DLL Basics

Reference:

Programming Applications for Microsoft Windows Fourth Edition p.p. 675

Introduction

- All the functions in the Windows API are contained **in DLLs**
- The three DLLs
  - **Kernel32.dll**: managing memory, processes, threads
  - **User32.dll**: user-interface tasks
  - **GDI32.dll**: drawing image, display text

other DLLs

AdvAPI32.dll ➔ object security, registry manipulation
ComDlg32.dll ➔ common dialog boxes
ComCtl32.dll ➔ all of the common window controls

你可以下 dumpbin /exports User32.dll 看看到底他提供了哪些 function?

Introduction

- DLL 爲一堆 function 的集合,通常不包含
  - message-loop
  - 建立視窗
- DLL 的 function 可以被其他執行檔或 **DLL**呼叫
- DLL 的執行
  - 使用呼叫者 **thread** 的 **stack** 儲存 local variable
  - 記憶體配置, 使用呼叫者 thread的 virtual space 配置
  - 全區域變數與靜態變數:
    - Win98 → 爲每一個process 配置一份複製品
    - Win2000以後 ➔ 使用 copy-on-write 策略

要注意的事情

- 一個Process address space 中,可能包含
  - 執行檔 image + 數個 DLL modules
- 可能使用你 DLL 的 module :
  - 有些 modules 使用 C++ run-time library
  - 有些使用 static 版本的 library
  - 有些 modules 不使用 C++ library

要注意的事情

```
VOID EXEFunc() {
   PVOID pv = DLLFunc();
   // Access the storage pointed to by pv...
   // Assumes that pv is in EXE's C/C++ run-time heap
   free(pv);
}

PVOID DLLFunc() {
   // Allocate block from DLL's
   // C/C++ run-time heap
   return(malloc(100));
}
```

使用你DLL的人可能
使用的是 static 版本的 free

在你的 DLL 中,使用 **run-time** 版本的 malloc

比較好的策略
在 DLL 配置記憶體,在 DLL 釋放他

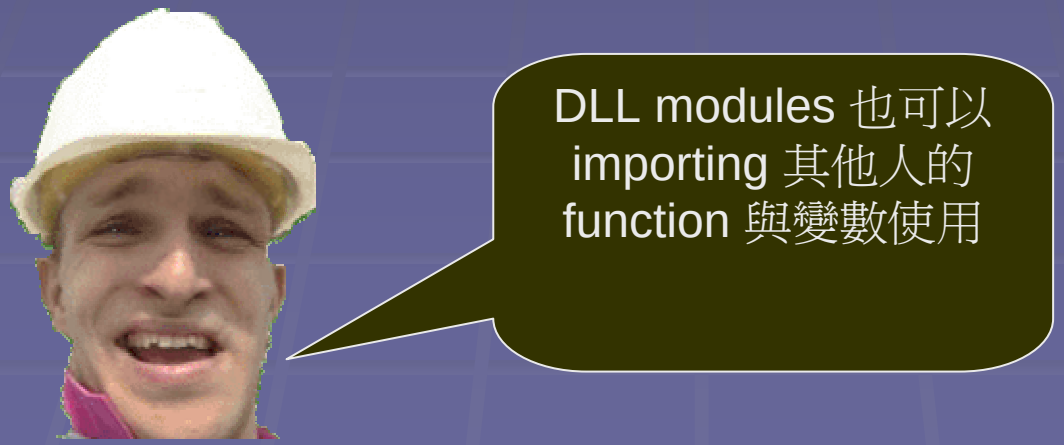
```
VOID EXEFunc() {
   PVOID pv = DLLFunc();
   // Access the storage pointed to by pv...
   // Assumes that pv is in EXE's C/C++ run-time heap
   DLLFreeFunc(pv);
}

PVOID DLLFunc() {
   // Allocate block from DLL's C/C++ run-time heap
   return(malloc(100));
}

BOOL DLLFreeFunc(PVOID pv) {
   // Free block from DLL's C/C++ run-time heap
   return(free(pv));
}
```

全部放在 DLL中處理

The Overall Picture

- 名詞的定義
  - Executable module ➔ 由 DLL importing 函式或變數來使用的 module
  - DLL modules ➔ exporting function 或變數 給 executable module使用的 modules

使用 DLL 的 Module



DLL 是如何被建立的
以及如何被應用程式 implicitly link

DLL export 的部分

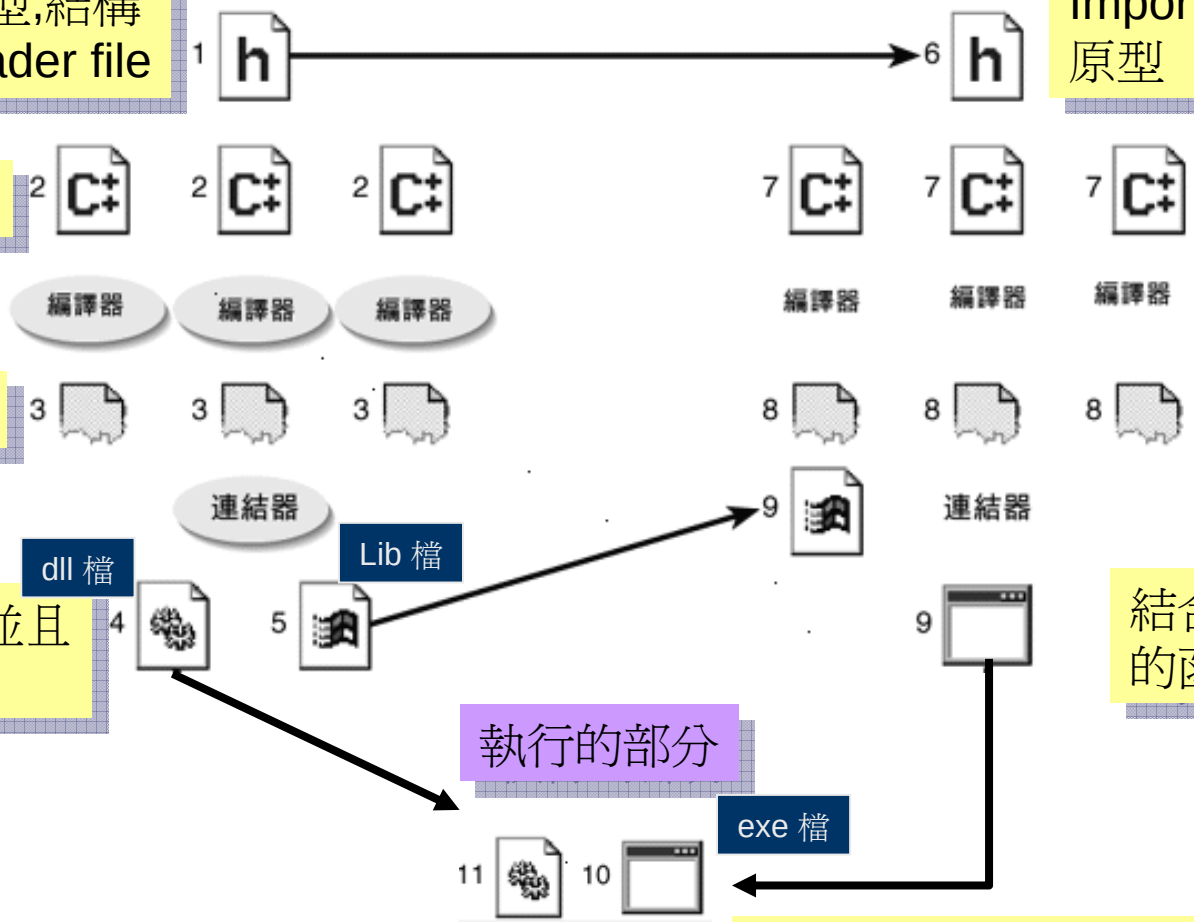
使用 DLL 的部分

export 函式原型,結構 Symbol 的 header file

Import 你要使用的函式原型

函式的實作

呼叫 DLL 函式



COFF 格式

產生 obj 檔

產生 obj

組合成 DLL 檔並且產生 lib 檔

結合 lib 解析出呼叫的函式 symbol

Loader 載入必要的 DLL 到process 的 address space

Loader 建立address space 給 .exe

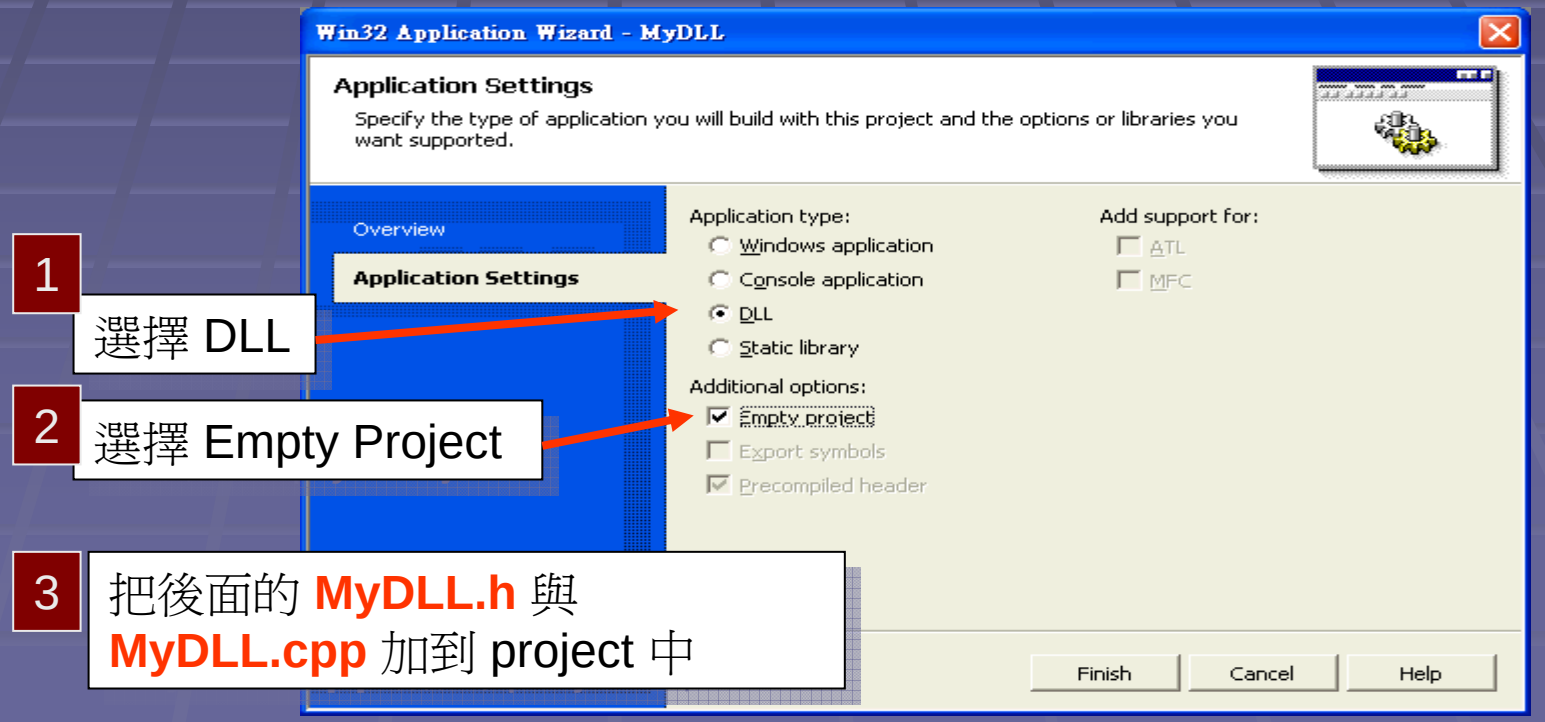
COFF 格式=Common Object File Format

建立一個最簡單的 DLL Module

- DLL 可以 export
  - Function, variable 或 C++ class

必須使用相同的編譯器才能 reference class

全區域變數: 影響抽象性, 且不易維護



1 選擇 DLL

2 選擇 Empty Project

3 把後面的 **MyDLL.h** 與 **MyDLL.cpp** 加到 project 中

Export Symbol

- 下面這個檔案,必須被散佈

當你要 export symbol時,使用這個

MyLib.h

```
#ifdef MYLIBExport
 #define MYLIBAPI extern “C” __declspec(dllexport)
#else
  #define  MYLIBAPI extern "C" __declspec(dllimport)
#endif

MYLIBAPI int g_nResult;
MYLIBAPI int Add(int nLeft, int nRight);
```

定義 exported 函式及變數的地方

當你要 import symbol時,使用這個

實作 export 函式

MyLib.h

```
#ifdef MYLIBExport
  #define MYLIBAPI extern "C" __declspec(dllexport)
#else
   #define  MYLIBAPI extern "C" __declspec(dllimport)
#endif
MYLIBAPI int g_nResult;
MYLIBAPI int Add(int nLeft, int nRight);
```

MyLibFile1.cpp

```
#include <windows.h>
#define MYLIBExport
#include "MyLib.h"

int g_nResult;
int Add(int nLeft, int nRight) {
   g_nResult = nLeft + nRight;
   return(g_nResult);
}
```

Include 標準的 windows 及 C++ run-time Library 標頭檔

定義目前是 export

Function 的 prototype

實作函式及變數的地方

extern "C" __declspec(**dllimport**) void myfunction() { … }

爲甚麼要加 extern “C” ?

- 因爲 C++ 編譯器會在 function 前面加上一些修飾 (**mangle**), 這樣會造成 C 程式無法 link 到正確的 function
- 加入 extern “C” function(); 可以要求編譯器不要加入 mangle
- 爲何 C++ 會對 function name 做 *name decoration* 呢?
  - Function overloading 與 c++ type-safe linkage

例如:

各家編譯器有不同的作法

```
void foo(int ,double);
Comeau C++ 會把 foo 改成 foo_Fid
 F ➔ function name
 I ➔ 第一個參數是 int;  d ➔ 第二個參數是 double
```

參考資料: http://www.comeaucomputing.com/techtalk/#externc

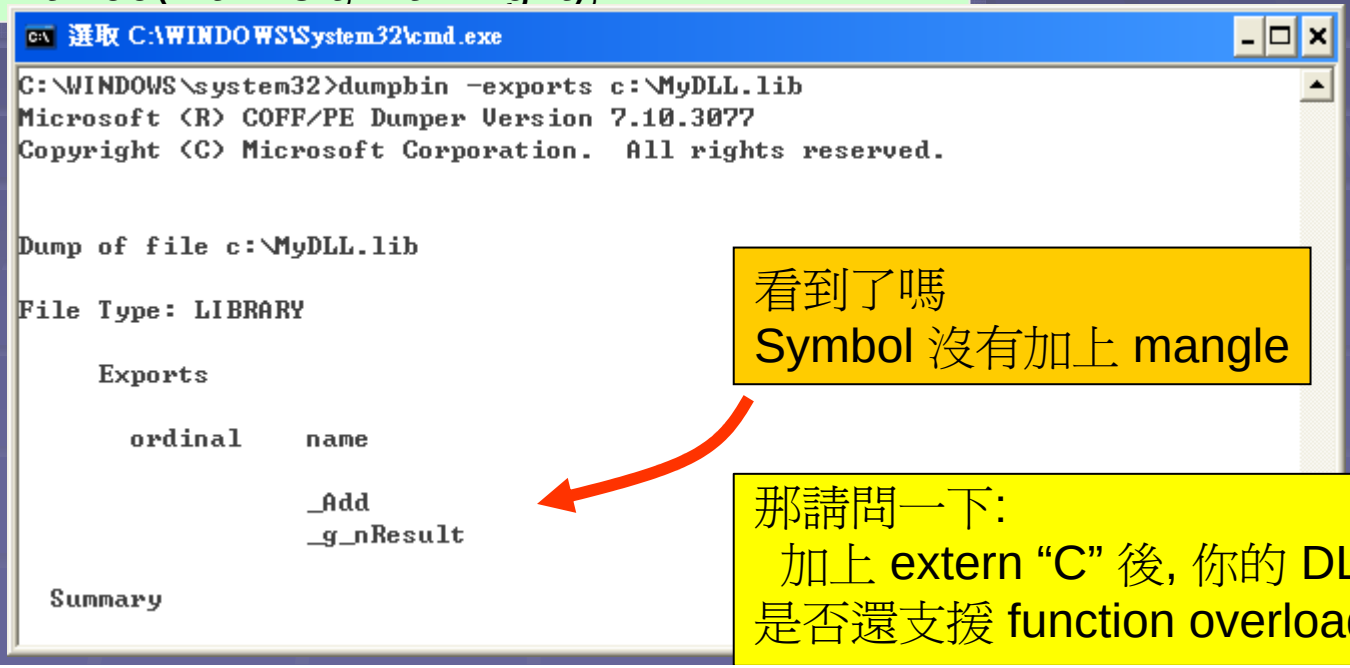
我們來看看差別在哪裡!

如果你的 MyLib.h 的內容如下

使用 “C” 的方式命名

```
#ifdef MYLIBExport
   #define MYLIBAPI extern “C” __declspec(dllexport)
#else
   #define  MYLIBAPI extern "C" __declspec(dllimport)
#endif
MYLIBAPI int g_nResult;
MYLIBAPI int Add(int nLeft, int nRight);
```

利用 dumpbin /exports 察看 MyDll.lib 結果



看到了嗎
Symbol 沒有加上 mangle

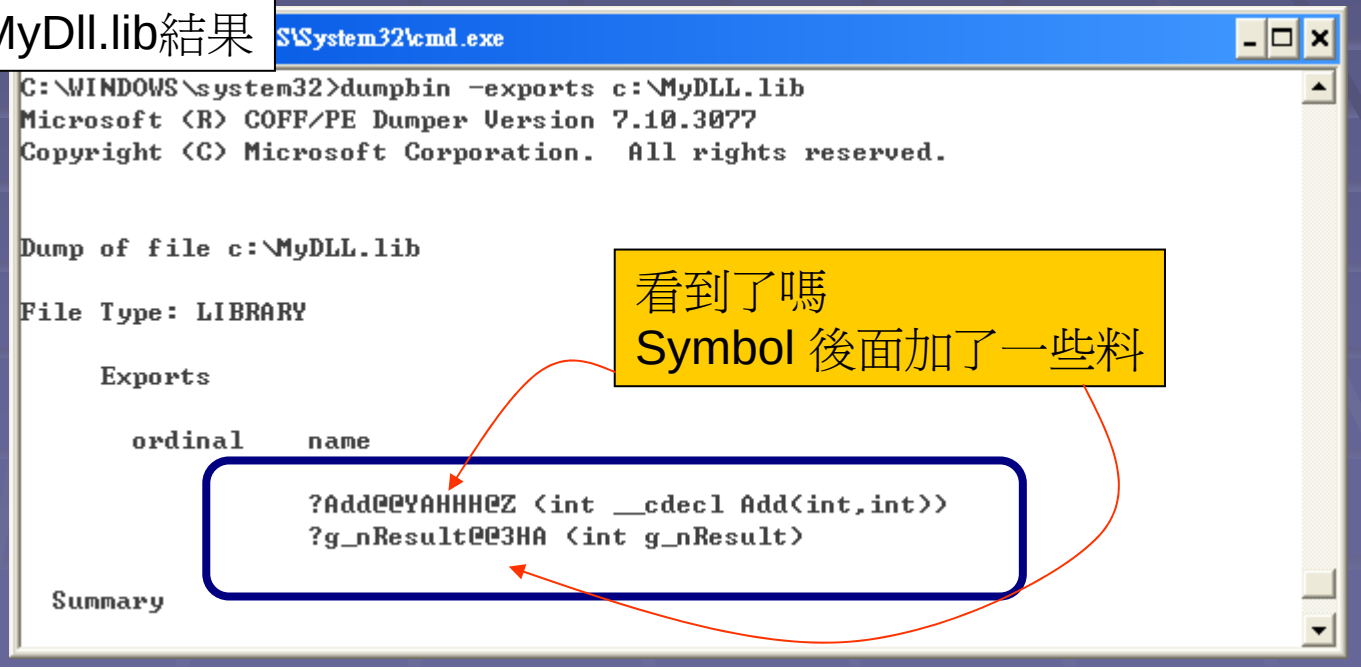
那請問一下:
加上 extern “C” 後, 你的 DLL 是否還支援 function overloading ?

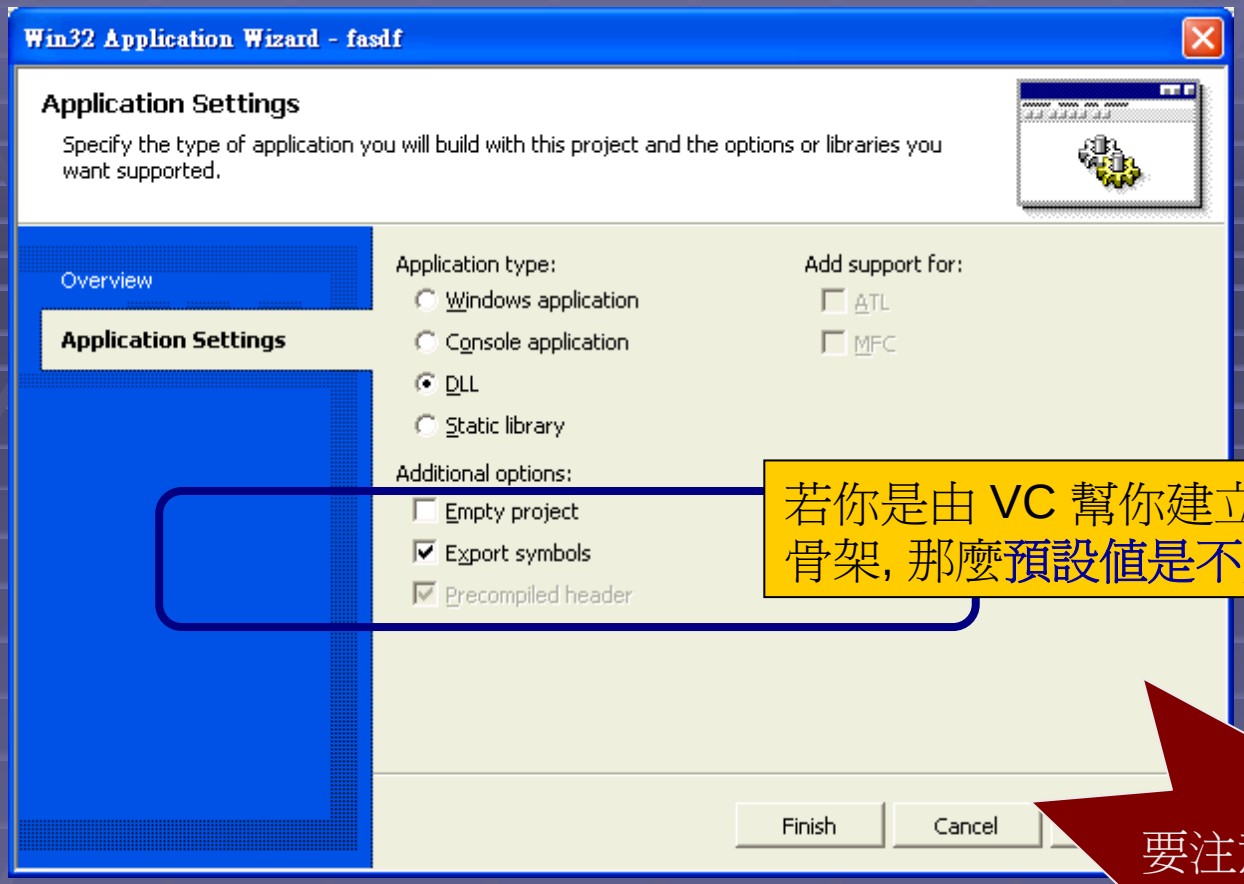
如果你的 MyLib.h 的內容如下

沒有使用 “C”

```
#ifdef MYLIBExport
   #define MYLIBAPI extern __declspec(dllexport)
#else
   #define  MYLIBAPI extern __declspec(dllimport)
#endif
MYLIBAPI int g_nResult;
MYLIBAPI int Add(int nLeft, int nRight);
```

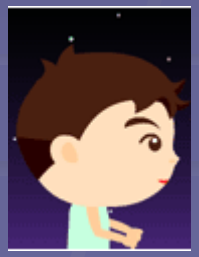
利用 dumpbin 察看 MyDll.lib結果





若你是由 VC 幫你建立 DLL export symbol 骨架, 那麼**預設值是不加上"C"**

要注意: 別弄錯了, 否則會發生 link 錯誤

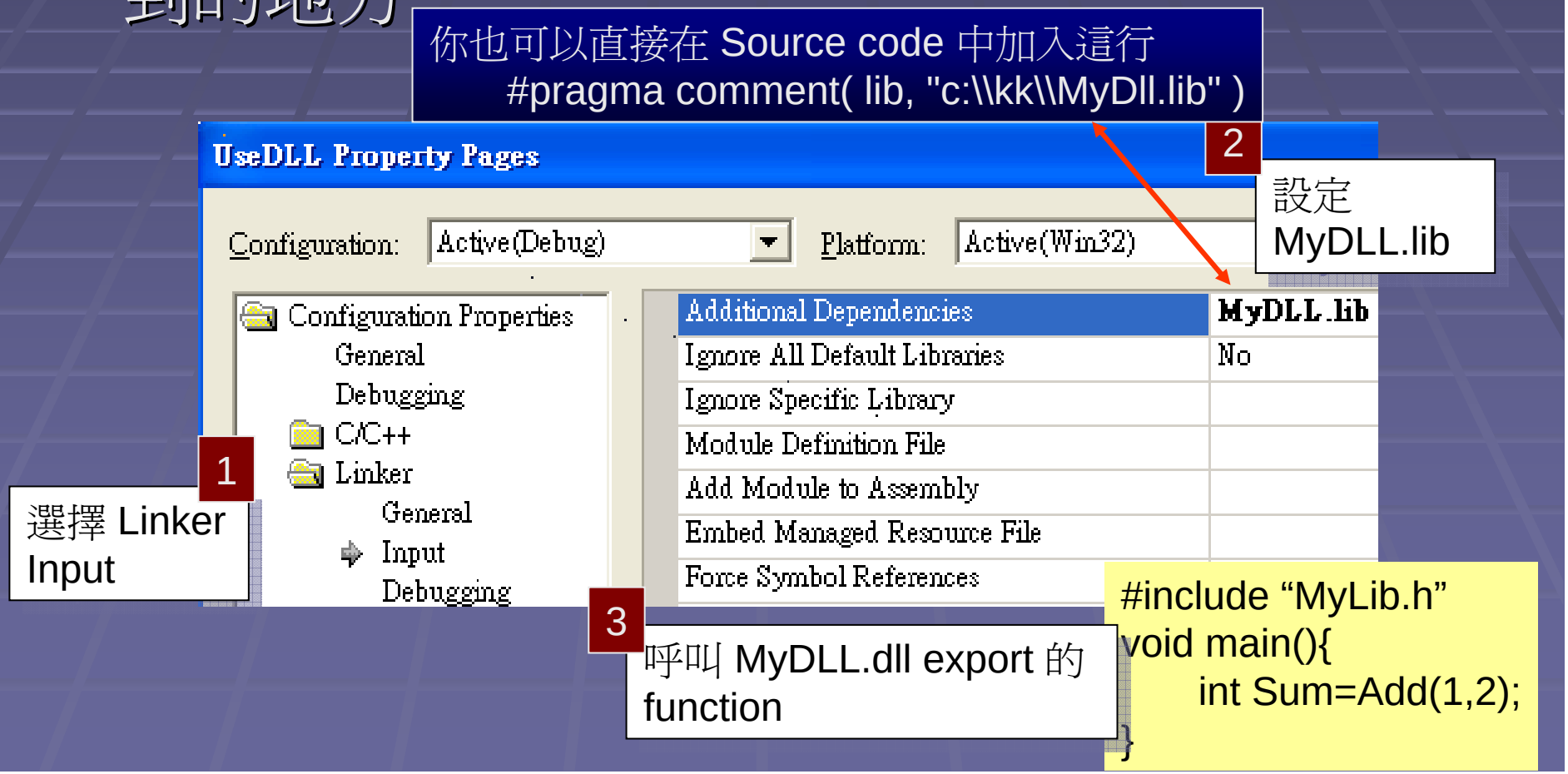


好了,接下來,就是直接使用 function

```
#include "MyLib.h"
void main(){
        int Sum=Add(1,2);
}
```

使用 MyDLL.dll

- Copy MyDLL.lib 與 MyDLL.dll到可以搜尋到的地方



__declspec(**dllexport**) 的意義

- **extern** "C" __declspec(**dllexport**) void myfunction() { … }

Microsoft VC 發現__declspec(**dllexport**)
會在 .obj 中加入額外的鏈結資訊

- 當 linker 產生 DLL檔時,會
  - 合併上述資訊產生 .lib 檔
  - 並且 embed 一個 symbol table 到 DLL檔中

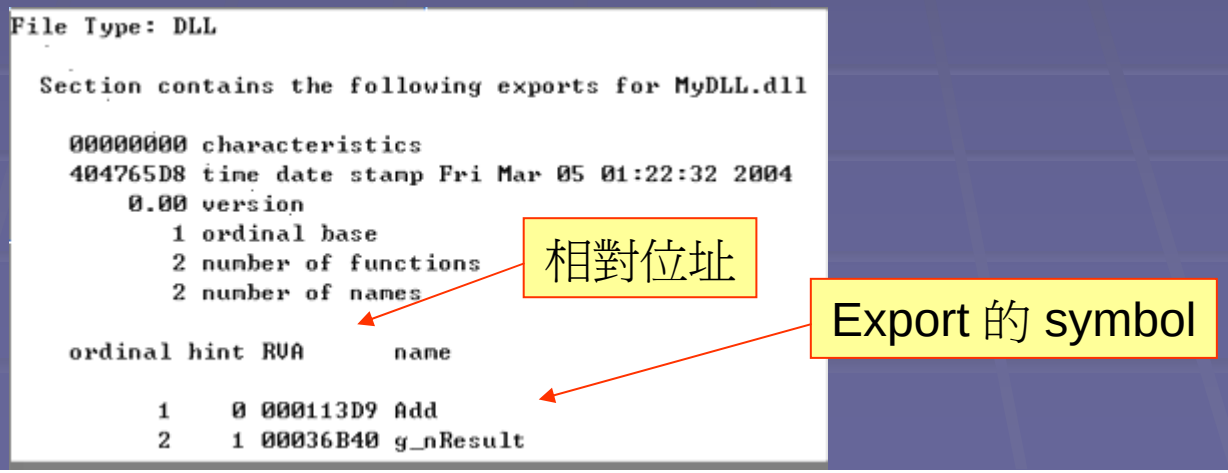
亦即 lib 檔中包含DLL 所**輸出的 symbol** 以及**相對的位址** (Relative Virtual Address)

查閱 DLL 檔所 export 的 symbol – 使用 DumpBin公用程式

- 使用前,先加入Path,使得相關的DLL檔可以找到

F:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE

- 執行 DumpBin.exe /exports MyDll.dll



讓其他的IDE 也能使用 Visual C++ 的DLL

- 問題是
  - 對於 **__stdcall** 修飾字的檔案,無論是 C/C++, Visual C++ 都會對 function name 加入修飾符號.

```
__declspec(dllexport) LONG __stdcall MyFunc(int a, int b);
```

_MyFunc@8

後面有 8 個 bytes

__stdcall:
1. 最右邊的參數先推入堆疊
2. 函式返回前,自行移除堆疊中的參數

*Win32 API 使用 __stdcall

因爲 C/C++ 不定參數的關係所以必須要在 function 後面加上需要 pop 的數字

讓其他的IDE 也能使用 Visual C++ 的DLL

- 解決方案: 要求 VC不要加額外的資訊到 DLL中
  - 方法一: 編輯 .def 檔

```
EXPORTS
MyFunc
```

  - 方法二: 在 DLL原始程式中加入

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

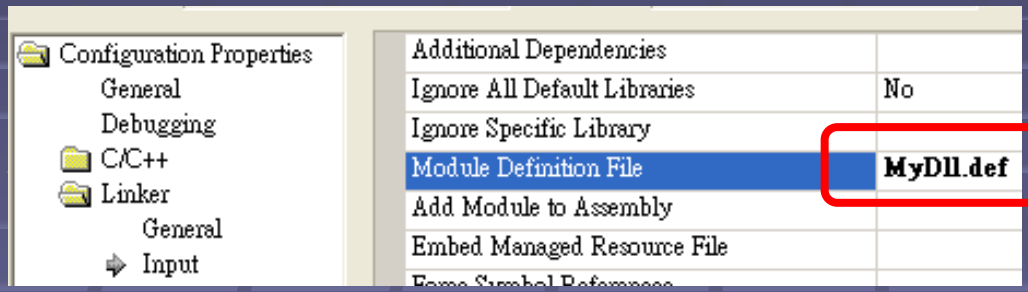
補充

```
ordinal hint RVA      name

      1    0 00011357 _Add@8
      2    1 00036B40 g_nResult
```

沒有加入 .DEF 檔

dumpbin /exports MyDLL.dll

加入 .DEF 檔



LIBRARY MYDll
EXPORTS
  Add

```
ordinal hint RVA      name

      1    0 00011357 Add
      2    1 00036B40 g_nResult
```

dumpbin /exports MyDLL.dll

Import 的實際意義

- 在使用 DLL import 的symbol 時你可以用 **extern** 代替 __declspec(dllimport)

extern "C" __declspec(**dllimport**) void myfunction() { … }

extern "C“ **extern** void myfunction() { … }

但是, 如果你想產生比較有效率的程式碼. 請使用 __declspec(dllimport)

Import 的實際意義

- Link 發現 __declspec(dllimport) 時，會在執行檔中的 import section 列出該檔所參考的 symbol
- 查閱執行檔所需的 symbol

DumpBin –imports UseDLL.exe

```
Dump of file UseDLL.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    MyDLL.dll
                45A39C Import Address Table
                45A1D4 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                    0 Add

    KERNEL32.dll
                45A204 Import Address Table
                45A03C Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                   EE FreeEnvironmentStringsW
                  306 SetEnvironmentVariableA
```

使用 MyDLL.dll 的部分

使用 Kernel32.dll 的部分

執行 Executable Module

Step1: Loader 會爲該 Process 建立 **virtual address**

Step2: 將 executable module **map** 到 virtual address

Step3: 檢查 import section 並且 map 需要的 DLL到 virtual space 中

搜尋 DLL 的順序

1. 目前執行檔的位置
2. 該 process 的目前目錄
3. Windows 的系統目錄
4. Windows 目錄
5. Path 環境變數

Step4: 檢查每個**DLL 的 import section** 並且 map 需要的 DLL

系統會紀錄每個 DLL 載入的次數
使得DLL 不會重複載入

當系統找不到 DLL 時,
所發出的錯誤訊息

Step5: 檢查所有模組的 **import symbol** 與
載入**DLL**的 **export** 是否一樣

DLL base addr + RVA

Step6: 計算每個 symbol 的實際位址
將其放到 executable module 的
import section

事實上,是 update import section
中的 Imported Address Table (ITA)

Reference: ms-help://MS.MSDNQTR.2004JAN.1033/dnmsj00/html/hood0200.htm

產生你的執行程式

- 當你的程式碼 reference 一個 import symbol
  - 可以由 import section 得到真實的位址, 因此可以成功的存取 DLL 的變數,function和類別
- 缺點
  - Loader 啓動你的程式前,需要
    - 重新計算每各 module 呼叫位址以及載入DLL 都需要時間

由於 DLL 的 base address可能會改變,
所以必須在 loading time 時,
才能決定 import function 真實所在位址

- End

附錄

Calling convention	VC++ name	VC++ (DEF used)	C++Builder Name
__stdcall	_MyFunction@4	MyFunction	MyFunction
__cdecl	MyFunction	MyFunction	_MyFunction

直接將 COFF 的 library format 轉成 OMF 格式

IMPLIB (destination lib name) (source dll)

例如:

IMPLIB mydll.lib mydll.dll