深入 C++Builder 3
探訪動態連結函式庫(Dynamic Linking Libraries,DLLs)

蕭永哲 martins1@ms3.hinet.net

打 Windows 從娘胎出生後，動態連結函式庫(Dynamic Linking Libraries,DLLs)宛如這個新生兒的血液一般的重要，DLLs 一直扮演著 Windows 的基石這個角色。到了 Win32 雄霸一方的世紀，DLLs 的威風更是不減當年，幾乎所有的 Win32 API 都以隱身在 DLLs 中的形式存在，Windows 血液裡流的盡乎都是 DLLs。在 Windows 下 DLL 通常是以『副檔名爲 DLL 的檔案』存在，DLL 內可以包含提供外部呼叫的函式、資源（Resource）以及各種的變數，當 DLLs 被應用程式載入時這些 DLLs 內含的資料都會變爲應用程式所屬行程的一部份。我們可以把 DLL 當成一個函式庫（Library），但因爲還要能夠被動態載入，這個函式庫自然比傳統的函式庫複雜些。

DLL 一二說

DLLs 的出現提供了程式設計師一個將程式模組化的方法，別於 C++類別的建構時期模組化，DLL 乃是執行時期模組化，因此程式設計師可以在建構程式時將所需要用到的函式分門別類的製造成 DLLs 的形式。但爲什麼我們要使用 DLLs 呢？DLLs 充其量不就你把程式碼、資源等獨立到另一個檔案裡頭去，到底有什麼好處值得我們大費周章地把部份程式寫成 DLLs 的形式呢？使用 DLL 的好處大致可以如下歸類：

1、有效率的重複使用程式碼

當程式設計師所撰寫的程式碼一多起來，必然地會發現有很多程式碼是在做相同的事情，通常當程式設計師遇到這些重複的程式碼，最平常不過的方法就是把這些重複的程式碼大則獨立成『函式』（Function）小則獨立成『巨集』（Marco）。但是，當這些函式不單單只在單一應用程式內會用到，而是在撰寫許多應用程式時都得用到時，這些常用的函式通常就會被製作成函式庫來使用，例如 C 語言的 Runtime Library（RTL）；但編譯器在編譯應用程式遇到函式庫時，會把這些隱身在函式庫裡頭的函式實體內容如同我們在程式裡頭撰寫這些函式的原始碼般地加進應用程式的執行檔中，也就是說當你所用的函式庫越多時，你的執行檔也就相對的會越來越龐大，這個做法也就是我們所謂的靜態連結（Static Linking）。因此爲了避免應用程式的過分龐大，有人提出了動態連結（Dynamic Linking）的做法，所謂動態連結就是提供了一個做法讓我們不需要把應用程式的執行檔變得如此龐大，但一樣可以享用這些使用頻率高的函式。也就是說這些函式會在程式執行時才被載入，而不是直接編譯在執行檔中。這樣一來可以讓我們更有效率使用這些函式。但相對的，當你所撰寫的程式得交給他人使用時，你除了得把你所編譯好的 EXE 檔交給他之外，還得一併把編譯好的 DLLs 檔交給他，否則執行起來一定會產生不可預期的錯誤。

2、區分程式碼

依據先前的第一點，若有朝一日發現這些被包裝在 DLLs 之中的函式的實作方法有點錯誤或是發現有更好的做法時，需要更動僅只有部份的 DLLs 原始碼，重新將修改過的 DLLs 給編譯後就可以達成更新程式的目的，至於應用程式端連動都不需要動一下；當然了，前提是這些個修改過的函式名稱與傳入的參數型別宣告都不能夠更動。根據這些個特性，咱們可以把整個應用程式中的函式依照功能或目的分類，並將這些分類好的函式組合成許多個 DLLs 模組，將執行檔給分割城數個小檔案，讓這些 DLLs 模組分工合作來完成應用程式所要達到的目的。這樣一來，對於應用程式的維護以及更新就不需要大費周章地從頭再編譯一次了，僅需把要有修改到的 DLLs 從新編譯就可以達成程式的更新。

3、節省記憶體的使用量

先前提到：DLLs 的載入是在應用程式執行時才被載入，甚至還可以是可以在應用程式所需用到函式時才被載入。此外 DLLs 還有一個很重要的特性：若不同的應用程式但需要相同的 DLL 中的函式時，DLL 僅在第一個使用到 DLL 的應用程式執行時載入，只要這個應用程式尚未結束而其他的應用程式又正好需要使用到同一 DLL 中的函式時，DLL 不需要再重新被載入到記憶體中就可以供第一個應用程式以外需要用到 DLL 的應用程式使用，一直到沒有任何應用程式使用這個 DLL 時，DLL 才會跟著最後一個使用 DLL 的應用程

式一起從記憶體裡頭消失，因此使用 DLLs 來包裝常用的函式是個不錯能夠節省記憶體與系統資源的做法。

4、將程式推向國際舞台

DLL 在設計時，就已經被設計不只是能夠放入函式而已，還能夠被放入許多的資源（Resource），如：可以放入選單資料、字串資料、圖形資料等等…。也因此 DLL 很常被拿來作爲應用程式邁向國際舞台的一個墊腳石。你可以在應用程式被執行時檢查執行應用程式的作業系統語言版本，之後把當地語言版本的 Resource DLL 給載入，讓所有的文字及畫面都達到當地語言化的目的。而這個功能在 C++Builder 3 當中經由 Borland C++Builder 部門工程師的努力，已經幫我們把這些煩人的步驟給簡化了許多，我們僅須按下選單上的 New 並選擇 Resource DLL Wizard，並將將需要更改的文字給更改成不同的語言，並不須在執行時檢查執行平台的語言爲何。其餘煩瑣的工作都已經被 Borland 工程師給完成了。我們最後只需要重新編譯這個 Resource DLL 並附在應用程式中就可以完成一個國際化的應用程式了。

既然 DLLs 在 Windows 上頭是那麼的重要且使用 DLLs 還有那麼多的好處，當然值得咱們來好好的了解一下。先來討論 DLLs 的基本架構。

Import 還是 Export？

我們已經知道當 DLL 被應用程式載入時，DLL 內所含有的資料都會成爲應用程式所屬行程中的一部份，這到底怎麼辦到的？其實在 DLL 被應用程式載入時，DLL 會被先設定一個基底位址（Base Address），若這個基底位址並沒有和應用程式中的其他資源互相衝突，則這個 DLL 檔會被映射到載入端行程內的相同位址上讓載入端應用。那 DLL 中到底有多少資訊可以被載入呢？先看看圖一：

```
Object table:
#   Name     VirtSize  RVA      PhysSize  Phys off  Flags
--  -------- --------  -------- --------  --------  --------
01  .text    00008000  00001000 00007C00  00000600  60000020 [CER]
02  .data    00001000  00009000 00000800  00008200  C0000040 [IRW]
03  .tls     00001000  0000A000 00000200  00008A00  C0000040 [IRW]
04  .idata   00001000  0000B000 00000A00  00008C00  40000040 [IR]
05  .edata   00001000  0000C000 00000200  00009600  40000040 [IR]
06  .rsrc    00001000  0000D000 00000C00  00009800  40000040 [IR]
07  .reloc   00001000  0000E000 00000C00  0000A400  50000040 [ISR]

Key to section flags:
  C - contains code
  E - executable
  I - contains initialized data
  R - readable
  S - shareable
  W - writeable
```

圖一、DLL 中的 Section 列表

圖一是使用 C++Builder 內附上的 tdump.exe 工具列出一個 DLL 檔內所有的 Section，tdump 是個非常好用的工具，之後使用次會不少，先說明用法，用法很簡單：

tdump inputfile outputfile

這樣就可以把 inputfile 裡頭的資料給格式化輸出到 outputfile 裡頭，當然了，你的 inputfile 得是 tdump 認得的檔案：DOS 下的執行檔、PE 格式執行檔（註 1）、.OBJ 檔、.LIB 檔，其餘的檔案若爲文字格式就直接輸出，而 Binary 格式的檔案就以 HEX Dump 格式輸出。此外，在 Microsoft Visual C++裡頭的 dumpbin.exe 也提供相同功能來分析這些檔案。

圖一中的這些 Section 表示著：

Section	包含	意義
.text	應用程式或 DLL 的程式碼	這個 section 包含了一般性的程式碼，就是先前提到的除了自己所撰寫的程式碼外還有 Runtime Library 的程式碼。
.data	具有初始值的資料	這個 section 存放了在編譯時期就已經具有初始值的資料；包括

		了全域變數（global variable）、靜態變數（static variable）以及”Hello World”這一類字串等等…。
.tls	執行緒內部儲存空間	thread local storage
.idata	輸入名稱表	這個 section 包含了有『從其他 DLLs 中輸入過來的函式與資料』的相關資訊。
.edata	輸出名稱表	這個 section 恰好與.idate 相反，是存放了由此 EXE 或 DLL 輸出給外頭使用的函式與資料的相關資訊。
.rsrc	資源	若你使用過 Microsoft Visual C++或是 Borland Resoure Workshop 來觀察過 EXE 或 DLL，你所看到的那些 resource date 就是儲存在這個 section 裡。也就是編譯器將應用程式所需要用到的 resource date 都整理好一起放到這個 section 裡頭。
.reloc	修正表資訊	這個 section 裡頭含有一個 base relocationsg 是一個調整值，先前說過當….會，但若無法載入到預設的位址，就會依據這個調整值來作調整。

除了了解這些 section 之外，你還必須知道的另一個觀念是所謂的相對虛擬位址（Relative Virtual Address，RVA)。PE 格式執行檔中有許多資料的位址都是以 RVA 表示。簡單的來說 RVA 是某一項資料從檔案被映射進來的起點算起的偏移值（offset）。舉個例子，我們說 Windows 載入器把一個 PE 格式執行檔檔映射到虛擬位址空間 0x400000 處，如果在此執行檔中有一個函式的函式指標起始於 0x40C000，那麼這個函式指標的 RVA 就是 0x C000：

虛擬位址（0x40C000）— 基底位址（0x400000） = RVA（0xC000）

只要把相對虛擬位址加上基底位址，相對虛擬位址就可以被轉換爲一個有用的指標。『基底位址』(Base Address)也是另一個重要名詞，通常基底位址是用來描述被映射到記憶體中的 EXE 或 DLL 的起始位址。

另外圖一中『Key to section flags』是這些 section 的屬性旗標種類，如：唯讀、共享或可寫入等等…每個 section 的屬相可以從『Object table』最後一欄的『Flag』中看出。

了解這些基本知識後，再回頭詳細看一看在圖一中的輸出輸入 section，我們已經知道，製作 DLL 的主要目的是製造一個模組化的函式或資料供其他的程式應用，而這種提供給其他 EXE 和 DLL 使用的方式就稱爲輸出（export），反之若取用其他的 EXE 或 DLL 中的函式，就稱爲輸入。在 DLL 中，你可以輸出任何想要輸出的資料，如函式、類別（class）或是資源等等…，我們把重點放在輸出輸入函式的部份，我們先來觀察一般的輸出函式：

```
C:\>tdump test1.dll

Turbo Dump  Version 5.0.16.4 Copyright (c) 1988, 1998 Borland International
                 Display of File TEST1.DLL
...
...
Exports from Test1.dll
  7 exported name(s), 7 export addresse(s).  Ordinal base is 1.
    Ordinal RVA      Name
    ------- --------  ----
    0000    00001019  Function_fastcall::
    0001    0000101e  MyFunc_Fast::
    0002    0000100a  Function_cdecl
    0003    00001005  Function_default
    0005    00001023  MyFunc_Cdecl
```

圖二、DLL 中的函式輸出表格

一樣可以使用先前提到的 tdump 來觀察 DLL 中的輸出表格，由圖二中可以看到輸出函式表格包含了 Ordinal、RVA 以及 Name 三個欄位表示。Name 就是輸出的函式名稱而 RVA—相對虛擬位址在前頭已經介

紹過了，至於 Ordinal 則是輸出表格中輸出函式的序號。這些輸出函式透過這個表格上的函式名稱與函式序號讓外界認得。當載入端最初在載入 DLL 時並不知道 DLL 內的輸出函式的正確位址只知道函式的序號與名稱，但在動態連結的過程中會建立起一個連連看表格將載入端的函式呼叫與被載入端內的函式正確位址給連結起來。

那我們要怎樣才能夠達成輸出的動作，其實很簡單，你只要在你的應用程式中需要輸出的函式前頭加上：__declspec(dllexport)即可，如：

```
__declspec(dllexport) void Function(void);
```

這樣一來就會把 Function(void)這個函式給放到輸出表格上頭了。
先前還有提到還可以將 C++類別透過 DLL 來輸出，可以的一樣是加上__declspec(dllexport)，如：

```
class __declspec(dllexport) __stdcall MyClass : public TObject{……};
```

當我們在看輸出表格時，會發現函式的輸出表格前頭好像有個如圖三一般的資料：

```
...
Imports from USER32.dll
    GetKeyboardType
    GetSystemMetrics
    LoadStringA
    MessageBoxA
...
```

圖三、DLL 中的輸入表格

這就是函式的輸入表格，這裡列出來的是 USER32.DLL 裡頭被我們使用到的函式。一樣的怎樣建立輸入表格呢？一般的 Win32 API 都已經在其所屬的 Header File 裡頭定義好了，我們只需加上#include <windows.h>就可以安心使用了，而對於自行打造的 DLL 中的輸出函式我們在載用此 DLL 的載入端就得在函式的宣告前面加上__declspec(dllimport)，如：

```
__declspec(dllimport) void Function(void);
```

目的是告訴編譯器 Function(void)這個函式是由外部輸入的。

Name Mangling

嗯！經過了先前的說明，讀者們應該知道若要在 DLLs 中將函式輸出，僅需要在函式的宣告中加上__declspec(dllexport)即可，若要載入別的 DLLs 中的函式則須在函式的宣告中加上__declspec(dllimport)，但這僅止於 C 編譯器，在 C++編譯器裡頭是行不通的，怎麼說呢？　就拿一個多載（overloading）的例子來說，如果函式的名稱都相同（當然所傳的參數型別不同），編譯器應該會如何處理？到底那個才是我們真正使用到的函式呢？其實在編譯器做編譯動作時，對這些同名的函式都動了點手腳讓同名的函式偷偷地變成不同名稱，以下面同名的三個函式爲例：

```
int Func(int X);
int Func(float X);
void Func(double *d);
```

使用 C++Builder 3.0 所編譯出來的函式名稱爲：（註 2）

```
@Func$qf
@Func$qi
@Func$qpd
```

而使用 Visual C++ 6.0 所編譯出來的函式名稱爲：（註 3）

```
?Func@@YAHH@Z
?Func@@YAHM@Z
?Func@@YAXPAN@Z
```

編譯器這個偷偷修改函式名稱的行爲稱爲『name mangling』，但除了函式名稱被改變外你是否發現還現另一件很嚴重的事情，就是同的編譯器竟然有著不同的 name mangling 做法，這表示著若咱們若使用 Borland

C++Builder 編譯器來開發應用程式時將無法使用一個經由 Microsoft Visual C++所編譯器完成的函式庫。此外，Naming Mangling 的作用不止於多載的函式上，C++程式中所有的 global 函式以及 class 中所有的成員（members）都會被 name mangling 這個動作給整型一下。那這樣不就沒戲唱了！若要在 C++Builder 下使用 Visual C++所編譯的 DLLs 豈不都得擁有 DLLs 的原始碼才能囉？其實不然，有方法可抑制 name mangling 的作用，就是在函式的宣告錢加上 extern “C”這個修飾詞，強制將函式以 C 語言的行台重現，而非以 C++語言的形態出現。但是要注意，多載函式可不能加上 extern ”C”這個修飾詞，因爲這個會造成一堆名稱相同的函式，若你硬是要使用 extern “C”在多載含上，編譯器一定會送你一個 ERROR（如下）做獎品。

in C++Builder：
[C++Error]Project1.cpp(13): Only one of s set of overloaded functions can be “C”.
in Visual C++
error C2733 : Second C linkage of overloaded function ‘Func’ not allowed

因此若拿原先多載的例子給加上 extern “C”：

```
extern “C” int Func(int X);
```

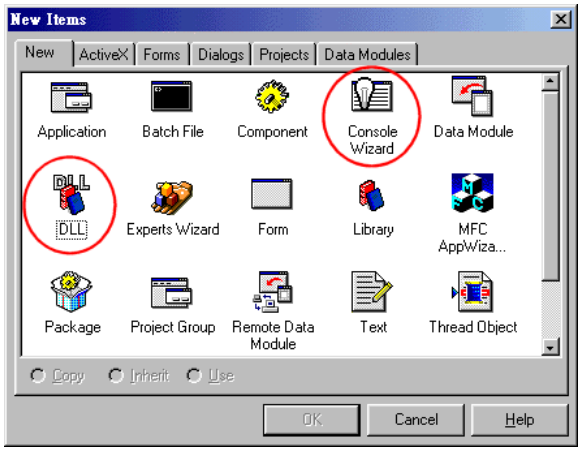
則會被編譯器給編譯成：

```
in Visual C++：
    _Func
in C++Builder：
    _Func
```

似乎兩個編譯器已經達成了一致的輸出。嗯！這樣一來就可以把 DLLs 互相使用了，不不不！還沒有那麼簡單，還有一個重要的議題『Call Conventions』，不過這個問題在此先不提，放到後頭在提，先談談怎樣使用 C++Builder 來建立 DLLs 吧！

建立 DLL

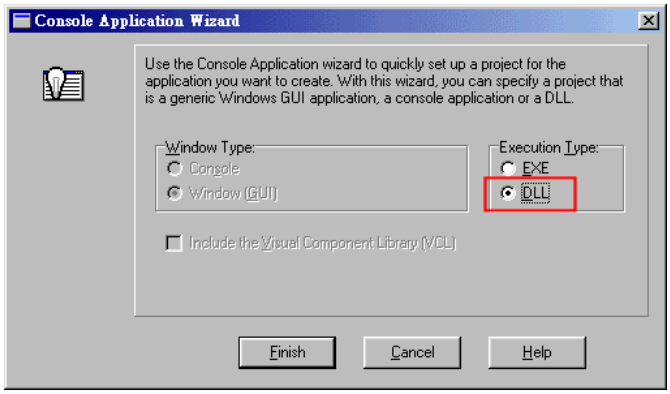
使用 C++Builder 來建立 DLL 並不是什麼難事，只需要按下幾下滑鼠的左鍵即可。在 C++Builder 下建立 DLLs 大致分成兩種方法，先按選單上的 File|New 後會出現 New Items 對話盒（Dialog）（圖四）：



圖四、New Item 對話盒中建立 DLL 的選項

1、選擇 New Items 對話盒中的 Console Wizard 選項按下 OK 後再選擇 DLL 即可。（圖五）
2、選擇 New Items 對話盒中的 DLL 那個選項按下 OK 即可。

方法一是建立一個標準的 DLL Project，不允許使用任何 VCL 類別，而方法二所打造出來的 DLL 則是可建立包含了 VCL 類別的 DLL（當然你也可以不使用 VCL 類別），而這兩個功能在 Visual C++中差可比擬的是 Win32 Dynamic-Link Library 與 MFC AppWizard(dll)。



圖五、New Item 對話盒中建立 DLL 的選項

按完 OK 或 Finish 後你會看到 DLL Project 與些許程式碼的產生，這段由 C++Builder 自動產生的程式碼中分兩大部分，第一部份是個很長一串的註解，最後就是所謂的 DLL 進入點。先來了解這一長串的註解，由方法一與方法二製造出來的 DLL Project 註解有點不相同，不過內容大致上差不多，內容如是說：如果我們的 DLL 內使用到了字串物件如：AnsiString，或是在輸出函式的參數或回傳值使用到長字串的話，就必須加入 MEMMGR.LIB 這個函式庫。另外，若我們在另一個模組（如 DLL）中使用了例如 new 或 GetMem 等方法來配置記憶體，而在不同的模組（如 EXE 應用程式）中使用了這塊記憶體或呼叫 FreeMem 等方法來釋放記憶體，則 MEMMGR.LIB 也是必須被加入的。此外還有一個值得注意的，就是 MEMMGR.LIB 必須加在所有要用到函式庫的最前頭，以便在其他函式庫之前優先載入並接手相關的記憶體維護。同時要記住的是若你使用了 MEMMGR.LIB 這個函式庫，那麼當你移交 DLL 或是應用程式時，你必須連同 BORLNDMM.DLL 一併移交給使用者。不過在這段聲明的倒數第二段中有提到，若要避免額外的檔案付給使用者（越多的檔案對使用這來說是一種負擔），你可以將有關字串的資料改由 char *或是 shortstring 來傳送，這樣可以不動用到 BORLNDMM.DLL 與 MEMMGR.LIB 來作記憶體的配置。另外，聲明的最後一段中有提到，若你在 Project\Options 裡頭的 Link 一頁勾選了 Use Dynamic RTL 一項時，就不須額外手動將 MEMMGR.LIB 給加到 Project 裡頭了，因爲 C++Builder 會自動幫你做這個動作。

緊接著咱們來看一下 DLL 的進入點

DLL 的進入點：DLLMain

以下是由方法二生產出來的 DLL 原始碼：

```
#0001.  //---------------------------------------------------------------------------
#0002.  #include <vcl.h>
#0003.  #pragma hdrstop
#0004.  //---------------------------------------------------------------------------
#0005.  int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
#0006.  {
#0007.      return 1;
#0008.  }
#0009.  //---------------------------------------------------------------------------
```

在 Win32 平台的程式設計中，明言規定 DLL 的進入點函式定義爲 DllMain，可是我們怎麼看到的是 DllEntryPoint 呢？這是 Borland 對於標準的 DLL 所動的一點點手腳，當然你若執意把 DllEntryPoint 給改成 DllMain 也是可以的，但是你要注意，當你更動 DllEntryPoint 時，C++Builder 工具列上頭的 Run 會被 Disable 喔！這是就是擺明了不給你編譯，即使你使用 Project\Build All 也會送給你錯誤訊息，不過你可以在 DllMain 函式之後加上#define WinMain 就可以編譯了，不過筆者並不建議在 C++Builder 下這麼用。

早在討論輸入輸出表時就已經知道若不考慮到各個編譯器間 DLL 的互相引用，我們大可把輸出函式這麼寫：

```
__declspec(dllexport) int MyFunc (void);
```

若要輸出整個類別則可以這樣寫：

```
class __declspec(dllexport) MyClass : public TObject{...};
```

但是注意，這若這樣寫僅可以在自己寫作 DLL 的編譯器中來使用這個 DLL 了，並沒辦法達到其他編譯器也可以使用的目的，大大的抹殺了軟體元件的構想，更何況 DLL 是採用模組化的設計。當然了！類別的輸出當然不在考慮範圍內，但若對函式的輸出咱們還是乖一點，加上 extern “C”來遏止 name mangling 對我們函式名稱所動的手腳：

```
extern “C” __declspec(dllexport) int MyFunc (void);
```

有了這些基礎知識後，咱們來來真正撰寫一個有用的 DLL 試試看吧！那要做些什麼呢？咱們就做個簡單的訊息視窗即可，怎麼做，其實說穿了就是把 Windows API MessageBox 給稍微包裝一下，不過還是用 MessageBox 這個函式來實作內容，讀者們可能覺得，這麼無聊還在包裝一次 MessageBox 函式幹嘛？這個嘛！不過做個測試嘛！

先使用 Consol Wizard 來產生 DLL Project，並將此 Project 儲存爲 ShowMsg.bpr，在 DLL 的進入點 DllEntryPoint 之後加上以下的程式碼：

```
#0001  extern “C” __declspec(dllexport) int ShowMsg(char *pText,HWND hWnd)
#0002  {
#0003    return MessageBox(hWnd,pText,"Information",MB_OK);
#0004  }
```

嗯！這樣一來咱們只要呼叫 `ShowMsg()`函式，並將要秀出來的文字與視窗代碼當作參數來傳即可。緊接著只需按下 `Ctrl+F9` 就開始編譯了並產生出 `ShowMsg.dll` 了。

使用 DLLs 中的函式—DLLs 的載入

知道了怎樣建立起一個 DLL 後，接著就是了解如何使用 DLL 裡頭所提供的函式的時候了，在使用這些函式之前還必須做個更重要的動作，就是將 DLL 給載入。載入 DLLs 的方法大致上可以分成兩個：Implicit Linking 與 Explicit Linking。

Implicit Linking

Implicitly Link（隱式聯結）又稱靜態載入，所謂靜態載入是指程式在聯結時期即與 DLLs 所對應的 import libraries 做靜態鏈結，於是可執行檔中便對所有的 DLL 函式都有一份重定位表格(relocation table)和待修正記錄(fixup record)。當程式被 Windows 載入器載入記憶體中，載入器會自動修正所有的 fixup records，而這個 fixup records 就是記錄由 DLL 中所有輸出資源的正確位址，也就是先前提到的 RVA 加上 DLL 被載入的基底位址，經過這樣的程序動態聯結便順利產生。也就是說，程式開始執行時，會用靜態載入方式所使用到的 DLLs 都載入到行程的記憶體裡。先來看看靜態載入放是的優點：

1、靜態載入方式所使用到的這個 DLL 會在應用程式執行時載入，然後就可以呼叫出所有由 DLL 中匯出的函式，就好像是包含在程式中一般。
2、動作較爲簡單，載入的方法由編譯器負責處理，咱們不須動腦筋。

而缺點是：

1、當這個程式靜態載入方式所使用到的這個 DLL 不存在時，這個程式在開始時就出現無法找到 DLL 的訊息而導致應用程式無執行。
2、編譯時需要加入額外的 import library。
3、若是要載入的 DLLs 一多，載入應用程式的速度會便慢。
4、若遇到不同品牌的 C++編譯器時，靜態載入可就沒有這麼簡單處理了，因爲當函式經過 Calling Conventions 的處理後，若要使用其他品牌編譯器所致造出的 DLL 須得大動干戈才行。

Implicit Linking 範例：

以先前建立的 ShowMsg.DLL 爲例子，我們已知這個 DLL 僅輸出一個函式：ShowMsg，且知道這個函式的原始定義：

extern “C” __declspec(dllexport) int ShowMsg(char *pText,HWND hWnd);

因此，若我們要載入這個函式則必須在應用程式中加入此輸入函式的宣告：

extern “C” __declspec(dllimport) int ShowMsg(char *pText,HWND hWnd);

此外，還要加上這個 DLL 的 import library File，要產生 import library 的方法有兩個：

1、Project\Options 的 Linker 中的 Geretate import library 勾選，在正常情況下，預設值是勾選起來的。

2、若不小心把 lib 檔案給刪除掉了，也可以利用 implib.exe 這個 C++Builder 所附上的工具來產生 lib 檔，implib 是文字模式下的程式，因此必須到文字模式下使用，以我們現在的例子來說，使用方式爲：implib ShowMsg.lib ShowMsg.dll，這樣就會產生 ShowMsg.lib 檔了。

緊接著就是加入這個 lib 檔到咱們的 Project 裡頭，可以使用【Project\Add to Project…】來加入 lib 檔。如此一來就可以在應用程式的任何地方使用 ShowMsg 函式了。

不過在此我們發現一個小問題，若每次要匯入 DLL 裡頭的函式，還必須把函式的原始定義給抄過來（雖然說複製—貼上這個動作很簡單），但有沒有更好的辦法呢？有的，咱們可以在 DLL 原始碼的 Header File 裡動點手腳，讓要使用 DLL 的應用程式只需 include 這個 Header File 就可以了，怎麼做呢？就是使用前置處理符號，若是使用 Borland C++或是 Borland C++Builder 來編譯 DLL 都必須加上#define __DLL__這個宣告，但是我們從 C++Builder 所幫我建立的 Project 裡面看不到這個，原因是 C++Builder 已經在 Make file 裡幫我們加上-WD 這個編譯參數來達成#define __DLL__所要的目的了。所以我們可以把 ShowMsg.h 給改成：

程式列表一、ShowMsg.h

```
#0001  //---------------------------------------------------------------------------
#0002  #ifndef ShowMsgH
#0003  #define ShowMsgH
#0004  //---------------------------------------------------------------------------
#0005  #ifdef __DLL__
#0006  #define DLLAPI  extern "C" __declspec(dllexport)
#0007  #else
#0008  #define DLLAPI  extern "C" __declspec(dllimport)
#0009  #endif
#0010
#0011  DLLAPI int ShowMsg(char *pText,HWND hWnd);
#0012
#0013  #endif
```

程式列表二、ShowMsg.cpp

```
#0001  #include <windows.h>
#0002  #pragma hdrstop
#0003  #include <condefs.h>
#0004  USEFILE("ShowMsg.h");
#0005  //---------------------------------------------------------------------------
#0006  #pragma argsused
#0007  int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
#0008  {
#0009    return 1;
#0010  }
#0011  //---------------------------------------------------------------------------
#0012  int ShowMsg(char *pText,HWND hWnd)
#0013  {
#0014    return MessageBox(hWnd,pText,"Information",MB_OK);
#0015  }
```

這樣一來，只需要在應用程式裡加入這個 Header File 並將 ShowMsg.lib 檔給加入這個 Project 裡頭，就大功告成了

Explicit Linking

而所謂 Explicitly link（顯式聯結）又稱動態載入，若是使用動態載入就是需要時才載入 DLL，然後在使用

過後即釋放 DLL，嗯！似乎是很不錯的選擇，這種方法的優點有：

1、DLL 只要需要時才會載入到記憶體中，可以更有效的使用記憶體。

2、應用程式載入的速度較使用隱式鏈結時快，因爲當程式開始載入時並不需要把 DLL 給載入到行程中。

3、編譯時不須額外的 import library 檔。

4、讓我們可以更清楚 DLL 的載入流程。

但不光只有優點也是點缺點的，缺點就是必須寫多一點程式碼。首先，必須使用 LoadLibrary 這個 Windows API 來手動載入 DLL，並使用 GetProcessAddress 來取得所要使用的函式的函式指標，最後不需要用到此 DLL 時使用 FreeLibrary 將 DLL 釋放。所以，在學會動態載入 DLL 時，必須先知道函式指標的用法。

Explicit Linking 範例：

咱們就拿一個常見的 DLL—控制 TWAIN32 界面的 DLL：『Eztw32.dll』來作動態載入的示範：eztw32.dll 裡頭有四十多個輸出函式可以使用，不過只有以下這四個函式是我們所要用到的：

1、void __stdcall TWAIN_SelectImageSource(HWND hwnd);
功能：用來選擇所要使用的 TWAIN 介面裝置

2、int __stdcall TWAIN_AcquireToClipboard(HWND hwnd, unsigned int pixmask);
功能：經由 TWAIN 介面將資料置放到剪貼簿中

3、int __stdcall TWAIN_LoadSourceManager(void);
功能：呼叫 TWAIN 介面裝置程式

4、int __stdcall TWAIN_UnloadSourceManager(void);
功能：關閉 TWAIN 介面裝置程式

再度使用 tdump 來觀察 eztw32.dll 裡頭的輸出函式表：

```
Exports from EZTW32.dll
  50 exported name(s), 50 export addresse(s).  Ordinal base is 1.
    Ordinal RVA      Name
    ------- --------  ----
    0000    00001000  DllMain
    ...
    0003    000012c0  TWAIN_AcquireToClipboard
    ...
    0029    00001650  TWAIN_LoadSourceManager
    ...
    0038    000010a0  TWAIN_SelectImageSource
    ...
    0046    00001980  TWAIN_UnloadSourceManager
    ...
```

先確認函式的名稱以及函式的序號，等一下會用到。

接著就是開始將 DLL 載入了，Win32 API 有兩個函式提供了將 DLL 載入的功能，分別是 LoadLibrary 與 LoadLibraryEx。通常都使用 LoadLibrary，先看看 LoadLibrary 的原始定義：

```
HINSTANCE LoadLibrary(
  LPCTSTR lpLibFileName  // address of filename of executable module
);
```

嗯！只需要傳入檔案名稱即可，若以我們所要使用的 Eztw32.dll 爲例：

```
HINSTANCE hDLL;
hDll = LoadLibrary("Eztw32.dll");
```

若我們沒有指定副檔名，則自動會以『.DLL』爲副檔名，或許會感到納悶，那我還沒有指定 DLL 的路徑啊！正確的做法應該是要指定路徑的，但是天曉得使用者會把 DLL 給放到哪裡去呢？所以使用 LoadLibrary 這個函式時，若參數中沒有指明路徑，系統會依特定的次序來找尋 DLL 的存在與否，若不存在則 LoadLibrary 函式則會回傳 NULL，以下就是搜尋次序：

1、被執行的應用程式所存在的路徑。

2、目前的目錄。

3、Windows 系統目錄，對 Windows 95/98 說是 Windows\System，而 Windows NT 則是 Winnt\System32。目錄名稱可以使用 GetSystemDirectory 這個 API 來取得。

4、Windows 目錄。目錄名稱可以使用 GetWindowsDirectory 這個 API 來取得。

5、最後由設定的 PATH 環境變數來尋找。

這也就是爲什麼我們的 Windows\System 目錄下有這麼多 DLL 的原因之一了。

把 DLL 載入記憶體後最重要的工作就是把函式指標指向函式在記憶體中正確的位址，要做到這個動作得透過 GetProcAddress 這個 API 來幫忙：

```
FARPROC GetProcAddress(
  HMODULE hModule,    // handle to DLL module
  LPCSTR lpProcName   // name of function
);
```

GetProcAddres 函式的第一個參數是經由 LoadLibrary 所取得的 DLL 的 Handle，而第二個參數是函式的名稱或是函式的輸出序號經由函式名稱取得函式的指標，以 eztw32.dll 中的 TWAIN_SelectImageSource 函式爲例，應由函式名稱取得函式的位址的方法爲：：

```
GetProcAddress(hDLL,"TWAIN_SelectImageSource");
```

若經由函式輸出序號取得函式的位址則爲：

```
GetProcAddress(hDLL, MAKEINTRESOURCE (39));
```

在此要注意序號的起始值是 1 不是 0，經由 tdump 所列出來的 Ordinal 是由起始值開始的位移植，而 Ordinal Base 爲 1，因此 TWAIN_SelectImageSource 的序號是 39 而不是 38。

接著在此先複習一下函式指標的使用方式，一樣 TWAIN_SelectImageSource 函式爲例，

TWAIN_SelectImageSource 函式的原始定義爲：

```
void __stdcall TWAIN_SelectImageSource(HWND hwnd);
```

那就可以用：

```
void (__stdcall *TWAIN_SelectImageSource)(HWND hwnd);
```

來宣告 TWAIN_SelectImageSource 爲一個函式指標，之後再用：

```
TWAIN_SelectImageSource = (void (__stdcall *)(HWND hwnd))
                    GetProcAddress(hDLL,"TWAIN_SelectImageSource");
```

來取得函式的位址，之後就在程式中若要使用 TWAIN_SelectImageSource 這個函式就可以像是靜態載入般使用了。但這樣似乎得打蠻多字的，偷懶的我通常都使用另一個方法，就是使用 typedef 來自訂型別。方法如下：

```
typedef void (__stdcall *_TWAIN_SelectImageSource)(HWND hwnd);
_TWA IN_SelectImageSource  TWAIN_SelectImageSource;
TWAIN_SelectImageSource = (_TWAIN_SelectImageSource)
                          GetProcAddress(hDLL, "TWAIN_SelectImageSource");
```

先使用 typedef 把_TWAIN_SelectImageSource 給定義成一個特殊的型別，之後就可以直接引用，的確是可以少打點字。

最後當 DLL 裡頭的函式不再需要使用時，咱們就得使用 FreeLibrary 將 DLL 從記憶體裡頭卸下來：

```
BOOL FreeLibrary(
    HMODULE hLibModule   // handle to loaded library module
);
```

使用方法很簡單只需將 LoadLibrary 所傳回來的 DLL Handle 當參數傳給 FreeLibrary 傳入即可。

懂得這些動態載入 DLL 的流程後，就可以實際動手來做做看。筆者發現 TWAIN32 這些功能實在很適合包裝成一個物件，當物件誕生時，立即自動去 LoadLibrary 並將函式指標的位址給連結起來，當這個物件被摧毀時，就自動去 FreeLibrary，嗯！似乎不錯，不過詳細的做法就不多做解釋了，相信讀者看了下面的程式

列表應該就懂了。

程式列表三、CTWAIN.h：

```
#0001  //---------------------------------------------------------------------------
#0002  #ifndef CTWAINH
#0003  #define CTWAINH
#0004  #include <clipbrd.hpp>
#0005  //---------------------------------------------------------------------------
#0006  //用 typedef 自訂函式指標的型別
#0007  typedef int  (__stdcall *_TWAIN_AcquireToClipboard)
#0008              (HWND hwnd, unsigned int pixmask);
#0009  typedef int  (__stdcall *_TWAIN_LoadSourceManager)
#0010              (void);
#0011  typedef void (__stdcall *_TWAIN_SelectImageSource)
#0012              (HWND hwnd);
#0013  typedef int  (__stdcall *_TWAIN_UnloadSourceManager)
#0014              (void);
#0015  class CTWAIN
#0016  {
#0017    public:
#0018         __fastcall CTWAIN(void);
#0019         __fastcall ~CTWAIN(void);
#0020         void __fastcall SelectImageSource(HWND hWnd);//選擇 TWAIN32 設備
#0021         void __fastcall Acqure(HWND hWnd);//經由 TWAIN32 設備取得資料
#0022         Graphics::TPicture *Picture;
#0023
#0024    protected:
#0025
#0026    private:
#0027         //宣告函式指標
#0028         _TWAIN_AcquireToClipboard   TWAIN_AcquireToClipboard;
#0029         _TWAIN_LoadSourceManager    TWAIN_LoadSourceManager;
#0030         _TWAIN_SelectImageSource    TWAIN_SelectImageSource;
#0031         _TWAIN_UnloadSourceManager  TWAIN_UnloadSourceManager;
#0032         HINSTANCE  hDLL;
#0033  };
#0034  //---------------------------------------------------------------------------
#0035  extern bool __stdcall CheckTWAINDLL();//檢驗 DLL 檔案是否存在的函式
#0036  #endif
```

程式列表四、CTWAIN.cpp：

```
#0001  //---------------------------------------------------------------------------
#0002  #include <vcl.h>
#0003  #pragma hdrstop
#0004
#0005  #include "CTWAIN.h"
#0006  //---------------------------------------------------------------------------
#0007  #pragma package(smart_init)
#0008  __fastcall CTWAIN::CTWAIN()
#0009  {
#0010    hDLL = ::LoadLibrary("Eztw32.dll");//載入 DLL 到行程的記憶體內
#0011    //取得所需函式的函式指標
#0012    TWAIN_AcquireToClipboard = (_TWAIN_AcquireToClipboard)
#0013                               ::GetProcAddress(hDLL,"TWAIN_AcquireToClipboard");
#0014    TWAIN_LoadSourceManager = (_TWAIN_LoadSourceManager)
#0015                               ::GetProcAddress(hDLL,"TWAIN_LoadSourceManager");
#0016    TWAIN_SelectImageSource = (_TWAIN_SelectImageSource)
#0017                               ::GetProcAddress(hDLL,"TWAIN_SelectImageSource");
#0018    TWAIN_UnloadSourceManager = (_TWAIN_UnloadSourceManager)
#0019                               ::GetProcAddress(hDLL,"TWAIN_UnloadSourceManager");
#0020    Picture = new Graphics::TPicture;
#0021  }
#0022  //---------------------------------------------------------------------------
#0023  __fastcall CTWAIN::~CTWAIN()
#0024  {
#0025    delete Picture;
#0026    ::FreeLibrary(hDLL);//從記憶體中釋放 DLL
#0027  }
#0028  //---------------------------------------------------------------------------
#0029  void __fastcall CTWAIN::SelectImageSource(HWND hWnd)//選擇 TWAIN32 設備
#0030  {
```

```
#0031     TWAIN_SelectImageSource(hWnd);
#0032     //使用 DLL 裡頭的 TWAIN_SelectImageSource 函式
#0033   }
#0034   //---------------------------------------------------------------------------
#0035   void __fastcall CTWAIN::Acqure(HWND hWnd)//經由 TWAIN32 設備取得資料
#0036   {
#0037     if (TWAIN_LoadSourceManager() > 0)
#0038     //使用 DLL 裡頭的 TWAIN_LoadSourceManager 函式
#0039     {
#0040          if (TWAIN_AcquireToClipboard(hWnd,0)>0)
#0041          //使用 DLL 裡頭的 TWAIN_AcquireToClipboard 函式
#0042          {
#0043               Clipboard()->Open();
#0044               if (Clipboard()->HasFormat(CF_PICTURE))
#0045                    Picture->Assign(Clipboard());
#0046               Clipboard()->Clear();
#0047               Clipboard()->Close();
#0048          }
#0049          TWAIN_UnloadSourceManager();
#0050          //使用 DLL 裡頭的 TWAIN_UnloadSourceManager 函式
#0051     }
#0052   }
#0053   //---------------------------------------------------------------------------
#0054   bool __stdcall CheckTWAINDLL()//檢驗 DLL 檔案是否存在的函式
#0055   {
#0056     HANDLE hDll = ::LoadLibrary("EZTW32.DLL");
#0057     if(hDll == NULL)
#0058     {
#0059          ::MessageDlg("Error! Could not found EZTWtw32.dll",
#0060                       mtInformation,TMsgDlgButtons()<<mbOK, 0);
#0061          return false;
#0062     }
#0063     ::FreeLibrary(hDll);
#0064     return true;
#0065   }
#0066   //---------------------------------------------------------------------------
```

當建構好這個 CTWAIN 類別後，若我們要使用這剛剛建好的類別，我們可以寫成兩個函式：

1、void SelectSource(HWND hWnd);

選擇 TWAIN32 設備的來源

2、void Acquire(HWND hWnd);

經由選定的 TWAIN32 設備來源將資料取得

```
#0001   //---------------------------------------------------------------------------
#0002   void SelectSource(HWND hWnd)//選擇 TWAIN32 設備的來源
#0003   {
#0004     if (CheckTWAINDLL())
#0005     {
#0006          CTWAIN TWAIN;
#0007          TWAIN.SelectImageSource(hWnd);
#0008     }
#0009   }
#0010   //---------------------------------------------------------------------------
#0011   void Acquire(HWND hWnd)//經由選定的 TWAIN32 設備來源將資料取得
#0012   {
#0013     if (CheckTWAINDLL())
#0014     {
#0015          CTWAIN TWAIN;
#0016          TWAIN.AcqureScanner(hWnd);
#0017          if (TWAIN.Picture->Graphic != NULL)
#0018          {
#0019               //Do the processing that you want to do with the TWAIN.Picture
#0020               TWAIN.Picture->Graphic = NULL;
#0021          }
#0022     }
#0023   }
#0024   //---------------------------------------------------------------------------
```

相形之下，似乎靜態載入的方式方便多了，不用下這麼多工夫在手動載入 DLL 並手動分派各函式的實際位址，若遇到一大堆輸出函式豈不光 Keyin GetProcAddress 那一段就得花上許多時間。但若用到不同品牌的 C++編譯器所產生的 DLL 時，採用動態載入加上以函式輸出序號來取得函式指標的方式，僅需利用 TDUMP

製造初一張序號表再加上勤勞的打字即可無須再動什麼腦筋。稍後會提到怎麼樣在 C++Builder 下使用靜態載入來載入 Visual C++所編譯出來的 DLL。這樣相形之下，反而又會覺得用動態載入方便些。

Calling Conventions

因爲不同的語言間有不同的傳遞參數的方法，而 C/C++編譯器爲了能夠使用由其他語言開發出來的函式庫加上了這些參數傳遞方式不同的方式稱爲 calling conventions（呼叫慣例），如在 C++Builder 裡頭常常看到的__fastcall；一般常用的呼叫慣例有以下四種：

呼叫慣例	說明	參數傳遞方式
__cdecl	傳遞的參數由呼叫函式清除。	由右往左的次序將參數傳遞到堆疊之中。
__stdcall	傳遞的參數由被呼叫函式清除。一般使用於 Win32 的標準呼叫，在 DLLs 間的呼叫大部分都使用__stdcall。	由右往左的次序將參數傳遞到堆疊之中。
__fastcall	傳遞的參數由被呼叫函式清除。在 C++Builder 中__fastcall 爲 VCL 元件使用的內定呼叫慣例。	**in C++Builder：** 由左往右的次序傳遞參數，第一個參數由 EAX 傳遞，第二個由 EDX 傳遞，第二個由 ECX 傳遞，其餘操超過的參數再交由堆疊傳遞。 **in Visual C++：** 由左往右的次序傳遞參數，第一個參數由 ECX 傳遞，第二個由 EDX 傳遞，其餘操超過的參數再交由堆疊傳遞。
__pascal	傳遞的參數由被呼叫函式清除，在 Windows 3.1 時期__pascall 爲標準用法，但到了 Windows 95/NT 後以鮮少使用。	由左往右的次序將參數傳遞到堆疊之中。Visual C++已經不支援__pascal 此呼叫方式。因此在此不做__pascal 的討論。

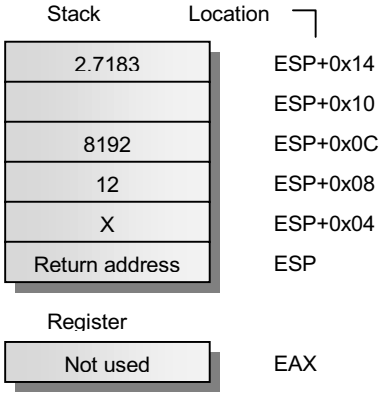
但光是說說很難了解到這些參數傳遞方式有何異同，以下函式分別使用__cdecl、__stdcall 與__fastcall 三種呼叫慣例當做範例：

```
void calltype MyFunc(char c, shorty s, int i , double f);
```

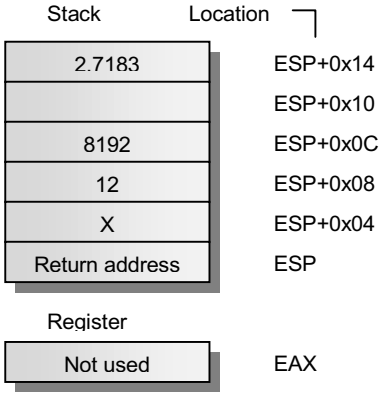
當我使用這個函式：

```
MyFunc("x",12,8192,2.7183);
```

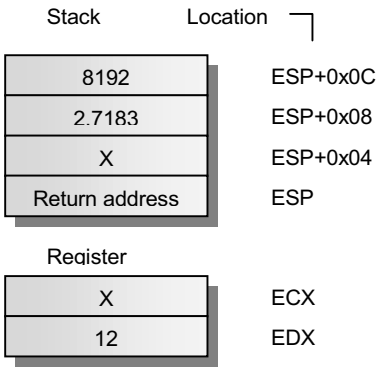
時會被編譯器編譯成如以下四圖：



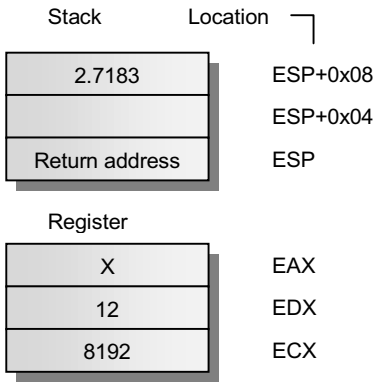
圖六、__cdecl 呼叫慣例



圖七、__stdcall 呼叫慣例



圖八、**Microsoft Visual C++** __fastcallc 呼叫慣例



圖九、**Borland C++ Builder** __fastcallc 呼叫慣例

由圖八與圖九中可以看出__fastcall 在兩個編譯器中有顯著的不同，C++Builder 使用了三個暫存器來存放參數，讓傳遞的速度更爲加快，這也就是 VCL 類別中的預設呼叫慣例爲__fastcall 的原因。暫且撇開__fastcall 的不同，呼叫慣例造成的還不只這一樣差異，還有著與 name mangling 有點類似的麻煩，就是函式的名稱更動問題。當你使用不同的呼叫慣例時，編譯器還是對動點手腳，動什麼手腳，筆者用以下的範例做觀察，先定義四個函式分別使用__fastcall、__stdcall 與__cdecl 與不指定四種呼叫慣例：

```
#0001   #define DLLEXP extern "C" __declspec(dllexport)
#0002   DLLEXP int MyFunc_Default(char *c,int X)
#0003   {
#0004     return X;
#0005   }
#0006   DLLEXP int __fastcall MyFunc_Fast(char *c,int X)
#0007   {
#0008     return X;
#0009   }
#0010   DLLEXP int __stdcall MyFunc_Std(char *c,int X)
#0011   {
#0012     return X;
#0013   }
#0014   DLLEXP int __cdecl MyFunc_Cdecl(char *c,int X)
#0015   {
#0016     return X;
#0017   }
```

定義好後分別使用 C++Builder 與 Visual C++將這四個函式編譯成 DLL 檔，編譯完成後使用 tdump 與 impdef 來觀察這些函式的輸出結果：（IMPDEF 的用法爲 IMPDEF def_file dll_file）

```
Exports from VC6Test.dll
  4 exported name(s), 4 export addresse(s).  Ordinal base is 1.
    Ordinal RVA      Name
    ------- -------- ----
    0000    00001019 MyFunc_Fast::
    0001    0000100f MyFunc_Cdecl
    0002    00001014 MyFunc_Default
    0003    0000100a _MyFunc_Std@8
```

圖十、使用 tdump 來觀察 VC6TEST.DLL 裡的輸出函式表

```
LIBRARY     VC6TEST.DLL
EXPORTS
    @MyFunc_Fast@8      =@MyFunc_Fast               @1
    MyFunc_Cdecl        @2
    MyFunc_Default      @3
    _MyFunc_Std@8       =_MyFunc_Std                @4
```

圖十一、使用 impdef 來觀察 VC6TEST.DLL 裡的輸出函式命名狀況

```
Exports from BCB3Test.dll
  4 exported name(s), 4 export addresse(s).  Ordinal base is 1.
    Ordinal RVA      Name
    ------- -------- ----
    0000    00001380 @MyFunc_Fast
    0001    00001394 MyFunc_Std
    0002    000013a0 _MyFunc_Cdecl
    0003    00001378 _MyFunc_Default
```

圖十二、使用 tdump 來觀察 BCB3TEST.DLL 裡的輸出函式表

```
LIBRARY    BCB3TEST.DLL
EXPORTS
   @MyFunc_Fast            @1
   MyFunc_Std              @2
   _MyFunc_Cdecl           @3
   _MyFunc_Default         @4
```

圖十三、使用 tdump 來觀察 BCB3TEST.DLL 裡的輸出函式表

）

經由上述簡單的實驗可以將兩者的差異列出，並且可以找出編譯器預設的呼叫慣例：

呼叫慣例	原始函式	Borland C++Builder	Microsoft Visual C++
__cdecl	MyFunc_cdcel	_MyFunc_cdcel	MyFunc_cdcel
__stdcall	MyFunc_std	MyFunc_std	_MyFunc_std@8
__fastcall	MyFunc_fast	@MyFunc_fast	@MyFunc_fast@8
—	MyFunc_default	_MyFunc_default	MyFunc_default
預設呼叫慣例	—	__cdecl	__cdecl

由表上可以看出兩者間的差異還不少呢！而爲什麼要討論到這一點呢？因爲接著就要討論到如何拿 Visual C++所編譯的 DLL 到 C++Builder 裡頭使用。

在 Borland C++Builder 下使用 Microsoft Visual C++所編譯的 DLLs

若已經解決了 name mangling 與 calling convention 的理想狀況下，由 C++Builder 下來呼叫 Visual C++所編譯出來的 DLLs 應該不是難事才對。但不幸的，只對了一半，怎麼說，幸運的那一半是，咱們可以利用先前提過的『動態載入』方式來載入 DLLs 中的函式，即使因爲 calling convention 的問題導致函式名稱在編譯後會被更動，但是只要知道函式的輸出序號就可以照樣載入；不幸的那一半是若採用『靜態載入』的方法就又會遇上了個大難題：Borland 與 Microsoft 所使用的 OBJs 檔案格式不相同；Borland 採用 Intel 所訂定的 OMF（Object Module Format）格式而 Microsoft 採用 COFF（Common Object File Format）格式，因此若要拿 Visual C++所編譯的 DLLs 與 LIBs 來使用，僅有 DLLs 能夠用而已，LIBs 毫無用武之地，但謝天謝地，Borland 提供了一個工具 IMPLIB.EXE，可以直接從任何編譯器所編譯出的 DLLs 裡頭將 OMF 格式的 LIBs 給製造出來。因此製造 C++Builder 相容的 OMF 格式 LIBs 不算是個大問題了。但對於 Calling Convention 所產生的問題就比較麻煩些，接下來咱們就來討論如何使用靜態載入來載入 Visual C++所製造出來的 DLLs。

要將 Visual C++所製造出了 DLLs 搬到 C++Builder 來用大致上分三個步驟：

A、檢驗輸出函式的設呼叫慣例
由先前呼叫慣例的實驗裡看得出來，兩種編譯器的預設呼叫慣例皆爲__cdecl，而__cdecl 與__stdcall 的參數傳遞方式兩個編譯器也相同，但是在 C++Builder 下__fastcall 爲 VCL 的標準呼叫慣例，而且爲了加速參數的傳遞，參數傳遞的方法與 Visual C++不同，若是硬搬到 C++Builder 來使用，恐有問題出現；因此若要拿

Visual C++所編譯的 DLL 來使用，切記只能使用__cdecl 與__stdcall 這兩種呼叫慣例。

B、查驗經過編譯器編譯後的函式名稱

由呼叫慣例的實驗結果裡看出，__cdecl 與__stdcall 在兩個編譯器下所編譯出的正式名稱稍有不同，以一個簡單的 void MyFunction(void); 爲例：

呼叫慣例	**Borland C++Builder**	**Microsoft Visual C++**
__cdecl	_MyFunction	MyFunction
__stdcall	MyFunction	_MyFunction@4

我們必須檢查看看哪些函式是使用__stdcall 而哪些函式是使用__cdecl，接下來下一個步驟就是轉換這些名稱，將名稱由 C++Builder 不認得變成認得。

C、製作 OMF 格式的 LIBs

由於經過編譯器處理過後的函式名稱已經與原先函式名稱不相同了，因此若直接轉換成 LIBs 檔也無啥效用，必須動點手腳。先前已經學過 IMPDEF 的使用方法，先前是用在觀察輸出函式，現在也是，但還多了動手腳的部份。

利用先前測試的那個 VC6TEST.DLL 來製造 DEF 檔案：（__fastcall 的呼叫慣例部份記得要先除去）

IMPDEF VC6TEST.DEF VC6TEST.DLL

製造出來的 DEF 檔：

```
LIBRARY     VC6TEST.DLL
EXPORTS
    MyFunc_Cdecl     @1
    MyFunc_Default   @2
    _MyFunc_Std@8    =_MyFunc_Std @3
```

這時候我們就來動手腳，把 C++Builder 不認得給改承認得的。改成如下：

```
EXPORTS
    ;use this type of aliasing
    ;(Borland name)  =   (Name exported by Visual C++)

    MyFunc_Std       =   _MyFunc_Std@8
     _MyFunc_Cdecl   =   MyFunc_Cdecl
     _MyFunc_Default     =   MyFunc_Default
```

最後再將 DEF 檔給還原成 LIB 檔，什麼？你有沒有說錯把 DEF 這個文字檔變回 LIB 檔？是的，其實靜態載入所需要用到的 LIB 檔案只是個函式表格罷了，並沒有真正函式內容在裡頭，因此我們可以經由動過手腳的 DEF 檔給還原成 LIB 檔：

```
IMPLIB    VC6TEST.lib VC6TEST.def
```

動完手腳後不忘還要去檢查一下是否函式輸出，可以使用 TLIB 來觀察最後的 test.lib 是否正確輸出：

```
TLIB VC6TEST.lib,VC6TEST.txt
```

```
Publics by module
_MyFunc_cdeclsize = 0
     _MyFunc_cdecl
_MyFunc_default   size = 0
     _MyFunc_default
MyFunc_std    size = 0
     MyFunc_std
```

可以由觀察 test.txt 來檢驗輸出的正確與否，在確定輸出名稱無誤後就可以直接拿進 C++Builder 做先前已經說明過的靜態載入測試了。我的測試方法爲建立一個 Console Application 來載入這個 DLL 做測試。

```
#0001  #pragma hdrstop
#0002  #include <condefs.h>
#0003
#0004  //---------------------------------------------------------------------------
```

```
#0005  USELIB("VC6test.lib");
#0006  //---------------------------------------------------------------------------
#0007  #pragma argsused
#0008  #define DLLIMP extern "C" __declspec(dllimport)
#0009  DLLIMP int MyFunc_default(char *c,int X);
#0010  DLLIMP int __stdcall MyFunc_std(char *c,int X);
#0011  DLLIMP int __cdecl MyFunc_cdecl(char *c,int X);
#0012
#0013  #include <stdio.h>
#0014  #include <conio.h>
#0015  int main(int argc, char **argv)
#0016  {
#0017     printf("int x = MyFunc_std(\"x\",123);\n");
#0018     int x = MyFunc_std("x",123);
#0019     printf("Result : x = %d\n",x);
#0020     printf("int y = MyFunc_cdecl(\"x\",2323);\n");
#0021     int y = MyFunc_cdecl("x",2323);
#0022     printf("Result : y = %d\n",y);
#0023     getch();
#0024     return 0;
#0025  }
```

輸出結果：

int x = MyFunc_std("x",123);
Result : x = 123
int y = MyFunc_cdecl("x",2323);
Result : y = 2323

這樣一來算大功告成，但在筆者的經驗中使用 Visual C++來寫作 DLL 供 Borland C++Builder 使用，有以下幾點是值得注意的：

1、抑制 name mangling 的行爲
一定要記住爲輸出函式加上 extern “C”來抑制 name mangling 的動手腳。
2、禁止輸出類別
爲什麼要禁止類別的輸出呢？因爲 Visual C++處理類別的方式與 C++Builder 有著很大的不同。所以少用爲妙。

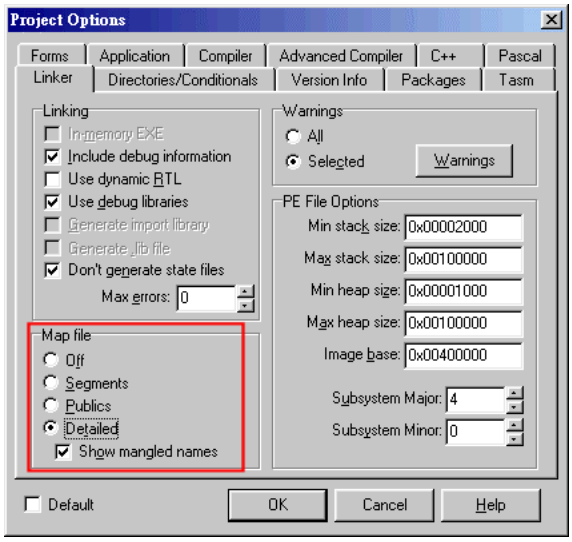
結語

這次筆者大多著墨於 DLLs 的使用，怎麼建立 DLLs 的方法卻只大略一題，因爲製造 DLLs 的方法有很多，也有很多的技巧，若把這些通通都給寫出來，可能這個主題就要變成連載小說般分成好幾期來刊載了，一期爲時一個月，討論這個主題的時間會拖得太久，且相差時間會過久，造成讀者的無法連貫，筆者只好忍痛割捨，不過筆者在此還是列出數本對於製作 DLLs 一題著墨相當詳細的書籍：

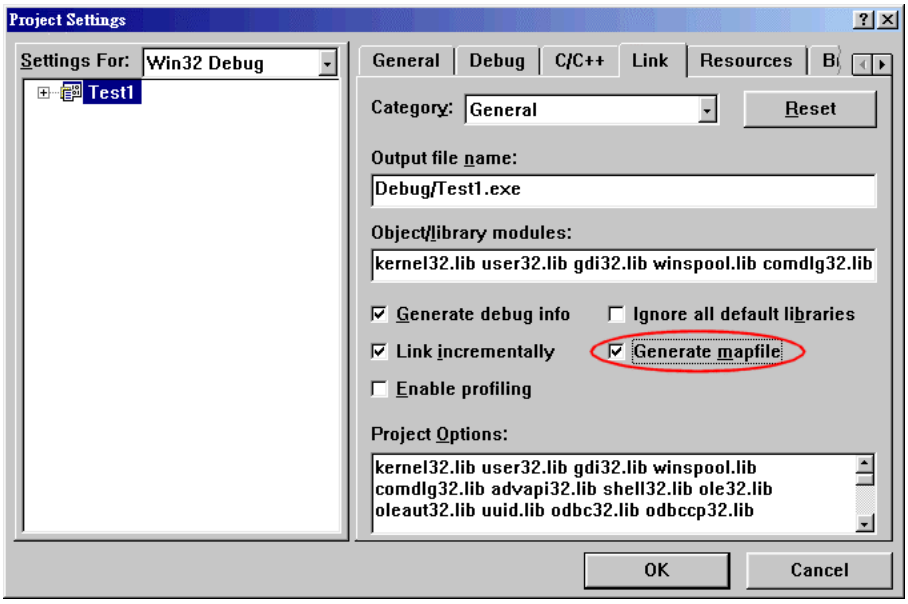
1、Adcanced Windows 3/E, Microsoft Press, Jeffrey Richter Chapter 12
2、Programming Windows 95, Microsoft Press, Charles Petzold Chapter 19
3、Multithreading Application in Win32, Addison Wesley, JimBeveridge, etc . Chapter 14
4、Borland C++Builder 3 Unleashed, Sams Publishing, Charlie Calvert, etc. Chapter 35

註 1、PE 格式的執行檔：PE，Portable Executable，是 Microsoft 設計用於其所有 Win32 作業系統（Win32s、Windows NT 及 Windows 95/98）的可執行檔格式。
註 2、在 C++Builder 下觀看編譯器產生的真正名稱方法：

選單上『Project/Options』裡的『Linker』一頁中的 Map File 選項中勾選 derailed：



註 3、在 Visual C++下觀看編譯器產生的真正名稱方法：
選單上『Project/Settings…』中『Link』一頁且 Category 爲 General 時將 Generate mapfile 選項勾選。



參考資料、

1. Windows 95 System Programmung SECRETS, IDG BOOKS, Matt Pietrek
2. Adcanced Windows 3/E, Microsoft Press, Jeffrey Richter
3. Programming Windows 95, Microsoft Press, Charles Petzold
4. Borland C++Builder 3 Unleashed, Sams Publishing, Charlie Calvert
5. High Preformance Borland C++Builder, Coriolis Group Books, Matt Telles