親手打造 C++Builder 的 TRACE Window

蕭永哲 martin@copystar.com.tw

曾經用過 Visual C++ 1.5 的讀者們都知道，你可以使用 TRACE 這個巨集將需要 Debug 的訊息送到一個名爲 DBWIN 的視窗上，而咱們可以一邊執行程式，一邊觀看 DBWIN 中由程式所輸出過來的 TRACE 訊息。但不幸的事，Visual C++ 1.5 的下一個版本 Visual C++ 4.X 竟然開始不包含了 DBWIN，爲什麼呢？原來是原先 DBWIN 裡頭所包含的所有功能都被整合到了 Visual C++的整合環境裡頭，且功能更被發揚光大，但 TRACE 巨集從現在起只存在於除錯模式，並且還被硬性規定只許應用程式在整合環境內執行時其 TRACE 功能才能夠使用。這對於只不過想要像使用 printf 一樣簡單的輸出一點訊息的咱們，卻必須動用到這樣龐大的除錯器似乎有點!@#$%^。

幸好在 Microsoft System Journal (MSJ) 上 Paul DiLascia 生生的兩篇文章爲我們打造了一個叫簡單 DBWIN 稱爲 TraceWin，稍微補足了這項小小的不快。對於使用 Visual C++ 4.x 的使用者來說，有如久旱逢甘霖一般。在當時筆者也跟著依樣畫葫蘆地自己打造了一個 TraceWin；但自從 1997 的白色情人節過後筆者與 C++Builder 結下了不解之緣，用 C++Builder 開發了許多專案，但總覺得缺少了一點什麼。對！就是 C++Builder 專用的 TraceWin。C++Builder 的除錯環境很難使用嗎？一點也不！功能強大的令人瞠目結舌；但我只是希望能夠簡單的檢查程式的流程、資料的內容，並不需要動用到這麼龐大的除錯環境。我要的只是希望能把 Debug 的動作弄得簡單點。

以往在 DOS 的文字模式下，printf 是個簡單又好用的除錯幫手，可以在需要檢查的位置適時地加入 printf 將除錯資訊輸出到螢幕上。而在 Windows Programming 中，printf 或許還能夠在 Consol Mode 裡佔據從前保有的江山，但在圖形界面裡 printf 已毫無用武之地，簡單的 Debug 動作可以由訊息對話盒（Message Box）來代替龐大的除錯環境，但使用 Message Box 會打斷原有程式的執行並且將原始視窗被遮蓋住，最討厭的是還得去按個 OK 才能讓程式繼續執行，並且在按下 OK 後原始視窗還收到一個不速之客：不必要的 WM_PAINT 訊息，這讓我們的程式多做了一個重繪的動作，訊息對話盒似乎是個很不好的選擇，但在沒有 TraceWin 之前，這是唯一也較簡單的選擇了，有多簡單呢？在 C++Builder 下只要使用 ShowMessage() 這個函示，就可產生一個文字訊息視窗。

■ 不同行程間的資料傳送

先想一想，咱們得把我們所要檢查的資料給送到另一個行程中的『除錯視窗』裡，那們我們何法子可以辦得到呢？對！就是行程間的通訊（Interprocess Communication，IPC）。但行程通訊的方法種類繁多，光是『行程通訊』這個主題就夠筆者寫上數萬字，該選擇哪一種來用都是個難題，所幸在眾多 Windows Programming 書籍裡都找得到討論行程通訊這主題的章節[1]，在此筆者就不多嘴，咱們就挑個背後技術不難且實作起來較容易的方法來用用，對！就是使用視窗訊息來傳送資料：

由以往程式寫作經驗告訴我們：要將較大量的資料在程式內部間傳遞，最方便的方法就是使用指標的傳遞來達成；但在 Win32 平台上不同行程間的定址空間各自獨立的，因此要藉由指標把資料傳送給另外一個行程在Win32平台上是不被允許但又是不可或缺的，想當然耳Microsoft當然會注意到這個問題，因此 Microsoft 提供了一個 WM_COPYDATA 這個視窗訊息來替個各行

程間透過指標來將傳遞資料，使用 WM_COPYDATA 這個視窗訊息時系統會自動幫我們把指標所指到的資料由發出端的行程定址空間內複製到接收端的行程定址空間裡頭，並將複製後的資料指標通知給接收端知道。因此使用這個方法就沒有必要去考慮到各個行程間的定址空間的問題。此外使用 WM_COPYDATA 是在不同執行緒（不論這兩個執行緒是否爲同一行程內）之間搬移資料最簡單的一種做法了，不過要接收這個訊息的目的地必須擁有一個視窗代碼（Window Handle)，也就是必須是個視窗，爲什麼呢？沒爲什麼，因爲你必須傳遞訊息的函式 SendMessage 的目的地必須是個視窗，SendMessage 的第一個參數就是接收訊息端的視窗代碼。

WM_COPYDATA 訊息的使用如下（發出訊息端）：

```
SendMessage(
            hwndReceiver,
            WM_COPYDATA,
            (WPARAM)hwndSender,
            (LPARAM)&cds
            );
```

hwndReceiver：接收訊息端的視窗代碼

hwndSender：發出訊息端的視窗代碼

cds：需要傳送的資料的指標

其中 LPARAM 參數中的 cds 指向一個特定的 Windows 資料結構：COPYDATASTRUCT，其結構如下：

```
typedef struct tagCOPYDATASTRUCT {  // cds
     DWORD dwData;
     DWORD cbData;
     PVOID lpData;
} COPYDATASTRUCT,*PCOPYDATASTRUCT;
```

dwData：爲使用者的自定值，使用者可以透過此值來通知接收端其傳送的資料類別以及傳送的資料用途。

cbData： lpData 所指到的資料大小，以 byte 爲單位。

LpData： 所要傳送資料的起始指標。資料的內容可以是任何的資料。

■ 取得接收端的視窗代碼

大約知道了怎麼把 WM_COPYDATA 訊息送出後，咱們再來要考慮的就是接收端如何收到訊息的問題。不過在此之前，在訊息的發出端咱們還得知道接收端的視窗代碼，因爲若沒有接收端的視窗代碼就好比郵差送信不知道收件人地址一般，當然是送不出去啦。

筆者最常用來取得接收端的視窗代碼的方法有兩種：

1. 使用 FindWindow

FindWindow 是 Windows API，FindWindow 的原始定義爲：

```
HWND FindWindow(
          LPCTSTR lpClassName,  // pointer to class name
          LPCTSTR lpWindowName  // pointer to window name
         );
```

lpClassName ：爲視窗在建立時所註冊的視窗類別

lpWindowName ：爲視窗的名稱，通常是視窗的標題(Caption)

由這個 API 的定義就不難看出，只要我們知道接收端的視窗標題或是知道接收端註冊的視窗類別，我們就有辦法把訊息給傳送出去。但若有相同的兩個視窗先後被執行時怎麼辦呢？FindWindow 會找到先執行的那個視窗，並取回其視窗代碼。

所以，只要接收端的視窗標題不會改變，那麼我們就可以使用 FindWindow 來取得接收端

的視窗代碼。

2. 註冊視窗訊息與廣播訊息（Register Window Message and Broadcast Message）

使用『註冊視窗訊息與廣播訊息』是在於你不想用 FindWindow 的狀況下使用的。爲什麼會不想要用 FindWindow 呢？就是先前才提到的：接收端的視窗標題會改變，若你寫的程式須常常變換視窗標題，那麼在使用 FindWindow 時不就很難處理嗎？還必須知道接收端的視窗標題改變成什麼。

因此可換成使用註冊視窗訊息與廣播訊息的方式來取得接收端的視窗代碼：在程式第一次執行時，發出端透過 SendMessage 將一個已經註冊好的視窗訊息給廣播出去[2]，當接收端收到這個已經註冊過的特定訊息時，會回傳另一個自訂的訊息給廣播訊息的發出端，而且這個回傳的訊息裡還帶有接收端的視窗代碼，如此一來就發出端就可以掌握住接收端的視窗代碼，這樣一來，即使接收端的視窗標題常常在變換，對發出端來說要將訊息傳送給指定的接收端也不是什麼難事了。但使用這種方法的前提是在於在發出訊息端程式執行前接收訊息端就必須得先執行了，不然沒有辦法傳回接收端的視窗代碼。當然了，若你要每次發送訊息前來做視窗代碼的傳遞也無不可，只是這樣一來一往會浪費掉許多訊息傳送的時間。

上述這兩個方法都很不錯，但使用註冊視窗訊息與廣播訊息一來一往傳遞訊息次數過於頻繁，且所撰寫需程式碼也較爲多些，筆者暫時不考慮使用。

■ 接收端接收訊息

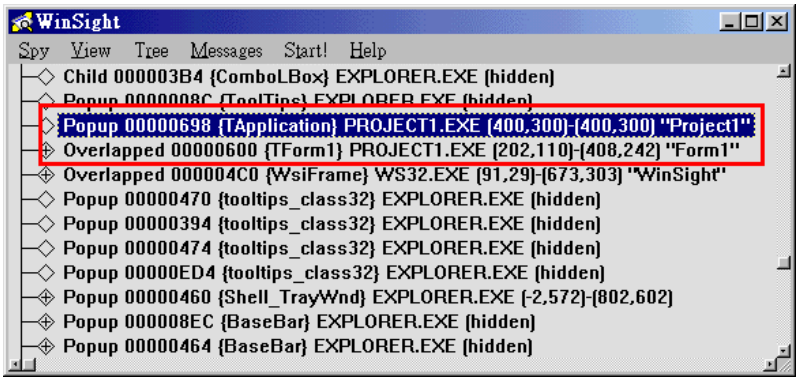
接收端在接收訊息這個部份處理流程較爲簡單，就是確認收到 WM_COPYDATA 訊息，並且將這個訊息依照內容做其該做的處理。這個主題曾在 53 期周先生的『BCB 與 Win32 訊息傳遞』一文裡已被討論到，若對這個主題尚有不了解的，筆者建議讀者們先把 53 期的那篇文章給先看過。但筆者在此將會對這個主題做更深入的探討，雖說接收訊的流程乍看之下很簡單，但在 C++Builder 的 VCL 龐大架構下，視窗訊息的處理程序已經經過 VCL 架構的層層包裝，在 VCL 的背後隱藏了許多高超的技巧，但對一般使用者來說，只不過是看到了其神祕面紗之外的形象，就讓筆者揭開這神祕面紗一窺究竟。

在 C++Builder 下，處理視窗訊息的方式有很多種，我們要選擇哪一種呢？都可以！你只要確定你能夠收到這個訊息並且正確的處理他就可以了，你可以送給這個應用程式的主控制視窗，也可以送到應用程式的主視窗。主控制視窗？主視窗？都快搞糊塗了，這到底怎麼說呢？說來話長，這得從 C++Builder 所開發出來的應用程式的啓動流程談起。

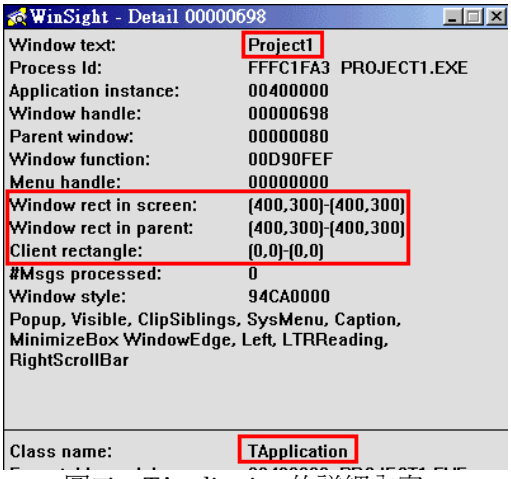
■ C++Builder 應用程式啓動流程

先讓咱們用 C++Builder 建立一個新的應用程式（使用選單上【File】\【New Application】），Project 與 Form 的名稱都保持預設值：Project 名稱爲 Project1，而那個上面什麼物件都沒有的 Form 爲 Form1。在編譯並執行後讓人直覺性的認爲這個 Form1 應該就是應用程式的主視窗，並且應該也掌控所有發出給此應用程式的訊息。其實不然，在 Form1 背後的黑手令有其人，那到底是誰呢？讓我們先用 C++Builder 所附上的 WinSight—這個好用的工具來觀察我們所執行的 Project1.exe（圖一），由圖一可以發現同一個 Project.exe 裡竟然會有兩個 Window Control 的出現，一個叫做 TApplication，而另一個叫做 TForm1。那到底哪個才是這個應用程式的主要幕後黑手呢？是 TApplication 還是 TForm1？

再利用 WinSight 先來看看 TApplication 的更深入的內容（圖二）， 由圖中可以看到 TApplication 的視窗大小是零，而其 Windows text（視窗標題）是 Project1，其 Class name（視



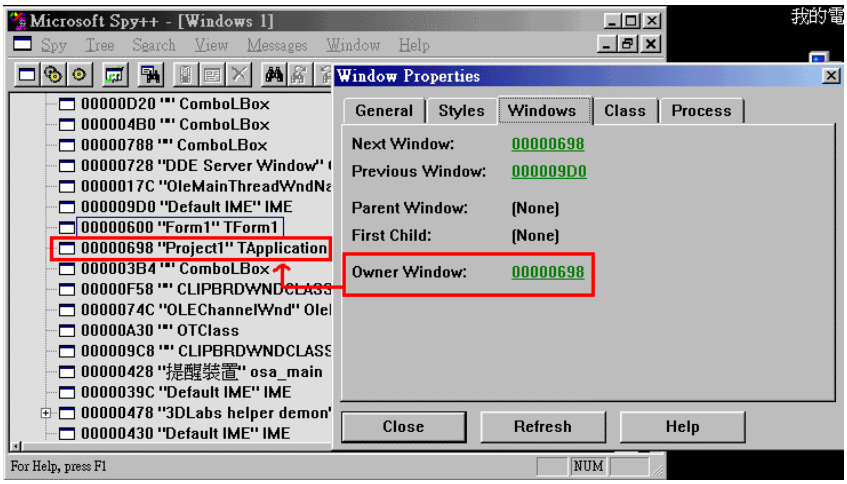
圖一、使用 WinSight 來觀察所建立的應用程式



圖二、TApplication 的詳細內容

窗類別）是 TApplication；視窗大小是『零』，也就是說這個 Window 是看不見的，TApplication 的隱形頗有躲在背後的意義。看起來似乎這 TApplication 似乎就是幕後的黑手，先聲明只是看起來而已，稍後會想辦法驗證的。而這裡所提到的『視窗標題』與『視窗類別』其實就是先前在提到 FindWindow 這個 API 所使用的 lpWindowName 與 lpClassName 了！但若使用 FindWindow 來傳送訊息給現在我們觀察的這個應用程式，我們到底應該把訊息送給誰呢？是送給 TApplication 還是送給 TForm1 呢？容後在述。

先看看是否真的如筆者所說 TForm1 真的是由 TApplication 所掌管呢？可惜 C++Builder 所



圖三、使用 Visual C++附上的 Spy++來觀察

附上的 WinSight 並沒有辦法查出 TForm1 的管理者也就是擁有者是那個視窗，所以我們無法由 WinSight 看出任何的端倪，不過幸好使用筆者使用 Visual C++所附上與 WinSight 功能相似的 Spy++倒是可以讓我們瞧見所要的證據（圖三）：

由圖三很清楚的可以看出，TForm1 的 Owner Window 就是隱藏起來的 TApplication。由以上所蒐集的資料來看 TApplication 類別，似乎 TApplication 類別是由 C++Builder 所建立的應用程式中的大總管，這一切也在 Project1 的 Project1.cpp 裡透露出這祕密：

程式列表一:、Project1.cpp：

```
#0001.  #include <vcl.h>
#0002.  #pragma hdrstop
#0003.  USERES("Project1.res");
#0004.  USEFORM("Unit1.cpp", Form1);
#0005.  //---------------------------------------------------------------
#0006.  WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0007.  {
#0008.     try
#0009.     {
#0010.           Application->Initialize();
#0011.           Application->CreateForm(__classid(TForm1), &Form1);
#0012.           Application->Run();
#0013.     }
#0014.     catch (Exception &exception)
#0015.     {
#0016.           Application->ShowException(&exception);
#0017.     }
#0018.     return 0;
#0019.  }
#0020.  //---------------------------------------------------------------
```

此處看到的 Application 就是 TApplication 類別的實體物件，並且是個全域物件。但由這幾行程式碼來看爲何 C++Builder 所建立的應用程式如此神奇，只需由 0011 這列到 0013 這列就可以讓程式持續的執行了？其實在這三列程式的背後可是有數千列的幕後黑手在撐著，且讓筆者先把這三行的動作內容給解釋一下：

`#0011.  Application->Initialize();`
進行 `Application` 物件的初始化，如 `COM` 與 `OLE Automation` 的初始化

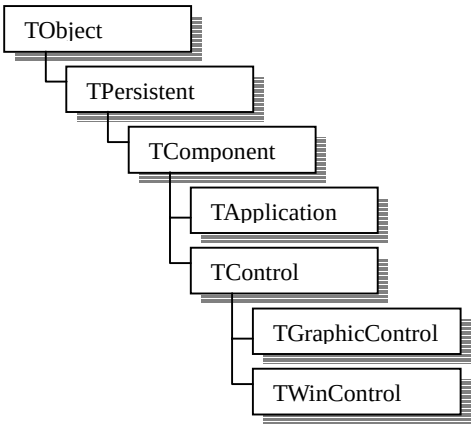
`#0012.  Application->CreateForm(__classid(TForm1), &Form1);`
開始建立主視窗，也就是 `Form1`，並顯示 `Form1`

`#0013.  Application->Run();`
進入訊息分派迴圈裡，一直到應用程式收到 `WM_QUIT` 訊息後才跳出訊息分派迴圈並結束應用程式。

喔！原來整個由 C++Builder 所建立的應用程式所收到的視窗訊息，都會由 Application 這全域物件來接收，並且透過訊息在 VCL 架構裡的流動把訊息分配到該分配的地方去，但僅止於他認得的訊息。爲什麼說僅止於他認得的呢？這得討論到 VCL 的內部架構，且聽筆者慢慢說下去。

■ VCL 對訊息的處理

暫且回到先前的問題，要把訊息從發出訊息端送給誰呢？是送給 Application 還是送給 Form1 呢？先前已經將答案給透露了，都可以啦！但是處理的方法有所不同罷了。在討論這個問題之前，先看看這一張簡化的 VCL 類別繼承架構圖（圖四），讓我們稍稍的了解一下 VCL 類別之間的關係。稍後會使用到這些關係：



圖四、簡化的 VCL 類別繼承架構

接下來就讓我們討論討論在 C++Builder 下接收視窗訊息的做法。在 C++Builder 下有很多做法，咱們一一來討論：

1、覆載 Dispatch 虛擬函式：（使用 Message Map）

在 53 期的『BCB 與 Win32 訊息傳遞』一文內曾用到以下的做法來接收並處理訊息：

```
BEGIN_MESSAGE_MAP
    VCL_MESSAGE_HANDLER(……)
END_MESSAGE_MAP(…)
```

但這個機制到底是怎麼做成的呢？咱們把他從 sysdef.h 裡給挖出來瞧一瞧：

程式列表二、sysdefs.h：

```
#601.   #define BEGIN_MESSAGE_MAP   virtual void __fastcall Dispatch(void *Message) \
#602.   {                                   \
#603.     switch  (((PMessage)Message)->Msg)      \
#604.     {
#605.
#606.   #define VCL_MESSAGE_HANDLER(msg,type,meth)        \
#607.           case   msg:                          \
#608.           meth(*((type *)Message));             \
#609.           break;
#610.
#611.   // NOTE: ATL defines a MESSAGE_HANDLER macro which conflicts with VCL's macro. The
#612.   //      VCL macro has been renamed to VCL_MESSAGE_HANDLER. If you are not using ATL,
#613.   //      MESSAGE_HANDLER is defined as in previous versions of BCB.
#614.   //
#615.   #if !defined(USING_ATL) && !defined(USING_ATLVCL) && !defined(INC_ATL_HEADERS)
#616.   #define MESSAGE_HANDLER  VCL_MESSAGE_HANDLER
#617.   #endif // ATL_COMPAT
#618.
#619.   #define END_MESSAGE_MAP(base)         default:   \
#620.           base::Dispatch(Message);   \
#621.           break;                    \
#622.           }                                \
#623.     }
```

在 sysdefs.h 裡定義了這三個巨集，讓我們來看一看這三個巨集到底在定義些什麼？

A、**BEGIN_MESSAGE_MAP**：（601 行至 603 行）

這個巨集在類別的定義中以虛擬的方式重新覆載了 Dispatch 函式，而 Dispatch 的原始定義到底是什麼呢？Dispatch 原以虛擬的方式定義在 TObject 裡，而其目地是在於視窗訊息的分配。而緊接著在這個虛擬函示宣告之後的是一個 switch 敘述，用來分派視窗訊息交給之後的

VCL_MESSAGE_HANDLER 巨集來處理。

B、**VCL_MESSAGE_HANDLER**：（606 行至 617 行）

先看到 615 至 617 這幾行程式，你可以發現，若沒有使用 ATL（Active Template Libray），MESSAGE_HANDLER 就自動被定義成 VCL_MESSAGE_HANDLER，但筆者在這裡還是建議你，多打幾個字會比較保險些，以免以後在移植程式時因爲這小小的疏忽而造成程式的錯誤，而一時無法察覺。

再看到 607 至 609 行，可以發現是直接在 BEGIN_MESSAGE_MAP 的 switch 敘述之後加入 case 進入點來處理指定視窗訊息的。舉個例子來說，若我們使用 VCL_MESSAGE_HANDLER 這個巨集如下：

```
VCL_MESSAGE_HANDLER(WM_COPYDATA,TMessage,OnCopyData)
```

則會會被展開成：

```
case WM_COPYDATA:
     OnCopyData(*((TMessage *) Message));
     break;
```

C、**END_MESSAGE_MAP**：（619 行至 623 行）

想當然耳，END_MESSAGE_MAP 巨集的用途應該就是來終結 switch 敘述，但你可以發現 620 行上頭的 base::Dispatch(Message)是用來將不認得的訊息交給父類別的 Dispatch 虛擬函式做處理。

因此所以若我們加入以下的範例：

```
BEGIN_MESSAGE_MAP
    VCL_MESSAGE_HANDLER(WM_COPYDATA,TMessage,OnCopyData)
END_MESSAGE_MAP(TForm);
```

則會被編譯成：

```
virtual void __fastcall Dispatch(void *Message)
{
    switch  (((PMessage)Message)->Msg)
    {
         case WM_COPYDATA:
              OnCopyData(*((TMessage *) Message)); //處理 WM_COPYDATA 的函式
              break;
         default:
              TForm::Dispatch(Message);
              break;
    }
}
```

Dispatch 的原始定義是在於 TObject 這個基礎類別上，只要是 TObject 的衍生類別都可以覆載這個虛擬函式來處理自己特定的訊息，但若 Dispatch 不認得這個送來的訊息最後會把處理程序丟改 TObject::DefaultHandler 這個函式處理，不過 TObject::DefaultHandler 並未實作任何的動作，真正的功能還必須透過以後的覆載來達成。挖 VCL 的原始碼很有趣吧！若還覺得光聽筆者這樣說說還不夠有趣的話，可以自行挖掘這段 VCL 原始碼；在此要先說明一下，若讀者們購買的是 C++Builder 專業版或更高級的版本，光碟片裡頭都會含有 VCL 的原始碼，但若是標準版就沒有附上原始始碼了。有志挖掘 VCL 架構的內部奧祕者都可以自己來探求 VCL 背後的真面目。OK！TObject::Dispatch 的原始碼你可以在：source\vcl\system.pas 裡頭找到，有興趣的讀者可以把這段程式給拿出來看，不過先透露一下，爲了加速訊息的傳遞，Borland 的 VCL 開發小組把 TObject::Dispatch 函式都採用組合語言來撰寫，已達加速處理的目的。

但僅只有覆載 Dispatch 這一種方法嗎？不還有很多呢！接著看下去。

2、覆載 WndProc 函式

剛才談過 TObject::Dispatch 的做法，再來談談 TControl 裡頭對於訊息的處理，在 TControl 裡頭處理訊息有兩個一個是由 TObject 衍生而來的 Dispatch，但在 TControl 裡頭並沒有複寫

Dispatch 函式，依舊採用 TObject 裡頭的 Dispatch 函式，而另一個方式就是 WndProc 函式，先來瞧瞧 TControl 類別裡頭的 WndProc 做些什麼：

程式列表三、controls.pas：

```
#2414.  procedure TControl.WndProc(var Message: TMessage);
#2415.  var
#2416.    Form: TCustomForm;
#2417.  begin
#2418.    if (csDesigning in ComponentState) then
#2419.    begin
#2420.      Form := GetParentForm(Self);
#2421.      if (Form <> nil) and (Form.Designer <> nil) and
#2422.        Form.Designer.IsDesignMsg(Self, Message) then Exit;
#2423.    end;
#2424.    if (Message.Msg >= WM_KEYFIRST) and (Message.Msg <= WM_KEYLAST) then
#2425.    begin
#2426.      Form := GetParentForm(Self);
#2427.      if (Form <> nil) and Form.WantChildKey(Self, Message) then Exit;
#2428.    end;
#2429.    if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
#2430.    begin
#2431.    if not (csDoubleClicks in ControlStyle) then
#2432.      case Message.Msg of
#2433.        WM_LBUTTONDBLCLK, WM_RBUTTONDBLCLK, WM_MBUTTONDBLCLK:
#2434.          Dec(Message.Msg, WM_LBUTTONDBLCLK - WM_LBUTTONDOWN);
#2435.      end;
#2436.      case Message.Msg of
#2437.        WM_MOUSEMOVE: Application.HintMouseMessage(Self, Message);
#2438.          WM_LBUTTONDOWN, WM_LBUTTONDBLCLK:
#2439.        begin
#2440.          if FDragMode = dmAutomatic then
#2441.          begin
#2442.            BeginDrag(True);
#2443.            Exit;
#2444.          end;
#2445.          Include(FControlState, csLButtonDown);
#2446.        end;
#2447.        WM_LBUTTONUP:
#2448.          Exclude(FControlState, csLButtonDown);
#2449.      end;
#2450.    end;
#2451.    Dispatch(Message);
#2452.  end;
```

由 2429 至 2451 行程式中可以看得到，TControl 類別僅處理的與滑鼠動作相關的視窗訊息，如：WM_MOUSEMOVE、WM_LBUTTONDOWN 這一類的訊息，而其他的訊息就丟給 Dispatch() 處理。從 VCL 的原始碼可以得知 TControl 類別是沒有包含視窗代碼的。怪怪！TControl 類別不是沒有視窗代碼嗎？怎麼可以收得這些必須有視窗代碼才收得到的到滑鼠訊息呢？別急，先接著看到 TControl 的衍生類別：TWinControl 類別，打從 TWinControl 類別開始才包含了視窗代碼，先來看看 TWinControl 類別中對於 WndProc 函式的覆載是怎麼寫的：

```
#3371.  procedure TWinControl.WndProc(var Message: TMessage);
#3372.  var
#3373.    Form: TCustomForm;
#3374.  begin
#3375.    case Message.Msg of
#3376.      WM_SETFOCUS:
#3377.      begin
#3378.        Form := GetParentForm(Self);
#3379.        if (Form <> nil) and not Form.SetFocusedControl(Self) then Exit;
#3380.      end;
#3381.      WM_KILLFOCUS:
#3382.        if csFocusing in ControlState then Exit;
#3383.      WM_NCHITTEST:
#3384.      begin
#3385.        inherited WndProc(Message);
#3386.        if (Message.Result = HTTRANSPARENT) and
#3387.           (ControlAtPos(ScreenToClient(SmallPointToPoint(
```

```
#3388.              TWMNCHitTest(Message).Pos)), False) <> nil) then
#3389.            Message.Result := HTCLIENT;
#3390.          Exit;
#3391.        end;
#3392.        WM_MOUSEFIRST..WM_MOUSELAST:
#3393.          if IsControlMouseMsg(TWMMouse(Message)) then Exit;
#3394.        WM_KEYFIRST..WM_KEYLAST:
#3395.          if Dragging then Exit;
#3396.        WM_CANCELMODE:
#3397.          if (GetCapture = Handle) and (CaptureControl <> nil) and
#3398.           (CaptureControl.Parent = Self) then
#3399.          CaptureControl.Perform(WM_CANCELMODE, 0, 0);
#3400.    end;
#3401.    inherited WndProc(Message);
#3402.  end;
```

可以由 3392 至 3393 行上看到：滑鼠訊息的處理部份交由 TWinControl::IsControlMouseMsg() 函式處理，而在 3401 行中 TWinControl 類別最後把不認得的訊息交給父類別 TControl 的 WndProc 做處理，TControl::WndProc 先前已經瞧過了，但這不是我們想要知道的重點，現在我們想知道的是爲什麼 TControl 類別可以收到由 TWinControl 才收得到的滑鼠訊息呢？讓我們追下去看 TWinControl::IsControlMouseMsg()：

```
#3349.  function TWinControl.IsControlMouseMsg(var Message: TWMMouse): Boolean;
#3350.  var
#3351.    Control: TControl;
#3352.    P: TPoint;
#3353.  begin
#3354.    if GetCapture = Handle then
#3355.    begin
#3356.      Control := nil;
#3357.      if (CaptureControl <> nil) and (CaptureControl.Parent = Self) then
#3358.        Control := CaptureControl;
#3359.      end else
#3360.        Control := ControlAtPos(SmallPointToPoint(Message.Pos), False);
#3361.      Result := False;
#3362.      if Control <> nil then
#3363.      begin
#3364.        P.X := Message.XPos - Control.Left;
#3365.        P.Y := Message.YPos - Control.Top;
#3366.        Control.Perform(Message.Msg,Message.Keys,Longint(
#3367.        PointToSmallPoint(P)));
#3368.        Result := True;
#3369.      end;
#3370.  end;
```

哦！原來 TControl::WndProc 所收到的滑鼠訊息是由 TWinControl::WndProc 裡頭的 IsControlMouseMsg()函式所傳遞出來的。但是由圖四來看，TWinControl 不是 TControl 的衍生類別嗎？怎麼可以…？可以的，在 TControl 類別上有一個很重要且必要的屬性，就是 TControl 類別必須是由 TWinControl 管理，正確的說：TControl 類別以或 TControl 的衍生類別的 Owner 必須是 TWinControl 或 TWinControl 的衍生類別；什麼是 Owner？Owner 可以稱爲『物件擁有者』，是打從 TComponent 類別開始有的，物件的 Owner 負責物件的管理以及訊息的分派。

就好比 TGraphicControl 的衍生類別：TImage 類別，當我們把 TImage 元件被我們置放到 TWinControl 的衍生類別 TForm1 上時，TForm1 就是 TImage 的 Owner，而 Owner 把該送給其所擁有的子物件的所有訊息給發送出去，如此一來這個 TImage 就能夠擁有處理滑鼠訊息的能力了。

由以上可知，若要在接收端接收訊息，除了覆載 Dispatch 函式外，也可以在接收端重新覆載 WndProc 這個函式來達到訊息的處理目的。方法如下：

```
#0001.  void __fastcall TForm1::WndProc(TMessage &Message)
#0002.  {
#0003.    switch(Message.Msg)
#0004.    {
#0005.         case WM_COPYDATA:
```

```
#0006.              OnCopyData(Message);//處理 WM_COPYDATA 的函式
#0007.              break;
#0008.    }
#0009.    TForm::WndProc(Message);
#0010.  }
```

這樣的做法與覆載 Dispatch()方法相似，但實際上還是有所不同，哪裡不同呢？由程式列表三的 2451 行上可以看到，在處理的程序上是先先由 WndProc 來作訊息的處理，若不認得訊息最後才把訊息丟給 Dispatch 處理，因此若使用覆載 WndProc 函式的方式來處理訊息會比覆載 Dispatch 函式早一點點收到並處理訊息。還有沒有比覆載 WndProc 函式方法更早攔截得到訊息的方法呢？有的！接下來咱們來看看 VCL 架構裡的訊息接收先鋒。

3、Application OnMessage

先前說過，Application 物件是整個應用程式的幕後黑手，由程式列表一的 13 行開始進入整個應用程式的訊息迴圈，一直到收到 WM_QUIT 這個訊息才跳出迴圈並結束程式的執行。

先來看看 TApplication::Run 到底是怎麼讓程式進入訊息迴圈：

程式列表四、forms.pas

```
#4903.  procedure TApplication.Run;
#4904.  begin
#4905.    FRunning := True;
#4906.    try
#4907.    AddExitProc(DoneApplication);
#4908.    if FMainForm <> nil then
#4909.    begin
#4910.         case CmdShow of
#4911.              SW_SHOWMINNOACTIVE: FMainForm.FWindowState := wsMinimized;
#4912.              SW_SHOWMAXIMIZED: MainForm.WindowState := wsMaximized;
#4913.         end;
#4914.         if FShowMainForm then
#4915.              if FMainForm.FWindowState = wsMinimized then
#4916.                   Minimize else
#4917.         FMainForm.Visible := True;
#4918.         repeat
#4919.              HandleMessage
#4920.         until Terminated;
#4921.    end;
#4922.    finally
#4923.         FRunning := False;
#4924.    end;
#4925.  end;
```

我們可以看出 TApplication::Run 所作的事情就是先將主視窗(FMainForm)的視窗形態（WindowsState，如:放到最大、縮到最小與正常大小)給設定好（4910 至 4916 行），並設定主視窗的顯示與否(Visible，4917 行)。而 4918 至 4920 行是個迴圈，由程式碼上可以看出這個迴圈會一直執行直到 Terminated 這個旗標被設定爲 true 時才中斷，這個近乎無窮迴圈也就是我們所謂的訊息處理迴圈。所有由外部進到應用程式的訊息都得透過這個迴圈做分派處理。咱們繼續追下去看：

```
#4832.  procedure TApplication.HandleMessage;
#4833.  begin
#4834.    if not ProcessMessage then Idle;
#4835.  end;
```

由 TApplication::Run 追下來到了 TApplication::HandleMessage，可以發現會先交給 TApplication::ProcessMessage 處理，若沒有訊息需要處理就交給 TApplication::Idle 做處理，也就是應用程式閒置時的處理函式。先看看 TApplication::ProcessMessage 到底做的是什麼：

```
#4802.  function TApplication.ProcessMessage: Boolean;
#4803.  var
#4804.    Handled: Boolean;
#4805.    Msg: TMsg;
#4806.  begin
#4807.    Result := False;
#4808.    if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then
#4809.    begin
#4810.         Result := True;
#4811.         if Msg.Message <> WM_QUIT then
#4812.         begin
#4813.              Handled := False;
#4814.              if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
#4815.              if not IsHintMsg(Msg) and not Handled and not IsMDIMsg(Msg) and
#4816.              not IsKeyMsg(Msg) and not IsDlgMsg(Msg) then
#4817.              begin
#4818.                   TranslateMessage(Msg);
#4819.                   DispatchMessage(Msg);
#4820.              end;
#4821.         end
#4822.         else
#4823.              FTerminate := True;
#4824.    end;
#4825.  end;
```

由 4808 行可以看到 TApplication 類別使用 PeekMessage 函式從訊息佇列裡取得訊息，之後先檢查訊息是否爲 WM_QUIT，若是則把 Terminated 旗標設定爲 true，讓應用程式給中斷。但若訊息並非 WM_QUIT，會先檢查是否有指定 OnMessage 函式，若有則優先把訊息交給所指定的 OnMessage 函式處理，若無指定則開始將訊息分配。

因此若由所指定的 OnMessage 函式來接收並訊息，那麼我們所收到的訊息應該就是第一手訊息，那麼我們的處理函式也就是最快速處理的囉！先看看 OnMessage 的原始定義：

```
void __fastcall ApplicationOnMessage(tagMSG &Msg, bool &Handled)
```

若你不希望 VCL 繼續處理這個訊息的話，只需把 Handled 這個旗標給設定成 true 即可。就讓筆者舉個簡單例子：

```
#0001.  void __fastcall TForm1::FormCreate(TObject *Sender)
#0002.  {
#0003.    Application->OnMessage = ApplicationOnMessage;
#0004.  }
#0005.  //---------------------------------------------------------------
#0006.  void __fastcall TForm1::ApplicationOnMessage(tagMSG &Msg, bool &Handled)
#0007.  {
#0008.    switch(Msg.message)
#0009.    {
#0010.         case WM_COPYDATA:
#0011.              TMessage Message;
#0012.              Message.Msg = Msg.message;
#0013.              Message.WParam = Msg.wParam;
#0014.              Message.LParam = Msg.lParam;
#0015.              OnCopyData(Message);//處理 WM_COPYDATA 的函式
#0016.              Handled = true;
#0017.              break;
#0018.    }
#0019.  }
#0020.  //---------------------------------------------------------------
```

這樣一來這個範例的功能就與先前兩個方法『覆載 TObject::Dispatch』與『覆載 TControl::WndProc』所得到的結果相同。但是有一點要注意的是，指定 OnMessage 的方法是直接由 TApplication 類別來處理，所以若使用指定 OnMessage 函式來接收處理特定的視窗訊息，則發出訊息端必須發給 Application 這個全域物件的視窗代碼，爲 Application->Handle，而若使用其餘兩個方法則要發給所要處理的視窗的視窗代碼，如：Form1->Handle。

以先前建立的那個什麼都沒做的應用程式爲例，使用 FindWindow 方式來取得視窗代碼有

兩種方法：

A、用 FindWindow("TApplication","Project1");

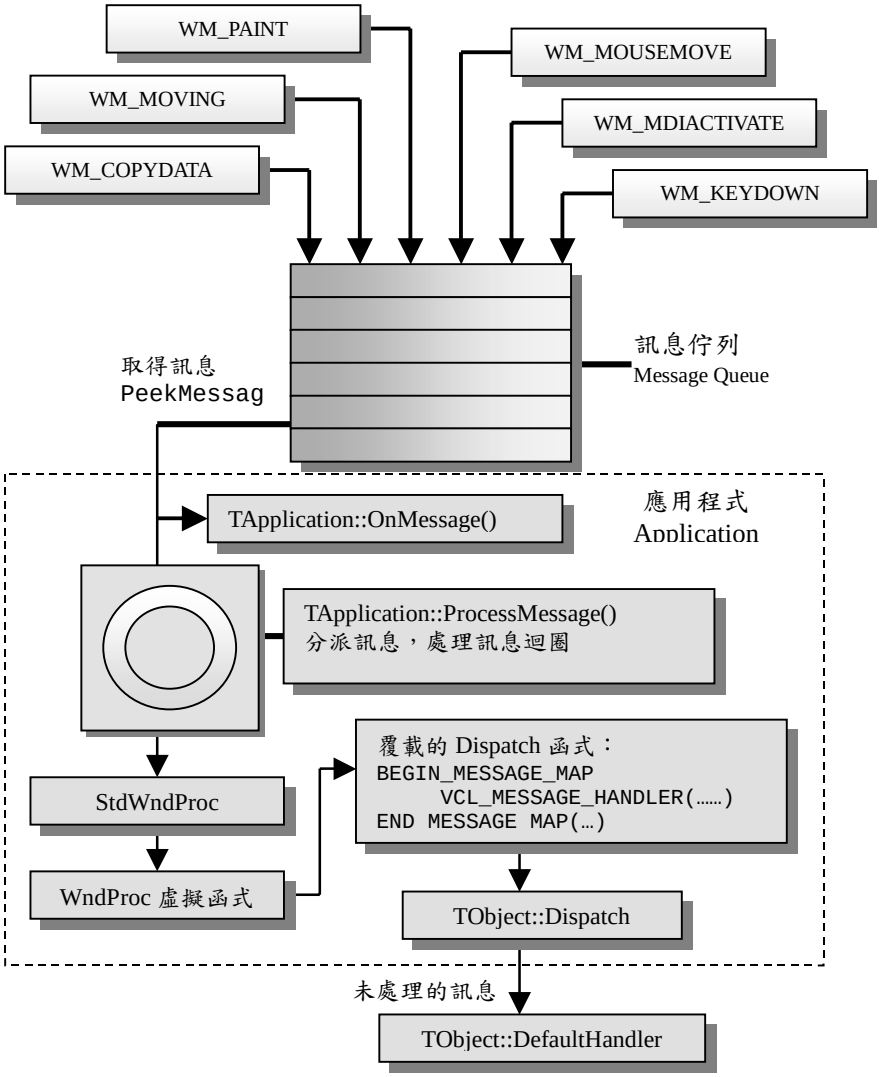
來取得 Application->Handle。

B、用 FindWindow("TForm1","Form1");

來取得 Form1->Handle。

■ VCL 視窗訊息的流動方向

由以上三種訊息的接收方法以及 VCL 的原始碼追蹤經驗，我們可以勾勒出在 VCL 所架構下的應用程式中視窗訊息流動的程序（圖五）。



圖五、VCL 內部視窗訊息處理流程

當程式執行到 Application->Run()時就進入到訊息處理迴圈裡，我們的應用程式透過 PeekMessage 這個 API 函式由訊息佇列裡取得發給本應用程式的訊息，並將此訊息從訊息佇列裡移除，之後把訊息交給 TApplication::ProcessMessage()來將訊息分派給該收到的物件上。但若指定了 TApplication 類別裡的 OnMessage 事件時，會先將訊息引導到指定的 OnMessage 函式上，

經過 OnMessage 函式處理過後的訊息才流到訊息迴圈裡。訊息經過 TApplication::ProcessMessage() 的分派後，訊息會先流到 StdWndProc，而 StdWndProc 的工作就是將訊息推進到 WndProc 函式裡做處理，進入到 WndProc 函式後被 WndProc 所認得的就優先被處理，最後不認得再交由覆載的 Dispatch 函式處理，若仍無法處理則交給父類別的 Dispatch 做處理，而 Dispatch 函式最後會把訊息推動到 DefaultHandler 函式裡去處理。嗯！又學到了一點，其實也可以重新覆載 DefaultHandler 來處理訊息，但這屬於訊息接收最後的底限，要從這裡才接收嗎？隨便你囉！

根據這張訊息流通圖來看，這三種方法所接收到的訊息先後爲：

1. TApplication::OnMessage
2. 覆載的 WinProc
3. 覆載的 Dispatch

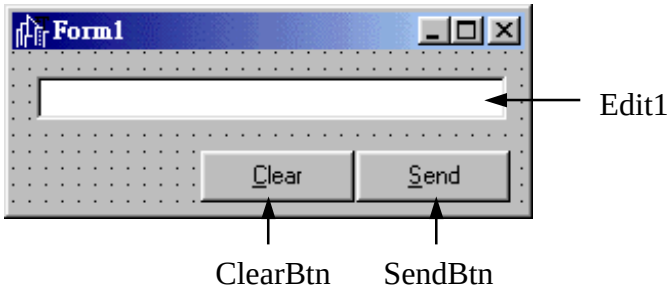
那讀者們要用哪一張網子來網住訊息呢？就看讀者們的喜好爲何囉！

■ 親手打造 TraceWin

現在我們終於懂得原來 VCL 裡有這般的程序將訊息傳遞到他應該到達的地方。了解了這些之後，應該是該始動手來實作咱們所要的 TraceWin。

1、訊息發出端

先來設計一個簡單板的訊息發出端，放置兩個 Button 元件以及一個 Edit 元件到 Form1 上面，其擺設如圖六。



圖六、訊息發出端元件設置

爲 Clear 按鈕加上 OnClick 事件，用來清除 Edit1 元件上的文字：

```
#0001.  void __fastcall TForm1::ClearBtnClick(TObject *Sender)
#0002.  {
#0003.    Edit1->Text = “”;// 清除 Edit1->Text 的內容
#0004.  }
```

爲 Send 按鈕加上 OnClick 事件，將訊息送出：

```
#0001.  void __fastcall TForm1::SendBtnClick(TObject *Sender)
#0002.  {
#0003.    char Msg[255];
#0004.    sprintf(Msg,"%s",Edit1->Text.c_str());
#0005.    COPYDATASTRUCT *pcp = new COPYDATASTRUCT;
#0006.    pcp->dwData = 0;
#0007.    pcp->cbData = sizeof(Msg);
#0008.    pcp->lpData = &Msg;
#0009.    SendMessage(FindWindow("TForm1","TRACE Window"),
#0010.    WM_COPYDATA, (WPARAM)NULL, (LPARAM)pcp);
#0011.    delete pcp;
#0012.  }
```

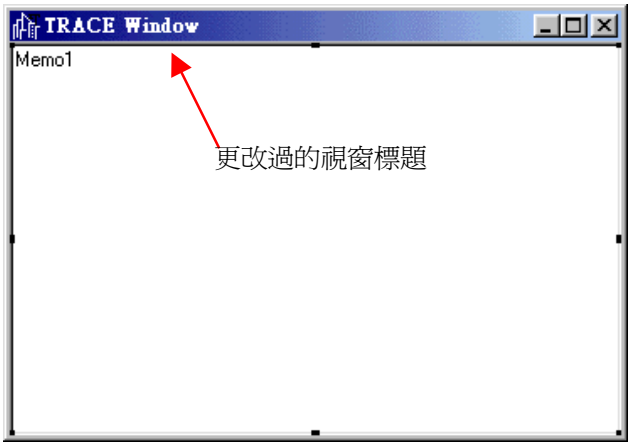
其中 9 至 10 行的 SendMessage 特別指明要將訊息送給視窗標題爲 Trace Window 且登錄的視窗類別爲 TForm1 的視窗，則表示訊息接收端必須使用『覆載 TObject::Dispatch 函式』或『覆載 TControl::WndProc 函式』才能夠收得到訊息。若要改成由『指定 Application::OnMessage 函式』的方法則必須將 9 至 10 行的 SendMessage 改爲：

```
#0009.    SendMessage(FindWindow("TApplication","TRACE Window"),
#0010.    WM_COPYDATA, (WPARAM)NULL, (LPARAM)pcp);
```

還有一個地方值得注意的是：先前說過使用 WM_COPYDATA 時，SendMessage 的第三個參數必須是發出端的視窗代碼，但由於我們在將所需檢查的訊息丟給 TraceWin 時，並不需要 TraceWin 對發出端有所回應，所以在此並不一定要將發出端的視窗代碼處給 TraceWin 那一端收到。

2、訊息接收端 — TraceWin：

OK!剛剛已經把簡單版本的訊息發出端給設置好，接下來就是要把接收端給搞定了。先做一個簡單的接收端來接收由發出端所發出的文字訊息並秀出來，所以在 Form1 上放置一個 Memo（如圖七），且設定 Memo 的 Align 屬性爲 alClient，並把其中 Lines 的內容給刪除。但是要記住，



圖七、訊息接收端元件設置

我們先得把 Form1 的 Caption 給設定成『TRACE Window』讓發出端能夠正確使用 FindWindow 來取得接收端的視窗代碼。另外爲了順便是一下這三種接收訊息的方法，也要記得把到 Project Option 裡把 Application 的 Title 給改成『TRACE Window』（如圖八），或是在 Project1.cpp 裡頭加上指定 Application->Title 的程式碼，如下：

```
#0010.          Application->Initialize();
#0011.          Application->Title = “TRACE Window”;
#0012.          Application->CreateForm(__classid(TForm1), &Form1);
```

先爲 Form1 建立 OnCopyData 函式以供處理訊息用：

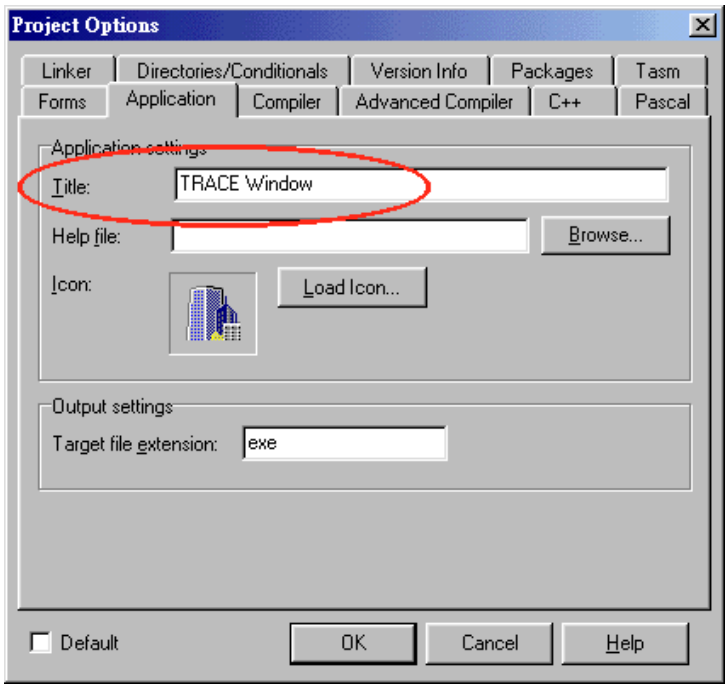
在 Unit1.h 裡加入：

```
private:  // User declarations
     void __fastcall OnCopyData(TMessage &Msg);
```

在 Unit1.cpp 裡加入：

```
#0001.  void __fastcall TForm1::OnCopyData(TMessage &Msg)
#0002.  {
#0003.     PCOPYDATASTRUCT pcp;
#0004.     pcp = (PCOPYDATASTRUCT)Msg.LParam;
#0005.     char* TraceMsg;
#0006.     TraceMsg = (char*)pcp->lpData;
#0007.     Memo1->Lines->Add(TraceMsg);//將接收到的陣列一行一行加到給到 Memo1 裡頭
```

```
#0008.  }
```



圖八、設定 Application 的 Title

再來就是選擇咱們要的訊息接收方式了，爲了複習上頭所說過的三種方法三種方式都試試看好了：

A、『覆載 Dispatch 函式』

在 unit1.h 裡加入：

```
#0001.  public:      // User declarations
#0002.  BEGIN_MESSAGE_MAP
#0003.     VCL_MESSAGE_HANDLER(WM_COPYDATA,TMessage,OnCopyData)
#0004.  END_MESSAGE_MAP(TForm)
```

這樣即可配合前頭的那個 OnCopyData 函式來處理訊息了。

B、『覆載 WndProc 函式』

在 unit1.h 裡加入：

```
#0001.  protected:        // User declarations
#0002.     void __fastcall WndProc(TMessage &Message);
```

在 unit1.cpp 裡加入：

```
#0001.  void __fastcall TForm1::WndProc(TMessage &Message)
#0002.  {
#0003.     switch(Message.Msg)
#0004.     {
#0005.          case WM_COPYDATA:
#0006.               OnCopyData(Message);//處理 WM_COPYDATA 的函式
#0007.               break;
#0008.     }
#0009.     TForm::WndProc(Message);
#0010.  }
```

這也一樣可配合前頭的那個 OnCopyData 函式來處理訊息了。

C、『指定 OnMessage 函式』

在 unit1.h 裡加入：

```
#0001.  protected:       // User declarations
#0002.    void __fastcall ApplicationOnMessage(tagMSG &Msg, bool &Handled);
```

在 unit1.cpp 裡加入：

```
#0001.  void __fastcall TForm1::FormCreate(TObject *Sender)
#0002.  {
#0003.    Application->OnMessage = ApplicationOnMessage;
#0004.  }
#0005.  //---------------------------------------------------------------
#0006.  void __fastcall TForm1::ApplicationOnMessage(tagMSG &Msg, bool &Handled)
#0007.  {
#0008.    switch(Msg.message)
#0009.    {
#0010.         case WM_COPYDATA:
#0021.              TMessage Message;
#0022.              Message.Msg = Msg.message;
#0023.              Message.WParam = Msg.wParam;
#0024.              Message.LParam = Msg.lParam;
#0011.              OnCopyData(Message);//處理 WM_COPYDATA 的函式
#0012.              Handled = true;
#0013.              break;
#0014.    }
#0015.  }
#0016.  //---------------------------------------------------------------
```

這也一樣可配合前頭的那個 OnCopyData 函式來處理訊息了。

不過誠如先前所提到的，採用這種做法相對的在訊息發出端就得把訊息送給 Application 物件而不是送給 Form1。發出端的 FindWindow 應該是；FindWindow("TApplication","TRACE Window")，這樣才能夠取得 Application->Handle。否則若把訊息直接發給 Application 而你在 Form1 上頭攔截訊息是絕對攔截不到的，反之亦然。

圖九、簡易訊息發出端及訊息接收端收發測試

3、訊息收發測試：

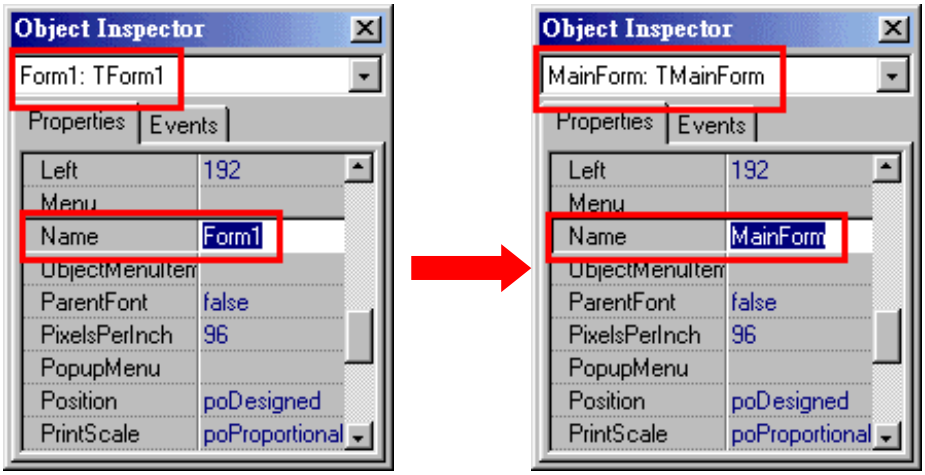
OK！已經完成了大半，先來測試一下吧（圖九）：

可以清楚的看到，我們可已經 Edit1 元件裡頭的文字完整的傳送到 TRACE Window 裡頭去，但這樣就完成了嗎？那可不！還有很多地方應該加強或修正的：

A、TRACE Window 的視窗類別修改

先前我們使用 FindWindow 來取得視窗代碼時使用 FindWindow(“TForm1”,”TRACE Window”)。但 C++Builder 所預設的視窗註冊類別爲 TForm1，我們一樣使用 TForm1 似乎有點怪怪的！筆者的習慣是改爲 TMainFrom，但怎麼改呢？

從物件檢視器（Object Inspector）上頭改就可以了，找到原先 TForm1 的 Name 屬性欄把他由 Form1 改爲 MainForm 則繼承至 TForm 類別的 TForm1 類別則會變爲 TMainForm 類別（圖十）。也因此原先所使用的 FindWindow 要改成 FindWindow(“TMainForm”,”TRACE Window”)。



圖十、修改 Form 的類別名稱

B、TRACE Window 的功能加強

嗯,改好了視窗類別後還缺什麼呢？缺的可多了！如把 Memo1 元件上的文字給存檔起來以便日後觀看、清除 Memo1 元件上的內容或是增加進制轉換等等的功能，說都說不完。這些都是應該且必須的，該怎麼做呢？這些功能筆者在此僅舉幾個簡單的說明，不再詳述：

a、Memo1 元件上的文字存檔：

在 VCL 架構中，所有的 VCL 元件都被包裝的非常完整，所以做起例行工作來非常的容易，要將 Memo1 元件上的資料存檔只需如下：

```
Memo1->Lines->SaveToFile("D:\\trace.txt");
```

這樣就會把資料存到 D:\trace.txt 裡頭了。

不過這樣似乎有點太過簡單，咱們再加上一個存檔對話盒吧：

```
#0001.  TSaveDialog *SD = new TSaveDialog(this);
#0002.  SD->Options << ofOverwritePrompt;//增加檔案覆蓋時的確認功能
#0003.  SD->Filter = "Text files (*.txt)|*.TXT";
#0004.  if (SD->Execute())
#0005.  {
#0006.     Memo1->Lines->SaveToFile(SD->FileName);
#0007.  }
#0008.  delete SD;
```

這樣一來，存檔時就會出現存檔對話盒讓你選擇要儲存的檔案位置了。

b、清除 Memo1 上的文字：

清除 Memo1 上的文字筆存檔更容易，只需使用 Clear()這個成員函式，如下：

```
Memo1->Clear();
```

這樣就可以輕輕鬆鬆的將 Memo1 元件上頭的文字給清除了。

C、訊息發出端的指令簡化

根據先前的訊息發出端範例來看，我們使用特定的型別才能夠轉換，可用性頗低，且還要打那麼多的字，若每次要使用時還得打那麼多字，豈不累人，因此咱們可以把他編成通用函式，等需要用到時只需 include 我們所編寫的函式檔案即可，我把函式名稱命名爲 **dout** 有 Debug Out 的意思,並且讓 dout 能夠支援 Unicode string、signal byte string 以及 VCL 裡頭特有的 AnsiString，如此一來這個表頭檔在 Visual C++、Borland C++以及 Borland C++Builder 等 Win32 平台上的 C++

編譯器都一樣可以使用，當然了對於一些特定的類別如 AnsiString 會用#define 這些前置指令將他隔離，讓不認得的編譯器不會編譯到這些個不認得的類別。

程式列表五、dout.h

```
#0001. //////////////////////////////////////////////////////////////////////
#0002. //
#0003. //  Easy Bug Tracer v 1.0.6
#0004. //  @(#) dout.h 1.0.6, last edit: 09/15/98 09:45
#0005. //  @(#) Copyright (C) 1998 Martin Hsiao (martins1@ms3.hinet.net)
#0006. //  @(#) Martin's WorkShop  (http://insidebcb.copystar.com.tw)
#0007. //
#0008. //     This program is distributed in the hope that it will be useful,
#0009. //     but WITHOUT ANY WARRANTY; without even the implied warranty of
#0010. //     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#0011. //
#0012. //////////////////////////////////////////////////////////////////////
#0013.
#0014. #ifndef __DOUT_H_
#0015. #define __DOUT_H_
#0016.
#0017. #include <windows.h>
#0018.
#0019. #ifndef __STDIO_H
#0020. #include <stdio.h>
#0021. #endif /* __STDIO_H */
#0022.
#0023. #ifndef __WCHAR_H
#0024. #include <wchar.h>
#0025. #endif /* __WCHAR_H */
#0026.
#0027. #define SMSG(X) SendMessage(FindWindow("TMainForm","TRACE Window") ,\
#0028.                             WM_COPYDATA, (WPARAM)NULL,(LPARAM)X);
#0029. #define DebugOut(X) \
#0030. {   \
#0031.    COPYDATASTRUCT *pcp = new COPYDATASTRUCT;\
#0032.    pcp->dwData = 0;\
#0033.    pcp->cbData = sizeof(X);\
#0034.    pcp->lpData = &X;\
#0035.    SMSG(pcp);\
#0036.    delete pcp;\
#0037. }
#0038.
#0039. void douth(long Data)//for Tracing Hex Value
#0040. {
#0041.    char TM[255];
#0042.    sprintf(TM,"0x%x",Data);
#0043.    DebugOut(TM);
#0044. }
#0045.
#0046. void dout(char* Data)
#0047. {
#0048.    char TM[255];
#0049.    sprintf(TM,"%s",Data);
#0050.    DebugOut(TM);
#0051. }
#0052.
#0053. void dout(WCHAR* Data) //for Tracing Unicode string
#0054. {
#0055.    char TM[255];
#0056.    #ifdef DSTRING_H
#0057.         AnsiString AnsiData = AnsiString(Data);
#0058.         sprintf(TM,"%s",AnsiData.c_str());
#0059.    #else
#0060.         sprintf(TM,"%ls",Data);
#0061.    #endif /* DSTRING_H */
#0062.    DebugOut(TM);
#0063. }
#0064.
#0065. #ifdef DSTRING_H
#0066. void dout(AnsiString* Data)//for Tracing AnsiString BCB only
#0067. {
#0068.    char TM[255];
#0069.    sprintf(TM,"%s",Data->c_str());
#0070.    DebugOut(TM);
```

```
#0071.  }
#0072.
#0073.  void dout(AnsiString Data) //for Tracing AnsiString BCB only
#0074.  {
#0075.     char TM[255];
#0076.     sprintf(TM,"%s",Data.c_str());
#0077.     DebugOut(TM);
#0078.  }
#0079.
#0080.  #endif /* DSTRING_H */
#0081.
#0082.  #endif
```

既然在 C++Builder 下可以使用，那們我們就一同造福 C++Builder 的哥兒們 Delphi 吧！讓這個 TRACE Window 一樣可以收到由 Delphi 所傳送出來的訊息。

以下是在 Delphi 下使用時的程式碼：

程式列表六、dout.pas

```
#0001.  ////////////////////////////////////////////////////////////////////////
#0002.  //
#0003.  //  Easy Bug Tracer v 1.0.6
#0004.  //  @(#) dout.pas 1.0.6, last edit: 09/15/98 09:45
#0005.  //  @(#) Copyright (C) 1998 Martin Hsiao (martins1@ms3.hinet.net)
#0083.  //  @(#) Martin's WorkShop  (http://insidebcb.copystar.com.tw)
#0006.  //
#0007.  //     This program is distributed in the hope that it will be useful,
#0008.  //     but WITHOUT ANY WARRANTY; without even the implied warranty of
#0009.  //     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#0010.  //
#0011.  ////////////////////////////////////////////////////////////////////////
#0012.
#0013.  unit douts;
#0014.
#0015.  interface
#0016.
#0017.  uses Windows, Messages;
#0018.  procedure dout(Data : pchar);overload;
#0019.  procedure dout(Data : String);overload;
#0020.
#0021.  implementation
#0022.
#0023.  procedure charout(Data : pchar);overload;
#0024.  var
#0025.     sSend : AnsiString;
#0026.     cdsData : TCopyDataStruct;
#0027.  begin
#0028.     sSend := Data;
#0029.     with cdsData do
#0030.     begin
#0031.          dwData := 0;
#0032.          cbData := Length(sSend) + 1;
#0033.          lpData := pchar(sSend);
#0034.     end;
#0035.     SendMessage(FindWindow('TMainFrame','TRACE Window'),
#0036.                     WM_COPYDATA,WPARAM(nil),LPARAM(@cdsData));
#0037.  end;
#0038.
#0039.  procedure dout(Data : pchar);overload;
#0040.  begin
#0041.     charout(Data);
#0042.  end;
#0043.
#0044.  procedure dout(Data : String);overload;
#0045.  begin
#0046.     charout(pchar(Data));
#0047.  end;
#0048.
#0049.  end.
```

對於使用 Win32 平台下 C++編譯器的讀者們，可以把 dout.h 給放置到各編譯器的\include 目錄裡頭，屆時要用到 dout 指令時只需加入 dout.h 這個標頭檔就可以了。而對於 Delphi 的使用者

來說，把 dout.pas 編譯後，把 dout.dcu 給放置到\lib 目錄下即可，此外使用時記得 use dout 喔。若讀者們懶得親手打這麼長的程式碼的話（長嗎？☺），可以到筆者的網站上頭抓，網址是 http://insidebcb.copystar.com.tw。此外筆者所寫的多功能版本 TRACE Window 也放在網站讓讀者們下載。

■ 結語

呼！似乎做了一個 VCL 裡頭的深度歷險，夠深嗎？一點也不！其實在 VCL 的領域裡，這只挖到一半罷了，更深入的祕密，更高階的技巧都還等著我們去挖掘呢！爲了挖掘出 VCL 架構裡的奧祕，文章內列出了不少使用 Object Pascal 所撰寫的 VCL 原始碼，希望沒有把讀者們搞得暈頭轉向。以目前的情況來看，下一個版本的 VCL 應該還是由 Object Pascal 爲主體撰寫而成，也因此不少的 C/C++好手們都因爲 Object Pascal 而裹足不前去挖掘這深奧的祕密，更對 VCL 的 Object Pascal 出身血統而打抱不平。在筆者使用 C++Builder 之前，對於 Object Pascal 是一蹺不通，因此對於 Object Pascal 的學習一直抱著能不費時間學就不學的態度。但不久後，因爲工作的需要得以開始鑽研 VCL 的內部構造，這才發現 VCL 的內部是如此的精妙，對於慣用 C/C++的我來說何嘗不是一個學習 Object Pascal 的機會呢？因此，在此呼籲各位讀者，若你真正想要把 C++Builder 與 VCL 架構給學好，該是自己開始研讀 VCL 的原始碼並學習 Object Pascal 的好時候了。

註一、與 IPC 主題相關的書籍有：
Advanced Windows, Microsoft Press, Jeffrey Richter, Chapter 10.
Multithreading Application in Win32, Addison Wesley, Jim Beveridge & Robert Wiener, Chapter13.
Programming Windows 95, Microsoft Press, Charles Petzold, Chapert 17.

註二、需要用到廣播的訊息必須先使用 RegisterWindowMessage 函式來向系統註冊。