《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多許多的大陸讀者來函。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。我已爲《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爲簡體。這已是我能爲各位盡力的極限。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爲您解決。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爲，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爲者，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爲，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。爲計久長，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

淺出 MFC 程式設計

第５章

總觀 Application Framework

帶藝術氣息的軟體創作行爲將在 Application Framework 出現後逐漸成爲工匠技術，
而我們都將成爲軟體 IC 裝配廠裡的男工女工。
但，不是亨利福特，我們又如何能夠享受大眾化的汽車？
或許以後會出現「純手工精製」的軟體，可我自己從來不嫌機器饅頭難吃。

什麼是 Application Framework？

還沒有學習任何一套 Application Framework 的使用之前，就給你近乎學術性的定義，我可以想像對你而言絕對是「形而上的」（超物質的無形哲理），尤其如果你對物件導向（Object Oriented）也還沒有深刻體會的話。形而上者謂之道，形而下者謂之器，我想能夠捨器而直接近道者，幾稀！但是，「定義」這種東西又似乎宜開宗明義擺在前頭。我誠摯地希望你在閱讀後續的技術章節時能夠時而回來看看這些形而上的敘述。當你有所感受，技術面應該也進入某個層次了。

侯捷怎麼說

首先我們看看侯捷在其無責任書評中是怎麼說的：

> 演化（evolution）永遠在進行，但這個世界卻不是每天都有革命性（revolution）的事物發生。動不動宣稱自己（或自己的產品）是劃時代的革命性的，帶來的影響就像時下滿街跑的大哥一樣使我們漸漸無動於衷（大哥不可能滿街跑）！但是 Application Framework 的的確確在我們軟體界稱得上具有革命精神。
>
> 什麼是 Application Framework？Framework 這個字眼有組織、框架、體制的意思，Application Framework 不僅是一般性的泛稱，它其實還是物件導向領域中的一個專有名詞。
>
> 基本上你可以說，Application Framework 是一個完整的程式模型，具備標準應用軟體所需的一切基本功能，像是檔案存取、列印預視、資料交換...，以及這些功能的使用介面（工具列、狀態列、選單、對話盒）。如果要以術語來說，Application Framework 就是由一整組合作無間的「物件」架構起來的大模型。嗯不不，當它還沒有與你的程式產生火花的時候，它還只是有形無體，應該說是一組合作無間的「類別」架構起來的大模型。
>
> 這帶來什麼好處呢？程式員只要帶個購物袋到「類別超級市場」採買，隨你要買 MDI 或 OLE 或 ODBC 或 Printing Preview，回家後就可以輕易拼湊出一個色香味俱全的大餐。
>
> 「類別超級市場」就是 C++ 類別庫，以產品而言，在 Microsoft 是 MFC，在 Borland 是 OWL，在 IBM 則是 OpenClass。這個類別庫不只是類別庫而已，傳統的函式庫（C Runtime 或 Windows API）相較於一般類別庫提供的是生鮮超市中的一條魚一支蔥一顆大白菜，彼此之間沒有什麼關聯，主掌中饋的你必須自己選材自己調理。能夠稱得上 Application Framework 者，提供的是火鍋拼盤（就是那種帶回家通通丟下鍋就好的那種），依你要的是白菜火鍋魚頭火鍋或是麻辣火鍋，菜色帶調理包都給你配好。當然這樣的火鍋拼盤是不能夠就地吃的，你得給它加點能量。放把火燒它吧，這火就是所謂的 application object（在 MFC 程式中就是衍生自 CWinApp 的一個全域性物件）。是這

個物件引起了連鎖反應（一連串的 'new'），使每一個形（類別）有了眞正的體（物件），把應用程式以及 Application Framework 整個帶動起來。一切因緣全自是起。

Application Framework 帶來的革命精神是，程式模型已經存在，程式員只要依個人需求加料就好：在衍生類別中改寫虛擬函式，或在衍生類別中加上新的成員函式。這很像你在火鍋拼盤中依個人口味加醬添醋。

由於程式碼的初期規模十分一致（什麼樣風格的程式應該使用什麼類別，是一成不變的），而修改程式以符合私人需要的基本動作很一致（我是指像「開闢一個空的畫幕函式」這種事情），你動不了 Application Framework 的大架構，也不需要動。這是福利不是約束。

應用程式碼單幕一致化的結果，使優越的軟體開發工具如 CASE（Computer Aid Software Engineering）tool 容易開發出來。你的程式碼大架構掌握在 Application Framework 設計者手上，於是他們就有能力製作出整合開發環境（Integrated Development Environment，IDE）了。這也是爲什麼 Microsoft、Borland、Symantec、Watcom、IBM 等公司的整合開發環境進步得如此令人咋舌的原因了。

有人認工學院中唯一保有人文氣息的只剩建築系，我總覺得資訊系也勉強可以算上。帶藝術氣息的軟體創作行爲（我一直是這麼認爲的）將在 Application Framework 出現後逐漸成爲工匠技術，而我們都將只是軟體 IC 裝配廠裡的男工女工。其實也沒什麼好顧影自憐，功成名就的冠冕從來也不曾落在程式員頭上；我們可能像紐約街頭的普普（POP）工作者，自認爲藝術家，可別人怎麼看呢？不得而知！話說回來，把開發軟體這件事情從藝術降格到工技，對人類只有好處沒有壞處。不是亨利福特，我們又如何能夠享受大眾化的汽車？或許以後會出現「純手工精製」的軟體，誰感興趣不得而知，我自己嘛...嗯...倒是從來不媒機客餘暇玩玩。

如果要三言兩語點出 Application Framework 的特質，我會這麼說：我們挖出別人早寫好的一整套模組（MFC 或 OWL 或 OpenClass）之中的一部份，給個引子（application object）使它們一一具象化動起來，並被允許修改其中某些零件使這程式更符合私人需

求，如是而已。

我怎麼說

侯捷的這一段話實在已經點出 Application Framework 的精神。凝聚性強、組織化強的類別庫就是 Application Framework。一組合作無間的物件，彼此藉訊息的流動而溝通，並且互相呼叫對方的函式以求完成任務，這就是 Application Framework。物件存在哪裡？在 MFC 中 ?! 這樣的說法不是十分完善，因爲 MFC 的各個類別只是「物件屬性（行爲）的定義」而已，我們不能夠說 MFC 中有實際的物件存在。唯有當程式被 application object（這是一個衍生自 MFC *CWinApp* 的全域物件）引爆了，才將我們選用的類別一一具象化起來，產生實體並開始動作。圖 **5-1** 是一個說明。

這樣子說吧，靜態情況下 MFC 是一組類別庫，但在程式執行時期它就生出了一群有活動力的物件組。最重要的一點是，這些物件之間的關係早已建立好，不必我們（程式員）操心。好比說當使用者按下選單的【File/Open】項，開檔對話盒就會打開；使用者選好檔名後，Application Framework 就開始對著你的資料類別，喚起一個名爲 *Serialize* 的特殊函式。這整個機制都埋好了，你只要把心力放在那個叫作 *Serialize* 的函式上即可。

選用標準的類別，做出來的產品當然就沒有什麼特色，因爲別人的零件和你的相同，兜起來的成品也就一樣。我指的是使用者介面（UI）物件。但你要知道，軟體工業發展到現階段這個世代，著重的已不再是 UI 的爭奇鬥豔，取巧譁眾；UI 已經漸漸走上標準化了。軟體一決勝負的關鍵在資料的處理。事實上，在「真正做事」這一點，整個 application framework 是無能爲力的，也就是說對於資料結構的安排，資料的處理，資料的顯示，Application Framework 所能提供的，無一不是單單一個空殼而已 -- 在 C++ 語言來講就是個虛擬函式。軟體開發人員必須想辦法改造（override）這些虛擬函式，才能符合個人所需。基於 C++ 語言的特性，我們很容易繼承既有之類別並加上自己的特色，這就是物件導向程式設計的主要精神。也因此，C++ 語言中有關於「繼承」性質的份量，在 MFC 程式設計裡頭佔有很重的比例，在學習使用 MFC 的同時，你應該對 C++ 的繼承性質和虛擬函式有相當的認識。第２章有我個人對 C++ 這兩個性質的心得。

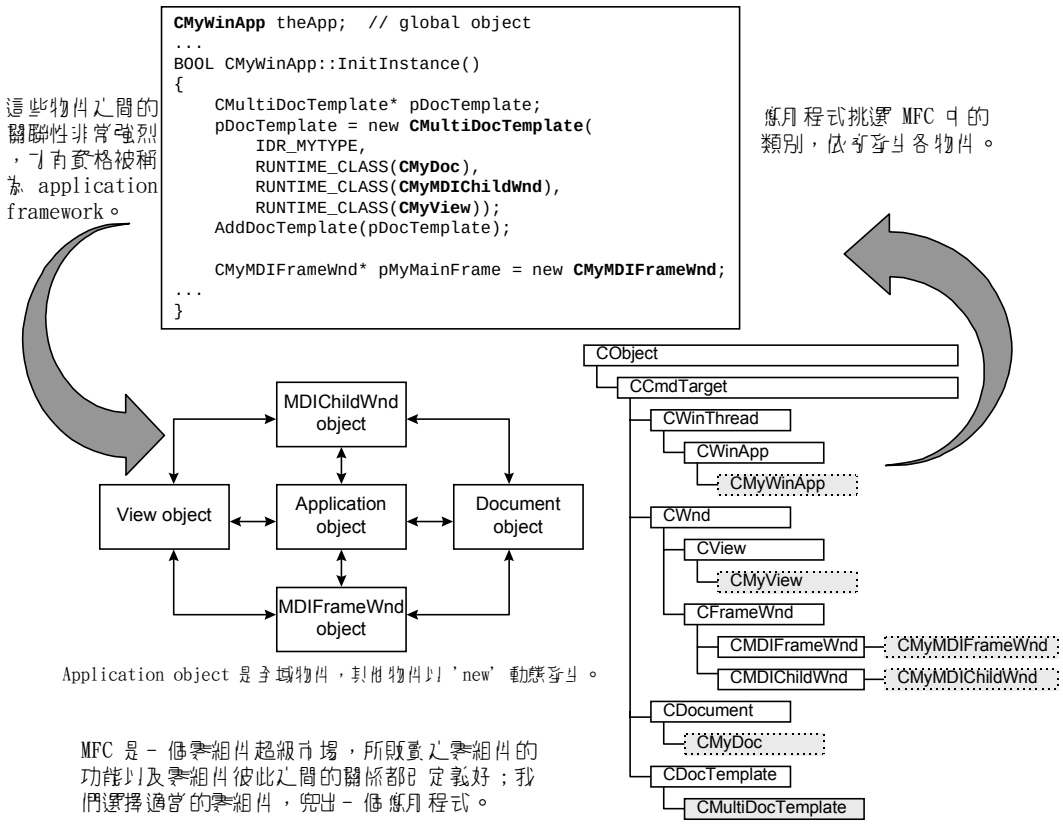


圖 5-1　MFC 是一個零組件超級市場，所販賣的零組件功能以及零組件彼此之間的關係都已定義好；我們選擇自己喜歡的零件，兜出一個應用程式。

Application Framework 究竟能提供我們多麼實用的類別呢？或者我們這麼問：哪些工作已經被處理掉了，哪些工作必須由程式員扛下來？比方說一個標準的 MFC 程式應該有一個可以讀寫檔案資料的功能，然而應用程式本身有它獨特的資料結構，從 MFC 得來的零件可能與我的個人需求搭配嗎？

我以另一個比喻做回答。

假設你買了一個整廠整線計劃，包括倉貯、物料、MIS、生管，各個部門之間的搭配也都建立起來了，包含在整廠整線計劃內。這個廠原先是爲了生產葡萄酒，現在改變主意要生產白蘭地，難道整個廠都不能用了嗎？不！只要把進貨原料改一改、發酵程序改一改、瓶裝程序改一改、整個廠的其它設備以及設備與設備之間的連線配合（這是頂頂重要的）都可以再利用。物料之後到生管，裝瓶之後到倉貯，倉貯之後到出貨，再加上 MIS 監控全廠，這些程序都是不必改變的。

一個整廠整線計劃，每一單元之間的連線溝通，合作關係，都已經建立起來，是一種建構好的運作模式。抽換某個單元的性質或部份性質，並不影響整體作業。「整廠整線」最重要最有價值的是各單元之間的流程與控制。

反映到物件導向程式設計裡頭，Application Framework 就是「整廠整線規劃」，Application Framework 提供的類別就是上述工廠中一個一個的單元。最具價值的就是各類別之間的交互運作模式。

雖然檔案讀寫（此動作在 MFC 稱爲 Serialize）單元必須改寫以符合個人所需，但是單元與單元之間的關係依然存在而且極富價值。當使用者在程式中選按【File/Open】或【File/Save】，主框視窗自動通知 Document 物件（內存資料），引發 Serialization 動作；而你爲了個人的需求，改寫了這個 *Serialize* 虛擬函式。你對 MFC 的改寫範圍與程度，視你的程式有多麼特異而定。

可是如果釀酒廠想改裝爲煉鋼廠？那可就無能爲力了。這種情況不會出現在軟體開發上，因爲軟體的必備功能使它們具有相當的相似性（尤其 Windows 又強調介面一致性）。好比說程式想支援 MDI 介面，想支援 OLE，這基本上是超越程式之專業應用領域之外的一種大格局，一種大架構，最適合 Application Framework 發揮。

很明顯，Application Framework 是一組超級的類別庫。能夠被稱爲 Framework 者必須其中的類別性質緊密咬合，互相呼應。因此你也就可以想像，Framework 所提供的類別是一夥的，不是單片包裝的。你把這夥東西放入程式裡，你也就得乖乖遵循一種特定的（Application Framework 所規定的）程式風格來進行程式設計工作。但是侯捷也告訴我

們，這是福利不是約束。

別人怎麼說

其他人又怎麼看 Application Framework？我將臚列數篇文章中的相關定義。若將原文譯爲中文，我恐怕力有未逮辭不達意，所以列出原文供你參考。

1985 年，Apple 公司的 MacApp 嚴格而系統化地定義出做爲一個商業化 Application Framework 所需要的關鍵理念：

> The key ideas of a commercial application framework : a generic app on steroids that provides a large amount of general-purpose functionality within a well-planned, well-tested, cohesive structure.

*cohesive 的意思是強而有力、有凝聚力的。

*steroid 是類固醇。目從加拿大 100 公尺名將班強生在漢城奧運比賽這藥物而奪得金牌並打破世界記錄，相信世人對這個名稱不會陌生（當然強生的這塊金牌和他的世界記錄後來是被取消的）。類固醇俗稱美國仙丹，是一種少膽固醇結構為基礎，衍生而來的荷爾蒙，對於發炎紅腫等症狀有極速療效。然而因為它是透過抑制人類免疫系統而得到療效，如果使用不當，會帶來極不良的副作用。運動員用於短時間內增強肌體機能的理性潤氯就是類固醇的一種，會影響脂肪代謝，服用過量會導致極大的副作用。

基本上 MacApp 以類固醇來比擬 Application Framework 雖是妙喻，但類固醇會對人體產生不好的副作用而 Application Framework 不會對軟體開發產生副作用-- 除非你認爲不能隨心所欲寫你的碼也算是一種副作用。

Apple 更一步更明確地定義一個 Application Framework 是：

> an extended collection of classes that cooperate to support a complete application architecture or application model, providing more complete application development support than a simple set of class libraries.

＊這裡所指的 support 並不只是視覺性 UI 元件如 menu、dialog、listbox...，還包括一個應用程式所需要的其他功能設備，像是 Document, View, Printing, Debugging。

另一個相關定義出現在 Ray Valdes 於 1992 年 10 月發表於 Dr. Dobb's Journal 的 "Sizing up Application Frameworks and Class Libraries" 一文之中：

> An application framework is an integrated object-oriented software system that offers all the application-level classes（documents, views, and commands）needed by a generic application.
>
> An application framework is meant to be used in its entirety, and fosters both design reuse and code reuse. An application framework embodies a particular philosophy for structuring an application, and in return for a large mass of prebuilt functionality, the programmer gives up control over many architectural-design decisions.

Donald G. Firesmith 在一篇名爲 "Frameworks : The Golden path of the object Nirvana" 的文章中對 Application Framework 有如下定義：

> What are frameworks？They are significant collections of collaborating classes that capture both the small-scale patterns and major mechanisms that, in turn, implement the common requirements and design in a specific application domain.

＊Nirvana 是涅槃、最高境界的意思。

Bjarne Stroustrup（C++ 原創者）在他的 The C++ Programming Language 一書中對於 Application Framework 也有如下敘述：

> Libraries build out of the kinds of classes described above support design and re-use of code by supplying building blocks and ways of combining them; the application builder designs a framework into which these common building blocks are fitted. An alternative, and sometimes more ambitious, approach to the support of design and re-use is to provide code that establishes a common framework into which the application builder fits application-specific code as building blocks. Such an approach is often called an application framework. The classes establishing such a framework often have such fat interfaces that they are hardly types in the traditional sense. They approximate the ideal of being complete applications, except that they don't do anything. The specific actions are supplied by the application programmer.

Kaare Christian 在 1994/02/08 的 PC Magazine 中有一篇 "C++ Application Frameworks" 文章，其中有下列敘述（節錄）：

> 那年前我在紐約北邊的鄉村蓋了一棟 post-and-beam 房子。在我到達之前我的木匠已經把每一根樑的外形設計好並製作好，把一根根的粗糙木材變成一塊塊鋸得漂漂亮亮的零件，一切準備就緒只待安裝。（註：所謂 post-and-beam 應是指那種樑柱都已規格化，可以郵購回來自己動手蓋的 DIY（Do It Yourself -- 房子）。
>
> 使用 Application Framework 建造一個 Windows 應用程式也有類似的過程。你使用一組早已做好的零件，它使你行進快速。由於這些零件堅強耐用而且穩固，後面的工作就簡單多了。但最重要的是，不論你使用規格化的樑柱框架來蓋一棟房子，或是使用 Application Framework 來建立一個 Windows 程式，工作型態已然改變，出現了一種完全嶄新的做事方法。在我的 post-and-beam 房子中，工作型態的改變並不總是帶來幫助；貿易商在預製樑柱的技巧上可能會遭遇到應上的困擾。同樣的事情最初也發生在 Windows 身上，因爲你原已具備的某些以 C 語言寫 Windows 程式的能力，現在在以

C++ 和 Application Framework 開發程式的過程中無非武之地。時間過去之後，Windows 程式設計的型態移轉終於帶來了偉大的利益與方便。Application Framework 本身把 message loops 和其他 Windows 的苦役都做掉了，它促進一個比較秩序井然的程式結構。

Application Framework -- 建立 Windows 應用軟體所用的 C++ 類別庫 -- 如今已行之有年，因爲物件導向程式設計已經快速地獲得了接受度。Windows API 是程序性的，Application Framework 則讓你寫物件導向式的 Windows 程式。它們提供預先寫好的機能（以 C++ 類別型式呈現出來），可以加速應用軟體的開發。

Application Framework 提供數種優點。或許最重要的，是它們在物件導向程式設計模式下對 Windows 程式設計過程的影響。你可以使用 Framework 來減輕例行但繁複的瑣事，目前的 Application Framework 可以在圖形、對話盒、列印、求助、OCX 控制元件、剪貼簿、OLE 等各方面幫助我們，它也可以產生漂亮的 UI 介面如工具列和狀態列。

藉著 Application Framework 的幫助寫出來的碼往往比較容易組織化，因爲 Framework 改變了 Windows 管理訊息的方法。也許有一天 Framework 還可以幫你維護單一一套碼以應付不同的執行平台。

你必須對 Application Framework 有很好的知識，才能夠修改由它附帶的軟體開發工具製作出來的骨幹程式。它們並不像 Visual Basic 那麼容易使用。但是對 Application Framework 專家而言，這些程式碼產生器可以省下大量時間。

使用 Application Framework 的主要缺點是，沒有單一一套產品廣被所有的 C++ 編譯器支援。所以當你選定一套 Framework，在某個範圍來說，你也等於是選擇了一個編譯器。

為什麼使用 Application Framework

雖然 Application Framework 並不是新觀念，它們卻在最近數年才成爲 PC 平台上軟體開發的主流工具。物件導向語言是具體實現 Application Framework 的理想載具，而 C++ 編譯器在 PC 平台上的出現與普及終於允許主流 PC 程式員能夠享受 Application Framework 帶來的利益。

從八十年代早期到九十年代初始，C++ 大都存在於 UNIX 系統和研究人員的工作站中，不在 PC 以及商業產品上。C++ 以及其他的物件導向語言（例如 Smalltalk-80）使一些大學和研究計劃生產出現今商業化 Application Framework 的鼻祖。但是這些早期產品並沒有明顯區隔出應用程式與 Application Framework 之間的界線。

今天應用軟體的功能愈來愈複雜，建造它們的工具亦復如此。Application Framework、Class Library 和 GUI toolkits 是三大類型的軟體開發工具（請見方塊說明），這三類工具雖然以不同的技術方式逼近目標，它們卻一致追求相同而基本的軟體開發關鍵利益：降低寫程式碼所花的精力、加速開發效率、加強可維護性、增加強固性（robustness）、爲組合式的軟體機能提供槓桿支點（有了這個支點，再大的軟體我也舉得起來）。

當我們面臨軟體工業革命，我們的第一個考量點是：我的軟體開發技術要從哪一個技術面切入？從 raw API 還是從高階一點的工具？如果答案是後者，第二個考量點是我使用哪一層級的工具？GUI toolkits 還是 Class Library 還是 Application Framework？如果答案又是後者，第三個考量點是我使用哪一套產品？MFC 或 OWL 或 Open Class Library？（目前 PC 上還沒有第四套隨編譯器附贈的 Application Framework 產品）

別認爲這是領導者的事情不是我（工程師）的事情，有這種想法你就永遠當不成領導者。也別認爲這是工程師的事情不是我（學生）的事情，學生的下一步就是工程師；及早想點工業界的激烈競爭，對你在學生階段規劃人生將有莫大助益。

我相信，Application Framework 是最好的槓桿支點。

Application Framework，Class Library，GUI toolkit

一般而言，Class Library 和 GUI toolkit 比 Application Framework 的規模小，定位也沒那麼高階宏觀。Class Library 可以定義為「一組具備物件導向性質的類別，它們便應用程式的某些功能實現起來容易一些，這些功能包括數值運算與資料結構、繪圖、記憶體管理等等；這些類別可以一大堆無瓜葛地併入應用程式內」。

請特別注意這個定義中所強調的「一大堆無瓜葛」，而不像 Application Framework 是大夥兒一併加入。因此，你儘可以隨意使用 Class Library，它並不會強迫你遵循任何特定的程式架構。Class Library 通常提供的不只是 UI 功能、也包括一般性質的機能，像資料結構的處理、日期與時間的轉換等等。

GUI toolkit 提供的服務類似 Class Library，但它的程式介面是程序導向而非物件導向。而且它的功能大都集中在圖形與 UI 介面上。GUI toolkit 的雙長期皆在物件導向語言之前，某些極為成功的產品甚至是以組合語言（assembly）寫成。不要心然地把 GUI 聯想到 Windows，GUI toolkit 也有 DOS 版本。我用過的 Chatter Box 就是 DOS 環境下的 GUI 工具（是一個函式庫）。

使用 Application Framework 的最直接原因是，我們受夠了日益暴增的 Windows API。把 MFC 想像爲第四代語言，單單一個類別就幫我們做掉原先要以一大堆 APIs 才能完成的事情。

但更深入地想，Application Framework 絕不只是爲了降低我們花在浩瀚無涯的 Windows API 的時間而已；它所帶來的物件導向程式設計觀念與方法，使我們能夠站在一群優秀工程師（MFC 或 OWL 的創造者）的努力心血上，繼承其成果而開發自己之所需。同時，因爲 Application Framework 特殊的工作型態，整體開發工具更容易製作，也製作的更完美。在我們決定使用 Application Framework 的同時，我們也獲得了這些整合性軟體開發環境的支援。在軟體開發過程中，這些開發工具角色之吃重不亞於 Application Framework 本身。

Application Framework 將成爲軟體技術中最重要的一環。如果你不知道它是什麼，趕快學習它；如果你還沒有使用它，趕快開始用。機會之窗不會永遠爲你打開，在你的競爭者把它關閉之前趕快進入！如果你認爲改朝換代還早得很，請注意兩件事情。第一，江山什麼時候變色可誰也料不準，當你埋首工作時，外面的世界進步尤其飛快；第二，物件導向和 Application Framework 可不是那麼容易學的，花多少時間才能登堂入室可還得憑各人資質和基礎呢。

浩瀚無涯的 Windows API

Windows 版本	推出日期	API 個數	訊息個數
1.0	1985.11	379	?
2.0	1987.11	458	?
3.0	1990.05	578	?
Multimedia Ex.	1991.12	120	?
3.1	1992.04	973	271
Win32s	1993.08	838	287
Win32	1993.08	1449（持續增加當中）	291（持續增加當中）

API Jungle

Microsoft Foundation Classes（MFC）

PC 世界裡出了三套 C++ Application Frameworks，並且有愈多愈多的趨勢。這三套是 Microsoft 的 MFC（Microsoft Foundation Classes），Borland 的 OWL（Object WindowLibrary），以及 IBM VisualAge C++ 的 Open Class Library。至於其他 C++ 編譯器廠商如 Watcom、Symantec、Metaware，只是供應整合開發環境（Integraded Development Environment，IDE），其 Application Framework 都是採用微軟公司的 MFC。

> Delphi（Pascal 語言），依我之見，也稱得上是一套 Application Framework。Java 語言本身內建一套標準類別庫，依我之見，也夠得上資格被稱爲 Application Framework。
>
> Delphi 和 Visual Basic，又被稱爲是一種應用程式快速開發工具（RAD，Rapid Application Development）。它們採用 PME（Properties-Method-Event）架構，寫程式的過程像是在一張畫布上拼湊一個個現成的元件（components）：設定它們的屬性（properties）、指定它們應該「有所感」的外來刺激（events），並決定它們面對此刺激時在預設行爲之外的行爲（methods）。所有動作都以拖拉、設定數值的方式完成，非常簡單。只有在設定元件與元件之間的互動關係時才牽涉到程式碼的寫作（這一小段碼也因此成爲順利成功的關鍵）。
>
> Borland 公司於 1997 年三月推出的 C++ Builder 也屬於 PME 架構，提供一套 Visual Component Library（VCL），內有許許多多的元件。因此 C++ Builder 也算得上是一套 RAD（應用程式快速開發工具）。

早初，開發 Windows 應用程式必須使用微軟的 SDK（Software Development Kit），直接呼叫 Windows API 函式，向 Windows 作業系統提出各種要求，例如配置記憶體、開啓視窗、輸出圖形...。

> 所謂 API（Application Programming Interface），就是開放給應用程式呼叫的系統功能。

數以千計的 Windows APIs，每個看起來都好像比重相若（至少你從手冊上看不出來孰輕孰重）。有些 APIs 彼此雖有群組關係，卻沒有相近或組織化的函式名稱。星羅棋佈，霧列星馳；又似雪球一般愈滾愈多，愈滾愈大。撰寫 Windows 應用程式需要大量的耐力與毅力，以及大量的小心謹慎！

MFC 幫助我們把這些浩繁的 APIs，利用物件導向的原理，邏輯地組織起來，使它們具備抽象化、封裝化、繼承性、多型性、模組化的性質。

1989 年微軟公司成立 Application Framework 技術團隊，名爲 AFX 小組，用以開發 C++ 物件導向工具給 Windows 應用程式開發人員使用。AFX 的 "X" 其實沒有什麼意義，只是爲了湊成一個響亮好唸的名字。

這個小組最初的「憲章」，根據記載，是要 "utilize the latest in object oriented technology to provide tools and libraries for developers writing the most advanced GUI applications on the market"，其中並未畫地自限與 Windows 作業系統有關。果然，其第一個原型產品，有自己的視窗系統、自己的繪圖系統、自己的物件資料庫、乃至於自己的記憶體管理系統。當小組成員以此產品開發應用程式，他們發現實在是太複雜，又悖離公司的主流系統 -- Windows -- 太遙遠。於是他們修改憲章變成 "deliver the power of object-oriented solutions to programmers to enable them to build world-class Windows based applications in C++." 這差不多正是 Windows 3.0 異軍崛起的時候。

C++ 是一個複雜的語言，AFX 小組預期 MFC 的使用者不可能人人皆爲 C++ 專家，所以他們並沒有採用所有的 C++ 高階性質（例如多重繼承）。許多「麻煩」但「幾乎一成不變」的 Windows 程式動作都被隱藏在 MFC 類別之中，例如 *WinMain*、*RegisterClass*、Window Procedure 等等等。

> 雖說這些被隱藏的 Windows 程式動作幾乎是一成不變的，但它們透露了 Windows 程式的原型奧秘，這也是爲什麼我要在本書之中鍥而不捨地挖出它們的原因。

爲了讓 MFC 儘可能地小，儘可能地快，AFX 小組不得不捨棄高度的抽象（導至過多的虛擬函式），而引進他們自己發明的機制，嘗試在物件導向領域中解決 Windows 訊息的處理問題。這也就是本書第 9 章深入探討的 Message Mapping 和 Message routing 機制。注意，他們並沒有改變 C++ 語言本身，也沒有擴大語言的功能。他們只是設計了一些令人拍案叫絕的巨集，而這些巨集背後隱藏著巨大的機制。

了解這些巨集（以及它們背後所代表的機制）的意義，以及隱藏在 MFC 類別之中的那些足以曝露原型機密的「麻煩事兒」，正是我認爲掌握 MFC 這套 Application Framework 的重要手段。

就如同前面那些形而上的定義，MFC 是一組凝聚性強、組織性強的類別庫。如果你要利用 MFC 發展你的應用程式，必須同時引用數個必要類別，互相搭配奧援。**圖 5-3** 是一個標準的 MFC 程式外貌。隱藏在精緻畫面背後更重要的是，就如我在前面說過，物件與物件之間的關係已經存在，訊息的流動程序也都已設定。當你要爲這個程式設計真正的應用功能，不必在意諸如「我如何得知使用者按左鍵？左鍵按下後我如何啓動某一個函式？參數如何傳遞過去...」等瑣事，只要專注在左鍵之後真正要做的功能動作就好。

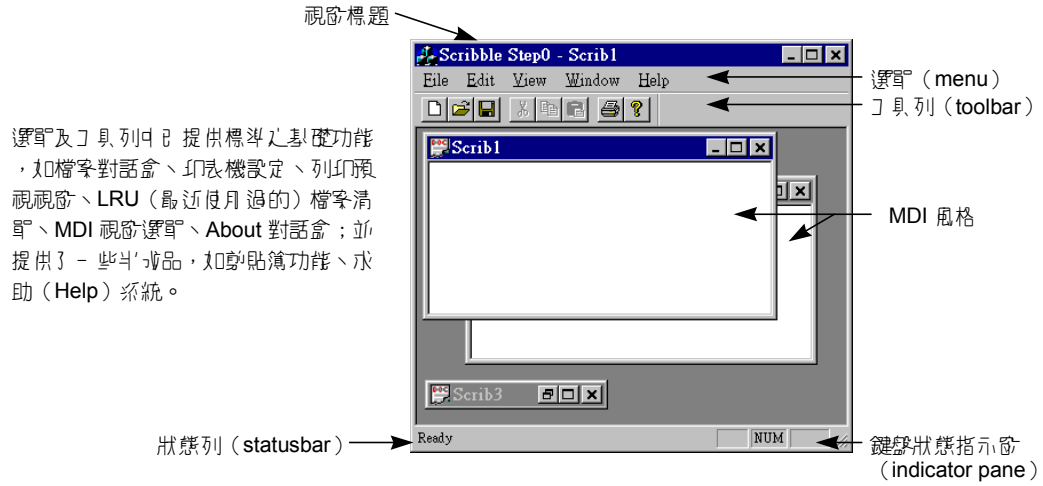


圖 5-3　標準 MFC 程式的風貌。

日頭穿衣話天數：Visual C++ 與 MFC

微軟公司於 1992/04 推出 C/C++ 7.0 產品時初次向世人介紹了 MFC 1.0，這個初試啼聲的產品包含了 20,000 行 C++ 原始碼，60 個以上的 Windows 相關類別，以及其他的一般類別如時間、資料處理、檔案、記憶體、診斷、字串等等等。它所提供的，其實是一個 "thin and efficient C++ transformation of the Windows API"。其 32 位元版亦在 1992/07 隨著 Win32 SDK 推出。

MFC 1.0 獲得的回響帶給 AFX 小組不少鼓舞。他們的下一個目標放在：

- 更高階的架構支援
- 罐裝元件（尤其在使用者介面上）

前者成就了 Document/View 架構，後者成就了工具列、狀態列、列印、預覽等極受歡迎的 UI 性質。當然，他們並沒有忘記相容性與移植性。雖然 AFX 小組並未承諾 MFC 可以跨不同作業系統如 UNIX XWindow、OS/2 PM、Mac System 7，但在其本家（Windows 產品線）身上，在 16 位元 Windows 3.x 和 32 位元 Windows 95 與 Windows NT 之間的移植性是無庸置疑的。雖然其 16 位元產品和 32 位元產品是分別包裝銷售，你的原始碼通常只需重新編譯聯結即可。

Visual C++ 1.0（也就是 C/C++ 8.0）搭配 MFC 2.0 於 1993/03 推出，這是針對 Windows 3.x 的 16 位元產品。接下來又在 1993/08 推出在 Windows NT 上的 Visual C++ 1.1 for Windows NT，搭配的是 MFC 2.1。這兩個版本有著相同的基本性質。MFC 2.0 內含近 60,000 行 C++ 程式碼，分散在 100 個以上的類別中。Visual C++ 整合環境的數個重要工具（大家熟知的 Wizards）本身即以 MFC 2.0 設計完成，它們的出現對於軟體生產效率的提升有極大貢獻。

微軟在 1993/12 又推出了 16 位元的 Visual C++ 1.5，搭配 MFC 2.5。這個版本最大的進步是多了 OLE2 和 ODBC 兩組類別。整合環境也爲了支援這兩組類別而做了些微改變。

1994/09，微軟推出 Visual C++ 2.0，搭配 MFC 3.0，這個 32 位元版本主要的特徵在於配合目標作業系統（Windows NT 和 Windows 95），支援多執行緒。所有類別都是 thread-safe。UI 物件方面，加入了屬性表（Property Sheet）、miniframe 視窗、可隨處停駐的工具列。MFC collections 類別改良爲 template-based。聯結器有重大突破，原使用的 Segmented Executable Linker 改爲 Incremental Linker，這種聯結器在對 OBJ 檔做聯結時，並不每次從頭到尾重新來過，而只是把新資料往後加，舊資料加記作廢。想當然耳，EXE 檔會累積許多不用的垃圾，那沒關係, 透過 Win32 memory-mapped file, 作業系統（Windows NT 及 Windows 95）只把欲使用的部份載入，絲毫不影響執行速度。必要時程式員也可選用傳統方式聯結，這些垃圾自然就不見了。對我們這些終日受制於 edit-build-run-debug 輪迴的程式員, Incremental Linker 可真是個好禮物。

1995/01，微軟又加上了 MAPI（Messaging API）和 WinSock 支援，推出 MFC 3.1（32 位元版），並供應 13 個通用控制元件，也就是 Windows 95 所提供的 tree、tooltip、spin、slider、progress、RTF edit 等等控制元件。

1995/07，MFC 有了 3.2 版，那是不值一提的小改版。

然後就是 1995/09 的 32 位元 MFC 4.0。這個版本納入了 DAO 資料庫類別、多執行緒同步控制類別，並允許製作 OCX containers。搭配推出的 Visual C++ 4.0 編譯器，也終於支援了 template、RTTI 等 C++ 語言特性。IDE 整合環境有重大的改頭換面行動，Class View、Resource View、File View 都使得專案的管理更直覺更輕鬆，Wizardbar 則活脫脫是一個簡化的 ClassWizard。此外，多了一個極好用的 Components Gallery，並允許程式員訂製 AppWizard。

1996 年上半年又推出了 MFC 4.1，最大的焦點在 ISAPI（Internet Server API）的支援，提供五個新類別，分別是 *CHttpServer*、*CHttpFilter*、*CHttpServerContext*、*CHttpFilterContext*、*CHtmlStream*，用以建立互動式 Web 應用程式。整合環境方面也對應地提供了一個 ISAPI Extension Wizard。在附加價值上，Visual C++ 4.1 提供了 Game SDK，幫助開發 Windows 95 上的高效率遊戲軟體。Visual C++ 4.1 還提供不少個由協力

公司完成的 OLE 控制元件（OCXs），這些 OLE 控制元件技術很快就要全面由桌上躍到網上，稱爲 ActiveX 控制元件。不過，遺憾的是，Visual C++ 4.1 的編譯器有些臭蟲，不能夠製作 VxD（虛擬裝置驅動程式）。

1996 年下半年推出的 MFC 4.2，提供對 ActiveX 更多的技術支援，並整合 Standard C++ Library。它封包一組新的 Win32 Internet 類別（統稱爲 WinInet），使 Internet 上的程式開發更容易。它提供 22 個新類別和 40 個以上的新成員函式。它也提供一些控制元件，可以繫結（binding）近端和遠端的資料源（data sources）。整合環境方面，Visual C++ 4.2 提供新的 Wizard 給 ActiveX 程式開發使用，改善了影像編輯器，使它能夠處理在 Web 伺服器上的兩個標準圖檔格式：GIF 和 JPEG。

1997 年五月推出的 Visual C++ 5.0，主要訴求在編譯器的速度改善，並將 Visual C++ 合併到微軟整個 Visual Tools 的終極管理軟體 Visual Studio 97 之中。所有的微軟虛擬開發工具，包括 Visual C++、Visual Basic、Visual J++、Visual InterDev、Visual FoxPro、都在 Visual Studio 97 的整合之下有更密切的彼此奧援。至於程式設計方面，MFC 本身沒有什麼變化（4.21 版），但附了一個 ATL（Active Template Library）2.1 版，使 ActiveX 控制元件的開發更輕鬆些。

我想你會發現，微軟正不斷地爲「爲什麼要使用 MFC」加上各式各樣的強烈理由，並強烈導引它成爲 Windows 程式設計的 C++ 標準介面。你會看到愈來愈多的 MFC/C++ 程式碼。對於絕大多數的技術人員而言，Application Framework 的抉擇之道無它，「MFC 是微軟公司欽定產品」，這個理由就很嗆人了。

縱覽 MFC

MFC 非常巨大（其他 application framework 也不差），在下一章正式使用它之前，讓我們先做個瀏覽。

> 請同時參考書後所附之 MFC 架構圖

MFC 類別主要可分爲下列數大群組：

- General Purpose classes - 提供字串類別、資料處理類別（如陣列與串列），異常情況處理類別、檔案類別...等等。
- Windows API classes - 用來封包 Windows API，例如視窗類別、對話盒類別、DC 類別...等等。
- Application framework classes - 組成應用程式骨幹者，即此組類別，包括 Document/View、訊息邦浦、訊息映射、訊息繞行、動態生成、檔案讀寫等等。
- High level abstractions - 包括工具列、狀態列、分裂視窗、捲動視窗等等。
- operation system extensions - 包括 OLE、ODBC、DAO、MAPI、WinSock、ISAPI 等等。

General Purpose classes

也許你使用 MFC 的第一個目標是爲了寫 Windows 程式，但並不是整個 MFC 都只爲此目的而活。下面這些類別適用於 Windows，也適用於 DOS。

CObject

絕大部份類別庫，往往以一個或兩個類別，做爲其他絕大部份類別的基礎。MFC 亦復如此。*CObject* 是萬類之首，凡類別衍生自 *CObject* 者，得以繼承數個物件導向重要性質，包括 RTTI（執行時期型別鑑識）、Persistence（物件保存）、Dynamic Creation（動態生成）、Diagnostic（錯誤診斷）。本書第 3 章對於這些技術已有了一份 DOS 環境下的模擬，第 8 章另有 MFC 相關原始碼的探討。其中，「物件保存」又牽扯到 *CArchive*，「診斷」又牽扯到 *CDumpContext*，「執行時期型別鑑識」以及「動態生成」又牽扯到 *CRuntimeClass*。

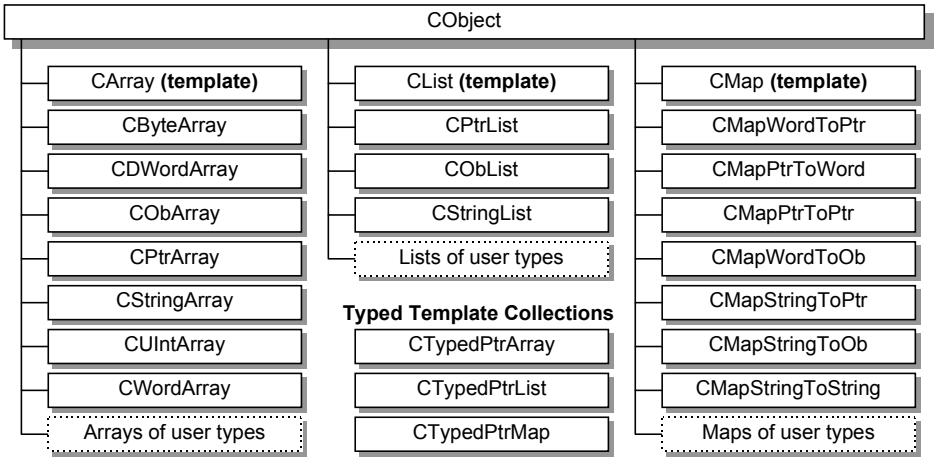
資料處理類別（collection classes）

所謂 collection，意指用來管理一「群」物件或標準型態的資料。這些類別像是 Array 或

List 或 Map 等等，都內含針對元素的「加入」或「刪除」或「巡訪」等成員函式。Array（陣列）和 List（串列）是資料結構這門課程的重頭戲，大家比較熟知，Map（可視之爲表格）則是由成雙成對的兩兩物件所構成，使你很容易由某一物件得知成對的另一物件；換句話說一個物件是另一個物件的鍵值（key）。例如，你可以使用 String-to-String Map，管理一個「電話-人名」資料庫；或者使用 Word-to-Ptr Map，以 16 位元數值做爲一個指標的鍵值。

最令人側目的是，由於這些類別都支援 Serialization，一整個陣列或串列或表格可以單一一行程式碼就寫到檔案中（或從檔案讀出）。第 8 章的 Scribble Step1 範例程式中你就會看到它的便利。

MFC 支援的 collection classes 有：



雜項類別

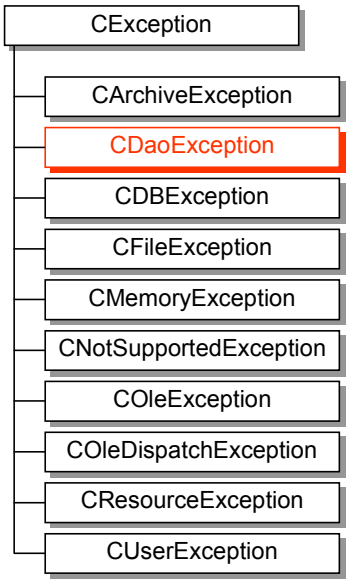
- *CRect* - 封裝 Windows 的 *RECT* 結構。這個類別在 Windows 環境中特別有用，因爲 *CRect* 常常被用作 MFC 類別成員函式的參數。
- *CSize* - 封裝 Windows 的 *SIZE* 結構。
- *CPoint* - 封裝 Windows 的 *POINT* 結構。這個類別在 Windows 環境中特別有用，

因爲 *CPoint* 常常被用作 MFC 類別成員函式的參數。

- *CTime* - 表現絕對時間，提供許多成員函式，包括取得目前時間（*static GetCurrentTime*）、將時間資料格式化、抽取特定欄位（時、分、秒）等等。它對於 +、-、+=、-+ 等運算子都做了多載動作。
- *CTimeSpan* - 以秒數表現時間，通常用於計時碼錶。提供許多成員函式，包括把秒數轉換爲日、時、分、秒等等。
- *CString* - 用來處理字串。支援標準的運算子如 =、+=、< 和 >。

異常處理類別（exception handling classes）

所謂異常情況（exception），是發生在你的程式執行時期的不正常情況，像是檔案打不開、記憶體不足、寫入失敗等等等。我曾經在第 2 章最後面介紹過異常處理的觀念及相關的 MFC 類別，並在第 4 章「Exception Handling」一節介紹過一個簡單的例子。與「異常處理」有關的 MFC 類別一共有以下 11 種：



Windows API classes

這是 MFC 聲名最著的一群類別。如果你去看看原始碼，就會看到這些類別的成員函式所對應的各個 Windows API 函式。

- *CWinThread* - 代表 MFC 程式中的一個執行緒。自從 3.0 版之後，所有的 MFC 類別就都已經是 thread-safe 了。SDK 程式中標準的訊息迴路已經被封裝在此一類別之中（你會在第 6 章看到我如何把這一部份開膛剖肚）。
- *CWinApp* - 代表你的整個 MFC 應用程式。此類別衍生自 *CWinThread*；要知道，任何 32 位元 Windows 程式至少由一個執行緒構成。*CWinApp* 內含有用的成員變數如 *m_szExeName*，放置執行檔檔名，以及有用的成員函式如 *ProcessShellCommand*，處理命令列選項。
- *CWnd* - 所有視窗，不論是主框視窗、子框視窗、對話盒、控制元件、view 視窗，都有一個對應的 C++ 類別，你可以想像「視窗 handle」和「C++ 物件」結盟。這些 C++ 類別統統衍生自 *CWnd*，也就是說，凡衍生自 *CWnd* 之類別才能收到 *WM_* 視窗訊息（*WM_COMMAND* 除外）。

  所謂「視窗 handle」和「C++ 物件」結盟，實際上是 *CWnd* 物件有一個成員變數 *m_hWnd*，就放著對應的視窗 handle。所以，只要你手上有一個 *CWnd* 物件或 *CWnd* 物件指標，就可以輕易獲得其視窗 handle：

```
HWND hWnd = pWnd->m_hWnd;
```

- *CCmdTarget* - *CWnd* 的父類別。衍生自它，類別才能夠處理命令訊息 *WM_COMMAND*。這個類別是訊息映射以及命令訊息繞行的大部份關鍵，我將在第 9 章推敲這兩大神秘技術。
- GDI 類別、DC 類別、Menu 類別。

Application framework classes

這一部份最爲人認知的便是 Document/View，這也是使 MFC 躋身 application framework 的關鍵。Document/View 的觀念是希望把資料的本體，和資料的顯像分開處理。由於文件產生之際，必須動態生成 Document/View/Frame 三種物件，所以又必須有所謂的 Document Template 管理之。

- *CDocTemplate*、*CSingleDocTemplate*、*CMultiDocTemplate* - Document Template 扮演黏膠的角色，把 Document 和 View 和其 Frame（外框視窗）膠黏在一塊兒。*CSingleDocTemplate* 一次只支援一種文件型態，*CMultiDocTemplate* 可同時支援多種文件型態。注意，這和 MDI 程式或 SDI 程式無關，換句話說，MDI 程式也可以使用 *CSingleDocTemplate*，SDI 程式也可以使用 *CMultiDocTemplate*。

  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*。

- *CDocument* - 當你爲自己的程式由 *CDocument* 衍生出一個子類別後，應該在其中加上成員變數，以容納文件資料；並加上成員函式，負責修改文件內容以及讀寫檔。讀寫檔由虛擬函式 *Serialize* 負責。第 8 章的 Scribble Step1 範例程式有極佳的示範。

- *CView* - 此類別負責將文件內容呈現到顯示裝置上：也許是螢幕，也許是印表機。文件內容的呈現由虛擬函式 *OnDraw* 負責。由於這個類別實際上就是你在螢幕上所看到的視窗（外再罩一個外框視窗），所以它也負責使用者輸入的第一線服務。例如第 8 章的 Scribble Step1 範例，其 View 類別便處理了滑鼠的按鍵動作。

High level abstractions

視覺性 UI 物件屬於此類，例如工具列 *CToolBar*、狀態列 *CStatusBar*、對話盒列 *CDialogBar*。加強型的 View 也屬此類，如可捲動的 *ScrollView*、以對話盒爲基礎的

CFormView、小型文字編輯器 *CEditView*、樹狀結構的 *CTreeView*，支援 RTF 文件格式的 *CRichEditView* 等等。

Afx 全域函式

還記得吧，C++ 並不是純種的物件導向語言（SmallTalk 和 Java 才是）。所以，MFC 之中得以存在有不屬於任何類別的全域函式，它們統統在函式名稱開頭冠以 *Afx*。

下面是幾個常見的 Afx 全域函式：

函式名稱	說明
AfxWinInit	被 *WinMain*（由 MFC 提供）呼叫的一個函式，用做 MFC GUI 程式初始化的一部份，請看第 6 章的「*AfxWinInit* - AFX 內部初始化動作」一節。 如果你寫一個 MFC console 程式，就得自行呼叫此函式（請參考 Visual C++ 所附之 Tear 範例程式）。
AfxBeginThread	開始一個新的執行緒（請看第 14 章，# 756 頁）。
AfxEndThread	結束一個舊的執行緒（請看第 14 章，# 756 頁）。
AfxFormatString1	類似 *printf* 一般地將字串格式化。
AfxFormatString2	類似 *printf* 一般地將字串格式化。
AfxMessageBox	類似 Windows API 函式 *MessageBox*。
AfxOutputDebugString	將字串輸往除錯裝置（請參考附錄 D，# 924 頁）。
AfxGetApp	取得 application object（*CWinApp* 衍生物件）的指標。
AfxGetMainWnd	取得程式主視窗的指標。
AfxGetInstance	取得程式的 instance handle。
AfxRegisterClass	以自定的 *WNDCLASS* 註冊視窗類別（如果 MFC 提供的數個視窗類別不能滿足你的話）。

MFC 巨集（macros）

CObject 和 *CRuntimeClass* 之中封裝了數個所謂的 object services，包括「取得執行時期的類別資訊」（RTTI）、Serialization（檔案讀寫）、動態產生物件...等等。所有衍生自 *CObject*

的類別，都繼承這些機能。我想你對這些名詞及其代表的意義已經不再陌生 -- 如果你沒有錯過第 3 章的「MFC 六大技術模擬」的話。

- 取得執行時期的類別資訊（RTTI），使你能夠決定一個執行時期的物件的類別資訊，這樣的能力在你需要對函式參數做一些額外的型態檢驗，或是當你要針對物件屬於某種類別而做特別的動作時，份外有用。
- Serialization 是指將物件內容寫到檔案中，或從檔案中讀出。如此一來物件的生命就可以在程式結束之後還延續下去，而在程式重新啓動之後，再被讀入。這樣的物件可說是 "persistent"（永續存在）。
- 所謂動態的物件生成（Dynamic object creation），使你得以在執行時期產生一個特定的物件。例如 document、view、和 frame 物件就都必須支援動態物件生成，因爲 framework 需要在執行時期產生它們（第 8 章有更詳細的說明）。

此外，OLE 常常需要在執行時期做物件的動態生成動作。例如一個 OLE server 程式必須能夠動態產生 OLE items，用以反應 OLE client 的需求。

MFC 針對上述這些機能，準備了一些巨集，讓程式能夠很方便地繼承並實作出上述四大機能。這些巨集包括：

巨集名稱	提供機能	出現章節
DECLARE_DYNAMIC	執行時期類別資訊	第 3 章、第 8 章
IMPLEMENT_DYNAMIC	執行時期類別資訊	第 3 章、第 8 章
DECLARE_DYNCREATE	動態生成	第 3 章、第 8 章
IMPLEMENT_DYNCREATE	動態生成	第 3 章、第 8 章
DECLARE_SERIAL	物件內容的檔案讀寫	第 3 章、第 8 章
IMPLEMENT_SERIAL	物件內容的檔案讀寫	第 3 章、第 8 章
DECLARE_OLECREATE	OLE 物件的動態生成	不在本書範圍之內
IMPLEMENT_OLECREATE	OLE 物件的動態生成	不在本書範圍之內

我也已經在第３章提過 MFC 的訊息映射（Message Mapping）與命令繞行（Command Routing）兩個特性。這兩個性質係由以下這些 MFC 巨集完成：

巨集名稱	提供機能	出現章節
DECLARE_MESSAGE_MAP	宣告訊息映射表資料結構	第３章、第９章
BEGIN_MESSAGE_MAP	開始訊息映射表的建置	第３章、第９章
ON_COMMAND	增加訊息映射表中的項目	第３章、第９章
ON_CONTROL	增加訊息映射表中的項目	本書未舉例
ON_MESSAGE	增加訊息映射表中的項目	???
ON_OLECMD	增加訊息映射表中的項目	本書未舉例
ON_REGISTERED_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_REGISTERED_THREAD_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_THREAD_MESSAGE	增加訊息映射表中的項目	本書未舉例
ON_UPDATE_COMMAND_UI	增加訊息映射表中的項目	第３章、第９章
END_MESSAGE_MAP	結束訊息映射表的建置	第３章、第９章

事實上，與其他 MFC Programming 書籍相比較，本書最大的一個特色就是，要把上述這些 MFC 巨集的來龍去脈交待得非常清楚。我認爲這對於撰寫 MFC 程式是非常重要的一件事。

MFC 資料型態（data types）

下面所列的這些資料型態，常常出現在 MFC 之中。其中的絕大部份都和一般的 Win32 程式（SDK 程式）所用的相同。

下面這些是和 Win32 程式（SDK 程式）共同使用的資料型態：

資料型態	意義
BOOL	Boolean 值（布林值，不是 TRUE 就是 FALSE）
BSTR	32-bit 字元指標
BYTE	8-bit 整數，未帶正負號
COLORREF	32-bit 數值，代表一個顏色值
DWORD	32-bit 整數，未帶正負號
LONG	32-bit 整數，帶正負號
LPARAM	32-bit 數值，做爲視窗函式或 callback 函式的一個參數
LPCSTR	32-bit 指標，指向一個常數字串
LPSTR	32-bit 指標，指向一個字串
LPCTSTR	32-bit 指標，指向一個常數字串。此字串可移植到 Unicode 和 DBCS（雙位元組字集）
LPTSTR	32-bit 指標，指向一個字串。此字串可移植到 Unicode 和 DBCS（雙位元組字集）
LPVOID	32-bit 指標，指向一個未指定型態的資料
LPRESULT	32-bit 數值，做爲視窗函式或 callback 函式的回返值
UINT	在 Win16 中是一個 16-bit 未帶正負號整數，在 Win32 中是一個 32-bit 未帶正負號整數。
WNDPROC	32-bit 指標，指向一個視窗函式
WORD	16-bit 整數，未帶正負號
WPARAM	視窗函式的 callback 函式的一個參數。在 Win16 中是 16 bits，在 Win32 中是 32 bits。

下面這些是 MFC 獨特的資料型態：

資料型態	意義
POSITION	一個數值，代表 collection 物件（例如陣列或串列）中的元素位置。常使用於 MFC collection classes。
LPCRECT	32-bit 指標，指向一個不變的 RECT 結構。

前面所說那些 MFC 資料型態與 C++ 語言資料型態之間的對應，定義於 WINDEF.H 中。我列出其中一部份，並且將不符合 (_MSC_VER >= 800) 條件式的部份略去。

```
#define NULL    0

#define far             // 侯俊傑註：Win32 不再有 far 或 near memory model，
#define near            // 而是使用所謂的 flat model。pascall 函式呼叫習慣
#define pascal __stdcall  // 也被 stdcall 函式呼叫習慣取而代之。

#define cdecl _cdecl
#define CDECL _cdecl

#define CALLBACK    __stdcall  // 侯俊傑註：在 Windows programming 演化過程中
#define WINAPI      __stdcall  // 曾經出現的 PASCAL、CALLBACK、WINAPI、
#define WINAPIV     __cdecl    // APIENTRY，現在都代表相同的意義，就是 stdcall
#define APIENTRY    WINAPI     // 函式呼叫習慣。
#define APIPRIVATE  __stdcall
#define PASCAL      __stdcall

#define FAR         far
#define NEAR        near
#define CONST       const

typedef unsigned long         DWORD;
typedef int                   BOOL;
typedef unsigned char         BYTE;
typedef unsigned short        WORD;
typedef float                 FLOAT;
typedef FLOAT                 *PFLOAT;
typedef BOOL near             *PBOOL;
typedef BOOL far              *LPBOOL;
typedef BYTE near             *PBYTE;
typedef BYTE far              *LPBYTE;
```

```
typedef int near         *PINT;
typedef int far          *LPINT;
typedef WORD near        *PWORD;
typedef WORD far         *LPWORD;
typedef long far         *LPLONG;
typedef DWORD near       *PDWORD;
typedef DWORD far        *LPDWORD;
typedef void far         *LPVOID;
typedef CONST void far   *LPCVOID;

typedef int              INT;
typedef unsigned int     UINT;
typedef unsigned int     *PUINT;

/* Types use for passing & returning polymorphic values */
typedef UINT WPARAM;
typedef LONG LPARAM;
typedef LONG LRESULT;

typedef DWORD   COLORREF;
typedef DWORD   *LPCOLORREF;

typedef struct tagRECT
{
   LONG   left;
   LONG   top;
   LONG   right;
   LONG   bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;

typedef const RECT FAR* LPCRECT;

typedef struct tagPOINT
{
   LONG  x;
   LONG  y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;

typedef struct tagSIZE
{
   LONG       cx;
   LONG       cy;
} SIZE, *PSIZE, *LPSIZE;
```

第 6 章　MFC 程式設計導論

MFC 程式的生死因果

理想如果不向實際做點妥協，理想就會歸於塵土。
中華民國還得十次革命才得建立，物件導向怎能把一切傳統都拋開。

以傳統的 C/SDK 撰寫 Windows 程式，最大的好處是可以清楚看見整個程式的來龍去脈和訊息動向，然而這些重要的動線在 MFC 應用程式中卻隱晦不明，因爲它們被 Application Framework 包起來了。這一章主要目的除了解釋 MFC 應用程式的長像，也要從 MFC 原始碼中檢驗出一個 Windows 程式原本該有的程式進入點（*WinMain*）、視窗類別註冊（*RegisterClass*）、視窗產生（*CreateWindow*）、訊息迴路（Message Loop）、視窗函式（Window Procedure）等等動作，抽絲剝繭徹底了解一個 MFC 程式的誕生與結束，以及生命過程。

爲什麼要安排這一章？了解 MFC 內部構造是必要的嗎？看電視需要知道映像管的原理嗎？開汽車需要知道傳動軸與變速箱的原理嗎？學習 MFC 不就是要一舉超越煩瑣的 Windows API？啊，廠商（不管是哪一家）廣告給我們的印象就是，藉由視覺化的工具我們可以一步登天，基本上這個論點正確，只是有個但書：你得學會操控 Application Framework。

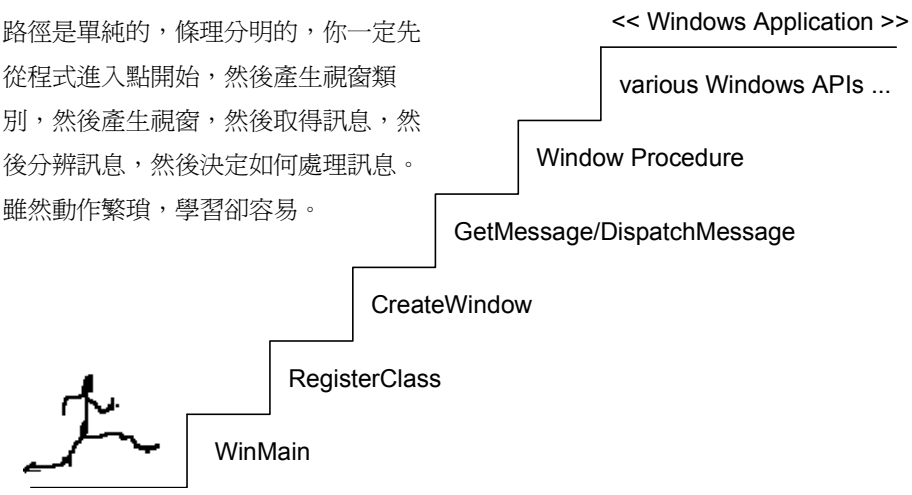
想像你擁有一部保時捷，風馳電摯風光得很，但是引擎蓋打開來全傻了眼。如果你懂汽車內部運作原理，那麼至少開車時「腳不要老是含著離合器，以免來令片磨損」這個道理背後的原理你就懂了，「踩煞車時絕不可以同時踩離合器，以免失去引擎煞車力」這個道理背後的原理你也懂了，甚至你的保時捷要保養維修時或也可以不假外力自己來。

不要把自己想像成這場遊戲中的後座車主，事實上作爲這本技術書籍的讀者的你，應該是車廠師傅。

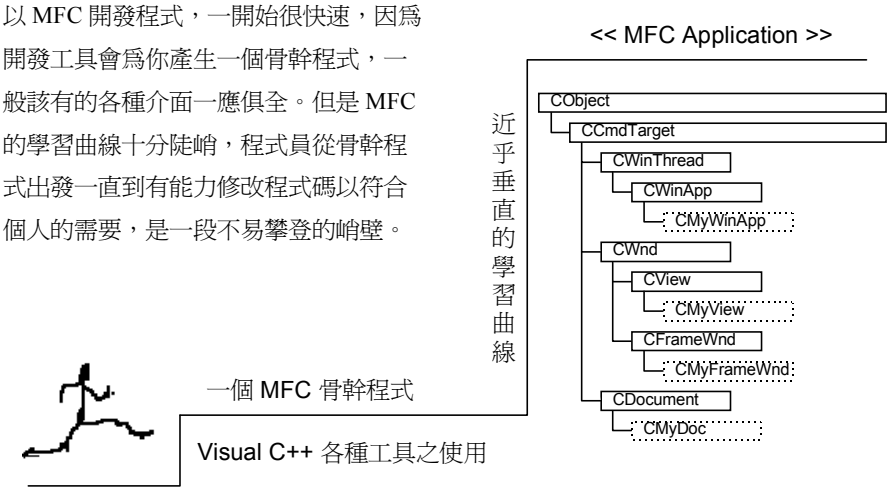
好，這個比喻不見得面面俱到，但起碼你知道了自己的身份。

題外話：我的朋友曾銘源（現在紐約工作）寫信給我說：『最近專案的壓力大，人員紛紛離職。接連一個多禮拜，天天有人上門面談。人事部門不知從哪裡找來這些阿哥，號稱有三年的 SDK/MFC 經驗，結果對起話來是雞同鴨講，*WinMain* 和 Windows Procedure 都搞不清楚。問他什麼是 message handler？只會在 ClassWizard 上 click、click、click !!! 拜 Wizard 之賜，人力市場上多出了好幾倍的 VC/MFC 程式員，但這些「Wizard 通」我們可不敢要』。

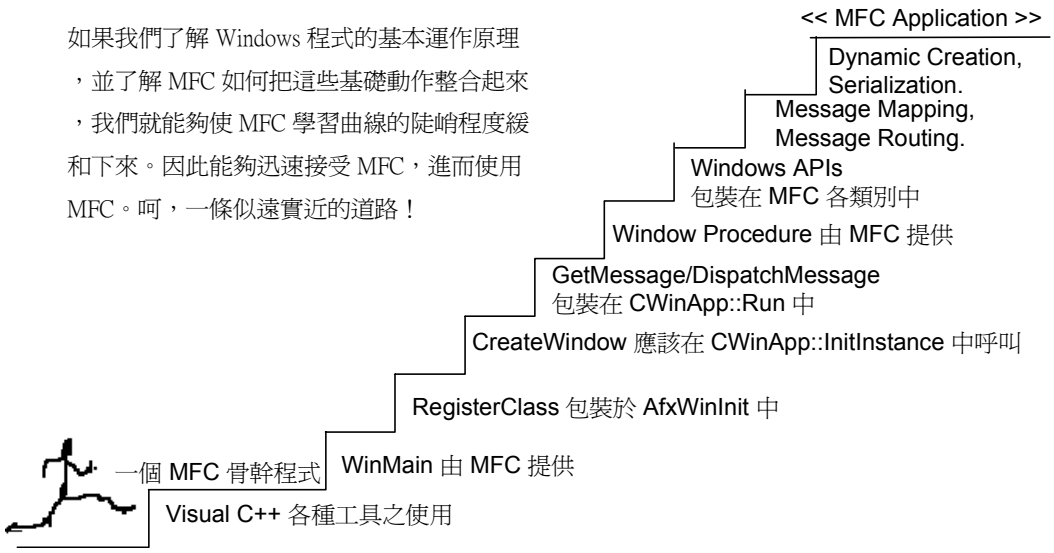
以 raw Windows API 開發程式，學習的路徑是單純的，條理分明的，你一定先從程式進入點開始，然後產生視窗類別，然後產生視窗，然後取得訊息，然後分辨訊息，然後決定如何處理訊息。雖然動作繁瑣，學習卻容易。



以 MFC 開發程式，一開始很快速，因爲開發工具會爲你產生一個骨幹程式，一般該有的各種介面一應俱全。但是 MFC 的學習曲線十分陡峭，程式員從骨幹程式出發一直到有能力修改程式碼以符合個人的需要，是一段不易攀登的峭壁。



如果我們了解 Windows 程式的基本運作原理，並了解 MFC 如何把這些基礎動作整合起來，我們就能夠使 MFC 學習曲線的陡峭程度緩和下來。因此能夠迅速接受 MFC，進而使用 MFC。呵，一條似遠實近的道路！



我希望你了解，本書之所以在各個主題中不厭其煩地挖 MFC 內部動作，解釋骨幹程式的每一條指令，每一個環節，是爲了讓你踏實地接受 MFC，進而有能力役使 MFC。你以爲這是一條遠路？呵呵，似遠實近！

不二法門：熟記 MFC 類別的階層架構

MFC 在 1.0 版時期的訴求是「一組將 SDK API 包裝得更好用的類別庫」，從 2.0 版開始更進一步訴求是一個「Application Framework」，擁有重要的 Document-View 架構；隨後又在更新版本上增加了 OLE 架構、DAO 架構...。爲了讓你有一個最輕鬆的起點，我把第一個程式簡化到最小程度，捨棄 Document-View 架構，使你能夠儘快掌握 C++/MFC 程式的面貌。這個程式並不以 AppWizard 製作出來，也不以 ClassWizard 管理維護，而是純手工打造。畢竟 Wizards 做出來的程式碼有一大堆註解，某些註解對 Wizards 有特殊意義，不能隨便刪除，卻可能會混淆初學者的視聽焦點；而且 Wizards 所產生的程式骨幹已具備 Document-View 架構，又有許多奇奇怪怪的巨集，初學者暫避爲妙。我們目前最想知道的是一個最陽春的 MFC 程式以什麼面貌呈現，以及它如何開始運作，如何結束生命。

SDK 程式設計的第一要務是了解最重要的數個 API 函式的意義和用法，像是 *RegisterClass*、*CreateWindow*、*GetMessage*、*DispatchMessage*，以及訊息的獲得與分配。MFC 程式設計的第一要務則是熟記 MFC 的類別階層架構，並清楚知曉其中幾個一定會用到的類別。本書最後面有一張 MFC 4.2 架構圖，疊床架屋，令人畏懼，我將挑出單單兩個類別，組合成一個 "Hello MFC" 程式。這兩個類別在 MFC 的地位如圖 **6-1** 所示。

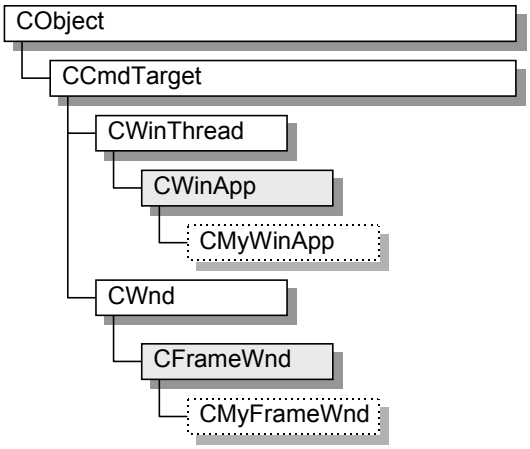


圖 6-1　本章範例程式所用到的 MFC 類別

需要什麼函式庫？

開始寫碼之前，我們得先了解程式碼以外的週邊環境。第一個必須知道的是，MFC 程式需要什麼函式庫？SDK 程式聯結時期所需的函式庫已在第一章顯示，MFC 程式一樣需要它們：

■ Windows C Runtime 函式庫（VC++ 5.0）

檔案名稱	檔案大小	說明
LIBC.LIB	898826	C Runtime 函式庫的靜態聯結版本
MSVCRT.LIB	510000	C Runtime 函式庫的動態聯結版本
MSVCRTD.LIB	803418	'D' 表示使用於 Debug 模式

＊ 這些函式庫不再區分 `Large/Medium/Small` 記憶體模式，因爲 32 位元作業系統不再有記憶體模式之分。這些函式庫的多緒版本，請參考本書 #38 頁。

■ DLL Import 函式庫（VC++ 5.0）

檔案名稱	檔案大小	說明
GDI32.LIB	307520	for GDI32.DLL（136704 bytes in Win95）
USER32.LIB	517018	for USER32.DLL（45568 bytes in Win95）
KERNEL32.LIB	635638	for KERNEL32 DLL（413696 bytes in Win95）
. . .		

此外，應用程式還需要聯結一個所謂的 MFC 函式庫，或稱爲 AFX 函式庫，它也就是 MFC 這個 application framework 的本體。你可以靜態聯結之，也可以動態聯結之，AppWizard 給你選擇權。本例使用動態聯結方式，所以需要一個對應的 MFC import 函式庫：

■ MFC 函式庫（AFX 函式庫)（VC++ 5.0，MFC 4.2）

檔案名稱	檔案大小	說明
MFC42.LIB	4200034	MFC42.DLL（941840 bytes）的 import 函式庫。
MFC42D.LIB	3003766	MFC42D.DLL（1393152 bytes）的 import 函式庫。
MFCS42.LIB	168364	
MFCS42D.LIB	169284	
MFCN42D.LIB	91134	
MFCD42D.LIB	486334	
MFCO42D.LIB	2173082	
. . .		

我們如何在聯結器（link.exe）中設定選項，把這些函式庫都聯結起來？稍後在 HELLO.MAK 中可以一窺全貌。

如果在 Visual C++ 整合環境中工作，這些設定不勞你自己動手，整合環境會根據我們圈選的項目自動做出一個合適的 makefile。這些 makefile 的內容看起來非常詰屈聱牙，事實上我們也不必太在意它，因爲那是整合環境的工作。這一章我不打算依賴任何開發工具，一切自己來，你會在稍後看到一個簡潔清爽的 makefile。

需要什麼含入檔？

SDK 程式只要含入 WINDOWS.H 就好，所有 API 的函式宣告、訊息定義、常數定義、巨集定義、都在 WINDOWS.H 檔中。除非程式另呼叫了作業系統提供的新模組（如 CommDlg、ToolHelp、DDEML...），才需要再各別含入對應的 .H 檔。

> WINDOWS.H 過去是一個巨大檔案，大約在 5000 行上下。現在已拆分內容爲數十個較小的 .H 檔，再由 WINDOWS.H 含入進來。也就是說它變成一個 "Master included file for Windows applications"。

MFC 程式不這麼單純，下面是它常常需要面對的另外一些 .H 檔：

- STDAFX.H - 這個檔案用來做爲 Precompiled header file（請看稍後的方塊說明），其內只是含入其他的 MFC 表頭檔。應用程式通常會準備自己的 STDAFX.H，例如本章的 Hello 程式就在 STDAFX.H 中含入 AFXWIN.H。
- AFXWIN.H - 每一個 Windows MFC 程式都必須含入它，因爲它以及它所含入的檔案宣告了所有的 MFC 類別。此檔內含 AFX.H，後者又含入 AFXVER_.H，後者又含入 AFXV_W32.H，後者又含入 WINDOWS.H（啊呼，終於現身）。
- AFXEXT.H - 凡使用工具列、狀態列之程式必須含入這個檔案。
- AFXDLGS.H - 凡使用通用型對話盒（Common Dialog）之 MFC 程式需含入此檔，其內部含入 COMMDLG.H。
- AFXCMN.H - 凡使用 Windows 95 新增之通用型控制元件（Common Control）之 MFC 程式需含入此檔。
- AFXCOLL.H - 凡使用 Collections Classes（用以處理資料結構如陣列、串列）之程式必須含入此檔。
- AFXDLLX.H - 凡 MFC extension DLLs 均需含入此檔。
- AFXRES.H - MFC 程式的 RC 檔必須含入此檔。MFC 對於標準資源（例如 File、Edit 等選單項目）的 ID 都有預設值，定義於此檔中，例如：

```
// File commands
#define ID_FILE_NEW        0xE100
#define ID_FILE_OPEN       0xE101
#define ID_FILE_CLOSE      0xE102
#define ID_FILE_SAVE       0xE103
#define ID_FILE_SAVE_AS    0xE104
...
// Edit commands
#define ID_EDIT_COPY       0xE122
#define ID_EDIT_CUT        0xE123
...
```

這些選單項目都有預設的說明文字（將出現在狀態列中），但說明文字並不會事先定義於此檔，AppWizard 爲我們製作骨幹程式時才把說明文字加到應用程式的 RC 檔中。第４章的骨幹程式 Scribble step0 的 RC 檔中就有這樣的字串表格：

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_NEW     "Create a new document"
  ID_FILE_OPEN    "Open an existing document"
  ID_FILE_CLOSE   "Close the active document"
  ID_FILE_SAVE    "Save the active document"
  ID_FILE_SAVE_AS  "Save the active document with a new name"
  ...
  ID_EDIT_COPY "Copy the selection and puts it on the Clipboard"
  ID_EDIT_CUT  "Cut the selection and puts it on the Clipboard"
  ...
END
```

所有 MFC 含入檔均置於 \MSVC\MFC\INCLUDE 中。這些檔案連同 Windows SDK 的含入檔 WINDOWS.H、COMMDLG.H、TOOLHELP.H、DDEML.H... 每每在編譯過程中耗費大量的時間，因此你絕對有必要設定 Precompiled header。

Precompiled Header
一個典型程式在發展過程中常常要不斷地編譯。Windows 程式會以的標準 .H 檔案非常巨大但內容不變，編譯器浪費在這上面的時間非常多。Precompiled header 就是將 .H 檔第一次編譯後的結果貯存起來，第二次再編譯時就可以直接從硬碟中取出來用。這種觀念在 Borland C/C++ 早已行之，Microsoft 這邊則是一直到 Visual C++ 1.0 才具備。

簡化的 MFC 程式架構－以 Hello MFC 為例

現在我們正式進入 MFC 程式設計。由於 Document/View 架構複雜，不適合初學者，所以我先把它略去。這裡所提的程式觀念是一般的 MFC Application Framework 的子集合。

本章程式名爲 Hello，執行時會在視窗中從天而降 "Hello, MFC" 字樣。Hello 是一個非常簡單而具代表性的程式，它的代表性在於：

- 每一個 MFC 程式都想從 MFC 中衍生出適當的類別來用（不然又何必以 MFC 寫程式呢），其中兩個不可或缺的類別 *CWinApp* 和 *CFrameWnd* 在 Hello 程式中會表現出來，它們的意義如**圖 6-2**。
- MFC 類別中某些函式一定得被應用程式改寫（例如 *CWinApp*::*InitInstance*），這在 Hello 程式中也看得到。
- 選單和對話盒，Hello 也都具備。

圖 6-3 是 Hello 原始檔案的組成。第一次接觸 MFC 程式，我們常常因爲不熟悉 MFC 的類別分類、類別命名規則，以至於不能在腦中形成具體印象，於是細部討論時各種資訊及說明彷如過眼雲煙。相信我，你必須多看幾次，並且用心熟記 MFC 命名規則。

圖 6-3 之後是 Hello 程式的原始碼。由於 MFC 已經把 Windows API 都包裝起來了，原始碼再也不能夠「說明一切」。你會發現 MFC 程式很有點見林不見樹的味道：

- 看不到 *WinMain*，因此不知程式從哪裡開始執行。
- 看不到 *RegisterClass* 和 *CreateWindow*，那麼視窗是如何做出來的呢？
- 看不到 Message Loop（*GetMessage*/*DispatchMessage*），那麼程式如何推動？
- 看不到 Window Procedure，那麼視窗如何運作？

我的目的就在鏟除這些困惑。

Hello 程式原始碼

- HELLO.MAK - makefile
- RESOURCE.H - 所有資源 ID 都在這裡定義。本例只定義一個 IDM_ABOUT。
- JJHOUR.ICO - 圖示檔，用於主視窗和對話盒。
- HELLO.RC - 資源描述檔。本例有一份選單、一個圖示、和一個對話盒。
- STDAFX.H - 含入 AFXWIN.H。
- STDAFX.CPP - 含入 STDAFX.H，爲的是製造出 Precompiled header。
- HELLO.H - 宣告 *CMyWinApp* 和 *CMyFrameWnd*。
- HELLO.CPP - 定義 *CMyWinApp* 和 *CMyFrameWnd*。

注意：沒有模組定義檔 .DEF？是的，如果你不指定模組定義檔，聯結器就使用預設值。

圖 6-2　Hello 程式中的兩個物件

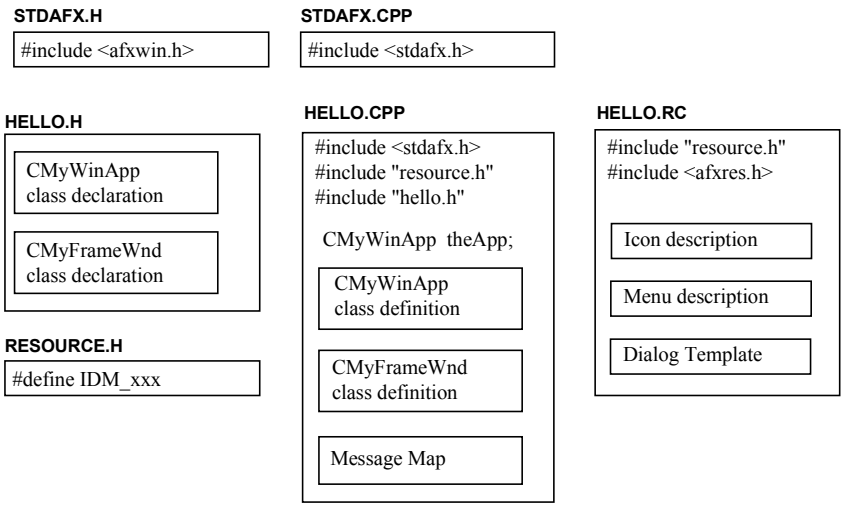


圖 6-3　Hello 程式的基本檔案架構。一般習慣爲每個類別準備一個 .H（宣告）和一個 .CPP（實作），本例把兩類別集中在一起是爲了簡化。

HELLO.MAK（請在 DOS 視窗中執行 nmake hello.mak。環境設定請參考 p.224）

```
#0001  # filename : hello.mak
#0002  # make file for hello.exe (MFC 4.0 Application)
#0003  # usage : nmake hello.mak (Visual C++ 5.0)
#0004
#0005  Hello.exe : StdAfx.obj Hello.obj Hello.res
#0006    link.exe /nologo /subsystem:windows /incremental:no \
#0007            /machine:I386 /out:"Hello.exe" \
#0008            Hello.obj StdAfx.obj Hello.res \
#0009            msvcrt.lib kernel32.lib user32.lib gdi32.lib mfc42.lib
#0010
#0011  StdAfx.obj : StdAfx.cpp StdAfx.h
#0012    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0013          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yc"stdafx.h"  \
#0014          /c StdAfx.cpp
#0015
#0016  Hello.obj : Hello.cpp Hello.h StdAfx.h
#0017    cl.exe /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" \
#0018          /D "_AFXDLL" /D "_MBCS" /Fp"Hello.pch" /Yu"stdafx.h"  \
#0019          /c Hello.cpp
#0020
#0021  Hello.res : Hello.rc Hello.ico jjhour.ico
#0022    rc.exe /l 0x404 /Fo"Hello.res" /D "NDEBUG" /D "_AFXDLL" Hello.rc
```

RESOURCE.H

```
#0001  // resource.h
#0002  #define IDM_ABOUT   100
```

HELLO.RC

```
#0001  // hello.rc
#0002  #include "resource.h"
#0003  #include "afxres.h"
#0004
#0005  JJHouRIcon        ICON  DISCARDABLE  "JJHOUR.ICO"
#0006  AFX_IDI_STD_FRAME  ICON  DISCARDABLE  "JJHOUR.ICO"
#0007
#0008  MainMenu MENU DISCARDABLE
#0009  {
#0010      POPUP "&Help"
#0011      {
#0012        MENUITEM "&About HelloMFC...", IDM_ABOUT
#0013      }
#0014  }
#0015
#0016  AboutBox DIALOG DISCARDABLE  34, 22, 147, 55
#0017  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
#0018  CAPTION "About Hello"
#0019  {
#0020    ICON         "JJHouRIcon",IDC_STATIC,11,17,18,20
#0021    LTEXT        "Hello MFC 4.0",IDC_STATIC,40,10,52,8
#0022    LTEXT        "Copyright 1996 Top Studio",IDC_STATIC,40,25,100,8
#0023    LTEXT        "J.J.Hou",IDC_STATIC,40,40,100,8
#0024    DEFPUSHBUTTON "OK",IDOK,105,7,32,14,WS_GROUP
#0025  }
```

STDAFX.H

```
#0001  // stdafx.h : include file for standard system include files,
#0002  // or project specific include files that are used frequently,
#0003  // but are changed infrequently
#0004
#0005  #include <afxwin.h>   // MFC core and standard components
```

STDAFX.CPP

```
#0001  // stdafx.cpp : source file that includes just the standard includes
#0002  //     Hello.pch will be the pre-compiled header
#0003  //     stdafx.obj will contain the pre-compiled type information
#0004
#0005  #include "stdafx.h"
```

HELLO.H

```
#0001  //----------------------------------------------------------------
#0002  //            MFC 4.0  Hello Sample Program
#0003  //       Copyright (c) 1996   Top Studio * J.J.Hou
#0004  //
#0005  // 檔名     : hello.h
#0006  // 作者     : 侯俊傑
#0007  // 編譯聯結  : 請參考 hello.mak
#0008  //
#0009  // 宣告 Hello 程式的兩個類別 : CMyWinApp 和 CMyFrameWnd
#0010  //----------------------------------------------------------------
#0011
#0012  class CMyWinApp : public CWinApp
#0013  {
#0014  public:
#0015     BOOL InitInstance();  // 每一個應用程式都應該改寫此函式
#0016  };
#0017
#0018  //----------------------------------------------------------------
#0019  class CMyFrameWnd : public CFrameWnd
#0020  {
#0021  public:
#0022     CMyFrameWnd();          // constructor
#0023     afx_msg void OnPaint();   // for WM_PAINT
#0024     afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
#0025
#0026  private:
#0027     DECLARE_MESSAGE_MAP()    // Declare Message Map
#0028     static VOID CALLBACK LineDDACallback(int,int,LPARAM);
#0029     // 注意: callback 函式必須是 "static"，才能去除隱藏的 'this' 指標。
#0030  };
```

HELLO.CPP

```
#0001  //---------------------------------------------------------------
#0002  //              MFC 4.0  Hello sample program
#0003  //       Copyright (c) 1996   Top Studio * J.J.Hou
#0004  //
#0005  // 檔名      : hello.cpp
#0006  // 作者      : 侯俊傑
#0007  // 編譯聯結  : 請參考 hello.mak
#0008  //
#0009  // 本例示範最簡單之 MFC 應用程式，不含 Document/View 架構。程式每收到
#0010  // WM_PAINT 即利用 GDI 函式 LineDDA() 讓 "Hello, MFC" 字串從天而降。
#0011  //---------------------------------------------------------------
#0012  #include "Stdafx.h"
#0013  #include "Hello.h"
#0014  #include "Resource.h"
#0015
#0016  CMyWinApp theApp;   // application object
#0017
#0018  //---------------------------------------------------------------
#0019  // CMyWinApp's member
#0020  //---------------------------------------------------------------
#0021  BOOL CMyWinApp::InitInstance()
#0022  {
#0023     m_pMainWnd = new CMyFrameWnd();
#0024     m_pMainWnd->ShowWindow(m_nCmdShow);
#0025     m_pMainWnd->UpdateWindow();
#0026     return TRUE;
#0027  }
#0028  //---------------------------------------------------------------
#0029  // CMyFrameWnd's member
#0030  //---------------------------------------------------------------
#0031  CMyFrameWnd::CMyFrameWnd()
#0032  {
#0033     Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault,
#0034           NULL, "MainMenu");     // "MainMenu" 定義於 RC 檔
#0035  }
#0036  //---------------------------------------------------------------
#0037  BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0038     ON_COMMAND(IDM_ABOUT, OnAbout)
#0039     ON_WM_PAINT()
#0040  END_MESSAGE_MAP()
#0041  //---------------------------------------------------------------
#0042  void CMyFrameWnd::OnPaint()
#0043  {
#0044  CPaintDC dc(this);
```

```
#0045  CRect rect;
#0046
#0047     GetClientRect(rect);
#0048
#0049     dc.SetTextAlign(TA_BOTTOM | TA_CENTER);
#0050
#0051     ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
#0052        (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
#0053  }
#0054  //---------------------------------------------------------------
#0055  VOID CALLBACK CMyFrameWnd::LineDDACallback(int x, int y, LPARAM lpdc)
#0056  {
#0057  static char szText[] = "Hello, MFC";
#0058
#0059     ((CDC*)lpdc)->TextOut(x, y, szText, sizeof(szText)-1);
#0060     for(int i=1; i<50000; i++);  // 純粹是爲了延遲下降速度，以利觀察
#0061  }
#0062  //---------------------------------------------------------------
#0063  void CMyFrameWnd::OnAbout()
#0064  {
#0065     CDialog about("AboutBox", this);  // "AboutBox" 定義於 RC 檔
#0066     about.DoModal();
#0067  }
```

上面這些程式碼中，你看到了一些 MFC 類別如 *CWinApp* 和 *CFrameWnd*，一些 MFC 資料型態如 *BOOL* 和 *VOID*，一些 MFC 巨集如 *DECLARE_MESSAGE_MAP* 和 *BEGIN_MESSAGE_END* 和 *END_MESSAGE_MAP*。這些都曾經在第5章的「縱覽 MFC」一節中露過臉。但是單純從 C++ 語言的角度來看，還有一些是我們不能理解的，如 HELLO.H 中的 *afx_msg*（#23 行）和 *CALLBACK*（#28 行）。

你可以在 WINDEF.H 中發現 *CALLBACK* 的意義：

```
#define CALLBACK  __stdcall  // 一種函式呼叫習慣
```

可以在 AFXWIN.H 中發現 *afx_msg* 的意義：

```
#define afx_msg   // intentional placeholder
                  // 故意安排的一個空位置。也許以後版本會用到。
```

MFC 程式的來龍去脈（causal relations）

讓我們從第1章的 C/SDK 觀念出發，看看 MFC 程式如何運作。

第一件事情就是找出 MFC 程式的進入點。MFC 程式也是 Windows 程式，所以它應該也有一個 *WinMain*，但是我們在 Hello 程式看不到它的蹤影。是的，但先別急，在程式進入點之前，更有一個（而且僅有一個）全域物件（本例名爲 *theApp*），這是所謂的 application object，當作業系統將程式載入並啓動，這個全域物件獲得配置，其建構式會先執行，比 *WinMain* 更早。所以以時間順序來說，我們先看看這個 application object。

我只做了兩個小小的類別：CWinApp 和 CFrameWnd

你已經看過了**圖 6-2**，作爲一個最最粗淺的 MFC 程式，Hello 是如此單純，只有一個視窗。回想第一章 Generic 程式的寫法，其主體在於 *WinMain* 和 *WndProc*，而這兩個部份其實都有相當程度的不變性。好極了，MFC 就把有著相當固定行爲之 *WinMain* 內部動作包裝在 *CWinApp* 中，把有著相當固定行爲之 *WndProc* 內部動作包裝在 *CFrameWnd* 中。也就是說：

- *CWinApp* 代表程式本體
- *CFrameWnd* 代表一個主框視窗（Frame Window）

但雖然我說，*WinMain* 內部動作和 *WndProc* 內部動作都有著相當程度的固定行爲，它們畢竟需要面對不同應用程式而有某種變化。所以，你必須以這兩個類別爲基礎，衍生自己的類別，並改寫其中一部份成員函式。

```
class CMyWinApp : public CWinApp
{
    ...
};
class CMyFrameWnd : public CFrameWnd
{
    ...
};
```

本章對衍生類別的命名規則是：在基礎類別名稱的前面加上 "*My*"。這種規則真正上戰場時不見得適用，大型程式可能會自同一個基礎類別衍生出許多自己的類別。不過以教學目的而言，這種命名方式使我們從字面就知道類別之間的從屬關係，頗爲理想（根據我的經驗，初學者會被類別的命名搞得頭昏腦脹）。

CWinApp－取代 WinMain 的地位

CWinApp 的衍生物件被稱爲 application object，可以想見，*CWinApp* 本身就代表一個程式本體。一個程式的本體是什麼？回想第 1 章的 SDK 程式，與程式本身有關而不與視窗有關的資料或動作有些什麼？系統傳進來的四個 *WinMain* 參數算不算？*InitApplication* 和 *InitInstance* 算不算？訊息迴路算不算？都算，是的，以下是 MFC 4.x 的 *CWinApp* 宣告（節錄自 AFXWIN.H）：

```
class CWinApp : public CWinThread
{
// Attributes
   // Startup args (do not change)
   HINSTANCE m_hInstance;
   HINSTANCE m_hPrevInstance;
   LPTSTR m_lpCmdLine;
   int m_nCmdShow;

   // Running args (can be changed in InitInstance)
   LPCTSTR m_pszAppName;  // human readable name
   LPCTSTR m_pszRegistryKey;   // used for registry entries

public:  // set in constructor to override default
   LPCTSTR m_pszExeName;      // executable name (no spaces)
   LPCTSTR m_pszHelpFilePath;  // default based on module path
   LPCTSTR m_pszProfileName;   // default based on app name

public:
   // hooks for your initialization code
   virtual BOOL InitApplication();

   // overrides for implementation
   virtual BOOL InitInstance();
   virtual int ExitInstance();
   virtual int Run();
   virtual BOOL OnIdle(LONG lCount);
...
};
```

幾乎可以說 *CWinApp* 用來取代 *WinMain* 在 SDK 程式中的地位。這並不是說 MFC 程式沒有 *WinMain*（稍後我會解釋），而是說傳統上 SDK 程式的 *WinMain* 所完成的工作現在由 *CWinApp* 的三個函式完成：

```
virtual BOOL InitApplication();
virtual BOOL InitInstance();
virtual int  Run();
```

WinMain 只是扮演役使它們的角色。

會不會覺得 *CWinApp* 的成員變數中少了點什麼東西？是不是應該有個成員變數記錄主視窗的 handle（或是主視窗對應之 C++ 物件）？的確，在 MFC 2.5 中的確有 *m_pMainWnd* 這麼個成員變數（以下節錄自 MFC 2.5 的 AFXWIN.H）：

```
class CWinApp : public CCmdTarget
{
// Attributes
   // Startup args (do not change)
   HINSTANCE m_hInstance;
   HINSTANCE m_hPrevInstance;
   LPSTR m_lpCmdLine;
   int m_nCmdShow;

   // Running args (can be changed in InitInstance)
   CWnd* m_pMainWnd;         // main window (optional)
   CWnd* m_pActiveWnd;       // active main window (may not be m_pMainWnd)
   const char* m_pszAppName;   // human readable name

public:  // set in constructor to override default
   const char* m_pszExeName;      // executable name (no spaces)
   const char* m_pszHelpFilePath;  // default based on module path
   const char* m_pszProfileName;   // default based on app name

public:
   // hooks for your initialization code
   virtual BOOL InitApplication();
   virtual BOOL InitInstance();

   // running and idle processing
   virtual int Run();
   virtual BOOL OnIdle(LONG lCount);
```

```
    // exiting
    virtual int ExitInstance();
...
};
```

但從 MFC 4.x 開始，*m_pMainWnd* 已經被移往 *CWinThread* 中了（它是 *CWinApp* 的父類別）。以下內容節錄自 MFC 4.x 的 AFXWIN.H：

```
class CWinThread : public CCmdTarget
{
// Attributes
    CWnd* m_pMainWnd;     // main window (usually same AfxGetApp()->m_pMainWnd)
    CWnd* m_pActiveWnd;   // active main window (may not be m_pMainWnd)

    // only valid while running
    HANDLE m_hThread;      // this thread's HANDLE
    DWORD m_nThreadID;     // this thread's ID

    int GetThreadPriority();
    BOOL SetThreadPriority(int nPriority);

// Operations
    DWORD SuspendThread();
    DWORD ResumeThread();

// Overridables
    // thread initialization
    virtual BOOL InitInstance();

    // running and idle processing
    virtual int Run();
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL PumpMessage();    // low level message pump
    virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing

public:
    // valid after construction
    AFX_THREADPROC m_pfnThreadProc;
...
};
```

熟悉 Win32 的朋友，看到 *CWinThread* 類別之中的 *SuspendThread* 和 *ResumeThread* 成員函式，可能會發出會心微笑。

CFrameWnd－取代 WndProc 的地位

CFrameWnd 主要用來掌握一個視窗，幾乎你可以說它是用來取代 SDK 程式中的視窗函式的地位。傳統的 SDK 視窗函式寫法是：

```
long FAR PASCAL WndProc(HWND hWnd, UINT msg, WORD wParam, LONG lParam)
{
    switch(msg) {
        case WM_COMMAND :
            switch(wParam) {
               case IDM_ABOUT :
                    OnAbout(hWnd, wParam, lParam);
                    break;
            }
            break;
        case WM_PAINT :
            OnPaint(hWnd, wParam, lParam);
            break;
        default :
            DefWindowProc(hWnd, msg, wParam, lParam);
    }
}
```

MFC 程式有新的作法，我們在 Hello 程式中也爲 *CMyFrameWnd* 準備了兩個訊息處理常式，宣告如下：

```
class CMyFrameWnd : public CFrameWnd
{
public:
        CMyFrameWnd();
        afx_msg void OnPaint();
        afx_msg void OnAbout();
        DECLARE_MESSAGE_MAP()
};
```

OnPaint 處理什麼訊息？*OnAbout* 又是處理什麼訊息？我想你很容易猜到，前者處理 *WM_PAINT*，後者處理 *WM_COMMAND* 的 *IDM_ABOUT*。這看起來十分俐落，但讓人搞不懂來龍去脈。程式中是不是應該有「把訊息和處理函式關聯在一起」的設定動作？是的，這些設定在 HELLO.CPP 才看得到。但讓我先著一鞭：*DECLARE_MESSAGE_MAP* 巨集與此有關。

這種寫法非常奇特，原因是 MFC 內建了一個所謂的 Message Map 機制，會把訊息自動送到「與訊息對映之特定函式」去；訊息與處理函式之間的對映關係由程式員指定。*DECLARE_MESSAGE_MAP* 另搭配其他巨集，就可以很便利地將訊息與其處理函式關聯在一起：

```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_WM_PAINT()
   ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

稍後我就來探討這些神秘的巨集。

引爆器－Application object

我們已經看過 HELLO.H 宣告的兩個類別，現在把目光轉到 HELLO.CPP 身上。這個檔案將兩個類別實作出來，並產生一個所謂的 application object。故事就從這裡展開。

下面這張圖包括右半部的 Hello 原始碼與左半部的 MFC 原始碼。從這一節以降，我將以此圖解釋 MFC 程式的啓動、運行、與結束。不同小節的圖將標示出當時的程式進行狀況。

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

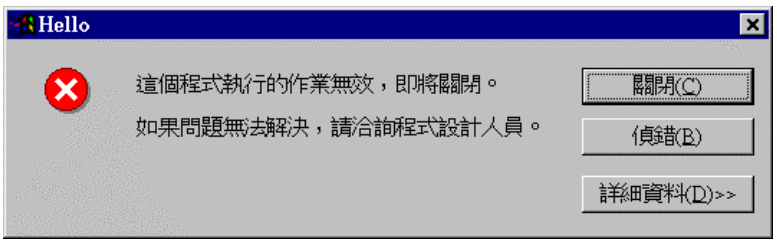
HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

上圖的 *theApp* 就是 Hello 程式的 application object，每一個 MFC 應用程式都有一個，而且也只有這麼一個。當你執行 Hello，這個全域物件產生，於是建構式執行起來。我們並沒有定義 *CMyWinApp* 建構式；至於其父類別 *CWinApp* 的建構式內容摘要如下（摘錄自 APPCORE.CPP）：

```
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    m_pszAppName = lpszAppName;

    // initialize CWinThread state
    AFX_MODULE_THREAD_STATE* pThreadState = AfxGetModuleThreadState();
    pThreadState->m_pCurrentWinThread = this;
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();

    // initialize CWinApp state
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    pModuleState->m_pCurrentWinApp = this;

    // in non-running state until WinMain
    m_hInstance = NULL;
    m_pszHelpFilePath = NULL;
    m_pszProfileName = NULL;
    m_pszRegistryKey = NULL;
    m_pszExeName = NULL;
    m_lpCmdLine = NULL;
    m_pCmdInfo = NULL;
    ...
}
```

CWinApp 之中的成員變數將因爲 *theApp* 這個全域物件的誕生而獲得配置與初值。如果程式中沒有 *theApp* 存在，編譯聯結還是可以順利通過，但執行時會出現系統錯誤訊息：



隱藏不明的 WinMain

HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

theApp 配置完成後，*WinMain* 登場。我們並未撰寫 *WinMain* 程式碼，這是 MFC 早已準備好並由聯結器直接加到應用程式碼中的，其原始碼列於圖 **6-4**。*_tWinMain* 函式的 ¡§¡¨是爲了支援 Unicode 而準備的一個巨集。

```
// in APPMODUL.CPP
extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPTSTR lpCmdLine, int nCmdShow)
{
       // call shared/exported WinMain
       return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

此外，在 DLLMODUL.CPP 中有一個 *DllMain* 函式。本書並未涵蓋 DLL 程式設計。

```
// in WINMAIN.CPP
#0001 ////////////////////////////////////////////////////////////////
#0002 // Standard WinMain implementation
#0003 //  Can be replaced as long as 'AfxWinInit' is called first
#0004
#0005 int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0006         LPTSTR lpCmdLine, int nCmdShow)
#0007 {
#0008     ASSERT(hPrevInstance == NULL);
#0009
#0010     int nReturnCode = -1;
#0011     CWinApp* pApp = AfxGetApp();
#0012
#0013     // AFX internal initialization
#0014     if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
#0015         goto InitFailure;
#0016
#0017     // App global initializations (rare)
#0018     ASSERT_VALID(pApp);
#0019     if (!pApp->InitApplication())
#0020         goto InitFailure;
#0021     ASSERT_VALID(pApp);
#0022
#0023     // Perform specific initializations
#0024     if (!pApp->InitInstance())
#0025     {
#0026         if (pApp->m_pMainWnd != NULL)
#0027         {
#0028             TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
#0029             pApp->m_pMainWnd->DestroyWindow();
#0030         }
#0031         nReturnCode = pApp->ExitInstance();
#0032         goto InitFailure;
#0033     }
#0034     ASSERT_VALID(pApp);
#0035
#0036     nReturnCode = pApp->Run();
#0037     ASSERT_VALID(pApp);
#0038
#0039 InitFailure:
#0040
#0041     AfxWinTerm();
#0042     return nReturnCode;
#0043 }
```

圖 6-4　Windows 程式進入點。原始碼可從 MFC 的 WINMAIN.CPP 中獲得。

稍加整理去蕪存菁，就可以看到這個「程式進入點」主要做些什麼事：

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
    int nReturnCode = -1;
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
    return nReturnCode;
}
```

其中，*AfxGetApp* 是一個全域函式，定義於 AFXWIN1.INL 中：

```
_AFXWIN_INLINE CWinApp* AFXAPI AfxGetApp()
       { return afxCurrentWinApp; }
```

而 *afxCurrentWinApp* 又定義於 AFXWIN.H 中：

```
#define afxCurrentWinApp  AfxGetModuleState()->m_pCurrentWinApp
```

再根據稍早所述 *CWinApp*::*CWinApp* 中的動作，我們於是知道，*AfxGetApp* 其實就是取得 *CMyWinApp* 物件指標。所以，*AfxWinMain* 中這樣的動作：

```
CWinApp* pApp = AfxGetApp();
pApp->InitApplication();
pApp->InitInstance();
nReturnCode = pApp->Run();
```

其實就相當於呼叫：

```
CMyWinApp::InitApplication();
CMyWinApp::InitInstance();
CMyWinApp::Run();
```

因而導至呼叫：

```
CWinApp::InitApplication();  // 因爲 CMyWinApp 並沒有改寫 InitApplication
CMyWinApp::InitInstance();   // 因爲 CMyWinApp 改寫了 InitInstance
CWinApp::Run();              // 因爲 CMyWinApp 並沒有改寫 Run
```

根據第 1 章 SDK 程式設計的經驗推測，*InitApplication* 應該是註冊視窗類別的場所？*InitInstance* 應該是產生視窗並顯示視窗的場所？*Run* 應該是攫取訊息並分派訊息的場所？有對有錯！以下數節我將實際帶你看看 MFC 的原始碼，如此一來就可以了解隱藏在 MFC 背後的玄妙了。我的終極目標並不在 MFC 原始碼（雖然那的確是學習設計一個 application framework 的好教材），我只是想拿把刀子把 MFC 看似朦朧的內部運作來個大解剖，挑出其經脈；有這種紮實的根基，使用 MFC 才能知其然並知其所以然。下面小節分別討論 *AfxWinMain* 的四個主要動作以及引發的行爲。

AfxWinInit－AFX 內部初始化動作

HELLO.CPP

```
❶ CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

❷   AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

我想你已經清楚看到了，*AfxWinInit* 是繼 *CWinApp* 建構式之後的第一個動作。以下是它的動作摘要（節錄自 APPINIT.CPP）：

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
   ASSERT(hPrevInstance == NULL);

   // set resource handles
   AFX_MODULE_STATE* pState = AfxGetModuleState();
   pState->m_hCurrentInstanceHandle = hInstance;
   pState->m_hCurrentResourceHandle = hInstance;

   // fill in the initial state for the application
   CWinApp* pApp = AfxGetApp();
   if (pApp != NULL)
   {
      // Windows specific initialization (not done if no CWinApp)
```

```
        pApp->m_hInstance = hInstance;
        pApp->m_hPrevInstance = hPrevInstance;
        pApp->m_lpCmdLine = lpCmdLine;
        pApp->m_nCmdShow = nCmdShow;
        pApp->SetCurrentHandles();
    }

    // initialize thread specific data (for main thread)
    if (!afxContextIsDLL)
        AfxInitThread();

    return TRUE;
}
```

其中呼叫的 *AfxInitThread* 函式的動作摘要如下（節錄自 THRDCORE.CPP）：

```
void AFXAPI AfxInitThread()
{
    if (!afxContextIsDLL)
    {
        // attempt to make the message queue bigger
        for (int cMsg = 96; !SetMessageQueue(cMsg) && (cMsg -= 8); )
            ;

        // set message filter proc
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        ASSERT(pThreadState->m_hHookOldMsgFilter == NULL);
        pThreadState->m_hHookOldMsgFilter = ::SetWindowsHookEx(WH_MSGFILTER,
            _AfxMsgFilterHook, NULL, ::GetCurrentThreadId());

        // intialize CTL3D for this thread
        _AFX_CTL3D_STATE* pCtl3dState = _afxCtl3dState;
        if (pCtl3dState->m_pfnAutoSubclass != NULL)
            (*pCtl3dState->m_pfnAutoSubclass)(AfxGetInstanceHandle());

        // allocate thread local _AFX_CTL3D_THREAD just for automatic termination
        _AFX_CTL3D_THREAD* pTemp = _afxCtl3dThread;
    }
}
```

如果你曾經看過本書前身 **Visual C++ 物件導向 MFC 程式設計**，我想你可能對這句話印象深刻：「*WinMain* 一開始即呼叫 *AfxWinInit*，註冊四個視窗類別」。這是一個已成昨日黃花的事實。MFC 的確會爲我們註冊四個視窗類別，但不再是在 *AfxWinInit* 中完成。稍後我會把註冊動作挖出來，那將是視窗誕生前一刻的行爲。

CWinApp::InitApplication

HELLO.CPP

```
CMyWinApp theApp;   // application object

BOOL CMyWinApp::InitInstance()
{
   m_pMainWnd = new CMyFrameWnd();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
   Create(NULL, "Hello MFC", ...,
          "MainMenu");
}

void CMyFrameWnd::OnPaint()  { ... }
void CMyFrameWnd::OnAbout()  { ... }

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
   ON_COMMAND(IDM_ABOUT, OnAbout)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

WINMAIN.CPP

```
int AFXAPI AfxWinMain (...)
{
    CWinApp* pApp = AfxGetApp();

    AfxWinInit(...);

    pApp->InitApplication();
    pApp->InitInstance();
    nReturnCode = pApp->Run();

    AfxWinTerm();
}
```

AfxWinInit 之後的動作是 *pApp->InitApplication*。稍早我說過了，*pApp* 指向 *CMyWinApp* 物件（也就是本例的 *theApp*），所以，當程式呼叫：

```
pApp->InitApplication();
```

相當於呼叫：

```
CMyWinApp::InitApplication();
```

但是你要知道，*CMyWinApp* 繼承自 *CWinApp*，而 *InitApplication* 又是 *CWinApp* 的一個虛擬函式；我們並沒有改寫它（大部份情況下不需改寫它），所以上述動作相當於呼叫：

```
CWinApp::InitApplication();
```

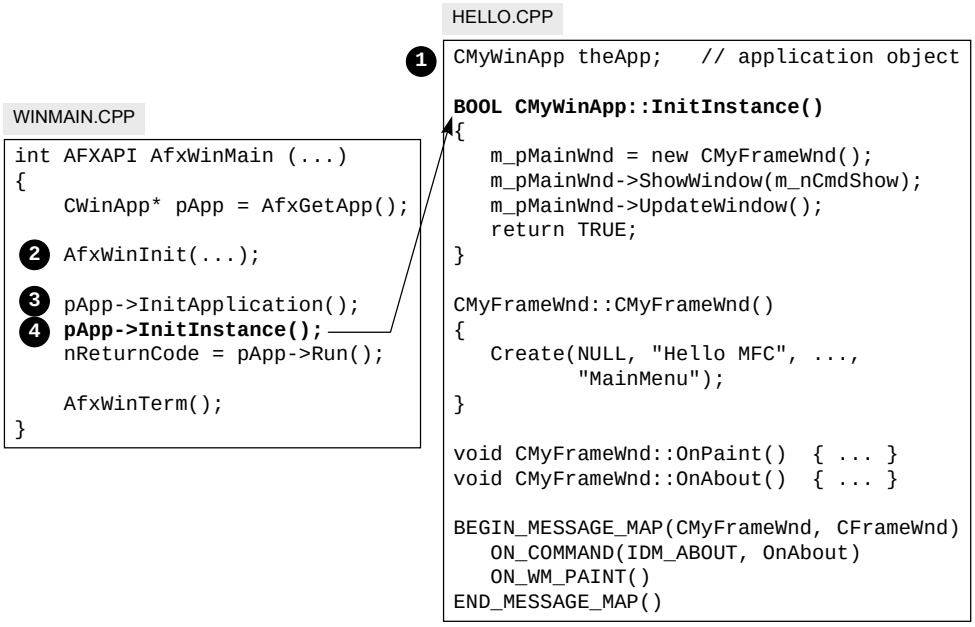
此函式之原始碼出現在 APPCORE.CPP 中：

```
BOOL CWinApp::InitApplication()
{
    if (CDocManager::pStaticDocManager != NULL)
    {
        if (m_pDocManager == NULL)
            m_pDocManager = CDocManager::pStaticDocManager;
        CDocManager::pStaticDocManager = NULL;
    }

    if (m_pDocManager != NULL)
        m_pDocManager->AddDocTemplate(NULL);
    else
        CDocManager::bStaticInit = FALSE;

    return TRUE;
}
```

這些動作都是 MFC 爲了內部管理而做的。

關於 Document Template 和 *CDocManager*，第7章和第8章另有說明。

CMyWinApp::InitInstance



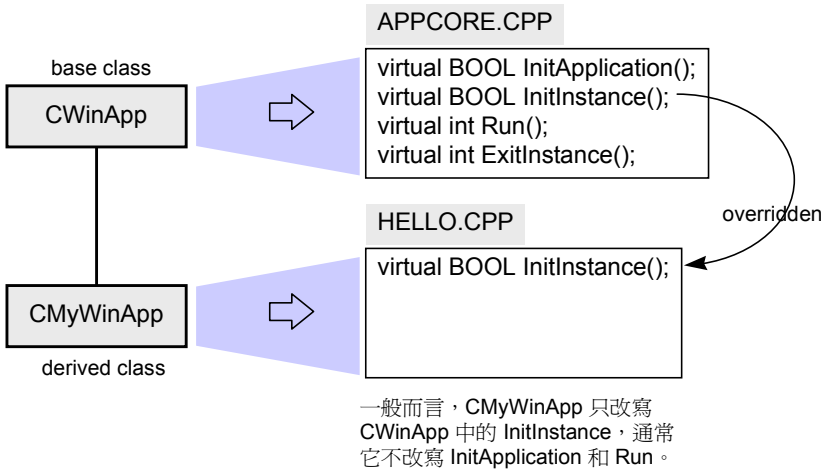
繼 *InitApplication* 之後，*AfxWinMain* 呼叫 *pApp->InitInstance*。稍早我說過了，*pApp* 指向 *CMyWinApp* 物件（也就是本例的 *theApp*），所以，當程式呼叫：

```
pApp->InitInstance();
```

相當於呼叫

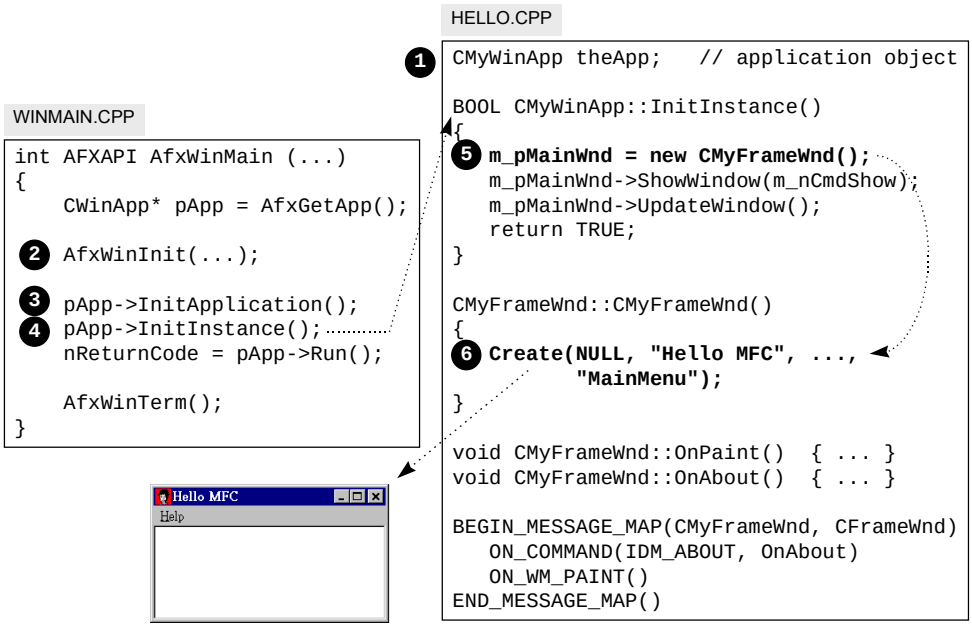
```
CMyWinApp::InitInstance();
```

但是你要知道，*CMyWinApp* 繼承自 *CWinApp*，而 *InitInstance* 又是 *CWinApp* 的一個虛擬函式。由於我們改寫了它，所以上述動作的的確確就是呼叫我們自己（*CMyWinApp*）的這個 *InitInstance* 函式。我們將在該處展開我們的主視窗生命。



注意：應用程式一定要改寫虛擬函式 *InitInstance*，因爲它在 *CWinApp* 中只是個空函式，沒有任何內建（預設）動作。

CFrameWnd::Create 產生主視窗（並先註冊視窗類別）



CMyWinApp::*InitInstance* 一開始 *new* 了一個 *CMyFrameWnd* 物件，準備用作主框視窗的 C++ 物件。*new* 會引發建構式：

```
CMyFrameWnd::CMyFrameWnd
{
    Create(NULL, "Hello MFC", WS_OVERLAPPEDWINDOW, rectDefault, NULL,
           "MainMenu");
}
```

其中 *Create* 是 *CFrameWnd* 的成員函式，它將產生一個視窗。但，使用哪一個視窗類別呢？

> 這裡所謂的「視窗類別」是由 *RegisterClass* 所註冊的一份資料結構，不是 C++ 類別。

根據 *CFrameWnd*::*Create* 的規格：

```
BOOL Create( LPCTSTR lpszClassName,
             LPCTSTR lpszWindowName,
             DWORD dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL,
             LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0,
             CCreateContext* pContext = NULL );
```

八個參數中的後六個參數都有預設值，只有前兩個參數必須指定。**第一個參數** *lpszClassName* 指定 *WNDCLASS* 視窗類別，我們放置 *NULL* 究竟代表什麼意思？意思是要以 MFC 內建的視窗類別產生一個標準的外框視窗。但，此時此刻 Hello 程式中根本不存在任何視窗類別呀！噢，*Create* 函式在產生視窗之前會引發視窗類別的註冊動作，稍後再解釋。

第二個參數 *lpszWindowName* 指定視窗標題，本例指定 "Hello MFC"。**第三個參數** *dwStyle* 指定視窗風格，預設是 *WS_OVERLAPPEDWINDOW*，也正是最常用的一種，它被定義爲（在 WINDOWS.H 之中）：

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION |
                             WS_SYSMENU | WS_THICKFRAME |
                             WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
```

因此如果你不想要視窗右上角的極大極小鈕，就得這麼做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME
       rectDefault,
       NULL,
       "MainMenu");
```

如果你希望視窗有垂直捲軸，就得在第三個參數上再加增 *WS_VSCROLL* 風格。

除了上述標準的視窗風格，另有所謂的擴充風格，可以在 *Create* 的**第七個參數** *dwExStyle* 指定之。擴充風格唯有以 ::*CreateWindowEx*（而非 ::*CreateWindow*）函式才能完成。事實上稍後你就會發現，*CFrameWnd*::*Create* 最終呼叫的正是 ::*CreateWindowEx*。

Windows 3.1 提供五種視窗擴充風格：

```
WS_EX_DLGMODALFRAME
WS_EX_NOPARENTNOTIFY
WS_EX_TOPMOST
WS_EX_ACCEPTFILES
WS_EX_TRANSPARENT
```

Windows 95 有更多選擇，包括 *WS_EX_WINDOWEDGE* 和 *WS_EX_CLIENTEDGE*，讓視窗更具 3D 立體感。Framework 已經自動爲我們指定了這兩個擴充風格。

Create 的**第四個參數** *rect* 指定視窗的位置與大小。預設值 *rectDefault* 是 *CFrameWnd* 的一個 static 成員變數，告訴 Windows 以預設方式指定視窗位置與大小，就好像在 SDK 程式中以 *CW_USEDEFAULT* 指定給 *CreateWindow* 函式一樣。如果你很有主見，希望視窗在特定位置有特定大小，可以這麼做：

```
Create(NULL,
       "Hello MFC",
       WS_OVERLAPPEDWINDOW,
       CRect(40, 60, 240, 460),  // 起始位置 (40,60)，寬 200，高 400）
       NULL,
       "MainMenu");
```

第五個參數 *pParentWnd* 指定父視窗。對於一個 top-level 視窗而言，此值應爲 *NULL*，表示沒有父視窗（其實是有的，父視窗就是 desktop 視窗）。

第六個參數 *lpszMenuName* 指定選單。本例使用一份在 RC 中準備好的選單 ¡§MainMenu¡¨。**第八個參數** *pContext* 是一個指向 *CCreateContext* 結構的指標，framework 利用它，在具備 Document/View 架構的程式中初始化外框視窗（第 8 章的「CDocTemplate 管理 CDocument / CView / CFrameWnd」一節中將談到此一主題）。本例不具備 Document/View 架構，所以不必指定 *pContext* 參數，預設值爲 *NULL*。

前面提過，*CFrameWnd*::*Create* 在產生視窗之前，會先引發視窗類別的註冊動作。讓我再扮一次 MFC 嚮導，帶你尋幽訪勝。你會看到 MFC 爲我們註冊的視窗類別名稱，及註冊動作。

◆ WINFRM.CPP

```
BOOL CFrameWnd::Create(LPCTSTR lpszClassName,
                       LPCTSTR lpszWindowName,
                       DWORD dwStyle,
                       const RECT& rect,
                       CWnd* pParentWnd,
                       LPCTSTR lpszMenuName,
                       DWORD dwExStyle,
                       CCreateContext* pContext)
{
   HMENU hMenu = NULL;
   if (lpszMenuName != NULL)
   {
      // load in a menu that will get destroyed when window gets destroyed
      HINSTANCE hInst = AfxFindResourceHandle(lpszMenuName, RT_MENU);
      hMenu = ::LoadMenu(hInst, lpszMenuName);
   }

   m_strTitle = lpszWindowName;    // save title for later

   CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle,
      rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
      pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext);

   return TRUE;
}
```

函式中呼叫 *CreateEx*。注意，*CWnd* 有成員函式 *CreateEx*，但其衍生類別 *CFrameWnd* 並無，所以這裡雖然呼叫的是 *CFrameWnd*::*CreateEx*，其實乃是從父類別繼承下來的 *CWnd*::*CreateEx*。

◆ WINCORE.CPP

```
BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
                    LPCTSTR lpszWindowName, DWORD dwStyle,
                    int x, int y, int nWidth, int nHeight,
                    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
   // allow modification of several common create parameters
   CREATESTRUCT cs;
   cs.dwExStyle = dwExStyle;
   cs.lpszClass = lpszClassName;
```

```
    cs.lpszName = lpszWindowName;
    cs.style = dwStyle;
    cs.x = x;
    cs.y = y;
    cs.cx = nWidth;
    cs.cy = nHeight;
    cs.hwndParent = hWndParent;
    cs.hMenu = nIDorHMenu;
    cs.hInstance = AfxGetInstanceHandle();
    cs.lpCreateParams = lpParam;

    PreCreateWindow(cs);
    AfxHookWindowCreate(this);  //此動作將在第９章探討。
    HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
                  cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
                  cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);
    ...
}
```

函式中呼叫的 *PreCreateWindow* 是虛擬函式，*CWnd* 和 *CFrameWnd* 之中都有定義。由於 *this* 指標所指物件的緣故，這裡應該呼叫的是 *CFrameWnd*::*PreCreateWindow*（還記得第２章我說過虛擬函式常見的那種行爲模式嗎？）

◆ WINFRM.CPP

```
// CFrameWnd second phase creation
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        cs.lpszClass = _afxWndFrameOrView;  // COLOR_WINDOW background
    }
    ...
}
```

其中 *AfxDeferRegisterClass* 是一個定義於 AFXIMPL.H 中的巨集。

◆ AFXIMPL.H

```
#define AfxDeferRegisterClass(fClass) \
  ((afxRegisteredClasses & fClass) ? TRUE : AfxEndDeferRegisterClass(fClass))
```

這個巨集表示，如果變數 *afxRegisteredClasses* 的值顯示系統已經註冊了 *fClass* 這種視窗類別，MFC 就啥也不做；否則就呼叫 *AfxEndDeferRegisterClass*(fClass)，準備註冊之。*afxRegisteredClasses* 定義於 AFXWIN.H，是一個旗標變數，用來記錄已經註冊了哪些視窗類別：

```
// in AFXWIN.H
#define afxRegisteredClasses  AfxGetModuleState()->m_fRegisteredClasses
```

◆ WINCORE.CPP：

```
#0001  BOOL AFXAPI AfxEndDeferRegisterClass(short fClass)
#0002  {
#0003      BOOL bResult = FALSE;
#0004
#0005      // common initialization
#0006      WNDCLASS wndcls;
#0007      memset(&wndcls, 0, sizeof(WNDCLASS));   // start with NULL defaults
#0008      wndcls.lpfnWndProc = DefWindowProc;
#0009      wndcls.hInstance = AfxGetInstanceHandle();
#0010      wndcls.hCursor = afxData.hcurArrow;
#0011
#0012      AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
#0013      if (fClass & AFX_WND_REG)
#0014      {
#0015          // Child windows - no brush, no icon, safest default class styles
#0016          wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0017          wndcls.lpszClassName = _afxWnd;
#0018          bResult =  AfxRegisterClass(&wndcls);
#0019          if (bResult)
#0020              pModuleState->m_fRegisteredClasses |= AFX_WND_REG;
#0021      }
#0022      else if (fClass & AFX_WNDOLECONTROL_REG)
#0023      {
#0024          // OLE Control windows - use parent DC for speed
#0025          wndcls.style |= CS_PARENTDC | CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0026          wndcls.lpszClassName = _afxWndOleControl;
#0027          bResult =  AfxRegisterClass(&wndcls);
#0028          if (bResult)
```

```
#0029            pModuleState->m_fRegisteredClasses |= AFX_WNDOLECONTROL_REG;
#0030        }
#0031        else if (fClass & AFX_WNDCONTROLBAR_REG)
#0032        {
#0033            // Control bar windows
#0034            wndcls.style = 0;   // control bars don't handle double click
#0035            wndcls.lpszClassName = _afxWndControlBar;
#0036            wndcls.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
#0037            bResult =  AfxRegisterClass(&wndcls);
#0038            if (bResult)
#0039                pModuleState->m_fRegisteredClasses |= AFX_WNDCONTROLBAR_REG;
#0040        }
#0041        else if (fClass & AFX_WNDMDIFRAME_REG)
#0042        {
#0043            // MDI Frame window (also used for splitter window)
#0044            wndcls.style = CS_DBLCLKS;
#0045            wndcls.hbrBackground = NULL;
#0046            bResult = RegisterWithIcon(&wndcls, _afxWndMDIFrame,
                                            AFX_IDI_STD_MDIFRAME);
#0047            if (bResult)
#0048                pModuleState->m_fRegisteredClasses |= AFX_WNDMDIFRAME_REG;
#0049        }
#0050        else if (fClass & AFX_WNDFRAMEORVIEW_REG)
#0051        {
#0052            // SDI Frame or MDI Child windows or views - normal colors
#0053            wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
#0054            wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
#0055            bResult = RegisterWithIcon(&wndcls, _afxWndFrameOrView,
                                            AFX_IDI_STD_FRAME);
#0056            if (bResult)
#0057                pModuleState->m_fRegisteredClasses |= AFX_WNDFRAMEORVIEW_REG;
#0058        }
#0059        else if (fClass & AFX_WNDCOMMCTLS_REG)
#0060        {
#0061            InitCommonControls();
#0062            bResult = TRUE;
#0063            pModuleState->m_fRegisteredClasses |= AFX_WNDCOMMCTLS_REG;
#0064        }
#0065
#0066        return bResult;
#0067    }
```

出現在上述函式中的六個視窗類別標籤代碼，分別定義於 AFXIMPL.H 中：

```
#define AFX_WND_REG             (0x0001)
#define AFX_WNDCONTROLBAR_REG   (0x0002)
#define AFX_WNDMDIFRAME_REG     (0x0004)
#define AFX_WNDFRAMEORVIEW_REG  (0x0008)
#define AFX_WNDCOMMCTLS_REG     (0x0010)
#define AFX_WNDOLECONTROL_REG   (0x0020)
```

出現在上述函式中的五個視窗類別名稱，分別定義於 WINCORE.CPP 中：

```
const TCHAR _afxWnd[] = AFX_WND;
const TCHAR _afxWndControlBar[] = AFX_WNDCONTROLBAR;
const TCHAR _afxWndMDIFrame[] = AFX_WNDMDIFRAME;
const TCHAR _afxWndFrameOrView[] = AFX_WNDFRAMEORVIEW;
const TCHAR _afxWndOleControl[] = AFX_WNDOLECONTROL;
```

而等號右手邊的那些 AFX_ 常數又定義於 AFXIMPL.H 中：

```
#ifndef _UNICODE
#define _UNICODE_SUFFIX
#else
#define _UNICODE_SUFFIX _T("u")
#endif

#ifndef _DEBUG
#define _DEBUG_SUFFIX
#else
#define _DEBUG_SUFFIX _T("d")
#endif

#ifdef _AFXDLL
#define _STATIC_SUFFIX
#else
#define _STATIC_SUFFIX _T("s")
#endif

#define AFX_WNDCLASS(s) \
  _T("Afx") _T(s) _T("42") _STATIC_SUFFIX _UNICODE_SUFFIX _DEBUG_SUFFIX

#define AFX_WND                 AFX_WNDCLASS("Wnd")
#define AFX_WNDCONTROLBAR       AFX_WNDCLASS("ControlBar")
#define AFX_WNDMDIFRAME         AFX_WNDCLASS("MDIFrame")
#define AFX_WNDFRAMEORVIEW      AFX_WNDCLASS("FrameOrView")
#define AFX_WNDOLECONTROL       AFX_WNDCLASS("OleControl")
```

所以，如果在 Windows 95（non-Unicode）中使用 MFC 動態聯結版和除錯版，五個視窗類別的名稱將是：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

如果在 Windows NT（Unicode 環境）中使用 MFC 靜態聯結版和除錯版，五個視窗類別的名稱將是：

```
"AfxWnd42sud"
"AfxControlBar42sud"
"AfxMDIFrame42sud"
"AfxFrameOrView42sud"
"AfxOleControl42sud"
```

這五個視窗類別的使用時機爲何？稍後再來一探究竟。

讓我們再回顧 *AfxEndDeferRegisterClass* 的動作。它呼叫兩個函式完成實際的視窗類別註冊動作，一個是 *RegisterWithIcon*，一個是 *AfxRegisterClass*：

```
static BOOL AFXAPI RegisterWithIcon(WNDCLASS* pWndCls,
       LPCTSTR lpszClassName, UINT nIDIcon)
{
   pWndCls->lpszClassName = lpszClassName;
   HINSTANCE hInst = AfxFindResourceHandle(
          MAKEINTRESOURCE(nIDIcon), RT_GROUP_ICON);
   if ((pWndCls->hIcon = ::LoadIcon(hInst, MAKEINTRESOURCE(nIDIcon))) == NULL)
   {
      // use default icon
      pWndCls->hIcon = ::LoadIcon(NULL, IDI_APPLICATION);
   }
   return AfxRegisterClass(pWndCls);
}
```

```
BOOL AFXAPI AfxRegisterClass(WNDCLASS* lpWndClass)
{
   WNDCLASS wndcls;
```

```
    if (GetClassInfo(lpWndClass->hInstance,
                    lpWndClass->lpszClassName, &wndcls))
    {
        // class already registered
        return TRUE;
    }

    ::RegisterClass(lpWndClass);
    ...
    return TRUE;
}
```

注意，不同類別的 *PreCreateWindow* 成員函式都是在視窗產生之前一刻被呼叫，準備用來註冊視窗類別。如果我們指定的視窗類別是 *NULL*，那麼就使用系統預設類別。從 *CWnd* 及其各個衍生類別的 *PreCreateWindow* 成員函式可以看出，整個 Framework 針對不同功能的視窗使用了哪些視窗類別：

```
// in WINCORE.CPP
BOOL CWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WND_REG);
        ...
        cs.lpszClass = _afxWnd;   （這表示 CWnd 使用的視窗類別是 _afxWnd）
    }
    return TRUE;
}

// in WINFRM.CPP
BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView; （這表示 CFrameWnd 使用的視窗
    }                                        類別是 _afxWndFrameOrView）
    ...
}

// in WINMDI.CPP
BOOL CMDIFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDMDIFRAME_REG);
        ...
        cs.lpszClass = _afxWndMDIFrame;  （這表示 CMDIFrameWnd 使用的視窗
    }                                      類別是 _afxWndMDIFrame）
    return TRUE;
}

// in WINMDI.CPP
BOOL CMDIChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
    ...
    return CFrameWnd::PreCreateWindow(cs);   （這表示 CMDIChildWnd 使用的視窗
}                                             類別是 _afxWndFrameOrView）

// in VIEWCORE.CPP
BOOL CView::PreCreateWindow(CREATESTRUCT & cs)
{
    if (cs.lpszClass == NULL)
    {
        AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG);
        ...
        cs.lpszClass = _afxWndFrameOrView; （這表示 CView 使用的視窗
    }                                       類別是 _afxWndFrameOrView）
    ...
}
```

題外話：「*Create* 是一個比較粗糙的函式，不提供我們對圖示（icon）或滑鼠游標的設定，所以在 *Create* 函式中我們看不到相關參數」。這樣的說法對嗎？雖然「不能夠讓我們指定視窗圖示以及滑鼠游標」是事實，但這本來就與 *Create* 無關。回憶 SDK 程式，指定圖示和游標形狀實爲 *RegisterClass* 的責任而非 *CreateWindow* 的責任！

MFC 程式的 *RegisterClass* 動作並非由程式員自己來做，因此似乎難以改變圖示。不過，MFC 還是開放了一個窗口，我們可以在 HELLO.RC 這麼設定圖示：

```
AFX_IDI_STD_FRAME  ICON  DISCARDABLE  "HELLO.ICO"
```

你可以從 *AfxEndDeferRegisterClass* 的第 55 行看出，當它呼叫 *RegisterWithIcon* 時，指定的 icon 正是 AFX_IDI_STD_FRAME。

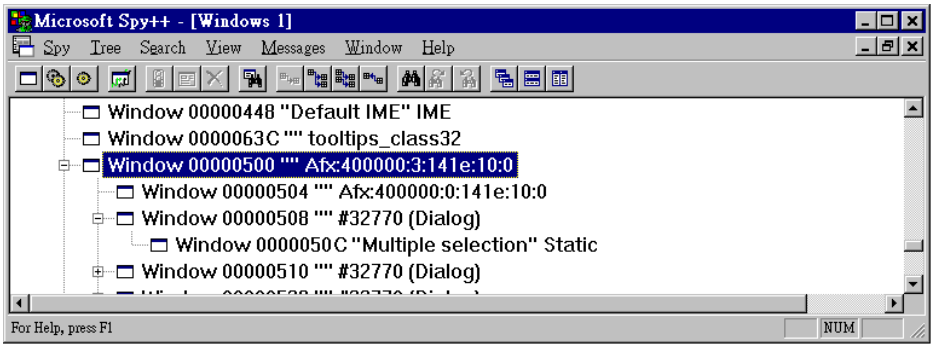
滑鼠游標的設定就比較麻煩了。要改變游標形狀，我們必須呼叫 *AfxRegisterWndClass*（其中有 ¡Cursor¡¨參數）註冊自己的視窗類別；然後再將其傳回值（一個字串）做爲 *Create* 的第一個參數。

奇怪的視窗類別名稱 Afx:b:14ae:6:3e8f

當應用程式呼叫 *CFrameWnd::Create*（或 *CMDIFrameWnd*::*LoadFrame*，第 7 章）準備產生視窗時，MFC 才會在 *Create* 或 *LoadFrame* 內部所呼叫的 *PreCreateWindow* 虛擬函式中爲你產生適當的視窗類別。你已經在上一節看到了，這些視窗類別的名稱分別是（假設在 Win95 中使用 MFC 4.2 動態聯結版和除錯版）：

```
"AfxWnd42d"
"AfxControlBar42d"
"AfxMDIFrame42d"
"AfxFrameOrView42d"
"AfxOleControl42d"
```

然而，當我們以 Spy++（VC++ 所附的一個工具）觀察視窗類別的名稱，卻發現：



視窗類別名稱怎麼會變成像 Afx:b:14ae:6:3e8f 這副奇怪模樣呢？原來是 Application Framework 玩了一些把戲，它把這些視窗類別名稱轉換爲 Afx:x:y:z:w 的型式，成爲獨一無二的視窗類別名稱：

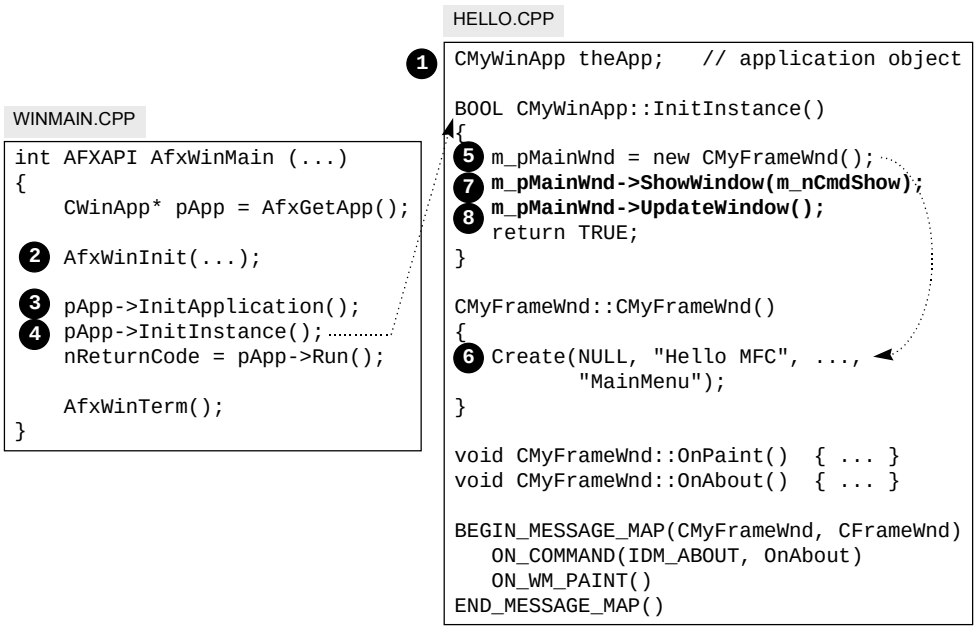
x:　視窗風格（window style）的 hex 值
y:　視窗滑鼠游標的 hex 值
z:　視窗背景顏色的 hex 值
w:　視窗圖示（icon）的 hex 值

如果你要使用原來的（MFC 預設的）那些個視窗類別，但又希望擁有自己定義的一個有意義的類別名稱，你可以改寫 *PreCreateWindow* 虛擬函式（因爲 *Create* 和 *LoadFrame* 的內部都會呼叫它），在其中先利用 API 函式 *GetClassInfo* 獲得該類別的一個副本，更改其類別結構中的 *lpszClassName* 欄位（甚至更改其 *hIcon* 欄位），再以 *AfxRegisterClass* 重新註冊之，例如：

```
#0000  #define MY_CLASSNAME  "MyClassName"
#0001
#0002  BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
#0003  {
#0004      static LPCSTR className = NULL;
#0005
#0006      if (!CFrameWnd::PreCreateWindow(cs))
#0007         return FALSE;
#0008
#0009      if (className==NULL) {
#0010          // One-time class registration
#0011          // The only purpose is to make the class name something
#0012          // meaningful instead of "Afx:0x4d:27:32:hup1hup:hike!"
#0013          //
#0014          WNDCLASS wndcls;
#0015          ::GetClassInfo(AfxGetInstanceHandle(), cs.lpszClass, &wndcls);
#0016          wndcls.lpszClassName = MY_CLASSNAME;
#0017          wndcls.hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
#0018          VERIFY(AfxRegisterClass(&wndcls));
#0019          className=TRACEWND_CLASSNAME;
#0020      }
#0021      cs.lpszClass = className;
#0022
#0023      return TRUE;
#0024  }
```

本書附錄D「以 MFC 重建 Debug Window（DBWIN）」會運用到這個技巧。

視窗顯示與更新



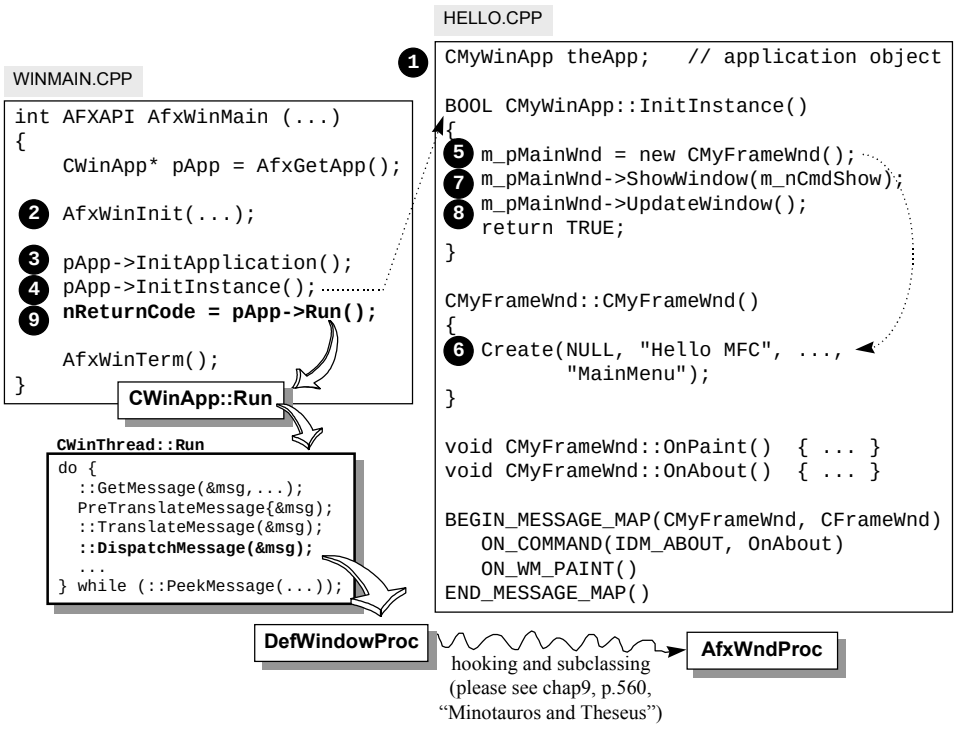
CMyFrameWnd::*CMyFrameWnd* 結束後，視窗已經誕生出來；程式流程又回到 *CMyWinApp*::*InitInstance*，於是呼叫 *ShowWindow* 函式令視窗顯示出來，並呼叫 *UpdateWindow* 函式令 Hello 程式送出 *WM_PAINT* 訊息。

我們很關心這個 *WM_PAINT* 訊息如何送到視窗函式的手中。而且，視窗函式又在哪裡？

MFC 程式是不是也像 SDK 程式一樣，有一個 *GetMessage*/*DispatchMesage* 迴路？是否每個視窗也都有一個視窗函式，並以某種方式進行訊息的判斷與處理？

兩者都是肯定的。我們馬上來尋找證據。

CWinApp::Run - 程式生命的活水源頭



Hello 程式進行到這裡，視窗類別註冊好了，視窗誕生並顯示出來了，*UpdateWindow* 被呼叫，使得訊息佇列中出現了一個 *WM_PAINT* 訊息，等待被處理。現在，執行的腳步到達 *pApp->Run*。

稍早我說過了，*pApp* 指向 *CMyWinApp* 物件（也就是本例的 *theApp*），所以，當程式呼叫：

```
pApp->Run();
```

相當於呼叫：

```
CMyWinApp::Run();
```

要知道，*CMyWinApp* 繼承自 *CWinApp*，而 *Run* 又是 *CWinApp* 的一個虛擬函式。我們並沒有改寫它（大部份情況下不需改寫它），所以上述動作相當於呼叫：

```
CWinApp::Run();
```

其原始碼出現在 APPCORE.CPP 中：

```
int CWinApp::Run()
{
    if (m_pMainWnd == NULL && AfxOleGetUserCtrl())
    {
        // Not launched /Embedding or /Automation, but has no main window!
        TRACE0("Warning: m_pMainWnd is NULL in CWinApp::Run - quitting
                application.\n");
        AfxPostQuitMessage(0);
    }
    return CWinThread::Run();
}
```

32 位元 MFC 與 16 位元 MFC 的巨大差異在於 *CWinApp* 與 *CCmdTarget* 之間多出了一個 *CWinThread*，事情變得稍微複雜一些。*CWinThread* 定義於 THRDCORE.CPP：

```
int CWinThread::Run()
{
    // for tracking the idle time state
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;

    // acquire and dispatch messages until a WM_QUIT message is received.
    for (;;)
    {
        // phase1: check to see if we can do idle work
        while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }

        // phase2: pump messages while available
        do
        {
            // pump message, but quit on WM_QUIT
```

```
            if (!PumpMessage())
                return ExitInstance();

            // reset "no idle" state after pumping "normal" message
            if (IsIdleMessage(&m_msgCur))
            {
                bIdle = TRUE;
                lIdleCount = 0;
            }

        } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }

    ASSERT(FALSE);  // not reachable
}

BOOL CWinThread::PumpMessage()
{
    if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
    {
        return FALSE;
    }

    // process this message
    if (m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
    {
        ::TranslateMessage(&m_msgCur);
        ::DispatchMessage(&m_msgCur);
    }
    return TRUE;
}
```

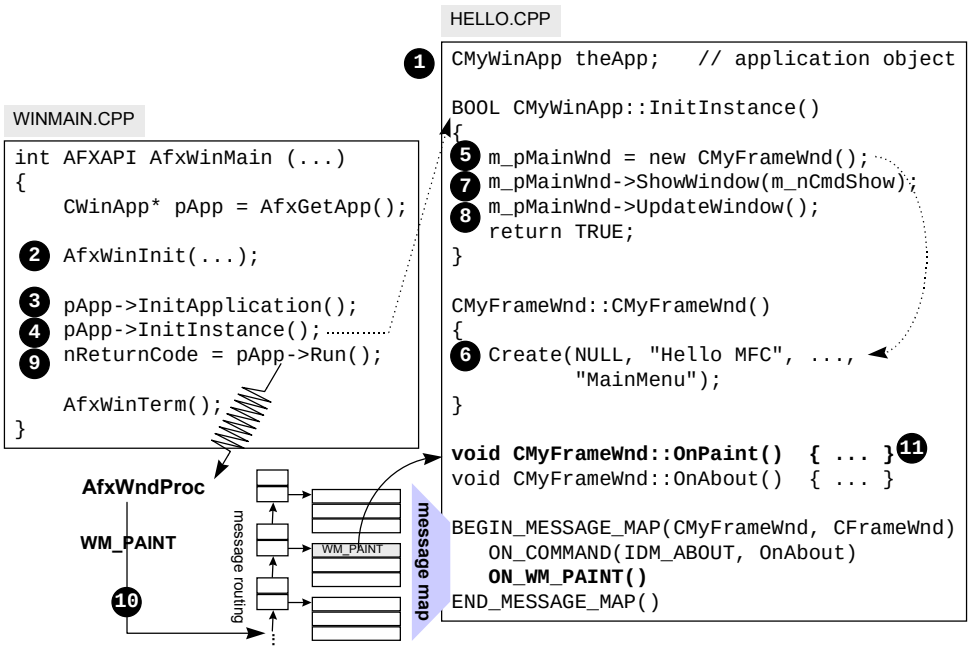
獲得的訊息如何交給適當的常式去處理呢？SDK 程式的作法是呼叫 *DispatchMessage*，把訊息丟給視窗函式；MFC 也是如此。但我們並未在 Hello 程式中提供任何視窗函式，是的，視窗函式事實上由 MFC 提供。回頭看看前面 *AfxEndDeferRegisterClass* 原始碼，它在註冊四種視窗類別之前已經指定視窗函式爲：

```
wndcls.lpfnWndProc = DefWindowProc;
```

注意，雖然視窗函式被指定爲 *DefWindowProc* 成員函式，但事實上訊息並不是被唧往該處，而是一個名爲 *AfxWndProc* 的全域函式去。這其中牽扯到 MFC 暗中做了大挪移的手腳（利用 hook 和 subclassing），我將在第 9 章詳細討論這個「乾坤大挪移」。

你看，*WinMain* 已由 MFC 提供，視窗類別已由 MFC 註冊完成、連視窗函式也都由 MFC 提供。那麼我們（程式員）如何爲特定的訊息設計特定的處理常式？MFC 應用程式對訊息的辨識與判別是採用所謂的「Message Map 機制」。

把訊息與處理函式連在一起：Message Map 機制



基本上 Message Map 機制是爲了提供更方便的程式介面（例如巨集或表格），讓程式員很方便就可以建立起訊息與處理常式的對應關係。這並不是什麼新發明，我在第 1 章示範了一種風格簡明的 SDK 程式寫法，就已經展現出這種精神。

MFC 提供給應用程式使用的「很方便的介面」是兩組巨集。以 Hello 的主視窗爲例，第一個動作是在 HELLO.H 的 *CMyFrameWnd* 加上 *DECLARE_MESSAGE_MAP*：

```
class CMyFrameWnd : public CFrameWnd
{
public:
	CMyFrameWnd();
	afx_msg void OnPaint();
	afx_msg void OnAbout();
	DECLARE_MESSAGE_MAP()
};
```

第二個動作是在 HELLO.CPP 的任何位置（當然不能在函式之內）使用巨集如下：

```
BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

這麼一來就把訊息 *WM_PAINT* 導到 *OnPaint* 函式，把 *WM_COMMAND*（*IDM_ABOUT*）導到 OnAbout 函式去了。但是，單憑一個 *ON_WM_PAINT* 巨集，沒有任何參數，如何使 *WM_PAINT* 流到 *OnPaint* 函式呢？

MFC 把訊息主要分爲三大類，Message Map 機制中對於訊息與函式間的對映關係也明定以下三種：

■ 標準 Windows 訊息（WM_xxx）的對映規則：

巨集名稱	對映訊息	訊息處理函式（名稱已由系統預設）
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
...		

■ 命令訊息（*WM_COMMAND*）的一般性對映規則是：

```
ON_COMMAND(<id>,<memberFxn>)
```

例如：

```
ON_COMMAND(IDM_ABOUT,    OnAbout)
ON_COMMAND(IDM_FILENEW,  OnFileNew)
ON_COMMAND(IDM_FILEOPEN, OnFileOpen)
ON_COMMAND(IDM_FILESAVE, OnFileSave)
```

■ 「Notification 訊息」（由控制元件產生，例如 *BN_xxx*）的對映機制的巨集分爲好幾種（因爲控制元件本就分爲好幾種），以下各舉一例做代表：

控制元件	巨集名稱	訊息處理函式
Button	ON_BN_CLICKED(<id>,<memberFxn>)	memberFxn
ComboBox	ON_CBN_DBLCLK(<id>,<memberFxn>)	memberFxn
Edit	ON_EN_SETFOCUS(<id>,<memberFxn>)	memberFxn
ListBox	ON_LBN_DBLCLK(<id>,<memberFxn>)	memberFxn

各個訊息處理函式均應以 afx_msg void 爲函式型式。

爲什麼經過這樣的巨集之後，訊息就會自動流往指定的函式去呢？謎底在於 Message Map 的結構設計。如果你把第 3 章的 Message Map 模擬程式好好研究過，現在應該已是成竹在胸。我將在第 9 章再討論 MFC 的 Message Map。

好奇心擺兩旁，還是先把實用上的問題放中間吧。如果某個訊息在 Message Map 中找不到對映記錄，訊息何去何從？答案是它會往基礎類別流竄，這個訊息流竄動作稱爲「Message Routing」。如果一直竄到最基礎的類別仍找不到對映的處理常式，自會有預設函式來處理，就像 SDK 中的 *DefWindowProc* 一樣。

MFC 的 *CCmdTarget* 所衍生下來的每一個類別都可以設定自己的 Message Map，因爲它們都可能（可以）收到訊息。

訊息流動是個頗爲複雜的機制，它和 Document/View、動態生成（Dynamic Creation），檔案讀寫（Serialization）一樣，都是需要特別留心的地方。

來龍去脈總整理

前面各節的目的就是如何將表面上看來不知所以然的 MFC 程式對映到我們在 SDK 程式設計中學習到的訊息流動觀念，從而清楚地掌握 MFC 程式的誕生與死亡。讓我對 MFC 程式的來龍去脈再做一次總整理。

程式的誕生：

- Application object 產生，記憶體於是獲得配置，初值亦設立了。
- *AfxWinMain* 執行 *AfxWinInit*，後者又呼叫 *AfxInitThread*，把訊息佇列儘量加大到 96。
- *AfxWinMain* 執行 *InitApplication*。這是 *CWinApp* 的虛擬函式，但我們通常不改寫它。
- *AfxWinMain* 執行 *InitInstance*。這是 *CWinApp* 的虛擬函式，我們必須改寫它。
- *CMyWinApp*::*InitInstance 'new'* 了一個 *CMyFrameWnd* 物件。
- *CMyFrameWnd* 建構式呼叫 *Create*，產生主視窗。我們在 *Create* 參數中指定的視窗類別是 *NULL*，於是 MFC 根據視窗種類，自行爲我們註冊一個名爲 "AfxFrameOrView42d" 的視窗類別。
- 回到 *InitInstance* 中繼續執行 *ShowWindow*，顯示視窗。
- 執行 *UpdateWindow*，於是發出 *WM_PAINT*。
- 回到 *AfxWinMain*，執行 *Run*，進入訊息迴圈。

程式開始運作：

- 程式獲得 *WM_PAINT* 訊息（藉由 *CWinApp*::*Run* 中的 ::*GetMessage* 迴路）。
- *WM_PAINT* 經由::*DispatchMessage* 送到視窗函式 *CWnd*::*DefWindowProc* 中。

- *CWnd*::*DefWindowProc* 將訊息繞行過訊息映射表格（Message Map）。
- 繞行過程中發現有吻合項目，於是呼叫項目中對應的函式。此函式是應用程式利用 *BEGIN_MESSAGE_MAP* 和 *END_MESSAGE_MAP* 之間的巨集設立起來的。
- 標準訊息的處理常式亦有標準命名，例如 *WM_PAINT* 必然由 *OnPaint* 處理。

以下是程式的死亡：

- 使用者選按【File/Close】，於是發出 *WM_CLOSE*。
- *CMyFrameWnd* 並沒有設置 *WM_CLOSE* 處理常式，於是交給預設之處理常式。
- 預設函式對於 *WM_CLOSE* 的處理方式是呼叫 ::*DestroyWindow*，並因而發出 *WM_DESTROY*。
- 預設之 *WM_DESTROY* 處理方式是呼叫::*PostQuitMessage*，因此發出 *WM_QUIT*。
- *CWinApp*::*Run* 收到 *WM_QUIT* 後會結束其內部之訊息迴路，然後呼叫 *ExitInstance*，這是 *CWinApp* 的一個虛擬函式。
- 如果 *CMyWinApp* 改寫了 *ExitInstance*，那麼 *CWinApp*::*Run* 所呼叫的就是 *CMyWinApp*::*ExitInstance*，否則就是 *CWinApp*::*ExitInstance*。
- 最後回到 *AfxWinMain*，執行 *AfxWinTerm*，結束程式。

Callback 函式

Hello 的 *OnPaint* 在程式收到 *WM_PAINT* 之後開始運作。爲了讓 "Hello, MFC" 字樣從天而降並有動畫效果，程式採用 *LineDDA* API 函式。我的目的一方面是爲了示範訊息的處理，一方面也爲了示範 MFC 程式如何呼叫 Windows API 函式。許多人可能不熟悉 *LineDDA*，所以我也一併介紹這個有趣的函式。

首先介紹 *LineDDA*：

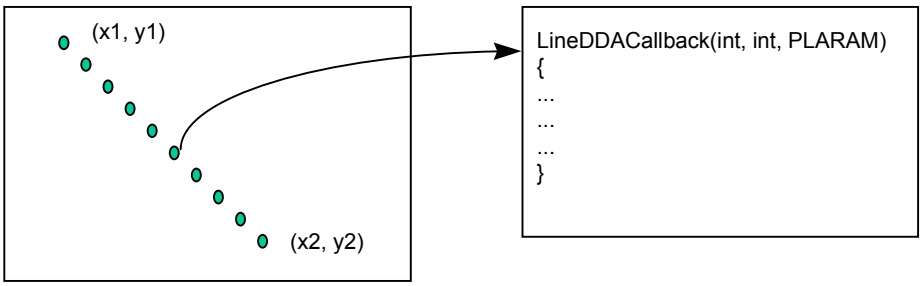
```
void WINAPI LineDDA(int, int, int, int, LINEDDAPROC, LPARAM);
```

這個函式用來做動畫十分方便，你可以利用前四個參數指定螢幕上任意兩點的 (x,y) 座標，此函式將以 Bresenham 演算法（註) 計算出通過兩點之直線中的每一個螢幕圖素座標；每計算出一個座標，就通知由 *LineDDA* 第五個參數所指定的 callback 函式。這個 callback 函式的型式必須是：

```
typedef void (CALLBACK* LINEDDAPROC)(int, int, LPARAM);
```

通常我們在這個 callback 函式中設計繪圖動作。玩過 Windows 的接龍遊戲嗎？接龍成功後撲克牌的跳動效果就可以利用 *LineDDA* 完成。雖然撲克牌的跳動路徑是一條曲線，但將曲線拆成數條直線並不困難。*LineDDA* 的第六個（最後一個）參數可以視應用程式的需要傳遞一個 32 位元指標，本例中 Hello 傳的是一個 Device Context。

> Bresenham 演算法是電腦圖學中爲了「顯示器（螢幕或印表機）係由圖素構成」的這個特性而設計出來的演算法，使得求直線各點的過程中全部以整數來運算，因而大幅提升計算速度。



你可以指定兩個座標點，LineDDA 將以 Bresenham 演算法計算出通過兩點之直線中每一個螢幕圖素的座標。每計算出一個座標，就以該座標爲參數，呼叫你所指定的 callback 函式。

圖 6-6　LineDDA 函式說明

LineDDA 並不屬於任何一個 MFC 類別，因此呼叫它必須使用 C++ 的 "scope operator"（也就是 ::）：

```
void CMyFrameWnd::OnPaint()
{
CPaintDC dc(this);
CRect rect;

   GetClientRect(rect);

   dc.SetTextAlign(TA_BOTTOM | TA_CENTER);

   ::LineDDA(rect.right/2, 0, rect.right/2, rect.bottom/2,
      (LINEDDAPROC) LineDDACallback, (LPARAM) (LPVOID) &dc);
}
```

其中 *LineDDACallback* 是我們準備的 callback 函式，必須在類別中先有宣告：

```
class CMyFrameWnd : public CFrameWnd
{
...
private:
   static VOID CALLBACK LineDDACallback(int,int,LPARAM);
};
```

請注意，如果類別的成員函式是一個 callback 函式， 你必須宣告它爲 "static"，才能把 C++ 編譯器加諸於函式的一個隱藏參數 *this* 去掉（請看方塊註解)。

以類別的成員函式作爲 Windows callback 函式

雖然現在來講這個題目，對初學者而言恐怕是過於艱深，但我想對就還是個好機會 --- 我可以在介紹如何使用 callback 函式的場合，順便介紹一些 C++ 的重要觀念。

首先我要很快地解釋一下什麼是 callback 函式。它是由你設計而卻由 Windows 系統呼叫的函式，統稱爲 callback 函式。這些函式都有一定的型態，以配合 Windows 的呼叫動作。

某些 Windows API 函式會要求以 callback 函式作爲其參數之一，這些 API 例如

SetTimer、*LineDDA*、*EnumObjects*。通常這種 API 會在進行某種行為之後或滿足某種狀態之時呼叫該 callback 函式。圖 6-6 解釋過 *LineDDA* 呼叫 callback 函式的時機；下面即將示範的 *EnumObjects* 則是在發現某個 Device Context 的 GDI object 符合我們的指定型態時，呼叫 callback 函式。

好，現在我們要討論的是，什麼函式有資格在 C++ 程式中做為 callback 函式？這個問題的背後是：C++ 程式中的 callback 函式有什麼特別的嗎？為什麼要特別提出討論？

是的，特別之處在於，C++ 編譯器為類別成員函式多準備了一個隱藏參數（程式碼中看不到），這使得函式型態與 Windows callback 函式的預設型態不符。

假設我們有一個 *CMyclass* 如下：

```
class CMyclass {
  private :
    int nCount;
    int CALLBACK _export
        EnumObjectsProc(LPSTR lpLogObject, LPSTR lpData);
  public :
    void enumIt(CDC& dc);
}
void CMyclass::enumIt(CDC& dc)
{
  // 註冊 callback 函式
  dc.EnumObjects(OBJ_BRUSH, EnumObjectsProc, NULL);
}
```

C++ 編譯器針對 *CMyclass*::*enumIt* 實際做出來的碼相當於：

```
void CMyclass::enumIt(CDC& dc)
{
  // 註冊 callback 函式
  CDC::EnumObjects(OBJ_BRUSH, EnumObjectsProc,
                   NULL, (CDC *)&dc);
}
```

你所看到的最後一個參數，(CDC *)&dc，其實就是 *this* 指標。類別成員函式都有 *this*

指標才得以抓到正確物件的資料。你要知道，記憶體中只會有一份類別成員函式，但卻可能有許多份類別成員變數 --- 每個物件擁有一份。

C++ 以隱藏的 *this* 指標指出正確的物件。當你這樣做：

```
nCount = 0;
```

其實是：

```
this->nCount = 0;
```

基於相同的道理，上例中的 *EnumObjectsProc* 既然是一個成員函式，C++ 編譯器也會為它多準備一個隱藏參數。

好，問題就出在這個隱藏參數。callback 函式是給 Windows 呼叫用的，Windows 並不經由任何物件呼叫這個函式，也就無由傳遞 *this* 指標給 callback 函式，於是導致堆疊中有一個隨機變數會成為 *this* 指標，而其結果當然是程式的崩潰了。

要把某個函式用作 callback 函式，就必須告訴 C++ 編譯器，不要放 *this* 指標作為該函式的最後一個參數。兩個方法可以做到這一點：

1. 不要使用類別的成員函式（也就是說，要使用全域函式）做為 callback 函式。
2. 使用 static 成員函式。也就是在函式前面加上 static 修飾詞。

第一種作法相當於在 C 語言中使用 callback 函式。第二種作法比較接近 OO 的精神。

我想再進一步提醒你的是，C++ 中的 static 成員函式特性是，即使物件還沒有產生，static 成員也已經存在（函式或變數都如此）。換句話說物件還沒有產生之前你已經可以呼叫類別的 static 函式或使用類別的 static 變數了。請參閱第二頁。

也就是說，凡宣告為 static 的東西（不管函式或變數）都並不和物件結合在一起，它們是類別的一部份，不屬於物件。

閒置時間（idle time）的處理：OnIdle

爲了讓 Hello 程式更具體而微地表現一個 MFC 應用程式的水準，我打算爲它加上閒置時間（idle time）的處理。

我已經在第 1 章介紹過了閒置時間，也簡介了 Win32 程式如何以 *PeekMessage*「偷閒」。Microsoft 業已把這個觀念落實到 *CWinApp*（不，應該是 *CWinThread*）中。請你回頭看看本章的稍早的「*CWinApp*::*Run* - 程式生命的活水源頭」一節，那一節已經揭露了 MFC 訊息迴路的秘密：

```
int CWinThread::Run()
{
    ...
    for (;;)
    {
        while (bIdle &&
               !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }
        ... // msg loop
    }
}
```

CThread::*OnIdle* 做些什麼事情呢？*CWinApp* 改寫了 *OnIdle* 函式，*CWinApp*::*OnIdle* 又做些什麼事情呢？你可以從 THRDCORE.CPP 和 APPCORE.CPP 中找到這兩個函式的原始碼，原始碼可以說明一切。當然基本上我們可以猜測 *OnIdle* 函式中大概是做一些系統（指的是 MFC 本身）的維護工作。這一部份的功能可以說日趨式微，因爲低優先權的執行緒可以替代其角色。

如果你的 MFC 程式也想處理 idle time，只要改寫 *CWinApp* 衍生類別的 *OnIdle* 函式即可。這個函式的型態如下：

```
virtual BOOL OnIdle(LONG lCount);
```

lCount 是系統傳進來的一個值，表示自從上次有訊息進來，到現在，*OnIdle* 已經被呼叫了多少次。稍後我將改寫 Hello 程式，把這個值輸出到視窗上，你就可以知道閒置時間是多麼地頻繁。*lCount* 會持續累增，直到 *CWinThread*::*Run* 的訊息迴路又獲得了一個訊息，此值才重置爲 0。

> 注意：Jeff Prosise 在他的 ***Programming Windows 95 with MFC*** 一書第 7 章談到 *OnIdle* 函式時，曾經說過有幾個訊息並不會重置 *lCount* 爲 0，包括滑鼠訊息、*WM_SYSTIMER*、*WM_PAINT*。不過根據我實測的結果，至少滑鼠訊息是會的。稍後你可在新版的 Hello 程式移動滑鼠，看看 *lCount* 會不會重設爲 0。

我如何改寫 Hello 呢？下面是幾個步驟：

1. 在 *CMyWinApp* 中增加 *OnIdle* 函式的宣告：

```
class CMyWinApp : public CWinApp
{
public:
   virtual BOOL InitInstance();      // 每一個應用程式都應該改寫此函式
   virtual BOOL OnIdle(LONG lCount);  // OnIdle 用來處理閒置時間（idle time）
};
```

2. 在 *CMyFrameWnd* 中增加一個 *IdleTimeHandler* 函式宣告。這麼做是因爲我希望在視窗中顯示 *lCount* 值，所以最好的作法就是在 *OnIdle* 中呼叫 *CMyFrameWnd* 成員函式，這樣才容易獲得繪圖所需的 DC。

```
class CMyFrameWnd : public CFrameWnd
{
public:
   CMyFrameWnd();          // constructor
   afx_msg void OnPaint();   // for WM_PAINT
   afx_msg void OnAbout();   // for WM_COMMAND (IDM_ABOUT)
   void IdleTimeHandler(LONG lCount);  // we want it call by CMyWinApp::OnIdle
   ...
};
```

3. 在 HELLO.CPP 中定義 *CMyWinApp*::*OnIdle* 函式如下：

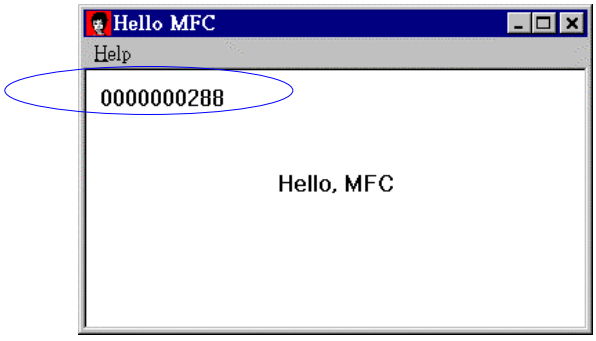
```
BOOL CMyWinApp::OnIdle(LONG lCount)
{
   CMyFrameWnd* pWnd = (CMyFrameWnd*)m_pMainWnd;
   pWnd->IdleTimeHandler(lCount);

   return TRUE;
}
```

4. 在 HELLO.CPP 中定義 *CMyFrameWnd::IdleTimeHandler* 函式如下：

```
void CMyFrameWnd::IdleTimeHandler(LONG lCount)
{
  CString str;
  CRect rect(10,10,200,30);
  CDC* pDC = new CClientDC(this);

  str.Format("%010d", lCount);
  pDC->DrawText(str, &rect, DT_LEFT | DT_TOP);
}
```

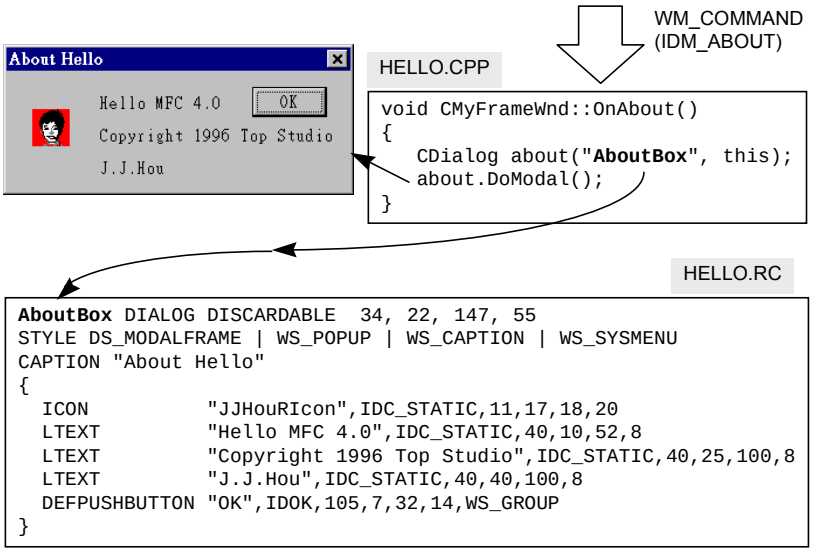
爲了輸出 *lCount*，我又動用了三個 MFC 類別：*CString*、*CRect* 和 *CDC*。前兩者非常簡單，只是字串與四方形結構的一層 C++ 包裝而且，後者是在 Windows 系統中繪圖所必須的 DC（Device Context）的一個包裝。

新版 Hello 執行結果如下。左上角的 *lCount* 以飛快的速度更迭。移動滑鼠看看，看 *lCount* 會不會重置爲 0。



Dialog 與 Control

回憶 SDK 程式中的對話盒作法：RC 檔中要準備一個對話盒的 Template，C 程式中要設計一個對話盒函式。MFC 提供的 *CDialog* 已經把對話盒的視窗函式設計好了，因此在 MFC 程式中使用對話盒非常地簡單：



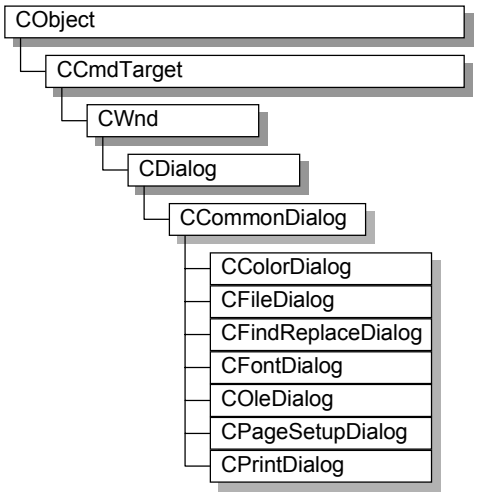
當使用者按下【File/About】選單，根據 Message Map 的設定，*WM_COMMAND*（*IDM_ABOUT*）被送到 *OnAbout* 函式去。我們首先在 *OnAbout* 中產生一個 *CDialog* 物件，名爲 *about*。*CDialog* 建構式容許兩個參數，第一個參數是對話盒的面板資源，第二個參數是 *about* 物件的主人。由於我們的 "About" 對話盒是如此地簡單，不需要改寫 *CDialog* 中的對話盒函式，所以接下來直接呼叫 *CDialog*::*DoModal*，對話盒就開始運作了。

通用對話盒（Common Dialogs）

有些對話盒，例如【File Open】或【Save As】對話盒，出現在每一個程式中的頻率是如此之高，使微軟公司不得不面對此一事實。於是，自從 Windows 3.1 之後，Windows API 多了一組通用對話盒（Common Dialogs）API 函式，系統也多了一個對應的 COMMDLG.DLL（32 位元版則爲 COMDLG32.DLL）。

MFC 也支援通用對話盒，下面是其類別與其型態：

類別	型態
CCommonDialog	以下各類別的父類別
CFileDialog	File 對話盒（Open 或 Save As）
CPrintDialog	Print 對話盒
CFindReplaceDialog	Find and Replace 對話盒
CColorDialog	Color 對話盒
CFontDialog	Font 對話盒
CPageSetupDialog	Page Setup 對話盒（MFC 4.0 新增）
COleDialog	Ole 相關對話盒



在 C/SDK 程式中，使用通用對話盒的方式是，首先填充一塊特定的結構如 *OPENFILENAME*，然後呼叫 API 函式如 *GetOpenFileName*。當函式回返，結構中的某些欄位便持有了使用者輸入的值。

MFC 通用對話盒類別，使用之簡易性亦不輸 Windows API。下面這段碼可以啓動【Open】對話盒並最後獲得檔案完整路徑：

```
char szFileters[] = "Text fiels (*.txt)|*.txt|All files (*.*)|*.*||"

CFileDialog opendlg (TRUE, "txt", "*.txt",
                 OFN_FILEMUSTEXIST | OFN_HIDEREADONLY, szFilters, this);

if (opendlg.DoModal() == IDOK) {
   filename = opendlg.GetPathName();
}
```

opendlg 建構式的第一個參數被指定爲 *TRUE*，表示我們要的是一個【Open】對話盒而不是【Save As】對話盒。第二參數 "txt" 指定預設副檔名；如果使用者輸入的檔案沒有副檔名，就自動加上此一副檔名。第三個參數 "*.txt" 出現在一開始的【file name】欄位中。*OFN_* 參數指定檔案的屬性。第五個參數 *szFilters* 指定使用者可以選擇的檔案型態，最後一個參數是父視窗。

當 *DoModal* 回返，我們可以利用 *CFileDialog* 的成員函式 *GetPathName* 取得完整的檔案路徑。也可以使用另一個成員函式 *GetFileName* 取其不含路徑的檔案名稱，或 *GetFileTitle* 取得既不含路徑亦不含副檔名的檔案名稱。

這便是 MFC 通用對話盒類別的使用。你幾乎不必再從其中衍生出子類別，直接用就好了。

本章回顧

乍看 MFC 應用程式碼，實在很難推想程式的進行。一開始是一個衍生自 *CWinApp* 的全域物件 application object，然後是一個隱藏的 *WinMain* 函式，呼叫 application object 的 *InitInstance* 函式，將程式初始化。初始化動作包括建構一個視窗物件（*CFrameWnd* 物件），而其建構式又呼叫 *CFrameWnd*::*Create* 產生真正的視窗（並在產生之前要求 MFC 註冊視窗類別）。視窗產生後 *WinMain* 又呼叫 *Run* 啓動訊息迴路，將 *WM_COMMAND*（*IDM_ABOUT*）和 *WM_PAINT* 分別交給成員函式 *OnAbout* 和 *OnPaint* 處理。

雖然刨根究底不易，但是我們都同意，MFC 應用程式碼的確比 SDK 應用程式碼精簡許多。事實上，MFC 並不打算讓應用程式碼比較容易理解，畢竟 raw Windows API 才是最直接了當的動作。許許多多細碎動作被包裝在 MFC 類別之中，降低了你寫程式的負擔，當然，這必須建立在一個事實之上：你永遠可以改變 MFC 的預設行爲。這一點是無庸置疑的，因爲所有你可能需要改變的性質，都被設計爲 MFC 類別中的虛擬函式了，你可以從 MFC 衍生出自己的類別，並改寫那些虛擬函式。

MFC 的好處在更精巧更複雜的應用程式中顯露無遺。至於複雜如 OLE 者，那就更是非 MFC 不爲功了。本章的 Hello 程式還欠缺許多 Windows 程式完整功能，但它畢竟是一個好起點，有點晦澀但不太難。下一章範例將運用 MDI、Document/View、各式各樣的 UI 物件...。

第 7 章

簡單而完整：MFC 骨幹程式

當技術愈來愈複雜，
入門愈來愈困難，
我們的困惑愈來愈深，
猶豫愈來愈多。

上一章的 Hello 範例，對於 MFC 程式設計導入很適合。但它只發揮了 MFC 的一小部份特性，只用了三個 MFC 類別（*CWinApp*、*CFrameWnd* 和 *CDialog*）。這一章我們要看一個完整的 MFC 應用程式骨幹（註），其中包括豐富的 UI 物件（如工具列、狀態列）的生成，以及很重要的 Document/View 架構觀念。

註：我所謂的 MFC 應用程式骨幹，指的是由 AppWizard 產生出來的 MFC 程式，也就是像第 4 章所產生的 Scribble step0 那樣的程式。

不二法門：熟記 MFC 類別階層架構

我還是要重複這一句話：MFC 程式設計的第一要務是熟記各類別的階層架構，並清楚了解其中幾個一定會用到的類別。一個 MFC 骨幹程式（不含 ODBC 或 OLE 支援）運用到的類別如圖 **7-1** 所示，請與圖 **6-1** 做個比較。

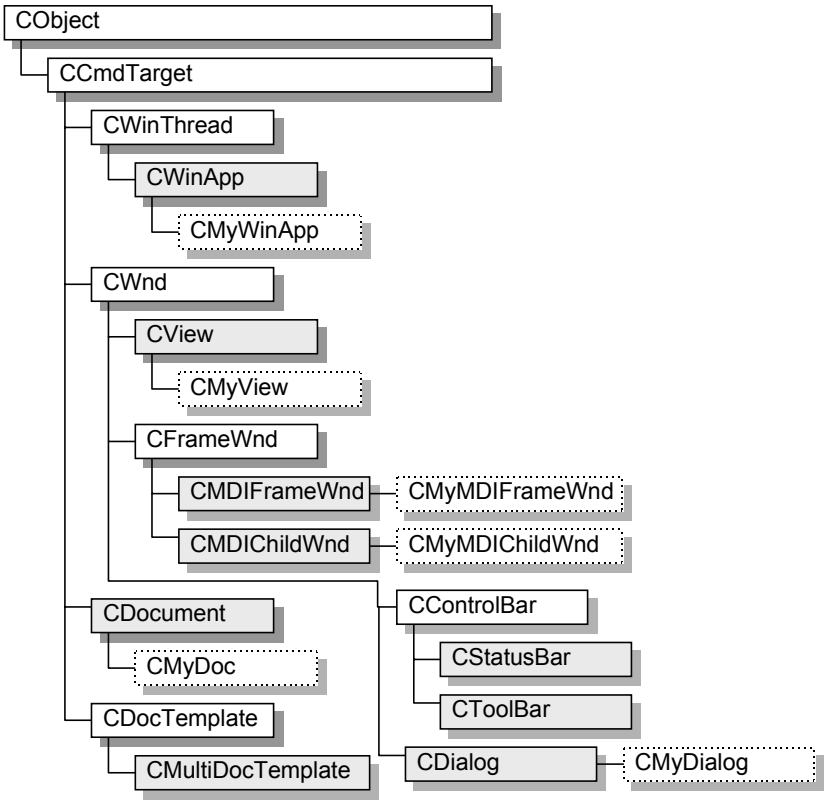
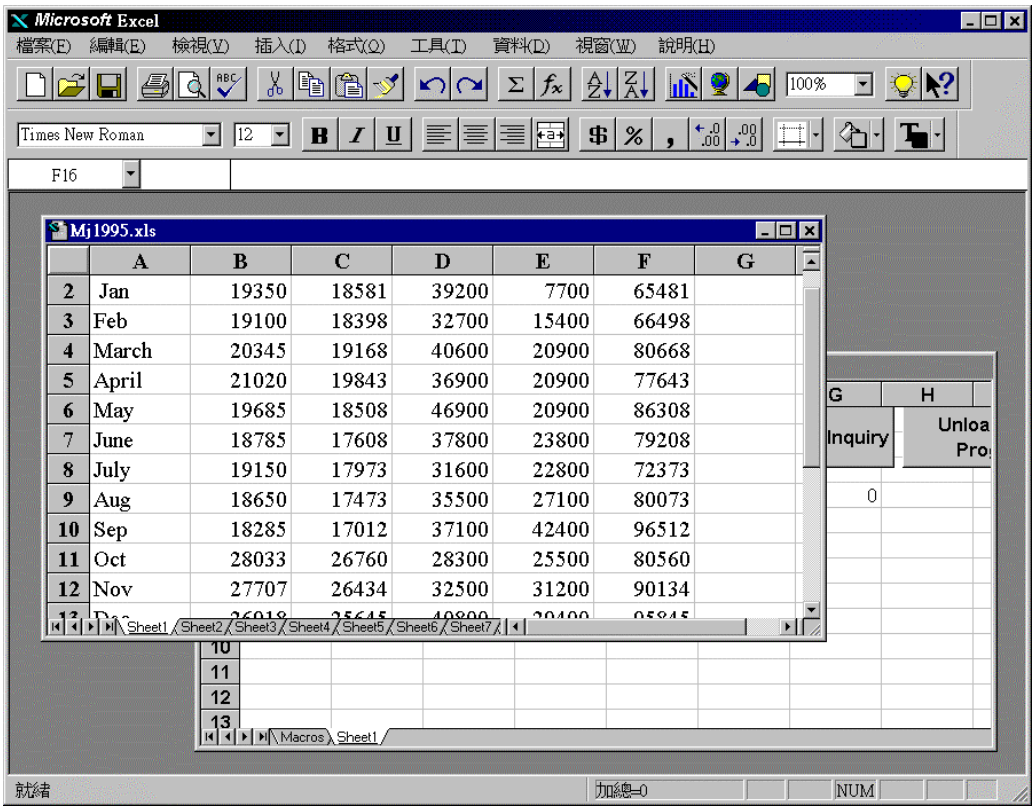


圖 7-1 本章範例程式所使用的 MFC 類別。請與圖 6-1 做比較。

MFC 程式的 UI 新風貌

一套好軟體少不得一幅漂亮的使用者介面。**圖 7-2** 是信手拈來的幾個知名 Windows 軟體，它們一致具備了工具列和狀態列等視覺物件，並擁有 MDI 風格。利用 MFC，我們很輕易就能夠做出同等級的 UI 介面。

圖 7-2a　Microsoft Word for Windows，允許使用者同時編輯多份文件，每一份文件就是所謂的 document，這些 document 視窗絕不會脫離 Word 主視窗的管轄。



	A	B	C	D	E	F	G
2	Jan	19350	18581	39200	7700	65481	
3	Feb	19100	18398	32700	15400	66498	
4	March	20345	19168	40600	20900	80668	
5	April	21020	19843	36900	20900	77643	
6	May	19685	18508	46900	20900	86308	
7	June	18785	17608	37800	23800	79208	
8	July	19150	17973	31600	22800	72373	
9	Aug	18650	17473	35500	27100	80073	
10	Sep	18285	17012	37100	42400	96512	
11	Oct	28033	26760	28300	25500	80560	
12	Nov	27707	26434	32500	31200	90134	

圖 7-2b　Microsoft Excel，允許同時製作多份報表。每一份報表就是一份 document。

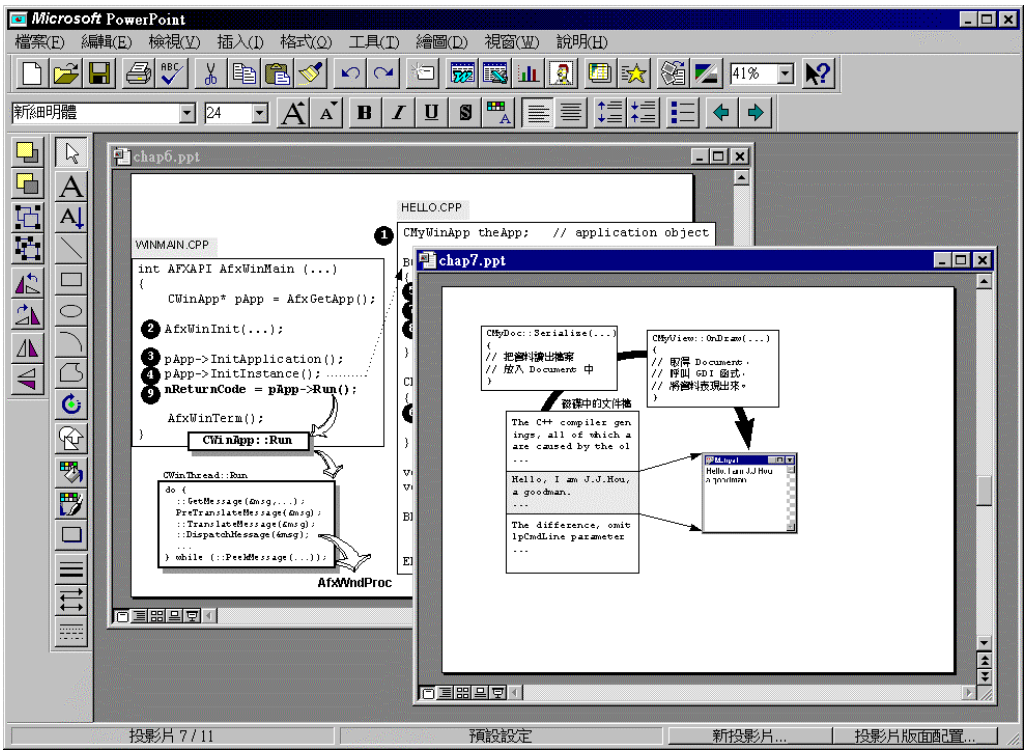
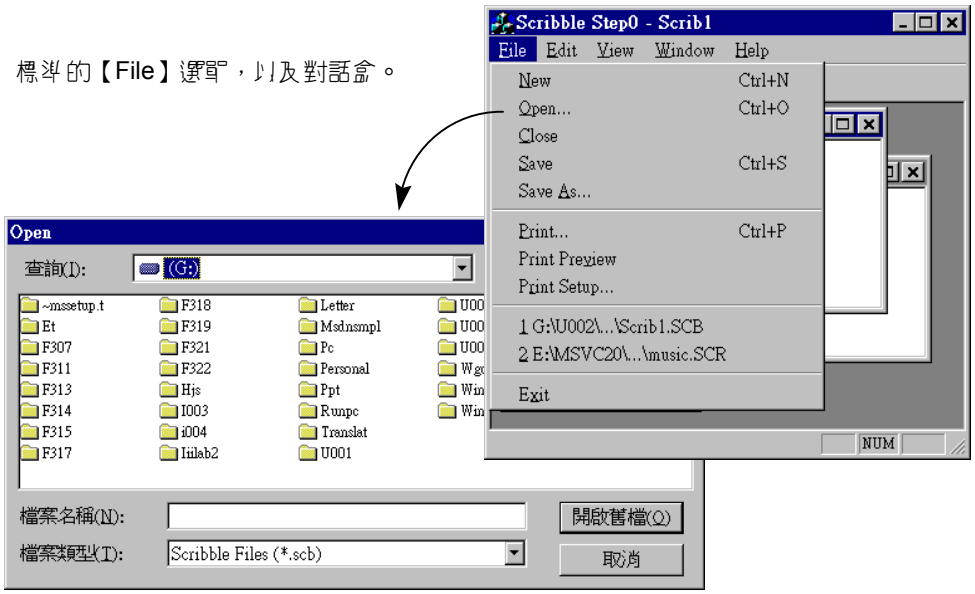


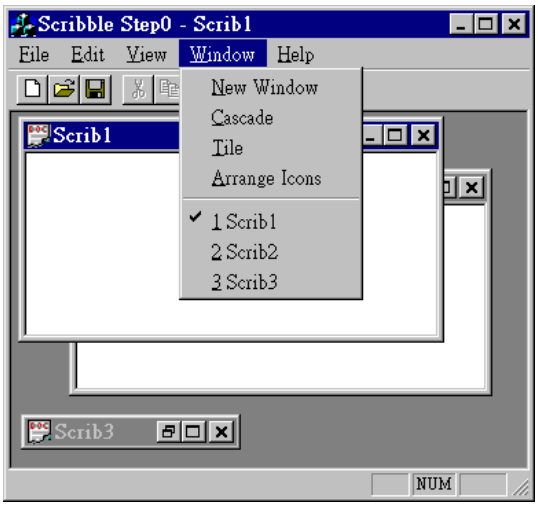
圖 7-2c　Microsoft PowerPoint 允許同時製作多份簡報資料，每一份簡報就是一份 document。

撰寫 MFC 程式，我們一定要放棄傳統的「純手工打造」方式，改用 Visual C++ 提供的各種開發工具。AppWizard 可以爲我們製作出 MFC 程式骨幹；只要選擇某些按鈕，不費吹灰之力你就可以獲得一個很漂亮的程式。這個全自動生產線做出來的程式雖不具備任何特殊功能（那正是我們程式員的任務），但已經擁有以下的特徵：

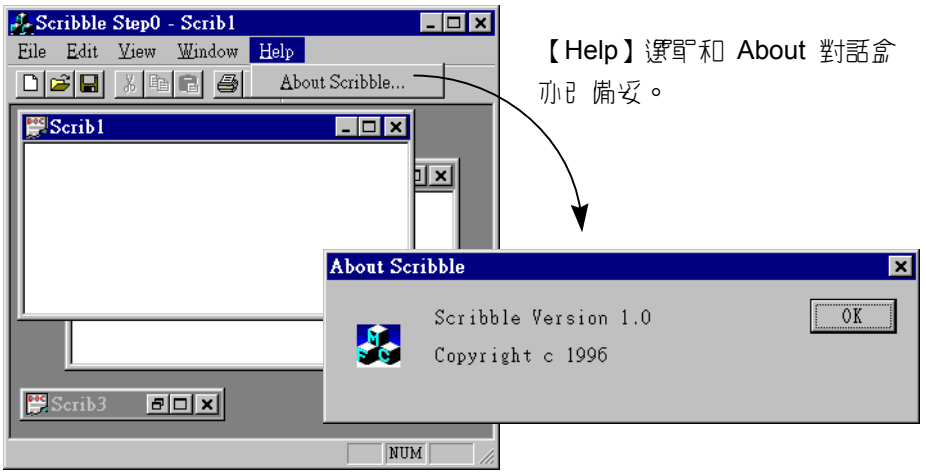
標準的【File】選單，以及對話盒。



標準的【Edit】選單（剪貼簿功能）。這份選單是否一開始就有功效，必須視你選用哪一種 View 而定，例如 CEditView 就內建有剪貼簿功能。



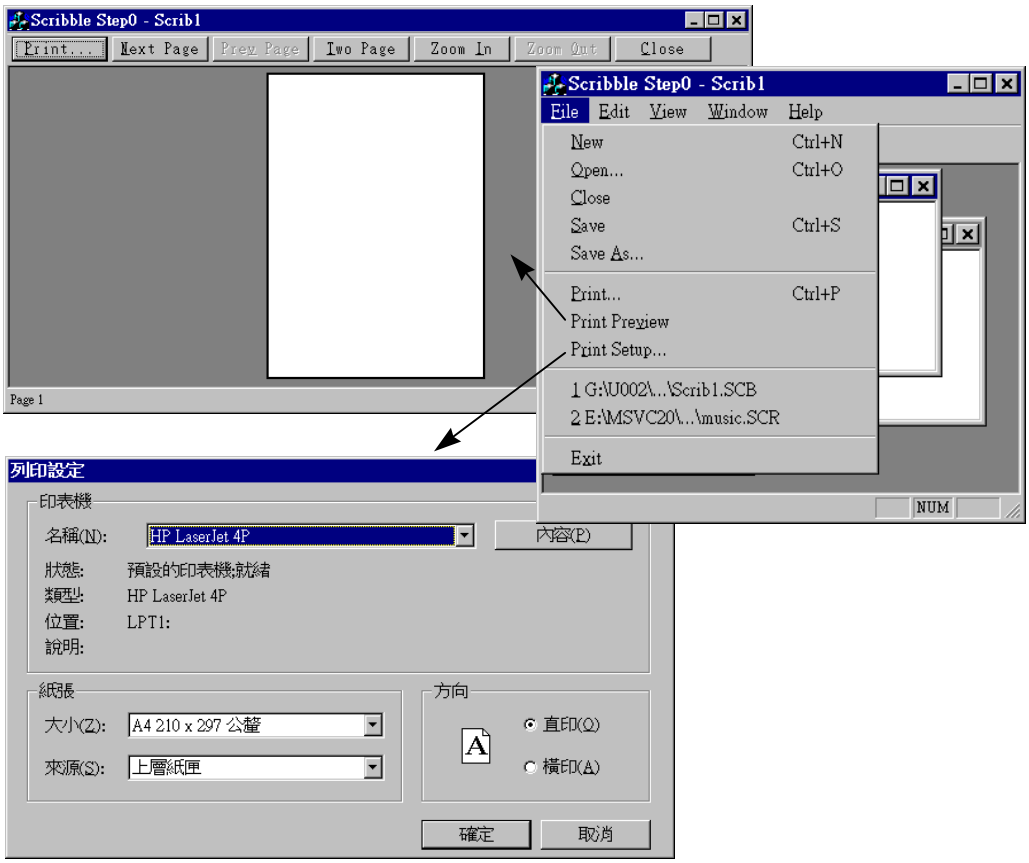
標準 MDI 程式框架具備的【Window】選單



【Help】選單和 About 對話盒亦已備妥。

此外，標準的工具列和狀態列也已備妥，並與選單內容建立起映射關係。所謂工具列，是將某幾個常用的選單項目以按鈕型式呈現出來，有一點熱鍵的味道。這個工具列可以隨處停駐（dockable）。所謂狀態列，是主視窗最下方的文字顯示區；只要選單拉下，狀態列就會顯示滑鼠座落的選單項目的說明文字。狀態列右側有三個小窗口（可擴充個數），用來顯示一些特殊按鍵的狀態。

列印與預視功能也已是半成品。【File】選單拉下來可以看到【Print...】和【Print Preview】兩項目：



骨幹程式的 Document 和 View 目前都還是白紙一張，需要我們加工，所以一開始看不出列印與預視的真正功能。但如果我們在 AppWizard 中選用的 View 類別是 *CEditView*（如同第 4 章 292 頁），使用者就可以列印其編輯成果，並可以在列印之前預視。也就是說，一行程式碼都不必寫，我們就獲得了一個可以同時編輯多份文件的文字編輯軟體。

Document/View 支撐你的應用程式

我已經多次強調，Document/View 是 MFC 進化爲 Application Framework 的靈魂。這個特徵表現於程式設計技術上遠多於表現在使用者介面上，因此使用者可能感覺不到什麼是 Document/View。程式員呢？程式員將因陌生而有一段陣痛期，然後開始享受它帶來的便利。

我們在 OLE 中看到各物件（註）的集合稱爲一份 Document；在 MDI 中看到子視窗所掌握的資料稱爲一個 Document；現在在 MFC 又看到 Document。"Document" 如今處處可見，再過不多久八成也要和 "Object" 一樣地氾濫了。

> OLE 物件指的是 PaintBrush 完成的一張 bitmap、 SoundRecorder 完成的一段 Wave 聲音、Excel 完成的一份試算表、Word 完成的一份文字等等等。爲了恐怕與 C++ 的「物件」混淆，有些書籍將 OLE object 稱爲 OLE item。

在 MFC 之中，你可以把 Document 簡單想作是「資料」。是的，只是資料，那麼 MFC 的 *CDocument* 簡單地說就是負責處理資料的類別。

問題是，一個預先寫好的類別怎麼可能管理未知的資料呢？MFC 設計之際那些偉大的天才們並不知道我們的資料結構，不是嗎 ?! 他怎麼知道我的程式要處理的資料是簡單如：

```
char  name[20];
char  address[30];
int   age;
bool  sex;
```

或是複雜如：

```
struct dbllistnode
      {
        struct dbllistnode *next, *prev;
        struct info_t
              {
               int left;
               int top;
```

```
            int width;
            int height;
            void (*cursor)();
           } *item;
};
```

的確，預先處理未知的資料根本是不可能的。*CDocument* 只是把空殼做好，等君入甕。它可以內嵌其他物件（用來處理基層資料型態如串列、陣列等等），所以程式員可以在 Document 中拼拼湊湊出實際想要表達的文件完整格式。下一章進入 Scribble 程式的實際設計時，你就能夠感受這一點。

CDocument 的另一價值在於它搭配了另一個重要的類別：*CView*。

不論什麼型式，資料總是有體有面。實際的資料數值就是體，顯示在螢幕上（甚而印表機上）的畫面就是面（圖 **7-3a**）。「數值的處理」應該使用位元組、整數、浮點數、串列、陣列等資料結構，而「數值的表現」應該使用繪圖工具如座標系統、筆刷顏色、點線圓弧、字形...。*CView* 就是爲了資料的表現而設計的。

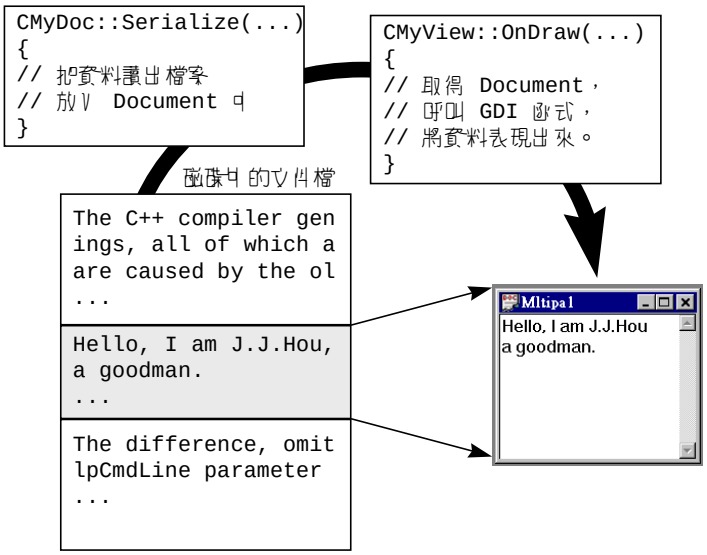


圖 7-3a　Document 是資料的體，View 是資料的面。

除了負責顯示，View 還負責程式與使用者之間的交談介面。使用者對資料的編輯、修改都需仰賴視窗上的滑鼠與鍵盤動作才得完成，這些訊息都將由 View 接受後再通知 Document (圖 **7-3b**)。

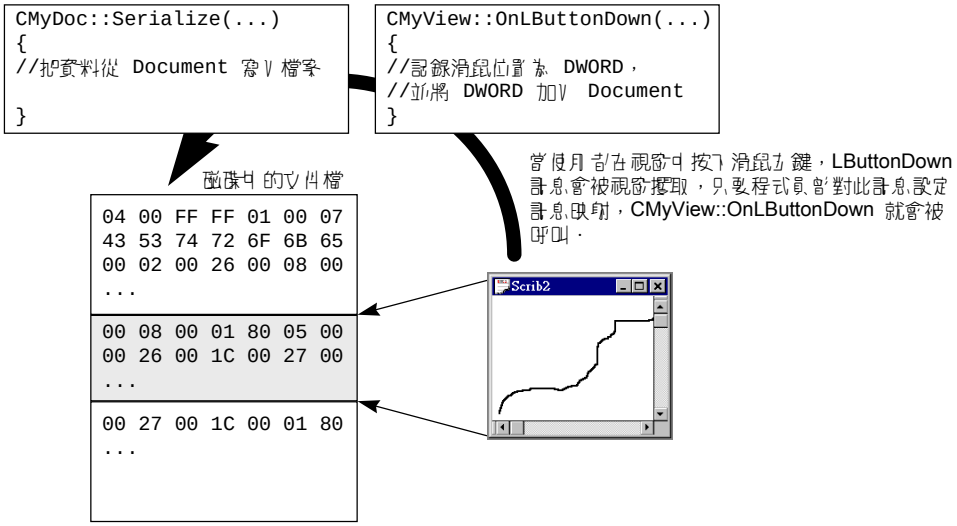


圖 7-3b　View 是 Document 的第一線，負責與使用者接觸。

Document/View 的價值在於，這些 MFC 類別已經把一個應用程式所需的「資料處理與顯示」的函式空殼都設計好了，這些函式都是虛擬函式，所以你可以（也應該）在衍生類別中改寫它們。有關檔案讀寫的動作在 *CDocument* 的 *Serialize* 函式進行，有關畫面顯示的動作在 *CView* 的 *OnDraw* 或 *OnPaint* 函式進行。當我爲自己衍生兩個類別 *CMyDoc* 和 *CMyView*，我只要把全付心思花在 *CMyDoc*::*Serialize* 和 *CMyView*::*OnDraw* 身上，其他瑣事一概不必管，整個程式自動會運作得好好的。

什麼叫做「整個程式會自動運作良好」？以下是三個例子：

- 如果按下【File/Open】，Application Framework 會啓動對話盒讓你指定檔名，然後自動呼叫 *CMyDoc*::*Serialize* 讀檔。Application Framework 還會呼叫 *CMyView*::*OnDraw*，把資料顯示出來。
- 如果螢幕狀態改變，產生了 *WM_PAINT*，Framework 會自動呼叫你的 *CMyView*::*OnDraw*，傳一個 Display DC 讓你重新繪製視窗內容。
- 如果按下【File/Print...】，Framework 會自動呼叫你的 *CMyView*::*OnDraw*，這次傳進去的是個 Printer DC，因此繪圖動作的輸出對象就成了印表機。

MFC 已經把程式大架構完成了，模組與模組間的訊息流動路徑以及各函式的功能職司都已確定好（這是 MFC 之所以夠格稱爲一個 Framework 的原因），所以我們寫程式的焦點就放在那些必須改寫的虛擬函式身上即可。軟體界當初發展 GUI 系統時，目的也是希望把程式員的心力導引到應用軟體的真正目標去，而不必花在使用者介面上。MFC 的 Document/View 架構希望更把程式員的心力導引到真正的資料結構設計以及真正的資料顯示動作上，而不要花在模組的溝通或訊息的流動傳遞上。今天，程式員都對 GUI 稱便，Document/View 也即將廣泛地證明它的貢獻。

Application Framework 使我們的程式寫作猶如做填充題；Visual C++ 的軟體開發工具則使我們的程式寫作猶如做選擇題。我們先做選擇題，再在骨幹程式中做填充題。的確，程式員的生活愈來愈像侯捷所言「只是軟體 IC 裝配廠裡的男工女工」了。

現在讓我們展開 MFC 深度之旅，徹底把 MFC 骨幹程式的每一行都搞清楚。你應該已經從上一章具體了解了 MFC 程式從啓動到結束的生命過程，這一章的例子雖然比較複雜，程式的生命過程是一樣的。我們看看新添了什麼內容，以及它們如何運作。我將以 AppWizard 完成的 Scribble Step0（第４章）爲解說對象，一行不改。然後我會做一點點修改，使它成爲一個多視窗文字編輯器。

利用 Visual C++ 工具完成 Scribble step0

我已經在第４章示範過 AppWizard 的使用方法，並實際製作出 Scribble Step0 程式，這裡就不再重複說明了。完整的骨幹程式原始碼亦已列於第４章。

這些由「生產線」做出來的程式碼其實對初學者並不十分合適，原因之一是容易眼花撩亂，有許多 `#if...#endif`、註解、奇奇怪怪的符號（例如 `//{` 和 `//}`）；原因之二是每一個類別有自己的 .H 檔和 .CPP 檔，整個程式因而幅員遼闊（六個 .CPP 檔和六個 .H 檔）。

圖 **7-4** 是 Scribble step0 程式中各類別的相關資料。

類別名稱	基礎類別	類別宣告於	類別定義於
CScribbleApp	*CWinApp*	Scribble.h	Scribble.cpp
CMainFrame	*CMDIFrameWnd*	Mainfrm.h	Mainfrm.cpp
CChildFrame	*CMDIChildWnd*	Childfrm.h	Childfrm.cpp
CScribbleDoc	*CDocument*	ScribbleDoc.h	ScribbleDoc.cpp
CScribbleView	*CView*	ScribbleView.h	ScribbleView.cpp
CAboutDlg	*CDialog*	Scribble.cpp	Scribble.cpp

圖 7-4　Scribble 骨幹程式中的重要組成份子

骨幹程式使用哪些 MFC 類別？

對，你看到的 Scribble step0 就是一個完整的 MFC 應用程式，而我保證你一定昏頭轉向茫無頭緒。沒有關係，我們才剛啓航。

如果把標準圖形介面（工具列和狀態列）以及 Document/View 考慮在內，一個標準的 MFC MDI 程式使用這些類別：

MFC 類別名稱	我的類別名稱	功能
CWinApp	*CScribbleApp*	application object
CMDIFrameWnd	*CMainFrame*	MDI 主視窗
CMultiDocTemplate	直接使用	管理 Document/View
CDocument	*CScribbleDoc*	Document，負責資料結構與檔案動作
CView	*CScribbleView*	View，負責資料的顯示與印表
CMDIChildWnd	*CChildFrame*	MDI 子視窗
CToolBar	直接使用	工具列
CStatusBar	直接使用	狀態列
CDialog	*CAboutDlg*	About 對話盒

應用程式各顯身手的地方只是各個可被改寫的虛擬函式。這九個類別在 MFC 的地位請看**圖 7-1**。下一節開始我會逐項解釋每一個物件的產生時機及其重要性質。

Document/View 不只應用在 MDI 程式，也應用在 SDI 程式上。你可以在 AppWizard 的「Options 對話盒」（**圖 4-2b**）選擇 SDI 風格。本書以 MDI 程式爲討論對象。

爲了對標準的 MFC 程式有一個大局觀，**圖 7-4** 顯示 Scribble step0 中各重要組成份子（類別），這些組成份子在執行時期的意義與主從關係顯示於**圖 7-5**。

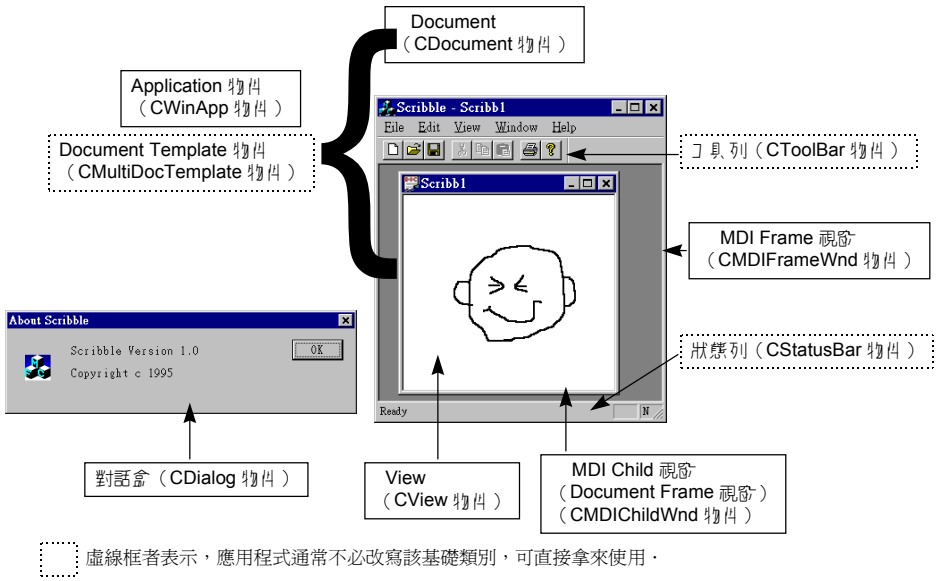


圖 7-5　Scribble step0 程式中的九個物件（幾乎每個 MFC MDI 程式都如此）。

圖 **7-6** 是 Scribble step0 程式縮影，我把執行時序標上去，對於整體概念的形成將有幫助。

```
#0001 class CScribbleApp : public CWinApp
#0002 {
#0003    virtual BOOL InitInstance();   // 注意：第 6 章的 HelloMFC 程式是在 CFrameWnd
#0004    afx_msg void OnAppAbout();     // 類別中處理 "About" 命令，這裡的 Scribble
#0005    DECLARE_MESSAGE_MAP()          // 程式卻在 CWinApp 衍生類別中處理之。到底，
#0006 };                                // 一個訊息可以（或應該）在哪裡被處理才是合理？
#0007                                   // 第 9 章「訊息映射與命令繞行」可以解決這個疑惑。
#0008 class CMainFrame : public CMDIFrameWnd
#0009 {
#0010    DECLARE_DYNAMIC(CMainFrame)
#0011    CStatusBar  m_wndStatusBar;
#0012    CToolBar    m_wndToolBar;
#0013    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
#0014    DECLARE_MESSAGE_MAP()
#0015 };
#0016
#0017 class CChildFrame : public CMDIChildWnd
#0018 {
```

```
#0019    DECLARE_DYNCREATE(CChildFrame)
#0020    DECLARE_MESSAGE_MAP()
#0021 };
#0022
#0023 class CScribbleDoc : public CDocument
#0024 {
#0025    DECLARE_DYNCREATE(CScribbleDoc)
#0026   virtual void Serialize(CArchive& ar);
#0027    DECLARE_MESSAGE_MAP()
#0028 };
#0029
#0030 class CScribbleView : public CView
#0031 {
#0032    DECLARE_DYNCREATE(CScribbleView)
#0033    CScribbleDoc* GetDocument();
#0034
#0035    virtual void OnDraw(CDC* pDC);
#0036   DECLARE_MESSAGE_MAP()
#0037 };
#0038
#0039 class CAboutDlg : public CDialog
#0040 {
#0041       DECLARE_MESSAGE_MAP()
#0042 };
#0043
#0044 //----------------------------------
#0045
#0046 CScribbleApp theApp; ①
#0047
#0048 BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
#0049    ON_COMMAND(ID_APP_ABOUT, OnAppAbout) ❹
#0050    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
#0051    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen) ❷
#0052    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
#0053 END_MESSAGE_MAP()
#0054
#0055 BOOL CScribbleApp::InitInstance() ⑤
#0056 {
#0057    ...
#0058    CMultiDocTemplate* pDocTemplate;
#0059    pDocTemplate = new CMultiDocTemplate( ⑥
#0060           IDR_SCRIBTYPE,
#0061           RUNTIME_CLASS(CScribbleDoc),
#0062           RUNTIME_CLASS(CChildFrame),
#0063           RUNTIME_CLASS(CScribbleView));
#0064    AddDocTemplate(pDocTemplate);
```

```
// MFC 內部
Int AfxAPI AfxWinMain(¡K
{
  CWinApp* pApp = AfxGetApp();
  AfxWinInit(¡K;                  ②
  pApp->InitApplication();        ③
  pApp->InitInstance();           ④
  nReturnCode = pApp->Run();      ❶
```

```
#0065
#0066    CMainFrame* pMainFrame = new CMainFrame;  ⑦
#0067     pMainFrame->LoadFrame(IDR_MAINFRAME);    ⑧
#0068     m_pMainWnd = pMainFrame;
#0069     ¡ K
#0070     pMainFrame->ShowWindow(m_nCmdShow);  ⑩
#0071     pMainFrame->UpdateWindow();
#0072     return TRUE;
#0073 }
#0074
#0075 BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
#0076 END_MESSAGE_MAP()
#0077
#0078 void CScribbleApp::OnAppAbout()  ❺
#0079 {
#0080     CAboutDlg aboutDlg;
#0081     aboutDlg.DoModal();
#0082 }
#0083
#0084 IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
#0085
#0086 BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
#0087     ON_WM_CREATE()
#0088 END_MESSAGE_MAP()
#0089
#0090 static UINT indicators[] =
#0091 {
#0092     ID_SEPARATOR,
#0093     ID_INDICATOR_CAPS,
#0094     ID_INDICATOR_NUM,
#0095     ID_INDICATOR_SCRL,
#0096 };
#0097
#0098 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)  ⑨
#0099 {
#0100     m_wndToolBar.Create(this);
#0101     m_wndToolBar.LoadToolBar(IDR_MAINFRAME);
#0102     m_wndStatusBar.Create(this);
#0103     m_wndStatusBar.SetIndicators(indicators,
#0104              sizeof(indicators)/sizeof(UINT));
#0105
#0106     m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
#0107           CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
#0108
#0109     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

```
// MFC 內部
CFrameWnd::Create
        ▼
CWnd::CreateEx
        ▼
::CreateWindowEx
```

WM_CREATE

```
#0110    EnableDocking(CBRS_ALIGN_ANY);
#0111    DockControlBar(&m_wndToolBar);
#0112    return 0;
#0113 }
#0114
#0115 IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
#0116
#0117 BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
#0118 END_MESSAGE_MAP()
#0119
#0120 IMPLEMENT_DYNCREATE(CScribbleDoc, CDocument)
#0121
#0122 BEGIN_MESSAGE_MAP(CScribbleDoc, CDocument)
#0123 END_MESSAGE_MAP()
#0124
#0125 void CScribbleDoc::Serialize(CArchive& ar)  ❸
#0126 {
#0127    if (ar.IsStoring())
#0128    {
#0129           // TODO: add storing code here
#0130    }
#0131    else
#0132    {
#0133           // TODO: add loading code here
#0134    }
#0135 }
#0136
#0137 IMPLEMENT_DYNCREATE(CScribbleView, CView)
#0138
#0139 BEGIN_MESSAGE_MAP(CScribbleView, CView)
#0140    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
#0141    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
#0142    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
#0143 END_MESSAGE_MAP()
#0144
#0145 void CScribbleView::OnDraw(CDC* pDC)
#0146 {
#0147    CScribbleDoc* pDoc = GetDocument();
#0148    // TODO: add draw code for native data here
#0149 }
```

圖 7-6　Scribble step0 執行時序。這是一張簡圖，有一些次要動作（例如滑鼠拉曳功能、設定對話盒底色）並未列出，但是在稍後的細部討論中會提到。

以下是圖 **7-6** 程式流程之說明：

①~④　動作與流程和前一章的 Hello 程式如出一轍。

⑤　我們改寫 *InitInstance* 這個虛擬函式。

⑥　*new* 一個 *CMultiDocTemplate* 物件，此物件規劃 Document、View 以及 Document Frame 視窗三者之關係。

⑦　*new* 一個 *CMyMDIFrameWnd* 物件，做爲主視窗物件。

⑧　呼叫 *LoadFrame*，產生主視窗並加掛選單等諸元，並指定視窗標題、文件標題、文件檔副檔名等（關鍵在 IDR_MAINFRAME 常數）。*LoadFrame* 內部將呼叫 *Create*，後者將呼叫 *CreateWindowEx*，於是觸發 *WM_CREATE* 訊息。

⑨　由於我們曾於 *CMainFrame* 之中攔截 *WM_CREATE*（利用 *ON_WM_CREATE* 巨集），所以 *WM_CREATE* 產生之際 Framework 會呼叫 *OnCreate*。我們在此爲主視窗掛上工具列和狀態列。

⑩　回到 *InitInstance*，執行 *ShowWindow* 顯示視窗。

❶　*InitInstance* 結束，回到 *AfxWinMain*，執行 *Run*，進入訊息迴圈。其間的黑盒子已在上一章的 Hello 範例中挖掘過。

❷　訊息經由 Message Routing 機制，在各類別的 Message Map 中尋求其處理常式。*WM_COMMAND*/*ID_FILE_OPEN* 訊息將由 *CWinApp*::*OnFileOpen* 函式處理。此函式由 MFC 提供，它在顯示過【File Open】對話盒後呼叫 *Serialize* 函式。

❸　我們改寫 *Serialize* 函式以進行我們自己的檔案讀寫動作。

❹　*WM_COMMAND*/*ID_APP_ABOUT* 訊息將由 *OnAppAbout* 函式處理。

❺　*OnAppAbout* 函式利用 *CDialog* 的性質很方便地產生一個對話盒。

Document Template 的意義

Document Template 是一個全新的觀念。

稍早我已提過 Document/View 的概念，它們互爲表裡。View 本身雖然已經是一個視窗，其外圍卻必須再包裝一個外框視窗做爲舞台。這樣的切割其實是爲了讓 View 可以非常獨立地放置於「MDI Document Frame 視窗」或「SDI Document Frame 視窗」或「OLE Document Frame 視窗」等各種應用之中。也可以說，Document Frame 視窗是 View 視窗的一個容器。資料的內容、資料的表象、以及「容納資料表象之外框視窗」三者是一體的，換言之，程式每打開一份文件（資料），就應該產生三份物件：

1. 一份 Document 物件，
2. 一份 View 物件，
3. 一份 *CMDIChildWnd* 物件（做爲外框視窗）

這三份物件由一個所謂的 Document Template 物件來管理。讓這三份物件產生關係的關鍵在於 *CMultiDocTemplate*：

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

如果程式支援不同的資料格式（例如一爲 TEXT 一爲 BITMAP），那麼就需要不同的 Document Template：

```
BOOL CMyWinApp::InitInstance()
{
  ...
  CMultiDocTemplate* pDocTemplate;

  pDocTemplate = new CMultiDocTemplate(
          IDR_TEXTTYPE,
          RUNTIME_CLASS(CTextDoc),
          RUNTIME_CLASS(CChildFrame),
          RUNTIME_CLASS(CTextView));
  AddDocTemplate(pDocTemplate);

  pDocTemplate = new CMultiDocTemplate(
          IDR_BMPTYPE,
          RUNTIME_CLASS(CBmpDoc),
          RUNTIME_CLASS(CChildFrame),
          RUNTIME_CLASS(CBmpView));
  AddDocTemplate(pDocTemplate);
  ...
}
```

這其中有許多值得討論的地方，而 *CMultiDocTemplate* 的建構式參數透露了一些端倪：

```
CMultiDocTemplate::CMultiDocTemplate(UINT nIDResource,
                                     CRuntimeClass* pDocClass,
                                     CRuntimeClass* pFrameClass,
                                     CRuntimeClass* pViewClass);
```

1. *nIDResource*：這是一個資源 ID，表示此一文件型態（文件格式）所使用的資源。本例爲 *IDR_SCRIBTYPE*，在 RC 檔中代表多種資源（不同種類的資源可使用相同的 ID）：


```
IDR_SCRIBTYPE ICON DISCARDABLE  "res\\ScribbleDoc.ico"
IDR_SCRIBTYPE MENU PRELOAD DISCARDABLE
{ ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME "Scribble Step0"
   IDR_SCRIBTYPE "\nScrib\nScrib\nScribble Files (*.scb)\n.SCB\n
                  Scribble.Document\nScrib Document"
END
```

原碼太長，我把它截爲兩半，實際上是一整行。有七個子字串各以 '\n' 分隔。這些子字串完整描述文件的型態。第一個子字串於 MDI 程式中用不著，故本例省略（成爲空字串）。

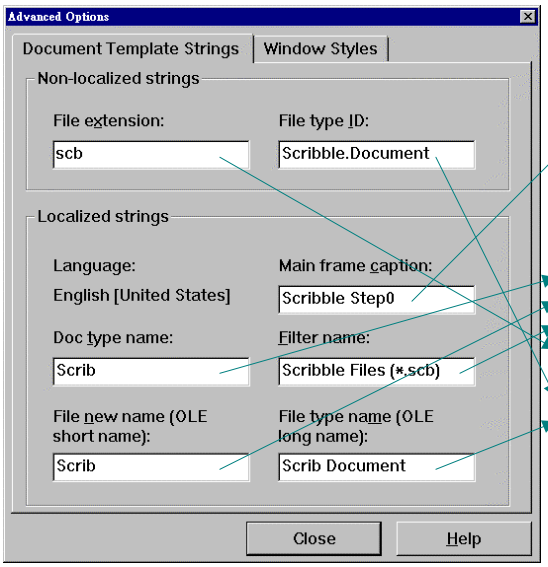
其中的 ICON 是文件視窗被最小化之後的圖示；MENU 是當程式存在有任何文件視窗時所使用的選單(如果沒有開啓任何文件視窗,選單將是另外一套,稍後再述)。至於字串表格（STRINGTABLE）中的字串，稍後我有更進一步的說明。

2. *pDocClass*。這是一個指標，指向 Document 類別（衍生自 *CDocument*）之「*CRuntimeClass* 物件」。
3. *pFrameClass*。這是一個指標，指向 Child Frame 類別（衍生自 *CMDIChildWnd*）之「*CRuntimeClass* 物件」。
4. *pViewClass*。這是一個指標，指向 View 類別(衍生自 *CView*)之「*CRuntimeClass* 物件」。

CRuntimeClass
我曾經在第 3 章「自製 RTTI」一節解釋過什麼是 *CRuntimeClass*。它就是「類別型錄網」裡列出的元素型態。任何一個類別只要在宣告時使用 *DECLARE_DYNAMIC* 或 *DECLARE_DYNCREATE* 或 *DECLARE_SERIAL* 巨集，就會擁有一個靜態的（static）*CRuntimeClass* 內嵌物件。

好，你看，Document Template 接受了三種類別的 *CRuntimeClass* 指標，於是每當使用者打開一份文件，Document Template 就能夠根據「類別型錄網」（第 3 章所述），動態生成出三個物件（document、view、document frame window）。如果你不記得 MFC 的動態生成是怎麼一回事兒，現在正是複習第 3 章的時候。我將在第 8 章帶你實際看看 Document Template 的內部動作。

前面曾提到，我們在 *CMultiDocTemplate* 建構式的第一個參數置入 *IDR_SCRIBTYPE*，代表 RC 檔中的選單（MENU）、圖示（ICON）、字串（STRING）三種資源，其中又以字串資源大有學問。這個字串以 '\n' 分隔爲七個子字串，用以完整描述文件型態。七個子字串可以在 AppWizard 的步驟四的【Advanced Options】對話盒中指定：



RC 檔中的字串資源

IDR_MAINFRAME：

```
"Scribble Step0"
```

IDR_SCRIBTYPE（7 個子字串）：

```
\n
Scrib\n
Scrib\n
Scribble Files (*.scb)\n
.SCB\n
Scribble.Document\n
Scrib Document
```

每一個子字串都可以在程式進行過程中取得，只要呼叫 *CDocTemplate*::*GetDocString* 並在其第二參數中指定索引值（1~7）即可，但最好是以 *CDocTemplate* 所定義的七個常數代替沒有字面意義的索引值。下面就是 *CDocTemplate* 的 7 個常數定義：

```
// in AFXWIN.H
class CDocTemplate : public CCmdTarget
{
  ...
  enum DocStringIndex
  {
    windowTitle,  // default window title
    docName,      // user visible name for default document
    fileNewName,  // user visible name for FileNew

    // for file based documents:
    filterName,   // user visible name for FileOpen
    filterExt,    // user visible extension for FileOpen

    // for file based documents with Shell open support:
    regFileTypeId,   // REGEDIT visible registered file type identifier
    regFileTypeName  // Shell visible registered file type name
  };
  ...
};
```

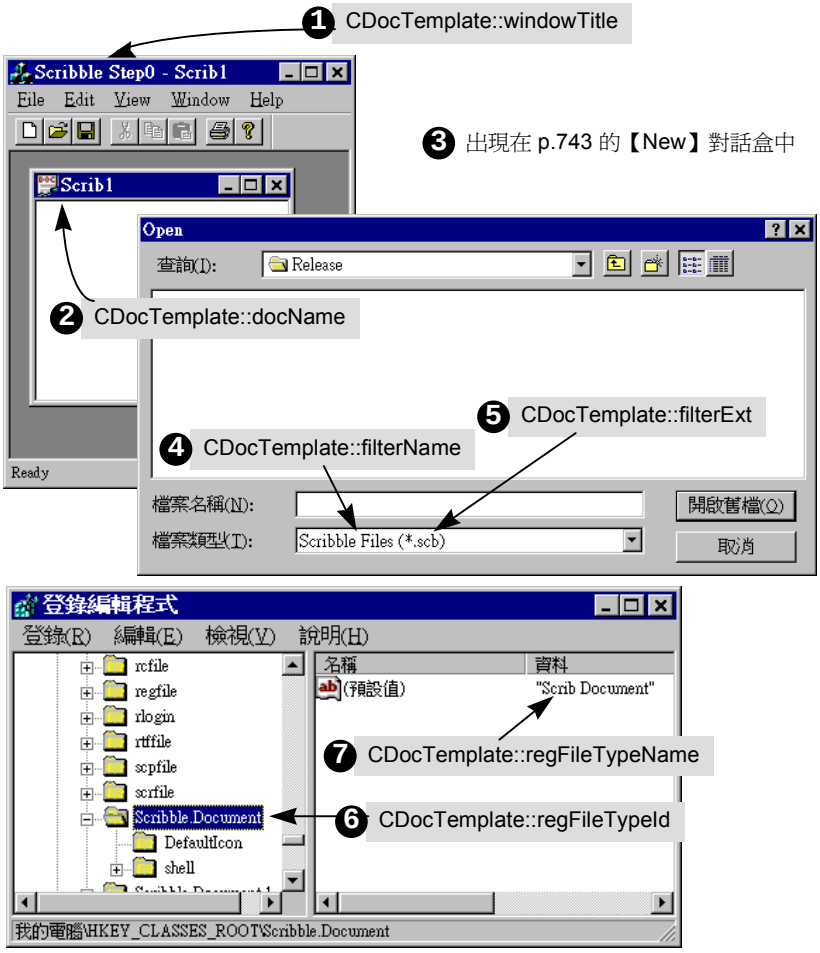
所以，你可以這麼做：

```
CString strDefExt, strDocName;
pDocTemplate->GetDocString(strDefExt, CDocTemplate::filterExt);
pDocTemplate->GetDocString(strDocName, CDocTemplate::docName);
```

七個子字串意義如下：

index	意義
1. *CDocTemplate::windowTitle*	主視窗標題欄上的字串。SDI 程式才需要指定它，MDI 程式不需要指定，將以 *IDR_MAINFRAME* 字串爲預設值。
2. *CDocTemplate::docName*	文件基底名稱（本例爲 "Scrib"）。這個名稱再加上一個流水號碼，即成爲新文件的名稱（例如 "Scrib1"）。如果此字串未被指定，文件預設名稱爲 "Untitled"。
3. *CDocTemplate::fileNewName*	文件型態名稱。如果一個程式支援多種文件，此字串將顯示在【File/New】對話盒中。如果沒有指明，就不能夠在【File/New】對話盒中處理此種文件。本例只支援一種文件型態，所以當你選按【File/New】，並不會出現對話盒。第 13 章將示範「一個程式支援多種文件」的作法（#743 頁）。
4. *CDocTemplate::filterName*	文件型態以及一個適用於此型態之萬用過濾字串 (wildcard filter string)。本例爲 "Scribble(*.scb)"。這個字串將出現在【File Open】對話盒中的【List Files Of Type】列示盒中。
5. *CDocTemplate::filterExt*	文件檔之副檔名，例如 "scb"。如果沒有指明，就不能夠在【File Open】對話盒中處理此種文件檔。
6. *CDocTemplate::regFileTypeId*	如果你以 ::RegisterShellFileTypes 對系統的登錄資料庫（Registry）註冊文件型態，此值會出現在 HKEY_CLASSES_ROOT 之下成爲其子機碼（subkey）並僅供 Windows 內部使用。如果未指定，此種文件型態就無法註冊，滑鼠拖放 (drag and drop) 功能就會受影響。
7. *CDocTemplate::regFileTypeName*	這也是儲存在登錄資料庫（Registry）中的文件型態名稱，並且是給人（而非只給系統）看的。它也會顯示於程式中用以處理登錄資料庫之對話盒內。

以下是 Scribble 範例中各個字串出現的位置：



我必須再強調一次，AppWizard 早已幫我們產生出這些字串。把這些來龍去脈弄清楚，只是爲了知其所以然。當然，同時也爲了萬一你不喜歡 AppWizard 準備的字串內容，你知道如何去改變它。

Scribble 的 Document/View 設計

用最簡單的一句話描述，Document 就是資料的體，View 就是資料的面。我們藉 *CDocument* 管理資料，藉 Collections Classes（MFC 中的一組專門用來處理資料的類別）處理實際的資料數據；我們藉 *CView* 負責資料的顯示，藉 *CDC* 和 *CGdiObject* 實際繪圖。人們常說一體兩面一體兩面，在 MFC 中一體可以多面：同一份資料可以文字描述之，可以長條圖描述之，亦可以曲線圖描述之。

Document/View 之間的關係可以圖 **7-3** 說明。View 就像一個觀景器（我避免使用「視窗」這個字眼，以免引起不必要的聯想），使用者透過 View 看到 Document，也透過 View 改變 Document。View 是 Document 的外顯介面，但它並不能完全獨立，它必須依存在一個所謂的 Document Frame 視窗內。

一份 Document 可以映射給許多個 Views 顯示，不同的 Views 可以對映到同一份巨大 Document 的不同區域。總之，請把 View 想像是一個鏡頭，可以觀看大畫布上的任何區域（我們可以選用 *CScrollView* 使之具備捲軸）；在鏡頭上加特殊的偏光鏡、柔光鏡、十字鏡，我們就會看到不同的影像 -- 雖然觀察的對象完全相同。

資料的管理動作有哪些？讀檔和寫檔都是必要的，檔案存取動作稱爲 Serialization，由 *Serialize* 函式負責。我們可以（而且也應該）在 *CMyDoc* 中改寫 *Serialize* 函式，使它符合個人需求。資料格式的建立以及檔案讀寫功能將在 Scribble step1 中加入，本例（step0）的 *CScribbleDoc* 中並沒有什麼成員變數（也就是說容納不了什麼資料），*Serialize* 則簡直是個空函式：

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

這也是我老說骨幹程式啥大事也沒做的原因。

除了檔案讀寫，資料的顯示也是必要的動作，資料的接受編輯也是必要的動作。兩者都由 View負責。使用者對 Document 的任何編輯動作都必須透過 Document Frame 視窗，訊息隨後傳到 *CView*。我們來想想我們的 View 應該改寫哪些函式？

1. 當 Document Frame 視窗收到 *WM_PAINT*，視窗內的 View 的 *OnPaint* 函式會被呼叫，*OnPaint* 又呼叫 *OnDraw*。所以爲了顯示資料，我們必須改寫 *OnDraw*。至於 *OnPaint*，主要是做「只輸出到螢幕而不到印表機」的動作。有關印表機，我將在第 12 章提到。
2. 爲了接受編輯動作，我們必須在 View 類別中接受滑鼠或鍵盤訊息並處理之。如果要接受滑鼠左鍵，我們應該改寫 View 類別中的三個虛擬函式：

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```

上述兩個動作在 Scribble step0 都看不到，因爲它是個啥也沒做的程式：

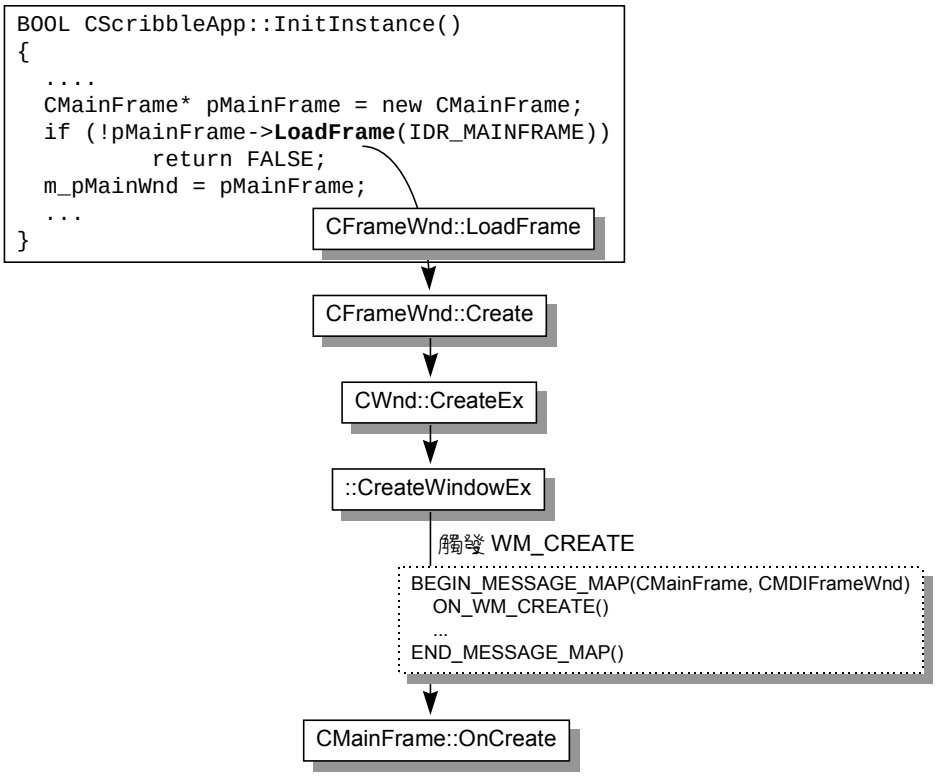
```
void CScribbleView::OnDraw(CDC* pDC)
{
       CScribbleDoc* pDoc = GetDocument();
       ASSERT_VALID(pDoc);

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

3 視窗的誕生

上一章那個極爲簡單的 Hello 程式，主視窗採用 *CFrameWnd* 類別。本例是 MDI 風格，將採用 *CMDIFrameWnd* 類別。

建構 MDI 主視窗，有兩個步驟。第一個步驟是 *new* 一個 *CMDIFrameWnd* 物件，第二個步驟是呼叫其 *LoadFrame* 函式。此函式內容如下：

```
// in WNDFRM.CPP
BOOL CFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle,
                          CWnd* pParentWnd, CCreateContext* pContext)
{
    CString strFullString;
    if (strFullString.LoadString(nIDResource))
        AfxExtractSubString(m_strTitle, strFullString, 0);  // first sub-string

    if (!AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG))
        return FALSE;

    // attempt to create the window
    LPCTSTR lpszClass = GetIconWndClass(dwDefaultStyle, nIDResource);
    LPCTSTR lpszTitle = m_strTitle;
    if (!Create(lpszClass, lpszTitle, dwDefaultStyle, rectDefault,
                pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext))
    {
           return FALSE;   // will self destruct on failure normally
    }

    // save the default menu handle
    m_hMenuDefault = ::GetMenu(m_hWnd);

    // load accelerator resource
    LoadAccelTable(MAKEINTRESOURCE(nIDResource));

    if (pContext == NULL)   // send initial update
           SendMessageToDescendants(WM_INITIALUPDATE, 0, 0, TRUE, TRUE);

    return TRUE;
}
```



視窗產生之際會發出 *WM_CREATE* 訊息，因此 *CMainFrame*::*OnCreate* 會被執行起來，那裡將進行工具列和狀態列的建立工作（稍後描述）。*LoadFrame* 函式的參數（本例爲 *IDR_MAINFRAME*）用來設定視窗所使用的各種資源，你可以從前一頁的 *CFrameWnd*::*LoadFrame* 原始碼中清楚看出。這些同名的資源包括：

```
// defined in SCRIBBLE.RC

IDR_MAINFRAME ICON DISCARDABLE   "res\\Scribble.ico"
IDR_MAINFRAME MENU PRELOAD DISCARDABLE  { ... }
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE  { ... }
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   IDR_MAINFRAME  "Scribble Step0"  （這個字串將成爲主視窗的標題）
   ..
END
```

這種作法（使用 *LoadFrame* 函式）與第 6 章的作法（使用 *Create* 函式）不相同，請注意。

工具列和狀態列的誕生（Toolbar & Status bar）

工具列和狀態列分別由 *CToolBar* 和 *CStatusBar* 掌管。兩個物件隸屬於主視窗，所以我們在 *CMainFrame* 中以兩個變數（事實上是兩個物件）表示之：

```
class CMainFrame : public CMDIFrameWnd
{
protected:  // control bar embedded members
      CStatusBar  m_wndStatusBar;
      CToolBar    m_wndToolBar;
...
};
```

主視窗產生之際立刻會發出 *WM_CREATE*，我們應該利用這時機把工具列和狀態列建立起來。爲了攔截 *WM_CREATE*，首先需在 Message Map 中設定「映射項目」：

```
BEGIN_MESSAGE_MAP(CMyMDIFrameWnd, CMDIFrameWnd)
      ON_WM_CREATE()
END_MESSAGE_MAP()
```

ON_WM_CREATE 這個巨集表示，只要 *WM_CREATE* 發生，我的 *OnCreate* 函式就應該被呼叫。下面是由 AppWizard 產生的 *OnCreate* 標準動作：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
         return -1;

   if (!m_wndToolBar.Create(this) ||
         !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
         TRACE0("Failed to create toolbar\n");
         return -1;     // fail to create
   }

   if (!m_wndStatusBar.Create(this) ||
         !m_wndStatusBar.SetIndicators(indicators,
           sizeof(indicators)/sizeof(UINT)))
   {
         TRACE0("Failed to create status bar\n");
         return -1;     // fail to create
   }
```

```
    // TODO: Remove this if you don't want tool tips or a resizeable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
          CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    return 0;
}
```

其中有四個動作與工具列和狀態列的產生及設定有關：

- m_wndToolBar.Create(this) 表示要產生一個隸屬於 *this*（也就是目前這個物件，也就是主視窗）的工具列。
- m_wndToolBar.LoadToolBar(IDR_MAINFRAME) 將 RC 檔中的工具列資源載入。*IDR_MAINFRAME* 在 RC 檔中代表兩種與工具列有關的資源：

```
IDR_MAINFRAME BITMAP MOVEABLE PURE "RES\\TOOLBAR.BMP"
```

```
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
```

LoadToolBar 函式一舉取代了前一版的 *LoadBitmap* + *SetButtons* 兩個動作。*LoadToolBar* 知道如何把 BITMAP 資源和 TOOLBAR 資源搭配起來，完成工具列的設定。當然啦，如果你不是使用 VC++資源工具來編輯工具列，BITMAP 資源和 TOOLBAR 資源就可能格數不符，那是不被允許的。TOOLBAR 資源中的各 ID 值就

是選單項目的子集合，因爲所謂工具列就是把比較常用的選單項目集合起來以按鈕方式提供給使用者。

- m_wndStatusBar.Create(this) 表示要產生一個隸屬於 *this* 物件（也就是目前這個物件，也就是主視窗）的狀態列。
- m_wndStatusBar.SetIndicators(,...) 的第一個參數是個陣列；第二個參數是陣列元素個數。所謂 Indicator 是狀態列最右側的「指示窗口」，用來表示大寫鍵、數字鍵等的 On/Off 狀態。AFXRES.H 中定義有七種 indicators：

```
#define ID_INDICATOR_EXT   0xE700  // extended selection indicator
#define ID_INDICATOR_CAPS  0xE701  // cap lock indicator
#define ID_INDICATOR_NUM   0xE702  // num lock indicator
#define ID_INDICATOR_SCRL  0xE703  // scroll lock indicator
#define ID_INDICATOR_OVR   0xE704  // overtype mode indicator
#define ID_INDICATOR_REC   0xE705  // record mode indicator
#define ID_INDICATOR_KANA  0xE706  // kana lock indicator
```

本例使用其中三種：

```
static UINT indicators[] =
{
        ID_SEPARATOR, // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};
```

滑鼠拖放（Drag and Drop）

MFC 程式很容易擁有 Drag and Drop 功能。意思是，你可以從 Shell（例如 Windows 95 的檔案總管）中以滑鼠拉動一個檔案，拖到你的程式中，你的程式因而打開此檔案並讀其內容，將內容放到一個 Document Frame 視窗中。甚至，使用者在 Shell 中以滑鼠對某個文件檔（你的應用程式的文件檔）快按兩下，也能啓動你這個程式，並自動完成開檔，讀檔，顯示等動作。

在 SDK 程式中要做到 Drag and Drop，並不算太難，這裡簡單提一下它的原理以及作法。當使用者從 Shell 中拖放一個檔案到程式 A，Shell 就配置一塊全域記憶體，填入被拖曳的檔案名稱（包含路徑），然後發出 *WM_DROPFILES* 傳到程式 A 的訊息佇列。程式 A 取得此訊息後，應該把記憶體的內容取出，再想辦法開檔讀檔。

> 並不是張三和李四都可以收到 *WM_DROPFILES*，只有具備 *WS_EX_ACCEPTFILES* 風格的視窗才能收到此一訊息。欲讓視窗具備此一風格，必須使用 *CreateWindowEx*（而不是傳統的 *CreateWindow*），並指定第一個參數爲 *WS_EX_ACCEPTFILES*。

剩下的事情就簡單了：想辦法把記憶體中的檔名和其他資訊取出（記憶體 handle 放在 *WM_DROPFILES* 訊息的 *wParam* 中）。這件事情有 *DragQueryFile* 和 *DragQueryPoint* 兩個 API 函式可以幫助我們完成。

SDK 的方法真的不難，但是 MFC 程式更簡單：

```
BOOL CScribbleApp::InitInstance()
{
    ...
    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    ...
}
```

這三個函式的用途如下：

- **CWnd::DragAcceptFile(BOOL bAccept=TRUE);** 參數 *TRUE* 表示你的主視窗以及每一個子視窗（文件視窗）都願意接受來自 Shell 的拖放檔案。*CFrameWnd* 內有一個 *OnDropFiles* 成員函式，負責對 *WM_DROPFIELS* 訊息做出反應，它會通知 application 物件的 *OnOpenDocument*（此函式將在第 8 章介紹），並夾帶被拖放的檔案的名稱。

- **CWinApp::EnableShellOpen();** 當使用者在 Shell 中對著本程式的文件檔快按兩下時，本程式能夠打開檔案並讀內容。如果當時本程式已執行，Framework 不會再執行起程式的另一副本，而只是以 DDE（Dynamic Data Exchange，動態資料交換）通知程式把檔案（文件）讀進來。DDE 處理常式內建在 *CDocManager* 之中（第 8 章會談到這個類別）。也由於 DDE 的能力，你才能夠很方便地把文件圖示拖放到印表機圖示上，將文件列印出來。

  通常此函式後面跟隨著 *RegisterShellFileTypes*。

- **CWinApp::RegisterShellFileTypes();** 此函式將向 Shell 註冊本程式的文件型態。有了這樣的註冊動作，使用者在 Shell 的雙擊動作才有著力點。這個函式搜尋 Document Template 串列中的每一種文件型態，然後把它加到系統所維護的 registry（登錄資料庫）中。

  在傳統的 Windows 程式中，對 Registry 的註冊動作不外乎兩種作法，一是準備一個 .reg 檔，由使用者利用 Windows 提供的一個小工具 regedit.exe，將 .reg 合併到系統的 Registry 中。第二種方法是利用 ::*RegCreateKey*、::*RegSetValue* 等 Win32 函式，直接編輯 Registry。MFC 程式的作法最簡單，只要呼叫 *CWinApp*::*RegisterShellFileTypes* 即可。

必須注意的是，如果某一種文件型態已經有其對應的應用程式（例如 .txt 對應 Notepad，.bmp 對應 PBrush，.ppt 對應 PowerPoint，.xls 對應 Excel），那麼你的程式就不能夠橫刀奪愛。如果本例 Scribble 的文件檔副檔名爲 .txt，使用者在 Shell 中雙擊這種檔案，啓動的將是 Notepad 而不是 Scribble。

另一個要注意的是，拖放動作可以把任何型態的文件檔拉到你的視窗中，並不只限於你所註冊的檔案型態。你可以把 .bmp 檔從 Shell 拉到 Scribble 視窗，Scribble 程式一樣會讀它並爲它準備一個視窗。想當然耳，那會是個無言的結局：

訊息映射（Message Map）

每一個衍生自 *CCmdTarget* 的類別都可以有自己的 Message Map 以處理訊息。首先你應該在類別宣告處加上 *DECLARE_MESSAGE_MAP* 巨集，然後在 .CPP 檔中使用 *BEGIN_MESSAGE_MAP* 和 *END_MESSAGE_MAP* 兩個巨集，巨集中間夾帶的就是「訊息與函式對映關係」的一筆筆記錄。

你可以從**圖 7-6** 那個濃縮的 Scribble 原始碼中看到各類別的 Message Map。

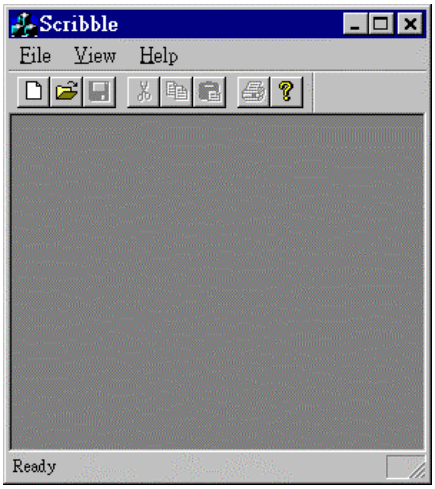
本例 *CScribbleApp* 類別接受四個 *WM_COMMAND* 訊息：

```
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()
```

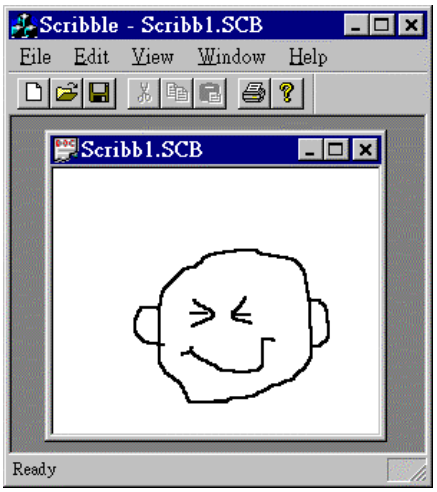
除了 *ID_APP_ABOUT* 是由我們自己設計一個 *OnAppAbout* 函式處理之，其他三個訊息都交給 *CWinApp* 成員函式去處理，因爲那些動作十分制式，沒什麼好改寫的。到底有哪些制式動作呢？看下一節！

標準選單 File / Edit / View / Window / Help

仔細觀察你所能蒐集到的各種 MDI 程式，你會發現它們幾乎都有兩組選單。一組是當沒有任何子視窗（文件視窗）存在時出現（本例代碼是 *IDR_MAINFRAME*）：



另一組則是當有任何子視窗（文件視窗）存在時出現（本例代碼是 *IDR_SCRIBTYPE*）：



前者多半只有【File】、【View】、【Help】等選項，後者就複雜了，程式所有的功能都在上面。本例的 *IDR_MAINFRAME* 和 *IDR_SCRIBTYPE* 就代表 RC 檔中的兩組選單。當使用者打開一份文件檔，程式應該把主視窗上的選單換掉，這個動作在 SDK 程式中由程式員負責，在 MFC 程式中則由 Framework 代勞了。

拉下這些選單仔細瞧瞧，你會發現 Framework 真的已經爲我們做了不少瑣事。凡是選單項目會引起對話盒的，像是 Open 對話盒、Save As 對話盒、Print 對話盒、Print Setup 對話盒、Find 對話盒、Replace 對話盒，都已經恭候差遣；Edit 選單上的每一項功能都已經可以應用在由 *CEditView* 掌控的文字編輯器上；File 選單最下方記錄著最近使用過的（所謂 LRU）四個檔案名稱（個數可在 Appwizard 中更改），以方便再開啓；View 選單允許你把工具列和狀態列設爲可見或隱藏；Window 選單提供重新排列子視窗圖示的能力，以及對子視窗的排列管理，包括卡片式（Cascade）或拼貼式（Tile）。

下表是預設之選單命令項及其處理常式的摘要整理。最後一個欄位「是否預有關聯」如果是 Yes，意指只要你的程式選單中有此命令項，當它被選按，自然就會引發命令處理常式，應用程式不需要在任何類別的 Message Map 中攔截此命令訊息。但如果是 No，表示你必須在應用程式中攔截此訊息。

選單內容	命令項 ID	預設的處理函式	預有關聯
File			
New	*ID_FILE_NEW*	*CWinApp::OnFileNew*	No
Open	*ID_FILE_OPEN*	*CWinApp::OnFileOpen*	No
Close	*ID_FILE_CLOSE*	*CDocument::OnFileClose*	Yes
Save	*ID_FILE_SAVE*	*CDocument::OnFileSave*	Yes
Save As	*ID_FILE_SAVEAS*	*CDocument::OnFileSaveAs*	Yes
Print	*ID_FILE_PRINT*	*CView::OnFilePrint*	No
Print Pre&view	*ID_FILE_PRINT_PREVIEW*	*CView::OnFilePrintPreview*	No
Print Setup	*ID_FILE_PRINT_SETUP*	*CWinApp::OnFilePrintSetup*	No
"Recent File Name"	*ID_FILE_MRU_FILE1~4*	*CWinApp::OnOpenRecentFile*	Yes

選單內容	命令項 ID	預設的處理函式	預有關聯
Exit	*ID_APP_EXIT*	*CWinApp::OnFileExit*	Yes
Edit			
Undo	*ID_EDIT_UNDO*	None	
Cut	*ID_EDIT_CUT*	None	
Copy	*ID_EDIT_COPY*	None	
Paste	*ID_EDIT_PASTE*	None	
View			
Toolbar	*ID_VIEW_TOOLBAR*	*FrameWnd::OnBarCheck*	Yes
Status Bar	*ID_VIEW_STATUS_BAR*	*FrameWnd::OnBarCheck*	Yes
Window（MDI only）			
New Window	*ID_WINDOW_NEW*	*MDIFrameWnd::OnWindowNew*	Yes
Cascade	*ID_WINDOW_CASCADE*	*MDIFrameWnd::OnWindowCmd*	Yes
Tile	*ID_WINDOW_TILE_HORZ*	*MDIFrameWnd::OnWindowCmd*	Yes
Arrange Icons	*ID_WINDOW_ARRANGE*	*MDIFrameWnd::OnWindowCmd*	Yes
Help			
About AppName	*ID_APP_ABOUT*	None	

上表的最後一欄位爲 No 者有五筆，表示雖然那些命令項有預設的處理常式，但你必須在自己的 Message Map 中設定映射項目，它們才會起作用。噢，AppWizard 此時又表現出了它的善體人意，自動爲我們做出了這些碼：

```
BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
        ...
        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()

BEGIN_MESSAGE_MAP(CScribbleView, CView)
        ...
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

對話盒

Scribble 可以啓動許多對話盒，前一節提了許多。唯一要程式員自己動手（我的意思是出現在我們的程式碼中）的只有 About 對話盒。

爲了攔截 *WM_COMMAND* 的 *ID_APP_ABOUT* 項目，首先我們必須設定其 Message Map：

```
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()
```

當訊息送來，就由 *OnAppAbout* 處理：

```
void CScribbleApp::OnAppAbout()
{
       CAboutDlg aboutDlg;
       aboutDlg.DoModal();
}
```

其中 *CAboutDlg* 是 *CDialog* 的衍生類別：

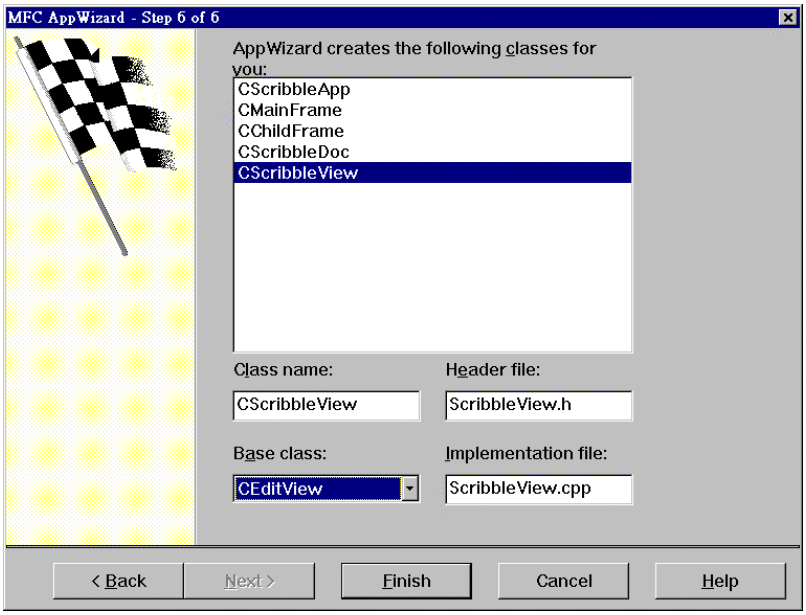
```
class CAboutDlg : public CDialog
{
    enum { IDD = IDD_ABOUTBOX };  // IDD_ABOUTBOX 是 RC 檔中的對話盒面板資源
    ...
    DECLARE_MESSAGE_MAP()
};
```

比之於 SDK 程式中的對話盒，這真是方便太多了。傳統 SDK 程式要在 RC 檔中定義對話盒面板（dialog template，也就是其外形），在 C 程式中設計對話盒函式。現在只需從 *CDialog* 衍生出一個類別，然後產生該類別之物件，並指定 RC 檔中的對話盒面板資源，再呼叫對話盒物件的 *DoModal* 成員函式即可。

第 10 章一整章將討論所謂的對話盒資料交換（DDX）與對話盒資料確認（DDV）。

改用 CEditView

Scribble step0 除了把一個應用程式的空殼做好，不能再貢獻些什麼。如果我們在 AppWizard 步驟六中把 *CScribbleView* 的基礎類別從 *CView* 改爲 *CEditView*，那可就有大妙用了：

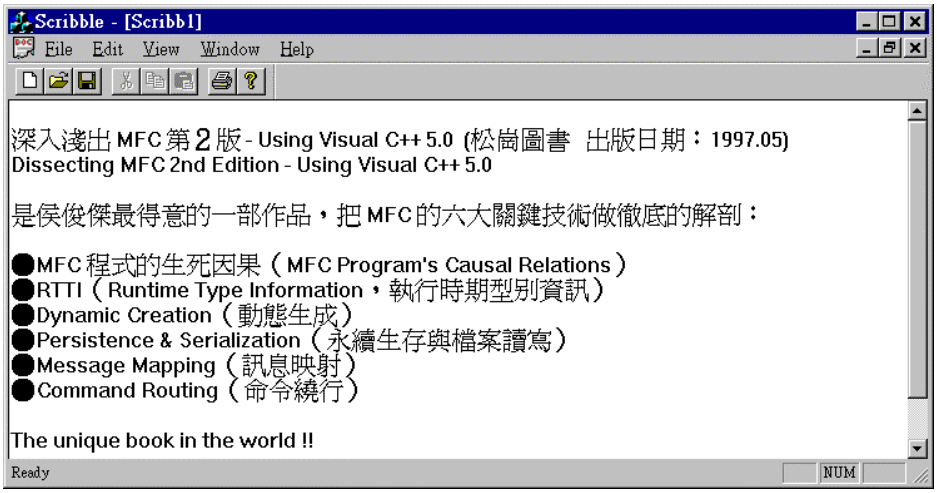


CEditView 是一個已具備文字編輯能力的類別，它所使用的視窗是 Windows 的標準控制元件之一 Edit，其 *SerializeRaw* 成員函式可以把 Edit 控制元件中的 raw text（而非「物件」所持有的資料）寫到檔案中。當我們在 AppWizard 步驟六選擇了它，程式碼中所有的 *CView* 統統變成 *CEditView*，而最重要的兩個虛擬函式則變成：

```
void CScribbleDoc::Serialize(CArchive& ar)
{
    // CEditView contains an edit control which handles all serialization
    ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}

void CScribbleView::OnDraw(CDC* pDC)
{
```

```
    CScribbleDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

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

就這樣，我們不費吹灰之力獲得了一個多視窗的文字編輯器：



並擁有讀寫檔能力以及預視能力：