
《深入淺出 MFC》2/e 電子書開放自由下載聲明

致親愛的大陸讀者

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

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

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

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

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

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

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

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

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

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

關於《深入淺出 MFC》2/e 電子書

《深入淺出 MFC》2/e 電子書共有五個檔案：

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

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

請告訴我您的資料

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

姓名：

現職：

畢業學校科系：

年齡：

性別：

居住省份（如是臺灣讀者，請寫縣市）：

對侯捷的建議：

-- the end

本系列丛书中的书籍

Visual C++ 5.0 入门与提高
Visual C++ 5.0 进阶与提高
Visual C++ 5.0 高级应用

Visual C++ 5.0 入门与提高
Visual C++ 5.0 进阶与提高
Visual C++ 5.0 高级应用

深入浅出

MFC 第2版

傅松源 著

Dissecting MFC and C++
using Visual C++ 5.0 & MFC 4.2

Copyright © 1998 by Tsinghua University Press. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, without permission in writing from Tsinghua University Press.

清华大学出版社

山高月小 水落石出

深入淺出 MFC

(第二版 使用 Visual C++ 5.0 & MFC 4.2)

Dissecting MFC

(Second Edition Using Visual C++ 5.0 & MFC 4.2)

侯俊傑 著

松崗電腦圖資料股份有限公司 印行

Pioneer is the one that an arrow on his back

讀者來函

■ 新竹市．高翠路．劉嘉均

1996 年 11 月，我在書店看到了 **深入浅出 MFC** 這本書，讓我想起自己曾經暗暗告訴過自己：Application Framework 真是一個好東西。我在書店駐足察看這本書五分鐘之後，我便知道這本書是一定要買下的。適巧我工作上的專案進度也到了一個即將完成的階段，所以我便一口氣將這本書給讀完了，而且是徹徹底底讀了兩遍。

我個人特別喜歡第 3 章：MFC 六大關鍵技術之模擬。這章內容的設計的確在 MFC 叢林中，大刀闊斧地披露出最重要的筋絡，我相信這正是所有學習 MFC 的人所需要的一種表明方式。對我而言，以往遺留的許多疑惑，在此都一一得到了解答。最重要的是，您曾經說過，學習 MFC 的過程中最重要的莫過於自我審視 MFC 程式碼的能力。很高興地，在我看完本書之後，我確實比以前更有能力來看 MFC 原始碼了。總之，我為自己能夠更深入了解 MFC 而要向您說聲謝謝。謝謝您為我們寫了 **深入浅出 MFC** 這本書。我受益匪淺。

■ 加拿大．溫哥華．陳宗泰

閣下之書，尚有人性，因此我參而再參，雖不悟，也是 enjoyable。看閣下之書的附帶效果是，重燃我追求知識的熱情。知也無涯，定慧談何容易。向閣下致敬：『Kruglinski 的 *Inside Visual C++* 和 Hou 的 *Dissecting MFC* 是通往 MFC Programming 的皇家大道』。

■ 香港 .lnlo@hkstart.com

我是你的一位讀者，住在香港。我剛買了你的翻譯的 *Inside Visual C++ 4.0*（中文版）。在此之前我買了你的另一本書 **深 入 淺 出 MFC**。在讀了 **深 入 淺 出 MFC** 前面 50~70 頁之後，我想我錯買了一本很艱深的書籍。我需要的是一本教我如何利用 MFC 來產生一個程式的書，而不是一本教我如何設計一套 MFC 的書。但是在我又讀了 30~40 頁之後，我想這本書真是棒，它告訴了我許多過去我不甚清楚的事情，像是 virtual function、template、exception...。

■ Lung Feng <h6003@kl.ntou.edu.tw>

我是您的忠實讀友，從您 1992 年出版的「Windows 程式設計」到現在的 **深 入 淺 出 MFC**，我已花不少銀子買下數本您的大作。雖然您的作品比其他國內出版的作品價格稍為高了一點，但我覺得很值得，因為我可以感受到您真的非常用心在撰寫，初閱讀您的作品時，有時不知其然，但只要用心品嚐，總是入味七分。有些書教人一邊看書一邊上機實作，會是一個比較好的學習曲線，但我是一個從基隆到台北通車的上班族，花很多時間在車上，在車上拜讀您的大作真是讓人愉快的事情（我回到家已晚，也只有車上能有時間充實自己）。這段時間內，我無法上機，卻能從中受益。而且一次再一次閱讀，常會有新的收穫，真如古人所說溫故而知新。

■ Asing Chang <asing@ms4.hinet.net>

今天抱著「無論如何一定要」的心情，把 *Dissecting MFC* 拿出來開封。序還看沒完就被深深地感動。這是一本太好的書，我想，我們是一群幸運的讀者。雖然我們沒有 Petzold 或 Pietrek，但是我們一樣能擁有最好的閱讀水準。

■ Jaguar <simon830@ms7.hinet.net>

有個問題想問您，為何在台灣要做基礎的事（R & D）總是很難如願。為何只有做近利的事才能被認可為成功之道。希望您能多出版一些像 **深入淺出 MFC** 的書，讓我們這些想要真正好書的人能大快朵頤一番。謝謝。

■ Shieh <lmy64621@mail.seeder.net.tw>

我是您忠實的讀者，您所寫的書一向是我在學習新的東西時很大的幫助。除此之外，我也十分喜歡看你為新書所寫的序，帶有哲學家的感覺！最近我想學 MFC，在市面上找到你的大作 **深入淺出 MFC**，拜讀後甚為興奮，也十分謝謝您寫書的用心。

■ 台北縣．土城．邱子文（資策會技研處）

深入淺出 MFC 是我所讀過之 MFC 書籍中最精采的一本。您大概不知道，我們人手一本 **深入淺出 MFC** 吧。**深入 Visual C++ 4.0** 的情況也差不了多少！

■ 台北市 yrleu@netrd.iii.org.tw

買了您的譯作 **深入 Visual C++ 4.0** 後，才認識您這在 Windows Programming 著作方面的頭號人物。十分佩服您的文筆，譯得通順流暢，可謂信達雅。之後偶然翻閱您另外的作品 **Windows 95 系統程式語言大叢書** 與 **深入淺出 MFC**，更是對您五體投地，立刻將這兩本書買下來，準備好好享受一下。對於 **深入淺出 MFC**，我給予極高的評價，因為它完全滿足我的需要。我去年才從台大電機博士班計算機科學組畢業，目前在資策會資訊技術處服國防役。先前作的純是理論研究，現在才開始接觸 Windows Programming。您的 **深入淺出 MFC** 對我而言是聖經。

真的很感謝您為知識傳授所作的努力！

■ 台中 Fox Wang

自從閱讀 **深入淺出 MFC** 之後，我便成了您的忠實讀者。我並不是一位專業程式設計師，而是一個對 Windows Programming 有濃厚興趣（和毅力）的大學生。在漫長而陡峭的學習曲線中，有幾本書對於我們而言就像茫茫大海中的一盞明燈，為我們指引明確的航道！我說的是 Charles Petzold 的 *Programming Windows 95*、David J.Kruglinski 的 *Inside Visual C++*，還有，當然，您的 **深入淺出 MFC**！

■ 印尼．雅加達 robin.hood@ibm.net

對您的書總是捧讀再三，即使翻爛了也值得。這本 **深入浅出 MFC**，不但具有學習價值，亦極具參考價值。

我買您的第一本書，好像是「記憶體管理與多工」。還記得當時熱中突破 640KB 記憶體，發現該書如獲至寶。數月前購買了 **深入浅出 MFC**，並利用閒暇時間翻閱學習（包括如廁時間...☺）。

我的學習曲線比較不同，我比較傾向於瞭解事情的因，而不是該如何做事情。比方說，「應該使用 MFC 的哪個類別」或「要改寫哪個虛擬函式」，對我而言還不如「CWinApp 何時何地呼叫了我的什麼函式」或「CDocManager 到底做了什麼」來得有趣（嗯，雖說是一樣重要啦）。這些「事情的因」在您的書中有大量詳細的介紹。

■ 新莊．輔大 skyman@tpts4.seed.net.tw

拜讀您的大作 **深入浅出 MFC** 令我感到無比興奮，對於您對電腦技術的專研如此深入，感到真是中國人之光。系上同學對於您的書籍愛恨交加，愛是如此清晰明瞭，恨是恨自己不成材呀！許多學長、同學、學弟都很喜愛您的作品，有些同學還拜您為偶像。因此想請您來演說，讓我們更深入認識程式語言的奧秘。大四學長知道要邀請您來，都非常高興，相當期待您的到來。

■ Rusty (楓橋驛站 CompBook 版)

深入浅出 MFC 我讀好幾遍了，講一句實在話，這本書給我的幫助真的很多！畢竟這樣深入挖 MFC 運作原理的書難找！要學 MFC 又沒有 Windows SDK 經驗者，建議跟 *Programming Windows 95 with MFC* 一起看，學起 MFC 會比較紮實。

若單純就「買了會不會後悔」來判斷一本書到底好不好，這本書我覺得物超所值！

■ 內壢．元智 Richard

剛才又把 **深入浅出 MFC** step0~step1 的程式看了一次，真的感觸良多。酒越陳越香，看老師您的書，真的是越看越「爽」，而且一定要晚上 10:00 以後看，哇，那種感覺真是過癮。

■ 桃園 Shelly

在書局看到您多本書籍，實在忍不住想告訴您我的想法！我是來謝謝您的。怎麼說呢？姑且不論英文能力，看原文書總是沒有看中文書來得直接啊！您也知曉的，許多翻譯書中的每個中文字都看得懂，但是整段落就是不知他到底在說啥！因此看到書的作者是您，感覺上就是一個品質上的保證，必定二話不說，抱回家囉！雖然眼前用不到，但是翻翻看，大致了解一下，待有空時或是工作上需要時再好好細讀。

網路書局的盛行，讓我也開始上網買些書。但是我只敢買像您的書！有品質嘛！其他的可就不敢直接買囉，總是必須到書局翻翻看，確定一下內容，才可能考慮。

■ 台北市 Jedi

Your books is already 100 times better than any translation on the market. I won't think of to get a Chinese computer book unless you wrote it or translated it.

■ shiowli@ms13.hinet.net

1997/11 月我看見了 **深入浅出 MFC**。仔細研讀後我知道這是我在 MFC 及 Windows 程式設計領域中的大衛之星。您的書一直都是我的良師，不但奠定了我的根基，也使我對 Windows 程式設計興趣大增。國內外介紹 MFC 程式設計的書很多，但看過範例後仍有一種被當成 puppet 的感覺。感謝侯先生毫不保留地攻堅 MFC，使我得到了豁然開朗的喜悅。

侯先生的文筆及胸襟令我佩服。有著 Charles Petzold 慈父般的講解，也有著 David J.Kruglinski 特有的風趣。您自述中說「先生不知何許人也」，嗯，我願意，我願意做一個默默祝福的人，好叫我在往後歲月裡能有更多喝采和大叫 eureka 的機會！

■ "anchor" <hcy89@mozart.ee.ncku.edu.tw>

I am a student of NCKU EE Department, I am also a reader of you books. Your Book give me a lot of help on my research.

■ 北投 z22356@ms13.hinet.net

選到這本書之前，我還在書架前猶豫：「又是一本厚厚大大卻一堆廢話的爛書，到底有沒有比較能讓我了解的書呀？」但是讀了您的 **深入浅出 MFC** 之後，把我 80% 的疑慮通通消除了。想永遠做您的讀者的讀者敬上。

■ David david@mail.u-p-c.com

我是您的讀者，雖然我尚未看完這本書，只看到第三篇的第 5 章，但我忍不住要把心得告訴您。去年我因為想寫 Windows 程式而買您的書，說老實話，我實在看不懂您所寫的文字（我真的懂 C++ 語法，也用 Visual Basic 寫過 Windows 程式），故放棄改買其他書籍來學習。雖把那些書全部看完了，也利用 MFC 來寫簡單程式，但心裡仍搞不懂這些程式的 How、What、Why。後來整理書架，發現這本書，就停下來拿來看，結果越看越搞懂這些 Windows 程式到底如何 How、What、Why。正如您的序所說「只用一樣東西，不明白它的道理，實在不高明」，感謝您能不計代價去做些不求近利的深耕工作，讓我們一群讀者能少走冤枉路。謝謝您!! 祝您身體健康 !!

■ Chengwei chengwei@accton.com.tw

最近正拜讀大作 **深入浅出 MFC**，明顯感受到作者寫書之負責用心（不論內容、排版設計、甚至用字遣詞乃至眾多的圖示）。雖然剛開始會覺得有些艱澀，但在反覆閱讀之後便能深深感受其中奧秘。謝謝你的努力！請再接再厲讓我們有更多好書可看！

■ Fox Wang <wych@ms10.hinet.net>

身為您的忠實讀者，總是希望能讓您聽到我們的聲音：記得您總是常在作品中強調讀者們寄給你的信對你具有很大的意義，但我要說，您可能不知道您帶給了某些讀者多大的正面意義！就我自己而言，從幾乎放棄對資訊科學的興趣，到留下來繼續努力，從排拒原文書到閱讀原文書成為習慣，從害怕閱讀原文期刊到慢慢發現閱讀它們的樂趣，然後，我打算往資訊方面的研究所前進。我想，不管將來我是否能將工作和興趣互相結合，您都豐富了我追求資訊科學這個興趣時的生活，非常感謝您！

當然，身為一位讀者，還是忍不住要自私地說，希望在很久很久以後，還可以看到您仍然在寫作！當然，身子也要好好照顧。

■ 上海 wuwei akira <hakira@hotmail.com>

I'm your reader in Shanghai JiaoTong University（按：上海交通大學）in mainland. Your <Programming WINDOWS MFC>（按：**深入浅出 MFC** 簡體版）is a very good book that I wanted to have for years. Thank you very much. So I want to know if there are another your book that I can buy in mainland? I hope to read your new books.

■ hon.bbs@bbs.ee.ncu.edu.tw

我非常喜歡你的書，不管是著作或是翻譯，給人的感覺真的是 "深入淺出"，我喜歡你用淺近的比喻說明，來解釋一些比較深入和抽象的東西，讀你的書，總讓我有突然 "頓悟" 的感覺，欣喜自己能在迷時找到良師。

■ 武漢 "wking" <wking@telekbird.com.cn>

Microsoft Developer Studio 與 MFC (Microsoft Foundation Classes) 相配合，構成了一個強大的利用 C++ 進行 32 位元 Windows 程序開發的工具，但是由于 MFC 系統相當龐大，內容繁雜，並且夾雜著大量令初學者莫明其妙的 macros，更加大了學習上的難度。

當今市面上有不少講解 C++ 和 VC++ 程序設計的書籍，但 C++ 書籍單純只講 C++，從 C++ 過渡到 VC++ 卻是初學者的一大難關；大多數講解 VC++ 的書都將重點放在如何使用 Microsoft Developer Studio，很少有對 MFC 進行深入而有系統的講解。而將 C++ 與 VC++ 相聯系，從 C++ 的角度來剖析 MFC 的運作，深入其設計原理與內部機制的書，更是鳳毛麟角。本人在市面上找了將近四個月，才發現這樣的一本，這就是由蜚聲海峽兩岸的著名電腦專家侯俊傑先生所著之《深入淺出 WINDOWS MFC 程序設計》（按：**深入淺出 MFC** 簡體版）。

本人在一月前購得此書，仔細研究月余，自我感覺比以前大有長進，其間由于印刷錯誤等原因，發現多處錯誤，于是向先生去信求教，得先生熱情支持和輔導。當先生得知本書（簡體本）未附光盤，且書中有多處誤印，深恐貽誤讀者，于是將原書光盤所附之源程序和執行文件 email 一份給我，囑我廣為散發，以惠大眾。

■ EricYang <robe@FreeBSD.csie.NCTU.edu.tw>

這真是本值得好好閱讀，好好保存的好書

■ cview.bbs@cis.nctu.edu.tw News / BBS 論壇 programming

深入淺出 MFC，侯 sir 自評為 MFC 四大天王之一，的確是傑作...

■ "lishyhan" <lishyhan@ms14.hinet.net>

我聽別人介紹，買了 **深入淺出 MFC** 第二版，的確是很適合我，之前買的書都太籠統了。

■ 美國 dengqi@glocom-us.com

侯俊傑先生：您好！從學校出來的七年間，我大多從事 embedded system software 的設計。在大陸，主要從事交換機系統軟件的設計，到了美國，主要從事衛星通信地面站系統軟件的設計。程序設計主要結合 C 和 Assembly。在大陸，embedded system 多採用 Intel 的 processor，在美國，embedded system 多採用 Motorola 的 processor。所以，我對 Intel 8086, 8051 系列及 Motorola 68000 系列的 assembly 語言比較熟悉，而對 framework 這樣的軟件製造思想和手段一直並不熟悉。近來偶有機會加入一個 project，要生成在 Win95 下運行的代碼，因此，想嘗試一下使用 framework 構造軟件。很幸運，我找到了您的書。講 VC++ MFC 的書很多，但能像您這樣做到「深入淺出」的，實在很少。看您的書，是享受。我手裡這本是簡體版，華中理工大學出版社出版。

News / BBS 論壇 (CompBook and/or programming)

☎ 請問，在 MFC 書籍之中，哪一本比較容易懂，因為我是初學 MFC，所以我需要的是比較基礎且容易瞭解的，想請大家推薦一下適合的書。

➤ ob9@bbs.ee.ntu.edu.tw：反正不管你是不是初學，只要你要繼續學，就應該看看 **深入淺出 MFC** 啦！

➤ os2.bbs@titan.cc.ntu.edu.tw：侯俊傑的書就對了!!

➤ openwin.bbs@cis.nctu.edu.tw：等你對 MFC 有一個程度的了解後再去看侯 sir 寫的 **深入淺出 MFC**... 保證讓你功力大增~~ ☺

➤ Rosario.bbs@bbs.ntu.edu.tw：**深入淺出 MFC** 這本比較好~~~不過之前最好買侯老師的 **型與虛擬**，把 C++ 弄清楚。最後看起 **深入 Visual C++** 就會吸收很快。

☎ 請問，想要從 DOS 跨足到 Windows 程式設計有哪些書值得推薦呢？

➤ hschin.bbs@bbs.cs.nthu.edu.tw：建議你看侯俊傑的 **深入淺出 MFC**，裡面除了對視窗程式的架構作基礎性的說明，讓你了解一些基礎概論，也說了不少視窗程式設計的課題，是非常不錯的一本書。

☎ 請問 VISUAL C++ 初學者適合的好書？

- wayne.bbs@bbs.ee.ncu：侯俊傑的 **深入 Visual C++** (*Inside visual C++* 中譯本) 不錯，適合初學者對 MFC 做初步的認識與應用。**深入淺出 MFC** 這一本原理講的較多。
- Sagitta.bbs@firebird.cs.ccu.edu.tw： *Inside Visual C++ 4.0* 不是初學者用的書，因為它未從最基本觀念講起。**深入淺出 MFC** 前半本都在描述(或說模擬) MFC 的內部技術，甚至挖出 MFC 部份原始程式碼來說明，透過這本書來學 MFC 會學得很紮實，不過自己要先對 Windows 這個作業系統的運作方式有一程度的瞭解，不然會看不懂，以某方面來說，也不是初學者用的書。基本上侯俊傑寫的書不論文筆或是內容都相當的好，相當有購買的價值，不過你別期望會是「初學用書」。

☎ 剛學 MFC 程式，是否可以推薦幾本你認為很好的工具書或者是參考書，原文的也沒關係，重要的是講的詳細。謝謝各位

- dickg.bbs@csie.nctu.edu.tw：我個人認為侯俊傑先生所著的 **深入淺出 MFC** 第二版不錯。這是一本受大眾推崇的好書，值得一再閱讀。但它的內容在某方面有些難度，so...需有耐心地一再翻閱，再輔以 on-line help 和其它 VC 書籍，如此定能收獲不少
- Rusty (Rusty)：我推薦 *Programming Windows 95 with MFC* (Jeff Prosise / Microsoft Press)。 *Inside Visual C++* 這本廣度夠，不過 MFC 初學者可能會看不懂；讀完了上一本之後再讀這本，你會活得快樂些 ☺。中文書嘛，大同小異的一大堆，不過侯俊傑的 **深入淺出 MFC** 非常獨特，值得一讀，很棒的一本書！

News / BBS 論壇 (CompBook and/or programming)

☎ 請推薦幾本 Visual C++ 的書

- kuhoung.bbs@csie.nctu.edu.tw : (1) *Inside Visual C++ 5.0* (2) *MFC Professional 5.0* (3) Mr. 侯俊傑 Any Books
- "howard" <lm3@ms22.hinet.net> : 先讀一點 SDK 著作,再讀 **深入淺出 MFC**,就夠了。剩下就多看 MSDN 吧。

☎ 我是一個剛學 VC 不久的人,想寫 Windows 程式,卻發現一大堆看不懂的函式或類別。查 help,都是英文,難懂其中意思。請問一下有沒有關於這方面的函式用法及意義的書籍呢? 有沒有這方面的初學書籍。我逛了幾間書店,是有買幾本 MFC 書籍,不過還是看不懂。

- "apexsoft" <lishyhan@ms14.hinet.net> : 如果要說書的話,侯俊傑先生翻譯的 **深入 Visual C++** 和他所寫的 **深入淺出 MFC** 兩本應該是夠用了。不然就再加上一本 SDK 書籍,這樣子應該是可以打個基礎了。
- CCA.bbs@cis.nctu.edu.tw : 函式名稱可以查 help,重要的是 C++ 的觀念。另外就是要了解 MFC 裡的 Document/View/Frame,以及 Dynamic Creation, Message mapping 等等。**深入淺出 MFC** 第二版對這些部份都有很深入的探討,把 MFC 裡的一些機制直接 trace code 加以說明。

☎ 我想請問以下巨集的意義及其使用時機和作用：DECLARE_DYNCREATE, DECLARE_DYNAMIC, IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP, END_MESSAGE_MAP。感激不盡，因為我常攪不清楚。

- titoni：可參考侯俊傑著的 **深入浅出 MFC 2/e** 第三章，第八章及第九章，書上的講解可以讓你有很大的收穫。

☎ 好像世界末日：最近買了 **深入浅出 MFC**。我一頁一頁仔細地閱讀。第一章...第二章...勉強有點概念，但是到了第三章，感覺好像世界末日了。MFC 六大技術的模擬...好像很難懂，讀起來非常吃力⊗ 是不是有其他書講得比較簡單的？我不是電腦科系學生，只是對電腦程式設計有興趣，一路由 basic -> FORTRAN -> C -> C++ 走來...

- szu.bbs@bbs.es.ncku.edu.tw：是的，第三章也許是世界末日，當初我看的時候也是跳過不看，不然就是看完 frame1 後就說再見了。但是你只要很努力地慢慢看，一步一步地看，你就會發現後面的章節是那麼清楚明瞭... 慢慢來吧，這第三章我也是看了三遍才弄懂了一次。我也非電腦科系學生，與你相同的路子走來，有點 SDK 概念和一點 Data structure 概念，對第三章會很容易懂的，加油。

- 軼名：我看第三章的時候也很辛苦，但懂了之後，後面的章節可是用飆的喔。

■ 武漢 bin_zhou

I am your reader of 《深入淺出 WINDOWS MFC 程序設計》（編按：深入淺出 MFC 簡體版）。I'm leaving in HUBEI__WUHAN（編按：湖北武漢）。Now, I have already get the book in HUA_ZHONG_LI_GONG_DA_XUE（編按：華中理工大學）。And I am interested in this book very much.

■ 屏東 a8756001@mail.npust.edu.tw

侯先生,您好，我是屏科大的學生，想要用 MFC 寫一個可以新增、修改、刪除資料等動作的程式，日前老師借了我您的書 深入淺出 MFC 第二版，我讀了很快樂，對於 Visual C++ 的 IDE 環境更為了解，對於 MFC 整個架構，有了比較明朗的感覺。

■ 大陸 Mike Dong <mikedong@online.sh.cn>

尊敬的侯俊傑先生：我叫董旬。我對 C/C++ 非常有興趣。暢讀了您寫的書《深入淺出 WINDOWS MFC 程序設計》（編按：深入淺出 MFC 簡體版），對我有非常大的幫助。在此，先感謝您。現在我感到對 C++ 語言本身和 MFC 框架十分了解，但在編程過程中仍然感到生疏，主要是函數的運用和函數的參數十分複雜。我對 WINDOWS SDK 編程較少，是否應該要熟悉 WINDOWS API 函數后，結合 MFC 框架編程？

侯俊傑回覆：的確如此。MFC 其實就是把 Windows API 做了一層薄薄包裝，包裝於各個設計良好的 classes 而已。所以，掌握了 MFC framework 架構組織之後，接下來在 programming 實務方面，就是去瞭解並運用各個 classes，而各個 classes 的 member functions 有許多就是 Windows APIs 的一對一化身。

■ 左營 luke@orchid.ee.ntu.edu.tw

侯先生你好：我現在是一名資訊預官，還在左營受訓。因為受訓的關係所以偶然間有機會讀到你寫的 **深入淺出 MFC** 第二版。本以為這麼大的一本書，一定很難 K，但從第一眼開始我就深深的被其中優雅且適當的文辭所吸引。尤其當閱讀第三章時，那些表格讓我回憶起以前修過 `advanced compiler` 去 `trace java compiler` 的那段過程，不禁發出會心一笑。

由於我本身學的是電機，所以不同於一般資訊人員所著重的應用層面。從大二時因為想充實自己的電腦實力，努力學寫程式開始，就在浩瀚的書海中發現你獨特的風格。尤其現今電書籍多是翻譯居多其中品質良莠不齊，你的作品尤其難能可貴。現今我仍然有時會去閱讀專業期刊或者雜誌，但礙於畢竟不是資訊教育訓練出身，有時會抓不住重點，甚者不求甚解。這是我覺得遺憾之處。但讀你的作品讓我在質量之間都獲得了相當的進步，且讀來相當輕鬆自然。你的序言中提到歡迎讀者的反應，這也是這封 `mail` 的動機。我想好的作家需要我們的鼓勵，當然也希望能從你處獲得更多的新知。謝謝。

■ 大陸 "BaiLu" <jinyang@public1.wx.js.cn>

侯先生：您好！以前我一直是用 `DELPHI` 和 `PB` 主要做數據機的，近日在看您編寫的《深入淺出 `WINDOWS MFC` 程序設計》（編按：**深入淺出 MFC** 簡體版），收益非淺，很佩服您的寫作水平，講得非常好。在大陸還是很少有您這般水準寫 `C++` 的書。在此表示感謝。

■ 北京 "Zhang Yongzhong" <yongzhongz@263.net>

尊敬的侯俊傑先生：您好！我是北京的一名計算機工作者，也是您的忠實讀者。有幸讀到您的一本非常優秀的著作《深入淺出 `WINDOWS MFC` 程序設計》，非常興奮，自感受益匪

淺，覺得是一本難得的好書。

深入淺出 MFC

二版五刷感言

我很開心地獲知，**深入淺出 MFC** 第二版即將進行第五刷。如果把第一版算進去，那就累積印製 9150 本了（不含簡體版）。也就是說，這本書擁有幾近一萬人（次）的讀者群（不含簡體版讀者）。

對一本如此高階又如此高價的技術書籍而言，誠不易也。我有許多感觸！

先從技術面談起。我閱讀過的 VC++ 或 MFC 書籍不算少，因此，我很有信心地說，這本書的內容有其獨步全球之處。本書企圖引領讀者進入 MFC 這個十分龐大並在軟體工具市場上極端重要之 application framework 的核心；我嘗試剖析其中美好的物件導向性質（註 1）的實作方式，亦嘗試剖析其中與 Windows 程式設計模型（註 2）息息相關之特殊性質（註 3）的實作方式。

註 1：此指 runtime type information、dynamic creation、persistence、document/view 之

註 2：此指 message based、event driven 之 programming model。

註 3：此指 message mapping、command routing 之

在技術層次上，唯 *MFC Internals* 堪與本書比擬（本書附錄A附有 *MFC Internals* 簡介）。

但是 *MFC Internals* 與 *Dissecting MFC*（本書之英文名稱）不僅在內容上各擅勝場，在訴求上亦截然不同。這本書並不是為精通 MFC programming 的老手而寫（雖然它通常亦帶給這樣的讀者不少幫助），而是為初窺 MFC programming 的新手所寫。*MFC Internals* 可以說是為技術而技術，探討深入，取材範圍極廣；*Dissecting MFC* 卻是為生活而技術，探討深入，但謹守主軸份際。所有我所鋪陳的核心層面的知識，都是為了建立起一份紮紮實實的 programming 基礎，讓你徹底瞭解 MFC 為你鋪陳的骨幹背後，隱藏了多少巧妙機關，做掉了多少煩瑣事務。

有了這份基礎，你才有輕鬆駕馭 MFC 的本錢。

唯有這份基礎，才能使你胸中自有丘壑。

如果夠用心，你還可以附帶地從本書概略學習到一個 application framework 的設計藍圖。雖然，99.99999% 的 programmer 終其一生不會設計一個 application framework，這樣的藍圖仍可以為你的物件導向觀念帶來許多面向的幫助。

我一直希望，能夠為此書發行英文國際版。囿於個人的語文能力以及時間，終未能行。但是看到來自世界各地的華人讀者的信函（加拿大、紐西蘭、越南、印尼、香港、中國大陸、美國...），也是另一種安慰。在 BBS 及 Internet News 看到各界對此書的評介，以及對此書內容的探討，亦讓我感到十分欣喜。

這本書（第二版）所使用的開發環境是 Visual C++ 5.0 & MFC 4.21。就在第五刷即將印行的今天，Visual C++ 6.0 也已問世；其中的 programming 關鍵，也就是 MFC，在主幹上沒有什麼變化，因此我不打算爲了 Visual C++ 6.0 而改版。

在此新刷中，我繼續修正了一些筆誤，並加上新的讀者來函。

未來，本書第三版，你會看到很大的變化。

侯俊傑 臺灣.新竹 1998.09.11

jjhou@ccca.nctu.edu.tw

FAX 886-3-5733976

第二版序

任何人看到前面的讀者來函，都會感動於一本電腦書籍的讀者與作者竟然能夠產生如此深厚的共鳴，以及似無若有的長期情感。

何況，身為這本書的作者的我！

我寫的是一本技術書籍，但是贏得未曾謀面的朋友們的信賴與感情。我知道，那是因為這本書裡面也有許多我自己的感情。每當收到讀友們針對這本書寄來的信件（紙張的或電子的），我總是懷著感恩的心情仔細閱讀。好幾位讀友，針對書中的可疑處或是可以更好的地方，不吝嗇地撥出時間，寫滿一大張一大張的信紙，一一向我指正。我們談的不只是技術，還包括用詞遣字的意境。新竹劉嘉均先生和加拿大陳宗泰先生給我非常寶貴的技術上的意見。陳先生甚至在一個月內來了五封航空信。

這些，怎不教我心懷感謝，並且更加戒慎恐懼！

感謝所有讀者促成這本書的更精緻化。Visual C++ 5.0 面世了，MFC 則停留在 4.2，程式設計的主軸沒有什麼大改變。對於新讀者，本書乃全新產品自不待言，您可以從目錄中細細琢磨所有的主題。對於老讀者，本書所帶給您的，是更精緻的製作，以及數章新增的內容（請看第 0 章「與前版本之差異」）。

最後，我要說，我知道，這本書真的帶給許多人很紮實的東西。而我所以願意不計代價去做些不求近利的深耕工作，除了這是身為專業作家的責任，以及個人的興趣之外，是的，我自己是工程師，我最清楚工程師在學習 MFC 時想知道什麼、在哪裡觸礁。

所有出自我筆下的東西，我自己受益最豐。

感謝你們。

侯俊傑 臺灣.新竹 1997.04.15

jjhou@ccca.nctu.edu.tw

FAX 886-3-5733976

第一版序

有一種軟體名曰 `version control`，用來記錄程式開發過程中的各種版本，以應不時之需，可以隨時反省、檢查、回覆過去努力的軌跡。

遺憾的是人的大腦沒有 `version control` 的能力。學習過程的徬徨猶豫、挫折困頓、在日積月累的漸悟或剎那之間的頓悟之後，彷彿都成了遙遠模糊的回憶；而屢起屢仆、大惑不解的地方，學成之後看起來則是那麼「理所當然」。

學習過往的艱辛，模糊而明亮，是學成冠冕上閃亮的寶石。過程愈艱辛，寶石愈璀璨。作為私人「想當年」的絕佳話題可矣，對於後學則無甚幫助。的確，誰會在一再跌倒的地方做上記號，永誌不忘？誰會把推敲再三的心得殷實詳盡地記錄下來，為後學鋪一條紅地毯？也許，沒有 `version control` 正是人類的本能，空出更多的腦力心力與精力，追求更新的事物。

但是，作為資訊教育體系一員的我，不能不有 `version control`。事實上我亦從來沒有忘記初學 MFC 的痛苦：C++ 語言本身的技術問題是其一，MFC 龐大類別庫的命名規則是其二，熟知的 Windows 程式基本動作統統不見了是其三，物件導向的觀念與 `application framework` 的包裝是其四。初學 MFC programming 時，我的腦袋猶如網目過大的篩子，什麼東西都留不住；各個類別及其代表意義，過眼即忘。

初初接觸 MFC 時，我對 Windows 作業系統以及 SDK 程式設計技術的掌握，實已處在眾人金字塔的頂端，困頓猶復如斯。實在是因爲，對傳統程式員而言，application framework 和 MFC 的運作機制太讓人陌生了。

目前市面上有不少講解 MFC 程式設計觀念的書籍，其中不乏很好的作品，包括 *Programming Windows 95 with MFC*（Jeff Prosise 著，Microsoft Press 出版），以及我曾經翻譯過的 *Inside Visual C++ 4.0*（David J. Kruglinski 著，Microsoft Press 出版）。**深入淺出 MFC** 的宗旨與以上二書，以及全世界所有的 MFC 或 Visual C++ 書籍，都不相同。全世界（呵，我的確敢這麼說）所有與 MFC 相關的書籍的重點，都放在如何使用各式各樣的 MFC 類別上，並供應各式各樣的應用實例，我卻意不在此。我希望提供的是對 MFC 應用程式基本架構的每一個技術環節的深入探討，其中牽扯到 MFC 本身的設計原理、物件導向的觀念、以及 C++ 語言的高級議題。有了基礎面的全盤掌握，各個 MFC 類別之使用對我們而言只不過是手冊查閱的功夫罷了。

本書書名已經自我說明了，這是一本既深又淺的書。深與淺是悖離的兩條射線，理不應同時存在。然而，沒有深入如何淺出？不入虎穴焉得虎子？

唯有把 MFC 骨幹程式的每一個基礎動作弄懂，甚至觀察其原始碼，才能實實在在掌握 MFC 這一套 application framework 的內涵，及其物件導向的精神。我向來服膺一句名言：原始碼說明一切，所以，我挖 MFC 原始碼給你看。

這是我所謂的深入。

唯有掌握住 MFC 的內涵，對於各式各樣的 MFC 應用才能夠如履平地，面對龐大的 application framework 也才能夠胸中自有丘壑。

這是我所謂的淺出。

本書分為四大篇。第一篇提出學習 MFC 程式設計之前的必要基礎，包括 Windows 程式的基本觀念以及 C++ 的高階議題。「學前基礎」是相當主觀的認定，不過，基於我個人的學習經驗以及教學經驗，我的挑選應該頗具說服力。第二篇介紹 Visual C++ 整合環境開發工具。本篇只不過是提綱挈領而已，並不企圖取代 Visual C++ 使用手冊。然而對於軟體使用的老手，此篇或已足以讓您掌握 Visual C++ 整合環境。工具的使用雖然談不上學問，但在視覺化軟體開發過程中扮演極重角色，切莫小覷它。

第三篇介紹 application framework 的觀念，以及 MFC 骨幹程式。所謂骨幹程式，是指 Visual C++ 的工具 AppWizard 所產生出來的程式碼。當然，AppWizard 會根據你的選項做出不同的程式碼，我所據以解說的，是大眾化選項下的產品。

第四篇以微軟公司附於 Visual C++ 光碟片上的一個範例程式 Scribble 為主軸，一步一步加上新的功能。並在其間深入介紹 Runtime Type Information (RTTI)、Dynamic Creation、Persistence (Serialization)、Message Mapping、Command Routing 等核心技術。這些技術正是其他書籍最缺乏的部份。此篇之最後數章則脫離 Scribble 程式，另成一格。

本書前身，1994/08 出版的 **Visual C++ 物件導向 MFC 程式設計 基礎篇**以及 1995/04 年出版的 **應用篇**，序言之中我曾經這麼說，全世界沒有任何書籍文章，能夠把 MFC 談得這麼深，又表現得這麼淺。這些話已有一半成為昨日黃花：Microsoft Systems Journal 1995/07 的一篇由 Paul Dilascia 所撰的文章 *Meandering Through the Maze of MFC Message and Command Routing*，以及 Addison Wesley 於 1996/06 出版的 *MFC Internals* 一書，也有了相當程度的核心涉獵，即連前面提及的 *Programming Windows 95 with MFC* 以及 *Inside Visual C++ 4.0* 兩本書，也都多多少少開始涉及 MFC 核心。我有一種「德不孤 必有鄰」的喜悅。

爲了維護本書更多的唯一性，也由於我自己又鑽研獲得了新的心得，本書增加了前版未有的 Runtime Type Information、Dynamic Creation 等主題，對於 Message Mapping 與 Command Routing 的討論也更詳細得多，填補了上一版的縫隙。更值得一提的是，我把這些在 MFC 中極神秘而又極重要的機制，以簡化到不能再簡化的方式，在 DOS 程式中模擬出來，並且補充一章專論 C++ 的高階技術。至此，整個 MFC 的基礎架構已經完全曝露在你的掌握之中，再沒有任何神秘咒語了。

本書從 MFC 的運用，鑽入 MFC 的內部運作，進而 application framework 的原理，再至物件導向的精神，然後回到 MFC 的運用。這會是一條迢迢遠路嗎？

似遠實近！

許多朋友曾經與我討論過，對於 MFC 這類 application framework，應該挖掘其內部機制到什麼程度？探究原始碼，豈不有違「黑盒子」初衷？但是，沒有辦法，他們也同意，不把那些奇奇怪怪的巨集和指令搞清楚，只能生產出玩具來。對付 MFC 內部機制，態度不必像對付 MFC 類別一樣；你只需好好走過那麼一回，有個印象，足矣。至於龐大繁複的整個 application framework 技術的鋪陳串接，不必人人都痛苦一次，我做這麼一次也就夠了☺。

林語堂先生在 **朱門** 一書中說過的一句話，適足作為我寫作本書的心境，同時也對我與朋友之間的討論做個總結：

「只用一樣東西，不明白它的道理，實在不高明」。

祝各位 胸中丘壑自成！

侯俊傑 新竹 1996.08.15

P.S. 愈來愈多的朋友在網路上與我打招呼，閒聊談心。有醫師、盲生、北京的作家、香港的讀者、從國中到研究所的各級學生。學生的科系範圍廣到令我驚訝，年齡的範圍也大到令我驚訝。對於深居簡出的作家而言，讀者群只是一個想像空間，哦，我真有這麼多讀者嗎?! 呵呵，喜歡這種感覺。回信雖然是一種壓力，不過這是個甜蜜的負擔。

你們常常感謝我帶給你們幫助。你們一定不知道，沒有你們細心研讀我的心血，並且熱心寫信給我，我無法忍受寫作的漫漫孤寂！我可以花三天的時間寫一篇序，也可以花一個上午設計一張圖。是的，我願意！我對擁有一群可愛可敬的讀者感到驕傲。

目 錄

(* 表示本版新增內容)

* 讀者來函	/ 1
* 第二版序	/ 5
第一版序	/ 7
目錄	/ 13
第 0 章 你一定要知道 (導讀)	/ 27
這本書適合誰	/ 27
你需要什麼技術基礎	/ 29
你需要什麼軟體環境	/ 29
讓我們使用同一種語言	/ 30
本書符號習慣	/ 34
磁片內容與安裝	/ 34
範例程式說明	/ 34
與前版本之差異	/ 39
如何聯絡作者	/ 40

第一篇 勿在浮砂築高臺 - 本書技術前提 / 001

第 1 章 Win32 程式基本觀念 / 003
Win32 程式開發流程 / 005
需要什麼函式庫 (.LIB) / 005
需要什麼表頭檔 (.H) / 006

以訊息為基礎，以事件驅動之	/ 007
一個具體而微的 Win32 程式	/ 009
程式進入點 WinMain	/ 015
視窗類別之註冊與視窗之誕生	/ 016
訊息迴路	/ 018
視窗的生命中樞 - 視窗函式	/ 019
訊息映射 (Message Map) 雛形	/ 020
對話盒的運作	/ 022
模組定義檔 (.DEF)	/ 024
資源描述檔 (.RC)	/ 024
Windows 程式的生與死	/ 025
閒置時間的處理：OnIdle	/ 027
* Console 程式	/ 028
* Console 程式與 DOS 程式的差別	/ 029
* Console 程式的編譯聯結	/ 031
* JBACKUP：Win32 Console 程式設計	/ 032
* MFCCON：MFC Console 程式設計	/ 035
* 什麼是 C Runtime Library 的多緒版本	/ 038
行程與執行緒 (Process and Thread)	/ 039
核心物件	/ 039
一個行程的誕生與死亡	/ 040
產生子行程	/ 041
一個執行緒的誕生與死亡	/ 044
* 以 _beginthreadex 取代 CreateThread	/ 046
執行緒優先權 (Priority)	/ 048
* 多緒程式設計實例	/ 050

第 2 章 C++ 的重要性質	/ 055
類別及其成員 - 談封裝 (encapsulation)	/ 056
基礎類別與衍生類別 - 談繼承 (Inheritance)	/ 057
this 指標	/ 061
虛擬函式與多型 (Polymorphism)	/ 062
類別與物件大解剖	/ 077
Object slicing 與虛擬函式	/ 082
靜態成員 (變數與函式)	/ 085
C++ 程式的生與死：兼談建構式與解構式	/ 088
* 四種不同的物件生存方式	/ 090
* 所謂 "Unwinding"	/ 092
執行時期型別資訊 (RTTI)	/ 092
動態生成 (Dynamic Creation)	/ 095
異常處理 (Exception Handling)	/ 096
Template	/ 100
Template Functions	/ 101
Template Classes	/ 104
Templates 的編譯與聯結	/ 106
第 3 章 MFC 六大關鍵技術之模擬	/ 109
MFC 類別階層	/ 111
Frame1 範例程式	/ 111
MFC 程式的初始化過程	/ 115
Frame2 範例程式	/ 118
RTTI (執行時期型別辨識)	/ 122
CRuntimeClass 與類別型錄網	/ 123
DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC 巨集	/ 125
Frame3 範例程式	/ 132

IsKindOf (型別辨識)	/ 140
Frame4 範例程式	/ 141
Dynamic Creation (動態生成)	/ 143
DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE 巨集	/ 144
Frame6 範例程式	/ 151
Persistence (永續生存) 機制	/ 160
Serialize (資料讀寫)	/ 161
DECLARE_SERIAL/IMPLEMENT_SERIAL 巨集	/ 167
沒有範例程式	/ 170
Message Mapping (訊息映射)	/ 170
Frame7 範例程式	/ 181
Command Routing (命令繞行)	/ 191
Frame8 範例程式	/ 203
* 本章回顧	/ 216

第二篇 欲善工事先利其器－ Visual C++ 5.0 開發工具 / 217

第 4 章 Visual C++ - 整合性軟體開發環境	/ 219
安裝與組成	/ 220
四個重要的工具	/ 234
內務府總管：Visual C++ 整合開發環境	/ 236
關於 project	/ 237
關於工具設定	/ 241
Source Browser	/ 243
Online Help	/ 247
除錯工具	/ 249
VC++ 除錯器	/ 251
Exception Handling	/ 255

程式碼產生器 - AppWizard	/ 257
東圈西點完成 MFC 程式骨幹	/ 258
Scribble Step0	/ 270
威力強大的資源編輯器	/ 294
Icon 編輯器	/ 295
Cursor 編輯器	/ 296
Bitmap 編輯器	/ 297
ToolBar 編輯器	/ 297
VERSIONINFO 資源編輯器	/ 299
String Table 編輯器	/ 300
Menu 編輯器	/ 301
Accelerator 編輯器	/ 303
Dialog 編輯器	/ 304
* Console 程式的專案管理	/ 305

第三篇 淺出 MFC 程式設計 / 309

第 5 章 總觀 Application Framework	/ 311
什麼是 Application Framework	/ 311
侯捷怎麼說	/ 312
我怎麼說	/ 314
別人怎麼說	/ 317
為什麼使用 Application Framework	/ 321
Microsoft Foundation Class (MFC)	/ 324
白頭宮女話天寶：Visual C++ 與 MFC	/ 327
縱覽 MFC	/ 329
General Purpose classes	/ 330
Windows API classes	/ 333

Application framework classes	/ 334
High level abstractions	/ 334
Afx 全域函式	/ 335
* MFC 巨集 (macros)	/ 335
* MFC 資料型態 (data type)	/ 338
第 6 章 MFC 程式設計導論 - MFC 程式的生死因果	/ 343
不二法門：熟記 MFC 類別的階層架構	/ 346
需要什麼函式庫 (.LIB)	/ 347
需要什麼含入檔 (.H)	/ 349
簡化的 MFC 程式架構 - 以 Hello MFC 為例	/ 351
Hello 程式原始碼	/ 352
MFC 程式的來龍去脈	/ 357
我只借用兩個類別：CWinApp 和 CFrameWnd	/ 358
CWinApp - 取代 WinMain 的地位	/ 359
CFrameWnd - 取代 WndProc 的地位	/ 362
引爆器 - Application object	/ 364
隱晦不明的 WinMain	/ 366
AfxWinInit - AFX 內部初始化動作	/ 370
CWinApp::InitApplication	/ 372
CMyWinApp::InitInstance	/ 374
CFrameWnd::Create 產生主視窗 (並註冊視窗類別)	/ 376
* 奇怪的視窗類別名稱 Afx:b:14ae:6:3e8f	/ 387
視窗顯示與更新	/ 389
CWinApp::Run - 程式生命的活水源頭	/ 390
把訊息與處理函式串接在一起：Message Map 機制	/ 394
來龍去脈總整理	/ 397
Callback 函式	/ 398

* 閒置時間 (idle time) 的處理：OnIdle	/ 403
Dialog 與 Control	/ 406
通用對話盒 (Common Controls)	/ 407
本章回顧	/ 409
第 7 章 簡單而完整：MFC 骨幹程式	/ 411
不二法門：熟記 MFC 類別的階層架構	/ 411
MFC 程式的 UI 新風貌	/ 412
Document/View 支撐你的應用程式	/ 419
利用 Visual C++ 工具完成 Scribble step0	/ 423
骨幹程式使用哪些 MFC 類別？	/ 423
Document Template 的意義	/ 430
Scribble 的 Document/View 設計	/ 436
主視窗的誕生	/ 438
工具列和狀態列的誕生 (Toolbar & Status bar)	/ 440
滑鼠拖放 (Drag and Drop)	/ 442
訊息映射 (Message Map)	/ 445
標準選單 File/Edit/View/Window/Help	/ 446
對話盒	/ 449
改用 CEditView	/ 450
第四篇 深入 MFC 程式設計	/ 453

第 8 章 Document-View 深入探討	/ 455
為什麼需要 Document-View (形而上)	/ 455
Document	/ 457
View	/ 458
Document Frame (View Frame)	/ 459

Document Template	/ 459
CDocTemplate 管理 CDocument / CView / CFrameWnd	/ 460
Scribble Step1 的 Document (資料結構設計)	/ 468
MFC Collection Classes 的選用	/ 469
Template-Based Classes	/ 471
Template-Based Classes 的使用方法	/ 471
CScribbleDoc 的修改	/ 473
SCRIBBLEDOC.H	/ 475
SCRIBBLEDOC.CPP	/ 477
文件：一連串的線條	/ 481
CScribbleDoc 的成員變數	/ 481
CObList	/ 481
CScribbleDoc 的成員函式	/ 482
線條與座標點	/ 484
CStroke 的成員變數	/ 484
CArray<CPoint, CPoint>	/ 484
CStroke 的成員函式	/ 484
Scribble Step1 的 View：資料重繪與編輯	/ 487
CScribbleView 的修改	/ 488
SCRIBBLEVIEW.H	/ 488
SCRIBBLEVIEW.CPP	/ 489
View 的重繪動作 - GetDocument 和 OnDraw	/ 493
CScribbleView 的成員變數	/ 493
CScribbleView 的成員函式	/ 493
View 與使用者的交談 (滑鼠訊息處理實例)	/ 495
ClassWizard 的輔佐	/ 496
WizardBar 的輔佐	/ 498
Serialize：物件的檔案讀寫	/ 498

Serialization 以外的檔案讀寫動作	/ 499
檯面上的 Serialize 動作	/ 501
檯面下的 Serialize 寫檔奧秘	/ 507
檯面下的 Serialize 讀檔奧秘	/ 514
DYNAMIC / DYNCREATE / SERIAL 三巨集	/ 522
Serializable 的必要條件	/ 527
CObject 類別	/ 529
IsKindOf	/ 529
IsSerializable	/ 530
CObject::Serialize	/ 531
CArchive 類別	/ 531
operator<< 和 operator>>	/ 532
效率考量	/ 536
自定 SERIAL 巨集給抽象類別使用	/ 537
在 CObList 中加入 CStroke 以外的類別	/ 537
Document 與 View 交流 - 爲 Scribble Step4 做準備	/ 543
第 9 章 訊息映射與命令繞行	/ 547
到底要解決什麼	/ 547
訊息分類	/ 549
萬流歸宗 Command Target (CCmdTarget)	/ 550
三個奇怪的巨集，一張巨大的網	/ 551
DECLARE_MESSAGE_MAP 巨集	/ 552
訊息映射網的形成：BEGIN_/ON_/END_ 巨集	/ 544
米諾托斯 (Minotauros) 與西修斯 (Theseus)	/ 560
兩萬五千里長征 - 訊息的流竄	/ 566
直線上溯 (一般 Windows 訊息)	/ 567
拐彎上溯 (WM_COMMAND 命令訊息)	/ 572

羅塞達碑石：AfxSig_xx 的秘密	/ 580
Scribble Step2：UI 物件的變化	/ 585
改變選單	/ 585
改變工具列	/ 588
利用 ClassWizard 連接命令項識別碼與命令處理函式	/ 590
維護 UI 物件狀態（UPDATE_COMMAND_UI）	/ 594
本章回顧	/ 599
 第 10 章 MFC 與對話盒	/ 601
對話盒編輯器	/ 602
利用 ClassWizard 連接對話盒與其專屬類別	/ 606
PNDLG.H	/ 610
PNDLG.CPP	/ 610
對話盒的訊息處理函式	/ 613
MFC 中各式各樣的 MAP	/ 615
對話盒資料交換與查核（DDX & DDV）	/ 617
MFC 中各式各樣的 DDx_ 函式	/ 621
如何喚起對話盒	/ 622
本章回顧	/ 625
 第 11 章 View 功能之加強與重繪效率之提昇	/ 627
同時修改多個 Views：UpdateAllViews 和 OnUpdate	/ 629
在 View 中定義一個 hint	/ 631
把 hint 傳給 OnUpdate	/ 635
利用 hint 增加重繪效率	/ 637
可捲動的視窗：CScrollView	/ 640
大視窗中的小窗口：Splitter	/ 650
分裂視窗的功能	/ 650

分裂視窗的程式概念	/ 651
分裂視窗之實作	/ 653
本章回顧	/ 657
第 12 章 印表與預覽	/ 659
概觀	/ 659
列印動作的背景原理	/ 663
MFC 預設的印表機制	/ 669
Scribble 列印機制的補強	/ 685
印表機的頁和文件的頁	/ 685
配置 GDI 繪圖工具	/ 687
尺寸與方向：關於映像模式（座標系統）	/ 688
分頁	/ 693
表頭（Header）與表尾（Footer）	/ 695
動態計算頁碼	/ 696
列印預覽（Print Preview）	/ 697
本章回顧	/ 698
第 13 章 多重文件與多重顯示	/ 701
MDI 和 SDI	/ 701
多重顯像（Multiple Views）	/ 703
視窗的動態分裂	/ 704
視窗的靜態分裂	/ 707
CreateStatic 和 CreateView	/ 709
視窗的靜態三叉分裂	/ 711
Graph 範例程式	/ 713
靜態分裂視窗之觀念整理	/ 724
同源子視窗	/ 725

CMDIFrameWnd::OnWindowNew	/ 726
Text 範例程式	/ 727
非制式作法的缺點	/ 734
多重文件	/ 736
新的 Document 類別	/ 736
新的 Document Template	/ 739
新的 UI 系統	/ 740
新文件的檔案讀寫動作	/ 742
* 第 14 章 MFC 多緒程式設計 (Multi-threaded Programming in MFC) / 745	
從作業系統層面看執行緒	/ 745
三個觀念：模組、行程、執行緒	/ 746
執行緒優先權 (Priority)	/ 748
執行緒排程 (Scheduling)	/ 751
Thread Context	/ 751
從程式設計層面看執行緒	/ 752
Worker Threads 和 UI Threads	/ 754
錯誤觀念	/ 754
正確態度	/ 755
MFC 多緒程式設計	/ 755
探索 CWinThread	/ 755
產生一個 Worker Thread	/ 759
產生一個 UI Thread	/ 761
執行緒的結束	/ 763
執行緒與同步控制	/ 763
MFC 多緒程式實例	/ 766

* 第 15 章	定製一個 AppWizard	/ 771
	到底 Wizard 是什麼？	/ 733
	Custom AppWizard 的基本操作	/ 774
	剖析 AppWizard Components	/ 779
	Dialog Templates 和 Dialog Classes	/ 780
	Macros	/ 781
	Directives	/ 783
	動手修改 Top Studio AppWizard	/ 784
	利用資源編輯器修改 IDD_CUSTOM1 對話窗畫面	/ 785
	利用 ClassWizard 修改 CCustom1Dlg 類別	/ 785
	改寫 OnDismiss 虛擬函式，在其中定義 macros	/ 787
	修改 text template	/ 788
	Top Studio AppWizard 執行結果	/ 789
	更多的資訊	/ 790
* 第 16 章	站上眾人的肩膀 - 使用 Components 和 ActiveX Controls	/ 791
	什麼是 Component Gallery	/ 792
	使用 Components	/ 795
	Splash screen	/ 795
	System Info for About Dlg	/ 797
	Tips of the Day	/ 798
	Components 實際運用：ComTest 程式	/ 799
	修改 ComTest 程式內容	/ 818
	使用 ActiveX Controls	/ 822
	ActiveX Control 基礎觀念：Properties、Methods、Events	/ 823
	ActiveX Controls 的五大使用步驟	/ 825
	使用 "Grid" ActiveX Control：OcxTest 程式	/ 827

第五篇 附錄 / 843

附錄 A	無責任書評：從搖籃到墳墓 - Windows 的完全學習	/ 845
	* 無責任書評：MFC 四大天王	/ 856
附錄 B	Scribble Step5 程式原始碼列表	/ 873
附錄 C	Visual C++ 5.0 MFC 範例程式總覽	/ 915
* 附錄 D	以 MFC 重建 Debug Window (DBWIN)	/ 921

第 0 頁

你－一定要知道（導讀）

這本書適合誰

深入浅出 MFC 是一本介紹 MFC（Microsoft Foundation Classes）程式設計技術的書籍。對於 Windows 應用軟體的開發感到興趣，並欲使用 Visual C++ 整合環境的視覺開發工具，以 MFC 為程式基礎的人，都可以從此書獲得最根本最重要的知識與實例。

如果你是一位對 Application Framework 和物件導向（Object Oriented）觀念感興趣的技術狂熱份子，想知道神秘的 Runtime Type Information、Dynamic Creation、Persistence、Message Mapping 以及 Command Routing 如何實作，本書能夠充分滿足你。事實上，依我之見，這些核心技術與徹底學會操控 MFC 乃同一件事情。

全書分為四篇：

第一篇【勿在浮砂築高台】提供進入 MFC 核心技術以及應用技術之前的所有技術基礎，包括：

- Win32 程式觀念：message based, event driven, multitasking, multithreading, console programming。
- C++ 重要技術：類別與物件、this 指標與繼承、靜態成員、虛擬函式與多型、

模板（template）類別、異常處理（exception handling）。

■ MFC 六大技術之簡化模擬（Console 程式）

第二篇【欲善工事先利其器】提供給對 Visual C++ 整合環境全然陌生的朋友一個導引。這一篇當然不能取代 *Visual C++ User's Guide* 的地位，但對整個軟體開發環境有全盤以及概觀性的介紹，可以讓初學者迅速了解手上掌握的工具，以及它們的主要功能。

第三篇【淺出 MFC 程式設計】介紹一個 MFC 程式的生死因果。已經有 MFC 程式經驗的朋友，不見得不會對本篇感到驚豔。根據我的了解，太多人使用 MFC 是「只知道這麼做，不知道為什麼」；本篇詳細解釋 MFC 程式之來龍去脈，為初入 MFC 領域的讀者奠定紮實的基礎。說不定本篇會讓你有醍醐灌頂之感。

第四篇【深入 MFC 程式設計】介紹各式各樣 MFC 技術。「只知其然 不知其所以然」的不良副作用，在程式設計的企圖進一步開展之後，愈來愈嚴重，最終會行不得也！那些最困擾我們的 MFC 巨集、MFC 常數定義，不得一窺堂奧的 MFC 黑箱作業，在本篇陸續曝光。本篇將使您高喊：Eureka！

阿基米德在洗澡時發現浮力原理，高興得來不及穿上褲子，跑到街上大喊：Eureka（我找到了）。

範例程式方面，第三章有數個 Console 程式（DOS-like 程式，在 Windows 系統的 DOS Box 中執行），模擬並簡化 Application Framework 六大核心技術。另外，全書以一個循序漸進的 Scribble 程式（Visual C++ 所附範例），從第七章開始，分章探討每一個 MFC 應用技術主題。第 13 章另有三個程式，示範 Multi-View 和 Multi-Document 的情況。14 章~16 章是第二版新增內容，主題分別是 MFC 多緒程式設計、Custom AppWizard、以及如何使用 Component Gallery 提供的 ActiveX controls 和 components。

你需要什麼技術基礎

從什麼技術層面切入 Windows 軟體開發領域？C/SDK？抑或 C++/MFC？這一直是個引起爭議的論題。就我個人觀點，C++/MFC 程式設計必須跨越四大技術障礙：

1. 物件導向觀念與 C++ 語言。
2. Windows 程式基本觀念（程式進入點、訊息流動、視窗函式、callback...）。
3. Microsoft Foundation Classes（MFC）本身。
4. Visual C++ 整合環境與各種開發工具（難度不高，但需熟練）。

換言之，如果你從未接觸 C++，千萬不要閱讀本書，那只會打擊你學習新技術的信心而已。如果已接觸過 C++ 但不十分熟悉，你可以一邊複習 C++ 一邊學習 MFC，這也是我所鼓勵的方式（很多人是爲了使用 MFC 而去學習 C++ 的）。C++ 語言的繼承（inheritance）特性對於我們使用 MFC 尤爲重要，因爲使用 MFC 就是要繼承各個類別並爲己用。所以，你應該對 C++ 的繼承特質（以及虛擬函式，當然）多加體會。我在第2章安排了一些 C++ 的必要基礎。我所挑選的題目都是本書會用到的技術，而其深度你不見得能夠在一般 C++ 書籍中發現。

如果你有 C++ 語言基礎，但從未接觸過 Win16 或 Win32 程式設計，只在 DOS 環境下開發過軟體，我在第1章爲你安排了一些 Win32 程式設計基礎。這個基礎至爲重要，只會在各個 Wizards 上按來按去，卻不懂所謂 message loop 與 window procedure 的人，不可能搞定 Windows 程式設計 -- 不管你用的是 MFC 或 OWL 或 Open Class Library，不管你用的是 Visual C++ 或 Borland C++ 或 VisualAge C++。

你需要什麼軟體硬體環境

一套 Windows 95（或 Windows NT）作業系統當然是必須的，中英文皆可。此外，你需要一套 Visual C++ 32 位元版。目前的最新版本是 Visual C++ 5.0，也是我使用的版本。

硬體方面，只要能跑上述兩種作業系統就算過關。記憶體（RAM）是影響運作速度的主因，多多益善。廠商宣稱 16MB RAM 是一個能夠使你工作舒適的數字，但我因此懷疑「舒適」這個字眼的定義。寫作本書時我的軟硬體環境是：

- Pentium 133
- 96M RAM
- 2GB 硬碟
- 17 吋顯示器。別以為顯示器和程式設計沒有關係。大尺寸螢幕使我們一次看多一點東西，不必在 Visual C++ 整合環境所提供的密密麻麻的畫面上捲來捲去。
- Windows 95（中文版）
- Visual C++ 5.0

讓我們使用另一種語言

要在電腦書籍不可或缺的英文術語與流利順暢的中文解說之間保持一個平衡，是多麼不容易的一件事。我曾經以為我通過了最大的考驗，但每次總有新鮮事兒發生。是該叫 `class` 好呢？還是叫「類別」好？該叫 `object` 好呢？還是叫「物件」好？`framework` 難道該譯為框架嗎？`Document` 譯為「文件」，可也，可 `View` 是什麼碗糕？我很傷腦筋耶。考慮了這本書的潛在讀者所具備的技術基礎與教育背景之後，原諒我，不喜歡在中文書中看到太多英文字的朋友，你終究還是在這本書上看到了不少原文名詞。只有幾已統一化、沒有異議、可以望文生義的中文名詞，我才使用。

雖然許多名詞已經耳熟能詳，我想我還是有必要把它們界定一下：

API - Application Programming Interface。系統開放出來，給程式員使用的介面，就是 API。一般人的觀念中 API 是指像 C 函式那樣的東西，不盡然！DOS 的中斷向量（`interrupt vector`）也可以說是一種 API，OLE Interface（以 C++ 類別的形式呈現）也可以說是一種 API。不是有人這麼說嗎：MFC 勢將成為 Windows 環境上標準的 C++ API（我個人認為這句話已成為事實）。

SDK - Software Development Kit，原指軟體開發工具。每一套環境都可能有自己的 SDK，例如 Phar Lap 的 386DOS Extender 也有自己的 SDK。在 Windows 這一領域，SDK 原是指 Microsoft 的軟體開發工具，但現在已經變成一個一般性名詞。凡以 Windows raw API 撰寫的程式我們通常也稱為 SDK 程式。也有人把 Windows API 稱為 SDK API。Borland 公司的 C++ 編譯器也支援相同的 SDK API（那當然，因為 Windows 只有一套）。本書如果出現「SDK 程式」這樣的名詞，指的就是以 Windows raw API 完成的程式。

MFC - Microsoft Foundation Classes 的縮寫，這是一個架構在 Windows API 之上的 C++ 類別庫（C++ Class Library），意圖使 Windows 程式設計過程更有效率，更符合物件導向的精神。MFC 在爭取成為「Windows 類別庫標準」的路上聲勢浩大。Symantec C++ 以及 WATCOM C/C++ 已向微軟取得授權，在它的軟體開發平台上供應 MFC。Borland C++ 也可以吃進 MFC 程式碼 -- 啊，OWL 的地位益形尷尬了。

OWL - Object Windows Library 的縮寫，這也是一個具備 Application Framework 架勢的 C++ 類別庫，附含在 Borland C++ 之中。

Application Framework - 在物件導向領域中，這是一個專有名詞。關於它的意義，本書第5章有不少介紹。基本上它可以說是一個更有凝聚力，關聯性更強的類別庫。並不是每一套 C++ 類別庫都有資格稱為 Application Framework，不過 MFC 和 OWL 都可入列，IBM 的 Open Class Library 也是。Application Framework 當然不一定得是 C++ 類別庫，Java 和 Delphi 應該也都稱得上。

為使全書文字流暢精簡，我用了一些縮寫字：

API - Application Programming Interface

DLL - Dynamic Link Library

GUI - Graphics User Interface

MDI - Multiple Document Interface

MFC - Microsoft Foundation Class

OLE - Object Linking & Embedded

OWL - Object Windows Library

SDK - Software Development Kit

SDI - Single Document Interface

UI - User Interface

WinApp : Windows Application

以下是本書使用之中英文名詞對照表：

control	控制元件，如 Edit、ListBox、Button...。
drag & drop	拖放（滑鼠左鍵按下，選中圖示後拖動，然後放開）
Icon	圖示（視窗縮小化後的小圖樣）
linked-list	串列
listbox	列示盒、列示清單
notification	通告訊息（發生於控制元件）
preemptive	強制性、先佔式、優先權式
process	行程（一個執行起來的程式）
queue	佇列
template	C++ 有所謂的 class template，一般譯為類別樣板； Windows 有所謂的 dialog template，我把它譯為對話盒面板； MFC 有所謂的 Document Template，我沒有譯它（其義請見第 7 章 和第 8 章）
window class	視窗類別（不是一種 C++ 類別）
window focus	視窗焦點（擁有焦點之視窗，將可以獲得鍵盤輸入）

類別	class
物件	object
建構式	constructor
解構式	destructor
運算子	operator
改寫	override
多載	overloading，亦有他書譯為「過荷」
封裝	Encapsulation
繼承	Inheritance
動態繫結	Dynamic Binding，亦即後期繫結（late binding）
虛擬函式	virtual function
多型	Polymorphism，亦有他書譯為「同名異式」
成員函式	member function
成員變數	data member，亦有他書譯為「資料成員」
基礎類別	Base Class，亦即父類別
衍生類別	Derived Class，亦即子類別

另有一些名詞很難說用什麼中文字眼才好。例如 "double click"，有時候我寫「雙擊」，有時候我寫「以滑鼠快按兩下」；而 "click"，我可能用「選按」「選擇」「以滑鼠按一下」等字眼，完全視上下文而定。雖沒有統一，但您在文字中一定會了解我的意思。我期盼寫出一本讀起來很順又絕對不會讓你誤解意思的中文電腦書。還有些名詞在某些場合使用中文而在某些場合使用原文，例如 Class（類別）和 Object（物件）和 Menu（選單），為的也是使上下文閱讀起來舒服一些。這些文字的使用都肇基於我個人對文字的認知以及習慣，如果與您的風格不符，深感抱歉。我已盡力在一個處處需要英文名詞的領域中寫一本儘可能閱讀順暢的中文技術書籍。

本書符號習慣

斜體字表示函式、常數、變數、語言保留字、巨集、識別碼等等，例如：

<i>CreateWindow</i>	這是 Win32 函式
<i>strtok</i>	這是 C Runtime 函式庫的函式
<i>WM_CREATE</i>	這是 Windows 訊息
<i>ID_FILE_OPEN</i>	這是資源識別碼 (ID)
<i>CDocument::Serialize</i>	這是 MFC 類別的成員函式
<i>m_pNewViewClass</i>	這是 MFC 類別的成員變數
<i>BEGIN_MESSAGE_MAP</i>	這是 MFC 巨集
<i>public</i>	這是 C++ 語言保留字

當我解釋程式操作步驟時，如果使用中括弧，例如【File/New】，表示選按 File 選單中的 New 命令項。或者用來表示一個對話窗，例如我寫：【New Project】對話窗。

磁片內容與安裝

本書光碟片內含書中所有的範例程式，包括原始碼與 EXE 檔。中介檔案（如 .OBJ 和 .RES 等）並未放入。所有程式都可以在 Visual C++ 5.0 整合環境中製作出來。安裝方式很簡單（根本沒有什麼安裝方式）：利用 DOS 外部指令，XCOPY，把整個光碟片拷貝到你的硬碟上即是了。

範例程式說明

- Generic（第 1 章）：這是一個 Win32 程式，主要用意在讓大家瞭解 Win32 程式的基本架構。
- Jbackup（第 1 章）：這是一個 Win32 console 程式，主要用意在讓大家瞭解

Visual C++ 整合環境中也可以做很單純的 DOS-like 程式，而且又能夠使用 Win32 API。

- MFCcon（第1章）：這是一個很簡單的 MFC console 程式，主要用意在讓大家瞭解 Visual C++ 整合環境中也可以做很單純的 DOS-like 程式，而且又能夠使用 MFC classes。
- MltiThrd（第1章）：這是一個 Win32 多緒程式，示範如何以 *CreateThread* 做出多個執行緒，並設定其虛懸狀態、優先權、重新啟動狀態、睡眠狀態。
- Frame1~8（第3章）：這些都是 console 程式（所謂 DOS-like 程式），模擬並簡化 Application Framework 的六大核心技術。只有 `IPersistence` 技術未模擬出來，因為那牽扯太廣。
 - Frame1：模擬 MFC 階層架構以及 application object
 - Frame2：模擬 MFC 的 *WinMain* 四大動作流程
 - Frame3：模擬 *CRuntimeClass* 以及 *DYNAMIC* 巨集，組織起所謂的類別型錄網
 - Frame4：模擬 *IsKindOf*（執行時期物件類別的鑑識能力，也就是所謂的 RTTI）
 - Frame5：模擬 Dynamic Creation（MFC 2.5 的作法）（在本新版中已拿掉）
 - Frame6：模擬 Dynamic Creation（MFC 4.x 的作法）
 - Frame7：模擬 Message Map
 - Frame8：模擬 Command Routing
- Hello 範例程式（第6章）：首先以最小量（兩個）MFC 類別，完成一個最簡單的 MFC 程式。沒有 Document/View -- 事實上這正是 MFC 1.0 版的應用程式風貌。本例除了提供你對 MFC 程式的第一印象，也對類別的靜態成員函式應用於 callback 函式做了一個示範。每有視窗異動（產生 *WM_PAINT*），就有一個 "Hello MFC" 字串從天而降。此外，也示範了閒置時間（idle time）的處理。

■ Scribble Step0~Step5：「Scribble」範例之於 MFC 程式設計，幾乎相當於「Generic」範例之於 SDK 程式設計。微軟的「官方手冊」*Visual C++ Class Library User's Guide* 全書即以本例為主軸，介紹這個可以讓你在視窗中以滑鼠左鍵繪圖的程式。Scribble 程式共有 Step1~Step7，七個階段的所有原始碼都可以在 Visual C++ 5.0 的 \DEVSTUDIO\VC\MFC\SAMPLES\SCRIBBLE 目錄中找到。本書只採用 Step1~Step5，並增列 Step0。Step6 是 OnLine Help 的製作，Step7 是 OLE Server 的製作，這兩個主題本書從缺。

■ Scribble Step0—由 MFC AppWizard 做出來的空殼程式，也就是所謂的 MFC 骨幹程式。完整原始碼列於第 4 章「東圈西點完成程式骨幹」一節。完整解說出現在第 7 章。

■ Scribble Step1—具備 Document/View 架構（第 8 章）：本例主旨在加上資料處理與顯示的能力。這一版的視窗沒有捲動能力。同一文件的兩個顯示視窗也沒有能夠做到即時更新的效果。當你在視窗甲改變文件內容，對映至同一文件的視窗乙並不會即時修正內容，必須等 *WM_PAINT* 產生（例如拉大視窗）。

這個版本已具備印表與預視能力，但並非「所見即所得」（What You See Is What You Get），印表結果明顯縮小，這是因為映像模式採用 *MM_TEXT*。15 吋監視器的 640 個圖素換到 300dpi 上才不過兩英吋多一點。

我們可以在這個版本中學習以 AppWizard 製作骨幹，並大量運用 ClassWizard 為我們增添訊息處理函式；也可以學習如何設計 Document，如何改寫 *CView::OnDraw* 和 *CDocument::Serialize*，這是兩個極端重要之虛擬函式。

■ Scribble Step2—修改使用者介面（第 9 章）：這個版本變化了選單，使程式多了筆寬設定功能。由於選單的變化，也帶動了工具列與狀態列的變化。

從這個版本中我們可以學習如何使用資源編輯器，製作各式各樣的程式資源。為了把選單命令處理函式放置在適當的類別之中，我們需要深入了解所謂的 Message

Mapping 和 Command Routing。

- **Scribble Step3**—增加「筆劃寬度對話盒」（第 10 章）：這個版本做出「畫筆寬度對話盒」，使用者可以在其中設定細筆寬度和粗筆寬度。預設的細筆為兩個圖素（pixel）寬，粗筆為五個圖素寬。

從這個版本中可以學習如何以對話盒編輯器設計對話盒面板，以 ClassWizard 增設對話盒處理函式，以及如何以 MFC 提供的 DDX/DDV 機制做出對話盒控制元件（control）的內容傳遞與內容查核。DDX（Dialog Data eXchange）的目的在簡化應用程式取得控制元件內容的過程，DDV（Dialog Data Validation）的目的則在加強應用程式對控制元件內容之數值合理化檢查。

- **Scribble Step4**—加強顯示能力 - 捲軸與分裂視窗（第 11 章）：Scribble 可以對同一份 Document 產生一個以上的 Views，但有一個缺點亟待克服，那就是你在視窗 A 的繪圖動作不能即時影響視窗 B -- 即使它們是同一份資料的一體兩面！

Step4 解決上述問題。主要關鍵在於我們必須想辦法通知所有同血源（同一份 Document）的兄弟（各個 Views），讓它們一起行動。但因此卻必須多考慮一個情況：當使用者的一個滑鼠動作可能引發許許多多程式繪圖動作時，繪圖效率就變得非常重要。因此在考量如何加強顯示能力時，我們就得設計所謂的「必要繪圖區」，也就是所謂的 Invalidate Region（不再適用的區域）。事實上每當使用者開始繪圖（增加新的線條），程式可以設定「必要繪圖區」為：該線條之最小外圍四方形。為了記錄這項資料，Step1 所設計並延續至今的 Document 資料結構必須改變。

Step1 的 View 視窗有一個缺點：沒有捲軸。新版本加上了垂直和水平捲軸，此外它也示範一種所謂的分裂視窗（Splitter）。

- **Scribble Step5**—印表與預視（第 12 章）：Step1 已有印表和預視能力，這當然歸功於 *CScribbleView::OnDraw*。現在要加強的是更細緻的印表能力，包括表

頭、表尾、頁碼、映像模式等等。座標系統（也就是映像模式，Mapping Mode）的選擇，關係到是否能夠「所見即所得」。爲了這個目的，必須使用能夠反應真實世界之尺寸（如英吋、公分）的映像模式，本例使用 MM_LOENGLISH，每個邏輯單位 0.01 英吋。

我們也在此版中學習如何設定文件的大小。有了大小，才能夠在列印時做分頁動作。

- **Graph** 範例程式(第 13 章)：這個程式示範如何在靜態分裂視窗的不同窗口中，以不同的方式（本例爲長條圖、點狀圖和文字形式）顯示同一份資料。
- **Text** 範例程式（第 13 章）：這個程式示範如何在同一份 Document 的各個「同源 view 視窗」中，以不同的顯示方法表現同一份資料，做到一體數面。
- **Graph2** 範例程式（第 13 章）：這個程式示範如何爲程式加上第二個 Document 型態。其間關係到新的 Document，新的 View，新的 UI。
- **MltiThrd** 範例程式（第 14 章）：這是第 1 章的同名程式的 MFC 版。我只示範 MFC 多緒程式的架構，原 Mltithrd 程式的繪圖部份留給讀者練習。
- **Top** 範例程式（第 15 章）：示範如何量身定做一個屬於自己的 AppWizard。我的這個 Top Studio AppWizard 架在系統的 MFC AppWizard 之上，增加一個開發步驟，詢問程式員名稱及其簡單聲明，然後就會在每一個產生出來的原始碼檔案最前端加上一段固定格式的說明文字。
- **ComTest** 範例程式（第 16 章）：此程式示範使用 Component Gallery 中的三個 components：**Splash Screen**、**SysInfo**、**Tip Of The Day**。
- **OcxTest** 範例程式（第 16 章）：此程式示範使用 Component Gallery 中的 **Grid** ActiveX control。

與前版本之差異

深入浅出 MFC 第二版與前一版本之重大差異在於：

1. 軟體工具由 Visual C++ 4.0 改為 Visual C++ 5.0，影響所及，第4章「Visual C++ - 整合性軟體開發環境」之內容改變極大。全書之中有關於 MFC 內部動作邏輯及其原始碼的變動不多，因為 Visual C++ 5.0 中的 MFC 版本還維持在 4.2。
2. 第1章增加 Console 程式設計，以及 Win32 多緒程式實例 Mltithrd。
3. 第2章增加「四種不同的物件生存方式」一節。
4. 第3章去除原有之 Frame5 程式（該程式以 MFC 2.5 的技術模擬 Dynamic Creation）。
5. 第4章全部改為 Visual C++ 5.0 使用畫面，並在最後增加一節「Console 程式的專案管理」。
6. 第6章增加「奇怪的視窗類別名稱 Afx:x:y:z:w」一節，以及增加 Hello 程式對 idle time 的處理。
7. 增加 14~16 三章。
8. 附錄A增加 <無責任書評/侯捷先生> 的「MFC 四大天王」一文。
9. 附錄D由原先之「OWL 程式設計一覽」，改為「以 MFC 重建 DBWIN」。

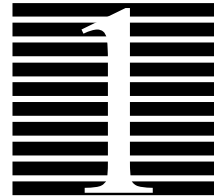
本書第一版之 Scribble 程式自 step1（加了 *CStroke*）之後，即無法在 Visual C++ 4.2 和 Visual C++ 5.0 上順利編譯。原因出在 VC++ 4.2 和 VC++ 5.0 似乎未能支援 "forward declaration of data structure class"（但是我懷疑 VC++ 怎麼會走退步？是不是有什麼選項可以設定）。無論如何，只要將 *CStroke* 的宣告搬移到 SCRIBBLEDOC.H 的最前面，然後再接續 *CScribbleDoc* 的宣告，即可順利編譯。請閱讀本書第8章「*CScribbleDoc* 的修改」一節之中於 SCRIBBLEDOC.H 原始碼列表後的一段說明（#477 頁）。

如何聯絡作者

我非常樂意和本書的所有讀者溝通，接受您對本書以及對我的指正和建議。請將溝通內容侷限在對書籍、對知識的看法，以及對本書誤謬之指正和建議上面，請勿要求我為您解決技術問題（例如您的程式臭蟲或您的專案瓶頸）。如果只是單純地想和我交個朋友聊聊天，我更倍感榮幸。

我的 Email 位址是 jjhou@ccca.nctu.edu.tw

我的永久通訊址是 新竹市建中一路 39 號 13 樓之二（FAX：03-5733976）



台島建築學在



第一篇 勿在浮砂築高台

Win32 基本程式觀念

程式設計領域裡，每一個人都想飛。
但是，還沒學會走之前，連跑都別想！

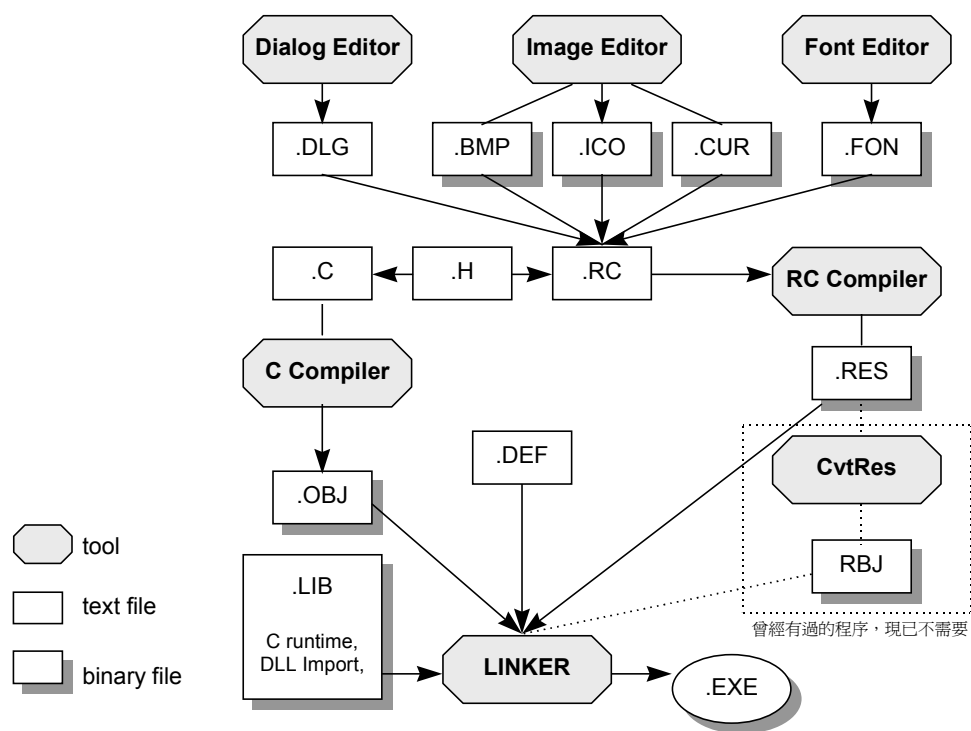
雖然這是一本深入講解 MFC 程式設計的書，我仍堅持要安排這第一章，介紹 Win32 的基本程式設計原理（也就是所謂的 SDK 程式設計原理）。

從來不曾學習過在「事件驅動(event driven)系統」中撰寫「以訊息為基礎(message based)之應用程式」者，能否一步跨入 MFC 領域，直接以 application framework 開發 Windows 程式，我一直抱持懷疑的態度。雖然有了 MFC（或任何其他的应用 framework），你可以繼承一整組類別，從而快速得到一個頗具規模的程式，但是 Windows 程式的運作本質（Message Based，Event Driven）從來不曾也不會改變。如果你不能了解其髓，空有其皮其肉或其骨，是不可能有所精進的，即使能夠操控 wizard，充其量卻也只是個 puppet，對於手上的程式碼，沒有自主權。

我認為學習 MFC 之前，必要的基礎是，對於 Windows 程式的事件驅動特性的了解（包括訊息的產生、獲得、分派、判斷、處理），以及對 C++ 多型（polymorphism）的精確體會。本章所提出的，是我對第一項必要基礎的探討，你可以從中獲得關於 Windows 程式的誕生與死亡，以及多工環境下程式之間共存的觀念。至於第二項基礎，將由第二章為你夯實。

讓我再強調一遍，本章就是我認為 Windows 程式設計者一定要知道的基礎知識。一個連這些基礎都不清楚的人，不能要求自己冒冒然就開始用 Visual C++、用 MFC、用物件導向的方式去設計一個你根本就不懂其運作原理的程式。

還沒學會走之前，不要跑！



Win32 程式開發流程

Windows 程式分為「程式碼」和「UI (User Interface) 資源」兩大部份，兩部份最後以 RC 編譯器整合為一個完整的 EXE 檔案（圖 1-1）。所謂 UI 資源是指功能選單、對話盒外貌、程式圖示、游標形狀等等東西。這些 UI 資源的實際內容（二進位碼）係借助各種工具產生，並以各種副檔名存在，如 .ico、.bmp、.cur 等等。程式員必須在一個所謂的資源描述檔 (.rc) 中描述它們。RC 編譯器 (RC.EXE) 讀取 RC 檔的描述後將所有 UI 資源檔集中製作出一個 .RES 檔，再與程式碼結合在一起，這才是一個完整的 Windows 可執行檔。

需要什麼函式庫 (.LIB)

眾所周知 Windows 支援動態連結。換句話說，應用程式所呼叫的 Windows API 函式是在「執行時期」才連結上的。那麼，「連結時期」所需的函式庫做什麼用？有哪些？

並不是延伸檔名為 .dll 者才是動態連結函式庫 (DLL, Dynamic Link Library)，事實上 .exe、.dll、.fon、.mod、.drv、.ocx 都是所謂的動態連結函式庫。

Windows 程式呼叫的函式可分為 C Runtimes 以及 Windows API 兩大部份。早期的 C Runtimes 並不支援動態連結，但 Visual C++ 4.0 之後已支援，並且在 32 位元作業系統中已不再有 small/medium/large 等記憶體模式之分。以下是它們的命名規則與使用時機：

- LIBC.LIB - 這是 C Runtime 函式庫的靜態連結版本。
- MSVCRT.LIB - 這是 C Runtime 函式庫動態連結版本 (MSVCRT40.DLL) 的 import 函式庫。如果連結此一函式庫，你的程式執行時必須有 MSVCRT40.DLL 在場。

另一組函式，Windows API，由作業系統本身（主要是 Windows 三大模組 GDI32.DLL 和 USER32.DLL 和 KERNEL32.DLL）提供（註）。雖說動態連結是在執行時期才發生「聯

結」事實，但在聯結時期，聯結器仍需先為呼叫者（應用程式本身）準備一些適當的資訊，才能夠在執行時期順利「跳」到 DLL 執行。如果該 API 所屬之函式庫尚未載入，系統也才因此知道要先行載入該函式庫。這些適當的資訊放在所謂的「import 函式庫」中。32 位元 Windows 的三大模組所對應的 import 函式庫分別為 GDI32.LIB 和 USER32.LIB 和 KERNEL32.LIB。

註：誰都知道，Windows 95 是 16/32 位元的混合體，所以旗下除了 32 位元的 GDI32.DLL、USER32.DLL 和 KERNEL32.DLL，又有 16 位元的 GDI.EXE、USER.EXE 和 KRNL386.EXE。32 位元和 16 位元兩組 DLLs 之間以所謂的 thunking layer 溝通。站在純粹 APIs 使用者的立場，目前我們不必太搭理這個事實。

Windows 發展至今，逐漸加上的一些新的 API 函式（例如 Common Dialog、ToolHelp）並不放在 GDI 和 USER 和 KERNEL 三大模組中，而是放在諸如 COMMDLG.DLL、TOOLHELP.DLL 之中。如果要使用這些 APIs，聯結時還得加上這些 DLLs 所對應的 import 函式庫，諸如 COMDLG32.LIB 和 TH32.LIB。

很快地，在稍後的範例程式「\$generic」的 makefile 中，你就可以清楚看到聯結時期所需的各式各樣函式庫（以及各種聯結器選項）。

需要什麼表頭檔（.H）

所有 Windows 程式都必須含入 WINDOWS.H。早期這是一個巨大的表頭檔，大約有 5000 行左右，Visual C++ 4.0 已把它切割為各個較小的檔案，但還以 WINDOWS.H 總括之。除非你十分清楚什麼 API 動作需要什麼表頭檔，否則為求便利，單單一個 WINDOWS.H 也就是了。

不過，WINDOWS.H 只照顧三大模組所提供的 API 函式，如果你用到其他 system DLLs，例如 COMMDLG.DLL 或 MAPI.DLL 或 TAPI.DLL 等等，就得含入對應的表頭檔，例如 COMMDLG.H 或 MAPI.H 或 TAPI.H 等等。

以訊息為基礎，以事件為驅動 (message based, event driven)

Windows 程式的進行係依靠外部發生的事件來驅動。換句話說，程式不斷等待（利用一個 `while` 迴路），等待任何可能的輸入，然後做判斷，然後再做適當的處理。上述的「輸入」是由作業系統捕捉到之後，以訊息形式（一種資料結構）進入程式之中。作業系統如何捕捉週邊設備（如鍵盤和滑鼠）所發生的事件呢？噢，USER 模組掌管各個週邊的驅動程式，它們各有偵測迴路。

如果把應用程式獲得的各種「輸入」分類，可以分為由硬體裝置所產生的訊息（如滑鼠移動或鍵盤被按下），放在系統佇列（`system queue`）中，以及由 Windows 系統或其它 Windows 程式傳送過來的訊息，放在程式佇列（`application queue`）中。以應用程式的眼光來看，訊息就是訊息，來自哪裡或放在哪裡其實並沒有太大區別，反正程式呼叫 `GetMessage` API 就取得一個訊息，程式的生命靠它來推動。所有的 GUI 系統，包括 UNIX 的 X Window 以及 OS/2 的 Presentation Manager，都像這樣，是以訊息為基礎的事件驅動系統。

可想而知，每一個 Windows 程式都應該有一個迴路如下：

```
MSG msg;
while (GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
// 以上出現的函式都是 Windows API 函式
```

訊息，也就是上面出現的 `MSG` 結構，其實是 Windows 內定的一種資料格式：

```
/* Queued message structure */
typedef struct tagMSG
{
    HWND      hwnd;
    UINT      message; // WM_xxx, 例如 WM_MOUSEMOVE, WM_SIZE...
    WPARAM    wParam;
    LPARAM    lParam;
    DWORD     time;
    POINT     pt;
} MSG;
```


接受並處理訊息的主角就是視窗。每一個視窗都應該有一個函式負責處理訊息，程式員必須負責設計這個所謂的「視窗函式」（window procedure，或稱為 window function）。如果視窗獲得一個訊息，這個視窗函式必須判斷訊息的類別，決定處理的方式。

以上就是 Windows 程式設計最重要的觀念。至於視窗的產生與顯示，十分簡單，有專門的 API 函式負責。稍後我們就會看到 Windows 程式如何把這訊息的取得、分派、處理動作表現出來。

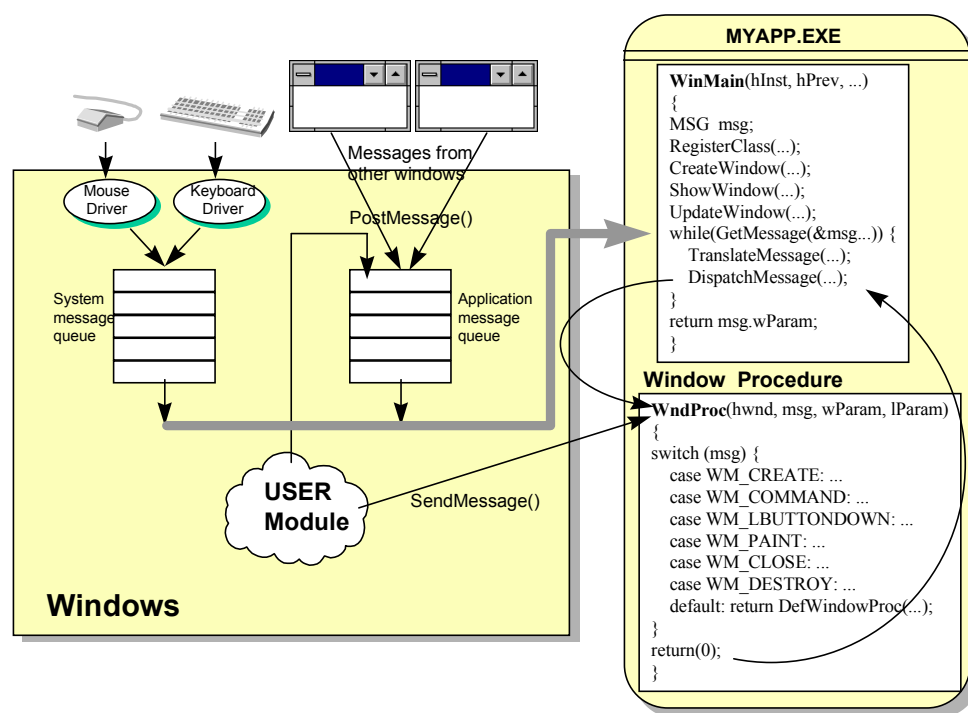


圖 1-2 Windows 程式的本體與作業系統之間的關係。

一個具體而微的 Win32 程式

許多相關書籍或文章嘗試以各種方式簡化 Windows 程式的第一步，因為單單一個 Hello 程式就要上百行，怕把大家嚇壞了。我卻寧願各位早一點接觸正統寫法，早一點看到全貌。Windows 的東西又多又雜，早一點一窺全貌是很有必要的。而且你會發現，經過有條理的解釋之後，程式碼的多寡其實構不成什麼威脅（否則無字天書最適合程式員閱讀）。再說，上百行程式碼哪算得了什麼！

你可以從圖 1-2 得窺 Win32 應用程式的本體與作業系統之間的關係。Win32 程式中最具代表意義的動作已經在該圖顯示出來，完整的程式碼展示於後。本章後續討論都圍繞著此一程式。

稍後會出現一個 makefile。關於 makefile 的語法，可能已經不再為大家所熟悉了。我想我有必要做個說明。

所謂 makefile，就是讓你能夠設定某個檔案和某個檔案相比 -- 比較其產生日期。由其比較結果來決定要不要做某些你所指定的動作。例如：

```
generic.res : generic.rc generic.h
    rc generic.rc
```

意思就是拿冒號(:)左邊的 generic.res 和冒號右邊的 generic.rc 和 generic.h 的檔案日期相比。只要右邊任一檔案比左邊的檔案更新，就執行下一行所指定的動作。這動作可以是任何命令列動作，本例為 rc generic.rc。

因此，我們就可以把不同檔案間的依存關係做一個整理，以 makefile 語法描述，以產生必要的編譯、聯結動作。makefile 必須以 NMAKE.EXE (Microsoft 工具) 或 MAKE.EXE (Borland 工具) 處理之，或其他編譯器套件所附的同等工具 (可能也叫做 MAKE.EXE) 處理之。

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

```
#0001 # filename : generic.mak
#0002 # make file for generic.exe (Generic Windows Application)
#0003 # usage : nmake generic.mak (Microsoft C/C++ 9.00) (Visual C++ 2.x)
#0004 # usage : nmake generic.mak (Microsoft C/C++ 10.00) (Visual C++ 4.0)
#0005
#0006 all: generic.exe
#0007
#0008 generic.res : generic.rc generic.h
#0009     rc generic.rc
#0010
#0011 generic.obj : generic.c generic.h
#0012     cl -c -W3 -Gz -D_X86_ -DWIN32 generic.c
#0013
#0014 generic.exe : generic.obj generic.res
#0015     link /MACHINE:I386 -subsystem:windows generic.res generic.obj \
#0016         libc.lib kernel32.lib user32.lib gdi32.lib
```

Generic.h

```
#0001 //-----
#0002 // 檔名 : generic.h
#0003 //-----
#0004 BOOL InitApplication(HANDLE);
#0005 BOOL InitInstance(HANDLE, int);
#0006 LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
#0007 LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
```

Generic.c (粗體代表 Windows API 函式或巨集)

```
#0001 //-----
#0002 //          Generic - Win32 程式的基礎寫法
#0003 //          Top Studio * J.J.Hou
#0004 // 檔名      : generic.c
#0005 // 作者      : 侯俊傑
#0006 // 編譯聯結 : 請參考 generic.mak
#0007 //-----
#0008
#0009 #include <windows.h> // 每一個 Windows 程式都需要含入此檔
#0010 #include "resource.h" // 內含各個 resource IDs
#0011 #include "generic.h" // 本程式之含入檔
#0012
#0013 HINSTANCE _hInst; // Instance handle
#0014 HWND      _hWnd;
```

```
#0015
#0016 char _szAppName[] = "Generic";    // 程式名稱
#0017 char _szTitle[]   = "Generic Sample Application"; // 視窗標題
#0018
#0019 //-----
#0020 // WinMain - 程式進入點
#0021 //-----
#0022 int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
#0023                     LPSTR lpCmdLine,    int nCmdShow)
#0024 {
#0025     MSG msg;
#0026
#0027     UNREFERENCED_PARAMETER(lpCmdLine);    // 避免編譯時的警告
#0028
#0029     if (!hPrevInstance)
#0030         if (!InitApplication(hInstance))
#0031             return (FALSE);
#0032
#0033     if (!InitInstance(hInstance, nCmdShow))
#0034         return (FALSE);
#0035
#0036     while (GetMessage(&msg, NULL, 0, 0)) {
#0037         TranslateMessage(&msg);
#0038         DispatchMessage(&msg);
#0039     }
#0040
#0041     return (msg.wParam); // 傳回 PostQuitMessage 的參數
#0042 }
#0043 //-----
#0044 // InitApplication - 註冊視窗類別
#0045 //-----
#0046 BOOL InitApplication(HINSTANCE hInstance)
#0047 {
#0048     WNDCLASS wc;
#0049
#0050     wc.style          = CS_HREDRAW | CS_VREDRAW;
#0051     wc.lpfnWndProc    = (WNDPROC)WndProc;    // 視窗函式
#0052     wc.cbClsExtra     = 0;
#0053     wc.cbWndExtra     = 0;
#0054     wc.hInstance      = hInstance;
#0055     wc.hIcon          = LoadIcon(hInstance, "jjhouricon");
#0056     wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
#0057     wc.hbrBackground  = GetStockObject(WHITE_BRUSH); // 視窗背景顏色
#0058     wc.lpszMenuName   = "GenericMenu";    // .RC 所定義的表單
#0059     wc.lpszClassName  = _szAppName;
#0060
```

```
#0061     return (RegisterClass(&wc));
#0062 }
#0063 //-----
#0064 // InitInstance - 產生視窗
#0065 //-----
#0066 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
#0067 {
#0068     _hInst = hInstance; // 儲存為全域變數，方便使用。
#0069
#0070     _hWnd = CreateWindow(
#0071         _szAppName,
#0072         _szTitle,
#0073         WS_OVERLAPPEDWINDOW,
#0074         CW_USEDEFAULT,
#0075         CW_USEDEFAULT,
#0076         CW_USEDEFAULT,
#0077         CW_USEDEFAULT,
#0078         NULL,
#0079         NULL,
#0080         hInstance,
#0081         NULL
#0082     );
#0083
#0084     if (!_hWnd)
#0085         return (FALSE);
#0086
#0087     ShowWindow(_hWnd, nCmdShow); // 顯示視窗
#0088     UpdateWindow(_hWnd);        // 送出 WM_PAINT
#0089     return (TRUE);
#0090 }
#0091 //-----
#0092 // WndProc - 視窗函式
#0093 //-----
#0094 LRESULT CALLBACK WndProc(HWND hWnd,     UINT message,
#0095                          WPARAM wParam, LPARAM lParam)
#0096 {
#0097     int wmId, wmEvent;
#0098
#0099     switch (message) {
#0100         case WM_COMMAND:
#0101
#0102             wmId    = LOWORD(wParam);
#0103             wmEvent  = HIWORD(wParam);
#0104
#0105             switch (wmId) {
#0106                 case IDM_ABOUT:
```

```

#0107         DialogBox(_hInst,
#0108             "AboutBox",      // 對話盒資源名稱
#0109             hWnd,            // 父視窗
#0110             (DLGPROC)About    // 對話盒函式名稱
#0111             );
#0112         break;
#0113
#0114         case IDM_EXIT:
#0115             // 使用者想結束程式。處理方式與 WM_CLOSE 相同。
#0116             DestroyWindow(hWnd);
#0117             break;
#0118
#0119         default:
#0120             return (DefWindowProc(hWnd, message, wParam, lParam));
#0121     }
#0122     break;
#0123
#0124     case WM_DESTROY: // 視窗已經被摧毀 (程式即將結束)。
#0125         PostQuitMessage(0);
#0126         break;
#0127
#0128     default:
#0129         return (DefWindowProc(hWnd, message, wParam, lParam));
#0130 }
#0131 return (0);
#0132 }
#0133 //-----
#0134 // About - 對話盒函式
#0135 //-----
#0136 LRESULT CALLBACK About(HWND hDlg,      UINT message,
#0137                         WPARAM wParam, LPARAM lParam)
#0138 {
#0139     UNREFERENCED_PARAMETER(lParam);    // 避免編譯時的警告
#0140
#0141     switch (message) {
#0142     case WM_INITDIALOG:
#0143         return (TRUE);    // TRUE 表示我已處理過這個訊息
#0144
#0145     case WM_COMMAND:
#0146         if (LOWORD(wParam) == IDOK
#0147             || LOWORD(wParam) == IDCANCEL) {
#0148             EndDialog(hDlg, TRUE);
#0149             return (TRUE); // TRUE 表示我已處理過這個訊息
#0150         }
#0151         break;
#0152     }

```

```
#0153     return (FALSE); // FALSE 表示我沒有處理這個訊息
#0154 }
```

Generic.rc

```
#0001 //-----
#0002 // 檔名 : generic.rc
#0003 //-----
#0004 #include "windows.h"
#0005 #include "resource.h"
#0006
#0007 jjhouricon ICON    DISCARDABLE    "jjhour.ico"
#0008
#0009 GenericMenu MENU DISCARDABLE
#0010 BEGIN
#0011     POPUP "&File"
#0012     BEGIN
#0013         MENUITEM "&New",            IDM_NEW, GRAYED
#0014         MENUITEM "&Open...",        IDM_OPEN, GRAYED
#0015         MENUITEM "&Save",            IDM_SAVE, GRAYED
#0016         MENUITEM "Save &As...",    IDM_SAVEAS, GRAYED
#0017         MENUITEM SEPARATOR
#0018         MENUITEM "&Print...",        IDM_PRINT, GRAYED
#0019         MENUITEM "P&rint Setup...", IDM_PRINTSETUP, GRAYED
#0020         MENUITEM SEPARATOR
#0021         MENUITEM "E&xit",            IDM_EXIT
#0022     END
#0023     POPUP "&Edit"
#0024     BEGIN
#0025         MENUITEM "&Undo\tCtrl+Z",    IDM_UNDO, GRAYED
#0026         MENUITEM SEPARATOR
#0027         MENUITEM "Cu&t\tCtrl+X",    IDM_CUT, GRAYED
#0028         MENUITEM "&Copy\tCtrl+C",    IDM_COPY, GRAYED
#0029         MENUITEM "&Paste\tCtrl+V",    IDM_PASTE, GRAYED
#0030         MENUITEM "Paste &Link",    IDM_LINK, GRAYED
#0031         MENUITEM SEPARATOR
#0032         MENUITEM "Lin&ks...",        IDM_LINKS, GRAYED
#0033     END
#0034     POPUP "&Help"
#0035     BEGIN
#0036         MENUITEM "&Contents",        IDM_HELPCONTENTS, GRAYED
#0037         MENUITEM "&Search for Help On...", IDM_HELPSEARCH, GRAYED
#0038         MENUITEM "&How to Use Help",    IDM_HELPHELP, GRAYED
#0039         MENUITEM SEPARATOR
#0040         MENUITEM "&About Generic...",    IDM_ABOUT
#0041     END
```

```
#0042 END
#0043
#0044 AboutBox DIALOG DISCARDABLE 22, 17, 144, 75
#0045 STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
#0046 CAPTION "About Generic"
#0047 BEGIN
#0048     CTEXT          "Windows 95",          -1,0, 5,144,8
#0049     CTEXT          "Generic Application",-1,0,14,144,8
#0050     CTEXT          "Version 1.0",          -1,0,34,144,8
#0051     DEFPUSHBUTTON  "OK",                  IDOK,53,59,32,14,WS_GROUP
#0052 END
```

程式進ㄧ點 WinMain

main 是一般 C 程式的進入點：

```
int main(int argc, char *argv[ ], char *envp[ ]);
{
...
}
```

WinMain 則是 Windows 程式的進入點：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
...
}
// 在 Win32 中 CALLBACK 被定義為 __stdcall，是一種函式呼叫習慣，關係到
// 參數擠壓到堆疊的次序，以及處理堆疊的責任歸屬。其他的函式呼叫習慣還有
// _pascal 和 _cdecl
```

當 Windows 的「外殼」（shell，例如 Windows 3.1 的程式管理員或 Windows 95 的檔案總管）偵測到使用者意欲執行一個 Windows 程式，於是呼叫載入器把該程式載入，然後呼叫 C startup code，後者再呼叫 *WinMain*，開始執行程式。*WinMain* 的四個參數由作業系統傳遞進來。

視窗類別之註冊與視窗之誕生

一開始，Windows 程式必須做些初始化工作，為的是產生應用程式的工作舞台：視窗。這沒有什麼困難，因為 API 函式 *CreateWindow* 完全包辦了整個巨大的工程。但是視窗產生之前，其屬性必須先設定好。所謂屬性包括視窗的「外貌」和「行為」，一個視窗的邊框、顏色、標題、位置等等就是其外貌，而視窗接收訊息後的反應就是其行為（具體地說就是指視窗函式本身）。程式必須在產生視窗之前先利用 API 函式 *RegisterClass* 設定屬性（我們稱此動作為註冊視窗類別）。*RegisterClass* 需要一個大型資料結構 *WNDCLASS* 做為參數，*CreateWindow* 則另需要 11 個參數。

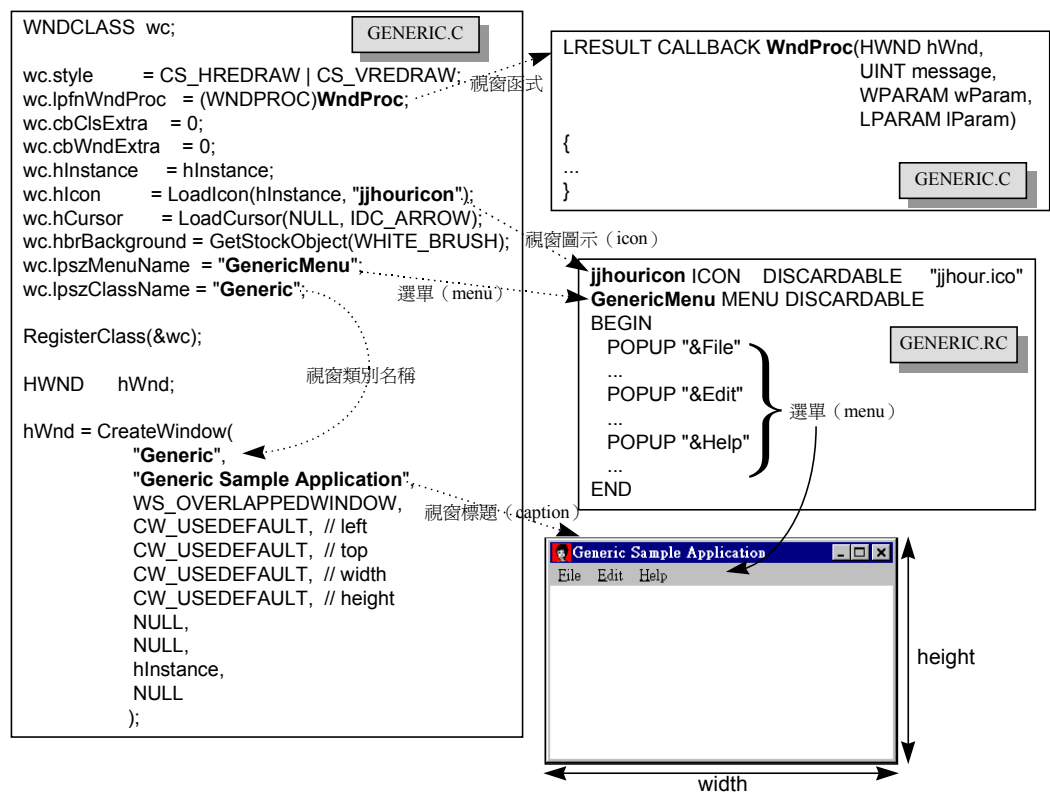


圖 1-3 RegisterClass 與 CreateWindow

從圖 1-3 可以清楚看出一個視窗類別牽扯的範圍多麼廣泛，其中 *wc.lpfnWndProc* 所指定的函式就是視窗的行為中樞，也就是所謂的視窗函式。注意，*CreateWindow* 只產生視窗，並不顯示視窗，所以稍後我們必須再利用 *ShowWindow* 將之顯示在螢幕上。又，我們希望先傳送個 *WM_PAINT* 給視窗，以驅動視窗的繪圖動作，所以呼叫 *UpdateWindow*。訊息傳遞的觀念暫且不表，稍後再提。

請注意，在 Generic 程式中，*RegisterClass* 被我包裝在 *InitApplication* 函式之中，*CreateWindow* 則被我包裝在 *InitInstance* 函式之中。這種安排雖非強制，卻很普遍：

```
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    if (!hPrevInstance)
        if (!InitApplication(hInstance))
            return (FALSE);

    if (!InitInstance(hInstance, nCmdShow))
        return (FALSE);

    ...
}
//-----
BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASS wc;
    ...
    return (RegisterClass(&wc));
}
//-----
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    _hWnd = CreateWindow(...);
    ...
}
```

兩個函式 (*InitApplication* 和 *InitInstance*) 的名稱別具意義：

- 在 Windows 3.x 時代，視窗類別只需註冊一次，即可供同一程式的後續每一個執行個體 (instance) 使用 (之所以能夠如此，是因為所有行程共在一個位址空間中)，所以我們把 *RegisterClass* 這個動作安排在「只有第一個執行個體才會

進入」的 *InitApplication* 函式中。至於此一行程是否是某個程式的第一個執行個體，可由 *WinMain* 的參數 *hPrevInstance* 判斷之；其值由系統傳入。

- 產生視窗，是每一個執行個體（instance）都得做的動作，所以我們把 *CreateWindow* 這個動作安排在「任何執行個體都會進入」的 *InitInstance* 函式中。

以上情況在 Windows NT 和 Windows 95 中略有變化。由於 Win32 程式的每一個執行個體（instance）有自己的位址空間，共用同一視窗類別已不可能。但是由於 Win32 系統令 *hPrevInstance* 永遠為 0，所以我們仍然得以把 *RegisterClass* 和 *CreateWindow* 按舊習慣安排。既符合了新環境的要求，又兼顧到了舊原始碼的相容。

InitApplication 和 *InitInstance* 只不過是兩個自定函式，為什麼我要對此振振有詞呢？原因是 MFC 把這兩個函式包裝成 *CWinApp* 的兩個虛擬成員函式。第 6 章「MFC 程式的生與死」對此有詳細解釋。

訊息迴路

初始化工作完成後，*WinMain* 進入所謂的訊息迴路：

```
while (GetMessage(&msg,...)) {  
    TranslateMessage(&msg); // 轉換鍵盤訊息  
    DispatchMessage(&msg); // 分派訊息  
}
```

其中的 *TranslateMessage* 是爲了將鍵盤訊息轉化，*DispatchMessage* 會將訊息傳給視窗函式去處理。沒有指定函式名稱，卻可以將訊息傳送過去，豈不是很玄？這是因爲訊息發生之時，作業系統已根據當時狀態，爲它標明了所屬視窗，而視窗所屬之視窗類別又已經明白標示了視窗函式（也就是 *wc.lpfnWndProc* 所指定的函式），所以 *DispatchMessage* 自有脈絡可尋。請注意圖 1-2 所示，*DispatchMessage* 經過 USER 模組的協助，才把訊息交到視窗函式手中。

訊息迴路中的 *GetMessage* 是 Windows 3.x 非強制性（non-preemptive）多工的關鍵。應用程式藉由此動作，提供了釋放控制權的機會：如果訊息佇列上沒有屬於我的訊息，我就把機會讓給別人。透過程式之間彼此協調讓步的方式，達到多工能力。Windows 95 和

Windows NT 具備強制性 (preemptive) 多工能力，不再非靠 *GetMessage* 釋放 CPU 控制權不可，但程式寫法依然不變，因為應用程式仍然需要靠訊息推動。它還是需要抓訊息！

視窗的生命週期：視窗函式

訊息迴路中的 *DispatchMessage* 把訊息分配到哪裡呢？它透過 USER 模組的協助，送到該視窗的視窗函式去了。視窗函式通常利用 *switch/case* 方式判斷訊息種類，以決定處置方式。由於它是由 Windows 系統所呼叫的（我們並沒有在應用程式任何地方呼叫此函式），所以這是一種 *call back* 函式，意思是指「在你的程式中，被 Windows 系統呼叫」的函式。這些函式雖然由你設計，但是永遠不會也不該被你呼叫，它們是為 Windows 系統準備的。

程式進行過程中，訊息由輸入裝置，經由訊息迴路的抓取，源源傳送給視窗並進而送到視窗函式去。視窗函式的體積可能很龐大，也可能很精簡，依該視窗感興趣的訊息數量多寡而定。至於視窗函式的型式，相當一致，必然是：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                           UINT message,
                           WPARAM wParam,
                           LPARAM lParam)
```

注意，不論什麼訊息，都必須被處理，所以 *switch/case* 指令中的 *default:* 處必須呼叫 *DefWindowProc*，這是 Windows 內部預設的訊息處理函式。

視窗函式的 *wParam* 和 *lParam* 的意義，因訊息之不同而異。*wParam* 在 16 位元環境中是 16 位元，在 32 位元環境中是 32 位元。因此，參數內容（格式）在不同作業環境中就有了變化。

我想很多人都會問這個問題：為什麼 Windows Programming Modal 要把視窗函式設計為一個 *call back* 函式？為什麼不讓程式在抓到訊息 (*GetMessage*) 之後直接呼叫它就好了？原因是，除了你需要呼叫它，有很多時候作業系統也要呼叫你的視窗函式（例如當

某個訊息產生或某個事件發生)。視窗函式設計為 `callback` 形式，才能開放出一個介面給作業系統叫用。

訊息映射 (Message Map) 的雛形

有沒有可能把視窗函式的內容設計得更模組化、更一般化些？下面是一種作法。請注意，以下作法是 MFC「訊息映射表格」(第 9 章)的雛形，我所採用的結構名稱和變數名稱，都與 MFC 相同，藉此讓你先有個暖身。

首先，定義一個 `MSGMAP_ENTRY` 結構和一個 `dim` 巨集：

```
struct MSGMAP_ENTRY {
    UINT nMessage;
    LONG (*pfn)(HWND, UINT, WPARAM, LPARAM);
};

#define dim(x) (sizeof(x) / sizeof(x[0]))
```

請注意 `MSGMAP_ENTRY` 的第二元素 `pfn` 是一個函式指標，我準備以此指標所指之函式處理 `nMessage` 訊息。這正是物件導向觀念中把「資料」和「處理資料的方法」封裝起來的一種具體實現，只不過我們用的不是 C++ 語言。

接下來，組織兩個陣列 `_messageEntries[]` 和 `_commandEntries[]`，把程式中欲處理的訊息以及訊息處理常式的關聯性建立起來：

```
// 訊息與處理常式之對照表格
struct MSGMAP_ENTRY _messageEntries[] =
{
    WM_CREATE,    OnCreate,
    WM_PAINT,     OnPaint,
    WM_SIZE,      OnSize,
    WM_COMMAND,   OnCommand,
    WM_SETFOCUS,  OnSetFocus,
    WM_CLOSE,     OnClose,
    WM_DESTROY,   OnDestroy,
};
```

↑	↑
這是訊息	這是訊息處理常式

```
// Command-ID 與處理常式之對照表格
struct MSGMAP_ENTRY _commandEntries =
{
    IDM_ABOUT,      OnAbout,
    IDM_FILEOPEN,   OnFileOpen,
    IDM_SAVEAS,     OnSaveAs,
};
```

↑
↑
 這是 WM_COMMAND 命令項 這是命令處理常式

於是視窗函式可以這麼設計：

```
//-----
// 視窗函式
//-----
LRESULT CALLBACK WndProc(HWND hWnd,      UINT message,
                          WPARAM wParam, LPARAM lParam)
{
    int i;

    for(i=0; i < dim(_messageEntries); i++) { // 訊息對照表
        if (message == _messageEntries[i].nMessage)
            return((*_messageEntries[i].pfn)(hWnd, message, wParam, lParam));
    }
    return(DefWindowProc(hWnd, message, wParam, lParam));
}
//-----
// OnCommand -- 專門處理 WM_COMMAND
//-----
LONG OnCommand(HWND hWnd, UINT message,
               WPARAM wParam, LPARAM lParam)
{
    int i;

    for(i=0; i < dim(_commandEntries); i++) { // 命令項目對照表
        if (LOWORD(wParam) == _commandEntries[i].nMessage)
            return((*_commandEntries[i].pfn)(hWnd, message, wParam, lParam));
    }
    return(DefWindowProc(hWnd, message, wParam, lParam));
}
//-----
LONG OnCreate(HWND hWnd, UINT wParam, LONG lParam)
{
    ...
}
```

```
//-----  
LONG OnAbout(HWND hWnd, UINT wParam, LONG lParam)  
{  
    ...  
}  
//-----
```

這麼一來，*WndProc* 和 *OnCommand* 永遠不必改變，每有新要處理的訊息，只要在 *_messageEntries[]* 和 *_commandEntries[]* 兩個陣列中加上新元素，並針對新訊息撰寫新的處理常式即可。

這種觀念以及作法就是 MFC 的 Message Map 的雛形。MFC 把其中的動作包裝得更好更精緻（當然因此也就更複雜得多），成為一張龐大的訊息地圖；程式一旦獲得訊息，就可以按圖上溯，直到被處理為止。我將在第 3 章簡單模擬 MFC 的 Message Map，並在第 9 章「訊息映射與繞行」中詳細探索其完整內容。

對話盒的運作

Windows 的對話盒依其與父視窗的關係，分為兩類：

1. 「令其父視窗除能，直到對話盒結束」，這種稱為 *modal* 對話盒。
2. 「父視窗與對話盒共同運行」，這種稱為 *modeless* 對話盒。

比較常用的是 *modal* 對話盒。我就以 Generic 的 *!About* 對話盒做為說明範例。

為了做出一個對話盒，程式員必須準備兩樣東西：

1. 對話盒面板（*dialog template*）。這是在 RC 檔中定義的一個對話盒外貌，以各種方式決定對話盒的大小、字形、內部有哪些控制元件、各在什麼位置...等等。
2. 對話盒函式（*dialog procedure*）。其型態非常類似視窗函式，但是它通常只處理 *WM_INITDIALOG* 和 *WM_COMMAND* 兩個訊息。對話盒中的各個控制元件也都是小小視窗，各有自己的視窗函式，它們以訊息與其管理者（父視窗，也就是對話盒）溝通。而所有的控制元件傳來的訊息都是 *WM_COMMAND*，再由其參數分辨哪一種控制元件以及哪一種通告（*notification*）。

Modal 對話盒的啟動與結束，靠的是 *DialogBox* 和 *EndDialog* 兩個 API 函式。請看圖 1-4。

對話盒處理過訊息之後，應該傳回 *TRUE*；如果未處理訊息，則應該傳回 *FALSE*。這是因為你的對話盒函式之上層還有一個系統提供的預設對話盒函式。如果你傳回 *FALSE*，該預設對話盒函式就會接手處理。

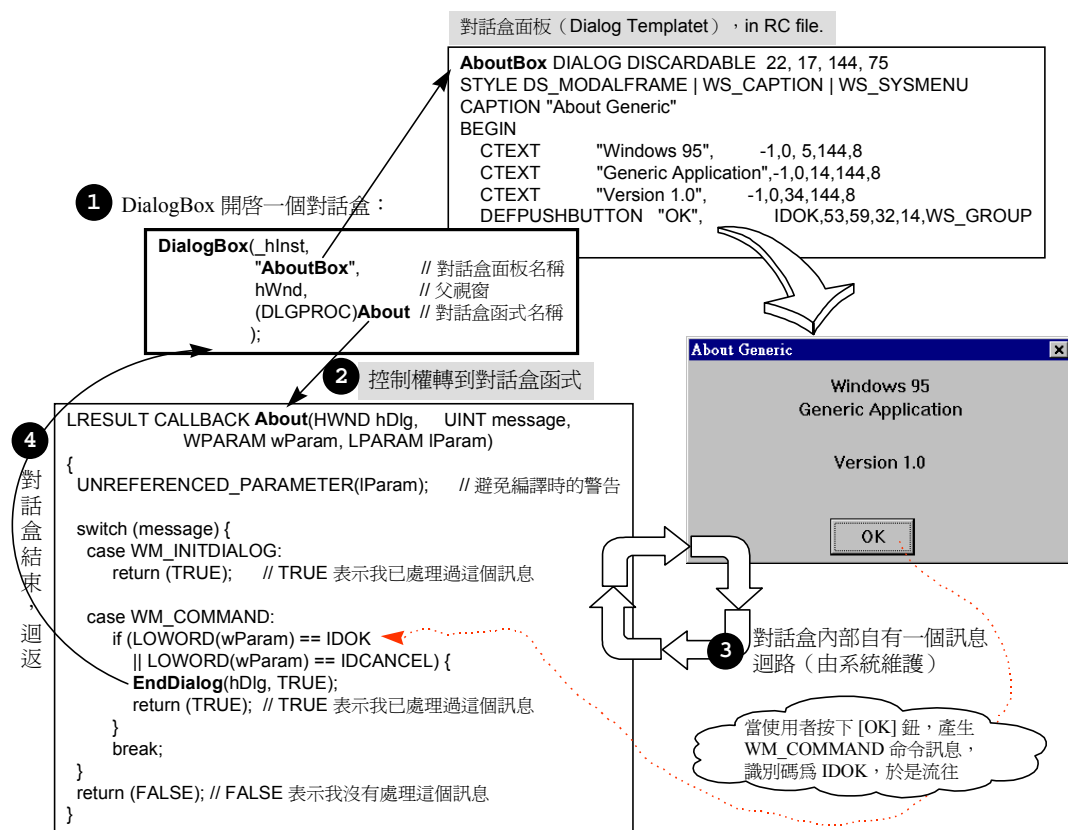


圖 1-4 對話盒的誕生、運作、結束

模組定義檔 (.DEF)

Windows 程式需要一個模組定義檔，將模組名稱、程式節區和資料節區的記憶體特性、模組堆積 (heap) 大小、堆疊 (stack) 大小、所有 callback 函式名稱...等等登記下來。下面是個實例：

```
NAME             Generic
DESCRIPTION      'Generic Sample'
EXETYPE          WINDOWS
STUB             'WINSTUB.EXE'
CODE             PRELOAD DISCARDABLE
DATA             PRELOAD MOVEABLE MULTIPLE
HEAPSIZE         4096
STACKSIZE        10240
EXPORTS
                MainWndProc @1
                AboutBox    @2
```

在 Visual C++ 整合環境中開發程式，不再需要特別準備 .DEF 檔，因為模組定義檔中的設定都有預設值。模組定義檔中的 STUB 指令用來指定所謂的 stub 程式（埋在 Windows 程式中的一個 DOS 程式，你所看到的 [This Program Requires Microsoft Windows](#) 或 [This Program Can Not Run in DOS mode](#) 就是此程式發出來的），Win16 允許程式員自設一個 stub 程式，但 Win32 不允許，換句話說在 Win32 之中 Stub 指令已經失效。

資源描述檔 (.RC)

RC 檔是一個以文字描述資源的地方。常用的資源有九項之多，分別是 ICON、CURSOR、BITMAP、FONT、DIALOG、MENU、ACCELERATOR、STRING、VERSIONINFO。還可能有新的資源不斷加入，例如 Visual C++ 4.0 就多了一種名為 TOOLBAR 的資源。這些文字描述需經過 RC 編譯器，才產生可使用的二進位碼。本例 Generic 示範 ICON、MENU 和 DIALOG 三種資源。

Windows 程式的生與死

我想你已經了解 Windows 程式的架構以及它與 Windows 系統之間的關係。對 Windows 訊息種類以及發生時機的透徹了解，正是程式設計的關鍵。現在我以視窗的誕生和死亡，說明訊息的發生與傳遞，以及應用程式的興起與結束，請看圖 1-5 及圖 1-6。

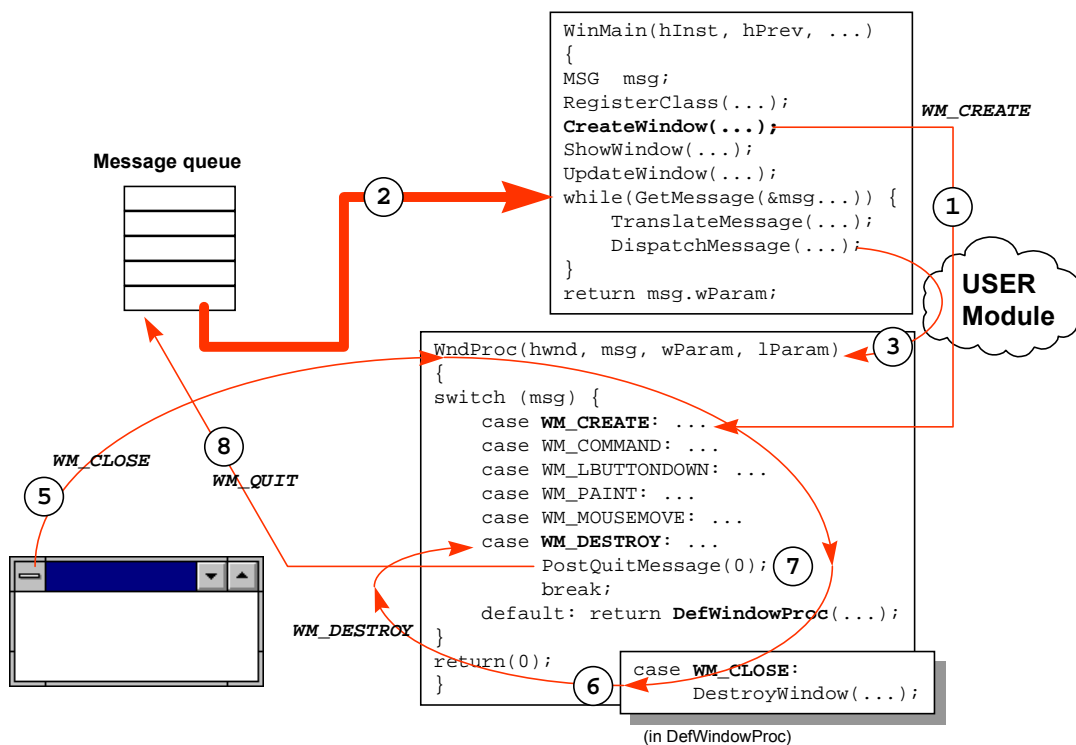


圖 1-5 視窗的生命週期（詳細說明請看圖 1-6）

1. 程式初始化過程中呼叫 *CreateWindow*，為程式建立了一個視窗，做為程式的螢幕舞台。*CreateWindow* 產生視窗之後會送出 *WM_CREATE* 直接給視窗函式，後者於是可以在此時機做些初始化動作（例如配置記憶體、開檔、讀初始資料...）。
2. 程式活著的過程中，不斷以 *GetMessage* 從訊息貯列中抓取訊息。如果這個訊息是 *WM_QUIT*，*GetMessage* 會傳回 0 而結束 *while* 迴路，進而結束整個程式。
3. *DispatchMessage* 透過 Windows USER 模組的協助與監督，把訊息分派至視窗函式。訊息將在該處被判別並處理。
4. 程式不斷進行 2. 和 3. 的動作。
5. 當使用者按下系統功能表中的 Close 命令項，系統送出 *WM_CLOSE*。通常程式的視窗函式不攔截此訊息，於是 *DefWindowProc* 處理它。
6. *DefWindowProc* 收到 *WM_CLOSE* 後，呼叫 *DestroyWindow* 把視窗清除。*DestroyWindow* 本身又會送出 *WM_DESTROY*。
7. 程式對 *WM_DESTROY* 的標準反應是呼叫 *PostQuitMessage*。
8. *PostQuitMessage* 沒什麼其他動作，就只送出 *WM_QUIT* 訊息，準備讓訊息迴路中的 *GetMessage* 取得，如步驟 2，結束訊息迴路。

圖 1-6 視窗的生命週期（請對照圖 1-5）

為什麼結束一個程式複雜如斯？因為作業系統與應用程式職司不同，二者是互相合作的關係，所以必需各做各的份內事，並互以訊息通知對方。如果不依據這個遊戲規則，可能就會有麻煩產生。你可以作一個小實驗，在視窗函式中攔截 *WM_DESTROY*，但不呼叫 *PostQuitMessage*。你會發現當選擇系統功能表中的 Close 時，螢幕上這個視窗消失了，（因為視窗摧毀及資料結構的釋放是 *DefWindowProc* 呼叫 *DestroyWindow* 完成的），但是應用程式本身並沒有結束（因為訊息迴路結束不了），它還留存在記憶體中。

閒置時間的處理：OnIdle

所謂閒置時間（idle time），是指「系統中沒有任何訊息等待處理」的時間。舉個例子，沒有任何程式使用計時器（timer，它會定時送來 *WM_TIMER*），使用者也沒有碰觸鍵盤和滑鼠或任何週邊，那麼，系統就處於所謂的閒置時間。

閒置時間常常發生。不要認為你移動滑鼠時產生一大堆的 *WM_MOUSEMOVE*，事實上夾雜在每一個 *WM_MOUSEMOVE* 之間就可能存在許多閒置時間。畢竟，電腦速度超乎想像。

背景工作最適宜在閒置時間完成。傳統的 SDK 程式如果要處理閒置時間，可以以下列迴路取代 *WinMain* 中傳統的訊息迴路：

```
while (TRUE) {
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) {
        if (msg.message == WM_QUIT)
            break;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else {
        OnIdle();
    }
}
```

原因是 *PeekMessage* 和 *GetMessage* 的性質不同。它們都是到訊息佇列中抓訊息，如果抓不到，程式的主執行緒（primary thread，是一個 UI 執行緒）會被作業系統虛懸住。當作業系統再次回來照顧此一執行緒，而發現訊息佇列中仍然是空的，這時候兩個 API 函式的行為就有不同了：

- *GetMessage* 會過門不入，於是作業系統再去照顧其他人。
- *PeekMessage* 會取回控制權，使程式得以執行一段時間。於是上述訊息迴路進入 *OnIdle* 函式中。

第 6 章的 HelloMFC 將示範如何在 MFC 程式中處理所謂的 idle time（p.403）。

Console 程式

說到 Windows 程式，一定得有 *WinMain*、訊息迴路、視窗函式。即使你只產生一個對話窗 (Dialog Box) 或訊息窗 (Message Box)，也有隱藏在 Windows API (*DialogBox* 和 *MessageBox*) 內裡的訊息迴路和視窗函式。

過去那種單單純純的 C/C++ 程式，有著簡單的 *main* 和 *printf* 的好時光到哪裡去了？夏天在陰涼的樹蔭下嬉戲，冬天在溫暖的爐火邊看書，啊，Where did the good times go？

其實說到 Win32 程式，並不是每個都如 Windows GUI 程式那麼複雜可怖。是的，你可以在 Visual C++ 中寫一個 "DOS-like" 程式，而且仍然可以呼叫部份的、不牽扯到圖形使用者介面 (GUI) 的 Win32 API。這種程式稱為 console 程式。甚至你還可以在 console 程式中使用部份的 MFC 類別 (同樣必須是與 GUI 沒有關連的)，例如處理陣列、串列等資料結構的 collection classes (*CArray*、*CList*、*CMap*)、與檔案有關的 *CFile*、*CStdioFile*。

我在 BBS 論壇上看到很多程式設計初學者，還沒有學習 C/C++，就想直接學習 Visual C++。並不是他們好高騖遠，而是他們以為 Visual C++ 是一種特殊的 C++ 語言。吃過苦頭的過來人以為初學所說的 Visual C++ programming 是指 MFC programming，所以大吃一驚 (沒有一點 C++ 基礎就要學習 MFC programming，當然是大吃一驚)。

在 Visual C++ 中寫純種的 C/C++ 程式？當然可以！不牽扯任何視窗、對話窗、控制元件，那就是 console 程式囉。雖然我這本書沒有打算照顧 C++ 初學者，然而我還是決定把 console 程式設計的一些相關心得放上來，同時也是因為我打算以 console 程式完成稍後的多緒程式範例。第 3 章的 MFC 六大技術模擬程式也都是 console 程式。

其實，除了 "DOS-like"，console 程式還另有妙用。如果你的程式和使用者之間是以巨量文字來互動，或許你會選擇使用 edit 控制元件 (或 MFC 的 *CEditView*)。但是你知道，電腦在一個純粹的「文字視窗」 (也就是 console 視窗) 中處理文字的顯現與捲動比較

快，你的程式動作也比較簡單。所以，你也可以在 Windows 程式中產生 console 視窗，獨立出來作業。

這也許不是你所認知的 console 程式。總之，有這種混合式的東西存在。

這一節將以我自己的一個極簡易的個人備份軟體 JBACKUP 為實例，說明 Win32 console 程式的撰寫，以及如何在其中使用 Win32 API（其實直接呼叫就是了）。再以另一個極小的程式 MFCCON 示範 MFC console 程式（用到了 MFC 的 *CStudioFile* 和 *CString*）。對於這麼小的程式而言，實在不需動用到整合環境下的什麼專案管理。至於複雜一點的程式，就請參考第 4 章最後一節「Console 程式的專案管理」。

Console 程式與 DOS 程式的差別

不少人把 DOS 程式和 console 程式混為一談，這是不對的。以下是各方面的比較。

製造方式

在 Windows 環境下的 DOS Box 中，或是在 Windows 版本的各種 C++ 編譯器套件的整合環境（IDE）中（第 4 章「Console 程式專案管理」），利用 Windows 編譯器、連結器做出來的程式，都是所謂 Win32 程式。如果程式是以 *main* 為進入點，呼叫 C runtime 函式和「不牽扯 GUI」的 Win32 API 函式，那麼就是一個 console 程式，console 視窗將成為其標準輸入和輸出裝置（*cin* 和 *cout*）。

過去在 DOS 環境下開發的程式，稱為 DOS 程式，它也是以 *main* 為程式進入點，可以呼叫 C runtime 函式。但，當然，不可能呼叫 Win32 API 函式。

程式能力

過去的 DOS 程式仍然可以在 Windows 的 DOS Box 中跑（Win95 的相容性極高，WinNT 的相容性稍差）。

Console 程式當然更沒有問題。由於 console 程式可以呼叫部份的 Win32 API（尤其是 KERNEL32.DLL 模組所提供的那一部份），所以它可以使用 Windows 提供的各種高階功能。它可以產生行程（processes），產生執行緒（threads）、取得虛擬記憶體的資訊、刺探作業系統的各種資料。但是它不能夠有華麗的外表 -- 因為它不能夠呼叫與 GUI 有關的各種 API 函式。

DOS 程式和 console 程式兩者都可以做 *printf* 輸出和 *cout* 輸出，也都可以做 *scanf* 輸入和 *cin* 輸入。

可執行檔格式

DOS 程式是所謂的 MZ 格式（MZ 是 Mark Zbikowski 的縮寫，他是 DOS 系統的一位主要建構者）。Console 程式的格式則和所有的 Win32 程式一樣，是所謂的 PE（Portable Executable）格式，意思是它可以被拿到任何 Win32 平台上執行。

Visual C++ 附有一個 DUMPBIN 工具軟體，可以觀察 PE 檔案格式。拿它來觀察本節的 JBACKUP 程式和 MFCCON 程式（以及第 3 章的所有程式），得到這樣的結果：

```
H:\u004\prog\jbackup.01>dumpbin /summary jbackup.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
```

```
Dump of file jbackup.exe
```

```
File Type: EXECUTABLE IMAGE
```

```
Summary
  5000 .data
  1000 .idata
  1000 .rdata
  5000 .text
```

拿它來觀察 DOS 程式，則得到這樣的結果：

```
C:\UTILITY>dumpbin /summary dsize.exe
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file dsize.exe
DUMPBIN : warning LNK4094: "dsize.exe" is an MS-DOS executable;
use EXEHDR to dump it

Summary
```

Console 程式的編譯聯結

你可以寫一個 `makefile`，編譯時指定常數 `/D_CONSOLE`，聯結時指定 `subsystem` 為 `console`，如下：

```
#0001 # filename : pedump.mak
#0002 # make file for pedump.exe
#0003 # usage : nmake pedump.msc (Visual C++ 5.0)
#0004
#0005 all : pedump.exe
#0006
#0007 pedump.exe: pedump.obj exedump.obj objdump.obj common.obj
#0008     link /subsystem:console /incremental:yes \
#0009         /machine:i386 /out:pedump.exe \
#0010         pedump.obj common.obj exedump.obj objdump.obj \
#0011         kernel32.lib user32.lib
#0012
#0013 pedump.obj : pedump.c
#0014     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c pedump.c
#0015
#0016 common.obj : common.c
#0017     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c common.c
#0018
#0019 exedump.obj : exedump.c
#0020     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c exedump.c
#0021
#0022 objdump.obj : objdump.c
#0023     cl /W3 /GX /Zi /YX /Od /DWIN32 /D_CONSOLE /FR /c objdump.c
```

如果是很簡單的情況，例如本節的 `JBACKUP` 只有一個 C 原始碼，那麼這樣也行（在命令列之下）：

```
cl jbackup.c <ENTER>    ← 將獲得 jbackup.exe
```


注意，環境變數要先設定好（請參考本章稍早的「如何產生 Generic.exe」一節）。

第 3 章的 Frame_ 程式則是這樣完成的：

```
cl my.cpp mfc.cpp <ENTER>    ← 將獲得 my.exe
```

至於到底該聯結哪些程式庫，全讓 CL.EXE 去傷腦筋就好了。

JBACKUP：Win32 Console 程式設計

撰寫 console 程式，有幾個重點請注意：

1. 進入點為 *main*。
2. 可以使用 *printf*、*scanf*、*cin*、*cout* 等標準輸出入裝置。
3. 可以呼叫和 GUI 無關的 Win32 API。

我的這個 JBACKUP 程式可以有一個或兩個參數，用法如下：

```
C:\SomeoneDir>JBACKUP SrcDir [DstDir]
```

例如 JBACKUP g: k:

將磁碟目錄 SrcDir 中的新檔案拷貝到磁碟目錄 DstDir，
並將 DstDir 的贅餘檔案殺掉。

如果沒有指定 DstDir，預設為 k:（那是我的可寫入光碟機 -- MO -- 的代碼啦）
並將 k: 的磁碟目錄設定與 SrcDir 相同。

例如 JBACKUP g:

而目前 g: 是 g:\u002\doc

那麼相當於把 g:\u002\doc 備份到 k:\u002\doc 中，並殺掉 k:\u002\doc 的贅餘檔案。

JBACK 檢查 SrcDir 中所有的檔案和 DstDir 中所有的檔案，把比較新的檔案從 SrcDir
中拷貝到 DstDir 去，並把 DstDir 中多出來的檔案刪除，使 SrcDir 和 DstDir 的檔案保

持完全相同。之所以不做 `xcopy` 完全拷貝動作，為的是節省拷貝時間（做為備份裝置，通常是軟碟或磁帶或可讀寫光碟 MO，讀寫速度並不快）。

JBACKUP 沒有能力處理 `SrcDir` 底下的子目錄檔案。如果要處理子目錄，漂亮的作法是使用遞迴（`recursive`），但是有點傷腦筋，這一部份留給你了。我的打字速度還算快，多切換幾次磁碟目錄不是問題，呵呵呵。

JBACKUP 使用以下數個 Win32 APIs：

```
GetCurrentDirectory
FindFirstFile
FindNextFile
CompareFileTime
CopyFile
DeleteFile
```

在處理完畢命令列參數中的 `SrcDir` 和 `DstDir` 之後，JBACKUP 先把 `SrcDir` 中的所有檔案（不含子目錄檔案）搜尋一遍，儲存在一個陣列 `srcFiles[]` 中，每個陣列元素是一個我自定的 `SRCFILE` 資料結構：

```
typedef struct _SRCFILE
{
    WIN32_FIND_DATA fd;
    BOOL bIsNew;
} SRCFILE;

SRCFILE srcFiles[FILEMAX];
WIN32_FIND_DATA fd;

// prepare srcFiles[...]...
bRet = TRUE;
iSrcFiles = 0;
hFile = FindFirstFile(SrcDir, &fd);
while (hFile != INVALID_HANDLE_VALUE && bRet)
{
    if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
        srcFiles[iSrcFiles].fd = fd;
        srcFiles[iSrcFiles].bIsNew = FALSE;
        iSrcFiles++;
    }
}
```

```
        bRet = FindNextFile(hFile, &fd);
    }
```

再把 DstDir 中的所有檔案（不含子目錄檔案）搜尋一遍，儲存在一個 destFiles[] 陣列中，每個陣列元素是一個我自定的 DESTFILE 資料結構：

```
typedef struct _DESTFILE
{
    WIN32_FIND_DATA fd;
    BOOL bMatch;
} DESTFILE;

DESTFILE destFiles[FILEMAX];
WIN32_FIND_DATA fd;

bRet = TRUE;
iDestFiles = 0;
hFile = FindFirstFile(DstDir, &fd);
while (hFile != INVALID_HANDLE_VALUE && bRet)
{
    if (fd.dwFileAttributes == FILE_ATTRIBUTE_ARCHIVE) {
        destFiles[iDestFiles].fd = fd;
        destFiles[iDestFiles].bMatch = FALSE;
        iDestFiles++;
    }
    bRet = FindNextFile(hFile, &fd);
}
```

然後比對 srcFiles[] 和 destFiles[] 之中的所有檔案名稱以及建檔日期，找出 srcFiles[] 中的哪些檔案比 destFiles[] 中的檔案更新，然後將其 bIsNew 欄位設為 TRUE。同時也對存在於 destFiles[] 中而不存在於 srcFiles[] 中的檔案，令其 bMatch 欄位為 FALSE。

最後，檢查 srcFiles[] 中的所有檔案，將 bIsNew 欄位為 TRUE 者，拷貝到 DstDir 去。並檢查 destFiles[] 中的所有檔案，將 bMatch 欄位為 FALSE 者統統殺掉。

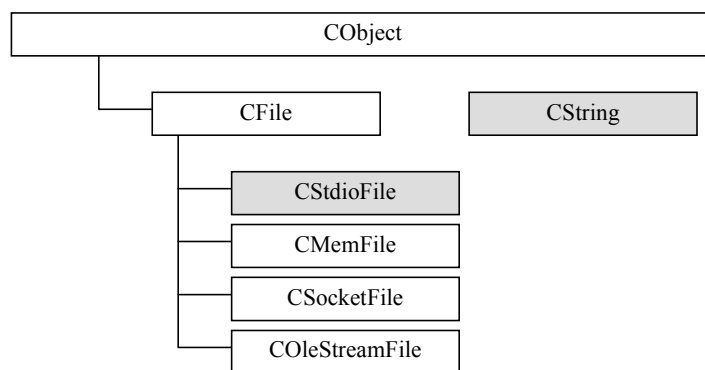
JBACKUP 的原始碼與可執行檔放在書附碟片的 Jbackup.01 子目錄中。

MFCCON：MFC Console 程式設計

當你的進度還在第 1 章的 Win32 基本程式觀念，我卻開始講如何設計一個 MFC console 程式，是否有點時地不宜？

是有一點！所以我挑一個最單純而無與別人攀纏糾葛的 MFC 類別，寫一個 40 行的小程式。目標純粹是爲了做一個導入，並與 Win32 console 程式做一比較。

我所挑選的兩個單純的 MFC 類別是 *CStdioFile* 和 *CString*：



在 MFC 之中，*CFile* 用來處理正常的檔案 I/O 動作。*CStdioFile* 衍生自 *CFile*，一個 *CStdioFile* 物件代表以 C runtime 函式 *fopen* 所開啓的一個 stream 檔案。Stream 檔案有緩衝區，可以文字模式（預設情況）或二進位模式開啓。

CString 物件代表一個字串，是一個完全獨立的類別。

我的例子用來計算小於 100 的所有費伯納契數列（Fabonacci sequence）。費伯納契數列的計算方式是：

1. 頭兩個數爲 1。
2. 接下來的每一個數是前兩個數的和。

以下便是 MFCCON.CPP 內容：

```
#0001 // File : MFCCON.CPP
#0002 // Author : J.J.Hou / Top Studio
#0003 // Date : 1997.04.06
#0004 // Goal : Fibonacci sequencee, less than 100
#0005 // Build : cl /MT mfcon.cpp (/MT means Multithreading)
#0006
#0007 #include <afx.h>
#0008 #include <stdio.h>
#0009
#0010 int main()
#0011 {
#0012     int lo, hi;
#0013     CString str;
#0014     CStdioFile fFibo;
#0015
#0016     fFibo.Open("FIBO.DAT", CFile::modeWrite |
#0017                 CFile::modeCreate | CFile::typeText);
#0018
#0019     str.Format("%s\n", "Fibonacci sequencee, less than 100 :");
#0020     printf("%s", (LPCTSTR) str);
#0021     fFibo.WriteString(str);
#0022
#0023     lo = hi = 1;
#0024
#0025     str.Format("%d\n", lo);
#0026     printf("%s", (LPCTSTR) str);
#0027     fFibo.WriteString(str);
#0028
#0029     while (hi < 100)
#0030     {
#0031         str.Format("%d\n", hi);
#0032         printf("%s", (LPCTSTR) str);
#0033         fFibo.WriteString(str);
#0034         hi = lo + hi;
#0035         lo = hi - lo;
#0036     }
#0037
#0038     fFibo.Close();
#0039     return 0;
#0040 }
```

以下是執行結果（在 console 視窗和 FIBO.DAT 檔案中，結果都一樣）：

Fibonacci sequence, less than 100 :

```
1
1
2
3
5
8
13
21
34
55
89
```

這麼簡單的例子中，我們看到 MFC Console 程式的幾個重點：

1. 程式進入點仍為 *main*
2. 需含入所使用之類別的表頭檔（本例為 *AFX.H*）
3. 可直接使用與 GUI 無關的 MFC 類別（本例為 *CStdioFile* 和 *CString*）
4. 編輯時需指定 */MT*，表示使用多執行緒版本的 C runtime 函式庫。

第 4 點需要多做說明。在 MFC console 程式中一定要指定多緒版的 C runtime 函式庫，所以必須使用 */MT* 選項。如果不做這項設定，會出現這樣的聯結錯誤：

```
Microsoft (R) 32-Bit Incremental Linker Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

/out:mfccon.exe
mfccon.obj
nafxcw.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __endthreadex
nafxcw.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __beginthreadex
mfccon.exe : fatal error LNK1120: 2 unresolved externals
```

表示它找不到 *__beginthreadex* 和 *__endthreadex*。怪了，我們的程式有呼叫它們嗎？沒有，但是 MFC 有！這兩個函式將在稍後與執行緒有關的小節中討論。

MFCCON 的原始碼與可執行檔放在書附碟片的 *mfccon.01* 子目錄中。

什麼是 C Runtime 函式庫的多緒版本

當 C runtime 函式庫於 1970s 年代產生出來時，PC 的記憶體容量還很小，多緒是個新奇觀念，更別提什麼多執行緒了。因此以當時產品為基礎所演化的 C runtime 函式庫在多緒（multithreaded）的表現上有嚴重問題，無法被多緒程式使用。

利用各種同步機制（synchronous mechanism）如 critical section、mutex、semaphore、event，可以重新開發支援多執行緒的 runtime 函式庫。問題是，加上這樣的威力，可能導致程式碼大小和執行效率都遭受不良波及 -- 即使你只啟動了一個執行緒。

Visual C++ 的折衷方案是提供兩種版本的 C runtime 函式庫。一種版本給單緒程式使用，一種版本給多緒程式使用。多緒版本的重大改變是，第一，變數如 errno 會現在變成每個執行緒各擁有一個。第二，多緒版中的資料結構以同步機制加以保護。

Visual C++ 一共有六個 C runtime 函式庫產品供你選擇：

◆ Single-Threaded (static)	libc.lib	898,826
◆ Multithreaded (static)	libcmtd.lib	951,142
◆ Multithreaded DLL	msvcr.lib	5,510,000
◆ Debug Single-Threaded (static)	libcd.lib	2,374,542
◆ Debug Multithreaded (static)	libcmtd.lib	2,949,190
◆ Debug Multithreaded DLL	msvcrtd.lib	803,418

Visual C++ 編譯器提供下列選項，讓我們決定使用哪一個 C runtime 函式庫：

◆ /ML	Single-Threaded (static)
◆ /MT	Multithreaded (static)
◆ /MD	Multithreaded DLL (dynamic import library)
◆ /MLd	Debug Single-Threaded (static)
◆ /MTd	Debug Multithreaded (static)
◆ /MDd	Debug Multithreaded DLL (dynamic import library)

行程與執行緒（Process and Thread）

OS/2、Windows NT 以及 Windows 95 都支援多執行緒，這帶給 PC 程式員一股令人興奮的氣氛。然而它帶來的不全然是利多，如果不謹慎小心地處理執行緒的同步問題，程式的錯誤以及除錯所花的時間可能使你發誓再也不碰「執行緒」這種東西。

我們習慣以行程（process）表示一個執行中的程式，並且以為它是 CPU 排程單位。事實上執行緒才是排程單位。

核心物件

首先讓我解釋什麼叫作「核心物件」（kernel object）。「GDI 物件」是大家比較熟悉的東西，我們利用 GDI 函式所產生的一支筆（Pen）或一支刷（Brush）都是所謂的「GDI 物件」。但什麼又是「核心物件」呢？

你可以說核心物件是系統的一種資源（噢，這說法對 GDI 物件也適用），系統物件一旦產生，任何應用程式都可以開啓並使用該物件。系統給予核心物件一個計數值（usage count）做為管理之用。核心物件包括下列數種：

核心物件	產生方法
event	<i>CreateEvent</i>
mutex	<i>CreateMutex</i>
semaphore	<i>CreateSemaphore</i>
file	<i>CreateFile</i>
file-mapping	<i>CreateFileMapping</i>
process	<i>CreateProcess</i>
thread	<i>CreateThread</i>

前三者用於執行緒的同步化：file-mapping 物件用於記憶體映射檔（memory mapping file），process 和 thread 物件則是本節的主角。這些核心物件的產生方式（也就是我們

所使用的 API) 不同，但都會獲得一個 `handle` 做為識別；每被使用一次，其對應的計數值就加 1。核心物件的結束方式相當一致，呼叫 `CloseHandle` 即可。

「process 物件」究竟做什麼用呢？它並不如你想像中用來「執行程式碼」；不，程式碼的執行是執行緒的工作，「process 物件」只是一個資料結構，系統用它來管理行程。

一個行程的誕生與死亡

執行一個程式，必然就產生一個行程 (process)。最直接的程式執行方式就是在 shell (如 Win95 的檔案總管或 Windows 3.x 的檔案管理員) 中以滑鼠雙擊某一個可執行檔圖示 (假設其為 `App.exe`)，執行起來的 `App` 行程其實是 shell 呼叫 `CreateProcess` 啟動的。讓我們看看整個流程：

1. shell 呼叫 `CreateProcess` 啟動 `App.exe`。
2. 系統產生一個「行程核心物件」，計數值為 1。
3. 系統為此行程建立一個 4GB 位址空間。
4. 載入器將必要的碼載入到上述位址空間中，包括 `App.exe` 的程式、資料，以及所需的動態連結函式庫 (DLLs)。載入器如何知道要載入哪些 DLLs 呢？它們被記錄在可執行檔 (PE 檔案格式) 的 `.idata section` 中。
5. 系統為此行程建立一個執行緒，稱為主執行緒 (primary thread)。執行緒才是 CPU 時間的分配對象。
6. 系統呼叫 C runtime 函式庫的 `Startup code`。
7. `Startup code` 呼叫 `App` 程式的 `WinMain` 函式。
8. `App` 程式開始運作。
9. 使用者關閉 `App` 主視窗，使 `WinMain` 中的訊息迴路結束掉，於是 `WinMain` 結束。
10. 回到 `Startup code`。
11. 回到系統，系統呼叫 `ExitProcess` 結束行程。

可以說，透過這種方式執行起來的所有 Windows 程式，都是 shell 的子行程。本來，母行程與子行程之間可以有某些關係存在，但 shell 在呼叫 *CreateProcess* 時已經把母子之間的臍帶關係剪斷了，因此它們事實上是獨立個體。稍後我會提到如何剪斷子行程的臍帶。

產生子行程

你可以寫一個程式，專門用來啟動其他的程式。關鍵就在於你會不會使用 *CreateProcess*。這個 API 函式有眾多參數：

```
CreateProcess(  
    LPCSTR lpApplicationName,  
    LPSTR lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL bInheritHandles,  
    DWORD dwCreationFlags,  
    LPVOID lpEnvironment,  
    LPCSTR lpCurrentDirectory,  
    LPSTARTUPINFO lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

第一個參數 *lpApplicationName* 指定可執行檔檔名。第二個參數 *lpCommandLine* 指定欲傳給新行程的命令列（command line）參數。如果你指定了 *lpApplicationName*，但沒有副檔名，系統並不會主動為你加上 .EXE 副檔名；如果沒有指定完整路徑，系統就只在目前工作目錄中尋找。但如果你指定 *lpApplicationName* 為 *NULL* 的話，系統會以 *lpCommandLine* 的第一個「段落」（我的意思其實是術語中所謂的 token）做為可執行檔檔名；如果這個檔名沒有指定副檔名，就採用預設的 ".EXE" 副檔名；如果沒有指定路徑，Windows 就依照五個搜尋路徑來尋找可執行檔，分別是：

1. 呼叫者的可執行檔所在目錄
2. 呼叫者的目前工作目錄
3. Windows 目錄
4. Windows System 目錄

5. 環境變數中的 path 所設定的各目錄

讓我們看看實例：

```
CreateProcess("E:\\CWIN95\\NOTEPAD.EXE", "README.TXT", ...);
```

系統將執行 E:\\CWIN95\\NOTEPAD.EXE，命令列參數是 "README.TXT"。如果我們這樣子呼叫：

```
CreateProcess(NULL, "NOTEPAD README.TXT", ...);
```

系統將依照搜尋次序，將第一個被找到的 NOTEPAD.EXE 執行起來，並轉送命令列參數 "README.TXT" 給它。

建立新行程之前，系統必須做出兩個核心物件，也就是「行程物件」和「執行緒物件」。CreateProcess 的第三個參數和第四個參數分別指定這兩個核心物件的安全屬性。至於第五個參數（TRUE 或 FALSE）則用來設定這些安全屬性是否要被繼承。關於安全屬性及其可被繼承的性質，礙於本章的定位，我不打算在此介紹。

第六個參數 dwCreationFlags 可以是許多常數的組合，會影響到行程的建立過程。這些常數中比較常用的是 CREATE_SUSPENDED，它會使得子行程產生之後，其主執行緒立刻被暫停執行。

第七個參數 lpEnvironment 可以指定行程所使用的環境變數區。通常我們會讓子行程繼承父行程的環境變數，那麼這裡要指定 NULL。

第八個參數 lpCurrentDirectory 用來設定子行程的工作目錄與工作磁碟。如果指定 NULL，子行程就會使用父行程的工作目錄與工作磁碟。

第九個參數 lpStartupInfo 是一個指向 STARTUPINFO 結構的指標。這是一個龐大的結構，可以用來設定視窗的標題、位置與大小，詳情請看 API 使用手冊。

最後一個參數是一個指向 PROCESS_INFORMATION 結構的指標：

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

當系統為我們產生「行程物件」和「執行緒物件」，它會把兩個物件的 `handle` 填入此結構的相關欄位中，應用程式可以從這裡獲得這些 `handles`。

如果一個行程想結束自己的生命，只要呼叫：

```
VOID ExitProcess(UINT fuExitCode);
```

就可以了。如果行程想結束另一個行程的生命，可以使用：

```
BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

很顯然，只要你有某個行程的 `handle`，就可以結束它的生命。*TerminateProcess* 並不被建議使用，倒不是因為它的權力太大，而是因為一般行程結束時，系統會通知該行程所開啓（所使用）的所有 DLLs，但如果你以 *TerminateProcess* 結束一個行程，系統不會做這件事，而這恐怕不是你所希望的。

前面我曾說過所謂割斷臍帶這件事情，只要你把子行程以 *CloseHandle* 關閉，就達到了目的。下面是個例子：

```
PROCESS_INFORMATION ProcInfo;
BOOL fSuccess;

fSuccess = CreateProcess(...,&ProcInfo);
if (fSuccess) {
    CloseHandle(ProcInfo.hThread);
    CloseHandle(ProcInfo.hProcess);
}
```

一個執行緒的誕生與死亡

程式碼的執行，是執行緒的工作。當一個行程建立起來，主執行緒也產生。所以每一個 Windows 程式一開始就有了一個執行緒。我們可以呼叫 *CreateThread* 產生額外的執行緒，系統會幫我們完成下列事情：

1. 配置「執行緒物件」，其 *handle* 將成為 *CreateThread* 的傳回值。
2. 設定計數值為 1。
3. 配置執行緒的 *context*。
4. 保留執行緒的堆疊。
5. 將 *context* 中的堆疊指標暫存器（SS）和指令指標暫存器（IP）設定妥當。

看看上面的態勢，的確可以顯示出執行緒是 CPU 分配時間的單位。所謂工作切換（*context switch*）其實就是對執行緒的 *context* 的切換。

程式若欲產生一個新執行緒，呼叫 *CreateThread* 即可辦到：

```
CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,  
            DWORD dwStackSize,  
            LPTHREAD_START_ROUTINE lpStartAddress,  
            LPVOID lpParameter,  
            DWORD dwCreationFlags,  
            LPDWORD lpThreadId  
            );
```

第一個參數表示安全屬性的設定以及繼承，請參考 API 手冊。Windows 95 忽略此一參數。第二個參數設定堆疊的大小。第三個參數設定「執行緒函式」名稱，而該函式的參數則在這裡的第四個參數設定。第五個參數如果是 0，表示讓執行緒立刻開始執行，如果是 *CREATE_SUSPENDED*，則是要求執行緒暫停執行（那麼我們必須呼叫 *ResumeThread* 才能令其重新開始）。最後一個參數是個指向 *DWORD* 的指標，系統會把執行緒的 ID 放在這裡。

上面我所說的「執行緒函式」是什麼？讓我們看個實例：

```

VOID ReadTime(VOID);
HANDLE hThread;
DWORD ThreadID;

hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ReadTime,
                        NULL, 0, &ThreadID);

...
//-----
// thread 函式。
// 不斷利用 GetSystemTime 取系統時間，
// 並將結果顯示在對話盒 _hWndDlg 的 IDE_TIMER 欄位上。
//-----
VOID ReadTime(VOID)
{
    char str[50];
    SYSTEMTIME st;

    while(1) {
        GetSystemTime(&st);
        sprintf(str, "%u:%u:%u", st.wHour, st.wMinute, st.wSecond);
        SetDlgItemText (_hWndDlg, IDE_TIMER, str);
        Sleep (1000); // 延遲一秒。
    }
}

```

當 *CreateThread* 成功，系統為我們把一個執行緒該有的東西都準備好。執行緒的主體在哪裡呢？就在所謂的執行緒函式。執行緒與執行緒之間，不必考慮控制權釋放的問題，因為 Win32 作業系統是強制性多工。

執行緒的結束有兩種情況，一種是壽終正寢，一種是未得善終。前者是執行緒函式正常結束退出，那麼執行緒也就自然而然終結了。這時候系統會呼叫 *ExitThread* 做些善後清理工作（其實執行緒中也可以自行呼叫此函式以結束自己）。但是像上面那個例子，執行緒根本是個無窮迴路，如何終結？一者是行程結束（自然也就導至執行緒的結束），二者是別的執行緒強制以 *TerminateThread* 將它終結掉。不過，*TerminateThread* 太過毒辣，非必要還是少用為妙（請參考 API 手冊）。

以 `_beginthreadex` 取代 `CreateThread`

別忘了 Windows 程式除了呼叫 Win32 API，通常也很難避免呼叫任何一個 C runtime 函式。爲了保證多緒情況下的安全，C runtime 函式庫必須爲每一個執行緒做一些簿記工作。沒有這些工作，C runtime 函式庫就不知道要爲每一個執行緒配置一塊新的記憶體，做爲執行緒的區域變數用。因此，`CreateThread` 有一個名爲 `_beginthreadex` 的外包函式，負責額外的簿記工作。

請注意函式名稱的底線符號。它必須存在，因爲這不是個標準的 ANSI C runtime 函式。`_beginthreadex` 的參數和 `CreateThread` 的參數其實完全相同，不過其型別已經被「淨化」了，不再有 Win32 型別包裝。這原本是爲了要讓這個函式能夠移植到其他作業系統，因爲微軟希望 `_beginthreadex` 能夠被實作於其他平台，不需要和 Windows 有關、不需要含入 `windows.h`。但實際情況是，你還是得呼叫 `CloseHandle` 以關閉執行緒，而 `CloseHandle` 卻是個 Win32 API，所以你還是需要含入 `windows.h`、還是和 Windows 脫離不了關係。微軟空有一個好主意，卻沒能落實它。

把 `_beginthreadex` 視爲 `CreateThread` 的一個看起來比較有趣的版本，就對了：

```
unsigned long _beginthreadex(  
    void *security,  
    unsigned stack_size,  
    unsigned (__stdcall *start_address)(void *),  
    void *arglist,  
    unsigned initflag,  
    unsigned* thrdaddr  
);
```

`_beginthreadex` 所傳回的 `unsigned long` 事實上就是一個 Win32 HANDLE，指向新執行緒。換句話說傳回值和 `CreateThread` 相同，但 `_beginthreadex` 另外還設立了 `errno` 和 `doserrno`。

下面是一個最簡單的使用範例：

```
#0001 #include <windows.h>  
#0002 #include <process.h>
```

```
#0003 unsigned __stdcall myfunc(void* p);
#0004
#0005 void main()
#0006 {
#0007     unsigned long thd;
#0008     unsigned tid;
#0009
#0010     thd = _beginthreadex(NULL,
#0011                          0,
#0012                          myfunc,
#0013                          0,
#0014                          0,
#0015                          &tid );
#0016     if (thd != NULL)
#0017     {
#0018         CloseHandle(thd);
#0019     }
#0020 }
#0021
#0022 unsigned __stdcall myfunc(void* p)
#0023 {
#0024     // do your job...
#0025 }
```

針對 Win32 API *ExitThread*，也有一個對應的 C runtime 函式：*_endthreadex*。它只需要一個參數，就是由 *_beginthreadex* 第 6 個參數傳回來的 ID 值。

關於 *_beginthreadex* 和 *_endthreadex*，以及執行緒的其他各種理論基礎、程式技術、使用技巧，可參考由 Jim Beveridge & Robert Wiener 合著，Addison Wesley 出版的 *Multithreading Applications in Win32* 一書（Win32 多緒程式設計 / 侯俊傑譯 / 碁峰出版）。

執行緒優先權 (Priority)

優先權是排程的重要依據。優先權高的執行緒，永遠先獲得 CPU 的青睞。當然啦，作業系統會視情況調整各個執行緒的優先權。例如前景執行緒的優先權應該調高一些，背景執行緒的優先權應該調低一些。

執行緒的優先權範圍從 0 (最低) 到 31 (最高)。當你產生執行緒，並不是直接以數值指定其優先權，而是採用兩個步驟。第一個步驟是指定「優先權等級 (Priority Class)」給行程，第二步驟是指定「相對優先權」給該行程所擁有的執行緒。圖 1-7 是優先權等級的描述，其中的代碼在 *CreateProcess* 的 *dwCreationFlags* 參數中指定。如果你不指定，系統預設給的是 *NORMAL_PRIORITY_CLASS* -- 除非父行程是 *IDLE_PRIORITY_CLASS* (那麼子行程也會是 *IDLE_PRIORITY_CLASS*)。

等級	代碼	優先權值
idle	IDLE_PRIORITY_CLASS	4
normal	NORMAL_PRIORITY_CLASS	9 (前景) 或 7 (背景)
high	HIGH_PRIORITY_CLASS	13
realtime	REALTIME_PRIORITY_CLASS	24

圖 1-7 Win32 執行緒的優先權等級劃分

- "idle" 等級只有在 CPU 時間將被浪費掉時 (也就是前一節所說的閒置時間) 才執行。此等級最適合於系統監視軟體，或螢幕保護軟體。
- "normal" 是預設等級。系統可以動態改變優先權，但只限於 "normal" 等級。當行程變成前景，執行緒優先權提昇為 9，當行程變成背景，優先權降低為 7。
- "high" 等級是為了立即反應的需要，例如使用者按下 Ctrl+Esc 時立刻把工作管理器 (task manager) 帶出場。
- "realtime" 等級幾乎不會被一般應用程式使用。就連系統中控制滑鼠、鍵盤、

磁碟狀態重新掃描、Ctrl+Alt+Del 等的執行緒都比 "realtime" 優先權還低。這種等級使用在「如果不在某個時間範圍內被執行的話，資料就要遺失」的情況。這個等級一定得在正確評估之下使用之，如果你把這樣的等級指定給一般的（並不會常常被阻塞的）執行緒，多工環境恐怕會癱瘓，因為這個執行緒有如此高的優先權，其他執行緒再沒有機會被執行。

上述四種等級，每一個等級又映射到某一範圍的優先權值。*IDLE_* 最低，*NORMAL_* 次之，*HIGH_* 又次之，*REALTIME_* 最高。在每一個等級之中，你可以使用 *SetThreadPriority* 設定精確的優先權，並且可以稍高或稍低於該等級的正常值（範圍是兩個點數）。你可以把 *SetThreadPriority* 想像是一種微調動作。

<i>SetThreadPriority</i> 的參數	微調幅度
<i>THREAD_PRIORITY_LOWEST</i>	-2
<i>THREAD_PRIORITY_BELOW_NORMAL</i>	-1
<i>THREAD_PRIORITY_NORMAL</i>	不變
<i>THREAD_PRIORITY_ABOVE_NORMAL</i>	+1
<i>THREAD_PRIORITY_HIGHEST</i>	+2

除了以上五種微調，另外還可以指定兩種微調常數：

<i>SetThreadPriority</i> 的參數	面對任何等級的調整結果：	面對 "realtime" 等級的調整結果：
<i>THREAD_PRIORITY_IDLE</i>	1	16
<i>THREAD_PRIORITY_TIME_CRITICAL</i>	15	31

這些情況可以以圖 1-8 作為總結。

優先權等級	idle	lowest	below normal	normal	above normal	highest	time critical
idle	1	2	3	4	5	6	15
normal (背景)	1	5	6	7	8	9	15
normal (前景)	1	7	8	9	10	11	15
high	1	11	12	13	14	15	15
realtime	16	22	23	24	25	26	31

圖 1-8 Win32 執行緒優先權

多緒程式設計實例

我設計了一個 MltiThrd 程式，放在書附碟片的 MltiThrd.01 子目錄中。這個程式一開始產生五個執行緒，優先權分別微調 -2、-1、0、+1、+2，並且虛懸不執行：

```

HANDLE _hThread[5]; // global variable
...
LONG APIENTRY MainWndProc (HWND hWnd, UINT message,
                           UINT wParam, LONG lParam)
{
    DWORD ThreadID[5];
    static DWORD ThreadArg[5] = {HIGHEST_THREAD,    // 0x00
                                  ABOVE_AVE_THREAD, // 0x3F
                                  NORMAL_THREAD,     // 0x7F
                                  BELOW_AVE_THREAD,   // 0xBF
                                  LOWEST_THREAD       // 0xFF
                                  }; // 用來調整四方形顏色
    ...
    for(i=0; i<5; i++) // 產生 5 個 threads
        _hThread[i] = CreateThread(NULL,
                                     0,
                                     (LPTHREAD_START_ROUTINE)ThreadProc,
                                     &ThreadArg[i],
                                     CREATE_SUSPENDED,
                                     &ThreadID[i]);

    // 設定 thread priorities
    SetThreadPriority(_hThread[0], THREAD_PRIORITY_HIGHEST);
    SetThreadPriority(_hThread[1], THREAD_PRIORITY_ABOVE_NORMAL);

```

```

SetThreadPriority(_hThread[2], THREAD_PRIORITY_NORMAL);
SetThreadPriority(_hThread[3], THREAD_PRIORITY_BELOW_NORMAL);
SetThreadPriority(_hThread[4], THREAD_PRIORITY_LOWEST);
...
}

```

當使用者按下【Resume Threads】選單項目後，五個執行緒如猛虎出柙，同時衝出來。這五個執行緒使用同一個執行緒函式 *ThreadProc*。我在 *ThreadProc* 中以不斷的 *Rectangle* 動作表示執行緒的進行。所以我們可以從畫面上觀察執行緒的進度。我並且設計了兩種延遲方式，以利觀察。第一種方式是在每一次迴路之中使用 *Sleep(10)*，意思是先睡 10 個毫秒，之後再醒來；這段期間，CPU 可以給別人使用。第二種方式是以空迴路 30000 次做延遲；空迴路期間 CPU 不能給別人使用（事實上 CPU 正忙碌於那 30000 次空轉）。

```

UINT _uDelayType=NODELAY; // global variable
...
VOID ThreadProc(DWORD *ThreadArg)
{
    RECT rect;
    HDC hDC;
    HANDLE hBrush, hOldBrush;
    DWORD dwThreadHits = 0;
    int iThreadNo, i;
    ...
    do
    {
        dwThreadHits++; // 計數器

        // 畫出四方形，代表 thread 的進行
        Rectangle(hDC, *(ThreadArg), rect.bottom-(dwThreadHits/10),
            *(ThreadArg)+0x40, rect.bottom);

        // 延遲...
        if (_uDelayType == SLEEPDELAY)
            Sleep(10);
        else if (_uDelayType == FORLOOPDELAY)
            for (i=0; i<30000; i++);
        else // _uDelayType == NODELAY)
            { }
    } while (dwThreadHits < 1000); // 巡迴 1000 次
    ...
}

```

圖 1-9 是執行畫面。注意，先選擇延遲方式 ("for loop delay" 或 "sleep delay")，再按下【Resume Thread】。如果你選擇「for loop delay」（圖 1-9a），你會看到執行緒 0（優先權最高）幾乎一路衝到底，然後才是執行緒 1（優先權次之），然後是執行緒 2（優先權再次之）...。但如果你選擇的「sleep delay」（圖 1-9b），所有執行緒不分優先權高低，同時行動。關於執行緒的排程問題，我將在第 14 章做更多的討論。

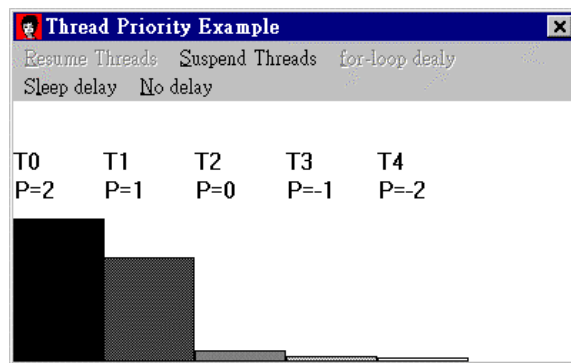


圖 1-9a MltiThrd.exe 的執行畫面（「for loop delay」）

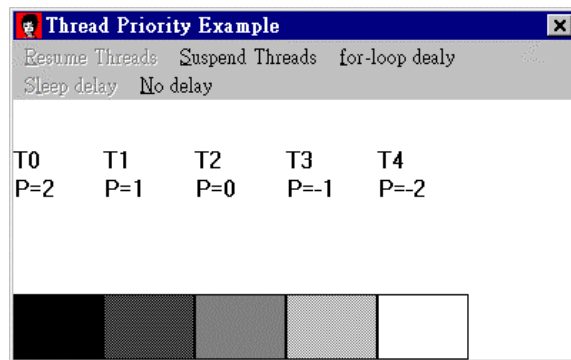


圖 1-9b MltiThrd.exe 的執行畫面（「sleep delay」）

注意：為什麼圖 1-9a 中執行緒 1 尚未完成，執行緒 2~4 竟然也有機會偷得一點點 CPU 時間呢？這是排程器的巧妙設計，動態調整執行緒的優先權。是啊，總不能讓優先權低的執行緒直到天荒地老，沒有一點點獲得。關於執行緒排程問題，第 14 章有更多的討論。

圖 1-10 是以 Process Viewer(Visual C++ 5.0 所附工具)觀察 Mltithrd.exe 的執行結果。圖上方出現目前所有的行程，點選其中的 MLTITHRD.EXE，果然在視窗下方出現六個執行緒，其中包括主執行緒（優先權已被調整為 10）。

The screenshot shows the 'Process Viewer Application' window. The top pane lists various processes, with 'MLTITHRD.EXE' selected. The bottom pane displays the threads for this process.

Process	PID	Base Priority	Num. Threads	Type	Full Path
MLTITHRD.EXE	FFC30445	8 (Normal)	6	32-Bit	H:\0004\PROG\MLTITHRD.01\MLTITHRD.EXE
PVIEW95.EXE	FFC3670D	8 (Normal)	1	32-Bit	E:\DEVSTUDIO\VC\BIN\WIN95\PVIEW95.EXE
WINOLDAP	FFC2A631	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINDA386.MOD
WINOLDAP	FFC2D4CD	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\WINDA386.MOD
WINWORD.EXE	FFC2E431	8 (Normal)	1	32-Bit	D:\MSOFFICE\WINWORD\WINWORD.EXE
MSDEV.EXE	FFFD770D	8 (Normal)	6	32-Bit	E:\DEVSTUDIO\SHARED\IDE\BIN\MSDEV.EXE
PPSHELL.EXE	FFFC8E41	8 (Normal)	1	32-Bit	C:\PPENSE\WIN32\PPSHELL.EXE
ETENSRV	FFFD7CF1	8 (Normal)	1	16-Bit	D:\Program Files\ET2K\BOX95\ETENSRV.EXE
SAGE.EXE	FFFCFA3D	8 (Normal)	2	32-Bit	D:\WIN95\SYSTEM\SAGE.EXE
SYSTRAY.EXE	FFFCF72D	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\SYSTRAY.EXE
INTERNAT.EXE	FFCD6BD	8 (Normal)	1	32-Bit	D:\WIN95\SYSTEM\INTERNAT.EXE
EXPLORER.EXE	FFFC0CF1	8 (Normal)	3	32-Bit	D:\WIN95\EXPLORER.EXE
MMTASK	FFFC60B5	8 (Normal)	1	16-Bit	D:\WIN95\SYSTEM\mmtask.tsk

TID	Owning PID	Thread Priority
FFC31669	FFC30445	6 (Lowest)
FFC314B1	FFC30445	7 (Below Normal)
FFC312D9	FFC30445	8 (Normal)
FFC30F01	FFC30445	9 (Above Norm...)
FFC37CB5	FFC30445	10 (Highest)
FFC308AD	FFC30445	10 (Highest)

圖 1-10 利用 Process Viewer (PVIEW95.EXE) 觀察 MLTITHRD.EXE 的執行，的確發現有六個執行緒，其中包括一個主執行緒。

第一篇 勿在浮砂築高台

第 2 頁

C++ 的性質

C++ 是一種扭轉程式員思維模式的語言。
一個人思維模式的扭轉，不可能輕而易舉一蹴而成。

邇來「物件導向」一詞席捲了整個軟體界。物件導向程式設計（Object Oriented Programming）其實是一種觀念，用什麼語言實現它都可以。但，當然，物件導向程式語言（Object Oriented Programming Language）是專門為物件導向觀念而發展出來的，以之完成物件導向的封裝、繼承、多型等特性自是最為便利。

C++ 是最重要的物件導向語言，因為它站在 C 語言的肩膀上，而 C 語言擁有絕對優勢的使用者。C++ 並非純然的物件導向程式語言，不過有時候混血並不是壞事，純種也不見得就多好。

所謂純物件導向語言，是指不管什麼東西，都應該存在於物件之中。JAVA 和 Small Talk 都是純物件導向語言。

如果你是 C++ 的初學者，本章不適合你（事實上整本書都不適合你），你的當務之急是去買一本 C++ 專書。一位專精 Basic 和 Assembly 語言的朋友問我，有沒有可能不會 C++ 而學會 MFC？答案是當然沒有可能。

如果你對 C++ 一知半解，語法大約都懂了，語意大約都不懂，本章是我能夠給你的最好禮物。我將從類別與物件的關係開始，逐步解釋封裝、繼承、多型、虛擬函式、動態繫結。不只解釋其操作方式，更要點出其意義與應用，也就是，為什麼需要這些性質。

C++ 語言範圍何其廣大，這一章的主題挑選完全是以 MFC Programming 所需技術為前提。下一章，我們就把這裡學到的 C++ 技術和 OO 觀念應用到 application framework 的模擬上，那是一個 DOS 程式，不牽扯 Windows。

類別及其成員 - 談封裝 (encapsulation)

讓我們把世界看成是一個由物件 (object) 所組成的大環境。物件是什麼？白一點說，「東西」是也！任何實際的物體你都可以說它是物件。為了描述物件，我們應該先把物件的屬性描述出來。好，給「物件的屬性」一個比較學術的名詞，就是「類別」(class)。

物件的屬性有兩大成員，一是資料，一是行為。在物件導向的術語中，前者常被稱為 property (Java 語言則稱之為 field)，後者常被稱為 method。另有一雙比較像程式設計領域的術語，名為 member variable (或 data member) 和 member function。為求統一，本書使用第二組術語，也就是 member variable (成員變數) 和 member function (成員函式)。一般而言，成員變數通常由成員函式處理之。

如果我以 CSquare 代表「四方形」這種類別，四方形有 color，四方形可以 display。好，color 就是一種成員變數，display 就是一種成員函式：

```
CSquare square;    // 宣告 square 是一個四方形。
square.color = RED; // 設定成員變數。RED 代表一個顏色值。
square.display();   // 呼叫成員函式。
```

下面是 C++ 語言對於 CSquare 的描述：

```
class CSquare // 常常我們以 C 作為類別名稱的開頭
{
private:
    int m_color; // 通常我們以 m_ 作為成員變數的名稱開頭

public:
    void display() { ... }
    void setcolor(int color) { m_color = color; }
};
```

成員變數可以只在類別內被處理，也可以開放給外界處理。以資料封裝的目的而言，自

然是前者較為妥當，但有時候也不得不開放。為此，C++ 提供了 *private*、*public* 和 *protected* 三種修飾詞。一般而言成員變數儘量宣告為 *private*，成員函式則通常宣告為 *public*。上例的 *m_color* 既然宣告為 *private*，我們勢必得準備一個成員函式 *setcolor*，供外界設定顏色用。

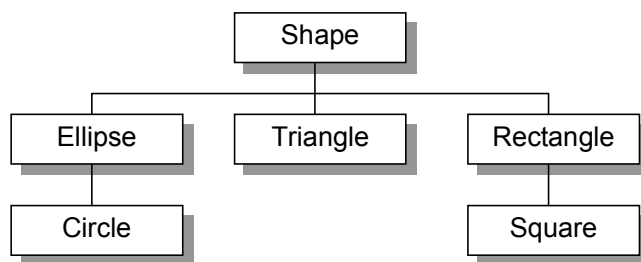
把資料宣告為 *private*，不允許外界隨意存取，只能透過特定的介面來操作，這就是物件導向的封裝（*encapsulation*）特性。

基礎類別與衍生類別：談繼承 (Inheritance)

其他語言欲完成封裝性質，並不太難。以 C 為例，在結構（*struct*）之中放置資料，以及處理資料的函式的指標（*function pointer*），就可得到某種程度的封裝精神。

C++ 神秘而特有的性質其實在於繼承。

矩形是形，橢圓形是形，三角形也是形。蒼蠅是昆蟲，蜜蜂是昆蟲，螞蟥也是昆蟲。是的，人類習慣把相同的性質抽取出來，成立一個基礎類別（*base class*），再從中衍化出衍生類別（*derived class*）。所以，關於形狀，我們就有了這樣的類別階層：



注意：衍生類別與基礎類別的關係是 “IsKindOf” 的關係。也就是說，
 Circle 「是一種」 Ellipse，Ellipse 「是一種」 Shape；
 Square 「是一種」 Rectangle，Rectangle 「是一種」 Shape。

```
#0001 class CShape          // 形狀
#0002 {
#0003 private:
#0004     int m_color;
#0005
#0006 public:
#0007     void setcolor(int color) { m_color = color; }
#0008 };
#0009
#0010 class CRect : public CShape    // 矩形是一種形狀
#0011 {                                // 它會繼承m_color 和 setcolor()
#0012 public:
#0013     void display() { ... }
#0014 };
#0015
#0016 class CEllipse : public CShape  // 橢圓形是一種形狀
#0017 {                                // 它會繼承m_color 和 setcolor()
#0018 public:
#0019     void display() { ... }
#0020 };
#0021
#0022 class CTriangle : public CShape // 三角形是一種形狀
#0023 {                                // 它會繼承m_color 和 setcolor()
#0024 public:
#0025     void display() { ... }
#0026 };
#0027
#0028 class CSquare : public CRect     // 四方形是一種矩形
#0029 {
#0030 public:
#0031     void display() { ... }
#0032 };
#0033
#0034 class CCircle : public CEllipse // 圓形是一種橢圓形
#0035 {
#0036 public:
#0037     void display() { ... }
#0038 };
```

於是你可以這麼動作：

```
CSquare square;
CRect  rect1, rect2;
CCircle circle;

square.setcolor(1); // 令 square.m_color = 1;
```

```

square.display();    // 呼叫 CSquare::display

rect1.setcolor(2);    // 於是 rect1.m_color = 2
rect1.display();      // 呼叫 CRect::display

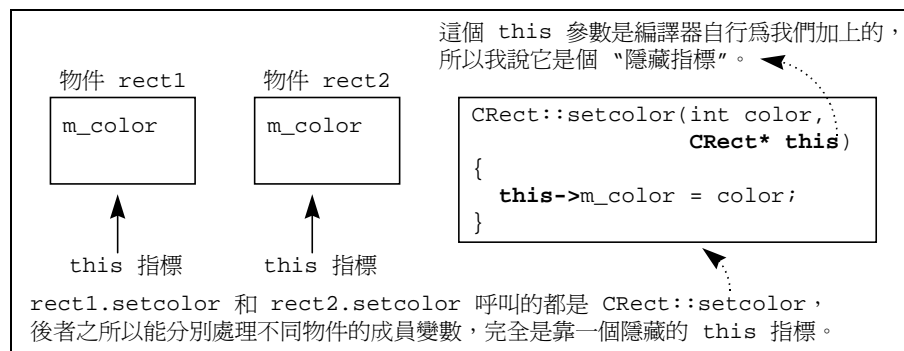
rect2.setcolor(3);    // 於是 rect2.m_color = 3
rect2.display();      // 呼叫 CRect::display

circle.setcolor(4);   // 於是 circle.m_color = 4
circle.display();     // 呼叫 CCircle::display

```

注意以下這些事實與問題：

1. 所有類別都由 *CShape* 衍生下來，所以它們都自然而然繼承了 *CShape* 的成員，包括變數和函式。也就是說，所有的形狀類別都「暗自」具備了 *m_color* 變數和 *setcolor* 函式。我所謂暗自（implicit），意思是無法從各衍生類別的宣告中直接看出來。
2. 兩個矩形物件 *rect1* 和 *rect2* 各有自己的 *m_color*，但關於 *setcolor* 函式卻是共用相同的 *CRect::setcolor*（其實更應該說是 *CShape::setcolor*）。我用這張圖表示其間的關係：



讓我替你問一個問題：同一個函式如何處理不同的資料？為什麼 *rect1.setcolor* 和 *rect2.setcolor* 明明都是呼叫 *CRect::setcolor*（其實也就是 *CShape::setcolor*），卻能夠有條不紊地分別處理 *rect1.m_color* 和 *rect2.m_color*？答案在於所謂的 *this* 指標。下一節我就會提到它。

3. 既然所有類別都有 *display* 動作，把它提昇到老祖宗 *CShape* 去，然後再繼承之，好嗎？不好，因為 *display* 函式應該因不同的形狀而動作不同。
4. 如果 *display* 不能提昇到基礎類別去，我們就不能夠以一個 *for* 迴路或 *while* 迴路乾淨漂亮地完成下列動作（此種動作模式在物件導向程式方法中重要無比）：

```
CShape shapes[5];

... // 令 5 個 shapes 各為矩形、四方形、橢圓形、圓形、三角形
for (int i=0; i<5; i++)
{
    shapes[i].display;
}
```

5. *Shape* 只是一種抽象意念，世界上並沒有「形狀」這種東西！你可以在一個 C++ 程式中做以下動作，但是不符合生活法則：

```
CShape shape;          // 世界上沒有「形狀」這種東西，
shape.setcolor();       // 所以這個動作就有點奇怪。
```

這同時也說出了第三點的另一個否定理由：按理你不能夠把一個抽象的「形狀」顯示出來，不是嗎？！

如果語法允許你產生一個不應該有的抽象物件，或如果語法不支援「把所有形狀（不管什麼形狀）都 *display* 出來」的**一般化動作**，這就是個失敗的語言。C++ 是成功的，自然有它的整治方式。

記住，「物件導向」觀念是描繪現實世界用的。所以，你可以以真實生活中的經驗去思考程式設計的邏輯。

this 指標

剛剛我才說過，兩個矩形物件 *rect1* 和 *rect2* 各有自己的 *m_color* 成員變數，但 *rect1.setcolor* 和 *rect2.setcolor* 卻都通往唯一的 *CRect::setcolor* 成員函式。那麼 *CRect::setcolor* 如何處理不同物件中的 *m_color*？答案是：成員函式有一個隱藏參數，名為 *this* 指標。當你呼叫：

```
rect1.setcolor(2); // rect1 是 CRect 物件
rect2.setcolor(3); // rect2 是 CRect 物件
```

編譯器實際上為你做出來的碼是：

```
CRect::setcolor(2, (CRect*)&rect1);
CRect::setcolor(3, (CRect*)&rect2);
```

不過，由於 *CRect* 本身並沒有宣告 *setcolor*，它是從 *CShape* 繼承來的，所以編譯器實際上產生的碼是：

```
CShape::setcolor(2, (CRect*)&rect1);
CShape::setcolor(3, (CRect*)&rect2);
```

多出來的參數，就是所謂的 *this* 指標。至於類別之中，成員函式的定義：

```
class CShape
{
...
public:
    void setcolor(int color) { m_color = color; }
};
```

被編譯器整治過後，其實是：

```
class CShape
{
...
public:
    void setcolor(int color, (CShape*)this) { this->m_color = color; }
};
```

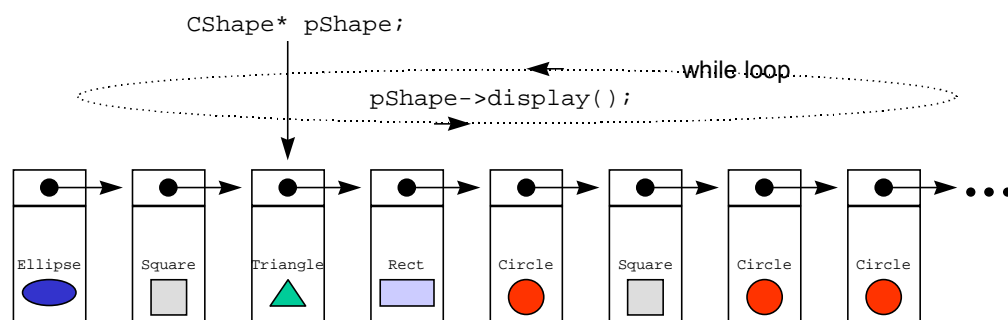
我們撥開了第一道疑雲。

虛擬函式與多型 (Polymorphism)

我曾經說過，前一個例子沒有辦法完成這樣的動作：

```
CShape shapes[5];
... // 令 5 個 shapes 各為矩形、四方形、橢圓形、圓形、三角形
for (int i=0; i<5; i++)
{
    shapes[i].display;
}
```

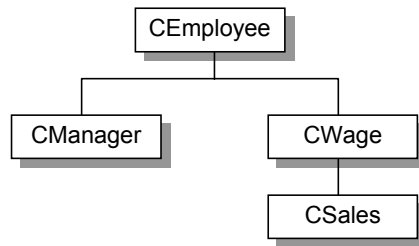
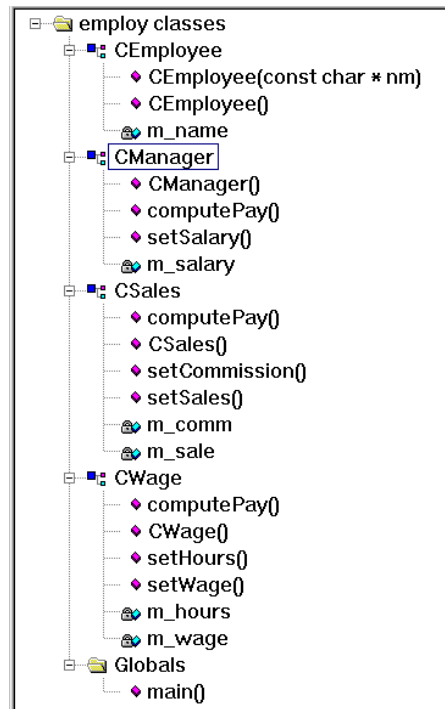
可是這種所謂物件操作的一般化動作在 application framework 中非常重要。作為 framework 設計者的我，總是希望能夠準備一個 *display* 函式，給我的使用者呼叫；不管他根據我的這一大堆形狀類別衍生出其他什麼奇形怪狀的類別，只要他想 *display*，像下面那麼做就行。



為了支援這種能力，C++ 提供了所謂的虛擬函式（virtual function）。

虛擬 + 函式 ?! 聽起來很恐怖的樣子。如果你了解汽車的離合器踩下去代表汽車空檔，空檔表示失去引擎本身的牽制力，你就會了解「高速行駛間煞車絕不能踩離合器」的道理並矢志遵行。好，如果你真的了解為什麼需要虛擬函式以及什麼情況下需要它，你就能夠掌握它的靈魂與內涵，真正了解它的設計原理，並且發現認為它非常人性。並且，真正知道怎麼用它。

讓我用另一個例子來展開我的說明。這個範例靈感得自 Visual C++ 手冊之一：
Introduction to C++。假設你的類別種類如下：



本圖以 Visual C++ 之「Class Info 視窗」獲得

程式碼實作如下：

```

#0001 #include <string.h>
#0002
#0003 //-----
#0004 class CEmployee // 職員
#0005 {
#0006 private:
#0007     char m_name[30];
#0008
#0009 public:
#0010     CEmployee();
#0011     CEmployee(const char* nm) { strcpy(m_name, nm); }
#0012 };
  
```



```
#0013 //-----
#0014 class CWage : public CEmployee    // 時薪職員是一種職員
#0015 {
#0016 private :
#0017     float m_wage;
#0018     float m_hours;
#0019
#0020 public :
#0021     CWage(const char* nm) : CEmployee(nm) { m_wage = 250.0; m_hours = 40.0; }
#0022     void setWage(float wg) { m_wage = wg; }
#0023     void setHours(float hrs) { m_hours = hrs; }
#0024     float computePay();
#0025 };
#0026 //-----
#0027 class CSales : public CWage    // 銷售員是一種時薪職員
#0028 {
#0029 private :
#0030     float m_comm;
#0031     float m_sale;
#0032
#0033 public :
#0034     CSales(const char* nm) : CWage(nm) { m_comm = m_sale = 0.0; }
#0035     void setCommission(float comm) { m_comm = comm; }
#0036     void setSales(float sale) { m_sale = sale; }
#0037     float computePay();
#0038 };
#0039 //-----
#0040 class CManager : public CEmployee    // 經理也是一種職員
#0041 {
#0042 private :
#0043     float m_salary;
#0044 public :
#0045     CManager(const char* nm) : CEmployee(nm) { m_salary = 15000.0; }
#0046     void setSalary(float salary) { m_salary = salary; }
#0047     float computePay();
#0048 };
#0049 //-----
#0050 void main()
#0051 {
#0052     CManager aManager("陳美靜");
#0053     CSales aSales("侯俊傑");
#0054     CWage aWager("曾銘源");
#0055 }
#0056 //-----
#0057 // 雖然各類別的 computePay 函式都沒有定義，但因為程式也沒有呼叫之，所以無妨。
```

如此一來，*CWage* 繼承了 *CEmployee* 所有的成員（包括資料與函式），*CSales* 又繼承了 *CWage* 所有的成員（包括資料與函式）。在意義上，相當於 *CSales* 擁有資料如下：

```
// private data of CEmployee
char m_name[30];

// private data of CWage
float m_wage;
float m_hours;

// private data of CSales
float m_comm;
float m_sale;
```

以及函式如下：

```
void setWage(float wg);
void setHours(float hrs);
void setCommission(float comm);
void setSale(float sales);
void computePay();
```

從 Visual C++ 的除錯器中，我們可以看到，上例的 *main* 執行之後，程式擁有三個物件，內容（我是指成員變數）分別為：



從薪水說起

虛擬函式的故事要從薪水的計算說起。根據不同職員的計薪方式，我設計 *computePay* 函式如下：

```
float CManager::computePay()
{
    return m_salary; // 經理以「固定週薪」計薪。
}

float CWage::computePay()
{
    return (m_wage * m_hours); // 時薪職員以「鐘點費 * 每週工時」計薪。
}

float CSales::computePay()
{
    // 銷售員以「鐘點費 * 每週工時」再加上「佣金 * 銷售額」計薪。
    return (m_wage * m_hours + m_comm * m_sale); // 語法錯誤。
}
```

但是 *CSales* 物件不能夠直接取用 *CWage* 的 *m_wage* 和 *m_hours*，因為它們是 *private* 成員變數。所以是不是應該改為這樣：

```
float CSales::computePay()
{
    return computePay() + m_comm * m_sale;
}
```

這也不好，我們應該指明函式中所呼叫的 *computePay* 究歸誰屬 -- 編譯器沒有厲害到能夠自行判斷而保證不出錯。正確寫法應該是：

```
float CSales::computePay()
{
    return CWage::computePay() + m_comm * m_sale;
}
```

這就合乎邏輯了：銷售員是一般職員的一種，他的薪水應該是以時薪職員的計薪方式作為底薪，再加上額外的銷售佣金。我們看看實際情況，如果有一個銷售員：

```
CSales aSales("侯俊傑");
```

那麼侯俊傑的底薪應該是：

```
aSales.CWage::computePay(); // 這是銷售員的底薪。注意語法。
```

而侯俊傑的全薪應該是：

```
aSales.computePay(); // 這是銷售員的全薪
```

結論是：要呼叫父類別的函式，你必須使用 **scope resolution operator** (::) 明白指出。

接下來我要觸及物件型態的轉換，這關係到指標的運用，更直接關係到為什麼需要虛擬函式。了解它，對於 **application framework** 如 **MFC** 者的運用十分十分重要。

假設我們有兩個物件：

```
CWage aWager;  
CSales aSales("侯俊傑");
```

銷售員是時薪職員之一，因此這樣做是合理的：

```
aWager = aSales; // 合理，銷售員必定是時薪職員。
```

這樣就不合理：

```
aSales = aWager; // 錯誤，時薪職員未必是銷售員。
```

如果你一定要轉換，必須使用指標，並且明顯地做型別轉換 (**cast**) 動作：

```
CWage* pWager;  
CSales* pSales;  
CSales aSales("侯俊傑");  
  
pWager = &aSales; // 把一個「基礎類別指標」指向衍生類別之物件，合理且自然。  
pSales = (CSales *)pWager; // 強迫轉型。語法上可以，但不符合現實生活。
```

真實世界中某些時候我們會以「一種動物」來總稱貓啊、狗啊、兔子猴子等等。為了某種便利（這個便利稍後即可看到），我們也會想以「一個通用的指標」表示所有可能的職員型態。無論如何，銷售員、時薪職員、經理，都是職員，所以下面動作合情合理：

```
CEmployee* pEmployee;
CWage    aWager("曾銘源");
CSales    aSales("侯俊傑");
CManager aManager("陳美靜");

pEmployee = &aWager;    // 合理，因為時薪職員必是職員
pEmployee = &aSales;    // 合理，因為銷售員必是職員
pEmployee = &aManager;  // 合理，因為經理必是職員
```

也就是說，你可以把一個「職員指標」指向任何一種職員。這帶來的好處是程式設計的巨大彈性，譬如說你設計一個串列（linked list），各個元素都是職員（哪一種職員都可以），你的 *add* 函式可能因此希望有一個「職員指標」作為參數：

```
add(CEmployee* pEmp); // pEmp 可以指向任何一種職員
```

晴天霹靂

我們漸漸接觸問題的核心。上述 C++ 性質使真實生活經驗的確在電腦語言中模擬了出來，但是萬里無雲的日子裡卻出現了一個晴天霹靂：如果你以一個「基礎類別之指標」指向一個「衍生類別之物件」，那麼經由此指標，你就只能夠呼叫基礎類別（而不是衍生類別）所定義的函式。因此：

```
CSales aSales("侯俊傑");
CSales* pSales;
CWage* pWager;

pSales = &aSales;
pWager = &aSales; // 以「基礎類別之指標」指向「衍生類別之物件」

pWager->setSales(800.0); // 錯誤（編譯器會檢測出來），
                        // 因為 CWage 並沒有定義 setSales 函式。
pSales->setSales(800.0); // 正確，呼叫 CSales::setSales 函式。
```

雖然 *pSales* 和 *pWager* 指向同一個物件，但卻因指標的原始型態而使兩者之間有了差異。

延續此例，我們看另一種情況：

```
pWager->computePay(); // 呼叫 CWage::computePay()
pSales->computePay(); // 呼叫 CSales::computePay()
```

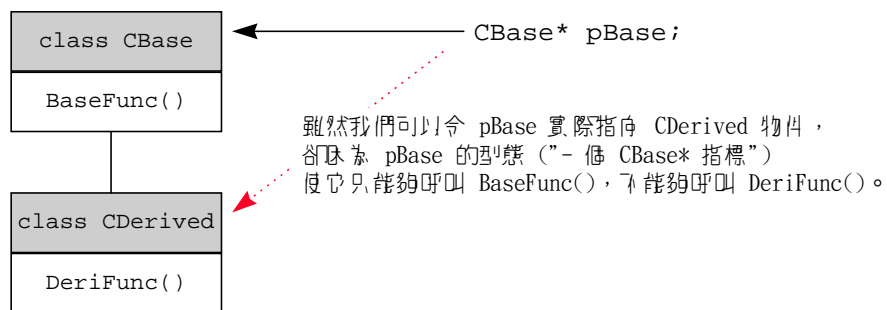
雖然 *pSales* 和 *pWager* 實際上都指向 *CSales* 物件，但是兩者呼叫的 *computePay* 卻不

相同。到底呼叫到哪個函式，必須視指標的原始型態而定，與指標實際所指之物件無關。

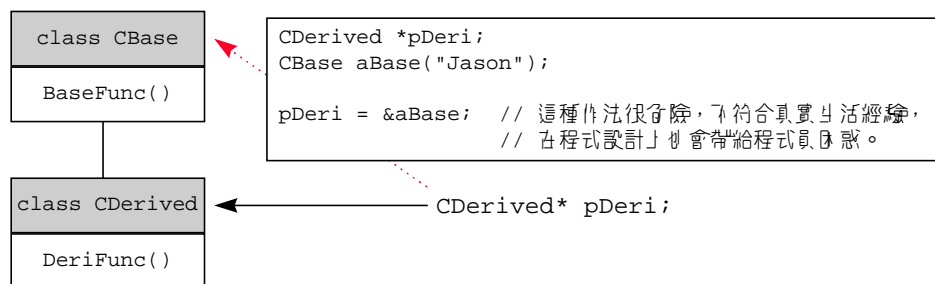
三個結論

我們得到了三個結論：

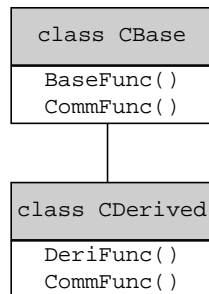
1. 如果你以一個「基礎類別之指標」指向「衍生類別之物件」，那麼經由該指標你只能夠呼叫基礎類別所定義的函式。



2. 如果你以一個「衍生類別之指標」指向一個「基礎類別之物件」，你必須先做明顯的轉型動作（explicit cast）。這種作法很危險，不符合真實生活經驗，在程式設計上也會帶給程式員困惑。



3. 如果基礎類別和衍生類別都定義了「相同名稱之成員函式」，那麼透過物件指標呼叫成員函式時，到底呼叫到哪一個函式，必須視該指標的原始型態而定，而不是視指標實際所指之物件的型態而定。這與第 1 點其實意義相通。



```

CBase* pBase;
CDerived* pDeri;
    
```

不論你把這兩個指標指向何方，由於它們的原始型態，使它們在呼叫同名的 CommFunc() 時有暫無可改變的宿命：

- pBase->CommFunc() 永遠是指 CBase::CommFunc
- pDeri->CommFunc() 永遠是指 CDerived::CommFunc

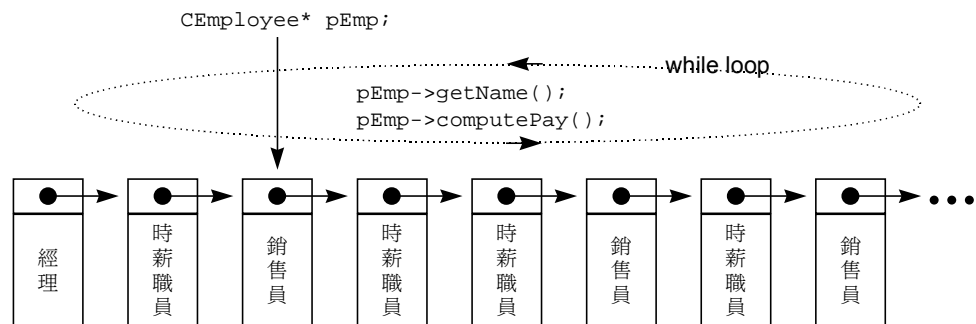
得到這些結論後，看看什麼事情會困擾我們。前面我曾提到一個由職員組成的串列，如果我想寫一個 *printNames* 函式走訪串列中的每一個元素並印出職員的名字，我們可以在 *CEmployee* (最基礎類別) 中多加一個 *getName* 函式，然後再設計一個 *while* 迴路如下：

```

int count = 0;
CEmployee* pEmp;
...
while (pEmp = anIter.getNext())
{
    count++;
    cout << count << ' ' << pEmp->getName() << endl;
}
    
```

你可以把 *anIter.getNext* 想像是一個可以走訪串列的函式，它傳回 *CEmployee**，也因此每一次獲得的指標才可以呼叫定義於 *CEmployee* 中的 *getName*。

計算薪資



但是，由於函式的呼叫是依賴指標的原始型態而不管它實際上指向何方（何種物件），因此如果上述 `while` 迴路中呼叫的是 `pEmp->computePay`，那麼 `while` 迴路所執行的將總是相同的運算，也就是 `CEmployee::computePay`，這就糟了（銷售員領到經理的薪水還不糟嗎）。更糟的是，我們根本沒有定義 `CEmployee::computePay`，因為 `CEmployee` 只是個抽象概念（一個抽象類別）。指標必須落實到具象型態上如 `CWage` 或 `CManager` 或 `CSales`，才有薪資計算公式。

虛擬函式與一般化

我想你可以體會，上述的 `while` 迴路其實就是把動作「一般化」。「一般化」之所以重要，在於它可以把現在的、未來的情況統統納入考量。將來即使有另一種名曰「顧問」的職員，上述計薪迴路應該仍然能夠正常運作。當然啦，「顧問」的 `computePay` 必須設計好。

「一般化」是如此重要，解決上述問題因此也就迫切起來。我們需要的是什麼呢？是能夠「依舊以 `CEmpolyee` 指標代表每一種職員」，而又能夠在「實際指向不同種類之職員」時，「呼叫到不同版本（不同類別中）之 `computePay`」這種能力。

這種性質就是多型（polymorphism），靠虛擬函式來完成。

再次看看那張計薪迴路圖：

- 當 `pEmp` 指向經理，我希望 `pEmp->computePay` 是經理的薪水計算式，也就是 `CManager::computePay`。
- 當 `pEmp` 指向銷售員，我希望 `pEmp->computePay` 是銷售員的薪水計算式，也就是 `CSales::computePay`。
- 當 `pEmp` 指向時薪職員，我希望 `pEmp->computePay` 是時薪職員的薪水計算式，也就是 `CWage::computePay`。

虛擬函式正是爲了對「如果你以一個基礎類別之指標指向一個衍生類別之物件，那麼透過該指標你就只能夠呼叫基礎類別所定義之成員函式」這條規則反其道而行的設計。

不必設計複雜的串列函式如 *add* 或 *getNext* 才能驗證這件事，我們看看下面這個簡單例子。如果我把職員一例中所有四個類別的 *computePay* 函式前面都加上 *virtual* 保留字，使它們成為虛擬函式，那麼：

```
CEmployee* pEmp;
CWage      aWager("曾銘源");
CSales     aSales("侯俊傑");
CManager   aManager("陳美靜");

pEmp = &aWager;
cout << pEmp->computePay(); // 呼叫的是 CWage::computePay
pEmp = &aSales;
cout << pEmp->computePay(); // 呼叫的是 CSales::computePay
pEmp = &aManager;
cout << pEmp->computePay(); // 呼叫的是 CManager::computePay
```

現在重新回到 *Shape* 例子，我打算讓 *display* 成為虛擬函式：

```
#0001 #include <iostream.h>
#0002 class CShape
#0003 {
#0004     public:
#0005     virtual void display() { cout << "Shape \n"; }
#0006 };
#0007 //-----
#0008 class CEllipse : public CShape
#0009 {
#0010     public:
#0011     virtual void display() { cout << "Ellipse \n"; }
#0012 };
#0013 //-----
#0014 class CCircle : public CEllipse
#0015 {
#0016     public:
#0017     virtual void display() { cout << "Circle \n"; }
#0018 };
#0019 //-----
#0020 class CTriangle : public CShape
#0021 {
#0022     public:
#0023     virtual void display() { cout << "Triangle \n"; }
#0024 };
```

```
#0025 //-----
#0026 class CRect : public CShape
#0027 {
#0028     public:
#0029     virtual void display() { cout << "Rectangle \n"; }
#0030 };
#0031 //-----
#0032 class CSquare : public CRect
#0033 {
#0034     public:
#0035     virtual void display() { cout << "Square \n"; }
#0036 };
#0037 //-----
#0038 void main()
#0039 {
#0040     CShape      aShape;
#0041     CEllipse     aEllipse;
#0042     CCircle      aCircle;
#0043     CTriangle    aTriangle;
#0044     CRect        aRect;
#0045     CSquare      aSquare;
#0046     CShape* pShape[6] = { &aShape,
#0047                           &aEllipse,
#0048                           &aCircle,
#0049                           &aTriangle,
#0050                           &aRect,
#0051                           &aSquare };
#0052
#0053     for (int i=0; i< 6; i++)
#0054         pShape[i]->display();
#0055 }
#0056 //-----
```

得到的結果是：

```
Shape
Ellipse
Circle
Triangle
Rectangle
Square
```

如果把所有類別中的 `virtual` 保留字拿掉，執行結果變成：

```
Shape
Shape
Shape
Shape
Shape
Shape
```

綜合 `Employee` 和 `Shape` 兩例，第一個例子是：

```
pEmp = &aWager;
cout << pEmp->computePay();
pEmp = &aSales;
cout << pEmp->computePay();
pEmp = &aBoss;
cout << pEmp->computePay();
```

這三行程式碼完全相同

第二個例子是：

```
CShape* pShape[6];
for (int i=0; i< 6; i++)
    pShape[i]->display(); // 此行程式碼執行了 6 次。
```

我們看到了一種奇特現象：程式碼完全一樣（因為一般化了），執行結果卻不相同。這就是虛擬函式的妙用。

如果沒有虛擬函式這種東西，你還是可以使用 `scope resolution operator` (`::`) 明白指出呼叫哪一個函式，但程式就不再那麼優雅與彈性了。

從操作型定義來看，什麼是虛擬函式呢？如果你預期衍生類別有可能重新定義某一個成員函式，那麼你就在基礎類別中把此函式設為 `virtual`。MFC 有兩個十分重要的虛擬函式：與 `document` 有關的 `Serialize` 函式和與 `view` 有關的 `OnDraw` 函式。你應該在自己的 `CMyDoc` 和 `CMyView` 中改寫這兩個虛擬函式。

多型 (Polymorphism)

你看，我們以相同的指令卻喚起了不同的函式，這種性質稱為 Polymorphism，意思是 "the ability to assume many forms"（多型）。編譯器無法在編譯時期判斷 *pEmp->computePay* 到底是呼叫哪一個函式，必須在執行時期才能評估之，這稱為後期繫結 *late binding* 或動態繫結 *dynamic binding*。至於 C 函式或 C++ 的 *non-virtual* 函式，在編譯時期就轉換為一個固定位址的呼叫了，這稱為前期繫結 *early binding* 或靜態繫結 *static binding*。

Polymorphism 的目的，就是要讓處理「基礎類別之物件」的程式碼，能夠完全透通地繼續適當處理「衍生類別之物件」。

可以說，虛擬函式是了解多型 (Polymorphism) 以及動態繫結的關鍵。同時，它也是了解如何使用 MFC 的關鍵。

讓我再次提示你，當你設計一套類別，你並不知道使用者會衍生什麼新的子類別出來。如果動物世界中出現了新品種名曰雅虎，類別使用者勢必在 *CAnimal* 之下衍生一個 *CYahoo*。饒是如此，身為基礎類別設計者的你，可以利用虛擬函式的特性，將所有動物必定會有的行為（例如哮叫 *roar*），規劃為虛擬函式，並且規劃一些一般化動作（例如「讓每一種動物發出一聲哮叫」）。那麼，雖然，你在設計基礎類別以及這個一般化動作時，無法掌握使用者自行衍生的子類別，但只要他改寫了 *roar* 這個虛擬函式，你的一般化物件操作動作自然就可以呼叫到該函式。

再次回到前述的 *Shape* 例子。我們說 *CShape* 是抽象的，所以它根本不該有 *display* 這個動作。但為了在各具象衍生類別中繪圖，我們又不得不在基礎類別 *CShape* 加上 *display* 虛擬函式。你可以定義它什麼也不做（空函式）：

```
class CShape
{
public:
    virtual void display() { }
};
```

或只是給個訊息：

```
class CShape
{
public:
    virtual void display() { cout << "Shape \n"; }
};
```

這兩種作法都不高明，因為這個函式根本就不應該被呼叫（*CShape* 是抽象的），我們根本就不應該定義它。不定義但又必須保留一塊空間（*spaceholder*）給它，於是 C++ 提供了所謂的純虛擬函式：

```
class CShape
{
public:
    virtual void display() = 0; // 注意 "= 0"
};
```

純虛擬函式不需定義其實際動作，它的存在只是爲了在衍生類別中被重新定義，只是爲了提供一個多型介面。只要是擁有純虛擬函式的類別，就是一種抽象類別，它是不能夠被具象化（*instantiate*）的，也就是說，你不能根據它產生一個物件（你怎能說一種形狀爲 'Shape' 的物體呢）。如果硬要強渡關山，會換來這樣的編譯訊息：

```
error : illegal attempt to instantiate abstract class.
```

關於抽象類別，我還有一點補充。*CCircle* 繼承了 *CShape* 之後，如果沒有改寫 *CShape* 中的純虛擬函式，那麼 *CCircle* 本身也就成爲一個擁有純虛擬函式的類別，於是它也是一個抽象類別。

是對虛擬函式做結論的時候了：

- 如果你期望衍生類別重新定義一個成員函式，那麼你應該在基礎類別中把此函式設爲 *virtual*。
- 以單一指令喚起不同函式，這種性質稱爲 *Polymorphism*，意思是 "the ability to assume many forms"，也就是多型。
- 虛擬函式是 C++ 語言的 *Polymorphism* 性質以及動態繫結的關鍵。

- 既然抽象類別中的虛擬函式不打算被呼叫，我們就不應該定義它，應該把它設為純虛擬函式（在函式宣告之後加上 "=0" 即可）。
- 我們可以說，擁有純虛擬函式者為抽象類別（**abstract class**），以別於所謂的具象類別（**concrete class**）。
- 抽象類別不能產生出物件實體，但是我們可以擁有指向抽象類別之指標，以便於操作抽象類別的各個衍生類別。
- 虛擬函式衍生下去仍為虛擬函式，而且可以省略 *virtual* 關鍵字。

類別與物件大解剖

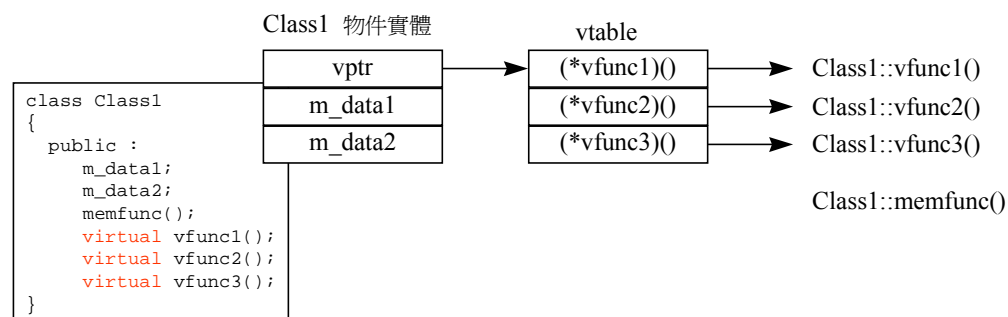
你一定很想知道虛擬函式是怎麼做出來的，對不對？

如果能夠了解 C++ 編譯器對於虛擬函式的實現方式，我們就能夠知道為什麼虛擬函式可以做到動態繫結。

為了達到動態繫結（後期繫結）的目的，C++ 編譯器透過某個表格，在執行時期「間接」呼叫實際上欲繫結的函式（注意「間接」這個字眼）。這樣的表格稱為虛擬函式表（常被稱為 **vtable**）。每一個「內含虛擬函式的類別」，編譯器都會為它做出一個虛擬函式表，表中的每一筆元素都指向一個虛擬函式的位址。此外，編譯器當然也會為類別加上一項成員變數，是一個指向該虛擬函式表的指標（常被稱為 **vptr**）。舉個例：

```
class Class1 {
public :
    data1;
    data2;
    memfunc();
    virtual vfunc1();
    virtual vfunc2();
    virtual vfunc3();
};
```

Class1 物件實體在記憶體中佔據這樣的空間：



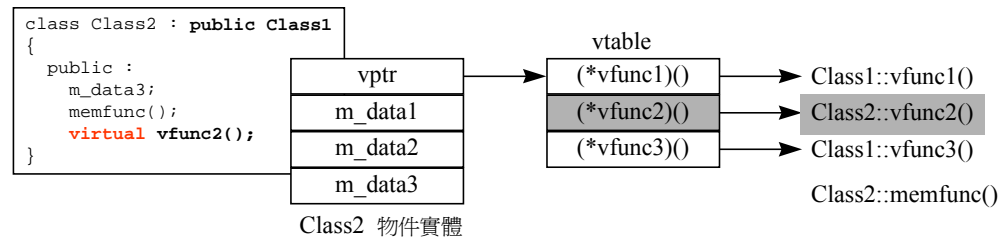
C++ 類別的成員函式，你可以想像就是 C 語言中的函式。它只是被編譯器改過名稱，並增加一個參數（this 指標），因而可以處理呼叫者（C++ 物件）中的成員變數。所以，你並沒有在 Class1 物件的記憶體區塊中看到任何與成員函式有關的任何東西。

每一個由此類別衍生出來的物件，都有這麼一個 vptr。當我們透過這個物件呼叫虛擬函式，事實上是透過 vptr 找到虛擬函式表，再找出虛擬函式的真正位址。

奧妙在於這個虛擬函式表以及這種間接呼叫方式。虛擬函式表的內容是依據類別中的虛擬函式宣告次序，一一填入函式指標。衍生類別會繼承基礎類別的虛擬函式表（以及所有其他可以繼承的成員），當我們在衍生類別中改寫虛擬函式時，虛擬函式表就受了影響：表中元素所指的函式位址將不再是基礎類別的函式位址，而是衍生類別的函式位址。看看這個例子：

```

class Class2 : public Class1 {
public:
    data3;
    memfunc();
    virtual vfunc2();
};
    
```



於是，一個「指向 *Class1* 所生物件」的指標，所呼叫的 *vfunc2* 就是 *Class1::vfunc2*，而一個「指向 *Class2* 所生物件」的指標，所呼叫的 *vfunc2* 就是 *Class2::vfunc2*。

動態繫結機制，在執行時期，根據虛擬函式表，做出了正確的選擇。

我們解開了第二道神秘。

口說無憑，何不看看實際。觀其位址，物焉度哉，下面是一個測試程式：

```

#0001 #include <iostream.h>
#0002 #include <stdio.h>
#0003
#0004 class ClassA
#0005 {
#0006 public:
#0007     int m_data1;
#0008     int m_data2;
#0009     void func1() { }
#0010     void func2() { }
#0011     virtual void vfunc1() { }
#0012     virtual void vfunc2() { }
#0013 };
#0014
#0015 class ClassB : public ClassA
#0016 {
#0017 public:
#0018     int m_data3;
#0019     void func2() { }
#0020     virtual void vfunc1() { }
#0021 };
#0022
#0023 class ClassC : public ClassB
#0024 {

```

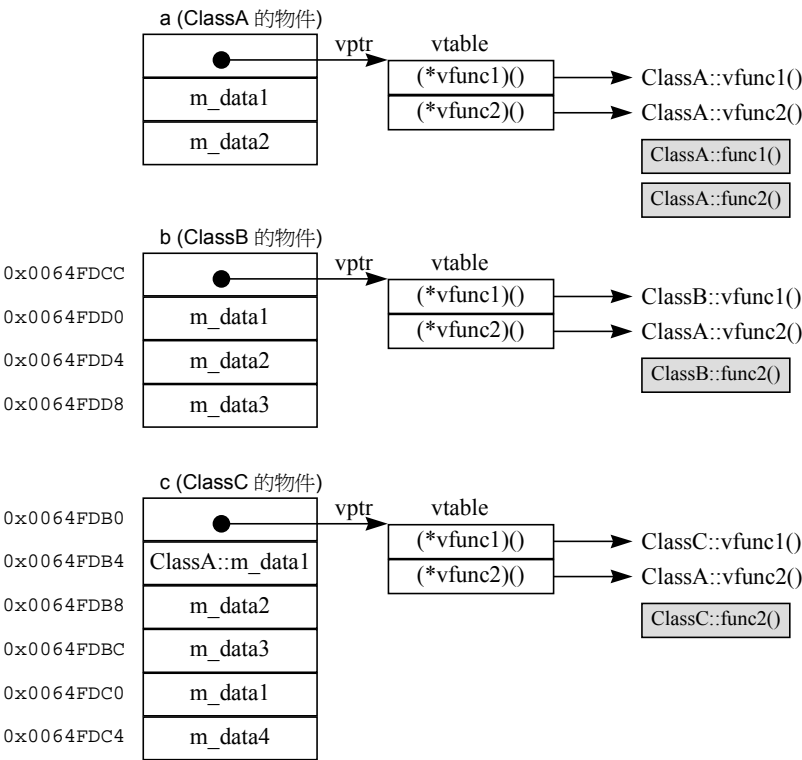


```
#0025 public:
#0026 int m_data1;
#0027 int m_data4;
#0028 void func2() { }
#0029 virtual void vfunc1() { }
#0030 };
#0031
#0032 void main()
#0033 {
#0034     cout << sizeof(ClassA) << endl;
#0035     cout << sizeof(ClassB) << endl;
#0036     cout << sizeof(ClassC) << endl;
#0037
#0038     ClassA a;
#0039     ClassB b;
#0040     ClassC c;
#0041
#0042     b.m_data1 = 1;
#0043     b.m_data2 = 2;
#0044     b.m_data3 = 3;
#0045     c.m_data1 = 11;
#0046     c.m_data2 = 22;
#0047     c.m_data3 = 33;
#0048     c.m_data4 = 44;
#0049     c.ClassA::m_data1 = 111;
#0050
#0051     cout << b.m_data1 << endl;
#0052     cout << b.m_data2 << endl;
#0053     cout << b.m_data3 << endl;
#0054     cout << c.m_data1 << endl;
#0055     cout << c.m_data2 << endl;
#0056     cout << c.m_data3 << endl;
#0057     cout << c.m_data4 << endl;
#0058     cout << c.ClassA::m_data1 << endl;
#0059
#0060     cout << &b << endl;
#0061     cout << &(b.m_data1) << endl;
#0062     cout << &(b.m_data2) << endl;
#0063     cout << &(b.m_data3) << endl;
#0064     cout << &c << endl;
#0065     cout << &(c.m_data1) << endl;
#0066     cout << &(c.m_data2) << endl;
#0067     cout << &(c.m_data3) << endl;
#0068     cout << &(c.m_data4) << endl;
#0069     cout << &(c.ClassA::m_data1) << endl;
#0070 }
```

執行結果與分析如下：

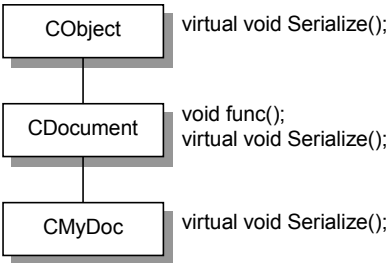
執行結果	意義	說明
12	Sizeof (ClassA)	2 個 <i>int</i> 加上一個 <i>vptr</i>
16	Sizeof (ClassB)	繼承自 <i>ClassA</i> ，再加上 1 個 <i>int</i>
24	Sizeof (ClassC)	繼承自 <i>ClassB</i> ，再加上 2 個 <i>int</i>
1	b.m_data1 的內容	
2	b.m_data2 的內容	
3	b.m_data3 的內容	
11	c.m_data1 的內容	
22	c.m_data2 的內容	
33	c.m_data3 的內容	
44	c.m_data4 的內容	
111	c.ClassA::m_data1 的內容	
0x0064FDCC	b 物件的起始位址	這個位址中的內容就是 <i>vptr</i>
0x0064FDD0	b.m_data1 的位址	
0x0064FDD4	b.m_data2 的位址	
0x0064FDD8	b.m_data3 的位址	
0x0064FDB0	c 物件的起始位址	這個位址中的內容就是 <i>vptr</i>
0x0064FDC0	c.m_data1 的位址	
0x0064FDB8	c.m_data2 的位址	
0x0064FDBC	c.m_data3 的位址	
0x0064FDC4	c.m_data4 的位址	
0x0064FDB4	c.ClassA::m_data1 的位址	

a、b、c 物件的內容圖示如下：



Object slicing 與 虛擬函式

我要在這裡說明虛擬函式另一個極重要的行為模式。假設有三個類別，階層關係如下：



以程式表現如下：

```
#0001 #include <iostream.h>
#0002
#0003 class CObject
#0004 {
#0005 public:
#0006     virtual void Serialize() { cout << "CObject::Serialize() \n\n"; }
#0007 };
#0008
#0009 class CDocument : public CObject
#0010 {
#0011 public:
#0012     int m_data1;
#0013     void func() { cout << "CDocument::func()" << endl;
#0014                 Serialize();
#0015                 }
#0016
#0017     virtual void Serialize() { cout << "CDocument::Serialize() \n\n"; }
#0018 };
#0019
#0020 class CMyDoc : public CDocument
#0021 {
#0022 public:
#0023     int m_data2;
#0024     virtual void Serialize() { cout << "CMyDoc::Serialize() \n\n"; }
#0025 };
#0026 //-----
#0027 void main()
#0028 {
#0029     CMyDoc mydoc;
#0030     CMyDoc* pmydoc = new CMyDoc;
#0031
#0032     cout << "#1 testing" << endl;
#0033     mydoc.func();
#0034
#0035     cout << "#2 testing" << endl;
#0036     ((CDocument*)&mydoc)->func();
#0037
#0038     cout << "#3 testing" << endl;
#0039     pmydoc->func();
#0040
#0041     cout << "#4 testing" << endl;
#0042     ((CDocument)mydoc).func();
#0043 }
```

由於 *CMyDoc* 自己沒有 *func* 函式，而它繼承了 *CDocument* 的所有成員，所以 *main* 之中的四個呼叫動作毫無問題都是呼叫 *CDocument::func*。但，*CDocument::func* 中所呼叫的 *Serialize* 是哪一個類別的成員函式呢？如果它是一般（non-virtual）函式，毫無問題應該是 *CDocument::Serialize*。但因為這是個虛擬函式，情況便有不同。以下是執行結果：

```
#1 testing
CDocument::func()
CMyDoc::Serialize()

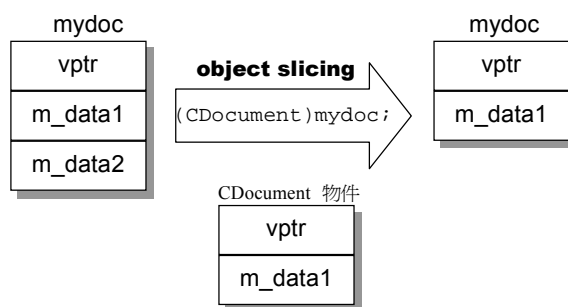
#2 testing
CDocument::func()
CMyDoc::Serialize()

#3 testing
CDocument::func()
CMyDoc::Serialize()

#4 testing
CDocument::func()
CDocument::Serialize() <-- 注意
```

前三個測試都符合我們對虛擬函式的期望：既然衍生類別已經改寫了虛擬函式 *Serialize*，那麼理當呼叫衍生類別之 *Serialize* 函式。這種行為模式非常頻繁地出現在 application framework 身上。後續當我追蹤 MFC 原始碼時，遇此情況會再次提醒你。

第四項測試結果則有點出乎意料之外。你知道，衍生物件通常都比基礎物件大（我是指記憶體空間），因為衍生物件不但繼承其基礎類別的成員，又有自己的成員。那麼所謂的 upcasting（向上強制轉型）：*(CDocument)mydoc*，將會造成物件的內容被切割（object slicing）：



當我們呼叫：

```
((CDocument)mydoc).func();
```

mydoc 已經是一個被切割得剩下半條命的物件，而 *func* 內部呼叫虛擬函式 *Serialize*；後者將使用的「*mydoc* 的虛擬函式指標」雖然存在，它的值是什麼呢？你是不是隱隱覺得有什麼大災難要發生？

幸運的是，由於 `((CDocument)mydoc).func()` 是個傳值而非傳址動作，編譯器以所謂的拷貝建構式（copy constructor）把 *CDocument* 物件內容複製了一份，使得 *mydoc* 的 *vtable* 內容與 *CDocument* 物件的 *vtable* 相同。本例雖沒有明顯做出一個拷貝建構式，編譯器會自動為你合成一個。

說這麼多，總結就是，經過所謂的 *data slicing*，本例的 *mydoc* 真正變成了一個完完全全的 *CDocument* 物件。所以，本例的第四項測試結果也就水落石出了。注意，"upcasting" 並不是慣用的動作，應該小心，甚至避免。

靜態成員（變數與函式）

我想你已經很清楚了，如果你依據一個類別產生出三個物件，每一個物件將各有一份成員變數。有時候這並不是你要的。假設你有一個類別，專門用來處理存款帳戶，它至少應該要有存戶的姓名、地址、存款額、利率等成員變數：

```
class SavingAccount
{
private:
    char m_name[40]; // 存戶姓名
    char m_addr[60]; // 存戶地址
    double m_total; // 存款額
    double m_rate; // 利率
    ...
};
```

這家行庫採用浮動利率，每個帳戶的利息都是根據當天的掛牌利率來計算。這時候 *m_rate* 就不適合成為每個帳戶物件中的一筆資料，否則每天一開市，光把所有帳戶內容

叫出來，修改 `m_rate` 的值，就花掉不少時間。`m_rate` 應該獨立在各物件之外，成為類別獨一無二的資料。怎麼做？在 `m_rate` 前面加上 `static` 修飾詞即可：

```
class SavingAccount
{
private:
    char m_name[40];          // 存戶姓名
    char m_addr[60];          // 存戶地址
    double m_total;           // 存款額
    static double m_rate;      // 利率
    ...
};
```

`static` 成員變數不屬於物件的一部份，而是類別的一部份，所以程式可以在還沒有誕生任何物件的時候就處理此種成員變數。但首先你必須初始化它。

不要把 `static` 成員變數的初始化動作安排在類別的建構式中，因為建構式可能一再被呼叫，而變數的初值卻只應該設定一次。也不要將初始化動作安排在表頭檔中，因為它可能會被含入許多地方，因此也就可能被執行許多次。你應該在實作檔中且類別以外的任何位置設定其初值。例如在 `main` 之中，或全域函式中，或任何函式之外：

```
double SavingAccount::m_rate = 0.0075; // 設立 static 成員變數的初值
void main() { ... }
```

這麼做可曾考慮到 `m_rate` 是個 `private` 資料？沒關係，設定 `static` 成員變數初值時，不受任何存取權限的束縛。請注意，`static` 成員變數的型別也出現在初值設定句中，因為這是一個初值設定動作，不是一個數量指定（assignment）動作。事實上，`static` 成員變數是在這時候（而不是在類別宣告中）才定義出來的。如果你沒有做這個初始化動作，會產生連結錯誤：

```
error LNK2001: unresolved external symbol "private: static double
SavingAccount::m_rate" (?m_rate@SavingAccount@@2HA)
```

關於 `static` 成員的使用實例，第 6 章的 `HelloMFC` 有一個，附錄 D 的「自製 `DBWIN` 工具（MFC 版）」也有一個。第 3 章的「`RTTI`（執行時期型別辨識）」一節模擬 `MFC` 的 `CRuntimeClass`，也有一個 `static` 應用實例。

下面是存取 *static* 成員變數的一種方式，注意，此刻還沒有誕生任何物件實體：

```
// 第一種存取方式
void main()
{
    SavingAccount::m_rate = 0.0075; // 欲此行成立，須把 m_rate 改為 public
}
```

下面這種情況則是產生一個物件後，透過物件來處理 *static* 成員變數：

```
// 第二種存取方式
void main()
{
    SavingAccount myAccount;
    myAccount.m_rate = 0.0075; // 欲此行成立，須把 m_rate 改為 public
}
```

你得搞清楚一個觀念，*static* 成員變數並不是因為物件的實現而才得以實現，它本來就存在，你可以想像它是一個全域變數。因此，第一種處理方式在意義上比較不會給人錯誤的印象。

只要 *access level* 允許，任何函式（包括全域函式或成員函式，*static* 或 *non-static*）都可以存取 *static* 成員變數。但如果你希望在產生任何 *object* 之前就存取其 *class* 的 *private static* 成員變數，則必須設計一個 *static* 成員函式（例如以下的 *setRate*）：

```
class SavingAccount
{
private:
    char m_name[40]; // 存戶姓名
    char m_addr[60]; // 存戶地址
    double m_total; // 存款額
    static double m_rate; // 利率
    ...
public:
    static void setRate(double newRate) { m_rate = newRate; }
    ...
};

double SavingAccount::m_rate = 0.0075; // 設立 static 成員變數的初值

void main()
```



```
{
    SavingAccount::setRate(0.0074); // 直接呼叫類別的 static 成員函式

    SavingAccount myAccount;
    myAccount.setRate(0.0074);      // 透過物件呼叫 static 成員函式
}
```

由於 *static* 成員函式不需要借助任何物件，就可以被呼叫執行，所以編譯器不會為它暗加一個 *this* 指標。也因為如此，*static* 成員函式無法處理類別之中的 *non-static* 成員變數。還記得嗎，我在前面說過，成員函式之所以能夠以單一份函式碼處理各個物件的資料而不紊亂，完全靠的是 *this* 指標的指示。

static 成員函式「沒有 *this* 參數」的這種性質，正是我們的 MFC 應用程式在準備 *callback* 函式時所需要的。第 6 章的 Hello World 例中我就會舉這樣一個實例。

C++ 程式的生與死：兼談建構式與解構式

C++ 的 *new* 運算子和 C 的 *malloc* 函式都是為了配置記憶體，但前者比之後者的優點是，*new* 不但配置物件所需的記憶體空間時，同時會引發建構式的執行。

所謂建構式（*constructor*），就是物件誕生後第一個執行（並且是自動執行）的函式，它的函式名稱必定要與類別名稱相同。

相對於建構式，自然就有個解構式（*destructor*），也就是在物件行將毀滅但未毀滅之前一刻，最後執行（並且是自動執行）的函式，它的函式名稱必定要與類別名稱相同，再在最前面加一個 *~* 符號。

一個有著階層架構的類別群組，當衍生類別的物件誕生之時，建構式的執行是由最基礎類別（*most based*）至最尾端衍生類別（*most derived*）；當物件要毀滅之前，解構式的執行則是反其道而行。第 3 章的 *frame1* 程式對此有所示範。

我以實例展示不同種類之物件的建構式執行時機。程式碼中的編號請對照執行結果。

```
#0001 #include <iostream.h>
#0002 #include <string.h>
#0003
#0004 class CDemo
#0005 {
#0006 public:
#0007     CDemo(const char* str);
#0008     ~CDemo();
#0009 private:
#0010     char name[20];
#0011 };
#0012
#0013 CDemo::CDemo(const char* str)    // 建構式
#0014 {
#0015     strcpy(name, str, 20);
#0016     cout << "Constructor called for " << name << '\n';
#0017 }
#0018
#0019 CDemo::~~CDemo()    // 解構式
#0020 {
#0021     cout << "Destructor called for " << name << '\n';
#0022 }
#0023
#0024 void func()
#0025 {
#0026     CDemo LocalObjectInFunc("LocalObjectInFunc"); // in stack ⑤
#0027     static CDemo StaticObject("StaticObject");    // local static ⑥
#0028     CDemo* pHeapObjectInFunc = new CDemo("HeapObjectInFunc"); // in heap ⑦
#0029
#0030     cout << "Inside func" << endl; ⑧
#0031
#0032 } ⑨
#0033
#0034 CDemo GlobalObject("GlobalObject"); // global static ①
#0035
#0036 void main()
#0037 {
#0038     CDemo LocalObjectInMain("LocalObjectInMain"); // in stack ②
#0039     CDemo* pHeapObjectInMain = new CDemo("HeapObjectInMain"); // in heap ③
#0040
#0041     cout << "In main, before calling func\n"; ④
#0042     func();
#0043     cout << "In main, after calling func\n"; ⑩
#0044
#0045 } ① ② ③
```

以下是執行結果：

- ❶ Constructor called for GlobalObject
- ❷ Constructor called for LocalObjectInMain
- ❸ Constructor called for HeapObjectInMain
- ❹ In main, before calling func
- ❺ Constructor called for LocalObjectInFunc
- ❻ Constructor called for StaticObject
- ❼ Constructor called for HeapObjectInFunc
- ❽ Inside func
- ❾ Destructor called for LocalObjectInFunc
- ❿ In main, after calling func
- ① Destructor called for LocalObjectInMain
- ② Destructor called for StaticObject
- ③ Destructor called for GlobalObject

我的結論是：

- 對於全域物件（如本例之 *GlobalObject*），程式一開始，其建構式就被執行（比程式進入點更早）；程式即將結束前其解構式被執行。MFC 程式就有這樣一個全域物件，通常以 **application object** 稱呼之，你將在第 6 章看到它。
- 對於區域物件，當物件誕生時，其建構式被執行；當程式流程將離開該物件的存活範圍（以至於物件將毀滅），其解構式被執行。
- 對於靜態（*static*）物件，當物件誕生時其建構式被執行；當程式將結束時（此物件因而將遭致毀滅）其解構式才被執行，但比全域物件的解構式早一步執行。
- 對於以 *new* 方式產生出來的區域物件，當物件誕生時其建構式被執行。解構式則在物件被 *delete* 時執行（上例程式未示範）。

物件的生存方式（in stack、in heap、global、local static）

既然談到了 *static* 物件，就讓我把所有可能的物件生存方式及其建構式呼叫時機做個整理。所有作法你都已經在前一節的小程式中看過。

在 C++ 中，有四種方法可以產生一個物件。第一種方法是在堆疊（*stack*）之中產生它：

```
void MyFunc()
{
    CFoo foo; // 在堆疊 (stack) 中產生 foo 物件
    ...
}
```

第二種方法是在堆積 (heap) 之中產生它：

```
void MyFunc()
{
    ...
    CFoo* pFoo = new CFoo(); // 在堆積 (heap) 中產生物件
}
```

第三種方法是產生一個全域物件（同時也必然是個靜態物件）：

```
CFoo foo; // 在任何函式範圍之外做此動作
```

第四種方法是產生一個區域靜態物件：

```
void MyFunc()
{
    static CFoo foo; // 在函式範圍 (scope) 之內的一個靜態物件
    ...
}
```

不論任何一種作法，C++ 都會產生一個針對 *CFoo* 建構式的呼叫動作。前兩種情況，C++ 在配置記憶體 -- 來自堆疊 (stack) 或堆積 (heap) -- 之後立刻產生一個隱藏的（你的原始碼中看不出來的）建構式呼叫。第三種情況，由於物件實現於任何「函式活動範圍 (function scope)」之外，顯然沒有地方來安置這樣一個建構式呼叫動作。

是的，第三種情況（靜態全域物件）的建構式呼叫動作必須靠 *startup* 碼幫忙。*startup* 碼是什麼？是更早於程式進入點 (*main* 或 *WinMain*) 執行起來的碼，由 C++ 編譯器提供，被聯結到你的程式中。*startup* 碼可能做些像函式庫初始化、行程資訊設立、I/O stream 產生等等動作，以及對 *static* 物件的初始化動作（也就是呼叫其建構式）。

當編譯器編譯你的程式，發現一個靜態物件，它會把這個物件加到一個串列之中。更精

確地說則是，編譯器不只是加上此靜態物件，它還加上一個指標，指向物件之建構式及其參數（如果有的話）。把控制權交給程式進入點（*main* 或 *WinMain*）之前，*startup* 碼會快速在該串列上移動，呼叫所有登記有案的建構式並使用登記有案的參數，於是就初始化了你的靜態物件。

第四種情況（區域靜態物件）相當類似 C 語言中的靜態區域變數，只會有一個實體（*instance*）產生，而且在固定的記憶體上（既不是 *stack* 也不是 *heap*）。它的建構式在控制權第一次移轉到其宣告處（也就是在 *MyFunc* 第一次被呼叫）時被呼叫。

所謂 "Unwinding"

C++ 物件依其生存空間，適當地依照一定的順序被解構（*destructured*）。但是如果發生異常情況（*exception*），而程式設計了異常情況處理程式（*exception handling*），控制權就會截奪取直接「直接跳」到你所設定的處理常式，這時候堆疊中的 C++ 物件有沒有機會被解構？這得視編譯器而定。如果編譯器有支援 *unwinding* 功能，就會在一個異常情況發生時，將堆疊中的所有物件都解構掉。

關於異常情況（*exception*）及異常處理（*exception handling*），稍後再一討論。

執行時期型別資訊（RTTI）

我們有可能在程式執行過程中知道某個物件是屬於哪一種類別嗎？這種在 C++ 中稱為執行時期型別資訊（*Runtime Type Information*，*RTTI*）的能力，晚近較先進的編譯器如 *Visual C++ 4.0* 和 *Borland C++ 5.0* 才開始廣泛支援。以下是一個實例：

```
#0001 // RTTI.CPP - built by C:\> cl.exe -GR rtti.cpp <ENTER>
#0002 #include <typeinfo.h>
#0003 #include <iostream.h>
#0004 #include <string.h>
#0005
#0006 class graphicImage
```

```
#0007 {
#0008 protected:
#0009     char name[80];
#0010
#0011 public:
#0012     graphicImage()
#0013     {
#0014         strcpy(name,"graphicImage");
#0015     }
#0016
#0017     virtual void display()
#0018     {
#0019         cout << "Display a generic image." << endl;
#0020     }
#0021
#0022     char* getName()
#0023     {
#0024         return name;
#0025     }
#0026 };
#0027 //-----
#0028 class GIFimage : public graphicImage
#0029 {
#0030 public:
#0031     GIFimage()
#0032     {
#0033         strcpy(name,"GIFimage");
#0034     }
#0035
#0036     void display()
#0037     {
#0038         cout << "Display a GIF file." << endl;
#0039     }
#0040 };
#0041
#0042 class PICTimage : public graphicImage
#0043 {
#0044 public:
#0045     PICTimage()
#0046     {
#0047         strcpy(name,"PICTimage");
#0048     }
#0049
#0050     void display()
#0051     {
#0052         cout << "Display a PICT file." << endl;
```

```
#0053     }
#0054 };
#0055 //-----
#0056 void processFile(graphicImage *type)
#0057 {
#0058     if (typeid(GIFimage) == typeid(*type))
#0059     {
#0060         ((GIFimage *)type)->display();
#0061     }
#0062     else if (typeid(PICTimage) == typeid(*type))
#0063     {
#0064         ((PICTimage *)type)->display();
#0065     }
#0066     else
#0067         cout << "Unknown type! " << (typeid(*type)).name() << endl;
#0068 }
#0069
#0070 void main()
#0071 {
#0072     graphicImage *gImage = new GIFimage();
#0073     graphicImage *pImage = new PICTimage();
#0074
#0075     processFile(gImage);
#0076     processFile(pImage);
#0077 }
```

執行結果如下：

```
Display a GIF file.
Display a PICT file.
```

這個程式與 RTTI 相關的地方有三個：

1. 編譯時需選用 /GR 選項 (/GR 的意思是 enable C++ RTTI)
2. 含入 `typeinfo.h`
3. 新的 `typeid` 運算子。這是一個多載 (overloading) 運算子，多載的意思就是擁有一個以上的型式，你可以想像那是一種靜態的多型 (Polymorphism)。 `typeid` 的參數可以是類別名稱 (如本例 #58 左)，也可以是物件指標 (如本例 #58 右)。它傳回一個 `type_info&`。 `type_info` 是一個類別，定義於 `typeinfo.h` 中：

```
class type_info {
```

```

public:
    virtual ~type_info();
    int operator==(const type_info& rhs) const;
    int operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
private:
    ...
};

```

雖然 Visual C++ 編譯器自從 4.0 版已經支援 RTTI，但 MFC 4.x 並未使用編譯器的能力完成其對 RTTI 的支援。MFC 有自己一套沿用已久的辦法（從 1.0 版就開始了）。喔，不要因為 MFC 的作法特殊而非難它，想想看它的悠久歷史。

MFC 的 RTTI 能力牽扯到一組非常神秘的巨集（*DECLARE_DYNAMIC*、*IMPLEMENT_DYNAMIC*）和一個非常神秘的類別（*CRuntimeClass*）。MFC 程式員都知道怎麼用它，卻沒幾個人懂得其運作原理。大道不過三兩行，說穿不值一文錢，下一章我就模擬出一個 RTTI 的 DOS 版本給你看。

動態生成（Dynamic Creation）

物件導向術語中有一個名為 *persistence*，意思是永續存留。放在 RAM 中的東西，生命受到電力的左右，不可能永續存留；唯一的辦法是把它寫到檔案去。MFC 的一個術語 *Serialize*，就是做有關檔案讀寫的永續存留動作，並且實做作出一個虛擬函式，就叫作 *Serialize*。

看起來永續存留與本節的主題「動態生成」似乎沒有什麼干連。有！你把你的資料儲存到檔案，這些資料很可能（通常是）物件中的成員變數；我把它讀出來後，勢必要依據檔案上的記載，重新 *new* 出那些個物件來。問題在於，即使我的程式有那些類別定義（就算我的程式和你的程式有一樣的內容好了），我能夠這麼做嗎：

```

char className[30] = getClassname(); // 從檔案（或使用者輸入）獲得一個類別名稱
CObject* obj = new classname;       // 這一行行不通

```


首先，`new classname` 這個動作就過不了關。其次，就算過得了關，`new` 出來的物件究竟該是什麼類別型態？雖然以一個指向 MFC 類別老祖宗（*CObject*）的物件指標來容納它絕對沒有問題，但終不好總是如此吧！不見得這樣子就能夠滿足你的程式需求啊。

顯然，你能夠以 *Serialize* 函式寫檔，我能夠以 *Serialize* 函式讀檔，但我就是沒辦法恢復你原來的狀態 -- 除非我的程式能夠「動態生成」。

MFC 支援動態生成，靠的是一組非常神秘的巨集（*DECLARE_DYNCREATE*、*IMPLEMENT_DYNCREATE*）和一個非常神秘的類別（*CRuntimeClass*）。第 3 章中我將把它抽絲剝繭，以一個 DOS 程式模擬出來。

異常處理（Exception Handling）

Exception（異常情況）是一個頗為新鮮的 C++ 語言特徵，可以幫助你管理執行時期的錯誤，特別是那些發生在深度巢狀（nested）函式呼叫之中的錯誤。Watcom C++ 是最早支援 ANSI C++ 異常情況的編譯器，Borland C++ 4.0 隨後跟進，然後是 Microsoft Visual C++ 和 Symantec C++。現在，這已成為 C++ 編譯器必需支援的項目。

C++ 的 exception 基本上是與 C 的 *setjmp* 和 *longjmp* 函式對等的東西，但它增加了一些功能，以處理 C++ 程式的特別需求。從深度巢狀的常式呼叫中直接以一條捷徑撤回到異常情況處理常式（exception handler），這種「錯誤管理方式」遠比結構化程式中經過層層的常式傳回一系列的錯誤狀態來的好。事實上 exception handling 是 MFC 和 OWL 兩個 application frameworks 的防彈中心。

C++ 導入了三個新的 exception 保留字：

1. *try*。之後跟隨一段以 { } 圈出來的程式碼，exception 可能在其中發生。
2. *catch*。之後跟隨一段以 { } 圈出來的程式碼，那是 exception 處理常式之所在。*catch* 應該緊跟在 *try* 之後。
3. *throw*。這是一個指令，用來產生（丟出）一個 exception。

下面是個實例：

```
try {
    // try block.
}
catch (char *p) {
    printf("Caught a char* exception, value %s\n",p);
}
catch (double d) {
    printf("Caught a numeric exception, value %g\n",d);
}
catch (...) { // catch anything
    printf("Caught an unknown exception\n");
}
```

MFC 早就支援 `exception`，不過早期它用的是非標準語法。Visual C++ 4.0 編譯器本身支援完整的 C++ `exceptions`，MFC 也因此有了兩個 `exception` 版本：你可以使用語言本身提供的性能，也可以沿用 MFC 古老的方法（以巨集形式出現）。人們曾經因為 MFC 的方案不同於 ANSI 標準而非難它，但是不要忘記它已經運作了多少年。

MFC 的 `exceptions` 機制是以巨集和 `exception types` 為基礎。這些巨集類似 C++ 的 `exception` 保留字，動作也滿像。MFC 以下列巨集模擬 C++ `exception handling`：

```
TRY
CATCH(type,object)
AND_CATCH(type,object)
END_CATCH
CATCH_ALL(object)
AND_CATCH_ALL(object)
END_CATCH_ALL
END_TRY
THROW()
THROW_LAST()
```

MFC 所使用的語法與日漸浮現的標準稍微不同，不過其間差異微不足道。為了以 MFC 捕捉 `exceptions`，你應該建立一個 `TRY` 區塊，下面接著 `CATCH` 區塊：

```
TRY {
    // try block.
}
CATCH (CMemoryException, e) {
```

```
        printf("Caught a memory exception.\n");
    }
    AND_CATCH_ALL (e) {
        printf("Caught an exception.\n");
    }
    END_CATCH_ALL
```

THROW 巨集相當於 C++ 語言中的 *throw* 指令；你以什麼型態做為 *THROW* 的參數，就會有一個相對應的 *AfxThrow_* 函式被呼叫（這是檯面下的行為）：

MFC Exception Type	MFC Throw Function	DOS support	Windows support
CException		v	v
CMemoryException	AfxThrowMemoryException	v	v
CFileException	AfxThrowFileException	v	v
CArchiveException	AfxThrowArchiveException	v	v
CNotSupportedException	AfxThrowNotSupportedException	v	v
CResourceException	AfxThrowResourceException		v
COleException	AfxThrowOleException		v
COleDispatchException	AfxThrowOleDispatchException		v
CDBException	AfxThrowDBException		v
CDaoException	AfxThrowDaoException		v
CUserException	AfxThrowUserException		v

以下是 MFC 4.x 的 exceptions 巨集定義：

```
// in AFX.H
/////////////////////////////////////////////////////////////////
// Exception macros using try, catch and throw
// (for backward compatibility to previous versions of MFC)

#ifdef _AFX_OLD_EXCEPTIONS

#define TRY { AFX_EXCEPTION_LINK _afxExceptionLink; try {
```

```

#define CATCH(class, e) } catch (class* e) \
    { ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
      _afxExceptionLink.m_pException = e;

#define AND_CATCH(class, e) } catch (class* e) \
    { ASSERT(e->IsKindOf(RUNTIME_CLASS(class))); \
      _afxExceptionLink.m_pException = e;

#define END_CATCH } }

#define THROW(e) throw e
#define THROW_LAST() (AfxThrowLastCleanup(), throw)

// Advanced macros for smaller code
#define CATCH_ALL(e) } catch (CException* e) \
    { { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
      _afxExceptionLink.m_pException = e;

#define AND_CATCH_ALL(e) } catch (CException* e) \
    { { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
      _afxExceptionLink.m_pException = e;

#define END_CATCH_ALL } } }

#define END_TRY } catch (CException* e) \
    { ASSERT(e->IsKindOf(RUNTIME_CLASS(CException))); \
      _afxExceptionLink.m_pException = e; } }

#else //_AFX_OLD_EXCEPTIONS

////////////////////////////////////
// Exception macros using setjmp and longjmp
// (for portability to compilers with no support for C++ exception handling)

#define TRY \
    { AFX_EXCEPTION_LINK _afxExceptionLink; \
      if (::setjmp(_afxExceptionLink.m_jumpBuf) == 0)

#define CATCH(class, e) \
    else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
    { class* e = (class*)_afxExceptionLink.m_pException;

#define AND_CATCH(class, e) \
    } else if (::AfxCatchProc(RUNTIME_CLASS(class))) \
    { class* e = (class*)_afxExceptionLink.m_pException;

```

```
#define END_CATCH \
    } else { ::AfxThrow(NULL); } }

#define THROW(e) AfxThrow(e)
#define THROW_LAST() AfxThrow(NULL)

// Advanced macros for smaller code
#define CATCH_ALL(e) \
    else { CException* e = _afxExceptionLink.m_pException;

#define AND_CATCH_ALL(e) \
    } else { CException* e = _afxExceptionLink.m_pException;

#define END_CATCH_ALL } }

#define END_TRY }

#endif //_AFX_OLD_EXCEPTIONS
```

Template

這並不是一本 C++ 書籍，我也並不打算介紹太多距離「運用 MFC」主題太遠的 C++ 論題。Template 雖然很重要，但它與「運用 MFC」有什麼關係？有！第8章當我們開始設計 Scribble 程式時，需要用到 MFC 的 collection classes，而這一組類別自從 MFC 3.0 以來就有了 template 版本（因為 Visual C++ 編譯器從 2.0 版開始支援 C++ template）。運用之前，我們總該了解一下新的語法、精神、以及應用。

好，到底什麼是 template？重要性如何？Kaare Christian 在 1994/01/25 的 PC-Magazine 上有一篇文章，說得很好：

無性生殖並不只是存在於遺傳工程上，對程式員而言它也是一個日來日久的動作。過去，我們只不過是以一個簡單而基本的工具，它就是一個字編輯器，重製我們的程式碼。今天，C++ 提供給我們一個更好的繁殖方法：template。

複製一段既存程式碼的一個最平常的理由就是為了改變資料型態。舉個例子，假設你寫了一個繪圖函式，使用整數 x, y 座標；突然之間你需要相同的程式碼，但座標值改採

long。你當然可以使用一個文字編輯器把這段碼拷貝一份，然後把其中的資料型態改變過來。在 C++，你甚至可以使用多載（overloaded）函式，那麼你就可以仍舊使用相同的函式名稱。函式的多載的確使我們在比較清爽的程式碼，但它們意味著你還是必須在你的程式的許多地方維護完全相同的演算法。

C 語言對此問題的解答是：使用巨集。雖然你因此對於相同的演算法只需要一次程式碼，但巨集有它自己的缺點。第一，它只適用於簡單的功能。第二個缺點比較嚴重：巨集不提供資料型態檢查，因此犧牲了 C++ 的一個主要效益。第三個缺點是：巨集並非函式，程式中任何呼叫巨集的地方都會被編譯器前置處理器原原本本地插入巨集所定義的那一段碼，而非只是一個函式呼叫，因此你每使用一次巨集，你的執行檔就會膨脹一點。

Templates 提供比較好的解決方案，它把「一般性的演算法」和其「對資料型態的實作部份」區分開來。你可以先寫演算法的程式碼，稍後在使用時再填入實際資料型態。新的 C++ 語法使「資料型態」以參數的型態出現。在 template，你可以擁有巨集「只需要一次」的優點，以及多載函式「型態檢查」的優點。

C++ 的 template 有兩種，一種針對 function，另一種針對 class。

Template Functions

假設我們需要一個計算數值幕次方的函式，名曰 *power*。我們只接受正幕次方數，如果是負幕次方，就讓結果為 0。

對於整數，我們的函式應該是這樣：

```
#0001 int power(int base, int exponent)
#0002 {
#0003     int result = base;
#0004     if (exponent == 0) return (int)1;
#0005     if (exponent < 0)  return (int)0;
#0006     while (--exponent) result *= base;
#0007     return result;
#0008 }
```

對於長整數，函式應該是這樣：

```
#0001 long power(long base, int exponent)
#0002 {
#0003     long result = base;
#0004     if (exponent == 0) return (long)1;
#0005     if (exponent < 0)  return (long)0;
#0006     while (--exponent) result *= base;
#0007     return result;
#0008 }
```

對於浮點數，我們應該...，對於複數，我們應該...。喔喔，為什麼不能夠把資料型別也變成參數之一，在使用時指定呢？是的，這就是 `template` 的妙用：

```
template <class T> T power(T base, int exponent);
```

寫成兩行或許比較清楚：

```
template <class T>
T power(T base, int exponent);
```

這樣的函式宣告是以一個特殊的 `template` 字首開始，後面緊跟著一個參數列（本例只一個參數）。容易讓人迷惑的是其中的 "class" 字眼，它其實並不一定表示 C++ 的 `class`，它也可以是一個普通的資料型態。`<class T>` 只不過是表示：T 是一種型態，而此一型態將在呼叫此函式時才給予。

下面就是 `power` 函式的 `template` 版本：

```
#0001 template <class T>
#0002 T power(T base, int exponent)
#0003 {
#0004     T result = base;
#0005     if (exponent == 0) return (T)1;
#0006     if (exponent < 0)  return (T)0;
#0007     while (--exponent) result *= base;
#0008     return result;
#0009 }
```

傳回值必須確保為型態 T，以吻合 `template` 函式的宣告。

下面是 `template` 函式的呼叫方法：

```

#0001 #include <iostream.h>
#0002 void main()
#0003 {
#0004     int i = power(5, 4);
#0005     long l = power(1000L, 3);
#0006     long double d = power((long double)1e5, 2);
#0007
#0008     cout << "i= " << i << endl;
#0009     cout << "l= " << l << endl;
#0010     cout << "d= " << d << endl;
#0011 }

```

執行結果如下：

```

i= 625
l= 1000000000
d= 1e+010

```

在第一次呼叫中，*T* 變成 *int*，在第二次呼叫中，*T* 變成 *long*。而在第三次呼叫中，*T* 又成為了一個 *long double*。但如果呼叫時候把資料型態混亂掉了，像這樣：

```
int i = power(1000L, 4); // 基值是個 long，傳回值卻是個 int。錯誤示範！
```

編譯時就會出錯。

`template` 函式的資料型別參數 *T* 究竟可以適應多少種型態？我要說，幾乎「任何資料型態」都可以，但函式中對該型態數值的任何運算動作，都必須支援 -- 否則編譯器就不知道該怎麼辦了。以 *power* 函式為例，它對於 *result* 和 *base* 兩個數值的運算動作有：

1. *T result = base;*
2. *return (T)1;*
3. *return (T)0;*
4. *result *= base;*
5. *return result;*

C++ 所有內建資料型態如 *int* 或 *long* 都支援上述運算動作。但如果你為某個 C++ 類別產生一個 *power* 函式，那麼這個 C++ 類別必須包含適當的成員函式以支援上述動作。

如果你打算在 `template` 函式中以 C++ 類別代替 `class T`，你必須清楚知道哪些運算動作曾被使用於此一函式中，然後在你的 C++ 類別中把它們全部實作出來。否則，出現的

錯誤耐人尋味。

Template Classes

我們也可以建立 `template classes`，使它們能夠神奇地操作任何型態的資料。下面這個例子是讓 `CThree` 類別儲存三個成員變數，成員函式 `Min` 傳回其中的最小值，成員函式 `Max` 則傳回其中的最大值。我們把它設計為 `template class`，以便這個類別能適用於各式各樣的資料型態：

```
#0001 template <class T>
#0002 class CThree
#0003 {
#0004 public :
#0005     CThree(T t1, T t2, T t3);
#0006     T Min();
#0007     T Max();
#0008 private:
#0009     T a, b, c;
#0010 };
```

語法還不至於太稀奇古怪，把 `T` 看成是大家熟悉的 `int` 或 `float` 也就是了。下面是成員函式的定義：

```
#0001 template <class T>
#0002 T CThree<T>::Min()
#0003 {
#0004     T minab = a < b ? a : b;
#0005     return minab < c ? minab : c;
#0006 }
#0007
#0008 template <class T>
#0009 T CThree<T>::Max()
#0010 {
#0011     T maxab = a < b ? b : a;
#0012     return maxab < c ? c : maxab;
#0013 }
#0014
#0015 template <class T>
#0016 CThree<T>::CThree(T t1, T t2, T t3) :
#0017     a(t1), b(t2), c(t3)
#0018 {
```

```
#0019     return;
#0020 }
```

這裡就得多注意些了。每一個成員函式前都要加上 `template <class T>`，而且類別名稱應該使用 `CThree<T>`。

以下是 `template class` 的使用方式：

```
#0001 #include <iostream.h>
#0002 void main()
#0003 {
#0004     CThree<int> obj1(2, 5, 4);
#0005     cout << obj1.Min() << endl;
#0006     cout << obj1.Max() << endl;
#0007
#0008     CThree<float> obj2(8.52, -6.75, 4.54);
#0009     cout << obj2.Min() << endl;
#0010     cout << obj2.Max() << endl;
#0011
#0012     CThree<long> obj3(646600L, 437847L, 364873L);
#0013     cout << obj3.Min() << endl;
#0014     cout << obj3.Max() << endl;
#0015 }
```

執行結果如下：

```
2
5
-6.75
8.52
364873
646600
```

稍早我曾說過，只有當 `template` 函式對於資料型別 `T` 支援所有必要的運算動作時，`T` 才得被視為有效。此一限制對於 `template classes` 亦屬實。爲了針對某些類別產生一個 `CThree`，該類別必須提供 `copy` 建構式以及 `operator<`，因為它們是 `Min` 和 `Max` 成員函式中對 `T` 的運算動作。

但是如果你用的是別人 `template classes`，你又如何知道什麼樣的運算動作是必須的呢？唔，該 `template classes` 的說明文件中應該有所說明。如果沒有，只有原始碼才能揭露秘密。C++ 內建資料型別如 `int` 和 `float` 等不需要在意這份要求，因為所有內建的資料型別都支援所有的標準運算動作。

Templates 的編譯與聯結

對程式員而言 C++ `templates` 可說是十分容易設計與使用，但對於編譯器和聯結器而言卻是一大挑戰。編譯器遇到一個 `template` 時，不能夠立刻為它產生機器碼，它必須等待，直到 `template` 被指定某種型態。從程式員的觀點來看，這意味著 `template function` 或 `template class` 的完整定義將出現在 `template` 被使用的每一個角落，否則，編譯器就沒有足夠的資訊可以幫助產生目的碼。當多個原始檔案使用同一個 `template` 時，事情更趨複雜。

隨著編譯器的不同，掌握這種複雜度的技術也不同。有一個常用的技術，Borland 稱之為 `Smart`，應該算是最容易的：每一個使用 `Template` 的程式碼的目的檔中都存在有 `template` 碼，聯結器負責複製和刪除。

假設我們有一個程式，包含兩個原始檔案 `A.CPP` 和 `B.CPP`，以及一個 `THREE.H`（其內定義了一個 `template` 類別，名為 `CThree`）。`A.CPP` 和 `B.CPP` 都含入 `THREE.H`。如果 `A.CPP` 以 `int` 和 `double` 使用這個 `template` 類別，編譯器將在 `A.OBJ` 中產生 `int` 和 `double` 兩種版本的 `template` 類別可執行碼。如果 `B.CPP` 以 `int` 和 `float` 使用這個 `template` 類別，編譯器將在 `B.OBJ` 中產生 `int` 和 `float` 兩種版本的 `template` 類別可執行碼。即使雖然 `A.OBJ` 中已經有一個 `int` 版了，編譯器沒有辦法知道。

然後，在聯結過程中，所有重複的部份將被刪除。請看圖 2-1。

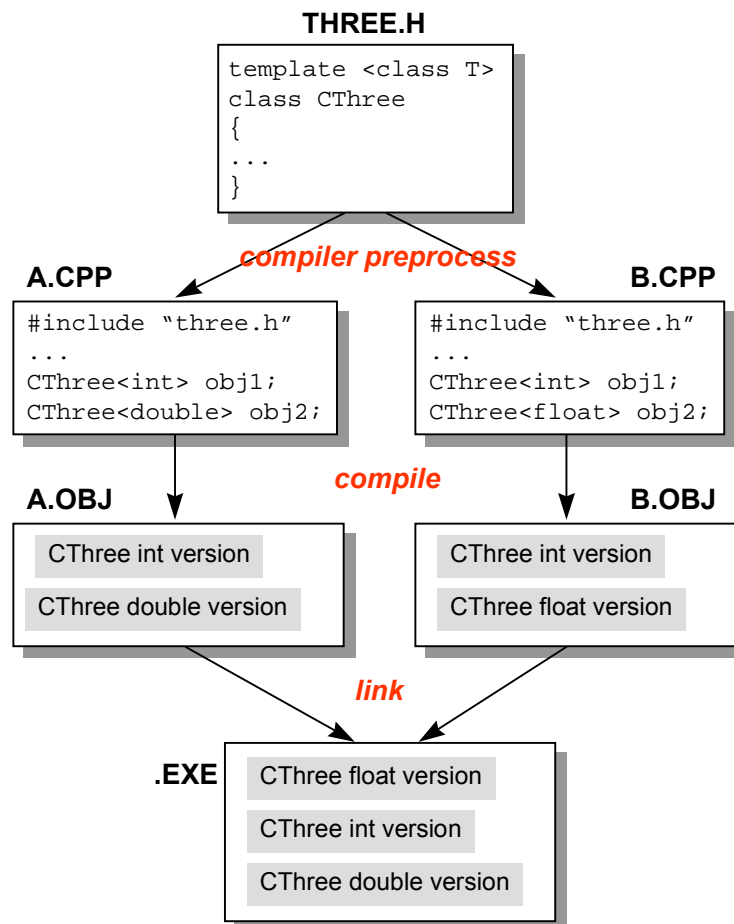


圖 2-1 聯結器會把所有贅餘的 **template** 碼剔除。這在 **Borland** 聯結器裡頭稱為 **smart** 技術。其他聯結器亦使用類似的技術。

第一篇 勿在浮砂築高台

第 3 頁

MFC 六大關鍵技術之模擬

演化（evolution）永遠在進行，
這個世界卻不是每天都有革命（revolution）發生。
Application Framework 在軟體界確實稱得上具有革命精神。

模擬 MFC？有必要嗎？意義何在？如何模擬？

我已經在序言以及導讀開宗明義說過了，這本書除了教導你使用 MFC，另一個重要的功能是你認識一個 application framework 的內部運作。以 MFC 為教學載具，我既可以讓你領略 application framework 的設計方式，更可以让你熟悉 MFC 類別，將來運用時得心應手。呵，雙效合一。

整個 MFC 4.0 多達 189 個類別，原始碼達 252 個實作檔，58 個表頭檔，共 10 MB 之多。MFC 4.2 又多加了 29 個類別。這麼龐大的對象，當然不是每一個類別每一個資料結構都是我的模擬目標。我只挑選最神秘又最重要，與應用程式主幹息息相關的題目，包括：

- MFC 程式的初始化過程
- RTTI（Runtime Type Information）執行時期型別資訊
- Dynamic Creation 動態生成
- Persistence 永續留存
- Message Mapping 訊息映射
- Message Routing 訊息繞行

MFC 本身的設計在 Application Framework 之中不見得最好，敵視者甚至認為它是個 Minotaur（註）！但無論如何，這是當今軟體霸主微軟公司的產品，從探究 application framework 設計的角度來說，實為一個重要參考；而如果從選擇一套 application framework 作為軟體開發工具的角度來說，單就就業市場的需求，我對 MFC 的推薦再加 10 分！

註：Minotaur 是希臘神話中的牛頭人身怪物，居住在迷宮之中。進入迷宮的人如果走不出來，就會被牠一口吃掉。

另一個問題是，為什麼要模擬？第三篇第四篇各章節不是還要挖 MFC 原始碼來看嗎？原因是 MFC 太過龐大，我必須撇開枝節，把唯一重點突顯出來，才容易收到教育效果。而且，模擬才能實證嘛！

如何模擬？我採用文字模式，也就是所謂的 Console 程式，這樣可以把程式結構的負荷降到最低。但是像訊息映射和訊息繞行怎麼辦？訊息的流動是 Windows 程式才有的特徵啊！唔，看了你就知道。

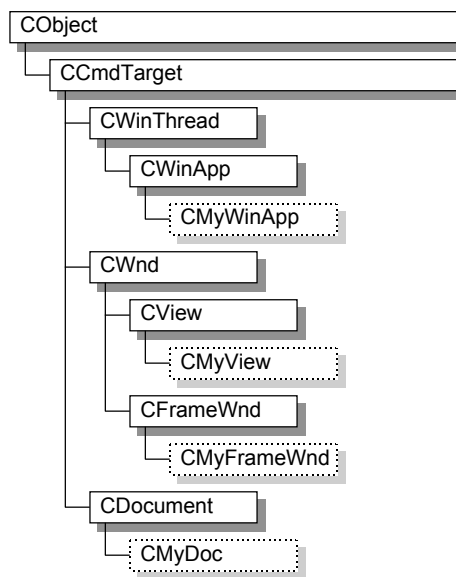
我的最高原則是：簡化再簡化，簡化到不能再簡化。

請注意，以下所有程式的類別階層架構、類別名稱、變數名稱、結構名稱、函式名稱、函式行為，都以 MFC 為模擬對象，具體而微。也可以說，我從數以萬行計的 MFC 原始碼中，「偷」了一些出來，砍掉旁枝末節，只露出重點。

在檔案的安排上，我把模擬 MFC 的類別都集中在 MFC.H 和 MFC.CPP 中，把自己衍生的類別集中在 MY.H 和 MY.CPP 中。對於自定類別，我的命名方式是在父類別的名稱前面加一個 "My"，例如衍生自 CWinApp 者，名為 CMyWinApp，衍生自 CDocument 者，名為 CMyDoc。

MFC 類別階層

首先我以一個極簡單的程式 Frame1，把 MFC 數個最重要類別的階層關係模擬出來：



這個實例模擬 MFC 的類別階層。後續數節中，我會繼續在這個類別階層上開發新的能力。在這些名為 Frame? 的各範例中，我以 MFC 原始碼為藍本，儘量模擬 MFC 的內部行為，並且使用完全相同的類別名稱、函式名稱、變數名稱。這樣的模擬對於我們在第三篇以及第四篇中深入探討 MFC 時將有莫大助益。相信我，這是真的。

Frame1 範例程式

MFC.H

```

#0001 #include <iostream.h>
#0002
#0003 class CObject
#0004 {
#0005 public:

```



```
#0006   CObject::CObject() { cout << "CObject Constructor \n"; }
#0007   CObject::~CObject() { cout << "CObject Destructor \n"; }
#0008 };
#0009
#0010   class CCmdTarget : public CObject
#0011   {
#0012   public:
#0013       CCmdTarget::CCmdTarget() { cout << "CCmdTarget Constructor \n"; }
#0014       CCmdTarget::~CCmdTarget() { cout << "CCmdTarget Destructor \n"; }
#0015   };
#0016
#0017   class CWinThread : public CCmdTarget
#0018   {
#0019   public:
#0020       CWinThread::CWinThread() { cout << "CWinThread Constructor \n"; }
#0021       CWinThread::~CWinThread() { cout << "CWinThread Destructor \n"; }
#0022   };
#0023
#0024   class CWinApp : public CWinThread
#0025   {
#0026   public:
#0027       CWinApp* m_pCurrentWinApp;
#0028
#0029   public:
#0030       CWinApp::CWinApp() { m_pCurrentWinApp = this;
#0031                           cout << "CWinApp Constructor \n"; }
#0032       CWinApp::~CWinApp() { cout << "CWinApp Destructor \n"; }
#0033   };
#0034   class CDocument : public CCmdTarget
#0035   {
#0036   public:
#0037       CDocument::CDocument() { cout << "CDocument Constructor \n"; }
#0038       CDocument::~CDocument() { cout << "CDocument Destructor \n"; }
#0039   };
#0040
#0041
#0042   class CWnd : public CCmdTarget
#0043   {
#0044   public:
#0045       CWnd::CWnd() { cout << "CWnd Constructor \n"; }
#0046       CWnd::~CWnd() { cout << "CWnd Destructor \n"; }
#0047   };
#0048
#0049   class CFrameWnd : public CWnd
#0050   {
```

```

#0051 public:
#0052     CFrameWnd::CFrameWnd() { cout << "CFrameWnd Constructor \n"; }
#0053     CFrameWnd::~CFrameWnd() { cout << "CFrameWnd Destructor \n"; }
#0054 };
#0055
#0056 class CView : public CWnd
#0057 {
#0058 public:
#0059     CView::CView() { cout << "CView Constructor \n"; }
#0060     CView::~CView() { cout << "CView Destructor \n"; }
#0061 };
#0062
#0063
#0064 // global function
#0065
#0066 CWinApp* AfxGetApp();

```

MFC.CPP

```

#0001 #include "my.h" // 原本含入 mfc.h 就好，但爲了 CMyWinApp 的定義，所以...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 CWinApp* AfxGetApp()
#0006 {
#0007     return theApp.m_pCurrentWinApp;
#0008 }

```

MY.H

```

#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp::CMyWinApp() { cout << "CMyWinApp Constructor \n"; }
#0008     CMyWinApp::~CMyWinApp() { cout << "CMyWinApp Destructor \n"; }
#0009 };
#0010
#0011 class CMyFrameWnd : public CFrameWnd
#0012 {
#0013 public:
#0014     CMyFrameWnd() { cout << "CMyFrameWnd Constructor \n"; }
#0015     ~CMyFrameWnd() { cout << "CMyFrameWnd Destructor \n"; }

```

```
#0016 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp; // global object
#0004
#0005 //-----
#0006 // main
#0007 //-----
#0008 void main()
#0009 {
#0010
#0011     CWinApp* pApp = AfxGetApp();
#0012
#0013 }
```

Frame1 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第 4 章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

Frame1 的執行結果是：

```
CObject Constructor
CCmdTarget Constructor
CWinThread Constructor
CWinApp Constructor
CMyWinApp Constructor

CMyWinApp Destructor
CWinApp Destructor
CWinThread Destructor
CCmdTarget Destructor
CObject Destructor
```

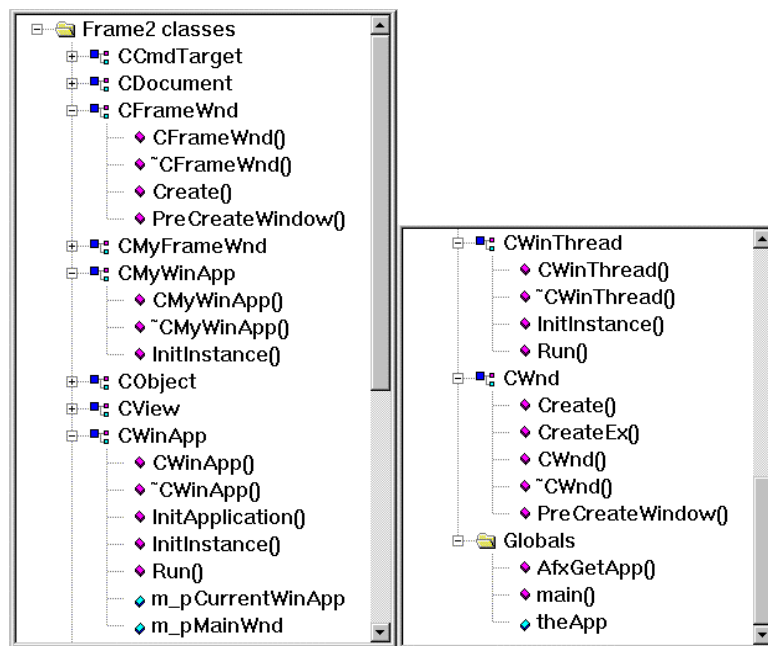
好，你看到了，Frame1 並沒有 *new* 任何物件，反倒是有一個全域物件 *theApp* 存在。C++ 規定，全域物件的建構將比程式進入點（在 DOS 環境為 *main*，在 Windows 環境為 *WinMain*）更早。所以 *theApp* 的建構式將更早於 *main*。換句話說你所看到的執行結果中的那些建構式輸出動作全都是在 *main* 函式之前完成的。

main 函式呼叫全域函式 *AfxGetApp* 以取得 *theApp* 的物件指標。這完全是模擬 MFC 程式的手法。

MFC 程式的初始化過程

MFC 程式也是個 Windows 程式，它的內部一定也像第 1 章所述一樣，有視窗註冊動作，有視窗產生動作，有訊息迴路動作，也有視窗函式。此刻我並不打算做出 Windows 程式，只是想交待給你一個程式流程，這個流程正是任何 MFC 程式的初始化過程的簡化。

以下是 *Frame2* 範例程式的類別階層及其成員。對於那些「除了建構式與解構式之外沒有其他成員」的類別，我就不再圖中展開他們了：



(本圖從 Visual C++ 的「Class View 視窗」中獲得)

就如我曾在第 1 章解釋過的，*InitApplication* 和 *InitInstance* 現在成了 MFC 的 *CWinApp* 的兩個虛擬函式。前者負責「每一個程式只做一次」的動作，後者負責「每一個執行個

體都得做一次」的動作。通常，系統會（並且有能力）為你註冊一些標準的視窗類別（當然也就準備好了一些標準的視窗函式），你（應用程式設計者）應該在你的 *CMyWinApp* 中改寫 *InitInstance*，並在其中把視窗產生出來 -- 這樣你才有機會在標準的視窗類別中指定自己的視窗標題和選單。下面就是我們新的 *main* 函式：

```
// MY.CPP
CMyWinApp theApp;
void main()
{
    CWinApp* pApp = AfxGetApp();

    pApp->InitApplication();
    pApp->InitInstance();
    pApp->Run();
}
```

其中 *pApp* 指向 *theApp* 全域物件。在這裡我們開始看到了虛擬函式的妙用（還不熟練者請快複習第 2 章）：

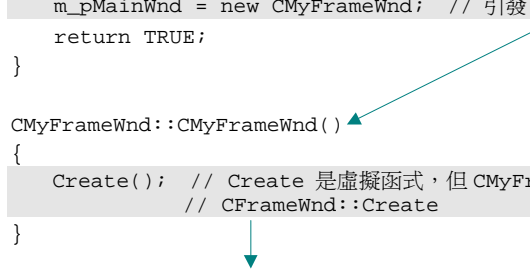
- *pApp->InitApplication()* 呼叫的是 *CWinApp::InitApplication*，
- *pApp->InitInstance()* 呼叫的是 *CMyWinApp::InitInstance*（因為 *CMyWinApp* 改寫它了），
- *pApp->Run()* 呼叫的是 *CWinApp::Run*，

好，請注意以下 *CMyWinApp::InitInstance* 的動作，以及它所引發的行為：

```
BOOL CMyWinApp::InitInstance()
{
    cout << "CMyWinApp::InitInstance \n";
    m_pMainWnd = new CMyFrameWnd; // 引發 CMyFrameWnd::CMyFrameWnd 建構式
    return TRUE;
}

CMyFrameWnd::CMyFrameWnd()
{
    Create(); // Create 是虛擬函式，但 CMyFrameWnd 未改寫它，所以引發父類別的
             // CFrameWnd::Create
}

BOOL CFrameWnd::Create()
{
```



```

    cout << "CFrameWnd::Create \n";
    CreateEx(); // CreateEx 是虛擬函式，但 CFrameWnd 未改寫之，所以引發
               // CWnd::CreateEx
    return TRUE;
}

BOOL CWnd::CreateEx()
{
    cout << "CWnd::CreateEx \n";
    PreCreateWindow(); // 這是一個虛擬函式，CWnd 中有定義，CFrameWnd 也改寫了
                      // 它。那麼你說這裡到底是呼叫 CWnd::PreCreateWindow 還是
                      // CFrameWnd::PreCreateWindow 呢？
    return TRUE;
}

BOOL CFrameWnd::PreCreateWindow()
{
    cout << "CFrameWnd::PreCreateWindow \n";
    return TRUE;
}

```

答案是 CFrameWnd::PreCreateWindow。
這便是在第2章的「Object slicing 與虛擬函式」一節所說提「虛擬函式的一個極重要的行為模式」。

你看到了，這些函式什麼正經事兒也沒做，光只輸出一個識別字串。我主要的目的是在讓你先熟悉 MFC 程式的執行流程。

Frame2 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第4章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

以下就是 Frame2 的執行結果：

```

CWinApp::InitApplication
CMyWinApp::InitInstance
CMyFrameWnd::CMyFrameWnd
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run

```

Frame2 範例程式

MFC.H

```
#0001 #define BOOL int
#0002 #define TRUE 1
#0003 #define FALSE 0
#0004
#0005 #include <iostream.h>
#0006
#0007 class CObject
#0008 {
#0009 public:
#0010     CObject::CObject() { }
#0011     CObject::~~CObject() { }
#0012 };
#0013
#0014 class CCmdTarget : public CObject
#0015 {
#0016 public:
#0017     CCmdTarget::CCmdTarget() { }
#0018     CCmdTarget::~~CCmdTarget() { }
#0019 };
#0020
#0021 class CWinThread : public CCmdTarget
#0022 {
#0023 public:
#0024     CWinThread::CWinThread() { }
#0025     CWinThread::~~CWinThread() { }
#0026
#0027     virtual BOOL InitInstance() {
#0028                                     cout << "CWinThread::InitInstance \n";
#0029                                     return TRUE;
#0030     }
#0031     virtual int Run() {
#0032                                     cout << "CWinThread::Run \n";
#0033                                     return 1;
#0034     }
#0035 };
#0036
#0037 class CWnd;
#0038
#0039 class CWinApp : public CWinThread
#0040 {
#0041 public:
```

```
#0042 CWinApp* m_pCurrentWinApp;
#0043 CWnd* m_pMainWnd;
#0044
#0045 public:
#0046 CWinApp::CWinApp() { m_pCurrentWinApp = this; }
#0047 CWinApp::~CWinApp() { }
#0048
#0049 virtual BOOL InitApplication() {
#0050     cout << "CWinApp::InitApplication \n";
#0051     return TRUE;
#0052 }
#0053 virtual BOOL InitInstance() {
#0054     cout << "CWinApp::InitInstance \n";
#0055     return TRUE;
#0056 }
#0057 virtual int Run() {
#0058     cout << "CWinApp::Run \n";
#0059     return CWinThread::Run();
#0060 }
#0061 };
#0062
#0063
#0064 class CDocument : public CCmdTarget
#0065 {
#0066 public:
#0067     CDocument::CDocument() { }
#0068     CDocument::~CDocument() { }
#0069 };
#0070
#0071
#0072 class CWnd : public CCmdTarget
#0073 {
#0074 public:
#0075     CWnd::CWnd() { }
#0076     CWnd::~CWnd() { }
#0077
#0078     virtual BOOL Create();
#0079     BOOL CreateEx();
#0080     virtual BOOL PreCreateWindow();
#0081 };
#0082
#0083 class CFrameWnd : public CWnd
#0084 {
#0085 public:
#0086     CFrameWnd::CFrameWnd() { }
#0087     CFrameWnd::~CFrameWnd() { }
```



```
#0088     BOOL Create();
#0089     virtual BOOL PreCreateWindow();
#0090 };
#0091
#0092 class CView : public CWnd
#0093 {
#0094 public:
#0095     CView::CView() { }
#0096     CView::~CView() { }
#0097 };
#0098
#0099
#0100 // global function
#0101 CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h" // 原該含入 mfc.h 就好，但爲了 CMyWinApp 的定義，所以...
#0002
#0003 extern CMyWinApp theApp; // external global object
#0004
#0005 BOOL CWnd::Create()
#0006 {
#0007     cout << "CWnd::Create \n";
#0008     return TRUE;
#0009 }
#0010
#0011 BOOL CWnd::CreateEx()
#0012 {
#0013     cout << "CWnd::CreateEx \n";
#0014     PreCreateWindow();
#0015     return TRUE;
#0016 }
#0017
#0018 BOOL CWnd::PreCreateWindow()
#0019 {
#0020     cout << "CWnd::PreCreateWindow \n";
#0021     return TRUE;
#0022 }
#0023
#0024 BOOL CFrameWnd::Create()
#0025 {
#0026     cout << "CFrameWnd::Create \n";
#0027     CreateEx();
#0028     return TRUE;
#0029 }
```

```
#0030
#0031 BOOL CFrameWnd::PreCreateWindow()
#0032 {
#0033     cout << "CFrameWnd::PreCreateWindow \n";
#0034     return TRUE;
#0035 }
#0036
#0037
#0038 CWinApp* AfxGetApp()
#0039 {
#0040     return theApp.m_pCurrentWinApp;
#0041 }
```

MY.H

```
#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp():CMyWinApp() { }
#0008     CMyWinApp::~CMyWinApp() { }
#0009
#0010     virtual BOOL InitInstance();
#0011 };
#0012
#0013 class CMyFrameWnd : public CFrameWnd
#0014 {
#0015 public:
#0016     CMyFrameWnd();
#0017     ~CMyFrameWnd() { }
#0018 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp; // global object
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
```

```
#0010 }
#0011
#0012 CMyFrameWnd::CMyFrameWnd()
#0013 {
#0014     cout << "CMyFrameWnd::CMyFrameWnd \n";
#0015     Create();
#0016 }
#0017
#0018 //-----
#0019 // main
#0020 //-----
#0021 void main()
#0022 {
#0023
#0024     CWinApp* pApp = AfxGetApp();
#0025
#0026     pApp->InitApplication();
#0027     pApp->InitInstance();
#0028     pApp->Run();
#0029 }
```

RTTI（執行時期型別辨識）

你已經在第 2 章看到，Visual C++ 4.0 支援 RTTI，重點不外乎是：

1. 編譯時需選用 /GR 選項（/GR 的意思是 enable C++ RTTI）
2. 含入 `typeinfo.h`
3. 使用新的 `typeid` 運算子。

RTTI 亦有稱為 Runtime Type Identification 者。

MFC 早在編譯器支援 RTTI 之前，就有了這項能力。我們現在要以相同的手法，在 DOS 程式中模擬出來。我希望我的類別庫具備 *IsKindOf* 的能力，能在執行時期偵測某個物件是否「屬於某種類別」，並傳回 *TRUE* 或 *FALSE*。以前一章的 *Shape* 為例，我希望：

```
CSquare* pSquare = new CSquare;
cout << pSquare->IsKindOf(CSquare);    // 應該獲得 1 (TRUE)
cout << pSquare->IsKindOf(CRect);      // 應該獲得 0 (FALSE)
cout << pSquare->IsKindOf(CShape);     // 應該獲得 1 (TRUE)
```

```
cout << pSquare->IsKindOf(CCircle); // 應該獲得 0 (FALSE)
```

以 MFC 的類別階層來說，我希望：

注意：真正的 *IsKindOf* 參數其實沒那麼單純

```
CMyDoc* pMyDoc = new CMyDoc;
cout << pMyDoc->IsKindOf(CMyDoc); // 應該獲得 1 (TRUE)
cout << pMyDoc->IsKindOf(CDocument); // 應該獲得 1 (TRUE)
cout << pMyDoc->IsKindOf(CCmdTarget); // 應該獲得 1 (TRUE)
cout << pMyDoc->IsKindOf(CWnd); // 應該獲得 0 (FALSE)
```

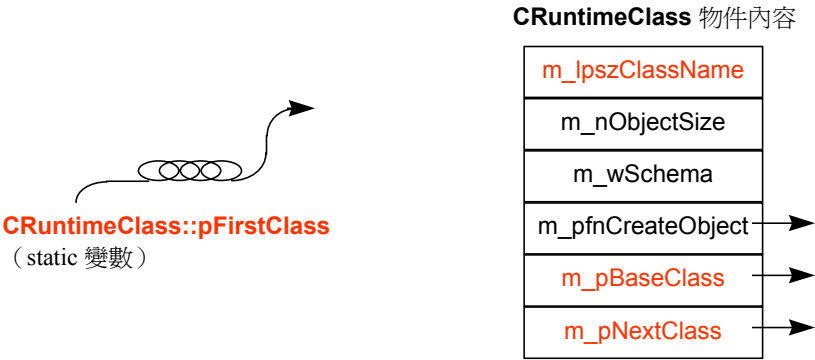
類別型錄與 CRuntimeClass

怎麼設計 RTTI 呢？讓我們想想，當你手上握有一種色澤，想知道它的 RGB 成份比，不查色表行嗎？當你持有一種產品，想知道它的型號，不查型錄行嗎？要達到 RTTI 的能力，我們（類別庫的設計者）一定要在類別建構起來的時候，記錄必要的資訊，以建立型錄。型錄中的類別資訊，最好以串列（linked list）方式串接起來，將來方便一一比對。

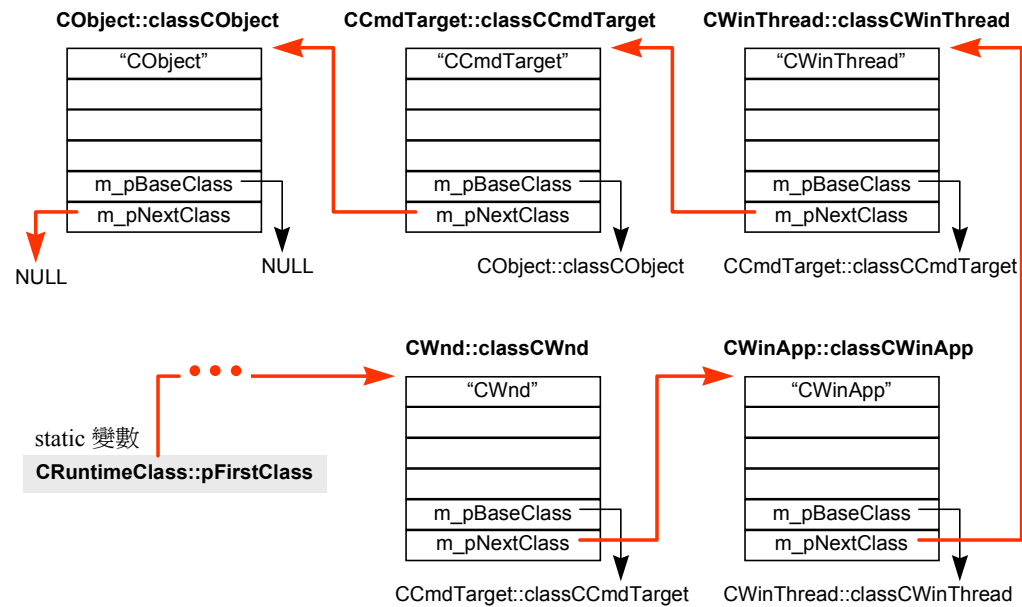
我們這份「類別型錄」的串列元素將以 *CRuntimeClass* 描述之，那是一個結構，內中至少需有類別名稱、串列的 Next 指標，以及串列的 First 指標。由於 First 指標屬於全域變數，一份就好，所以它應該以 static 修飾之。除此之外你所看到的其他 *CRuntimeClass* 成員都是為了其他目的而準備，陸陸續續我會介紹出來。

```
// in MFC.H
struct CRuntimeClass
{
    // Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema; // schema number of the loaded class
    CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
    CRuntimeClass* m_pBaseClass;

    // CRuntimeClass objects linked together in simple list
    static CRuntimeClass* pFirstClass; // start of class list
    CRuntimeClass* m_pNextClass; // linked list of registered classes
};
```



我希望，每一個類別都能擁有這樣一個 *CRuntimeClass* 成員變數，並且最好有一定的命名規則（例如在類別名稱之前冠以 "class" 作為它的名稱），然後，經由某種手段將整個類別庫建構好之後，「類別型錄」能呈現類似這樣的風貌：



DECLARE_DYNAMIC / IMPLEMENT_DYNAMIC 巨集

爲了神不知鬼不覺把 *CRuntimeClass* 物件塞到類別之中，並宣告一個可以抓到該物件位址的函式，我們定義 *DECLARE_DYNAMIC* 巨集如下：

```
#define DECLARE_DYNAMIC(class_name) \
public: \
    static CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const;
```

出現在巨集定義之中的 *##*，用來告訴編譯器，把兩個字串繫在一起。如果你這麼使用此巨集：

```
DECLARE_DYNAMIC(CView)
```

編譯器前置處理器爲你做出的碼是：

```
public:
    static CRuntimeClass classCView;
    virtual CRuntimeClass* GetRuntimeClass() const;
```

這下子，只要在宣告類別時放入 *DECLARE_DYNAMIC* 巨集即萬事 OK 嘍。

不，還沒有 OK，類別型錄（也就是各個 *CRuntimeClass* 物件）的內容指定以及串接工作最好也能夠神不知鬼不覺，我們於是再定義 *IMPLEMENT_DYNAMIC* 巨集：

```
#define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
```

其中的 *_IMPLEMENT_RUNTIMECLASS* 又是一個巨集。這樣區分是爲了此一巨集在「動態生成」（下一節主題）時還會用到。

```
#define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
    static char _lpsz##class_name[] = #class_name; \
    CRuntimeClass class_name::class##class_name = { \
        _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
        RUNTIME_CLASS(base_class_name), NULL }; \
    static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
    CRuntimeClass* class_name::GetRuntimeClass() const \
    { return &class_name::class##class_name; } \
```

其中又有 *RUNTIME_CLASS* 巨集，定義如下：

```
#define RUNTIME_CLASS(class_name) \
    (&class_name::class##class_name)
```

看起來整個 *IMPLEMENT_DYNAMIC* 內容好像只是指定初值，不然，其曼妙處在於它所使用的一個 struct *AFX_CLASSINIT*，定義如下：

```
struct AFX_CLASSINIT
{ AFX_CLASSINIT(CRuntimeClass* pNewClass); };
```

這表示它有一個建構式（別驚訝，C++ 的 struct 與 class 都有建構式），定義如下：

```
AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
{
    pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
    CRuntimeClass::pFirstClass = pNewClass;
}
```

很明顯，此建構式負責 linked list 的串接工作。

整組巨集看起來有點嚇人，其實也沒有什麼，文字代換而已。現在看看這個實例：

```
// in header file
class CView : public CWnd
{
    DECLARE_DYNAMIC(CView)
    ...
};

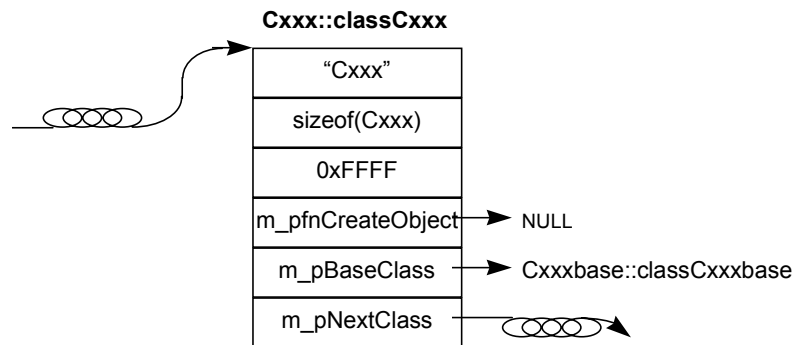
// in implementation file
IMPLEMENT_DYNAMIC(CView, CWnd)
```

上述的碼展開來成為：

```
// in header file
class CView : public CWnd
{
public:
    static CRuntimeClass classCView; \
    virtual CRuntimeClass* GetRuntimeClass() const;
    ...
};
```

```
// in implementation file
static char _lpzCView[] = "CView";
CRuntimeClass CView::classCView = {
    _lpzCView, sizeof(CView), 0xFFFF, NULL,
    &CWnd::classCWnd, NULL };
static AFX_CLASSINIT _init_CView(&CView::classCView);
CRuntimeClass* CView::GetRuntimeClass() const
{ return &CView::classCView; }
```

於是乎，程式中只需要簡簡單單的兩個巨集 *DECLARE_DYNAMIC(Cxxx)* 和 *IMPLEMENT_DYNAMIC(Cxxx, Cxxxbase)*，就完成了建構資料並加入串列的工作：



可是你知道，串列的頭，總是需要特別費心處理，不能夠套用一般的串列行為模式。我們的類別根源 *CObject*，不能套用現成的巨集 *DECLARE_DYNAMIC* 和 *IMPLEMENT_DYNAMIC*，必須特別設計如下：

```
// in header file
class CObject
{
public:
    virtual CRuntimeClass* GetRuntimeClass() const;
    ...
public:
    static CRuntimeClass classCObject;
};
```



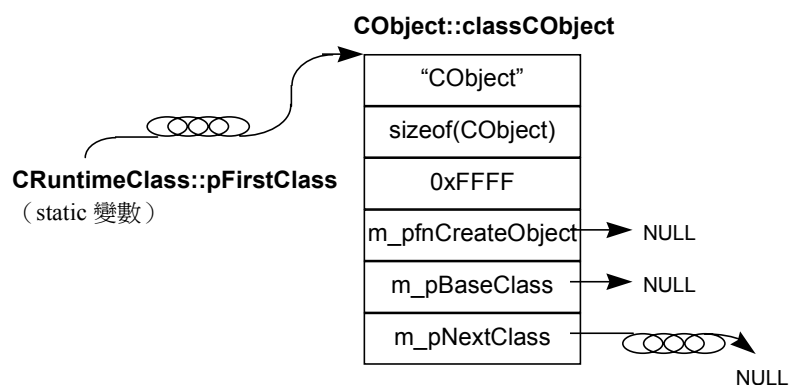
```
// in implementation file
static char szCObject[] = "CObject";
struct CRuntimeClass CObject::classCObject =
    { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
static AFX_CLASSINIT _init_CObject(&CObject::classCObject);

CRuntimeClass* CObject::GetRuntimeClass() const
{
    return &CObject::classCObject;
}
```

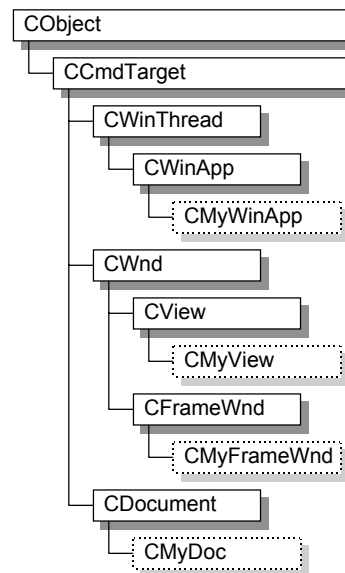
並且，*CRuntimeClass* 中的 *static* 成員變數應該要初始化（如果你忘記了，趕快複習第 2 章的「靜態成員（變數與函式）」一節）：

```
// in implementation file
CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
```

終於，整個「類別型錄」串列的頭部就這樣形成了：



範例程式 Frame3 在 .h 檔中有這些類別宣告：



```

class CObject
{
...
};
class CCmdTarget : public CObject
{
    DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
    DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
    DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocument)

```

```
...
};
class CWnd : public CCmdTarget
{
    DECLARE_DYNAMIC(CWnd) // 其實在 MFC 中是 DECLARE_DYNCREATE(), 見下節。
...
};
class CFrameWnd : public CWnd
{
    DECLARE_DYNAMIC(CFrameWnd) // 其實在 MFC 中是 DECLARE_DYNCREATE(), 見下節。
...
};
class CView : public CWnd
{
    DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
{
... // 其實在 MFC 應用程式中這裡也有 DECLARE_DYNCREATE(), 見下節。
};
class CMyDoc : public CDocument
{
... // 其實在 MFC 應用程式中這裡也有 DECLARE_DYNCREATE(), 見下節。
};
class CMyView : public CView
{
... // 其實在 MFC 應用程式中這裡也有 DECLARE_DYNCREATE(), 見下節。
};
```

範例程式 Frame3 在 .cpp 檔中有這些動作：

```
IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNAMIC(CWnd, CCmdTarget) // 其實在 MFC 中它是 IMPLEMENT_DYNCREATE(), 見下節。
IMPLEMENT_DYNAMIC(CFrameWnd, CWnd) // 其實在 MFC 中它是 IMPLEMENT_DYNCREATE(), 見下節。
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
```

於是組織出圖 3-1 這樣一個大綱。

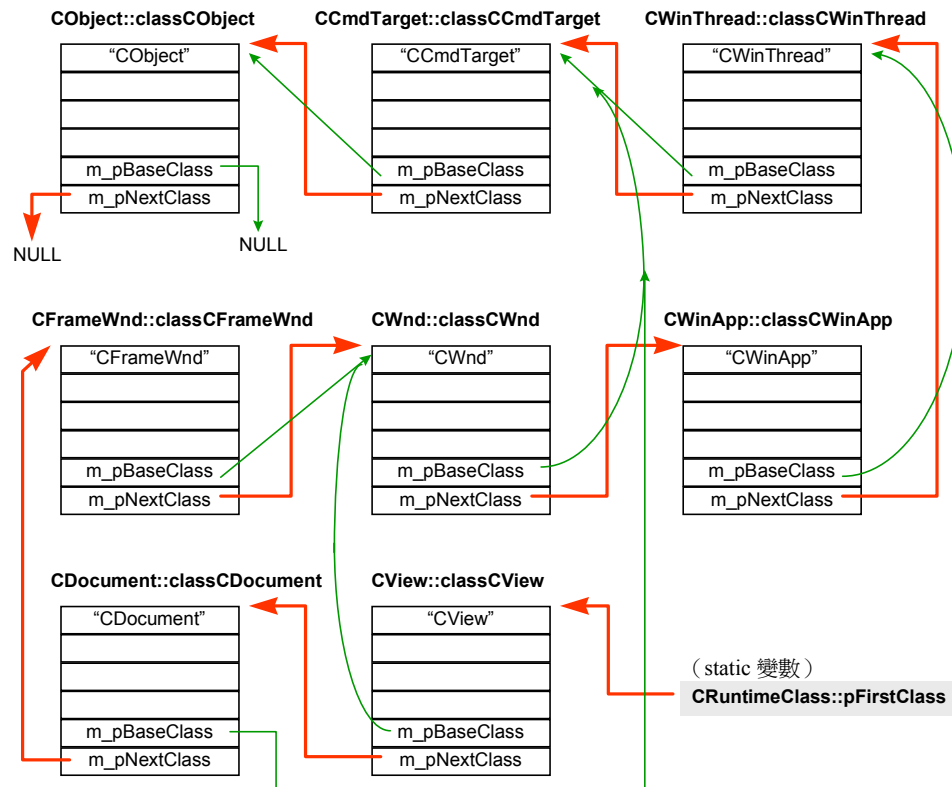


圖 3-1 **CRuntimeClass** 物件構成的類別型錄網。本圖只列出與 RTTI 有關係的成員。

爲了實證整個類別型錄網的存在，我在 `main` 函式中呼叫 `PrintAllClasses`，把串列中的每一個元素的類別名稱、物件大小、以及 `schema no.` 印出來：

```
void PrintAllClasses()
{
    CRuntimeClass* pClass;

    // just walk through the simple list of registered classes
    for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
         pClass = pClass->m_pNextClass)
    {
        cout << pClass->m_lpszClassName << "\n";
    }
}
```

```
        cout << pClass->m_nObjectSize << "\n";  
        cout << pClass->m_wSchema << "\n";  
    }  
}
```

Frame3 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第 4 章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

Frame3 的執行結果如下：

```
CView  
4  
65535  
CDocument  
4  
65535  
CFrameWnd  
4  
65535  
CWnd  
4  
65535  
CWinApp  
12  
65535  
CWinThread  
4  
65535  
CCmdTarget  
4  
65535  
CObject  
4  
65535
```

Frame3 範例程式

MFC.H

```
#0001 #define BOOL int  
#0002 #define TRUE 1  
#0003 #define FALSE 0
```

```

#0004 #define LPCSTR LPSTR
#0005 typedef char* LPSTR;
#0006 #define UINT int
#0007 #define PASCAL _stdcall
#0008
#0009 #include <iostream.h>
#0010
#0011 class CObject;
#0012
#0013 struct CRuntimeClass
#0014 {
#0015     // Attributes
#0016     LPCSTR m_lpszClassName;
#0017     int m_nObjectSize;
#0018     UINT m_wSchema; // schema number of the loaded class
#0019     CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0020     CRuntimeClass* m_pBaseClass;
#0021
#0022     // CRuntimeClass objects linked together in simple list
#0023     static CRuntimeClass* pFirstClass; // start of class list
#0024     CRuntimeClass* m_pNextClass; // linked list of registered classes
#0025 };
#0026
#0027 struct AFX_CLASSINIT
#0028 { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0029
#0030 #define RUNTIME_CLASS(class_name) \
#0031     (&class_name::class##class_name)
#0032
#0033 #define DECLARE_DYNAMIC(class_name) \
#0034     public: \
#0035         static CRuntimeClass class##class_name; \
#0036         virtual CRuntimeClass* GetRuntimeClass() const;
#0037
#0038 #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0039     static char _lpsz##class_name[] = #class_name; \
#0040     CRuntimeClass class_name::class##class_name = { \
#0041         _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0042         RUNTIME_CLASS(base_class_name), NULL }; \
#0043     static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0044     CRuntimeClass* class_name::GetRuntimeClass() const \
#0045     { return &class_name::class##class_name; } \
#0046
#0047 #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0048     _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0049

```

```
#0050 class CObject
#0051 {
#0052 public:
#0053     CObject::CObject() {
#0054     }
#0055     CObject::~~CObject() {
#0056     }
#0057
#0058     virtual CRuntimeClass* GetRuntimeClass() const;
#0059
#0060 public:
#0061     static CRuntimeClass classCObject;
#0062 };
#0063
#0064 class CCmdTarget : public CObject
#0065 {
#0066     DECLARE_DYNAMIC(CCmdTarget)
#0067 public:
#0068     CCmdTarget::CCmdTarget() {
#0069     }
#0070     CCmdTarget::~~CCmdTarget() {
#0071     }
#0072 };
#0073
#0074 class CWinThread : public CCmdTarget
#0075 {
#0076     DECLARE_DYNAMIC(CWinThread)
#0077 public:
#0078     CWinThread::CWinThread() {
#0079     }
#0080     CWinThread::~~CWinThread() {
#0081     }
#0082
#0083     virtual BOOL InitInstance() {
#0084         return TRUE;
#0085     }
#0086     virtual int Run() {
#0087         return 1;
#0088     }
#0089 };
#0090
#0091 class CWnd;
#0092
#0093 class CWinApp : public CWinThread
#0094 {
#0095     DECLARE_DYNAMIC(CWinApp)
```

```
#0096 public:
#0097     CWinApp* m_pCurrentWinApp;
#0098     CWnd* m_pMainWnd;
#0099
#0100 public:
#0101     CWinApp::CWinApp() {
#0102         m_pCurrentWinApp = this;
#0103     }
#0104     CWinApp::~CWinApp() {
#0105     }
#0106
#0107     virtual BOOL InitApplication() {
#0108         return TRUE;
#0109     }
#0110     virtual BOOL InitInstance() {
#0111         return TRUE;
#0112     }
#0113     virtual int Run() {
#0114         return CWinThread::Run();
#0115     }
#0116 };
#0117
#0118 class CDocument : public CCmdTarget
#0119 {
#0120     DECLARE_DYNAMIC(CDocument)
#0121 public:
#0122     CDocument::CDocument() {
#0123     }
#0124     CDocument::~CDocument() {
#0125     }
#0126 };
#0127
#0128 class CWnd : public CCmdTarget
#0129 {
#0130     DECLARE_DYNAMIC(CWnd)
#0131 public:
#0132     CWnd::CWnd() {
#0133     }
#0134     CWnd::~CWnd() {
#0135     }
#0136
#0137     virtual BOOL Create();
#0138     BOOL CreateEx();
#0139     virtual BOOL PreCreateWindow();
#0140 };
#0141
```



```
#0142 class CFrameWnd : public CWnd
#0143 {
#0144     DECLARE_DYNAMIC(CFrameWnd)
#0145 public:
#0146     CFrameWnd::CFrameWnd() {
#0147     }
#0148     CFrameWnd::~~CFrameWnd() {
#0149     }
#0150     BOOL Create();
#0151     virtual BOOL PreCreateWindow();
#0152 };
#0153
#0154 class CView : public CWnd
#0155 {
#0156     DECLARE_DYNAMIC(CView)
#0157 public:
#0158     CView::CView() {
#0159     }
#0160     CView::~~CView() {
#0161     }
#0162 };
#0163
#0164
#0165 // global function
#0166 CWinApp* AfxGetApp();
```

MFC.CPP

```
#0001 #include "my.h" // 原該含入 mfc.h 就好，但爲了 CMyWinApp 的定義，所以...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 static char szCObject[] = "CObject";
#0006 struct CRuntimeClass CObject::classCObject =
#0007     { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008 static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010 CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012 AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013 {
#0014     pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015     CRuntimeClass::pFirstClass = pNewClass;
#0016 }
#0017
#0018 CRuntimeClass* CObject::GetRuntimeClass() const
```

```
#0019 {  
#0020     return &CObject::classCObject;  
#0021 }  
#0022  
#0023 BOOL CWnd::Create()  
#0024 {  
#0025     return TRUE;  
#0026 }  
#0027  
#0028 BOOL CWnd::CreateEx()  
#0029 {  
#0030     PreCreateWindow();  
#0031     return TRUE;  
#0032 }  
#0033  
#0034 BOOL CWnd::PreCreateWindow()  
#0035 {  
#0036     return TRUE;  
#0037 }  
#0038  
#0039 BOOL CFrameWnd::Create()  
#0040 {  
#0041     CreateEx();  
#0042     return TRUE;  
#0043 }  
#0044  
#0045 BOOL CFrameWnd::PreCreateWindow()  
#0046 {  
#0047     return TRUE;  
#0048 }  
#0049  
#0050 IMPLEMENT_DYNAMIC(CCmdTarget, CObject)  
#0051 IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)  
#0052 IMPLEMENT_DYNAMIC(CWinApp, CWinThread)  
#0053 IMPLEMENT_DYNAMIC(CWnd, CCmdTarget)  
#0054 IMPLEMENT_DYNAMIC(CFrameWnd, CWnd)  
#0055 IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)  
#0056 IMPLEMENT_DYNAMIC(CView, CWnd)  
#0057  
#0058 // global function  
#0059 CWinApp* AfxGetApp()  
#0060 {  
#0061     return theApp.m_pCurrentWinApp;  
#0062 }
```

MY.H

```
#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp::CMyWinApp() {
#0008     }
#0009     CMyWinApp::~CMyWinApp() {
#0010     }
#0011
#0012     virtual BOOL InitInstance();
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017 public:
#0018     CMyFrameWnd();
#0019     ~CMyFrameWnd() {
#0020     }
#0021 };
#0022
#0023
#0024 class CMyDoc : public CDocument
#0025 {
#0026 public:
#0027     CMyDoc::CMyDoc() {
#0028     }
#0029     CMyDoc::~CMyDoc() {
#0030     }
#0031 };
#0032
#0033 class CMyView : public CView
#0034 {
#0035 public:
#0036     CMyView::CMyView() {
#0037     }
#0038     CMyView::~CMyView() {
#0039     }
#0040 };
#0041
#0042 // global function
#0043 void PrintAllClasses();
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     m_pMainWnd = new CMyFrameWnd;
#0008     return TRUE;
#0009 }
#0010
#0011 CMyFrameWnd::CMyFrameWnd()
#0012 {
#0013     Create();
#0014 }
#0015
#0016 void PrintAllClasses()
#0017 {
#0018     CRuntimeClass* pClass;
#0019
#0020     // just walk through the simple list of registered classes
#0021     for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0022          pClass = pClass->m_pNextClass)
#0023     {
#0024         cout << pClass->m_lpszClassName << "\n";
#0025         cout << pClass->m_nObjectSize << "\n";
#0026         cout << pClass->m_wSchema << "\n";
#0027     }
#0028 }
#0029 //-----
#0030 // main
#0031 //-----
#0032 void main()
#0033 {
#0034     CWinApp* pApp = AfxGetApp();
#0035
#0036     pApp->InitApplication();
#0037     pApp->InitInstance();
#0038     pApp->Run();
#0039
#0040     PrintAllClasses();
#0041 }
```

IsKindOf (類別辨識)

有了圖 3-1 這張「類別型錄」網，要實現 *IsKindOf* 功能，再輕鬆不過了：

1. 為 *CObject* 加上一個 *IsKindOf* 函式，於是此函式將被所有類別繼承。它將把參數所指定的某個 *CRuntimeClass* 物件拿來與類別型錄中的元素一一比對。比對成功（在型錄中有發現），就傳回 *TRUE*，否則傳回 *FALSE*：

```
// in header file
class CObject
{
public:
    ...
    BOOL IsKindOf(const CRuntimeClass* pClass) const;
};

// in implementation file
BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
{
    CRuntimeClass* pClassThis = GetRuntimeClass();
    while (pClassThis != NULL)
    {
        if (pClassThis == pClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;    // walked to the top, no match
}
```

注意，*while* 迴路中所追蹤的是「同宗」路線，也就是憑藉著 *m_pBaseClass* 而非 *m_pNextClass*。假設我們的呼叫是：

```
CView* pView = new CView;
pView->IsKindOf(RUNTIME_CLASS(CWinApp));
```

IsKindOf 的參數其實就是 *&CWinApp::classCWinApp*。函式內利用 *GetRuntimeClass* 先取得 *&CView::classCView*，然後循線而上（從圖 3-1 來看，所謂循線分別是指 *CView*、*CWnd*、*CCmdTarget*、*CObject*），每獲得一個 *CRuntimeClass* 物件指標，就拿來和 *CView::classCView* 的指標比對。靠這個土方法，完成了 *IsKindOf* 能力。

2. *IsKindOf* 的使用方式如下：

```

CMyDoc* pMyDoc = new CMyDoc;
CMyView* pMyView = new CMyView;

cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc)); // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument)); // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget)); // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CObject)); // 應該獲得 TRUE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp)); // 應該獲得 FALSE
cout << pMyDoc->IsKindOf(RUNTIME_CLASS(CView)); // 應該獲得 FALSE

cout << pMyView->IsKindOf(RUNTIME_CLASS(CView)); // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CObject)); // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CWnd)); // 應該獲得 TRUE
cout << pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd)); // 應該獲得 FALSE

```

IsKindOf 的完整範例放在 Frame4 中。

Frame4 範例程式

Frame4 與 Frame3 大同小異，唯一不同的就是前面所說的，在 *CObject* 中加上 *IsKindOf* 函式的宣告與定義，並將私有類別（non-MFC 類別）也掛到「類別型錄網」中：

```

// in header file
class CMyFrameWnd : public CFrameWnd
{
    DECLARE_DYNAMIC(CMyFrameWnd) // 在 MFC 程式中這裡其實是 DECLARE_DYNCREATE()
    ...                          // 稍後我便會模擬 DECLARE_DYNCREATE() 給你看
};

class CMyDoc : public CDocument
{
    DECLARE_DYNAMIC(CMyDoc) // 在 MFC 程式中這裡其實是 DECLARE_DYNCREATE()
    ...                      // 稍後我便會模擬 DECLARE_DYNCREATE() 給你看
};

class CMyView : public CView
{
    DECLARE_DYNAMIC(CMyView) // 在 MFC 程式中這裡其實是 DECLARE_DYNCREATE()
    ...                      // 稍後我便會模擬 DECLARE_DYNCREATE() 給你看
};

```

```
// in implementation file
...
IMPLEMENT_DYNAMIC(CMyFrameWnd, CFrameWnd) // 在 MFC 程式中這裡其實是 IMPLEMENT_DYNCREATE()
// 稍後我便會模擬 IMPLEMENT_DYNCREATE() 給你看
...
IMPLEMENT_DYNAMIC(CMyDoc, CDocument) // 在 MFC 程式中這裡其實是 IMPLEMENT_DYNCREATE()
// 稍後我便會模擬 IMPLEMENT_DYNCREATE() 給你看
...
IMPLEMENT_DYNAMIC(CMyView, CView) // 在 MFC 程式中這裡其實是 IMPLEMENT_DYNCREATE()
// 稍後我便會模擬 IMPLEMENT_DYNCREATE() 給你看
```

我不在此列出 Frame4 的原始碼，你可以在書附光碟片中找到完整的檔案。Frame4 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第 4 章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

以下即是 Frame4 的執行結果：

```
pMyDoc->IsKindOf(RUNTIME_CLASS(CMyDoc))      1
pMyDoc->IsKindOf(RUNTIME_CLASS(CDocument))    1
pMyDoc->IsKindOf(RUNTIME_CLASS(CCmdTarget))   1
pMyDoc->IsKindOf(RUNTIME_CLASS(CObject))      1
pMyDoc->IsKindOf(RUNTIME_CLASS(CWinApp))      0
pMyDoc->IsKindOf(RUNTIME_CLASS(CView))        0

pMyView->IsKindOf(RUNTIME_CLASS(CView))       1
pMyView->IsKindOf(RUNTIME_CLASS(CObject))     1
pMyView->IsKindOf(RUNTIME_CLASS(CWnd))        1
pMyView->IsKindOf(RUNTIME_CLASS(CFrameWnd))   0

pMyWnd->IsKindOf(RUNTIME_CLASS(CFrameWnd))    1
pMyWnd->IsKindOf(RUNTIME_CLASS(CWnd))         1
pMyWnd->IsKindOf(RUNTIME_CLASS(CObject))      1
pMyWnd->IsKindOf(RUNTIME_CLASS(CDocument))    0
```

Dynamic Creation (動態生成)

基礎有了，做什麼都好。同樣地，有了上述的「類別型錄網」，各種應用紛至沓來。其中一個應用就是解決棘手的動態生成問題。

我已經在第二章描述過動態生成的困難點：你沒有辦法在程式執行期間，根據動態獲得的一個類別名稱（通常來自讀檔，但我將以螢幕輸入為例），要求程式產生一個物件。上述的「類別型錄網」雖然透露出解決此一問題的些微曙光，但是技術上還得加把勁兒。

如果我能夠把類別的大小記錄在類別型錄中，把建構函式（注意，這裡並非指 C++ 建構式，而是指即將出現的 *CRuntimeClass::CreateObject*）也記錄在類別型錄中，當程式在執行時期獲得一個類別名稱，它就可以在「類別型錄網」中找出對應的元素，然後呼叫其建構函式（這裡並非指 C++ 建構式），產生出物件。

好主意！

類別型錄網的元素型式 *CRuntimeClass* 於是有了變化：

```
// in MFC.H
struct CRuntimeClass
{
// Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema; // schema number of the loaded class
    CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
    CRuntimeClass* m_pBaseClass;

    CObject* CreateObject();
    static CRuntimeClass* PASCAL Load();

// CRuntimeClass objects linked together in simple list
    static CRuntimeClass* pFirstClass; // start of class list
    CRuntimeClass* m_pNextClass;      // linked list of registered classes
};
```


DECLARE_DYNCREATE / IMPLEMENT_DYNCREATE 巨集

爲了因應 *CRuntimeClass* 中新增的成員變數，我們再添兩個巨集，*DECLARE_DYNCREATE* 和 *IMPLEMENT_DYNCREATE*：

```
#define DECLARE_DYNCREATE(class_name) \
    DECLARE_DYNAMIC(class_name) \
    static CObject* PASCAL CreateObject();

#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
    CObject* PASCAL class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
        class_name::CreateObject)
```

於是，以 *CFrameWnd* 爲例，下列程式碼：

```
// in header file
class CFrameWnd : public CWnd
{
    DECLARE_DYNCREATE(CFrameWnd)
    ...
};

// in implementation file
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
```

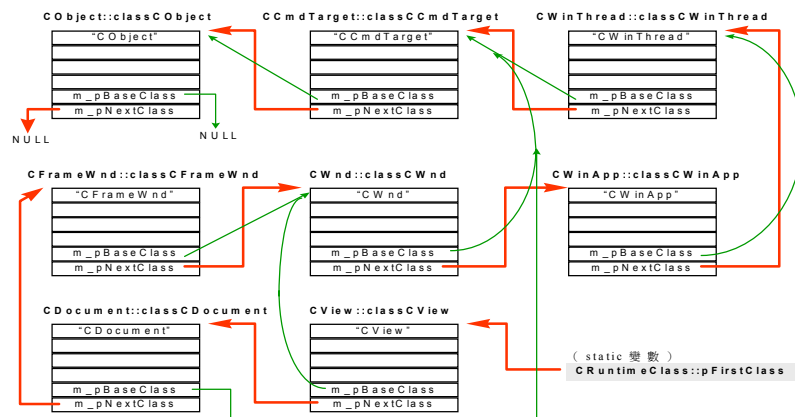
就被展開如下（注意，編譯器選項 /P 可得前置處理結果）：

```
// in header file
class CFrameWnd : public CWnd
{
public:
    static CRuntimeClass classCFrameWnd;
    virtual CRuntimeClass* GetRuntimeClass() const;
    static CObject* PASCAL CreateObject();
    ...
};

// in implementation file
CObject* PASCAL CFrameWnd::CreateObject()
{ return new CFrameWnd; }
```

```
static char _lpszCFrameWnd[] = "CFrameWnd";
CRuntimeClass CFrameWnd::classCFrameWnd = {
    _lpszCFrameWnd, sizeof(CFrameWnd), 0xFFFF, CFrameWnd::CreateObject,
    RUNTIME_CLASS(CWnd), NULL };
static AFX_CLASSINIT _init_CFrameWnd(&CFrameWnd::classCFrameWnd);
CRuntimeClass* CFrameWnd::GetRuntimeClass() const
{ return &CFrameWnd::classCFrameWnd; }
```

圖示如下：



「物件生成器」*CreateObject* 函式很簡單，只要說 *new* 就好。

從巨集的定義我們很清楚可以看出，擁有動態生成（Dynamic Creation）能力的類別庫，必然亦擁有執行時期型態識別（RTTI）能力，因為 *_DYNCREATE* 巨集涵蓋了 *_DYNAMIC* 巨集。

注意：以下範例直接跳到 Frame6。本書第一版有一個 Frame5 程式，用以模擬 MFC 2.5 對動態生成的作法。往事已矣，讀者曾經來函表示沒有必要提過去的東西，徒增腦力負荷。我想也是，況且 MFC 4.x 的作法更好更容易瞭解，所以我把 Frame5 拿掉了，但仍保留著序號。

範例程式 Frame6 在 .h 檔中有這些類別宣告：

```
class CObject
{
...
};
class CCmdTarget : public CObject
{
    DECLARE_DYNAMIC(CCmdTarget)
...
};
class CWinThread : public CCmdTarget
{
    DECLARE_DYNAMIC(CWinThread)
...
};
class CWinApp : public CWinThread
{
    DECLARE_DYNAMIC(CWinApp)
...
};
class CDocument : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocument)
...
};
class CWnd : public CCmdTarget
{
    DECLARE_DYNCREATE(CWnd)
...
};
class CFrameWnd : public CWnd
{
    DECLARE_DYNCREATE(CFrameWnd)
...
};
class CView : public CWnd
{
    DECLARE_DYNAMIC(CView)
...
};
class CMyWinApp : public CWinApp
{
...
};
class CMyFrameWnd : public CFrameWnd
```

```

{
    DECLARE_DYNCREATE(CMyFrameWnd)
    ...
};
class CMyDoc : public CDocument
{
    DECLARE_DYNCREATE(CMyDoc)
    ...
};
class CMyView : public CView
{
    DECLARE_DYNCREATE(CMyView)
    ...
};

```

在 .cpp 檔中又有這些動作：

```

IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)
IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
IMPLEMENT_DYNAMIC(CView, CWnd)
IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
IMPLEMENT_DYNCREATE(CMyView, CView)

```

於是組織出圖 3-2 這樣一個大綱。

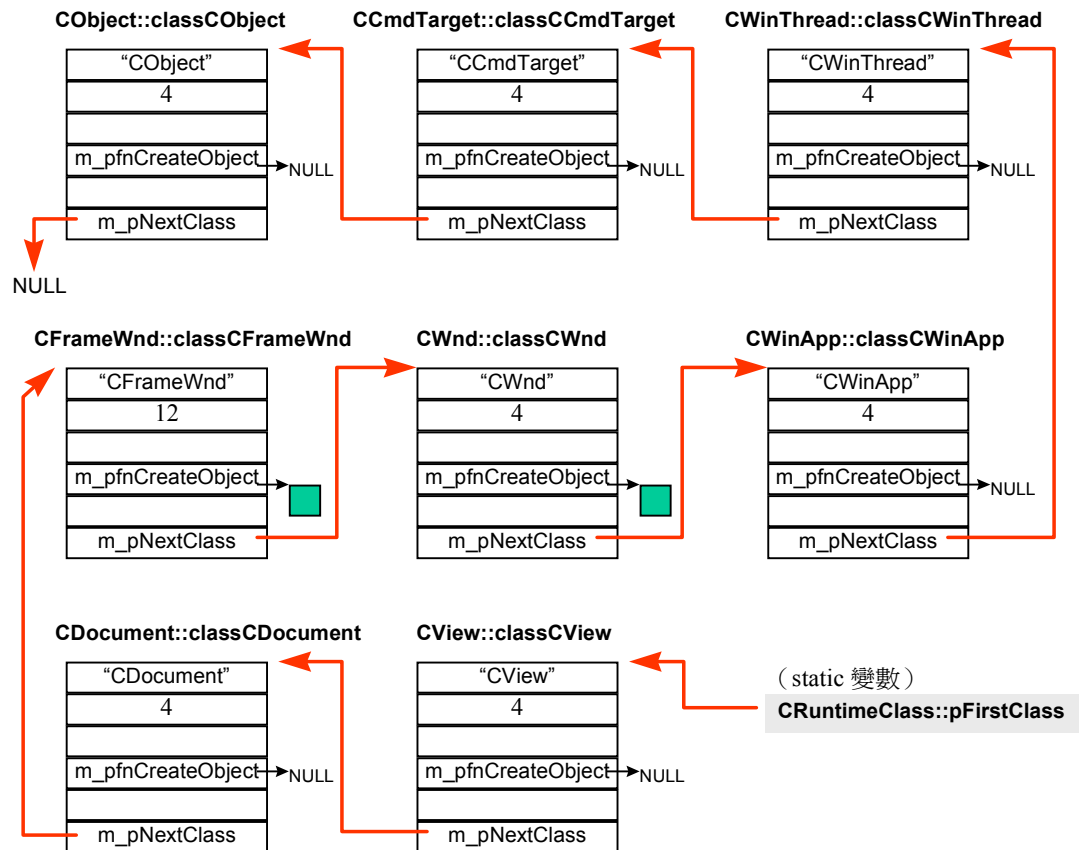


圖 3-2 以 **CRuntimeClass** 物件構成的「類別型錄網」。本圖只列出與動態生成 (Dynamic Creation) 有關係的成員。凡是 **m_pfnCreateObject** 不為 **NULL** 者，即可動態生成。

現在，我們開始模擬動態生成。首先在 **main** 函式中加上這一段碼：

```
void main()
{
    ...
    //Test Dynamic Creation
    CRuntimeClass* pClassRef;
    CObject* pOb;
    while(1)
    {
```

```

        if ((pClassRef = CRuntimeClass::Load()) == NULL)
            break;

        pObj = pClassRef->CreateObject();
        if (pObj != NULL)
            pObj->SayHello();
    }
}

```

並設計 *CRuntimeClass::CreateObject* 和 *CRuntimeClass::Load* 如下：

```

// in implementation file
CObject* CRuntimeClass::CreateObject()
{
    if (m_pfnCreateObject == NULL)
    {
        TRACE1("Error: Trying to create object which is not "
               "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
               m_lpszClassName);
        return NULL;
    }

    CObject* pObject = NULL;
    pObject = (*m_pfnCreateObject)();

    return pObject;
}

CRuntimeClass* PASCAL CRuntimeClass::Load()
{
    char szClassName[64];
    CRuntimeClass* pClass;
    // JJHOU : instead of Load from file, we Load from cin.
    cout << "enter a class name... ";
    cin >> szClassName;

    for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
    {
        if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
            return pClass;
    }

    TRACE1("Error: Class not found: %s \n", szClassName);
    return NULL; // not found
}

```

然後，爲了驗證這樣的動態生成機制的確有效（也就是物件的確被產生了），我讓許多個類別的建構式都輸出一段文字，而且在取得物件指標後，真的去呼叫該物件的一個成員函式 *SayHello*。我把 *SayHello* 設計爲虛擬函式，所以根據不同的物件型態，會呼叫到不同的 *SayHello* 函式，出現不同的輸出字串。

請注意，*main* 函式中的 *while* 迴路必須等到 *CRuntimeClass::Load* 傳回 *NULL* 才會停止，而 *CRuntimeClass::Load* 是在它從整個「類別型錄網」中找不到它要找的那個類別名稱時，才傳回 *NULL*。這些都是我爲了模擬與示範，所採取的權宜設計。

Frame6 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第 4 章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

下面是 Frame6 的執行結果。粗體表示我（程式執行者）在螢幕上輸入的類別名稱：

```
enter a class name... CObject
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CObject.

enter a class name... CView
Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CView.

enter a class name... CMyView
CWnd Constructor
CMyView Constructor
Hello CMyView

enter a class name... CMyFrameWnd
CWnd Constructor
CFrameWnd Constructor
CMyFrameWnd Constructor
Hello CMyFrameWnd

enter a class name... CMyDoc
CMyDoc Constructor
Hello CMyDoc
```

enter a class name... **CWinApp**

Error: Trying to create object which is not DECLARE_DYNCREATE
or DECLARE_SERIAL: CWinApp.

enter a class name... **CJjhou** (故意輸入一個不在「類別型錄網」中的類別名稱)

Error: Class not found: CJjhou (程式結束)

Frame6 範例程式

MFC.H

```
#0001 #define BOOL int
#0002 #define TRUE 1
#0003 #define FALSE 0
#0004 #define LPCSTR LPSTR
#0005 typedef char* LPSTR;
#0006 #define UINT int
#0007 #define PASCAL _stdcall
#0008 #define TRACE1 printf
#0009
#0010 #include <iostream.h>
#0011 #include <stdio.h>
#0012 #include <string.h>
#0013
#0014 class CObject;
#0015
#0016 struct CRuntimeClass
#0017 {
#0018     // Attributes
#0019     LPCSTR m_lpszClassName;
#0020     int m_nObjectSize;
#0021     UINT m_wSchema; // schema number of the loaded class
#0022     CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
#0023     CRuntimeClass* m_pBaseClass;
#0024
#0025     CObject* CreateObject();
#0026     static CRuntimeClass* PASCAL Load();
#0027
#0028     // CRuntimeClass objects linked together in simple list
#0029     static CRuntimeClass* pFirstClass; // start of class list
#0030     CRuntimeClass* m_pNextClass; // linked list of registered classes
#0031 };
#0032
#0033 struct AFX_CLASSINIT
```



```
#0034         { AFX_CLASSINIT(CRuntimeClass* pNewClass); };
#0035
#0036 #define RUNTIME_CLASS(class_name) \
#0037     (&class_name::class##class_name)
#0038
#0039 #define DECLARE_DYNAMIC(class_name) \
#0040 public: \
#0041     static CRuntimeClass class##class_name; \
#0042     virtual CRuntimeClass* GetRuntimeClass() const;
#0043
#0044 #define DECLARE_DYNCREATE(class_name) \
#0045     DECLARE_DYNAMIC(class_name) \
#0046     static COBJECT* PASCAL CreateObject();
#0047
#0048 #define _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, pfnNew) \
#0049     static char _lpsz##class_name[] = #class_name; \
#0050     CRuntimeClass class_name::class##class_name = { \
#0051         _lpsz##class_name, sizeof(class_name), wSchema, pfnNew, \
#0052         RUNTIME_CLASS(base_class_name), NULL }; \
#0053     static AFX_CLASSINIT _init_##class_name(&class_name::class##class_name); \
#0054     CRuntimeClass* class_name::GetRuntimeClass() const \
#0055     { return &class_name::class##class_name; } \
#0056
#0057 #define IMPLEMENT_DYNAMIC(class_name, base_class_name) \
#0058     _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, NULL)
#0059
#0060 #define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
#0061     COBJECT* PASCAL class_name::CreateObject() \
#0062     { return new class_name; } \
#0063     _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
#0064         class_name::CreateObject)
#0065
#0066 class COBJECT
#0067 {
#0068 public:
#0069     COBJECT::COBJECT() {
#0070     }
#0071     COBJECT::~~COBJECT() {
#0072     }
#0073
#0074     virtual CRuntimeClass* GetRuntimeClass() const;
#0075     BOOL IsKindOf(const CRuntimeClass* pClass) const;
#0076
#0077 public:
#0078     static CRuntimeClass classCOBJECT;
#0079     virtual void SayHello() { cout << "Hello COBJECT \n"; }
```

```
#0080 };
#0081
#0082 class CCmdTarget : public CObject
#0083 {
#0084     DECLARE_DYNAMIC(CCmdTarget)
#0085 public:
#0086     CCmdTarget::CCmdTarget() {
#0087     }
#0088     CCmdTarget::~CCmdTarget() {
#0089     }
#0090 };
#0091
#0092 class CWinThread : public CCmdTarget
#0093 {
#0094     DECLARE_DYNAMIC(CWinThread)
#0095 public:
#0096     CWinThread::CWinThread() {
#0097     }
#0098     CWinThread::~CWinThread() {
#0099     }
#0100
#0101     virtual BOOL InitInstance() {
#0102         return TRUE;
#0103     }
#0104     virtual int Run() {
#0105         return 1;
#0106     }
#0107 };
#0108
#0109 class CWnd;
#0110
#0111 class CWinApp : public CWinThread
#0112 {
#0113     DECLARE_DYNAMIC(CWinApp)
#0114 public:
#0115     CWinApp* m_pCurrentWinApp;
#0116     CWnd* m_pMainWnd;
#0117
#0118 public:
#0119     CWinApp::CWinApp() {
#0120         m_pCurrentWinApp = this;
#0121     }
#0122     CWinApp::~CWinApp() {
#0123     }
#0124
#0125     virtual BOOL InitApplication() {
```

```

#0126                                     return TRUE;
#0127                                     }
#0128     virtual BOOL InitInstance()      {
#0129                                     return TRUE;
#0130                                     }
#0131     virtual int Run() {
#0132                                     return CWinThread::Run();
#0133                                     }
#0134 };
#0135
#0136
#0137 class CDocument : public CCmdTarget
#0138 {
#0139     DECLARE_DYNAMIC(CDocument)
#0140 public:
#0141     CDocument::CDocument() {
#0142     }
#0143     CDocument::~~CDocument() {
#0144     }
#0145 };
#0146
#0147 class CWnd : public CCmdTarget
#0148 {
#0149     DECLARE_DYNCREATE(CWnd)
#0150 public:
#0151     CWnd::CWnd() {
#0152         cout << "CWnd Constructor \n";
#0153     }
#0154     CWnd::~~CWnd() {
#0155     }
#0156
#0157     virtual BOOL Create();
#0158     BOOL CreateEx();
#0159     virtual BOOL PreCreateWindow();
#0160     void SayHello() { cout << "Hello CWnd \n"; }
#0161 };
#0162
#0163 class CFrameWnd : public CWnd
#0164 {
#0165     DECLARE_DYNCREATE(CFrameWnd)
#0166 public:
#0167     CFrameWnd::CFrameWnd() {
#0168         cout << "CFrameWnd Constructor \n";
#0169     }
#0170     CFrameWnd::~~CFrameWnd() {
#0171     }

```

```

#0172     BOOL Create();
#0173     virtual BOOL PreCreateWindow();
#0174     void SayHello() { cout << "Hello CFrameWnd \n"; }
#0175 };
#0176
#0177 class CView : public CWnd
#0178 {
#0179     DECLARE_DYNAMIC(CView)
#0180 public:
#0181     CView::CView() {
#0182     }
#0183     CView::~CView() {
#0184     }
#0185 };
#0186
#0187 // global function
#0188 CWinApp* AfxGetApp();

```

MFC.CPP

```

#0001 #include "my.h" // it should be mfc.h, but for CMYWinApp definition, so...
#0002
#0003 extern CMYWinApp theApp;
#0004
#0005 static char szCObject[] = "CObject";
#0006 struct CRuntimeClass CObject::classCObject =
#0007     { szCObject, sizeof(CObject), 0xffff, NULL, NULL };
#0008 static AFX_CLASSINIT _init_CObject(&CObject::classCObject);
#0009
#0010 CRuntimeClass* CRuntimeClass::pFirstClass = NULL;
#0011
#0012 AFX_CLASSINIT::AFX_CLASSINIT(CRuntimeClass* pNewClass)
#0013 {
#0014     pNewClass->m_pNextClass = CRuntimeClass::pFirstClass;
#0015     CRuntimeClass::pFirstClass = pNewClass;
#0016 }
#0017
#0018 CObject* CRuntimeClass::CreateObject()
#0019 {
#0020     if (m_pfnCreateObject == NULL)
#0021     {
#0022         TRACE1("Error: Trying to create object which is not "
#0023             "DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n",
#0024             m_lpszClassName);
#0025         return NULL;
#0026     }

```

```

#0027
#0028     CObject* pObject = NULL;
#0029     pObject = (*m_pfnCreateObject)();
#0030
#0031     return pObject;
#0032 }
#0033
#0034 CRuntimeClass* PASCAL CRuntimeClass::Load()
#0035 {
#0036     char szClassName[64];
#0037     CRuntimeClass* pClass;
#0038
#0039     // JJHOU : instead of Load from file, we Load from cin.
#0040     cout << "enter a class name... ";
#0041     cin >> szClassName;
#0042
#0043     for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
#0044     {
#0045         if (strcmp(szClassName, pClass->m_lpszClassName) == 0)
#0046             return pClass;
#0047     }
#0048
#0049     TRACE1("Error: Class not found: %s \n", szClassName);
#0050     return NULL; // not found
#0051 }
#0052
#0053 CRuntimeClass* CObject::GetRuntimeClass() const
#0054 {
#0055     return &CObject::classCObject;
#0056 }
#0057
#0058 BOOL CObject::IsKindOf(const CRuntimeClass* pClass) const
#0059 {
#0060     CRuntimeClass* pClassThis = GetRuntimeClass();
#0061     while (pClassThis != NULL)
#0062     {
#0063         if (pClassThis == pClass)
#0064             return TRUE;
#0065         pClassThis = pClassThis->m_pBaseClass;
#0066     }
#0067     return FALSE; // walked to the top, no match
#0068 }
#0069
#0070 BOOL CWnd::Create()
#0071 {
#0072     return TRUE;

```

```

#0073 }
#0074
#0075 BOOL CWnd::CreateEx()
#0076 {
#0077     PreCreateWindow();
#0078     return TRUE;
#0079 }
#0080
#0081 BOOL CWnd::PreCreateWindow()
#0082 {
#0083     return TRUE;
#0084 }
#0085
#0086 BOOL CFrameWnd::Create()
#0087 {
#0088     CreateEx();
#0089     return TRUE;
#0090 }
#0091
#0092 BOOL CFrameWnd::PreCreateWindow()
#0093 {
#0094     return TRUE;
#0095 }
#0096
#0097 CWinApp* AfxGetApp()
#0098 {
#0099     return theApp.m_pCurrentWinApp;
#0100 }
#0101
#0102 IMPLEMENT_DYNAMIC(CCmdTarget, CObject)
#0103 IMPLEMENT_DYNAMIC(CWinThread, CCmdTarget)
#0104 IMPLEMENT_DYNAMIC(CWinApp, CWinThread)
#0105 IMPLEMENT_DYNAMIC(CDocument, CCmdTarget)
#0106 IMPLEMENT_DYNCREATE(CWnd, CCmdTarget)
#0107 IMPLEMENT_DYNAMIC(CView, CWnd)
#0108 IMPLEMENT_DYNCREATE(CFrameWnd, CWnd)

```

MY.H

```

#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp::CMyWinApp() {

```

```
#0008     }
#0009     CMyWinApp::~CMyWinApp() {
#0010     }
#0011
#0012     virtual BOOL InitInstance();
#0013 };
#0014
#0015 class CMyFrameWnd : public CFrameWnd
#0016 {
#0017     DECLARE_DYNCREATE(CMyFrameWnd)
#0018 public:
#0019     CMyFrameWnd();
#0020     ~CMyFrameWnd() {
#0021     }
#0022     void SayHello() { cout << "Hello CMyFrameWnd \n"; }
#0023 };
#0024
#0025 class CMyDoc : public CDocument
#0026 {
#0027     DECLARE_DYNCREATE(CMyDoc)
#0028 public:
#0029     CMyDoc::CMyDoc() {
#0030         cout << "CMyDoc Constructor \n";
#0031     }
#0032     CMyDoc::~CMyDoc() {
#0033     }
#0034     void SayHello() { cout << "Hello CMyDoc \n"; }
#0035 };
#0036
#0037 class CMyView : public CView
#0038 {
#0039     DECLARE_DYNCREATE(CMyView)
#0040 public:
#0041     CMyView::CMyView() {
#0042         cout << "CMyView Constructor \n";
#0043     }
#0044     CMyView::~CMyView() {
#0045     }
#0046     void SayHello() { cout << "Hello CMyView \n"; }
#0047 };
#0048
#0049 // global function
#0050 void AfxPrintAllClasses();
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     m_pMainWnd = new CMyFrameWnd;
#0008     return TRUE;
#0009 }
#0010
#0011 CMyFrameWnd::CMyFrameWnd()
#0012 {
#0013     cout << "CMyFrameWnd Constructor \n";
#0014     Create();
#0015 }
#0016
#0017 IMPLEMENT_DYNCREATE(CMyFrameWnd, CFrameWnd)
#0018 IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
#0019 IMPLEMENT_DYNCREATE(CMyView, CView)
#0020
#0021 void PrintAllClasses()
#0022 {
#0023     CRuntimeClass* pClass;
#0024
#0025     // just walk through the simple list of registered classes
#0026     for (pClass = CRuntimeClass::pFirstClass; pClass != NULL;
#0027         pClass = pClass->m_pNextClass)
#0028     {
#0029         cout << pClass->m_lpszClassName << "\n";
#0030         cout << pClass->m_nObjectSize << "\n";
#0031         cout << pClass->m_wSchema << "\n";
#0032     }
#0033 }
#0034 //-----
#0035 // main
#0036 //-----
#0037 void main()
#0038 {
#0039     CWinApp* pApp = AfxGetApp();
#0040
#0041     pApp->InitApplication();
#0042     pApp->InitInstance();
#0043     pApp->Run();
#0044 }
```



```
#0045 //Test Dynamic Creation
#0046 CRuntimeClass* pClassRef;
#0047 CObject* pObj;
#0048 while(1)
#0049 {
#0050     if ((pClassRef = CRuntimeClass::Load()) == NULL)
#0051         break;
#0052
#0053     pObj = pClassRef->CreateObject();
#0054     if (pObj != NULL)
#0055         pObj->SayHello();
#0056 }
#0057 }
```

Persistence（永續生存）機制

物件導向有一個術語：Persistence，意思就是把物件永久保留下來。Power 一關，啥都沒有，物件又如何能夠永續存留？

當然是寫到檔案去囉。

把資料寫到檔案，很簡單。在 Document/View 架構中，資料都放在一份 document（文件）裡頭，我們只要把其中的成員變數依續寫進檔案即可。成員變數很可能是個物件，而面對物件，我們首先應該記載其類別名稱，然後才是物件中的資料。

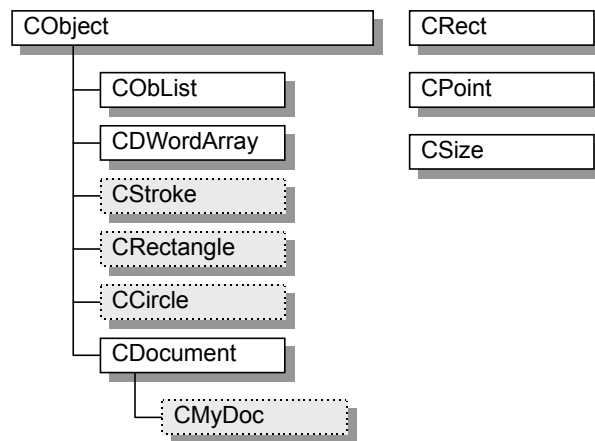
讀檔就有點麻煩了。當程式從檔案中讀到一個類別名稱，它如何實現（instantiate）一個物件？呵，這不就是動態生成的技術嗎？我們在前一章已經解決掉了。

MFC 有一套 Serialize 機制，目的在於把檔名的選擇、檔案的開關、緩衝區的建立、資料的讀寫、萃取運算子（>>）和嵌入運算子（<<）的多載（overload）、物件的動態生成...都包裝起來。

上述 Serialize 的各部份工作，除了資料的讀寫和物件的動態生成，其餘都是支節。動態生成的技術已經解決，讓我們集中火力，分析資料的讀寫動作。

Serialize（資料讀寫）

假設我有一份文件，用以記錄一張圖形。圖形只有三種基本元素：線條（Stroke）、圓形、矩形。我打算用以下類別，組織這份文件：



其中 *CObList* 和 *CDWordArray* 是 MFC 提供的類別，前者是一個串列，可放置任何從 *CObject* 衍生下來的物件，後者是一個陣列，每一個元素都是 "double word"。另外三個類別：*CStroke* 和 *CRectangle* 和 *CCircle*，是我從 *CObject* 中衍生下來的類別。

```

class CMyDoc : public CDocument
{
    CObList m_graphList;
    CSize m_sizeDoc;
    ...
};
class CStroke : public CObject
{
    CDWordArray m_ptArray; // series of connected points
    ...
};
class CRectangle : public CObject
{
    CRect m_rect;
    ...
};
class CCircle : public CObject
  
```

```
{
    CPoint  m_center;
    UINT    m_radius;
    ...
};
```

假設現有一份文件，內容如圖 3-3，如果你是 Serialize 機制的設計者，你希望怎麼做呢？

把圖 3-3 寫成這樣的檔案內容好嗎：

06 00	;COBList elements count
07 00	;class name string length
43 53 74 72 6F 6B 65	;"CStroke"
02 00	;DWordArray size
28 00 13 00	;point
28 00 13 00	;point
0A 00	;class name string length
43 52 65 63 74 61 6E 67 6C 65	;"CRectangle"
11 00 22 00 33 00 44 00	;CRect
07 00	;class name string length
43 43 69 72 63 6C 65	;"CCircle"
55 00 66 00 77 00	;CPoint & radius
07 00	;class name string length
43 53 74 72 6F 6B 65	;"CStroke"
02 00	;DWordArray size
28 00 35 00	;point
28 00 35 00	;point
0A 00	;class name string length
43 52 65 63 74 61 6E 67 6C 65	;"CRectangle"
11 00 22 00 33 00 44 00	;CRect
07 00	;class name string length
43 43 69 72 63 6C 65	;"CCircle"
55 00 66 00 77 00	;CPoint & radius

還算堪用。但如果考慮到螢幕捲動的問題，以及印表輸出的問題，應該在最前端增加「文件大小」。另外，如果這份文件有 100 條線條，50 個圓形，80 個矩形，難不成我們要記錄 230 個類別名稱？應該有更好的方法才是。

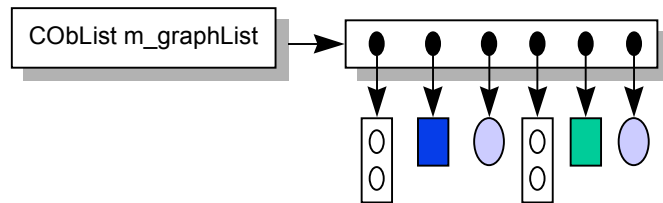


圖 3-3 一個串列，內含三種基本圖形：線條、圓形、矩形。

我們可以在每次記錄物件內容的時候，先寫入一個代碼，表示此物件之類別是否曾在檔案中記錄過了。如果是新類別，乖乖地記錄其類別名稱；如果是舊類別，則以代碼表示。這樣可以節省檔案大小以及程式用於解析的時間。啊，不要看到檔案大小就想到硬碟很便宜，桌上的一切都將被帶到網上，你得想想網路頻寬這回事。

還有一個問題。文件的「版本」如何控制？舊版程式讀取新版文件，新版程式讀取舊版文件，都可能出狀況。爲了防弊，最好把版本號碼記錄上去。最好是每個類別有自己的版本號碼。

下面是新的構想，也就是 **Serialization** 的目標：

```

20 03 84 03                ;Document Size
06 00                      ;CObList elements count

FF FF                    ;new class tag
02 00                      ;schema
07 00                      ;class name string length
43 53 74 72 6F 6B 65       ;"CStroke"
02 00                      ;DWordArray size
28 00 13 00                ;point
28 00 13 00                ;point

FF FF                    ;new class tag
01 00                      ;schema
0A 00                      ;class name string length
43 52 65 63 74 61 6E 67 6C 65 ;"CRectangle"
11 00 22 00 33 00 44 00    ;CRect

FF FF                    ;new class tag
01 00                      ;schema
07 00                      ;class name string length
43 43 69 72 63 6C 65       ;"CCircle"
55 00 66 00 77 00          ;CPoint & radius

01 80                    ;old class tag
02 00                      ;DWordArray size
28 00 35 00                ;point
28 00 35 00                ;point

03 80                    ;old class tag
11 00 22 00 33 00 44 00    ;CRect

05 80                    ;old class tag
55 00 66 00 77 00          ;CPoint & radius

```

我希望有一個專門負責 Serialization 的函式，就叫作 *Serialize* 好了。假設現在我的 Document 類別名稱爲 *CScribDoc*，我希望有這麼便利的程式方法（請仔細琢磨琢磨其便利性）：

```

void CScribDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_sizeDoc;
    else
        ar >> m_sizeDoc;
    m_graphList.Serialize(ar);
}

void CObList::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {

```

```
        ar << (WORD) m_nCount;
        for (CNode* pNode = m_pNodeHead; pNode != NULL; pNode = pNode->pNext)
            ar << pNode->data;
    }
    else {
        WORD nNewCount;
        ar >> nNewCount;
        while (nNewCount-- > 0) {
            CObject* newData;
            ar >> newData;
            AddTail(newData);
        }
    }
}

void CStroke::Serialize(CArchive& ar)
{
    m_ptArray.Serialize(ar);
}

void CDWordArray::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
        ar << (WORD) m_nSize;
        for (int i = 0; i < m_nSize; i++)
            ar << m_pData[i];
    }
    else {
        WORD nOldSize;
        ar >> nOldSize;
        for (int i = 0; i < m_nSize; i++)
            ar >> m_pData[i];
    }
}

void CRectangle::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_rect;
    else
        ar >> m_rect;
}

void CCircle::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
```

```

        ar << (WORD)m_center.x;
        ar << (WORD)m_center.y;
        ar << (WORD)m_radius;
    }
    else {
        ar >> (WORD&)m_center.x;
        ar >> (WORD&)m_center.y;
        ar >> (WORD&)m_radius;
    }
}

```

每一個可寫到檔案或可從檔案中讀出的類別，都應該有它自己的 *Serialize* 函式，負責它自己的資料讀寫檔動作。此類別並且應該改寫 << 運算子和 >> 運算子，把資料導流到 archive 中。archive 是什麼？是一個與檔案息息相關的緩衝區，暫時你可以想像它就是檔案的化身。當圖 3-3 的文件寫入檔案時，*Serialize* 函式的呼叫次序如圖 3-4。

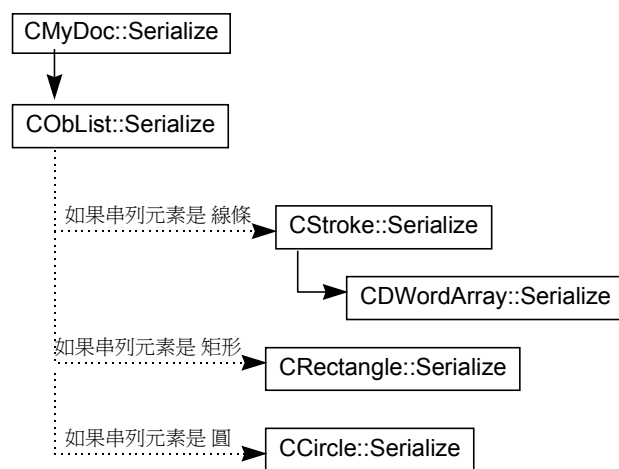


圖 3-4 圖 3-3 的文件內容寫入檔案時，*Serialize* 函式的呼叫次序。

DECLARE_SERIAL / IMPLEMENT_SERIAL 巨集

要將 << 和 >> 兩個運算子多載化，還要讓 *Serialize* 函式神不知鬼不覺地放入類別宣告之中，最好的作法仍然是使用巨集。

類別之能夠進行檔案讀寫動作，前提是擁有動態生成的能力，所以，MFC 設計了兩個巨集 *DECLARE_SERIAL* 和 *IMPLEMENT_SERIAL*：

```
#define DECLARE_SERIAL(class_name) \
    DECLARE_DYNCREATE(class_name) \
    friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);

#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* PASCAL class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
        class_name::CreateObject) \
    CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name)); \
        return ar; } \
```

爲了在每一個物件被處理（讀或寫）之前，能夠處理瑣屑的工作，諸如判斷是否第一次出現、記錄版本號碼、記錄檔名等工作，*CRuntimeClass* 需要兩個函式 *Load* 和 *Store*：

```
struct CRuntimeClass
{
// Attributes
    LPCSTR m_lpszClassName;
    int m_nObjectSize;
    UINT m_wSchema; // schema number of the loaded class
    CObject* (PASCAL* m_pfnCreateObject)(); // NULL => abstract class
    CRuntimeClass* m_pBaseClass;

    CObject* CreateObject();
    void Store(CArchive& ar) const;
    static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);

    // CRuntimeClass objects linked together in simple list
    static CRuntimeClass* pFirstClass; // start of class list
    CRuntimeClass* m_pNextClass; // linked list of registered classes
};
```


你已經在上一節看過 *Load* 函式，當時爲了簡化，我把它的參數拿掉，改爲由螢幕上獲得類別名稱，事實上它應該是從檔案中讀一個類別名稱。至於 *Store* 函式，是把類別名稱寫入檔案中：

```
// Runtime class serialization code
CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar, UINT* pwSchemaNum)
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    ar >> (WORD&)(*pwSchemaNum) >> nLen;

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

    for (pClass = pFirstClass; pClass != NULL; pClass = pClass->m_pNextClass)
    {
        if (lstrcmp(szClassName, pClass->m_lpszClassName) == 0)
            return pClass;
    }
    return NULL; // not found
}

void CRuntimeClass::Store(CArchive& ar) const
    // stores a runtime class description
{
    WORD nLen = (WORD)lstrlenA(m_lpszClassName);
    ar << (WORD)m_wSchema << nLen;
    ar.Write(m_lpszClassName, nLen*sizeof(char));
}
```

圖 3-4 的例子中，爲了讓整個 *Serialization* 機制運作起來，我們必須做這樣的類別宣告：

```
class CScribDoc : public CDocument
{
    DECLARE_DYNCREATE(CScribDoc)
    ...
};
class CStroke : public CObject
{
    DECLARE_SERIAL(CStroke)
public:
```

```

        void Serialize(CArchive&);
...
};
class CRectangle : public CObject
{
    DECLARE_SERIAL(CRectangle)
public:
    void Serialize(CArchive&);
...
};
class CCircle : public CObject
{
    DECLARE_SERIAL(CCircle)
public:
    void Serialize(CArchive&);
...
};

```

以及在 .CPP 檔中做這樣的動作：

```

IMPLEMENT_DYNCREATE(CScribDoc, CDocument)
IMPLEMENT_SERIAL(CStroke, CObject, 2)
IMPLEMENT_SERIAL(CRectangle, CObject, 1)
IMPLEMENT_SERIAL(CCircle, CObject, 1)

```

然後呢？分頭設計 *CStroke*、*CRectangle* 和 *CCircle* 的 *Serialize* 函式吧。

當然，毫不令人意外地，MFC 原始碼中的 *CObList* 和 *CDWordArray* 有這樣的內容：

```

// in header files
class CDWordArray : public CObject
{
    DECLARE_SERIAL(CDWordArray)
public:
    void Serialize(CArchive&);
...
};
class CObList : public CObject
{
    DECLARE_SERIAL(CObList)
public:
    void Serialize(CArchive&);
...
};

// in implementation files

```

```
IMPLEMENT_SERIAL(CObList, CObject, 0)
IMPLEMENT_SERIAL(CDWordArray, CObject, 0)
```

而 *CObject* 也多了一個虛擬函式 *Serialize*：

```
class CObject
{
public:
    virtual void Serialize(CArchive& ar);
    ...
}
```

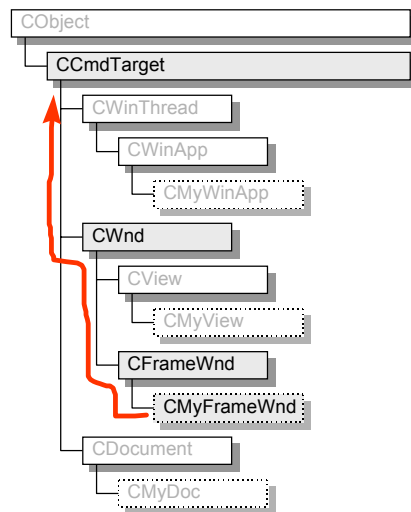
沒有範例程式

抱歉，我沒有準備 DOS 版的 *Serialization* 範例程式給你。你看到了，很多東西需要模擬：*CFile*、*CArchive*、*CObList*、*CDWordArray*、*CRect*、*CPoint*、運算子多載、*Serialize* 函式…。我乾脆在本書第 8 章直接為你解釋 MFC 的作法，更好。

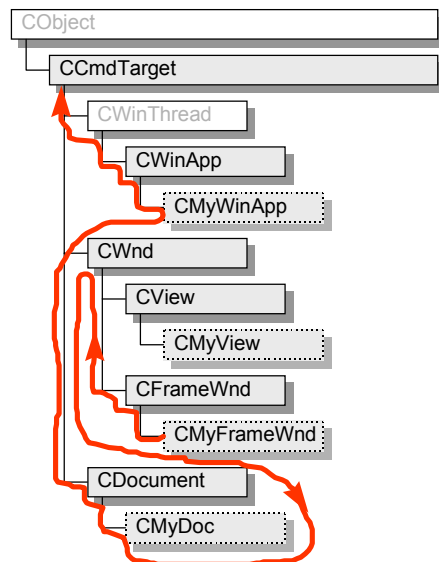
Message Mapping (訊息映射)

Windows 程式靠訊息的流動而維護生命。你已經在第一章看過了訊息的一般處理方式，也就是在視窗函式中藉著一個大大的 *switch/case* 比對動作，判別訊息再呼叫對應的處理常式。爲了讓大大的 *switch/case* 比對動作簡化，也讓程式碼更模組化一些，我在第 1 章提供了一個簡易的訊息映射表作法，把訊息和其處理常式關聯起來。

當我們的類別庫成立，如果其中與訊息有關的類別（姑且叫作「訊息標的類別」好了，在 MFC 之中就是 *CCmdTarget*）都是一條鞭式地繼承，我們應該爲每一個「訊息標的類別」準備一個訊息映射表，並且將基礎類別與衍生類別之訊息映射表串接起來。然後，當視窗函式做訊息的比對時，我們就可以想辦法導引它沿著這條路走過去：



但是，MFC 之中用來處理訊息的 C++ 類別，並不呈單鞭發展。作為 application framework 的重要架構之一的 document/view，也具有處理訊息的能力（你現在可能還不清楚什麼是 document/view，沒有關係）。因此，訊息藉以攀爬的路線應該有橫流的機會：



訊息如何流動，我們暫時先不管。是直線前進，或是中途換跑道，我們都暫時不管，本節先把這個攀爬路線網建立起來再說。這整個攀爬路線網就是所謂的訊息映射表（Message Map）；說它是一張地圖，當然也沒有錯。將訊息與表格中的元素比對，然後呼叫對應的處理常式，這種動作我們也稱之為訊息映射（Message Mapping）。

為了儘量降低對正常（一般）類別宣告和定義的影響，我們希望，最好能夠像 RTTI 和 Dynamic Creation 一樣，用一兩個巨集就完成這巨大蜘蛛網的建構。最好能夠像 *DECLARE_DYNAMIC* 和 *IMPLEMENT_DYNAMIC* 巨集那麼方便。

首先定義一個資料結構：

```
struct AFX_MSGMAP
{
    AFX_MSGMAP* pBaseMessageMap;
    AFX_MSGMAP_ENTRY* lpEntries;
};
```

其中的 *AFX_MSGMAP_ENTRY* 又是另一個資料結構：

```
struct AFX_MSGMAP_ENTRY // MFC 4.0 format
{
    UINT nMessage;        // windows message
    UINT nCode;           // control code or WM_NOTIFY code
    UINT nID;             // control ID (or 0 for windows messages)
    UINT nLastID;         // used for entries specifying a range of control id's
    UINT nSig;            // signature type (action) or pointer to message #
    AFX_PMSG pfn;        // routine to call (or special value)
};
```

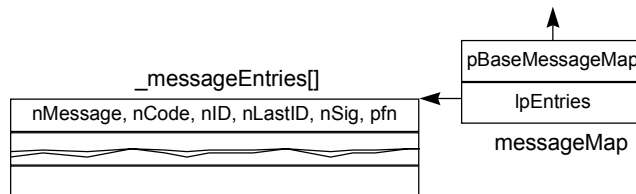
其中的 *AFX_PMSG* 定義為函式指標：

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

然後我們定義一個巨集：

```
#define DECLARE_MESSAGE_MAP() \
    static AFX_MSGMAP_ENTRY _messageEntries[]; \
    static AFX_MSGMAP messageMap; \
    virtual AFX_MSGMAP* GetMessageMap() const;
```

於是，`DECLARE_MESSAGE_MAP` 就相當於宣告了這樣一個資料結構：



這個資料結構的內容填塞工作由三個巨集完成：

```

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

#define ON_COMMAND(id, memberFxn) \
    { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },

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

```

其中的 `AfxSig_end` 定義為：

```

enum AfxSig
{
    AfxSig_end = 0,    // [marks end of message map]
    AfxSig_vv,
};

```

`AfxSig_xx` 用來描述訊息處理常式 `memberFxn` 的型態（參數與回返值）。本例純為模擬與簡化，所以不在這上面作文章。真正講到 MFC 時（第四篇 p580），我會再解釋它。

於是，以 *CView* 為例，下面的原始碼：

```
// in header file
class CView : public CWnd
{
public:
    ...
    DECLARE_MESSAGE_MAP()
};

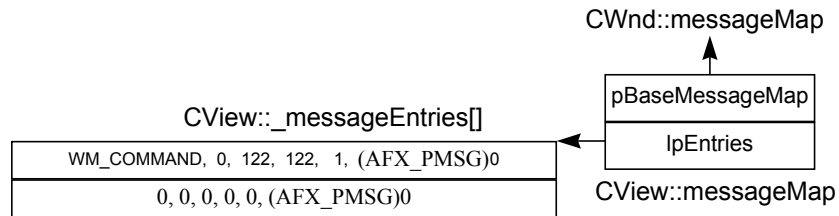
// in implementation file
#define CViewid 122
...
BEGIN_MESSAGE_MAP(CView, CWnd)
    ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()
```

就被展開成爲：

```
// in header file
class CView : public CWnd
{
public:
    ...
    static AFX_MSGMAP_ENTRY _messageEntries[];
    static AFX_MSGMAP messageMap;
    virtual AFX_MSGMAP* GetMessageMap() const;
};

// in implementation file
AFX_MSGMAP* CView::GetMessageMap() const
{ return &CView::messageMap; }
AFX_MSGMAP CView::messageMap =
{ &(CWnd::messageMap),
  (AFX_MSGMAP_ENTRY*) &(CView::_messageEntries) };
AFX_MSGMAP_ENTRY CView::_messageEntries[] =
{
    { WM_COMMAND, 0, (WORD)122, (WORD)122, 1, (AFX_PMSG)0 },
    { 0, 0, 0, 0, 0, (AFX_PMSG)0 }
};
```

以圖表示則爲：



我們還可以定義各種類似 `ON_COMMAND` 這樣的巨集，把各式各樣的訊息與特定的處理常式關聯起來。MFC 裡頭就有名為 `ON_WM_PAINT`、`ON_WM_CREATE`、`ON_WM_SIZE`... 等等的巨集。

我在 Frame7 範例程式中為 `CCmdTarget` 的每一衍生類別都產生類似上圖的訊息映射表：

```
// in header files
class CObject
{
... // 注意：CObject 並不屬於訊息流動網的一份子。
};
class CCmdTarget : public CObject
{
...
DECLARE_MESSAGE_MAP()
};
class CWinThread : public CCmdTarget
{
... // 注意：CWinThread 並不屬於訊息流動網的一份子。
};
class CWinApp : public CWinThread
{
...
DECLARE_MESSAGE_MAP()
};
class CDocument : public CCmdTarget
{
...
DECLARE_MESSAGE_MAP()
};
class CWnd : public CCmdTarget
{
...
DECLARE_MESSAGE_MAP()
```



```
};  
class CFrameWnd : public CWnd  
{  
...  
    DECLARE_MESSAGE_MAP()  
};  
class CView : public CWnd  
{  
...  
    DECLARE_MESSAGE_MAP()  
};  
class CMyWinApp : public CWinApp  
{  
...  
    DECLARE_MESSAGE_MAP()  
};  
class CMyFrameWnd : public CFrameWnd  
{  
...  
    DECLARE_MESSAGE_MAP()  
};  
class CMyDoc : public CDocument  
{  
...  
    DECLARE_MESSAGE_MAP()  
};  
class CMyView : public CView  
{  
...  
    DECLARE_MESSAGE_MAP()  
};
```

並且把各訊息映射表的關聯性架設起來，給予初值（每一個映射表都只有 *ON_COMMAND* 一個項目）：

```
// in implementation files  
BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)  
    ON_COMMAND(CWndid, 0)  
END_MESSAGE_MAP()  
  
BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)  
    ON_COMMAND(CFrameWndid, 0)  
END_MESSAGE_MAP()  
  
BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
```

```

ON_COMMAND(CDocumentid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CView, CWnd)
ON_COMMAND(CViewid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
ON_COMMAND(CWinAppid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
ON_COMMAND(CMyWinAppid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
ON_COMMAND(CMyFrameWndid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
ON_COMMAND(CMyDocid, 0)
END_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(CMyView, CView)
ON_COMMAND(CMyViewid, 0)
END_MESSAGE_MAP()

```

同時也設定了訊息的終極鏢靶 *CCmdTarget* 的映射表內容：

```

AFX_MSGMAP CCmdTarget::messageMap =
{
    NULL,
    &CCmdTarget::_messageEntries[0]
};

AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
{
    { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
};

```

於是，整個訊息流動網就隱然成形了（圖 3-5）。

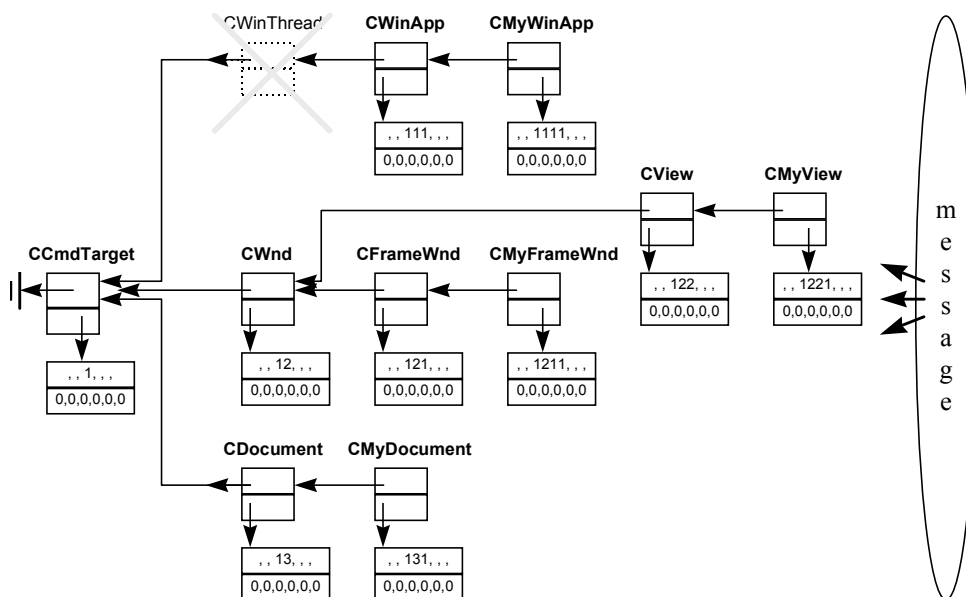


圖 3-5 Frame7 程式所架構起來的訊息流動網（也就是 Message Map）。

為了驗證整個訊息映射表，我必須在映射表中做點記號，等全部建構完成之後，再一一追蹤把記號顯示出來。我將為每一個類別的訊息映射表加上這個項目：

```
ON_COMMAND(Classid, 0)
```

這樣就可以把 Classid 嵌到映射表中當作記號。正式用途（於 MFC 中）當然不是這樣，這只不過是權宜之計。

在 main 函式中，我先產生四個物件（分別是 CMyWinApp、CMyFrameWnd、CMyDoc、CMyView 物件）：

```
CMyWinApp theApp; // theApp 是 CMyWinApp 物件
void main()
{
    CWinApp* pApp = AfxGetApp();
    pApp->InitApplication();
    pApp->InitInstance(); // 產生 CMyFrameWnd 物件
    pApp->Run();
}
```

```

CMyDoc* pMyDoc = new CMyDoc;    // 產生 CMyDoc 物件
CMyView* pMyView = new CMyView; // 產生 CMyView 物件
CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
...
}

```

然後分別取其訊息映射表，一路追蹤上去，把每一個訊息映射表中的類別記號列印出來：

```

void main()
{
...
    AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
    cout << endl << "CMyView Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pMyDoc->GetMessageMap();
    cout << endl << "CMyDoc Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pMyFrame->GetMessageMap();
    cout << endl << "CMyFrameWnd Message Map : " << endl;
    MsgMapPrinting(pMessageMap);

    pMessageMap = pApp->GetMessageMap();
    cout << endl << "CMyWinApp Message Map : " << endl;
    MsgMapPrinting(pMessageMap);
}

```

下面這個函式追蹤並列印訊息映射表中的 classid 記號：

```

void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
{
    for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap)
    {
        AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
        printlpEntries(lpEntry);
    }
}

void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
{
    struct {
        int classid;
        char* classname;
    } classinfo[] = {
        CCmdTargetid, "CCmdTarget",
        CWinThreadid, "CWinThread",

```

```
        CWinAppid,      "CWinApp",
        CMyWinAppid,    "CMyWinApp",
        CWndid,         "CWnd",
        CFrameWndid,    "CFrameWnd",
        CMyFrameWndid,  "CMyFrameWnd",
        CViewid,        "CView",
        CMyViewid,       "CMyView",
        CDocumentid,    "CDocument",
        CMyDocid,       "CMyDoc",
        0,              "      "
    };

    for (int i=0; classinfo[i].classid != 0; i++)
    {
        if (classinfo[i].classid == lpEntry->nID)
        {
            cout << lpEntry->nID << "      ";
            cout << classinfo[i].classname << endl;
            break;
        }
    }
}
```

Frame7 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第 4 章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

Frame7 的執行結果是：

```
CMyView Message Map :
1221  CMyView
122   CView
12    CWnd
1     CCmdTarget

CMyDoc Message Map :
131   CMyDoc
13    CDocument
1     CCmdTarget

CMyFrameWnd Message Map :
1211  CMyFrameWnd
121   CFrameWnd
12    CWnd
```

```

1    CCmdTarget

CMyWinApp Message Map :
1111    CMyWinApp
111    CWinApp
1    CCmdTarget

```

Frame7 範例程式

MFC.H

```

#0001 #define TRUE 1
#0002 #define FALSE 0
#0003
#0004 typedef char* LPSTR;
#0005 typedef const char* LPCSTR;
#0006
#0007 typedef unsigned long DWORD;
#0008 typedef int BOOL;
#0009 typedef unsigned char BYTE;
#0010 typedef unsigned short WORD;
#0011 typedef int INT;
#0012 typedef unsigned int UINT;
#0013 typedef long LONG;
#0014
#0015 #define WM_COMMAND 0x0111
#0016 #define CObjectid 0xffff
#0017 #define CCmdTargetid 1
#0018 #define CWinThreadid 11
#0019 #define CWinAppid 111
#0020 #define CMyWinAppid 1111
#0021 #define CWndid 12
#0022 #define CFrameWndid 121
#0023 #define CMyFrameWndid 1211
#0024 #define CViewid 122
#0025 #define CMyViewid 1221
#0026 #define CDocumentid 13
#0027 #define CMyDocid 131
#0028
#0029 #include <iostream.h>
#0030
#0031 ///////////////////////////////////////////////////
#0032 // Window message map handling
#0033

```

```
#0034 struct AFX_MSGMAP_ENTRY;          // declared below after CWnd
#0035
#0036 struct AFX_MSGMAP
#0037 {
#0038     AFX_MSGMAP* pBaseMessageMap;
#0039     AFX_MSGMAP_ENTRY* lpEntries;
#0040 };
#0041
#0042 #define DECLARE_MESSAGE_MAP() \
#0043     static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0044     static AFX_MSGMAP messageMap; \
#0045     virtual AFX_MSGMAP* GetMessageMap() const;
#0046
#0047 #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0048     AFX_MSGMAP* theClass::GetMessageMap() const \
#0049     { return &theClass::messageMap; } \
#0050     AFX_MSGMAP theClass::messageMap = \
#0051     { &(baseClass::messageMap), \
#0052       (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0053     AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0054     {
#0055
#0056 #define END_MESSAGE_MAP() \
#0057     { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0058     };
#0059
#0060 // Message map signature values and macros in separate header
#0061 #include "afxmsg_.h"
#0062
#0063 class CObject
#0064 {
#0065 public:
#0066     CObject::CObject() {
#0067     }
#0068     CObject::~~CObject() {
#0069     }
#0070 };
#0071
#0072 class CCmdTarget : public CObject
#0073 {
#0074 public:
#0075     CCmdTarget::CCmdTarget() {
#0076     }
#0077     CCmdTarget::~~CCmdTarget() {
#0078     }
#0079     DECLARE_MESSAGE_MAP()          // base class - no {{ }} macros
```

```
#0080 };
#0081
#0082 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0083
#0084 struct AFX_MSGMAP_ENTRY // MFC 4.0
#0085 {
#0086     UINT nMessage; // windows message
#0087     UINT nCode;     // control code or WM_NOTIFY code
#0088     UINT nID;       // control ID (or 0 for windows messages)
#0089     UINT nLastID;   // used for entries specifying a range of control id's
#0090     UINT nSig;      // signature type (action) or pointer to message #
#0091     AFX_PMSG pfn;   // routine to call (or special value)
#0092 };
#0093
#0094 class CWinThread : public CCmdTarget
#0095 {
#0096 public:
#0097     CWinThread::CWinThread() {
#0098     }
#0099     CWinThread::~CWinThread() {
#0100     }
#0101
#0102     virtual BOOL InitInstance() {
#0103         cout << "CWinThread::InitInstance \n";
#0104         return TRUE;
#0105     }
#0106     virtual int Run() {
#0107         cout << "CWinThread::Run \n";
#0108         return 1;
#0109     }
#0110 };
#0111
#0112 class CWnd;
#0113
#0114 class CWinApp : public CWinThread
#0115 {
#0116 public:
#0117     CWinApp* m_pCurrentWinApp;
#0118     CWnd* m_pMainWnd;
#0119
#0120 public:
#0121     CWinApp::CWinApp() {
#0122         m_pCurrentWinApp = this;
#0123     }
#0124     CWinApp::~CWinApp() {
#0125     }
```



```

#0126
#0127     virtual BOOL InitApplication() {
#0128                                     cout << "CWinApp::InitApplication \n";
#0129                                     return TRUE;
#0130                                     }
#0131     virtual BOOL InitInstance() {
#0132                                     cout << "CWinApp::InitInstance \n";
#0133                                     return TRUE;
#0134                                     }
#0135     virtual int Run() {
#0136                                     cout << "CWinApp::Run \n";
#0137                                     return CWinThread::Run();
#0138                                     }
#0139
#0140     DECLARE_MESSAGE_MAP()
#0141 };
#0142
#0143 typedef void (CWnd::*AFX_PMSGW)(void);
#0144 // like 'AFX_PMSG' but for CWnd derived classes only
#0145
#0146 class CDocument : public CCmdTarget
#0147 {
#0148 public:
#0149     CDocument::CDocument() {
#0150     }
#0151     CDocument::~~CDocument() {
#0152     }
#0153     DECLARE_MESSAGE_MAP()
#0154 };
#0155
#0156 class CWnd : public CCmdTarget
#0157 {
#0158 public:
#0159     CWnd::CWnd() {
#0160     }
#0161     CWnd::~~CWnd() {
#0162     }
#0163
#0164     virtual BOOL Create();
#0165     BOOL CreateEx();
#0166     virtual BOOL PreCreateWindow();
#0167
#0168     DECLARE_MESSAGE_MAP()
#0169 };
#0170
#0171 class CFrameWnd : public CWnd

```

```

#0172 {
#0173 public:
#0174     CFrameWnd::CFrameWnd() {
#0175     }
#0176     CFrameWnd::~~CFrameWnd() {
#0177     }
#0178     BOOL Create();
#0179     virtual BOOL PreCreateWindow();
#0180
#0181     DECLARE_MESSAGE_MAP()
#0182 };
#0183
#0184 class CView : public CWnd
#0185 {
#0186 public:
#0187     CView::CView() {
#0188     }
#0189     CView::~~CView() {
#0190     }
#0191     DECLARE_MESSAGE_MAP()
#0192 };
#0193
#0194 // global function
#0195 CWinApp* AfxGetApp();

```

AFXMSG_.H

```

#0001 enum AfxSig
#0002 {
#0003     AfxSig_end = 0,    // [marks end of message map]
#0004     AfxSig_vv,
#0005 };
#0006
#0007 #define ON_COMMAND(id, memberFxn) \
#0008     { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },

```

MFC.CPP

```

#0001 #include "my.h" // 原該含入 mfc.h 就好，但爲了 CMyWinApp 的定義，所以...
#0002
#0003 extern CMyWinApp theApp;
#0004
#0005 BOOL CWnd::Create()
#0006 {
#0007     cout << "CWnd::Create \n";

```

```
#0008     return TRUE;
#0009 }
#0010
#0011 BOOL CWnd::CreateEx()
#0012 {
#0013     cout << "CWnd::CreateEx \n";
#0014     PreCreateWindow();
#0015     return TRUE;
#0016 }
#0017
#0018 BOOL CWnd::PreCreateWindow()
#0019 {
#0020     cout << "CWnd::PreCreateWindow \n";
#0021     return TRUE;
#0022 }
#0023
#0024 BOOL CFrameWnd::Create()
#0025 {
#0026     cout << "CFrameWnd::Create \n";
#0027     CreateEx();
#0028     return TRUE;
#0029 }
#0030
#0031 BOOL CFrameWnd::PreCreateWindow()
#0032 {
#0033     cout << "CFrameWnd::PreCreateWindow \n";
#0034     return TRUE;
#0035 }
#0036
#0037 CWinApp* AfxGetApp()
#0038 {
#0039     return theApp.m_pCurrentWinApp;
#0040 }
#0041
#0042 AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0043 {
#0044     return &CCmdTarget::messageMap;
#0045 }
#0046
#0047 AFX_MSGMAP CCmdTarget::messageMap =
#0048 {
#0049     NULL,
#0050     &CCmdTarget::_messageEntries[0]
#0051 };
#0052
#0053 AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
```

```

#0054 {
#0055     // { 0, 0, 0, 0, AfxSig_end, 0 }    // nothing here
#0056     { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0057
#0058 };
#0059
#0060 BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0061 ON_COMMAND(CWndid, 0)
#0062 END_MESSAGE_MAP()
#0063
#0064 BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0065 ON_COMMAND(CFrameWndid, 0)
#0066 END_MESSAGE_MAP()
#0067
#0068 BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0069 ON_COMMAND(CDocumentid, 0)
#0070 END_MESSAGE_MAP()
#0071
#0072 BEGIN_MESSAGE_MAP(CView, CWnd)
#0073 ON_COMMAND(CViewid, 0)
#0074 END_MESSAGE_MAP()
#0075
#0076 BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0077 ON_COMMAND(CWinAppid, 0)
#0078 END_MESSAGE_MAP()

```

MY.H

```

#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp::CMyWinApp() {
#0008     }
#0009     CMyWinApp::~CMyWinApp() {
#0010     }
#0011
#0012     virtual BOOL InitInstance();
#0013     DECLARE_MESSAGE_MAP()
#0014 };
#0015
#0016 class CMyFrameWnd : public CFrameWnd
#0017 {
#0018 public:

```

```
#0019     CMyFrameWnd();
#0020     ~CMyFrameWnd() {
#0021         }
#0022     DECLARE_MESSAGE_MAP()
#0023 };
#0024
#0025 class CMyDoc : public CDocument
#0026 {
#0027 public:
#0028     CMyDoc::CMyDoc() {
#0029     }
#0030     CMyDoc::~~CMyDoc() {
#0031     }
#0032     DECLARE_MESSAGE_MAP()
#0033 };
#0034
#0035 class CMyView : public CView
#0036 {
#0037 public:
#0038     CMyView::CMyView() {
#0039     }
#0040     CMyView::~~CMyView() {
#0041     }
#0042     DECLARE_MESSAGE_MAP()
#0043 };
```

MY.CPP

```
#0001 #include "my.h"
#0002
#0003 CMyWinApp theApp;
#0004
#0005 BOOL CMyWinApp::InitInstance()
#0006 {
#0007     cout << "CMyWinApp::InitInstance \n";
#0008     m_pMainWnd = new CMyFrameWnd;
#0009     return TRUE;
#0010 }
#0011
#0012 CMyFrameWnd::CMyFrameWnd()
#0013 {
#0014     Create();
#0015 }
#0016
#0017 BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
#0018 ON_COMMAND(CMyWinAppid, 0)
```

```
#0019 END_MESSAGE_MAP()
#0020
#0021 BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022 ON_COMMAND(CMyFrameWndid, 0)
#0023 END_MESSAGE_MAP()
#0024
#0025 BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026 ON_COMMAND(CMyDocid, 0)
#0027 END_MESSAGE_MAP()
#0028
#0029 BEGIN_MESSAGE_MAP(CMyView, CView)
#0030 ON_COMMAND(CMyViewid, 0)
#0031 END_MESSAGE_MAP()
#0032
#0033 void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034 {
#0035     struct {
#0036         int classid;
#0037         char* classname;
#0038     } classinfo[] = {
#0039         CCmdTargetid, "CCmdTarget",
#0040         CWinThreadid, "CWinThread",
#0041         CWinAppid, "CWinApp",
#0042         CMyWinAppid, "CMyWinApp",
#0043         CWndid, "CWnd",
#0044         CFrameWndid, "CFrameWnd",
#0045         CMyFrameWndid, "CMyFrameWnd",
#0046         CViewid, "CView",
#0047         CMyViewid, "CMyView",
#0048         CDocumentid, "CDocument",
#0049         CMyDocid, "CMyDoc",
#0050         0, ""
#0051     };
#0052
#0053     for (int i=0; classinfo[i].classid != 0; i++)
#0054     {
#0055         if (classinfo[i].classid == lpEntry->nID)
#0056         {
#0057             cout << lpEntry->nID << " ";
#0058             cout << classinfo[i].classname << endl;
#0059             break;
#0060         }
#0061     }
#0062 }
#0063
#0064 void MsgMapPrinting(AFX_MSGMAP* pMessageMap)
```

```
#0065 {
#0066     for(; pMessageMap != NULL; pMessageMap = pMessageMap->pBaseMessageMap) {
#0067         AFX_MSGMAP_ENTRY* lpEntry = pMessageMap->lpEntries;
#0068         printlpEntries(lpEntry);
#0069     }
#0070 }
#0071
#0072 //-----
#0073 // main
#0074 //-----
#0075 void main()
#0076 {
#0077
#0078     CWinApp* pApp = AfxGetApp();
#0079
#0080     pApp->InitApplication();
#0081     pApp->InitInstance();
#0082     pApp->Run();
#0083
#0084     CMyDoc* pMyDoc = new CMyDoc;
#0085     CMyView* pMyView = new CMyView;
#0086     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0087
#0088     // output Message Map construction
#0089     AFX_MSGMAP* pMessageMap = pMyView->GetMessageMap();
#0090     cout << endl << "CMyView Message Map : " << endl;
#0091     MsgMapPrinting(pMessageMap);
#0092
#0093     pMessageMap = pMyDoc->GetMessageMap();
#0094     cout << endl << "CMyDoc Message Map : " << endl;
#0095     MsgMapPrinting(pMessageMap);
#0096
#0097     pMessageMap = pMyFrame->GetMessageMap();
#0098     cout << endl << "CMyFrameWnd Message Map : " << endl;
#0099     MsgMapPrinting(pMessageMap);
#0100
#0101     pMessageMap = pApp->GetMessageMap();
#0102     cout << endl << "CMyWinApp Message Map : " << endl;
#0103     MsgMapPrinting(pMessageMap);
#0104 }
```

Command Routing (命令繞行)

我們已經在上一節把整個訊息流動網架設起來了。當訊息進來，會有一個邦浦推動它前進。訊息如何進來，以及邦浦函式如何推動，都是屬於 Windows 程式設計的範疇，暫時不管。我現在要模擬出訊息的流動繞行路線 -- 我常喜歡稱之為訊息的「二萬五千里長征」。

訊息如果是從子類別流向父類別（縱向流動），那麼事情再簡單不過，整個 Message Map 訊息映射表已規劃出十分明確的路線。但是正如上一節一開始我說的，MFC 之中用來處理訊息的 C++ 類別並不呈單鞭發展，作為 application framework 的重要架構之一的 document/view，也具有處理訊息的能力（你現在可能還不清楚什麼是 document/view，沒有關係）；因此，訊息應該有橫向流動的機會。MFC 對於訊息繞行的規定是：

- 如果是一般的 Windows 訊息（*WM_XXX*），一定是由衍生類別流向基礎類別，沒有旁流的可能。
- 如果是命令訊息 *WM_COMMAND*，就有奇特的路線了：

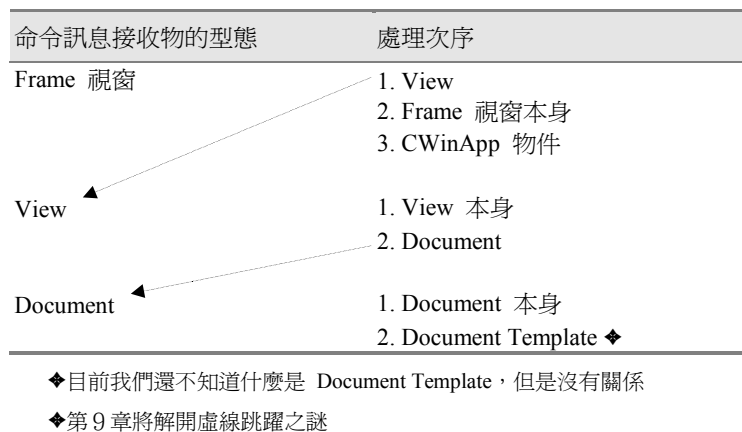


圖 3-6 MFC 對於命令訊息 *WM_COMMAND* 的特殊處理順序。

不管這個規則是怎麼定下來的，現在我要設計一個推動引擎，把它模擬出來。以下這些函式名稱以及函式內容，完全模擬 MFC 內部。有些函式似乎贅餘，那是因為我刪掉了許多主題以外的動作。不把看似贅餘的函式拿掉或合併，是爲了留下 MFC 的足跡。此外，爲了追蹤呼叫過程（call stack），我在各函式的第一行輸出一串識別文字。

首先我把新增加的一些成員函式做個列表：

類別	與訊息繞行有關的成員函式	注意
none	<i>AfxWndProc</i>	global
none	<i>AfxCallWndProc</i>	global
<i>CcmdTarget</i>	<i>OnCmdMsg</i>	virtual
<i>CDocument</i>	<i>OnCmdMsg</i>	virtual
<i>CWnd</i>	<i>WindowProc</i>	virtual
	<i>OnCommand</i>	virtual
	<i>DefWindowProc</i>	virtual
<i>CFrameWnd</i>	<i>OnCommand</i>	virtual
	<i>OnCmdMsg</i>	virtual
<i>CView</i>	<i>OnCmdMsg</i>	virtual

全域函式 *AfxWndProc* 就是我所謂的推動引擎的起始點。它本來應該是在 *CWinThread::Run* 中被呼叫，但爲了實驗目的，我在 *main* 中呼叫它，每呼叫一次便推送一個訊息。這個函式在 MFC 中有四個參數，爲了方便，我加上第五個，用以表示是誰獲得訊息（成爲繞行的起點）。例如：

```
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
```

表示 *pMyFrame* 獲得了一個 *WM_CREATE*，而：

```
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
```

表示 *pMyView* 獲得了一個 *WM_COMMAND*。

下面是訊息流動的過程：

```

LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
                  CWnd *pWnd) // last param. pWnd is added by JJHou.
{
    cout << "AfxWndProc()" << endl;
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}

LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
                      WPARAM wParam, LPARAM lParam)
{
    cout << "AfxCallWndProc()" << endl;
    LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
    return lResult;
}

```

pWnd->WindowProc 究竟是呼叫哪一個函式？不一定，得視 *pWnd* 到底指向何種類別之物件而定 -- 別忘了 *WindowProc* 是虛擬函式。這正是虛擬函式發揮它功效的地方呀：

- 如果 *pWnd* 指向 *CMyFrameWnd* 物件，那麼呼叫的是 *CFrameWnd::WindowProc*。而因為 *CFrameWnd* 並沒有改寫 *WindowProc*，所以呼叫的其實是 *CWnd::WindowProc*。
- 如果 *pWnd* 指向 *CMyView* 物件，那麼呼叫的是 *CView::WindowProc*。而因為 *CView* 並沒有改寫 *WindowProc*，所以呼叫的其實是 *CWnd::WindowProc*。

雖然殊途同歸，意義上是不相同的。切記！切記！

CWnd::WindowProc 首先判斷訊息是否為 *WM_COMMAND*。如果不是，事情最單純，就把訊息往父類別推去，父類別再往祖父類別推去。每到一個類別的訊息映射表，原本應該比對 *AFX_MSGMAP_ENTRY* 的每一個元素，比對成功就呼叫對應的處理常式。不過在這裡我不作比對，只是把 *AFX_MSGMAP_ENTRY* 中的類別識別代碼印出來（就像上一節的 *Frame7* 程式一樣），以表示「到此一遊」：

```
LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;

    if (nMsg == WM_COMMAND) // special case for commands
    {
        if (OnCommand(wParam, lParam)) ❶
            return 1L; // command handled
        else
            return (LRESULT)DefWindowProc(nMsg, wParam, lParam); ❹
    }

    pMessageMap = GetMessageMap();

    for (; pMessageMap != NULL;
        pMessageMap = pMessageMap->pBaseMessageMap)
    {
        lpEntry = pMessageMap->lpEntries;
        printlpEntries(lpEntry);
    }
    return 0; // J.J.Hou: if find, should call lpEntry->pfn,
              // otherwise should call DefWindowProc.
              // for simplification, we just return 0.
}
```

如果訊息是 `WM_COMMAND`，`CWnd::WindowProc` 呼叫 ❶ `OnCommand`。好，注意了，這又是一個 `CWnd` 的虛擬函式：

1. 如果 `this` 指向 `CMyFrameWnd` 物件，那麼呼叫的是 `CFrameWnd::OnCommand`。
2. 如果 `this` 指向 `CMyView` 物件，那麼呼叫的是 `CView::OnCommand`。而因為 `CView` 並沒有改寫 `OnCommand`，所以呼叫的其實是 `CWnd::OnCommand`。

這次可就沒有殊途同歸了。

我們以第一種情況為例，再往下看：

```

BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CFrameWnd::OnCommand()" << endl;
    // ...
    // route as normal command
    return CWnd::OnCommand(wParam, lParam); ❷
}

BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
{
    cout << "CWnd::OnCommand()" << endl;
    // ...
    return OnCmdMsg(0, 0); ❸
}

```

又一次遭遇虛擬函式。經過前兩次的分析，相信你對此很有經驗了。❸ *OnCmdMsg* 是 *CCmdTarget* 的虛擬函式，所以：

1. 如果 *this* 指向 *CMyFrameWnd* 物件，那麼呼叫的是 *CFrameWnd::OnCmdMsg*。
2. 如果 *this* 指向 *CMyView* 物件，那麼呼叫的是 *CView::OnCmdMsg*。
3. 如果 *this* 指向 *CMyDoc* 物件，那麼呼叫的是 *CDocument::OnCmdMsg*。
4. 如果 *this* 指向 *CMyWinApp* 物件，那麼呼叫的是 *CWinApp::OnCmdMsg*。而因為 *CWinApp* 並沒有改寫 *OnCmdMsg*，所以呼叫的其實是 *CCmdTarget::OnCmdMsg*。

目前的情況是第一種，於是呼叫 *CFrameWnd::OnCmdMsg*：

```

BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CFrameWnd::OnCmdMsg()" << endl;
    // pump through current view FIRST
    CView* pView = GetActiveView();
    if (pView->OnCmdMsg(nID, nCode)) ❹
        return TRUE;

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

    // last but not least, pump through app
    CWinApp* pApp = AfxGetApp();
}

```

```
        if (pApp->OnCmdMsg(nID, nCode)) ❸
            return TRUE;

        return FALSE;
    }
```

這個函式反應出圖 3-6 Frame 視窗處理 *WM_COMMAND* 的次序。最先呼叫的是 ❹ *pView->OnCmdMsg*，於是：

```
BOOL CView::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CView::OnCmdMsg()" << endl;
    if (CWnd::OnCmdMsg(nID, nCode)) ❺
        return TRUE;

    BOOL bHandled = FALSE;
    bHandled = m_pDocument->OnCmdMsg(nID, nCode); ❻
    return bHandled;
}
```

這又反應出圖 3-6 View 視窗處理 *WM_COMMAND* 的次序。最先呼叫的是 ❺ *CWnd::OnCmdMsg*，而 *CWnd* 並未改寫 *OnCmdMsg*，所以其實就是呼叫 *CCmdTarget::OnCmdMsg*：

```
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CCmdTarget::OnCmdMsg()" << endl;
    // Now look through message map to see if it applies to us
    AFX_MSGMAP* pMessageMap;
    AFX_MSGMAP_ENTRY* lpEntry;
    for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
         pMessageMap = pMessageMap->pBaseMessageMap)
    {
        lpEntry = pMessageMap->lpEntries;
        printlpEntries(lpEntry);
    }

    return FALSE;    // not handled
}
```

這是一個走訪訊息映射表的動作。注意，*GetMessageMap* 也是個虛擬函式（隱藏在 *DECLARE_MESSAGE_MAP* 巨集定義中），所以它所得到的訊息映射表將是 *this*（以目前而言是 *pMyView*）所指物件的映射表。於是我們得到了這個結果：

```
pMyFrame received a WM_COMMAND, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
CFrameWnd::OnCommand()
CWnd::OnCommand()
CFrameWnd::OnCmdMsg()
CFrameWnd::GetActiveView()
CView::OnCmdMsg()
CCmdTarget::OnCmdMsg()
1221    CMyView
122    CView
12    CWnd
1    CCmdTarget
```

如果在映射表中找到了對應的訊息，就呼叫對應的處理常式，然後也就結束了二萬五千里長征。如果沒找到，長征還沒有結束，這時候退守回到 *CView::OnCmdMsg*，呼叫

❶ *CDocument::OnCmdMsg*：

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
{
    cout << "CDocument::OnCmdMsg()" << endl;
    if (CCmdTarget::OnCmdMsg(nID, nCode))
        return TRUE;

    return FALSE;
}
```

於是得到這個結果：

```
CDocument::OnCmdMsg()
CCmdTarget::OnCmdMsg()
131    CMyDoc
13    CDocument
1    CCmdTarget
```

如果在映射表中還是沒找到對應訊息，二萬五千里長征還是未能結束，這時候退守回到 *CFrameWnd::OnCmdMsg*，呼叫 ❶ *CWnd::OnCmdMsg*（也就是 *CCmdTarget::OnCmdMsg*），得到這個結果：

```
CCmdTarget::OnCmdMsg()  
1211    CMyFrameWnd  
121    CFrameWnd  
12    CWnd  
1    CCmdTarget
```

如果在映射表中還是沒找到對應訊息，二萬五千里長征還是未能結束，再退回到 *CFrameWnd::OnCmdMsg*，呼叫 ❷ *CWinApp::OnCmdMsg*（亦即 *CCmdTarget::OnCmdMsg*），得到這個結果：

```
1111    CMyWinApp  
111    CWinApp  
1    CCmdTarget
```

萬一還是沒找到對應的訊息，二萬五千里長征可也窮途末路了，退回到 *CWnd::WindowProc*，呼叫 ❸ *CWnd::DefWindowProc*。你可以想像，在真正的 MFC 中這個成員函式必是呼叫 Windows API 函式 *::DefWindowProc*。爲了簡化，我讓它在 *Frame8* 中是個空函式。

故事結束！

我以圖 3-7 表示這二萬五千里長征的呼叫次序（call stack），圖 3-8 表示這二萬五千里長征的訊息流動路線。

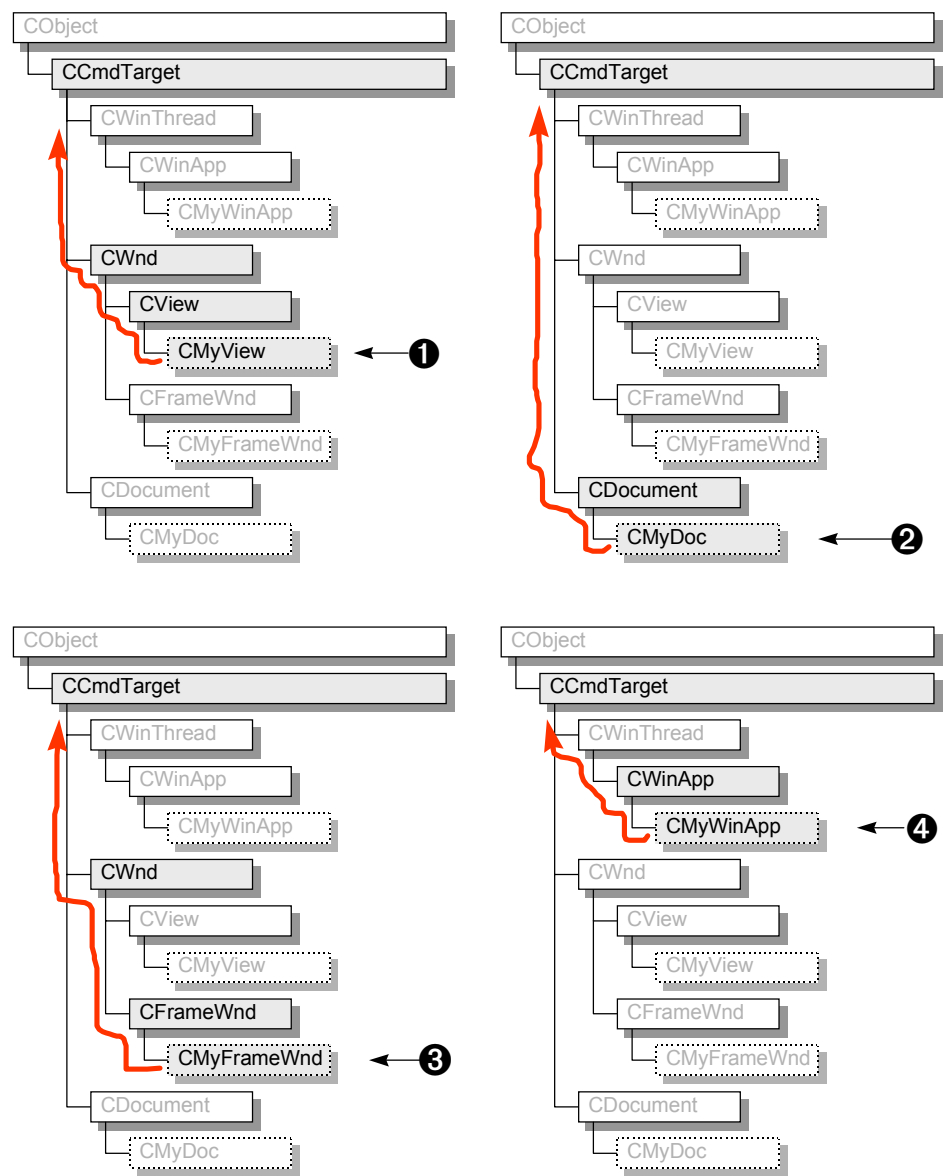


圖 3-8 當 `CMyFrameWnd` 物件獲得一個 `WM_COMMAND`，所引起的訊息流動路線。

Frame8 測試四種情況：分別從 frame 物件和 view 物件中推動訊息，訊息分一般 Windows 訊息和 *WM_COMMAND* 兩種：

```
// test Message Routing
AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
```

Frame8 的命令列編譯聯結動作是（環境變數必須先設定好，請參考第4章的「安裝與設定」一節）：

```
cl my.cpp mfc.cpp <Enter>
```

以下是 Frame8 的執行結果：

```
CWinApp::InitApplication
CMyWinApp::InitInstance
CFrameWnd::Create
CWnd::CreateEx
CFrameWnd::PreCreateWindow
CWinApp::Run
CWinThread::Run

pMyFrame received a WM_CREATE, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1211 CMyFrameWnd
121 CFrameWnd
12 CWnd
1 CCmdTarget

pMyView received a WM_PAINT, routing path and call stack:
AfxWndProc()
AfxCallWndProc()
CWnd::WindowProc()
1221 CMyView
122 CView
12 CWnd
1 CCmdTarget

pMyView received a WM_COMMAND, routing path and call stack:
AfxWndProc()
```

```
AfxCallWndProc()  
CWnd::WindowProc()  
CWnd::OnCommand()  
CView::OnCmdMsg()  
CCmdTarget::OnCmdMsg()  
1221   CMyView  
122   CView  
12   CWnd  
1   CCmdTarget  
CDocument::OnCmdMsg()  
CCmdTarget::OnCmdMsg()  
131   CMyDoc  
13   CDocument  
1   CCmdTarget  
CWnd::DefWindowProc()
```

pMyFrame received a WM_COMMAND, routing path and call stack:

```
AfxWndProc()  
AfxCallWndProc()  
CWnd::WindowProc()  
CFrameWnd::OnCommand()  
CWnd::OnCommand()  
CFrameWnd::OnCmdMsg()  
CFrameWnd::GetActiveView()  
CView::OnCmdMsg()  
CCmdTarget::OnCmdMsg()  
1221   CMyView  
122   CView  
12   CWnd  
1   CCmdTarget  
CDocument::OnCmdMsg()  
CCmdTarget::OnCmdMsg()  
131   CMyDoc  
13   CDocument  
1   CCmdTarget  
CCmdTarget::OnCmdMsg()  
1211   CMyFrameWnd  
121   CFrameWnd  
12   CWnd  
1   CCmdTarget  
CCmdTarget::OnCmdMsg()  
1111   CMyWinApp  
111   CWinApp  
1   CCmdTarget  
CWnd::DefWindowProc()
```

Frame8 範例程式

MFC.H

```

#0001 #define TRUE 1
#0002 #define FALSE 0
#0003
#0004 typedef char* LPSTR;
#0005 typedef const char* LPCSTR;
#0006
#0007 typedef unsigned long DWORD;
#0008 typedef int BOOL;
#0009 typedef unsigned char BYTE;
#0010 typedef unsigned short WORD;
#0011 typedef int INT;
#0012 typedef unsigned int UINT;
#0013 typedef long LONG;
#0014
#0015 typedef UINT WPARAM;
#0016 typedef LONG LPARAM;
#0017 typedef LONG LRESULT;
#0018 typedef int HWND;
#0019
#0020 #define WM_COMMAND 0x0111
#0021 #define WM_CREATE 0x0001
#0022 #define WM_PAINT 0x000F
#0023 #define WM_NOTIFY 0x004E
#0024
#0025 #define CObjectid 0xffff
#0026 #define CCmdTargetid 1
#0027 #define CWinThreadid 11
#0028 #define CWinAppid 111
#0029 #define CMyWinAppid 1111
#0030 #define CWndid 12
#0031 #define CFrameWndid 121
#0032 #define CMyFrameWndid 1211
#0033 #define CViewid 122
#0034 #define CMyViewid 1221
#0035 #define CDocumentid 13
#0036 #define CMyDocid 131
#0037
#0038 #include <iostream.h>
#0039
#0040 //////////////////////////////////////

```

```
#0041 // Window message map handling
#0042
#0043 struct AFX_MSGMAP_ENTRY;          // declared below after CWnd
#0044
#0045 struct AFX_MSGMAP
#0046 {
#0047     AFX_MSGMAP* pBaseMessageMap;
#0048     AFX_MSGMAP_ENTRY* lpEntries;
#0049 };
#0050
#0051 #define DECLARE_MESSAGE_MAP() \
#0052     static AFX_MSGMAP_ENTRY _messageEntries[]; \
#0053     static AFX_MSGMAP messageMap; \
#0054     virtual AFX_MSGMAP* GetMessageMap() const;
#0055
#0056 #define BEGIN_MESSAGE_MAP(theClass, baseClass) \
#0057     AFX_MSGMAP* theClass::GetMessageMap() const \
#0058     { return &theClass::messageMap; } \
#0059     AFX_MSGMAP theClass::messageMap = \
#0060     { &(baseClass::messageMap), \
#0061       (AFX_MSGMAP_ENTRY*) &(theClass::_messageEntries) }; \
#0062     AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
#0063     {
#0064
#0065 #define END_MESSAGE_MAP() \
#0066     { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
#0067     };
#0068
#0069 // Message map signature values and macros in separate header
#0070 #include "afxmsg_.h"
#0071
#0072 class CObject
#0073 {
#0074 public:
#0075     CObject::CObject() {
#0076     }
#0077     CObject::~~CObject() {
#0078     }
#0079 };
#0080
#0081 class CCmdTarget : public CObject
#0082 {
#0083 public:
#0084     CCmdTarget::CCmdTarget() {
#0085     }
#0086     CCmdTarget::~~CCmdTarget() {
```

```
#0087         }
#0088
#0089     virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0090
#0091     DECLARE_MESSAGE_MAP()          // base class - no {{ }} macros
#0092 };
#0093
#0094 typedef void (CCmdTarget::*AFX_PMSG)(void);
#0095
#0096 struct AFX_MSGMAP_ENTRY // MFC 4.0
#0097 {
#0098     UINT nMessage; // windows message
#0099     UINT nCode;    // control code or WM_NOTIFY code
#0100     UINT nID;      // control ID (or 0 for windows messages)
#0101     UINT nLastID;  // used for entries specifying a range of control id's
#0102     UINT nSig;     // signature type (action) or pointer to message #
#0103     AFX_PMSG pfn;  // routine to call (or special value)
#0104 };
#0105
#0106 class CWinThread : public CCmdTarget
#0107 {
#0108 public:
#0109     CWinThread::CWinThread() {
#0110     }
#0111     CWinThread::~CWinThread() {
#0112     }
#0113
#0114     virtual BOOL InitInstance() {
#0115         cout << "CWinThread::InitInstance \n";
#0116         return TRUE;
#0117     }
#0118     virtual int Run() {
#0119         cout << "CWinThread::Run \n";
#0120         // AfxWndProc(...);
#0121         return 1;
#0122     }
#0123 };
#0124
#0125 class CWnd;
#0126
#0127 class CWinApp : public CWinThread
#0128 {
#0129 public:
#0130     CWinApp* m_pCurrentWinApp;
#0131     CWnd* m_pMainWnd;
#0132
```

```
#0133 public:
#0134     CWinApp::CWinApp() {
#0135         m_pCurrentWinApp = this;
#0136     }
#0137     CWinApp::~CWinApp() {
#0138     }
#0139
#0140     virtual BOOL InitApplication() {
#0141         cout << "CWinApp::InitApplication \n";
#0142         return TRUE;
#0143     }
#0144     virtual BOOL InitInstance() {
#0145         cout << "CWinApp::InitInstance \n";
#0146         return TRUE;
#0147     }
#0148     virtual int Run() {
#0149         cout << "CWinApp::Run \n";
#0150         return CWinThread::Run();
#0151     }
#0152
#0153     DECLARE_MESSAGE_MAP()
#0154 };
#0155
#0156 typedef void (CWnd::*AFX_PMSGW)(void);
#0157         // like 'AFX_PMSG' but for CWnd derived classes only
#0158
#0159 class CDocument : public CCmdTarget
#0160 {
#0161 public:
#0162     CDocument::CDocument() {
#0163     }
#0164     CDocument::~CDocument() {
#0165     }
#0166
#0167     virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0168
#0169     DECLARE_MESSAGE_MAP()
#0170 };
#0171
#0172 class CWnd : public CCmdTarget
#0173 {
#0174 public:
#0175     CWnd::CWnd() {
#0176     }
#0177     CWnd::~CWnd() {
#0178     }
```

```
#0179
#0180     virtual BOOL Create();
#0181     BOOL CreateEx();
#0182     virtual BOOL PreCreateWindow();
#0183     virtual LRESULT WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam);
#0184     virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
#0185     virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0186
#0187     DECLARE_MESSAGE_MAP()
#0188 };
#0189
#0190 class CView;
#0191
#0192 class CFrameWnd : public CWnd
#0193 {
#0194 public:
#0195     CView* m_pViewActive;        // current active view
#0196
#0197 public:
#0198     CFrameWnd::CFrameWnd() {
#0199     }
#0200     CFrameWnd::~CFrameWnd() {
#0201     }
#0202     BOOL Create();
#0203     CView* GetActiveView() const;
#0204     virtual BOOL PreCreateWindow();
#0205     virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
#0206     virtual BOOL OnCmdMsg(UINT nID, int nCode);
#0207
#0208     DECLARE_MESSAGE_MAP()
#0209
#0210     friend CView;
#0211 };
#0212
#0213 class CView : public CWnd
#0214 {
#0215 public:
#0216     CDocument* m_pDocument;
#0217
#0218 public:
#0219     CView::CView() {
#0220     }
#0221     CView::~CView() {
#0222     }
#0223
#0224     virtual BOOL OnCmdMsg(UINT nID, int nCode);
```



```
#0225
#0226     DECLARE_MESSAGE_MAP()
#0227
#0228     friend CFrameWnd;
#0229 };
#0230
#0231 // global function
#0232 CWinApp* AfxGetApp();
#0233 LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0234                 CWnd* pWnd); // last param. pWnd is added by JJHOU.
#0235 LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg, WPARAM wParam,
#0236                        LPARAM lParam);
```

AFXMSG_.H

```
#0001 enum AfxSig
#0002 {
#0003     AfxSig_end = 0,      // [marks end of message map]
#0004     AfxSig_vv,
#0005 };
#0006
#0007 #define ON_COMMAND(id, memberFxn) \
#0008     { WM_COMMAND, 0, (WORD)id, (WORD)id, AfxSig_vv, (AFX_PMSG)memberFxn },
```

MFC.CPP

```
#0001 #include "my.h" // 原該含入 mfc.h 就好，但爲了 extern CMyWinApp 所以...
#0002
#0003 extern CMyWinApp theApp;
#0004 extern void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry);
#0005
#0006 BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode)
#0007 {
#0008     // Now look through message map to see if it applies to us
#0009     AFX_MSGMAP* pMessageMap;
#0010     AFX_MSGMAP_ENTRY* lpEntry;
#0011     for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
#0012         pMessageMap = pMessageMap->pBaseMessageMap)
#0013     {
#0014         lpEntry = pMessageMap->lpEntries;
#0015         printlpEntries(lpEntry);
#0016     }
#0017     return FALSE; // not handled
#0018
#0019 }
```

```
#0020
#0021 BOOL CWnd::Create()
#0022 {
#0023     cout << "CWnd::Create \n";
#0024     return TRUE;
#0025 }
#0026
#0027 BOOL CWnd::CreateEx()
#0028 {
#0029     cout << "CWnd::CreateEx \n";
#0030     PreCreateWindow();
#0031     return TRUE;
#0032 }
#0033
#0034 BOOL CWnd::PreCreateWindow()
#0035 {
#0036     cout << "CWnd::PreCreateWindow \n";
#0037     return TRUE;
#0038 }
#0039
#0040 LRESULT CWnd::WindowProc(UINT nMsg, WPARAM wParam, LPARAM lParam)
#0041 {
#0042     AFX_MSGMAP* pMessageMap;
#0043     AFX_MSGMAP_ENTRY* lpEntry;
#0044
#0045     if (nMsg == WM_COMMAND) // special case for commands
#0046     {
#0047         if (OnCommand(wParam, lParam))
#0048             return 1L; // command handled
#0049         else
#0050             return (LRESULT)DefWindowProc(nMsg, wParam, lParam);
#0051     }
#0052
#0053     pMessageMap = GetMessageMap();
#0054
#0055     for (; pMessageMap != NULL;
#0056          pMessageMap = pMessageMap->pBaseMessageMap)
#0057     {
#0058         lpEntry = pMessageMap->lpEntries;
#0059         printlpEntries(lpEntry);
#0060     }
#0061     return 0; // add by JJHou. if find, should call lpEntry->pfn,
#0062              // otherwise should call DefWindowProc.
#0063 }
#0064
#0065 LRESULT CWnd::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
```

```
#0066 {
#0067     return TRUE;
#0068 }
#0069
#0070 BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0071 {
#0072     // ...
#0073     return OnCmdMsg(0, 0);
#0074 }
#0075
#0076 BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
#0077 {
#0078     // ...
#0079     // route as normal command
#0080     return CWnd::OnCommand(wParam, lParam);
#0081 }
#0082
#0083 BOOL CFrameWnd::Create()
#0084 {
#0085     cout << "CFrameWnd::Create \n";
#0086     CreateEx();
#0087     return TRUE;
#0088 }
#0089
#0090 BOOL CFrameWnd::PreCreateWindow()
#0091 {
#0092     cout << "CFrameWnd::PreCreateWindow \n";
#0093     return TRUE;
#0094 }
#0095
#0096 CView* CFrameWnd::GetActiveView() const
#0097 {
#0098     return m_pViewActive;
#0099 }
#0100
#0101 BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode)
#0102 {
#0103     // pump through current view FIRST
#0104     CView* pView = GetActiveView();
#0105     if (pView->OnCmdMsg(nID, nCode))
#0106         return TRUE;
#0107
#0108     // then pump through frame
#0109     if (CWnd::OnCmdMsg(nID, nCode))
#0110         return TRUE;
#0111 }
```

```
#0112 // last but not least, pump through app
#0113 CWinApp* pApp = AfxGetApp();
#0114 if (pApp->OnCmdMsg(nID, nCode))
#0115     return TRUE;
#0116
#0117 return FALSE;
#0118 }
#0119
#0120 BOOL CDocument::OnCmdMsg(UINT nID, int nCode)
#0121 {
#0122     if (CCmdTarget::OnCmdMsg(nID, nCode))
#0123         return TRUE;
#0124
#0125     return FALSE;
#0126 }
#0127
#0128 BOOL CView::OnCmdMsg(UINT nID, int nCode)
#0129 {
#0130     if (CWnd::OnCmdMsg(nID, nCode))
#0131         return TRUE;
#0132
#0133     BOOL bHandled = FALSE;
#0134     bHandled = m_pDocument->OnCmdMsg(nID, nCode);
#0135     return bHandled;
#0136 }
#0137
#0138 AFX_MSGMAP* CCmdTarget::GetMessageMap() const
#0139 {
#0140     return &CCmdTarget::messageMap;
#0141 }
#0142
#0143 AFX_MSGMAP CCmdTarget::messageMap =
#0144 {
#0145     NULL,
#0146     &CCmdTarget::_messageEntries[0]
#0147 };
#0148
#0149 AFX_MSGMAP_ENTRY CCmdTarget::_messageEntries[] =
#0150 {
#0151
#0152     { 0, 0, CCmdTargetid, 0, AfxSig_end, 0 }
#0153 };
#0154
#0155 BEGIN_MESSAGE_MAP(CWnd, CCmdTarget)
#0156 ON_COMMAND(CWndid, 0)
#0157 END_MESSAGE_MAP()
```

```
#0158
#0159 BEGIN_MESSAGE_MAP(CFrameWnd, CWnd)
#0160 ON_COMMAND(CFrameWndid, 0)
#0161 END_MESSAGE_MAP()
#0162
#0163 BEGIN_MESSAGE_MAP(CDocument, CCmdTarget)
#0164 ON_COMMAND(CDocumentid, 0)
#0165 END_MESSAGE_MAP()
#0166
#0167 BEGIN_MESSAGE_MAP(CView, CWnd)
#0168 ON_COMMAND(CViewid, 0)
#0169 END_MESSAGE_MAP()
#0170
#0171 BEGIN_MESSAGE_MAP(CWinApp, CCmdTarget)
#0172 ON_COMMAND(CWinAppid, 0)
#0173 END_MESSAGE_MAP()
#0174
#0175 CWinApp* AfxGetApp()
#0176 {
#0177     return theApp.m_pCurrentWinApp;
#0178 }
#0179
#0180 LRESULT AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam,
#0181                  CWnd *pWnd) // last parameter pWnd is added by JJHou.
#0182 {
#0183     //...
#0184     return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
#0185 }
#0186
#0187 LRESULT AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
#0188                       WPARAM wParam, LPARAM lParam)
#0189 {
#0189     LRESULT lResult = pWnd->WindowProc(nMsg, wParam, lParam);
#0190     return lResult;
#0191 }
```

MY.H

```
#0001 #include <iostream.h>
#0002 #include "mfc.h"
#0003
#0004 class CMyWinApp : public CWinApp
#0005 {
#0006 public:
#0007     CMyWinApp():CMyWinApp() {
#0008     }
```

```

#0009  CMyWinApp::~CMyWinApp() {
#0010      }
#0011  virtual BOOL InitInstance();
#0012  DECLARE_MESSAGE_MAP()
#0013  };
#0014
#0015  class CMyFrameWnd : public CFrameWnd
#0016  {
#0017  public:
#0018      CMyFrameWnd();
#0019      ~CMyFrameWnd() {
#0020      }
#0021      DECLARE_MESSAGE_MAP()
#0022  };
#0023
#0024  class CMyDoc : public CDocument
#0025  {
#0026  public:
#0027      CMyDoc::CMyDoc() {
#0028      }
#0029      CMyDoc::~~CMyDoc() {
#0030      }
#0031      DECLARE_MESSAGE_MAP()
#0032  };
#0033
#0034  class CMyView : public CView
#0035  {
#0036  public:
#0037      CMyView::CMyView() {
#0038      }
#0039      CMyView::~~CMyView() {
#0040      }
#0041      DECLARE_MESSAGE_MAP()
#0042  };

```

MY.CPP

```

#0001  #include "my.h"
#0002
#0003  CMyWinApp theApp; // global object
#0004
#0005  BOOL CMyWinApp::InitInstance()
#0006  {
#0007      cout << "CMyWinApp::InitInstance \n";
#0008      m_pMainWnd = new CMyFrameWnd;
#0009      return TRUE;

```

```
#0010 }
#0011
#0012 CMyFrameWnd::CMyFrameWnd()
#0013 {
#0014     Create();
#0015 }
#0016
#0017 BEGIN_MESSAGE_MAP(CMyWinApp, CWinApp)
#0018 ON_COMMAND(CMyWinAppid, 0)
#0019 END_MESSAGE_MAP()
#0020
#0021 BEGIN_MESSAGE_MAP(CMyFrameWnd, CFrameWnd)
#0022 ON_COMMAND(CMyFrameWndid, 0)
#0023 END_MESSAGE_MAP()
#0024
#0025 BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
#0026 ON_COMMAND(CMyDocid, 0)
#0027 END_MESSAGE_MAP()
#0028
#0029 BEGIN_MESSAGE_MAP(CMyView, CView)
#0030 ON_COMMAND(CMyViewid, 0)
#0031 END_MESSAGE_MAP()
#0032
#0033 void printlpEntries(AFX_MSGMAP_ENTRY* lpEntry)
#0034 {
#0035     struct {
#0036         int classid;
#0037         char* classname;
#0038     } classinfo[] = {
#0039         CCmdTargetid , "CCmdTarget  ",
#0040         CWinThreadid , "CWinThread  ",
#0041         CWinAppid    , "CWinApp    ",
#0042         CMyWinAppid  , "CMyWinApp  ",
#0043         CWndid       , "CWnd       ",
#0044         CFrameWndid  , "CFrameWnd  ",
#0045         CMyFrameWndid, "CMyFrameWnd",
#0046         CViewid      , "CView      ",
#0047         CMyViewid    , "CMyView    ",
#0048         CDocumentid  , "CDocument  ",
#0049         CMyDocid     , "CMyDoc     ",
#0050         0            , "            ",
#0051     };
#0052
#0053     for (int i=0; classinfo[i].classid != 0; i++)
#0054     {
#0055         if (classinfo[i].classid == lpEntry->nID)
```

```
#0056     {
#0057         cout << lpEntry->nID << "    ";
#0058         cout << classinfo[i].classname << endl;
#0059         break;
#0060     }
#0061 }
#0062 }
#0063 //-----
#0064 // main
#0065 //-----
#0066 void main()
#0067 {
#0068     CWinApp* pApp = AfxGetApp();
#0069
#0070     pApp->InitApplication();
#0071     pApp->InitInstance();
#0072     pApp->Run();
#0073
#0074     CMyDoc* pMyDoc = new CMyDoc;
#0075     CMyView* pMyView = new CMyView;
#0076     CFrameWnd* pMyFrame = (CFrameWnd*)pApp->m_pMainWnd;
#0077     pMyFrame->m_pViewActive = pMyView;
#0078     pMyView->m_pDocument = pMyDoc;
#0079
#0080     // test Message Routing
#0081     cout << endl << "pMyFrame received a WM_CREATE, routing path : " << endl;
#0082     AfxWndProc(0, WM_CREATE, 0, 0, pMyFrame);
#0083
#0084     cout << endl << "pMyView received a WM_PAINT, routing path : " << endl;
#0085     AfxWndProc(0, WM_PAINT, 0, 0, pMyView);
#0086
#0087     cout << endl << "pMyView received a WM_COMMAND, routing path : " << endl;
#0088     AfxWndProc(0, WM_COMMAND, 0, 0, pMyView);
#0089
#0090     cout << endl << "pMyFrame received a WM_COMMAND, routing path : " << endl;
#0091     AfxWndProc(0, WM_COMMAND, 0, 0, pMyFrame);
#0092 }
```


本書回顧

像外科手術一樣精準，我們拿起鋒利的刀子，劃開 MFC 堅韌的皮膚，再一刀下去，剖開它的肌理。掏出它的內臟，反覆觀察研究。終於，藉著從 MFC 掏挖出來的原始碼清洗整理後完成的幾個小小的 C++ console 程式，我們徹底瞭解了所謂 Runtime Class、Runtime Time Information、Dynamic Creation、Message Mapping、Command Routing 的內部機制。

咱們並不是要學著做一套 application framework，但是這樣的學習過程確實有必要。因為，「只用一樣東西，不明白它的道理，實在不高明」。況且，有什麼比光靠三五個一兩百行小程式，就搞定物件導向領域中的高明技術，更值得的事？有什麼比欣賞那些由 Runtime Class 所構成的「類別型錄網」示意圖、訊息的實際流動圖、訊息映射表的架構圖，更令人心曠神怡？

把 Frame1~Frame8 好好研究一遍，你已經對 MFC 的架構成竹在胸。再來，就是 MFC 類別的實際運用，以及 Visual C++ 工具的熟練囉。