目　　錄

使用 EXCEL 呼叫 C++撰寫的動態函式庫

一般而言，目前業界處理商品定價、避險參數計算以及財務報表計算等財務數值問題時，通常使用 Excel 作爲計算工具。Excel 不但有友善的使用者介面，也支援各式圖表的繪製，並提供各式財務、數學以及統計函數，讓使用者可在試算表中直接呼叫。此外，Excel 也提供 VBA（Visual Basic for Applications），讓使用者可以撰寫簡易的 Visual Basic 程式。當執行簡單的運算時（例如使用選擇權的評價公式求選擇權價格），使用 Excel 是一個不錯的選擇。

但處理複雜而且大量的數值運算時（例如使用蒙地卡羅法求選擇權價格），Excel 的執行效率就會變差。針對這個問題，本章提供一套解決方案：先使用 C++撰寫動態連結函式庫（Dynamic Linked Library，簡稱 DLL），再使用 Excel 呼叫該 DLL 中的函式來完成計算工作。由於 C++程式計算功能強大且效率高，所以使用 Excel 呼叫 DLL 可以明顯地提升執行效率。我們使用蒙地卡羅法計算陽春買權爲例，比較使用 VBA 與 DLL 的執行效率[1]，如下表所示：

蒙地卡羅的模擬次數	10000	30000	50000	100000
VBA(ms)	62.5	183.5	308.5	605.5
DLL(ms)	15	30	62	109

本章將先介紹函式庫的基本概念，比較動態連結函式庫和靜態連結函式庫（Static Linked Library，簡稱 SLL）。接下來介紹如何使用 Visual Studio 來撰寫 DLL，並討論如何使用 Excel 呼叫 DLL 中的函式。此外，微軟公司也特別爲 Excel 開發一套 DLL，稱爲 XLL（Extensible Linking Language），XLL 中的函式可透過 Excel 增益集的設定，就可以在試算表中直接呼叫。本章也會介紹如何撰寫 XLL，並透過 Excel 增益集的設定，呼叫 XLL 中的函式。

函式庫的介紹

函式庫由提供特定功能的函式構成，可讓外界呼叫並進行應用。函式庫分爲兩類：靜態連結函式庫（SLL）和動態連結函式庫（DLL）。

[1] 使用的電腦其 CPU 爲 Intel Core Duo 1.83G ，記憶體爲 1G。

SLL 在應用程式編譯時連結到應用程式並成爲執行檔的一部份，使得執行檔較大。因此若連結較多 SLL時，執行檔的大小也隨之增加。根據這樣的特性，當許多不同的執行檔連結到同一個 SLL 時，便會造成記憶體的浪費；相反地，DLL 在需要用到時才會載到記憶體中，通常是在應用程式開始執行時，所以其執行檔不包含函式庫，執行檔較小，可節省記憶體使用空間，此特性也讓程式在更新修改上較爲容易。

第一節　DLL 的建構以及呼叫

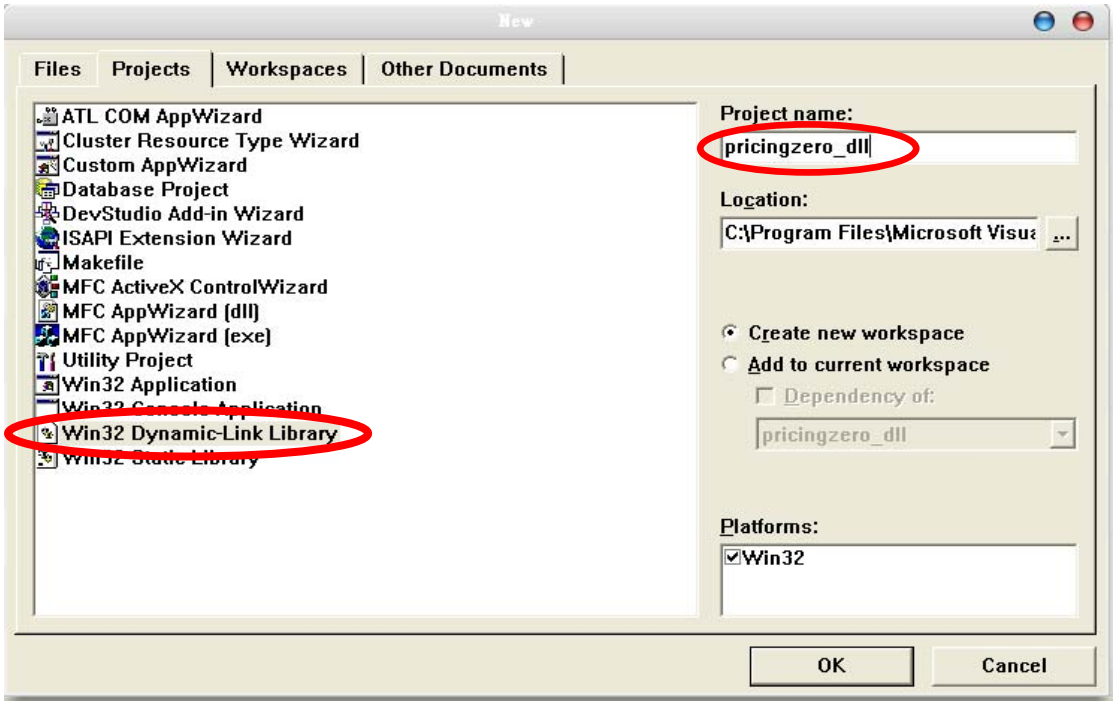
在 Excel 裡呼叫 DLL 中的函式以便進行後續計算共可分爲兩大步驟。

- 以 C++撰寫 DLL（前五節均以 Visual C++ 6.0 爲示範平台）
- 透過 Excel 裡的 VBA 來呼叫 DLL 進行實際應用。

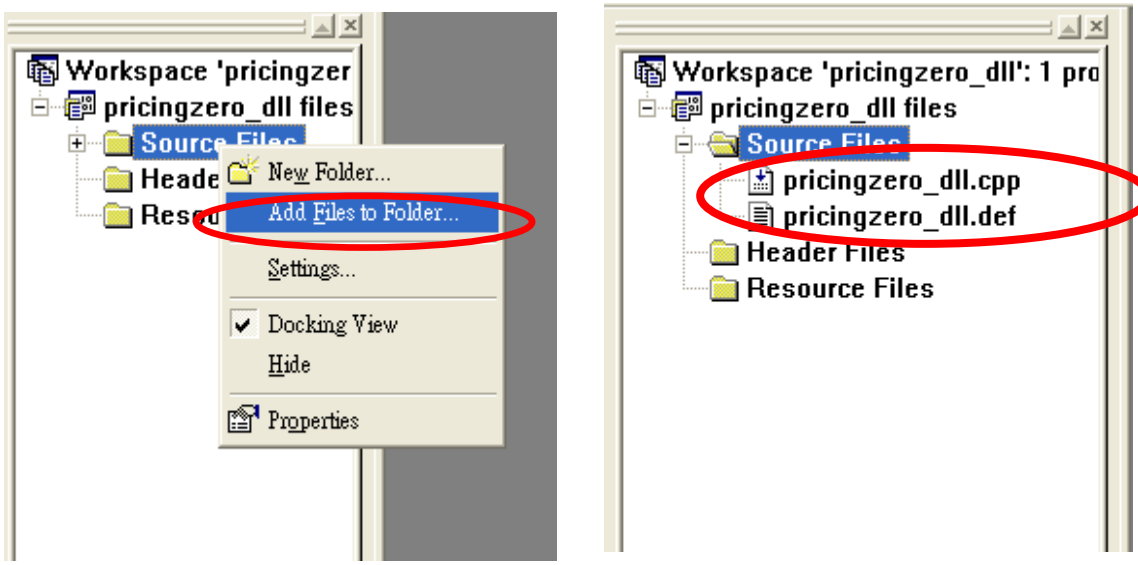
本節以評價零息債券爲例，示範以上兩大步驟。

（一）DLL 的撰寫

首先開啓 Visual C++ 6.0 按「File/New」，選擇「Win-32 Dynamic-Link Library」並在右方欄位決定專案名稱及其儲存位置。此處將專案命名爲：pricingzero_dll



專案建立後在「Source File」檔案夾上按右鍵後選取「Add Files to Folder」並增加兩個檔案，分別命名爲：pricingzero_dll.cpp、pricingzero_dll.def



上述兩個檔案在稍後完成編譯後將會合成爲一個完整的 DLL 檔案，其中在 pricingzero_dll.cpp 裡的主要內容爲運算程式的主架構，也就是關於評價零息債券的計算過程都將撰寫於此（以 C++撰寫，故副檔名爲 cpp）。之後 Excel 利用 VBA 呼叫 DLL 裡的函式時，會先在 DLL 檔案 pricingzero_dll.def 中尋找函式是否有被輸出，接著才透過此連結進入.cpp 的部分進行函式運算。

開始在這兩個檔案加上程式碼，首先 pricingzero_dll.cpp 部分的程式碼將由評價零息債券的運算組成。由前幾章的介紹可知，評價零息債券的公式爲：

$$price = \frac{FV}{(1+yield)^{period}} = FV \times (1+yield)^{-period}$$

由於計算式中包含了-*period* 次方，所以需要 pow()函式來撰寫程式，因此在程式的一開頭，必須先加入#include <math.h>以告訴編譯器，程式要呼叫存在於 math.h 函式庫的函式。

此外，公式可拆成前後兩部分：債券的面額 FV 以及折現因子 $(1+yield)^{-period}$，所以分別用 discount_factor()和 pricing_zero()兩個函式來計算折現因子以及債券價格。

計算折現因子的部分，建立一個將計算結果以 double 表示，名爲 discount_factor 的函式。而根據上述評價公式，這個函式裡所需要的變數爲殖利率（型態爲 double）以及總期數（型態爲 long）。

計算完折現因子後，便可直接乘上面額得到零息債券的價格。因此在 pricing_zero()函式中，建立一個將計算結果以 double 表示，名爲 pricing_zero 的函式。而這個函式所需要的變數即爲債券面額（double）。

因此依據以上兩部分說明，pricingzero_dll.cpp 檔案中的程式碼如下。

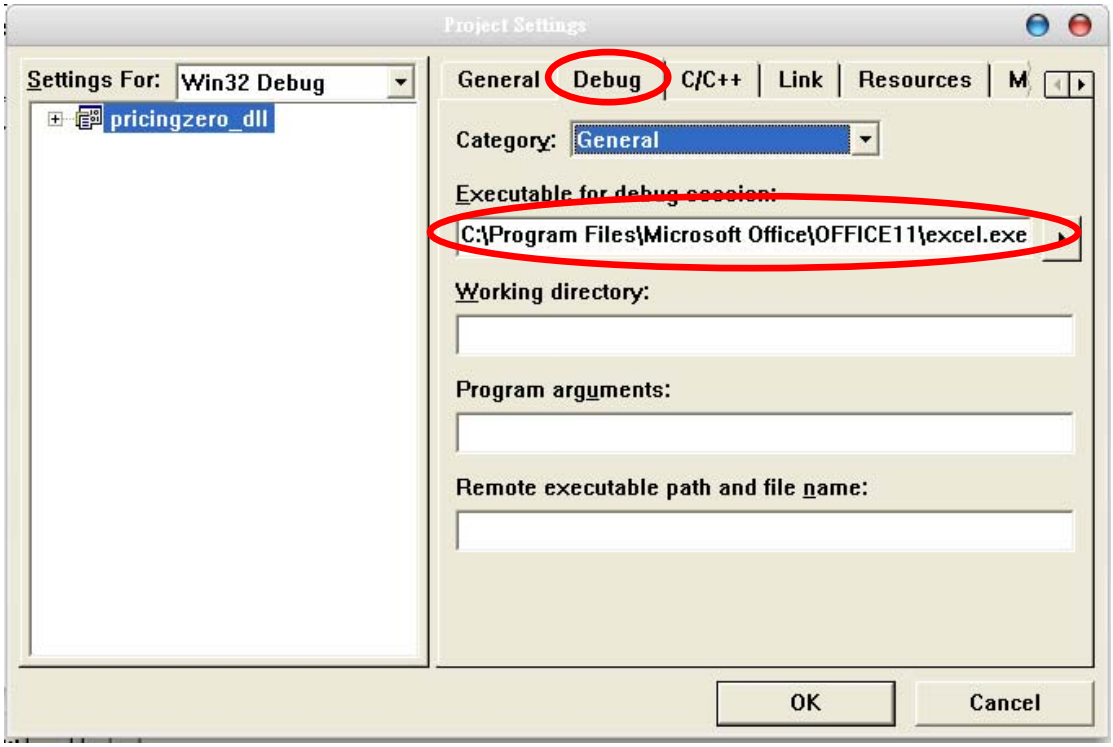
```
#include <math.h>

double discount_factor(double yield,long period)
{
    double discount=pow(1+yield,-period);
    return discount;
}

double _stdcall pricing_zero(double FV, double yield, long period)
{
    double price=FV*discount_factor(yield, period);
    return price;
}
```

接著則是 pricingzero_dll.def 部分的程式碼，其功用在於定義有哪些函式可提供其他程式呼叫。EXPORTS 的意義即爲建立函式對外部應用程式的連結，因此只需將函式名稱在其下列示即可，而本例中將會連結到的函式爲.cpp 檔中的 pricing_zero，因此這部分的程式碼爲：

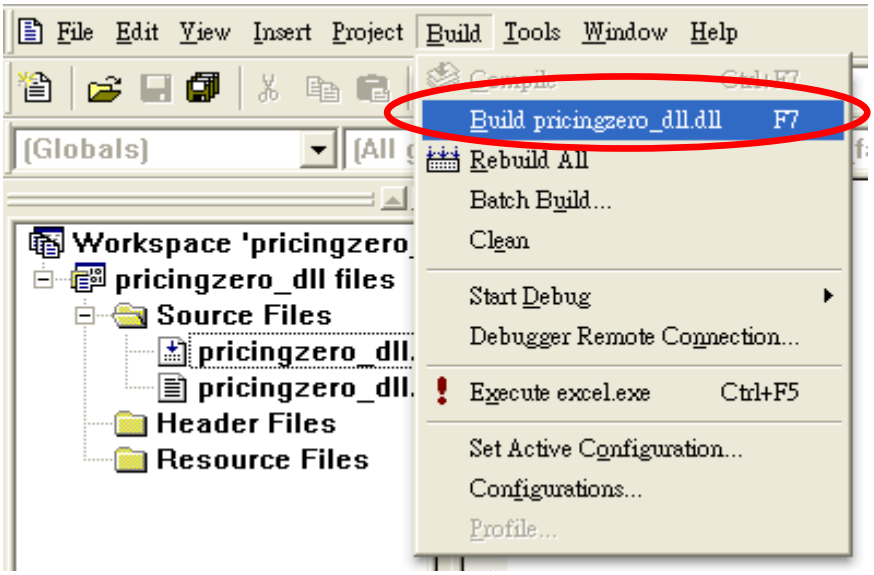
```
EXPORTS
    pricing_zero
```

（二）DLL 的除錯

至此程式碼完成之後，除了利用 Visual C++ 6.0 裡編譯的功能除錯外，一些編譯器無法找到的錯誤可設置程式碼斷點以及利用外部的應用程式進行除錯。爲了設定除錯環境，首先按「Project / Settings」，選擇 Debug 標籤後指定外部應用程式的路徑（例如指定 Excel 爲除錯的外部應用程式）



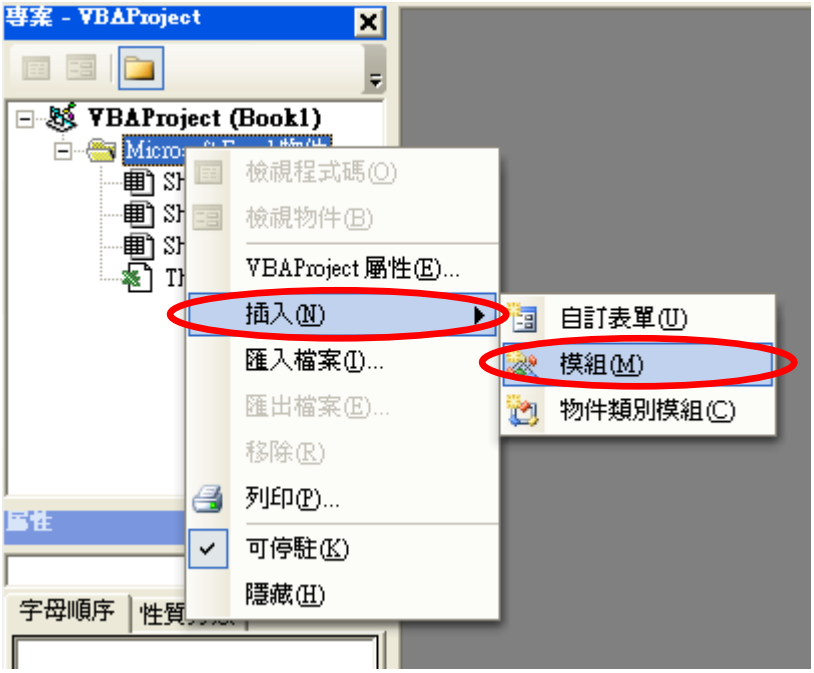
待程式碼均通過編譯無誤後，便可將撰寫完成的部分建構成 DLL 檔案以結束第一步驟。「Build / Build pricingzero_dll.dll」或直接按熱鍵 F7 均可。



（三）利用 Excel VBA 呼叫 DLL 函式

接著則進行第二步驟：透過 Excel 裡的 VBA 呼叫 DLL 以便計算。延續評價零息債券的示範，此處即爲透過 VBA 來呼叫 pricingzero_dll.dll 中的 pricing_zero 函式進行對零息債券的評價。

首先打開 Excel，按「工具 / 巨集 / Visual Basic 編輯器」或熱鍵 Alt+F11 進入 Visual Basic 的編輯環境。並在 VBA Project 中任意一處按右鍵插入模組。



接著則在這個新建立起的模組中撰寫程式碼以呼叫 pricingzero_dll.dll 檔案中的 pricing_zero 函式。由於函式爲外部程式，因此呼叫時必須指出存有此函式的 DLL 檔案所在位置，需注意路徑名稱不得含有中文。此外，因爲使用函式時牽涉到變數數值資料的傳送，所以必須將變數以 call by value 的方式呼叫。

此處值得注意的是程式碼若需要斷行，則在斷行處的最後加上底線符號。VB 程式的相關語法可參考市面上的眾多教學書籍，此處礙於篇幅，不多贅述。

由於評價零息債券的計算程式先前已完整建構在 DLL 檔案中，因此在本例中 VBA 呼叫 pricing_zero 函式後便可直接在 Excel 中使用，但若函式還需進一步地擴增其功能，便可繼續在此環境下以 VB 語言撰寫接下來所需的程式碼，端賴使用者的需要。

例如想要計算兩個同樣條件的零息債券價格總和，可撰寫一 VBA 函式 pricing_twozero()，此函式便是呼叫 DLL 函式（pricing_zero()）後進行的擴增應用。

```
Function pricing_twozero(FV As Double, yield As Double, period As Long) As Double
    pricing_twozero = 2 * pricing_zero(FV, yield, period)
End Function
```

完成呼叫函式以及其擴增後，便可回到 Excel 裡進行計算。此處以評價面額 100 元、殖利率 5%，還有五年到期的零息利率爲例，示範 Excel 可能的計算方法。

下表中 A 欄爲不同的計算方法，B 欄爲計算結果，C 欄則是使用在 B 欄中的公式。

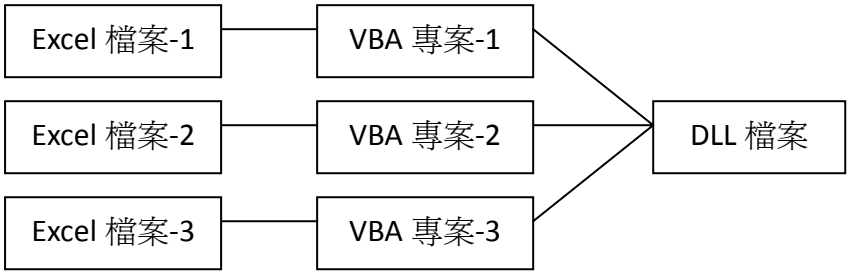
	A	B	C
1	FV	100	
2	yield	0.05	
3	period	5	
4			
5	Excel cell calculation	78.3526	B1/(1+B2)^(B3)
6	Excel PV function	-$78.3526	PV(0.05,5,0,100,0)
7	pricing_zero	78.3526	pricing_zero(100,0.05,5)
8	pricing_twozero	156.7052	pricing_twozero(100,0.05,5)

第五列展示的第一種方法爲 Excel 的格位計算，對照評價公式可直接以此方法簡單計算。而第六列則是 Excel 內建的 PV( )公式，輸入所需的變數資料後同樣可以求出正確的價格。（負數表示購買此債券現金向外的流向）

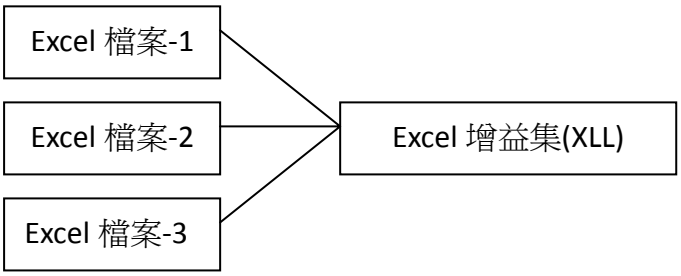
第七列以及第八列則是使用者自建的公式。第七列爲直接呼叫並使用 DLL 函式，而第八列計算兩張債券的價格即爲利用 VBA 對其所呼叫進來的函式之擴展應用。

第二節　XLL 的簡介

瞭解 DLL 的建構及其在 Excel 裡的應用方法後，使用者不難發現在進行財務計算時，縱使呼叫 DLL 函式的效率較高，但仍有其複雜之處。因爲 DLL 函式在應用時必須透過 VBA 的呼叫才能使用，所以當不同 Excel 檔案呼叫同一個 DLL 函式時，使用者就必須在每個 Excel 檔案下的 VBA 撰寫呼叫程式碼。如下圖示：



爲了改善這樣的不方便，前文曾提及微軟公司特別爲 Excel 開發出一套專屬的 DLL，稱爲 XLL（Extensible Linking Language）。XLL 較 DLL 方便之處就在於建構完畢後可透過相關設定，直接在 Excel 的增益集中勾選使用，而不必撰寫另一層呼叫的程式碼。因此便可避免使用 DLL 時必須不斷撰寫 VBA 程式碼的麻煩。

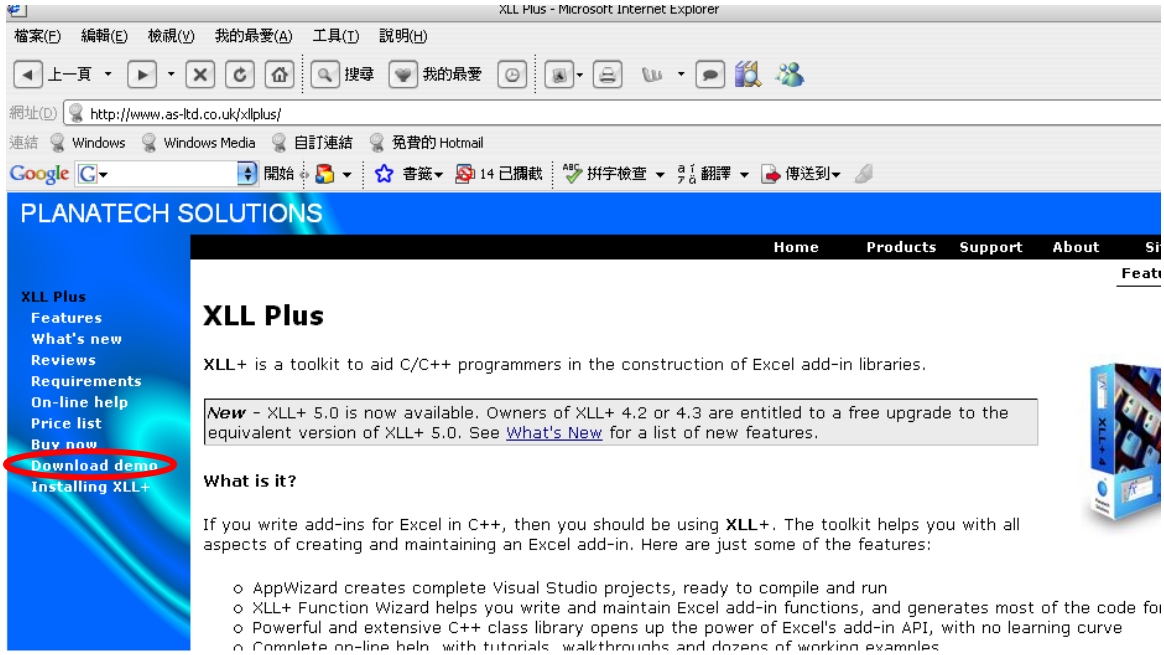


XLL 的建構方法則分爲兩類：利用 XLL Plus Toolkit 以及 Microsoft Excel 97 Deverloper's Kit。以下兩節分別舉例介紹。

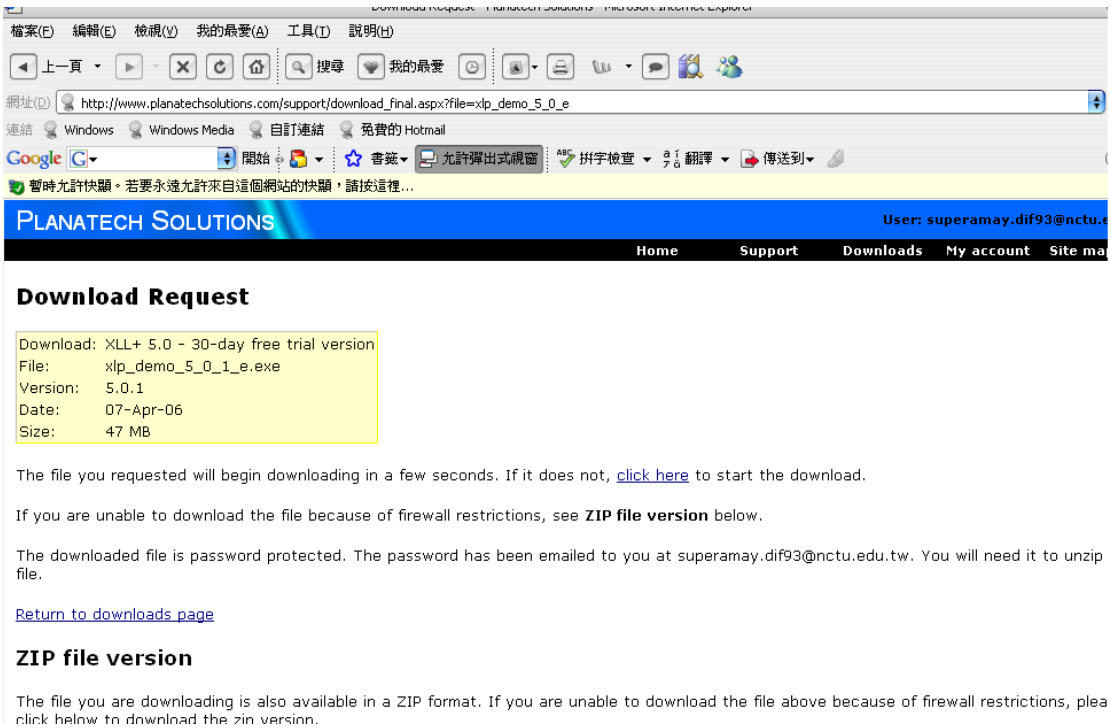
第三節　XLL 建構方法 I －
XLL Plus Toolkit

XLL Plus Toolkit 是目前市面上既有的套裝軟體，只需根據其步驟便可簡單完成 XLL 檔案之建構。首先免費下載 PLANATECH SOLUTIONS 提供的 XLL Plus Toolkit 的試用版本來完成任務。

首先進入產品網頁 http://www.as-ltd.co.uk/xllPlus/後直接進行下載。



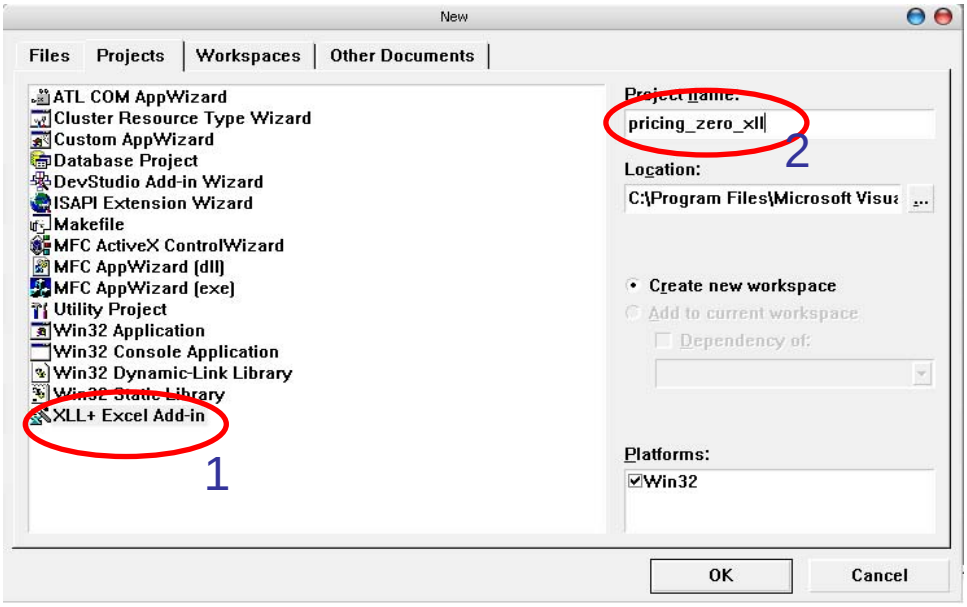
之後只要註冊會員並登入之後，就可直接下載 XLL Plus Toolkit。



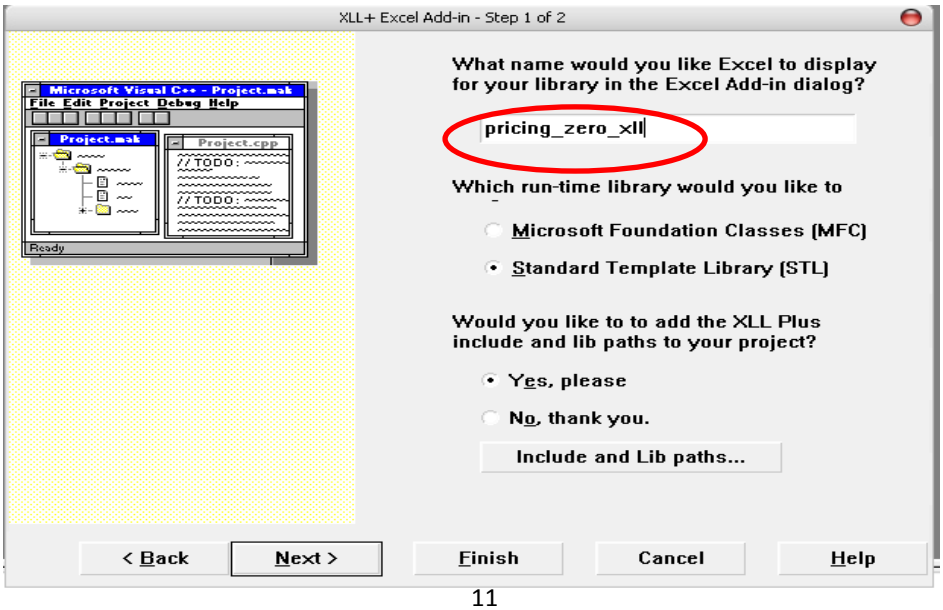
下載完成後進行解密安裝，安裝完之後 Visual C++ 6.0 的 Project 選項中，會多一項 XLL+ Excel Add-in 的選項，接著就可使用此方便的套裝軟體來建構 XLL。但值得注意的是，這裡所安裝的免費版本僅提供 30 天的試用期。試用到期後，使用者想繼續使用就只能掏錢購買這項產品了。

以下以評價零息債券爲例，示範從 Visual C++ 6.0 的工作環境下使用方便的 XLL Plus Toolkit。（以 C++程式評價含息或零息債券的方法請參見本書 3-23）

首先。開啓 Visual C++ 6.0 後，按「File/New」開始建構 XLL。開啓「New」的對話視窗後，首先選取 XLL+ Excel Add-in，並在「Project name」的欄位填入檔案名稱。此處使用 pricing_zero_xll 來當作評價零息債券的名稱。完成之後按「OK」繼續往下。

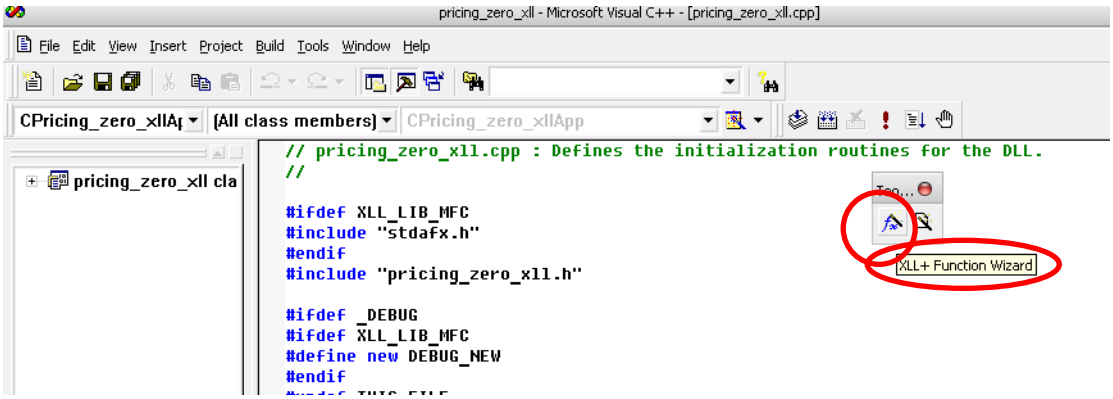


輸入名稱之後，按「Next」進入下一個步驟。

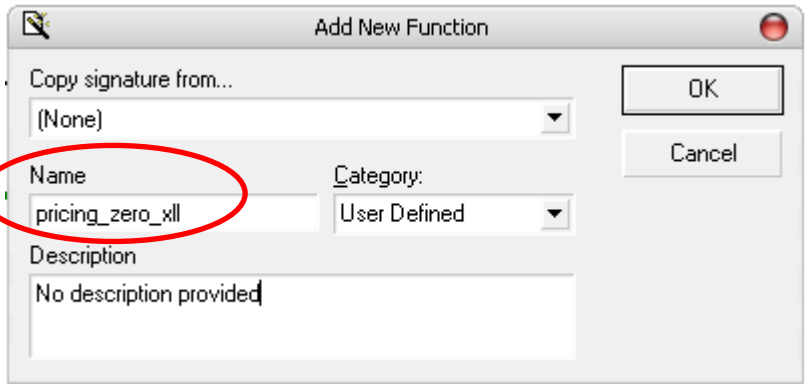


之後跳出「New Project Information」提供使用者相關資訊，按確定之後可以發現大部分的程式碼 XLL Plus Toolkit 都已經撰寫完成了。

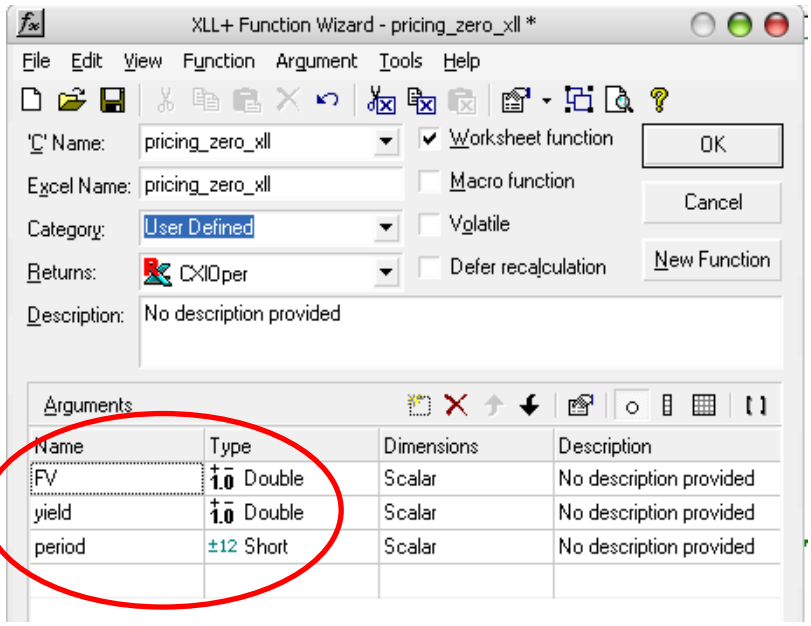
但事實上這裡寫好的僅是基礎環境的設定，真正的程式主架構（在本例中即爲評價零息債券）則需要進一步地補足：例如變數的宣告以及主架構的程式碼。而 XLL Plus Toolkit 的另一個好處就是能幫助使用者快速的宣告變數，使用者只需要利用 XLL+ function wizard，跟著其步驟就可以設計好所需的變數。



開啓了 XLL+ Function Wizard 之後，首先更改之後將會用到的函數名稱，這裡仍舊沿用 pricing_zero_xll 來當函數名稱。爲了使用上方便，通常會使用該函式的功能來爲函式命名，以免以後函數使用到 Excel 上時會搞不清楚這個函數的功能是什麼。



名稱更改完畢之後，繼續按 OK 進入下一步設定變數的部分。這裡爲評價零息債券，所以使用者輸入的變數將包含債券的面額、殖利率以及總期數。使用者便可直接在變數宣告的欄位裡輸入所需的變數名稱和資料型態。這裡以 FV 代表面額（double）、yield 代表殖利率（double）、period 則表示總期數（short）。



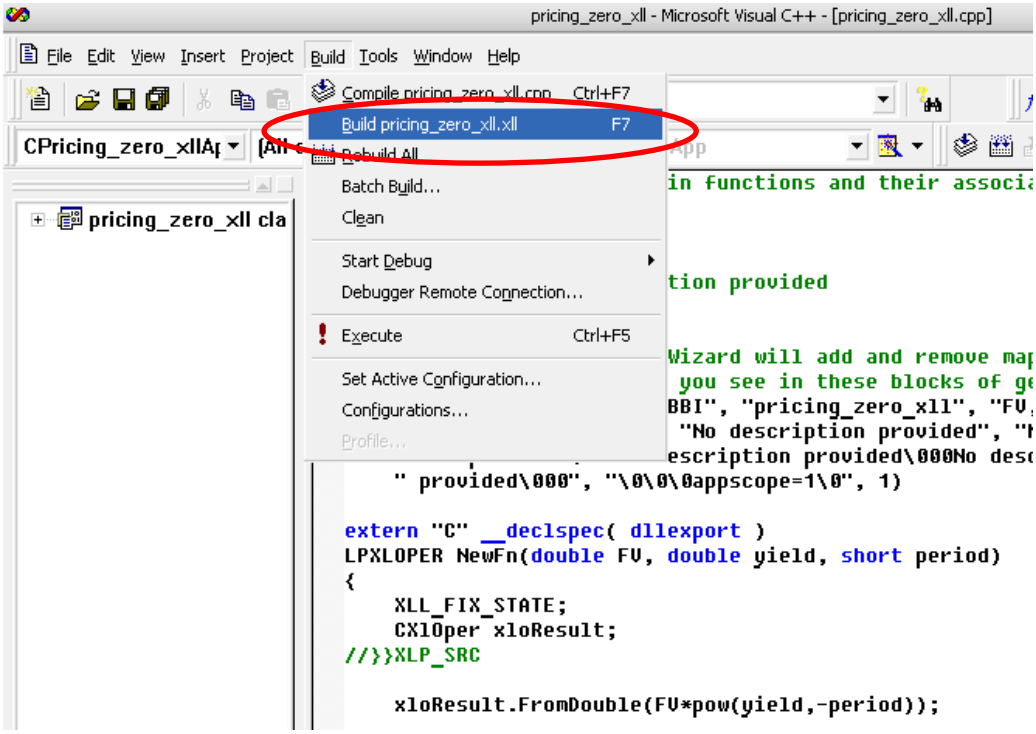
變數宣告完成後，使用者會發現原先的程式碼增加了這些剛設定完的變數，而新程式碼即爲 XLL Plus Toolkit 透過對話視窗將其直接轉換爲程式碼的結果。

接著爲主架構的程式碼，這部分則不再贅述，讀者可參考第一節內容（使用者只需根據需求補足框內的程式碼即可，其餘均爲系統產生）。其中值得注意的是，由於算出來的價格將會以 double 的型態存在，所以在這裡利用以下程式碼來儲存計算出來的價格。

```
xloResult.FromDouble();
```

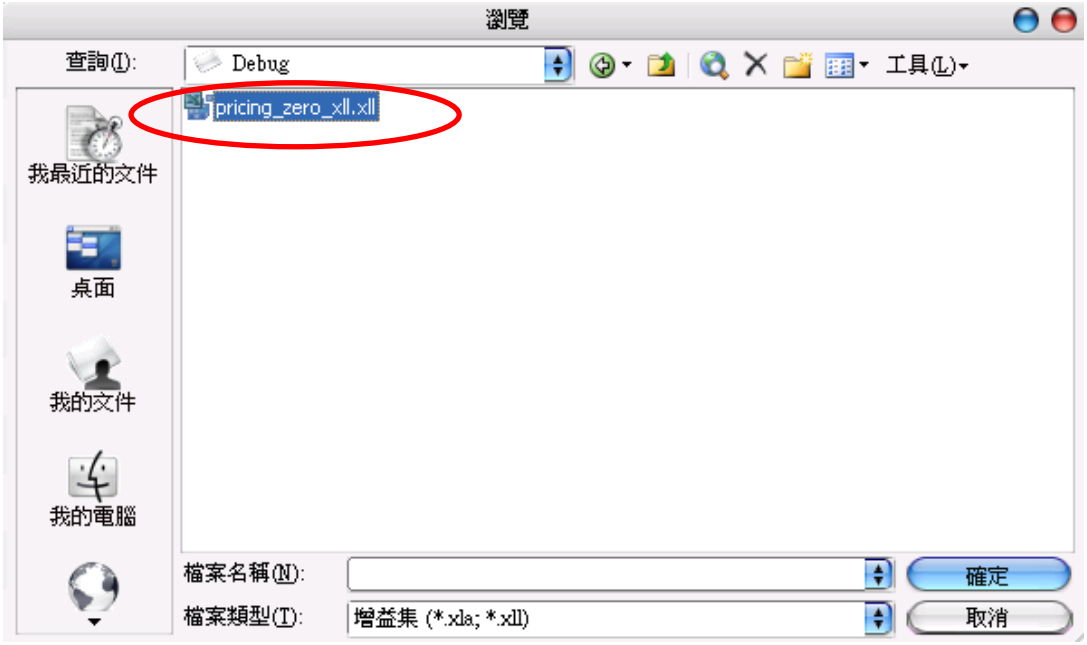
```
extern "C" __declspec( dllexport )
LPXLOPER pricing_zero_xll(double FV, double yield, short period)
{
    XLL_FIX_STATE;
    CXlOper xloResult;
//}}XLP_SRC

    // TODO - Set the value of xloResult
    xloResult.FromDouble(FV*pow(1+yield,-period));

    return xloResult.Ret();
}
```

完成所有程式碼後，便可將程式建構成 XLL 檔案。按「Build/ Build pricing_zero_xll」或直接按熱鍵 F7 後程式就會開始編譯。若程式無誤通過編譯後，XLL 檔案就完成了。

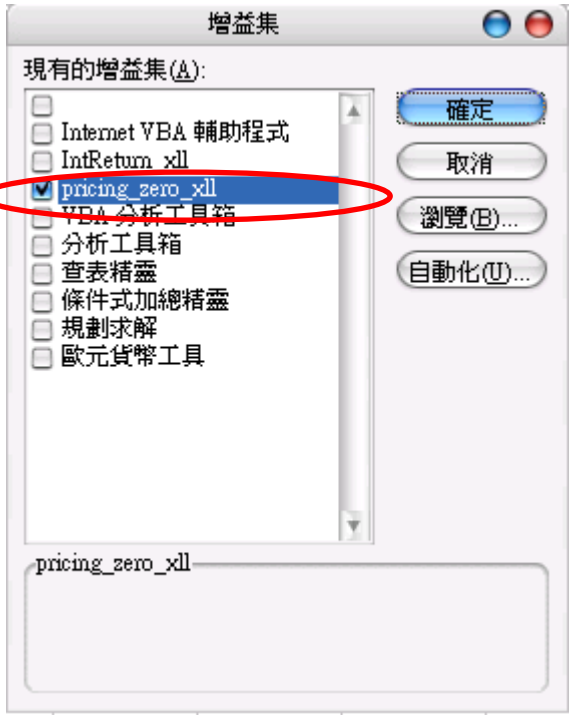


至此 XLL 算是建構完畢，接著可直接在 Excel 裡進行應用。但此 XLL 檔案尚未匯入至 Excel 以供使用，所以必須透過 Excel 增益集（「工具 / 增益集」）將公式 pricing_zero_xll 匯入 Excel 裡。

開啓增益集之後，按下瀏覽去找尋新建立好的 XLL 檔案。（預設位置在 VC6，my project 的檔案夾中。）



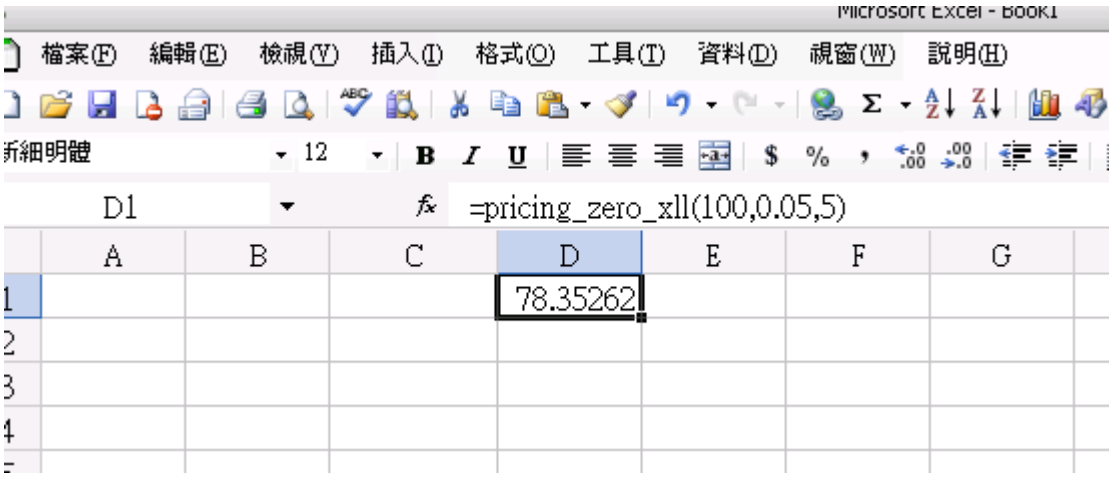
找到之後，公式便可匯入 Excel 中直接使用了。



接著回到 Excel 試算表，公式的使用方式就如同使用者所熟悉。例如想評價一個面額 1000 元，殖利率爲 5%的五年零息債券，那麼就可以直接選取一個格位，並依照公式要求的變數輸入格式進行這樣的計算：

```
=princing_zero_xll(1000,0.05,5)
```

而 Excel 在收到這樣的計算公式後，也會算出正確的答案。

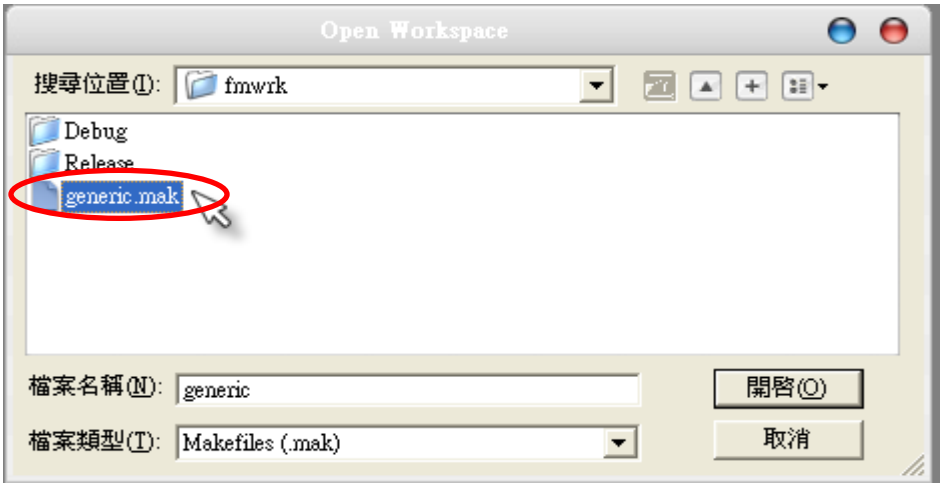


第四節 XLL 建構方法Ⅱ－ Microsoft Excel 97 Developer's Kit

雖然市面上有方便的套裝軟體可供使用，但其經過 30 天的試用期之後便需付費的特性卻也令人望之卻步，因此這裡介紹另一種建構 XLL 的方法，利用 Microsoft Excel 97 Developer's Kit（又稱 Excel 97 SDK，爲 Microsoft 爲 Excel 所設計之軟體開發套件），雖然其建構方式較爲複雜不若 XLL Plus Toolkit 方便快速，但卻有完全免費且不受任何試用期限制的優點。此外，軟體開發套件具有類似開放原始程式碼的基礎環境特性，使得使用者在充分瞭解各個函式後便可依個人需要發展不同的功能，也因此在使用上擁有相當大的彈性。

首先至 Microsoft 下載中心下載 Frmwrk32.exe（Excel 97 SDK 的樣版）http://download.microsoft.com/download/platformsdk/sample27/1/nt4/en-us/frmwrk32.exe，下載完成後解壓縮便可依循其安裝步驟完成 XLL 檔案。接著同樣以評價零息債券爲例，示範如何利用 Microsoft Excel 97 Developer's Kit 建構 XLL 檔案並在 Excel 裡進行財務計算。

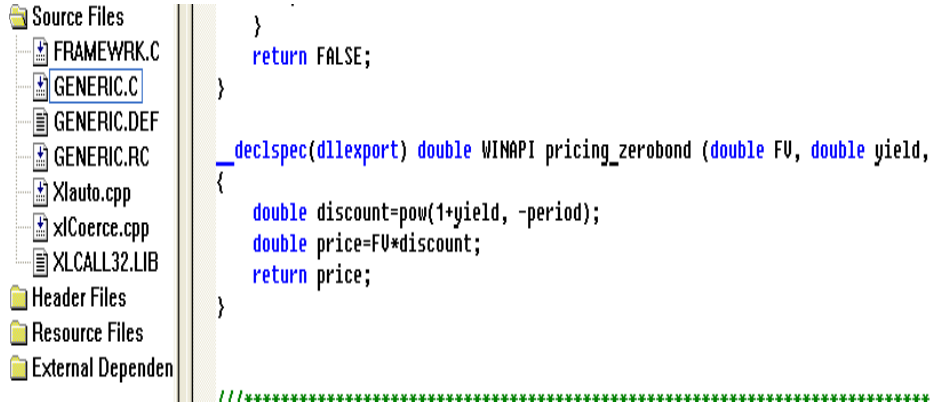
第一步開啓 Visual C++ 6.0 後，按「File/Open Workspace」並瀏覽至剛下載好的樣版開啓 generic.mak。



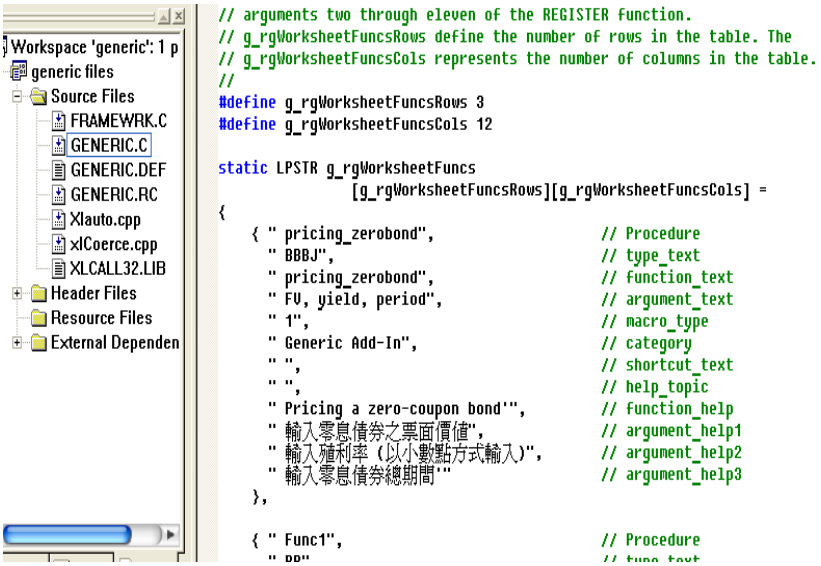
同樣地，可以看到專案裡有許多已完成的函式。例如 xlAutoOpen 即在 Excel 開啓或是增益集讀取時被呼叫使用，而 xlAutoClose 則在關閉時被呼叫使用（其他函式見下表整理）。但若使用者有特殊需要，可以直接在函式裡添加程式碼。舉例而言，若使用者在使用 XLL 時希望匯入外部資料，例如某期間的股價資料等，便可在 xlAutoOpen 內撰寫相關程式碼進行呼叫，而使用完畢關閉外部資料的程式碼則是加在 xlAutoClose 裡。

函式名稱	功能
xlAutoOpen	當 Excel 開啓或增益集載入時呼叫此函式
xlAutoClose	當 Excel 關閉或增益集卸載時呼叫此函式
xlAutoAdd	當 XLL 新增至增益集時呼叫此函式
xlAutoRemove	當 XLL 自增益集移除時呼叫此函式
xlAddInManagerInfo	當增益集管理程式第一次開啓時呼叫此函式
xlAutoRegister	當巨集註冊一個沒有設定引數及回傳值型態的函式時會呼叫此函式

瞭解這些函式之後，接著示範建構 XLL 檔案的相關步驟。首先在專案裡的 generic.c 檔案裡撰寫評價零息債券的主要程式碼－pricing_zerobond 的函式（內容與之前示範都相同，此處不再說明）。其中值得注意的是函式開頭處的關鍵字：**_declspec(dllexport)**，其意義爲匯出 DLL 的資料、函式、類別或類別成員函式。



接著則是針對 Excel 裡出現的公式輸入視窗做相關設定（設定完成後的結果如同圖 x）。首先在 generic.c 裡找到 g_rgWorksheetFuncs 的函式，並將函式上方的定義裡將列數改爲 3（預設爲 2），此處做更改是因爲除了系統提供的兩個範例函式外（Func1 與 FuncSum），必須加入自訂的 pricing_zerobond 函式。因此若之後在建構 XLL 檔案時需加入多個新函式時，則視其個數在此處做更動。



接著則仿照原有程式碼加入使用者所需的宣告以及設定，以下分別介紹：

1. Procedure：函式名稱。如 pricing_zerobond。
2. type_text：變數的資料型態代碼。在本例中爲回傳值（債券價格，double）、面額（double）、殖利率（double）、總期間（int），又 B 代表 double 而 J 代表 long int，因此此處輸入 BBBJ。（其餘代碼見表）

```
_declspec(dllexport) double WINAPI pricing_zerobond (double FV, double yield, long period)
{
```

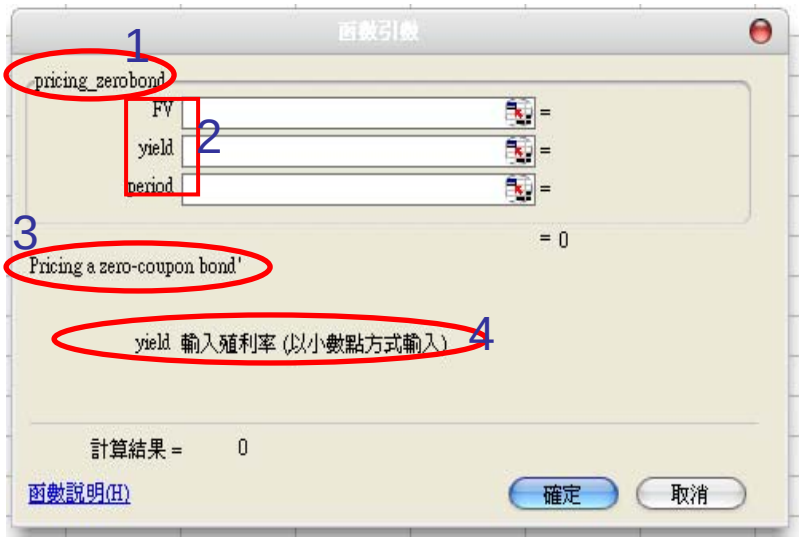
Code	Description	Pass by	C Declaration
A	Logical (FALSE = 0), TRUE = 1)	Value	short int
B	IEEE 8-byte floating-point number	Value (Windows) Reference (Macintosh)	double (Windows) double * (Macintosh)
C	Null-terminated string (maximum string length = 255)	Reference	char *
D	Byte-counted string (first byte contains length of string, maximum string length = 255 characters)	Reference	Unsigned char *
E	IEEE 8-byte floating-point number	Reference	double *
F	Null-terminated string (maximum string length = 255 characters)	Reference (modify in place)	char *
G	Byte-counted string (first byte contains length of string, maximum string length = 255 characters)	Reference (modify in place)	unsigned char *
H	Unsigned 2-byte integer	Value	unsigned short int
I	Signed 2-byte integer	Value	short int
J	Signed 4-byte integer	Value	long int
K	Array	Reference	FP *
L	Logical (FALSE = 0, TRUE = 1)	Reference	short int *
M	Signed 2-byte integer	Reference	short int *
N	Signed 4-byte integer	Reference	long int *
O	Array	Reference	Three arguments are passed: unsigned short int * unsigned short int * double [ ]
P	Microsoft Excel OPER data structure	Reference	OPER *
R	Microsoft Excel XLOPER data structure	Reference	XLOPER *

3. function_text：出現在 Excel 裡的公式名稱。如圖 x-1
4. argument_text：出現在 Excel 公式視窗裡的變數輸入名稱。如圖 x-2
5. macro_type：巨集型態，1 表公式、2 表命令。此處爲評價零息債券之公式因此輸入 1。
6. category：公式的分類類別。使用時可依公式類別歸入財務或統計等。此處則採用原始分類 Generic Add-In 表經由 XLL 建構出的自定公式。
7. shortcut_text：命令的熱鍵設定。此處爲公式而非命令所以沒有任何設定。
8. help_topic：說明文件的檔案，使用者若有任何針對公式及命令的說明檔案便可在此處附加。
9. function_help：顯示在 Excel 公式視窗裡的公式說明。如圖 x-3
10. argument_help：顯現在 Excel 公式視窗裡的變數輸入說明。如圖 x-4，由於此處變數輸入格共有三個，因此在此例中較預設的 argument_help 外多加了兩個說明設定。（argument_help2 及 argument_help3）

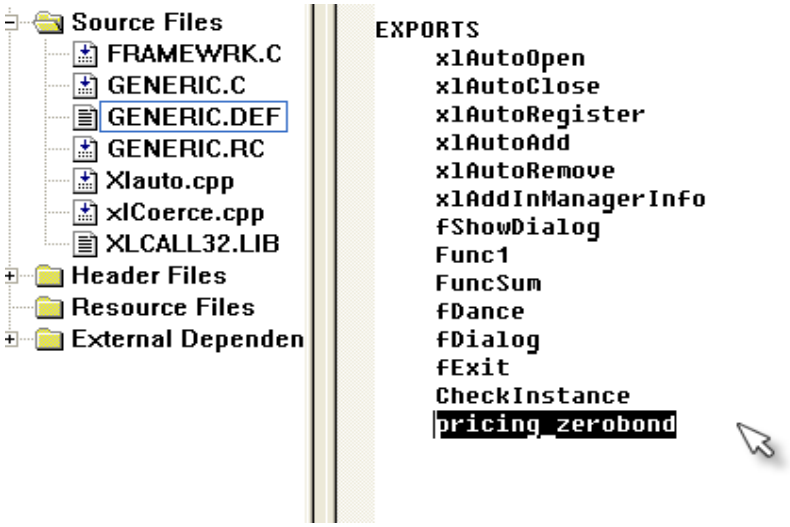
由於多加了兩項說明，因此在上方定義 g_rgWorksheetFuncsCols 也必須將欄位修改至 12。除此之外，使用者也必須針對這裡欄位的增加而對原有樣版的程式碼進行修改：搜尋 g_rgWorksheetFuncsCols 並修改其在迴圈裡的程式碼。

```
for (i=0; i<g_rgWorksheetFuncsRows; i++) {
      Excel(xlfRegister, 0, 1+ g_rgWorksheetFuncsCols,
          (LPXLOPER) &xDLL,
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][0]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][1]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][2]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][3]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][4]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][5]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][6]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][7]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][8]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][9]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][10]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][11]));
  }
for (i=0; i<g_rgWorksheetFuncsRows; i++) {
  if (!lpstricmp(g_rgWorksheetFuncs[i][0], pxName->val.str))  {
    Excel(xlfRegister, 0, 1+ g_rgWorksheetFuncsCols,
          (LPXLOPER) &xDLL,
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][0]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][1]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][2]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][3]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][4]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][5]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][6]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][7]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][8]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][9]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][10]),
          (LPXLOPER) TempStr(g_rgWorksheetFuncs[i][11]));
    /// Free oper returned by xl ///
```

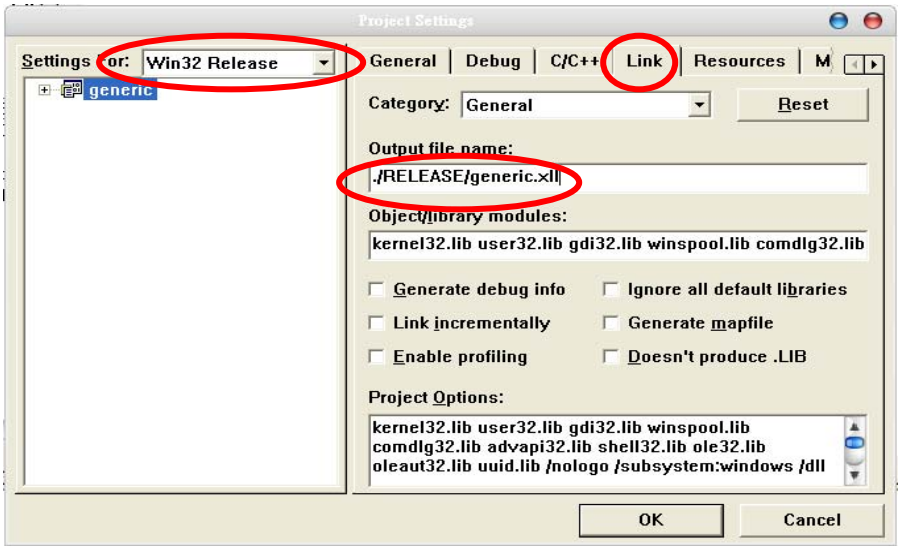
而所有設定完成之後，在 Excel 執行時顯示結果將有如下圖 X：



在 generic.c 檔案裡完成程式碼之後，接著的步驟則類似建構 DLL 檔案，必須到 generic.def 檔案裡將新函式匯出 Excel 使用。



至此程式碼已完成，接著匯出成 XLL 檔案。按 「Project/Settings」或熱鍵 Alt+F7 指定 XLL 檔案匯出的位置。



在匯出 XLL 檔案之前，先至「Build/ Set Active Configuration」將匯出方式改爲 Win32 Release，這裡做此更動是因爲以 Release Mode 匯出之 XLL 將較以 Debug Mode 匯出之檔案爲小。但若使用者欲利用 Debug Mode 來爲程式除錯，其使用方法就類似於先前在第一節所介紹的步驟。

最後一個步驟則是在 Excel 增益集裡將建構完成的 XLL 檔案匯入使用，步驟與前一節介紹的相同，而且此時也可以看到之前所做的設定都出現在 Excel 的公式視窗裡了。

第五節 XLOPER 資料結構

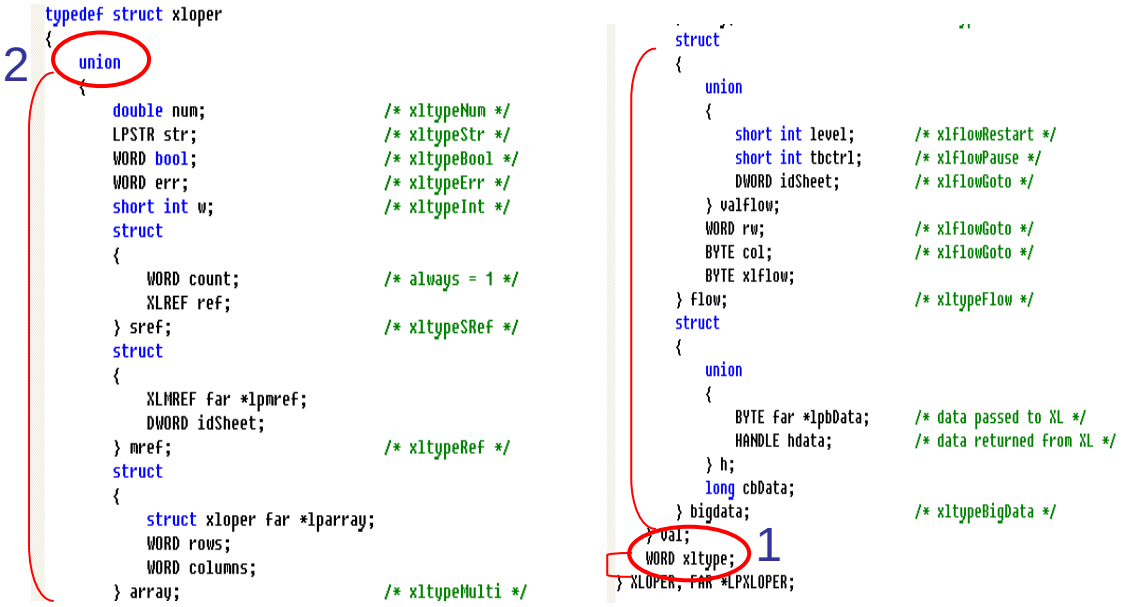
前一節評價零息債券的例子是建構 XLL 中較爲簡單的範例，因爲在使用公式時無論輸入或輸出的變數都僅是單一的數字，所以資料可在 Excel 與 XLL 中直接傳遞。但在許多財務計算問題中，輸入或輸出變數可能爲其他的型態而不僅是單一的數字，所以在撰寫程式時將會面對例如陣列或 Excel 格位等的資料型態，但 Excel 或 C/C++往往無法直接支援這些資料型態，因此本節的重點即爲介紹如何處理在 Excel 與 DLL 之間較繁複的資料存取方式。

基本上有四種方式可以處理 Excel 與 DLL 之間的資料傳送，以下分別介紹其特性及優缺點：

1. 透過 Excel 直接轉換爲 C/C++的資料型態：Excel 能處理各類數字的資料型態（例如 int,double,…），但無法支援 bool 與 char 等資料型態。若 Excel 遇到此類無法支援的資料型態時，在使用上不會呼叫函式而是直接回傳 #VALUE!的錯誤字樣。
2. 透過 xl_array 的資料結構：可進一步處理 8-byte double 的二維陣列。但陣列中僅能存放數字，且使用完後也較難釋放之前動態配置的記憶體。
3. 透過 xloper 的資料結構：能處理較多種的資料型態，包括陣列以及 Excel 的格位等等，使用上功能較爲強大。
4. 透過 oper 的資料結構：簡化型的 xloper 資料結構。

因此根據以上說明，此節將主要介紹能涵蓋最多種資料型態的 xloper 資料結構。而 xloper 資料結構分爲兩部分：（如同下圖）

1. 2-byte WORD（xltype）：儲存變數的資料型態。
2. 8-byte C union：xloper 所支援的資料型態，包含數字（xltypeNum）、字串（xltypeStr）、布林（xltypeBool）、陣列（xltypeMulti）、單一儲存格（xltypeSRef）、多重儲存格（xltypeRef）等。




```
typedef struct xloper
{
    union
    {
        double num;                 /* xltypeNum */
        LPSTR str;                  /* xltypeStr */
        WORD bool;                  /* xltypeBool */
        WORD err;                   /* xltypeErr */
        short int w;                /* xltypeInt */
        struct
        {
            WORD count;             /* always = 1 */
            XLREF ref;
        } sref;                     /* xltypeSRef */
        struct
        {
            XLMREF far *lpmref;
            DWORD idSheet;
        } mref;                     /* xltypeRef */
        struct
        {
            struct xloper far *lparray;
            WORD rows;
            WORD columns;
        } array;                    /* xltypeMulti */
        struct
        {
            union
            {
                short int level;    /* xlflowRestart */
                short int tbctrl;   /* xlflowPause */
                DWORD idSheet;      /* xlflowGoto */
            } valflow;
            WORD rw;                /* xlflowGoto */
            BYTE col;               /* xlflowGoto */
            BYTE xlflow;
        } flow;                     /* xltypeFlow */
        struct
        {
            union
            {
                BYTE far *lpbData;  /* data passed to XL */
                HANDLE hdata;       /* data returned from XL */
            } h;
            long cbData;
        } bigdata;                  /* xltypeBigData */
    } val;
    WORD xltype;
} XLOPER, far *LPXLOPER;
```

認識了 xloper 的資料結構之後，爲了瞭解其如何在 Excel 與 XLL 之間傳送資料，在這裡以第四章 bootstrap 的程式爲例，示範如何處理這類具有較複雜資料型態的財金問題。

首先回顧並分析第四章 bootstrap 的程式，主要可分爲以下三個部份：

1. 使用者輸入：包含利息、期間以及殖利率（以陣列儲存）。
2. 計算：利用拔靴法計算每一期的零息利率。
3. 結果回傳：將每一期零息利率（以陣列儲存）回傳。

從以上的分析看來，可以瞭解第二部分對於撰寫 XLL 檔案沒有什麼特別的差異，但反觀第一及第三部分，以陣列儲存的殖利率以及零息利率就必須以先前介紹的 xloper 來處理，以便分別讀入 XLL 以及輸出至 Excel。

也因此若與第四章的程式比較，本節程式主要不同點即在於殖利率的輸入以及回傳零息利率這兩個部分。簡單而言，由於使用者在輸入變數時是在 Excel 的工作環境中，所以資料無法直接以普通的陣列儲存，而必須靠 xloper 讀入以陣列表示的殖利率；此外，在回傳零息利率至 Excel 時，同樣的需要以 xloper 來處理，以便將資料轉換爲 Excel 所能支援的型態。

在函式一開始宣告變數及回傳型態的部分，必須針對殖利率和回傳的零息利率宣告特殊的資料型態：XLOPER *。也就是相對於第四章原始程式，將多期的殖利率資料存入以 XLOPER 資料結構爲資料型態的變數 YieldRate。同時宣告一個 XLOPER 資料結構的 xResult 取代原始程式的陣列 ZeroRate[]，因此最後的回傳結果，原始程式以迴圈回傳 ZeroRate[]陣列的值，此處則直接回傳&xResult 即可。

```
_declspec(dllexport) XLOPER * WINAPI bootstrap(double FV,double coupon,int period,XLOPER * YieldRate)
{
    static XLOPER xResult;
      .
      .
      .
      .
      .
      .
    return &xResult;
}
```

雖然最後直接回傳 xResult 的位置即可，但在本例中的結果爲多個零息利率並讓 Excel 以多重格位的方式回傳，因此使用者必須先替所有回傳值指配空間，包含其列數以及欄位。

```
_declspec(dllexport) XLOPER * WINAPI bootstrap(doub
{
    static XLOPER xResult;
    int i,j,k;

    set_to_xltypeMulti(&xResult,1,period);
```

```
BOOL set_to_xltypeMulti(XLOPER *p_op,WORD rows, WORD cols)
{
    int size=rows*cols;
    if(!p_op||!size)
        return FALSE;

    p_op->xltype=xltypeMulti;
    p_op->val.array.lparray=(XLOPER *) malloc(sizeof(XLOPER)*size);

    p_op->val.array.rows=rows;
    p_op->val.array.columns=cols;

    return TRUE;
}
```

指定資料型態爲 array

配置記憶體空間

指定列、欄位

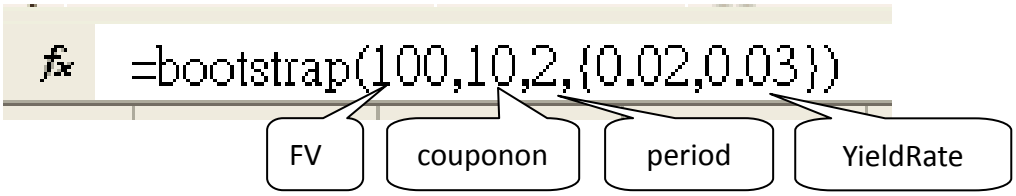
此函式的目的即爲將 xResult 的型態設定爲 Excel 裡的多重格位，並根據期數指定其記憶體大小。

至此程式的第一及第三部分的修改已完成，接著則回到第二部分：計算。由於之前所有以陣列表達的資料都已修改成 XLOPER 的資料型態，也就是 xResult 以及 YieldRate，因此這部分的修改僅需將第四章程式中原先的陣列完全取代即可。但其中值得注意的是，將數值資料儲存在 xResult 裡時，必須先將資料型態定義爲 xltypeNum（數字）。

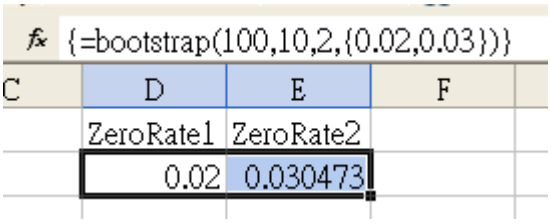
```
    set_to_xltypeMulti(&xResult,1,period);

    xResult.val.array.lparray[0].xltype=xltypeNum;
    xResult.val.array.lparray[0].val.num=YieldRate->val.array.lparray[0].val.num;

    for(i=1;i<=2;i++)
    {
        double BondValue=0;
        for(j=0;j<=i;j=j++)
        {
            double Discount=1;
            for(k=0;k<=j;k++)
                Discount=Discount/(1+YieldRate->val.array.lparray[i].val.num);
            BondValue=BondValue+Discount*coupon;
            if(j==i)
                BondValue=BondValue+Discount*FV;
        }
        for(j=0;j<i;j=j+1)
        {
            double PV=coupon;
            for(k=0;k<=j;k++)
                PV=PV/(1+xResult.val.array.lparray[j].val.num);
            BondValue=BondValue-PV;
        }

        xResult.val.array.lparray[i].xltype=xltypeNum;
        xResult.val.array.lparray[i].val.num=pow((coupon+FV)/BondValue,1.0/(i+1))-1;
    }

    return &xResult;
}
```

接著的步驟則完全雷同於前一節的介紹，包括至 g_rgWorksheetFuncs 函式裡做公式的各種相關設定，以及至 generic.def 檔中將 bootstrap 函式輸出等等。

程式至此完全完成之後，如同前幾節透過 Excel 增益集的方式將公式匯入 Excel 中使用。而使用公式時輸入的各期殖利率必須以陣列的方式輸入，因此各期殖利率以逗點分隔並以大括號表示陣列。例如下圖即爲透過面額 100 元，每期利息爲 10 元，共兩期而殖利率分別爲 0.02、0.03 的債券來計算零息利率的例子。



此外，由於回傳的零息利率也是一個陣列（個數由使用者所輸入的 period 決定），所以使用者完成所有變數的輸入後，必須先將回傳數值個數的格位圈選起來，再按 Ctrl+Shift+Enter 才能將陣列結果完整的表達出來。

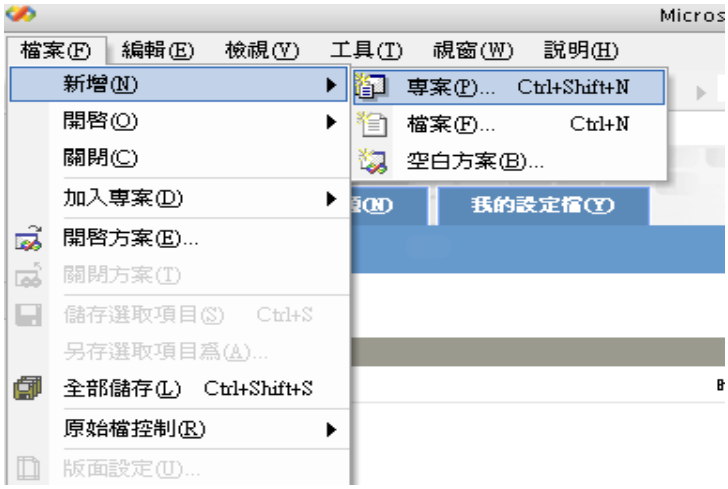


第六節　以 Microsoft Visual Studio .Net 2003 撰寫 DLL 與 XLL

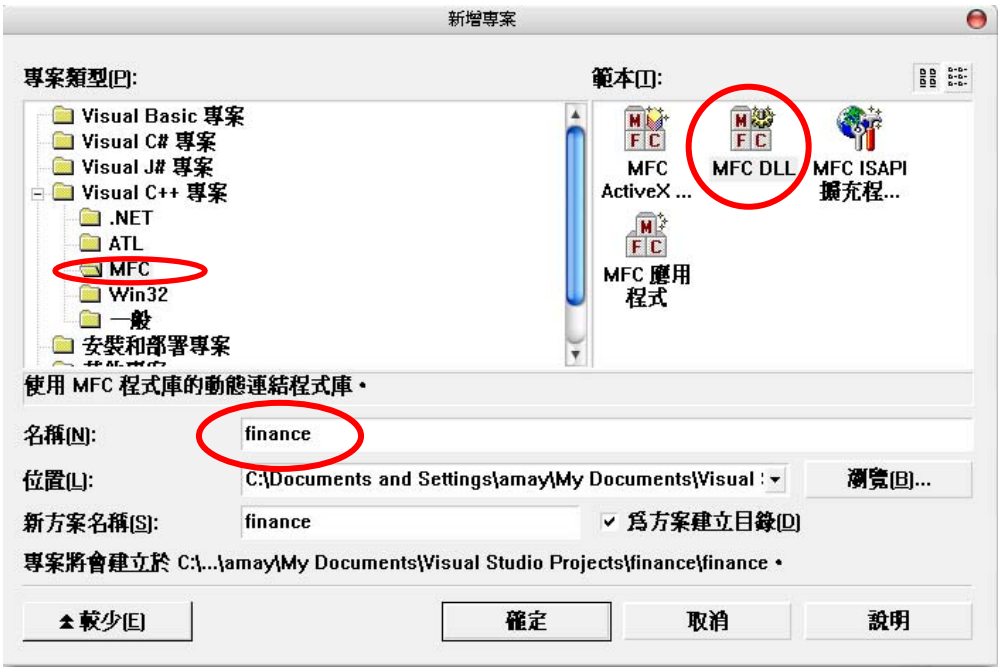
在前幾節的內容中都以 Microsoft Visual C++ 6.0 爲平台示範，現在則另外以 Microsoft Visual Studio .Net 2003 示範 DLL 與 XLL 在 Excel 的應用方法。

（一）DLL，同樣以評價零息債券為例。

首先開啓工作環境後，新增一個專案

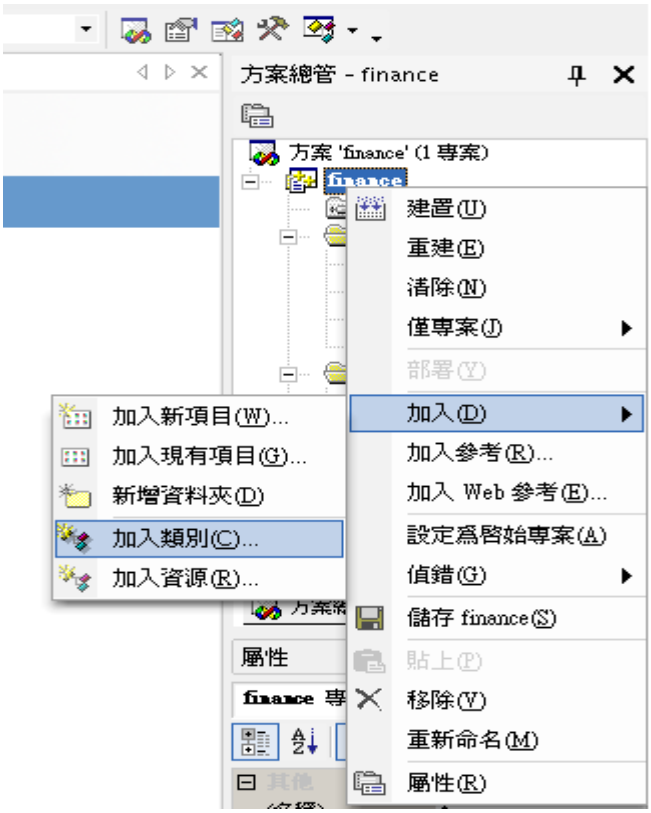


接著選取專案類型爲 Visual C++專案/MFC，以及範本爲 MFC DLL，並在下方爲此專案命名。



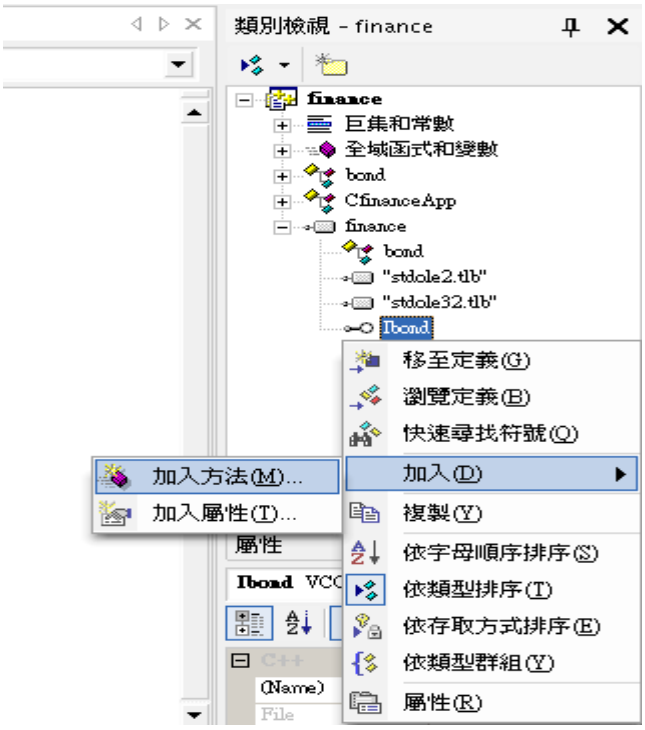
之後將跳出視窗針對應用程式做相關設定。由於之後會再透過 Excel 對完成的 DLL 進行呼叫以及應用，因此這裡必須勾選 Automation，表示將 DLL 物件公開給指令碼工具或其他應用程式。

專案完成後加入類別，也同樣地將類別範本選取爲 MFC 類別。

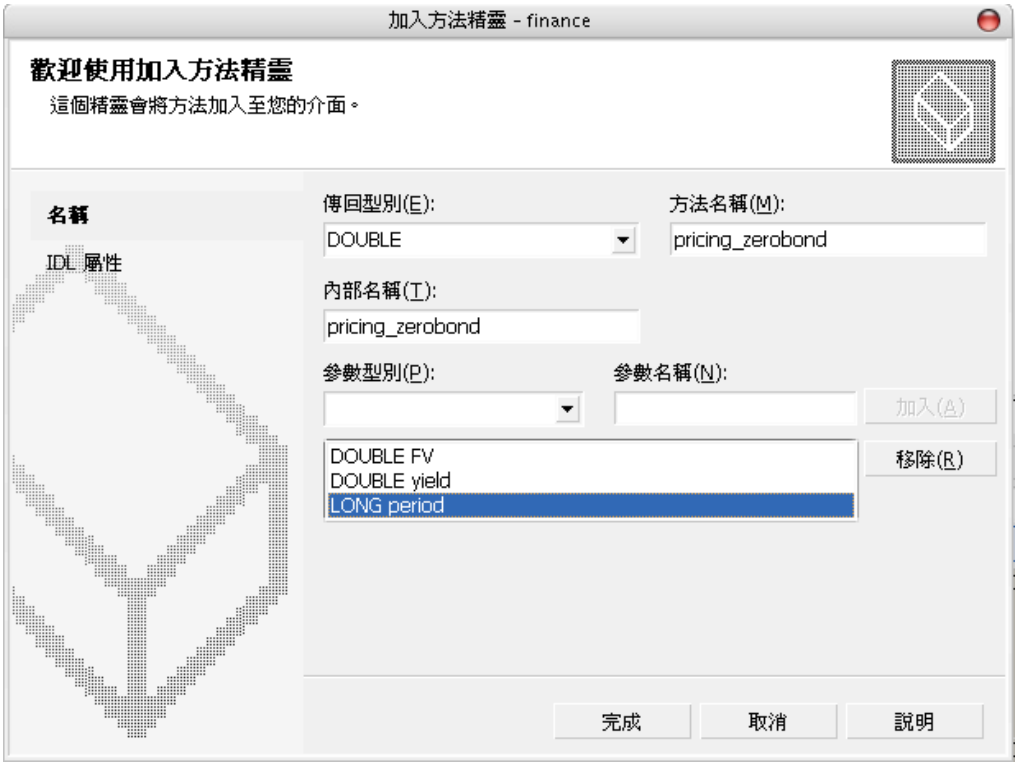


接著針對此新增的類別做設定，包含類別名稱以及基底類別等設定，這裡基底類別選擇 CCmdTarget，除了是常用而且重要的類別以外，CCmdTarget 也提供訊息處理的機能，可以接受命令訊息並找到與命令對應的處理函式。

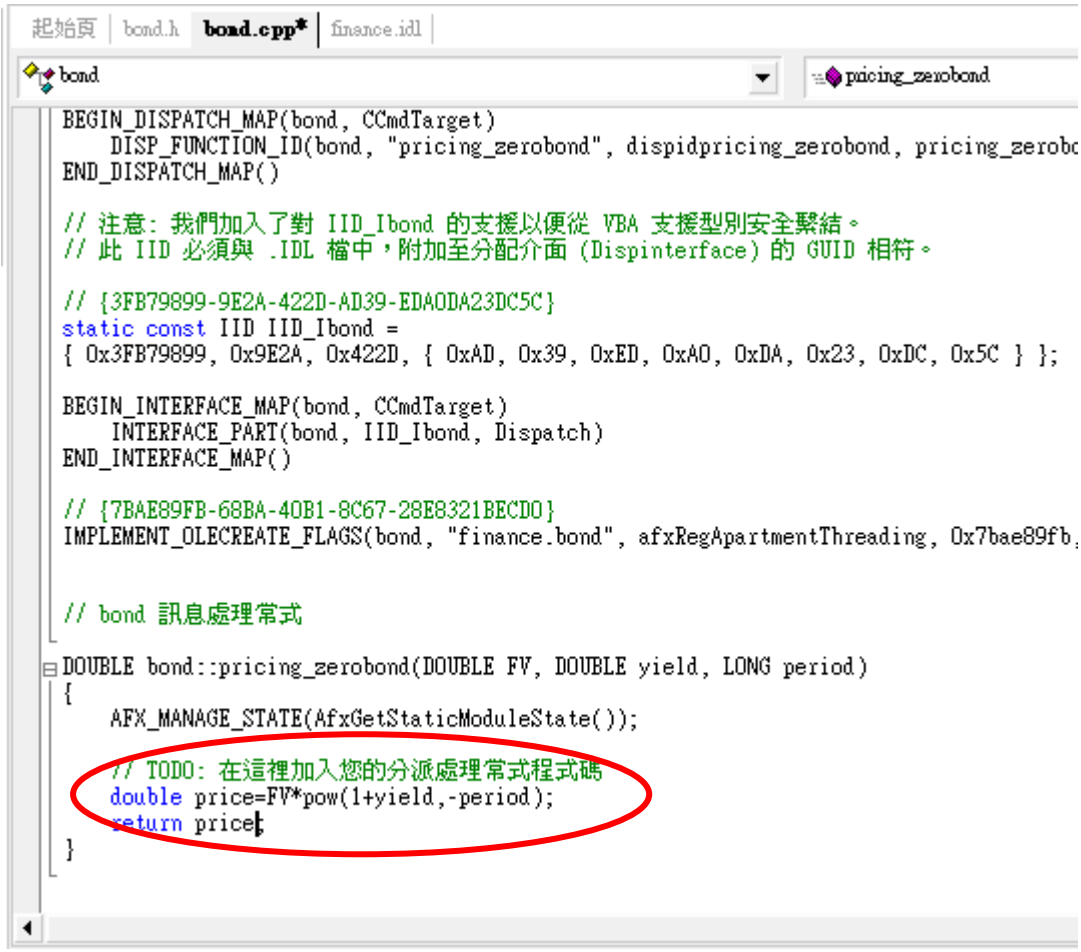
類別設定完成後，繼續類似步驟加入方法。



透過精靈的設定，決定方法的名稱、傳回型別以及參數的名稱與型別。以評價零息債券而言，所需參數分別爲面值（double）、殖利率（double）和期數（long），而最後的回傳型別則爲 double。

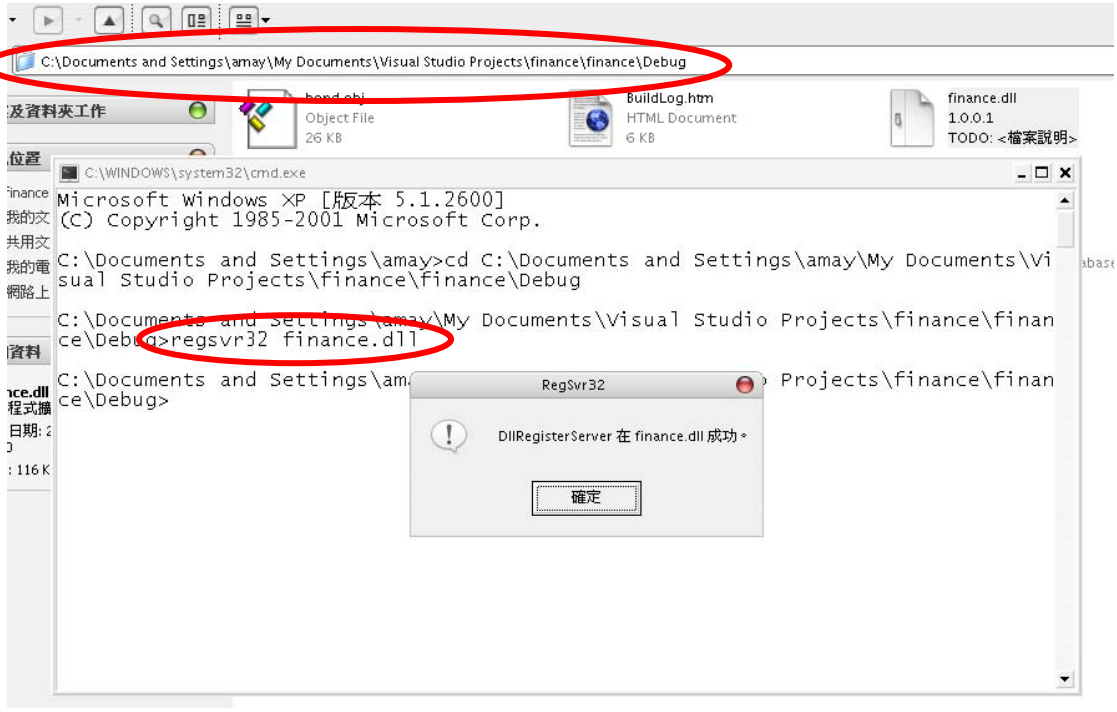


一切設定妥當後便可開始撰寫所需的程式碼。到「類別名稱.cpp」下找到剛加入的方法，接著便可針對使用者的需求加入程式碼，由於都是在評價零息債券，所以這部份的程式碼除稍微經過簡化外，均與前幾節的程式相仿。

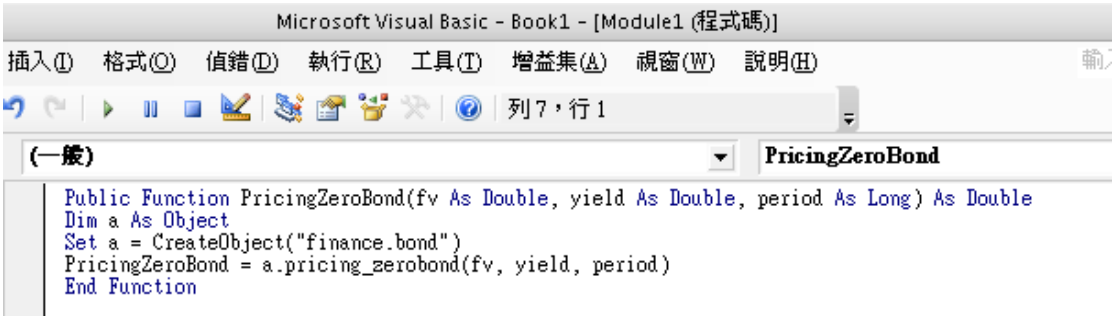


程式碼也完成之後便可建置方案，準備完成 DLL。

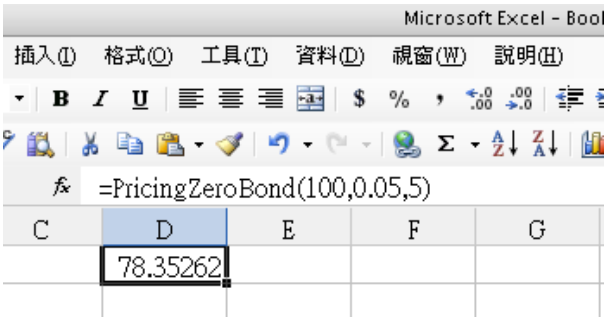
至此 DLL 已建構完成，但仍須再透過一個步驟將 DLL 登錄至系統中。打開命令提示字元（開始 / 所有程式 / 附屬應用程式 或 開始 / 執行 / cmd），將路徑指定到剛建構完成 DLL 檔案的存放位置，並將此 DLL 檔案做登錄動作。



接著就可以回到 Excel 中應用此完成的 DLL，打開 VBA，仿照前一部份的示範，在 VBA 專案插入模組並加入呼叫 DLL 的程式碼。此處程式碼的意義爲宣告一個名爲 finance 函式庫下 bond 類別的 a 物件（程式碼第二行表示宣告一個 a 物件，第三行則是將 a 物件內容指定爲 finance.bond），並將自行設定的 Excel 公式內容指定爲 a 物件之下的 pricing_zerobond 方法。

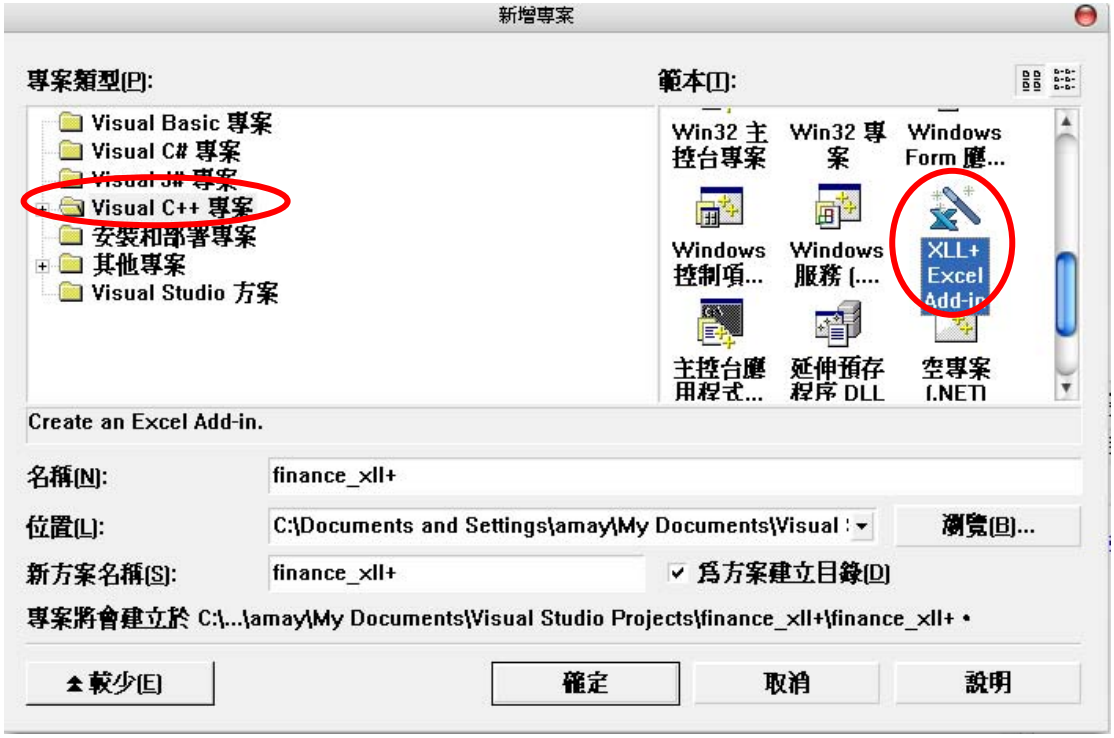


呼叫程式碼完成後便可直接在 Excel 中使用公式了。

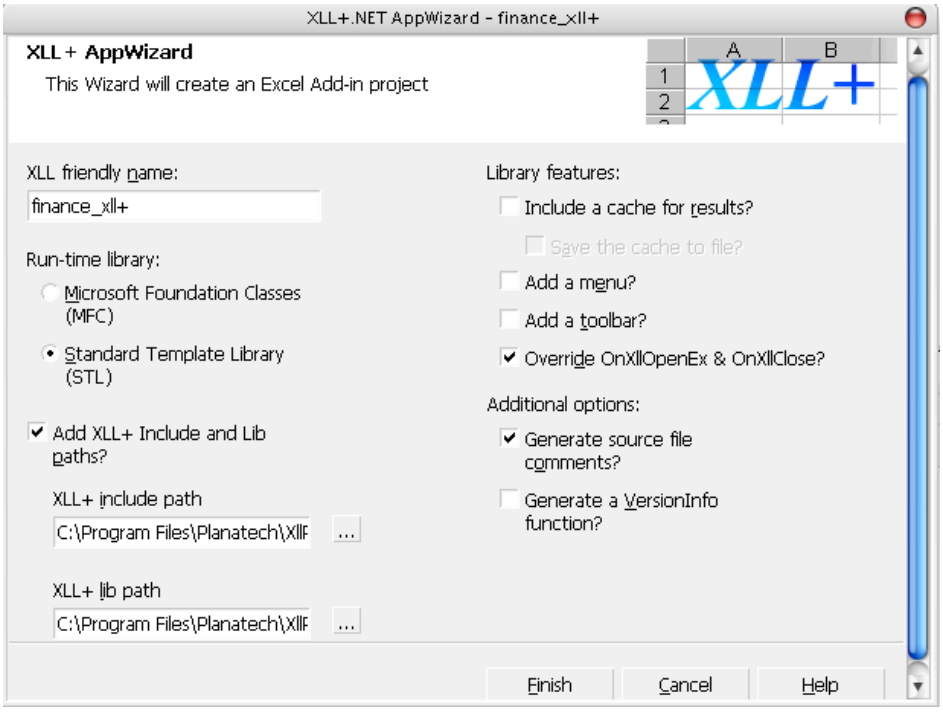


（二）XLL 套裝軟體，XLL Plus Toolkit

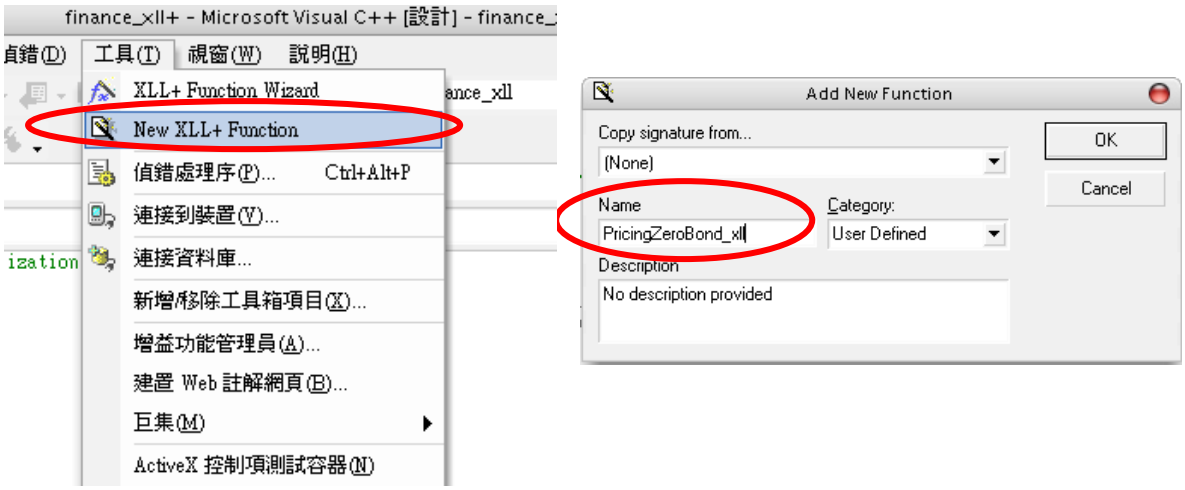
如同在 VC6 中使用 XLL Plus Toolkit 一般，安裝完此套裝軟體之後，在 Microsoft Visual Studio .Net 2003 上新增專案，會發現 Visual C++專案範本裡較安裝 XLL Plus Toolkit 前多了「XLL+ Excel Add-in」這個選項，首先選擇這個範本並爲專案命名。



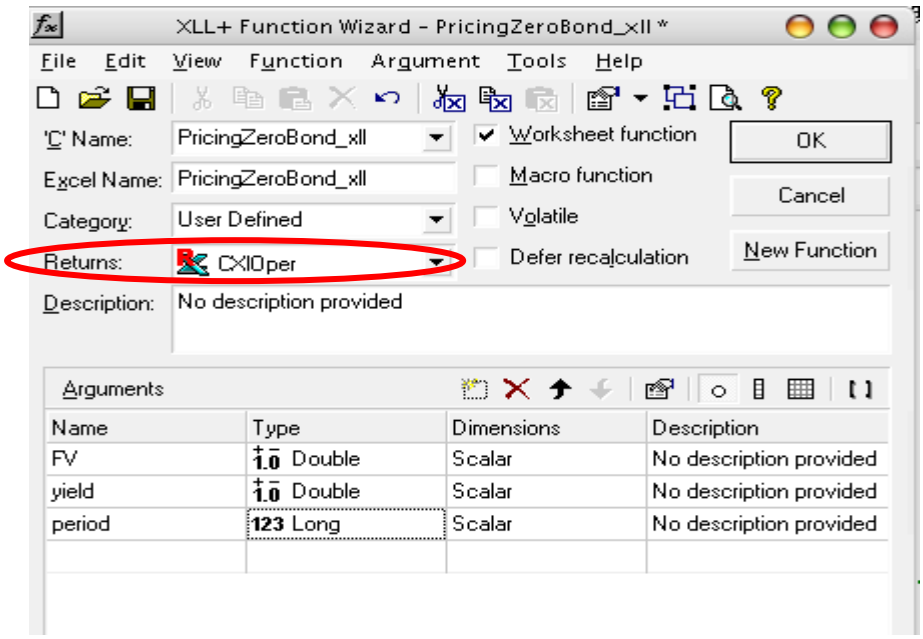
之後會跳出設定精靈的視窗，基本上不用作任何更動。



接著按工具 / New XLL+ Function 以新增函式，並在之後的新增函式視窗爲此函式命名。



命名完成後接著針對資料回傳型態以及所需參數進行設定，此處預設的回傳資料型態 CXlOper 與先前介紹過的 xloper 基本相同，都是能讓 Excel 與 C++之間傳遞資料的特殊資料結構。使用者在這裡輸入的參數也可以套用 CXlOper 資料型態，設計出例如 bootstrap 較複雜的程式。（bootstrap 詳細內容請見第五節，此處範例程式則收錄在本節程式碼中。）



參數和回傳型態決定之後便可加入程式碼並建置方案，XLL 檔案也就此完成，之後可直接回 Excel 的增益集將此 XLL 檔案匯入使用。（內容與步驟如同第三節介紹）

第七節 結語

爲了加強 Excel 處理財務計算問題的效能，本章介紹了動態連結函式庫，並示範如何以 C＋＋撰寫動態連結函式庫以及使用 Excel 呼叫函式以進行運算，其中包含了 DLL 的撰寫、呼叫和 XLL 的撰寫、呼叫，而 XLL 的撰寫方式又可分爲 XLL Plus Toolkit（套裝軟體）與 Excel 97 SDK 兩大部分，後者除了能處理簡單的單一數字問題以外，更能支援較複雜的資料型態。

	簡介	Microsoft Visual C++ 6.0	Microsoft Visual Studio .NET 2003
DLL	前言	第一節	第六節
XLL	第二節	第三節：XLL Plus Toolkit 第四節：Excel 97 SDK 第五節：XLOPER 資料結構	

在本章中花了相當的篇幅示範各類型的財務程式，只願使用者在應用 Excel 解決財務計算問題時能達到方便而又高效率的要求。