
《深入淺出 MFC》2/e 電子書開放自由下載聲明

致親愛的大陸讀者

我是侯捷（侯俊傑）。自從華中理工大學於 1998/04 出版了我的《深入淺出 MFC》1/e 簡體版（易名《深入淺出 Windows MFC 程序設計》）之後，陸陸續續我收到了許許多多的大陸讀者來函。其中對我的讚美、感謝、關懷、殷殷垂詢，讓我非常感動。

《深入淺出 MFC》2/e 早已於 1998/05 於臺灣出版。之所以遲遲沒有授權給大陸進行簡體翻譯，原因我曾於回覆讀者的時候說過很多遍。我在此再說一次。

1998 年中，本書之發行公司松崗（UNALIS）即希望我授權簡體版，然因當時我已在構思 3/e，預判 3/e 繁體版出版時，2/e 簡體版恐怕還未能完成。老是讓大陸讀者慢一步看到我的書，令我至感難過，所以便請松崗公司不要進行 2/e 簡體版之授權，直接等 3/e 出版後再動作。沒想到一拖經年，我的 3/e 寫作計劃並沒有如期完成，致使大陸讀者反而沒有《深入淺出 MFC》2/e 簡體版可看。

《深入淺出 MFC》3/e 沒有如期完成的原因是，MFC 本體架構並沒有什麼大改變。《深入淺出 MFC》2/e 書中所論之工具及程式碼雖採用 VC5+MFC42，仍適用於目前的 VC6+MFC421（唯，工具之畫面或功能可能有些微變化）。

由於《深入淺出 MFC》2/e 並無簡體版，因此我時時收到大陸讀者來信詢問購買繁體版之管道。一來我不知道是否臺灣出版公司有提供海外郵購或電購，二來即使有，想必帶給大家很大的麻煩，三來兩岸消費水平之差異帶給大陸讀者的負擔，亦令我深感不安。

因此，此書雖已出版兩年，鑑於仍具閱讀與技術上的價值，鑑於繁簡轉譯製作上的費時費工，鑑於我對同胞的感情，我決定開放此書內容，供各位免費閱讀。我已為《深入淺出 MFC》2/e 製作了 PDF 格式之電子檔，放在 <http://www.jjhou.com> 供自由下載。北京 <http://expert.csdn.net/jjhou> 有侯捷網站的一個 GBK mirror，各位也可試著自該處下載。

我所做的這份電子書是繁體版，我沒有精力與時間將它轉為簡體。這已是我能為各位盡力的極限。如果（萬一）您看不到檔案內容，可能與字形的安裝有關——雖然我已嘗試內嵌字形。anyway，閱讀方面的問題我亦沒有精力與時間為您解決。請各位自行開闢討論區，彼此交換閱讀此電子書的 solution。請熱心的讀者告訴我您閱讀成功與否，以及網上討論區（如有的話）在哪裡。

曾有讀者告訴我，《深入淺出 MFC》1/e 簡體版在大陸被掃描上網。亦有讀者告訴我，大陸某些書籍明顯對本書侵權（詳細情況我不清楚）。這種不尊重作者的行為，我雖感遺憾，並沒有太大的震驚或難過。一個社會的進化，終究是一步一步衍化而來。臺灣也曾經走過相同的階段。但盼所有華人，尤其是我們從事智慧財產行為者，都能夠儘快走過灰暗的一面。

在現代科技的協助下，文件影印、檔案複製如此方便，智財權之尊重有如「君子不欺暗室」。沒有人知道我們私下的行為，只有我們自己心知肚明。《深入淺出 MFC》2/e 雖免費供大家閱讀，但此種作法實非長久之計。為計久長，我們應該尊重作家、尊重智財，以良好（至少不差）的環境培養有實力的優秀技術作家，如此才有源源不斷的好書可看。

我的近況，我的作品，我的計劃，各位可從前述兩個網址獲得。歡迎各位寫信給我（jjhou@ccca.nctu.edu.tw）。雖然不一定能夠每封來函都回覆，但是我樂於知道讀者的任何點點滴滴。

關於《深入淺出 MFC》2/e 電子書

《深入淺出 MFC》2/e 電子書共有五個檔案：

檔名	內容	大小 bytes
dissecting MFC 2/e part1.pdf	chap1~chap3	3,384,209
dissecting MFC 2/e part2.pdf	chap4	2,448,990
dissecting MFC 2/e part3.pdf	chap5~chap7	2,158,594
dissecting MFC 2/e part4.pdf	chap8~chap16	5,171,266
dissecting MFC 2/e part5.pdf	appendix A,B,C,D	1,527,111

每個檔案都可個別閱讀。每個檔案都有書籤（亦即目錄連結）。每個檔案都不需密碼即可開啓、選擇文字、列印。

請告訴我您的資料

每一位下載此份電子書的朋友，我希望您寫一封 email 給我（jjhou@ccca.nctu.edu.tw），告訴我您的以下資料，俾讓我對我的讀者有一些基本瞭解，謝謝。

姓名：

現職：

畢業學校科系：

年齡：

性別：

居住省份（如是臺灣讀者，請寫縣市）：

對侯捷的建議：

-- the end

4

深入 MFC 程式設計



Document-View 深入探討

形而上者謂之道，形而下者謂之器。

對於 Document/View 而言，很少有人能夠先道而後器。

完全由 AppWizard 代勞做出的 Scribble step0，應用程式的整個架構（空殼）都已經建構起來了，但是 Document 和 View 還空著好幾個最重要的函式（都是虛擬函式）等著你設計其實體。這就像一部汽車外面的車體以及內部的油路電路都裝配好了，但還等著最重要的發動機（引擎）植入，才能夠產生動力，開始「有所為」。

我已經在第 7 章概略介紹了 Document/View 以及 Document Template，還有更多的秘密將在本章揭露。

為什麼需要 Document-View（形而下）

MFC 之所以為 Application Framework，最重要的一個特徵就是它能夠將管理資料的程式碼和負責資料顯示的程式碼分離開來，這種能力由 MFC 的 Document/View 提供。Document/View 是 MFC 的基石，了解它，對於有效運用 MFC 有極關鍵的影響。甚至 OLE 複合文件（compound document）都是建築在 Document/View 的基礎上呢！

幾乎每一個軟體都致力於資料的處理，畢竟資訊以及資料的管理是電腦技術的主要用途。把資料管理和顯示方法分離開來，需要考慮下列幾個議題：

1. 程式的哪一部份擁有資料
2. 程式的哪一部份負責更新資料
3. 如何以多種方式顯示資料
4. 如何讓資料的更改有一致性
5. 如何儲存資料（放到永久儲存裝置上）
6. 如何管理使用者介面。不同的資料型態可能需要不同的使用者介面，而一個程式可能管理多種型態的資料。

其實 Document / View 不是什麼新主意，Xerox PARC 實驗室是這種觀念的濫觴。它是 Smalltalk 環境中的關鍵性部份，在那裡它被稱為 **Model-View-Controller (MVC)**。其中的 Model 就是 MFC 的 Document，而 Controller 相當於 MFC 的 Document Template。

回想在沒有 Application Framework 幫助的時代（並不太久以前），你如何管理資料？只要程式需要，你就必須想出各種表現資料的方法；你有責任把資料的各種表現方法和資料本體調解出一種關係出來。100 位程式員，有 100 種作法！如果你的程式只處理一種資料型態，情況還不至於太糟。舉個例，文字處理軟體可以使用巨大的字串陣列，把文字統統含括進來，並以 ASCII 型式顯示之，頂多嘛，變換一下字形！

但如果你必須維護一種以上的資料型態，情況又當如何？想像得到，每一種資料型態可能需要獨特的處理方式，於是需要一套功能選單；每一種資料型態顯現在視窗中，應該有獨特的視窗標題以及縮小圖示；當資料編輯完畢要存檔，應該有獨特的副檔名；登錄在 Registry 之中應該有獨特的型號。再者，如果你以不同的視窗，不同的顯現方式，秀出一份資料，當資料在某一視窗中被編輯，你應該讓每一視窗的資料顯像與實際資料之間常保一致。吧啦吧啦吧啦；K 繁雜事務不勝枚舉。

很快地，問題就浮顯出來了。程式不僅要做資料管理，更要做「與資料型態相對應的 UI」的管理。幸運的是，解決之道亦已浮現，那就是物件導向觀念中的 **Model-View-Controller (MVC)**，也就是 MFC 的 Document / View。

Document

名稱有點令人懼怕 -- *Document* 令我們想起文書處理軟體或試算表軟體中所謂的「文件」。但，這裡的 *Document* 其實就是資料。的確是，不必想得過份複雜。有人用 *data set* 或 *data source* 來表示它的意義，都不錯。

Document 在 MFC 的 *CDocument* 裡頭被具體化。*CDocument* 本身並無實務貢獻，它只是提供一個空殼。當你開發自己的程式，應該從 *CDocument* 衍生出一個屬於自己的 *Document* 類別，並且在類別中宣告一些成員變數，用以承載（容納）資料。然後再（至少）改寫專門負責檔案讀寫動作的 *Serialize* 函式。事實上，AppWizard 為我們把空殼都準備好了，以下是 *Scribble step0* 的部份內容：

```
class CScribbleDoc : public CDocument
{
    DECLARE_DYNCREATE(CScribbleDoc)
    ...
    virtual void Serialize(CArchive& ar);
    DECLARE_MESSAGE_MAP()
};

void CScribbleDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

由於 *CDocument* 衍生自 *CObject*，所以它就有了 *CObject* 所支援的一切性質，包括執行時期型別資訊（RTTI）、動態生成（Dynamic Creation）、檔案讀寫（Serialization）。又由於它也衍生自 *CCmdTarget*，所以它可以接收來自選單或工具列的 *WM_COMMAND* 訊息。

View

View 負責描述（呈現）Document 中的資料。

View 在 MFC 的 *CView* 裡頭被具體化。*CView* 本身亦無實務貢獻，它只是提供一個空殼。當你開發自己的程式，應該從 *CView* 衍生出一個屬於自己的 View 類別，並且在類別中（至少）改寫專門負責顯示資料的 *OnDraw* 函式（針對螢幕）或 *OnPrint* 函式（針對印表機）。事實上，AppWizard 為我們把空殼都準備好了，以下是 Scribble step0 的部份內容：

```
class CScribbleView : public CView
{
    DECLARE_DYNCREATE(CScribbleView)
    ...
    virtual void OnDraw(CDC* pDC);
    DECLARE_MESSAGE_MAP()
};

void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}
```

由於 *CView* 衍生自 *CWnd*，所以它可以接收一般 Windows 訊息（如 *WM_SIZE*、*WM_PAINT* 等等），又由於它也衍生自 *CCommandTarget*，所以它可以接收來自選單或工具列的 *WM_COMMAND* 訊息。

在傳統的 C/SDK 程式中，當視窗函式收到 *WM_PAINT*，我們（程式員）就呼叫 *BeginPaint*，獲得一個 Device Context（DC），然後在這個 DC 上作畫。這個 DC 代表螢幕裝置。在 MFC 裡頭，一旦 *WM_PAINT* 發生，Framework 會自動呼叫 *OnDraw* 函式。

View 事實上是個沒有邊框的視窗。真正出現時，其外圍還有一個有邊框的視窗，我們稱為 Frame 視窗。

Document Frame (View Frame)

如果你的程式管理兩種不同型態的資料，譬如說一個是 TEXT，一個是 BITMAP，作為一位體貼的程式設計者，我想你很願意為你的使用者考慮多一些：你可能願意在使用者操作 TEXT 資料時，換一套 TEXT 專屬的使用者介面，在使用者操作 BITMAP 資料時，換一套 BITMAP 專屬的使用者介面。這份工作正是由 Frame 視窗負責。

乍見這個觀念，我想你會驚訝為什麼 UI 的管理不由 View 直接負責，卻要交給 Frame 視窗？你知道，有時候機能與機能之間要有點黏又不太黏才好，把 UI 管理機能隔離出來，可以降低彼此之間的依存性，也可以使機能重複使用於各種場合如 SDI、MDI、OLE in-place editing（即地編輯）之中。如此一來 View 的彈性也會大一些。

Document Template

MFC 把 Document/View/Frame 視為三位一體。可不是嗎！每當使用者欲打開（或新增）一份文件，程式應該做出 Document、View、Frame 各一份。這個「三口組」成為一個運作單元，由所謂的 Document Template 掌管。MFC 有一個 *CDocTemplate* 負責此事。它又有兩個衍生類別，分別是 *CMultiDocTemplate* 和 *CSingleDocTemplate*。

所以我在上一章說了，如果你的程式能夠處理兩種資料型態，你必須製造兩個 Document Template 出來，並使用 *AddDocTemplate* 函式將它們一一加入系統之中。這和程式是不是 MDI 並沒有關係。如果你的程式支援多種資料型態，但卻是個 SDI，那只不過表示你每次只能開啓一份文件罷了。

但是，逐漸地，MDI 這個字眼與它原來的意義有了一些出入（要知道，這個字眼早在 SDK 時代即有了）。因此，你可能會看到有些書籍這麼說：『MDI 程式使用 *CMultiDocTemplate*，SDI 程式使用 *CSingleDocTemplate*』，那並不是很精準。

CDocTemplate 是個抽象類別，定義了一些用來處理「Document/View/Frame 三口組」的基礎函式。

CDocTemplate 管理 CDocument / CView / CFrameWnd

好，我們說 Document Template 管理「三口組」，誰又來管理 Document Template 呢？

答案是 *CWinApp*。下面就是 *InitInstance* 中應有的相關作為：

```
BOOL CScribbleApp::InitInstance()  
{  
    ...  
    CMultiDocTemplate* pDocTemplate;  
    pDocTemplate = new CMultiDocTemplate(  
        IDR_SCRIBTYPE,  
        RUNTIME_CLASS(CScribbleDoc),  
        RUNTIME_CLASS(CChildFrame),  
        RUNTIME_CLASS(CScribbleView));  
    AddDocTemplate(pDocTemplate);  
    ...  
}
```

想一想文件是怎麼開啓的：使用者選按【File/New】或【File/Open】（前者開啓一份空文件，後者讀檔放到文件中），然後在 View 視窗內展現出來。我們很容易誤以爲是 *CWinApp* 直接產生 Document：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)  
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)  
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)  
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)  
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)  
END_MESSAGE_MAP()
```

其實才不，是 Document Template 的傑作：

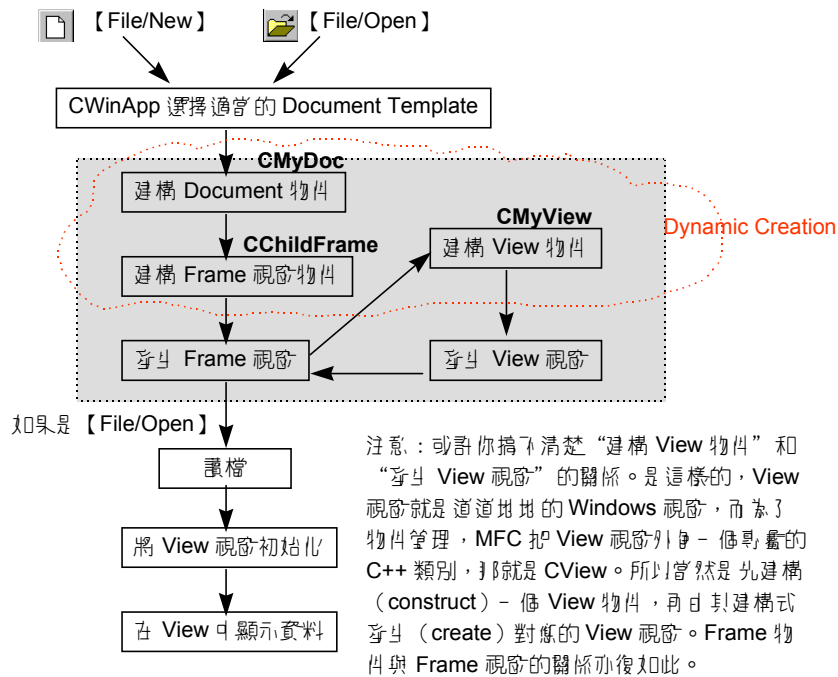


圖 8-1 Document/View/Frame 的產生

圖 8-1 的灰色部份，正是 Document Template 動態產生「三位一體之 Document/View/Frame」的行動。下面的流程以及 MFC 原始碼足以澄清一切疑慮。在 *CMultiDocTemplate::OpenDocumentFile*（註）出現之前的所有流程，我只做文字敘述，不顯示其原始碼。本章稍後有一節「檯面下的 Serialize 讀檔奧秘」，則會將每一環節的原始碼呈現在你眼前，讓你無所掛慮。

註：如果是 SDI 程式，那麼就是 *CSingleDocTemplate::OpenDocumentFile* 被呼叫。但「多」比「單」有趣，而且本書範例 *Scribble* 程式也使用 *CMultiDocTemplate*，所以我就以此為說明對象。

CSingleDocTemplate 只支援一種文件型態，所以它的成員變數是：

```
class CSingleDocTemplate : public CDocTemplate
{
...
protected: // standard implementation
    CDocument* m_pOnlyDoc;
};
```

CMultiDocTemplate 支援多種文件型態，所以它的成員變數是：

```
class CMultiDocTemplate : public CDocTemplate
{
...
protected: // standard implementation
    CPtrList m_docList;
};
```

當使用者選按【File/New】命令項，根據 AppWizard 為我們所做的 Message Map，此一命令由 *CWinApp::OnFileNew* 接手處理。後者呼叫 *CDocManager::OnFileNew*，後者再呼叫 *CWinApp::OpenDocumentFile*，後者再呼叫 *CDocManager::OpenDocumentFile*，後者再呼叫 *CMultiDocTemplate::OpenDocumentFile*（這是觀察 MFC 原始碼所得結果）：

```
// in AFXWIN.H
class CDocTemplate : public CCmdTarget
{
...
    UINT m_nIDResource; // IDR_ for frame/menu/accel as well
    CRuntimeClass* m_pDocClass; // class for creating new documents
    CRuntimeClass* m_pFrameClass; // class for creating new frames
    CRuntimeClass* m_pViewClass; // class for creating new views
    CString m_strDocStrings; // '\n' separated names
...
}

// in DOCMULTI.CPP
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
    BOOL bMakeVisible)
{
    CDocument* pDocument = CreateNewDocument();
    ...
    CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL);
    ...
    if (lpszPathName == NULL)
    {
        // create a new document - with default document name
    }
}
```

```

        ...
    }
    else
    {
        // open an existing document
        ...
    }
    InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
    return pDocument;
}

```

顧名思義，我們很容易作出這樣的聯想：*CreateNewDocument* 動態產生 Document，*CreateNewFrame* 動態產生 Document Frame。的確是這樣沒錯，它們利用 *CRuntimeClass* 的 *CreateObject* 做「動態生成」動作：

```

// in DOCTEMPL.CPP
CDocument* CDocTemplate::CreateNewDocument()
{
    ...
    CDocument* pDocument = (CDocument*)m_pDocClass->CreateObject();
    ...
    AddDocument(pDocument);
    return pDocument;
}

CFrameWnd* CDocTemplate::CreateNewFrame(CDocument* pDoc, CFrameWnd* pOther)
{
    // create a frame wired to the specified document
    CCreateContext context;
    context.m_pCurrentFrame = pOther;
    context.m_pCurrentDoc = pDoc;
    context.m_pNewViewClass = m_pViewClass;
    context.m_pNewDocTemplate = this;
    ...
    CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();
    ...
    // create new from resource
    pFrame->LoadFrame(m_nIDResource,
        WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, // default frame styles
        NULL, &context)
    ...
    return pFrame;
}

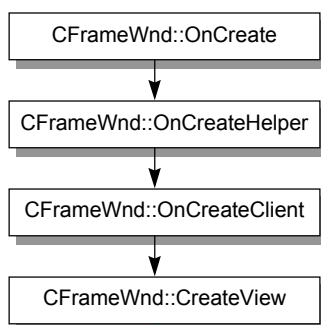
```

在 *CreateNewFrame* 函式中，不僅 Frame 被動態生成出來了，其對應視窗也以

LoadFrame 產生出來了。但有兩件事情令人不解。第一，我們沒有看到 *View* 的動態生成動作；第二，出現一個奇怪的傢伙 *CCreateContext*，而前一個不解似乎能夠著落到這個奇怪傢伙的身上，因為 *CDocTemplate::m_pViewClass* 被塞到它的一個欄位中。

但是線索似乎已經中斷，因為我們已經看不到任何可能的呼叫動作了。等一等！*context* 被用作 *LoadFrame* 的最後一個參數，這意味什麼？還記得第六章「*CFrameWnd::Create* 產生主視窗（並先註冊視窗類別）」那一節提過 *Create* 的最後一個參數嗎，正是這 *context*。那麼，是不是 *Document Frame* 視窗產生之際由於 *WM_CREATE* 的發生而刺激了什麼動作？

雖然其結果是正確的，但這樣的聯想也未免太天馬行空了些。我只能說，經驗累積出判斷力！是的，*WM_CREATE* 引發 *CFrameWnd::OnCreate* 被喚起，下面是相關的呼叫次序（經觀察 MFC 原始碼而得知）：



```
// in WINFRM.CPP
CWnd* CFrameWnd::CreateView(CCreateContext* pContext, UINT nID)
{
    ...
    CWnd* pView = (CWnd*)pContext->m_pNewViewClass->CreateObject();
    ...
    // views are always created with a border!
    pView->Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0,0,0,0), this, nID, pContext))
    ...
    if (afxData.bWin4 && (pView->GetExStyle() & WS_EX_CLIENTEDGE))
    {

```

```

        // remove the 3d style from the frame, since the view is
        // providing it.
        // make sure to recalc the non-client area
        ModifyStyleEx(WS_EX_CLIENTEDGE, 0, SWP_FRAMECHANGED);
    }
    return pView;
}

```

不僅 View 物件被動態生成出來了，其對應的實際 Windows 視窗也以 *Create* 函式產生出來。

正因為 MFC 把 View 物件的動態生成動作包裝得如此詭譎奇險，所以我才在圖 8-1 中把「建構 View 物件」和「產生 View 視窗」這兩個動作特別另立一旁：

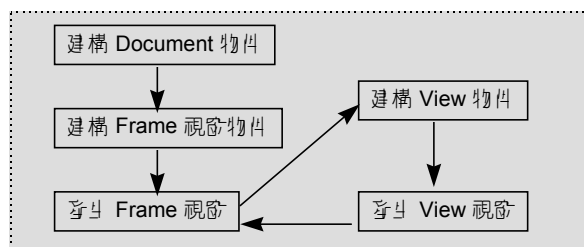


圖 8-2 解釋 *CDocTemplate*、*CDocument*、*CView*、*CFrameWnd* 之間的關係。下面則是一份文字整理：

- *CWinApp* 擁有一個物件指標：*CDocManager* m_pDocManager*。
- *CDocManager* 擁有一個指標串列 *CPtrList m_templateList*，用來維護一系列的 Document Template。一個程式若支援兩「種」文件型態，就應該有兩份 Document Templates，應用程式應該在 *CMyWinApp::InitInstance* 中以 *AddDocTemplate* 將這些 Document Templates 加入由 *CDocManager* 所維護的串列之中。
- *CDocTemplate* 擁有三個成員變數，分別持有 Document、View、Frame 的 *CRuntimeClass* 指標，另有一個成員變數 *m_nIDResource*，用來表示此 Document 顯現時應該採用的 UI 物件。這四份資料應該在 *CMyWinApp::InitInstance* 函式建構 *CDocTemplate*（註 1）時指定之，成為建構式的參數。當使用者欲打開一份文件（通常是藉著【File/Open】或【File/New】命令項），*CDocTemplate* 即可藉由 Document/View/Frame 之 *CRuntimeClass* 指標（註 2）進行動態生成。

註 1：在此我們必須有所選擇，要不就使用 *CSingleDocTemplate*，要不就使用 *CMultiDocTemplate*，兩者都是 *CDocTemplate* 的衍生類別。如果你選用 *CSingleDocTemplate*，它有一個成員變數 *CDocument* m_pOnlyDoc*，亦即它一次只能打開一份 Document。如果你選用 *CMultiDocTemplate*，它有一個成員變數 *CPtrList m_docList*，表示它能同時打開多個 Documents。

註 2：關於 *CRuntimeClass* 與動態生成，我在第 3 章已經以 DOS 程式模擬之，本章稍後亦另有說明。

- *CDocument* 有一個成員變數 *CDocTemplate* m_pDocTemplate*，回指其 Document Template；另有一個成員變數 *CPtrList m_viewList*，表示它可以同時維護一系列的 Views。
- *CFrameWnd* 有一個成員變數 *CView* m_pViewActive*，指向目前正作用中的 View。
- *CView* 有一個成員變數 *CDocument* m_pDocument*，指向相關的 Document。

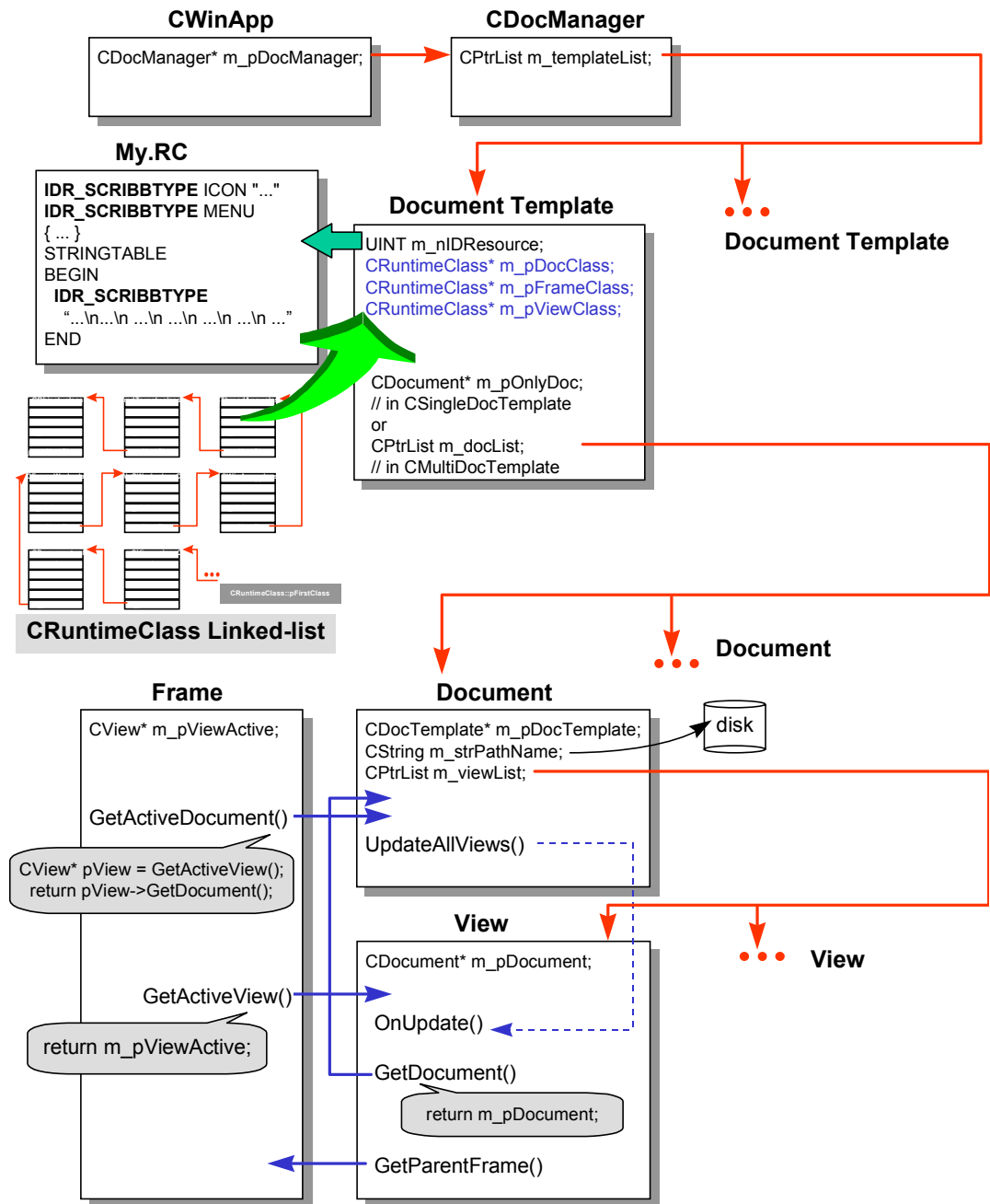


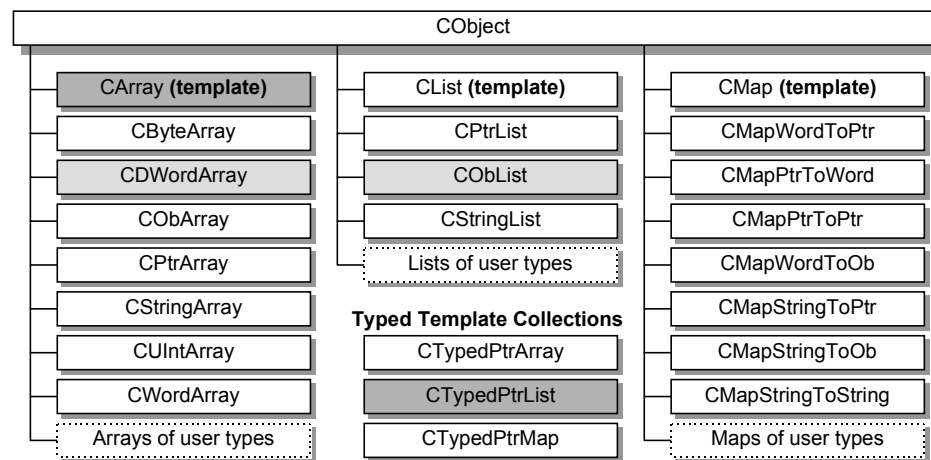
圖 8-2 CDocTemplate、CDocument、CView、CFrameWnd 之間的關係

我把 Document/View/Frame 的觀念以狂風驟雨之勢對你做了一個交待。模糊？晦暗？沒有關係，馬上我們就開始實作 Scribble Step1，你會從實作過程中慢慢體會上述觀念。

Scribble Step1 的 Document - 資料結構設計

Scribble 允許使用者在視窗中畫圖，畫圖的方式是以滑鼠做為畫筆，按下左鍵拖曳拉出線條。每次按下滑鼠左鍵後一直到放開為止的連續座標點構成線條（stroke）。整張圖（整份文件）由線條構成，線條可由點、筆寬、筆色等等資料構成（但本例並無筆色資料）。

MFC 的 Collections Classes 中有許多適用於各種資料型態（如 Byte、Word、DWord、Ptr）以及各種資料結構（如陣列、串列）的現成類別。如果我們儘可能把這些現成的類別應用到程式的資料結構上面，就可以節省許多開發時間：



我們的設計最高原則就是盡量使用 MFC 已有的類別，提高軟體 IC 的重複使用性。上圖淺色部份是 Scribble 範例程式在 16 位元 MFC 中採用的兩個類別。深色部份是 Scribble 範例程式在 32 位元 MFC 中採用的兩個類別。

MFC Collection Classes 的選用

第5章末尾我曾經大致提過 MFC Collection Classes。它們分為三種型態，用來管理一大群物件：

- **Array**：陣列，有次序性（需依序處理），可動態增減大小，索引值為整數。
- **List**：雙向串列，有次序性（需依序處理），無索引。串列有頭尾，可從頭尾或從串列的任何位置安插元素，速度極快。
- **Map**：又稱為 **Dictionary**，其內物件成對存在，一為鍵值物件（**key object**），一為實值物件（**value object**）。

下面是其特性整理：

型態	有序	索引	插入元素	搜尋特定元素	複製元素
List	Yes	No	快	慢	可
Array	Yes	Yes (利用整數索引值)	慢	慢	可
Map	No	Yes (利用鍵值)	快	快	鍵值 (key) 不可複製， 實值 (value) 可複製。

MFC Collection classes 所收集的物件中，有兩種特別需要說明，一是 **Ob** 一是 **Ptr**：

- **Ob** 表示衍生自 *CObject* 的任何物件。MFC 提供 *CObList*、*CObArray* 兩種類別。
- **Ptr** 表示物件指標。MFC 提供 *CPtrList*、*CPtrArray* 兩種類別。

當我們考慮使用 MFC collection classes 時，除了考慮上述三種型態的特性，還要考慮以下幾點：

- 是否使用 C++ template（對於 type-safe 極有幫助）。
- 儲存於 collection class 之中的元素是否要做檔案讀寫動作（Serialize）。
- 儲存於 collection class 之中的元素是否要有傾印（dump）和錯誤診斷能力。

下表是對所有 collection classes 性質的一份摘要整理（參考自微軟的官方手冊：*Programming With MFC and Win32*）：

類別	C++ template	Serializable	Dumpable	type-safe
<i>CArray</i>	Yes	Yes ①	Yes ①	No
<i>CTypedPtrArray</i>	Yes	Depends ②	Yes	Yes
<i>CByteArray</i>	No	Yes	Yes	Yes ③
<i>CDWordArray</i>	No	Yes	Yes	Yes ③
<i>CObArray</i>	No	Yes	Yes	No
<i>CPtrArray</i>	No	No	Yes	No
<i>CStringArray</i>	No	Yes	Yes	Yes ③
<i>CWordArray</i>	No	Yes	Yes	Yes ③
<i>CUIntArray</i>	No	No ④	Yes	Yes ③
<i>CList</i>	Yes	Yes ①	Yes ①	No
<i>CTypedPtrList</i>	Yes	Depends ②	Yes	Yes
<i>CObList</i>	No	Yes	Yes	No
<i>CPtrList</i>	No	No	Yes	No
<i>CStringList</i>	No	Yes	Yes	Yes ③
<i>CMap</i>	Yes	Yes ①	Yes ①	No
<i>CTypedPtrMap</i>	Yes	Depends ②	Yes	Yes
<i>CMapPtrToWord</i>	No	No	Yes	No
<i>CMapPtrToPtr</i>	No	No	Yes	No
<i>CMapStringToOb</i>	No	Yes	Yes	No
<i>CMapStringToPtr</i>	No	No	Yes	No
<i>CMapStringToString</i>	No	Yes	Yes	Yes ③
<i>CMapWordToOb</i>	No	Yes	Yes	No
<i>CMapWordToPtr</i>	No	No	Yes	No

① 若要檔案讀寫，你必須明白呼叫 collection object 的 `Serialize` 函式；若要內容傾印，你必須明白呼叫其 `Dump` 函式。不能夠使用 `archive << obj` 或 `dmp << obj` 這種型式。

② 究竟是否 `Serializable`，必須視其內含物件而定。舉個例，如果一個 `typed pointer array` 是以 `CObArray` 為基礎，那麼它是 `Serializable`；如果它是以 `CPtrArray` 為基礎，那麼它就不是 `Serializable`。一般而言，`Ptr` 都不能夠被 `Serialized`。

③ 雖然它是 `non-template`，但如果照預定計劃去使用它（例如以 `CByteArray` 儲存 `bytes`，而不是用來儲存 `char`），那麼它還是 `type-safe` 的。

④ 手冊上說它並非 `Serializable`，但我存疑。各位不妨試驗之。

Template-Based Classes

本書第2章末尾已經介紹過所謂的 C++ `template`。MFC 的 `collection classes` 裡頭有一些是 `template-based`，對於型態檢驗的功夫做得比較好。這些類別區分為：

- 簡單型 - `CArray`、`CList`、`CMap`。它們都衍生自 `CObject`，所以它們都具備了檔案讀寫、執行時期型別鑑識、動態生成等性質。
- 型態指標型 - `CTypedPtrArray`、`CTypedPtrList`、`CTypedPtrMap`。這些類別要求你在參數中指定基礎類別，而基礎類別必須是 MFC 之中的 `non-template pointer collections`，例如 `CObList` 或 `CPtrArray`。你的新類別將繼承基礎類別的所有性質。

Template-Based Classes 的用法 (注意：需含入 `afxtempl.h`，如 p.903 `stdafx.h`)

簡單型 `template-based classes` 使用時需要指定參數：

- `CArray<TYPE, ARG_TYPE>`
- `CList<TYPE, ARG_TYPE>`
- `CMap<KEY, ARG_KEY, VALUE, ARG_VALUE>`

其中 `TYPE` 用來指定你希望收集的物件的型態，它們可以是：

- C++ 基礎型別，如 `int`、`char`、`long`、`float` 等等。
- C++ 結構或類別。

ARG_TYPE 則用來指定函式的參數型態。舉個例，下面程式碼表示我們需要一個 *int* 陣列，陣列成員函式（例如 *Add*）的參數是 *int*：

```
CArray<int, int> m_intArray;
m_intArray.Add(15);
```

再舉一例，下面程式碼表示我們需要一個由 *int* 組成的串列，串列成員函式（例如 *AddTail*）的參數是 *int*：

```
CList<int, int> m_intList;
m_intList.AddTail(36);
m_intList.RemoveAll();
```

再舉一例，下面程式碼表示我們需要一個由 *CPoint* 組成的陣列，陣列成員函式（例如 *Add*）的參數是 *CPoint*：

```
CArray<CPoint, CPoint> m_pointArray;
CPoint point(18, 64);
m_pointArray.Add(point);
```

「型態指標」型的 *template-based classes* 使用時亦需指定參數：

- *CTypedPtrArray<BASE_CLASS, TYPE>*
- *CTypedPtrList<BASE_CLASS, TYPE>*
- *CTypedPtrMap<BASE_CLASS, KEY, VALUE>*

其中 *TYPE* 用來指定你希望收集的物件的型態，它們可以是：

- C++ 基礎型別，如 *int*、*char*、*long*、*float* 等等。
- C++ 結構或類別。

BASE_CLASS 則用來指定基礎類別，它可以是任何用來收集指標的 *non-template collection classes*，例如 *CObList* 或 *CObArray* 或 *CPtrList* 或 *CPtrArray* 等等。舉個例子，下面程式碼表示我們需要一個衍生自 *CObList* 的類別，用來管理一個串列，而串列組成份子為 *CStroke**：

```
CTypedPtrList<CObList, CStroke*> m_strokeList;
CStroke* pStrokeItem = new CStroke(20);
m_strokeList.AddTail(pStrokeItem);
```

CScrubbleDoc 的修改

了解了 Collection Classes 中各類別的特性以及所謂 `template/nontemplate` 版本之後，以本例之情況而言，很顯然：

- 不定量的線條數可以利用串列 (linked list) 來表示，那麼 MFC 的 *CObList* 恰可用來表現這樣的串列。*CObList* 規定其每個元素必須是一個「*CObject* 衍生類別」的物件實體，好啊，沒問題，我們就設計一個名為 *CStroke* 的類別，衍生自 *CObject*，代表一條線條。為了 `type-safe`，我們選擇 `template` 版本，所以設計出這樣的 Document：

```
class CScrubbleDoc : public CDocument
{
...
public:
    CTypedPtrList<CObList, CStroke*> m_strokeList;
...
}
```

- 線條由筆寬和座標點構成，所以 *CStroke* 應該有 *m_nPenWidth* 成員變數，但一長串的座標點以什麼來管理好呢？陣列是個不錯的選擇，至於陣列內要放什麼型態的資料，我們不妨先著一鞭，想想這些座標是怎麼獲得的。這些座標顯然是滑鼠左鍵按下時進入程式之中，也就是利用 *OnLButtonDown* 函式的參數 *CPoint*。*CPoint* 符合前一節所說的陣列元素型態條件，所以 *CStroke* 的成員變數可以這麼設計：

```
class CStroke : public CObject
{
...
protected:
    UINT m_nPenWidth;
public:
    CArray<CPoint, CPoint> m_pointArray;
...
}
```

至於 *CPoint* 實際內容是什麼，就甬管了吧。

事實上 *CPoint* 是一個由兩個 *long* 組成的結構，兩個 *long* 各代表 *x* 和 *y* 座標。

CScribble Step1 Document : (為了說明方便, 以 CObList 代替實際使用的 CTypedPtrList)

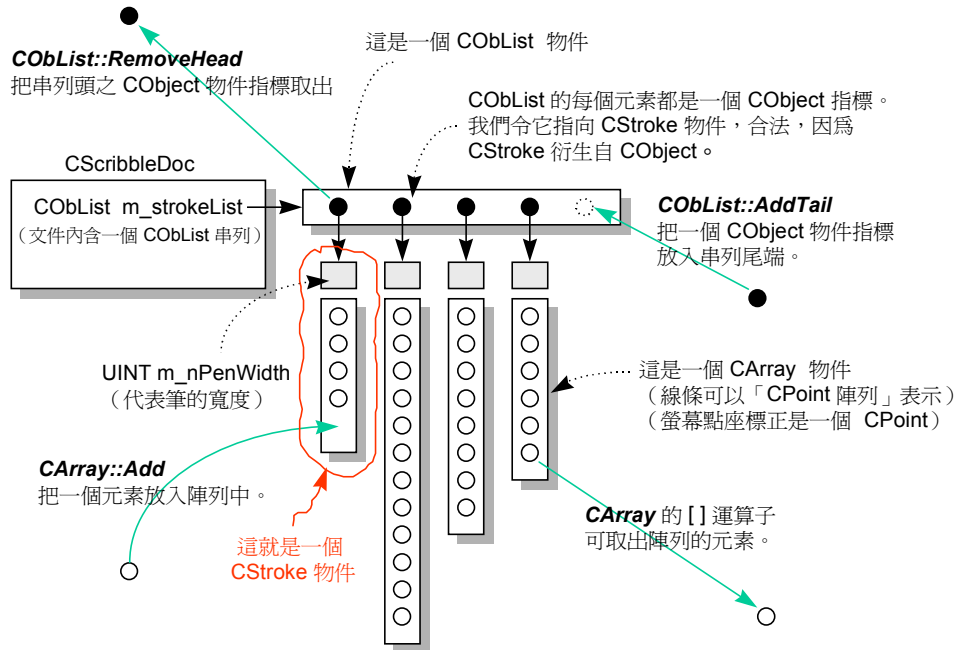


圖 8-3a Scribble Step1 的文件由線條構成，線條又由點陣列構成。

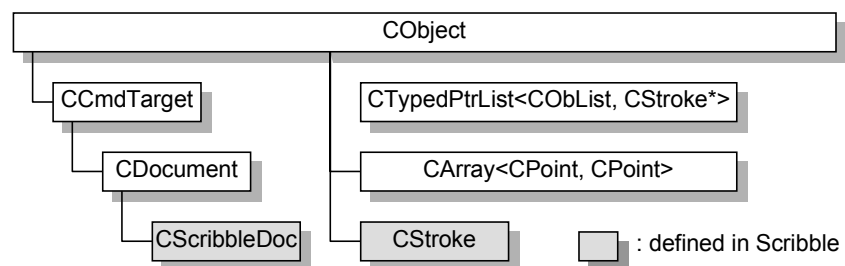


圖 8-3b Scribble Step1 文件所使用的類別

CScribbleDoc 內嵌一個 *CObList* 物件，*CObList* 串列中的每個元素都是一個 *CStroke* 物件指標，而 *CStroke* 又內嵌一個 *CArray* 物件。下面是 Step1 程式的 Document 設計。

SCRIBBLEDOC.H (陰影表示與 Step0 的差異)

```
#0001 ///////////////////////////////////////////////////
#0002 // class CStroke
#0003 //
#0004 // A stroke is a series of connected points in the scribble drawing.
#0005 // A scribble document may have multiple strokes.
#0006
#0007 class CStroke : public CObject
#0008 {
#0009 public:
#0010     CStroke(UINT nPenWidth);
#0011
#0012 protected:
#0013     CStroke();
#0014     DECLARE_SERIAL(CStroke)
#0015
#0016 // Attributes
#0017 protected:
#0018     UINT    m_nPenWidth;    // one pen width applies to entire stroke
#0019 public:
#0020     CArray<CPoint,CPoint> m_pointArray;    // series of connected
points
#0021
#0022 // Operations
#0023 public:
#0024     BOOL DrawStroke(CDC* pDC);
#0025
#0026 public:
#0027     virtual void Serialize(CArchive& ar);
#0028 };
#0029
#0030 ///////////////////////////////////////////////////
#0031
#0032 class CScribbleDoc : public CDocument
#0033 {
#0034 protected: // create from serialization only
#0035     CScribbleDoc();
#0036     DECLARE_DYNCREATE(CScribbleDoc)
#0037
#0038 // Attributes
#0039 protected:
```

```
#0040         // The document keeps track of the current pen width on
#0041         // behalf of all views. We'd like the user interface of
#0042         // Scribble to be such that if the user chooses the Draw
#0043         // Thick Line command, it will apply to all views, not just
#0044         // the view that currently has the focus.
#0045
#0046         UINT   m_nPenWidth;    // current user-selected pen width
#0047         CPen   m_penCur;      // pen created according to
#0048                                 // user-selected pen style (width)
#0049     public:
#0050         CTypedPtrList<CObList,CStroke*>    m_strokeList;
#0051         CPen*      GetCurrentPen() { return &m_penCur; }
#0052
#0053     // Operations
#0054     public:
#0055         CStroke* NewStroke();
#0056
#0057     // Overrides
#0058         // ClassWizard generated virtual function overrides
#0059         //{{AFX_VIRTUAL(CScribbleDoc)
#0060     public:
#0061         virtual BOOL OnNewDocument();
#0062         virtual void Serialize(CArchive& ar);
#0063         virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
#0064         virtual void DeleteContents();
#0065         //}}AFX_VIRTUAL
#0066
#0067     // Implementation
#0068     public:
#0069         virtual ~CScribbleDoc();
#0070     #ifdef _DEBUG
#0071         virtual void AssertValid() const;
#0072         virtual void Dump(CDumpContext& dc) const;
#0073     #endif
#0074
#0075     protected:
#0076         void InitDocument();
#0077
#0078     // Generated message map functions
#0079     protected:
#0080         //{{AFX_MSG(CScribbleDoc)
#0081         // NOTE - the ClassWizard will add and remove member functions here.
#0082         //      DO NOT EDIT what you see in these blocks of generated code !
#0083         //}}AFX_MSG
#0084         DECLARE_MESSAGE_MAP()
#0085     };
```

如果你把本書第一版（使用 VC++ 4.0）的 Scribble step1 原封不動地在 VC++ 4.2 或

VC++ 5.0 中編譯，你會獲得好幾個編譯錯誤。問題出在 SCRIBBLEDOC.H 檔案：

```
// forward declaration of data structure class
class CStroke;

class CScribbleDoc : public CDocument
{
    ...
};

class CStroke : public CObject
{
    ...
};
```

並不是程式設計上有什麼錯誤，你只要把 *CStroke* 的宣告由 *CScribbleDoc* 之後搬移到 *CScribbleDoc* 之前即可。由此觀之，VC++ 4.2 和 VC++ 5.0 的編譯器似乎不支援 forward declaration。真是沒道理！

SCRIBBLEDOC.CPP（陰影表示與 Step0 的差異）

```
#0001 #include "stdafx.h"
#0002 #include "Scribble.h"
#0003
#0004 #include "ScribbleDoc.h"
#0005
#0006 #ifdef _DEBUG
#0007 #define new DEBUG_NEW
#0008 #undef THIS_FILE
#0009 static char THIS_FILE[] = __FILE__;
#0010 #endif
#0011
#0012 //////////////////////////////////////////////////
#0013 // CScribbleDoc
#0014
#0015 IMPLEMENT_DYNCREATE(CScribbleDoc, CDocument)
#0016
#0017 BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
#0018     //{AFX_MSG_MAP(CScribbleDoc)
#0019         // NOTE - the ClassWizard will add and remove mapping macros here.
#0020         // DO NOT EDIT what you see in these blocks of generated code!
#0021     //}AFX_MSG_MAP
#0022 END_MESSAGE_MAP()
#0023
#0024 //////////////////////////////////////////////////
```

```

#0025 // CScribbleDoc construction/destruction
#0026
#0027 CScribbleDoc::CScribbleDoc()
#0028 {
#0029     // TODO: add one-time construction code here
#0030
#0031 }
#0032
#0033 CScribbleDoc::~CScribbleDoc()
#0034 {
#0035 }
#0036
#0037 BOOL CScribbleDoc::OnNewDocument()
#0038 {
#0039     if (!CDocument::OnNewDocument())
#0040         return FALSE;
#0041     InitDocument();
#0042     return TRUE;
#0043 }
#0044
#0045 //////////////////////////////////////////////////
#0046 // CScribbleDoc serialization
#0047
#0048 void CScribbleDoc::Serialize(CArchive& ar)
#0049 {
#0050     if (ar.IsStoring())
#0051     {
#0052     }
#0053     else
#0054     {
#0055     }
#0056     m_strokeList.Serialize(ar);
#0057 }
#0058
#0059 //////////////////////////////////////////////////
#0060 // CScribbleDoc diagnostics
#0061
#0062 #ifdef _DEBUG
#0063 void CScribbleDoc::AssertValid() const
#0064 {
#0065     CDocument::AssertValid();
#0066 }
#0067
#0068 void CScribbleDoc::Dump(CDumpContext& dc) const
#0069 {
#0070     CDocument::Dump(dc);
#0071 }
#0072 #endif // _DEBUG

```

```

#0073
#0074 //////////////////////////////////////////////////
#0075 // CScribbleDoc commands
#0076
#0077 BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
#0078 {
#0079     if (!CDocument::OnOpenDocument(lpszPathName))
#0080         return FALSE;
#0081     InitDocument();
#0082     return TRUE;
#0083 }
#0084
#0085 void CScribbleDoc::DeleteContents()
#0086 {
#0087     while (!m_strokeList.IsEmpty())
#0088     {
#0089         delete m_strokeList.RemoveHead();
#0090     }
#0091     CDocument::DeleteContents();
#0092 }
#0093
#0094 void CScribbleDoc::InitDocument()
#0095 {
#0096     m_nPenWidth = 2; // default 2 pixel pen width
#0097     // solid, black pen
#0098     m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0));
#0099 }
#0100
#0101 CStroke* CScribbleDoc::NewStroke()
#0102 {
#0103     CStroke* pStrokeItem = new CStroke(m_nPenWidth);
#0104     m_strokeList.AddTail(pStrokeItem);
#0105     SetModifiedFlag(); // Mark the document as having been modified, for
#0106                       // purposes of confirming File Close.
#0107     return pStrokeItem;
#0108 }
#0109
#0110
#0111
#0112
#0113 //////////////////////////////////////////////////
#0114 // CStroke
#0115
#0116 IMPLEMENT_SERIAL(CStroke, CObject, 1)
#0117 CStroke::CStroke()
#0118 {
#0119     // This empty constructor should be used by serialization only
#0120 }

```

```

#0121
#0122 CStroke::CStroke(UINT nPenWidth)
#0123 {
#0124     m_nPenWidth = nPenWidth;
#0125 }
#0126
#0127 void CStroke::Serialize(CArchive& ar)
#0128 {
#0129     if (ar.IsStoring())
#0130     {
#0131         ar << (WORD)m_nPenWidth;
#0132         m_pointArray.Serialize(ar);
#0133     }
#0134     else
#0135     {
#0136         WORD w;
#0137         ar >> w;
#0138         m_nPenWidth = w;
#0139         m_pointArray.Serialize(ar);
#0140     }
#0141 }
#0142
#0143 BOOL CStroke::DrawStroke(CDC* pDC)
#0144 {
#0145     CPen penStroke;
#0146     if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
#0147         return FALSE;
#0148     CPen* pOldPen = pDC->SelectObject(&penStroke);
#0149     pDC->MoveTo(m_pointArray[0]);
#0150     for (int i=1; i < m_pointArray.GetSize(); i++)
#0151     {
#0152         pDC->LineTo(m_pointArray[i]);
#0153     }
#0154
#0155     pDC->SelectObject(pOldPen);
#0156     return TRUE;
#0157 }

```

爲了了解線條的產生經歷了哪些成員函式，使用了哪些成員變數，我把圖 8-3 所顯示的各類別的成員整理於下。讓我們以 top-down 的方式看看文件組成份子的運作。

文件：－連串的線條

Scribble 文件本身由許多線條組合而成。而你知道，以串列（linked list）表示不定個數的東西最是理想了。MFC 有沒有現成的「串列」類別呢？有，*COBList* 就是。它的每一個元素都必須是 *CObject**。回想一下我在第二章介紹的「職員」例子：

我們有一個職員串列，串列的每一個元素的型態是「指向最基礎類別之指標」。如果基礎類別有一個「計薪」方法（虛擬函式），那麼我們就可以一個「一般性」的迴路把串列巡訪一遍；巡到不同的職員型別，就呼叫該型別的計薪方法。

如今我們選用 *COBList*，情況不就和上述職員例子如出一轍嗎？*CObject* 的許多好性質，如 *Serialization*、*RTTI*、*Dynamic Creation*，可以非常簡便地應用到我們極為「一般性」的操作上。這一點在稍後的 *Serialization* 動作上更表現得淋漓盡致。

C_ScribbleDoc 的成員變數

- *m_strokeList*：這是一個 *COBList* 物件，代表一個串列。串列中的元素是什麼型態？答案是 *CObject**。但實際運作時，我們可以把基礎類別之指標指向衍生類別之物件（還記得第 2 章我介紹虛擬函式時特別強調的吧）。現在我們想讓這個串列成為「由 *CStroke* 物件構成的串列」，因此顯然 *CStroke* 必須衍生自 *CObject* 才行，而事實上它的確是。
- *m_nPenWidth*：每一線條都有自己的筆寬，而目前使用的筆寬記錄於此。
- *m_penCur*：這是一個 *CPen* 物件。程式依據上述的筆寬，配置一支筆，準備用來畫線條。筆寬可以指定，但那是第 10 章的事。注意，筆寬的設定對象是線條，不是單一的點，也不是一整張圖。

COBList

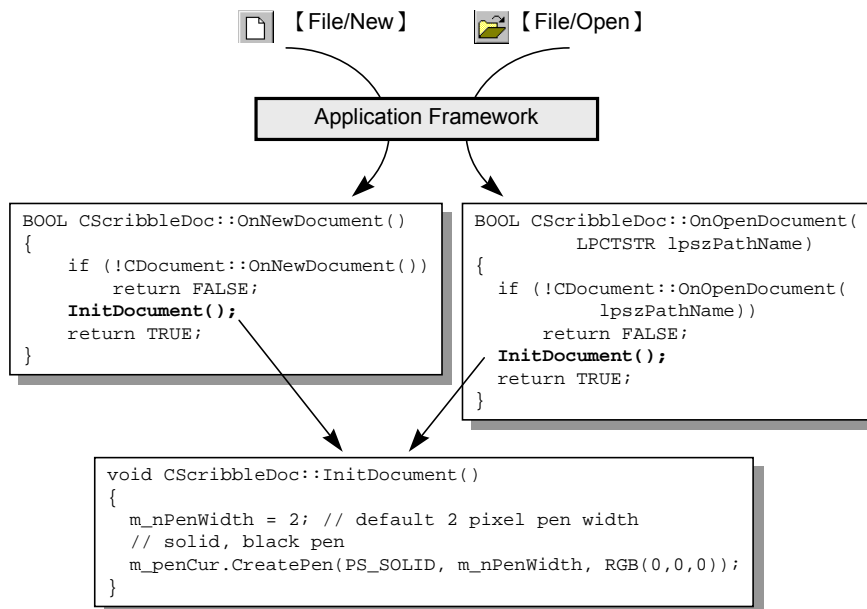
這是 MFC 的內建類別，提供我們串列服務。串列的每個元素都必須是 *CObject**。本處將用到四個成員函式：

- *AddTail*：在串列尾端加上一個元素。
- *IsEmpty*：串列是否為空？
- *RemoveHead*：把串列整個拿掉。
- *Serialize*：檔案讀寫。這是個空的虛擬函式，改寫它正是我們稍後要做的努力。

CScrubbleDoc 的 成員 函 式

- *OnNewDocument*、*OnOpenDocument*、*InitDocument*。產生 Document 的時機有二，一是使用者選按【File/New】，一是使用者選按【File/Open】。當這兩種情況發生，Application Framework 會分別呼叫 Document 類別的 *OnNewDocument* 和 *OnOpenDocument*。為了應用程式本身的特性考量（例如本例畫筆的產生以及筆寬的設定），我們應該改寫這些虛擬函式。

本例把文件初始化工作（畫筆以及筆寬的設定）分割出來，獨立於 *InitDocument* 函式中，因此上述的 *OnNew*_ 和 *OnOpen*_ 兩函式都呼叫 *InitDocument*。



- *NewStroke*。這個函式將產生一個新的 *CStroke* 物件，並把它加到串列之中。很顯然這應該在滑鼠左鍵按下時發生（我們將在 *CScribbleView* 之中處理滑鼠訊息）。本函式動作如下：

```
CStroke* CScribbleDoc::NewStroke()
{
    CStroke* pStrokeItem = new CStroke(m_nPenWidth);
    m_strokeList.AddTail(pStrokeItem);
    SetModifiedFlag(); // Mark the document as having been modified, for
                       // purposes of confirming File Close.
    return pStrokeItem;
}
```

這就產生了一個新線條，設定了線條寬度，並將新線條加入串列尾端。

- *DeleteContent*。利用 *COBList::RemoveHead* 把串列的最前端元素拿掉。

```
void CScribbleDoc::DeleteContents()
{
    while (!m_strokeList.IsEmpty())
    {
        delete m_strokeList.RemoveHead();
    }
    CDocument::DeleteContents();
}
```

- *Serialize*。這個函式負責檔案讀寫。由於文件掌管線條串列，線條串列又掌管各線條，我們可以善用這些階層關係：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
    }
    else
    {
    }
    m_strokeList.Serialize(ar);
}
```

我們有充份的理由認為，*COBList::Serialize* 的內部動作，一定是以一個迴路巡訪所有的元素，一一呼叫各元素（是個指標）所指向的物件的 *Serialize* 函式。就好像第2章「職員」串列中的計薪方法一樣。

馬上我們就會看到，*Serialize* 如何層層下達。那是很深入的探討，你要先有心理準備。

線條與座標點

Scribble 的文件資料由線條構成，線條又由點陣列構成，點又由 (x,y) 座標構成。我們將設計 *CStroke* 用以描述線條，並直接採用 MFC 的 *CArray* 描述點陣列。

CStroke 的成員變數

- *m_pointArray*：這是一個 *CArray* 物件，用以記錄一系列的 *CPoint* 物件，這些 *CPoint* 物件由滑鼠座標轉化而來。
- *m_nPenWidth*：一個整數，代表線條寬度。雖然 Scribble Step1 的線條寬度是固定的，但第 10 章允許改變寬度。

CArray<CPoint, CPoint>

CArray 是 MFC 內建類別，提供陣列的各種服務。本例利用其 *template* 性質，指定陣列內容為 *CPoint*。本例將用到 *CArray* 的兩個成員函式和一個運算子：

- *GetSize*：取得陣列中的元素個數。
- *Add*：在陣列尾端增加一個元素。必要時擴大陣列的大小。這個動作會在滑鼠左鍵按下後被持續呼叫，請看 *ScribbleView::OnLButtonDown*。
- *operator[]*：以指定之索引值取得或設定陣列元素內容。

它們的詳細規格請參考 *MFC Class Library Reference*。

CStroke 的成員函式

- *DrawStroke*：繪圖原本是 *View* 的責任，為什麼卻在 *CStroke* 中有一個 *DrawStroke*？因為線條的內容只有 *CStroke* 自己知道，當然由 *CStroke* 的成員函式把它畫出來最是理想。這麼一來，*View* 就可以一一呼叫線條自己的繪圖函式，很輕鬆。

此函式把點座標從陣列之中一個一個取出，畫到視窗上，所以你會看到整個原始繪圖過程的重現，而不是一整張圖啪一下子出現。想當然耳，這個函式內會有 *CreatePen*、*SelectObject*、*MoveTo*、*LineTo* 等 GDI 動作，以及從陣列中取座標點的動作。取點動作直接利用 *CArray* 的 *operator[]* 運算子即可辦到：

```

BOOL CStroke::DrawStroke(CDC* pDC)
{
    CPen penStroke;
    if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
        return FALSE;
    CPen* pOldPen = pDC->SelectObject(&penStroke);
    pDC->MoveTo(m_pointArray[0]);
    for (int i=1; i < m_pointArray.GetSize(); i++)
    {
        pDC->LineTo(m_pointArray[i]);
    }
    pDC->SelectObject(pOldPen);
    return TRUE;
}

```

- *Serialize*：讓我們這麼想像寫檔動作：使用者下命令給程式，程式發命令給文件，文件發命令給線條，線條發命令給點陣列，點陣列於是把一個個的座標點寫入磁碟中。請注意，每一線條除了擁有點陣列之外，還有一個筆劃寬度，讀寫檔時可不要忘了這份資料。

```

void CStroke::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    { // 寫檔
        ar << (WORD)m_nPenWidth;
        m_pointArray.Serialize(ar);
    }
    else
    { // 讀檔
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}

```

肯定你會產生兩個疑問：

1. 為什麼點陣列的讀檔寫檔動作完全一樣，都是 *Serialize(ar)* 呢？
2. 線條串列的 *Serialize* 函式如何能夠把命令交派到線條的 *Serialize* 函式呢？

第一個問題的答案很簡單，第二個問題的答案很複雜。稍後我對此有所解釋。

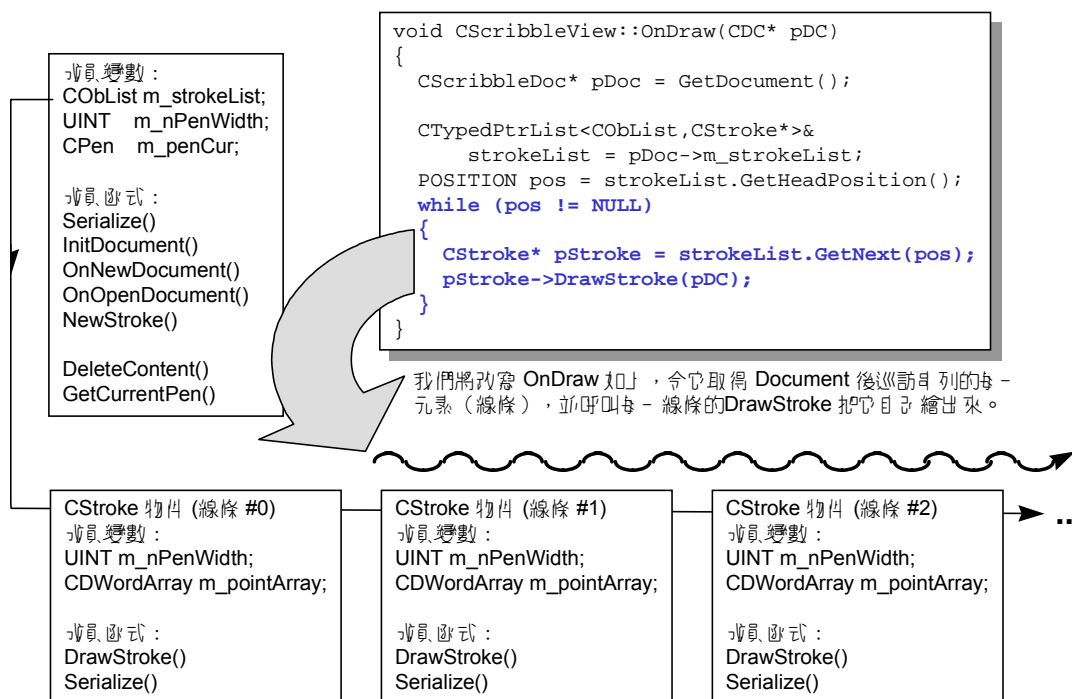


圖 8-4 Scribble 的 Document/View 成員鳥瞰

圖 8-4 把 Scribble Step1 的 Document/View 重要成員集中在一起顯示，幫助你做大局觀。請注意，雖然本圖把「成員函式」和「成員變數」畫在每一個物件之中，但你知道，事實上 C++ 類別的成員函式另放在物件記憶體以外，並不是每一個物件都有一份函式碼。只有 non-static 成員變數，才會每個物件各有一份。這個觀念我曾在第 2 章強調過。

Scribble Step1 的 View：資料重繪與編輯

View 有兩個最重要的任務，一是負責資料的顯示，另一是負責資料的編輯（透過鍵盤或滑鼠）。本例的 *CScribbleView* 包括以下特質：

- 解讀 *CScribbleDoc* 中的資料，包括筆寬以及一系列的 *CPoint* 物件，畫在 View 視窗上。
- 允許使用者以滑鼠左鍵充當畫筆在 View 視窗內塗抹，換句話說 *CScribbleView* 必須接受並處理 *WM_LBUTTONDOWN*、*WM_MOUSEMOVE*、*WM_LBUTTONUP* 三個訊息。

當 Framework 收到 *WM_PAINT*，表示畫面需要重繪，它會呼叫 *OnDraw*（註），由 *OnDraw* 執行真正的繪圖動作。什麼時候會產生重繪訊息 *WM_PAINT* 呢？當使用者改變視窗大小，或是將視窗圖示化之後再恢復原狀，或是來自程式（自己或別人）刻意的製造。除了在必須重繪時重繪之外，做為一個繪圖軟體，Scribble 還必須「即時」反應滑鼠左鍵在視窗上移動的軌跡，不能等到 *WM_PAINT* 產生了才有所反應。所以，我們必須在 *OnMouseMove* 中也做繪圖動作，那是針對一個點一個點的繪圖，而 *OnDraw* 是大規模的全部重繪。

註：其實 Framework 是先呼叫 *OnPaint*，*OnPaint* 再呼叫 *OnDraw*。關於 *OnPaint*，第 12 章談到印表機時再說。

繪圖前當然必須獲得資料內容，呼叫 *GetDocument* 即可獲得，它傳回一個 *CScribbleDoc* 物件指標。別忘了 View 和 Document 以及 Frame 視窗早在註冊 Document Template 時就建立彼此間的關聯了。所以，從 *CScribbleView* 發出的 *GetDocument* 函式當然能夠獲得 *CScribbleDoc* 的物件指標。View 可以藉此指標取得 Document 的資料，然後顯示。

CScribbleView 的修改

以下是 Step1 程式的 View 的設計。其中有滑鼠介面，也有資料顯示功能 *OnDraw*。

SCRIBBLEVIEW.H (陰影表示與 Step0 的差異)

```
#0001 class CScribbleView : public CView
#0002 {
#0003     protected: // create from serialization only
#0004         CScribbleView();
#0005         DECLARE_DYNCREATE(CScribbleView)
#0006
#0007     // Attributes
#0008     public:
#0009         CScribbleDoc* GetDocument();
#0010
#0011     protected:
#0012         CStroke* m_pStrokeCur; // the stroke in progress
#0013         CPoint m_ptPrev; // the last mouse pt in the stroke in progress
#0014
#0015     // Operations
#0016     public:
#0017
#0018     // Overrides
#0019         // ClassWizard generated virtual function overrides
#0020         //{AFX_VIRTUAL(CScribbleView)
#0021         public:
#0022         virtual void OnDraw(CDC* pDC); // overridden to draw this view
#0023         virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
#0024         protected:
#0025         virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
#0026         virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
#0027         virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
#0028         //}}AFX_VIRTUAL
#0029
#0030     // Implementation
#0031     public:
#0032         virtual ~CScribbleView();
#0033     #ifdef _DEBUG
#0034         virtual void AssertValid() const;
#0035         virtual void Dump(CDumpContext& dc) const;
#0036     #endif
#0037
#0038     protected:
#0039
```

```

#0040 // Generated message map functions
#0041 protected:
#0042     //{AFX_MSG(CScribbleView)
#0043     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
#0044     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
#0045     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
#0046     //}}AFX_MSG
#0047     DECLARE_MESSAGE_MAP()
#0048 };
#0049
#0050 #ifndef _DEBUG // debug version in ScribVw.cpp
#0051 inline CScribbleDoc* CScribbleView::GetDocument()
#0052     { return (CScribbleDoc*)m_pDocument; }
#0053 #endif

```

SCRIBBLEVIEW.CPP (陰影表示與 Step0 的差異)

```

#0001 #include "stdafx.h"
#0002 #include "Scribble.h"
#0003
#0004 #include "ScribbleDoc.h"
#0005 #include "ScribbleView.h"
#0006
#0007 #ifdef _DEBUG
#0008 #define new DEBUG_NEW
#0009 #undef THIS_FILE
#0010 static char THIS_FILE[] = __FILE__;
#0011 #endif
#0012
#0013 //////////////////////////////////////
#0014 // CScribbleView
#0015
#0016 IMPLEMENT_DYNCREATE(CScribbleView, CView)
#0017
#0018 BEGIN_MESSAGE_MAP(CScribbleView, CView)
#0019     //{AFX_MSG_MAP(CScribbleView)
#0020     ON_WM_LBUTTONDOWN()
#0021     ON_WM_LBUTTONUP()
#0022     ON_WM_MOUSEMOVE()
#0023     //}}AFX_MSG_MAP
#0024     // Standard printing commands
#0025     ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
#0026     ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
#0027     ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
#0028 END_MESSAGE_MAP()

```



```

#0029
#0030 //////////////////////////////////////////////////
#0031 // CScribbleView construction/destruction
#0032
#0033 CScribbleView::CScribbleView()
#0034 {
#0035     // TODO: add construction code here
#0036
#0037 }
#0038
#0039 CScribbleView::~CScribbleView()
#0040 {
#0041 }
#0042
#0043 BOOL CScribbleView::PreCreateWindow(CREATESTRUCT& cs)
#0044 {
#0045     // TODO: Modify the Window class or styles here by modifying
#0046     // the CREATESTRUCT cs
#0047
#0048     return CView::PreCreateWindow(cs);
#0049 }
#0050
#0051 //////////////////////////////////////////////////
#0052 // CScribbleView drawing
#0053
#0054 void CScribbleView::OnDraw(CDC* pDC)
#0055 {
#0056     CScribbleDoc* pDoc = GetDocument();
#0057     ASSERT_VALID(pDoc);
#0058
#0059     // The view delegates the drawing of individual strokes to
#0060     // CStroke::DrawStroke().
#0061     CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
#0062     POSITION pos = strokeList.GetHeadPosition();
#0063     while (pos != NULL)
#0064     {
#0065         CStroke* pStroke = strokeList.GetNext(pos);
#0066         pStroke->DrawStroke(pDC);
#0067     }
#0068 }
#0069
#0070 //////////////////////////////////////////////////
#0071 // CScribbleView printing
#0072
#0073 BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
#0074 {
#0075     // default preparation
#0076     return DoPreparePrinting(pInfo);

```

```

#0077 }
#0078
#0079 void CScribbleView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
#0080 {
#0081     // TODO: add extra initialization before printing
#0082 }
#0083
#0084 void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
#0085 {
#0086     // TODO: add cleanup after printing
#0087 }
#0088
#0089 //////////////////////////////////////////////////
#0090 // CScribbleView diagnostics
#0091
#0092 #ifdef _DEBUG
#0093 void CScribbleView::AssertValid() const
#0094 {
#0095     CView::AssertValid();
#0096 }
#0097
#0098 void CScribbleView::Dump(CDumpContext& dc) const
#0099 {
#0100     CView::Dump(dc);
#0101 }
#0102
#0103 CScribbleDoc* CScribbleView::GetDocument() // non-debug version is inline
#0104 {
#0105     ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CScribbleDoc)));
#0106     return (CScribbleDoc*)m_pDocument;
#0107 }
#0108 #endif // _DEBUG
#0109
#0110 //////////////////////////////////////////////////
#0111 // CScribbleView message handlers
#0112
#0113 void CScribbleView::OnLButtonDown(UINT, CPoint point)
#0114 {
#0115     // Pressing the mouse button in the view window starts a new stroke
#0116
#0117     m_pStrokeCur = GetDocument()->NewStroke();
#0118     // Add first point to the new stroke
#0119     m_pStrokeCur->m_pointArray.Add(point);
#0120
#0121     SetCapture(); // Capture the mouse until button up.
#0122     m_ptPrev = point; // Serves as the MoveTo() anchor point
#0123                     // for the LineTo() the next point,
#0124                     // as the user drags the mouse.

```

```
#0125
#0126     return;
#0127 }
#0128
#0129 void CScribbleView::OnLButtonUp(UINT, CPoint point)
#0130 {
#0131     // Mouse button up is interesting in the Scribble application
#0132     // only if the user is currently drawing a new stroke by dragging
#0133     // the captured mouse.
#0134
#0135     if (GetCapture() != this)
#0136         return; // If this window (view) didn't capture the mouse,
#0137                 // then the user isn't drawing in this window.
#0138
#0139     CScribbleDoc* pDoc = GetDocument();
#0140
#0141     CClientDC dc(this);
#0142
#0143     CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
#0144     dc.MoveTo(m_ptPrev);
#0145     dc.LineTo(point);
#0146     dc.SelectObject(pOldPen);
#0147     m_pStrokeCur->m_pointArray.Add(point);
#0148
#0149     ReleaseCapture(); // Release the mouse capture established at
#0150                     // the beginning of the mouse drag.
#0151     return;
#0152 }
#0153
#0154 void CScribbleView::OnMouseMove(UINT, CPoint point)
#0155 {
#0156     // Mouse movement is interesting in the Scribble application
#0157     // only if the user is currently drawing a new stroke by dragging
#0158     // the captured mouse.
#0159
#0160     if (GetCapture() != this)
#0161         return; // If this window (view) didn't capture the mouse,
#0162                 // then the user isn't drawing in this window.
#0163
#0164     CClientDC dc(this);
#0165     m_pStrokeCur->m_pointArray.Add(point);
#0166
#0167     // Draw a line from the previous detected point in the mouse
#0168     // drag to the current point.
#0169     CPen* pOldPen = dc.SelectObject(GetDocument()->GetCurrentPen());
#0170     dc.MoveTo(m_ptPrev);
#0171     dc.LineTo(point);
#0172     dc.SelectObject(pOldPen);
```

```
#0173         m_ptPrev = point;
#0174         return;
#0175     }
```

View 的重繪動作：GetDocument 和 OnDraw

以下是 *CScribbleView* 中與重繪動作有關的成員變數和成員函式。

CScribbleView 的成員變數

- *m_pStrokeCur*：一個指標，指向目前正在工作的線條。
- *m_ptPrev*：線條中的前一個工作點。我們將在這個點與目前滑鼠按下的點之間畫一條直線。雖說理想情況下滑鼠軌跡的每一個點都應該被記錄下來，但如果滑鼠移動太快來不及記錄，只好在兩點之間拉直線。

CScribbleView 的成員函式

- *OnDraw*：這是一個虛擬函式，負責將 *Document* 的資料顯示出來。改寫它是程式員最大的責任之一。
- *GetDocument*：AppWizard 為我們做出這樣的碼，以 *inline* 方式定義於表頭檔：


```
inline CScribbleDoc* CScribbleView::GetDocument()
{ return (CScribbleDoc*)m_pDocument; }
```

其中 *m_pDocument* 是 *CView* 的成員變數。我們可以推測，當程式設定好 *Document Template* 之後，每次 Framework 動態產生 *View* 物件，其內的 *m_pDocument* 已經被 Framework 設定指向對應之 *Document* 了。

View 物件何時被動態產生？答案是當使用者選按【File/Open】或【File/New】。每當產生一個 *Document*，就會產生一組 *Document/View/Frame*「三口組」。

- *OnPreparePrinting*，*OnBeginPrinting*，*OnEndPrinting*：這三個 *CView* 虛擬函式將用來改善印表行為。AppWizard 只是先幫我們做出空函式。第 12 章才會用到它們。

我們來看看 *CView* 之中居最重要地位的 *OnDraw*，面對 *Scribble Document* 的資料結構，將如何進行繪圖動作。爲了獲得資料，*OnDraw* 一開始先以 *GetDocument* 取得 *Document* 物件指標；然後以 *while* 迴路一一取得各線條，再呼叫 *CStroke::DrawStroke* 繪圖。想像中繪圖函式應該放在 *View* 類別之內(繪圖不正是 *View* 的責任嗎)，但是 *DrawStroke* 卻否！原因是把線條的資料和繪圖動作一併放在 *CStroke* 中是最好的包裝方式。

```
void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // The view delegates the drawing of individual strokes to
    // CStroke::DrawStroke().
    CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        CStroke* pStroke = strokeList.GetNext(pos);
        pStroke->DrawStroke(pDC);
    }
}
```

其中用到兩個 *CObList* 成員函式：

- *GetNext*：取得下一個元素。
- *GetHeadPosition*：傳回串列之第一個元素的「位置」。傳回來的「位置」是一個型態爲 *POSITION* 的數值，這個數值可以被使用於 *CObList* 的其他成員函式中，例如 *GetAt* 或 *SetAt*。你可以把「位置」想像是串列中用以標示某個節點 (node) 的指標。當然，它並不真正是指標。

View 與使用者的互動 (滑鼠訊息處理實例)

爲了實現「以鼠代筆」的功能，*CScribbleView* 必須接受並處理三個訊息：

```
BEGIN_MESSAGE_MAP(CScribbleView, CView)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    ...
END_MESSAGE_MAP()
```

三個訊息處理常式的內容總括來說就是追蹤滑鼠軌跡、在視窗上繪圖、以及呼叫 *CStroke* 成員函式以修正線條內容 --- 包括產生一個新的線條空間以及不斷把座標點加上去。三個函式的重要動作摘記於下。這些函式的骨幹及其在 **Message Map** 中的映射項目，不勞我們動手，有 **ClassWizard** 代勞。下一個小節我會介紹其操作方法。

```
void CScribbleView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 當滑鼠左鍵按下，
    // 利用 CScribbleDoc::NewStroke 產生一個新的線條空間；
    // 利用 CArray::Add 把這個點加到線條上去；
    // 呼叫 SetCapture 取得滑鼠捕捉權 (mouse capture)；
    // 把這個點記錄爲「上一點」(m_ptPrev)；
}

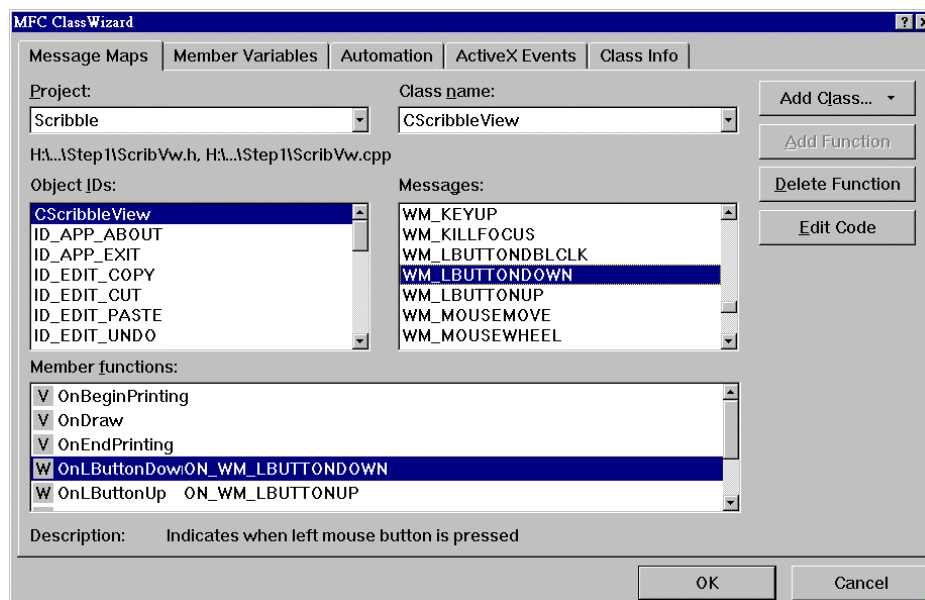
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
    // 當滑鼠左鍵按住並開始移動，
    // 利用 CArray::Add 把新座標點加到線條上；
    // 在上一點 (m_ptPrev) 和這一點之間畫直線；
    // 把這個點記錄爲「上一點」(m_ptPrev)；
}

void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
    // 當滑鼠左鍵放開，
    // 在上一點 (m_ptPrev) 和這一點之間畫直線；
    // 利用 CArray::Add 把新的點加到線條上；
    // 呼叫 ReleaseCapture() 釋放滑鼠捕捉權 (mouse capture)。
}
```

ClassWizard 的輔佐

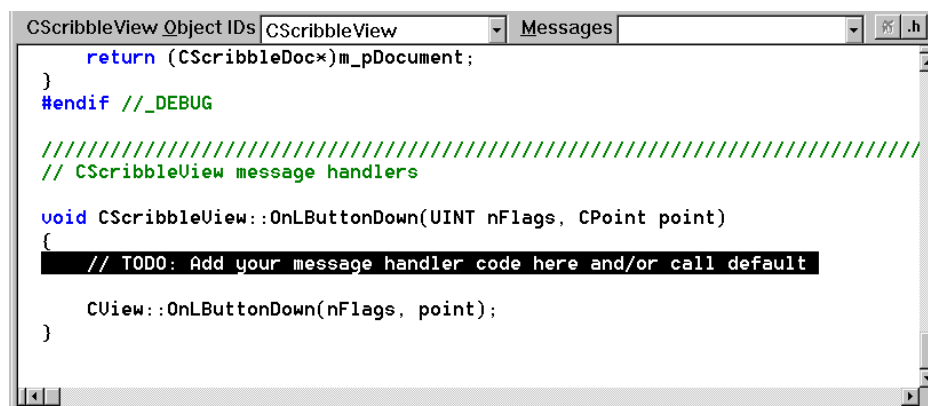
前述三個 *CScribbleView* 成員函式 (*OnLButtonDown* , *OnLButtonUp* , *OnMouseMove*) 是 Message Map 的一部份，ClassWizard 可以很方便地幫助我們完成相關的 Message Map 設定工作。

首先，選按【View/ClassWizard】啟動 ClassWizard，選擇其【Message Map】附頁：



在圖右上側的【Class Name】清單中選擇 *CScribbleView*，然後在圖左側的【Object IDs】清單中選擇 *CScribbleView*，再在圖右側的【Messages】清單中選擇 *WM_LBUTTONDOWN*，然後選按圖右的【Add Function】鈕，於是圖下側的【Member functions】清單中出現一筆新項目。

然後，選按【Edit Code】鈕，文字編輯器會跳出來，你獲得了一個 *OnLButtonDown* 函式空殼，請在這裡鍵入你的程式碼：



另兩個訊息處理常式的實作作法雷同。

Message Map 因此有什麼變化呢？ClassWizard 為我們自動加上了三筆映射項目：

```

BEGIN_MESSAGE_MAP(CScrubbleView, CView)
    //{{AFX_MSG_MAP(CScrubbleView)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

```

此外 ScribbleView 的類別宣告中也自動有了三個成員函式的宣告：

```

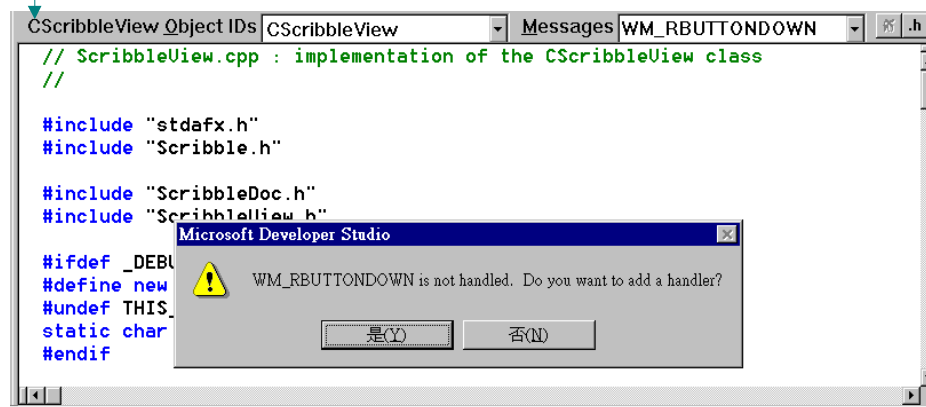
class CScrubbleView : public CView
{
...
// Generated message map functions
protected:
    //{{AFX_MSG(CScrubbleView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    //}}AFX_MSG
...
};

```


WizardBar 的輔位

WizardBar 是 Visual C++ 4.0 之後的新增工具，也就是文字編輯器上方那個有著【Object IDs】和【Messages】清單的橫桿。關於修改 Message Map 這件事，WizardBar 可以取代 ClassWizard 這個大傢伙。

首先，進入 ScribbleView.cpp（因為我們確定要在這裡加入三個滑鼠訊息處理常式），選擇 WizardBar 上的【Object IDs】為 *C ScribbleView*，再選擇【Messages】為 *WM_LBUTTONDOWN*，出現以下畫面：



回答 Yes，於是你獲得一個 *OnLButtonDown* 函式空殼，一如在 ClassWizard 中所得。請在函式空殼中輸入你的程式碼。

Serialize：物件的檔案讀寫

你可能對 Serialization 這個名詞感覺陌生，事實上它就是物件導向世界裡的 Persistence（永續生存），只是後者比較抽象一些。物件必須能夠永續生存，也就是它們必須能夠在程式結束時儲存到檔案中，並且在程式重新啟動時再恢復回來。儲存和恢復物件的過程在 MFC 之中就稱為 serialization。負責這件重要任務的，是 MFC *CObject* 類別中一個名為 *Serialize* 的虛擬函式，檔案的「讀」「寫」動作均透過它。

如果文件內容是藉著層層類別向下管理（一如本例），那麼只要每一層把自己份內的工作做好，層層交待下來就可以完成整份資料的檔案動作。

Serialization 以外的檔案讀寫動作

其實有時候我們希望在重重包裝之中返璞歸真一下，感受一些質樸的動作。在介紹 *Serialization* 的重重包裝之前，這裡給你一覽檔案實際讀寫動作的機會。

檔案 I/O 服務是任何作業系統的主要服務。Win32 提供了許多檔案相關 APIs：開檔、關檔、讀檔、寫檔、搜尋資料...。MFC 把這些操作都包裝在 *CFile* 之中。可想而知，它必然有 *Open*、*Close*、*Read*、*Write*、*Seek*... 等等成員函式。下面這段程式碼示範 *CFile* 如何讀檔：

```
char* pBuffer = new char[0x8000];
CFile file("mydoc.doc", CFile::modeRead); // 打開 mydoc.doc 檔，使用唯讀模式。
UINT nBytesRead = file.Read(pBuffer, 0x8000); // 讀取 8000h 個位元組到 pBuffer 中。
```

上述程式片段中，物件 *file* 的建構式將打開 *mydoc.doc* 檔。並且由於此物件產生於函式的堆疊之中，當函式結束，*file* 的解構式將自動關閉 *mydoc.doc* 檔。

開檔模式有許多種，都定義在 *CFile* (AFX.H) 之中：

```
enum OpenFlags {
    modeRead = 0x0000, // 唯讀
    modeWrite = 0x0001, // 唯寫
    modeReadWrite = 0x0002, // 可讀可寫
    shareCompat = 0x0000,
    shareExclusive = 0x0010, // 唯我使用
    shareDenyWrite = 0x0020,
    shareDenyRead = 0x0030,
    shareDenyNone = 0x0040,
    modeNoInherit = 0x0080,
    modeCreate = 0x1000, // 產生新檔（甚至即使已有相同名稱之檔案存在）
    modeNoTruncate = 0x2000,
    typeText = 0x4000, // typeText and typeBinary are used in
    typeBinary = (int)0x8000 // derived classes only
};
```

再舉一例，下面這段程式碼可將檔案 mydoc.doc 的所有文字轉換為小寫：

```
char* pBuffer = new char[0x1000];
CFile file("mydoc.doc", CFile::modeReadWrite);
DWORD dwBytesRemaining = file.GetLength();
UINT nBytesRead;
DWORD dwPosition;

while (dwBytesRemaining) {
    dwPosition = file.GetPosition();
    nBytesRead = file.Read(pBuffer, 0x1000);
    ::CharLowerBuff(pBuffer, nBytesRead);
    file.Seek((LONG)dwPosition, CFile::begin);
    file.Write(pBuffer, nBytesRead);
    dwBytesRemaining -= nBytesRead;
}
delete[] pBuffer;
```

檔案的操作常需配合對異常情況（exception）的處理，因為檔案的異常情況特別多：檔案找不到啦、檔案 handles 不足啦、讀寫失敗啦...。上一例加入異樣情況處理後如下：

```
char* pBuffer = new char[0x1000];

try {
    CFile file("mydoc.doc", CFile::modeReadWrite);
    DWORD dwBytesRemaining = file.GetLength();
    UINT nBytesRead;
    DWORD dwPosition;

    while (dwBytesRemaining) {
        dwPosition = file.GetPosition();
        nBytesRead = file.Read(pBuffer, 0x1000);
        ::CharLowerBuff(pBuffer, nBytesRead);
        file.Seek((LONG)dwPosition, CFile::begin);
        file.Write(pBuffer, nBytesRead);
        dwBytesRemaining -= nBytesRead;
    }
}
catch (CFileException* e) {
    if (e->cause == CFileException::fileNotFound)
        MessageBox("File not found");
    else if (e->cause == CFileException::tooManyOpenFiles)
        MessageBox("File handles not enough");
    else if (e->cause == CFileException::hardIO)
```

```

        MessageBox("Hardware error");
    else if (e->cause == CFileException::diskFull)
        MessageBox("Disk full");
    else if (e->cause == CFileException::badPath)
        MessageBox("All or part of the path is invalid");
    else
        MessageBox("Unknown file error");
    e->Delete();
}
delete[] pByffer;

```

檯面上的 Serialize 動作

讓我以 *Scribble* 為例，向你解釋檯面上的（應用程式碼中可見的）serialization 動作。根據圖 8-3 的資料結構，*Scribble* 程式的檔案讀寫動作是這麼分工的：

- Framework 呼叫 *CScribbleDoc::Serialize*，用以對付文件。
- *CScribbleDoc* 再往下呼叫 *CStroke::Serialize*，用以對付線條。
- *CStroke* 再往下呼叫 *CArray::Serialize*，用以對付點陣列。

讀也由它，寫也由它，究竟 *Serialize* 是讀還是寫？這一點不必我們操心。Framework 呼叫 *Serialize* 時會傳來一個 *CArchive* 物件（稍後我會解釋 *CArchive*），你可以想像它代表一個檔案，透過其 *IsStoring* 成員函式，即可知道究竟要讀還是寫。圖 8-5 是各層級的 *Serialize* 動作示意圖，文字說明已在圖片之中。

注意：*Scribble* 程式使用 *CArray<CPoint, CPoint>* 儲存滑鼠位置座標，而 *CArray* 是一個 *template class*，解釋起來比較複雜。所以稍後我挖給各位看的 *Serialize* 函式原始碼，採用 *CDWordArray* 的成員函式而非 *CArray* 的成員函式。Visual C++ 1.5 版的 *Scribble* 範例程式就是使用 *CDWordArray*（彼時還未有 *template class*）。

然而，為求完備，我還是在此先把 *CArray* 的 *Serialize* 函式原始碼列出：

```

template<class TYPE>
void AFXAPI SerializeElements(CArchive& ar, TYPE* pElements, int nCount)
{
    ASSERT(nCount == 0 ||

```

```

        AfxIsValidAddress(pElements, nCount * sizeof(TYPE));

        // default is bit-wise read/write
        if (ar.IsStoring())
            ar.Write((void*)pElements, nCount * sizeof(TYPE));
        else
            ar.Read((void*)pElements, nCount * sizeof(TYPE));
    }

template<class TYPE, class ARG_TYPE>
void CArray<TYPE, ARG_TYPE>::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nSize);
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize, -1);
    }
    SerializeElements(ar, m_pData, m_nSize);
}

```

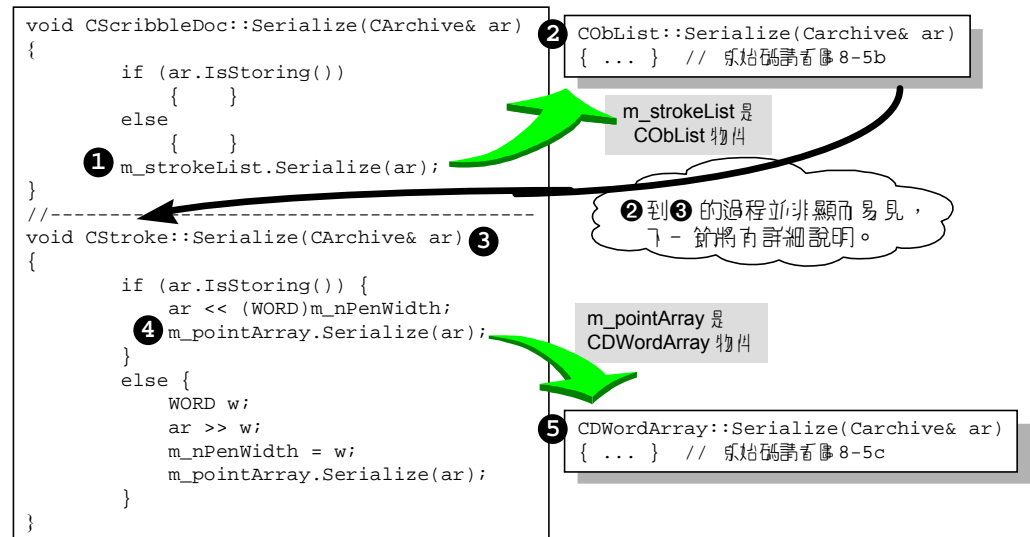


圖 8-5a Scribble Step1 的文件讀寫（檔）動作

```

void CObList::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    COBJECT::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nCount);
        for (CNode* pNode = m_pNodeHead; pNode != NULL;
             pNode = pNode->pNext)
        { // J.J.Hou : 針對串列中的每一個元素寫檔
            ASSERT(AfxIsValidAddress(pNode, sizeof(CNode)));
            ar << pNode->data;
        }
    }
    else
    {
        DWORD nNewCount = ar.ReadCount();
        COBJECT* newData;
        while (nNewCount-->0)
        { // J.J.Hou : 讀入檔案內容，加入串列
            ar >> newData;
            AddTail(newData);
        }
    }
}

```

這將引發 CArchive 的多載
運算子，稍後有深入說明。

圖 8-5b CObList::Serialize 原始碼

```

void CDWordArray::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    COBJECT::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nSize); // 把陣列大小（元素個數）寫入 ar
        ar.Write(m_pData, m_nSize * sizeof(DWORD)); // 把整個陣列寫入 ar
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize); // 從 ar 中讀出陣列大小（元素個數）
        ar.Read(m_pData, m_nSize * sizeof(DWORD)); // 從 ar 中讀出整個陣列
    }
}

```

圖 8-5c CDWordArray::Serialize 原始碼

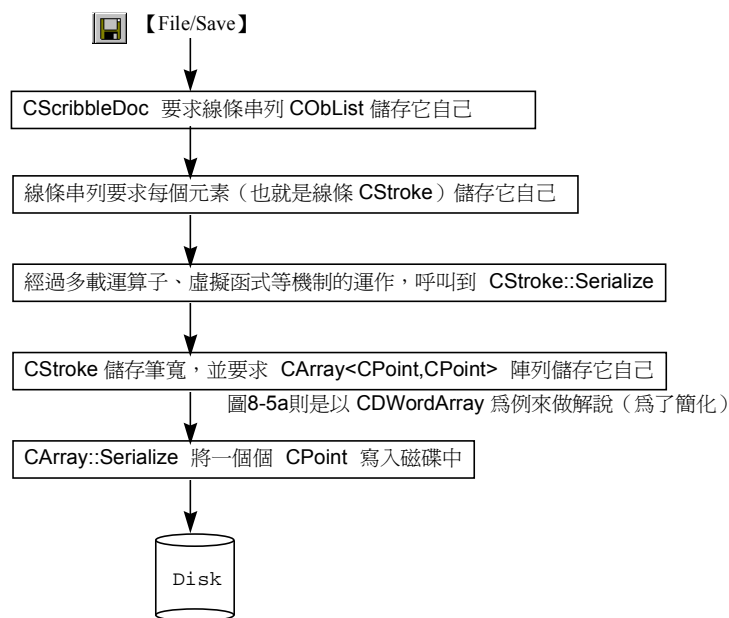


圖 8-5d Scribble Document 的 Serialize 動作細部分解。

實際看看儲存在磁碟中的 .SCB 檔案內容，對 *Serialize* 將會有深刻的體會。圖 8-6a 是使用者在 Scribble Step1 程式的繪圖畫面及存檔內容（以 Turbo Dump 觀察獲得），圖 8-6b 是檔案內容的解釋。我們必須了解隱藏在 MFC 機制中的 serialization 細部動作，才能清楚這些二進位資料的產生原由。如果你認為看傾印碼（dump code）是件令人頭暈的事情，那麼你會錯失許多美麗事物。真的，傾印碼使我們了解許多深層結構。

我在 Scribble 中作畫並存檔。為了突顯筆寬的不同，我用了第 10 章的 Step3 版本，該版本的 Document 格式與 Step1 的相同，但允許使用者設定筆寬。圖 8-6a 第一條線條的筆寬是 2，第二條是 5，第三條是 10，第四條是 20。文件儲存於 PENWIDTH.SCB 檔案中。

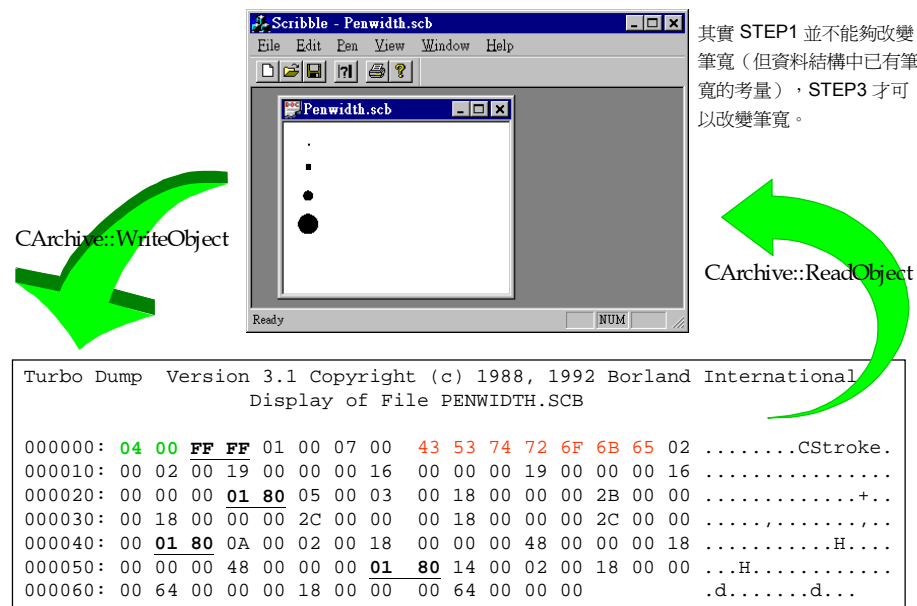


圖 8-6a 在 Scribble 中作畫並存檔。PENWIDTH.SCB 檔案全長 109 個位元組。

數值（hex）	說明
0004	表示此檔有四個 <i>COBList</i> 元素。
FFFF	FFFF 亦即 -1，表示 New Class Tag（稍後詳述）。既然是新類別，就得記錄一些相關資訊（版本號碼和類別名稱）
0001	這是 Schema no.，代表物件的版本號碼。此數值由 <i>IMPLEMENT_SERIAL</i> 巨集的第三個參數指定。
0007	表示後面接著的「類別名稱」有 7 個字元。
43 53 74 72 6F 6B 65	"CStroke"（類別名稱）的 ASCII 碼。
0002	第一條線條的寬度。
0002	第一條線條的點陣列大小（點數）。
00000019,00000016	第一條線條的第一個點座標（ <i>CPoint</i> 物件）。
00000019,00000016	第一條線條的第二個點座標（ <i>CPoint</i> 物件）。

數值 (hex)	說明
8001	這是 (wOldClassTag nClassIndex) 的組合結果，表示接下來的物件仍舊使用舊類別 (稍後詳述)
0005	第二條線條的寬度。
0003	第二條線條的點陣列大小 (點數)。
00000018,0000002B	第二條線條的第一個點座標 (CPoint 物件)。
00000018,0000002C	第二條線條的第二個點座標 (CPoint 物件)。
00000018,0000002C	第二條線條的第三個點座標 (CPoint 物件)。
8001	表示接下來的物件仍舊使用舊類別。
000A	第三條線條的寬度。
0002	第三條線條的點陣列大小 (點數)。
00000018,00000048	第三條線條的第一個點座標 (CPoint 物件)。
00000018,00000048	第三條線條的第二個點座標 (CPoint 物件)。
8001	表示接下來的物件仍舊使用舊類別。
0014	第四條線條的寬度。
0002	第四條線條的點陣列大小 (點數)。
00000018,00000064	第四條線條的第一個點座標 (CPoint 物件)。
00000018,00000064	第四條線條的第二個點座標 (CPoint 物件)。

圖 8-6b PENWIDTH.SCB 檔案內容剖析。別忘了 Intel 採用 "little-endian" 位元組排列方式，每一個字組的前後位元組係顛倒放置。

檯面下的 Serialize 檔案奧秘

你屬於打破砂鍋問到底，不到黃河心不死那一型嗎？我會滿足你的好奇心。

從應用程式碼的層面來看，關於檔案的讀寫，我們有許多環節無法打通，類別的層層呼叫動作似乎有幾個缺口，而圖 8-6a 文件檔傾印碼中神秘的 FF FF 01 00 07 00 43 53 74 72 6F 6B 65 也曖昧難明。現在讓我來抽絲剝繭。

在挖寶過程之中，我們當然需要一些工具。我不選用昂貴的電鑽、空壓機或怪手（因為你可能沒有），我只選用簡單的鶴嘴鋤和鏟子：一個文字搜尋工具，一個檔案傾印工具，一個 Visual C++ 內含的除錯器。

- GREP.COM：UNIX 上赫赫有名的文字搜尋工具，Borland C++ 編譯器套件附了一個 DOS 版。此工具可以為我們搜尋檔案中是否有特定字串。PC Tools 也有這種功能，但 PC Tools 屬於重量級裝備，不符合我的選角要求。GREP 的使用方式如下：

```
E:\MSDEV\MFC\SRC> grep -d Serialize *.cpp <Enter>
```

- TDUMP.EXE：Turbo Dump，Borland C++ 所附工具，可將任何檔案以 16 進位碼顯示。使用方式如下：

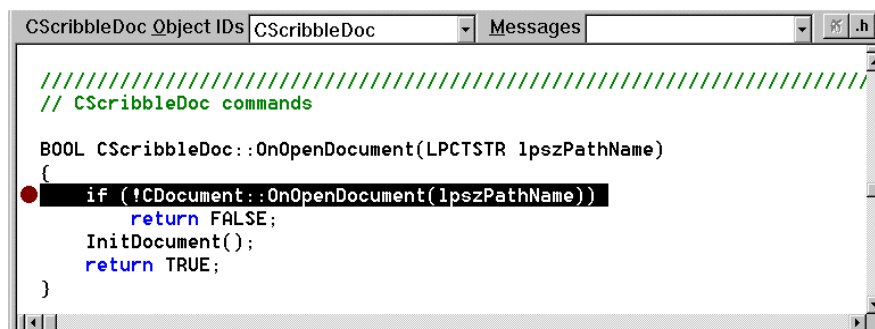
```
C:\> tdump penwidth.scb （輸出結果將送往螢幕）
```

或

```
C:\> tdump penwidth.scb > filename （輸出結果將送往檔案）
```

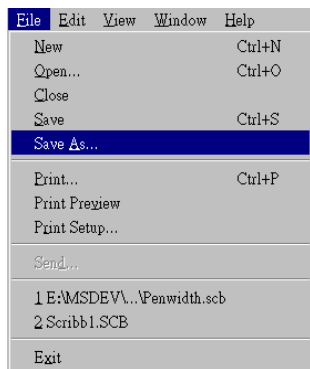
- Visual C++ 除錯器：我已在第 4 章介紹過這個除錯器。我假設你已經懂得如何設定中斷點、觀察變數值，並以 Go、Step Into、Step Over、Step Out、Step to Cursor 進行除錯。這裡我要補充的是如何觀察 "Call Stack"。

如果我把中斷點設在 `CScribbleDoc::OnOpenDocument` 函式中的第一行，



然後以 Go 進入除錯程序，當我在 Scribble 中打開一份文件（首先面對一個對話盒，然後指定檔名），程式停留在中斷點上，然後我選按【View/Call Stack】，出現【Call Stack】視窗，把中斷點之前所有未結束的函式列出來。這份資料可以幫助我們挖掘 MFC。

好，圖 8-5a 的函式流程使圖 8-6a 的文件檔傾印碼曙光乍現，但是其中有些關節仍還模模糊糊，旋明旋暗。那完全是因為 *CObList* 在處理每一個元素（一個 *CObject* 衍生類別之物件實體）的檔案動作時，有許多幕後的、不易觀察到的機制。讓我們從使用者按下【Save As】選單項目開始，追蹤程式的進行。



遍尋 Scribble 程式，並沒有發現曾經在哪裡攔截過【Save As】命令訊息，那麼必是某個「CCommandTarget 衍生類別」曾經在其 Message Map 中設定過對此訊息之處理函式。我猜想 CDocument 最有這個可能：

```
BEGIN_MESSAGE_MAP(CDocument, CCommandTarget)
    //{{AFX_MSG_MAP(CDocument)
    ON_COMMAND(ID_FILE_CLOSE, OnFileClose)
    ON_COMMAND(ID_FILE_SAVE, OnFileSave)
    ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

賓果！於是【Save As】引發 CDocument::OnFileSaveAs 被呼叫。

```
void CDocument::OnFileSaveAs()
{
    if (!DoSave(NULL))
        TRACE0("Warning: File save-as failed.\n");
}
```

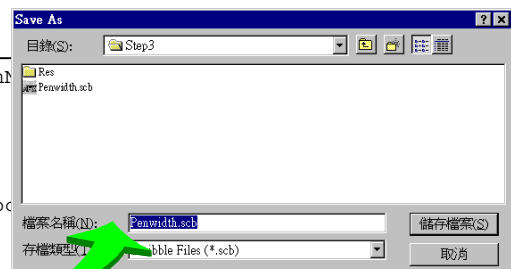
```
BOOL CDocument::DoSave(LPCTSTR lpszPathName)
{
    CString newName = lpszPathName;
    if (newName.IsEmpty())
    {
        CDocTemplate* pTemplate = GetDocTemplate();
        newName = m_strPathName;
        ...

        if (!AfxGetApp()->DoPromptFileName(newName,
            bReplace ? AFX_IDS_SAVEFILE : AFX_IDS_SAVEFILECOPY,
            OFN_HIDEREADONLY | OFN_PATHMUSTEXIST, FALSE, pTemplate))
            return FALSE;    // don't even attempt to save
    }

    CWaitCursor wait;

    if (!OnSaveDocument(newName))
    { ... }

    ...
}
```



下頁

```

BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
        CFile::modeReadWrite | CFile::shareExclusive, &fe);

    CArchive saveArchive(pFile, CArchive::store |
        CArchive::bNoFlushOnDelete);
    saveArchive.m_pDocument = this;
    saveArchive.m_bForceFlat = FALSE;
    TRY
    {
        CWaitCursor wait;
        Serialize(saveArchive);
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    ...
}

```

雖然看起來像是呼叫 CDocument::Serialize，但事實上因為 Serialize 是虛擬函式，而 CScribbleDoc 已改寫它，而且目前的 this 指標是指向 CScribbleDoc 物件（別忘了整個追蹤路線的起源是 Scribble Document；我會在下一章訊息繞行這一主題中解釋更詳盡一些），所以這裡呼叫的是 CScribbleDoc::Serialize 函式。哈，這就是虛擬函式的妙用！

```

void CScribbleDoc::Serialize(CArchive& ar)
{
    ...
    m_strokeList.Serialize(ar);
}

```

m_strokeList 是個 CObList 物件

```

void CObList::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar.WriteCount(m_nCount);
        for (CNode* pNode = m_pNodeHead;
            pNode != NULL; pNode = pNode->pNext)
        {
            ar << pNode->data;
        }
    }
    else
    {
        _AFX_INLINE CArchive& AFXAPI operator<<(CArchive& ar,
            const CObject* pObj)
        {
            { ar.WriteObject(pObj); return ar; }
        }
    }
}

```

本例之 CObList 物件內有 4 個元素，所以輸出資料 0004

CArchive 已針對 << 運算子做了多載（overloading）動作。

下頁 呼叫 CArchive::WriteObject

```
void CArchive::WriteObject(const CObject* pObj)
{
    DWORD nObjIndex;
    // make sure m_pStoreMap is initialized
    MapObject(NULL);

    if (pObj == NULL)
    { ... }
    else if ((nObjIndex = (DWORD)(*m_pStoreMap)[(void*)pObj]) != 0)
    { ... }
    else
    {
        // write class of object first
        CRuntimeClass* pClassRef = pObj->GetRuntimeClass();
        WriteClass(pClassRef);

        B // enter in stored object table, checking for overflow
        CheckCount();
        (*m_pStoreMap)[(void*)pObj] = (void*)m_nMapCount++;
        A

        // cause the object to serialize itself
        ((CObject*)pObj)->Serialize(*this);
    }
}
```

欲寫入類別資訊到檔案中，首先要從「類別型錄網」中取出 `CRuntimeClass` 資料（還記得第3章的模擬嗎？）

C

```

#define wNullTag      ((WORD)0)
#define wNewClassTag  ((WORD)0xFFFF)
#define wClassTag     ((WORD)0x8000)
#define dwBigClassTag ((DWORD)0x80000000)
#define wBigObjectTag ((WORD)0x7FFF)
#define nMaxMapCount  ((DWORD)0x3FFFFFFE)

void CArchive::WriteClass(const CRuntimeClass* pClassRef)
{
    if (pClassRef->m_wSchema == 0xFFFF)
    {
        TRACE1("Warning: Cannot call WriteClass/WriteObject for %hs.\n",
            pClassRef->m_lpszClassName);
        AfxThrowNotSupportedException();
    }
    ...
    DWORD nClassIndex;
    if ((nClassIndex = (DWORD)(*m_pStoreMap)[(void*)pClassRef]) != 0)
    {
        // previously seen class, write out the index tagged by high bit
        if (nClassIndex < wBigObjectTag)
            *this << (WORD)(wClassTag | nClassIndex);
        else
        {
            *this << wBigObjectTag;
            *this << (dwBigClassTag | nClassIndex);
        }
    }
    else
    {
        // store new class
        *this << wNewClassTag;
        pClassRef->Store(*this);
        ...
    }
}

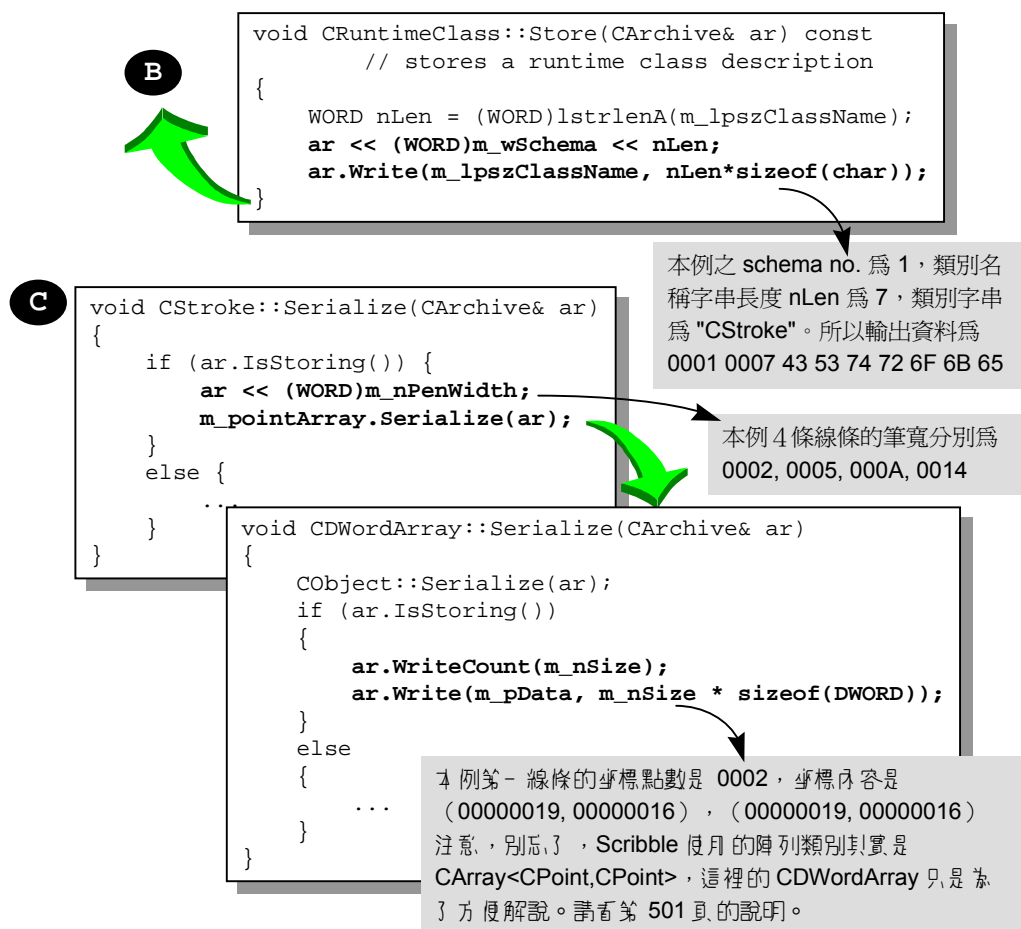
```

A

本例 CObList 串列內之元素種類（也就是「CObject 衍生類別」）只有一種（CStroke），所以 nClassIndex 永遠為 1。於是輸出資料 8001。

遇到串列的新元素（新類別），就輸出資料 FFFF。

下頁



你屬於打破砂鍋問到底，不到黃河心不死那一型嗎？這段刨根究底的過程應能解你疑惑。根據我的經驗，經過這麼一次巡禮，我們就能夠透析 MFC 的內部運作並確實掌握 MFC 的類別運用了。換言之，我們現在到達知其所以然的境界了。

檔的 Serialize 讀檔奧秘

大大地喘口氣吧，能夠把 MFC 的 *Serialize* 寫檔動作完全摸透，是件值得慰勞自己的「功績」。但是你只能輕鬆一下下，因為讀檔動作還沒有討論過，而讀檔絕不只是「寫檔的逆向操作」而已。

把物件從檔案中讀進來，究竟技術關鍵在哪裡？讀取資料當然沒問題，問題是「Document/View/Frame 三口組」怎麼產生？從檔案中讀進一個類別名稱，又如何動態產生其物件？當我從檔案讀到 "CStroke" 這個字串，並且知道它代表一個類別名稱，然後我怎麼辦？我能夠這麼做嗎：

```
CString aStr;
... // read a string from file to aStr
CStroke* pStroke = new aStr;
```

不行！這是語言版的動態生成；沒有任何一個 C++ 編譯器支援這種能力。那麼我能夠這麼做嗎：

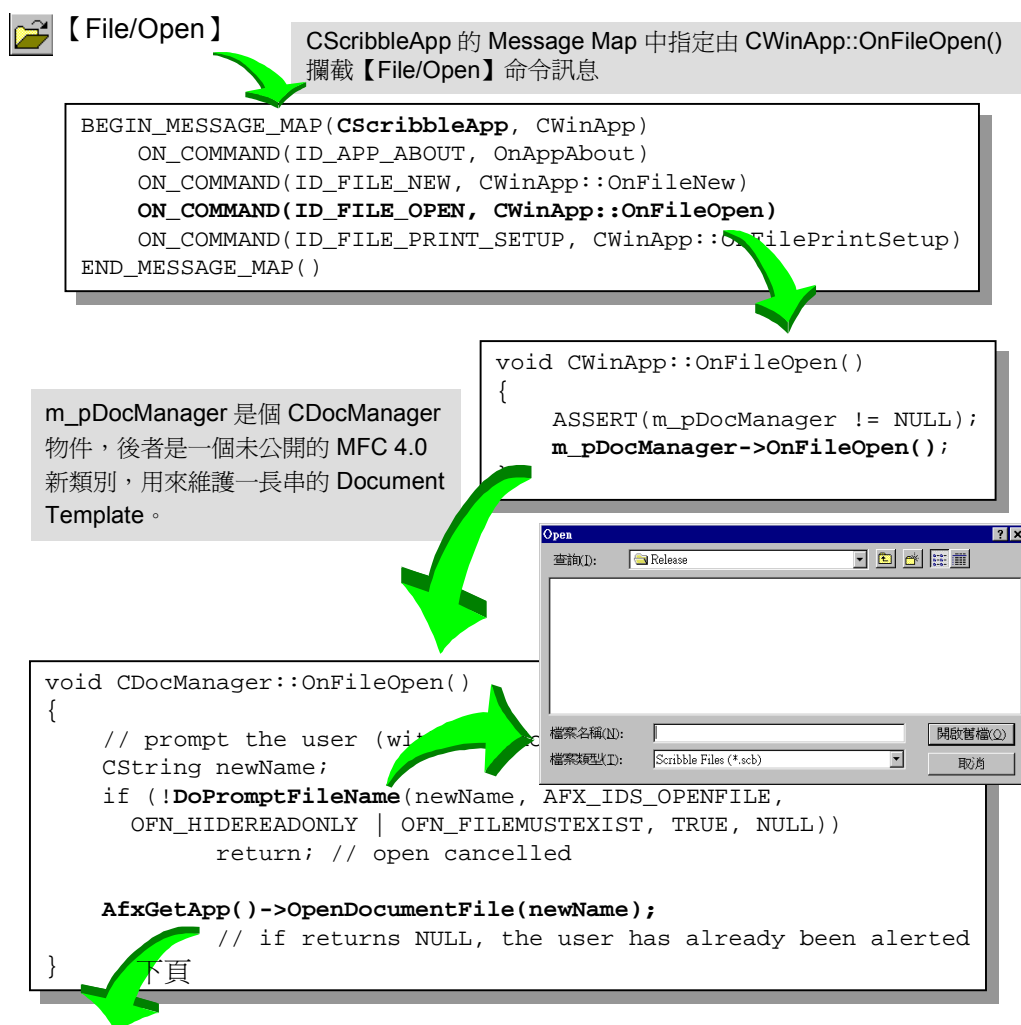
```
CString aStr;
CStroke* pStroke;
... // read a string from file to aStr
if (aStr == CString("CStroke"))
    CStroke* pStroke = new CStroke;
else if (aStr == CString("C1"))
    C1* pC1 = new C1;
else if (aStr == CString("C2"))
    C2* pC2 = new C2;
else if (aStr == CString("C3"))
    C1* pC3 = new C3;
else ...
```

可以，但真是粗糙啊。萬一再加上一種新類別呢？萬一又加上一種新類別呢？不勝其擾也！

第 3 章已經提出動態生成的觀念以及實作方式了。主要關鍵還在於一個「類別型錄網」。這個型錄網就是 *CRuntimeClass* 組成的一個串列。每一個想要享有動態生成機能的類別，都應該在「類別型錄網」上登記有案，登記資料包括物件的建構函式的指標。也就是說，上述那種極不優雅的比對動作，被 MFC 巧妙地埋起來了；應用程式可以風姿優雅地，

單單使用 `DECLARE_SERIAL` 和 `IMPLEMENT_SERIAL` 兩個巨集，就獲得檔案讀寫以及動態生成兩種機制。

我將仿效前面對於寫檔動作的探索，看看讀檔的程序如何。



```
CDocument* CWinApp::OpenDocumentFile(LPCTSTR lpszFileName)
{
    ASSERT(m_pDocManager != NULL);
    return m_pDocManager->OpenDocumentFile(lpszFileName);
}
```

很多原先在 CWinApp 中做掉的有關於 Document Template 的工作，如 AddDocTemplate、OpenDocumentFile 和 NewDocumentFile，自從 MFC 4.0 之後已隔離出來由 CDocManager 負責。

```
CDocument* CDocManager::OpenDocumentFile(LPCTSTR lpszFileName)
{
    // find the highest confidence
    CDocTemplate* pBestTemplate = NULL;
    CDocument* pOpenDocument = NULL;
    TCHAR szPath[_MAX_PATH];
    ... // 從「Document Template 串列」中找出最適當之 template，
    ... // 放到 pBestTemplate 中。
    return pBestTemplate->OpenDocumentFile(szPath);
}
```

由於 CMultiDocTemplate 改寫了 OpenDocumentFile，所以呼叫的是 CMultiDocTemplate::OpenDocumentFile。

下頁

```

CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
        BOOL bMakeVisible)
{
    CDocument* pDocument = CreateNewDocument();
    ...
    CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL);
    ...

    if (lpszPathName == NULL)
    {
        // create a new document - with default document name
        ...
    }
    else
    {
        // open an existing document
        CWaitCursor wait;
        if (!pDocument->OnOpenDocument(lpszPathName))
        {
            ...
        }
        pDocument = ...
    }
}

BOOL CScribbleDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    InitDocument();
    return TRUE;
}

InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
return pDocument;
}

```

原始碼請見本章前部之“CDocTemplate管理 CDocument/CView/CFrameWnd”一節。

由於 CScribbleDoc 改寫了 OnOpenDocument，
所以呼叫的是 CScribbleDoc::OnOpenDocument

下頁

```

BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = GetFile(lpszPathName,
        CFile::modeRead|CFile::shareDenyWrite, &fe);

    DeleteContents();
    SetModifiedFlag(); // dirty during de-serialize

    CArchive loadArchive(pFile, CArchive::load |
        CArchive::bNoFlushOnDelete);
    loadArchive.m_pDocument = this;
    loadArchive.m_bForceFlat = FALSE;
    TRY
    {
        CWaitCursor wait;
        if (pFile->GetLength() != 0)
            Serialize(loadArchive); // load me
        loadArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    ...
}

```

由於 CScribbleDoc 改寫了 Serialize，
所以呼叫的是 CScribbleDoc::Serialize

```

void CScribbleDoc::Serialize(CArchive& ar)
{
    ...
    m_strokeList.Serialize(ar);
}

```

```

void CObList::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        DWORD nNewCount = ar.ReadCount();
        CObject* newData;
        while (nNewCount--)
        {
            ar >> newData;
            AddTail(newData);
        }
    }
}

```

本例讀入 0004

operator>> 被多載 (overloading) 化

```

_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar,
    CObject*& pObj)
{ pObj = ar.ReadObject(NULL); return ar; }

```

呼叫 CArchive::ReadObject

下頁

```

CObject* CArchive::ReadObject(const CRuntimeClass*
pClassRefRequested)
{
    ...
    // attempt to load next stream as CRuntimeClass
    UINT nSchema;
    DWORD obTag;
    CRuntimeClass* pClassRef = ReadClass(pClassRefRequested,
&nSchema, &obTag);

    // check to see if tag to already loaded object
    CObject* pObj;
    if (pClassRef == NULL)
    { ... }
    else
    {
        // allocate a new object based on the CRuntimeClass
        pObj = pClassRef->CreateObject();

        // Add to mapping array BEFORE de-serialize
        CheckCount();
        m_pLoadArray->InsertAt(pObj);

        // Serialize the object
        UINT nSchemaSave = m_nObjectSchema;
        m_nObjectSchema = nSchema;
        pObj->Serialize(*this);
        m_nObjectSchema = nSchemaSave;
    }

    return pObj;
}

```

IMPLEMENT_SERIAL(CStroke,...)
會展開出一個函式如下，此即動態生
成的奧秘：

```

CObject* PASCAL CStroke::CreateObject()
{
    return new CStroke;
}

```

呼叫 CStroke::Serialize

```

CRuntimeClass* CArchive::ReadClass(const CRuntimeClass* pClassRefRequested,
UINT* pSchema, DWORD* pObTag)
{
    WORD wTag;
    *this >> wTag;

    ...
    CRuntimeClass* pClassRef;
    UINT nSchema;
    if (wTag == wNewClassTag)
    {
        // new object follows a new class id
        if ((pClassRef = CRuntimeClass::Load(*this, &nSchema)) == NULL)
        { ... }
    }
    else
    {
        DWORD nClassIndex;
        // 判斷 nClassIndex 為舊類別，於是從類別型錄網中取出
        // 其 CRuntimeClass，放在 pClassRef 中。
        ...
    }
    ...
    return pClassRef;
}

```

本例讀入 FFFF

```

B CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar,
      UINT* pwSchemaNum) // loads a runtime class description
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    WORD wTemp;
    ar >> wTemp;
    *pwSchemaNum
    ar >> nLen;

    if (nLen >= _countof(szClassName) ||
        ar.Read(szClassName, nLen*sizeof(char)) != nLen*sizeof(char))
    {
        return NULL;
    }
    szClassName[nLen] = '\0';

    // search app specific classes
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    AfxLockGlobals(CRIT_RUNTIMECLASSLIST);
    for (pClass = pModuleState->m_classList; pClass != NULL;
        pClass = pClass->m_pNextClass)
    {
        if (lstrcmpA(szClassName, pClass->m_lpszClassName) == 0)
        {
            AfxUnlockGlobals(CRIT_RUNTIMECLASSLIST);
            return pClass;
        }
    }
    ...
}
    
```

本例讀入 0001

本例讀入 0007

本例讀入 "CStroke"

檢驗整個「類別型錄網」

D

```

void CStroke::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}

```

本例讀入 0002

```

void CDWordArray::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);

    if (ar.IsStoring())
    {
        ...
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize);
        ar.Read(m_pData, m_nSize * sizeof(DWORD));
    }
}

```

本例讀入 0002

本例讀入 00000019, 00000016, 00000019, 00000016

注意，別忘了，Scribble 使用的陣列類別其實是 CArray<CPoint, CPoint>，這裡的 CDWordArray 只是為了方便解說。請看本書“檯面”的 Serialize 動作（#501 頁）中的方塊說明。

DYNAMIC / DYNCREATE / SERIAL 巨集

我猜你被三組看起來難分難解的巨集困擾著，它們是：

- `DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC`
- `DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE`
- `DECLARE_SERIAL / IMPLEMENT_SERIAL`

事實上我已經在第 3 章揭露其原始碼及其觀念了。這裡再以圖 8-7 三張圖片把巨集原始碼、展開結果、以及帶來的影響做個整理。`SERIAL` 巨集中比較令人費解的是它對 `>>` 運算子的多載動作。稍後我有一個 *CArchive* 小節，會交待其中細節。

你將在圖 8-7abc 中看到幾個令人困惑的大寫常數，像是 `AFXAPI`、`AFXDATA` 等等。它們的意義可以在 VC++ 5.0 的 `\DEVSTUDIO\VC\MFC\INCLUDE\AFXVER_.H` 中獲得：

```
// AFXAPI is used on global public functions
#ifndef AFXAPI
    #define AFXAPI __stdcall
#endif

#define AFX_DATA
#define AFX_DATADEF
```

後二者就像 `afx_msg` 一樣（我曾經在第 6 章的 Hello MFC 原始碼一出現之後解釋過），是一個 "intentional placeholder"，可能在將來會用到，目前則為「無物」。

爾曰顯淺，彼云艱深，唯其深入，所以淺出

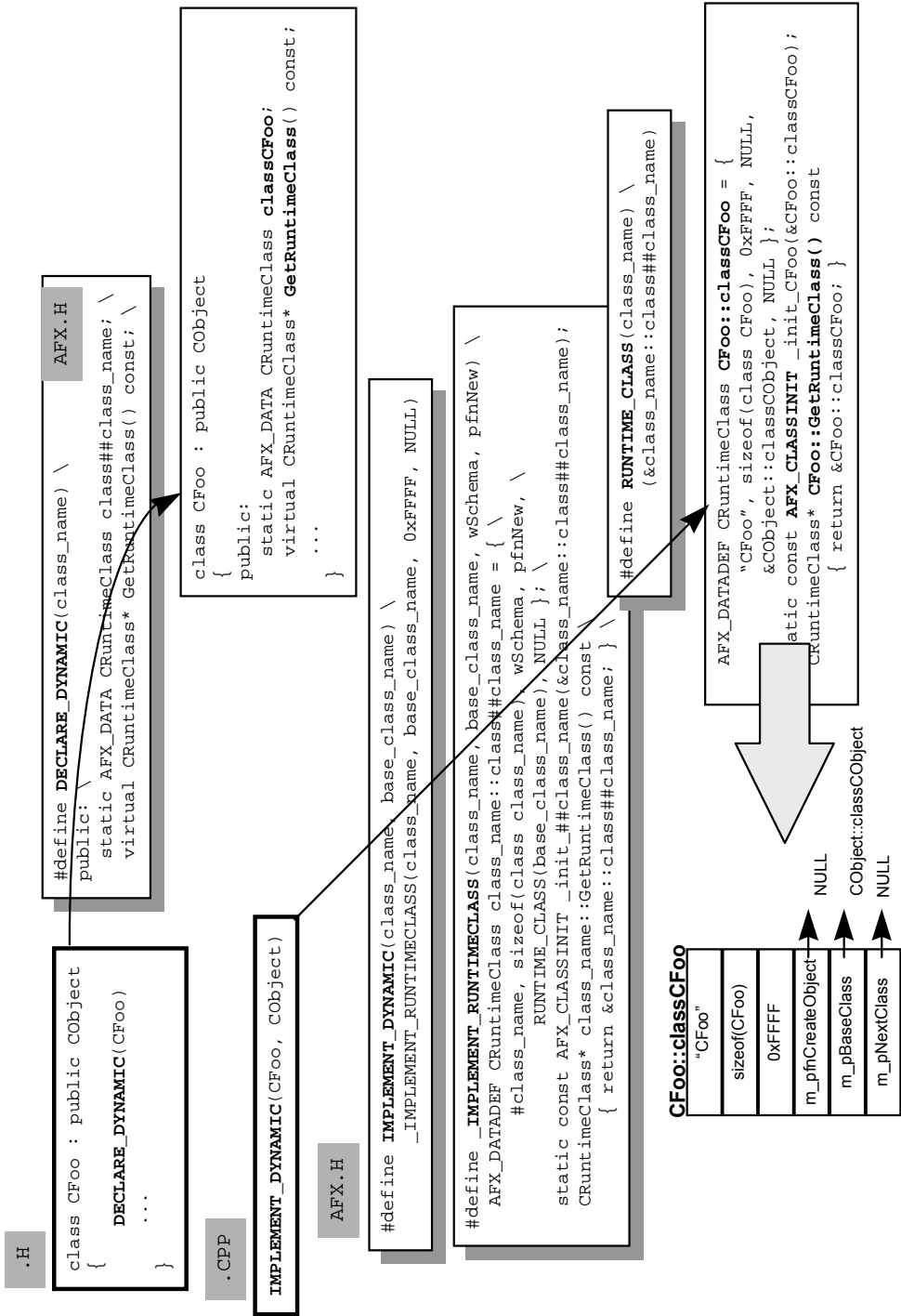


圖 8-7a DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC

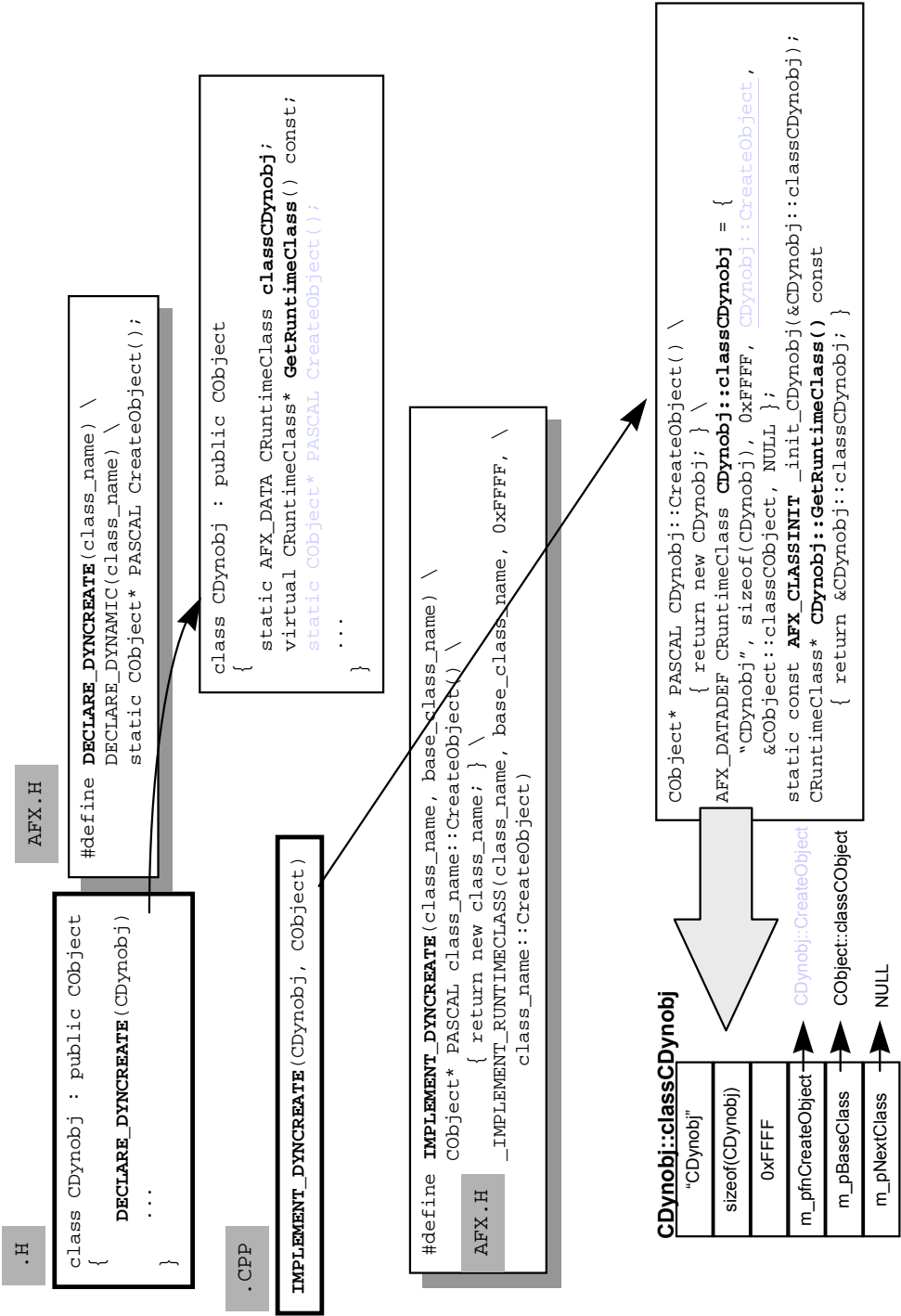


圖8-7b DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE

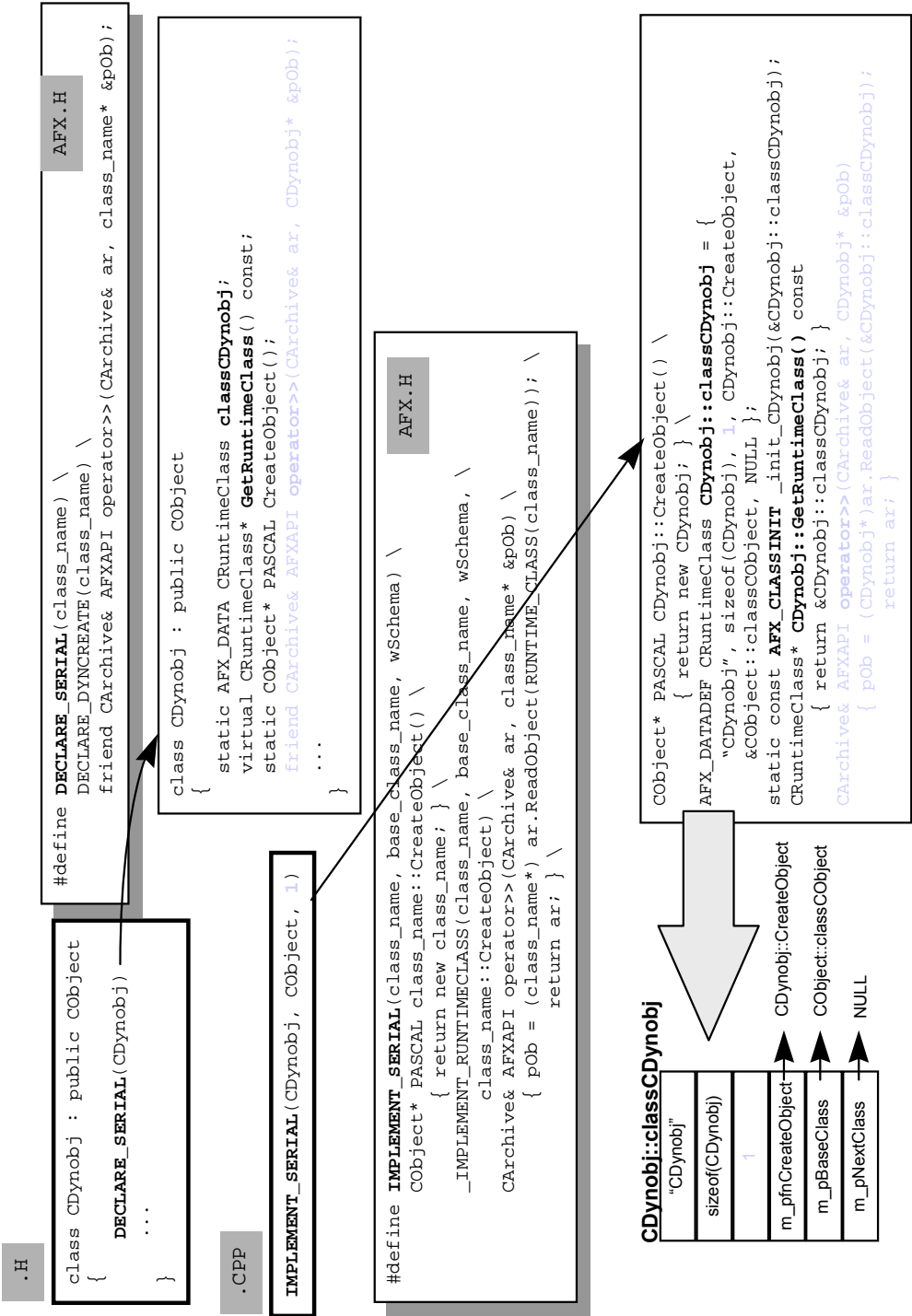


圖 8-7c DECLARE_SERIAL / IMPLEMENT_SERIAL

DYNAMIC / *DYNCREATE* / *SERIAL* 三套巨集分別在 *CRuntimeClass* 所組成的「類別型錄網」中填寫不同的記錄，使 MFC 類別（以及你自己的類別）分別具備三個等級的性能：

- 基礎機能以及物件診斷（可利用 *afxDump* 輸出診斷訊息），以及 Run Time Type Information (RTTI)。也有人把 RTTI 稱為 Run Time Class Information (RTCI)。
- 動態生成（Dynamic Creation）
- 檔案讀寫（Serialization）

你的類別究竟擁有什麼等級的性能，得視其所使用的巨集而定。三組巨集分別實現不同等級的功能，如圖 8-8。

巨集 \ 功能	RTTI CObject::IsKindOf	Dynamic Creation CRuntimeClass::CreateObject	Serialize CArchive::operator>> CArchive::operator<<
DYNAMIC	Yes	No	No
DYNCREATE	Yes	Yes	No
SERIAL	Yes	Yes	Yes

圖 8-8 三組巨集分別實現不同等級的功能。

Scribble Step1 程式中與主結構相關的六個類別，所使用的各式巨集整理如下：

類別名稱	基礎類別	使用之巨集
CScribbleApp	CWinApp	None
CMainFrame	CMDIFrameWnd	DECLARE_DYNAMIC(CMainFrame) IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
CChildFrame	CMDIChildWnd	DECLARE_DYNCREATE(CChildFrame) IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
CScribbleDoc	CDocument	DECLARE_DYNCREATE(CScribbleDoc) IMPLEMENT_DYNCREATE(CscribbleDoc, CDocument)
CStroke	CObject	DECLARE_SERIAL(CStroke) IMPLEMENT_SERIAL(Cstroke, Cobject, 1)
CScribbleView	CView	DECLARE_DYNCREATE(CScribbleView) IMPLEMENT_DYNCREATE(CscribbleView, CView)

Serializable 的必要條件

欲讓一個物件有 `Serialize` 能力，它必須衍生自一個 `Serializable` 類別。一個類別意欲成為 `Serializable`，必須有下列五大條件；至於其原因，前面的討論已經全部交待過了。

1. 從 `CObject` 衍生下來。如此一來可保有 `RTTI`、`Dynamic Creation` 等機能。
2. 類別的宣告部份必須有 `DECLARE_SERIAL` 巨集。此巨集需要一個參數：類別名稱。
3. 類別的實作部份必須有 `IMPLEMENT_SERIAL` 巨集。此巨集需要三個參數：一是類別名稱，二是父類別名稱，三是 `schema no.`。
4. 改寫 `Serialize` 虛擬函式，使它能夠適當地把類別的成員變數寫入檔案中。
5. 為此類別加上一個 `default` 建構式（也就是無參數之建構式）。這個條件常為人所忽略，但它是必要的，因為若一個物件來自檔案，`MFC` 必須先動態生成它，而且在沒有任何參數的情況下呼叫其建構式，然後才從檔案中讀出物件資料。

如此，讓我們再複習一次本例之 *CStroke*，看看是否符合上述五大條件：

```
// in SCRIBBLEDOC.H
class CStroke : public CObject // 衍生自 CObject (條件1)
{
public:
    CStroke(UINT nPenWidth);

protected:
    CStroke(); // 擁有一個 default constructor (條件5)
    DECLARE_SERIAL(CStroke) // 使用 SERIAL 巨集 (條件2)

protected:
    UINT m_nPenWidth;
public:
    CArray<CPoint,CPoint> m_pointArray;

public:
    virtual void Serialize(CArchive& ar); // 改寫 Serialize 函式 (條件4)
};

// in SCRIBBLEDOC.CPP
IMPLEMENT_SERIAL(CStroke, CObject, 1) // 使用 SERIAL 巨集 (條件3)

CStroke::CStroke() // 擁有一個 default constructor (條件5)
{
    // This empty constructor should be used by serialization only
}

void CStroke::Serialize(CArchive& ar) // 改寫 Serialize 函式 (條件4)
{
    CObject::Serialize(ar); // 手冊上告訴我們最好先呼叫此函式。
                           // 目前 MFC 版本中它是空函式，所以不呼叫也沒關係。
    if (ar.IsStoring())
    {
        ar << (WORD)m_nPenWidth;
        m_pointArray.Serialize(ar);
    }
    else
    {
        WORD w;
        ar >> w;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}
```

CObject 類別

爲什麼絕大部份的 MFC 類別，以及許多你自己的類別，都要從 *CObject* 衍生下來呢？因爲當一個類別衍生自 *CObject*，它也就繼承了許多重要的性質。*CObject* 這個「老祖宗」至少提供兩個機能（兩個虛擬函式）：*IsKindOf* 和 *IsSerializable*。

IsKindOf

當 Framework 掌握「類別型錄網」這張王牌，要設計出 *IsKindOf* 根本不是問題。所謂 *IsKindOf* 就是 RTTI 的化身，用白話說就是「xxx 物件是一種 xxx 類別嗎？」例如「長臂猿是一種哺乳類嗎？」「藍鯨是一種魚類嗎？」凡支援 RTTI 的程式就必須接受這類詢問，並對前者回答 Yes，對後者回答 No。

下面是 *CObject::IsKindOf* 虛擬函式的原始碼：

```
// in OBJCORE.CPP
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
    // simple SI case
    CRuntimeClass* pClassThis = GetRuntimeClass();
    return pClassThis->IsDerivedFrom(pClass);
}

                                ↓
BOOL CRuntimeClass::IsDerivedFrom(const CRuntimeClass* pBaseClass) const
{
    // simple SI case
    const CRuntimeClass* pClassThis = this;
    while (pClassThis != NULL)
    {
        if (pClassThis == pBaseClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;           // walked to the top, no match
}
```

這項作爲，也就是在圖 8-9 中藉著 *m_pBaseClass* 尋根。只要在尋根過程中比對成功，就傳回 *TRUE*，否則傳回 *FALSE*。而你知道，圖 8-9 的「類別型錄網」是靠 *DECLARE_DYNAMIC* 和 *IMPLEMENT_DYNAMIC* 巨集建構起來的。第 3 章的

「RTTI」一節對此多有說明。

IsSerializable

一個類別若要是能夠進行 *Serialization* 動作，必須準備 *Serialize* 函式，並且在「類別型錄網」中自己的那個 *CRuntimeClass* 元素裡的 *schema* 欄位裡設立 0xFFFF 以外的號碼，代表資料格式的版本（這樣才能提供機會讓設計較佳的 *Serialize* 函式能夠區分舊版資料或新版資料，避免牛頭不對馬嘴的困惑）。這些都是 *DECLARE_SERIAL* 和 *IMPLEMENT_SERIAL* 巨集的责任範圍。

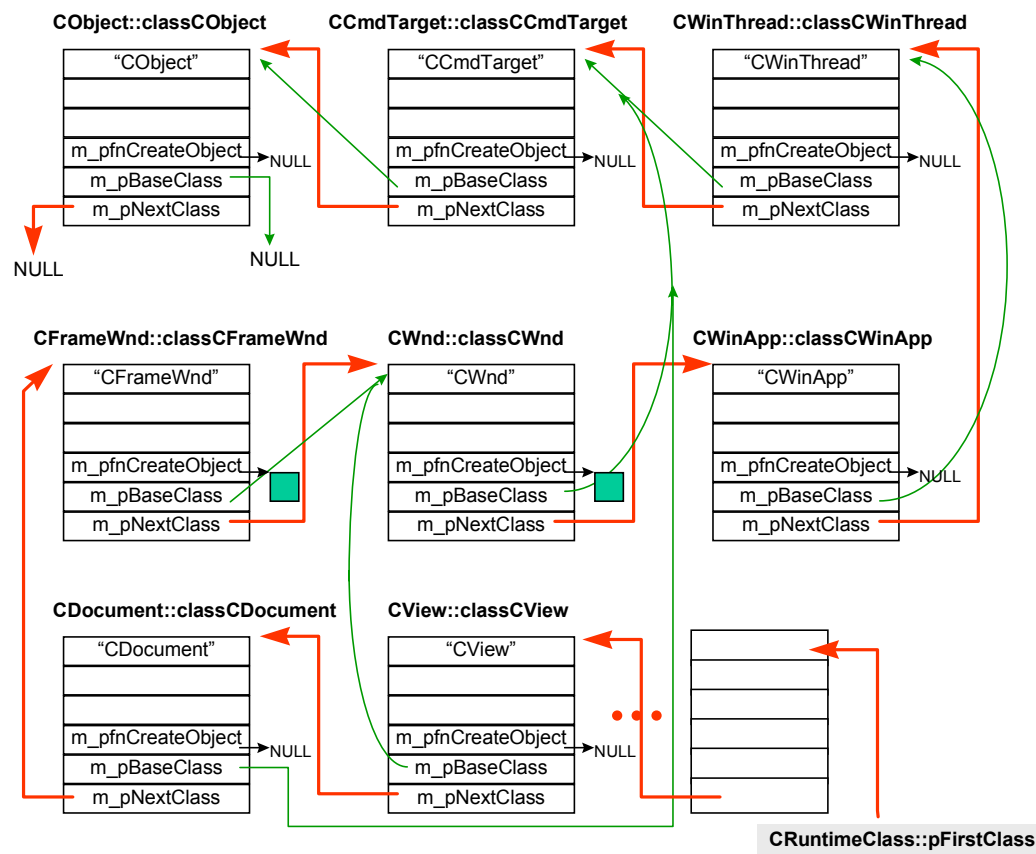


圖 8-9 *DECLARE_* 和 *IMPLEMENT_* 巨集合力建構起這張網。於是 RTTI 和 Dynamic Creation 和 *Serialization* 等機能便可輕易達成。

CObject 提供了一個虛擬函式，讓程式在執行時期判斷某類別的 *schema* 號碼是否為 0xFFFF，藉此得知它是否可以 *Serialize*：

```
BOOL CObject::IsSerializable() const
{
    return (GetRuntimeClass()->m_wSchema != 0xffff);
}
```

CObject::Serialize

這是一個虛擬函式。每一個希望具備 *Serialization* 能力的類別都應該改寫它。事實上 *Wizard* 為我們做出來的程式碼中也都會自動加上這個函式的呼叫動作。MFC 手冊上總是說，每一個你所改寫的 *Serialize* 函式都應該在第一時間呼叫此一函式，那麼是不是 *CObject::Serialize* 之中有什麼重要的動作？

```
// in AFX.INL
_AFX_INLINE void CObject::Serialize(CArchive&)
{ /* CObject does not serialize anything by default */ }
```

不，什麼也沒有。所以，現階段（至少截至 MFC 4.0）你可以不必理會手冊上的諄諄告誨。然而，Microsoft 很有可能改變 *CObject::Serialize* 的內容，屆時沒有遵循告誨的人恐怕就後悔了。

CArchive 類別

談到 *Serialize* 就不能不談 *CArchive*，因為 *serialize* 的對象（無論讀或寫）是一個 *CArchive* 物件，這一點相信你已經從上面數節討論中熟悉了。基本上你可以想像 *archive* 相當於檔案，不過它其實是檔案之前的一個記憶體緩衝區。所以我們才會在前面的「檯面下的 *Serialize* 奧秘」中看到這樣的動作：

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
        CFile::modeReadWrite | CFile::shareExclusive, &fe);

    // 令 file 和 archive 產生關聯
```

```

        CArchive saveArchive(pFile, CArchive::store |
                               CArchive::bNoFlushOnDelete);
        ...
        Serialize(saveArchive);    // 對著 archive 做 serialize 動作
        ...
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }

    BOOL CDocument::OnOpenDocument(LPCTSTR lpszPathName)
    {
        CFile* pFile = GetFile(lpszPathName,
                                CFile::modeRead|CFile::shareDenyWrite, &fe);

        // 令 file 和 archive 產生關聯
        CArchive loadArchive(pFile, CArchive::load |
                               CArchive::bNoFlushOnDelete);
        ...
        Serialize(loadArchive);    // 對著 archive 做 serialize 動作
        ...
        loadArchive.Close();
        ReleaseFile(pFile, FALSE);
    }

```

operator<< 和 operator>>

CArchive 針對許多 C++ 資料型態、Windows 資料型態以及 *CObject* 衍生類別，定義

operator<< 和 *operator>>* 多載運算子：

```

// in AFX.H
class CArchive
{
public:
    // Flag values
    enum Mode { store = 0, load = 1, bNoFlushOnDelete = 2, bNoByteSwap = 4 };
    CArchive(CFile* pFile, UINT nMode, int nBufSize = 4096, void* lpBuf = NULL);
    ~CArchive();

    // Attributes
    BOOL IsLoading() const;
    BOOL IsStoring() const;
    BOOL IsByteSwapping() const;
    BOOL IsBufferEmpty() const;

    CFile* GetFile() const;

```

```

UINT GetObjectSchema(); // only valid when reading a CObject*
void SetObjectSchema(UINT nSchema);

// pointer to document being serialized -- must set to serialize
// COleClientItems in a document!
CDocument* m_pDocument;

// Operations
UINT Read(void* lpBuf, UINT nMax);
void Write(const void* lpBuf, UINT nMax);
void Flush();
void Close();
void Abort(); // close and shutdown without exceptions

// reading and writing strings
void WriteString(LPCTSTR lpsz);
LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
BOOL ReadString(CString& rString);

public:
    // Object I/O is pointer based to avoid added construction overhead.
    // Use the Serialize member function directly for embedded objects.
    friend CArchive& AFXAPI operator<<(CArchive& ar, const CObject* pObj);

    friend CArchive& AFXAPI operator>>(CArchive& ar, CObject* pObj);
    friend CArchive& AFXAPI operator>>(CArchive& ar, const CObject* pObj);

    // insertion operations
    CArchive& operator<<(BYTE by);
    CArchive& operator<<(WORD w);
    CArchive& operator<<(LONG l);
    CArchive& operator<<(DWORD dw);
    CArchive& operator<<(float f);
    CArchive& operator<<(double d);

    CArchive& operator<<(int i);
    CArchive& operator<<(short w);
    CArchive& operator<<(char ch);
    CArchive& operator<<(unsigned u);

    // extraction operations
    CArchive& operator>>(BYTE& by);
    CArchive& operator>>(WORD& w);
    CArchive& operator>>(DWORD& dw);
    CArchive& operator>>(LONG& l);
    CArchive& operator>>(float& f);
    CArchive& operator>>(double& d);

```

```

CArchive& operator>>(int& i);
CArchive& operator>>(short& w);
CArchive& operator>>(char& ch);
CArchive& operator>>(unsigned& u);

// object read/write
CObject* ReadObject(const CRuntimeClass* pClass);
void WriteObject(const CObject* pObj);
// advanced object mapping (used for forced references)
void MapObject(const CObject* pObj);

// advanced versioning support
void WriteClass(const CRuntimeClass* pClassRef);
CRuntimeClass* ReadClass(const CRuntimeClass* pClassRefRequested = NULL,
    UINT* pSchema = NULL, DWORD* pObjTag = NULL);
void SerializeClass(const CRuntimeClass* pClassRef);
...
protected:
    // array/map for CObject* and CRuntimeClass* load/store
    UINT m_nMapCount;
    union
    {
        CPtrArray* m_pLoadArray;
        CMapPtrToPtr* m_pStoreMap;
    };
    // map to keep track of mismatched schemas
    CMapPtrToPtr* m_pSchemaMap;
    ...
};

```

這些多載運算子均定義於 AFX.INL 檔案中。另有些函式可能你會覺得眼熟，沒錯，它們在稍早的「檯面下的 Serialize 奧秘」中已經出現過了，它們是 *ReadObject*、*WriteObject*、*ReadClass*、*WriteClass*。

各種型態的 *operator>>* 和 *operator<<* 多載運算子，正是為什麼你可以將各種型態的資料（甚至包括 *CObject**）讀出或寫入 archive 的原因。一個「C++ 類別」（而非一般資料型態）如果希望有 *Serialization* 機制，它的第一要件就是直接或間接衍生自 *CObject*，為的是希望自 *CObject* 繼承下列三個運算子：

```

// in AFX.INL
_AFX_INLINE CArchive& AFXAPI operator<<(CArchive& ar, const CObject* pObj)
{ ar.WriteObject(pObj); return ar; }
_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar, CObject*& pObj)

```

```

        { pObj = ar.ReadObject(NULL); return ar; }
_AFX_INLINE CArchive& AFXAPI operator>>(CArchive& ar, const CObject*& pObj)
    { pObj = ar.ReadObject(NULL); return ar; }

```

其中 *CArchive::WriteObject* 先把類別的 *CRuntimeClass* 資訊寫出，再呼叫類別的 *Serialize* 函式。*CArchive::ReadObject* 的行為類似，先把類別的 *CRuntimeClass* 資訊讀入，再呼叫類別的 *Serialize* 函式。*Serialize* 是 *CObject* 的虛擬函式，因此你必須確定你的類別改寫的 *Serialize* 函式的回返值和參數型態都符合 *CObject* 中的宣告：傳回值為 *void*，唯一一個參數為 *CArchive&*。

注意：*CString*、*CRect*、*CSize*、*CPoint* 並不衍生自 *CObject*，但它們也可以直接使用針對 *CArchive* 的 *<<* 和 *>>* 運算子，因為它們自己設計了一套：

```

// in AFX.H
class CString
{
    friend CArchive& AFXAPI operator<<(CArchive& ar, const CString& string);
    friend CArchive& AFXAPI operator>>(CArchive& ar, CString& string);
    ...
};

// in AFXWIN.H
// Serialization
CArchive& AFXAPI operator<<(CArchive& ar, SIZE size);
CArchive& AFXAPI operator<<(CArchive& ar, POINT point);
CArchive& AFXAPI operator<<(CArchive& ar, const RECT& rect);
CArchive& AFXAPI operator>>(CArchive& ar, SIZE& size);
CArchive& AFXAPI operator>>(CArchive& ar, POINT& point);
CArchive& AFXAPI operator>>(CArchive& ar, RECT& rect);

```

一個類別如果希望有 *Serialization* 機制，它的第二要件就是使用 *SERIAL* 巨集。這個巨集包容 *DYNCREATE* 巨集，並且在類別的宣告之中加上：

```
friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pObj);
```

在類別的實作檔中加上：

```

CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pObj) \
    { pObj = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
      return ar; } \

```

如果我的類別名為 *CStroke*，那麼經由

```
class CStroke : public CObject
{
    ...
    DECLARE_SERIAL(CStroke)
}
```

和

```
IMPLEMENT_SERIAL(CStroke, CObject, 1)
```

我就獲得了兩組和 *CArchive* 讀寫動作的關鍵性程式碼：

```
class CStroke : CObject
{
    ...
    friend CArchive& AFXAPI operator>>(CArchive& ar, CStroke* &pOb);
}

CArchive& AFXAPI operator>>(CArchive& ar, CStroke* &pOb)
{ pOb = (CStroke*) ar.ReadObject(RUNTIME_CLASS(CStroke));
  return ar; }
```

好，你看到了，為什麼只改寫 `operator>>`，而沒有改寫 `operator<<`？原因是 *WriteObject* 並不需要 *CRuntimeClass* 資訊，但 *ReadObject* 需要，因為在讀完檔案後還要動態生成的動作。

效率考量

我想你一定在前面解剖文件檔傾印碼時就注意到了，當文件檔內含有許多物件資料時，凡物件隸屬同一類別者，只有第一個物件才連同類別的 *CRuntimeClass* 資訊一併寫入，此後同類別之物件僅以一個代碼表示，例如圖 8-6c 中時而出現的 8001 代碼。為了效率的考量，這是有必要的。想想看，如果一張 *Scribble* 圖形有成千上萬個線條，難不成要寫入成千上萬個 *CStroke* 資訊不成？在哈滴（Hard Disk）極為便宜的今天，考慮的重點並不是檔案的大小，而是檔案大小背後所影響的讀寫時間，以及網路傳輸時間。別忘了，一切桌上的東西都將躍於網上。

CArchive 維護類別資訊的作法是，當它做輸出動作，物件名稱以及參考值被維護在一個 *map* 之中；當它做讀入動作，它把物件維護在一個 *array* 之中。*CArchive* 中的成員變數

m_pSchemaMap 就是爲此而來：

```
union
{
    CPtrArray* m_pLoadArray;
    CMapPtrToPtr* m_pStoreMap;
};
// map to keep track of mismatched schemas
CMapPtrToPtr* m_pSchemaMap;
```

自定 SERIAL 巨集給抽象類別使用

你是知道的，所謂抽象類別就是包含純虛擬函式的類別，所謂純虛擬函式就是只有宣告沒有定義的虛擬函式。所以，你不可能將抽象類別具現化（instantiated）。那麼，*IMPLEMENT_SERIAL* 展開所得的這段碼：

```
CObject* PASCAL class_name::CreateObject() \
{ return new class_name; } \
```

面對如果一個抽象類別 *class_name* 就行不通了，編譯時會產生錯誤訊息。這時你得自行定義巨集如下：

```
#define IMPLEMENT_SERIAL_MY(class_name, base_class_name, wSchema) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, NULL) \
    CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
      return ar; } \
```

也就是，令 *CreateObject* 函式爲 *NULL*，這才能夠使用於抽象類別之中。

在 CObList 中加進 CStroke 以外的類別

Scribble Document 傾印碼中的那個代表「舊類別」的 8001 一直令我如坐針氈。不知道什麼情況下會出現 8002？或是 8003？或是什麼其他東東。因此，我打算做點測試。除了 *CStroke*，我打算再加上 *CRectangle* 和 *CCircle* 兩個類別，並把其物件掛到 *CObList* 中。這個修改純粹爲了測試不同類別寫到文件檔中會造成什麼後果，沒有考慮使用者介面或任何週邊因素，我並不是真打算爲 Scribble 加上畫四方形和畫圓形的功能（不過如

果你喜歡，這倒是能夠給你作為一個導引)，所以我把 Step1 拷貝一份，在拷貝版上做文章。

首先我必須宣告 *CCircle* 和 *CRectangle*。在新檔案中做這件事當然可以，但考慮到簡化問題，以及它們與 *CStroke* 可能會有彼此前置參考的情況，我還是把它們放在原有的 *ScribbleDoc.h* 中好了。為了能夠 `serialize`，它們都必須衍生自 *CObject*，使用 *DECLARE_SERIAL* 巨集，並改寫 *Serialize* 虛擬函式，而且擁有 default constructor。

CRectangle 有一個成員變數 *CRect m_rect*，代表四方形的四個點；*CCircle* 有一個成員變數 *CPoint m_center* 和一個成員變數 *UINT m_radius*，代表圓心和半徑：

```
#0001 class CRectangle : public CObject
#0002 {
#0003 public:
#0004     CRectangle(CRect rect);
#0005
#0006 protected:
#0007     CRectangle();
#0008     DECLARE_SERIAL(CRectangle)
#0009
#0010 // Attributes
#0011     CRect m_rect;
#0012
#0013 public:
#0014     virtual void Serialize(CArchive& ar);
#0015 };
#0016
#0017 class CCircle : public CObject
#0018 {
#0019 public:
#0020     CCircle(CPoint center, UINT radius);
#0021
#0022 protected:
#0023     CCircle();
#0024     DECLARE_SERIAL(CCircle)
#0025
#0026 // Attributes
#0027     CPoint      m_center;
#0028     UINT        m_radius;
#0029
#0030 public:
#0031     virtual void Serialize(CArchive& ar);
#0032 };
```

接下來我必須在 `ScribbleDoc.cpp` 中使用 `IMPLEMENT_SERIAL` 巨集，並定義成員函式。手冊上要求每一個 `Serializable` 類別都應該準備一個空的建構式（`default constructor`）。照著做吧，免得將來遺憾：

```
#0001 IMPLEMENT_SERIAL(CRectangle, CObject, 1)
#0002
#0003 CRectangle::CRectangle()
#0004 {
#0005     // this empty constructor should be used by serialization only
#0006 }
#0007
#0008 CRectangle::CRectangle(CRect rect)
#0009 {
#0010     m_rect = rect;
#0011 }
#0012
#0013 void CRectangle::Serialize(CArchive& ar)
#0014 {
#0015     if (ar.IsStoring())
#0016     {
#0017         ar << m_rect;
#0018     }
#0019     else
#0020     {
#0021         ar >> m_rect;
#0022     }
#0023 }
#0024
#0025 IMPLEMENT_SERIAL(CCircle, CObject, 1)
#0026
#0027 CCircle::CCircle()
#0028 {
#0029     // this empty constructor should be used by serialization only
#0030 }
#0031
#0032 CCircle::CCircle(CPoint center, UINT radius)
#0033 {
#0034     m_radius = radius;
#0035     m_center = center;
#0036 }
#0037
#0038 void CCircle::Serialize(CArchive& ar)
#0039 {
#0040     if (ar.IsStoring())
```

```

#0041      {
#0042          ar << m_center;
#0043          ar << m_radius;
#0044      }
#0045      else
#0046      {
#0047          ar >> m_center;
#0048          ar >> m_radius;
#0049      }
#0050 }

```

接下來我應該改變使用者介面，加上選單或工具列，以便在塗鴉過程中得隨時加上一個四方形或一個圓圈。但我剛才說了，我只是打算做個小小的文件檔格式測試而已，所以簡單化是我的最高指導原則。我打算搭現有之使用者介面的便車，也就是每次滑鼠左鍵按下開始一條線條之後，再 *new* 一個四方形和一個圓形，並和線條一起加入 *CObList* 之中，然後才開始接受左鍵的座標...

所以，我修改 *CScribDoc::NewStroke* 函式如下：

```

#0001 CStroke* CScribDoc::NewStroke()
#0002 {
#0003     CStroke* pStrokeItem = new CStroke(m_nPenWidth);
#0004     CRectangle* pRectItem = new CRectangle(CRect(0x11,0x22,0x33,0x44));
#0005     CCircle* pCircleItem = new CCircle(CPoint(0x55,0x66),0x77);
#0006     m_strokeList.AddTail(pStrokeItem);
#0007     m_strokeList.AddTail(pRectItem);
#0008     m_strokeList.AddTail(pCircleItem);
#0009
#0010     SetModifiedFlag(); // Mark the document as having been modified,
#0011                       // for purposes of confirming File Close.
#0012     return pStrokeItem;
#0013 }

```

並將 *scribbledoc.h* 中的 *m_strokeList* 修改為：

```
CTypedPtrList<CObList, CObject*> m_strokeList;
```

重新編譯連結，獲得結果如圖 8-10a。圖 8-10b 對此結果有詳細的剖析。

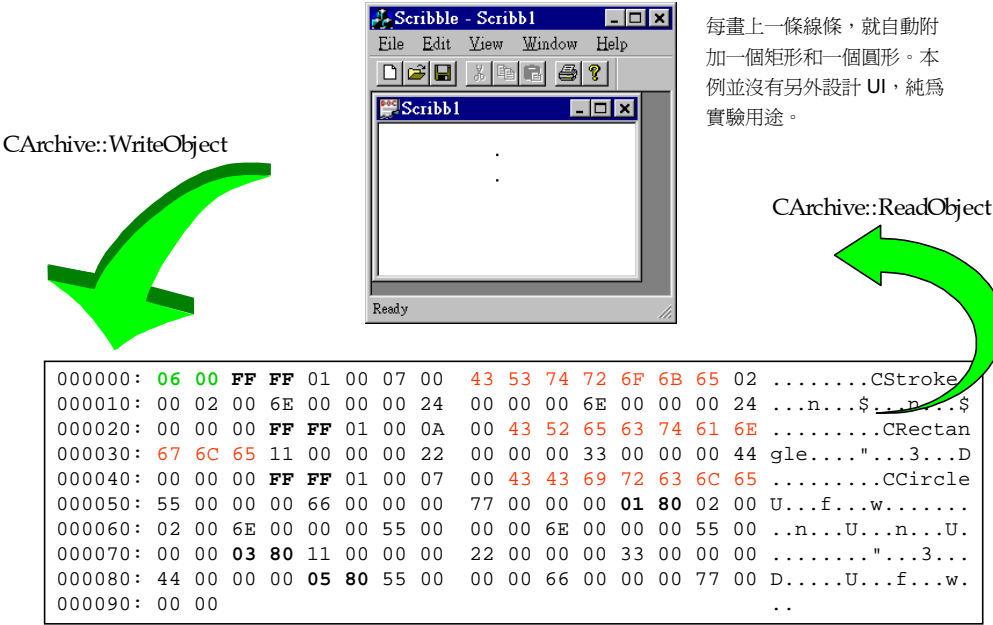
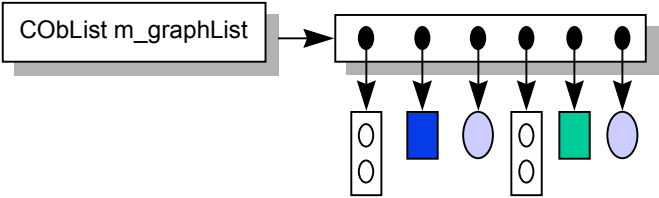


圖 8-10a TEST.SCB 檔案內容，檔案全長 146 個位元組。



每次滑鼠左鍵按下，開始一條線條，圖 8-10a 中的程式立刻 new 一個四方形和一個圓形，並和線條一起加入 COBList 之中，然後才開始接受左鍵的座標。所以圖 8-10a 的執行畫面造成本圖的資料結構。

數值 (hex)	說明
0006	表示此檔有六個 <i>COBList</i> 元素
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼
0007	表示後面接著的「類別名稱」有 7 個字元
43 53 74 72 6F 6B 65	"CStroke" (類別名稱) 的 ASCII 碼
0002	第一條線條的寬度
0002	第一條線條的點陣列大小 (點數)
00000066,0000001B	第一條線條的第一個點座標
00000066,0000001B	第一條線條的第二個點座標
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼。
000A	後面接著的「類別名稱」有 Ah 個字元。
43 52 65 63 74 61 6E 67 6C 65	"CRectangle" (類別名稱) 的 ASCII 碼。
00000011	第一個四方形的左
00000022	第一個四方形的上
00000033	第一個四方形的右
00000044	第一個四方形的下
FFFF	FFFF 亦即 -1，表示 New Class Tag
0001	這是 Schema no.，代表資料的版本號碼。
0007	後面接著的「類別名稱」有 7 個字元。
43 43 69 72 63 6C 65	"CCircle" (類別名稱) 的 ASCII 碼。
00000055	第一個圓形的中心點 X 座標
00000066	第一個圓形的中心點 Y 座標
00000077	第一個圓形的半徑
8001	這是 (<i>wOldClassTag</i> <i>nClassIndex</i>) 的組合結果，表示接下來的物件使用索引為 1 的舊類別。
0002	第二條線條的寬度
0002	第二條線條的點陣列大小 (點數)

數值 (hex)	說明
00000066,00000031	第二條線條的第一個點座標
▼ 00000066,00000031	第二條線條的第二個點座標
8003	這是 (<i>wOldClassTag</i> <i>nClassIndex</i>) 的組合結果，表示接下來的物件使用索引為 3 的舊類別。
00000011	第二個四方形的左
00000022	第二個四方形的上
00000033	第二個四方形的右
00000044	第二個四方形的下
8005	這是 (<i>wOldClassTag</i> <i>nClassIndex</i>) 的組合結果，表示接下來的物件使用索引為 5 的舊類別。
00000055	第二個圓形的中心點 X 座標
00000066	第二個圓形的中心點 Y 座標
00000077	第二個圓形的半徑

圖 8-10b TEST.SCB 檔案內容剖析。別忘了 Intel 採用 "little-endian" 位元組排列方式，每一字組的前後位元組係顛倒放置。本圖已將之擺正。

Document 與 View 之流 - Step4 做準備

雖然 Scribble Step1 已經可以正常工作，有些地方仍值得改進。

在一個子視窗上作畫，然後選按【Window/New Window】，會蹦出一個新的子視窗，內有第一個子視窗的圖形，同時，第一個子視窗的標題加上 :1 字樣，第二個子視窗的標題則有 :2 字樣。這是 Document/View 架構帶給我們的禮物，換句話說，想以多個視窗觀察同一份資料，程式員不必負擔什麼任務。但是，如果此後使用者在其中一個子視窗上作畫而不縮放視窗尺寸的話（也就是沒有產生 *WM_PAINT*），另一個子視窗內看不到新的繪圖內容：



這不是好現象！一體的兩面怎麼可以不一致呢？！

那麼，讓「作用中的 View 視窗」以訊息通知隸屬同一份 Document 的其他「兄弟視窗」，是不是就可以解決這個問題？是的，而且 Framework 已經把這樣的機制埋伏下去了。

CView 之中的三個虛擬函式：

- *CView::OnInitialUpdate* - 負責 View 的初始化。
- *CView::OnUpdate* - 當 Framework 呼叫此函式，表示 Document 的內容已有變化。
- *CView::OnDraw* - Framework 將在 *WM_PAINT* 發生後，呼叫此函式。此函式應負責更新 View 視窗的內容。

這些函式往往成為程式員改寫的目標。Scribble 第一版就是因為只改寫了其中的 *OnDraw* 函式，所以才有「多個 View 視窗不能同步更新」的缺失。想要改善這項缺失，我們必須改寫 *OnUpdate*。

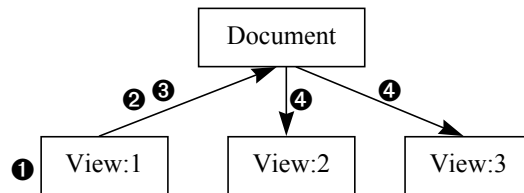
讓所有的 View 視窗「同步」更新資料的關鍵在於兩個函式：

- *CDocument::UpdateAllViews* - 如果這個函式執行起來，它會巡訪所有隸屬同一 Document 的各個 Views，找到一個就通知一個，而所謂「通知」就是呼叫 View 的 *OnUpdate* 函式。

- *CView::OnUpdate* - 這是一個虛擬函式，我們可以改寫它，在其中設計繪圖動作，也許全部重繪（這比較笨一點），也許想辦法只繪必要的一小部份（這樣速度比較快，但設計上比較複雜些）。

因此，當一個 Document 的資料改變時，我們應該設法呼叫其 *UpdateAllViews*，通知所有的 Views。什麼時候 Scribble 的資料會改變？答案是滑鼠左鍵按下時！所以你可能猜測到，我打算在 *CView::OnLButtonDown* 內呼叫 *CDocument::UpdateAllViews*。這個猜測的立論點是對的而結果是錯的，Scribble Step4 的作法是在 *CView::OnButtonUp* 內部呼叫它。

CView::OnUpdate 被呼叫，代表著 View 被告知：「嘿，Document 的內容已經改變了，請你準備修改你的顯示畫面」。如果你想節省力氣，利用 *Invalidate(TRUE)* 把視窗整個設為重繪區（無效區）並產生 *WM_PAINT*，再讓 *CView::OnDraw* 去傷腦筋算了。但是全部重繪的效率低落，程式看起來很笨拙，Step4 將有比較精緻的作法。



- 1 使用者在 View:1 做動作（View 扮演使用者界面的第一線）。
- 2 View:1 呼叫 *GetDocument*，取得 Document 指標，更改資料內容。
- 3 View:1 呼叫 Document 的 *UpdateAllViews*。
- 4 View:2 和 View:3 的 *OnUpdate* ——被呼叫起來，這是更新畫面的時機。

圖 8-11 假設一份 Document 聯結了三個 Views

注意：在 MFC 手冊或其他書籍中，你可能會看到像「View1 以訊息通知 Document」或「Document 以訊息通知 View2、View3」的說法。這裡所謂的「訊息」是物件導向學術界的術語，不要和 Windows 的訊息混淆了。事實上整個過程中並沒有任何一個 Windows 訊息參與其中。

訊息映射與命令繞行

Message Mapping and Command Routing

訊息映射機制與命令繞行，活像是米諾托斯的迷宮，
是 MFC 最曲折幽深的神秘地帶。

你已經從前一章中徹底了解了 MFC 程式極端重要的 Document/View 架構。本章的重點有兩個，第一個是修改程式的人機介面，增添選單項目和工具列按鈕。這一部份藉 Visual C++ 工具之助，非常簡單，但是我們往往不知道該在程式的什麼地方（哪一個類別之中）處理來自選單和工具列的訊息（也就是 `WM_COMMAND` 訊息）。本章第二個重點就是要解決這個迷惑，我將對所謂的訊息映射（Message Map）和命令繞行（Command Routing）機制做深入的討論。這兩個機制宛如 MFC 最曲折幽深的神秘地帶，是把雜亂無章的 Windows API 函式和 Windows 訊息物件導向化的大功臣。

到底要解決什麼

Windows 程式的本質係藉著訊息來維持脈動。每一個訊息都有一個代碼，並以 `WM_` 開頭的常數表示之。訊息在傳統 SDK 程式方法中的流動以及處置方式，在第 1 章已經交待得很清楚。

各種訊息之中，來自選單或工具列者，都以 `WM_COMMAND` 表示，所以這一類訊息我們又稱之為命令訊息（Command Message），其 `wParam` 記錄著此一訊息來自哪一個選單項目。

除了命令訊息，還有一種訊息也比較特殊，出現在對話盒函式中，是控制元件（controls）傳送給父視窗（即對話盒）的訊息。雖然它們也是以 `WM_COMMAND` 為外衣，但特別歸類為「notification 訊息」。

注意：Windows 95 新的控制元件（所謂的 common controls）不再傳送 `WM_COMMAND` 訊息給對話盒，而是送 `WM_NOTIFY`。這樣就不會糾纏不清了。但為了回溯相容，舊有的控制元件（如 edit、list box、combo box...）都還是傳送 `WM_COMMAND`。

訊息會循著 Application Framework 規定的路線，遊走於各個物件之間，直到找到它的依歸（訊息處理函式）。找不到的話，Framework 最終就把它交給 `::DefWindowProc` 函式去處理。

但願你記憶猶新，第 6 章曾經挖掘 MFC 原始碼，得知 MFC 在為我們產生視窗之前，如果我所指定的視窗類別是 NULL，MFC 會自動先註冊一個適當的視窗類別。這個類別在動態聯結、除錯版、非 Unicode 環境的情況下，可能是下列五種視窗類別之一：

- "AfxWnd42d"
- "AfxControlBar42d"
- "AfxMDIFrame42d"
- "AfxFrameOrView42d"
- `_AFXOLECONTROL42D_`

每一個視窗類別有它自己的視窗函式。根據 SDK 的基礎，我們推想，不同視窗所獲得的訊息，應該由不同的視窗函式來處理。如果都沒有能夠處理，最後再交由 Windows API 函式 `::DefWindowProc` 處理。

這是很直覺的想法，而且對於一般訊息（如 `WM_MOVE`、`WM_SIZE`、`WM_CREATE` 等）也是天經地義的。但是今天 Application Framework 比傳統的 SDK 程式多出了一個 Document/View 架構，試想，如果選單上有個命令項關乎文件的處理，那麼讓這個命令

訊息流到 `Document` 類別去不是最理想嗎？一旦流入 `Document` 大本營，我們（程式員）就可以很方便地取得 `Document` 成員變數、呼叫 `Document` 成員函式，做愛做的事。

但是 `Document` 不是視窗，也沒有對應的視窗類別，怎麼讓訊息能夠七拐八彎地流往 `Document` 類別去？甚至更往上流向 `Application` 類別去？這就是所謂的命令繞行機制！而為了讓訊息的流動有線路可循，MFC 必須做出一個巨大的網，實現所有可能的路線，這個網就是所謂的訊息映射地圖（`Message map`）。最後，MFC 還得實現一個訊息推動引擎，讓訊息依 `Framework` 的意旨前進，該拐的時候拐，該彎的時候彎，這個邦浦機制埋藏在各個類別的 `WindowProc`、`OnCommand`、`OnCmdMsg`、`DefWindowProc` 虛擬函式中。

沒有命令繞行機制，`Document/View` 架構就像斷了條胳臂，會少掉許多功用。

很快你就會看到所有的秘密。很快地，它們統統不再對你構成神秘。

訊息分類

Windows 的訊息都是以 `WM_XXX` 為名，`WM_` 的意思是 "Windows Message"。訊息可以是來自硬體的「輸入訊息」，例如 `WM_LBUTTONDOWN`，也可以是來自 `USER` 模組的「視窗管理訊息」，例如 `WM_CREATE`。這些訊息在 MFC 程式中都是隱晦的（我的意思是不像在 SDK 程式中那般顯明），我們不必在 MFC 程式中撰寫 `switch case` 指令，不必一一識別並處理由系統送過來的訊息；所有訊息都將依循 `Framework` 制定的路線，並參照路中是否有攔路虎（你的訊息映射表格）而流動。`WM_PAINT` 一定流往你的 `OnPaint` 函式去，`WM_SIZE` 一定流往你的 `OnSize` 函式去。

所有的訊息在 MFC 程式中都是暗潮洶湧，但是表面無波。

MFC 把訊息分為三大類：

- 命令訊息（`WM_COMMAND`）：命令訊息意味著「使用者命令程式做某些動作」。凡由 UI 物件產生的訊息都是這種命令訊息，可能來自選單或加速鍵或工具列

按鈕，並且都以 `WM_COMMAND` 呈現。如何分辨來自各處的命令訊息？SDK 程式主要靠訊息的 `wParam` 辨識之，MFC 程式則主要靠選單項目的識別碼（menu ID）辨識之 -- 兩者其實是相同的。

什麼樣的類別有資格接受命令訊息？凡衍生自 `CCmdTarget` 之類別，皆有資格。從 `command target` 的字面意義可知，這是命令訊息的目的地。也就是說，凡衍生自 `CCmdTarget` 者，它的骨子裡就有了一種特殊的機制。把整張 MFC 類別階層圖攤開來看，幾乎建構應用程式的最重要的幾個類別都衍生自 `CCmdTarget`，剩下的不能接收訊息的，是像 `CFile`、`CArchive`、`CPoint`、`CDao`（資料庫）、Collection Classes（純粹資料處理）、GDI 等等「非主流」類別。

- 標準訊息 - 除 `WM_COMMAND` 之外，任何以 `WM_` 開頭的都算是這一類。任何衍生自 `CWnd` 之類別，均可接收此訊息。
- Control Notification - 這種訊息由控制元件產生，為的是向其父視窗（通常是對話盒）通知某種情況。例如當你在 `ListBox` 上選擇其中一個項目，`ListBox` 就會產生 `LBN_SELCHANGE` 傳送給父視窗。這類訊息也是以 `WM_COMMAND` 形式呈現。

訊息流歸宗 Command Target (`CCmdTarget`)

你可以在程式的許多類別之中設計攔路虎（我是指「訊息映射表格」），接收並處理訊息。只要是 `CWnd` 衍生類別，就可以攔下任何 Windows 訊息。與視窗無關的 MFC 類別（例如 `CDocument` 和 `CWinApp`）如果也想處理訊息，必須衍生自 `CCmdTarget`，並且只可能收到 `WM_COMMAND` 命令訊息。

會產生命令訊息的，不外就是 UI 物件：選單項目和工具列按鈕都是。命令訊息必須有一個對應的處理函式，把訊息和其處理函式「綁」在一塊兒，這動作稱為 `Command Binding`，這個動作將由一堆巨集完成。通常我們不直接手工完成這些巨集內容，也就是說我們並不以文字編輯器一行一行地撰寫相關的碼，而是藉助於 `ClassWizard`。

一個 `Command Target` 物件如何知道它可以處理某個訊息？答案是它會看看自己的訊息映

射表。訊息映射表使得訊息和函式的對映關係形成一份表格，進而全體形成一張網，當 Command Target 物件收到某個訊息，便可由表格得知其處理函式的名稱。

三個奇怪的巨集，一張巨大的網

早在本書第1章我就介紹過訊息映射的雛形了，不過那是小把戲，不登大雅之堂。第3章以 DOS 程式模擬訊息映射，就頗有可觀之處，因為那是「偷」MFC 的原始碼完成的，可以說具體而微。

試著思考這個問題：C++ 的繼承與多型性質，使衍生類別與基礎類別的成員函式之間有著特殊的關聯。但這當中並沒有牽扯到 Windows 訊息。的確，C++ 語言完全沒有考慮 Windows 訊息這一回事（那當然）。如何讓 Windows 訊息也能夠在物件導向以及繼承性質中扮演一個角色？既然語言沒有支援，只好自求多福了。訊息映射機制的三個相關巨集就是 MFC 自求多福的結果。

「訊息映射」是 MFC 內建的一個訊息分派機制，只要利用數個巨集以及固定形式的寫法，類似填表格，就可以讓 Framework 知道，一旦訊息發生，該循哪一條路遞送。每一個類別只能擁有一個訊息映射表格，但也可以沒有。下面是 Scribble Document 建立訊息映射表的動作：

- 首先你必須在類別宣告檔（.H）聲明擁有訊息映射表格：

```
class CScribbleDoc : public CDocument
{
    ...
    DECLARE_MESSAGE_MAP()
};
```

- 然後在類別實作檔（.CPP）實現此一表格：

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
   //{{AFX_MSG_MAP(CScribbleDoc)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
    ...
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

這其中出現三個巨集。第一個巨集 *BEGIN_MESSAGE_MAP* 有兩個參數，分別是擁有此訊息映射表之類別，及其父類別。第二個巨集是 *ON_COMMAND*，指定命令訊息的處理函式名稱。第三個巨集 *END_MESSAGE_MAP* 作為結尾記號。至於夾在 *BEGIN_* 和 *END_* 之中奇奇怪怪的說明符號 *//}* 和 *//{*，是 ClassWizard 產生的，也是用來給它自己看的。記住，前面我就說了，很少人會自己親手鍵入每一行碼，因為 ClassWizard 的表現相當不俗。

夾在 *BEGIN_* 和 *END_* 之中的巨集，除了 *ON_COMMAND*，還可以有許多種。標準的 Windows 訊息並不需要由我們指定處理函式的名稱。標準訊息的處理函式，其名稱也是「標準」的（預設的），像是：

巨集名稱	對映訊息	訊息處理函式
ON_WM_CHAR	WM_CHAR	OnChar
ON_WM_CLOSE	WM_CLOSE	OnClose
ON_WM_CREATE	WM_CREATE	OnCreate
ON_WM_DESTROY	WM_DESTROY	OnDestroy
ON_WM_LBUTTONDOWN	WM_LBUTTONDOWN	OnLButtonDown
ON_WM_LBUTTONUP	WM_LBUTTONUP	OnLButtonUp
ON_WM_MOUSEMOVE	WM_MOUSEMOVE	OnMouseMove
ON_WM_PAINT	WM_PAINT	OnPaint
...		

DECLARE_MESSAGE_MAP 巨集

訊息映射的本質其實是一個巨大的資料結構，用來為諸如 *WM_PAINT* 這樣的標準訊息決定流動路線，使它得以流到父類別去；也用來為 *WM_COMMAND* 這個特殊訊息決定流動路線，使它能夠七拐八彎地流到類別階層結構的旁支去。

觀察機密的最好方法就是挖掘原始碼：

```
// in AFXWIN.H
#define DECLARE_MESSAGE_MAP() \
private: \
    static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
    static AFX_DATA const AFX_MSGMAP messageMap; \
    virtual const AFX_MSGMAP* GetMessageMap() const; \
```

注意：static 修飾詞限制了資料的配置，使得每個「類別」僅有一份資料，而不是每一個「物件」各有一份資料。

我們看到兩個陌生的型態：*AFX_MSGMAP_ENTRY* 和 *AFX_MSGMAP*。繼續挖原始碼，發現前者是一個 *struct*：

```
// in AFXWIN.H
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;    // windows message
    UINT nCode;       // control code or WM_NOTIFY code
    UINT nID;         // control ID (or 0 for windows messages)
    UINT nLastID;     // used for entries specifying a range of control id's
    UINT nSig;        // signature type (action) or pointer to message #
    AFX_PMSG pfn;     // routine to call (or special value)
};
```

很明顯你可以看出它的最主要作用，就是讓訊息 *nMessage* 對應於函式 *pfn*。其中 *pfn* 的資料型態 *AFX_PMSG* 被定義為一個函式指標：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

出現在 *DECLARE_MESSAGE_MAP* 巨集中的另一個 *struct*，*AFX_MSGMAP*，定義如下：

```
// in AFXWIN.H
struct AFX_MSGMAP
{
    const AFX_MSGMAP* pBaseMap;
    const AFX_MSGMAP_ENTRY* lpEntries;
};
```

其中 *pBaseMap* 是一個指向「基礎類別之訊息映射表」的指標，它提供了一個走訪整個繼承串鏈的方法，有效地實作出訊息映射的繼承性。衍生類別將自動地「繼承」其基礎

類別中所處理的訊息，意思是，如果基礎類別處理過 A 訊息，其衍生類別即使未設計 A 訊息之訊息映射表項目，也具有對 A 訊息的處理能力。當然啦，衍生類別也可以針對 A 訊息設計自己的訊息映射表項。

喝，真像虛擬函式！但 Message Map 沒有虛擬函式所帶來的巨大的 overhead（額外負擔）

透過 `DECLARE_MESSAGE_MAP` 這麼簡單的一個巨集，相當於為類別宣告了圖 9-1 的資料型態。注意，只是宣告而已，還沒有真正的實體。

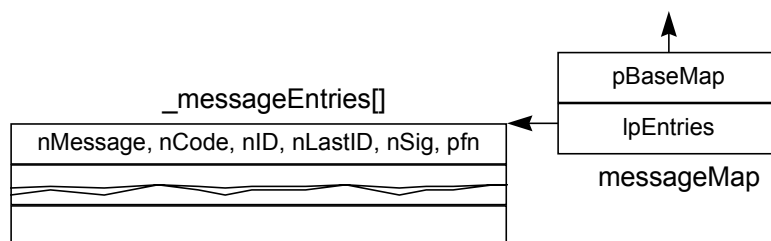


圖 9-1 `DECLARE_MESSAGE_MAP` 巨集相當於宣告了這樣的資料結構。

訊息映射表的形成：BEGIN, ON, END, 巨集

前置準備工作完成了，接下來的課題是如何實現並填充圖 9-1 的資料結構內容。當然你馬上就猜到了，使用的是另一組巨集：

```

BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_PAINT()
    ON_WM_CREATE()
    ...
END_MESSAGE_MAP()
    
```

奧秘還是在原始碼中：

// 以下原始碼在 AFXWIN.H

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
    const AFX_MSGMAP* theClass::GetMessageMap() const \
    { return &theClass::messageMap; } \
    AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
    { &baseClass::messageMap, &theClass::_messageEntries[0] }; \
    const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
    { \

#define END_MESSAGE_MAP() \
    {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
}; \
```

注意：AfxSig_end 在 AFXMSG.H 中被定義為 0

// 以下原始碼在 AFXMSG.H

```
#define ON_COMMAND(id, memberFxn) \
    { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },

#define ON_WM_CREATE() \
    { WM_CREATE, 0, 0, 0, AfxSig_is, \
      (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
#define ON_WM_DESTROY() \
    { WM_DESTROY, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnDestroy },
#define ON_WM_MOVE() \
    { WM_MOVE, 0, 0, 0, AfxSig_vvii, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(int, int))OnMove },
#define ON_WM_SIZE() \
    { WM_SIZE, 0, 0, 0, AfxSig_vwii, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, int, int))OnSize },
#define ON_WM_ACTIVATE() \
    { WM_ACTIVATE, 0, 0, 0, AfxSig_vwWb, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, CWnd*, \
      BOOL))OnActivate },
#define ON_WM_SETFOCUS() \
    { WM_SETFOCUS, 0, 0, 0, AfxSig_vW, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(CWnd*))OnSetFocus },
#define ON_WM_PAINT() \
    { WM_PAINT, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnPaint },
#define ON_WM_CLOSE() \
    { WM_CLOSE, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnClose },
...
```

於是，這樣的巨集：

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_CREATE()
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

便被展開成為這樣的碼：

```
const AFX_MSGMAP* CMyView::GetMessageMap() const
{ return &CMyView::messageMap; }
AFX_DATADEF const AFX_MSGMAP CMyView::messageMap =
{ &CView::messageMap, &CMyView::_messageEntries[0] };
const AFX_MSGMAP_ENTRY CMyView::_messageEntries[] =
{
    { WM_CREATE, 0, 0, 0, AfxSig_is, \
      (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
    { WM_PAINT, 0, 0, 0, AfxSig_vv, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(void))OnPaint },
    { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
};
```

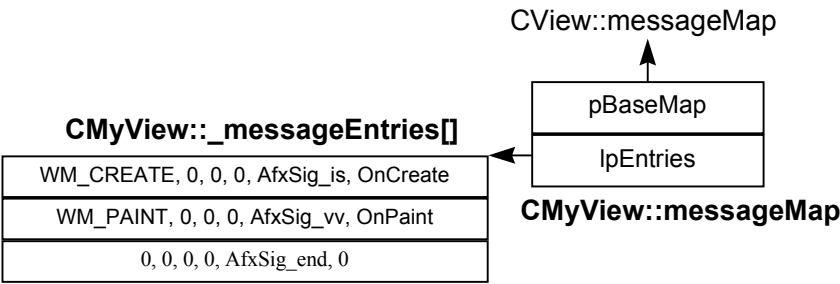
其中 *AFX_DATADEF* 和 *AFX_MSG_CALL* 又是兩個看起來很奇怪的常數。你可以在兩個檔案中找到它們的定義：

```
// in \DEVSTUDIO\VC\MFC\INCLUDE\AFXVER_.H
#define AFX_DATA
#define AFX_DATADEF

// in \DEVSTUDIO\VC\MFC\INCLUDE\AFXWIN.H
#define AFX_MSG_CALL
```

顯然它們就像 *afx_msg* 一樣（我曾經在第 6 章的 *HelppMFC* 原始碼一出現之後解釋過），都只是個 "intentional placeholder"（刻意保留的空間），可能在將來會用到，目前則為「無物」。

以圖表示 *BEGIN_* *KON_* *END_* *K* 巨集的結果為：



注意：圖中的 *AfxSig_vv* 和 *AfxSig_is* 都代表簽名符號 (Signature)。這些常數在 *AFXMSG_H* 中定義，稍後再述。

前面我說過了，所有能夠接收訊息的類別，都應該衍生自 *CCmdTarget*。那麼我們這麼推論應該是合情合理的：每一個衍生自 *CCmdTarget* 的類別都應該有 *DECLARE_* *BEGIN_* *END_* 巨集組？

唔，錯了，*CWinThread* 就沒有！

可是這麼一來，*CWinApp* 通往 *CCmdTarget* 的路徑不就斷掉了嗎？呵呵，難道 *CWinApp* 不能跳過 *CWinThread* 直接連上 *CCmdTarget* 嗎？看看下面的 MFC 原始碼：

```
// in AFXWIN.H
class CWinApp : public CWinThread
{
...
    DECLARE_MESSAGE_MAP()
};

// in APPCORE.CPP
BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget) // 注意第二個參數是 CCmdTarget，
// {AFX_MSG_MAP(CWinApp)                // 而不是 CWinThread。
// Global File commands
```

```
ON_COMMAND(ID_APP_EXIT, OnAppExit)
// MRU - most recently used file menu
ON_UPDATE_COMMAND_UI(ID_FILE_MRU_FILE1, OnUpdateRecentFileMenu)
ON_COMMAND_EX_RANGE(ID_FILE_MRU_FILE1, ID_FILE_MRU_FILE16, OnOpenRecentFile)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

讓我們看看具體的情況。圖 9-2 就是 MFC 的訊息映射表。當你的衍生類別使用了 *DECLARE_BEGIN_END* 巨集，你也就把自己的訊息映射表掛上去了 -- 當然是掛在尾端。

如果沒有把 *BEGIN_MESSAGE_MAP* 巨集中的兩個參數（也就是類別本身及其父類別的名稱）按照規矩來寫，可能會發生什麼結果呢？訊息可能在不應該流向某個類別時流了過去，在應該被處理時卻又跳離了。總之，完美的機制有了破綻。程式沒當掉算你幸運！

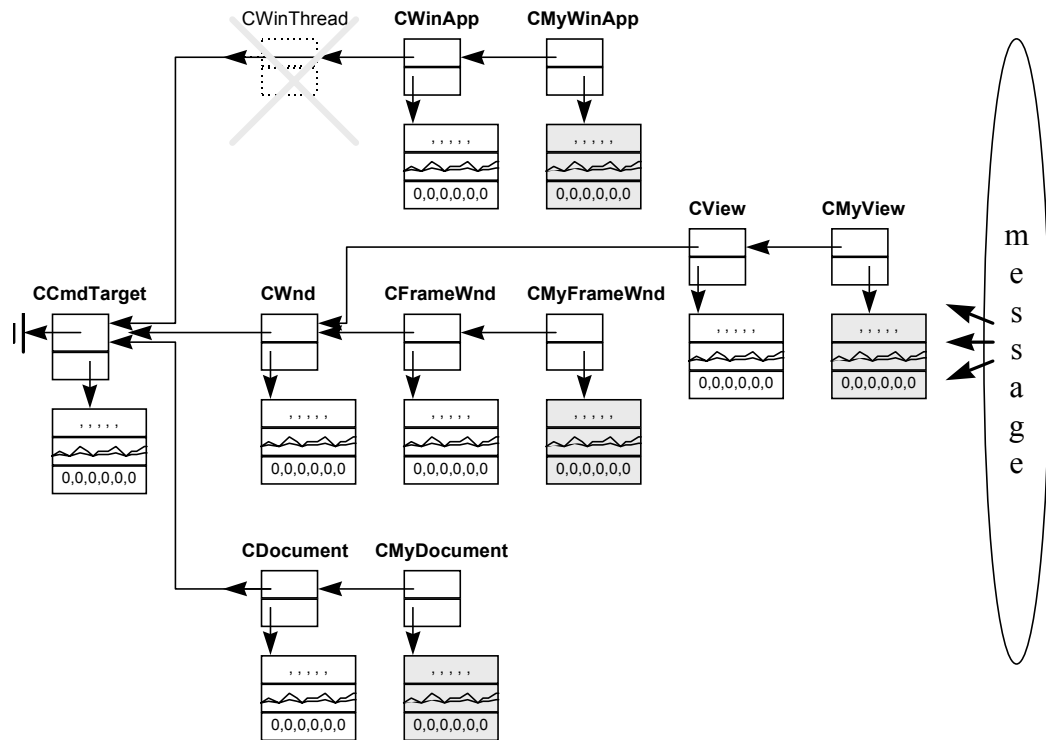


圖 9-2 MFC 訊息映射表（也就是訊息流動網）

我們終於了解，Message Map 既可說是一套巨集，也可以說是巨集展開後所代表的一套資料結構；甚至也可以說 Message Map 是一種動作，這個動作，就是在剛剛所提的資料結構中尋找與訊息相吻合的項目，從而獲得訊息的處理常式的函式指標。

雖然，C++ 程式員看到多型（Polymorphism），直覺的反應就是虛擬函式，但請注意，各個 Message Map 中的各個同名函式雖有多型的味道，卻不是虛擬函式。乍想之下使用虛擬函式是合理的：你產生一個與視窗有關的 C++ 類別，然後為此視窗所可能接收的任何訊息都提供一個對應的虛擬函式。這的確散發著 C++ 的味道和物件導向的精神，但現實與理想之間總是有些距離。

要知道，虛擬函式必須經由一個虛擬函式表（virtual function table，vtable）實作出來，每一個子類別必須有它自己的虛擬函式表，其內至少有父類別之虛擬函式表的內容複本（請參考第 2 章「類別與物件大解剖」一節）。好哇，虛擬函式表中的每一個項目都是一個函式指標，價值 4 位元組，如果基礎類別的虛擬函式表有 100 個項目，經過 10 層繼承，開枝散葉，總共需耗費多少記憶體在其中？最終，系統會被巨大的額外負擔（overhead）拖垮！

這就是為什麼 MFC 採用獨特的訊息映射機制而不採用虛擬函式的原因。

米諾托斯（Minotaurus）與忒修斯（Theseus）

截至目前我還有一些細節沒有交待清楚，像是訊息的比對動作、訊息處理常式的呼叫動作、以及參數的傳遞等等，但至少現在可以先繼續進行下去，我的目標瞄準訊息唧筒（叫邦浦也可以啦）。

視窗接收訊息後，是誰把訊息唧進訊息映射網中？是誰決定訊息該直直往父映射表走去？還是拐向另一條路（請回頭看看圖 9-2）？訊息的繞行路線，以及 MFC 的訊息唧筒的設計，活像是米諾托斯的迷宮。不過別擔心，我將扮演忒修斯，讓你免遭毒手。

米諾托斯（Minotaurus），希臘神話裡牛頭人身的怪獸，為克里特島國王邁諾斯之妻所生。邁諾斯造迷宮將米諾托斯藏於其中，每有人誤入迷宮即遭吞噬。怪獸後為雅典王子忒修斯（Theseus）所殺。

MFC 2.5（注意，是 2.5 而非 4.x）曾經在 *WinMain* 的第一個重要動作 *AfxWinInit* 之中，自動為程式註冊四個 Windows 視窗類別，並且把視窗函式一致設為 *AfxWndProc*：

```

//in APPINIT.CPP (MFC 2.5)
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine, int nCmdShow)
{
    ...
    // register basic WndClasses (以下開始註冊視窗類別)
    WNDCLASS wndcls;
    wndcls.lpfWndProc = AfxWndProc;

    // Child windows - no brush, no icon, safest default class styles
    ...
    wndcls.lpszClassName = _afxWnd;
    ❶ if (!::RegisterClass(&wndcls))
        return FALSE;

    // Control bar windows
    ...
    wndcls.lpszClassName = _afxWndControlBar;
    ❷ if (!::RegisterClass(&wndcls))
        return FALSE;

    // MDI Frame window (also used for splitter window)
    ...
    ❸ if (!RegisterWithIcon(&wndcls, _afxWndMDIFrame, AFX_IDI_STD_MDIFRAME))
        return FALSE;

    // SDI Frame or MDI Child windows or views - normal colors
    ...
    ❹ if (!RegisterWithIcon(&wndcls, _afxWndFrameOrView, AFX_IDI_STD_FRAME))
        return FALSE;
    ...
}

```

下面是 *AfxWndProc* 的內容：

```

// in WINCORE.CPP (MFC 2.5)
LRESULT CALLBACK AFX_EXPORT
AfxWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    CWnd* pWnd;

    pWnd = CWnd::FromHandlePermanent(hWnd);
    ASSERT(pWnd != NULL);
    ASSERT(pWnd->m_hWnd == hWnd);

```



```

LRESULT lResult = _AfxCallWndProc(pWnd, hWnd, message, wParam, lParam);
return lResult;
}

// Official way to send message to a CWnd
LRESULT PASCAL _AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
    LRESULT lResult;
    ...
    TRY
    {
        ...
        lResult = pWnd->WindowProc(message, wParam, lParam);
    }
    ...
    return lResult;
}

```

MFC 2.5 的 *CWinApp::Run* 呼叫 *PumpMessage*，後者又呼叫 *::DispatchMessage*，把訊息源推往 *AfxWndProc*（如上），最後流向 *pWnd->WindowProc* 去。拿 SDK 程式的本質來做比對，這樣的邏輯十分容易明白。

MFC 4.x 仍舊使用 *AfxWndProc* 作為訊息唧筒的起點，但其間卻隱藏了許多關節。

但願你記憶猶新，第 6 章曾經說過，MFC 4.x 適時地為我們註冊 Windows 視窗類別（在第一次產生該種型式之視窗之前）。這些個 Windows 視窗類別的視窗函式各是「視窗所對應之 C++ 類別中的 *DefWindowProc* 成員函式」，請參考第 4 章「**CFrameWnd::Create 產生主視窗**」一節。這就和 MFC 2.5 的作法（所有視窗類別共用同一個視窗函式）有了明顯的差異。那麼，推動訊息的心臟，也就是 *CWinThread::PumpMessage* 中呼叫的 *::DispatchMessage*（請參考第 4 章「**CWinApp::Run 程式生命的活水源頭**」一節），照說應該把訊息唧到對應之 C++ 類別的 *DefWindowProc* 成員函式去。但是，我們發現 MFC 4.x 中仍然保有和 MFC 2.5 相同的 *AfxWndProc*，仍然保有 *AfxCallWndProc*，而且它們扮演的角色也沒有變。

事實上，MFC 4.x 利用 hook，把看似無關的動作全牽聯起來了。所謂 hook，是 Windows 程式設計中的一種高階技術。通常訊息都是停留在訊息佇列中等待被所隸屬之視窗抓取，如果你設立 hook，就可以更早一步抓取訊息，並且可以抓取不屬於你的訊息，送往你設定的一個所謂「濾網函式 (filter)」。

請查閱 Win32 API 手冊中有關於 *SetWindowsHook* 和 *SetWindowsHookEx* 兩函式，以獲得更多的 hook 資訊。（可參考 *Windows 95: A Developer's Guide* 一書第 6 章 *Hooks*）

MFC 4.x 的 hook 動作是在每一個 *CWnd* 衍生類別之物件產生之際發生，步驟如下：

```
// in WINCORE.CPP (MFC 4.x)
// 請回顧第 6 章「CFrameWnd::Create 產生主視窗」一節
BOOL CWnd::CreateEx(...)
{
    ...
    PreCreateWindow(cs); //第 6 章曾經詳細討論過此一函式。
    AfxHookWindowCreate(this);
    HWND hWnd = ::CreateWindowEx(...);
    ...
}

// in WINCORE.CPP (MFC 4.x)
void AFXAPI AfxHookWindowCreate(CWnd* pWnd)
{
    ...
    pThreadState->m_hHookOldCbtFilter = ::SetWindowsHookEx(WH_CBT,
        _AfxCbtFilterHook, NULL, ::GetCurrentThreadId());
    ...
}
```

WH_CBT 是眾多 hook 型態中的一種，意味著安裝一個 Computer-Based Training (CBT) 濾網函式。安裝之後，Windows 系統在進行以下任何一個動作之前，會先呼叫你的濾網函式：

- 令一個視窗成為作用中的視窗 (*HCBT_ACTIVATE*)
- 產生或摧毀一個視窗 (*HCBT_CREATEWND*、*HCBT_DESTROYWND*)
- 最大化或最小化一個視窗 (*HCBT_MINMAX*)

- 搬移或縮放一個視窗 (*HCBT_MOVESIZE*)
- 完成一個來自系統選單的系統命令 (*HCBT_SYSTEMCOMMAND*)
- 從系統佇列中移去一個滑鼠或鍵盤訊息 (*HCBT_KEYSKIPPED*、*HCBT_CLICKSKIPPED*)

因此，經過上述 hook 安裝之後，任何視窗即將產生之前，濾網函式 *_AfxCbtFilterHook* 一定會先被呼叫：

```
_AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)
{
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    if (code != HCBT_CREATEWND)
    {
        // wait for HCBT_CREATEWND just pass others on...
        return CallNextHookEx(pThreadState->m_hHookOldCbtFilter, code,
                               wParam, lParam);
    }
    ...
    if (!afxData.bWin31)
    {
        // perform subclassing right away on Win32
        _AfxStandardSubclass((HWND)wParam);
    }
    else
    {
        ...
    }
    ...
    LRESULT lResult = CallNextHookEx(pThreadState->m_hHookOldCbtFilter, code,
                                     wParam, lParam);
    return lResult;
}

void AFXAPI _AfxStandardSubclass(HWND hWnd)
{
    ...
    // subclass the window with standard AfxWndProc
    oldWndProc = (WNDPROC)SetWindowLong(hWnd, GWL_WNDPROC,
                                         (DWORD)AfxGetAfxWndProc());
}

WNDPROC AFXAPI AfxGetAfxWndProc()
{

```

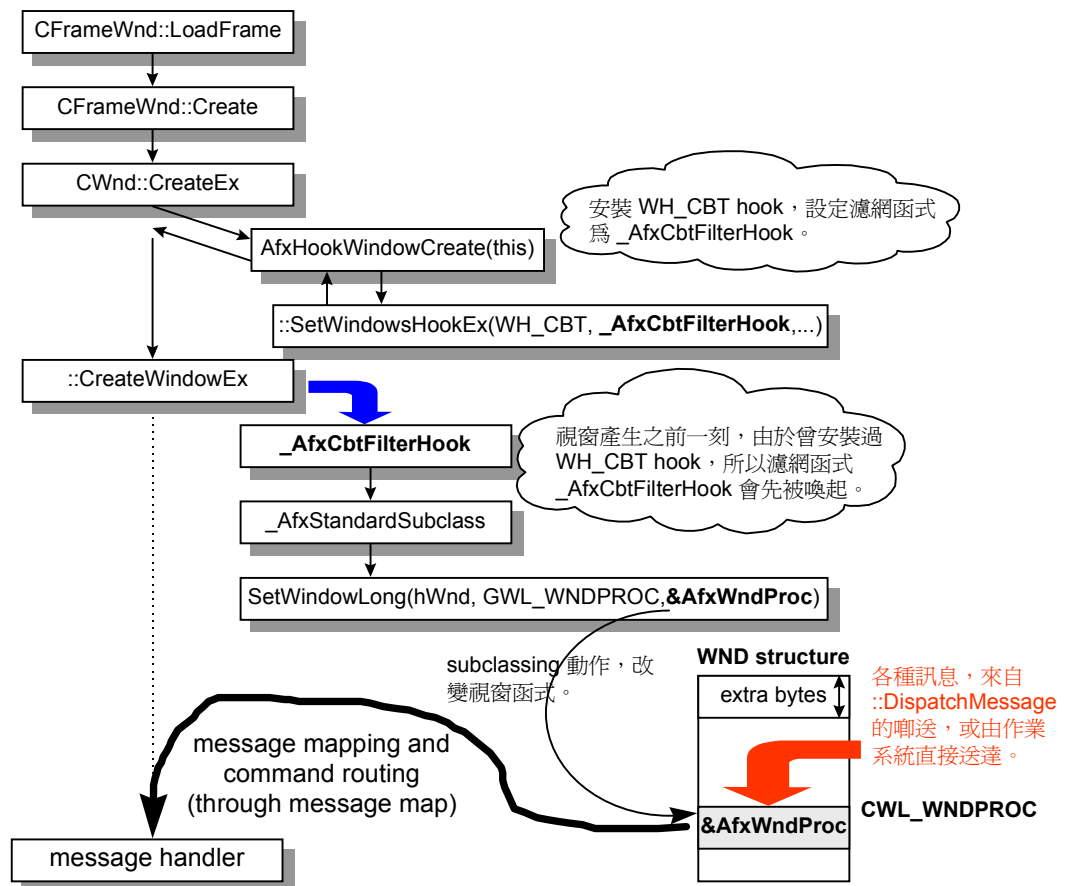
```

...
return &AfxWndProc;
}

```

啊，非常明顯，上面的函式合力做了偷天換日的勾當：把「視窗所屬之 Windows 視窗類別」中所記錄的視窗函式，改換為 *AfxWndProc*。於是，*::DispatchMessage* 就把訊息源源推往 *AfxWndProc* 去了。

這種看起來很迂迴又怪異的作法，是爲了包容新的 3D Controls(細節就容我省略了吧)，並與 MFC 2.5 相容。下圖把前述的 hook 和 subclassing 動作以流程圖顯示出來：



不能稍息，我們還沒有走出迷宮！*AfxWndProc* 只是訊息兩萬五千里長征的第一站！

兩萬五千里長征 - 訊息的流動

一個訊息從發生到被攫取，直至走向它的歸宿，是一條漫漫長路。上一節我們來到了漫漫長路的起頭 *AfxWndProc*，這一節我要帶你看看訊息實際上如何推動。

訊息的流動路線已隱隱有脈絡可尋，此脈絡是指由 *BEGIN_MESSAGE_MAP* 和 *END_MESSAGE_MAP* 以及許許多多 *ON_WM_XXX* 巨集所構成的訊息映射網。但是唧筒與方向盤是如何設計的？一切的線索還是要靠原始碼透露：

```
// in WINCORE.CPP (MFC 4.x)
LRESULT CALLBACK AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    ...
    // messages route through message map
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}

LRESULT AFXAPI AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
    WPARAM wParam = 0, LPARAM lParam = 0)
{
    ...
    // delegate to object's WindowProc
    lResult = pWnd->WindowProc(nMsg, wParam, lParam);
    ...
    return lResult;
}
```

整個 MFC 中，擁有虛擬函式 *WindowProc* 者包括 *CWnd*、*CControlBar*、*COleControl*、*COlePropertyPage*、*CDialog*、*CReflectorWnd*、*CParkingWnd*。一般視窗（例如 Frame 視窗、View 視窗）都衍生自 *CWnd*，所以讓我們看看 *CWnd::WindowProc*。這個函式相當於 C++ 中的視窗函式：

```
// in WINCORE.CPP (MFC 4.x)
LRESULT CWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
```


```

{
    // OnWndMsg does most of the work, except for DefWindowProc call
    LRESULT lResult = 0;
    if (!OnWndMsg(message, wParam, lParam, &lResult))
        lResult = DefWindowProc(message, wParam, lParam);
    return lResult;
}

LRESULT CWnd::DefWindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    if (m_pfnSuper != NULL)
        return ::CallWindowProc(m_pfnSuper, m_hWnd, nMsg, wParam, lParam);

    WNDPROC pfnWndProc;
    if ((pfnWndProc = *GetSuperWndProcAddr()) == NULL)
        return ::DefWindowProc(m_hWnd, nMsg, wParam, lParam);
    else
        return ::CallWindowProc(pfnWndProc, m_hWnd, nMsg, wParam, lParam);
}

```



直線「溯」（一般 Windows 訊息）

CWnd::WindowProc 呼叫的 *OnWndMsg* 是用來分辨並處理訊息的專職機構；如果是命令訊息，就交給 *OnCommand* 處理，如果是通告訊息（Notification），就交給 *OnNotify* 處理。*WM_ACTIVATE* 和 *WM_SETCURSOR* 也都有特定的處理函式。而一般的 Windows 訊息，就直接在訊息映射表中上溯，尋找其歸宿（訊息處理常式）。為什麼要特別區隔出命令訊息 *WM_COMMAND* 和通告訊息 *WM_NOTIFY* 兩類呢？因為它們的上溯路徑不是那麼單純地只往父類別去，它們可能需要拐個彎。

```

#0001 // in WINCORE.CPP (MFC 4.0)
#0002 BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)
#0003 {
#0004     LRESULT lResult = 0;
#0005
#0006     // special case for commands
#0007     if (message == WM_COMMAND)
#0008     {
#0009         OnCommand(wParam, lParam);
#0010         ...
#0011     }
#0012

```

```
#0013     // special case for notifies
#0014     if (message == WM_NOTIFY)
#0015     {
#0016         OnNotify(wParam, lParam, &lResult);
#0017         ...
#0018     }
#0019     ...
#0020     const AFX_MSGMAP* pMessageMap; pMessageMap = GetMessageMap();
#0021     UINT iHash; iHash = (LOWORD((DWORD)pMessageMap) ^ message) & (iHashMax-1);
#0022     AfxLockGlobals(CRIT_WINMSGCACHE);
#0023     AFX_MSG_CACHE msgCache; msgCache = _afxMsgCache[iHash];
#0024     AfxUnlockGlobals(CRIT_WINMSGCACHE);
#0025
#0026     const AFX_MSGMAP_ENTRY* lpEntry;
#0027     if (...) // 檢查是否在 cache 之中
#0028     {
#0029         // cache hit
#0030         lpEntry = msgCache.lpEntry;
#0031         if (lpEntry == NULL)
#0032             return FALSE;
#0033
#0034         // cache hit, and it needs to be handled
#0035         if (message < 0xC000)
#0036             goto LDispatch;
#0037         else
#0038             goto LDispatchRegistered;
#0039     }
#0040     else
#0041     {
#0042         // not in cache, look for it
#0043         msgCache.nMsg = message;
#0044         msgCache.pMessageMap = pMessageMap;
#0045
#0046         for (/* pMessageMap already init'ed */; pMessageMap != NULL;
#0047              pMessageMap = pMessageMap->pBaseMap)
#0048         {
#0049             // 利用 AfxFindMessageEntry 尋找訊息映射表中
#0050             // 對應的訊息處理常式。如果找到，再依 nMsg 為一般訊息
#0051             // (< 0xC000) 或自行註冊之訊息 (> 0xC000) 分別跳到
#0052             // LDispatch: 或 LDispatchRegistered: 去執行。
#0053
#0054             // Note: catch not so common but fatal mistake!!
#0055             // BEGIN_MESSAGE_MAP(CMyWnd, CMyWnd)
#0056
#0057             if (message < 0xC000)
#0058             {
```

```

#0059         // constant window message
#0060         if ((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,
#0061             message, 0, 0)) != NULL)
#0062         {
#0063             msgCache.lpEntry = lpEntry;
#0064             goto LDispatch;
#0065         }
#0066     }
#0067     else
#0068     {
#0069         // registered windows message
#0070         lpEntry = pMessageMap->lpEntries;
#0071         while ((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0))
#0072             != NULL)
#0073         {
#0074             UINT* pnID = (UINT*)(lpEntry->nSig);
#0075             ASSERT(*pnID >= 0xC000);
#0076             // must be successfully registered
#0077             if (*pnID == message)
#0078             {
#0079                 msgCache.lpEntry = lpEntry;
#0080                 goto LDispatchRegistered;
#0081             }
#0082             lpEntry++;    // keep looking past this one
#0083         }
#0084     }
#0085 }
#0086 msgCache.lpEntry = NULL;
#0087 return FALSE;
#0088 }
#0089 ASSERT(FALSE);    // not reached
#0090
#0091 LDispatch:
#0092     union MessageMapFunctions mmf;
#0093     mmf.pfn = lpEntry->pfn;
#0094
#0095     switch (lpEntry->nSig)
#0096     {
#0097     case AfxSig_bd:
#0098         lResult = (this->*mmf.pfn_bd)(CDC::FromHandle((HDC)wParam));
#0099         break;
#0100
#0101     case AfxSig_bb:    // AfxSig_bb, AfxSig_bw, AfxSig_bh
#0102         lResult = (this->*mmf.pfn_bb)((BOOL)wParam);
#0103         break;
#0104

```



```

#0105     case AfxSig_bWww: // really AfxSig_bWiw
#0106         lResult = (this->*mmf.pfn_bWww)(CWnd::FromHandle((HWND)wParam),
#0107             (short)LOWORD(lParam), HIWORD(lParam));
#0108         break;
#0109
#0110     case AfxSig_bHELPINFO:
#0111         lResult = (this->*mmf.pfn_bHELPINFO)((HELPINFO*)lParam);
#0112         break;
#0113
#0114     case AfxSig_is:
#0115         lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
#0116         break;
#0117
#0118     case AfxSig_lwl:
#0119         lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
#0120         break;
#0121
#0122     case AfxSig_vv:
#0123         (this->*mmf.pfn_vv)();
#0124         break;
#0125     ...
#0126     }
#0127     goto LReturnTrue;
#0128
#0129 LDispatchRegistered: // for registered windows messages
#0130     ASSERT(message >= 0xC000);
#0131     mmf.pfn = lpEntry->pfn;
#0132     lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
#0133
#0134 LReturnTrue:
#0135     if (pResult != NULL)
#0136         *pResult = lResult;
#0137     return TRUE;
#0138 }

#0001 AfxFindMessageEntry(const AFX_MSGMAP_ENTRY* lpEntry,
#0002     UINT nMsg, UINT nCode, UINT nID)
#0003 {
#0004     #if defined(_M_IX86) && !defined(_AFX_PORTABLE)
#0005     // 32-bit Intel 386/486 version.
#0006     ... // 以組合語言碼處理，加快速度。
#0007     #else // _AFX_PORTABLE
#0008     // C version of search routine
#0009     while (lpEntry->nSig != AfxSig_end)
#0010     {
#0011         if (lpEntry->nMessage == nMsg && lpEntry->nCode == nCode &&

```

```

#0012         nID >= lpEntry->nID && nID <= lpEntry->nLastID)
#0013     {
#0014         return lpEntry;
#0015     }
#0016     lpEntry++;
#0017 }
#0018     return NULL;    // not found
#0019 #endif // _AFX_PORTABLE
#0020 }

```

直線上溯的邏輯實在是相當單純的了，唯一做的動作就是比對訊息映射表，如果吻合就呼叫表中項目所記錄的函式。比對的對象有二，一個是原原本本的訊息映射表（那個巨大的結構），另一個是 MFC 為求快速所設計的一個 cache（cache 的實作太過複雜，我並沒有把它的原始碼表現出來）。比對成功後，呼叫對應之函式時，有一個巨大的 switch/case 動作，那是為了確保型態安全（type-safe）。稍後我有一個小節詳細討論之。

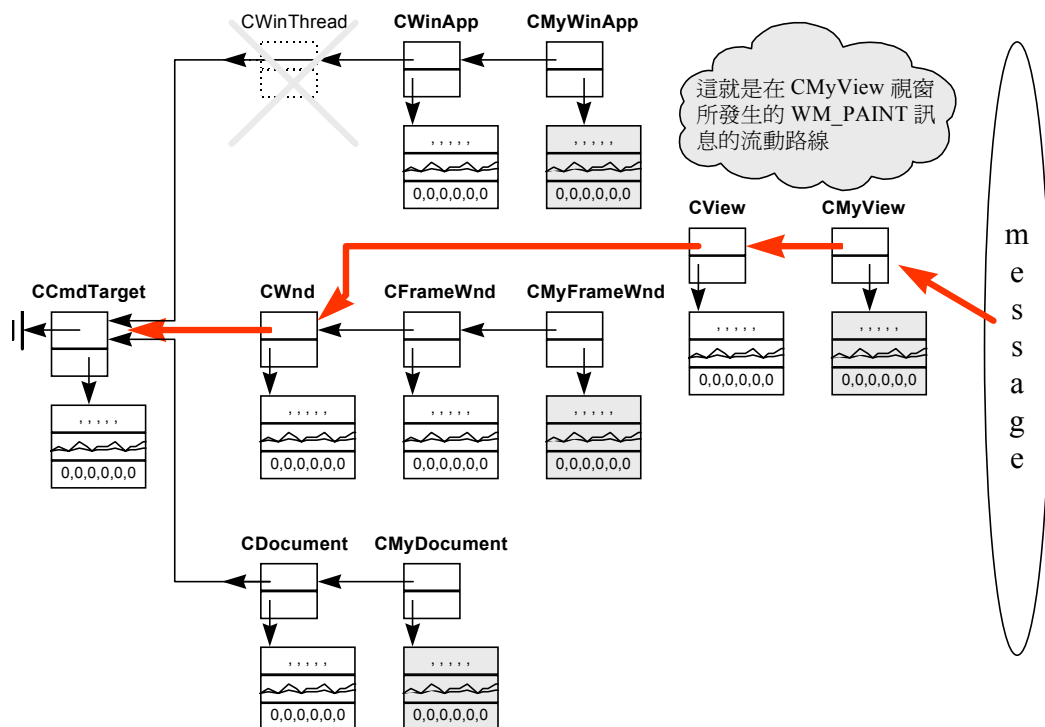


圖 9-3 當 WM_PAINT 發生於 View 視窗，訊息的流動路線。

拐彎「湖」(WM_COMMAND 命令訊息)

如果訊息是 *WM_COMMAND*，你看到了，*CWnd::OnWndMsg* (上節所述) 另闢蹊蹺，交由 *OnCommand* 來處理。這並不一定就指的是 *CWnd::OnCommand*，得視 *this* 指標指向哪一種物件而定。在 MFC 之中，以下數個類別都改寫了 *OnCommand* 虛擬函式：

```
class CWnd : public CCmdTarget
class CFrameWnd : public CWnd
class CMDIFrameWnd : public CFrameWnd
class CSplitterWnd : public CWnd
class CPropertySheet : public CWnd
class ColePropertyPage : public CDialog
```

我們挑一個例子來看。假設訊息是從 *CFrameWnd* 進來的就好了，於是：

```
// in FRMWND.CPP (MFC 4.0)
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    ...

    // route as normal command
    return CWnd::OnCommand(wParam, lParam);
}
// in WINCORE.CPP (MFC 4.0)
BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    ...
    return OnCmdMsg(nID, nCode, NULL, NULL);
}
```

這裡呼叫的 *OnCmdMsg* 並不一定就是指 *CWnd::OnCmdMsg*，得看 *this* 指標指向哪一種物件而定。目前情況是指向一個 *CFrameWnd* 物件，而 MFC 之中「擁有」*OnCmdMsg* 的類別（注意，此話有語病，我應該說 MFC 之中「曾經改寫」過 *OnCmdMsg* 的類別）是：

```
class CCmdTarget: public CObject
class CFrameWnd : public CWnd
class CMDIFrameWnd : public CFrameWnd
class CView : public CWnd
class CPropertySheet : public CWnd
class CDialog : public CWnd
class CDocument : public CCmdTarget
class ColeDocument : public CDocument
```

顯然我們應該往 *CFrameWnd* 追蹤：

```
// in FRMWND.CPP (MFC 4.0)
BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
    // pump through current view FIRST
    CView* pView = GetActiveView();
    if (pView != NULL && pView->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // then pump through frame
    if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // last but not least, pump through app
    CWinApp* pApp = AfxGetApp();
    if (pApp != NULL && pApp->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    return FALSE;
}
```

這裡非常明顯地兵分三路，正是爲了實踐 MFC 這個 Application Framework 對於命令訊息的繞行路線的規劃：

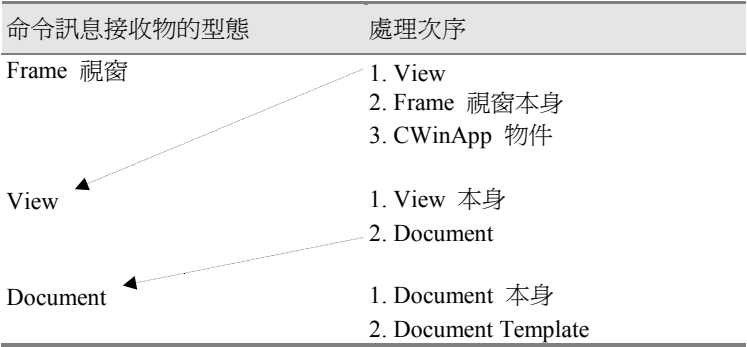


圖 9-4 MFC 對於命令訊息 WM_COMMAND 的特殊處理順序。

讓我們鍥而不捨地追蹤下去：

```
// in VIEWCORE.CPP (MFC 4.0)
BOOL CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
    // first pump through pane
    if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // then pump through document
    BOOL bHandled = FALSE;
    if (m_pDocument != NULL)
    {
        // special state for saving view before routing to document
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        CView* pOldRoutingView = pThreadState->m_pRoutingView;
        pThreadState->m_pRoutingView = this;
        bHandled = m_pDocument->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
        pThreadState->m_pRoutingView = pOldRoutingView;
    }

    return bHandled;
}
```

這反應出圖 9-4 搜尋路徑中「先 View 而後 Document」的規劃。由於 *CWnd* 並未改寫 *OnCmdMsg*，所以函式中呼叫的 *CWnd::OnCmdMsg*，其實就是 *CCmdTarget::OnCmdMsg*：

```
// in CMDTARG.CPP (MFC 4.0)
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
    ...
    // look through message map to see if it applies to us
    for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
        pMessageMap = pMessageMap->pBaseMap)
    {
        lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries, nMsg, nCode, nID);
        if (lpEntry != NULL)
        {
            // found it
            return DispatchCmdMsg(this, nID, nCode,
                lpEntry->pfn, pExtra, lpEntry->nSig, pHandlerInfo);
        }
    }
    return FALSE; // not handled
}
```

其中的 *AfxFindMessageEntry* 動作稍早我已列出。

當命令訊息兵分三路的第一路走到訊息映射網的末尾一個類別 *CCmdTarget*，沒有辦法再「節外生枝」，只能乖乖比對 *CCmdTarget* 的訊息映射表。如果沒有發現吻合者，傳回 *FALSE*，引起 *CView::OnCmdMsg* 接下去呼叫 *m_pDocument->OnCmdMsg*。如果有吻合者，呼叫全域函式 *DispatchCmdMsg*：

```
static BOOL DispatchCmdMsg(CCmdTarget* pTarget, UINT nID, int nCode,
    AFX_PMSG pfn, void* pExtra, UINT nSig, AFX_CMDHANDLERINFO* pHandlerInfo)
    // return TRUE to stop routing
{
    ASSERT_VALID(pTarget);
    UNUSED(nCode); // unused in release builds

    union MessageMapFunctions mmf;
    mmf.pfn = pfn;
    BOOL bResult = TRUE; // default is ok
    ...
    switch (nSig)
    {
    case AfxSig_vv:
        // normal command or control notification
        (pTarget->*mmf.pfn_COMMAND)();
        break;

    case AfxSig_bv:
        // normal command or control notification
        bResult = (pTarget->*mmf.pfn_bCOMMAND)();
        break;

    case AfxSig_vw:
        // normal command or control notification in a range
        (pTarget->*mmf.pfn_COMMAND_RANGE)(nID);
        break;

    case AfxSig_bw:
        // extended command (passed ID, returns bContinue)
        bResult = (pTarget->*mmf.pfn_COMMAND_EX)(nID);
        break;

    ...
    default: // illegal
        ASSERT(FALSE);
    }
```

```
        return 0;
    }
    return bResult;
}
```

以下是另一路 *CDocument* 的動作：

```
// in DOCCORE.CPP
BOOL CDocument::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
    if (CCommandTarget::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // otherwise check template
    if (m_pDocTemplate != NULL &&
        m_pDocTemplate->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    return FALSE;
}
```

圖 9-5 畫出 *FrameWnd* 視窗收到命令訊息後的四個嘗試路徑。第 3 章曾經以一個簡單的 DOS 程式模擬出這樣的繞行路線。

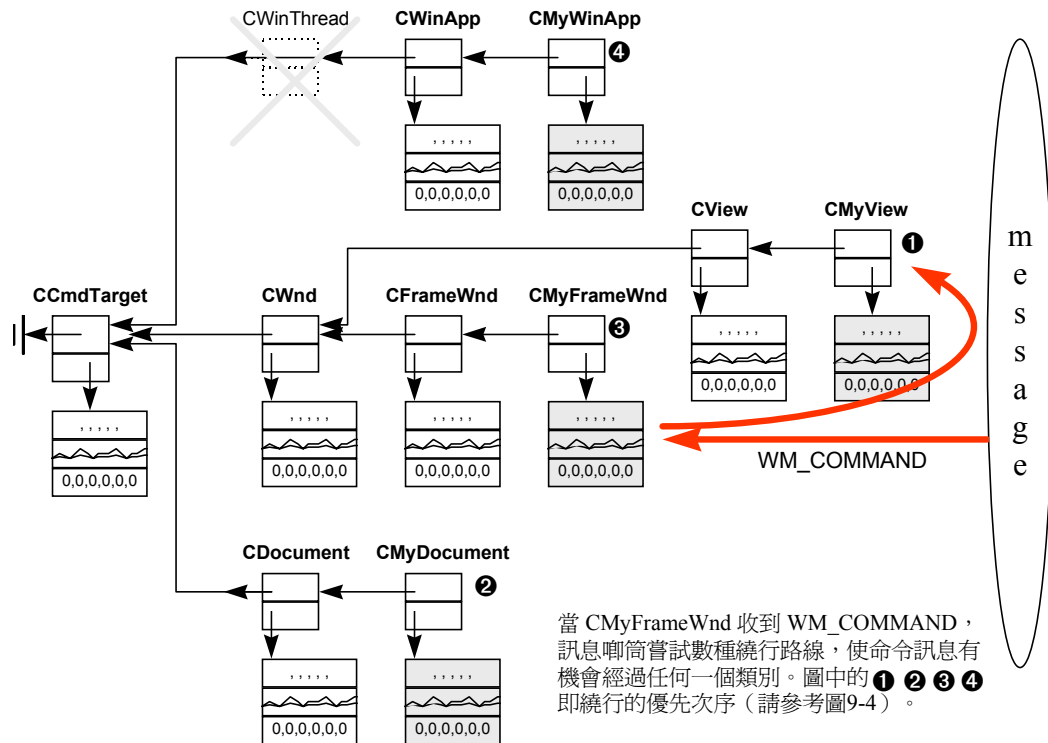


圖 9-5 FrameWnd 視窗收到命令訊息後的四個嘗試路徑。第 9 章曾經以一個簡單的 DOS 程式模擬出這樣的繞行路線。

OnCmdMsg 是各類別專門用來對付命令訊息的函式。每一個「可接受命令訊息之物件」（Command Target）在處理命令訊息時都會（都應該）遵循一個遊戲規則：呼叫另一個目標類別的 *OnCmdMsg*。這才能夠將命令訊息傳送下去。如果說 *AfxWndProc* 是訊息流動的「唧筒」，各類別的 *OnCmdMsg* 就是訊息流動的「轉轍器」。

以下我舉一個具體例子。假設命令訊息從 Scribble 的【Edit/Clear All】發出，其處理常式位在 *CScribbleDoc*，下面是這個命令訊息的流浪過程：

1. MDI 主視窗（*CMDIFrameWnd*）收到命令訊息 *WM_COMMAND*，其 ID 為 *ID_EDIT_CLEAR_ALL*。
2. MDI 主視窗把命令訊息交給目前作用中的 MDI 子視窗（*CMDIChildWnd*）。
3. MDI 子視窗給它自己的子視窗（也就是 *View*）一個機會。
4. *View* 檢查自己的 *Message Map*。
5. *View* 發現沒有任何處理常式可以處理此命令訊息，只好把它傳給 *Document*。

6. *Document* 檢查自己的 *Message Map*，它發現了一個吻合項：

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    ...
END_MESSAGE_MAP()
```

於是呼叫該函式，命令訊息的流動路線也告終止。

如果上述的步驟 6 仍沒有找到處理函式，那麼就：

7. *Document* 把這個命令訊息再送到 *Document Template* 物件去。
8. 還是沒被處理，於是命令訊息回到 *View*。
9. *View* 沒有處理，於是又回給 MDI 子視窗本身。
10. 傳給 *CWinApp* 物件 -- 無主訊息的終極歸屬。

圖 9-6 是構成「訊息邦浦」之各個函式的呼叫次序。此圖可以對前面所列之各個原始碼組織出一個大局觀來。

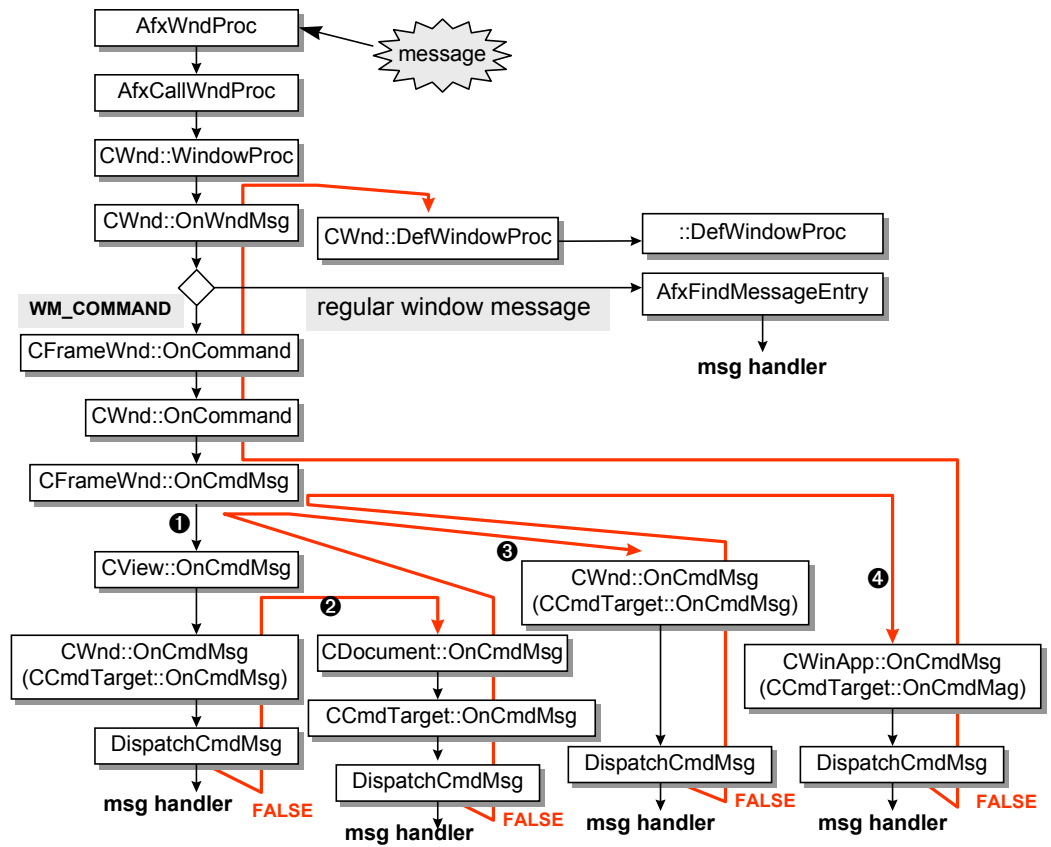


圖 9-6 構成「訊息邦浦」之各個函式的呼叫次序

羅塞達碑石：AfxSig_xx 的奧秘

大架構建立起來了，但我還沒有很仔細地解釋在訊息映射「網」中的 `_messageEntries[]` 陣列內容。為什麼訊息經由推動引擎（上一節談的那整套傢伙）推過這些陣列，就可以找到它的處理常式？

Paul DiLascia 在他的文章（*i Meandering Through the Maze of MFC Message and Command Routing*，*Microsoft Systems Journal*，1995/07）中形容這些陣列之內一筆一筆的記錄像是羅塞達碑石，呵呵，就靠它們揭開訊息映射的最後謎底了。

羅塞達碑石（Rosetta Stone），1799 年拿破崙遠征埃及時，由一名官員在尼羅河口羅塞達發現，揭開了古埃及象形文字之謎。石碑是黑色玄武岩，高 114 公分，厚 28 公分，寬 72 公分。經法國學者 Jean-Francois Champollion 研究後，世人因得順利研讀古埃及文獻。

訊息映射表的每一筆記錄是這樣的形式：

```
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;    // windows message
    UINT nCode;       // control code or WM_NOTIFY code
    UINT nID;         // control ID (or 0 for windows messages)
    UINT nLastID;     // used for entries specifying a range of control id's
    UINT nSig;        // signature type (action) or pointer to message #
    AFX_PMSG pfn;     // routine to call (or special value)
};
```

內中包括一個 Windows 訊息、其控制元件 ID 以及通告碼（notification code，對訊息的更多描述，例如 `EN_CHANGED` 或 `CBN_DROPDOWN` 等）、一個簽名記號、以及一個 `CCmdTarget` 衍生類別的成員函式。任何一個 `ON_` 巨集會把這六個項目初始化起來。例如：

```
#define ON_WM_CREATE() \
{ WM_CREATE, 0, 0, 0, AfxSig_is, \
  (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL CWnd::*)(LPCREATESTRUCT))OnCreate },
```

你看到了可怕的型態轉換動作，這完全是為了保持型態安全（type-safe）。

有一個很莫名其妙的東西：*AfxSig_*。要了解它作什麼用，你得先停下來幾分鐘，想想另一個問題：當上一節的推動引擎比對訊息並發現吻合之後，就呼叫對應的處理常式，但它怎麼知道要交給訊息處理常式哪些參數呢？要知道，不同的訊息處理常式需要不同的參數（包括個數和型態），而其函式指標（*AFX_PMSG*）卻都被定義為這付德行：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

這麼簡陋的資訊無法表現應該傳遞什麼樣的參數，而這正是 *AfxSig_* 要貢獻的地方。當推動引擎比對完成，欲呼叫某個訊息處理常式 *lpEntry->pfn* 時，動作是這樣子地（出現在 *CWnd::OnWndMsg* 和 *DispatchCmdMsg* 中）：

```
union MessageMapFunctions mmf;
mmf.pfn = lpEntry->pfn;

switch (lpEntry->nSig)
{
case AfxSig_is:
    lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
    break;

case AfxSig_lwl:
    lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
    break;

case AfxSig_vv:
    (this->*mmf.pfn_vv)();
    break;
...
}
```

注意兩樣東西：*MessageMapFunctions* 和 *AfxSig_*。*AfxSig_* 定義於 *AFXMSG.H* 檔：

```
enum AfxSig
{
    AfxSig_end = 0,        // [marks end of message map]

    AfxSig_bD,            // BOOL (CDC*)
    AfxSig_bb,            // BOOL (BOOL)
    AfxSig_bWww,          // BOOL (CWnd*, UINT, UINT)
    AfxSig_hDWw,          // HBRUSH (CDC*, CWnd*, UINT)
    AfxSig_hDw,           // HBRUSH (CDC*, UINT)
```

```

AfxSig_iwWw,    // int (UINT, CWnd*, UINT)
AfxSig_iww,     // int (UINT, UINT)
AfxSig_iWww,    // int (CWnd*, UINT, UINT)
AfxSig_is,      // int (LPTSTR)
AfxSig_lwl,     // LRESULT (WPARAM, LPARAM)
AfxSig_lwwM,    // LRESULT (UINT, UINT, CMenu*)
AfxSig_vv,      // void (void)

AfxSig_vw,      // void (UINT)
AfxSig_vww,     // void (UINT, UINT)
AfxSig_vvii,    // void (int, int) // wParam is ignored
AfxSig_vwww,    // void (UINT, UINT, UINT)
AfxSig_vwii,    // void (UINT, int, int)
AfxSig_vwl,     // void (UINT, LPARAM)
AfxSig_vbWW,    // void (BOOL, CWnd*, CWnd*)
AfxSig_vD,      // void (CDC*)
AfxSig_vM,      // void (CMenu*)
AfxSig_vMwb,    // void (CMenu*, UINT, BOOL)

AfxSig_vW,      // void (CWnd*)
AfxSig_vWww,    // void (CWnd*, UINT, UINT)
AfxSig_vWp,     // void (CWnd*, CPoint)
AfxSig_vWh,     // void (CWnd*, HANDLE)
AfxSig_vwW,     // void (UINT, CWnd*)
AfxSig_vwWb,    // void (UINT, CWnd*, BOOL)
AfxSig_vwwW,    // void (UINT, UINT, CWnd*)
AfxSig_vwwx,    // void (UINT, UINT)
AfxSig_vs,      // void (LPTSTR)
AfxSig_vOWNER,  // void (int, LPTSTR), force return TRUE
AfxSig_iis,     // int (int, LPTSTR)
AfxSig_wp,      // UINT (CPoint)
AfxSig_wv,      // UINT (void)
AfxSig_vPOS,    // void (WINDOWPOS*)
AfxSig_vCALC,   // void (BOOL, NCCALCSIZE_PARAMS*)
AfxSig_vNMHDRpl, // void (NMHDR*, LRESULT*)
AfxSig_bNMHDRpl, // BOOL (NMHDR*, LRESULT*)
AfxSig_vvNMHDRpl, // void (UINT, NMHDR*, LRESULT*)
AfxSig_bwNMHDRpl, // BOOL (UINT, NMHDR*, LRESULT*)
AfxSig_bHELPINFO, // BOOL (HELPINFO*)
AfxSig_vwSIZING, // void (UINT, LPRECT) -- return TRUE

// signatures specific to CCmdTarget
AfxSig_cmdui,   // void (CCmdUI*)
AfxSig_cmduiw,  // void (CCmdUI*, UINT)
AfxSig_vpv,     // void (void*)
AfxSig_bpv,     // BOOL (void*)

```

```

// Other aliases (based on implementation)
AfxSig_vwwh,          // void (UINT, UINT, HANDLE)
AfxSig_vwvp,          // void (UINT, CPoint)
AfxSig_bw = AfxSig_bb, // BOOL (UINT)
AfxSig_bh = AfxSig_bb, // BOOL (HANDLE)
AfxSig_iw = AfxSig_bb, // int (UINT)
AfxSig_ww = AfxSig_bb, // UINT (UINT)
AfxSig_bv = AfxSig_wv, // BOOL (void)
AfxSig_hv = AfxSig_wv, // HANDLE (void)
AfxSig_vb = AfxSig_vw, // void (BOOL)
AfxSig_vbh = AfxSig_vww, // void (BOOL, HANDLE)
AfxSig_vbw = AfxSig_vww, // void (BOOL, UINT)
AfxSig_vhh = AfxSig_vww, // void (HANDLE, HANDLE)
AfxSig_vh = AfxSig_vw, // void (HANDLE)
AfxSig_viSS = AfxSig_vwl, // void (int, STYLESTRUCT*)
AfxSig_bwl = AfxSig_lwl,
AfxSig_vwMOVING = AfxSig_vwSIZING, // void (UINT, LPRECT) -- return TRUE
};

```

MessageMapFunctions 定義於 WINCORE.CPP 檔：

```

union MessageMapFunctions
{
    AFX_PMSG pfn; // generic member function pointer

    // specific type safe variants
    BOOL (AFX_MSG_CALL CWnd::*pfn_bD)(CDC*);
    BOOL (AFX_MSG_CALL CWnd::*pfn_bb)(BOOL);
    BOOL (AFX_MSG_CALL CWnd::*pfn_bWww)(CWnd*, UINT, UINT);
    BOOL (AFX_MSG_CALL CWnd::*pfn_bHELPINFO)(HELPINFO*);
    HBRUSH (AFX_MSG_CALL CWnd::*pfn_hDWw)(CDC*, CWnd*, UINT);
    HBRUSH (AFX_MSG_CALL CWnd::*pfn_hDw)(CDC*, UINT);
    int (AFX_MSG_CALL CWnd::*pfn_iwWw)(UINT, CWnd*, UINT);
    int (AFX_MSG_CALL CWnd::*pfn_iww)(UINT, UINT);
    int (AFX_MSG_CALL CWnd::*pfn_iWww)(CWnd*, UINT, UINT);
    int (AFX_MSG_CALL CWnd::*pfn_is)(LPTSTR);
    LRESULT (AFX_MSG_CALL CWnd::*pfn_lwl)(WPARAM, LPARAM);
    LRESULT (AFX_MSG_CALL CWnd::*pfn_lwwM)(UINT, UINT, CMenu*);
    void (AFX_MSG_CALL CWnd::*pfn_vv)(void);

    void (AFX_MSG_CALL CWnd::*pfn_vw)(UINT);
    void (AFX_MSG_CALL CWnd::*pfn_vww)(UINT, UINT);
    void (AFX_MSG_CALL CWnd::*pfn_vvii)(int, int);
    void (AFX_MSG_CALL CWnd::*pfn_vwww)(UINT, UINT, UINT);
    void (AFX_MSG_CALL CWnd::*pfn_vwii)(UINT, int, int);
}

```

```

void (AFX_MSG_CALL CWnd::*pfn_vw1)(WPARAM, LPARAM);
void (AFX_MSG_CALL CWnd::*pfn_vbWW)(BOOL, CWnd*, CWnd*);
void (AFX_MSG_CALL CWnd::*pfn_vD)(CDC*);
void (AFX_MSG_CALL CWnd::*pfn_vM)(CMenu*);
void (AFX_MSG_CALL CWnd::*pfn_vMwb)(CMenu*, UINT, BOOL);

void (AFX_MSG_CALL CWnd::*pfn_vW)(CWnd*);
void (AFX_MSG_CALL CWnd::*pfn_vWww)(CWnd*, UINT, UINT);
void (AFX_MSG_CALL CWnd::*pfn_vWp)(CWnd*, CPoint);
void (AFX_MSG_CALL CWnd::*pfn_vWh)(CWnd*, HANDLE);
void (AFX_MSG_CALL CWnd::*pfn_vwW)(UINT, CWnd*);
void (AFX_MSG_CALL CWnd::*pfn_vwWb)(UINT, CWnd*, BOOL);
void (AFX_MSG_CALL CWnd::*pfn_vwwW)(UINT, UINT, CWnd*);
void (AFX_MSG_CALL CWnd::*pfn_vwwx)(UINT, UINT);
void (AFX_MSG_CALL CWnd::*pfn_vs)(LPTSTR);
void (AFX_MSG_CALL CWnd::*pfn_vOWNER)(int, LPTSTR); // force return TRUE
int (AFX_MSG_CALL CWnd::*pfn_iis)(int, LPTSTR);
UINT (AFX_MSG_CALL CWnd::*pfn_wp)(CPoint);
UINT (AFX_MSG_CALL CWnd::*pfn_wv)(void);
void (AFX_MSG_CALL CWnd::*pfn_vPOS)(WINDOWPOS*);
void (AFX_MSG_CALL CWnd::*pfn_vCALC)(BOOL, NCCALCSIZE_PARAMS*);
void (AFX_MSG_CALL CWnd::*pfn_vwp)(UINT, CPoint);
void (AFX_MSG_CALL CWnd::*pfn_vwwh)(UINT, UINT, HANDLE);
};

```

其實呢，真正的函式只有一個 *pfn*，但通過 *union*，它有許多型態不同的形象。*pfn_vv* 代表「參數為 *void*，傳回值為 *void*」；*pfn_hwl* 代表「參數為 *wParam* 和 *lParam*，傳回值為 *LRESULT*」；*pfn_is* 代表「參數為 *LPTSTR* 字串，傳回值為 *int*」。

相當精緻，但是也有點兒可怖，是不是？使用 MFC 或許應該像吃蜜餞一樣；蜜餞很好吃，但你最好不要看到蜜餞的生產過程！唔，我真的不知道！

無論如何，我把所有的神秘都揭開在你面前了。

山高月小 水落石出

Scribble Step2：UI 物件的變化

理論基礎建立完畢，該是實作的時候。Step2 將新增三個選單命令項，一個工具列按鈕，並維護這些 UI 物件的使用狀態。

改變選單

Step2 將增加一個【Pen】選單，其中有兩個命令項目；並在【Edit】選單中增加一個【Clear All】命令項目：

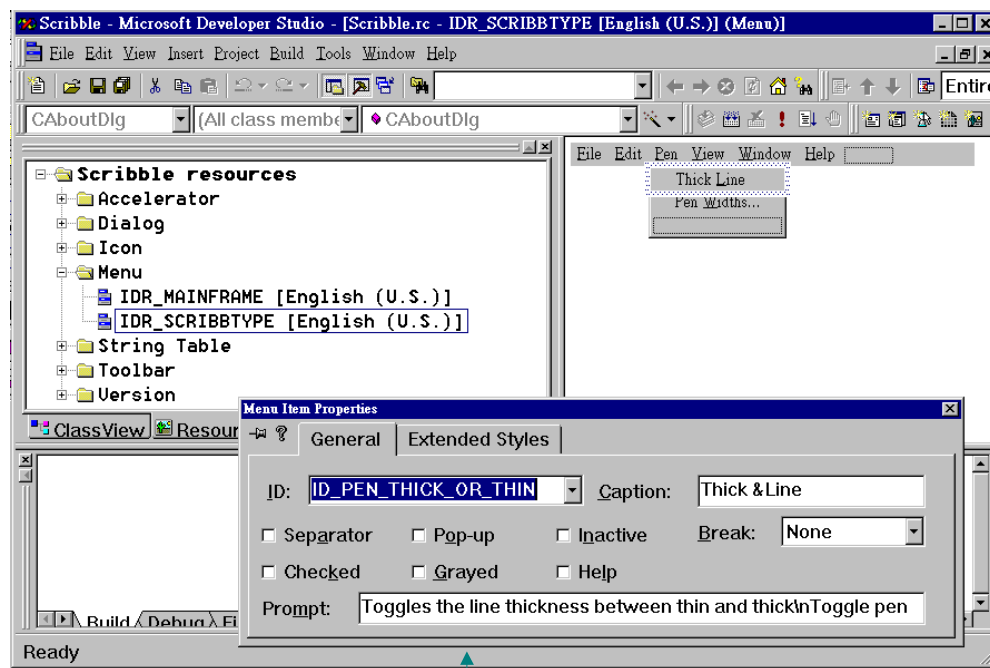


- 【Pen/Thick Line】：這是一個切換開關，允許設定使用粗筆或細筆。如果使用者設定粗筆，我們在這項目的旁邊打個勾（所謂的 checked）；如果使用者選擇細筆（也就是在打勾的本項目上再按一下），我們就把勾號去除（所謂的 unchecked）。
- 【Pen/Pen Widths】：這會喚起一個對話盒，允許設定筆的寬度。對話盒的設計並不在本章範圍，那是下一章的事。
- 【Edit/Clear All】：清除目前作用之 Document 資料。當然對應之 View 視窗內容也應該清乾淨。

Visual C++ 整合環境中的選單編輯器擁有非常方便的滑鼠拖放（drag and drop）功能，所以做出上述的選單命令項不是難事。不過這些命令項目還得經過某些動作，才能與程式碼關聯起來發生作用，這方面 ClassWizard 可以幫助我們。稍後我會說明這一切。

以下利用 Visual C++ 整合環境中的選單編輯器修改選單：

- 啟動選單編輯器（請參考第 4 章）。Scribble 有兩份選單，IDR_MAINFRAME 適用於沒有任何子視窗的情況，IDR_SCRIBBTYPE 適用於有子視窗的情況。我們選擇後者。



- IDR_SCRIBBTYPE 選單內容出現於畫面右半側。加入新增的三個命令項。每個命令項會獲得一個獨一無二的識別碼，定義於 RESOURCE.H 或任何你指定的檔案中。圖下方的【Menu Item Properties】對話盒在你雙擊某個命令項後出現，允許你更改命令項的識別碼與提示字串（將出現在狀態列中）。如果你對操作過程不熟練，請參考 [Visual C++ User's Guide](#)（Visual C++ Online 上附有此書之電子版）。
- 三個新命令項的 ID 值以及提示字串整理於下：

【Pen/Thick Line】
 ID : ID_PEN_THICK_OR_THIN
 prompt : "Toggles the line thickness between thin and thick\nToggle pen"

```

【Pen/Pen Widths】
ID : ID_PEN_WIDTHS
prompt : "Sets the size of the thin and thick pen\nPen thickness"

【Edit/Clear All】
ID : ID_EDIT_CLEAR_ALL (這是一個預先定義的 ID，有預設的提示字串，請更改如下)
prompt : "Clears the drawing\nErase All"

```

注意：每一個提示字串都有一個 \n 子字串，那是作為工具列按鈕的「小黃標籤」的標籤內容。「小黃標籤」（學名叫作 tool tips）是 Windows 95 新增的功能。

對 Framework 而言，命令項的 ID 是用以識別命令訊息的唯一依據。你只需在【Properties】對話盒中鍵入你喜歡的 ID 名稱（如果你不滿意選單編輯器自動給你的那個），至於它真正的數值不必在意，選單編輯器會在你的 RESOURCE.H 檔中加上定義值。

經過上述動作，選單編輯器影響我們的程式碼如下：

```

// in RESOURCE.H
#define ID_PEN_THICK_OR_THIN      32772
#define ID_PEN_WIDTHS            32773
(註：另一個 ID ID_EDIT_CLEAR_ALL 已預先定義於 AFXRES.H 中)

// in SCRIBBLE.RC
IDR_SCRIBBTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ...
    POPUP "&Edit"
    BEGIN
        ...
        MENUITEM "Clear &All",      ID_EDIT_CLEAR_ALL
    END
    POPUP "&Pen"
    BEGIN
        MENUITEM "Thick &Line",      ID_PEN_THICK_OR_THIN
        MENUITEM "Pen &Widths...",    ID_PEN_WIDTHS
    END
    ...
END

```

```

STRINGTABLE DISCARDABLE
BEGIN
    ID_PEN_THICK_OR_THIN  "Toggles the line thickness between thin and thick\nToggle pen"
    ID_PEN_WIDTHS         "Sets the size of the thin and thick pen\nPen thickness"
END

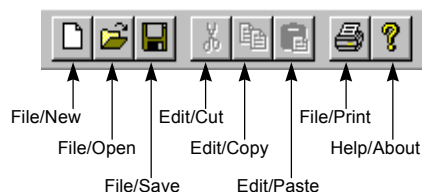
STRINGTABLE DISCARDABLE
BEGIN
    ID_EDIT_CLEAR_ALL     "Clears the drawing\nErase All"
    ...
END

```

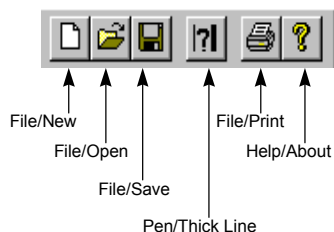
改變工具列

過去，也就是 Visual C++ 4.0 之前，改變工具列有點麻煩。你必須先以圖形編輯器修改工具列對應之 bitmap 圖形，然後更改程式碼中對應的工具列按鈕識別碼。現在可就輕鬆多了，工具列編輯器讓我們一氣呵成。主要原因是，工具列現今也成為了資源的一種。

下面是 Scribble Step1 的工具列：

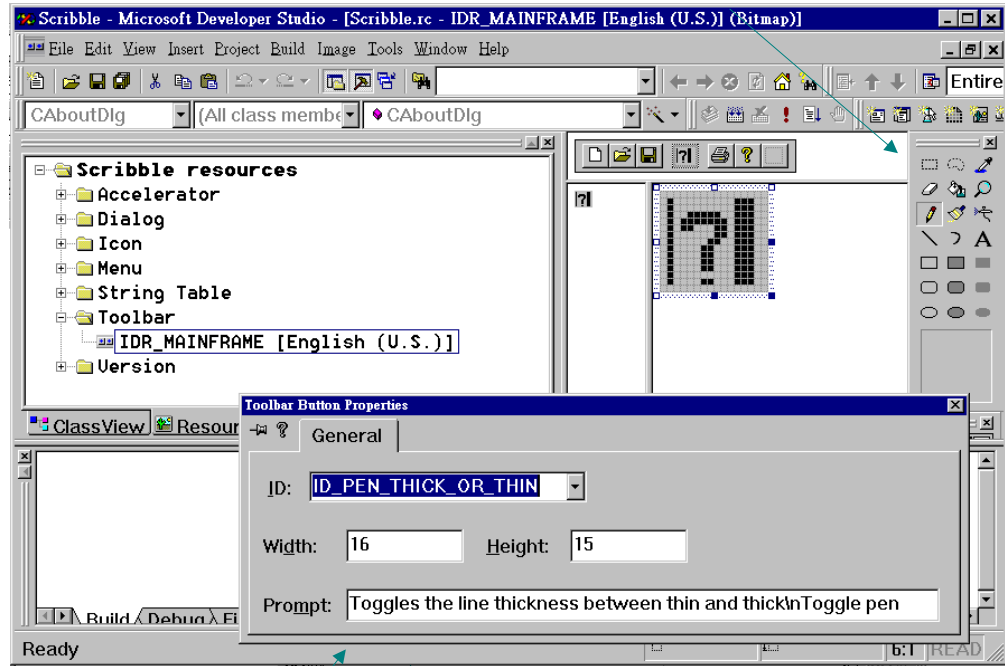


現在我希望為【Pen/Thick Line】命令項設計一個工具列按鈕，並且把 Scribble 用不到的三個預設按鈕去除（分別是 Cut、Copy、Paste）：



編輯動作如下：

- 啟動工具列編輯器，選擇 `IDR_MAINFRAME`。有一個繪圖工具箱出現在最右側。



- 將三個用不著的按鈕除去：以滑鼠拖拉這些按鈕，拉到工具列以外即可。
- 在工具列最右側的空白按鈕上作畫，並將它拖拉到適當位置。
- 為了讓這個新的按鈕起作用，必須指定一個 ID 給它。我們希望這個按鈕相當於【Pen/Thick Line】命令項，所以它的 ID 當然應該與該命令項的 ID 相同，也就是 `ID_PEN_THICK_OR_THIN`。雙擊這個新按鈕，出現【Toolbar Button Properties】對話盒，請選擇正確的 ID。注意，由於此一 ID 先前已定義好，所以其提示字串以及小黃標籤也就與此一工具列按鈕產生了關聯。
- 存檔。
- 工具列編輯器為我們修改了工具列的 bitmap 圖形檔內容：


```
IDR_MAINFRAME BITMAP MOVEABLE PURE "res\\Toolbar.bmp"
```

 同時，工具列項目也由原來的：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE 16, 15
BEGIN
    BUTTON        ID_FILE_NEW
    BUTTON        ID_FILE_OPEN
    BUTTON        ID_FILE_SAVE
    SEPARATOR
    BUTTON        ID_EDIT_CUT
    BUTTON        ID_EDIT_COPY
    BUTTON        ID_EDIT_PASTE
    SEPARATOR
    BUTTON        ID_FILE_PRINT
    BUTTON        ID_APP_ABOUT
END
```

改變為：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE 16, 15
BEGIN
    BUTTON        ID_FILE_NEW
    BUTTON        ID_FILE_OPEN
    BUTTON        ID_FILE_SAVE
    SEPARATOR
    BUTTON        ID_PEN_THICK_OR_THIN
    SEPARATOR
    BUTTON        ID_FILE_PRINT
    BUTTON        ID_APP_ABOUT
END
```

利用 ClassWizard 連接命令項識別碼與命令處理函式

新增的三個命令項和一個工具列按鈕，都會產生命令訊息。接下來的任務就是為它們指定一個對應的命令訊息處理常式。下面是一份整理：

UI 物件（命令項）	項目識別碼	處理常式
【Pen/Thick Line】	ID_PEN_THICK_OR_THIN	OnPenThickOrThin
【Pen/Pen Widths】	ID_PEN_WIDTHS	OnPenWidths（第 10 章再處理）

【Edit/Clear All】	ID_EDIT_CLEAR_ALL	OnEditClearAll
------------------	-------------------	----------------

訊息與其處理常式的連接關係是在程式的 Message Map 中確立，而 Message Map 可藉由 ClassWizard 或 WizardBar 完成。第8章已經利用這兩個工具成功地為三個標準的 Windows 訊息（WM_LBUTTONDOWN、WM_LBUTTONUP、WM_MOUSEMOVE）設立其訊息處理函式，現在我們要為 Step2 新增的命令訊息設立訊息處理常式。過程如下：

- 首先你必須決定，在哪裡攔截【Edit/Clear All】才好？本章前面對於訊息映射與命令繞行的深度討論這會兒派上了用場。【Edit/Clear All】這個命令的目的是要清除文件，文件的根本是在資料的「體」，而不在資料的「面」，所以把文件的命令處理常式放在 Document 類別中比放在 View 類別來得高明。命令訊息會不會流經 Document 類別？經過前數節的深度之旅，你應該自有定論了。
- 所以，讓我們在 CScribbleDoc 的 WizardBar 選擇【Object IDs】為 ID_EDIT_CLEAR_ALL，並選擇【Messages】為 COMMAND。
- 猜猜看，如果你在【Object IDs】中選擇 CScribbleDoc，右側的【Messages】清單會出現什麼？什麼都沒有！因為 Document 類別只可能接受 WM_COMMAND，這一點你應該已經從前面所說的訊息遞送過程中知道了。如果你在 CScribbleApp 的 WizardBar 上選擇【Object IDs】為 CScribbleApp，右側的【Messages】清單中也是什麼都沒有，道理相同。
- 你會獲得一個對話盒，詢問你是否接受一個新的處理常式。選擇 Yes，於是文字編輯器中出現該函式之骨幹，等待你的幸臨...

這樣就完成了命令訊息與其處理函式的連接工作。這個工作稱為 "command binding"。我們的原始碼獲得以下修改：

- Document 類別之中多了一個函式宣告：
- ```
class CScribbleDoc : public CDocument
{
protected:
 afx_msg void OnEditClearAll();
 ...
}
```

- Document 類別的 Message Map 中多了一筆記錄：

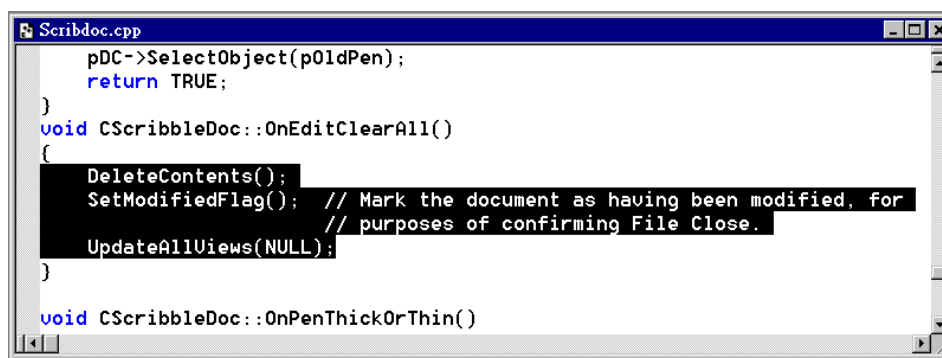
```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
 ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
 ...
END_MESSAGE_MAP()
```

- Document 類別中多了一個函式空殼：

```
void CScribbleDoc::OnEditClearAll()
{
}

```

- 現在請寫下 *OnEditClearAll* 函式碼：



依此要領，我們再設計 *OnPenThickOrThin* 函式。此一函式用來更改現行的筆寬，與 Document 有密切關係，所以在 Document 類別中放置其訊息處理常式是適當的：

```
void CScribbleDoc::OnPenThickOrThin()
{
 // Toggle the state of the pen between thin or thick.
 m_bThickPen = !m_bThickPen;

 // Change the current pen to reflect the new user-specified width.
 ReplacePen();
}
```

```

void CScribbleDoc::ReplacePen()
{
 m_nPenWidth = m_bThickPen? m_nThickWidth : m_nThinWidth;

 // Change the current pen to reflect the new user-specified width.
 m_penCur.DeleteObject();
 m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)); // solid black
}

```

注意，*ReplacePen* 並非由 WizardBar (或 ClassWizard) 加上，所以我們必須自行在 *CScribbleDoc* 類別中加上這個函式的宣告：

```

class CScribbleDoc : public CDocument
{
protected:
 void ReplacePen();
 ...
}

```

*OnPenThickOrThin* 函式用來更換筆的寬度，所以 *CScribbleDoc* 勢必需要加些新的成員變數。變數 *m\_bThickPen* 用來記錄目前筆的狀態 (粗筆或細筆)，變數 *m\_nThinWidth* 和 *m\_nThickWidth* 分別記錄粗筆和細筆的筆寬 -- 在 Step2 中此二者固定為 2 和 5，原本並不需要變數的設置，但下一章的 Step3 中粗筆和細筆的筆寬可以更改，所以這裡未雨綢繆：

```

class CScribbleDoc : public CDocument
{
// Attributes
protected:
 UINT m_nPenWidth; // current user-selected pen width
 BOOL m_bThickPen; // TRUE if current pen is thick
 UINT m_nThinWidth;
 UINT m_nThickWidth;
 CPen m_penCur; // pen created according to
 ...
}

```



現在重新考慮文件初始化的動作，將 Step1 的：

```
void CScribbleDoc::InitDocument()
{
 m_nPenWidth = 2; // default 2 pixel pen width
 // solid, black pen
 m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0));
}
```

改變為 Step2 的：

```
void CScribbleDoc::InitDocument()
{
 m_bThickPen = FALSE;
 m_nThinWidth = 2; // default thin pen is 2 pixels wide
 m_nThickWidth = 5; // default thick pen is 5 pixels wide
 ReplacePen(); // initialize pen according to current width
}
```

## 維護 UI 物件狀態 (UPDATE\_COMMAND\_UI)

上一節我曾提過 WizardBar 右側的【Messages】清單中，針對各個命令項，會出現 COMMAND 和 UPDATE\_COMMAND\_UI 兩種選擇。後者做什麼用？

一個選單拉下來，使用者可以從命令項的狀態（打勾或沒打勾、灰色或正常）得到一些狀態提示。如果 Document 中沒有任何資料的話，【Edit/Clear All】照道理就不應該起作用，因為根本沒資料又如何 "Clear All" 呢?! 這時候我們應該把這個命令項除能（disable）。又例如在粗筆狀態下，程式的【Pen/Thick Line】命令項應該打一個勾（所謂的 check mark），在細筆狀態下不應該打勾。此外，選單命令項的狀態應該同步影響到對應之工具列按鈕狀態。

所有 UI 物件狀態的維護可以依賴所謂的 UPDATE\_COMMAND\_UI 訊息。

傳統 SDK 程式中要改變選單命令項狀態，可以呼叫 *EnableMenuItem* 或是 *CheckMenuItem*，但這使得程式雜亂無章，因為你沒有一個固定的位置和固定的原則處理命令項狀態。MFC 提供一種直覺並且仍舊依賴訊息觀念的方式，解決這個問題，這就

是 `UPDATE_COMMAND_UI` 訊息。其設計理念是，每當選單被拉下並尚未顯示之前，其命令項（以及對應之工具列按鈕）都會收到 `UPDATE_COMMAND_UI` 訊息，這個訊息和 `WM_COMMAND` 有一樣的繞行路線，我們（程式員）只要在適當的類別中放置其處理函式，並在函式中做某些判斷，便可決定如何顯示命令項。

這種方法的最大好處是，不但把問題的解決方式統一化，更因為 Framework 傳給 `UPDATE_COMMAND_UI` 處理常式的參數是一個「指向 `CCmdUI` 物件的指標」，而 `CCmdUI` 物件就代表著對應的選單命令項，因此你只需呼叫 `CCmdUI` 所準備的，專門用來處理命令項外觀的函式（如 `Enable` 或 `SetCheck`）即可。我們的工作量大為減輕。

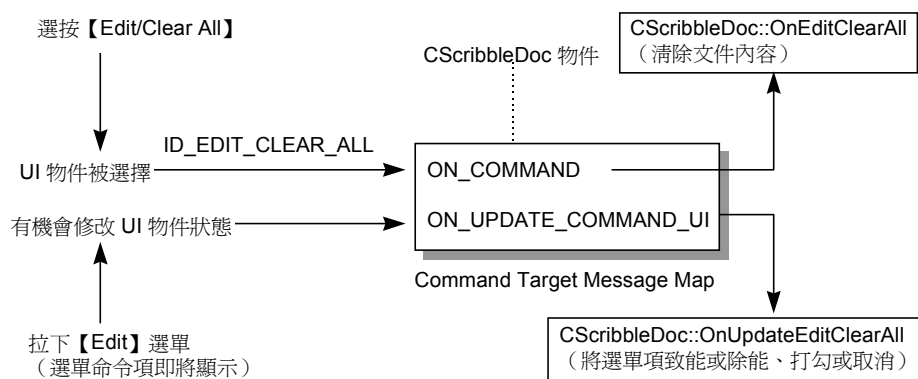


圖 9-7 `ON_COMMAND` 和 `ON_UPDATE_COMMAND_UI` 的運作

圖 9-7 以【Edit/Clear All】實例說明 `ON_COMMAND` 和 `ON_UPDATE_COMMAND_UI` 的運作。為了攔截 `UPDATE_COMMAND_UI` 訊息，你的 Command Target 物件（也許是 Application，也許是 windows，也許是 Views，也許是 Documents）要做兩件事情：

1. 利用 WizardBar（或 ClassWizard）加上一筆 Message Map 項目如下：

```
ON_UPDATE_COMMAND_UI (ID_xxx, OnUpdatexxx)
```

2. 提供一個 *OnUpdatexxx* 函式。這個函式的寫法十分簡單，因為 Framework 傳來一個代表 UI 物件（也就是選單命令項或工具列按鈕）的 *CCmdUI* 物件指標，而對 UI 物件的各種操作又都已設計在 *CCmdUI* 成員函式中。舉個例子：

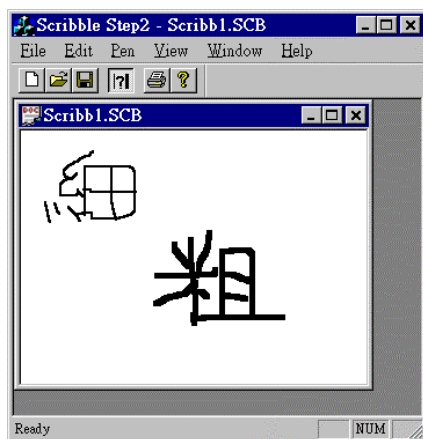
```
void CScribbleDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
 pCmdUI->Enable(!m_strokeList.IsEmpty());
}

void CScribbleDoc::OnUpdatePenThickOrThin(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(m_bThickPen);
}
```

如果命令項與某個工具列按鈕共用同一個命令 ID，上述的 *Enable* 動作將不只影響命令項，也影響按鈕。命令項的打勾（checked）即是按鈕的按下（depressed），命令項沒有打勾（unchecked）即是按鈕的正常化（鬆開）。

現在，Scribble 第二版全部修改完畢，製作並測試之：

- 在整合環境中按下【Build/Build Scribble】編譯並聯結。
- 按下【Build/Execute】執行 Scribble。測試細筆粗筆的運作情況，以及【Edit /Clear All】是否生效。



從寫程式（而不是挖背後意義）的角度去看 Message Map，我把 Step2 所進行的選單改變對 Message Map 造成的影響做個總整理。一共有四個相關成份會被 ClassWizard（或 WizardBar）產生出來，下面就是相關原始碼，其中只有第 4 項的函式內容是我們撰寫的，其它都由工具自動完成。

### 1. CSRIBBLEDOC.CPP

```
BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
 //{AFX_MSG_MAP(CScribbleDoc)
 ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
 ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
 ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
 ON_UPDATE_COMMAND_UI(ID_PEN_THICK_OR_THIN, OnUpdatePenThickOrThin)
 //}AFX_MSG_MAP
END_MESSAGE_MAP()
```

不要去掉 //{ 和 //}，否則下次 ClassWizard 或 WizardBar 不能正常工作。

### 2. CSRIBBLEDOC.H

```
class CScribbleDoc : public CDocument
{
...
// Generated message map functions
protected:
 //{AFX_MSG(CScribbleDoc)
 afx_msg void OnEditClearAll();
 afx_msg void OnPenThickOrThin();
 afx_msg void OnUpdateEditClearAll(CCmdUI* pCmdUI);
 afx_msg void OnUpdatePenThickOrThin(CCmdUI* pCmdUI);
 //}AFX_MSG
...
};
```

### 3. RESOURCE.H

```
#define ID_PEN_THICK_OR_THIN 32772
#define ID_PEN_WIDTHS 32773
（另一個項目 ID_EDIT_CLEAR_ALL 已經在 AFXRES.H 中定義了）
```

#### 4. SCRIBBLED OC.CPP

```
void CScribbleDoc::OnEditClearAll()
{
 DeleteContents();
 SetModifiedFlag(); // Mark the document as having been modified, for
 // purposes of confirming File Close.
 UpdateAllViews(NULL);
}

void CScribbleDoc::OnPenThickOrThin()
{
 // Toggle the state of the pen between thin or thick.
 m_bThickPen = !m_bThickPen;

 // Change the current pen to reflect the new user-specified width.
 ReplacePen();
}

void CScribbleDoc::ReplacePen()
{
 m_nPenWidth = m_bThickPen? m_nThickWidth : m_nThinWidth;

 // Change the current pen to reflect the new user-specified width.
 m_penCur.DeleteObject();
 m_penCur.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)); // solid black
}

void CScribbleDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
 // Enable the command user interface object (menu item or tool bar
 // button) if the document is non-empty, i.e., has at least one stroke.
 pCmdUI->Enable(!m_strokeList.IsEmpty());
}

void CScribbleDoc::OnUpdatePenThickOrThin(CCmdUI* pCmdUI)
{
 // Add check mark to Draw Thick Line menu item, if the current
 // pen width is "thick".
 pCmdUI->SetCheck(m_bThickPen);
}
```

## 本章回顧

這一章主要為 Scribble Step2 增加新的選單命令項。在這個過程中我們使用了工具列編輯器和 ClassWizard（或 Wizardbar）等工具。工具的使用很簡單，但是把訊息的處理常式加在什麼地方卻是關鍵。因此本章一開始先帶你深入探索 MFC 原始碼，了解訊息的遞送以及所謂 Message Map 背後的意義，並且也解釋了命令訊息(WM\_COMMAND) 特異的繞行路線及其原因。

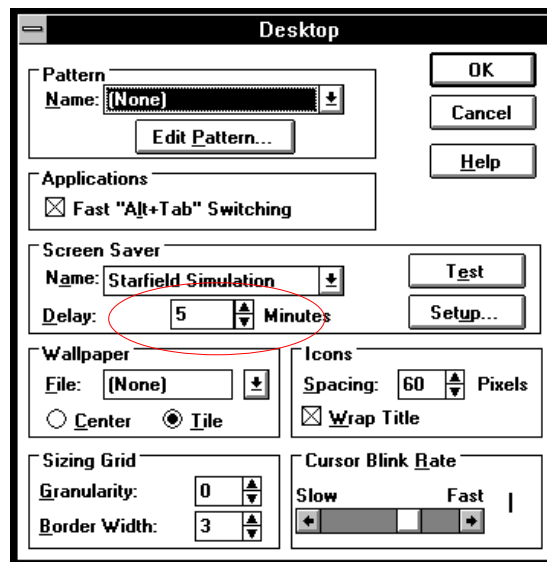
我在本章中挖出了許多 MFC 原始碼，希望藉由原始碼的自我說明能力，加深你對訊息映射與訊息繞行路徑的了解。這是對 MFC「知其所以然」的重要關鍵。這個知識基礎不會因為 MFC 的原始碼更動而更動，我要強調的，是其原理。



## MFC 與對話盒

上一章我們為 Scribble 新增了一個【Pen】選單，其中第二個命令項【Pen Width...】準備用來提供一個對話盒，讓使用者設定筆的寬度。每一線條都可以擁有自己的筆寬。原預設粗筆是 5 個圖素寬，細筆是 2 個圖素寬。

爲了這樣的目的，在對話盒中放個 Spin 控制元件是極佳的選擇。Spin 就是那種有著上下小三角形箭頭、可搭配一個文字顯示器的控制元件，有點像轉輪，用來選擇數字最合適：





但是，Scribble Step3 只是想示範如何在 MFC 程式中經由選單命令項喚起一個對話盒，並示範所謂的資料交換與資料檢驗（DDX/DDV）。所以，筆寬對話盒中只選用兩個小小的 Edit 控制元件而已。

本章還可以學習到如何利用對話盒編輯器設計對話盒的面板，並利用 ClassWizard 製作一個對話盒類別，定義訊息處理函式，把它們與對話盒「綁」在一塊兒。

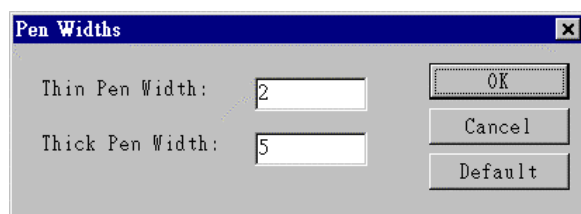


圖 10-1 【Pen Widths】對話盒

## 對話盒 編輯器

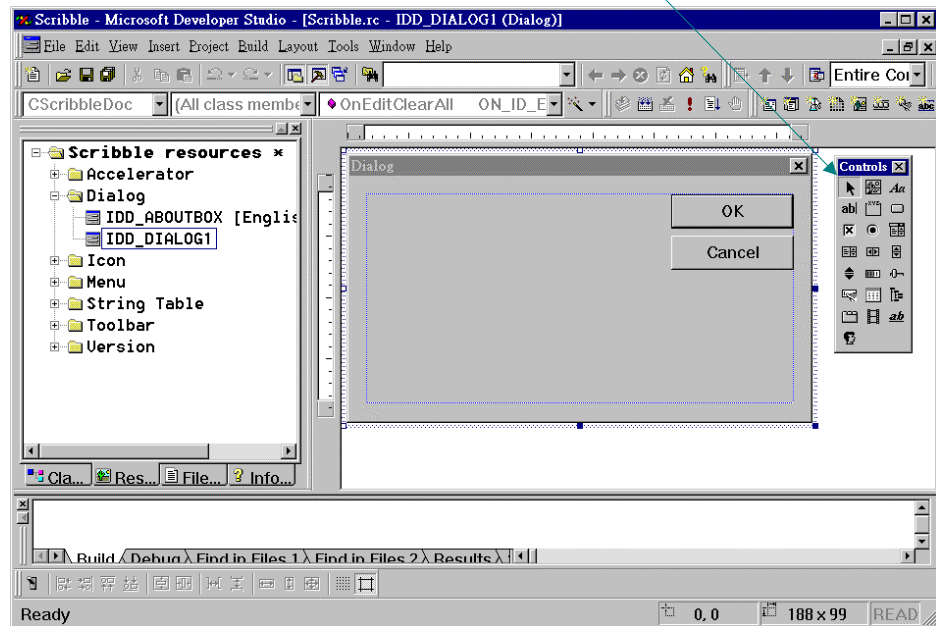
把對話盒函式拋在一旁，把所有程式煩惱拋在一旁，我們先享受一下 Visual C++ 整合環境中的對話盒編輯器帶來的對話盒面板（Dialog Template）設計快感。

設計對話盒面板，有兩個重要的步驟，第一是從工具箱中選擇控制元件（control，功能各異的小小零組件）加到對話盒中，第二是填寫此一控制元件的標題、ID、以及其他性質。

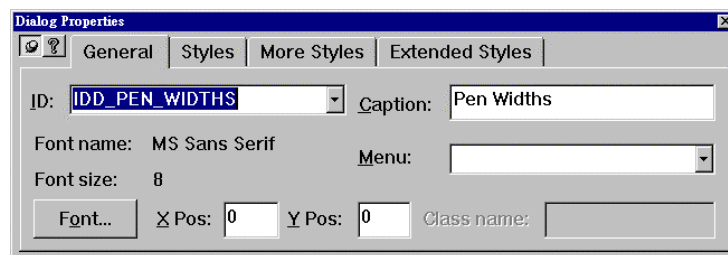
以下就是利用對話盒編輯器設計【Pen Widths】對話盒的過程。

- 在 Visual C++ 整合環境中選按【Insert/Resource】命令項，並在隨後來到的【Insert Resource】對話盒中，選擇【resource types】為 Dialog。
- 或是直接在 Visual C++ 整合環境中按下工具列的【New Dialog】按鈕。

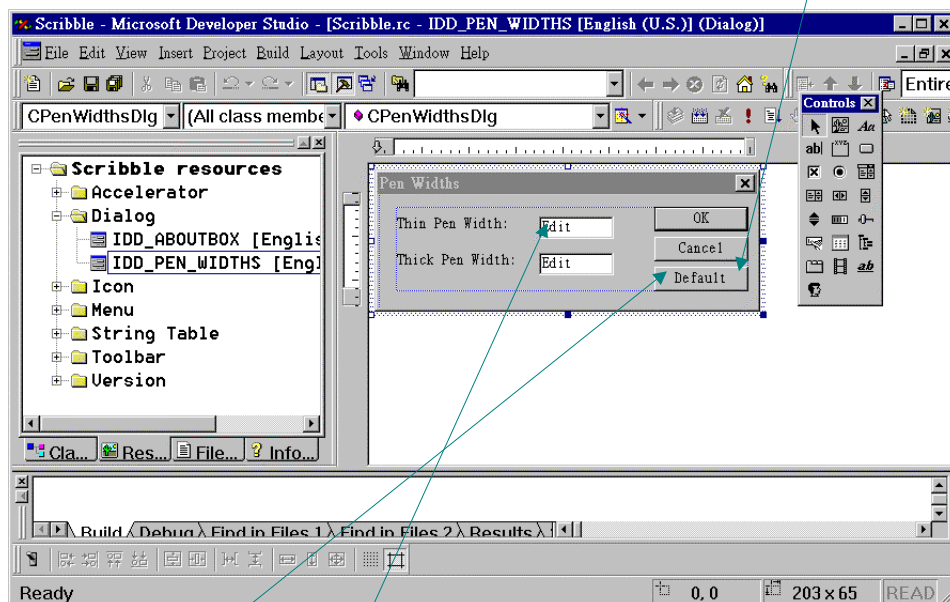
- Scribble.rc 檔會被打開，對話盒編輯器出現，自動給我們一個空白對話盒，內含兩個按鈕，分別是【OK】和【Cancel】。控制元件工具箱出現在畫面右側，內含許多控制元件。



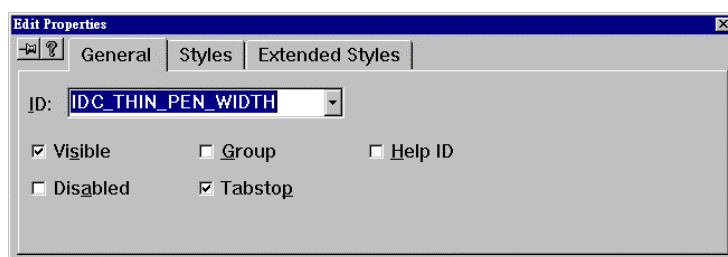
- 為了設定控制元件的屬性，必須用到【Dialog Properties】對話盒。如果它最初沒有出現，只要以右鍵選按對話盒的任何地方，就會跑出一份選單，再選擇其中的「Properties」，即會出現此對話盒。按下對話盒左上方的 push-pin 鈕（大頭針）可以常保它浮現為最上層視窗。現在把對話盒 ID 改為 `IDD_PEN_WIDTHS`，把標題改為 "Pen Widths"。



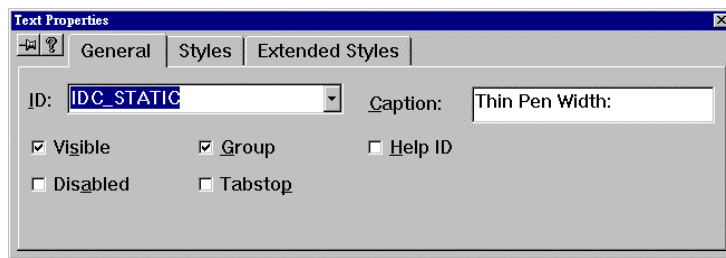
- 為對話盒加入兩個 Edit 控制元件，兩個 Static 控制元件，以及一個按鈕。



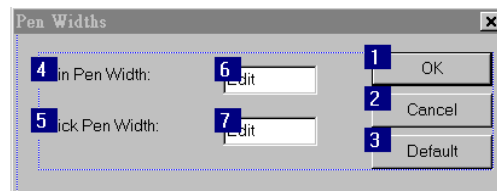
- 右鍵選按新增的按鈕，在 Property page 中把其標題改為 "Default"，並把 ID 改為 `IDC_DEFAULT_PEN_WIDTHS`。
- 右鍵選按第一個 Edit 控制元件，在 Property page 中把 ID 改為 `IDC_THIN_PEN_WIDTH`。以同樣的方式把第二個 Edit 控制元件的 ID 改為 `IDC_THICK_PEN_WIDTH`。



- 右鍵選按第一個 Static 控制元件，Property page 中出現其屬性，現在把文字內容改為 "Thin Pen Width: "。以同樣的方式把第二個 Static 控制元件的文字內容改為 "Thick Pen Width: "。不必在意 Static 控制元件的 ID 值，因為我們根本不可能在程式中用到 Static 控制元件的 ID。



- 調整每一個控制元件的大小位置，使之美觀整齊。
- 調整 tab order。所謂 tab order 是使用者在操作對話盒時，按下 Tab 鍵後，鍵盤輸入焦點在各個控制元件上的巡迴次序。調整方式是選按 Visual C++ 整合環境中的【Layout/Tab Order】命令項，出現帶有標號的對話盒如下，再依你所想要的次序以滑鼠點選一遍即可。



- 測試對話盒。選按 Visual C++ 整合環境中的【Layout/Test】命令項，出現運作狀態下的對話盒。你可以在這種狀態下測試 tab order 和預設按鈕（default button）。若欲退出，請選按【OK】或【Cancel】或按下 ESC 鍵。

注意：所謂 default button，是指與 <Enter> 鍵相通的那個按鈕。

所有調整都完成之後，存檔。於是 SCRIBBLE.RC 增加了下列內容（一個對話盒面板）：

```
IDD_PEN_WIDTHS DIALOG DISCARDABLE 0, 0, 203, 65
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Pen Widths"
FONT 8, "MS Sans Serif"
BEGIN
 DEFPUSHBUTTON "OK",IDOK,148,7,50,14
 PUSHBUTTON "Cancel",IDCANCEL,148,24,50,14
 PUSHBUTTON "Default",IDC_DEFAULT_PEN_WIDTHS,148,41,50,14
 LTEXT "Thin Pen Width:",IDC_STATIC,10,12,70,10
 LTEXT "Thick Pen Width:",IDC_STATIC,10,33,70,10
 EDITTEXT IDC_THIN_PEN_WIDTH,86,12,40,13,ES_AUTOHSCROLL
 EDITTEXT IDC_THICK_PEN_WIDTH,86,33,40,13,ES_AUTOHSCROLL
END
```

## 利用 ClassWizard 連接對話盒與其專屬類別

一旦完成了對話盒的外貌設計，再來就是設計其行為。我們有兩件事要做：

1. 從 MFC 的 *CDialog* 中衍生出一個類別，用來負責對話盒行為。
2. 利用 ClassWizard 把這個類別和先前你產生的對話盒資源連接起來。通常這意味著你必須宣告某些函式，用以處理你感興趣的對話盒訊息，並將對話盒中的控制元件對應到類別的成員變數上，這也就是所謂的 Dialog Data eXchange (DDX)。如果你對這些變數內容有任何「確認規則」的話，ClassWizard 也允許你設定之，這就是所謂的 Dialog Data Validation (DDV)。

注意：所謂「確認規則」是指對某些特殊用途的變數進行內容查驗工作。例如月份一定只可能在 1~12 之間，日期一定只可能在 1~31 之間，人名一定不會有數字夾雜其中，金錢數額不能夾帶文字，新竹的電話號碼必須是 03 開頭後面再加 7 位數... 等等等。

所有動作當然都可以手工完成，然而 ClassWizard 的表現非常好，讓我們快速又輕鬆地完成這些事樣。它可以為你的對話盒產生一個 .H 檔，一個 .CPP 檔，內有你的對話盒類別、函式骨幹、一個 Message Map、以及一個 Data Map。哎呀，我們又看到了新東西，稍後我會解釋所謂的 Data Map。

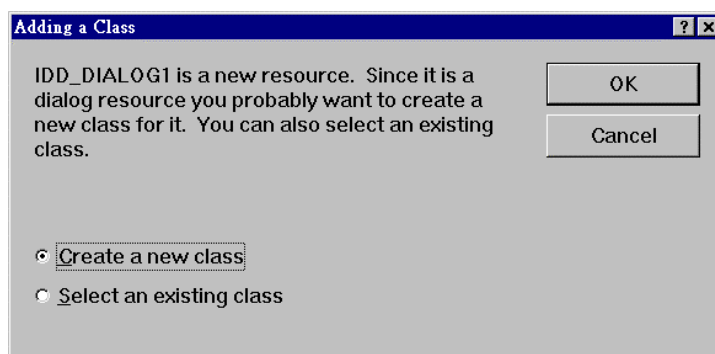
回憶 Scribble 誕生之初，程式中有一個 About 對話盒，寄生於 SCRIBBLE.CPP 中。AppWizard 並沒有詢問我們有關這個對話盒的任何意見，就自作主張地放了這些碼：

```
#0001 ///
#0002 // CAboutDlg dialog used for App About
#0003
#0004 class CAboutDlg : public CDialog
#0005 {
#0006 public:
#0007 CAboutDlg();
#0008
#0009 // Dialog Data
#0010 //{AFX_DATA(CAboutDlg)
#0011 enum { IDD = IDD_ABOUTBOX };
#0012 //}AFX_DATA
#0013
#0014 // ClassWizard generated virtual function overrides
#0015 //{AFX_VIRTUAL(CAboutDlg)
#0016 protected:
#0017 virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
#0018 //}AFX_VIRTUAL
#0019
#0020 // Implementation
#0021 protected:
#0022 //{AFX_MSG(CAboutDlg)
#0023 // No message handlers
#0024 //}AFX_MSG
#0025 DECLARE_MESSAGE_MAP()
#0026 };
#0027
#0028 CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
#0029 {
#0030 //{AFX_DATA_INIT(CAboutDlg)
#0031 //}AFX_DATA_INIT
#0032 }
#0033
#0034 void CAboutDlg::DoDataExchange(CDataExchange* pDX)
```

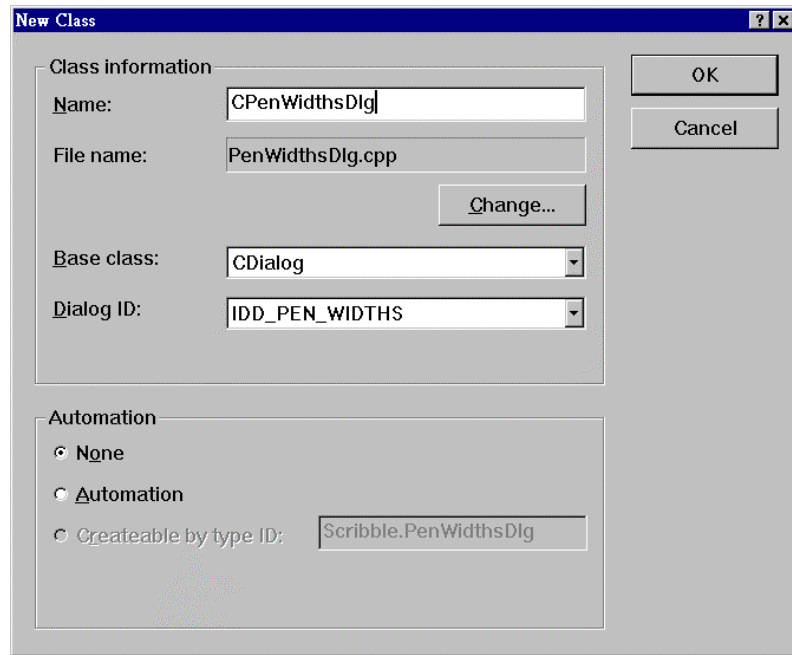
```
#0035 {
#0036 CDialog::DoDataExchange(pDX);
#0037 //{AFX_DATA_MAP(CAboutDlg)
#0038 //}AFX_DATA_MAP
#0039 }
#0040
#0041 BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
#0042 //{AFX_MSG_MAP(CAboutDlg)
#0043 // No message handlers
#0044 //}AFX_MSG_MAP
#0045 END_MESSAGE_MAP()
#0046
#0047 // App command to run the dialog
#0048 void CScribbleApp::OnAppAbout()
#0049 {
#0050 CAboutDlg aboutDlg;
#0051 aboutDlg.DoModal();
#0052 }
```

*CAboutDlg* 雖然衍生自 *CDialog*，但太簡陋，不符合我們新增的這個【Pen Width】對話盒所需，所以我們首先必須另為【Pen Width】對話盒產生一個類別，以負責其行徑。步驟如下：

- 接續剛才完成對話盒面板的動作，選按整合環境的【View/ClassWizard】命令項（或是直接在對話盒面板上快按兩下），進入 ClassWizard。這時候【Adding a Class】對話盒會出現，並以剛才的 *IDD\_PEN\_WIDTHS* 為新資源，這是因為 ClassWizard 知道你已在對話盒編輯器中設計了一個對話盒面板，卻還未設計其對應類別（整合環境就是這麼便利）。好，按下【OK】。



- 在【Create New Class】對話盒中設計新類別。鍵入 "CPenWidthsDlg" 做為類別名稱。請注意類別的基礎型態為 *CDialog*，因為 ClassWizard 知道目前是由對話盒編輯器過來：



- ClassWizard 把類別名稱再加上 .cpp 和 .h，作為預設檔名。毫無問題，因為 Windows 95 和 Windows NT 都支援長檔名。如果你不喜歡，按下上圖右側的【Change】鈕去改它。本例改用 PENDLG.CPP 和 PENDLG.H 兩個檔名。
- 按下上圖的【OK】鈕，於是類別產生，回到 ClassWizard 畫面。

這樣，我們就進賬了兩個新檔案：



## PENDLG.H

```
#0001 // PenDlg.h : header file
#0002 //
#0003
#0004 //////////////////////////////////////
#0005 // CPenWidthsDlg dialog
#0006
#0007 class CPenWidthsDlg : public CDialog
#0008 {
#0009 // Construction
#0010 public:
#0011 CPenWidthsDlg(CWnd* pParent = NULL); // standard constructor
#0012
#0013 // Dialog Data
#0014 //{AFX_DATA(CPenWidthsDlg)
#0015 enum { IDD = IDD_PEN_WIDTHS };
#0016 // NOTE: the ClassWizard will add data members here
#0017 //}AFX_DATA
#0018
#0019
#0020 // Overrides
#0021 // ClassWizard generated virtual function overrides
#0022 //{AFX_VIRTUAL(CPenWidthsDlg)
#0023 protected:
#0024 virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
#0025 //}AFX_VIRTUAL
#0026
#0027 // Implementation
#0028 protected:
#0029
#0030 // Generated message map functions
#0031 //{AFX_MSG(CPenWidthsDlg)
#0032 afx_msg void OnDefaultPenWidths();
#0033 //}AFX_MSG
#0034 DECLARE_MESSAGE_MAP()
#0035 };
```

## PENDLG.CPP

```
#0001 // PenDlg.cpp : implementation file
#0002 //
#0003
#0004 #include "stdafx.h"
#0005 #include "Scribble.h"
```

```
#0006 #include "PenDlg.h"
#0007
#0008 #ifdef _DEBUG
#0009 #define new DEBUG_NEW
#0010 #undef THIS_FILE
#0011 static char THIS_FILE[] = __FILE__;
#0012 #endif
#0013
#0014 //
#0015 // CPenWidthsDlg dialog
#0016
#0017
#0018 CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
#0019 : CDialog(CPenWidthsDlg::IDD, pParent)
#0020 {
#0021 //{{AFX_DATA_INIT(CPenWidthsDlg)
#0022 // NOTE: the ClassWizard will add member initialization here
#0023 //}}AFX_DATA_INIT
#0024 }
#0025
#0026
#0027 void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
#0028 {
#0029 CDialog::DoDataExchange(pDX);
#0030 //{{AFX_DATA_MAP(CPenWidthsDlg)
#0031 // NOTE: the ClassWizard will add DDX and DDV calls here
#0032 //}}AFX_DATA_MAP
#0033 }
#0034
#0035
#0036 BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
#0037 //{{AFX_MSG_MAP(CPenWidthsDlg)
#0038 ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
#0039 //}}AFX_MSG_MAP
#0040 END_MESSAGE_MAP()
#0041
#0042 //
#0043 // CPenWidthsDlg message handlers
#0044
#0045 void CPenWidthsDlg::OnDefaultPenWidths()
#0046 {
#0047 // TODO: Add your control notification handler code here
#0048
#0049 }
```

稍早我曾提過，ClassWizard 會為我們做出一個 Data Map。此一 Data Map 將放在 *DoDataExchange* 函式中。目前 Data Map 還沒有什麼內容，*CPenWidthsDlg* 的 Message Map 也是空的，因為我們還未透過 ClassWizard 加料呢。

請注意，*CPenWidthsDlg* 建構式會先引發基礎類別 *CDialog* 的建構式，後者會產生一個 modal 對話盒。 *CDialog* 建構式的兩個參數分別是對話盒 ID 以及父視窗指標：

```
#0018 CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
#0019 : CDialog(CPenWidthsDlg::IDD, pParent)
#0020 {
#0021 //{{AFX_DATA_INIT(CPenWidthsDlg)
#0022 // NOTE: the ClassWizard will add member initialization here
#0023 //}}AFX_DATA_INIT
#0024 }
```

ClassWizard 幫我們把 *CPenWidthsDlg::IDD* 塞給第一個參數，這個值定義於 PENDING.H 的 AFX\_DATA 區中，其值為 *IDD\_PEN\_WIDTHS*：

```
#0013 // Dialog Data
#0014 //{{AFX_DATA(CPenWidthsDlg)
#0015 enum { IDD = IDD_PEN_WIDTHS };
#0016 // NOTE: the ClassWizard will add data members here
#0017 //}}AFX_DATA
```

也就是【Pen Widths】對話盒資源的 ID：

```
// in SCRIBBLE.RC
IDD_PEN_WIDTHS DIALOG DISCARDABLE 0, 0, 203, 65
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Pen Widths"
FONT 8, "MS Sans Serif"
BEGIN
 DEFPUSHBUTTON "OK",IDOK,148,7,50,14
 PUSHBUTTON "Cancel",IDCANCEL,148,24,50,14
 PUSHBUTTON "Default",IDC_DEFAULT_PEN_WIDTHS,148,41,50,14
 LTEXT "Thin Pen Width:",IDC_STATIC,10,12,70,10
 LTEXT "Thick Pen Width:",IDC_STATIC,10,33,70,10
 EDITTEXT IDC_THIN_PEN_WIDTH,86,12,40,13,ES_AUTOHSCROLL
 EDITTEXT IDC_THICK_PEN_WIDTH,86,33,40,13,ES_AUTOHSCROLL
END
```

對話盒類別 *CPenWidthsDlg* 因此才有辦法取得「RC 檔中的對話盒資源」。

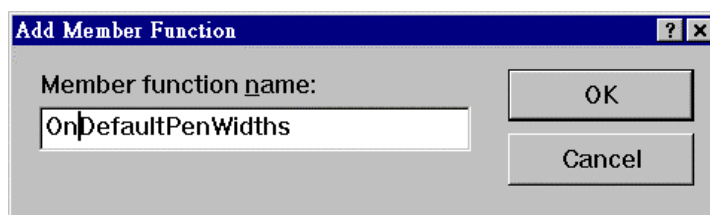
## 對話盒的訊息處理函式

*CDialog* 本就定義有兩個按鈕【OK】和【Cancel】，【Pen Widths】對話盒又新增一個【Default】鈕，當使用者按下此鈕時，粗筆與細筆都必須回復為預設寬度（分別是 5 個圖素和 2 個圖素）。那麼，我們顯然有兩件工作要完成：

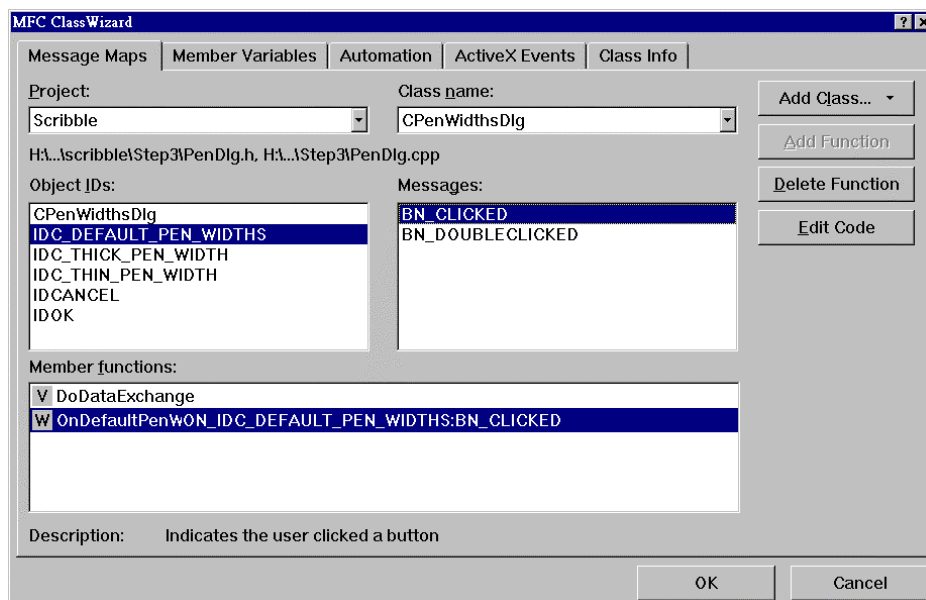
1. 在 *CPenWidthsDlg* 中增加兩個變數，分別代表粗筆與細筆的寬度。
2. 在 *CPenWidthsDlg* 中增加一個函式，負責【Default】鈕被按下後的動作。

以下是 ClassWizard 的操作步驟（增加一個函式）：

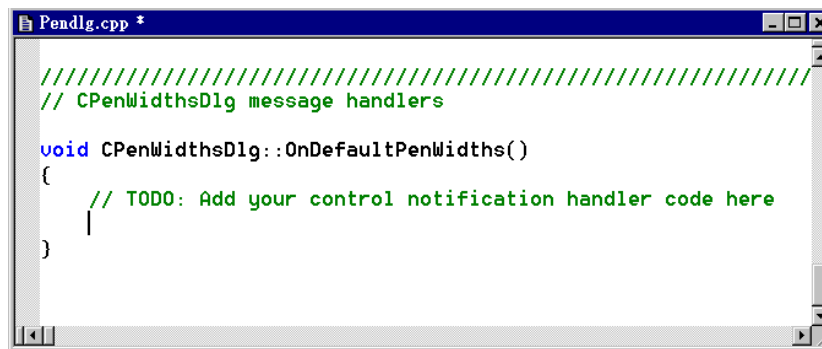
- 進入 ClassWizard，選擇【Message Maps】附頁，再選擇【Class name】清單中的 *CPenWidthsDlg*。
- 左側的【Object IDs】清單列出對話盒中各個控制元件的 ID。請選擇其中的 *IDC\_DEFAULT\_PEN\_WIDTHS*（代表【Default】鈕）。
- 在右側的【Messages】中選擇 *BN\_CLICKED*。這和我們在前兩章的經驗不同，如今我們處理的是控制元件，它所產生的訊息是特別的一類，稱為 Notification 訊息，這種訊息是控制元件用來通知其父視窗（通常是個對話盒）某些狀況發生了，例如 *BN\_CLICKED* 表示按鈕被按下。至於不同的 Notification 所代表的意義，畫面最下方的 "Description" 會顯示出來。
- 按下【Add Function】鈕，接受預設的 *OnDefaultPenWidths* 函式（也可以改名）：



- 現在，【Member Functions】清單中出現了新函式，以及它所對映之控制元件與 Notification 訊息。



- 按下【Edit Code】鈕，游標落在 *OnDefaultPenWidths* 函式身上，我們看到以下內容：



上述動作對原始碼造成的影響是：

```
// in PENDLG.H
class CPenWidthsDlg : public CDialog
{
protected:
 afx_msg void OnDefaultPenWidths();
 ...
};

// in PENDLG.CPP
BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
 ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
END_MESSAGE_MAP()

void CPenWidthsDlg::OnDefaultPenWidths()
{
 // TODO : Add your control notification handler here
}
```

### MFC 中各式各樣的 MAP

如果你以為 MFC 中只有 Message Map 和 Data Map，那你就錯了。另外還有一個 Dispatch Map，使用於 OLE Automation，下面是其形式：

```
DECLARE_DISPATCH_MAP() // .H 檔中的巨集，宣告 Dispatch Map。

BEGIN_DISPATCH_MAP(CClickDoc, CDocument) // .CPP 檔中的 Dispatch Map
 //{AFX_DISPATCH_MAP(CClickDoc)
 DISP_PROPERTY(CClickDoc, "text", m_str, VT_BSTR)
 DISP_PROPERTY_EX(CClickDoc, "x", GetX, SetX, VT_I2)
 DISP_PROPERTY_EX(CClickDoc, "y", GetY, SetY, VT_I2)
 //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

此外還有 Event Map，使用於 OLE Custom Control（也就是 OCX），下面是其形式：

DECLARE\_EVENT\_MAP() // .H 檔中的巨集，宣告 Event Map。

```
BEGIN_EVENT_MAP(CSmileCtrl, COleControl) // .CPP 檔中的 Event Map
 //{AFX_EVENT_MAP(CSmileCtrl)
 EVENT_CUSTOM("Inside", FireInside, VTS_I2 VTS_I2)
 EVENT_STOCK_CLICK()
 //}}AFX_EVENT_MAP
END_EVENT_MAP()
```

至於 Message Map，我想你一定已經很熟悉了：

DECLARE\_MESSAGE\_MAP()// .H 檔中的巨集，宣告 Message Map。

```
BEGIN_MESSAGE_MAP(CScribDoc, CDocument) // .CPP 檔中的 Message Map
 //{AFX_MSG_MAP(CScribDoc)
 ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
 ON_COMMAND(ID_PEN_THICK_OR_THIN, OnPenThickOrThin)
 ON_COMMAND(ID_PEN_WIDTHS, OnPenWidths)
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

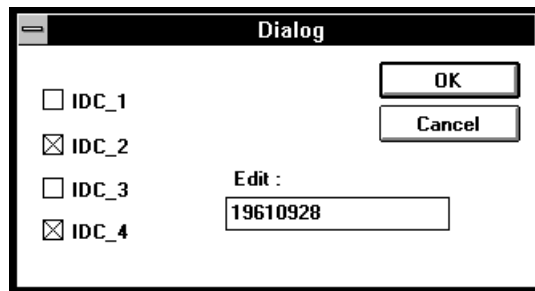
MFC 所謂的 Map，其實就是一種類似表格的東西，它的背後是什麼？可能是一個巨大的資料結構（例如 Message Map）。最和其他 Map 形式不同的，就屬 Data Map 了，它的形式是：

```
//{AFX_DATA_MAP(CPenWidthsDlg) // .CPP 檔中的 Data Map
DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
//}}AFX_DATA_MAP
```

針對同一個資料目標（成員變數），Data Map 之中每組有兩筆記錄，一筆負責 DDX，一筆負責 DDV。

## 對話盒資料的採與查核 (DDX & DDV)

在解釋 DDX/DDV 的來龍去脈之前，我想先描述一下 SDK 程式處理對話盒資料的作法。如果你設計一個對話盒如下圖：



當【OK】鈕被按下，程式應該一一取得按鈕狀態以及 Edit 內容：

```
char _OpenName[128];
GetDlgItemText(hwndDlg, IDC_EDIT, _OpenName, 128);

If (IsDlgButtonChecked(hwndDlg, IDC_1))
 ...;
If (IsDlgButtonChecked(hwndDlg, IDC_2))
 ...;
If (IsDlgButtonChecked(hwndDlg, IDC_3))
 ...;
If (IsDlgButtonChecked(hwndDlg, IDC_4))
 ...;
// hwndDlg 代表對話盒的視窗 handle
```

雖然 Windows 95 和 Windows NT 有所謂的通用型對話盒（Common Dialog，第 6 章末尾曾介紹過），某些個標準對話盒的設計因而非常簡單，但非標準的對話盒還是得像上面那樣自己動手。

MFC 的方式就簡單多了。它提供的 DDX（X 表示 eXchange），允許程式員事先設定控制元件與變數的對應關係。我們不但可以令控制元件的內容一有改變就自動傳送到變數去，也可以藉 MFC 提供的 DDV（V 表示 Validation）設定欄位的合理範圍。如果使用



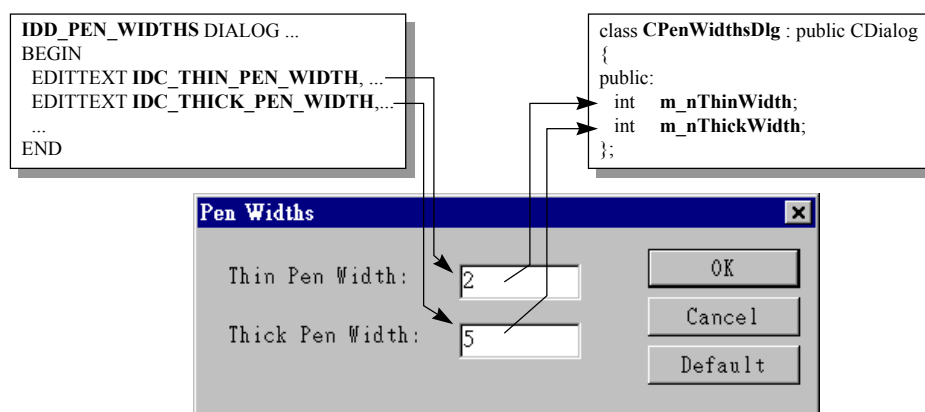
者在欄位上鍵入超出合理範圍的數字，就會在按下【OK】後出現類似以下的畫面：



資料的查核 (Data Validation) 其實是一件瑣碎又耗人力的事情，各式各樣的資料都應該要檢查其合理範圍，程式才算面面俱到。例如日期欄位絕不能允許 12 以上的月份以及 31 以上的日子(如果程式還能自動檢查 2 月份只有 28 天而遇閏年有 29 天那就更棒了)；金額欄位裡絕不能允許文字出現，電話號碼欄位一定只有 9 位(至少臺灣目前是如此)。爲了解決這些瑣碎又累人的工作，市售有一些程式庫，專門做資料查核工作。

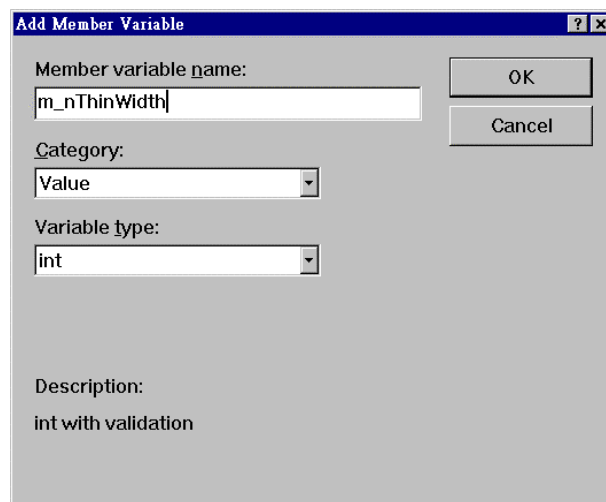
然而不要對 MFC 的 DDV 能力期望過高，稍後你就會看到，它只能滿足最低層次的要求而已。就 DDV 而言，Borland 的 OWL 表現較佳。

現在我打算以兩個成員變數映射到對話盒上的兩個 Edit 欄位。我希望當使用者按下【OK】鈕，第一個 Edit 欄位的內容自動儲存到 *m\_nThinWidth* 變數中，第二個 Edit 欄位的內容自動儲存到 *m\_nThickWidth* 變數中：

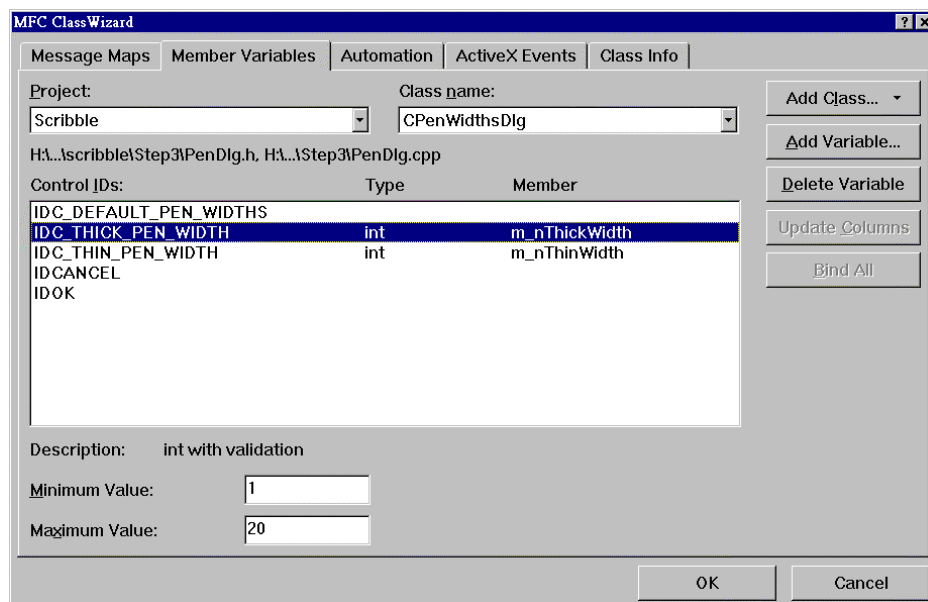


下面是 ClassWizard 的操作步驟(為對話盒類別增加兩個成員變數,並設定 DDX / DDV):

- 進入 ClassWizard, 選擇【Member Variables】附頁, 再選擇 *CPenWidthsDlg*。  
對話盒中央部份有一大塊區域用來顯示控制元件與變數間的對映關係(見下一頁圖)。
- 選擇 *IDC\_THIN\_PEN\_WIDTH*, 按下【Add Variable...】鈕, 出現對話盒如下。
- 鍵入變數名稱為 *m\_nThinWidth*。
- 選擇變數型別為 *int*。



- 按下【OK】鍵, 於是 ClassWizard 為 *CPenWidthsDlg* 增加了一個變數 *m\_nThinWidth*。
- 在 ClassWizard 對話盒最下方(見下一頁圖)填入變數的數值範圍, 以為 DDV 之用。
- 重複前述步驟, 為 *IDC\_THICK\_PEN\_WIDTH* 也設定一個對應變數, 範圍也是 1~20。



上述動作影響我們的程式碼如下：

```
class CPenWidthsDlg : public CDialog
{
// Dialog Data
//{{AFX_DATA(CPenWidthsDlg)
enum { IDD = IDD_PEN_WIDTHS };
int m_nThinWidth;
int m_nThickWidth;
//}}AFX_DATA
...

CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
: CDialog(CPenWidthsDlg::IDD, pParent)
{
 m_nThickWidth = 0;
 m_nThinWidth = 0;
 ...
}

void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
 CDialog::DoDataExchange(pDX);
 //{{AFX_DATA_MAP(CPenWidthsDlg)
```

```

 DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
 DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
 DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
 DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
 //}}AFX_DATA_MAP
}

```

只要資料「有必要」在成員變數與控制元件之間搬移，Framework 就會自動呼叫 *DoDataExchange*。我所說的「有必要」是指，對話盒初次顯示在螢幕上，或是使用者按下【OK】離開對話盒等等。*CPenWidthsDlg::DoDataExchange* 由一組一組的 DDX/DDV 函式完成之。先做 DDX，然後做 DDV，這是遊戲規則。如果你純粹借助 ClassWizard，就不必在意此事，如果你要自己動手完成，就得遵循規則。

該是完成上一節的 *OnDefaultPenWidths* 的時候了。當【Default】鈕被按下，Framework 會呼叫 *OnDefaultPenWidths*，我們應該在此設定粗筆細筆兩種寬度的預設值：

```

void CPenWidthsDlg::OnDefaultPenWidths()
{
 m_nThinWidth = 2;
 m_nThickWidth = 5;
 UpdateData(FALSE); // causes DoDataExchange()
 // bSave=FALSE means don't save from screen,
 // rather, write to screen
}

```

### MFC 中各式各樣的 DDX\_ 函式

如果你以為 MFC 對於對話盒的照顧，只有 DDX 和 DDV，那你就又錯了，另外還有一個 DDP，使用於 OLE Custom Control（也就是 OCX）的 Property page 中，下面是它的形式：

```

//{{AFX_DATA_MAP(CSmilePropPage)
 DDP_Text(pDX, IDC_CAPTION, m_caption, _T("Caption"));
 DDX_Text(pDX, IDC_CAPTION, m_caption);
 DDP_Check(pDX, IDC_SAD, m_sad, _T("sad"));
 DDX_Check(pDX, IDC_SAD, m_sad);
//}}AFX_DATA_MAP

```

什麼是 Property page？這是最新流行（Microsoft 強力推銷？）的介面。這種介面用來解決過於擁擠的對話盒。ClassWizard 就有四個 Property page，我們又稱為 tag（附頁）。擁有 property page 的對話盒稱為 property sheet，也就是 tagged dialog（帶有附頁的對話盒）。

## 如何喚起對話盒

【Pen Widths】對話盒是一個所謂的 Modal 對話盒，意思是除非它關閉（結束），否則它會緊抓住這個程式的控制權，但不影響其他程式。相對於 Modal 對話盒，有一種 Modeless 對話盒就不會影響程式其他動作的進行；通常你在文書處理軟體中看到的文字搜尋對話盒就是 Modeless 對話盒。

過去，MFC 有兩個類別，分別負責 Modal 對話盒和 Modeless 對話盒，它們是 *CModalDialog* 和 *CDialog*。如今已經合併為一，就是 *CDialog*。不過為了回溯相容，MFC 有這麼一個定義：

```
#define CModalDialog Cdialog
```

要做出 Modal 對話盒，只要呼叫 *CDialog::DoModal* 即可。

我們希望 Step3 的命令項【Pen/Pen Widths】被按下時，【Pen Widths】對話盒能夠執行起來。要喚起此一對話盒，得做到兩件事情：

1. 產生一個 *CPenWidthsDlg* 物件，負責管理對話盒。
2. 顯示對話盒視窗。這很簡單，呼叫 *DoModal* 即可辦到。

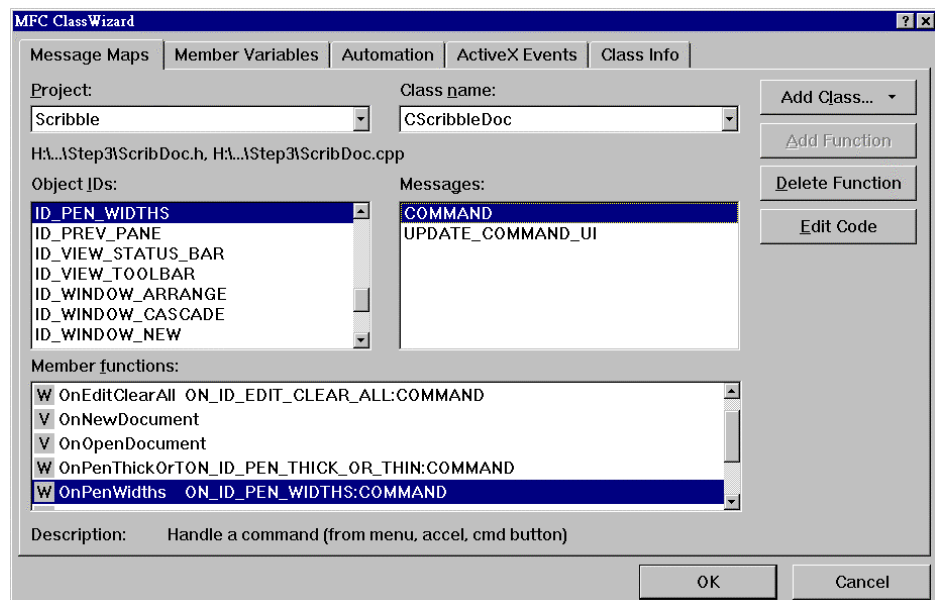
爲了把命令訊息連接上 *CPenWidthsDlg*，我們再次使用 ClassWizard，這一次要爲 *CScribbleDoc* 加上一個命令處理常式。爲什麼選擇在 *CScribbleDoc* 而不是其它類別中處理此一命令呢？因爲不論是粗筆或細筆，乃至於目前正在使用的筆，其寬度都被記錄在 *CScribbleDoc* 中成爲它的一個成員變數：

```
// in SCRIBDOC.H
class CScribbleDoc : public CDocument
{
protected:
 UINT m_nPenWidth; // current user-selected pen width
 UINT m_nThinWidth;
 UINT m_nThickWidth;
 ...
}
```

所以由 *CScribDoc* 負責喚起對話盒，接受筆寬設定，是很合情合理的事。

如果命令訊息處理常式名為 *OnPenWidths*，我們希望在這個函式中先喚起對話盒，由對話盒取得粗筆和細筆的寬度，然後再把這兩個值設定給 *CScribbleDoc* 中的兩個對應變數。下面是設計步驟。

- 執行 ClassWizard，選擇【Message Map】附頁，並選擇 *CScribbleDoc*。
- 在【Object IDs】清單中選擇 *ID\_PEN\_WIDTHS*。
- 在【Messages】清單中選擇 *COMMAND*。
- 按下【Add Function】鈕並接受 ClassWizard 給予的函式名稱 *OnPenWidths*。



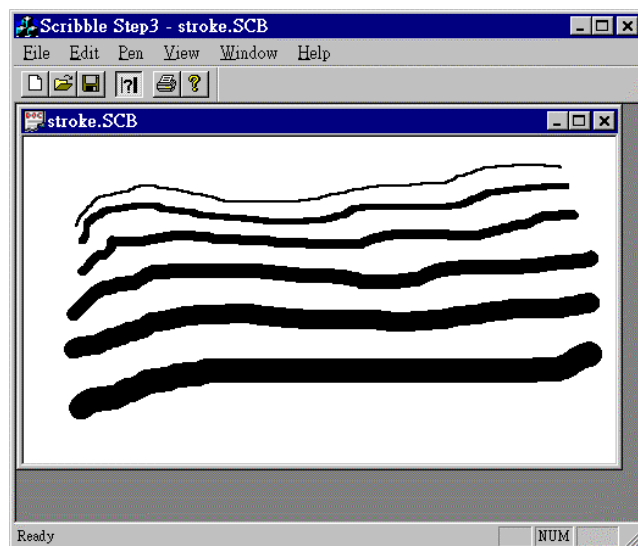
- 按下【Edit Code】鈕，游標落在 *OnPenWidths* 函式內，鍵入以下內容：

```
// SCRIBDOC.CPP
#include "pendlg.h"
...
void CScribbleDoc::OnPenWidths()
{
 CPenWidthsDlg dlg;
 // Initialize dialog data
 dlg.m_nThinWidth = m_nThinWidth;
 dlg.m_nThickWidth = m_nThickWidth;

 // Invoke the dialog box
 if (dlg.DoModal() == IDOK)
 {
 // retrieve the dialog data
 m_nThinWidth = dlg.m_nThinWidth;
 m_nThickWidth = dlg.m_nThickWidth;

 // Update the pen that is used by views when drawing new strokes,
 // to reflect the new pen width definitions for "thick" and "thin".
 ReplacePen();
 }
}
```

現在，Scribble Step3 全部完成，製作並測試之。



## 本章回顧

上一章我們為 `Scribble` 加上三個新的選單命令項。其中一個命令項【`Pen/Pen Widths...`】將引發對話盒，這個目標在本章實現。

製作對話盒，我們需要為此對話盒設計面板（`Dialog Template`），這可藉 `Visual C++` 整合環境之對話盒編輯器之助完成。我們還需要一個衍生自 `CDialog` 的類別（本例為 `CPenWidthsDlg`）。`ClassWizard` 可以幫助我們新增類別，並增加該類別的成員變數，以及設定對話盒之 `DDX/DDV`。以上都是透過 `ClassWizard` 以滑鼠點點選選而完成，過程中不需要寫任何一行程式碼。

所謂 `DDX` 是讓我們把對話盒類別中的成員變數與對話盒中的控制元件產生關聯，於是當對話盒結束時，控制元件的內容會自動傳輸到這些成員變數上。

所謂 `DDV` 是允許我們設定對話盒控制元件的內容型態以及資料（數值）範圍。

對話盒的寫作，在 `MFC` 程式設計中輕鬆無比。你可以嘗試練習一個比較複雜的對話盒。





第 11 章

## View 功能之加強 與 重繪效率之提昇

前面數章中，我們已經看到了 View 如何扮演 Document 與使用者之間的媒介：View 顯示 Document 的資料內容，並且接受滑鼠在視窗上的行為（左鍵按下、放開、滑鼠移動），視為對 Document 的操作。

Scribble 可以對同一份 Document 產生一個以上的 Views，這是 MDI 程式的「天賦」MDI 程式標準的【Window/New Window】表單項目就是為達此目標而設計的。但有一個缺點還待克服，那就是你在視窗 A 的繪圖動作不能即時影響視窗 B，也就是說它們之間並沒有所謂的同步更新 -- 即使它們是同一份資料的一體兩面！

Scribble Step4 解決上述問題。主要關鍵在於想辦法通知所有相同血源（同一份 Document）的各兄弟（各個 Views），讓它們一起行動。但卻因此必須考慮這個問題：如果使用者的一個滑鼠動作引發許多許多的程式繪圖動作，那麼「同步更新」的繪圖效率就變得非常重要。因此在考量如何加強顯示能力時，我們就得設計所謂的「必要繪圖區」，也就是所謂的 Invalidate Region，或稱「不再適用的區域」或「重繪區」。每當使用者增加新的線條，Scribble Step4 便把「包圍該線條之最小四方形」設定為「必要繪圖區」。為了記錄這項資料，從 Step1 延用至今的 Document 資料結構必須有所改變。

Step4 的同步更新，是以一筆畫為單位，而非以一個點為單位。換句話說在一筆畫未完

成之前，不打算讓同源的多個 View 視窗同步更新 -- 那畢竟太傷效率了。

Scribble Step4 的另一項改善是為 Document Frame 視窗增加垂直和水平捲軸，並且示範一種所謂的分裂視窗（Splitter window），如圖 11-1。這種視窗的主要功能是當使用者欲對文件做一體兩面（或多面）觀察時，各個觀察子視窗可以集中在一個大的母視窗中。在這裡，子視窗被稱為「窗口」（pane）。

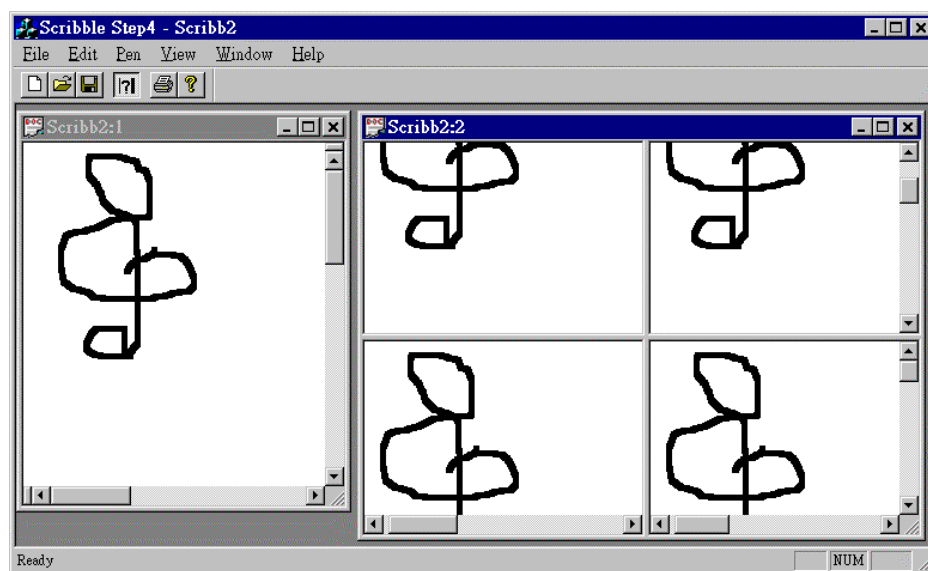


圖 11-1 Scribble Step4，同源（同一份 Document）之各個 View 視窗具備同步更新的能力。Document Frame 視窗具備分裂窗口與捲軸。

## 同時修改多個 Views：UpdateAllViews 和 OnUpdate

在 Scribble View 上繪圖，然後選按【Window/New Window】，會蹦出另一個新的 View，其內的圖形與前一個 View 相同。這兩個 Views 就是同一份文件的兩個「觀景窗」。新視窗的產生導至 *WM\_PAINT* 產生，於是 *OnDraw* 發生效用，把文件內容畫出來：



圖 11-2 一份 Document 連結兩個 Views，沒有同步修正畫面。

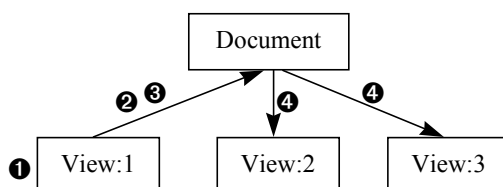
但是，此後如果你在 *Scrib1:1* 視窗上繪圖而未縮放其尺寸的話（也就是不產生 *WM\_PAINT*），*Scrib1:2* 視窗內看不到後續繪圖內容。我們並不希望如此，不幸的是上一章的 *Scribble Step3* 正是如此。

不能同步更新的關鍵在於，沒有人通知所有的兄弟們（Views）一起動手 -- 動手呼叫 *OnDraw*。你是知道的，只有當 *WM\_PAINT* 產生，*OnDraw* 才會被呼叫。因此，解決方式是對每一個兄弟都發出 *WM\_PAINT*，或任何其他方法 -- 只要能通知到就好。也就是說，讓附屬於同一 Document 的所有 Views 都能夠立即反應 Document 內容變化的方法就是，始作俑者（被使用者用來修改 Document 內容的那個 View）必須想辦法通知其他兄弟。

經由 *CDocument::UpdateAllViews*，MFC 提供了這樣的一個標準機制。

讓所有的 Views 同步更新資料的關鍵在於兩個函式：

1. *CDocument::UpdateAllViews* - 這個函式會巡訪所有隸屬同一份 Document 的各個 Views，找到一個就通知一個，而所謂「通知」就是呼叫其 *OnUpdate* 函式。
2. *CView::OnUpdate* - 我們可以在這個函式中設計繪圖動作。或許是全部重繪（這比較笨一點），或許想辦法只繪必要的一小部份（這比較聰明一些）。



- 1 使用者在 View:1 做動作（View 扮演使用者介面的第一線）。
- 2 View:1 呼叫 *GetDocument*，取得 Document 指標，更改資料內容。
- 3 View:1 呼叫 Document 的 *UpdateAllViews*。
- 4 View:2 和 View:3 的 *OnUpdate* 一一被呼叫起來，這是更新畫面的時機。

如果想讓繪圖程序聰明一些，不要每次都全部重繪，而是只擇「必須重繪」的區域重繪，那麼 *OnUpdate* 需要被提示什麼是「必須重繪的區域」，這就必須借助於 *UpdateAllViews* 的參數：

```
virtual void UpdateAllViews(CView* pSender,
 LPARAM lHint,
 CObject* pHint);
```

- 第一個參數代表發出此一通牒的始作俑者。這個參數的設計無非是希望避免重複而無謂的通牒，因為始作俑者自己已經把畫面更新過了（在滑鼠訊息處理常式中），不需要再被通知。

- 後面兩個參數 *lHint* 和 *pHint* 是所謂的提示參數（Hint），它們會被傳送到同一 Document 所對應的每一個 Views 的 *OnUpdate* 函式去。*lHint* 可以是一些特殊的提示值，*pHint* 則是一個衍生自 *CObject* 的物件指標。靠著設計良好的「提示」，*OnUpdate* 才有機會提高繪圖效率。要不然直接通知 *OnDraw* 就好了，也不需要再搞出一個 *OnUpdate*。

另一方面，*OnUpdate* 收到三個參數（由 *CDocument::UpdateAllViews* 發出）：

```
virtual void OnUpdate(CView* pSender,
 LPARAM lHint,
 CObject* pHint);
```

因此，一旦 Document 資料改變，我們應該呼叫 *CDocument::UpdateAllViews* 以通知所有相關的 Views。而在 *CMyView::OnUpdate* 函式中我們應該以效率為第一考量，利用參數中的 *hint* 設定重繪區，使後續被喚起的 *OnDraw* 有最快的工作速度。注意，通常你不應該在 *OnUpdate* 中執行繪圖動作，所有的繪圖動作最好都應該集中在 *OnDraw*；你在 *OnUpdate* 函式中的行為應該是計算哪一塊區域需要重繪，然後呼叫 *CWnd::InvalidateRect*，發出 *WM\_PAINT* 讓 *OnDraw* 去畫圖。

結論是，改善同步更新以及繪圖效率的前置工作如下：

1. 定義 *hint* 的資料型態，用以描述已遭修改的資料區域。
2. 當使用者透過 View 改變了 Document 內容，程式應該產生一個 *hint*，描述此一修改，並以它做為參數，呼叫 *UpdateAllViews*。
3. 改寫 *CMyView::OnUpdate*，利用 *hint* 設計高效率繪圖動作，使 *hint* 描述區之外的區域不要重畫。

## 在 View 中定義一個 hint

以 *Scribble* 為例，當使用者加上一段線條，如果我們計算出包圍此一線條之最小四方形，那麼只有與此四方形有交集的其他線條才需要重畫，如圖 11-3。因此在 Step4 中把 *hint* 設計為 *RECT* 型態，差堪為用。

效率考量上，當然我們還可以精益求精取得各線條與此四方形的交點，然後只重繪四方形內部的那一部份即可，但這麼做是否動用太多計算，是否使工程太過複雜以至於不划算，你可得謹慎評估。

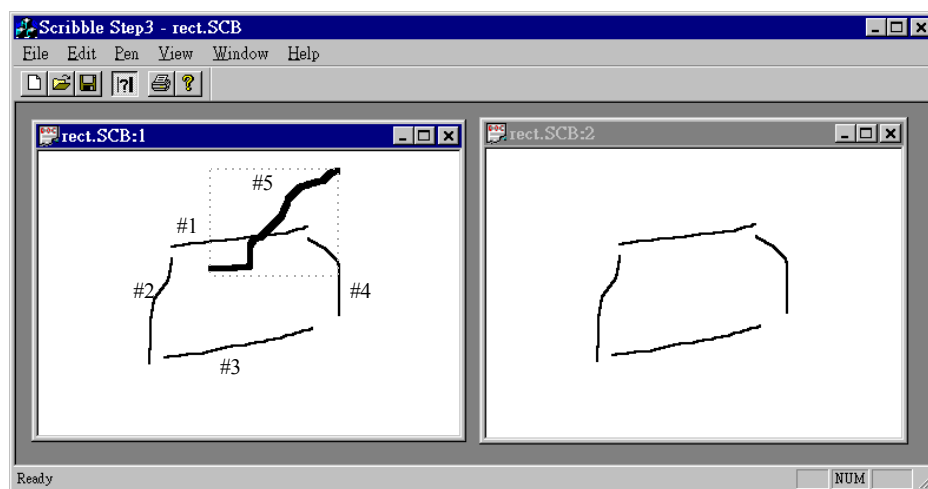


圖 11-3 在 `rect.SCB:1` 視窗中新增一線條 #5，那麼，只有與虛線四方形（此四方形將 #5 包起來）有交集之其他線條，也就是 #1 和 #4，才有必要在 `rect.SCB:2` 視窗中重畫。

前面曾說 `UpdateAllViews` 函式的第三個參數必須是 `CObject` 衍生物件之指標。由於本例十分單純，與其爲了 `Hint` 特別再設計一個類別，勿寧在 `CStroke` 中增加一個變數（事實上是一個 `CRect` 物件），用以表示前述之 `hint` 四方形，那麼每一條線條就外罩了一個小小的四方殼。但是我們不能把 `CRect` 物件指標直接當做參數來傳，因爲 `CRect` 並不衍生自 `CObject`。稍後我會說明該怎麼做。

可以預期的是，日後一定需要一一從每一線條中取出這個「外圍四方形」，所以現在先宣告並定義一個名爲 `GetBoundingRect` 的函式。另外再宣告一個 `FinishStroke` 函式，其作用主要是計算這四方形尺寸。稍後我會詳細解釋這些函式的行爲。

```

// in SCRIBBLEDRAW.H
class CStroke : public CObject
{
...
public:
 UINT m_nPenWidth;
 CDWordArray m_pointArray;
 CRect m_rectBounding; // smallest rect that surrounds all
 // of the points in the stroke

public:
 CRect& GetBoundingRect() { return m_rectBounding; }
 void FinishStroke();
...
};

```

我想你早已熟悉了 Scribble Document 的資料結構。爲了應付 Step4 的需要，現在每一線條必須多一個成員變數，那是一個 *CRect* 物件，如圖 11-4 所示。

#### CStroke Step4 Document :

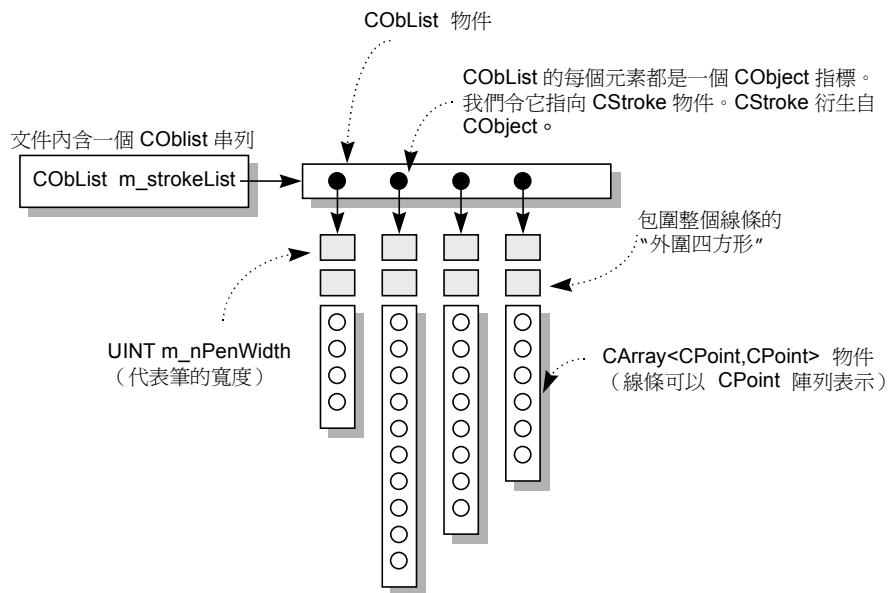


圖 11-4 CStrokeDoc 物件中的各項資料



設計觀念分析完畢，現在動手吧。我們必須在 SCRIBDOC.CPP 中的 Document 初始化動作以及檔案讀寫動作都加入 *m\_rectBounding* 這個新成員：

```
// in SCRIBDOC.CPP
IMPLEMENT_SERIAL(CStroke, CObject, 2) // 注意 schema no. 改變為 2

CStroke::CStroke(UINT nPenWidth)
{
 m_nPenWidth = nPenWidth;
 m_rectBounding.SetRectEmpty();
}

void CStroke::Serialize(CArchive& ar)
{
 if (ar.IsStoring())
 {
 ar << m_rectBounding;
 ar << (WORD)m_nPenWidth;
 m_pointArray.Serialize(ar);
 }
 else
 {
 ar >> m_rectBounding;
 WORD w;
 ar >> w;
 m_nPenWidth = w;
 m_pointArray.Serialize(ar);
 }
}
```

如果我們改變了檔案讀寫的格式，我們就應該改變 schema number（可視為版本號碼）。由於 Scribble 資料檔（.SCB）格式改變了，多了一個 *m\_rectBounding*，所以我們在 *IMPLEMENT\_SERIAL* 巨集中改變 Document 的 Schema no.，以便讓不同版本的 Scribble 程式識得不同版本的文件檔。如果你以 Scribble Step3 讀取 Step4 所產生的文件，會因為 Schema 號碼的不同而得到這樣的訊息：



我們還需要一個函式，用以計算「線條之最小外包四方形」，這件事情當然是在線條完成後進行之，所以此一函式命名為 *FinishStroke*。每當一筆畫結束（滑鼠左鍵放開，產生 *WM\_LBUTTONDOWN*），*OnLButtonDown* 就呼叫 *FinishStroke* 讓它計算邊界。計算方法很直接，取出線條中的座標點，比大小而已：

```
// in SCRIBDOC.CPP
void CStroke::FinishStroke()
{
 // 計算外圍四方形。爲了靈巧而高效率的重繪動作，這是必要的。

 if (m_pointArray.GetSize()==0)
 {
 m_rectBounding.SetRectEmpty();
 return;
 }
 CPoint pt = m_pointArray[0];
 m_rectBounding = CRect(pt.x, pt.y, pt.x, pt.y);

 for (int i=1; i < m_pointArray.GetSize(); i++)
 {
 // 如果點在四方形之外，那麼就將四方形膨脹，以含入該點。
 pt = m_pointArray[i];
 m_rectBounding.left = min(m_rectBounding.left, pt.x);
 m_rectBounding.right = max(m_rectBounding.right, pt.x);
 m_rectBounding.top = min(m_rectBounding.top, pt.y);
 m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
 }

 // 在四方形之外再加上筆的寬度。
 m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
 return;
}
```

## 把 hint 傳給 OnUpdate

下一步驟是想辦法把 hint 交給 *UpdateAllViews*。讓我們想想什麼時候 *Scribble* 的資料開始產生改變？答案是滑鼠左鍵按下時！或許你會以為要在 *OnLButtonDown* 中呼叫 *CDocument::UpdateAllViews*。這個猜測的論點可以成立但是結果錯誤，因為左鍵按下後還

有一連串的滑鼠軌跡移動，每次移動都導至資料改變，新的點不斷被加上去。如果我們等左鍵放開，線條完成，再來呼叫 *UpdateAllViews*，事情會比較單純。因此 *Scribble Step4* 是在 *OnButtonUp* 中呼叫 *UpdateAllViews*。當然我們現在就可以預想得到，一筆畫完成之前，同一 *Document* 的其他 *Views* 沒有辦法即時顯示最新資料。

下面是 *OnButtonUp* 的修改內容：

```
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
 ...
 m_pStrokeCur->m_pointArray.Add(point);

 // 已完成加點的動作，現在可以計算外圍四方形了
 m_pStrokeCur->FinishStroke();

 // 通知其他的 views，使它們得以修改它們的圖形。
 pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
 ...
 return;
}
```

程式邏輯至為簡單，唯一需要說明的是 *UpdateAllViews* 的第三個參數 (hint)，原本我們只需設此參數為 *m\_rectBounding*，即可滿足需求，但 MFC 規定，第三參數必須是一個 *CObject* 指標，而 *CRect* 並不衍生自 *CObject*，所以我們乾脆就把目前正作用中的整個線條（也就是 *m\_pStrokeCur*）傳過去算了。*CStroke* 的確是衍生自 *CObject*！

```
// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
protected:
 CStroke* m_pStrokeCur; // the stroke in progress
 ...
};

// in SCRIBBLEVIEW.CPP
void CScribbleView::OnLButtonDown(UINT, CPoint point)
{
 ...
 m_pStrokeCur = GetDocument()->NewStroke();
 m_pStrokeCur->m_pointArray.Add(point);
}
```

```

 ...
}
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
 ...
 m_pStrokeCur->m_pointArray.Add(point);
 ...
}
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
 ...
 m_pStrokeCur->m_pointArray.Add(point);
 m_pStrokeCur->FinishStroke();
 pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
 ...
}

```

*UpdateAllViews* 會巡訪 *CScribbleDoc* 所連接的每一個 Views（始作俑者那個 View 除外），呼叫它們的 *OnUpdate* 函式，並把 *hint* 做為參數之一傳遞過去。

## 利用 hint 增加重繪效率

預設情況下，*OnUpdate* 所收到的無效區（也就是重繪區），是 Document Frame 視窗的整個內部。但誰都知道，原已存在而且沒有變化的圖形再重繪也只是浪費時間而已。上一節你已看到 *Scribble* 每加上一整個線條，就在 *OnLButtonUp* 函式中呼叫 *UpdateAllViews* 函式，並且把整個線條（內含其四方邊界）傳過去，因此我們可以想辦法在 *OnUpdate* 中重繪這個四方形小區域就好。

話說回來，如何能夠只重繪一個小區域就好呢？我們可以一一取出 Document 中每一線條的四方邊界，與新線條的四方邊界比較，若有交點就重繪該線條。*CRect* 有一個 *IntersectRect* 函式正適合用來計算四方形交集。

但是有一點必須注意，繪圖動作不是集中在 *OnDraw* 嗎？因此 *OnUpdate* 和 *OnDraw* 之間的分工有必要釐清。前面數個版本中這兩個函式的動作是：

- *OnUpdate* - 啥也沒做。事實上 *CScribbleView* 原本根本沒有改寫此一函式。
- *OnDraw* - 迭代取得 *Document* 中的每一線條，並呼叫 *CStroke::DrawStroke* 將線條繪出。

*Scribble Step4* 之中，這兩個函式的動作如下：

- *OnUpdate* - 判斷 *Framework* 傳來的 *hint* 是否為 *CStroke* 物件。如果是，設定無效區域（重繪區域）為該線條的外圍四方形；如果不是，設定無效區域為整個視窗區域。「設定無效區域」（也就是呼叫 *CWnd::InvalidateRect*）會引發 *WM\_PAINT*，於是引發 *OnDraw*。
- *OnDraw* - 迭代取得 *Document* 中的每一線條，並呼叫 *CStroke::GetBoundingRect* 取線條之外圍四方形，如果與「無效區域」有交集，就呼叫 *CStroke::DrawStroke* 繪出整個線條。如果沒有交集，就跳過不畫。

以下是新增的 *OnUpdate* 函式：

```
// in SCRIBVW.CPP
void CScribbleView::OnUpdate(CView* /* pSender */, LPARAM /* lHint */,
 CObject* pHint)
{
 // Document 通知 View 說，某些資料已經改變了

 if (pHint != NULL)
 {
 if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
 {
 // hint 提示我們哪一線條被加入（或被修改），所以我們要把該線條的
 // 外圍四方形設為無效區。
 CStroke* pStroke = (CStroke*)pHint;
 CClientDC dc(this);
 OnPrepareDC(&dc);
 CRect rectInvalid = pStroke->GetBoundingRect();
 dc.LPtoDP(&rectInvalid);
 InvalidateRect(&rectInvalid);
 return;
 }
 }
 // 如果我們不能解釋 hint 內容（也就是說它不是我們所預期的
```

```

// CStroke 物件)，那就讓整個視窗重繪吧（把整個視窗設為無效區）。
Invalidate(TRUE);
return;
}

```

為什麼 *OnUpdate* 之中要呼叫 *OnPrepareDC*？這關係到捲軸，我將在介紹分裂視窗時再說明。另，*GetBoundingRect* 動作如下：

```
CRect& GetBoundingRect() { return m_rectBounding; }
```

*OnDraw* 函式也為了高效能重繪動作之故，做了以下修改。陰影部份是與 *Scribble Step3* 不同之處：

```

// SCRIBVW.CPP
void CScribbleView::OnDraw(CDC* pDC)
{
 CScribbleDoc* pDoc = GetDocument();
 ASSERT_VALID(pDoc);

 // 取得視窗的無效區。如果是在列印狀態情況下，則取
 // printer DC 的截割區 (clipping region)。
 CRect rectClip;
 CRect rectStroke;
 pDC->GetClipBox(&rectClip);

 // 注意：CScrollView::OnPrepare 已經在 OnDraw 被呼叫之前先一步
 // 調整了 DC 原點，用以反應出目前的捲動位置。關於 CScrollView，
 // 下一節就會提到。

 // 呼叫 CStroke::DrawStroke 完成無效區中各線條的繪圖動作
 CTypedPtrList<CObList, CStroke*>& strokeList = pDoc->m_strokeList;
 POSITION pos = strokeList.GetHeadPosition();
 while (pos != NULL)
 {
 CStroke* pStroke = strokeList.GetNext(pos);
 rectStroke = pStroke->GetBoundingRect();
 if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
 continue;
 pStroke->DrawStroke(pDC);
 }
}

```

## 可捲動的視窗：CScrollView

到目前為止我們還沒有辦法觀察一張比視窗還大的圖，因為我們沒有捲軸。

一個 View 視窗沒有捲軸，是很糟糕的事，因為通常 Document 範圍大而觀景窗範圍小。我們不能老讓 Document 與 View 視窗一樣大。一個具備捲軸的 View 視窗更具有「觀景窗」的意義。

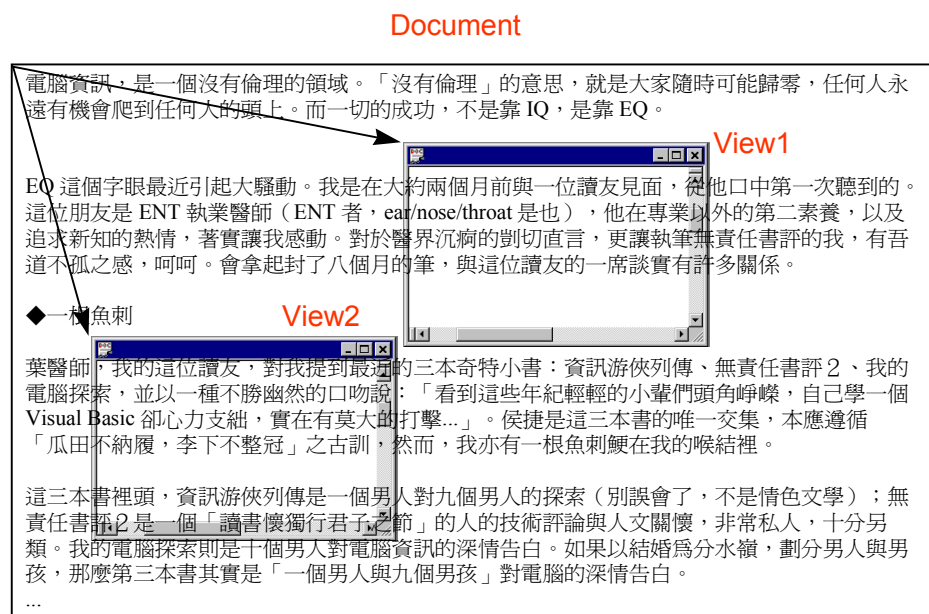


圖 11-5a 一個具備捲軸的 View 視窗更具「觀景窗」的意義

如果你有 SDK 程式設計經驗，你就會知道設計一個可捲動的視窗是多麼煩瑣的事（文字的捲動還算好，圖形的捲動更慘）。MFC 當然不可能對此一般性功能坐視不管，事實上它已設計好一個 CScrollView，其中的捲軸有即時捲動（邊拉捲動桿邊跑）的效果。

基本上要使 View 視窗具備捲軸，你必須做到下列事情：

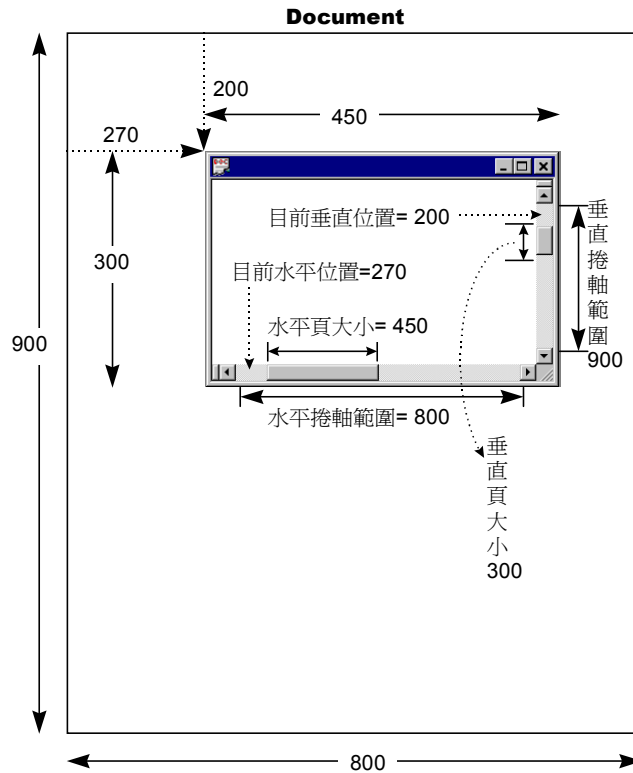


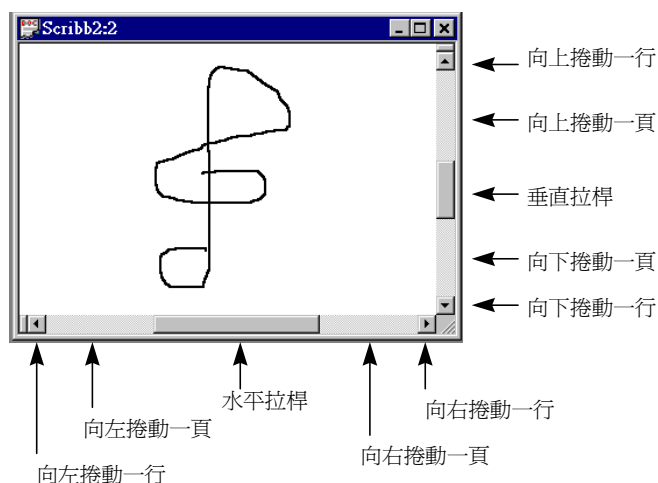
圖 11-5b 捲軸 View 視窗與 Document 之間的關係

- 定義 Document 大小。如果沒有大小，Framework 就沒有辦法計算捲軸尺寸，以及捲動比例。這個大小可以是常數，也可以是個儲存在每一 Document 中的變數，隨著執行時期變動。
- 以 *CScrollView* 取代 *CView*。
- 只要 Document 的大小改變，就將尺寸傳給 *CScrollView* 的 *SetScrollSizes* 函式。如果程式設定 Document 為固定大小（本例就是如此），那麼當然只要一開始做一次捲軸設定動作即可。
- 注意裝置座標（視窗座標）與邏輯座標（Document 座標）的轉換。關於此點稍後另有說明。



Application Framework 對捲軸的貢獻是：

- 處理 *WM\_HSCROLL* 和 *WM\_VSCROLL* 訊息，並相對地捲動 Document（其實是移動 View 落在 Document 上的位置）以及移動「捲軸拉桿」（所謂的 thumb）。拉桿位置可以表示出目前視窗中顯示的區域在整個 Document 的位置。如果你按下捲軸兩端箭頭，捲動的幅度是一行（line），至於一行代表多少，由程式員自行決定。如果你按下的是拉桿兩旁的桿子，捲動的幅度是一頁（page），一頁到底代表多少，也由程式員自行決定。



- 視窗一旦被放大縮小，立刻計算視窗的寬度高度與捲軸長度的比例，以重新設定捲動比例，也就是一行或一頁的大小。

以下分四個步驟修改 Scribble 原始碼：

- 1 定義 Document 的大小。我們的作法是設定一個變數，代表大小，並在 Document 初始化時設定其值，此後全程不再改變（以簡化問題）。MFC 中有一個 *CSize* 很適合當作此一變數型態。這個成員變數在文件進行檔案讀寫（*Serialization*）時也應該併入文件內容中。回憶一下，上一章講到筆寬時，由於每一線條有自己的一個寬度，所以筆寬資料應該在 *CStroke::Serialize* 中讀寫，現在我們所討論的文件大小卻是屬於整份文件的資料，所以應該在 *CScribbleDoc::Serialize* 中讀寫：

```

// in SCRIBBLEDOC.H
class CScribbleDoc : public CDocument
{
protected:
 CSize m_sizeDoc;
public:
 CSize GetDocSize() { return m_sizeDoc; }
 ...
};

// in SCRIBBLEDOC.CPP
void CScribbleDoc::InitDocument()
{
 m_bThickPen = FALSE;
 m_nThinWidth = 2; // default thin pen is 2 pixels wide
 m_nThickWidth = 5; // default thick pen is 5 pixels wide
 ReplacePen(); // initialize pen according to current width

 // 預設 Document 大小為 800 x 900 個螢幕圖素
 m_sizeDoc = CSize(800,900);
}

void CScribbleDoc::Serialize(CArchive& ar)
{
 if (ar.IsStoring())
 {
 ar << m_sizeDoc;
 }
 else
 {
 ar >> m_sizeDoc;
 }
 m_strokeList.Serialize(ar);
}

```

**2** 將 *CScribbleView* 的父類別由 *CView* 改變為 *CScrollView*。同時準備改寫其虛擬函式 *OnInitialUpdate*，為的是稍後我們要在其中，根據 *Document* 的大小，設定捲動範圍。

```

// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
public:
 virtual void OnInitialUpdate();
 ...
}

```

```
};

// in SCRIBBLEVIEW.CPP
IMPLEMENT_DYNCREATE(CScribbleView, CScrollView)

BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
 ...
END_MESSAGE_MAP()
```

**3** 改寫 *OnInitialUpdate*，在其中設定捲軸範圍。這個函式的被呼叫時機是在 View 第一次附著到 Document 但尚未顯現時，由 Framework 呼叫之。它會呼叫 *OnUpdate*，不帶任何 Hint 參數（於是 *lHint* 是 0 而 *pHint* 是 *NULL*）。如果你需要做任何「只做一次」的初始化動作，而且初始化時需要 Document 的資料，那麼在這裡做就最合適了：

```
// in SCRIBVW.CPP
void CScribbleView::OnInitialUpdate()
{
 SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
 // 這是 CScrollView 的成員函式。
}
```

*SetScrollSizes* 總共有四個參數：

- `int nMapMode`：代表映像模式（Mapping Mode）
- `SIZE sizeTotal`：代表文件大小
- `const SIZE& sizePage`：代表一頁大小（預設是文件大小的 1/10）
- `const SIZE& sizeLine`：代表一行大小（預設是文件大小的 1/100）

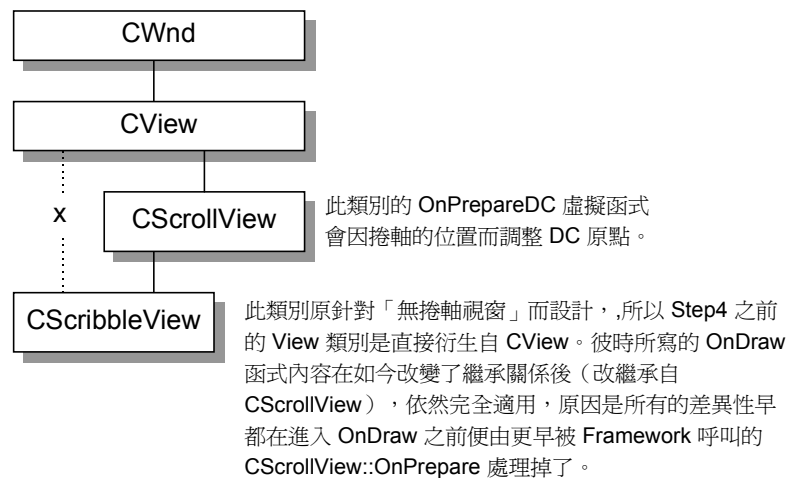
本例的文件大小是固定的。另一種比較複雜的情況是可變大小，那麼你就必須在文件大小改變之後立刻呼叫 *SetScrollSizes*。

視窗上增加捲軸並不會使 View 的 *OnDraw* 負擔加重。我們並不因為捲軸把觀察鏡頭移動到 Document 的中段或尾段，而就必須在 *OnDraw* 中重新計算繪圖原點與平移向量，原因是繪圖座標與我們所使用的 DC 有關。當捲軸移動了 DC 原點，*CScrollView* 自動

會做調整，讓資料的某一部份顯示而某一部份隱藏。

讓我做更詳細的說明。「GDI 原點」是 DC（註）的重要特徵，許許多多 CDC 成員函式的繪圖結果都會受它的影響。如果我們想在繪圖之前（也就是進入 *OnDraw* 之前）調整 DC，我們可以改寫虛擬函式 *OnPrepareDC*，因為 Framework 是先呼叫 *OnPrepareDC*，然後才呼叫 *OnDraw* 並把 DC 傳進去。好，視窗由無捲軸到增設捲軸的過程中，之所以不必修改 *OnDraw* 函式內容，就是因為 *CScrollView* 已經改寫了 *CView* 的 *OnPrepareDC* 虛擬函式。Framework 先呼叫 *CScrollView::OnPrepareDC* 再呼叫 *CScribbleView::OnDraw*，所有因為捲軸而必須做的特別處理都已經在進入 *OnDraw* 之前完成了。

注意上面的敘述，別把 *CScrollView* 和 *CScribbleView* 混淆了。下圖是個整理。



DC 就是 Device Context，在 Windows 中凡繪圖動作之前一定要先獲得一個 DC，它可能代表螢幕，也可能代表一個視窗，或一塊記憶體，或印表機...。DC 中有許多繪圖所需的元素，包括座標系統（映像模式）、原點、繪圖工具（筆、刷、顏色...）等等。它還連接到低階的輸出裝置驅動程式。由於 DC，我們在程式中對螢幕作畫和對印表機作畫的動作才有可能完全相同。

**4 修正滑鼠座標。**雖說 *OnDraw* 不必因為座標原點的變化而有任何改變，但是幕後出力的 *CScrollView::OnPrepareDC* 卻不知道什麼是 Windows 訊息！這話看似牛頭和馬嘴，但我一點你就明白了。*CScrollView::OnPrepareDC* 雖然能夠因著捲軸行為而改變 GDI 原點，但「改變 GDI 原點」這個動作卻不會影響你所接收的 *WM\_LBUTTONDOWN* 或 *WM\_LBUTTONUP* 或 *WM\_MOUSEMOVE* 的座標值，原因是 Windows 訊息並非 DC 的一個成份。因此，我們作畫的基礎，也就是滑鼠移動產生的軌跡點座標，必須由「以視窗繪圖區左上角為原點」的視窗座標系統，改變為「以文件左上角為原點」的邏輯座標系統。文件中儲存的，也應該是邏輯座標。

下面是修改座標的程式動作。其中呼叫的 *OnPrepareDC* 是哪一個類別的成員函式？想想看，*CScribbleView* 衍生自 *CScrollView*，而我們並未在 *CScribbleView* 中改寫此一函式，所以程式中呼叫的是 *CScrollView::OnPrepareDC*。

```
// in SCRIBVW.CPP
void CScribbleView::OnLButtonDown(UINT, CPoint point)
{
 // 由於 CScrollView 改變了 DC 原點和映像模式，所以我們必須先把
 // 裝置座標轉換為邏輯座標，再儲存到 Document 中。
 CClientDC dc(this);
 OnPrepareDC(&dc);
 dc.DPtoLP(&point);

 m_pStrokeCur = GetDocument()->NewStroke();
 m_pStrokeCur->m_pointArray.Add(point);

 SetCapture(); // 抓住滑鼠
 m_ptPrev = point; // 做為直線繪圖的第一個點

 return;
}
void CScribbleView::OnLButtonUp(UINT, CPoint point)
{
 ...
 if (GetCapture() != this)
 return;

 CScribbleDoc* pDoc = GetDocument();

 CClientDC dc(this);
```

```

 OnPrepareDC(&dc); // 設定映像模式和 DC 原點
 dc.DPtoLP(&point);
 ...
 }
void CScribbleView::OnMouseMove(UINT, CPoint point)
{
 ...
 if (GetCapture() != this)
 return;

 CClientDC dc(this);
 OnPrepareDC(&dc);
 dc.DPtoLP(&point);

 m_pStrokeCur->m_pointArray.Add(point);
 ...
}

```

除了上面三個函式，還有什麼函式牽扯到座標？是的，線條四週有一個外圍四方形，那將在 *OnUpdate* 中出現，也必須做座標系統轉換：

```

void CScribbleView::OnUpdate(CView* /* pSender */, LPARAM /* lHint */,
 CObject* pHint)
{
 if (pHint != NULL)
 {
 if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
 {
 // hint 的確是一個 CStroke 物件。現在將其外圍四方形設為重繪區
 CStroke* pStroke = (CStroke*)pHint;
 CClientDC dc(this);
 OnPrepareDC(&dc);
 CRect rectInvalid = pStroke->GetBoundingRect();
 dc.LPtoDP(&rectInvalid);
 InvalidateRect(&rectInvalid);
 return;
 }
 }
 // 無法識別 hint, 只好假設整個畫面都需重繪。
 Invalidate(TRUE);
 return;
}

```

注意，上面的 *LPointDP* 所受參數竟然不是 *CPoint\**，而是 *CRect\**，那是因為 *LPointDP* 有多載函式（overloaded function），既可接受點，也可接受四方形。*DPointLP* 也有類似的多載能力。

線條的外圍四方形還可能出現在 *CStroke::FinishStroke* 中，不過那裡只是把線條陣列中的點拿出來比大小，決定外圍四方形罷了；而你知道，線條陣列的點已經在加入時做過座標轉換了（分別在 *OnLButtonDown*、*OnMouseMove*、*OnLButtonUp* 函式中的 *AddPoint* 動作之前）。

至此，Document 的資料格式比起 Step1，有了大幅的變動。讓我們再次分析文件檔的格式，以期獲得更深入的認識與印證。我以圖 11-6a 為例，共四條線段，寬度分別是 2, 5, 10, 20（十進位）。分析內容顯示在圖 11-6b。

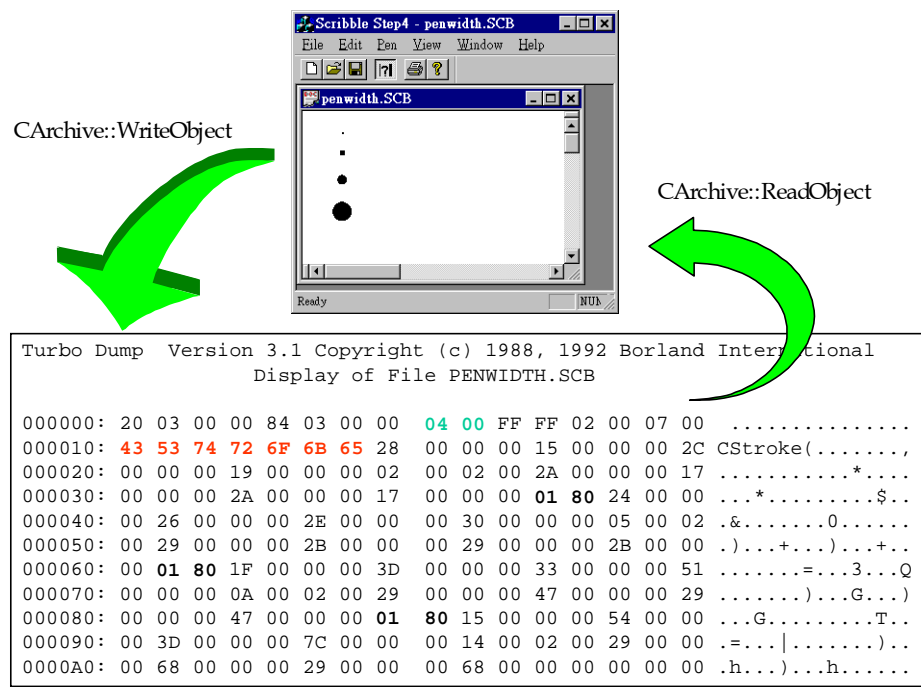


圖 11-6a 四條線段的圖形與文件檔傾印碼。

| 數值 (hex)             | 說明 (共 173 bytes)            |
|----------------------|-----------------------------|
| 00000320             | Document 寬度 (800)           |
| 00000384             | Document 高度 (900)           |
| 0004                 | 表示此檔有四個 <i>COBList</i> 元素   |
| FFFF                 | FFFF 亦即 -1，表示 New Class Tag |
| 0002                 | Scheme no.，代表 Document 版本號碼 |
| 0007                 | 表示後面接著的「類別名稱」有 7 個字元        |
| 43 53 74 72 6F 6B 65 | 類別名稱 "CStroke" 的 ASCII 碼    |
| 00000028 00000015    | 外圍四方形的左上角座標 (膨脹一個筆寬)        |
| 0000002C 00000019    | 外圍四方形的右下角座標 (膨脹一個筆寬)        |
| 0002                 | 第一條線條的寬度                    |
| 0002                 | 第一條線條的點數                    |
| 0000002A,00000017    | 第一條線條的第一個點座標                |
| 0000002A,00000017    | 第一條線條的第二個點座標                |
| 8001                 | 表示接下來的物件仍舊使用舊的類別            |
| 00000024 00000026    | 外圍四方形的左上角座標 (膨脹一個筆寬)        |
| 0000002E 00000030    | 外圍四方形的右下角座標 (膨脹一個筆寬)        |
| 0005                 | 第二條線條的寬度                    |
| 0002                 | 第二條線條的點數                    |
| 00000029,0000002B    | 第二條線條的第一個點座標                |
| 00000029,0000002B    | 第二條線條的第二個點座標                |
| 8001                 | 表示接下來的物件仍舊使用舊的類別            |
| 0000001F 0000003D    | 外圍四方形的左上角座標 (膨脹一個筆寬)        |
| 00000033 00000051    | 外圍四方形的右下角座標 (膨脹一個筆寬)        |
| 000A                 | 第三條線條的寬度                    |
| 0002                 | 第三條線條的點數                    |
| 00000029,00000047    | 第三條線條的第一個點座標                |
| 00000029,00000047    | 第三條線條的第二個點座標                |



| 數值 (hex)          | 說明 (共 173 bytes)     |
|-------------------|----------------------|
| 8001              | 表示接下來的物件仍舊使用舊的類別     |
| 00000015 00000054 | 外圍四方形的左上角座標 (膨脹一個筆寬) |
| 0000003D 0000007C | 外圍四方形的右下角座標 (膨脹一個筆寬) |
| 0014              | 第四條線條的寬度             |
| 0002              | 第四條線條的點數             |
| 00000029 00000068 | 第四條線條的第一個點座標         |
| 00000029 00000068 | 第四條線條的第二個點座標         |

圖 11-6b 文件檔 (圖 11-6a) 的分析

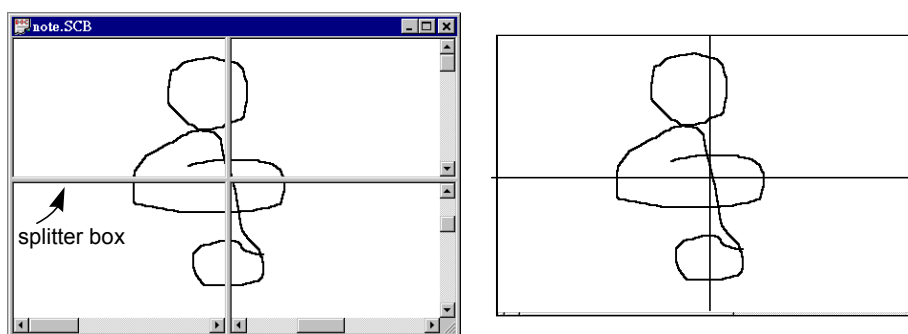
## 大視窗叫的小窗口：Splitter

MDI 程式的標準功能是允許你為同一份 Document 開啓一個以上的 Views。這種情況類似我們以多個觀景窗觀看同一份資料。我們可以開啓任意多個 Views，各有捲軸，那麼我們就可以在螢幕上同時觀察一份資料的不同區域。這許多個 View 視窗各自獨立運作，因此它們的觀看區可能互相重疊。

如果這些隸屬同一 Document 的 Views 能夠結合在一個大視窗之內，又各自有獨立的行為（譬如說有自己的捲軸），似乎可以帶給使用者更好的感覺和更方便的使用，不是嗎？

### 分裂視窗的功能

把 View 做成所謂的「分裂視窗 (splitter)」是一種不錯的想法。這種視窗可以分裂出數個窗口，如圖 11-7，每一個窗口可以映射到 Document 的任何位置，窗口與窗口之間彼此獨立運作。



在 Splitter Box 上以滑鼠左鍵快按兩下，就可以將視窗分裂開來。Splitter Box 有水平和垂直兩種。分裂視窗的窗口個數，由程式而定，本例是 2x2。不同的窗口可以觀察同一份 Document 的不同區域。本例雖然很巧妙地安排出一張完整的圖出來，其實四個窗口各自看到原圖的某一部份。

圖 11-7 分裂視窗 (splitter window)

在 Splitter Box 上以滑鼠左鍵快按兩下，就可以將視窗分裂開來。Splitter Box 有水平和垂直兩種。分裂視窗的窗口個數，由程式而定，本例是 2x2。不同的窗口可以觀察同一份 Document 的不同區域。本例雖然很巧妙地安排出一張完整的圖出來，其實四個窗口各自看到原圖的某一部份。

## 分裂視窗的程式概念

回憶第 8 章所說的 Document/View 架構，每次打開一個 Document，需有兩個視窗通力合作才能完成顯示任務，一是 *CMDIChildWnd* 視窗，負責視窗的外框架與一般行為，一是 *CView* 視窗，負責資料的顯示。但是當分裂視窗引入，這種結構被打破。現在必須有三個視窗通力合作完成顯示任務（圖 11-8）：

1. Document Frame 視窗：負責一般性視窗行為。其類別衍生自 *CMDIChildWnd*。
2. Splitter 視窗：負責管理各窗口。通常直接使用 *CSplitterWnd* 類別。
3. View 視窗：負責資料的顯示。其類別衍生自 *CView*。

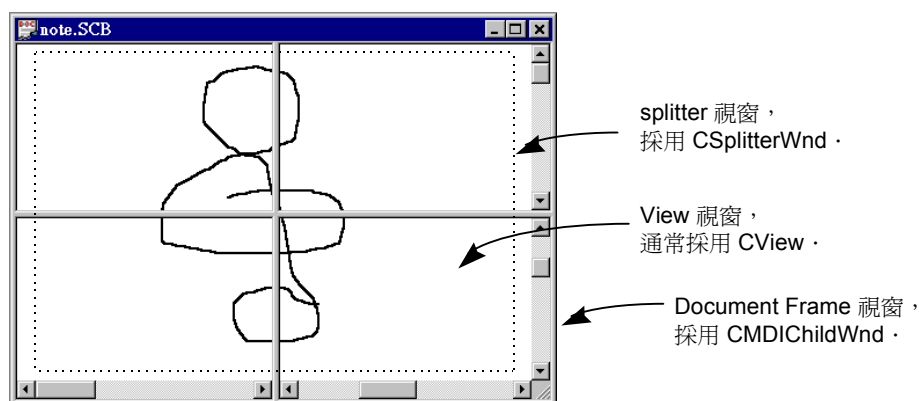
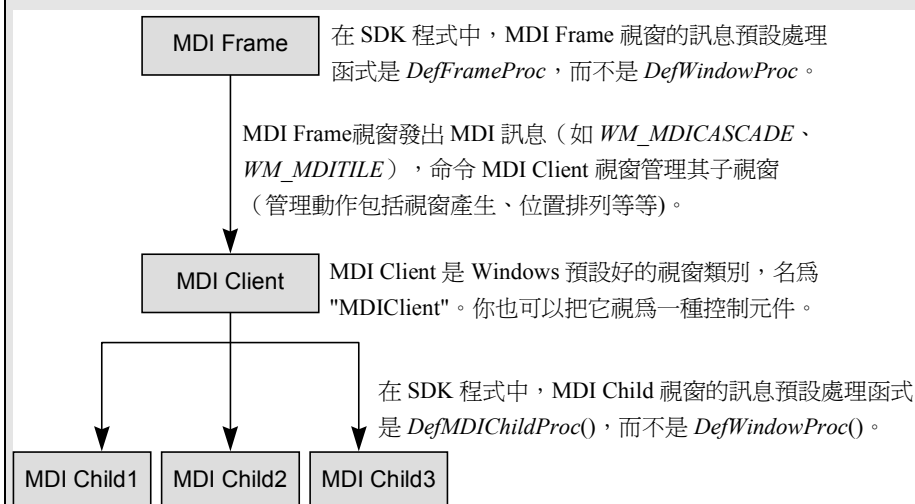


圖 11-8 欲使用分裂視窗，必須三個物件合作才能完成顯示任務，一是 Document Frame 視窗，負責一般性視窗行為；二是 CSplitterWnd 視窗，管理視窗內部空間（各個窗口）；三是 CView 視窗，負責顯示資料。

## 給 SDK 程式員

你有以 SDK 撰寫 MDI 程式的經驗嗎？MDI 程式有三層視窗架構：



程式員想要控制 MDI Child 視窗的大小、位置、排列狀態，必須藉助另一個已經由 Windows 系統定義好的視窗，此視窗稱為 MDI Client 視窗，其類別名稱爲 "MDICLIENT"。

Frame 視窗、Client 視窗和 Child 視窗構成 MDI 的三層架構。Frame 視窗產生之後，通常在 *WM\_CREATE* 時機就以 *CreateWindow("MDICLIENT",...)* 的方式建立 Client 視窗，此後幾乎所有對 Child 視窗的管理工作，諸如產生新的 Child 視窗、重新排列視窗、重新排列圖示、在選單上列出已開啓視窗...等等，都由 Client 代勞，只要 Frame 視窗向 Client 視窗下命令（送 MDI 訊息如 *WM\_MDICREATE* 或 *WM\_MDITILE* 就去）即可。

你可以把 *CSplitterWnd* 物件視為 MDI Client，觀念上比較容易打通。

## 分裂視窗之實作

讓我先把 Scribble 目前使用的類別之中凡與本節主題有關的，做個整理。

Visual C++ 4.0 以前的版本，AppWizard 爲 Scribble 產生的類別是這樣子的：

| 用途             | 類別名稱                            | 基礎類別（MFC 類別）        |
|----------------|---------------------------------|---------------------|
| main frame     | <i>CMainFrame</i>               | <i>CMDIFrameWnd</i> |
| document frame | 直接使用 MFC 類別 <i>CMDIChildWnd</i> | <i>CMDIChildWnd</i> |
| view           | <i>CScribbleView</i>            | <i>CView</i>        |
| document       | <i>CScribbleDoc</i>             | <i>CDocument</i>    |

而其 *CMultiDocTemplate* 物件是這樣子的：

```
pDocTemplate = new CMultiDocTemplate(
 IDR_SCRIBTYPE,
 RUNTIME_CLASS(CScribbleDoc),
 RUNTIME_CLASS(CMDIChildWnd),
 RUNTIME_CLASS(CScribbleView));
```

爲了加上分裂視窗，我們必須利用 ClassWizard 新增一個類別（在 Scribble 程式中名爲

*CScribbleFrame*)，衍生自 *CMDIChildWnd*，並讓它擁有一個 *CSplitterWnd* 物件，名為 *m\_wndSplitter*。然後為 *CScribbleFrame* 改寫 *OnCreateClient* 虛擬函式，在其中呼叫 *m\_wndSplitter.Create* 以產生分裂視窗、設定窗口個數、設定窗口的最初尺寸等初始狀態。最後，當然，我們不能夠再直接以 *CMDIChildWnd* 負責 document frame 視窗，而必須以 *CScribbleFrame* 取代之。也就是說，得改變 *CMultiDocTemplate* 建構式的第三個參數：

```
pDocTemplate = new CMultiDocTemplate(
 IDR_SCRIBTYPE,
 RUNTIME_CLASS(CScribbleDoc),
 RUNTIME_CLASS(CScribbleFrame),
 RUNTIME_CLASS(CScribbleView));
```

俱往矣！Visual C++ 4.0 之後的 AppWizard 為 Scribble 產生的類別是這個樣子：

| 用途             | 類別名稱                 | 基礎類別                |
|----------------|----------------------|---------------------|
| main frame     | <i>CMainFrame</i>    | <i>CMDIFrameWnd</i> |
| document frame | <i>CChildFrame</i>   | <i>CMDIChildWnd</i> |
| view           | <i>CScribbleView</i> | <i>CView</i>        |
| document       | <i>CScribbleDoc</i>  | <i>CDocument</i>    |

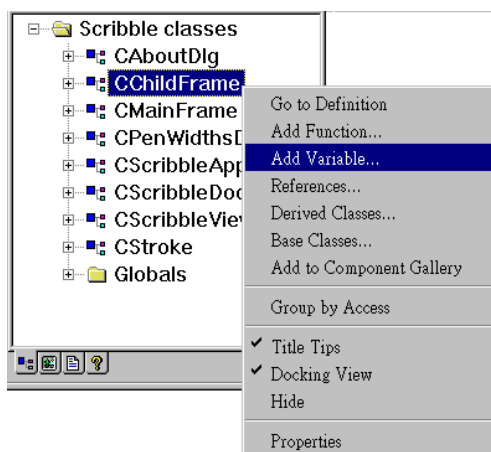
而其 *CMultiDocTemplate* 物件是這樣子的：

```
pDocTemplate = new CMultiDocTemplate(
 IDR_SCRIBTYPE,
 RUNTIME_CLASS(CScribbleDoc),
 RUNTIME_CLASS(CChildFrame),
 RUNTIME_CLASS(CScribbleView));
```

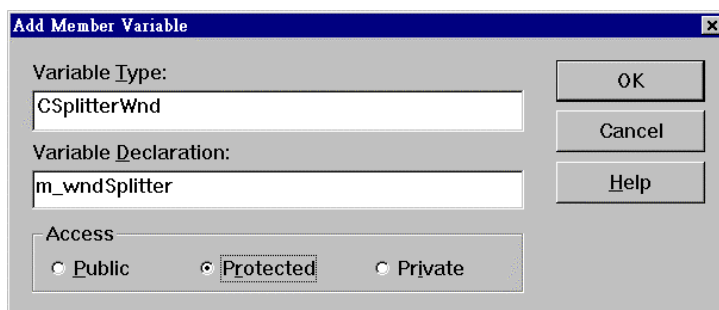
這就方便多了，*CChildFrame* 相當於以前（MFC 4.0 之前）你得自力完成的 *CScribbleFrame*。現在，我們可以從「為此類別新添成員變數」開始作為。

以下是加上分裂視窗的步驟：

- 在 ClassView（注意，不是 ClassWizard）中選擇 *CChildFrame*。按下右鍵，選擇突冒式選單中的【Add Variable】



- 出現【Add Member Variable】對話盒。填充如下，然後選按【OK】。



現在你可以從 ClassView 畫面中即時看到 *CChildFrame* 的新變數。

- 打開 ChildFrm.cpp，在 WizardBar 的【Messages】清單中選擇 *OnCreateClient*。
  - 以 Yes 回答 WizardBar 的詢問，產生處理常式。
  - 在函式空殼中鍵入以下內容：
- ```
return m_wndSplitter.Create(this, 2, 2, CSize(10, 10), pContext);
```

■ 回到 `ClassView` 之中，你可以看到新的函式。

`CSplitterWnd::Create` 正是產生分裂視窗的關鍵，它有七個參數：

1. 表示父視窗。這裡的 `this` 代表的是 `CChildFrame` 視窗。
2. 分裂視窗的水平窗口數（row）
3. 分裂視窗的垂直窗口數（column）
4. 窗口的最小尺寸（應該是一個 `CSize` 物件）
5. 在窗口上使用哪一個 `View` 類別。此參數直接取用 Framework 交給 `OnCreateClient` 的第二個參數即可。
6. 指定分裂視窗的風格。預設值是：`WS_CHILD|WS_VISIBLE|WS_HSCROLL|WS_VSCROLL|SPLS_DYNAMIC_SPLIT`，意思就是一個可見的子視窗，有著水平捲軸和垂直捲軸，並支援動態分裂。關於動態分裂（以及所謂的靜態分裂），第 13 章將另有說明。
7. 分裂視窗的 ID。預設值是 `AFX_IDW_PANE_FIRST`，這將成為第一個窗口的 ID。

我們的原始碼有了下列變化：

```
// in CHILDFRM.H
class CChildFrame : public CMDIChildWnd
{
protected:
    CSplitterWnd    m_wndSplitter;
protected:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
    ...
};

// in CHILDFRM.CPP
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT /* lpcs */,
                                CCreateContext* pContext
{
    return m_wndSplitter.Create(this,
                                2, 2,          // TODO: adjust the number of rows, columns
                                CSize(10, 10), // TODO: adjust the minimum pane size
```

```

        pContext);
    }

```

本章回顧

這一章裡我們追求的是精緻化。

Scribble Step3 已經有繪圖、檔案讀寫、變化筆寬的基本功能，但是「連接到同一份 Document 的不同的 Views」之間卻不能夠做到同步更新的視覺效果，此外 View 視窗中沒有捲軸也是遺憾的事情。

Scribble Step4 彌補了上述遺憾。它讓「連接到同一份 Document 的不同的 Views」之間做到同步更新 -- 關鍵在於 *CDocument::UpdateAllViews* 和 *CView::Update* 兩個虛擬函式。而由於同步更新引發的繪圖效率問題，所以我們又學會了如何設計所謂的 hint，讓繪圖動作更聰敏些。也因為 hint 緣故，我們改變了 Document 的格式，為每一線條加上一個外圍四方形記錄。

在捲軸方面，MFC 提供了一個名為 *CScrollView* 的類別，內有捲軸功能，因此直接拿來用就好了。我們唯一要擔心的是，從 *CView* 改為 *CScrollView*，原先的 *OnDraw* 繪圖動作要不要修改？畢竟，捲來捲去把原點都不知捲到哪裡去了，何況還有映像模式（座標系統）的問題。這一點是甬擔心了，因為 application framework 在呼叫 *OnDraw* 之前，已經先呼叫了 *OnPrepareDC*，把問題解決掉了。唯一要注意的是，送進 *OnDraw* 的滑鼠座標點應該改為邏輯座標，以文件左上角為原點。*DP2LP* 函式可以幫我們這個忙。

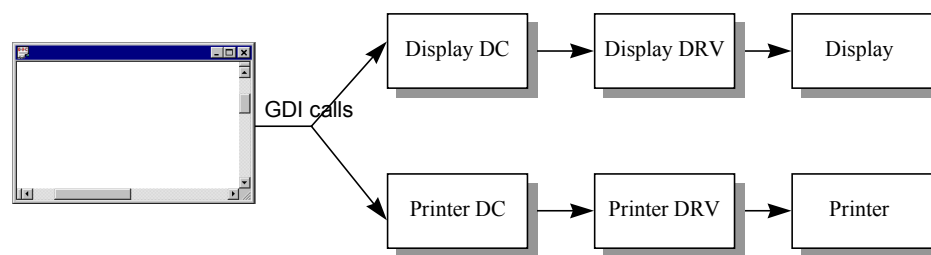
此外，我們接觸了另一種新而且更精緻的 UI 介面：分裂視窗，讓一個視窗分裂為數個窗口，每一個窗口容納一個 View。MFC 提供 *CSplitterWnd* 做此服務。

列印與預覽

「列印」絕對是個大工程，「列印預覽」是個更大的工程。如果你是一位 SDK 程式員，而你分配到的工作是為公司的繪圖軟體寫一個印前預瀏系統，那麼我真的替你感到憂鬱。可如果你使用 MFC，情況又大不相同了。

概觀

Windows 的 DC 觀念，在程式的繪圖動作與實際設備的驅動程式之間做了一道隔離，使得繪圖動作完全不需修改就可以輸出到不同的設備上：



即便如此，列印仍然有其瑣碎的工作需要由程式員承擔。舉個例子，螢幕視窗有捲動桿，印表機沒有，於是「分頁」就成了一門學問。另外，如何中斷列印？如何設計水平方向

（landscape）或垂直方向（portrait）的列印輸出？

landscape，風景畫，代表橫向列印；portrait，人物畫，代表縱向列印。

如果曾經有過 SDK 程式經驗，你一定知道，把資料輸出到螢幕上和輸出到印表機上幾乎是相同的一件事，只要換個 DC（註）就好了。MFC 甚至不要求程式員的任何動作，即自動供應列印功能和預覽功能。拿前面各版本的 Scribble 為例，我們可曾爲了輸出任何東西到印表機上而特別考慮什麼程式碼？完全沒有！但它的確已擁有列印和預覽功能，你不妨執行 Step4 的【File/Print...】以及【File/Print Preview】看看，結果如圖 12-1a。

註：DC 就是 Device Context，在 Windows 中凡繪圖動作之前一定要先獲得一個 DC，它可能代表全螢幕，也可能代表一個視窗，或一塊記憶體，或印表機...。DC 中有許多繪圖所需的元素，包括座標系統（映像模式）、原點、繪圖工具（筆、刷、顏色...）等等。它還連接到低階的輸出裝置驅動程式。由於 DC，我們在程式中對螢幕作畫和對印表機作畫的動作才有可能完全相同。

Scribble 程式之所以不費吹灰之力即擁有列印與預覽功能，是因為負責資料顯示的 *CScribbleView::OnDraw* 函式接受了一個 DC 參數，此 DC 如果是 display DC，所有的輸出就往螢幕送，如果是 printer DC，所有輸出就往印表機送。至於 *OnDraw* 到底收到什麼樣的 DC，則由 Framework 決定 -- 想當然耳 Framework 會依使用者的動作決定之。

MFC 把整個列印機制和預覽機制都埋在 application framework 之中了，我們因此也有了標準的 UI 介面可以使用，如標準的【列印】對話盒、【列印設定】對話盒、【列印中】對話盒等等，請看圖 12-1。

我將在這一章介紹 MFC 的印表與預覽機制，以及如何強化它。

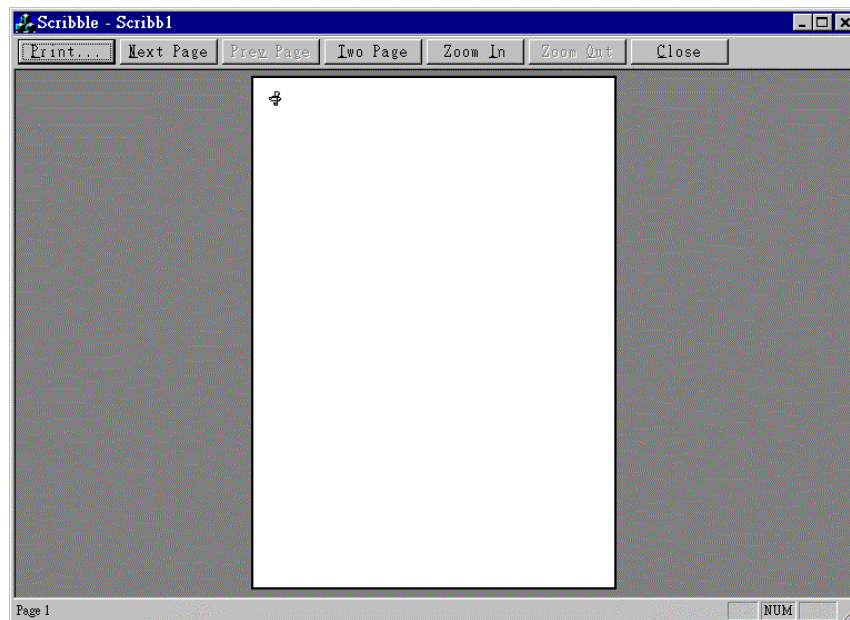


圖 12-1a 不需考慮任何與列印相關的程式動作，Scribble 即已具備印表與預覽功能（只要我們一開始在 AppWizard 的步驟四對話盒中選擇【Printing and Print Preview】項目）。列印出來的圖形大小並不符合理想，從預覽畫面中就可察知。這正是本章要改善的地方。

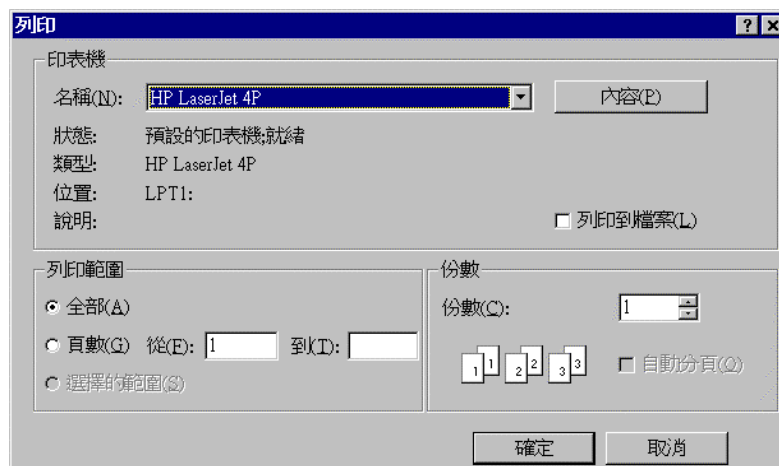


圖 12-1b 標準的列印 UI 介面。本圖是選按 Scribble 的【File/Print...】命令項之後獲得的【列印】對話盒。



圖 12-1c 你可以選按 Scribble 的【File/Print Setup...】命令項，獲得設定印表機的機會。

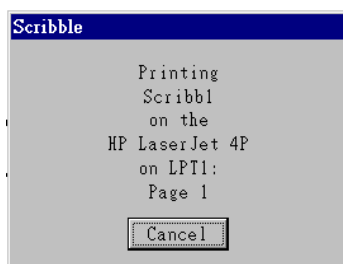


圖 12-1d 列印過程中會出現一個標準的【列印狀態】對話盒，允許使用者中斷列印動作。

Scribble Step5 加強了印表功能以及預覽功能。MFC 各現成類別之中已有印表和預覽機制，我要解釋的是它的運作模式、執行效果、以及改善之道。圖 12-2 就是 Scribble Step5 的預覽效果，UI 方面並沒有什麼新東西，主要的改善是，圖形的輸出大小比較能夠被接受了，每一份文件並且分為兩頁，第一頁是文件名稱（檔案名稱），第二頁才是真正的文件內容，上有一表頭。

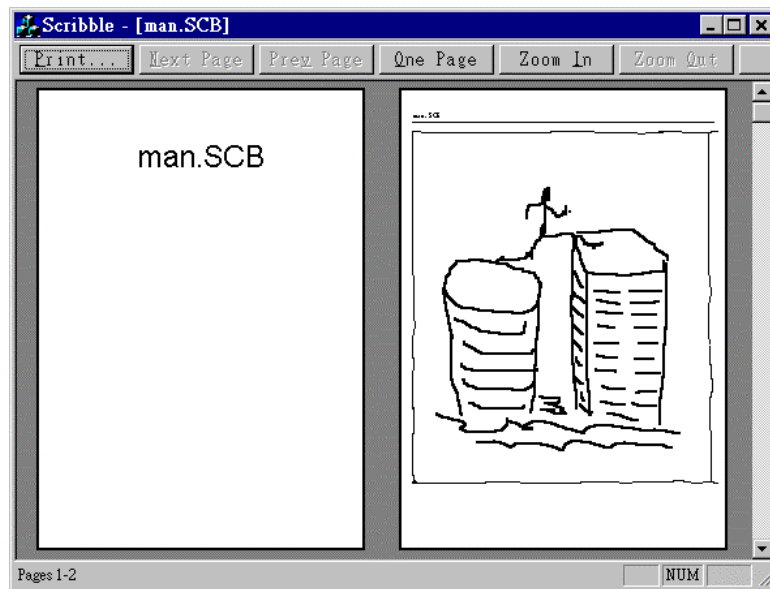


圖 12-2 Scribble Step5 列印預覽。第一頁是文件名稱，第二頁是文件內容。

列印動作的背景原理

開始介紹 MFC 的列印機制之前，我想，如果先讓你了解列印的背後原理，可以幫助你掌握其本質。

Windows 的所有繪圖指令，都集中在 GDI 模組之中，稱為 GDI 繪圖函式，例如：

```
TextOut(hPr, 50, 50, szText, strlen(szText));    // 輸出一字串
Rectangle(hPr, 10, 10, 50, 40);                  // 畫一個四方形
Ellipse(hPr, 200, 50, 250, 80);                  // 畫一個橢圓形
Pie(hPr, 350, 50, 400, 100, 400, 50, 400, 100); // 畫一個圓餅圖
MoveTo(hPr, 50, 100);                             // 將畫筆移動到新位置
LineTo(hPr, 400, 50);                             // 從前一位置畫直線到新位置
```

圖形輸往何方？關鍵在於 DC，這是任何 GDI 繪圖函式的第一個參數，可以是 *GetDC* 或 *BeginPaint* 函式所獲得的「顯示幕 DC」（以下是 SDK 程式寫法）：

```
HDC hDC;
PAINTSTRUCT ps;    // paint structure
hDC = BeginPaint(hWnd, &ps);
```

也可以是利用 *CreateDC* 獲得的一個「印表機 DC」：

```
HDC hPr;
hPr = CreateDC(lpPrintDriver, lpPrintType, lpPrintPort, (LPSTR) NULL);
```

其中前三個參數分別是與印表機有關的資訊字串，可以從 WIN.INI 的【windows】section 中獲得，各以逗號分隔，例如：

```
device=HP LaserJet 4P/4MP,HPPCL5E,LPT1:
```

代表三項意義：

- Print Driver = HP LaserJet 4P/4MP
- Print Type = HPPCL5E
- Print Port = LPT1:

SDK 程式中對於列印所需做的努力，最低限度到此為止。顯然，困難度並不高，但是其中尚未參雜對印表機的控制，而那是比較麻煩的事兒。換句話說我們還得考慮「分頁」的問題。以文字為例，我們必須取得一頁（一張紙）的大小，以及字形的高度，從而計算扣除留白部份之後，一頁可容納幾行：

```
TEXTMETRIC TextMetric;
int LineSpace;
int nPageSize;
int LinesPerPage;

GetTextMetrics(hPr, &TextMetric); // 取得字形資料
LineSpace = TextMetric.tmHeight + TextMetric.tmExternalLeading; // 計算字高
nPageSize = GetDeviceCaps(hPr, VERTRES); // 取得紙張大小
LinesPerPage = nPageSize / LineSpace - 1; // 一頁容納多少行
```

然後再以迴圈將每一行文字送往印表機：

```
Escape(hPr, STARTDOC, 4, "PrntFile text", (LPSTR) NULL);
CurrentLine = 1;
for (...) {
    ... // 取得一行文字，放在 char pLine[128] 中，長度為 LineLength。
```

```

TextOut(hPr, 0, CurrentLine*LineSpace, (LPSTR)pLine, LineLength);
if (++CurrentLine > LinesPerPage ) {
    CurrentLine = 1; // 重設行號
    IOStatus = Escape(hPr, NEWFRAME, 0, 0L, 0L); // 換頁
    if (IOStatus < 0 || bAbort)
        break;
}
}
if (IOStatus >= 0 && !bAbort) {
    Escape(hPr, NEWFRAME, 0, 0L, 0L);
    Escape(hPr, ENDDOC, 0, 0L, 0L);
}
}

```

其中的 *Escape* 用來傳送命令給印表機（印表機命令一般稱為 *escape code*），它是一個 Windows API 函式。

列印過程中我們還應該提供一個中斷機制給使用者。*Modeless* 對話盒可以完成此一使命，我們可以讓它出現在列印過程之中。這個對話盒應該在列印程序開始之前先做起來，外形類似圖 12-1d：

```

HWND hPrintingDlgWnd; // 這就是【Printing】對話盒
FARPROC lpPrintingDlg; // 【Printing】對話盒的視窗函式

lpPrintingDlg = MakeProcInstance(PrintingDlg, hInst);
hPrintingDlgWnd = CreateDialog(hInst, "PrintingDlg", hWnd, lpPrintingDlg);
ShowWindow(hPrintingDlgWnd, SW_NORMAL);

```

負責此一中斷機制的對話盒函式很簡單，只檢查【OK】鈕有沒有被按下，並據以改變 *bAbort* 的值：

```

int FAR PASCAL PrintingDlg(HWND hDlg, unsigned msg, WORD wParam, LONG lParam)
{
    switch(msg) {
        case WM_COMMAND:
            return (bAbort = TRUE);

        case WM_INITDIALOG:
            SetFocus(GetDlgItem(hDlg, IDCANCEL));
            SetDlgItemText(hDlg, IDC_FILENAME, FileName);
            return (TRUE);
    }
    return (FALSE);
}

```


從應用程式的眼光來看，這樣就差不多了。然而資料真正送到印表機上，還有一段曲折過程。每一個送往印表機 DC 的繪圖動作，其實都只被記錄為 metafile (註) 儲存在你的 TEMP 目錄中。當你呼叫 *Escape(hPr, NEWFRAME, ...)*，印表機驅動程式 (.DRV) 會把這些 metafile 轉換為印表機語言(control sequence 或 Postscript)，然後通知 GDI 模組，由 GDI 把它儲存為 ~SPL 檔案，也放在 TEMP 目錄中，並刪除對應之 metafile。之後，GDI 模組再送出訊息給列印管理器 Print Manager，由後者呼叫 *OpenComm*、*WriteComm* 等低階通訊函式(也都是 Windows API 函式)，把印表機命令傳給印表機。整個流程請參考圖 12-3。

註：metafile 也是一種圖形記錄規格，但它記錄的是繪圖動作，不像 bitmap 記錄的是真正的圖形資料。所以播放 metafile 比播放 bitmap 慢，因為多了一層繪圖函式解讀動作；但它的大小比 bitmap 小很多。用在有許多四形、圓形、工程幾何圖形上最為方便。

這個曲折過程之中就產生了一個問題。~SPL 這種檔案很大，如果你的 TEMP 目錄空間不夠充裕，怎麼辦？如果 Printer Manager 把積存的 ~SPL 內容消化掉後能夠空出足夠磁碟空間的話，那麼 GDI 模組就可以下命令(送訊息)給 Printer Manager，先把積存的 ~SPL 檔處理掉。問題是，在 Windows 3.x 之中，我們的程式此刻正忙著做繪圖動作，GDI 沒有機會送訊息給 Printer Manager (因為 Windows 3.x 是個非強制性多工系統)。解決方法是你先準備一個 callback 函式，名稱隨你取，通常名為 *AbortProc*：

```
FARPROC lpAbortProc;
lpAbortProc = MakeProcInstance(AbortProc, hInst);
Escape(hPr, SETABORTPROC, NULL, (LPSTR)(long)lpAbortProc, (LPSTR)NULL);
```

GDI 模組在執行 *Escape(hPr, NEWFRAME...)* 的過程中會持續呼叫這個 callback 函式，想辦法讓你的程式釋放出控制權：

```
int FAR PASCAL AbortProc(hDC hPr, int Code)
{
    MSG msg;

    while (!bAbort && PeekMessage(&msg, NULL, NULL, NULL, TRUE))
        if (!IsDialogMessage(hAbortDlgWnd, &msg)) {
            TranslateMessage(&msg);
```

```

        DispatchMessage(&msg);
    }

    return (!bAbort);
}

```

你可以從 VC++ 4.0 所附的這個範例程式獲得有關列印的極佳實例：

\MSDEV\SAMPLES\SDK\WIN32\PRINTER

也可以在 Charles Petzold 所著的 *Programming Windows 3.1* 第 15 章，或是其新版 *Programming Windows 95* 第 15 章，獲得更深入的資料。

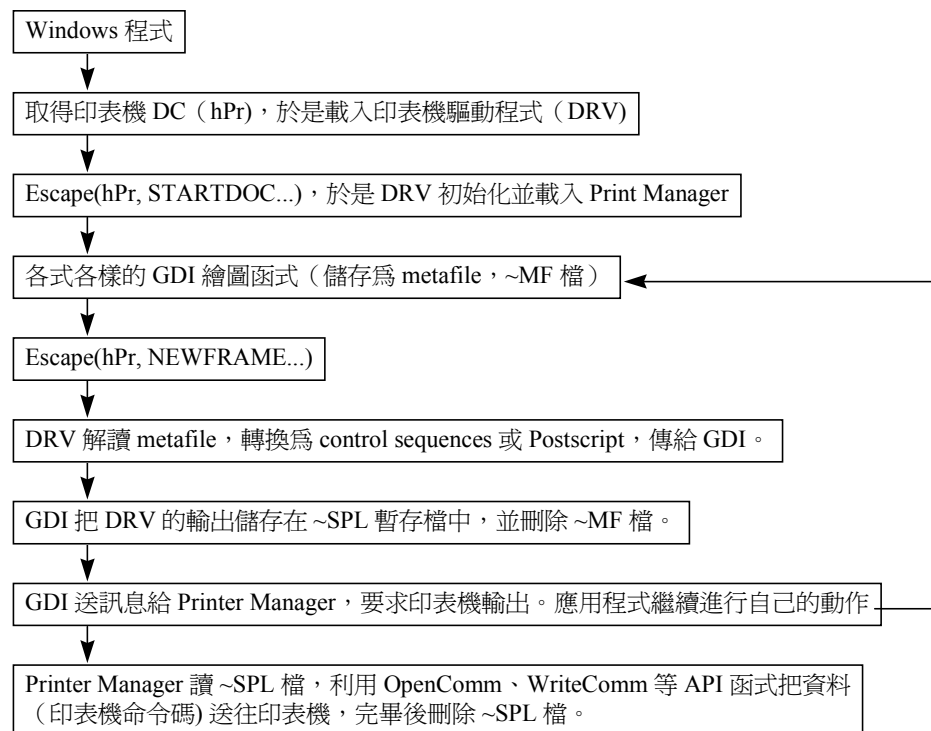


圖 12-3 Windows 程式的印表機輸出動作詳解

以下就是 SDK 程式中有關列印程序的一個實際片段。

```
#01 hSaveCursor = SetCursor(hHourGlass); // 把滑鼠游標設為砂漏狀
#02 hPr = CreateDC("HP LaserJet 4P/4MP", "HPPCL5E", "LPT1:", (LPSTR) NULL);
#03
#04 // 設定 AbortProc callback 函式
#05 lpAbortProc = MakeProcInstance(AbortProc, hInst);
#06 Escape(hPr, SETABORTPROC, NULL, (LPSTR) (long) lpAbortProc, (LPSTR) NULL);
#07 bAbort = FALSE;
#08
#09 Escape(hPr, STARTDOC, 4, "PrntFile text", (LPSTR) NULL);
#10
#11 // 設定 Printing 對話盒及其視窗函式
#12 lpPrintingDlg = MakeProcInstance(PrintingDlg, hInst);
#13 hPrintingDlgWnd = CreateDialog(hInst, "PrintingDlg", hWnd, lpPrintingDlg);
#14 ShowWindow(hPrintingDlgWnd, SW_NORMAL);
#15 EnableWindow(hWnd, FALSE); // 令其父視窗 (也就是程式的主視窗) 除能
#16 SetCursor(hSaveCursor); // 滑鼠游標形狀還原
#17
#18 GetTextMetrics(hPr, &TextMetric);
#19 LineSpace = TextMetric.tmHeight + TextMetric.tmExternalLeading;
#20 nPageSize = GetDeviceCaps(hPr, VERTRES);
#21 LinesPerPage = nPageSize / LineSpace - 1;
#22 dwLines = SendMessage(hEditWnd, EM_GETLINECOUNT, 0, 0L);
#23 CurrentLine = 1;
#24
#25 for (dwIndex = IOStatus = 0; dwIndex < dwLines; dwIndex++) {
#26     pLine[0] = 128;
#27     pLine[1] = 0;
#28     LineLength = SendMessage(hEditWnd, EM_GETLINE,
#29         (WORD)dwIndex, (LONG)((LPSTR)pLine));
#30     TextOut(hPr, 0, CurrentLine*LineSpace, (LPSTR)pLine, LineLength);
#31     if (++CurrentLine > LinesPerPage) {
#32         CurrentLine = 1;
#33         IOStatus = Escape(hPr, NEWFRAME, 0, 0L, 0L);
#34         if (IOStatus<0 || bAbort)
#35             break;
#36     }
#37 }
#38
#39 if (IOStatus >= 0 && !bAbort) {
#40     Escape(hPr, NEWFRAME, 0, 0L, 0L);
#41     Escape(hPr, ENDDOC, 0, 0L, 0L);
#42 }
```

```
#43
#44 EnableWindow(hWnd, TRUE);
#45
#46 DestroyWindow(hPrintingDlgWnd);
#47 FreeProcInstance(lpPrintingDlg);
#48 FreeProcInstance(lpAbortProc);
#49 DeleteDC(hPr);
```

上述各個 *Escape* 呼叫，是在 Windows 3.0 下的傳統作法，在 Windows 3.1 以及 Win32 之中有對應的 API 函式如下：

Windows 3.0 作法	Windows 3.1 作法
Escape(hPr, SETABORTPROC, ...)	SetAbortProc(HDC hdc, ABORTPROC lpAbortProc)
Escape(hPr, STARTDOC, ...)	StartDoc(HDC hdc, CONST DOCINFO* lpdi)
Escape(hPr, NEWFRAME, ...)	EndPage(HDC hdc)
Escape(hPr, ENDDOC, ...)	EndDoc(HDC hdc)

MFC 預設的列印機制

好啦，關於列印，其實有許多一成不變的動作！為什麼開發工具不幫我們做掉呢？好比如說，從 WIN.INI 中取得目前印表機的資料然後利用 *CreateDC* 取得印表機 DC，又好比如說設計標準的【列印中】對話盒，以及標準的列印中斷函式 *AbortProc*。

事實上 MFC 的確已經幫我們做掉了一大部份的工作。MFC 已內含印表機制，那麼將 Framework 整個納入 EXE 檔案中的你當然也就不費吹灰之力得到了印表功能。只要 *OnDraw* 函式設計好了，不但可以在螢幕上顯示資料，也可以在印表機上顯示資料。有什麼是我們要負擔的？沒有了！Framework 傳給 *OnDraw* 一個 DC，視情況的不同這個 DC 可能是顯示幕 DC，也可能是印表機 DC，而你知道，Windows 程式中的圖形輸出對象完全取決於 DC：

- 當你改變視窗大小，產生 *WM_PAINT*，*OnDraw* 會收到一個「顯示幕 DC」。
- 當你選按【File/Print...】，*OnDraw* 會收到一個「印表機 DC」。

數章之前討論 *CView* 時我曾經提過，*OnDraw* 是 *CView* 類別中最重要的成員函式，所有的繪圖動作都應該放在其中。請注意，*OnDraw* 接受一個「CDC 物件指標」做為它的參數。當視窗接受 *WM_PAINT* 訊息，Framework 就呼叫 *OnDraw* 並把一個「顯示幕 DC」傳過去，於是 *OnDraw* 輸出到螢幕上。

Windows 的圖形裝置介面 (GDI) 完全與硬體無關，相同的繪圖動作如果送到「顯示幕 DC」，就是在螢幕上繪圖，如果送到「印表機 DC」，就是在印表機上繪圖。這個道理很容易就解釋了為什麼您的程式碼沒有任何特殊動作卻具備印表功能：當使用者按下【File/Print】，application framework 送給 *OnDraw* 的是一個「印表機 DC」而不再是「顯示幕 DC」。

在 MFC 應用程式中，View 和 application framework 分工合力完成印表工作。Application framework 的責任是：

- 顯示【Print】對話盒，如圖 12-1b。
- 為印表機產生一個 CDC 物件。
- 呼叫 CDC 物件的 *StartDoc* 和 *EndDoc* 兩函式。
- 持續不斷地呼叫 CDC 物件的 *StartPage*，通知 View 應該輸出哪一頁；一頁列印完畢則呼叫 CDC 物件的 *EndPage*。

我們（程式員）在 View 物件上的責任是：

- 通知 application framework 總共有多少頁要列印。
- application framework 要求列印某特定頁時，我們必須將 Document 中對應的部份輸出到印表機上。
- 配置或釋放任何 GDI 資源，包括筆、刷、字形...等等。
- 如果需要，送出任何 escape 碼改變印表機狀態，例如送紙、改變列印方向等等。

送出 `escape` 碼的方式是，呼叫 `CDC` 物件的 `Escape` 函式。

現在讓我們看看這兩組工作如何交叉在一起。為實現上述各項交互動作，`CView` 定義了幾個相關的成員函式，當你在 `AppWizard` 中選擇【Printing and Print Preview】選項之後，除了 `OnDraw`，你的 `View` 類別內還被加入了三個虛擬函式空殼：

```
// in SCRIBBLEVIEW.H
class CScribbleView : public CScrollView
{
    ...
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    ...
};

// in SCRIBBLEVIEW.CPP
BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CScribbleView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}
```

改寫這些函式有助於我們在 `framework` 的列印機制與應用程式的 `View` 物件之間架起溝通橋樑。

爲了了解 `MFC` 中的列印機制，我又動用了我的法寶：`Visual C++ Debugger`。我發現，`AppWizard` 爲我的 `View` 做出這樣的 `Message Map`：

```
BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
...
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

顯然，當【File/Print...】被按下，命令訊息將流往 *CView::OnFilePrint* 去處理，於是我以 Debugger 進入該位置並且一步一步執行，得到圖 12-4 的結果。

```
// in VIEWPRNT.CPP
#0001 void CView::OnFilePrint()
#0002 {
#0003     // get default print info
#0004     ❶ CPrintInfo printInfo;
#0005     ASSERT(printInfo.m_pPD != NULL);    // must be set
#0006
#0007     if (GetCurrentMessage()->wParam == ID_FILE_PRINT_DIRECT)
#0008     {
#0009         CCommandLineInfo* pCmdInfo = AfxGetApp()->m_pCmdInfo;
#0010
#0011         if (pCmdInfo != NULL)
#0012         {
#0013             ❷ if (pCmdInfo->m_nShellCommand == CCommandLineInfo::FilePrintTo)
#0014             {
#0015                 printInfo.m_pPD->m_pd.hDC = ::CreateDC(pCmdInfo->m_strDriverName,
#0016                 pCmdInfo->m_strPrinterName, pCmdInfo->m_strPortName, NULL);
#0017                 if (printInfo.m_pPD->m_pd.hDC == NULL)
#0018                 {
#0019                     AfxMessageBox(AFX_IDP_FAILED_TO_START_PRINT);
#0020                     return;
#0021                 }
#0022             }
#0023         }
#0024
#0025         printInfo.m_bDirect = TRUE;
#0026     }
#0027
#0028     ❸ if (OnPreparePrinting(&printInfo))
#0029     {
#0030         // hDC must be set (did you remember to call DoPreparePrinting?)
#0031         ASSERT(printInfo.m_pPD->m_pd.hDC != NULL);
#0032
#0033         ❹ // gather file to print to if print-to-file selected
```

```

#0034     CString strOutput;
#0035     if (printInfo.m_pPD->m_pd.Flags & PD_PRINTTOFILE)
#0036     {
#0037         // construct CFileDialog for browsing
#0038         CString strDef(MAKEINTRESOURCE(AFX_IDS_PRINTDEFAULTTEXT));
#0039         CString strPrintDef(MAKEINTRESOURCE(AFX_IDS_PRINTDEFAULT));
#0040         CString strFilter(MAKEINTRESOURCE(AFX_IDS_PRINTFILTER));
#0041         CString strCaption(MAKEINTRESOURCE(AFX_IDS_PRINTCAPTION));
#0042         CFileDialog dlg(FALSE, strDef, strPrintDef,
#0043             OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT, strFilter);
#0044         dlg.m_ofn.lpstrTitle = strCaption;
#0045
#0046         if (dlg.DoModal() != IDOK)
#0047             return;
#0048
#0049         // set output device to resulting path name
#0050         strOutput = dlg.GetPathName();
#0051     }
#0052
#0053     ⑤ // set up document info and start the document printing process
#0054     CString strTitle;
#0055     CDocument* pDoc = GetDocument();
#0056     if (pDoc != NULL)
#0057         strTitle = pDoc->GetTitle();
#0058     else
#0059         GetParentFrame()->GetWindowText(strTitle);
#0060     if (strTitle.GetLength() > 31)
#0061         strTitle.ReleaseBuffer(31);
#0062     DOCINFO docInfo;
#0063     memset(&docInfo, 0, sizeof(DOCINFO));
#0064     docInfo.cbSize = sizeof(DOCINFO);
#0065     docInfo.lpszDocName = strTitle;
#0066     CString strPortName;
#0067     int nFormatID;
#0068     if (strOutput.IsEmpty())
#0069     {
#0070         docInfo.lpszOutput = NULL;
#0071         strPortName = printInfo.m_pPD->GetPortName();
#0072         nFormatID = AFX_IDS_PRINTONPORT;
#0073     }
#0074     else
#0075     {
#0076         docInfo.lpszOutput = strOutput;
#0077         AfxGetFileTitle(strOutput,
#0078             strPortName.GetBuffer(_MAX_PATH), _MAX_PATH);
#0079         nFormatID = AFX_IDS_PRINTTOFILE;

```



```

#0080     }
#0081
#0082 ⑥    // setup the printing DC
#0083     CDC dcPrint;
#0084     dcPrint.Attach(printInfo.m_ppd->m_pd.hDC); // attach printer dc
#0085     dcPrint.m_bPrinting = TRUE;
#0086 ⑦    OnBeginPrinting(&dcPrint, &printInfo);
#0087 ⑧    dcPrint.SetAbortProc(_AfxAbortProc);
#0088
#0089     // disable main window while printing & init printing status dialog
#0090 ⑨    AfxGetMainWnd()->EnableWindow(FALSE);
#0091     CPrintingDialog dlgPrintStatus(this);
#0092
#0093     CString strTemp;
#0094     dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_DOCNAME, strTitle);
#0095
#0096     dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PRINTERNAME,
#0097         printInfo.m_ppd->GetDeviceName());
#0098     AfxFormatString1(strTemp, nFormatID, strPortName);
#0099     dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PORTNAME, strTemp);
#0100
#0101     dlgPrintStatus.ShowWindow(SW_SHOW);
#0102     dlgPrintStatus.UpdateWindow();
#0103
#0104     // start document printing process
#0105 ⑩    if (dcPrint.StartDoc(&docInfo) == SP_ERROR)
#0106     {
#0107         // enable main window before proceeding
#0108         AfxGetMainWnd()->EnableWindow(TRUE);
#0109
#0110         // cleanup and show error message
#0111         OnEndPrinting(&dcPrint, &printInfo);
#0112         dlgPrintStatus.DestroyWindow();
#0113         dcPrint.Detach(); // will be cleaned up by CPrintInfo destructor
#0114         AfxMessageBox(AFX_IDP_FAILED_TO_START_PRINT);
#0115         return;
#0116     }
#0117
#0118     // Guarantee values are in the valid range
#0119     UINT nEndPage = printInfo.GetToPage();
#0120     UINT nStartPage = printInfo.GetFromPage();
#0121
#0122     if (nEndPage < printInfo.GetMinPage())
#0123         nEndPage = printInfo.GetMinPage();
#0124     if (nEndPage > printInfo.GetMaxPage())
#0125         nEndPage = printInfo.GetMaxPage();

```

```

#0126
#0127         if (nStartPage < printInfo.GetMinPage())
#0128             nStartPage = printInfo.GetMinPage();
#0129         if (nStartPage > printInfo.GetMaxPage())
#0130             nStartPage = printInfo.GetMaxPage();
#0131
#0132         int nStep = (nEndPage >= nStartPage) ? 1 : -1;
#0133         nEndPage = (nEndPage == 0xffff) ? 0xffff : nEndPage + nStep;
#0134
#0135         VERIFY(strTemp.LoadString(AFX_IDS_PRINTPAGENUM));
#0136
#0137         // begin page printing loop
#0138         BOOL bError = FALSE;
#0139         ① for (printInfo.m_nCurPage = nStartPage;
#0140             printInfo.m_nCurPage != nEndPage; printInfo.m_nCurPage += nStep)
#0141             {
#0142             ② OnPrepareDC(&dcPrint, &printInfo);
#0143
#0144             // check for end of print
#0145             if (!printInfo.m_bContinuePrinting)
#0146                 break;
#0147
#0148             // write current page
#0149             TCHAR szBuf[80];
#0150             wsprintf(szBuf, strTemp, printInfo.m_nCurPage);
#0151             ③ dlgPrintStatus.SetDlgItemText(AFX_IDC_PRINT_PAGENUM, szBuf);
#0152
#0153             // set up drawing rect to entire page (in logical coordinates)
#0154             printInfo.m_rectDraw.SetRect(0, 0,
#0155                 dcPrint.GetDeviceCaps(HORZRES),
#0156                 dcPrint.GetDeviceCaps(VERTRES));
#0157             dcPrint.DPtoLP(&printInfo.m_rectDraw);
#0158
#0159             // attempt to start the current page
#0160             ④ if (dcPrint.StartPage() < 0)
#0161                 {
#0162                     bError = TRUE;
#0163                     break;
#0164                 }
#0165
#0166             // must call OnPrepareDC on newer versions of Windows because
#0167             // StartPage now resets the device attributes.
#0168             if (afxData.bMarked4)
#0169                 OnPrepareDC(&dcPrint, &printInfo);
#0170
#0171             ASSERT(printInfo.m_bContinuePrinting);

```

```
#0172
#0173          // page successfully started, so now render the page
#0174 ⑤      OnPrint(&dcPrint, &printInfo);
#0175 ⑥      if (dcPrint.EndPage() < 0 || !_AfxAbortProc(dcPrint.m_hDC, 0))
#0176      {
#0177          bError = TRUE;
#0178          break;
#0179      }
#0180  }
#0181
#0182      // cleanup document printing process
#0183      if (!bError)
#0184 ⑦      dcPrint.EndDoc();
#0185      else
#0186          dcPrint.AbortDoc();
#0187
#0188      AfxGetMainWnd()->EnableWindow();    // enable main window
#0189
#0190 ⑧      OnEndPrinting(&dcPrint, &printInfo);    // clean up after printing
#0191 ⑨      dlgPrintStatus.DestroyWindow();
#0192
#0193 ⑩      dcPrint.Detach();    // will be cleaned up by CPrintInfo destructor
#0194  }
#0195  }
```

圖 12-4 CView::OnFilePrint 原始碼，這是列印命令的第一戰場。標出號碼的是重要動作，稍後將有補充說明。

以下是 *CView::OnFilePrint* 函式之中重要動作的說明。你可以將這份說明與上一節「列印動作的背景原理」做一比對，就能夠明白 MFC 在什麼地方為我們做了什麼事情，也才因此能夠體會，究竟我們該在什麼地方改寫虛擬函式，放入我們自己的補強程式碼。

❶ *OnFilePrint* 首先在堆疊中產生一個 *CPrintInfo* 物件，並建構之，使其部份成員變數擁有初值。*CPrintInfo* 是一個用來記錄印表機資料的結構，其建構式配置了一個 Win32 通用列印對話盒（common print dialog）並將它指定給 *m_pPD*：

```
// in AFXEXT.H
struct CPrintInfo // Printing information structure
{
    CPrintDialog* m_pPD;    // pointer to print dialog
    BOOL m_bPreview;       // TRUE if in preview mode
    BOOL m_bDirect;        // TRUE if bypassing Print Dialog
    ...
};
```

上述的成員變數 *m_bPreview* 如果是 *TRUE*，表示處於預覽模式，*FALSE* 表示處於列印模式；成員變數 *m_bDirect* 如果是 *TRUE*，表示省略【列印】對話盒，*FALSE* 表示需顯示【列印】對話盒。

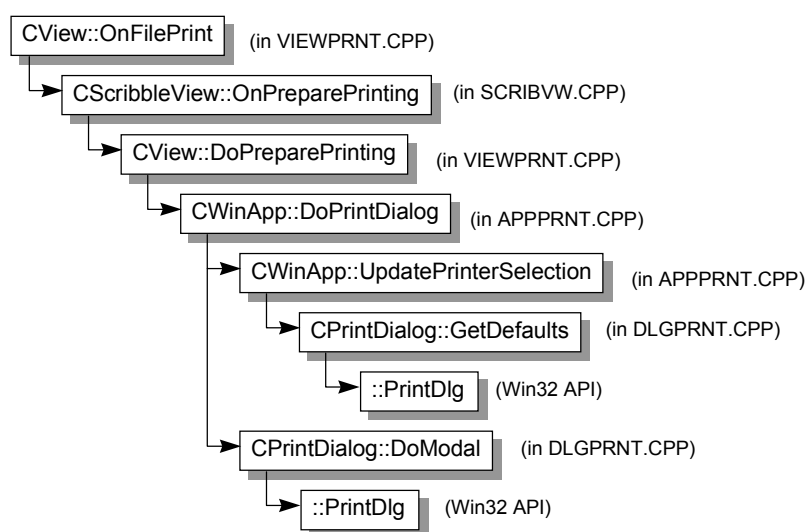
上面出現過的 *CPrintDialog*，用來更貼近描述列印對話盒：

```
class CPrintDialog : public CCommonDialog
{
public:
    PRINTDLG& m_pd;

    BOOL GetDefaults();
    LPDEVMODE GetDevMode() const; // return DEVMODE
    CString GetDriverName() const; // return driver name
    CString GetDeviceName() const; // return device name
    CString GetPortName() const; // return output port name
    HDC GetPrinterDC() const; // return HDC (caller must delete)
    HDC CreatePrinterDC();
    ...
};
```

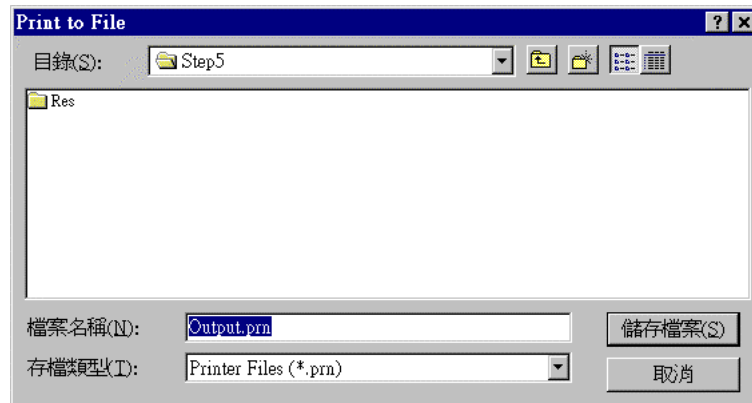
❷ 如果必要（從命令列參數中得知要直接列印某個檔案到印表機上），利用 *::CreateDC* 產生一個「印表機 DC」，並做列印動作。注意，*printInfo.m_bDirect* 被設為 *TRUE*，表示跳過列印對話盒，直接列印。

③ *OnPreparePrinting* 是一個虛擬函式，所以如果 *CView* 的衍生類別改寫了它，控制權就移轉到衍生類別手中。本例將移轉到 *CScribbleView* 手中。*CScribbleView::OnPreparePrinting* 的預設內容（AppWizard 自動為我們產生）是呼叫 *DoPreparePrinting*，它並不是虛擬函式，而是 *CView* 的一個輔助函式。以下是其呼叫堆疊，直至【列印】對話盒出現為止。



CView::DoPreparePrinting 將貯存在 *CPrintInfo* 結構中的對話盒 *CPrintDialog** *m_pPD* 顯示出來，借此收集使用者對印表機的各種設定，然後產生一個「印表機 DC」，儲存在 *printinfo.m_pPD->m_pd.hDC* 之中。

④ 如果使用者在【列印】對話盒中選按【列印到檔案】，則再顯示一個【Print to File】對話盒，讓使用者設定檔名。



⑤ 接下來取文件名稱和輸出設備的名稱（可能是印表機也可能是個檔案），並產生一個 *DOCINFO* 結構，設定其中的 *lpszDocName* 和 *lpszOutput* 欄位。此一 *DOCINFO* 結構將在稍後的 *StartDoc* 動作中用到。

⑥ 如果使用者在【列印】對話盒中按下【確定】鈕，*OnFilePrint* 就在堆疊中製造出一個 *CDC* 物件，並把前面所完成的「印表機 DC」附著到 *CDC* 物件上：

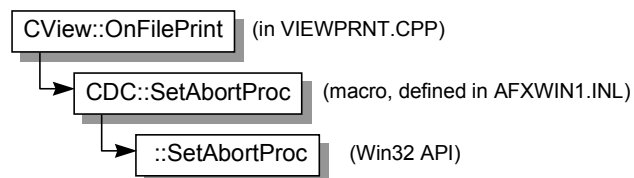
```

CDC dcPrint;
dcPrint.Attach(printInfo.m_pPD->m_pd.hDC);
dcPrint.m_bPrinting = TRUE;

```

⑦ 一旦 *CDC* 完成，*OnFilePrint* 把 *CDC* 物件以及前面的 *CPrintInfo* 物件傳給 *OnBeginPrinting* 作為參數。*OnBeginPrinting* 是 *CView* 的一個虛擬函式，原本什麼也沒做。你可以改寫它，設定列印前的任何初始狀態。

⑧ 設定 *AbortProc*。這應該是一個 callback 函式，MFC 有一個預設的簡易函式 *_AfxAbortProc* 可茲利用。

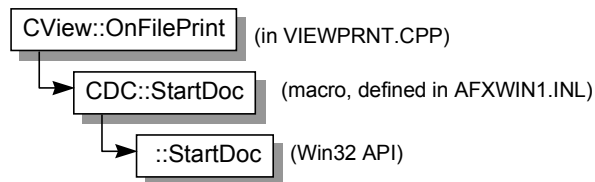


- ⑨ 把父視窗除能，產生【列印狀態】對話盒，根據文件名稱以及輸出設備名稱，設定對話盒內容，並顯示之：

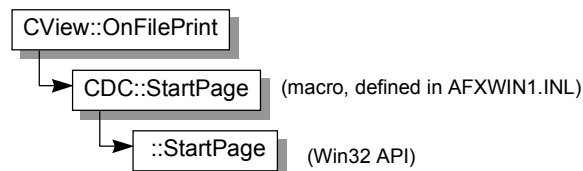
```
AfxGetMainWnd()->EnableWindow(FALSE);
CPrintingDialog dlgPrintStatus(this);
... // 設定對話盒內容
dlgPrintStatus.ShowWindow(SW_SHOW);
dlgPrintStatus.UpdateWindow();
```



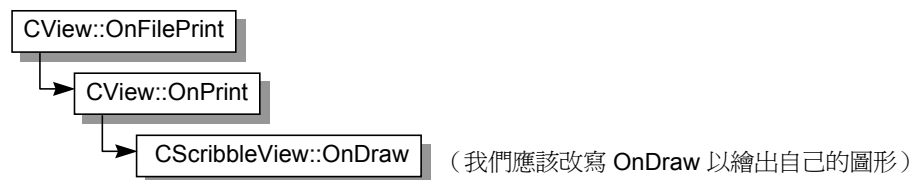
- ⑩ `StartDoc` 通知印表機開始嶄新的列印工作。這個函式其實就是啟動 Windows 列印引擎。



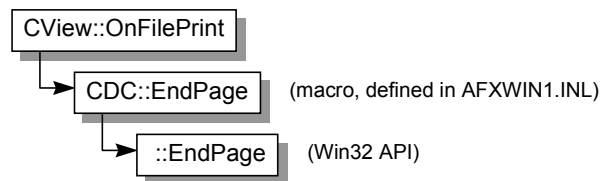
- ① 以 `for` 迴圈針對文件中的每一頁開始做列印動作。
- ② 呼叫 `CView::OnPrepareDC`。此函式什麼也沒做。如果你要在每頁前面加表頭，就請改寫這個虛擬函式。
- ③ 修改【列印狀態】對話盒中的頁次。
- ④ `StartPage` 開始新的一頁。



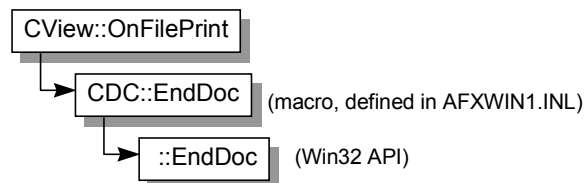
⑤ 呼叫 *CView::OnPrint*，它的內部只有一個動作：呼叫 *OnDraw*。我們應該在 *C ScribbleView* 中改寫 *OnDraw* 以繪出自己的圖形。



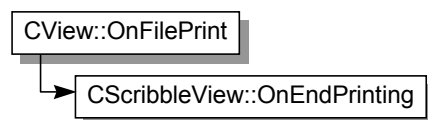
⑥ 一頁結束，呼叫 *dcPrint.EndPage*



⑦ 文件結束，呼叫 *EndDoc*



⑧ 整個列印工作結束。如果有些什麼繪圖資源需要釋放，你應該改寫 *OnEndPrinting* 函式並在其中釋放之。



⑨ 去除【列印狀態】對話盒。

⑩ 將「印表機 DC」解除附著，*CPrintInfo* 的解構式會把 DC 還給 Windows。

從上面這些分析中歸納出來的結論是，一共有六個虛擬函式可以改寫，請看圖 12-5。

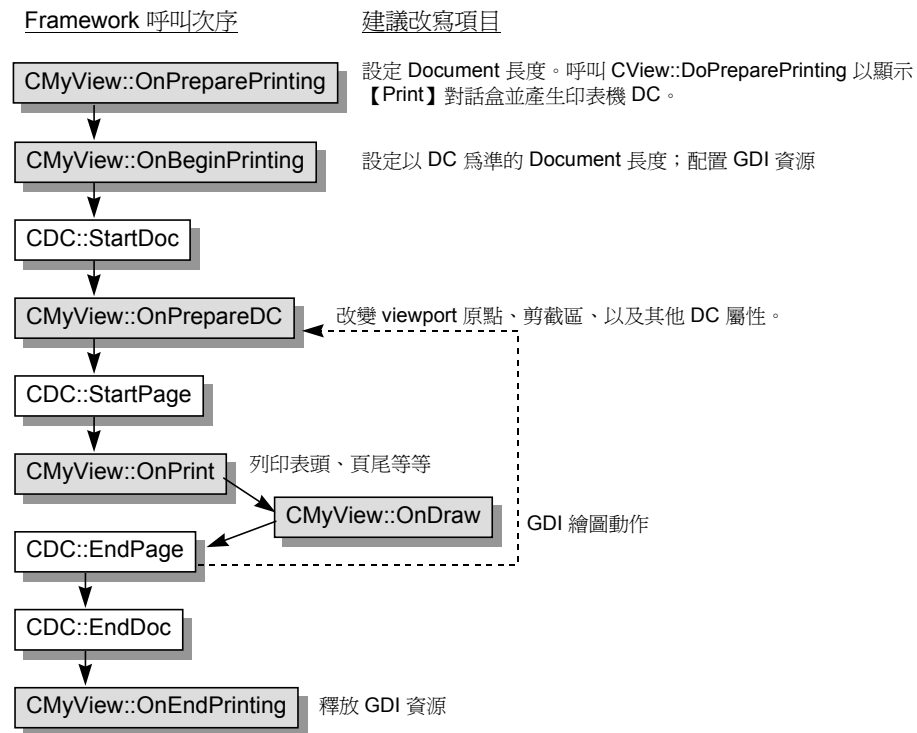


圖 12-5 MFC 列印流程與我們的著力點

以下是圖 12-5 的補充說明。

- 當使用者按下【File/Print】命令項，Application Framework 首先呼叫 *CMyView::OnPreparePrinting*。這個函式接受一個 *CPrintInfo* 指標做為參數，允許使用者設定 Document 的列印長度（從第幾頁到第幾頁）。預設頁碼是 1 至 0xFFFF，程式員應該在 *OnPreparePrinting* 中呼叫 *SetMaxPage* 預設頁數。*SetMaxPage* 之後，程式應該呼叫 *CView::DoPreparePrinting*，它會顯示【列印】對話盒，並產生一個印表機 DC。當對話盒結束，*CPrintInfo* 也從中獲得了使用者設定的各個印表項目（例如從第 n1 頁印到第 n2 頁）。

Framework 如何得知使用者對於列印狀態的設定？*CPrintInfo* 有五個函式可用，下一節有更詳細的說明。

- 針對每一頁，Framework 會呼叫 *CMyView::OnPrepareDC*，這函式在前一章介紹 *CScrollView* 時也曾提過，當時是因為我們使用捲動視窗，而由於捲動的關係，繪圖之前必須先設定 DC 的映像模式和原點等性質。這次稍有不同的是，它收到印表機 DC 做為第一參數，*CPrintInfo* 物件做為第二參數。我們改寫這個函式，使它依目前的頁碼來調整 DC，例如改變列印原點和截割區域以保證印出來的 Document 內容的合適性等等。
- 稍早我一再強調所有繪圖動作都應該集中在 *OnDraw* 函式中，Framework 會自動呼叫它。更精確地說，Framework 其實是先呼叫 *OnPrint*，傳兩個參數進去，第一參數是個 DC，第二參數是個 *CPrintInfo* 指標。*OnPrint* 內部再呼叫 *OnDraw*，這次只傳 DC 過去，做為唯一參數：

```
// in VIEWCORE.CPP
void CView::OnPrint(CDC* pDC, CPrintInfo*)
{
    ASSERT_VALID(pDC);

    // Override and set printing variables based on page number
    OnDraw(pDC);    // Call Draw
}
```

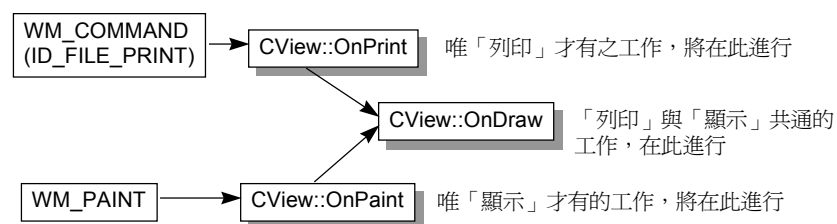
有了這樣的差異，我們可以這麼區分這兩個函式的功能：

- *OnPrint*：負責「只在印表時才做（螢幕顯示時不做）」的動作。例如印出表頭和頁尾。
- *OnDraw*：共通性繪圖動作（包括輸出到螢幕或印表機上）都在此完成。

看看另一個函式 *OnPaint*：

```
// in VIEWCORE.CPP
void CView::OnPaint()
{
    // standard paint routine
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

你會發現原來它們是這麼分工的：



所謂「顯示」是指輸出到螢幕上，「列印」是指輸出到印表機上。

由同一函式完成顯示（display）與列印（print）動作，才能夠達到「所見即所得」（What You See Is What You Get, WYSIWYG）的目的。如果你不需要一個 WYSIWYG 程式，可以改寫 *OnPrint* 使它不要呼叫 *OnDraw*，而呼叫另一個繪圖常式。

不要認為什麼情況下都需要 WYSIWYG。一個文字編輯器可能使用粗體字列印但使用控制碼在螢幕上代表這粗體字。

Scribble 列印機制的補強

MFC 預設的列印機制夠聰敏了，但還沒有聰敏到解決所有的問題。這些問題包括：


- 列印出來的影像可能不是你要的大小
- 不會分頁
- 沒有表頭（header）
- 沒有頁尾（footer）

畢竟螢幕輸出和印表機輸出到底還是有著重大的差異。視窗有捲動桿而印表機沒有，這伴隨而來的就是必須計算 `Document` 的大小和紙張的大小，以解決分頁的問題；此外，我們必須想想，在 MFC 預設的列印機制中，改寫哪一個地方，才能讓我們有辦法在 `Document` 的輸出頁加上表頭或頁尾。

印表機的頁和文件的頁


首先，我們必須區分「頁」對於 `Document` 和對於印表機的不同意義。從印表機觀點來看，一頁就是一張紙，然而一張紙並不一定容納 `Document` 的一頁。例如你想印一些通訊資料，這些資料可能是要被折疊起來的，因此一張紙印的是 `Document` 的第一頁和最後一頁（親愛的朋友，想想你每天看的報紙）。又例如印一個巨大的試算表，它可能是 `Document` 上的一頁，卻佔據兩張 A4 紙。

MFC 這個 Application Framework 把關於列印的大部份資訊都記錄在 `CPrintInfo` 中，其中數筆資料與分頁有密切關係。下表是取得分頁資料的相關成員，其中只有 `SetMaxPage` 和 `m_nCurPage` 和 `m_nNumPreviewPages` 在 Scribble 程式中會用到，原因是 Scribble 程式對許多問題做了簡化。

CPrintInfo 成員名稱	參考到的列印頁
GetMinPage/SetMinPage	Document 中的第一頁
GetMaxPage/SetMaxPage	Document 中的最後一頁
GetFromPage	將被印出的第一頁（出現在【列印】對話盒，  12-1b）
GetToPage	將被印出的最後一頁（出現在【列印】對話盒）
m_nCurPage	目前正被印出的一頁（出現在【列印狀態】對話盒）
m_nNumPreviewPages	預覽視窗中的頁數（稍後將討論之）

註：頁碼從 1（而不是 0）開始。

CPrintInfo 結構中記錄的「頁」數，指的是印表機的頁數；Framework 針對每一「頁」呼叫 *OnPrepareDC* 以及 *OnPrint* 時，所指的「頁」也是印表機的頁。當你改寫 *OnPreparePrinting* 時指定 Document 的長度，所用的單位也是印表機的「頁」。如果 Document 的一頁恰等於印表機的一頁（一張紙），事情就單純了；如果不是，你必須在兩者之間做轉換。

Scribble Step5 設定讓每一份 Document 使用印表機的兩頁。第一頁只是單純印出文件名稱（檔案名稱），第二頁才是文件內容。假設我利用 View 視窗捲動桿在整個 Document 四週畫一四方圈的話，我希望這一四方圈落入第二頁（第二張紙）中。當然，邊界留白必須考慮在內，如 12-6。除此之外，我希望第二頁（文件內容）最頂端留一點空間，做為表頭。本例在表頭中放的是檔案名稱。

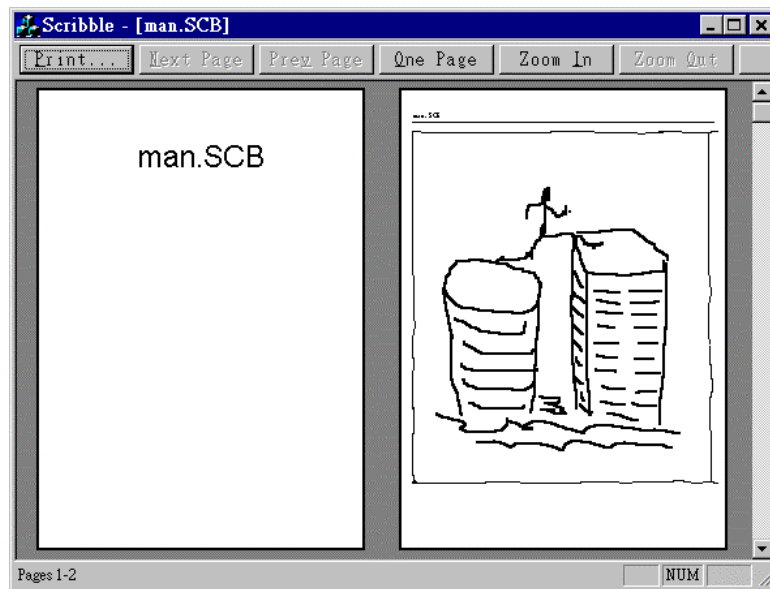


圖 12-6 Scribble Step5 的每一份文件列印時有兩頁，第一頁是文件名稱，第二頁是文件內容，最頂端留有一個表頭。

配置 GDI 繪圖工具

繪圖難免需要各式各樣的筆、刷、顏色、字形、工具。這些 GDI 資源都會佔用記憶體，而且是 GDI 模組的 heap。雖說 Windows 95 對於 USER 模組和 GDI 模組的 heap 已有大幅改善，使用 32 位元 heap，不再侷限 64KB，但我們當然仍然不希望看到浪費的情況發生，因此最好的方式就是在列印之前配置這些 GDI 繪圖物件，並在列印後立刻釋放。

看看圖 12-5，配置 GDI 物件的最理想時機顯然是 *OnBeginPrinting*，兩個理由：

1. 每當 Framework 開始一份新的列印工作，它就會呼叫此函式一次，因此不同列印工作所需的工具可在此有個替換。
2. 此函式的參數是一個和「印表機 DC」有附著關係的 CDC 物件指標，我們直接從此 CDC 物件中配置繪圖工具即可。

配置得來的 GDI 物件可以儲存在 View 的成員變數中，供整個列印過程使用。使用時機當然是 *OnPrint*。如果你必須對不同的列印頁使用不同的 GDI 物件，*CPrintInfo* 中的 *m_nCurPage* 可以幫你做出正確的決定。

釋放 GDI 物件的最理想時機當然是在 *OnEndPrinting*，這是每當一份列印工作結束後，Application Framework 會呼叫的函式。

Scribble 沒有使用什麼特殊的繪圖工具，因此下面這兩個虛擬函式也就沒有修改，完全保留 AppWizard 當初給我們的樣子：

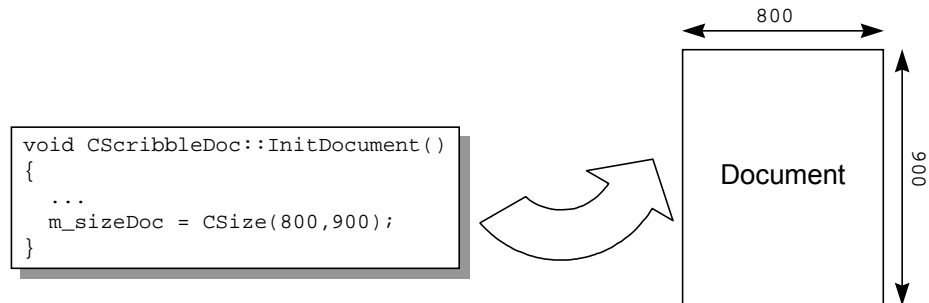
```
void CScribbleView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CScribbleView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}
```

凡司與方申：關於映像模式（座標系統）

回憶所謂的座標系統，我已經在上一章描述過 *CScrollView* 如何為了捲動效果而改變座標系統的原點。除了改變原點，我們甚至可以改變座標系統的單位長度，乃至於改變座標系統的橫縱比例（scale）。這些就是這一節要討論的重點。

Document 有大小可言嗎？有的，在列印過程中，為了計算 Document 對應到印表機的頁數，我們需要 Document 的尺寸。*CScribbleDoc* 的成員變數 *m_sizeDoc*，就是用來記錄 Document 的大小。它是一個 *CSize* 物件：



事實上，所謂「邏輯座標」原本是沒有大小的，如果我們說一份 Document 寬 800 高 900，那麼若邏輯座標的單位是英吋，這就是 8 英吋寬 9 英吋高；若邏輯座標的單位是公分，這就是 8 公分寬 9 公分高。如果邏輯單位是圖素 (Pixel) 呢？那就是 800 個圖素寬 900 個圖素高。圖素的大小隨著輸出裝置而改變，在 14 吋 Super VGA (1024x768) 顯示器上，800x900 個圖素大約是 21.1 公分寬 23.6 公分高，而在一部 300 DPI (Dot Per Inch，每英吋點數) 的雷射印表機上，將是 2-2/3 英吋寬 3 英吋高。

預設情況下 GDI 繪圖函式使用 *MM_TEXT* 映像模式 (Mapping Mode，也就是座標系統，註)，於是邏輯座標等於裝置座標，也就是說一個邏輯單位是一個圖素。如果不重新設定映像模式，可以想見螢幕上的圖形一放到 300 DPI 印表機上都嫌太小。

解決的方法很簡單：設定一種與真實世界相符的邏輯座標系統。Windows 提供的八種映像模式中有七種是所謂的 *metric* 映像模式，它們的邏輯單位都建立在公分或英吋的基礎上，這正是我們所要的。如果把 *OnDraw* 內的繪圖動作都設定在 *MM_LOENGLISH* 映像模式上 (每單位 0.01 英吋)，那麼不論輸出到螢幕上或到印表機上都獲得相同的尺度。真正要為「多少圖點才能畫出一英吋長」傷腦筋的是裝置驅動程式，不是我們。

註：GDI 的八種映像模式及其意義如下：

- ◆ **MM_TEXT**：以圖素（pixel）為單位，Y 軸向下為正，X 軸向右為正。
- ◆ **MM_LOMETRIC**：以 0.1 公分為單位，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_HIMETRIC**：以 0.01 公分為單位，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_LOENGLISH**：以 0.01 英吋為單位，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_HIENGLISH**：以 0.001 英吋為單位，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_TWIPS**：以 1/1440 英吋為單位，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_ISOTROPIC**：單位長度可任意設定，Y 軸向上為正，X 軸向右為正。
- ◆ **MM_ANISOTROPIC**：單位長度可任意設定，且 X 軸單位長可以不同於 Y 軸單位長（因此圖可能變形）。Y 軸向上為正，X 軸向右為正。

回憶上一章爲了捲動視窗，曾有這樣的動作：

```
void CScribbleView::OnInitialUpdate()
{
    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
    CScrollView::OnInitialUpdate();
}
```

映像模式可以在 *SetScrollSizes* 的第一個參數指定。現在我們把它改爲：

```
void CScribbleView::OnInitialUpdate()
{
    SetScrollSizes(MM_LOENGLISH, GetDocument()->GetDocSize());
    CScrollView::OnInitialUpdate();
}
```

注意，*OnInitialUpdate* 更在 *OnDraw* 之前被呼叫，也就是說我們在真正繪圖動作 *OnDraw* 之前完成了映像模式的設定。

映像模式不僅影響邏輯單位的尺寸，也影響 Y 軸座標方向。*MM_TEXT* 是 Y 軸向下，*MM_LOENGLISH*（以及其他任何映像模式）是 Y 軸向上。但，雖然有此差異，我們的 Step5 程式碼卻不需爲此再做更動，因爲 *DPtoLP* 已經完成了這個轉換。別忘了，滑鼠

左鍵傳來的點座標是先經過 *DPToLP* 才儲存到 *CStroke* 物件並且然後才由 *LineTo* 畫出的。

然而，程式的某些部份還是受到了 Y 軸方向改變的衝擊。映像模式只會改變 GDI 各相關函式，不使用 DC 的地方，就不受映像模式的影響，例如 *CRect* 的成員函式就不知曉所謂的映像模式。於是，本例中凡使用到 *CRect* 的地方，要特別注意做些調整：

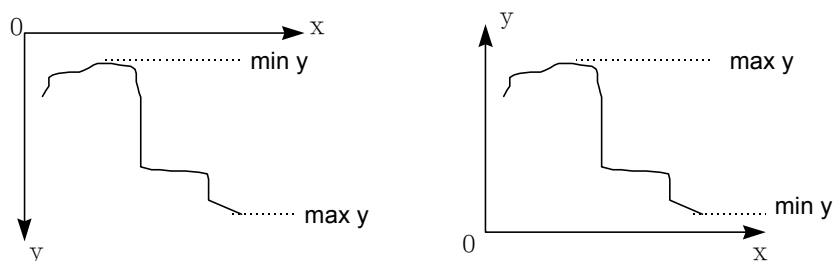
1. 修正「線條外圍四方形」的計算方式。原計算方式是在 *FinishStroke* 中這麼做：

```
for (int i=1; i < m_pointArray.GetSize(); i++)
{
    pt = m_pointArray[i];
    m_rectBounding.left   = min(m_rectBounding.left, pt.x);
    m_rectBounding.right  = max(m_rectBounding.right, pt.x);
    m_rectBounding.top    = min(m_rectBounding.top, pt.y);
    m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
}
m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
```

新的計算方式是：

```
for (int i=1; i < m_pointArray.GetSize(); i++)
{
    pt = m_pointArray[i];
    m_rectBounding.left   = min(m_rectBounding.left, pt.x);
    m_rectBounding.right  = max(m_rectBounding.right, pt.x);
    m_rectBounding.top    = max(m_rectBounding.top, pt.y);
    m_rectBounding.bottom = min(m_rectBounding.bottom, pt.y);
}
m_rectBounding.InflateRect(CSize(m_nPenWidth, -(int)m_nPenWidth));
```

這是因為在 Y 軸向下的系統中，四方的最頂點位置應該是找 Y 座標最小者；而在 Y 軸向上的系統中，四方的最頂點位置應該是找 Y 座標最大者；同理，對於四方的最底點亦然。



2. 我們在 *OnDraw* 中曾經以 *IntersectRect* 計算兩個四方形是否有交集。這個函式也是 *CRect* 成員函式，它假設：一個四方形的底座標 Y 值必然大於頂座標的 Y 值（這是從裝置座標，也就是 *MM_TEXT*，的眼光來看）；如果事非如此，它根本不可能找出兩個四方形的交集。因此我們必須在 *OnDraw* 中做以下修改，把邏輯座標改為裝置座標：

```
void CScribbleView::OnDraw(CDC* pDC)
{
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Get the invalidated rectangle of the view, or in the case
    // of printing, the clipping region of the printer dc.
    CRect rectClip;
    CRect rectStroke;
    pDC->GetClipBox(&rectClip);
    pDC->LPtoDP(&rectClip);
    rectClip.InflateRect(1, 1); // avoid rounding to nothing

    // Note: CScrollView::OnPaint() will have already adjusted the
    // viewport origin before calling OnDraw(), to reflect the
    // currently scrolled position.

    // The view delegates the drawing of individual strokes to
    // CStroke::DrawStroke().
    CTypedPtrList<CObList, CStroke*> strokeList = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        CStroke* pStroke = strokeList.GetNext(pos);
        rectStroke = pStroke->GetBoundingRect();
    }
}
```

```

        pDC->LPtoDP(&rectStroke);
        rectStroke.InflateRect(1, 1); // avoid rounding to nothing
        if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
            continue;
        pStroke->DrawStroke(pDC);
    }
}

```

分頁

Scribble 程式的 Document 大小固定是 800x900，而且我們讓它填滿印表機的一頁。因此 Scribble 並沒有「將 Document 分段列印」這種困擾。如果真要分段列印，Scribble 應該改寫 *OnPrepareDC*，在其中視列印的頁數調整 DC 的原點和截割區域。

即便如此，Scribble 還是在分頁方面加了一些動作。本例一份 Document 列印時被視為一張標題和一張圖片的組合，因此列印一份 Document 固定要耗掉兩張印表紙。我們可以這麼設計：

```

BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(2); // 文件總共有兩頁經線：
                          // 第一頁是標題頁 (title page)
                          // 第二頁是文件頁 (圖形)

    BOOL bRet = DoPreparePrinting(pInfo); // default preparation
    pInfo->m_nNumPreviewPages = 2; // Preview 2 pages at a time
    // Set this value after calling DoPreparePrinting to override
    // value read from .INI file
    return bRet;
}

```

接下來打算設計一個函式用以輸出標題頁，一個函式用以輸出文件頁。後者當然應該由 *OnDraw* 負責囉，但因為這文件頁不是單純的 Document 內容，還有所謂的表頭，而這是列印時才做的東西，螢幕顯示時並不需要的，所以我們希望把列印表頭的工作獨立於 *OnDraw* 之外，那麼最好的安置地點就是 *OnPrint* 了（請參考圖 12-5 之後的補充說明的最後一點）。

Scribble Step5 把列印表頭的工作獨立為一個函式。總共這三個額外的函式應該宣告於 SCRIBBLEVIEW.H 中，其中的 *PrintPageHeader* 在下一節列出。

```
class CScribbleView : public CScrollView
{
public:
    virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
    void PrintTitlePage(CDC* pDC, CPrintInfo* pInfo);
    void PrintPageHeader(CDC* pDC, CPrintInfo* pInfo, CString& strHeader);
    ...
}

#0001 void CScribbleView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
#0002 {
#0003     if (pInfo->m_nCurPage == 1) // page no. 1 is the title page
#0004     {
#0005         PrintTitlePage(pDC, pInfo);
#0006         return; // nothing else to print on page 1 but the page title
#0007     }
#0008     CString strHeader = GetDocument()->GetTitle();
#0009
#0010     PrintPageHeader(pDC, pInfo, strHeader);
#0011     // PrintPageHeader() subtracts out from the pInfo->m_rectDraw the
#0012     // amount of the page used for the header.
#0013
#0014     pDC->SetWindowOrg(pInfo->m_rectDraw.left, -pInfo->m_rectDraw.top);
#0015
#0016     // Now print the rest of the page
#0017     OnDraw(pDC);
#0018 }
#0019
#0020 void CScribbleView::PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
#0021 {
#0022     // Prepare a font size for displaying the file name
#0023     LOGFONT logFont;
#0024     memset(&logFont, 0, sizeof(LOGFONT));
#0025     logFont.lfHeight = 75; // 3/4th inch high in MM_LOENGLISH
#0026                             // (1/100th inch)
#0027     CFont font;
#0028     CFont* pOldFont = NULL;
#0029     if (font.CreateFontIndirect(&logFont))
#0030         pOldFont = pDC->SelectObject(&font);
#0031
#0032     // Get the file name, to be displayed on title page
#0033     CString strPageTitle = GetDocument()->GetTitle();
```

```

#0034
#0035      // Display the file name 1 inch below top of the page,
#0036      // centered horizontally
#0037      pDC->SetTextAlign(TA_CENTER);
#0038      pDC->TextOut(pInfo->m_rectDraw.right/2, -100, strPageTitle);
#0039
#0040      if (pOldFont != NULL)
#0041          pDC->SelectObject(pOldFont);
#0042  }

```

表頭與頁尾

文件名稱以及文件內容的頁碼應該有地方呈現出來。螢幕上沒有問題，文件名稱可以出現在視窗標題，頁碼可以出現在狀態列；但輸出到印表機上時，我們就應該設計文件的表頭與頁尾，分別用來放置文件名稱與頁碼，或其他任何你想要放的資料。顯然，即使是「所見即所得」，在印表機輸出與螢幕輸出兩方面仍然存在至少這樣的差異。

我們設計了另一個輔助函式，專門負責列印表頭，並將 *OnPrint* 的參數（一個印表機 DC）傳給它。有一點很容易被忽略，那就是你必得在 *OnPrint* 呼叫 *OnDraw* 之前調整視窗的原點和範圍，以避免該頁的主內容把表頭頁尾給蓋掉了。

要補償被表頭頁尾佔據的空間，可以利用 *CPrintInfo* 結構中的 *m_rectDraw*，這個欄位記錄著本頁的可繪圖區域。我們可以在輸出主內容之前先輸出表頭頁尾，然後扣除 *m_rectDraw* 四方形的一部份，代表表頭頁尾所佔空間。*OnPrint* 也可以根據 *m_rectDraw* 的數值決定有多少內容要放在列印頁的主體上。

我們甚至可能因為表頭頁尾的加入，而需要修改 *OnDraw*，因為能夠放到一張印表紙上的文件內容勢必將因為表頭頁尾的出現而減少。不過，還好本例並不是這個樣子。本例不設頁尾，而文件大小在 *MM_LOENGLISH* 映像模式下是 8 英吋寬 9 英吋高，放在一頁 A4 紙張（210 x 297 公厘）或 Letter Size（8-1/2 x 11 英吋）紙張中都綽綽有餘。

```

#0001 void CScribbleView::PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
#0002     CString& strHeader)
#0003 {
#0004     // Print a page header consisting of the name of
#0005     // the document and a horizontal line
#0006     pDC->SetTextAlign(TA_LEFT);
#0007     pDC->TextOut(0,-25, strHeader); // 1/4 inch down
#0008
#0009     // Draw a line across the page, below the header
#0010     TEXTMETRIC textMetric;
#0011     pDC->GetTextMetrics(&textMetric);
#0012     int y = -35 - textMetric.tmHeight; // line 1/10th inch below text
#0013     pDC->MoveTo(0, y); // from left margin
#0014     pDC->LineTo(pInfo->m_rectDraw.right, y); // to right margin
#0015
#0016     // Subtract out from the drawing rectangle the space used by the header.
#0017     y -= 25; // space 1/4 inch below (top of) line
#0018     pInfo->m_rectDraw.top += y;
#0019 }

```

動態計算頁碼

某些情況下 View 類別在開始列印之前沒辦法事先知道 Document 的長度。

假設你的程式並不支援「所見即所得」，那麼螢幕上的 Document 就不會對應到它列印時真正的長度。這就引起了一個問題，你沒有辦法在改寫 *OnPreparePrinting* 時，利用 *SetMaxPage* 為 *CPrintInfo* 結構設定一個最大頁碼，因為這時候的你根本不知道 Document 的長度。而如果使用者不能夠在【列印】對話盒中指定「結束頁碼」，Framework 也就不知道何時才停止列印的迴圈。唯一的方法就是邊印邊看，View 類別必須檢查是否目前已經印到 Document 的尾端，並在確定之後通知 Framework。

那麼我們的當務之急是找出在哪一個點上檢查 Document 結束與否，以及如何通知 Framework 停止列印。從圖 12-5 可知，列印的迴圈動作的第一個函式是 *OnPrepareDC*，我們可以改寫此一函式，在此設一道關卡，如果檢查出 Document 已到尾端，就要求中止列印。

Framework 是否結束列印，其實全賴 *CPrintInfo* 的 *m_bContinuePrinting* 欄位。此欄位如果是 *FALSE*，Framework 就中止列印。預設情況下 *OnPrepareDC* 把此欄位設為 *FALSE*。小心，這表示如果 Document 長度沒有指明，Framework 就假設這份 Document 只有一頁長。因此你在呼叫基礎類別的 *OnPrepareDC* 時需格外注意，可別總以為 *m_bContinuePrinting* 是 *TRUE*。

列印預覽 (Print Preview)

什麼是列印預覽？簡單地說，把螢幕模擬為印表機，將圖形輸出於其上就是了。預覽的目的是為了讓使用者在印表機輸出之前，先檢查他即將獲得的成果，檢查的重要項目包括圖案的佈局以及分頁是否合意。

為了完成預覽功能，MFC 在 *CDC* 之下設計了一個子類別，名為 *CPreviewDC*。所有其他的 *CDC* 物件都擁有兩個 DC，它們通常井水不犯河水；然而 *CPreviewDC* 就不同，它的第一個 DC 表示被模擬的印表機，第二個 DC 是真正的輸出目的地，也就是螢幕（預覽結果輸出到螢幕，不是嗎？！）

一旦你選擇【File/Print Preview】命令項，Framework 就產生一個 *CPreviewDC* 物件。只要你的程式曾經設定印表機 DC 的特徵（即使沒有動手設定，也有其預設值），Framework 就會把同樣的性質也設定到 Preview DC 上。舉個例子，你的程式選擇了某種列印字形，Framework 也會對螢幕選擇一個模擬印表機輸出的字形。一旦程式要做列印預覽，Framework 就透過模擬的印表機 DC，再把結果送到顯示幕 DC 去。

為什麼我不再像前面那樣去看整個預覽過程中的呼叫堆疊並追蹤其原始碼呢？因為預覽對我們而言太完善了，幾乎不必改寫什麼虛擬函式。唯一在 *Scribble Step5* 中與列印預覽有關係的，就是下面這一行：

```
BOOL CScribbleView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(2);    // the document is two pages long:
                           // the first page is the title page
                           // the second is the drawing
}
```



```
        BOOL bRet = DoPreparePrinting(pInfo);    // default preparation
        pInfo->m_NumPreviewPages = 2; // Preview 2 pages at a time
        // Set this value after calling DoPreparePrinting to override
        // value read from .INI file
        return bRet;
    }
```

現在，Scribble Step5 全部完成。

本章回顧

前面數章中早就有了列印功能，以及預覽功能。我們什麼也沒做，只不過在 AppWizard 的第四個步驟中選了【Printing and Print Preview】項目而已。這足可說明 MFC 為我們做掉了多少工作。想想看，一整個列印與預覽系統耶。

然而我們還是要為列印付出寫碼代價，原因是預設的列印大小不符理想，再者當我們想加點標題、表頭、頁尾時，必得親自動手。

延續前面的風格，我還是把 MFC 提供的列印系統的背後整個原理挖了出來，使你能夠清楚知道在哪裡下藥。在此之前，我也把 Windows 的列印原理（非關 MFC）整理出來，這樣你才有循序漸進的感覺。然後，我以各個小節解釋我們為 MFC 列印系統所做的補強工作。

現在的 Scribble，具備了繪圖能力，檔案讀寫能力，列印能力，預覽能力，豐富的視窗表現能力。除了 Online Help 以及 OLE 之外，所有大型軟體該具備的能力都有了。我並不打算在本書之中討論 Online Help，如果你有興趣，可以參考 *Visual C++ Tutorial*（可在 Visual C++ 的 Online 資料中獲得）第 10 章。

我也不打算在本書之中討論 OLE，那牽扯太多技術，不在本書的設定範圍。

Scribble Step5 的完整原始碼，列於附錄 B。

第 13 章

多文件與多顯示

你可能會以【Window/New Window】為同一份文件製造出另一個 View 視窗，也可能設計分裂視窗，以多個窗口呈現文件的不同角落（如第 11 章所為）。但，這兩種情況都是以相同的顯示方式表達文件的內容。

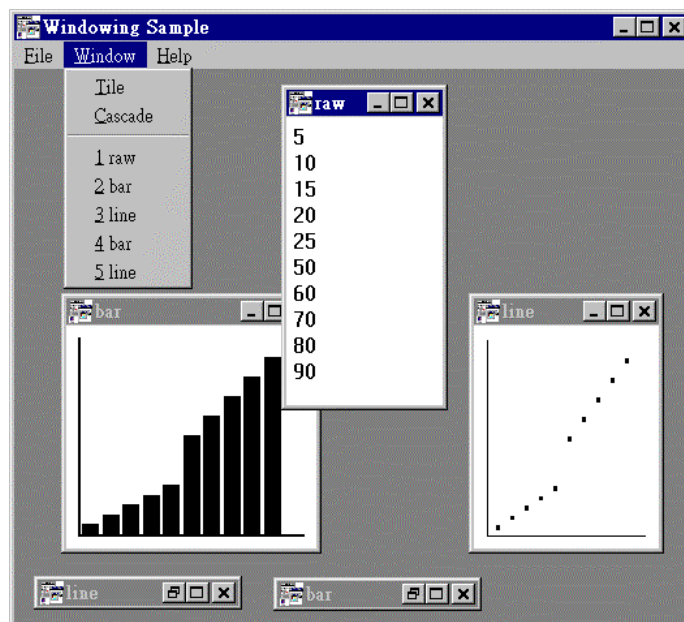
如何突破一成不變的顯示方法，達到豐富的表現效果？

這一章我將對於 Document/View 再作各種深入應用。重要放在顯象技術以及多重文件的技術上。

MDI 和 SDI

首先再讓我把 MDI 和 SDI 的觀念釐清楚。

在傳統的 SDK 程式設計中，所謂 MDI 是指「一個大外框視窗，內部可容納許多小子視窗」的這種程式風格。內部的小子視窗即是「Document 視窗」-- 雖然當時並未有如 MFC 所謂的 Document 觀念。此外，「MDI 風格」還包括程式必須有一個 Window 選單，提供對於小子視窗的管理，包括 tile、cascade、icon arrange 等命令項：



至於 SDI 程式，就是一般的、沒有上述風格的 non-MDI 程式。

在 MFC 的定義中，MDI 表示可「同時」開啓一份以上的 Documents，這些 Documents 可以是相同型態，也可以是不同型態。許多份 Documents 同時存在，必然需要許多個子視窗容納之，每個子視窗其實是 Document 的一個 View。即使你在 MDI 程式中只開啓一份 Document，但以【Window/New Window】的方式打開第二個 view、第三個 view...，亦需佔用多個子視窗。因此這和 SDK 所定義的 MDI 有異曲同工的意義。

至於 SDI 程式，同一時間只能開啓一份 Document。一份 Document 只佔用一個子視窗（亦即其 View 視窗），因此這也與 SDK 所定義的 SDI 意義相同。當你要在 SDI 程式中開啓第二份 Document，必須先把第一份 Document 關閉。

MDI 程式未必一定得提供一個以上的 Document 型態。所謂不同的 Document 型態是指程式提供不同的 *CDocument* 衍生類別，亦即有不同的 Document Template。軟體工業早期曾經流行一種「全效型」軟體，既處理試算表、又作文書處理、又能繪圖作畫；K 偉大得不得了，這種軟體就需要數種文件型態：試算表、文書、圖形；K

多重顯像 (Multiple Views)

只要是具備 MDI 性質的 MFC 程式(也就是你曾在 AppWizard 步驟一中選擇【Multiple Documents】項目)，天生就具備了「多重顯像」能力。「天生」的意思是你不必動手，application framework 已經內含了這項功能：隨便執行任何一版的 Scribble，你都可以在【Window】選單中找到【New Window】這個命令項，按下它，就可以獲得「同源子視窗」如圖 13-1。

我將以「多重顯像」來稱呼 Multiple Views。多重顯像的意思是資料可以不同的型態顯現出來。並以「同源子視窗」代表「顯示同一份 Document 而又各自分離的 View 視窗」。

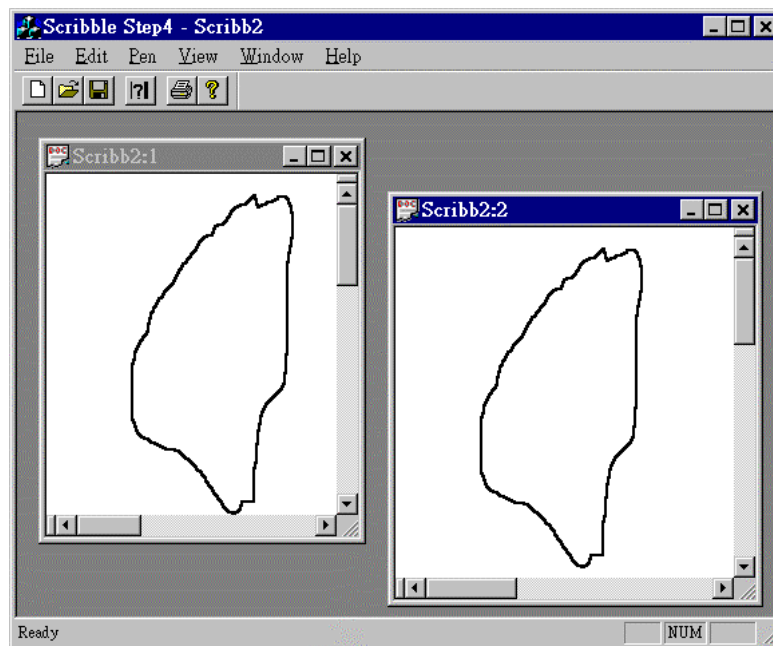


圖 13-1 【Window/New Window】可以為「目前作用中的 View 所對應的 Document 再開一個 View 視窗。」

另外，第 11 章也介紹了一種變化，是利用分裂視窗的各個窗口，顯示 Document 內容。這些窗口雖然集中在一個大視窗中，但牠們的視野卻可以各自獨立，也就是說牠們可以看到 Document 中的不同區域，如圖 13-2。

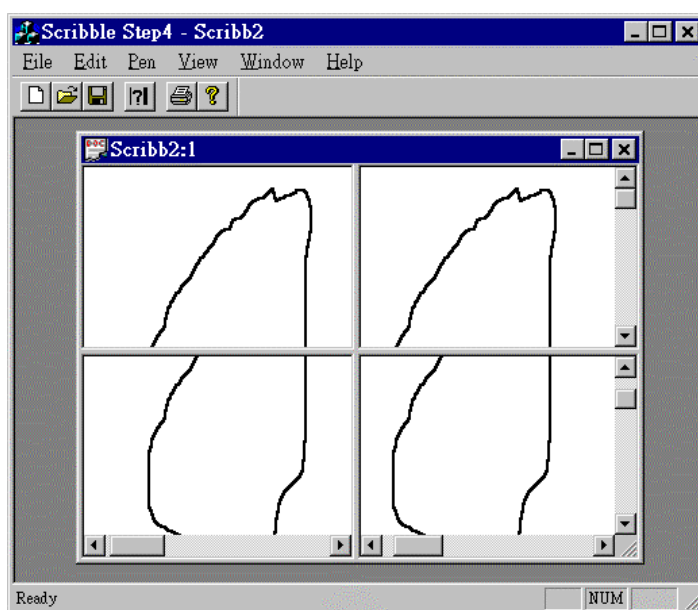


圖 13-2 分裂視窗的不同窗口可以觀察同一 Document 資料的不同區域。

但是我們發現，不論是同源子視窗或分裂視窗的窗口，都是以相同的方式（也就是同一個 `CMyView::OnDraw`）表現 Document 內容。如果我們希望表達力豐富一些，如何是好？到現在為止我們並沒有看到任何一個 Scribble 版本具備了多種顯像能力。

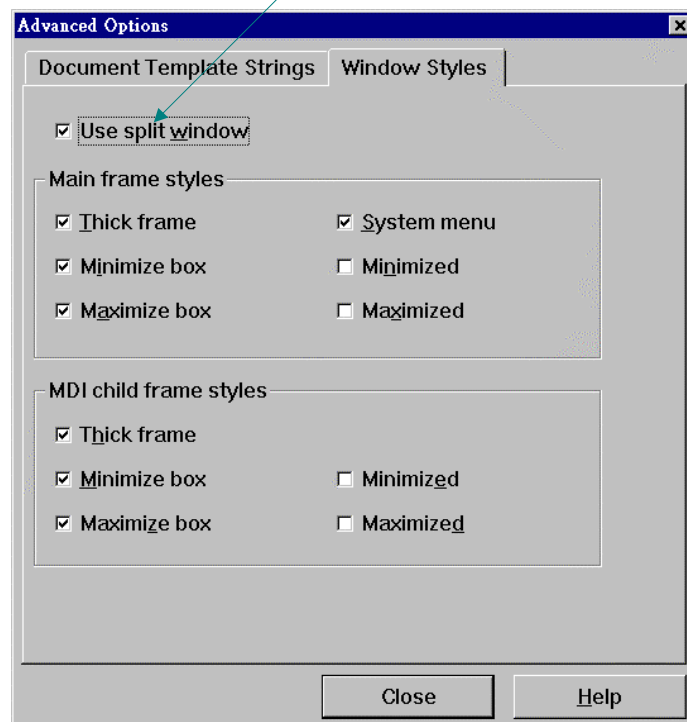
視窗的動態分裂

動態分裂視窗由 `CSplitterWnd` 提供服務。這項技術已經在第 11 章的 Scribble Step4 示範過了。它並沒有多重顯像的能力，因為每一個窗口所使用的 View 類別完全相同。當第

一個窗口形成（也就是分裂視窗初產生的時候），它將使用 Document Template 中登記的 View 類別，作為其 View 類別。爾後當分裂發生，也就是當使用者拖拉捲軸之上名為分裂棒（splitter box）的橫桿，導至新窗口誕生，程式就以「動態生成」的方式產生出新的 View 視窗。

因此，View 類別一定必須支援動態生成，也就是必須使用 `DECLARE_DYNCREATE` 和 `IMPLEMENT_DYNCREATE` 巨集。請回顧第 8 章。

AppWizard 支援動態分裂視窗。當你在 AppWizard 步驟四的【Advanced】對話盒的【Windows Styles】附頁中選按【Use split window】選項：



你的程式比起一般未選【Use split window】選項者有如下差異（陰影部份）：


```
// in CHILDFRM.H
class CChildFrame : public CMDIChildWnd
{
...
protected:
    CSplitterWnd m_wndSplitter;

public:
    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildFrame)
    public:
        virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL
    ...
};

// in CHILDFRM.CPP
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    return m_wndSplitter.Create( this,
        2, 2,          // TODO: adjust the number of rows, columns
        CSize( 10, 10 ), // TODO: adjust the minimum pane size
        pContext );
}

◆ CSplitterWnd::Create 的詳細規格請回顧第 11 章。
```

這些其實也就是我們在第 11 章為 **Scribble Step4** 親手加上的碼。如果你一開始就打定主意要使用動態分裂視窗，如上便是了。

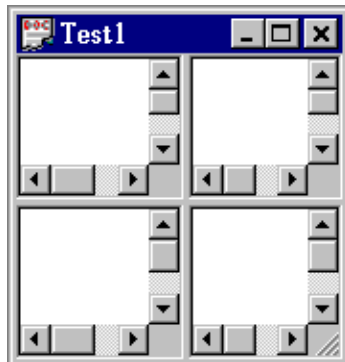
窗口 (Panels) 之間的同步更新，其機制著落在兩個虛擬函式 *CDocument::UpdateAllViews* 和 *CView::OnUpdate* 身上，與第 11 章的情況完全相同。

動態分裂的實作，非常簡單。但它實在稱不上「怎麼樣」！除了擁有「動態」增減窗口的長處之外，短處有二：第一，每一個窗口都使用相同的 **View** 類別，因此顯示出來的東西千篇一律；第二，窗口之間並非完全獨立。同一水平列的窗口，使用同一個垂直捲軸；同一垂直行的窗口，使用同一個水平捲軸，如圖 **13-2**。

視窗的靜態分裂

動態分裂視窗的短處正是靜態分裂視窗的長處，動態分裂視窗的長處正是靜態分裂視窗的短處。

靜態分裂視窗的窗口個數一開始就固定了，窗口所使用的 `view` 必須在分裂視窗誕生之際就準備好。每一個窗口的活動完全獨立自主，有完全屬於自己的水平捲軸和垂直捲軸。



靜態分裂視窗的窗口個數限制是 16 列 x 16 行，

動態分裂視窗的窗口個數限制是 2 列 x 2 行。

欲使用靜態分裂視窗，最方便的辦法就是先以 `AppWizard` 產生出動態分裂碼（如上一節所述），再修改其中部份程式。

不論動態分裂或靜態分裂，分裂視窗都由 `CSplitterWnd` 提供服務。動態分裂視窗的誕生是靠 `CSplitterWnd::Create`，靜態分裂視窗的誕生則是靠 `CSplitterWnd::CreateStatic`。爲了靜態分裂，我們應該把上一節由 `AppWizard` 產生的函式碼改變如下：

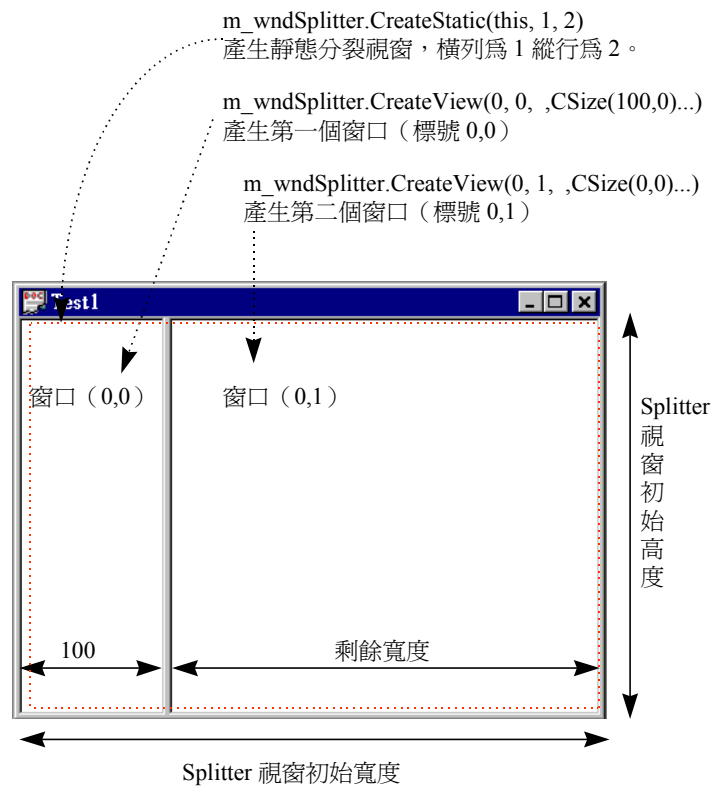
```

BOOL CChildFrame::OnCreateClient( LPCTSTR /*lpcs*/,
    CCreateContext* pContext)
{
    // 產生靜態分裂視窗，橫列為 1，縱行為 2。
    m_wndSplitter.CreateStatic(this, 1, 2);

    // 產生第一個窗口（標號 0,0）的 view 視窗。
    m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CTextView),
        CSize(100, 0), pContext);

    // 產生第二個窗口（標號 0,1）的 view 視窗。
    m_wndSplitter.CreateView(0, 1, RUNTIME_CLASS(CBarView),
        CSize(0, 0), pContext);
}
    
```

這會產生如下的分裂視窗：



CreateStatic 和 CreateView

靜態分裂用到兩個 *CSplitterWnd* 成員函式：

◆ CreateStatic：

這個函式的規格如下：

```
BOOL CreateStatic( CWnd* pParentWnd, int nRows, in nCols,
                  DWORD dwStyle = WS_CHILD | WS_VISIBLE,
                  UINT nID = AFX_IDW_PANE_FIRST );
```

第一個參數代表此分裂視窗之父視窗。第二和第三參數代表橫列和縱行的個數。第四個參數是視窗風格，預設為 *WS_CHILD | WS_VISIBLE*，第五個同時也是最後一個參數代表窗口（也是一個視窗）的 ID 起始值。

◆ CreateView

這個函式的規格如下：

```
virtual BOOL CreateView( int row, int col, CRuntimeClass* pViewClass,
                        SIZE sizeInit, CCreateContext* pContext );
```

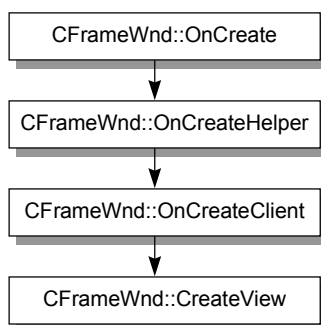
第一和第二參數代表窗口的標號（從 0 起算）。第三參數是 View 類別的 *CRuntimeClass* 指標，你可以利用 *RUNTIME_CLASS* 巨集（第 3 章和第 8 章提過）取此指標，也可以利用 *OnCreateClient* 的第二個參數 *CCreateContext* pContext* 所儲存的一個成員變數 *m_pNewViewClass*。你大概已經忘了這個變數吧，但我早提過它了，請看第 8 章的「*CDocTemplate* 管理 *CDocument* / *CView* / *CFrameWnd*」一節。所以，對於已在 *CMultiDocTemplate* 中登記過的 View 類別，此處可以這麼寫：

```
// 產生第一個窗口（標號 0,0）的 view 視窗。
m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CMyView),
                          CSize(100, 0), pContext);
```

也可以這麼寫：

```
m_wndSplitter.CreateView(0, 0, pContext->m_pNewViewClass,
                          CSize(100, 0), pContext);
```

讓我再多提醒你一些，第 8 章的「CDocTemplate 管理 CDocument / CView / CFrameWnd」一節主要是說明當使用者打開一份文件，MFC 內部有關於 Document / View / Frame「三位一體」的動態生成過程。其中 View 的動態生成是在 *CFrameWnd::OnCreate* 被喚起後，經歷一連串動作，最後才在 *CFrameWnd::CreateView* 中完成的：



而我們現在，為了分裂視窗，正在改寫其中第三個虛擬函式 *CFrameWnd::OnCreateClient* 呢！

好了，回過頭來，*CreateView* 的第四參數是窗口的初始大小，*CSize(100, 0)* 表示窗口寬度為 100 個圖素。高度倒是不為 0，對於橫列為 1 的分裂視窗而言，窗口高度永遠為視窗高度，Framework 並不理會你在 *CSize* 中寫了什麼高度。至於第二個窗口的大小 *CSize(0, 0)* 道理雷同，Framework 並不加理會其值，因為對於縱行為 2 的分裂視窗而言，右邊窗口的寬度永遠是視窗總寬度減去左邊窗口的寬度。

程式進行中如果需要窗口的大小，只要在 *OnDraw* 函式（通常是這裡需要）中這麼寫即可：

```
RECT rc; this->GetClientRect(&rc);
```

CreateView 的第五參數是 *CCreateContext* 指標。我們只要把 *OnCreateClient* 獲得的第二個參數依樣畫葫蘆地傳下去就是了。

視窗的靜態三分裂

分裂的方向可以無限延伸。我們可以把一個靜態分裂視窗的窗口再做靜態分裂，下面的程式碼展現了這種可能性：

```
// in header file
class CChildFrame : public CMDIChildWnd
{
...
protected:
    CSplitterWnd m_wndSplitter1;
    CSplitterWnd m_wndSplitter2;

public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildFrame)
public:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL
...
};

// in implementation file
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    // 產生靜態分裂視窗，橫列為 1，縱行為 2。
    m_wndSplitter1.CreateStatic(this, 1, 2);

    // 產生分裂視窗的第一個窗口（標號 0,0）的 view 視窗。
    m_wndSplitter1.CreateView(0, 0, RUNTIME_CLASS(CTextView),
        CSize(300, 0), pContext);

    // 產生第二個分裂視窗，橫列為 2，縱行為 1。位在第一個分裂視窗的（0,1）窗口
    m_wndSplitter2.CreateStatic(&m_wndSplitter1, 2, 1,
        WS_CHILD | WS_VISIBLE, m_wndSplitter1.IdFromRowCol(0, 1));

    // 產生第二個分裂視窗的第一個窗口（標號 0,0）的 view 視窗。
    m_wndSplitter2.CreateView(0, 0, RUNTIME_CLASS(CBarView),
        CSize(0, 150), pContext);

    // 產生第二個分裂視窗的第二個窗口（標號 1,0）的 view 視窗。
```

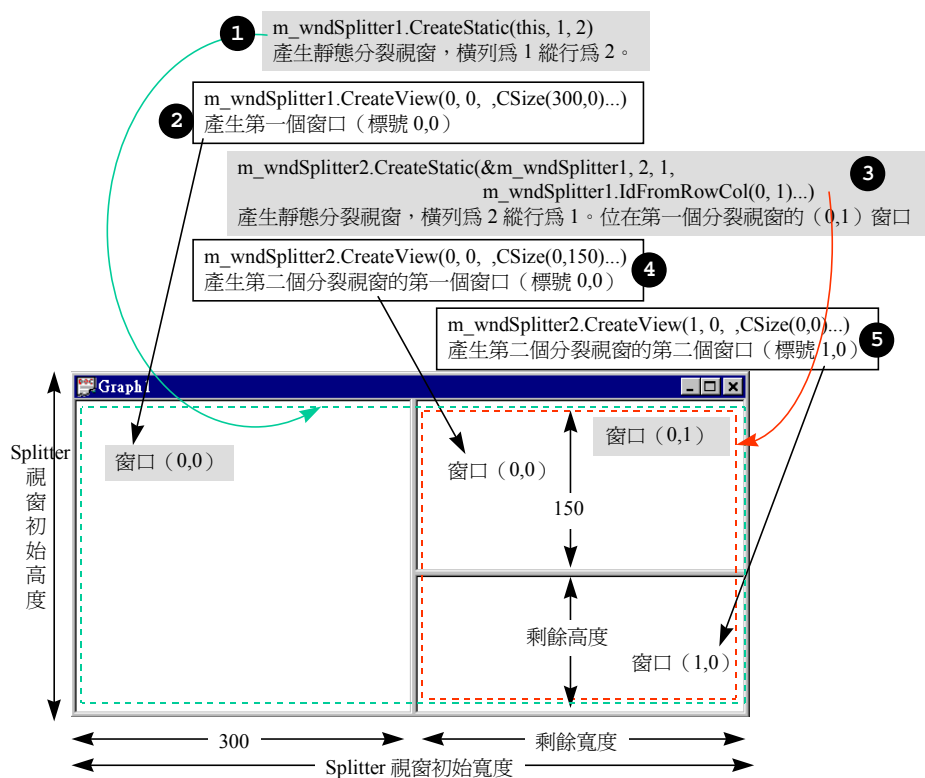
```

m_wndSplitter2.CreateView(1, 0, RUNTIME_CLASS(CCurveView),
    CSize(0, 0), pContext);

return TRUE;
}

```

這會產生如下的分裂視窗：



第二個分裂視窗的 ID 起始值可由第一個分裂視窗的窗口之一獲知（利用 *IdFromRowCol* 成員函式），一如上述程式碼中的動作。

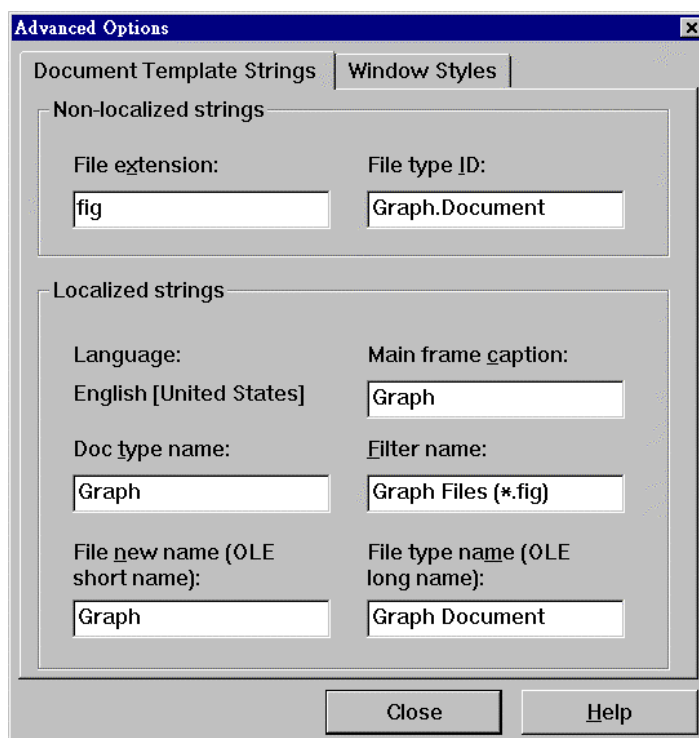
剩下的問題，就是如何設計許多個 View 類別了。

Graph 範例程式

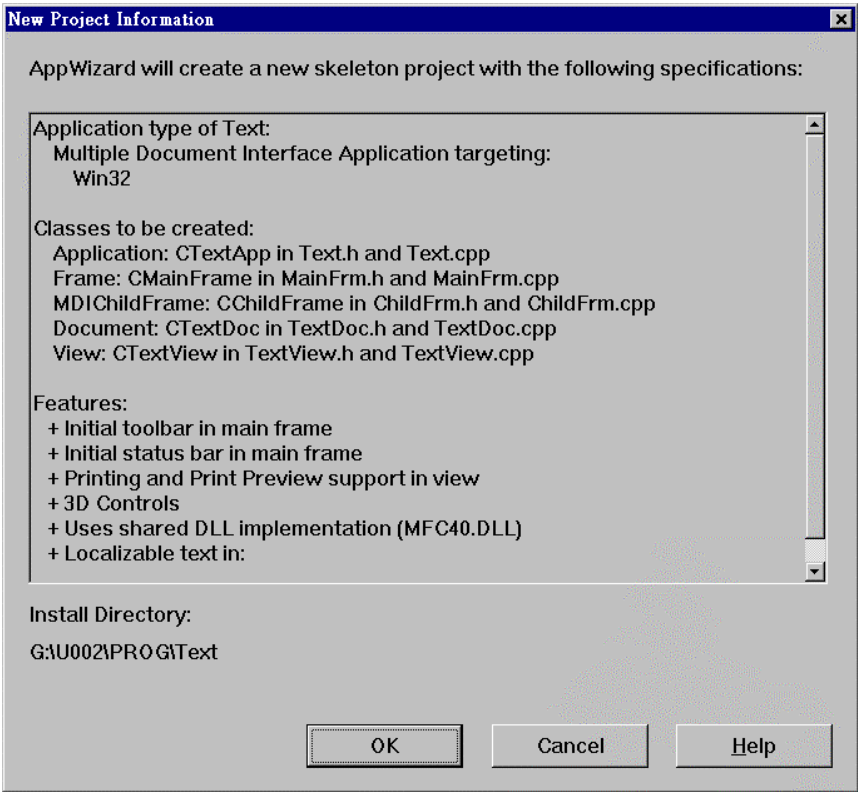
Graph 是一個具備靜態三叉分裂能力的程式。左側窗口以文字方式顯示 10 筆整數資料，右上側窗口顯示該 10 筆資料的長條圖，右下側窗口顯示對應的曲線圖。

進行至這一章，相信各位對於工具的基本操作技術都已經都熟練了，這裡我只列出 Graph 程式的製作大綱：

- 進入 AppWizard，製造一個 Graph 專案。採用預設的選項，但在第四步驟的【Advanced】對話盒的【Windows Styles】附頁中，將【Use split window】致能（enabled）起來。並填寫【Documents Template Strings】附頁如下：



最後，AppWizard 給我們這樣一份清單：



我們獲得的主要類別整理如下：

類別	基礎類別	檔案	
<i>CGraphApp</i>	<i>CWinApp</i>	GRAPH.CPP	GRAPH.H
<i>CMainFrame</i>	<i>CMDIFrameWnd</i>	MAINFRM.CPP	MAINFRM.H
<i>CChildFrame</i>	<i>CMDIChildWnd</i>	CHILDFRM.CPP	CHILDFRM.H
<i>CGraphDoc</i>	<i>CDocument</i>	GRAPHDOC.CPP	GRAPHDOC.H
<i>CGraphView</i>	<i>CView</i>	GRAPHVIEW.CPP	GRAPHVIEW.H

- 進入整合環境的 Resource View 視窗中，選擇 *IDR_GRAPHTYPE* 選單，在【Window】之前加入一個【Graph Data】選單，並添加三個項目，分別是：

選單項目名稱	識別碼 (ID)	提示字串
Graph Data&1	ID_GRAPH_DATA1	"Graph Data 1"
Graph Data&2	ID_GRAPH_DATA2	"Graph Data 2"
Graph Data&3	ID_GRAPH_DATA3	"Graph Data 3"

於是 GRAPH.RC 的選單資源改變如下：

```
IDR_GRAPHTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ...
    POPUP "&Graph Data"
    BEGIN
        MENUITEM "Data&1",    ID_GRAPH_DATA1
        MENUITEM "Data&2",    ID_GRAPH_DATA2
        MENUITEM "Data&3",    ID_GRAPH_DATA3
    END
    ...
END
```

- 回到整合環境的 Resource View 視窗，選擇 IDR_MAINFRAME 工具列，增加三個按鈕，放在 Help 按鈕之後，並使用工具箱上的 Draws Text 功能，為三個按鈕分別塗上 1, 2, 3 畫面：



這三個按鈕的 IDs 採用先前新增的三個選單項目的 IDs。

於是，GRAPH.RC 的工具列資源改變如下：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE 16, 15
BEGIN
    ...
    BUTTON    ID_FILE_PRINT
    BUTTON    ID_APP_ABOUT
    BUTTON    ID_GRAPH_DATA1
    BUTTON    ID_GRAPH_DATA2
    BUTTON    ID_GRAPH_DATA3
END
```

■ 進入 ClassWizard，為新增的這些 UI 物件製作 Message Map。由於這些命令項會影響到我們的 Document 內容（當使用者按下 Data1，我們必須為他準備一份相關資料；按下 Data2，我們必須再為他準備一份相關資料），所以在 CGraphDoc 中處理這些命令訊息甚為合適：

UI 物件	Messages	訊息處理常式
ID_GRAPH_DATA1	COMMAND	OnGraphData1
	UPDATE_COMMAND_UI	OnUpdateGraphData1
ID_GRAPH_DATA2	COMMAND	OnGraphData2
	UPDATE_COMMAND_UI	OnUpdateGraphData2
ID_GRAPH_DATA3	COMMAND	OnGraphData3
	UPDATE_COMMAND_UI	OnUpdateGraphData3

原始碼改變如下：

```
// in GRAPHDOC.H
class CGraphDoc : public CDocument
{
...
// Generated message map functions
protected:
    //{AFX_MSG(CGraphDoc)
    afx_msg void OnGraphData1();
    afx_msg void OnGraphData2();
    afx_msg void OnGraphData3();
    afx_msg void OnUpdateGraphData1(CCmdUI* pCmdUI);
    afx_msg void OnUpdateGraphData2(CCmdUI* pCmdUI);
    afx_msg void OnUpdateGraphData3(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

// in GRAPHDOC.CPP
BEGIN_MESSAGE_MAP(CGraphDoc, CDocument)
    //{AFX_MSG_MAP(CGraphDoc)
    ON_COMMAND(ID_GRAPH_DATA1, OnGraphData1)
    ON_COMMAND(ID_GRAPH_DATA2, OnGraphData2)
    ON_COMMAND(ID_GRAPH_DATA3, OnGraphData3)
    ON_UPDATE_COMMAND_UI(ID_GRAPH_DATA1, OnUpdateGraphData1)
```

```

        ON_UPDATE_COMMAND_UI ( ID_GRAPH_DATA2, OnUpdateGraphData2)
        ON_UPDATE_COMMAND_UI ( ID_GRAPH_DATA3, OnUpdateGraphData3)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

- 利用 ClassWizard 產生兩個新類別，做為三叉分裂視窗中的另兩個窗口的 View 類別：

類別名稱	基礎類別	檔案
<i>CTextView</i>	<i>CView</i>	TEXTVIEW.CPP TEXTVIEW.H
<i>CBarView</i>	<i>CView</i>	BARVIEW.CPP BARVIEW.H

- 改寫 *CChildFrame::OnCreateClient* 函式如下（這是本節的技術重點）：

```

#include "stdafx.h"
#include "Graph.h"

#include "ChildFrm.h"
#include "TextView.h"
#include "BarView.h"
...
BOOL CChildFrame::OnCreateClient( LPCTSTR /*lpcs*/,
    CCreateContext* pContext)
{
    // 產生靜態分裂視窗，橫列為 1，縱行為 2。
    m_wndSplitter1.CreateStatic(this, 1, 2);

    // 產生分裂視窗的第一個窗口（標號 0,0）的 view 視窗，採用 CTextView。
    m_wndSplitter1.CreateView(0, 0, RUNTIME_CLASS(CTextView),
        CSize(300, 0), pContext);

    // 產生第二個分裂視窗，橫列為 2 縱行為 1。位在第一個分裂視窗的（0,1）窗口
    m_wndSplitter2.CreateStatic(&m_wndSplitter1, 2, 1,
        WS_CHILD | WS_VISIBLE, m_wndSplitter1.IdFromRowCol(0, 1));

    // 產生第二個分裂視窗的第一個窗口（標號 0,0）的 view 視窗，採用 CBarView。
    m_wndSplitter2.CreateView(0, 0, RUNTIME_CLASS(CBarView),
        CSize(0, 150), pContext);

    // 產生第二個分裂視窗的第二個窗口（標號 1,0）的 view 視窗，採用 CGraphView。
    m_wndSplitter2.CreateView(1, 0, pContext->m_pNewViewClass,
        CSize(0, 0), pContext);
}

```

```
// 設定 active pane
SetActiveView((CView*)m_wndSplitter1.GetPane(0,0));
return TRUE;
}
```

爲什麼最後一次 *CreateView* 時我以 *pContext->m_pNewViewClass* 取代 *RUNTIME_CLASS(CGraphView)* 呢？後者當然也可以，但卻因此必須含入 *CGraphView* 的宣告；而如果你因爲這個原因而含入 *GraphView.h* 檔，又會產生三個編譯錯誤，挺麻煩！

- 至此，Document 中雖然沒有任何資料，但程式的 UI 已經完備，編譯聯結後可得以下執行畫面：



- 修改 *CGraphDoc*，增加一個整數陣列 *m_intArray*，這是真正存放資料的地方，我採用 MFC 內建的 *CArray<int,int>*，爲此，必須在 *STDAFX.H* 中加上一行：

```
#include <afxtempl.h> // MFC templates
```

爲了設定陣列內容，我又增加了一個 *SetValue* 成員函式，並且在【Graph Data】選單命令被執行時，爲 *m_intArray* 設定不同的初值：

```
// in GRAPHDOC.H
class CGraphDoc : public CDocument
{
...
public:
    CArray<int,int> m_intArray;

public:
    void SetValue(int i0, int i1, int i2, int i3, int i4,
                  int i5, int i6, int i7, int i8, int i9);
    ...
};

// in GRAPHDOC.CPP
CGraphDoc::CGraphDoc()
{
    SetValue(5, 10, 15, 20, 25, 78, 64, 38, 29, 9);
}

void CGraphDoc::SetValue(int i0, int i1, int i2, int i3, int i4,
                          int i5, int i6, int i7, int i8, int i9)
{
    m_intArray.SetSize(DATANUM, 0);
    m_intArray[0] = i0;
    m_intArray[1] = i1;
    m_intArray[2] = i2;
    m_intArray[3] = i3;
    m_intArray[4] = i4;
    m_intArray[5] = i5;
    m_intArray[6] = i6;
    m_intArray[7] = i7;
    m_intArray[8] = i8;
    m_intArray[9] = i9;
}

void CGraphDoc::OnGraphData1()
{
    SetValue(5, 10, 15, 20, 25, 78, 64, 38, 29, 9);
    UpdateAllViews(NULL);
}

void CGraphDoc::OnGraphData2()
{
    SetValue(50, 60, 70, 80, 90, 23, 68, 39, 73, 58);
}
```

```

        UpdateAllViews(NULL);
    }
    void CGraphDoc::OnGraphData3()
    {
        SetValue(12, 20, 8, 17, 28, 37, 93, 45, 78, 29);
        UpdateAllViews(NULL);
    }
    void CGraphDoc::OnUpdateGraphData1(CCmdUI* pCmdUI)
    {
        pCmdUI->SetCheck(m_intArray[0] == 5);
    }
    void CGraphDoc::OnUpdateGraphData2(CCmdUI* pCmdUI)
    {
        pCmdUI->SetCheck(m_intArray[0] == 50);
    }
    void CGraphDoc::OnUpdateGraphData3(CCmdUI* pCmdUI)
    {
        pCmdUI->SetCheck(m_intArray[0] == 12);
    }

```

各位看到，爲了方便，我把 *m_intArray* 的資料封裝屬性設爲 *public* 而非 *private*，檢查「*m_intArray* 內容究竟是哪一份資料」所用的方法也非常粗糙，呀，不要非難我，重點不在這裡呀！

- 在 RESOURCE.H 檔案中加上兩個常數定義：

```

#define DATANUM 10
#define DATAMAX 100

```

- 修改 *CGraphView*，在 *OnDraw* 成員函式中取得 *Document*，再透過 *Document* 物件指標取得整數陣列，然後將 10 筆資料的曲線圖繪出：

```

#0001 void CGraphView::OnDraw(CDC* pDC)
#0002 {
#0003     CGraphDoc* pDoc = GetDocument();
#0004     ASSERT_VALID(pDoc);
#0005
#0006     int      cxDot,cxDotSpacing,cyDot, cxGraph,cyGraph, x,y, i;
#0007     RECT      rc;
#0008
#0009     CPen  pen (PS_SOLID, 1, RGB(255, 0, 0)); // red pen
#0010     CBrush brush(RGB(255, 0, 0));           // red brush

```

```

#0011     CBrush* pOldBrush = pDC->SelectObject(&brush);
#0012     CPen*   pOldPen   = pDC->SelectObject(&pen);
#0013
#0014     cxGraph = 100;
#0015     cyGraph = DATAMAX; // defined in resource.h
#0016
#0017     this->GetClientRect(&rc);
#0018     pDC->SetMapMode(MM_ANISOTROPIC);
#0019     pDC->SetWindowOrg(0, 0);
#0020     pDC->SetViewportOrg(10, rc.bottom-10);
#0021     pDC->SetWindowExt(cxGraph, cyGraph);
#0022     pDC->SetViewportExt(rc.right-20, -(rc.bottom-20));
#0023
#0024     // 我們希望圖形佔滿視窗的整個可用空間（以水平方向為準）
#0025     // 並希望曲線點的寬度是點間距寬度的 1.2，
#0026     // 所以 (dot_spacing + dot_width) * num_datapoints = graph_width
#0027     // 亦即 dot_spacing * 3/2 * num_datapoints = graph_width
#0028     // 亦即 dot_spacing = graph_width / num_datapoints * 2/3
#0029
#0030     cxDotSpacing = (2 * cxGraph) / (3 * DATANUM);
#0031     cxDot = cxDotSpacing/2;
#0032     if (cxDot<3) cxDot = 3;
#0033     cyDot = cxDot;
#0034
#0035     // 座標軸
#0036     pDC->MoveTo(0, 0);
#0037     pDC->LineTo(0, cyGraph);
#0038     pDC->MoveTo(0, 0);
#0039     pDC->LineTo(cxGraph, 0);
#0040
#0041     // 畫出資料點
#0042     pDC->SelectObject(::GetStockObject (NULL_PEN));
#0043     for (x=0+cxDotSpacing,y=0,i=0; i<DATANUM; i++,x+=cxDot+cxDotSpacing)
#0044         pDC->Rectangle(x, y+pDoc->m_intArray[i],
#0045                        x+cxDot, y+pDoc->m_intArray[i]-cyDot);
#0046
#0047     pDC->SelectObject(pOldBrush);
#0048     pDC->SelectObject(pOldPen);
#0049 }

```

- 修改 *CTextView* 程式碼，在 *OnDraw* 成員函式中取得 *Document*，再透過 *Document* 物件指標取得整數陣列，然後將 10 筆資料以文字方式顯示出來：

```

#0001 #include "stdafx.h"
#0002 #include "Graph.h"

```



```
#0003 #include "GraphDoc.h"
#0004 #include "TextView.h"
#0005 ...
#0006 void CTextView::OnDraw(CDC* pDC)
#0007 {
#0008     CGraphDoc* pDoc = (CGraphDoc*)GetDocument();
#0009
#0010     TEXTMETRIC tm;
#0011     int x,y, cy, i;
#0012     char sz[20];
#0013     pDC->GetTextMetrics(&tm);
#0014     cy = tm.tmHeight;
#0015     pDC->SetTextColor(RED); // red text
#0016     for (x=5,y=5,i=0; i<DATANUM; i++,y+=cy)
#0017     {
#0018         wsprintf (sz, "%d", pDoc->m_intArray[i]);
#0019         pDC->TextOut (x,y, sz, lstrlen(sz));
#0020     }
#0021 }
```

- 修改 *CBarView* 程式碼，在 *OnDraw* 成員函式中取得 *Document*，再透過 *Document* 物件指標取得整數陣列，然後將 10 筆資料以長條圖繪出：

```
#0001 #include "stdafx.h"
#0002 #include "Graph.h"
#0003 #include "GraphDoc.h"
#0004 #include "TextView.h"
#0005 ...
#0006 void CBarView::OnDraw(CDC* pDC)
#0007 {
#0008     CGraphDoc* pDoc = (CGraphDoc*)GetDocument();
#0009
#0010     int cxBar,cxBarSpacing, cxGraph,cyGraph, x,y, i;
#0011     RECT rc;
#0012
#0013     CBrush brush(RED); // red brush
#0014     CBrush* pOldBrush = pDC->SelectObject(&brush);
#0015     CPen pen(PEN_SOLID, 1, RED); // red pen
#0016     CPen* pOldPen = pDC->SelectObject(&pen);
#0017
#0018     cxGraph = 100;
#0019     cyGraph = DATAMAX; // defined in resource.h
#0020
#0021     this->GetClientRect(&rc);
#0022     pDC->SetMapMode(MM_ANISOTROPIC);
```

```

#0023     pDC->SetWindowOrg(0, 0);
#0024     pDC->SetViewportOrg(10, rc.bottom-10);
#0025     pDC->SetWindowExt(cxGraph, cyGraph);
#0026     pDC->SetViewportExt(rc.right-20, -(rc.bottom-20));
#0027
#0028     // 長條圖的條狀物之間距離是條狀物寬度的 1/3。
#0029     // 我們希望條狀物能夠填充視窗的整個可用空間。
#0030     // 所以 (bar_spacing + bar_width) * numBars = graph_width
#0031     // 亦即 bar_width * 4/3 * numBars = graph_width
#0032     // 亦即 bar_width = graph_width / numBars * 3/4
#0033
#0034     cxBar = (3 * cxGraph) / (4 * DATANUM);
#0035     cxBarSpacing = cxBar/3;
#0036     if (cxBar<3) cxBar=3;
#0037
#0038     // 座標軸
#0039     pDC->MoveTo(0, 0);
#0040     pDC->LineTo(0, cyGraph);
#0041     pDC->MoveTo(0, 0);
#0042     pDC->LineTo(cxGraph, 0);
#0043
#0044     // 長條圖
#0045     for (x=0+cxBarSpacing,y=0,i=0; i< DATANUM; i++,x+=cxBar+cxBarSpacing)
#0046         pDC->Rectangle(x, y, x+cxBar, y+pDoc->m_intArray[i]);
#0047
#0048     pDC->SelectObject(pOldPen);
#0049     pDC->SelectObject(pOldBrush);
#0050 }

```

- 如果你要令三個 view 都有列印預視能力，必須在每一個 view 類別中改寫以下三個虛擬函式：

```

virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

```

至於其函式內容，從 *CGraphView* 的同名函式中依樣畫葫蘆拷貝一份過來即可。

- 本例不示範檔案讀寫動作，所以 *CGraphDoc* 沒有改寫 *Serialize* 虛擬函式。

圖 13-3 是 Graph 程式的執行畫面。

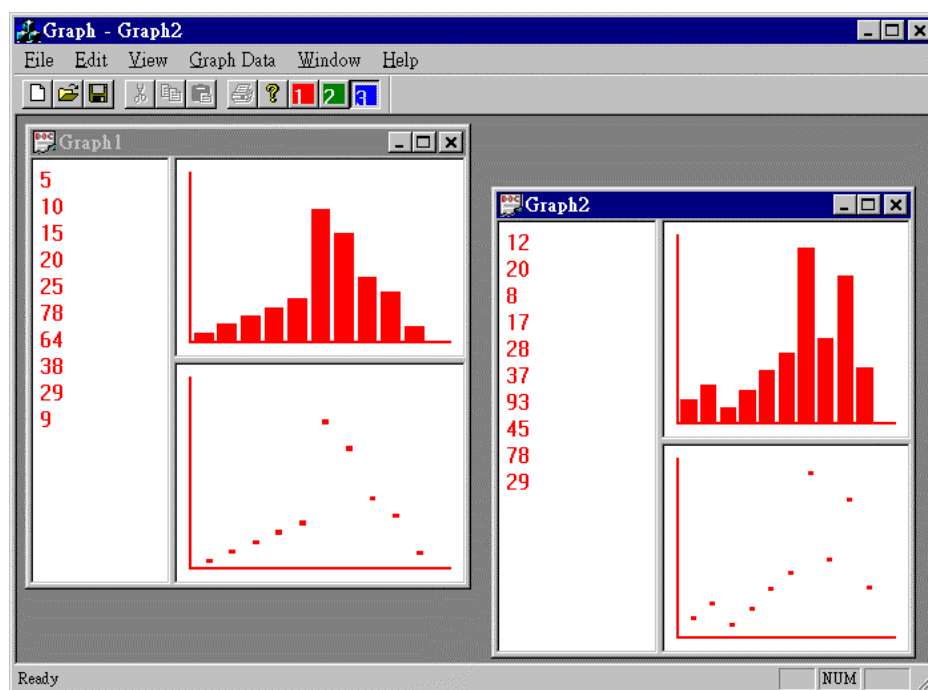


圖 13-3 Graph 執行畫面。每當選按【File/New】（或工具列上的對應按鈕）打開一份新文件，其內就有 10 筆整數資料。你可選按【Graph Data】（或工具列上的對應按鈕）改變資料內容。

靜態分裂視窗之觀念整理

我想你已經從前述的 *OnCreateClient* 函式中認識了靜態分裂視窗的相關函式。我可以用圖 13-4 解釋各個類別的關係與運用。

基本上圖 13-4 三個窗口可以視為三個完全獨立的 view 視窗，有各自的類別，以各自的方式顯示資料。不過，資料倒是來自同一份 Document。試試看預視效果，你會發現，哪一個窗口為「作用中」，哪一個窗口的繪圖動作就主宰預視視窗。你可以利用 *SetActivePane* 設定作用中的窗口，也可以呼叫 *GetActivePane* 獲得作用中的窗口。但是，你會發現，從外觀上很難看出哪一個窗口是「作用中的」窗口。

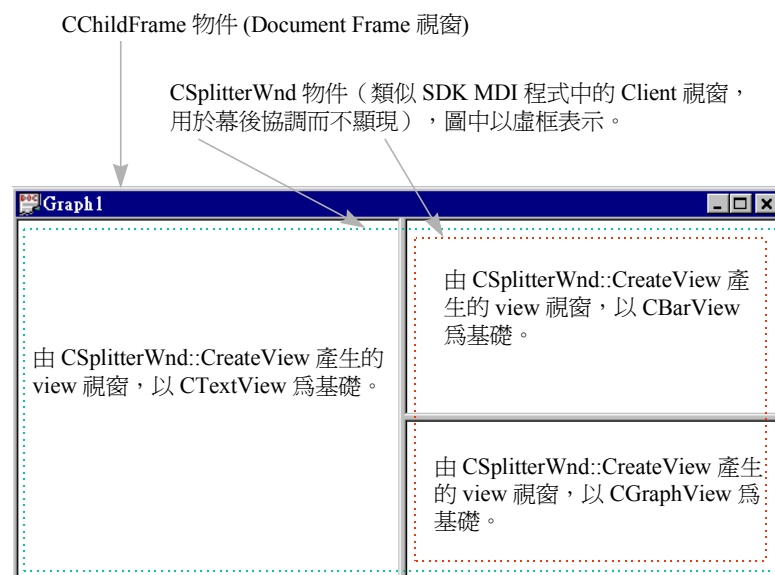


圖 13-4 靜態分裂視窗的類別運用 (以 Graph 為例)

原子視窗

雖然我說靜態分裂視窗的窗口可視為完全獨立的 view 視窗，但畢竟它們不是！它們還框在一個大視窗中。如果你不喜歡分裂視窗（誰知道呢，當初我也不太喜歡），我們來試點新鮮的。

點子是從【Window/New Window】開始。這個選單項目令 Framework 為我們做出目前作用中的 view 視窗的另一份拷貝。如果我們能夠知道 Framework 是如何動作，是不是可以引導它使用另一個 view 類別，以不同的方式表現同一份資料？

這就又有偷窺原始碼的需要了。MFC 並沒有提供正常的管道讓我們這麼做，我們需要 MFC 原始碼。

CMDIFrameWnd::OnWindowNew

如果你想在程式中設計中斷點，一步一步找出【Window/New Window】的動作，就像我在第 12 章對付 *OnFilePrint* 和 *OnFilePrintPreview* 一樣，那麼你會發現沒有著力點，因為 AppWizard 並不會做出像這樣的訊息映射表格：

```
BEGIN_MESSAGE_MAP(CScribbleView, CScrollView)
...
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

你根本不知道【Window/New Window】這個命令流到哪裡去了。第 7 章的「標準選單 File / Edit / View / Window / Help」一節，也曾說過這個命令項是屬於「與 Framework 預有關聯型」的。

那麼我如何察其流程？1/3 用猜的，1/3 靠字串搜尋工具 GREP（第 8 章介紹過），1/3 靠勤讀書。然後我發現，【New Window】命令流到 *CMDIFrameWnd::OnWindowNew* 去了。

圖 13-5 是其原始碼（MFC 4.0 的版本）。

```
#0001 void CMDIFrameWnd::OnWindowNew()
#0002 {
#0003     CMDIChildWnd* pActiveChild = MDIGetActive();
#0004     CDocument* pDocument;
#0005     if (pActiveChild == NULL ||
#0006         (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0007     {
#0008         TRACE0("Warning: No active document for WindowNew command.\n");
#0009         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0010         return; // command failed
#0011     }
#0012
#0013     // otherwise we have a new frame !
#0014     CDocTemplate* pTemplate = pDocument->GetDocTemplate();
#0015     ASSERT_VALID(pTemplate);
#0016     CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0017     if (pFrame == NULL)
#0018     {
#0019         TRACE0("Warning: failed to create new frame.\n");
```

```

#0020             return;    // command failed
#0021     }
#0022
#0023     pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0024 }

```

圖 13-5 `CMDIFrameWnd::OnWindowNew` 原始碼 (in `WINMDI.CPP`)

我們的焦點放在 `CMDIFrameWnd::OnWindowNew` 函式的第 14 行，該處取得我們在 `InitInstance` 函式中做好的 Document Template，而你知道，Document Template 中記錄有 View 類別。好，如果我們能夠另準備一個嶄新的 View 類別，有著不同的 `OnDraw` 顯示方式，並再準備好另一份 Document Template，記錄該新的 View 類別，然後改變圖 13-5 的第 14 行，讓它使用這新的 Document Template，大功成矣。

當然，我們絕不是要去改 MFC 原始碼，而是要改寫虛擬函式 `OnWindowNew`，使為我們所用。這很簡單，我只要把【Window / New Window】命令項改變名稱，例如改為【Window / New Hex Window】，然後為它撰寫命令處理函式，函式內容完全仿照圖 13-5，但把第 14 行改設定為新的 Document Template 即可。

Text 範例程式

Text 程式提供【Window / New Text Window】和【Window / New Hex Window】兩個新的選單命令項目，都可以產生出 view 視窗，一個以 ASCII 型式顯示資料，一個以 Hex 型式顯示資料，資料來自同一份 Document。

以下 Text 程式的是製作過程：

- 進入 AppWizard，製造一個 Text 專案，採用各種預設的選項。獲得的主要類別如下：

類別	基礎類別	檔案	
<i>CTextApp</i>	<i>CWinApp</i>	TEXT.CPP	TEXT.H
<i>CMainFrame</i>	<i>CMDIFrameWnd</i>	MAINFRM.CPP	MAINFRM.H
<i>CChildFrame</i>	<i>CMDIChildWnd</i>	CHILDFRM.CPP	CHILDFRM.H
<i>CTextDoc</i>	<i>CDocument</i>	TEXTDOC.CPP	TEXTDOC.H
<i>CTextView</i>	<i>CView</i>	TEXTVIEW.CPP	TEXTVIEW.H

- 進入整合環境中的 Resource View 視窗，選擇 *IDR_TEXTTYPE* 選單，在【Window】選單中加入兩個新命令項：

命令項目名稱	識別碼 (ID)	提示字串
New Text Window	<i>ID_WINDOW_TEXT</i>	New a Text Window with Active Document
New Hex Window	<i>ID_WINDOW_HEX</i>	New a Hex Window with Active Document

- 再在 Resource View 視窗中選擇 *IDR_MAINFRAME* 工具列，增加兩個按鈕，安排在 Help 按鈕之後：



這兩個按鈕分別對應於新添加的兩個選單命令項目。

- 進入 ClassWizard，為兩個 UI 物件製作 Message Map。這兩個命令訊息並不會影響 Document 內容（不像上一節的 GRAPH 例那樣），我們在 *CMainFrame* 中處理這兩個命令訊息頗為恰當。

UI 物件	訊息	訊息處理常式
<i>ID_WINDOW_TEXT</i>	<i>COMMAND</i>	<i>OnWindowText</i>
<i>ID_WINDOW_HEX</i>	<i>COMMAND</i>	<i>OnWindowHex</i>

- 利用 ClassWizard 產生一個新類別，準備做為同源子視窗的第二個 View 類別：

類別名稱	基礎類別	檔案
<i>CHexView</i>	<i>CView</i>	HEXVIEW.CPP HEXVIEW.H

- 修改程式碼，分別為兩個 view 類別都做出對應的 Document Template：

```
// in TEXT.H
class CTextApp : public CWinApp
{
public:
    CMultiDocTemplate* m_pTemplateTxt;
    CMultiDocTemplate* m_pTemplateHex;
    ...
public:
    virtual BOOL InitInstance();
    virtual int  ExitInstance();
    ...
};

// in TEXT.CPP
...
#include "TextView.h"
#include "HexView.h"
...
BOOL CTextApp::InitInstance()
{
    ...
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_TEXTTYPE,
        RUNTIME_CLASS(CTextDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CTextView));
    AddDocTemplate(pDocTemplate);

    m_pTemplateTxt = new CMultiDocTemplate(
        IDR_TEXTTYPE,
        RUNTIME_CLASS(CTextDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CTextView));

    m_pTemplateHex = new CMultiDocTemplate(
```



```

        IDR_TEXTTYPE,
        RUNTIME_CLASS(CTextDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CHexView));

    ...
}

int CTextApp::ExitInstance()
{
    delete m_pTemplateTxt;
    delete m_pTemplateHex;
    return CWinApp::ExitInstance();
}

```

- 修改 *CTextDoc* 程式碼，添加成員變數。Document 的資料是 10 筆字串：

```

// in TEXTDOC.H
class CTextDoc : public CDocument
{
public:
    CStringArray m_stringArray;
    ...
};

// in TEXTDOC.CPP
BOOL CTextDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_stringArray.SetSize(10);
    m_stringArray[0] = "If you love me let me know, ";
    m_stringArray[1] = "if you don't then let me go, ";
    m_stringArray[2] = "I can take another minute ";
    m_stringArray[3] = " of day without you in it. ";
    m_stringArray[4] = " ";
    m_stringArray[5] = "If you love me let it be, ";
    m_stringArray[6] = "if you don't then set me free";
    m_stringArray[7] = "... ";
    m_stringArray[8] = "SORRY, I FORGET IT! ";
    m_stringArray[9] = " J.J.Hou 1995.03.22 19:26";

    return TRUE;
}

```

- 修改 *CTextView::OnDraw* 函式碼，在其中取得 Document 物件指標，然後把文字顯現出來：

```
// in TEXTVIEW.CPP
void CTextView::OnDraw(CDC* pDC)
{
    CTextDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    int i, j, nHeight;
    TEXTMETRIC tm;

    pDC->GetTextMetrics(&tm);
    nHeight = tm.tmHeight + tm.tmExternalLeading;

    j = pDoc->m_stringArray.GetSize();
    for (i = 0; i < j; i++) {
        pDC->TextOut(10, i*nHeight, pDoc->m_stringArray[i]);
    }
}
```

- 修改 *CHexView* 程式碼，在 *OnDraw* 函式中取得 Document 物件指標，把 ASCII 轉換為 Hex 碼，再以文字顯示出來：

```
#0001 #include "stdafx.h"
#0002 #include "Text.h"
#0003 #include "TextDoc.h"
#0004 #include "HexView.h"
#0005 ...
#0006 void CHexView::OnDraw(CDC* pDC)
#0007 {
#0008     // CDocument* pDoc = GetDocument();
#0009     CTextDoc* pDoc = (CTextDoc*)GetDocument();
#0010
#0011     int i, j, k, l, nHeight;
#0012     long n;
#0013     char temp[10];
#0014     CString Line;
#0015     TEXTMETRIC tm;
#0016
#0017     pDC->GetTextMetrics(&tm);
#0018     nHeight = tm.tmHeight + tm.tmExternalLeading;
#0019
#0020     j = pDoc->m_stringArray.GetSize();
```

```

#0021         for(i = 0; i < j; i++) {
#0022             wsprintf(temp, "%02x    ", i);
#0023             Line = temp;
#0024             l = pDoc->m_stringArray[i].GetLength();
#0025             for(k = 0; k < l; k++) {
#0026                 n = pDoc->m_stringArray[i][k] & 0x00FF;
#0027                 wsprintf(temp, "%02lx ", n);
#0028                 Line += temp;
#0029             }
#0030             pDC->TextOut(10, i*nHeight, Line);
#0031         }
#0032     }

```

- 定義 *CMainFrame* 的兩個命令處理常式：*OnWindowText* 和 *OnWindowHex*，使選單命令項目和工具列按鈕得以發揮效用。函式內容直接拷貝自圖 13-5，只要修改其中第 14 行即可。這兩個函式是本節的技術重點。

```

#0001 void CMainFrame::OnWindowText()
#0002 {
#0003     CMDIChildWnd* pActiveChild = MDIGetActive();
#0004     CDocument* pDocument;
#0005     if (pActiveChild == NULL ||
#0006         (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0007     {
#0008         TRACE0("Warning: No active document for WindowNew command\n");
#0009         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0010         return;    // command failed
#0011     }
#0012
#0013     // otherwise we have a new frame!
#0014     CDocTemplate* pTemplate = ((CTextApp*) AfxGetApp())->m_pTemplateTxt;
#0015     ASSERT_VALID(pTemplate);
#0016     CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0017     if (pFrame == NULL)
#0018     {
#0019         TRACE0("Warning: failed to create new frame\n");
#0020         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0021         return;    // command failed
#0022     }
#0023
#0024     pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0025 }
#0026
#0027 void CMainFrame::OnWindowHex()

```

```

#0028 {
#0029     CMDIChildWnd* pActiveChild = MDIGetActive();
#0030     CDocument* pDocument;
#0031     if (pActiveChild == NULL ||
#0032         (pDocument = pActiveChild->GetActiveDocument()) == NULL)
#0033     {
#0034         TRACE0("Warning: No active document for WindowNew command\n");
#0035         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0036         return;    // command failed
#0037     }
#0038
#0039     // otherwise we have a new frame!
#0040     CDocTemplate* pTemplate = ((CTextApp*) AfxGetApp())->m_pTemplateHex;
#0041     ASSERT_VALID(pTemplate);
#0042     CFrameWnd* pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
#0043     if (pFrame == NULL)
#0044     {
#0045         TRACE0("Warning: failed to create new frame\n");
#0046         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
#0047         return;    // command failed
#0048     }
#0049
#0050     pTemplate->InitialUpdateFrame(pFrame, pDocument);
#0051 }

```

- 如果你要兩個 view 都有列印預視的能力，必須在 *CHexView* 中改寫下面三個虛擬函式，至於它們的內容，可以依樣畫葫蘆地從 *CTextView* 的同名函式中拷貝一份過來：

```

// in HEXVIEW.H
class CHexView : public CView
{
...
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CHexView)
protected:
    virtual void OnDraw(CDC* pDC);    // overridden to draw this view
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL
...
};

```

```
// in HEXVIEW.CPP
BEGIN_MESSAGE_MAP(CHexView, CView)
   //{{AFX_MSG_MAP(CHexView)
        // NOTE - the ClassWizard will add and remove mapping macros here.
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////
// CTextView printing

BOOL CHexView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CHexView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

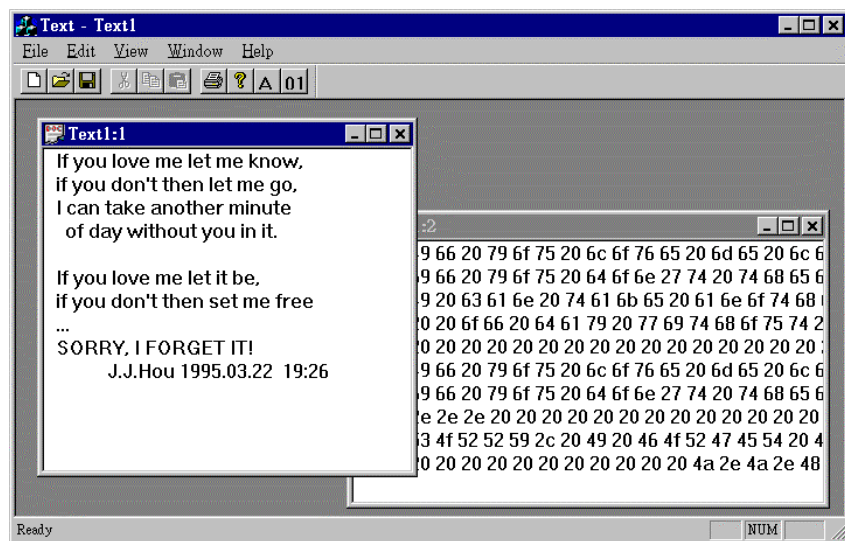
void CHexView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}
```

■ 本例並未示範 Serialization 動作。

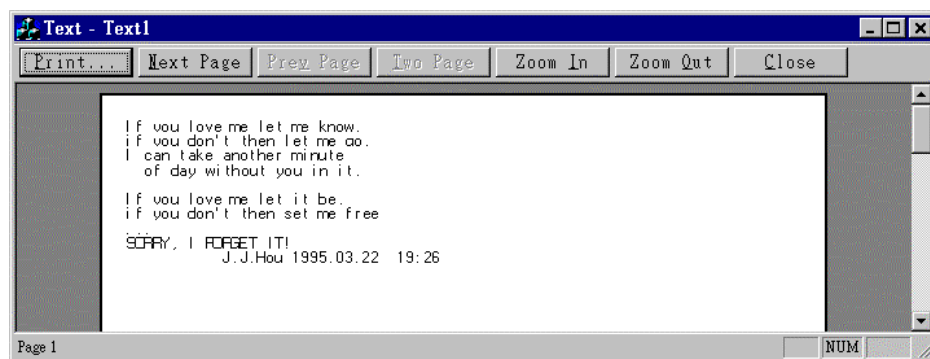
非制式作法的缺點

既然是走後門，就難保哪一天出問題。如果 MFC 的版本變動，*CMDIFrameWnd::OnWindowNew* 內容改了，你就得注意本節這個方法還能適用否。

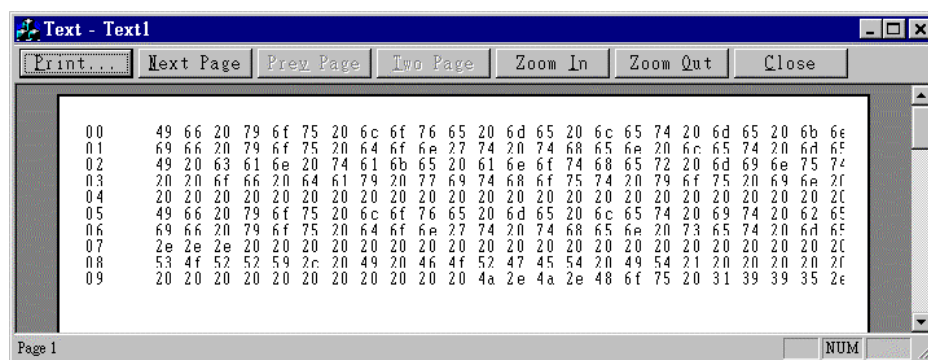
下面是 Text 程式的執行畫面。我先開啓一個 Text 視窗，再選按【Window/New Hex Window】或工具列上的對應按鈕，開啓另一個 Hex 視窗。兩個 View 視窗以不同的方式顯示同一份文件資料。



當你選按【File/Preview】命令項，哪一個視窗為 active 視窗，那個視窗的內容就出現在預視畫面中。以下是 Text 視窗的列印預視畫面：



以下是 Hex 視窗的列印預視畫面：



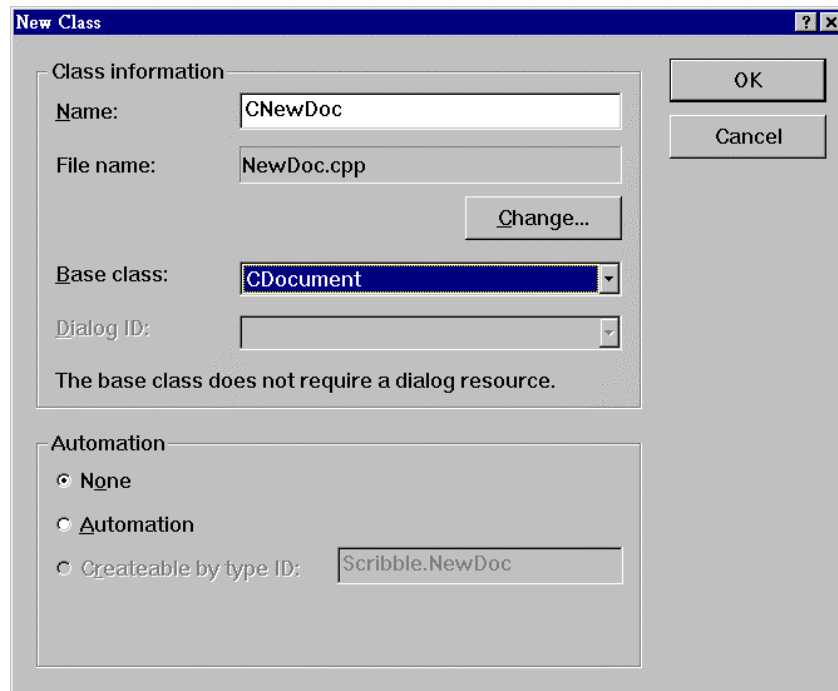
多重事件

截至目前，我所談的都是如何以不同的方式在不同的視窗中顯示同一份文件資料。如果我想寫那種「多功能」軟體，必須支援許多種文件型態，該怎麼辦？

就以前一節的 Graph 程式為基礎，繼續我們的探索。Graph 的文件型態原本是一個整數陣列，數量有 10 筆。我想在上面再多支援一種功能：文字編輯能力。

新的 Document 類別

首先，我應該利用 ClassWizard 新添一個 Document 類別，並以 *CDocument* 為基礎。啓動 ClassWizard，選擇【Member Variables】附頁，按下【Add Class...】鈕，出現對話盒，填寫如下：



下面是 ClassWizard 為我們做出來的碼：

```
#0001 // NewDoc.cpp : implementation file
#0002 //
#0003
#0004 #include "stdafx.h"
#0005 #include "Graph.h"
#0006 #include "NewDoc.h"
#0007
#0008 #ifdef _DEBUG
#0009 #define new DEBUG_NEW
#0010 #undef THIS_FILE
#0011 static char THIS_FILE[] = __FILE__;
#0012 #endif
#0013
#0014 //////////////////////////////////////
#0015 // CNewDoc
#0016
```



```

#0017 IMPLEMENT_DYNCREATE(CNewDoc, CDocument)
#0018
#0019 CNewDoc::CNewDoc()
#0020 {
#0021 }
#0022
#0023 BOOL CNewDoc::OnNewDocument()
#0024 {
#0025     if (!CDocument::OnNewDocument())
#0026         return FALSE;
#0027     return TRUE;
#0028 }
#0029
#0030 CNewDoc::~CNewDoc()
#0031 {
#0032 }
#0033
#0034
#0035 BEGIN_MESSAGE_MAP(CNewDoc, CDocument)
#0036     //{AFX_MSG_MAP(CNewDoc)
#0037     // NOTE - the ClassWizard will add and remove mapping macros here.
#0038     //}AFX_MSG_MAP
#0039 END_MESSAGE_MAP()
#0040
#0041 //////////////////////////////////////
#0042 // CNewDoc diagnostics
#0043
#0044 #ifdef _DEBUG
#0045 void CNewDoc::AssertValid() const
#0046 {
#0047     CDocument::AssertValid();
#0048 }
#0049
#0050 void CNewDoc::Dump(CDumpContext& dc) const
#0051 {
#0052     CDocument::Dump(dc);
#0053 }
#0054 #endif // _DEBUG
#0055
#0056 //////////////////////////////////////
#0057 // CNewDoc serialization
#0058
#0059 void CNewDoc::Serialize(CArchive& ar)
#0060 {
#0061     if (ar.IsStoring())
#0062     {

```

```

#0063          // TODO: add storing code here
#0064      }
#0065      else
#0066      {
#0067          // TODO: add loading code here
#0068      }
#0069
#0070      // CEditView contains an edit control which handles all serialization
#0071      ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
#0072  }
#0073
#0074  //////////////////////////////////////
#0075  // CNewDoc commands

```

註：陰影中的這兩行碼（#0070 和 #0071）不是 ClassWizard 產生的，是我自己加的，提前與你見面。稍後我會解釋為什麼加這兩行。

新的 Document Template

然後，我應該為此新的文件型態產生一個 Document Template，並把它加到系統所維護的 DocTemplate 串列中。注意，為了享受現成的文字編輯能力，我選擇 *CEditView* 做為與此 Document 搭配之 View 類別。還有，由於 *CChildFrame* 已經因為第一個文件型態 Graph 的三叉靜態分裂而被我們改寫了 *OnCreateClient* 函式，已不再適用於這第二個文件型態（NewDoc），所以我決定直接採用 *CMDIChildWnd* 做為 NewDoc 文件型態的 MDI Child Frame 視窗：

```

#include "stdafx.h"
#include "Graph.h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "GraphDoc.h"
#include "GraphView.h"
#include "NewDoc.h"
...
BOOL CGraphApp::InitInstance()
{
    ...
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_GRAPHTYPE,
        RUNTIME_CLASS(CGraphDoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame

```

```

        RUNTIME_CLASS(CGraphView));
AddDocTemplate(pDocTemplate);

    pDocTemplate = new CMultiDocTemplate(
        IDR_NEWTYPE,
        RUNTIME_CLASS(CNewDoc),
        RUNTIME_CLASS(CMDIChildWnd), // use directly
        RUNTIME_CLASS(CEditView));
    AddDocTemplate(pDocTemplate);
    ...
}

```

CMultiDocTemplate 的第一個參數 (resource ID) 也不能再延用 *Graph* 文件型態所使用的 *IDR_GRAPHTYPE* 了。要知道，這個 ID 值關係非常重大。我們得自行設計一套適用於 *NewDoc* 文件型態的 UI 系統出來（包括選單、工具列、檔案存取對話盒的內容、文件圖示、視窗標題...）。

怎麼做？第 7 章的深入討論將在此開花結果！請務必回頭複習複習「Document Template 的意義」一節，我將直接動作，不再多做說明。

新的 UI 系統

下面就是爲了這新的 *NewDoc* 文件型態所對應的 UI 系統，新添的檔案內容（沒有什麼好工具可以幫忙，一般文字編輯器的 copy/paste 最快）：

```

// in RESOURCE.H
#define IDD_ABOUTBOX            100
#define IDR_MAINFRAME           128
#define IDR_GRAPHTYPE           129
#define IDR_NEWTYPE              130
...

// in GRAPH.RC
IDR_NEWTYPE ICON DISCARDABLE "res\\NewDoc.ico" // 此 icon 需自行備妥

IDR_NEWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN

```

```

        MENUITEM "&New\tCtrl+N",          ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM "&Close",                ID_FILE_CLOSE
        MENUITEM "&Save\tCtrl+S",          ID_FILE_SAVE
        MENUITEM "Save &As...",            ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Print...\tCtrl+P",      ID_FILE_PRINT
        MENUITEM "Print Pre&view",          ID_FILE_PRINT_PREVIEW
        MENUITEM "P&rint Setup...",         ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File",             ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                   ID_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",          ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",            ID_EDIT_CUT
        MENUITEM "&Copy\tCtrl+C",           ID_EDIT_COPY
        MENUITEM "&Paste\tCtrl+V",           ID_EDIT_PASTE
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar",                ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar",             ID_VIEW_STATUS_BAR
    END
    POPUP "&Window"
    BEGIN
        MENUITEM "&New Window",             ID_WINDOW_NEW
        MENUITEM "&Cascade",                ID_WINDOW_CASCADE
        MENUITEM "&Tile",                    ID_WINDOW_TILE_HORZ
        MENUITEM "&Arrange Icons",           ID_WINDOW_ARRANGE
        MENUITEM "S&plit",                   ID_WINDOW_SPLIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About Graph...",          ID_APP_ABOUT
    END
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME "Graph"
    IDR_GRAPHTYPE "Graph\nGraph\nGraph\nGraph Files
                  (*.fig)\n.FIG\nGraph.Document\nGraph Document"

```

```
IDR_NEWTYPE "NewDoc\nNewDoc\nNewDoc\nNewDoc Files
(*.txt)\n.TXT\nNewDoc.Document\nNewDoc Document"
END
```

新文件的檔案讀寫動作

你大概還沒有忘記，第 7 章最後曾經介紹過，當我們在 AppWizard 中選擇 *CEditView*（而不是 *CView*）作為我們的 View 類別基礎時，AppWizard 會為我們在 *CMyDoc::Serialize* 函式中放入這樣的碼：

```
void CMyDoc::Serialize(CArchive& ar)
{
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead()->SerializeRaw(ar);
}
```

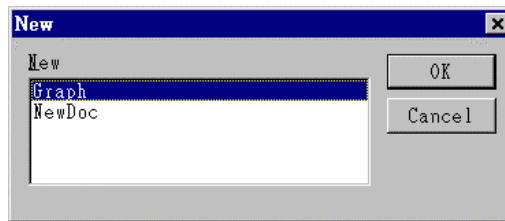
當你使用 *CEditView*，編輯器視窗所承載的文字是放在 Edit 控制元件自己的一個記憶體區塊中，而不是切割到 Document 中。所以，文件的檔案讀寫動作只要呼叫 *CEditView* 的 *SerializeRaw* 函式即可。

為了這 NewDoc 文件型態能夠讀寫檔，我們也依樣畫葫蘆地把上一段碼陰影部份加到 Graph 程式新的 Document 類別去：

```
void CNewDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }

    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead()->SerializeRaw(ar);
}
```

現在一切完備，重新編輯聯結並執行。一開始，由於 *InitInstance* 函式會自動為我們 New 一個新文件，而 Graph 程式不知道該 New 哪一種文件型態才好，所以會給我們這樣的對話盒：

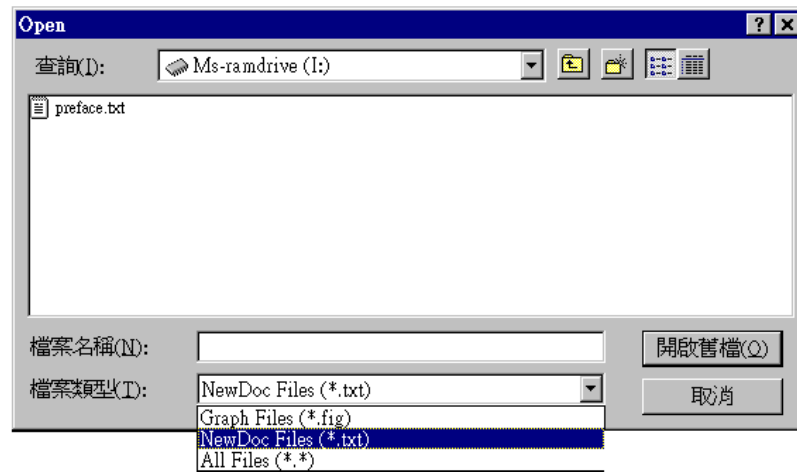


往後每一次選按【File/New】，都會出現上述對話盒。

以下是我們打開 Graph 文件和 NewDoc 文件各一份的畫面。注意，當 active 視窗是 NewDoc 文件，工具列上屬於 Graph 文件所用的最後三個按鈕是不起作用的：



以下是【Open】對話盒（用來開檔）。注意，檔案有 .fig 和 .txt 和 *.* 三種選擇：



這個新的 Graph 版本放在書附光碟片的 \GRAPH2.13 目錄中。

MFC 多緒程式設計

Multi-threaded Programming in MFC

緒 (thread)，是執行緒 (thread of execution) 的簡單稱呼。"Thread" 這個字的原意是「線」。中文字裡頭的「緒」也有「線」的意思，所以我採用「緒」、「執行緒」這樣的中文名稱。如果你曾經看過「多線」這個名詞，其實就是本章所謂的「多緒」。

我曾經在第 1 章以三兩個小節介紹 Win32 環境下的行程與執行緒觀念，並且以程式直接呼叫 *CreateThread* 的形式，示範了幾個 Win32 小例子。現在我要更進一步從作業系統的層面談談執行緒的學理基礎，然後帶引各位看看 MFC 對於「執行緒」支援了什麼樣的類別。然後，實際寫個 MFC 多緒程式。

從作業系統層面看執行緒

書籍推薦：如果要從作業系統層面來瞭解執行緒，Matt Pietrek 的 *Windows 95 System Programming SECRETS* (Windows 95 系統程式設計大奧秘/侯俊傑譯/旗標出版) 無疑是最佳知識來源。Matt 把作業系統核心模組 (KERNEL32.DLL) 中用來維護執行緒生存的資料結構都挖掘出來，非常詳盡。這是對執行緒的最基礎認識，直達其靈魂深處。

你已經知道，*CreateThread* 可以產生一個執行緒，而「緒」的本體就是 *CreateThread* 第 3 個參數所指定的一個函式（一般我們稱之為「執行緒函式」）。這個函式將與目前的「執行事實」同時並行，成為另一個「執行事實」。執行緒函式的執行期，也就是該執行緒的生命期。

作業系統如何造成這種多工並行的現象？執行緒對於作業系統的意義到底是什麼？系統如何維護許多個執行緒？執行緒與其父親大人（行程）的關係如何維持？CPU 只有一個，執行緒卻有好幾個，如何擺平優先權與排程問題？這些疑問都可以在下面各節中獲得答案。

三個觀念：模組、行程、執行緒

試著回答這個問題：行程（process）是什麼？給你一分鐘時間。

z z z Z Z...

你的回答可能是：『一個可執行檔執行起來，就是一個行程』。唔，也不能算錯。但能不能夠有更具體的答案？再問你一個問題：模組（module）是什麼？可能你的回答還是：『一個可執行檔執行起來，就是一個模組』。這也不能夠算錯。但是你明明知道，模組不等於行程。KERNEL32 DLL 是一個模組，但不是一個行程；Scribble EXE 是一個模組，也是一個行程。

我們需要更具體的資料，更精準的答案。

如果我們能夠知道作業系統如何看待模組和行程，就能夠給出具體的答案了。一段可執行的程式（包括 EXE 和 DLL），其程式碼、資料、資源被載入到記憶體中，由系統建置一個資料結構來管理它，就是一個模組。這裡所說的資料結構，名為 Module Database（MDB），其實就是 PE 格式中的 PE 表頭，你可以從 WINNT.H 檔中找到一個 *IMAGE_NT_HEADER* 結構，就是它。

好，解釋了模組，那麼行程是什麼？這就比較抽象一點了。這樣說，行程就是一大堆擁有權（ownership）的集合。行程擁有位址空間（由 memory context 決定）、動態配置而來的記憶體、檔案、執行緒、一系列的模組。作業系統使用一個所謂的 Process Database（PDB）資料結構，來記錄（管理）它所擁有的一切。

執行緒呢？執行緒是什麼？行程主要表達「擁有權」的觀念，執行緒則主要表達模組中的程式碼的「執行事實」。系統也是以一個特定的資料結構（Thread Database，TDB）記錄執行緒的所有相關資料，包括執行緒區域儲存空間（Thread Local Storage，TLS）、訊息佇列、handle 表格、位址空間（Memory Context）等等等。

最初，行程是以一個執行緒（稱為主執行緒，primary thread）做為開始。如果需要，行程可以產生更多的執行緒（利用 *CreateThread*），讓 CPU 在同一時間執行不同段落的碼。當然，我們都知道，在只有一顆 CPU 的情況下，不可能真正有多工的情況發生，「多個執行緒同時工作」的幻覺主要是靠排程器來完成 -- 它以一個硬體計時器和一組複雜的遊戲規則，在不同的執行緒之間做快速切換動作。以 Windows 95 和 Windows NT 而言，在非特殊的情況下，每個執行緒被 CPU 照顧的時間（所謂的 timeslice）是 20 個 milliseconds。

如果你有一部多 CPU 電腦，又使用一套支援多 CPU 的作業系統（如 Windows NT），那麼一個 CPU 就可以分配到一個執行緒，真正做到實實在在的多工。這種作業系統特性稱為 symmetric multiprocessing (SMP)。Windows 95 沒有 SMP 性質，所以即使得多 CPU 電腦上跑，也無法發揮其應有的高效能。

圖 14-1 表現出一個行程（PDB）如何透過「MODREF 串列」連接到其所使用的所有模組。圖 14-2 表現出一個模組資料結構（MDB）的細部內容，最後的 *DataDirectory*[16] 記錄著 16 個特定節區（sections）的位址，這些 sections 包括程式碼、資料、資源。圖 14-3 表現出一個執行緒資料結構（PDB）的細部內容。

當 Windows 載入器將程式載入記憶體中，KERNEL32 挖出一些記憶體，建構出一個 PDB、一個 TDB、一個以上的 MDBs（視此程式使用到多少 DLL 而定）。針對 TDB，作業系統又要產生出 memory context（就是在作業系統書籍中提到的那些所謂 page tables）、訊息佇列、handle 表格、環境資料結構（EDB）...。當這些系統內部資料結構都建構完畢，指令指位器（Instruction Pointer）移到程式的進入點，才開始程式的執行。

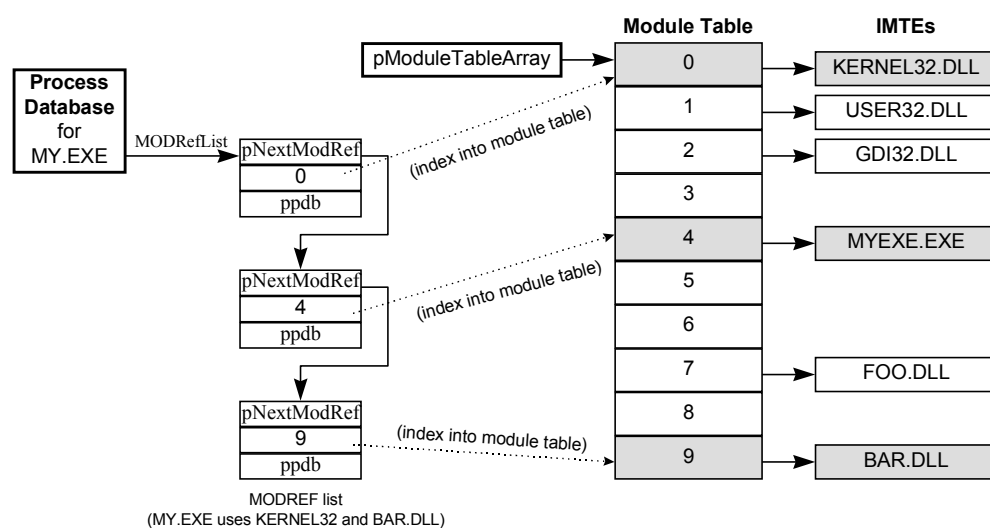


圖 14-1 行程（PDB）透過「MODREF 串列」連接到其所使用的所有模組

執行緒優先權（Priority）

我想我們現在已經能夠用很具體的形象去看所謂的行程、模組、執行緒了。「執行事實」發生在執行緒身上，而不在行程身上。也就是說，CPU 排程單位是執行緒而非行程。排程器據以排序的，是每個執行緒的優先權。

優先權的設定分為兩個階段。我已經在第 1 章介紹過。執行緒的「父親大人」（行程）

擁有所謂的優先權等級 (priority class, 圖 1-7), 可以在 *CreateProcess* 的參數中設定。執行緒基本上繼承自其「父親大人」的優先權等級, 然後再加上 *CreateThread* 參數中的微調差額 (-2~+2)。獲得的結果 (圖 1-8) 便是執行緒的所謂 base priority, 範圍從 0~31, 數值愈高優先權愈高。::*SetThreadPriority* 是調整優先權的工具, 它所指定的也是微調差額 (-2~+2)。

IMTE 結構

00h	DWORD	un1
04h	PIMAGE_NT_HEADERS	pNTHdr
08h	DWORD	un2
0Ch	PSTR	pszFileName
10h	PSTR	pszModName
14h	WORD	cbFileName
16h	WORD	cbModName
18h	DWORD	un3
1Ch	DWORD	cSections
20h	DWORD	un5
24h	DWORD	baseAddress/Module Handle
28h	WORD	hModule16
2Ah	WORD	cUsage
2Ch	DWORD	un7
30h	PSTR	pszFileName2
34h	WORD	cbFileName2
36h	DWORD	pszModName2
3Ah	WORD	cbModName2

Module Database (MDB)

```

IMAGE_NT_HEADERS :
  DWORD Signature
IMAGE_FILE_HEADER FileHeader :
  WORD Machine;
  WORD NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD SizeOfOptionalHeader;
  WORD Characteristics;
IMAGE_OPTIONAL_HEADER OptionalHeader :
  WORD Magic;
  BYTE MajorLinkerVersion;
  BYTE MinorLinkerVersion;
  DWORD SizeOfCode;
  DWORD SizeOfInitializedData;
  DWORD SizeOfUninitializedData;
  DWORD AddressOfEntryPoint;
  DWORD BaseOfCode;
  DWORD BaseOfData;
  DWORD ImageBase;
  DWORD SectionAlignment;
  DWORD FileAlignment;
  WORD MajorOperatingSystemVersion;
  WORD MinorOperatingSystemVersion;
  WORD MajorImageVersion;
  WORD MinorImageVersion;
  WORD MajorSubsystemVersion;
  WORD MinorSubsystemVersion;
  DWORD Reserved1;
  DWORD SizeOfImage;
  DWORD SizeOfHeaders;
  DWORD CheckSum;
  WORD Subsystem;
  WORD DllCharacteristics;
  DWORD SizeOfStackReserve;
  DWORD SizeOfStackCommit;
  DWORD SizeOfHeapReserve;
  DWORD SizeOfHeapCommit;
  DWORD LoaderFlags;
  DWORD NumberOfRvaAndSizes;
IMAGE_DATA_DIRECTORY DataDirectory[16]
  // 指向各個 sections,
  // 包括程式碼, 資料, 資源

```

圖 14-2 模組資料結構 MDB 的細部內容 (資料整理自 *Windows 95 System Programming SECRETS*, Matt Pietrek, IDG Books)

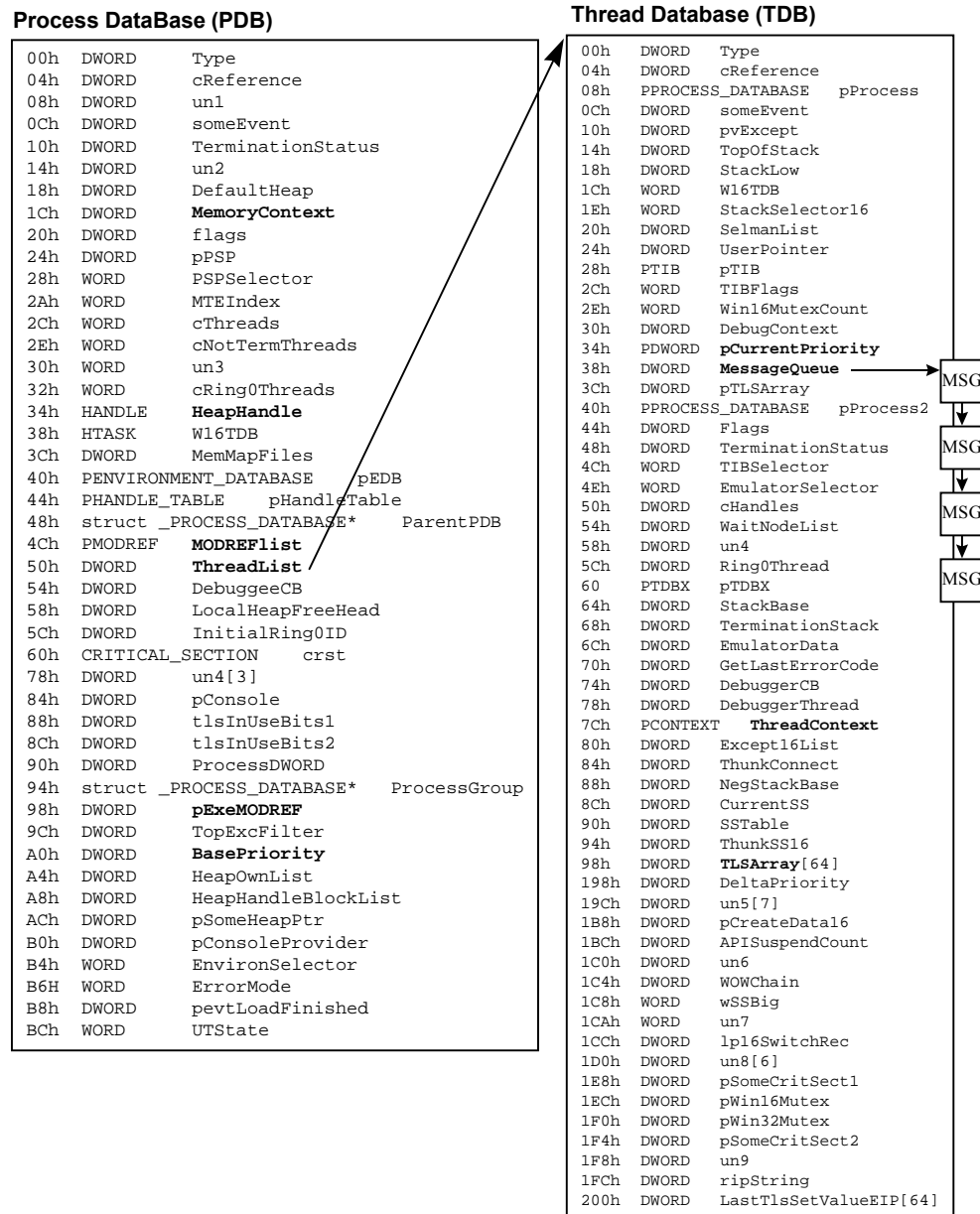


圖 14-3 執行緒資料結構 (PDB) 的細部內容 (資料整理自 Windows 95 System Programming SECRETS, Matt Pietrek, IDG Books)

執行緒排程 (Scheduling)

排程器挑選「下一個獲得 CPU 時間的執行緒」的唯一依據就是：執行緒優先權。如果所有等待被執行的執行緒中，有一個是優先權 16，其他所有執行緒都是優先權 15（或更低），那麼優先權 16 者便是下一個奪標者。如果執行緒 A 和 B 同為優先權 16，排程器會挑選等待比較久的那個（假設為執行緒 A）。當 A 的時間切片（timeslice）終了，如果 B 以外的其他執行緒的優先權仍維持在 15（以下），執行緒 B 就會獲得執行權。

「如果 B 以外的其他執行緒的優先權仍維持在 15（以下）...」，唔，這聽起來彷彿優先權會變動似的。的確是。為了避免朱門酒肉臭、路有凍死骨的不公平情況發生，排程器會彈性調整執行緒優先權，以強化系統的反應能力，並且避免任何一個執行緒一直未能接受 CPU 的潤澤。一般的執行緒優先權是 7，如果它被切換到前景，排程系統可能暫時地把它調昇到 8 或 9 或更高。對於那些有著輸入訊息等待被處理的執行緒，排程系統也會暫時調高其優先權。

對於那些優先權本來就高的執行緒，也並不是有永久的保障權利。別忘了 Windows 畢竟是個訊息驅動系統，如果某個執行緒呼叫 `::GetMessage` 而其訊息佇列卻是空的，這個執行緒便被凍結，直到再有訊息進來為止。凍結的意思就是不管你的優先權有多高，暫時退出排班行列。執行緒也可能被以 `::SuspendThread` 強制凍結住（`::ResumeThread` 可以解除凍結）。

會被凍結，表示這個執行緒「要去抓取訊息，而執行緒所附帶的訊息佇列中卻沒有訊息」。如果一個執行緒完全和 UI 無關呢？是否它就沒有訊息佇列？倒不是，但它的程式碼中沒有訊息迴路倒是事實。是的，這種執行緒稱為 **worker thread**。正因它不可能會被凍結，所以它絕對不受 `Win16Mutex` 或其他因素而影響其強制性多工性質，及其優先權。

Thread Context

Context 一詞，我不知道有沒有什麼好譯名，姑且就用原文吧。它的直接意思是「前後關係、脈絡；環境、背景」。所以我們可以說 **Thread Context** 是構成執行緒的「背景」。

那是指什麼呢？狹義來講是指一組暫存器值（包括指令指位器 IP）。因為執行緒常常會被暫停，被要求把 CPU 擁有權讓出來，所以它必須將暫停之前一刻的狀態統統記錄下來，以備將來還可以恢復。

你可以在 WINNT.H 中找到一個 *CONTEXT* 資料結構，它可以用來儲存 Thread Context。::GetThreadContext 和 ::SetThreadContext 可以取得和設定某個執行緒的 context，因而改變該執行緒的狀態。這已經是非常低階的行爲了。Matt Pietrek 在其 *Windows 95 System Programming SECRETS* 一書第 10 章，寫了一個 Win32 API Spy 程式，就充份運用了這兩個函式。

我想我們在作業系統層面上的執行緒學理基礎已經足夠了，現在讓我們看看比較實際一點的東西。

從程式設計層面看執行緒

書籍推薦：如果要從程式設計層面來瞭解執行緒，Jim Beveridge 和 Robert Wiener 合著的 *Multithreading Applications in Win32*（Win32 多緒程式設計/侯俊傑譯/碁峰出版）是很值得推薦的一份知識來源。這本書介紹執行緒的學理觀念、程式方法、同步控制、資料一致性的保持、C runtime library 的多緒版本、C++ 的多緒程式方法、MFC 中的多緒程式方法、除錯、行程通訊（IPC）、DLLs...，以及約 50 頁的實際應用。

書籍推薦：Jeffrey Richter 的 *Advanced Windows* 在行程與執行緒的介紹上（第 2 章和第 3 章），也有非常好的表現。他的切入方式是詳細而深入地敘述相關 Win32 API 的規格與用法。並舉實例佐證。

如何產生執行緒？我想各位都知道了，::CreateThread 可以辦到。圖 14-4 是與執行緒有關的 Win32 API。

與執行緒有關的 Win32 API	功能
AttachThreadInput	將某個執行緒的輸入導向另一個執行緒
CreateThread	產生一個執行緒
ExitThread	結束一個執行緒
GetCurrentThread	取得目前執行緒的 handle
GetCurrentThreadId	取得目前執行緒的 ID
GetExitCodeThread	取得某一執行緒的結束代碼（可用以決定執行緒是否已結束）
GetPriorityClass	取得某一行程的優先權等級
GetQueueStatus	傳回某一執行緒的訊息佇列狀態
GetThreadContext	取得某一執行緒的 context
GetThreadDesktop	取得某一執行緒的 desktop 物件
GetThreadPriority	取得某一執行緒的優先權
GetThreadSelectorEntry	除錯器專用，傳回指定之執行緒的某個 selector 的 LDT 記錄項
ResumeThread	將某個凍結的執行緒恢復執行
SetPriorityClass	設定優先權等級
SetThreadPriority	設定執行緒的優先權
Sleep	將某個執行緒暫時凍結。其他執行緒將獲得執行權。
SuspendThread	凍結某個執行緒
TerminateThread	結束某個執行緒
TlsAlloc	配置一個 TLS（Thread Local Storage）
TlsFree	釋放一個 TLS（Thread Local Storage）
TlsGetValue	取得某個 TLS（Thread Local Storage）的內容
TlsSetValue	設定某個 TLS（Thread Local Storage）的內容
WaitForInputIdle	等待，直到不再有輸入訊息進入某個執行緒中

圖 14-4 與執行緒有關的 Win32 API 函式

注意，多執行緒並不能讓程式執行得比較快（除非是在多 CPU 機器上，並且使用支援 symmetric multiprocessing 的作業系統），只是能夠讓程式比較「有反應」。試想某個程式在某個選單項目被按下後要做一個小時的運算工作，如果這份工作在主執行緒中做，而且沒有利用 *PeekMessage* 的技巧時時觀看訊息佇列的內容並處理之，那麼這一個小時內這個程式的使用者介面可以說是被凍結住了，將毫無反應。但如果沉重的運算工作是由另一個執行緒來負責，使用者介面將依然靈活，不受影響。

Worker Threads 和 UI Threads

從 Windows 作業系統的角度來看，執行緒就是執行緒，並未再有什麼分類。但從 MFC 的角度看，則把執行緒劃分為和使用著介面無關的 *worker threads*，以及和使用著介面 (UI) 有關的 *UI threads*。

基本上，當我們以 *::CreateThread* 產生一個執行緒，並指定一個執行緒函式，它就是一個 *worker thread*，除非在它的生命中接觸到了輸入訊息 -- 這時候它應該有一個訊息迴路，以抓取訊息，於是該執行緒搖身一變而為 *UI thread*。

注意，執行緒本來就帶有訊息佇列，請看圖 14-3 的 TDB 結構。而如果執行緒程式碼中帶有一個訊息迴路，就稱為 *UI thread*。

錯誤觀念

我記得曾經在微軟的技術文件中，也曾經在微軟的範例程式中，看到他們鼓勵這樣的作法：為程式中的每一個視窗產生一個執行緒，負責視窗行為。這種錯誤的示範尤其存在於 MDI 程式中。是的，早期我也沾沾自喜地為 MDI 程式的每一個子視窗設計一個執行緒。基本上這是錯誤的行為，要付出昂貴的代價。因為子視窗一切換，上述作法會導至執行緒也切換，而這卻要花費大量的系統資源。比較好的作法是把所有 UI (User Interface) 動作都集中在主執行緒中，其他的「純種運算工作」才考慮交給 *worker threads* 去做。

正確態度

什麼是使用多執行緒的好時機呢？如果你的程式有許多事要忙，但是你還要隨時保持注意某些外部事件（可能來自硬體或來自使用者），這時就適合使用多執行緒來幫忙。

以通訊程式為例。你可以讓主執行緒負責使用者介面，並保持中樞的地位。而以一個分離的執行緒處理通訊埠，

MFC 多緒程式設計

我已經在第 1 章以一個小節介紹了 Win32 多緒程式的寫法，並給了一個小範例 MltiThrd。這一節，我要介紹 MFC 多緒程式的寫法。

探索 CWinThread

就像 *CWinApp* 物件代表一個程式本身一樣，*CWinThread* 物件代表一個執行緒本身。這個 MFC 類別我們曾經看過，第 6 章講「MFC 程式的生死因果」時，講到「*CWinApp::Run* - 程式生命的活水源頭」，曾經追蹤過 *CWinApp::Run* 的源頭 *CWinThread::Run*（裡面有一個訊息迴路）。可見程式的「執行事實」係發生在 *CWinThread* 物件身上，而 *CWinThread* 物件必須要（必然會）產生一個執行緒。

我希望「*CWinThread* 物件必須要（必然會）產生一個執行緒」這句話不會引起你的誤會，以為程式在 application object（*CWinApp* 物件）的建構式必然有個動作最終呼叫到 *CreateThread* 或 *_beginthreadex*。不，不是這樣。想想看，當你的 Win32 程式執行起來，你的程式並沒有呼叫 *CreateProcess* 為自己做出代表自己的那個行程，也沒有呼叫 *CreateThread* 為自己做出代表自己的主執行緒（primary thread）的那個執行緒。為你的程式產生第一個行程和執行緒，是系統載入器以及核心模組（KERNEL32）合作的結果。

所以，再次循著第 6 章一步步剖析的步驟，MFC 程式的第一個動作是 *CWinApp::CWinApp*（比 *WinMain* 還早），在那裡沒有「產生執行緒」的動作，而是已經開始在收集執行緒

的相關資訊了：

```
// in MFC 4.2 APPCORE.CPP
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    ...
    // initialize CWinThread state
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;
    ASSERT(AfxGetThread() == NULL);
    pThreadState->m_pCurrentWinThread = this;
    ASSERT(AfxGetThread() == this);
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();
    ...
}
```

雖然 MFC 程式只會有一個 *CWinApp* 物件，而 *CWinApp* 衍生自 *CWinThread*，但並不是說一個 MFC 程式只能有一個 *CWinThread* 物件。每當你需要一個額外的執行緒，不應該在 MFC 程式中直接呼叫 *::CreateThread* 或 *_beginthreadex*，應該先產生一個 *CWinThread* 物件，再呼叫其成員函式 *CreateThread* 或全域函式 *AfxBeginThread* 將執行緒產生出來。當然，現在你必然已經可以推測到，*CWinThread::CreateThread* 或 *AfxBeginThread* 內部呼叫了 *::CreateThread* 或 *_beginthreadex*（事實上答案是 *_beginthreadex*）。

這看起來頗有值得商榷之處：為什麼 *CWinThread* 建構式不幫我們呼叫 *AfxBeginThread* 呢？似乎 *CWinThread* 為德不卒。

■ **14-5** 就是 *CWinThread* 的相關原始碼。

```
#0001 // in MFC 4.2 THRD CORE.CPP
#0002 CWinThread::CWinThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam)
#0003 {
#0004     m_pfnThreadProc = pfnThreadProc;
#0005     m_pThreadParams = pParam;
#0006
#0007     CommonConstruct();
#0008 }
#0009
#0010 CWinThread::CWinThread()
#0011 {
#0012     m_pThreadParams = NULL;
#0013     m_pfnThreadProc = NULL;
#0014
#0015     CommonConstruct();
#0016 }
#0017
#0018 void CWinThread::CommonConstruct()
#0019 {
#0020     m_pMainWnd = NULL;
#0021     m_pActiveWnd = NULL;
#0022
#0023     // no HTHREAD until it is created
#0024     m_hThread = NULL;
#0025     m_nThreadID = 0;
#0026
#0027     // initialize message pump
#0028 #ifdef _DEBUG
#0029     m_nDisablePumpCount = 0;
#0030 #endif
#0031     m_msgCur.message = WM_NULL;
#0032     m_nMsgLast = WM_NULL;
#0033     ::GetCursorPos(&m_ptCursorLast);
#0034
#0035     // most threads are deleted when not needed
#0036     m_bAutoDelete = TRUE;
#0037
#0038     // initialize OLE state
#0039     m_pMessageFilter = NULL;
#0040     m_lpfnOleTermOrFreeLib = NULL;
#0041 }
#0042
#0043 CWinThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam,
#0044     int nPriority, UINT nStackSize, DWORD dwCreateFlags,
#0045     LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0046 {
```

```
#0047     CWinThread* pThread = DEBUG_NEW CWinThread(pfnThreadProc, pParam);
#0048
#0049     if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED, nStackSize,
#0050         lpSecurityAttrs))
#0051     {
#0052         pThread->Delete();
#0053         return NULL;
#0054     }
#0055     VERIFY(pThread->SetThreadPriority(nPriority));
#0056     if (!(dwCreateFlags & CREATE_SUSPENDED))
#0057         VERIFY(pThread->ResumeThread() != (DWORD)-1);
#0058
#0059     return pThread;
#0060 }
#0061
#0062 CWinThread* AFXAPI AfxBeginThread(CRuntimeClass* pThreadClass,
#0063     int nPriority, UINT nStackSize, DWORD dwCreateFlags,
#0064     LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0065 {
#0066     ASSERT(pThreadClass != NULL);
#0067     ASSERT(pThreadClass->IsDerivedFrom(RUNTIME_CLASS(CWinThread)));
#0068
#0069     CWinThread* pThread = (CWinThread*)pThreadClass->CreateObject();
#0070
#0071     pThread->m_pThreadParams = NULL;
#0072     if (!pThread->CreateThread(dwCreateFlags|CREATE_SUSPENDED, nStackSize,
#0073         lpSecurityAttrs))
#0074     {
#0075         pThread->Delete();
#0076         return NULL;
#0077     }
#0078     VERIFY(pThread->SetThreadPriority(nPriority));
#0079     if (!(dwCreateFlags & CREATE_SUSPENDED))
#0080         VERIFY(pThread->ResumeThread() != (DWORD)-1);
#0081
#0082     return pThread;
#0083 }
#0084
#0085 BOOL CWinThread::CreateThread(DWORD dwCreateFlags, UINT nStackSize,
#0086     LPSECURITY_ATTRIBUTES lpSecurityAttrs)
#0087 {
#0088     // setup startup structure for thread initialization
#0089     _AFX_THREAD_STARTUP startup; memset(&startup, 0, sizeof(startup));
#0090     startup.pThreadState = AfxGetThreadState();
#0091     startup.pThread = this;
#0092     startup.hEvent = ::CreateEvent(NULL, TRUE, FALSE, NULL);
```

```

#0093     startup.hEvent2 = ::CreateEvent(NULL, TRUE, FALSE, NULL);
#0094     startup.dwCreateFlags = dwCreateFlags;
#0095     ...
#0096     // create the thread (it may or may not start to run)
#0097     m_hThread = (HANDLE)_beginthreadex(lpSecurityAttrs, nStackSize,
#0098         &_AfxThreadEntry, &startup, dwCreateFlags | CREATE_SUSPENDED, (UINT*)&m_nThreadID);
#0099     ...
#0100 }

```

圖 14-5 CWinThread 的相關原始碼

產生執行緒，為什麼不直接用 `::CreateThread` 或 `_beginthreadex`？為什麼要透過 `CWinThread` 物件？我想你可以輕易從 MFC 原始碼中看出，因為 `CWinThread::CreateThread` 和 `AfxBeginThread` 不只是 `::CreateThread` 的一層包裝，更做了一些 application framework 所需的內部資料初始化工作，並確保使用正確的 C runtime library 版本。原始碼中有：

```

#ifdef _MT
    ... // 做些設定工作，不產生執行緒，回返。
#else
    ... // 真正產生執行緒，回返。
#endif // !_MT

```

的動作，只是被我刪去未列出而已。

接下來我要把 worker thread 和 UI thread 的產生步驟做個整理。它們都需要呼叫 `AfxBeginThread` 以產生一個 `CWinThread` 物件，但如果要產生一個 UI thread，你還必須先定義一個 `CWinThread` 衍生類別。

產生一個 Worker Thread

Worker thread 不牽扯使用者介面。你應該為它準備一個執行緒函式，然後呼叫 `AfxBeginThread`：

```

CWinThread* pThread = AfxBeginThread(ThreadFunc, &Param);
...
UINT ThreadFunc (LPVOID pParam)
{
    ...
}

```

AfxBeginThread 事實上一共可以接受六個參數，分別是：

```

CWinThread* AFXAPI AfxBeginThread(AFX_THREADPROC pfnThreadProc,
                                   LPVOID pParam,
                                   int nPriority = THREAD_PRIORITY_NORMAL,
                                   UINT nStackSize = 0,
                                   DWORD dwCreateFlags = 0,
                                   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

```

參數一 *pfnThreadProc* 表示執行緒函式。參數二 *pParam* 表示要傳給執行緒函式的參數。參數三 *nPriority* 表示優先權的微調值，預設為 *THREAD_PRIORITY_NORMAL*，也就是沒有微調。參數四 *nStackSize* 表示堆疊的大小，預設值 0 則表示堆疊最大容量為 1MB。參數五 *dwCreateFlags* 如果為預設值 0，就表示執行緒產生後立刻開始執行；如果其值為 *CREATE_SUSPENDED*，就表示執行緒產生後先暫停執行。之後你可以使用 *CWinThread::ResumeThread* 重新執行它。參數六 *lpSecurityAttrs* 代表新執行緒的安全防護屬性。預設值 *NULL* 表示此一屬性與其產生者（也是個執行緒）的屬性相同。

在這裡我們遭遇到一個困擾。執行緒函式是由系統呼叫的，也就是個 *callback* 函式，不容許有 *this* 指標參數。所以任何一般的 C++ 類別成員函式都不能夠拿來當做執行緒函式。它必須是個全域函式，或是個 C++ 類別的 *static* 成員函式。其原因我已經在第 6 章的「*Callback* 函式」一節中描述過了，而採用全域函式或是 C++ *static* 成員函式，其間的優劣因素我也已經在該節討論過。

執行緒函式的型態 *AFX_THREADPROC* 定義於 *AFXWIN.H* 之中：

```

// in AFXWIN.H
typedef UINT (AFX_CDECL *AFX_THREADPROC) (LPVOID);

```

所以你應該把本身的執行緒函式宣告如下（其中的 *pParam* 是個指標，在實用上可以指向程式員自定的資料結構）：

```

UINT ThreadFunc (LPVOID pParam);

```

否則，編譯時會獲得這樣的錯誤訊息：

```
error C2665: 'AfxBeginThread' : none of the 2 overloads can convert
parameter 1 from type 'void (unsigned long *)'
```

有時候我們會讓不同的執行緒使用相同的執行緒函式，這時候你就得特別注意到執行緒函式使用全域變數或靜態變數時，資料共用所引發的嚴重性（有好有壞）。至於放置在堆疊中的變數或物件，都不會有問題，因為每一個執行緒自有一個堆疊。

寫一個 UI Thread

UI thread 可不能夠光由一個執行緒函式來代表，因為它要處理訊息，它需要一個訊息迴路。好得很，*CWinThread::Run* 裡頭就有一個訊息迴路。所以，我們應該先從 *CWinThread* 衍生一個自己的類別，再呼叫 *AfxBeginThread* 產生一個 *CWinThread* 物件：

```
class CMyThread : public CWinThread
{
    DECLARE_DYNCREATE(CMyThread)

public:
    void BOOL InitInstance();
};

IMPLEMENT_DYNCREATE(CMyThread, CWinThread)

BOOL CMyThread::InitInstance()
{
    ...
}

CWinThread *pThread = AfxBeginThread(RUNTIME_CLASS(CMyThread));
```

我想你對 *RUNTIME_CLASS* 巨集已經不陌生了，第 3 章和第 8 章都有這個巨集的原始碼展現以及意義解釋。*AfxBeginThread* 是上一小節同名函式的一個 overloaded 函式，一共可以接受五個參數，分別是：


```
CWinThread* AFXAPI AfxBeginThread(CRuntimeClass* pThreadClass,
                                   int nPriority = THREAD_PRIORITY_NORMAL,
                                   UINT nStackSize = 0,
                                   DWORD dwCreateFlags = 0,
                                   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

最後四個參數的意義和預設值比上一節同名函式相同，但是少接受一個 LPVOID pParam 參數。

你可以在 AFXWIN.H 中找到 *CWinThread* 的定義：

```
class CWinThread : public CCmdTarget
{
    DECLARE_DYNAMIC(CWinThread)

    BOOL CreateThread(DWORD dwCreateFlags = 0, UINT nStackSize = 0,
                     LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
    ...
    int GetThreadPriority();
    BOOL SetThreadPriority(int nPriority);
    DWORD SuspendThread();
    DWORD ResumeThread();
    BOOL PostThreadMessage(UINT message, WPARAM wParam, LPARAM lParam);
    ...
};
```

其中有許多成員函式和圖 14-4 中的 Win32 API 函式有關。在 *CWinThread* 的成員函式中，有五個函式只是非常單純的 Win32 API 的包裝而已，它們被定義於 AFXWIN2.INL 檔案中：

```
// in AFXWIN2.INL
// CWinThread
_AFXWIN_INLINE BOOL CWinThread::SetThreadPriority(int nPriority)
{ ASSERT(m_hThread != NULL); return ::SetThreadPriority(m_hThread, nPriority); }
_AFXWIN_INLINE int CWinThread::GetThreadPriority()
{ ASSERT(m_hThread != NULL); return ::GetThreadPriority(m_hThread); }
_AFXWIN_INLINE DWORD CWinThread::ResumeThread()
{ ASSERT(m_hThread != NULL); return ::ResumeThread(m_hThread); }
_AFXWIN_INLINE DWORD CWinThread::SuspendThread()
{ ASSERT(m_hThread != NULL); return ::SuspendThread(m_hThread); }
_AFXWIN_INLINE BOOL CWinThread::PostThreadMessage(UINT message, WPARAM wParam, LPARAM lParam)
{ ASSERT(m_hThread != NULL); return ::PostThreadMessage(m_nThreadID, message, wParam, lParam); }
```

執行緒的結束

既然 worker thread 的生命就是執行緒函式本身，函式一旦 return，執行緒也就結束了，自然得很。或者執行緒函式也可以呼叫 *AfxEndThread*，結束一個執行緒。

UI 執行緒因為有訊息迴路的關係，必須在訊息佇列中放一個 *WM_QUIT*，才能結束執行緒。放置的方式和一般 Win32 程式一樣，呼叫 *::PostQuitMessage* 即可辦到。亦或者，在執行緒的任何一個函式中呼叫 *AfxEndThread*，也可以結束執行緒。

AfxEndThread 其實也是個外包裝，其內部呼叫 *_endthreadex*，這個動作才真正把執行緒結束掉。

別忘了，不論 worker thread 或 UI thread，都需要一個 *CWinThread* 物件，當執行緒結束，記得把該物件釋放掉（利用 *delete*）。

執行緒與同步控制

看起來執行緒的誕生與結束，以及對它的優先權設定、凍結、重新啟動，都很容易。但是我必須警告你，多緒程式的設計成功關鍵並不在此。如果你的每一個執行緒都非常獨立，彼此沒有干聯，也就罷了。但如果許多個執行緒互有關聯呢？有經驗的人說多緒程式設計有多複雜多困難，他們說的並不是執行緒本身，而是指執行緒與執行緒之間的同步控制。

原因在於，沒有人能夠預期執行緒的被執行。在一個合作型多工系統中（例如 Windows 3.x），作業系統必須得到程式的允許才能夠改變執行緒。但是在強制性多工系統中（如 Win95 或 WinNT），控制權被排程器強制移轉，也因此兩個執行緒之間的執行次序變得不可預期。這不可預期性造成了所謂的 *race conditions*。

假設你正在一個檔案伺服器中編輯一串電話號碼。檔案打開來內容如下：

Charley	572-7993
Graffie	573-3976
Dennis	571-4219

現在你打算為 Sue 加上一筆新資料。正當你輸入 Sue 電話號碼的時候，另一個人也打開檔案並輸入另一筆有關於 Jason 的資料。最後你們兩人也都做了存檔動作。誰的資料會留下來？答案是比較晚存檔的那個人，而前一個人的輸入會被覆蓋掉。這兩個人面臨的就是 race condition。

再舉一個例子。你的程式產生兩個執行緒，A 和 B。執行緒 B 的任務是設定全域變數 X。執行緒 A 則要去讀取 X。假設執行緒 B 先完成其工作，設定了 X，然後執行緒 A 才執行，讀取 X，這是一種好的情況，如圖 14-6a。但如果執行緒 A 先執行起來並讀取全域變數 X，它會讀到一個不適當的值，因為執行緒 B 還沒有完成其工作並設定適當的 X。如圖 14-6b。這也是 race condition。

另一種執行緒所造成的可能問題是：死結（deadlock）。圖 14-7 可以說明這種情況。

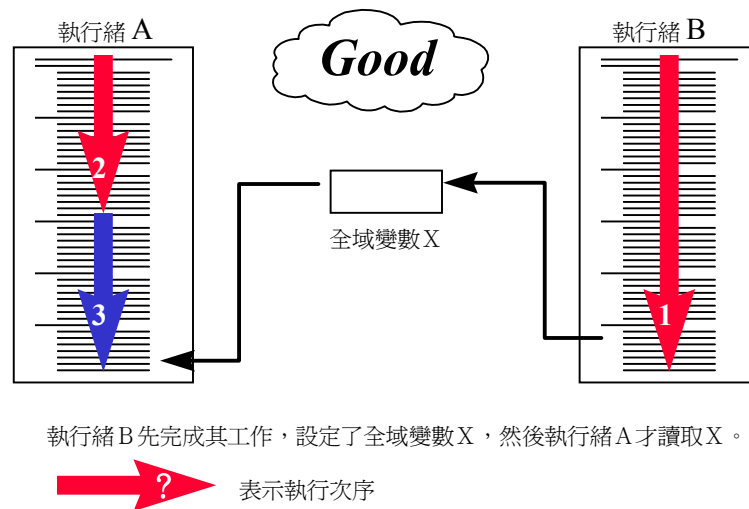
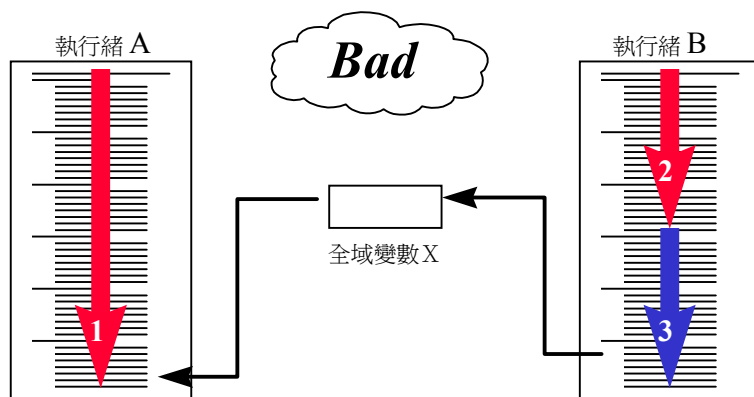


圖 14-6a race condition (good)



執行緒 A 先執行並讀取全域變數 X，然後執行緒 B 才設定 X。太遲了。

 表示執行次序

圖 14-6b race condition (bad)

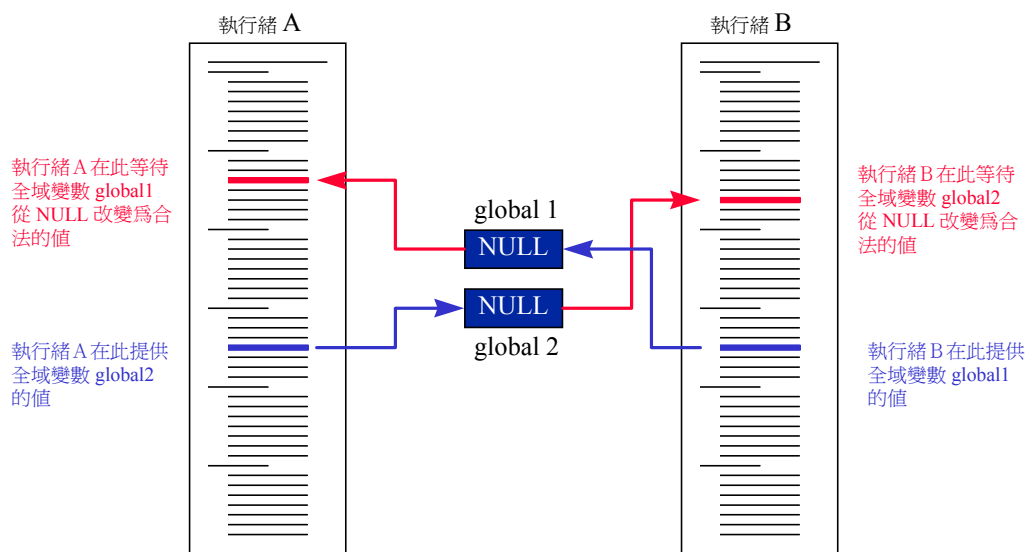
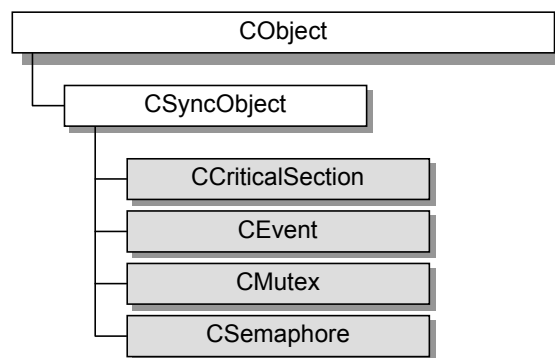


圖 14-7 死結 (deadlock)

要解決這些問題，必須有辦法協調各個執行緒的執行次序，讓某個緒等待某個緒。
Windows 系統提供四種同步化機制，幫助程式進行這種工作：

1. Critical Section（關鍵區域）
2. Semaphore（號誌）
3. Event（事件）
4. Mutex（Mutual Exclusive，互斥器）

MFC 也提供了四個對應的類別：



一想到本書的厚度，我就打消介紹同步機制的念頭了。你可以在許多 Visual C++ 或 MFC 程式設計書籍中得到這個主題的知識。

MFC 多緒程式實例

我將在此示範如何把第 1 章最後的一個 Win32 多緒程式 `MltiThrd` 改裝為 MFC 程式。我只示範主架構（與 `CWinThread`、`AfxBeginThread`、`ThreadFunc` 有關的部份），程式繪圖部份留給您做練習。

首先我利用 MFC AppWizard 產生一個 Mltithrd 專案，放在書附碟片的 Mltithrd.14 子目錄中，並接受 MFC AppWizard 的所有預設選項。

接下來我在 resource.h 中加上一些定義，做為執行緒函式的參數，以便在繪圖時能夠把代表各執行緒的各個長方形塗上不同的顏色：

```
#define HIGHEST_THREAD      0x00
#define ABOVE_AVE_THREAD   0x3F
#define NORMAL_THREAD      0x7F
#define BELOW_AVE_THREAD   0xBF
#define LOWEST_THREAD      0xFF
```

然後我在 Mltithrd.cpp 中加上一些全域變數（你也可以把它們放在 CMLtithrdApp 之中。我只是為了圖個方便）：

```
CMLtithrdApp theApp;
CWinThread* _pThread[5];
DWORD _ThreadArg[5] = { HIGHEST_THREAD,    // 0x00
                        ABOVE_AVE_THREAD,   // 0x3F
                        NORMAL_THREAD,      // 0x7F
                        BELOW_AVE_THREAD,   // 0xBF
                        LOWEST_THREAD       // 0xFF
                      }; // 用來調整四方形顏色
```

然後在 CMLtithrdApp::InitInstance 函式最後面加上一些碼：

```
// create 5 threads and suspend them
int i;
for (i= 0; i< 5; i++)
{
    _pThread[i] = AfxBeginThread(CMLtithrdView::ThreadFunc,
                                &_ThreadArg[i],
                                THREAD_PRIORITY_NORMAL,
                                0,
                                CREATE_SUSPENDED,
                                NULL);
}
```

```
// setup relative priority of threads
_pThread[0]->SetThreadPriority(THREAD_PRIORITY_HIGHEST);
_pThread[1]->SetThreadPriority(THREAD_PRIORITY_ABOVE_NORMAL);
_pThread[2]->SetThreadPriority(THREAD_PRIORITY_NORMAL);
_pThread[3]->SetThreadPriority(THREAD_PRIORITY_BELOW_NORMAL);
_pThread[4]->SetThreadPriority(THREAD_PRIORITY_LOWEST);
```

這樣一來我就完成了五個 worker threads 的產生，並且將其優先權做了 -2~+2 範圍之間的微調。

接下來我應該設計執行緒函式。就如我在第 1 章已經說過，這個函式的五個執行緒可以使用同一個執行緒函式。本例中是設計為全域函式好呢？還是 static 成員函式好？如果是後者，應該成為哪一個類別的成員函式好？

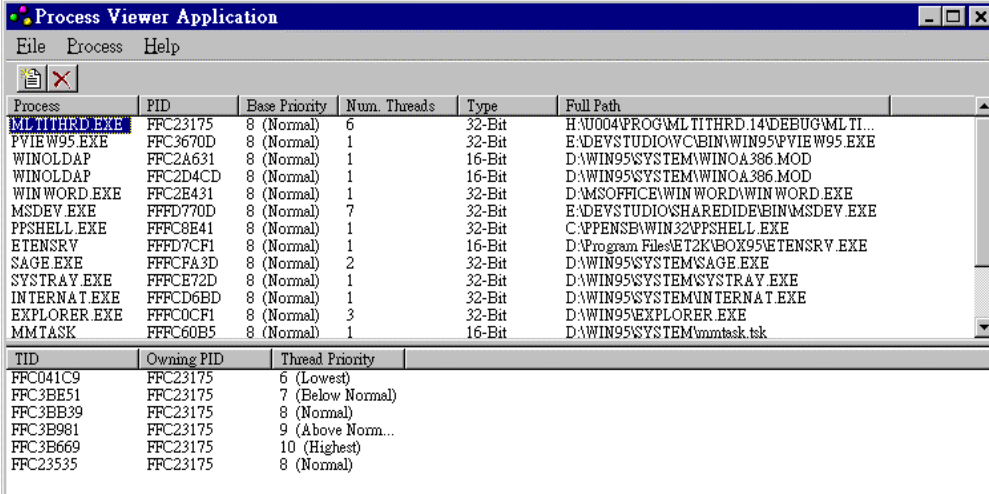
為了「要在執行緒函式做視窗繪圖動作」的考量，我把執行緒函式設計為 *CMLtithrdView* 的一個 static 成員函式，並遵循應有的函式型態：

```
// in MltithrdView.h
class CMLtithrdView : public CView
{
...
public:
    CMLtithrdDoc* GetDocument();
    static UINT ThreadFunc(LPVOID);
...
};

// in MltithrdView.cpp
UINT CMLtithrdView::ThreadFunc(LPVOID ThreadArg)
{
    DWORD dwArg = *(DWORD*)ThreadArg;

    // ... 在這裡做如同第 1 章的 MltitThrd 一樣的繪圖動作
    return 0;
}
```

好，到此為止，編譯聯結，獲得的程式將在執行後產生五個執行緒，並全部凍結。以 Process Viewer (Visual C++ 5.0 所附工具) 觀察之，證明它的確有六個執行緒（包括一個主執行緒以及我們所產生的另五個執行緒）：



Process	PID	Base Priority	Num. Threads	Type	Full Path
MLTITHRD.EXE	FFC23175	8 (Normal)	6	32-Bit	H:\004\PROG\MLTITHRD.14\DEBUG\MLTITHRD.EXE
PVIEW95.EXE	FFC3670D	8 (Normal)	1	32-Bit	E:\DEVSTUDIO\WC\BIN\WIN95\PVIEW95.EXE
WINOLDAP	FFC2A631	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINOA386.MOD
WINOLDAP	FFC2D4CD	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINOA386.MOD
WINWORD.EXE	FFC2E431	8 (Normal)	1	32-Bit	D:\MSOFFICE\WINWORD\WINWORD.EXE
MSDEV.EXE	FFFD770D	8 (Normal)	7	32-Bit	E:\DEVSTUDIO\SHARED\IDE\BIN\MSDEV.EXE
PPSHELL.EXE	FFFC8E41	8 (Normal)	1	32-Bit	C:\PPENSB\WIN32\PPSHELL.EXE
ETENSRV	FFFD7CF1	8 (Normal)	1	16-Bit	D:\Program Files\ET2K\BOX95\ETENSRV.EXE
SAGE.EXE	FFFCFA3D	8 (Normal)	2	32-Bit	D:\WIN95\SYSTEM\SAGE.EXE
SYSTRAY.EXE	FFFCF72D	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\SYSTRAY.EXE
INTERNAT.EXE	FFCD6BD	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\INTERNAT.EXE
EXPLORER.EXE	FFFC0CF1	8 (Normal)	3	32-Bit	D:\WIN95\EXPLORER.EXE
MMTASK	FFFC60B5	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\mmtask.tsk

TID	Owning PID	Thread Priority
FFC041C9	FFC23175	6 (Lowest)
FFC3BE51	FFC23175	7 (Below Normal)
FFC3BB39	FFC23175	8 (Normal)
FFC3B981	FFC23175	9 (Above Norm...)
FFC3B669	FFC23175	10 (Highest)
FFC23535	FFC23175	8 (Normal)

接下來，留給你的作業是：

1. 利用資源編輯器為程式加上各選單項目，如圖 1-9。
2. 設計上述選單項目的命令處理常式。
3. 在執行緒函式 *ThreadFunc* 內加上計算與繪圖能力。並判斷使用者選擇何種延遲方式，做出適當反應。

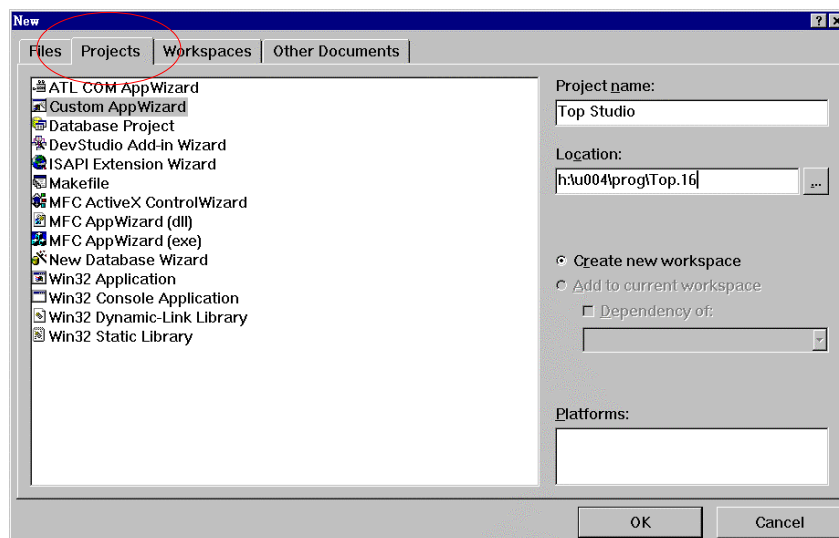
第 15 頁

定製一個 AppWizard

我們的 Scribble 程式一路走來，大家可還記得它一開始並不是平地而起，而是由 AppWizard 以「程式碼產生器」的身份，自動為我們做出一個我所謂的「骨幹程式」來？

Developer's Studio 提供了一個開放的 AppWizard 介面。現在，我們可以輕易地擴充 AppWizard：從小規模的擴充，到幾乎改頭換面成為一種全新型態的程式碼產生器。

Developer's Studio 提供了許多種不同的專案型態，供你選擇。當你選按 Visual C++ 5.0 整合環境中的【File/New】命令項，並選擇【Projects】附頁，便得到這樣的對話窗畫面：



除了上述這些內建的程式型態，它還可以顯示出任何自定程式型態（custom types）。Developer's Studio（整合環境）和 AppWizard 之間的介面藉著一組類別和一些元件表現出來，使我們能夠輕易訂製合乎自己需求的 AppWizard。製造出來的所謂 custom AppWizard（一個副檔名為 .AWX 的動態連結函式庫，註），必須被放置於磁碟目錄 \DevStudio\SharedIDE\Template 中，才能發揮效用。Developers Studio 和 AppWizard 和 AWX 之間的基本架構如圖 15-1。

註：我以 DUMPBIN（Visual C++ 附的一個觀察檔案型態的工具）觀察 .AWX 檔，得到結果如下：

```
E:\DevStudio\SharedIDE\BIN\IDE>dumpbin addinwz.awx
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
```

```
Dump of file addinwz.awx
```

```
File Type: DLL <--- 這證明 .AWX 的確是個動態連結函式庫。
```

```
Summary
  1000 .data
  1000 .reloc
 3A000 .rsrc
  3000 .text
```

事實上 AWX（Application Wizard extension）就是一個 32 位元的 MFC extension DLL。

是不是 Visual C++ 系統中早已存在有一些 .AWX 檔了呢？當然，它們是：

```
Directory of E:\DevStudio\SharedIDE\BIN\IDE
```

```
ADDINWZ  AWX      255,872  03-29-97  16:43  ADDINWZ.AWX
ATLWIZ   AWX      113,456  03-29-97  16:43  ATLWIZ.AWX
CUSTMWZ  AWX      278,528  03-29-97  16:43  CUSTMWZ.AWX
INETAWZ  AWX       91,408  03-29-97  16:43  INETAWZ.AWX
MFCTLWZ  AWX      146,272  03-29-97  16:43  MFCTLWZ.AWX
          5 file(s)      885,536 bytes
```

請放心，你只能夠擴充（新增）專案型態，不會一不小心取代了某一個原已存在的專案型態。

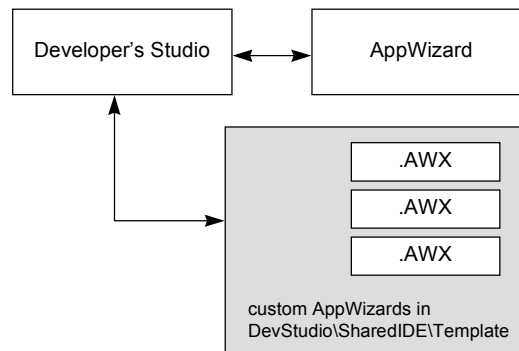


圖 15-1 Developers Studio 和 AppWizard 和 *.AWX 之間的基本架構。

到底 Wizard 是什麼？

所謂 Wizard，就是一個副檔名為 .AWX 的動態連結函式庫。Visual C++ 的 "project manager" 會檢查整合環境中的 Template 子目錄（\DevStudio\SharedIDE\Template），然後顯示其圖示於【New Project】對話窗中，供使用者選擇。

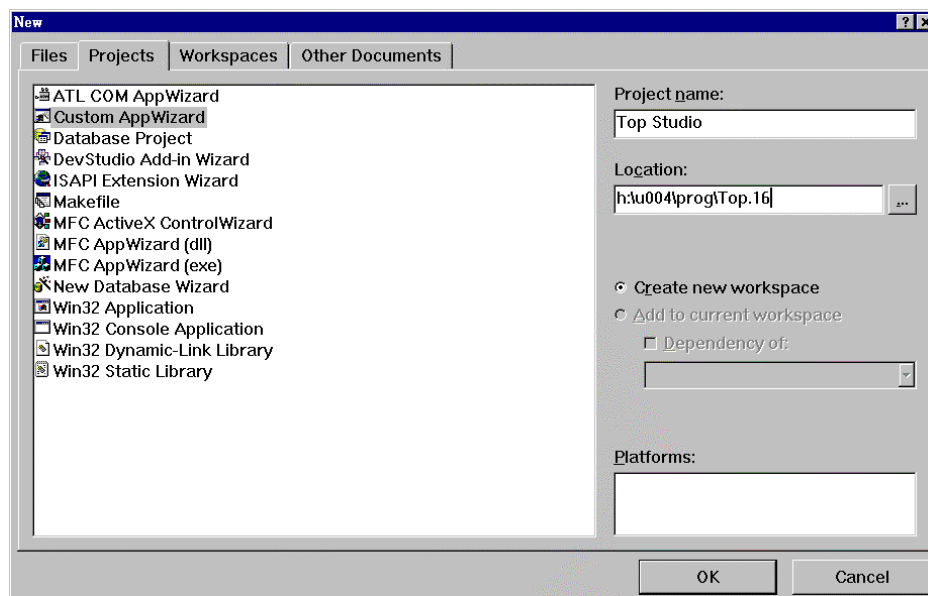
Wizard 本身是一個所謂的「template 直譯器」。這裡所謂的 "template" 是一些文字檔，內有許多特殊符號（也就是本章稍後要介紹的 macros 和 directives）。Wizard 讀取這些 template，對於正常文字，就以正常的 output stream 輸出到另一個檔案中；對於特殊符號或保留字，就解析它們然後再把結果以一般的 output stream 輸出到檔案中。Wizard 所顯示給使用者看的「步驟對話窗」可以接受使用者的指定項目或文字輸出，於是會影響 template 中的特殊符號的內容或解析，連帶也就影響了 Wizard 的 stream 輸出。這些 stream 輸出，最後就成為你的專案的原始檔案。

Custom AppWizard 的基本操作

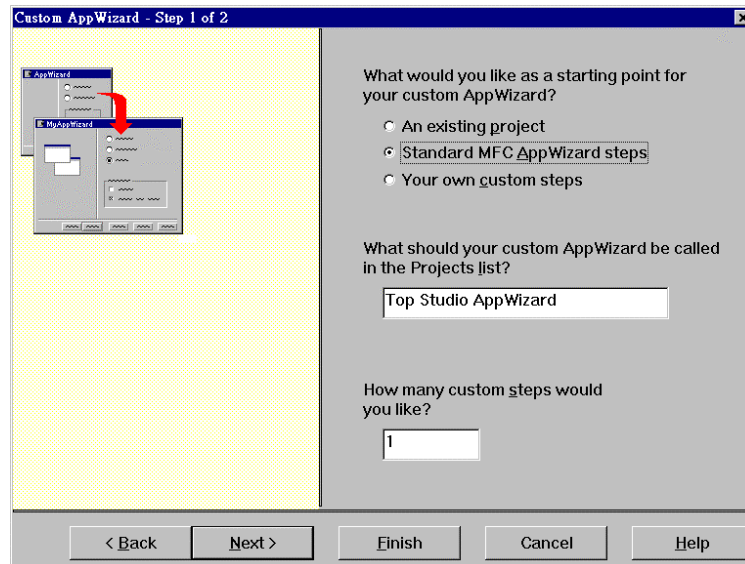
Developers Studio 提供了一個讓我們製作 custom AppWizard 的 Wizard，就叫作 **Custom AppWizard**。讓我們先實地操作一下這個工具，再來談程式技術問題。

注意：以下我以 **Custom AppWizard** 表示 Visual C++ 所附的工具，custom AppWizard 表示我們希望做出來的「訂製型 AppWizard」。

選按【File/New】，在對話窗中選擇 **Custom AppWizard**，然後在右邊填寫你的專案名稱：



按下【OK】鈕，進入步驟一畫面：

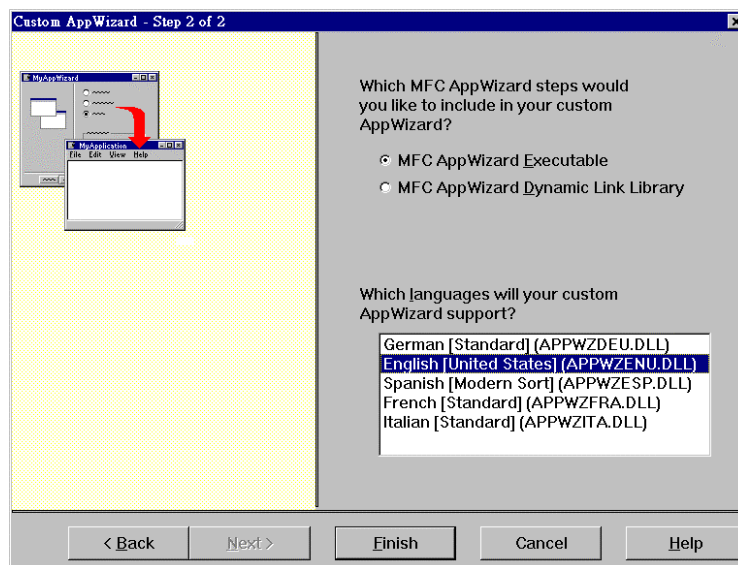


你可以選擇三種可能的擴充型式：

1. **An existing project**：根據一個原已存在的專案檔（*.dsp）來產生一個 custom AppWizard。
2. **Standard MFC AppWizard steps**：根據某個原有的 AppWizard，為它加上額外的幾個步驟，成為一個新的 custom AppWizard。這是一般最被接受的一種方式。
3. **Your own custom steps**：有全新的步驟和全新的對話窗畫面。這當然是最大彈性的展現啦，並同時也是最困難的一種作法，因為你要自行負責所有的工作。哪些工作呢？稍後我有一個例子使用第二種型式，將介紹所謂的 macros 和 directievs，你可以從中推而想之這第三種型式的繁重負擔。

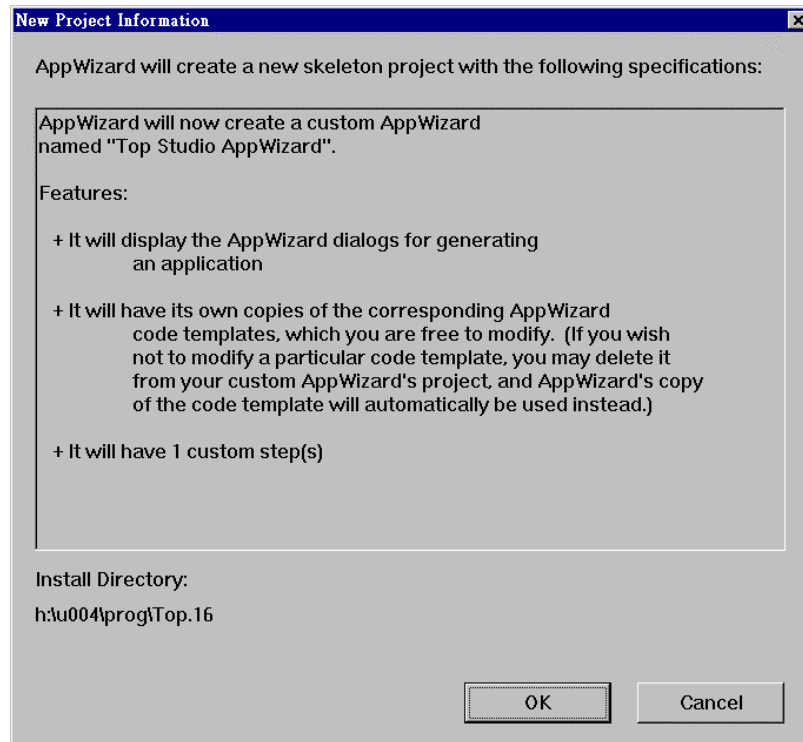
我的目的是做出一個屬於我個人研究室 ("Top Studio") 專用的 custom AppWizard，以原本的 **MFC AppWizard (exe)** 為基礎（有六個步驟），再加上一頁（一個步驟），讓程式員填入姓名、簡易說明，然後 **Top Studio AppWizard** 能夠把這些資料加到每一個原始碼檔案最前端。所以，我應該選擇上述三種情況的第二種：**Standard MFC AppWizard steps**，並在上圖下方選擇欲增加的步驟數量。本例為 1。

接下來按【Next】進入 Custom AppWizard 的第二頁：



既然剛剛選擇的是 **Standard MFC AppWizard steps**，這第二頁便問你要製造出 MFC Exe 或 MFC Dll。我選擇 MFC Exe。並在對話窗下方選擇使用的文字：英文。很可惜目前這裡沒有中文可供選擇。

這樣就完成了訂製的程序。按下【Finish】鈕，你獲得一張清單：



再按下【OK】鈕，開始產生程式碼。然後點選整合環境中的【Build/Top Studio.awx】。整合環境下方出現 "Making help file..." 字樣。這時候你要注意了，上個廁所喝杯咖啡後它還是那樣，一點動靜都沒有。原來，整合環境啟動了 Microsoft Help Workshop，而且把它極小化；你得把它叫出來，讓它動作才行。

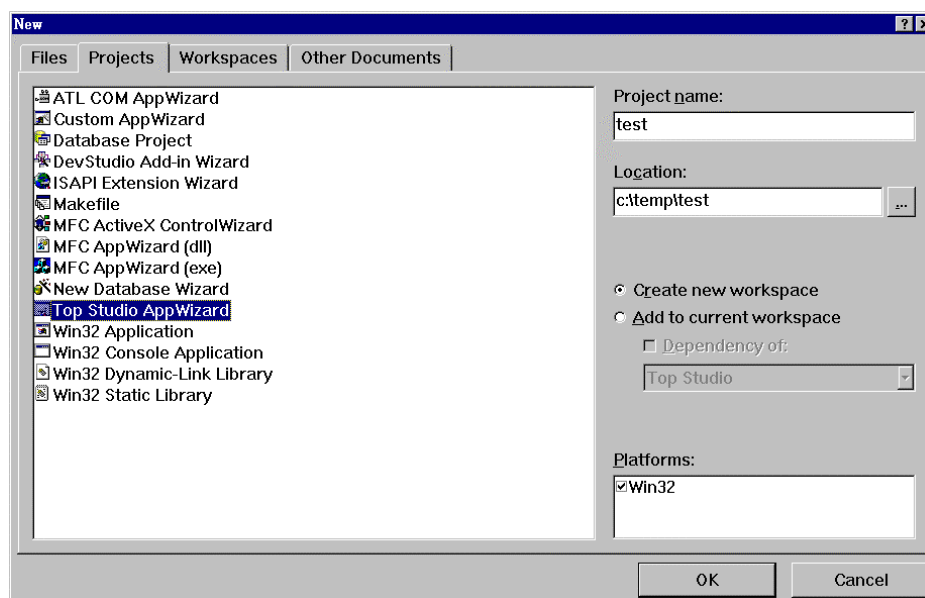
如果你不想要那些佔據很大磁碟空間的 HLP 檔和 HTM 檔，也可以把 Microsoft Help Workshop 關掉，控制權便會回到整合環境來，開始進行編譯連結的工作。

建造過程完畢，我們獲得了一個 "Top Studio.Awx" 檔案。這個檔案會被整合環境自動拷貝到 \DevStudio\SharedIDE\Template 磁碟目錄中：

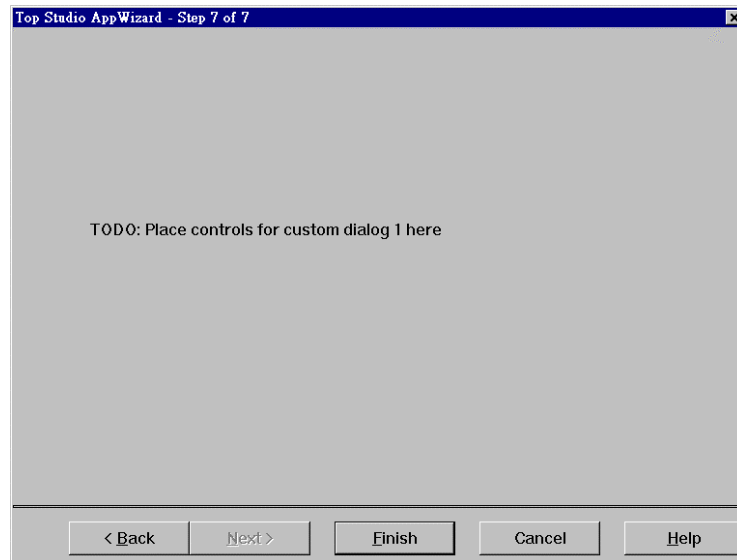
Directory of **E:\DevStudio\SharedIDE\Template**

ATL	<DIR>	03-29-97	14:12	ATL
MFC	RCT	4,744	12-04-95	16:09 MFC.RCT
README	TXT	115	10-30-96	17:54 README.TXT
TOPSTU~1	AWX	523,776	04-07-97	17:01 Top Studio.awx
TOPSTU~1	PDB	640,000	04-07-97	17:01 Top Studio.pdb

現在，再一次選按整合環境的【File/New】，在【Projects】對話窗中我們看到 **Top Studio AppWizard** 出現了：



試試它的作用。請像使用一般的 **MFC AppWizard** 那樣使用它（像第 4 章那樣），你會發現它有 7 個步驟。前 6 個和 **MFC AppWizard** 完全一樣，第 7 個畫面如下：



哇喔，怎麼會這樣？當然是這樣，因為你還沒有做任何程式動作嘛！目前 **Top Studio AppWizard** 產生出來的程式碼和第 4 章的 **Scribble step0** 完全相同。

剖析 AppWizard Components

圖 **15-2** 是 AppWizard components 的架構圖。所謂 AppWizard components，就是架構出一個 AppWizard 的所有「東西」，包括：

1. Dialog Templates (Dialog Resources)
2. Dialog Classes
3. Text Templates (Template 子目錄中的所有 .H 檔和 .CPP 檔)
4. Macro Dictionary
5. Information Files

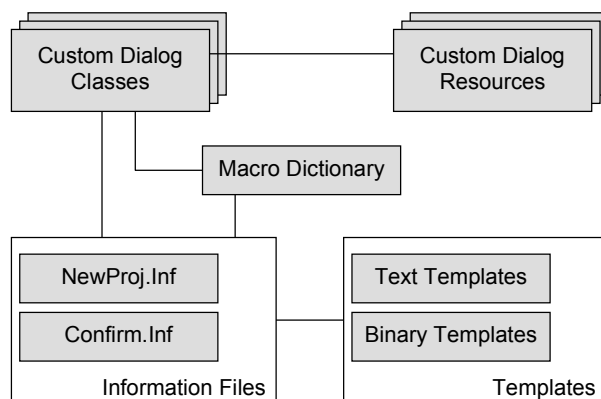


圖 15-2 用以產生一個 custom AppWizard 的各種 components。

Dialog Templates 和 Dialog Classes

以 **Top Studio AppWizard** 為例，由於多出一個對話窗畫面，我們勢必需要產生一個對話窗面板（template），還要為這面板產生一個對應的 C++ 類別，並以 DDX/DDV（第 10 章）取得使用者的輸入資料。這些技術我們已經在第 10 章中學習過。

獲得的使用者輸入資料如何放置到程式碼產生器所產生的專案原始碼中？

喔，到底誰是程式碼產生器？老實說我也沒有辦法明確指出是哪個模組，哪個檔案（也許就是 AWX 本身）。但是我知道，程式碼產生器會讀取 .AWX 檔，做出適當的原始碼來。而 .AWX 不正是前面才剛由 **Custom AppWizard** 做出來嗎？裡面有些什麼蹊蹺呢？是的，有許多所謂的 macros 和 directives 存在於 **Custom AppWizard** 所產生的 "text template"（也就是 template 子目錄中的所有 .CPP 和 .H 檔）中。以 **Top Studio AppWizard** 為例，我們獲得這些檔案：

```

H:\U004\PROG\TOP.15 :
Top Studio.h
StdAfx.h
Top StudioAw.h
Debug.h
Resource.h
    
```

```

Chooser.h
CstmlDlg.h      <---- 稍後要修改此檔內容
Top_Studio.cpp
StdAfx.cpp
Top_StudioAw.cpp
Debug.cpp
Chooser.cpp
CstmlDlg.cpp    <---- 稍後要修改此檔內容
; K

H:\U004\PROG\TOP.15\TEMPLATE :      <---- 稍後要修改所有這些檔案的內容
DlgRoot.h
Dialog.h
Root.h
StdAfx.h
Frame.h
ChildFrm.h
Doc.h
View.h
RecSet.h
SrvrItem.h
IpFrame.h
CntrItem.h
DlgRes.h
Resource.h
DlgRoot.cpp
Dialog.cpp
Root.cpp
StdAfx.cpp
Frame.cpp
ChildFrm.cpp
Doc.cpp
View.cpp
RecSet.cpp
SrvrItem.cpp
IpFrame.cpp
CntrItem.cpp
NewProj.inf
Confirm.inf

```

Macros

我們慣常所說的程式中的 macro，通常帶有「動作」。這裡的 macro 則是用來代表一個常數。前後以 \$\$ 包夾起來的字串即為一個 macro 名稱，例如：

```
class $$FRAME_CLASS$$ : public $$FRAME_BASE_CLASS$$
```

程式碼產生器看到這樣的句子，如果發現 `$$FRAME_CLASS$$` 被定義為 "CMDIFrameWnd"，`$$FRAME_BASE_CLASS$$` 被定義為 "CFrameWnd"，就產生出這樣的句子：

```
class CMDIFrameWnd : public CFrameWnd
```

Developer Studio 系統已經內建一組標準的 macros 如下，給 AppWizard 所產生的每一個專案使用：

巨集名稱	意義
APP	應用程式的 <i>CWinApp</i> -driven class.
FRAME	應用程式的 main frame class.
DOC	應用程式的 document class.
VIEW	應用程式的 view class.
CHILD_FRAME	應用程式的 MDI child frame class (如果有的話)
DLG	應用程式的 main dialog box class (在 dialog-based 程式中)
RESET	應用程式的 recordset class (如果有的話)
SRVRITEM	應用程式的 main server-item class (如果有的話)
CNTRITEM	應用程式的 main container-item class (如果有的話)
IPFRAME	應用程式的 in-place frame class (如果有的話)

另外還有一組 macro，可以和前面那組搭配運用：

巨集名稱	意義
class	類別名稱 (小寫)
CLASS	類別名稱 (大寫)
base_class	基礎類別的名稱 (小寫)
BASE_CLASS	基礎類別的名稱 (大寫)
ifile	實作檔名稱 (.CPP 檔，不含副檔名) (小寫)

IFILE	實作檔名稱（.CPP 檔，不含副檔名）（大寫）
hfile	表頭檔名稱（.H 檔，不含副檔名）（小寫）
HFILE	表頭檔名稱（.H 檔，不含副檔名）（大寫）
ROOT	應用程式的專案名稱（全部大寫）
root	應用程式的專案名稱（全部小寫）
Root	應用程式的專案名稱（可以引大小寫）

圖 15-3 列出專案名稱爲 Scribble 的某些個標準巨集內容。

巨集	實際內容
APP_CLASS	CScribbleApp
VIEW_IFILE	SCRIBBLEVIEW
DOC_HFILE	SCRIBBLEDOC
doc_hfile	scribbledoc
view_hfile	scribbleview

圖 15-3 專案名稱爲 Scribble 的數個標準巨集內容。

Directives

所謂 directives，類似程式語言中的條件控制句（像是 if、else 等等），用來控制 text templates 中的流程。字串前面如果以 \$\$ 開頭，就是一個 directive，例如：

```

$$IF (PROJTYPE_MDI)
...
$$ELSE
...
$$ENDIF

```

每一個 directive 必須出現在每一行的第一個字元。

系統提供了一組標準的 directives 如下：

```
$$IF  
$$ELIF  
$$ELSE  
$$ENDIF  
$$BEGINLOOP  
$$ENDLOOP  
$$SET_DEFAULT_LANG  
$$//  
$$INCLUDE
```

動手修改 Top Studio AppWizard

我的目的是做出一個屬於我個人研究室專用的 **Top Studio AppWizard**，以原本的 **MFC AppWizard (exe)** 為基礎，加上第 7 個步驟，讓程式員填入姓名、簡易說明，然後 **Top Studio AppWizard** 就能夠把這些資料加到每一個原始碼檔案最前端。

看來我們已經找到出口了。我們應該先為 **Top Studio AppWizard** 產生一個對話窗，當做步驟 7 的畫面，再產生一個對應的 C++ 類別，於是 DDX 功能便能夠取得對話窗所接收到的輸入字串（程式員姓名和程式主旨）。然後我們設計一些 macros，再撰寫一小段碼（其中用到那些 macros），把這一小段碼加到每一個 .CPP 和 .H 檔的最前面。大功告成。

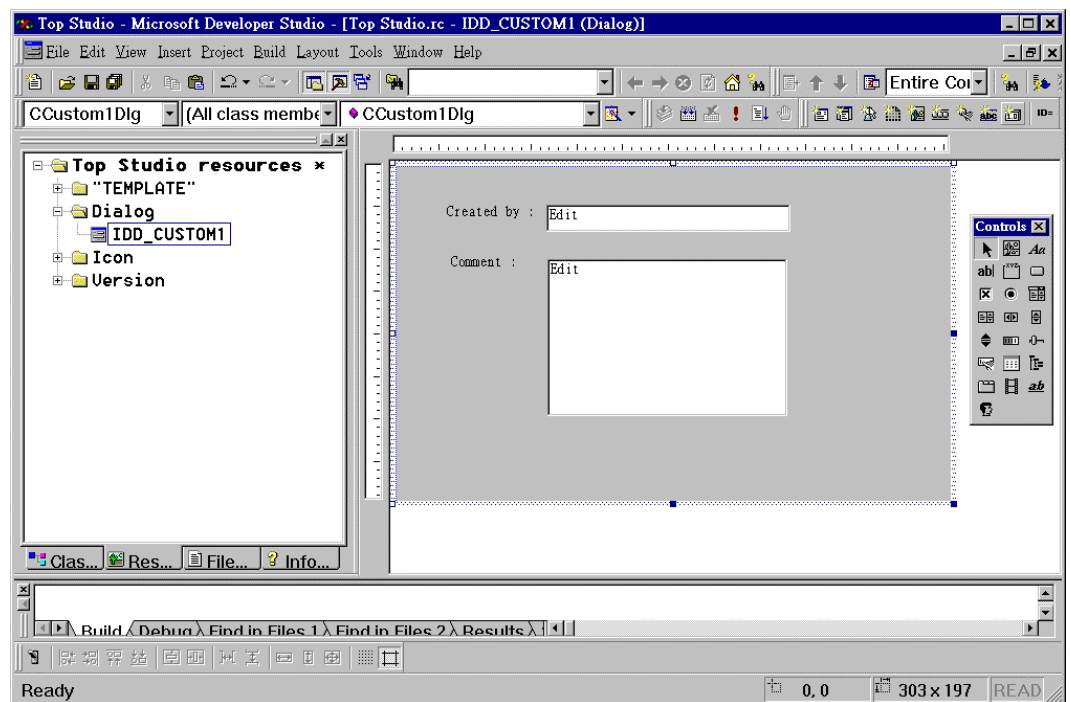
本例不需要我們動手寫 directives。

我想我遺漏了一個重要的東西。Macros 如何定義？放在什麼地方？我曾經在本書第 8 章介紹 Scribble 的資料結構時，談到 collection classes。其中有一種資料結構名為 Map（也就是 Dictionary）。Macros 正是被定義並儲存在一個 Map 之中，並以 macro 名稱做為鍵值（key）。

讓我們一步一步來。

利用資源編輯器修改 IDD_CUSTOM1 對話窗畫面

請參考第 4 章和第 10 章，修改 *IDD_CUSTOM1* 對話窗畫面如下：



兩個 edit 控制元件的 ID 如圖 15-4 所示。

利用 ClassWizard 修改 IDD_CUSTOM1 對話窗的對應類別 CCustom1Dlg

圖 15-4 列出每一個控制元件的型態、識別碼及其對應的變數名稱等資料。變數將做為 DDX 所用。修改動作如圖 15-5。

control ID	名稱	種類	變數型態
IDC_EDIT_AUTHOR	m_szAuthor	Value	CString
IDC_EDIT_COMMENT	m_szComment	Value	CString

圖 15-4 IDD_CUSTOM1 對話窗控制元件的型態、ID、對應的變數名稱

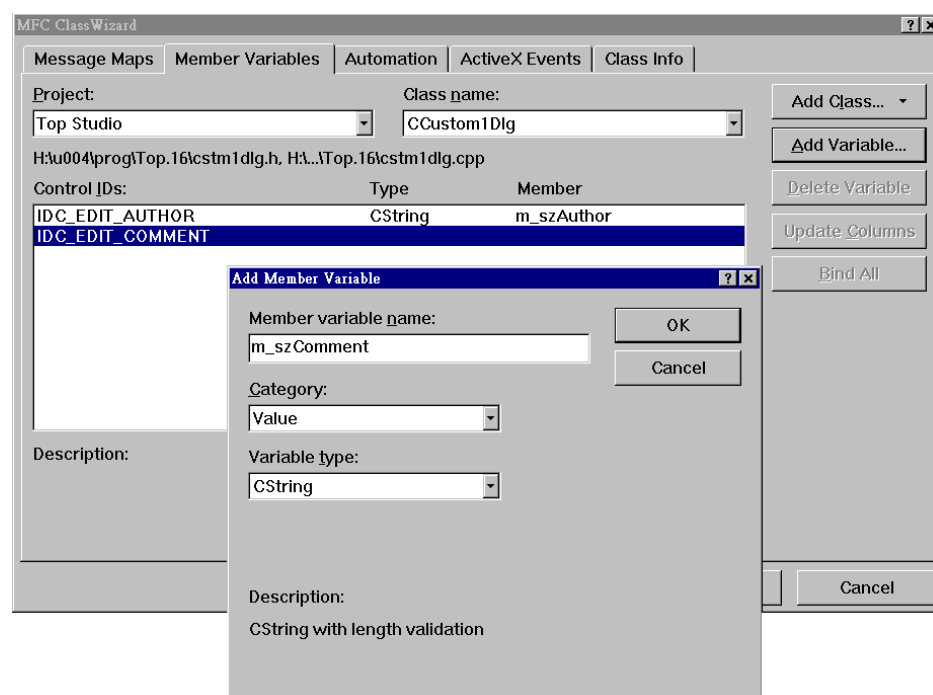


圖 15-5 利用 ClassWizard 為 IDD_CUSTOM1 對話窗的兩個 edit 控制元件加上兩個對應的變數 m_szAuthor 和 m_szComment，以為 DDX 所用。

Custom AppWizard 為我們做出來的這個 *CCustom1Dlg* 必定衍生自 *CAppWizStepDlg*。你不會在 MFC 類別架構檔中發現 *CAppWizStepDlg*，它是 Visual C++ 的 *mfcapwz.dll* 所提供的一個類別。此類別有一個虛擬函式 *OnDismiss*，當 AppWizard 的使用者選按【Back】或【Next】或【Finish】鈕時就會被喚起。如果它傳回 *TRUE*，AppWizard 就可

以切換對話窗；如果傳回的是 *FALSE*，就不能。我們可以在這個函式中做數值檢驗的工作，更重要的是做 *macros* 的設定工作。

修改 OnDismiss 虛擬函式，在其中定義 macros

前面我已經說過，*macros* 的定義儲存在一個 *Map* 結構中。它在哪裡？

整個 **Top Studio AppWizard**（以及其他所有的 *custom AppWizard*）的主類別係衍生自系統提供的 *CCustomAppWiz*：

```
// in Top StudioAw.h
class CTopStudioAppWiz : public CCustomAppWiz
{
    ....
};

// in "Top StudioAw.cpp"
CTopStudioAppWiz TopStudioaw; // 類似 application object。
                                // 物件命名規則是 "專案名稱" + "aw"。
```

你不會在 *MFC* 類別架構檔中發現 *CCustomAppWiz*，它是 *Visual C++* 的 *mfcapwz.dll* 所提供的一個類別。此類別擁有一個 *CMapStringToString* 物件，名為 *m_Dictionary*，所以 *TopStudioaw* 自然就繼承了 *m_Dictionary*。這便是儲存 *macros* 定義的地方。我們可以利用 *TopStudioaw.m_Dictionary[xxx] = xxx* 的方式來加入一個個的 *macros*。

現在，改寫 *OnDismiss* 虛擬函式如下：

```
#0001 // This is called whenever the user presses Next, Back, or Finish with this step
#0002 // present. Do all validation & data exchange from the dialog in this function.
#0003 BOOL CCustom1Dlg::OnDismiss()
#0004 {
#0005     if (!UpdateData(TRUE))
#0006         return FALSE;
#0007
#0008     if( m_szAuthor.IsEmpty() == FALSE )
#0009         TopStudioaw.m_Dictionary["PROJ_AUTHOR"] = m_szAuthor;
#0010     else
```

```
#0011         TopStudioaw.m_Dictionary["PROJ_AUTHOR"] = "";
#0012
#0013     if( m_szComment.IsEmpty() == FALSE )
#0014         TopStudioaw.m_Dictionary["PROJ_COMMENT"] = m_szComment;
#0015     else
#0016         TopStudioaw.m_Dictionary["PROJ_COMMENT"] = "";
#0017
#0018     CTime date = CTime::GetCurrentTime();
#0019     CString szDate = date.Format( "%A, %B %d, %Y" );
#0020     TopStudioaw.m_Dictionary["PROJ_DATE"] = szDate;
#0021
#0022     return TRUE; // return FALSE if the dialog shouldn't be dismissed
#0023 }
```

這麼一來我們就定義了三個 macros：

macro 名稱	macro 內容
PROJ_AUTHOR	m_szAuthor
PROJ_DATE	szDate
PROJ_COMMENT	m_szComment

修改 text template

現在，為 **Top Studio AppWizard** 的 `template` 子目錄中的每一個 `.H` 檔和 `.CPP` 檔增加一小段碼，放在檔案最前端：

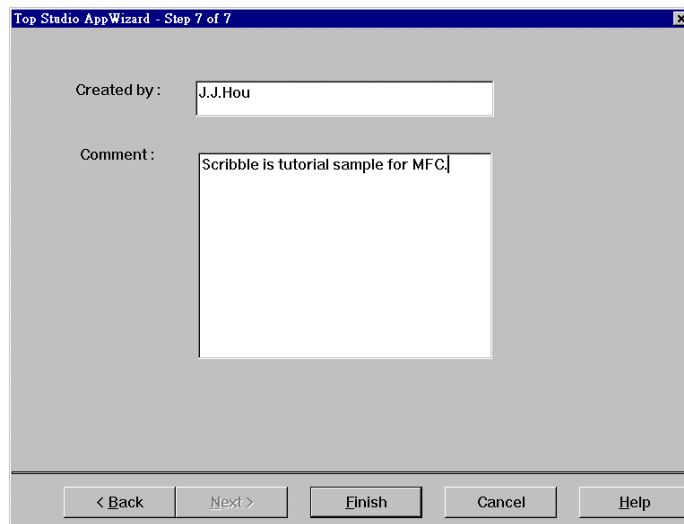
```
/*
    This project was created using the Top Studio AppWizard

    $$PROJ_COMMENT$$

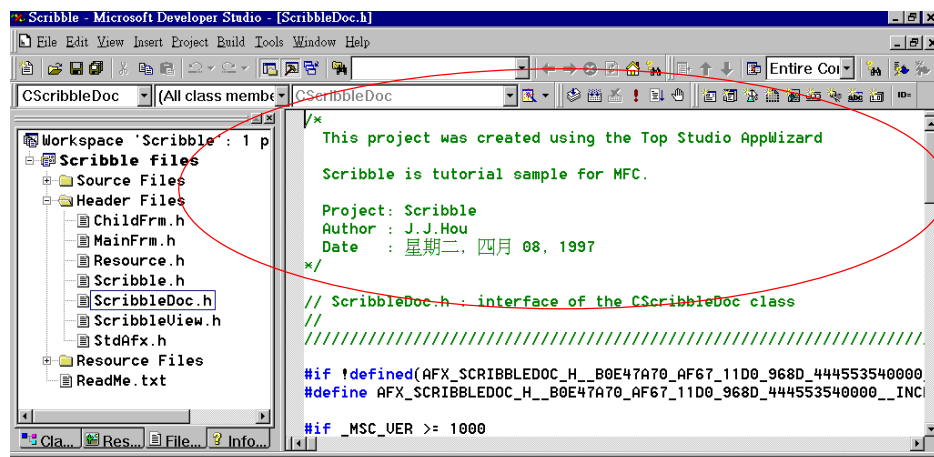
    Project: $$Root$$
    Author  : $$PROJ_AUTHOR$$
    Date   : $$PROJ_DATE$$
*/
```

Top Studio AppWizard 執行結果

重新編譯連結，然後使用 **Top Studio AppWizard** 產生一個專案。第 7 個步驟的畫面如下：



由 **Top Studio AppWizard** 產生出來的程式碼中，每一個 .CPP 和 .H 檔最前面果然有下面數行文字，大功告成。



更多的參考

我在本章中只是簡單示範了一下「繼承自原有之 Wizard，再添加新功能」的作法。這該算是半自助吧。全自助的作法就複雜許多。Walter Oney 有一篇 "Pay No Attention to the Man Behind the Curtain! Write Your Own C++ AppWizards" 文章，發表於 *Microsoft Systems Journal* 的 1997 三月號，裡面詳細描述了全自助的作法。請注意，他是以 Visual C++ 4.2 為演練對象。不過，除了畫面不同，技術上完全適用於 Visual C++ 5.0。

Dino Esposito 有一篇文章 "a new assistant"，發表於 *Windows Tech Journal* 的 1997 三月號，也值得參考。1997 年五月份的 *Dr. Dobbs's Journal* 也有一篇名為 "Extending Visual C++ : Custom AppWizards make it possible" 的文章，作者是 John Roberts。

第 16 頁

站上眾人的肩膀 - 使用 Components & ActiveX Controls

從 Visual Basic 開始，可以說一個以 components（軟體元件）為中心的程式設計時代，逐漸拉開了序幕。隨後 Delphi 和 C++ Builder 陸續登場。Visual Basic 使用 VBX (Visual Basic eXtension) 元件，Delphi 和 C++ Builder 使用 VCL (Visual Component Library) 元件，Visual C++ 則使用 OCX (OLE Control eXtension) 元件。如今 OCX 又演化到所謂 ActiveX 元件（其實和 OCX 大同小異）。

Microsoft 的 Visual Basic (使用 Basic 語言)，Borland 的 Delphi (使用 Pascal 語言)，以及 Borland 的 C++ Builder (使用 C++ 語言)，都稱得上是一種快速開發工具 (RAD, Rapid Application Development)。它們所使用的元件都是 PME (Properties-Method-Event) 架構。這使得它們的整合環境 (IDE) 能夠做出非常視覺化的開發工具，以拖放、填單的方式完成絕大部份的程式設計工作。它們的應用程式開發程序大約是這個樣子：

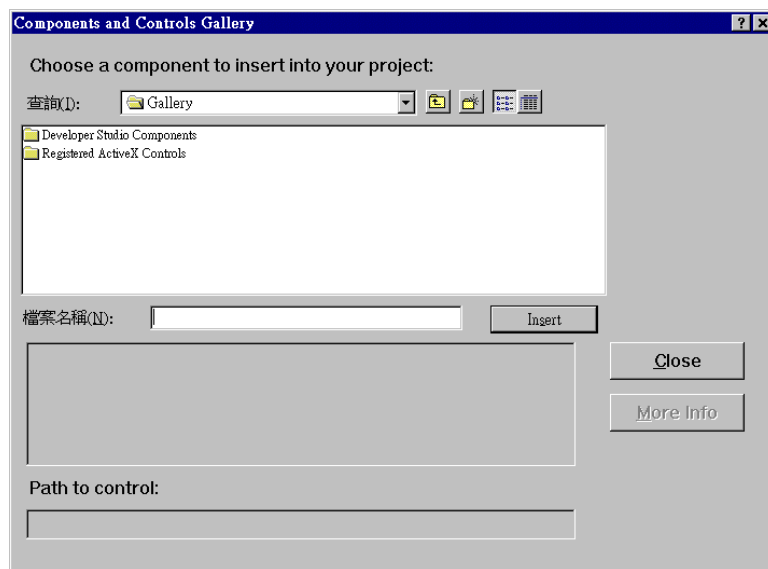
1. 選擇一些適當的軟體元件 (VBX 或 VCL)。
2. 打開一個 form，把那些軟體元件拖放到 form 中適當的位置。
3. 在 Properties 清單中填寫適當的屬性。例如精確位置、寬度高度、或是讓 A 元件的某個屬性連接到 B 元件...等等。
4. 撰寫程式碼 (method)，做為某種 event 發生時的處理常式。

依我的看法，Visual C++ 還不能夠算是 RAD。雖然，MFC 程式所能夠使用的 OCX 也是 PME（Properties-Method-Event）架構，但 Visual C++ 整合環境沒有能夠提供適當工具讓我們以那麼視覺化的方式（像 VB 或 Delphi 或 C++ Builder 那樣拖放、填單）就幾乎完成一個程式。

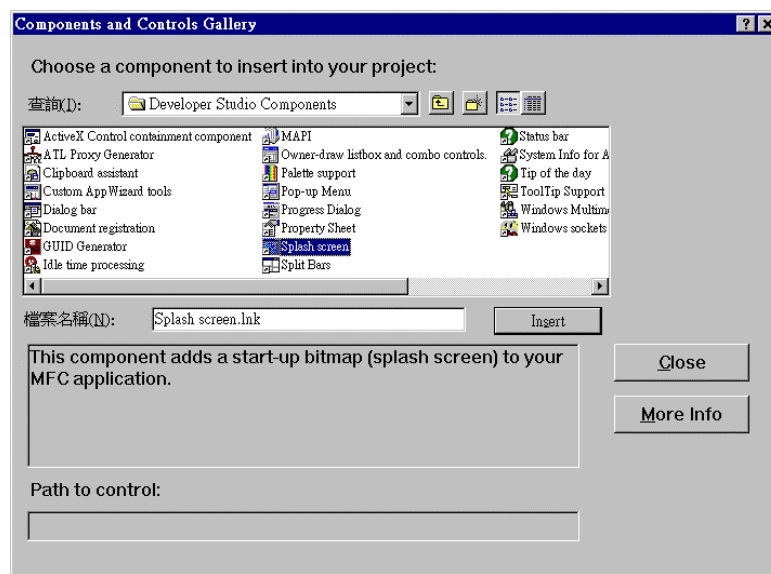
什麼是 Component Gallery

Component Gallery 是自從 Visual C++ 4.0 之後，整合環境中新增的一個東西。你可以把它想像成一個資料庫，儲存著 ActiveX controls 和可重複使用的 C++ 類別（也就是本章所謂的 components）。VC++ 5.0 的 Component Gallery 的使用介面和 VC++ 4.x 有某種程度的不同，不過操控原則基本上是一致的。

當你安裝了 Visual C++ 5.0，Component Gallery 已經內含了一些微軟所提供的 components 和 ActiveX controls（註：以下我將把這兩樣東西統稱為「元件」）。選按整合環境的【Project / Add To Project / Components and Controls...】選單項目，你就可以看到 Component Gallery：



其中有 Developer Studio Components 和 Registered ActiveX Controls 兩個資料夾，打開任何一個，就會出現目前系統所擁有的「貨色」：

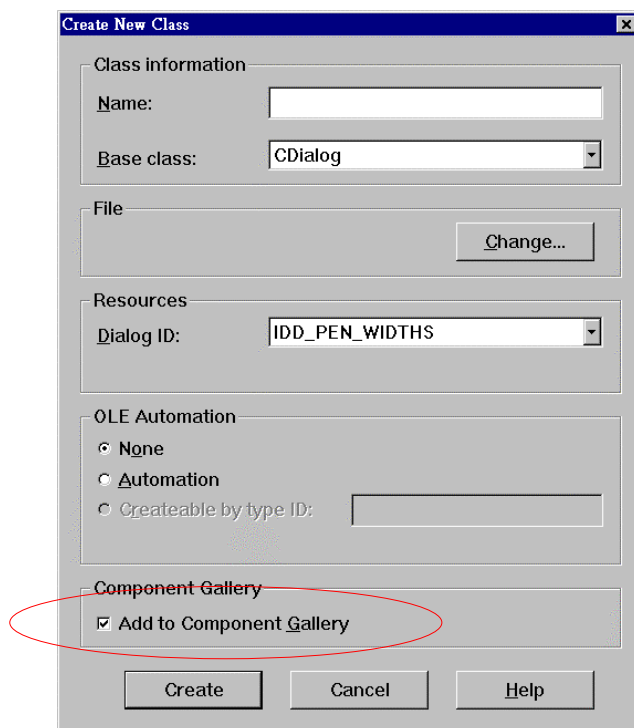


如果你以為這些元件儲存在兩個地方（一個是它本來的位置，另一份拷貝放在 Component Gallery 之中），那你就錯了。Component Gallery 只是存放那些元件的位置資料而已。你可以說，只是存放一個「聯結」而已。

為什麼元件在此分為 Components 和 ActiveX controls 兩種？有什麼不同。簡單地說，Components 是一些已寫好的 C++ 類別。基本上 C++ 類別本來就具有重複使用性，Component Gallery 只是把它們多做一些必要的包裝，連同其他資源放在一起成為一個包裹。當你需要某個 component，Component Gallery 給你的是該 components 的原始碼。

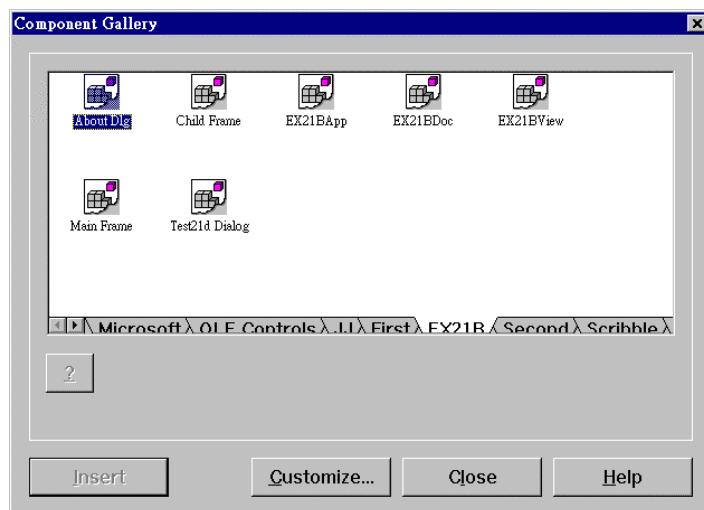
ActiveX controls 不一樣。當你選用某個 ActiveX controls，Component Gallery 當然也會為你填入一些碼，但它們不是元件的本體。那些碼只是使用元件時所必須的碼，元件本身在 .OCX 檔案中（通常註冊後的 OCX 檔都放在 Windows\System 磁碟子目錄）。

ActiveX controls 是很完整的一個有著 PME (Properties-Method-Event) 架構的控制元件，但一般欲被重複使用的 C++ 類別卻不會有那麼完整的設計或包裝。要把一個 C++ 類別做成完好的包裝，放到 Component Gallery 中，它必須變為一個單一檔案，內含類別資訊以及任何必須的資源。這在過去的 Visual C++ 4.x 中是很容易的事情，因為每次你使用 ClassWizard 新增一個類別，就有一個核示盒詢問你要不要加到 Component Gallery：



Visual C++ 4.x 的 ClassWizard 新增類別對話窗

但這一選項已在 Visual C++ 5.0 中拿掉(你可以在第 10 章增加對話窗類別時看到新的畫面)。看來似乎要增加 components 不再是那麼方便了。這倒也不是壞事，我想許多人在設計程式時忽略了上圖那個選項，於是每一個專案中的每一個類別，都被包裝到 Component Gallery 去，而其中許多根本是沒有價值的：



Visual C++ 4.x 的 **Component Gallery**。常常因為程式員的疏忽，而產生了一大堆沒有價值的 **components**。

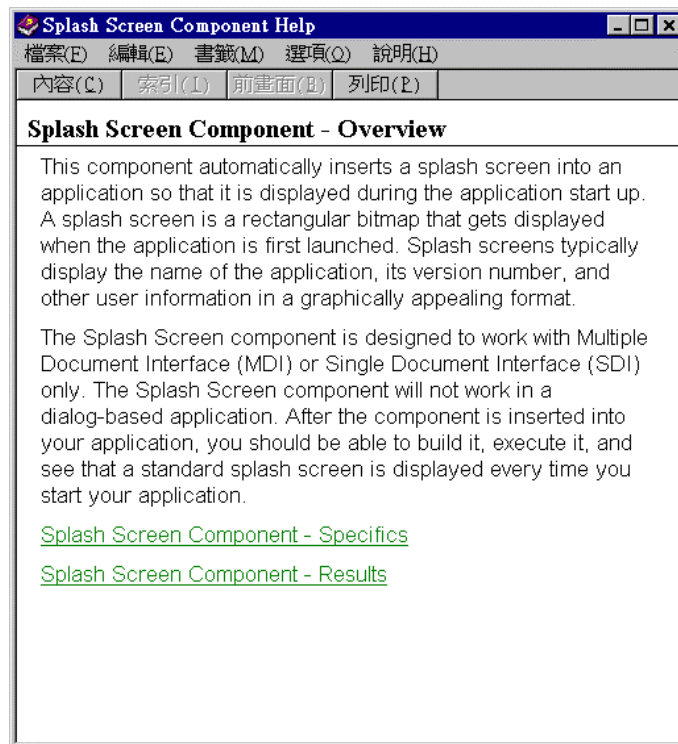
使用 Components

當你選擇 **Component Gallery** 中的 **Developer Studio Components** 資料夾，出現許多的 **components**。面對形形色色的「貨」，你的心裡一定嘀咕著：怎麼用嘛？幸好畫面上有一個【**More Info**】按鈕，可以提供你比較多的資訊。以下我挑三個最簡單的 **components** 做示範。

Splash screen

所謂 **Splash Screen**，你可以說它是一個「炫耀畫面」。玩過微軟的 **Office** 嗎？每一個 **Office** 軟體一出場，在它做初始化的那段時間裡，都會出現一個畫面，就是 **Splash screen**。

Splash Screen 的【**More Info**】出現這樣的畫面：



選按上圖下方的 "Splash Screen Component - Specifics"，你會獲得一張使用規格說明，大意如下：

欲插入 **splash Screen** component，你必須：

1. 打開你希望安插 **Splash Screen** component 的那個專案。
2. 選擇整合環境中的【Project/Add To Project/Components and Controls】選單項目。
3. 選擇 "Developer's Studio Components" 資料夾。
4. 選擇資料夾中的 **Splash Screen** component 並按下【Insert】鈕。
5. 設定必要的 **Splash Screen** 選項然後按下【OK】鈕。
6. 重建（重新編譯連結）專案。

如果要把 **Splash Screen** 加到一個以對話窗為主（dialog-based）的程式中，你必須在插

入這個 component 之後做以下事情：

1. 找到你的 *InitInstance* 函式。

2. 在你呼叫：

```
int nResponse = dlg.DoModal();
```

之前，加上一行：

```
spl.ShowSplashScreen(FALSE);
```

增加這一行碼，可以確保 **Splash Screen** 在主對話窗被顯示之前，會被清除掉。

看來很簡單的樣子 ☺

System Info for About Dlg

看過 WordPad 的【About】對話窗嗎：



如果你也想讓自己的對話窗有點系統資訊的顯示能力，可以採用 Component Gallery 提供的這個 **System Info for About Dlg** component。它的規格說明文字如下：

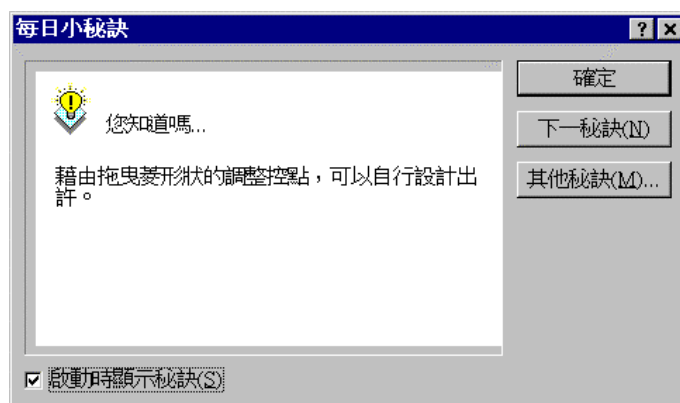
SysInfo component 可以為你的程式的 About 對話窗中加上一些系統資訊（可用記憶體數量以及磁碟剩餘空間）。你的程式必須以 MFC AppWizard 完成。請參考 WordPad 說

明文件以獲得更多資訊。

這份規格書不夠詳細。稍後我會在修改程式碼時加上我自己的說明。

Tip of the Day

看過這種畫面嗎（微軟的 Office 軟體就有）：



這就是「每日小秘訣」。Component Gallery 提供的 **Tips for the Day** component 讓你很方便地為自己加上「每日小秘訣」。這個 component 的使用規格是：

小秘訣文字檔（TIPS.TXT）：

擁有 **Tips for the Day** component 的程式將搜尋磁碟中的工作子目錄，企圖尋找 TIPS.TXT 讀取秘訣內容。如果你希望這個秘訣文字檔有不同的名稱或是放在不同的位置，你可以修改 CTIP.CPP 中的 CTIP 類別建構式。CTIP 是預設的類別名稱。

（侯俊傑註：最後這句話是錯誤的。我使用這個 component，接受所有的預設項目，獲得的類別名稱卻是 CTIPDLG，檔案則為 TIPDLG.CPP）

■ TIPS.TXT 的格式如下：

1. 檔案必須是 ASCII 文字，每一個秘訣以一行文字表示。

2. 如果某一行文字以分號(;)開頭，表示這是一行說明文字，不生實效。說明文字必須有自己單獨的一行。
3. 空白行會被忽略。
4. 每一個小秘訣最多 1000 個字元。
5. 每一行不能夠以空白或定位符號(tab)開始。

■ 小秘訣顯示次序：

預設情況下，小秘訣的出現次序和它們在檔案中的排列次序相同。如果全部都出現過了，就再循環一遍。如果檔案被更改過了，顯示次序就會從頭開始。

■ 錯誤情況：

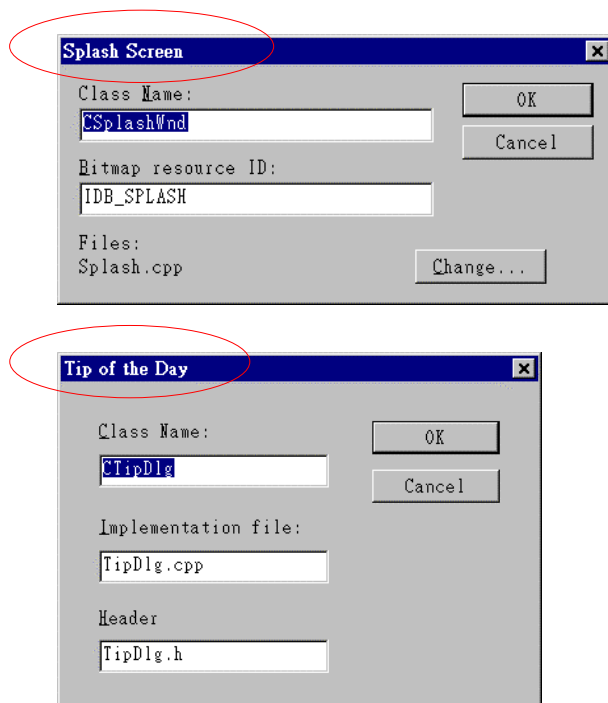
這個元件希望在 MFC 程式中被使用。你的程式應該只有一個衍生自 *CWinApp* 的類別。如果有許多個 *CWinApp* 衍生類別，此元件會選擇其中第一個做為實作的對象。其他的錯誤情況包括秘訣文字檔不存在，或格式不對等等。

■ 在程式的【Help】選單中加上 **Tip of The Day** 項目：

這個元件會修改主框視窗的 *OnInitMenu* 函式，並且在你的【Help】選單下加掛一個 **Tip of The Day** 項目。如果你的程式原本沒有【Help】選單，此元件就自動為你產生一個。

Components 實際運用：ComTest 程式

現在，動手吧。首先利用 MFC AppWizard 產生一個專案，就像第 4 章的 Scribble step0 那樣。我把它命名為 ComTest（放在書附光碟的 ComTest.17 子目錄中）。然後，不要離開這個專案，啟動 Component Gallery，進入 Developer Studio Components 資料夾，分別選擇 **Splash Screen** 和 **System Info for About Dlg** 和 **Tips of the Day** 三個元件，分別按下【Insert】鈕。**Splash Screen** 和 **Tips of the Day** 元件會要求我們再指定一些訊息：



新增檔案

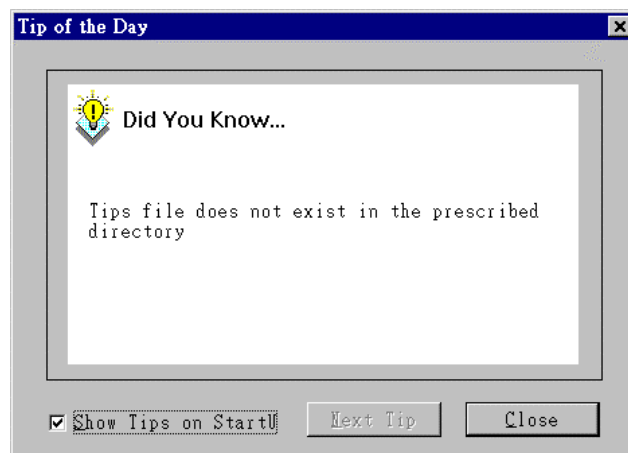
這時候 ComTest 專案中的原始碼有了一些變動（被 Component Gallery 改變）。被改變的檔案是：

```
STDAFX.H  
RESOURCE.H  
COMTEST.H  
COMTEST.CPP  
COMTEST.RC  
MAINFRM.H  
MAINFRM.CPP  
SPLASH.H  
SPLASH.CPP  
SPLSH16.BMP  
TIPDLG.CPP  
TIPDLG.H
```

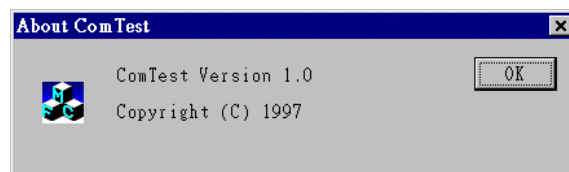
選按整合環境的【Build / Build ComTest.Exe】，把這個程式建造出來。建造完畢試執行之，你會發現在主視窗出現之前，一開始先有一張畫面顯現：



然後是每日小秘訣：



然後才是主視窗。至於 About 對話窗，畫面如下（沒啥變化）：



看來，我們只要修改一下 **Splash Screen** 畫面，並增加一個 TIPS.TXT 文字檔，再變化一下 About 對話窗，就成了。程式編修動作的確很簡單，不過我還是要把這三個元件加諸於你的程式的每一條痕印都揭發出來。

相關變化

讓我們分析分析 Component Gallery 爲我們做了些什麼事情。

STDAFX.H (陰影部份爲新增內容)

```
...
#include <afxwin.h>           // MFC core and st
#include <afxext.h>           // MFC extensions
#include <afxdisp.h>          // MFC OLE automat
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>           // MFC
#endif // _AFX_NO_AFXCMN_SUPPORT
#include <H:\u002p\prog\ComTest.16\TipDlg.h>
...
```

RESOURCE.H

下面是針對三個元件新增的一些常數定義。凡是稍後修改程式時會用到的常數，我都加上註解，提醒您特別注意。

```
...
#define IDB_SPLASH             102 // Splash screen 所加，代表一張 16 色 bitmap 畫面
#define CG_IDS_PHYSICAL_MEM    103
#define CG_IDS_DISK_SPACE      104
#define CG_IDS_DISK_SPACE_UNAVAIL 105
#define IDB_LIGHTBULB          106
#define IDD_TIP                 107
#define CG_IDS_TIPOFTHEDAY      108 // Tips 所加，一個字串。稍後我要把它改爲中文內容。
#define CG_IDS_TIPOFTHEDAYMENU 109
#define CG_IDS_DIDYOUKNOW       110 // Tips 所加，一個字串。稍後我要把它改爲中文內容。
#define CG_IDS_FILE_ABSENT      111
#define CG_IDP_FILE_CORRUPT     112
#define CG_IDS_TIPOFTHEDAYHELP 113
#define IDC_PHYSICAL_MEM        1000 // SysInfo 所加，代表「可用記憶體」這個 static 欄位
```

```

#define IDC_BULB                1000
#define IDC_DISK_SPACE          1001 // SysInfo 所加，代表「磁碟剩餘空間」這個 static 欄位
#define IDC_STARTUP             1001
#define IDC_NEXTTIP             1002
#define IDC_TIPSTRING           1004
...

```

COMTEST.H (陰影部份為新增內容)

```

class CComTestApp : public CWinApp
{
public:
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    CComTestApp();

...

private:
    void ShowTipAtStartup(void);
private:
    void ShowTipOfTheDay(void);
}

```

COMTEST.CPP (陰影部份為新增內容)

```

#0001 ...
#0002 #include "Splash.h"
#0003 #include <dos.h>
#0004 #include <direct.h>
#0005
#0006 BEGIN_MESSAGE_MAP(CComTestApp, CWinApp)
#0007     ON_COMMAND(CG_IDS_TIPOTHE DAY, ShowTipOfTheDay)
#0008     //{AFX_MSG_MAP(CComTestApp)
#0009     ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
#0010     // NOTE - the ClassWizard will add and remove mapping macros here.
#0011     //     DO NOT EDIT what you see in these blocks of generated code!
#0012     //}AFX_MSG_MAP
#0013     // Standard file based document commands
#0014     ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
#0015     ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
#0016     // Standard print setup command
#0017     ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
#0018 END_MESSAGE_MAP()
#0019
#0020 BOOL CComTestApp::InitInstance()
#0021 {
#0022     // CG: The following block was added by the Splash Screen component.

```

```

#0023     {
#0024         CCommandLineInfo cmdInfo;
#0025         ParseCommandLine(cmdInfo);
#0026         CSplashWnd::EnableSplashScreen(cmdInfo.m_bShowSplash);
#0027     }
#0028
#0029     AfxEnableControlContainer();
#0030     ...
#0031
#0032     // CG: This line inserted by 'Tip of the Day' component.
#0033     ShowTipAtStartup();
#0034
#0035     return TRUE;
#0036 }
#0037 ...
#0038 BOOL CComTestApp::PreTranslateMessage(MSG* pMsg)
#0039 {
#0040     // CG: The following lines were added by the Splash Screen component.
#0041     if (CSplashWnd::PreTranslateAppMessage(pMsg))
#0042         return TRUE;
#0043
#0044     return CWinApp::PreTranslateMessage(pMsg);
#0045 }
#0046
#0047 BOOL CAboutDlg::OnInitDialog()
#0048 {
#0049     CDialog::OnInitDialog(); // CG: This was added by System Info Component.
#0050
#0051     // CG: Following block was added by System Info Component.
#0052     {
#0053         CString strFreeDiskSpace;
#0054         CString strFreeMemory;
#0055         CString strFmt;
#0056
#0057         // Fill available memory
#0058         MEMORYSTATUS MemStat;
#0059         MemStat.dwLength = sizeof(MEMORYSTATUS);
#0060         GlobalMemoryStatus(&MemStat);
#0061         strFmt.LoadString(CG_IDS_PHYSICAL_MEM);
#0062         strFreeMemory.Format(strFmt, MemStat.dwTotalPhys / 1024L);
#0063
#0064         //TODO: Add a static control to your About Box to receive the memory
#0065         //      information. Initialize the control with code like this:
#0066         // SetDlgItemText(IDC_PHYSICAL_MEM, strFreeMemory);
#0067
#0068         // Fill disk free information

```

```

#0069         struct _diskfree_t diskfree;
#0070         int nDrive = _getdrive(); // use current default drive
#0071         if (_getdiskfree(nDrive, &diskfree) == 0)
#0072         {
#0073             strFmt.LoadString(CG_IDS_DISK_SPACE);
#0074             strFreeDiskSpace.Format(strFmt,
#0075                                     (DWORD)diskfree.avail_clusters *
#0076                                     (DWORD)diskfree.sectors_per_cluster *
#0077                                     (DWORD)diskfree.bytes_per_sector / (DWORD)1024L,
#0078                                     nDrive-1 + _T('A'));
#0079         }
#0080         else
#0081             strFreeDiskSpace.LoadString(CG_IDS_DISK_SPACE_UNAVAIL);
#0082
#0083         //TODO: Add a static control to your About Box to receive the memory
#0084         //         information. Initialize the control with code like this:
#0085         // SetDlgItemText(IDC_DISK_SPACE, strFreeDiskSpace);
#0086     }
#0087
#0088     return TRUE;    // CG: This was added by System Info Component.
#0089 }

```

COMTEST.RC (陰影部份為新增內容)

```

IDB_SPLASH BITMAP DISCARDABLE "Splsh16.bmp"
...
IDD_TIP_DIALOG DISCARDABLE 0, 0, 231, 164
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Tip of the Day"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL        "", -1, "Static", SS_BLACKFRAME, 12, 11, 207, 123
    LTEXT           "Some String", IDC_TIPSTRING, 28, 63, 177, 60
    CONTROL        "&Show Tips on StartUp", IDC_STARTUP, "Button",
    BS_AUTOCHECKBOX | WS_GROUP | WS_TABSTOP, 13, 146, 85, 10
    PUSHBUTTON     "&Next Tip", IDC_NEXTTIP, 109, 143, 50, 14, WS_GROUP
    DEFPUSHBUTTON  "&Close", IDOK, 168, 143, 50, 14, WS_GROUP
    CONTROL        "", IDC_BULB, "Static", SS_BITMAP, 20, 17, 190, 111
END
...
STRINGTABLE DISCARDABLE
BEGIN
    CG_IDS_PHYSICAL_MEM        "%lu KB"
    CG_IDS_DISK_SPACE          "%lu KB Free on %c:"
    CG_IDS_DISK_SPACE_UNAVAIL  "Unavailable"
    CG_IDS_TIPOFTHEDAY         "Displays a Tip of the Day."

```

```

CG_IDS_TIPOFTHEDAYMENU    "Ti&p of the Day..."
CG_IDS_DIDYOUKNOW         "Did You Know..."
CG_IDS_FILE_ABSENT        "Tips file does not exist in the prescribed directory;"
END

STRINGTABLE DISCARDABLE
BEGIN
    CG_IDP_FILE_CORRUPT    "Trouble reading the tips file"
    CG_IDS_TIPOFTHEDAYHELP "&Help"
END

```

MAINFRM.H (陰影部份為新增內容)

```

class CMainFrame : public CMDIFrameWnd
{
...
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL
...
// Generated message map functions
protected:
    afx_msg void OnInitMenu(CMenu* pMenu);
    ...
};

```

MAINFRM.CPP (陰影部份為新增內容)

```

#0001 ...
#0002 #include "Splash.h"
#0003 ...
#0004 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
#0005 {
#0006     ...
#0007     // CG: The following line was added by the Splash Screen component.
#0008     CSplashWnd::ShowSplashScreen(this);
#0009
#0010     return 0;
#0011 }
#0012 ...
#0013 //////////////////////////////////////
#0014 // CMainFrame message handlers
#0015

```

```
#0016 void CMainFrame::OnInitMenu(CMenu* pMenu)
#0017 {
#0018     CMDIFrameWnd::OnInitMenu(pMenu);
#0019
#0020     // CG: This block added by 'Tip of the Day' component.
#0021     {
#0022         // TODO: This code adds the "Tip of the Day" menu item
#0023         // on the fly. It may be removed after adding the menu
#0024         // item to all applicable menu items using the resource
#0025         // editor.
#0026
#0027         // Add Tip of the Day menu item on the fly!
#0028         static CMenu* pSubMenu = NULL;
#0029
#0030         CString strHelp; strHelp.LoadString(CG_IDS_TIPOFTHEDAYHELP);
#0031         CString strMenu;
#0032         int nMenuCount = pMenu->GetMenuItemCount();
#0033         BOOL bFound = FALSE;
#0034         for (int i=0; i < nMenuCount; i++)
#0035         {
#0036             pMenu->GetMenuString(i, strMenu, MF_BYPOSITION);
#0037             if (strMenu == strHelp)
#0038             {
#0039                 pSubMenu = pMenu->GetSubMenu(i);
#0040                 bFound = TRUE;
#0041                 ASSERT(pSubMenu != NULL);
#0042             }
#0043         }
#0044
#0045         CString strTipMenu;
#0046         strTipMenu.LoadString(CG_IDS_TIPOFTHEDAYMENU);
#0047         if (!bFound)
#0048         {
#0049             // Help menu is not available. Please add it!
#0050             if (pSubMenu == NULL)
#0051             {
#0052                 // The same pop-up menu is shared between mainfrm and
frame
#0053
#0054                 // with the doc.
#0055                 static CMenu popUpMenu;
#0056                 pSubMenu = &popUpMenu;
#0057                 pSubMenu->CreatePopupMenu();
#0058                 pSubMenu->InsertMenu(0, MF_STRING|MF_BYPOSITION,
CG_IDS_TIPOFTHEDAY, strTipMenu);
#0059             }
#0060             pMenu->AppendMenu(MF_STRING|MF_BYPOSITION|MF_ENABLED|MF_POPUP,
```

```
#0061         (UINT)pSubMenu->m_hMenu, strHelp);
#0062         DrawMenuBar();
#0063     }
#0064     else
#0065     {
#0066         // Check to see if the Tip of the Day menu has already been
added.
#0067         pSubMenu->GetMenuString(0, strMenu, MF_BYPOSITION);
#0068
#0069         if (strMenu != strTipMenu)
#0070         {
#0071             // Tip of the Day submenu has not been added to the
#0072             // first position, so add it.
#0073             pSubMenu->InsertMenu(0, MF_BYPOSITION); // Separator
#0074             pSubMenu->InsertMenu(0, MF_STRING|MF_BYPOSITION,
#0075                 CG_IDS_TIPOTHEEDAY, strTipMenu);
#0076         }
#0077     }
#0078 }
#0080 }
```

SPLASH.H (全新內容)

```
#0001 // CG: This file was added by the Splash Screen component.
#0002
#0003 #ifndef _SPLASH_SCRN_
#0004 #define _SPLASH_SCRN_
#0005
#0006 // Splash.h : header file
#0007
#0008 //////////////////////////////////////
#0009 //   Splash Screen class
#0010
#0011 class CSplashWnd : public CWnd
#0012 {
#0013 // Construction
#0014 protected:
#0015     CSplashWnd();
#0016
#0017 // Attributes:
#0018 public:
#0019     CBitmap m_bitmap;
#0020
#0021 // Operations
#0022 public:
#0023     static void EnableSplashScreen(BOOL bEnable = TRUE);
```

```

#0024         static void ShowSplashScreen(CWnd* pParentWnd = NULL);
#0025         static BOOL PreTranslateAppMessage(MSG* pMsg);
#0026
#0027 // Overrides
#0028         // ClassWizard generated virtual function overrides
#0029         //{AFX_VIRTUAL(CSplashWnd)
#0030         //{AFX_VIRTUAL
#0031
#0032 // Implementation
#0033 public:
#0034     ~CSplashWnd();
#0035     virtual void PostNcDestroy();
#0036
#0037 protected:
#0038     BOOL Create(CWnd* pParentWnd = NULL);
#0039     void HideSplashScreen();
#0040     static BOOL c_bShowSplashWnd;
#0041     static CSplashWnd* c_pSplashWnd;
#0042
#0043 // Generated message map functions
#0044 protected:
#0045     //{AFX_MSG(CSplashWnd)
#0046     afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
#0047     afx_msg void OnPaint();
#0048     afx_msg void OnTimer(UINT nIDEvent);
#0049     //{AFX_MSG
#0050     DECLARE_MESSAGE_MAP()
#0051 };
#0052
#0053 #endif

```

SPLASH.CPP (全新內容)

```

#0001 // CG: This file was added by the Splash Screen component.
#0002 // Splash.cpp : implementation file
#0003
#0004 #include "stdafx.h" // e. g. stdafx.h
#0005 #include "resource.h" // e.g. resource.h
#0006
#0007 #include "Splash.h" // e.g. splash.h
#0008
#0009 #ifdef _DEBUG
#0010 #define new DEBUG_NEW
#0011 #undef THIS_FILE
#0012 static char BASED_CODE THIS_FILE[] = __FILE__;
#0013 #endif

```



```

#0014
#0015 //////////////////////////////////////////////////
#0016 //  Splash Screen class
#0017
#0018 BOOL CSplashWnd::c_bShowSplashWnd;
#0019 CSplashWnd* CSplashWnd::c_pSplashWnd;
#0020 CSplashWnd::CSplashWnd()
#0021 {
#0022 }
#0023
#0024 CSplashWnd::~CSplashWnd()
#0025 {
#0026     // Clear the static window pointer.
#0027     ASSERT(c_pSplashWnd == this);
#0028     c_pSplashWnd = NULL;
#0029 }
#0030
#0031 BEGIN_MESSAGE_MAP(CSplashWnd, CWnd)
#0032     //{AFX_MSG_MAP(CSplashWnd)
#0033     ON_WM_CREATE()
#0034     ON_WM_PAINT()
#0035     ON_WM_TIMER()
#0036     //}AFX_MSG_MAP
#0037 END_MESSAGE_MAP()
#0038
#0039 void CSplashWnd::EnableSplashScreen(BOOL bEnable /*= TRUE*/)
#0040 {
#0041     c_bShowSplashWnd = bEnable;
#0042 }
#0043
#0044 void CSplashWnd::ShowSplashScreen(CWnd* pParentWnd /*= NULL*/)
#0045 {
#0046     if (!c_bShowSplashWnd || c_pSplashWnd != NULL)
#0047         return;
#0048
#0049     // Allocate a new splash screen, and create the window.
#0050     c_pSplashWnd = new CSplashWnd;
#0051     if (!c_pSplashWnd->Create(pParentWnd))
#0052         delete c_pSplashWnd;
#0053     else
#0054         c_pSplashWnd->UpdateWindow();
#0055 }
#0056
#0057 BOOL CSplashWnd::PreTranslateAppMessage(MSG* pMsg)
#0058 {
#0059     if (c_pSplashWnd == NULL)

```

```
#0060         return FALSE;
#0061
#0062         // If we get a keyboard or mouse message, hide the splash screen.
#0063         if (pMsg->message == WM_KEYDOWN ||
#0064             pMsg->message == WM_SYSKEYDOWN ||
#0065             pMsg->message == WM_LBUTTONDOWN ||
#0066             pMsg->message == WM_RBUTTONDOWN ||
#0067             pMsg->message == WM_MBUTTONDOWN ||
#0068             pMsg->message == WM_NCLBUTTONDOWN ||
#0069             pMsg->message == WM_NCRBUTTONDOWN ||
#0070             pMsg->message == WM_NCMBUTTONDOWN)
#0071         {
#0072             c_pSplashWnd->HideSplashScreen();
#0073             return TRUE;    // message handled here
#0074         }
#0075
#0076         return FALSE;    // message not handled
#0077     }
#0078
#0079     BOOL CSplashWnd::Create(CWnd* pParentWnd /*= NULL*/)
#0080     {
#0081         if (!m_bitmap.LoadBitmap(IDB_SPLASH))
#0082             return FALSE;
#0083
#0084         BITMAP bm;
#0085         m_bitmap.GetBitmap(&bm);
#0086
#0087         return CreateEx(0,
#0088             AfxRegisterWndClass(0, AfxGetApp()->LoadStandardCursor(IDC_ARROW)),
#0089             NULL, WS_POPUP | WS_VISIBLE, 0, 0, bm.bmWidth, bm.bmHeight,
#0090             pParentWnd->GetSafeHwnd(), NULL);
#0091     }
#0092     void CSplashWnd::HideSplashScreen()
#0093     {
#0094         // Destroy the window, and update the mainframe.
#0095         DestroyWindow();
#0096         AfxGetMainWnd()->UpdateWindow();
#0097     }
#0098
#0099     void CSplashWnd::PostNcDestroy()
#0100     {
#0101         // Free the C++ class.
#0102         delete this;
#0103     }
#0104
```

```
#0105 int CSplashWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
#0106 {
#0107     if (CWnd::OnCreate(lpCreateStruct) == -1)
#0108         return -1;
#0109
#0110     // Center the window.
#0111     CenterWindow();
#0112
#0113     // Set a timer to destroy the splash screen.
#0114     SetTimer(1, 750, NULL);
#0115
#0116     return 0;
#0117 }
#0118
#0119 void CSplashWnd::OnPaint()
#0120 {
#0121     CPaintDC dc(this);
#0122
#0123     CDC dcImage;
#0124     if (!dcImage.CreateCompatibleDC(&dc))
#0125         return;
#0126
#0127     BITMAP bm;
#0128     m_bitmap.GetBitmap(&bm);
#0129
#0130     // Paint the image.
#0131     CBitmap* pOldBitmap = dcImage.SelectObject(&m_bitmap);
#0132     dc.BitBlt(0, 0, bm.bmWidth, bm.bmHeight, &dcImage, 0, 0, SRCCOPY);
#0133     dcImage.SelectObject(pOldBitmap);
#0134 }
#0135
#0136 void CSplashWnd::OnTimer(UINT nIDEvent)
#0137 {
#0138     // Destroy the splash screen window.
#0139     HideSplashScreen();
#0140 }
```

TIPDLG.H (全新內容)

```
#0001 #if !defined(TIPDLG_H_INCLUDED_)
#0002 #define TIPDLG_H_INCLUDED_
#0003
#0004 // CG: This file added by 'Tip of the Day' component.
#0005
#0006 //////////////////////////////////////
#0007 // CTipDlg dialog
```

```

#0008
#0009 class CTipDlg : public CDialog
#0010 {
#0011 // Construction
#0012 public:
#0013     CTipDlg(CWnd* pParent = NULL); // standard constructor
#0014
#0015 // Dialog Data
#0016     //{AFX_DATA(CTipDlg)
#0017     // enum { IDD = IDD_TIP };
#0018     BOOL    m_bStartup;
#0019     CString m_strTip;
#0020     //}AFX_DATA
#0021
#0022     FILE* m_pStream;
#0023
#0024 // Overrides
#0025     // ClassWizard generated virtual function overrides
#0026     //{AFX_VIRTUAL(CTipDlg)
#0027     protected:
#0028     virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
#0029     //}AFX_VIRTUAL
#0030
#0031 // Implementation
#0032 public:
#0033     virtual ~CTipDlg();
#0034
#0035 protected:
#0036     // Generated message map functions
#0037     //{AFX_MSG(CTipDlg)
#0038     afx_msg void OnNextTip();
#0039     afx_msg HBRUSH OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor);
#0040     virtual void OnOK();
#0041     virtual BOOL OnInitDialog();
#0042     afx_msg void OnPaint();
#0043     //}AFX_MSG
#0044     DECLARE_MESSAGE_MAP()
#0045
#0046     void GetNextTipString(CString& strNext);
#0047 };
#0048
#0049 #endif // !defined(TIPDLG_H_INCLUDED_)

```

TIPDLG.CPP (全新內容)

```
#0001 #include "stdafx.h"
```

```
#0002 #include "resource.h"
#0003
#0004 // CG: This file added by 'Tip of the Day' component.
#0005
#0006 #include <winreg.h>
#0007 #include <sys\stat.h>
#0008 #include <sys\types.h>
#0009
#0010 #ifdef _DEBUG
#0011 #define new DEBUG_NEW
#0012 #undef THIS_FILE
#0013 static char THIS_FILE[] = __FILE__;
#0014 #endif
#0015
#0016 //////////////////////////////////////
#0017 // CTipDlg dialog
#0018
#0019 #define MAX_BUFLen 1000
#0020
#0021 static const TCHAR szSection[] = _T("Tip");
#0022 static const TCHAR szIntFilePos[] = _T("FilePos");
#0023 static const TCHAR szTimeStamp[] = _T("TimeStamp");
#0024 static const TCHAR szIntStartup[] = _T("Startup");
#0025
#0026 CTipDlg::CTipDlg(CWnd* pParent /*=NULL*/)
#0027 : CDialog(IDD_TIP, pParent)
#0028 {
#0029     //{AFX_DATA_INIT(CTipDlg)
#0030     m_bStartup = TRUE;
#0031     //}AFX_DATA_INIT
#0032
#0033     // We need to find out what the startup and file position parameters are
#0034     // If startup does not exist, we assume that the Tips on startup is checked TRUE.
#0035     CWinApp* pApp = AfxGetApp();
#0036     m_bStartup = !pApp->GetProfileInt(szSection, szIntStartup, 0);
#0037     UINT iFilePos = pApp->GetProfileInt(szSection, szIntFilePos, 0);
#0038
#0039     // Now try to open the tips file
#0040     m_pStream = fopen("tips.txt", "r");
#0041     if (m_pStream == NULL)
#0042     {
#0043         m_strTip.LoadString(CG_IDS_FILE_ABSENT);
#0044         return;
#0045     }
#0046
#0047     // If the timestamp in the INI file is different from the timestamp of
```

```
#0048 // the tips file, then we know that the tips file has been modified
#0049 // Reset the file position to 0 and write the latest timestamp to the
#0050 // ini file
#0051 struct _stat buf;
#0052 _fstat(_fileno(m_pStream), &buf);
#0053 CString strCurrentTime = ctime(&buf.st_ctime);
#0054 strCurrentTime.TrimRight();
#0055 CString strStoredTime =
#0056     pApp->GetProfileString(szSection, szTimeStamp, NULL);
#0057 if (strCurrentTime != strStoredTime)
#0058 {
#0059     iFilePos = 0;
#0060     pApp->WriteProfileString(szSection, szTimeStamp, strCurrentTime);
#0061 }
#0062
#0063 if (fseek(m_pStream, iFilePos, SEEK_SET) != 0)
#0064 {
#0065     AfxMessageBox(CG_IDP_FILE_CORRUPT);
#0066 }
#0067 else
#0068 {
#0069     GetNextTipString(m_strTip);
#0070 }
#0071 }
#0072
#0073 CTipDlg::~CTipDlg()
#0074 {
#0075     // This destructor is executed whether the user had pressed the escape key
#0076     // or clicked on the close button. If the user had pressed the escape key,
#0077     // it is still required to update the filepos in the ini file with the
#0078     // latest position so that we don't repeat the tips!
#0079
#0080     // But make sure the tips file existed in the first place....
#0081     if (m_pStream != NULL)
#0082     {
#0083         CWinApp* pApp = AfxGetApp();
#0084         pApp->WriteProfileInt(szSection, szIntFilePos, ftell(m_pStream));
#0085         fclose(m_pStream);
#0086     }
#0087 }
#0088
#0089 void CTipDlg::DoDataExchange(CDataExchange* pDX)
#0090 {
#0091     CDialog::DoDataExchange(pDX);
#0092     //{{AFX_DATA_MAP(CTipDlg)
#0093     DDX_Check(pDX, IDC_STARTUP, m_bStartup);
```

```
#0094     DDX_Text(pDX, IDC_TIPSTRING, m_strTip);
#0095     //{AFX_DATA_MAP
#0096 }
#0097
#0098 BEGIN_MESSAGE_MAP(CTipDlg, CDialog)
#0099     //{AFX_MSG_MAP(CTipDlg)
#0100     ON_BN_CLICKED(IDC_NEXTTIP, OnNextTip)
#0101     ON_WM_CTLCOLOR()
#0102     ON_WM_PAINT()
#0103     //{AFX_MSG_MAP
#0104 END_MESSAGE_MAP()
#0105
#0106 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#0107 // CTipDlg message handlers
#0108
#0109 void CTipDlg::OnNextTip()
#0110 {
#0111     GetNextTipString(m_strTip);
#0112     UpdateData(FALSE);
#0113 }
#0114
#0115 void CTipDlg::GetNextTipString(CString& strNext)
#0116 {
#0117     LPTSTR lpsz = strNext.GetBuffer(MAX_BUFLEN);
#0118
#0119     // This routine identifies the next string that needs to be
#0120     // read from the tips file
#0121     BOOL bStop = FALSE;
#0122     while (!bStop)
#0123     {
#0124         if (_fgetts(lpsz, MAX_BUFLEN, m_pStream) == NULL)
#0125         {
#0126             // We have either reached EOF or encountered some problem
#0127             // In both cases reset the pointer to the beginning of the file
#0128             // This behavior is same as VC++ Tips file
#0129             if (fseek(m_pStream, 0, SEEK_SET) != 0)
#0130                 AfxMessageBox(CG_IDP_FILE_CORRUPT);
#0131         }
#0132         else
#0133         {
#0134             if (*lpsz != ' ' && *lpsz != '\t' &&
#0135                 *lpsz != '\n' && *lpsz != ';')
#0136             {
#0137                 // There should be no space at the beginning of the tip
#0138                 // This behavior is same as VC++ Tips file
#0139                 // Comment lines are ignored and they start with a semicolon
```

```
#0140             bStop = TRUE;
#0141         }
#0142     }
#0143 }
#0144     strNext.ReleaseBuffer();
#0145 }
#0146
#0147 HBRUSH CTipDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
#0148 {
#0149     if (pWnd->GetDlgCtrlID() == IDC_TIPSTRING)
#0150         return (HBRUSH)GetStockObject(WHITE_BRUSH);
#0151
#0152     return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
#0153 }
#0154
#0155 void CTipDlg::OnOK()
#0156 {
#0157     CDialog::OnOK();
#0158
#0159     // Update the startup information stored in the INI file
#0160     CWinApp* pApp = AfxGetApp();
#0161     pApp->WriteProfileInt(szSection, szIntStartup, !m_bStartup);
#0162 }
#0163
#0164 BOOL CTipDlg::OnInitDialog()
#0165 {
#0166     CDialog::OnInitDialog();
#0167
#0168     // If Tips file does not exist then disable NextTip
#0169     if (m_pStream == NULL)
#0170         GetDlgItem(IDC_NEXTTIP)->EnableWindow(FALSE);
#0171
#0172     return TRUE; // return TRUE unless you set the focus to a control
#0173 }
#0174
#0175 void CTipDlg::OnPaint()
#0176 {
#0177     CPaintDC dc(this); // device context for painting
#0178
#0179     // Get paint area for the big static control
#0180     CWnd* pStatic = GetDlgItem(IDC_BULB);
#0181     CRect rect;
#0182     pStatic->GetWindowRect(&rect);
#0183     ScreenToClient(&rect);
#0184
#0185     // Paint the background white.
```



```
#0186     CBrush brush;
#0187     brush.CreateStockObject(WHITE_BRUSH);
#0188     dc.FillRect(rect, &brush);
#0189
#0190     // Load bitmap and get dimensions of the bitmap
#0191     CBitmap bmp;
#0192     bmp.LoadBitmap(IDB_LIGHTBULB);
#0193     BITMAP bmpInfo;
#0194     bmp.GetBitmap(&bmpInfo);
#0195
#0196     // Draw bitmap in top corner and validate only top portion of window
#0197     CDC dcTmp;
#0198     dcTmp.CreateCompatibleDC(&dc);
#0199     dcTmp.SelectObject(&bmp);
#0200     rect.bottom = bmpInfo.bmHeight + rect.top;
#0201     dc.BitBlt(rect.left, rect.top, rect.Width(), rect.Height(),
#0202             &dcTmp, 0, 0, SRCCOPY);
#0203
#0204     // Draw out "Did you know..." message next to the bitmap
#0205     CString strMessage;
#0206     strMessage.LoadString(CG_IDS_DIDYOUKNOW);
#0207     rect.left += bmpInfo.bmWidth;
#0208     dc.DrawText(strMessage, rect, DT_VCENTER | DT_SINGLELINE);
#0209
#0210     // Do not call CDialog::OnPaint() for painting messages
#0211 }
```

修改 ComTest 程式內容

以下是對於上述新增檔案的分析與修改。稍早我曾分析過，只要修改一下 **Splash Screen** 畫面，增加一個 TIPS.TXT 文字檔，再變化一下 About 對話窗，就成了。

COMTEST.RC

要把自己準備的圖片做為「炫耀畫面」，有兩個還算方便的作法。其一是直接編修 **Splash Screen** 元件帶給我們的 **Splsh16.bmp** 的內容，其二是修改 RC 檔中的 **IDB_SPLASH** 所對應的檔案名稱。我選擇後者。所以我修改 RC 檔中的一行：

```
IDB_SPLASH BITMAP DISCARDABLE "Dissect.bmp"
```

Dissect.bmp 圖檔內容如下：



此外我也修改 RC 檔中的一些字串，使它們呈現中文：

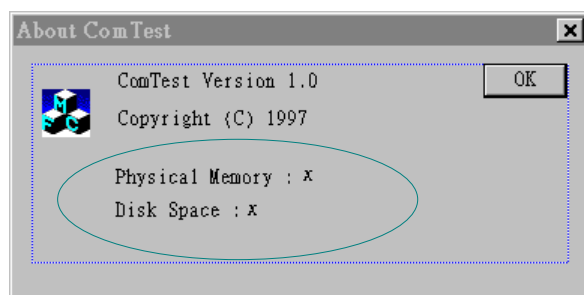
```
IDD_TIP_DIALOG DISCARDABLE 0, 0, 231, 164
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "今日小秘訣"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL        "", -1, "Static", SS_BLACKFRAME, 12, 11, 207, 123
    LTEXT           "Some String", IDC_TIPSTRING, 28, 63, 177, 60
    CONTROL         "程式啟動時顯示小秘訣", IDC_STARTUP, "Button",
        BS_AUTOCHECKBOX | WS_GROUP | WS_TABSTOP, 13, 146, 85, 10
    PUSHBUTTON      "下一個小秘訣", IDC_NEXTTIP, 109, 143, 50, 14, WS_GROUP
    DEFPUSHBUTTON   "關閉", IDOK, 168, 143, 50, 14, WS_GROUP
    CONTROL         "", IDC_BULB, "Static", SS_BITMAP, 20, 17, 190, 111
END

STRINGTABLE DISCARDABLE
BEGIN
    ...
    // CG_IDS_DIDYOUKNOW        "Did You Know..."
    CG_IDS_DIDYOUKNOW        "侯俊傑著作年表..."
END
```

增加一個 TIPS.TXT

這很簡單，使用任何一種文字編輯工具，遵循前面說過的 TIPS.TXT 檔案格式，做出你的每日小秘訣。

修改 RC 檔中的 About 對話窗畫面



我增加了四個 static 控制元件，其中兩個做為標籤使用，不必在乎其 ID。另兩個準備給 ComTest 程式在【About】對話窗出現時設定系統資訊使用，ID 分別設定為 *IDC_PHYSICAL_MEM* 和 *IDC_DISK_SPACE*，配合 **System Info for About Dlg** 元件的建議。

COMTEST.CPP

在 *CAboutDlg::OnInitDialog* 中利用 *SetDlgItemText* 設定稍早我們為對話窗畫面新增的兩個 static 控制元件的文字內容（Component Gallery 已經為我們做出這段程式碼，只是暫時把它標記為說明文字。我只要把標記符號 // 去除即可）：

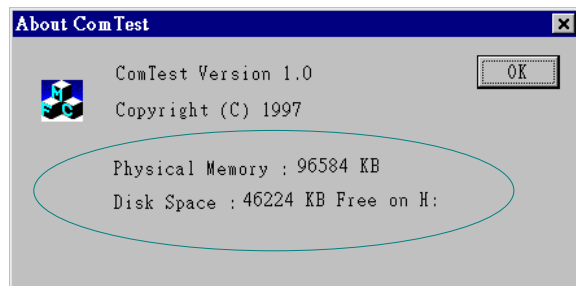
```
BOOL CAboutDlg::OnInitDialog()  
{  
    ...  
    SetDlgItemText(IDC_PHYSICAL_MEM, strFreeMemory);  
    ...  
    SetDlgItemText(IDC_DISK_SPACE, strFreeDiskSpace);  
}  
return TRUE;    // CG: This was added by System Info Component.  
}
```

ComTest 修改結果

一切盡如人意。現在我們有了理想的 **Splash Screen** 畫面如前所述，也有了 **Tips of the Day** 對話窗：



以及一個內含系統資訊的 **About** 對話窗：



使用 ActiveX Controls

Microsoft 的 Visual Basic 自 1991 年推出以來，已經成為 Windows 應用軟體開發環境中的佼佼者。它的成功極大部份要歸功於其開放性質：它所提供的 VBXs 被認為是一種極佳的物件導向程式設計架構。VBX 是一種動態聯結函式庫 (DLL)，類似 Windows 的訂製型控制元件 (custom control)。

VBX 不適用於 32 位元環境。於是 Microsoft 再推出另一規格 OCX。不論是 VBX 或 OCX，或甚至 Borland 的 VCL，都提供 Properties-Method-Event (PME) 介面。Visual Basic 之於 VBX，以及 Borland C++ Builder 和 Delphi 之於 VCL，都提供了整合開發環境 (IDE) 與 PME 介面之間的極密切結合，使得程式設計更進一步到達「以拖拉、填單等簡易動作就能夠完成」的視覺化境界。也因此沒有人會反對把 Visual Basic 和 Delphi 和 C++ Builder 歸類為 RAD (Rapid Application Development，快速軟體開發工具) 的行列。但是 Visual C++ 之於 OCX，還沒能夠有這麼好的整合。

我怎麼會談到 OCX 呢？本節不是 ActiveX Control 嗎？噢，OCX 就是 ActiveX Control！由於微軟把它所有的 Internet 技術都稱為 ActiveX，所以 OLE Controls 就變成了 ActiveX Controls。

我不打算討論 ActiveX Control 的撰寫，我打算把全部篇幅用到 ActiveX Control 的使用上。

如果對 ActiveX Control 的開發感興趣，Adam Denning 的 *ActiveX Control Inside Out* 是一本很不錯的書 (ActiveX 控制元件徹底研究，侯俊傑譯 / 松崗)

ActiveX Control 基礎觀念：Properties、Methods、Events

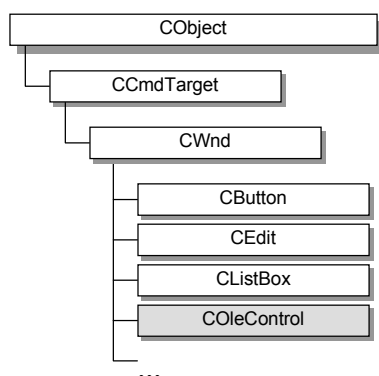
你必須了解 ActiveX Control 三種介面的意義，並且充份了解你打算使用的某個 ActiveX Control 有些什麼特殊的介面，然後才能夠使用它。

基本上你可以拿你已經很熟悉的 C++ 類別來比較 ActiveX control。類別也是一個包裝良好的元件，有它自己的成員變數，以及處理這些成員變數的所謂成員函式，是個自給自足的體系。ActiveX control 的三個介面也有類似性質：

- property - 相當於 C++ 類別的成員變數
- method - 相當於 C++ 類別的成員函式
- event - 相當於 Windows 控制元件發出的 notification 訊息

ActiveX Control 規格中定有一些標準的（庫存的）介面，例如 *BackColor* 和 *FontName* 等 properties，*AddItem* 和 *Move* 和 *Refresh* 等 methods，以及 *CLICK* 和 *KEYDOWN* 等 events。也就是說，任何一個 ActiveX Control 大致上都會有一些必備的、基礎的性質和能力。

以下針對 ActiveX Control 的三種介面與 C++ 類別做個比較。至於它們的具體展現以及如何使用，稍後在實例中可以看到。



methods

設計自己的 C++ 類別，你當然可以在其中設計成員函式。此一函式之呼叫者必須在編譯時期知道這一函式的功能以及它的參數。搭配 Windows 內建之控制元件（如 *Edit*、*Button*）而設計的類別（如 *CEdit*、*CButton*），內部固定會設計一些成員函式。某些成員函式（如 *CEdit::GetLineCount*）只適用於特定類別，但某些根類別的成員函式（例如 *CWnd::GetDlgItemText*）則適用於所有的子類別。

ActiveX Control 的 method 極類似 C++ 類別中的成員函式。但它們被限制在一個有限的集合之中，集合內的名單包括 *AddItem*、*RemoveItem*、*Move* 和 *Refresh* 等等。並不是所有的 ActiveX Controls 都對每一個 method 產生反應，例如 *Move* 就不能夠在每一個 ActiveX Control 中運作自如。

properties

基本上 properties 用來表達 ActiveX Control 的屬性或資料。一個名為 *Date* 的元件可能會定義一個所謂的 *DateValue*，內放日期，這就表現了元件的資料。它還可能定義一個所謂的 *DateFormat*，允許使用者取得或設定日期表現形式，這就表現了元件的屬性。

你可以說 ActiveX Control 的 properties 相當於 C++ 類別的成員變數。每一個 ActiveX Control 可以定義屬於它自己的 properties，可以是一個字串，可以是一個長整數，也可以是一個浮點數。有一組所謂的 properties 標準集合（被稱為 *stock properties*），內含 *BackColor*、*FontName*、*Caption* 等等 properties，是每個 ActiveX control 都會擁有的。

一般而言 properties 可分為四種型態：

- Ambient properties
- Extended properties
- Stock properties
- Custom properties

events

Windows 控制元件以所謂的 *notification*（通告）訊息送給其父視窗（通常是對話窗），例如按鈕元件可能傳送出一個 *BN_CLICKED*。ActiveX Control 使用完全相同的方法，不過現在 *notification* 訊息被稱為 *event*，用來表示某種狀況發生了。Events 的發射可以使 ActiveX Control 有能力通知其宿主（container，也就是 VB 或 VC 程式），於是對方有機會處理。大部份 ActiveX Controls 送出標準的 events，例如 *CLICK*、*KEYDOWN*、*KEYUP* 等等，某些 ActiveX Controls 會送出獨一無二的訊息（例如 *ROWCOLCHANGE*）。

一般而言 events 可分為兩種型態：

- Stock events
- Custom events

ActiveX Controls 的 1 大使用步驟

欲在程式中加上 ActiveX Controls，基本上需要五個步驟：

1. 建立新專案時，在 AppWizard 的步驟 3 中選擇【ActiveX Controls】。這會使程式碼多出一行：

```
BOOL COcxTestApp::InitInstance()
{
    AfxEnableControlContainer();
    ...
}
```

2. 進入 Component Gallery，把 ActiveX Controls 安插到你的程式中。
3. 使用 ActiveX Controls。通常我們在對話窗中使用它。我們可以把資源編輯器的工具箱裡頭的 ActiveX Controls 拖放到目標對話窗中。
4. 利用 ClassWizard 產生對話窗類別，並處理相關的 Message Maps、訊息處理常式、變數定義、對話盒函式等等。
5. 編譯聯結。

我將以系統內建（已註冊過）的 **Grid ActiveX Control** 做為示範的對象。**Grid** 具有小型試算表能力，當然它遠比不上 Excel（不然 Excel 怎麼賣），不過你至少可以獲得一個中規中矩的 7x14 試算表，並且有基本的編輯和運算功能。

容我先解釋我的目標。**圖 16-1** 是我期望的結果，這個試算表完全為了家庭記賬而量身設計，假設你有五種收入（真讓人羨慕），這個表格可以讓你登錄每個月的每一種收入，並計算月總收入和年總收入，以及各分項總收入。

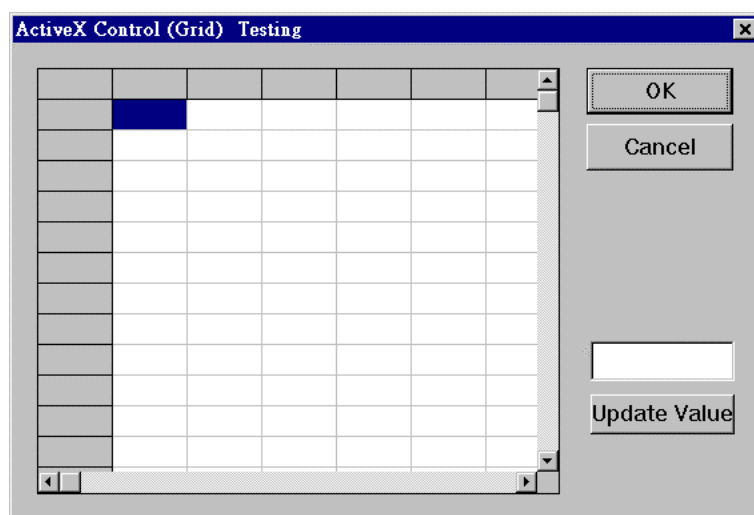


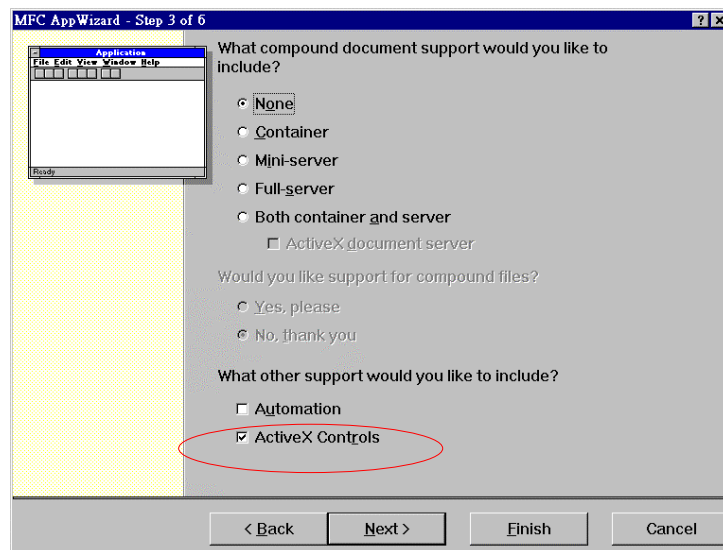
圖 16-1 在對話窗中使用 **Grid ActiveX control**。每一橫列或縱行的最後一欄都是總和。

由於 **Grid** 本身並不提供編輯能力，我們以試算表右側的一個 **edit** 欄位做為編輯區域。使用者所選擇的方格的内容會顯示在這 **edit** 欄位中，並且允許被編輯内容。數值填入後必須按下 <Enter> 鍵，或是在【**Update Value**】鈕上按一下，試算表内容才會更新。如果要直接在試算表欄位上做編輯動作，並不是不可以，把 **edit** 不偏不倚貼到欄位也就是了！

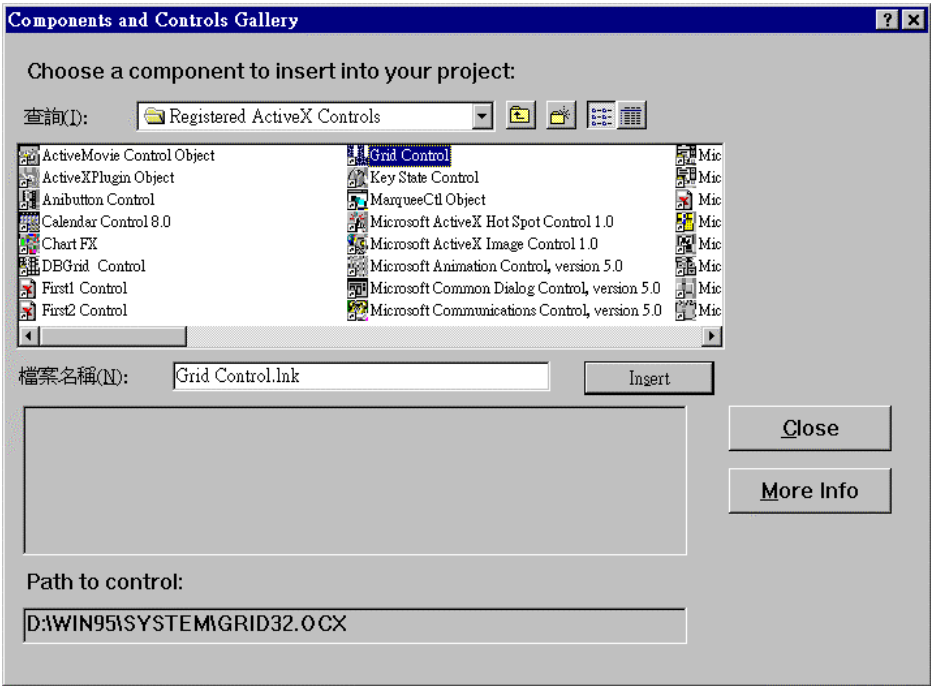
本書進行到這裡，我想你對於工具的使用應該已經嫺熟了，我將假設你對於像「利用 ClassWizard 為 *CMainFrame* 攔截一個 *ID_GridTest* 命令，並指名其處理常式為 *OnGridTest*」這樣的敘述，知道該怎麼去動手。

使用 Grid ActiveX Control：OcxTest 程式

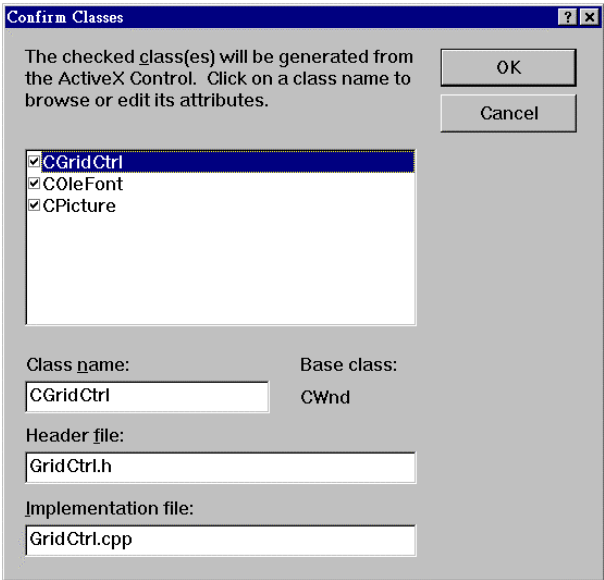
首先利用 MFC AppWizard 做出一個 OcxTest 專案。記得在步驟 3 選擇【ActiveX Controls】：



然後進入 Component Gallery，將 **Grid** 安插到專案中：



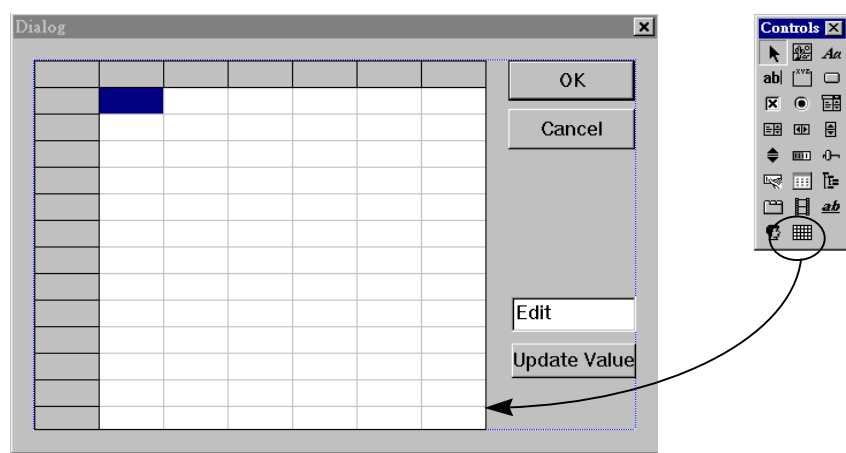
你必須回答一個對話窗：



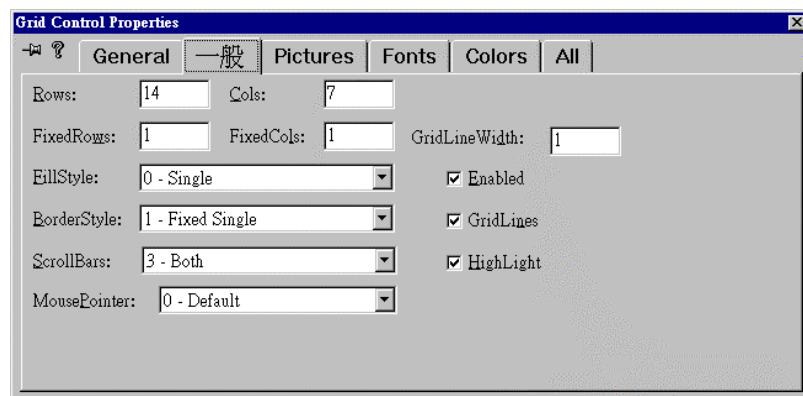
對話窗的設計

產生一個嶄新的對話窗。這個動作與你在第 10 章為 Scribble 加上 "Pen Width" 對話窗的步驟完全一樣。請把新對話窗的 ID 從 `IDD_DIALOG1` 改變為 `IDD_GRID`。

從工具箱中抓出控制元件來，把對話窗佈置如下。



雖然你把 Grid 拉大，它卻總是只有 2x2 個方格。你必須使用右鍵把它的 Control Properties 引出來（如下），進入 Control 附頁，這時候會出現各個 properties：



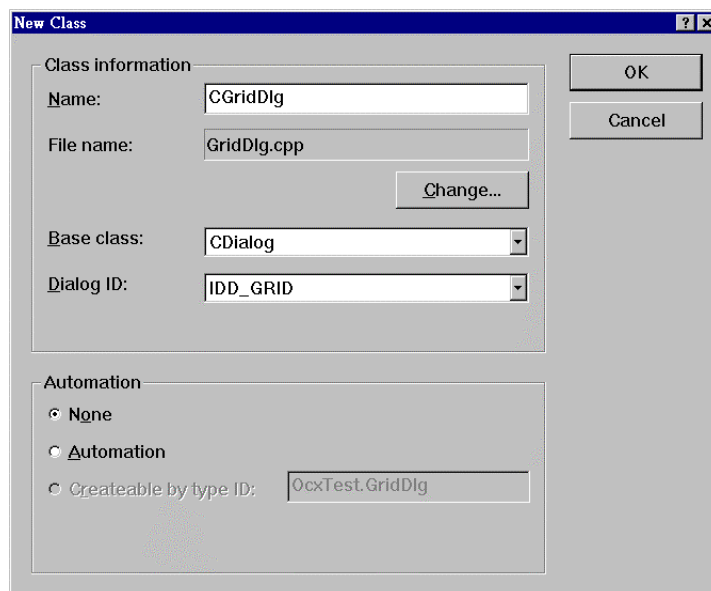
Grid Control 附頁在中文 Windows 中竟然變成「一般」。這是否也算是一隻臭蟲？

現在選擇 *Rows*，設定為 14，再選擇 *Cols*，設定為 7。你還可以設定行的寬度和列的高度，以及方格初值...。噢，記得給這個 **Grid** 元件一個 ID，叫做 *IDC_GRID* 好了。

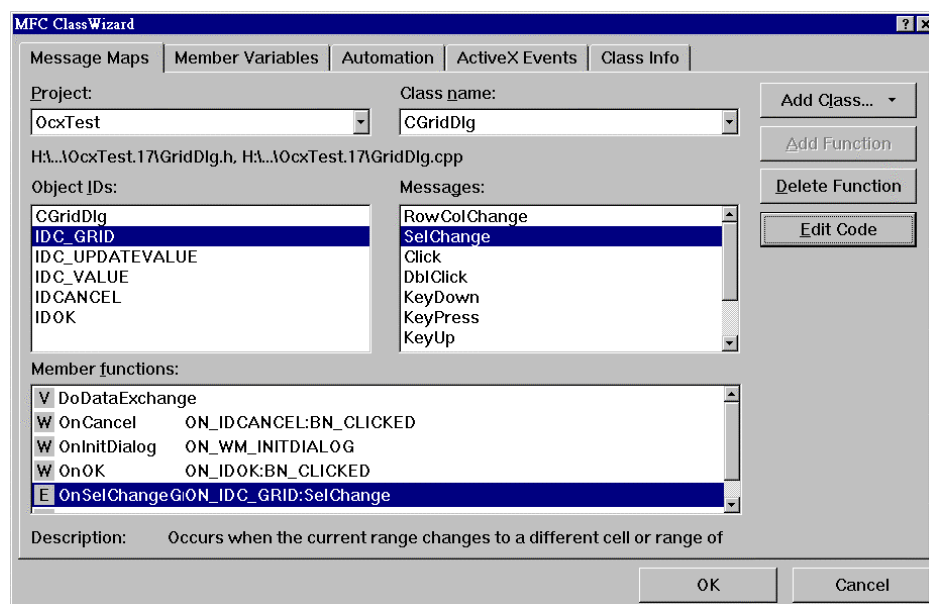
整個對話窗的設計規格如下：

物件	ID	文字內容
對話窗	IDD_GRID	ActiveX Control (Grid) Testing
OK 按鈕	IDOK	OK
Cancel 按鈕	IDCANCEL	Cancel
Edit	IDC_VALUE	
Update Value 按鈕	IDC_UPDATEVALUE	Update Value
Grid	IDC_GRID	

現在準備設計 *IDD_GRID* 的對話窗類別。這件事我們在第 10 章也做過。進入 CClassWizard，填寫【Add Class】對話窗如下，然後按下【OK】鈕：



回到 ClassWizard 主畫面，準備為元件們設計訊息處理常式。步驟是先選擇一個元件 ID，再選擇一個訊息，然後按下【Add Function】鈕。注意，如果你選擇到一個 ActiveX Control，"Messages" 清單中列出的就是該元件所能發出的 events。



本例的訊息處理常式的設計規格如下：

物件 ID	訊息	處理函式名稱
CGridDlg	WM_INITDIALOG	OnInitDialog
IDOK	BN_CLICK	OnOk
IDCANCEL	BN_CLICK	OnCancel
IDC_VALUE		
IDC_UPDATEVALUE	BN_CLICK	OnUpdatevalue
IDC_GRID	VBN_SELCHANGE	OnSelchangeGrid

到此為止，我們獲得這些新檔案：

```
RESOURCE.H
OCXTEST.RC
GRIDCTRL.H      <-- 本例不處理這個檔案
GRIDCTRL.CPP    <-- 本例不處理這個檔案
FONT.H          <-- 本例不處理這個檔案
FONT.CPP        <-- 本例不處理這個檔案
PICTURE.H       <-- 本例不處理這個檔案
PICTURE.CPP     <-- 本例不處理這個檔案
GRIDDLG.H       <-- 本例主要的修改對象
GRIDDLG.CPP     <-- 本例主要的修改對象
```

其中重要的相關程式碼我特別挑出來做個認識：

OCXTEST.RC

```
IDD_GRID_DIALOG DISCARDABLE 0, 0, 224, 142
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "ActiveX Control (Grid) Testing"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON    "OK",IDOK,172,7,44,14
    PUSHBUTTON       "Cancel",IDCANCEL,172,24,44,14
    CONTROL           " ",IDC_GRID,"{A8C3B720-0B5A-101B-B22E-00AA0037B2FC}",
                     WS_TABSTOP,7,7,157,128
    PUSHBUTTON       "Update Value",IDC_UPDATEVALUE,173,105,43,12
    EDITTEXT         IDC_VALUE,173,89,43,12,ES_AUTOHSCROLL
END
```

GRIDDLG.H

```
class CGridDlg : public CDialog
{
...
// Implementation
protected:

    // Generated message map functions
   //{{AFX_MSG(CGridDlg)
    virtual BOOL OnInitDialog();
    virtual void OnOK();
    virtual void OnCancel();
    afx_msg void OnUpdatevalue();
```

```

afx_msg void OnSelChangeGrid();
DECLARE_EVENTSINK_MAP()
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

```

GRIDDLG.CPP

```

BEGIN_MESSAGE_MAP(CGridDlg, CDialog)
   //{{AFX_MSG_MAP(CGridDlg)
    ON_BN_CLICKED(IDC_UPDATEVALUE, OnUpdatevalue)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
////
// CGridDlg message handlers

BEGIN_EVENTSINK_MAP(CGridDlg, CDialog)
   //{{AFX_EVENTSINK_MAP(CGridDlg)
    ON_EVENT(CGridDlg, IDC_GRID, 2 /* SelChange */, OnSelChangeGrid, VTS_NONE)
   //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

BOOL CGridDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here

    return TRUE; // return TRUE unless you set the focus to a control
                // EXCEPTION: OCX Property Pages should return FALSE
}

void CGridDlg::OnOK()
{
    // TODO: Add extra validation here

    CDialog::OnOK();
}

void CGridDlg::OnCancel()
{
    // TODO: Add extra cleanup here

    CDialog::OnCancel();
}

```



```

}

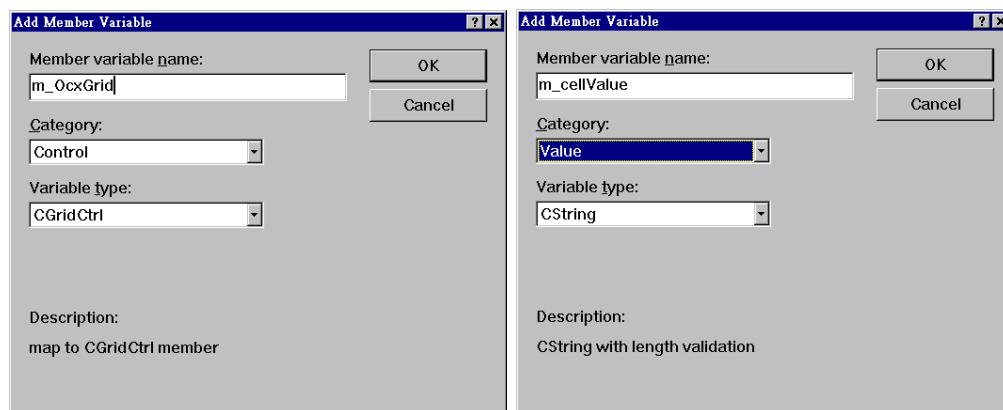
void CGridDlg::OnUpdatevalue()
{
    // TODO: Add your control notification handler code here
}

void CGridDlg::OnSelChangeGrid()
{
    // TODO: Add your control notification handler code here
}

```

對話盒加上一些變數

進入 ClassWizard，進入【Member Variables】附頁，選按其中的【Add Variable】鈕，為 OcxTest 加上兩筆成員變數。其中一筆用來儲存目前被選中的試算表方格內容，另一筆資料用來做為 **Grid** 物件，其變數型態是 *CGridCtrl*：



這兩個動作為我們帶來這樣的程式碼：

GRIDDLG.H

```

class CGridDlg : public CDialog
{
// Dialog Data
   //{{AFX_DATA(CGridDlg)
    enum { IDD = IDD_GRID };

```

```

CGridCtrl  m_OcxGrid;
CString    m_cellValue;
//}}AFX_DATA

...
};

```

GRIDDLG.CPP

```

CGridDlg::CGridDlg(CWnd* pParent /*=NULL*/)
: CDialog(CGridDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CGridDlg)
    m_cellValue = _T("");
    //}}AFX_DATA_INIT
}

void CGridDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CGridDlg)
    DDX_Control(pDX, IDC_GRID, m_OcxGrid);
    DDX_Text(pDX, IDC_VALUE, m_cellValue);
    //}}AFX_DATA_MAP
}

```

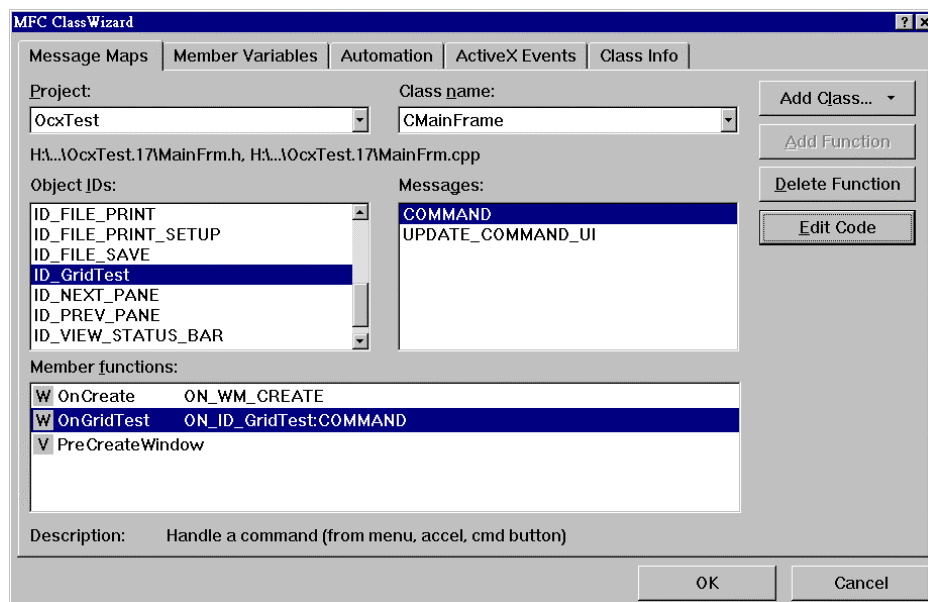
新增一個菜單項目

利用資源編輯器，將菜單修改如下：



注意，我所改變的菜單是 *IDR_MAINFRAME*，這是在沒有任何子視窗存在時才會出現的菜單。所以如果你要執行 *OcxTest* 並看到 **Grid** 元件，你必須先將所有的子視窗關閉。

現在利用 *ClassWizard* 在主視窗的訊息映射表中攔截它的命令訊息：



獲得對應的程式碼如下：

MAINFRM.H

```
class CMainFrame : public CMDIFrameWnd
{
...
protected:
    //{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnGridTest();
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

MAINFRM.CPP

```
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    ON_COMMAND(ID_GridTest, OnGridTest)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
void CMainFrame::OnGridTest()
{
    // TODO:
}
```

爲了讓這個新增選單命令真正發揮效用，將 Grid 對話窗喚起，我在 *OnGridTest* 函式加兩行：

```
#include "GridDlg.h"
...
void CMainFrame::OnGridTest()
{
    CGridDlg dlg;           // constructs the dialog
    dlg.DoModal();          // starts the dialog
}
```

現在，將 *OcxTest* 編譯連結一遍，得到一個可以順利執行的程式，但 **Grid** 之中全無內容。

Grid 相關程式設計

現在我要開始設計 **Grid** 相關函式。我的主要的工作是：

- 準備一個二維（7x14）的 **DWORD** 陣列，用來儲存 **Grid** 的方格內容。
- 程式初始化時就把二維陣列的初值設定好（本例不進行檔案讀寫），並產生 **Grid** 對話盒。
- 對話盒一出現，程式立刻把試算表的行、列、寬、高，以及欄位名稱都設定好，並且把二維陣列的數值放到對應方格中。初值的總和也一併計算出來。
- 把計算每一列每一行總和的工作獨立出來，成立一個 *ComputeSums* 函式。

爲了放置試算表內容，必須設計一個 7x14 二維陣列。雖然試算表中某些方格（如列標題或行標題）不必有內容，不過爲求簡化，還是完全配合試算表的大小來設計數值陣列好了。注意，不能把這個變數放在 *AFX_DATA* 之內，因爲我並非以 *ClassWizard* 加入此變數。

GRIDDLG.H

```
#define MAXCOL 7
#define MAXROW 14

class CGridDlg : public CDialog
{
...
// Dialog Data
    double m_dArray[MAXCOL][MAXROW];

private:
    void ComputeSums();
};
```

爲了設定 **Grid** 中的表頭以及初值，我在 *OnInitDialog* 中先以一個 **for loop** 設定橫列表頭再以一個 **for loop** 設定縱行表頭，最後再以巢狀（兩層）**for loop** 設定每一個方格內容，然後才呼叫 *ComputeSums* 計算總和。

當使用者選擇一個方格，其值就被 *OnSelchangeGrid* 拷貝一份到 **edit** 欄位中，這時候就可以開始輸入了。

OnUpdatevalue（【Update Value】按鈕的處理常式）有兩個主要任務，一是把 **edit** 欄位內容轉化爲數值放到目前被選擇的方格上，一是修正總和。

OnOk 必須能夠把每一個方格內容（一個字串）取出，利用 *atof* 轉換爲數值，然後儲存到 *m_dArray* 二維陣列中。

GRIDDLG.CPP

```
#0001
#0002 BOOL CGridDlg::OnInitDialog()
#0003 {
#0004     CString str;
#0005     int i, j;
```

```
#0006     CRect rect;
#0007
#0008     CDialog::OnInitDialog();
#0009
#0010     VERIFY(m_OcxGrid.GetCols() == (long)MAXCOL);
#0011     VERIFY(m_OcxGrid.GetRows() == (long)MAXROW);
#0012
#0013     m_OcxGrid.SetRow(0);           // #0 Row
#0014     for (i = 0; i < MAXCOL; i++) { // 所有的 Cols
#0015         if (i) { // column headings
#0016             m_OcxGrid.SetCol(i);
#0017             if (i == (MAXCOL-1))
#0018                 m_OcxGrid.SetText(CString("Total"));
#0019             else
#0020                 m_OcxGrid.SetText(CString('A' + i - 1));
#0021         }
#0022     }
#0023
#0024     m_OcxGrid.SetCol(0);           // #0 Col
#0025     for (j = 0; j < MAXROW; j++) { // 所有的 Rows
#0026         if (j) { // row headings
#0027             m_OcxGrid.SetRow(j);
#0028             if (j == (MAXROW-1))
#0029                 m_OcxGrid.SetText(CString("Total"));
#0030             else {
#0031                 str.Format("%d", j);
#0032                 m_OcxGrid.SetText(str);
#0033             }
#0034         }
#0035     }
#0036
#0037     // sets the spreadsheet values from m_dArray
#0038     for (i = 1; i < (MAXCOL-1); i++) {
#0039         m_OcxGrid.SetCol(i);
#0040         for (j = 1; j < (MAXROW-1); j++) {
#0041             m_OcxGrid.SetRow(j);
#0042             str.Format("%8.2f", m_dArray[i][j]);
#0043             m_OcxGrid.SetText(str);
#0044         }
#0045     }
#0046
#0047     ComputeSums();
#0048
#0049     // be sure there's a selected cell
#0050     m_OcxGrid.SetCol(1);
#0051     m_OcxGrid.SetRow(1);
```

```
#0052     m_cellValue = m_OcxGrid.GetText();
#0053     UpdateData(FALSE); // calls DoDataExchange to update edit control
#0054     return TRUE;
#0055 }
#0056
#0057 void CGridDlg::OnOK()
#0058 {
#0059     int i, j;
#0060
#0061     for (i = 1; i < (MAXCOL-1); i++) {
#0062         m_OcxGrid.SetCol(i);
#0063         for (j = 1; j < (MAXROW-1); j++) {
#0064             m_OcxGrid.SetRow(j);
#0065             m_dArray[i][j] = atof(m_OcxGrid.GetText());
#0066         }
#0067     }
#0068     CDialog::OnOK();
#0069 }
#0070
#0071 void CGridDlg::OnUpdatevalue()
#0072 {
#0073     CString str;
#0074     double value;
#0075     // LONG   lRow, lCol;
#0076     int   Row, Col;
#0077
#0078     if (m_OcxGrid.GetCellSelected() == 0) {
#0079         AfxMessageBox("No cell selected");
#0080         return;
#0081     }
#0082
#0083     UpdateData(TRUE);
#0084     value = atof(m_cellValue);
#0085     str.Format("%8.2f", value);
#0086
#0087     // saves current cell selection
#0088     Col = m_OcxGrid.GetCol();
#0089     Row = m_OcxGrid.GetRow();
#0090
#0091     m_OcxGrid.SetText(str); // copies new value to
#0092                             // the selected cell
#0093     ComputeSums();
#0094
#0095     // restores current cell selection
#0096     m_OcxGrid.SetCol(Col);
#0097     m_OcxGrid.SetRow(Row);
```

```
#0098 }
#0099
#0100 void CGridDlg::OnSelChangeGrid()
#0101 {
#0102     if (m_OcxGrid) {
#0103         m_cellValue = m_OcxGrid.GetText();
#0104         UpdateData(FALSE); // calls DoDataExchange to update edit
control
#0105         GotoDlgCtrl(GetDlgItem(IDC_VALUE)); // position edit control
#0106     }
#0107 }
#0108
#0109 void CGridDlg::ComputeSums()
#0110 {
#0111     int    i, j, nRows, nCols;
#0112     double sum;
#0113     CString str;
#0114
#0115     // adds up each row and puts the sum in the right col
#0116     // col count could have been changed by add row/delete row
#0117     nCols = (int) m_OcxGrid.GetCols();
#0118     for (j = 1; j < (MAXROW-1); j++) {
#0119         m_OcxGrid.SetRow(j);
#0120         sum = 0.0;
#0121         for (i = 1; i < nCols - 1; i++) {
#0122             m_OcxGrid.SetCol(i);
#0123             sum += atof(m_OcxGrid.GetText());
#0124         }
#0125         str.Format("%8.2f", sum);
#0126         m_OcxGrid.SetCol(nCols - 1);
#0127         m_OcxGrid.SetText(str);
#0128     }
#0129
#0130     // adds up each column and puts the sum in the bottom row
#0131     // row count could have been changed by add row/delete row
#0132     nRows = (int) m_OcxGrid.GetRows();
#0133     for (i = 1; i < MAXCOL; i++) {
#0134         m_OcxGrid.SetCol(i);
#0135         sum = 0.0;
#0136         for (j = 1; j < nRows - 1; j++) {
#0137             m_OcxGrid.SetRow(j);
#0138             sum += atof(m_OcxGrid.GetText());
#0139         }
#0140         str.Format("%8.2f", sum);
#0141         m_OcxGrid.SetRow(nRows - 1);
#0142         m_OcxGrid.SetText(str);
```



```
#0143     }  
#0144 }
```

下圖是 OcxTest 的執行畫面。

