StringGrid1->Cells[ col ]
          [ row ] = 1;
      else
        StringGrid1->Cells[ col ]
          [ row ] = 0;
     }
  }
}
```

Because you also aren't concerned about the header row (row 0), you check for it and simply return if the user clicked on that cell. The *OnMouseUp* event handler is called for left and right mouse clicks; so, you check to be sure the left-click is being processed. Finally, as before, you make sure this cell is in the Boolean column (column 1).

To toggle the underlying value, simply change the stored value from a 0 to a 1 or vice-versa. You don't toggle the graphic in this code--the *OnDrawCell* event handler we discussed earlier toggles the graphic after queueing off the underlying 0 or 1 set here in the *OnMouseUp* event handler.

OnSelectCell

Again, you could walk away now and have a functioning grid that lets the user click on the graphic to toggle its display and underlying value. There's just one cosmetic problem: Since editing is allowed and highlighting is turned on, a blue highlight background appears in the cell if the bitmap doesn't extend all the way to the cell edges. For example, if the user widens the column so the cell is wider or taller than the bitmap, the highlighted background will show. You could solve this problem by stretching the bitmap instead of simply drawing it onto the cell. However, doing so will probably distort the graphic. Another thought would be to disable editing on that column--but editing can be Watch the use of "only" -- its placement affects meaning.

You changed the phrase to: *"editing can only* be enabled or disabled for the entire grid." This means that all you can do with editing is enable or disable it. However, the phrase as

written, "editing can be enabled or *disabled only* for the entire grid," means that editing is enabled or disabled for the entire grid rather than part of it. enabled or disabled only for the entire grid.

Since you must allow editing on other columns, you must use other means to intercept edits in the Boolean column but allow them everywhere else. Listing D shows the code to add to the *OnSelectCell* event handler, which is called when the user selects a cell using the mouse, arrow, or [Tab] key. By setting the CanSelect parameter, you can effectively turn off editing for individual cells.

Listing D: OnSelectCell handler

```
void __fastcall TForm1::StringGrid1SelectCell
    (TObject *Sender, int Col, int Row,
      bool &CanSelect)
{
  // Don't let user actually edit the
  // underlying value in State column.
  // Only allow clicking on green or
  // blank light.
  if ( Col == 1 ) CanSelect = false;
}
```

Regarding cell highlighting, note that you may want to set the *goRangeSelect Option* property to False. If this option is True (the default), the Boolean cells won't be selected individually--but they can still fall within a selected range, which may make the cell highlighting visible.

Conclusion

The TStringGrid component is very useful. With the ability to create user drawings via the *OnDrawCell* event handler, the component can serve many project needs. In this article, we demonstrated how to display graphics in grid cells. You can easily expand on our discussion to incorporate buttons, checkboxes, combo boxes, and other elements in your grids, expanding their usefulness even further.

September 1998

Debugging with diagnostic macros

by Kent Reisdorph

Debugging is a never-ending process. Although it's often thought of as a process to find and eliminate bugs in your programs, that's only half of the story. Debugging is also a programming tool. This type of debugging might better be thought of as *diagnostics*, which tell you what your program is doing at any given moment. While breakpoints will tell you much of what you need to know, other diagnostic methods are also available. In this article, we'll discuss one such tool: diagnostic macros. We'll show you what they are and how to use them in your programs.

What macros?

The Borland C++ line of products offers a wide variety of diagnostic macros. (Some macros look and act much like functions, but are implemented differently.) Only two of these macros, however, are generally available in C++Builder: TRACE and WARN. These macros allow your program to output messages to a special file, as we'll explain in just a bit. For example, if you want to trace program execution from function to function, you can sprinkle calls to TRACE throughout your code. A call to TRACE looks like this:

```
TRACE("In my OnCreate handler...");
```

The text within quotes is sent to the diagnostic log file. Both TRACE and WARN let you use the iostream streaming operators within the macro. For example, to output the value of a particular variable to the diagnostic log file, you could write code like the following:

```
TRACE("Entering OnFormCreate handler...");
TRACE("ParamCount = " << ParamCount());
TRACE("ParamStr(0) = " << ParamStr(0));
```

In this case, the value of the ParamCount() function is output followed by the value of ParamCount(0). (The ParamCount() function returns the number of command-line parameters, and ParamStr() returns an individual command line parameter. ParamStr(0) always returns the path and filename of the executable.) The previous code snippet would result in the following entry to the diagnostic log file:

```
Trace Unit1.cpp 20: [Def]
  Entering OnFormCreate handler...
```

```
Trace Unit1.cpp 21: [Def] ParamCount = 0

Trace Unit1.cpp 22: [Def]
  ParamStr(0) = E:\Project1.exe
```

We split some of the lines here--the text would all appear on a single line in the log file. We'll talk in just a minute about the format of the log file and how to use it. The WARN macro works like the TRACE, but WARN macro lets you test for a condition as well as send a text string to the log file. If the condition is met, the text is sent to the diagnostic log file; if the condition isn't met, then the text isn't sent. Here's an example:

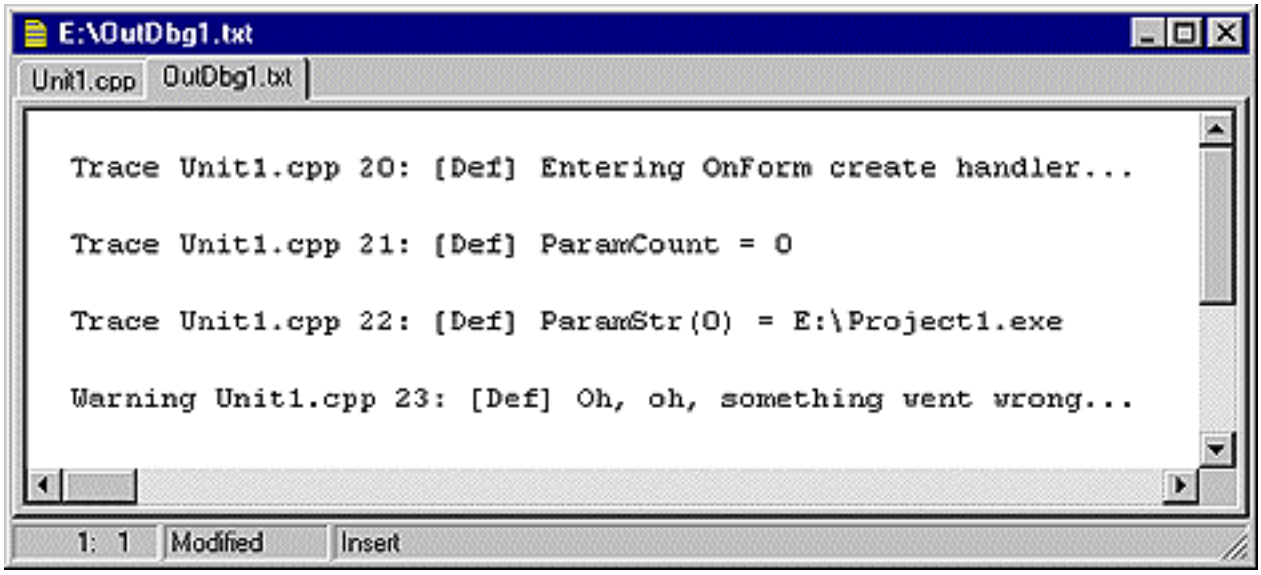
```
WARN(Handle == 0,
        "Something went wrong...");
```

The value of Handle is checked for a non-zero value. If Handle is 0, then the message is sent to the log file; if Handle contains a valid value, then nothing is written to the log file. WARN allows you to output only the messages that you want to see for a given condition. This helps to eliminate clutter in the log file so you can easily find the information you're looking for.

The log file: OutDbg1.txt

You may be thinking, "So the diagnostic macros send text messages to a log file. But what is log file and how do I use it?" The diagnostic macros send output to a special file called OUTDBG1.TXT. It's special for a couple of reasons. First, any time TRACE or WARN messages are generated, OUTDBG1.TXT automatically appears as a new file in the C++Builder Code Editor, as shown in Figure A.

Figure A: When a message is generated, the OUTDBG1.TXT file appears in the Code Editor.



Second, OUTDBG1.TXT isn't saved as part of your project when you use the Save or Save All command. It's treated as a temporary file, and its contents are discarded when you close the file or the project. In addition, you'll never be prompted to save OUTDBG1.TXT. You can explicitly save the log file with File | Save, if you wish.

Note: DLL messages
You may have encountered OUTDBG1.TXT from time to time as you've worked with C++Builder. Once in a while, you'll see a message that warns of a DLL collision. This message is just for informational purposes. The warning simply tells you that the DLL couldn't be loaded at the requested memory location and therefore was loaded in another place in memory. You can happily ignore such messages.

The messages sent to OUTDBG1.TXT are appended to any text already in the file. In other words, if you run your program repeatedly, the contents of OUTDBG1.TXT will grow--old message aren't removed, and new TRACE and WARN messages are added to the end of the file.

Getting ready

In order to get the TRACE and WARN macros to work, you need to perform two steps:

- Include the CHECKS.H header.
- Define __TRACE and __WARN in your application.

Begin by including CHECKS.H in any units that use the TRACE and WARN macros, as follows:

```
#include <checks.h>
```

If you forget to include this file, you'll get a compiler error about TRACE or WARN being undefined symbols. Next, define the __TRACE and __WARN symbols. If you fail to define these symbols, the TRACE and WARN macros won't work. You won't see any indication that the macros aren't working--you simply won't get any output in OUTDBG1.TXT.

You can define these symbols two ways. First, you can define them in one of your source code units or headers:

```
#define __TRACE
#define __WARN
```

Note that each symbol has two underscores preceeding the symbol name. An arguably better way to define the __TRACE and __WARN symbols is to add them to the Conditional Defines field in the Project Options dialog box. To do this, open the Project Options dialog box, then move to the Directories/ Conditionals page. Enter the following text in the Conditional Defines field:

```
__TRACE;__WARN
```

Again, be sure you put a double underscore before each symbol. Notice that a semicolon separates the symbols and that there are no spaces between them. Once you've performed these steps, you can begin to put the TRACE and WARN macros to work. Listing A contains a short program that illustrates how to use these C++Builder macros. The program consists of a form with an edit control and a button.

Listing A: DIAGSU.CPP

```
#define __TRACE
#define __WARN
//---------------------------
        -----------------

#include <vcl\vcl.h>
#include <checks.h>
```

```
#pragma hdrstop

#include "DiagsU.h"
//---------------------------
        -----------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------
        -----------------__fastcall
                TForm1::TForm1
                        (TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------
        -------------------
void __fastcall TForm1::
        FormCreate(TObject *Sender)
{
  TRACE("Entering OnForm
        Create handler...");
  TRACE("ParamCount = "
        << ParamCount());
  TRACE("ParamStr(0) = "
        << ParamStr(0));
}
//---------------------------
        -------------------

void __fastcall TForm1::
                        Button1Click
                        (TObject *Sender)
{
  TRACE("In OnClick handler
         for " <<
    dynamic_cast<TComponent*
        >(Sender)->Name
    << "...");
  static int count;
  count++;
  // Output message if button has been
  // clicked more than 5 times.
  WARN(count > 5, "Button has
```

```
        been clicked "
    << count << " times.")
  TRACE("Edit1 contains: " <<
    Edit1->Text.c_str());
}
//---------------------------
        -----------------void __fastcall
                TForm1::FormClose(TObject
                 *Sender, TCloseAction
                       &Action)
{
  TRACE("Entering OnFormClose handler...");
  TRACE("Application closing...");
}
```

Conclusion

A good programmer will use every available resource when developing and debugging applications. Add the TRACE and WARN macros to your debugging arsenal, and you'll be a more productive programmer.

September 1998

by Jim Bailey

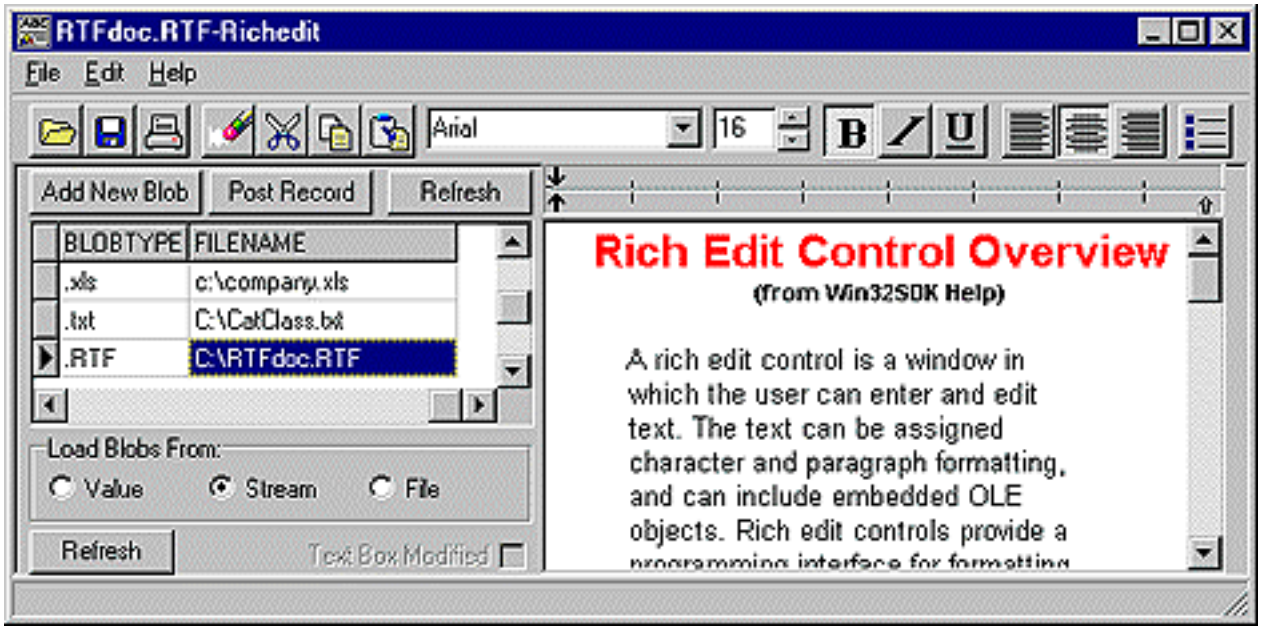
How can I use BLOBs with C++Builder?

Binary Large Objects, or BLOBs, get considerable attention in Borland's newsgroups. Much of this notice is negative, but BLOBs are more reliable than ever in C++Builder 3.0. Essentially, a BLOB can store any data, regardless of its format or size. This is one of the few times you'll experience no bounds or rules in a development environment. Graphical objects, FAX files, and formatted or plain text memos are just a few of the uses for BLOB fields. In this article, we'll discuss some of the BLOB-related differences between C++Builder 1.0 and 3.0. We'll demonstrate how to use BLOBs in an application to create and store formatted memos as well as the contents of a file. And, we'll describe the methods to retrieve, store, and access information in BLOB fields.

About the sample application

You can download our sample application from www.cobb.com/cpb. We developed this application using C++Builder 3.0 and Interbase 4.2; the download ZIP file includes a script file that creates the Interbase GDB file. Many environments lack the ability to create formatted memos in a database. For instance, the standard TDBMemo component doesn't allow formatting--this frustrates users who want to format memo documents. We addressed this need by using Borland's RichEdit example application as the foundation of our sample application. It's a reasonably powerful example that uses TRichEdit to provide a broad range of formatting features. Figure A shows the application's visible components. (Borland included a data-aware TDBRichEdit component in C++Builder 3.0; however, we chose not to use it because the purpose of this article is to demonstrate BLOB methods for accessing and exchanging data.)

Figure A: Our sample application uses these visible components.



BLOB objects

There are three fundamental BLOB objects: TBlobField and its descendents, TmemoField, and TGraphicField. According to the documentation What is this referring to? , the only difference in these field types from C++Builder 1.0 to 3.0 lies in the *DataType* property. Borland cleverly specifies the possible *DataType* settings as ftBlob, ftMemo, and ftGraphic.

Accessing BLOB data

You can access data in a BLOB field three ways: the *Value* property, SaveToStream/LoadFromStream, and SaveToFile/ LoadFromFile. The code in Listings A and B demonstrates these techniques by exchanging data between a BLOB field and a TRichEdit component. Listing A puts data into a BLOB field; this code is called when the memo text is modified to write changes back into the database. On the other hand, the code in >Listing B gets data out of a BLOB field. This code is called each time the current record changes to retrieve BLOB contents into the TRichEdit object.

Listing A: Inserting data into a BLOB field

```
case USE_VALUE:
  qBlobTestBLOB1->Value =
        RichEdit1->Text;
  break;

case USE_STREAM:
  TBlobStream* theBStream;
```

```
  theBStream = new TBlobStream
        (BlobField,
                bmWrite);
  RichEdit1->Lines->SaveToStream
        (theBStream);
  delete theBStream;
  break;

case USE_FILE:
  //Save formatted file.
  RichEdit1->Lines->SaveToFile
        (TEMP_FILE_NAME);

  //Load formatted file into Blob.
  BlobField->LoadFromFile(
        TEMP_FILE_NAME);
  break;
```

Listing B: Getting data from a BLOB field

```
case USE_VALUE:
  RichEdit1->Text =
        qBlobTestBLOB1->Value;
  break;

case USE_STREAM:
  TBlobStream* theBStream;
  theBStream = new TBlobStream
        (BlobField,
    bmRead);
  RichEdit1->Lines->
        LoadFromStream
        (theBStream);
  delete theBStream;
  break;

case USE_FILE:
//Save formatted file into Blob.
  BlobField->SaveToFile
        (TEMP_FILE_NAME);

//Load formatted file.
```

```
  RichEdit1->Lines->
        LoadFromFile(
    TEMP_FILE_NAME);
  break;
```

The *Value* property is always available, of course, but it brings in RTF formatting as text. That defeats the purpose, since our target is formatted memos. To preserve formatting, use the TBlobStream VCL object, which offers several powerful methods for handling BLOB data.

Storing database data in a BLOB field

That brings us to the second objective of the sample application: storing the contents of any database file in a BLOB field. For example, you may need to archive all related files for a particular project. You can include any file type--drawing files, document files, spreadsheets, faxes, and more. The BlobType field, shown in the grid in Figure A, isn't related to the VCL property of the same name. If BlobType (the field) is set to *.txt* or *.rtf*, the BLOB's contents are displayed in the TRichEdit object. Otherwise, the BLOB field's contents are ignored. However, you can transfer the contents to a file by clicking the disk icon or choosing the File | SaveAs menu item.

In the sample application, we saved an Excel spreadsheet file in the database. Naturally, you must be able to retrieve all data from the database in its original format. Doing so is a little tricky because it requires some BLOB field configuration. The configuration is even more important in the sample application, because the same table stores files *and* memos.

Listing C shows some options for setting up a BLOB field for a data transfer. The Transliterate is crucial. If Transliterate is set true, it mangles binary data like that in an Excel, Word, or fax file--the data then can't be retrieved from the BLOB field for use by the originating application. This result is totally unacceptable, of course. However, setting Transliteration to false allows you to transfer a BLOB's contents with absolute accuracy.

Listing C: Configuring a BLOB field to transfer data

```
case RTF_BLOB
  BlobField->Transliterate = true;
  BlobField->SetFieldType(ftMemo);
  RichEdit1->PlainText = false;
  break;

case TXT_BLOB)
```

```
  BlobField->Transliterate = true;
  BlobField->SetFieldType(ftMemo);
  RichEdit1->PlainText = true;
  break;

default:
  BlobField->Transliterate = false;
  BlobField->SetFieldType(ftBlob);
  break;
```

Other functionality

In the sample application, the file functions (Open, Save, and so on) operate on the BLOB field. For example, the Open command opens a file and copies its contents to a BLOB field, while the Save command retrieves data from the BLOB field and copies it to a file. The sample application also presents an excellent paradigm for file archival and the creation of formatted memos. BLOBs provide an excellent tool for storing data when its size and type are unpredictable. The properties and methods provided in the VCL protect developers from the nitpicking details of moving data between files, visual text components, and databases.

I'd like to extend special thanks to Borland TeamB members Ralph Friedman *and* Scott Samet *for their assistance in developing this topic.*

September 1998

TRichEdit and RTF

by Jim Bailey

Many word-processing applications can use Rich Text Format (RTF) files. However, some RTF formatting features supported by Word and other powerful products--such as the page break--aren't supported by the TRichText component. Even so, the TRichEdit's supported RTF formatting commands offer a rich collection of advanced formatting tools. You apply formatting via hidden commands in the text files. For example, an RTF file without any visible text might contain the following default headers and footers:

```
{\rtf1\ansi\deff0\deftab720{\fonttbl{\f0\fswiss
  MS Sans Serif;}{\f1\froman\fcharset2
  Symbol;}{\f2\fswiss\fcharset1 MS Sans
  Serif;}{\f3\fswiss\fcharset1 MS Sans Serif;}}
{\colortbl\red0\green0\blue0;}
\deflang1033\pard\li500\ri-60\
  fi-20\plain\f3\fs16\cf0
\par }
```

To see these hidden RTF format commands, execute the following code:

```
//No visible commands appear.
  ShowMessage(RichEdit1->Text);
 TStringList *L= new TStringList();
  RichEdit1->Lines->SaveToFile
        ("C:\Temp.Txt");
  L->LoadFromFile("C:\Temp.Txt");
  //This command shows RTF commands.
  ShowMessage(L->Text);
  delete L;
```

September 1998

Horizontal scrollbars for list boxes

by Kent Reisdorph

It's not uncommon for a list box to contain text that's too long to fit the box's width. Such a list box needs a horizontal scrollbar so users can see the rest of the text. The vertical scrollbar in a list box appears automatically when the number of items in the list box exceeds the box's height. A horizontal scrollbar, however, requires some work to implement. In this article, we'll demonstrate how to add a horizontal scrollbar to a list box.

Keep on scrolling

To add a horizontal scrollbar to a list box, you'll first determine the width, in pixels, of the longest text item in the list box. Next, you'll send a Windows message to the list box telling it to add the horizontal scrollbar. Let's work through these steps.

How wide is wide?

When you create the horizontal scrollbar, you tell Windows how far to scroll. But before you can do that, you must determine the width of the widest item in the list box. VCL makes this part easy by providing the TCanvas class's TextWidth method, which gives the width, in pixels, of a text string. TextWidth calculates the value using the current font for the canvas. Here's an example:

```
int x = ListBox1->Canvas->
        TextWidth("Hello!");
```

Now it's a simple matter of checking all the items in the list box to see which is the longest. Depending on how you fill your list box, you can check the lengths as you add items or after you've added all the items. To check the items' length after adding them, you can use code like this:

```
int length = 0;
for (int i=0;i<ListBox1->Items->
        ;Count;i++) {
  String text = ListBox1->Items->
        Strings[i];
  int l = ListBox1->Canvas->
```

```
        TextWidth(text);
  if (l > length) length = l;
}
```

If your list box contains a large number of items, this may not be the most efficient method of determining the longest item's length. However, it will suffice for most cases.

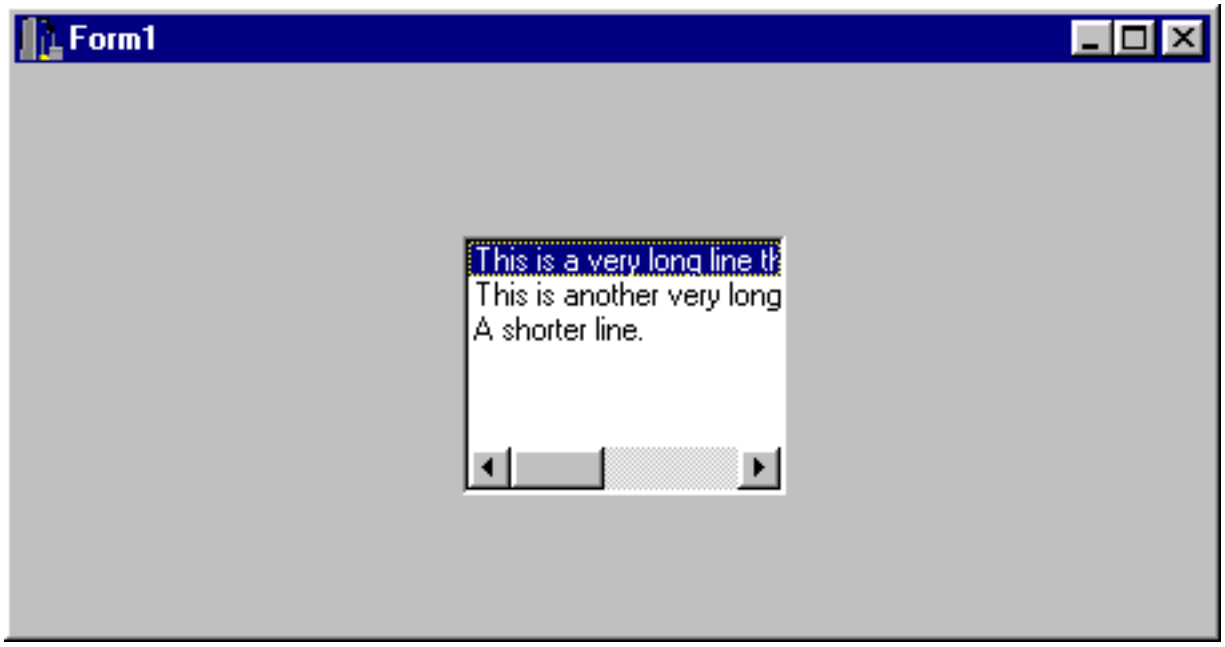
Adding the horizontal scrollbar

Now that you have the length of the longest item, you can tell Windows to add a horizontal scrollbar to the list box. You do so with the LB_SETHORIZONTALEXTENT Windows message, as follows:

```
SendMessage(ListBox1->Handle,
  LB_SETHORIZONTALEXTENT, length, 0);
```

You pass the scrollbar's horizontal extent in the wParam of the LB_SETHORIZONTALEXTENT message. If the list box's width is already wider than the value of wParam, Windows won't add the scrollbar. However, if the list box's width is reduced after you send the message, the scrollbar will appear. To test this theory, start a new project in C++Builder and drop a ListBox component on the form. Double-click on the form's background to generate an event handler for the OnCreate event, then enter the code from Listing A. When you run the program, the list box will display a horizontal scrollbar, as shown in Figure A.

Figure A: Our list box displays a horizontal scrollbar.



Listing A: Adding a Horizontal Scrollbar

```
void __fastcall
TForm1::FormCreate(TObject *Sender)
{
  // Put some items in the list box.
  ListBox1->Items->Add
        ("This is a very long "
    "line that is too wide for
         the list box.");
  ListBox1->Items->Add(
    "This is another very long
        line that is "
    "too wide for the listbox.");
  ListBox1->Items->Add
        ("A shorter line.");

  // Find the length of the longest item.
  int length = 0;
  for (int i=0;i<ListBox1->
        Items->Count;i++) {
    String text = ListBox1->Items-
        >Strings[i];
    int l = ListBox1->Canvas->
        TextWidth(text);
    if (l > length) length = l;
  }

  // Add a little for a good
        visual effect.
  length += 10;

  // Tell Windows to create the scrollbar.
  SendMessage(ListBox1->Handle,
    LB_SETHORIZONTALEXTENT, length, 0);
}
```

August 1998

Low-level wave audio, part 2

by Kent Reisdorph

Last month, in part 1 of this series, we talked about RIFF files and how to use them in low-level audio operations. This month, we'll talk about playing wave audio with the low-level audio functions, including waveOutOpen, waveOutPrepareHeader, waveOutWrite, waveOutUnprepareHeader, and waveOutClose. Finally, we'll show how you can change the volume of the wave output device.

Playing a wave file

Playing a wave file requires several steps. First, you must have data to play. Once you have data, you need to open the wave device, prepare the wave header, and start playback of the wave data. Let's analyze these steps one at a time.

Getting the wave data

Last month we talked about reading wave data from a wave file on disk. That's probably the most obvious way of obtaining wave data. (The OpenBtnClick method in Listing B shows how to read wave data from a file, if you need a refresher course.) A wave file stored as a resource provides another source of wave audio data. The resource can be contained either in a DLL or in your program's executable file. See the article "Playing Wave Resources " for a description of creating and using wave files as resources.

A much less common way of obtaining wave data involves creating the data via some algorithm in your program. For example, you might prefer to create sound effects programmatically rather than loading a wave file from disk. You probably won't use this method of creating wave data very often, though.

Regardless of the source of your wave data, you'll need to allocate a buffer and fill it with wave data before you can move on to the next step. Remember that you must load the wave format header in addition to the data. Again, refer to the OpenBtnClick method in Listing B for an example. Next, you can open the wave audio device in preparation for playback.

Opening the wave-out device

Let's spend a moment talking about wave device identifiers. When you open a wave device, you must specify its device ID. A particular system may have more than one wave output device--for example, a computer may have two sound cards installed. In that case, you'd need to obtain the device ID of the sound card on which you wanted to play your sound. Fortunately, most systems have only one sound card, so you don't have to worry about it much. Windows provides the WAVE_MAPPER constant for use if you're unsure of the sound card's ID. Most of the time, you'll use WAVE_MAPPER and leave it at that. You open a wave output device by calling the waveOutOpen function (all the wave-out functions are located in MMSYSTEM.H). Before you try to open the device, though, it's a good idea to query the device to see whether it supports the format of your wave data. To query a device, call waveOutOpen and specify the WAVE_FORMAT_QUERY flag, as follows:

```
HWAVEOUT WaveHandle;
int Res = waveOutOpen(&WaveHandle, WAVE_MAPPER, &WaveFmt, 0, 0, WAVE_FORMAT_QUERY);
```

This code assumes that you previously filled in the wave format in a WAVEFORMATEX structure called

WaveFmt (usually obtained when you read a wave file). When called in this way, waveOutOpen checks to see whether the device supports the given format, but doesn't actually open the device. If the wave format is supported by the device, waveOutOpen returns 0--you can then move on to opening the device. If the wave format isn't supported, waveOutOpen returns an error code, usually WAVERR_BADFORMAT (decimal 32). The code that opens the device is almost identical to the preceding example:

```
Res = waveOutOpen(&WaveHandle, WAVE_MAPPER, &WaveFmt, MAKELONG(Handle, 0), 0,
CALLBACK_WINDOW);
```

There's a lot going on in this function call, so we'll take some time to explain it. For the first parameter, we pass the address of a variable that will contain a handle to the wave output device. If waveOutOpen is successful (returns 0), you can use the handle returned in WaveHandle when calling subsequent wave output functions. In both the preceding examples, we use the WAVE_MAPPER constant for the second parameter, the device ID. This constant tells Windows to select the sound card as the wave output device. The third parameter is a pointer to the structure that holds the wave format data.

The fourth parameter uses the MAKELONG macro to create a DWORD from the form's window handle. This is the proper way of converting a window handle into a DWORD. (I'll explain why we need the form's window handle when I discuss the flags parameter in just a bit.)

The fifth parameter contains any user-defined data. In this case, we aren't using user-defined data, so we pass 0 for this argument. You might use user-defined data to pass additional data to the wave output callback function, as we'll discuss in the next section.

Finally, we pass the CALLBACK_WINDOW constant for the final parameter. This constant tells Windows that we want any wave-out messages sent to our form's window procedure. Other possibilities for this parameter include CALLBACK_FUNCTION, CALLBACK_EVENT, CALLBACK_THREAD, or CALLBACK_NULL. These flags tell Windows to send the wave-out messages to a callback function, an event, a thread, or not to send messages at all.

Out of these possibilities, the only one you're likely to use is the CALLBACK_FUNCTION flag, which instructs Windows to use a callback function to report wave device status. Unfortunately, you can call only a specific set of Windows functions from within a callback function. Attempting to call any other functions will result in a deadlock. All in all, the CALLBACK_WINDOW flag is the easiest to implement. By the way, this is why we passed the form's window handle in the fourth parameter--because we specified the CALLBACK_WINDOW flag, we had to tell Windows which window procedure to send the messages to. (For more information, see "Wave-Out Messages" in this issue.)

You should always check the return value from waveOutOpen to be sure the device was opened properly and that the wave-out handle is valid. A return value of 0 indicates the function completed successfully. If an error occurs, the return value will be one of the multimedia error codes. To get a textual description of the error message you can call the waveOutGetErrorText function (see the CheckWaveError method in Listing B).

Preparing the wave header

The next step in playing a wave file is to prepare a wave header. The wave header is an instance of the WAVEHDR structure, and includes information about the wave buffer that contains the wave data. Specifically, it holds a pointer to the buffer and the size of the buffer in bytes. After you create and initialize a wave header structure, you call the waveOutPrepareHeader function to assign the wave header to the currently open wave device, as follows:

```
WAVEHDR WaveHeader;
WaveHeader.lpData = WaveData;
WaveHeader.dwBufferLength = DataSize;
Res = waveOutPrepareHeader( WaveHandle, &WaveHeader, sizeof(WAVEHDR));
```

This example assumes you have wave data in a buffer called WaveData, and that the size of the buffer is contained in a variable called DataSize. Notice that we pass the waveOutPrepareHeader function the wave handle obtained when we opened the device. We also pass a pointer to the WAVEHDR structure and, for the final parameter, the size of the structure. At this point, the wave output device is open, the header has been prepared, and we're ready to play the data in the buffer.

Starting playback

As is usually the case, the act of playing the wave data is the easiest part. It's the preparation it takes to get to this point that requires all the work! To start playback, call the waveOutWrite function as follows:

```
Res = waveOutWrite(
  WaveHandle, &WaveHeader, sizeof
        (WAVEHDR));
```

Once again, the WaveHandle variable tells Windows which device we're sending data to. We pass a pointer to the WaveHeader structure, prepared in the preceding step, as the second parameter. The final parameter is the size of the wave header. If waveOutWrite is successful (returns 0), the wave file starts playing and control is immediately returned to your application. The next step is to detect when the wave file has finished playing so you can clean up the wave header and close the wave device.

Clean up and close

After the wave data has completed playback, you need to unprepare the header that you prepared earlier and close the wave output device. Here's an example of the OnWaveDone function that performs those tasks:

```
void TForm1::OnWaveDone
        (TMessage& msg)
{
  if (msg.Msg == MM_WOM_DONE) {
    waveOutUnprepareHeader(WaveHandle, &WaveHeader, sizeof(WAVEHDR));
    waveOutClose(WaveHandle);
  }
}
```

As you can see, there isn't much to it. First we call waveOutUnprepareHeader to unprepare the header. As always, we pass the WaveHandle variable as the first parameter. The second and third parameters are identical to those passed when we called waveOutPrepareHeader earlier. Finally, the waveOutClose function closes the wave device. (If you don't close the wave device, it will be unavailable to other applications or to Windows.) Note that the wave file is always played asynchronously. The waveOutWrite function starts the wave playing and immediately returns control to your application. This means your application is fully operational while the wave file is being played. For this reason, you must use the MM_WOM_DONE message to determine when the file has finished playing. In addition, you'll have to do some more work if you want the

sound to be played synchronously.

Setting the wave-out volume

Naturally, you'll want to be able to set the volume at which your wave data plays--and it's pretty easy to do once you have a valid wave-out device. You can use the waveOutGetVolume and waveOutSetVolume functions to get and set the volume, respectively. The volume is stored in a DWORD. The high word specifies the right channel volume setting, and the low word specifies the left channel volume setting. If the device doesn't have the capability of setting the right and left channel volumes independently, then the low word sets the volume and the high word is ignored. A value of 0 represents no volume, and a value of 0xFFFF specifies full volume. For instance, the following code will set the volume of both channels to 50%:

```
waveOutSetVolume(WaveHandle, 0x80008000);
```

In this case, we're using the handle returned from waveOutOpen to set the volume. You can use a device ID rather than a wave-out handle if you prefer. Since the sound card is typically wave device 0, the following code will set the sound card volume to full volume:

```
waveOutSetVolume(0, 0xFFFFFFFF);
```

This method has the benefit of not requiring an open wave out device in order to set the volume. Setting the volume is as easy as that. Note that waveOutSetVolume sets the volume only for the wave output device, not the master volume. The master volume can be set only through the multimedia mixer control (but that's a discussion best saved for another article).

The big picture

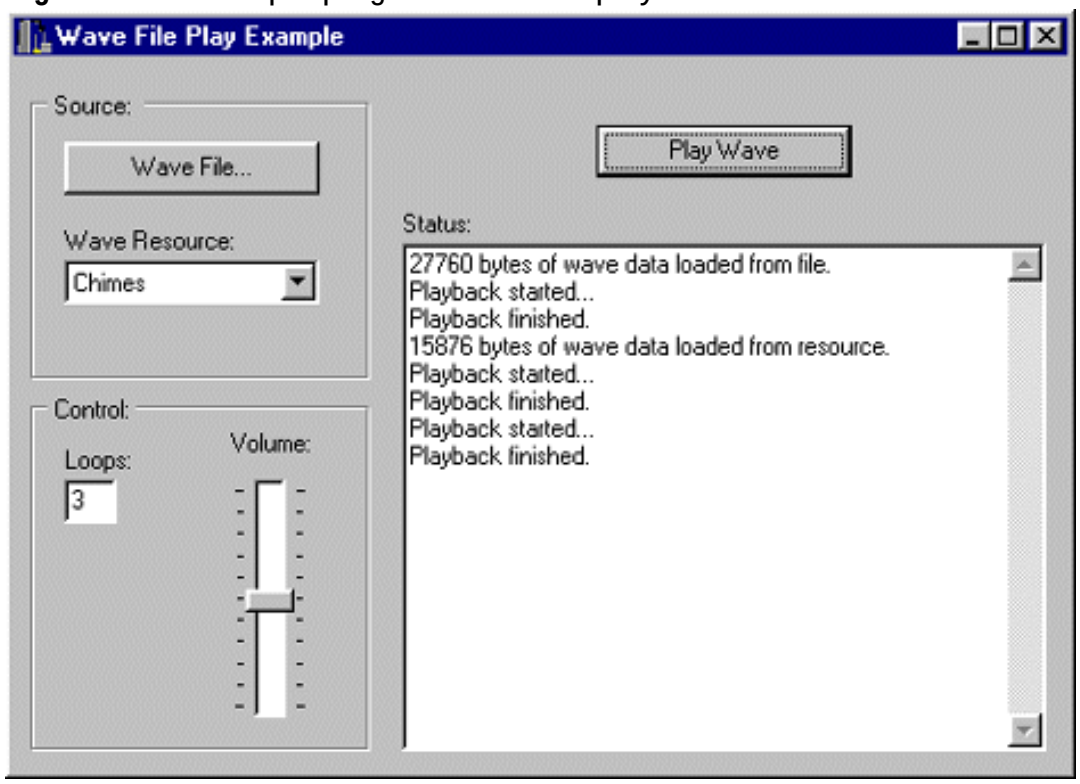
Listings Aand B contain the header file and source for an example program that demonstrates the concepts discussed in this article and in the companion article, "Playing Wave Resources." The example contains the primary components shown in Table A.

Table A: Example program components

Component	Name
TButton	OpenBtn
TButton	PlayBtn
TComboBox	Resources
TMemo	Memo1
TEdit	LoopEdit
TTrackBar	VolumeBar

This program loads wave data either from a file or from one of four resources bound to the program's EXE file. Once the data is loaded, you can click the Play button to play the wave data. The TrackBar component allows you to change the output volume, and the Memo component reports the status of various operations as they occur. An Edit component allows you to specify the number of times to play the wave data. Figure A shows the application after we loaded and played several sound resources.

Figure A: Our sample program loads and plays wave data.



Conclusion

Is it worth all this work just to play a simple sound? It certainly is if you need complete control over your wave audio data. Whenever possible, use the PlaySound function to play wave audio files. Save the use of the low-level wave audio functions for those times when you need the kind of power they provide. Next month, in part 3 of this series, we'll tackle recording wave audio.

Listing A: PlayWavU.H

```
//---------------------------------------------
#ifndef PlayWavUH
#define PlayWavUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Dialogs.hpp>
#include <mmsystem.h>
#include <ComCtrls.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  TButton *PlayBtn;
  TOpenDialog *OpenDialog1;
```

```
  TMemo *Memo1;
  TGroupBox *GroupBox1;
  TLabel *Label1;
  TButton *OpenBtn;
  TComboBox *Resources;
  TGroupBox *GroupBox2;
  TLabel *Label3;
  TTrackBar *VolumeBar;
  TLabel *Label2;
  TEdit *LoopEdit;
  TLabel *Label4;
  void __fastcall OpenBtnClick
        (TObject *Sender);
  void __fastcall PlayBtnClick
        (TObject *Sender);
  void __fastcall FormCreate
        (TObject *Sender);
  void __fastcall FormDestroy
        (TObject *Sender);
  void __fastcall ResourcesChange
        (TObject *Sender);
  void __fastcall VolumeBarChange
        (TObject *Sender);
private:  // User declarations
  char* WaveData;
  WAVEFORMATEX WaveFmt;
  HWAVEOUT WaveHandle;
  int DataSize;
  WAVEHDR WaveHeader;
  void CheckWaveError(DWORD code);
  void CheckMMIOError(DWORD code);
  void OnWaveDone(TMessage& msg);
public:    // User declarations
__fastcall TForm1(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(
      MM_WOM_DONE, TMessage,
        OnWaveDone)
  END_MESSAGE_MAP(TForm)
};
//---------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: PlayWavU.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#include "PlayWavU.h"

//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//---------------------------------------------
__fastcall TForm1::TForm1 (TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  WaveData = 0;
  // Set the volume bar position to the
  // current volume setting.
  DWORD volume;
  waveOutGetVolume(0, &volume);
  VolumeBar->Position = 0xFFFF - LOWORD(volume);
}
//---------------------------------------------

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  // Be sure to delete the buffer.
  if (WaveData)
    delete[] WaveData;
}
//---------------------------------------------

void __fastcall TForm1::OpenBtnClick(TObject *Sender)
{
  // Show the Open Dialog.
  if (!OpenDialog1->Execute()) return;

  // Declare the structures we'll need.
  MMCKINFO ChunkInfo;
  MMCKINFO FormatChunkInfo;
  MMCKINFO DataChunkInfo;

  // Zero out the ChunkInfo structure.
  memset(&ChunkInfo, 0, sizeof
        (MMCKINFO));

  // Open the file.
  HMMIO handle = mmioOpen(
  OpenDialog1->FileName.c_str(), 0,
          MMIO_READ);
  if (!handle) {
```

```
    MessageBox(0,
     "Error opening file.",
        "Error Message", 0);
    return;
  }

  // Find the RIFF chunk.
  DWORD Res = mmioDescend(handle, &ChunkInfo, 0, MMIO_FINDRIFF);
  CheckMMIOError(Res);

  // Descend into the format chunk.
  FormatChunkInfo.ckid = mmioStringToFOURCC("fmt", 0);
  Res = mmioDescend(handle, &FormatChunkInfo, &ChunkInfo, MMIO_FINDCHUNK);
  CheckMMIOError(Res);

  // Read the wave format.
  memset(&WaveFmt, 0, sizeof(WAVEFORMATEX));

  // mmioRead and mmioWrite return the number of
  // bytes read or written so don't call the
  // CheckMMIOError fucntion for those.
  mmioRead(handle, (char*)&WaveFmt, FormatChunkInfo.cksize);

  // Ascend out of the format chunk.
  Res = mmioAscend(handle, &FormatChunkInfo, 0);
  CheckMMIOError(Res);

  // Descend into the data chunk.
  DataChunkInfo.ckid = mmioStringToFOURCC("data", 0);
  Res = mmioDescend(handle, &DataChunkInfo, &ChunkInfo, MMIO_FINDCHUNK);
  CheckMMIOError(Res);

  // Read the data into a buffer.
  DataSize = DataChunkInfo.cksize;
  if (WaveData)
    delete[] WaveData;
  WaveData = new char[DataSize];
  mmioRead(handle, WaveData, DataSize);

  // Close the file
  mmioClose(handle, 0);
  Memo1->Lines->Add(String
        (DataSize) +
    " bytes of wave data loaded
        from file.");
  PlayBtn->Enabled = true;
}
//---------------------------------------------
void TForm1::CheckMMIOError (DWORD code)
{
  // Report an mmio error, if
        one occurred.
```

```
  if (code == 0) return;
  char buff[256];
  wsprintf(buff,
    "MMIO Error. Error Code: %d", code);
  Application->
    MessageBox(buff, "MMIO Error", 0);
}
//---------------------------------------------

void __fastcall TForm1::PlayBtnClick(TObject *Sender)
{
  // Query the device and see if it can play
  // this wave format. If so, open the device.
  int Res = waveOutOpen(&WaveHandle, WAVE_MAPPER, &WaveFmt,  0, 0,
WAVE_FORMAT_QUERY);
  CheckWaveError(Res);
  if (Res) return;
  Res = waveOutOpen(&WaveHandle, WAVE_MAPPER, &WaveFmt,
    MAKELONG(Handle, 0), 0, CALLBACK_WINDOW);
  CheckWaveError(Res);

  // Set up the wave header.
  memset(&WaveHeader, 0, sizeof(WaveHeader));
  WaveHeader.lpData = WaveData;
  WaveHeader.dwBufferLength = DataSize;

  // If the LoopEdit contains a value greater
  // than 1 then set the loop count and flags.
  int loops = LoopEdit->Text.ToIntDef(0);
  if (loops > 1) {
    WaveHeader.dwLoops = loops;
    WaveHeader.dwFlags =
      WHDR_BEGINLOOP | WHDR_ENDLOOP;
  }

  // Prepare the wave header.
  Res = waveOutPrepareHeader(WaveHandle, &WaveHeader, sizeof(WAVEHDR));
  CheckWaveError(Res);

  // Start playback.
  Res = waveOutWrite(WaveHandle, &WaveHeader, sizeof(WAVEHDR));
  CheckWaveError(Res);
  Memo1->Lines->Add("Playback started...");
}
//---------------------------------------------

void TForm1::CheckWaveError (DWORD code)
{
  // Report a wave out error, if one occurred.
  char buff[256];
  if (code == 0) return;
```

```
  waveOutGetErrorText(code, buff, sizeof(buff));
  MessageBox(Handle, buff, "Wave Error", MB_OK);
}
//---------------------------------------

void TForm1::OnWaveDone (TMessage& msg)
{
  // We only care about the WOM_DONE message.
  // When we get this message we know that the
  // sound has finished playing. We can then
  // unprepare the header and close the device.

  if (msg.Msg == WOM_DONE) {
    Memo1->Lines->Add("Playback finished.");
    int Res = waveOutUnprepareHeader(WaveHandle, &WaveHeader, sizeof(WAVEHDR));
    CheckWaveError(Res);
    Res = waveOutClose(WaveHandle);
    CheckWaveError(Res);
  }
}
//---------------------------------------

void __fastcall TForm1::ResourcesChange (TObject *Sender)
{
  // Load a wave resource into a
        TResourceStream
  TResourceStream* res = new TResourceStream((int)HInstance,
    Resources->Items->Strings[Resources->ItemIndex], "Wave");

  // Locate the fmt chunk.
  res->Position = 0;
  int x, i;
  int ckid = mmioStringToFOURCC("fmt ", 0);
  for (i=0;i<100;i++) {
    res->Read(&x, 4);
    if (x == ckid) break;
    res->Position -= 3;
  }

  // Read the size of the format data.
  int size;
  res->Read(&size, 4);

  // Read the wave format.
  memset(&WaveFmt, 0, sizeof(WAVEFORMATEX));
  res->Read(&WaveFmt, size);

  // Locate the data chunk and get the size.
  ckid = mmioStringToFOURCC("data", 0);
  for (int i=0;i<100;i++) {
    res->Read(&x, 4);
```

```
    if (x == ckid) break;
    res->Position -= 3;
  }
  res->Read(&size, 4);

  // Allocate the buffer. Free first if needed.
  if (WaveData)
    delete[] WaveData;
  DataSize = size;
  WaveData = new char[DataSize];

  // Read the wave data into the buffer.
  res->Read(WaveData, DataSize);
  delete res;

  PlayBtn->Enabled = true;
  Memo1->Lines->Add(String(DataSize) +
   " bytes of wave data loaded from
        resource.");
}
//---------------------------------------------

void __fastcall TForm1::VolumeBarChange (TObject *Sender)
{
  // Set the volume to the current
  // track bar position.
  int value = 0xFFFF - VolumeBar->Position;
  waveOutSetVolume(0, MAKELONG(value, value));
}
//---------------------------------------------
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1998

Wave-out messages

by Kent Reisdorph

Windows defines three messages for wave output devices, as described in Table A.

Table A: Windows wave-out messages

Message	Description
MM_WOM_OPEN	The device has been opened.
MM_WOM_CLOSE	The device has been closed.
MM_WOM_DONE	The device has finished playing

Of these messages, you'll usually care only about MM_WOM_DONE, which tells you when the buffer has finished playing. You need to know when the buffer is finished so you can perform cleanup chores. The MM_WOM_DONE message is sent either when the sound finishes normally or when the sound is interrupted. (To stop playback of the sound once it's started, call the waveOutReset function.)

As we mention in the accompanying article, you can use a callback function rather than having Windows send messages to your window procedure. According to the Multimedia Programmer's Reference (MMEDIA.HLP), the notification messages sent to the callback function have slightly different names: WOM_OPEN, WOM_CLOSE, and WOM_DONE, respectively. It doesn't matter which set of constants you use, though, because they contain exactly the same values.

Catching wave-out messages

In order to catch the MM_WOM_DONE message, you need to implement a message map. You can do so with C++Builder's MESSAGE_MAP macro. We've discussed handling custom messages in past journals, but you can probably benefit from a refresher course. The first step in implementing a handler for a particular message is to declare a generic message-handling function. Next, add a message map to your class declaration. The following example shows the pertinent parts of a main form's class declaration, which implements a message map entry for the MM_WOM_DONE message:

```
class TForm1 : public TForm
{
// things omitted
private:
  void OnWaveDone(TMessage& msg);

public:
  __fastcall TForm1(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(
      MM_WOM_DONE, TMessage, OnWaveDone)
  END_MESSAGE_MAP(TForm)
};
```

Finally, you need to define the OnWaveDone function in your source unit. The empty OnWaveDone function is as follows:

```
void TForm1::OnWaveDone(TMessage& msg)
{
}
```

August 1998

Playing wave resources

by Kent Reisdorph

Most of the time, you play wave audio from a file on disk. Sometimes, however, you want to be able to play a sound contained as a resource. The resource might be in a DLL, or it might be in your EXE. This article will show you how to bind a wave audio resource to your EXE and how to play that wave resource at runtime.

Creating the resource

In order to bind a wave audio resource to your EXE, you must create a resource script (RC) file. A resource script file is nothing more than a text file with an extension of RC, containing text in a format that can be read by C++Builder's resource compiler. By far the easiest way of getting wave audio data into an executable is to start with a wave file on disk. Then, you can create a wave resource with a single line in the RC file:

```
MY_WAVE_RESOURCE WAVE "chimes.wav"
```

That's it! The format for creating a custom resource is as follows:

```
RESNAME RESTYPE FILENAME
```

To create a resource script file, create a new text file from the C++Builder Object Repository. Enter your resource text in the blank file, then save the file with an RC extension. (Don't forget to remove the default **.TXT* extension in the File Save dialog box or you'll end up with a filename that looks like MYRES.TXT.RC.) Here's a sample RC file that defines four wave resources:

```
// RESNAME        RESTYPE       FILENAME
// -------------------------------------
   CHIMES          Wave         "chimes.wav"
   CHORD           Wave         "chord.wav"
   DING            Wave         "ding.wav"
   TADA            Wave         "tada.wav"
```

(We put comments in the file so you could see the format of the resources, but they aren't necessary.) You'll typically see resource names in all uppercase, but that isn't a requirement. In fact, it doesn't matter whether you use uppercase or lowercase, because resource names aren't case sensitive.

Binding the resources to the EXE

Binding the resources to the EXE sounds like a big job. In reality, doing so requires nothing more than adding the RC file you just created to your application's project. Simply use C++Builder's Add To Project option and add the resource script to your project. The next time you compile your application, the resource script file will be compiled (by the resource compiler) and bound to your EXE (or DLL, if you have a DLL project). Be sure your resource file doesn't contain errors. If the resource compiler encounters errors, you'll get a message like the following:

```
[RCError] Wave.RC(4): Cannot open file: Wave.
[RCFatal Error] Compile.
```

The format for specifying wave resources in RC files is pretty basic, so you shouldn't have problems with resource compiler errors. Naturally, you'll want to check that any wave files referenced in your resource script are in your project directory, or that you fully qualify the path to the wave file.

Tip: Quick resource compile
If you have C++Builder 3.0, you can right-click on the RC file in the Project Manager to compile the resource file.

Putting the wave resource to work

Now that you have a wave resource bound to your EXE, you can put it to work. There are two ways to play a wave file contained as a resource. Let's take a look.

The easy way

It's easiest to play a sound resource by using the Windows API function PlaySound. (We discussed PlaySound last month in the article "Low-level Wave Audio, Part 1.") Here's how the code looks:

```
PlaySound("Chimes", HInstance, SND_RESOURCE);
```

You didn't know it would be so easy, did you? Note that the resource name is passed as the first parameter to PlaySound. The second parameter is the instance handle to the module that contains the wave resource. In this case, we use HInstance to tell Windows to look in the EXE for the wave resource. If your wave resources are contained in a DLL, you can load the DLL using LoadLibrary and use the returned instance handle in your call to PlaySound. For example:

```
HINSTANCE dllInstance;
dllInstance = LoadLibrary("myres.dll");
PlaySound("raygun", dllInstance, SND_RESOURCE);
```

The final parameter of PlaySound can contain many flags, but you must specify the SND_RESOURCE flag in addition to any other flags you provide. This flag tells Windows that the sound you're playing is a resource (as opposed to a file on disk or an alias for a system sound).

The hard way

The hard way to play a sound resource involves using the low-level wave audio functions as discussed in the article, "Low-level Wave Audio, Part 2." You won't typically have to go to this extent to play wave resources, but sometimes it's necessary. For example, TAPI (the telephony API) allows you to play a wave file through a voice modem. The mechanism specified by TAPI for playing wave files uses the low-level wave audio functions (waveOutOpen and others). Hence, if you want to use wave resources with TAPI, you must go to the trouble of implementing the low-level wave audio functions.

The TResourceStream class is great for loading a resource into memory. Once it's loaded, you can manipulate the resource as needed. Here's the code to load the CHIMES resource used in the previous example:

```
TResourceStream* res = new TResourceStream(
  (int)HInstance, "CHIMES", "Wave");
```

The TResourceStream constructor takes three parameters. The first parameter is the instance handle of the module that contains the resource. In this case, the instance handle must be cast to an int because VCL expects the instance handle to be an integer. The second parameter is the name of the resource to load, and the final parameter is the resource type as defined in the RC file. Now that you have the wave resource loaded into memory, you can copy the important bits of data from the resource stream. A wave file in memory is, of course, in a pre-defined format. The format is identical to a wave file stored on disk. You can go the hard way and use mmioOpen to open the resource as a memory-mapped file, or you can just cheat a little and locate the wave format header and wave data using TResourceStream. For example, here's how to read the wave format header of a wave resource:

```
// Locate the fmt chunk.
int x, i;
int ckid = mmioStringToFOURCC("fmt ", 0);
for (i=0;i<100;i++) {
  res->Read(&x, 4);
  if (x == ckid) break;
  res->Position -= 3;
}
// Read the size of the format data.
int size;
res->Read(&size, 4);
// Read the wave format.
memset(&WaveFmt, 0, sizeof(WAVEFORMATEX));
res->Read(&WaveFmt, size);
```

You can do the same thing to read the actual wave data. For instance, the ResourceChange method in Listing A of the accompanying article, "Low-level Wave Audio, Part 2," shows how you can load a wave resource into a buffer in preparation for playback. Once you've loaded the wave format header and loaded the wave data into a buffer, you can play the wave data using the low-level audio functions as described in the accompanying article.

Conclusion

Playing a wave resource isn't difficult once you know how to go about it. Whenever possible, you should opt for the easy way and use the PlaySound function to play the wave resource. When nothing else will do, though, the low-level wave audio functions provide an alternate means of playing a wave resource.

August 1998

Using OWL TDialogs in a VCL application

by Kent Reisdorph

Let's assume for a minute that you have an OWL application you're moving to C++Builder. As part of the conversion process, you'll have to consider converting your dialog boxes to VCL forms. You could argue that VCL forms are easier to use than traditional dialog boxes. But if you're converting a large OWL application to C++Builder, it will be tedious to convert all of the standard dialog boxes that OWL uses to VCL forms. What if you have dozens or even hundreds of dialog boxes? Wouldn't it be nice to know you could use them, and the OWL TDialog class, in your VCL applications? The good news is that you can, as long as you have Borland C++ 5.02. This article will show you how.

A world of troubles

If you don't know the magic combination, you might conclude that it's just not possible to use a TDialog in a VCL application. Your first attempt will probably be rife with compiler errors. If you finally get everything to compile, you'll then be faced with linker errors that seem insurmountable. The main problem is one of namespace pollution. *Namespace pollution* is a programming buzz phrase that refers to two variables using the same name at the same time but in different contexts. Think about this for a minute. What's the class name for the standard VCL edit component? And what's the class name for the OWL class that represents edit controls? The answer in both cases is TEdit. If both OWL and VCL have a TEdit class, which one is the compiler supposed to use? And this is just one example--both VCL and OWL have classes called TMenu, TBitmap, TListBox, TComboBox, and so on.

In theory, Borland has "fixed" this problem by allowing you to declare the BI_NAMESPACE symbol (BI_NAMESPACE was first implemented in version 5.02 of Borland C++). When this symbol is defined, the OWL classes are wrapped in the namespace called OWL. This means that you should be able to do something like the following:

```
OWL::TEdit* owlEdit;      // an OWL TEdit
Stdctrls::TEdit* vclEdit; // a VCL TEdit
```

The problem is that in some cases, defining BI_NAMESPACE results in compiler errors in the OWL headers, so you never get this far. Borland's plan was a good one, but somewhere along the way it wasn't completely implemented. So, we have to go back to the drawing board. But rather than dwell on the negative, let's look ahead to what's required to be able to use a TDialog in a VCL application.

Step by step

To use a TDialog in a C++Builder VCL application, you'll need to follow these steps:

1. Add an RC file containing a dialog box resource to your C++Builder project.
2. Add the OWL library and include paths to your project's paths.
3. Create a new source unit that contains the code to call the OWL dialog box.
4. Write code in your VCL unit to execute the code in the previous step.
5. Add the proper OWL library files to your C++Builder project.

Let's take a look at these steps individually. Along the way, we'll explain why they're necessary.

Add the resource script file to your project

The first step is to have a dialog box created as a resource and contained in a resource script (RC) file. We'll assume that you already have a resource script file or know how to create one. (You can use the Borland C++ 5 IDE to create your dialog box resources, or even the standalone Resource Workshop from earlier versions of Borland C++.) Once you've created the RC file, add it to your C++Builder project using the Add To File option. At build time, C++Builder will see the resource script file in your project and will compile the resources and bind them to the executable file during the link phase.

You'll probably have an RH file as well as the RC file. The RH file is a resource header that contains identifiers for the dialog box resource and for the controls contained on the dialog box. A resource header for a simple dialog box with only two edit controls would look like this:

```
#define IDD_DIALOG          101
#define IDC_FIRSTNAMEEDIT   102
#define IDC_LASTNAMEEDIT    103
```

You should include the resource header in any units that reference the resources contained in the RC file. (More on this in just a bit.)

Add the OWL library and include paths to your project

This step is fairly simple. Just open the Project Options dialog box, then enter the path to

your OWL INCLUDE directory in the Include Path field and enter the path to your OWL LIB directory in the Library Path field. Be sure to separate the directories with a semicolon. The resulting Include Path field might look like this:

```
$(BCB)\include;$(BCB)\include\vcl;c:\bc5\
èinclude
```

Create a new unit

Next you need to create a unit that will contain the OWL code. By using a new unit, you'll avoid the namespace issue mentioned earlier--any OWL and VCL classes that have the same name never appear in the same unit. (While there are other ways to avoid the namespace issue, this is by far the easiest.) This new unit (let's call it OWLUNIT.CPP) might contain just one function, which will be called from the main form. The entire unit might look something like this:

```
#include <owl\dialog.h>
#pragma hdrstop

#include "OWLUnit.h"
#include "OWLDlg.rh"

void CallOWLDialog(HWND hWndParent)
{
  TWindow vclWindow(hWndParent);
  TDialog* dlg =
    new TDialog(&vclWindow, IDD_DIALOG);
  dlg->Execute();
  delete dlg;
}
```

Let's take a moment to analyze this code. First, notice that we include the DIALOG.H header file, which is the header for the OWL TDialog class. Notice also that we don't include VCL.H. Since a new unit always adds a line that includes VCL.H, you'll need to delete that line for any units that contain OWL code exclusively. Failure to remove that line will result in the very namespace problems we're trying to avoid.

Next, we include the files OWLUNIT.H and OWLDLG.RH. The header for this unit would probably contain only the function prototype of any public functions in the unit. For example:

```
#ifndef OWLUnitH
#define OWLUnitH

extern "C"
void CallOWLDialog(HWND hWndParent);

#endif
```

Note that we use the extern "C" syntax as we would for a function called from a DLL. Doing so is necessary to avoid linker errors later on. The code within the function is as follows:

```
TWindow vclWindow(hWndParent);
TDialog* dlg =
  new TDialog(&vclWindow, IDD_DIALOG);
dlg->Execute();
delete dlg;
```

The first line creates an OWL TWindow object from the window handle of the VCL application's main form. This process is known as *aliasing* a non-OWL window--it's a great OWL feature. We do this because we need a TWindow pointer to call the dialog box, and aliasing the VCL window serves this purpose. Once we have the TWindow alias, we can construct the TDialog object just as we would in an OWL application. In an OWL program, you normally pass this for the parent of the dialog box. Here, we pass the address of the TWindow object we just created. We also pass the resource ID of the dialog box (defined in the RC file when the dialog box was created).

Finally, we call the Execute() method of TDialog to show the modal dialog box. After the dialog box returns, we free the memory associated with it by calling delete. All in all, it's a pretty simple operation.

Write code to execute the dialog box from the VCL application

Normally, you'll display a dialog box as the result of a menu item selection or a button click. Assuming you're showing the OWL dialog box as the result of a button click, the code is as follows:

```
void __fastcall
```

```
TForm1::Button1Click(TObject *Sender)
{
  CallOWLDialog(Handle);
}
```

That's all there is to it. We need the window handle (HWND) of the form to create an OWL TWindow alias for the VCL form, so we pass the Handle property to the CallOWLDialog() function. At this point, the application will compile. It won't link, however, because the linker doesn't know where to find the definition of the TDialog class and its functions.

Add the OWL library files to the project

For the final step, you need to add the appropriate OWL library files to your project. These library files contain the OWL code needed to execute the OWL dialog box. If you forget to add the OWL library files to your application, you'll get linker errors. There are two types of library files: *import* and *static*. Most OWL programmers know that you can create an OWL application in one of two ways: with dynamic linking or with static linking. (In VCL applications, there's only one choice: static linking.) If you use dynamic linking, the executable file contains no OWL code itself. Instead, the code is called from the OWL and runtime library DLLs when needed by the application. These DLLs must be shipped with any application that uses dynamic linking. Dynamic linking requires the use of import library files.

When you static link, however, no DLLs are required. Instead, the OWL code needed to run your application is extracted from the OWL and runtime library (RTL) static libraries and is linked into your application at link time.

In theory, you should be able to choose either one of these options for the OWL code in your VCL applications. In reality, there are some problems associated with dynamic linking of the OWL libraries. I won't go into the details, but suffice it to say that the safest route is to choose static linking. While this isn't the final word on the subject, you should note that static linking is the easier and more reliable method at this time.

So how do you choose either dynamic or static linking? By the specific libraries you link into your C++Builder application. For static linking, you need to add the following library files to your project:

```
OWLWF.LIB
BIDSF.LIB
```

If you were to attempt dynamic linking, then you'd need to add the OWLWFI.LIB and BIDSFI.LIB files. The easiest way to add the library files to your project is with the C++Builder's Add To Project feature. Since you've already modified your project's library path to point to your OWL library path, you can just add the filenames without path information. For example, when you choose Add To Project, a file selection dialog box opens; simply enter *OWLWF.LIB* in the edit box and then click the Open button. The alternative is to navigate to the \BC5\OWL\LIB directory and find the appropriate library file.

When you use Add To Project to add a library file, C++Builder will add the following lines to your project's source file:

```
USELIB("owlwf.lib");
USELIB("bidsf.lib");
```

If you prefer, you can manually add these lines to the project source file.

And away we go...

After following the steps we've outlined, you'll have a VCL application that can display an OWL TDialog. ListingsA, B, C, D, E, and F contain a program that displays an OWL dialog box from within a VCL application. The sample dialog box lets you enter a first name and last name. If you click the OK button, then the data entered in the dialog box is displayed in two edit components on the VCL form. Our example goes the extra mile by using OWL's transfer buffer mechanism to transfer data to and from the OWL dialog box. When you look at Listing F (the RC file), you should be aware that we've broken the lines to fit our columns. The lines are broken with the continuation character (a backslash) and will compile as listed although they look a bit odd.

Conclusion

You probably wouldn't set out to write a C++Builder application that uses OWL dialogs. But if you have a legacy OWL app that you want to slowly move to C++Builder, then the ability to display OWL dialogs might be very important to you. Now you know that calling an OWL dialog from a VCL application, although unusual, is possible.

Listing A: OWLDLGU.H

```
//---------------------------------------------
```

```
        #ifndef OWLDlgUH
#define OWLDlgUH
//---------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------

class TForm1 : public TForm
{
__published:  // IDE-managed Components
  TButton *Button1;
  TLabel *Label3;
  TLabel *Label4;
  TEdit *FirstNameEdit;
  TEdit *LastNameEdit;
  void __fastcall Button1Click(TObject *Sender);
private:        // User declarations
public:    // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: OWLDLGU.CPP

```
//---------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "OWLDlgU.h"
#include "OWLUnit.h"

//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
```

```
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject
  *Sender)
{
  // Create transfer buffer structure, fill it
  // with the contents of the edit components,
  // and pass it to the CallOWLDialog function.
  TransferBuffer buffer;
  strcpy(buffer.FirstName,
    FirstNameEdit->Text.c_str());
  strcpy(buffer.LastName,
    LastNameEdit->Text.c_str());
  // Call the OWL dialog.
  CallOWLDialog(Handle, buffer);
  // Transfer the results from the transfer
  // buffer structure to the edit components.
  FirstNameEdit->Text = buffer.FirstName;
  LastNameEdit->Text  = buffer.LastName;
}
```

Listing C: OWLUNIT.H

```
#ifndef OWLUnitH
#define OWLUnitH

struct TransferBuffer {
  char FirstName[20];
  char LastName[20];
};

extern "C"
void CallOWLDialog(
  HWND hWndParent, TransferBuffer& tb);

#endif
```

Listing D: OWLUNIT.CPP

```
#include <owl\dialog.h>
#include <owl\edit.h>
#pragma hdrstop
```

```
#include "OWLUnit.h"
#include "OWLDlg.rh"

void CallOWLDialog(HWND hWndParent, TransferBuffer& tb)
{
  // Create a TWindow alias for the VCL form.
  TWindow vclWindow(hWndParent);
  // Create the TDialog object.
  TDialog* dlg =
    new TDialog(&vclWindow, IDD_DIALOG);
  // Create the edit controls required for the
  // transfer buffer.
  new TEdit(dlg, IDC_FIRSTNAMEEDIT, 20);
  new TEdit(dlg, IDC_LASTNAMEEDIT, 20);
  // Set the transfer buffer.
  dlg->SetTransferBuffer(&tb);
  // Show the dialog.
  dlg->Execute();
  delete dlg;
}
```

Listing E: OWLDLG.RH

```
#define IDD_DIALOG         101
#define IDC_FIRSTNAMEEDIT  102
#define IDC_LASTNAMEEDIT   103
```

Listing F: OWLDLG.RC

```
#include "OWLDlg.rh"

IDD_DIALOG DIALOG 0, 0, 240, 120
STYLE DS_MODALFRAME | DS_3DLOOK | \
  DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE | \
  WS_CAPTION | WS_SYSMENU
CAPTION "OWL Dialog in a VCL App"
FONT 8, "MS Sans Serif"
{
 CONTROL "", IDC_FIRSTNAMEEDIT, "edit", \
  ES_LEFT | WS_CHILD | WS_VISIBLE | \
  WS_BORDER | WS_TABSTOP, 60, 44, 100, 12
 CONTROL "", IDC_LASTNAMEEDIT, "edit", \
  ES_LEFT | WS_CHILD | WS_VISIBLE | \
```

```
  WS_BORDER | WS_TABSTOP, 60, 72, 100, 12
 CONTROL "OK", IDOK, "BUTTON", BS_PUSHBUTTON |\
  BS_CENTER | WS_CHILD | WS_VISIBLE |\
  WS_TABSTOP, 186, 6, 50, 14
 CONTROL "Cancel", IDCANCEL, "BUTTON", \
  BS_PUSHBUTTON | BS_CENTER | WS_CHILD |\
  WS_VISIBLE | WS_TABSTOP, 186, 26, 50, 14
 CONTROL "This is an OWL Dialog.", 101, \
  "static", SS_LEFT | WS_CHILD | WS_VISIBLE, \
  44, 20, 88, 8
 CONTROL "Frame1", 102, "static",SS_ETCHEDFRAME\
    | WS_CHILD | WS_VISIBLE, 8, 9, 160, 99
 CONTROL "First Name:", 104, "static", \
  SS_LEFT | WS_CHILD | WS_VISIBLE, 20, 47, \
  40, 8
 CONTROL "Last Name:", 105, "static", \
  SS_LEFT | WS_CHILD | WS_VISIBLE, 20, 73, \
  40, 8
}
```

July 1998

Message-handling for non-visual components

by Kent Reisdorph

A non-visual component must occasionally respond to either Windows operating system or user-defined messages. However, a non-visual component has no window and, therefore, no window handle. And without a window handle, it can't receive messages. In this article, we'll explain how to create a hidden window so your non-visual components can receive messages.

What you'll need...

In order to create a hidden window for your component, you'll need the following:

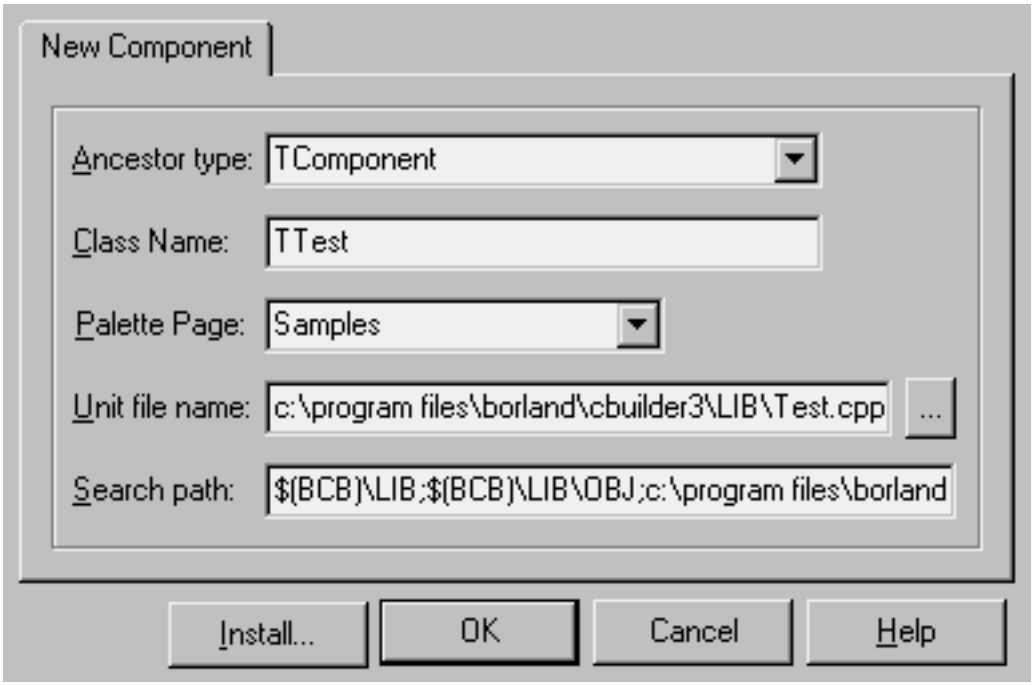
- A private variable of type HWnd to hold the window handle.
- A function to catch the messages that Windows will send to the component (a WndProc).
- A call to AllocateHWnd to create the window handle and set up the WndProc.

That's really all there is to it. Since you'll want to see this process in action, let's build a simple component to illustrate this concept.

Creating the component

First, we'll create a new component. Choose File | New... from the C++Builder main menu and double-click the Component icon in order to display the New Component dialog. Change the Ancestor type to *TComponent* and the Class Name to *TTest*, as shown in Figure A.

Figure A: Select the ancestor and name of your new class.



The Palette Page field is already set to Samples, so we'll leave that field alone. Click OK to create the component.

Now we have a new unit containing the basic source code for our component. (The code generated for the component will vary slightly depending on whether you're using C++Builder 1 or C++Builder 3. The differences aren't important for this discussion, so we won't go over them here.) Go ahead and save your work. Choose File | Save All to open the Save As dialog box. Save Unit1.cpp and Unit1.h as *TestBCB.cpp* and *TestBCB.h*, respectively.

Switch to the component's header file. Place the following declarations in the private section of the class declaration:

```
HWnd FHandle;
void __fastcall WndProc (TMessage& Msg);
```

The first line declares an HWnd variable called FHandle. This variable will hold the window handle after the window is created. The second line declares the WndProc function that will receive the messages. The function declaration must have the signature shown here in order to qualify as a WndProc. Next, move down to the public section of the class declaration and add this line below the constructor declaration:

```
void DoIt();
```

This is a public function that we'll use to test the component. Your class declaration should now look like this (if you're using C++Builder 1, the declaration won't have the PACKAGE modifier):

```
class PACKAGE TTest : public
        TComponent
{
private:
    HWnd FHandle;
    void __fastcall WndProc
        (TMessage& Msg);
protected:
public:
    __fastcall TTest
        (TComponent* Owner);
    void DoIt();
__published:
};
```

Now switch to the component's source unit. Enter the following line near the top of the unit (just above the ValidCtrCheck function is probably a good place):

```
#define MY_MESSAGE WM_USER + 1
```

This line declares a user-defined message that the component will send to itself when the DoIt function is called. At this point, we must allocate a window handle for the component. This handle will provide a hidden window that we can use to catch messages in the component. Locate the component's constructor, and add a line so that the constructor looks like this:

```
__fastcall Test::Test
        (TComponent* Owner)
        : TComponent(Owner)
{
  FHandle = AllocateHWnd
        (WndProc);
}
```

That's all there is to that particular step. The AllocateHWnd function creates a hidden window and returns the window handle of the hidden window. Notice we pass the address of the WndProc function so that Windows knows where to send any messages. Speaking of WndProc, let's create that function next. Add this code to your source file:

```
void __fastcall TTest::WndProc
        (TMessage& Msg)
```

```
{
  if (Msg.Msg == MY_MESSAGE)
    MessageBox(0, "Got here!",
         "Message", 0);
  try {
    Dispatch(&Msg);
  }
  catch (...) {
    Application->
        HandleException(this);
  }
}
```

Windows calls this function whenever Windows sends a message to the component. The code in this function does two things. First, it checks to see whether the message received is our user-defined message, MY_MESSAGE. If it is, a message box is displayed so you can see that the message was, in fact, received.

In addition, this code passes the message on for processing by the system (or by VCL). The Dispatch function performs this service. The try/catch block is used to ensure that, in the event an exception is thrown, it will be handled in the default manner. In a nutshell, the WndProc function watches for our custom message, while passing all other messages on for default handling.

Now we only have to create the DoIt function, and our component will be finished. Add this code to your source unit:

```
void TTest::DoIt()
{
  PostMessage(FHandle,
        MY_MESSAGE, 0, 0);
}
```

This function simply posts a message to the component's window handle (remember, the window handle was previously saved in the FHandle data member). We've finished creating the component. Select File | Close All to save your work. You'll be prompted to save any outstanding changes.

Testing the component

Naturally, our next step is to test the component. If you're using C++Builder 1, just install the component to the component palette using Component | Install. However, if you're using

C++Builder 3, then you must add the component to a package, again using Component | Install. (You can use the DCLSTD35 package for quick tests like this.) In either case, select the TestBCB.cpp file you just saved. Once you've installed the component, it will show up on the component palette. It has a default icon, of course, but that won't stop you from testing it. Drop the TTest component and a regular button component on a form. Double-click the button and enter this code in the OnClick event handler for the button:

```
Test1->DoIt();
```

Now, run the program. When you click the button, you should see a message box that says, "Got here!". Sure enough, the component works as advertised. Listings A and B contain the header and the source code for the TTest component. The code provided is for the C++Builder 3 version of the component. You can download both versions from our Web site at www.cobb.com/cpb; click on the Source Code hyperlink.

Conclusion

A non-visual component that can respond to Windows messages has many uses. The most obvious are for components, which encapsulate some aspect of the Windows API. For example, both TAPI and Winsock send messages to inform the user of events. If you write a component that encapsulates one of these APIs, you'll need a way to catch the messages that Windows sends. Adding a hidden window to your component allows you to do just that.

Listing A: TESTBCB.H

```
//--------------------------
        --------------------
#ifndef TestBCBH
#define TestBCBH
//--------------------------
        --------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//--------------------------
        --------------------
class PACKAGE TTest : public TComponent
{
  private:
    HWnd FHandle;
    void __fastcall WndProc(TMessage& Msg);
```

```
  protected:
  public:
    __fastcall TTest (TComponent* Owner);
    void DoIt();
  __published:
};
//---------------------------------------------
#endif
```

Listing B: TESTBCB.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "TestBCB.h"
#pragma package(smart_init)

// Declare a user-defined message.
#define MY_MESSAGE WM_USER + 1

//---------------------------------------------

// ValidCtrCheck is used to assure that
// the components created do not have
// any pure virtual functions.

static inline void ValidCtrCheck(TTest *)
{
  new TTest(NULL);
}
//---------------------------------------------

__fastcall TTest::TTest
        (TComponent* Owner)
    : TComponent(Owner)
{
  // Allocate a hidden window for the component.
  // We store the return value in FHandle, and
  // pass the WndProc function as a parameter.
  FHandle = AllocateHWnd(WndProc);
}
//---------------------------------------------
```

```
void __fastcall TTest::WndProc(TMessage& Msg)
{
  // Respond to the MY_MESSAGE message, and do
  // default handling for all other messages.
  if (Msg.Msg == MY_MESSAGE)
    MessageBox(0, "Got here!", "Message", 0);
  try {
    Dispatch(&Msg);
  }
  catch (...) {
    Application->HandleException(this);
  }
}

void TTest::DoIt()
{
  // This public function sends a user-defined
  // message to the component's hidden window.
  // This will result in the WndProc function
  // being called.

  PostMessage(FHandle, MY_MESSAGE, 0, 0);
}

// The usual registration stuff
namespace Testbcb
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] =
      {__classid(TTest)};
      RegisterComponents
      ("Samples", classes, 0);}
}
//---------------------------------------------
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1998

Low-level wave audio, Part I

by Kent Reisdorph

The TMediaPlayer component that comes with C++Builder is great for playing wave files, MIDI files, and AVI files. Sometimes, however, you need to take a lower-level approach to audio programming. For example, you might need to manipulate wave data in order to change volume, add special effects, and so on. In this series of articles, we'll focus on low-level wave audio at the system level. We'll begin this month by discussing Microsoft's Media Control Interface (MCI), then we'll cover RIFF files. In part 2, we'll go into detail about waveform output. Part 3 will cover recording waveform audio.

What is MCI?

MCI refers to Microsoft's Media Control Interface--not the telephone giant. As the heart of multimedia in Windows, MCI handles wave input and output, MIDI input and output, AVI video playback and capture, and more. In fact, MCI even has the capability to control videodisc players and VCRs. Basically, MCI can control any device that can install an MCI driver. For example, you can use MCI to play Autodesk Animator animations--provided that the Autodesk drivers are installed properly. MCI provides a relatively high-level interface between you and the system's hardware. This high-level interface takes a huge burden off the programmer because she doesn't have to write device drivers for every known sound card, video card, or other hardware. In fact, you don't really care about the hardware the user has installed because MCI takes care of all of that. All you have to know is the basic MCI functions and how to use them.

The MCI command set includes commands for audio playing, recording, saving, and positioning; also, video capture, video output control (display window size and position), playback in forward, reverse, fast, or slow, and so on. Utility commands allow you to query a device to identify commands that device supports, discern the driver manufacturer and version, and determine whether the device is capable of certain operations (such as stereo output and volume control). All in all, MCI has a very rich command set.

All of the MCI is contained in MMSYSTEM.DLL. To use the MCI functions, you simply include MMSYSTEM.H in your source units and start calling the functions you need. Basically, the 32-bit MCI API has four levels; we've listed them here in order of increasing complexity:

- The PlaySound function
- The string interface

- The command interface
- Low-level routines

The PlaySound function is a simple means of playing a waveform audio file from within a 32-bit Windows program. We won't discuss the PlaySound function here, but you can look it up in the Win32 API help file for more information. The string interface provides a very high-level approach to MCI. Using the string interface you can write code such as

```
char* cmd = "play test.wav
        from 0 to 3000 wait";
mciSendString(cmd, 0, 0, 0);
```

This example plays the first three seconds of a wave file called TEST.WAV. As you might expect, the string interface doesn't give you as much control over MCI as you might need in all circumstances. The MCI command interface is much more complex than the string interface, but it also gives you more control. The heart of the MCI command interface is the mciSendCommand function. (Although we won't go into detail on the MCI command interface here because of its complexity, you can check out an article by Kent featuring mciSendCommand at http://www.borland.com/borlandcpp/news/cobb/bcj3_1a.html.)

Finally, the low-level MCI routines provide you with the most control over multimedia operations. As you might expect, that kind of power comes with a price--you have to do a lot of the work that the higher-level interfaces do for you. However, when you need that kind of control, then the MCI low-level routines are just the ticket.

Why MCI?

You might be wondering, "Why use MCI? I thought DirectX was the way to handle multimedia in a Windows program." Don't believe everything you hear! While DirectX has some great features, it also has several drawbacks. First, it's an ActiveX library, which means you have to distribute DirectX with your application and make sure it's properly installed. In addition, your application is at the mercy of other application developers who also use DirectX. If an unknowing or uncaring developer installs an older version of DirectX on a user's machine, your application may stop working properly.

If you're a component developer and you need sound in one of your components, then using MCI is the only way to go. A VCL component that relies on an ActiveX control wouldn't be very well received by the public (components that encapsulate ActiveX controls excluded, of course). Also, since MCI is installed as part of Windows, you don't have to worry about your users installing anything but your component.

RIFF files

Wave audio is stored in RIFF files. RIFF is an acronym for *resource interchange file format*. Working with RIFF files is the less glamorous part of dealing with low-level audio. Still, it's a subject that we must cover. A RIFF file is organized into sections called *chunks*. The RIFF architecture allows for a hierarchical method of storing data in a file with chunks containing *subchunks,* as well as data. These subchunks may contain data, as well as their own subchunks. In the case of a wave file, there isn't a lot of data to store, so the file format is fairly straightforward.

A wave file contains both a format chunk that holds the wave format header and a data chunk that contains the actual waveform data. The hierarchy looks like this:

Root chunk

- Wave Format Chunk

- Data - Data Chunk - Data

You navigate a RIFF file by descending and ascending through the chunk layers. Now that you've had a brief introduction to RIFF files, let's look at how to read a wave file.

Reading a wave file

While reading a wave file isn't a trivial pursuit, neither is it very difficult. Although I can't say that once you've read a wave file you'll never forget how, I can guarantee you that this task is definitely something you can master. Reading a wave file requires the structure and functions in Table A.

Table A: Wave file structure and functions

Item	Description
MMCKINFO	Forms the chunk information structure.
mmioOpen	Opens the file.
mmioDescend	Descends into a chunk.

mmioAscend	Ascends out of a chunk.
mmioRead	Reads data from a chunk.
mmioClose	Closes the file.

We'll separate this operation into sections to make it easier to understand. However, we won't go into intimate detail on each of these functions; we simply don't have time. Instead, you'll learn by the best teacher: example.

Step 1: Open the wave file

The first step, naturally, is to open the file. The mmioOpen command looks like this:

```
HMMIO mmioOpen(LPSTR szFilename,
        LPMMIOINFO lpmmioinfo,
DWORD dwOpenFlags);
```

where szFilename is the address of a string containing the filename of the file to open. The lpmmioinfo parameter is the address of an MMIOINFO structure containing extra parameters used by mmioOpen. The final parameter, dwOpenFlags, are flags for the open operation. Using mmioOpen is pretty easy as we show here:

```
HMMIO handle =
  mmioOpen("test.wav", 0,
          MMIO_READ);
if (!handle) {
  MessageBox(Handle,
    "Error opening file.",
         "Error Message", 0);
  return;
}
```

This code opens the file and assigns the resulting file handle to the handle variable. The handle is checked for validity, and an error message displays if the file wasn't opened successfully.

Step 2: Read the RIFF chunk

The next step is to read the file's root chunk. The root chunk in a RIFF file is called the RIFF chunk. We need the RIFF chunk to access the format and data chunks. You'll read the RIFF chunk by calling mmioDescend,with the following format:

```
MMRESULT mmioDescend
        (HMMIO hmmio, LPMMCKINFO lpck,
LPMMCKINFO lpckParent,
UINT wFlags);
```

In this code, hmmio is the file handle of an open RIFF file and lpck is the address an application-defined MMCKINFO structure. Next, lpckParent is the address of an optional application-defined MMCKINFO structure. Finally, wFlags specifies search parameters. Our example specifies the MMIO_FINDRIFF flag. The following code shows how to read the RIFF chunk:

```
MMCKINFO ChunkInfo;
memset(&ChunkInfo,
         0, sizeof(MMCKINFO));
Res = mmioDescend(handle,
  &ChunkInfo, 0,
        MMIO_FINDRIFF);
if (Res)
  MessageBox(0, "Error",
        "Error", 0);
```

First, we declare an instance of the MMCKINFO structure, ChunkInfo, and zero it out. Next, we call the mmioDescend function, passing a pointer to the chunk structure. Once again, we check the return value from mmioDescend to be sure the function succeeded. If the call to mmioDescend succeeds, the ChunkInfo variable contains the chunk information for the file's RIFF chunk. (Note**:** Because of space considerations, in subsequent examples we won't include the code that checks the return value of each function. You should check the return values of each function in your own code.)

Step 3: Read the wave format header chunk

Now that we have the RIFF chunk, we can use it to extract the format and data chunks. Here's the code for extracting the wave format header chunk:

```
MMCKINFO FormatChunkInfo;
FormatChunkInfo.ckid =
  mmioStringToFOURCC
        ("fmt", 0);
mmioDescend(handle,
  &FormatChunkInfo,
         &ChunkInfo,
                MMIO_FINDCHUNK);
WAVEFORMATEX waveFmt;
mmioRead(handle,
  (char*)&waveFmt,
        FormatChunkInfo.cksize);
```

The first line of this code snippet declares another instance of the MMCKINFO structure. (We need a second structure to hold the sub-chunk information.) The second line uses the mmioStringToFOURCC macro to convert four characters into a FOURCC value and assign that value to the ckid member of the MMCKINFO structure. The mmioStringToFOURCC macro is defined as

```
FOURCC mmioStringToFOURCC
        (LPCSTR sz, UINT wFlags);
```

where sz is the address of the null-terminated string we want to convert to a four-character code and wFlags specifies conversion options. MCI uses FOURCC values to identify chunks. A FOURCC value is simply a DWORD created out of four characters. The four characters that identify the wave format header are fmt and a space. Keep in mind that we have to specify only three characters, because mmioStringToFOURCC will use blank spaces to pad the string out to four characters.

Once the ckid member has been set, we call mmioDescend to descend from the RIFF chunk into the fmt chunk. After the call to mmioDescend completes, the FormatChunkInfo structure will be filled with the chunk's information, including the size of the chunk's data. Next, we create an instance of the WAVEFORMATEX structure. This structure will hold the wave format header.

Finally, we use the mmioRead function to read the wave header into the waveFmt structure. The function is declared as

```
LONG mmioRead(HMMIO hmmio,
        HPSTR pch, LONG cch);
```

where hmmio is the handle of the file to be read, pch is the address of a buffer to contain the data read from the file, and cch is the number of bytes to read from the file. We must cast the address of the waveFmt structure to a char*, since that's the type mmioRead requires for this parameter. Notice that we pass the size of the chunk as the size parameter (the cksize member of MMCKINFO contains the size of the chunk's data). This ensures that we read only as many bytes as the chunk actually contains. At this point, the wave header structure, waveFmt, contains the wave format information about the wave file (sample rate, bits per sample, mono or stereo, and so on).

Step 4: Read the data chunk

Now we get to the good stuff--we're ready to read the wave data. To read the wave data, we need to ascend out of the format chunk (where we are now) and descend into the data chunk. We'll first create another MMCKINFO structure to hold the data-chunk's information, and set its ckid data member to the ID of the data chunk. Here's how it looks:

```
MMCKINFO DataChunkInfo;
mmioAscend(handle, &
        FormatChunkInfo, 0);
DataChunkInfo.ckid =
  mmioStringToFOURCC
        ("data", 0);
mmioDescend(handle,
  &DataChunkInfo,
        &ChunkInfo,
                MMIO_FINDCHUNK);
```

This code is almost identical to the previous code where we descended into the format chunk. Notice that the string value of the FOURCC for the data chunk is data. Here, we're introducing the mmioAscend function. Its definition is

```
MMRESULT mmioAscend(HMMIO
```

```
        hmmio, LPMMCKINFO
                lpck, UINT
                 wFlags);
```

where hmmio is the file handle of an open RIFF file. The lpck parameter is the address of an application-defined MMCKINFO structure previously filled by the mmioDescend or mmioCreateChunk function. The wFlags parameter is reserved and must be zero. Now that we have the chunk information for the data chunk, we can actually read the data. Remember that the cksize member of the MMCKINFO structure contains the size of the data in the chunk. We'll use this size to allocate a buffer for the data and to read the data. Here's the code:

```
unsigned int size =
        DataChunkInfo.cksize;
char* data1 = new
        char[size];
mmioRead(handle,
        data1, size);
mmioClose(handle, 0);
```

That's all there is to it. The data1 character array now holds all of the wave file's data. (We'll do something with that data next.) After we've read the data, we close the file with the mmioClose function, which simply accepts the handle of the file to close and flags for the close operation.

Writing a wave file

Now we'll take a quick look at how to write a wave file. You already know most of what you need to know to do so. Most of the code is just a variation on the code we used previously to read a wave file. An obvious exception is the use of mmioWrite to write the data to the file where we used mmioRead when we read the file. The mmioWrite function accepts three parameters, as shown here:

```
LONG mmioWrite(HMMIO hmmio,
         char _huge* pch, LONG cch);
```

You use the first parameter, hmmio, to specify the handle of the file. The latter two, pch and cch, are the address of the buffer to be written to the file and the number of bytes to write to the file, respectively.

We're going to write the data we just read to a new file. Just to add spice, we'll reverse the data so that the wave file plays backwards. Because the wave format won't change, and we're writing the exact number of bytes that we just read, we'll use the same FormatChunkInfo and DataChunkInfo structures that we used when we read the file. Since they contain all the necessary data, we'll just reuse them to write the file.

You must follow these steps to write the wave file:

1. Reverse the wave data.
2. Create and open the new file.
3. Create the RIFF chunk
   .
4. Create the fmt chunk.
5. Write the fmt chunk data.
6. Ascend out of the fmt chunk.
7. Create the data chunk.
8. Write the data chunk.
9. Close the file.

We'll give you the code all in one go. Here's how it looks:

```
// Create the new data buffer.
char* data2 = new char[size];

// Copy the original
        data into the new
// buffer in reverse order.
for (unsigned int i=0;i<
        size;i++) {
  data2[size - i] = data1[i];
}

// Open a new file.
handle = mmioOpen(
  "test.wav", 0, MMIO_CREATE
        | MMIO_WRITE);

// Write the RIFF chunk.
mmioCreateChunk(
```

```
  handle, &ChunkInfo,
         MMIO_CREATERIFF);

// Create and write the
         format chunk.
mmioCreateChunk(handle,
        &FormatChunkInfo, 0);
mmioWrite(handle,
  (char*)&waveFmt, sizeof
        (WAVEFORMATEX) - 2);

// Ascend out of the format
        chunk.
mmioAscend(handle, &
        FormatChunkInfo, 0);

// Create and write the data
        chunk.
mmioCreateChunk(handle, &
        DataChunkInfo, 0);
mmioWrite(handle, data2, Data
        ChunkInfo.cksize);

// Close the file.
mmioClose(handle, 0);
```

This code is fairly straightforward, especially once you understand the structure of a wave file. However, you should notice one thing: The line that writes the wave format header looks like this:

```
mmioWrite(handle,
  (char*)&waveFmt, sizeof
        (WAVEFORMATEX) - 2);
```

Note how we subtract 2 from the size of a WAVEFORMATEX structure when we write the structure. If we don't do this subtraction, then the Sound Recorder program that comes with Windows 95 won't be able to play the wave file we created (although, it works fine with the Windows NT Sound Recorder). The reason is that the Win95 Sound Recorder expects the wave format header to be a PCMWAVEFORMAT structure rather than a WAVEFORMATEX structure. The latter is two bytes longer than the former, so we just cheat a little and subtract two bytes when we write the structure to the file. The extra two bytes we're cutting off are for

ADPCM file formats and aren't used with PCM wave files. We took a shortcut here, but any code you write should do the right thing based on the wave-format type.

When you play the TEST.WAV produced by this code, you'll hear a wave file that plays backwards (a la Pink Floyd or the Electric Light Orchestra). You can find the complete example program for this code on our Web site at www.cobb.com/cpb; click on the Source Code hyperlink.

The example program, RIFFTEST, takes a wave file, reverses its data, and saves it to a new wave file. The contents of each chunk are displayed in a memo control so you can view them. The program also allows you to play both the original file and the converted file.

Conclusion

RIFF files aren't very exciting. However, if you're going to do low-level wave audio work, then you need to know about RIFF files. Next month, we'll continue this series and discuss how to play wave files directly from a data buffer.

July 1998

Automating table activation

by Mark G. Wiseman

Whenever you're working with classes derived from TDataSet, such as TQuery and TTable, you often need to open and close their underlying database tables. To do so, you can use the Open and Close methods of the TDataSet class. Calling Open and Close has the same effect as setting the *Active* property of TDataSet to True and False, respectively. You may wonder why you'd repeatedly open and close a table. One reason is to save system resources--keep an infrequently-used table closed and open it only when you need it. Or, you may want to modify the table in some way that requires you to first close the table.

Manually activating a table

For an example, consider how you'd switch between a local and network database. You can easily accomplish this switch by changing the value of the *DatabaseName* property of a TTable component. However, if you attempt to change *DatabaseName* before closing the table, an exception will occur. To prevent this problem, you'll need to determine whether the table is open or closed before you close it. You must also remember to restore the previous state of the table when you're finished. The code in Listing A shows how to switch databases.

Listing A: Changing the database for a TTable

```
ChangeDatabase(TTable *myTable,
        String dbName)
   {
   bool holdActive = myTable->
        Active;
   myTable->Active = false;
   myTable->DatabaseName = dbName;
   myTable->Active = holdActive;
   }
```

A better solution: the TDSActive Class

Setting the open or closed state of a table, then restoring the previous state is a common and

error-prone activity. However, TDSActive, a very simple class shown in Listing B, makes your code easier to write. It sets and restores the table state, thereby reducing errors.

Listing B: The TDSActive class

```
#ifndef TDSActiveH
#define TDSActiveH

#include <db.hpp>


class TDSActive
   {
   public:
      TDSActive(TDataSet *dataset,
        bool active = true);
      ~TDSActive();

   private:
      TDataSet *dataset;
      bool holdActive;
   };

inline TDSActive::TDSActive(TDataSet
        *dataset, bool active) :
                dataset(dataset)
   {
   holdActive = dataset->Active;
   dataset->Active = active;
   }

inline TDSActive::~TDSActive()
   {
   dataset->Active = holdActive;
   }

#endif
```

The TDSActive constructor accepts two arguments. The first, dataset, is a pointer to a TDataSet or a class derived from TDataSet. The second, active, is a bool variable that determines whether the TDataSet will be made active or inactive. The latter argument defaults to True. To open a TTable object pointed to by myTable, you'd write the code

```
TDSActive(myTable)
```

To close the TTable object, you'd write

```
TDSActive(myTable, false);
```

The TDSActive constructor remembers the open or closed state of the table by storing the table's *Active* property in the bool variable, holdActive. Finally, when the TDSActive object goes out of scope, the destructor restores the table's *Active* property using holdActive.

Listing C shows the code from Listing A rewritten to take advantage of TDSActive. The code no longer cares about the current state of the object pointed to by myTable. In addition, you don't need to worry about restoring that state after changing databases.

Listing C: Changing the database for a TTable using TDSActive

```
ChangeDatabase(TTable *myTable,
          String dbName)
   {
   TDSActive(myTable, false);
   myTable->DatabaseName = dbName;
   }
```

To use TDSActive, you need to declare it in the header file, as shown in Listing B. There is no source file. TDSActive takes advantage of the C++ rules concerning the creation and destruction of objects on the stack. You benefit because it makes working with database tables a little simpler.

July 1998

Sorting a TList

by Kent Reisdorph

The TList class is great for storing objects of any type. You can store structures, classes, integers, doubles, and well, just about any type of data. Frequently, you may need a sorted list; TList has a Sort method that allows you to sort the list. Unfortunately, the Sort method is a bit confusing at first and often misunderstood. In this article, we'll show you how to use the Sort method to sort your lists, no matter what they contain.

The Sort method

You use the Sort method of TList to sort the list. That's not too surprising, is it? The declaration for Sort looks like this:

```
void __fastcall Sort(TListSort
        Compare Compare);
```

The single parameter for the Sort method is a pointer to a comparison function, which will be called for every item in the list. We'll talk more about the comparison function in a minute.

Once you've created a comparison function, you can call Sort with the following lines:

```
TList* list = new TList;
// Add some items to the list.
list->Sort(Comparison
        Function);
```

Believe it or not, that's all there is to calling the Sort method. Obviously, there must be more to calling Sort than these few lines of code. There is, of course, so let's examine the comparison function. Then we'll return to the Sort method.

The comparison function

The comparison function does all of the work of sorting the list. It accepts two items from the list and returns a value. The comparison function must be of type TListSortCompare, which is declared as follows:

```
typedef int __fastcall
        (*TListSortCompare)
  (void * Item1, void * Item2);
```

That declaration might not mean much to you, so let me clarify a bit. The comparison function follows a specific function signature. Here's a sample declaration for a comparison function:

```
int __fastcall Sorter(void* Item1,
        void* Item2);
```

As you can see, the function has two parameters. The two parameters are pointers to two of the objects in the list. (It doesn't matter to you which two items in the list are being passed in, just how you compare the two.)

Notice that both parameters are void pointers. Since a TList just stores a list of pointers, that's exactly what you get in the comparison function. Put another way, TList doesn't know what it contains, so it just gives you two pointers and leaves it up to you to decide what to do with them. You'll cast the void pointers to something useful, compare the two items, then return a value based on the comparison.

The comparison function must be either a regular function or, if the function is a member of a class, it must be declared as static. The easiest approach is to just make it a regular function.

Sorting simple data

The value returned from the comparison function determines how the list is sorted. Table A shows what you should return from the comparison function.

Table A: Return value from the comparison function.

Condition	Return Value
Item1 = Item2	0

Item1 > Item2	Any positive number
Item1 < Item2	Any negative number

We'll look at a simple example to illustrate the point. Let's say you had a list of integers that you wanted to sort. In that case, the comparison function would look like this:

```
int __fastcall SortInt
        void* Item1, void* Item2)
{
  int X1 = (int)Item1;
  int X2 = (int)Item2;
  return X1 - X2;
}
```

Because the list contains integers, we must cast the void pointers to an int to get the actual value of the item. (As I've said in past articles, I'm a big fan of the C++ casting operators over the old C-style casts. However, in this case, you don't gain anything by using the C++ casting operator, static_cast, so the C-style cast is perfectly acceptable.)

Now look at the return statement in this function. If X1 is greater than X2, then a positive value will be returned. If X1 is less than X2, then the return statement produces a negative value. When X1 and X2 are equal, the return value is 0: A single statement takes care of it all.

This one statement sorts the list of integers in ascending order. But what if you want to sort the integers in descending order? Not a problem; just reverse the variables in the return statement:

```
return X2 - X1;
```

The list is now sorted in descending order. To sort the list in either ascending or descending order, you'll have to use two separate comparison functions (or implement a global variable that holds the sort direction).

Sorting complex objects

You can sort complex objects as well as simple objects. For example, you may have a list of classes that you want to sort. In that case, the sort is only slightly more complex than it would be if you were sorting integral data types. So what do you use as a sort criteria when sorting classes? That's up to you. Since you know what your class contains, you can sort it in any way you like.

Let me give you an example of a class that contains an integer data member. First, we'll look at this class:

```
class MyNumberClass {
  public:
    MyNumberClass(int data)
        { Data = data; }
    int Data;
  // More of the class here.
};
```

Now, let's say you had a list of MyNumberClass objects. To sort the list in ascending order, you'd write a comparison function like this:

```
int __fastcall IntSort
        (void* Item1, void* Item2)
{
  MyNumberClass* MC1 =
        (MyNumberClass*)Item1;
  MyNumberClass* MC2 =
        (MyNumberClass*)Item2;
  return MC1->Data - MC2->Data;
}
```

Notice that this function differs only slightly from the integer sorting function you saw earlier. First, the void pointers are cast to MyNumberClass pointers, then the Data member of the two objects is used to perform the comparison. Similarly, if you had a class that contained an AnsiString data member, you could sort the list like this:

```
int __fastcall StrSort
        (void* Item1, void* Item2)
```

```
{
  MyStringClass* MS1 =
        (MyStringClass*)Item1;
  MyStringClass* MS2 =
        (MyStringClass*)Item2;
  return MS1->Text.AnsiCompare
        (MS2->Text);
}
```

Although these are pretty simple cases, my point is that you can sort any type of object --in any way you wish. Deciding how the objects should be sorted is completely up to you. All you have to do is return 0, a positive number, or a negative number from the comparison routine.

Listing A contains the main form unit of a program that demonstrates sorting TList container classes. The program sorts two types of classes: one by an integer value and the other according to a string data member. (We are showing only the CPP file. The header is all C++Builder-generated, so there's no point in showing it here.)

The program's main form contains two buttons. The first button sorts the list of integer-based classes, and the second button sorts a list of string-based classes. Both ascending and descending sorting routines are provided. The results of the sort are displayed in a memo, so you can see the result.

To create this program, place a Memo component and two buttons on a form. Then enter the code below. Alternatively, you can download the code from our Web site at www.cobb.com/cpb ; click on the Source Code hyperlink.

Listing A: LISTSORT.CPP

```
//-------------------------
\       --------------------
#include <vcl.h>
#pragma hdrstop

#include "SortU.h"
//-------------------------
        --------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-------------------------
        --------------------
```

```
__fastcall TForm1::TForm1
        (TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------
        --------------------

// A class that contains
        an int data member.
class MyNumberClass {
  public:
    MyNumberClass(int data)
        { Data = data; }
    int Data;
};

// A class that contains a
        String data member.
class MyStringClass {
  public:
    MyStringClass(String text)
        { Text = text; }
    String Text;
};

// The ascending sort function
        for MyNumberClass.
int __fastcall
SortAscending(void* Item1, void* Item2)
{
  MyNumberClass* MC1 =
        (MyNumberClass*)Item1;
  MyNumberClass* MC2 =
        (MyNumberClass*)Item2;
  return MC1->Data - MC2->Data;
}

// The descending sort function
        for MyNumberClass.
int __fastcall
SortDescending(void* Item1, void* Item2)
{
  MyNumberClass* MC1 =
```

```
        (MyNumberClass*)Item1;
  MyNumberClass* MC2 =
        (MyNumberClass*)Item2;
  return MC2->Data - MC1->Data;
}

// The ascending string sort function
// for MyStringClass.
int __fastcall
StringSortAscending
        (void* Item1, void* Item2)
{
  MyStringClass* MS1 =
        (MyStringClass*)Item1;
  MyStringClass* MS2 =
        (MyStringClass*)Item2;
  return MS1->Text.AnsiCompare
        (MS2->Text);
}

// The descending string sort function
// for MyStringClass.
int __fastcall
StringSortDescending
        (void* Item1, void* Item2)
{
  MyStringClass* MS1 =
        (MyStringClass*)Item1;
  MyStringClass* MS2 =
        (MyStringClass*)Item2;
  return MS2->Text.AnsiCompare
        (MS1->Text);
}

void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  // Create a TList and fill it
        with instances
  // of the MyNumberClass class.
  TList* list = new TList;
  list->Add(new MyNumberClass(100));
  list->Add(new MyNumberClass(1));
  list->Add(new MyNumberClass(23));
```

```
list->Add(new MyNumberClass(54));
list->Add(new MyNumberClass(77));
list->Add(new MyNumberClass(89));
list->Add(new MyNumberClass(12));
list->Add(new MyNumberClass(17));
list->Add(new MyNumberClass(22));
list->Add(new MyNumberClass(94));
list->Add(new MyNumberClass(87));
list->Add(new MyNumberClass(55));
list->Add(new MyNumberClass(33));
list->Add(new MyNumberClass(62));

// Sort the list in ascending order and
// display the results in the memo.
list->Sort(SortAscending);
Memo1->Text = "Ascending Sort";
for (int i=0;i<list->Count;i++) {
  MyNumberClass* mc =
    (MyNumberClass*)list->Items[i];
  Memo1->Lines->Add("Item " +
    String(i) + ": " + String
      (mc->Data));
}
Memo1->Lines->Add("");
Memo1->Lines->Add("");

// Sort the list in descending order and
// display the results in the memo.
list->Sort(SortDescending);
Memo1->Lines->Add
      ("Descending Sort");
for (int i=0;i<list->Count;i++) {
  MyNumberClass* mc =
    (MyNumberClass*)list->Items[i];
  Memo1->Lines->Add("Item " +
    String(i) + ": " + String
      (mc->Data));
}
for (int i=0;i<list->Count;i++) {
  MyNumberClass* mc =
    (MyNumberClass*)list->Items[i];
  delete mc;
}
```

```
  delete list;
}
//-------------------------
        --------------------
void __fastcall TForm1::Button2Click
        (TObject *Sender)
{
  // Create a TList and fill it with instances
  // of the MyNumberClass class.
  TList* list = new TList;
  list->Add(new MyStringClass
        ("Hello there!"));
  list->Add(new MyStringClass
        ("What's going on?"));
  list->Add(new MyStringClass
        ("This is a test."));
  list->Add(new MyStringClass
        ("Nothing new here!"));
  list->Add(new MyStringClass
        ("TurboPower Software"));
  list->Add(new MyStringClass
        ("Borland International"));
  list->Add(new MyStringClass(
        "A short string."));
  list->Add(new MyStringClass
        ("Hello again!"));
  list->Add(new MyStringClass(22));
  list->Add(new MyStringClass(94));
  list->Sort(StringSortAscending);

  // Sort the list in ascending order and
  // display the results in the memo.
  Memo1->Text = "Ascending Sort";
  for (int i=0;i<list->Count;i++) {
    MyStringClass* mc =
      (MyStringClass*)list->Items[i];
    Memo1->Lines->Add(mc->Text);
  }
  Memo1->Lines->Add("");
  Memo1->Lines->Add("");

  // Sort the list in descending order and
  // display the results in the memo.
```

```
  list->Sort(StringSortDescending);
  Memo1->Lines->Add
        ("Descending Sort");
  for (int i=0;i<list->Count;i++) {
    MyStringClass* mc =
      (MyStringClass*)list->Items[i];
    Memo1->Lines->Add(mc->Text);
  }

  // Delete all the objects and then
  // delete the list itself.
  for (int i=0;i<list->Count;i++) {
    MyStringClass* mc =
      (MyStringClass*)list->Items[i];
    delete mc;
  }
  delete list;
}
```

July 1998

Displaying your bitmaps quickly

by Andrea Fasano

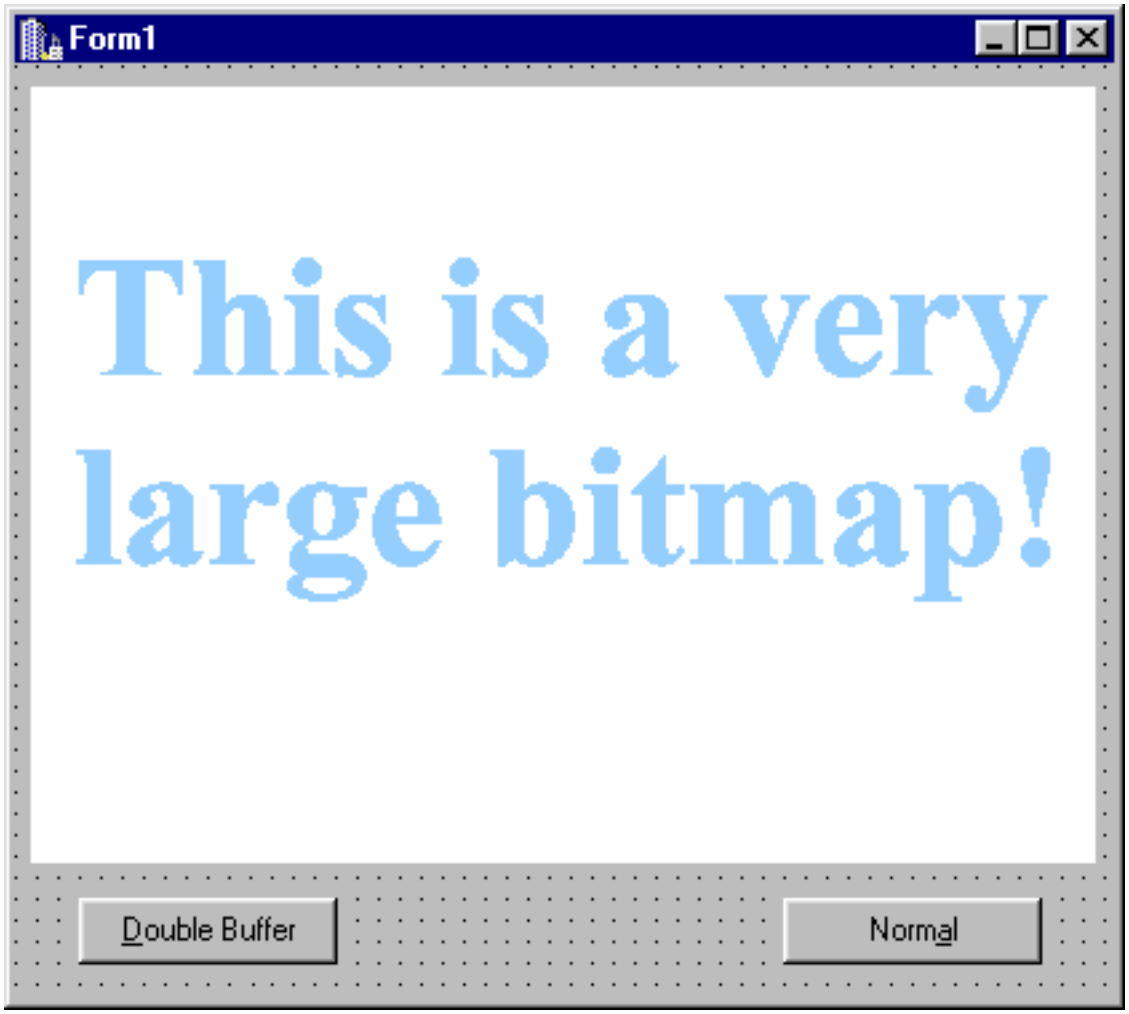
During application development, you must often manage various graphic elements, such as images and icons. Fortunately, C++ Builder provides the developer with several tools that use these graphic elements. In this article, we'll study the concept of offscreen bitmaps and the double-buffering technique. We'll use these tools to help us efficiently manage the graphics in our program.

A simple graphic application

In order to appreciate the advantages of using offscreen bitmaps and double-buffering, we'll first build an application using the normal C++ Builder features. Then, we'll implement the same application using an offscreen bitmap. The application itself will simply filter an image. Specifically, the filter will produce the negative of a given image (for instance, a white picture becomes a black picture). We begin designing our application by following these steps:

1. Create a new application choosing File | New Application from the main menu.
2. Choose an Image component from the Component Palette and place it on the form. (The component is located on the Additional tab.)
3. Now double-click the Image component, Image1, to open the Picture Editor utility.
4. Click the Load button on the Picture Editor. The Load picture dialog box is displayed. Locate and choose the file SampleImage.bmp (you can find this file in the archive). Click OK.
5. Press F11 to switch to the Object Inspector and change the AutoSize property of Image1 to True. Notice how Image1 resizes to accommodate the image.
6. Click on the Standard tab on the Component Palette and choose a Button component. Place the button, Button1, under the image and to the left.
7. Change the Name property to buttonNormal and the Caption property to Normal.
8. Add another button, Button2, and place it under and to the right of Image1.
9. Change Button2's Name and Caption properties to buttonDoubleBuffer, and DoubleBuffer, respectively. At this point the application should resemble Figure A.

Figure A: Your form will first display an image, then display its negative.



The slooooww code

Now we're ready to add the code "behind" the buttons. First, double-click the Normal button to create an OnClick event shell. The Code Editor displays the following method:

```
void __fastcall
TForm1::buttonNormalClick
        (Tobject *Sender)
{

}
```

We'll add the code that produces the negative image here. The steps to achieve a negative image are very simple. First, you must get the color value of each pixel in the image; then you must invert every bit. Afterwards, the new color value will be stored at the same coordinates (x, y) of the old value and displayed on screen. You can read individual pixels simply by using the Pixels property. Just access the Pixels property of the canvas, letting the

indexes be the x- and y-coordinates of the pixel you want to read: Image->Picture->Bitmap->Canvas->Pixels[x][y]. Remember that the upper left corner of the canvas is the origin of the coordinate system. Once you've gained the color, you must do a bitwise exclusive-OR between this value and 0x00ffffff to obtain the inverted color (Thankfully, Windows uses only 24-bits to manage colors). So, for a single pixel, you'll write a line of code similar to this one:

```
Image1->Picture->Bitmap->Canvas->Pixels
        [x][y]= Image1->Picture->Bitmap->
                Canvas->Pixels[x][y] ^
                        0x00ffffff;
```

Because we must execute this operation for each pixel of the image, we need to scan the entire bitmap using two loops. The first loop is for the horizontal size, and the second is for the vertical size. You can determine these values using the properties Image1-> Picture->Width and Image1->Picture->Height. The final code in the buttonNormalClick method should resemble Listing A. Make sure you add all of the highlighted code.

Listing A: The buttonNormalClick method

```
void __fastcall TForm1::buttonNormalClick
        (TObject *Sender)
{
 Screen->Cursor = crHourGlass;

 for (int y=0; yPicture->
        Height; y++)
   for (int x=0; xPicture->
        Width; x++)
     Image1->Picture->Bitmap->Canvas->
        Pixels [x][y] =
       Image1->Picture->Bitmap->Canvas->
        Pixels [x][y] ^0x00ffffff;

 Screen->Cursor = crArrow;
}
```

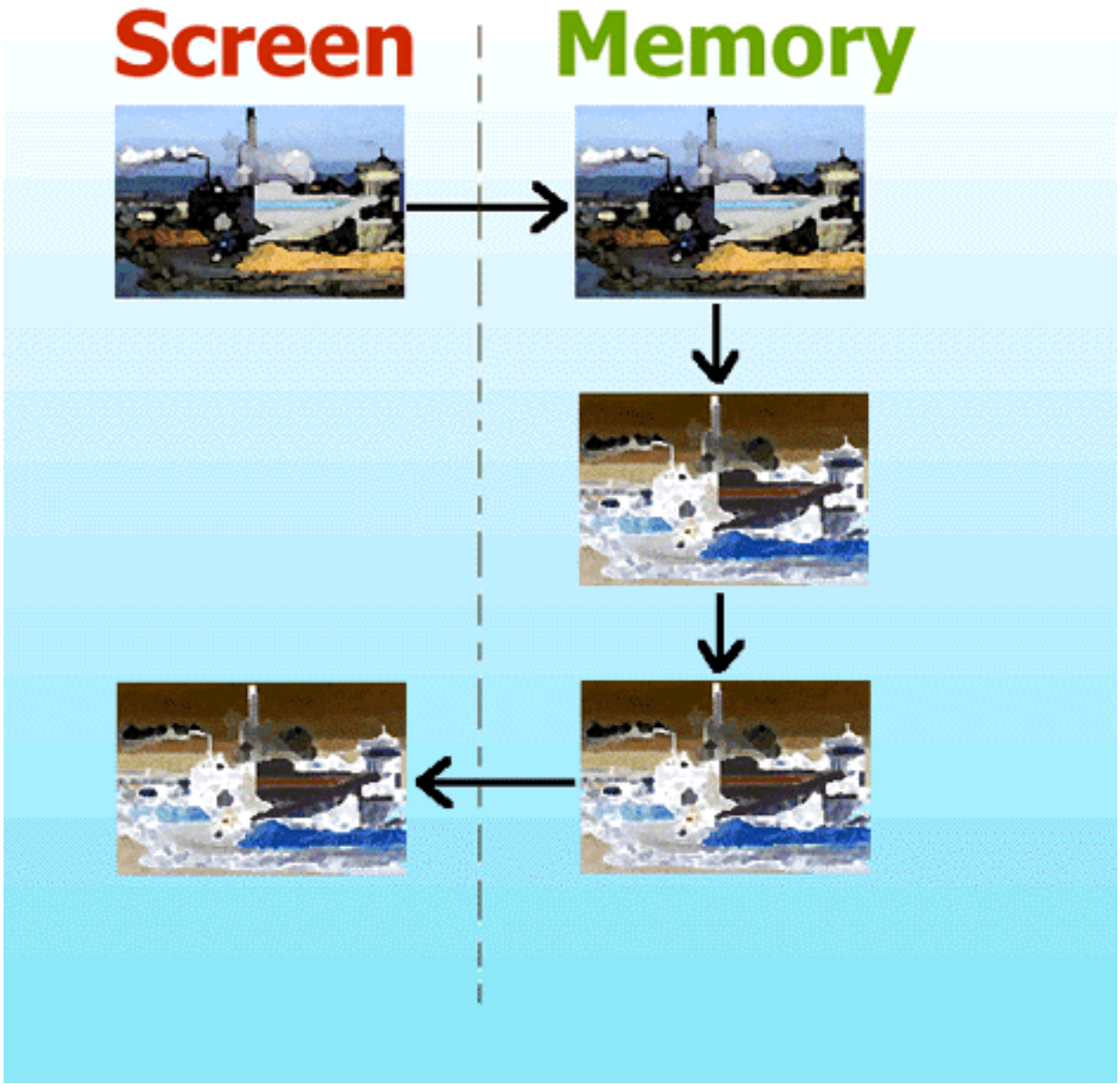
To designate the beginning and the end of the graphic operation within the function, I inserted two lines of code. The first line, Screen->Cursor = crHourGlass, marks the start of the operation by changing the mouse cursor to a sand-glass. The second line, Screen->Cursor

= crArrow, marks the end of the operation by setting the mouse cursor to the normal arrow shape. Now you're ready to test the application for yourself. First, press the [F9] key to compile and run the program. Next, press the Normal button and watch how much time the application takes to finish the graphic operation. (As a reference, the program takes about 10 seconds to produce the negative image on my P166.) Of course, we can't consider this result a good performance, especially because we used a really simple filter. The performance is slow because of the tremendous number of calls necessary to invert the Pixels property. Each time a pixel is read, it's immediately displayed on the screen. Since the sample image is 400x300 pixels, there are a total of 240,000 calls to the Pixels property (each pixel requires one call to read the color and one to write the inverted color) and a total of 120,000 pixels printed. To avoid the performance hit involved in such large numbers, you can use a technique called double-buffering, which we'll discuss next.

Double-buffering

Double-buffering is a simple trick: All the changes applied to the source image aren't immediately displayed. Instead, they're stored in a buffer, the offscreen bitmap. Then, at the end of the graphic operation, the entire buffer is copied onto the screen. This technique is illustrated in Figure B.

Figure B: The original image is copied, updated, copied back, and redisplayed.



Now, let's see how we implement this very useful technique. First, note that you'll use an offscreen bitmap as a buffer. We can describe an offscreen bitmap as simply a work area not visible on the form itself; it also serves as the destination of some graphic operations. The offscreen bitmap stores the changes to the original image. Using a non-visible working area offers many advantages. A primary benefit occurs because a non-visible work area greatly reduces the execution time needed by graphic operations. To create an offscreen bitmap, you add its declaration to the definition of your form, like so:

```
…
Public:
 Graphics::TBitmap *DoubleBuffer;
…
```

You also must remember to initialize and destroy DoubleBuffer when the application starts

and ends. So, select Form1 and switch to the Object Inspector. Locate the OnCreate event and double-click it. Then add the following highlighted code into the Code Editor:

```
void __fastcall TForm1::FormCreate
        (TObject *Sender)
{
 DoubleBuffer = new Graphics::TBitmap;
}
```

Now repeat the preceding steps for the OnDestroy event, and add this line of code:

```
void __fastcall TForm1::FormDestroy
        (TObject *Sender)
{
 delete DoubleBuffer;
}
```

Next, double-click the buttonDoubleBuffer button to create its OnClick event shell. We'll use the double buffer technique to place the code that produces a negative image inside the buttonDoubleBufferClick method. To do so, we'll use three logical steps:

1. Copy the source image (Image1) to DoubleBuffer.
2. Apply the filter to DoubleBuffer.
3. Copy DoubleBuffer to the destination image (Image1).

To perform the first step, we can use the Assign method that copies the bitmap image contained in its Source parameter to the bitmap object. Therefore, you'll write the first line of code similar to this one:

```
DoubleBuffer->Assign
        (Image1->Picture->Bitmap);
```

The second step is very easy to develop. You can reuse the same code that lies inside the buttonNormalClick method. Of course, in this case, you must use DoubleBuffer instead of Image1->Picture->Bitmap. The last step is the opposite of the first one. You assign DoubleBuffer to Image1->Picture->Bitmap. The finished buttonDoubleBufferClick method should resemble Listing B. Be sure you add all the highlighted code.

Listing B: The buttonDoubleBufferClick method

```
void __fastcall TForm1::buttonDouble
```

```
        BufferClick(TObject *Sender)
{
 DoubleBuffer->Assign
        (Image1->Picture->Bitmap);
 Screen->Cursor = crHourGlass;

 for (int y=0; y
        Height; y++)
   for (int x=0; x
        Width; x++)
     DoubleBuffer->Canvas->Pixels
        [x][y] =
            DoubleBuffer->Canvas->Pixels
         [x][y] ^ 0x00ffffff;

 Screen->Cursor = crArrow;
 Image1->Picture->Bitmap->Assign \
        (DoubleBuffer);
}
```

Now it's time to see the offscreen bitmap technique at work. Compile and run the application, and press the DoubleBuffer button. How much time does the application take to produce the negative image? On my PC, it takes less than two seconds! The graphic operation now is about five times faster than before.

Conclusions

In this article, we studied the double buffer technique to make our bitmaps display more quickly. You could use this technique in many other situations and really increase your overall performance. In addition, you saw how to use the Assign method to copy the bitmap contained in one object to another object. As you can see, VCL graphic components offer many other methods to execute the copy. We'll explore these methods in future articles.

June 1998

Detecting disk errors

by Kent Reisdorph

If your application uses floppy disks, then you should be prepared to handle any potential disk errors. A disk error occurs when there's no disk in a floppy drive, when a disk in the floppy drive isn't formatted, when the disk is damaged, or for many other reasons. Handling disk errors gracefully makes your application more robust and saves your users from endless frustration. In this article, we'll explain how to handle disk errors in your application. We'll also look at the SetErrorMode and GetLastError functions of the Windows API.

Why handle disk errors?

Windows handles critical errors at the operating-system level by default. If, for example, you try to write to a floppy disk and the diskette in the drive isn't formatted, Windows displays a message box stating that a disk error has occurred. An example of this type of message box is shown in Figure A.

Figure A: Figure A shows an error message box for an unformatted disk (Windows NT 4).

While handling critical errors at the operating-system level is fine for many applications, some applications need better control of such errors. For example, if your application writes to diskette, you may want to suppress the normal Windows error message box in order to do some processing behind the scenes. Or maybe you just don't like the message boxes that Windows provides and you want to replace them with your own message boxes. Whatever the reason, you occasionally need to take control of critical errors from Windows.

Two steps to success

Handling disk errors is a two-step process. First, we must inform Windows that we don't want it to display critical error messages. We can accomplish this by calling the SetErrorMode function. Second, we must determine whether an error has occurred for any operations that could produce a diskette critical error. The GetLastError function provides an error code for the last system function called. We can use that error code to determine what went wrong. Let's take a look at each of these steps in more detail.

Setting the error mode

As we indicated, Windows usually takes the responsibility for handling system errors. You can override that behavior by calling SetErrorMode with the appropriate flags for the types of errors you want to handle. In this case, the flag in

which we're interested is the SEM_FAILCRITICALERRORS flag. To tell Windows that you want to handle this type of error yourself, call SetErrorMode with the line:

```
SetErrorMode(SEM_FAILCRITICALERRORS);
```

After you've called this function, Windows will no longer handle critical errors. That's not the whole story, though. Make sure you save the old error mode and reset it when you're finished with a particular operation. We've done so in this example:

```
int OldMode;
OldMode = SetErrorMode(SEM_FAILCRITICALERRORS);

// Do some things that might cause an error and then reset the error mode.
SetErrorMode(OldMode);
```

As you might have surmised, SetErrorMode returns the previous error mode setting, thus allowing you to save the current error mode so you can restore it after you've completed a particular operation. Reset the error mode as soon as you've finished your processing. If you fail to do so, then the user may not receive notification of certain errors that are out of your control as a programmer. In short, do the following:

- Set the error mode as desired.
- Do your thing.
- Set the error mode back to what it was before.

By following this guideline, your applications can have custom error handling--yet not interfere with normal Windows operations. By the way, you can catch other errors besides critical errors. See the SetErrorMode topic in the Win32 API help file for complete information.

Detecting errors

Once you've told Windows that you'll be detecting critical errors, you must check with Windows to know when an error occurs. You use the GetLastError function to do this. Let's say, for example, that you wanted to copy a file to a diskette in drive A. You might go about it with the following code:

```
OldMode = SetErrorMode(SEM_FAILCRITICALERRORS);
CopyFile("test.dat", "A:\\test.dat", false);
int error = GetLastError();

// code to handle error here
SetErrorMode(OldMode);
```

Since you could get any number of errors from this operation, you must be prepared to handle them all. For example, the CopyFile function in the previous example could result in any of the following errors:

```
ERROR_FILE_NOT_FOUND
ERROR_PATH_NOT_FOUND
ERROR_TOO_MANY_OPEN_FILES
```

```
ERROR_ACCESS_DENIED
ERROR_NOT_ENOUGH_MEMORY
ERROR_OUTOFMEMORY
ERROR_INVALID_DRIVE
ERROR_NOT_SAME_DEVICE
ERROR_NO_MORE_FILES
ERROR_WRITE_PROTECT
ERROR_BAD_UNIT
ERROR_NOT_READY
ERROR_CRC
ERROR_NOT_DOS_DISK
ERROR_WRITE_FAULT
ERROR_GEN_FAILURE
ERROR_SHARING_VIOLATION
ERROR_LOCK_VIOLATION
ERROR_HANDLE_DISK_FULL
ERROR_FILE_EXISTS
ERROR_CANNOT_MAKE
ERROR_DISK_FULL
ERROR_UNRECOGNIZED_MEDIA
```

And this is just a sampling. Other errors could also apply. (Note**:** For a complete list of error codes, check out WINERROR.H. This header file contains a complete list of Windows error codes and a brief description of each.) *You must take great care to handle any possible errors that might occur as the result of a file operation (or any other operation for that matter).* You don't need to handle each and every error specifically, but you must at least let the user know about errors that you aren't handling. (See the "Getting Error Message Text" sidebar for a description of how to obtain the Windows error message text for errors that you don't handle directly.) You should also be aware of one other thing concerning GetLastError. Windows NT and Windows 95 don't necessarily return the same error code for identical disk error conditions. For example, if you attempt to set the current directory to a floppy drive that contains an unformatted disk under Windows NT, GetLastError will report an error code of ERROR_UNRECOGNIZED_MEDIA. On the other hand, under Windows 95 GetLastError might report ERROR_GEN_FAILURE or ERROR_INVALID_DRIVE for the same operation. Be sure to test your applications thoroughly under both operating systems so that you know your code will work on either platform.

Conclusion

Listings A and B contain a program that illustrates using SetErrorMode and GetLastError to detect disk errors and acting accordingly. As an added bonus, the program uses the FormatMessage function to display any errors not handled by the program. The program checks whether the diskette in drive A is good. If not, it reports an error. You can also get the code for this example from our Web site at www.cobb.com/cpb; click on the Source Code hyperlink. Custom error handling is vital in certain types of applications and a nice feature in others. Although you might not need SetErrorMode often, when you do, you should at least know how it works.

Listing A: ERRMODEU.H

```
#ifndef ErrModeUH
```

```
#define ErrModeUH
//---------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
  __published:  // IDE-managed Components
    TButton *Button2;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
  private:  // User declarations
    void ShowErrorMessage(int code);
  public:  // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------

extern TForm1 *Form1;
//---------------------------------------------

#endif
```

Listing B: ERRMODEU.CPP

```
#include <vcl.h>
#pragma hdrstop

#include "ErrModeU.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  // Display the hourglass cursor.
  Screen->Cursor = crHourGlass;

  // Set the error mode to not show critical
  // errors. Save the current error mode.
  int OldMode = SetErrorMode(SEM_FAILCRITICALERRORS);
```

```
// Set the last error code to -1 so we can
// tell if an error occurred.
SetLastError(-1);

// Attempt to find a file on Drive A. This
// will generate an error if no diskette is
// in the drive or if the diskette is not
// formatted.
WIN32_FIND_DATA fd;
FindFirstFile("A:\\*.*", &fd);

// Display the default cursor.
Screen->Cursor = crDefault;

// Check the last error code.
int error = GetLastError();

switch (error) {
  // No error, diskette is OK. Also,
  // ignore ERROR_FILE_NOT_FOUND error since
  // the disk might be formatted but blank.
  case ERROR_FILE_NOT_FOUND :
  case -1 : {
    MessageBox(Handle, "The diskette is OK.",
      "Disk OK", MB_OK | MB_ICONASTERISK);
    break;
  }

  // Win95 might report a general device
  // failure or an invalid drive rather
  // than unrecognized media as NT does.
  case ERROR_GEN_FAILURE :
  case ERROR_INVALID_DRIVE : {
    int res = MessageBox(Handle, "Error"
      " reading drive A. Either there is no"
      " diskette in the drive or the diskette"
      " is not formatted. Do you want to"
      " format a diskette?", "Disk Error",
      MB_YESNO | MB_ICONQUESTION);

    if (res == IDYES)
      // code to format disk here
    break;
  }

  // If the error is ERROR_UNRECOGNIZED_MEDIA
  // then assume that the diskette is not
  // formatted.
```

```
    case ERROR_UNRECOGNIZED_MEDIA : {
      int res = MessageBox(Handle, "The disk in"
        " drive A is not formatted. Do you wish"
        " to format it?", "Disk Error",
        MB_YESNO | MB_ICONQUESTION);
      if (res == IDYES)
        // code to format disk here
      break;
    }

    // If the error is ERROR_NOT_READY then most
    // likely there is no diskette in the drive.
    case ERROR_NOT_READY : {
      MessageBox(Handle, "Drive A is not ready."
        " Be sure a diskette is in Drive A and"
        " try again.", "Disk Error",
        MB_OK | MB_ICONWARNING);
      break;
    }

    // Some other error occurred, so let
    // the user know.
    default :
      ShowErrorMessage(error);
  }

  // Reset the error mode.
  SetErrorMode(OldMode);
}

void TForm1::ShowErrorMessage(int code)
{
  char buff[1024];
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0, code, 0, buff, sizeof(buff), 0);
  MessageBox(Handle, buff, "System Error", MB_OK | MB_ICONWARNING);
}
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

June 1998

File operations

by Kent Reisdorph

Perhaps you're working with an application that performs a lot of file manipulation. Your application may have to copy files, move files, delete files, or rename files. Many programmers--even some very experienced programmers--don't know that the Windows API has functions for all of these operations. Programmers coming from DOS or even from 16-bit Windows may overlook these functions because they've become accustomed to doing things "the hard way." In this article, we'll show you how to use the basic file manipulation functions in the Win32 API. We'll even explain how to copy and move files complete with animation and full prompting.

Basic file operations

Basic file operations in 32-bit Windows are very straightforward. We won't spend much time going over these functions since they're so easy to use. These functions include CopyFile, CopyFileEx, DeleteFile, MoveFile, and SHFileOperation.

CopyFile and CopyFileEx

The CopyFile function copies a file from one location to another. The basic syntax is as follows:

```
CopyFile("src.fil", "dest.fil", true);
```

You can copy files to the same drive, to another drive, or even to another machine on a network. The third parameter indicates what Windows should do if the file already exists in the target location. When this parameter is true, then Windows doesn't overwrite the existing file; it reports an error instead. When this parameter is false, then Windows will overwrite the existing file.

You can attain more control over copy operations by using CopyFileEx rather than CopyFile. Among other things, CopyFileEx allows you to use a callback function that's called periodically during the copy operation. You could use this callback function to display the progress of the copy operation. Unfortunately, CopyFileEx is available only in the Windows NT API.

DeleteFile

The DeleteFile function is even easier to use than CopyFile. Here's an example:

```
DeleteFile("somefile.txt");
```

DeleteFile deletes a single file only--wild cards aren't allowed. Under Windows 95, DeleteFile will delete a file even if it's currently in use, so take care when deleting files under Windows 95.

MoveFile and MoveFileEx

You can use the MoveFile function to move an individual file or an entire directory. The basic syntax is

```
MoveFile("source.fil", "destination.fil");
```

You can also rename a file using MoveFile, as shown in this example:

```
MoveFile("MyFile.dat", "MyFile.old");
```

MoveFile also has an extended version called MoveFileEx that provides more control over file move operations.

File operations the cool way

The previously-mentioned file operation functions are the unglamorous ways of performing file operations. The SHFileOperation function (one of the Shell API functions) provides a higher-level route to file operations. This is the same function Windows Explorer uses to copy, move, or delete files. The main advantage to using SHFileOperation is that the Windows shell user interface is included in the bargain. For example, when you copy a file using SHFileOperation, you can elect to display the file-copy animation that Windows Explorer uses when copying files (the piece of paper flying from one folder to the next).

Before you can use SHFileOperation, you must include the SHELLAPI.H header file. Let's first look at the function declaration for SHFileOperation, then we'll discuss how to use this function. Here's the declaration:

```
int SHFileOperation(LPSHFILEOPSTRUCT lpFileOp);
```

This function takes a pointer to a SHFILEOPSTRUCT and returns an integer that indicates success or failure. As you might guess, the real work is in filling out the SHFILEOPSTRUCT structure. Table A lists the members of this structure and a description of each.

Table A: Members of SHFILEOPSTRUCT

Data Member	Description
HWND hwnd	The window handle of the parent window.
UINT wFunc	The function to perform (FO_MOVE, FO_COPY, FO_DELETE, or FO_RENAME).
char* pFrom	The source file or directory.
char* pTo	The destination file or directory.
WORD fFlags	Flags that control how the file operation will be carried out.
BOOL fAnyOperationsAborted	After SHFileOperation returns, this member will be true if the user cancelled any aspect of the operation.
void* hNameMappingsa	Used to specify name mappings.
char* lpszProgressTitle	The title of the progress dialog, used only if flags include FOF_SIMPLEPROGRESS.

The hwnd and wFunc parameters are self-explanatory. However, the pFrom field requires some explanation. You use this field to specify the files to copy, rename, move, or delete. Unlike the CopyFile function, wild cards are allowed. In addition, you can specify a file list. The situation becomes confusing at this point because each file in the file list must be separated by a terminating null character and the last file in the list must be followed by a double terminating null.

To make matters worse, the pFrom field must *always* end in a double terminating null even when specifying just one file or a file mask (such as *.*). The AnsiString class, used so much by VCL, doesn't lend itself well to this kind of arrangement. It's better to use good old character arrays when specifying filenames or file masks. Frequently, however, you'll take the filename from an AnsiString property, such as TEdit's Text property, or TOpenDialog's

FileName property.

Your best bet, then, is to create a character array, fill it with zeros (thus solving the double terminating null problem), and copy the contents of the AnsiString to the character array. Look at this example:

```
// Create a char array and fill it with zeros.
char src[MAX_PATH];
memset(src, 0, sizeof(src));
// Create a SHFILOPSTRUCT and zero it, too.
SHFILEOPSTRUCT fos;
memset(&fos, 0, sizeof(fos));
// Copy from an AnsiString to the char array.
strcpy(src, OpenDialog->FileName.c_str());
// Assign to the pFrom field of SHFILEOPSTRUCT.
fos.pFrom = src;
```

Although this technique may seem like a lot of trouble, it is necessary nevertheless. This same scenario applies to the pTo field as well. Once you know how to handle the pFrom and pTo fields, the rest of the process is relatively easy.

Another field that requires some explanation is the fFlags field. Table B lists the primary values that you can use for the fFlags field. You can use these either singly or in combination. These flags provide a great deal of control over how SHFileOperation carries out its tasks.

Table B: Primary flags for the fFlags field

Flag	Description
FOF_ALLOWUNDO	Sends deleted files to the recycle bin, as well as allowing other undo information.
FOF_FILESONLY	Performs file operations only on files if a wild card is specified (*.*, for example).
FOF_MULTIDESTFILES	Stipulates that the pTo field contains a list of files rather than a directory where all files should be placed.
FOF_NOCONFIRMATION	Performs the operation without asking for confirmation (same as the user pressing Yes To All button).

FOF_NOCONFIRMMKDIR	If a directory must be created, it will be created without prompting the user.
FOF_RENAMEONCOLLISION	Creates a new filename (Copy of TEMP.TXT for example) if the file being copied, moved, or renamed already exists.
FOF_SILENT	Performs the operation without displaying a status dialog.
FOF_SIMPLEPROGRESS	Shows a simple progress box but doesn't display the filename.

The fAnyOperationsAborted field of the SHFILEOPSTRUCT is a pointer to a Boolean variable that will contain true if the user cancelled any operation or false otherwise. You can check this field after SHFileOperation finishes to see whether the user cancelled any part of the file operation.

The hNameMappings field is beyond the scope of this article, so we won't cover it here. You can set this parameter to 0 when calling SHFileOperation. You use the lpszProgressTitle field to specify a string where the filename would normally appear. This flag applies only if the fFlags field includes FOF_SIMPLEPROGRESS.

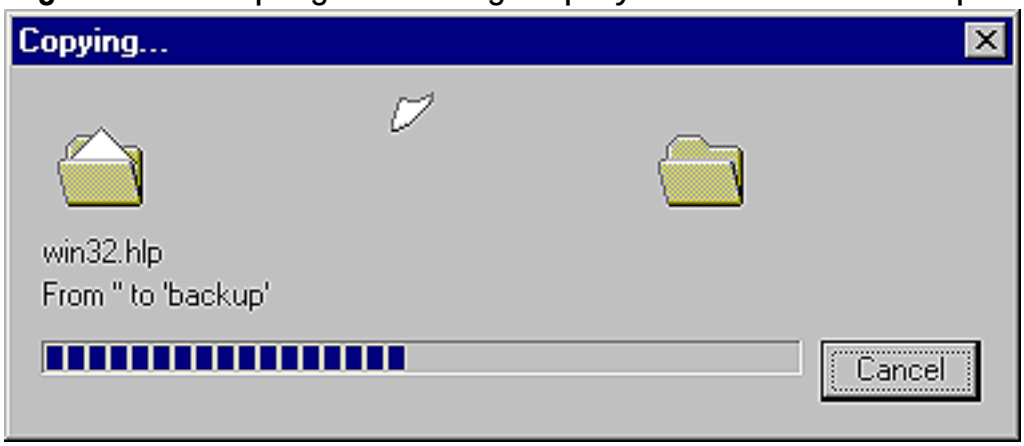
Right about now you're probably ready for an example. Let's assume you wanted to copy all of the files in the current directory to a directory on the current drive called BACKUP. Let's further assume that you want to allow undo and that you don't want to be prompted if the destination directory doesn't exist. In that case, the code would look like this:

```
// Create the source and destination buffers
// and fill them with zeros.
char src[MAX_PATH];
char dst[MAX_PATH];
memset(src, 0, sizeof(src));
memset(dst, 0, sizeof(dst));
// Create the SHFILEOPSTRUCT and zero it.
SHFILEOPSTRUCT fos;
memset(&fos, 0, sizeof(fos));
// Set hwnd to the Handle property to make the
// application the owner of the progress dialog.
fos.hwnd = Handle;
// Specify a copy operation.
fos.wFunc = FO_COPY;
// Set the from and to parameters.
strcpy(src, "*.*");
fos.pFrom = src;
```

```
strcpy(dst, "\\backup");
fos.pTo = dst;
// Set the flags.
fos.fFlags = FOF_ALLOWUNDO | FOF_NOCONFIRMMKDIR;
// Do it.
SHFileOperation(&fos);
```

Figure A shows the progress dialog that's displayed when this code executes. Keep in mind that if the copy operation is very short, the progress dialog won't be displayed. That's one reason I use WIN32.HLP when testing these things--it's 12MB, so you can really see what's going on!

Figure A: This progress dialog displays when the SHFileOperation executes.

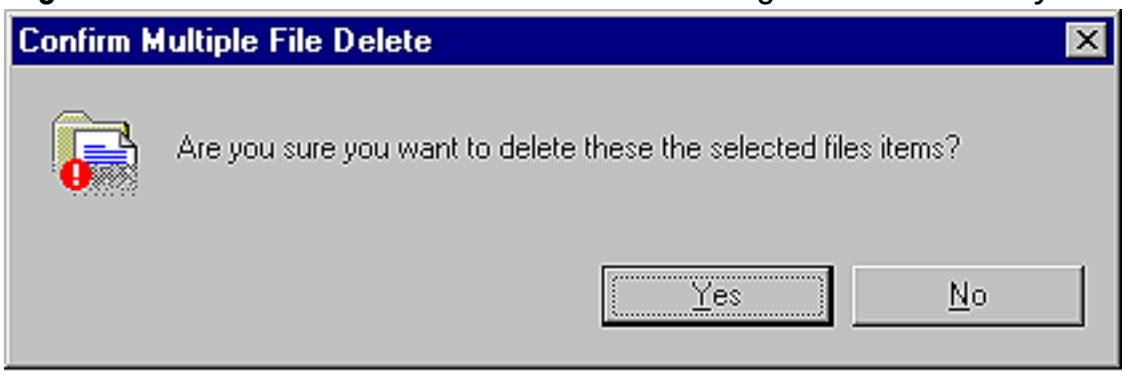


As long as you've set the FOF_ALLOWUNDO flag, you can go to Windows Explorer and choose File | Undo from the main menu to undo the last operation. If the last operation was a delete (FO_DELETE), then the deleted files will be in the Windows Recycle Bin as well. Speaking of deleting files, let's do one more example. Here's how you could delete all of the files in a directory called BACKUP. We'll skip the preliminary code and just show you the pertinent parts:

```
fos.wFunc = FO_DELETE;
strcpy(src, "\\backup\\*.*");
fos.pFrom = src;
fos.fFlags = FOF_ALLOWUNDO;
SHFileOperation(&fos);
```

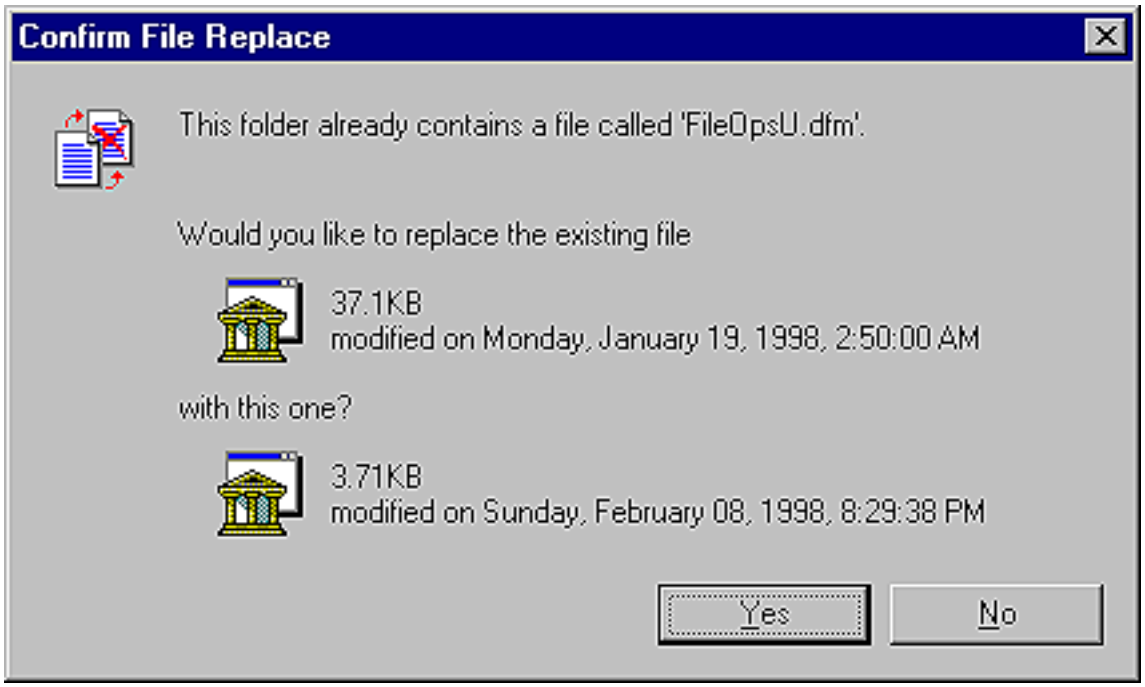
Figure B shows the dialog that's displayed when this code executes. If we'd set the FOF_NOCONFIRMATION flag, then this dialog wouldn't have displayed, and all files would've been deleted without user intervention.

Figure B: The delete files confirmation dialog double checks your decision to delete files.



Windows also shows a confirmation dialog when copying files if a file exists in the target location and neither the FOF_NOCONFIRMATION nor the FOF_RENAMEONCOLLISION flags are set. The confirmation is the usual Do You Want To Replace This File dialog, as shown in Figure C.

Figure C: The file overwrite confirmation dialog offers you another chance to change your mind.



SHFileOperation is a very useful function. It can easily handle all of your file copy, move, delete, and rename operations. As an added bonus, it gives your applications a polished, professional look.

Conclusion

Performing file operations in Windows programs is easy once you know which functions are

available to you. Our Web site contains an example program called FILEOPS that illustrates the use of the CopyFile, CopyFileEx, DeleteFile, MoveFile, and SHFileOperation functions. Go to www.cobb.com/cpb. Click on the Source Code hyperlink.

June 1998

Formatting a floppy from within an application

by Kent Reisdorph

Whenever you have an application that allows saving data to a floppy disk, you may run into situations where the disk in the floppy drive is unformatted. At that point, you have two choices. First, you could ask the user to format a floppy and put it into the drive--a clumsy solution at best. A better way is to allow the user to format the floppy from within your application. Although this may seem like a simple task, the truth is that there's no obvious way to format a floppy from within a Windows program. In this article, we'll show you how to format a floppy via code using the undocumented Shell API function, SHFormatDrive.

This should be easy...

At first glance, it appears that formatting a floppy should be a very simple task. Surely the VCL contains a routine that will copy files? If not, there's certainly a Windows API function, right? Sorry, but no to both. You could use the DOS function format by shelling out to the command prompt. While that command might work, it's a hack solution. Besides, it doesn't give your user any control over disk-formatting options. Next, you turn to the Windows API. After hours of searching the Win32 API online help, you stumble on the DeviceIoControl function and its IOCTL_DISK_FORMAT_TRACKS flag. Sure enough, DeviceIoControl will allow you to format a disk. Unfortunately, since it's a typical Windows API function, you're required to write dozens of lines of code to carry out what should be a simple operation. To make matters worse, you discover that you can't use DeviceIoControl anyway, since the IOCTL_DISK_FORMAT_TRACKS flag is available only in Windows NT. Is there any hope of finding a documented way of formatting a floppy in Windows, something that will work in both Win95 and NT, something that will give you and your users control over disk-formatting operations? Unfortunately, the answer is no.

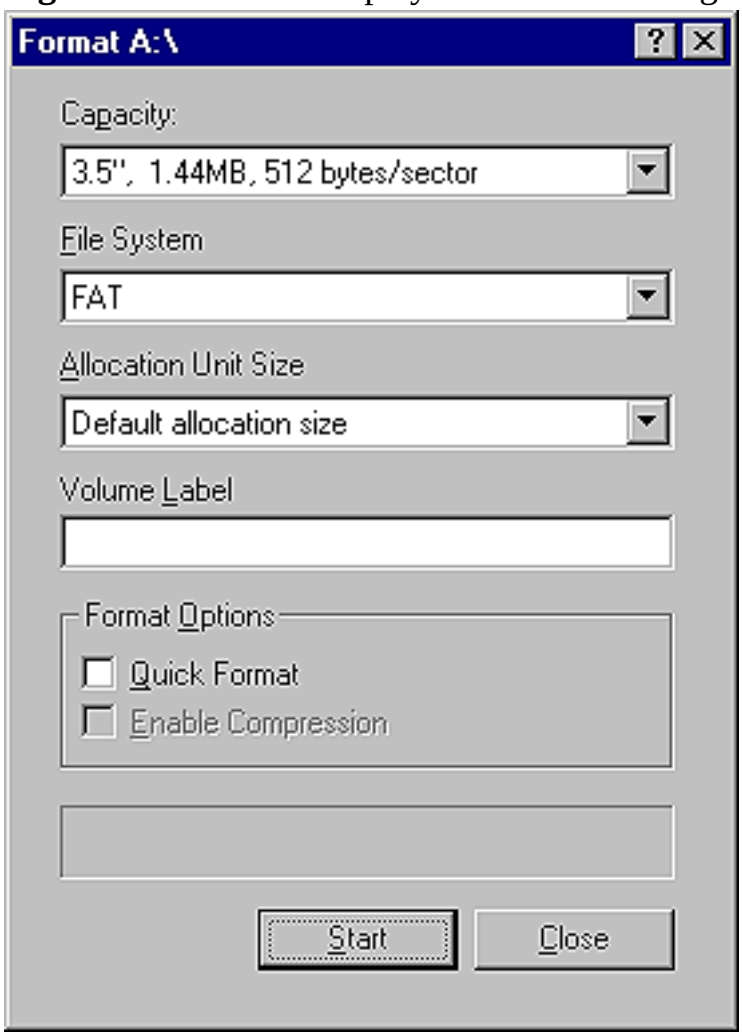
Undocumented Windows: SHFormatDrive

If the story ended there, this wouldn't make much of an article! Fortunately, the story doesn't end on such a sad note. As a matter of fact, the Windows Shell API is a very powerful programming interface. Using the Shell API, you can execute programs (ShellExecute), copy, rename, move, or delete files (SHFileOperation), allow users to select a directory (SHBrowseForFolder), create shortcuts (via IShellLink), create tray icons (Shell_NotifyIcon), and much, much more. In this case, we're interested in an undocumented Windows Shell API function called SHFormatDrive. This function gives you everything you need to format a floppy disk from your C++Builder programs. Why this function isn't

documented is a mystery. SHFormatDrive isn't one of those super-secret Windows internal functions that only those with the secret decoder ring can use. More likely, it was just overlooked when the Shell API was documented. For whatever reason, it remains undocumented even though years have passed since the Shell API was written.

When you call SHFormatDrive, Windows will display the Format dialog. The exact dialog you get depends on whether you're running Windows 95 or Windows NT. Figure A shows the Format dialog as it appears under Windows NT 4.

Figure A: Windows displays the Format dialog when you call SHFormatDrive.



Now, let's look at what you need to do to call SHFormatDrive in your C++Builder programs.

Declare SHFormatDrive and constants

Since SHFormatDrive is undocumented, it's not declared in the SHELLAPI.H header file. (Although SHFormatDrive is undeclared in SHELLAPI.H, it *is* contained in SHELLAPI.LIB and ready to use once you've declared it.) Because SHFormatDrive isn't declared in the Windows headers, you must declare the function, and the constants it uses, yourself. The declaration of the constants and the function itself

resembles the following:

```
#define SHFMT_ID_DEFAULT   0xFFFF

#define SHFMT_OPT_QUICK    0x0000
#define SHFMT_OPT_FULL     0x0001
#define SHFMT_OPT_SYSONLY  0x0002

#define SHFMT_ERROR    0xFFFFFFFFL
#define SHFMT_CANCEL   0xFFFFFFFEL
#define SHFMT_NOFORMAT 0xFFFFFFFDL

extern "C"
DWORD WINAPI SHFormatDrive(HWND hwnd,
                           UINT drive,
                           UINT fmtID,
                           UINT options);
```

Let's take a moment to discuss the constants, then we'll go over the function declaration. First, the SHFMT_ID_DEFAULT constant is a special value that's passed for the fmtID parameter of SHFormatDrive--at this time, the only valid value that can be passed for the fmtID parameter.

The next group of constants (SHFMT_OPT_QUICK, SHFMT_OPT_FULL, and SHFMT_OPT_SYSONLY) are flags that determine which format options are selected when the Format dialog is initially displayed. If SHFMT_OPT_QUICK is specified, then the Quick Format check box on the Format dialog will be checked when the dialog is displayed. If you specify SHFMT_OPT_FULL, then the Quick Format box is cleared. However, if you specify the SHFMT_OPT_SYSONLY flag, then the Copy System Files option will be checked when the Format dialog is displayed.

Note that this option applies only to Windows 95. If you attempt to specify SHFMT_OPT_SYSONLY under Windows NT, the call to SHFormatDrive will fail. These flags only determine which options on the Format dialog are selected. The user can always change the options before formatting the disk.

You'll use the SHFMT_ERROR, SHFMT_CANCEL, and SHFMT_NOFORMAT constants to check the return value of SHFormatDrive for errors. We'll discuss the return value of SHFormatDrive in just a moment.

Calling SHFormatDrive

Now that you have declared the SHFormatDrive function, you can call it from your C++Builder applications. Let's take another look at the function declaration for SHFormatDrive. Here it is again (minus the extern "C" specifier):

```
DWORD WINAPI SHFormatDrive(HWND hwnd,
                           UINT drive,
                           UINT fmtID,
                           UINT options);
```

The hwnd parameter specifies the window handle of the window that should act as the parent for the Format dialog. For C++Builder VCL applications, you should use your form's **Handle** property here. You'd use the drive parameter to specify the drive to format. The A drive is drive 0, the B drive is drive 1, and so on. Attempting to set the drive number to an invalid drive number or to any hard drive will result in an error. The fmtID parameter isn't fully implemented and must be set to SHFMT_ID_DEFAULT. As discussed earlier, you'll use the options parameter to tell Windows which options on the Format dialog should be set when the dialog is displayed.

Let's say you wanted to format the A drive and prompt the user to do a full format. In that case, the call to SHFormatDrive would look like this:

```
SHFormatDrive(Handle,
  0, SHFMT_ID_DEFAULT, SHFMT_OPT_FULL);
```

The process is really pretty simple once you've declared SHFormatDrive and its constants. This example ignores the return value since we haven't discussed the return value yet. We'll look at that next.

SHFormatDrive return values

The value returned from SHFormatDrive varies depending on whether the user is running Windows 95 or Windows NT. You should either detect the operating system being used or plan for the lowest common denominator should an error occur during formatting. By lowest common denominator, we mean that you should be prepared to handle any and all possible return values returned by SHFormatDrive. You might find it interesting to note that Windows

95 actually handles SHFormatDrive better than does Windows NT. Under Windows 95, the return value will be a positive number if the format was successfully carried out; or if an error occurred, the result will be one of the error constants mentioned earlier.

The error code returned is usually either SHFMT_ERROR or SHFMT_CANCEL. The SHFMT_ERROR code occurs if an error occurs when formatting a disk. Such an error might occur if the disk in the drive is bad, if the user removes the disk from the drive during the format, if the disk type is wrong (attempting to format a 720K disk in the 1.44 format), or for any number of other reasons.

SHFormatDrive returns SHFMT_CANCEL if the user cancels the format operation. The SHFMT_NOFORMAT code is returned if the disk in the drive can't be formatted. Although it's theoretically possible to get an error code of SHFMT_NOFORMAT, I've yet to see that error code in practice.

Windows NT handles SHFormatDrive differently. The return value is 0 if the function succeeded or SHFMT_ERROR if the disk wasn't formatted. Put another way, NT doesn't differentiate between the user canceling the format or an error occurring. In either case, SHFormatDrive will return SHFMT_ERROR under Windows NT.

Naturally, you should check the return value of SHFormatDrive. Then, take appropriate action if the disk in the drive remains unformatted.

Fortuitous floppy formatting

Formatting a floppy disk in Windows is easy once you know the secret of SHFormatDrive. Listing A contains the main unit of a program that formats a floppy disk in drive A at the click of a button. The program checks the return value of SHFormatDrive and displays a message accordingly. This code will work better on Windows 95 than on Windows NT since the error codes are more meaningful on Windows 95. Unfortunately, SHFormatDrive has no option to format the disk silently without user intervention. That inflexibility may not be all bad, though, since formatting a drive without letting the user know is probably not a good idea anyway. Knowing about SHFormatDrive will save you lots of time and hair-pulling when you must format a floppy disk from within your applications.

Listing A: FMTDRIVE.CPP

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#define SHFMT_ID_DEFAULT   0xFFFF

#define SHFMT_OPT_QUICK    0x0000
#define SHFMT_OPT_FULL     0x0001
#define SHFMT_OPT_SYSONLY  0x0002

#define SHFMT_ERROR    0xFFFFFFFFL
#define SHFMT_CANCEL   0xFFFFFFFEL
#define SHFMT_NOFORMAT 0xFFFFFFFDL

extern "C" DWORD WINAPI
SHFormatDrive(HWND, UINT, UINT, UINT);

#include "FormatU.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  int Res = SHFormatDrive(Handle,
    0, SHFMT_ID_DEFAULT, SHFMT_OPT_FULL);
  if (Res < 0) {
    String S;
    switch (Res) {
      case SHFMT_ERROR : {
        S = "Error formatting disk.";
        break;
      }
      case SHFMT_CANCEL : {
        S = "Format cancelled.";
        break;
      }
      case SHFMT_NOFORMAT :
        S = "This disk cannot be formatted.";
    }
    MessageBox(Handle, S.c_str(),
```

```
      "Error", MB_ICONEXCLAMATION);
  }
  else
    MessageBox(
      Handle, "Disk formatted successfully!",
      "My Application", MB_OK);
}
```

June 1998

Getting error message text

by Kent Reisdorph

Although you may have your application set to handle critical errors, you probably won't handle every error that might occur. For those errors you don't handle, you'll need to inform the user that an error occurred. Fortunately, the FormatMessage function provides an easy way to do just that. This Win32 API function takes the error code returned from GetLastError and returns the error message text associated with that error code. Using FormatMessage, you can write a short function that displays the error message text for any Windows error. The function would look like this:

```
void TForm1::ShowErrorMessage(int code)
{
  if (code == 0) return;
  char buff[1024];
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,
    0, code, 0, buff, sizeof(buff), 0);
  MessageBox(Handle, buff,
    "System Error", MB_OK | MB_ICONWARNING);
}
```

Your error handling code, then, might resemble these lines:

```
SetErrorMode(SEM_FAILCRITICALERRORS);
// Some code here which might cause an error
int error = GetLastError();
if (error == ERROR_PATH_NOT_FOUND)
  // Do something!
else ShowErrorMessage(error);
```

Using such code, you can catch any errors that interest you and notify the user of any other errors that occur.

June 1998

Using Panel components to simplify resizing

If you've used C++Builder for any significant project, you've probably noticed the Align property of many visual components. However, you may not have realized how useful this property can be. For many applications, you'll want to create on a form some static regions that maintain relative position as the user resizes the form. It's common for beginning C++Builder programmers to perform resizing tasks using the **OnResize** event, moving or resizing individual controls as needed.

However, you can use Panel components to simplify this process and eliminate much of the tedious resizing logic. In this article, we'll show how you can use a Panel component's **Align** property to handle many of these tasks--with little or no code.

Edge alignment vs. client alignment

If you examine the possible values for the **Align** property, you'll find alBottom, alClient, alLeft, alNone (the default value), alRight, and alTop. As you could guess, alNone specifies no automatic alignment; the Panel component's size and position will be a result of changes you make. The remaining values fall into two categories: edge and client alignment values. The alClient property value specifies that C++Builder should resize and position the control so that it covers the entire client region of the parent window.

In contrast, the alBottom, alLeft, alRight, and alTop values specify an edge of the client region that the control will adhere to. For example, if you specify a value of alTop for a control's **Align** property, the control will maintain the design-time height, but C++Builder will adjust the control's width to match the form's client area width.

Panels as owners

A Panel component is one on which you can place other components. Any components that you place on a panel will then recognize the panel as their owner. The panel is responsible for initializing and destroying the components it owns. In addition, the position of the components on the panel will be relative to the upper-right corner of the panel itself. For instance, if you place a Panel component on a form and place a Button component on the panel, moving the panel will also move the button.

If you combine the characteristics of Panel components with the setting of the **Align** property to various values, you'll notice that you can use these features in tandem. However, C++Builder specifies the

following default order of evaluation for alignment options:

- alBottom and alTop
- alLeft and alRight
- alClient

This code means that if you place two Panel components on a form and set one's **Align** property to alTop and the other's to alLeft, the one set for alTop will always appear above the alLeft one--no matter which one you place first!

Panel delivery

Now let's create an example that demonstrates how you can use Panel components to manage the placement of components on complex forms. To begin, create a blank-form project and place two Panel components on it. Set the **Align** property of the first panel to alLeft and its **Width** property to 100. Set the **Align** property of the second panel to alClient. Next, place a third Panel component on the second, and set its **Align** property to alTop and its **Height** property to 50. Finally, place a Memo component below the third panel (but on the second), and set its **Align** property to alClient.

Now, build and run the application. When the main form appears, you'll notice that resizing the form doesn't affect the location of the Memo component's upper-right corner, as shown in Figures A and B.

Figure A: You can use Panel components...

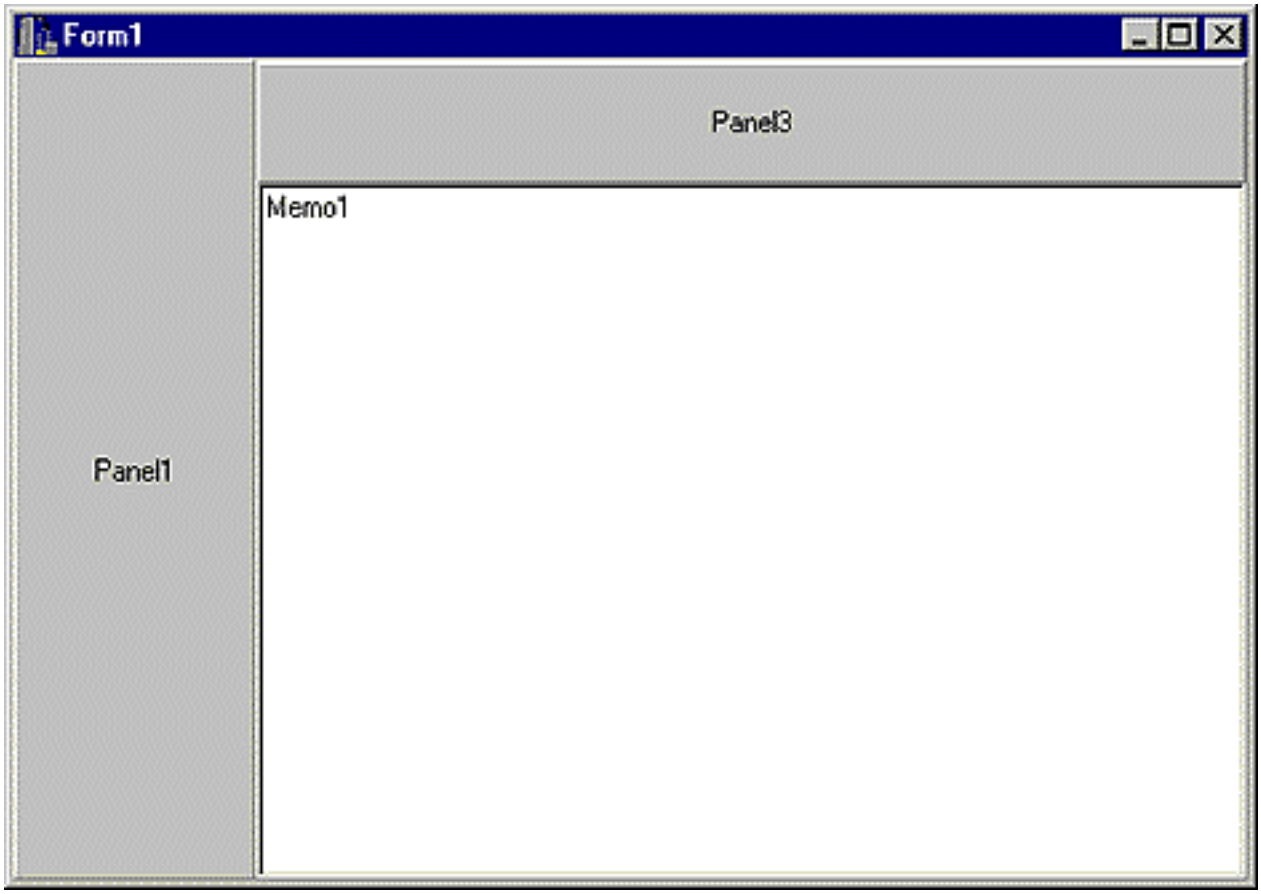
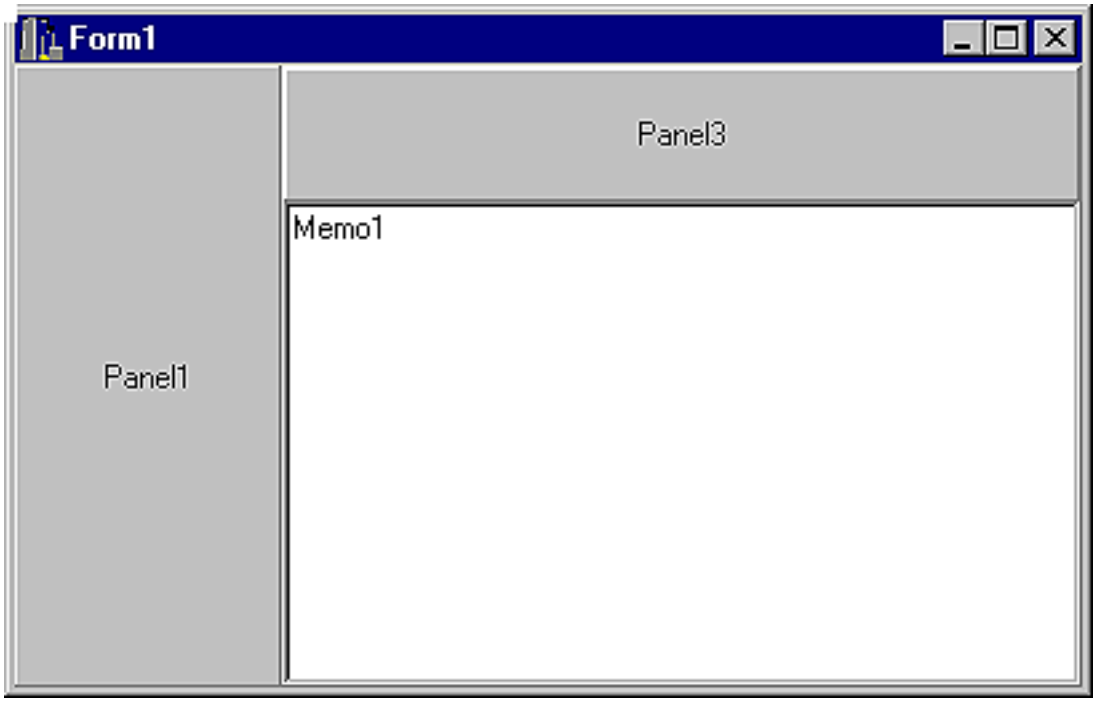


Figure B: ...to control the position of other components relative to each other.



Problems with MDI forms

For Single Document Interface (SDI) applications, the above technique works well. However, there are limits to how you can use this technique for a Multiple Document Interface (MDI)

application. Specifically, if you cover the main form entirely with Panel components, you won't be able to see the MDI child forms because the position of the child forms is relative to the main form area that's not covered by Panel components.

Conclusion

You may not always be able to use code in an **OnResize** event handler to correctly modify component positions and size. However, by using combinations of Panel components and **Align** property settings, you can greatly simplify the task of sizing and positioning many form elements.

June 1998

Exploring the Menu Designer

Rarely do you see a Windows program that doesn't include a menu bar. For that reason, your users have definite expectations when it comes to the menus you create. They'll assume the menus are organized in typical Microsoft manner. They'll also anticipate that these menus are accessible by standard mouse and keystroke controls. There's no reason your programs can't meet those expectations. Borland had the foresight to include a Menu Designer utility with C++Builder. With it, you can prepare main and popup menus while organizing and assigning accessibility to their items. The utility also comes with menu templates and allows you to prepare and save your own templates. In addition to these features, the tool works with the Object Inspector, making it easy to attach code to any of the menu items.

In this article, we'll show you how to get the most from Menu Designer. Along the way, we'll discuss the typical expectations of users and some details about the TMainMenu and TPopupMenu components. We'll finish with an example that walks you through the Menu Designer.

Great expectations

For better or worse, the folks at Microsoft have created some widely accepted standards. One is the Windows main menu bar. Most instances of these bars contain the same selections: File, Edit, View, Window, and Help. Each of these in turn has typical sub-items. We won't describe them all, but File generally has New..., Open..., Close, Save, Save As..., Page Setup, Print, and Exit. Like any other rule, you'll find exceptions to this one, too. For example, Microsoft's own Calculator accessory lacks File and Window menus. Sometimes, you'll find application-specific menu items. A brokerage application, for example, may have a menu to access common financial formulas.

Another standard concerns how users access menu items. There are three access methods: mouse navigation, accelerator keys, and shortcuts. We're all familiar with using the mouse, so let's focus on the other two methods. *Accelerator keys*, or just plain accelerators, consist of an underlined letter in the menu item's caption, i.e., Print Preview.

You define an accelerator by placing an ampersand (&) symbol in the **Caption** property of the menu item. More specifically, you put it front of the key letter, i.e., *Print Pre&view*. Now, the user selects the menu item by pressing [Alt] plus its accelerator letter, i.e., [Alt] V. Because they're arranged in a hierarchy, like menus, the user may have to use multiple accelerators to select the desired item.

Shortcuts save even more time because they don't require the user to follow the menu hierarchy. Instead, you assign a series of unique keystrokes to a menu item. When the user types the keystroke series, your

program invokes the event handler of that menu item. You define a shortcut by selecting it from the menu item's **ShortCut** property dropdown list.

Accelerators and shortcuts aren't without their flaws. Straying from the accepted standards will confuse and irritate users. Also, be warned that C++Builder doesn't protect you from assigning duplicate accelerators and shortcuts. It's up to you to police your own assignments. In case you're wondering, it's OK to assign the same accelerator to two or more menu items. However, the items must be in separate menu hierarchies, i.e., File|Close and Edit|Copy.

Using TMainMenu

The TMainMenu component does a lot and asks for little in return. It encapsulates all the functionality (properties, methods, and events) for your form's menu bar. It associates dropdown menus and submenus with accelerators and shortcuts. Most importantly, it connects items to event handlers. The items on the menu bar and in its dropdown menus are specified with the **Items** property of TMainMenu. The **Items** property is an array of type TMenuItem. Each TMenuItem stores the properties, methods, and events for each menu item. Your application will use the **Items** property to access a particular command on the menu.

Using TPopupMenu

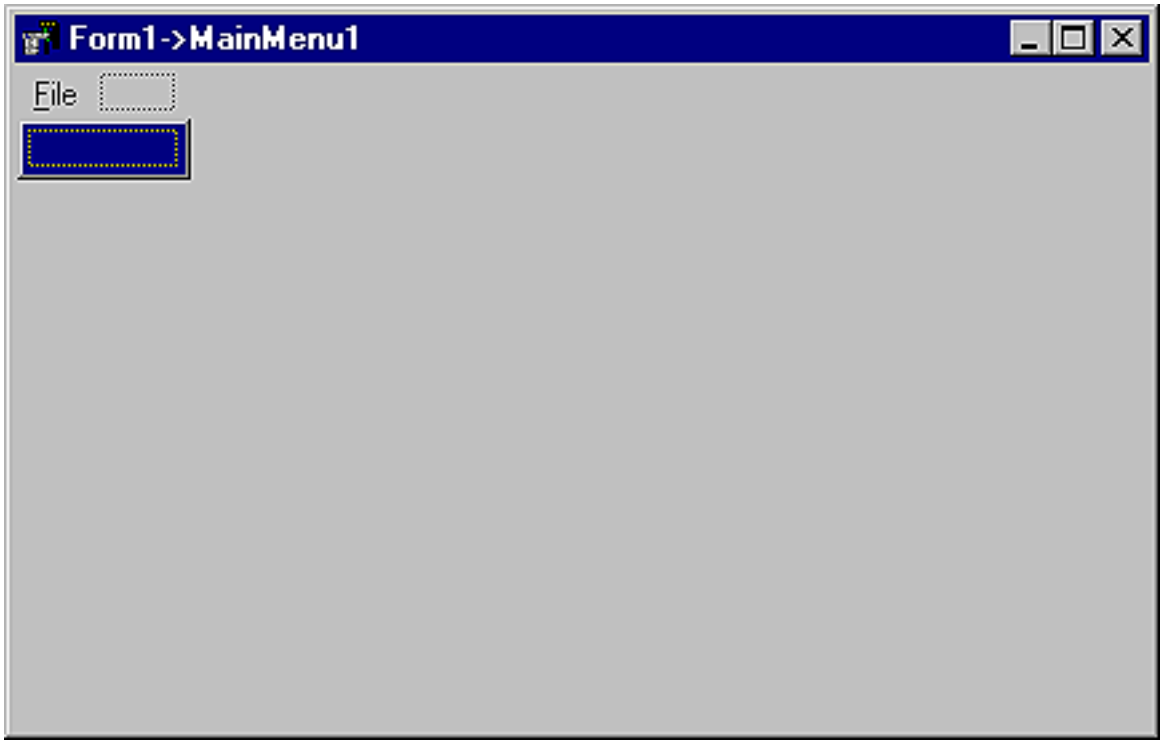
The TPopupMenu component shares many of the characteristics of TMainMenu. It too encapsulates functionality--this time for a pop-up menu. Pop-up menus are available to forms and controls when the user selects the component and clicks the right mouse button. To make a pop-up menu available, assign a TPopupMenu component to the **PopupMenu** property of your form or control. The items on the pop-up menu bar and its dropdown menus are specified with the **Items** property of TPopupMenu. The properties, methods, and events for each pop-up menu item are stored in the **Items** array. Of course, the **Items** property also accesses a particular command on the menu.

Our designer example

Let's start the Menu Designer and discuss its interface. Begin by selecting New Application from the C++Builder File menu and dropping a main menu component onto the form. Invoke the Designer either by double-clicking on the component or by selecting Menu Designer... from its pop-up menu. Our example employs a main menu; however, you can substitute a pop-up menu. The Designer starts by

displaying a blank menu item and adds a TMenuItem component in the Object Inspector. Name the item by setting its **Caption** property to *&File* and press [Enter]. Click anywhere on the Menu Designer. Now the Designer is displaying the File item and is waiting for you to add the first sub-item. Your menu designer should look like Figure A.

Figure A: Menu Designer is ready to add the first item in your submenu.

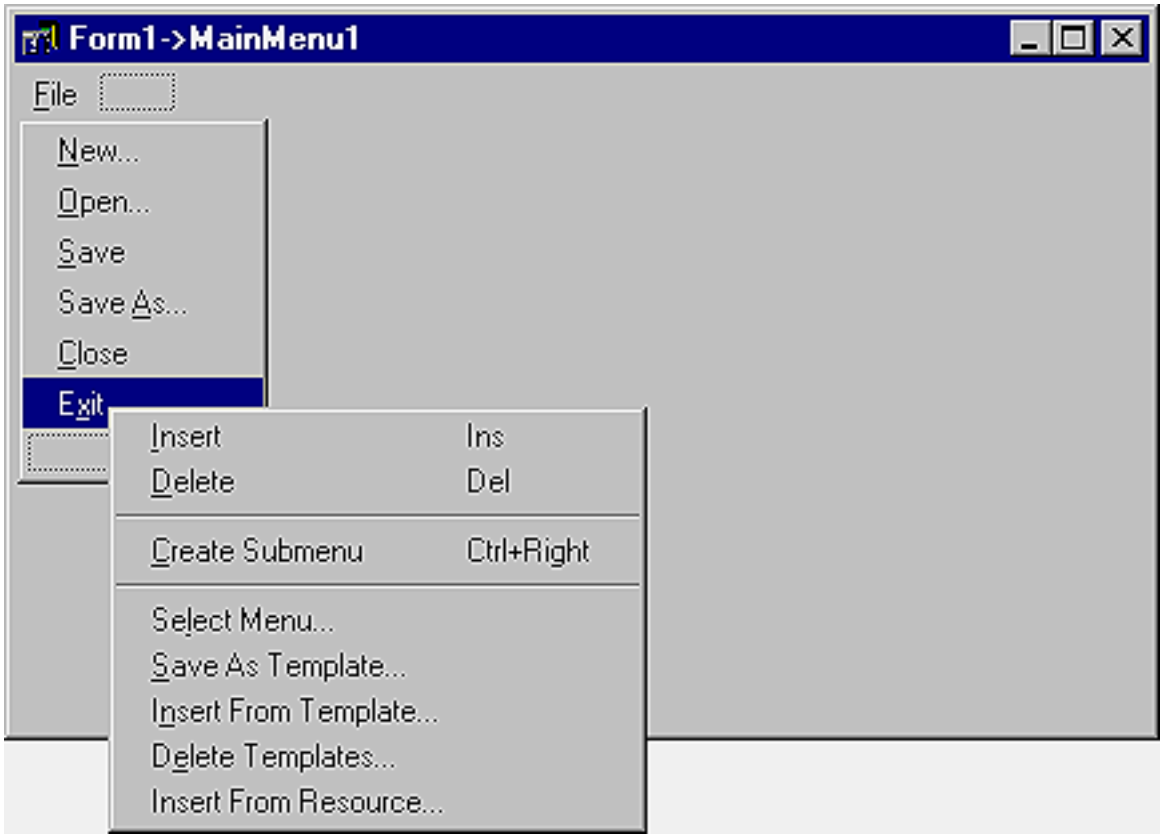


To add the next sub-item, simply enter *&New...* into its **Caption**. Following each sub-item you add, Menu Designer will add a new item. For this example, add these remaining sub-items: *&Open...*, *&Save*, *Save &As...*, *&Close*, and *E&xit*. Your completed design is a typical Windows File menu.

Inserting and deleting menu items

How often do you get your design perfect on the first pass? Right. That's why Menu Designer provides the Speed Menu. This pop-up menu makes customizing your menus a snap. While you could use this technique to insert new menu choices, we're going to use it to add a separator bar to our menu. Separator bars help visually organize your menus by grouping similar commands together. To start, click on the Exit item to highlight it. Now, right-click on it to open the Speed Menu, and you should see something like Figure B.

Figure B: You'll use the Speed Menu to customize your menu design.



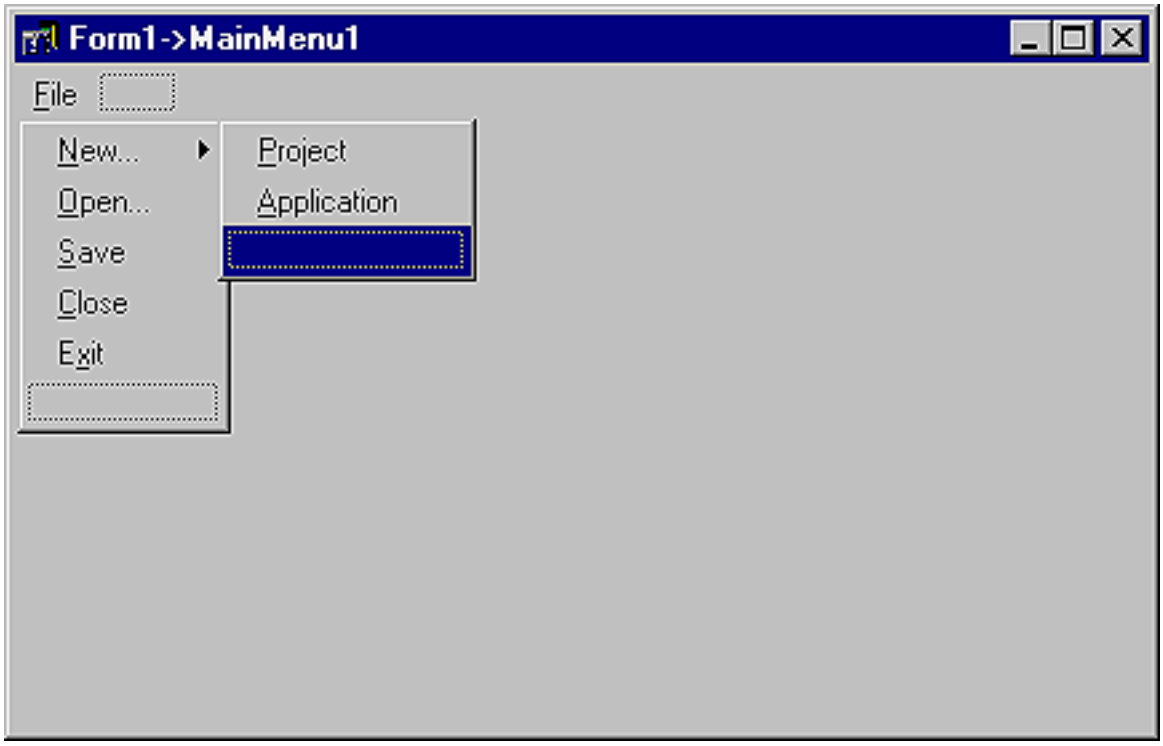
Select Insert from the Speed Menu to add a blank item to your design. To make it a separator bar, place a single dash in its **Caption** property. Note: With only one exception, inserted items are always added above the currently highlighted menu choice. In the case of menu choices on the menu bar, i.e., File, inserted items appear to the left of the currently highlighted menu choice.

The Speed Menu makes removing menu items just as easy as adding them. Highlight the Save As... item and invoke the Speed Menu. Select the Delete option and remove Save As... from the menu.

You can also use the Speed Menu to insert submenus. Submenus are choices that belong to a single menu item. An example is C++Builder's File|Reopen menu. To create your own submenu, highlight and right-click the New... menu item. Select Create Submenu from the Speed Menu. You'll notice two changes to New....

The first change is the addition of a submenu marker to the right of the New... menu item. The other change is the blank menu item attached to New.... Change its **Caption** property to *&Project*. Finally, add a second item, *&Application*, to the submenu. Your design should resemble Figure C.

Figure C: C++Builder automatically adds the submenu marker to your menu design.

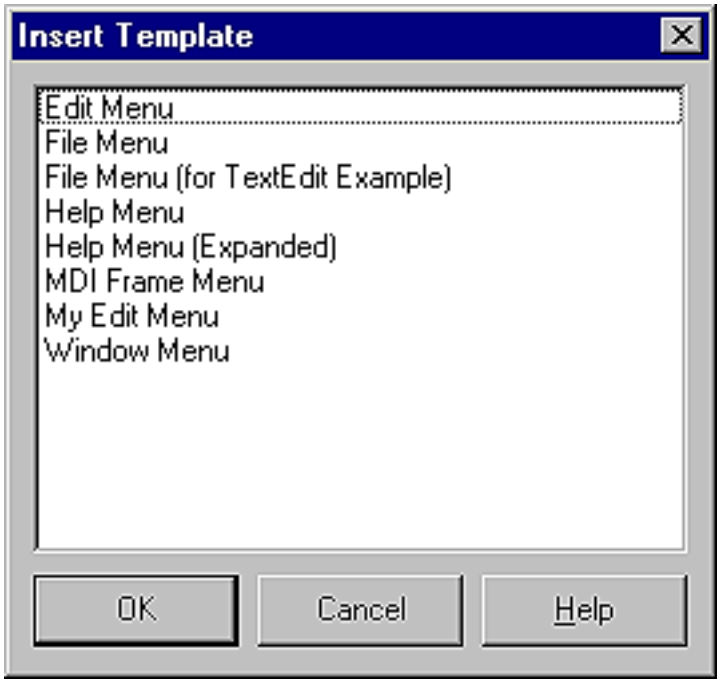


Template advantages

Menu templates are yet another feature of Menu Designer. C++Builder comes with several templates, and you can create your own as well. Templates create a consistent look and feel for your applications, and you can use them as the foundation of your custom menus. Using a menu template isn't much different from inserting a menu item. In the Designer, click on the empty menu item to the right of File. (The item appears as the rectangular outline shown in Figure C.) Next, right-click the item to invoke the Speed Menu and select the Insert From Template option.

Now you should see the Insert Template dialog box. It presents a variety of standard menus, as shown in Figure D. Go ahead and select the Edit Menu template to add the Edit menu to your menu structure. Once a template is added to your design, it's as easy to customize as any menu you build yourself.

Figure D: C++Builder provides a repository of default and custom menu templates.



If the predefined menus aren't suitable, you can create and save your own. In your Edit menu, highlight Links, then right-click and select Delete from the Speed Menu. For more convenience, you don't even have to call the Speed Menu. Select the separator bar above the Object item and Object itself and press the [Delete] key.

Let's save your custom Edit menu as a template. Highlight the Edit item, then invoke the Speed Menu. Select Save As Template... to open the Save Template dialog box. Enter a brief description of your menu (*My Edit Menu)* in the Template Description edit box. Click OK to save your template and exit the dialog.

C++Builder displays the description only in the Save Template, Insert Template (Figure D), and Delete Template dialog boxes. It's not associated with the **Name** or **Caption** property for the menu. When you save a menu as a template, C++Builder doesn't save its Name, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, C++Builder then generates new names for it and all its items.

You might save many templates over the course of several projects, then find yourself using only a few with the best designs. You can remove the ones you don't want by using the Delete Templates item of the Speed Menu. In the Delete Templates dialog box, highlight the templates you don't want and click OK.

C++Builder also imports menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can add these menus directly to your C++Builder project with the Speed Menu. In the Menu Designer, highlight the blank menu item where you want the imported menu to appear. Right-click to invoke the Speed Menu and select the Insert From Resource option. When the Insert Menu From Resource dialog box appears, select the resource file you want to import. Use imported menus to take advantage of your previous work.

Attaching code to menus

After you've added a menu to your form, you'll want to add the code that responds to user selections. While attaching code doesn't require the Menu Designer, it's appropriate to mention the technique in this article. You wouldn't normally create a menu without including processing logic. Begin by dropping an Open Dialog control on your form. Now select Open from the File menu on *your form's* menu bar. The editor window will open. Each menu item has only one event handler, **OnClick**. Add the following highlighted line to the event handler:

```
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  OpenDialog1->Execute();
}
```

Let's test your menu design and code. Press [F9] to compile and run the example. Clicking the File|Open choice invokes the Open dialog box. Of course, nothing happens when you click any of the other menu items.

Please note that C++Builder doesn't save your event handlers when you save a menu as a template. It wouldn't be appropriate because your menu item may need different logic in other applications. What you can do is associate menu items in the template with existing event handlers in the form.

Handy features

Let's finish by reviewing one last feature of the Menu Designer: managing multiple menus. Let's say your form has a main menu and two pop-up menus. By clicking on any one of these components and invoking the Speed Menu, you can choose the Select Menu option. The Select Menu dialog box will open and list the menus on your form. You'll appreciate this feature when you're managing several menus on the same form.

Conclusion

Menus are part of almost every commercial software package, and your users take them for

granted. That doesn't mean you should take them for granted, too. Use C++Builder's Menu Designer to save time and create the menus your users expect.

May 1998

Using packages in C++Builder 3

by Kent Reisdorph

As we mentioned in the April article "What's New in C++Builder 3," C++Builder 3 includes support for *packages*, which contain code for C++Builder components. When you get right down to it, a package is just a DLL with the extension BPL. C++Builder uses two types of packages: runtime packages and design packages. Let's take a moment to discuss how components worked in C++Builder 1. Then we'll contrast that with packages and examine how both types of package fit into C++Builder 3.

Components in C++Builder 1

In C++Builder 1, when you installed components to the component palette, C++Builder added the code for the components to the component library DLL, CMPLIB32.CCL. This file loads when C++Builder 1 starts. When you place a component on a form, C++Builder 1 retrieves from CMPLIB32.CLL the code needed to display the component on the form and any property editors for the component. This process allows the component to function at design time. When you build the application, C++Builder 1 extracts the code for the component from the component's object files (OBJ or DCU) and links the code to the application. The application now has everything needed for the component to function at runtime. With that bit of history behind us, let's move on to how C++Builder 3 uses packages. We'll begin with runtime packages.

Runtime packages

Runtime packages contain all the code for a component or group of components. For example, the code for the basic VCL components is contained in a package called VCL30.BPL. When you create an application, you can choose whether to use runtime packages. If you build your application without using runtime packages, it's created essentially the same as it would have been in C++Builder 1. If you use runtime packages, however, none of the code for the components is contained in your application's EXE file. Instead, the code is pulled from the runtime packages at runtime as needed. This is the traditional EXE/DLL relationship.

To enable runtime packages, open the Project Options dialog box and select the Packages page. Select the Use Runtime Packages check box, and C++Builder will build your application so that it uses runtime packages.

Note: A place on the path

Runtime packages called by design packages must be located somewhere on your system's path. The recommended location for these runtime packages is the WINDOWS\SYSTEM directory for Windows 95 or WINDOWS\SYSTEM32 for Windows NT. If you start C++Builder and it can't find a runtime package referenced by a design package, you'll get an error message saying something to the effect of *Could not load SOME.BPL. A required package was not found. Do you want to load this package the next time C++Builder starts?*

Using runtime packages has advantages and disadvantages. One advantage is that all your applications and DLLs use a common code base (the packages). This commonality eliminates code duplication in every EXE or DLL. As a result, EXEs and DLLs that use runtime packages are smaller. However, this isn't always a good thing, as you'll see next.

The disadvantage to using runtime packages is that you have to ship them with your application. The runtime packages are relatively large, so you must consider the size of your EXE with and without packages to determine whether you're saving anything. For example, let's say your application built without runtime packages is 500KB. The size of the application using runtime packages might be as little as 75KB--but VCL30.BPL itself is 1.4MB, so using runtime packages wouldn't make much sense in that case. However, if you have an application and DLLs that total 10MB, you can probably benefit from runtime packages.

Another disadvantage of using runtime packages is the extra work involved in deploying your application. Let's look at deployment issues in more detail.

Deploying an application using packages

You must know what packages to ship with your application to ensure that you're installing all the right packages and that you're installing them to the correct location on your users' machines. C++Builder comes with several runtime packages. If you choose to use runtime packages, you'll always need to ship VCL30.BPL. If your application is a database application, then you also need VCLDB30.BPL; if you're using any of the Quick Report components, you need QRPT30.BPL; and so on. Besides the packages that come with C++Builder, you may also be using third-party components. If so, you must ship any packages those components require, as well. Use care when you deploy an application that uses runtime packages. The C++Builder license agreement states that you can't alter the VCL packages, and this stipulation helps avoid the problem of mismatched packages (packages having the same name but containing different code). You should use a proven installation program when you deploy applications that use runtime packages. If a package you're installing already exists on a user's system, the installation program will check the version numbers to ensure that the version being installed isn't older than the existing version.

Tip: Which packages do you need?

You can find out which packages your application requires by running the TDUMP utility on the EXE

file. The output from TDUMP will show imports from any packages your application requires. Simply scan the output from TDUMP for imports from package files (files with a BPL extension)--you'll have to ship any package files you find.

Dynamic RTL

Hand-in-hand with runtime packages is the option to use the dynamic version of the C++ runtime library (RTL). You can enable this option by selecting the Use Dynamic RTL check box on the Linker page of the Project Options dialog box. When this option is enabled, the C++ library code isn't linked to your application, but instead is pulled from CP3240MT.DLL. Like the VCL packages, this DLL is fairly large (1.2MB), so weigh the advantages of using the dynamic version of the RTL carefully. Generally speaking, if you choose to use runtime packages you should also choose to use the dynamic RTL. It doesn't make much sense to use one without the other.

Design packages

The C++Builder IDE uses design packages for components at design time. These packages generally contain the resources required to show the component's icon on the component palette, along with the code and forms for any property editors the component implements. You can use a single package as both a runtime and a design package, but proper implementation dictates that no unnecessary code is contained in the runtime package. For example, there's no reason to include a lot of code and resources (forms) for component editors and property editors in the runtime package, because that code is needed only at design time. If you have a simple component with no property editors, then you can probably get by with a single package that acts as both a design package and a runtime package. A good component library, though, will probably have both design and runtime packages to minimize the size of the runtime package.

You must use a design package to install components on the component palette. Almost all commercial component vendors provide design packages with their component libraries. Sometimes, however, you'll run across a component that doesn't contain a design package. You can still install these components, because C++Builder contains a default design package for any components that don't have their own. Remember, design packages are only for the C++Builder IDE's use--don't ship them with your applications.

Installing design packages

Installing a design package is fairly simple. Just choose Component | Install Packages from

the C++Builder main menu to open the Project Options dialog box's Packages page. When you click the Add button to add a package, the Add Design Package dialog opens. Locate the package file (BPL) you wish to add and click the Open button. The package will be listed in the Design Packages list box along with the other installed packages. When you click OK, a message box will open, telling you which new components have been added to the component palette. Click OK again, and the components in the design package will be displayed on the component palette. Notice that there's no rebuilding of the component palette, as there was in C++Builder 1. The C++Builder 3 IDE simply has to load the design packages that are already built (remember, they're really just DLLs) and update the component palette tabs.

You can use most packages created for Delphi 3 in C++Builder, but you'll probably need to rebuild them using the DCC32.EXE that comes with C++Builder. If you have the package source files, you can build the packages from the command line. If you don't have the package source files, then you should contact the component vendor and ask for a package compatible with C++Builder.

You can remove a design package from the component palette in one of two ways. One way is to remove the component completely. To do so, select the component in the Design Packages list box (on the Packages page of the Project Options dialog box) and click the Remove button. The package will be removed from the list and from the component palette. The other way to remove a design package is to leave it in the list of installed packages but clear the check box next to its name. Doing so temporarily removecs the package from the component palette but leaves the component in the list of installed components in case you want to add it again later.

Conclusion

Packages, although a bit confusing at first, are a nice addition to C++Builder. Design packages greatly speed up the process of adding components to the component palette. Runtime packages give you the flexibility to link your application statically (without packages) or dynamically (using runtime packages). Ultimately, you'll create packages for components you create yourself--but that's a discussion for another day.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

May 1998

Owner-drawn list boxes

by Matt Telles

Every time I work with a Windows list box control, the same thing happens--I want more! Most recently, I wanted to be able to change the color in which the list box displayed items. It seemed like a good idea to have the more interesting (and appropriate) items highlighted in red, while the rest of the items were black. The problem with this desire, of course, is that the list box simply wasn't made to handle different colors in its display. There's a solution to this dilemma, though, and its name is *owner-drawn* list boxes.

What does "owner-drawn" mean?

Normally, when you work with list boxes, you add strings to the list box and Windows does the rest. Each item is displayed by the code that makes up the list box control within Windows. When you use an owner-drawn list box, however, all the display code falls within your control. Owner-drawn list boxes allow you to display each item any way you want. This can mean drawing check boxes for the items (as the Delphi 3.0 TChecklistBox component does) or changing the color for each item, as I wanted to do. Let's work through an example by creating a TColorListBox component.

Creating the color list box component

The first step in building a list box whose colors you can change is to create a component that encapsulates the problem. To do this, you create a component based on the TCustomListBox. If you've never created a component from scratch, don't fret, it's easy! (For an overview of the process, see the three-part "Building Components" article series in the September, October, and November issues of *C++Builder Developer's Journal*.) First, select Component | New from the main IDE menu. You'll see a dialog box that asks for the name of your new component; a combo box lets you select the base class to extend. Enter *TColorListBox* as the new component's name, then select TCustomListBox for the base class and click OK. The wizard will do the rest, generating the basic files needed for creating and registering the component in the C++Builder system.

Making it owner drawn

Once you've created the new component, the next step is to make Windows let you draw each element of the component. If you were adding a list box to a form, you could do this by selecting the **Style** property and changing its value to Owner Draw Variable. For this component, though, you want the owner-drawn style to be automatic. You must be able to modify the style of the control before it's displayed onscreen. There are several places you could do this, but it's easiest in the class constructor. Open the component

unit in the editor, and you'll see the constructor as one of the first class methods. Add the following code to the constructor:

```
__fastcall TColorListBox::TColorListBox(TComponent* Owner)
  : TCustomListBox(Owner)
{
    Style = lbOwnerDrawVariable;
}
```

s This code simply tells the underlying list box component that it will be of the owner-draw variable style. The owner-draw part indicates to the component that you--the programmer--will handle the drawing part of the display. The variable part indicates that you'd also like control over the height of each item in the list. This step isn't strictly necessary for this component, but you might as well learn about it!

Handling the item size

When the underlying Windows list box is displayed, it first finds out how much space to reserve for each item. It does so by sending the list box a message that requests the size of the item currently being drawn. For the TCustomListBox class, the MeasureItem method handles this message. Since you made your list box variable size, this method will be called for each item in the list. Here's one implementation for the MeasureItem method:

```
void __fastcall TColorListBox::MeasureItem(
                int Index, int &Height)
{
    Height = 12;
}
```

In this case, you simply want to make each item be a certain size: 12 pixels high. Note that you pass the Height parameter by *reference*, allowing it to be modified within the method and returned to the calling function.

Note: Changing the font

If you want to change the font for each item in the list box as well as the item's color, you can determine the height of each item by checking the font that you assigned to it.

Drawing the item

Once the item has been measured and fitted for its little square of the list box, the next step is to actually draw that item into its drawing area. You accomplished this task using the DrawItem method of the TCustomListBox class. Add a new method to the class by modifying the source and header file for the new TColorListBox class. The complete code for the new method is as follows:

```
void __fastcall TColorListBox::DrawItem(
                int Index, const Windows::TRect
                &Rect, TOwnerDrawState State)
{
   Canvas->Font->Color = GetColor(Index);
   Canvas->TextRect(Rect, Rect.Left, Rect.Top,
                    Items->Strings[Index].
                    c_str());

}
```

There isn't a lot to the method. You get the color for your index by passing the index of the item (given to you by the list box itself) to the GetColor method. This method, which you'll write very shortly, simply returns the color assigned to a given list box item.

Once you have the color, you just use the TextRect method of the **Canvas** property to display the string in the area given to you for this item. The area is defined by the height of the item (returned by MeasureItem) and the width of the list box. Given this information, you can easily display the string in your desired color.

Note that you use the text stored in the list box's **Strings** property. Doing so permits the list box to work exactly like a normal list box, but with added functionality!

Adding the Color property

The last step to make this a fully functioning component is to add the **Color** property. Normally, you'd make this a published property and let the form designer enter the values for the colors at design time. In our example, you won't take that route. In order to implement the **Color** property correctly, it needs to be an array of colors that refer to the strings in the list box. Creating an array property is simple enough, as you'll see, but making an editor for that property is a non-trivial matter. For this reason, you'll make the property runtime only. (The colors for the list box are useless without strings, which provides additional motivation.)

To add the new property, you must first decide how to represent it internally. The internal representation may have nothing to do with the external representation to the programmer, but it will drive how that external representation works. In this case, since there's an

indeterminate number of entries, I decided to use the Standard Template Library (STL) vector class to represent the colors. To do so, add the following section to your header file for TColorListBox:

```
private:
   std::vector<TColor> FColors;
```

Next, you need to add the **Color** property definition to the class, using a public entry like the following:

```
__property TColor Colors[int nIndex] =
  {read=GetColor, write=SetColor};
```

Again, there's nothing surprising here. The property is an array, so it takes an index to represent the number of the item for which to set the color. You also need to add the SetColor and GetColor methods. Here's the header file entry for these methods:

```
protected:
virtual void __fastcall SetColor(
                        int Index, TColor clr );
  virtual TColor __fastcall GetColor(
                             int Index );
```

Finally, you need to implement the SetColor and GetColor methods. Listing A contains the relevant code extracts, which go in the source file (TColorListBox.cpp).

Listing A: The SetColor and Get Color methods

```
void __fastcall TColorListBox::SetColor(
                int Index, TColor clr)
{
   if ( Index < 0 )
      return;
   // See if we already have this many elements
   // in the array. If we don't do this it will
   // throw an exception when we try to set the
   // color.
   if ( Index < FColors.size() )
      FColors[Index] = clr;
   else
```

```
   {
      // Hold onto the current size
      int nCurSize = FColors.size();
      // Resize the array
      FColors.resize( Index+1 );
      // Initialize the remainder of the
      // entries to black (our default)
      for ( int i=nCurSize; i<=Index; ++i )
          FColors[i] = clBlack;
// Finally, set the one they want
      FColors[Index] = clr;
   }
}

TColor __fastcall TColorListBox::GetColor(int Index)
{
   if ( Index >= 0 && Index < FColors.size() )
      return FColors[Index];
   return clBlack;
}
```

The code is pretty straightforward. For the set case, you simply check to see if that many entries have already been created. If so, the entry the user wants to set replaces the one already in the array. If the entry doesn't already exist, the array is resized to the proper size and all elements are initialized to the default black color.

For the get case, which we also use in the drawing code, you once again consult the array to see whether the element is already there. If so, you return the value at that location. If the array isn't yet that size, the user hasn't set a color for this item, and you return the default black color. Note that in all cases, you check for invalid entries, such as an index less than zero. Doing so is just good programming practice that you should use in your own coding.

Conclusion

That's all there is to it! If you complete this code exercise and install the component, you'll have a complete color list box that you can use in your own applications. Good luck and happy coding!

May 1998

MDI 101

by Gerry Myers

The structure of a Windows application falls into one of three categories: SDI (Single Document Interface), MDI (Multiple Document Interface), or Dialog-as-main-window interface. The easiest way to understand SDI and MDI is to think about a word-processing program. An SDI application is something like Notepad or Write. In these programs, you can open and work in only one text file (document) at a time. Before you can open another document, you must close the current document--hence the name *Single* Document Interface. SDI applications are fairly straightforward to write; the menu is generally static, and your data comes and goes from only one data file.

An MDI word-processor would operate something like MS Word, AmiPro or WordPerfect. These programs let the user have several documents open at a time. Because users can work on several files (documents) simultaneously in MDI applications, they find such programs more convenient. Cutting and pasting between two documents is much easier in MDI since both files can be open. (You can cut and paste in SDI applications, but there's no drag-and-drop option.)

An MDI application is somewhat harder to write than other kinds of programs, since the menu can change depending on which open document has focus. In addition, you must take care not to put data in the wrong open file.

Both SDI and MDI mention a *document*. We generally think of a document as a disk file, but it can really be anything that can give and get data. A communications port can be a document, as can a PC sound system. In our word-processing analogy, a document is indeed a disk file containing textual information.

The use of the term *document* to differentiate the two application types (SDI versus MDI) isn't completely accurate. What sets MDI apart from SDI is the fact that MDI applications have *child* windows. All the display work in an SDI application takes place right on the application's main window. However, in an MDI application, you generally draw nothing on the main window--all presentation takes place on child windows displayed within the borders of the main window (the *parent* window). Actually, in a multiple *document* interface application, you can view the data from a single open *document* (disk file) in different ways through different child windows (one document, multiple child windows). This ability to contain child windows draws the line between MDI and SDI.

I've used a word-processing analogy for discussion purposes only. This example in no way means that SDI and MDI apply only to word processing. It would be safe to argue that the initial use of the term *document* sprang from Microsoft's Word and Excel projects back during the early days of Windows 3.x, and then it stuck. Nevertheless, developers are using the SDI and MDI structures for nearly every Windows program regardless of the type of data and where the data comes from.

Just for completeness, let me mention the Dialog-as-main-window interface, which appears in applications like Windows 95's Add/Remove Programs control panel. When you start such an application, the main window that opens is a dialog box. A dialog box generally doesn't have a menu, but is populated with buttons, list boxes, edit boxes, check boxes, and so on. Such programs are fairly straightforward to write.

May 1998

MDI made easy

by Gerry Myers

While reading through several C++Builder online forums, I came across repeated questions concerning MDI (Windows Multiple Document Interface). I saw comments like, "How do I find the parent of my MDI child?" and "From whom do I derive my MDI child?" and "Why does my program crash when I close my MDI child?". I must admit that coming from Borland C++ (with OWL), I was also a little confused at first about C++Builder's handling of MDI. However, after some trial, error, and research, I realized that I was confused because it was so easy--too easy, apparently. C++Builder never ceases to amaze me with the simple way it handles form interaction. When you start working with MDI in C++Builder, forget everything you had to learn about the Document Manager, the document, and the view classes. In C++Builder land, all forms are created equal. In this article, we'll discuss how C++Builder creates MDI parents and children and passes information between them. If you're unfamiliar with the MDI concept, see "MDI 101" for an introduction.)

C++Builder MDI

If you've used C++Builder for more than five minutes, you're familiar with the TForm class. TForm contains properties for setting a window's color, border style, caption, and so on. In fact, you'll set almost any characteristic a window can have through the TForm base class. This symmetry isn't lost when it comes to C++Builder MDI windows.

Let's take a couple of minutes and have C++Builder create a new MDI application. To do so, choose File | New from the main menu, then select the Projects tab and double-click on the MDI Application icon. Now, take a look at the header files shown in Listings A and B for the main form (the parent) and the child form (TMDIChild). You'll notice that both are derived from TForm.

Listing A: TMainForm header, main.h

```
class TMainForm : public TForm
{
__published:
   TMainMenu *MainMenu1;
   TMenuItem *File1;
   TMenuItem *FileNewItem;
   TMenuItem *FileOpenItem;
   TMenuItem *FileCloseItem;
   TMenuItem *Window1;
```

```
   TMenuItem *Help1;
   TMenuItem *N1;
   TMenuItem *FileExitItem;
   TMenuItem *WindowCascadeItem;
   TMenuItem *WindowTileItem;
   TMenuItem *WindowArrangeItem;
   TMenuItem *HelpAboutItem;
   TOpenDialog *OpenDialog;
   TMenuItem *FileSaveItem;
   TMenuItem *FileSaveAsItem;
   TMenuItem *Edit1;
   TMenuItem *CutItem;
   TMenuItem *CopyItem;
   TMenuItem *PasteItem;
   TMenuItem *WindowMinimizeItem;
   TPanel *SpeedPanel;
   TSpeedButton *OpenBtn;
   TSpeedButton *SaveBtn;
   TSpeedButton *CutBtn;
   TSpeedButton *CopyBtn;
   TSpeedButton *PasteBtn;
   TSpeedButton *ExitBtn;
   TStatusBar *StatusBar;
void __fastcall FormCreate(TObject *Sender);
   void __fastcall FileNewItemClick(TObject *Sender);
   void __fastcall WindowCascadeItemClick(
                   TObject *Sender);
   void __fastcall UpdateMenuItems(TObject *Sender);
   void __fastcall WindowTileItemClick(
                   TObject *Sender);
   void __fastcall WindowArrangeItemClick(
                   TObject *Sender);
   void __fastcall FileCloseItemClick(
                   TObject *Sender);
   void __fastcall FileOpenItemClick(
                   TObject *Sender);
   void __fastcall FileExitItemClick(
                   TObject *Sender);
   void __fastcall FileSaveItemClick(
                   TObject *Sender);
   void __fastcall FileSaveAsItemClick(
                   TObject *Sender);
   void __fastcall CutItemClick(TObject *Sender);
```

```
   void __fastcall CopyItemClick(TObject *Sender);
   void __fastcall PasteItemClick(TObject *Sender);
   void __fastcall WindowMinimizeItemClick(
                   TObject *Sender);
   void __fastcall FormDestroy(TObject *Sender);
private:
   void __fastcall CreateMDIChild(const String Name);
   void __fastcall ShowHint(TObject *Sender);
public:
   virtual __fastcall TMainForm(TComponent *Owner);
};

extern TMainForm *MainForm;
```

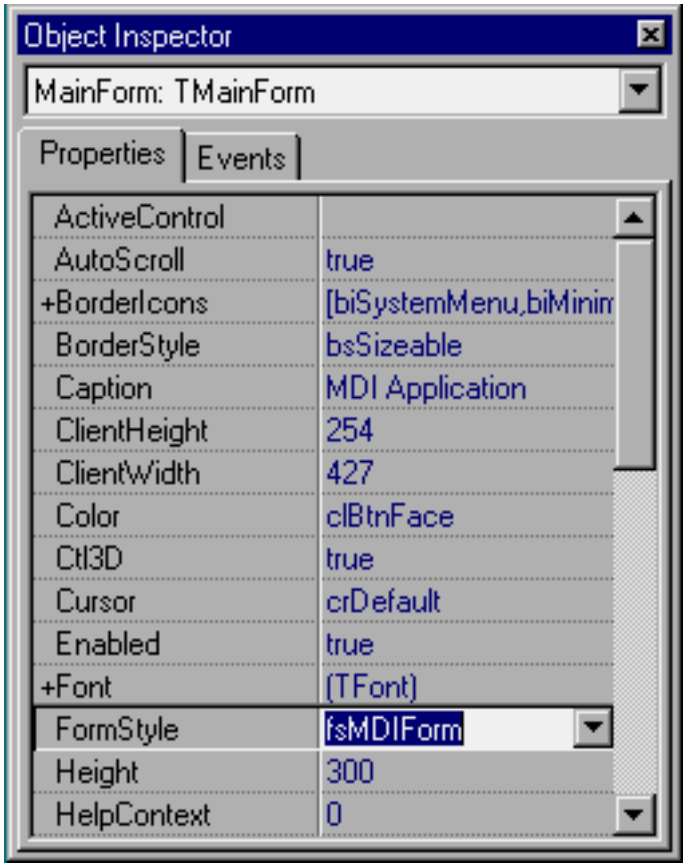
Listing B: TMDIChild header, childwin.h

```
class TMDIChild : public TForm
{
__published:
   void __fastcall FormClose(TObject *Sender,
                   TCloseAction &Action);
private:
public:
   virtual __fastcall TMDIChild(TComponent *Owner);
};
```

You should also notice something strange: The new TMDIChild class is almost empty. I would have suspected that TMDIChild added special properties and methods for handling the required MDI tasks above and beyond the normal TForm capabilities. But no! As it turns out, all the MDI capabilities are built into the TForm class--non-MDI programs simply don't use them. It appears that some special child-related methods are added in the TMainForm (parent form) header, but they turn out to be simple menu-event handlers.

If all the MDI capabilities pre-exist in the TForm class, what triggers TForm to use them? This questions brings us back to the wonder of C++Builder. Click on your main form and look in the Object Inspector for the **FormStyle** property, as shown in Figure A.

Figure A: The parent window's FormStyle property is set to fsMDIForm.



It's set to fsMDIForm. Now click on the child form--its **FormStyle** property is set to fsMDIChild. That's all that's needed to turn a form into an MDI parent or an MDI child. Pretty neat!

C++Builder checks the **FormStyle** property during the form object's construction. It has only four possible values: fsMDIForm, fsMDIChild, fsNormal, and fsStayOnTop. For most standard forms, the value is fsNormal, which bypasses all MDI processes. When **FormStyle** is set to fsMDIForm or fsMDIChild, C++Builder establishes the structures that link the parent to the child.

MDI structures

Let's begin our exploration of MDI-related properties by examining **ActiveMDIChild**. This property (as well as the others we'll discuss) applies only to the parent form. **ActiveMDIChild**, which is of type TMDIChild*, points to the MDI child contained within the parent that holds focus. You can read it to get the active child form or set it to change the focus to a different child. The TForm class takes care of updating **ActiveMDIChild** as the user clicks between different child forms. The next property to be concerned with is **MDIChildren**, which is a list of all the child forms contained in the parent. The TForm class maintains this list for you, so there's nothing you need to do except use it. You probably shouldn't set or modify this list, since you can inadvertently lose child forms if you make a mistake. TForm maintains the list in the reverse order in which the child forms were created--element 0 points to the newest

child form.

You can reference child forms directly from the **MDIChildren** list using bracket notation. For example, to get a pointer to the third-*youngest* child form, you'd write

```
TMDIChild* mdiChild = MDIChildren[ 2 ];
```

The last property we'll discuss is **MDIChildCount**. It's of integer type and simply lets you know how many elements are in the **MDIChildren** list (how many child windows exist). I recommend that you only *get* this value. If you *set* it to an improper value, your program will either crash (if the value is larger than the number of actual child forms) or some child forms will be inaccessible (if the value is smaller than the actual number of child forms). It's safest to let the **TForm** class modify this value.

We've listed three parent form properties for accessing children, but how does the child form access the parent? Doing so turns out to be just as easy. You'll notice at the bottom of the main form's header file the statement

```
extern TMainForm *MainForm;
```

This pointer is available to any class that includes the main form's header (which most forms do). Since the child class does include the main form's header, you can call the parent's methods and access the parent's properties simply through pointer reference. For example, if you want a child to grab focus, you can place the following statement somewhere in the child:

```
MainForm->ActiveMDIChild = this;
```

Doing so will make the parent recognize the child as the active child.

Caution

As you can see, MDI applications in C++Builder are a snap. All the MDI capabilities are built right into the TForm base class, and you must learn only a handful of properties to manipulate the parent-child connection. With all this ease of use, I need to make one point clear. Remember how you normally create a component at runtime? For a button, you write code something like the following:

```
TButton* btn;
btn = new TButton( this );
btn->Parent = this;
```

When you create a component, it needs to know its owner (this in the constructor) as well as its parent. The owner is the object responsible for deleting the component when the owner is destroyed. The parent is the object responsible for displaying the component (clipping, scrolling, moving, and so on). With most components, it's fine to set both the owner and the parent to the form onto which you're dropping them (this). However, this isn't the case with MDI children. When you create an MDI child, you should set *only* the owner (in the constructor statement)--don't worry about the parent. If you set the parent to this, everything will appear to work fine until you close the child window. At that point, your program will crash.

For example, the following code creates the child window with the application as the owner:

```
TMDIChild* child;
child = new TMDIChild( Application );
child->Parent = this; // runtime error
```

That's fine. The error comes by setting the parent in the next line.

You'd expect to be required to set the **Parent** property of the child to point to the parent window (TMainForm), but this isn't the case. The TForm class takes care of the parent property the way it wants to, and that's just the way it is.

Conclusion

MDI applications were a little intimidating in the days of straight Windows programming and continued to be so with products like Borland's OWL and Microsoft's MFC. You had to be very familiar with the linkage between the parent and child as well as the document manager class. C++Builder has again made our programming lives easier: Now we have to make use of only three TForm properties to master MDI programming.

May 1998

Capturing a form to disk

by Kent Reisdorph

Did you ever want to capture a portion of your application as a bitmap file on disk? If so, you'll be glad to learn that doing so isn't difficult. Getting it right, though, takes a little work. In this article, we'll show you how to capture a VCL form to a Windows bitmap (BMP) file on disk. We'll also exlain how to get the palette for your form so you are sure the image will be saved properly.

The big picture

When you come right down to it, the whole Windows desktop display is one big bitmap. Sure, it looks like you have lots of windows onscreen, but it's just your mind (and Microsoft) playing tricks on you. You *perceive* a two-dimensional (or even three-dimensional) plane with multiple windows floating around, but it's no more than visual trickery. Capturing a form to disk should just be a matter of copying bitmap bits from the appropriate part of the desktop display to memory, then to a disk file. Sound complicated? It would be if VCL didn't do most of the work for you. But thanks to VCL, most of the operation is fairly easy. The required steps are as follows:

- Create a TCanvas object to represent the desktop device context.
- Create a TBitmap object to hold the bitmap in memory.
- Copy the bits representing the form onscreen from the desktop canvas to the bitmap in memory.
- Save the bitmap in memory to disk

Most of this process is straightforward; we'll show you exactly how to perform each step. However, one item doesn't appear on our list: grabbing the form's palette. We'll leave that step for a little later.

A canvas for the desktop

VCL doesn't provide a built-in canvas for drawing directly on the desktop. After all, that's Windows' job--Windows provides a way of getting to the desktop device context. You can loosely think of a Windows device context (commonly know as a *DC*) as a drawing surface. Working with DCs at the API level can be messy. If you don't jump through all the hoops--and in the right order--your application will leak GDI resources. Fortunately, VCL provides the TCanvas class to make working with DCs easier. The trick, then, is to create a TCanvas object that allows access to the desktop DC. Hold that thought for a moment, and we'll get back to it in just a bit.

As you know, some VCL components come with a **Canvas** property and some don't. A PaintBox

component has a **Canvas** property, naturally, so you can draw on the paint box. A Memo component, on the other hand, has no **Canvas** property. Windows is responsible for drawing the contents of a Memo component, and most mere mortals shouldn't dabble in that area (nor would you want to, in most cases). The good news is that you can create a Canvas for *any* window, provided you know the window handle.

For example, suppose you were to tempt fate and draw on a Memo component. You could create a TCanvas object for that purpose with this code:

```
TCanvas* canvas = new TCanvas;
canvas->Handle = GetDC(Memo->Handle);
// do some stuff
delete canvas;
```

When you call delete on the TCanvas object, VCL performs all the nasty cleanup tasks required to properly dispose of a device context. Clean and easy (that's what C++Builder is all about). But the desktop isn't a window, per se. So how do you create a device context for the desktop? The Windows API provides a way. When you call GetDC() with a window handle of 0, Windows happily creates a device context for the desktop. Combining that with the previous code snippet, you can create a canvas for the desktop as follows:

```
TCanvas* canvas = new TCanvas;
canvas->Handle = GetDC(0);
```

Now you can draw on the desktop to your heart's content--but you *shouldn't*, of course. You should usually consider a desktop canvas a read-only object. That's how we'll use the desktop canvas in this case.

How's your memory (bitmap)?

In order to create a working bitmap in memory at the API level, you need to create a memory device context and a bitmap. You then select the bitmap into the memory device context. Here again, VCL does a great deal of behind-the-scenes work for you. The TBitmap class contains both a bitmap and a memory DC. Creating a TBitmap object is simple:

```
Graphics::TBitmap*
    bitmap = new Graphics::TBitmap;
```

As you can see, you must specify the Graphics namespace so the compiler knows you're creating a VCL TBitmap object (as opposed to the "other" TBitmap, which is a typedef for a Windows HBITMAP handle). That's not the whole story, though, because you also need to set the height and width of the bitmap prior to writing data to it. Since you'll be copying the

form's image to the bitmap, you can use the height and width of the form to size the bitmap:

```
bitmap->Width = Width;
bitmap->Height = Height;
```

You're now ready to get on with the business of copying the form's bitmap bits from the desktop canvas to the bitmap in memory.

Copy and save to disk

Copying the bits is a simple process. You need only three lines to accomplish this part of the operation:

```
TRect src = BoundsRect;
TRect dest = Rect(0, 0, Width, Height);
bitmap->Canvas->CopyRect(dest, dtCanvas, src);
```

The first line sets the rectangle called src to the size and position of the form's outer dimensions (the **BoundsRect** property provides these dimensions). You'll use this rectangle as the source of your copy operation. The second line sets the destination rectangle. In this case, the destination rectangle is the entire size of the memory bitmap.

The last line in the code snippet copies the bitmap bits from the source rectangle to the destination rectangle. The CopyRect function of TCanvas does all the work. Behind the scenes, this step translates to a call to the Windows API function BitBlt().

All that remains is to save the bitmap in memory to a disk file. Don't blink or you'll miss it:

```
bitmap->SaveToFile("form.bmp");
```

Yes, that's really all there is to it. Now you have a copy of your form captured on disk.

But wait, there's more!

If your form contains only standard Windows controls, then you can stop reading--you're finished. If, however, your form contains bitmaps composed of more than 16 colors, you'll need to do a little more work to ensure that the form is saved with the correct colors--you must delve into the mysterious world of *palettes*. Let's say you have a form with an Image component, and that the Image component contains a 256-color bitmap (for example, the HANDSHAKE.BMP file that comes with C++Builder). You can save the form to disk using the code we presented earlier. When you do, however, it will come out looking like the image

shown in Figure A.

Figure A: This saved form has a bad palette.

Even in black and white, you can see that something is drastically wrong with this picture. The reason is that the palette information for the 256-color bitmap wasn't saved when the file was saved. To make the bitmap display correctly, you have to obtain the form's palette and assign it to the memory bitmap's **Palette** property before saving the file.

In this case, VCL doesn't provide an easy way to obtain a form's palette. You'll have to get your hands dirty and use the API. Here's the code:

```
int nColors =
  GetDeviceCaps(Canvas->Handle, SIZEPALETTE);
LOGPALETTE* logPal = (LOGPALETTE*)new Byte[
  sizeof(LOGPALETTE) + (nColors - 1) *
  sizeof(PALETTEENTRY)];
logPal->palVersion  = 0x300;
logPal->palNumEntries = (Word)nColors;
GetSystemPaletteEntries(Canvas->Handle,
  0, nColors, logPal->palPalEntry);
bitmap->Palette = CreatePalette(logPal);
delete[] logPal;
```

Yikes! Yes, folks, this is the way people used to write Windows programs. In summary, the code snippet gets the number of colors for the current display driver, creates a LOGPALETTE

structure using that information, and then gets the palette for the form by calling GetSystemPaletteEntries(). This Windows API function requires a handle to a device context, so you pass the **Handle** property for the form's canvas. After that, you create a new palette with the CreatePalette() function and assign the result to the memory bitmap's **Palette** property. Now the color information will be saved when the bitmap is saved to disk.

Note that if you have a single 256-color object on a form, you can take a shortcut and assign that component's **Palette** to the **Palette** of the bitmap. For example:

```
bitmap->Palette = Image1->Picture->Palette;
```

This is simplest method of preserving palette information, but it isn't always the most reliable. The method we've outlined ensures that the entire form's palette will be preserved rather than relying on the palette of a single component. Figure B shows the result when the palette information is saved properly. As you can see, this image shows a dramatic improvement.

Figure B: When we save the form with the proper palette, it looks much better.

A full example

You can save the image of any form to disk using the method we've outlined in this article. Note, however, that the form must be fully visible onscreen for this technique to work. Listing A contains the source unit of a program that saves its main form to disk. We didn't list the header to save space, but the pertinent details are provided. As usual, you can obtain the entire project from **www.cobb.com/cpb, as part of the file may98.zip.**

Listing A: FORMCAPU.CPP

```
//---------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "FormCapU.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(
                TObject *Sender)
{
  // Create TCanvas object for desktop DC.
  TCanvas* dtCanvas = new TCanvas;
  dtCanvas->Handle = GetDC(0);

  // Create new TBitmap object and set its
  // size to the size of the form.
  Graphics::TBitmap*
    bitmap = new Graphics::TBitmap;
  bitmap->Width = Width;
  bitmap->Height = Height;

  // Create palette from form's Canvas; assign
  // that palette to bitmap's Palette property.
  int nColors =
    GetDeviceCaps(Canvas->Handle, SIZEPALETTE);
  LOGPALETTE* logPal = (LOGPALETTE*)new Byte[
     sizeof(LOGPALETTE) + (nColors - 1) *
     sizeof(PALETTEENTRY)];
  logPal->palVersion  = 0x300;
  logPal->palNumEntries = (Word)nColors;
  GetSystemPaletteEntries(Canvas->Handle,
    0, nColors, logPal->palPalEntry);
  bitmap->Palette = CreatePalette(logPal);
```

```
  delete[] logPal;

  // Copy section of screen from
  // desktop canvas to the bitmap.
  TRect src = BoundsRect;
  TRect dest = Rect(0, 0, Width, Height);
  bitmap->Canvas->CopyRect(dest, dtCanvas, src);

  // Save it to disk.
  bitmap->SaveToFile("form.bmp");

  // Clean up and go home.
  delete bitmap;
  delete dtCanvas;
}
```

April 1998

What's new in C++Builder 3

by Kent Reisdorph

C++Builder 3 is out, and it has plenty of new features to tempt you. In this article, we'll cover some of the most important new features. Many of them are borrowed from Delphi 3, but some are features that even Delphi doesn't have yet. (You've probably already noticed that the version following C++Builder 1 is C++Builder 3. Borland did this to synchronize C++Builder's version number with that of Delphi.)

Project groups

A project group is, not surprisingly, a group of C++Builder projects. When you open a project group, you have immediate access to all projects in the group. You can have more than one project open at a time, which is convenient if the projects interact. The most obvious example is the scenario in which you're creating a DLL and you have a test application that calls the DLL. With project groups, you can open both projects at once and switch between them as needed. A project group also lets you build every project in the group at once, a serious time saver if you have lots of projects that you need to build on a regular basis. You'll find many more advantages to project groups once you begin working with them.

The new Project Manager

The Project Manager is completely new in C++Builder 3. In fact, it's so new that Delphi doesn't even have it yet! The Project Manager is a tree-view control that displays the project group at the top and any projects as nodes under the project group. Within the project nodes are nodes for units, forms, or support files (RC files or LIB files, for example). You can even add your own nodes, such as text files or your units' header files. When you double-click on a form node, C++Builder displays that form in the Form Designer. If you double-click on a unit node, C++Builder loads that unit and displays it in the Code Editor.

The Project Manager has the usual features you'd expect, such as the ability to add files to projects or remove them. We can't discuss all its new capabilities here, but once you try it, you'll quickly see how great an improvement it is over C++Builder 1.

C++Builder packages

C++Builder 3 supports packages. A *package* is essentially just a DLL. There are two types of packages: design packages and runtime packages. At design time, the IDE uses design packages, which contain the code to display a component's icon on the Component Palette and to display the component on a form at

design time. The runtime package contains all the code necessary for a component to operate at runtime. Obviously, there's some code overlap in these two packages. In order to eliminate duplicate code, you usually create a design package so that it uses code from the runtime package. C++Builder can use packages created for Delphi 3, but you should recompile them using the DCC32.EXE that comes with C++Builder 3.

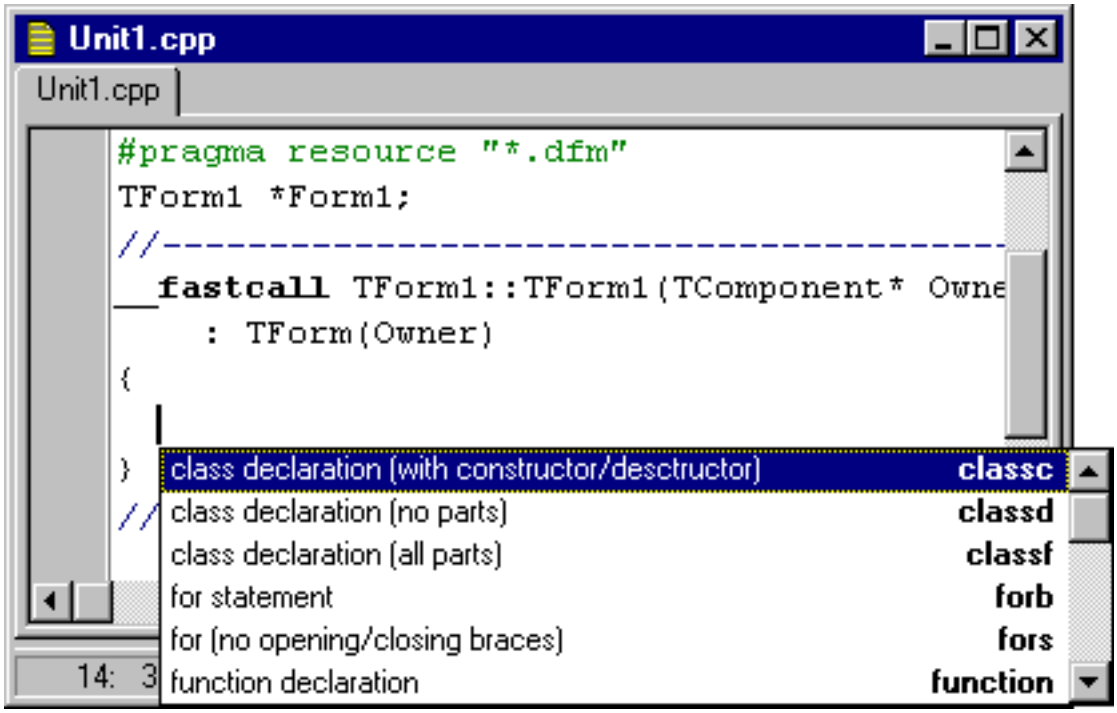
You can build programs written with C++Builder 3 to use packages or not use them. If you don't use packages (the default), then your applications will be just as they were in C++Builder 1. In other words, the code needed for your application to run is linked into your application's EXE file. If you choose to use packages, then your application will be much smaller--but you'll have to ship the runtime packages with your application (which can easily amount to 3MB or more). If you previously used Borland C++, then you'll recognize this as the old static linking versus dynamic linking scenario.

New Code Editor features

The Code Editor has some new features that are worth pointing out. I'll also describe some features that don't pertain directly to the Code Editor, but are related. The Code Editor now supports drag-and-drop text editing. To move a block of text, you select it, then place your mouse pointer over the highlighted text, click, and drag. The pointer will change to the drag shape, accompanied by a gray vertical bar. The vertical bar indicates where the text will be placed when you stop dragging. Once you have the mouse pointer where you want it, release the mouse button to move the text. To copy text, repeat these steps but hold down the [Ctrl] key when you release the mouse button. The Code Editor's drag-and-drop text editing works just like other Windows programs that implement this feature, so you should feel right at home with it.

Code templates are another nice feature of the Code Editor. Code templates let you quickly and easily insert prewritten sections of code into your programs. When you want to insert a code template, you first press [Ctrl]J. A list box containing the available code templates pops up, as shown in Figure A. Once you've selected a code template, the text for that template is inserted into your application at the cursor location. You can then fill in the gaps and add additional code as needed.

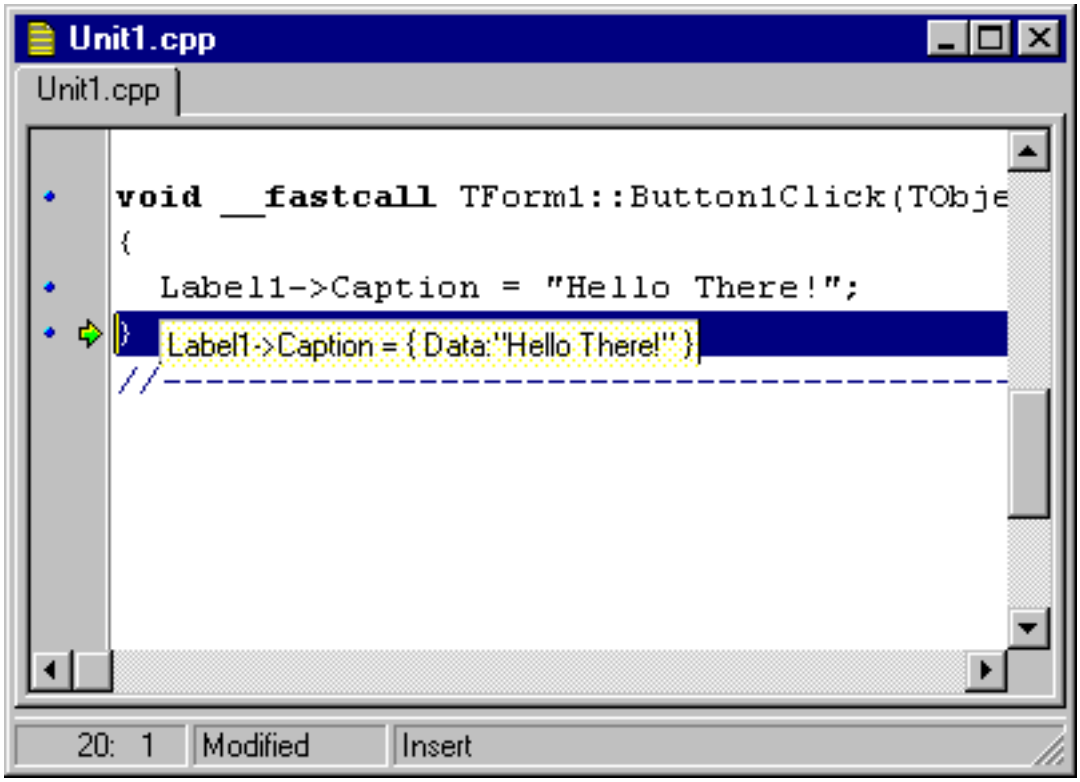
Figure A: This list box contains the available code templates.



C++Builder 3 comes with a number of code templates ready for you to use. For example, most of the typical C++ language features such as the for loop, the while loop, the switch statement, and a typical class declaration are predefined. The great thing about code templates is that you can add your own to those provided by C++Builder. You add code templates through the Code Insight page of the Environment Options dialog box or by directly editing the BCB.DCI file (found in the BIN directory).

Another new feature of the Code Editor is tool-tip expression evaluation. (This is really a feature of the debugger, but you use it when stepping through code in the Code Editor.) When your program pauses at a breakpoint, you can place your mouse pointer over a variable, and a tool tip will appear showing that variable's value. If you're a Borland C++ 5.0 user, you probably use this feature frequently. Figure B shows the tool-tip expression evaluator in action.

Figure B: Tool-tip expression evaluation is a new debugger feature.



The Code Editor now has a visible gutter along the left edge of the editor window, as you can see in Figure B. This feature makes it much easier to detect the gutter and avoid it when editing code (I always found myself inadvertently setting a breakpoint when trying to click on the first column in the Code Editor). You can change the width of the gutter or hide it altogether. The gutter contains glyphs that indicate lines for which code is generated, a breakpoint, or the current execution point.

Find In Files

Finally, GREP has been integrated into the C++Builder IDE! The Find In Files option allows you to search all open files, all files in the current project, or files in directories. The results of the search are displayed in the Code Editor's message window. If you double-click on one of the search listings in the message window, the corresponding file will open with the line containing the search text highlighted. This is a great feature and one that many people have asked for since C++Builder 1 first hit the streets.

Debugger enhancements

The debugger has several new features. One of the best is the tool-tip expression evaluator we discussed in the previous section. A similar feature lets you view local variables. When you choose Run | Inspect Local Variables from the main menu, a Debug Inspector window pops up showing all variables local to the current function.

Speaking of the Debug Inspector, it's mostly unchanged from the first version of C++Builder. However, there's one important new IDE feature that affects the Debug Inspector: The Debugging page of the

Environment Options dialog box contains an Inspectors Stay On Top check box. When this option is on, all Debug Inspector windows will stay on top of the Code Editor window. This is a very welcome feature, since most of the time you want your Debug Inspector windows on top of the Code Editor. The Watch Window also has some improvements. Most noticeably, when you add a watch, C++Builder immediately adds to the Watch Window the variable to watch, without first displaying the Watch Properties dialog box. In C++Builder 1, the Watch Properties dialog box appears first, forcing you to click OK or press [Enter]. This one-step process is a welcome improvement.

Another feature of the Watch Window is the ability to set a variable breakpoint. You can add a variable to the Watch Window and then choose Break When Changed from the Watch Window speed menu. When that variable changes value, the debugger will pause program execution as if it had hit a breakpoint.

The Event Log--another new feature of the IDE and the debugger--replaces and augments the OutDbg1.txt file from C++Builder 1. It shows status messages for the modules your program loads, when your program starts and stops, when the program hits a breakpoint, and other information. To send your own messages to the Event Log, use the Windows API function OutputDebugString(). You can save the messages in the Event Log to a file, if you wish.

As I alluded to earlier, the Environment Options dialog box has a new Debugging page. This page lets you set some new options for the debugger and the Event Log. More importantly, the page contains settings that determine how exceptions are handled under the debugger. You'll have to spend some time with the debugger to really appreciate this feature.

The debugger now lets you start a host application when debugging a DLL. This is an indispensable feature when you're writing a DLL that will be called from non-C++Builder applications. To specify a host application, you choose Run | Parameters from the main menu and type an EXE name in the Host Application field of the Run Parameters dialog box. When you click the Load button, the host application will load; you can then debug your DLL as usual. The C++Builder IDE also has a new utility called the Modules window, which you invoke by choosing View | Modules from the main menu. This window shows the modules that are currently loaded and the entry points (functions) of each module. When you double-click on an entry point, the Code Editor will take you to the source line containing that entry point (or the CPU view if no source code exists for the entry point). This is one of those options that is probably of interest primarily to power users.

Component templates

A component template is a group of components and their associated code that you save as a unit. Component templates give you quick access to components that you frequently use in combination with one another. Any templates you've saved appear on the Templates tab of the Object Inspector. To reuse a component template, just choose it from the Component Palette and drop it on a form. All the components in the template will be placed on the form, complete with the code in any of their event

handlers. This feature is very convenient if you use certain groups of components over and over again.

New components

C++Builder 3 has many new components that you should be sure to check out. I'll categorize the new components by the Component Palette tab on which they appear.

Additional

TSplitter is a new VCL component. You place it between any two windows you want to split either horizontally or vertically. If you move the splitter, the associated windows will resize. The TCheckListBox component is a scrolling list of check boxes. The TStaticText component is a label component with its own window handle. This component is important for applications that depend on a label based on the Windows static text control.

TChart is a major new native VCL component (unlike the TVtChart ActiveX control) that makes charting easy. Be sure to look into TChart.

Win32

The Win32 tab was renamed from Win95 and contains several new components. The TAnimate component encapsulates the Windows animation control and lets you play AVI videos in a window. The TDateTimePicker component allows you to choose a date or a time in a preformatted style. The **Kind** property determines whether the component acts as a date picker or a time picker. When operating as a date picker, the component displays a drop-down calendar you can use to choose a date.

The two most significant components on the Win32 tab are TToolbar and TCoolbar. The TToolbar component lets you create a toolbar on which all the components (buttons, combo boxes, and so on) are automatically a uniform height. TCoolbar is a container component used primarily as a holder for TToolbars. You can arrange the bands (usually toolbars) on a TCoolbar and size them horizontally with sizing grips on the left side of the band. The Toolbar and Coolbar components take some time to figure out, but once you get the hang of how they work, they're very flexible.

Internet

The Internet tab contains all the Internet components and controls. This tab includes both VCL components and ActiveX controls. The NetManage ActiveX controls that shipped with C++Builder 1 have been replaced with new controls from NetMasters (the company that bought NetManage). The components on this page are so numerous and varied that we'll just have to say this tab is definitely worth checking out, and leave it at that.

Database Components

The new database components are primarily designed to make multitier database programming easier. The TProvider, TClientDataSet, and TRemoteServer components fall into this category. TDBChart is another great new database component. It's the data-aware equivalent of the TChart component we discussed earlier. This component makes it easy to generate charts from your database data.

The Client/Server version of C++Builder 3 contains an additional tab--Decision Cube. This tab's six components provide advanced data analysis. The QReport tab also has some new database-related components you'll want to examine.

Dialogs

The Dialogs tab has two new components, TOpenPictureDialog and TSavePictureDialog, that are similar to the existing file open and file save dialog boxes. However, they also provide a preview window for viewing graphics. Use these components when your application requires users to open BMP, WMF, EMF, or ICO files.

New language features

C++Builder 3 offers many new language features of which you should be aware. First, C++Builder now ships with Rogue Wave's Standard C++ Library 2.0. This is a combination of the classes formerly known as STL and some new classes, including new templatized IOStream classes. For the most part, your programs won't know the difference. In some cases, however, the new Standard C++ Library may require you to modify existing programs. You'll find a couple of new keywords in C++Builder 3. This version introduces the __finally keyword, which allows you to use the try/__finally construct. See the C++Builder documentation for more information.

Another extension allows dynamic functions in classes derived from TObject. A dynamic function is similar to a virtual function. The difference is that a dynamic function occupies space in the virtual table only in classes that define the function. A virtual function occupies space in *every* derived class, regardless of whether that class defines the virtual function. Dynamic functions make for smaller executables--but at the expense of execution speed--since the virtual tables of all derived classes are searched each time a dynamic function is called. The __declspec keyword has been expanded to take the dynamic parameter, as follows:

```
void __declspec(dynamic) MyFunction();
```

More project options

If you open the Project Options dialog box in C++Builder 3, you may be shocked at the

number of tabs it contains. There are now three tabs that represent the basic C++ compiler options: Compiler, Advanced Compiler, and C++. You can leave most of these options on their default settings. However, power users should appreciate the availability of these compiler options.

The Linker page has changed somewhat. Perhaps the most significant addition to the Linker page pertains to DLL projects. The Generate Import Library option tells C++Builder to create a LIB file for the DLL when the DLL is built. This means that you no longer have to run IMPLIB from the command line to create a LIB file for your DLL. The Version Info tab is another addition to the Project Options dialog box. This page contains everything you need to add version information to your projects. There's even an option to automatically increment the build number each time you make the project. The Packages tab lets you select whether your application will use runtime packages. The TASM page allows you to set the options that Turbo Assembler uses if your program contains inline assembly.

You'll notice that C++Builder 3 doesn't provide an option to turn off the incremental linker (ILINK). ILINK is the linker of choice and is the only linker available from the IDE. You can still use the old linker, TLINK, from the command line, but there's really no reason to do so.

ActiveX support

C++Builder now supports creating ActiveX controls using the Active Template Library (ATL). You can also create active forms, which are entire forms converted into ActiveX controls.

Database enhancements

In addition to the new database components we listed earlier, C++Builder 3 comes with a new Borland Database Engine (BDE). This version, 4.0, has many new features, so be sure to check the C++Builder documentation to see what's new. In general, there's much better support for multitiered database applications. Another database enhancement is the new SQL Builder application. This tool replaces the Visual Query Builder and allows you much greater flexibility over SQL statement creation. The SQL Explorer also has been enhanced.

Finally, shared common files!

This can hardly be considered a hot new feature, but users who keep several versions of C++Builder and Delphi on the same machine will be glad to know that Borland has created a new directory containing all the files common to C++Builder and Delphi. Most notably, the Images directory and the Win32 help files reside in the common directory. Now you can finally put all those space-hogging files in one location and share them among your Borland applications.

Documentation

This version of C++Builder has improved documentation. The book *Teach Yourself C++Builder in 14 Days* (written by yours truly) is included in all versions of C++Builder. This book is a subset of *Teach Yourself C++Builder in 21 Days* and is provided in lieu of a user's guide. The *Developer's Guide* provides more in-depth coverage of C++Builder and is a natural companion to the *Teach Yourself* book. Overall, you should find the new documentation (printed and online) an improvement over the previous version of C++Builder.

Conclusion

While this article doesn't cover everything that's new in C++Builder 3, it gives you a good idea what to expect when you upgrade. In general, C++Builder 3 is a more polished and professional product than C++Builder 1.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

April 1998

- [What's new in C++Builder 3]
- [DB->dilemmas]
- [Building graphic applications]
- [Manipulating memory with TMemoryStream]

April 1998

Building graphic applications

by A. Fasano

Some of the most interesting aspects of C++Builder are the tools it offers for the production of graphic applications. In this article, we'll demonstrate how you can use these tools quickly and simply by building a simple painter and applying some techniques of graphic rendering.

What's a painter?

A *painter* is a program that lets you create drawings and manipulate images. To give you a better idea, think in terms of Microsoft Paint in Windows. Starting with something simple, we'll develop a painter that contains the basic graphic operations--after that, you can expand its characteristics as you wish. Our program must support the following tasks:

- Freehand drawing
- Circle and rectangle drawing
- Filling an area with a color
- Selecting a color at a given point in an image
- Selecting a color from a palette
- Opening and saving an image file in the BMP format

Now that we've established the objectives of the application, let's take the first step in its development: designing the application.

Designing the form

To begin, start C++Builder and choose File | New Application. Save the project in a directory of your choice, renaming Unit 1 as *MainPainterForm* and Project1 as *Painter.* Then, select the form, press [F11] to move to the Object Inspector, and change the **Name** property to *PainterForm.* Next, you'll insert four elements, each of which has its own well-defined task. First, select the StatusBar component on the Win95 palette and position it inside the form (it doesn't make any difference where you click the mouse--C++Builder automatically places the status bar at the bottom of the form). The status bar will give information to the user--for example, the current position of the mouse.

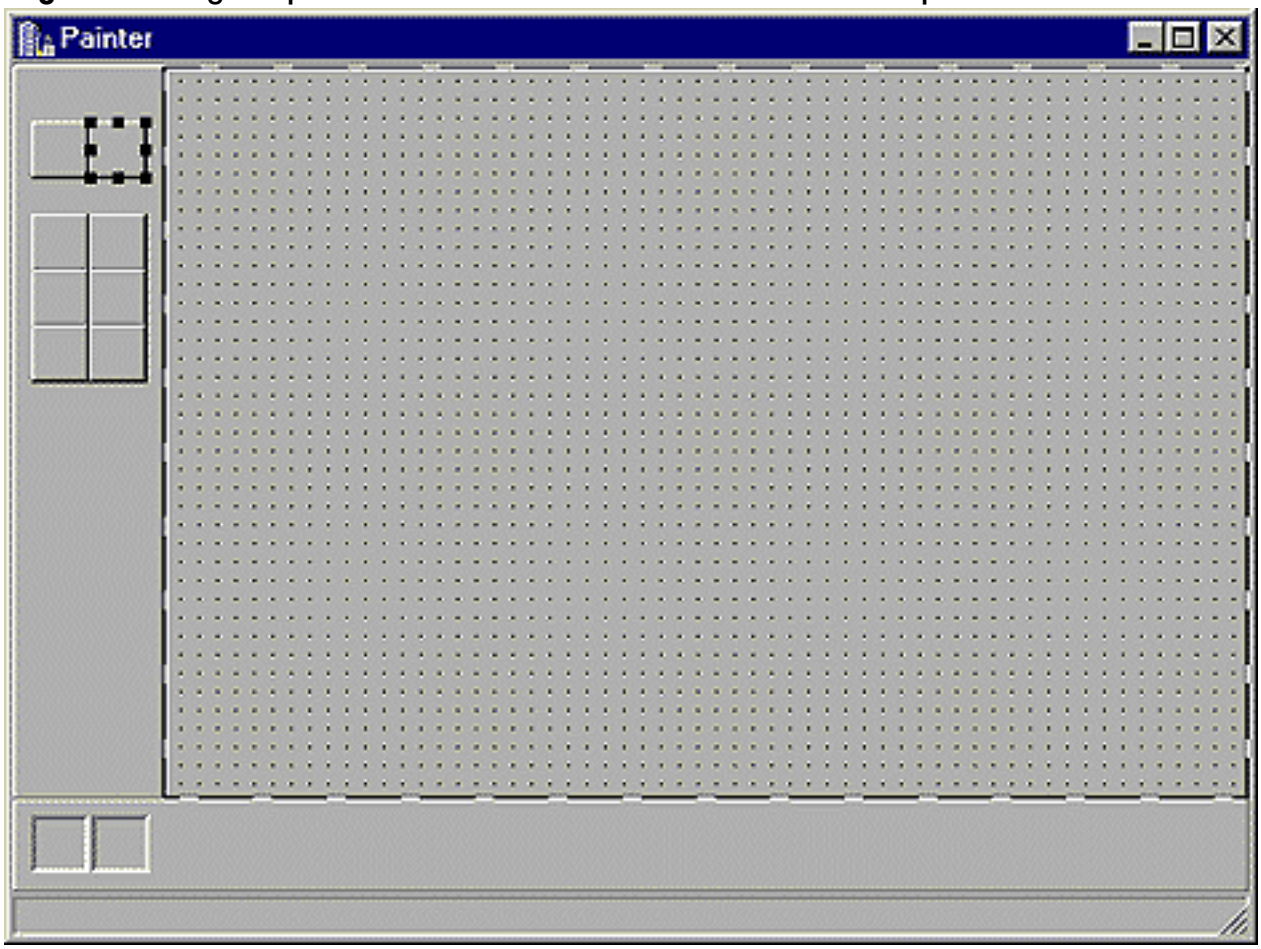
Now, select the Panel component on the Standard palette, and place it inside the form. This object will contain the colors used; give it a meaningful name, such as *PanelColors*. Click on the **Align** property in the Object Inspector and choose the option alBottom. Doing so will place the panel above the status bar, using all the available space.

You must place another panel on the form to contain the speed buttons. These buttons are the real heart of the program, since the user will click them to select an action to follow. Call this new panel *PanelButtons*, and set the **Align** property to alLeft.

Now you've arrived at the most important component of the application, which allows the realization of the true graphic operations. Select the PaintBox component from the System palette and place it in the form. Change the **Align** property to alClient, so that the component uses all the available space. (Note that this component encapsulates the TCanvas class, one of the most important classes oriented to VCL graphics. We'll discuss some of its characteristics in a moment.) To highlight the fact that the area the PaintBox component uses is dedicated to drawing, change the **Cursor** property to crCross. Now the mouse pointer's shape will turn into a small cross when the mouse scans above the paint box.

At this stage, you can begin adding the buttons that will affect the operations of the painter. Choose the SpeedButton component from the Additional palette and place eight buttons inside PanelButtons, as shown in Figure A.

Figure A: Eight speed buttons let the user control the painter's actions.



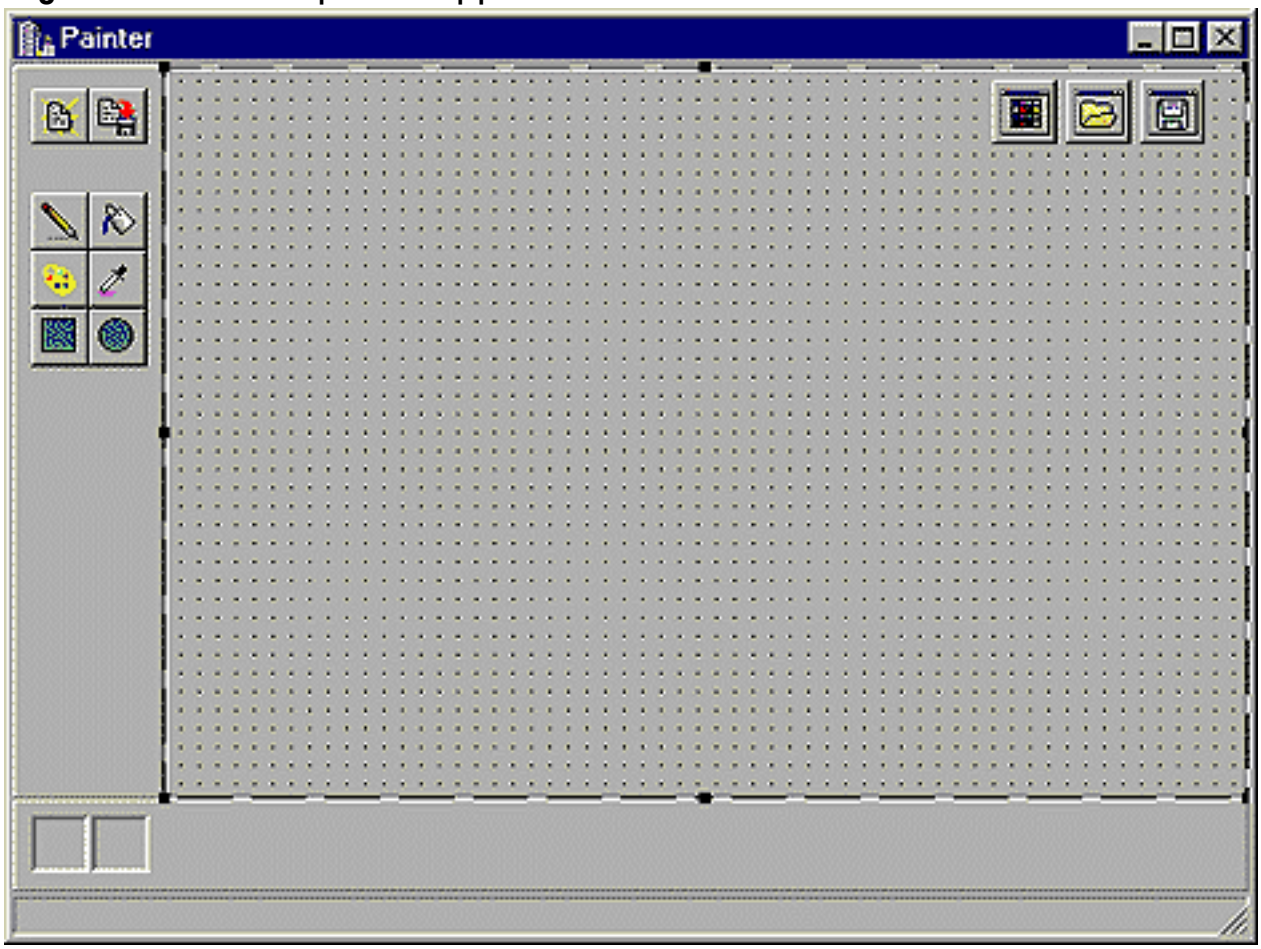
The first two buttons open and save the image; name these buttons *sbFileOpen* and *sbFileSave*. Click on the **Glyph** property to choose an image to associate with each button (the images Fileopen and Filesave in the directory \CBuilder\Images\Buttons are good for this

purpose). Name the other buttons *sbDrawPencil*, *sbFloodFill, sbPalette, sbPickColor, sbDrawRectangle*, and *sbDrawCircle.* Note that since these six buttons represent the user's actions, you must set each button's **AllowAllUp** property to false and set the **GroupIndex** property equal to 1. Assign meaningful icons so the user will immediately recognize the buttons; the icons we used came from the source file.

The last two essential visual components for the painter are two small panels. These panels will show the primary color associated with the left button on the mouse and the secondary color associated with the right button. Create two panels inside PanelColors and call them *pPenColor* and *pBrushColor*. Size these panels to 25 by 25 pixels; and, for an improved aesthetic effect, change the **BevelInner** and **BevelOuter** properties to bvLowered.

At this point you need to add the non-visual components for opening and saving files and for selecting color. These components, which are dialog boxes, reside on the Dialog palette: OpenDialog, SaveDialog, and ColorDialog. You can position them wherever you like, since they won't appear at runtime. Your application should now appear as shown in Figure B.

Figure B: The completed application form looks like this.



Let's draw!

Finally, you can set about writing a few lines of code. You need to keep the following tasks in mind:

- Starting the application
- Recognizing and executing the graphic applications

The solution for the first task is fairly simple: For now, just assign a pen color and a brush color. To do so, select the form, go to the Object Inspector, and double-click on the OnCreate event to open the Code Editor. Insert the following lines of code into the new method:

```
pPenColor->Color = clBlack;
pBrushColor->Color = clWhite;
PaintBox->Canvas->Pen->Color =
  pPenColor->Color;
PaintBox->Canvas->Brush->Color =
  pBrushColor->Color;
```

Now, the colors shown in the two small panels are really those employed by Canvas in the PaintBox. Let's move on. The user can perform six actions--draw freehand, fill an area, select a color from the palette or from the image, and draw circles and rectangles--but can do only one action at a time. Graphically, the speed buttons display this limitation by releasing each clicked button as another is clicked. This action is also handy (as you'll see) for defining the application internally.

In the Code Editor, right-click and select the first option, Open Source/Header File, from the speed menu. Given the simplicity of the situation, you can employ within MainPainterForm.h a type that contains all the possible actions, as follows:

```
enum TUserAction {aDrawPencil, aFloodFill,
  aChooseColor, aPickColor, aDrawRectangle,
  aDrawCircle, noAction};
```

Now, in the public section of the TFormPainter class declaration, add a variable of type TUserAction, as follows:

```
/*** class definition ***/
  public:
      TUserAction UserAction;
```

This variable will count each action as the user runs it. How do you use UserAction? First, it must be filled when the user clicks one of the six buttons. A good way to do this is to use a shared event handler. Select a button on the form, then hold down the [Shift] key and select

the other five buttons. Press [F11] to open the Object Inspector. Then, select the OnClick event, type the word *SelectAction*, and press [Enter]. The Code Editor will open with the following method:

```
void __fastcall
  TFormPainter::SelectAction(TObject *Sender)
{

}
```

In this way, every time the user clicks a design button, the SelectAction method will be invoked. You can determine which button invoked the event handler through the Sender parameter. Filling the UserAction variable then becomes easy:

```
if (Sender == sbDrawPencil) UserAction =
  aDrawPencil;

if (Sender == sbDrawCircle) UserAction =
  aDrawCircle;
```

Now you have all the necessary tools to implement the various graphic operations.

The pencil draw and the flood fill

Suppose the user clicks the sbDrawPencil button. It's possible that he'll move the mouse pointer over the paint box and, holding down either the left or right mouse button, move the mouse to draw something. In this case, you must intercept two mouse actions:

- A simple click
- A movement while one of the mouse buttons is held down

The first action doesn't create any great problems. Select the PaintBox object on the form, then double-click on the OnMouseDown event in the Object Inspector. In the code editor, you'll see the PaintBoxMouseDown method, which is invoked every time the user single-, double-, or triple-clicks above the PaintBox object. Within this method, going by the user's action (with the value of UserAction), you can have various solutions. A switch handles the task of filtering, as follows:

```
void __fastcall
  TFormPainter::PaintBoxMouseDown(
  TObject *Sender,
  TMouseButton Button, TShiftState Shift,
```

```
    int X, int Y)
{
  switch (UserAction)
  {
   case aDrawPencil: /*** some code ***/
     break;
   case aFloodFill:  /*** some code ***/
     break;

   /*** other actions ***/
  }
}
```

In the related case of aDrawPencil, you need to turn on a pixel in the mouse position and handle the mouse click. Fortunately, the method parameters provide all the necessary information: X and Y contain the position of the mouse, while Button contains the button pressed. To turn on a pixel, you can use the **Pixels** property of Canvas in PaintBox:

```
case aDrawPencil:
   if (Button == mbLeft)
     PaintBox->Canvas->Pixels [X][Y] =
       pPenColor->Color;
   else if (Button == mbRight)
     PaintBox->Canvas->Pixels [X][Y] =
       pBrushColor->Color;
   break;
```

Have a go at compiling the code, and have some fun! When you're finished, you'll see something a little strange: If you hold down the mouse button and move the mouse, the application doesn't draw a continuous line of points--only a few of the pixels turn on. This happens because you haven't yet dealt with the management of the *moving* mouse. Let's fix that.

Select the PaintBox object, go again to the Object Inspector, and double-click on OnMouseMove. The PaintBoxMouseMove method that appears in the Code Editor will be invoked at runtime every time the user moves the mouse above the component. As in the last example, you must insert a switch to filter the user's actions.

For freehand drawings, however, it isn't enough to turn on just one pixel, because the mouse's movement can be so fast that the drawn line won't be continuous. To solve the problem, you can take advantage of a little trick: Draw a line between the current position of the mouse and its previous position. You can do so thanks to two methods in the TCanvas class: MoveTo and LineTo.

Unfortunately, the OnMouseMove event doesn't give you information about the previous position of the mouse. So, you'll need to add OldMouseX and OldMouseY variables to the TFormPainter class. You must initialize these variables at the press of a button and update them during the mouse's movement. You then have a situation like the following:

```
void __fastcall TFormPainter::PaintBoxMouseMove(
  TObject *Sender,
  TShiftState Shift, int X, int Y))
{
  switch (UserAction)
  {
   case aDrawPencil:
     if (Shift.Contains (ssLeft)) {
       PaintBox->Canvas->MoveTo (OldMouseX,
         OldMouseY);
       PaintBox->Canvas->LineTo (X, Y);
       OldMouseX = X;
       OldMouseY = Y;
     }
     break;

 /*** other actions ***/
  }
}
```

First, you need to check whether a button has been clicked, in this case by using the Contains method of the Shift set. If a button has been clicked, you move the first point of the drawing to the last position of the mouse, thanks to the MoveTo method. Then you trace a line to the current position using the LineTo method. Finally, you update the variables. (In the source code, the right mouse button is also checked.)

Now, let's look at how you fill an area. Doing so is fairly simple, since the TCanvas class provides a function. Within the PaintBoxMouseDown method, in the case relative to the flood fill, insert the following code:

```
case aFloodFill:
  PaintBox->Canvas->FloodFill (
    X, Y, PaintBox->Canvas->Pixels[X][Y],
    fsSurface);
  break;
```

The FloodFill method fills an area with the color of the brush. This method needs to know the first point to fill, so you'll use X and Y, the coordinates of the mouse. fsSurface specifies that the fill must occur in an area where a color is well-defined. In this case, you'll take the color of the pixel relative to the position of the mouse, which is indicated by PaintBox-> Canvas-> Pixels[X][Y]. As an alternative, you can use fsBorder in place of fsSurface. When you do, you can fill the area as long as it doesn't contain the indicated color.

Drawing rectangles and circles, and picking a color

Drawing circles and rectangles is really very simple, since TCanvas offers some ready-made methods. To draw, the user must click the sbDrawRectangle or sbDrawCircle button, then move the mouse while holding down the left button to indicate the initial point of the drawing. Place the following code in the PaintBoxMouseDown method:

```
case aDrawRectangle:
case aDrawCircle:
  OldMouseX = X;
  OldMouseY = Y;
  break;
```

In the PaintBoxMouseMove method, on the other hand, you can draw a rectangle using the Rectangle method. This method automatically uses the pen color for the rectangle's borders and the brush color for its inside. The same is true for a circle, using the Ellipse method:

```
case aDrawRectangle:
  if (Shift.Contains (ssLeft))
     PaintBox->Canvas->Rectangle (
       OldMouseX, OldMouseY, X, Y)
  break;
```

Let's move on to setting colors. When we discussed the flood fill, you saw how to determine which color is in a given position within the drawing area. If the user clicks the sbPickColor button, you proceed in a similar fashion: If the left button is clicked, you assign the chosen color to the pen; if the right button is clicked, you assign the chosen color to the brush. Insert the following code into the PaintBoxMouseDown method:

```
case aPickColor:
TColor color = PaintBox->Canvas->Pixels [X][Y];
  if (Button == mbLeft) {
    pPenColor->Color = color;
    PaintBox->Canvas->Pen->Color = color;
  } else if (Button == mbRight) {
```

```
    pBrushColor->Color = color;
    PaintBox->Canvas->Brush->Color = color;
    }
  break;
```

To make things easier when choosing a color from a palette, you can just insert this code directly into the SelectAction method. Fortunately, the ColorDialog dialog box will do all the work for you. It behaves in an analogous way to the other dialog boxes: After you execute it via the Execute method, if it's successful, it assigns the color chosen by the user to the **Color** property:

```
if (ColorDialog->Execute())
  {
   pPenColor->Color = ColorDialog->Color;
   PaintBox->Canvas->Pen->Color =
     colorDialog->Color;
  }
```

Conclusion

In this article, we've shown you how to produce some simple graphic elements, primarily using the characteristics of the TCanvas class. We also began developing a drawing program, analyzing the various problems associated with it. In a future article, we'll increase this program's potential by introducing the concept of offscreen bitmaps.

April 1998

Manipulating memory with TMemoryStream

by Kent Reisdorph

In the March issue *of C++Builder Developer's Journal*, the article "File I/O" explained the TFileStream class and how you can use this class to read and write to files on disk. Sometimes, however, you need to manipulate a chunk of data in memory rather than on disk. The good news is that TFileStream has a cousin called TMemoryStream designed for exactly this purpose. In this article, we'll show you how to use TMemoryStream to treat memory the same way you treated a file in our March article. We'll demonstrate how to use the same functions to read and write to memory as you would when performing file I/O.

OK, but why?

It may not immediately be apparent why you might want to deal with memory as you would a file. There are several good reasons, but the most compelling is speed of execution. Any time you manipulate data, execution speed is a concern. Let's say, for instance, that you want to save data to a file at a time-critical point in your program. You might not be able to afford the overhead associated with writing to a file on disk during a real-time process. Instead, you could write the data to memory, then copy the data from memory to disk after the real-time process is complete.

Here's another example: Sometimes you need to read a file from disk, manipulate the file in some way (encryption/decryption, for instance), and then write the file out again. Writing the file to memory and making the modifications there is much faster than reading the file, modifying data, then writing the file every time changes are necessary.

Finally, if you've ever performed operations on string data and then copied those strings to a TMemo, you know how slow this process can be. As an alternative, you can load the strings into memory and work with them there, copying the result to TMemo only when the data has been fully altered. We'll show you an example in just a bit, but for now let's take a closer look at what TMemoryStream offers.

TMemoryStream basics

TMemoryStream provides several properties and methods for working with data in memory. Table A lists the primary ones.

Table A: Important TMemoryStream properties and methods

Property	Description
Position	The current value of the stream position indicator.
Position	is a read/write property.
Size	The number of bytes of data currently in the stream.
Method CopyFrom()	Copies a specified number of bytes from a stream to this stream.
LoadFromFile()	Fills the memory stream with the contents of the specified file.
LoadFromStream()	Fills the memory stream from another stream (either file or memory).
Read()	Reads a specified number of bytes from the stream to the specified memory location (such as an array).
Write()	Writes a specified number of bytes from a memory location to the stream.
SaveToFile()	Saves the contents of the memory stream to a disk file.
SaveToStream()	Saves the contents of the memory stream to another stream.
Seek()	Moves the file-position indicator by the specified amount from the start of the file, from the end of the file, or from the current position.
SetSize()	Allocates the specified amount of memory for the stream.

We covered the general concept of streams last month in "File I/O." The TMemoryStream properties and methods work identically to the TFileStream properties and methods discussed in that article. Rather than cover that ground again, let's take a look at some of the methods particular to TMemoryStream.

The Loadxxx and Savexxx methods simply make it easy to load or save a stream to and from either a file on disk or another memory stream. For example, you could load a file on disk, manipulate the data in memory, and save it back to disk, as follows:

```
TMemoryStream* stream = new TMemoryStream;
stream->LoadFromFile("myfile.dat");
// do some stuff to the stream
stream->SaveToFile("myfile.dat");
delete stream;
```

The SetSize() method allows you to set the amount of memory allocated for the stream. Setting the size isn't a requirement, because TMemoryStream will automatically allocate and deallocate memory as needed. If, however, you know the minimum initial size you'll need, you can set the size to that value, saving quite a few clock cycles if your operation is time-critical. For example, we ran tests on writing a 324Kb memory stream one byte at a time. Memory for streams is allocated in 8Kb blocks. Therefore, writing 324Kb of data to a stream

results in 40 memory reallocations. When we set the stream size to 324Kb prior to writing the data, the average time of the operation decreased from 950 milliseconds to 570 milliseconds by letting VCL handle the memory automatically. That's a significant savings.

An example

Probably the best thing we can do at this point is to provide an example. Let's say you want to obfuscate a text file. You can do so simply by altering the ASCII value of each character slightly, in effect scrambling the file to make it unreadable. Later, you can descramble the file to view it. Or, better yet, you can descramble the file and send the text directly to a Memo component. The whole operation might look as follows:

```
TMemoryStream* stream = new TMemoryStream;
stream->LoadFromFile(OpenDialog1->FileName);
stream->Position = 0;
for (unsigned int i=0;i<stream->Size;i++) {
  char c;
  stream->Read(&c, 1);
  c += 32;
  stream->Position--;
  stream->Write(&c, 1);
}
// file is scrambled in memory
// later... descramble and display
stream->Position = 0;
for (unsigned int i=0;i<stream->Size;i++) {
  char c;
  stream->Read(&c, 1);
  c -= 32;
  stream->Position--;
  stream->Write(&c, 1);
}
stream->Position = 0;
Memo1->Lines->LoadFromStream(stream);
```

Most of this code is pretty straightforward. Notice this line, though:

```
stream->Position--;
```

After you read the character at the current stream position, the stream-position counter advances by one. You need to back up one character before you write the scrambled character to the stream. Notice also that once you've descrambled the stream, you can easily

load it into the Memo component using the LoadFromStream() method. This method, which is actually a member of TStringList, is very fast and powerful. You can use it in combination with memory streams anywhere you use string lists (and many VCL components use TStringList to store their data).

Listings A and B contain a program that scrambles a text file, descrambles it, and then displays the file in a Memo component. The elapsed time of the operation is displayed in a Label component. This program illustrates the power of memory streams and shows the speed of working with data via memory streams compared to working directly with the Memo's text.

Listing A: STREAMU.H

```
#ifndef StreamUH
#define StreamUH
//---------------------------------
        ---------#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include <vcl\Dialogs.hpp>
//-------------------------------
        --------------class TForm1 : public TForm
{
__published:  // IDE-managed Components
  TMemo *Memo1;
  TLabel *Label1;
  TButton *Button3;
  TLabel *Label2;
  TOpenDialog *OpenDialog1;
  TGroupBox *GroupBox1;
  TButton *Button1;
  TButton *Button2;
  void __fastcall Button1Click(TObject *Sender);
  void __fastcall Button2Click(TObject *Sender);
  void __fastcall Button3Click(TObject *Sender);
  void __fastcall FormCreate(TObject *Sender);
private:  // User declarations
public:   // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//-------------------------------
        --------------extern TForm1 *Form1;
```

```
//---------------------------------------------#endif
```

Listing B: STREAMU.CPP

```
#include <vcl\vcl.h>
#pragma hdrstop

#include "StreamU.h"
//---------------------------
        ----------------#pragma resource "*.dfm"
TForm1 *Form1;
//-------------------------------------
        -------__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//-----------------------------------------
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
  int start = GetTickCount();
  Memo1->Lines->
    LoadFromFile(OpenDialog1->FileName);
  char* buff = Memo1->Lines->GetText();
  Screen->Cursor = crHourGlass;
  for (unsigned int i=0;i<strlen(buff);i++) {
    char c = buff[i];
    c += 32;
    buff[i] = c;
  }
  for (unsigned int i=0;i<strlen(buff);i++) {
    char c = buff[i];
    c -= 32;
    buff[i] = c;
  }
  Memo1->Lines->Text = buff;
  Screen->Cursor = crDefault;
  char buff2[20];
  sprintf(buff2, "%.02f seconds",
    (GetTickCount() - start)/1000.0);
  Label2->Caption = buff2;
}
//-----------------------------------------
```

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  TMemoryStream* stream = new TMemoryStream;
  stream->LoadFromFile(OpenDialog1->FileName);
  Screen->Cursor = crHourGlass;
  stream->Position = 0;
  for (unsigned int i=0;i<stream->Size;i++) {
    char c;
    stream->Read(&c, 1);
    c += 32;
    stream->Position--;
    stream->Write(&c, 1);
  }
  stream->Position = 0;
  for (unsigned int i=0;i<stream->Size;i++) {
    char c;
    stream->Read(&c, 1);
    c -= 32;
    stream->Position--;
    stream->Write(&c, 1);
  }
  stream->Position = 0;
  Memo1->Lines->LoadFromStream(stream);
  Screen->Cursor = crDefault;
  delete stream;
  char buff[20];
  sprintf(buff, "%.02f seconds",
    (GetTickCount() - start)/1000.0);
  Label2->Caption = buff;
}
//---------------------------------------------
void __fastcall
TForm1::Button3Click(TObject *Sender)
{
  if (OpenDialog1->Execute()) {
    Button1->Enabled = true;
    Button2->Enabled = true;
    Memo1->Text =
      "File Selected: " + OpenDialog1->FileName;
  }
}
//---------------------------------------------
void __fastcall
```

```
TForm1::FormCreate(TObject *Sender)
{
  Button1->Enabled = false;
  Button2->Enabled = false;
}
```

Run the program and try both methods. Be sure you try the program with some larger text files (more than 50Kb) in order to see the difference that memory streams can make. You'll find that the memory stream method is as much as 100 times faster than working directly with the Memo text.

Conclusion

Memory streams are very powerful. Once you learn how to use TMemoryStream, particularly in combination with TFileStream and TStringList, new programming vistas will open before your eyes.

April 1998

Are you a good tipper?

Have you discovered great C++Builder tips during your development work? We'd like to feature your handy shortcuts and quick code tricks on our free weekly online ZDTips service (**www.zdtips.com**). We'll give you a byline and include your E-mail address, company name, and Web URL. Send your tips to **cbuilder_dev@cobb.com** or:

C++Builder Developer's Journal
The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

. zdtips.com .

Web Search

Mon, 23 Dec 2002 GMT

Top Searches

- Weight Loss
- Jewelry
- Flowers
- Gift Baskets
- Wine
- Wedding Gifts
- Gift Certificate
- Books
- Computer Games
- Electronics
- Cars
- Computers
- Toys
- Movies
- Music
- Hotels
- Travel
- Airline Tickets
- Health Insurance
- Sports

Hobbies
Fitness
Sports
Cooking
Gardening
Crafts
Home Decorating

Travel
Airline Tickets
Vacations
Hotels
Travel
Maps
Adventure Travel

Finance
Investing
Stocks
Credit Cards
Real Estate
Insurance
Loan

Health
Weight Loss
Nutrition
Fitness
Women's Health
Health Insurance
Life Insurance

Home
Home Loan
Home Buying
Home Improvement
Gardening
Pets
Interior Design

Shopping
Cars
Computers
Toys
Books
Gifts
Electronics

Gifts
Jewelry
Flowers
Gift Baskets
Wine
Wedding Gifts
Gift Certificate

Entertainment
Sports
Music
Games
Movies
Travel
Arts

Internet
Computers
Computer Hardware
Software
Computer Games
Computer Networking
Web Design

March 1998

File I/O

by Kent Reisdorph

At some point, every programmer needs to do file input and output. When we talk about file I/O, we don't mean database files--the database tools in C++Builder are great for file operations that fit within the database paradigm. However, sooner or later you'll need specialized file I/O. In this article, we'll show you the ins and outs of file I/O. We'll discuss the many different types of file I/O available to you in C++Builder, and we'll demonstrate how to use two types: the C++ iostream classes and the VCL TFileStream class.

This article is long because of the complicated nature of file I/O, but the material presented is vital to understanding file operations. Take it a little at a time and be sure to experiment along the way.

So many choices!

C++Builder gives you several ways to perform file I/O. These choices include:

- The C-style FILE mechanism
- The C++ iostream classes
- The VCL streaming classes (TFileStream)
- The VCL database mechanism

Given this selection of file I/O methods, which do you choose? The answer, of course, depends on the type of file I/O you're doing and, to a degree, on your programming experience. For example, if you come from a C background, then you might already be familiar with that method of doing file I/O. If you come from a C++ background, then you might be familiar with iostreams. If you come from Delphi, then you might already have experience with TFileStream. The important thing is that you know what your choices are and then pick the appropriate method for a given programming chore.

So, back to the question of which method to choose. Database programming is...well, database programming, so we can cross the last choice off the list for general file I/O.

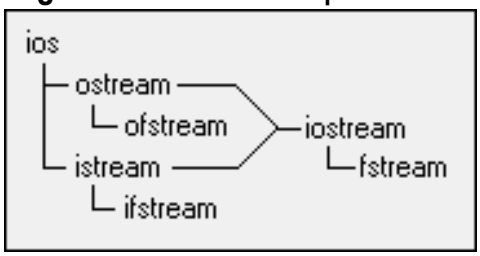
We're going to cross the C-style FILE mechanism off the list as well, even though for some folks this is the old standby method. If you currently use this method for file I/O, then there's certainly no harm in continuing to do so. However, this mechanism lacks the object-oriented design that's so prevalent in programming today. The C-style file I/O functions use the FILE structure as a sort of file handle. Functions in this group include fopen(), fread(), fwrite(), fclose(), fseek(), and ftell(). We'll add one thing in defense of the C-style file I/O functions: They're very portable. If portability across multiple platforms is key for you, then you might consider this method. On the other hand, if you're using C++Builder, then portability probably isn't a concern.

That leaves us with two remaining choices: the C++ iostream classes and the VCL streaming class, TFileStream. These two methods of file I/O are quite similar, but there are some major differences. Having a good knowledge of what each method offers allows you to make an informed choice about which to use in a given situation. We'll look at each method in detail; if you'd first like an overview of the *streaming* concept, see "Today's Buzzword: Streams," which follows this article.

C++ iostream classes

A discussion of the C++ standard library streaming classes could easily occupy an entire book. Obviously we can't go into everything here, so we'll just hit the high points. The main classes you'll be concerned with are the ifstream class (for reading files), the ofstream class (for writing files), and the fstream class (for reading and writing files simultaneously). These classes ultimately derive from the granddaddy of all C++ classes, ios. (Even though all the stream classes are derived from ios, they're often referred to as the "iostream classes." Figure A shows the hierarchy of the ifstream, ofstream, and fstream classes.

Figure A: Here is a portion of the ios hierarchy.



As you can see, the fstream class is derived from iostream, which is derived from both ostream and istream. This structure allows simultaneous reading and writing of a file. (By simultaneous, we mean that you can open a file, write to part of it, change the file position pointer, read from the file at the new location, and so on. The need to write and read a file simultaneously is fairly specialized, so we won't spend much time talking about it.)

Overall, the iostream classes are very powerful and also quite portable (again, if portability is important to you). If you're new to C++ programming, then you may be somewhat dismayed to learn that Borland inadvertently omitted the Help file describing the iostream classes from the C++Builder CD. If you have a previous Borland C++ compiler, you can look in CLASSLIB.HLP for help on the iostream classes; in addition, we'll try to give you enough information so that you can at least begin to use the stream classes. First, let's discuss some information common to all file operations. Then we'll look at reading and writing files individually.

iostream basics

As Figure A shows, the file stream classes derive from ios. As such, most of the basic streaming functionality comes from the ios class. Still, the ios class has only a couple of items in which

we're interested when discussing file I/O. First are the open_mode flags. open_mode is an enumeration defined as follows:

```
enum open_mode { app, ate, in, out, binary, trunc, nocreate, noreplace };
```

Table A lists the open_mode members and their descriptions.

Table A: open_mode enumeration members

Member	Description
app	Append data--always write at end of file.
ate	Seek to end of file upon original open.
in	Open for input (default for ifstreams).
out	Open for output (default for ofstreams).
binary	Open file in binary mode.
trunc	Discard contents if file exists (default if out is specified and neither ate nor app is specified).
nocreate	If file does not exist, open fails.
noreplace	If file exists, open for output fails unless ate or app is set.

The open_mode enumeration acts as a set of flags that you can or together to specify the file mode when you open a file. For example, let's say you want to open a file in binary mode and append data to the end of the file. The code will look something like this:

```
ifstream file;
file.open("data.txt", ios::app | ios::binary);
```

Note that you need to qualify the flag values with the ios class name since open_mode is defined within the ios class. As another example, suppose you want to open a file for writing but ensure that you don't accidentally overwrite an existing file of the same name. You can use the following code:

```
ofstream file;
file.open("data.txt", ios::out | ios::nocreate);
```

In this case, the call to open() will fail if a file already exists by the same filename. The other significant aspect of the ios class is the eof() method. This method returns a non-zero value when the stream position indicator reaches the end of the file. You can use eof() to determine when you've reached the end of the file when reading data files. We'll show you examples a little later.

ios insertion and extraction operators

The stream insertion operator (<<) writes data to a stream. Let's say you've opened a file for writing. Once the file is open, you can write a line of text to the file like this:

```
file << "This is a test";
```

You can chain multiple insertion operations. For example, the following line of code has the same effect as the previous example:

```
file << "This is " << "a test";
```

Note that neither example terminates the string with a carriage return. You can add a CR in one of two ways:

```
file << "This is a test\n";
// or...
file << "This is a test" << endl;
```

The first case appends the standard C "\n" escape sequence to the string. The second case uses the endl operator (endl is short for "end of line"). You can also write variables to a file using the insertion operator. For example:

```
char* buff = "This is a character array.";
int x = 100;
double d = 99.9;
char c = `a';
file << buff << endl << x
  << endl << d << endl << c << endl;
```

Executing this code will result in a text file with the following contents:

```
This is a character array.
100
99.9
a
```

The important thing to realize here is that the numeric values are converted to strings and stored in the file as text strings, not as binary data as you might expect. The extraction operator (>>) works in the opposite manner… more or less. You'd think that to extract the characters from the file written in the previous example, you could use code like the following:

```
char buff[40];
int x;
double d;
char c;
file >> buff >> x >> d >> c;
```

The problem is that the extraction operator stops at the first white-space character it encounters (*white space* includes things like spaces, tabs, and the new-line character). For instance in the sentence "This is a character array," the word *This* is read into the variable buff, the space character is ignored, and then the word *is* is read into the integer variable x--obviously not what you had in mind. From this point on, things go haywire and data corruption is inevitable. So while the extraction operator is great in some cases, it isn't the do-all method of reading files. In just a bit we'll talk about the ifstream class, and then we'll show you how to correctly read the file in the previous example. Note that you can define your own operator << and operator >> for your C++ classes. Once you've defined these operators, you can write code like this:

```
MyClass mc1;
MyClass mc2;
// save the class data to a file
file << mc1 << mc2;
```

And then later...

```
MyClass mc1;
MyClass mc2;
// read the class data from the file
file >> mc1 >> mc2;
```

Provided that you've written your insertion and extraction operators correctly, the class is saved and restored as easily as that. Unfortunately, a discussion of overloading the << and >> operators is beyond the scope of this article, so we'll leave you with that little tidbit running around in your head. Now, let's move on to a detailed look at file I/O.

ifstream: reading files

Reading files is perhaps more common than writing files, so we'll start there. The ifstream class reads files on disk (the *if* in ifstream stands for *input file*). Table B lists the basic ifstream methods. Keep in mind that many of these methods come from ifstream's ancestor class, istream. Most of these functions are overloaded to provide extended functionality. The get() function, for instance, can read a single character or a block of characters, depending on the calling parameters.

Table B: Important ifstream methods

Method	Description
close()	Closes the file.
get()	Reads one or more characters from the file.
getline()	Reads a line of text from a text file or reads data from a binary file until the specified delimiter is read.
open()	Opens the file for reading
peek()	Looks at the next character in the stream but doesn't extract the character.
read()	Reads a specified number of characters from the file into memory.
seekg()	Moves the file position indicator to a specific location in the file.
tellg()	Returns the value of the file position indicator.

As always, an example will go a long way toward furthering your understanding of file I/O operations using ifstream. The code shown in Listing A reads each line of a text file and then adds the line to a Memo component.

Listing A: Using ifstream

```
#include <fstream.h>
// create an instance of ifstream
ifstream file;
// open the file
file.open("unit1.cpp");
// something went wrong
if (!file) return;
// create a buffer for storage
char buff[80];
// loop while not at the end of file
while (!file.eof()) {
  // read a line from the file
  file.getline(buff, sizeof(buff));
  // add it to the Memo
  Memo1->Lines->Add(buff);
}
// close the file
file.close();
```

Notice first that you must include the FSTREAM.H header file, which contains the declarations for all the file-streaming classes. Next, notice that you use the open() method to open a file. You don't have to specify any of the open_mode flags because you want the default ifstream

flags of ios::in when reading a text file.

After the call to open(), you check to see whether the file was opened successfully and return if the open operation failed. Next, you read one line at a time from the file with the getline() method, which will retrieve characters until it encounters a CR pair. Now you add the line to the Memo component.

At the top of the loop, you check for the end of file with the eof() method. When the entire file has been read, you close the file with the close() method. (This example ignores the fact that the Lines property of TMemo has a LoadFromFile() method, which is a much easier way of loading a file into a Memo component.)

In this example, we used the open() and close() functions simply to illustrate a point--they aren't strictly needed. The open() function isn't needed because you can use one of the ifstream constructors to create the file object and open the file all at once, as follows:

```
ifstream file("unit1.cpp");
if (!file) return;
```

The close() function isn't strictly necessary because the ifstream destructor will close the file for you. (You can certainly call close() explicitly if you wish.) Finally, we wrote the last few steps of the sample code in a way that highlights their purposes--but as written, the code adds one extra blank line of text to the Memo. In order to be technically correct, the code that reads the lines of text should look like this:

```
while (!file.getline(buff, sizeof(buff)).eof())
  Memo1->Lines->Add(buff);
```

While this code is sort of ugly and hard to read, it's nevertheless the proper way to check for the end-of-file indicator. Remember our earlier example of the extraction operator that incorrectly read a file? Here's the proper way to read the data:

```
ifstream file("temp.txt");
char buff[40];
int x;
double d;
char c;
file.getline(buff, sizeof(buff));
file >> x >> d >> c;
```

First you get the line of text with the getline() method. After that, you can use the extraction operator to read the remaining data from the file. A little later we'll talk more about reading and writing binary data files, but for now let's move on to writing files with the ofstream class.

ofstream: writing files

The ofstream class is the functional opposite of the ifstream class. You'll use ofstream to write data to files on disk. When it comes right down to it, writing files isn't all that complicated. Table C lists the important methods of the ofstream class. For the most part, these functions require little explanation; we'll provide examples in just a bit.

Table C: Important ofstream methods

Method	Description
close()	Closes the file.
put()	Writes a single character to the file.
open()	Opens the file for writing.
write()	Writes a specified number of bytes from memory to the file.
seekp()	Moves the file position indicator to a specific location in the file.
tellp()	Returns the value of the file position indicator.

Earlier, we discussed the ios insertion and extraction operators. The insertion operator works very well to write text. The following example writes 10 lines of text to a file:

```
ofstream outfile("temp.txt");
if (!outfile) return;
for (int i=1;i<11;i++)
  outfile << "This is line #" << i << endl;
outfile.close();
```

Notice that each line is terminated using the endl manipulator. After this code executes, the file TEMP.TXT looks like this:

```
This is line #1
This is line #2
This is line #3
This is line #4
This is line #5
This is line #6
This is line #7
This is line #8
This is line #9
This is line #10
```

To prove this, create a new project in C++Builder and enter the previous code in response to a button click (don't forget to include FSTREAM.H.). After you run the program, open TEMP.TXT in the C++Builder editor, and the file should contain the lines we showed you. You can create this same file using the write() method, but it's more cumbersome. To do so, the for loop in the code would look like this:

```
for (int i=1;i<11;i++) {
  char buff[20];
  sprintf(buff, "This is line #%d\n", i);
  outfile.write(buff, strlen(buff));
}
```

The end result is the same, but using the << operator is much cleaner. While the write() method isn't the best for writing text files, it's much more important when writing binary files. Let's take a look at that next.

Dealing with binary data

Dealing with binary data is somewhat different from dealing with text data. For one thing, the data must be written in some logical arrangement and then read in exactly the same way. A data structure like the following allows you to do that fairly easily:

```
struct Data {
  char Name[20];
  char Phone[20];
  int  Age;
  int  ID;
};
```

This is a logical, albeit simple, data arrangement. Writing this structure to disk using ofstream is as simple as

```
Data MyData = {"Billy Bob", "none", 36, 1};
ofstream outfile("names.dat", ios::binary);
outfile.write((char*)&MyData, sizeof(Data));
```

The write() method expects a char* rather than a void*, so you must take the address of the structure and cast it to a char*. You write the exact number of bytes contained in a data structure (sizeof(Data)), which means that the same number of bytes is written regardless of the data in the structure. The code writes the file in block format with each block occupying the same number of bytes. (Later, you can use this arrangement to read a particular block in the file.) Notice that we used the ios::binary flag when we opened the file for writing. If you write a file in binary mode, then you also need to specify the ios::binary flag when you open the file for

reading. Speaking of reading a file, reading the binary data is just as easy:

```
ifstream infile("names.dat", ios::binary);
if (!infile) return;
Data MyData;
infile.read((char*)&MyData, sizeof(Data));
```

Here, the structure is filled with the bytes read from the file. You can then do whatever you want with the data in the structure. To read raw binary data from a file one byte at a time, use get(); to write to a file, use put(). For example, a file-copy operation might look like this:

```
ifstream infile("names.dat", ios::binary);
ofstream outfile("temp.fil", ios::binary);
infile.seekg(0, ifstream::end);
int numBytes = infile.tellg();
infile.seekg(0);
for (int i=0;i<numBytes;i++) {
  char c;
  infile.get(c);
  outfile.put(c);
}
```

The type of data you're dealing with will dictate the method you use to read and write binary data.

Random file access

Imagine you have a file containing 1,000 records like those we just discussed. Let's further say you want to read record number 999. You could loop through the file, reading records until you finally get to record 999. Obviously this isn't very efficient. A better way would be to set the file-stream pointer to the exact location of record 999 in the file, then read just that record. In the world of file I/O, this is known as *seeking*. Seeking works effectively only when a file is filled with records of a known size, or if you know the exact layout of a file. In this case, you know the record size, so getting the correct file position is a matter of a simple calculation:

```
int pos = 998 * sizeof(Data);
```

Now you can open the file, seek to record number 999, and read the record at that position, as follows:

```
ifstream infile("names.dat", ios::binary);
infile.seekg(pos);
Data MyData;
```

```
infile.read((char*)&MyData, sizeof(Data));
```

Note that since the first record is at file position 0, the 999th record is at 998 multiplied by the size of a record. Similarly, you can replace or update a record in a file. To do so, you need to open the file in update mode:

```
ofstream outfile(
  "names.dat", ios::binary | ios::ate);
int pos = 998 * sizeof(Data);
outfile.seekp(pos);
outfile.write((char*)&MyData, sizeof(Data));
```

Here you use the ios::ate flag in addition to the ios::binary flag. You could use ios::app (append mode) and achieve the same results. If you hadn't specified ios::ate, then the previous file would have been overwritten when the file was opened (oops!). The rest is pretty straightforward--just seek to the appropriate place in the file and write the new information to the file. Finally, you could open the file for simultaneous reading and writing by using the fstream class rather than using ofstream to write a file and ifstream to read the file. Since fstream is derived from both ostream and istream (the ancestor classes of ofstream and ifstream, respectively), all the previously mentioned functions are available for use in fstream. The example in Listing B shows how you could use fstream to swap the first and the tenth records in a file.

Listing B: Swapping records using fstream

```
Data record1;
Data record2;
// open the file in read and write mode
fstream iofile("temp3.dat",
  ios::binary | ios::in | ios::out);
// seek to the 10th record and read the record
iofile.seekg(9 * sizeof(Data));
iofile.read((char*)&record1, sizeof(Data));
// seek to the first record and read it
iofile.seekg(0);
iofile.read((char*)&record2, sizeof(Data));
// seek to the first record again and write
// the data read from the 10th record
iofile.seekg(0);
iofile.write((char*)&record1, sizeof(Data));
// go back to record #10 and write the data
// read from record #0
iofile.seekg(9 * sizeof(Data));
iofile.write((char*)&record2, sizeof(Data));
iofile.close();
```

Notice that you set the ios::in and ios::out flags in the constructor. You must do this to tell iostream that you'll be doing both read and write operations on the file. Aside from that, the code snippet in Listing B contains nothing new, other than the fact that you use the read() and write() functions together.

VCL's TFileStream

Now that you know how to read and write files using iostream, let's take a quick look at VCL's answer to file I/O: TFileStream. This section will be fairly short for two reasons: First, most of the concepts discussed earlier also apply to TFileStream. Second, TFileStream is less complicated (and slightly less capable) than iostream. Table D lists the primary properties and methods of TFileStream.

Table D: Important TFileStream properties and methods

Property	**Description**
Position	The current value of the file position indicator. Position is a read/write property.
Size	The current size of the file's data.
Method constructor	Opens a file in a specific mode (create, read, write, or read/write).
CopyFrom()	Copies a specified number of bytes from a stream to this stream.
Read()	Reads a specified number of bytes from the file to the specified memory location.
Write()	Writes a specified number of bytes from a memory location to the file.
Seek()	Moves the file position indicator by the specified amount either from the start of the file, the end of the file, or from the current position.

Right away, you should notice that TFileStream has much in common with the iostream classes. In particular, the Read() and Write() methods are functionally identical to the read() and write() methods of the iostream classes. The **Position** property of TFileStream simplifies seeking in a file and performs the same function as the ifstream methods tellg() and seekg(). You can read **Position** to determine the current file position, or you can write to **Position** to move the file position. Doing so is much easier and more intuitive than using tellg() and seekg() as you do in iostream operations.

One rather odd omission in TFileStream is the lack of an equivalent to the ifstream readline() method. As a result, TFileStream is less than ideal for reading text files on a line-by-line basis. This isn't a big problem, however, since many VCL components (TMemo, TListBox, TComboBox, TTreeView, and so on) have LoadFromFile() and SaveToFile() methods that make saving and reloading their contents frightfully easy.

You can use TFileStream as an input file, an output file, or both--you specify the mode and filename when you create a TFileStream object. The following example opens a file for reading:

```
TFileStream* fs =
  new TFileStream("names.dat", fmOpenRead);
```

The next example opens a TFileStream in read/write mode:

```
TFileStream* fs =
  new TFileStream("names.dat", fmOpenReadWrite);
```

In addition to the file mode, you can also specify the share mode. The share mode allows you to specify whether the file is opened for exclusive use or others are allowed to read and write the file while you have it open. For full details on the file open and share modes, see the VCL documentation for TFileStream. Earlier we showed an example of swapping two records in a file using the fstream class. Listing C shows that same example using TFileStream instead of fstream.

Listing C: Swapping files using TFileStream

```
TFileStream* fs =
  new TFileStream("names.dat",
  fmOpenReadWrite);
fs->Position = 998 * sizeof(Data);
fs->Read(&record1, sizeof(Data));
fs->Position = 0;
fs->Read(&record2, sizeof(Data));
fs->Position = 0;
fs->Write(&record1, sizeof(Data));
fs->Position = 998 * sizeof(Data);
fs->Write(&record2, sizeof(Data));
delete fs;
```

Note that the TFileStream destructor will close the file when you delete the TFileStream object. In fact, strange as it may seem, TFileStream doesn't provide methods for opening and closing files--the constructor and destructor take care of those chores. As you can see from the previous example, the basic concept is the same as with iostream, although you may find the TFileStream way of doing things a little cleaner.

One major difference between TFileStream and the iostream classes lies in error handling. TFileStream throws exceptions if something goes wrong (such as "file not found" or trying to seek past the end of a file), whereas the iostream classes leave error control up to the programmer. Here again, the TFileStream way of doing things is slightly superior.

Wrap up

Given these choices, what type of file access do you use? It comes down to personal preference. Both iostream and TFileStream have their strengths and weaknesses. As always, your choice of file access method might depend on the specific task at hand. While TFileStream is a little easier to understand, it lacks some of the power of iostream. We should point out that you could use TFileStream and iostream interchangeably. In other words, a file written with ofstream can be read by TFileStream and vice versa.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

March 1998

Today's buzzword: Streams

by Kent Reisdorph

Streams and *streaming* are programming buzzwords that sound more ominous than they really are. We won't try to provide an all-encompassing description of streams, but we'll give you a quick overview. Simply put, *streaming* is a mechanism that lets you read data, write data, and move around within the data. When we talk about streaming, we mean general input/output operations and not specifically file I/O. Put another way, streaming is about more than just disk files. Certainly, disk files are one type of stream--other types include memory streams (reading and writing data in memory), string streams (for manipulating string data), and console streams (reading and writing to a console window).

Perhaps one of the most important aspects of the streaming mechanism is the concept of the *stream position*. The stream position is a numerical value that indicates where the next bits of data will be read from (in the case of an input stream) or written to (in the case of an output stream). When you initially open a stream, the stream position is 0. If you read 10 bytes of data, then the stream position advances to 10. If you then read an additional 20 bytes of data, the stream position advances by 20 to contain the value 30. A similar update process happens when you write to a stream. Although the stream position is automatically updated, streaming classes still allow you to ask for the current stream position or to change the current stream position, thereby giving you full control over the contents of the stream.

Naturally, streaming classes have methods that let you read and write data. This is a basic need, and no streaming class would be complete without some way of accessing the data.

March 1998

Synchronizing two list boxes

by Matt Telles

The following question came up recently on one of the borland.public newsgroups: How do you synchronize two list boxes so that when one of them scrolls, the other will scroll as well? The responses ranged from simply trapping for list-box changes to overriding the owner-draw aspects of the list itself to watch for scrolling. I found it interesting that so many people would go to such lengths to solve what is really a pretty simple problem. A good solution was eventually posted, but by then I'd written my own. The essence of this synchronization question lies in understanding the Windows messaging system. You really can't program in Windows without knowing at least a little something about what's going on "under the hood." (For a detailed discussion on messages, see "Incorporate Custom Message-Handling in Your Applications" in the Premiere issue *of C++Builder Developer's Journal*.) However, with the advent of programming environments like C++Builder, Delphi, and even Visual Basic, programmers have been shielded from the horrors of "real" Windows programming. Since this list box problem would allow me to scratch the surface of the Windows programming model and expose some of the stuff underneath without writing an entire tome (something I've also done), I thought it would make a good article.

What's really going on in there?

When you click on the vertical scrollbar of a list box, a number of things happen. First, Windows catches the fact that you clicked the mouse in a window. Next, that window is queried to find out where the click happened. When the message is routed to the scrollbar, the scrollbar then informs the parent window (in this case, the list box) that the user has asked to scroll the list. It sends this information via a Windows message. Windows messages form the heart of the entire event-handling system that the VCL uses to "talk" to the outside world. Events are really just Windows message handlers that have been customized by the nice folks at Borland (or whoever wrote the component) so you can interface with them without knowing anything more about them. When you want to work with messages that aren't directly exposed, it takes a little more work.

Creating the synchronized list component

The first step in building a list box you can synchronize with another list box is to create a component that encapsulates the problem. To do this, you need to create a component based on the TCustomListBox.

Tip: Deriving from existing components

When you derive components from existing C++Builder components to extend functionality, always look for a TCustomxxx version of the component (where xxx is the name of the component you want to extend). All the basic functionality of the component will be exposed at this level without the overrides provided by levels farther down the food chain.

If you've never created a component from scratch, don't fret, it's easy! (For an overview of the process, see the three-part "Building Components" series in the September, October, and November 1997 issues of *C++Builder Developer's Journal*.)

First, select Component | New from the main IDE menu. You'll see a dialog box that asks for the name of your new component; a combo box lets you select the base class to extend. Enter *TSyncListBox* as the new component's name, then select TCustomListBox for the base class and click OK. The wizard will do the rest, generating the basic files needed for creating and registering the component in the C++Builder system.

Adding the message handler

Once you've generated the component skeleton, the next step is to create the specific code to extend this component. In this case, that means trapping for a specific Windows message and doing something when you get it. You're looking for the Windows message WM_VSCROLL, which is sent when the user clicks a control's scrollbar. The message's contents include the type of click (moving up or down a row, paging up or down, or dragging the scrollbar up or down) and the name of the control that should receive that data. All this information is contained within a Windows MSG structure. In the Borland Delphi/C++Builder world, the information is further contained within the TMessage class.

To handle a new component message, you need to add a message map to the component. Borland provides macros to make this task easy. In the component's header file, add the following lines in the protected section of the component definition:

```
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_VSCROLL, TMessage,
      HandleVScroll)
END_MESSAGE_MAP(TCustomListBox)
```

This block of code defines a new message entry for the class to handle the WM_VSCROLL message. The second item in the MESSAGE_HANDLER macro call (TMessage) indicates the type of message structure this method expects to receive. For some messages, you'll probably create your own custom structures based on TMessage, but in this case you'll use the plain TMessage structure. The macro's final argument is the name of the method you want called when the component receives the message. Once you've told the compiler which

method to call when the message is received, you must define that method. To do so, add the following line to the private section of the component's header file:

```
  virtual void __fastcall HandleVScroll(
    TMessage& msg );
```

You might wonder why the method is defined as private if the system is going to call it. The reason is simple: The method isn't called by the system at all! In fact, the internals of the object call this method. Here's how it works. When you use the BEGIN_MESSAGE_MAP macro in your header file, you're defining an inline method called Dispatch. C++Builder calls the Dispatch method for each message received by a component. If you don't define the Dispatch method, it's automatically handled for you by the TObject class, from which all components (and everything else) are derived.

If you look at the definition of the BEGIN_MESSAGE_MAP, MESSAGE_HANDLER, and END_MESSAGE_MAP methods, you'll see that the code generated for the items you just added to the component header file look like this:

```
virtual void __fastcall Dispatch(
  void *Message_)
{
  switch  (((PMessage)Message)->Msg)
    case  WM_VSCROLL:
       HandleVScroll(*((TMessage *)Message));
       break;
    default:
      TCustomListBox::Dispatch(Message);
      break;
  }
}
```

As you can see, when a WM_VSCROLL message is received, the switch statement determines which message it is. If it is, in fact, a WM_VSCROLL message, it's routed to the HandleVScroll method. Since Dispatch is a method within the class, it can safely call the private method for you. All that's left for you to do is to write the actual implementation of the class.

Implementing the scroll handler method

Let's look at the implementation of the scroll handler method first, then fill in the details

and missing pieces. Add the new method to your class implementation file (cpp) and enter the following code into the method:

```
void __fastcall TSyncListBox::HandleVScroll(
  TMessage& msg )
{
  // Let the default occur
  TCustomListBox::Dispatch( &msg );
  // Now, sync the two if necessary
  if ( pSyncBox )
    pSyncBox->TopIndex = TopIndex;
}
```

The actual processing is easy. First, you let the underlying Windows list box deal with the event as it normally would--which might cause the list box to scroll in either direction. If it does, you simply tell the list box that's synchronized with this one to move to the same position. You needn't worry about which way the list box scrolled or even *if* it scrolled! The TopIndex of the TListBox class allows you to control the index of the item that appears at the top of the list box display. All that's missing is the code that tells the list box to synchronize itself with another list. You need to add to the class a member variable representing the other list box and a method allowing the programmer to set this member variable. This code goes in the component's header file; Listing A contains the complete header file for the class.

Listing A: TSyncListBox.h

```
//---------------------------------------------
#ifndef TSyncListBoxH
#define TSyncListBoxH
//---------------------------------------------
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
#include <vcl\StdCtrls.hpp>
//---------------------------------------------
class TSyncListBox : public TCustomListBox
{
private:
  virtual void __fastcall HandleVScroll(
                          TMessage& msg );
  TCustomListBox *pSyncBox;
```

```
protected:
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_VSCROLL, TMessage,
                    HandleVScroll)
    END_MESSAGE_MAP(TCustomListBox)
public:
  __fastcall TSyncListBox(TComponent* Owner);
   void SetSyncListBox(
         TCustomListBox *pListBox )
   {
       pSyncBox = pListBox;
   }
__published:
};
//---------------------------------------------
#endif
```

Finally, you must initialize the list box pointer to NULL; otherwise, you'd have no way of knowing whether there's anything attached to this list with which to synchronize. Besides, when you have a pointer as a member variable, it's essential to initialize it before you use it. Since the class constructor is intended to handle all the initialization code, that's where the change goes:

```
__fastcall
  TSyncListBox::TSyncListBox(TComponent* Owner)
  : TCustomListBox(Owner)
{
  pSyncBox = NULL;
}
```

That's all there is to it! With a couple of lines of code, you've created a brand new component for use in your own applications.

March 1998

Quick ZDtips

Would you like to receive a weekly C++Builder tip in your E-mailbox, free of charge? Then, visit **www.zdtips.com,** where you can sign up for C++Builder tips, as well as tips covering a wide range of other applications. Here are some samples of the tips you'll receive each week.

Converting via the Watch List

by Kent Reisdorph

You can use the Watch List as a handy tool to convert decimal and hexadecimal numbers. Choose Run | Add Watch... from C++Builder's main menu to open the Watch Properties dialog box, then enter the number in the Expression field. If you're converting a hexadecimal number, click OK to see both the hexadecimal and decimal versions displayed in the Watch List window. If you're converting a decimal, click on the Hexadecimal radio button in the dialog box, then click OK.

Icons large and small

by Kent Reisdorph

In 32-bit Windows, an icon really consists of two icons: one large and one small. Both reside in the same ICO file. The large icon measures 32 by 32 pixels; Windows uses it for shortcuts, in dialog boxes, and in Explorer when you choose to view files as large icons. The small icon measures 16 by 16 pixels; Windows uses it on the taskbar, on application title bars, in the File | Open... dialog box, and in Explorer when you choose to view files as small icons. When you're creating your own icon, Windows doesn't require you to provide the small version. Instead, you can create only a large icon and Windows will shrink it in situations requiring a small icon. Be aware, however, that the shrunken icon's quality may not be high enough to meet your standards; it's often worth the extra effort to create both sizes.

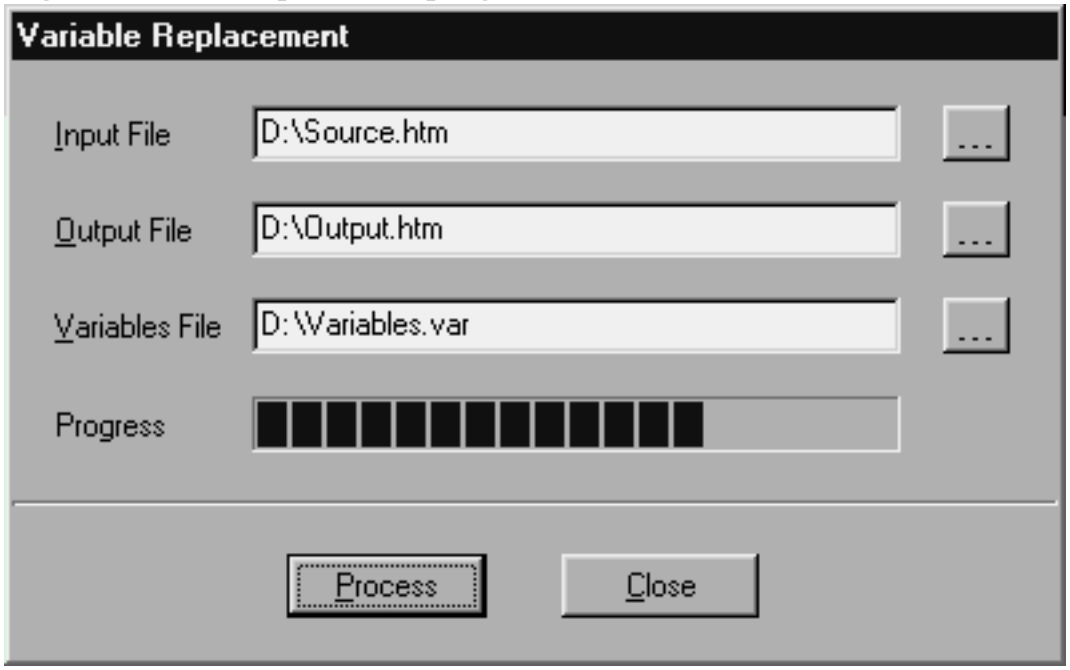
February 1998

Replacing variables in HTML documents

by Mark G. Wiseman

I recently helped construct a new Web site for my company. Early in the project, we realized we could reduce the time spent maintaining our Web site by automating some of the tasks. We noticed that, over time, the structure of many Web pages remained the same--only portions of the text and some of the graphics had to be changed. What we needed was a program that would search through an HTML file and replace variables with updated values. In this article, we'll create a Replace.exe program, shown in Figure A, to do just that.

Figure A: Our Replace.exe program looks like this in action.



Inputs and outputs

Listing A contains a simple HTML input file with the following four variables:

* <!--VAR:Site-->
* <!--VAR:Link-->
* <!--VAR:Saying-->
* <!--VAR:Today-->

We decided to wrap our variable names with the HTML comment delimiters <!-- and -->. By making the

variable names HTML comments, we can use HTML editors on our files. We also decided to begin each variable name with VAR: to make the variables easier to distinguish from actual HTML comments. Replace.exe doesn't require either of these conventions, though, so you can name your variables differently if you desire.

Listing A: HTML source file

```
<html>
<head>
<title>Really Great Links</title>
</head>

<body bgcolor="#FFFFFF">

<h1>Really Great Links</h1>

<hr>

<h3>The following links will take you to
  <!--VAR:Site--> and other great sites.</h3>

<p><!--VAR:Link--></p>

<p><a href="http://www.borland.com">Borland</a></p>

<hr>

<table border="0" width="100%">
   <tr>
      <td valign="bottom">Saying for the day:
        <!--VAR:Saying--></td>
      <td align="right" valign="bottom"><h5>
        Last updated: <!--VAR:Today--></h5></td>
   </tr>
</table>

<p align="left"> </p>
</body>
</html>
```

Replace.exe replaces the variable names using the values in the variable file shown in Listing B, then produces the HTML output file shown in Listing C. Note that variables don't have to be text--they can be links to other pages, such as <!--VAR:Link-->, links to graphics files, or even HTML formatting codes.

Listing B: Variables file

```
<!--VAR:Site-->=The Cobb Group
<!--VAR:Link-->=<a href="http://www.cobb.com/">
  The Cobb Group</a>
<!--VAR:Saying-->=A stitch in time saves nine.
<!--VAR:Today-->=February 1, 1998
```

Listing C: HTML output file

```
<html>
<head>
<title>Really Great Links</title>
</head>

<body bgcolor="#FFFFFF">

<h1>Really Great Links</h1>

<hr>

<h3>The following links will take you to
  The Cobb Group and other great sites.</h3>

<p><a href="http://www.cobb.com/">
  The Cobb Group</a></p>

<p><a href="http://www.borland.com">Borland</a></p>

<hr>

<table border="0" width="100%">
   <tr>
      <td valign="bottom">Saying for the day:
        A stitch in time saves nine.</td>
      <td align="right" valign="bottom"><h5>
        Last updated: February 1, 1998</h5></td>
   </tr>
</table>

<p align="left"> </p>
</body>
</html>
```

This example really shows off the power of C++Builder. By using two classes, AnsiString and TStringList, you can--in only a dozen or so lines of code--replace all the variables in an HTML file.

Tip: Tying strings together

If you aren't quite sure how to use those AnsiStrings, see the article "An AnsiString Class Reference" in the August 1997 issue of *C++Builder Developer's Journal*.

Here's how

Listings D and E contain all the code you need. One method of the TMainForm class, ProcessClick(), does nearly everything by itself.

Listing D: Replace header

```
#ifndef MainH
#define MainH

#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include <vcl\ComCtrls.hpp>
#include <vcl\ExtCtrls.hpp>
#include <vcl\Dialogs.hpp>

class TMainForm : public TForm
   {
   __published:
      TEdit *InputEdit;
      TEdit *OutputEdit;
      TEdit *VariablesEdit;
      TButton *InputButton;
      TLabel *Label;
      TButton *OutputButton;
      TButton *VariablesButton;
      TButton *ProcessButton;
      TButton *CloseButton;
      TProgressBar *ProgressBar;
      TBevel *Bevel;
      TOpenDialog *InputFileDialog;
```

```
      TSaveDialog *OutputFileDialog;
      TOpenDialog *VariablesFileDialog;

      void __fastcall CloseClick(TObject *Sender);
      void __fastcall ProcessClick(TObject *Sender);
      void __fastcall InputButtonClick(
        TObject *Sender);
      void __fastcall OutputButtonClick(
        TObject *Sender);
      void __fastcall VariablesButtonClick(
        TObject *Sender);

   public:
      __fastcall TMainForm(TComponent* Owner);
   };

extern TMainForm *MainForm;

#endif
```

Listing E: Replace source

```
#include <vcl\vcl.h>
#pragma hdrstop

#include "Main.h"

#include <memory>

using namespace std;

#pragma resource "*.dfm"
TMainForm *MainForm;

__fastcall TMainForm::TMainForm(TComponent* Owner) :
  TForm(Owner)
   {
   Caption = Application->Title;
   }

void __fastcall TMainForm::CloseClick(
  TObject *Sender)
   {
   Close();
   }
```

```
void __fastcall TMainForm::ProcessClick(
  TObject *Sender)
   {
   auto_ptr<TStringList> buffer(new TStringList);
   buffer->LoadFromFile(InputEdit->Text);
   String temp = buffer->Text;

   auto_ptr<TStringList> variables(new TStringList);
   variables->LoadFromFile(VariablesEdit->Text);

   for (int i = 0; i < variables->Count; i++)
      {
      String var = variables->Names[i];
      String val = variables->Values[var];

      int len = var.Length();
      int loc = temp.Pos(var);

      while (loc)
         {
         temp.Delete(loc, len);
         temp.Insert(val, loc);
         loc = temp.Pos(var);
         }

      ProgressBar->Position = (TProgressRange)((
        100 * i) / (variables->Count - 1));
      }

   buffer->Text = temp;
   buffer->SaveToFile(OutputEdit->Text);

   Application->MessageBox("Process Complete",
     Application->Title.c_str(), MB_OK);
   ProgressBar->Position = 0;
   }

void __fastcall TMainForm::InputButtonClick(
  TObject *Sender)
   {
   if (InputFileDialog->Execute()) InputEdit->Text =
     InputFileDialog->FileName;
   }
```

```
void __fastcall TMainForm::OutputButtonClick(
  TObject *Sender)
   {
   if (OutputFileDialog->Execute()) OutputEdit->
     Text = OutputFileDialog->FileName;
   }

void __fastcall TMainForm::VariablesButtonClick(TObject *Sender)
   {
   if (VariablesFileDialog->Execute())
     VariablesEdit->Text =
     VariablesFileDialog->FileName;
   }
```

First you create buffer--a TStringList--and use it to load the input file specified in the input edit control. You then copy all the text in buffer into an AnsiString string named temp.

Next, you load the variable file into variables, another TStringList. Each line of text in variables represents a variable name-value pair. The pair must be a single line of text, and the name must be separated from the value by an equal sign, like this:

```
name=value
```

You don't have to parse variables, because TStringList will automatically separate the variable name and its value when you use the properties Names and Values. You cycle through each name-value pair in variables using a for loop. Within the for loop, you use a while loop and the AnsiString methods Pos(), Delete(), and Insert() to replace each occurrence of a variable name in temp with its value. Finally, you copy temp back into buffer so you can use the TStringList method SaveToFile() to write your output file to disk.

To take advantage of the Windows dialog boxes for opening and saving files, you add three very short methods: InputButtonClick(), OutputButtonClick(), and VariablesButtonClick(). Well, their names may not be short, but the code in each method is only one line. These methods are assigned to the buttons next to the input, output, and variable file edit controls. To be really clever, you assign these methods to the respective OnDblClick events for the edit controls and their labels.

That's it. Amazing! C++ Builder and the VCL do everything else for you. The VCL even takes care of error handling--try entering a blank or bad filename and see what happens.

After replacement

While looking at the code for Replace.exe, you might think that this program will work on any text file, not just HTML. Well, you're right.

Without any changes, the code in this article compiles to a very useful program. However, you might want to modify Replace.exe to accept command-line arguments for the input, output, and variable filenames. Doing so would allow you to process files in batch mode. If you make this modification, you should probably let the user interrupt the replacement process before completion.

If you'd like to see how my company has used Replace.exe, check out the Corporate Art Gallery at

www.cosolutions.com

The entire gallery is periodically generated at random from a larger group of image files. Visitors to our gallery get to see different pictures, and we don't have to edit a single line of HTML to change them.

Our Corporate Art Gallery

Northwest C++ User's Group

Last Updated: 04/06/2002

000624

Corner Office Solutions, Inc.

Corner Office Solutions, Inc. develops business software for customers and clients in the United States and Canada. We specialize in PC-based software for Microsoft Windows 9x, Windows ME and Windows NT/2000/XP. We offer both prepackaged and custom software solutions for management and human resources.

One of our products, the *Executive Compensation Profile* (ECP) is a software-based, total compensation statement. ECP helps executives choose the best combination of offerings from their company's compensation and benefits plans. ECP allows executives to vary assumptions affecting base compensation, short-term and long-term incentive plans, stock options, life and health insurance, retirement plans, and other perquisites. ECP then shows the executives how these changes will affect their total compensation from the present until retirement.

We also design and maintain Web sites for a number of our clients.

February 1998

- [Replacing variables in HTML documents]
- [Creating a font toolbar]
- [TQuery with parameters]
- [DB->dilemmas]

February 1998

Creating a font toolbar

by Kent Reisdorph

You've seen font-selection toolbars in many Windows programs. These toolbars let you select a font, the font size, and attributes such as bold, italic, or underline. A font toolbar is a great feature for any application that uses fonts. In this article, we'll demonstrate how to set up a font toolbar. In particular, we'll show you how to use the API function EnumFontFamilesEx() to fill a combo box with a list of available fonts.

The feature set

The requirements of a font toolbar could vary widely from user to user and from application to application. Regardless of the application, though, a font toolbar should have a few basic elements:

- A font selection combo box
- A font point-size combo box
- A bold button
- An italic button
- An underline button

A font toolbar also might have buttons for setting text alignment to left, right, or centered. Naturally, the toolbar could include other application-specific items as well. The font selection combo box contains a list of installed fonts. This combo box should have the csDropDownList style so users can pick a font name only from the list, and its **Sorted** property should be true so the list of fonts is alphabetical. (The font name is technically called the *typeface*, but most folks use the term *font*.) The question then becomes one of which fonts to display. You can show raster fonts, device fonts (ATM fonts, for example), or TrueType fonts.

The font size selection combo box is pretty basic: You simply load it with the font sizes you'd like to support. Although you'll see all sorts of point sizes listed in different Windows applications, showing the even numbers from 8 to 40 seems to be a sensible approach. This combo box should have the csDropDown style so users can enter any font size in addition to selecting a size from the list. It should also include code that changes the font size when the user presses [Enter].

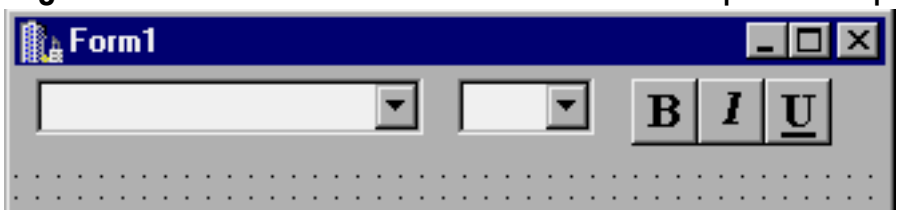
The font attribute buttons (bold, italic, and underline) should be shown in either an up state or a down state. For example, if the bold style is in effect, then the bold button should appear in the down state. The buttons should operate independently, so that any font attribute can be on or off.

Building the font bar

The actual task of building the font bar is pretty trivial. Place a panel on a form, then change its **Height** property to 32, its **Align** property to alTop, and its **Caption** property to an empty string. I prefer a font bar with a flat look, so I also set the **BevelOuter** property to bvNone. Next, place two combo boxes on the form, to display the font name and the font size. Naturally, the combo box for the font name should be wide enough to display the font name in its entirety (a width of 150 should do). The font size combo box can be fairly narrow, since it has to show only three digits. Assign the combo boxes meaningful **Name** properties--for example, FontCombo and FontSizeCombo.

The next step is to add the buttons for selecting font attributes. Place three SpeedButton components on the panel to the right of the font size combo box. Change the captions as required, or create bitmaps for the buttons. In order for the buttons to toggle, you must set their **AllowAllUp** property to true. Then, set each button's **GroupIndex** property to a unique value: 1 for the bold button, 2 for the italic button, and 3 for the underline button, for example. Your form should now resemble the one shown in Figure A. Now you can move on to writing the code that will make the font toolbar useful.

Figure A: The font toolbar looks like this up to this point.



Populating the combo boxes

The biggest chore you'll face is filling the font name combo box with the names of the fonts currently installed on the user's system. To do so, you'll use the EnumFontFamiliesEx()API function, which enumerates all the fonts installed in Windows. EnumFontFamiliesEx() makes use of a callback function--we'll return to it shortly. First, let's look at how to call EnumFontFamiliesEx().

EnumFontFamiliesEx()

Your combo box should contain the font names when the application starts, so you'll fill the box in the form's OnCreate event handler. Before calling EnumFontFamiliesEx(), you must create a LOGFONT structure and set a few parameters that control which fonts are enumerated, as follows:

```
LOGFONT lf;
lf.lfCharSet = DEFAULT_CHARSET;
strcpy(lf.lfFaceName, "");
lf.lfPitchAndFamily = 0;
```

The lfCharSet member controls the character set whose fonts will be enumerated. In this case, you want all fonts in the default Windows character set. Next, you set the lfFaceName member to an empty string, telling Windows that you want a list of all fonts in the specified character set. If you set lfFaceName to a specific typeface name (Arial, for example) then only the fonts in that font family will be enumerated. Finally, you must set the lfPitchAndFamily member to 0 (except in a couple of special cases we won't go into here). Having done all of that, you can call EnumFontFamiliesEx() to have Windows enumerate all the fonts:

```
EnumFontFamiliesEx(Canvas->Handle,
  &lf, (WNDENUMPROC)FontEnumProc, 0, 0);
```

The first parameter takes a handle to a device context; here you use the canvas's **Handle** property. The second parameter takes a pointer to the LOGFONT structure you just created. The third parameter takes a pointer to the callback function, which will be called once for each font in the system. The cast to WNDENUMPROC is necessary to get the code to compile (Windows C programmers love that kind of thing). The fourth parameter can be used for any application-specific data you want to pass to the callback function (we won't use it in our case), and the final parameter is reserved and should always be 0. When you call EnumFontFamiliesEx(), Windows begins enumerating the fonts. The callback function is called once for each installed font. Let's take a look at the callback function, since that's where all the action takes place.

The callback function

As we've said, the callback function is called once for each font enumerated. The callback function gives you, among other things, the name of the font for which the function has been called. It's up to you to do something with the font names--in this case, you'll populate the font name combo box. The callback function's declaration looks like this:

```
int CALLBACK EnumFontFamProc(
  ENUMLOGFONTEX *lpelfe,
  NEWTEXTMETRICEX *lpntme,
  long FontType,
  LPARAM lParam
);
```

As you can see, the function gives you four parameters. The lpelfe parameter is a pointer to a function that contains information about the font (more on that in just a bit). The lpntme parameter is a pointer to a structure that gives you the text metrics for the font; the FontType parameter tells you whether the font is a raster font, a device font, or a TrueType font; and the lParam parameter holds any user-defined data you want to pass to the function. The only parameter we're concerned with in this case is the lpelfe parameter--a pointer to an instance of the ENUMLOGFONTEX structure. (You could also make use of the FontType parameter if, for example, you wanted to limit the list to TrueType fonts.) This structure contains a data member called elfFullName, which contains the name of the font. You can add that text string directly to the combo box using the Add() method, as follows:

```
int CALLBACK FontEnumProc(
  ENUMLOGFONTEX *lpelfe,
  NEWTEXTMETRICEX *lpntme, long FontType,
  LPARAM lParam)
{
  Form1->FontCombo->Items->Add(
    (char*)lpelfe->elfFullName);
  return 1;
}
```

This code illustrates how to get the font name and add it to the combo box. In a real-world application, you'll need to have some method of ensuring that the combo box doesn't contain duplicate strings. You'll get duplicate strings because Windows will execute the callback function for each font style in a particular font family. For example the Arial font contains the typeface names Arial Bold, Arial Italic, Arial Bold Italic, and so on. The elfFullName member of the ENUMLOGFONTEX structure provides the font name itself, and the elfStyle member contains the style. If desired, you could build the font list by combining the elfFullName member with the elfStyle member.

You might think to set the combo box's **Sorted** property to true and the **Duplicates** property to false, thereby preventing duplicate strings in the combo box. For some odd reason, TComboBox::Items is of type TStrings rather than of type TStringList (as it is with list boxes), so there's no **Duplicates** property. Oh, well, I guess we'll have to do it the hard way. We won't show the code here, but see Listing B for an example of how to eliminate duplicate strings. After enumerating all of the fonts, EnumFontFamiliesEx() returns and the combo box contains a list of available fonts.

Filling the font size combo box

Filling the font size combo box is an easy task. A simple loop will do:

```
for (int i=8;i<40;i+=2) {
  FontSizeCombo->Items->Add(i);
}
```

This code puts font sizes from 8 to 40 in the font size combo box, showing only even font sizes. Some programs use a rather odd algorithm for displaying font sizes in their font size selection combo boxes. You can follow those conventions or you can just go the easy route and use our code.

Handling clicks on the font bar

There's really only one thing left to do to have a functioning font bar: You need to respond to clicks on the various font bar components. Obviously, this sample application must have a component that uses fonts, so you'll use a Memo component. The font bar will control the memo's font, font size, and font style. The good news is that a single event handler will handle clicks on any font bar component. First, select all the components on the font bar (two combo boxes and three speed buttons). Now, switch to the Events tab in the Object Inspector, type *FontBarClick* to the right of the OnClick event, and press [Enter]. C++Builder will create an event handler for you, and you can begin typing code.

Since you don't care which component was clicked, you'll set all the font attributes each time the OnClick handler is called. The main body of the OnClick handler contains this code:

```
Memo->Font->Name = FontCombo->Text;
Memo->Font->Size =
  FontSizeCombo->Text.ToIntDef(10);
TFontStyles styles;
if (BoldBtn->Down) styles << fsBold;
if (ItalicBtn->Down) styles << fsItalic;
if (UnderlineBtn->Down) styles << fsUnderline;
Memo->Font->Style = styles;
```

You extract the font name from the font combo box and assign it to the Memo font's **Name** property. Since Windows gave you the font names, you know that any font the user selects is valid. Next, the code sets the font's **Size** property to the value contained in the font size combo box. As you can see, the AnsiString class's ToIntDef() method converts the text in the combo box to an integer. Remember, the user may type a font size directly rather than choosing a font size from the dropdown list. The ToIntDef() method ensures that if the user types invalid characters, VCL will supply a default value (10) rather than throwing an exception.

Finally, you read the states of the three font style buttons and build a TFontStyles set based on which buttons are down. After that, you assign the results of the set to the **Style** property of the Memo's font.

All finished?

Listings A and B contain the code for a program that demonstrates the font toolbar, as shown in Figure B. (You can download our sample files from **www.cobb.com/cpb**, as part of the file feb98.zip.)

Listing A: FONTBARU.H

```
//---------------------------------------------
#ifndef FontBarUH
#define FontBarUH
/---------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include <vcl\ExtCtrls.hpp>
#include <vcl\Buttons.hpp>
#include <vcl\ComCtrls.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
  TPanel *Panel1;
  TComboBox *FontCombo;
  TComboBox *FontSizeCombo;
  TSpeedButton *BoldBtn;
  TSpeedButton *ItalicBtn;
  TSpeedButton *UnderlineBtn;
  TRichEdit *RichEdit1;
  TMemo *Memo;
  void __fastcall FormCreate(TObject *Sender);
  void __fastcall FontBarClick(TObject *Sender);


  void __fastcall FontSizeComboKeyPress
    (TObject *Sender, char &Key);
private:        // User declarations
```

```
public:         // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------
extern TForm1 *Form1;
```

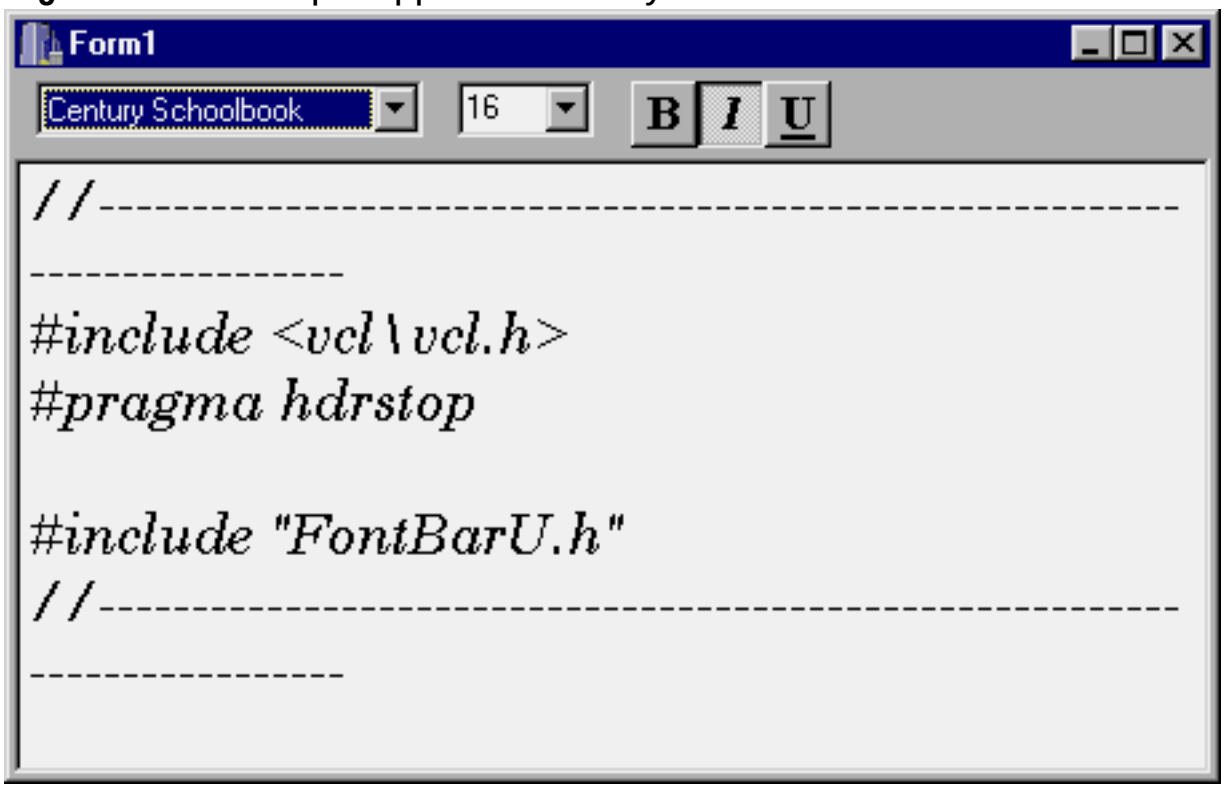
Listing B: FONTBARU.CPP

```
//---------------------------------
        ------------#include <vcl\vcl.h>
#pragma hdrstop

#include "FontBarU.h"
//-------------------------------------
        ----------#pragma resource "*.dfm"
TForm1 *Form1;

int CALLBACK FontEnumProc(ENUMLOGFONTEX *lpelfe,
  NEWTEXTMETRICEX *lpntme, long FontType,
    LPARAM lParam)
{
  // static String object to hold text of last font
  static String S;
  // If the string is different, then save it for
  // the next time and add it to the combo box.
  if (S != String((char*)lpelfe->elfFullName)) {
    S = (char*)lpelfe->elfFullName;
    Form1->FontCombo->Items->Add(S);
  }
  // Return a non-zero value to keep iterating.
  return 1;
}
//-------------------------------------
        -------__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//-------------------------------------
        --------void __fastcall TForm1::FormCreate
        (TObject *Sender)
{
  // Set up the logfont structure with the
  // values required to enumerate all fonts.
```

```
  LOGFONT lf;
  lf.lfCharSet = DEFAULT_CHARSET;
  strcpy(lf.lfFaceName, "");
  lf.lfPitchAndFamily = 0;
  // Start enumerating, passing the address
  // of the callback function.
  EnumFontFamiliesEx(Canvas->Handle,
    &lf, (WNDENUMPROC)FontEnumProc, 0, 0);
  // Fill in the font size combo box.
  for (int i=8;i<40;i+=2)
    FontSizeCombo->Items->Add(i);
  // Select the "10" item in the combo box
  // since the initial point size is 10.
  FontSizeCombo->ItemIndex =
    FontSizeCombo->Items->IndexOf("10");
  // Select the Courier New font so the font
  // combo box shows the current font when
  // initially displayed.
  FontCombo->ItemIndex =
    FontCombo->Items->IndexOf("Courier New");
  // Set the initial font and font size, and
  // load this program's source into the memo.
  Memo->Font->Name = "Courier New";
  Memo->Font->Size = 10;
  Memo->Lines->LoadFromFile("FontBarU.cpp");
}
//-------------------------------------
        ---------void __fastcall
TForm1::FontBarClick(TObject *Sender)
{
  // Set the Font's Name property to the font
  // selected in the combo box.
  Memo->Font->Name = FontCombo->Text;
  // Set the font Size to the value in the
  // font size combo box.
  Memo->Font->Size =
    FontSizeCombo->Text.ToIntDef(10);
  // Set the Style as needed based on which
  // of the style buttons are down.
  TFontStyles styles;
  if (BoldBtn->Down) styles << fsBold;
  if (ItalicBtn->Down) styles << fsItalic;
  if (UnderlineBtn->Down) styles << fsUnderline;
```

```
  Memo->Font->Style = styles;
}
//---------------------------------------------
void __fastcall TForm1::FontSizeComboKeyPress
  (TObject *Sender, char &Key)
{
  // If the enter key on the font size combo
  // box was pressed then swallow the keystroke
  // and call FontBarClick() to update the
  // font in the memo component.
  if (Key == VK_RETURN) {
    Key = 0;
    FontBarClick(0);
    Memo->SetFocus();
  }
}
```

Figure B: Our sample application lets you control the Memo control's font characteristics.



You'll see that we've added a few niceties to make the toolbar behave properly. For example, the size value changes when the user presses [Enter] in the font size combo box. It's natural to press [Enter] after typing a new value, but this action will result in an annoying beep and nothing more unless you write code to account for that possibility. As you can see in Listing B, the code simply responds to the OnKeyPress event and converts the [Enter] keypress into a 0, eliminating the beep. After that, you call the FontBarClick() method to update the font.

Now that you have the basic idea, it will be relatively simple to add text-alignment buttons next to the font-style buttons. You could add buttons for left-justified, centered, or right-justified text. You'll have to change the Memo component to a RichEdit component, but for the most part it's a trivial exercise. One hint: Change the **GroupIndex** value for the alignment buttons to the same value. Doing so will ensure that only one of the buttons can be down at a time. Also, provide OnClick handlers for each of the buttons. For example, the handler for the Center button would contain just one line of code:

```
RichEdit->Paragraph->Alignment = taCenter;
```

Conclusion

Adding a font selection toolbar is easy once you know how to obtain a list of fonts from Windows. Now, your applications can have a polished look that's common to many Windows programs.

February 1998

TQuery with parameters

by Gerry Myers

As part of my current project, I needed to display records from a database table. My immediate choice was to use a grid (TDBGrid) linked to the database table (TTable)--simple enough and straightforward. I could easily drop the grid, table, and data source components onto my form, link everything at design time, and be finished in a matter of minutes.

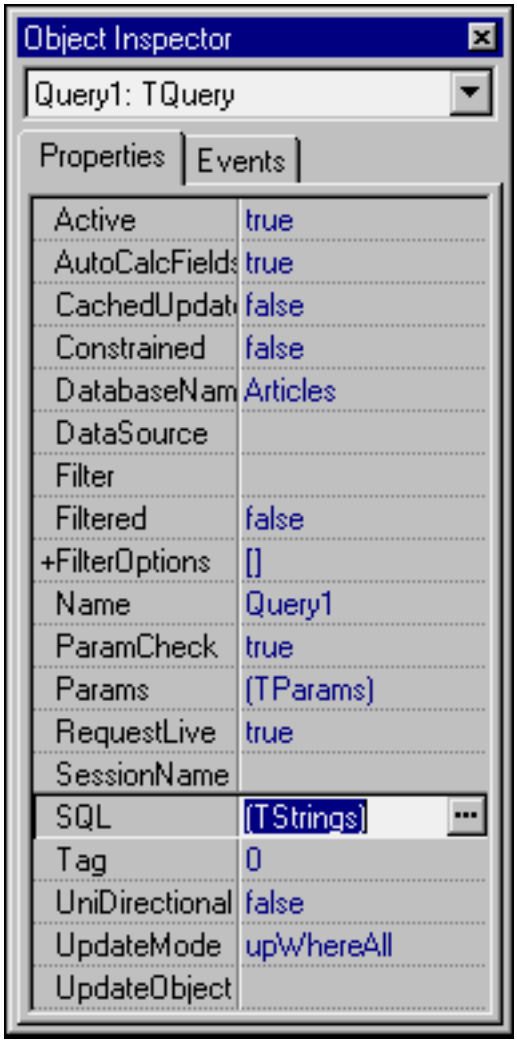
However, after some thought I realized that the users would probably (later, of course) want some mechanism to view a subset of the data. As you probably know if you've used the TTable/TDBGrid combination, unless you specify a master/slave relationship or some kind of filter, every record in the table is displayed. In this case, I didn't have another table to serve as the master, and I didn't want to use a filter. Instead of the ever-familiar TTable, I decided to use a TQuery component to connect to the underlying database table. The main advantage for my application was that TQuery would let me specify parameters (variables) in the query's SQL statement that I could set during runtime.

If you've been using TTables for all your database connections, you may want to consider TQuery instead on your next project. The flexibility of the SQL alone makes it a worthwhile tool--but with the addition of parameters, the TQuery becomes an extremely powerful database component. In this article, we'll discuss TQuery parameters and how to set them in your code.

TQuery and SQL

The heart of the TQuery component is the **SQL** property, shown in Figure A. The SQL is the "rule" that the query uses to determine which fields and records to pull out of which database tables. It also can specify constraints (for instance, ORDER BY) and can even contain subqueries that determine how the records will be presented.

Figure A: The TQuery component has these properties.



Listing A shows a simple SQL statement. It selects all fields (columns) from the underlying database table (Measure.db), but only those records (rows) that have the string *DegF* in the Units field. You'd use such a statement if the user wanted to see only records that deal with temperature in degrees Fahrenheit.

Listing A: A simple SQL statement

```
SELECT * FROM Measure.db
WHERE Units = "DegF"
```

As I mentioned earlier, you could perform a similar task with a TTable by having another table serve as the master. You'd link the two tables to form a master/slave relationship tied through the Units field. When the user selected a record in the master, that record's Units field would automatically be used to pull out only those records in the slave's table with matching Units values.

That technique requires another table and depends on the user to select the master's records (although it's possible to select the master's records through code). There's a much more straightforward approach that requires fewer components and links. It makes use of the

TQuery component instead of the TTable and also introduces parameters into the SQL statement.

Parameters

A parameter is a variable in the SQL statement that you can fill at design time or runtime. There are no structures to declare or fill out and no additional program variables to worry about--the TQuery component automatically recognizes the parameter and allows you to set its value. The parameter is generally the value of a database field. For example, the SQL statement shown in Listing B is similar to the one in Listing A, but it uses a parameter instead of the hard-coded DegF Units value. However, SQL parameters aren't limited to the WHERE portion of the query--you can specify which fields to SELECT, the sort order (ORDER BY), the database from which to pull the records, and much more. You also can easily use two or more parameters in the same SQL statement to allow greater flexibility. Note that a parameter must begin with a semicolon (:) and end with the name of the field containing the value.

Listing B: A simple SQL statement with a parameter

```
SELECT * FROM Measure.db
WHERE Units = :Units
```

Where exactly does the :Units parameter get its value? You set its value in your code with just a few simple lines. The link between your TQuery component and the underlying table is made for you automatically when you specify both the TQuery's **Database** property and the table name in the SQL statement's FROM section.

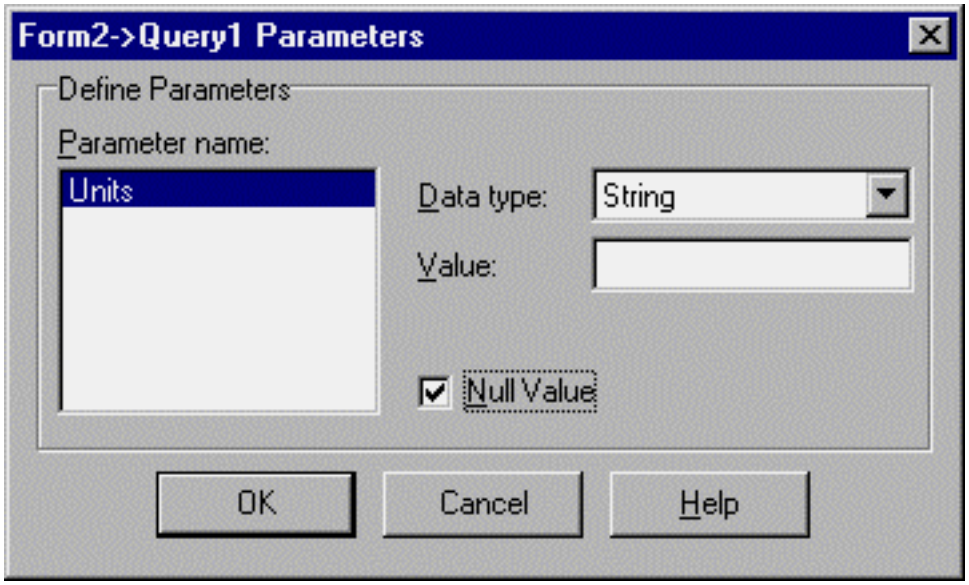
As you can see in Figure A, my example database is called Articles; the Measure.db table in that database will supply the data. Measure.db has the following fields: MeasId, Description, Units, Chassis, Slot, Channel, and Reading. The Units field will be searched for a match to the parameter value. Only those records in Measure.db whose Units value matches the :Units value will be displayed in the grid.

Defining parameters at design time

The easiest way to use parameters with a TQuery is to define the SQL statement and parameters at design time. Double-click the TQuery's **SQL** property in the Object Inspector to open the standard String List Editor window; you can then enter your SQL statement directly. Defining the SQL statement that contains a parameter is only half the task; you must also define the actual parameter. You do so by double-clicking the TQuery's Params property to open the Parameters dialog box shown in Figure B. The Parameter Name is displayed

automatically, since you entered it (:Units) in the SQL statement. The TQuery knows that it has a parameter in its SQL statement, but it needs to be told the parameter's data type and whether the parameter has an initial value. In this case, the Units field is of String (ASCII) type, and its initial value is NULL.

Figure B: You define a parameter in the TQuery's Parameters dialog box.



Defining parameters at runtime

You can also define the SQL statement and its parameters during runtime. When you drop the TQuery component on the form, you could choose to ignore the **SQL** and **Params** properties and let your code set them. In order to run the program without an SQL statement in the TQuery, you must leave the TQuery's **Active** property set to false. As a matter of fact, if you try to set it to true during design-time (without an SQL statement), you'll get an error message. When you've gathered all the information you need to construct your SQL statement, you'll build the query using code like that found in Listing C. The first line makes sure the TQuery component is closed (Active=false). This is just good practice, since messing with the **SQL** property on an open query will generate a program error.

Listing C: Building an SQL statement with a parameter at runtime

```
Query1->Close();
Query1->UnPrepare();
Query1->SQL->Clear();
Query1->SQL->Add
(
  "SELECT * FROM Measure.db
   WHERE Units = :Units"
);
```

The next line "unprepares" the query. Before a query is executed, it's best to "prepare" the underlying database. Doing so lets the database know ahead of time that a query is coming, so it can optimize the way it will handle the query. You need to prepare the query only the first time you open it, as long as the query doesn't change. Therefore, before you make any changes, it's a good idea to unprepare the query so the underlying database can release any resources that it set aside for the query's repeated execution.

Next in the example code comes the actual **SQL** property, which is nothing more than a TStrings object. To ensure that any previous query is erased, you clear the **SQL** value. You then write the new query to the **SQL** property using TStrings' Add() method.

You may have noticed that you haven't yet opened (activated) the query--at this point, it still lacks one thing. The SQL statement contains a reference to a parameter called :Units. But so far, you haven't set the value of the parameter--and you don't have to until the query is opened.

Setting the Params property

The beauty of the TQuery is that you can set the **SQL** property once, then change the parameter over and over. In the previous code example's SQL, simply changing the :Units parameter will return different records from the database. Just as you can set the **SQL** property during design or at runtime, you also can set the value of the parameter at either time. Look at the Tquery's Parameters dialog box in Figure B; notice the Value field in which you can enter the parameter's initial value. For example, if you wanted all the temperature measurements to be presented to the user on program startup, you could set the initial parameter value to DegF.

Actually, if you wanted to check the database before running the program, you could make an entry in the dialog box's Value field and set the TQuery's **Active** property to true. Doing so would cause the appropriate records to be pulled from the database (according to the SQL statement) and displayed in my design-time grid. To see records with other Units values at design-time, you'd deactivate the TQuery (Active=false), change the contents of the Parameters dialog box's Value field, and reactivate the query (Active=true).

Setting Params at runtime

You can also set the value of the **Params** property at runtime. The code snippet shown in Listing D does just that.

Listing D: Setting the Params property at runtime

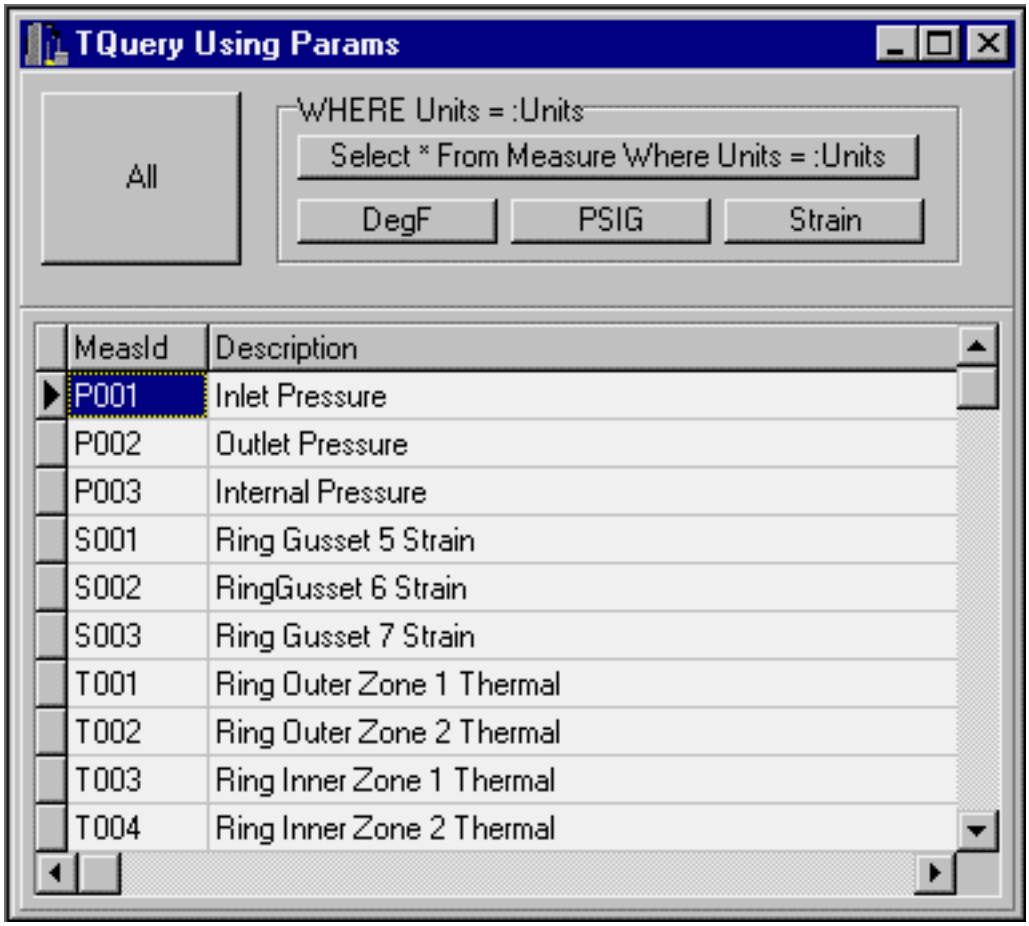
```
Query1->Close();
Query1->UnPrepare();
Query1->Params->ParamByName
  ( "Units" )->AsString = "DegF";
Query1->Prepare();
Query1->Open();
```

Again, it's a good practice to make sure the query is closed and unprepared before making changes to properties. Next, the code searches for the current parameter ("Units") using the function ParamByName(), since there can be multiple parameters. The ParamByName() function exposes a TString property that you can set to the desired value ("DegF"). After the parameter is set, the code prepares the query to allow the underlying database to do any optimizations. Finally, the last line activates (opens) the query. The records will now be pulled from the database and displayed in your grid. Whenever the user wants a different type of measurement, you simply re-execute this code with the appropriate new value in place of "DegF".

Putting it together

My application form, shown in Figure C, requires the user to select groups of records simply by clicking different buttons.

Figure C: Our sample interface lets the user select records by clicking buttons on this form.



Depending on the groups of buttons selected, the application sets two SQL statements. If the user clicks the All button, the application uses the code from Listing E to display all records. As you can see, this SQL statement doesn't use a parameter.

Listing E: Displaying all records at runtime

```
Query1->Close();
Query1->UnPrepare();
Query1->SQL->Clear();
Query1->SQL->Add
  ( "SELECT * FROM Measure.db" );
Query1->Prepare();
Query1->Open();
```

You'll notice in Figure C that the buttons other than All are grouped together. These grouped buttons use the SQL statement from Listing C. This query depends on a parameter that must be set before the TQuery can be opened. The user must click the Select *... button first, to establish the SQL statement and create the parameter. After that, the user can click the DegF, PSIG, and Strain buttons to set the :Units parameter value. Once the parameter value is set, the application can activate (open) the query and display the records. For example, when the user clicks the DegF button, the application runs the event handler shown in Listing

F.

Listing F: Setting the parameter and opening the query

```
// If the current query doesn't use any
// parameters, let the user know.
if ( !Query1->ParamCount )
{
  Application->MessageBox(
    "Set SQL statement first.",
    "SQL Info", MB_OK |
    MB_ICONEXCLAMATION );
  return;
}

// Setup and execute the hard-coded Param value.
Query1->Close();
Query1->UnPrepare();
Query1->Params->ParamByName
  ( "Units" )->AsString = "DegF";
Query1->Prepare();
Query1->Open();
```

The code first checks to be sure the correct SQL statement has been loaded, in case the All button's query is currently selected. Since that query doesn't take any parameters, trying to set a parameter would produce a runtime error. The rest of the code should look familiar from our discussion of Listing D. At this point, the grid will display only those records in which the Units field contains the value DegF.

Conclusion

Once you understand the SQL, the hardest part of using the TQuery component with a parameter is working through the closing, unpreparing, preparing, and opening sequence in the correct order, to avoid a runtime error. Whether you set up the TQuery at design-time or runtime, if you follow the simple examples we've presented in this article, your queries will be successful. Developers with even a limited knowledge of SQL will find the TQuery component and its parameters to be valuable tools in the database toolkit.

January 1998

Persistence pays off

by Mark G. Wiseman

In the December 1997 article "Starting Your Application Minimized," we showed you how to start your application with the main form minimized, maximized, or in its normal state. This month, we'll demonstrate how to make the state, position, and size of your application's main form persistent. To make the form persistent, we'll make a few modifications to the TMinMaxForm class we created last time, and we'll add a new class: WWindowPlacement.

A helpful helper class

WWindowPlacement, shown in Listings A and B, is a simple wrapper class around the Windows API structure WINDOWPLACEMENT. WWindowPlacement has two methods, Get() and Set(). These methods are based on the Windows API functions GetWindowPlacement() and SetWindowPlacement(). WWindowPlacement also overloads the I/O stream insertion and extraction operators, << and >>. These operators are what make the class useful. Using <<, you can write a form's state, position, and size to a file or to memory (by using strstream). Then, you can use the >> operator to read this information back when you need it.

Listing A: WWindowPlacement header

```
#ifndef winplace_h
#define winplace_h

#include <vcl\vcl.h>
#pragma hdrstop

class WWindowPlacement : public WINDOWPLACEMENT
   {
   friend  ostream & operator <<
      (ostream & os, const
      WWindowPlacement & placement);
   friend  istream & operator >>
      (istream & is,
      WWindowPlacement & winplace);

   public:
      WWindowPlacement(TForm *form = 0);
      virtual ~WWindowPlacement() {}
```

```
      void Get(TForm *form);
      void Set(TForm *form);
   };
```

Listing B: WWindowPlacement source

```
#include <vcl\vcl.h>
#pragma hdrstop

#include "winplace.h"

WWindowPlacement::WWindowPlacement
   (TForm *form)
   {
   length = sizeof(WINDOWPLACEMENT);

   if (form == 0)
      {
      if (::GeWWindowPlacement
         (::GetDesktopWindow(), this)
         == FALSE)
         {
         ptMinPosition.x = ptMinPosition.y = -1;
         ptMaxPosition.x = ptMaxPosition.y = -1;
         }

      flags = 0;
      showCmd = SW_SHOWNORMAL;
      rcNormalPosition.left=rcNormalPosition.top = 0;
      rcNormalPosition.right = 639;
      rcNormalPosition.bottom = 479;
      }
   else
      Get(form);
   }

void WWindowPlacement::Get
   (TForm *form)
   {
   ::GeWWindowPlacement(form->Handle, this);

   // If no parent, then this window is on the
   // desktop and when minimized we should let
```

```
   // Windows place the icon
   if (form->ParentWindow == 0)
      flags &= ~WPF_SETMINPOSITION;
   }

void WWindowPlacement::Set
   (TForm *form)
   {
   ::SeWWindowPlacement(form->Handle, this);
   }

ostream & operator <<(ostream &os,
   const WWindowPlacement &placement)
   {
   char sep = ',';

   os << '[' << placement.flags << sep
      << placement.showCmd << sep;
   os << placement.ptMinPosition.x << sep
      << placement.ptMinPosition.y << sep;
   os << placement.ptMaxPosition.x << sep
      << placement.ptMaxPosition.y << sep;
   os << placement.rcNormalPosition.left << sep
      << placement.rcNormalPosition.top << sep;
   os << placement.rcNormalPosition.right << sep
      << placement.rcNormalPosition.bottom << ']';

   return(os);
   }

istream & operator >>(istream &is,
   WWindowPlacement &placement)
   {
   char sep;

   is >> sep >> placement.flags >> sep
      >> placement.showCmd >> sep;
   is >> placement.ptMinPosition.x >> sep
      >> placement.ptMinPosition.y >> sep;
   is >> placement.ptMaxPosition.x >> sep
      >> placement.ptMaxPosition.y >> sep;
   is >> placement.rcNormalPosition.left >> sep
      >> placement.rcNormalPosition.top >> sep;
   is >> placement.rcNormalPosition.right >> sep
```

```
      >> placement.rcNormalPosition.bottom >> sep;

   return(is);
   }
```

Safe assumption

You may have noticed that when a WWindowPlacement object is constructed without a pointer to a form, the object is initialized to a location of (0, 0) and a size of 640 by 480. You can assume that the users of your application will use a screen resolution of at least 640 by 480--however, you can't safely assume they'll use a higher resolution. When a user first runs your application, the application won't have saved its position and size yet; by defaulting to 640 by 480, it should fit the user's screen.

The main form

To make use of the WWindowPlacement class, you need to make a few changes to the TMinMaxForm class we created last month. Listings C and D contain the modified TMinMaxForm; we've highlighted the changes in color.

Listing C: TMinMaxForm header

```
//---------------------------------
#ifndef Main2H
#define Main2H
//---------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------
class TMinMaxForm : public TForm
   {
   __published:
      void __fastcall
         FormCreate(TObject *Sender);
      void __fastcall
         FormClose(TObject *Sender,
         TCloseAction &Action);

   private:
      bool restoreToMaximized;
```

```
   public:
      __fastcall
         TMinMaxForm(TComponent* Owner);
      void __fastcall
         OnRestore(TObject *Sender);
   };

//---------------------------------
extern TMinMaxForm *MinMaxForm;
//---------------------------------

#endif
```

Listing D: TMinMaxForm source

```
//---------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Main2.h"
#include "winplace.h"
#include <fstream.h>
//---------------------------------
#pragma resource "*.dfm"

TMinMaxForm *MinMaxForm;
//---------------------------------
__fastcall
TMinMaxForm::TMinMaxForm(TComponent*
   Owner) : TForm(Owner)
   {
   restoreToMaximized = false;

   Application->OnRestore = OnRestore;
   }
//---------------------------------
void __fastcall
TMinMaxForm::FormCreate(TObject *Sender)
   {
   WWindowPlacement place;
   ifstream is("minmax.set");
   is >> place;
```

```
int cmdShow = System::CmdShow;

if (cmdShow == SW_SHOWDEFAULT ||
    cmdShow == SW_HIDE)
   {
   STARTUPINFO startupInfo;
   GetStartupInfo(&startupInfo);

   if ((startupInfo.dwFlags &
      STARTF_USESHOWWINDOW) &&
      startupInfo.wShowWindow != SW_NORMAL &&
      startupInfo.wShowWindow != SW_RESTORE)
      cmdShow = startupInfo.wShowWindow;
   else
      cmdShow = place.showCmd;
   }

if (cmdShow == SW_MINIMIZE ||
   cmdShow == SW_SHOWMINIMIZED ||
   cmdShow == SW_SHOWMINNOACTIVE)
   {
   ::ShowWindow(Application->Handle, SW_HIDE);
   ::ShowWindow(Application->Handle, SW_MINIMIZE);
   Application->ShowMainForm = false;
   }
else if (cmdShow == SW_MAXIMIZE ||
   cmdShow == SW_SHOWMAXIMIZED)
   WindowState = wsMaximized;

if (cmdShow == SW_MINIMIZE ||
   cmdShow == SW_SHOWMINIMIZED ||
   cmdShow == SW_SHOWMINNOACTIVE)
   {
   ::ShowWindow(Application->Handle, SW_HIDE);
   ::ShowWindow(Application->Handle, SW_MINIMIZE);
   Application->ShowMainForm = false;
   if (place.showCmd == SW_MAXIMIZE)
      place.flags |= WPF_RESTORETOMAXIMIZED;
   cmdShow = SW_HIDE;
   }

place.showCmd = cmdShow;
```

```
   place.Set(this);

   if (place.flags & WPF_RESTORETOMAXIMIZED)
         restoreToMaximized = true;
   }
//-------------------------------------
void __fastcall
TMinMaxForm::OnRestore(TObject *Sender)
   {
   if (restoreToMaximized)
      {
      restoreToMaximized = false;
      WindowState = wsMaximized;
      }

   Visible = true;
   }
//-------------------------------------
void __fastcall
TMinMaxForm::FormClose(
   TObject *Sender,
   TCloseAction &Action)
   {
   WWindowPlacement place(this);
   if (::IsIconic(Application->Handle))
      {
      if (::IsZoomed(Handle))
         place.flags |= WPF_RESTORETOMAXIMIZED;
      place.showCmd = SW_MINIMIZE;
      }

   ofstream os("minmax.set");
   if (os) os << place;
   }
//-------------------------------------
```

You'll first change the FormCreate() function. The added code creates an instance of WWindowPlacement called place, then attempts to fill place with the state, position, and size data stored in a file named minmax.set.

If the user has specified a special startup state for the form, through a Windows shortcut or some other means, the code sets place.showCmd equal to that state. Otherwise, you use the state already stored in place.

Finish FormCreate() by checking to see whether place.flags has the WPF_RESTORETOMAXIMIZED flag set. If this flag is set, then set restoreToMaximized to true. You've declared restoreToMaximized as a bool in the private section of TMinMaxForm. You'll need restoreToMaximized in the TMinMaxForm::OnRestore() function.

If a form is maximized, then minimized, and then closed, WwindowPlacement will store this information. The form will appear minimized the next time the application runs. When the user restores the form, it should appear maximized. You accomplish this by checking restoreToMaximized in OnRestore(). If restoreToMaximized is true, you set the form's WindowState property to wsMaximized.

The new function FormClose() handles the OnClose event. In FormClose() you create an instance of WWindowPlacement that contains the data for the TMinMaxForm and saves it to your settings file.

You'll recall from last month's article that the main form is never really minimized. Rather, the VCL hides the main form and minimizes the application's hidden main window. Before writing place to your settings file, you need to check whether the application's main window, Application->Handle, is minimized. If it is, you set place.showCmd equal to SW_MINIMIZE. If the main window is minimized and your main form is maximized, you need to add the WPF_RESTORETOMAXIMIZED flag to place.flags.

Before closing

WWindowPlacement is a very useful and versatile class. In this article, we've chosen to write the state information to a file; however, you can easily modify the code to save this information in the Registry. You can also use WWindowPlacement, without any changes, to save the information for MDI child windows in your application. The underlying Windows API is smart enough to realize that the placement data is relative to a parent window instead of the desktop.

Does this register?

For more information about using the Registry to store application-specific information, see "Putting the Registry to Work" in the November 1997 issue of *C++Builder Developer's Journal*.

January 1998

Cheating with TUpdateSQL

by Gerry Myers

I recently had to add database support to a C++Builder application. I'd really gotten to like the VCL and I knew C++Builder had built-in database components, so I wasn't worried. My database needed to display its rows in different sorted orders, so I decided to use the TQuery component. I dropped the component onto my form, set the required properties (Database and SQL), linked in a data source and grid, and clicked Run. When the program opened, I thought, "Wow! I just added database access to my program in less than five minutes. I have the rest of the week to goof off." My database grid displayed the proper sorted order (thanks to the SQL statement in the TQuery component), and I could scroll through the records with no problems.

Next, I decided to change a value in one of the grid fields just to verify that everything worked, but the grid wouldn't let me change the data. I could select and highlight any cell, but that was it.

I consoled myself with the idea that I must not have set the grid's ReadOnly property to false--or perhaps I forgot to set the TQuery's RequestLive property to true. Unfortunately, it wasn't that easy. All my component properties were set correctly. The grid was *not* set for ReadOnly, and the TQuery *was* requesting a live resultant set from the SQL's SELECT statement.

When I dug into the Help files and accompanying C++Builder database manual, I found that the Borland Database Engine (BDE) will *attempt* to return a live resultant set from a SELECT query, but in some cases it just can't. If the SELECT statement contains an ORDER BY clause (which mine did), the BDE returns a *non-live* resultant set. Non-live means that the data can be displayed--in a grid or edit field, for instance--but can't be modified.

If you're considering using C++Builder for database applications, you're probably thinking the same thing I was: What good is a query component if the data flows only one way? Luckily for you and me, the Borland folks recognized this limitation as well and provided a back door to non-live data sets. This capability is wrapped in a VCL component called TUpdateSQL.

In this article, we'll introduce you to the TUpdateSQL component and demonstrate how to use it to do the "dirty work" for the TQuery. It turns out that integrating the TUpdateSQL component with TQuery is surprisingly easy. By the time I had everything working in my program, I still had a few days left to goof off--but don't tell anyone.

A query's resultant set

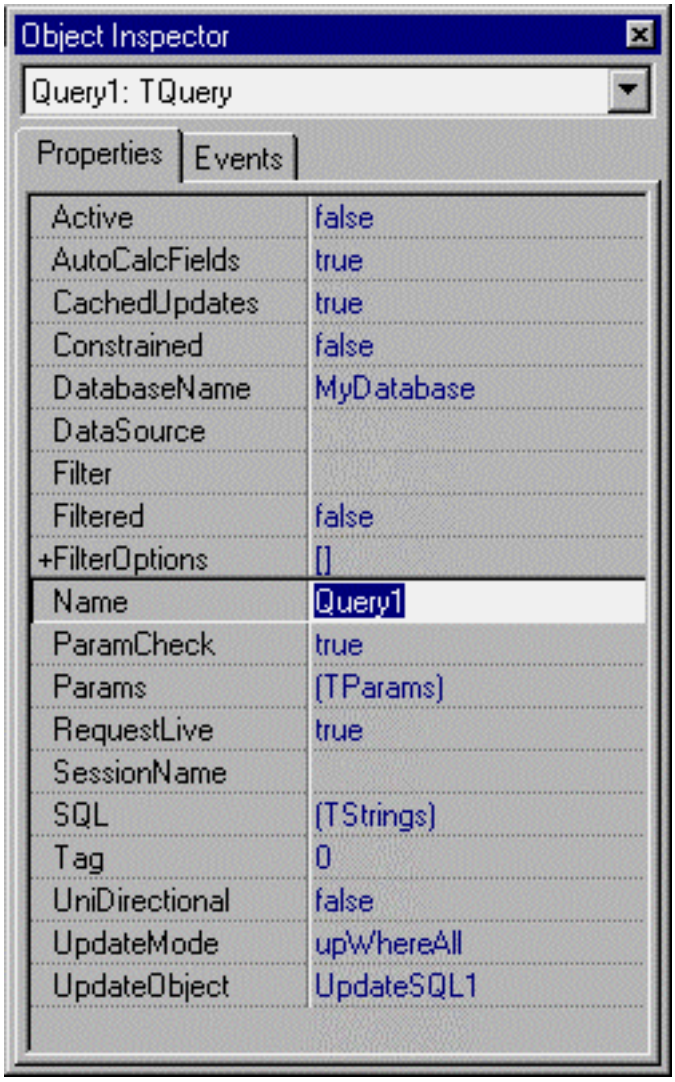
Before we dive into the TUpdateSQL component, let's discuss a few bits of background information. First, you need to understand that data returned from a query is called a *resultant set*. If the query contains a SELECT SQL statement, the query returns a resultant set (data from the database) that you can use and/or display. However, if the query contains a DELETE, INSERT, or UPDATE SQL statement, a resultant set isn't returned (that is, no data are returned). As far as this article is concerned, the important thing to remember is that a SELECT query returns a resultant set. Resultant sets come in two flavors: live and not live. A simple SELECT query returns a live resultant set. This data can be displayed, modified, and saved back to the database. If the SELECT query contains qualifiers or constraints like ORDER BY or sub-queries, it returns a non-live resultant set. Figure A shows examples of simple and complex queries.

Figure A: These examples illustrate simple and complex SQL SELECT queries.

```
// Simple SELECT query returning
// a live resultant set.
SELECT MeasId, Description
  FROM MeasTable
  WHERE GroupIndex = Pressure

// Complex SELECT query returning
// a non-live resultant set due to
// the ORDER BY clause.
SELECT MeasId, Description
  FROM MeasTable
  WHERE GroupIndex = Pressure
  ORDER BY MeasId

// Complex SELECT query returning
// a non-live resultant set due to
// the sub-query.
SELECT MeasId, Description
  FROM MeasTable
  WHERE GroupIndex =
    SELECT GroupIndex
      FROM GroupTable
      WHERE TestIndex = 2
```

When you drop a TQuery component on your form and scroll through its properties, you'll come across a property called RequestLive, as you can see in Figure B.

Figure B: The TQuery's properties include RequestLive.



This property defaults to false, which tells the BDE that you don't need a live resultant set even if you are *legally* entitled to one. This result is good if you want read-only data, but that isn't what I needed. When you set RequestLive equal to true, the BDE will *try* to return a live resultant set. However, if your SELECT query statement is complex, the BDE can't oblige you. This is where I sat--I was requesting a live resultant set, but because my SELECT query was complex, the BDE could only return a non-live resultant set. What to do?

Cached updates

The TQuery's CachedUpdates property also plays a role in modifying non-live resultant sets. You generally use this property when accessing your database over a network. You can tell the BDE (by setting CachedUpdates equal to true) that you want it to locally hold all changes that you or the user makes to the data. The BDE *caches* the updates in a local buffer and writes them to the database as a single transaction when you tell it to later. This process cuts down on network traffic. CachedUpdates has another application, though, when working with local databases. The BDE won't let you modify a non-live resultant set--however, it can hold all your changes in a local buffer, since that doesn't violate the non-live resultant set

rule. The actual data in the database isn't changed, so the SQL cops are happy.

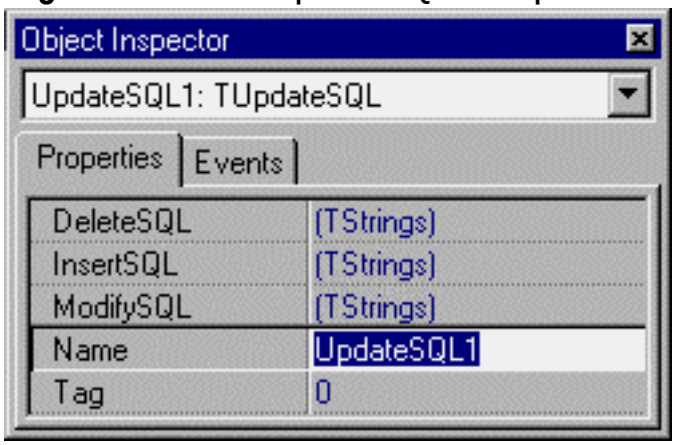
Back in my program, I set the TQuery's CachedUpdates property to true. When I ran the program, the grid still wouldn't let me change any data values. I knew I was on the right track, but something else was missing.

TUpdateSQL

As I read through the Help files concerning resultant sets and cached updates, I discovered a component called TUpdateSQL. You can link this component to a TQuery component and use it to make changes to a non-live result set. To allow the TQuery and TUpdateSQL components to work correctly together, the TQuery must cache its updates. It looked as though I was on the right track. How exactly does the TUpdateSQL link to the TQuery? This is where the "back door" comes in. Whenever the user or the code makes a data modification to a TQuery resultant set, the component pointed to in the TQuery's UpdateObject property is executed. As you can see in Figure B, the TQuery's UpdateObject property is set to my new TUpdateSQL component. Each time the user changes the data by editing the values in the grid cells, my TUpdateSQL is executed. The TUpdateSQL actually makes the change to the database. This is kind of like the Godfather who can't go out and do his own dirty work--he sends out his hit man, instead. Each time a record is changed, the TUpdateSQL is executed to take care of this one change to the database.

The TUpdateSQL property list shown in Figure C is short, so there's not much to learn.

Figure C: The TUpdateSQL component has relatively few properties.



As the properties show, you can use TUpdateSQL for deleting, inserting, and generally modifying a database. Each of the properties can contain separate SQL statements. I used the following SQL statement to allow the user's changes in the grid to be written to the cached update buffer:

```
UPDATE Measure.DB
```

```
SET Reading = :Reading
WHERE MeasId = :OLD_MeasId
```

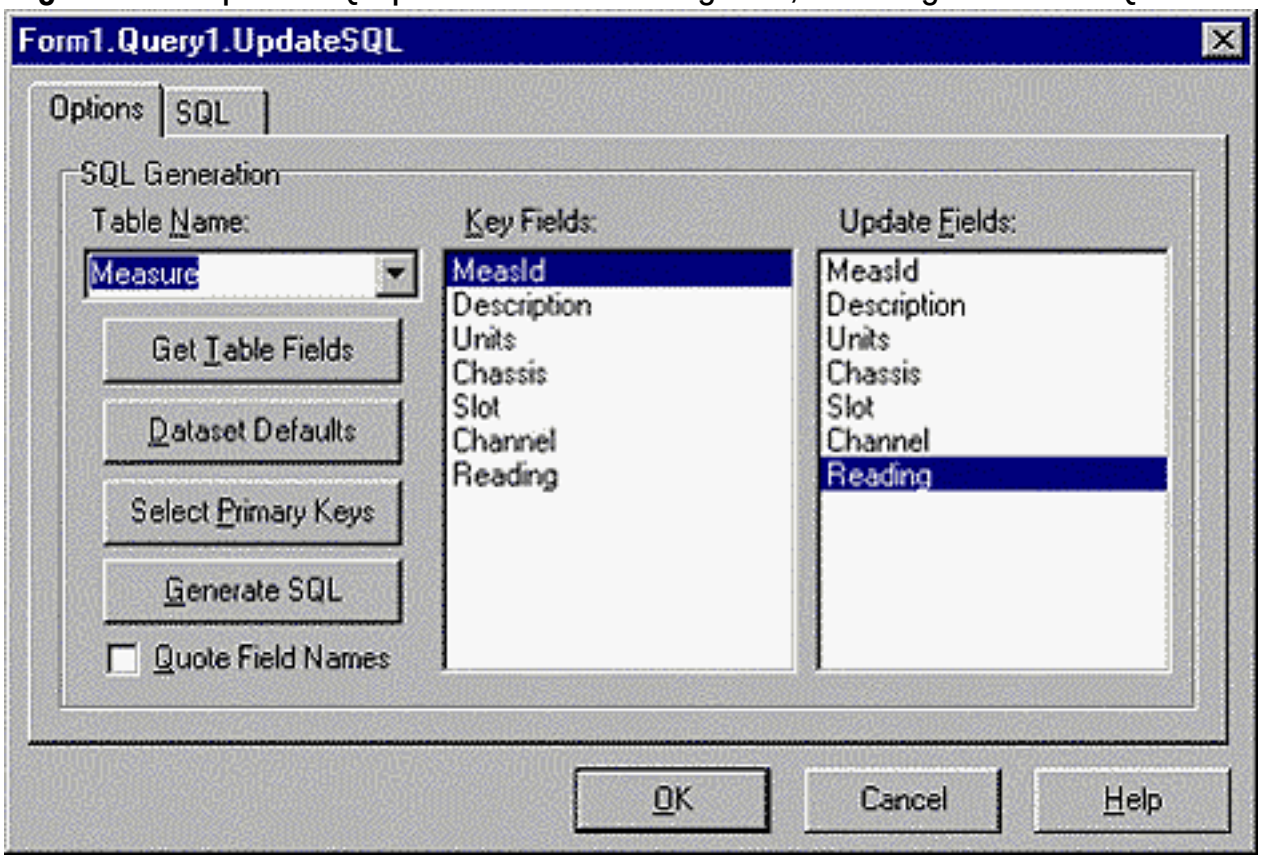
The only field that I allowed to change was Reading.

A couple of things may look strange to you in this SQL statement. The first is :Reading, which is a variable for the database field called Reading. The TQuery component supplies this variable when the TUpdateSQL component is executed. It holds the value in the Reading field that was just entered by the user or modified by the program. The other oddity is :OLD_MeasId--this is the value from the database's MeasId field *before* the current change was made to the database. The TQuery component also supplies this value.

In plain English, my SQL statement finds the record in the underlying database that matches the MeasId of the current record (the WHERE part). It then updates the value of that record's Reading field with the new value (the SET part). So far, the data change goes only into the cached buffer with the other modifications--and not yet all the way to the underlying database.

TUpdateSQL includes an SQL generator dialog box, shown in Figure D, which will create for you an SQL statement similar to my example.

Figure D: TUpdateSQL provides this dialog box, which generates SQL statements.



You simply make a few selections in the dialog box and select the Generate SQL button. To open the dialog box, double-click on the TUpdateSQL component on your form. You make your selections on the Options tab. The SQL tab lets you see and modify the generated SQL statement. The TQuery component that's linked to your TUpdateSQL provides the values in the Key Fields and Update Fields lists. If you like, you can manually enter your own SQL statement into the appropriate TUpdateSQL property (DeleteSQL, InsertSQL, or ModifySQL).

Apply/Commit cached updates

s As we mentioned, the Cached Updates buffer holds any changes to the data made by the user (through the grid) or by the application. If you ran your program at this point, it would appear that everything was working. You'd be able to make changes to the grid's Reading field (thanks to the TUpdateSQL component), but that's as far as it would go. Since the new values haven't been written to the actual database, the updates would be lost when you exited the program. To save changes from the Cached Update buffer to the actual database, you must tell the database to apply and commit them. The following code accomplishes this task:

```
// Setting tiDirtyRead is
// required for Paradox tables.
Query1->Database->
    TransIsolation = tiDirtyRead;

// Tell the database to Apply and
// Commit the cached updates.
Query1->Database->
    ApplyUpdates(
        OPENARRAY(TDBDataSet*,
            (Query1)));
```

(For Paradox tables, the transaction isolation property--TransIsolation--must be set to tiDirtyRead.) This code allows other applications to read the applied (but maybe not committed) data from the actual database. The second line of code commands the database to apply (and commit) the data in the Cached Update buffer. The single database method ApplyUpdates() both applies and commits. It's acceptable to instead call the *dataset's* ApplyUpdates() method, but the changes won't be permanent and may be subject to a later rollback. If you do use the dataset's ApplyUpdates() method, you must also call the *database's* Commit() method to make the changes permanent. You'd enter this code in your application at the point that you want to make the updates permanent--for instance, when the program exits, when a button or menu is chosen, or when the grid loses focus.

Putting it all together

In this article, we discuss the elements that allow you to modify data from a non-live resultant set. Here's a handy reference guide to those elements.

1. Drop a TQuery and a TDataSource component onto your form and set their properties to point to your database. Enter an SQL statement to return the desired non-live resultant set.

2. Set the TQuery's RequestLive and CachedUpdates properties equal to true.

3. Drop a TUpdateSQL component on your form.

4. Set the TQuery's UpdateObject property to point to your TUpdateSQL component (link the TUpdateSQL to your TQuery).

5. Double-click the TUpdateSQL component to start up the SQL generator, then make your field selections. An alternative is to double-click one of the TUpdateSQL's SQL properties and manually enter your SQL statement.

6. Drop a data-aware grid (or other data-aware component) onto your form and set its properties to point to your TQuery. Set its ReadOnly property to false.

7. Run the program and make changes. Exit and restart the application to verify that your updates were in fact written to the database even though the TQuery returned a non-live resultant set.

Conclusion

Sometimes it can be frustrating to integrate a database system into your application when the rules of SQL seem to get in the way. However, you can use the TUpdateSQL component in many situations to update almost any dataset. TUpdateSQL is powerful, but it can also be confusing, since it doesn't operate on its own behalf. Instead it's linked to a TQuery component and carries out actions that the TQuery isn't allowed to perform. When I use TUpdateSQL, I feel as though I'm cheating, since the data returned from my query isn't intended to be modified. Come to think of it, maybe that's why I like using it so much--we programmers don't have many chances to enjoy a feeling of power and control!

January 1998

Rotated fonts

by Kent Reisdorph

Have you ever needed to print text at an angle? Maybe you wanted a sideways caption for a chart's X-axis, or you needed to print text upside down. If so, you probably noticed right away that the standard Label component can only display text normally--no rotation allowed. In this article, we'll demonstrate how to print rotated text. We'll show you how to use the Windows API to rotate TrueType fonts. By combining the information in this article with the information in our earlier series on creating components, you can easily create your own rotated label component. (See "Building Components" parts 1, 2, and 3 in the September, October, and November 1997 issues of *C++Builder Developer's Journal*.) If you'd like to review some text-drawing basics before tackling rotated text, see "Text Drawing 101."

The LOGFONT structure

One of the great things about the VCL is that it shields you from the API. The Font property is a perfect example. At the API level, you create fonts in Windows by filling out a LOGFONT structure and then calling the CreateFontIndirect() API function. The LOGFONT structure looks like this:

```
typedef struct tagLOGFONTA
{
    LONG      lfHeight;
    LONG      lfWidth;
    LONG      lfEscapement;
    LONG      lfOrientation;
    LONG      lfWeight;
    BYTE      lfItalic;
    BYTE      lfUnderline;
    BYTE      lfStrikeOut;
    BYTE      lfCharSet;
    BYTE      lfOutPrecision;
    BYTE      lfClipPrecision;
    BYTE      lfQuality;
    BYTE      lfPitchAndFamily;
    CHAR      lfFaceName[LF_FACESIZE];
} LOGFONTA;
```

The various members of this function specify the font's name, point size, style (bold, italic, underline, strikeout), and other attributes. One of the special attributes is, of course, the font's angle. Most of the time you don't have to worry about all this, because the VCL fills in this structure behind the scenes and creates the font for you. All you have to do is set the Font property's Name, Size, or Style, and VCL takes care of the nitty-gritty details--this is great, because you can deal with fonts at the component level. However, when you get to specialized font operations--such as setting the rotation angle--you need to go to the API and temporarily work around the VCL.

We're concerned with two members of the LOGFONT structure: lfEscapement and lfOrientation. The lfEscapement member controls the font's angle, specified in tenths of degrees (for example, a value of 900 would mean 90 degrees). The lfOrientation member controls the rotation of the individual letters within the text string. This feature is supported only under Windows NT 4.0, so it's probably not something you'll use regularly. If you want to know what each member of the LOGFONT structure does, look up LOGFONT in the Win32 online Help.

Modifying a VCL Font object

To change the escapement and orientation for a VCL Font object, you need to perform the following steps:

1. Extract the LOGFONT information for the currently selected font.
2. Modify the lfEscapement and lfOrientation members.
3. Create a new font from the LOGFONT structure.
4. Assign the new font to the VCL Font object's Handle property.

You'll extract the LOGFONT settings from the VCL Font object by calling the GetObject() API function. Given a handle to a font object, this function will fill in a LOGFONT structure. The TFont class's Handle property contains the font's handle (HFONT). It looks like this:

```
LOGFONT lf;
GetObject(Canvas->Font->Handle,
  sizeof(LOGFONT), &lf);
```

At this point, the LOGFONT structure has been filled in with information obtained from the form's Font property. The next step is to change the escapement and orientation. Both of these parameters are specified in tenths of degrees. For example, to rotate the font 45 degrees, you'd set lfEscapement and lfOrientation to 450:

```
lf.lfEscapement = 450;
lf.lfOrientation = 450;
```

That's easy enough. Next, you need to create a new font from the modified LOGFONT structure and assign that font to the Font object's Handle property. You can do so in one step:

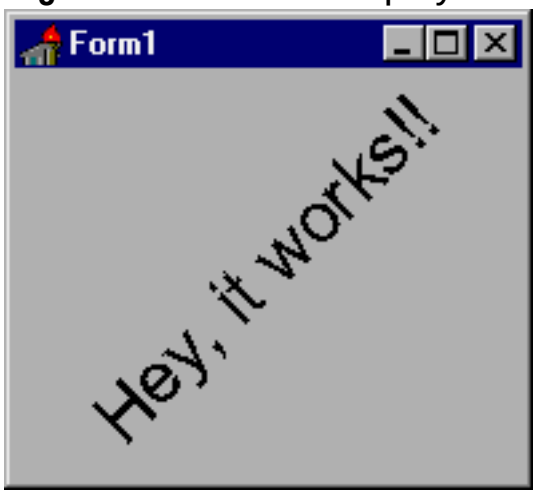
```
Canvas->Font->Handle = CreateFontIndirect(&lf);
```

The CreateFontIndirect() function returns a handle to the new font, which you can assign directly to the Font object's Handle property. Now that the new font is ready to go, you can call TextOut() or DrawText() to display the rotated text. The complete code from all the steps is as follows:

```
Canvas->Font->Name = "Arial";
Canvas->Font->Size = 20;
LOGFONT lf;
GetObject(Canvas->Font->Handle,
  sizeof(LOGFONT), &lf);
lf.lfEscapement = 450;
lf.lfOrientation = 450;
Canvas->Font->Handle = CreateFontIndirect(&lf);
Canvas->Brush->Style = bsClear;
Canvas->TextOut(20, 120, "Hey, it works!!");
```

To test this code, create a new project and place a button on the form. Enter the code in the event handler for the button's OnClick event. When you click the button, the text will be displayed at a 45-degree angle, as shown in
Figure A.

Figure A: Our form displays text at a 45-degree angle.



Additional considerations

There are a few things you need to know before putting rotated fonts to work. First, rotating fonts in this manner works only with TrueType fonts. Interestingly, the default font for VCL visual components is MS Sans Serif, which isn't a TrueType font. In the preceding example, we set the font name to Arial--if we hadn't done this, then the font rotation would have failed because the default font can't be rotated. You can get around this problem by forcing Windows to use a TrueType font regardless of the font selected. To do so, set the lfOutPrecision member to OUT_TT_ONLY_PRECIS when you set up the LOGFONT structure. For example, the line

```
lf.lfOutPrecision = OUT_TT_ONLY_PRECIS;
```

will cause Windows to pick the closest matching TrueType font in the event that the current font isn't TrueType. Another thing to keep in mind is that any time you access the Font property in the normal VCL way, the contents of the LOGFONT structure will be reset. Given that, the following code will fail:

```
lf.lfEscapement = 450;
lf.lfOutPrecision = OUT_TT_ONLY_PRECIS;
Canvas->Font->Handle = CreateFontIndirect(&lf);
Canvas->Font->Size = 20;
Canvas->TextOut(20, 120, "Test");
```

All your work setting up the LOGFONT structure is wiped out when you set the font's Size property. The moral here is to set any VCL Font object attributes *before* modifying the LOGFONT structure. The Win32 online Help states that the lfOrientation member isn't used under Win95 but *is* used under NT. It further states that you should set the orientation and escapement to the same value. You really don't need to set the orientation--but, just to be safe, you might heed Microsoft's instructions and set both the orientation and escapement.

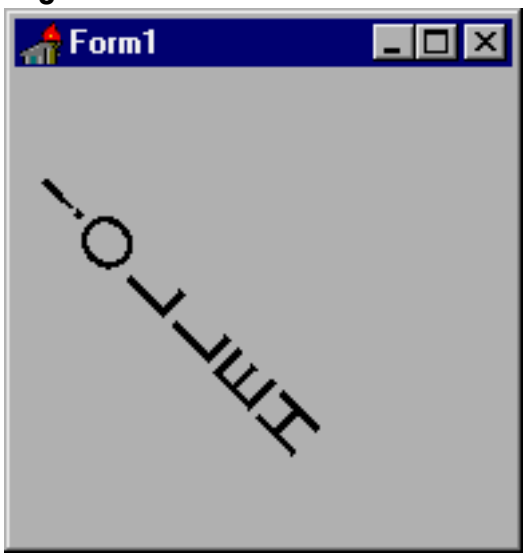
Finally, under Windows NT, you can rotate both the line of text and the individual characters within the line. To do so, you must set the escapement to the desired angle for the text line and the orientation to the desired angle for the characters within the text. That's not the whole story, though--none of this will have any effect unless you set the graphics mode to use the advanced graphics routines. You do that by calling SetGraphicsMode() as follows:

```
SetGraphicsMode(Canvas->Handle, GM_ADVANCED);
Canvas->TextOut(20, 120, "Test");
```

Now both the orientation and escapement values will be taken into account. You may notice some odd spacing problems when rotating characters. Plus, since this is a Windows NT-only feature, you probably won't be able to take advantage of changing the orientation of the characters in most cases. Figure B shows the result of setting the escapement to 1350 and

the orientation to 450 in Windows NT 4.0.

Figure B: Here we rotated both the font and the character.



An example

Listings A and B contain a program that illustrates the concepts we've discussed in this article. (You can download our sample files from **www.cobb.com/cpb** as part of the file jan98.zip.) To create this program, start with a new project, create an event handler for the OnPaint event, and enter the code. Experiment with different escapement and orientation settings (if you're using Windows NT) and see what effects those changes have on the way the text is displayed.

Listing A: FONTTSTU.H

```
//---------------------------------------------
#ifndef FontTstUH
#define FontTstUH
//---------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormPaint(TObject *Sender);
private:  // User declarations
public:  // User declarations
```

```
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: FONTTSTU.CPP

```
//---------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "FontTstU.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  // Start with a reasonable size.
  Canvas->Font->Size = 20;
  // Fill in the LOGFONT structure.
  LOGFONT lf;
  GetObject(Canvas->Font->Handle,
    sizeof(LOGFONT), &lf);
  // Change escapement, orientation, output precision
  lf.lfEscapement = 450;
  lf.lfOrientation = 450;
  lf.lfOutPrecision = OUT_TT_ONLY_PRECIS;
  // Create new font; assign to Canvas Font's Handle.
  Canvas->Font->Handle =
    CreateFontIndirect(&lf);
  // The following only works on NT!
  SetGraphicsMode(Canvas->Handle, GM_ADVANCED);
  // Set the brush style to clear.
  Canvas->Brush->Style = bsClear;
  // Display the text in the middle of the form.
```

```
  Canvas->TextOut(
    Width/2, Height/2, "Hello World!");
}
```

January 1998

Drawing text 101

by Kent Reisdorph

Let's examine the basics of displaying text onscreen. The TCanvas class has a TextOut() method that you can use to display text on a canvas. The following code uses TextOut() to display a line of text on a form starting at position 20, 20 (20 pixels from the left of the form and 20 pixels from the top):

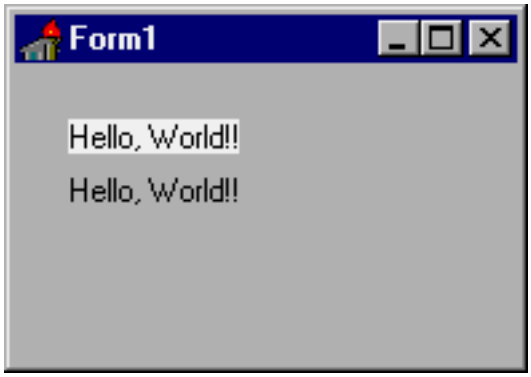
```
Canvas->TextOut(20, 20, "C++Builder Rules!");
```

The text is drawn on the form using the form's Font property settings. Testing this method is easy: Simply place a button on a form and enter our sample line of code in the button's OnClick handler. When you run the program, you'll notice that the text has a white background--this background is the default for text displayed with TextOut(), but it isn't very appealing. In order to let the background color show through, you must make the text transparent by setting the brush style to bsClear. To do so, add one line to the previous code:

```
Canvas->Brush->Style = bsClear;
Canvas->TextOut(20, 20, "C++Builder Rules!");
```

You could accomplish the same result by setting the brush color to the form's background color, but setting the style to bsClear is preferable because you don't have to worry about what's on the form beneath the text. Figure A shows a program that displays text using TextOut(). The program displayed the first line before setting the brush style to clear and the second line after setting the brush style to clear.

Figure A: Our sample program displays text using TextOut().



The TextOut() method of TCanvas simply calls the Windows API function of the same name. There's another text function in the Windows API that we should mention: DrawText(). It provides a more flexible method of displaying text on a canvas. DrawText() lets you center

text horizontally and vertically, right-justify text, left-justify text, truncate too-long text with an ellipsis (...), and perform other specialized functions.

For whatever reason, the TCanvas class doesn't have a DrawText() method. So to use DrawText(),you'll have to call on the API. A typical DrawText() call would look something like this:

```
TRect r = Rect(20, 20, 120, 40);
DrawText(Canvas->Handle, "DrawText in action",
  -1, (RECT*)&r, DT_SINGLELINE);
```

While this code might seem like a lot more work, keep in mind that sometimes DrawText() is the only way to display text in exactly the way you want. We won't go into detail about DrawText() here; see the Win32 API online Help for more information. This is one function that should definitely be in your arsenal if you're going to write your own components.

January 1998

Use a mutex to achieve synchronization

by Sam Azer

Over the past few months, a few people on the CPB-Thread listserv have asked how to prevent more than one instance of an application from being started. (To subscribe to CPB-Thread, visit **www.cobb.com/cpb**.) Several people have worked very hard on a variety of answers to this question. It seems, though, that the simplest solution is one suggested by members of Borland's TeamB on their news server: a *named mutex*. As you know, Windows 95 provides a number of built-in functions for running multiple tasks and threads within a task. Accordingly, a group of *Synchronization functions* is available to ensure that your tasks and threads are able to work together effectively.

One of the more common problems in a multitasking environment is resource sharing. If you're using a resource--for example, a serial port--you don't want somebody else to start using it before you're finished. The standard solution to this problem is for you to get a *mutual exclusion lock* (mutex) on the resource. If you're able to get the lock, you can use the resource. If, while trying to lock the resource, you find that somebody else has it--you must wait.

In this article, we'll build a small and very simple function to return true if an instance of a program is already running. To do this, we'll use a simple named mutex. (You can download the code from this article at **www.cobb.com/cpb**.) Let's start by looking at how you can use a mutex.

Using a named mutex

A mutex is a very simple object. It has a name, access rights, and a state. The name must be unique among all mutex, semaphore, and file-mapping objects. In most cases, the access rights will be MUTEX_ALL_ACCESS. The state can be Owned or Not Owned. If you create a resource that can be used by only one caller at a time, you'll probably also create a mutex for it. In this case, your callers try to open the mutex to see if the resource exists. Once they have a valid handle to the mutex, they use wait functions to gain ownership of the resource and start using it. Wait functions sleep until a mutex isn't owned, then take ownership of it and use the resource. (You can optionally place a time limit on a wait function.) Eventually, the caller will release the mutex, marking it not owned and available to the next caller. In this way, all callers wait to take ownership of the resource when they need it and release it when they're finished. If a caller has no further need for a resource, the mutex handle can be closed.

Once a mutex exists, any tasks trying to create it again will get a handle to it and an *error already exists* error. If a task ends, all the mutex handles that it owns are closed. When all tasks have closed their

handles to a mutex, it's destroyed. Therefore, a single call to the CreateMutex() function is all you need to find out if an instance of an application is already running!

Using CreateMutex()

The CreateMutex() function takes only three parameters: a pointer to a security descriptor, a flag to request immediate ownership, and the name of the mutex to create. It returns a HANDLE. It ignores the pointer to the security descriptor under Windows 95. Under Windows NT, you can get default security settings by passing a NULL pointer. The mutex name is simply any unique null-terminated string (up to MAXPATH characters in length). If the returned HANDLE is null, something undesirable happened. In any case, GetLastError() should return zero to indicate that the mutex was created. A value of ERROR_ALREADY_EXISTS indicates that the mutex already exists. Use SysErrorString() to translate the other error codes into English.

By now you've probably realized that there really isn't much to it--you can very easily detect an existing instance of an application using this method. Listing A shows the source code for one way to do it.

Listing A: Detecting another instance of an application

```
//---------------------------------
        -----------#include <vcl\vcl.h>
#pragma hdrstop
//---------------------------------
        ------------USEFORM("OneOnlyForm.cpp", Form1);
USERES("OneOnly.res");
//---------------------------------
        ------------// the name of the mutex for this program
const char *MutexName = "OneOnlyDemo";
//---------------------------
        -----------------HANDLE
                CheckInstance( const char *Name )
{
   // 1st: create mutex. Request ownership.
   HANDLE Mutex = CreateMutex(NULL,true,Name);
   // Next, error result - should be zero
   int r = GetLastError();
   // if r != 0, probably ERROR_ALREADY_EXISTS
   if ( r != 0 )
      return 0; // disaster or another instance
   return Mutex; // else, return the handle.
}
//---------------------------
```

```
        ----------------WINAPI WinMain
                (HINSTANCE,HINSTANCE,LPSTR, int)
{
   HANDLE Mutex = CheckInstance( MutexName );
   if ( !Mutex )
   {  MessageBox(HInstance,
"Another Instance is running!",
"Sorry", MB_OK );
      ReleaseMutex( Mutex );
      return 1;
   };

   try
   {
      Application->Initialize();
      Application->CreateForm(__classid(
        TForm1),&Form1);
      Application->Run();
   }
   catch (Exception &exception)
   {
      Application->ShowException(&exception);
   }
   return 0;
}
//---------------------------------------------
```

Wrap up

Do a keyword search for *Synchronization* in the Win32 (Platform) SDK Help files for detailed information on the functions available for synchronizing--they're quite handy. In many cases the code presented in this article is enough to prevent problems from occurring between two instances of a simple database program. However, for MDI applications you'll probably want to send a message to the original instance. Watch for an article on inter-process communications in a future issue of *C++Builder Developer's Journal*.

December 1997

Detecting changing rows in a TDBGrid

by Jim Bailey

How can we detect the data cursor's movement-s from one row to another in a data grid? There's no direct event available to signal the change of a grid row--a little C++Builder problem that leaves me scratching my head and saying "Why did they do it that way?"

The OnColEnter event fires every time the column changes. When I began working with C++Builder, I was sure OnColEnter had a close cousin in the OnRowEnter category. I found no reference to such an event in the documentation, but there are many great features Borland's documentation didn't mention--surely the row-change event had to be one of them.

Well, after posting several forum messages and getting no response, I've come to believe the TDBGrid's row change doesn't exist, after all. However, I needed a workaround, and I did find one that had economical code and didn't require developing a new component. I later learned of a more elegant method, but I seldom use it. Let's look at both methods.

Method #1

On a form with a TDBGrid component, add a TDBEdit component and set its Visible property to False. Set the TDBEdit component's DataSource to point to the same TdataSource as the grid. Set the DataField property to the primary key of the data set feeding the grid. Next, set up the TDBEdit component's OnChange event, which fires every time the row changes--assuming the primary key is a single-field index. Applications with multiple-field primary keys can use the second method I'll describe, but the TDBEdit component's OnChange event is adequate in many cases. Plus, it offers another advantage. I often need to detect a change in a certain field while scrolling through a database. Detecting such changes requires considerable high-maintenance code like that shown in Figure A.

Figure A: Detecting a change in a database field requires some high-maintenance code.

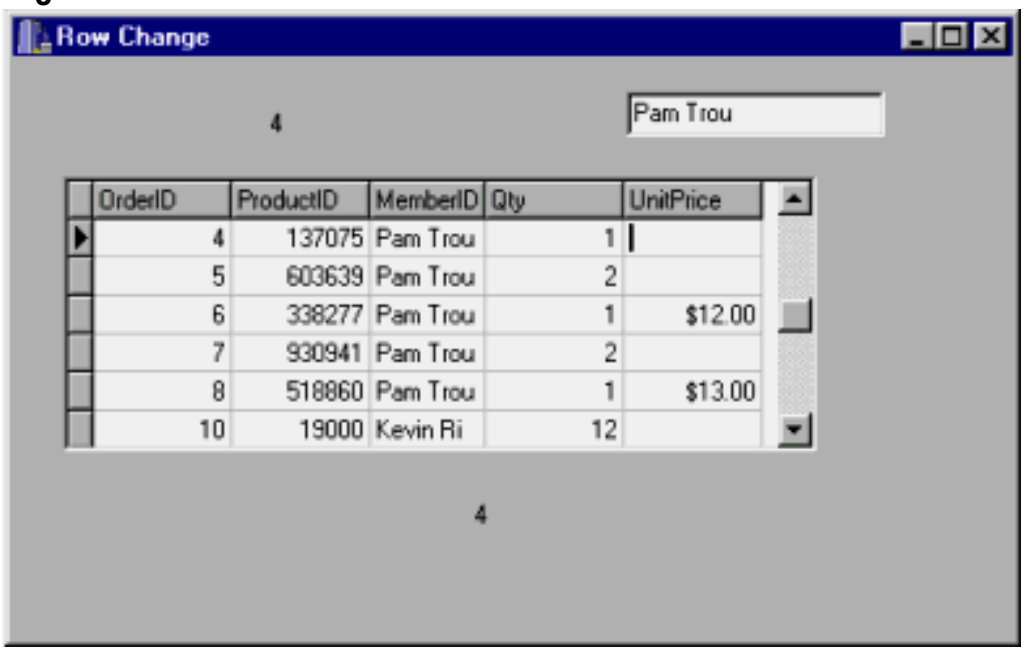
```
bool FieldTextChanged()
{
  bool returnValue= false;
  static String ThisWas("v o i d");
  String ThisIs;

  //Get the target fields value.
  ThisIs = TargetGrid->Columns->Items[1]->Field->AsString;
```

```
  // Track column movement or this won't work.
  if(ThisWas == "v o i d")      // first iteration
    ThisWas = ThisIs;
  else {
    if(ThisIs != ThisWas) {
      //Wake up Herman, the data changed!
      ThisWas = ThisIs;
      returnValue = true;
    }
    return returnValuea;
  }
}
```

There's no need to analyze this code, because it's completely unnecessary. You can track the entire process without a single line of code using the method we just described. How economical can you get!

Consider the grid shown in Figure B. You can quite easily determine when the MemberID changes. Simply add a TDBEdit component, set its DataSource property as described earlier, and set the DataField property to the MemberID field. The control's OnChange event will fire only when the text in the MemberID field changes.

Figure B: You can use our method to determine when the MemberID in this table changes.



I used to think of Method #1 as a kludge, but I'm beginning to consider it a respected technique. Think of the code you can eliminate, and the flexibility you have at revision time. But as much as I like this technique, there are a few reasons not to use it--and some programmers won't like the non-standard use of the TDBEdit control. However, I'll take an invisible control with no code over a slick, high-tech, three-lines-of- code *proper* method

almost every time.

Method #2

Now let's look at a second, somewhat more elegant technique for achieving our goal. First, create an OnDataChange event in the TDataSource referenced in the grid's DataSource property. According to the help file, the event operates as follows: "OnDataChange fires when the current record has been edited and the application moves from one field or record to another in a dataset associated with the data source component." The event does *not* fire when the application moves from one *field* to another unless the previous field was changed. It does fire when dataset changes *state* from dsBrowse to dsEdit or dsInsert. It also fires when the data cursor moves from one record to another. This technique requires at least a few lines of supporting code, as you can see in Figure C. If the TDataSource is located in a DataModule form, the supporting code for this method could be considerable, especially if the TDataSource is used by multiple components and/or forms.

Figure C: Our second method requires some supporting code.

```
void __fastcall TfrmRowChange::dsDataDataChange(
  TObject *Sender,
  TField *Field)
{
  TDataSource *ds = dynamic_cast<TDataSource *>(Sender);

  //Get the dataset from Sender.
  if (ds) //The cast worked.
    if (ds->DataSet->State != dsBrowse)
      lblChangeNotice->Caption = tblDataOrderID->AsString;

      // If the state is Browse, the record
      // can't be changing.
    else
      do nothing;  // The event fired because
                   // the user was typing.
```

Searching a data set for a particular record

C++Builder database programmers often wonder how to locate a specific record. A database containing 15,000 records serves no purpose unless a user can go directly to the record she needs. Let's say you have a user who wants to access information about a company's office locations. The code shown in Listing A will serve that purpose.

Listing A: Locating the first record that contains a given value

```
TLocateOptions SearchOptions;

//Make the search case insensitive.
SearchOptions << loCaseInsensitive;
bool locateSuccess = Query1->Locate("CustName", "A Bank",SearchOptions);

//Find 1st record where CustName equals "A Bank".

if(locateSuccess)
  ShowMessage("Located Record");
else
  ShowMessage("Did Not Locate Record.");
```

This code will move the data cursor to the first record containing *A Bank* in the CustName field. However, the poor user might then discover that A Bank has offices all over the world. Locating an office in a particular city requires a search for the contents of two or more fields. The code in Listing B illustrates a search on two fields.

Listing B: Searching a database on two fields

```
  #define COLUMNS_TO_MATCH 2

  //Two field names
  String columns ("CustName;CustCity");

  //Initialize string with two column names
  //separated by a semicolon.

  Variant matchText =
    VarArrayCreate(OPENARRAY(int,(
    0,COLUMNS_TO_MATCH)),varVariant);

  //Declare a variant array to contain text to match in the search.

  //Array element zero= CustName field's match
  matchText.PutElement("A Bank",0);

  //Array element one = CustCity field's match
  matchText.PutElement("houston",1);

  TLocateOptions SearchOptions;
  SearchOptions << loCaseInsensitive;
```

```
  bool locateSuccess =
    Query1->Locate(columns,matchText,SearchOptions);

  //Locate the first record where the CustName
  //field's text equals "A Bank" and the
  //CustCity field's text equals "Houston".

  if(locateSuccess)
    ShowMessage("Located Record");
  else
    ShowMessage("Did Not Locate Record.");
```

The TlocateOptions variable brings powerful flexibility to the search criteria. The variable offers two options, loCaseInsensitive and loPartialKey. You can select both values using the following syntax:

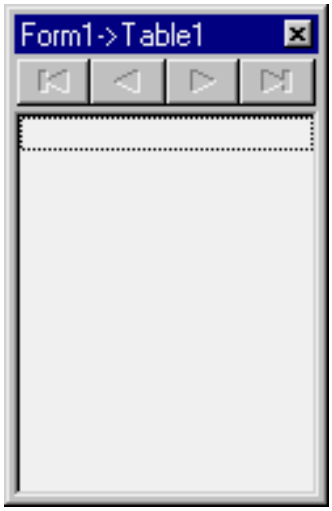
```
SearchOptions << loCaseInsensitive <<
  loPartialKey ;
```

The loCaseInsensitive option locates matches without regard to case. The loPartialKey option finds the first match beginning with the target text. For example, the target text *HAM* would match both *HAMMER* and *hamburger*.

Formatting output from data-aware components

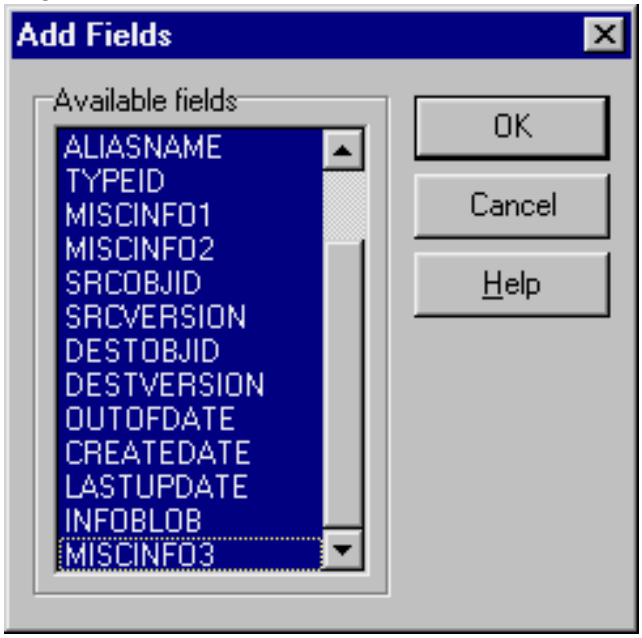
Formatting numeric and date output in grids and other data-aware components is one of the easiest tasks in C++Builder. Details are readily available--however, they can be hard to find. Some of the persistant data fields have a DisplayFormat property you can use, but accessing persistent fields is a little like a treasure hunt. Here are the clues. To begin, add a TTable object to a form or data module. Assign values to the DataBaseName and TableName properties as required. Now, double-click on the TTtable to open an empty field-editor window, as shown in Figure D. Right-click on the window, and a speed menu with two enabled items--Add Fields... and New Fields...--will appear.

Figure D: Double-click on the TTable component to open this empty editor window.



Select the Add Fields... option to open an Add Fields dialog box like the one shown in Figure E, containing a list of every field in the table. (Seriously, isn't this like a treasure hunt?) Click OK to add all the fields to the TTable object as persistent fields.

Figure E: Click OK to add all these fields to your TTable object.



I'm worn out--let's take a break. Such rote instructions are uncommon in RAD development environments. Mastering the process isn't difficult, but it isn't intuitive, either--unless you know how to do it. Okay, break's over.

At this point, every persistent field is selected in the field-editor window. Select one of the fields containing numeric or date data and press [F11]. When you do so, the properties for that persistent field will appear in the Object Inspector. One of those properties is the treasure: the long-sought DisplayFormat property.

The latest help files provide adequate information on formatting options.

Table A contains some examples for numeric fields.

Table A: Sample numeric data formats

Format	Numeric 1	Numeric 2	Numeric 3	Numeric 4
Raw Data*	1276.234	9324587.235	.235	.2351
.000	1276.234	9324587.235	.235	.235
0.000	1276.234	9324587.235	0.235	0.235
0.00	1276.23	9324587.23	0.23	0.24
,.00	1,276.23	9,324,587.23	.23	.24
,0.00	1,276.23	9,324,587.23	0.23	0.24
$,0.00	$1,276.23	$9,324,587.23	$0.23	$0.24
$, 0.00	$ 1,276.23	$ 9,324,587.23	$ 0.23	$ 0.24

*Data as it appears without a DisplayFormat property setting

Note that C++Builder contains a rounding discrepancy. The value .235 should round to the hundredth's place as .24--but it rounds to .23 instead. Any value greater than .005 is rounded correctly in the hundredth's place. For example, the value .2351, correctly rounds to .24. This discrepancy doesn't repeat when rounding to the tenth's place--the values .450 and .451 both round to .5. It's amazing--a measly ten-thousandth of a cent could keep the bookkeeping department up all night trying to find the penny that shows up in Paradox and is lost in C++Builder (Paradox correctly rounds .2350 to .24). Just another example of life's hidden treasures. Table B illustrates several date formats, which won't provide as many thrills. Once you set a persistent field's DisplayFormat property, the formatting cascades to every data-aware component referencing the field.

Table B: Sample date formats

Format	Date
Raw Date*	07/01/97
m-d-yy	7-1-97
mm/dd/yyyy	07/01/1997
yyyymmdd	19970701
mmm dd, yyyy	Jul 01, 1997
mmmm d, yyyy	July 1, 1997

*Data as it appears without a DisplayFormat property setting

Detecting selected rows in a TDBGrid control

Sometimes users need to select multiple rows in a TDBGrid and issue a command to process the selection(s). This task isn't difficult, but the documentation isn't replete with examples. OK, actually there aren't *any* examples--but the documentation is improving steadily. Let's look at a working example of our own. To begin, click on a grid object and open the Object Inspector by pressing [F11]. Locate the Options property and double-click on that line to expand the list of options.

Two option settings are critical for row selection: The dgRowSelect option must be True or the user can't select even one row, and dgMultiSelect must be True to allow multiple row selections. Both options are False by default. After the user selects one or more rows, run the code segment shown in Listing C to detect the selections.

Listing C: Detecting selected rows

```
TDataSet *ds = dgData->DataSource->DataSet;
//Get data set that feeds grid (dgData).

int selectedRows = dgData->SelectedRows->Count ;

//Determine how many rows are selected.

if(!selectedRows)
  ShowMessage("No Rows Are Selected.");
else
  for (int i = 0;i<selectedRows;i++)
  {
    ds->Bookmark = dgData->SelectedRows->Items[i];
    //Moves data cursor to selected row.

    ShowMessage("Wake up, Herman. I found another one!");
    //Perform your operations here.
  }
```

December 1997

The right list for the right job

by Kent Reisdorph

Storing a list of objects for later reference is a common programming technique. For example, a paint program might store a list of drawing objects so it can reconstruct the drawing on demand. A game program might store a list of player moves so the game state can be restored or replayed. Or, perhaps you just want to store a list of strings to be used for error messages. Whatever your needs, VCL has classes you can use to handle your lists. In last month's article "Implementing a Recent Files List," we put the TStringList class to work. This month, we'll look at the TStringList and TList classes in depth. We'll show where you'd use each type of list and how to add items to and retrieve them from the list.

Container classes

Classes that store objects (instances of a class) are often called *container classes*. These classes do a lot of work for you behind the scenes. The primary task a container class performs is managing memory. Let's first look at the traditional C++ array; then we can compare it to container classes.

Out with the old...

Consider a regular C++ array of objects. For example, say you wanted to maintain a list of 20 strings. The code would look like this:

```
String* MyStrings[20];
MyStrings[0] = new String("Hello There");
// etc.
```

Here you have an array of AnsiString pointers. You must assign a valid pointer to each item in the array and you must be sure to delete the memory associated with each element of the array when you no longer need the array. These requirements lead to a common programming error: Since you may not use the entire array, you may accidentally try to delete an object that doesn't exist. In order to prevent such an occurrence, you'd first have to set each element of the array to 0, then check for a valid value before deleting the object. The code looks something like this:

```
String* MyStrings[20];
```

```
for (int i=0;i<20;i++)
  MyStrings[i] = 0;
MyStrings[0] = new String("Hello There");
// Add more strings
// later when we're done with the array...
for (int i=0;i<20;i++)
  if (MyStrings[i]) delete MyStrings[i];
```

It's a bit cumbersome, but certainly not a major problem. A bigger issue lies in the static nature of our example array. What if you've misjudged the number of strings you need? You may suddenly find yourself needing 25 strings, or 30, or 100! However, the array size is limited to 20 strings--so you're stuck. You'll have to rewrite your code to allow for more strings.

That problem brings up another issue: If you allocate space for 100 strings but you only use 20, then you're wasting memory. Clearly, using regular arrays is not the best method when the size of the array may vary.

...in with the new

Thanks to the advent of object-oriented programming, there's a better way. Earlier, we said that one of the major roles of a container class is memory management. The VCL container classes dynamically allocate and free memory for the array as needed. As a result, the container is able to grow and shrink as items are added and removed. You can add items without fear of a memory overrun, and the burden of memory management is largely removed from your shoulders. Here's how the process works. VCL container classes have an initial capacity. No memory is allocated when the container is initially created. When you add the first item, the container allocates enough space for four objects (actually, since the containers store only pointers to objects, the space allocated is for four *pointers*, not four actual *objects*). When you add the fifth object , the container allocates space for another four objects. The amount by which the container grows is called the *delta*.

Now events get more complicated. When you add the ninth object, the container again allocates more memory--but this time it allocates enough memory for an additional eight objects. Finally, when you add the seventeenth object, the container allocates enough memory for an additional 16 objects. From this point on, the delta remains at 16 and the container will grow by that amount each time the current capacity is exceeded.

This rather peculiar architecture is designed to save both time and memory. It saves memory initially because only small chunks of memory are allocated--an efficient mechanism in terms of memory, but costly in terms of processor time. Each reallocation of memory requires a

certain amount of overhead; so, if you're adding hundreds of items to a list, the number of allocations required is time consuming. Once the array size grows to 17 elements, then additional memory is allocated in 64-byte chunks (16 pointers times 4 bytes per pointer). The actual process of inserting elements speeds up, since the reallocation of memory only happens every sixteenth add operation rather than every fourth. If you know beforehand exactly how many elements your list will contain, then it's best to set the initial size before adding elements. We'll get back to that subject in just a bit.

Personally, I'd prefer a container in which I can set the delta and initial capacity myself. It's easy enough to derive a class from TList and add that functionality, so I won't complain too loudly.

Another feature that a container class adds is the ability to easily manipulate the list. List manipulation includes tasks like adding elements, inserting elements, removing elements, and clearing the list (removing all elements at one time). The container class keeps track of indexes and adjusts them as necessary when you add, insert, or delete items.

VCL list classes

The two primary VCL container classes are TStringList and TList. The TStringList class, as its name implies, allows you to maintain a list of text strings. The TList class, on the other hand, lets you store any type of object in a list. For example, you might want to store a list of TPoint objects, a list of TBitmap objects, or a list of objects of your own creation. These two list classes aren't derived from a common base class, but they have several common methods, as described in Table A.

Table A: List class common methods

Method	Description
Add	Adds an item to the end of the list
Clear	Clears all items from the list
Delete	Removes an item from the list
Exchange	Swaps the position of two items
Insert	Inserts an item at a specified index
Move	Moves an item from one location in the list to another location
Sort	Sorts the list

In addition to these common methods, TList and TStringList each have a Count property. You can read the value of this property any time you need to know how many items are in the

list. Now, let's take a quick look at these classes individually.

String lists: TStringList

Programmers often need to maintain a list of strings, and VCL provides the TStringList class for this purpose. Many programmers just starting out with C++Builder commit a common programming error: They attempt to create an instance of the TStrings class. In fact, TStrings is the base class for TStringList and is an *abstract base class*. You can't create an instance of an abstract base class--you must create an instance of a class derived from the base class. In this case, you must create an instance of TStringList, not TStrings. Once you've created an instance of TStringList, you can begin adding strings, as follows:

```
TStringList* List = new TStringList;
List->Add("String One");
List->Add("String Two");
// etc.
```

You can then reference a particular string in the list by its index number:

```
Label1->Caption = List->Strings[10];
```

The list is 0-based, so the first item in the list is at index 0, the second is at index 1, and so on. String lists can be sorted or not sorted as determined by the Sorted property. By default, string lists aren't sorted. The Duplicates property is a Boolean property that determines whether the string list will allow duplicate entries. The Strings property allows access to the strings held in the container. To access a particular string, you'd use the syntax from the proceeding example. The Text property, in contrast, will return all the strings in the list as a single string. The TStringList class has one other important property--Objects--which you can use to store additional data associated with each string. We'll return to this subject after we've discussed the TList class, since there's a correlation between them.

TStringList has quite an array (no pun intended) of methods. This article isn't meant as a reference to the list classes, so I won't cover each method. Instead, I'll refer you to Table B, which contains some of the most useful methods.

Table B: Important TStringList methods

Method	Description

AddObject	Adds both a string and a pointer to additional string data
Find	Searches a sorted list and returns an index to the location the string will occupy when added to the list
IndexOf	Searches a list (sorted or unsorted) for the first occurrence of a particular string
LoadFromFile	Loads the string list from a text file
SaveToFile	Saves the string list to a text file

Of these methods, perhaps the most powerful are LoadFromFile() and SaveToFile(). These methods allow you to quickly load and save the contents of a string list using simple text files.

By now you may have figured out that the TStringList class is used throughout VCL. The ListBox, Memo, ComboBox, and RadioGroup components all use some variation of TStringList to store their data (as do many other VCL components). Actually, these classes use some derivation of the TStrings class, but it all looks the same to you and me.

The TStringList class is handy for storing lists of strings. It suffers somewhat from poor performance if you have many strings to add, but all in all, it's very useful.

TList, the all-purpose list class

The TList class stores a list of any type of data--you can store pointers to VCL objects or pointers to your own classes The TList class maintains an array of pointers (four-byte values). It's up to you to decide what to store in those four bytes. The Add() and Insert() methods take a void*, so you can pass a pointer to any type of object you like. Later, when you want to retrieve the values, you'll have to cast the pointer to the type of object you stored (more on that in just a bit). The most important TList property is Capacity, which contains the current maximum capacity of the list. This value isn't an upper limit, because the list can always be expanded if needed. The importance of the Capacity property is most obvious when you're creating large lists. Earlier we talked about the... er ... silly (for lack of a better word) allocation algorithm used by the VCL list classes. This algorithm tends to be very slow when working with large lists because all those memory reallocations are very costly in terms of speed. To avoid this slowdown, you can set the Capacity property to a desired size prior to adding any elements to the list. Memory for the list will be allocated when you set the Capacity property and won't be reallocated until that capacity is exceeded. If you're going to need a large list, then you should always set the capacity immediately after creating the list, as follows:

```
TList* MyList = new TList;
MyList->Capacity = 1000;
MyList->Add(new TMyObject);
// etc.
```

Doing so will prevent all the reallocations that would be necessary if you didn't set the capacity. The previous code snippet illustrates one way of adding objects to a TList container. You can add any type of object in this manner.

Adding objects to the list is one thing, but getting them out again is another. In order to retrieve the stored objects, you'll have to cast from a void* to the type of object you stored. For example, let's say your list contains pointers to a class you created. The code to extract an object from the list would look like this:

```
TMyObject* O = static_cast<TMyObject*>(MyList->Items[0]);
// Now you can do something with `O'
```

Notice I used static_cast to cast the void* to a TMyObject*. I'm a firm believer in the C++ casting operators. However, static_cast is not a typesafe cast--so in this case, there's little value in using static_cast over the old C-style cast. We could have written the above example as:

```
TMyObject* O = (TMyObject*)MyList->Items[0];
```

The net result is the same. The moral here is that you need to be sure what type of object your list contains when performing casts--performing a non-typesafe cast can have undesirable results if you cast to the wrong type.

Do you delete or do I?

It's important to note that the TList class doesn't "own" the objects it stores. In other words, you're responsible for freeing the memory associated with each object in the list. The VCL documentation is somewhat confusing on this issue. Depending on how you read the documentation for Tlist, you might conclude that TList will free the memory associated with each object in the array when the list itself is deleted. This is *not* the case. It's *your* responsibility to delete all the objects in the container before deleting the container itself. The same is true if you delete a particular item from the list--you must delete the object. In short, don't assume anything. A typical cleanup operation for a list might look like this:

```
for (int i=0;i<List->Count;i++) {
  // Cast to the correct type
  TMyObject* mo = (TMyObject*)List->Items[i];
  // Delete the object and set its pointer to 0
  delete mo;
  List->Items[i] = 0;
}
// Call Clear() to free memory for the pointers
List->Clear();
// Delete the TList object
delete List;
```

This code is necessary since the list stores only void pointers. The list doesn't know what type of objects it contains, so it can't properly delete the objects.

Back up a bit…

Earlier, we mentioned the TStringList class's Object property--an indexed property that stores additional information along with a string. You could, for example, store a list of strings and a corresponding object such as a pointer to a TBitmap instance. You can use the AddObject method to store both the string and the object at the same time:

```
TStringList* MyList = new TStringList;
Graphics::TBitmap* bm = new Graphics::TBitmap;
String s = "bitmap1.bmp";
bm->LoadFromFile(s);
MyList->AddObject(s, bm);
```

The Object property stores TObject pointers, so you'll need to cast the pointer back to a particular type when you use the object, as follows:

```
bm = dynamic_cast
  <Graphics::TBitmap*>(MyList->Objects[0]);
Canvas->Draw(x, y, bm);
```

As with TList, don't make any assumptions about the objects stored in TStringList's Object property with respect to ownership. You have the responsibility of deleting the objects when

you're done with the list.

One more thing...

TList and TStringList aren't the only game in town. The Standard Template Library (STL) is considered the default C++ library for container classes, and it has many types of container classes. The STL classes provide support for arrays, deques, queues, stacks, and more. You can easily adapt the basic STL containers to create other types of lists, such as trees and hash tables. If you plan on porting your applications to other C++ platforms, then you should consider using STL for your container class needs. STL comes with C++Builder, so you already have this resource at your disposal.

Conclusion

Programmers frequently use lists and arrays. Wise use of lists can save you programming time and will make your application more system-friendly. Use TStringList for lists of strings and TList for lists of other types of objects. And, don't forget about the STL containers. If you're educated about the options available, you can make informed choices about the types of containers to use in your applications.

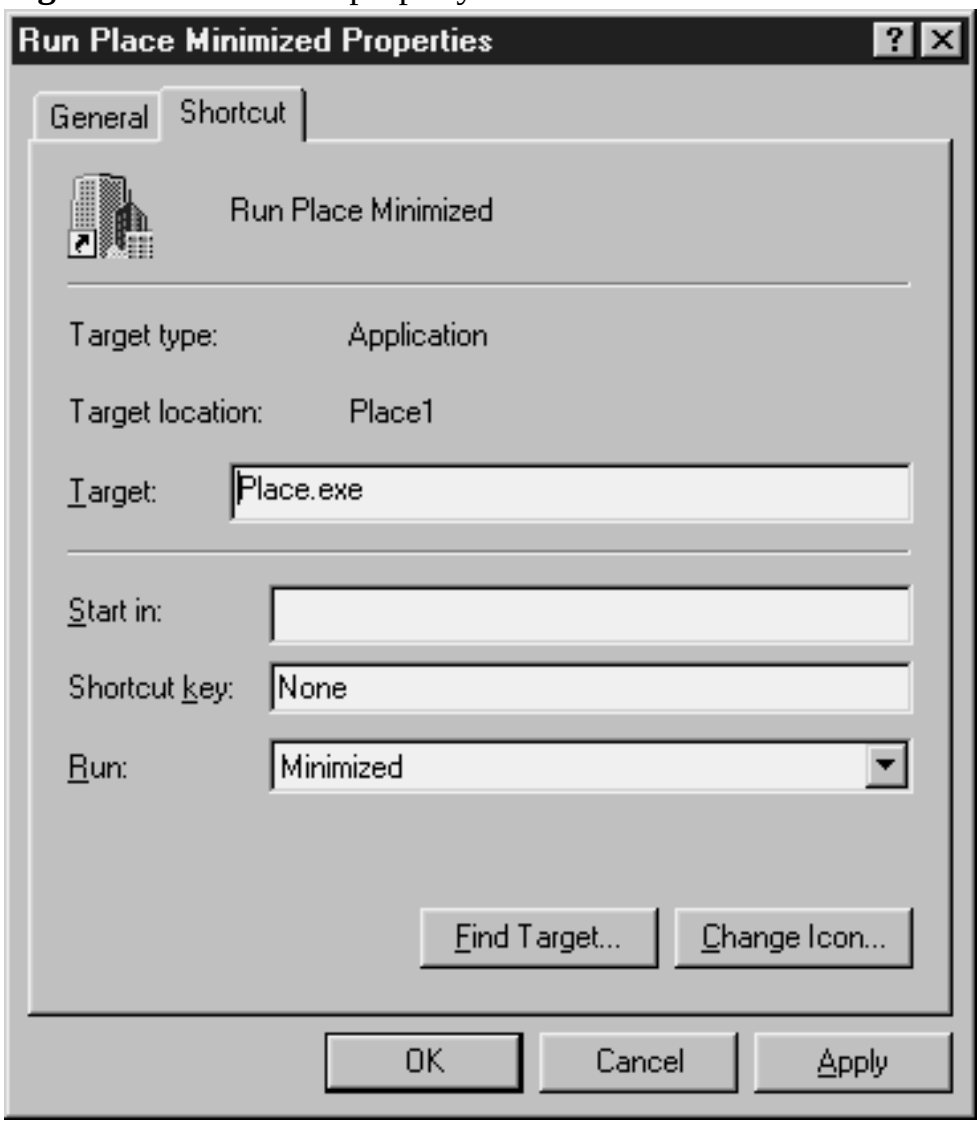
December 1997

Starting your application minimized

by Mark G. Wiseman

I want you to try something before your users do. First, create a Windows shortcut to an application you've written with C++ Builder. Then, right-click on the shortcut, choose Properties from the speed menu, and select the Shortcut tab in the resulting dialog box. In the Run dropdown box, select either Minimized or Maximized, as shown in Figure A.

Figure A: Set the Run property to Minimized.



Now run your application using the shortcut. Did your application's main window appear minimized or maximized? No? Then read on. In this article, we'll explain why your application didn't appear in the proper state and how you can easily work around this problem.

The bug and the hidden window

Borland constructed two obstacles in the VCL that make it difficult to run an application minimized or maximized. The first is a bug in the VCL's start-up code. Windows passes the parameter nCmdShow to applications through the WinMain() startup function. This parameter, an int, specifies how your main window or form should be shown to the user. Borland hard-coded this parameter with the value SW_SHOWDEFAULT--and that's the bug. The code in this article works around the bug, and should continue to work after Borland fixes the bug. Working around this bug will allow your programs to start maximized. To make your application start minimized, you'll need to surmount the second obstacle.

Borland included the following obstacle in the VCL on purpose: Every VCL-based application is a hidden window. The VCL makes Windows, the operating system, think this hidden window--rather than your main form--is your application's main window.When a user minimizes your application, this hidden window is minimized and the VCL hides your main form. Your main form is never actually minimized. Borland has tried to ease your pain by including methods and events in the TApplication class such as Minimize() and OnMinimize. We'll use another event from Tapplication--OnRestore--in our code. Let's see how to use our workaround.

Working around

Listing A includes all the code our sample project needs. (You can download our project files from

www.cobb.com/cpb

as part of the file dec97.zip; click the Source Code hyperlink.) To begin, create a new project with a single form named MinMaxForm. Add an OnCreate event named FormCreate() to the form, as shown in Listing A. FormCreate() will do nearly all the work.

Listing A: TMinMaxForm

```
//---------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------
#pragma resource "*.dfm"

TMinMaxForm *MinMaxForm;
//---------------------------------
__fastcall
TMinMaxForm::TMinMaxForm(TComponent*
```

```
   Owner) : TForm(Owner)
   {
   Application->OnRestore = OnRestore;
   }
//---------------------------------
void __fastcall
TMinMaxForm::FormCreate(TObject
   *Sender)
   {
   int cmdShow = System::CmdShow;

   if (cmdShow == SW_SHOWDEFAULT ||
      cmdShow == SW_HIDE)
      {
      STARTUPINFO startupInfo;
      GetStartupInfo(&startupInfo);

      if (startupInfo.dwFlags &
         STARTF_USESHOWWINDOW)
         cmdShow =
            startupInfo.wShowWindow;
      }

   if (cmdShow == SW_MINIMIZE ||
      cmdShow == SW_SHOWMINIMIZED ||
      cmdShow == SW_SHOWMINNOACTIVE)
      {
      ::ShowWindow(
         Application->Handle,
            SW_HIDE);
      ::ShowWindow(
         Application->Handle,
            SW_MINIMIZE);
      Application->ShowMainForm =
         false;
      }
   else if (cmdShow == SW_MAXIMIZE ||
      cmdShow == SW_SHOWMAXIMIZED)
      WindowState = wsMaximized;
}
//---------------------------------
void __fastcall
   TMinMaxForm::OnRestore(TObject
      *Sender)
```

```
      {
      Visible = true;
      }
//---------------------------------
```

First, you declare a cmdShow variable and set it equal to System::nCmdShow. This should be the value passed into your program by Windows--but due to Borland's bug, it will be SW_SHOWDEFAULT or SW_HIDE. You check the value here so the program will still work when Borland fixes the bug.If cmdShow is equal to SW_SHOWDEFAULT or SW_HIDE, the code checks the application's STARTUPINFO structure. You fill in startupinfo--an instance of this structure--by calling the Windows API function GetStartupInfo(). If the STARTF_USESHOWWINDOW flag is set in startupinfo.dwFlags, then you set cmdShow equal to the value in startupInfo.wShowWindow; otherwise, you ignore the STARTUPINFO structure.

At this point in the code, you've worked around the bug. Now it's time to tackle the hidden window. If the value of cmdShow indicates that you need to minimize your window, the code hides the hidden window and then minimizes it. Next, you tell TApplication not to show the main form. This step is equivalent to setting the main form's Visible property to false.

Why do you need to hide the hidden window? Actually, the "hidden" window isn't hidden at all--it's visible, but its width and height are set to zero, so you can't see it! If you didn't hide it first, you'd see the Windows zoom animation when the window minimized.

Not enough zoom

You may have noticed that the animation Windows displays when minimizing a window doesn't work with the main form in C++Builder programs. This flaw is caused by the hidden window problem we describe in the accompanying article. If you'd like to make your main form animate when minimizing, see the article "Zooming Main Forms Under Windows 95" in the October 1996 issue of The Cobb Group's *Delphi Developer's Journal* (**www.cobb.com/ddj**).

If cmdShow indicates that you should maximize the main form, the code simply sets the WindowState property to wsMaximized. There's one final detail to address. Remember, if cmdShow has you minimize your main form, you're really minimizing the hidden window and hiding the main form. If you then try to restore the main form, the hidden window will be restored (and still invisible) but the main form won't be made visible. To make everything work as expected, create an OnRestore() function in TminMaxForm; in TminMaxForm's constructor, assign the function to Tapplication's OnRestore event. In the OnRestore() function, you simply make sure the main form's Visible property is set to true.

Make life easier

Other examples that work around the problem of starting a C++Builder application minimized modify the WinMain() function code in the project source file. As a result, every time you create a new C++Builder application, you must remember to modify this code. I encourage you to make life easier. Use the code in this article and save TMinMaxForm into C++Builder's Repository. Then, you can use TMinMaxForm as the base form for all your applications, and your C++Builder programs will work as your users expect them to.

Your users will also expect your application to remember its size, position, and state between uses. In a future article, we'll show you how to enhance TMinMaxForm to do just this.

December 1997

Reconnoitering resources

by Sam Azer

A few questions seem to pop up over and over again on The Cobb Group's CPB-Thread list. For example, "Is there another resource editor for C++Builder?" The answer is, "Many--but you don't often need them." (Participation on CPB-Thread is a useful way to keep up with C++Builder issues and learn about its diverse range of features. To subscribe, visit **www.cobb.com/cpb**.)

If you're using Borland C++ 4.5, you already have a copy of Resource Workshop. (If you use version 5, it's built into the IDE.) In fact, any editor that will produce RC files will do--and such editors are readily available. For example, search a Delphi Super Page mirror for Triplex++, a freeware package by John Howe. Triplex includes a stand alone resource editor and a great Grep utility.

However, look closely at the resources you use most often, and you'll find you already have the tools you need--C++Builder does most of the work for you. It can edit dialog boxes and menus better than any other utility. The image editor is fairly basic, but it can create bitmaps, cursors and icons. That leaves string tables, version information, user defined resource objects, and such items as fonts and audio wave files. You'll find that these remaining resources are easy to create manually, if you know how.

Included with C++Builder is the RCC32 resource script compiler, which can compile any standard RC file. You don't have to do anything special to use the resource compiler--simply add any resource script to your project and C++Builder will automatically compile it and bind it to your DLL or EXE.

In this article, we'll make and use some resources. We'll start with a bit of background information, then we'll build a string table and add version information. Finally, we'll add a wave and play it. You can download our sample files from **www.cobb.com/cpb** as part of the file dec97.com; click the Source Code hyperlink. (Note that the resource part was so easy that the example was too boring--I jazzed it up with a splash screen, volume controls, and a demonstration of the three different ways to use PlaySound().) Let's get started.

An introduction to resources

Resource files are a convenient way to gather the various little objects a project needs: cursors, icons, bitmaps, meta files, MIDI and wave sounds, to name a few. You can also store user-defined objects and data as resources. Windows takes care of everything--memory allocation, loading, and unloading are all handled for you. In many cases, all you do is specify what resource you want to use. A *resource script* is a simple text file that tells Windows about your resources. Each resource has an ID--either a unique number or a string to identify the resource. Small resources like cursors can be converted to text and

saved with the script. However, most resources are prepared using specialized tools that have nothing to do with programming. For example, you might create a small video for your application using Corel's PhotoPaint. It makes no sense to convert the video into text and store it in the resource file--it's easier to put the video's filename into the script. Later, the resource compiler will bind that file to your EXE or DLL.

Resource scripts support several preprocessor directives: #define, #include, #if/else/endif, #ifdef/ifndef, and #undef are all available. If you want to add some notes to your script, you can use the old C-style comments (/*...*/).

You can identify resources using a name, but the code runs faster if you use a unique numeric ID, instead. It's common practice to #include in your project and in your script a header that defines ID numbers. Doing so makes it easier to change the ID numbers later (in case of a collision with another resource ID).

String resources

You can embed any number of string tables within your resource script. Each string table keeps related strings together to help Windows better manage available memory. A string table, as with all resources, has a simple structure. Here's a resource header file:

```
/* resource header file */

#define IDS_NOWAVE    1001
#define IDS_SOUNDOFF  1002
```

And the following lines illustrate a resource script:

```
#include "resources.h"

Welcome Wave "welcome.wav"
Splash  Bitmap "splash.bmp"

stringtable
begin
    IDS_NOWAVE, "No wave devices"
    IDS_SOUNDOFF, "Sorry - Sound Disabled"
end
```

As you can see, each string resource consists of nothing more than an ID for each string, a comma, and the string itself. To use one of these strings in your code, load it using the

LoadStr() function provided by VCL, as follows:

```
Edit1->Text = LoadStr( IDS_NOWAVE );
```

The LoadStr() function searches the currently executing task for a string table resource with the specified numeric ID. It returns an AnsiString result. If the resource isn't found, the function returns a null string.

Version information

You can precisely identify each product and executable file you release using version information. To do this, you need only include a version table in your resource script. The easy way is to copy an existing version script, then edit it. Listing A shows the version resource for our sample program.

Listing A: Version Information in Version.Rc

```
VERSIONINFO_1 VERSIONINFO
FILEVERSION 1, 2, 3, 4 /*file vers#, 4 parts*/
PRODUCTVERSION 1, 5, 6, 7 /* product vers#*/
FILEOS VOS_NT_WINDOWS32 /* WIN 32/NT */
FILETYPE VFT_APP /* use VFT_DLL for a DLL */
{
 BLOCK "StringFileInfo" /* Don't Touch This! */
 {
  BLOCK "040904E4" /* Don't Touch This! */
  {
   /* edit this for your app */
   VALUE "CompanyName", "Sam Azer\000\000"
   VALUE "FileDescription", "PlayResource Demo
     shows how to make a resource file by
     hand (and use the resources!)\000"
   VALUE "FileVersion", "1.2.3.4\000\000"
   VALUE "InternalName", "PlayRes\000"
   VALUE "LegalCopyright", "Copyright \
     251 1997\000\000"
   VALUE "OriginalFilename", "PlayRes.exe\000"
  }
 }

 BLOCK "VarFileInfo"
 {
```

```
  /* U.S. English, Windows ANSI code page */
  VALUE "Translation", 0x409, 1252
 }
}
```

Your file and product version numbers can consist of up to four parts. If you don't need that many, set the unused values to zero. C++Builder will only produce code that runs under Windows 95 or NT, so you don't have to change the FILEOS value. If your project output is a DLL, change the FileType from VFT_APP to VFT_DLL. Don't touch the two block headers--Windows looks for them. The two Translation values at the bottom are *Language* and *Code Page*; the values shown are for U.S. English under the U.S. ANSI code page for Windows. Other values are listed in the Win32 Reference.

After you've built your project, you can view the version information from the Windows Explorer. Use the Explorer to find your project's EXE or DLL file. Right-click on the file, choose the Properties option at the bottom of the speed menu, then click on the Version tab.

Other resources

The remaining resources you might want to bind to your application probably already exist in a file somewhere. Using a single line of text in a resource script, you can assign an identifier and specify the filename. Later, the resource will be bound to your project. The format of this line is as follows:

```
id type [PRELOAD] filename
```

The ID can be a unique resource number or a name. Type is a string that describes the resource. There are a few standard types, such as BITMAP, CURSOR, ICON, and FONT, or you can make up your own. The PRELOAD option forces Windows to load the resource right away, which can be handy if you want to avoid delays with an often-used resource. If you use the following code in a form's OnPaint method, it will load and draw the sample bitmap Splash.BMP:

```
TBitmap *Splash = new Graphics::TBitmap;
Splash->LoadFromResourceName( (int)HInstance,
"Splash" );
// Set the form size to match the splash image
ClientWidth = Splash->Width;
ClientHeight = Splash->Height;
Canvas->CopyMode = cmSrcCopy;
Canvas->Draw( 0, 0, Splash );
```

Note the use of HInstance, a global variable provided by VCL that gives the handle for the currently executing task. Windows searches that EXE or DLL file for the requested resource. The next line of code will play the sample wave file Welcome.WAV. Take a look:

```
PlaySound("Welcome", HInstance, SND_RESOURCE);
```

What about all those resources that Windows doesn't understand? For user-defined data, the standard resource type is RCDATA. If you want to include the data in the resource script, it will look like this:

```
MyResource RCDATA
Begin
   "this is a string\000" /* note the null */
   1,2,3,4              /* integers */
   0x01, 0x02, 0x03, 0x04 /* Hex ints */
   '01', '02 03 04'     /* Hex bytes */
End
```

Windows provides five functions to handle user defined resources. FindResource()finds any type of resource, LoadResource()brings it into memory, and LockResource()returns a pointer to it and makes it stay put. SizeofResource() returns the size of a resource in bytes.

Finally, you don't have to get all your resources out of the currently executing application--you can use the LoadLibrary() function to get an HInstance handle for any DLL or EXE. Check the Win32 Platform SDK help files for details. Note that if you get an error return value after calling one of these functions, you can use the GetLastError() function to find out what went wrong. Use VCL's SysErrorMessage() function to translate the information into English.

Wrap up

We've covered only the basics of resources in this article--it's a big topic and there's plenty more ground to cover. The main thing to keep in mind is that it doesn't take much effort to create or use resources. And, you won't require many extra tools--C++Builder comes with just about everything you need.

December 1997

Learning the ropes of TStringGrid

by Gerry Myers

On my current project, I needed a visual matrix to hold some data and to allow the user to make changes. I figured a grid component would do the trick nicely, so I clicked on the Additional tab in the component palette and selected the StringGrid component. When I dropped the component on the form, it presented a nice grid-like appearance and some fairly straightforward properties in the Object Inspector. As I was patting myself on the back for migrating to C++Builder and making my programming so much easier, I opened the VCL help file to read the information about TStringGrid--but I couldn't find any. At first I thought I'd typed in the wrong name in the Help Topics dialog box--but no. I even checked the paper reference manuals that shipped with C++Builder, but I still found nothing on TStringGrid.

I called the Borland Assist line and was told that, sure enough, they'd left several of the component help files out of the original VCL help file during the rush to get a first-release product to market. Fortunately, you can download an updated VCL help file from Borland's Web site, **www.borland.com**; look in the C++Builder section under Developer Support.

Now, don't stop reading just because you can get the new Help file that includes TStringGrid. In this article, I'll describe how to use some of TStringGrid's unique properties.

A grid by any other name

Not knowing about the undated VCL help file, I convinced myself that help files were for wimps anyway, and started digging into the TStringGrid class. I knew about the DBGrid component (a database control), so I checked its help information first. Although it was informative, it didn't help me with TStringGrid--TDBGrid and TStringGrid come from different branches of the hierarchical tree. I finally broke down and did a text search for *TStringGrid* in the C++Builder Include directory, to find the header file for the class. What I found was the Grids.HPP file in the \Include\Vcl directory. You can look for yourself if you're interested in the private, protected, and published items. I was concerned with the public section, along with TStringGrid's base class.

It turns out that TStringGrid is derived from TDrawGrid. And guess what? TDrawGrid *was* in the VCL help file. But although it was helpful to understand the properties, methods, and events of TDrawGrid (and indirectly TStringGrid), I still had to figure out how the public properties that were unique to TStringGrid worked. For that I had to turn back to the TStringGrid header file.

Cells

The TStringGrid constructor contains four public properties, as follows:

```
__property System::AnsiString Cells
  [int ACol][int ARow] = {read=GetCells,
  write=SetCells};
__property Classes::TStrings* Cols[int Index] =
  {read=GetCols, write=SetCols};
__property System::TObject* Objects
  [int ACol][int ARow] = {read=GetObjects,
  write=SetObjects};
__property Classes::TStrings* Rows[int Index] =
  {read=GetRows, write=SetRows};
```

The most straight-forward property is Cells[i][j], which you can use to read or write the string value of any cell. For example, you can set the value of every cell in the grid by performing a double loop like this:

```
for ( int i=0;  i<MyGrid->ColCount;  i++ )
  for ( int j=0;  j<MyGrid->RowCount;  j++ )
     MyGrid->Cells[ i ][ j ] = "n/a";
```

As the component name implies, this grid works with strings (AnsiString, to be specific). Therefore, you can either assign property Cells[i][j] a string, or the property can assign its string to another AnsiString object.

You may have noticed something peculiar (at least, it seemed backwards to me): The cells are indexed in a column major format, meaning that the first index is the column and the second is the row. You should be aware of this indexing, since it mixed me up a couple of times.

Rows and Cols

The Cells property is all well and good--but wouldn't it be easier, at times, to assign the value of an entire row or column in one statement? Luckily, TStringGrid provides the Rows[i] and Cols[i] properties for doing just that. These properties read and write objects of type TStrings (remember how Cells[i][j] reads and writes objects of type AnsiString). Let's touch briefly on the TStrings class. If you're familiar with the AnsiString class, you know that it can contain only a single string. The TStrings class is able to hold multiple AnsiString objects--you can think of it as an array of strings.

The Rows and Cols properties work with TStrings in such a way that the first string in the TStrings object goes to the first cell in that row or column; the second string goes to the next

cell, and so on. By using the Rows and Cols properties, you can avoid the double-loop needed when using Cells[i][j].

An added benefit of using the Rows and Cols properties is that you can download the TStrings class's strings automatically from a disk file. TStrings has a LoadFromFile(*filename*) method (and its cousin LoadFromStream(*stream*)). Each line in the text file becomes a different string in the TStrings object and hence a different cell in the grid. For example, the single statement

```
MyGrid->Rows[0]->LoadFromFile( "xyz.txt" )
```

will fill the first row of the grid with successive lines from the text file. Another helpful property of the TStrings class is CommaText, which allows the array of strings to be built from a single comma delimited string. For example, the line

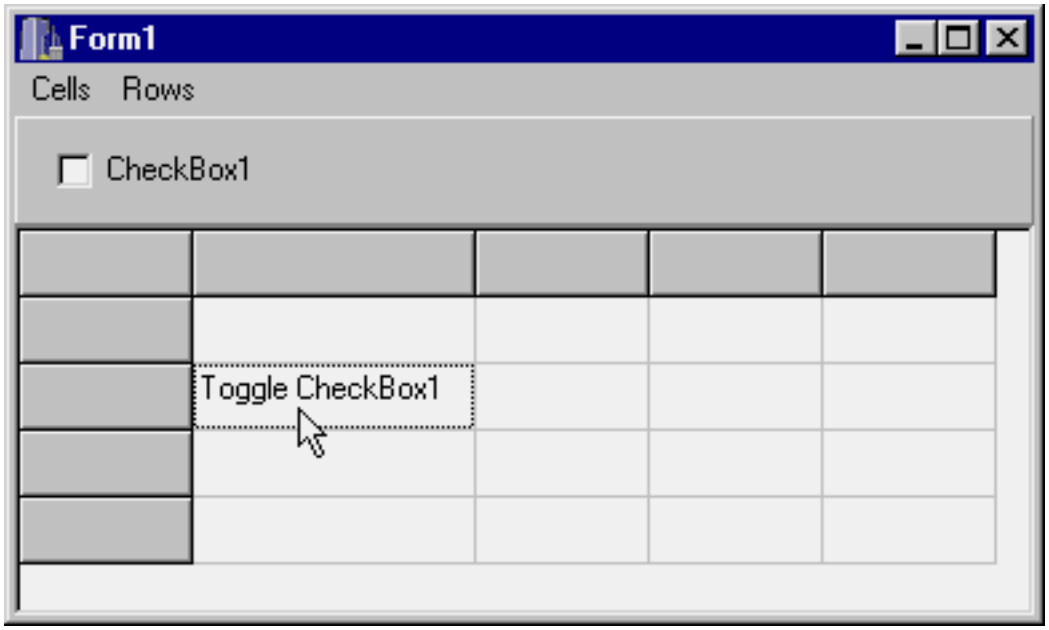
```
MyGrid->Rows[0]->CommaText = "MeasId,
  Description, Units"
```

will fill the first cell of row 0 with the string *MeasId*, the second cell of row 0 with *Description*, and the third cell with *Units*--all using a single statement.

Objects

The last public property unique to TStringGrid is Objects[i][j]. It's very similar to the Cells property, but instead of reading and writing strings, it reads and writes items of type TObject. You may recall that all VCL components are derived either directly or indirectly from TObject--if you look at any VCL component's hierarchy, you'll see TObject at the top. TStringGrid's Objects property lets you *associate* any VCL object with a grid cell. For example, let's say you have a CheckBox component on your form, as shown in Figure A, and you want to be able to toggle its visibility when a certain grid cell is selected.

Figure A: Selecting the first cell in the second row toggles the appearance of a check box.



To do this, you can write code like that in Listing A.

Listing A: Associating an object with a cell

```
void __fastcall TStringGridForm::FormCreate(TObject *Sender)
{
  // Set up the string and object association
  // for the cell in column 1 row 2. Text
  //string MUST be set before the object.
  StringGrid1->Cells[ 1 ][ 2 ] =
    "Toggle CheckBox1";

  // CheckBox1 is an existing component.
  StringGrid1->Objects[ 1 ][ 2 ] = CheckBox1;
}

void __fastcall TStringGridForm::StringGrid1SelectCell(
  TObject *Sender, long Col, long Row,
  bool &CanSelect)
{
  // Get object associated with selected cell.
  TCheckBox* cBox = ( TCheckBox* )
    ( StringGrid1-> Objects[ Col ][ Row ] );

  // If selected cell is associated with a
  // checkbox, toggle checkbox's visibility.
  if ( cBox ) cBox->Visible = !cBox->Visible;
}
```

First, the TCheckBox component and the text string are associated with the grid cell when the grid is created. (Note that a string must be loaded into the cell before the object is associated with it, or you'll get a runtime error.) Then, you set up an OnSelectCell event handler to toggle the checkbox's visibility. (You can download our sample project from **www.cobb.com/cpb** as part of the file dec97.zip; click the Source Code hyperlink.)

When you execute the application, the check box will appear and disappear each time you select the specified cell. You could use this approach to present the user with a matrix (grid) of pseudo-buttons for toggling the visibility of various controls.

Conclusion

The TStringGrid component offers four unique properties: Cells, Rows, Cols, and Objects. You can access individual cells in the grid by indexing the Cells property, or index entire rows and columns through the Rows and Cols properties. Indexing the Objects property lets you associate individual TObject components with any cell in the grid. In this article, we've discussed the major functionality of this grid class to the point that you should feel comfortable playing with it. As we've demonstrated, the TStringGrid component is quite versatile and easy to use.

Borland®

Shop

Products | Downloads | Services | Support | Partners | News & Events | Company | Developers

In this digital era, the source of competitive advantage for business lies in the creation of value.

Breaking News

Site Map | Search | Contact

International Sites

November 1997

Building components, part 3

by Sam Azer

In the September 1997 issue of *C++Builder Developer's Journal*, part 1 of this series looked at the basic elements of component building in C++ Builder. In part 2, we built a small example component. This month, we'll demonstrate how to add some polish to your component--finishing touches like writing a help file to document the component, creating custom property and component editors to make it easier to use, and adding an icon to set the component apart from others. This article is, in essence, three articles in one; so feel free to jump to the section(s) of interest to you.

Let's start with the most important task to perform once the component is working: writing a help file.

A custom help file

As I mentioned last month, the only way to avoid sitting in front of a debugger day and night is to have your project organized in your head before you start working. Doing so requires documentation. Once your component is ready for deployment, you can polish up the documentation to produce online, context-sensitive pop-up help. Making help files is no great joy, but the Microsoft Help Workshop compiler (HCW) that comes with C++Builder greatly simplifies the process. And, the C++Builder IDE makes integration of the help file virtually automatic. We'll take a quick look at the essential steps here.

Basically, you'll use your favorite word processor to prepare the text of your help file. But instead of chapters containing pages, you structure your help file as *books* containing *topics*. At the very least, your help file must contain a general overview and a topic for each new property and event your component publishes. Each topic can be as long or as short as you like--there are no physical page constraints. You can easily embed text, images, and simple tables. Don't expect to be able to do too much fancy formatting--the help compiler will choke on some of the more sophisticated features of the major word processors. Stick to simple tables, avoid all but the standard fonts, and try to use 16- or 256-color Windows bitmap (BMP) files. Another trap to avoid is the "keep (paragraph) with next" feature that's used for most headings in a normal document. HCW uses it to mark *non-scrolling* regions that are allowed only once at the top of a topic--so, don't use your favorite style sheet for your help file topics! Mark the end of each topic using a hard page break ([Ctrl][Enter] in Microsoft Word). HCW requires specific information about each topic in order to produce the final help file. You must title the main topics, provide key words and phrases for the index, and assign *Topic IDs* to allow for hypertext jumps. If you want the reader to be able to browse through your topics, you must also assign a *browse sequence* to tell HCW the order of the topics. You pass all this information to HCW by embedding it in *footnotes*. When you find a footnote marker in a book--like this*--you know that at the bottom of the page will appear a note with an asterisk in front of it. HCW reads your topics in much the same way. If it finds a footnote that uses a dollar sign--like this$--it takes the corresponding footnote text to be the title of the current topic. Let's look at the other

footnotes you'll need for each topic, along with the characters that identify them.

K--keyword

Keywords are listed in the index. You can have as many keywords or phrases as you like for each topic, separated by semicolons, as follows:

```
K TDBShortList; ComboBox; Short List
```

When a user selects a keyword from the index, the titles of all the topics containing that keyword are listed. The user can then view a topic by clicking on a title.

+--browse code

If you enable browse buttons in your help file, the user can press the Next or Previous button at the top of the help window to move from page to page. The sequence of the pages is determined by the browse code:

```
+ 001
```

This code is alphanumeric and is sorted as such--a very handy feature. You can start with browse sequence codes of 001, 002, 003, and so on; then, if you need to add topics later, you can give them codes of 001-01 and 001-02, rather than renumbering all the pages.

```
#--topic IDs
```

Each topic must have a unique number, like the following:

```
# 1001
```

You can use topic IDs in hypertext jumps. Mark your hypertext using the double-underline attribute, then add the topic ID you want to jump to (don't allow any space between the hypertext and the topic ID), as follows:

```
..the ListDefaults1001 property..
```

This example shows the topic ID in color for emphasis, but you should mark the topic ID with the *Hidden* attribute to indicate to the help compiler that it's a target and not regular text. After you've compiled your help file, the hypertext will appear in green with a single underline:

```
..the ListDefaults property..
```

When you click on it, the help engine will jump to the specified topic.

A--A-Keyword

If you select a property or event from the Object Inspector and press [F1], the IDE will search all the A-Keywords and display the matching topic. If more than one match is found, the user will be able to choose from a list.

For a component, the A-footnote must be formatted as the name of the component followed by _Object, a semicolon, and the name of the component again, as follows:

```
A TDBShortList_Object;TDBShortList
```

Properties and events are formatted in much the same way. Here are the A-footnotes for the ListDefaults property and the OnAddNew event of

```
TDBShortList:

A TDBShortList_ListDefaults; ListDefaults_Property;
 ListDefaults

A TDBShortList_OnAddNew; OnAddNew_Event; OnAddNew
```

Compiling your help file

Once you've prepared your topic file with the proper footnotes, you must export it to *Rich Text Format* (RTF). If you get frustrated with the RTF-related idiosyncrasies of your word processor, you're not alone. The HCW documentation explains the RTF codes--they're not much more complicated than HTML.

The next step is to make your project file. Start by running HCW.EXE, which resides in the CBuilder\Help\Tools directory. (It helps to put a shortcut to this program into your Start Menu.)

Microsoft's Help Workshop is very easy to use. If you've used previous versions of Microsoft's HC, you'll be impressed with the GUI interface and the new features. In fact, it's so simple to use that you don't really need any instructions here. However, it's worth mentioning that HCW has a nasty bug that causes it to crash if you don't define a main window for your project. So, start by clicking on the Windows... button and adding a window called *main*. Make a Contents file and a Project file, then compile everything to produce your help file.

After compiling your project, you'll have a working TDBShortList.HLP file. Integrating it with the C++Builder IDE is easy. The documented procedure is to use the OpenHelp tool, but that didn't work on my system--my OpenHelp.INI file was corrupt. An alternate technique that's guaranteed to work is to copy your help and contents files (TDBShortList.HLP/CNT) to the CBuilder\Help directory; then, add the following two lines to your BcbHelp.CFG file (in the same directory):

```
:INDEX TDBShortList=TDBShortList.hlp
:LINK TDBShortList.hlp
```

You must delete the Bcb.GID and FTS files to force WinHelp to make new ones. The next time you try to open Bcb.HLP, WinHelp will read BcbHelp.CFG. It will then create a Bcb.GID file that includes references to the topics in TDBShortList.HLP.

To test the keywords in your help topics, click on Bcb.HLP from the Explorer and type TDBShortList in the Index. You should find it and be able to jump to that topic directly. To test your A-Keywords, click on TDBShortList in the C++Builder component palette, then press [F1]. Doing so should give you the same results. Repeat this procedure for each property and event.

Property and component editors

Once you've finished your documentation, your component is complete. However, you may find that it publishes some properties that aren't easy to set. Or, you may want to write a wizard to walk a programmer through the configuration. The C++Builder IDE can be extended to handle either case.

At the moment you drop your component on a form, the form editor scans an internal list of registered component editors for one that can be used with your component. The form editor instantiates (creates and initializes an instance of) this editor, which then repeats the same process--it scans a list of registered property editors and instantiates one for each property your component publishes. The TDefaultEditor and TPropertyEditor classes implement the behavior of a component and property editor, respectively. In both cases, you often can implement the customized functionality you want by deriving a new editor class that overrides only four simple methods! Then, you can register your new editor classes with the C++Builder IDE. To demonstrate how this process works, we're going to build two property editors. We'll use a TOpenDialog for the ListName property, to allow the programmer to browse for the list file. For the ListDefaults property, we'll make a new dialog box to act as a simple list editor. After that, we'll build a basic component editor. Let's get started.

Property editors

First, an important note: Due to limitations in the 1.0 release of C++Builder, the compiler will register editors only for VCL-derived properties. To see this problem, try compiling a line like this one:

```
void *p = __typeinfo(AnsiString);
```

The compiler insists that you use a VCL-derived class. In practice, this problem forces all properties requiring a new editor to use a wrapper. Also, because a VCL-derived class must be instantiated using the new operator, the property itself must be a pointer to a wrapper class. As we mentioned, you must override four basic methods to implement most basic property editors. They are as follows:

```
TPropertyAttributes __fastcallGetAttributes(void);
AnsiString __fastcall GetValue(void);
void __fastcall SetValue(AnsiString v);
void __fastcall Edit(void);
```

GetAttributes() tells the Object Inspector about the characteristics of your property editor. There are a number of attributes you can use--to see a list, run a keyword search from the Help menu for GetAttributes.

The Object Inspector calls Edit() to activate your editor. GetValue() and SetValue() are simple functions to convert your property to and from text form for display in the Object Inspector. The only way your editor can actually get access to the value is through one of the methods provided by TPropertyEditor. Again, due to temporary restrictions in the 1.0 release of C++Builder, the primary method to use is GetOrdValue(). It was designed to return an integer value, but it can return a pointer as well. The pointer can then be cast into the desired type and dereferenced. **Listing A** shows the code for the ListName property and editor. It's a little kludgy, but it still works. I hope there will be a void * GetPtrValue() method in the 2.0 release--and no restrictions on the __typeinfo() function. Designer is also a member of the TPropertyEditor class. You must call the Modified() method of any property editor's Designer member after making changes to the value of the property. This is the only way to alert the Object Inspector to the fact that changes have taken place.

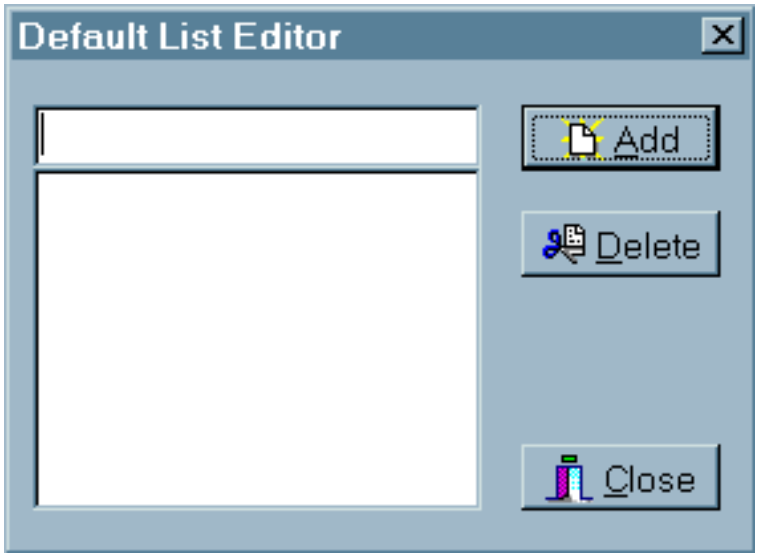
The RegisterPropertyEditor function, as you can see in **Listing A**, takes four parameters. The first is a pointer to the type information for the property to be edited. The last is a pointer to the editor class for the specified property. The remaining two are optional. If you specify a class and property name, the property editor will be used only for that class and property. Otherwise, it's used for all instances of the property type.

After you've added the required code to TDBShortList, an ellipsis will appear at the end of the ListName property. Clicking the ellipsis will open a TOpenDialog box. If the modal result of

that box is true, the new ListName is copied to the property.

The ListDefaults property is a little more complicated. Choose File | New Form... from the main menu to create a blank form. Populate it with an edit field, a list box, and Add, Delete, and Close buttons. Then, write the code that adds the text from the edit field to the list box and that deletes the currently selected list-box item. **Figure A** shows my version of this form.

Figure A:*We designed this CommaText List Editor form.*



The code for most of the CommaText list editor is found in **Listing B** on the following page. You'll need to write the handlers for the Add and Delete buttons, but the code is typical for a dialog box. The twist is that it's easier to register the control if you copy the component header and code into the respective form files and use them as your new component files. You'll eventually want to use more dialog boxes and resources, at which point you'll be forced to use additional files. When that happens, use the link and resource pragmas to tell C++Builder about the files.

The only thing that's new in **Listing B** is the GetEditLimit() method of TPropertyEditor. By default, this method returns a maximum property length of 256 characters; you must override it if you need more.

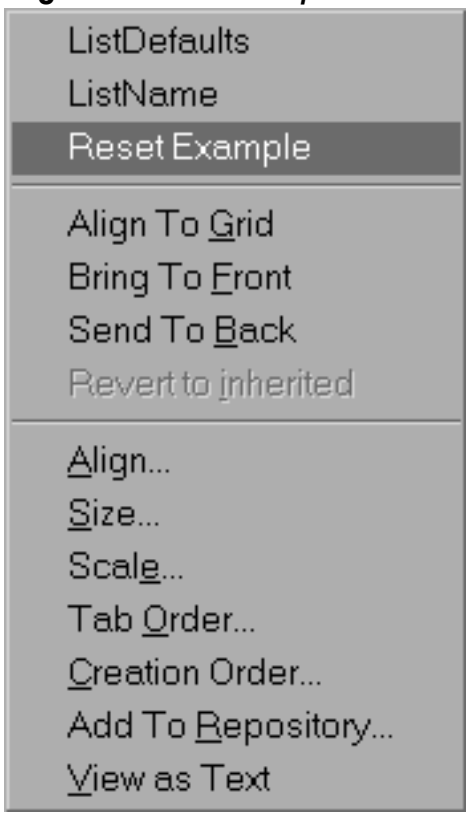
A component editor

Component editors let you perform operations on a component as a whole. Again, there are four basic methods to override in most cases. This time, though, a pointer to the component is passed as a parameter to the component editor's constructor--which helps make the code more clear.

As we mentioned earlier, the form editor instantiates a component editor at the moment you drop your component on the form. At that time, the form editor also asks the component

editor how many tools it provides. Your component editor can actually offer many support tools to the programmer; they're listed at the top of the speed menu that appears when you right-click your component from the form editor. Each of these tools is called a *verb* in VCL-speak. **Figure B** on the next page shows the speed menu produced by the component editor code from **Listing C** on page 7.

Figure B:*Our component editor generates this speed menu for TDBShortList.*



You must override the GetVerbCount() method of TDefaultEditor to return the number of verbs your editor offers. You must also override the GetVerb(i) method to return the strings used in the speed menu for each verb--the method is called once for each verb, where i is 0 for the first verb, 1 for the second, and so on. Finally, to run your verbs, you must override the ExecuteVerb() method.

If the programmer double-clicks on the component, the form editor calls ExecuteVerb(0)--the first verb. If a component editor is selected from the speed menu, ExecuteVerb(i) is called with i representing the verb to execute in the same order as returned by GetVerb(i). Again, the pointer to your component is passed in the constructor for your component editor. The constructor also gets a pointer to a class called TFormDesigner, which has another method called Modified(). As before, you must call that method if your component editor changes any properties. TDefaultEditor has an Edit() method that loops through the list of properties for a component. For each property, Edit() calls EditProperty(). As you can see from the end of **Listing C**, the first two verbs are handled by calling the property editors for ListDefaults and ListName. The second parameter to EditProperty() is a flag that tells Edit() to stop looping through the list of properties. The last parameter is a flag that can be used to destroy the

property editor--you may want to avoid that one!

A custom icon

Well, at this point you've done just about everything. The component is available to the programmer, it's documented, and it's easy to use. There's just one last thing to do: Make a distinctive icon to set your component apart from the others.

Creating the icon is very easy: Simply make a RES file containing an icon and give it the same name as your component. Let's work through the steps involved.

Start the Image Editor and create a new resource file by choosing File | New | Resource File. Then, select New Icon from the Resource menu. The dialog box that opens doesn't give you many choices--it specifically lacks the 24- by 24-bit Icon option that the *Component Writer's Guide* says you'll need. No problem; go for the closest thing--32 by 32 bits with 16 colors. It turns out that C++Builder will try to fit whatever you give it, so this option will work. Click OK, and you'll have an icon named Icon1.

Right-click on the icon and select Rename from the speed menu, then assign the icon the same name as your component. Be sure to use all uppercase: TDBSHORTLIST, for example. Next, right-click on the icon and select the Edit option to edit your new icon.

Personally, I'm no artist. Even if I was, 32 by 32 bits isn't much space to work with. I put together an icon based on a Notepad document, as shown in **Figure C**--it's not fancy, but it will do.

Figure C:*We created this icon for TDBShortList.*

Finally, save the resource file by using the name of the component. Place TDBShortList.RES in the same directory as the other component files, then rebuild the component palette by choosing Component | Rebuild. Your custom icon will now appear in place of the default icon.

Wrap up

If you've been following the three parts of this series, by now you should be feeling pretty confident about writing new components. Our example, TDBShortList, demonstrates the basic concepts involved in creating a new component by extending an existing one. In most cases,

the information and techniques presented in this series will be enough to get the job done. There's more, though--be sure to look for additional details in future issues of *C++Builder Developer's Journal!* **Listing A:** *The ListName property and editor*

```
// VCL wrapper class for an AnsiString
class TText : public TPersistent
{ public:
  __fastcall TText() : TPersistent() {};
  __fastcall ~TText() {};
  PROP_PUBLISHED(AnsiString,Text,nodefault);
}
.
.
// from class TDBShortList, change ListName to:
PROP_PUBLISHED(TText*,ListName,nodefault);
.
.
class TTextProperty : public TPropertyEditor
{ public:
  virtual AnsiString __fastcall
    GetValue(void) { return
    ((TText*)GetOrdValue())-&gtText; };
  virtual void __fastcall
    SetValue(AnsiString v) {
    ((TText*)GetOrdValue())-&gtText = v; };
  virtual TPropertyAttributes __fastcall
    GetAttributes(void)
    { return (TPropertyAttributes() <<
      paDialog << paRevertable); };
  TTextProperty() : TPersistent() {};
};
.
.
class TFileNameProperty : public TTextProperty
{ public:
  virtual void __fastcall Edit(void);
  TFileNameProperty() : TTextProperty() {};
};
.
.
// in the register() function:
RegisterPropertyEditor(
  __typeinfo(TText),
```

```
  __classid(TDBShortList),"ListName",
  __classid(TFileNamesProperty) );
.
.
// in the constructor for TDBShortList:
  ListName = new TText;
//in the destructor for TDBShortList: delete ListName;
.
.
#include "dir.h"
void __fastcall TFileNameProperty::Edit(void)
{
  TOpenDialog *o=new TOpenDialog(Application);

  o-&gtFileName = GetValue();
  o-&gtDefaultExt = "*.lst";
  o-&gtFilter = "List Files (*.lst)|*.lst"
           "|All Files (*.*)|*.*";
  o-&gtFilterIndex = 1;
  o-&gtTitle = "List File Name";

  char buf[MAXPATH+1];
  o-&gtInitialDir = getcwd( buf, sizeof(buf) );

  if ( o-&gtExecute() == true )
  {  SetValue( o-&gtFileName );
    Designer-&gtModified();
  };
  delete o;
}
```

Listing B: *The CommaText property editor*

```
// max chars in ListDefaults string
#define MAXLISTDEFAULTSTEXT 4096
.
.
class TCommaTextProperty : public TTextProperty
{
  public:
    virtual int __fastcall GetEditLimit(void)
    { return MAXLISTDEFAULTSTEXT; };

    virtual void __fastcall Edit(void);
```

```
    TCommaTextProperty() : TTextProperty() {};
};


class TCommaTextForm : public TForm
{
__published:   // IDE-managed Components
  TListBox *ListBox;
  TBitBtn *BClose;
  TBitBtn *BAdd;
  TBitBtn *BDelete;
  TEdit *Item;
  void __fastcall BAddClick(TObject *Sender);
  void __fastcall BDeleteClick(TObject *Sender);
private:// User declarations
  AnsiString Get(void)
    { return ListBox-&gtItems-&gtCommaText; };
  void Set(AnsiString v)
    { ListBox-&gtItems-&gtCommaText = v; };
public:// User declarations
  __fastcall TCommaTextForm(TComponent* Owner) :
                            TForm(Owner) { };
  __property AnsiString CommaText = {
                           read=Get, write=Set };
};
.
.
// in the register() function:
RegisterPropertyEditor( __typeinfo(TText),
                        __classid(TDBShortList),
                        "ListDefaults",
                        __classid(TCommaTextProperty)
                      );
.
.
// in the constructor for TDBShortList:
  ListDefaults = new TText;
// in the detructor for TDBShortList: delete ListDefaults;
.
.
void __fastcall TCommaTextProperty::Edit(void)
{
  TCommaTextForm *f= new TCommaTextForm(Application);
```

```
  f-&gtCommaText = GetValue();
  f-&gtShowModal();

  if ( GetValue() != f-&gtCommaText )
  { if (f-&gtCommaText.Length() > MAXLISTDEFAULTSTEXT)
    {  MessageBox( 0, ("The limit is " +
       AnsiString(MAXLISTDEFAULTSTEXT) +
       " characters. The text will be truncated.")
       .c_str(), "Sorry - too much text",
       MB_ICONEXCLAMATION );

       f-&gtCommaText.SetLength(MAXLISTDEFAULTSTEXT);
  };
    SetValue( f-&gtCommaText );
    Designer-&gtModified();
  };

   delete f;
}
```

Listing C: T*he TDBShortList component editor*

```
class TDBShorListEditor : public TDefaultEditor
{
  private:
    TDBShortList *Component;
    TFormDesigner *Designer;
    int index;

  public:
    virtual void __fastcall EditProperty(
            TPropertyEditor *pe,
            bool &con, bool &fre );

    void Reset(void); // demos use of comp ptr

    virtual int __fastcall GetVerbCount(void)
    { return 3; };

    virtual AnsiString __fastcall GetVerb(int i)
    { const char *verb[] = { "ListDefaults",
                    "ListName", "Reset Example" };
      return verb[i]; };

    virtual void __fastcall ExecuteVerb(int i)
```

```
    { if ( i < 2 ) { index = i; Edit(); }
         else Reset(); };

    void Modified(void)
    { if ( Designer ) Designer-&gtModified(); };

    __fastcall virtual ~TDBShorListEditor(void) {};
    __fastcall virtual DBShorListEditor(
                 TComponent* c, TFormDesigner* d) :
                 TDefaultEditor(c, d)
    {Component= (TDBShortList *)c; Designer = d;};
};
.
.
RegisterComponentEditor(__classid(TDBShortList),
                        __classid(TDBShorListEditor));
.
.
void TDBShorListEditor::Reset( void )
{
  if ( !Component || !Designer ) // be careful
    return;

  if ( MessageBox( 0,
    "Reset the component to the example defaults?",
    "Are You Sure?", MB_YESNO ) != IDYES )
  return; // customer said no... so quit

  Component-&gtListDefaults-&gtText = "Red, Yellow, Blue";
  Component-&gtListName-&gtText = "color.txt";
  Modified();
}

void __fastcall TDBShorListEditor::EditProperty(
     TPropertyEditor *pe, bool &con, bool &fre )
{
  if ( !strcmpi( pe-&gtGetName().c_str(),
       !index ? "ListDefaults" : "ListName" ) )
  {  pe-&gtEdit();
    con = false;
  };
}
```

November 1997

Putting the Registry to work

by Kent Reisdorph

Back in the days of Windows 3.x, using configuration files (a.k.a. INI files) was the prescribed method of storing your application-specific data. An application might use an INI file to store data pertaining to the last window size and position, user options, a list of files last accessed by the user, or any other configuration data.

In 32-bit Windows, however, using configuration files is out. Now, we store application data in the Windows Registry Database (called the Registry for short). The Registry existed in 16-bit Windows, but it wasn't widely used by application developers--it was used primarily by Windows itself. There's nothing stopping you from using configuration files in 32-bit Windows programs. However, the Registry is the approved way of storing application data, and you should learn how to use it.

This article will show you how to access the Registry by introducing you to the TRegistry class. We'll explain how to create a Registry key, how to write information to that key, and how to retrieve the information.

What is the Registry?

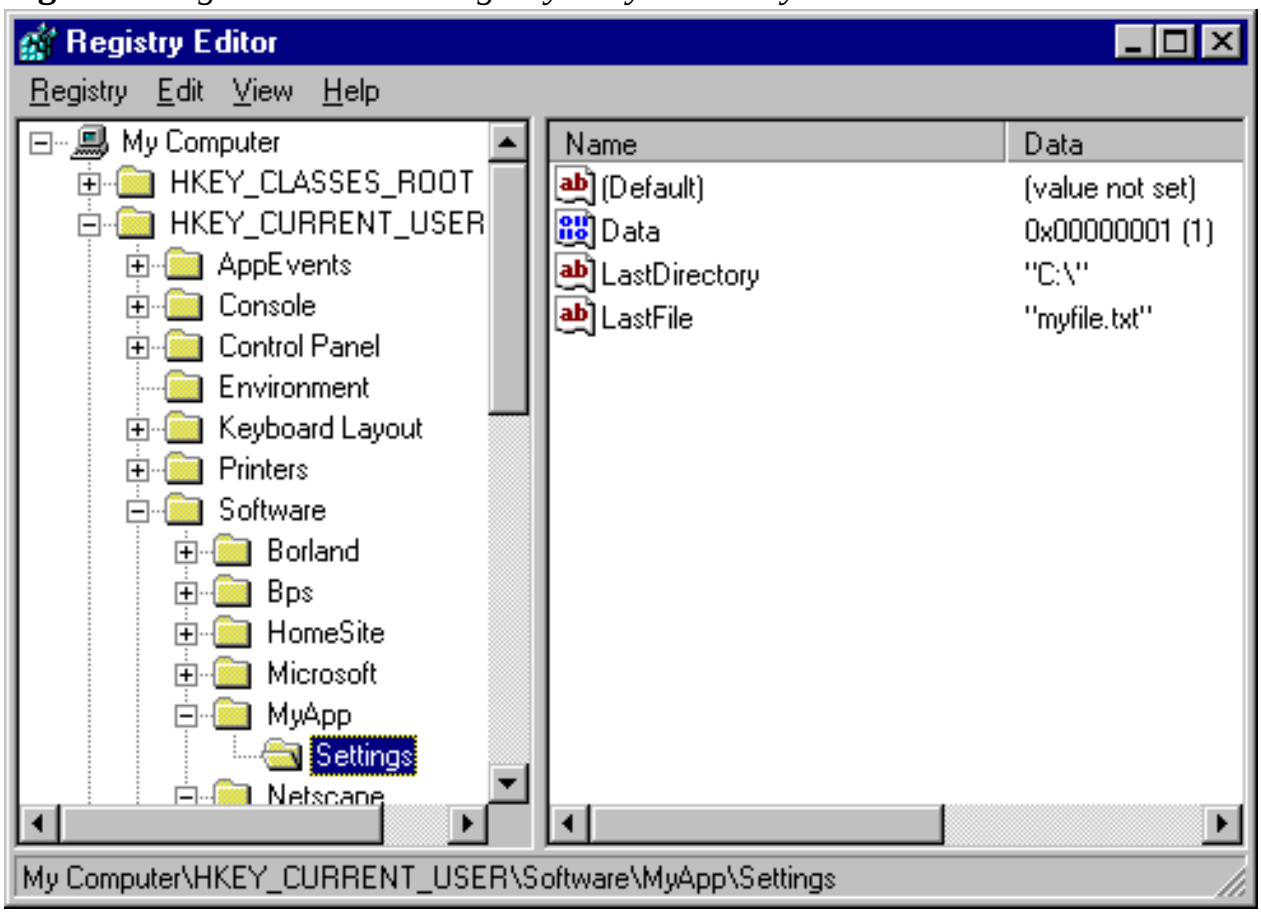
The Registry is simply a database that contains information about Windows and about your Windows configuration, and which also contains application-specific data. One of the Registry's tasks is to store OLE information so that OLE objects can talk to each other. (That was one of the original uses of the Registry.) The Registry also keeps track of associations--which filename extension is associated with which application. For example, if you double-click on a text file in Explorer, Windows will start Notepad (on most systems) and load the text file. This process is made possible by information stored in the Registry. Applications also use the Registry to store configuration data.

The Registry is arranged in a hierarchy that resembles a disk drive's directories and sub- directories. The top-level nodes are called keys and the dependent nodes are called subkeys. For the most part, we tend to call all entries keys and not get hung up on terminology. The Registry has four main keys, as described in Table A. Of these keys, the only one you'll usually be concerned with is HKEY_CURRENT_USER. You'll use this root key to store your application's configuration data.

The Registry is simply a binary database. As such, it can only be manipulated through code or via the Windows Registry Editor utility, Regedit, which you can run from the Windows Start menu. Regedit displays the Registry hierarchy in Explorer-like fashion, with the list of keys on the left and the key data on the right, as shown in **Figure A**. Regedit lets you search the Registry, view Registry information, and

modify the Registry; obviously, you shouldn't modify Registry keys unless you know what you're doing.

Figure A:*Regedit shows the Registry's key hierarchy and data.*



Click on image to view full size.

The Registry and you

So, what data do you store in the Registry? That depends, of course, on what your application does. Here are some items that are commonly stored in the Registry:

- The application's last size and position
- The application's state (normal, maximized, or minimized)
- The last size and position of child windows
- The last file opened
- The path of the last file opened
- A list of most recently used files
- User preferences
- File locations
- Application-specific items

The last item can include just about any data specific to your application. For instance, all of C++Builder's user-configurable options (most of which are accessed via the Environment Options dialog box) are stored in the Registry. This information includes compiler and linker options, editor options, form designer options, the component palette settings, and so on.

The bottom line is, it's up to you to decide what data your application will store in the Registry.

I'll point out here that a quality application will use the Registry to make the user's life easier. For example, if I open a file in an application, then I expect the File Open dialog box to return to that same directory the next time I open a file. I also want to see a list of most recently used files on the file menu. And, I expect my preferences to remain the same between application instances. The Registry can simplify saving and retrieving this type of data.

TRegistry

VCL provides the TRegistry class to assist you in working with the Registry. TRegistry lets you create keys, open keys, write data, read data, delete keys, move keys, and more. (To see an example program that illustrates the use of TRegistry, take a look at **Listing A** in the article "Implementing a Recent-Files List" on page 11.) TRegistry provides methods to read and write data in the following formats:

- Binary data (user-defined)
- Boolean (bool)
- Currency (Currency class)
- Date (TDateTime class)
- DateTime (TDateTime class)
- Float (float or double)
- Integer (int)
- String (AnsiString class)
- Time (TDateTime class)

The Registry really stores only three

types of data: string, integer, and binary; VCL makes the appropriate conversions for other data types. For example, VCL converts the Currency, Date, DateTime, Float, and Time data types to binary data and then stores them in the Registry. The data is converted back to the appropriate type when the object is retrieved from the Registry. These steps are transparent to you and you don't have to worry about them, but you should at least be aware of them.

Though these data types are available, you may opt to store many of your configuration values as strings--even your integer data. If you look at the Registry entries for C++Builder, you'll see that integer and Boolean values are stored as strings. It's up to you which method you use.

TRegistry provides methods to read and write each of the data types we've mentioned. The method to read string data is called ReadString, the method to write string data is called

WriteString, the methods to read and write integer values are called ReadInteger and WriteInteger, and so on.

Creating a key

One of the first things you'll do when working with the Registry is to create a key--a simple task. Before you can use Tregistry, you must add a line to your main form's header to include the REGISTRY.HPP header file, as follows:

```
#include &ltvcl\Registry.hpp>
```

Once you've done so, you can create a key:

```
TRegistry* reg = new TRegistry;
reg->OpenKey("Software\\MyApp\\Settings", true);
delete reg;
```

There are several items to note in this code sample. First, you create an instance of TRegistry, which is a runtime-only class (we use the words component and class interchangeably when talking about VCL objects). Once you have a TRegistry object, you can use the OpenKey method to create and open a key at the same time. You specify true for OpenKey's final parameter, to tell VCL to create the key if it doesn't yet exist.

You can see in the code that the name of the key is Settings, but where in the Registry hierarchy is the key created? The answer lies in TRegistry's Root, which is set to HKEY_CURRENT_USER by default. Usually you won't need to change the Root property, but you may need to do so if you must access data in other keys (refer to **Table A**). In our example, the Settings key will be created in HKEY_CURRENT_USER\Software\MyApp. This fact brings up yet another point in Registry usage: Tradition has it that the Software key is the root key for application-specific data. You should create a key under Software that has your application's name (or your company name, if you prefer). Under that key, you can create subkeys for the specific types of data you need to store. If you fire up Regedit and look at the Software key, you'll probably see several applications listed. For instance, the root key for C++Builder data is HKEY_CURRENT_USER\Software\Borland\C++Builder\1.0. Follow this pattern for storing data in the Registry, and your users will thank you should they ever have to manually edit the Registry. We should point out that TRegistry has a method called CreateKey. While CreateKey does indeed create a key, it has a peculiar quirk--the key is created but not opened. As a result, you must call OpenKey after calling CreateKey. Since OpenKey can both create and open a key, it makes more sense to use OpenKey to begin with rather than CreateKey.

Finally, note that we don't call CloseKey to close the Registry key--the TRegistry destructor

closes the key for us. We could call CloseKey explicitly, but it's not necessary. CloseKey is primarily for those times when you have to close one Registry key in order to open another.

Reading, writing, and 'rithmetic

After you've created a key, you can write any number of data items to it. Your application can later read this data and use it accordingly. You can create a data item using any of TRegistry's WriteXXX methods. For example, the code segment

```
TRegistry* reg = new TRegistry;
reg->OpenKey("Software\\MyApp\\Settings", true);
// write some data
reg->WriteString("LastFile", "myfile.txt");
reg->WriteString("LastDirectory", "C:\\");
reg->WriteInteger("Data", 1);
delete reg;
```

creates a key and three data items under that key. In **Figure A**, the Registry Editor displays the key and data created by this code.

As you can see, the Settings key has three data items called Data, LastDirectory, and LastFile. You might notice that this is not the order in which we added the data items--Regedit automatically sorts the data items so you can easily find a particular data item when editing the Registry. Reading from the Registry is a trivial task. Let's say, for example, that you wanted to retrieve the last file used. Here's how you'd do so:

```
TRegistry* reg = new TRegistry;
reg->OpenKey("Software\\MyApp\\Settings", true);
String FileName = reg->ReadString("LastFile");
delete reg;
```

By the way, the primary Registry keys are always open, so accessing the Registry is very quick. Using the Registry is almost certainly faster than using configuration files.

TRegistry miscellany

If you look up TRegistry in the VCL help file, you'll see that it includes a great many methods. A large segment of these methods read and write the various data types supported by TRegistry--and these are TRegistry methods that, frankly, you'll probably never use.

The DeleteKey method deletes a key, all subkeys under that key, and all data items. Be sure

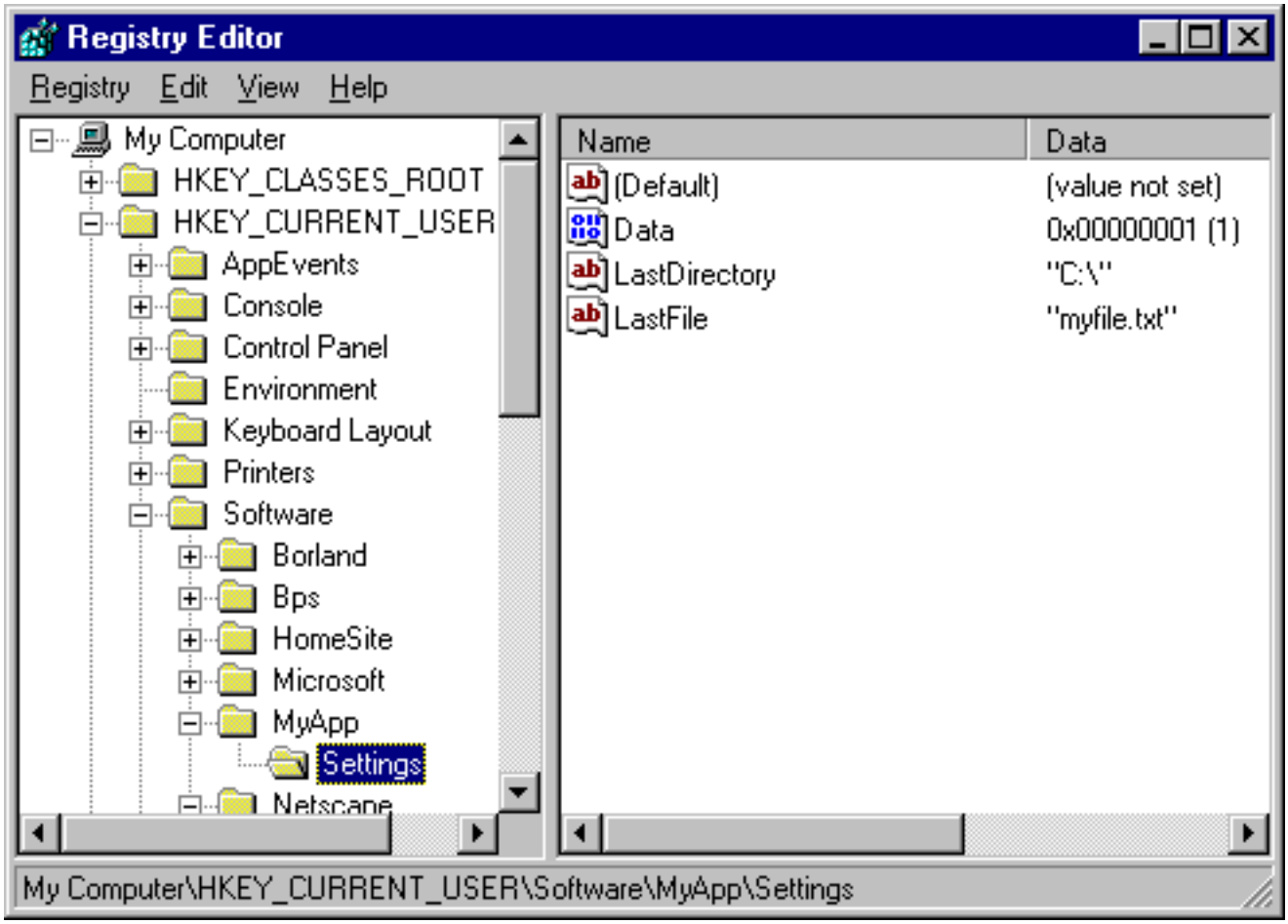
that you clean up after yourself when using the Registry--there's nothing worse than a Registry cluttered with entries from programs that are long gone. Be diligent in deleting keys that your program no longer uses. The better commercial installation programs include an uninstall option that will perform this task for you. If you're creating temporary keys through code, then be sure to delete them when you're done with them.

VCL provides two other classes for storing configuration data. The TIniFile class stores data in a configuration file. While you certainly may use INI files in your applications, doing so isn't recommended. For this reason, you probably won't use TIniFile in your C++Builder applications.

Another VCL class, TRegIniFile, eases the transition from using INI files to using the Registry. For example, let's say you had an existing application that used TIniFile extensively. You could switch to TRegIniFile, and your configuration data would be stored in the Registry rather than in an INI file. The use of TRegIniFile will probably be limited in C++Builder since you're unlikely to have legacy C++Builder applications that use TIniFile.

Table A: *Main Registry keys*

Key	Description
HKEY_CLASSES_ROOT	Contains OLE and Windows shell information
HKEY_CURRENT_USER	Contains various information about the current user's setup; can be application-specific data or Windows data
HKEY_LOCAL_MACHINE	Generally contains hardware information about the system; may contain other data as well
HKEY_USERS	Contains default user setup information and information common to all users on the system


Registry Editor
Registry Edit View Help
My Computer
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
AppEvents
Console
Control Panel
Environment
Keyboard Layout
Printers
Software
Borland
Bps
HomeSite
Microsoft
MyApp
Settings
Netscape
Name
Data
(Default)
(value not set)
Data
0x00000001 (1)
LastDirectory
"C:\"
LastFile
"myfile.txt"
My Computer\HKEY_CURRENT_USER\Software\MyApp\Settings

November 1997

Implementing a recent-files list

by Kent Reisdorph

Most programs that use document files maintain a list of most-recently-used files. This list, often called an MRU list, is dynamically created by the application as the user works with document files.

VCL doesn’t provide built-in support for an MRU list. So, in order to implement an MRU list in your C++Builder applications, you’ll have to do it yourself. In this article, we’ll show you how to create and maintain such a list. You’ll use the Registry to store the list of files (see "Putting the Registry to Work," on page 8, for details on using the Registry). Along the way, we’ll introduce some of the features of the TMenu class.

A typical MRU list

While no standards committee governs the use of MRU lists, you’ll see certain common features in most MRU lists. Let’s take a quick look at the feature set for a typical list.

First, the list shouldn’t be visible when you run the application for the first time. In other words, if there are no recently used files, then the list should be hidden. As files are opened, they’ll appear in the list. Typically, an MRU list will have a maximum number of entries--generally five to ten. The list should be maintained on a first-in, first-out (FIFO) basis so the last file used is always at the top of the list. Once the list grows to the maximum size, the oldest file is removed to make room for the newest file.

Typically, the MRU list shows the complete path and filename of the document file. The list usually associates a hotkey with each menu item. For example, the hotkey for the first item is usually 1, the hotkey for the second item is 2, and so on. The MRU list should appear on the File menu, just above the Exit menu item.

Not every application uses this exact approach, but such a layout seems to be the most widely used. (Note that C++Builder uses a different scheme; it implements the MRU list on a pop-up menu called Reopen. I don’t particularly like this non-standard approach, and I wouldn’t write my own applications using this method.) Some applications place the MRU list at the very end of the File menu, which is a slight variation. In any case, the decision of where to put the list is up to you. Now that you know how the MRU list should look and act, you can get to work on the design stage.

Designing the MRU list

There are many ways to approach the creation of an MRU list. First, let’s decide what ingredients we

need. At a minimum, you'll require the following:

- A way to show the MRU list when it's needed and hide it when it isn't
- A way to detect when an MRU menu item is clicked
- An event handler that performs some action when an MRU item is clicked
- A routine to manage the list (adding new items and deleting old items)
- A place to store the filename strings while the program is running
- A more persistent place to store the filename strings between application instances

VCL easily handles the first three items via the TMenu class and the use of events. For string maintenance, you'll use a TStringList to store the list of filenames at runtime. Finally, you'll use the Registry for persistent storage of your MRU list. Let's break this process down into three pieces so it makes more sense.

Managing the menu

You can insert the items into the File menu in one of several ways. You could use TMenuItem's Insert method to insert the menu items into the File menu as needed. You could also use the API menu functions. There's an easier way, however--you'll create a menu separator and as many blank menu items as needed for your MRU list. Then you'll just hide the new menu items until you're ready to display them. You can set the Visible property of the menu items and separator to False at design time, then set it to True at runtime when you need to display them.

Begin by dropping a MainMenu component on a blank form. Then, double-click the MainMenu icon to start the C++Builder Menu Designer. To quickly insert a pre-built File menu into the main menu, use the Insert From Template option. Now, create a separator just above the Exit item separator (type a dash in the menu item's Caption property and press [Enter]) and change its Name property to *MRUSeparator*. Create five menu items under the separator and name them *MRU1* through MRU5. You can leave the Caption property blank, since the menu item text will be supplied at runtime. Select all five blank menu items and set the OnClick event handler to *MRUClick*--doing so will allow all the menu items to use the same OnClick event handler. Set the Visible property for the separator and the blank menu items to False.

Now, close the Menu Designer and test the menu. It should look like a regular File menu, since the MRU list is initially hidden. But at what point do you set the Visible property to True? Once again, it's VCL to the rescue. You can create an OnClick handler for the top-level File menu. This OnClick event will give you a chance to modify the menu before it's displayed--a perfect time to show the necessary MRU menu items. You can't set up the event handler for the MRU click events at this point, because you haven't yet determined how you're going to store the filenames. Let's take a look at that issue now, then return later to the OnClick event handler.

Storing the list of files

You could use the Tag property of each menu item to store the filenames--an idea that has some merit, but also some drawbacks. Instead, you'll store the filenames in a separate TStringList object, which will be a member of your main form's class.

As an added benefit, the TStringList gives you a convenient way to implement the FIFO aspect of the MRU list. When the user opens a new file, you'll add the filename to the top of the list and delete the last item if necessary. For example,

```
const int maxItems = 5;
...
if (OpenDialog1->Execute()) {
// Open a file, etc.
// Add the filename to the MRU list.
MruList->Insert(0, OpenDialog1->FileName);
// Remove the last item if the list is full.
if (MruList->Count == maxItems + 1)
MruList->Delete(maxItems);
}
```

is all the code required to maintain a FIFO list of filenames--slick and easy. Naturally, you'll create your string list in the form's OnCreate event handler and delete it in the OnDestroy event handler.

The OnClick event handler

Now that you know how to store the file list, let's back up and address the issue of the OnClick event handler for the MRU menu items. You need to translate the menu item clicked to an item in the string list. Since you know that the first MRU item is named *MRU1*, you can get the menu index of that menu item and figure out from there which MRU item was clicked, as follows:

```
void __fastcall
TForm1::MRUClick(TObject *Sender)
{
// cast Sender to a TMenuItem*
TMenuItem* itemClicked =
dynamic_cast(Sender);
// cCalculate 0-based index from there
int index =
itemClicked->MenuIndex - MRU1->MenuIndex;
// MruList[index] is the string we're after
```

```
String FileName = MruList[index];
// do something with FileName
}
```

You can use similar logic to determine which of the MRU menu items to make visible. See **Listing A** for details.

Storing the strings in the Registry

You'll load the string list from the Registry when the form is created and write the string list to the Registry again when the form is destroyed. See **Listing B** on the following page for the code to save and restore the MRU list. Using the Registry to store the strings provides a near-foolproof method of keeping the MRU list around after the application has closed.

An example

Listings A and **B** contain a program that implements an MRU list. (You can download our sample files from www.cobb.com/cpb; click the Source Code hyperlink.) The program contains a MainMenu component, a Memo component, an OpenDialog component, and a CheckBox component (the check box deletes the Registry key created by the program). The first time you run the program, the MRU list will be empty and hidden. You can use the File | Open... menu item to open any text file, which will be displayed in the Memo component.

As you open each file, its filename will be added to the MRU list. Close and restart the program, and you'll see that the MRU list is persistent between application instances. When you click on one of the MRU items, the file corresponding to that menu item will then be loaded in the Memo component.

Listing A: *MRUUNIT.H*

```
//---------------------------------------------------
#ifndef MruUnitH
#define MruUnitH
//---------------------------------------------------
#include
#include
#include
#include
#include
#include
#include
```

```
//---------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
TMainMenu *MainMenu1;
TMenuItem *File1;
TMenuItem *New1;
TMenuItem *Open1;
TMenuItem *Save1;
TMenuItem *SaveAs1;
TMenuItem *N2;
TMenuItem *Print1;
TMenuItem *PrintSetup1;
TMenuItem *N1;
TMenuItem *Exit1;
TMenuItem *MRUSeparator;
TMenuItem *MRU1;
TMenuItem *MRU2;
TMenuItem *MRU3;
TMenuItem *MRU4;
TMenuItem *MRU5;
TMemo *Memo1;
TOpenDialog *OpenDialog1;
TCheckBox *CheckBox1;
void __fastcall MRUClick(TObject *Sender);
void __fastcall File1Click(TObject *Sender);
void __fastcall FormCreate(TObject *Sender);
void __fastcall FormDestroy(TObject *Sender);
void __fastcall Open1Click(TObject *Sender);
private:        // User declarations
TStringList* MruList;
public:   // User declarations
__fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------
#endif
```

Listing B: *MRUUNIT.CPP*

```
//---------------------------------------------------
#include
```

```
#pragma hdrstop

#include "MruUnit.h"
//---------------------------------------------------
#pragma resource "*.dfm"
const int MruCount = 5;
const char* RegKey = "Software\\BCBJournal\\MruTestProgram";
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//---------------------------------------------------
void __fastcall TForm1::MRUClick(TObject *Sender)
{
// Load a file in the Memo component based on
// the MRU item that was clicked.
TMenuItem* itemClicked =
dynamic_cast(Sender);
int index =
itemClicked->MenuIndex - MRU1->MenuIndex;
Memo1->Lines->LoadFromFile(
MruList->Strings[index]);
}
//---------------------------------------------------
void __fastcall
TForm1::File1Click(TObject *Sender)
{
// This is the OnClick handler for the top
// level File menu. Here we check each string
// in the MRU list and show the associated MRU
// menu item if the string is not empty.
int index = MRU1->MenuIndex;
for (int i=0;iStrings[i] != "") {
MRUSeparator->Visible = true;
File1->Items[index + i]->Visible = true;
char buff[MAX_PATH];
sprintf(buff, "&%d %s",
i + 1, MruList->Strings[i].c_str());
File1->Items[index + i]->Caption = buff;
    }
  }
```

```
}
//---------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
// Create the TStringList class.
MruList = new TStringList;
// Load the strings from the registry. First
// try to open the key. If that fails, then we
// know that the program is being run for the
// first time and that we need to create the
// key and set up the MRU items in the key.
TRegistry* reg = new TRegistry;
if (!reg->OpenKey(RegKey, false)) {
reg->OpenKey(RegKey, true);
reg->WriteString("MRU1", "");
reg->WriteString("MRU2", "");
reg->WriteString("MRU3", "");
reg->WriteString("MRU4", "");
reg->WriteString("MRU5", "");
  }
// Read each string from the registry. Some of
// the strings could be empty, but that's OK.
MruList->Add(reg->ReadString("MRU1"));
MruList->Add(reg->ReadString("MRU2"));
MruList->Add(reg->ReadString("MRU3"));
MruList->Add(reg->ReadString("MRU4"));
MruList->Add(reg->ReadString("MRU5"));
// Delete the TRegistry object.
delete reg;
}
//---------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
// Save the string list to the registry. If
// the check box is checked, then delete the
// key and don't save the MRU list (duh).
TRegistry* reg = new TRegistry;
if (CheckBox1->Checked)
reg->DeleteKey(RegKey);
else {
reg->OpenKey(RegKey, true);
// Write the list to the registry.
reg->WriteString(
```

```
"MRU1", MruList->Strings[0]);
reg->WriteString(
"MRU2", MruList->Strings[1]);
reg->WriteString(
"MRU3", MruList->Strings[2]);
reg->WriteString(
"MRU4", MruList->Strings[3]);
reg->WriteString(
"MRU5", MruList->Strings[4]);
  }
// Clean up.
delete reg;
delete MruList;
}
//---------------------------------------------------
void __fastcall TForm1::Open1Click(TObject *Sender)
{
// Load the selected file into the Memo and
// update the MRU list. Add the FileName to
// the top of the string list and delete the
// last item if the string list is full. We
// don't check for duplicate strings, but that
// would be a good feature to implement.
if (OpenDialog1->Execute()) {
Memo1->Lines->LoadFromFile(
OpenDialog1->FileName);
MruList->Insert(0, OpenDialog1->FileName);
if (MruList->Count > MruCount)
MruList->Delete(MruCount);
  }
}
//---------------------------------------------------
```

November 1997

Bitmaps on demand

by Kent Reisdorph

Did you ever wonder how you could get your hands on certain bitmaps found in Windows? Maybe your application has owner-drawn controls. Or, perhaps you're writing a custom component and you need it to look as much as possible like a standard Windows control. The minimize button, maximize button, close button, scroll bar arrows, menu check marks--they all have to come from somewhere. But how do you get hold of them? In this article, we'll show you their hiding places.

LoadBitmap

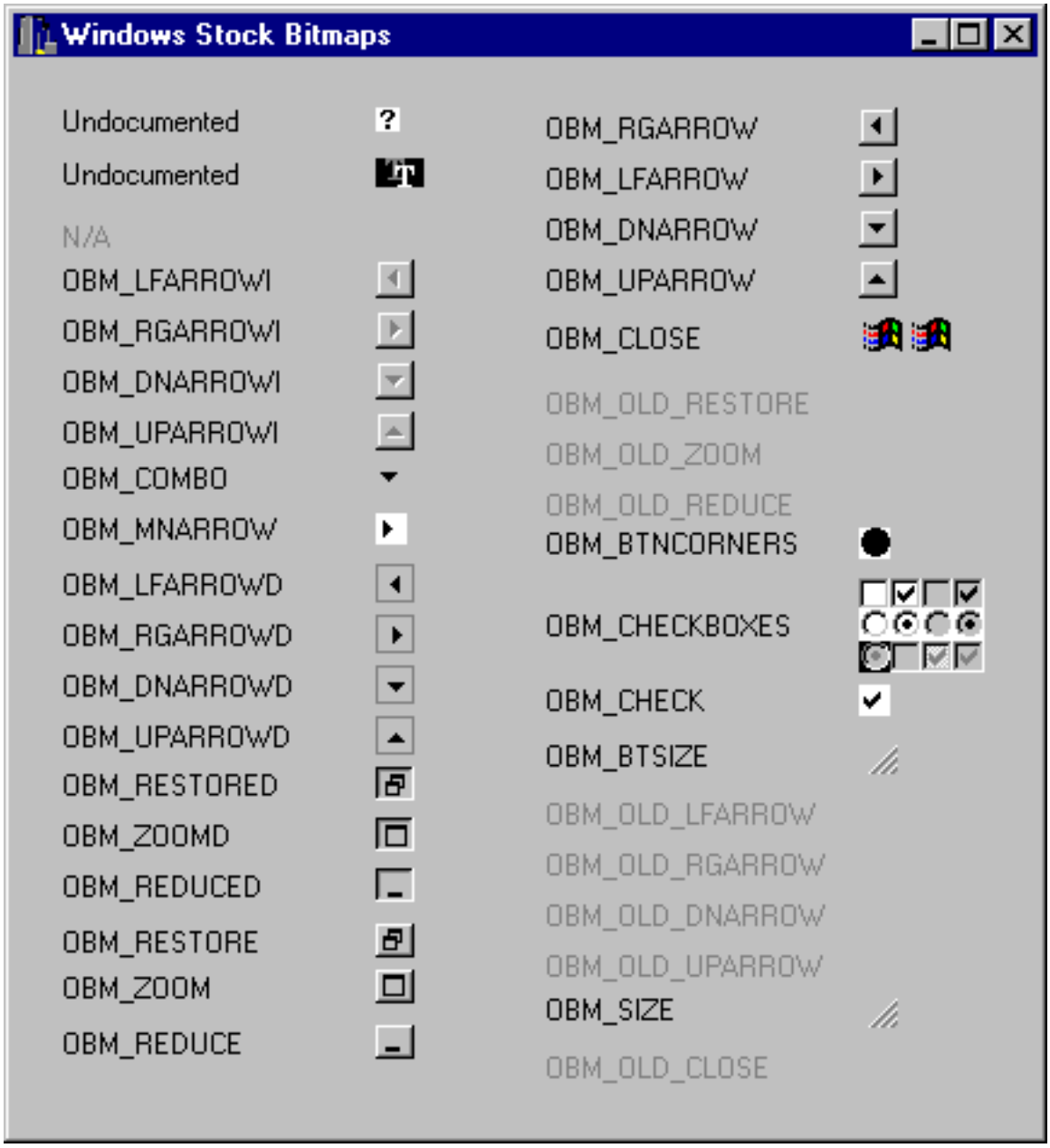
The secret lies in the Windows API function LoadBitmap(), which you can use to load a bitmap contained in your application as a resource. Typically, you create a resource with a resource editor, bind it to the executable file when you build your application, and then load and display it at runtime.

But LoadBitmap() can also extract the stock Windows bitmaps. The Win32 online help contains a complete list of bitmaps that you can obtain using LoadBitmap(), along with their constants. To obtain the flying Windows logo, for example, you could use code like the following:

```
#include
HBITMAP hBitmap =
  LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE));
```

This code loads the bitmap identified by OBM_CLOSE (the close-box bitmap). **Figure A** shows a program that displays all the bitmaps available to you, along with their constant names. Note that some bitmap names begin with OBM_OLD. These bitmaps don't display, because the constants are leftover from prior versions of Windows. If you want to use the resource names, then you must include the WINSRESRC.H file in which the constant names are defined.

Figure A: *Our example program shows all the stock Windows bitmaps.*



So now what?

OK, so our example doesn't tell you much about what to do with the bitmap once you've pried it out of Windows. A typical VCL use of one of these bitmaps might look like this:

```
Graphics::TBitmap* bm = new Graphics::TBitmap;
bm-&gtHandle =
  LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE));
if (bm-&gtHandle)
  Canvas-&gtDraw(0, 0, bm);
delete bm;
```

This code loads the close-box bitmap from Windows and displays it in the upper-left corner of

the form. Once you give the bitmap handle to VCL, you no longer have to worry about it--the TBitmap class will take care of freeing the resource. **Listing A** contains the paint routine we used to produce the program shown in **Figure A**. A quick look at this listing will show you how to load and display bitmaps in VCL.

That's it!

Does this technique sound simple? Well, so it is. When you need your owner-drawn controls to look just like the Windows controls, you know where to get the bitmaps. The pre-defined Windows bitmaps are there for the taking--why bother creating your own? **Listing A:** *OnPaint handler*

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TStringList* Names = new TStringList;
  Names-&gtAdd(“Undocumented”);
  Names-&gtAdd(“Undocumented”);
  Names-&gtAdd(“N/A”);
  Names-&gtAdd(“OBM_LFARROWI”);
  Names-&gtAdd(“OBM_RGARROWI”);
  Names-&gtAdd(“OBM_DNARROWI”);
  Names-&gtAdd(“OBM_UPARROWI”);
  Names-&gtAdd(“OBM_COMBO”);
  Names-&gtAdd(“OBM_MNARROW”);
  Names-&gtAdd(“OBM_LFARROWD”);
  Names-&gtAdd(“OBM_RGARROWD”);
  Names-&gtAdd(“OBM_DNARROWD”);
  Names-&gtAdd(“OBM_UPARROWD”);
  Names-&gtAdd(“OBM_RESTORED”);
  Names-&gtAdd(“OBM_ZOOMD”);
  Names-&gtAdd(“OBM_REDUCED”);
  Names-&gtAdd(“OBM_RESTORE”);
  Names-&gtAdd(“OBM_ZOOM”);
  Names-&gtAdd(“OBM_REDUCE”);
  Names-&gtAdd(“OBM_RGARROW”);
  Names-&gtAdd(“OBM_LFARROW”);
  Names-&gtAdd(“OBM_DNARROW”);
  Names-&gtAdd(“OBM_UPARROW”);
  Names-&gtAdd(“OBM_CLOSE”);
  Names-&gtAdd(“OBM_OLD_RESTORE”);
  Names-&gtAdd(“OBM_OLD_ZOOM”);
  Names-&gtAdd(“OBM_OLD_REDUCE”);
```

```
  Names-&gtAdd(“OBM_BTNCORNERS”);
  Names-&gtAdd(“OBM_CHECKBOXES”);
  Names-&gtAdd(“OBM_CHECK”);
  Names-&gtAdd(“OBM_BTSIZE”);
  Names-&gtAdd(“OBM_OLD_LFARROW”);
  Names-&gtAdd(“OBM_OLD_RGARROW”);
  Names-&gtAdd(“OBM_OLD_DNARROW”);
  Names-&gtAdd(“OBM_OLD_UPARROW”);
  Names-&gtAdd(“OBM_SIZE”);
  Names-&gtAdd(“OBM_OLD_CLOSE”);
  Graphics::TBitmap* bm = new Graphics::TBitmap;
  int y = 20;
  int x = 20;
  int Offset;
  Canvas-&gtBrush-&gtStyle = bsClear;
  for (int i = 1;i < 38;i++) {
    bm-&gtHandle =
      LoadBitmap(NULL, MAKEINTRESOURCE(i + 32730));
    if (bm-&gtHandle) {
      Canvas-&gtFont-&gtColor = clBlack;
      Canvas-&gtDraw(x + 130, y, bm);
      Offset = (y + (bm-&gtHeight / 2)) - 6;
    }
    else {
      Canvas-&gtFont-&gtColor = clGray;
      Offset = y + 5;
    }
    Canvas-&gtTextOut(x, Offset, Names-&gtStrings[i - 1]);
    if (bm-&gtHeight > 21) y += bm-&gtHeight + 5;
    else y+= 21;
    if (i > 18 && x == 20) {
      y = 20;
      x = 220;
    }
  }
  delete bm;
  delete Names;
}
```

November 1997

Quick C++Builder tips

If you'd like to receive a free C++Builder tip similar to these via E-mail each week, visit www.zdtips.com. You can also sign up for other ZDTips services and receive free tips on a variety of software and operating system topics.

Adding a DOS Window tool

by Sam Azer

Sometimes the easiest way to run command line tools from the IDE is through a DOS window. To be able to instantly pop one open in your project directory, use the Tools | Configure Tools... menu command. Click the Add... button, then set the Title field to Dos Window, the Program field to c:\command.com, and the Parameters field to /k $PATH(). Finally, click OK. Now you've got instant access to a DOS window in your project's directory--any time you need it!

Delete unnecessary C++Builder files

by Kent Reisdorph

Running low on hard disk space? Take inventory of those C++Builder projects you're not working on at the moment, then delete any files from those projects that have the extension OBJ, RES, or TDW. You also can delete files whose extensions begin with IL. Since these files can become quite large, removing them can free significant space on your hard disk.

The transparent pixel

by Kent Reisdorph

When you place a glyph (a picture, generally a BMP file) on a button, be aware that the color of the pixel in the glyph's lower-left corner will be used as the transparent color. Any pixels that color will be transparent on the button. If you don't want any transparent pixels on your button bitmaps, you must be sure to use a color in the lower-left corner that you don't use anywhere else in the glyph.

October 1997

Using the Find and Replace dialog boxes

by Kent Reisdorph

C++Builder provides dozens of components that will make your programming chores easier. VCL even includes one group of components that encapsulate the Win32 common dialog boxes. You'll no doubt use the OpenDialog and SaveDialog components from this group most often. However, other dialog box components are also available, including PrinterSetupDialog, PrintDialog, ColorDialog, and FontDialog. All of these dialog boxes have fairly complete implementations and are reasonably straightforward to use.

Two other components, FindDialog and ReplaceDialog, are less well known. These components exist in relative obscurity partly because Find and Replace dialog boxes aren't appropriate for every type of application. Moreover, these dialog boxes aren't particularly intuitive to use.

In this article, we'll explain how to use the Find and Replace dialog boxes in your C++Builder applications. To do so, we'll examine the properties and methods of the FindDialog and ReplaceDialog components. We'll pay particular attention to the events these dialog boxes generate.

The Find dialog box

When you come right down to the nitty-gritty details, the FindDialog component doesn't do much. It basically displays the Find dialog box and notifies you when the user clicks the Find Next button--but you have to do all the work. Let's begin by looking at what the Find dialog box does offer.

The Find dialog box allows the user to select from three basic options: Match Whole Word Only, Match Case, and Direction (up or down), as shown in **Figure A**. When you select Match Whole Word Only, you indicate that the search text must be a whole word and not merely part of another word (*the* in Mother, for example). The Match Case option dictates whether the search will be case sensitive. The Direction option indicates whether the search should move forward through the document (down) or backward (up).

Figure A: *The Find dialog box offers three options.*

Properties

The FindDialog component has surprisingly few properties. You can control the component's position only through its Top, Left, or Position properties.

The FindText property contains the text to search for--you can read this property to retrieve the string the user wants to

find. If you assign a value to this property prior to displaying the Find dialog box, then the Find What field will contain that value when the dialog box is displayed.

The Options property is a set. By adding values to the set, you can check or uncheck options (such as the Match Case option) or you can hide them altogether. For example, if you don't want to let your users search both up and down, you can hide the Direction option by adding the frHideUpDown value from the Options property. At runtime, you can read the Options property to determine which options the user had set at the time he or she clicked the Find Next button.

Methods

The TFindDialog class has only two methods (aside from the constructor and destructor). The Execute() method displays the dialog box, and the CloseDialog() method closes it. Note that the Find and Replace dialog boxes, unlike the other common dialog boxes, are modeless. That is, they allow the user to work with the application in the background while the dialog box remains onscreen. In order to dismiss the dialog box programmatically, you need to call the CloseDialog() method.

Events

The FindDialog component has only one event: OnFind. This event occurs when the user clicks the Find Next button. You might expect the event handler for the OnFind event to be packed with information. In reality, OnFind is a simple notification event that gives you no information other than the Sender parameter. It's up to you to extract information from the Find dialog box and act on it accordingly.

Find what?

Normally, when you're using the Find dialog box, you're searching for text in some kind of edit control. Since C++Builder generates 32-bit applications, you have three choices of edit components: Edit (single-line edit control), Memo (multi-line edit control), and RichEdit (multi-line super edit control).

If you're writing an application that requires search operations, then the RichEdit component is the only way to go. Not only can this component read, display, and save plain text files, it can also handle rich-text format (RTF) files. In addition, the RichEdit component includes built-in text search features that greatly simplify your work.

As we mentioned, the OnFind event will fire when the user clicks the Find Next button. In your event handler for this event, you'll need to extract the search text and the options the user has set. Once you have that information, you can perform your search.

Dealing with the options

Typically, the first step in obtaining the search text and options is to extract the user's choices from the Options property. At the least, you'll have to deal with the Match Whole Word Only and Match Case options. Since the Options property is a set, you can retrieve information from that property using the Contains() function, as follows:

```
if (FindDialog-$gt Options.Contains(frMatchCase))
  // do something
```

But retrieving the options is only half of the story--you have to actually do something with the options.

Let's look at an example. We're going to use the RichEdit component's FindText() method to find the text. This method requires a parameter, Options, which tells the rich-edit control how to perform the search. The Options parameter is also a set, named TSearchTypes. A picture is still worth a thousand words--so take a look at how all this fits together:

```
TSearchTypes Options;
if (FindDialog-$gtOptions.Contains(frWholeWord))
  Options << stWholeWord;
if (FindDialog-$gtOptions.Contains(frMatchCase))
  Options << stMatchCase;
// Deselect any selected text.
int Pos =
 RichEdit-$gtFindText(FindDialog-$gtFindText,
   StartPos, Length, Options);
```

You can see that we declare an Options variable of type TSearchTypes, read the Find dialog box's options, and add corresponding elements to our Options variable. We then pass the Options set to the FindText() method, which takes over from there.

Let's examine the FindText() method for a moment. The first parameter is the text for which to search, the second parameter is the starting position, the third parameter is the length of text to search, and the fourth parameter holds the search options. FindText() returns the character position of the text if the text was found or -1 if there was no match.

One thing that FindText() won't do for you is search backward. If you're going to allow backward searches, then you'll need to write the necessary code to perform such a search.

Searching tips

Searching a document involves more than just the steps we've outlined. First, the FindText() method does nothing other than return the location of the found text. It doesn't highlight the text, nor does it scroll the edit window to display the located text. Once you find the text, you need to write code to display and highlight it. Fortunately, this process doesn't require much programming. A typical find operation might look like this:

```
int Pos = RichEdit-$gtFindText(FindDialog-$gtFindText, StartPos, Length, Options);
if (Pos != -1) {
  RichEdit-$gtSelStart = Pos;
  RichEdit-$gtSelLength = FindDialog-$gtFindText.Length();
  RichEdit-$gtPerform(EM_SCROLLCARET, 0, 0);
  RichEdit-$gtSetFocus();
}
```

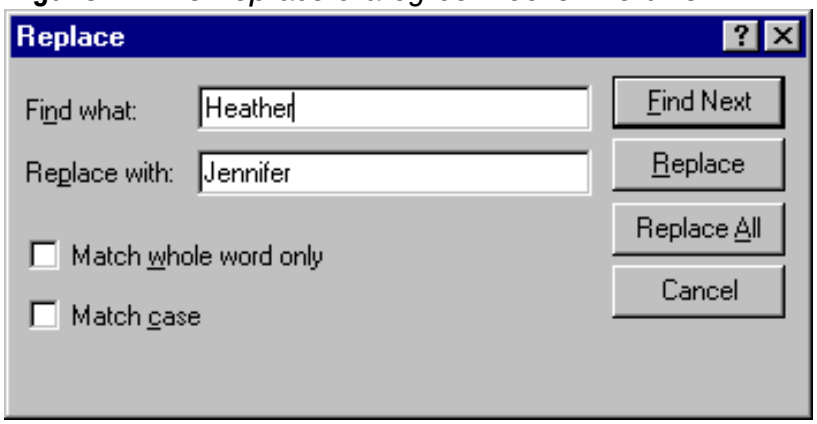
If this code finds the text, it sets the SelStart property to the location of the found text and the SelLength property to the length of the search text. Next, the code sends an EM_SCROLLCARET message to the RichEdit component to scroll the editor window to the caret location. (For some reason, the RichEdit component doesn't have a method to scroll to the caret, so you have to send a Windows message to perform that action--think of it as Windows programming the old-fashioned way.) Finally, the code sets focus to the RichEdit component so the text is highlighted.

After a successful search, you should move the SelStart property the entire length of the selected text. If you don't take this step, then you'll find the same occurrence of the text repeated again and again.

The Replace dialog box

Believe it or not, you already know most of what you need to know to use the ReplaceDialog component. Since TReplaceDialog is derived from TFindDialog, everything we've talked about up to this point also pertains to the Replace dialog box. Obviously you'll find a few extras in the ReplaceDialog component, but it has no methods besides those in the FindDialog component. **Figure B** shows the Replace dialog box.

Figure B: *The Replace dialog box looks like this.*



Properties

Besides having all the properties of the FindDialog component, the ReplaceDialog component has the ReplaceText property, which contains the value of the Replace dialog box's Replace With field. Use the FindText and ReplaceText properties to perform your search-and-replace operations.

Events

The ReplaceDialog component, like FindDialog, has an OnFind event, which works exactly as it does with the Find dialog box. If your application has Find and Replace dialog boxes, you can use the same OnFind event handler for both. Use dynamic_cast to determine which dialog box generated the OnFind event:

```
if (dynamic_cast(Sender))
  // it was the find dialog box
else
  // it was the replace dialog box
```

You can then perform special actions based on the result.

In addition, the Replace dialog box has an OnReplace event that's generated when the user clicks the Replace or Replace All button. You can read the Options property to determine which button the user clicked.

Replacing Text

Naturally, when you receive notification that the user has clicked the Replace button or the Replace All button, you'll have to do something. If the user clicked Replace, you can assume that text was previously found and simply replace it with the new text. You do this using the SetSelTextBuf() method, as follows:

```
RichEdit-$gtSetSelTextBuf(
  ReplaceDialog-$gtReplaceText.c_str());
```

If the user clicked Replace All, then you'll need to search the document for all occurrences of FindText and replace them with ReplaceText.

Listing A shows a program that illustrates the concepts we've discussed in this article. This program contains a RichEdit component, a Find dialog box, a Replace dialog box, and two buttons (Find and Replace). On program startup, the RichEdit component is loaded with the FINDMAIN.CPP file (the source code unit for the program's main form). You can search for text, replace text--even replace all. The OnFind and OnReplace event handlers do all the work--examine them closely to see just how they operate. To create this application, place the components on a form and then enter the code from **Listing A**. We've marked in color the code you'll need to type. Note that this example doesn't perform a backward search, but it does perform the basic find and replace operations. It should give you a good start on implementing the Find and Replace dialog boxes in your applications.

Listing A: *FINDMAIN.CPP*

```
#include &ltvcl\vcl.h>
#pragma hdrstop

#include "FindMain.h"
//---------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(Tcomponent *Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------
void __fastcall TForm1::FindBtnClick(TObject *Sender)
{
  FindDialog-$gtExecute();
}
//---------------------------------------------------
void __fastcall TForm1::ReplaceBtnClick(TObject *Sender)
{
  ReplaceDialog-$gtExecute();
}
//---------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  RichEdit-$gtLines-$gtLoadFromFile("FindMain.cpp");
}
//---------------------------------------------------
```

```
void __fastcall TForm1::FindDialogFind(TObject *Sender)
{
  // Dialog that generated the event could be either
  // Find or Replace dialog so just cast
  // Sender to a TFindDialog.
  TFindDialog* Dlg =
     dynamic_cast(Sender);

  // Set up the options.
  TSearchTypes Options;
  if (Dlg-$gtOptions.Contains(frWholeWord))
    Options << stWholeWord;
  if (Dlg-$gtOptions.Contains(frMatchCase))
    Options << stMatchCase;

  // Deselect any selected text and advance the cursor
  // position. Prevents us from selecting the same
  // text over and over again.
  if (RichEdit-$gtSelLength) {
    RichEdit-$gtSelStart += RichEdit-$gtSelLength;
    RichEdit-$gtSelLength = 0;
  }

  // Find the text.
  int Pos = RichEdit-$gtFindText(Dlg-$gtFindText,
    RichEdit-$gtSelStart, RichEdit-$gtGetTextLen(),
    Options);

  // If text is found, highlight it, scroll window to
  // found text, and set focus to RichEdit control.
  // If no text is found, let user know.
  if (Pos != -1) {
    RichEdit-$gtSelStart = Pos;
    RichEdit-$gtSelLength = Dlg-$gtFindText.Length();
    RichEdit-$gtPerform(EM_SCROLLCARET, 0, 0);
    RichEdit-$gtSetFocus();
  }
  else
    MessageBox(Application-$gtHandle,
      "Search text not found.", "Find Text", MB_OK);
}
//---------------------------------------------------
void __fastcall TForm1::ReplaceDialogReplace(TObject *Sender)
{
  // If user clicked Replace All, then do find/replace
  // operation until no more matches are found.
  if (ReplaceDialog-$gtOptions.Contains(frReplaceAll)) {
    static int Count;
    TSearchTypes Options;
    if (ReplaceDialog-$gtOptions.Contains(frWholeWord))
```

```
    Options << stWholeWord;
  if (ReplaceDialog-$gtOptions.Contains(frMatchCase))
    Options << stMatchCase;

  // Find the first match.
  int Pos = RichEdit-$gtFindText(
    ReplaceDialog-$gtFindText, RichEdit-$gtSelStart,
    RichEdit-$gtGetTextLen(), Options);

  // Loop while we are still finding text.
  while (Pos != -1)
  {
    // Same code as in the OnFind event handler.
    RichEdit-$gtSelStart = Pos;
    RichEdit-$gtSelLength =
      ReplaceDialog-$gtFindText.Length();

    // Replace the found text with the new text.
    RichEdit-$gtSetSelTextBuf(
      ReplaceDialog-$gtReplaceText.c_str());
    RichEdit-$gtPerform(EM_SCROLLCARET, 0, 0);

    // Increment the counter so we can tell the user
    // how many replacements we made.
    Count++;

    // Do another search.
    Pos = RichEdit-$gtFindText(
      ReplaceDialog-$gtFindText, RichEdit-$gtSelStart,
      RichEdit-$gtGetTextLen(), Options);
  }

  // Dismiss the Replace dialog.
  ReplaceDialog-$gtCloseDialog();

  // Tell the user what we did.
  char buff[40];
  sprintf(buff, "Finished searching text.\r\n%d replacements made.", Count);
  MessageBox(Application-$gtHandle, buff, "Replace Text", MB_OK);
  // Reset the counter.
  Count = 0;
}

// If not doing a Replace All, then just replace the
// found text with the new text.
else {
  // No text selected? Then return.
  if (RichEdit-$gtSelLength == 0) return;
  RichEdit-$gtSetSelTextBuf(
    ReplaceDialog-$gtReplaceText.c_str());
```

```
    RichEdit-$gtSelLength = 0;
  }

  // Set focus to the RichEdit.
  RichEdit-$gtSetFocus();
}
```

October 1997

Building components, part 2

by Sam Azer

In our September article "Building Components, Part 1," we looked at how a component interacts with the development environment and the program it's linked into. We covered enough material that you could easily create your own non-windowed controls without further information. Windowed controls, on the other hand, are a different ball game--creating such controls requires more integration with the Windows environment and a deeper understanding of VCL.

This month, we'll build an example component. In most cases, you build new components on top of existing ones. So, our example component is an extension to a database combo box that remembers the values you put into it. Since it's also a windowed control, you'll be doing business with VCL. Any project--even a small example--should begin with a detailed analysis. Having a clear idea of what you want to do and how you want to do it is the only way to avoid sitting in front of a debugger day after day. Unfortunately, we don't have space for such an analysis here, so we'll start with a component and a simple program to test it.

Quick start

Listings A and **B** contain the code for the TDBShortList component and a single debug message function. (You can download our sample files as part of the file oct97.zip from the *C++Builder Developer's Journal* Web site, www.cobb.com/cpb.) However, please don't install the component right away. If you're still working on it, you wouldn't want to rebuild the component palette after each small code change. We'll begin by demonstrating how you can develop much of the component without installing it first. **Listing A:** *TDBShortList.H*

```
#ifndef TDBShortListH
#define TDBShortListH
/* TDBShortList.h of June 11th,
for Building Components, Part 2. */

#include
#include
#include
#include
#include
#include
```

```
#define PROP_PUBLISHED(typ, nam, str ) \
private: typ f##nam; \
__published: __property typ nam = \
{ read=f##nam, write=f##nam, str }

#define PROP_EPUBLISHED(typ, nam, str ) \
private: typ F##nam; \
__published: __property typ On##nam = \
{ read=F##nam, write=F##nam, str }

// some helpful but unrelated stuff
#ifndef Okay
#define Okay 0
int debugf( const char *srcfile, const int srcline,
            const char *fmt, ... );
#define LOC __FILE__, __LINE__
#endif

enum SelectType_e { ST_Error=-1, ST_None, ST_Delete,
                    ST_CR, ST_Click, ST_Exit };

typedef __fastcall void (__closure *TUpdateEvent)
( TObject *Sender, Boolean DblClick, AnsiString Text
  );

typedef __fastcall void (__closure *TSelectEvent)
( TObject *Sender, SelectType_e EventType,
int ItemsIndex, AnsiString Text, bool IndexChanged,
bool TextChanged );

class TDBShortList : public TDBComboBox
{
   private:

      bool isLoaded; // true if list has been loaded

      __fastcall void CheckResult(void);
      __fastcall void CallSelect(SelectType_e e);
      __fastcall void CallAddNew(void);
      __fastcall void CallDelete(void);

   protected:

      AnsiString EnterValue;  // value of text
                              // at OnEnter
```

```
   int   EnterIndex;       // value of index
                           // at OnEnter

   virtual void __fastcall DblClick();
   virtual void __fastcall KeyPress(char &Key);

   virtual __fastcall void CheckFirstUse(void)
   { if ( !isLoaded ) { isLoaded++; ListRead();
      }; };

   virtual void __fastcall Click(void)
   { TDBComboBox::Click(); CallSelect( ST_Click );
      };

   virtual void __fastcall DoExit(void)
   {   debugf(LOC,"at DoExit");
      TDBComboBox::DoExit();
   CheckResult(); CallSelect(ST_Exit); };

   virtual void __fastcall DoEnter(void)
   { debugf(LOC,"at DoEnter");
      TDBComboBox::DoEnter();
     CheckFirstUse(); EnterValue = Text;
     EnterIndex = Items->IndexOf(Text); };

public:

  bool ListDirty; // true==list to write to disk

   __fastcall TDBShortList(TComponent* Owner);
   __fastcall ~TDBShortList()
   { debugf(LOC, "destroying TDBShortList" ); };

   virtual void __fastcall ListRead(void);
   virtual void __fastcall ListWrite(void);

   virtual __fastcall void ItemAddNew(
      AnsiString Text);
   virtual __fastcall int ItemDelete(
      AnsiString Text);

__published:

   PROP_PUBLISHED(AnsiString, ListName, nodefault);
   PROP_PUBLISHED(AnsiString, ListDefaults,
```

```
        nodefault);
      PROP_PUBLISHED(bool, AutoWrite, default=true);

      PROP_EPUBLISHED(TUpdateEvent, AddNew,
        nodefault);
      PROP_EPUBLISHED(TUpdateEvent, Delete,
        nodefault);
      PROP_EPUBLISHED(TSelectEvent, Select,
        nodefault);

}; // end of class TDBShortList

#endif
```

Listing B: *TDBShortList.CPP*

```
/* TDBShortList.cpp of June 11th,
for Building Components, Part 2. */

#include
#include
#include
#pragma hdrstop

#include "TDBShortList.h"

static inline TDBShortList *ValidCtrCheck()
{ return new TDBShortList(NULL); }

namespace Tdbshortlist
{ void __fastcall Register()
  { TComponentClass classes[1] =
    {__classid(TDBShortList)};
     RegisterComponents("Cobb Group", classes, 0);
  }
}

__fastcall
TDBShortList::TDBShortList(TComponent* Owner)
                        : TDBComboBox(Owner)
{
    debugf(LOC, "constructing TDBShortList" );
    FAddNew = NULL; FDelete = NULL; FSelect = NULL;
    AutoWrite = true;
```

```
    isLoaded = false;
}

void __fastcall
TDBShortList::ItemAddNew( AnsiString itemText )
{
    debugf( LOC, "ItemAddNew( \"%s\" )",
                 itemText.c_str() );

    Items->Add(itemText);
    if ( Text != itemText )
       Text = itemText;

    ListDirty = True;

    if ( AutoWrite )
       ListWrite();
}

int __fastcall
TDBShortList::ItemDelete( AnsiString itemText )
{
    int i = Items->IndexOf(itemText);

    debugf( LOC, "ItemDelete(\"%s\") indexof()==%d\n",
                         itemText.c_str(), i );

    if ( i == -1 )
       return !Okay;  // item not found

    Items->Delete(i);
    ListDirty = True;

    if ( AutoWrite )
       ListWrite();

    return Okay;
}

void __fastcall TDBShortList::ListWrite(void)
{
    // The ComboBox list is copied to a temp. list
    TStringList *Work = new TStringList;
    Work->AddStrings( Items );
```

```
    // Default items are also copied to a temp. list
    TStringList *Defs = new TStringList;
    Defs->CommaText = ListDefaults;

    // Default items removed from the temp. list
    for ( int i=0; i < Defs->Count; i++ )
    {  int r = Work->IndexOf( Defs->Strings[i] );
       if ( r != -1 )
          Work->Delete(r);
    };

    delete Defs;

    // Then the list of items is written to disk
    Work->SaveToFile( ListName );
    ListDirty = False;
    delete Work;

    debugf( LOC, "%d List Items written to %s",
                  Items->Count, ListName.c_str() );
}

void __fastcall TDBShortList::ListRead(void)
{
    // Start by loading default items
    Items->Clear();
    Items->CommaText = ListDefaults;
    debugf(LOC, "Default List Items Loaded" );

    // Check if we have read access to the list
    if ( access( ListName.c_str(), 4 ) )
       return; // and return if we don't

    // Then add the items in the file
    TStringList *FileList = new TStringList;
    FileList->LoadFromFile( ListName );
    Items->AddStrings( FileList );
    delete FileList;

    debugf(LOC, "List loaded from %s, %d items",
               ListName.c_str(), Items->Count );
};

void __fastcall TDBShortList::KeyPress(char &Key)
{
```

```
    // Let the combobox see the key first
    TDBComboBox::KeyPress(Key);

    // If it's a return key, process any new entries
    if (Key == '\r')
    {  CheckResult();
       CallSelect(ST_CR);
    };
};

// Call user's select event handler, if one exists
void __fastcall
TDBShortList::CallSelect( SelectType_e EventType )
{
    static const char *seltypes[] = {
    "ST_Error","ST_None","ST_Delete","ST_CR",
    "ST_Click","ST_Exit"  };

    debugf( LOC, "@ CallSelect( SelectType_e %s )",
                           seltypes[EventType + 1] );

    if ( FSelect )
    {  int NewIndex = Items->IndexOf(Text);
       FSelect( this, EventType, NewIndex, Text,
        NewIndex != EnterIndex, Text != EnterValue );
    };
}

void __fastcall TDBShortList::CallDelete(void)
{
    if ( FDelete )
       FDelete( this, True, Text );
    else
    {  AnsiString s = "Would you like to remove \""
                     + Text + "\" from this list?";

       if ( ::MessageBox( Handle, s.c_str(),
            "Delete Item?",
            MB_ICONQUESTION + MB_YESNO ) == IDYES )
          ItemDelete( Text );

       CallSelect(ST_Delete);
    };
```

```
    if ( AutoWrite && ListDirty )
       ListWrite();
}

void __fastcall TDBShortList::CallAddNew(void)
{
    if ( FAddNew ) // If one exists...
       FAddNew( this, False, Text );
    else
    {  AnsiString Text2 = Text,
       s = "Would you like to add \"" + Text2
                     + "\" to this list?\r\n";

       if ( ::MessageBox( Handle, s.c_str(),
          "New Item",
           MB_ICONINFORMATION | MB_YESNO ) == IDYES )
          ItemAddNew( Text2 );
    };

    if ( AutoWrite && ListDirty )
       ListWrite();
}

void __fastcall TDBShortList::CheckResult(void)
{
    if ( AutoWrite && ListDirty )
       ListWrite();

    // Quit if nothing or no change...
    if ( (Text == "") || (EnterValue == Text) )
       return;

    // Check if this is a new item
    if ( Items->IndexOf( Text ) == -1 )
       CallAddNew();
}


/*    If the operator dblClicks on an existing entry,
      offer to delete it; if the operator dblClicks
      on a new entry, offer to add it. */
void __fastcall TDBShortList::DblClick()
{
    // Let the ComboBox see the dblClick
```

```
    TDBComboBox::DblClick();
    debugf( LOC, "DblClick(), Text == \"%s\"",
            Text.c_str() );

    if (Text == "")
       return; // no text, so ignore this event

    // Now check for new item
    if ( Items->IndexOf(Text) == -1 )
    {  CallAddNew();
       return;
    } else CallDelete();
};

/* Debugf() - a helpful function but don't
   overflow the internal buffer... */

int debugf( const char *f, const int l,
                           const char *fmt, ... )
{
    char buffer[1024];
    sprintf( buffer, "%s @ %d: ", f, l );

    va_list argptr;
    va_start(argptr, fmt);
    int n = vsprintf( buffer + strlen(buffer),
                                  fmt, argptr );
    va_end(argptr);

    OutputDebugString(buffer);
    return n;
}
```

To begin, choose File | New Application from C++ Builder's main menu. Save the resulting Unit1.CPP file as Develop.CPP and the project as Dev.MAK. In Develop.H, add to the list of includes the line

```
 #include "TDBShortList.h"
```

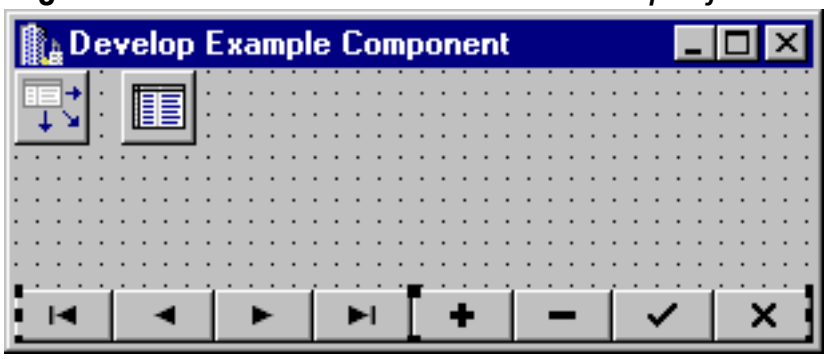
In the class definition for TForm1, add the following lines of code:

```
private: // User declarations
  TDBShortList *Example;
public:  // User declarations
```

```
  __fastcall ~TForm1()
  { debugf(LOC,"destroying form1");};
```

Now, use the form editor to add a DataSource component, a DBNavigator component, and a TTable component to TForm1. Set the TableName property for the table to Color.DB; set the other properties as you'd expect: The DataSet for the DataSource is Table1 and the DataSource for the Navigator is DataSource1. All other properties keep their default values. At this point, your form should look like the one shown in **Figure A**. You'll need a small data file to play with. Since you require only a single small text field, it's no problem to have the program create the file as needed. For this example, add the code from **Listing C** to Form1's OnFormActivate event.

Figure A: *Create this form in the Dev project.*



Listing C: *Form1's OnFormActivate event*

```
// Check if the file exists
if ( access( Table1->TableName.c_str(), 0 ) )
  {  // No, so create it.
  debugf( LOC, "%s not found, creating...",
  Table1->TableName.c_str() ); // Should be
                               //color.db
  Table1->Active = false;
  Table1->TableType = ttParadox;
  Table1->FieldDefs->Clear();
  Table1->FieldDefs->Add("Color", ftString, 20,
                         false);
  Table1->IndexDefs->Clear();
  Table1->CreateTable();
  Table1->Active = true;
} else
  {  debugf( LOC, "%s exists!",
     Table1->TableName.c_str() );
     Table1->Active = true;
};
```

As we discussed last month, the form editor normally writes property values to a form file that's later bound to the executable as a resource. When you execute the form, code in TForm loads the resource, then creates and initializes an instance of each component. Since you're trying to test this component without the form editor for now, you must take a different approach to get the component onto the form: Do it programmatically. In Develop.CPP's constructor for TForm1, add the code shown in **Listing D**.

Listing D: *Develop.CPP's Tform1 constructor*

```
debugf(LOC,
  "adding the Example component to TForm1" );

Example = new TDBShortList(this);
Example->Parent = this;

Example->Name = "Example1";
Example->Left = 75;   Example->Top = 25;
Example->Width = 150;   Example->Height = 24;

Example->DataField = "Color";
Example->DataSource = DataSource1;
Example->ListName = "colors.txt";
Example->ListDefaults = "Red,Yellow,Blue";
Example->AutoWrite = "True";
Example->Enabled = true;
Example->Visible = true;

Example->TabOrder = 1; Example->TabStop = true;
Example->DropDownCount = 8;
Example->ItemHeight = 16;
Example->Style = csDropDown;
```

This code begins by creating an instance of a TDBShortList using the new operator, but the most important step is the next one:

```
Example->Parent = this.
```

This assignment looks like it points the component's Parent property to the form--which would be useless, since the form is supposed to control the component. Fortunately, the assignment is deceiving.

It turns out that Parent is a property of TControl--it's not a variable. Parent's setter function is SetParent, which executes the equivalent of a

```
this->InsertControl(Example)
```

method call. In other words, our innocent-looking assignment statement doesn't tell the component anything about the form--instead, it tells the form about the component! Specifically, it adds a pointer to TDBShortList to TForm1's list of components, thereby making the desired connection between the two. The next time TForm1 is told to show itself, it will display your TDBShortList. The other assignments in **Listing D** set the remaining property values using hard-coded constants.

At this point, you're almost ready to run your demo program. In the C++Builder Project Manager, click the Add File To Project button and add TDBShortList.CPP--a.k.a., your component--to the project, as shown in **Figure B**. Then, save the project. As usual, you can press [F9] to run Dev.EXE.

When the program begins running, there's a short delay as it searches for the Color.DB file. Since it can't find the file, the program creates a new one. When the program shows the form, the component appears, as illustrated in **Figure C**.

Figure B: *Add your component to the project.*

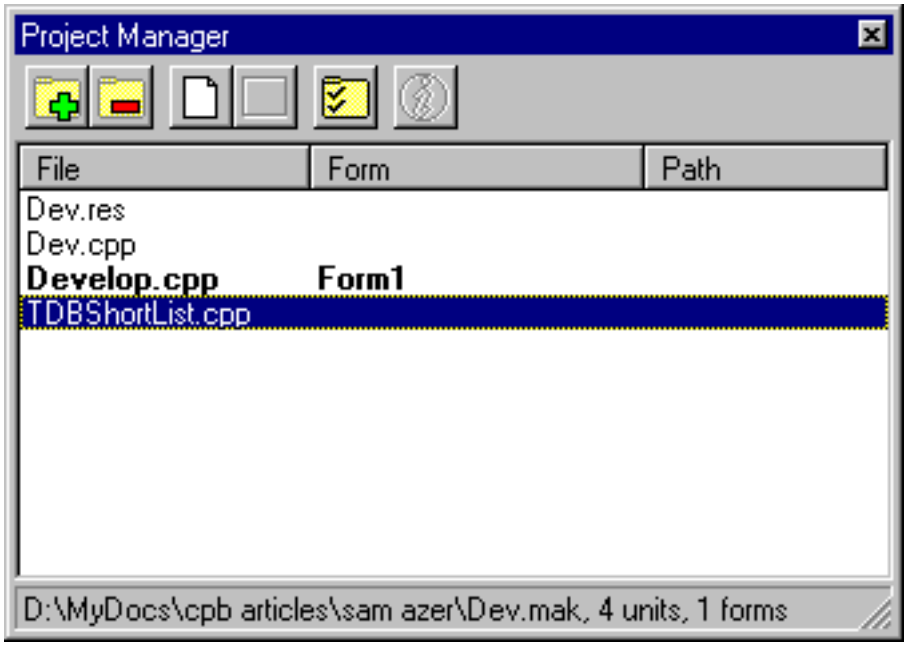
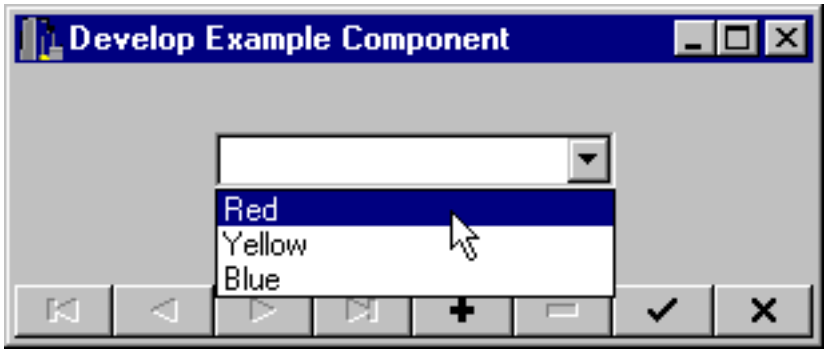


Figure C: *Running Dev.EXE displays your component.*



This is a good time to play with Dev.EXE. Type different color names into the various records of the data file, then look carefully at the debug output that appears in the IDE when you press [Alt][F4]. You can add event handlers for the OnSelect, OnAddNew, and OnDelete event properties. Just add your event-handling method to the TForm1 class, then point the property to it in the constructor as with the other properties. I suggest you also try putting an extra input field on the test form--otherwise, the DoExit() event isn't called when you click on a navigator button. Finally, you can add more debug statements throughout the code to see what happens.

Inside TDBShortList

Each record in Color.DB contains a single text field called Color. The navigator lets you add records to and move around in the file. The TDBComboBox component displays the contents of each record and lets you enter a new color name. All these chores are handled outside of TDBShortList--let's take a look at what TDBShortList does.

TDBShortList's main job is to monitor the values entered into the TDBComboBox text field. So, you need to know when the combo box gets and loses focus. When it gets focus, you want to check the color name; when it loses focus, you want to check again to see if the color name has changed. If the new color isn't in your dropdown list, you should prompt the operator to find out what to do.

TDBComboBox offers OnEnter and OnExit events for this purpose, but you don't want to use those--they should be available to the programmer who'll be using your component. However, a solution is close by: The VCL documentation refers to two virtual methods--DoExit()and DoEnter()--which reside in TWinControl, an ancestor of TDBComboBox. You can override these methods by defining your own DoExit() and DoEnter() methods. They appear as follows in **Listing A**:

```
virtual void __fastcall DoExit(void)
{
   debugf(LOC,"at DoExit");
   TDBComboBox::DoExit();
```

```
        CheckResult();
        CallSelect(ST_Exit);
    };

    virtual void __fastcall DoEnter(void)
    {
        debugf(LOC,"at DoEnter");
        TDBComboBox::DoEnter();
        CheckFirstUse();
        EnterValue = Text;
        EnterIndex = Items->IndexOf(Text);
    };
```

Now, when your component gets and loses focus, TWinControl tries to call its DoExit() and DoEnter() methods--but TDBShortList's versions get called instead.

In this case, you're trying to add your own tasks to the process. You're not replacing TWinControl's DoExit() and DoEnter() and functionality. In fact, you may not know exactly what those functions do or if they've already been overridden by other classes in the hierarchy. So, use the scope resolution operator to call TDBComboBox::DoExit() and TDBComboBox::DoEnter().

Calling a virtual method's ancestors ensures that whatever tasks those methods are supposed to do get done. In this case, the documentation says that the ancestor is TWinControl, but there's no need to tell the compiler--you inherited TDBComboBox, so use its methods and let runtime binding determine which class actually gets the call.

Now getting back to our component, TDBShortList lets the operator add or delete items by double-clicking. TDBShortList also performs better if you check for clicks and carriage returns. TWinControl offers a virtual method called KeyPress(), and TControl offers virtual methods called Click() and DblClick(). Again, override these methods to add your processing to the associated events, and remember to call the ancestors to ensure they do their jobs.

Generating events

So far, you've seen how TDBShortList can override virtual functions within VCL in order to add or replace existing event-handling functionality. Now, let's look at how you create your own events.

The CheckResult() method in **Listing B** (TDBShortList.CPP) is called by DoExit(). Its main purpose is to check the TDBComboBox text to see whether the color name is in the Items list. If not, CheckResult() calls the CallAddNew() function. Effectively, AddNew is an event. In this

case, its trigger is simply new text that has been detected in the combo box. The CallAddNew() function first checks FAddNew, the closure that's published as the OnAddNew property. If the closure is null, the function uses a default event handler. Otherwise, it calls the method that the closure points to.

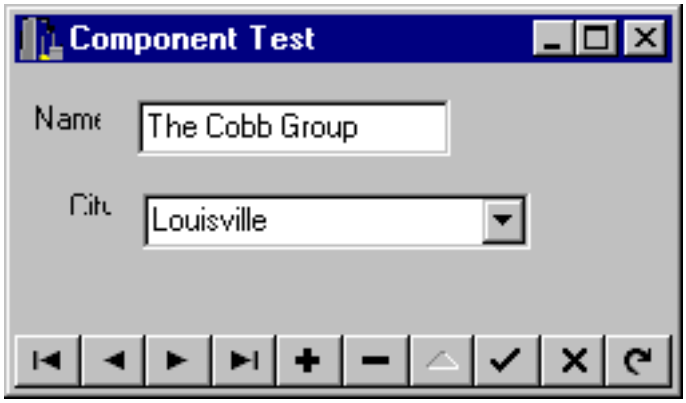
A more complete test

In most cases, a small program like Dev.EXE is the perfect way to get your component started. But there are differences between installing a component using code and installing one using the form editor. The main difference is obvious: You won't be able to test design-time functionality. There are more subtle issues, though. For example, since the component is being installed by the form's constructor (TForm1) rather than by TForm, the order of construction is different. Let's install TDBShortList into the palette and try a more complete test. First, choose Component | Install... from C++Builder's main menu, then click Add... in the Install Components dialog box. Click Browse in the Add Module dialog box, then select the TDBShortList.CPP file. Click Open, then click OK to close the dialog box. When C++Builder finishes the installation, your component palette will have a new Cobb Group tab that includes a TDBComboBox icon, as shown in **Figure D**. Next, create a small program that uses the TDBShortList. **Figure E** shows our Test application. Check the debug messages again--you'll see that the order of construction and destruction has changed, just as you'd expect.

Figure D: *After installation, you'll have a new tab and icon on your component palette.*

Click on image to view full size.

Figure E: *We created this Test application to use our component.*



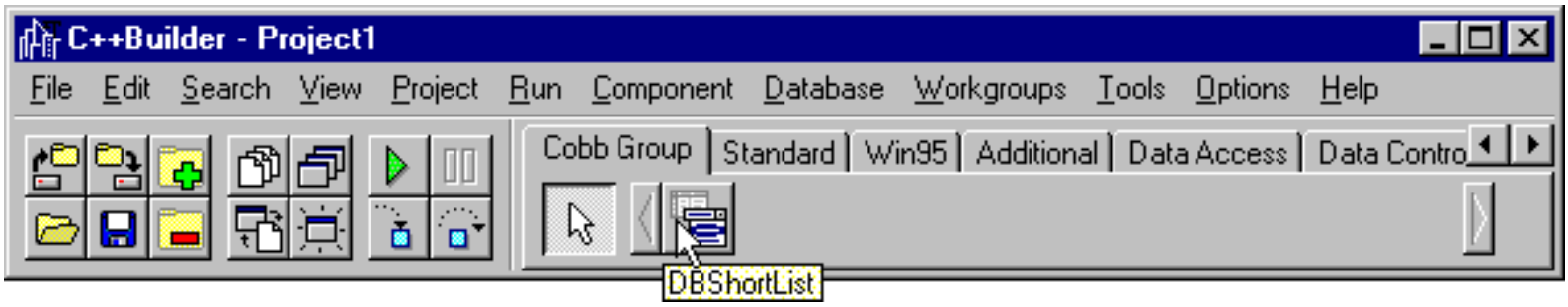
Wrap-up

Not long ago, we assigned constants to global variables and passed long lists of parameters to

static library functions to get things done. We kept stacks of reference books on hand so we could look up names and parameter lists.

Those days are gone. The visual age brings all the elements together for us, and at a surprisingly low cost. Our library code, for example, must now communicate with the development environment; and we publish variables to the object inspector to set their values instead of hard-coding them. It's not a big change--but it sure saves time.

In the first two articles of this series, we've covered the main issues surrounding component building. There's more, though. Next time, we'll add a Help file and build property and component editors for our component. Last--but certainly not least--we'll use a custom icon.


C++Builder - Project1
File Edit Search View Project Run Component Database Workgroups Tools Options Help
Cobb Group Standard Win95 Additional Data Access Data Contro
DBShortList

October 1997

Property macros

by Sam Azer

A moderately complicated class definition can begin to look rather messy as you add properties. For example, in order to define a published integer property called Count that displays as multiples of 123 and has no default value, you'd have to enter code like the following:

```
private:int fCount;
private:int __fastcall GetCount(void)
{ return fCount * 123; };
void __fastcall SetCount(int v)
{ fCount = v/123; };
__published: __property int Count2 =
{ read=GetCount, write=SetCount, nodefault };
```

All this code for a single property! You might have a dozen or more properties in a medium-sized component, creating an unruly mess that's difficult to edit. To make such code look better, the class member definitions often appear together in a group, with the get/set functions following and the properties at the end--but then each property is divided into three separate sections of the class definition.

Fortunately, there's a ready solution to our dilemma. Because the structure of the property-related code is so consistent, the C preprocessor can automate most of the grunt work. Included with this month's sample files in the oct97.zip file is a header called Prop.H, which contains the required macros. You can write all the code from the previous example using the following macro call:

```
PROP_PUBLISHED_LGS(int,Count,nodefault,
{ return fCount * 123; },
{ fCount = v/123; } );
```

The main advantage of using the macros is that everything related to each property is brought together in one section. Also, the code is easier to read and maintain, thereby reducing the chance of errors.

The macros are really quite simple, but they're not fun to read. The previous example is implemented in a way similar to the following:

```
#define PROP_PUBLISHED_LGS(typ,nam,str,g,s) \
private: typ f##nam; \
private: typ __fastcall Get##nam(void) g; \
void __fastcall Set##nam(typ v) s; \
published: __property typ nam = \
{ read=Get##nam, write=Set##nam, str }
```

The documentation at the beginning of Prop.H provides general information on the other available macros.

If you're getting into C++ from Delphi, you may not be familiar with the C preprocessor. To help you see how a macro expands, C++Builder comes with its own preprocessor: CPP32. To try it, copy the Prop.H file to P.CPP (the example code won't compile without a CPP extension) and run the command line

```
cpp32 -P- -DPROPHTEST p.cpp
```

Doing so will leave you with a file called P.I, which you won't be able to open with Notepad. Instead, use WordPad and drag the scrollbar down. Skip the first 79,000 mostly blank lines and go to the bottom of the file--there you'll find a few macro expansions.

October 1997

Owner-drawn status bars

by Kent Reisdorph

In our September article "Sprucing Up Your Status Bars," we discussed the basics of status bars and showed you how to lend your status bars a bit of flash. This month we'll continue that discussion by introducing you to owner-drawn status bar panels.

You can display 3-D text, icons, bitmaps, or other drawings in your owner-drawn status bar panels. These visual goodies not only make your status bars more appealing, they also can provide vital feedback to your users. We'll begin by demonstrating how to set up a status bar for owner drawing. Then we'll show you how to respond to the OnDrawPanel event and how to draw your status bar panel.

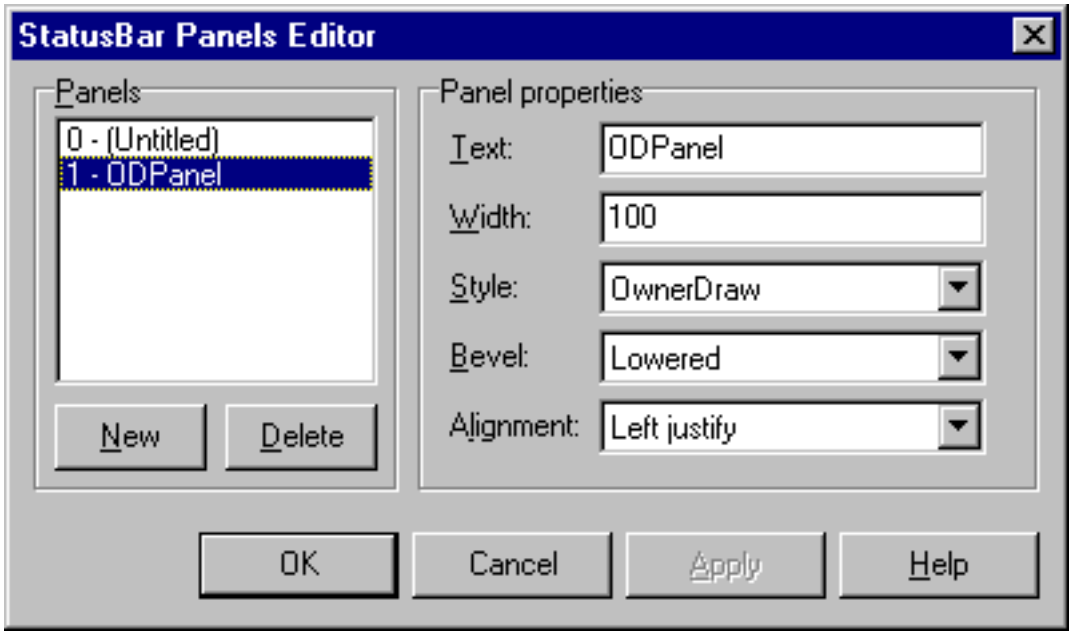
Setting up

You first need to use the StatusBar Panels Editor to tell VCL that you want to draw a particular status bar panel. To invoke the StatusBar Panels Editor, place a StatusBar component on your form, then double-click to the right of its Panels property in the Object Inspector.

The panels editor lets you set up as many panels as you want. Your panels don't have to have the Text property set, since you'll probably refer to the panel by its index in your code. Leave the Style property set to Text for any panels you aren't drawing yourself. Set the Style property to OwnerDraw for those panels you do wish to draw. For example, we edited a status bar containing two panels, one of which was set up for owner-drawing; our panels editor looked like the one shown in **Figure A**.

You can mix owner-drawn and regular panels in your status bars. Typically, you'll reserve the first panel for normal hint text, as we discussed in last month's article. Naturally, this panel shouldn't be owner-drawn unless you want to incorporate some fancy presentation with your hint text. The remaining panels can be owner-drawn or regular panels, as you choose. Once you've created your panels and set the appropriate styles, you're ready to draw.

Figure A: *Use the StatusBar Panels Editor to tell VCL that you want to draw a status bar panel.*



Drawing the panel

Besides having all the events most visual components have, the StatusBar component has an OnDrawPanel event that's generated each time you need to draw an owner-drawn status bar panel. To generate an event handler for this event, locate OnDrawPanel in the Object Inspector and double-click on the value column. C++Builder will generate an empty event handler that looks like this:

```
void __fastcall TForm1::StatusBar1DrawPanel(
  TStatusBar *StatusBar, TStatusPanel *Panel,
  const TRect &Rect)
{

}
```

This event handler gives you a pointer to the status bar (in case you have more than one), a pointer to the particular panel that needs drawing, and a TRect object that represents the size and position of the panel. These items contain all the information you need to draw the panel. Let's take a look.

The Canvas property

Like most visual components, the StatusBar component has a Canvas property. In VCL, the Canvas property, represented by the TCanvas class, encapsulates a Windows device context. You can use the canvas to draw on the status bar; its clipping area is set so that any drawing will automatically be clipped to the size of the current panel.

Let's suppose you want to draw the text Working... in 3-D letters on a panel. The code will be

as follows:

```
StatusBar->Canvas->Brush->Style = bsClear;
TRect temp = Rect;
temp.Top += 1;
temp.Left += 2;
StatusBar->Canvas->Font->Color = clWhite;
DrawText(StatusBar->Canvas->Handle,
  "Working...",
  -1, (RECT*)&temp, DT_SINGLELINE | DT_CENTER);
StatusBar->Canvas->Font->Color = clBlack;
DrawText(StatusBar->Canvas->Handle,
  "Working...",
  -1, (RECT*)&Rect, DT_SINGLELINE | DT_CENTER);
```

We used the API DrawText() function because it lets us draw the text centered on the panel. The code draws the text once in white (offset from center by a couple of pixels), then redraws it in black. Doing so gives the text a 3-D appearance. (Note that the brush style is set to bsClear so the bottom text shows through.) By the way, the cast

```
(RECT*)&Rect
```

is necessary because DrawText() requires a Windows RECT structure. There's really no restriction on the type of drawing you can do in a panel. You can display a bitmap, an icon, text, drawing objects--anything you want.

Getting the right panel

If you have several owner-drawn panels, you'll get an OnDrawPanel event for each one. To draw your panels properly, first determine which panel to draw for a given OnDrawPanel event. You'll do so by using the Panel parameter--check the panel index and do your drawing accordingly. An if statement or switch statement takes care of this task, as follows:

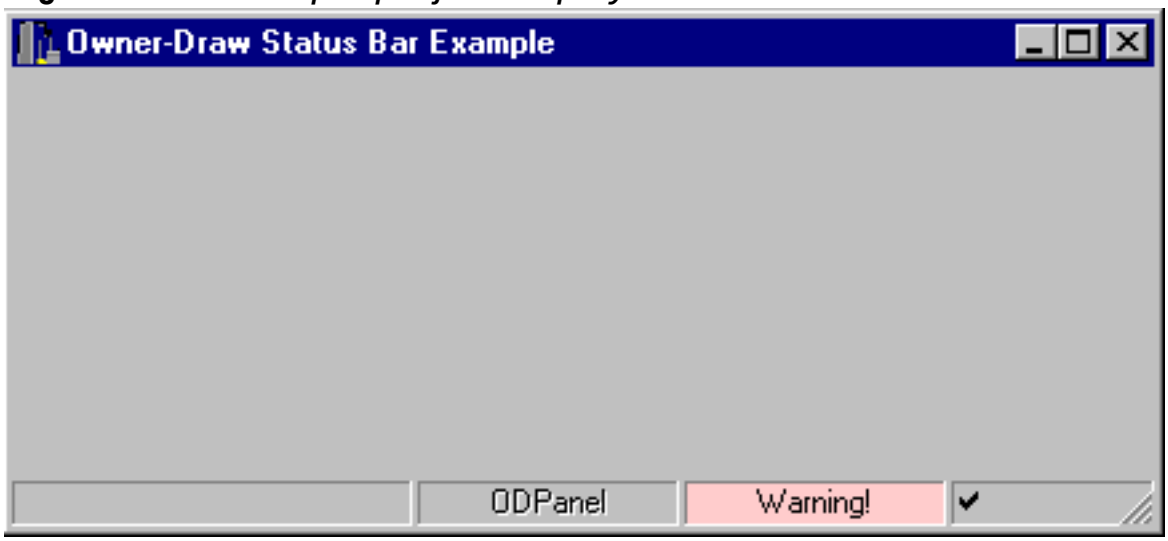
```
switch (Panel->Index) {
  case 1 : DrawPanel1(); break;
  case 2 : DrawPanel2(); break;
  case 3 : DrawPanel3();
}
```

You can also check against the Text property if you've set the text for the panels, but checking the panel index is definitely the most straightforward method.

Putting it together

Listing A shows a sample OnDrawPanel event handler. This code draws three panels. The first panel's text is drawn in 3-D; the next owner-drawn panel is red and displays the text *Warning!;* and the third panel displays a checkmark bitmap loaded from Windows, as shown in **Figure B**.

Figure B: *Our sample project displays three owner-drawn status bar panels.*



Listing A: *OnDrawPanel event handler*

```
void __fastcall
TForm1::StatusBar1DrawPanel(
  TStatusBar *StatusBar, TStatusPanel *Panel,
  const TRect &Rect)
{
  TCanvas& c = *StatusBar->Canvas;
  switch (Panel->Index) {
    case 1 : {
      c.Brush->Style = bsClear;
      TRect temp = Rect;
      temp.Top += 1;
      temp.Left += 1;
      c.Font->Color = clWhite;
      DrawText(c.Handle, Panel->Text.c_str(),
         -1, (RECT*)&temp,
         DT_SINGLELINE | DT_CENTER);
      c.Font->Color = clBlack;
      DrawText(c.Handle, Panel->Text.c_str(),
         -1, (RECT*)&Rect,
```

```
        DT_SINGLELINE | DT_CENTER);
      break;
    }
    case 2: {
      c.Brush->Color = clRed;
      c.FillRect(Rect);
      DrawText(c.Handle, "Warning!",
        -1, (RECT*)&Rect,
        DT_SINGLELINE | DT_CENTER);
      break;
    }
    case 3: {
      Graphics::TBitmap* bm =
         new Graphics::TBitmap;
      bm->Handle =
        LoadBitmap(NULL,
        MAKEINTRESOURCE(32760));
      c.Draw(Rect.Left, Rect.Top, bm);
      delete bm;
      break;
    }
  }
}
```

To try this code, create a new C++Builder application. Place a StatusBar component on the form and create four panels in the status bar. Double-click on the value column next to the OnDrawPanel event in the Object Inspector and enter the code from **Listing A**. (You can also download our sample files as part of oct97.zip from www.cobb.com/cpb.) Try different drawing methods to get a feel for what you can accomplish with owner-drawn status bars.

October 1997

Structuring your code to avoid leaks

by Dan P. Plunkett

Exceptions are an exciting addition to C++ because they allow you to write clearer and safer code. Many C++ experts will also tell you that by using exceptions, you'll end up with code that's both easier to maintain and that contains fewer bugs. I firmly agree, although with a few caveats.

Improperly used, exceptions can lead to resource leaks that are very hard to find. In this article, we'll present a technique called *resource allocation is initialization.* This technique encapsulates the allocation and eventual deallocation of resources, freeing you from doing it yourself.

Here's the problem

Consider the following code snippet:

```
bool process_file(const char *fname)
{
   FILE *f;

   if ((f = fopen(fname, "wb")) == 0)
      return false;
   read_file(f);
   fclose(f);
   return true;
}
```

At first glance, this function doesn't appear to have a problem. However, there's a potential for leaking a file handle. What if the read_file function throws an exception? Since there's no try/catch block protecting the call to fclose, any exception thrown by read_file won't close the file handle. You might try to remedy this dilemma as follows:

```
bool process_file(const char *fname)
{
   FILE *f;
   if ((f = fopen(fname, "wb")) == 0)
      return false;
   try
      {
```

```
      read_file(f);
      }
   catch (…)
      {
      fclose(f);
      throw; /* rethrow exception */
      }
   fclose(f);
   return true;
}
```

While this approach solves the resource leak, it's tiresome, prone to error, and not very elegant. Thankfully, there's an easier way to resolve the problem. Remember the property that automatic variables have? Their destructors are called whenever they fall out of scope, including *each time an exception is thrown*. With this in mind, I'll introduce the following class:

```
class File
{
   FILE *p;
public:
   File(const char *fname, char *mode)
      {
      if ((p = fopen(fname, mode)) == 0)
         throw "Error opening file";
      }
   ~File() { fclose(p); }
   operator FILE*() { return p; }
};
```

With this class, the function becomes trivial (and safer):

```
void process_file(const char *fname)
{
   File f(fname, "wb");
   read_file(f);
}
```

By encapsulating the allocation and deallocation of the file handle into a class and taking advantage of the inherent properties of destructors, we've removed from you the responsibility of freeing the file handle. Now, regardless of whether an exception is thrown, the file handle will always be freed. Note that the File class contains a casting operator that

allows the class to function exactly like a FILE pointer. Therefore, the following code is still perfectly legal:

```
void f(void)
{
   File f(“test.tmp”,”rb”);
   char buffer[80];

   fread(buffer, sizeof(buffer), 1, f);
   …
}
```

Applying the technique to memory allocation

Memory is the most frequently leaked resource in any application, and such leaks are often difficult to find. Fortunately, you can evolve the *resource allocation is initialization* scheme to handle memory allocations as well:

```
template  class Memory
{
   T *p;
public:
   Memory(int size) { p = new T[size]; }
   ~Memory() { delete [] p; }
   operator T*() { return p; }
};
```

This template allows you to encapsulate all your memory allocations in such a way as to guarantee that they’re freed when they fall out of scope, as follows:

```
void f(void)
{
   Memory buffer(512);

   //…use buffer
}
```

As you can see, you can easily extend this solution to any type of resource: file handles, memory, locks, semaphores, and database cursors, just to name a few.

auto_ptr is your friend

Buried in memory.h, which is included with Borland C++Builder, is a generic template class called auto_ptr, which implements the *resource allocation is initialization* scheme generically so you can use it with any VCL object. For example, does your application ever need to instantiate TQuery in a function? Use auto_ptr to ensure that the database cursor is properly destroyed before your function exits, as follows:

```
void __fastcall TForm::ButtonClick(
  TObject *Sender)
{
   auto_ptr Query(new TQuery(this));
   //…use Query
}
```

Conclusion

Finding and correcting resource leaks can be one of the most expensive tasks in a development effort. By using the powerful resource allocation is initialization technique properly, you'll reduce the cost of developing and maintaining your application. Considering that software development comprises more than 90 percent of the expense of an overall system, minimizing costs to correct such leaks is of paramount importance.

September 1997

Sprucing up your status bars

by Kent Reisdorph

If you write applications, then it probably won't be long before you write one that uses a status bar. The basic status bar sits at the bottom of the screen and displays hint text or status information. More sophisticated status bars show key states, cursor position, mouse coordinates, the current time, or other application-specific information. A good application makes proper use of a status bar when and where appropriate, and a status bar can spruce up an otherwise boring user interface. In this article, we'll show how you can add a little pizzazz to your status bars. You'll learn how to create multiple-panel status bars and how to add a clock to your status bar. As an added bonus, we'll show you how to put a progress bar on your status bar.

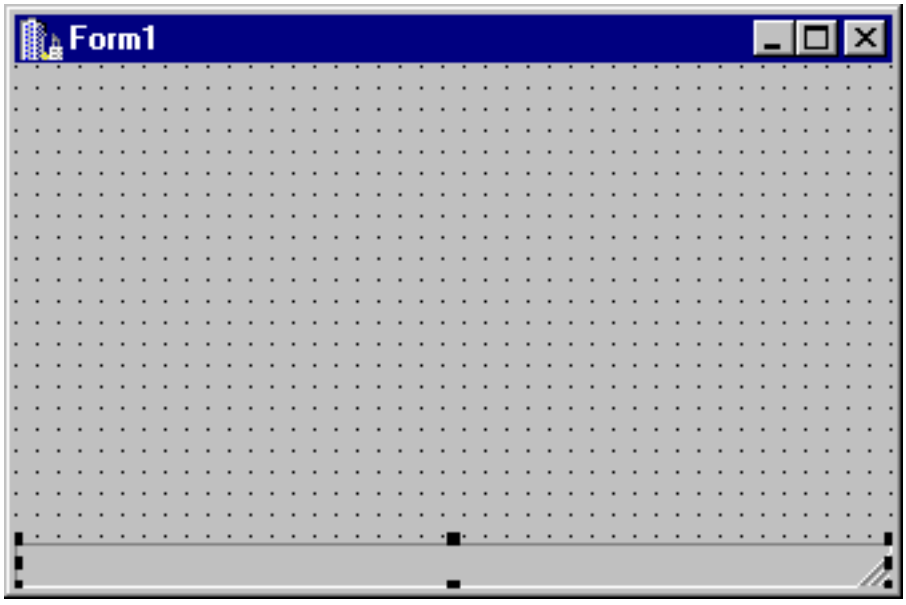
Status bar basics

The StatusBar component encapsulates the Win32 status bar control. Developers who must support 16-bit operating systems dream wistfully of the day when they'll be able to use the StatusBar component (it's available only in 32-bit programs). But don't worry; you can just drop a StatusBar component on your form and hit the ground running. The StatusBar component provides the following features:

- A single panel or multiple panels
- Text displayed in panels
- Owner-drawing of panels
- A grab bar for easy form sizing

When you drop a StatusBar component on your form, you'll notice that it immediately aligns itself along the bottom of the form, as shown in **Figure A**.

Figure A: *The status bar automatically docks itself along the full width of the bottom of*



your form.

VCL makes some assumptions about your design, and in most cases the bottom of the form is exactly where you want your status bar to be. The status bar initially includes a single panel, and the sizing grip appears in the lower-right corner. You'll notice that if you size the form, the status bar also resizes--it always takes up the entire width of the form. These features are courtesy of the Align property, which is set to alBottom by default. The SizeGrip property determines whether the sizing grip appears in the lower-right corner of the status bar. This feature simply makes it easier to size the window on which the status bar resides--its presence doesn't give the user any added functionality, nor does its absence prevent the user from sizing the window. For more on status bar basics, see "Simple or Complex Status Bars?"

Displaying hints in the status bar

Displaying hint text is one of the primary uses for a status bar. The global Application object has an event called OnHint , which is fired whenever the mouse cursor passes over a control whose ShowHint property is set to True. This event doesn't do anything unless you respond to the event. To respond to the OnHint event, you first need to create a function that will handle the event. Once you've created the function, you need to tell VCL to call your event handler when the OnHint event occurs. Finally, in your event handler, you can grab the hint text from the Application object's Hint property and display it in your status bar. That's a lot to absorb without breaking it down into chunks, so let's examine each step in more detail. To create an event handler for the OnHint event, you first need to declare the event-handling function in your main form's class declaration. Switch to your main form's header and type the following line in the private section:

void __fastcall MyOnHint(TObject* Sender);

Now, switch back to the source file for the main form and enter the following lines:

```
void __fastcall TForm1::MyOnHint( TObject* Sender)
{
  StatusBar->SimpleText = Application->Hint;
}
```

This code takes the text from the Hint property of the Application object and displays it in the SimpleText property of the status bar--assuming, of course, that you have on the form a status bar whose SimplePanel property is set to True. The application still won't display hint text at this point, however, because VCL doesn't know to call your event handler. So, you need to hook your event handler to the Application object's OnHint event--the event handler for the form's OnCreate event is generally a good place to perform this step. Double-click on your form's background, and C++Builder will create an event handler for the form's OnCreate event. Type the following line in the event handler:

```
Application->OnHint = &MyOnHint;
```

Now VCL will call your event handler any time the OnHint event occurs. Note that before you can see any hint text displayed in the status bar, you must include components that have their ShowHint property set to True and that have text in their Hint property. Further, the text you enter in the Hint property should be in two parts: the text that will appear in a pop-up window when the mouse cursor pauses over the component, and the text that your MyOnHint() function will display in the status bar. You separate the two parts using the pipe character (|). For example, let's say you have on your form a button that opens a file. You'll enter text like this for the button's Hint property:

Open a File|Opens a file for editing

You can omit either the long hint text or the short hint text if you prefer. For example, suppose you don't want the pop-up hint to show for the button, but you want the long hint text to appear in the application status bar. In that case, you'll enter the following text for the Hint property:

```
|Opens a file for editing
```

Notice that you still use the pipe to indicate that the text you've entered is for the long hint text. If you only want to use the short hint text, then you don't need the pipe.

Time, anyone?

You'll frequently see the current time displayed on the right side of an application's status bar. Fortunately, this effect is fairly easy to achieve. In order to simplify displaying the time,

you should create a status bar with multiple panels; the last panel's text alignment should be right justified. (Use the StatusBar Panels Editor to set the properties for the status bar panel, as we explain in "Simple or Complex Status Bars?")

VCL provides a function that makes getting the current time easy: The Time() function returns the current time based on the regional settings that are in effect on the user's machine. Once you have the time, you need to convert it to a string and format it so that it appears in a standard time format. Again, you don't have to do much work, because VCL provides the FormatString() function. Let's look at a simple example to see how these functions work. The following code snippet gets the current time, formats it in 12-hour format, and stores the result to a String object:

```
TDateTime currentTime = Time();
String format = "h:nn ampm";
String timeString = currentTime.FormatString(format);
```

You can then display the formatted string in a status bar panel. For instance, if the current time is 8:30 p.m., the displayed string will be

```
8:30 PM
```

To display the time in military (24-hour) time, including seconds, use the time-string format of hh:nn:ss. (For a complete description of time formatting options, see the C++Builder online help or the SYSUTILS.PAS source file.) You can condense the time formatting code a bit by combining the previous statements on a single line. In fact, you can display the time in the status bar panel and format it all in one fell swoop, as follows:

```
StatusBar->Panels->Items[2]->Text =
Time().FormatString("h:nn ampm    ");
```

This example assumes that the status bar panel used to display the time is the third panel (panel index 2). Now you know how to display the time, but we haven't talked about when to display the time. You'll likely want to display the time when the application first appears. The main form's OnCreate event is a good place for the initial time display. To continuously update the clock as time marches on (and it always does), you'll need a Timer component. Typically you'll set the timer's Interval property to 1000 so that the timer fires once per second (every 1000 milliseconds). Windows timers are notoriously inaccurate, but they're good enough for the purpose of updating a clock on the status bar. All you need to do is drop a Timer component on your form and then update the status bar in the Timer's OnTimer event. By the way, the sizing grip can get in the way when you display text in the last panel of the status bar, especially if you right-justify the time string. Be aware that you may need to pad the end of the time string with a few blank spaces--otherwise, the sizing grip will cut

off the end of the time display.

Pilgrim's progress?

Many applications need to display a progress bar to show the progress of some lengthy process. In some cases, the status bar is the perfect place to show this kind of indicator. The most obvious example is a word processor that shows a progress bar when the user saves the current file. You can easily add such a feature to your C++Builder applications. VCL already provides the StatusBar and ProgressBar components--all you need to know is how to combine the two.

Placing a basic progress bar on a status bar requires creating an instance of a TProgressBar component at runtime. The progress bar is parented to the status bar so that it appears on top of the status bar. Before you can create the progress bar at runtime, you need to declare a TProgressBar pointer in your main form's class declaration. To do so, switch to your main form's header file and type the following declaration in the private section:

```
TProgressBar* ProgressBar;
```

The TProgressBar class is contained in COMCTRLS.HPP

Normally, you'd have to add the include for this header file in the form's header. This step isn't necessary here, as long as you place the status bar on the form before you create the progress bar (a logical sequence of events). Since both classes are in the same header, C++Builder automatically adds the include for COMCTRLS.HPP. You can now write the code to create the progress bar. Once again, the OnCreate event handler for the form is a good place for this step. (You could also place the code in the form's constructor.) The code to create a typical progress bar on a form might look like this:

```
ProgressBar = new TProgressBar(StatusBar);
ProgressBar->Parent = StatusBar;
ProgressBar->Left   = 2;
ProgressBar->Top    = 4;
ProgressBar->Height = 13;
ProgressBar->Width  = 200;
```

Notice that we use the name of the status bar as both the owner of the progress bar (in the progress bar's constructor) and the parent (when we assign the Parent property). Doing so ensures that the progress bar belongs to the status bar and appears on top of it. When you create a component at runtime, remember to supply the values for any properties you need to modify. In some cases you'll want the default values, and in other cases you'll need to supply different values. In this case, we supply the size and position of the progress bar but

let the other properties remain at their default values. Note that the Left and Top property values are relative to the upper-left corner of the status bar, not of the form. At this point, the progress bar is ready to use. You can assign a value to the Position property just as you would if the progress bar were on your form. You can use this same technique (creating a component at runtime) to place any type of component on your status bar.

Not so fast, progress boy

There are a couple of other things you may want to consider before shipping your application. First, where are you going to display the progress bar? The code in the previous section places it in the first panel, which is where hint text usually appears. If you want to display the progress bar in a panel that will also display text, then you need to hide the status bar until you're ready to show it. For example, your code to display the progress bar might look like this:

```
ProgressBar->Visible = true;
// show progress
ProgressBar->Visible = false;
```

If you use this technique, remember to set the Visible property to False when you initially create the progress bar. If you've dedicated a panel to your progress bar, then you can leave the progress bar visible. Another problem involves aesthetics. The normal progress bar has a sunken 3-D appearance. While fine for most progress bars, this look might clutter up your status bar more than you like. For this reason, you might want to remove the progress bar's border. Your first thought will probably be to set the BorderStyle property to bsNone, as you would for other components--but the ProgressBar component doesn't have a BorderStyle property. To remove the border, you must drop back to the Windows API. The code looks like this:

```
long style = GetWindowLong(ProgressBar->Handle, GWL_EXSTYLE);
style &= ~WS_EX_STATICEDGE;
SetWindowLong(ProgressBar->Handle, GWL_EXSTYLE,style);
```

You first use GetWindowLong() to get the value that contains the extended styles for the component. You then remove the

```
WS_EX_STATICEDGE
```

style (which gives the progress bar its 3-D look). Finally, you reset the extended style using SetWindowLong(). Voila! The border is removed.

For example

Listings A and **B** contain a program that illustrates the concepts we've presented in this article. **Listing A:** *SBARMAIN.H*

```
//---------------------------------------------
#ifndef SBarMainH
#define SBarMainH
//---------------------------------------------
#include
#include
#include
#include
#include
#include
#include
//---------------------------------------------
class TForm1 : public TForm
{
  __published:    // IDE-managed Components
    TStatusBar *StatusBar1;
    TButton *Button1;

    TTimer *Timer1;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall Timer1Timer(TObject *Sender);
  private:        // User declarations
    TProgressBar* ProgressBar;
    void __fastcall MyOnHint(TObject* Sender);
  public: // User declarations
    __fastcall TForm1(TComponent* Owner);
};

//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------
#endif
```

Listing B: *SBARMAIN.CPP*

```
//---------------------------------------------
#include
#pragma hdrstop
```

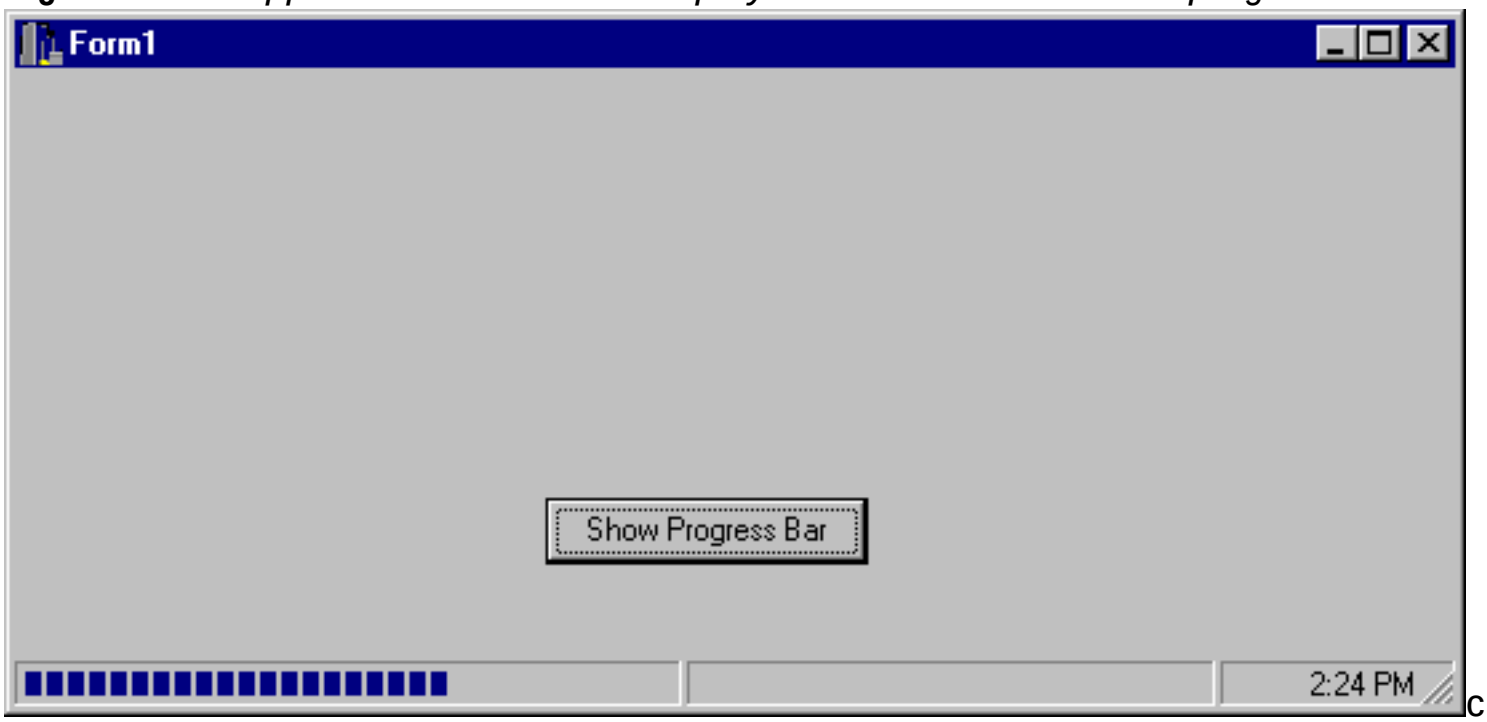
```
#include "SBarMain.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//---------------------------------------------
void __fastcall TForm1::FormCreate(
TObject *Sender)
{
  // Assign hint event to Application's OnHint.
  Application->OnHint = &MyOnHint;

  // Create progress bar, setting the status bar
  // as the parent and owner.
  ProgressBar = new TProgressBar(StatusBar1);
  ProgressBar->Parent = StatusBar1;
  ProgressBar->Left = 2;
  ProgressBar->Top = 4;
  ProgressBar->Height = 13;
  ProgressBar->Width =
  StatusBar1->Panels->Items[0]->Width - 4;

  // Hide progress bar.
  ProgressBar->Visible = false;

  // Change style to remove progress bar's
  // border. Must use API, since progress bar
  // doesn't have a Border property.
  // First get current extended style.

  long style = GetWindowLong(ProgressBar->Handle, GWL_EXSTYLE);

  // Remove the WS_EX_STATICEDGE style.
  style &= ~WS_EX_STATICEDGE;

  // Set extended style again.
  SetWindowLong(ProgressBar->Handle, GWL_EXSTYLE, style);
```

```
  // Show time in last panel in 12-hour format.
  int lastPanel = StatusBar1->Panels->Count - 1;
  StatusBar1->Panels->Items[lastPanel]->Text =
  Time().FormatString("h:nn ampm    ");
}
//---------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  // Show progress bar.
  ProgressBar->Visible = true;

  // A loop to display progress bar.
  for (short i=0;i<100;i++) {ProgressBar->Position = i;
    // Slight delay so it doesn't go too fast.
    Sleep(1);
  }

  // Hide progress bar again.
  ProgressBar->Visible = false;
  StatusBar1->Panels->Items[0]->Text = "Done!";
}
//---------------------------------------------

void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  // Display time on each OnTimer event.
  // First find last panel. Count property
  // is undocumented but is there nonetheless.
  int lastPanel = StatusBar1->Panels->Count - 1;

  // Display current time using 12-hour format.
  // Add a little space to end so size grip
  // doesn't wipe out the end of the text.
  StatusBar1->Panels->Items[lastPanel]->
  Text = Time().FormatString("h:nn ampm    ");
}
//---------------------------------------------

void __fastcall TForm1::MyOnHint(TObject* Sender)
{
  // Set status bar text to value of the
```

```
  // TApplication::Hint property.
  StatusBar1->Panels->
  Items[0]->Text = Application->Hint;
}
```

To run this program, place a StatusBar component, a Button component, and a Timer component on a blank form. Next, modify the status bar's Panels property and create three panels with widths of 250, 200, and 50. (The width of the last panel is immaterial, since it will be automatically sized when the form is sized.) Next, enter the code from **Listings A** and **B** that appears in color. When you run the program, the current time will appear in the last panel of the status bar. Click the button, and a progress bar similar to the one shown in **Figure B** will appear.

Figure B: *Our application's status bar displays the current time and a progress bar.*



cKent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

September 1997

Simple or complex status bars?

by Kent Reisdorph

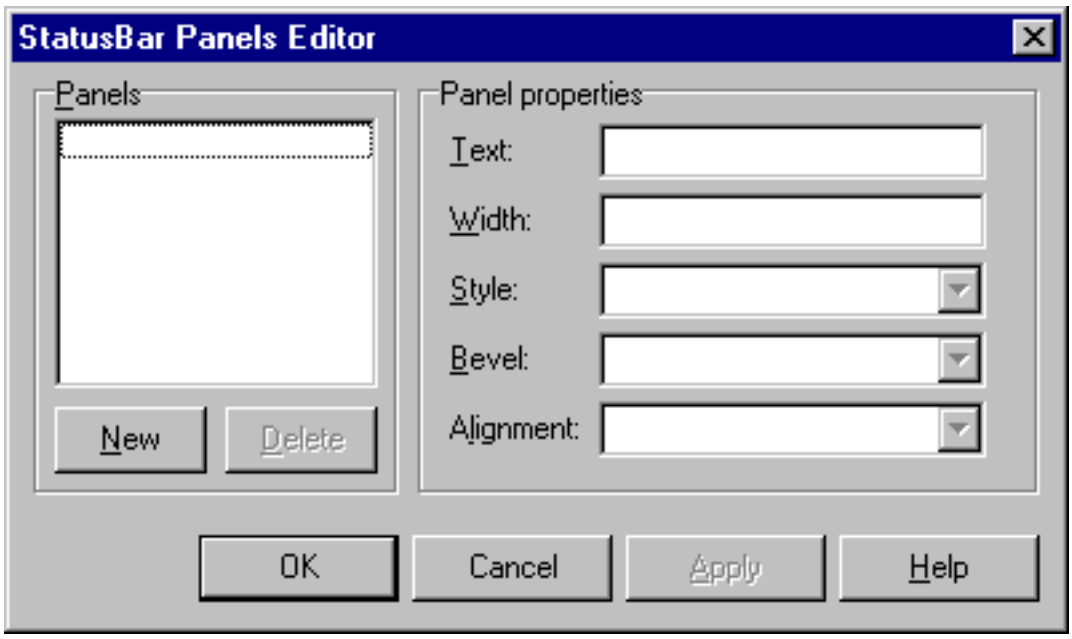
A status bar can include just one panel or multiple panels. A single-panel status bar has one long panel that occupies the entire width of the panel--this appearance is the default and results from setting the SimplePanel property to True. When SimplePanel is True, you can display text in the status bar by modifying the SimpleText property:

```
StatusBar->SimpleText = "Ready";
```

When the SimplePanel property is set to False, C++Builder assumes that your status bar will have multiple panels. For example, an application used to edit text might display status information in the first panel, the cursor position in the next panel, and the editing mode (overtype or insert) in a third panel. The panels serve to visually distinguish the different types of information.

To create the individual panels at design time, you invoke the StatusBar Panels Editor. To see this editor, place a StatusBar component on your form, locate the Panels property in the Object Inspector, and then double-click in the value column. The StatusBar Panels Editor will appear, as shown in **Figure A**, letting you create the individual panels.

Figure A: *You can create and format multiple status bar panels using the Panels Editor.*



We won't describe every panel option, because you can figure them out for yourself--experiment with the Panels Editor to see the effect of different option combinations on the appearance of your status bar.

If your status bar has multiple panels, then the syntax to set the text of the individual panels is a bit more complex than it is for a single-panel status bar. The following code snippet illustrates how to display text in both the first and the second panels of a multi-panel status bar:

```
StatusBar->Panels->Items[0]->Text = "Ready";
StatusBar->Panels->Items[1]->Text = "X: 0, Y: 0";
```

The first panel is panel 0, the second panel is panel 1, and so on. You can modify the properties for each panel (width, text alignment, bevel style, and so on) at runtime as well as at design time, although for most applications such modifications aren't necessary.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

September 1997

Building components, part 1

by Sam Azer

The Borland C++Builder Component palette is a toolbox of powerful classes. It's similar to the compiled function libraries of yesteryear, but with the advantage that you can manipulate each class visually. After you've had a chance to discover how valuable the components are, you'll want to roll your own. In many cases, your components will be extensions to existing ones; in other cases, you'll implement entirely new functionality. In all cases, implementing a component is surprisingly simple, once you know what goes on behind the scenes. In this article, we'll give you a peek.

What's a component?

A component is a class that you must write in such a way that it performs two jobs. First, when you drop a component from the Component palette onto a form, the component talks to the Object Inspector and the form editor in what is called design state. In this state, you edit properties using the Object Inspector and tie in additional event-handling code. The form editor collects property values and writes them to the form file, which is later bound as a resource to the final program. This process allows each component to use persistent data (property values) to adjust its behavior. All these interactions affect the second job that a component must do: perform its run-time program-related duties. To make this process possible, C++Builder enhances ANSI C++ with a few language extensions and adds runtime support code through the Visual Component Library (VCL). Let's take a look.

Language extensions

Three major language extensions support component writing in C++ Builder: __property, __published, and __closure. We'll begin by examining __property and __published.

__property and __published

The __published keyword is another version of the usual C++ public member-access specifier. The two are identical, except that __published properties appear in the Object Inspector at design time. The __property keyword offers another way to control access to member data within a class. However, its most important functions are related to the Object Inspector and support for the persistent data feature of objects in C++Builder.

In its simplest usage, a property is another name for a class member. The value assigned to the member at design time (through the Object Inspector or otherwise) is stored in the form file by default:

```
protected:
  AnsiString FFileName;
__published:
  __property AnsiString FileName =
     {read = FFileName, write=FFileName };
```

This code allows you to set the value of FFileName through the Object Inspector at design time. The value is then saved in the form file. At runtime, the VCL loads the value back into FFileName after construction. If you don't want the value to be persistent, you can disable storage by adding stored=false, as follows:

```
__property AnsiString FileName = {read =
FFileName, write=FFileName, stored=false };
```

Components in the real world need to offer quite a few options to maintain flexibility. They often have long lists of properties with usable default values. Storing all this information in the form file wastes space, and reading it back takes time--which is not desirable, considering that in most cases, few of the defaults change.

To minimize the volume of data in the form file, you can specify a default value for each property. Note that doing so doesn't set the default value--the constructor is responsible for doing that. When writing to the form file, the form editor skips any properties whose values haven't been changed:

```
__property AnsiString FileName = {read =
  FFileName, write=FFileName,
  default="MyFile.txt", stored=true };
```

The __property mechanism also allows you to specify getter and setter member functions to be called instead of directly accessing member data. For simple properties, the Get function takes no arguments, but returns a value of the property's type. The Set function takes an argument of the property's type, but returns no value. You usually use these functions to check for invalid data, as in this example code:

```
int FMy123;
void SetMy123(int x) { if (x >=1 and x <=3)
FMy123=x; } // else do nothing...
int GetMy123(void) { return FMy123; };
__property int My123 = { read=GetMy123,
write=SetMy123 };
```

However, you can also use them for more complex operations, such as database lookups and

updates.

Events and __closure

Generally, components are designed to support a concept without actually handling all the details. For example, C++Builder's System Timer component is designed to call an event at a specified interval. The concept behind the design is that some task must be performed on a regular basis--the component doesn't know or care what the task is. To handle this situation, the component contains a pointer to the code to be executed: the event handler. At design time, you code a new handler and set the pointer. At runtime, the component checks the pointer. If it's null, the component does something sensible--in most cases, it ignores the event. If the pointer isn't null, the component calls the handler.

In C or ANSI C++, you'd normally use a function pointer for this purpose. But in C++Builder, doing so would cause problems, because most event handlers are methods in an instance of a form class. To call them, a pointer must be passed to the form object as a hidden parameter to the method--something a function pointer can't do. To get around this problem, C++Builder introduces pointers of type __closure. These objects contain two pointers: one that points to the class object, and another that points to the method being called. In all other respects, __closure pointers are the same as function pointers.

You'll want to follow a few conventions regarding closures--they aren't enforced, but respecting them is a good idea. First, use a typedef for each event you want to support in your component; the name you use should start with a T and end with the word Event. Next, declare a private or protected member whose name starts with the letter F. Finally, if you want to control access to this member or if you want to publish it in the Object Inspector, create a property whose name starts with On. For example, consider these lines:

```
typedef void __fastcall
  (__closure *TMyUpdateEvent)( TObject *Sender);
...
TMyUpdateEvent FMyUpdate; // user proc. to
                          // override default
                          // update behavior
...
__property TMyUpdateEvent OnMyUpdate =
  { read=FMyUpdate, write=FMyUpdate };
```

You can set a closure to point to a member function in the code or through the Object Inspector. As with any pointer to a function, you should test it before calling it, like this:

```
if ( FMyUpdate )   // if not null
```

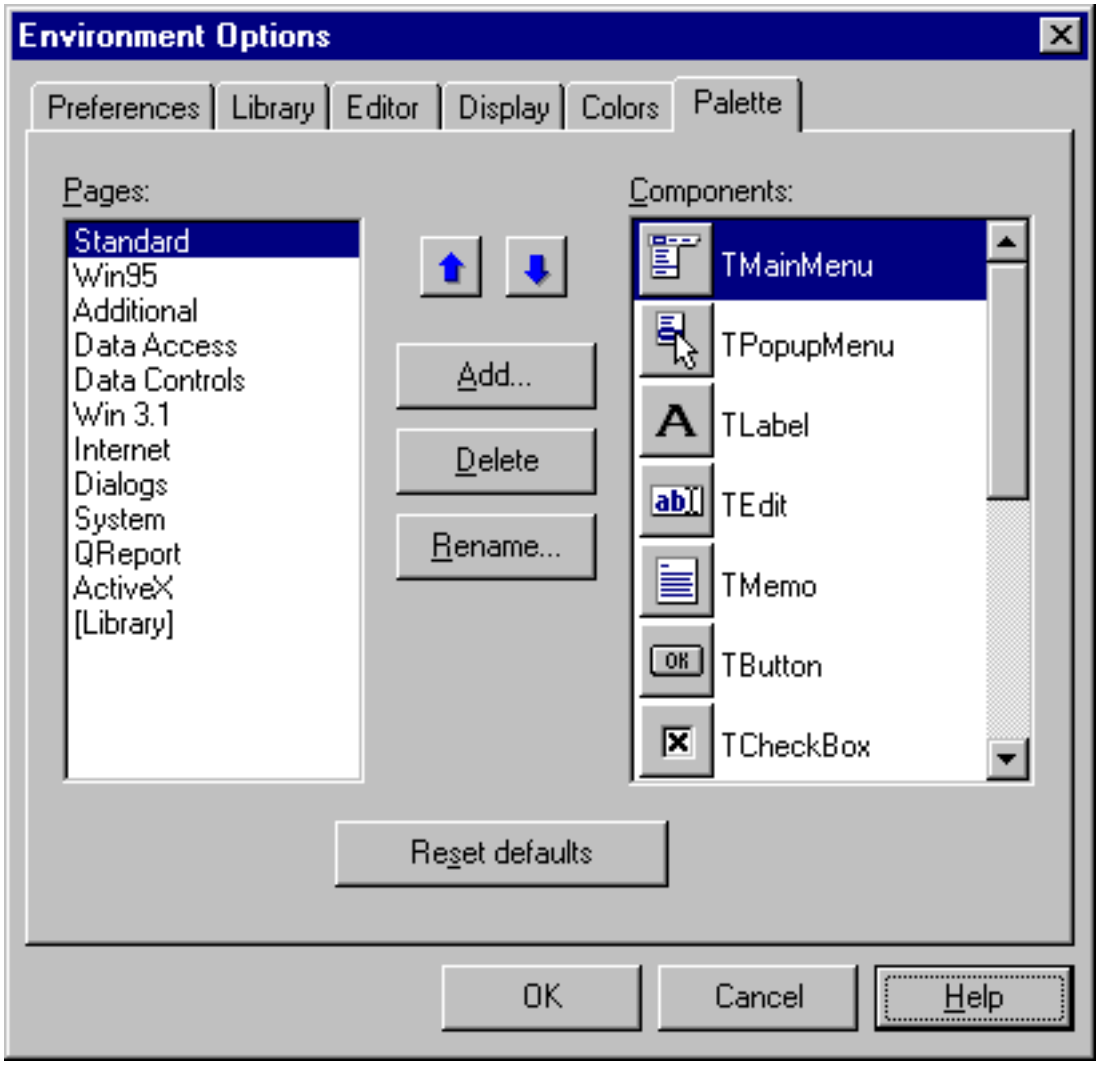
```
FMyUpdate(this); // then call it
```

Design and run

Now that you know what language tools are available, let's look at what happens when the component is being used. We'll begin by examining the Component palette.

The Component palette

To add a component, C++Builder compiles the component's source code and then links the resulting object code into the palette. The default Component palette is a DLL called CMPLIB32.CCL; **Figure A** shows this palette in the Environment Options dialog box. **Figure A:** *C++Builder opens the default Component palette on the Palette tab of the Environment Options dialog box.*



When C++Builder opens a Component palette, it calls a registration function, Register(), for each component.

Register() calls RegisterComponents() to actually install the component in the IDE. If the directory from which the component was loaded includes a resource (RES) file that has the same name as the component file and that contains a 24-bit square icon, then RegisterComponents() uses that icon to represent the component in the palette. Otherwise, it uses a default icon.

The Register() function can also install new property editors for the Object Inspector and component editors for the form editor. Component editors are listed in a component's speed menu. Usually, they're forms that help the programmer configure the component.

Note that all the registration functions have the same name: Register(). You differentiate among them by the namespace in which they're defined, which must have the same name as the header file for the component. For example, if the header file is MyHeader.H, the namespace must be MyHeader.

TComponent and RTTI

VCL includes an important class called TComponent, which handles most of the interfacing between a component and the IDE. This class is defined to use Run Time Type Identification (RTTI); so, any class that inherits TComponent will also use RTTI automatically. As a result, the compiler will collect information about your component class that can be presented in a human-readable form. For example, if you click on an event field in the Object Inspector, the IDE will create a blank function with the correct name and parameter list. This bit of magic is possible because of the RTTI on the closure.

The Object Inspector also uses type information to determine the correct property editor to use. Default editors are available for a variety of types, and you can register any new editors as needed.

Design state

When you drop a component on a form, the form editor creates an instance of the component. This instance is actually running at design time--your constructor and property getter/setter functions will certainly be called, as may other functions. In some cases, this behavior can be a problem. You may need to detect the design state in your code to have the component behave properly. There are different ways to express this test, but they all basically check the csDesigning flag in the ComponentState member of TComponent, as follows:

```
// do nothing if this instance is being
// edited on a form
```

```
if (ComponentState.Contains(csDesigning)) return;
```

Runtime

As we mentioned, the form editor calls TComponent to get property values and writes them to the form file. At runtime, the component will eventually be constructed. Later, functions in TComponent will load property values from the form, which has been bound by the linker as a resource in the EXE file. To allow last-minute initialization after all properties have been loaded, TComponent defines a virtual function, Loaded(), which you can override. Remember to declare the function as virtual in your component's class, and don't forget to call any ancestors!

Wrap-up

Components are an important part of C++Builder. In a future article, we'll demonstrate just how easy it is to build a useful component: a smarter TDBComboBox. After that, we'll examine property and component editors.

September 1997

What a drag!

by Kent Reisdorph

Let's face it: As computer users we live in a drag-and-drop world. Sure, drag-and-drop is a buzz phrase, but this technique is also very useful. When I get a new software package, it isn't long before I find myself trying to drag and drop some element of the program. If dragging and dropping works, I'm happy and on my way again. If it doesn't work, I grumble a little. To produce high-quality applications, you need to implement the features users have come to expect in recent years--and drag-and-drop is definitely one of those features. VCL offers a certain degree of drag-and-drop capability. For example, you can drag and drop from one component to another on a form. You can even drag from a component on one form to a component on another form. However, VCL lacks support for dragging and dropping files from Windows Explorer to an application.

In this article, we'll demonstrate how you can let your C++Builder applications accept files dropped from applications such as Windows Explorer, the Find Files dialog box, or the Windows Desktop. In the course of our discussion, we'll examine the Windows API functions DragAcceptFiles() and DragQueryFile(), along with the WM_DROPFILES message. For a discussion of dragging and dropping between VCL components, see "A Drop in the Bucket."

Laying the groundwork

You'll probably be glad to learn that implementing drag-and-drop in your C++Builder applications is quite simple. You just need to perform three tasks:

1. Tell Windows that your application will accept dropped files.
2. Catch and handle the WM_DROPFILES message.
3. Process the dropped files.

The third step certainly requires the most work. Let's examine these steps individually so you'll have a clearer picture of what's required to support drag-and-drop in your applications.

Receiving

The first step is to tell Windows that you'll accept dropped files. To do so, simply call the DragAcceptFiles() function. This function is prototyped in SHELLAPI.H, so you'll need to include that header in any source code units that use the drag-and-drop functions. After you've included the header, you need to call DragAcceptFiles() as follows:

DragAcceptFiles(Handle, true);

The first parameter of DragAcceptFiles() specifies the window handle of the window that will receive dropped files. Once you've called DragAcceptFiles(), two things happen. First, the mouse cursor will automatically change to the drag cursor when you drag files over the window. Second, the window will receive a WM_DROPFILES message when you drop the dragged files on the window. Windows takes care of these details for you automatically, so you don't have to do anything more than register the window and catch the WM_DROPFILES message.

The second parameter of the DragAcceptFiles() function is a Boolean value that specifies whether the window will accept dropped files. To accept dropped files, pass true for this parameter. To stop accepting dropped files, call DragAcceptFiles() again with this parameter set to false.

Your program may or may not accept dropped files depending on different states, so it's important to be able to turn off drag-and-drop capabilities. You can call DragAcceptFiles() at almost any time, but your form's OnCreate event handler is a good place to initially register your application to accept dropped files.

The window that will accept dropped files can be either a form or a specific component on a form. For example, you may want only a Memo component to accept dropped files, rather than the entire form. In this case, you can pass the Handle property of the Memo component to DragAcceptFiles(). However, doing so won't help you much unless you create a component derived from TMemo, which processes the WM_DROPFILES message. Creating a component is necessary because the window handle passed to DragAcceptFiles() is the window handle that will receive the WM_DROPFILES message when you drop files. If you make the form the recipient of dropped files, then your life will be a little easier. The lesson here is to add drag-and-drop support for individual components only when necessary, since doing so involves extra work.

Catch the wave

Now that you've told Windows you'll accept dropped files, you need to be prepared to catch the WM_DROPFILES message. VCL doesn't provide an event for processing this message, so you'll have to do it the hard way (which, thankfully, isn't too hard). You'll take the easy route and have the main form of your application accept dropped files. In order to catch the WM_DROPFILES message, you'll create a message map table in the form's class declaration. We discussed message maps in our premier issue in the article "Incorporate Custom Message-Handling in Your Applications." Rather than cover that same ground, we'll give you a quick overview. The class declaration for a form that handles the WM_DROPFILES message would look like this:

```
class TForm1 : public TForm
{
__published:    // IDE-managed Components
```

```
private:        // User declarations
  void __fastcall WmDropFiles(
  TWMDropFiles& Message);
public:         // User declarations
  __fastcall TForm1(TComponent* Owner);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TForm)
};
```

Notice the declaration for the WmDropFiles() function in the private section and the message map in the public section. The WmDropFiles() function is our message handler for the WM_DROPFILES message, and the message map table tells the application to call the message handler when that particular message is received.

Getting the drop on the competition

Next, you need to create the function that handles the WM_DROPFILES message. In the previous section, you declared the WmDropFiles() function--now you'll define that function. First, look at this minimal message handler for the WM_DROPFILES message:

```
void __fastcall TForm1::WmDropFiles(TWMDropFiles& Message)
{
  char buff[MAX_PATH];
  HDROP hDrop = (HDROP)Message.Drop;
  int numFiles =
     DragQueryFile(hDrop, -1, NULL, NULL);
  for (int i=0;i < numFiles;i++) {
     DragQueryFile(hDrop, i, buff, sizeof(buff));
     // process the file in 'buff'
  }
  DragFinish(hDrop);
}
```

Let's examine this code one step at a time. You begin by declaring a character array that will hold the filenames you retrieve from Windows. The character array is declared using size MAX_PATH (260) because no filename will be longer than that.

Next comes a call to DragQueryFile(), which gets the number of files dropped. (For more information on this function, see the sidebar "The DragQueryFile() Function.") Now, a for loop retrieves each filename. Notice that the DragQueryFile() call inside the for loop passes the value of i in the iFile parameter.

After the call to DragQueryFile(), the variable buff contains the name of the file corresponding to that index. The first time through the loop, buff will contain the name of the first file dropped; the second time, it will contain the name of the second file dropped; and so on. At this point, you'd do something with each filename as it's retrieved from Windows.

Finally, notice that at the end of the function you call the DragFinish() function, passing the hDrop handle. DragFinish() frees the memory that Windows allocated for the dropped-files information. Without this call, your application will leak a little memory each time you drop files.

Final thoughts

Let's look briefly at one other function that pertains to drag-and-drop operations. You can use the DragQueryPoint() function to determine the mouse cursor's location when files were dropped. Doing so lets you handle the drag-and-drop operation at the form level but use the mouse cursor point to determine which component the mouse was over when the files were dropped. Listings A and B contain a program that implements drag-and-drop in a C++Builder program.

Listing A: *DDMAIN.H*

```
//---------------------------------------------
#ifndef DDMainH
#define DDMainH
//---------------------------------------------
#include
#include
#include
#include
#include
#include
//---------------------------------------------
class TForm1 : public TForm
{
  __published:    // IDE-managed Components
    TMemo *Memo1;
    void __fastcall FormCreate(TObject *Sender);
  private:        // User declarations
    void __fastcall
    WmDropFiles(TWMDropFiles& Message);
  public:         // User declarations
```

```
    __fastcall TForm1(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
  END_MESSAGE_MAP(TForm)
};

//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------

#endif
```

Listing B: *DDMAIN.CPP*

```
//---------------------------------------------
#include
#pragma hdrstop

#include "DDMain.h"
//---------------------------------------------

#pragma resource "*.dfm"
TForm1 *Form1;

//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
}
//---------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  DragAcceptFiles(Handle, true);
}
//---------------------------------------------

void __fastcall TForm1::WmDropFiles(TWMDropFiles& Message)
{
  char buff[MAX_PATH];
  volatile int x = -1;
  HDROP hDrop = (HDROP)Message.Drop;
  int numFiles = DragQueryFile(hDrop, -1, NULL, NULL);
  Memo1->Lines->Clear();
```

```
  for (int i=0;i < numFiles;i++) {
    DragQueryFile(hDrop, i, buff, sizeof(buff));
    if (numFiles == 1)
      Memo1->Lines->LoadFromFile(buff);
    else
      Memo1->Lines->Add(buff);
  }
  DragFinish(hDrop);
}
```

The program, which consists of nothing more than the main form and a Memo component, lets you drag and drop a text file onto the application. When you drop a single file, the Memo component displays the contents of the file. If you drop several files on the application, the Memo component lists the files dropped.

To create this program, place a Memo component on a form. Now, enter the code from Listings A and B into the source unit and header as required. Compile and run the program. Then, experiment with dragging and dropping several files at a time and also with individual text files (source code files will work too, of course).

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

September 1997

A drop in the bucket

by Kent Reisdorph

As we mention in the article "What a Drag!", VCL provides built-in support for drag-and-drop operations between components. Implementing this type of drag-and-drop operation is fairly simple, but a couple of small issues can hold you up if you're not familiar with how VCL's drag-and-drop mechanism actually works. In this article, we'll describe how to accomplish drag-and-drop between components. To do so, we'll examine the OnDragDrop, OnDragOver, OnEndDrag, and OnStartDrag events. We'll also take a look at the DragMode property.

Drag and drop the easy way

If you want to perform simple drag-and-drop operations between VCL components, the process is fairly simple:

1. Set up the source component to allow dragging.
2. Provide event handlers in the destination component for the OnDragOver and OnDragDrop events.

The following sections describe how to accomplish these tasks.

Setting up the source component

You'll begin by setting up the drag source component to allow dragging. The DragMode property controls whether dragging is enabled for a component. This property has two possible values: dmAutomatic and dmManual. The default value is dmManual. When Drag-Mode is set to dmAutomatic, the control will automatically support dragging. When you do support dragging, you'll usually want to set DragMode to dmAutomatic. If, however, you wish to control when dragging is enabled, you'll want to set this property to dmManual. When you do so, dragging won't be enabled until you call the component's BeginDrag() method.

For example, you may want dragging to be possible only when a particular flag is set to true in your application. In this case, your OnMouseDown event handler might contain code like the following:

```
if (allowDrag == true)
ListBox1->BeginDrag(false);
```

The BeginDrag() method's Immediate parameter, determines whether dragging should begin immediately or whether the mouse has to be moved a short distance before dragging begins. Typically, you'll call BeginDrag() with the Immediate parameter set to false so that a mouse click on a control doesn't register as a drag operation. (The rest of this article will assume that you have the DragMode property set to dmAutomatic.)

Once you've prepared the source component for dragging, VCL will display one of two drag cursors during dragging. The no-drop cursor (a white circle with a diagonal slash through it) will appear over a component that doesn't accept dropped items. When the cursor passes over a component that allows dropping, then the cursor will change to the drag cursor (an arrow with a blank piece of paper). The destination component controls whether or not it allows dropping.

The source component is now ready to allow dragging. So, let's turn our attention to the destination component.

Setting up the destination component

Basically, the destination component is where the action is. The first thing you need to do is tell VCL that you'll allow dragging, via the OnDragOver event. When you have C++Builder create an event handler for this event, the program will generate a function like the following:

```
void __fastcall TForm1::Memo1DragOver(
TObject *Sender, TObject *Source, int X,
int Y, TDragState State, bool &Accept)
{
}
```

While this function includes a lot of parameters, you'll mostly be concerned with the Accept parameter. Set this parameter to true to allow dropping and to false to disallow dropping. The Source parameter determines the component that originated dragging. For example, let's say you have a form with two list boxes (ListBox1 and ListBox2) and a Memo component (Memo1). You want to let the user drag from ListBox1 to the Memo component, but you don't want to let the user drag from ListBox2 to the memo. The OnDragOver event handler will look something like this:

```
void __fastcall TForm1::Memo1DragOver(
TObject *Sender, TObject *Source, int X,
int Y, TDragState State, bool &Accept)
{
if (Source == ListBox1) Accept = true;
```

```
else Accept = false;
}
```

That's all the code you have to write. VCL takes care of displaying the drag cursor when the user drags from ListBox1 to the memo and the no-drop cursor when the user tries to drag from ListBox2 to the memo. The user can let go of the mouse when the no-drop cursor is displayed, but VCL won't generate an OnDragDrop event unless the component has said that it will accept dropping. Speaking of the OnDragDrop event, let's take a look at it now. (For a description of the OnDragOver event's other parameters, see the VCL online help.) If a component says that it will accept dropping, then an OnDragDrop event will fire when the user drops something on the component.

For instance, continuing with our previous example, suppose you want to display the contents of the item dragged from ListBox1 to the Memo component. The OnDragDrop event handler will be as follows:

```
void __fastcall TForm1::Memo1DragDrop(
TObject *Sender, TObject *Source, int X, int Y)
{
int index = ListBox1->ItemIndex;
String item = ListBox1->Items->Strings[index];
Memo1->Lines->Add(item);
}
```

You've already determined that the only list box for which you'll allow dropping is ListBox1. Accordingly, you know that if you receive an OnDragDrop event, the source component is ListBox1. As a result, you don't perform any checks to see who the sender is--you simply get the text for the current list box index and add it to the memo's text.

Odds and ends

You can use two other events to further control drag-and-drop operations. The OnEndDrag event is fired for the source component when the user releases the mouse during a drag operation. This event will be fired regardless of whether the mouse cursor was released over a component that accepts dropping. If the mouse was released over a component that accepts dropping, then the event's Target parameter will contain a pointer to the destination component. If the mouse was released over any other component, then the Target parameter will be NULL. Similarly, the OnStartDrag event is fired when dragging begins. This event gives you an opportunity to perform any initialization needed at the start of the drag operation. The OnStartDrag event handler has a DragObject parameter that you can use to create a drag cursor or other drag objects; this parameter also initiates dragging within a control. If you set the DragObject parameter to NULL, then the drag operation will take place within the

control itself. List boxes frequently use this technique to let the user change the item order by dragging items from one position to another. Note that when you use this technique, the object must still respond to the OnDragOver event and return true for the Accept parameter.

Listing A contains a program that let you drag items from a list box to a Memo component. (You can download our sample project files from www.cobb.com/cpb.) The Memo component won't accept items dropped from a second list box on the form. To run this program, place a Memo component, a Label component, and two ListBox components on a form. Set the DragMode property for both list boxes to dmAutomatic. Double-click on the events for the list boxes and memos to generate the event handlers. Run the program and observe how the different components react when you attempt to drag and drop between them. **Listing A:** *VCLDDTst.CPP*

```
//---------------------------------------------------
#include
#pragma hdrstop
#include "DDMain2.h"
//---------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//---------------------------------------------------
void __fastcall TForm1::Memo1DragOver(
TObject *Sender,
TObject *Source, int X, int Y,
TDragState State, bool &Accept)
{
if (Source == ListBox1) Accept = true;
else Accept = false;
}
//---------------------------------------------------
void __fastcall TForm1::Memo1DragDrop(
TObject *Sender,
TObject *Source, int X, int Y)
{
Memo1->Lines->Add(
ListBox1->Items->Strings[
ListBox1->ItemIndex]);
}
```

```
//---------------------------------------------------
void __fastcall TForm1::ListBox1StartDrag(
TObject *Sender, TDragObject *&DragObject)
{
Label1->Caption = "Status: Drag Started";
}
//---------------------------------------------------
void __fastcall TForm1::ListBox1EndDrag(
TObject *Sender, TObject *Target, int X, int Y)
{
if (Target == Memo1)
Label1->Caption = "Status: Drag Accepted";
else
Label1->Caption = "Status: Drag Aborted";
}
```

Kent Reisdorph is a senior software engineer at TurboPower Software and a member of TeamB, Borland's volunteer online support group. He's the author of Teach Yourself C++Builder in 21 Days and Teach Yourself C++Builder in 14 Days. You can contact Kent at kentr@turbopower.com.

September 1997

Command-prompt tips

by Kent Reisdorph

In the August article "Get a Grep!" we discussed running the Grep utility from a command-line prompt. You may not know it, but many tricks are available to make your command-prompt windows more useable. In this article, we'll demonstrate some of those techniques.

Windows 95 versus NT 4.0

I've been using Windows NT 4.0 for so long that I was a bit disappointed when I went to Windows 95 to check out its command-prompt windows. The Windows 95 MS-DOS Prompt window isn't nearly as flexible as the NT 4.0 version.

Windows 95 doesn't inherently possess some of the command-line editing features that NT 4.0 offers--at least, not out of the box. However, the good news is that you can achieve the behavior described in this article by running the DOSKEY utility (provided with Windows 95) from the command line. You'll probably want to go ahead and place DOSKEY.COM in your Windows 95 AUTOEXEC.BAT file so it will load each time you start your computer.

Command-line goodies

The NT 4.0 command prompt (and the Windows 95 command prompt with DOSKEY loaded) offers all sorts of goodies. For instance, you can use the up and down arrows to recall command lines you typed previously. Suppose you change your directory to

```
E:\PROGRAM FILES\BORLAND\CBUILDER\SOURCE\VCL
```

Now, let's say you run a Grep search of the VCL source files, view the results, and then change to some other directory. A short time later you need to go back to the VCL source directory. Aagghhh! You have to type that nasty path again! Or do you? Just press the up arrow key a few times and you'll find the command line you used to change the directory the first time. When the line appears, press [Enter] to re-execute that command.

As long as the command-prompt window remains open, it will maintain a list of the command lines you've entered. The [Page Up] key will take you to the first command entered in the current instance of this command- prompt window, and the [Page Down] key will take you to the last command entered.

Another feature of the NT 4.0 command prompt is a non-destructive backspace. Let's say you typed the long path we mentioned earlier and then realized that you mistyped Program Files. No problem--just press the left arrow key until the cursor is in the proper place, then correct your typing mistake. You can use both the right and left arrow keys to move along the command line.

You also have the ability to type in either insert or overtype mode. For some reason, the default is overtype mode. However, you can toggle between modes by pressing the [Insert] key. You can also use the [Home], [End], [Backspace], and [Delete] keys to edit your command lines. Another handy feature to keep in mind is that the [Esc] key will erase any command-line text.

You can even cut and paste in command-prompt boxes. Simply highlight text in the command-prompt box with the mouse, then right-click on the window's title bar to open the speed menu. Choose Edit | Copy to copy text to the Clipboard. Similarly, if you've copied text from another application (such as a C++Builder help file, perhaps), you can choose Edit | Paste from the command-prompt speed menu to paste the text into the command line.

Command-prompt options

I mentioned that the command-line default is overtype mode. Personally, I prefer insert mode. Not to worry--you can easily change the default editing mode for your command-prompt window. In addition, you can change the command-prompt window's size, font, and foreground and background colors, along with the directory in which the command prompt will start. You can even assign a command-prompt window a hot-key combination that opens the window without using the mouse. (Note that not all of these features are available under Windows 95.)

You accomplish these changes by editing the properties of the command-prompt window. In order to edit these properties, you must first find the shortcut for the window. For the command-prompt icon that appears on the Start menu, you'll find the shortcut in one of the subdirectories of the

```
\WINNT\PROFILES\XX\START MENU\PROGRAMS
```

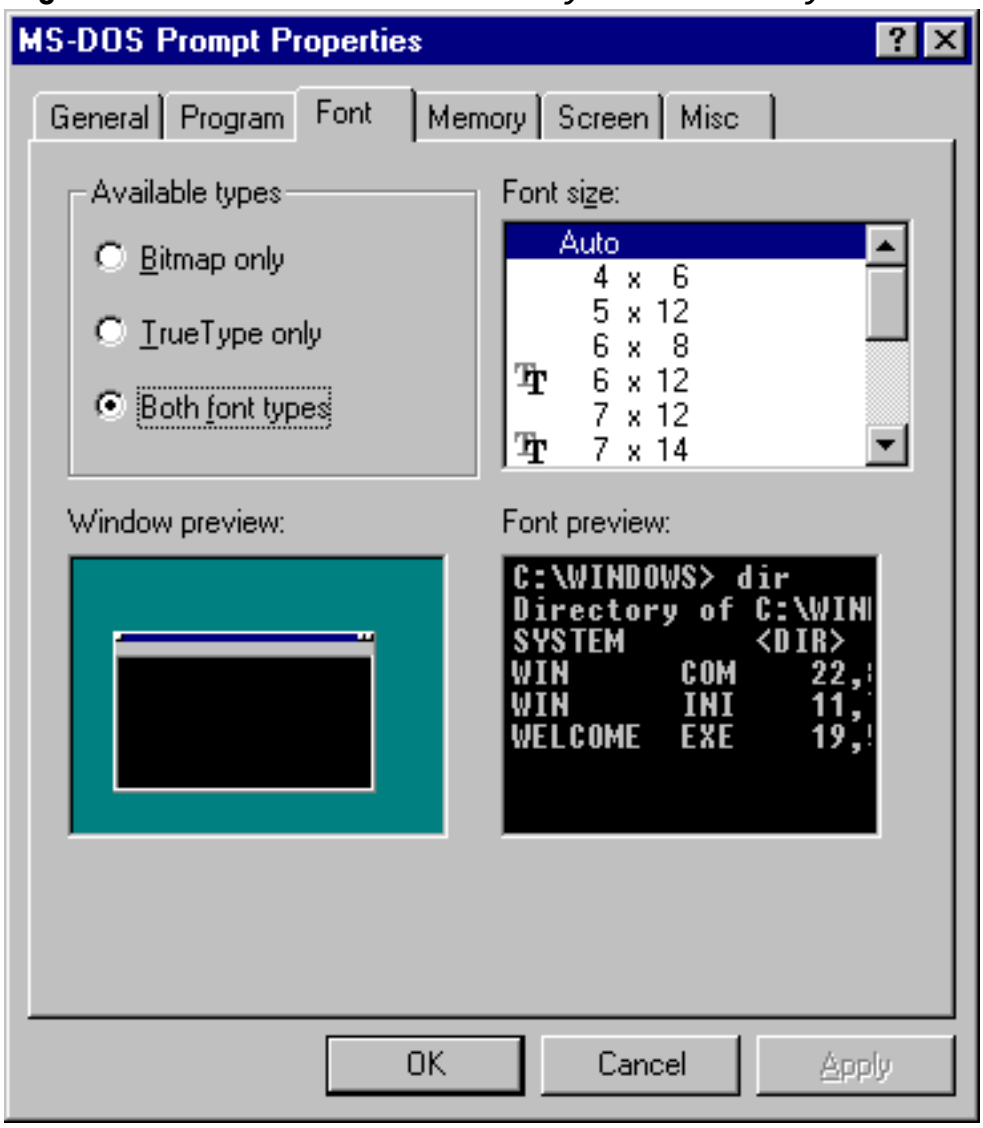
directory, where XX is your user name. If you're logged on as the administrator, then you'll need to look in the ADMINISTRATOR directory. Windows 95 users can locate the shortcut in

\WINDOWS\START MENU\PROGRAMS

Now, right-click on the shortcut for the command prompt, then choose Properties to open the

Command Prompt Properties dialog box. This dialog box has tabs that let you set many options to customize your command-prompt window--for instance, Figure A shows the Font tab in Windows 95.

Figure A: *You can customize many elements of your command-prompt window.*



You can also right-click on the title bar of an open command-prompt window to open the Properties dialog box, but some of the tabs won't appear.

Multiple setups

My system has several command-prompt shortcuts, which facilitate actions that I perform frequently. I simply copied one shortcut in Windows Explorer, pasted it as many times as needed, then renamed the shortcuts to something meaningful and set the options.

One of the most useful features of shortcuts is the Start In field on the Shortcut tab of the Command Prompt Properties dialog (in Windows 95, this is the Working field on the Program

tab). You can enter different paths for the directories in which you want the various command-prompt windows to start.

You'll also find it very handy to assign a hot key to a shortcut. For instance, any time I press [Ctrl][Alt]v, my VCL source command-prompt window appears with the VCL source directory showing. I also created a shortcut for the WIN32.HLP file and assigned [Ctrl][Alt]w as its hot key--now Win32 API help is just a keystroke (or three) away.

Kent Reisdorph is a senior software engineer at TurboPower Software and a member of TeamB, Borland's volunteer online support group. He's the author of Teach Yourself C++Builder in 21 Days and Teach Yourself C++Builder in 14 Days. You can contact Kent at kentr@turbopower.com.

August 1997

Using OWL classes in C++Builder

by Kent Reisdorph

When I first delved into C++Builder, as an OWL programmer I was a little dismayed to find I had to learn a new class library. Still, enough similarity existed between OWL and VCL that I could find my way around.

I noticed right away that VCL, like OWL, has classes called TRect, TPoint, and TSize. "Well, I know what to do with those!" I thought. But it didn't take long to realize that the VCL classes were like the OWL classes in name only--they were just wrappers around the Windows RECT, POINT, and SIZE structures, respectively. Bummer! I *like* those OWL classes!

Take TRect, for instance. It includes lots of overloaded constructors to let you easily create a TRect object. In addition, the overloaded operators and functions make manipulating rectangles easy with the OWL version of TRect. If only I could use the OWL TRect and other great OWL helper classes in my VCL applications.

In this article, we'll show how to incorporate some of the OWL helper classes in your C++Builder applications. In order to use the classes discussed in this article, you'll need Borland C++ 5.02. While you might be able to use classes from other versions of OWL, this article assumes that you have the C++ 5.02 update. We'll also discuss a couple of OWL classes that you probably don't know exist. Let's begin by looking at some specific classes you might want to use.

Winsys, here we come!

The TRect, TPoint, and TSize classes, along with a couple of other classes we'll discuss later, aren't really OWL classes at all. They're listed in the OWL help file, but they're actually what the OWL designers call *Winsys* classes. You can consider the Winsys classes part of the BC++ class library (Classlib). The BC++ 5.0 BC5\INCLUDE\WINSYS directory includes the headers for these files, and the directory BC5\SOURCE\CLASSLIB contains the classes' source files. Since these classes aren't OWL classes per se, you can use them independently of OWL.

Several other classes in the Classlib/Winsys family may be of use--TProfile (an INI file-manipulation class), TRegistry, TString, and others. However, VCL provides good classes for these needs, so you don't have to bother with the Winsys classes that deal with INI files, the registry, string manipulation, and so on.

Putting Winsys classes to work

Assuming you want to use some of the Winsys classes in your C++Builder applications, how do you go about it? There are several ways:

- Add the appropriate Winsys source files directly to your application.
- Link the classes statically using BIDSF.LIB.
- Link the classes dynamically using BIDSF.DLL and the BIDSFI.LIB import library.

The method you use depends largely on your needs, although in most cases, I think you'll prefer the first method. Let's take a look at each of these options and how to implement them in a C++Builder application.

Adding Winsys source files to your project

Adding the source files directly to your project will probably make the most sense for the majority of your C++Builder applications; this is the most complete method and results in the fewest headaches for the programmer. You achieve virtually the same effect as static linking using the static library (which we'll discuss in the next section), but you gain one important benefit: You can use namespaces to distinguish the Winsys classes from their VCL counterparts.

We'll get back to namespaces in a minute, but first, let's take a look at how to add the source files to your project. Implementing this method is a four-step process:

1. Add the required source files to the project.
2. Change the Include path in the Project Options dialog box.
3. Add the include directives to include the necessary headers.
4. Add a define to the Conditional Defines field in the Project Options dialog box.

First, you must decide which source files you need to add to your project. **Listing A** contains a program that uses three Winsys classes: TRect, TUIMetric, and TSystem. You'll find these classes in the source files GEOMETRY.CPP, UIMETRIC.CPP, and SYSTEM.CPP, respectively. In this case, you'll add these files to your C++Builder project.

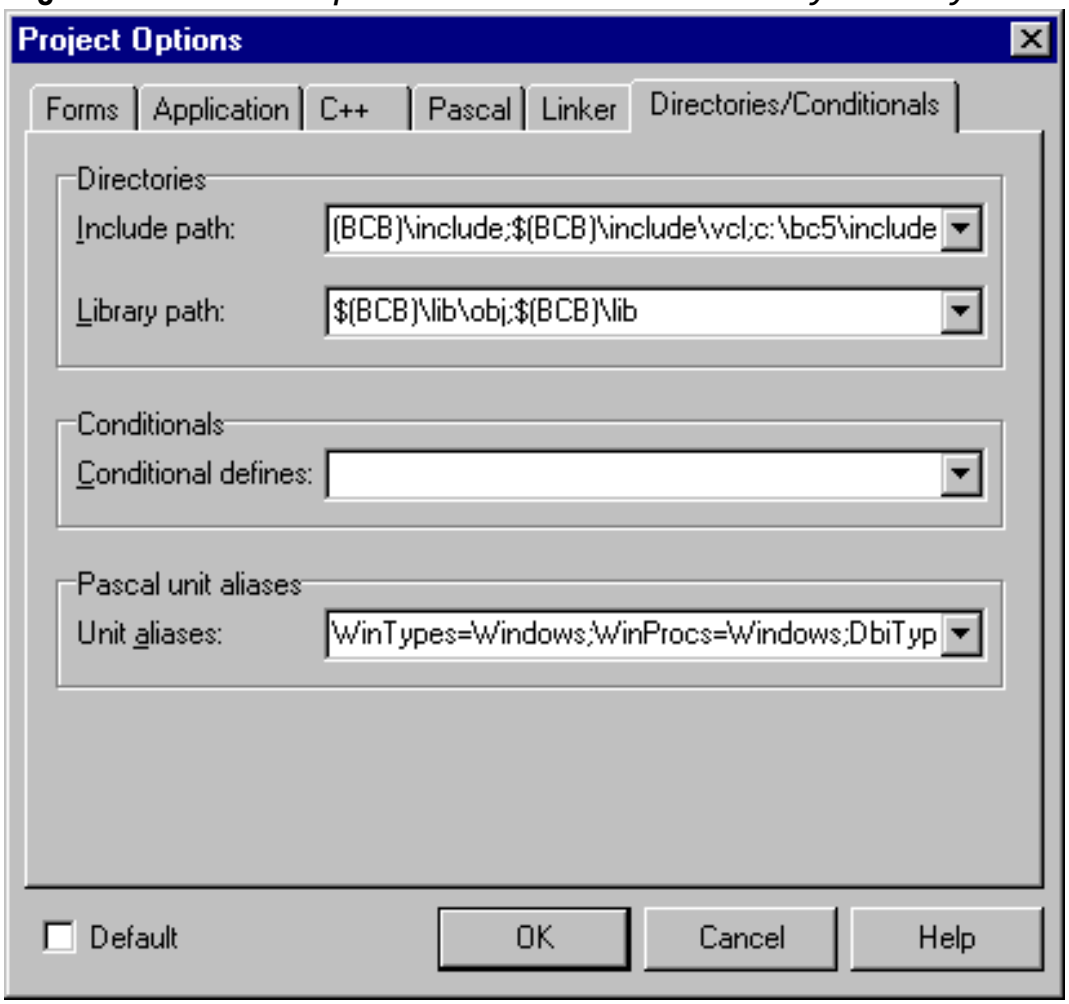
The easiest way to add the files is to use C++Builder's Add File To Project command. To access this feature, simply click the Add File To Project button on the C++Builder speed bar. Doing so opens the file-selection common dialog box. Navigate to your \BC5\SOURCE\CLASSLIB directory and select the GEOMETRY.CPP, UIMETRIC.CPP, and SYSTEM.CPP files. To add the files all at once, hold down the [Ctrl] key while clicking on each of the filenames. Then, click the Open button, which closes the dialog box and adds the source files to your project.

Next, choose Project... from the C++Builder Options menu. In the resulting Project Options dialog box, click on the Directories/Conditionals tab. At the end of the Include Path field, add the path to the INCLUDE directory in which you've installed Borland C++ 5.02. In our case, we added the text

```
;c:\bc5\include
```

as shown in **Figure A**. Be sure to use a semicolon to separate the path you type from the other paths, and take care not to remove the paths already listed. This step is necessary because when you compile the application, the Winsys files will need to include other Winsys headers that C++Builder must be able to locate.

Figure A: *Enter the path to the INCLUDE directory where you installed BC++ 5.02.*



As long as you're in the Project Options dialog box, let's take care of step 4 in this process by adding a symbol to the Conditional Defines field. Type the following in that field:

```
BI_NAMESPACE
```

This entry will tell the compiler to use the ClassLib namespace when it compiles the Winsys files. (For more information about namespaces, see "The Magic of Namespaces" below.) Click OK to close the dialog box.

For the final step (besides writing code), you need to include the headers for the files. For example, the include directive for the GEOMETRY header looks like this:

```
#include <winsys\geometry.h>
```

Since you've already added the base path (C:\BC5\INCLUDE, for example) to the project's include path, you don't need to type the entire path to the Winsys directory when you include the Winsys headers. *This point is important*: You must place the include directive for the Winsys classes *above* any VCL includes. Otherwise, the compiler will confuse classes with common names (the VCL TRect versus the Winsys TRect, for example). Take a peek at **Listing A** to see where we placed the includes for the Winsys files.

At this point, you're ready to type some code and compile. You can begin using the Winsys classes just as you would use VCL classes--with one important distinction. For more information on that distinction, see "The Magic of Namespaces."

Statically linking Winsys classes

You may prefer to link to the pre-built static libraries that come with Borland C++ 5.02 rather than adding the source files directly to your project. Doing so might be desirable if you want to use a number of the Winsys classes in one of your applications. You might find it easier just to add one library file to your project than to add a dozen or more source files. Again, you're dealing with a multi-step process:

1. Add the Borland C++ BIDSF.LIB file to the project.
2. Change the Include path in the Project Options.
3. Add the include directives to include the necessary headers.

In step 1, you'll again use Add File To Project to add the static library file. In this case, add \BC5\LIB\BIDSF.LIB to your project--the Winsys classes reside in this static library file.

Steps 2 and 3 are identical to their counterparts discussed in the previous section. Notice that you don't add the Conditional Define for BI_NAMESPACE in this case. The static library is built without namespaces enabled, so adding that define would have no effect.

This method has a minor drawback. Since the ClassLib namespace hasn't been defined, you can't use it to differentiate between the two versions of TRect that the compiler must deal with. But, because the Winsys version of TRect was compiled without a namespace, it's in the *global namespace* by default.

As a result, you can modify an example shown in "The Magic of Namespaces" and use code like this:

```
using ::TRect;
TRect rect;
```

The double colon by itself signifies the global namespace. This method gives you a little less control over namespace pollution, but it's viable nonetheless. Once you've taken the steps we've

outlined, you're ready to begin using the Winsys classes in your applications.

Dynamically linking the Winsys classes

If you want to use the Winsys classes in a suite of programs, it may make more sense to use the routines in a DLL rather than statically linking them to your application. When you link dynamically, you can use one DLL for all your applications (or DLLs) that use the library routines contained in the DLL, thereby avoiding code duplication.

VCL itself can't be used in a DLL--and for that reason, all C++Builder applications are statically linked. However, that fact doesn't stop you from using the Winsys classes in a DLL. The DLL has already been built for you, so all you have to do is put it to use. Using the Winsys classes in a DLL requires the following steps:

1. Add the Borland C++ BIDSFI.LIB file to the project.
2. Change the Include path in the Project Options.
3. Add the include directives to include the necessary headers.

This list might look like the one from the previous section, but if you look closely you'll see that the library file is BIDSFI.LIB, not BIDSF.LIB.BIDSFI.LIB is the import library file for the BIDS52F.DLL, which you'll use for the Winsys classes.

To add the import library file, use C++Builder's Add File To Project feature to add the BIDSFI.LIB file (located in the \BC5\LIB directory) to your C++Builder project. Steps 2 and 3 are the same as in the previous section.

Whenever you add an import library file to a C++Builder project, the associated DLL will automatically load when your application starts. You'll need to be sure that BIDS52F.DLL is either in the application's directory or somewhere on the path--you can probably find BIDS52F.DLL in your \BC5\BIN directory. As with the previous methods, once you've set up the project to use the DLL, you can start using the Winsys classes as you would any VCL classes.

An example, please

As we mentioned, **Listing A** contains a sample program that uses three of the Winsys classes. To try this example, start with a new project. Place a Label component and a Button component on the form. Then, enter the code in the form's OnCreate handler and the button's OnClick handler as found in **Listing A**. (We've highlighted the code you need to enter in color.)

As you look over the code in **Listing A**, notice the FormCreate() function. This function uses the TSystem class to determine which operating system the user is running. It then displays the operating system information in a Label component.

The Button1Click() function uses Winsys TRect objects to create and manipulate two rectangles. The first rectangle changes the size and position of the application's window. The second rectangle displays a string on the form using the Windows API function DrawText(). The code determines the X position of the displayed text by using TUIMetric to obtain the height of a standard menu. The code finds the Y position by using TUIMetric to obtain the width of a vertical scroll bar.

When you use the Winsys TRect with API functions rather than the VCL TRect, the fourth parameter of the DrawText() function requires a Windows RECT structure pointer. For reasons that are beyond my understanding, the VCL TRect class doesn't have an operator to convert a TRect pointer to a RECT pointer. However, the Winsys version of TRect *does* have such a conversion operator. If you used a VCL TRect, then the DrawText() function would require a cast, and you'd have to write it like this:

```
DrawText(Canvas->Handle, "HelloThere!", -1, (RECT*)&rect, DT_SINGLELINE);
```

Because the Winsys TRect class has a RECT* conversion operator, you can write the DrawText() function without a cast, like so:

```
DrawText(Canvas->Handle, "Hello There!", -1, &rect, DT_SINGLELINE);
```

While this difference probably isn't enough to compel me to use the Winsys TRect on a regular basis, it's an added benefit to using the Winsys TRect over using the VCL TRect. The same benefit ratio is true of the TPoint and TSize classes. For more information on Winsys classes, check out "Undocumented Winsys"

Listing A*MAINFORM.CPP*

```
//---------------------------------------------------
#include <winsys\geometry.h>
#include <winsys\uimetric.h>
#include <winsys\system.h>
#include <vcl.h>
#pragma hdrstop
#include "MainForm.h"

//---------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;

//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
```

```
{
}

//---------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  // Create a string.
  String s = "Operating System: ";

  // If we're running Win95 append "Win95" to the end
  // of the string.

  if (TSystem::IsWin95())
    s += "Win95";

  // Otherwise must be NT so add "Windows NT" to
  //  the end of the string. Also add the NT version
  // number and the build number to the string.

  else {
    s += "Windows NT ";
    s += String((int)TSystem::GetMajorVersion());
    s += ", Build No. " +
    String((int)TSystem::GetBuildNumber());
  }

  // Display the string in the label.
  Label1->Caption = s;
}

//---------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  // Tell the compiler we want the global version
  // of TRect and not the VCL version.
  using ClassLib::TRect;

  // Create a TRect object.
  TRect wndRect;

  // Get the window rect using API GetWindowRect.
  GetWindowRect(Handle, &wndRect);

  // Inflate the rect by 100 pixels all around.
  wndRect.Inflate(100, 100);
```

```
  // Move the window to the new size and location.
  MoveWindow(Handle, wndRect.Top(), wndRect.Left(),
    wndRect.Width(), wndRect.Height(), true);

  // Make x the height of a menu.
  int x = TUIMetric::CyMenu;

  // Make y the width of a vertical scrollbar.
  int y = TUIMetric::CxVScroll;

  // Create a TRect object using x and y as the top
  // and left coordinates.
  TRect rect(x, y, 200, 40);

  // Use it in the API DrawText call.
  DrawText(Canvas->Handle, "Hello There!", -1, &rect, DT_SINGLELINE);
}
//---------------------------------------------------
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1997

The magic of namespaces

by Kent Reisdorph

VCL includes TRect, TPoint, and TSize classes as well as Winsys classes with the same names. So, as we discuss in the accompanying article, you need a way to tell the compiler to differentiate between the two classes. The C++ namespace feature provides that means. Namespaces prevent the compiler from confusing two different classes having the same name.

When you add Winsys source files to your project in the accompanying article, you add an include to the Conditional Defines. Let me explain why. Each Winsys header contains a section that looks like this:

```
#if defined(BI_NAMESPACE)
  namespace ClassLib {
#endif
```

This code tells the compiler, "If the symbol BI_NAMESPACE is defined, then place the following code in the ClassLib namespace." Now, what does that statement mean? Take the following code as an example:

```
TRect rect;
```

This code obviously declares and instantiates a TRect object. But is it the VCL TRect class or the Winsys TRect class? At this point the compiler can't tell, so it will issue a compiler error stating Ambiguity between Windows::TRect and ClassLib::TRect. (The VCL TRect class is in the VCL Windows namespace.)

To avoid this problem, you need to declare the TRect variable using a namespace qualifier. Let's say you intend to use the Winsys version of TRect. In this case, you should write the code as follows:

```
ClassLib::TRect rect;
```

Now the compiler knows which TRect class to use and goes merrily on its way compiling the unit.

However, you can take this process one step further and make the code even more simple. To do so, incorporate the using keyword as follows:

```
using ClassLib::TRect;
TRect rect;
```

This code tells the compiler, "From here on, when you see TRect, use the ClassLib version of TRect." You can always change back later to the VCL TRect by issuing another using statement. Such code prevents you from having to add the namespace qualifier to every TRect declaration in your code.

You'll notice in Listing A of the accompanying article, "Using OWL Classes in C++Builder," that no namespace qualifier is required for the TSystem and TUIMetric classes--those class names are unique to the Winsys classes and don't even exist in VCL. Since there's no chance of conflict, you don't need to qualify those declarations with namespaces. If you had a class of your own named TSystem, then you'd again need the namespace qualifier to differentiate the Winsys TSystem class from your own.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1997

Undocumented Winsys

by Kent Reisdorph

A couple of Winsys classes aren't documented in Borland C++. One of these is the TUIMetric class, which encapsulates the Windows API function GetSystemMetrics(). While the accompanying article, "Using OWL Classes in C++Builder," isn't about system metrics, a quick example will help you understand just what this class does. Let's say you want to find the height of a single-line menu bar. You could use the API as follows:

```
int width = GetSystemMetrics(SM_CYMENU);
```

Or, you could use TUIMetric to accomplish the same task. By using the TUIMetric class you can shorten the line a bit:

```
int width = TUIMetric::CyMenu;
```

It's up to you to decide whether using TUIMetric provides enough savings to make using it in your applications worthwhile.

You may also want to check out the Winsys class TSystem. This class provides functions for querying the operating system for information like the current OS (Windows 95 or NT), the version number, the build number, and so on. Functions include IsNT(), IsWin95(), Has3dUI(), GetProcessorType(), GetBuildNumber(), and more.

Like TUIMetric, TSystem is undocumented as far as I know, so even if you're a seasoned OWL programmer you may not be aware that it exists. This class may prove handy if your application needs to know such OS information as the specific version of Windows the user is running under.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1997

Get a Grep!

by Kent Reisdorph

Finding a specific text string in a source file is a common task for any programmer. You may need to find a method or variable in one of your own source files, or you may want to search third-party source files and headers for a certain text string. Regardless of what you're searching for, you need a quick and easy way to perform the search.

C++Builder comes with a command-line utility called GREP.EXE (commonly called Grep) that lets you search for text in files. You can use Grep to perform simple text searches as well as more complex searches. In addition, you can search across directories and look in many different file types. In this article, we'll explain how to get the most out of Grep. We'll describe how to search for text in files, how to capture the Grep output to a text file, and how to perform more complex searches.

Grep basics

Despite its power, Grep isn't integrated into the C++Builder IDE; instead, you must run it from the command line. While this process might seem a bit old fashioned, you'll get used to it in a hurry--and you might even learn to like it!

If you're a new C++Builder user, you may not see the need to search files for text. However, over time you'll come to understand the benefit of searching source files for a particular text string. For example, I sometimes wonder how a certain method works in VCL, but I can never remember which VCL source files contain which methods. It doesn't matter, though, because I can Grep the VCL source for the method I want to inspect. Let's take a look at how you use Grep.

The syntax for Grep is fairly straightforward:

```
grep [options] searchstring filestosearch
```

Here's a real-world example that illustrates this syntax. Let's say you want to search all C++ source files in the current directory for the text string TForm. You first open a command prompt box, then switch to the directory containing the files you wish to search and type the following at the command line:

```
grep TForm *.cpp
```

When you press [Enter], Grep will search all files with a CPP extension and list any matches it

finds. For example, this search resulted in the following output on my system:

```
File SPMain.cpp:
: TForm(Owner)
else TForm::Dispatch(Message);
File SPAbout.cpp:
: TForm(Owner)
File RegMain.cpp:
: TForm(Owner)
File MsgTBar.cpp:
: TForm(Owner)
File MsgMain.cpp:
: TForm(Owner)
// checked then we just call TForm::Dispatch() and
TForm::Dispatch(Message);
```

If Grep finds any matches, it displays in the console window the name of the file containing the search string, followed by any occurrences of the string. As you can see, Grep displays the entire line containing the matched string. (Note that Grep is a search utility only--you can't use it to search and replace text in files.)

Simple searches

Now, let's look at a few aspects of Grep that will save you some frustration. First, Grep searches are case sensitive by default. We'll discuss later how you can turn off case sensitivity through Grep's command line options.

Also, if you're searching for a multiple-word phrase, you must enclose the phrase in double quotes (single quotes won't work). For example, let's say you want to search all C++ source files for the text = new. The command line will look like this:

```
grep "= new" *.cpp
```

If you typed this line and failed to use the quotes, then Grep would generate a listing of all lines with an equal sign--it would ignore the new part of the string.

You can search multiple file types by separating each file type with a space. Suppose you want to find all occurrences of TForm in both your source files and your headers. In this case, you'll type the following at the command line:

```
grep TForm *.cpp *.h
```

Grep will search both CPP and H files and display all lines that contain the search text.

Complex searches via regular expressions

Grep can search for certain special characters and conditions using regular expressions. Regular expressions are enabled by default. Using regular expressions, you can search for the end of a line, the beginning of a line, numbers only, characters only, and more. Table A shows the Grep regular expression characters along with their descriptions. (This discussion is pertinent whether you run Grep from the command line or from a Grep add-on such as Triplex.)

Table A: *Grep regular expressions*

Expression	Description
^	Start of line
$	End of line
.	Any character
\	Quote next character
*	Match zero or more
+	Match one or more
[az1-3]	Match a, z, and 1 through 3
[^az]	Match all but a and z

A picture is always worth a thousand words, so let's look at how you might use some of these expressions. For our first example, let's say you want to find the variable x--but only where it starts a line of code. In this case, you'll execute Grep with this command line:

```
grep ^x *.cpp
```

The circumflex (^) tells Grep to show all occurrences of the variable x, but only if that variable occurs at the beginning of the line.

The end-of-line character ($) works in a similar way. For example, if you want to find the variable x anywhere it occurs at the end of a line, you can use the following statement:

```
grep x;$ *.cpp
```

The backslash character (\) tells Grep to treat the character following the backslash not as a

regular expression character, but rather as a literal character. For example, let's suppose you want to search for all occurrences of MyString., to find any places where you called a method of the MyString object. You'd probably first try this line:

```
grep MyString. *.cpp
```

The problem is, the period is a regular-expression character that matches any character in the file you're searching. So, Grep will find matches for MyString followed by any character. Consequently, you need to tell Grep that you want to search for a literal period and not to use the period as a special character. The correct syntax is as follows:

```
grep MyString\. *.cpp
```

Notice the backslash before the period. Now Grep will report only occurrences of MyString followed by a period. Remember, you'll also need the backslash when the search string includes square brackets ([ ]) or parentheses (( )).

Let's look at one more example of a regular expression search. Say you want to search for a class named TMyBigLongClassName. Rather than typing in the entire class name, you can use the + regular expression character--this character is similar to the asterisk wildcard used in filenames. For example, this statement

```
grep TMyBig+ *.cpp
```

will tell Grep to search for words beginning with TMyBig and ending with any number of characters.

Using regular expressions can help narrow your searches so you receive more meaningful search results. Note that you can also use regular expressions in the C++Builder code editor's Find Text and Replace Text dialog boxes. By using regular expressions in your find and replace operations, you can more quickly and more accurately deal with text in your source code units.

Grep options

Grep provides several command-line options that let you narrow and refine your searches. We won't discuss every possible option, but we'll touch on those you'll probably use the most. Table B lists the most commonly used Grep options.

Table B: *Commonly used Grep options*

Option	Description
-d	Search subdirectories
-i	Ignore case
-n	List line numbers
-r	Regular-expression search (on by default)
-z	Verbose output

Most of the command-line options are self-explanatory. To search the current directory and all subdirectories for a given string, you use the -d option as follows:

```
grep -d TForm *.cpp
```

To add to this example, you can also opt to perform a case-insensitive search by using the -i switch. In that case, the command line will look like this:

```
grep -d -i tform *.cpp
```

To remove an option from the search, add a minus sign after the option. For example, regular expressions are on by default. If you want to perform a search without using regular expressions, you can use a line like the following:

```
grep -r- (MyString. *.cpp
```

This example turns off regular expressions, thereby allowing Grep to treat the parenthesis and the period as literal characters.

For a complete list of Grep command-line options, look for the Grep topic in the help file BCBTOOLS. Note that the -u option, which is supposed to let you set the default Grep options, has no effect with the version of Grep that comes with C++Builder.

Capturing Grep output

When you run Grep from the command line, the search will often yield a large number of matches that quickly scroll off the screen. You can press [Ctrl]s to pause Grep momentarily so that you can read the results of the search. (You can also press the [Pause] key if your keyboard has one.) Pressing [Ctrl]s once again resumes the display. Press [Ctrl]c once to abort the search.

Sometimes, though, you'll want to capture the results of a Grep search to a file. You can do

that easily by redirecting the output of the console window to a text file. To redirect the Grep output to a file, simply follow the Grep command line with a closing angle bracket (>) followed by the name of a text file. For example, the line

```
grep -d TForm *.cpp > results.txt
```

will place the results of the search, if any, in a file called RESULTS.TXT. (When you redirect output to a text file, Grep doesn't display any search matches in the console window. You'll have to open the text file to see the results.)

Once you've redirected the search results to a text file, you can open the file with any text editor and view the results. Note that the redirection is a function of the operating system and not a function of Grep itself: You can redirect the output from any of the C++Builder command-line utilities, such as TDUMP.EXE.

In a similar vein, you can use the Clipboard to copy text directly from the command prompt window. Next month, we'll discuss how you can use the Clipboard with command prompt boxes and offer some other handy command prompt tips.

In a perfect world...

Many current development environments contain a built-in Grep. When properly implemented, as in Borland C++ 5.0, the integrated Grep is even more helpful. Unfortunately, C++Builder doesn't have a built-in Grep utility, although rumor has it that the next version of C++Builder will have one.

In the meantime, you can continue to run Grep from the command line, or you might want to look at third-party Grep utilities. One of the most popular Grep add-ons for Delphi--and now for C++Builder--is part of John Howe's Triplex suite. Triplex is a collection of experts for Delphi and C++Builder, one of which is a Grep Expert. For the moment, Triplex is freeware and is available for download at ds.dial.pipex.com/triplex. You'll definitely want to take a look.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1997

An AnsiString class reference

by Kent Reisdorph

The AnsiString class (or String, for short) that comes with C++Builder is designed to work and play well with VCL. However, the documentation for the String class is somewhat lacking--OK, it's terrible. We'll do our best to remedy that situation in this article by describing the most commonly used String class operators, constructors, and functions. (We won't deal with wide-character support functions, however.)

Constructors

The String class includes constructors to let you create a String object from different data types. Table A shows the constructors and provides an example of each constructor's use.

Table A: *String class constructors*

Constructor	Description	Example
AnsiString();	Constructs an empty String object	String s;
AnsiString(const char* src);	Constructs a String object from a char* (a character array)	String ("This is a test"); char* buff = "Hello"; String s2(buff);
AnsiString(const AnsiString& src);	Constructs a String object from another String object	String s1 = "String One"; String s2(s1);
AnsiString(const char* src, unsigned char len);	Constructs a String object from a char* with the length specified in len	String s("", 255);
AnsiString(char src);	Constructs a String object from a char data type	char c = '\n'; String lf(c);
AnsiString(int src);	Constructs a String object from an int data type	int x = 200; String s(x);
AnsiString(double src);	Constructs a String object from a double data type	double d = 24.53; String s(d);

Operators

Table B shows the String operators and gives an example of each. These operators let you perform operations on strings using common-sense syntax rather than relying on cryptic function names.

Table B: *String class operators*

Operator	Description	Example
=	Assignment	s = "Test";
+	Concatentation	s = path + filename;
==	Equality	if (s == "Hello World")…
!=	Inequality	if (s != "Logon"). . .
<	Less than	if (s < "a")…
>	Greater than	if (s > "z")…
<=	Less than or =	if (s <= "1")…
>=	Greater than or =	if (s > = "10")…
[ ]	Index	
Label1->Caption = s[10];		

Primary functions

The String class's many functions let you manipulate strings just about any way you choose. We'll describe the functions you'll probably use most often with the String class.

AnsiCompare()

The AnsiCompare() function compares one string to another, using the following syntax:

```
int AnsiCompare(const AnsiString& rhs);
```

The return value will be -1 if the string is less than rhs, 1 if the string is greater than rhs, and 0 if the string's contents equal rhs. You don't use this function to test for equality, but rather to sort strings. For example,

```
String s1 = "Adam";
int result = s1.AnsiCompare("Bob");
```

In this case, result will be -1 because A is less than B. You can also use the < and > operators to sort strings.

```
AnsiCompareIC()
```

The AnsiCompareIC() function compares one string to another without regard to case sensitivity, using the following syntax:

```
int AnsiCompareIC(const AnsiString& rhs);
```

AnsiLastChar()

The AnsiLastChar() function uses the syntax

```
char* AnsiLastChar();
```

to return the last character in a string. For example, the code

```
String s = "This is a test";
Label1->Text = s.AnsiLastChar();
```

displays t in the Label component.

c_str()

The c_str() function returns a pointer to the character array used to hold the string's text. It uses this syntax:

```
char* __fastcall c_str();
```

You must use this function any time you need a char pointer to the string's text.

For example, the Windows API function DrawText() requires a pointer to a character buffer. In such a case, you'll need to use the c_str() function as follows:

```
DrawText(Canvas->Handle,
s.c_str(), s.Length(), &rect, DT_CALCRECT);
```

The c_str() function is also handy if you need to use a String object with the C-style string manipulation functions, such as strcpy().

Delete()

The Delete() function deletes count characters from the string beginning at index. The function's syntax is as follows:

```
void Delete(int index, int count);
```

In this example,

```
String s = "This is a test";
s.Delete(1, 5);
Label1->Caption = s;
```

the label will display is a test, since the code deletes the first five characters of the original string.

Remember that the string indexing is 1-based, not 0-based.

Insert()

The Insert() function uses the syntax

```
void Insert(const AnsiString& str, int index);
```

to insert the string str starting at position index. For example, if you use the code

```
String s = "data.txt";
s.Insert("c:\\myprog\\", 1);
```

s will contain the text c:\myprog\data.txt after the call to Insert().

IsEmpty()

The IsEmpty() function has this syntax:

```
bool IsEmpty();
```

The function returns true if the string is empty and false if the string contains text.

IsPathDelimiter()

The IsPathDelimiter() function returns true if the character at the specified index is a backslash and false if it isn't. The function's syntax is as follows:

```
bool IsPathDelimiter(int index);
```

IsPathDelimiter() could be useful for traversing a string containing a path and filename.

LastDelimiter()

The LastDelimiter() function uses the following syntax:

```
int LastDelimiter(const AnsiString& delimiters);
```

You can use this function to find the last character specified in delimiters.

For instance, to extract just the path from a string containing a path and filename, you could use these lines:

```
String s = "c:\\myprog\\data.txt";
```

```
int pos = s.LastDelimiter("\\");
s.Delete(1, pos);
```

This code finds the last backslash in the string and then deletes everything from the beginning of the string to that position.

Length()

The Length() function returns the number of characters in the string. Its syntax is:

```
int Length();
```

The returned value will be the length of the text in the string, unless you've modified the length using the SetLength() function. The returned value doesn't include a terminating null character.

For example, you could use these lines:

```
String s = "Hello World";
int length = s.Length();
```

In this case, length will contain the value 11, since there are 11 actual characters in the text string.

LowerCase()

The LowerCase() function uses the syntax

```
AnsiString LowerCase();
```

This function converts the string to lowercase and returns the converted string. (However, it doesn't modify the text in the string object.) LowerCase() is useful for converting a string to lowercase (such as strings containing filenames) or for comparing two strings regardless of case.

For example, in the line

```
if (s1.LowerCase() == s2.LowerCase()) DoSomething();
```

the code doesn't modify the text in the strings--it simply compares the strings' lowercase representations for equality. If you actually want to convert a string to lowercase you'll need to use code like the following:

```
String s = "TEST";
s = s.LowerCase();
```

Pos()

The Pos() function uses the following syntax:

```
int Pos(const AnsiString& subStr);
```

It returns the position of the string passed in subStr. If the string can't be found, then Pos() returns 0. The AnsiPos() functions perform exactly the same way.

The following code finds the position of a substring within the target string:

```
String s = "This is a test";
int p = s.Pos("test");
Label1->Caption = "Pos: " + String(p);
```

This code will display the text Pos: 11 in the label component. Again, remember that the string index is 1-based, not 0-based.

SetLength()

You can use the SetLength() function to set the length of the buffer holding the string object's text. Its syntax is as follows:

```
void SetLength(int newLength);
```

Normally, the String class dynamically allocates space for the text in the string as needed, so there's rarely any reason to set the length yourself. For example:

```
String s; // length is 0
s = "test"; // length is 4
s.SetLength(50); // length is 50
s = "new string"; // length is 10
```

If you set the length to a value less than the current length of the text in the string, then the string will be truncated.

StringOfChar()

The StringOfChar() function has the following syntax:

```
AnsiString StringOfChar(char ch, int count);
```

This function creates a string filled with the character ch, with a length of count. For example, the following code fills a string with 20 spaces:

String s; s = s.StringOfChar(' ', 20); SubString()

The SubString() function uses this syntax:

```
AnsiString SubString(int index, int count);
```

It returns a substring of the string object starting at index and having a length of count. For example, if you execute the code

```
String s = "My name is BillyBob";
String sub = s.SubString(12, 8);
```

then the value of sub will be BillyBob.

ToDouble()

The ToDouble() function converts a string to a double and returns the result. The syntax is simply

```
double ToDouble();
```

If the string can't be converted to a double, then VCL throws an EConvertError exception. Here's an example:

```
String s = "123.456";
double d = s.ToDouble();
d *= .3;
```

In order for the string to be converted, it must contain a valid floating-point number and no alpha characters.

ToInt()

The ToInt() function converts a string to an integer value and returns the result. The syntax is as follows:

```
int ToInt();
```

For example,

```
String s = "123";
int i = s.ToInt();
```

If the string can't be converted, then VCL throws an EConvertError exception.

ToIntDef()

The ToIntDef() function converts a string to an integer value and returns the result. Its syntax is

```
int ToIntDef(int defaultValue);
```

If the string can't be converted, then the function returns the value supplied in defaultValue; VCL doesn't throw an exception.

Trim(), TrimLeft(), TrimRight()

The three Trim functions trim blank spaces from a string object and return the result, using the following syntaxes:

```
AnsiString Trim();
AnsiString TrimLeft();
AnsiString TrimRight();
```

The Trim() function removes both leading and trailing spaces, TrimLeft() removes leading spaces, and TrimRight() removes only trailing spaces. These functions don't modify the text in the string object.

In this example,

```
String editText = Edit1->Text;
editText = editText.Trim();
```

Properties such as Caption, Title, and Text are String properties. As a result, you can condense the example code to a single line:

```
String editText = Edit1->Text.Trim();
```

UpperCase()

The UpperCase() function converts a string to upper- case and returns the result. In the syntax

```
AnsiString UpperCase();
```

the string's text isn't modified as in the example

```
s = s.UpperCase();
```

Other functions

Now, let's take a look at some useful utility functions. You may use these functions from time to time in your work with the String class.

CurrToStr()

The CurrToStr() function converts a currency value to a string and returns the string. The function uses the syntax

```
AnsiString CurrToStr(Currency value);
```

which doesn't change the string object for which you call it, but rather formats and returns a string. For example, the lines

```
String s;
Currency c(19.95);
c *= 1.06; // add sales tax
Label1->Caption = s.CurrToStr(c);
```

won't modify the String object s.

CurrToStrF()

The CurrToStrF() function uses this syntax:

```
AnsiString CurrToStrF(Currency value, TStringFloatFormat format, int digits);
```

It converts a currency value to a string, applies formatting, and returns the string. The lines

```
String s;
Currency c(19.95);
c *= 1.06; // add sales tax
Label1->Caption = s.CurrToStrF(c, AnsiString::sffCurrency, 2);
```

display the text $21.15 in the label. The text is formatted for currency, including the dollar sign and two decimal places.

FloatToStrF()

The FloatToStrF() function formats and returns a floating point number, using this syntax:

```
AnsiString FloatToStrF(
long double value, TStringFloatFormat format,
int precision, int digits);
```

The resulting string is based on the format, precision, and digits parameters. For example, the lines

```
String s;
```

```
float f = 2219.3446237795;
Label1->Caption = s.FloatToStrF(f,
AnsiString::sffExponent, 8, 0);
```

format the given floating point value in scientific notation using eight-digit precision.

Format()

The Format() function builds a string using a format string and the supplied arguments. The syntax is as follows:

AnsiString Format(const AnsiString& format,const TVarRec *args, int size);

This function works in a way similar to the sprintf() and wsprintf() functions in C++. For example, the lines

```
String s = Format("%s%d, %d",
OPENARRAY(TVarRec, ("Values: ", 10, 20)));
Label1->Caption = s;
```

display the string Values: 10, 20 in the label. Notice that the OPENARRAY macro passes the values to the Format() function. This process is necessary because C++ doesn't have open arrays--however, AnsiString::Format() calls the VCL version of Format(), which takes an open array as a parameter.

The Format() function is one of the few cases in which Pascal and C++ just don't work well together. You can use one of two alternative methods for building the string from the previous example:

```
// example 1
char buff[20];
sprintf(buff, "Value: %d, %d", 10, 20);
String s = buff;
```

```
// example 2
String s = "Value: " + String(10) + ", " + String(20);
```

In my humble opinion, these methods are easier to use than the Format() function.

FmtLoadStr()

The FmtLoadStr() function is a combination of the Format() and LoadStr() functions. The syntax

```
AnsiString FmtLoadStr(int ident, const TVarRec *args, int size);
```

loads the string resource ident and formats it according to the value of the args parameter. See the descriptions of Format() and LoadStr() for more information.

IsDelimiter()

The IsDelimiter() function uses this syntax:

```
bool IsDelimiter(const AnsiString& delimiters, int index);
```

The function returns true if the character at index matches one of the characters in the string delimiters. Here's an example:

```
String s = "c:\\myprog\\data.txt";
if (s.IsDelimiter("\\", 3)) DoSomething();
```

The delimiters parameter can be a string of several delimiters you want to test against. For instance, if you wanted to check to see whether the character at position 10 was a space, a minus sign, a plus sign, a comma, or a dollar sign, you'd use this code:

```
String s = "This is a string";
bool test = s.IsDelimiter(" -+,$", 10);
LoadStr()
```

The LoadStr() function loads a string resource using the following syntax:

```
AnsiString LoadStr(int ident);
```

Of course, you must have a string table resource bound to your executable program, as does the following example:

```
String s;
s = s.LoadStr(100); //load string with ID of 100
```

Poor class design?

Note that the CurrToStr(), CurrToStrF(), FloatToStrF(), Format(), FormatFloat(), FrmtLoadString(), and LoadStr() functions really don't belong in the AnsiString class at all. They don't operate on the string object for which you call them, nor do they use the string object in any way. In addition, these functions are provided as standalone functions in the VCL Sysutils unit. It's always easier to use the standalone version of these functions rather than the AnsiString version, as shown in these two examples:

```
// method 1
String s;
s = s.LoadStr(ID_STRING1);

// method 2
String s = LoadStr(ID_STRING1);
```

I suspect that Borland put these functions into the AnsiString class at some early point in C++Builder's development, then forgot them later, when they were no longer necessary. Regardless, they're part of the AnsiString class and they're useful functions, so we've listed them here.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

August 1997

All set?

by Kent Reisdorph

As you probably know by now, the VCL that comes with C++Builder is written in Pascal. While this isn't a problem, the situation does raise certain issues for the C++Builder programmer--some features found in Pascal don't exist in C++, and vice versa. One Pascal feature that C++ doesn't have is the set. In this article, we'll explore sets and how C++Builder handles them. Let's begin with a little background information.

Sets in Pascal

In Pascal, a set is "a collection of like objects." Hmm... that definition doesn't say much, does it? An example that comes to mind is VCL's Font.Style property. This property can have one or more of the following values: fsBold, fsItalic, fsUnderline, or fsStrikeout. Let's detour for a moment and explain where those values come from.

Typically, the individual elements you use with a set come from an enumeration. For instance, the font styles we just mentioned come from the TFontStyle enumeration. The TFontStyles set, then, is created as a set of TFontStyle values. The VCL source declaring these objects is as follows:

```
TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
TFontStyles = set of TFontStyle;
```

Most of the sets you'll use in VCL follow this same architecture--you have an enumeration that declares a set, which can contain values of that enumeration.

Typically, you'll turn the font's individual Style values on or off at design time. Sometimes, however, you need to set the font Style property at runtime. In that case, you'd use code like the following:

```
Font.Style := [];
Font.Style := [fsBold, fsItalic];
```

The first line clears the Style set by assigning an empty set (designated by empty brackets) to the Style property. The next line adds the fsBold and fsItalic values to the set. When you use the font, your text will appear in bold italics. (Note that the first line of code in this example isn't really necessary. I included it to show you how to clear a set.)

Now, let's say that you want the font to be bold and underlined, but not italic. You need to remove the italic style and add the underline style, while leaving the bold style in the set:

```
Font.Style := Font.Style - [fsItalic];
Font.Style := Font.Style + [fsUnderline];
```

And that, my friends, is how you add elements to--and remove them from--a set in Pascal.

Besides changing set elements, you may want to see if a particular item is in a set. Let's say you want to know whether the font is currently bold. You can determine if the fsBold style is in the set as follows:

```
if fsBold in Font.Style then
Font.Style := Font.Style - [fsBold];
```

The in keyword returns true if the particular value is in the set and false if it isn't in the set.

Sets in C++

The C++ language includes no built-in support for sets. C++ programmers have typically used bitfields where Pascal programmers use sets. For example, the MessageBox() function has a flags parameter that specifies which buttons and icons will appear on the message box. The following example illustrates the use and manipulation of a bitfield:

```
// the initial bitfield with two styles
unsigned int flags = MB_OK | MB_ICONEXCLAMATION;
// remove the icon style
flags &= ~MB_ICONEXCLAMATION;
// add a help button to the message box
flags |= MB_HELP;
// show the message box
::MessageBox(Handle, "Test Message",
"Message", flags);
```

This mechanism works fine, but it isn't very useful when you're dealing with VCL, because VCL uses Pascal sets. For this reason, Borland had to come up with a way to emulate sets in C++Builder. As an added benefit, the concept of sets is easier for beginning C++ programmers to understand than is the concept of bitfields. (C++Builder is all about rapid application development, and part of RAD is getting programmers up to speed as quickly as possible.)

The Set template

While there are several ways to solve this particular riddle in C++, Borland decided to implement sets in the form of a template class called, predictably, Set. Fortunately, you don't need to know the nitty-gritty details of templates to use the Set template.

Let's return to our earlier examples of how to add and remove elements from a set in Pascal:

```
{start with an empty set}
Font.Style := [];
{add the bold and italic styles}
Font.Style := [fsBold, fsItalic];
{remove the italic style}
Font.Style := Font.Style - [fsItalic];
{add the underline style}
Font.Style := Font.Style + [fsUnderline];
```

Now, let's see how the corresponding code looks in C++Builder.

```
// start with an empty set
Font-$gtStyle.Clear();
// add the bold and italic styles
Font-$gtStyle << fsBold << fsItalic;
// remove the italic style
Font-$gtStyle $gt$gt fsItalic;
// add the underline style
Font-$gtStyle << fsUnderline;
```

First, notice that you can use the Clear() function--a member of the Set template--to remove all elements from the set. Next, notice how you can add elements to the set using the insertion operator (<<). If you have C++ experience, then you've seen similar syntax in the C++ streaming classes. The insertion operator adds the item to the right of the operator to the set; you can chain insertion operators to add several elements to a set at one time.

Finally, the extraction operator (gtgt) removes elements from the set. You can also chain extraction operators to remove several elements from the set simultaneously.

Who's in there?

In the section on Pascal sets, we talked about checking a set to see which elements it contains. We illustrated the technique with this code:

```
if fsBold in Font.Style then
```

```
Font.Style := Font.Style - [fsBold];
```

The code checks to see if the bold style is set and, if so, removes it. However, C++ doesn't have the in keyword. Instead, the C++ Set template has a function called Contains(), which serves the same purpose. The C++ equivalent of the previous code is as follows:

```
if (Font-$gtStyle.Contains(fsBold))
Font-$gtStyle $gt$gt $gt$gt;
```

Sets on the fly

Sometimes you need to create a set on the fly. The VCL MessageDlg() function provides a good example. (Note that this function isn't documented in the C++Builder help files.) This function expects a set of TMsgButtons as its third parameter; the set tells VCL which buttons to put in the dialog box.

Once again, let's look first at how to create the set in Pascal:

```
MessageDlg("An error has occurred...",
mtError, [mbOK, mbCancel], 0);
```

Now, look at the C++ code for the same set:

```
MessageDlg("An error has occurred...",
$lt$lt mbOK $lt$lt mbCancel, 0);
```

In this case, the Pascal code is a bit more elegant than the C++ implementation. You create the Pascal temporary set simply by using the square brackets with the elements you wish to include in the set. In C++, you create a temporary set, then add the elements using the insertion operator. That part of the C++ code is as follows:

```
TMsgDlgButtons() << mbOK << <&lte$gt$gt $gt$gtn
```

use this simple syntax any time you need a temporary set.

Conclusion

Sets are a way of life in VCL and in C++Builder programming. The C++Builder Set template is your route to VCL sets. The extraction and insertion operators, while odd-looking at first, provide a simple method for you to add and remove set elements.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1997

Strings, strings, strings!

by Kent Reisdorph

Back in the good old days of C++ programming, you had two choices when it came to strings and string manipulation: You could use the C-style character array or the C++ string class. The situation is more complicated in C++Builder, because there are at least five different string-handling mechanisms:

- The C-style character array
- The C++ cstring class
- The Standard Template Library (STL) basic_string class
- The VCL AnsiString class
- The VCL SmallString template

This whole string thing can be confusing, whether you come from a C++ or Delphi background or you're a newcomer to the sport. This article will help clear up the string issue. We'll take a look at each of the string-handling mechanisms and discuss the situations in which you might use each one.

In the beginning...

In the beginning, of course, there was C. If you didn't already know it, C++ is based on C--everything that's in C is also in C++. In addition, C++ contains the goodies you've come to know and love: classes, overloaded functions, operator overloading, templates, and so on.

In C, there's no such thing as a true string data type--C handles strings as an array of characters (an array of the char data type). A character array declaration looks like this:

```
char buffer[50];
```

This statement creates a character buffer with a length of 50.

Character arrays are *null-terminated*--which means the end of the string is marked by the terminating null character. The terminating null character is 0, or in C parlance, \0. For example, let's say you store the word *John* in a character array using the

```
strcpy()
```

function, as follows:

```
char buffer[8];
strcpy(buffer,"John");
```

In memory, the character array looks like this:

```
J O H N \0
```

The terminating null character tells the compiler, "This is the end of the string." Since the character array is eight characters long, any characters following the terminating null will contain random values--garbage, if you will. These random characters don't matter, though, because as far as the compiler is concerned, the string ends at the terminating null.

This issue becomes important when you consider the declared length of a string. In the previous example, we declared a character array with an eight-character length. However, you can store only seven characters in the array, because C++ will add the terminating null character to any string you store.

So, if you try to store the eight-character string *BillyBob* in the array, you'll overrun the array by one character when C++ adds the terminating null. Your code will cause an access violation, either immediately or at some future point in the program.

The interesting factor is that C++ won't complain if you try to store 100 characters in an array that you declared to hold only 20 characters. It's up to you to keep track of how large your array is and to ensure that you don't overrun the end of the array.

You manipulate character arrays using a host of C++ functions. **Table A** lists a few of the most common string manipulation functions.

Table A: String manipulation functions

Function	**Description**
sprintf()	Builds a string with formatting
strcpy()	Copies one string to another
strcmp()	Compares one string to another
strcmpi()	Performs a case-insensitive comparison
strcat()	Concatenates one string to another
strlen()	Returns the length of the string up to, and excluding, the terminating null

strstr()	Searches a string for an occurrence of another string

While the character array is as old as dust, we shouldn't discard it as an antiquated tool. When you want fast string manipulation, character arrays are the way to go. In some cases, in fact, you have no choice but to use character arrays. For instance, many Windows API functions require pointers to character arrays as parameters. So, the character array won't disappear any time soon.

C++ string classes

Over the years, the C++ language has includedvarious flavors of string classes. Each compiler vendor had its own string class, and those classes naturally evolved. Borland programmers need to make special note of two string classes: cstring and basic_string. Let's take a look at these classes.

The C++ string class, cstring

The C++ string class is the original string class. The name of the class is actually string (lowercase s). This class is declared in the CSTRING.H header file, and is sometimes called cstring. We'll refer to it as cstring, since we'll discuss several string class variations in this article. Although the cstring class has evolved over time, the current version has remained unchanged for several years.

Why does C++ have a string class? Primarily because the C-style character arrays aren't very convenient when you need to manipulate many strings. Classes let you more easily truncate strings, concatenate strings, delete portions of a string, search a string for certain character combinations, and so on.

For example, the following code demonstrates two ways to add four strings together, using the string class and then again using a C-style character array:

```
string s = "Now is the time " +
  "for all good men " +
  "to come to the aid " +
  "of their country."

char buff[80]; // be sure it's big enough

strcpy(buff, "Now is the time");
strcat(buff, "for all good men");
strcat(buff, "to come to the aid");
```

```
strcat(buff, "of their country");
```

This code illustrates two important points about string classes. First, notice that you don't have to worry about specifying the size of the string. The cstring class will allocate memory as needed to accommodate an operation such as concatenation. And, speaking of concatenation, the string class overrides the + operator and lets you use + to add to the end of the string. With character arrays, you must use the strcat() function to add strings.

Testing for equality is another operation that's much easier with a string class than with character arrays. For instance, consider the following lines of code:

```
if (!strcmp(buff, "test")) // strings match
if (s == "test") // strings match
```

The cstring version is easier to read and more intuitive than the character array version, particularly because the strcmp() function returns 0 if the strings match and a non-zero value if the strings don't match. The first line above seems to read "if *not* buff, compare to *test*," when in fact, the line compares for equality. Again, the string class version is more readable and understandable.

Another big plus about string classes is that you can call cstring class functions to perform operations on the string. **Table B** describes several cstring functions. This list isn't complete, but it provides an idea of the kinds of operations you can perform on a string object.

Table B: *cstring class functions*

Name	**Description**
append()	Adds text to the end of a string
contains()	Determines whether a string contains another string
c_str()	Returns the character array buffer used by the string class to store a string's data
find()	Finds characters within the string class and returns a string's position
find_first_of()	Returns an index to the first occurrence of specified characters within a string
find_last_of()	Returns an index to the last occurrence of specified characters within a string
insert()	Inserts text into a string at the specified location
length()	Returns the length of the text in a string

prepend()	Adds text to the beginning of a string
remove()	Removes characters from a string
strip()	Removes trailing or leading characters, such as trailing blank, from a string
substring()	Creates a new string from characters in a string
to_lower()	Converts a string to lowercase
to_upper()	Converts a string to uppercase

For example, you may need to strip the path and extension off a filename, using code like the following:

```
string s = "c:\\myprog\\readme.txt";

// find last backslash
int pos = s.find_last_of("\\");

// remove all characters from beginning of
// string to character following backslash
s.remove(0, pos + 1);

// chop extension off the end of string
s.remove(s.length() - 4, 4);
s.prepend("MyProgram - ");

// string now contains 'MyProgram - readme'
SetWindowText(Handle, s.c_str());
```

This code illustrates three points. First, you can easily manipulate the text in a cstring class object (the code might not seem easy, until you contrast it to the equivalent code using char arrays). Second, to specify the backslash in a C++ string literal, you must use a double backslash. Since a single backslash is an escape character used to enter special codes, to indicate an actual backslash in the string you must use a double backslash.

Finally, this code illustrates the c_str() function, which yields a char* representation of your string. In this case, the Windows API function SetWindowText() wouldn't understand if you tried to pass it the cstring object itself. The c_str() function isn't pretty, and it's a pain to type, but you'll need it if you're going to use string classes with any functions requiring a char*.

One problem with string classes is that, depending on how they're written, they can be fairly slow when performing much string manipulation. For instance, each time the string length changes, memory must be reallocated to account for the new string length. Depending on

how thestring class was originally written, this reallocation can cost you manyclock cycles.

STL basic_string

The Standard Template Library (STL) was written in an attempt to standardize often-used C++ classes such as string classes, array classes, lists, double lists, queues, and others. STL also provides iterator classes that traverse the various container classes. The basic_string class is part of STL. This class is typedef'd, so we refer to it simply as string. You may encounter problems since cstring and basic_string ultimately have the same name, as we'll discuss in a moment.

The STL string class has many functions in common with cstring, and the class also provides most of the same overloaded operators as cstring. Although STL is considered the new wave in terms of general C++ programming classes, I don't anticipate using STL strings extensively in C++Builder. For one thing, you can't place the STL headers in the C++Builder pre-compiled headers. Compiling a unit that contains the lines

```
#include <vcl\vcl.h>
#include <string>
#pragma hdrstop
```

will generate the compiler warning *Could not create pre-compiled header: code in header*. In order to eliminate the compiler warning, you must write the code this way:

```
#include <vcl\vcl.h>
#pragma hdrstop
#include <string>
```

The bottom line is that a program using STL's basic_string class will take longer to compile because the STL header has to compile each and every time the unit compiles.

Another problem comes into play when you use cstring and basic_string, because both use the common name string. When you use the STL version of string, you must declare the std namespace. To do so, you can place the following declaration at the beginning of the unit that uses STL string:

```
using namespace std;
```

This line tells the compiler, "Use the string class found in the std namespace." You can also declare the std namespace by explicitly declaring the string class with the std namespace specified, as follows:

```
std::string s = "This is an STL string.";
```

It doesn't matter which method you use. This requirement, along with the header issue, makes me avoid using the STL string class all together.

VCL string classes

As you know by now, VCL is written in Pascal. Pascal offers some language features C++ doesn't have, and vice versa. For instance, Pascal has three string data types: PChar, short string, and large string. The PChar data type is equivalent to char*. The short string data type is a string--limited to 255 characters--whose length is specified when you declare it. The large string data type is a string object whose size is limited only by available memory. C++Builder provides two classes to emulate Pascal's string data types: SmallString and AnsiString (there's no need for a class to emulate PChar). Let's examine these classes.

The SmallString template

C++Builder implements the SmallString class as a template. Borland provides this class for Pascal compatibility, and I don't see any reason to use it when you're writing new C++Builder code. If you were to use this class, the syntax would look something like this:

```
SmallString<30> s;
s = "Test";
```

The String class

The VCL String class is yet another C++ string class. Before we get into the specifics of this class, let's consider a little background.

VCL makes heavy use of the string data type (long string). Nearly all text-based VCL properties are of the Pascal string data type. For example, the Text, Name, and Caption properties are string properties. VCL also uses the string data type in various methods and event-handling functions.

You should understand two things about this data type. First, it's an actual language data type, not just a character array. Second, C++ has no built-in equivalent for the Pascal string data type. Since string is used so heavily in VCL, and since C++Builder uses the Pascal VCL, Borland created a C++ class called AnsiString to approximate the Pascal string data type. You can use this C++ class wherever you require a Pascal string data type.

Let's face it--the name AnsiString isn't particularly appealing. Somewhere in SYSDEFS.H, you'll

find the following line:

```
typedef AnsiString String;
```

This line lets you use the name String (uppercase *S*) to declare an instance of the AnsiString class. To illustrate, you can write a line like this:

```
String s = "This is a  test";
```

Since String is the recommended alias for the AnsiString class, there's no reason to use the name AnsiString in your C++Builder programs.

The String class, like the other C++ classes, has several functions to make string manipulation easier. The class's constructors allow you to create a String object from a char*, an int, or a double. In addition, the String class has several overridden operators to ease tasks like concatenation (+ and +=), assignment (=), and testing for equality (==). Finally, the conversion operators make mixing and matching String and other object types invisible to you.

Consider the following code, for example:

```
char* buff = "Test";
String s = "Test";
if (s == buff)
DoSomething();
```

This code works because the String class has a char* conversion operator (as well as the overridden == operator). The char* conversion operator performs an implicit conversion in this case, which allows you to test the contents of the String object and the contents of the character array for equality. The whole process happens automatically--you don't have to worry about *how* it works, just understand that it *does* work.

The String class, like cstring, has a c_str() function, which is required when you want to get the character buffer of the String object. For example, if you use the Windows API function DrawText(), you have to write code something like this:

```
String s = "This is a test";
DrawText(Canvas->Handle, s.c_str(), s.Length(), &rect, DT_SINGLELINE);
```

Since the second parameter of the DrawText() function requires a pointer to a character buffer, you must use the c_str() function.

At this point, we need to mention an oddity of the String class: The index operator ([ ]) can reference a particular element of a string. For example, the lines

```
String s = "Hello World!";
Label->Caption = s[7];
```

assign the character *W* to the Caption property of a label component. Note that the first element of the string is at array index 1--not array index 0, as with other C++ arrays.

While the 1-based index is required for technical reasons, I suspect this feature will cause C++Builder programmers some grief. For instance, the following code will fail silently:

```
String s = "c:\\myprog\\myprog.exe";
int index = s.LastDelimiter("\\");
s.Delete(0, index);
```

The code will fail because 0 isn't a valid index number for a string. The following line of code

```
s.Delete(1, index);
```

on the other hand, is correct.

So many choices!

You have five choices of string types in C++Builder (four, if you exclude SmallString). Which of the five types are you going to use?

I've found that the String class works well for almost all of my string needs. (This endorsement is based primarily on the fact that the String class was designed to be used with VCL properties and methods.) For day-to-day programming using VCL, the String class is a clear choice. It works seamlessly with VCL and includes enough functions to handle most string-manipulation chores. But, as usual, the choice of string types depends on the task at hand in your code. It's always best, though, to know your options. In this article, we've attempted to shed some light on the subject of handling strings in C++Builder.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1997

Using forms in console applications

by Kent Reisdorph

It's no secret that C++Builder is an incredible programming tool. C++Builder really shines when it comes to designing the user interface for an application. In addition, C++Builder's visual programming capability lets you prototype your application in the shortest possible amount of time, as compared to other C++ programming environments.

But C++Builder is more than just another pretty face--you can use it to write console applications and Dynamic Link Libraries (DLLs), too. Even though a console application is character-based (non-graphical), you can still call common dialog boxes or VCL forms from the application if you need to. In this article, we'll explain how. (If you need a refresher on console applications, see "What's a Console Application?")

Getting started

The first thing you need to do is to create the console application. Fortunately, setting up the shell of a console application doesn't require a lot of work--C++Builder will do it for you. Begin by choosing File | New... from the C++Builder main menu. The New Items dialog box will open, showing you the objects in the Object Repository. Double-click on the Console App icon, and C++Builder will create the shell of a console application and display it in the Code Editor.

Okay, that was easy enough. Now you need to create a form that you can display from the console app. To do so, click the New Form button on the C++Builder ToolBar, or choose File | New Form from the main menu. Obviously, we haven't yet written any code to display the form, but you might be surprised to learn that the application won't even compile right now. The console application has no knowledge of VCL at this point, so you'll have to educate it. To better understand this process, let's talk a little about how C++Builder operates when you're building a GUI application.

When you add a form to a Windows GUI application, C++Builder places the following line in the project source file:

```
USEFORM("Unit1.cpp", Form1);
```

USEFORM is a VCL macro--it alerts VCL that you're using a form and provides the name of the form's source unit. This process allows C++Builder to auto-create forms and to set up some of the other housekeeping chores that VCL does in the background on your behalf. It isn't black magic, but it is VCL's way of hiding details that you don't need to worry about (after all, that's part of the beauty of RAD development).

When you add a form to a console application, C++Builder adds the same line to the application's source code. The problem is that your console application--since it isn't a VCL application--doesn't know what the USEFORM macro is. So, you need to include the VCL.H header. To do that, add the following line below the other includes and just above the #pragma hdrstop line:

```
#include <vcl\vcl.h>
```

Now the console application will compile, because the compiler can see the USEFORM macro declaration as well as the rest of the VCL declarations.

Next, you need to be sure that the console application can find the declaration for the form's class. When you

add a new form to a GUI application, you make the form available to any units that reference the form by choosing File | Include Unit Hdr... from the C++Builder main menu. For some inexplicable reason, this menu item is unavailable when you're building a console application. As a result, at the time you add a form to a console application, you must manually enter the code to include the form's header. To do so, add a line like the following to the project's main unit, this time below the #pragma hdrstop line:

```
#include "Unit1.h";
```

This line ensures that the compiler will be able to find the class declaration when you reference the form in your console application's code.

Showing the form

Now that you've set up the console application to handle a VCL form, the actual code to show the form is pretty basic. Once again, we'll refer to the way C++Builder handles this process in a GUI application.

In a GUI application, C++Builder auto-creates forms unless you specify that it shouldn't. If a form is auto-created, then you can show it without any fuss, as follows:

```
Form1->ShowModal();
```

C++Builder creates a pointer to every form it adds to a project via the New Form menu item. The pointer's variable name is the same as the form's Name property. In the previous example, the pointer name is Form1, and the pointer is type TForm1. The parent object automatically deletes the pointer when the parent itself is deleted. The process is essentially automatic in a C++Builder GUI application--you don't have to spend any time thinking about it.

But, in real-world applications, not all forms are auto-created. If a form isn't auto-created, you need to specifically create the form before you can show it. In that case, the code looks like the following:

```
Form1 = new
TForm1(this);
Form1->ShowModal();
```

A console applicationdoesn't allow auto-creation of forms, so you must explicitly create the form object before you try to use the form's pointer. In addition, you must explicitly delete the pointer when you're done with it. In this case, we recommend that you use a local variable instead of the global variable created by C++Builder, as follows:

```
TForm1* form = new TForm1(0);
form->ShowModal();
delete form;
```

While you're not required to use a local variable, you'll probably find that doing so better documents what's happening in your code. The previous code segment creates the TForm1 object, shows the form, and then destroys the TForm1 object when the form is dismissed. In this situation, you take the responsibility for creating and deleting the object.

Notice that this code specifies 0 as the form's parent, rather than this. We'll discuss the reason next.

Mom, is that you?

In the normal VCL scheme of things, a form is usually parented to another form or--in the case of auto-created forms--to the VCL Application object. The TForm constructor takes a TComponent pointer as its single parameter, which specifies the owner of the form.

Typically, when you create a form at runtime in a GUI application, you'll use code like the following:

```
TForm1* form = new TForm1(this);
```

In a GUI application, the this pointer is some descendant of TComponent and everything works fine. In a console mode application this (no pun intended) approach won't work because the this pointer is only valid within a class. Or, to put it another way, there is no this pointer within the main() function of a console application.

So what should you do? You have two basic choices. One choice is to supply no owner for the form. In other words, you can just pass a 0 in a form's constructor, as follows:

```
TForm1* form = new TForm(0);
```

If you do so, you have the responsibility of deleting the form at some point before the application terminates. If you fail to delete the pointer, then the application will leak memory.

Your second choice is to supply the Application object as the form's parent. (Even though you've written a console application, VCL still creates an Application object--it does so automatically whenever you use VCL objects in a console application.) If you choose this route, you'll write code something like this:

```
TForm1* form = new TForm(Application);
```

The advantage here is that you don't have to delete the form specifically, because the Application object is the owner of the form--it will delete the form when the application closes. It doesn't really matter which method you use, but we prefer to use 0 as the owner and take the responsibility of deleting the form as soon as we're done with it.

More parenting headaches

The issue of ownership and parenting brings up another point you must deal with when you display a form from a console application. In order to explain, we must first describe what happens in a GUI application when you display a form using the ShowModal() method.

As you probably know, once you display a modal window, the user must close the window before he or she can do anything more with the application. In effect, the parent window is disabled while the modal window is being displayed.

In the wonderful world of the Windows API, displaying a modal dialog box is fairly straightforward because Windows takes care of disabling the parent window for you. You don't have to do anything but display the modal dialog box (although it takes about 50 lines of code to get to that point!). To the VCL programmer, displaying a form modally is even easier--all you need to do is call the ShowModal() method.

However, what goes on behind the scenes in a VCL application is a little more complicated. Because a form isn't a true dialog box, VCL must take responsibility for disabling the parent window and then re-enabling the parent window every time ShowModal() is called. While this process isn't exactly rocket science, it's one of the little things VCL does for you that you probably take for granted.

The point is that when you call a form from a console application, you can't parent the form to the console window, and no automatic enabling and disabling of the parent window takes place. Therefore, it's up to you to disable the console window prior to calling ShowModal(), and it's also up to you to re-enable the console window after ShowModal() returns.

First, you must determine the window handle of the console window. (For more information on finding the window handle for a console application, see "Finding a Console Window.") Once you know the window handle, the code to simulate a modal dialog box in a console application looks like the following:

```
EnableWindow(hWnd, false);
TForm1* form = new TForm1(0);
form->ShowModal();
delete form;
EnableWindow(hWnd, true);
```

This code will allow the console application to behave somewhat like a GUI application. That is, it will be disabled as long as the form is being displayed.

Note that there's at least one problem associated with this technique: Users won't be able to use [Alt][Tab] to tab to the console application while it's disabled. Well, nothing's perfect.

Another issue you'll contend with when simulating a modal dialog box is window focus. Since the form technically doesn't belong to the console window, the console window won't automatically have focus when the form closes. A call to the API function BringWindowToTop() after the form closes will remedy that situation.

Weird stuff, man

When you run a console application that contains a form, you'll notice a VCL quirk. Specifically, two icons for your application will appear on the Windows taskbar: The first icon is the console application, and the second represents the VCL Application class. As we mentioned, the Application object is created automatically when you execute ac program that contains VCL forms or classes. You can hide the Application icon using this line:

```
ShowWindow(Application->Handle, SW_HIDE);
```

Listing A contains a program that illustrates the use of a form and the use of the Open File dialog box in a console application. When the application runs, you can choose menu selection 1 to display the Open File dialog box or menu selection 2 to display a form. To try this program, create a new console application, add a form of any flavor you like, and then enter the code.

Conclusion

Displaying a form or a common dialog box from a console application may not be something that fits into your

immediate plans. However, when the time comes, you'll know what pitfalls to avoid and how to go about implementing VCL objects in your console applications.

Listing A: *ConsForm.cpp*

```
//---------------------------------------------------
#include <vcl\condefs.h>
#include <stdio.h>
#include <stdlib.h>

// Add these includes:
#include <vcl\vcl.h>
#include <conio.h>
#include <iostream.h>
#pragma hdrstop

// Include STL string header.
#include <string>

// Include the form's header.
#include "Form.h"

// Declaration for the GetHWND() function.
HWND GetHWND();

//---------------------------------------------------
USERES("ConsForm.res");
USEFORM("Form.cpp", Form1);
//---------------------------------------------------

int main(int argc, char **argv)
{
  // Get the window handle of the console window.
  HWND hWnd = GetHWND();
  // This is just so that the console window has
  // focus when run from the debugger.
  BringWindowToTop(hWnd);
  // A little menu loop.
  bool done = false;
  do {
    clrscr();
    cout << endl;
    cout << "Please choose one of the following.";
    cout << endl << endl;
    cout << "0. Quit" << endl;;
    cout << "1. Simulate File Open" << endl;
    cout << "2. Show Form" << endl;
    int choice;
    do {
      choice = getch();
      choice -= 48;
```

```
} while (choice < -1 || choice > 2);

// Do something based on the choice made.
switch (choice) {
// All done. Get out of loop.
  case 0 : {
    done = true;
    break;
  }

// Show the common file open dialog.
  case 1 : {
    // Disable the console window.
    EnableWindow(hWnd, false);

    // Create an instance of the TOpenDialog
    // class and set the filter.
    TOpenDialog* dlg = new TOpenDialog(0);
    dlg->Filter = "All files (*.*)|*.*";

    // Show the dialog.
    bool result = dlg->Execute();

    // Enable the console window and bring it to the top.
    EnableWindow(hWnd, true);
    BringWindowToTop(hWnd);

    // If the OK button was pressed then
    // show the name of the file chosen.
    if (result) {
      cout << endl << "The file opened was: " << dlg->FileName << endl << endl;
      cout << "Press any key to continue...";
      getch();
    }

    // Delete the dialog object.
    delete dlg;
    break;
  }
  case 2 : {
    // Disable the console window.
    EnableWindow(hWnd, false);

    // Create an instance of the TForm
    // class and show the form.
    TForm1* Form1 = new TForm1(0);
    int x = Form1->ShowModal();

    // Enable the console window and
    // bring it to the top.
    EnableWindow(hWnd, true);
    BringWindowToTop(hWnd);
```

```
        // Display the results of the modal form.
        cout << endl << "Modal result: " << x << endl;
        cout << endl <<  "Press any key to continue...";
        getch();
        delete Form1;
        break;
      }
    }
  } while(!done);
  return 0;
}

//---------------------------------------------------------------
// This routine gets the window handle of the console
// window. Win95 and NT handle console windows
// differently so we have to check and see what OS
// we're running under and take appropriate measures.
HWND GetHWND()
{
  char title[256] = "";
  char className[20] = "";

  // Get the name of this executable.
  GetModuleFileName(0, title, sizeof(title));

  // See if we're running on Win95 or NT.
  TOSVersionInfo info;
  info.dwOSVersionInfoSize = sizeof(info);
  GetVersionEx(&info);
  int platform = info.dwPlatformId;

  // If NT then the class name is 'ConsoleWindowClass'
  // and the title is the entire path and filename.
  if (platform == VER_PLATFORM_WIN32_NT)
    strcpy(className, "ConsoleWindowClass");

  // If Win95 then the class name is 'tty' and
  // the window title is just the filename
  // without path and extension.
  else {
    strcpy(className, "tty");

    // Strip off everything but the file name.
    std::string name = title;
    int pos = name.find_last_of("\\");
    name.remove(0, ++pos);
    name.remove(name.length() - 4, 4);
    strcpy(title, name.c_str());
  }

  // Return the result of FindWindow().
  return FindWindow(className, title);
```

```
}
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1997

What's a console application?

by Kent Reisdorph

Most of you probably didn't buy C++Builder to write console applications. In fact, many of you don't even know what a console application is. Let's take care of that right away: A *console application* is a 32-bit program that runs in a DOS-prompt dialog box under Windows. If you prefer, you can think of a Win32 console application as the 32-bit cousin of the DOS program.

Like an old DOS program, a console application differs from the traditional Windows application in several ways. For example, it doesn't have a main window, a menu, dialog boxes, or any of the other graphical goodies that mark a Windows GUI application. Also, console applications have no Window procedure, meaning you can write console applications with less fuss than is necessary with GUI applications.

For example,**Figure A** shows the source code for the traditional *Hello World!* console application. This program displays the text *Hello World!* in a command prompt box, then waits for the user to press a key before terminating. (Without the getch() function, Windows would open a command prompt box, display the text, and then immediately close the command prompt box.) That's the entire program.

So what's a console application good for? Such applications are convenient in specialty situations like servers, utility programs, or quick-test programs. They're particularly useful for learning C++ programming techniques without all the distractions of a GUI application.

Console applications are more widely used than you might think. Many of the command-line tools that come with C++Builder are Win32 console applications, as are many Windows utilities. You probably have quite a few console applications on your hard drive--you can think of them as the workhorses of the Windows world.

A console application generally involves a minimum amount of user interaction. Still, a console application can benefit from a little GUI help now and then. For example, let's say you have a server application that logs incoming mail messages. You could write a custom file-selection screen for your console application, but that would take a lot of work. It would be much more convenient to let the user choose the log file via the Windows Open File dialog box. In this case, you'd like to leverage some of Windows' GUI aspects in a console application.

Or, maybe you have a specialty application that crunches numbers in the background and displays its progress when requested. In this case, you might like to use a VCL form to display the requested information. The good news is, as we explain in the accompanying article, that you can use the common dialog boxes and VCL forms from a console application.

Figure A: *Here's the code for the Hello World! console application.*

```
//---------------------------------------------
#include <vcl\condefs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#pragma hdrstop

//---------------------------------------------
USERES("Project1.res");
//---------------------------------------------

int main(int argc, char **argv)
{
  printf("Hello World!");
  getch();
  return 0;
}
//---------------------------------------------
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1997

Using Delphi components in C++Builder

by Kent Reisdorph

One of C++Builder's most promising features is that it lets you use your existing Delphi components. In many cases, you can use these components as-is. Sometimes, however, a Delphi component simply won't work in C++Builder. You may get compiler errors when you add the component to the Component Palette. Or the Delphi component may install without incident, but give you compiler or linker errors when you try to build the application that uses the component.

In this article, we'll point out some problem areas you may encounter when you're installing Delphi components in C++Builder. We'll first examine the most common (and most obvious) problems and also describe some of the more subtle difficulties that aren't as easy to resolve.

How C++Builder handles Delphi components

When you install a component in C++Builder, C++Builder looks first at the component's file extension. If it's OBJ, then C++Builder simply links the object file into the component library. However, if the file extension is CPP, then C++Builder invokes the C++ compiler to produce an object file that it can link to the component library.

If the file extension is PAS, C++Builder invokes the Pascal compiler, which is slightly different from the one in Delphi. In C++Builder, the Pascal compiler will produce not only the DCU file for the component, as Delphi does, but will also produce an OBJ file and a header file with the extension HPP. The object file for the Pascal unit is necessary so C++Builder programs can link the component's code to the C++Builder executable file. The header file lets your application see the component's class declaration.

If you think about this process, you may shake your head in wonder. You have a Pascal source code unit that's compiled into an object file, which is compatible with C++ code. Not only that, C++Builder creates a header file for you so you can treat your Pascal classes as if they were C++ classes. All you have to do is add the Delphi component to the Component Palette, and C++Builder takes it from there.

Amazing! Well, it *is* amazing when it works (and it works 90 percent of the time), but there are still a few minor details to deal with. Some Pascal language features just don't translate well to C++, and those problem features can cause you headaches if you don't handle them properly.

As you're installing Delphi components in C++Builder, you may encounter problems on at least three fronts: while compiling the Pascal code when you install the component on the C++Builder Component Palette; in the headers generated by C++Builder (these problems won't show up until you try to use the component in a C++Builder application); and while linking a C++Builder application using your Delphi component. Let's examine these areas individually.

Pascal compiler issues

When you add a Delphi component to the Component Palette, C++Builder invokes the Pascal compiler to compile your component. You may encounter compiler errors for components that installed fine under Delphi, because Borland deemed some Pascal language features undesirable for support in C++Builder (with good reason, in most cases).

For the most part, these are legacy language features left over from TurboPascal days. It's unlikely that you'll use these features in your code--but we won't leave anything to chance. Let's look at three language features C++Builder doesn't support.

The Real data type

Simply put, C++Builder doesn't support the Real data type. If you have any Real variables in your component, you'll get the error *Unsupported language feature: Real* when you attempt to install the component to the Component Palette. In most cases, you can change any Real variables to Double variables to avoid negative ramifications.

The Pascal object model

Early versions of ObjectPascal included the object model. The object was the forerunner to the ObjectPascal class. If you have legacy code you've been carrying forward through the years, you may have some old object-model objects kicking around.

The fix for this problem isn't nearly as easy as that for the Real data type problem--you must take any code that uses the object model and convert the objects into classes. Doing so is a bit of a pain, since the object and class modules handle several items differently (de-referencing pointers, for instance).

Empty sets

At TurboPower Software, we like to create components with a fairly high level of abstraction. That abstraction manifests itself when we create constants to use as default values for properties. Later, if we want to change the default behavior of a particular component, we need to change only the constant declarations--and that change is reflected throughout the code.

In most of our units, this design goal yields const sections of code which look like the following:

```
const:
  DefComNumber = 0;
  DefBaud = 19200;
  DefParity = pNone;
  DefDatabits = 8;
  DefStopbits = 1;
  DefHWFlowOptions = [];
```

Note that the last line creates a constant that's an empty set. For some reason, C++Builder's Pascal compiler can't handle that particular declaration. The result is the compiler error *Error: Unsupported language feature: 'initialized set > 4 bytes in size'*. Chalk this problem up to the mysteries of trying to combine two languages in one development environment. The moral of this story is that you shouldn't

use empty sets in const declarations.

Header-generation issues

Once you've installed a Delphi component on the Component Palette, you may still run into problems when you try to use the component in a C++Builder application. You won't encounter some of these pitfalls until you try to use the header that C++Builder generates for your Delphi component. When the C++ compiler tries to compile your header, it may hit snags that produce C++ errors. Let's take a look at some examples.

Variant records

A *record* in Pascal is synonymous with a *structure* in C++ (except that C++ structures can have functions, but that's beside the point). In Pascal, part of a record may exist as a *variant* portion. A record that has a variant section is called a *variant record* and looks like this:

```
TMyRecord = record
  Ch : Char;
  Cmd : Byte;
  X, Y : Byte;
  case Byte of
    1 : (Other : array[1..11] of Byte);
    2 : (OtherStr : String[10]);
  end;
end;
```

This record's final data member can be either an array of bytes called Other or a Pascal short string called OtherStr.

Note that a record will be the same size, in bytes, regardless of the presence of variant record members. Ultimately, the size of the record will be the sum of the sizes of the non-variant members of the record plus the size of the largest of the variant members. The compiler figures out which variant member is the largest and uses that size to determine the overall size of the record.

C++ supports the concept of the variant record, albeit with different terminology. In C++, a structure may contain an *anonymous union*. The C++ equivalent to the Pascal record we just discussed would be as follows:

```
struct TMyRecord
{
  char Ch;
  Byte Cmd;
  Byte X;
  Byte Y;
  union
  {
```

```
    System::SmallString<10> OtherStr;
    Byte Other[11];
  };
};
```

If you try to compile a program containing this code, you'll get a compiler error that states *Union member TMyRecord::OtherStr is of type class with constructor*.

The bottom line is that a class with a constructor can't be used in a union. But where does the System::SmallString<10> come from in the code snippet? The answer is that C++Builder implements the Pascal short string as a template, a template is a class, a class typically has a constructor, but a class can't have a constructor in a union. End game.

It's not just the Pascal short string that suffers this fate. The Pascal long string, Comp, and Currency data types are also implemented as classes in C++Builder. As a result, you can't use any of these types in the variant part of a variant record. The only way to work around this problem is to design your Pascal records accordingly.

Multiple constructors

You can use multiple constructors in your Delphi components, but you need to take care how you do so. In Pascal, you distinguish multiple constructors by using a different name for each constructor. In C++, all constructors must have the same name (the name of the class itself). The only way to distinguish constructors in C++ is to create a unique parameter list for each constructor.

Take the following Pascal constructors as an example:

```
constructor Create(Owner : TObject; X : Integer); constructor CreateEx(Owner :
TObject; Y : Integer);
```

At the same time C++Builder creates a header for the Pascal unit in which you use these constructors, it will generate the following declarations:

```
__fastcall MyComponent(TObject* Owner, int X);
__fastcall MyComponent(TObject* Owner, int Y);
```

Note that these declarations contain exactly the same number of parameters, and that the parameters appear in the same order.

If you try to use this component in a C++Builder application, you'll get a compiler error. In order to compile this component, you'll have to modify one of the constructors so that the two constructors have unique parameter lists. Doing so may be as simple as adding a dummy parameter to one of the constructors. Be sure that all your constructors have unique parameter lists, including those in base classes.

Properties of arrays

In Pascal, it's legal to return an array by value from a function. In C++, you can't return an array from a function--you can return a *pointer* to an array, but you can't return an array by value.

Many properties use read and write methods to perform special processing when the property is read or written to. A read method for a property must return the value of the data member that's storing the property's data. Since it's illegal to return an array by value in C++, you should avoid using a property that has an array as the property type. This isn't to say that you can't have array properties--array properties are something entirely different. (It's confusing, I know.)

More const issues

Earlier, I mentioned that we use a lot of const declarations in our development here at TurboPower Software. Besides the case mentioned previously, we've run into a couple of other situations C++Builder can't handle properly.

Take this line of code, for instance:

```
const
DefPointer = nil;
```

This declaration causes C++Builder to generate the following declaration in the header for this unit:

```
const void * DefPointer = (void *)(0x0);
```

Although this declaration looks bizarre, the greater sin is that this const declaration appears in a header file.

What's the implication? Let me answer with an illustration. Let's say you have a project that has multiple source code units. In addition, more than one of those source units #includes the header containing this declaration. The application will compile with no problems, but it will fail at link time, complaining that the variable DefPointer is declared in one unit and again in another unit.

I don't know why this particular declaration is translated into a const declaration rather than a #define. Perhaps Borland will fix this problem in future versions of C++Builder. For the time being, you can work around it by declaring the constant as follows:

```
const DefPointer = 0;
```

Another problem with const declarations is less severe. Let's say you have a library of components with several units, and one of those units contains this declaration:

```
const DefaultWidth = 100;
```

So far, so good. But now suppose that the following declaration appears in another unit:

```
const DefaultWidth = 50;
```

If you use both of these components in a C++Builder application, you'll get the linker warning *Redeclaration of DefaultWidth is not identical*. While this is just a compiler warning, not an error, remember that anyone who uses your components will see this warning, too. To fix this minor problem, make sure that your constants have unique names, even if they're contained in separate units.

Some scenarios leading to linker errors

Finally, we've come to the type of error that's sometimes the most frustrating of all: the linker error. In the previous section, I describe a situation that will produce linker errors if you don't remedy it. Let's examine two other issues that will also lead to linker errors.

Pascal class methods

The Pascal class method is a bit of an oddity. You declare a *class method* within a class using the class keyword. You can call this type of method without first creating an instance of the class in which you declare the method.

For example, let's say you have a DoIt class method in a MyClass Pascal class. You could write this line of code:

```
MyClass.DoIt;
```

It's hard to tell much from this code snippet, but you'll see that I didn't create an instance of MyClass before I called the DoIt method.

C++ doesn't have a true equivalent to the Pascal class method--the nearest is a static class member function. (In Delphi, constructors are class methods.) However, implementing static member functions in C++ is different than implementing class methods in Delphi. For example, in Delphi, a class method can use the Self keyword, but in this context it represents the class type and not a class instance.

Borland's engineers worked out a way to deal with Pascal class methods in C++Builder. Unfortunately, the technique fell short of its goal. The function declaration generated by C++Builder for a Pascal class method will let you compile an application, but that method fails at link time.

The best thing you can do is avoid using class methods as public methods in your components. Note that it's fine to use class methods that you reference from your own code--the problem arises when users try to use a class method from a C++Builder application.

HWND versus hWnd

Of all of the problems the Delphi programmer might face when writing components for use in both Delphi and C++Builder, HWND versus hWnd may be the most treacherous. Throughout the development of C++Builder, Borland engineers were plagued by the differences between the way VCL handles the

HWND data type and the way the rest of the world handles this type. In VCL, the hWnd type is declared as an Integer. In the 32-bit C and C++ world, HWND is declared as a void pointer. In one specific case, this difference causes a big problem for component writers.

Suppose a component has a public method that takes a window handle as a parameter or that returns a window handle. For the sake of argument, let's say the function is declared in the Pascal source code as follows:

```
function GetHandle : hWnd;
```

This function returns an hWnd (or HWND or hwnd or... it doesn't really matter in Pascal).

When you install this component, C++Builder will generate a header with the following declaration:

```
HWND GetHandle();
```

This component will install without incident and may even appear to work in a C++Builder application. However, if you attempt to call the GetHandle() function for this component, you'll get a linker error. The linker is looking for a function that has a void* as its return type, but the function in the component's OBJ file was created with an int as its return type. The linker will never be able to find this function because of the difference between the VCL and C++ versions of HWND.

A couple of solutions exist for this problem. One is to use the Integer data type anywhere you'd normally use hWnd. However, the code can be confusing to read later. Another solution is to declare a new type that does essentially the same thing as hWnd:

```
type TMyHwnd = Integer;
```

Now you can use TMyHwnd anywhere you'd typically use hWnd. The advantage to this method is that anyone reading your code will have a clue that something special is going on with this type.

Note that what applies to HWND also applies to HINSTANCE. In VCL, hInstance is declared as a LongInt. In C and C++, HINSTANCE is a void*. Be sure you take this difference into account if you have public functions that use HINSTANCE.

Beware the alignment!

And now, one final word about using Delphi components in C++Builder. C++Builder expects that you'll build your Pascal source code with the default data alignment setting: word alignment. (You control data alignment with the $A Pascal compiler directive.) If you try to use byte alignment, your component will experience strange and inexplicable behavior.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

July 1997

Finding a console window

by Kent Reisdorph

If you write console applications, you may occasionally need to determine the window handle (HWND) of your console application. You may even need the window handle of another application's console window. With that information, you can shut down the console window, move it, hide it, or perform any of a number of other operations.

In this article, we'll explain how to obtain the window handle of a console application. We'll demonstrate how to use the FindWindow() function and some of the tricks involved in finding console windows in both Windows 95 and Windows NT.

FindWindow

You can use the FindWindow() Windows API function to obtain the window handle of any window--main application windows, controls, dialog boxes, and, of course, console windows. The syntax for FindWindow() is as follows:

```
HWND hWnd = FindWindow("MyWindowClass", "My Program");
```

The first parameter specifies the class name of the window, and the second parameter specifies the window's text--typically, the window's caption.

If the second parameter is NULL, then the function will find any window with the specified class name. For this reason, you should always specify the window text parameter unless you know that the window you're looking for has a unique class name.

A console application has an associated window. So, you can use FindWindow() to obtain the window handle of the console application just as you'd use it in a conventional GUI Windows program. The trick is to supply the proper class name and window text.

Making a difference

It probably comes as no surprise that Microsoft handles some console-window issues differently in Windows 95 than in Windows NT 4--in particular, the class name and the console window's caption. **Table A** illustrates these differences. In order to account for these variations, you must check to see which operating system your application is running under,

and then take the appropriate steps to find the console window handle.

Table A:*Console application differences between Windows 95 and Windows NT*

	Windows 95	**Windows NT**
Console window class name	tty	ConsoleWindowClass
Console window caption	Program name only	Full path, filename, and extension

Once you determine which operating system you're using, you can call FindWindow() with the appropriate parameters. As I hinted before, just knowing the class name usually isn't good enough to ensure that you'll locate the correct window. For example, the user may have several console applications or command prompt boxes open at any given time. Supplying only the class name will cause FindWindow() to return the window handle of the first console window it encounters--which may or may not be the one you're looking for.

Where am I?

To determine which operating system you're running under, you can use the Windows API function GetVersionEx(). This function returns information about the operating system in a OSVERSIONINFO structure. VCL defines the typedef TOSVersionInfo, so you can use that if you like.

Before you call GetVersionEx(), you must create an instance of the TOSVersionInfo structure and set the dwOSVersionInfoSize member of the structure to the size of the structure itself. For example, you can use code like the following:

```
TOSVersionInfo info;
info.dwOSVersionInfoSize = sizeof(info);
GetVersionEx(&info);
```

Now the TOSVersionInfostructure contains information about the operating system currently in use. The only data member of this structure that you care about in this case is dwPlatformId. This data member will contain the value

```
VER_PLATFORM_WIN32_WINDOWS
```

if the user is running Windows 95, and the value

```
VER_PLATFORM_WIN32_NT
```

if the user is running Windows NT.

Armed with that operating system knowledge, you can write code like the following:

```
if (info.dwPlatformId == VER_PLATFORM_WIN32_NT)
  // running NT
else
  // running Win95
```

With this code snippet, you can find a console window regardless of whether the user is running Windows 95 or Windows NT.

An example

Listing A contains a function you can use to determine the window handle of a console application. This function detects the OS version, then calls FindWindow() with the appropriate parameters. The function returns the result of the FindWindow() function--either the window handle of the console window, if the function found the window, or NULL if the function couldn't find the window.

Listing A: *GetConsoleHWND() Function*

```
// This routine gets the window handle of
// a console window.
HWND GetConsoleHWND()
{
  char title[256] = "";
  char className[20] = "";

  // Get the name of this executable.
  GetModuleFileName(0, title, sizeof(title));

  // See if we're running on Win95 or NT.
  TOSVersionInfo info;
  info.dwOSVersionInfoSize = sizeof(info);
  GetVersionEx(&info);
  int platform = info.dwPlatformId;
```

```
  // If NT then the class name is 'ConsoleWindowClass'
  // The title is the entire path and filename.
  if (platform == VER_PLATFORM_WIN32_NT)
    strcpy(className, "ConsoleWindowClass");

    // If Win95 then the class name is 'tty' and
    // the window title is just the filename
    // without path and extension.
  else {
    strcpy(className, "tty");

    // Strip off everything but the filename.
    std::string name = title;
    int pos = name.find_last_of("\\");
    name.remove(0, ++pos);
    name.remove(name.length() - 4, 4);
   strcpy(title, name.c_str());
  }

  // Return the result of FindWindow().
  return FindWindow(className, title);
}
```

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

Premiere Issue

Incorporate custom message-handling in your applications

by Kent Reisdorph

At the operating system level, Windows is all about messages. Much of what happens in Windows is due to Windows dispatching one message or another. VCL provides events for the most often used Windows messages; you can easily handle a wide variety of Windows messages through these events. The OnKeyDown event provides the mechanism for responding to Windows WM_KEYDOWN messages, the OnMouseDown event lets you respond to WM_LBUTTONDOWN and WM_RBUTTONDOWN messages, and so on. Your C++Builder programs will handle most Windows messages through events.

However, VCL doesn't provide events for every possible Windows message--there are just too many of them. If you want to handle Windows messages that aren't encapsulated in VCL, if you want to implement user-defined messages, or if you need custom handling of Windows messages, then you'll have to roll up your sleeves and do a little more work. In this article, we'll explain how to handle messages in your applications when you need to work outside of the VCL event mechanism.

Messages and message cracking

Let's briefly review how the Windows message system works. First of all, as you probably know, Windows is driven by messages. Windows sends messages to programs (to specific windows, actually) for many different reasons, perhaps to tell a window to perform certain tasks. For example, Windows might tell a window to repaint itself by sending a WM_PAINT message. You could think of this as a command message, since Windows expects the application to respond by carrying out a task.

Windows also sends messages to notify an application that particular events have occurred. Perhaps an event has occurred within a window that the application might like to know about (messages such as WM_SIZE and WM_MOVE), or maybe something has happened within Windows itself (WM_SYSCOLORCHANGE or WM_WININICHANGE, for example).

Since controls are windows, Windows sends notification messages when an event happens within a control. Examples include a text change in an edit control (EN_UPDATE) or a user's selection of a list box item (LBN_SELCHANGE). In all, there are hundreds of Windows messages. It's up to the application to respond to messages it deems pertinent to its operation.

Message parameters

A Windows message has two parameters--the WPARAM and the LPARAM--that contain information specific to the message being sent. The WPARAM (word parameter) is a 16-bit value in 16-bit Windows but is a 32-bit value in 32-bit Windows. The LPARAM (long parameter) is a 32-bit value in either 16- or 32-bit Windows.

Windows manipulates these two parameters to carry information about the particular message being sent. For example, the WM_SIZE message contains three pieces of information: the type of sizing that's occurring (minimizing, maximizing, or restoring), the window's new width, and the window's new height. In this case, the WPARAM contains the size type and the LPARAM contains the new width and height of the window. The LPARAM is packed to hold the width and height values--the first 16 bits (called the *low-order word*) contain the new width, and the last 16 bits (the *high-order word*) contain the new height.

Most Windows messages include information sent in this manner. It's up to the application to crack the message parameters and retrieve the information Windows is sending. In some cases, a great deal of information is involved--Windows sometimes passes a pointer to a structure as part of the LPARAM parameter.

The VCL shield

Part of the job of a framework such as VCL is to shield you from the vagaries of handling messages at this level. For example, the OnKeyDown event that handles the WM_KEYDOWN message cracks the message parameters for you. The message handler that C++Builder generates for the OnKeyDown event looks like this:

```
void __fastcall
TForm1::FormKeyDown(
TObject *Sender, WORD &Key,
TShiftState Shift)
{
}
```

As you can see, you don't get the raw WPARAM and LPARAM--the parameters are already cracked.

In this case, the WPARAM is in the form of a variable named Key, which tells you which key was pressed. The LPARAM (which is very complicated in the WM_KEYDOWN message) is in the form of a TShiftState object that tells you which, if any, of the [Shift], [Alt], and [Ctrl] keys were down when the key was pressed. The message-cracking that VCL does for you greatly simplifies the message- handling process in Windows programming.

The VCL message structures

Cracking the WPARAM and LPARAM values can be a pain. Fortunately, VCL provides message-cracking structures that have done most of the work for you. (You'll find these structures in the MESSAGES.HPP file, if you want to examine them.) You can use these structures when you create message-handling functions in your applications, as we'll discuss later.

The generic message structure, which could handle any message, looks something like this:

```
struct TMessage
{
  unsigned int Msg;
  long WParam;
  long LParam;
  long Result;
};
```

(I say it "looks something like this" because the actual TMessage structure contains a union that muddies the waters--so, I simplified things a bit.) In this case, you pass the WPARAM and LPARAM as-is, and you don't crack the message parameters. If you use this message structure, then it's up to you to extract the individual message values from the WPARAM and LPARAM members of TMessage. You could use this particular structure for user-defined messages or for messages that pass no values. Most of the time, however, you'll use the structure specific to the message you're handling.

All of the message-cracking structures have two members in common: Msg and Result. The Msg member contains the number of the message sent. For example, the WM_ERASEBKGND message has a value of 20. So if a WM_ERASEBKGND message was received, then the value of the Msg member of the TMessage structure would be 20. Most of the time, it isn't necessary to examine the Msg member in your message-handling functions.

How you use the Result data member depends on the message being sent. For example, if you handle the WM_ERASEBKGND message in your application, then you should return true from your message handler for that message. If you don't handle the message, then you should return false. In C++Builder applications, you manage this process by setting the value of the Result data member at the end of your message handler for WM_ERASEBKGND. For some messages, you don't modify the Result data member at all. In other cases, the value of Result could be a numerical value--it depends entirely on the message being processed.

Aside from the Msg and Result data members, each message-handling structure will have a number of parameters. For example, the WM_ERASEBKGND message's message-handling structure, TWMEraseBkgnd, looks like this:

```
struct TWMEraseBkgnd
{
  unsigned int Msg;
  HDC DC;
  long Unused;
  long Result;
};
```

This code defines the DC data member (the WPARAM for this message) as a handle to a device context (HDC); the LPARAM is unused. In this case, there isn't much message cracking to do.

Now, let's look at the message-cracking structure for the WM_SIZE message:

```
struct TWMSize
{
  unsigned int Msg;
  long SizeType;
  unsigned short Width;
  unsigned short Height;
  long Result;
} ;
```

Here, the WPARAM is the size type (SizeType) and the LPARAM is broken down into the Width and Height from having to break down the WPARAM and LPARAM for every message you wish to handle. Later, we'll demonstrate how you can use the message-cracking structures in your C++Builder applications.

Handling VCL events

I won't go into much detail on handling VCL events because event-handling is a basic part of programming in C++Builder (you should be familiar with it by this time). Instead, I'll hit the high points.

As I said earlier, VCL provides events for many of the most commonly used Windows messages. This VCL feature allows you to attach an event-handling function to an event. When the event occurs, your function will be called automatically. In your event-handling function, you write code that performs tasks specific to that message. VCL makes handling the most common Windows messages so simple that it's easy to forget how tedious message handling can be in a traditional Windows program.

Handling other messages

From time to time, you'll need to handle messages for which VCL doesn't provide events. WM_ERASEBKGND and WM_SETCURSOR come to mind as messages you might want to handle in your applications. To handle messages outside the normal VCL event model, you'll have to do the following:

- Create a message-map table in the class declaration for the window that will handle the message
- Add an entry to the message-map table for the message you want to handle
- Provide a message-handling function

Let's examine these steps more closely.

Create a message-map table

In order for you to handle messages for which no VCL event exists--including user-defined messages--you'll need to create a message-map table. A typical message-map table looks like this:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(MYMESSAGE, TMessage, OnMyMessage)
  MESSAGE_HANDLER(WM_ERASEBKGND, TWMEraseBkgnd, WmEraseBkgnd)
END_MESSAGE_MAP(TForm)
```

If you're familiar with Borland's Object Windows Library (OWL), then you might recognize this code as similar to OWL's response table. However, elements in the C++Builder message-map table are rearranged slightly.

Like the OWL response table, the C++Builder message map is made up of macros. The first macro, BEGIN_MESSAGE_MAP, marks the beginning of the message map; this macro takes no parameters. Next, you need a MESSAGE_HANDLER macro for each message you wish to handle. The MESSAGE_HANDLER macro identifies the name of the message in the first parameter, the name of the message structure in the second parameter, and the name of the message-handling function in the last parameter. Finally, the END_MESSAGE_MAP macro marks the end of the message-map table.

You place the message-map table in the header of the class that will handle the messages. The table goes in the public section of the class declaration--usually at the end. For example, if you handle the WM_ERASEBKGND message in your main form, then the declaration for the

TForm1 class will look something like the code shown in **Figure A**.

Figure A: *You can use a TForm1 class declaration like this one if you handle the WM_ERASEBKGND message in your main form.*

```
//---------------------------------------------

#ifndef MyMainH
#define MyMainH

//---------------------------------------------

#include
<vcl\Classes.hpp>

#include
<vcl\Controls.hpp>

#include
<vcl\StdCtrls.hpp>

#include
<vcl\Forms.hpp>

#include
<vcl\ExtCtrls.hpp>

//---------------------------------------------

class TForm1 : public TForm
{
  __published: // IDE-managed Components

  private: // User declarations

  //
  // Message handler declaration
  //

  void __fastcall WmEraseBkgnd(TWMEraseBkgnd &Message);
  public: // User declarations

  virtual __fastcall TForm1(TComponent* Owner);

  //
  // The message-map table.
```

```
  //

  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_ERASEBKGND,
    TWMEraseBkgnd, WmEraseBkgnd)
  END_MESSAGE_MAP(TForm)

};

//---------------------------------------------
extern TForm1 *MainForm;

//---------------------------------------------

#endif
```

The message-map table goes in the class declaration because the message-map macros expand to an overridden Dispatch() function. The Dispatch() VCL function handles Windows messages and dispatches them to the appropriate event-handling function, if any. (If you'd like to study this topic more closely, you can find the message-map macros in SYSDEFS.H.) The Dispatch() function calls the appropriate message-handling function when it receives a message from Windows. If no message handler exists for a message, then the message passes on to the base class Dispatch() function for handling.

Add message-map table entries

After you've created the message-map table, you need to add a table entry for each message you want to handle in your application. The actual entry is a MESSAGE_HANDLER macro. For the WM_ERASEBKND message, the entry will look like this:

MESSAGE_HANDLER(WM_ERASEBKGND, TWMEraseBkgnd, WmEraseBkgnd)

As I mentioned earlier, the message-map entry for the message you want to handle will contain the name of the message to handle, the name of the message structure corresponding to that message, and the name of the message-handling function. For user-defined messages, you can use the TMessage or TWMNoParams structure, or you can create your own message-handling structure.

Create the message-handling function

Finally, to process a message, you need a message-handling function. This function must follow a particular signature: It has just one parameter, returns void, and uses the __fastcall

modifier. The single parameter is a reference to a message structure.

Begin by declaring the message handler in the private section of the class that will be handling the message. For the WM_ERASEBKGND message, the declaration would look something like this:

```
void __fastcall
WmEraseBkgnd(

TWMEraseBkgnd&
Message);
```

The name of the function isn't important, but by convention, you'll want to give it the same name as the message you're handling.

Next, you'll need to provide the implementation of the message-handling function, which goes in the source unit of the form or class that's responding to the message. For example, the following WM_ERASEBKGND message handler draws a hatched pattern on the form's background:

```
void __fastcall

TForm1::WmEraseBkgnd(TWMEraseBkgnd&Message)
{
  TCanvas* canvas = new TCanvas();
  canvas->Handle = Message.DC;
  canvas->Brush->Handle = CreateHatchBrush(HS_DIAGCROSS, clBlue);
  canvas->FillRect(ClientRect);
  Message.Result = true;
  delete canvas;
}
```

This code first creates a TCanvas object and assigns the HDC passed in the message-handling structure to its Handle property. By doing so, the code lets you take a raw device context and manipulate it using the TCanvas properties and member functions. Next, the code creates a hatched brush and assigns it to the canvas's Brush property. After that, the code fills the client area rectangle with the current, hatched brush. You then set the Result member of the message structure to true to tell Windows that you handled the message and there's nothing more for Windows to do. Finally, you delete the TCanvas object so you don't leak memory. You'll need to follow this process to create a message-handling function for each message you expect to handle.

Default handling for a message

Frequently, when you're handling a Windows message, you want to perform some action but also receive the default Windows behavior for that message. For instance, you might want to force all characters in an edit control to be uppercase. Doing so would entail modifying the characters, then sending the message to Windows for handling. In that case, you'd modify the Key parameter of the message-handling structure, then call the base class's Dispatch() member function to pass the message to Windows, as follows:

```
void __fastcall

TForm1::OnKeyDown(TWMKeyDown&Message)
{
  if (Message.CharCode > 96 && Message.CharCode < 123)
  Message.CharCode -= 32;
  TForm::Dispatch(&Message);
}
```

Since Dispatch() requires a pointer to a TMessage structure, you need to take the address of the Message variable when you pass the structure to VCL (which in turn passes the message to Windows for further processing). Whether you call Dispatch() before or after you modify the message parameters depends, again, on the particular message you're handling.

You can also get default handling for a message by calling DefaultHandler(). In theory, calling Dispatch() should suffice, but I've found that in some situations, calling DefaultHandler() produced different results than calling Dispatch(). Whether the difference was due to oddities in my code or in VCL, I don't know--but you should be aware that there's more than one way to pass a message to Windows.

Hatching an example

Listings A and **B** contain a program that uses custom message-handling to draw a hatched background on a form. The program works by catching the WM_ERASEBKGND message and painting the background in the WmEraseBkgnd message handler. Create a new project and substitute the code in these two listings for the code produced by C++Builder. When you run the program, the form will have a blue hatched background, as **Figure B** shows.

Listing A: MSGTEST.H

```
#include
```

```
      <vcl\Forms.hpp>


//---------------------------------------------

class TForm1 : public TForm {
  __published: // IDE-managed components private:

  // User declarations
  void __fastcall WmEraseBkgnd( TWMEraseBkgnd& Message);

  public: // User declarations
  virtual __fastcall TForm1( TComponent* Owner);

  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER( WM_ERASEBKGND, TWMEraseBkgnd, WmEraseBkgnd)
  END_MESSAGE_MAP(TForm)

};

//---------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------
```

Listing B: MSGTEST.CPP

```
//---------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"

//---------------------------------------------------
#pragma resource "*.dfm"
TForm1 * Form1;

//---------------------------------------------------
__fastcall TForm1::TForm1( TComponent* Owner) : TForm(Owner) { }

//---------------------------------------------------
void __fastcall TForm1::WmEraseBkgnd(TWMEraseBkgnd& Message) {
  Canvas* canvas = new TCanvas();
  canvas->Handle = Message.DC;
```

```
  canvas->Brush->Handle = CreateHatchBrush(HS_DIAGCROSS, clBlue);
  canvas->FillRect(ClientRect);
  Message.Result = true; delete canvas;
}

//---------------------------------------------------
```

Figure B: ***Our program produces a blue hatched background on the form.***

Conclusion

The built-in VCL events do a great job of allowing you to handle most Windows messages. However, the VCL events don't cover every Windows message. Using the information we've provided in this article, you now have the tools to handle any Windows message, regardless of whether VCL provides an event for it.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at editor@bridgespublishing.com.

Premiere Issue

C++Builder extensions to C++

by Kent Reisdorph

By now, C++Builder is on the streets and into the hands of thousands of users. Also on the streets is the news that Borland wrote extensions to C++ so that C++Builder could take full advantage of the power of the PME (property, method, and event) model. This news probably elicits one of two responses from you: either "So what?" or "That's bad news!"

Regardless of your initial reaction, it pays to have an informed opinion, so it's important that you get all the facts before you make your final judgment. In this article, we'll examine several significant C++Builder extensions and discuss what they mean to you as a C++Builder programmer.

Standard C++ is alive and well and living in C++Builder

First of all, you need to realize that you can compile any standard C++ code with C++Builder. Yes, Borland added extensions to C++ to maximize the power of the PME model, but that doesn't mean C++Builder isn't an ANSI-compliant C++ compiler. It is. However, I assume you didn't buy C++Builder to write OWL or MFC applications, but rather to take full advantage of the rapid application development (RAD) power C++Builder offers. If you want to utilize C++Builder's RAD capabilities, then you'll have to accept its extensions to C++.

My advice is simple: Don't sweat it. If you're in an environment where a higher power dictates that you must limit your code to ANSI-compliant C++, then you're out of luck as far as C++Builder goes. On the other hand, if you have the freedom to choose your development environment, then I suggest you jump into C++Builder with both feet and not look back.

To extend or not to extend?

It's safe to assume that Borland didn't arrive at a final conclusion without a great deal of thought and research. As I understand it, Borland conducted surveys to ask developers what they wanted in a RAD C++ product. The survey results indicated that plenty of developers would be willing to tolerate a small-scale extension of the C++ language in hopes of RAD coming to C++. The response was strong enough for Borland to determine that producing C++Builder was worth the effort.

As far as I know, all the big compiler vendors (those who produce compilers for the PC, anyway) have extended C++ to suit their own needs. Given that assumption, we're now talking about questions of degree (Borland extended C++ *more* than Microsoft did), and such discussion can be a particularly fruitless debate.

As I've said, no one is forcing you to use the C++ extensions that C++Builder introduces. The decision to use C++Builder or not is entirely up to you. Just don't get drawn into the idea that the other C++ compilers are 100 percent standards-compliant, because they certainly aren't. (The one possible exception is Borland C++, which

has always maintained the high road in the quest for a C++ compiler that adheres to the standards.)

So what are these extensions?

So you want to know what these extensions are all about, do you? All right, let's take a look. C++Builder adds several new keywords to C++:

```
__automated
__classid
__closure
__int8
__int16
__int32
__int64
__property
__published
```

Some of these extensions are for C++Builder's internal use only, more or less (__classid, for instance). Others are fairly obvious (__int8, __int16, and so on), so we won't examine each and every one of them. Instead, we'll focus on the most significant extensions. (Note that while the __automated keyword is a significant extension, we can't provide adequate coverage in the space of this article. Therefore, we'll save a discussion of __automated for another time.)

__closure

The __closure extension is easily one of the biggest changes that C++Builder makes to the C++ language. You use the __closure keyword to declare a *closure*, which is an eight-byte pointer that contains the function address in its first four bytes and the instance of the class where the function resides in its last four bytes. This arrangement makes it possible for you not only to call a function of a particular class, but to call a function in a particular *instance* of that class, as well. Such a capability exists in Object Pascal (called *method pointers* in OP), but doesn't inherently exist in C++. Closures are vital for implementing events in VCL. (We'll talk more about events in a future issue.)

__property

You use the __property keyword within a component class declaration to identify a component's properties. A basic property declaration looks like this:

```
__property int Width;
```

However, a property declaration can be much more involved than this simple example. For instance, a property might use direct access to read the property's value and a write method when the property is written to. The property might also have a default value. In that case, the property

declaration would look something like this:

```
__property int Width =  {read=FWidth, write=SetWidth, default=200}
```

As you can see, this code is not C++ as you've previously known it. The __property keyword allows this syntax.

__published

A property can be either published or non-published. If a property is published, the Object Inspector displays it at design time. A nonpublished property doesn't appear in the Object Inspector at design time. (Runtime-only properties are nonpublished.) To declare a property as published, you place its declaration in the __published section of the component's class declaration using code like that shown in **Figure A**. As you can see, you use the __published keyword just as you use the private, protected, and public keywords in standard C++.

Figure A: *You declare a property as published by declaring it in the __published section of the class declaration.*

```
class MyEditComponent : public TCustomEdit
{
  private:
  // Private data members and
  // function declarations here.

  protected:
  // Protected declarations here.

  public:
  // Public declarations here.

  __published:
  // The component's published
  // properties go here.

  __property int Width = {read=FWidth, write=SetWidth, default=200}
};
```

Conclusion

If you want RAD C++, then you must accept the C++Builder extensions to C++. If you can't live with the C++ extensions, then you can continue with your current Windows programming environment. However, I believe you'll find that the benefits of being able to use RAD make your extra efforts to master the extensions well worth the work.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

Premiere Issue

Controlling the cursor

by Kent Reisdorph

Windows lets you control the mouse cursor's appearance. When used properly, cursors give your application a polished look while providing visual feedback for users. (Of course, as with all GUI features, you shouldn't overdo it.) Although VCL makes cursor changes a breeze, you need to be aware of certain aspects, which we'll discuss in this article.

VCL cursor architecture

The VCL cursor architecture is relatively simple, but you can easily get lost if you're new to C++Builder and VCL. Let's examine how VCL manages and implements cursors.

The Cursor property

All VCL components derived from TControl have a Cursor property. This property controls the cursor that Windows displays when the user passes the mouse pointer over a component. (Remember, forms are components, too.)

By default, the Cursor property is set to crDefault. The crDefault style tells Windows to display the default cursor for that type of window (or component, if you prefer). For instance, if you have an Edit component on a form, then Windows will, by default, change the cursor to the I-beam shape when you pass it over that component. (This change happens only at runtime, not at design time.) You don't have to do anything to get the default cursor for your components, since the Cursor property always defaults to crDefault.

Besides changing cursors for an individual component, you can change the Cursor property for the Screen object. In VCL, the Screen object represents your application's client area. If you want a cursor change to take place regardless of the component that the cursor is over, then you'll need to modify the Cursor property of the Screen object, rather than the Cursor property of the components or of the form. Let's see why this is so.

Cursors operate on at least three levels as far as VCL is concerned. If you change the Cursor property for an individual component, then the cursor changes only when it's over that component. If you change the Cursor property for the form, then the cursor will change when it's over the form but not when it's over any of the components on the form. However, if you change the Cursor property for the Screen object, then the cursor will change when it's over the form *and* when it's over all components on the form. Whether you change the cursor for an individual component, for the form, or for the entire application

depends on your needs for the present circumstance.

The Cursors property

In addition to the Cursor property, we need to mention the Cursors property, a member of the TScreen class. *Cursors* is an array property that contains a list of the cursors currently available to your application. (At first you may find it somewhat confusing to have both a Cursor property and a Cursors property, but after you've been around the block a time or two, you'll get the hang of it.)

An out-of-the-box C++Builder application includes many cursors. **Table A** lists them and briefly describes each one.

Table A: *VCL cursors*

Cursor	Description
crDefault	Default cursor for the component
crNone	No cursor; cursor disappears when over the component
crArrow	Standard arrow
crCross	Crosshair
crIBeam	Text-editing I-beam
crSize	Same as crArrow
crSizeNESW	Sizing cursor oriented diagonally from northeast to southwest
crSizeNS	Sizing cursor oriented vertically
crSizeNWSE	Sizing cursor oriented diagonally from northwest to southeast
crSizeWE	Sizing cursor oriented horizontally
crUpArrow	Arrow pointing up
crHourGlass	Hourglass
crDrag	Arrow with a blank page in the lower-right corner
crNoDrop	Diagonal slash through a white circle
crHSplit	Black double-vertical bar with arrows pointing right and left
crVSplit	Black double-horizontal bar with arrows pointing up and down
crMultiDrag	Arrow with three blank pages in the lower-right corner
crSQLWait	SQL beneath a wait cursor
crNo	Diagonal slash through a black circle

crAppStart	Arrow combined with an hourglass
crHelp	Arrow next to a black question mark

VCL provides some of the cursors listed in **Table A**; the rest are built into Windows. It isn't necessary to know which are which.

If you don't want to use one of the built-in cursors, you can add your own custom cursors to the Cursors array. Before we discuss changing cursors at runtime, let's take a look at installing custom cursors.

Installing custom cursors

Although C++Builder provides the most commonly used cursors, you still may require custom cursors for your applications. Creating and implementing custom cursors isn't terribly complicated, but you need to know which hoops to jump through and in what order to jump. Adding a custom cursor requires that you take the following steps:

1. Create the cursor with a resource editor.
2. Bind the cursor resource to your EXE file.
3. Add the new cursor to the Cursors array at runtime.

Let's examine each of these actions in more detail.

Create the cursor

The first thing you must do is to create the cursor itself--you can use any resource editor for this job. The Image Editor utility that comes with C++Builder works pretty well for creating simple resources such as cursors; if you're a longtime Borland user, you might prefer to use Resource Workshop. Either way, it's a trivial task. (Note that if you use Image Editor to create your cursor, you need to start with a new resource project and add a cursor resource to the project. When you save the project, the Image Editor will compile the project into a binary resource file that the linker will need in order to bind the cursor resource to the executable file.)

I'm not going to describe the exact steps required to create a cursor--check resource editor documentation for details. The important thing is that when you're done, you'll end up with either a compiled resource file (RES) or a resource script file (RC).

Bind the cursor to your EXE file

Before your application can access a custom cursor, you must bind the resource to the executable file. (You could also place the cursor in a DLL, but we'll leave that discussion for another article.) The linker actually carries out the process of binding the cursor resource to the executable file--all you have to do is tell the linker where to find the cursor resource.

To add the cursor resource to your EXE, just choose Add To Project from C++Builder's Project menu or click the Add To Project button on the ToolBar. When the Add To Project dialog box opens, locate the RES or RC file in which the cursor resides and click OK. Now, the C++Builder IDE will add either a

```
USERES()
```

or

```
USERC()
```

directive to your project source file. For example, if you had a cursor in a file called CROSS.RES, the IDE would add the following line to your project source file:

```
USERES("cross.res");
```

The next time you click the Run button, do a Make, or do a Build All, the linker will bind the cursor resource to the executable file. Now that the cursor is in the EXE, you need to put it to use.

Add the cursor to the Cursors array

The Cursors array contains a list of all the cursors available to the application. Before you can use your custom cursor, you must load it into the Cursors array. The predefined cursors occupy indexes -17 through 0 in the array--you may use any other index number when you add a custom cursor.

Add the cursor to the array using the Windows API function LoadCursor(), which looks like this:

```
Screen->Cursors[1]= LoadCursor(HInstance,"IC_PENCIL");
```

This line tells VCL to load the cursor called

```
IC_PENCIL
```

from the program's executable file and assign it to index 1 of the Cursors array. If you have several cursors, you may want to define and use constants rather than raw index numbers, as follows:

```
const TCursor crPencil = 1;
Screen->Cursors[crPencil]
=
LoadCursor(HInstance,
"IC_PENCIL");
```

Once you've loaded your new cursor into the Cursors array, you can use it. Let's look at how to do so.

Changing cursors

To change the cursor for a particular component at design time, just alter the Cursor property in the Object Inspector. Each time the mouse cursor passes over the component, the cursor will change accordingly. Set the cursor for any specialty components at design time whenever applicable.

While you can change cursors at design time, you'll most often want to change the cursor at runtime. For example, if your program starts a lengthy process, you should let the user know by changing the cursor to the hourglass shape (sometimes called the *wait cursor*) for the duration of the process, then back to the arrow shape when the process finishes.

To change a cursor at runtime, you set the Cursor property of the form, Screen object, or other component to one of the cursors contained in the Cursors array:

```
Screen->Cursor = crHourGlass;
```

It's that simple for the built-in types. You can use any of the constants from **Table A** to quickly and easily change the cursors in your applications at runtime.

If you want to change to a custom cursor that you've previously added to the Cursors array, then you specify the actual array index:

```
Screen->Cursor = (TCursor)1;
```

While the cast to TCursor isn't strictly necessary, the step avoids the compiler warning *Assigning int to TCursor*. The program would probably work fine anyway, but I always apply the cast to avoid the warning.

A better approach is to use a constant for your cursor, as we discussed earlier. Then you can use this line:

```
Screen->Cursor = crPencil;
```

Now you have more readable code that will compile without issuing a warning.

Cursor concerns

Earlier, I mentioned that you can change the cursor for the Screen object to achieve a global cursor change effect. However, when you change the cursor for the Screen object, you may get some unexpected results. Specifically, you'll notice that Windows uses your cursor shape while the cursor is over any part of your application *except* the title bar and the menu bar (the nonclient area of the application). When the cursor is over the title bar, it reverts to the arrow shape. The result is that users can minimize or close the application, move the window around onscreen, and even access the menu. This is standard Windows behavior, but it's probably not what you want to have happen.

In many circumstances, you don't want the cursor to revert to the arrow shape when it's over the nonclient areas. For example, a critical process in an application should alert the user by displaying the hourglass cursor and should then prevent the user from doing anything else in the application until the process has finished. If this type of behavior is what you're after, then you must take extra steps to get VCL to perform for you.

Essentially, you need to grab the WM_SETCURSOR message and handle it yourself. (See the article "Incorporate Custom Message-Handling in Your Applications" for details on handling messages outside of VCL.) After you set up the message map for your application, your custom message handler for the WM_SETCURSOR message should look something like this:

```
void __fastcall TMainForm::OnSetCursor(TWMSetCursor &Message) {
if (working) {
  // load the stock Windows hourglass cursor
  ::SetCursor(LoadCursor(NULL, IDC_WAIT));
   Message.Result= true;
}
else
  TForm::Dispatch(&Message);
}
```

When you use this technique to change the cursor, it won't revert to the arrow shape when it moves over the nonclient area of the application. While this technique involves a little more

work, it gives you more flexibility than you get with the pure VCL method.

Conclusion

Effective use of cursors can help to distinguish your application from the competition's. As we've explained in this article, changing cursors--even adding custom cursors--is simple once you know how to do it.

Kent Reisdorph is a editor of the *C++Builder Developer's Journal* as well as director of systems and services at TurboPower Software Company, and a member of TeamB, Borland's volunteer online support group. He's the author of *Teach Yourself C++Builder in 21 Days* and *Teach Yourself C++Builder in 14 Days*. You can contact Kent at **editor@bridgespublishing.com**.

Premiere Issue

Display forms without captions

by Tim Gooch

Captions are ubiquitous in Windows applications. However, on certain occasions, you might not want to display a caption for a window.

For instance, you may want to create a splash screen during application startup or a full-screen window that obscures everything else onscreen (as some installation applications do). In this article, we'll show you two ways you can create forms that don't display captions.

"A form with no name..."

Forms without captions aren't unheard of, but they're not a terribly common user-interface element. However, C++Builder does provide two techniques you can use to remove a form's caption: adjusting the BorderStyle property and overriding the CreateParams() member function to specify custom window attributes.

BorderStyle

The most common way to eliminate a form's caption is to set its BorderStyle property to the value bsNone. This technique is simple to implement, but it has a significant drawback. As **Figure A** shows, a bsNone border style really means no borders at all.

Figure A: *If you set a form's BorderStyle property to bsNone, the form's borders will disappear.*

Obviously, you don't encounter many situations in which you want to display a truly borderless form. One exception to this convention is a full-screen form that you might use to intentionally obscure the other windows on the system, such as a screen saver.

To see this type of form in action, create a blank-form project. Then, set the form's BorderStyle property to bsNone and the Color property to clBlack. Next, create an OnKeyDown event handler like the following:

```
void__fastcall TForm1::FormKeyDown(TObject*Sender, WORD &Key, TShiftState Shift)
{
```

```
  Close();
}
```

Finally, create an OnCreate event handler like this one:

```
void__fastcall TForm1::FormCreate(TObject *Sender)
{
  Top = 0;
  Left = 0;
  Width = Screen->Width;
  Height = Screen->Height;
  Cursor = crNone;
}
```

This code resizes the form to the screen's dimensions, then hides the mouse cursor. Build and run the application to confirm that it turns the screen black and doesn't display its border.

CreateParams()

The second technique we'll demonstrate takes advantage of the CreateParams() member function. To eliminate the caption from a form's window, you must adjust the

`Params.Style`

parameter field; however, this isn't an intuitive change. As with many attribute adjustments using Windows API functions, the different styles you can combine in the

`Params.Style`

field interact in some surprising ways. To achieve the desired effect (a captionless form), we must modify the style in a deliberate manner.

The WS_POPUP style specifies a standard window that may or may not have a caption and a border. In contrast, the WS_DLGFRAME style specifies only that the window must have a caption.

Since the standard border styles typically provide some sort of border (with the exception of bsNone), we can take the initial style settings (which come from the BorderStyle property) and modify them using the other two styles. Combining the WS_POPUP style with the default form style makes it possible for us to eliminate either the window border or the caption.

Next, we'll apply the WS_DLGFRAME style in reverse. That is, we'll invert this style and apply it to the existing style.

Now let's create a captionless form using this technique. First, create a blank-form project, and add the following

`CreateParams()`

member function declaration to the private section of the form's class declaration. (Right-click on the code-editing window for the main form and choose Open Source/Header File from the pop-up menu to display the declaration.)

```
void__fastcall
CreateParams(TCreateParams &Params);
```

Then, add the following method body to the main form's source file:

```
void__fastcall TForm1::CreateParams(TCreateParams &Params)
{
  TForm::CreateParams(Params);
  Params.Style |= WS_POPUP;
  Params.Style ^= WS_DLGFRAME;
}
```

Finally, add the line shown in color to the form's OnClick event handler:

```
void__fastcall TForm1::FormClick(TObject *Sender)
{
  Close();
}
```

You're now ready to build and run the application. When the main form appears, you'll notice that it doesn't have a caption. However, it does have the standard, resizeable border, as shown in **Figure B.**

Figure B: *You can override the CreateParams() member function to remove the caption from a standard form.*

Click on the form to exit, change the form's BorderStyle property to another style, then build and run the application again. This time, the caption remains hidden, but the form's border will display the new border style.

Premiere Issue

Getting IN to SQL

by Brian Schaffner

If you find yourself using SQL (Structured Query Language) statements like the following:

```
SELECT *
FROM ORDERS
WHERE EmpNo='11'
OR EmpNo='12'
OR EmpNo='114'
```

then you may be using more keystrokes than you really need to. This query uses multiple criteria on a single field. In this article, we'll tell you about the SQL IN operator and how it can help you with SQL statements like the one above. We'll begin by defining situations in which you can use IN. Then, we'll show you how you can use the IN operator in place of other SQL statements.

Where IN the world can you use it?

You use the IN operator when your criterion (the WHERE clause) uses a single field with multiple values--in other words, when you want to know whether the value of a single field falls within a set of values. In the previous example, the SQL statement tests to see if the EmpNo field contains the value 11, 12, or 114. Let's look at how the IN operator works and how it can help us with this SQL statement.

The IN implementation

The IN operator lets you specify a set of values to which you want to compare a field. The syntax for the IN operator is as follows:

```
SELECT fields
FROM table
WHERE field IN ('value', 'value', ...)
```

Now we can take the example SQL statement and rewrite it using an IN operator instead of multiple OR clauses. Simply drop the fields, table, and values into the IN syntax, as follows:

```
SELECT *
FROM Orders
WHERE EmpNo IN ('11', '12', '114')
```

This new statement performs the same query and looks better.

SELECT sets

In addition to using static values to create the set, you can use SELECT statements as well. In other words, instead of using the statement

```
WHERE EmpNo IN ('11', '12', '114')
```

you can use a SELECT statement in place of ('11','12','114'). You can't use just any SELECT statement, however. The SELECT statement must return a single-column (or field) table. If the statement returns more than one column, the database engine will become confused because it won't know which column to use.

For example, consider the following legal SELECT statement:

```
SELECT EmpNo
FROM Employee
WHERE Salary > 40000
```

This SELECT returns a single column that contains EmpNo values where Salary is greater than $40,000.

You can plug this statement into our previous example to get a list of employee orders for employees whose salary is more than $40,000. To do so, use the following statement:

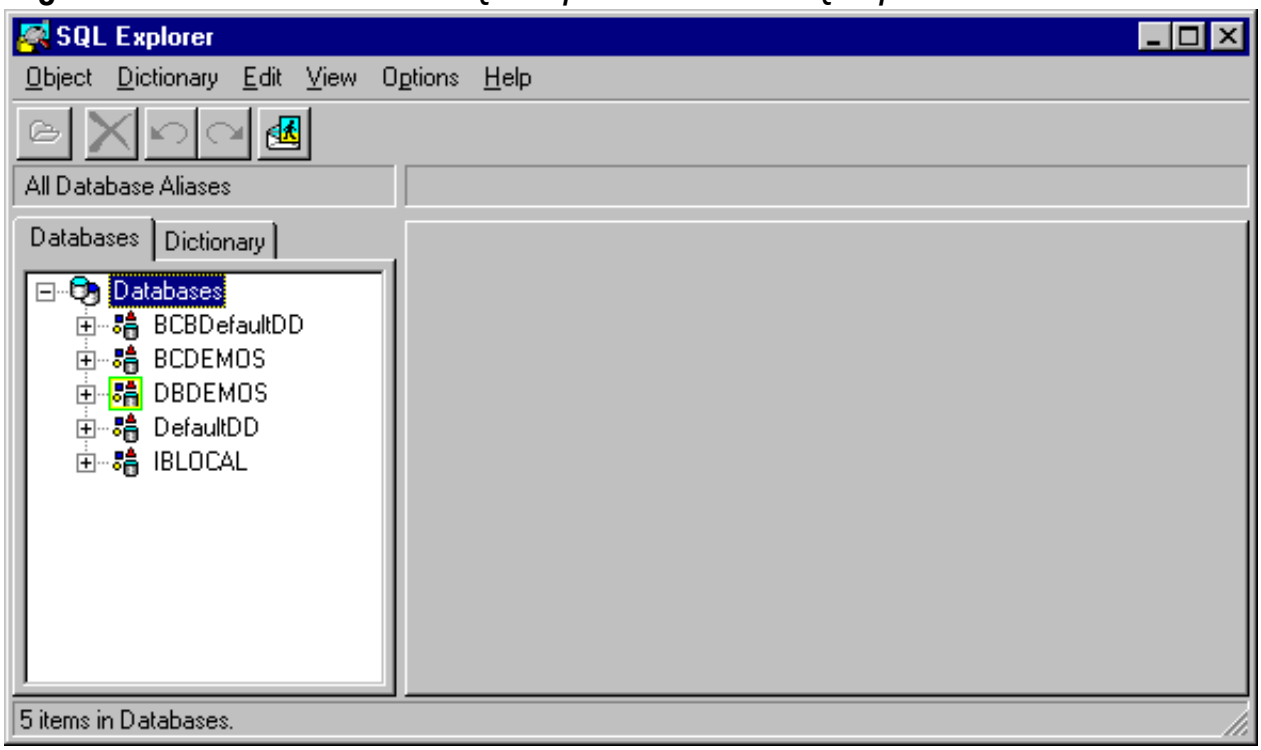
```
SELECT *
FROM Orders
WHERE EmpNo IN
( SELECT EmpNo
FROM Employee
WHERE Salary > 40000 )
```

It's important to note that the result set of the SELECT statement following IN must return a column whose data type is the same as the field that you're comparing. In other words, since EmpNo is an integer field, the SELECT statement must return a set of integers. If it returns a set of character fields, the database engine will respond with an error.

You can do it

You can try all of these examples in the C++Builder Database Explorer. Simply select Explore from the Database menu in C++Builder. The SQL Explorer will appear, as shown in **Figure A**. These examples use the DBDEMOS database that comes with all versions of C++Builder.

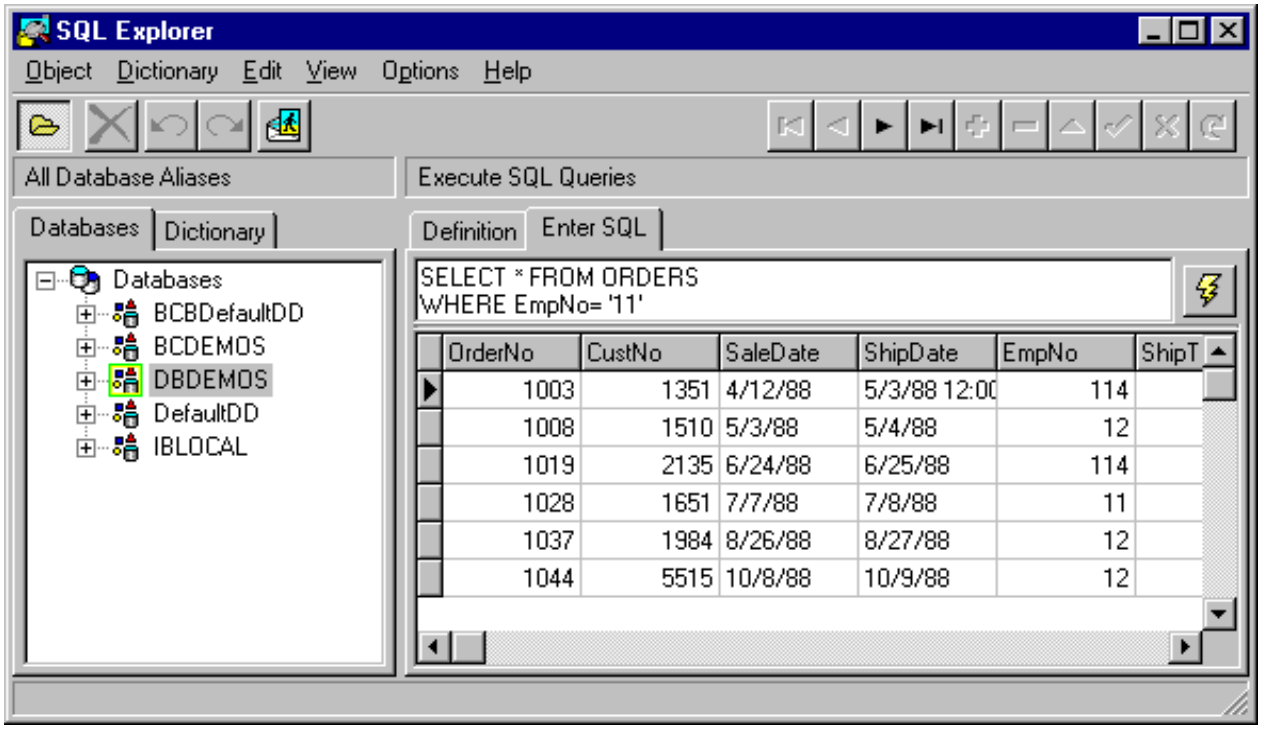
Figure A: *You can use the SQL Explorer to run SQL queries.*



Click on image to view full size.

Select the DBDEMOS database in the Databases tree. Next, select the Enter SQL tab in the right-hand window. Type any of our sample SQL statements in the edit window and click the lightning bolt button to run the query. The results will appear in the window below the edit window, as shown in **Figure B**.

Figure B: *C++Builder's SQL Explorer displays the SQL query results.*

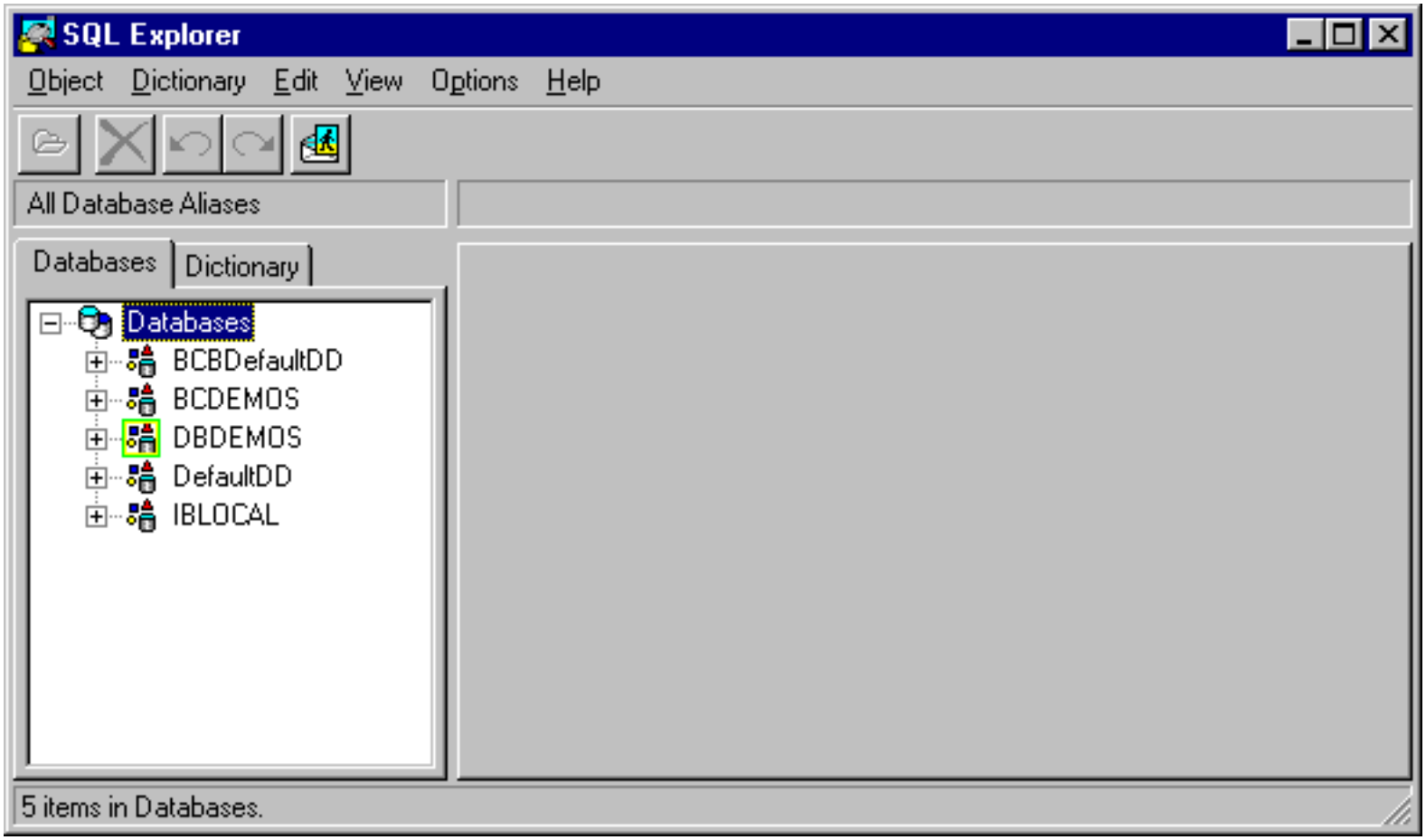


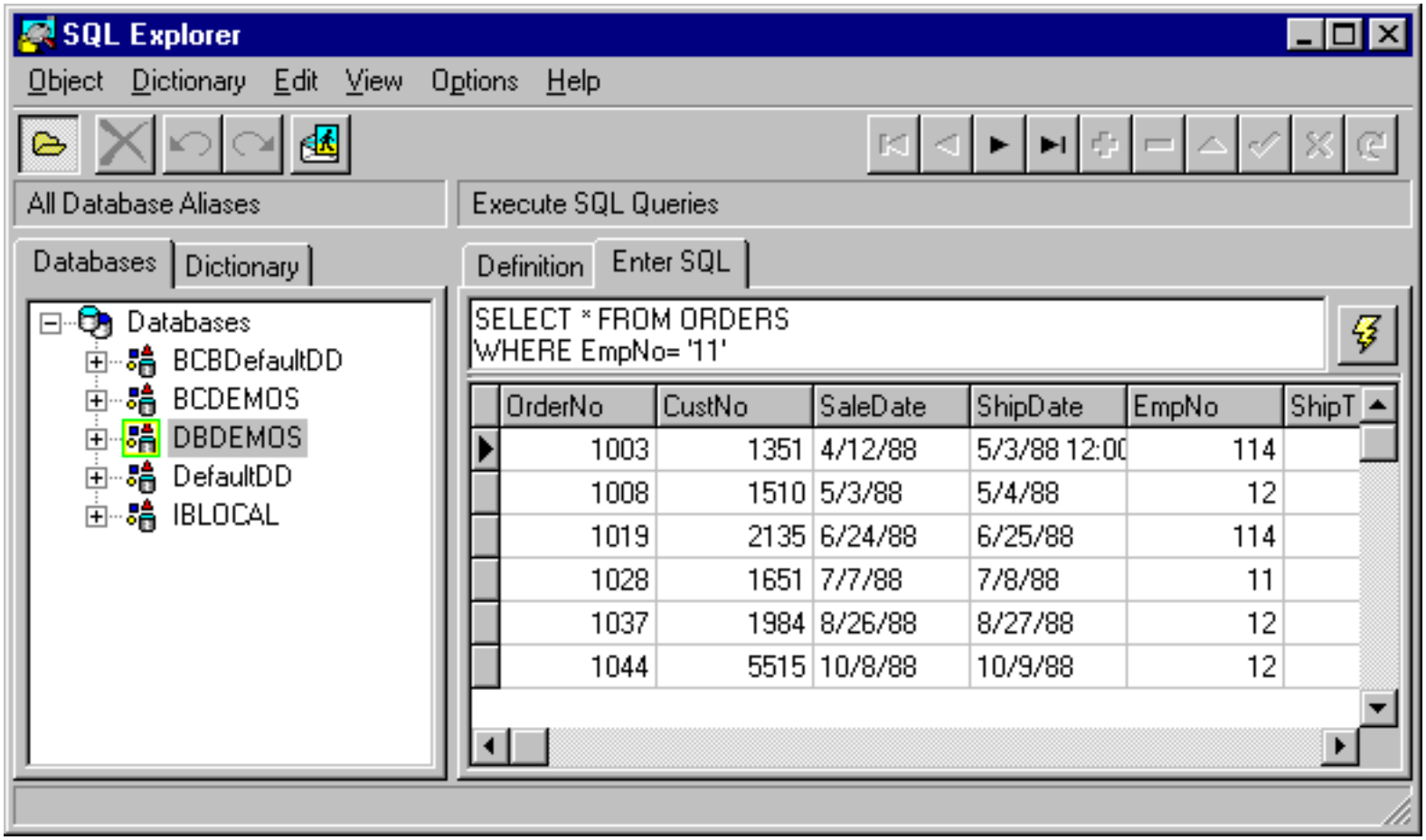
Click on image to view full size.

Conclusion

SQL provides a robust interface for retrieving data from databases. In this article, we've shown you how the IN operator can help you get better results from SQL.

Brian Schaffneris the former editor-in-chief of *Paradox for Windows Developer's Journal*.


SQL Explorer
Object Dictionary Edit View Options Help
All Database Aliases
Databases Dictionary
Databases
BCBDefaultDD
BCDEMOS
DBDEMOS
DefaultDD
IBLOCAL
5 items in Databases.


SQL Explorer
Object Dictionary Edit View Options Help
All Database Aliases
Execute SQL Queries
Databases Dictionary
Definition Enter SQL
Databases
BCBDefaultDD
BCDEMOS
DBDEMOS
DefaultDD
IBLOCAL
SELECT * FROM ORDERS
WHERE EmpNo= '11'
OrderNo CustNo SaleDate ShipDate EmpNo ShipT
1003 1351 4/12/88 5/3/88 12:00 114
1008 1510 5/3/88 5/4/88 12
1019 2135 6/24/88 6/25/88 114
1028 1651 7/7/88 7/8/88 11
1037 1984 8/26/88 8/27/88 12
1044 5515 10/8/88 10/9/88 12

SpeedButton tricks

by Tim Gooch

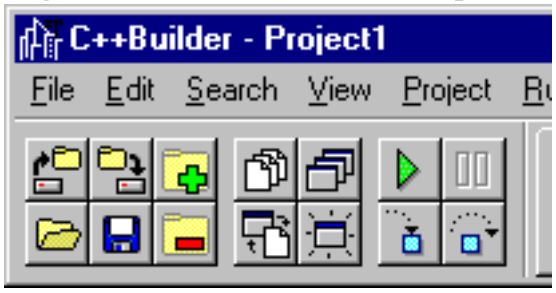
With its visual-design toolsand form-based orientation, C++Builder makes it easy for you to create highly interactive applications. The Visual Control Library (VCL), a critical element of C++Builder, lets you maintain a great degree of control over the look and feel of the resulting application. SpeedButton components are a perfect example of this phenomenon.

Commands that really click

If you create a form and place several buttons on it, they will behave in a way similar to the alphanumeric keys on your keyboard. When you click on a button it will stay down, but only as long as you continue to depress the mouse button.

The most common use for SpeedButton components configured this way is to place them on a toolbar on which each button executes a command. C++Builder uses SpeedButton components on the left side of the toolbar (the file, view, and debugging commands), as shown in **Figure A**.

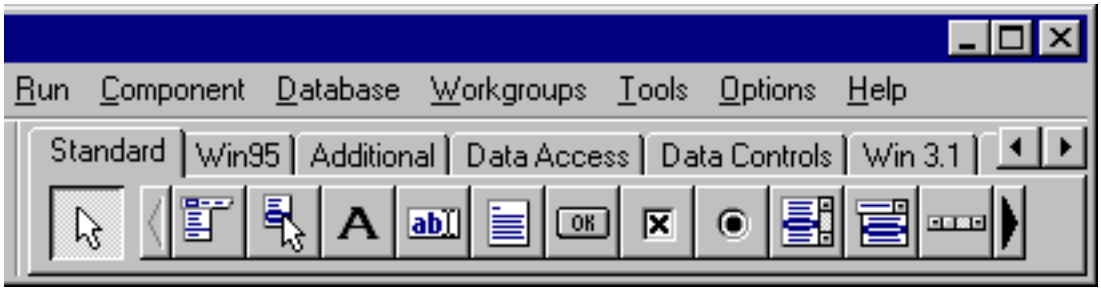
Figure A: *C++Builder uses SpeedButton components to implement the command buttons on the toolbar.*



Click 'em when they're up...

In some situations, you'll want to create a set of buttons that implement a mutually exclusive relationship. For example, on the right side of the C++Builder toolbar, you'll find the arrow icon and a set of icons comprising the Component Palette, shown in **Figure B**. If you select a component, the button for that component stays down until you place the component or select a new component (or click the arrow button).

Figure B: *The right side of the toolbar displays the Component Palette, which consists of several radio speedbuttons.*



In effect, these buttons, which we'll call *radio speedbuttons*, act in the same way as a set of radio buttons: One button must be down initially, but you can't choose more than one at a time. To implement radio speedbuttons with SpeedButton components, place several of them on a form, then change each one's GroupIndex property to the same value.

Unlike standard radio buttons, you can create more than one set of radio speedbuttons within a given form without having them interact. To do so, use a different GroupIndex value for each set of radio speedbuttons.

Click 'em when they're down

For most situations, the previous two styles will be sufficient. However, what if you want to prompt the user for one of a group of items, or none at all?

For example, how would you design a dialog box that asks users which of a set of optional, supplementary dictionaries to use? In other words, you want them to specify one of the supplementary dictionaries to use, but only one--or, if they prefer, no dictionary at all.

Using a set of SpeedButton components, you can define this type of button set, which we'll call *exclusive option buttons*. To do so, first set the AllowAllUp property to True for each button, then specify the same GroupIndex property for the set.

State of the button

The final variation we'll cover is the *state button*. This button behaves just like [Caps Lock] on your keyboard: Press it once to turn it on; press it again to turn it off.

To create a state button, first set its AllowAllUp property to True, then set its GroupIndex property to a unique value. As you create additional state buttons, you'll need to use a new value for the GroupIndex property each time to keep them from becoming exclusive option buttons.

Premiere Issue

Editing string lists in the code-editing window

by Tim Gooch

C++Builder provides many powerful user-interface features, most of them obvious. However, some features are more subtle. One such feature is the code-editor interface for string lists.

Suppose you need to modify a string list that contains database parameters, SQL query statements, list box items, or similar information. Typically, you'll double-click the appropriate property in the Object Inspector to display the String List Editor. Unfortunately, this is a modal window, and you must save your changes explicitly before you can return to the code-editing window to make further changes to your source code.

If you look closely at the String List Editor window, you'll notice a Code Editor... button at the bottom, as shown in **Figure A**. When you click this button, C++Builder will open a new page in the code-editing window and load the string list text into it, as shown in **Figure B**.

Figure A: *Clicking the Code Editor... button in the String List Editor window...*

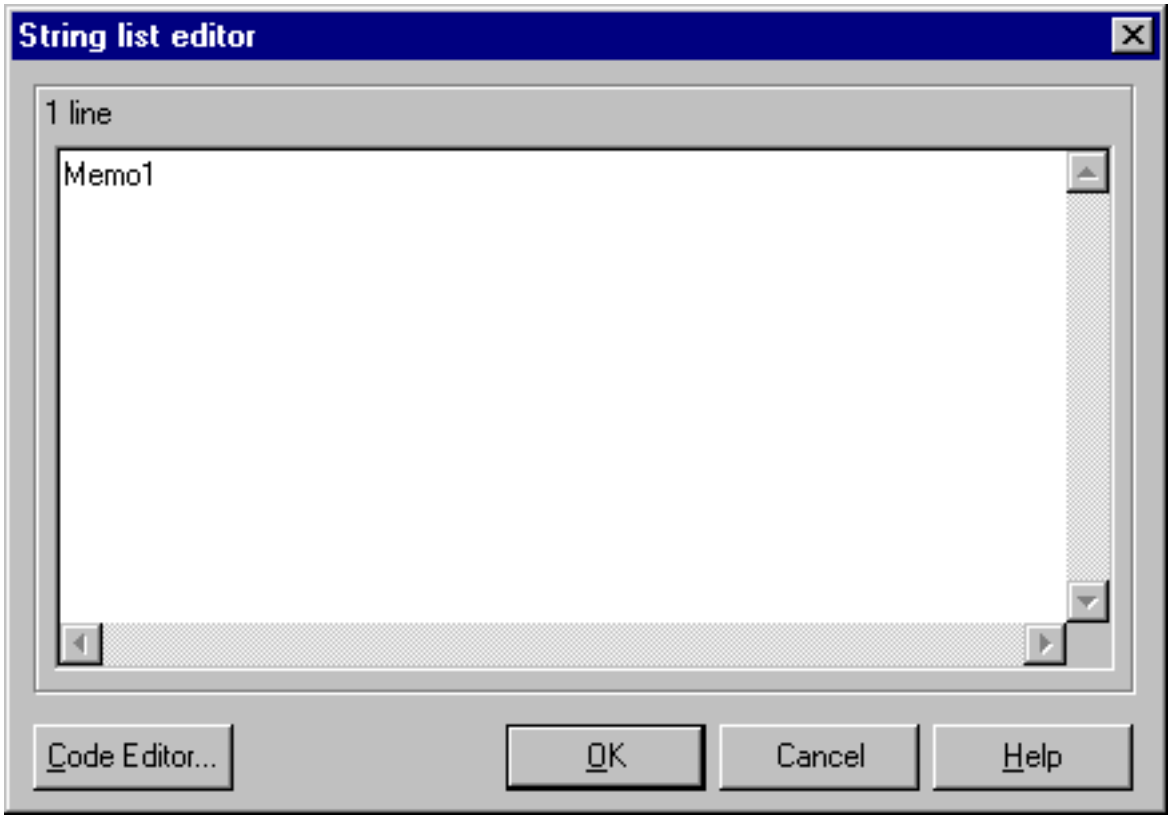
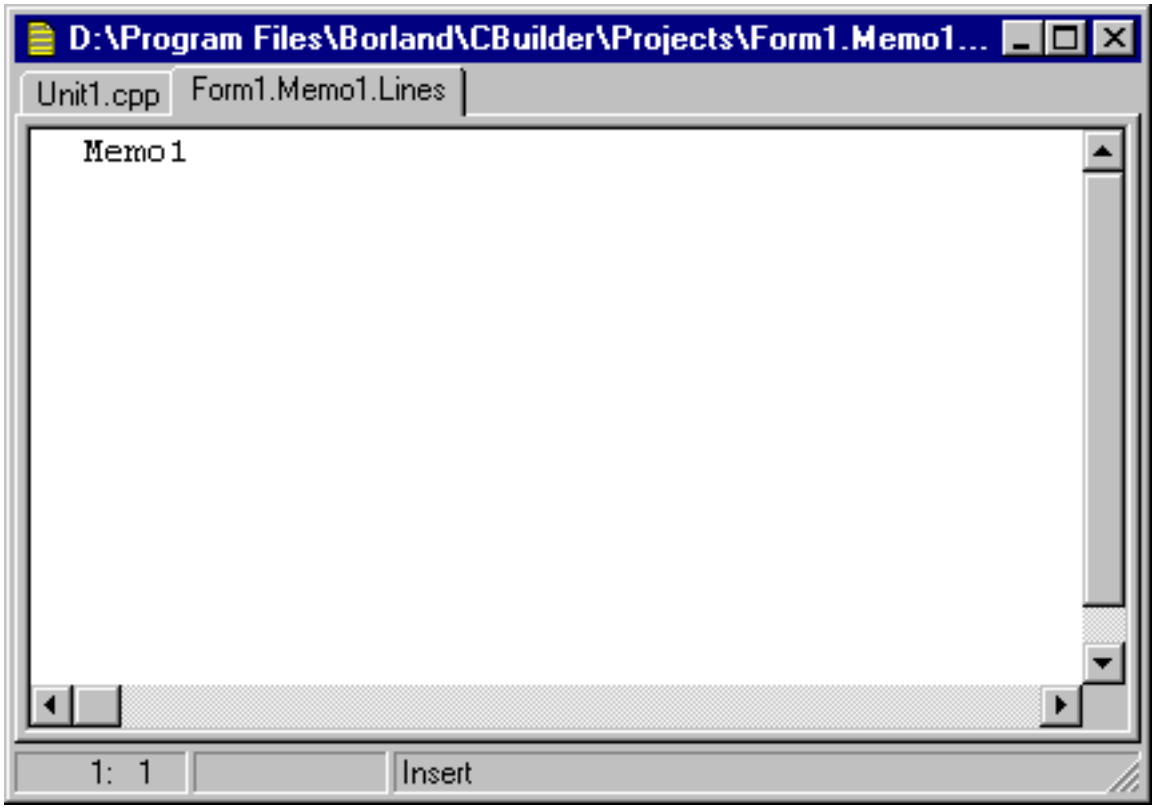


Figure B: *...places the string list properties in the code-editing window.*



Unlike in the String List Editor, you don't have to save the text you enter for a string list explicitly via the code-editing window. As soon as you move to a different page in the code-editing window or start building your project, C++Builder will save your text into the appropriate property of the component that contains the text.

We think you'll find modifying string lists using the code-editing window to be much faster than using the String List Editor. Plus, if you display a string list in the code-editing window, you can choose New Edit Window from the code-editing window's pop-up menu to create the best of both worlds--a separate window for editing string lists.