

GOTOP



編者
陳昇瑋

C++ Builder



深度體驗



碁峯
www.gotop.com.tw



陳昇瑋
(原名：陳寬達)



姓名標示-非商業性-相同方式分享 2.5

您可自由：

- 重製、散布、展示及演出本著作
- 創作衍生著作

惟需遵照下列條件：



姓名標示. 您必須按照作者或授權人所指定的方式，保留其姓名標示。



非商業性. 您不得為商業目的而使用本著作。



相同方式分享. 若您改變、轉變或改作本著作，僅在遵守與本著作相同的授權條款下，您始得散布由本著作而生的衍生著作。

- 為再使用或散布本著作，您必須向他人清楚說明本著作所適用的授權條款。
- 如果您取得著作權人之許可，這些條件中任一項都能被免除。

您合理使用的權利及其他的權利，不因上述內容而受影響。

這是一份讓一般人易於了解的[法律條款（完整的授權條款）](#)摘要。

[免責聲明](#) 

謹將此書獻給

最親愛的父母及家人

侯捷 序

我很高興將寬達這個人和他的這本書，介紹給還不認識他的人。

● 關於人

寬達是個熱血青年！熱情表現在他對技術寫作、軟體創作、心得分享、排難解惑上面。

1997.09 在網路上發表「拋磚引玉」一文後，我收到寬達的來信，他說看了文章後再也忍不住，所以寫信給我。我因此結識寬達、達智、匡正、蘇轍等一批 Delphi 好手。

此後我們一直保有聯絡 -- 通常是我請教他網路觀念以及各種工具的使用居多。寬達精通的東西多樣，面向很廣；Delphi / C++Builder 是他最喜歡也最上手的開發工具，我從他（以及那批高手）身上，知道了這兩套開發工具的神奇魔力。

寬達年紀還輕，閱歷卻已很廣，不僅（協同）主持極有名的「Delphi 深度歷險」技術性網站、寫過不少雜誌稿、參加過 ACM 程式設計大賽，也曾擔任雜誌編輯，並實際為業界開發過一些案子。現在，他的成績單上又多了一項：著書。

這雖然是尚在學生身份的他的第一本書，我卻看到經驗豐富、文筆流暢的專業水準。

● 關於書

我一直期盼看到一種電腦書籍，專講程式開發的各色經驗與心得。這種書的英文書名大抵會被冠上“workshop”字眼，通常是多人的心血結晶，結集成冊。

寬達這本書有 workshop 的味道，只不過所有內容他一個人全包了。☺

這本書並不把重點放在 C++Builder 本身的學習上。本書特色是，作者將各個主題的個人經驗、深度觀察、實作心得整理下來，與讀者分享。這是一本有趣的書，納入的主題是諸如計時器、控制台、桌面技術、遊戲軟體...這類「生活化」的東西。千萬不要乍見之下以為這些只是雕蟲小技，以為對「嚴肅的」軟體開發專案於事無補。在極重視使用介面（UI）以及便利性、親和性的現代軟體開發觀念中，這些看似與「嚴肅的」軟體專案無關的題目，背後隱藏許多好點子，以及與 Windows 作業系統之間的深層對話。

我喜歡這本書！

侯捷 1999/11/29 于新竹
2002/01/23 重潤

寬達註：

侯大哥此序為 1999 年底拙著 Delphi 深度歷險 出版時所作，而今此書 C++Builder 深度歷險 即將出版，得侯大哥同意後沿用。VCL Team 所作之序情況亦同，不再贅述。

VCL Team 序

從首次見到寬達至今轉眼已四年，借用 Dracula 對寬達的四字形容詞，來表達四年來對這個有為年輕人貼切的感覺，那就是－「閃閃動人」。

也許我和 Dracula 已經過了充滿活力與憧憬的年齡，每當看到寬達對於喜愛的技術領域孜孜不倦的鑽研、對於網路族群無私無我的貢獻、對於理想毫不退縮地努力，「閃閃動人」的風采就會促使我倆重新反省自己對於生活的態度。在您手中的這本書；正是寬達實現自我理想、貢獻所學後所得到的結晶。

C++Builder 與其它 RAD 開發工具間最大的差異便是在於應用範圍的廣度與深度，絕大多數的使用者將 C++Builder 用於開發資料庫應用程式；並不代表 C++Builder 只適用在這個特定的領域，書中寬達選擇了多個有趣的實作主題，以輕鬆淺白的文筆導引讀者了解視窗系統背後運作的細節，倘若讀者對於系統程式設計領域不得其門而入，何不藉助已熟悉的 C++Builder 從窗外一窺奧秘呢？

深度歷險網站成立以來，寬達從旁協助不曾間斷。在此，我很樂意將網站名稱，獻給這本名實相符的好書，也樂於推薦每位使用 C++Builder 的朋友，由此書開啓另一個新眼界。

李匡正

¹ VCL Team的全名為"Very Cool & Lazy Team"，成員包含李匡正、錢達智、陳寬達及另一位實力深不可測、深居簡出的Dracula Su。VCL Team的名號主要來自C++Builder所使用的application framework－VCL，而全名是依據我們共同的特性－Lazy而訂。:P

可惡的匡正，把我原本要講的話都說完了！;-)

寬達除了「閃閃動人」之外，還是個體力與腦力似乎永遠都發洩不完的「宇宙無敵超級過動兒」！他的程式讓我的電腦變得很炫很酷；他的文章帶給我的興奮，跟看 Andrew Schulman 與 Matt Pietrek 的書不相上下。跟他討論到與電腦有關的事情，總是讓我「廢寢忘食」，因為他一大早六點多就 Call 我，把我叫起床，有時一講就講好久好久，害我買的冬粉湯居然膨脹的跟炒烏龍沒什麼兩樣。更糟的是，他把 System-Level Programming 的技巧用在 User-Level Program 裡頭，所以你寫的程式不只在 User Interface 比不上他（因為他用 C++Builder、C++Builder 沒提供的元件自己寫、C++Builder 不足的再用 SDK 補強），在功能上也比不過他（你很少看到一般程式用到 Multi-Threading、Memory Mapped File、Object Persistence、DLL Injection 吧），不知道要我們這些 Programmer 怎麼混下去！

我相信，不管你平常喜愛、熟悉的是哪一種程式語言、開發工具，只要你是個有理想、有抱負的 Programmer，你都應該看看這本書。寬達在書裡頭也許只扯到 Windows Platform、也許只用了 C++Builder 做示範，但是這些我覺得都不是重點。重點是：你不能從他的文章中，學習到一個 Programmer 怎麼憑著自己的能力與手邊的工具，把一個想法落實為一套軟體的歷程。

VCL Team 已經有兩個人出書了。下一個會是誰呢？

蘇轍 (Dracula Su)

天份若是再加上狂熱，結果可能十分恐怖，不過，寬達卻不是一個恐怖的人，事實上，他十分迷人。

寬達迷人之處在於善於將其天份發揮在他善長的地方。如果你在街上遇到他，他看起來與鄰家男孩沒什麼兩樣，不過，只要一停下來與他聊起電腦，立刻就會感受到他的神采奕奕。我想，在你開始閱讀本書後，一定就能體會到我所謂的「神采奕奕」是什麼意思。

這種「神采奕奕」的感覺，會隨著閱讀的持續進行而逐漸加強，最後終於演變成贊嘆的程度。自從認識寬達以來，每當他聊起最近又讀了哪些書、寫了什麼程式（包括文章）、接了什麼案子、參加什麼比賽、教了哪些課程...時，那種充分發揮天份時自信神采，簡直是太迷人了，只是，在每次之驚異之餘，卻也不時折磨著我早已傷亡殆盡的創意細胞，幾次下來，對他簡直到了欽佩的地步。

很喜愛這本好書，也推薦給您一同分享，在這本書中，除了技術，還有一位年少天才的創意與自信。

錢達智

自序

“To have fun”，這是我想要表達的。

在程式設計的領域內，我向來不崇拜技術高超的專業作家或軟體開發人員，頂多只是對其專業能力的讚嘆。真正吸引我的，是那些兼具知識、能力及赤子之心的玩家。

資訊科學界大師 Donald Knuth 為了撰寫他的 *The Art of Computer Programming*，想要用一個叫做 ladders 的英文單字遊戲貫穿全書。雖名為研究，結果他自個玩得不亦樂乎，還特別寫了一本專書 *The Stanford GraphBase*，想要介紹給大家一塊玩這個遊戲。

為學研究如此，程式設計也是。

對我而言，好奇心及創意源源不絕地湧入腦海，對世間萬物如是，對電腦裡頭的奇妙世界亦如是。滿足好奇心的最佳解藥是，徹底瞭解電腦軟硬體的運作方式；而讓創意盡情發揮的最佳方式是，拿出稱手的開發工具，一一將創意溶入軟體設計之中。

這本書，正是我在 Win32 作業系統中，乘著 C++Builder 魔毯恣意遊歷後所寫下的遊記。藉由它的記錄，希望你也能領略我所得的欣喜及見識。

陳寬達 2002/01/23 于南港

kuan@ilife.cx

致謝

都是你們，我可愛的朋友，滿腔的感謝使我不得不趁此篇幅稍稍宣洩...

侯大哥，您的努力，對於臺灣的電腦技術書籍產生深遠的影響；您的登高一呼，使我驚覺此領域的貧困缺乏，進而踏入技術寫作的園地。謝謝您。

Dracula 大哥，幸虧有您，否則這本書與我就要雙雙進入「錯字連篇排行榜」了。真的很謝謝您耐心地為我校閱每一個章節，細心地將每個錯誤的拼字、標點符號、大小寫、圖片編號等等，一一列出讓我比對正誤，您絕對是成就這本書的最大功臣。

匡正大哥，三年來，您每天花上幾個小時整理、更新 Delphi 深度歷險網站，我好佩服您這股不可思議的毅力及服務熱忱，在此代表全世界曾經受惠於深度歷險網站的 Delphi / C++Builder 同好，說聲謝謝您。您的確證明了您所說的：「如果你願意，憑一己之力就能夠讓這個世界變得更美好」。

達智大哥，我老是有什麼小問題就打電話吵您，或跟您要什麼文件檔案，謝謝您總是不厭其煩地幫助我解決問題。您的拼勁，您的鼓勵，也讓我不敢有絲毫鬆懈，畢竟我們同樣都屬於「達」字輩！:p

李維大哥，也許您已經忘了，不過，當初是您幫助我解決 *TWinSaver* 元件的棘手問題，讓我對 Delphi，對 C++Builder，對 VCL 產生莫大的興趣，才有今日成書的我。謝謝您。

金興昇大哥，要不是您強烈地鼓勵我，叫我將專欄文章整理後即刻出書，別再猶豫，否則至今我一定仍猶豫不決，無法靜下心來完成這本書。就像這本書，您的話一向給我

大的幫助，我只有由衷的感謝。

明新，捨專業美術設計人員而就四年未曾提筆繪畫的你，因為我相信你，謝謝你撥出那麼多準備期中考及作業的時間為我設計這本書的封面。我喜歡你為我在每篇每章前頭所繪的圖畫，高中三年同學果然不是當假的，我們只試著溝通想法及感覺，你就能很快地繪出我心中的那幅景象，真是帥呆了！

猩猩、鳥頭、小P、BB翔、踢小米，謝謝你們，在我這段閉關寫書時期，除了得經常忍受我半夜三更的音樂及燈光，還得經常帶食物回來「養」我，水餃、自助餐、吉野家、水木、麥當勞外加各式飲料，讓我連三餐奔波都省了，專心著作。雖然不是老媽或女友的照料，不過你們這群好朋友的體貼，我永遠忘不了。

飛哥、肯立、慶和，這本書總算完工，我會立刻回到崗位和你們並肩作戰的。謝謝你們在這段時期幫我做我份內的工作，而我還常常拿小事麻煩你們呢。趕快練好格鬥天王吧，我早已蓄勢待發了。:p

william、qing、jyhuang，你們是我在資訊科學領域內，一直效法學習的對象。謝謝你們的鼓勵及指正，我才擁有足夠的能量在學術和產業之間取得平衡，將最自然的我，表現出來。

彭彭、俊宏，謝謝你們的「探監」與建議。Sean、Cody、Raymond、LingJie，感謝你們的包容，肯讓我任性地暫時將這本書放在前頭，讓我原本該做的事延滯下來。另外謝謝碁峰出版社的Molly，我常為了芝麻蒜皮小事情不斷打電話麻煩忙得要死的妳，此書完成後，應該可讓妳耳根子清淨不少。:P

最後，對所有關心、幫助我的朋友們，說聲謝謝，我很幸運有你們的關懷。

目錄

侯捷 序	I
VCL Team 序	III
自序	VI
致謝	VII
目錄	IX
第零章 導讀	/001
第一篇 基礎觀念	
<hr/>	
第一章 RAD 無罪論	/009
第二章 VCL 基本心法	/033
第二篇 作業系統	
<hr/>	
第三章 控制你的控制台	/083
第四章 分秒必爭，細說計時器	/121
第三篇 桌面秘笈	
<hr/>	
第五章 一頭栽入桌面的世界	/179
第六章 佈景主題工具實戰	/243
第七章 螢幕保護？我用計劃表！	/301
第四篇 遊戲快打	
<hr/>	
第八章 足球番	/335
第九章 坦克大決戰	/397

第五篇 軟體開發

第十章 Fancy 軟體撰寫手則 /483

附錄

附錄A 我的程式庫 /535

附錄B 我的工具箱 /563

附錄C 參考書目 /595

第 0 章 導讀

這本書適合誰	/001
全書架構	/002
書籍體例與用語	/005
範例程式風格	/006
光碟內容	/007
介紹給你	/007
與作者連繫	/008

第一章 RAD 無罪論

不得不為的選擇	/012
狂熱份子的信仰	/013
學習動機	/013
目前基礎	/016
個人偏好	/016
RAD 的原罪	/020
開發工具的差異	/024
Win32 開發工具的演進	/024
RAD 無罪，輕鬆有理	/025
實作與理論	/026
參與者的類型	/026
參與者的落腳處	/028
這些技術是什麼？	/029
通通都在裡頭	/030
你看到了哪些？	/030

第二章 VCL 基本心法

C++Builder 程式的組成	/034
執行檔成分解析	/034
組成份子	/045
VCL 的多重面貌	/053
單身時期	/054
死會時期	/055
VCL 類別架構	/059
核心類別	/061
控制項類別	/072
程式運作類別	/077
RAD 支援類別	/079

第三章 控制你的控制台

控制台觀測站	/085
呼叫呼叫，聽到請回答！	/086
CPL 檔的真實身份	/091
行為剖析	/093
實作時間	/100
Hello, World !!	/100
撰寫自己的控制台	/105
VCL 的控制台支援	/113
新增的單元及類別	/113
範例元件－觀測記憶體使用狀況	/114
背後支援的 TAppletApplication	/118

第四章 分秒必爭，細說計時器

計時器 API	/123
建立計時器	/123
WM_TIMER 訊息	/124
消滅計時器	/125
視窗是必要的嗎？	/126
量測計時器的精確度	/128
更精確的計時器	/133
多媒體應用程式的需求	/133
取得解析度範圍	/134
視需求調整解析度	/135
使用多媒體計時器	/136
內部運作	/143
取得系統使用時間	/145
精益求精－高解析度效能計數器	/146
延遲函式	/149
八風請不動，只待時限到	/151
TApplication::ProcessMessages	/152
精確的延遲函式	/156
TTimer 元件	/161
內部剖析	/162
執行緒中的計時器	/164
工作執行緒隱含的陷阱	/165
撿到便宜的 TThread 建構函式	/168
解決工作執行緒的計時需求	/169
方案一：借助主執行緒的訊息迴圈	/169
方案二：使用不依賴視窗訊息的多媒體計時器	/170
方案三：使用可等待計時器	/172
方案四：使用可接受訊息的等待函式	/173

第五章 一頭栽入桌面的世界

桌面的構成	/179
唯一的桌面視窗	/181
它叫 Shell，不是貝殼	/183
桌面上的特殊視窗	/190
桌面上的把戲	/191
席捲桌面，氣吞四海	/191
桌面塗鴨程式	/194
畫面截取	/197
桌面隨意貼	/199
Desktop Illusionist	/207
源起	/208
程式目的	/209
程式手法	/210
控制桌面視窗是否繪製桌面底色、填圖樣式及桌布	/211
更改背景視窗的繪製動作	/213
位於桌面圖示下方的按鈕	/217
程式手法小結	/221
使用技術	/221
Subclassing	/222
Hook	/225
記憶體映射檔案	/235
程式撰寫	/238
成果品嘗	/240

第六章 佈景主題工具實戰

XTheme Manager 簡介	/248
認識佈景主題	/254

佈景元件	/254
佈景描述檔案	/255
XTheme Manager Lite	/261
功能設定	/261
介面設計	/262
系統顏色	/264
取得系統顏色	/268
設定系統顏色	/269
儲存設定值	/270
滑鼠指標	/271
取得滑鼠指標	/273
設定系統滑鼠指標	/276
儲存設定值	/277
系統音效	/278
事件敘述	/279
取得及設定系統音效	/279
播放系統音效	/279
系統字型	/280
取得及寫入系統字型	/281
各種字型物件的處理	/282
桌面圖示	/285
數到三，快快從桌面上消失...	/286
揮之不去的四劍客	/286
桌布及樣式	/290
填圖樣式	/290
桌布式樣設定	/291
範例程式－WallPaper Changer / Pattern Viewer	/294
桌布自動更換軟體	/295
有趣的 PaintDesktop API	/296

回到 XHTML	/297
預視功能	/297
成果大觀	/299

第七章 螢幕保護？我用計劃表！

知而後行	/302
螢幕保護程式的構成	/303
相關的系統登錄設定	/304
禁！螢幕保護退散	/305
啓動螢幕保護	/307
實作預備課程	/308
原來是個窗	/309
事件處理	/309
訊息攔截	/310
剖析參數	/311
運作核心	/311
預視功能	/312
功能設定	/313
取個響噹噹的好名字	/314
XEssay Screen Saver	/316
程式功能規劃	/316
Main Form 的設計	/317
建立核心執行緒	/318
提供預視功能	/320
設定對話盒	/326
剖析命令列參數	/327
編譯及執行	/329
安裝	/330
成果賞玩	/330

第八章 足球番

系統規劃	/337
TTiles 類別	/338
TMap 類別	/339
TRole 類別	/340
類別實作	/340
TTiles 圖庫類別及子類別	/341
TMap 地圖類別	/346
TRole 主角類別	/355
圖庫編輯器	/360
雙重「物」格的 FTiles	/362
繪製圖庫圖片	/369
地圖編輯器	/372
程式初始化	/375
繪製編輯畫面	/377
「足球番」主程式	/385
三個小時鐘	/387
遊戲狀態的初始化	/389
繪製遊戲畫面	/390
處理使用者輸入	/392

第九章 坦克大決戰

任天堂版坦克大決戰	/398
設計自己的坦克大決戰	/403
系統規劃	/404
地圖子系統	/406
角色子系統	/407
地圖子系統	/411

圖庫處理	/413
地圖處理	/418
圖庫編輯器	/426
新增及移除圖片群組	/428
圖片群組描述的永續性	/431
地圖編輯器	/434
靈活的圖片群組操作功能	/435
地圖編輯模式	/437
地圖圖層的資料設定	/437
破碎圖格的編輯能力	/439
圖層檢視選擇	/440
角色子系統	/441
TSprite 類別	/443
TTank 坦克抽象類別	/455
子彈及爆炸	/462
遊戲的誕生	/465
繪製遊戲畫面	/465
遊戲主迴圈	/470
處理使用者輸入	/477
熬呀熬出頭	/478

第十章 Fancy 軟體撰寫手則

與系統字型起舞	/486
TStatusBar::UseSystemFont 屬性	/487
可憐沒人愛的 TTreeView 及 TListView 元件	/488
TControl.DesktopFont 屬性	/489
字型的設定及維持	/490
將字型資訊轉換為字串	/491

使用 TFont 物件的永續機制	/491
處理 SDK 提供的 TLogFont 結構	/493
帶著字型走	/494
動態安裝及卸除字型	/494
藏起拖油瓶	/496
狀態列小圖示	/498
管理 TrayIcon	/500
留下 TrayIcon，其餘的都不要	/501
檔案捷徑管理	/504
COM 物件及介面	/505
ShellLink 物件及 IShellLink 介面	/505
系統資料匣的真正位置	/506
維持視窗屬性	/509
執行一份足矣	/511
尋找前一份副本	/511
傳遞參數及資訊	/516
檔案拖曳支援	/521
使用檔案拖曳支援函式	/522
取得檔案拖曳資訊	/524
DragDrop 範例程式	/525
行程的最後一刻	/526
萬無一失的善後工作	/527
寫封伊媚兒	/529
Mailto URL Scheme	/530
Mailto URL 的應用	/532

附錄

附錄 A 我的程式庫

xCONTROLS	/539
-----------	------

xDARRAY	/541
xDESKTOP	/544
xFILES	/545
xFONTS	/550
xGRAPHICS	/551
xKERNEL	/552
xMEMORY	/553
xREGISTRY	/554
xSTREAMS	/554
xSTRINGS	/555
xTIMES	/560
xUTILS	/560
xWINDOWS	/561
附錄 B 我的工具箱	
檔案分析／解譯	/563
DUMPBIN	/563
W32Dasm	/566
行程／視窗行為刺探	/568
Spy++	/568
BoundsChecker	/570
APISPY32	/571
Socket Spy/32	/573
Registry Monitor	/574
即時偵錯／除錯	/575
SoftICE	/575
DebugView	/577
資源檢視／修改	/578
Resource Workshop	/578
Microsoft Developer Studio	/579
Language Localizator	/580

系統資訊觀察	/583
OLE/COM Object Viewer	/583
Process Viewer	/584
Dependency Walker	/585
RegDump	/586
TCPView	/587
OSR Driver and Device Explorer	/588
雜項	/589
Hex Workshop	/589
Source Code Colorizer	/590
WinDiff	/591
XReplace-32	/592
Windows Help Designer	/593
附錄 C 參考書目	/595

導讀

這是一本十分特別的程式設計書籍，有技術，有趣味，通通都是要與你分享的私房菜。

這本書適合誰

C++Builder深度歷險，書如其名，它絕對不是一本入門書。

本書不教你如何使用 C++Builder 整合環境，不講解 C/C++ 程式語言，不是 Win32 基礎教學書籍，也不打算介紹 COM/OLE/ADO/.NET 等等新技術。我只想以自身的經驗告訴你，在擁有基本的程式設計能力後，如何提升為能夠自行發掘問題、解決問題、並在程式設計中尋找樂趣的更高層次。

這本書的預設讀者群為，已脫離入門階段，對 Windows SDK 有基本認識的 C++Builder 使用者。

如果你是甫進入 Windows / C++Builder 程式設計領域的新手，那麼這本書不適合你，請先築好自己的基礎，再來看這本書。

如果你已經寫了不少 C++Builder 程式，但對 Windows SDK / API 概念全無，那也好，你可由本書認清 VCL 與 SDK 的曖昧關係，並看到 VCL / SDK 雙劍合璧的各種活用。不過看過本書之後，你還是得找一本專講 SDK 的程式設計書籍來讀，本書不講基礎技術，只講求研究與活用。

我認為本書對不同程度的讀者可產生不一樣的功效：

□ 不懂 SDK 的讀者，可以由此認清 VCL 與 SDK 的關係及地位，並由各章範例得到

SDK 的基本概識。

- 稍懂 SDK 的讀者，可由此為出發點，將 VCL 撥繭抽絲，看看 VCL 裡頭的模樣，研究 VCL 為何能將 SDK 包裝得如此方便好用。
- 熟悉 SDK 但程式經驗尚淺的讀者，請看看我如何巧妙地運用 VCL 與 SDK，以漂亮精簡的形式撰寫各式各樣的應用程式。
- 熟悉 SDK 且程式經驗豐富的讀者，請將本書當作是陳寬達的 C++Builder 心得割記來閱讀。且看看，在 Win32 遊戲場內，拿著 C++Builder 及各式工具當玩具的我，如何盡情玩耍嬉戲。

全書架構

全書分為「基礎觀念」、「作業系統」、「桌面秘笈」、「遊戲快打」、「軟體開發」五大篇。

除了第一篇「基礎觀念」及第二篇「作業系統」，本書的所有主題都採由上而下的說明方式，而不是一般程式設計書籍採用的由下而上說明方式。簡單地說，這些主題都是**先有應用，有問題，有想法，想要解決它，接著才有解法，有應對的技術以及說明**，而不是說明瞭一大堆技術後，想辦法找個範例程式來驗證方才的解說。

這樣的說明方式好不好？見人見智。不過至少與電腦使用者的生活或思考貼近些，我想會是比較愉快，比較有趣的學習方式。

下列簡短介紹全書各章綱要。

第一篇 基礎觀念

第一章「RAD 無罪論」

從開發工具的選擇談起，闡明 RAD 開發工具「雖帶原罪但無罪」的特性，最後試著以現

實生活的例子來說明實作與理論的差別及辨別它們的重要性。

第二章 「VCL 基本心法」

這是最著重基礎剖根的一章，務求對所有內容瞭解透徹。

此章的重點在於釐清 RTL、VCL、Windows API 等等函式或服務的差別，深入作業系統之前，這是一定得跨越的門檻。後半部從 VCL 的內部著手，一一解說組成 VCL 的重要類別。

第二篇 作業系統

第三章 「控制你的控制臺」

此章詳細介紹控制臺元件的行為，並分別實作控制臺元件及控制臺程式來驗證這些技術及資訊。

第四章 「分秒必爭，細說計時器」

計時器是許多應用程式的必要能力，但 VCL 的 *TTimer* 元件經常不敷使用。此章由 Win32 計時器函式的各種使用方式開始，逐一介紹多媒體計時器、高解析度效能計數器，並以各種計時器函式來製作精確好用的延遲函式。最後，詳細討論計時器在多執行緒環境下的使用方式及可能遇到的問題，並提出多種解決方案。

第三篇 桌面秘笈

第五章 「一頭栽入桌面的世界」

此章先解說桌面的構成，接著一一實作桌面塗鴉、畫面截取、桌面隨意貼等桌面遊戲／工具程式。此章的重心在於奇特的 Desktop Illusionist 程式，它是全書技術難度最高的程式，使用技術包括 subclassing、hook、memory mapped file，目的在於完全掌握桌面，製造動畫桌面效果。Desktop Illusionist 使用的技術及架構可延伸使用在各類型的應用程式

上頭，值得好好研究品味。

第六章 「佈景主題工具實戰」

佈景主題套件包含各種使用者介面元件的設定值及相關檔案，從系統顏色、滑鼠指標、系統聲音...到螢幕保護程式通通都有。此章一一講解各種使用者介面元件設定值的取得及修改方式，以及如何在應用程式中使用它們等等。最後，實作一個完整的佈景主題預視/安裝工具。

第七章 「螢幕保護？我用計劃表！」

螢幕保護程式是個絕佳的創意發揮場，只要符合固定的架構，在視窗上秀出各種不同的資訊及有趣的玩意，很快地又是一個螢幕保護程式出爐。我想用每個學期初訂立的計劃表當作螢幕保護程式，因此本章由研究螢幕保護程式的架構開始，以實作一個漂漂的計劃表展示螢幕保護程式作為結尾。

第四篇 遊戲快打

第八章 「足球番」

第九章 「坦克大決戰」

面對 C++Builder 這類型的 RAD 開發工具，總有人質疑：C++Builder 到底能不能寫遊戲？

這兩章就是我的回答。我分別以 C++Builder 完整實作出足球番及坦克大決戰兩套經典遊戲，而且大部份的程式碼依賴著 VCL 類別來運作。

這兩個章節只有實作，所以我將重心移至實作的架構分析上，在實際撰寫程式前，先為遊戲進行完整的系統規劃，包括所需的子系統、類別及任務分配等等，希望能讓實作經驗不多的讀者們從其中學習程式規劃的能力。

第五篇 軟體開發

第十章 「Fancy 軟體撰寫手則」

對於軟體開發人員來說，通常以功能強大為首要標的，至於介面的設計、細節的考量就不管它了，反正有客戶服務部解決問題。此章針對軟體開發人員而寫，我試著討論一些大家容易忽略的小細節，並討論各種解決的方法，企圖使軟體在功能強大外，再掛上體貼使用者的封號。

書籍體例與用語

若你在本書看到「【】」符號，就表示是操作步驟，例如：「C++Builder 主選單【File / New Application】開始一個新的專案」。

列出原始碼時，程式列號只為方便解說程式，並不是程式碼的一部分。

程式列表除外，特別以斜體字表現的，代表該字是語言保留字、類別、函式、常數、識別字等等，例如：

<i>class</i>	這是 C++ 保留字。
<i>TButton</i>	這是 VCL 的一個類別。
<i>WM_MOUSEMOVE</i>	這是一個 Windows 訊息。
<i>CreateFile</i>	這是一個 Windows API 函式。
<i>Format</i>	這是一個 C++Builder 提供的函式。
<i>OnClick</i>	這是一個事件。
<i>clBtnFace</i>	這是一個常數。

關於譯名，我個人的翻譯原則是：「**翻得過去就要翻得回來，否則就不翻**」。

因此，對於各項技術名詞，若有直接對映的中文名詞就翻，否則就不翻。因此，component

我會翻元件，class 我會翻為類別，property 我會翻為屬性，但 form 就不翻，誰知道表格或表單指的是 Word 裡頭的 table 還是關聯式資料庫裡的 table 呢？

範例程式風格

對於書籍的範例程式，我是這樣認為的：上乘讀者只看**想法**，中乘讀者想看**作法**，下乘讀者老是想看一堆**程式碼**。

我想本書的讀者都有一定的程度，因此程式碼列的不多，能夠只列片段就列片段。畢竟沒有人想在書上仔細研讀幾百行上千行的程式碼，真的要 K 程式碼時，坐到電腦前面，將書附光碟放進去，直接在電腦上看就夠了。還可以執行、測試，玩玩看範例程式呢！

關於範例程式的撰寫風格，不可諱言的，我的程式總是大量攙或著 Windows API 及 VCL 元件的方法、屬性及事件，再加上一些 VCL 或 C++Builder 獨家提供的類別或機制，例如屬性、陣列屬性及集合等等。某些範例還會用上 C++ 標準函式庫提供的容器類別，例如 *vector*、*list* 等等。

以最基本的與設備無關的點陣圖¹為例，傳統的API寫法必須維護一個BITMAPINFO結構，一個調色盤陣列（若有的話）及一個指向影像本身的指標，還得透過幾道麻煩的API函式才能與「與設備有關的點陣圖」DDB相互轉換、儲存及載入等等。如果你跟我一樣是Windows API及VCL雙聲道的話，就可使用VCL提供的TBitmap類別，TBitmap可以裝載DIB或DDB影像，透過它的Handle屬性可以取得此bitmap的handle，透過其Canvas->Handle屬性可以取得此bitmap使用的device context handle，換言之，只要善用這兩個屬性，VCL的TBitmap類別可與API合作愉快；不只是TBitmap，絕大部分的VCL類別

¹ Device-Independent Bitmap，簡稱DIB，是與Device-Dependent Bitmap（DDB）對比而言。「與設備無關」一詞是指，不論你使用的顯示模式為何，此bitmap皆以它原來的格式儲存，但DDB就會因使用的顯示模式不同而有不同的儲存及表現方式。

也都可以經由類似的方法與API結合使用。

因此，若你在程式中看到下列的用法時，請別緊張，這不過是 VCL 及 API 的混合體罷了：

```
// 將工作列上的圖示隱藏起來
ShowWindow(Application->Handle, SW_HIDE);

// 讓 Canvas 使用 Image1 的調色盤
SelectObject(Canvas->Handle, Image1->Picture->Bitmap->Palette);

// 經由 lf 結構建立出 HFONT，再設定給 Font 物件使用
Font->Handle = CreateFontIndirect(&lf);
```

VCL 是由 Object Pascal 撰寫而成，因此不可避免地，當我要列舉 VCL 的某些片段佐證或說明時，必須直接引用 Object Pascal 程式碼。這是 C++Builder 先天上的毛病：它無法脫離 Object Pascal 而存在；這也是 C++Builder 程式員的宿命：若要徹底地瞭解、掌握 VCL，你還是先得熟悉 Object Pascal 語法。

幸好 Object Pascal 語法與 C++ 十分類似，除了某些獨有的語言設施，Object Pascal 幾乎可說是 C++ 的子集，相信只要拋開心理障礙，你也能輕易地讀懂 Object Pascal 程式碼。

光碟內容

光碟內有下列檔案：

- 所有範例程式原始碼及可執行檔，我為 C++Builder 5 及 C++Builder 6 使用者分別建構一份，置於 BCB5 及 BCB6 目錄。
- 完整的 Delphi 深度歷險網站（2002.01.26）

介紹給你

有兩個網站，如果你還未曾聽聞，我一定要介紹給你知道。相信我，如果未來你將持續

在 Delphi / C++Builder 領域中奮鬥，這兩個園地一定可以助你一臂之力。

□ Delphi 深度歷險 <http://www.vclxx.org/>

全球最大的中文 VCL 元件整理站臺。

□ Programmer 深度論壇 <http://forum.vclxx.org/>

最有深度的 Borland Delphi / Kylix / C++Builder / Java / OOSE 討論社群。

與作者連繫

這是一本紙張裝訂而成的書，很遺憾我不能在上頭放藍色斜體的 URL，讓你用手指雙擊來開啓郵件程式寫信給我。

我很希望知道你對此書的看法，對書籍內容的建議，或者文字上或技術上的指正。另外，只要你的目的不是要我為你解決專案遇到的技術瓶頸，我很樂意與你討論各式問題。

若你要與我連絡 —

□ EMail Address : ktchen@iis.sinica.edu.tw

□ Web Page : <http://www.iis.sinica.edu.tw/~ktchen>

第一篇

基礎觀念



第一章

RAD 無罪論

易使初學者陷入迷途是 RAD 的原罪，
但 RAD 所帶來的好處豈容輕易抵消。我想，RAD 無罪，
它只是使門檻降低點，讓程式設計更為簡單輕鬆，何罪之有？



對於剛接觸 Windows 程式設計的新手而言，要在各家說法紛云、眾多開發工具環伺的情況下，選擇一個明智有把握又最適合自己的開發工具，可說是最難的一道入關匣門。不只是新手，就連許多老將也容易執著於同一套開發工具，寧可使用鍾愛的工具埋頭苦幹，無視於身旁更方便好用的解決方案，即使可以省下三倍的工作時間。

的確，進入新的程式設計領域前，開發工具的選擇是最重要的一項課題。選擇正確，讓你天天有時間快活，可以陪陪女友看看電視逗逗小狗；萬一不幸選到難學難用或者根本不適合專案性質的開發工具，那麼，帶給你的可能是在每一個漫漫長夜裡，面對電腦獨自摸索嘗試的命運。

這種選擇是十分無奈難為的。如同年紀輕、閱歷淺、視野小、知識窄的年輕人非得在剛放下歷史課本、偉人傳記的青年時期做出影響一生路途的抉擇，做好人生的規劃；甫接觸一行行冷澀生硬程式碼的入門者，也勢必在概念模糊、謠言四起、真偽難分的情況下，面臨選擇程式語言及開發工具的難題。

不得不為的選擇

對於「我是 Windows 程式設計的初學者，該選擇什麼開發工具好呢？」這類經常見到也會永遠存在的問題，我曾在網路上見過這樣的回答：「Visual C++ 最多人用，所以推薦給你」、「VB 好學，所以選它一定沒錯」、「C++Builder 是 RAD 工具，簡單易學，我學三天就會寫了」等等十分沒有概念，說是不負責任也不為過的言論。

不想流於空談，還是來談談我自己的看法好了。就像人類戰士通常雙手握著巨劍，侏儒通常肩著戰斧，而魔法師無可選擇地手執魔杖一樣，選擇一把稱手武器還是得視個人的情況及需求來評估衡量。

狂熱份子的信仰

現在有許多電腦玩家常不由自主地對自己所鍾情的作業系統、開發工具、應用軟體甚至遊戲及軟體公司產生幾近宗教崇拜式不明究理的狂熱。在此情況下，很容易去找到一個貌似敵對的「對手」來反，以堅定自己的信仰、壯大自己的聲勢，例如 PC vs MAC、FreeBSD/Linux vs Microsoft Windows、Visual C++ vs C++Builder、Delphi vs VB 等等。

在情況越益嚴重無法遏抑其勢的今日，只能期待點醒一些狂熱份子，擁 X 反 Y 並沒有什麼不對，但是：

一個人若是爲有了宗教信仰而驕傲、自滿，甚至因此鄙薄無信仰的人，或動輒排斥與他信仰稍稍不同的人，便表示他自己還沒有找到信仰，所以，他也在他自己鄙薄和排斥之列。

<<疑神>> 楊牧

TANet 一位大哥級的人物也曾語重心長地說：「科技的出現應該是要爲人來服務的，你盡可以擁抱 X 痛罵 Y，不過切記，知己知彼百戰百勝」。這是看到網路上一堆連保護模式、檔案系統、系統軟體、排程問題、虛擬記憶體都不知何物的網友痛罵 Windows 作業系統，連 API、DLL、物件導向觀念、application framework 都摸不清脈絡的新手卻大聲疾呼推行或反對某某程式語言或開發工具的怪現狀所發的無奈之語吧！吾亦有同感。

學習動機

就我自身的經驗而言，程式設計初學者的學習動機大致可分爲下列數點：

不爲什麼，就是想學

有趣的是，這類型初學者佔用的比例還不小。也許是羨慕程式高手對於電腦系統的瞭若

指掌，也許是希望也能擁有創作軟體的能力，也許就是單純地覺得撰寫程式碼是件神秘有趣的事，目的說不上來，沒有很強烈的學習欲望。

這類型的初學者由於動機不足，通常只能維持三分鐘熱度，往往「XX 語言入門手冊」看了一整年還停留在第二章...若你屬於這個類型，我的建議是：「順著感覺走」。程式設計只是程式語言的撰寫，與電腦聊天，請它為你做事的溝通方式罷了。對於一個不打電視遊樂器，也從來不想到日本去遊玩或留學，工作上也沒有日文需求的人來說，日文的聽說讀寫能力不具太大意義。那麼，如果你沒有事情想直接跟電腦溝通，學習程式語言又有何益？

如果沒有具體的需求，最好現在就訂出一個來。有了目標的存在，才能確認正確的學習方向，千萬不要無所求式地盲目學習，那只是白費力氣罷了，看電視綜藝節目都還比這值得。

專案需求

由於專案的需求，各行各業有無程式設計基礎的人們都得進入 Windows 程式設計領域，這是十分無奈但最常見的情形。

也許是專業分工的認知還不足。若要面對外國客戶時，我們不會要求某個秘書開始三個月前去學該國語言，而會花高薪請一位專業翻譯人員幫忙；但是對於程式設計，常常會要求自己的員工，無論是什麼背景，丟到教育訓練中心學個一兩個星期，回來後就期望他們能夠將需要一年半載開發時程的軟體專案順利做好。

當然，口語的學習及程式語言的學習在時間上還是有差別的一口語比程式語言靈活太多了。不過，口語學到一半的人們說起話來結結巴巴，半天湊不出字來，我們聽來十分清楚；程式語言學到一半的人們，寫起程式來缺三漏四，往往連邏輯觀念都還沒搞清楚，就得從上命開始撰寫成千上萬行規模的程式，他們的窘困，旁人看不見。

我心目中的理想狀態是，有著足夠數量足夠專業素養的電腦程式設計人員，供各行各業大小公司短期或長期聘雇，與兼具電腦素養及 domain knowledge 的人員溝通合作，解決各領域的電腦軟體需求。

讓「人能盡其才」是最重要的，我已看過許多物理、光電、材料、數學領域的同學及朋友，爲了實驗室或計劃的需求，得暫時拋下自己的專業，去學習程式設計來控制 RS232 連接埠啦，撰寫介面卡驅動程式啦等等，或爲物件導向概念昏頭轉向，或是被平臺相關的煩瑣細節整得心煩意亂、無計可施。一來短期出家所製成的軟體品質堪慮，二來浪費太多能夠增強專業能力的時間。

如果因爲專案的需求而進入程式設計領域，一般來說，程式語言及開發工具的選擇上不會有太大的彈性，通常必須配合上司的好惡或團隊原本的解決方案來斟酌考量。

追求理想

許多人當初都是爲了一個偉大崇高的理想才進入程式設計領域的。然而就我所聽聞，遊戲程式設計似乎經常扮演這個「偉大崇高理想」的角色。

其實，筆者我就是活生生血淋淋的一個例子。高中時代，迷上創世紀系列遊戲的我，成天幻想著撰寫一套很棒的 RPG 遊戲，而這個夢想就成爲驅使我自學 C 語言及組合語言的強烈動機。一腳踏入程式設計領域後，無法自拔的我很快地全身淪陷，接著就是你們現在所見的這個狀況了....p

嗯，既然提到我的興趣，順便來談談遊戲程式設計初學者的選擇吧。

遊戲程式設計是程式設計各領域中，狂熱及理想成分比例超高的一群，有許多各種性質的程式設計師，當初都是因爲熱衷遊戲設計，而就這樣持續掉入程式設計的漩渦呢。學習動機是毫無疑問地：「想要具備撰寫遊戲的程式功力」。不過，依遊戲類型的不同，日後學習的方向與重心也迥異，例如 2D RPG，重點在於畫面設計及故事劇情，外加

DirectDraw 提供的快速捲軸能力；3D 動作遊戲，Direct3D 或 OpenGL 是跑不掉的，對於圖學的理論，演算法及應用面，也最好花心思時間去努力研究學習；再如多人連線棋類遊戲，DirectDraw、Direct3D、DirectInput 我想都用不太上，好好研究提供連線功能的 DirectPlay 或發展一套穩固的即時資料傳輸程式庫才是重點。

目前基礎

現有的基礎是決定工具及語言上手度的最重要因素。

許多人的高中計概課程教的是 Basic，通常以 Quick Basic 作為開發工具，上大學後接著學習 VB，銜接得恰恰好。大多數大專院校的計概課程教的是 C 或 C++ 語言，也有些資訊相關科系的計概課程以 Pascal 語言做為教學內容。熟悉語言之後，他們正好可以選擇 C++Builder、Visual C++ 或 Delphi 做為 Windows 環境下的程式開發利器。

除了 Basic 以外（原因下述），我很贊成就配合你目前的所學，擁有 Pascal 語言基礎就選擇 Delphi；愛用 C/C++ 語言就選用 C++Builder 或者 Visual C++。

個人偏好

Basic、C/C++、Object Pascal 這三個程式語言，雄霸著整個 Win32 程式設計領域。Basic 易學易用，Pascal 嚴謹明確，C++ 強大複雜，各有各的擁護者及理由。

易學易用的 Basic

在 Win32 環境下，Microsoft Visual Basic 是使用 Basic 語言最著名的開發工具。

Basic 語言十分簡單易懂，微軟希望 VB 及 VBA 維持著簡單到任何想依賴電腦進行自動化程序的電腦用戶都可以輕易地上手的程度。因此雖然附加的功能不斷上疊，語言本身

維持著 Basic 的原有特性。

Talk

西元 1964 年，BASIC 語言（全名為 **B**eginners' **A**ll-Purpose **I**nstruction **C**ode）誕生，由 Dartmouth 大學的 Thomas Kurtz 及 John Kemeny 兩位教授共同發展，他們希望 BASIC 語言能作為學生在學習功能比較強大的程式語言（如 FORTRAN、ALGOL 等等）前的踏腳石。

Basic 易學、易用的特性是眾所皆知的，同時它還兼具容易轉譯為機械碼的特性，最著名的故事就是當年 Bill Gates 在毫無測試除錯機會的情況下，撰寫出一套程式碼加資料不到 4K 記憶體的基本直譯器。

第一個 Basic 編譯器是 Quick Basic（在這之前，Basic 程式都經由直譯器來執行），由 Microsoft 發展。Quick Basic 發展至 4.5 版後，Microsoft 另外發展 Quick Basic Extended，也稱為 PDS Basic（Personal Development System），PDS Basic 發展至 7.1 版告停，接著就是大家所熟知的 Visual Basic 了。

必須承認的是，Basic 並不適用於中大型應用程式的開發。

它的先天條件不良，例如執行速度緩慢、未提供完整的物件導向程式設計機制等等，再加上後天失調，例如硬加入部分的物件導向機制（只有封裝）、其它規格的修改配合（例如 COM 的 *IDispatch* 介面），使得 VB 已成為微軟揠苗助長下的犧牲品。即使有微軟如此強而有力的老大哥極力護盤，先天缺陷仍舊無法根除，除了易學外，實在找不出其它應該使用 VB 的理由，因此我會建議所有的初學者，若能夠接受其它的語言，轉移陣地為上策。

Talk

VB 支援 COM，而 COM 主張的是 **interface inheritance**，不是 **implementation inheritance**，所以也算具有繼承及多型的觀念。

嚴謹明確的 Pascal

在 Win32 環境下，Borland Delphi 是使用 Pascal 語言最著名的開發工具。

Niklaus Wirth (1984, ACM Turing Award 得主) 於西元 1968 年發明 Pascal 時，他主要的目的是想發展出一套能兼顧發展及執行效率，而且具有高度結構性及組織的程式語言。另外，他也希望 Pascal 能夠適合教學目的，讓學生易於理解電腦程式設計的概念。

Pascal 之名是由偉大的數學家 Blaise Pascal 而來。Niklaus Wirth 曾參與 ALGOL 60 語言的發展計劃，而 Pascal 就是直接從 ALGOL 60 改良衍生的程式語言，後來它又陸續從 ALGOL 68 及 ALGOL-W 吸收許多語言上的優點。

Talk

在 Pascal 之後，Niklaus Wirth 還發明了 Modula-2 語言，另外還有 Oberon 語言，從 Modula-2 語言改良而來。

有趣的是，Niklaus Wirth 一直想設計飛機，只是他發現他需要更好的工具，於是他才設計了一個個的程式語言，並造了自己的電腦 Lilith (Module-2 語言就是為 Lilith 而設計的)；這與 Donald Knuth 為了 *The Art of Computer Programming* 驚世鉅作的排版，特別花十年的工夫發展 T_EX (極著名的排版語言，尤其適合數學及科學方面的文件排版) 有異曲同工之妙，兩位大宗師真不愧為大宗師。：)

在今天看來，Pascal 仍然保留著當年 Niklaus Wirth 所規劃的發展原則—嚴謹明確的特性。由於 Borland 對 Pascal 語言的全盤掌握，使 Pascal 語言能夠順利演化為真正的物件導向程式語言—Object Pascal。就像 FreeBSD 的 coreteam 全盤控制所有 FreeBSD 套件的更新撰寫一般，Pascal 控制權控制在 Borland 一小撮人手中，雖失去開放性，但維持著該有的堅持及清新。

我認為它的物向導向支援恰得其所，該支援的全都支援了，不必要的也不貪多，適當地

加入。它與 C++ 的優劣是沒有答案、見人見智的，正如同大禮服及小洋裝，好不好看，適不適合，因人而異。

強大複雜的 C++

在 Win32 環境下，使用 C++ 語言的開發工具為數不少，最有名的就屬 Microsoft Visual C++、Borland C++Builder 及 Symantec C++。

Talk

西元 1980 年，Bjarne Stroustrup 發明了“C with Classes”語言，因為他想為 C 語言加入如同 Simula67 語言般的事件驅動機制。直到 1983 年夏天，才由 Rick Mascitti 創造出 C++ 語言名稱。

如果你懂得 C 語言，那麼可以這樣看待 C++ 語言：

- 它是比較好的 C 語言
- 它提供資料抽象化機制
- 它支援物件導向程式設計，所提供的語法及字眼讓你可以很輕易地將物件導向精神落實到 C++ 程式中

C++ 語言的功能強大，無庸致疑，template、exception handling、RTTI、Standard Library 等等功能不斷地加強翻新。由於使用者眾，要求必多，期望必高，再加上 C++ 本身定位在功能強大範圍廣泛的通用性語言，如江海之納百川，C++ 自然日益複雜。著名的雜誌 C++ Journal 上曾有段話讓我印象頗深：

如果你認為 C++ 還不算太複雜，那麼請你解釋何謂 protected abstract virtual base pure virtual private destructor，而你又會在何時需要它呢？

Tom Cargill C++ Journal Fall 1990

雖然是最流行的物件導向程式語言，但除非你有足夠的耐心及精神來全盤掌握它，否則輕易嘗試的後果可能只會得到一臉的挫折。當然囉，十分的複雜也帶來十分的便利及高

度的成就感及樂趣，我有一位朋友，工作上使用其它語言，但將 C++ 當作興趣來研究把玩，真是酷斃了。

RAD 的原罪

每過一段時間，TANet 網路論壇上就會興起一陣 Visual C++ 與 C++Builder 孰優孰劣的討論。前些時日，討論周期又到了，果然又是口誅筆伐的一場論戰。樂於當潛水艇欣賞文章的我，捕捉到一些頗為中肯的言論：

發信人: Meou@m2.dj.net.tw (Dadai)

這兩個東西都蠻好用的。但是現在我摸了幾個月之後，我反而比較喜歡 VC++。VC++ 的使用者介面看來繁瑣，但是真的用心花個幾天把 VC++ 的功能摸熟，用起來還蠻順手的；再加上把 VC++ 的 Application Wizard 的來龍去脈搞清楚，把 Class Wizard 的用法弄懂，MFC 一點一點慢慢弄熟之後，它的功能還蠻強大的。再加上 VC++ 的 HTML Help 比 BCB 好上百倍，我現在覺得 VC++ 比較好用。

這兩個工具，一個是倚天劍，一個是屠龍刀，放在不會用的人，那一把都不順手。這兩個工具都需要相當好的 C++ 基礎。好的 C++ 基礎對 MFC、OWL、VCL 來說都是基本功夫而已。程式學了一陣子，覺得 MFC 摸熟之後，建構介面不比 C++Builder 來得慢，**重點在於介面之下你到底要如何解決問題**。這個問題遠比 Application Framework 及那一套開發工具比較好來得重要多了。

發信人: dyliu@ms1.hinet.net (四眼的王蟲)

VC++ 的線上說明系統的確比 Borland 的 Delphi、BCB 等好太多了，Borland 在這一方面一直沒有甚麼進步。有時候我用 Delphi 或 BCB 時，還會執行 VC++，目的就是要用 VC++ 的線上說明系統，Borland 實在應該在這一方面大力改進才是。

介面方面 MFC 實在是比不上 VCL，簡單的介面還好，複雜一點的 MFC 就得搞上老半天。

發信人: oesd@email.gcn.net.tw (Dadai)

VC++/MFC 的學習曲線看起來比較長，但是值得。因為你如果搞懂的話，再配合上一些 SDK 的知識及合適的開發工具，你幾乎可以在 Windows 下做到任何你想做的事（剩下的只是你怎樣解決你要解決的問題）。

我自己覺得 MFC 要用的有點基礎，最起碼你要能把 VC++ Wizard 能做的東西知道如何全部用手工打造出來，不然用 Application Wizard 建出來的東西，你一樣看不懂，更不要講如何去改它了。

其實像 BCB 這種 RAD 開發工具要學得好，我覺得比 MFC 還難，因為在那漂亮介面下的底層機制往往比 MFC 複雜許多。

真的大聲喊 BCB 好簡單的，我只看到兩種人。一種是在 Win32 SDK 裡面打滾多年，幾千條 Win32 API 就算沒用過也都大概摸過，沒摸過也知道大概會叫什麼名字，該往那邊找。問他一個問題，腦袋裡面會自動列出一堆 Win32 API，一條條過濾該如何解決。寫 Windows 程式可能打字到手指頭都長肌腱炎了。BCB 對他們是種解脫，VCL 更是不成問題。

而另一種則是完完全全的初學者，拿 BCB 來學 Windows 程式設計。最多只能學到元件有提供的功能都會用，元件不會的他也不會，元件不行的他也不行，元件的限制就是他的限制。應用程式寫到一半，裡頭要呼叫 Win32 API 函式，他大概就掛了，要做到現有元件做不到的功能？那你要不要花錢買 VCL 程式庫？

另外，再聽聽 Visual C++ 知名作家侯捷在他的「懷璧其罪 RAD」一文中怎麼說：

RAD 並非罪惡，而是優點。要怎麼用它則是 developer 自己的問題。

侯捷對於 RAD 工具如 Visual Basic 或 Delphi 或 C++Builder 並不擅長，但我知道 Visual Basic 可以呼叫 Windows API，做相對低階的動作；當然，以軟體工程角度來看，VB 是比較弱，因為它不具 OO 特性。至於 Delphi，我有兩位這方面的專家朋友（錢達智與陳寬達），他們可以使用 Delphi 做任何事情，沒有任何你想像中 RAD「該有」的限制；C++Builder 和 Delphi 系出同門，一樣沒有什麼限制。

以下引一段我在【汗如雨下·雜感·1998/11】的文章片段：

● 關於 RAD (Rapid Application Development)

<Delphi 學習筆記> 作者錢達智先生，是我的好友。我們之間對於 RAD 有段討論，或許你想聽。

■ 侯：BBS/News 上時有關於 RAD 的工具見解。我認為你很有資格說些話。不少人對 RAD 有誤解。如果你能點醒大家：

1. RAD 是很好的開發工具
2. 使用 RAD 並不代表不需要底層紮實的基礎，那麼誤解的人就會比較少一點，知道該怎麼做的人就會比較多一點。

■ 錢：過去在 DelphiChat、News Group 以及我的書中，這樣的想法都不只一次宣揚過。

其實，就我接觸過的人，不論是網路或者是學生，RAD 的使用者的確是比較急功近利，也難怪會有這樣的刻板印象。：（

雖然說過，但還是要持續宣揚這種理念，就如大哥說的，誤解的人會少一點，知道該怎麼做的人就會比較多一點。

RAD 真是「匹夫無罪，懷璧其罪」呀。

Visual C++ 及 C++Builder，一個採 MFC 一個用 VCL，一個必須配合 Wizards 撰寫大量程式碼，一個雖然可用滑鼠完成大部分的介面設計，不過程式邏輯及核心還是得撰寫程式碼，此時 RAD 派不上用場。不論如何，他們提到了幾個重點：

1. 一個是倚天劍，一個是屠龍刀，兩者皆是絕世神兵，但交給不會用的人，那一把都不順手。
2. 重點在於介面之下**你要如何解決問題**。
3. RAD 開發工具要學得好，不比 non-RAD 開發工具簡單，在漂亮介面下的底層機制往往出人意料地複雜。
4. RAD 及 non-RAD 開發工具，擁有相同的「願望達成能力」，功夫底子好的人要起來，樣樣都能實現。

所以，重點在人身上。不管是 RAD 或 non-RAD 開發工具，用得嚇嚇叫的那些人永遠可以做出他們想要的成果；不管是 RAD 或 non-RAD 開發工具，學得不夠透徹的人們永遠也只能抱怨程式語言太難、開發工具太爛，無法享受程式設計的樂趣。

換個角度來看，同樣是 RAD 開發工具的使用者，有的是全然解脫，輕巧駕御開發工具，善用 RAD 的特性來提升程式開發速度及品質；有些卻只能拉拉元件，能力侷限於別人製作的元件功能，因為跨不出開發工具的格局，完全被 RAD 的服務範圍限制綁死。

Talk

也許有人抱持著相同的懷疑：Delphi / C++Builder 到底能做些什麼？

我必須再一次大聲呼喊：「在 Win32 下，除了驅動程式撰寫¹以外，只要是其它開發工具辦得到的，Delphi / C++Builder 就辦得到！」。

就我所見到的，埋怨開發工具能力不足的人，通常同時在揭露著自身能力的不足。而真正見到開發工具能力不足或設計不良的那些人，不會大落落地成天埋怨責怪，他們總有辦法另找出路來解決問題。

¹ 其實，若搭配如 WinDriver 的發展套件，Delphi / C++Builder 也可以拿來撰寫驅動程式。

開發工具的差異

比較 Visual C++、C++Builder 及 Delphi 這三套開發工具，我將它們之間重要的差異列成下表：

表 1-1 / Visual C++、C++Builder 及 Delphi 三套工具的比較

比較項目	Visual C++	C++Builder	Delphi
設計公司	Microsoft		Borland
前端語言	C++	C++	Object Pascal
Application Framework	MFC		VCL
介面設計方式	傳統 (Class Wizard及手工打造) ²		RAD (拖拉點按)
程式核心	手工打造 (有許多程式庫及類別可供運用)		手工打造 (有許多元件、程式庫及類別可供運用)
運作原理	呼叫 Windows API		呼叫 Windows API

其實，不論什麼程式語言，不論什麼開發工具，只要在同一個作業系統內，它們的運作原理都是一樣的：呼叫作業系統提供的服務（通常以函式呼叫的方式）。在 Win32 環境下，我們稱這些系統服務為 Windows API。

Win32 開發工具的演進

原本，Win32 程式設計師撰寫程式的唯一方式就是呼叫 Win32 API 函式，這些 Win32 API 函式由 KERNEL32.DLL、USER32.DLL 和 GDI32.DLL 三大模組及其它大小 DLL 所提供。這些 API 函式分為許多種類，各擅其職，只要好好善用它們，在作業系統提供的

² 有些廠商發展出極像 VB 設計介面的 VC++ Add-On，雖然用的是自己的 class library，不過至少也讓 Visual C++ 朝 RAD 工具邁進了一步。

能力範圍內，沒有做不到的事。不過，API 函式既多且繁雜，而且如同 RISC CPU 提供的指令集，每一道函式所做的事情並不多，連一些頻繁使用的例行公事，例如建立新視窗、註冊視窗類別、更改按鈕顏色等等動作，還得花上十幾行程式碼來做，麻煩透了。

需求乃創造之本，於是程式庫出現了；挾著物件導向的浪潮，緊接著類別程式庫也出現了。類別程式庫慢慢發展，功能不斷加入，規模越見龐大，負責的範圍也逐漸涉及應用程式的生滅及運作核心，最後終於蛻變為更高層級的 application framework，最負盛名的兩套 application framework 就是 Microsoft 的 MFC 及 Borland 的 OWL。雖然有類別程式庫及精靈、專家等工具的輔助，仍有人不知足地想要發展能夠更快地開發應用程式的方法，於是就有了 Visual Basic 這類可靠滑鼠完成大部分介面設計工作的 RAD 開發工具，最後才是 Delphi 及 C++Builder。

RAD 無罪，輕鬆有理

隨著時間的腳步，人們總要適應大環境的變遷及進化，RAD 的確為程式開發員省下不少介面開發的時間。但相對地來說，因為它大為降低程式設計的門檻，使得太多的初學者沈溺於 RAD 元件的強大及使用，不知道 application framework 及 Win32 API 的地位，無論真正解決問題的資料結構及演算法，甚至連程式語法都不太熟悉的狀況下就可硬湊出亮麗端莊的程式外觀！我想這就是 RAD 開發工具開始受到部分人們的質疑之故。

看透傳統開發工具及 RAD 開發工具，抹穿 MFC 及 VCL 這兩套 application framework，它們只是包裝一薄一厚，用法各異罷了。

MFC 薄薄的一片，讓你擁有全盤掌握的滿足，相對地，學習曲線既陡峭且高峻，需有足夠的背景知識才能充份融入 MFC，享受它的好處。VCL 的包裝並不徹底，但厚厚地這一層，讓人完全看不到骨子裏的究竟，如同寒流一來，一個身穿五六件襯衫外加夾克兩條的女人打從你眼前走過，天知道究竟是蛇腰豐臀亦或瘦骨嶙峋。就介面打造來說，VCL 包裝的真是好用方便，不過 VCL 常有力有未逮，包裝不足時，此刻，是 RAD 也好，不是 RAD 也好，任何工具幫不上忙，只有瞧自己琢磨 Win32 API 的功夫。若沒有三兩下

子，馬腳隨著 framework 的不足就立即露出了。

讓我想到幾年前曾發生的「命令列優劣之爭」。有些高手們及 UNIX fans 喜愛命令列，一來速度快，二來有全盤掌握的感覺；而另一派當然是傾向圖形使用介面囉。傳統確實有傳統獨到之處，否則為什麼在電子音響大行其道的今日，仍有玩家願意傾幾十倍的價格，把玩真空管音響呢（我老爸就是一個）？命令列用得熟，操作速度比圖形使用介面還來得快上幾倍倒是真的，我覺得這是命令列需要存在的最大理由。有人對我說，「命令列較讓人懂得電腦真正的運作原理」，我告訴他，「為什麼圖形使用介面就會妨礙到我們去瞭解電腦的運作原理呢？」守舊也得要合理的理由，否則只是戀舊，潛意識裏的觀感其實是怕跨入新的領域而失去一向保有的優勢。

所以呢，手工打造的好，還是拖拉點放的方便？手工打造的快，還是拖拉點放的省事？我想答案是很明顯的，寧可在例行公事上能省時間就多省點時間精神，我們還有未來等著去創造呢，焉可鎮日沈迷在手工打造全盤掌握之感覺中呢？易使初學者陷入迷途是 RAD 的原罪，但 RAD 所帶來的好處豈容輕易抵消。我想，RAD 無罪，它只是使門檻降低點，讓程式設計更為簡單輕鬆，何罪之有？

實作與理論

實作、理論往往罄竹難分，讓人搞不清楚究竟什麼是「淺嘗即可的實作」、什麼是「實作架構」，什麼是「實作理論」，以及什麼是「真正的理論基礎」。糟糕的是，對於沒有攙雜任何程式碼的大量文字，許多人往往以為這就是所謂的「理論」。

參與者的類型

常跟朋友聊起，面對開發工具及程式語言的選擇，約略可將所有的參與者分為三大類：

新手型

對所有的開發工具程式語言甚至開發平臺全然陌生，大略聽過一些開發工具的名稱，經常弄錯也是常有的事，例如“Virtual C++”、“Virtual Basic”、“Dephli”、“Borland C Builder”等等。這個族群所佔的比例最高，往往在網路論壇上詢問「該學什麼程式語言？」、「我想寫遊戲，該用什麼語言？」、「XXX 及 YYY 究竟什麼比較好？」這類問題，常是引發程式語言及開發工具優劣論辯大戰的導火線，雖然是懵懂無辜的。

程式設計對他們而言是未知的領域，充滿好奇、期待但找不著進入的門口。

專家型

所謂專家，即是訓練有素的...呃，技術實力高人一等的...呃，專才。他們通常精通某一廠商的開發工具，獨鍾一派程式語言，擅於撰寫特定領域的程式，對於相關的函式庫、application framework 及實作細節捉摸得十分清楚，熟悉得不得了。

但是，獨鍾特定工具、語言甚至實作細節的結果，並不代表擁有深厚的資訊科學基礎及寬廣的資訊視野。長期緊追產業技術或侷限某開發工具的結果，容易導致「**技術即是知識，實作即是能力，實作架構即是理論**」的錯誤觀點。

缺乏足夠的背景知識，故步於狹窄的領域，這類型的玩家常是網路論壇上專屬某某特定語言或開發工具的超級打手。

駭客型

理論面上，資料結構、演算法、編譯器、作業系統、計算理論等等，是必備基礎；實作面上，他們往往至少熟悉兩種以上的開發工具及程式語言，並將火力集中在與語言無關的方法論。

對他們而言，若要開發主從式資料庫專案，拿出 Delphi 拉拉資料庫元件；若要撰寫 Web 伺服器端程式，以 C 語言來撰寫 apache module；有跨平臺的需求時，祭出 JDK 或 Symantec Café、Borland JBuilder 來撰寫 Java 程式；專案用到 VxD、WDM 或 kernel mode driver 時，

捲起袖子拿出 SDK、DDK 加上 Visual C++，再買套 VToolsD、Driver::Work 或 WinDriver 立即動工。無所為無所不為，不執著於任何開發工具及語言，自然不會被任何公司的規劃（如 Microsoft 的 VBA 吃遍天下）或美好遠景（如 Microsoft 的 DNA 架構、Inprise 的 Information Network）等解決方案所羈絆，而隨波逐流了。

擁有足夠的背景知識，所有的語言及軟體終究只是工具，能夠以超然的態度來面對、比較，這類型的玩家通常把工具當玩具，把寫程式當成散文寫作看待。

參與者的落腳處

見過許多程式設計的初學者，老愛劈頭就問：「是否有快速的入門方法？」，我總不覺莞爾。為什麼小時候學々々口，長大學物理、化學、數學、經濟學時，從來不會奢求所謂的「快速入門法」，一旦面對程式設計的學習關卡時，就想走捷徑了呢？這是十分有趣的現象。

我想，**態度與角度**是決定各個參與者的落腳處。**態度指學習這些知識技術的態度，角度指看待程式語言及開發工具的角度。**

不論你身為哪種類型的程式設計領域參與者，請試著站在駭客型玩家的角度來觀察（事實上，只要努力的方向正確，你遲早也屬於他們）。對無入而不自得的駭客們而言，基礎知識及能力既足，各種程式語言只是與電腦對話的各種方言，發音撰寫各有不同而已；而各個開發工具只是各家廠商提供的不同翻譯美眉，環肥燕瘦各有優缺而已。以這樣的角度來看待程式語言及開發工具，你覺得還必須擁 XXX 反 YYY，或者執著各項實作細節嗎？

面對任何一個程式語言及開發工具時，應該明瞭它們只是幫助你撰寫程式，用來控制電腦達成目的的幫手，絕非學習的目的。很多人誤會了這點，將開發工具視為學習目的，往往花了數月辛勤不倦地熟悉開發工具的每一吋細節以及程式語言的所有語法後，才發現程式設計原來不是將一堆符合程式語法的語言元素堆砌起來就成。別因為程式語言及開發工具的複雜或新穎就慌了手腳：**程式語言是工具，程式的架構、邏輯及設計是手段，**

而程式執行的結果才是我們的目的。切莫將程式語言或開發工具學過了頭，忘了程式設計真正的目的。

這些技術是什麼？

舉例來說，RFC 是載明 Internet 上眾多協定、規格及架構的文件。雖然完全都是文字，只有極少數包含資料結構的定義，看起來像是很值得仔細研究閱讀的教學文件。但其實它是實作規格，只有實作者才需要嚴謹地閱讀它，搞清楚所有的細節。雖然其中的規範及協定煞有其事，但這仍是實作的一部分，如同公司進行專案前所撰寫的程式企劃。不同的是，RFC 是國際性的、免錢的、標準的規格書。

往下一層來看，支撐整個網際網路的 TCP/IP 協定群，如 ARP、IP、ICMP、TCP、UDP 等等，坊間有太多的書籍為它們撰寫專書，說明這些通訊協定的細節及通訊流程，它們是什麼？實作規格。

撰寫主從式或多層式架構資料庫程式時，涉及的主題可能包括 VCL、BDE、SQL、ODBC、ADO、COM、DCOM、CORBA、MTS、MIDAS...等等；撰寫網際網路應用程式時，可能涉及 TCP/UDP、POP3、SMTP、FTP、TELNET、BSD socket、WinSock、WOSA...等等；撰寫 Web 應用程式時，可能涉及 HTTP、SSL、SET、HTML、DTD、XML、CGI、NSAPI、ISAPI、ASP、Java Applet、Java Script、Java Servlet、VB Script、ActiveX、Netscape Plug-in、PHP、Perl、mSQL...等等。

以上列出的一堆技術名詞，有些是程式語言、有些是程式語言的應用、有些是文件語言、有些是文件語言的語法規則、有些是實作架構、有些是作業系統提供的機制或服務、有些是軟體、有些是軟體提供的機制及服務、有些是通訊協定、有些是分散式物件模型...等等。不論它們是什麼，請記得，**它們都是實作，或是因實作而來的規劃或架構，或是由實作衍生的副產品。**

通通都在裡頭

難道我就一直貶抑這些許多人抱在懷裡的各項技術及知識嗎？當然不。

它們都是實作，可是理論都在裡頭。

研讀 TCP 通訊協定，非常好。可是你研讀的只是 TCP/IP 各協定的封包格式、協定規格、傳遞流程呢？還是試圖去瞭解背後支持它，維護效率的 Nagle Algorithm、Slow Start Mechanism、Congestion Avoidance Algorithm、Fast Retransmit and Fast Recovery Algorithm 呢？認識這些傢伙，才能理解所謂「可信賴的傳輸層協定」，才能將對 TCP 的瞭解推廣到其它通訊技術上。

深入作業系統的內部架構及底層實作，非常好。這可使你徹底瞭解作業系統的實作方式，暫時拋開紙上談兵的作業系統教科書，詳細瞧瞧在現實世界中，架構在現代硬體上的現代作業系統，究竟如何妥善地擔任電腦硬體與應用軟體之間的橋樑。以 Windows 95 為例，你研究的是它與 DOS 的親密關係、INT 21h 的使用時機、噁心的 thinking 機制呢？還是研究它如何提供 LPC、IPC³，thread scheduling / page replacement 機制，如何處理 memory thrashing / page thrashing 現象？

你看到了哪些？

列出以上這些「實作」、「理論」模擬兩可的說明，我想表達的是「淬取」、「蒸餾」、「抽象化」這些字眼。

事實就是這樣，面對相同的技術、規格或架構，有人能夠很快看穿外頭的包裝，拆開外衣檢視內裏，瞭解背後的精神及理論基礎，挾著新的收獲朝更廣闊的領域邁進。

³ LPC 為 Local Procedure Call 的縮寫；IPC 為 Inter-Process Communication 的縮寫。

另外的一些人，為其能力或效果所感動迷惑，再三留戀，將實作細節摸得一清二楚，卻往往將焦點置於使用方式或實作機制，雖然下方的基礎及學問才距離幾公分，總是無緣相見。雖然得到了高超的技術能力，卻更加狹隘了視角，短淺了目光。

面對新的技術、語言及工具，以定義上看來，它們通通都是實作相關的產物。但是你看到了哪些？是實作細節？還是理論基礎？你才是決定答案的人。

第二章

VCL 基本心法

所謂擒賊先擒王，學 C++Builder 首要就在學好 VCL，
吾未曾見不精 VCL 心法的 C++Builder 高手。



瞧我整天將 VCL 掛在嘴邊，VCL 長 VCL 短的，不但與幾位好友組成 VCL Team，連申請的網域名稱“vclxx.org”都含有 VCL 這個字，十分誇張，有人還以為我開了一家公司叫 VCL 咧！沒關係，那人一定沒學過 C++Builder，我們大人有大量，就原諒他了。

我們知道，VCL 全名叫做 Visual Component Library，VCL 是 Delphi 及 C++Builder 所用的 application framework，VCL 提供許多元件，可供程式設計師在整合環境中操作、使用，除了這些具體的事實外，VCL 的「本體」究竟是啥東東？以生物的角度來看，它究竟是單細胞或多細胞、動物或植物、軟骨還是脊椎？或者，從檔案的角度來看，它究竟包含了哪些檔案？而我們又如何才能見到它，跟它打聲招呼？

線索並不多，不過我想，既然 VCL 是 C++Builder 的 application framework，那麼，在 C++Builder 產生的執行檔中，一定可以找到些蛛絲馬跡，至少採點指紋什麼的...

C++Builder 程式的組成

在 C++Builder 整合環境中，選取功能表的【File / New Application】選項，建立一個新的專案。專案建立後，立即將檔案儲存起來，專案命名為 Project1，單元命名為 Unit1，唯一的 form 則直接使用預設名稱 Form1。現在我手上就有了一個最正常的 C++Builder 應用程式，它是我們接下來的觀察對象。

執行檔成分解析

從功能表選取【Project / Options】選項，在「Project Options」對話盒中把「Linker」頁次的 Map file 選項調整至“Publics”等級，並在「Packages」頁次將“Build with runtime packages”選項取消。然後選擇【Project / Build】選項，編譯、連結完成後，可以在專案目錄下找到 Map file 檔案 Project1.map。嚇，Project1.map 竟然包含七千餘行文字！我擷取其中一部分列在下頭，仔細分析 Map file 所包含的資訊。

區段

#0001	Start	Length	Name	Class
#0002	0001:00401000	00000163AH	_TEXT	CODE
#0003	0002:00403000	000000664H	_DATA	DATA
#0004	0003:00403664	000000050H	_BSS	BSS
#0005	0004:00000000	00000009CH	_TLS	TLS

這些列出此執行檔擁有的區段（section），區段可包含程式碼、資料、資源、除錯資訊或其它任何資訊。執行檔載入時，整個執行檔，包括這些區段都會直接映射到該行程的位址空間，讓作業系統可在記憶體中直接取用區段資料及直接執行區段內程式碼，使得載入、執行 PE 格式執行檔的動作遠比古早時候的 NE 格式執行檔來得輕鬆多了。

Info

PE 為 Portable Executable 的頭字語，它是由 COFF（Common Object File Format）格式改良過來的可執行檔格式。

NE 為 New Executable 的頭字語，為 Win16 的可執行檔格式。

C++Builder 執行檔通常包括以下這些區段：

表 2-1 / C++Builder 執行檔通常包含的幾個區段

區段名稱	存放內容
_TEXT	程式碼。
_DATA	初始化資料。
_BSS	未初始化靜態變數及全域變數。
.idata	Import table，記錄隱式連結的外部模組函式。
.edata	Export table，記錄此模組提供給外界使用的函式。
.rdata	可能包含數種資料，例如程式敘述字串、GUID 等等。
.rsrc	各種資源，如字串、圖示、滑鼠指標等等。
.tls	執行緒私有變數。

.reloc 重定位表格。

以 DUMPBIN 來查看 Project1.exe 包含的區段：

```
c:\borland\C++Builder5\Bin>dumpbin project1.exe

Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file project1.exe

File Type: EXECUTABLE IMAGE

Summary

    1000 .data
    1000 .edata
    1000 .idata
    1000 .rdata
    5000 .reloc
    3000 .rsrc
    1000 .text
    1000 .tls
```

由上頭可看出，除了 BSS 區段，Project1.exe 正好包含表 2-1 所列的八個區段。至於為什麼 MAP file 沒有列出除了 _TEXT、_DATA、_BSS 之外的五個區段，我想只是它的設計考量，只希望列出與程式碼相關的區段吧！

單元

```
009  0001:00000000 C=CODE M=C0W32.OBJ ACBP=A9
#0010  0001:0000014F C=CODE M=OBJ\SYSINIT.OBJ ACBP=A9
#0011  0001:000002C0 C=CODE M=C:\TEMP\PROJECT1.OBJ ACBP=A9
#0012  0001:000004E0 C=CODE M=C:\TEMP\UNIT1.OBJ ACBP=A9
#0013  0001:00000CDC C=CODE M=RELEASE\VCL50.LIB|Contnrs ACBP=A9
#0014  0001:00000EA0 C=CODE M=RELEASE\VCL50.LIB|UrlMon ACBP=A9
#0015  0001:00000ED8 C=CODE M=RELEASE\VCL50.LIB|StdActns ACBP=A9
#0016  0001:00001034 C=CODE M=RELEASE\VCL50.LIB|ImgList ACBP=A9
#0017  0001:000024C4 C=CODE M=RELEASE\VCL50.LIB|ActnList ACBP=A9
#0018  ...
```

```
#0019 0001:00035378 C=CODE M=RELEASE\VCL50.LIB|CommCtrl ACBP=A9
#0020 0001:000353BC C=CODE M=RELEASE\VCL50.LIB|Math ACBP=A9
#0021 0001:000353FC C=CODE M=RELEASE\VCL50.LIB|SysConst ACBP=A9
#0022 0001:000356C4 C=CODE M=RELEASE\VCL50.LIB|SysUtils ACBP=A9
#0023 0001:0003972C C=CODE M=RELEASE\VCL50.LIB|System ACBP=A9
```

這裡列的是程式使用到的所有單元，對於每個單元你都可以在C++Builder\Lib、專案目錄及搜尋目錄下找到對應的 .OBJ、.LIB、.DCU¹ 或 .PAS檔案。

公開符號

接下來列出的是此程式使用到的，可分享給其它模組 (*extern*、*non-static*) 存取/呼叫的常數、變數、程序及函式。對於類別來說，只要是程式中使用到並且符合下列任一條件的成員函式都會列於此：

- 宣告於 `protected`、`public`、`__published` 區段。
- 使用到的屬性的屬性存取函式。

公開符號的種類不少，以下分爲 C/C++ RTL 函式、Object Pascal RTL 函式、VCL 函式、Win32 API 函式、VCL 元件等等幾大類來討論。

C/C++ RTL² 函式

```
#0022 Address          Publics by Name
#0023
#0024 000517B4 _strcat
#0025 000510F4 _malloc
#0026 00053510 _isascii
```

這三個函式都是 C 語言標準函式庫所提供的，由 ANSI C 規範，你可以在所有符合 ANSI

¹ Delphi Compiled Unit的頭字語，意思為編譯過的單元，相當於C/C++ 語言的OBJ檔。

² Run-Time Library的頭字語，意思為支援程式運作執行的函式庫。

C 標準的 C/C++ 編譯器／開發環境中找到同樣的幾個函式。它們的宣告如下：

```
_str.h
// 串接兩個字串
char _FAR * strcat(char _FAR *__dest, const char _FAR *__src);
alloc.h
// 從 heap 配置一塊記憶體
void* _RTLENTY _EXPFUNC malloc(_SIZE_T __size);
ctype.h
// 判定一個字元是否為 ASCII 字元
int _RTLENTY _EXPFUNC isascii (int __c);
```

Object Pascal RTL 函式

```
#0022 Address          Publics by Name
#0023
#0024 0004CF20 System::__linkproc__ __fastcall GetMem(int)
#0025 00036CEC __fastcall Sysutils::CompareMem(void *, void *, int)
#0026 000353BC __fastcall Math::Max(int, int)
```

這三個函式來自於 Object Pascal 語言，存在於 C++Builder 執行檔的主要目的是提供給程式執行所需的基本設施給 VCL 使用（因 VCL 以 Object Pascal 撰寫）。

雖然在 C++Builder 程式中，我們可以也自由地叫用 Object Pascal RTL 函式，但**除非必要**，否則**不建議使用**。無謂地叫用 Object Pascal RTL，會使得程式的維護性降低，可移植性更會受到影響。

在這兒面對了一個尷尬的情況：Object Pascal RTL（也就是組成 Object Pascal RTL 的 System、SysUtils 及 Math 三個單元所提供的服務）並不是純粹的「Object Pascal 語言 Run-Time Library」。Object Pascal 語言完全由 Borland 公司自行定義、實作³，在設計之初，完全就只為了 Delphi 這一套 Windows 上的應用程式開發工具設想，實作上並沒有太多跨平台的考量。所以你可以在 Object Pascal RTL 裡看到許多與 Windows 開發平台緊密相關，但與 Object Pascal 語言本身沒有太大關係的服務。例如下列宣告：

³ 雖然有 ANSI Pascal 標準，不過... Borland 的 Object Pascal 的確是自行定義的非標準規格。

```
System.hpp
extern HINSTANCE MainInstance; // 啟動目前行程的執行模組(.EXE) instance
extern unsigned MainThreadID; // 主執行緒 ID

SysUtils.hpp
extern PACKAGE int Win32MajorVersion;
extern PACKAGE int Win32MinorVersion;
```

其它的平台就不會有 *HINSTANCE* 這種資料型別，不一定具備多執行緒支援，且執行緒編號不一定是 *unsigned* 型別。

正因如此，所以雖然我們以 C++ 語言來撰寫 C++Builder 程式，卻時常發現想用的函式卻藏在 Object Pascal RTL 裡頭。例如：

```
System.hpp
extern bool IsLibrary; // 此模組是否為 DLL ?
// 取得命令列參數數目及字串
extern int __fastcall ParamCount(void);
extern AnsiString __fastcall ParamStr(int Index);

SysUtils.h
// 載入及卸除 package 模組
extern unsigned __fastcall LoadPackage(const AnsiString Name);
extern void __fastcall UnloadPackage(unsigned Module);
```

由上頭所舉的例子可以看出，Borland Delphi 的設計中，並沒有特意將 Object Pascal 與 Delphi 的 Run-Time Library 區分開來，而是將它們混合在一起，再依功能性切割，置於 System、SysUtils 及 Math 三個單元之中。

VCL 由 Object Pascal 語言撰寫，當然依賴著 Object Pascal RTL 的存在才能運作。

Borland C++Builder 架構於 VCL 之上，因此，它也必須延用 Object Pascal RTL，不能將它捨去。這造成了一套 C++Builder 開發環境內擁有兩套 Run-Time Library（C/C++ RTL 及 Object Pascal RTL）的奇怪景象。

Object Pascal RTL 函式提供的是十分基本，且與任何類別、元件無關的功能，分別來自 System、SysUtils 及 Math 單元（Object Pascal RTL 正是由此三個單元構成）。它們的宣

告如下：

```
System.hpp
// 從 heap 配置一塊記憶體
// 即 Object Pascal 裡的 GetMem() 函式
PACKAGE void * __cdecl GetMemory(int Size);
SysUtils.hpp
// 比較兩段記憶體範圍的異同
bool __fastcall CompareMem(void *P1, void *P2, int Length);
Math.hpp
// 比較兩個整數，傳回較大數值
int __fastcall Max(int A, int B);
__int64 __fastcall Max(__int64 A, __int64 B);
float __fastcall Max(float A, float B);
double __fastcall Max(double A, double B);
Extended __fastcall Max(Extended A, Extended B);
```

VCL ⁴ 函式

```
#0032 __fastcall Graphics::ColorToIdent(int, System::AnsiString&)
#0033 __fastcall Classes::RegisterClass(System::TMetaClass*)
#0034 __fastcall Controls::SetImeMode(unsigned int, TImeMode)
```

這三個函式分別由 Graphics、Classes 及 Controls 單元提供，屬於 VCL 的一部分。它們的宣告如下：

```
Graphics.hpp
// 將顏色數值轉換為字串描述
bool __fastcall ColorToIdent(int Color, AnsiString &Ident);
Classes.hpp
// 將類別登記至全域類別表格，使類別能在物件永續機制內使用
void __fastcall RegisterClass(TMetaClass* AClass);
Controls.hpp
// 設定輸入法使用狀態
void __fastcall SetImeMode(HWND hWnd, TImeMode Mode);
```

你可以發現，同樣都是函式，我卻把某些歸類為 RTL 函式，有些歸類為 VCL 函式，分類的依據是什麼呢？其實，光從函式的名稱及宣告很難分辨，必須根據函式的用途及種

⁴ 應該要記起來囉，它是 Visual Component Library 的頭字語。

類而定，大致上是這樣區分的：

- RTL 提供的函式大部分與程式語言本身、編譯器、作業系統及行程相關。
- RTL 提供的函式通常很普遍，在其它開發工具內也找得到類似的函式。
- RTL 宣告的類別都直接繼承自 *TObject*（只限於 Object Pascal RTL）或 RTL 內部的類別，與 VCL 的類別及元件完全不相干。
- 只要與 VCL 類別或元件有任何一丁點直接或間接關係的函式，就置於 VCL。

Win32 API⁵

#0028	0001:000050F0	BitBlt
#0029	0001:000052D8	ClientToScreen
#0030	0001:00004F60	CreateFile

以上三個函式由 Windows 單元提供，隸屬於 Win32 API，函式功能的實際提供者分別是作業系統的 GDI32.DLL、USER32.DLL 及 KERNEL32.DLL。它們的宣告如下：

```
// 將 SrcDC 上某塊區域的影像複製到 DestDC 上頭
WINGDIAPI BOOL WINAPI BitBlt( IN HDC, IN int, IN int, IN int, IN int,
    IN HDC, IN int, IN int, IN DWORD);

// 將座標由視窗客戶區域相對座標轉換為畫面絕對座標
WINUSERAPI BOOL WINAPI ClientToScreen(IN HWND hWnd,
    IN OUT LPPOINT lpPoint);

// 可建立與 I/O manager 相關的物件，並傳回物件 handle
// 例如檔案、pipes、mailslots、通訊裝置、consoles、目錄等等
WINBASEAPI HANDLE WINAPI CreateFileA(
    IN LPCSTR lpFileName,
    IN DWORD dwDesiredAccess,
    IN DWORD dwShareMode,
    IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    IN DWORD dwCreationDisposition,
    IN DWORD dwFlagsAndAttributes,
    IN HANDLE hTemplateFile
);
```

⁵ Application Programming Interface的頭字語，為作業系統提供給應用程式的服務介面。

這些函式的呼叫及使用方法雖然與 C++Builder 提供的函式（我指的是，由 RTL 或 VCL 提供的函式）並無二異，但事實上，這些函式的程式碼並不存在於我們的執行檔 Project1.exe 中，它們是由 DLL（動態連結函式庫）提供的。

對於被應用程式 *implicitly linked* 的 DLL，每當應用程式執行並建立新的行程時，這些 DLL 就會被自動載入該行程的位址空間。所以，若是被 *implicitly linked* 的 DLL 不存在，建立行程就會出現錯誤訊息，導致該程式無法執行。這也是為什麼老是有許多人叫著：「在我的電腦上明明可以執行，可是拿到別人電腦上就無法執行，說缺少檔案什麼的」的實際原因。

因此，對於那些並不是每部電腦上都有的 DLL，有時我們會採用 *explicitly linking* 方式。做法是，程式執行時期，利用 *LoadLibrary* API 函式將 DLL 載入，再呼叫 *GetProcAddress* API 函式取得函式位址，再根據得到的函式位址直接叫用。此時 DLL 的載入成功與否並不會影響行程的建立，頂多只是程式的某些功能無法使用罷了。

無論如何，不論是 *implicitly* 或 *explicitly* 連結方式，呼叫 DLL 提供的函式之前，DLL 必須先載入到程式本身的位址空間內，此函式才能順利叫用。下圖是應用程式呼叫 DLL 函式的動作，由於在同一個位址空間內，所以可以直接呼叫：

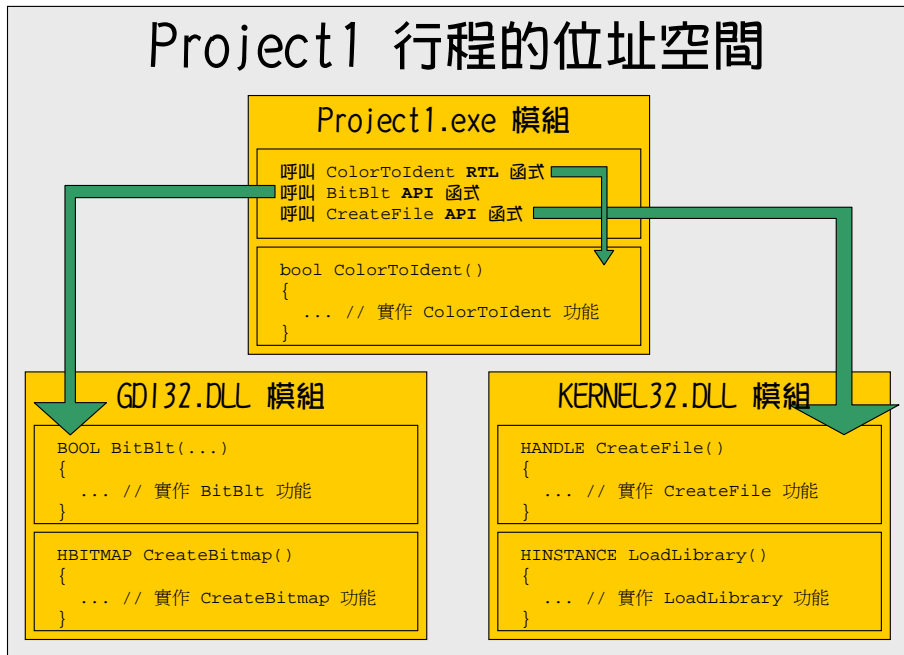


圖 2-2 / 呼叫 C++Builder 本身函式以及 DLL 函式的不同情況

從上圖你可得到「模組」的概念。在行程的位址空間內，除了執行檔本身外，還有許多其它模組存在，這些模組的本體通常是 DLL。

KERNEL32.DLL、USER32.DLL及GDI32.DLL三者合稱為Win32 API三大模組，這是因為它們幾乎被所有的Windows程式使用⁶，C++Builder程式當然也不例外。

如何得知程式使用到哪些模組呢？C++Builder 整合環境提供的 Modules 除錯視窗就可以看到，如下圖。可以看出，除了 Project1.exe 本身及 Win32 三大模組外，Project1 行程位址空間中還存在著不少模組呢：

⁶ 若你寫的是console mode程式，那麼可能就不會使用到GDI32.DLL。

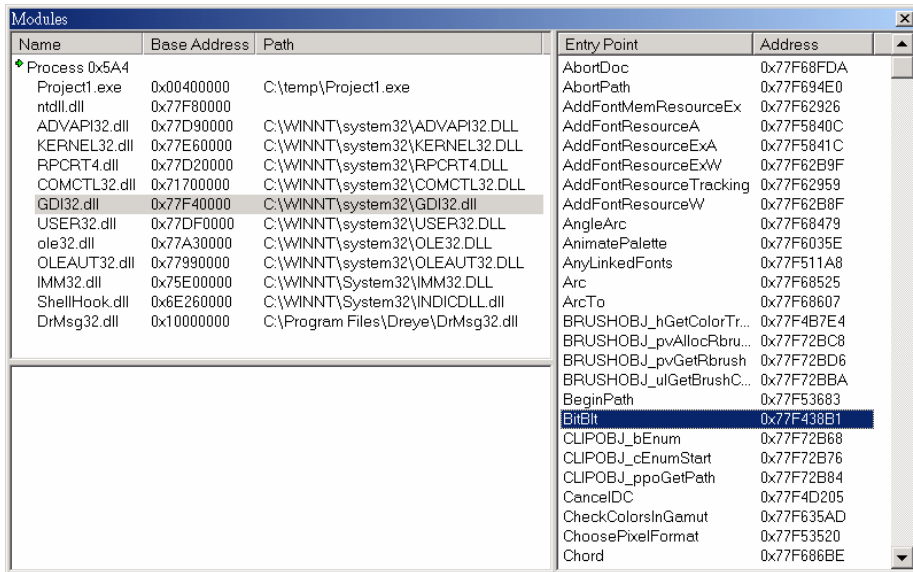


圖 2-3 / C++Builder 整合環境提供的 Modules 視窗，可以看到 GDI32.DLL 提供的 BitBlt 函式及它的位址

這些模組大部份是應用程式主動要求載入的，例如 Win32 API 三大模組、負責登錄資料庫處理的 ADVAPI32.DLL、負責提供輸入法的 IMM32.DLL 等等，都是因為 Project1.exe 本身會用到它們的功能，所以主動載入的；有些模組則由被載入的 DLL 間接載入。另外還有些模組是系統或其它程式要求進入的，例如最下頭 DrMsg32.DLL 模組，它們是被 Dr.Eye 即時翻譯軟體硬塞入的。關於這些由應用程式提供，主動進入其它程式位址空間的不速之客，我在第五章「一頭栽入桌面的世界」談到跨行程 subclassing 技術時會再說明。

VCL 元件

```
#0036 0003433C __fastcall TComponent::TComponent(TComponent *)
#0037 ...
#0038 000344A4 __fastcall TComponent::WriteTop(Classes::TWriter *)
#0039 ...
#0040 00024A54 __fastcall TControl::ActionChange(TObject *, bool)
#0041 ...
```

```
#0042 00023FB4 __fastcall TControl::WndProc(Messages::TMessage&)
#0043 ...
#0044 00018F9C __fastcall TCustomForm::Activate()
#0045 ...
#0046 00016DF4 __fastcall TCustomForm::WriteTextHeight(TWriter*)
#0047 ...
#0048 0003D2C8 __fastcall TObject::AfterConstruction()
#0049 ...
#0050 0003D2BC __fastcall TObject::SafeCallException(TObject*,void*)
#0051 ...
#0052 00029A54 __fastcall TWinControl::ActionChange(TObject *, bool)
#0053 ...
#0054 00026974 __fastcall TWinControl::WndProc(Messages::TMessage&)
```

上面所列出的並不是函式，而是 VCL 類別的成員函式。目前 Project1 程式只包含一個 form，且 form 上面空空如也，所以這裡列出的一定是跟 *TForm* 類別有關的類別，可能是它的父代類別，也可能是它的父代類別或 *TForm* 類別本身包含／使用的其它類別。例如上列的 *TObject*、*TComponent*、*TControl*、*TWinControl* 等等類別，通通都是 *TForm* 的父代類別。

只要一扯上元件，或是 *TPersistent* 類別，或是 *TPersistent* 類別的後代類別，就一定屬於 VCL 的範疇。VCL 包括我們在整合環境中看到的所有元件，以及任何與元件有關的類別、函式及機制。

組成份子

從執行檔中，我們可以看到，原來一個程式的組成這麼複雜。除了自行撰寫的程式碼以外，光是 C++Builder 提供的程式庫就可分為 RTL 及 VCL 兩部分，另外有些函式由作業系統以 DLL 形式提供，必須透過「連結」、「載入」等機制，將 DLL 載入到行程的地址空間內，才可以使用。

來張圖將這些應用程式的組成份子釐清。下圖是一般 C++Builder 應用程式的程式使用／呼叫情形：

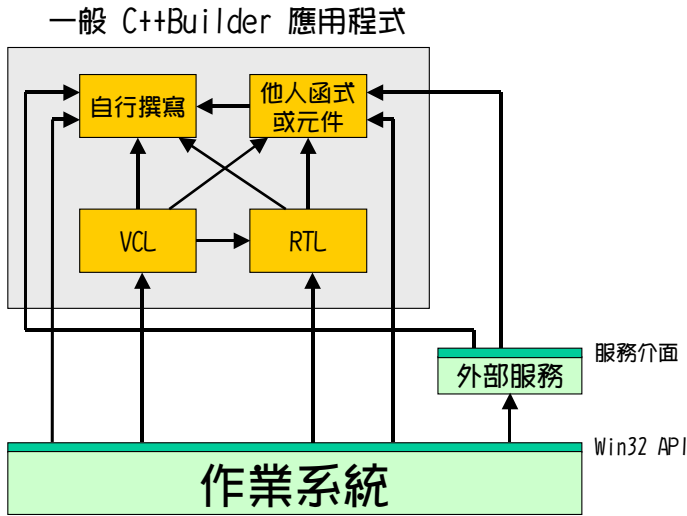


圖 2-6 / 一般 C++Builder 應用程式的程式使用 / 呼叫情形

圖中的線條牽來扯去，有點複雜。我歸納說明一下：

- 作業系統提供許多服務讓應用程式使用，這些服務通稱為system call⁷。System call 在各種作業系統下可能有不同的稱號，也可能有不同的提供方式：
 - 在 DOS 中，系統服務以 21h 中斷服務常式 (ISR) 的形式存在，稱為 DOS function calls。
 - 在 UNIX 中，系統服務以 shared library 形式存在，稱為 system calls。
 - 在 Win32 作業系統中，系統服務以 DLL 形式存在，稱為 Win32 API。
- 此處所指的Win32 API，泛指所有作業系統提供給ring 3⁸應用程式的服務，服務提供者不限於Win32 的KERNEL32、USER32、GDI32 三大模組，服務形式也不限於

⁷ 雖然現代作業系統的系统服務不見得以“call”、“呼叫”方式使用，不過習慣上仍通稱為system call。

⁸ 雖然Intel x86 家族CPU提供四種保護模式層級，不過Win32 作業系統只使用其中ring 0 及ring 3 兩種：Ring 0 規劃為kernel mode，ring 3 規劃為user mode。

函式呼叫。例如Multimedia API、MAPI、TAPI、OpenGL、Pen API等等服務，都由三大模組以外的DLL提供；而COM、OLE、Shell API、DirectX等等服務，都架構於COM之上，以物件、介面形式提供。這些服務總稱為Win32 API。

- 外部服務泛指非作業系統提供，且不包含於程式執行檔的服務。例如程式可能使用某個支援 ZIP 格式解壓縮的 DLL 來進行解壓縮工作，以函式呼叫的形式取得服務；也可能在程式中使用某個網頁瀏覽控制元件，此元件是別人撰寫的 ActiveX Control，雖然 ActiveX Control 通常也置於 DLL 檔案（將 DLL 檔當作載具），但服務的提供是基於 COM 上進行的。
- RTL 及 VCL 的根本能力都來自於作業系統，許多功能都只是系統服務的再包裝，將相同的功能包裝得更方便、更好用、更適合程式撰寫呼叫。

以記憶體管理功能為例，雖然 Win32 API 提供 *VirtualXXXX* 及 *HeapXXXX* 兩組記憶體管理函式，但是於效率、於性質而言，並不適合一般應用程式使用，因為一般應用程式通常有記憶體配置／歸還動作次數多、配額少的特性。因此，幾乎所有開發工具的 RTL 都會自行提供一套記憶體管理函式，例如 C 的 *malloc* 及 *free* 函式、C++ 的 *new* 及 *delete* 保留字，以及 Pascal 的 *GetMem* 及 *AllocMem* 函式等等。

- 撰寫程式時，大部分的需求都可以 RTL 或 VCL 來解決；若不滿足於 RTL 及 VCL 的功能時，也可以直接使用 Win32 API 或其它外部服務。

接著，針對這些組成份子，分別為你做進一步的介紹。

RTL

RTL 是 C++Builder 程式活動的動力來源。在 C++Builder 中，根本沒有辦法寫出不使用 RTL 又能夠執行的程式。本章稍前已經提過，C++Builder 的 Run-Time Library 包含取自 Delphi 的 Object Pascal RTL，以及 C/C++ 本身的標準函式庫。

C/C++ 標準函式庫皆由 ANSI 制定，再由各家開發工具廠商分別實作，所以除了實作上的差異，其它部分可視為相同，在此就不贅述，只介紹取自於 Delphi 的 Object Pascal Run-Time Library。

Object Pascal RTL 大致包括這些傢伙：

表 2-7 / Object Pascal RTL 包含的單元及貢獻

單元	貢獻
System	<ul style="list-style-type: none">■ <i>TObject</i> 類別：所有類別的始祖■ 記憶體管理■ <i>Variant</i> 型別及動態陣列支援
SysUtils	檔案處理、例外處理、字串處理、日期處理、Package 管理
Math	數學相關函式

VCL

只要你使用到 C++Builder 整合環境所提供的 RAD 特性及工具，例如 Form Designer、物件檢視器、元件盤等等，就一定用到了 VCL，只不過不一定將 VCL 作為程式的 application framework。下表列出 VCL 包含的主要單元及貢獻：

表 2-8 / VCL 包含的主要單元及貢獻

單元	貢獻
Classes	<ul style="list-style-type: none">■ <i>TPersistent</i> 類別：引入物件永續機制■ <i>TComponent</i> 類別：VCL 元件的始祖■ <i>TList</i>、<i>TStrings</i>、<i>TCollection</i> 等工具類別
Controls	<ul style="list-style-type: none">■ <i>TControl</i> 類別：可視元件的始祖■ <i>TWinControl</i> 類別：視窗元件的始祖
StdCtrls	Windows 標準控制項的包裝
ExtCtrls	VCL 自行提供的可視元件
ComCtrls	Win32 控制項的包裝
Graphics	高效率、易使用的 GDI 封裝類別
Forms	<ul style="list-style-type: none">■ <i>TForm</i> 類別：應用程式視窗■ <i>TApplication</i> 類別：視窗訊息的處理
TypeInfo	執行時期型別資訊 (RTTI)

Win32 API

RTL 及 VCL 都是架構於 Win32 API 上的程式庫，沒有作業系統的支援，它們只是兩沓什麼事也做不了的程式碼。

不論何種服務形式，絕大部分的 Win32 API 都必須先向編譯器宣告它們的存在，以及想要使用它們的強烈欲望，而後才能在程式中順利使用它們，所以 C++Builder 內附不少 Win32 API 宣告單元。如果要使用某某 Win32 API 服務時，只要將對應的宣告單元含入（使用 `#include` 編譯指示），將對應的 .LIB 檔一併連結，且系統上存在著提供該服務的苦主（通常是 DLL），程式就可以順利使用該服務。

以最常見的函式呼叫服務而言，例如我希望透過 WSocket32.DLL 提供的 `send` 函式來發送封包，只要程式中有下面這幾行宣告：

```
int PASCAL FAR send(  
    IN SOCKET s,  
    IN const char FAR * buf,  
    IN int len,  
    IN int flags);
```

程式啟動載入時，WSocket32.DLL 會被自動載入行程的位址空間，載入程式會將程式呼叫的 `send` 函式位址指向 WSocket32.DLL 模組中的 `send` 函式主體。當然，這兩行宣告不必我們自己來，C++Builder 內附的 `winsock.h` 內含 WSocket32.DLL 所有函式的宣告，只要將 `winsock.h` 標頭檔含入即可，連結器會自動在指定的目錄下尋找對應的 .LIB 檔案一併連結。

又，假設我想使用 Shell API 提供的 `IShellIcon` 介面，程式中必須先行宣告：

```
DECLARE_INTERFACE_(IShellIcon, IUnknown) // shi  
{  
    // *** IUnknown methods ***  
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, void **ppv) PURE;  
    STDMETHOD_(ULONG,AddRef) (THIS) PURE;  
    STDMETHOD_(ULONG,Release) (THIS) PURE;
```

```
// *** IShellIcon methods ***
STDMETHOD(GetIconOf)(THIS_ LPCITEMIDLIST pidl, UINT flags,
                    LPINT lpIconIndex) PURE;
};
```

同樣地，這宣告也不必我們自己來，它宣告於 C++Builder 內附的 shlobj.h 標頭檔。

下表列出部分 C++Builder 5 內附的 Win32 API 宣告單元以及它們宣告的服務：

表 2-9 / 部分 C++Builder 內附 Win32 API 宣告標頭檔及宣告的服務

宣告標頭檔	宣告的服務
winbase.h	檔案輸出入、行程、執行緒、通訊、記憶體 登錄資料庫處理
winuser.h	使用者介面、視窗、對話盒、視窗訊息 視窗訊息編號及視窗訊息結構
wingdi.h	繪圖
comdef.h	基本的 COM 介面宣告
cpl.h	控制台及控制台元件相關宣告
imm.h	輸入法 API
mmsystem.h	多媒體 API
gl.h	OpenGL ⁹ API 宣告
shellapi.h	Shell API 宣告

外部服務

除了作業系統提供的服務，任何人也可以撰寫服務提供程式來輔助其它應用程式的運作。這些服務的提供也與 Win32 API 相同，可以有各種形式，例如函式呼叫、COM 介面或 ActiveX Control 等等。

⁹ 由SGI公司主導發展的 2D、3D圖形rendering程式庫。

例如，網路上有個很有名的免費影像檔載入／儲存函式庫NViewLib¹⁰，它以DLL函式呼叫的形式提供服務。將NViewLib整個套件下載回來後，發現套件裡只附著DLL檔、範例程式及說明文件。說明文件上註明著它提供的函式原型，下面列出其中兩個：

```
// 載入各種格式的影像檔，傳入檔案，取得 bitmap handle
extern HBITMAP _stdcall NViewLibLoad(char* FileName, bool bProgress);

// 將目前顯示的影像儲存為 JPG 格式影像檔
extern bool _stdcall NViewLibSaveAsJPG(int Quality, char* FileName);
```

因為這是自行撰寫的DLL，並不是所有C++Builder程式設計師都會用到，所以C++Builder不可能內附此DLL的宣告單元。因此，你必須根據說明文件，將上述兩個函式的宣告加入程式中（或另外撰寫一個宣告單元也成），並將NViewLib.DLL置於專案目錄或系統目錄下，並且連結對應的.LIB檔案，此後，你的應用程式就可以順利使用NViewLib所提供的函式。

由於DLL函式呼叫、COM介面和ActiveX Control等等服務都建構於可跨越程式語言及開發工具的機制，因此C++Builder程式設計師也可使用以Delphi、VB、Visual C++等等開發工具撰寫出來的各種服務。在Win32的各種服務種類中，以C/C++撰寫的函式呼叫型DLL居多。

通常，當我們由網路下載別人撰寫的DLL時，整個套件通常包括下列檔案：

表 2-10 / 他人撰寫的 DLL 套件通常會包括的檔案

檔案種類	描述
.DLL 檔案	DLL 檔，這是與語言、開發工具無關的。
.LIB 檔案	C/C++ 開發工具必須使用的 import library。
.H 函式標題檔	C/C++ 語言撰寫的函式宣告檔。
說明文件	通常為英文撰寫的函式使用說明。

¹⁰ 書附光碟內的「Delphi深度歷險」網站就有收錄。

其中，.DLL 檔案以及 .H 標頭檔是最重要的，缺一不可。但是 .LIB 檔案可能就沒有附上，或者若此 LIB 由 Visual C++ 產生，C++Builder 無法使用（因 Visual C++ 使用 COFF 格式、而 C++Builder 採用 OMF 格式），此時我們就必須自行產生 .LIB 檔案。

產生 .LIB 檔案的步驟十分簡單，假設我們要為 NViewLib.DLL 產生 NViewLib.LIB 檔案：

```
c:\ NViewLib >implib NViewLib.lib NViewLib.dll

Borland ImplibVersion 3.0.22 Copyright (c) 1991, 2000 Inprise Corporation

c:\NViewLib>
```

此 .LIB 怎麼使用呢？請開啓 Project Manager，將它加入此專案裡頭，連結器就會將此 .LIB 檔與程式一併連結，如此才可順利叫用 DLL，如下圖。

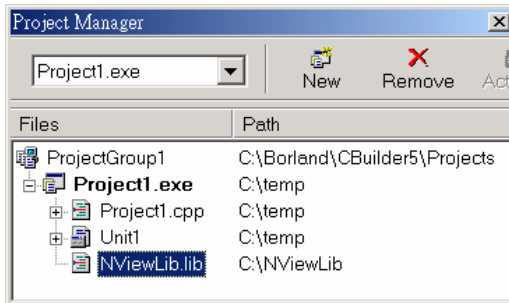


圖 2-11 / 在 Project Manger 中指定連結某一個 LIB 檔案

他人撰寫的程式或元件

我們也可以拿別人撰寫的程式或元件放在程式內，一併連結使用。通常有下列幾種使用形式：

- C/C++ 程式碼
 - 可能只是一小段程式、幾個函式、幾個類別、幾個單元或整套函式庫，可以 C++、OBJ 或 LIB 檔案形式納入專案。

- Object Pascal 程式碼
C++Builder 可以編譯／連結 Object Pascal 程式碼，但必須以 Object Pascal 的單元（Unit）為單位，且只能以原始碼形式 PAS 檔案形式納入專案。
- VCL 元件
通常是一個或數個單元，單元內包含 *TComponent* 後代類別的宣告及實作，並提供註冊函式（我指的是 *Register* 函式），供使用者透過功能表的【Component / Install Component】選項將元件註冊至整合環境。VCL 元件可以 PAS、CPP、OBJ 或 BPL 檔案形式移交。

自行撰寫的程式碼

這裏指的就是我們直接在專案裡頭撰寫的每一行程式碼。

由於應用程式是最後才寫出來的，因此可以享有最多的自由及資源。撰寫程式碼時，可以任意呼叫／使用 RTL、VCL、Win32 API、外部服務、他人撰寫的程式或元件，務必以達成應用程式目的為目的，手段請隨意，各種服務都可使用。

VCL 的多重面貌

經過以上的分析介紹，相信 VCL 對你來說應該不再是虛無飄渺的名詞，可以確實感受到它的存在。不過，對於 VCL 這麼可愛的玩意兒，光憑感覺是不夠的，最好看得到、摸得到、或是咬得到，對吧！

我大概沒有辦法讓你摸到或咬到 VCL，不過，可以帶你從各種時機及各種角度來看看 VCL 的各種面貌哦！Hey, guys and gals！Follow me！

VCL 的觀察時機有兩種：一種是獨立的，與任何應用程式、執行檔無關的，也就是開始被 C++Builder 程式設計師使用前，我稱此為「單身時期」；另一種是使用 VCL 的程式

在編譯、連結後，必須使用 VCL 來輔助程式的運行，與 VCL 有著不可分離的關係，我稱此為「死會時期」。讓我們分別從這兩個時期來觀察 VCL 的真面目。

單身時期

原始碼

什麼是 VCL？VCL 在哪裏？

VCL 就是以 Object Pascal 語言撰寫而成的超大型類別程式庫，分置於下列目錄：

- C++Builder\Source\Vcl
- C++Builder\Source\Decision Cube
- C++Builder\Source\Internet
- C++Builder\Source\Comservers
- C++Builder\Source\Toolsapi
- C++Builder\Source\Webmidas

統計的結果，C++Builder 5 的 VCL 原始碼¹¹ 約二十六萬行左右。

編譯後的單元

什麼是 VCL？VCL 在哪裏？

VCL 就是編譯過的 Object Pascal 單元，置於 C++Builder\Lib 目錄，以 DCU、OBJ 及 DCP 檔案格式存在。

¹¹ 並不是所有的 C++Builder 版本都提供 VCL 原始程式碼，只有 Enterprise、Client/Server 及 Professional 版本才提供。

C++Builder 5 的 VCL DCU 檔案約有近三百個。

類別及元件

什麼是 VCL？VCL 在哪裏？

VCL 就是撰寫 C++Builder 程式時，提供給程式設計師的那些現成類別及元件。元件也是類別，只不過是給 *TComponent* 後代類別的特別稱號，程式碼中可以使用的類別及元件依整合環境所載入的 design-time package 而定。

C++Builder 5 VCL 包含的類別約有一千五百個左右，其中元件約有二百餘個。

死會時期

執行檔

什麼是 VCL？VCL 在哪裏？

VCL 就是 C++Builder 執行檔裡頭，至少佔有好幾百 K 大小的機械碼。

接下來的兩個範例，將「Build with Runtime Packages」及「Dynamic RTL」選項皆取消，完全使用靜態連結，做個公平的比較。

請看，這是一個什麼事都沒做的 console mode 程式：

```
20002/01/21 06:35p          47,104 Project1.exe
```

大小只有 47K，因為它只用到 RTL 的 System 單元，與 VCL 完全無關。

而這個呢，是一個什麼事都沒做，但包含一個 form 的 VCL 程式：

2002/01/21 06:38p	359,936 Project1.exe
-------------------	----------------------

359936 位元組減掉 47104 位元組等於 312832 位元組，多出來的這些就是 VCL 機械碼，裡頭包含 *TForm*、*TWinControl*、*TControl*、*TPersistent*、*TObject*、*TFont*、*TApplication*、*TScreen*... 等等幾十個類別以及相關類別的成員函式、屬性程式碼。由於 C++Builder 連結程式具有“Smart Linking”功能，並不是以整個單元為單位進行連結動作，一定是程式中用到的類別才會被連結進去。

Run-Time Package

什麼是 VCL？VCL 在哪裏？

VCL 就是隨 C++Builder 附送的一堆 runtime packages 檔案，這些 runtime packages 其實是特殊形式的 DLL，只有同一個版本的 C++Builder 應用程式才能認得（在 RTL 的支援下）這些 runtime packages，並且使用置於其中的 VCL 類別、物件、函式及變數。

在正常情況下，DLL 只能提供一道道的函式供程式呼叫，沒有辦法經由 DLL 供給類別或物件，簡單地說，DLL 對外的介面必須是函式。C++Builder 提供的 packages 克服了這個問題，它設計出特別的 DLL 介面，讓置於裡頭的類別、物件及變數也可以直接被載入它的 C++Builder 應用程式使用。

表 2-12 / C++Builder VCL Runtime Packages 一覽表

Package 名稱	單元名稱
VCL50.BPL	Ax, Buttons, Classes, Clipbrd, Comctrls, Commctrl, Commdl, Comobj, Comstrs, Consts, Controls, Ddeml, Dialogs, Dlgs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Forms, Graphics, Grids, Imm, IniFiles, Isapi, Isapi2, Istreams, Libhelp, Libintf, Lzexpand, Mapi, Mask, Math, Menu, Messages, Mmsystem, Nsapi, Ole2I, Oleconst, Olectrls, Olectrls, Oledlg, Penwin, Printers, Proxies, Registry, Regstr, Richedit, Shellapi, Shlobj, Stdctrls, Stdvcl, Sysutils, Tlhelp32, Toolintf, Toolwin, Typinfo, Vclcom, Virtintf, Windows, Wininet, Winsock, Winspool, Winsvc
VCLX50.BPL	Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs
VCLDB50.BPL	Bde, Bdeconst, Bdeprov, Db, Dbcgrids, Dbclient, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables, Dsintf, Provider, SMintf
VCLDBX50.BPL	Dblookup, Report
DSS50.BPL	Mxarrays, Mxbutton, Mxcommon, Mxconsts, Mxldb, Mxcube, Mxdssqry, Mxgraph, Mxgrid, Mxpivsrc, Mxqedcom, Mxqparse, Mxqryedt, Mxstore, Mxtables, Mxqvb
QRPT50.BPL	Qr2const, Qrabout, Qralias, Qrctrls, Qrdatasu, Qrexpld, Qrextra, Qrprev, Qrprgres, Qrprntr, Qrqred32, Quickrpt
TEE50.BPL	Arrowcha, Bubblech, Chart, Ganttch, Series, Teeconst, Teefunci, Teengine, Teeprocs, Teeshape
TEEDB50.BPL	Dbchart, Qrtee
TEEUI50.BPL	Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Brushdlg, Bubbledi, Custedit, Dbedit, Editchar, Flineedi, Ganttedi, Ieditcha, Pendlg, Pieedit, Shapeedi, Teeabout, Teegally, Teelish, Teeprevi, Teeexport
VCL50.BPL	Sampreg, Smpconst

這是一個什麼事都沒做，包含一個 form 的一般 VCL 程式（靜態連結）：

2002/01/21 08:10p	359,936 Project1.exe
-------------------	----------------------

這是一個什麼事都沒做，包含一個 form 的 VCL 程式，但指定使用 runtime packages：

2001/01/21 08:20p	76,800 Project1.exe
-------------------	---------------------

76800 位元組減掉 359936 位元組等於 -283136 位元組，少掉的那些機械碼到哪去了？

它們是憑空消失了沒錯，因為，只要在 C++Builder 版本相同的前提下，VCL 就是 VCL，

放在哪都沒差。所以指定使用 runtime packages 的結果是，不像平常一樣將 VCL 編入執行檔，VCL 機械碼改由 runtime packages 提供。好處是，執行檔的大小可以像做夢般地縮小；缺點是，系統上必須存在著該程式使用到的 runtime packages 才行。

檢查看看瘦身後的 Project1.exe 是否和預期一樣使用 runtime packages：

```
c:\Borland\C++Builder5\Bin>dumpbin /imports project1.exe

Section contains the following imports:

VCL50.BPL
  416240 Import Address Table
  4161E0 Import Name Table
    0 time date stamp
    0 Index of first forwarder reference

    0 @System@initialization$qqrv
    0 @System@Finalization$qqrv
    0 @System@UnregisterModule$qqrpl7System@TLibModule
    0 @System@RegisterModule$qqrpl7System@TLibModule
    0 @System@LoadResourceModule$qqrpc
    0 @System@FindHInstance$qqrpv
    ...

VCL50.BPL
  416580 Import Address Table
  4164C8 Import Name Table
    0 time date stamp
    0 Index of first forwarder reference

    0 @Forms@initialization$qqrv
    0 @Forms@Finalization$qqrv
    0 @Forms@TApplication@Run$qqrv
    ...

...
```

果然，此時 Project1.exe 必須依賴 VCL50.BPL 才可以執行，瘦身總是有代價的：執行檔檔案雖小，但是無法在沒有 VCL50.BPL 檔案的系統上執行。下圖是不用 runtime packages 及使用 runtime packages 的 C++Builder 程式執行狀況：

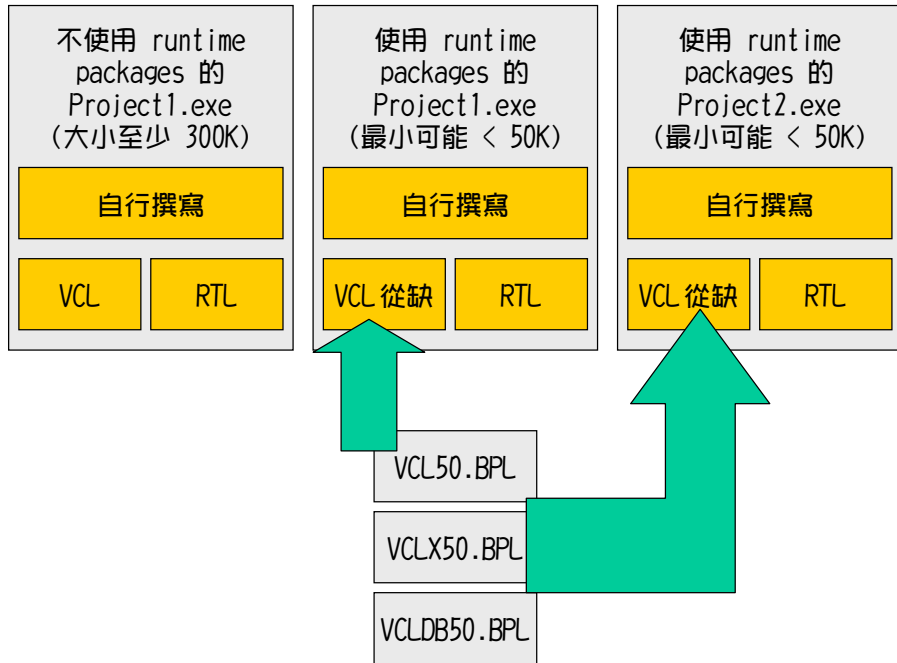


圖 2-13 / Runtime packages 就是 DLL，可被多個行程分享使用

VCL 類別架構

花了這麼多篇幅研究 VCL 後，壓下我心愛鍵盤上的【V】、【C】、【L】三個按鍵時都覺得有點於心不忍，按壓的感覺有點遲頓，它們三兄弟似乎快被這一章大量出現的「VCL」給操壞了～。

之前的研究以 VCL 的角色、VCL 的任務、VCL 的面貌（才剛說完，又是一堆 VCL...）等等特性為主題，只著重 VCL 的整體觀，完全不論及 VCL 的內部情形。

現在鏡頭切換，既然已清楚 VCL 的角度及任務，那麼就可放心地鑽入 VCL 內部進行探索，看看架構龐大的 VCL，究竟如何掌管、分類門下眾多的類別、元件，認識一下 VCL 內的堂口、幫派，跟堂主、幫主打打招呼問個好，以便日後在 VCL 世界裡混口飯吃。

其實，由於物件導向的繼承特性，類別庫（也就是包含一大堆類別的程式庫）比起一般的函式庫還要好記、好學。比如說，今天學習類別 A 的特性及行爲後，明天你叫我學習類別 B，但我一查書，發現類別 B 直接繼承自類別 A，根據我對類別 A 的瞭解，我已經曉得類別 B 大致的特性及行爲了。接下來，只要查清楚類別 B 究竟更改了什麼特性，新增了什麼行爲，再花一點點力氣，類別 B 也就學會囉。

舉個 VCL 的實例。假設我知道下列幾件事實：

- 我知道 *TComponent* 是元件的始祖，有 *Name* 及 *Tag* 兩個屬性。
- 我知道 *TControl* 是可視元件的始祖，有 *Left* 及 *Right* 兩個屬性，有 *Show* 及 *Hide* 兩個函式。
- 我知道 *TButton* 是元件，而且是看得見的元件，所以它一定是 *TControl* 類別的後代，也是 *TComponent* 類別的後代，因為 *TControl* 繼承自 *TComponent*。

那麼，就可以歸納出結論—*TButton* 一定有 *Name*、*Tag*、*Left*、*Right* 四個屬性，也一定有 *Show* 及 *Hide* 兩個函式可用。

瞧，很簡單吧。所以學習類別庫時切勿死背，絕對是有訣竅的：

- 先求對類別庫的整體架構有著清楚的大局觀，並瞭解為什麼要有這樣的設計。
- 牢記重要類別在類別庫中的地位、繼承關係以及它們的屬性與成員函式（即特性與行爲），別擔心，就算是再大的類別庫，重要的類別頂多也只十來個。
- 對於不清楚其繼承關係的類別，試著藉由特性及行爲來判斷它在類別庫中的位置。
- 對於不清楚其特性及行爲的類別，試著藉由它在類別庫中的位置來進行判定。

因為，幾乎可以說，若你能徹底瞭解核心類別所扮演的角色，且能夠堅定無誤地針對每個核心類別回答出下列問題：

- 它為什麼在那兒？
- 它在那兒為類別庫帶來什麼好處？
- 它若不在那兒，會對類別庫帶來什麼負面的影響？

網提其綱，則眾目自伸，若衣挈其領，則群纒必直。雖然 VCL 裡有上千個類別，但我只要指出最重要的幾個類別給你看，將你腦海中的 VCL 框架築好，接下來的細節再慢慢填塞就行了。框架搭得好，VCL 學好只是遲早的問題。

核心類別

正如其名，下列三個核心類別（呈灰色陰影者）是 VCL 裡最重要的類別，它們是其它類別的基礎建設，請務必清楚認識它們。

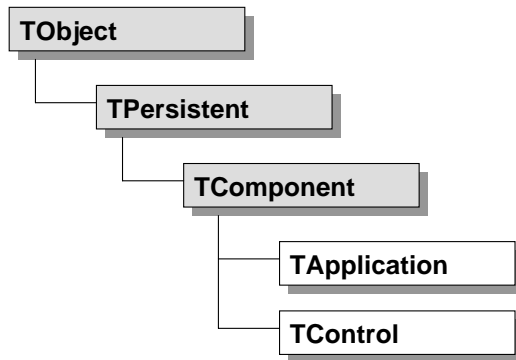


圖 2-14 / VCL 核心類別的階層關係圖

這三個類別，缺一不可，任意去掉一個，VCL 就無法成爲 VCL 了。下表先簡單列出這幾個核心類別與其主要貢獻，讓你先有個概括性的認識，接著再詳細介紹每個類別。

表 2-15 / VCL 核心類別及主要貢獻

類別	父類別	宣告單元	主要貢獻
<i>TObject</i>	無	System	Object Pascal / VCL 裡所有類別的始祖
<i>TPersistent</i>	<i>TObject</i>	Classes	資料流讀寫能力
<i>TComponent</i>	<i>TPersistent</i>	Classes	所有 VCL 元件的始祖

TObject

Object Pascal 和 Java 一樣，具有唯一的始祖類別，這種架構稱為 *single-rooted hierarchy*，它的特性是：程式員沒有辦法建立一個不繼承自始祖類別的類別。在 Object Pascal 語言中，始祖類別叫做 *TObject*；在 Java 語言中，始祖類別叫做 *Object*。

例如，以 Object Pascal 語言隨手定義這麼一個類別：

```
TDog = class
  procedure Wow; // 汪汪叫
end;
```

即使 *class* 保留字之後沒有接著任何類別，但在 Object Pascal 編譯器的眼中，看起來和下列宣告一模一樣：

```
TDog = class(TObject) // TDog 是 TObject 的子類別
  procedure Wow; // 汪汪叫
end;
```

這種架構由編譯器支援，強制所有類別一定都直接或間接繼承自 *TObject* 類別。

這種架構有什麼好處呢？好處可多囉！以 Object Pascal 的 *TObject* 類別為例：

- 對於程式中的任何物件，不論是自己生的，或是路上撿的，通通可以將它當成 *TObject* 物件來處理，因為該物件的類別必定是 *TObject* 後代類別。
- 可在 *TObject* 類別加入每個物件都要具備的能力，例如建立／摧毀物件本身、訊息處理機制、RTTI 支援、垃圾回收（*garbage collection*）機制等等。
- 由於 *TObject* 類別的建立／摧毀機制，強迫所有物件都建立在累堆（*heap*）中，這可大大簡化物件傳遞的處理¹²。

¹² 如果你知道 C++ 爲了支援物件傳遞動作而設計的複雜機制，就會瞭解爲什麼我這麼說。

相對而言，因為 C++ 不是 single-rooted hierarchy 架構，得到了多一點的使用彈性及靈活性，但是以上的優點通通沒有，所以若要實作如 RTTI、訊息處理等機制時，總得透過較複雜較麻煩的手法來達成，使得 C++ 程式撰寫的複雜度再加三級，快到天庭了。

挾著 single-rooted hierarchy 架構的優勢，VCL 的類別總長 – *TObject* 不負眾望地，提供下列諸多能力：

配置、初始化及歸還物件本身的記憶體

由於這一點，強迫 Object Pascal 中所有物件都只能建構於累堆中，無法和 C++ 一樣，能夠在堆疊、資料區段、累堆三種地方建立物件。假設 *TCat* 是一個類別，那麼你在 C++ 程式碼中可以看到這樣的物件建立方式：

```
#0001 void foo(void)
#0002 {
#0003     TCat cat; // 在堆疊中建立 TCat 類別的 cat 物件，進入 foo 時才建立
#0004
#0005     cat.Meow(); // 呼叫 cat 物件的 Meow() 函式
#0006 }
```

或是

```
#0001 TCat cat; // 在資料節區中建立 TCat 類別的 cat 物件，程式啟動後即建立
#0002
#0003 void foo(void)
#0004 {
#0005     cat.Meow(); // 呼叫 cat 物件的 Meow() 函式
#0006     ...
#0007     // 離開函式時自動摧毀堆疊內的 cat 物件
#0008 }
```

不過，在 Object Pascal 中，以上兩種物件的建立方式都是不被容許的。*TObject* 只容許物件以這種方式建立：

```
#0001 procedure foo;
#0002 var
#0003     cat: TCat; // Object Pascal 中的物件皆是參考 (reference) 型別
#0004 begin
#0005     cat := TCat.Create; // 在累堆中建立 TCat 類別的 cat 物件，取得位址
#0006
```

```
#0007   cat.Meow; // 相當於 C++ 的 cat->Meow() 呼叫，記住，cat 其實是指標
#0008   ...
#0009   // 離開函式後，會遺失目前 cat 指向的物件，找不回來，所以要記得先摧毀它
#0010   end;
```

建立在累堆中的方式比較麻煩，因為我們一定要主動呼叫建構函式¹³來建立物件。*TObject::Create*建構函式的任務是：配置物件記憶體、初始化物件資料，最後傳回物件位址。

因為 *TObject* 的此特性，就算在 C++Builder 裡，由於 VCL 類別皆是 *TObject* 的後代，所以一樣帶有此特性。因此，VCL 物件的產生不能像一般的 C++ 物件一樣靈活：

```
#0001 // 在資料節區中建立 TCat 物件
#0002 TCat g_cat;
#0003 // 想在資料節區建立 TButton 物件，不行，編譯無法通過
#0004 TButton g_button;
#0005
#0006 void foo(void)
#0007 {
#0008   // 在堆疊中建立 TCat 物件，進入 foo 時才建立
#0009   TCat cat;
#0010   // 想在堆疊建立 TButton 物件，不行，編譯無法通過
#0011   TButton button;
#0012
#0013   // 在累堆中建立 TCat 物件
#0014   TCat* h_cat = new TCat;
#0015   // 在累堆中建立 TButton 物件，沒有問題
#0016   // 建立 VCL 物件只有此方法可用
#0017   TButton* h_button = new TButton(NULL);
#0018 }
```

¹³ 也因此，Object Pascal 不像 C++ 有 default constructor 這種機制。

類別資訊及 RTTI 支援

```

TObject = class
...
class function ClassName: ShortString;
class function ClassNameIs(const Name: string): Boolean;
class function ClassParent: TClass;
class function ClassInfo: Pointer;
end;

```

上面列出 *TObject* 類別的四個成員函式，前三個函式提供類別名稱及父類別資訊，這些資訊由編譯器的虛擬方法表格（virtual method table）而來。第四個函式 *ClassInfo* 提供一個指標，指向的是此類別的 RTTI。

RTTI 包含非常多的資訊，主要的目的是，讓程式在執行時期也能得知資料型態及類別的資訊，這些資訊十分詳細地記錄資料型態及類別的所有細節，幾乎包括所有我們可從程式碼得知的資訊。

舉例來說，*TForm::FormStyle* 屬性為 *TFormStyle* 列舉型態，它的宣告如下：

```

enum TFormStyle {fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop};

```

在程式碼中，我們能夠任意使用以上的 *fsNormal*、*fsMDIChild*、*fsMDIForm*、*fsStayOnTop* 等名稱來表示 *TFormStyle* 變數的實際值，不過在程式執行時，所有列舉型態的值將視為整數型態來處理（理由是節省記憶體空間，而且運算快速），所以只知道值，無法取得這些值的宣告名稱，我們能做的只有比較、遞增、遞減運算而已。簡單地說，沒有辦法從 *Form1->FormStyle* 屬性值得到“fsNormal”這個字串。

不過，有了 RTTI 的支援，編譯器及連結程式會通力合作，將 *TFormStyle* 列舉型態資訊一併存入執行檔內。如此一來，即使在執行時期，也可以取出 *TFormStyle* 型態變數值的宣告名稱：

```

#0001 void __fastcall TForm1::Button1Click(TObject *Sender)
#0002 {
#0003     // 取得 FormStyle 的屬性資訊

```

```
#0004   PPropInfo PropInfo = GetPropInfo((TTypeInfo*)ClassInfo(),
#0005     "FormStyle");
#0006   // *PropInfo->PropType 為 TFormStyle 的型別資訊
#0007   ShowMessage(GetEnumName(*PropInfo->PropType, (int)FormStyle));
#0008 }
```

GetPropInfo 是 *TypeInfo* 單元所提供的一道函式，根據類別的型別資訊（指向 RTTI 的指標，以 *ClassInfo* 函式取得）以及屬性名稱來取得屬性資訊（*PPropInfo* 結構指標）。*GetEnumName* 也同樣宣告於 *TypeInfo* 單元，用來取得列舉型態值的宣告名稱。上述程式的執行結果可能是“fsNormal”字串，也可能是“fsMDIChild”或其它字串，視當時 *FormStyle* 屬性值而定。

再舉一個例子。撰寫程式時，我們常常需要一次更改多個元件的同一個屬性。例如，將 form 上所有元件的 *Color* 屬性改為藍色、將所有元件的 *Font* 屬性的字型放大、將所有元件的 *OnClick* 事件指派到某個事件處理函式等等。但是，並不是所有元件都有 *Color* 屬性、也不是所有元件都有 *Font* 屬性及 *OnClick* 事件，所以我們必須在程式執行時進行檢查，看看每個元件是否擁有某個特定的屬性。

怎麼做？總不能叫程式在執行時查閱說明文件、閱讀 VCL 原始碼，那是人類才能做的事。最優雅的作法是藉由 RTTI 的支援，來判定物件是否擁有某特定屬性，而且還可以直接更改該屬性的值。

下列程式做的正是這件事情：尋訪 form 上所有元件，檢查每個元件是否擁有 *Color* 屬性，如果有的話，就將它設為紅色：

```
#0001 void __fastcall TForm1::Button2Click(TObject *Sender)
#0002 {
#0003   PPropInfo PropInfo;
#0004
#0005   // 尋訪 form 上的所有元件
#0006   for (int i = 0; i < ComponentCount; i++) {
#0007     // 檢查 Components[i] 是否擁有 Color 屬性
#0008     PropInfo = GetPropInfo((TTypeInfo*)Components[i]->ClassInfo(),
#0009       "Color");
#0010     // 如果有的話，就設定為紅色
#0011     if (PropInfo) SetOrdProp(Components[i], PropInfo, clRed);
#0012   }
#0013 }
```

如此一來，不論 form 上的元件有多少、種類為何，這段程式絕對可以達成目的。在這段程式中，我呼叫 `TObject::ClassInfo` 函式來取得元件的 RTTI，交給 `GetPropInfo` 函式，嘗試取得該元件的 `Color` 屬性，如果有的話，就傳回該屬性的屬性資訊（這也是 RTTI 機制的功勞），再利用此 `PPropInfo` 型態指標來設定屬性值。

Tips

上段程式其實不夠嚴謹，因為它只檢查名稱爲“Color”的屬性，而不管該屬性的資料型態。萬一哪天我設計出一個新元件，它也有一個名稱爲“Color”的屬性，只不過型態不是 `TColor`，而是 `string` 字串型態，那麼上段程式就會出問題囉！

所以應該再修改程式，除了檢查屬性名稱外，還要檢查屬性型態的名稱，確定是“TColor”型態後，才能安心地指定屬性值。

訊息處理支援

Object Pascal 語法直接支援訊息分派處理機制，這是 `TObject` 類別與編譯器合作的結果。我隨手從 VCL 原始碼取出一段：

```
TScrollingWinControl = class(TWinControl)
private
    ...
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

上述宣告的意思是，當 `TScrollingWinControl` 物件收到 `WM_SIZE` 視窗訊息時，就執行 `WMSize` 函式；另外，當 `TScrollingWinControl` 物件收到 `WM_PAINT` 視窗訊息時，就執行 `WMPaint` 函式。

這些行爲的成立關鍵有二：首先，編譯器遇到 `message` 保留字時，會將此敘述指定的訊息編號（事實上是四位元組正整數）及對應的函式位址記錄下來¹⁴；其次，由 `TObject` 類別

¹⁴ 記錄在類別的動態方法表格（dynamic method table）內。

提供 *Dispatch* 函式來進行訊息的分派動作：

```
TObject = class
...
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
end;
```

呼叫 *Dispatch* 函式時，傳入訊息結構（此訊息結構型態可以任意自訂，唯一的要求是：訊息結構的第一個欄位必須是四個位元組的正整數，代表訊息編號），*Dispatch* 函式就會逐一尋找類別的訊息處理函式，看看是否有人登記想要處理此訊息。如果有，就交給該訊息處理函式去做；如果找不到，就呼叫 *DefaultHandler* 函式，丟給 *DefaultHandler* 函式處理。

上述的 *message* 保留字是 Object Pascal 的特權，遵從 ANSI C++ 標準的 C++ 語言不能這麼做。於是 C++Builder 提供三道巨集，一塊合作，實作出類似 Object Pascal *message* 保留字的訊息處理機制，分別是 *BEGIN_MESSAGE_MAP*、*VCL_MESSAGE_HANDLER*、以及 *END_MESSAGE_MAP*。它們的使用及實作方式請參考下述範例以及線上說明的「*BEGIN_MESSAGE_MAP* macro」主題。

下列這段程式示範的是，任何 VCL 物件都支援訊息分派處理動作：

```
#0001 // 定義自己的訊息結構
#0002 typedef struct {
#0003     Cardinal Msg;
#0004     AnsiString WowStr;
#0005 } TWowMessage;
#0006
#0007 class TDog : public TObject {
#0008 private:
#0009     // 收到訊息 100 時，會呼叫此函式
#0010     void Wow(TWowMessage& Message);
#0011 public:
#0012     BEGIN_MESSAGE_MAP
#0013         VCL_MESSAGE_HANDLER(100, TWowMessage, Wow);
#0014     END_MESSAGE_MAP(TObject);
#0015 };
#0016
#0017 void TDog::Wow(TWowMessage& Message)
```

```
#0018 {
#0019 // 根據訊息結構的資料，叫一聲
#0020 ShowMessage(Message.WowStr);
#0021 }
#0022
#0023 void __fastcall TForm1::Button1Click(TObject *Sender)
#0024 {
#0025 TDog* Dog = new TDog; // 繼承 TObject，一定要建立在 heap
#0026 try {
#0027     TWowMessage Msg;
#0028
#0029     // 先指定訊息結構內容
#0030     Msg.Msg = 100;
#0031     Msg.WowStr = "Ouch !!";
#0032
#0033     // 分派此訊息
#0034     Dog->Dispatch(&Msg);
#0035 } __finally {
#0036     delete Dog;
#0037 }
#0038 }
```

Dispatch 函式可接受任何訊息結構，只要：

1. 訊息處理函式也接受同樣的資料結構。
2. 訊息結構的第一個欄位必須是代表訊息編號的四個位元組正整數。

因此，雖然 *TObject* 的訊息分派支援主要是為 Windows 的視窗訊息處理及 VCL 內部的元件訊息處理而設計，但具有高度彈性，可方便地應用在其它場合中。

TPersistent

TPersistent 類別最主要的功能是，能被 *TFiler* 物件永續機制類別讀寫。這是因為 *TFiler* 類別特別指明它只想處理 *TPersistent* 物件的結果，所以可說此能力是被動地產生，而非 *TPersistent* 類別本身的建樹。

物件複製機制

TPersistent 類別的另一個貢獻是，提供物件複製機制，但並不提供物件複製時的真正動作。它的貢獻在於宣告以下這兩個函式：

```
TPersistent = class(TObject)
protected
    ...
    procedure AssignTo(Dest: TPersistent); virtual;
public
    ...
    procedure Assign(Source: TPersistent); virtual;
end;
```

這二個函式，一個主動將物件本身的內容指派給別人，一個是讓別人將它的內容指派給自己。*TPersistent::Assign* 函式的動作十分簡單，它只是呼叫對方的 *AssignTo* 函式，將物件複製工作這個燙手山芋丟給對方：

```
procedure TPersistent.Assign(Source: TPersistent);
begin
    if Source <> nil then Source.AssignTo(Self) else AssignError(nil);
end;
```

當任何類別希望擁有指派物件的能力時，此類別至少必須改寫 *Assign* 或 *AssignTo* 函式其中之一，實際撰寫複製物件內容的程式碼才行。*TPersistent* 類別只提供 *Assign* 及 *AssignTo* 函式，對於實際的物件複製動作，完全幫不上忙。

RTTI 資訊的產生

第三個不太明顯的貢獻是，編譯器會為 *TPersistent* 類別的所有子類別產生 RTTI。我想這就是為什麼 *TPersistent* 類別獨立存在的最大原因（其實可把物件永續機制支援及物件指派機制一併納入 *TObject* 類別）：並不是每個類別都需要 RTTI 的支援，若產生不必要的 RTTI，只會浪費執行檔空間，徒增檔案大小而已。在 C++Builder 中，RTTI 最主要的貢獻是：

- 設計時期，在整合環境中，讓 Form Designer、物件檢視器、元件盤等等 RAD 設計工具順利運作。

- 執行時期，與 VCL 的物件永續機制配合，從資源區段讀取 form 的描述資料，動態建立 form 及其上所有元件，重現設計時期的狀態。

所以，只要是 VCL 元件以及它們所包含／使用的屬性資料型態、類別等等，通通需要 RTTI 的支援；若沒有 RTTI，C++Builder 根本不可能成為 RAD 開發工具。

此處的關鍵是 { $\$M$ } 編譯指示。在 Classes 單元中，*TPersistent* 類別宣告的前後分別以 { $\$M+$ } 及 { $\$M-$ } 編譯指示含括，{ $\$M$ } 編譯指示的影響範圍是類別本身及它的子類別，因此，只要是 *TPersistent* 的子類別，通通擁有 RTTI。

```
{ TPersistent abstract class }

{ $\$M+$ }
  TPersistent = class(TObject)
  ...
  end;
{ $\$M-$ }
```

可以做個簡單的實驗：試著宣告兩個新類別，分別以 *TObject* 及 *TPersistent* 類別為父類別，然後在程式執行時，檢驗它們的 *ClassInfo* 函式傳回值。可以發現，直接由 *TObject* 繼承而來的新類別的 *ClassInfo* 函式傳回 *NULL*，表示它不具有 RTTI；但由 *TPersistent* 繼承的新類別的 *ClassInfo* 函式將會傳回指向 RTTI 的非 *NULL* 指標，這就是 { $\$M$ } 編譯指示的效果。

TComponent

TComponent 類別以下，直接或間接繼承自 *TComponent* 的類別，就叫做元件。

TComponent 類別的特性是：

- 能夠安裝到 C++Builder 整合環境，出現在元件盤上，並且可被拖曳至 Form Designer 上頭操作。
- 具備擁有其它元件的能力。

- 提供 *Name* 屬性，所以 VCL 元件有所謂的「名稱」，便於設計時期的操作。
- 能將 ActiveX Control 或 COM 物件包裹起來，成為 VCL 元件。

控制項類別

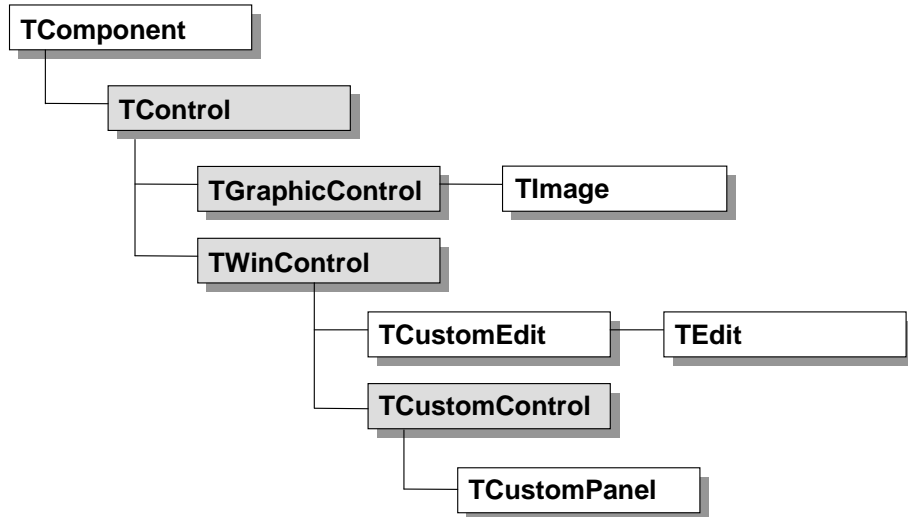


圖 2-16 / VCL 的重要控制項類別階層架構

上圖列出 VCL 的重要控制項類別。稱它們為「控制項」類別的原因是，它們的類別名稱全都以“Control”結尾，且在執行時期都可以顯示在畫面上。它們都是 *TComponent* 的後代，所以通通都是元件，只是特性用途各有不同。

表 2-17 / VCL 的重要控制項類別

類別	父類別	宣告單元	貢獻
<i>TControl</i>	<i>TComponent</i>	Controls	所有 VCL 可視元件的始祖
<i>TWinControl</i>	<i>TControl</i>	Controls	所有 VCL 視窗元件的始祖
<i>TGraphicControl</i>	<i>TControl</i>	Controls	自製圖形元件的始祖
<i>TCustomControl</i>	<i>TWinControl</i>	Controls	自製視窗元件的始祖

TControl

TControl 是所有可視元件的始祖，程式執行時，使用者能夠看到並且操作它們。

TControl 類別的特性為：

- 提供重繪本身的能力以及其它與顯示能力相關的函式及屬性。
- 提供 VCL 元件訊息處理機制。
- 擁有元件拖曳、置放或嵌入其它 VCL 元件的能力。

TWinControl

TWinControl 是所有視窗元件的始祖，它們不只是可視元件，因為它們本身會建立一個視窗，擁有視窗函式，在 Windows 標準的視窗訊息驅動架構下運作。

TWinControl 類別的重要性為：

- 擁有視窗、視窗 handle 及視窗函式。
- 能夠擁有鍵盤輸入焦點。
- 能夠包含 (contains) 其它 *TControl* 元件，請注意，「包含」與「擁有」並不同。

TGraphicControl

TGraphicControl 是所有自製圖形元件的始祖，它們是可視元件，不過與 *TWinControl* 元件不同，這些元件不會建立視窗，因此無法擁有鍵盤輸入焦點。

TGraphicControl 的特性、行為與它的父類別 *TControl* 完全相同，只不過多了兩點加強：

- 攔截 *WM_PAINT* 訊息，提供 *Paint* 虛擬函式供後代類別改寫。
- 提供 *Canvas* 屬性供後代類別使用。

因此，*TGraphicControl* 類別的主要功用是，擔任自製圖形元件的父類別，讓元件撰寫者繼承使用。由它衍生新的元件時，只要改寫 *Paint* 函式，就可在其中繪製元件外觀，創造出新的 *non-windowed* 視覺元件。

TCustomControl

TCustomControl 是所有自製視窗元件的始祖，它們具有視窗，擁有視窗函式，在 Windows 標準的視窗訊息驅動架構下運作。

如同 *TGraphicControl* 與 *TControl* 的關係，*TCustomControl* 的特性、行為與它的父類別 *TWinControl* 完全相同，只不過多了兩點加強：

- 攔截 *WM_PAINT* 訊息，提供 *Paint* 虛擬函式供後代類別改寫。
- 提供 *Canvas* 屬性供後代類別使用。

因此，*TCustomControl* 類別的主要功用是，擔任自製視窗元件的父類別，讓元件撰寫者繼承使用。由它衍生新的元件時，只要改寫 *Paint* 函式，就可在其中繪製元件外觀，創造出新的 *windowed* 視覺元件。

以上這四個控制項類別，其實這樣分類會更清楚：

- 具有代表性，提供主要功能：
 - 不具視窗：*TControl*
 - 擁有視窗：*TWinControl*
- 供元件撰寫者繼承使用：
 - 不具視窗：*TGraphicControl*
 - 擁有視窗：*TCustomControl*

最後舉個範例程式請你親眼看看所謂的圖形元件（不具視窗）與視窗元件（擁有視窗）到底差在哪兒。由於 VCL 包裝得漂亮的緣故，雖然這兩類元件骨子裡運作的方式截然不同，但是不管是用起來、看起來都一模一樣，難以分別。不過，只要藉由適當的輔助工具來觀察，就可輕易地讓它們現出原形。

下圖是 *Form1*，我在上頭置放五個不同的可視元件，由上而下分別是 *TLabel*、*TShape*、*TStaticText*、*TButton* 及 *TPanel* 元件。它們的設計及執行時期外觀如下：

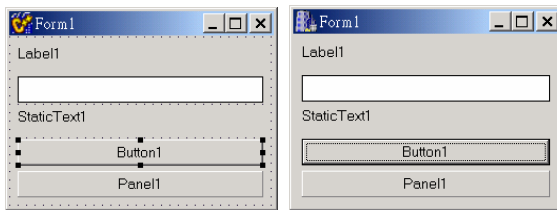


圖 2-18 / 有五個不同的元件

光從元件外觀是絕對無法判別某個元件是否具有視窗的，別說是書上圖案太小看不出來，就算視窗活生生地待在我們面前，也沒有人能夠光憑外觀分辨元件是否擁有視窗。不過沒關係，聰明的小孩愛玩玩具，聰明的程式設計師愛用工具，我用 **SoftICE** 來觀察 *Form1* 視窗：

```
:hwnd -c project1
```

Handle	Class	WinProc	TID	Module
03071A	IME	77E952BA	ED	00000000
030724	TForm1	00254477	ED	00010100
050732	TStaticText	011B0FAE	ED	00000000
050730	TPanel	011B0FBB	ED	00010000
03072E	TButton	011B0FC8	ED	00000000
030718	TApplication	011B0FEF	ED	0100:0000

嘿嘿，你也見到了，*TForm1* 視窗上只有三個視窗，它們的視窗類別名稱分別為 *TStaticText*、*TPanel*、*TButton*。這表示 *Form1* 上的五個元件中，*StaticText1*、*Panel1* 及 *Button1* 各自擁有一個視窗，而 *Label1* 及 *Shape1* 都是不具視窗的元件。請對照下列五個元件的類別階層圖來驗證：是不是具有視窗的元件都是 *TWinControl* 後代？是不是不具有視窗的元件都是 *TGraphicControl* 後代？

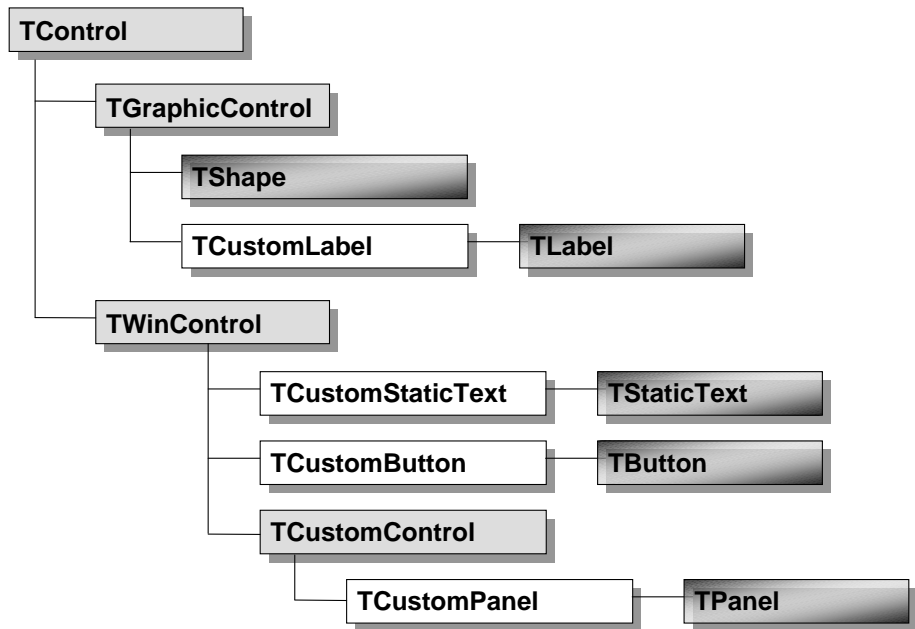


圖 2-19 / 範例程式中五個元件的類別階層圖

所以，簡單地說，作業系統只能感覺到具有視窗的元件，對於不具視窗的元件，作業系統完全不曉得，那是 VCL 自己的事情。對於作業系統來說，VCL 的 *TButton* 元件是一個真正的視窗，視窗類別名稱叫做 *TButton*；而 VCL 的 *TLabel* 元件呢？因為它不具視窗，所以不在作業系統的掌控範圍內，作業系統只能說：「啥？啥是 *TLabel*？」。

視窗是如何繪製在畫面上的呢？因為作業系統瞭解它的存在，所以當它有重繪的必要時（例如出現在畫面上，或本來覆蓋它的視窗移開），視窗會收到 *WM_PAINT* 訊息，視窗函式就會將元件外觀繪製出來。

但是對於不具有視窗的圖形元件呢？它們沒有視窗，所以系統不會送給它們視窗訊息，那它們在什麼時機才能重繪元件外觀呢？關鍵在於：

任何不具視窗的 *TControl* 元件都必須以一個 *TWinControl* 元件為 parent 元件，也就是被一個 *TWinControl* 元件包含，它才能夠出現在畫面上。

既然作業系統會適時告知視窗重繪，那麼那些不具視窗的元件就可由包含它的視窗元件負責，當視窗元件的重繪區域包含某個不具視窗的元件時，就告訴該元件：請重繪元件本身。下圖展示此機制的流程：

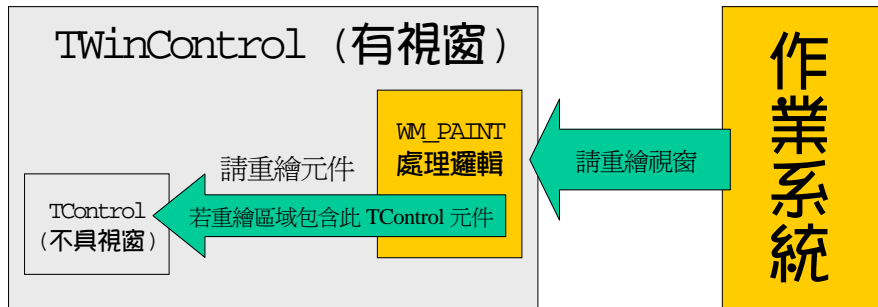


圖 2-20 / TWinControl 元件必須負責其上圖形元件的繪製

Controls 單元中，`TWinControl::PaintControls` 函式做的就是這件事。當其上的元件也需要一併繪製時，它會呼叫該元件的 `Perform` 函式，傳送 `WM_PAINT` 訊息給它，請它重繪。

事實上，不只是 `WM_PAINT` 訊息，所有的滑鼠訊息也都必須仰賴 `TWinControl` 元件的幫忙，不具視窗的圖形元件才能得知滑鼠訊息，才能感應使用者的操作而做出回應。

程式運作類別

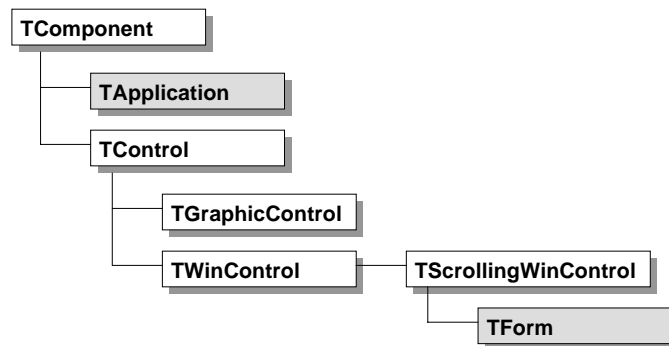


圖 2-21 / VCL 程式運作類別階層架構

這裡列出兩個類別，雖然它們並不是 VCL 元件架構的關鍵類別，但是它們是 VCL 程式運作的重要角色，也因為它們的存在，VCL 才能從 class library 升級為 application framework。

表 2-22 / VCL 程式運作類別

類別	父類別	宣告單元	貢獻
<i>TForm</i>	<i>TCustomForm</i>	Forms	應用程式視窗
<i>TApplication</i>	<i>TComponent</i>	Forms	訊息擷取及分派

TForm

TForm 是應用程式視窗的包裝，由於巧妙地設計，VCL 的 *TForm* 類別一手包辦 SDK 中的 window 及 dialog 兩種應用程式視窗。

TForm 具有以下特性：

- C++Builder 整合環境的介面設計單位（C++Builder 5 新增 *TFrame* 元件，也能以 *TFrame* 為設計單位）。
- 能夠藉由屬性的更改，提供 MDI 使用者介面。
- 有兩種使用模式，分成「不含訊息迴圈」及「包含訊息迴圈」兩種使用模式。

TApplication

TApplication 是 VCL 應用程式的運作關鍵，因為它負責主執行緒的視窗訊息擷取／分派動作。

TApplication 有如下特性：

- 設計時期不會出現，程式運行後自動產生。

- 掌握程式的運作及結束時機¹⁵。
- 當程式的 main form 關閉時，結束程式的運作。

RAD 支援類別

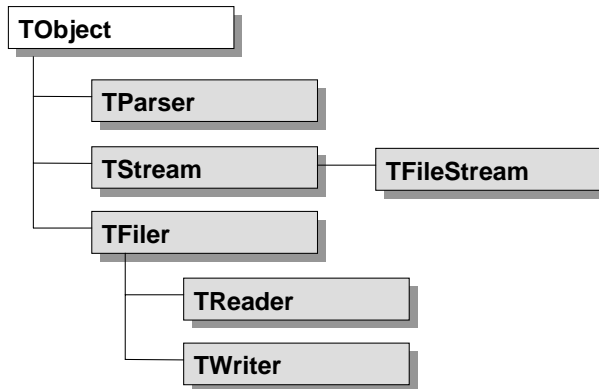


圖 2-23 / VCL RAD 支援類別階層架構

這些是實作物件永續機制的類別，由於它們的輔助，我們才能在 C++Builder 整合環境的 Form Designer 上頭，隨心所欲地以所見即所得方式設計 form 的介面，而後在程式執行時直接取出整個 form 的資訊及狀態，開始使用。

表 2-24 / VCL RAD 支援類別

類別	父類別	宣告單元	貢獻
<i>TStream</i>	<i>TObject</i>	Classes	所有資料流類別的始祖
<i>TFileStream</i>	<i>TStream</i>	Classes	檔案資料流
<i>TFileer</i>	<i>TObject</i>	Classes	具有讀寫元件能力的類別的始祖
<i>TReader</i>	<i>TFileer</i>	Classes	將元件由資料流讀出

¹⁵ 若要脫離 *Application* 物件的控制，自行建立視窗迴圈也行，此時 *Application* 物件就不再具有影響力，請參看第七章的 XEssay 螢幕保護程式範例。

<i>TWriter</i>	<i>TFile</i>	Classes	將元件寫入資料流
<i>TParser</i>	<i>TObject</i>	Classes	將以文字型式表示的元件，轉換成以二進位型式來表示

VCL 重要類別至此介紹完畢，若你是第一次接觸這些類別，雖然我講得煞有其事，但感覺應該十分抽象，畢竟這些都不是平常可在整合環境的元件盤、Form Designer 等等設計工具上能實際接觸的咚咚。

沒關係，先對它們有基礎的認識及印象就好，基本心法的通徹是絕對無法一蹴可幾的。平日的 C++Builder 程式開發中，或多或少一定有接觸及使用這些類別的機會，屆時就請勿摸魚矇混過去，把握時機好好瞭解這批 VCL 核心份子，它們是開啓 VCL 寶庫的鑰匙。

第二篇

作業系統



第三章

控制你的控制台

小小的腦袋瓜兒內有許多創意，
小小的控制台內也裝著許多有趣的玩意兒，
讓我告訴你控制台要怎麼玩！



就讀高中時，還是那個開機得先進 DOS，打 WIN 三個字母才能進入 Windows 3.1 的史前時代。每回打開控制台，硬碟就得喀啦喀啦地亂攪一陣，狀態列閃動過十數個檔名，一切就緒後，才能開始操作。即使如此，對於我這種有事沒事就喜歡換桌布，上百個螢幕保護程式天天更替的人來說，控制台還是除了「踩地雷」、「接龍」、「小畫家」等程式外最好玩有趣的「遊戲」。

Windows 95 出現後，小小的控制台裝入更多內容，變得更有意思了！新的「新增移除程式」、「網路」、「安裝新的硬體」、「PLUS!」等元件，讓「控制台」真的成爲名符其實「控制」整部系統的基地「台」。對於這麼一個看起來似乎屬於系統元件的程式，你會不會動過它的腦筋，想要改變它的行爲，或者至少新增、移除、改變其中的元件呢？

我就愛動這種怪腦筋，觀察程式，看看它有什麼特性，程式的哪些行爲是內定的，而哪些行爲是可供外界修改的，進而利用各種工具程式及程式開發工具來「玩弄」軟體。

好，現在即使手上沒有任何工具，就由平日把玩軟體的經驗中，好像也可以看出一些端倪：

- 安裝 Microsoft Office 後，控制台內悄悄地多了「Find Fast」及「ODBC」等元件。
- 安裝 Internet Explorer 時，會偷偷幫你塞進「Internet」元件。
- 安裝 Borland C++ Builder 後，「BDE¹ Administrator」也會進駐控制台。
- 安裝某些顯示卡驅動程式後，「顯示器」元件內也會多出一兩個此顯示卡專用的設定頁面。

除此之外，還有許多許多程式也會提供自己的控制台設定元件，也有程式本身就以控制台元件的形式存在的，例如我的電腦中，就可以找到設定使用環境的「Tweak UI」元件、用來更新 Symantec 軟體的「Symantec LiveUpdate」元件、檢視 DirectX 安裝情形及設定功能的「DirectX」元件、設定 QuickTime 軟硬體的「QuickTime」元件等等。來來來，

¹ BDE：Borland Database Engine 的簡稱，Borland 的許多產品皆以 BDE 做爲資料庫引擎。

比較一下我的控制台和你的控制台是不是差別不小呢？

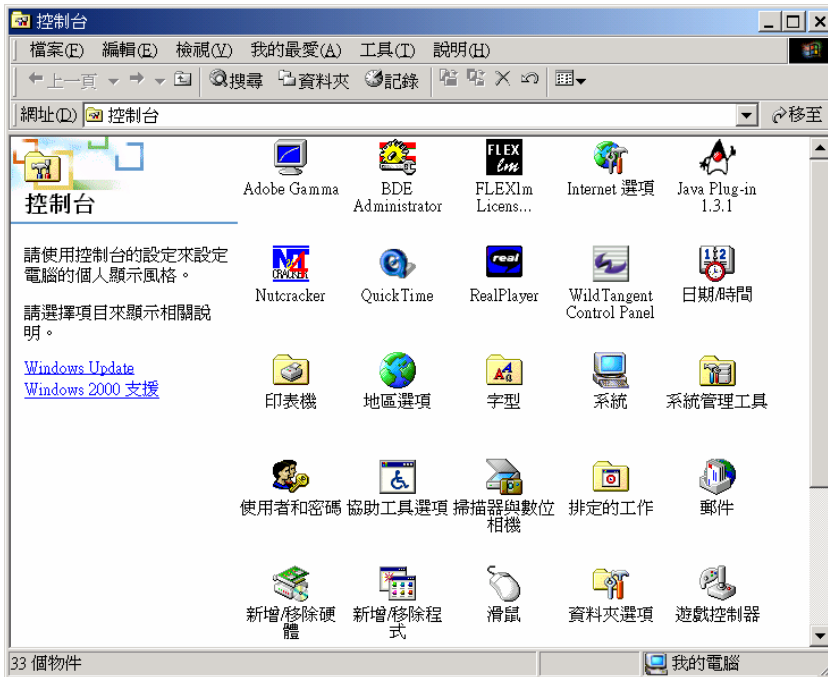


圖 3-1 / 我的控制台（系統為 Windows 2000）

這現象告訴我們一件事：「控制台裡頭的元件可讓使用者自行安裝、移除，且具高度自訂性」。

這正是今天的主題－控制台及其中的控制台元件，且讓我們由淺入深、徹徹底底地探討這個鮮少受人注目的主題吧！

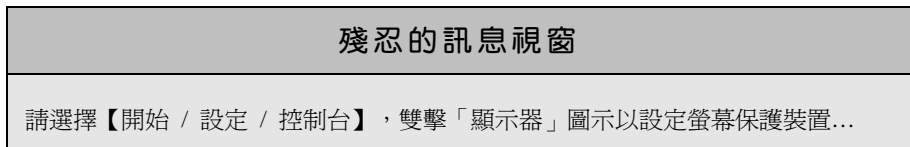
控制台觀測站

控制台觀測站提供三項服務：控制台的呼叫方式、CPL 檔的真實身份及行為分析。

呼叫呼叫，聽到請回答！

通常，控制台的使用方法是使用者由【開始 / 設定 / 控制台】選項叫出，再雙擊其中的元件圖示。但有時候，我們會希望使用者手動透過控制台進行某些程式碼無法達成或實作上相當麻煩的動作。比如說，有些驅動程式只要由使用者執行一個安裝程式，透過INF設定檔，即可完成所有的登錄設定及檔案複製動作；但也有許多驅動程式廠商較懶，要請使用者打開控制台的「加入新的硬體」、「數據機」、「多媒體」等元件，自行加入新的數據機、顯示卡等等硬體的驅動程式。除了驅動程式的安裝動作外，也有不少程式需要控制台元件的支援，例如新字型的安裝、服務的控制、新增／移除程式等等。

對程式員來說，最簡單的方法就是殘忍地顯示一個訊息視窗給使用者：



這方式雖然可行，不過...一來不夠人性化，二來對於初學者過於殘忍，三來這個程式也太拙了點！比較好的做法當然是直接開啓控制台元件供使用者直接操作囉！

Tips

事實上，控制台元件能夠做到的事，我們利用程式應該也都能做到。因此，最好的做法是不必藉由使用者的輔助，直接以程式完成所有動作。

其實做法並不難，你可以馬上試試。請在命令列下，或選擇【開始 / 執行】，鍵入「CONTROL」，再按下Enter，控制台立刻就蹦出來了。位於系統目錄²下的CONTROL.EXE即是控制台的主程式。

² 系統目錄指的是GetSystemDirectory API函式所取得的目錄，一般而言為Windows目錄下的SYSTEM或SYSTEM32目錄。

呼叫此 CONTROL 程式時，可以使用如下參數：

```
control [CPL 檔案][,元件編號][,頁碼]
```

例如：

- 變更時間、日期及時區資料
`control timedate.cpl,@0,1`
- 設定鍵盤速度、輸入法及鍵盤類型
`control main.cpl,@1`

注意每個參數及逗號之間不能留有任何空格，否則什麼都不會出現，連錯誤訊息也懶得給。

至於有哪些 CPL 檔案可用，只要到你的系統目錄下將所有的副檔名為 CPL 的檔案列出即可得知，其中大部分都可以根據檔名猜出它大概包含哪些元件。我們可由第二個參數「元件編號」的存在得知，每個 CPL 檔內不一定只含有一個控制台元件。至於每個 CPL 檔案中提供哪些元件呢？先用猜的吧，稍後我們將會撰寫自己的控制台，即可清楚地檢視每個 CPL 檔案所包含的元件。

在嘗試使用 CONTROL.EXE 時，我發現兩個小臭蟲：

- 「新增／移除程式」這個元件的頁次編號從 1 開始，而其它元件的頁次編號都是以 0 開始的。因此，下面這兩道指令：
`control appwiz.cpl,,1`
`control desk.cpl,,1`
相同的頁次參數，「新增／移除程式」元件會切換到第一頁，而「顯示器」元件卻切換到第二頁。
- 若元件的某頁面是由 Control Panel Extension 技術安插進來的，例如「顯示器內容」對話盒的「Plus!」頁面，那麼利用頁次編號參數將無法指向它，這表示在控制台元件中頁碼及頁面對照已經寫死在程式中，而不是根據 *TabControl* 控制項的頁碼來切換。

另類呼叫法

前面提到執行 CONTROL.EXE 的方法雖然十分簡單，但微軟的文件卻鼓勵我們使用另一種方式來啟動控制台元件：

```
rundll32.exe shell32.dll,Control_RunDLL [CPL 檔案][,元件編號][,頁碼]
```

後三個參數的順序及意義與 CONTROL.EXE 一模一樣。另外要注意的是，*Control_RunDLL* 這個字的大小寫不能有任何錯誤，如果不小心打錯了，就會出現「SHELL32.DLL 發生錯誤，遺失項目 *Control_RunDLL*」這樣的錯誤訊息來，你知道為什麼嗎？

如果你手邊正好有DUMPBIN、TDUMP或DUMPEXE³等檔案傾印工具，用它來觀察SHELL32.DLL的exports section，可以在其中發現剛剛所叫用的*Control_RunDLL* 函式，好巧，不是嗎？:P

```
C:\WINNT\SYSTEM32>dumpbin /exports shell32.dll
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file shell32.dll

File Type: DLL

    Section contains the following Exports for SHELL32.dll

    ordinal hint  name
           3     0  CheckEscapesA (0007DAD4)
           6     1  CheckEscapesW (0007DA2A)
           7     2  CommandLineToArgvW (0007DD4A)
           8     3  Control_FillCache_RunDLL (000320EB)
          12     4  Control_FillCache_RunDLLA (000320EB)
```

³ DUMPBIN來自Win32 SDK及Visual C++；TDUMP來自Borland；DUMPEXE來自Symantec C++。

14	5	Control_FillCache_RunDLLW (000321C7)
22	6	Control_RunDLL (00031FC4)
41	7	Control_RunDLLA (00031FC4)
42	8	Control_RunDLLW (00023D02)
44	9	DllGetClassObject (00009FC0)

正因為我們叫用的是SHELL32.DLL所匯出的*Control_RunDLL*函式，而DLL的函式名稱是大小寫相異的，所以一字也差不得。你可以在列表中看到，SHELL32.DLL提供六個與控制台相關的函式，但事實上只有兩種功能，每種功能有三道函式，分別是Unicode版本、ASCII版本及預設版本⁴。除了現在討論的*Control_RunDLL*函式，另一道名稱更長的*Control_FillCache_RunDLL*函式將會在後頭討論。

RUNDLL32.EXE 是個很有趣的小工具，使用方法是：

```
rundll32.exe DLL 檔名, DLL 函式名稱 函式參數[, 函式參數]...
```

將 DLL 檔名、欲呼叫的函式名稱及參數傳入後，RUNDLL32 就會為你載入 DLL，呼叫指定的函式並且傳入指定的參數。雖然簡單，任何稍具經驗的程式員都可以馬上自己寫一個功能相同的程式出來，但是意義非凡。它的好處是：

- 可讓我們在命令列下達指令就載入 DLL 並呼叫函式。
- 即使是 16 bit 的行程也能方便地透過RUNDLL32 程式使用 32 bit 的DLL檔 – Windows 95 的控制台就是一例。它本身是 16 bit 程式，但卻可以同時使用 16 bit⁵ 及 32 bit 的CPL檔，而這就是RUNDLL32.EXE存在的最大理由。

曾不曾發現，包括「踩地雷」、「小畫家」等幾乎所有的附屬應用程式，它們的「關於」對話盒都長的一模一樣，擺明是由同個模子打造出來的。事實上，這個模子就擺在SHELL32.DLL的*ShellAboutA*⁶ 函式中，現在試著在命令列打入：

⁴ 並不是每個Win32 平臺皆提供預設版本，Windows 98 就是一例。

⁵ 在Windows NT/2000 中，控制台（CONTROL.EXE）改成 32 bit 的程式了。

⁶ 函式名稱最後頭的字母A，指此函式為ASCII版本。另外提供功能相同的*ShellAboutW*

```
rundll32.exe shell32.dll,ShellAboutA Hello, How do you do ?
```

只要沒打錯的話，螢幕上會立刻出現熟悉的關於對話盒，而且在「版權所有」的下一行，寫著就是我們所傳入的參數“Hello”。你會發現原來連一行程式都不用寫，竟然也可以使用 DLL。:P 當然囉，這不是正確用法，只是讓我們瞭解 RUNDLL32.EXE 的能力，偶爾拿來唬唬人，過過癮罷！

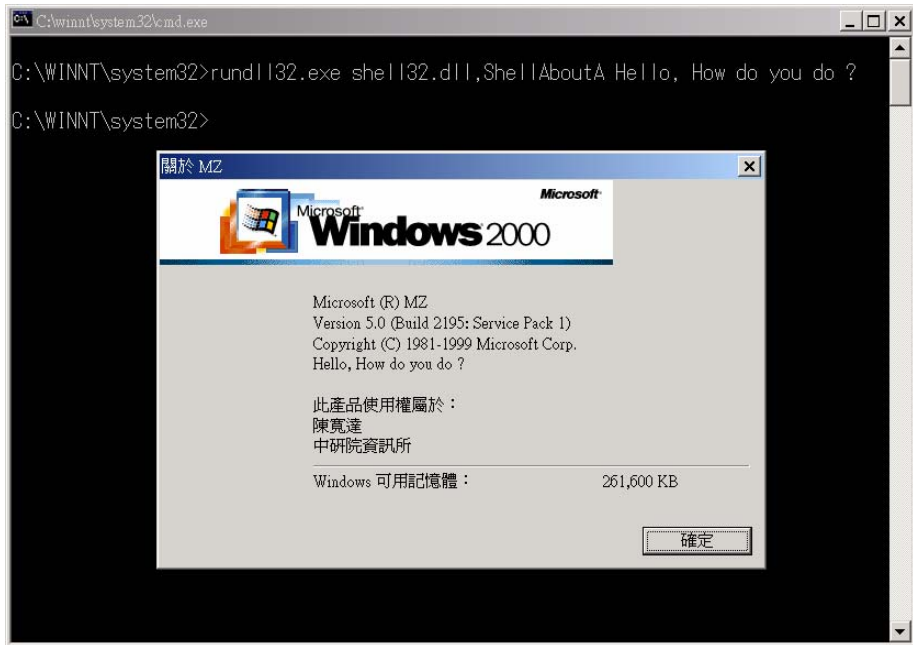


圖 3-2 / 透過 RUNDLL32 召喚出標準「關於對話盒」

做個小結，當你必須在自己的程式中叫用控制台元件時，如下撰寫即可：

```
WinExec("rundll32.exe shell32.dll,Control_RunDLL xx.cpl,yy", SW_NORMAL);
```

連檔案總管及控制台本身都是這樣叫用控制台元件的，所以趕快多多熟悉我們的新朋友 RUNDLL32.EXE 吧！

函式，為 wide character / Unicode 版本。

Tips

你也許好奇，我怎麼確定檔案總管及控制台也都是透過 RUNDLL32.EXE 來叫用控制台元件的？

這很簡單，請將系統目錄下的 RUNDLL32.EXE 暫時移至別處（千萬別砍掉呀！），然後在檔案總管或控制台中雙擊任一控制台元件，看到畫面上出現「無法找到檔案 'rundll32.exe'（或其中一個元件）...」錯誤訊息，這就是答案了。

CPL 檔的真實身份

讓我路邊隨手抓個 CPL 檔案來研究研究：

```
C:\WINNT\system32>dumpbin desk.cpl

Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file desk.cpl

File Type: DLL (由此得知 CPL 檔即 DLL 檔)

C:\WINNT\system32>dumpbin /exports desk.cpl

Dump of file desk.cpl

File Type: DLL

        Section contains the following Exports for Display.dll

        ordinal hint  name
            1     0  CPLApplet (00001000)
            2     1  DeskSetCurrentScheme (000086A8)
            ...

C:\WINNT\system32>dumpbin /exports joy.cpl

Dump of file joy.cpl
```

```
File Type: DLL

Section contains the following Exports for JOY.dll

ordinal hint name
          1 0 CPlApplet (00001117)
          2 1 ShowJoyCPL (00001018)
```

由上面的驗證動作，可歸納出下面兩點：

- CPL 檔案其實就是一個 DLL 檔，只是副檔名不同罷了！相同的情形也出現在副檔名為 386、DRV、OCX 的檔案身上。
- 一個可以正常運作的 CPL 檔一定會匯出 *CPlApplet* 這個函式（區分大小寫）。

簡而言之，CPL 檔就是至少匯出 *CPlApplet* 函式的 DLL 檔案，只是掛上 CPL 的副檔名而已。由此可見，*CPlApplet* 函式一定非常重要，是構成 CPL 檔的核心成員：

LONG APIENTRY CPlApplet(

HWND hwndCPL,
UINT uMsg,
LPARAM lParam1,
LPARAM lParam2

);

參數

hwndCPL 啟動控制台元件的應用程式視窗handle。

uMsg 由外界傳入的控制訊息。

lParam1 根據訊息種類而定的參數一。

lParam2 根據訊息種類而定的參數二。

回返值

根據訊息種類而定。

嚴格看來，*CPlApplet* 函式並不算是典型的回呼函式－因為它並沒有進行任何向別人註冊位址的動作。但是背後所代表的意義是一樣的：「嘿！我在這兒，有需要就找我」。雖

然少了註冊的動作，但它匯出一個眾所皆知的名稱—*CPIApplet*，讓所有想啟動控制台元件的程式透過此函式來取得服務，可說是另一種實作回呼函式的方式。

行為剖析

當控制台、SHELL32.DLL、或任何應用程式想要啟動此 CPL 檔所包含的控制台元件時，就會載入此 CPL 檔，取得 *CPIApplet* 函式位址然後呼叫它。而 *CPIApplet* 函式的任務是，根據 *uMsg* 訊息參數得知外界的請求並做出適當回應。

可能傳入的訊息種類有下列幾種：

表 3-3 / *CPIApplet* 函式必須處理的訊息

訊息	時機	意涵
<i>CPL_INIT</i>	CPL 檔載入後	供 CPL 檔有機會進行各控制台元件的資料結構、變數及配置記憶體等初始化及前置工作。
<i>CPL_GETCOUNT</i>	緊接在 <i>CPL_INIT</i> 取得 CPL 檔提供的控制台元件數目。訊息後	
<i>CPL_INQUIRE</i>	緊接在 <i>CPL_GETCOUNT</i> 訊息後	取得個別控制台元件的資訊。
<i>CPL_NEWINQUIRE</i>	緊接在 <i>CPL_GETCOUNT</i> 訊息後	取得個別控制台元件的資訊，作用與訊息 <i>CPL_INQUIRE</i> 相同，但應用場合不同，元件通常只須選擇兩者任一支援即可。
<i>CPL_DBLCLK</i>	啟動個別元件前	啟動某一控制台元件，通常的回應都是顯示 Modal 對話框供使用者設定。
<i>CPL_STOP</i>	關閉個別元件後	啟動中的控制台元件關閉後，控制台程式會送出此訊息進行個別元件的善後工作。
<i>CPL_EXIT</i>	釋放 CPL 檔前	控制台程式要釋放此 CPL 檔前會送出此訊息，提供資源釋放或其它善後工作的機會。

接下來的幾個小節，分別就這七道訊息一一詳述。

CPL_INIT

```
lParam1 : 未使用。  
lParam2 : 未使用。  
傳回值  : 成功傳回非零值；失敗傳回零。
```

CPL 程式第一個收到的訊息一定是 *CPL_INIT*，通常我們會利用此機會初始化所有控制台元件共享的資料結構及變數。

若初始化失敗，傳回零，這會使控制台不繼續傳送其它訊息給此 CPL 檔，而此 CPL 包含的所有控制台元件將不會出現在控制台內。如果真要這樣做，最好能利用訊息盒告知使用者錯誤原因，免得讓人搞不清楚為什麼控制台裡平白消失了幾個元件。

CPL_GETCOUNT

```
lParam1 : 未使用。  
lParam2 : 未使用。  
傳回值  : 傳回此 CPL 檔所支援的控制台元件數目。
```

繼 *CPL_INIT* 訊息後，緊接著會收到 *CPL_GETCOUNT* 訊息，此時必須傳回 CPL 檔所支援的控制台元件數目。

大部分的 CPL 檔都只包含一個元件，也有例外的：如 *MAIN.CPL* 就包含了「滑鼠」、「鍵盤」、「印表機」及「字型」等四個控制台元件。如果希望某個控制台元件能視情形出現或消失，這兒是一個切入點。傳回值設為欲顯示的元件數目 *N*，控制台就只會追問 *N* 個元件的資訊，其它不欲出現的元件就可以隱藏起來。例如，在我的電腦上沒有安裝搖桿的驅動程式，所以將 *CPL_GETCOUNT* 訊息傳遞給 *JOY.CPL* 時，它會傳回零，因此「搖桿」控制台元件就不會出現在我的控制台中。

CPL_INQUIRE

lParam1：元件編號，值為 0 ~ (CPL_GETCOUNT 傳回數目 - 1)。
 lParam2：指向 TCPLInfo 結構的指標。
 傳回值：成功傳回零；失敗傳回非零。

主菜端上桌囉，CPL_INQUIRE 訊息及下一個 CPL_NEWINQUIRE 訊息是最重要的兩個訊息，控制台就是利用這兩個訊息來取得每個元件的名稱、描述及圖示。參數一傳入元件編號，我們必須根據元件編號將對應的元件資訊填入參數二指向的 TCPLInfo 結構中。

TCPLInfo 結構定義如下：

```
typedef struct tagCPLINFO { // cpli
    int idIcon; // 指向元件圖示
    int idName; // 指向元件名稱
    int idInfo; // 指向元件描述
    LONG lData; // 使用者自訂參數
} CPLINFO;

typedef tagCPLINFO TCPLInfo;
```

idIcon、idName、idInfo 三個欄位所存放的內容是「資源代碼」（resource identifier），分別指向圖示、名稱及描述，我們必須分別呼叫 LoadString 及 LoadIcon API 函式，傳入資源代碼，從 CPL 檔的資源區段取得實際字串及圖示。

lData 欄位的型態為 LONG，為長度四個位元組的無號整數，不過也可轉型為指標變數來使用，你可以用它來指向任何一個結構，現在填入的 lData 值，會在控制台程式下次傳送 CPL_DBLCLK 及 CPL_STOP 訊息時由 lParam2 參數傳回供元件使用，你可以自由決定它的功用。

控制台取得每個元件的 TCPLInfo 結構後，會將它保存（cache）起來，日後除非 CPL 檔有更動，否則不會再次傳入 CPL_INQUIRE 訊息來重新取得，這也是為什麼 Windows 95 的控制台的開啓動作遠比 Windows 3.1 時代的開啓動作要快得多的原因。

有些時候我們會希望元件的名稱、描述及圖示會根據系統目前的狀態而變更，讓使用者

可以根據控制台中元件的圖示外觀輕易得知目前的狀況，例如當數據機連線中時，「撥號監視器」就可換個代表連線狀態的圖示，而平日又換回非連線狀態的圖示等等。對於有此項需求的屬性，當你收到 *CPL_INQUIRE* 要求取得元件資訊時，請在該欄位中填入 *CPL_DYNAMIC_RES* 值，表示此欄位不是固定值，於是控制台程式會再發出 *CPL_NEWINQUIRE* 訊息來探詢目前的屬性值。因此，若 *TCPLInfo* 結構的 *idIcon*、*idName*、*idInfo* 有任一欄位值為 *CPL_DYNAMIC_RES*，就必須再處理 *CPL_NEWINQUIRE* 訊息。

CPL_NEWINQUIRE

```
lParam1 : 元件編號，值為 0 ~ (CPL_GETCOUNT 傳回數目 - 1)。  
lParam2 : 指向 TNewCPLInfo 結構的指標。  
傳回值   : 成功傳回零；失敗傳回非零。
```

TNewCPLInfo 結構較 *TCPLInfo* 結構稍微複雜：

```
typedef struct tagNEWCPLINFO {  
    DWORD dwSize;           // 結構佔用長度  
    DWORD dwFlags;         // 旗標，目前未使用  
    DWORD dwHelpContext;   // 說明頁編號，目前未使用  
    LONG lData;            // 使用者自訂參數  
    HICON hIcon;           // 元件圖示  
    TCHAR szName[32];      // 元件名稱  
    TCHAR szInfo[64];     // 元件描述  
    TCHAR szHelpFile[128]; // 說明檔名，目前未使用  
} NEWCPLINFO;  
  
typedef tagNEWCPLINFO TNewCPLInfo;
```

szName、*szInfo*、*hIcon* 及 *lData* 等四個欄位與 *TCPLInfo* 結構中對應的四個欄位作用完全相同，唯一的差別是 *szName*、*szInfo*、*hIcon* 欄位不再是資源代碼，而是字串及 *HICON* 型態，這使得 *CPL* 程式可以動態地根據情況來產生及回傳元件的名稱、描述及圖示，不必事先將所有可能的值全放在資源區段中。除了這四個相同的欄位，其餘的欄位幾乎目前皆未使用，只有 *dwSize* 欄位例外，它指向整個結構的長度，請用 *sizeof* 運算子來取得。

CPL_NEWINQUIRE 與 *CPL_INQUIRE* 有何異同呢？由訊息名稱來看，很顯然的，

CPL_INQUIRE 出現較早，後來才有 *CPL_NEWINQUIRE* 訊息的出現。一般而言，若 Windows API 函式及視窗訊息有新舊版本之分，原則上，我們都傾向於使用新的函式及訊息。但這兒可就是個例外...

CPL_NEWINQUIRE 訊息使用的 *TNewCPLInfo* 結構雖然較完整，但美中不足的是，它的資料並不能為控制台保留／快取。從 Windows 95 及 Windows NT 4.0 後，控制台新增快取能力⁷—可將每個由 *CPL_INQUIRE* 訊息所取得 *TCPLInfo* 結構保存起來，這可以大幅增快開啓控制台的速度；但是不能保存 *TNewCPLInfo*，這也就是為什麼目前的控制台元件還是偏好 *CPL_INQUIRE* 的緣由。

經過觀察，快取區似乎是由檔案構成，因為我發現即使重新開機，控制台也不會重新發送 *CPL_INQUIRE* 訊息—表示快取資料依舊存在，除非把該 CPL 檔從系統目錄下移除，重新開啓、關閉控制台，再將 CPL 拷回系統目錄下，如此控制台才會再一次發送 *CPL_INQUIRE* 訊息以取得元件資訊。

那麼，控制台到底是何時發送 *CPL_INQUIRE* 訊息？又啥時傳遞 *CPL_NEWINQUIRE* 訊息呢？

- 若此 CPL 檔的控制台元件資訊尚未進入快取資料庫，則控制台列出元件時會送出 *CPL_INQUIRE* 來取得元件資訊。
- 雙擊控制台元件，送出 *CPL_DBLCLK* 訊息啓動它前，會先送出 *CPL_INQUIRE* 訊息。
- 只要送出 *CPL_INQUIRE* 取得元件資訊，則 *CPL_NEWINQUIRE* 訊息也會送出，只是兩者順序不一定，誰先誰後皆有可能。

正因為 *CPL_INQUIRE* 的資料結構具有可快取的特性，所以微軟文件註明著，除非必要，否則請盡量使用 *CPL_INQUIRE* 訊息來傳遞元件資訊。若在 *CPL_INQUIRE* 訊息的

⁷ 快取能力其實是由 SHELL32.DLL 所提供，可由它所匯出的 *Control_FillCacheRunDLL* 函式略知一二。

TCPLInfo 結構中沒有包含 *CPL_DYNAMIC_RES* 屬性值，則元件可以不必理會 *CPL_NEWINQUIRE* 訊息。

另一種情況，微軟文件內並沒有說明若上述兩個訊息皆傳回合法資料的話，控制台會優先使用哪一個呢？在好奇心的驅使下，我就做了這個十分無聊的實驗。實驗結果相當奇怪，元件名稱及描述會使用 *CPL_NEWINQUIRE* 所傳回的值，而圖示會使用 *CPL_INQUIRE* 所傳回的值。當然囉，這是定義及實作不夠嚴謹的情況下才會出現的怪現象，笑笑就好，不必認真討論。：)

CPL_DBLCLK

<p>lParam1：元件編號，值為 0 ~ (<i>CPL_GETCOUNT</i> 傳回數目 - 1)。 lParam2：由 <i>TCPLInfo</i> 或 <i>TNewCPLInfo</i> 所傳回的 <i>lData</i> 欄位值。 傳回值：成功傳回零；失敗傳回非零。</p>
--

當使用者雙擊控制台中的元件圖示或者呼叫 *SHELL32.DLL* 的 *Control_RunDLL* 函式來開啓元件時，就會觸發 *CPL_DBLCLK* 訊息。

通常對此訊息的回應是開啓一個對話框供使用者調整設定，例如「顯示器」元件提供的對話框就可供使用者設定螢幕保護程式、背景顏色、視窗外觀、桌布甚至螢幕的垂直掃描頻率等等；「字型」元件提供的對話框可讓你新增、移除、瀏覽字型等等。此訊息的處理動作可說是啓動元件能力的鑰匙。

當然囉，除了開啓對話框外，你也可以進行任何想做的動作—撥放歡迎音樂、秀一段動畫、啓動另一個程式，甚至提供小遊戲都行...儘管將它當成一般的應用程式來寫，唯一的不同是—不能使用正常模式的視窗，只能建立 *Modal* 對話框。建立正常模式視窗的結果是—視窗一閃即逝，什麼都還沒看到就結束啦！

Info

正常視窗會一閃即逝的原因是，控制台元件的執行時間，只容許在從 *CPLApplet* 函式收到 *CPL_DBLCLK* 訊息的那一刻起，直到此次函式回返為止。而函式回返之後，會接著收到 *CPL_STOP* 訊息，進行該控制台元件的善後工作。所以可以得知，收到 *CPL_DBLCLK* 訊息後，不可以立即返回，必須進入自己的訊息迴圈內進行處理，直到使用者設定完畢，將對話框關閉後才能離開。

因為 *TForm::ShowModal* 函式裡面有一個訊息迴圈，可以讓元件在收到 *CPL_DBLCLK* 訊息之後，不會馬上返回，所以 *Modal* 對話框可以用在這個地方。而 *Show* 函式只是將 *Visible* 屬性改成 *true*，然後就返回了，所以不適用於這個場合。

CPL_STOP

lParam1：元件編號，值為 0 ~ (*CPL_GETCOUNT* 傳回數目 - 1)。
lParam2：由 *TCPLInfo* 或 *TNewCPLInfo* 所取得的 *lData* 欄位值。
傳回值：成功傳回零；失敗傳回非零。

這個訊息提供機會讓個別元件進行善後工作，例如釋放 *GDI* 或核心物件等等。如果一切無誤，傳回零。

CPL_EXIT

lParam1：未使用。
lParam2：未使用。
傳回值：成功傳回零；失敗傳回非零。

這個訊息告訴 *CPL* 檔案說：「掰掰，有空再 *CALL* 你！」，接著就一腳把你踹出門外：~。你必須利用這最後喘息的機會做些釋放記憶體等善後工作。對於這個訊息的傳回值，因為沒有失敗的餘地，也一律設為零。

實作時間

Hello, World !!

自K&R⁸首開先例後，各種程式語言的「Hello, World !!」程式也一律出籠，於是，我們也不落人後，來寫個「Hello, World !!」控制台元件吧！:P

首先必須決定此元件是否支援動態元件資訊更新，亦即是否支援 *CPL_NEWINQUIRE* 訊息，我想，既然這是最簡單的第一個控制台元件，麻煩就免了，只支援 *CPL_INQUIRE* 訊息就好。

那麼，爲了提供元件資訊所需的字串及圖示資源，我們必須提供 RES 檔案與 CPL 檔案連結。於是我撰寫 HELLO.RC 檔如下：

HELLO.RC

```
STRINGTABLE
BEGIN
1 "Hello World !!"
2 "Hello, I am the simplest control panel applet in the world !!"
END
5 ICON "yoshi.ico"
```

確定同一個目錄下放著 YOSHI.ICO 圖示檔案後，在命令列下達指令來編譯 RC 檔：

```
brcc32 hello.rc
```

若一切無誤，BRCC32 會產生 HELLO.RES 檔，內含我們所需的字串及圖示。若你手上有可編輯字串及圖示資源的資源編輯器，也可略過 RC 檔直接建立 RES 檔。Image Editor 程式只能編輯圖示、游標及點陣圖三種資源，所以在此並不適用，下圖爲使用 Borland

⁸ 指Brian Kernighan及Dennis Ritchie兩人在「The C Programming Language」書中的HELLO.C範例程式。

Resource Workshop 來製作 HELLO.RES 檔的情形。

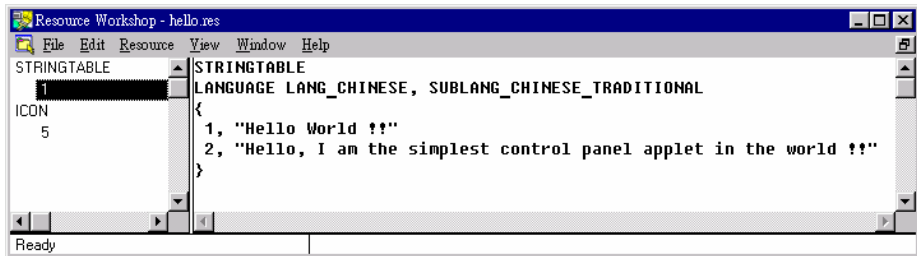


圖 3-4 / 以 Resource Workshop 來編輯 HELLO.RES 資源檔

兩個字串及圖示資源代碼分別採 1、2、5 號，號碼是任意選擇的，沒有其它涵義。

這個最簡單的控制台元件，唯一的作用就是在收到各訊息時跳出訊息盒來告知我們，並且在啟動元件時也秀出一段訊息表明身份。這段程式碼總共不到九十行，很適合作為撰寫進階控制台元件的樣版。

HELLOCPL_UNIT.CPP

```
#0001 #include <windows.h>
#0002
#0003 // 撰寫 Control Panel Applet 程式
#0004 #include <cpl.h>
#0005 #include <cpl.hpp>
#0006
#0007 // 連結資源檔
#0008 #pragma resource "hello.res"
#0009
#0010 void MsgBox(const char* Msg)
#0011 {
#0012     MessageBox(0, Msg, "Hello, Control Panel", MB_ICONINFORMATION);
#0013 }
#0014
#0015 // 匯出 CPlApplet 函式
#0016 extern "C" {
#0017     __declspec(dllexport) LONG _stdcall CPlApplet(HWND, UINT,
#0018         LPARAM, LPARAM);
#0019 }
#0020
#0021 LONG _stdcall CPlApplet(HWND hwndCpl, UINT msg, LPARAM lParam1,
```

```
#0022 LPARAM lParam2)
#0023 {
#0024     PCPLInfo pInfo;
#0025
#0026     switch (msg) {
#0027         case CPL_INIT:
#0028             MsgBox("收到 CPL_INIT !!");
#0029             return 1; // 成功
#0030
#0031         case CPL_GETCOUNT:
#0032             MsgBox("收到 CPL_GETCOUNT !!");
#0033             return 1; // 只包含一個元件
#0034
#0035         case CPL_INQUIRE:
#0036             MsgBox("收到 CPL_INQUIRE !!");
#0037
#0038             pInfo = (PCPLInfo)lParam2;
#0039
#0040             pInfo->idName = 1; // 名稱的 Resource ID
#0041             pInfo->idInfo = 2; // 描述的 Resource ID
#0042             pInfo->idIcon = 5; // 圖示的 Resource ID
#0043             pInfo->lData = 0; // 不需使用者自訂資料
#0044             break;
#0045
#0046         case CPL_NEWINQUIRE:
#0047             MsgBox("收到 CPL_NEWINQUIRE !!");
#0048             break;
#0049
#0050         case CPL_DBLCLK:
#0051             MsgBox("收到 CPL_DBLCLK !!");
#0052
#0053             MsgBox("嗨, 我是全世界最小巧的控制台元件 !!");
#0054
#0055             return 0; // 成功
#0056
#0057         case CPL_STOP:
#0058             MsgBox("收到 CPL_STOP !!");
#0059             return 1; // 成功
#0060
#0061         case CPL_EXIT:
#0062             MsgBox("收到 CPL_EXIT !!");
#0063             return 0; // 成功
#0064     }
#0065
#0066     return 0;
#0067 }
```

```
#0068
#0069 #pragma argsused
#0070 int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason,
#0071     void* lpReserved)
#0072 {
#0073     return 1;
#0074 }
```

由於這是一個 DLL 檔，因此沒有 *WinMain* 函式，取而代之的是 0070 列的 *DllEntryPoint* 函式。0004 及 0005 列分別將 Windows SDK 及 C++Builder 所提供的控制台相關常數與結構型態宣告單元納入編譯。

0008 列利用編譯指示將方才製作出的 HELLO.RES 資源嵌入 DLL，接下來就是此程式最重要的函式—*CPLApplet* 了。*CPLApplet* 函式內部為一個大大的 *switch* 敘述，分別就每個可能收到的訊息進行回應：收到 *CPL_INQUIRE* 訊息時，將 *Param1* 參數轉型為 *PCPLInfo* 型態，將它所指向的結構欄位一一填入對應的資源代碼；收到 *CPL_DBLCLK* 訊息時，呼叫 *MsgBox* 函式秀出訊息—「嗨，我是全世界最小巧的控制台元件 !!」。

測試、安裝及移除

，程式雖然寫完了，先別急著按下【F9】執行程式，這可不是執行檔，而是 DLL，所以不能直接執行，必須透過控制台或 RUNDLL32.EXE 來執行。

控制台元件的副檔名規定為 CPL，而非 DLL，我們可透過【Project Options】對話盒 Applications 頁面的“Target file extension”指定程式連結的結果副檔名為 CPL。



Info

若檔案關聯設定無誤，你也可以在檔案總管中，直接雙擊 CPL 檔案來開啓此 CPL 檔所提供的第一個控制台元件。

在寫作控制台元件時，最繁瑣的工作就是不斷地編譯、安裝、測試、移除的無奈循環。較方便的方式是撰寫批次檔案負責元件的安裝及測試。以下提供兩個批次檔，TESTRUN.BAT 直接測試控制台元件；INSTALL.BAT 將 CPL 檔複製到系統目錄：

TESTRUN.BAT

```
rundll32 shell32.dll,Control_RunDLL helloctl.cpl
```

TESTRUN.BAT 直接呼叫 RUNDLL32 來執行 HELLOCPL.CPL。若此 CPL 檔含有多個控制台元件，則請在 RUNDLL32 敘述後加上元件編號。

INSTALL.BAT

```
del c:\winnt\system32\helloctl.cpl  
copy helloctl.cpl c:\winnt\system32
```

其中 C:\WINNT 是我的 Windows 2000 安裝目錄，請自行更改以符合你的需求。至於移除動作就更簡單了，只要將 CPL 檔直接刪除即可—若檔案無法刪除，應該是控制台正開啓著，請先將控制台程式關閉。

Tips

若你的元件利用 *CPL_INQUIRE* 訊息回傳元件資訊，且欄位值不是全部設為 *CPL_DYNAMIC_RES*，則每次更改程式內的 *TCPLInfo* 資料結構內容後，必須移除 CPL 檔，開啓、關閉控制台，再重新安裝 CPL 檔，下次啓動控制台時，控制台才會再一次發出 *CPL_INQUIRE* 訊息來取得新的元件資訊，否則你將會一直看到快取區裡的資料，無論程式怎麼改都沒有用。

執行 INSTALL.BAT 之後，打開控制台，可以看到控制台內多出一個可愛的 YOSHI，元件名稱為「Hello World !!」。雙擊 YOSHI 圖示，會先依序出現「收到 CPL_INIT !!」、「收到 CPL_GETCOUNT !!」、「收到 CPL_INQUIRE !!」、「收到 CPL_NEWINQUIRE !!」、「收到 CPL_DBLCLK !!」等訊息，最後終於跑出一個「嗨，我是全世界最小巧的控制台元件 !!」訊息—這也是此元件唯一功能，著實簡單，酷吧！按下訊息盒的確定按鈕後，會出現「收到 CPL_STOP !!」、「收到 CPL_EXIT !!」，一個個訊息接著我們預期中照

順序出現，控制台元件至此執行完畢。

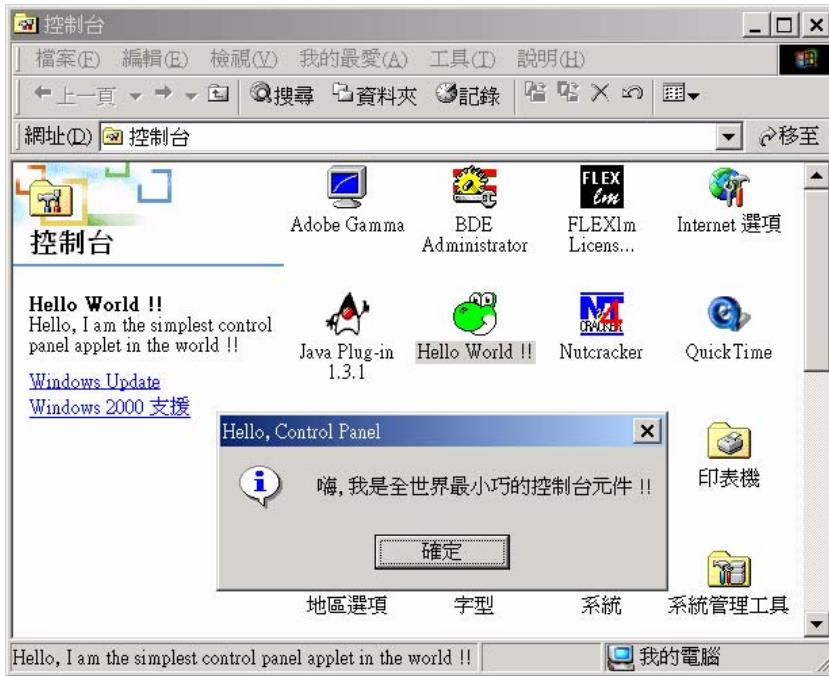


圖 3-5 / 全世界最小巧的控制台元件問世了！

撰寫自己的控制台

移動鏡頭，換個角度，且讓我們反客為主，根據我們對控制台元件的瞭解，想想我們是不是也能自個兒實作一個控制台程式呢？

控制台與 RUNDLL32

在「另類呼叫」小節裏曾經提到，RUNDLL32.EXE 存在的理由，主要是讓 16 bit 的控制台程式也可輕鬆叫用 32 bit 的控制台元件，雖然在 Windows NT/2000 之後，控制台程式

及所有的控制台元件皆是 32 bit 程式，但這個傳統沿襲下來，即使是控制台及檔案總管本身，還是呼叫 RUNDLL32.EXE 來啟動控制台程式。

所以，測試我們的 HELLOCPL.CPL 元件時，你會發現，雖然在控制台開啓時，已經收到 *CPL_INIT*、*CPL_GETCOUNT*、*CPL_INQUIRE* 等訊息了，但雙擊圖示啓動元件時，還是會重新收到 *CPL_INIT*、*CPL_GETCOUNT*、*CPL_INQUIRE* 等訊息，因為控制台還是必須呼叫 RUNDLL32.EXE，讓 RUNDLL32 重新載入 HELLOCPL.CPL、初始化並啓動該元件。

而這個行爲也導致，雖然控制台元件建立的是 Modal 對話框，但我們還是可以將背後的控制台拉到最上層，不像一般 Modal 對話框那樣，會導致背後的視窗無法致能—因為 Modal 視窗的建立者是 RUNDLL32，而非控制台。

邏輯規劃

實作自己的控制台？憑著我們對控制台元件的瞭解，我想答案是肯定的，而且作法也可能簡單得令人無法相信。條列式地將控制台程式正常行爲的進行步驟列出也許可以增加點信心...

啓動控制台程式時...

- 呼叫 *GetSystemDirectory* API 函式取得系統目錄。
- 利用 *FindFirst*、*FindNext* 及 *FindClose* 這組函式來取得系統目錄下所有副檔名為 CPL 的檔案。
- 對於每個 CPL 檔案進行如下動作：
 1. 呼叫 *LoadLibrary* API 函式載入此 CPL 檔。
 2. 呼叫 *GetProcAddress* API 函式取得 *CPLApplet* 函式位址。
 3. 呼叫 *CPLApplet* 函式，分別遞送 *CPL_INIT* 及 *CPL_GETCOUNT* 訊息。

4. 若 *CPL_INIT* 訊息傳回 0，或 *CPL_GETCOUNT* 訊息傳回 0，則放棄對此 CPL 檔的處理。
5. 由 *CPL_GETCOUNT* 訊息的回返值得知控制台元件的數目，針對每個控制台元件發送 *CPL_INQUIRE* 及 *CPL_NEWINQUIRE* 訊息取得個別元件的名稱、描述及圖示，然後再送上 *CPL_STOP* 訊息。
6. 呼叫 *CPLApplet*，遞送 *CPL_EXIT* 訊息。
7. 呼叫 *FreeLibrary* API 函式，釋放此 CPL 檔。

啓動控制台元件時...

當使用者指名要執行某個元件時，取得元件所屬的 CPL 檔名及元件編號後：

1. 呼叫 *LoadLibrary* API 函式載入此 CPL 檔。
2. 呼叫 *GetProcAddress* API 函式取得 *CPLApplet* 函式位址。
3. 呼叫 *CPLApplet*，遞送 *CPL_INIT* 訊息。
4. 呼叫 *CPLApplet*，送上 *CPL_DBLCLK* 訊息，此次呼叫不會立即回返，會直到使用者關閉對話框後才回返。
5. 呼叫 *CPLApplet*，遞送 *CPL_STOP*、*CPL_EXIT* 訊息讓元件及 CPL 檔進行善後工作。
6. 呼叫 *FreeLibrary* API 函式，釋放 CPL 檔。

之所以自行載入、呼叫控制台元件，是希望控制台元件的視窗出現時，能成爲控制台式的 Modal 視窗，不會像控制台那樣，玩一玩就可能將元件視窗玩不見了，得跑到所有視窗下層去找。以這種設計方式，會讓我們的控制台式同一時間只可以啓動一個控制台元件，不能像作業系統提供的控制台那樣，可以啓動某個元件後再切回控制台啓動另一個元件。

嗯，按照這樣的行進邏輯，撰寫控制台式就不是問題了。第一步，將程式介面拉出：

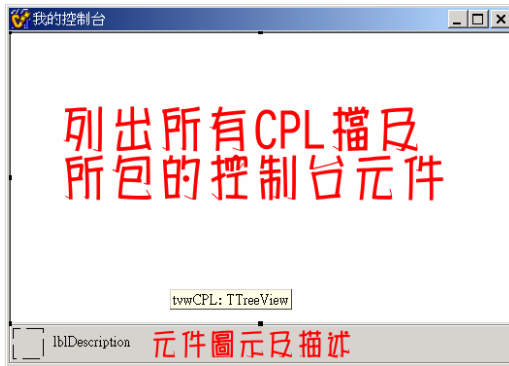


圖 3-6 / 「我的控制台」設計時期模樣

介面很簡單，上面一個樹狀檢視元件，以階層方式將所有 CPL 檔及所包含的控制台元件列出，下面則顯示目前所選擇元件的元件圖示及描述。

按照上述「啟動控制台程式」的行為，開始撰寫取得 CPL 檔及控制台元件資訊的 *GetCPLInformation* 函式：

```
#0001 // 取得所有 CPL 檔及控制台元件資訊
#0002 void __fastcall TMainForm::GetCPLInformation()
#0003 {
#0004     int iResult;
#0005     TSearchRec SearchRec;
#0006     TTreeNode* NewNode;
#0007
#0008     // 開始尋找系統目錄下的 CPL 檔案
#0009     iResult = FindFirst(SystemDirFile("*.CPL").c_str(), faAnyFile,
#0010         SearchRec);
#0011     while (iResult == 0) {
#0012
#0013         // 每找到一個 CPL 檔，就建立新的節點，並將節點標題設為檔名
#0014         NewNode = twvCPL->Items->AddChild(NULL,
#0015             AnsiLowerCase(SearchRec.Name));
#0016
#0017         // 再呼叫 ProcessNode 處理新節點
#0018         ProcessNode(NewNode);
#0019         // 將節點展開
#0020         NewNode->Expand(true);
#0021
#0022         // 尋找下一個 CPL 檔
#0023         iResult = FindNext(SearchRec);
```

```
#0024     }
#0025     FindClose(SearchRec);
#0026
#0027     // 選擇第一個節點
#0028     tvwCPL->Selected = tvwCPL->Items->GetFirstNode();
#0029 }
#0030
#0031 void TMainForm::ProcessNode(TTreeNode* ANode)
#0032 {
#0033     HINSTANCE hLib;
#0034     TCPLApplet CPLProc; // 指向 CPLApplet 函式的函式指標
#0035     int iCount;
#0036     TTreeNode* NewNode;
#0037
#0038     // 用來取得元件資訊的結構
#0039     TCPLInfo CPLInfo;
#0040     TNewCPLInfo NewCPLInfo;
#0041
#0042     // 記錄元件的名稱、描述及圖示
#0043     char sName[256], sInfo[256];
#0044     HICON hAppIcon;
#0045
#0046     // 載入 CPL 檔，檔名即是此節點標題
#0047     hLib = LoadLibrary(ANode->Text.c_str());
#0048     if (!hLib) {
#0049         ANode->Text = ANode->Text + " - Can't be loaded !!";
#0050         return;
#0051     }
#0052
#0053     try {
#0054         // 取得 CPLApplet 函式位址
#0055         CPLProc = (TCPLApplet)GetProcAddress(hLib, "CPLApplet");
#0056         if (!CPLProc) return; // 確定它有 CPLApplet 函式
#0057
#0058         // 初始化 applet
#0059         if (CPLProc((THandle)Handle, CPL_INIT, 0, 0) == 0) return;
#0060
#0061
#0062
#0063         // 取得此 CPL 檔支援的控制台元件數目
#0064         iCount = CPLProc((THandle)Handle, CPL_GETCOUNT, 0, 0);
#0065
#0066         for (int i = 0; i < iCount; i++) {
#0067
#0068             // 遞送 CPL_INQUIRE 訊息，取得 TCPLInfo 結構
#0069             memset(&CPLInfo, 0, sizeof(TCPLInfo));
```

```

#0070     if (CPLProc((THandle)Handle, CPL_INQUIRE, i,
#0071         Longint(&CPLInfo)) != 0) return;
#0072     // 從資源表載入元件名稱, 描述及圖示
#0073     LoadString(hLib, CPLInfo.idName, sName, sizeof(sName));
#0074     LoadString(hLib, CPLInfo.idInfo, sInfo, sizeof(sInfo));
#0075     hAppIcon = LoadIcon(hLib,
#0076         MAKEINTRESOURCE(CPLInfo.idIcon));
#0077
#0078     // 若有任一欄位值為 CPL_DYNAMIC_RES, 則...
#0079     if (CPLInfo.idName == CPL_DYNAMIC_RES ||
#0080         CPLInfo.idInfo == CPL_DYNAMIC_RES ||
#0081         CPLInfo.idIcon == CPL_DYNAMIC_RES) {
#0082
#0083         // 遞送 CPL_NEWINQUIRE 訊息, 取得 TNewCPLInfo 結構
#0084         memset(&NewCPLInfo, 0, sizeof(TNewCPLInfo));
#0085         if (CPLProc((THandle)Handle, CPL_NEWINQUIRE, i,
#0086             (Longint)&NewCPLInfo) != 0) return;
#0087
#0088         // 取得元件名稱
#0089         if (CPLInfo.idName == CPL_DYNAMIC_RES)
#0090             StrCopy(sName, NewCPLInfo.szName);
#0091
#0092         // 取得元件描述
#0093         if (CPLInfo.idInfo == CPL_DYNAMIC_RES)
#0094             StrCopy(sInfo, NewCPLInfo.szInfo);
#0095
#0096         // 取得元件圖示
#0097         if (CPLInfo.idIcon == CPL_DYNAMIC_RES)
#0098             hAppIcon = NewCPLInfo.hIcon;
#0099     }
#0100
#0101     CPLProc((THandle)Handle, CPL_STOP, i, CPLInfo.lData);
#0102
#0103     // 為每一元件建立子節點
#0104     strcat(sName, " - ");
#0105     strcat(sName, sInfo);
#0106     NewNode = tvwCPL->Items->AddChild(ANode, sName);
#0107     // 利用 Data 屬性來儲存 Icon 的 Handle
#0108     NewNode->Data = hAppIcon;
#0109 }
#0110
#0111     CPLProc((THandle)Handle, CPL_EXIT, 0, 0);
#0112 } __finally {
#0113     FreeLibrary(hLib); // 別忘了釋放 CPL 檔
#0114 }
#0115 }

```

0034 列宣告 *CPLProc* 變數，它的型態是 *TCPLApplet*。*TCPLApplet* 的宣告如下（定義於 *cpl.h*）：

```

LONG APIENTRY CPLApplet(
    HWND hwndCpl,    // handle to Control Panel window
    UINT uMsg,       // message
    LONG lParam1,    // first message parameter
    LONG lParam2     // second message parameter
);

typedef CPLApplet TCPLApplet;

```

順利載入 *CPL* 檔後，0055 列呼叫 *GetProcAddress* API 函式取得 *CPLApplet* 函式位址，然後轉型為 *TCPLApplet* 型別，再指派給 *CPLProc* 變數。此後，我們就可把此 *CPLProc* 變數當作是 *CPL* 檔內的 *CPLApplet* 函式，直接呼叫使用，就如同它真的是 *CPLApplet* 函式那般：

```

#0055  CPLProc = (TCPLApplet)GetProcAddress(hLib, "CPLApplet");
#0057  ...
#0059  if (CPLProc((THandle)Handle, CPL_INIT, 0, 0) == 0) return;
#0062  ...
#0064  iCount = CPLProc((THandle)Handle, CPL_GETCOUNT, 0, 0);

```

接下來，就按照應當的步驟，傳入 *CPL_INIT* 訊息初始化，傳入 *CPL_GETCOUNT* 訊息取得元件數目，一一傳入 *CPL_INQUIRE* 訊息取得元件資訊。但是，並不一定傳入 *CPL_NEWINQUIRE* 訊息。先檢查 *CPL_INQUIRE* 訊息傳回的 *TCPLInfo* 結構，若有任一欄位值為 *CPL_DYNAMIC_RES*，才送出 *CPL_NEWINQUIRE* 訊息即可，否則送了也是白搭，取得的結果根本沒用。接著才送出 *CPL_STOP* 訊息，傳入方才取得的 *TCPLInfo* 結構的 *IData* 欄位值。

取得正確的元件資訊後，為每個元件建立一個節點，成為 *CPL* 節點的子節點。我將元件的名稱及描述拼湊為節點的標題，而將元件的圖示記錄在節點 *TTreeNode::Data* 欄位，因此當使用者點選此元件時，可以輕鬆取得元件圖示。全部元件處理過後，送出 *CPL_EXIT* 訊息，再釋放 *CPL* 檔。

個別 *CPL* 檔的處理流程就是這樣，而 0011 ~ 0024 列的迴圈將系統目錄下的所有 *CPL* 檔

全部依樣處理一遍，並為每個 CPL 檔在第一層建立一個新節點，不論它是否包含任何元件。最後，0028 列，將樹狀檢視元件 *tvwCPL* 的焦點節點指向它的第一個節點，之所以採用 *tvwCPL->Items->GetFirstNode* 函式而不直接指派為 *tvwCPL->Items[0]* 的原因是，萬一一個節點都沒有，則 *tvwCPL->Items[0]* 敘述會引發錯誤，而 *GetFirstNode* 函式不會。

好，最主要的 *GetCPLInformation* 函式出爐後，程式也差不多快完成了，不過還有一個重點動作忘了，就是當使用者雙擊元件節點時，必須遞送 *CPL_DBLCLK* 訊息以執行此元件。所以撰寫 *tvwCPL* 元件的 *OnDbClick* 事件處理函式如下：

```
#0001 void __fastcall TMainForm::tvwCPLDbClick(TObject *Sender)
#0002 {
#0003     HINSTANCE hLib;
#0004     TCPLApplet CPLProc; // 指向 CPLApplet 函式的函式指標
#0005     TTreeNode* ANode;
#0006
#0007     TCPLInfo CPLInfo; // 用來取得元件資訊的結構
#0008
#0009     ANode = tvwCPL->Selected;
#0010     if (ANode == NULL || ANode->Level != 1) return;
#0011
#0012     // 載入 CPL 檔，CPL 檔名即為元件節點的父節點標題
#0013     hLib = LoadLibrary(ANode->Parent->Text.c_str());
#0014     if (!hLib) {
#0015         ShowMessage("無法載入 " + ANode->Parent->Text);
#0016         return;
#0017     }
#0018
#0019     // 取得 CPLApplet 函式位址
#0020     CPLProc = (TCPLApplet)GetProcAddress(hLib, "CPLApplet");
#0021     if (CPLProc) {
#0022         THandle h = (THandle)Handle; // conform to type checking
#0023
#0024         // 啟動元件，元件編號即是此節點的次序 (ANode.Index)
#0025         CPLProc(h, CPL_INIT, 0, 0);
#0026         memset(&CPLInfo, 0, sizeof(TCPLInfo));
#0027         CPLProc(h, CPL_INQUIRE, ANode->Index, (Longint)&CPLInfo);
#0028         CPLProc(h, CPL_DBLCLK, ANode->Index, CPLInfo.lData);
#0029         CPLProc(h, CPL_STOP, ANode->Index, CPLInfo.lData);
#0030         CPLProc(h, CPL_EXIT, 0, 0);
#0031     }
#0032     FreeLibrary(hLib); // 別放了釋放 CPL 檔
#0033 }
```


大功告成後，按下【F9】執行程式，等待約三秒鐘的元件初始時間，「我的控制台」視窗終於出現在畫面上：

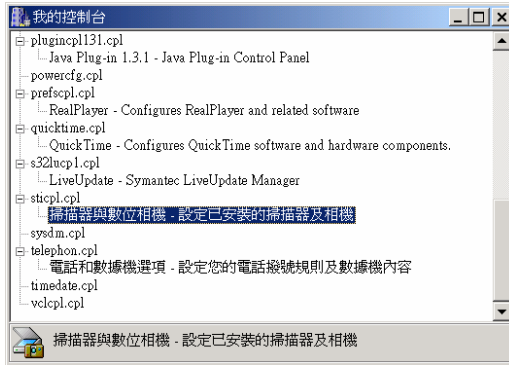


圖 3-7 / 「我的控制台」執行畫面

以樹狀檢視方式，我們可以很清楚地看到，哪些 CPL 檔提供了哪些控制台元件。除了介面較粗糙以及未提供快取功能外，「我的控制台」幾乎提供了「正版」控制台所提供的所有功能，但是因為沒有快取的緣故，每次執行前會花上約三至五秒的時間在搜集所有控制台元件資訊，實在是一大致命傷。不過反正只是寫好玩的，平常還是使用「正版」控制台，所以這個缺點就當作沒看見好了。:P

VCL 的控制台支援

雖然控制台於整個系統中只算小角色，寫起來也不麻煩，但 C++Builder 從第五版開始連這點麻煩都要為程式員省下，從善如流地提供控制台元件的撰寫支援（對於撰寫控制台程式並無支援）。

新增的單元及類別

在這個版本的 VCL 中，新增了一個單元（CtlPanel）及兩個類別：

- *TAppletApplication* 類別
 1. 繼承自 *TComponent* 類別。
 2. 代表一個 CPL 檔。
 3. 為 *TAppletModule* 物件的容器，能包含一或多個 *TAppletModule* 物件。
- *TAppletModule* 類別
 1. 繼承自 *TDataModule* 類別。
 2. 代表一個控制台元件。
 3. 必須加入 *TAppletApplication* 物件中才可為控制台所用。

只是名詞的不同，架構上並沒有差異，想來直接上手應該是輕而易舉的事。我們就使用這兩個類別來實作一個小範例元件好了。

範例元件－觀測記憶體使用狀況

選擇 C++Builder 的【File / New】選項，開啓「New Items」對話盒，只見列表中多了兩個圖示，分別是「Control Panel Application」及「Control Panel Module」。由於我們尚未建立 *TAppletApplication* 物件，所以必須選擇「Control Panel Application」來建立新的控制台元件應用程式，亦即 CPL 檔。

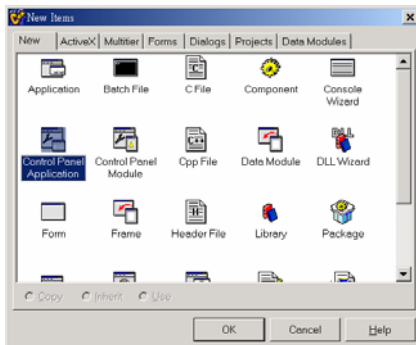
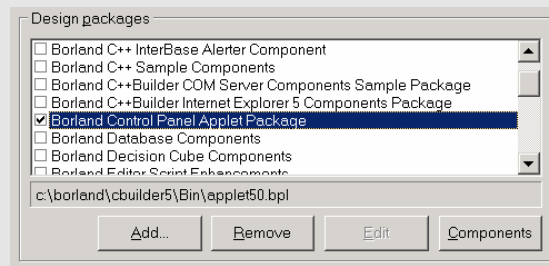


圖 3-12 / 「New Items」對話盒的兩個控制台元件相關圖示

Info

若你的 C++Builder 版本為 5.0 或更新的版本，但無論如何找不到圖 3-12 所標示的控制台相關元件，代表此時整合環境未將控制台元件的設計時期套件（design-time package）applet50.bpl 載入。請由【Project Options】對話盒的 Packages 頁面來進行載入，將「Borland Control Panel Applet Package」項目打勾即可，請參考下圖。



以滑鼠左鍵雙擊「Control Panel Application」圖示後，只見畫面一閃，整合環境已為我們建立一個新的專案，畫面上出現的不是熟悉的 Form，而是 Applet Module，它的设计模式與 Data Module 一模一樣：

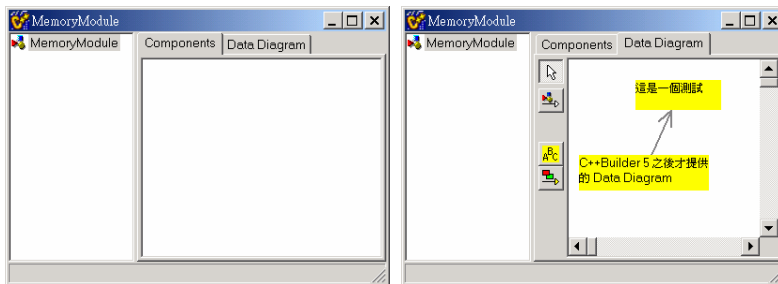


圖 3-13 / TAppletModule 物件的設計模式畫面

設計視窗分為左右兩欄，右欄又分為兩頁，分別是放置非可見元件的「元件頁」與設計資料流程圖的「流程圖頁」。

對於這個控制台元件的用途，我想讓它回應 `CPL_NEWINQUIRE` 訊息，將元件描述設定為當時整個系統記憶體的使用狀況。由於每次打開控制台時，控制台程式就會重新發送

CPL_NEWINQUIRE 訊息給控制台元件，所以雖然記憶體使用狀況不是即時更新，至少也會保持在開啓控制台程式的那瞬間。

戰戰兢兢地按下【F11】叫出物件檢視器，看到 *TAppletModule* 的屬性及事件列表，我想大家都可以鬆口氣，心中大概有個底了：

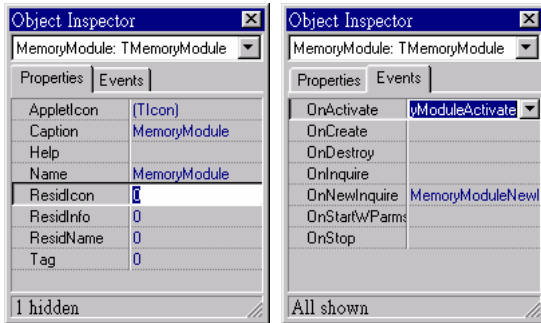


圖 3-14 / *TAppletModule* 的屬性及事件

看來，只要指定好它的 *AppletIcon* 屬性，然後分別撰寫 *OnNewInquire* 事件處理函式提供元件屬性，再撰寫 *OnActivate* 事件處理函式讓元件啓動時秀段訊息，就大功告成了。

```
#0001 void __fastcall TMemoryModule::AppletModuleNewInquire(
#0002     TObject *Sender, int &lData, HICON &hIcon,
#0003     AnsiString &AppletName, AnsiString &AppletInfo)
#0004 {
#0005     TMemoryStatus MemStat;
#0006
#0007     // 指定元件名稱
#0008     AppletName = Format("記憶體使用狀況 (%s)",
#0009         OPENARRAY(TVarRec, (TimeToStr(Now()))));
#0010
#0011     // 呼叫 API 函式取得目前記憶體使用狀況
#0012     MemStat.dwLength = sizeof(TMemoryStatus);
#0013     GlobalMemoryStatus(&MemStat);
#0014
#0015     // 指定元件描述
#0016     AppletInfo = Format("全部: %d KB 剩餘: %d KB",
#0017         OPENARRAY(TVarRec,
#0018             ((int)MemStat.dwTotalPhys / 1024,
#0019             (int)MemStat.dwAvailPhys / 1024)));
#0020 }
```

OnNewInquire 事件處理函式 *MemoryModuleNewInquire* 是元件的重點函式，它呼叫 *GlobalMemoryStatus* API 函式取得當前的記憶體使用狀況，然後設定為元件的描述；而元件名稱則加上目前時間，讓使用者不致搞混記憶體狀況的取得時間。

TMemoryStatus 結構包含不少有用的資訊，不過在此我們只用到 *dwTotalPhys* 及 *dwAvailPhys* 兩個（單位皆是位元組）：

```
typedef struct _MEMORYSTATUS { // mst
    DWORD dwLength;           // 結構長度
    DWORD dwMemoryLoad;       // 記憶體使用比例
    DWORD dwTotalPhys;        // 實體記憶體大小
    DWORD dwAvailPhys;        // 剩餘記憶體大小
    DWORD dwTotalPageFile;    // 分頁檔大小
    DWORD dwAvailPageFile;    // 剩餘分頁檔大小
    DWORD dwTotalVirtual;     // 全部的行程可用記憶體大小
    DWORD dwAvailVirtual;     // 剩餘的行程可用記憶體大小
} MEMORYSTATUS, *LPMEMORYSTATUS;

typedef MEMORYSTATUS TMemoryStatus;
```

使用者雙擊圖示時會送出 *CPL_DBLCLK* 訊息，導致觸發 *OnActivate* 事件，所以我們也要為它撰寫事件處理函式：

```
#0001 void __fastcall TMemoryModule::AppletModuleActivate(
#0002     TObject *Sender, int Data)
#0003 {
#0004     ShowMessage("VCL 的 Control Panel 支援測試");
#0005 }
```

很快，一個簡單的控制台元件又完成了。在 *Applet Module* 中按下滑鼠右鍵，出現快捷功能表，分別是【安裝】、【移除】、【啟動控制台】等功能，哇哈，真的很方便呢。立刻經由此功能表將控制台元件安裝到系統目錄下，接著啟動控制台：

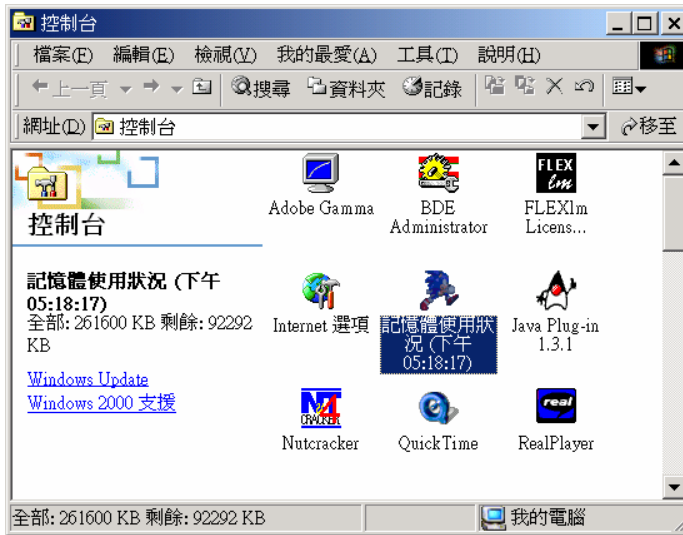


圖 3-15 / 依賴 VCL 支援寫出的「記憶體狀況偵詢」控制台元件

一切就如預期般上演著...

背後支援的 TAppletApplication

等等，不是有兩個新類別嗎？為什麼到現在只見 *TAppletModule* 獨領風騷，*TAppletApplication* 為何總是不見蹤影呢？

點選【Project / View Source】，打開專案原始碼：

```
#0001 #include <Ctlpanel.hpp>
#0002
#0003 USEFORM("memory.cpp", MemoryModule); /* TAppletModule: File Type */
#0004
#0005 extern "C" __declspec(dllexport) __stdcall
#0006     int CPlApplet(unsigned hwndCPl, unsigned uMsg,
#0007                 int lParam1, int lParam2)
#0008 {
#0009     return (Ctlpanel::CPlApplet(hwndCPl, uMsg, lParam1, lParam2));
#0010 }
#0011
#0012 int WINAPI DllEntryPoint(HINSTANCE hinst,
```

```
#0013 unsigned long reason, void*)
#0014 {
#0015 using Ctlpanel::Application;
#0016 Application->Initialize();
#0017 Application->CreateForm(__classid(TMemoryModule), &MemoryModule);
#0018 Application->Run();
#0019 return 1;
#0020 }
```

0012 列的 *DllEntryPoint* 代表此專案是 DLL，沒錯，因為 CPL 檔正是 DLL 檔。0006 列讓此 DLL 檔匯出 *CPlApplet* 函式，此 *CPlApplet* 函式宣告於 *CtlPanel* 單元中。

還是不見 *TAppletApplication* 類別呀？原來，0015 列的 *using* 敘述讓此處的 *Application* 變數不再代表常見的 *TApplication* 類別物件，而是使用由 *CtlPanel* 單元提供的 *TAppletApplication* 類別物件。你可以在 *CtlPanel.hpp* 單元中找到下列宣告：

```
extern PACKAGE TAppletApplication* Application;
```

而同樣位於 *CtlPanel* 單元的 *CPlApplet* 函式，每當收到訊息時，便將訊息交由 *Application* 及指定的 *TAppletModule* 物件處理，這也正是整個控制台元件程式運作的核心：

```
#0001 function CPlApplet(hwndCpl: THandle; uMsg: DWORD;
#0002 lParam1, lParam2: Longint): Longint;
#0003 var
#0004 Temp: Boolean;
#0005
#0006 begin
#0007 Result := 0;
#0008 Temp := True;
#0009
#0010 with Application, Application.Modules[lParam1] do
#0011 begin
#0012 case (uMsg) of
#0013 CPL_INIT : DoInit(Temp);
#0014 CPL_GETCOUNT:
#0015 begin
#0016 Result := ModuleCount;
#0017 DoCount(Result);
#0018 Exit;
#0019 end;
#0020 CPL_INQUIRE : DoInquire(PCplInfo(lParam2)^);
#0021 CPL_NEWINQUIRE : DoNewInquire(PNewCplInfo(lParam2)^);
#0022 CPL_DBLCLK : DoActivate(LongInt(lParam2));
```

```
#0023      CPL_STOP          : DoStop(LongInt(LParam2));
#0024      CPL_EXIT         : DoExit;
#0025      CPL_STARTWPARMS  : DoStartWParms(PChar(LParam2));
#0026      CPL_SETUP        : DoSetup;
#0027      end;
#0028      end;
#0029
#0030      Result := Integer(Temp);
#0031      end;
```

看到終於連控制台的支援也加入的 VCL，我開始幻想著哪天也加入 DirectX、OpenGL 的遊戲設計的強力支援，或者加入 VxD、kernel mode driver、WDM 等驅動程式撰寫的類別庫支援，讓我們可以動一動滑鼠，就把遊戲或驅動程式的程式架構完成，程式設計師們不個個樂透了才怪。

第四章

分秒必爭，細說計時器

雖然深諳 Win32 計時器，
不過煮了六十秒與七十五秒的藍山咖啡，
我總是喝不出其中差別。



記得大二時修組合語言課程，教到「常駐程式」章節時，老師出的作業是撰寫一個永遠擺在螢幕右上角的小時鐘，不論處在文字模式或繪圖模式下皆能正常顯示，最好還能做到整點報時及鬧鐘功能。除了安裝、解除常駐程式及繪製時鐘較為繁瑣外，重點就在於程式要如何計時，才能每隔一秒鐘自動更新畫面。

系統的clock chip每 54.925 毫秒（約每秒 18.2 次¹）會對CPU做一次中斷請求，觸發 08h 及 1Ch中斷²，因此若要在程式中計時，或是進行規律性的動作，我們通常都改寫 08h或 1Ch中斷的中斷處理函式（ISR，Interrupt Service Routine）來達成。最簡單的方式是宣告一個整數變數作為計數器，每次進入中斷處理函式時就遞增，如此一來，當計數器的值達到 18 時，表示時間過了約一秒鐘，但實際上是 18/18.2 秒，此時便可更新時鐘或畫面及將計數器歸零。當然啦，另有較準確的方法，否則依這方法寫出來的計時程式每三分鐘就有兩秒鐘的誤差，太可怕啦！

聽起來很麻煩吧？幸好這是過去的事了！在 Windows 環境下，系統即內建一組計時器函式供我們取用。不論你要秀動畫、寫遊戲或是想寫個每隔三十分鐘將你從網路上踢下來的程式，只要是計時的工作，就可以交給這組計時器函式來做。

計時器的用途很廣，除了遊戲製作及多媒體應用，在網路連線程式中等待回應時，也經常利用計時器來偵測連線時間是否過長，以免讓使用者等待太久。此外，有些未註冊的共享軟體只容許一次使用五分鐘，超過時間就取消某些功能；具有自動存檔功能的文書處理軟體或程式發展工具，每隔五或十分鐘且系統負載低時會十分貼心地自動存檔；唉呀！差點忘了，螢幕底邊狀態列最右方的系統時鐘，正是計時器函式的最佳應用。

¹ 再精確一點地說，是每秒 18.2064819336 次。

² 由計時器IC 8253的channel 0 連接至 8259 中斷控制器產生INT 8h中斷，在其中斷處理函式中再執行INT 1Ch中斷。

計時器 API

USER 模組提供一組計時器 API 函式，讓我們在撰寫程式時不必理會任何硬體中斷、軟體中斷、中斷處理函式及 CPU 時脈等繁瑣細節，就可以同時擁有多個以毫秒為計時單位的計時器，並且更容易呼叫使用。

這組計時器函式十分簡單，只有兩個函式，分別用來建立及消滅計時器。

建立計時器

建立一個計時器，必須呼叫 *SetTimer* 函式。

```
UINT SetTimer(  
    HWND hWnd,  
    UINT nIDEvent,  
    UINT uElapse,  
    TIMERPROC lpTimerFunc  
);
```

參數

<i>hWnd</i>	程式擁有的視窗handle。
<i>nIDEvent</i>	計時器編號，可為大於零、小於 $2^{32}-1$ 的任意整數，對於使用同一個視窗handle的所有計時器來說，編號必須唯一。
<i>uElapse</i>	每次觸發的間隔時間，單位為毫秒（millisecond），數值可從 0 至 4294967295（2 的 32 次方減 1），約為 49.7 天。
<i>lpTimerFunc</i>	觸發時所呼叫的回呼函式位址，若你希望使用視窗訊息的觸發方式，傳入NULL即可。

回返回值

如果成功，傳回計時器編號；如果失敗則傳回零。

TIMERPROC 的函式原型為：

```
VOID CALLBACK TimerProc (HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime);
```

間隔時間的最小值可為 0，*WM_TIMER* 訊息將以儘可能最快的速度送出，也就是上述的 54.925 毫秒；最大值可設為 4294967295 毫秒，可換算為 49 天 17 小時 2 分 47 秒又 295 毫秒。事實上，實測的結果是，若間隔時間設定值大於 2147483647，也就是 0x7FFFFFFF 毫秒的話，效果與將間隔時間設定為 0 毫秒一模一樣－源源不絕的 *WM_TIMER* 訊息將會讓你措手不及。

WM_TIMER 訊息

WM_TIMER 傳遞的訊息結構如下：

```
struct TWMTimer {
    unsigned    Msg;           // 訊息編號
    int         TimerID;      // 計時器編號
    void        TimerProc;    // 回呼函式
    int         Result;       // 傳回值
};
```

SetTimer 的第四個參數 *lpTimerFunc* 是一個典型的回呼函式－將函式位址交給系統，供系統在需要時呼叫。特別的是，這個回呼函式並不是必要的，若沒有提供，則當計時器觸發時，視窗將會收到 *WM_TIMER* 視窗訊息。

每當計時器觸發，系統會產生一個 *WM_TIMER* 訊息至擁有該視窗的執行緒之訊息佇列，當訊息迴圈中的 *DispatchMessage* 遇上 *WM_TIMER* 訊息時，會進行特別處理：檢查欲傳送的 *WM_TIMER* 訊息結構中，*TimerProc* 欄位是否為 *NULL*：

- 若 *TimerProc* 為 *NULL*，表示我們呼叫 *SetTimer* API 函式時並沒有傳入函式位址，此時 *DispatchMessage* 會按照正常的步驟，如同處理其它訊息般，呼叫該視窗的視窗程序，將 *WM_TIMER* 訊息交由該視窗程序處理。
- 若 *TimerProc* 不為 *NULL*，此時 *DispatchMessage* 不會呼叫該視窗的視窗程序，而直接呼叫 *TimerProc*。

`WM_TIMER` 跟 `WM_PAINT` 訊息相同，都屬於低優先權的視窗訊息。只有在訊息佇列沒有其它訊息的情況下，這兩個訊息才會被取出處理。另外它們還有一個特性——一個訊息佇列絕不會同時含有超過一個的 `WM_TIMER` 及 `WM_PAINT` 訊息。若上回觸發時的 `WM_TIMER` 訊息還未被處理，計時器又觸發了，則此較新的 `WM_TIMER` 訊息就不會進入訊息佇列，而被直接捨棄掉，當作沒發生過這檔事。

Windows 3.1 中，每個訊息佇列預設只能存放八個訊息，若沒有這樣的設計，只要間隔時間短了點，或程式的處理動作慢了點，訊息佇列將很快地被 `WM_TIMER` 訊息塞滿。在 Win32 下，訊息佇列的資料結構大幅改進，從陣列變成串列，取消訊息佇列容量的限制，但是同一時間只能持有一個 `WM_TIMER` 訊息的原則依舊不變。否則，一旦計時器觸發動作太頻繁，光是處理這些接踵而至的 `WM_TIMER` 訊息就措手不及，程式大概也無法進行正常的動作，只有請出「工作管理員」，將程式暴力終結一途了。

既然 `WM_TIMER` 訊息有被捨棄的可能，程式中就不可以太依賴 `WM_TIMER` 的觸發次數來進行任何動作及決策，例如利用每秒觸發一次的計時器來撰寫碼錶程式等。由於來不及處理的 `WM_TIMER` 訊息會被系統直接捨棄，我們無法得知到底錯失了多少 `WM_TIMER` 訊息。

消滅計時器

當某個計時器已經圓滿地達成任務，欲功成身退時，只要呼叫 `KillTimer` 函式，傳入它的計時器編號即可。

```
BOOL KillTimer(
    HWND          hWnd,
    UINT          ulIDEvent
);
```

參數

hWnd 呼叫 `SetTimer` 所傳入的視窗 handle。

ulIDEvent `SetTimer` 傳回的計時器編號。

回返回值

如果成功，傳回 *true*，否則傳回 *false*。

即使程序結束時會自動將所有計時器消滅，最好還是養成「有借有還」的習慣，程式結束前記得將所有尚未消滅的計時器一一關閉。

有時候，程式的錯誤會發生在視窗關閉在即，某些資料結構已經釋放，但計時器仍努力地照常觸發之時。可能就在這極為巧合的一瞬間，計時器觸發，而計時器處理函式卻使用了已被回收的資料，造成非法記憶體存取或其它錯誤，這可不是件好玩的事。因此，呼叫 *KillTimer* 除了將計時器消滅，不再觸發外，對於訊息佇列中待處理的 *WM_TIMER* 訊息，也會一併清除掉。

視窗是必要的嗎？

截至目前為止，我們在呼叫 *SetTimer* 時皆乖乖地傳入一個視窗 *handle*，如果我們的程式沒有視窗呢？是的，有時候我們的程式可能連視窗 *handle* 都沒有，例如在 *console* 程式或沒有產生任何視窗的工作執行緒中，這時就需要另一種方式來建立計時器。

呼叫 *SetTimer* 函式時，第一個參數 *hWnd* 填入 0，表示不提供視窗 *handle*，同時第二個參數 *nIDEvent* 會被忽略。原因我們在前頭提過，計時器編號只用來辨視所有使用同一個視窗 *handle* 的計時器，現在我們不提供視窗 *handle*，系統勢必將這些可能來自不同程式，卻同樣沒有提供視窗 *handle* 的計時器們做區別，最簡單的解決方法即是不讓使用者自行指定計時器編號，而改由系統分派。所以若程式擁有多個計時器共用同一個回呼函式，必須將 *SetTimer* 的傳回值，也就是系統分派的計時器編號記錄下來，才能在計時器觸發時，依計時器編號來分辨訊息是由哪個計時器觸發。因為呼叫 *KillTimer* 時必須得將計時器編號傳入，所以即使程式中只用到一個計時器，別偷懶，*SetTimer* 傳回的計時器編號還是得乖乖記錄下來。

當然囉，即使你的程式擁有視窗，你還是可以以不傳入視窗 *handle* 的方式來使用計時器

函式。

我們前頭提過，回呼函式是由 *DispatchMessage* 在處理 *WM_TIMER* 視窗訊息時所呼叫的。注意：「*WM_TIMER* 訊息是得知計時器觸發的必要條件」。因此，若你的程式沒有訊息佇列，並且沒有訊息迴圈將訊息由佇列中取出，且呼叫 *DispatchMessage* 來分發訊息，就無法接收計時器的觸發訊息！

訊息佇列及訊息迴圈才是計時器函式正常運作的要素，有沒有視窗並不重要！

現在我們以一個 console 程式為例，首先建立計時器回呼函式：

```
#0001 void _stdcall TimerProc(HWND Window, UINT Message, UINT idEvent,
#0002     DWORD dwTime)
#0003 {
#0004     printf("Timer triggered !!\n");
#0005 }
```

主函式設定好間隔時間為一秒的計時器後，會停住不動，等待按下換行鍵，最後消滅計時器，結束程式。

```
#0001 // 建立計時器
#0002 int TimerID = SetTimer(0, 0, 1000, (TIMERPROC)&TimerProc);
#0003
#0004 getchar(); // 等待換行鍵
#0005
#0006 // 摧毀計時器
#0007 KillTimer(0, TimerID);
```

一切看起來都很合理，但是程式執行之後，即使你將電腦擱著跑去看場電影再回來，我們期待出現的「Timer triggered !!」字串依舊杳無音信，怎麼回事？很簡單，console 程式沒有訊息佇列，導致系統沒地方可放 *WM_TIMER* 訊息，兩方沒辦法溝通，而程式只能像呆頭鵝似的痴痴地等著不可能到達的訊息。知道問題的原因，對症下藥，讓我們為它加上訊息佇列及提取訊息的訊息迴圈試試。

所有的執行緒建立時都沒有訊息佇列，直到它呼叫任何一個 USER 或 GDI 函式為止。

上頭的問題在於訊息佇列的缺席。作業系統設計小組的出發點很好，沒有呼叫到任何 `USER` 或 `GDI` 函式的執行緒根本用不著訊息佇列，所以訊息佇列並不隨著執行緒的產生而自動出現，而是在執行緒呼叫第一個 `USER` 或 `GDI` 函式時才建立。基於這個原則，既然我們曉得構成訊息迴圈的主角—`GetMessage` 及 `DispatchMessage` 都是 `USER` 模組的成員，所以放心地加入訊息迴圈，執行緒必然會建立起訊息佇列。因此我們將主程式改成：

```
#0001 // 建立計時器
#0002 int TimerID = SetTimer(0, 0, 1000, (TIMERPROC)&TimerProc);
#0003
#0004 // 取得訊息，收到 WM_QUIT 時跳離迴圈
#0005 TMsg Msg;
#0006 while (GetMessage(&Msg, 0, 0, 0))
#0007     DispatchMessage(&Msg);
#0008
#0009 getchar(); // 等待換行鍵
#0010
#0011 // 摧毀計時器
#0012 KillTimer(0, TimerID);
```

因為我們只會接收到 `WM_TIMER` 及 `WM_QUIT` 兩個訊息，這裡使用的是簡化版的訊息迴圈。將程式執行起來，朝思暮想的觸發訊息如願出現了。同樣的，你也可以將這技巧應用在不具任何視窗的背景執行緒中，延伸計時器函式的應用場合。

量測計時器的精確度

介紹完 Windows 提供的計時器函式後，也許你已經覺得很滿意了，但是我們對它的效果仍一無所知，說不定它根本不守時，經常遲到呢！讓我們做個精確度的實驗。

程式很單純，一邊利用間隔時間為 10 毫秒的計時器累計觸發次數，另一邊呼叫 `GetTickCount` 來取得正確的時間差。咦？前段文章才剛交待，別用計時器來計數，怎麼馬上自打嘴巴啦？這個實驗設計的目的是測量計時器的精確度，而累加計數正是使誤差明顯化最直覺的方法，所以只是個實驗加上錯誤的示範，供作借鏡，撰寫程式時千萬記得別依樣畫葫蘆就是。

坦白說，看到這個實驗，我的國中理化老師一定很後悔當初沒把我當掉！這個對照組實驗乍看之下頗為合理，一個是計數器，一個是 *GetTickCount* API，問題在於這是個量測精確度的實驗，而我們尚未驗證 *GetTickCount* 的精確度呢！沒辦法，先假設 *GetTickCount* 至少擁有 10 毫秒的精確度吧，下個小節介紹高精確度的計時器後，回頭再來驗證 *GetTickCount* 的精確度³，不然器材不足，實驗也做不下去了。:p

程式提供三種方式來計數，第一、二種都是採用前頭說明的 *SetTimer*，但前者不傳入回呼函式位址，以 *WM_TIMER* 訊息來觸發，而後者傳入函式位址，由系統呼叫回呼函式；第三種先擱著吧，我們稍後馬上提到，先來看看 Windows 計時器函式的表現。

雖然三種計時器觸發方式皆不同，但我為它們撰寫一個共用的 *CommonTimerProc* 程序，不論哪種方式觸發，所作的動作皆相同：首先呼叫 *GetTickCount* 取得目前時間，再減掉設置計時器時的時間 *StartTime*，得到實際花費時間，接著累加計數器，最後計算誤差，並分別由三個不同的 *TLabel* 元件顯示。

```
#0001  DWORD TimerInterval = 10; // 預設間隔時間為 10 毫秒
#0002
#0003  int MMTimerID; // 儲存 MMTimer 的 ID
#0004  DWORD StartTime; // 啟動計時器時由 GetTickCount 取得的時間
#0005  DWORD TriggerCount; // 觸發次數
#0006  DWORD StopCount; // 若不為 0，觸發次數到達此值後自動停止
#0007
#0008  void _stdcall TimerProc(HWND Window, UINT message, UINT idEvent,
#0009     DWORD dwTime)
#0010  {
#0011     Form1->CommonTimerProc();
#0012  }
#0013
#0014  void _stdcall MMTimerProc(UINT uTimerID, UINT uMessage, DWORD
#0015     dwUser, DWORD dw1, DWORD dw2)
#0016  {
#0017     Form1->CommonTimerProc();
#0018  }
#0019
#0020  void __fastcall TForm1::WMTimer(TWMTimer& message)
#0021  {
```

³ 事實上，*GetTickCount* 的確擁有約 10 毫秒的精確度。

```
#0022 CommonTimerProc();
#0023 }
#0024
#0025 void TForm1::CommonTimerProc()
#0026 {
#0027     DWORD TimeDiff; // 實際花費時間
#0028
#0029     // 計算實際花費時間
#0030     TimeDiff = GetTickCount() - StartTime;
#0031     lblElapsed->Caption = "Time : " + IntToStr(TimeDiff) + " ms";
#0032
#0033     // 累加計數器
#0034     TriggerCount++;
#0035     lblCount->Caption = "Count: " + IntToStr(TriggerCount) + " times";
#0036
#0037     // 誤差 = 實際實際花費時間 - 計數器 * 間隔時間
#0038     lblError->Caption = "Error: " + IntToStr(TimeDiff -
#0039         TriggerCount * TimerInterval) + " ms";
#0040
#0041     if (StopCount != 0 && TriggerCount == StopCount)
#0042         btnStopClick(btnStop);
#0043 }
#0044
#0045 void __fastcall TForm1::btnStartClick(TObject *Sender)
#0046 {
#0047     TimerInterval = StrToIntDef(txtInterval->Text, 10);
#0048     StopCount = StrToIntDef(txtStopCount->Text, 0);
#0049
#0050     rgpTimerType->Enabled = false;
#0051     TriggerCount = 0;
#0052     StartTime = GetTickCount();
#0053
#0054     // 根據選擇產生不同的計時器
#0055     switch (rgpTimerType->ItemIndex) {
#0056         case 0: SetTimer(Handle, 1, TimerInterval, NULL);
#0057             break;
#0058
#0059         case 1: SetTimer(Handle, 1, TimerInterval,
#0060             (TIMERPROC)&TimerProc);
#0061             break;
#0062
#0063         case 2: MMTimerID = timeSetEvent(TimerInterval, 0,
#0064             (LPTIMECALLBACK)&MMTimerProc, 0,
#0065             TIME_PERIODIC | TIME_CALLBACK_FUNCTION);
#0066             break;
#0067     }
```

```

#0068
#0069   btnStop->Enabled = true;
#0070 }
#0071
#0072 void __fastcall TForm1::btnStopClick(TObject *Sender)
#0073 {
#0074     switch (rgpTimerType->ItemIndex) {
#0075         case 0: case 1: KillTimer(Handle, 1);
#0076             break;
#0077
#0078         case 2: if (MMTimerID != 0)
#0079             timeKillEvent(MMTimerID);
#0080     }
#0081
#0082     rgpTimerType->Enabled = true;
#0083     btnStop->Enabled = false;
#0084 }

```

執行方法十分簡單：先選擇計時器的種類，填好間隔時間及欲停止計數次數，按下【Start】按鈕，讓計時器一邊計數，一邊呼叫 *GetTickCount* 將真正花費的時間顯示出來。

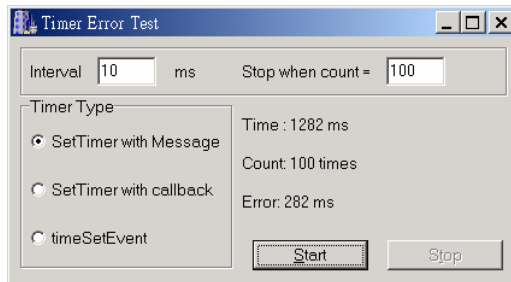


圖 4-1 / Windows NT 下 TimerErr 程式執行畫面

下表是在 Windows 95 及 Windows NT 兩個平臺上，使用視窗訊息觸發的計時器函式，在不同間隔時間測試出來的數據，由此可以很明顯地看出之間的差異：

表 4-2 / TimerErr 程式執行結果，採用計時器函式來計數，每次觸發的平均間隔時間（單位為毫秒）

平臺 \ 間隔	1 毫秒	10 毫秒	20 毫秒	50 毫秒	100 毫秒	1000 毫秒	10000 毫秒
Windows 95	55.04	55.89	57.03	57.96	113.6	1039.4	10018.6
Windows NT	10.21	10.21	20.03	50.07	100.2	1001.5	10004.4

由這些實測數據可以歸納出以下結論：

- Windows 95 下，不論要求多麼短の間隔時間，實際上每次觸發間隔時間必定大於 54 毫秒，由此可見 Windows 95 使用的是前頭所提的，8253 IC channel 0 每 54.925 毫秒會產生一次 08h 中斷的特性，將它包裝為 API 而已。因此雖然提供的時間設計單位為毫秒，但骨子裡卻是以 54.925 毫秒為計數單位；這代表當你設定間隔時間為一秒（1000 毫秒）時，效果與設定為 $54.925 * 18 = 989$ 毫秒根本一模一樣；再換個說法，即使將間隔時間設定為小於 55 毫秒的值，系統仍舊只能每隔 54.925 毫秒觸發一次。
- Windows NT 下，計時器函式大約擁有 10 毫秒的精確度，因此若需用的間隔時間小於 10 毫秒，請勿使用計時器函式。而間隔時間大於 10 毫秒時，誤差通常不會超過 10 毫秒，表現差強人意，但至少比 Windows 95 穩當多了。

計時器函式的罩門 – WM_TIMER

你也可以在自個兒的電腦進行上頭實驗，但是為了測量的精確度，請注意...

執行 TimeErr 程式前，請先關閉任何會不斷使用 CPU 及定時產生視窗訊息的應用程式，按下【Start】按鈕後一直到計數完成前，請勿移動滑鼠，也不要敲打鍵盤，更不要同時操作其它程式，換句話說：「讓 CPU 全力服務 TimeErr 程式」就對了！

如果不這麼做，甚至打開一些「重量級」或是較吃 CPU 資源的軟體，如 Microsoft Office、Developer Studio 或 Borland 的 Delphi、C++Builder 甚至 JBuilder，不然開個 WinPlay3 或 WinAMP 來聽聽 MP3 音樂，在我的 Windows NT 系統上，十分輕鬆愉快地，馬上就可以產生出數十秒的誤差，不論使用 WM_TIMER 視窗訊息或回呼函式的方式。

為什麼會有這麼大的誤差呢？主要原因就是漏失掉的 WM_TIMER 訊息，而回呼函式也是透過 WM_TIMER 訊息才發生的，因此兩種方式的效果一致。在系統中，所有訊息產生出來後，會先放到系統的訊息佇列中，再一個個拿出，分發至各個執行緒的訊息佇列中。而每個執行緒訊息佇列中的訊息，也必須由訊息迴圈中的 GetMessage 或 PeekMessage 一

個個取出處理。所以一旦 CPU 的工作重了些，來不及在 10 毫秒內將其它較重要的任務完成，也許控制權根本沒落到我們的程式，也許程式得到了 CPU 工作權，但是忙著處理其它的訊息，低優先權的 *WM_TIMER* 訊息常就這樣被犧牲掉。

因此，我們知道，Windows 提供的計時器函式，會很認真地趕赴每一場約會，但不順遂事十常八九，有時塞車有時睡過頭有時被問卷調查的小妹纏住，可能常常遲到或缺席，對於重大任務，可別完全信任它。即使如此，對於大部分不要求精確度的應用程式，Windows 提供的這套計時器函式也足夠了。

更精確的計時器

多媒體應用程式的需求

對於許多人而言，多媒體電腦幾乎等於「可以聽 MP3、看 VCD/DVD 的電腦」。微軟的 Video for Windows 標準推出之後，即使當年的 CPU 好慢、顯示卡好爛，看著螢幕上小小的、「寸動式」且解析度極低的撥放畫面，還是看得樂不可支。大家都知道，動態影像其實只是一幕幕靜態影像，以十分穩定的頻率快速切換，利用眼睛的視覺暫留原理，達成動態影像的效果。在電腦中撥放 VCD 也是一樣，必須至少以每秒 15 至 30 張的頻率快速更新畫面，才能讓使用者十分流暢地欣賞動畫。要達到這撥放速率也許不難，但是重點是撥放速率必須十分穩定，不可以時快時慢或延遲輸出。因此，要製作一個動畫撥放程式，首先，必須擁有精確、穩定的計時能力。

媒體撥放程式，或說的更廣泛些，舉凡音效、影像、動畫的撥放及錄製，皆需要精細嚴格的時間控制。現在大概沒有人能夠忍受撥三秒，停半秒的撥放品質，或者是錄製成的音效檔語調忽高忽低，若緩若急。處理多媒體資料時，時間是極為重要的品質要素。Windows 計時器函式太容易產生誤差，無法擔負起如此重責大任，因此，微軟另外提供一套多媒體計時器（Multimedia Timer）函式，專供多媒體應用程式使用。

取得解析度範圍

看來這組計時器函式確實是有備而來，我們可以取得計時器所支援的解析度（resolution）範圍。

```
MMRESULT timeGetDevCaps(  
    LPTIMECAPS lpTimeCaps,  
    UINT        uSize  
);
```

參數

lpTimeCaps 指向 *TIMECAPS* 結構的指標，用以取得解析度資訊。

uSize *TIMECAPS* 結構的大小，你應該透過 *sizeof* 運算子取得它。

回返值

若成功，傳回 *TIMERR_NOERROR*，否則傳回 *TIMERR_STRUCT*，表示 *uSize* 參數與預期不同。

TIMECAPS 結構定義如下：

```
typedef struct {  
    UINT wPeriodMin; // 所支援的解析度下限，亦即最大誤差  
    UINT wPeriodMax; // 所支援的解析度上限，亦即最小誤差  
} TIMECAPS;
```

何謂解析度？到電腦商店購買顯示器時，我們可以看到規格表上寫著，某某顯示器的解析度可高達 1280 x 1024 或 1600 x 1280 等等，以 1600 x 1280 最高解析度為例，表示這部顯示器的畫面最多可以切割成橫向 1600 單位、縱向 1280 單位。而現在所指的計時器解析度意思不大相同，指的是「最大誤差容許範圍」。我們可以使用如下函式取得多媒體計時器所支援的解析度上下極限：

```
#0001 void __fastcall TForm1::btnCapsClick(TObject *Sender)  
#0002 {  
#0003     TTimeCaps Caps;  
#0004  
#0005     // 取得並顯示多媒體計時器所支援的解析度上下極限  
#0006     if (timeGetDevCaps(&Caps, sizeof(TTimeCaps)) == TIMERR_NOERROR) {
```

```
#0007     ShowMessage("minimum supported resolution: " +
#0008     IntToStr(Caps.wPeriodMin)
#0009     + "\nmaximum supported resolution: " +
#0010     IntToStr(Caps.wPeriodMax));
#0011     } else
#0012     ShowMessage("Size of TTimeCaps structure error.");
#0013     }
```

取得解析度範圍後，在建立計時器之前，我們可以根據程式目的及需求來設定解析度。

視需求調整解析度

好比電腦螢幕，解析度當然是越高越好，看得越清楚。那為什麼不將解析度固定設為最高，而特別提供這組調整解析度的函式呢？

如果你常玩 DOS 底下的遊戲，可以發現雖然我們的顯示卡至少都會支援 1024 x 768 全彩模式，絕大部分的遊戲仍採用 320 x 200 256 色或 320 x 240 256 色（俗稱 X Mode）這兩種模式。除了架構簡單，易於實作外，最重要的原因無他：「快」。解析度一旦提高，別的不說，光是視訊緩衝區（video buffer）裡頭的資料量就大上幾倍，在多數玩家的 CPU、顯示卡不夠「有力」的情形下，效率與畫面美觀折衷考量的結果，這兩種最簡單的繪圖模式往往成了遊戲設計者的不二選擇。

多媒體計時器也是。即使明知它的能力不僅於此，只要調整至符合程式用途的需求即可。這易如折枝的設定動作即可使多媒體計時器節省大量不必要的功夫—使計時器系統輕鬆點，只要於每個解析度時間去詢問，是不是該觸發了就可以。

調整解析度後的結果—例如當解析度為 5 毫秒時，而間隔時間設定為 100 毫秒時，多媒體計時器會在 95 至 105 毫秒之間觸發。**解析度的選擇原則，就是在程式需求及使用者可接受的合理範圍內，調整至可接受的解析度就對了。**

呼叫下列函式可以調整及還原多媒體計時器的解析度：

```
MMRESULT timeBeginPeriod(  
    UINT          uPeriod  
);
```

```
MMRESULT timeEndPeriod(  
    UINT          uPeriod  
);
```

參數

uPeriod 欲設定或還原的解析度，單位為毫秒。

回返值

欲設定或還原的解析度若超出該系統所能支援的範圍（由 *timeGetDevCaps* 取得）時，會傳回 *TIMERR_NOCANDO*，否則傳回 *TIMERR_NOERROR*。

遵守這個原則：「建立計時器前呼叫 *timeBeginPeriod* 函式設定解析度；消滅該計時器後呼叫 *timeEndPeriod* 函式還原解析度」。

若欲建立多個解析度不同的多媒體計時器，遵守此原則將使計時器系統的負擔減至最少，呼叫幾次 *timeBeginPeriod* 就得有幾個 *timeEndPeriod*，別忘了我們「有借有還」的好習慣。當然啦，若所有計時器都希望使用相同的解析度時，就不必在各個計時器的前後調整、還原解析度，呼叫一次就可以了。

使用多媒體計時器

計時器型式

對我這種懶蟲來說，鬧鐘是絕對不可缺乏的生活必需品，否則上課約會出遊，我大概沒一樣不遲到的。但是，即使有時睡前辛辛苦苦跟室友借了兩三個鬧鐘來，桌上擺一個，床底擺一個，櫃子上再擺一個。鬧鐘響了，耳朵還沒聽見，人就反射性地迅速從床上跳

起，兩秒鐘內將三個鬧鐘，正在響的，還沒響的通通關掉，再立刻跳回床上繼續睡，兩三個鬧鐘依舊沒用。沒關係，舊式的鬧鐘不行，有一種新型的「死纏爛打型」鬧鐘，關掉了？沒關係，五分鐘後再響，再關再響，而且越來越大聲，吵到睡興全無，非把人叫起床不可。當然啦，這種鬧鐘對於我這種會直接把電池拆起來，或直接丟到櫃子裡去的賴床鬼還是沒用...

鬧鐘有兩種，多媒體計時器也分兩種：一種是響過就算，不管你會不會遲到；另一種較體貼，週期性地響，非把你叫起來不可。

建立及消滅

與 Windows 計時器函式不同的是，多媒體計時器不使用容易漏失的視窗訊息，它提供兩種方式來觸發一回呼函式或 event。

```
MMRESULT timeSetEvent(
    UINT          uDelay,
    UINT          uResolution,
    LPTIMECALLBACK lpFunction,
    DWORD        dwUser,
    UINT          uFlags
);
```

參數

<i>uDelay</i>	計時器觸發的間隔時間，單位為毫秒。
<i>uResolution</i>	計時器欲使用的解析度，單位為毫秒。設為 0 表示盡可能使用最高精確度，但相對地會使用較多的 CPU 資源，你必須在精確度及 CPU 資源間做取捨。
<i>lpFunction</i>	若觸發方式為回呼函式，此為指向回呼函式的指標；若採用 event 觸發方式，則為 event object 的 handle。
<i>dwUser</i>	DWORD 型態，使用者自訂資料，可為任意變數或結構位址，它會在計時器觸發時回傳給回呼函式。
<i>uFlags</i>	計時器型式，共有兩組旗標，分別為：

<i>TIME_ONESHOT</i>	只觸發一次。
<i>TIME_PERIODIC</i>	周期性觸發，直到呼叫 <i>timeKillEvent</i> 函式摧毀計時器後才能中止。
<i>TIME_CALLBACK_FUNCTION</i>	採用回呼函式觸發方式。
<i>TIME_CALLBACK_EVENT_SET</i>	採用 event 觸發方式，觸發時會將 event object 設為 signaled 狀態。
<i>TIME_CALLBACK_EVENT_PULSE</i>	採用 event 觸發方式，觸發時會將 event object 設為 non-signaled 狀態。

回返回值

成功的話，傳回非零值，代表計時器編號；否則傳回零。

LPTIMECALLBACK 的函式原型為：

```
typedef TIMECALLBACK FAR *LPTIMECALLBACK;  
  
typedef void (CALLBACK TIMECALLBACK)(UINT uTimerID, UINT uMsg,  
    DWORD dwUser, DWORD dw1, DWORD dw2);
```

不論建立的是只觸發一次或周期性觸發的計時器，嫌它吵時，趕快將電池拆下來...噢，不對，是不再需要該計時器時，請呼叫 *timeKillEvent* 消滅之。

MMRESULT timeKillEvent(

UINT uTimerID

);

參數

uTimerID 建立計時器時所傳回的計時器編號。

回返回值

若計時器編號無誤，傳回 *TIMERR_NOERROR* 表示成功消滅；否則傳回 *MMSYSERR_INVALIDPARAM*。

另外要注意的一點是，多媒體計時器的個數是有限制的，如表 4-3。能力越強，破壞力越大，使用起來也得格外小心，我們馬上可以見識到它的威力（破壞力！？）。

表 4-3 / 系統提供的多媒體計時器個數上限

作業系統	16 bit 程式	32 bit 程式
Windows 3.11	8	0 (Win32s 不支援)
Windows 95/98	32	32
Windows NT/2000	16	16 (每個程序)

使用 Event 觸發方式

大部分情況下我們會使用回呼函式觸發方式，但 event 觸發方式在多執行緒之間共享計時器時特別有用。令人不解的是，我手邊的【Win32 Developer's Reference】對於多媒體計時器的 event 觸發方式毫無著墨，以下是簡單的使用範例：

MMTIMER\UNIT1.CPP

```
#0001 int main(int argc, char* argv[])
#0002 {
#0003     int TriggerCount = 0;
#0004
#0005     // 建立 event object: automatic reset, initial non-signaled state
#0006     HANDLE hEvent = CreateEvent(NULL, false, false, NULL);
#0007
#0008     // 設立多媒體計時器，注意旗標設定
#0009     int TimerID = timeSetEvent(1000, 0, (LPTIMECALLBACK)hEvent, 0,
#0010     TIME_PERIODIC | TIME_CALLBACK_EVENT_SET);
#0011
#0012     do {
#0013         // 等待 hEvent 被設定為 signaled
#0014         WaitForSingleObject(hEvent, INFINITE);
#0015         printf("Timer Triggered: #%d\n", TriggerCount);
#0016         TriggerCount++;
#0017     } while (TriggerCount <= 10);
#0018
#0019     timeKillEvent(TimerID);
#0020     CloseHandle(hEvent);
```

```
#0021
#0022  printf("Press anykey to exit ...");
#0023  getchar();
#0024
#0025  return 0;
#0026 }
```

躍動的多環圈！！

當我第一眼瞧見 GDI 中的 *PolyBezier* 函式時，簡直是興奮極了！任意傳入四對以上的 X、Y 座標，就可以畫出千變萬化的 *Bezier* 曲線—即使完全不懂它的原理。

Bezier 曲線公式
一段 <i>Bezier</i> 曲線是由兩個端點，兩個控制點所構成。若兩端點分別為 (X0, Y0) 及 (X3, Y3)；兩控制點為 (X1, Y1) 及 (X2, Y2)，則由這四點構成的 <i>Bezier</i> 曲線公式為：
$X(T) = (1-T) * 3X0 + 3t * (1-t)2X1 + 3t^2 * (1-t)X2 + t^3X3$
$Y(T) = (1-t) * 3Y0 + 3t * (1-t)2Y1 + 3t^2 * (1-t)Y2 + t^3Y3$

PolyBezier 函式的原型如下：

```
BOOL PolyBezier(  
    HDC          DC,  
    CONST POINT* Points,  
    DWORD       Count  
);
```

```
BOOL PolyBezierTo(  
    HDC          DC,  
    CONST POINT* Points,  
    DWORD       Count  
);
```

參數

DC 欲繪製曲線的 device context。

Points	指向由 <i>TPoint</i> 結構組成的陣列指標。
Count	座標數目。因為每段 Bezier 曲線需要兩個端點、兩個控制點，若欲繪製 <i>n</i> 段曲線， <i>PolyBezier</i> 函式需要 $3n + 1$ 個座標， <i>PolyBezierTo</i> 函式需要 $3n$ 個座標。

回返值

若繪製成功，傳回 *true*，否則傳回 *false*。

在範例程式 Bezier「躍動的多環圈」中，按下【Caps】按鈕可以取得系統所支援解析度的上下限。設定好欲使用的解析度及觸發間隔時間，按下【Start】按鈕後，就可以看到主角—Bezier 曲線。你可以發現繪製出來的曲線是一個封閉的環，那是故意將兩個端點，即 *Points[0]* 及 *Points[n-1]* 設為同一座標的結果，因為我覺得，封閉曲線看起來似乎比孤零零兩個端點的曲線快樂多了。:P

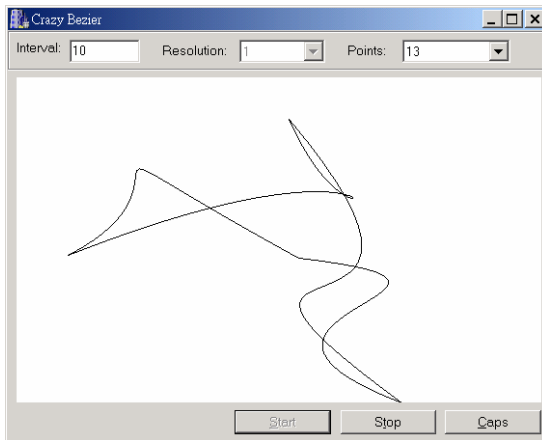


圖 4-4 / 「躍動的多環圈」程式執行畫面

```
#0001 const int SPEED = 5; // 最大移動速率
#0002
#0003 int TimerID; // 計時器編號
#0004 int PointNum; // 端點數目
#0005 TPoint Points[20], Steps[20];
#0006
#0007 int pnlWidth, pnlHeight; // 儲存 pnlCanvas 的寬度及高度
#0008 HWND pnlHandle; // 儲存 pnlCanvas 的視窗 handle
#0009
```

```

#0010 // 供多媒體計時器使用的回呼函式
#0011 void _stdcall TimerProc(UINT uTimerID, UINT uMessage, DWORD dwUser,
#0012     DWORD dw1, DWORD dw2)
#0013 {
#0014     for (int i = 0; i <= PointNum - 2; i++)
#0015     {
#0016         // 更新每個端點的座標
#0017         Points[i].x += Steps[i].x;
#0018         Points[i].y += Steps[i].y;
#0019
#0020         // 判測座標是否超出 pnlCanvas 範圍, 如果超出, 則反向其移動方向,
#0021         // 並重新以亂數取得速率
#0022         if (Points[i].x <= 0 || Points[i].x >= pnlWidth)
#0023             InvertDirection((int)Steps[i].x);
#0024
#0025         if (Points[i].y <= 0 || Points[i].y >= pnlHeight)
#0026             InvertDirection((int)Steps[i].y);
#0027     }
#0028
#0029     Points[PointNum - 1] = Points[0];
#0030
#0031     // 將 pnlCanvas 清除為白色並重新繪製 Bezier 曲線
#0032     HDC DC = GetDC(pnlHandle);
#0033     TRect R;
#0034     GetClientRect(pnlHandle, &R);
#0035     FillRect(DC, &R, GetStockObject(WHITE_BRUSH));
#0036     PolyBezier(DC, Points, PointNum);
#0037     ReleaseDC(pnlHandle, DC);
#0038 }
#0039
#0040 void __fastcall TForm1::btnStartClick(TObject *Sender)
#0041 {
#0042     // 設定所需之精確度
#0043     if (timeBeginPeriod(StrToInt(cbxResolution->Text)) !=
#0044         TIMERR_NOERROR) {
#0045         ShowMessage("Something wrong while setting timer period !!");
#0046         return;
#0047     }
#0048
#0049     btnStart->Enabled = false;
#0050     btnStop->Enabled = true;
#0051     cbxResolution->Enabled = false;
#0052
#0053     // 記下 pnlCanvas 的視窗 handle 及寬高點數
#0054     pnlHandle = pnlCanvas->Handle;
#0055     pnlWidth = pnlCanvas->ClientWidth;

```

```
#0056     pnlHeight = pnlCanvas->ClientHeight;
#0057
#0058     PointNum = StrToInt(cbxPointNum->Text);
#0059     for (int i = 0; i <= PointNum - 2; i++)
#0060     {
#0061         // 以亂數取得每個端點的初始位置
#0062         Points[i].x = random(pnlWidth);
#0063         Points[i].y = random(pnlHeight);
#0064
#0065         // 以亂數取得每個端點的移動速率及方向
#0066         if (random < 0.5)
#0067             Steps[i].x = random(SPEED) + 1;
#0068         else
#0069             Steps[i].x = - (random(SPEED) + 1);
#0070
#0071         if (random < 0.5)
#0072             Steps[i].y = random(SPEED) + 1;
#0073         else
#0074             Steps[i].y = - (random(SPEED) + 1);
#0075     }
#0076     // 最後一點與第一點重合
#0077     Points[PointNum - 1] = Points[0];
#0078
#0079     // 設定計時器，若輸入的間隔時間不合法，則設為 10 毫秒
#0080     TimerID = timeSetEvent(StrToIntDef(txtInterval->Text, 10),
#0081         StrToInt(cbxResolution->Text), TimerProc, 0, TIME_PERIODIC);
#0082 }
#0083
#0084 void __fastcall TForm1::btnStopClick(TObject *Sender)
#0085 {
#0086     // 摧毀計時器
#0087     timeKillEvent(TimerID);
#0088     // 若曾呼叫 timeBeginPeriod，記得在計時器摧毀後呼叫 timeEndPeriod，
#0089     // 並傳入相同參數
#0090     timeEndPeriod(StrToInt(cbxResolution->Text));
#0091
#0092     btnStart->Enabled = true;
#0093     btnStop->Enabled = false;
#0094     cbxResolution->Enabled = true;
#0095 }
```

內部運作

執行上節的「躍動的多環圈」程式，祭出你手上任何一個可以觀察系統中所有活動執行

緒狀態的工具，讓我們來好好觀察它，筆者使用的是 NT Resource Kit 中所附的【Process Viewer】。觀察之後可以發現，啓動計時器前，程式十分正常，只有一個主執行緒；但啓動計時器，開始繪製多環圈之後，竟然冒出另一個執行緒！

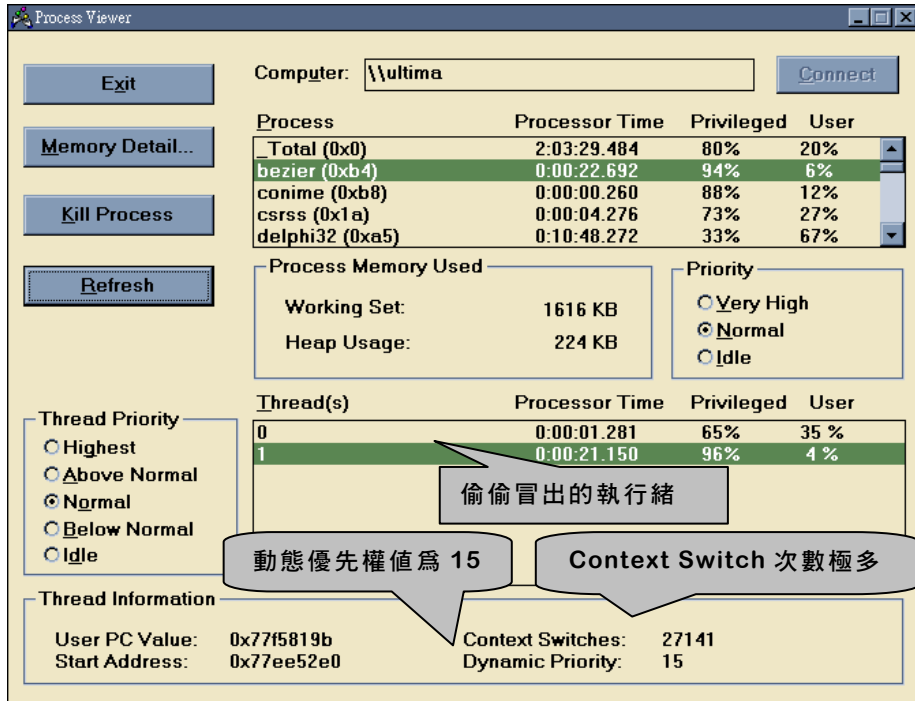


圖 4-5 / 以【Process Viewer】觀察「躍動的多環圈」

原來這就是多媒體計時器的真面目，它會在我們的程式中，另外建立一個執行緒－我們暫且稱之為「計時器執行緒」。在我們使用過多媒體計時器之後，即使將它們全部消滅，只要程序尚未結束，就一定會有一個「計時器執行緒」，不增不減，不多不少。

當任何計時器觸發時，「計時器執行緒」會暫停下來，進行context switch，切換至設定計時器的執行緒，執行回呼函式，不管原執行緒正在做什麼事⁴。很霸道，但這也是多

⁴ 較精確的說法是，如果原執行緒正在等待或處理訊息（藉由GetMessage、PeekMessage或WaitMessage）時，「計時器執行緒」才能被插斷；若目的執行緒還有其它也是由別的

媒體計時器硬是精確的原因。「計時器執行緒」還有一個十分重要的特性，它的執行緒優先層級（thread priority level）設定為`THREAD_PRIORITY_TIME_CRITICAL`，所以不論行程的優先權等級（process priority class）為何，「計時器執行緒」的優先權數值永遠為 15—這比絕大部分的執行緒優先權數值都還高，所以只要它想觸發，優先權數值較低的其它執行緒得讓出一條路來讓它過，當然不容易遲到囉！

你可以試著將間隔時間調小一點，例如 1 或 2 毫秒，看看會發生什麼狀況？在我的 Windows NT 4.0 上，只要間隔時間小於 2 毫秒，馬上吃掉所有的 CPU 資源，連【工作管理員】都無法開啓，只有關掉電源重新開機一途了。因為這個原因，在撰寫這個程式時，使得我重新開機不下五次，破壞力真是強大！將程式修改一下，試著同時產生五個間隔時間為十毫秒的計時器，下場也是一樣慘兮兮—當掉了。

Info

事實上，這個臭蟲曾在網路上聲名大噪，有位仁兄寫出一個優先權為 16 的無窮迴圈程式，立刻使號稱強固的 Windows NT 動彈不得！微軟隨後推出的 NT Service Pack 4 已經除去這個臭蟲，不過採取的解決方式是將【工作管理員】執行緒的優先權提升為 16。因此若你的 Windows NT 已安裝過 SP4，【工作管理員】的優先權比【多媒體計時器執行緒】還要高，就不會發生和我一樣的慘況。

根據微軟的文件，不論設定的解析度為多少，Win32 多媒體計時器在絕大多數情形下可以達到 10 毫秒以內的精確度，應該足夠滿足大多數程式的需求。

取得系統使用時間

除了提供多媒體計時器，多媒體子系統另外提供取得系統時間的 `timeGetSystemTime` 及 `timeGetTime` 函式。兩個函式作用相同，皆傳回系統啓動至今經過的時間，以毫秒為單位；

執行緒傳送的訊息尚未處理時，會先處理掉，再接著處理新的要求。

不同的是，`timeGetTime` 會直接傳回一個 `DWORD` 值，而呼叫 `timeGetSystemTime` 時必須傳入一個 `MMTIME` 結構指標，將結果經由此指標傳回。反正我們不是在撰寫多媒體應用程式，通常會選擇較簡單的 `timeGetTime` 函式。

前頭所介紹設定計時器解析度的 `timeBeginPeriod`、`timeEndPeriod` 函式也可用於此處，舉例來說，若你將最小解析度設為 5 毫秒，則可以保證程式中連續兩次的 `timeGetTime` 傳回值的差異一定不超過 5 毫秒。

Info

Windows NT 的預設最小精確度為 5 毫秒，可以利用 `timeBeginPeriod`、`timeEndPeriod` 函式更改；而 Windows 95 的預設最小精確度為 1 毫秒，但 `timeBeginPeriod`、`timeEndPeriod` 對 `timeGetTime`、`timeGetSystemTime` 函式沒有作用。

精益求精 – 高解析度效能計數器

如果...如果你覺得 10 毫秒等級的精確度還不足以滿足你的需求，別著急，還有好菜未上桌，請繼續往下看。

是的，Win32 裡頭還有精確度更高的計時器—「高解析度效能計數器」(high-resolution performance counter)。它只提供兩個函式可供呼叫，取得計數器頻率的 `QueryPerformanceFrequency` 及取得計數器數值的 `QueryPerformanceCounter`。

```
BOOL QueryPerformanceFrequency(  
    LARGE_INTEGER* lpFrequency  
);
```

參數

`lpFrequency` 型態為指向 `TLargeInteger` 的指標，藉此取得計數器的頻率，單位為次／每秒。

回返值

若成功傳回 *true*，否則傳回 *false*，表示該系統不支援「高解析度效能計數器」，呼叫 *GetLastError* 可以獲得更詳盡的失敗資訊。

```
BOOL QueryPerformanceFrequency(
    LARGE_INTEGER* lpPerformanceCount
);
```

參數

lpPerformanceCount 型態為指向 *TLargeInteger* 的指標，藉此取得目前的計數值。

回返值

若成功傳回 *true*，否則傳回 *false*，表示該系統不支援「高解析度效能計數器」。

Info

QueryPerformanceCounter 在 Intel x86 CPU 上擁有約 0.8 微秒的解析度；在 MIPS 上則約為 CPU 時脈的兩倍。

望名生義，這組 API 的設計原本是拿來作效能分析用的，不過我們儘管拿來用也無妨。*TLargeInteger* 型態你可能從沒見過，它是長度為八個位元組的整數，在 C++Builder 裏頭有個相對應的型態叫做 *__int64*，範圍為 2 的負 63 次方至 2 的 63 次方減 1，即 -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807，夠噲人吧！前幾天到天文館看星象，腦中浮現的卻是：*__int64* 可以容納這麼大的數目，但宇宙中的行星、恆星、慧星及流星群加起來的數目，與 *__int64* 型態的上限相較之下又如何呢？大概是好幾億兆、甚至是好幾個級數以上的倍數吧。究思之下愈覺人類及地球的渺小、穹蒼之浩瀚，還是乖乖地吃手中的熱狗，計劃明天的行程比較實在...

LARGE_INTEGER 是 Windows API 所用的結構型態，*TLargeInteger* 是 C++Builder 另行定義的另一個型別，事實上為同樣的結構，定義如下：

```
typedef __int64 LONGLONG;

typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
```

```
        LONG HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;

typedef _LARGE_INTEGER TLargeInteger;
```

LONG 是四個位元組的整數，而此處將 *LONGLONG* 定義為 `__int64`，即佔用八個位元組的整數，嗯，是挺有道理的。只是我懷疑以後出現十二個位元組甚或十六、三十二個位元組整數時，是不是也要將它們取名為 *LONGLONGLONG*、*LONGLONGLONGLONG* 或者 *VeryLong*、*UltraLong*、*SuperLong* 才合理了。

一個好消息是，由於 C++Builder 4 以後編譯器對於 `__int64` 型態有直接的支援，因此我們可由 *TLargeInteger* 型態取出宣告為 *LONGLONG* 的 *QuadPart* 欄位視為一般的整數型態來運算處理，常見的加減乘除運算皆可直接使用 +、-、*、/ 等等運算子來操作。唯一的不同是，若需將字串轉為 `__int64` 整數時，C++Builder 另外準備 *StrToInt64* 及 *StrToInt64Def* 兩道函式供我們使用。

呼叫 *QueryPerformanceFrequency* 取得計數器頻率，在我的電腦上頻率為 3579545 次/秒，亦即時隔時間約為 279×10^{-9} 次方秒，雖然比擺在原子能科學委員會裏頭的原子鐘慢多了，不過保證十分夠用了一除非你真想拿來寫個模擬原子鐘的玩意兒。

我們可以在迴圈中不斷呼叫 *QueryPerformanceCounter* 取得目前計數值並與先前取得的計數值相減，再除以頻率得到耗費的時間，如此一來便可自行實作簡單但不失準確的計時器，因為實作方法完全掌控在我們手中，所以計時器的觸發方式可以視需要使用視窗訊息、回呼函式、布林變數甚至 mutex 或 event 核心物件等等。寫法大致就像這樣：

```
#0001 void My_Timer_Procedure()
#0002 {
#0003     TLargeInteger StartTime, L;
#0004     int Freq;
#0005
#0006     // 取得頻率，並將單位轉換為 "次/毫秒"
#0007     QueryPerformanceFrequency(&L);
#0008     Freq = (int)(L.QuadPart / 1000);
#0009
```

```

#0010 // 進入迴圈前取得開始計數值
#0011 QueryPerformanceCounter(&StartTime);
#0012 do {
#0013     // 不斷取得目前計數值
#0014     QueryPerformanceCounter(&L);
#0015
#0016     /* QuadPart 欄位是 Comp 型態，因此可以直接進行加減乘除運算
#0017     計算由上回觸發到現在的時間，是否又該觸發了？ */
#0018     if ((L.QuadPart - StartTime.QuadPart) / Freq > TimerInterval) {
#0019         // 時間到，呼叫回呼函式或是其它通知動作
#0020         Timer_Triggered_Proc();
#0021         // 將開始計數值重設
#0022         StartTime = L;
#0023     }
#0024 } while (!StopCounter && !Application->Terminated); // 計時器結束
#0026 }

```

範例程式 PECounter 可以取得頻率及計數值，及簡單的計數動作，留給讀者自行參考囉！

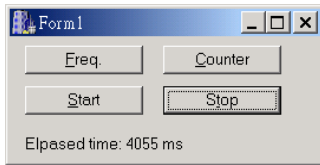


圖 4-6 / PECounter 範例程式執行畫面

延遲函式

假想今天你寫出了一個共享軟體供大眾自由下載使用，付錢註冊過的大哥及尚未註冊的試用者總要有點差別待遇，不然以後就沒人想向你註冊了。對於老是「忘記」註冊卻仍用得很高興的試用者而言，一個常見且有用的「提醒」方式即是在程式啟動時加點延遲時間，故意讓使用者多等個十秒二十秒，而且隨著使用次數的增加，延遲時間越長，不趕快寄錢來註冊的使用者就慢慢等吧，哼哼！

於是很高興地設計出開頭畫面，上頭告訴使用者：「你還沒註冊唷，趕快寄錢錢來！不然就請你等十秒鐘才能進入程式！」。

”等待十秒鐘”？這句話表示我們將要用到延遲函式，當然，也可以選擇使用計時器，延遲函式只是更方便的選擇。如果你曾在 DOS 下使用 Turbo C/C++ 或 Borland C++，一定還記得 *delay* 函式，它可讓程式延遲一段時間，十分方便。但此函式並沒有出現於 C++Builder，取而代之的是 Windows API 中的 *Sleep* 函式：

```
VOID Sleep(  
    DWORD          dwMilliseconds  
);
```

參數

dwMilliseconds 延遲時間，單位為毫秒；傳入零表示要將執行緒目前 *time slice* 所剩餘的時間禮讓給其餘相同優先權的執行緒先使用，若目前系統上沒有其它優先權相同的執行緒，則函式立即回返；傳入 *INFINITE* 表示無限等待，呼叫後將一去不復返。

看來它正是我們所要的。打開 *Form1*，建立 *OnCreate* 事件處理函式：

```
#0001 void __fastcall TForm1::FormCreate(TObject* Sender)  
#0002 {  
#0003     TForm2* frm = new TForm2(this);  
#0004     try {  
#0005         frm->Show(); // 顯示出來  
#0006         frm->Update(); // 立即更新畫面，此時在畫面上才看得到 Form2  
#0007         Sleep(10000); // 延遲十秒鐘  
#0008     } __finally {  
#0009         delete frm;  
#0010     }  
#0011 }
```

程式一執行，在尚未與 *Form1* 打過照面前，*Form2* 就無聲無息地突然出現，硬生生地擋在前頭，哇！一切順利...！怎麼？不對！似乎...好像...有點小詭異...這十秒鐘內，為什麼無法用滑鼠拉動它，不但如此，滑鼠游標還呈忙碌狀態呢？



圖 4-7 / 催促使用者快快註冊的程式開頭畫面（請注意滑鼠游標狀態）

八風請不動，只待時限到

短短幾行的程式，卻碰上兩項麻煩事，狀況說明如下，請同時想想底層的動作：

- 無法用滑鼠拖曳視窗，亦即沒有任何回應。

要讓滑鼠能夠拖曳視窗，視窗函式必須回應滑鼠動作的諸多訊息，如 `WM_NCHITTEST`、`WM_NCMOUSEMOVE`、`WM_NCLBUTTONDOWN`、`WM_MOUSEMOVE`、`WM_NCLBUTTONUP`...等等，再加上其它視窗訊息，如 `WM_WINDOWPOSCHANGED`、`WM_SYSCOMMAND`、`WM_GETMINMAXINFO`、`WM_MOVE`、`WM_PAINT`...等等，不勝枚舉，許許多多的訊息通力合作之下，視窗才能順利拖曳，才有能力回應使用者的各項操作。

- 滑鼠游標不正常，應該為正常游標 (`crDefault`)，但卻呈現忙碌游標 (`crHourGlass`)。

當沒有任何程式以 `SetCapture` 設定捕捉滑鼠動作時，每當滑鼠游標在視窗範圍內移動，系統會送出 `WM_SETCURSOR` 訊息提供程式設定游標的機會，在正常情況下，我們會在收到 `WM_SETCURSOR` 時呼叫 `SetCursor` 將游標設定為預設游標 (`crDefault`)。若不處理的話，`DefWindowProc` 會將游標設定為註冊視窗類別時所登記的游標。

所有的問題都出在視窗訊息的不正常處理。換句話說，程式對於 `send` 或 `post` 過來的視窗訊息並沒有作出應有的回應！！

看出問題了嗎？呼叫 `Sleep` 函式後，在等待回返的同時，執行緒都待在 `USER` 模組的 `Sleep` 函式中，根本沒有機會執行程式的訊息迴圈，理所當然地，無法對使用者的動作作出回應囉。

這也是為什麼在 0005 列呼叫 `Show` 函式後，下一行緊接著呼叫 `Update` 函式的原因—`Show` 函式會使視窗顯現，但它只是丟一個 `WM_PAINT` 訊息（利用 `PostMessage`）到訊息佇列中，在尚未被訊息迴圈派送，被視窗函式處理前，視窗是不會真正被畫出來的，所以必須直接呼叫 `Update` 函式，它會呼叫 `UpdateWindow` API，直接呼叫視窗函式來處理已存在的 `WM_PAINT`，此時視窗才會出現在畫面上。你可以試著把 0006 列的 `Update` 函式註解起來再執行看看，十秒鐘內，也就是 `TForm1::OnCreate` 事件處理函式返回前，是不可能看到 `Form2` 視窗的⁵。

TApplication::ProcessMessages

既然 `Sleep` 函式有這樣的特性，很顯然地它並不適合我們的需求，我們需要一個「擁有延遲時間功能並且同時不失去訊息處理能力」的延遲函式。

文章前頭提過三個可以取得系統時間或計數的函式，分別是 `GetTickCount`、`timeGetTime` 及 `QueryPerformanceCounter`。製作自己的延遲函式其實很簡單，只要不斷地呼叫 `Application->ProcessMessages` 函式，同時檢查時間是否終了就可以了。例如底下這個 `Delay` 函式：

```
#0001 void Delay(DWORD MSecs)
#0002 {
#0003     DWORD BeginTime;
```

⁵ Windows 95 及 Windows NT 有些不同。Windows 95 完全看不到，Windows NT 會有一個視窗外框顯現。


```
#0004
#0005   BeginTime = GetTickCount();
#0006   do {
#0007       Application->ProcessMessages();
#0008   } while (GetTickCount() - BeginTime < MSecs);
#0009 }
```

剛剛的範例程式中，你可以直接以 *Delay* 函式取代 *Sleep*，同時也可以將 *Update* 函式拿掉（因為很快地 *WM_PAINT* 馬上會被處理），開頭畫面就可以十分正常地顯示、運作。

訊息迴圈

話說每個擁有視窗的執行緒都必須有一個訊息迴圈，將訊息佇列中的視窗訊息一一取出，呼叫 *DispatchMessage* API 函式分派給適當的視窗函式處理，VCL 當然也不能例外。我們撰寫的程式中，背地裏確實有這麼一個訊息迴圈負責著訊息的取出及分派工作，它就包含在 *TApplication::Run* 函式中。

選取【Project / View Source】，可以看到類似的幾行程式碼：

```
#0001 WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0002 {
#0003     try
#0004     {
#0005         Application->Initialize();
#0006         Application->CreateForm(__classid(TForm1), &Form1);
#0007         Application->Run();
#0008     }
#0009     catch (Exception &exception)
#0010     {
#0011         Application->ShowException(&exception);
#0012     }
#0013     return 0;
#0014 }
```

別小看這 0007 列，也許很難令人相信，程式一開始執行時，就執行這 0007 列的 *Application->Run* 呼叫，直到程式結束前，在整個程式的生命週期裡，執行範圍從來沒有離開過這個函式哦。0007 列的函式呼叫時間就等於程式的執行時間。

```
#0001 procedure TApplication.Run;
#0002 begin
#0003   FRunning := True;
#0004   try
#0005     AddExitProc(DoneApplication);
#0006     if FMainForm <> NULL then
#0007       begin
#0008         case CmdShow of
#0009           SW_SHOWMINNOACTIVE: FMainForm.FWindowState := wsMinimized;
#0010           SW_SHOWMAXIMIZED: MainForm.WindowState := wsMaximized;
#0011         end;
#0012         if FShowMainForm then
#0013           if FMainForm.FWindowState = wsMinimized then
#0014             Minimize else
#0015             FMainForm.Visible := True;
#0016         repeat
#0017           HandleMessage
#0018         until Terminated;
#0019       end;
#0020       finally
#0021         FRunning := False;
#0022       end;
#0023     end;
```

而在此 *TApplication::Run* 函式中，0016 ~ 0018 列的 *repeat...until* 迴圈，正是我所指的訊息迴圈，除非你的程式中另有訊息迴圈，否則主執行緒中的視窗訊息，每一個訊息都是由 0017 列所呼叫的 *HandleMessage* 函式所取出、分派處理的。

跟隨著進入 *TApplication::HandleMessage* 函式：

```
#0001 procedure TApplication.HandleMessage;
#0002 var
#0003   Msg: TMsg;
#0004 begin
#0005   if not ProcessMessage(Msg) then Idle(Msg);
#0006 end;
```

唉呀，層層包裹，硬是再扯出 *TApplication::ProcessMessage* 函式來：

```
#0001 function TApplication.ProcessMessage(var Msg: TMsg): Boolean;
#0002 var
#0003   Handled: Boolean;
#0004 begin
#0005   Result := False;
#0006   if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then // 取出訊息
```

```

#0007   begin
#0008     Result := True;
#0009     if Msg.Message <> WM_QUIT then // 程式是否結束 ?
#0010       begin
#0011         Handled := False;
#0012         if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
#0013         if not IsHintMsg(Msg) and not Handled and not IsMDIMsg(Msg)
#0014           and not IsKeyMsg(Msg) and not IsDlgMsg(Msg) then
#0015           begin
#0016             TranslateMessage(Msg); // 轉譯鍵盤訊息
#0017             DispatchMessage(Msg); // 分派至視窗函式處理
#0018           end;
#0019         end
#0020       else
#0021         FTerminate := True;
#0022       end;
#0023     end;

```

TApplication::ProcessMessage 函式就是我們的目的地，也就是訊息迴圈的大黑手，它會從訊息佇列中取出一個視窗訊息來分派處理，若目前沒有視窗訊息等待處理，則傳回 *False*。

而 *TApplication::ProcessMessage* 函式就是當訊息佇列空空的時候，趁此機會處理提示視窗，並觸發 *TApplication::OnIdle* 事件，若還是沒事，就呼叫 *WaitMessage* API 函式等待下一個視窗訊息的到來。

說到這兒，怎麼還沒見到 *TApplication::ProcessMessages* 函式呀？原來它就在 *TApplication::ProcessMessage* 函式的隔壁。

```

#0001 procedure TApplication.ProcessMessages;
#0002   var
#0003     Msg: TMsg;
#0004   begin
#0005     while ProcessMessage(Msg) do {loop};
#0006   end;

```

原來它這麼簡單，就是一個迴圈，與訊息迴圈一樣，在迴圈內不斷呼叫 *ProcessMessage* 函式來取出分派訊息，並在訊息佇列內的訊息處理完畢後立即返回。所以能夠暫時性地取代訊息迴圈的工作，不讓視窗訊息等待過久，都沒有人來處理。

精確的延遲函式

雖說做人不要太愛鑽牛角尖，凡事太愛吹毛求疵，有些事情還是細心一點好。與計時器相同，雖然傳入的間隔時間單位為毫秒，若說傳入十毫秒就延遲十毫秒，傳入一毫秒就延遲一毫秒，讓人不禁懷疑：真的是這樣嗎？

在分析 *Sleep* 函式的精確度之前，我再提供三個自製的延遲函式：

1. *GetTickCount* 版本

2. *QueryPerformanceCounter* 版本

QueryPerformanceCounter 版本是第一回出現，因為 `__int64` 型態需要較多的運算時間，所以必須將運算動作盡量簡化。

3. *QueryPerformanceCounter* 校正 (calibrated) 版本

QueryPerformanceCounter 校正版本新增一個 *CalibratePerformanceCounterOverhead* 函式，沒什麼技術，純粹呼叫一千次 *QueryPerformanceCounter*，取得總共花費時間再求取平均值。

因為在 *QueryPerformanceCounter* 版本中，進入函式主迴圈之前我們的準備動作包括各呼叫一次 *QueryPerformanceFrequency* 及 *QueryPerformanceCounter* 函式，這兩個函式比較耗時，所以最好將它們所花費的時間扣除。你可以看到程式第 0061 列，*MyDelay3* 函式中，主迴圈之前將延遲時間減掉呼叫時間的兩倍。

```
#0001 TLargeInteger PerformanceCounterOverhead;
#0002
#0003 // MyDelay1 - use GetTickCount
#0004 void _stdcall MyDelay1(DWORD MSecs)
#0005 {
#0006     DWORD BeginTime;
#0007
#0008     BeginTime = GetTickCount();
#0009     do {
#0010     } while (GetTickCount() - BeginTime <= MSecs);
#0011 }
#0012
#0013 // MyDelay2 - use QueryPerformanceCounter
#0014 void _stdcall MyDelay2(DWORD MSecs)
```

```
#0015 {
#0016     TLargeInteger rStart, rEnd, rFreq;
#0017
#0018     // 取得頻率
#0019     QueryPerformanceFrequency(&rFreq);
#0020     rEnd.QuadPart = MSecs * rFreq.QuadPart / 1000;
#0021
#0022     // 進入迴圈前取得開始計數值
#0023     QueryPerformanceCounter(&rStart);
#0024     rEnd.QuadPart += rStart.QuadPart;
#0025
#0026     // 主迴圈
#0027     do {
#0028         QueryPerformanceCounter(&rFreq); // 取得目前計數值
#0029     } while (rFreq.QuadPart <= rEnd.QuadPart);
#0030 }
#0031
#0032 // Calls the performance counter to determine the time overhead
#0033 TLargeInteger CalibratePerformanceCounterOverhead()
#0034 {
#0035     TLargeInteger rStart, rEnd;
#0036
#0037     QueryPerformanceCounter(&rStart);
#0038
#0039     for (int i = 1; i <= 1000; i++)
#0040         QueryPerformanceCounter(&rEnd);
#0041
#0042     // 取得每次呼叫 QueryPerformanceCounter 函式所花費時間之平均值
#0043     TLargeInteger r;
#0044     r.QuadPart = rEnd.QuadPart - rStart.QuadPart;
#0045     r.QuadPart /= 1000;
#0046     return r;
#0047 }
#0048
#0049 // MyDelay3 - use QueryPerformanceCounter with Calibration
#0050 void _stdcall MyDelay3(DWORD MSecs)
#0051 {
#0052     TLargeInteger rStart, rEnd, rFreq;
#0053
#0054     // 取得頻率
#0055     QueryPerformanceFrequency(&rFreq);
#0056     rEnd.QuadPart = MSecs * rFreq.QuadPart / 1000;
#0057
#0058     // 進入迴圈前取得開始計數值
#0059     QueryPerformanceCounter(&rStart);
#0060     rEnd.QuadPart = rEnd.QuadPart + rStart.QuadPart
```

```

#0061     - PerformanceCounterOverhead.QuadPart * 2;
#0062
#0063     // 主迴圈
#0064     do {
#0065         QueryPerformanceCounter(&rFreq); // 取得目前計數值
#0066     } while (rFreq.QuadPart <= rEnd.QuadPart);
#0067 }
#0068
#0069 void __fastcall TForm1::btnStartClick(TObject *Sender)
#0070 {
#0071     const AnsiString DescStr[4] = {"Win32 API Sleep",
#0072     "GetTickCount version",
#0073     "QueryPerformanceCounter version",
#0074     "Calibrated QueryPerformanceCounter version"};
#0075
#0076     // 轉換測試次數, 預設值為 100 次
#0077     int TryTimes = StrToIntDef(txtTimes->Text, 100);
#0078     txtTimes->Text = IntToStr(TryTimes);
#0079
#0080     // 轉換延遲時間, 預設值為 100 毫秒
#0081     int Delaytime = StrToIntDef(txtDelay->Text, 100);
#0082     txtDelay->Text = IntToStr(Delaytime);
#0083
#0084     for (int TestKind = 0; TestKind < 4; TestKind++) {
#0085
#0086         void _stdcall (*DelayProc)(DWORD MSecs);
#0087
#0088         // 將 DelayProc 指向所要測試的函式
#0089         switch (TestKind) {
#0090             case 0: DelayProc = Sleep;
#0091                 break;
#0092             case 1: DelayProc = MyDelay1;
#0093                 break;
#0094             case 2: DelayProc = MyDelay2;
#0095                 break;
#0096             case 3: DelayProc = MyDelay3;
#0097                 break;
#0098         }
#0099
#0100         TLargeInteger rStart, rEnd, rFreq;
#0101
#0102         // 記錄開始時間
#0103         QueryPerformanceCounter(&rStart);
#0104
#0105         for (int TryNo = 1; TryNo <= TryTimes; TryNo++)
#0106

```

```

#0107     DelayProc(Delaytime);
#0108
#0109     // 記錄結束時間並計數所花費時間，單位為毫秒
#0110     QueryPerformanceCounter(&rEnd);
#0111     QueryPerformanceFrequency(&rFreq); // 取得頻率
#0112     float ElapsedTime = (rEnd.QuadPart - rStart.QuadPart) /
#0113         (rFreq.QuadPart / 1000);
#0114
#0115     mmoResult->Lines->Add(
#0116         Format("%s\r\n 花費時間: %f 毫秒\r\n 誤差: %f 毫秒\r\n",
#0117             OPENARRAY(TVarRec, (DescStr[TestKind], ElapsedTime,
#0118                 ElapsedTime - Delaytime * TryTimes)))));
#0119     }
#0120 }
#0121
#0122 void __fastcall TForm1::FormCreate(TObject *Sender)
#0123 {
#0124     // Calibration for MyDelay3 function
#0125     PerformanceCounterOverhead =
#0126         CalibratePerformanceCounterOverhead();
#0127 }

```

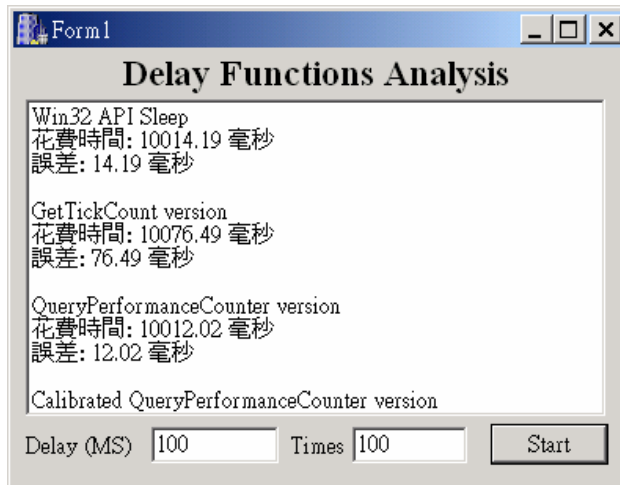


圖 4-8 / Delay Functions Analysis 程式執行畫面

設定延遲時間及測試次數後，程式分別呼叫四個延遲函式，為了減少程式碼及判斷動作，我使用函式指標變數 *DelayProc*，分別指向四個函式。0086 列將 *DelayProc* 宣告為下列型態：

```
void _stdcall (*DelayProc)(DWORD MSecs);
```

細心點的話，可以發現 *MyDelay1*、*MyDelay2*、*MyDelay3* 三個函式也都宣告為 *stdcall* 呼叫慣例 (calling convention)，這是為了搭配 *Sleep* 函式所做的，因為所有的 Windows API 函式皆使用 *stdcall* 呼叫習慣。

測試結果十分出人意料，有幾點有趣的現象可由表中呼之欲出，下面將結果分為 Windows 95 及 Windows NT 兩平臺來討論：

表 4-9 / Windows 95 下，四種延遲函式平均誤差時間 (單位為毫秒)

延遲 \ 函式	<i>Sleep</i>	<i>GetTickCount</i>	<i>Query</i>	<i>Calibrated Query</i>
1 毫秒	12.421	12.729	0.011	-0.004
10 毫秒	3.752	3.706	0.015	-0.002
100 毫秒	9.762	9.826	0.012	-0.003
1000 毫秒	2.764	1.656	0.014	-0.003
10000 毫秒	9.412	9.587	0.012	-0.002

1. *Sleep* 及 *GetTickCount* 函式對於 10 毫秒以下的延遲時間毫無辦法，誤差高得離譜；且在 1000 毫秒延遲時間時誤差特別小，十分有趣的現象，值得再進一步探討。
2. *QueryPerformance* 系列函式於各延遲時間誤差皆小於 0.015 毫秒，十分穩定。
3. 修正版的 *QueryPerformance* 函式於各延遲時間誤差皆為負值，表示「修正過頭」，意謂修正函式仍有改良空間。

表 4-10 / Windows NT 下，四種延遲函式平均誤差時間 (單位為毫秒)

延遲 \ 函式	<i>Sleep</i>	<i>GetTickCount</i>	<i>Query</i>	<i>Calibrated Query</i>
1 毫秒	9.005	9.014	0.025	0.016
10 毫秒	-0.00025	0.00006	0.00003	0.00001
100 毫秒	0.136	0.144	0.026	0.011
1000 毫秒	1.439	1.440	0.032	0.014
10000 毫秒	4.639	4.381	0.500	0.280

1. *Sleep* 及 *GetTickCount* 對於 1 毫秒的延遲時間根本無能為力，誤差高達 9 毫秒。
2. 10 毫秒似乎是「Magic Time Period」，四種延遲函式對 10 毫秒間隔時間的誤差皆遠小於其它間隔時間。
3. 呼叫 *Sleep* 延遲 10 毫秒的誤差竟然是負值，再觀察 1 毫秒間隔時間的誤差，可以由此看出 *Sleep* 函式的精確度約為 10 毫秒，但略小於 10 毫秒。

而兩個平臺也有不少相同的特點：

1. *Sleep* 與 *GetTickCount* 於不同間隔時間的誤差皆十分接近，表示內部似乎由同一個機制運作。
2. 修正版的 *QueryPerformanceCounter* 無論在何種情況下皆比原版本精確，證明修正動作是值得的。
3. 對於所有的間隔時間，誤差值約為「*Sleep* \approx *GetTickCount* \gg *QueryPerformanceCounter* $>$ *Calibrated QueryPerformanceCounter*」。

擁有這些會說話的數據，日後撰寫程式時，相信選擇合適的延遲或計時函式對你來說已不再是個難題。

TTimer 元件

VCL 的 *TTimer* 元件大概是所有元件使用最容易的一個，四個屬性加上一個事件，唔，要是所有元件都這麼簡單的話，我想我們可以試著來推動「三百萬人學 C++Builder」運動了。

由於 *TTimer* 純粹只是 *SetTimer* 及 *KillTimer* 函式的包裝，因此也繼承了所有的特性，以及精確度的問題。但因其簡易性及低負擔，在絕大多數精確度並不是那麼重要的場合，*TTimer* 元件是極佳的選擇。

***TTimer* Class**

Unit	ExtCtrls
Ancestor	TComponent
Description	將 API 中的「計時器函式」 <i>SetTimer</i> 及 <i>KillTimer</i> 包裝起來。
Usage	將 <i>Enabled</i> 屬性設為 <i>true</i> 後開始計時，每隔 <i>Interval</i> 時間（單位為毫秒）會觸發 <i>OnTimer</i> 事件。

重要屬性

<i>Interval</i>	間隔時間（單位為毫秒），最大為 4294967295，約為 49.7 天。
<i>Enabled</i>	控制計時器的啟動及結束。

重要事件

<i>OnTimer</i>	計時器啟動後，每經過間隔時間所觸發的事件。
----------------	-----------------------

內部剖析

前頭提到，*SetTimer* 函式可有兩種觸發方式，一為訊息 *WM_TIMER*，另一為回呼函式。回呼函式必須是一般函式型態，不可以是類別或物件的成員函式，所以若以回呼函式觸發方式來實作，勢必又得跟 VCL 包裝視窗函式的方法一樣，花費好大一番功夫才行，因此 VCL 設計小組決定採用接收 *WM_TIMER* 訊息的設計。

問題來了，*TTimer* 繼承自 *TComponent* 類別，並不具視窗，遑論視窗 *handle* 及視窗函式；沒有視窗 *handle*，如果呼叫 *SetTimer* 時視窗 *handle* 指定為零，即使系統將 *WM_TIMER* 丟進訊息佇列，*TApplication* 中的訊息迴圈⁶ 也成功地提取出來後，*DispatchMessage* 該呼叫哪個視窗的視窗函式呢？沒概念耶。呃，不是我沒有概念，是資料不足，*DispatchMessage* 面對視窗 *handle* 為零的訊息結構該怎麼處理呢？無能為力，愛莫能助，只好不理它囉。

沒關係，這不打緊，兵來將擋，水來土掩，咱們有的是辦法。VCL 的 *Forms* 單元提供一組 *AllocateHWND / DeallocHWND* 函式專來對付這種情況。簡單地說，傳入一個視窗函式，

⁶ 指的就是 *TApplication::ProcessMessage* 函式。

AllocateHWND 會為你建立一個隱形視窗，並傳回它的視窗 *handle*，你就可以利用它來接收及回應視窗訊息。

TTimer 能夠順利運作的關鍵就在這兒：

```
#0001 constructor TTimer.Create(AOwner: TComponent);
#0002 begin
#0003     inherited Create(AOwner);
#0004     ...
#0005     FWindowHandle := AllocateHWND(WndProc);
#0006 end;
#0007
#0008 destructor TTimer.Destroy;
#0009 begin
#0010     ...
#0011     DeallocateHWND(FWindowHandle);
#0012     inherited Destroy;
#0013 end;
#0014
#0015 procedure TTimer.WndProc(var Msg: TMessage);
#0016 begin
#0017     with Msg do
#0018         if Msg = WM_TIMER then
#0019             try
#0020                 Timer;
#0021             except
#0022                 Application.HandleException(Self);
#0023             end
#0024         else
#0025             Result := DefWindowProc(FWindowHandle, Msg, wParam,
#0026                 lParam);
#0027     end;
#0028
#0029 procedure TTimer.Timer;
#0030 begin
#0031     if Assigned(FOnTimer) then FOnTimer(Self);
#0032 end;
```

第 0020 列，我們可以見到每當收到 *WM_TIMER* 訊息時，就呼叫 *Timer* 函式，而 *Timer* 函式只有短短一行：呼叫程式員設定的 *OnTimer* 事件處理函式。

每當 *Enabled*、*Interval* 屬性或 *OnTimer* 事件改變時，就會呼叫 *UpdateTimer*，首先利用 *KillTimer* 摧毀計時器，若需要的話，再重新建立：

```
#0001 procedure TTimer.UpdateTimer;
#0002 begin
#0003   KillTimer(FWindowHandle, 1);
#0004   if (FInterval <> 0) and FEnabled and Assigned(FOnTimer)
#0005   then
#0006     if SetTimer(FWindowHandle, 1, FInterval, NULL) = 0 then
#0007       raise EOutOfResources.Create(SNoTimers);
#0008 end;
```

這兒有個小毛病，當`Enabled`屬性為`true`時，更改`Interval`屬性也會迫使`UpdateTimer`先呼叫`KillTimer`摧毀計時器再呼叫`SetTimer`建立計時器。事實上，若要更改間隔時間，可以重覆呼叫`SetTimer`，傳入相同的視窗`handle`、計時器編號及新的間隔時間即可 — 只需更改大樓的外觀顏色，不需整棟拆掉再重建⁷。

執行緒中的計時器

所有的執行緒大致上可分為兩類：

- 不具視窗、訊息佇列及訊息迴圈的工作執行緒，它的任務通常是費時冗長的資料萃取、計算或等待等動作，不需要外界（使用者）的互動便能運作，運算結束或結果取得後即停止執行，將結果交還主執行緒。
- 具有視窗、訊息佇列及訊息迴圈的 GUI 執行緒，它的任務可能也與主執行緒相同，負責處理某些視窗的視窗訊息；也可能與工作執行緒相同。

工作執行緒的特點是「無暇處理視窗訊息，它有它自己專心應付的事務」。但是，對於`SetTimer`、`KillTimer`計時器函式，以及使用它們的`TTimer`元件來說，訊息佇列及處理、分派視窗訊息的訊息迴圈是正常運作的必要條件，所以得到的初步結論是：在工作執行

⁷ 話是這麼說，不過爲了克服設計上的困難度，光是更改顏色式樣就必須先摧毀再重建的例子在SDK中屢見不鮮。例如若要更改`edit`控制元件的文字對齊方式（`alignment`）、邊框樣式（`border styles`）或捲軸（`scroll bars`）樣式，唯一的方法就是先摧毀它，再利用新的樣式重新建立一個。

緒中無法使用計時器函式及 *TTimer* 元件。

工作執行緒隱含的陷阱

上述的結論只是初步的結論，事實上，你還是可以在執行緒內使用計時器函式及 *TTimer* 元件的，只是有很多陷阱及問題要小心避開，一一克服。

假設我現在希望撰寫一個多執行緒的伺服器程式，除了主執行緒，另外建立一工作執行緒來等待客戶端的請求，同時每三十秒測試目前客戶端的連線狀態。這當然就需要計時器的支援了。

若 Unit1 單元為 main form，Unit2 單元為執行緒類別，程式碼大略會是這樣：

UNIT1.CPP

```
#0001 void __fastcall TForm1::Button1Click(TObject* Sender)
#0002 {
#0003     new TTimerThread(); // 建立 TTimerThread 物件，同時開始執行
#0004 }
```

UNIT2.CPP

```
#0001 class TTimerThread: public TThread {
#0002 private:
#0003     TTimer* FTimer;
#0004
#0005     void __fastcall TimerOnTimer(TObject* Sender);
#0006 protected:
#0007     void __fastcall Execute();
#0008 public:
#0009     __fastcall TTimerThread(bool CreateSuspended);
#0010     __fastcall ~TTimerThread();
#0011 }
#0012
#0013 // TTimerThread
#0014
#0015 __fastcall TTimerThread():TTimerThread()
#0016     : TThread(CreateSuspended)
#0017 {
```

```
#0018   FTimer = new TTimer();
#0019   FTimer->Interval = 30 * 1000; // 30 秒
#0020   FTimer->OnTimer = TimerOnTimer; // 指定事件處理函式
#0021   FTimer->Enabled = true;
#0022 }
#0023
#0024 __fastcall TTimerThread::~TTimerThread()
#0025 {
#0026     delete FTimer;
#0027 }
#0028
#0029 void __fastcall TTimerThread::TimerOnTimer(TObject* Sender)
#0030 {
#0031     ... // 每三十秒觸發一次
#0032 }
```

很直覺地，我們會分別在 *TTimerThread* 類別的建構函式及解構函式中，分別建立及摧毀 *TTimer* 物件，並為 *TTimer::OnTimer* 事件指定其事件處理函式，接收每三十秒觸發一次的事件。

你看得出來，這一段怎麼看都沒問題的程式碼犯了什麼錯誤嗎？請多想一會兒再往下看解答。

宣告答案了。視窗是由執行緒擁有的，我們知道 *TTimer* 元件會分別在建立及摧毀時呼叫 *Forms* 單元的 *AllocateHwnd* 及 *DeallocateHwnd* 函式來建立及摧毀一個專門用來處理計時器訊息的隱形視窗，且我們分別在執行緒的建構及解構函式中建立及摧毀 *TTimer* 元件，如下圖流程：

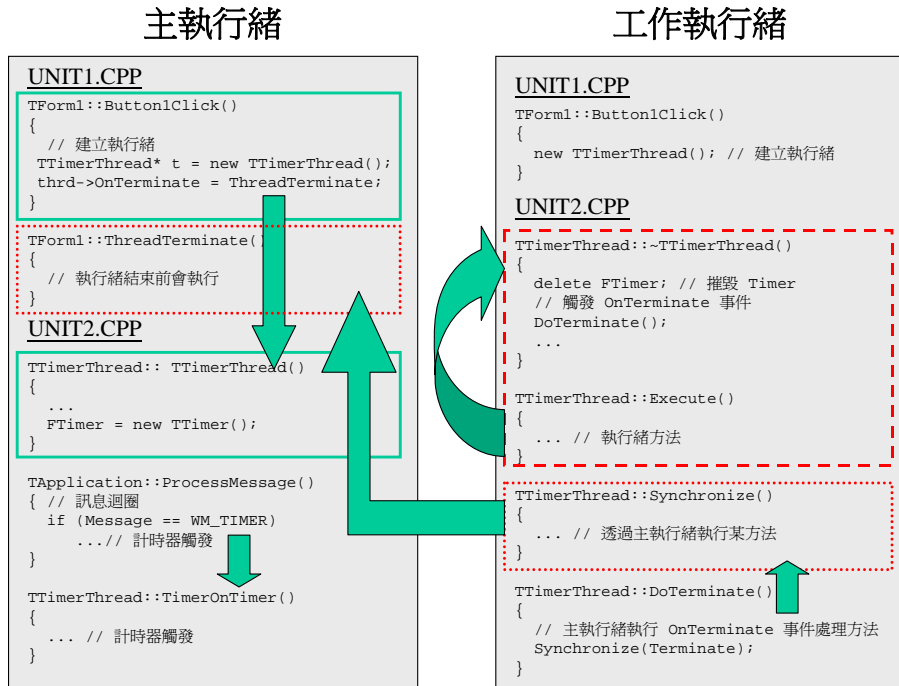


圖 4-11 / 在工作執行緒中使用 TTimer 元件的程序流程

由以上的流程圖，可以歸納出下列現象：

- *TThread* 的建構函式是由建立 *TThread* 物件的原執行緒執行的，而不是由新的執行緒執行。
- *TThread* 的解構函式是由 *TThread* 物件本身的執行緒執行的，與執行建構函式的執行緒不同。
- *TThread* 的 *OnTerminate* 事件處理函式是經由 *Synchronize* 呼叫，切換至主執行緒執行的。

因此，在 *TTimerThread* 建構函式建立的視窗，由於由不同執行緒執行，是無法成功地在 *TTimerThread* 解構函式內被摧毀的。

雖然按照上述作法在執行緒內使用 *TTimer* 元件，你可能絲毫感覺不出有哪裏不對勁，這

是因為在 *DeallocateHwnd* 函式中，呼叫 *DestroyWindow* API 函式時並未檢查返回值，所以我們從未察覺這類型隱含的錯誤。事實上，*TTimer* 元件所使用的視窗並未成功歸還給系統。若程式中十分頻繁地建立、摧毀 *TTimerThread* 物件，過不久可能就會出現系統資源不足的錯誤訊息，讓人完全不知所以然。

解決的方法很簡單：*TTimer* 元件在哪個執行緒產生的，就在那個執行緒摧毀它即可。

TThread 類別執行它的 *OnTerminate* 事件處理函式時，會切換至主執行緒執行，所以我們可以依賴這個特性：在 *TThread* 建構函式內建立 *TTimer* 元件，然後在 *TThread::OnTerminate* 事件處理函式中摧毀它，這個問題便可獲得初步的解決。

撿到便宜的 TThread 建構函式

你有沒有發現，前頭大聲疾呼的訊息佇列及訊息迴圈問題，怎麼都沒有發生？在 *TThread* 建構函式內建立 *TTimer* 物件後，還不是照常使用？

這正是因為 *TThread* 建構函式是由主執行緒執行的，所以 *TTimer* 元件的視窗訊息由主執行緒負責，換句話說，我們正好可以搭順風車，由包含在 *TApplication::Run* 函式內的訊息迴圈一併為我們處理送給此 *TTimer* 元件的 *WM_TIMER* 訊息。算是撿到便宜了，很輕鬆地就避開訊息迴圈的問題。

不過...前提是，*TThread* 物件必須由主執行緒產生，要是某個執行緒在工作期間也建立了一些需要使用計時器的工作執行緒，就沒便宜好撿了。若你的 *TThread* 物件不是由主執行緒產生的，那麼它...

- 建構函式內產生的 *TTimer* 元件將無法順利運作，因為沒有人可以幫它處理 *WM_TIMER* 視窗訊息了。
- 不能在 *OnTerminate* 事件處理函式摧毀 *TTimer* 元件，因為 *TTimer* 元件是由另一個工作執行緒產生，而非主執行緒產生。

解決工作執行緒的計時需求

看來主執行緒不太可靠，視窗訊息由誰來處理仍然問題重重，事到如今，必得提出放諸四海皆準的解決方案。

方案一：借助主執行緒的訊息迴圈

方法很簡單，既然不確定產生新執行緒的究竟是哪個執行緒，那麼就自力救濟，每當建立或摧毀 *TTimer* 元件時，就切換至主執行緒執行，讓主執行緒來處理視窗訊息就對了。

```
#0001 class TWorkerThread : public TThread {
#0002 private:
#0003     TTimer* FTimer;
#0004
#0005     void __fastcall CreateTimerMethod(); // 建立 TTimer 物件
#0006     void __fastcall DestroyTimerMethod(); // 摧毀 TTimer 物件
#0007
#0008     void __fastcall TimerOnTimer(TObject* Sender);
#0009 protected:
#0010     void __fastcall Execute();
#0011 public:
#0012     __fastcall TWorkerThread(bool CreateSuspended);
#0013     __fastcall ~TWorkerThread();
#0014 }
#0015
#0016 // TWorkerThread
#0017
#0018 __fastcall TWorkerThread()::TWorkerThread()
#0019     : TThread(CreateSuspended)
#0020 {
#0021     // 從主執行緒建立視窗
#0022     Synchronize(CreateTimerMethod);
#0023 }
#0024
#0025 __fastcall TWorkerThread::~~TWorkerThread()
#0026 {
#0027     // 從主執行緒摧毀視窗
#0028     Synchronize(DestroyTimerMethod);
#0029 }
```

```
#0030
#0031 void __fastcall TWorkerThread::CreateTimerMethod()
#0032 {
#0033     FTimer = new TTimer();
#0034     FTimer->Interval = 30 * 1000; // 30 秒
#0035     FTimer->OnTimer = TimerOnTimer; // 指定事件處理函式
#0036     FTimer->Enabled = true;
#0037 }
#0038
#0039 void __fastcall TWorkerThread::DestroyTimerMethod()
#0040 {
#0041     delete FTimer;
#0042 }
#0043
#0044 void __fastcall TWorkerThread::TimerOnTimer(TObject* Sender)
#0045 {
#0046     ... // 每三十秒觸發一次
#0047 }
```

你可以看到與正常錯誤版本唯一的不同就是，將 *TTimer* 元件的建立及摧毀動作置於兩個不需參數的函式，分別在 *TThread* 的建構及解構函式中，交由 *TThread::Synchronize* 函式執行之。*TThread::Synchronize* 函式的特性是，會將指定的 *TThread* 的函式，切換至主執行緒的身份來執行，這就達成 *TTimer* 元件的生滅皆由同一個具有訊息迴圈的執行緒執行的目的。

方案二：使用不依賴視窗訊息的多媒體計時器

第二個方案較為偷懶，直接迴避關於視窗訊息的所有問題，改用不依賴視窗訊息的多媒體計時器即可。由於多媒體計時器支援 *event* 的觸發通知方式，因此對於等待事件型的工作執行緒再適合也不過了。

例如，*TWaitThread*執行緒的任務是監看應用程式目錄下是否有任何檔案變更，每當任何檔案變動時，就會觸發*hFileChangeEvent*事件。通常我們會使用*WaitXXXX*一系列的核物件⁸等待函式來等待核物件的觸發，在這裡用的是等待單一核物件的

⁸ 核物件 (kernel object) 指的是 *event*、*mutex*、*semaphore*、*process*、*thread* 等等由系

WaitForSingleObject API函式：

```
#0001 void __fastcall TWaitThread::Execute()
#0002 {
#0003     do {
#0004         // 等待 hFileChangeEvent 被設定為 signaled
#0005         WaitForSingleObject(hFileChangeEvent, INFINITE);
#0006
#0007         ... // 對變更檔案進行處理...
#0008     } while (!Terminated);
#0009 }
```

若欲加入計時器的支援，除了原本的 *hFileChangeEvent* 事件，只要將 *WaitForSingleObject* 函式改為 *WaitForMultipleObjects* 函式，再加上另一個 *hTimerEvent* 事件同時等待，連程式架構都不必更動，就可以放心地擁有計時能力了：

```
#0001 void __fastcall TWaitThread::Execute()
#0002 {
#0003     int TimerID;
#0004     HANDLE Events[2]; // 要等待的 events 陣列
#0005
#0006     // 設立多媒體計時器
#0007     TimerID = timeSetEvent(1000, 0, (TFNTimeCallBack)hTimerEvent, 0,
#0008     TIME_PERIODIC | TIME_CALLBACK_EVENT_SET);
#0009
#0010     Events[0] = hFileChangeEvent;
#0011     Events[1] = hTimerEvent;
#0012
#0013     do {
#0014         // 等待任何一個 event 被設定為 signaled
#0015         switch (WaitForMultipleObjects(2, Events, false, INFINITE)) {
#0016             case WAIT_OBJECT_0: ... // 檔案變更
#0017                 break;
#0018
#0019             case WAIT_OBJECT_0 + 1: ... // 計時器觸發
#0020                 break;
#0021
#0022         }
#0023     } while (!Terminated);
#0024 }
```

統核心管理的「物件」。核心物件最大的特徵是它們的擁有者是系統核心，而不是建立它們的行程。

```
#0025   timeKillEvent(TimerID);
#0026 }
```

不必依賴視窗訊息，多媒體計時器也可以運作得好好的，若你也想為執行緒加入計時能力時，不妨暫時丟下 *TTimer* 元件，使用多媒體計時器來試試。

方案三：使用可等待計時器

除了多媒體計時器，其實還有另外一種計時器也不需要視窗函式的配合才能啟用，它叫做可等待計時器（waitable timer）。與計時器函式及多媒體計時器最大的不同是，可等待計時器為核心物件，擁有權為系統核心，所以可以跨行程使用。這樣的好處是，它可以同時觸發多個行程中多個等待此計時器的執行緒，而這是計時器函式及多媒體計時器辦不到的。

不過很可惜的是，可等待計時器只被 Windows NT 所支援，Windows 95/98 都不能使用這項能力。它的使用方法十分複雜，可能牽涉到 APC（asynchronous procedure call），已經超出本章預定範圍，所以在此我並不打算詳細介紹它。請見下列範例，瞭解可等待計時器的用途及大致的叫用方法：

```
#0001 void __fastcall TWaitThread::Execute()
#0002 {
#0003     HANDLE hTimer;
#0004     __int64 DueTime;
#0005
#0006     HANDLE Events[2]; // 要等待的 events 陣列
#0007
#0008     // 建立可等待計時器核心物件
#0009     hTimer = CreateWaitableTimer(NULL, false, NULL);
#0010
#0011     // 第一次觸發時間：5 秒後
#0012     DueTime = -5000;
#0013     // 每間隔五秒鐘觸發一次
#0014     SetWaitableTimer(hTimer, DueTime, 5000, NULL, NULL, false);
#0015
#0016     Events[0] = hFileChangeEvent;
#0017     Events[1] = hTimerEvent;
#0018 }
```

```

#0019 do {
#0020 // 等待任何一個 event 被設定為 signaled
#0021 switch (WaitForMultipleObjects(2, Events, false, INFINITE)) {
#0022     case WAIT_OBJECT_0: ... // 檔案變更
#0023         break;
#0024
#0025     case WAIT_OBJECT_0 + 1: ... // 計時器觸發
#0026         break;
#0027
#0028     }
#0029 } while (!Terminated);
#0030
#0031 // 關閉可等待計時器
#0032 CloseHandle(hTimer);
#0033 }

```

方案四：使用可接受訊息的等待函式

前兩個方案皆呼叫 *WaitForMultipleObjects* 函式來同時等待多個 event 的觸發，因為執行緒大部分時間皆「卡」在此等待函式，所以無法另闢訊息迴圈來取得視窗訊息，得知計時器函式的觸發事件（即 *WM_TIMER* 訊息）。

不過有一個函式可以解決這個問題，它是 *WaitForMultipleObjects* 函式的加強版—*MsgWaitForMultipleObjects* 函式。*MsgWaitForMultipleObjects* 函式多了一個 *DWORD* 參數 *dwWakeMask*，用來指定欲接收的視窗訊息種類，它可以是下列旗標的任意組合：

表 4-12 / *dwWakeMask* 參數的所有旗標

旗標	含意
<i>QS_ALLINPUT</i>	所有在訊息佇列中的訊息。
<i>QS_HOTKEY</i>	接收 <i>WM_HOTKEY</i> 訊息。
<i>QS_INPUT</i>	所有的輸入裝置（鍵盤、滑鼠等等）訊息。
<i>QS_KEY</i>	<i>WM_KEYDOWN</i> 、 <i>WM_KEYUP</i> 、 <i>WM_SYSKEYDOWN</i> 等鍵盤按鍵訊息。
<i>QS_MOUSE</i>	所有的滑鼠相關訊息。
<i>QS_MOUSEBUTTON</i>	<i>WM_LBUTTONDOWN</i> 、 <i>WM_LBUTTONUP</i> 等滑鼠鈕操作訊息。

<i>QS_MOUSEMOVE</i>	接收滑鼠移動訊息。
<i>QS_PAINT</i>	接收 <i>WM_PAINT</i> 訊息。
<i>QS_POSTMESSAGE</i>	所有在訊息佇列中，但不屬於以上旗標的訊息。
<i>QS_SENDMESSAGE</i>	從其它執行緒傳送過來的訊息。
<i>QS_TIMER</i>	<i>WM_TIMER</i> 訊息。

只要指定適當的 *dwWakeMask* 參數，*MsgWaitForMultipleObjects* 函式會在傳入的任何一個 event 被設為 signaled 狀態或收到 *dwWakeMask* 參數所包含的訊息來到時返回，接著只要檢查傳回值，就可得知究竟發生何種事件。

從上表中可看到，*QS_TIMER* 旗標正好符合我們的要求。一旦有 *WM_TIMER* 視窗訊息進入訊息佇列時，*MsgWaitForMultipleObjects* 函式就會返回，傳回值為 *WAIT_OBJECT_0* 加上等待的核心物件數目。利用這項能力，可讓原本只能等待事件的迴圈變成訊息迴圈及事件等待迴圈的綜合體：同時處理視窗訊息的取得、分派，以及等待核心物件的觸發，進行適當的處理。

```
#0001 void __fastcall TWaitThread::Execute()
#0002 {
#0003     DWORD TimerID; // 計時器 ID
#0004     HANDLE Events[2]; // 要等待的 events 陣列
#0005     TMsg Msg; // 訊息結構
#0006
#0007     // 設設計時器
#0008     TimerID = SetTimer(0, 0, 1000, NULL);
#0009
#0010     Events[0] = hFileChangeEvent;
#0011     Events[1] = hOtherEvent;;
#0012
#0013     do {
#0014         // 等待任何一個 event 被設定為 signaled, 或 WM_TIMER 來到
#0015         switch (MsgWaitForMultipleObjects(2, Events, false, INFINITE,
#0016             QS_TIMER)) {
#0017
#0018             case WAIT_OBJECT_0: ... // 檔案變更
#0019                 break;
#0020
#0021             case WAIT_OBJECT_0 + 1: ... // 計時器觸發
#0022                 break;
#0023
```

```
#0024     case WAIT_OBJECT_0 + 2: // 收到 WM_TIMER 訊息
#0025         // 取出 WM_TIMER 訊息結構
#0026         GetMessage(&Msg, 0, WM_TIMER, WM_TIMER);
#0027         ... // 進行適當處理
#0028         break;
#0029     }
#0030 } while (!Terminated);
#0031
#0032 KillTimer(0, TimerID); // 摧毀計時器
#0033 }
```

0012 ~ 0027 列的迴圈包含 *MsgWaitForMultipleObjects* 函式呼叫，它就是我所謂的「訊息迴圈及事件等待迴圈的綜合體」，對於等待事件型的執行緒十分管用。

不過，若執行緒等待的是 **blocking** 函式（例如預設狀況下，讀取檔案內容的 *ReadFile* 函式或讀取封包資料的 *read* 函式），此方案就派不上用場，必須使用上述的其它方案才行。

第三篇

桌面秘笈



第五章

一頭栽入桌面的世界

雖然離家在外，沒有媽媽整天要我收拾書桌，
不過我還是會每天乖乖地收拾好桌面，
只不過這個桌面在電腦螢幕裡頭...



來，伸出你拿筷子的那隻手，抓住滑鼠，請你跟我這樣做：移到工作列上，點按滑鼠右鍵，在蹦現出來的功能表上選擇「所有視窗縮到最小」¹。接下來，你看到了什麼？除了桌面背景圖、捷徑圖示外，螢幕上還剩下什麼東東？就是它，它就是我們今天的主角——桌面視窗。對於我們天天在其上設計程式、撰寫文件、把玩軟體、拖拉捷徑的桌面，除了偶爾換換桌布、更改底色外，你瞭解多少？

剛接觸 Windows 時，最大的娛樂除了「遊樂場」裡頭的「踩地雷」、「接龍」外，就是用小畫家來塗塗鴉，然後打開「控制台」，以不同的樣式來設定桌面底圖。生來就沒什麼美術細胞的我，雖然只能畫些簡簡單單的圖形，如小狗、小貓、或池邊的呆頭鵝等等，不過看著自己的心血結晶煞有其事地排排列出現在螢光幕上，還是挺有成就感的。

如同每回寫作業前總會先將書桌收拾乾淨，才能有好心情專心作功課。桌面畢竟還是桌面，對咱們電腦族而言，螢幕上的桌面予人的視覺感受似乎也會影響工作情緒，至少對我而言是這樣的。上網閒逛前，我會將桌面底色換成淺粉藍，選張漂漂的美女圖設定為背景圖，字型改為如「Comic Sans MS」帶點詼諧的字體，同時打開 MP3 一邊哼唱一邊與網友打屁聊天看文章；撰寫程式、文章時，我會將桌面底色設為全黑、移除背景圖、字型換成正經八百的「Bookman Old Style」字體，專心工作。這兩套桌面給我截然不同的感受，前者像是讓人放鬆心情、可以盡情嘻鬧的吧台，後者卻像是要我心無旁騖、全神貫注於工作的辦公桌。

除了更換桌面底色、填圖樣式及背景圖案外，Windows 並沒有提供別的方法供我們佈置桌面。本文將由分析桌面視窗的構成切入，觀察桌面視窗的特性及行爲；接著介紹作業系統與使用者之間的中介服務—shell，依序探討使用者介面的各項元件特性及設定；最後長驅直入，直闖 shell 執行程序內部，一舉將背景視窗的掌控權拿下，自此之後即可隨心所欲揮灑自如，桌面外觀完全為我們所控，不再只是換換樣式改改桌布而已。

¹ 擁有【Microsoft Windows】熱鍵的使用者，可以按下【Win - M】來達成全部視窗最小化，十分方便。

孫子曰：「知彼知己，勝乃不殆；知天知地，勝乃可全」。在攻克要塞之前，必先掌握所有可用資訊才行，且讓我先派出偵防機探探「桌面」的底細。

桌面的構成

我們常說「桌面」、「桌面佈景」、「桌面圖示」的，而「桌面」究竟是什麼東東？由哪些成份組成的？它是誰提供的？它是一個視窗嗎？謎題即將解開...

唯一的桌面視窗

在 Windows 中，「視窗」是無所不在的。Windows 的視窗系統採階層架構，每個視窗皆有父視窗，可能擁有數個子視窗及數個兄弟（siblings）視窗。以資料結構的術語來說，整個視窗系統可看成是一株多元樹。

視窗們之間的樹狀／階層關係究竟如何？待我打開 Microsoft Spy++，仔細觀來。

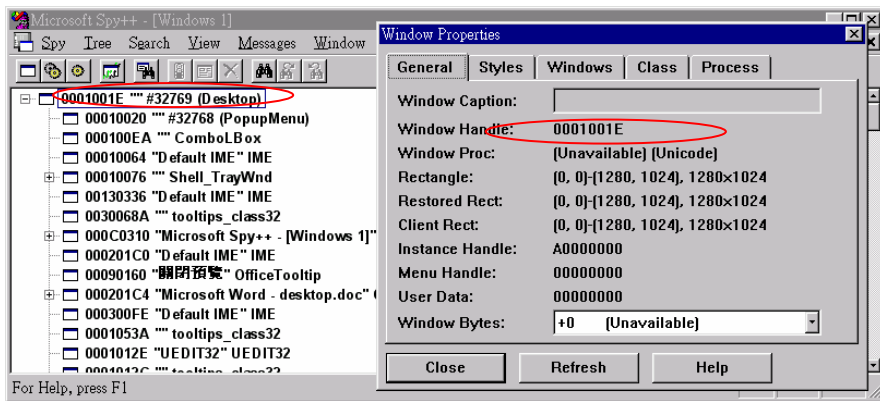


圖 5-1 / 以 Microsoft Spy++ 觀看目前視窗列表

從上圖可以很明顯地看出，handle 為 0001001E，類別名稱為「#32769」的那個視窗是整株「視窗樹」的根節點，它是所有視窗的始祖，也是整個系統中唯一沒有父視窗的視窗。

反過來說，系統中除此之外所有的視窗皆是其直接或間接後代，這個視窗有個特別的稱號，叫做「桌面視窗」（desktop window）。

Info

事實上，再仔細觀察看看，你可能會發現其實有許多最上層視窗的 parent window handle 為零，而不是我們預期中的桌面視窗 handle。對此，系統的處理方式是，若視窗的 parent window handle 為零，就代表其父視窗是桌面視窗。

在 Microsoft Spy++ 中，可以看到不少 parent window handle 為零的最上層視窗；但是在 Borland WinSight32 及 Numega SoftICE 等工具查詢這些視窗時，會主動將它們的 parent window handle 顯示為桌面視窗的視窗 handle。這是個很明顯的例子，表示系統在許多狀況下會將視窗 handle 0 視為桌面視窗。

桌面視窗，不消說，永遠只有一個，不多不少。它有幾點特性：

- 桌面視窗的視窗類別名稱爲「#32769」²，夠怪吧！
- 桌面視窗只有一個，呼叫 *GetDesktopWindow* API 函式可取得其視窗 handle。
- 桌面視窗的大小永遠與螢幕解析度相同，例如我使用的是 1280 x 1024 的畫面解析度，桌面視窗的範圍即爲 (0, 0) – (1280, 1024)³。
- 桌面視窗由系統產生⁴，無法摧毀。即使具有系統管理者權限，若嘗試著呼叫 *DestroyWindow* API 函式來消滅它時，也會得到 *ERROR_ACCESS_DENIED* 錯誤訊息。

² 一般情況下，井字號開頭的視窗類別名稱表示此視窗類別名稱是一個數值型別的全域 atom，經由 *MAKEINTATOM* 及 *GlobalAddAtom* 函式加入全域 atom 表格。

³ 我不但使用 1280 x 1024 解析度，而且還是小型字，一個畫面可以塞很多字哦！:P

⁴ 再靠近點看，桌面視窗是由 SYSTEM 模組的數十個執行緒其中之一所產生。

- 呼叫 `CreateWindow` 或 `CreateWindowEx` 等函式建立新視窗時，若傳入的視窗風格包含 `WS_OVERLAPPED` 或 `WS_POPUP` 旗標，新視窗會成爲桌面視窗的子視窗，也就是所謂的最上層視窗（top-level window）。這麼說來，似乎得將桌面視窗叫成唯一的「最最上層視窗」才對，因爲它不但是最上層視窗的父視窗，更具有唯一性，絕無僅有。
- 雖然 0 不是個合法的視窗 handle，但若呼叫與 device context 相關的 API 函式（例如 `GetDC`、`CreateCompatibleDC` 等等）時，可用 0 來取代桌面視窗的視窗 handle 值。打開 `winuser.h`，可以在裏頭找到下列的宣告：

```
/*
 * Special value for CreateWindow, et al.
 */
#define HWND_DESKTOP      ((HWND)0)
```

由此可見，除了前面所提的 parent window handle 情況，有不少地方也可以 0 來代表桌面視窗，這舉動不但是合法的，而且還是系統內定的標準行爲。

尋根工作雖然告一段落，但真相似乎尙未水落石出。我們已經找到所有視窗的祖先視窗——桌面視窗，但我們平日見到的桌布、圖示及下方的工作列等等，是由桌面視窗提供給我們的嗎？

它叫 Shell，不是貝殼

玩過 X Window 的朋友，是否曾驚豔其炫麗、多變的使用者介面呢？同樣一套 X Window 系統，只需執行不同的 window manager，便可擁有各式各樣功能、外觀各有巧妙的操作介面，甚至連視窗行爲特性、快速列、功能選單、程序列表皆可變化加強，其中尤以酷似 Windows 95 的 `fvwm95`、`kwm`、`xwm` 等等 window manager 最負盛名。之所以如此方便，乃因 X Window 採開放架構，人人皆可自行發展 window manager 嵌入系統中，供其它使用者安裝執行。而這兒所說的 window manager，其實就是圖形作業環境下 shell 的另一個稱號。

Talk

Windows 上如今也有不少可以加強使用者介面的 window manager 類似產品，其中的佼佼者為 LiteStep，下圖就是由 LiteStep 製作出的一個使用者介面，我個人還蠻喜歡的：蒼茫、深邃，以及魔法師所帶來的神秘氣氛。

產生興趣了嗎？請速至 <http://www.litestep.net/> 一遊。:-)



Shell，定義上而言，它擔任作業系統與終端使用者之間的介面，接收使用者的輸入，將請求轉告作業系統，並將執行結果回應予使用者。在傳統 DOS 及 UNIX 的純文字操作模式下，shell 程式只需擔負命令解譯的工作 (command interpreter)，處理少數內部指令 (如 DOS 的 DIR、TYPE、MKDIR 或 UNIX 上的 ls、ln 等指令) 及行程控制 (job control) 的工作即可。有些 shell 內建 script 語言 (如 DOS 的批次檔，以及大部分 UNIX shell 都提供的 shell script) 的支援，以便使用者進行大量或單調工作的自動化處理。

隨著視窗概念的出現及圖形化使用者介面的普及，shell不再只負責命令解譯，視窗介面作業系統的shell還必須負責視窗風格及外觀的繪製⁵、視窗之間的位置排列協調、快捷選單的提供、行程及視窗切換的操作方法等等許多涉及使用者介面的細節。

Shell 的貢獻

在 Windows 中，shell 與作業系統緊密地結合在一起，所以雖然你平日可能沒有察覺，但是 shell 程式的確存在。它不是別人，正是大家熟悉的 EXPLORER.EXE。

Info

此處我不用「檔案總管」而寫明執行檔名 EXPLORER.EXE 是有原因的。沒錯，「檔案總管」是由 EXPLORER.EXE 提供，但它並不只提供「檔案總管」，shell 也是它的其中一項功能，請別將兩者搞混了。

打開「工作管理員」，切換至「處理程序列表」，找到 EXPLORER.EXE 程序，請將滑鼠指標移到【結束執行程序】按鈕上方，閉上眼睛，狠下心來按下滑鼠鍵，再度張開眼睛時，你將發現...

1. 畫面下方⁶的工作列消失了，連帶著熟悉的【開始】按鈕、可層層爬越的【程式集】、上層視窗列表及小時鐘也無影無蹤。
2. 桌面圖示全部消失，只剩下桌布，桌布以外的區域完全填滿桌面系統顏色。此時在桌面上點按滑鼠右鍵，無所不在的快速功能選單不復出現。
3. 由於桌面捷徑的失蹤，原本設定的捷徑啟動熱鍵皆失去作用。

⁵ 視作業系統的開放性而定，如Windows下的視窗風格及外觀繪製就不是shell的工作，完全交由系統本身的USER模組負責。

⁶ 當然吶，也有可能是在畫面左方、右方或上方，甚至設定為自動隱藏，端視工作列原本的位置設定而不同。



圖 5-2 / 結束 EXPLORER.EXE 程序的前後畫面

Info

上述動作只可在 Windows NT/2000 下進行。我在 Windows 98 上試驗的結果，發現只要一幹掉 EXPLORER 行程，它就會立即重新啓動，所以根本沒有機會見到 shell 不存在的情況。

因此，後續的「尋回失落的 Shell」及「更換 Shell」兩個小節，也只適用於 Windows NT 系列平臺，使用 Windows 95/98 的讀者們請當作新聞看看就好。

幸好，每個視窗看起來還都是好端端的，拉大、拉小、移動、最大化、最小化各項操作十分正常，【ALT - TAB】熱鍵也仍可以使用，一點也不受 shell 消失的影響。毫無疑問地，可以得知 EXPLORER.EXE 只是個使用者等級的應用程式，擔負使用者介面的加強工作，存在與否不會影響應用程式的正常運作。

工作區域 (Workarea)

延續剛才的話題，將shell程序終結之後，可以觀察到一個有趣的現象－工作列突然消失，但是工作區域並沒有隨著更正為螢幕大小，此時試著將視窗最大化，螢幕底下會殘留一

條空白區域，挺呆的⁷。

追本溯源，原本 Windows 的視窗系統並沒有管理視窗對齊排列（alignment）的機制，爲了 Windows 95 新增的「工作列」，設計者只好加上「工作區域」的概念，並在視窗的操作及行爲上新增兩道特性：

- 視窗最大化時，位置及大小會正好涵蓋整個工作區域。
- 拖曳視窗時，滑鼠指標的「Hot Spot」⁸無法脫離工作區域範圍。

呼叫 *SystemParametersInfo* 函式，傳入 *SPI_GETWORKAREA* 及 *SPI_SETWORKAREA* 動作代碼，可分別取得及設定目前工作區域的範圍。

重量級 API – SystemParametersInfo

我不得不提醒各位，若對 Windows 的使用者介面程式設計有興趣，就必須常與 *SystemParametersInfo* 這支 API 函式打交道。*SystemParametersInfo* 是個多功能的函式，根據傳入的動作代碼，截取或設定各式各樣的系統參數，這些系統參數大部分與使用者介面及操控有關，種類繁多，請花點時間閱讀線上說明的 *SystemParametersInfo* 主題，可以在其中發現不少新奇好玩的系統設定。

⁷ 這裏所描述的狀況只有在工作列沒有設定爲「自動隱藏」的情況下才會發生。

⁸ Hot Spot指的是滑鼠指標圖形真正作用的「點」，通常會在圖形的左上角，但隨著滑鼠指標的形狀不同，有的在正上方，有的在右下角，哪邊都有可能。

BOOL SystemParametersInfo (

UINT uiAction,
UINT uiParam,
Pointer pvParam,
UINT fWinIni

);

參數

- uiAction* 動作代碼，可選擇欲截取或設定哪一個系統參數。
- uiParam* *UINT*參數，含意視動作而定。
- pvParam* *Pointer*參數，含意視動作而定。
- fWinIni* 如果動作為「設定」，由此參數指定「設定成功」後的接續動作：
 - SPIF_UPDATEINIFILE* 將新的設定值寫入系統登錄。
 - SPIF_SENDWININICHANGE* 廣播 *WM_SETTINGCHANGE* 視窗訊息至系統中所有最上層視窗。

回返值

如果成功，傳回 *true*；否則傳回 *false*。

表 5-1 / 與工作區域相關的 *SystemParametersInfo* 函式動作代碼

動作代碼	含意
<i>SPI_SETWORKAREA</i>	設定或取得工作區域， <i>pvParam</i> 指向欲設定為工作區域的
<i>SPI_GETWORKAREA</i>	<i>TRect</i> 結構。

例如，下列程式碼可以將目前工作區域設定成與 *Form1* 相同大小的區域：

```
#0001 void __fastcall TForm1::btnSetWorkareaClick(TObject *Sender)
#0002 {
#0003     TRect R;
#0004
#0005     R = BoundsRect;
#0006     SystemParametersInfo(SPI_SETWORKAREA, 0, &R, SPIF_UPDATEINIFILE);
#0007 }
```

設定新的工作區域後，雖然立即發生效用，但畫面上現存的視窗並不會自動更新它們的位置及大小。如果你希望做到如同改變工作列位置時，畫面所有視窗也隨之調整，就必須另行撰寫程式碼，呼叫 *EnumWindows* 函式一一查驗所有最上層視窗並進行適當調整。

尋回失落的 Shell

見不著螢幕下方熟悉的工作列，連單擊滑鼠右鍵也不見快捷功能表蹦出，雖然將它踢出家門才剛剛發生的事，此刻卻開始想念親切的 shell，於是一面哼著優客李林的「認錯」，一面找尋...

尋回 shell 的過程實在輕鬆愉快，只要執行 Windows 目錄下的 EXPLORER.EXE，shell 就會再度出現，畫面回復原狀，彷彿啥事都沒發生過似的...。

噢，如果終結 shell 之前沒有預先在畫面上保留命令列視窗或其它可以執行、載入新行程的程式，請按下【CTRL - ALT - DEL】鍵叫出【工作管理員】，利用它的【新程序...】功能來啟動 EXPLORER.EXE。

EXPLORER.EXE 有幾項好玩的特性：

1. 首度執行時會成為 shell；此後每次執行則會開啓新的「檔案總管」視窗。
2. 若有任何不當行為損毀shell行程，系統會重新啓動shell⁹。

更換 Shell

其實 shell 不一定非得是 EXPLORER.EXE 不可，我們也可以自行撰寫應用程式來取代它。在 Windows NT 中，shell 的執行檔名置於登錄資料庫的 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\Current Version\WinLogon\Shell 鍵值。我曾將它改為「踩地雷」遊戲的執行檔 WINMINE.EXE，重新開機之後，果然畫面上只有空空的桌面及「踩地雷」視窗。而習以為常的工作列及【開始】按鈕是

⁹ 此項功能由登錄資料庫的HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\AutoRestartShell鍵值決定。

EXPLORER.EXE 提供的，所以不會出現，無聊得很。所以，還是趕緊開啓登錄編輯器，把 shell 設定值改回來吧！

當然，「踩地雷」不可能比 EXPLORER.EXE 更適任 shell 的工作，在此講述的重點是，「Shell 是作業系統之上的一層，不是由作業系統提供，所以我們可以任意地移除、終止甚至抽換 shell」。

桌面上的特殊視窗

Shell 啓動後，會在桌面上建立兩個最上層視窗，它們的視窗類別名稱分別是 *Progman* 及 *Shell_TrayWnd*。這兩個視窗又各自擁有子視窗、孫視窗及曾孫視窗等等，階層排列如下¹⁰：

:hwnd -c Progman				
Handle	Class	WinProc	TID	Module
01006A	Progman	016C27FA	91	0080:0000
01006C	SHELLDLL_DefView	77C21127	91	00000000
01006E	SysListView32	00249BB5	91	00000000
010072	SysHeader32	0024F16C	91	00000000

:hwnd -c shell_traywnd				
Handle	Class	WinProc	TID	Module
010076	Shell_TrayWnd	016C1222	91	0188:0000
010078	Button	016C1DE7	91	00000000
01007A	TrayNotifyWnd	016C1F82	91	00020000
01007C	TrayClockWClass	016C1000	91	00000000
010082	MSTaskSwWClass	016C22B6	91	00800000
010084	SysTabControl32	00242900	91	00000000

Progman 類別名稱是由 Windows 3.1 時代延用下來的，它的視窗大小與畫面相同；*SHELLDLL_DefView* 及 *SysListView32* 視窗佔有工作區域，即工作列外的全部範圍；

¹⁰ 我在 Windows NT 4.0 上取得這些資料，微軟十分可能會在將來的版本更新 shell 的視窗架構。

Shell_TrayWnd 視窗即工作列，與 *Progman* 視窗一樣，也是最上層視窗；*Shell_TrayWnd* 視窗有一個 *Button* 子視窗，就是大家熟悉的「開始」按鈕。

顯示桌面圖示的 *SysListView32* 視窗通常是我們最感興趣的對象，也就是我們俗稱的「桌面」，因此在此為它們正名：

- 桌面視窗（desktop window）指的是所有視窗之始，最上層視窗的父代視窗，由系統產生。
- 背景視窗指的是顯示桌面圖示的視窗，視窗類別是 *SysListView32*，由 shell 程序產生。

由於背景視窗是：

「「「類別為 *Progman* 的最上層視窗」的第一個子視窗」的第一個子視窗」

可以利用下列呼叫取得它的視窗 handle：

```
HWND GetDesktopListView()  
{  
    return GetWindow(GetWindow(FindWindow("Progman", NULL),  
        GW_CHILD), GW_CHILD);  
}
```

十分有趣的是，背景桌布及填圖樣式是由桌面視窗顯示，隸屬 **SYSTEM** 模組；而桌面圖示、工作列等其它功能由 shell 程序提供，看來是由 Windows 3.1 承襲而來的習慣。

桌面上的把戲

桌面十足是個可供創意、想像力及技術能力盡情揮灑的場地，請等著瞧，看筆者我能玩出什麼把戲。

席捲桌面，氣吞四海

絕大部分的應用程式通常都乖乖地待在自己的視窗內繪圖，沒事絕不會跨出領土半步—

你啥時見過「小畫家」可以把線條拉出視窗外啦？但總有些激進份子就是不甘寂寞，除了自己的視窗外，還愛闖到別人視窗裏摧堅陷陣。其實，只要你喜歡，可以在桌面上任何地方清除及繪製圖形，享受「舉目四望，皆我畫布」的豪情快意！

從哪邊下筆呢？其實都行，只要能夠取得目的視窗的視窗 `handle`，就有辦法在其上下筆。通常我們會選擇桌面視窗或背景視窗，唯一的差別是：桌面視窗包含整個畫面，而背景視窗只包含工作區域。

相信你已十分熟悉 Windows 的 GDI 繪圖函式庫，利用 GDI 進行任何繪圖動作之前，請先取得 `device context handle`。有一點必須注意的是，由於未知原因，呼叫 `GetDC` 函式時請以 0 代替桌面視窗的視窗 `handle`，否則雖然可以正確呼叫 GDI 函式，卻無法成功繪製上去。繪製工作大致如下：

```
#0001 HDC DC = GetDC(0); // 取得桌面視窗的 device context
#0002 try {
#0003     ... // 利用取得的 DC 進行任何繪製動作，隨心所欲
#0004 } __finally {
#0005     ReleaseDC(0, DC); // 釋放 device context
#0006 }
```

Info

0001 列經由視窗 `handle 0` 取得 DC 後，再以 `WindowFromDC` API 函式來驗證，傳入剛取得的 DC `handle`，可以得到桌面視窗的視窗 `handle`，這表示以視窗 `handle 0` 來取得桌面視窗的 DC 是沒錯的。但若經由桌面視窗的視窗 `handle` 取得 DC，就無法在其上繪圖，真是奇怪的设计。

如果你偏好使用 VCL 的 `TCanvas` 類別，也可將由 `GetDC` API 函式取得的 `device context handle` 指派給 `TCanvas::Handle` 屬性，就可利用此 `TCanvas` 物件進行繪製：


```

#0001 TCanvas* Canvas = new TCanvas;
#0002 try {
#0003     // 取得 device context, 並指派給 TCanvas::Handle 屬性
#0004     Handle = GetDC(0);
#0005     ... // 使用任何 TCanvas 提供的屬性及函式進行繪製動作
#0006 } __finally {
#0007     ReleaseDC(0, Handle); // 歸還 device context
#0008     delete Canvas; // 釋放 TCanvas 物件
#0009 }

```

TCanvas 是個十分特別的類別，它將大部分的 GDI 函式封裝進來，卻獨留 *CreateHandle* 這個虛擬函式留待後代繼承，標準的「萬事俱備，獨缺東風」型態類別。VCL 設計者提供如此簡易的繼承方式供我們衍生新類別，不試試看實在可惜。於是，由 *TCanvas* 繼承，我建立一個新的 *TScreenCanvas* 類別，改寫 *CreateHandle* 及解構函式，並新增 *FreeContext* 函式，負責歸還 DC handle。

SCREENCANVAS.H

```

#0001 class TScreenCanvas : public TCanvas {
#0002 private:
#0003     void __fastcall FreeContext();
#0004 protected:
#0005     virtual void __fastcall CreateHandle(void);
#0006 public:
#0007     __fastcall virtual ~TScreenCanvas();
#0008 };

```

SCREENCANVAS.CPP

```

#0001 __fastcall TScreenCanvas::~TScreenCanvas()
#0002 {
#0003     FreeContext();
#0004 }
#0005
#0006 void __fastcall TScreenCanvas::FreeContext()
#0007 {
#0008     ReleaseDC(0, Handle);
#0009     Handle = 0;
#0010 }
#0011
#0012 void __fastcall TScreenCanvas::CreateHandle(void)
#0013 {
#0014     Handle = GetDC(0);
#0015 }

```

桌面塗鴉程式

有這麼一套叫做 Desktop Toys 的小程式，它提供八種有趣的工具，包括電鋸、機關槍、火焰槍、超人、橡皮擦、強力彈簧等等，讓你可在桌面上盡情地轟炸、射擊、放火、塗色，隨心所欲地蹂躪桌面上所有視窗及圖形。不論你當時正在撰寫程式或是打報告、看 VCD，它才不管三七二十一，直接包下整張桌面供你胡搞，直到桌面亂得不能再亂，破的破、碎的碎、焦的焦，心滿意足地按下【ESC】鍵時，程式才將原來的畫面還給你，好像啥事都沒發生過似的。瞧，這是我的桌面被小小地蹂躪過的下場：

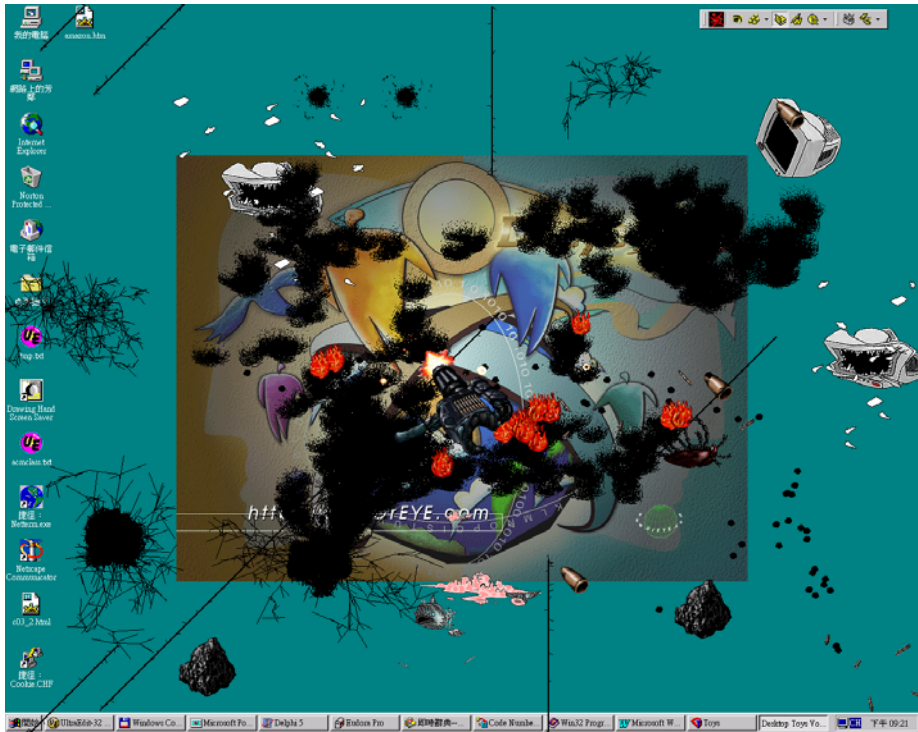


圖 5-3 / Desktop Toys 程式的執行下場，你看，桌面是不是一團糟呢？

現在，我們擁有可在桌面上繪製圖形的 *TScreenCanvas* 類別後，撰寫一個類似的程式也不是問題。雖然乍看之下，這些破壞好像都直接發生在桌面視窗上，其實真正的作法十

分簡單，這類型的程式，通常只是在程式啟動時，將本身的視窗設為工作區域大小，並把目前的畫面影像拷貝到自己的視窗上來，就在那麼一眨眼間，偷天換日地以自己的視窗取代原有的桌面。

如此一來，所有的繪製及遊戲動作都發生在自己的視窗範圍內，桌面蹂躪程式也變得不特別了，跟一般的 GDI 程式沒有什麼兩樣。好，弄清這類型程式的底細後，藉著 *TScreenCanvas* 類別的輔助，我們也來寫一個，寫一套可使用各色畫筆在桌面上塗鴉的桌面程式。

下圖是桌面塗鴉程式的設計時期畫面：右方是 *TColorGrid* 元件，供使用者選取畫筆顏色；左方塗鴉區由 *TImage* 元件提供，具有保留影像的能力。



圖 5-4 / 桌面塗鴉程式的設計時期畫面，右方是 *TColorGrid* 元件

程式撰寫的部分並沒有什麼特別，大致說來包含下列工作：

- 將 *WindowState* 屬性設為 *wsMaximized*，使視窗呈最大化，佔據整個工作區域。
- *Form* 的 *OnActivate* 事件觸發時，此時主視窗尚未出現，趁此機會將目前的畫面儲存下來，同時複製到 *TImage* 元件 *imgMain* 上頭。
- 滑鼠指標的處理比較麻煩：收到 *WM_SETFOCUS* 訊息時，呼叫 *ShowCursor* API 函式將系統提供的滑鼠指標隱藏；收到 *WM_KILLFOCUS* 訊息時，同樣呼叫

ShowCursor 函式顯示滑鼠指標。

- 利用 *imgMain* 的 *OnMouseMove* 事件追蹤滑鼠座標，將榔頭圖形正確地畫在滑鼠位置上。由於每當滑鼠位置一變，先前畫的榔頭圖形就會破壞桌面，所以必須另外使用一個 *TBitmap* 物件來儲存滑鼠位置的榔頭大小圖形，當榔頭移開後，立即將原有圖形恢復。
- *imgMain* 的 *OnMouseDown* 及 *OnMouseUp* 事件觸發時，分別進入「塗鴉」及「移動」狀態。若滑鼠移動且正處於塗鴉狀態，就呼叫 *TCanvas::Ellipse* 函式畫出一個實心圓形，顏色依當時在 *grdColor* 元件的選擇而定。
- 當使用者按下【ALT - F4】來結束程式時，除了進行正常的資源釋放動作外，不必進行額外的處理。因為我們塗鴉的範圍只在自家視窗內，完全沒有影響到其它視窗。

下圖是程式執行後，我拿著榔頭在畫面上隨手亂塗的結果。許久沒有塗鴉的我發現到，握著滑鼠寫字、亂畫圖案也挺好玩的耶！

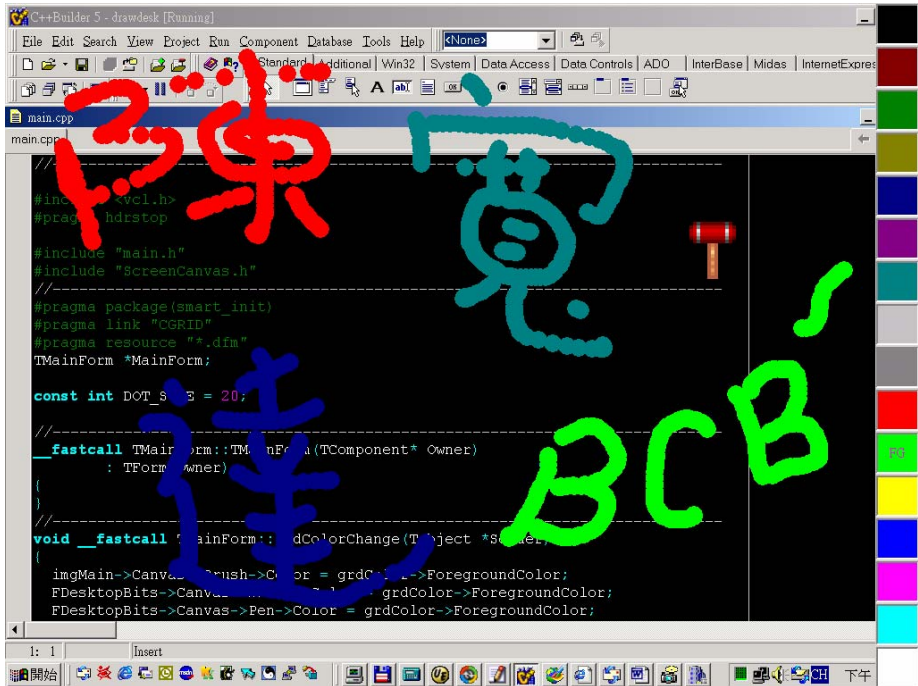


圖 5-5 / DrawDesk 程式畫面，拿著榔頭在桌面上塗鴉

畫面截取

俗諺說：「施比受更有福」。在桌面上作畫已不是問題，截取桌面圖形自然更不成問題囉！唯一不同的是，剛剛是在桌面上頭作畫，現在改為將桌面上的影像搬進自己的視窗裡。

我希望撰寫一個靈活的視窗影像捕捉工具，滑鼠指標移到哪，就立即將滑鼠指標下方的視窗影像截取下來，並且可以將截取結果儲存起來。程式的主畫面設計如下：



圖 5-6 / 「視窗捕捉程式」設計時期畫面

視窗上的綠色圓形是一個 *TShape* 元件，中央放置 *TScrollBar* 元件，裏頭再擺上存放影像的 *TImage* 元件。*dlgSavePicture* 元件的類別是 *TSavePictureDialog*，用來將截取的影像儲存到檔案。

tmrDelay 是 *TTimer* 元件，用來擔任影像延遲捕捉的計時工作。我發現若是每當滑鼠指標移到不同視窗，就立即捕捉該視窗影像的話，很容易將滑鼠移動時「只是經過」的視窗

影像也抓下來，這使得滑鼠的移動變得一頓一頓的，而且也白白浪費影像抓取所耗費的資源及時間。所以特別加上此計時器元件，若滑鼠指標移動到一個視窗上且在 500 毫秒內沒有離開此視窗的話，程式才會進行影像截取動作。

程式中唯一需要克服的難處是，必須由滑鼠指標的座標，取得該座標下方的最上層視窗。雖然系統提供了 *ChildWindowFromPoint* API 函式可供使用，而且根據函式名稱，看起來完全符合我們的需求。但是它在尋找視窗時會連同非可見視窗一併尋找，所以由它取得的視窗可能是在畫面上根本看不見的視窗，而看不見的視窗對我們畫面截取程式毫無意義，只好自己動手寫一個囉。

以下是 *FindTopMostWindow* 函式，傳入父視窗及相對座標，就傳回該座標上的最上層子視窗。此函式正是遞迴呼叫的絕佳應用：

```
#0001 HWND FindTopmostWindow(HWND ParentWnd, TPoint PT)
#0002 {
#0003     TRect R;
#0004
#0005     HWND Wnd = GetWindow(ParentWnd, GW_CHILD); // 取得第一個子視窗
#0006
#0007     while (Wnd != 0) {
#0008         GetWindowRect(Wnd, &R); // 取得視窗矩形區域
#0009
#0010         if (IsWindowVisible(Wnd) && PtInRect(&R, PT)) {
#0011             // 忽略程式的 Main Form
#0012             if (Wnd == Application->MainForm->Handle) return 0;
#0013
#0014             // 找到視窗，遞迴呼叫取得子視窗
#0015             return FindTopmostWindow(Wnd, PT);
#0016         }
#0017
#0018         Wnd = GetWindow(Wnd, GW_HWNDNEXT); // 取得下一個 siblings 視窗
#0019     }
#0020
#0021     return ParentWnd; // 找不到，它就是我們要找的
#0022 }
```

0012 列特別檢查取得的視窗 handle 是否等於 *Application->MainForm->Handle*，即程式的 main form，如果是的話，就跳過不處理。若不這樣做，「視窗捕捉程式」捕捉到自己的影像，豈不等於拿塊磚頭砸自己腳嘛。*GetWindow* 是尋訪視窗時十分好用的函式，可

由此函式依賴 *GetWindow* 函式的程度得知，請務必學會它的使用。

另外還有一點也十分有趣，當進行影像截取動作時，若發現被截取的視窗範圍與截取程式本身視窗有重疊時，必須先將本身隱藏起來或是最小化，複製影像時才不會連本身一塊拍進去。

下圖是「視窗捕捉程式」的執行畫面。程式以 C++Builder 撰寫而成，但是它執行後，第一個捕捉的就是 C++Builder 的主視窗畫面，這算不算是過河拆橋啊？:P



圖 5-7 / 「視窗捕捉程式」執行畫面，我用它抓下 C++Builder 的主視窗

桌面隨意貼

大約是兩三年前的事了，那時我周圍的電腦，幾乎人人都裝上一套 3M 的 Post-It Software Notes 軟體，中文譯名為「桌面隨意貼」，它可讓你在桌面各處放置一張張的便條紙，嫌佔空間時又可縮為小方塊，以滑鼠左鍵雙擊的方式切換，十分方便。此程式的執行畫面請見圖 5-8。

這些便條紙不但可以隨心所欲地「釘」在桌面，能夠不同的顏色、字形來呈現，還可以搬來移去，將多餘或過期的便條紙放到「字紙簍」存放。不過呢，我想無論電腦螢幕中的便條紙再怎麼方便，還是不如手邊隨時可用的原子筆及筆記本，或是規劃完整的群組軟體，這類型的隨意貼軟體似乎大辣辣地風光一陣後就逐漸式微了。

即使如此，桌面隨意貼的點子依然令人激賞，撰寫這類型的程式似乎也挺有趣的，今天就讓我們來挑戰它吧！



圖 5-8 / 3M Post-It Software Notes 執行畫面

萬事起頭難，哦，不管不管，我決定先設計兩個 form，一個是管理便條紙的主視窗 *TMainForm*，另一個是擔任便條紙視窗的 *TNoteForm*。

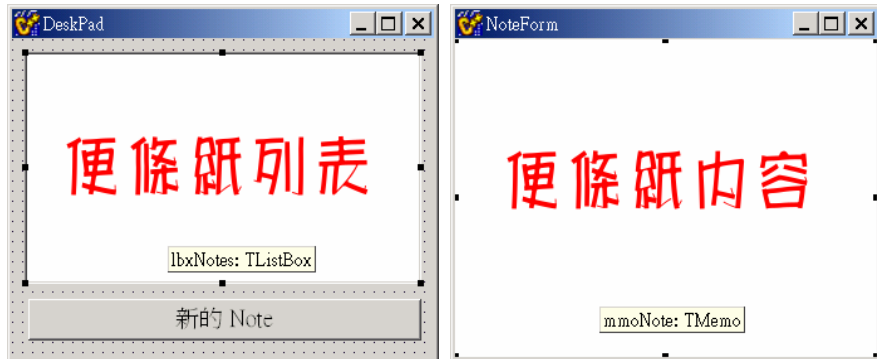


圖 5-9 / DeskPad 設計時期畫面，分別是主視窗及便條紙視窗

主視窗的任務

主視窗的重要任務為：

- 使用者按下【新的 Note】按鈕時，建立新的 *TNoteForm* 便條紙視窗。
- 程式結束時，將所有的便條紙視窗及內文儲存起來，下次啓動時再全部載入回來；換言之，使便條紙視窗具有永續性。
- 以滑鼠左鍵雙擊列示盒時，將對應的便條紙視窗帶到最上層，成為前景視窗。

爲了方便起見，我直接呼叫 *TStream::WriteComponent* 函式及 *TStream::ReadComponent* 函式來寫入及讀取便條紙視窗的內容。

在 *TMainForm::SaveNotes* 函式中，先建立 *TFileStream* 物件開啓檔案資料流，尋訪 *TScreen::Forms* 陣列，呼叫 *WriteComponent* 函式將主視窗之外的所有視窗（我們的程式中，除了主視窗外必定是便條紙視窗 *TNoteForm*）寫入資料流。

而在 *TMainForm::LoadNotes* 函式中，首先建立新的 *TNoteForm* 物件，指定 *TNoteForm* 的父視窗爲桌面視窗，再呼叫 *TStream::ReadComponent* 函式從檔案資料流將此便條紙視窗的內容讀出。

建立新的 *TNoteForm* 時，為什麼使用需要兩個參數的建構函式，而不使用常見的，只需要一個 *Owner* 參數的建構函式呢？帶有兩個參數（第二個參數沒有任何涵義，純粹用來區別不同版本的建構函式）的建構函式相當於 VCL 的 *TObject::CreateNew* 建構函式，而平日所用的是 *TObject::Create* 函式。

若按照正常程序來建立 *TNoteForm*，VCL 會自動為我們從執行檔的 *RCDATA* 資源區段將 form 及元件讀入，而接著我們又呼叫 *TStream::ReadComponent* 自行讀入 form 及元件，這不但多此一舉，還會造成元件重覆讀取及元件同名的例外發生。所以在此使用 *CreateNew* 建構函式，此建構函式與 *Create* 建構函式的唯一差別就是：form 建立後，不會從資源檔讀取屬性及元件，正好符合我們的需求。

```
#0001 const AnsiString FN_NOTES = "notes.dat";
#0002
#0003 void __fastcall TMainForm::LoadNotes()
#0004 {
#0005     TNoteForm* NewNote;
#0006     TFileStream* fs = new TFileStream(FN_NOTES, fmOpenRead);
#0007
#0008     try {
#0009         while (fs->Position < fs->Size) {
#0010             // 不要 streaming, 建立新的便條紙視窗
#0011             NewNote = new TNoteForm((TComponent*)NULL,
#0012                 0 /* dummy, invoking CreateNew */);
#0013             NewNote->ParentWindow = HWND_DESKTOP;
#0014
#0015             fs->ReadComponent(NewNote); // 讀取便條紙視窗內容
#0016         }
#0017     } __finally {
#0018         delete fs;
#0019     }
#0020
#0021     ListNotes(); // 重新列出目前便條紙視窗
#0022 }
#0023
#0024 void __fastcall TMainForm::SaveNotes()
#0025 {
#0026     TFileStream* fs = new TFileStream(FN_NOTES,
#0027         fmCreate | fmOpenWrite);
#0028     try {
#0029         for (int i = 0; i < Screen->FormCount; i++)
#0030             // 對於程式中的每一個 form
```

```

#0031     if (Screen->Forms[i] != this) // 不寫入主視窗本身
#0032         // 將便條紙視窗內容寫入
#0033         fs->WriteComponent(Screen->Forms[i]);
#0034     } __finally {
#0035         delete fs;
#0036     }
#0037 }

```

0013 列設定 *TNoteForm* 的 *ParentWindow* 屬性，指向 *HWND_DESKTOP*（桌面視窗）。*Parent* 及 *ParentWindow* 屬性分別由 *TControl* 及 *TWinControl* 類別宣告，這兩個屬性互斥，同時只有其中一個有效：若父視窗為 VCL 元件，請使用 *Parent* 屬性，型別為 *TWinControl*；否則的話，使用 *ParentWindow* 屬性，型別為 *HWND*。

```

TControl = class
    ...
    property Parent: TWinControl read FParent write SetParent;
end;

TWinControl = class
    ...
    property ParentWindow: HWND read FParentWindow write SetParentWindow;
end;

```

當使用者按下【新的 Note】按鈕，建立新的 *TNoteForm* 便條紙視窗時，因為新視窗的父視窗不為 VCL 元件，所以使用另一個接收 *HWND* 參數的建構函式，它會以此參數作為此元件的 *ParentWindow*：

```

#0001 TNoteForm* __fastcall TMainForm::CreateNewNote()
#0002 {
#0003     // 沒有 VCL 父元件，以桌面視窗為父視窗
#0004     TNoteForm* frm = new TNoteForm(HWND_DESKTOP);
#0005
#0006     // 設定新便條紙視窗的位置
#0007     frm->Left = Left;
#0008     frm->Top = Top - frm->Height;
#0009     if (frm->Top < 0)
#0010         frm->Top = Top + Height;
#0011
#0012     UpdateWindow(frm->Handle); // 立即重繪視窗
#0013 }

```

便條紙視窗的設計

做為便條紙視窗，除了 form 上放置一個 *TMemo* 元件來撰寫顯示便條內文外，*TNoteForm* 還必須達成下列效果：

更改視窗風格

視窗風格與普通視窗不同，它必須位於所有視窗最下層、生滅時不必通知父視窗（桌面視窗）、使用小標題列（工具列視窗的特色）、可以改變大小等等。因此我改寫其 *CreateParams* 及 *CreateWnd* 兩個虛擬函式來修正它的視窗風格及視窗 Z order 位置：

```
#0001 void __fastcall TNoteForm::CreateParams(TCreateParams &Params)
#0002 {
#0003     TForm::CreateParams(Params);
#0004
#0005     // 設定正確的視窗風格及延伸視窗風格
#0006     Params.Style = WS_SYSMENU | WS_SIZEBOX | WS_VISIBLE |
#0007                 WS_CLIPSIBLINGS;
#0008     Params.ExStyle = WS_EX_TOOLWINDOW | WS_EX_WINDOWEDGE |
#0009                 WS_EX_NOPARENTNOTIFY;
#0010
#0011     // 直接置於桌面視窗上
#0012     Params.WndParent = HWND_DESKTOP;
#0013 }
#0014
#0015 void __fastcall TNoteForm::CreateWnd(void)
#0016 {
#0017     TForm::CreateWnd();
#0018
#0019     SetWindowPos(Handle, HWND_BOTTOM, 0, 0, 0, 0,
#0020                 SWP_NOACTIVATE | SWP_NOSIZE | SWP_NOMOVE | SWP_SHOWWINDOW);
#0021
#0022     // 加入"更改標題"選單項目
#0023     AddMyMenuItem();
#0024 }
```

修改系統功能表

在視窗標題列按下滑鼠右鍵時，會蹦現一個快捷功能表，我希望在此加上「更改主題」選項，讓使用者可以隨時更動便條紙主題。由於此快捷功能表由系統提供，非 VCL 管理

的範疇，因此必須呼叫 API 函式來加入分隔線及新的選項。主要藉由 *GetSystemMenu* 及 *InsertMenuItem* 兩道函式來達成：

```
#0001 const int SC_CHANGE_CAPTION = 0x10;
#0002
#0003 void __fastcall TNoteForm::AddMyMenuItem()
#0004 {
#0005     HMENU SysMenu;
#0006     TMenuItemInfo MenuItemInfo;
#0007
#0008     SysMenu = GetSystemMenu(Handle, false); // 取得系統選單 handle
#0009
#0010     MenuItemInfo.cbSize = sizeof(MenuItemInfo);
#0011     MenuItemInfo.fMask = MIIM_TYPE;
#0012     MenuItemInfo.fType = MFT_SEPARATOR; // 分隔線
#0013     // 加入橫線
#0014     InsertMenuItem(SysMenu, (DWORD)-1, true, &MenuItemInfo);
#0015
#0016     MenuItemInfo.fMask = MIIM_ID | MIIM_TYPE;
#0017     MenuItemInfo.fType = MFT_STRING; // 字串
#0018     MenuItemInfo.wID = SC_CHANGE_CAPTION;
#0019     MenuItemInfo.dwTypeData = "更改主題\tAlt+C";
#0020     // 加入選項
#0021     InsertMenuItem(SysMenu, (DWORD)-1, true, &MenuItemInfo);
#0022 }
```

處理新的功能表選項

將「更改主題」選項加入系統功能表後，當使用者選取此選項時，視窗就會收到 *WM_SYSCOMMAND* 視窗訊息，我們必須攔截此訊息：若其訊息結構的 *CmdType* 欄位和 *0xFFFF0* 數值進行 AND 運算¹¹ 後的結果為自訂的 *SC_CHANGE_CAPTION* 識別碼，就進行詢問及更動主題的動作，同時通知主視窗重新顯示便條紙列表：

```
#0001 void __fastcall TNoteForm::WMSysCommand(TWMSysCommand& Message)
#0002 {
#0003     // 使用者選取更改主題選項
#0004     if ((Message.CmdType & 0xFFFF0) == SC_CHANGE_CAPTION) {
#0005         // 詢問並更新主題
#0006         Caption = InputBox(Application->Title, "更改主題", Caption);
#0007         // 通知主視窗重新顯示列表
#0008         PostMessage(Application->MainForm->Handle,
```

¹¹ 這是因為 *CmdType* 欄位的最低四位元可能為系統所用，所以要將它篩除。

```
#0009     WM_NOTE_CHANGED, 0, 0);
#0010     } else
#0011         TForm::Dispatch(&Message);
#0012     }
```

視窗標題列雙擊處理

最後，希望當使用者按下【ALT - C】熱鍵或以滑鼠左鍵雙擊視窗標題列時，也進行更改主題的功能。所以必須撰寫 *TNoteForm::OnKeyDown* 事件的事件處理函式並攔截 *WM_NCLBUTTONDBLCLK* 視窗訊息來處理滑鼠左鍵雙擊事件：

```
#0001 void __fastcall TNoteForm::FormKeyDown(TObject *Sender, WORD &Key,
#0002     TShiftState Shift)
#0003 {
#0004     // 若按下 ALT-C ...
#0005     if (Key == 'C' && Shift.Contains(ssAlt))
#0006         Perform(WM_SYSCOMMAND, SC_CHANGE_CAPTION, 0);
#0007 }
#0008
#0009 void __fastcall TNoteForm::WMNCLButtonDbLClk(
#0010     TWMNCLButtonDbLClk& Message)
#0011 {
#0012     // 若雙擊標題列，視同要求更改主題
#0013     if (Message.HitTest == HTCAPTION)
#0014         Perform(WM_SYSCOMMAND, SC_CHANGE_CAPTION, 0);
#0015     else
#0016         TForm::Dispatch(&Message);
#0017 }
```

底下即是 DeskPad 程式的執行畫面，便條紙內容與圖 5-8 3M Post-It 程式執行時完全相同。3M Post-It 的視窗標題列及外框皆為自行繪製，所以光是視窗外觀一點，相比之下就差遠了。不過這畢竟只是原始碼加起來三百行不到的範例程式，潛力十足呢。

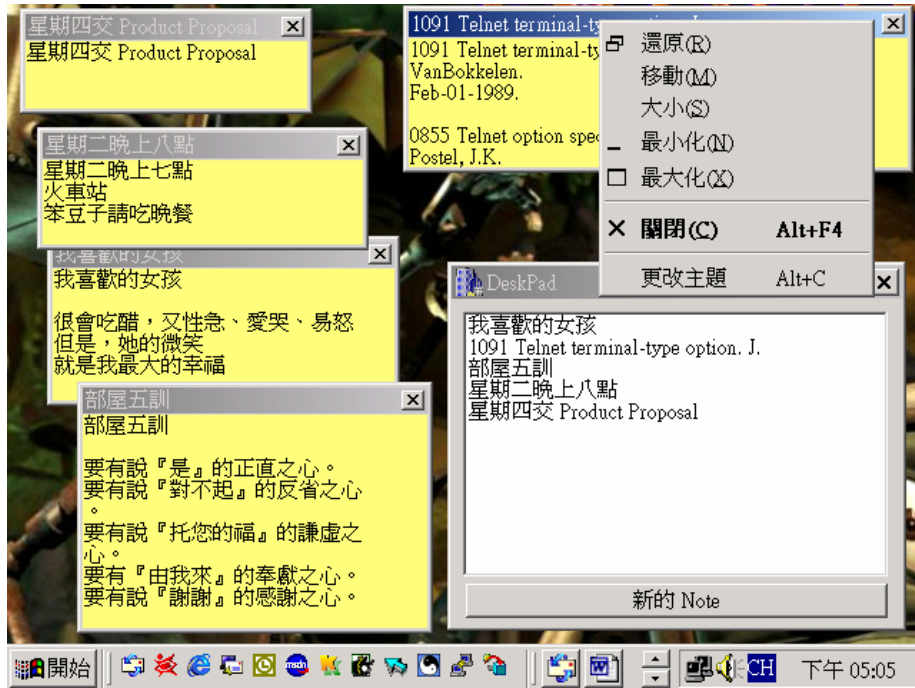


圖 5-10 / DeskPad 執行畫面，在便條紙視窗按下滑鼠右鍵可更改主題

Desktop Illusionist

坦白說，我本身就是個十足的桌布一族，平日到處搜集漂亮的風景畫、可愛的卡漫圖、養眼的美女圖，利用桌布自動更換軟體每十分鐘替換一次桌面。雖然可以時時欣賞不同的漂亮桌布，心悅神馳，不過每回當鍵盤的回應變慢，耳邊又傳來硬碟無來由的吱吱嘎嘎響時，就知道自動更換軟體又在換桌布了，簡直可當成定時提醒我別沈迷網路太久的小鬧鐘。

不過呢，一張桌面大小、全彩的BMP圖檔，檔案大小至少要以MB計¹²，雖然平日皆以

¹² 以我慣用的 1280 x 1024 解析度為例，桌面大小的全彩BMP圖形檔案大小為 3.75MB。

JPEG、PNG或GIF格式儲存在硬碟中，經過壓縮後明顯節省不少磁碟空間¹³，但由於Windows的桌布支援限制，任何影像檔要成為桌布前必須先轉為BMP格式。一來格式轉換、資料解壓縮需要大量的CPU時間，我的CPU平日就嫌慢了，轉換過程幾乎會把所有CPU工作能力完全霸佔，很是煩人；二來才剛轉換好、寫入磁碟的BMP檔案，呼叫 *SystemParametersInfo* API函式設定桌布時，還會再一次地將影像資料由檔案讀出，載入至桌面視窗行程，桌面視窗重繪後，再廣播視窗訊息請桌面上全部視窗進行重繪動作... 噢，我受不了這種沈重的負荷，只不過想換張桌布耶！

有人說：「永不知足及好奇心是人類進步的最大原動力」。嗯，不得不同意，爲了上述的種種理由，也爲了讓我的桌面更炫、更自由，酷得發亮的 Desktop Illusionist 誕生了...

源起

最早，撰寫 XDesktop 桌布管理／更換工具時，萌發讓更換桌布變成動畫的點子。目前更換桌布時總是舊的桌布「啪」的一聲突然就變成新的桌布，太冷血了，太沒感情了。如果能讓新設定的桌布從畫面邊緣慢慢地「滑」進桌面，或是如百葉窗的方式展開，漸進式地取代原有的桌布，我想更換桌布的過程會變得更有趣。

後來，曾在朋友的電腦上看到他用影像處理軟體繪製的月曆及通訊錄，儲存成 BMP 影像檔後，再設定爲桌布。桌面上不再放置無意義的風景或美女圖，而是最切身的資訊，這些資訊就在桌面上，只要沒有其它視窗擋住，就能一眼看盡。唯一的缺點就是必須將所有資料用繪圖軟體製作成 BMP 檔，十分麻煩；如果能夠很方便地更新資料，或放上計劃表，或者放上座右銘及喜愛的箴言，又不會像桌面隨意貼那類型的軟體一樣，在畫面上擺置一堆零亂的視窗，真是最理想的桌面應用。這個想法一直擺在我的腦中，等待著機會來實現。

又有一回，在另一位朋友的電腦上發現 Aptiva Desktop Customization 這套軟體，這套程

¹³ 以最常見的JPEG壓縮格式爲例，平均可爲影像檔縮減二十～三十倍的儲存空間。

式可真有奇門異術，它可讓桌面圖示「動」起來。雙擊任一桌面圖示後，圖示會快樂地轉幾圈再啟動程式，連【開始】功能表左方的 Microsoft Windows 圖案都成了不停捲動的走馬燈。

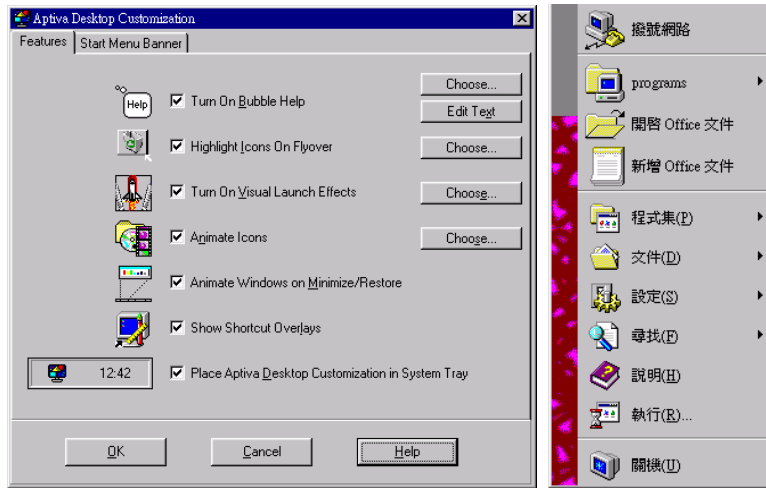


圖 5-11 / Aptiva Desktop Customization Desktop 設定視窗，可由此得知它的主要功能；右方是變成走馬燈的【開始】功能表

這麼奇特的軟體當然值得仔細研究一番囉！研究的結果是，它依賴 hook 機制來 subclass shell 行程的視窗，才能夠如此程度地控制桌面圖示的一舉一動。瞭解內部的實作方法後，這類型的搞怪程式也就見怪不怪了。

就這樣，如同我的其它許多作品，皆是醞釀了好久好久，才會在某一次的機緣下擦槍走火...噢，說錯了，是開花結果。一方面作為本章的範例程式，示範如何終極地將桌面完全佔領，自此以後對於桌面再無懵懂；另一方面想驗證長期以來盤踞在腦海中的實作規劃，程式嘛，不真正寫出來總怕流於空想，無法正確掌握程式的困難度及重點。

程式目的

我為 Desktop Illusionist 制訂如下的要求：

- 在桌布之上、桌面圖示之下使用可自訂的字形顏色顯示短文，欲顯示的短文必須能夠隨時修改，並且立即更新顯示。
- 在桌面上以秒為單位顯示目前時間，而且不會閃爍。
- 平時隱藏設定視窗，可藉由放置在桌面上的按鈕叫出設定視窗。此設定按鈕也必須在桌布之上、桌面圖示之下。
- 能夠隨時切換是否顯示桌布及填圖樣式。
- 能夠隨時執行、隨時結束，不影響其它應用程式。
- 能夠適應任何解析度及工作區域設定，程式執行中也能自動隨解析度或工作區域的變更而調整。
- 適用於所有 Win32 平臺。

程式手法

要在「桌布之上、桌面圖示之下」進行繪製，這些繪製工作很顯然地必須由背景視窗或桌面視窗¹⁴的身份來進行。正常情況下，畫面的繪製動作大致可分為下列步驟：

1. 桌面視窗將整個桌面填上桌面底色，若填圖樣式存在，則以填圖樣式填滿。
2. 桌面視窗貼上桌布。
3. 背景視窗繪出桌面圖示。
4. 依 Z order 為順序，由下而上，系統一一呼叫每個可見的最上層視窗繪製本身。
5. Shell 繪出工作列。

如此看來，若要達成在「桌布之上、桌面圖示之下」進行繪製的目的，並且能夠任意決定是否顯示桌布以及自訂桌面底色，必須在步驟三之前取得繪製短文的機會，同時控制

¹⁴ 若不清楚「桌面視窗」或「背景視窗」的定義，請參閱本章「桌面上的特殊視窗」小節。

步驟一及步驟二的進行動作才行。

控制桌面視窗是否繪製桌面底色、填圖樣式及桌布

步驟一及步驟二，也就是桌面視窗繪製桌面底色、填圖樣式及桌布的動作，是什麼狀況下進行的呢？答案是當桌面視窗收到 `WM_ERASEBKGD` 視窗訊息時，它才重繪這些背景圖案。

而桌面視窗在什麼情況下會收到 `WM_ERASEBKGD` 視窗訊息呢？必須是：

1. 畫面上有任何視窗移動、隱藏、縮小或最小化，使桌面原本看不見的區域顯露出來。
2. 此時系統會發送 `WM_ERASEBKGD` 視窗訊息給背景視窗，因為它是特殊視窗中的最上層視窗，換句話說，就是我們所看見的，最下層的視窗下方緊臨的那個視窗。
3. 一般來說，背景視窗不會處理此 `WM_ERASEBKGD` 訊息。此視窗訊息會不斷傳遞給父視窗，只要沒人處理，此視窗訊息就會不斷傳遞下去，從背景視窗送到 `SHELLDLL_DefView` 視窗，再送給 `Progman` 視窗，最後來到大頭目－桌面視窗為止，而桌面視窗一定會處理此訊息。
4. 桌面視窗於是進行繪製桌面底色、填圖樣式及桌布的動作。

瞭解訊息流程後，很簡單，只要將流經背景視窗的 `WM_ERASEBKGD` 視窗訊息攔下，桌面視窗就無從進行桌面圖案的繪製，而我們也可趁機自行繪製桌面底色及圖案。

攔截 `WM_ERASEBKGD` 視窗訊息

`WM_ERASEBKGD` 屬於「流動型」的視窗訊息，若視窗不處理它，則此訊息會繼續傳遞給它的父視窗處理。因為我們必須切換桌布的顯示與否以及動態地決定桌面底色，所以勢必得攔截 `WM_ERASEBKGD` 訊息，不使它無條件地傳送給桌面視窗處理。

攔截 `WM_ERASEBKGD` 訊息的好處，除了防止桌面視窗繪製桌面底色、填圖樣式及桌布外，也可以在此繪製自己的桌面。至於在哪邊攔截 `WM_ERASEBKGD` 訊息呢？我的

選擇是背景視窗，由於桌面圖示也是由它負責，攔截它可以得到額外的許多好處。以下是新的背景視窗 `WM_ERASEBKGD` 訊息處理函式虛擬碼，我們可以這樣來攔截：

```
#0001 void __fastcall NewSysListView32::WMEraseBkgnd(TWMEraseBkgnd&
#0002     Message)
#0003 {
#0004     if (bNeedShowDesktopWallPaper) // 如果需要秀出原有桌面
#0005         // 傳遞給桌面視窗去繪製桌面底色、填圖樣式及桌布
#0006         Dispatch to origin window procedure
#0007
#0008     // 進行自己的桌面繪製
#0009     ...
#0010
#0011     // 告知此 WM_ERASEBKGD 訊息處理完畢
#0012     Message.Result = 1;
#0013 }
```

在這裏進行的繪製結果，就成為桌面背景，理所當然地位於桌面圖示下方。但要注意的是，這邊的繪製動作由 `WM_ERASEBKGD` 視窗訊息所驅動，所以只有在特殊情形（如更換桌布、更改系統顏色或強制重繪）發生時才會重繪。而且，當此處重繪時，重繪區域上方的所有視窗或圖形也都需要重繪，這會造成十分嚴重的閃動效果，所以只適合繪製不常變動的圖案。經常變動的圖形必須交給由 `WM_PAINT` 視窗訊息所驅動的函式來繪製才行。

Note

「程式手法」這個小節所列的視窗訊息處理函式皆是虛擬碼。我們並不是要撰寫新的類別，所以在真正的程式碼中，將直接以視窗函式的型式來攔截訊息，而不是此處的類別視窗訊息函式攔截方式。「類別視窗訊息函式」攔截方式指的是在類別宣告中，使用由 `TObject` 類別所提供的訊息攔截方式，就像這樣：

```
BEGIN_MESSAGE_MAP
    VCL_MESSAGE_HANDLER(WM_FOOBAR, TMessage, WMFooBar);
END_MESSAGE_MAP(ParentClass);
```

更改背景視窗的繪製動作

背景視窗的視窗類別為 *SysListView32*，若想在繪製圖示之前，另外進行其它繪製動作，至少有兩種攔截方式：

攔截 WM_PAINT 視窗訊息

眾所皆知的 *WM_PAINT* 視窗訊息是所有視窗通用的繪製請求訊息。每當視窗的客戶端區域 (client area)¹⁵ 有任何部分需要重繪時，系統就會設定好需要重繪的無效區域，接著發出 *WM_PAINT* 訊息到該視窗所屬執行緒的訊息佇列中。視窗收到 *WM_PAINT* 訊息後，會先將無效區域合法化 (validate)，視情形送出 *WM_ERASEBKGD* 訊息先清除 (或繪製) 背景，再進行正常的繪製動作。

SysListView32 控制項的 *WM_PAINT* 訊息處理流程大致如下：

```
#0001 void __fastcall SysListView32::WMPaint(TWMPaint& Message)
#0002 {
#0003     TPaintStruct PaintStruct;
#0004
#0005     // 開始 WM_PAINT 處理, 取得 DC
#0006     HDC DC = BeginPaint(Wnd, &PaintStruct);
#0007
#0008     if (PaintStruct.fErase) // 如果需要清除背景
#0009         SendMessage(Wnd, WM_ERASEBKGD, DC, 0); // 清除背景
#0010
#0011     // 詢問父視窗是否要自行繪製
#0012     if (!AskParentWindowForCustomDraw)
#0013         DrawItems(); // 若父視窗不要自訂繪製, 則繪出所有的項目文字及圖示
#0014
#0015     EndPaint(Wnd, &PaintStruct); // 結束 WM_PAINT 處理
#0016 }
```

對於我們的需求而言，必須在桌面視窗的「桌布繪製」及背景視窗的「圖示繪製」動作之間進行自訂的繪製動作；換句話說，也就是在 0009 列 *WM_ERASEBKGD* 訊息處理之

¹⁵ 當視窗的非客戶端區域需要重繪時，系統會送出 *WM_NCPAINT* 視窗訊息。

後以及 0013 列呼叫 *DrawItems* 進行圖示繪製之前，加入自己的繪製工作程式碼。可是想了又想，似乎無論怎麼攔截，都沒有辦法將程式碼正確地擺到裡頭：

```
#0001 void __fastcall NewSysListView32::WMPaint(TWMPaint& Message)
#0002 {
#0003     call origin window procedure // 進行 SysListView32::WMPaint 處理
#0004
#0005     ... // 我們的繪製動作，太慢了，WM_PAINT 訊息處理完畢
#0006     // 1.無效區域資訊已被消除
#0007     // 2.桌面圖示已經畫上去，而我們必須比它早畫
#0008 }
```

或者

```
#0001 void __fastcall NewSysListView32::WMPaint(TWMPaint& Message)
#0002 {
#0003     ... // 我們的繪製動作，太早了，WM_ERASEBKGDND 訊息還未送出
#0004     // 現在即使畫了，待會可能會被 WM_ERASEBKGDND 訊息所觸發的
#0005     // 桌面繪製動作清除。
#0006
#0007     call origin window procedure // 進行 SysListView32::WMPaint 處理
#0008 }
```

以上兩種攔截方式都不行，一個太晚，一個太早。因此我們必須另覓它法，將目光移向 0012 列的 *AskParentWindowForCustomDraw* 步驟。

攔截傳遞給父視窗的 WM_NOTIFY 視窗訊息

WM_PAINT 訊息攔截不成，我們必須利用 *SysListView32* 視窗本身的 custom draw 支援機制。如上述 0012 列的虛擬碼，*SysListView32* 視窗在繪製節點之前，會先向它的父視窗發出詢問，讓父視窗決定是否進行 custom draw，也就是自訂繪製動作。

詢問的方式是由 *SysListView32* 控制項送出 WM_NOTIFY 訊息給父視窗，傳遞一個指向 TNMHDR 結構的指標，同時為該 TNMHDR 結構的 code 欄位指定 NM_CUSTOMDRAW 代碼。TNMHDR 結構定義如下：

```
typedef struct tagNMHDR {
    HWND hwndFrom; // 發送訊息的視窗 handle
    UINT idFrom;   // 發送訊息的控制項 ID
    UINT code;     // NM_XXXX 代碼
} NMHDR;
```

```
typedef NMHDR TNMHDR, *PNMHDR;
```

父視窗收到 *WM_NOTIFY* 訊息後，若發現 *TNMHDR::code* 欄位為 *NM_CUSTOMDRAW*，就會將指向 *TNMHDR* 結構的指標轉型為 *PNMLVCustomDraw* 結構指標型態來處理。*PNMLVCustomDraw* 指標型態指向 *TNMLVCustomDraw* 結構，結構定義如下：

```
typedef struct tagNMLVCUSTOMDRAW {
    NMCUSTOMDRAW nmcd;        // TNMCustomDraw 結構
    COLORREF clrText;         // 項目文字的颜色
    COLORREF clrTextBk;       // 項目文字的底色
    int iSubItem;              // 正在繪製的子項目 (subitem) 編號
} NMLVCUSTOMDRAW, *LPNMLVCUSTOMDRAW;

typedef NMLVCUSTOMDRAW TNMLVCustomDraw, *PNMLVCustomDraw;

typedef struct tagNMCUSTOMDRAWINFO {
    NMHDR hdr;                // TNMHDR 結構
    DWORD dwDrawStage;        // 繪製階段
    HDC hdc;                   // 控制項的 DC
    RECT rc;                   // 可繪製的矩形範圍
    DWORD_PTR dwItemSpec;     // 項目資訊
    UINT uItemState;          // 項目的狀態
    LPARAM lParam;            // 項目的資料
} NMCUSTOMDRAW, *LPNMCUSTOMDRAW;

typedef NMCUSTOMDRAW TNMCustomDraw, *PNMCustomDraw;
```

嗯，這種巨大的資料結構很嚇人吧，早點習慣也好，在 Win32 API 中打滾，遲早得遇到的。此時，父視窗就可根據隨 *WM_NOTIFY* 視窗訊息帶來的 *TNMLVCustomDraw* 結構，進行適當的自訂繪製動作：可以利用 *hdc* 欄位進行繪圖，也可以更改 *TNMLVCustomDraw* 結構的欄位，例如項目文字的颜色 *clrText* 等等，影響 *SysListView32* 視窗的繪製結果。不論動作為何，父視窗必須回傳某個數值，告知 *SysListView32* 視窗下一步該怎麼做：

- 若 *TNMCustomDraw::dwDrawStage* 欄位為 *CDDS_PREPAIN*
 - *CDRF_DODEFAULT*
讓子視窗完全依原有的方式繪製，父視窗不想參與。
 - *CDRF_NOTIFYITEMDRAW*
每次繪製項目時，讓父視窗參與。

- *CDRF_NOTIFYITEMERASE*
每次清除項目時，讓父視窗參與。
 - *CDRF_NOTIFYPOSTERASE*
每次清除項目後，通知父視窗。
 - *CDRF_NOTIFYPOSTPAINT*
每次繪製項目後，通知父視窗。
 - *CDRF_NOTIFYSUBITEMDRAW*
每次繪製子項目時，讓父視窗參與。
- 若 *TNMCustomDraw::dwDrawStage* 欄位為 *CDDS_ITEMPREPAINT*
- *CDRF_NEWFONT*
父視窗希望控制繪製項目時使用的字型及顏色。
 - *CDRF_SKIPDEFAULT*
子視窗不要繪製項目，完全由父視窗繪製。

沒辦法，爲了要控制背景視窗，非得先瞭解它的訊息傳遞行爲才行，現在我想各位對於 *SysListView32* 視窗的 *WM_PAINT* 訊息處理流程應該已經十分清楚了。而自行繪製動作的插入點也呼之欲出，就在 *SysListView32* 視窗的父視窗的 *WM_NOTIFY* 視窗訊息處理函式裏，只要攔截 *SysListView32* 的父視窗的視窗函式，處理 *SysListView32* 視窗的 custom draw 程序即可：

```
#0001 void __fastcall NewSysListView32ParentWindow::WMNotify(TWMNotify&
#0002     Message)
#0003 {
#0004     if (((PNMHDR)Message.lParam)->id == NM_CUSTOMDRAW) {
#0005         ...
#0006         // 我們自己的繪製動作
#0007
#0008         // 讓 SysListView32 進行正常的繪製動作，不再干涉
#0009         return CDRF_DODEFAULT;
#0010     }
#0011
#0012     // 繼續原本的 WM_NOTIFY 訊息處理動作
#0013     call origin window procedure
#0014 }
```

在 0005 ~ 0006 列這邊加入我們自己的繪製動作，因爲我們的繪製動作早於繪製項目的動作，所以繪製出來的圖形就會置於桌面圖示之下，這正是我們需要的結果。

位於桌面圖示下方的按鈕

我承認，這個要求十分奇特，我們什麼時候見過位於桌面圖示下方的視窗了？只要是正常的視窗，就會在桌面視窗及背景視窗上方，沒有人跑到背景視窗後面去的。這個問題並不是將按鈕放到背景視窗後就可解決，因為背景視窗會將按鈕蓋住，反而使按鈕看不見。我需要的是，不被背景視窗蓋住，但在背景視窗所繪製的桌面圖示下方的按鈕，嘿，嘿。

讓我們來考慮幾個可能的方案：

- 將按鈕擺在背景視窗下方
背景視窗會蓋住按鈕，看不到按鈕，不行。
- 將按鈕擺在背景視窗上方
按鈕會蓋住桌面圖示，不符合要求，不行。

嗚，兩個方案都簡單地一句話就被打死，是不是已經窮途末路了呢？仔細想一想，我再提一個方案：

- 將按鈕擺在背景視窗下方，把按鈕當作桌面背景「畫出來」。

這個方案聽起來似乎可行，若這樣做，真的可造出「在桌面圖示下方的按鈕」，只要記得在新的背景視窗視窗函式處理 `WM_ERASEBKGD` 訊息時，順便送個 `WM_PAINT` 訊息給按鈕，讓按鈕將本身的圖形畫出來就行了。

不過，在背景視窗下方的按鈕，雖然看得到，但事實上只是圖形而已，背景視窗其實蓋在真正的按鈕上面。那麼，當我在按鈕「圖形」處按下滑鼠左鍵時，按鈕「本尊」知道這件事情嗎？又，當真送個 `WM_PAINT` 訊息給按鈕，按鈕就會乖乖地畫在桌面上了嗎？接下來，我們分別討論按鈕產生、繪製及操作上的問題。

產生位於背景視窗下方的按鈕

假設此按鈕名為 *ConfigButton*，類別為 *TConfigButton*，為 *TButton* 的後代類別。那麼，如何產生一個 *TConfigButton* 按鈕，將它恰恰置於背景視窗的下方呢？

首先是父視窗的選擇，既然要擺在背景視窗下方，那就不能以背景視窗為父視窗，所以我選定背景視窗的父視窗做為 *ConfigButton* 按鈕的父視窗；換句話說，讓 *ConfigButton* 按鈕成為背景視窗的「哥哥」。

至於產生此 *ConfigButton* 按鈕的方法，我已在「桌面隨意貼」一節提過：建立元件時，若其父視窗不是 VCL 元件，就必須改用傳入一個 *HWND* 參數的建構函式，它會以此參數作為此元件的 *ParentWindow*。所以它是這樣建立的：

```
ConfigButton = new TConfigButton(SysListView32_ParentWnd);
```

繪製按鈕圖形

處理 *SysListView32* 視窗接收的 *WM_ERASEBKGD* 訊息時，首先進行自己的桌面繪製動作後，接著再呼叫 *ConfigButton* 的 *Invalidate* 及 *Update* 函式，強制它立即重繪。程式碼會像這樣：

```
#0001 void __fastcall NewSysListView32::WMEraseBkgnd(TWMEraseBkgnd&
#0002     Message)
#0003 {
#0004     if (bNeedShowDesktopWallPaper) // 如果需要秀出原有桌面
#0005         // 傳遞給桌面視窗去繪製桌面底色、填圖樣式及桌布
#0006         call origin window procedure
#0007
#0008     // 進行自己的桌面繪製
#0009     ...
#0010
#0011     // 繪製按鈕"圖形"
#0012     if (ConfigButton != NULL) {
#0013         // 使按鈕區域成為無效區域
#0014         ConfigButton->Invalidate();
#0015         // 呼叫按鈕的視窗函式，立即處理 WM_PAINT 訊息
#0016         ConfigButton->Update();
#0017     }
#0018
#0019     // 告知此 WM_ERASEBKGD 訊息處理完畢
```

```
#0020     Message.Result = 1;  
#0021 }
```

看起來沒有問題，可是實際測試的結果讓人十分失望—完全看不到 *ConfigButton* 按鈕，明明傳送 *WM_PAINT* 視窗訊息給它，請它重新繪製了說！

原因是，有一個我們甚少碰觸的視窗風格旗標在做怪，它叫做 *WS_CLIPSIBLINGS*。通常絕大部分視窗的視窗風格，包括我們所有的 VCL 視窗控制項，預設情況下都包含著 *WS_CLIPSIBLINGS* 旗標。它的作用是，若有兄弟視窗覆蓋住視窗本身，則在 *WM_PAINT* 視窗訊息處理前，系統會主動將此視窗被覆蓋的部分從無效區域中除去。

以下圖為例，*Button1* 元件有部分區域被 *Button2* 元件覆蓋。因為 *Button1* 元件的視窗風格包含 *WS_CLIPSIBLINGS* 旗標，所以 *Button1* 繪製本身時，將不會重繪被 *Button2* 蓋住的那塊區域，要不然只要 *Button1* 一重繪，反而就變成 *Button1* 蓋住 *Button2* 了。由此可見，*WS_CLIPSIBLINGS* 視窗風格旗標正是實作視窗 Z order 的機制之一。

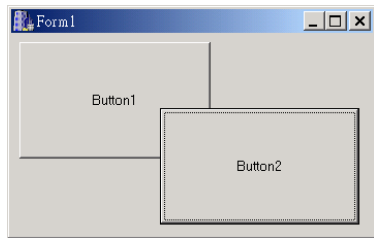


圖 5-12 / Button2 蓋住 Button1

結論是，只要將 *TConfigButton* 的 *WS_CLIPSIBLINGS* 視窗風格旗標移除，使 *ConfigButton* 成為暴力份子。當它重繪時，不管三七二十一，不管 Z order，也不管長幼有序，畫就對了。而這正好符合我們的需求，如此一來，*ConfigButton* 就會畫在背景視窗上，顯示出來。

不過，萬一有某個桌面圖示正好待在 *ConfigButton* 按鈕上方，當 *ConfigButton* 一重繪，按鈕就會蓋住原本的桌面圖示，所以按鈕重繪之後，還要強制按鈕區域的 *SysListView32* 視窗重繪，以確保桌面圖示不會被按鈕覆蓋。

至此為止，大致已瞭解 *TConfigButton* 的特殊需求，先將它實作出來：

```
#0001 class TConfigButton : public TButton { // 設定按鈕類別
#0002 private:
#0003     void __fastcall WMPaint(TWMPaint& Message);
#0004 protected:
#0005     virtual void __fastcall CreateParams(TCreateParams &Params);
#0006 public:
#0007
#0008 BEGIN_MESSAGE_MAP
#0009     VCL_MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint);
#0010 END_MESSAGE_MAP(TButton);
#0011 };
#0012
#0013 void __fastcall TConfigButton::WMPaint(TWMPaint& Message)
#0014 {
#0015     TButton::Dispatch(&Message);
#0016
#0017     TRect R;
#0018
#0019     // 防止按鈕本身蓋在桌面圖示上方，若按鈕重繪時，
#0020     // 也重繪其上的桌面圖示
#0021     R = BoundsRect;
#0022     InvalidateRect(g_Data->Wnd, &R, false);
#0023 }
#0024
#0025 void __fastcall TConfigButton::CreateParams(TCreateParams &Params)
#0026 {
#0027     TButton::CreateParams(Params);
#0028
#0029     // 不需要管兄弟視窗，照畫不誤
#0030     Params.Style = Params.Style & ~WS_CLIPSIBLINGS & ~WS_BORDER &
#0031     ~WS_DLGFRAME;
#0032 }
```

按鈕的操作

既然按鈕實際上位於背景視窗下方，那麼它就沒有理由接收到任何滑鼠訊息，所有看似按在按鈕上方的滑鼠訊息，都跑到背景視窗去了。所以，送佛送上西天，背景視窗還得負責將按鈕區域的滑鼠訊息轉送給按鈕本身，好像滑鼠真的按在按鈕上似的。

因此，subclass 背景視窗後，必須改寫它的視窗函式，檢查所有收到的視窗訊息，若視窗訊息的編號為 `WM_MOUSEFIRST` 至 `WM_MOUSELAST` 之間，表示此訊息為滑鼠操作相關訊息，則進行以下處理：

1. 取得訊息發生時的滑鼠指標位置。
2. 檢查是否按到背景視窗本身的任何項目或子項目，如果是則跳離。
3. 檢查是否按在 *ConfigButton* 按鈕上，如果不是則跳離。
4. 將滑鼠座標換算為 *ConfigButton* 按鈕客戶端區域的相對座標。
5. 呼叫 *SendMessage* 函式將視窗訊息轉交給 *ConfigButton* 按鈕。
6. 重繪本身（桌面視窗）的按鈕所在區域。

如此一來，雖然我們所看到的按鈕是假的，由背景視窗的 *WM_ERASEBKGD* 訊息處理程式畫出來的，但是它能夠接收、回應所有的滑鼠訊息，也會出現「壓下」、「彈起」狀態。看起來就跟真的一樣，「以假亂真」這句成語，今天真實地感受到了。:p

程式手法小結

1. 攔截背景視窗的父視窗的 *WM_NOTIFY* 訊息，繪製每秒鐘更動的目前時間。
2. 攔截背景視窗的 *WM_ERASEBKGD* 訊息以繪製內容不常變動的短文；同時控制桌布及桌面底色、填圖樣式的顯示與否。
3. 建立 *TConfigButton* 設定按鈕時，以背景視窗的父視窗為父視窗，即讓 *ConfigButton* 成為背景視窗的哥哥。
4. 攔截背景視窗的 *WM_MOUSEFIRST* ~ *WM_MOUSELAST* 訊息，將欲傳送給設定按鈕的滑鼠訊息轉交給 *ConfigButton*。

使用技術

由以上歸納的程式手法看來，並沒有什麼特別之處，只有兩個主要動作：

- 攔截桌面視窗的視窗訊息
- 攔截桌面視窗的父視窗的視窗訊息

都是攔截視窗訊息而已，需要牽扯到什麼深奧的技術嗎？道理是這樣沒錯，不過說來容易做起來難，實作上總會有許多問題不斷衍生，繼續往下看，很快你就會明白。

在這之前，你必須對下列主題有基本的認識，才有足夠的技術背景進行接下來的討論：

- **Subclassing**

更換既有視窗的視窗函式，以攔截視窗訊息或改變視窗的行為。

- **Hook**

作業系統提供的事件攔截機制，可得知或截下各種訊息或事件的發生。

- **Memory-mapped file**

記憶體映射檔案，便於檔案的讀取及處理，同時可使多個行程共享記憶體。

如果對它們仍不甚清楚，請自行閱讀相關的 SDK 書籍。

Subclassing

由於想要攔截訊息的目標視窗並不是 VCL 元件，而且攔截的目標是早已存在的視窗，因此必須使用 subclassing 技術。只要運用 subclassing，將欲攔截訊息的桌面視窗及其父視窗的視窗函式移花接木，指向我們事先準備好的視窗函式，就可以攔截任何傳遞給它們的視窗訊息。

跨行程的 subclassing

但是，問題來了。Subclassing 動作雖然簡單：先利用 *GetWindowLong* 函式取得原來的視窗函式位址，備份起來，再呼叫 *SetWindowLong* 函式設定新的視窗函式位址。但是在 Win32 下，每一個行程擁有獨立的記憶體空間，所以新的視窗函式必須位於同一個行程內，視窗才有辦法叫用到它。

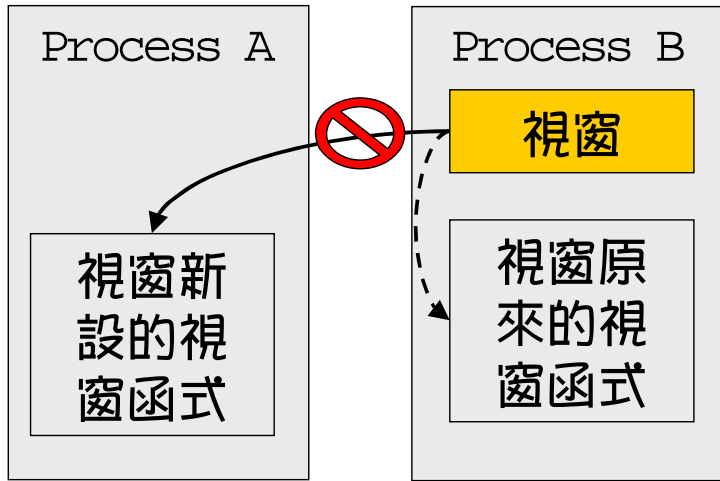


圖 5-13 / 跨行程的 subclassing 動作是不合法的

道理很簡單，若行程 A 想要 subclass 一個行程 B 裡頭的視窗，它準備好一個視窗函式，假設此函式位址為 0x42345678，它將這個位址丟給行程 B 的視窗，設定為該視窗的視窗函式。可是行程 B 的 0x42345678 位址與行程 A 的 0x42345678 位址是不相干的，也許指向某個變數、某道指令的中間、甚至是不合法、未配置的記憶體，一旦真的在程式 B 中執行這個視窗函式，十有八九會引起記憶體存取例外，剩下一成的機率大概是非法指令之類的錯誤。

以背景視窗為例，它是由 EXPLORER 行程建立的，所以它的視窗函式及所有程式碼皆位於 EXPLORER 行程內部。但是我們提供的視窗函式位於自己的行程，背景視窗所屬的執行緒並沒有辦法執行位於我們自己行程內的程式碼。

所以，SetWindowLong API 函式會防止這類動作的發生：呼叫 SetWindowLong 函式且傳入 GWL_WNDPROC 參數設定視窗函式時，若呼叫 SetWindowLong 函式的執行緒與將要更改視窗函式的視窗分別處於不同的行程，這個呼叫將會失敗，傳回零。

不支援跨行程 subclassing 的理由

目前的 Win32 作業系統皆不支援跨行程的 subclassing 動作，因為：

- 這種需求並不常見，絕大部分的 subclassing 動作都發生在同一行程內。
- 若跨行程 subclassing 是合法動作，當視窗所屬的執行緒欲傳送訊息給該視窗的視窗函式（藉由 *SendMessage* 函式）時，或者視窗所屬執行緒的訊息迴圈呼叫 *DispatchMessage* 函式分派訊息時，勢必要進行 context switch 轉換到另一個行程內的另一個執行緒執行視窗函式，而大量的 context switch 動作將導致系統效率的降低。
- 若跨行程 subclassing 是合法動作，當要執行位於另一行程式的視窗函式時，必須切換到該行程內的執行緒（才能夠存取該行程的記憶體），但是如果對方行程擁有多個執行緒，要切換到哪個執行緒去執行呢？還是要建立一個新的執行緒來執行呢？

當然還有比較省事的做法。只要作業系統注意到跨行程的 subclassing 動作發生時，主動將新的視窗函式程式碼拷貝到被 subclass 的視窗的行程內，或是使用記憶體映射機制，將兩行程的視窗函式對應到相同的位址，那麼，當分派視窗訊息到新的視窗函式時，就不必進行 context switch 來切換執行緒了。

顯然，這兩個方案還是有問題的，因為：

- 拷貝程式碼
這方法太浪費記憶體了。在十分有限的實體記憶體內放置兩份（或更多份）不可能變更內容的程式碼是十分不智的舉動，若哪套作業系統真的這樣做，保證被眾人口誅筆伐，千刀萬里追。
- 對應到相同的位址
問題是，何時進行對應到相同位址的動作？如果對方行程的相同位址已放置資料，無法直接進行對應，該怎麼辦？如果將視窗函式搬到另一個雙方都未使用的位址，會影響包含新視窗函式行程的正常運作，例如：

```
Ptr = &NewWindowProc;  
SetWindowLong(Wnd_In_Another_Process, GWL_WNDPROC, &NewWindowProc);  
// 假設 NewWindowProc 被搬動了，此時 Ptr 變數會指向什麼？  
// 要如何修正 Ptr 變數的內容呢？如果有一百個值為 &NewWindowProc 的變數呢？
```

所以，即使勉強支援得來，但所增加的負擔實在太多，所以 Win32 並不支援跨行程的 subclassing 機制。

若真有此需求的話，也行，不過程式設計師得自己動手來處理許多細節，Win32 程式設計又變得更複雜了。

Hook

解決跨行程 subclassing 問題的最佳解法是：將新的視窗函式及 subclass 的動作程式碼塞入被 subclass 視窗的行程內。這樣做的結果是，跨行程的 subclassing 動作變成行程內的 subclassing 動作，由「跨行程」三個字衍生出來的問題就完全解決了，請見下圖。

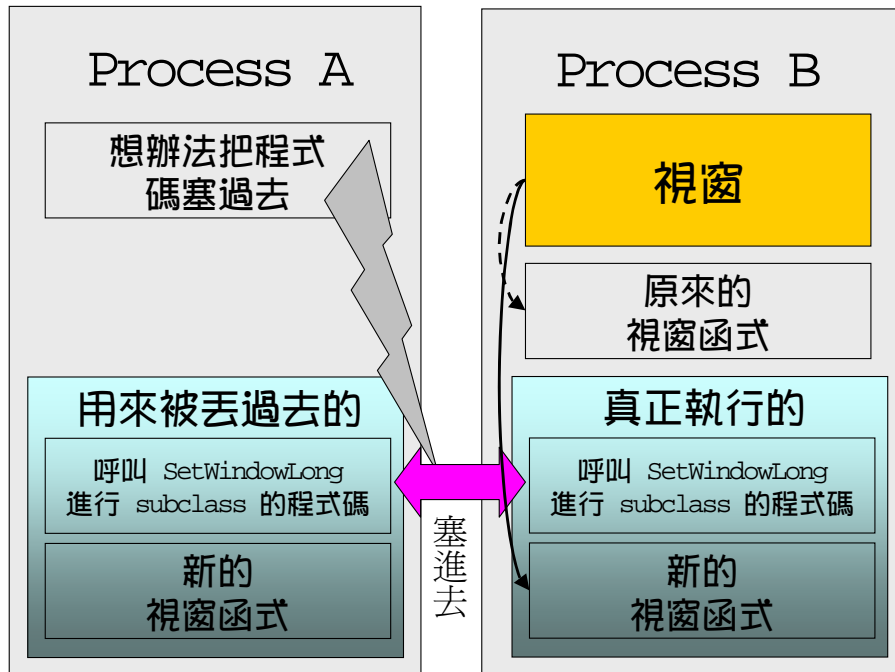


圖 5-14 / 跨行程 subclassing 的解決方法

如何塞入程式碼？

不過，怎麼把程式碼「塞」過去呢？Jeffrey Richter 是這方面的專家，他將此種「塞」程

式碼的動作稱為「DLL 注射」(DLL inject)，此名稱的來源是因為解決方法皆必須依賴 DLL 的行程內部載入特性，才能將程式碼放到另一個行程裡頭。在他的 *Programming Applications for Microsoft Windows* 一書中，Jeffrey Richter 特別以「Breaking Through Process Boundary Walls」整個章節來講解 DLL 注射的各種作法。他提出三種不同的作法，分別是：

- 利用登錄資料庫設定來注射 DLL
- 利用 hook 來注射 DLL
- 利用遠端執行緒來注射 DLL

利用登錄資料庫設定的方法，DLL 會被強迫注射到所有使用 USER32 模組的行程中，而且更改登錄設定值後，必須重新開機才能啟用注射功能，太麻煩而且牽扯到不必要的行程。至於利用遠端執行緒的方法，必須呼叫 *CreateRemoteThread* API 函式，而此函式在 Windows 95/98 中並沒有實作，所以此技術不能用在 Windows 95/98 上，不符合我們的需求。

以刪去法除掉兩個不適合的方法後，別無選擇，只剩下一個方法了。因此在 Desktop Illusionist 程式中，我將透過 hook 機制來進行 DLL 注射，將 subclassing 動作程式碼及新的視窗函式丟到背景視窗的行程內（即 EXPLORER.EXE）。至於對其它兩種 DLL 注射方法有興趣的朋友，請自行閱讀 Jeffrey Richter 的 *Programming Applications for Microsoft Windows*。

爲什麼 Hook 會注射程式碼？

在這兒我們使用的是 *WH_GETMESSAGE* hook，它的目的是監看丟到目標執行緒訊息佇列的所有訊息。安裝 hook 時，必須傳入一個指向函式的指標，此函式稱為 hook 函式。每當被監看的執行緒訊息 佇列有任何訊息被取出時，hook 函式就會被呼叫，傳入關於此視窗訊息的所有資料。我們可以在 hook 函式中檢視、記錄或修改存放視窗訊息資訊的 *TMsg* 結構。

Hook 函式必須爲 *GetMsgProc* 型別：

```
LRESULT CALLBACK (*GetMsgProc)(int code, WPARAM wParam, LPARAM lParam);
```

準備好 hook 函式後，呼叫 *SetWindowsHookEx* 函式即可安裝 hook：

```
// GetMsgProc 為 hook 函式指標  
// hDLLInstance 為包含 hook 函式的 DLL instance  
// ThreadID 為要監看的訊息佇列所屬的執行緒 ID，若為零，表示要監看系統中所有執行佇列  
hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hDLLInstance,  
    ThreadId);
```

假設行程 A 呼叫 *SetWindowsHookEx* 函式時，將行程 B 的某個執行緒 ID 傳入，hook 被成功地「掛上」。接著，每當行程 B 的該執行緒呼叫 *GetMessage* 或 *PeekMessage* API 函式，欲從訊息佇列取出任何視窗訊息時，就會進行下列處理：

1. 系統發現此訊息佇列所屬的執行緒被掛上 *WH_GETMESSAGE* hook。
2. 系統檢查註冊的 hook 函式是否存在於行程 B 的記憶體空間內。
3. 如果沒有，那麼系統會將包含 hook 函式的 DLL 載入到行程 B，並將此 DLL 在行程 B 的使用計數加一。
4. 系統計算 hook 函式在行程 B 內的位址。由於 DLL 在行程 A 和行程 B 的載入基底位址可能不同，所以對於兩行程來說，hook 函式的位址也可能不同。
5. 將此 DLL 在行程 B 的使用計數加一。
6. 呼叫位於行程 B 內的 hook 函式，傳入視窗訊息結構。
7. 將此 DLL 在行程 B 的使用計數減一。

最後，當行程 A 完成任務，不想再監看行程 B 該執行緒訊息佇列的視窗訊息時，只要呼叫 *UnhookWindowsHookEx* 函式解除 hook：

```
UnhookWindowsHookEx(hHook);
```

此時hook解除¹⁶，包含hook函式的DLL在行程B的使用計數也減一。若行程B沒有另行載入包含hook函式的DLL，該DLL在行程B的使用計數會變成零，被系統釋放掉。

¹⁶ 若hook函式當時正在執行，則系統會等到此次hook函式執行結束後，才卸除hook。

Hook 會將包含 hook 函式的 DLL 載入進來的原因很簡單，若是 DLL 未載入，那麼 hook 函式根本不存在行程 B 的位址空間內，系統要如何叫用呢？必須將 DLL 載入後才能叫用 hook 函式。

從以上的流程中可看到，由於 hook 函式的關係，包含 hook 函式的整個 DLL 都會被載入行程 B 中，真是典型的「一人得道，雞犬升天」。所以，透過 hook 機制來注射 DLL，沒有辦法準確控制要將 DLL 的哪些程式碼注射到其它行程中，要就整個 DLL 放進去，不然就拉倒。整個流程及架構的示意圖如下：

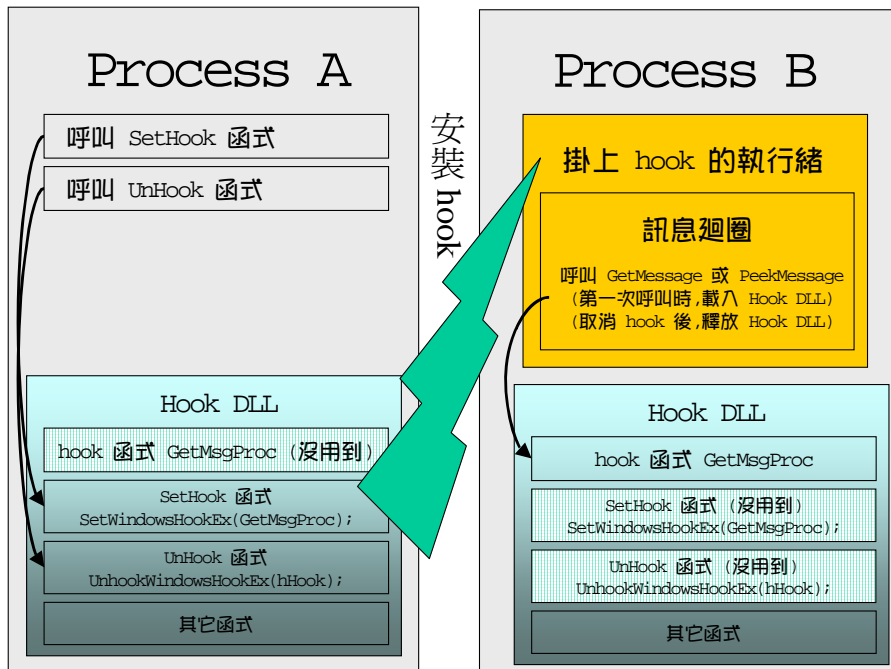


圖 5-15 / 以 hook 將整個 DLL 注射到另外一個行程

程式的切割

由上圖可看出，若要將某個 DLL 放入別的程序中，除了撰寫包含 `SetWindowsHookEx`、`UnhookWindowsHookEx` 呼叫及 hook 函式的 DLL 外，還必須另外撰寫一個負責啟動 hook

及取消 hook 的載入程式（也就是上圖的行程 A）。這個載入程式最重要的功能，就是呼叫 DLL 所提供的設定及取消 hook 的函式（如上圖的 *SetHook* 及 *UnHook* 函式）。

為什麼 *SetWindowsHookEx* 及 *UnhookWindowsHookEx* 呼叫會放在 DLL 內部呢？其實放在哪邊都行，只是放在 DLL 裡會比較方便。主要的原因是，hook 函式必須放在 DLL 裡，而呼叫 *SetWindowsHookEx* API 函式時，必須傳入 hook 函式的位址，所以：

- 若在 DLL 中呼叫 *SetWindowsHookEx* 函式，可以直接傳入 hook 函式的位址，不必另外取得。
- 若在程式中呼叫 *SetWindowsHookEx* 函式，必須透過 *GetProcAddress* API 或 DLL 另外提供的函式取得 hook 函式的位址才行。
- 通常在 hook 安裝之後，還必須進行其它的相關設定，並不是單單一行 *SetWindowsHookEx* 呼叫而已，對於這些動作複雜且重用性高的程式碼，放在 DLL 較為合適。

那麼，Desktop Illusionist 至少就得切割成兩個程式，分別為是載入程式及 Hook DLL。它們的工作分配為：

1. 載入程式

- 呼叫 DLL 匯出的設定及取消 hook 等函式。
- 提供使用者介面讓使用者切換選項、輸入文字等等，並將設定值及資料提供給 Hook DLL。

2. Hook DLL

- 提供 hook 函式，負責 subclass/unsubclass 背景視窗及其父視窗。
- 提供新的背景視窗視窗函式及新的背景視窗父視窗視窗函式。
- 匯出設定及取消 hook 的函式，供載入程式叫用。
- 接收由載入程式傳遞過來的選項切換及文字輸入，傳遞給位於 EXPLORER 行程內部的 Hook DLL。

subclass 的合法性及時機

終於，在 Hook DLL 的任務中看到 subclass 背景視窗及其父視窗這麼一項。因為 hook 函式一定只由被掛上 hook 的執行緒來執行，所以當 hook 函式被呼叫時，一定處於

EXPLORER 行程內。又，背景視窗和它的父視窗都是 EXPLORER 行程產生的，所以終於可以放心地 subclass 背景視窗及其父視窗了。

那麼，應該在什麼時候 subclass 視窗？又應該在什麼時候 unsubclass 視窗（即還原視窗函式）？subclassing 及 unsubclassing 的時機有幾點考量：

- Subclassing 必須在 hook 函式內進行，此地才可確保處於 EXPLORER 行程內。
- Unsubclassing 也必須在 EXPLORER 行程內進行，除了 hook 函式被呼叫時，當視窗函式被呼叫時，也可確定當時處於 EXPLORER 行程內。但在視窗函式內進行 unsubclass 總是比較麻煩，所以也決定由 hook 函式來進行 unsubclass 動作。
- Unsubclassing 必須在 hook 解除前完成。若 hook 解除前尚未還原視窗函式，一旦 hook 解除，新的視窗函式連同 Hook DLL 被釋放，一旦有訊息傳送至該視窗，欲執行視窗函式時，就會引發記憶體存取例外。

Hook/unhook/subclassing/unsubclassing 四項動作皆是功能強大，毫不留情的技巧，換句話說，萬一程式進行時有任何差錯，程式通常會當得很慘。所以在這裡，我們先將所有動作的流程規劃好，實際撰寫程式時，再依流程一步步實作，一方面培養事前規劃的能力，另一方面又可確保程式至少不會在邏輯及行進步驟上出差錯。安裝 hook 及 subclass 視窗的流程如下：

1. 載入程式

呼叫 Hook DLL 提供的 *SetDeskHook* 函式，傳入 *true*。

2. *SetDeskHook* 函式

呼叫 *SetWindowsHookEx* API 函式，掛上 hook。

3. *SetDeskHook* 函式

Hook 成功掛上後，投遞一個無用的視窗訊息 *WM_NULL* 給被 hook 的執行緒，接著進入自己的訊息等待迴圈，直到收到 *WM_MYHOOK* 訊息爲此。

4. *GetMsgProc* 函式

因爲 hook 已經掛上，再加上監看的訊息佇列被丟入 *WM_NULL* 訊息。很快地，訊息迴圈呼叫 *GetMessage* 或 *PeekMessage* 取出訊息時（不一定是 *SetDeskHook* 函式送的

WM_NULL 訊息，有可能正好有人給你早一步送達，也有可能訊息佇列裡還有未被處理的訊息），GetMsgProc 函式會被呼叫。GetMsgProc 函式必須檢查這是不是它第一次被叫用，如果是的話，就進行 subclassing 動作，將背景視窗（SysListView32 視窗）及背景視窗的父視窗（SHELLDLL_DefView 視窗）的視窗函式換成我們自己的。

5. GetMsgProc 函式

Subclassing 完成後，送出 WM_MYHOOK 自訂訊息給位於載入程式內的 Hook DLL，SetDeskHook 函式正在那兒等著。

6. SetDeskHook 函式

收到 WM_MYHOOK 訊息，hook 及 subclass 動作皆告完成。

這是安裝 hook 及進行 subclassing 動作的流程，而 unsubclassing 及解除 hook 的動作也採類似的流程，就不再詳細條列，所有的 hook/unhook/subclassing/unsubclassing 動作皆可由下圖看出究竟。

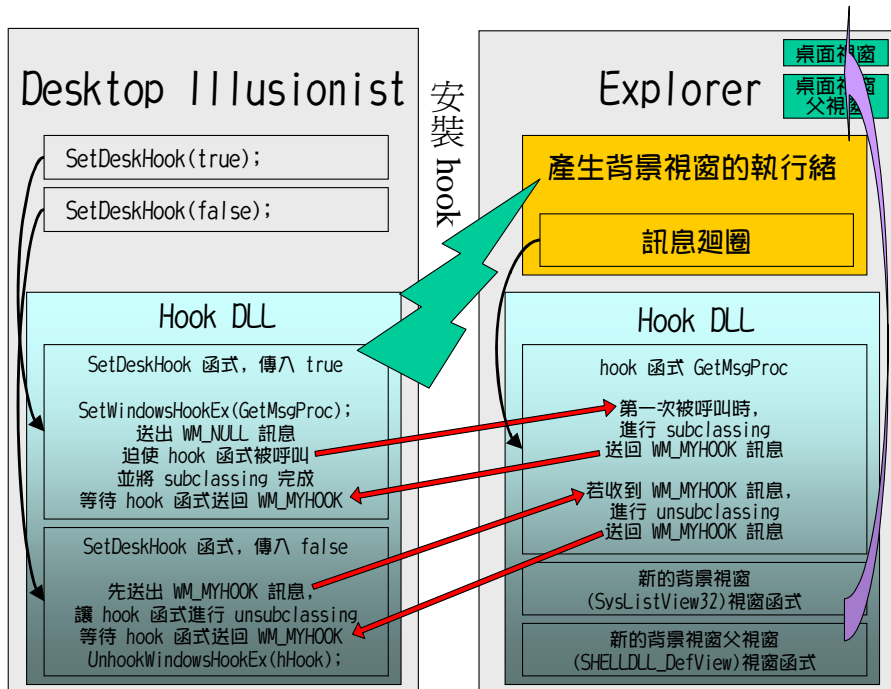


圖 5-16 / Hook / Unhook / Subclassing / Unsubclassing 動作流程

Hook Handle 的問題

截至目前為止，看起來進展十分順利，似乎馬上就可以將背景視窗和它的父視窗 subclass，達成控制桌面的目的。不過，事實上我一直隱瞞著一個挺嚴重的問題，現在就老實跟你們招了吧。

是這樣的，在每個 hook 函式中，正常的處理動作完成後，通常會呼叫 *CallNextHookEx* 函式，將訊息或其它事件繼續傳遞下去，可能是其它 hook 函式，也可能回到事件原本的流程。*CallNextHookEx* 的函式原型如下：

```
LRESULT CallNextHookEx(HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam);
```

除了第一個參數 *hhk*，意指 hook handle 外，其它三個參數都會在呼叫 hook 函式時一併傳入，hook 函式只要照著傳就行了，問題在於第一個參數：hook handle。

當載入程式呼叫 Hook DLL 提供的 *SetDeskHook* 函式，間接呼叫 *SetWindowsHookEx* 函式安裝 hook 時，*SetWindowsHookEx* 函式會回傳 hook handle，我們可用一個 *HHOOK* 型態的變數來接收它。hook 一旦成功掛上，背景視窗執行緒隨時可能將 Hook DLL 載入到 EXPLORER 行程，並呼叫 hook 函式，而 hook 函式在呼叫 *CallNextHookEx* 時，必須傳入 hook handle。

但是，hook 函式怎麼知道 hook handle 是多少？目前只有位於載入程式內部的那一份 Hook DLL 得知，至於位於 EXPLORER 行程內部的那一份 Hook DLL，雖然是雙胞胎兄弟，但是打死他也不知道。這到底是什麼情景呢？請看圖 5-17。

Info

其實，在 Windows NT 下呼叫 *CallNextHookEx* 函式時，不論你傳入的 hook handle 參數是否正確，*CallNextHookEx* 函式都可以順利地運作。這是因為 Windows NT 自行維護所有 hook 及 hook 函式資訊，不需要依賴此 hook handle 參數。

但是在 Windows 95/98 下就沒這麼好康了，呼叫 *CallNextHookEx* 函式時，必須乖乖傳入正確的 hook handle 才行。

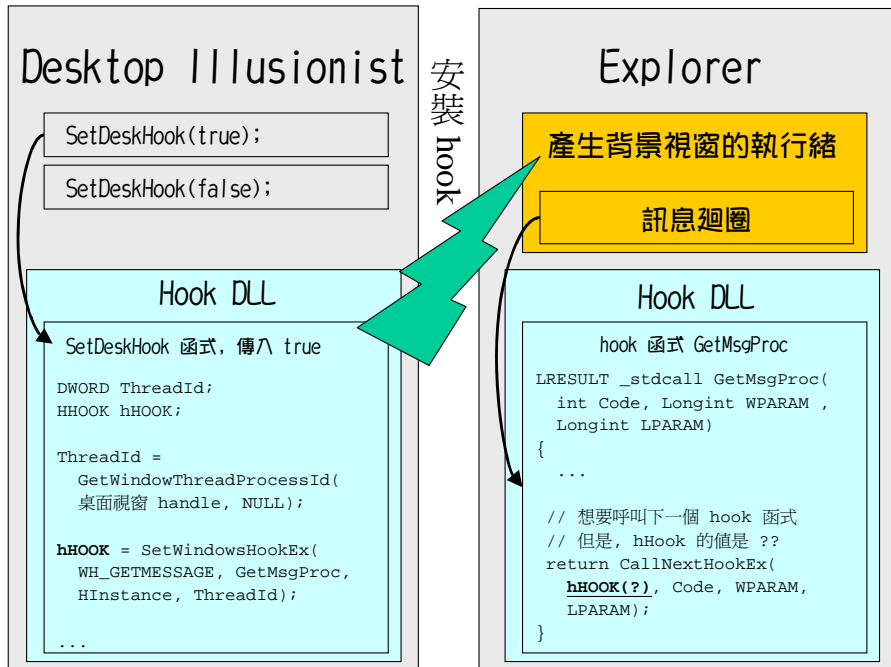


圖 5-17 / Hook 函式被呼叫時，無從得知 hook handle 值

因此，我們必須再提供一個機制，使載入程式所用的 Hook DLL 的 *hHook* 變數值，能與 EXPLORER 所用的 Hook DLL 分享。有幾個方法列入考慮：

□ 使用共享資料節區

有許多 C++ 編譯器／連結程式皆支援共享資料節區的建立。以 Visual C++ 為例，只要如下撰寫：

```
#pragma data_seg("Shared")
HHOOK hHook = NULL; // 對於各個行程來說，此 hHook 是同一個變數
#pragma data_seg()

#pragma comment(linker, "/section:Shared,rws");
```

就可以建立名為 *Shared* 的可讀、可寫、可在各行程間共享的資料節區。這是在行程間共享資料最方便的方法。很可惜的是，C++Builder 的編譯器並不支援這項能力，嘗試與其它編譯器合作使用也告失敗，只好望著 C++Builder 嘆氣，另覓它法了！

□ 將 hook handle 存入視窗的 property list

視窗的 property list 可以供我們寫入 *Handle* 型態的數值，所以正好可將 hook handle 存入載入程式主視窗的 property list，由於 Hook DLL 一定曉得載入程式視窗如何取得（藉由 *FindWindow* 等函式），所以此法可行。不過，它不能適用於共享資料可能隨時變更的情況。

□ 共享記憶體映射區域

既然不能直接使用由編譯器、連結程式提供的共享資料節區，那麼就自力救濟，自行建立記憶體映射區域。只要各行程的 Hook DLL 皆開啓同一個記憶體映射區域，就可達成共享記憶體的效果。它的好處是，要共享多少資料就可共享多少資料，不限資料型別，不限資料項數目，所以還可兼任兩份 Hook DLL 之間資料傳遞機制。

考量的結果，我決定使用共享記憶體映射區域的方式，一方面解決 hook handle 傳遞的問題，另一方面也同時解決在兩份 Hook DLL 之間傳輸大量資料的需求。你很快就可以看到，短文、字型、顏色等由 Desktop Illusionist 載入程式而來的設定資料並不少，若是沒有共享記憶體機制的協助，還真不曉得該用什麼方法來傳遞呢¹⁷。

¹⁷ 若不使用共享記憶體映射區域的方法，一般會透過 *WM_COPYDATA* 視窗訊息以跨越行程傳遞資料。

記憶體映射檔案

藉由 xMemory 單元提供的兩支函式 – *MapGlobalData* 及 *ReleaseGlobalData*，只要事先定義好準備共享的資料結構，就可以很方便地享用共享記憶體所帶來的便利。

```
// 全域資料區域，由所有 DLL instance 共享
typedef struct {
    HHOOK HHOOK; // Hook handle
} TGlobalData, *PGlobalData;
```

在 Hook DLL 中，宣告 *PGlobalData* 型別的 *g_Data* 變數，指向共享的 *TGlobalData* 結構。只要分別在 DLL 的載入點及釋放點建立及釋放記憶體映射區域，並將 *g_Data* 指向共享區域，不論在哪个行程中，存放 hook handle 的 *g_Data->hHook* 變數將永遠指向相同的四個位元組，位於 EXPLORER 行程的 hook 函式自然就沒有不曉得 hook handle 的道理囉。

問題似乎解決了，目前的情況如下圖：

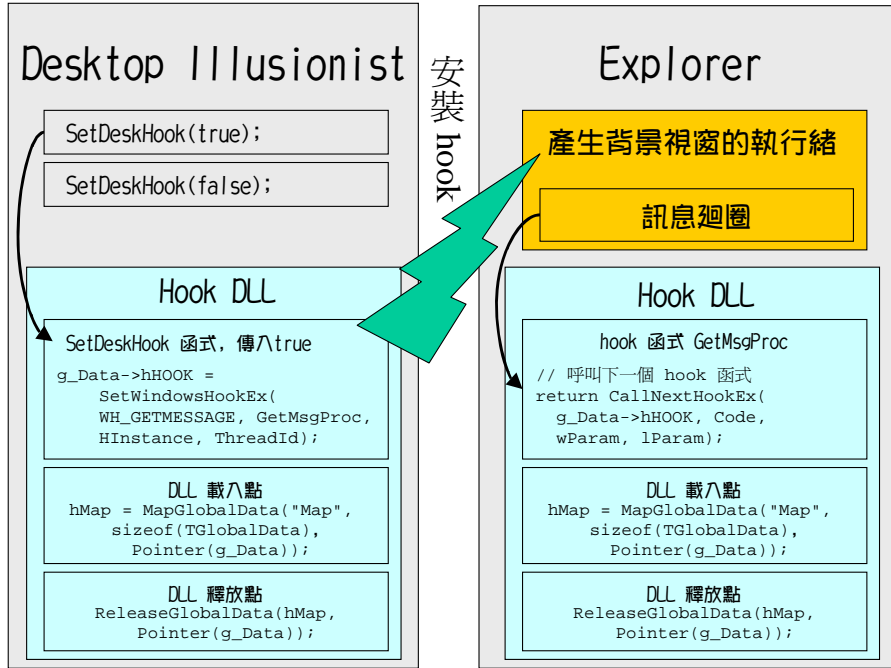


圖 5-18 / 以共享記憶體解決 hook handle 傳遞問題

同步問題

在 *SetHook* 函式中，呼叫 *SetWindowsHookEx* 函式後，將它的傳回值指定給 *hHook* 變數：

```
g_Data->hHOOK = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
    HInstance, ThreadId);
```

SetWindowsHookEx 函式一旦呼叫，hook 掛上之後，在那一瞬間，呼叫 *SetWindowsHookEx* 函式的執行緒可能剛好 time slice 結束，執行權被奪走，此時 *SetWindowsHookEx* 函式尚未返回，所以 *hHook* 變數的指定也還沒執行。

就是這麼剛好，被掛上 hook 的目標執行緒的訊息佇列裡尚有視窗訊息未處理，所以當目標執行緒呼叫 *GetMessage* 或 *PeekMessage* 函式取出訊息時，Hook DLL 被載入，接著 hook 函式被呼叫。在 hook 函式中，呼叫 *CallNextHookEx* 函式時，不是必須傳入 *hHook* 變數嗎？但是請注意，由於 *hHook* 變數尚未指定，所以此時 *hHook* 變數並不是我們所預期的

hook handle 呢！？

這真是很糟糕的事，發生的可能性雖然不高，可是還是有可能。身為思考活潑做事嚴謹的程式設計師，我們必須徹底地防止此事發生才行。

在這兒只要使用任何一種可跨行程的執行緒同步物件，即可達成變數保護效果，我選用的是 event 核心物件。首先，在呼叫 *SetWindowsHookEx* 函式的前後加上保護機制：在呼叫前建立保護 hook handle 的 event，並且在呼叫後將 event 設為 signaled 狀態，若 hook 函式此時正在等待，就等於通知 hook 函式內的 event，hook handle 已經可用了。

```
#0001 // 建立保護 hook handle 的 event
#0002 hEvent = CreateEvent(NULL, true, false, HOOK_EVENT_NAME);
#0003 // 安裝 WH_GETMESSAGE hook
#0004 g_Data->HHOOK = SetWindowsHookEx(WH_GETMESSAGE,
#0005 (HOOKPROC)GetMsgProc, g_hinstDLL, ThreadId);
#0006 // 將 event 設為 signaled 狀態，若 hook 函式正在等待，等於通知它，
#0007 // hook handle 已經 ready
#0008 SetEvent(hEvent);
#0009 // 關閉 event
#0010 CloseHandle(hEvent);
```

同時，在 hook 函式第一次被叫用時，嘗試開啓 event，若開啓成功，表示 hook handle 尚未被寫入，此時只好呼叫 *WaitForSingleObject* 函式等待 hook handle 確定下來；如果開啓失敗，表示 hook handle 老早就準備好了，沒有問題，可以直接使用。

```
#0001 // 嘗試開啓保護 hook handle 的 event
#0002 hEvent = OpenEvent(SYNCHRONIZE, false, HOOK_EVENT_NAME);
#0003 if (hEvent) { // 如果開啓成功，表示 hook handle 尚未被寫入
#0004 // 等待 hook handle 確定下來
#0005 WaitForSingleObject(hEvent, INFINITE);
#0006 CloseHandle(hEvent); // 關閉 event
#0007 }
#0008
#0009 // 呼叫下一個 hook 函式
#0010 return CallNextHookEx(g_Data->HHOOK, Code, WPARAM, LPARAM);
```

哇啊，囉哩囉嗦的好多篇幅，總算將程式所用到的技巧及技術解說完畢。沒關係，苦盡才有甘來，接下來就輕鬆多了，進入真正的程式撰寫階段。

程式撰寫

經過上面的詳細討論，將底層的技术細節及實作方法完全揭露後，所有的技巧加起來只有兩句話：

- 為跨行程的 subclassing 動作做準備
- Subclass 背景視窗及其父視窗

所有的技術及知識皆已備足後，程式撰寫就成了驗證想法及享受成就感的休閒活動。

你相信嗎？雖然前述的程式手法及使用技術讀起來十分繁雜、驚險萬分，但是程式寫出來後，計算載入程式及 Hook DLL 的原始碼加總，結果不到一千行，只有八百行上下呢。下面的程式實作部分，我將不會列出程式碼，純粹以程式流程及函式骨架的大局觀來說明。

載入程式

載入程式的任務為：

- 啟動時，呼叫 *SetDeskHook* 函式，傳入 *true* 來進行 hook 及 subclassing 動作。並將短文內容、顯示參數及桌布資訊寫入共享資料結構供背景視窗及其父視窗使用。
- 每當使用者經由功能表更動任何參數或短文內容時，將新的資訊寫入共享資料結構，並重繪桌面。
- 按下關閉鈕或由系統選單選擇關閉時，只將視窗隱藏起來。
- 將短文內容、顯示參數及桌布資訊存入檔案及系統登錄，啟動時載入使用。
- 結束前，呼叫 *SetDeskHook* 函式，傳入 *false* 來取消所有 subclassing 及 hook 動作。

DIHOOK DLL

DIHOOK DLL 的任務為：

- 在載入點及釋放點建立及釋放記憶體映射區域。
- 提供 *SetDeskHook* 及 *hook* 函式供載入程式叫用，安裝或取消 *hook*。
- *Hook* 函式第一次被呼叫時，進行背景視窗及背景視窗父視窗的 subclassing 動作。
- 根據「程式手法」一節，分別在兩個視窗的視窗函式內進行訊息攔截、繪製短文及轉交滑鼠訊息等動作。
- 利用計時器函式，每秒鐘更新視窗右下角的時間。
- 若 *ConfigButton* 按鈕被按下，叫出載入程式視窗供使用者設定。

背景視窗的視窗函式

在背景視窗的視窗函式中，攔截這些視窗訊息：

- *WM_MOUSEFIRST ~ WM_MOUSELAST*
攔截所有的滑鼠相關訊息，將必要的滑鼠訊息轉交給 *ConfigButton* 按鈕。
- *WM_SETTINGCHANGE*、*WM_SYSCOLORCHANGE*
重新取得桌面底色、桌布圖形、工作區域，並且重新計算時間字串及設定按鈕的顯示座標。
- *WM_TIMER*
每秒鐘收到一次，更新畫面右下角的時間。
- *WM_SUBCLASSED*
Hook 函式 subclass 完成後會送出此自訂訊息，供視窗函式進行初始化。
- *WM_BEFORE_UNSUBCLASS*
Hook 函式 unsubclass 之前會送出此自訂訊息，供視窗函式進行善後工作。
- *WM_ERASEBKGD*
根據使用者設定，選擇是否將桌布隱藏起來。同時依選擇的字型及顏色在畫面上繪

製短文。

背景視窗父視窗的視窗函式

在背景視窗父視窗的視窗函式中，只攔截一道視窗訊息：

□ `WM_NOTIFY`

若動作代碼為 `NM_CUSTOMDRAW` (`TNMHDR::code` 欄位)，則在畫面右下角繪出時間字串，並傳回 `CDRF_DODEFAULT` 使背景視窗繼續進行正常的繪製動作。

成果品嘗

下面兩張程式執行畫面，分別以「不顯示桌布」及「顯示桌布」狀態來進行。在 `Desktop Illusionist` 主視窗中，你隨時可以修改短文內容，按下熱鍵【`CTRL - W`】來更新畫面。短文的字型、顏色、分隔線顏色都可以自由設定。如果主視窗消失了，別緊張，請按一下在畫面右下角的「設定」鈕，就可將 `Desktop Illusionist` 呼喚回來。

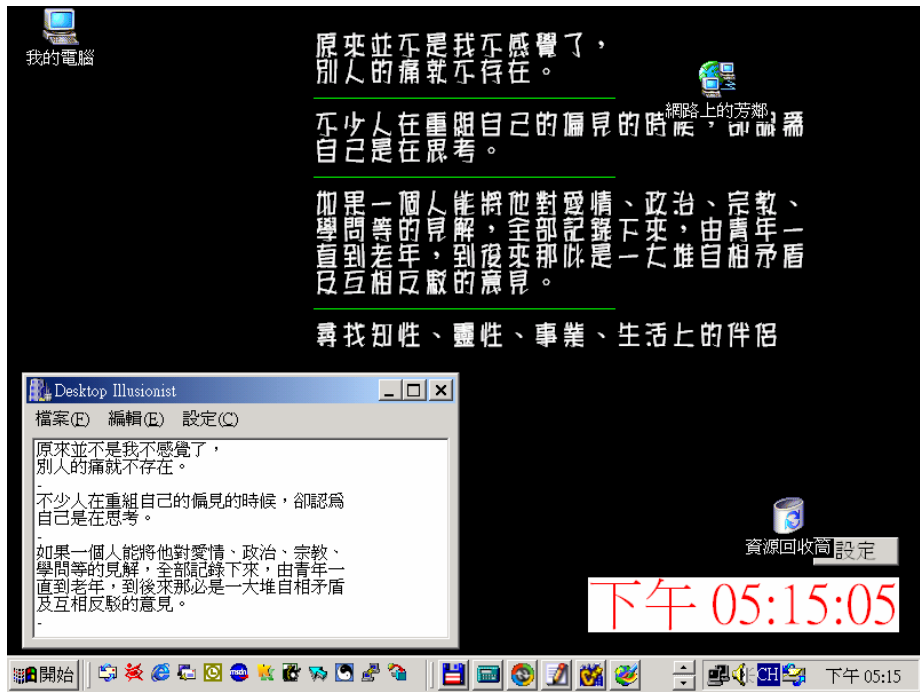


圖 5-19 / Desktop Illusionist 執行畫面，不顯示桌布



圖 5-20 / Desktop Illusionist 執行畫面，顯示桌布

看看書上的圖片，若你看不出來 Desktop Illusionist 的特別之處，請拿出書附光碟，親自在你的電腦上執行看看，相信你馬上就可以感受它的與眾不同。

現在，你是不是也覺得，技術能力夠，創意足，再加上膽大心細的實作功夫，什麼東西都變得十分好玩呢？！:)

第六章

佈景主題工具實戰

對於愛把桌面弄得漂漂卻又美術細胞全無的我來說，

佈景主題這設計可說是一大福音。

嫌 Microsoft Plus! 佈景安裝工具不夠力說...

還不簡單，寫一套給你看！



同樣的桌面，同樣的背景顏色、桌布，同樣的字型、系統顏色、滑鼠指標...不論畫面如何華麗，看久了總覺得乏味，更何況，我們得天天盯著看好多個小時呢！

在「控制台」的「顯示器」元件中，可以更動桌布、系統顏色、字型、螢幕保護程式等等桌面與視窗外觀，「滑鼠」元件可讓我們設定滑鼠指標，「聲音」元件可供設定各種事件的音效，不過只能一項一項慢慢地更動（如圖 6-1）。如果通通都想更改，那可真是一項費力耗時的大工程，顏色、字型、滑鼠指標、音效、圖示等等一大堆設定，要完全手動更換不中途而廢者，幾希！



圖 6-1 / 分別透過「顯示器」、「滑鼠」、「聲音」控制台元件來更動設定

幸好，微軟為懶人們提出「佈景主題」的操作概念。「佈景主題」是一整套包含聲音、圖示、滑鼠指標、桌布、顏色、字型、螢幕保護及視窗顯示參數等等設定值及相關檔案的套件。事實上，它包含的全部設定，幾乎也都可以透過控制台手動修改，而佈景主題只是將這些設定值記錄下來，將所有相關檔案集中為套件，並使設定工作自動化，如此而已。

如今，網際網路上可以找到數以千計的佈景主題供人下載，在搜尋引擎中打入「佈景主題」或「desktop theme; download」等關鍵字，輕輕鬆鬆即可找到一大堆專門供應佈景主題下載的網站。國內的佈景主題網站還成立了「桌面聯盟」，讓使用者永遠有下載不完

的佈景主題可賞玩。以國內知名的佈景主題下載網站「桌面王」¹（圖 6-2）來說，每日竟然可有數十萬瀏覽人／次，可見佈景主題已成為電腦玩家們的新寵。

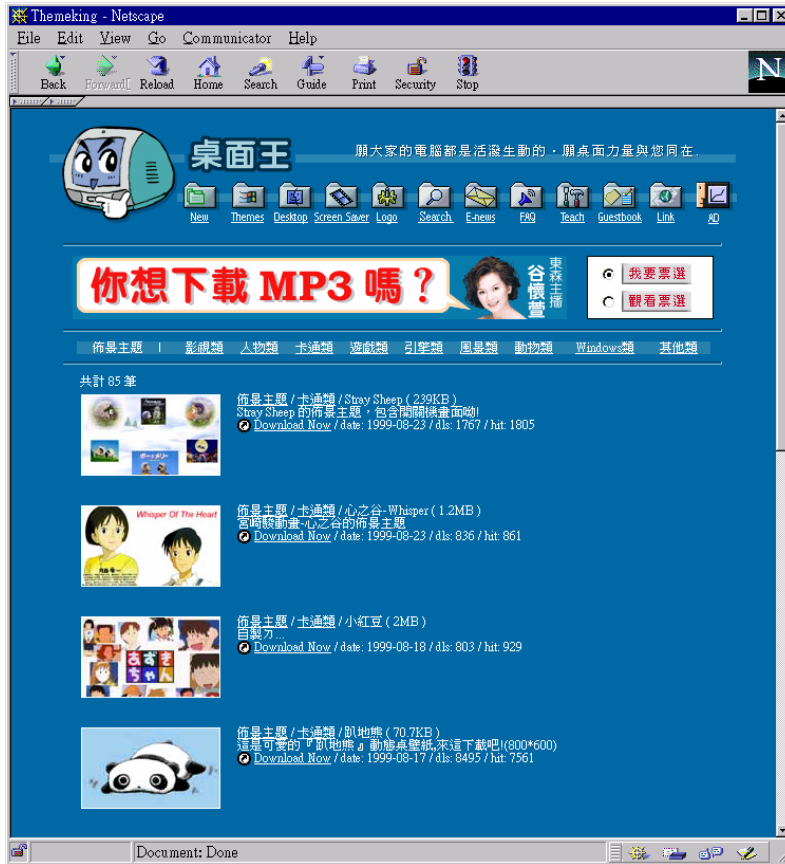


圖 6-2 / 國內知名的佈景主題下載網站「桌面王」

早期，我們會安裝 Microsoft Plus! 佈景主題工具來安裝這些佈景主題。與現今眾多的強力佈景主題管理工具相較，Microsoft Plus! 佈景主題工具的功能顯得相當薄弱，陽春多了。它只能夠有限地預視佈景元件，選擇欲安裝的佈景元件種類，按下「套用」或「確定」按鈕來安裝佈景主題。

¹ 「桌面王」網站的URL為<http://www.themeking.com/>。

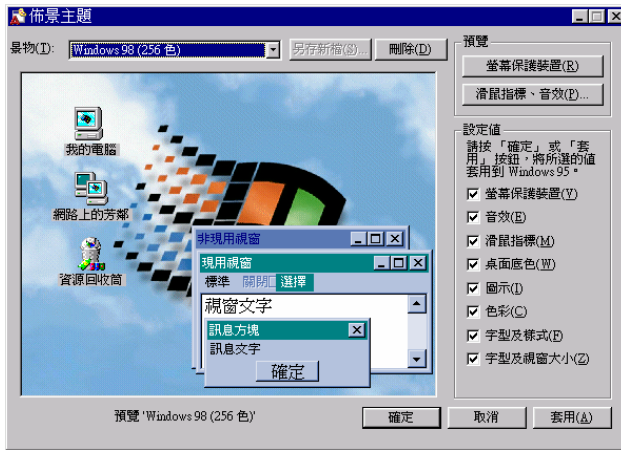


圖 6-3 / Microsoft Plus! 佈景主題工具

後來，Left Side Software公司開發了一套Desktop Themes工具²。你可由執行畫面看出，它沿襲Microsoft Plus! 佈景主題工具的介面，最大的改進為支援佈景主題的編輯功能。編輯功能讓玩家們可從使用者搖身一變而成佈景主題設計大師，自行設計、繪製、調配賞心悅目的佈景主題，還可上傳到佈景主題網站與全世界的網友分享。



圖 6-4 / Desktop Themes 執行畫面

² Desktop Themes的URL為<http://www.lss.com.au/lss/windows/lsswindows.htm>。

特別提出 Desktop Themes 介紹的原因是，由於出現時間極早的緣故，雖然提供的功能及介面並不特別，但它一直是佔有率最高的佈景主題管理工具。由此更可以看出，市場進入時間對於產品銷售的重要性。

對於桌面設定及佈景主題十分講究的我來說，Microsoft Plus! 佈景工具及 Desktop Themes 的功能實在不敷使用。於是，本著程式設計師雙手萬能的志氣，1997 年暑假，燠熱的炎夏中，打著赤膊邊吃冰棒邊敲鍵盤的我，終於撰寫出心目中理想的佈景主題管理工具－XTheme Manager。

XTheme Manager 簡介

XTheme Manager 是 Microsoft Plus! 佈景主題工具的加強版。除了擁有它所有的功能外，還新增許多強大的新功能，接下來為你一一介紹：

- 完美的佈景預視。包括背景、圖示、顏色及視窗設定，通通看得見，如下圖。

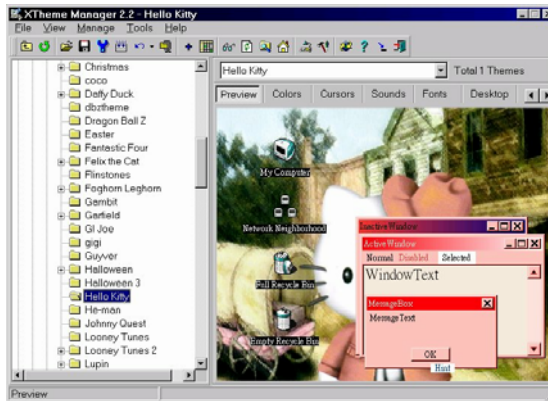


圖 6-5 / 完美的佈景預視，背景、圖示、顏色及視窗設定，通通看得見

- 每種元件通通可以預視，顏色、圖示、指標、聲音、字型、桌面、桌布、螢幕保護程式，甚至連開關機 Logo 都行。換句話說，在安裝佈景主題之前，你可以藉由 XTheme Manager 事先得知所有安裝後的改變（圖 6-6 ~ 圖 6-9）。
- 支援 Windows 98，包括視窗標題漸層顏色，及 Web View 圖示，如圖 6-10。
- 完全自訂的安裝程序。你可以分別選擇單一的顏色、指標、聲音、字型、圖示、Logo 來安裝。例如，你只想要維持「我的電腦」這個圖示及背景顏色，其它的都想換掉，XTheme Manager 的自訂安裝可以讓你達成願望。
- 佈景回復功能，可以記錄之前最多 100 次的佈景安裝動作。
- 可在任意目錄下預視／編輯／安裝佈景主題，不必和 Microsoft Plus! 佈景主題工具一樣，必須放在固定目錄下才可使用。採用這種獨立目錄方式，就不必再把數百個檔案通通塞到同一個佈景目錄了。

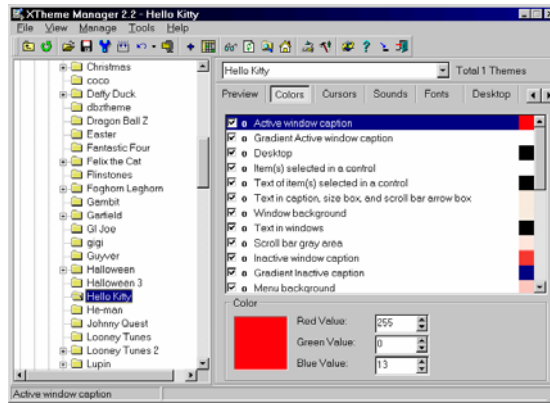


圖 6-6 / 系統顏色設定畫面

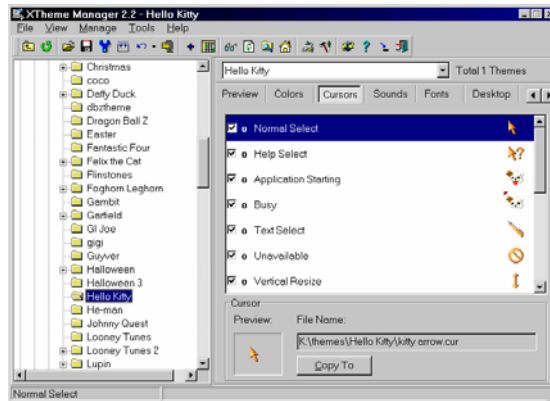


圖 6-7 / 系統指標設定畫面

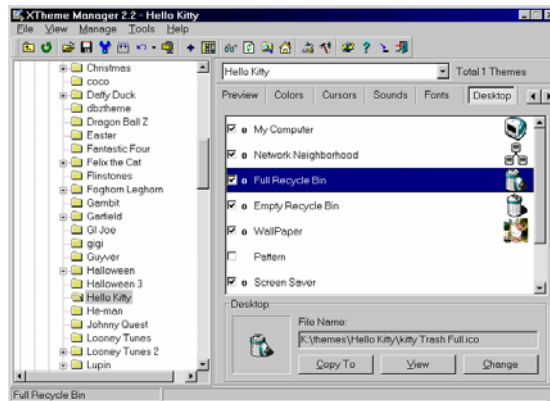


圖 6-8 / 桌面設定畫面，包括圖示、桌布、螢幕保護及視窗設定

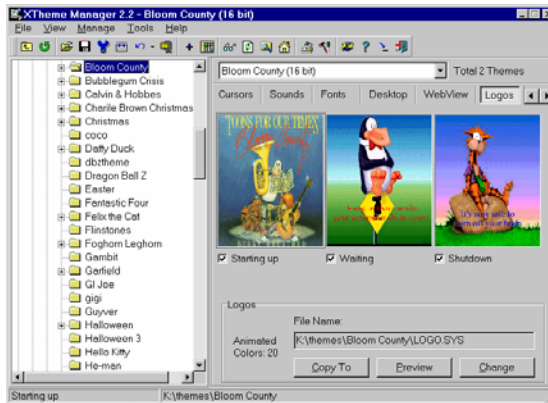


圖 6-9 / 開關機畫面設定

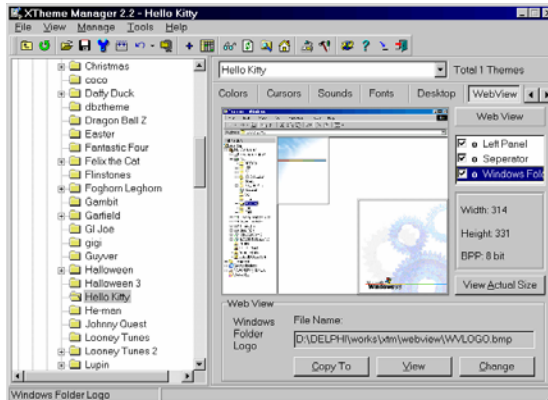


圖 6-10 / Windows 98 檔案總管的 Web 檢視設定

- 重建 icon cache 及 font folders 功能。
- 自動搜尋開關機畫面及自訂字型。Microsoft Plus! 佈景主題工具並未支援 Windows 95/98 的開關機畫面，所以 XTheme Manager 在佈景描述檔案 (.THEME) 中新增三筆項目：

```
[Logos]
Startup=Windows\logo.sys
Wait=Windows\logow.sys
Shutdown=Windows\logos.sys
```

如果你的佈景檔案沒有指定開關機畫面檔案位置，XTheme Manager 會自動幫你找出來；自訂字型 (.FON、.FNT、.TTF) 也是如此。

- 佈景移交精靈，讓你可以輕易地將自己製作的佈景及檔案打包起來，壓成ZIP檔，可直接製作自解壓縮檔或一般ZIP檔³。



圖 6-11 / 佈景移交精靈的第一個畫面，共有四個畫面

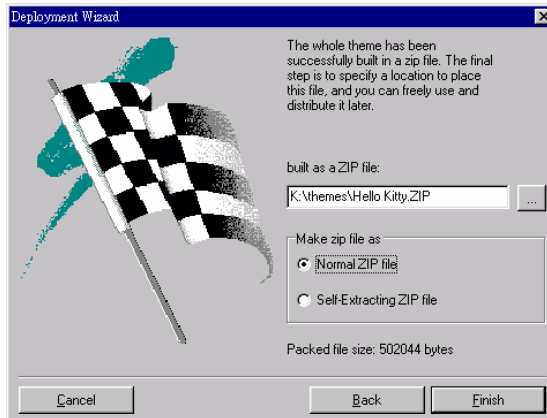


圖 6-12 / 佈景移交精靈的完成畫面，可選擇正常壓縮檔或自解壓縮檔

- 自動更換佈景功能。就像其它的桌布更換軟體（如XDesktop⁴:P）一樣，XTheme Manager也提供了自動更換佈景的功能。請先喜愛的佈景加入「更換列表」，設定好間隔時間及佈景輪替方式後，每隔一段時間，它就會幫你自動更換佈景。

³ 這是桌面王站長Dicky Ho的建議，再次感謝。

⁴ XDesktop是我撰寫的第一套共享軟體，欲知詳情，請來我的網站玩玩。

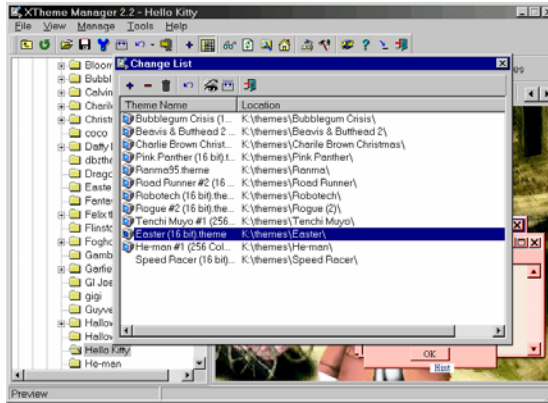


圖 6-13 / 自動更換佈景列表視窗

- 移除佈景主題檔案及元件功能，節省你刪除佈景主題時一一找尋檔案的時間。

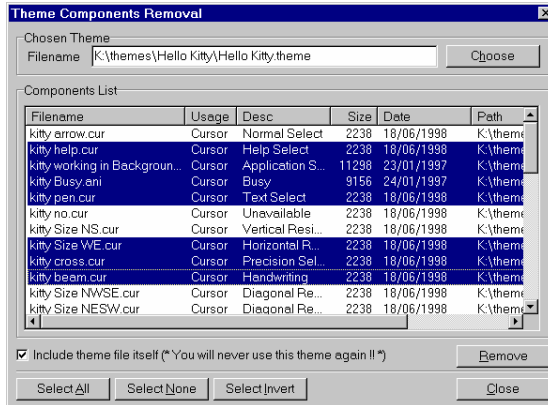


圖 6-14 / 移除佈景主題檔案及元件功能

- 可從 DLL、ICL 檔或其它可能包含圖示資源的檔案選擇桌面圖示。

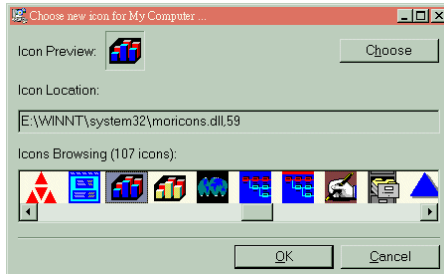


圖 6-15 / 從 DLL、ICL 檔或其它圖示資源檔案內選擇圖示

- 內建近一百個佈景網站資料，包含網站描述、聯絡資訊及評分等等。

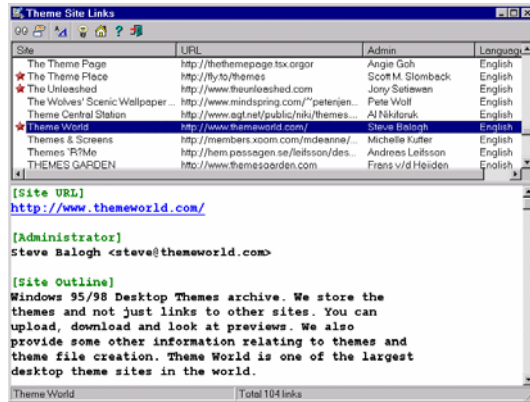


圖 6-16 / 內建近一百個佈景網站資料

或許是運氣好，加上支援的功能的確不少，XTheme Manager 甫推出不久即獲得不少共享軟體網站評鑑的獎項：))



圖 6-17 / XTheme Manager 所獲得的獎項

好了，阿達賣瓜也賣得差不多了。接下來，我將秉著撰寫 XTheme Manager 的經驗，帶領你從無到有，從分析到實作，學習桌面設定的操控及程式撰寫，並發展一套完整的佈景主題預覽／安裝工具。

認識佈景主題

佈景主題包括佈景描述檔案（.THEME）及個別佈景元件的圖示、指標、字型、影像檔案等等。而佈景主題安裝工具的任務便是，分析佈景描述檔案，一一將對應的佈景元件安裝到系統中。

佈景元件

一套完整的佈景主題可能包括下列元件：

表 6-18 / 佈景主題元件及所需檔案類型

佈景元件種類	所需檔案類型
佈景顏色	不需要
指標	.CUR、.ANI
聲音	.WAV
字型	.FON、.FNT、.TTF
圖示	.ICO
桌布	.BMP、.JPG、.GIF 等任何影像檔
螢幕保護程式	.SCR
Web View	.BMP、.GIF
開關機畫面	.BMP、.JPG、.GIF 等任何影像檔

所以，一套佈景主題通常是由一個佈景描述檔案（.THEME），以及在同一個目錄下的數十個佈景元件檔案配合而成。

佈景描述檔案

今年夏天最流行什麼？沒錯，就是Kitty貓⁵。

Talk

1999年八月，麥當勞推出 Kitty 貓贈品活動。每在新樣式推出當天，全省各地的麥當勞門口就出現大排長龍的人群，許多人從凌晨就開始排隊，還有人為搶奪 Kitty 貓而大打出手，真可謂世紀末之吃飽太閒怪現象。

噢，聽說同樣是日本貨的皮卡丘成功攻下美國市場，這麼卡哇依的口袋怪物我看又會在全世界造成一股流行風潮。

我們就以可愛的Hello Kitty⁶ 佈景為研究對象，瞧瞧它的佈景描述檔案－Hello Kitty.theme 長的什麼模樣。

HELLO KITTY.THEME

□ 檔頭

```
#0001 ; Contact http://www.vclxx.org/xtm/xtheme\_manager.htm
#0002 ; This theme was created / edited by XTheme Manager version 2.2
```

由於採 INI 檔案格式，所以同樣地，註解以分號開頭。

⁵ 哦，第二波流行的主角是無尾熊。兩隻遠從澳洲來做客的無尾熊，正好在七夕情人節前開放參觀，讓木柵動物園連續幾個星期，天天都是數萬人潮。

⁶ 可能是Kitty貓感謝我在書中幫她造勢哦！今天晚上我順利地以十塊錢在路口雜貨店前的夾娃娃機中抓出一隻可愛的Kitty貓，大大的圓圓臉，紅色的吊帶褲，頭上的吊線還有一個吸盤，可以掛在車窗內那種，太棒了！:)

□ 桌面圖示

```
#0004 ; 我的電腦圖示
#0005 [CLSID\{20D04FE0-3AEA-1069-A2D8-08002B30309D}\DefaultIcon]
#0006 DefaultValue=%ThemeDir%Kitty My Computer.ico,0
#0007
#0008 ; 網路上的芳鄰圖示
#0009 [CLSID\{208D2C60-3AEA-1069-A2D7-08002B30309D}\DefaultIcon]
#0010 DefaultValue=%ThemeDir%Kitty Network Group.ico,0
#0011
#0012 ; 資源回收筒圖示
#0013 [CLSID\{645FF040-5081-101B-9F08-00AA002F954E}\DefaultIcon]
#0014 full=%ThemeDir%Kitty Trash Full.ico,0
#0015 empty=%ThemeDir%Kitty Trash Empty.ico,0
```

特殊桌面圖示的圖示檔案。Windows 98 內附的 Microsoft Plus! 佈景工具新增「我的文件夾」圖示，其區段名稱為：

```
[CLSID\{450D8FBA-AD25-11D0-98A8-0800361B1103}\DefaultIcon]
```

□ 系統顏色

```
#0017 ; 系統顏色
#0018 [Control Panel\Colors]
#0019 ActiveTitle=255 0 13
#0020 Background=0 0 0
#0021 Hilight=255 255 255
#0022 HilightText=0 0 0
#0023 TitleText=240 232 216
#0024 Window=240 232 216
#0025 WindowText=0 0 0
#0026 Scrollbar=253 224 223
#0027 InactiveTitle=242 56 51
#0028 Menu=251 192 191
#0029 WindowFrame=0 0 0
#0030 MenuText=0 0 0
#0031 ActiveBorder=251 192 191
#0032 InactiveBorder=251 192 191
#0033 AppWorkspace=242 56 51
#0034 ButtonFace=251 192 191
#0035 ButtonShadow=242 56 51
#0036 GrayText=242 56 51
#0037 ButtonText=0 0 0
#0038 InactiveTitleText=0 0 0
#0039 ButtonHilight=253 224 223
```



```
#0040 ButtonDkShadow=0 0 0
#0041 ButtonLight=251 192 191
#0042 InfoText=0 64 128
#0043 InfoWindow=255 255 255
#0044 GradientActiveTitle=255 255 255
#0045 GradientInactiveTitle=0 0 128
```

各個系統顏色的紅綠藍三原色色量字串。0044 ~ 0045 列所指定的 *GradientActiveTitle* 及 *GradientInactiveTitle* 兩種系統顏色為視窗標題漸層色，只有 Windows 98 之後的版本才支援。

□ 滑鼠指標

```
#0047 ; 滑鼠指標
#0048 [Control Panel\Cursors]
#0049 Arrow=%ThemeDir%Kitty arrow.cur
#0050 Help=%ThemeDir%Kitty help.cur
#0051 AppStarting=%ThemeDir%Kitty working in Background.ani
#0052 Wait=%ThemeDir%Kitty Busy.ani
#0053 NWPen=%ThemeDir%Kitty pen.cur
#0054 No=%ThemeDir% Kitty no.cur
#0055 SizeNS=%ThemeDir%Kitty Size NS.cur
#0056 SizeWE=%ThemeDir%Kitty Size WE.cur
#0057 Crosshair=%ThemeDir%Kitty cross.cur
#0058 IBeam=%ThemeDir%Kitty beam.cur
#0059 SizeNWSE=%ThemeDir%Kitty Size NWSE.cur
#0060 SizeNESW=%ThemeDir%Kitty Size NESW.cur
#0061 SizeAll=%ThemeDir%Kitty move.cur
#0062 UpArrow=%ThemeDir%Kitty up.cur
#0063 DefaultValue=Windows default
```

系統滑鼠指標，可以設定為 .CUR 一般指標檔案或 .ANI 動態指標檔案。

□ 桌布及填圖樣式

```
#0066 ; 桌布及填圖樣式
#0067 [Control Panel\Desktop]
#0068 Wallpaper=%ThemeDir%Kitty.jpg
#0069 TileWallpaper=0
#0070 WallpaperStyle=0
#0071 Pattern=(None)
```

桌布檔案、桌布樣式及背景填圖樣式，桌布檔案必須是佈景安裝工具支援的格式才行。

□ 系統音效

```
#0073 ; 各種事件的音效
#0074 [AppEvents\Schemes\Apps\.Default\.Default\.Current]
#0075 DefaultValue=%ThemeDir%Kitty default sound.wav
#0076
#0077 [AppEvents\Schemes\Apps\.Default\AppGPFault\.Current]
#0078 DefaultValue=%ThemeDir%Kitty program error.wav
#0079
#0080 [AppEvents\Schemes\Apps\.Default\Maximize\.Current]
#0081 DefaultValue=%ThemeDir%Kitty maximize.wav
#0082
#0083 [AppEvents\Schemes\Apps\.Default\MenuCommand\.Current]
#0084 DefaultValue=%ThemeDir%Kitty menu command.wav
#0085
#0086 [AppEvents\Schemes\Apps\.Default\MenuPopup\.Current]
#0087 DefaultValue=%ThemeDir%Kitty menu popup.wav
#0088
#0089 [AppEvents\Schemes\Apps\.Default\Minimize\.Current]
#0090 DefaultValue=%ThemeDir%Kitty minimize.wav
#0091
#0092 [AppEvents\Schemes\Apps\.Default\Open\.Current]
#0093 DefaultValue=
#0094
#0095 [AppEvents\Schemes\Apps\.Default\Close\.Current]
#0096 DefaultValue=
#0097
#0098 [AppEvents\Schemes\Apps\.Default\RestoreDown\.Current]
#0099 DefaultValue=%ThemeDir%Kitty restore down.wav
#0100
#0101 [AppEvents\Schemes\Apps\.Default\RestoreUp\.Current]
#0102 DefaultValue=%ThemeDir%Kitty restore up.wav
#0103
#0104 [AppEvents\Schemes\Apps\.Default\RingIn\.Current]
#0105 DefaultValue=%ThemeDir%Kitty Ring.wav
#0106
#0107 [AppEvents\Schemes\Apps\.Default\Ringout\.Current]
#0108 DefaultValue=
#0109
#0110 [AppEvents\Schemes\Apps\.Default\SystemAsterisk\.Current]
#0111 DefaultValue=%ThemeDir%Kitty asterisk.wav
#0112
#0113 [AppEvents\Schemes\Apps\.Default\SystemDefault\.Current]
#0114 DefaultValue=
#0115
#0116 [AppEvents\Schemes\Apps\.Default\SystemExclamation\.Current]
```

```

#0117 DefaultValue=%ThemeDir%Kitty exclamation.wav
#0118
#0119 [AppEvents\Schemes\Apps\.Default\SystemExit\.Current]
#0120 DefaultValue=%ThemeDir%Kitty Windows Exit.wav
#0121
#0122 [AppEvents\Schemes\Apps\.Default\SystemHand\.Current]
#0123 DefaultValue=%ThemeDir%kitty critical stop.wav
#0124
#0125 [AppEvents\Schemes\Apps\.Default\SystemQuestion\.Current]
#0126 DefaultValue=%ThemeDir%Kitty question.wav
#0127
#0128 [AppEvents\Schemes\Apps\.Default\SystemStart\.Current]
#0129 DefaultValue=%ThemeDir%Kitty Windows Start.wav
#0130
#0131 [AppEvents\Schemes\Apps\Explorer\EmptyRecycleBin\.Current]
#0132 DefaultValue=%ThemeDir%Kitty empty recycle bin.wav

```

□ 圖示及視窗相關參數

```

#0135 ; 圖示相關參數及視窗相關參數
#0136 [Metrics]
#0137 IconMetrics=76 0 0 0 77 0 0 0 75 0 0 0 1 0 0 0 244 255 255 255 0 0
#0138 0 0 0 0 0 0 0 0 0 144 1 0 0 0 0 0 136 0 0 0 2 183 115 178 211 169
#0139 250 197 233 0 0 119 97 118 0 117 110 100 46 119 97 118 0 0 0 0 150
#0140 0 0 19 11 0 0

```

TIconMetrics 結構的位元組傾印，記錄圖示相關參數，包括圖示間距、圖示字形等等。

```

#0142 NonClientMetrics=84 1 0 0 1 0 0 0 18 0 0 0 18 0 0 0 18 0 0 0 18 0
#0143 0 0 244 255 255 255 0 0 0 0 0 0 0 0 0 0 144 1 0 0 0 0 136 0
#0144 0 0 2 183 115 178 211 169 250 197 233 0 92 116 101 109 112 46 84 104
#0145 101 109 101 0 221 64 0 1 0 0 0 252 252 4 193 13 0 0 0 14 0 0 0 244
#0146 255 255 255 0 0 0 0 0 0 0 0 0 0 188 2 0 0 0 0 0 136 0 0 0 2 183
#0147 115 178 211 169 250 197 233 0 92 116 101 109 112 46 84 104 101 109
#0148 101 0 221 64 0 1 0 0 0 252 252 4 193 18 0 0 0 18 0 0 0 244 255 255
#0149 255 0 0 0 0 0 0 0 0 0 144 1 0 0 0 0 0 136 0 0 0 2 183 115 178
#0150 211 169 250 197 233 0 92 116 101 109 112 46 84 104 101 109 101 0 221
#0151 64 0 1 0 0 0 252 252 4 193 244 255 255 255 0 0 0 0 0 0 0 0 0 0
#0152 144 1 0 0 0 0 0 136 0 0 0 2 183 115 178 211 169 250 197 233 0 92 116
#0153 101 109 112 46 84 104 101 109 101 0 221 64 0 1 0 0 0 252 252 4 193
#0154 244 255 255 255 0 0 0 0 0 0 0 0 0 0 144 1 0 0 0 0 0 136 0 0 0
#0155 2 183 115 178 211 169 250 197 233 0 92 116 101 109 112 46 84 104 101
#0156 109 101 0 221 64 0 1 0 0 0 252 252 4 193

```

TNonClientMetrics 結構的位元組傾印，記錄視窗相關參數，包括各種系統字型、邊框寬度、選項高度、捲軸寬度。

□ 螢幕保護程式

```
#0158 ; 螢幕保護程式
#0159 [boot]
#0160 SCRNSAVE.EXE=%ThemeDir%Kitty\Kitty.SCR
```

□ 佈景工具專屬設定

```
#0162 ; MS Microsoft Plus! 的佈景安裝工具專用
#0163 [MasterThemeSelector]
#0164 MTSM=DABJDKT
#0165 Stretch=0
#0166
#0167 ; XTheme Manager 的 Web View 支援
#0168 [WebView]
#0169 WVLEFT.BMP=c:\C++BUILDER\works\xtm\webview\wvleft.bmp
#0170 WVLINE.GIF=c:\C++BUILDER\works\xtm\webview\wvline.bmp
#0171 WVLOGO.GIF=c:\C++BUILDER\works\xtm\webview\WVLOGO.bmp
```

Windows 98 檔案總管 Web 畫面所用的圖形檔，請見圖 6-10。

```
#0173 ; XTheme Manager 的開關機畫面支援
#0174 [Logos]
#0175 Startup=
#0176 Wait=
#0177 Shutdown=
```

Windows 95/98 的開機／關機中／關機畫面，必須是 320 x 240 大小，256 色的影像檔。

```
#0179 ; XTheme Manager 專用
#0180 [XTheme Manager]
#0181 IntallColors=11111111111111111111111111111111
#0182 IntallCursors=1111111111111111
#0183 IntallSounds=1111111111111111111111111111
#0184 IntallDesktop=11111011
#0185 IntallFonts=111111
#0186 IntallLogos=000
```

呼，整整近兩百行的佈景描述檔案，看下來十分累人！

明眼人一看，就知道它採用的是 Windows 的 INI 檔案格式，也就是以中括號 [Section] 定義區段名稱，以等號區分屬性名稱及屬性值 (Property = Value) 的格式。

XTheme Manager Lite

發展 XTheme Manager 時，我從對佈景主題的一無所知，研究佈景主題描述檔案時的一知半解，撰寫各項佈景元件預視／操作功能時的一籌莫展，解決各平臺差異問題時的一波三折，最後終於一股作氣地將它完成，進而包裝為共享軟體，分享給全世界的佈景迷們。

本章中，讓我們以 XTheme Manager 為參考模型，抽出佈景主題工具必備的功能，另外撰寫一套簡易版的佈景主題管理工具，我稱它為 XTheme Manager Lite。

有沒有發現，XTheme Manager Lite 這個名稱的頭字語寫起來很容易讓人誤解。X...T...M...L，像是 HTML（HyperText Markup Language）及 XML（eXtensible Markup Language）的混合體，一定會有人以為這又是哪個即將取代 HTML 和 XML 成為 WWW 文件新標準的玩意兒，哈哈，有時學學英語系國家玩玩頭字語真的挺有趣的。

功能設定

前頭看過 XTheme Manager 的功能簡介後，也許你會好奇，究竟需要多少程式碼，才能完成這套功能強大的玩具軟體⁷呢？答案是約一萬餘行 Object Pascal 程式碼，這還不包括程式裏使用的自製元件及程式庫的原始碼。當然囉，程式碼行數不是評估軟體功能及困難度的絕對指標，用兩三千行程式碼寫出九九乘法表也不是不可能的事，這兒只是給你一個概念－即使是玩具型軟體，也不是輕輕鬆鬆就可以生出來的。

唔，被我這麼一嚇，看來佈景主題工具似乎不是很好寫，不是短短幾十行就能解決的小玩意。那麼，對於現在要進行的佈景工具 XTML，功能上就得好好瘦身一番了。我只選擇最重要的幾個功能來支援：

⁷ 這類吃飽太閒沒事才拿來玩玩的桌面軟體，不算玩具軟體算什麼？:P

- 預視現行佈景設定
程式開啓後，載入佈景主題檔案前，能夠讀取並預視目前的佈景設定值。
- 預視任何目錄下的佈景主題
必須能夠載入任意目錄下的佈景主題檔案，並載入對應的佈景元件以供預視及檢閱。
- 安裝佈景主題
將佈景主題套用到系統上，立即更新所有對應的佈景設定，並將新設定寫入登錄資料庫。

其它尚有幾個有用的功能，例如選擇性安裝、佈景編輯及儲存、自動更換等等功能都不打算加入 X`HTML` 內，因為這些功能幾乎都大量涉及使用者介面的操作及互動，勢必又得增加不少與佈景主題本身無關的程式碼，於是只好割愛。

介面設計

仿照 X`Theme Manager` 的方式，X`HTML` 以分頁方式來呈現各佈景元件的資訊。

主視窗上放置一個大小與視窗相同的 `TPageControl` 元件，並建立六個頁次，分別是「預視」、「顏色」、「指標」、「聲音」、「字型」、「桌面」；除了「預視」頁面，每個頁面各自負責不同種類佈景元件的預視資訊。它們的設計時期畫面如下：

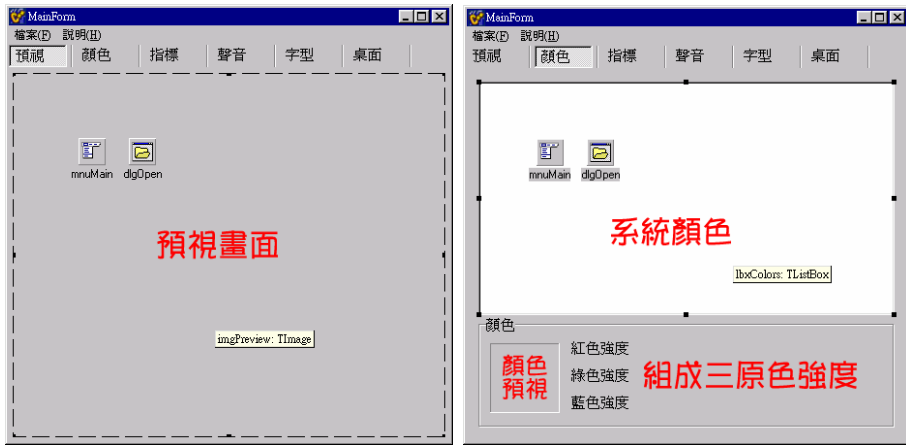


圖 6-19 / 設計時期的「預視」、「顏色」頁面



圖 6-20 / 設計時期的「指標」、「聲音」頁面

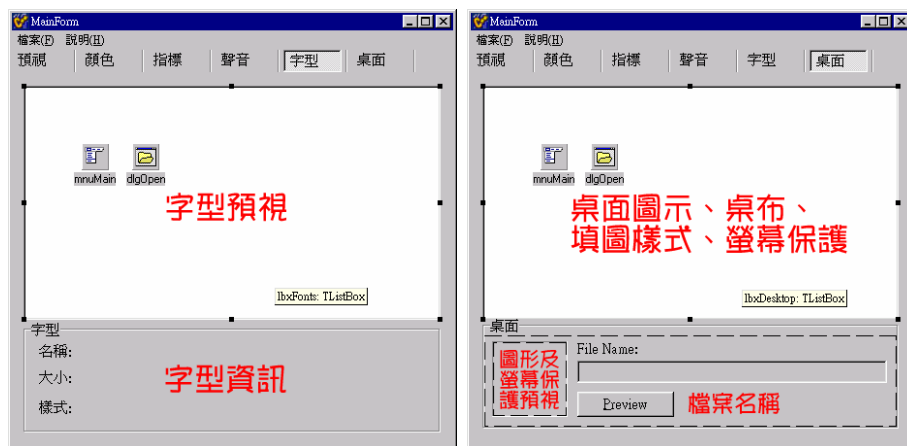


圖 6-21 / 設計時期的「字型」、「桌面」頁面

介面的設計相當直覺，將所有佈景元件分為五個種類，分別在五個頁面透過 *TListBox* 元件來列表及檢視，並在第一個頁面提供佈景預覽，使用的是 *TImage* 元件。

接下來，我們來看看，Windows 95 的使用者介面中，有哪些東東是可以被更動的。又，如何以程式來操控這些設定。我將所有的使用者介面元件分為「系統顏色」、「滑鼠指標」、「系統音效」、「字型及其它設定」、「桌面圖示」、「桌布及樣式」六大項來說明。

系統顏色

一種米養百種人，隨便抓兩個倒楣的路人來拷問，關於他們的小動作、日常用語、生活習慣、地域觀念等等，應該都存在著相當大的差異。即使是雙胞胎兄弟，也十分可能一個愛上清瘦高挑的長腿姑娘，另一個卻偏好肉肉胖美眉。審美觀如此，關於顏色的感受亦是如此，尤其這些主觀的感覺通常還會受到心情、狀態、環境等等的影響。

因此，程式員在撰寫程式時，除非背後有超強美工幫你撐腰，否則設計介面時請循規蹈矩，儘量不要搞怪，否則很容易引起使用者反感。每回看到一些花花綠綠，紅按鈕藍背

景外加紫色邊框，配色十分怪異的應用程式，既談不上新潮，也稱不順眼，大概就可猜到這一定是初入 Windows 程式設計領域的程式員所寫的。

我的建議是，身為 Windows 程式設計師，於使用者介面設計風格的十字路口，只有三條路可走：

□ 標準的 Windows 風格應用程式

所有的視窗及控制項皆採用系統標準控制項，除非標準控制項不符合應用程式的需求。所有的控制項皆一板一眼地排列著，要一絲不苟地對齊擺放，偏一點都不行。「檔案總管」及「附屬應用程式」群組的那些應用程式是最標準的示範。

□ 由專業美工設計

介面、控制項及所有的圖形皆由專業美工事先規劃，在這個前提之下，介面怎麼作怪都行。視窗、按鈕不必是矩形，蘋果形狀、跑車形狀，發亮、陰影、光澤都好，控制項的擺放也完全不必依規則行事，唯一的準則就是一好看，雖然這又帶了主觀成分在裏頭。

簡言之，除非事先規劃好，否則不要隨興地更改使用者介面元件的任何屬性。下面是 WinAMP 程式分別套用兩種不同 skin 的模樣，雖然不見得人人喜愛，但我想至少不是令人厭惡的介面吧！

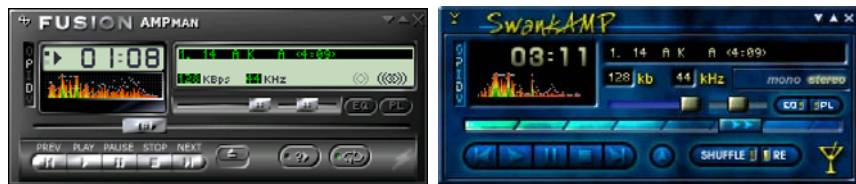


圖 6-22 / 分別套用「Fusion」及「Swank」skin 的 WinAMP 視窗

□ 不要寫視窗程式了，改寫 console mode 程式。

第三點當然只是玩笑，否則我也老早不該寫視窗程式了。因為我就是那種沒有專業美工撐腰，又總是喜歡在使用者介面上搞怪的傢伙。:Q

回到主題，提到這些，跟系統顏色有什麼關係呢？當然有關係，我指的是，當你希望遵照標準應用程式的開發原則，規規矩矩地撰寫標準使用者介面程式時，就必須藉由系統

顏色的支援，才能使應用程式在每個人的電腦上都能夠按照使用者的需求顯示。

打開「控制台」的「顯示器」元件，切換到「外觀」頁次，在這兒可以設定「使用中視窗標題列」、「應用程式背景」、「功能表」等等元件所使用的顏色，這些「代號」及「顏色」的對應，就是所謂的「系統顏色」。

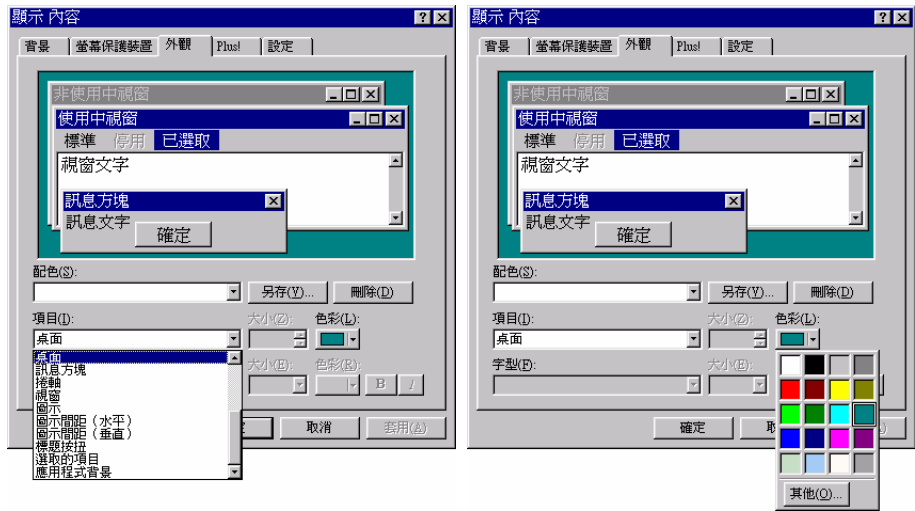


圖 6-23 / 控制台的「顯示器」元件「外觀」頁次可以設定系統顏色

這也就是為什麼同一支程式的功能表文字，在我的電腦上是黑色，在你的電腦上是灰色，在小白的電腦上卻是紅色的原因了—每個人都有偏好的顏色，都有一套屬於自己的系統顏色設定值。

因此，撰寫應用程式時，不應將程式使用到的任何顏色訂死，應該盡量使用系統顏色，讓每個使用者保有自己的選擇空間及需求，除非你確定程式所呈現的色彩風格能夠滿足所有的使用者。

VCL 中，大部分元件的顏色預設值都是系統顏色。在尚未更動的情況下，form 的 *Color* 屬性應該是 *clBtnFace*、*Font->Color* 屬性應該是 *clWindowText*，而不是 *clLtGray*（淺灰色）及 *clBlack*（黑色），即使你可能一直這樣以為。

下列是所有系統顏色的 API 顏色常數、VCL 顏色常數及敘述對照表：

表 6-24 / 系統顏色常數及敘述對照表

API 顏色常數	VCL 顏色常數	敘述
<i>COLOR_ACTIVECAPTION</i>	<i>clActiveCaption</i>	使用中視窗標題列
<i>COLOR_GRADIENTACTIVECAPTION</i>	無	使用中視窗漸層標題列
<i>COLOR_BACKGROUND</i>	<i>clBackground</i>	桌面
<i>COLOR_HIGHLIGHT</i>	<i>clHighlight</i>	選取項目
<i>COLOR_HIGHLIGHTTEXT</i>	<i>clHighlightText</i>	選取項目文字
<i>COLOR_CAPTIONTEXT</i>	<i>clCaptionText</i>	使用中視窗標題文字
<i>COLOR_WINDOW</i>	<i>clWindow</i>	視窗
<i>COLOR_WINDOWTEXT</i>	<i>clWindowText</i>	視窗文字
<i>COLOR_SCROLLBAR</i>	<i>clScrollBar</i>	捲軸
<i>COLOR_INACTIVECAPTION</i>	<i>clInactiveCaption</i>	非使用中視窗標題列
<i>COLOR_GRADIENTINACTIVECAPTION</i>	無	非使用中視窗漸層標題列
<i>COLOR_MENU</i>	<i>clMenu</i>	功能表
<i>COLOR_WINDOWFRAME</i>	<i>clWindowFrame</i>	視窗邊框
<i>COLOR_MENUTEXT</i>	<i>clMenuText</i>	功能表文字
<i>COLOR_ACTIVEBORDER</i>	<i>clActiveBorder</i>	使用中視窗邊界
<i>COLOR_INACTIVEBORDER</i>	<i>clInactiveBorder</i>	非使用中視窗邊界
<i>COLOR_APPWORKSPACE</i>	<i>clAppWorkspace</i>	應用程式背景
<i>COLOR_BTNFACE</i>	<i>clBtnFace</i>	立體物件
<i>COLOR_BTNshadow</i>	<i>clBtnShadow</i>	按鈕陰影
<i>COLOR_GRAYTEXT</i>	<i>clGrayText</i>	灰色文字
<i>COLOR_BTNTEXT</i>	<i>clBtnText</i>	按鈕文字
<i>COLOR_INACTIVECAPTIONTEXT</i>	<i>clInactiveCaptionText</i>	非使用中視窗標題字
<i>COLOR_BTNHIGHLIGHT</i>	<i>clBtnHighlight</i>	按鈕高亮度
<i>COLOR_3DDKSHADOW</i>	<i>cl3DDkShadow</i>	立體物件暗色陰影
<i>COLOR_3DLIGHT</i>	<i>cl3DLight</i>	立體物件高亮度

<i>COLOR_INFOTEXT</i>	<i>clInfoText</i>	提示文字
<i>COLOR_INFOBK</i>	<i>clInfoBk</i>	提示背景

「顯示器」控制台元件所列供使用者設定的系統顏色，都可以在此找到對應的API及VCL顏色常數⁸。

取得系統顏色

若要取得目前的系統顏色設定，必須呼叫 *GetSysColor* API 函式，傳入系統顏色代碼，取回真正的顏色值：

```
DWORD GetSysColor(  
    int          nIndex  
);
```

參數

nIndex Windows系統顏色常數，如*COLOR_BTNFACE*、*COLOR_INFOTEXT*等等。

回返值

nIndex 常數對應的真正顏色。

在使用系統顏色的場合裡，通常得呼叫 *GetSysColor* API 函式來取得系統顏色常數所對應的真正顏色。若是 API 顏色常數、VCL 顏色常數及 *TColor* 型態混用的情形，必須先瞭解各種型別及常數間的轉換函式才行。以 *SetTextColor* API 函式、*TCanvas.Pen.Color* 屬性及「立體物件」顏色常數為例，可以這樣使用：

- 呼叫 API 函式

```
SetTextColor(GetSysColor(COLOR_BTNFACE));
```

⁸ 「使用中視窗漸層標題列」及「非使用中視窗漸層標題列」這兩個顏色常數只有在 Windows 98 或安裝 4.0 以上版本Internet Explorer的系統才適用。

或

```
SetTextColors( ColorToRGB( clBtnFace );
```

- 使用 VCL 屬性或函式

```
Canvas->Pen->Color = (TColor)GetSysColor( COLOR_BTNFACE );
```

或

```
Canvas->Pen->Color = clBtnFace;
```

有些 Win32 API 函式對系統顏色常數提供特別的禮遇，例如使用筆刷將矩形區域填滿的 *FillRect* API 函式：

```
int FillRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr);
```

FillRect 函式的第三個參數是 brush handle，指向一個 GDI brush 物件，型別為 *HBRUSH*。但如果你只希望以某個系統顏色填滿，特別建立一個 brush 物件不但太麻煩而且浪費 GDI 資源，此時只須傳入系統顏色常數加一的數值，*FillRect* 函式就會使用指定的系統顏色填滿 *lprc* 矩形區域。例如，若要將 (0, 0) – (100, 100) 區域塗滿「選取項目」系統顏色，可以這樣叫用：

```
FillRect(DC, Classes::Rect(0, 0, 100, 100), COLOR_HIGHLIGHT + 1);
```

設定系統顏色

為了達成批次設定系統顏色的能力，設定系統顏色要比取得系統顏色來得麻煩多了。呼叫 *SetSysColors* API 函式可設定系統顏色：

```
BOOL SetSysColors(  
    int cElements,  
    CONST INT* lpaElements;  
    CONST COLORREF* lpaRgbValues  
);
```

參數

cElements 欲設定的系統顏色數目。

lpaElements 欲設定的系統顏色常數陣列，元素型別為 *Integer*，陣列大小由 *cElements*

參數指定。

lpRgbValues 欲設定的系統顏色值陣列，元素型別為 *COLORREF*，陣列大小由 *cElements* 參數指定。

回返值

如果成功，傳回 *true*；否則傳回 *false*。

每次設定系統顏色時，*SetSysColors* 函式都會廣播 *WM_SYSCOLORCHANGE* 視窗訊息給系統所有的最上層視窗，導致所有視窗重繪，產生不小的系統負荷，並且十分耗時，所以批次設定系統顏色的功能是十分必要的。雖然使用方法稍微複雜，但是如此一來，一道 *SetSysColors* 函式呼叫就可以同時為全部的系統顏色設定新值，長痛不如短痛，畫面閃爍一陣就過去了。

比如說，我想要同時更改「使用中視窗標題列」及「非使用中視窗標題列」兩種系統顏色，前者改為紅色，後者改成藍色，只要進行如下呼叫：

```
#0001 int ColorNo[2];
#0002 COLORREF Colors[2];
#0003
#0004 ColorNo[0] = COLOR_ACTIVECAPTION; // 使用中視窗標題列
#0005 ColorNo[1] = COLOR_INACTIVECAPTION; // 非使用中視窗標題列
#0006
#0007 Colors[0] = RGB(255, 0, 0); // 紅色
#0008 Colors[1] = RGB(0, 0, 255); // 藍色
#0009
#0010 // 同時設定兩種系統顏色
#0011 SetSysColors(2, ColorNo, Colors);
```

儲存設定值

呼叫 *SetSysColors* 函式雖然可以更改目前的系統顏色，但是並不會將設定值儲存起來，這些影響只限於目前的 Windows 階段。如果希望這些設定能夠持續到下次登入，必須自行將新的設定值寫入登錄資料庫才行。

系統顏色設定儲存於登錄資料庫的 `HKEY_CURRENT_USER\Control Panel\Colors`

鍵碼⁹，儲存格式為顏色值的紅、綠、藍三原色色量組成的字串，每個使用者可以擁有自己的偏好設定。呼叫`ColorToRGBString`函式可將`TColor`顏色轉為三原色字串，再透過`TRegIniFile`類別來存取登錄資料庫。

同樣以「使用中視窗標題列」及「非使用中視窗標題列」系統顏色為範例，前者寫入紅色，後者寫入藍色：

```
#0001 // 儲存在登錄資料庫中
#0002 TRegIniFile* r = new TRegIniFile("Control Panel");
#0003 try {
#0004     // 寫入 "255 0 0" 字串到 Colors 機碼的 ActiveTitle 字串值
#0005     r->WriteString("Colors", "ActiveTitle",
#0006         ColorToRGBString(clRed));
#0007     // 寫入 "0 255 0" 字串到 Colors 機碼的 InactiveTitle 字串值
#0008     r->WriteString("Colors", "InactiveTitle",
#0009         ColorToRGBString(clBlue));
#0010 } __finally {
#0011     delete r;
#0012 }
```

由於`TRegistry/TRegIniFile`類別的`RootKey`屬性預設值是`HKEY_CURRENT_USER`，所以傳入“Control Panel”字串建立`TRegIniFile`物件後，不需額外指定主鍵即可使用。

滑鼠指標

與系統顏色一樣，系統滑鼠指標也有許多對應的常數，分別在不同場合使用。下列是系統滑鼠指標的預設圖案與呼叫`LoadCursor` API 函式時使用的常數、VCL 常數及敘述對照表：

⁹ 事實上存放在`HKEY_USERS\User GUID\Control Panel\Colors`機碼，只是當使用者登入後，系統會自動將該使用者的資料`HKEY_USERS\User GUID` 對映至`HKEY_CURRENT_USER`，所以直接使用`HKEY_CURRENT_USER`機碼比較方便。

表 6-25 / 系統滑鼠指標常數及敘述對照

預設圖案	資源指標常數	VCL 常數	敘述
	<i>IDC_ARROW</i>	<i>crArrow</i>	一般選取
	<i>IDC_CROSS</i>	<i>crCross</i>	精確選取
	<i>IDC_IBEAM</i>	<i>crIbeam</i>	文字選取
	<i>IDC_SIZEALL</i>	<i>crSize</i>	移動
	<i>IDC_SIZENESW</i>	<i>crSizeNESW</i>	對角線調整 2
	<i>IDC_SIZENS</i>	<i>crSizeNS</i>	垂直調整
	<i>IDC_SIZENWSE</i>	<i>crSizeNWSE</i>	對角線調整 1
	<i>IDC_SIZEWE</i>	<i>crSizeWE</i>	水平調整
	<i>IDC_UPARROW</i>	<i>crUpArrow</i>	其它選取
	<i>IDC_WAIT</i>	<i>crHourGlass</i>	忙碌中
	<i>IDC_NO</i>	<i>crNo</i>	不可用
	<i>IDC_APPSTARTING</i>	<i>crAppStart</i>	背景作業
	<i>IDC_HELP</i>	<i>crHelp</i>	說明選取

取得滑鼠指標

取得滑鼠指標？說得明白一些，我指的是取得指向 GDI cursor 物件的滑鼠指標 handle。和其它 GDI 物件相同，擁有滑鼠指標 handle 後，才能夠自由地進行 GDI cursor 物件（即滑鼠指標）的相關操作。獲取滑鼠指標 handle 的方法真不少，分為下列數種情況來討論：

取得自訂指標

最常見的用法就是，自行設計一個指標，連結時併入執行檔的資源區段。然後在程式執行時，呼叫 *LoadCursor* API 函式將指標取出使用。

假設我已將某個滑鼠指標圖案加入 MYCURSOR.RES 資源檔，資源名稱訂為 “MyCursor”，並在專案原始碼中加入 {*\$R MYCURSOR.RES*} 資源編譯指示，指示連結程式將資源檔案連同執行檔一併連結。

如此一來，在此專案中，呼叫下列程式碼就可將自訂滑鼠指標取出，並指定為 *Mem01* 元件的滑鼠指標：

```
Screen->Cursors[100] = LoadCursor(hInstance,  
    MAKEINTRESOURCE("MyCursor"));  
Mem01->Cursor = 100;
```

Screen 是 *TScreen* 類別的全域變數，它的 *Cursors* 陣列屬性存放著所有可用的滑鼠指標 handle。編號 100 是隨意訂下的，你可為自訂指標指派任何大於零的正整數編號。將 *LoadCursor* 函式傳回的指標 handle 指派給 *Screen->Cursors* 屬性後，即可自由使用此滑鼠指標，你不必擔心資源是否釋放的問題，VCL 會負責歸還事宜。

取得系統指標

若想取得使用者在控制台所設定的滑鼠指標，同樣地呼叫 *LoadCursor* API 函式，並傳入表 6-25 中所列的資源指標常數，即可取得其指標 *handle*。以「忙碌中」滑鼠指標為例，它的資源指標常數為 *IDC_WAIT*：

```
hc = LoadCursor(hInstance, IDC_WAIT);
```

其實不必這麼麻煩，系統指標 *handle* 可以直接由 *Screen->Cursors* 陣列屬性取得：

```
hc = Screen->Cursors[crHourGlass];
```

從檔案讀取指標

如果希望取出位於 *.CUR* 滑鼠指標檔案或 *.ANI* 動態滑鼠指標檔案內的指標來使用，呼叫 *LoadCursorFromFile* API 函式是最簡單的做法，傳入檔案名稱，就可以取得指標 *handle*。例如：

```
hc = LoadCursorFromFile("rocket.cur");  
hc = LoadCursorFromFile("animated_dog.ani");
```

取得作業系統預設指標

找遍 Win32 技術文件，我不曾見過如何取得作業系統預設指標的作法。我所謂的「作業系統預設指標」指的就是表 6-25 最左欄所列出的指標圖案，雖然不曉得該怎麼做，但我相信一定有辦法。Why？因為控制台「滑鼠」元件的【使用預設值】按鈕做的就是這件事：無論目前的系統指標設定為何，它都可以復原為作業系統的預設滑鼠指標。







圖 6-26 / 「滑鼠」元件的「使用預設值」功能不知從何而來？










沒關係，筆者我可是身經百戰的刁鑽狙擊手，拿出 BoundsChecker，再配合 SoftICE 追蹤「滑鼠」控制台元件的行為，很快地發現，哇咧，原來它使用了未公開的函式呼叫常數，真是 oOxX！這個未公開的函式呼叫是這樣的：呼叫 LoadImage API，指標編號（第二個參數）傳入某些特定數值，載入旗標（最後一個參數）設定為 0x240，就可取得作業系統的預設滑鼠指標 handle，我將這些特定的指標編號列於表 6-27。

以「忙碌中」滑鼠指標為例，下列呼叫可取得作業系統預設的「忙碌中」指標 handle：

```
hc = LoadImage(0, 0x66, IMAGE_CURSOR, 0, 0, 0x240);
```

表 6-27 / 系統滑鼠指標預設圖案及相關常數對照

預設圖案	API 常數	登錄資料庫鍵值	LoadImage 未公開常數
	OCR_NORMAL	Arrow	0x64
	OCR_CROSS	Crosshair	0x67
	OCR_IBEAM	NWPen	0x65
	OCR_SIZEALL	SizeAll	0x6D

	<i>OCR_SIZE_NESW</i>	<i>SizeNESW</i>	<i>0x6A</i>
	<i>OCR_SIZE_NS</i>	<i>SizeNS</i>	<i>0x6C</i>
	<i>OCR_SIZE_NWSE</i>	<i>SizeNWSE</i>	<i>0x69</i>
	<i>OCR_SIZE_WE</i>	<i>SizeWE</i>	<i>0x6B</i>
	<i>OCR_UP</i>	<i>UpArrow</i>	<i>0x68</i>
	<i>OCR_WAIT</i>	<i>Wait</i>	<i>0x66</i>
	<i>OCR_NO</i>	<i>No</i>	<i>0x6E</i>
	<i>OCR_APPSTARTING</i>	<i>AppStarting</i>	<i>0x6F</i>
	<i>OIC_QUES</i>	<i>Help</i>	<i>0x70</i>

設定系統滑鼠指標

呼叫 *SetSystemCursor* 函式可以指定目前的系統滑鼠指標，若希望新設定延續至下次登入，必須自行將指標檔案路徑寫入登錄資料庫，*SetSystemCursor* 函式不會幫你做這件事。

SetSystemCursor 函式使用方法如下：

BOOL SetSystemCursor (

HCURSOR hcur,

DWORD id

);

參數

hcur 滑鼠指標 handle，可呼叫 *LoadCursor*、*LoadCursorFromFile* 或其它 API 函式取得。

id 系統滑鼠指標 API 常數，請參照表 6-27。

回返值

如果成功，傳回 *true*；否則傳回 *false*。

這個函式的使用極為簡單，若我希望將「一般選取」指標改為 COW.CUR 檔案裡的指標，這麼設定就行了：

```
hc = LoadCursorFromFile("cow.cur");  
if (hc != 0) SetSystemCursor(hc, OCR_NORMAL);
```

Info

在 Windows 95/98 下呼叫 *SetSystemCursor* 函式設定系統滑鼠指標時，若傳入從 .ANI 檔案（利用 *LoadCursorFromFile* 函式）載入的動態滑鼠指標 handle，會使作業系統完全當掉。請小心，這臭蟲白白浪費了我好多時間呀 :~

不過，若將動態滑鼠指標檔案路徑寫入登錄資料庫，重新啟動系統，讓系統自動載入使用，就不會出問題。

儲存設定值

滑鼠指標設定儲存於登錄資料庫的 HKEY_CURRENT_USER\Control Panel\Cursors 機碼，個別指標的對應鍵值列於表 6-27，每回系統重新啟動時，就會根據此處設定值讀取滑鼠指標檔案。例如，下列程式碼可將「忙碌中」系統滑鼠指標更換為

“c:\cursors\beauty.ani” 指標檔案：

```
TRegIniFile* r = new TRegIniFile("Control Panel");  
try {  
    // 將指標檔案寫入登錄資料庫  
    r->WriteString("Cursors", "Crosshair", "c:\\cursors\\beauty.ani");  
} __finally {  
    delete r;  
}
```

系統音效

Windows 可真沒浪費它對多媒體的支援，讓我們為使用者界面的常見操作或事件設定系統音效，每當該動作或事件進行時，就會根據設定值撥放出指定的音效或音樂。

剛接觸 Windows 95 時，我真的被這個功能吸引住了。我老愛不停地更換、試用不同的系統音效，為各個事件指定各種不搭調的音效，不絕的系統音效能夠奇妙地營造出熱鬧的氣氛，尤其在孤寂的冬夜裡，更可以驅走獨自面對電腦的寂寞。

下列是系統音效的事件名稱、登錄資料庫區段及事件敘述對照表：

表 6-28 / 系統音效及相關常數對照

音效／事件名稱	登錄資料庫區段	事件敘述
<i>.Default</i>	<i>.Default\Default\Current</i>	預設嗶聲
<i>AppGPFault</i>	<i>.Default\AppGPFault\Current</i>	程式錯誤
<i>Maximize</i>	<i>.Default\Maximize\Current</i>	最大化
<i>MenuCommand</i>	<i>.Default\MenuCommand\Current</i>	功能表指令
<i>MenuPopup</i>	<i>.Default\MenuPopup\Current</i>	快顯功能表
<i>Minimize</i>	<i>.Default\Minimize\Current</i>	最小化
<i>Open</i>	<i>.Default\Open\Current</i>	開啓程式
<i>Close</i>	<i>.Default\Close\Current</i>	關閉程式
<i>RestoreDown</i>	<i>.Default\RestoreDown\Current</i>	往下還原
<i>RestoreUp</i>	<i>.Default\RestoreUp\Current</i>	往上還原
<i>RingIn</i>	<i>.Default\RingIn\Current</i>	來電
<i>Ringout</i>	<i>.Default\Ringout\Current</i>	外撥
<i>SystemAsterisk</i>	<i>.Default\SystemAsterisk\Current</i>	星號
<i>SystemDefault</i>	<i>.Default\SystemDefault\Current</i>	SystemDefault
<i>SystemExclamation</i>	<i>.Default\SystemExclamation\Current</i>	驚歎聲
<i>SystemExit</i>	<i>.Default\SystemExit\Current</i>	結束 Windows

<i>SystemHand</i>	<i>.Default\SystemHand\.Current</i>	緊急停止
<i>SystemQuestion</i>	<i>.Default\SystemQuestion\.Current</i>	問題
<i>SystemStart</i>	<i>.Default\SystemStart\.Current</i>	啓動 Windows
<i>EmptyRecycleBin</i>	<i>Explorer\EmptyRecycleBin\.Current</i>	清理資源回收筒

事件敘述

很特別的是，系統音效的事件敘述放在登錄資料庫內。我們可以根據系統音效名稱，從登錄資料庫取得系統音效的事件敘述，這些敘述字串置於 `HKEY_CURRENT_USER\AppDataEvents\EventLabels` 機碼內各個系統音效／事件名稱相對應的鍵值中。

由登錄資料庫取得系統音效事件敘述的好處是：在中文 Windows 上，取得的是中文敘述；在德文 Windows 上，取得的就是德文敘述，可以省下軟體國際化的麻煩。

取得及設定系統音效

每個系統音效事件都對應到一個音效檔案，檔案的路徑也存放於登錄資料庫，位置是：`HKEY_CURRENT_USER\AppDataEvents\Schemes\Apps\EventLabels` 機碼內各個系統音效／事件名稱相對應的鍵值中，請參考表 6-28 來查詢系統音效對應的登錄資料庫區段。

播放系統音效

播放系統音效最直覺的方法就是，從登錄資料庫取得音效檔名，再交給支援音效檔播放的函式或元件進行播放。

不過系統提供了比較方便的作法，讓我們不必預先取得音效檔名也能播放系統音效。方法是，呼叫 `PlaySound` API 函式，檔名（第一個參數）傳入系統音效名稱，播放旗標（第

三個參數) 傳入 `SND_ALIAS`，就可直接以系統音效名稱來播放音效。例如下列呼叫即可播放「啓動 Windows」系統音效：

```
PlaySound("SystemStart", 0, SND_ALIAS);
```

系統字型

字型也是維持使用者介面一致性的重要元素。Windows 共有六種系統字型，分別是「圖示文字」、「視窗標題列」、「色板標題」、「功能表」、「狀態列」、「訊息視窗」。所有的系統字型皆依賴 `SystemParametersInfo` API 函式來取得、設定，它們所使用的動作代碼及資料欄位如下：

表 6-29 / 系統字型的取得方法

系統字型	動作代碼	欄位
圖示	<code>SPI_GETICONTITLELOGFONT</code>	直接取得
視窗標題列	<code>SPI_GETNONCLIENTMETRICS</code>	<code>TNonClientMetrics::lfCaptionFont</code>
色板標題	<code>SPI_GETNONCLIENTMETRICS</code>	<code>TNonClientMetrics::lfSmCaptionFont</code>
功能表	<code>SPI_GETNONCLIENTMETRICS</code>	<code>TNonClientMetrics::lfMenuFont</code>
狀態列	<code>SPI_GETNONCLIENTMETRICS</code>	<code>TNonClientMetrics::lfStatusFont</code>
訊息視窗	<code>SPI_GETNONCLIENTMETRICS</code>	<code>TNonClientMetrics::lfMessageFont</code>

`TNonClientMetrics` 結構除了包含五種系統字型外，還包括不少視窗的顯示參數。

`TNonClientMetrics` 結構宣告如下：

```
typedef struct tagNONCLIENTMETRICS
{
    UINT    cbSize;           // 結構長度，請用 sizeof 運算子取得
    int     iBorderWidth;    // 視窗邊框寬度點數
    int     iScrollWidth;   // 縱向捲軸寬度點數
    int     iScrollHeight;  // 橫向捲軸高度點數
    int     iCaptionWidth;  // 視窗標題按鈕寬度點數
    int     iCaptionHeight; // 視窗標題按鈕高度點數
    LOGFONTA lfCaptionFont; // 視窗標題列字型
}
```



```

int    iSmCaptionWidth;           // 小型視窗標題 10 按鈕寬度點數
int    iSmCaptionHeight;         // 小型視窗標題按鈕高度點數
LOGFONTA lfSmCaptionFont;       // 小型視窗標題列字型
int    iMenuWidth;              // 功能表寬度點數
int    iMenuHeight;            // 功能表寬高度點數
LOGFONTA lfMenuFont;           // 功能表字型
LOGFONTA lfStatusFont;         // 狀態列字型
LOGFONTA lfMessageFont;        // 訊息視窗內文字型
} NONCLIENTMETRICS, *PNONCLIENTMETRICS, FAR* LPNONCLIENTMETRICS;

typedef tagNONCLIENTMETRICS TNonClientMetrics;

```

開發一般應用程式時，這些參數可能極少派上用場，但如果你需要自行撰寫視覺元件，例如具有立體陰影效果的功能表、可顯示多重顏色的訊息視窗等等，視窗大小及字型設定就必須使用這些參數，維持一致的風格及介面。

取得及寫入系統字型

由表 6-29 可以得知各種系統字型的取得方法。下列程式碼中，我取出「圖示文字」字型，然後將它指派給「功能表」字型，讓功能表也使用與圖示標題相同的字型：

```

#0001 void __fastcall TForm1::Button1Click(TObject* Sender)
#0002 {
#0003     TLogFont lf;
#0004     TNonClientMetrics NM;
#0005
#0006     // 取出「圖示文字」字型
#0007     SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(lf),
#0008         &lf, 0);
#0009     // 取出整個 TNonClientMetrics 結構
#0010     SystemParametersInfo(SPI_GETNONCLIENTMETRICS, sizeof(NM),
#0011         &NM, 0);
#0012
#0013     NM.lfMenuFont = lf; // 指派「功能表」字型
#0014     // 設定整個 TNonClientMetrics 結構

```

10 當 *TForm* 的 *BorderStyle* 為 *bsSizeToolWin* 或 *bsToolWindow* 時，就會使用小型視窗標題列的設定值。

```
#0015     SystemParametersInfo(SPI_SETNONCLIENTMETRICS, sizeof(NM),
#0016         &NM, 0);
#0017 }
```

雖然只是簡單的系統字型指派動作，但做起來比想像中麻煩些，原因是「功能表」字型位於 *TNonClientMetrics* 結構，所以雖然只想更改「功能表」字型，還是得先將目前的 *TNonClientMetrics* 結構取出，修改 *lfMenuFont* 欄位後，再將整個 *TNonClientMetrics* 結構寫回。

通常我們不會進行這種無意義的指派工作，取出系統字型的目的是在程式中使用它們，但這 *TLogFont* 結構究竟要怎麼用？

各種字型物件的處理

呼叫 *SystemParametersInfo* 函式可以取回的所謂「系統字型」，指的是 *TLogFont* 結構，與我們平日所用的 VCL 字型類別 *TFont* 大不相同。若要在 VCL 應用程式中順利使用系統字型，就必須先瞭解它們之間的相互關係及轉換方式才行。

TLogFont 是 Win32 SDK 定義的字型描述結構，宣告如下：

```
typedef struct tagLOGFONTA
{
    LONG         lfHeight;           // 高度
    LONG         lfWidth;           // 寬度
    LONG         lfEscapement;      // 旋轉角度
    LONG         lfOrientation;    // 旋轉角度
    LONG         lfWeight;         // 粗細
    BYTE        lfItalic;          // 是否斜體
    BYTE        lfUnderline;       // 是否加底線
    BYTE        lfStrikeOut;       // 是否加刪除線
    BYTE        lfCharSet;         // 字元集
    BYTE        lfOutPrecision;    // 輸出精確度
    BYTE        lfClipPrecision;   // 裁剪精確度
    BYTE        lfQuality;         // 輸出品質
    BYTE        lfPitchAndFamily;  // 字型種類
    CHAR        lfFaceName[LF_FACESIZE]; // 字型名稱
} LOGFONTA, *PLOGFONTA, NEAR *NPLOGFONTA, FAR *LPLOGFONTA;
```

```
typedef tagLOGFONTA TLogFont;
```

通常，我們會將 *TLogFont* 結構當作「字型申請書」來使用，將想要的字型高度、寬度、旋轉角度、式樣、名稱等等屬性通通填入，再呼叫 *CreateFont* 或 *CreateFontIndirect* 函式來建立 GDI font 物件。也許系統上沒有安裝你要求的字型，也許該字型無論達到你要求的效果，不過無論如何，font mapper 機制都會回覆請求，想辦法提供一個最接近要求的字型。取得 font 物件的 handle 後，我們就可以開始使用這個字型。

例如，若要使用特別扁及逆時針旋轉 345 度的字型各繪出一行文字，只要將 *TLogFont* 結構填好後呈報上去，取回 font handle 後就可以畫了！下列程式碼示範「以 *TLogFont* 申請字型」的做法：

```
#0001 void __fastcall TForm1::Button1Click(TObject *Sender)
#0002 {
#0003     TLogFont lf;
#0004
#0005     memset(&lf, 0, sizeof(lf));
#0006     lf.lfHeight = 20; // 高度 20
#0007     lf.lfWidth = 40; // 寬度，故意設得很寬
#0008     lf.lfWeight = FW_NORMAL;
#0009     lf.lfCharSet = DEFAULT_CHARSET;
#0010     lf.lfOutPrecision = OUT_DEFAULT_PRECIS;
#0011     lf.lfClipPrecision = CLIP_DEFAULT_PRECIS;
#0012     lf.lfQuality = DEFAULT_QUALITY;
#0013     lf.lfPitchAndFamily = DEFAULT_PITCH | FF_DONTCARE;
#0014     strcpy(lf.lfFaceName, "新細明體");
#0015
#0016     Image1->Canvas->Font->Handle = CreateFontIndirect(&lf);
#0017     Image1->Canvas->TextOut(10, 10, "我是特別扁的文字");
#0018
#0019     lf.lfHeight = 60; // 高度 60
#0020     lf.lfWidth = 0; // 使用預設寬度
#0021     lf.lfEscapement = 345 * 10; // 逆時針旋轉 345 度
#0022
#0023     Image1->Canvas->Font->Handle = CreateFontIndirect(&lf);
#0024     Image1->Canvas->TextOut(20, 50, "我是會旋轉的文字");
#0025 }
```

執行結果如下：

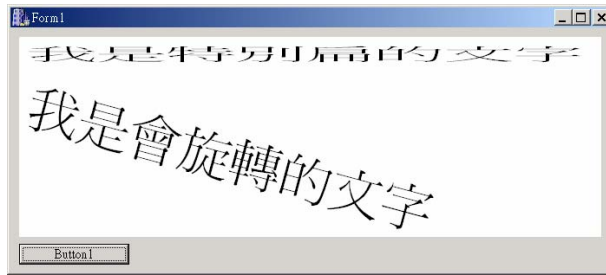


圖 6-30 / 特別扁及會旋轉的文字

簡言之，*TLogFont* 是「死的」字型資訊，它只是一個資料結構，用來描述字型，無法直接使用；GDI 的 font 物件是「活的」字型，你可以使用它來繪製文字，不過 font 物件一旦建立，它的字型屬性就不能更動，若想更換字型，必須另外建立新的 font 物件；而 VCL 的 *TFont* 類別結合兩者的優點，它是 GDI font 物件的包裝類別，透過它可以很方便地取用現成的 font 物件，直接繪製文字，但是也可以隨時更動字型的屬性，*TFont* 會在必要時建立 font 物件，為我們處理所有的繁瑣細節。

TLogFont、font handle 及 *TFont* 三者如何轉換呢？你可將 font handle 視為 *TLogFont* 結構及 *TFont* 類別之間的橋樑，這兩者之間的轉換必須經過 font handle 才行。所以，只要知道 *TLogFont* 結構、*TFont* 類別兩者與 font handle 的轉換方式，三者之間的互換就不成問題了。*TLogFont* 結構與 font handle 的關聯是：

- 從 font handle 取得 *TLogFont* 結構
`GetObject(font_handle, sizeof(TLogFont), &lf);`
- 以 *TLogFont* 結構請求建立 font 物件，取回 font handle
`font_handle = CreateFontIndirect(&lf);`

至於 *TFont* 類別與 font handle 的轉換方式就簡單了：*TFont.Handle* 是個可讀寫的屬性，直接讀取或指派此屬性即可：

```
font_handle = Font->Handle; // 讀取 font handle  
Font->Handle = font_handle; // 指派 font handle
```

瞭解轉換規則後，就可以自由地操控 Windows 字型及系統字型。

最後來段範例，下列這段程式碼可使 *Button1* 元件使用「功能表」字型，並將「圖示文字」字型指定為 *Button2* 元件目前的字型：

```
#0001 void __fastcall TForm1::Button1Click(TObject* Sender)
#0002 {
#0003     TLogFont lf;
#0004     TNonClientMetrics NM;
#0005
#0006     // 取出「圖示文字」字型
#0007     SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(lf),
#0008         &lf, 0);
#0009     // 以 CreateFontIndirect 建立 font 物件，指派給 Button1.Font
#0010     Button1->Font->Handle = CreateFontIndirect(&NM.lfMenuFont);
#0011
#0012     // 取得描述 Button2->Font 的 TLogFont 結構
#0013     GetObject(Button2->Font->Handle, sizeof(TLogFont), &lf);
#0014     // 呼叫 SystemParametersInfo 將此 TLogFont 結構設定為圖示字型
#0015     SystemParametersInfo(SPI_SETICONTITLELOGFONT, sizeof(lf),
#0016         &lf, 0);
#0017 }
```

桌面圖示

觀察使用者的 Windows 桌面，或許是瞭解他電腦使用習慣的最佳方式。

有的人桌面凌亂不堪，充斥隨手丟置的文件及程式捷徑，甚至連暫存檔、解壓縮程式的目的資料夾都擺到桌面來了，要在他的桌面上找到「我的電腦」圖示可真是件難事。

有的人桌面一絲不苟，不使用桌面快捷功能表提供的「自動排列」功能，每個圖示的位置完全手工擺設，分門別類地擺在桌面的不同角落，圖示圖形還經常整套整套地更換風格。在這種桌面工作真是種享受，井然有序，就像是媽媽剛整理過似的。

有的人桌面圖示眾多，資料夾裡頭的檔案不算，少說也有五六十個，我就是這類型的使用者。將常用的程式、有趣的文件等等通通置於桌面，不必花太多時間整理，工作效率卻能大大增加，雖然每個看過的朋友都被佔滿桌面的圖示「群」給嚇壞了！:p

最後一種使用者是追求美感的桌布族，他們認為桌面就是要擺設漂漂亮亮桌布用，放一堆捷徑啊，文件啊，資料夾啊在上頭，強烈破壞整體性，是絕對不可饒恕的行為。他們是呼喊著「只要桌面，不要圖示」的桌布一族。

數到三，快快從桌面上消失...

要達成桌布族的夢想並不難，呼叫`ShowWindow`函式，傳入背景視窗¹¹的視窗handle並指定新的視窗顯示狀態，即可控制桌面圖示的顯示狀態：

```
ShowWindow(GetDesktopListView(), SW_SHOW); // 顯示桌面視窗
```

或

```
ShowWindow(GetDesktopListView(), SW_HIDE); // 隱藏桌面視窗
```

由於桌面圖示是由桌面視窗提供的，因此將桌面視窗隱藏後，桌面圖示就會完全消失。此時，背景的桌布及填圖樣式還好端端地留在畫面上，其中原理已在上一章「一頭栽入桌面的世界」講述得十分詳細，若你還不甚明白，請回頭閱讀。

揮之不去的四劍客

桌面上有四個特別的圖示，分別是「我的電腦」、「網路上的芳鄰」、「我的文件夾」及「資源回收筒」，除了無法以快捷功能表的【內容】選項更換圖示外，無論按下【Delete】鍵或拉進資源回收筒，也都無法刪除它們。

辦法總是有的，咱們技高一籌，直接切入登錄資料庫更改它們的標題及圖示：

¹¹ 背景視窗指的是shell所產生的`SysListView32`視窗，而不是指桌面視窗。

表 6-31 / 特殊圖示及其圖示檔案登錄位置對照

圖示名稱	圖示檔案登錄位置
我的電腦	{20D04FE0-3AEA-1069-A2D8-08002B30309D}\DefaultIcon\Default
網路上的芳鄰	{208D2C60-3AEA-1069-A2D7-08002B30309D}\DefaultIcon\Default
我的文件夾	{450D8FBA-AD25-11D0-98A8-0800361B1103}\DefaultIcon\Default
資源回收筒 (空)	{645FF040-5081-101B-9F08-00AA002F954E}\DefaultIcon\Empty
資源回收筒 (滿)	{645FF040-5081-101B-9F08-00AA002F954E}\DefaultIcon\Full

圖示標題字串值由圖示檔名登錄位置上一層機碼的預設值決定。這些登錄位置並非唯一，可以是 HKEY_CURRENT_USER\Software\Classes\CLSID 機碼，也可以放在 HKEY_LOCAL_MACHINE\Software\Classes\CLSID 機碼下。這表示使用者可以自行擁有一套特殊桌面圖示設定值，不必理會系統設定，除非該使用者沒有使用者設定值，才會使用系統設定。

撰寫更改這些設定值的軟體時，最好修改 HKEY_CURRENT_USER\Software\Classes\CLSID 鍵值下的設定，也就是使用者設定值。一來避免影響到其它使用者的設定，二來假使該用戶早有使用者設定值，那麼更改系統設定也不會影響他的桌面，因為使用者設定的優先權較高，shell 會優先考慮使用者的設定。

Info

讓人無法理解的是，只有圖示檔案才能由使用者自行設定，圖示標題一律使用系統設定。

所以，若要同時更改圖示標題及檔案，必須先至系統設定處更改圖示標題，再到使用者設定處更換圖示檔案。

例如，以下列程式碼將「我的電腦」改為「今天心情很好」，將「網路上的芳鄰」改為

「街坊鄰居」，同時更換它們的圖示檔案：

```
#0001 void __fastcall TForm1::Button1Click(TObject* Sender)
#0002 {
#0003     const AnsiString MY_COMPUTER_GUID =
#0004     "Software\\Classes\\CLSID\\{20D04FE0-3AEA-1069-A2D8-08002B30309D}";
#0005     const AnsiString NEIGHBORHOOD_GUID =
#0006     "Software\\Classes\\CLSID\\{208D2C60-3AEA-1069-A2D7-08002B30309D}";
#0007
#0008     TRegistry* r = new TRegistry;
#0009     try {
#0010         r->RootKey = HKEY_LOCAL_MACHINE; // 系統設定
#0011
#0012         if (r->OpenKey(MY_COMPUTER_GUID, true)) {
#0013             r->WriteString("", "今天心情很好"); // 改為「今天心情很好」
#0014             r->CloseKey();
#0015         }
#0016
#0017         if (r->OpenKey(NEIGHBORHOOD_GUID, true)) {
#0018             r->WriteString("", "街坊鄰居"); // 改為「街坊鄰居」
#0019             r->CloseKey();
#0020         }
#0021
#0022         r->RootKey = HKEY_CURRENT_USER; // 使用者設定
#0023
#0024         if (r->OpenKey(MY_COMPUTER_GUID + "\\DefaultIcon", true)) {
#0025             // 更換「我的電腦」圖示
#0026             r->WriteString("", "C:\\WINNT\\EXPLORER.EXE,1");
#0027             r->CloseKey();
#0028         }
#0029
#0030         if (r->OpenKey(NEIGHBORHOOD_GUID + "\\DefaultIcon", true)) {
#0031             // 更換「網路上的芳鄰」圖示
#0032             r->WriteString("", "E:\\WINNT\\EXPLORER.EXE,1");
#0033             r->CloseKey();
#0034         }
#0035     } __finally {
#0036         delete r;
#0037     }
#0038 }
```

按下 *Button1* 按鈕，修改完成後，趕緊看一下桌面，咦，怎麼什麼事都沒發生？

哦～原來此時背景視窗還不曉得登錄資料庫已被修改，所以只要將輸入焦點移至背景視窗，再按下【F5】，強迫背景視窗更新顯示，就可以看到修改後的結果。

不過，總不能要求使用者也跟我們一樣，執行程式修改登錄資料後，再按【F5】來更新桌面吧！一個簡單的解決方法是，呼叫xDesktop單元的*RebuildIconCache*函式¹²，此函式會重新建立圖示快取（icon cache），這會強迫所有正顯示在畫面上的圖示重新載入，所以背景視窗也會重新讀取圖示檔案，更新桌面圖示。程式如下所列：

```
#0001 void RebuildIconCache()
#0002 {
#0003     int IconW;
#0004
#0005     IconW = GetSystemMetrics(SM_CXICON);
#0006     TRegIniFile* r = new TRegIniFile("Control Panel\\Desktop");
#0007     try {
#0008         r->WriteString("WindowMetrics", "Shell Icon Size",
#0009             IntToStr(IconW - 1));
#0010         SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, 0);
#0011         r->WriteString("WindowMetrics", "Shell Icon Size",
#0012             IntToStr(IconW));
#0013         SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, 0);
#0014     } __finally {
#0015         delete r;
#0016     }
#0017 }
```

你瞧，原來只是簡單地更改系統的預設圖示大小，再廣播 *WM_WININICHANGE* 視窗訊息，就可以強迫圖示快取重新建立。

另外還有一些圖示相關參數，不過又得跟 *SystemParametersInfo* 函式打照面了，相關功能及動作代碼列表如下：

表 6-32 / 與桌布圖示有關的 *SystemParametersInfo* 函式動作代碼

動作代碼	含意
<i>SPI_GETICONMETRICS</i> <i>SPI_SETICONMETRICS</i>	擷取／設定 <i>TIconMetrics</i> 結構，包含水平間距、垂直間距、圖示標題是否自動換行及圖示標題字型。
<i>SPI_GETICONTITLEWRAP</i> <i>SPI_SETICONTITLEWRAP</i>	擷取／設定目前圖示標題是否自動換行。

¹² 請參考附錄A「我的程式庫」對此函式的其它說明。

桌布及樣式

設定桌布及樣式大概可說是 Win32 API 中最簡單，但其影響也最長遠的動作了，只要影像檔不刪除，登錄資料庫也沒更動，桌布及樣式就可以永久地駐留在桌面上。

桌布及樣式設定有關的動作代碼如下：

表 6-33 / 桌布及背景填圖樣式相關的 SystemParametersInfo 動作代碼

動作代碼	含意
<i>SPI_SETDESKPATTERN</i>	設定背景樣式， <i>pvParam</i> 指向八個由 0 ~ 255 整數組成的字串，用來表示長寬各為 8 點的單色點陣圖。例如，「170 85 170 85 170 85 170 85」代表「50% 灰色」、「127 65 65 65 65 65 127 0」代表「方塊」樣式。
<i>SPI_GETDESKWALLPAPER</i> <i>SPI_SETDESKWALLPAPER</i>	擷取／設定背景圖， <i>pvParam</i> 指向 BMP 影像檔名。 更換桌布前請先將相關設定值寫入登錄資料庫（請參考表 6-35）；若影像檔名為空白字串，則會除移桌布。

填圖樣式

於是乎，我們可以執行下列呼叫來設定填圖樣式。我選擇的是棋盤式黑白交錯的樣式：

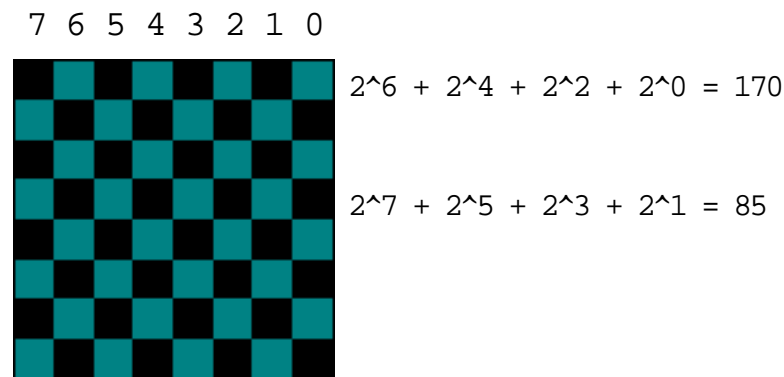


圖 6-34 / 填圖樣式的計算

拼湊起來，填圖樣式字串為「170 85 170 85 170 85 170 85」，下列呼叫即可更換桌面的填圖樣式：

```
AnsiString sDeskPattern = "170 85 170 85 170 85 170 85";  
SystemParametersInfo(SPI_SETDESKPATTERN, 0, sDeskPattern.c_str(),  
    SPIF_UPDATEINIFILE);
```

遺憾的是，Win32 文件中找不到 *SPI_GETDESKPATTERN* 常數，也就是說，Win32 並沒有提供取得目前桌面填圖樣式的方法，如果真有必要，只好自力救濟了！系統會將目前使用的填圖樣式儲存在登錄資料庫的 `HKEY_CURRENT_USER\Control Panel\Pattern` 字串鍵值，直接取用即可；必須注意的是，若目前沒有設定樣式，取得的字串值不是空字串，而是“(None)”。

Talk

更遺憾的是，Windows NT 4.0 在中文化的過程中，竟然連此鍵值的名稱及空白指示字串也翻譯成中文，他們將此處的“Pattern”改為“類型”、“(None)”改為“(無)”，重點是內部還彼此不相同。*SystemParametersInfo* 函式會將樣式寫入“類型”，而「顯示器」控制台元件會從“Pattern”讀取，真是亂七八糟！

在此誠摯地祈禱，下次中文化時，請不要連登錄資料庫的鍵值名稱一塊中文化，翻譯人員的無知及順手雞婆，和中文化部門的把關不嚴格，是我們程式設計師無盡噩夢的來源。

樣式字串的組成及剖析並沒有現成的函式可以叫用，所幸並不太難，即將登場的範例程式「WallPaper Changer / Pattern Viewer」就具有樣式字串剖析的能力，我將它獨立出來為 *ParsePatternString* 程序，如果你需要的是樣式字串組成功能，依樣畫葫蘆反推回去便成。

桌布式樣設定

幸好，桌布不只能夠呆板地擺在畫面中央，我們可以自由選擇它的擺設方法：可以原尺

寸大小擺在畫面中央、可以重覆排列堆滿整個畫面、可以放大至全螢幕大小，還有較不為人知的座標指定法，可以指定桌布擺放的位置。雖然式樣種類不是太多，總算聊勝於無，比永遠只能以原尺寸置於畫面中央的情況好多了。

從表 6-33 中可以發現，除了桌布圖形檔名及 *SPI_SETDESKWALLPAPER* 動作代碼外，*SystemParametersInfo* 函式並不需要額外的參數。原來，桌布擺置的樣式必須先自行寫入登錄資料庫，當透過 *SystemParametersInfo* 函式設定桌布時，它就會根據這些設定值來擺置桌布：

表 6-35 / 與桌布有關的登錄資料庫設定（位於 HKEY_CURRENT_USER\Control Panel\Desktop 機碼）

鍵值名稱	含意
<i>WallPaper</i>	桌布檔名，必須是 BMP 格式影像檔。
<i>TileWallPaper</i>	若為 0，將桌布置於桌面中央；若為 1，以桌布填滿桌面。
<i>WallPaperStyle</i>	若為 0，將桌布以原尺寸大小置中顯示； 若為 1，將桌布填滿桌面，與 <i>TileWallPaper</i> 數值為 1 時相同； 若為 2，則將桌布放大至畫面大小，必須注意若原圖比例與螢幕比例不相符，圖形將會失真。
<i>WallPaperOriginX</i>	當 <i>TileWallPaper</i> 及 <i>WallPaperStyle</i> 皆為 0 時，此數值表示放置圖形的左上角 X 軸座標。
<i>WallPaperOriginY</i>	當 <i>TileWallPaper</i> 及 <i>WallPaperStyle</i> 皆為 0 時，此數值表示放置圖形的左上角 Y 軸座標。

Tips

上表中，*TileWallPaper* 設定值優先等級較 *WallPaperStyle* 設定值為高。換句話說，當 *TileWallPaper* 設定數值為 1 時，無論 *WallPaperStyle* 設定值為何，桌布一定是排列填滿桌面的。為了簡化程式邏輯，我個人的習慣是永遠將 *TileWallPaper* 設定為 0，桌布式樣完全由 *WallPaperStyle* 設定值決定。

設定桌布

設定桌布之前，記得先填好登錄資料庫，再呼叫 *SystemParametersInfo*：

```
#0001 // 首先，先在登錄資料庫寫入桌布樣式
#0002 TRegIniFile* r = new TRegIniFile("\\Control Panel");
#0003 try {
#0004     // 我的習慣：永遠將 TileWallPaper 設為零
#0005     r->WriteInteger("Desktop", "TileWallPaper", 0);
#0006
#0007     // 將 WallpaperStyle 設為適當值
#0008     // 0 = Center
#0009     // 1 = Tile
#0010     // 2 = Fit to screen
#0011     r->WriteInteger("Desktop", "WallPaperStyle", Wallpaper_Style);
#0012 } __finally {
#0013     delete r;
#0014 }
#0015
#0016 // 正式更換桌布
#0017 SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, FileName.c_str(),
#0018     SPIF_UPDATEINIFILE | SPIF_SENDWININICHANGE);
```

Tips

細心的你可能已經發現這回首次使用 *SPIF_SENDWININICHANGE* 常數。根據實際測試，在某些平臺的某些狀況下，若不加上此常數，桌布更換之後並不會正常地顯現出來，所以更換桌布時，請務必加上 *SPIF_SENDWININICHANGE* 常數。

範例程式 – WallPaper Changer / Pattern Viewer

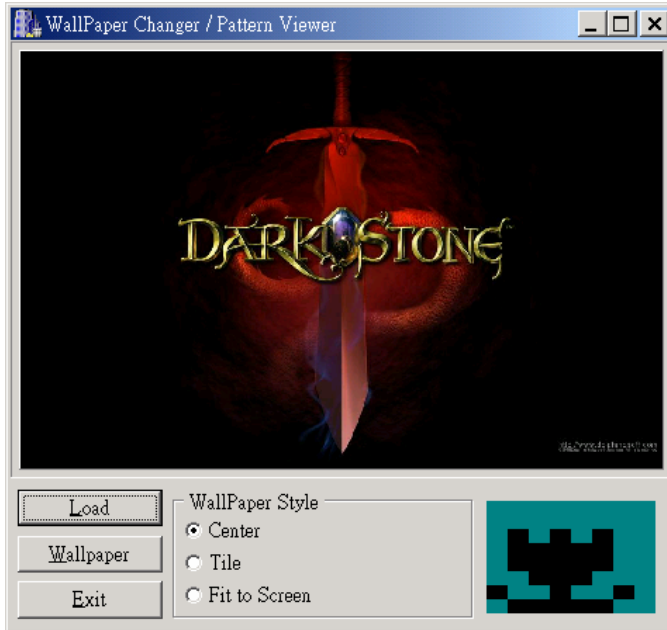


圖 6-36 / WallPaper Changer / Pattern Viewer 執行畫面

WallPaper Changer / Pattern Viewer 範例程式只有簡單的三項功能：一是載入 BMP 圖形檔預視；二是根據桌布式樣設定桌布；最後，它可將目前使用中的填圖樣式以 8 x 8 圖形顯示在視窗右下角。

桌布自動更換軟體

更換桌布是如此地簡單，但其受限於 BMP 影像檔格式的特性，總使桌布愛好者十分苦惱。因此許多程式員自行發展管理／定時更換桌布的軟體，支援各式各樣的圖形檔格式，我的作品 XDesktop 及 Arcata Pet 公司的 WWPlus 即是其中的佼佼者。

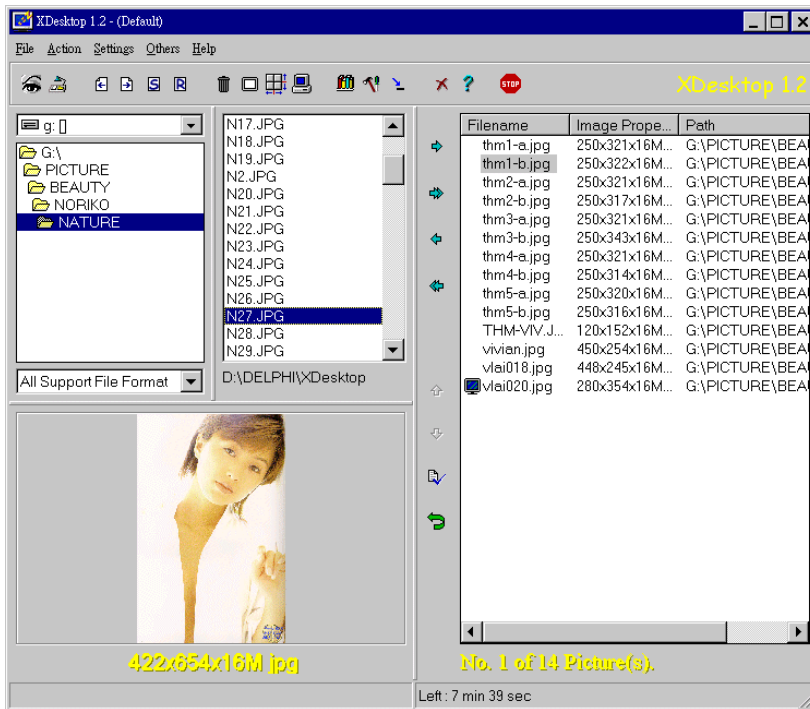


圖 6-37 / 筆者自製的 XDesktop 桌布自動更換工具

它們是如何做到支援各式影像檔格式的呢？其實很簡單，當使用者指定非 BMP 檔案格式的影像檔為桌布時，此類程式就會暗中呼叫圖形轉換函式庫將影像檔轉成 BMP 格式，然後再呼叫 *SystemParametersInfo* 函式設定桌布。

至於其它的功能，例如將桌布擺置於螢幕任何位置，或顯示／隱藏桌面圖示，前頭都介

紹過了；利用計時器函式或 VCL 的 *TTimer* 元件也可輕易辦到定時更換的功能，相信看完本文之後，這類軟體對你已毫無秘密可言。

有趣的 PaintDesktop API

Win32 提供 *PaintDesktop* 函式讓應用程式繪製背景桌面及填圖樣式，而且不論視窗的位置及大小，繪製出來的結果皆與該視窗相同區域的背景一致。

範例程式 *PaintDesk* 啥事也不做，唯一的工作是當它收到 *WM_PAINT*、*WM_MOVE*、*WM_SIZE* 等視窗訊息時，就呼叫 *PaintDesktop* 函式繪製視窗本身，所以程式執行起來就像是開個小窗口透視桌面，使部分桌面出來透透氣，十分有趣，如下圖。

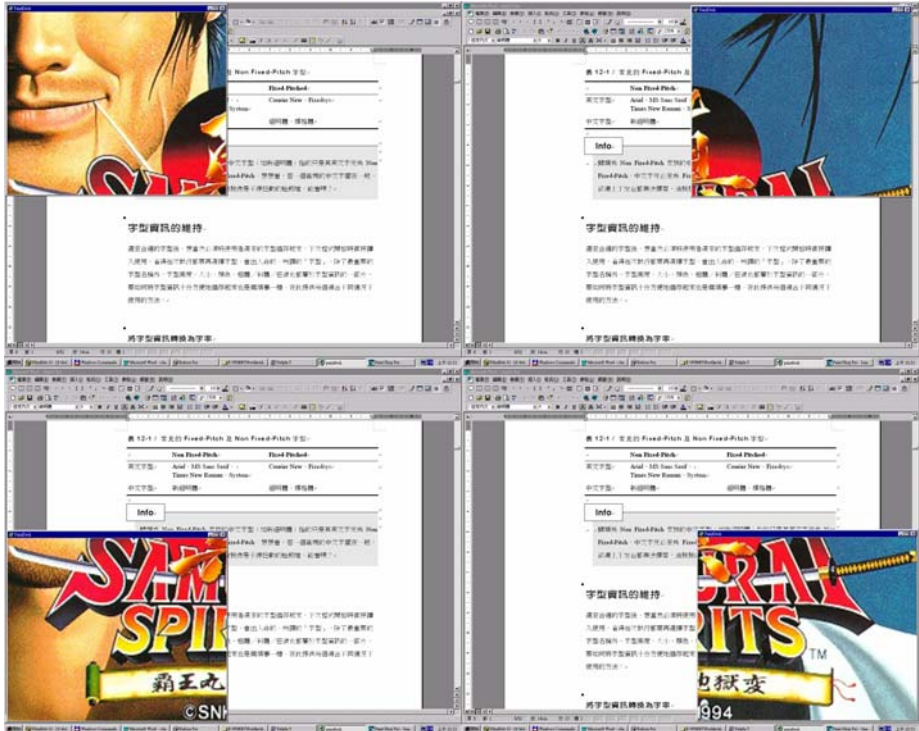


圖 6-38 / 分別將 *PaintDesk* 視窗拉至畫面上四個不同位置所呈現的情形

爲什麼要提供 *PaintDesktop* 這道 API 呢？提供的對象又是誰？

根據技術文件，它是爲 shell 程式所設計的，但是 shell 程式會在什麼情況下使用 *PaintDesktop* 函式呢？我百思不得其解。列出 shell 程式—EXPLORER.EXE 的 import table，可得知它的確使用了 USER32.DLL 提供的這道 *PaintDesktop* 函式，在瞭解使用目的前，暫且將它當成是系統提供給 EXPLORER.EXE 的特別服務好了。

不過，*PaintDesktop* 函式正巧就在前一章的 Desktop Illusionist 程式中派上用場呢！我利用它背地裡繪製桌面圖形，並在上頭繪製文字後，再一口氣貼到桌面，這是 double-buffering 貼圖技巧，以防止畫面閃動。

回到 XTML

至此爲止，所有撰寫佈景主題工具所需的技術資料及相關知識皆已齊備，剩下的只是程式碼的撰寫。我不打算列出程式碼一一講解，讓你看著我的程式碼，根據我的程式流程來走，還不如告訴你所有的規格及技術，你自己想一套程式流程。這只是實作，在別人的程式碼中，你應該試著吸收他人的經驗、程式撰寫風格、規劃能力及你尚未瞭解的技術，剩下的邏輯、流程、撰寫能力，應該自己來，這是別人無法給你的。若你對 XTML 的實作有任何疑慮，原始碼就在書附光碟裡。Use the source, Luke !!

預視功能

預視畫面是如何做出的呢？答案可能嚇你一跳—完完全全以 *TCanvas* 類別及 GDI 函式畫出來的。

雖然 Windows 提供完整的視窗繪製能力，不過它只能依照目前系統設定值來繪出視窗，若需要繪製與目前系統設定值不同的畫面時，就必須自己來。舉凡你在預視畫面中所看到的每個視窗、每個按鈕、每條線段，都必須先計算好座標，再根據欲繪製的系統設定

值調整比例尺及座標，最後才畫到上面。

舉個例子好了，光是「使用中視窗」（請參考圖 6-39、6-40）捲軸列上的捲軸按鈕，就得使用這段程式碼來繪製：

```
#0001 void __fastcall TMainForm::DrawScrollButton(TCanvas* ACanvas, TRect
#0002   ARect, int Kind)
#0003 {
#0004   TRect RT = ARect;
#0005
#0006   Frame3D(ACanvas, RT, FMyColors[ButtonFace],
#0007     FMyColors[ButtonDkShadow], 1);
#0008   Frame3D(ACanvas, RT, FMyColors[ButtonHighlight],
#0009     FMyColors[ButtonShadow], 1);
#0010   ACanvas->Brush->Color = FMyColors[ButtonFace];
#0011   ACanvas->FillRect(RT);
#0012
#0013   ACanvas->Brush->Color = FMyColors[ButtonDkShadow];
#0014   int W = RT.Right - RT.Left;
#0015   int H = RT.Bottom - RT.Top;
#0016
#0017   // 0: up arrow 1: down arrow
#0018   Windows::TPoint points[3];
#0019   switch (Kind) {
#0020     case 0:
#0021       points[0] = Point(RT.Left + W * 0.16, RT.Top + H * 0.65);
#0022       points[1] = Point(RT.Left + W * 0.74, RT.Top + H * 0.65);
#0023       points[2] = Point(RT.Left + W * 0.46, RT.Top + H * 0.25);
#0024       ACanvas->Polygon(points, 2);
#0025       break;
#0026
#0027     case 1:
#0028       points[0] = Point(RT.Left + W * 0.16, RT.Top + H * 0.25);
#0029       points[1] = Point(RT.Left + W * 0.74, RT.Top + H * 0.25);
#0030       points[2] = Point(RT.Left + W * 0.46, RT.Top + H * 0.65);
#0031       ACanvas->Polygon(points, 2);
#0032       break;
#0033   }
#0034 }
```

0006 ~ 0009 列繪出兩個顏色不同的陰影方框，營造出按鈕的立體效果。0010 ~ 0011 列以 `COLOR_BTNFACE` 系統顏色塗滿按鈕。0024 及 0031 列呼叫 `TCanvas::Polygon` 函式分別繪出上面的捲軸按鈕及下面的捲軸按鈕。

很辛苦吧，光是預視畫面的繪製工程大約就佔了四百多行，好在我總是能用 *TCanvas* 類別時就盡量用，*TCanvas* 類別力有未逮時再直接呼叫 *GDI* 函式，否則至少要多寫一倍的程式碼呢。

成果大觀



圖 6-39 / 可以讀取目前設定值

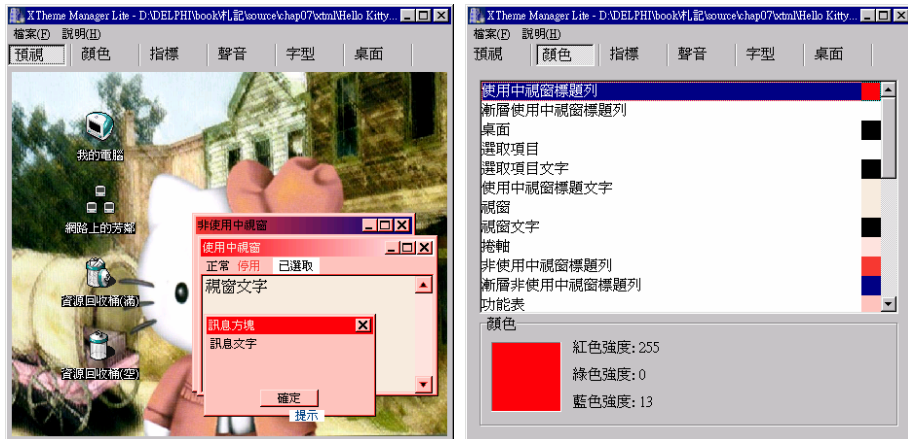


圖 6-40 / 預視畫面及「顏色」頁面

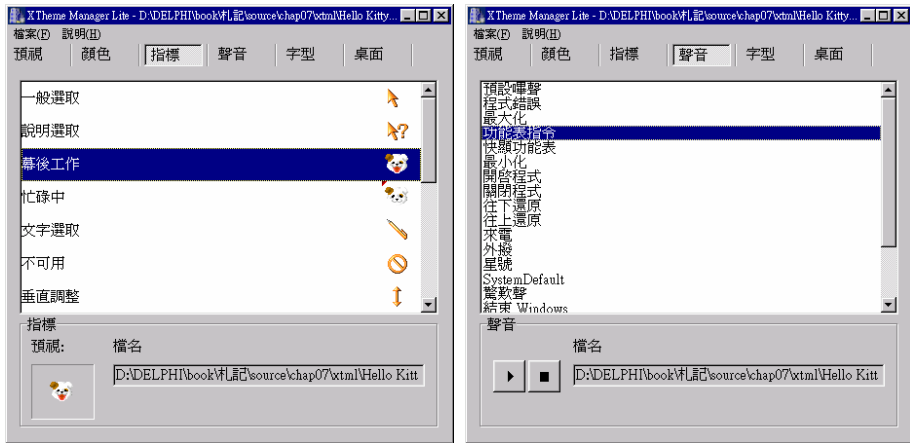


圖 6-41 / 「指標」及「聲音」頁面

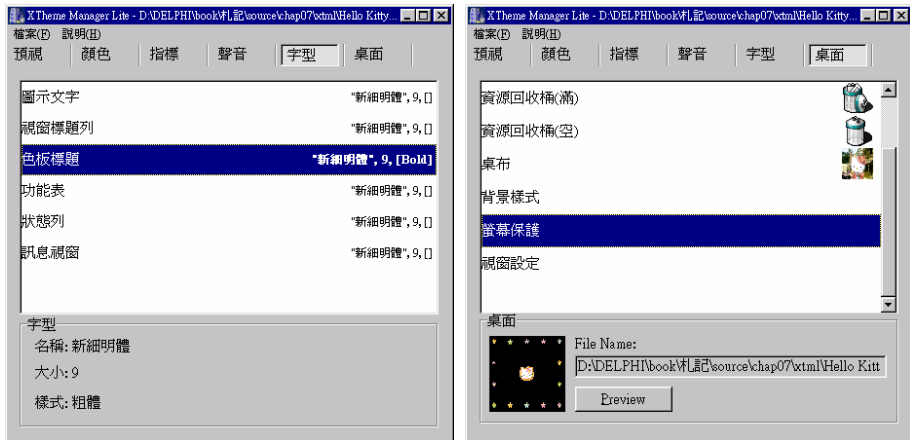


圖 6-42 / 「字型」及「桌面」頁面，「桌面」頁面正預視著螢幕保護程式

第七章

螢幕保護？我用計劃表！

螢幕保護如果老是放些煙火、碎形或是傑克蘿絲，
那真是太無趣太喪志了！
有為青年應該不放過任何一個砥礪提醒自己的機會，
瞧我把計劃表放進螢幕保護程式。



每年的二月及九月，是新學期到來的季節。不能說是習慣，但在每個學期初，我總會志氣昂揚地訂下洋洋灑灑一大張的計劃表，雖然不確定能否徹底實踐，至少新學期總要有新希望嘛！更何況，古有明訓：「學者必先立志，志立則心定，心定則事成」，先賢都這麼說了，所以先立個計劃表準沒錯。:P

經過一番深思熟慮、再三推敲，埋首 UltraEdit 老半天後，好不容易完成本學期的十大理想、十大目標及十大戒律。嗯！越看越滿意，開心極了。突然有股衝動，想將它印下來裱個裱加個框掛在牆上天天欣賞！

呃，不行，那貼滿房間的菜菜子、深田恭子還有貞子的海報怎麼辦？我一張也捨不得撕下來呢，真是痛苦的抉擇，讓人左右為難，苦不堪言啊！躊躇之際，螢幕保護程式突然悄悄啟動，看到畫面上變幻萬千的幾何圖形，靈光一現，有了，螢幕保護程式不正是個絕佳的計劃表展示場？

接下來，只要找個具備展示純文字能力，功能不需太過花俏，能夠讓使用者清楚閱讀文字的螢幕保護程式就成了。可惜的是，找遍幾個較大的軟體下載站台，就是尋不著這類型的螢幕保護程式，大部分的螢幕保護程式都以圖形、影像、動畫及聲音為主，極盡花俏之能事，因此符合上述要求的清淡口味螢幕保護程式反而付之闕如...

沒關係，具備程式設計能力的好處就是，找不到合適的軟體時，不打緊，自己動手寫一個便是。除了成品可以百分之百符合個人的需求外，還可充分享受 DIY 的樂趣。近年來國內電腦界越來越流行組裝電腦 DIY 的趨勢，誰說只有硬體可以 DIY？也許門檻高了些，但軟體 DIY 不但不必額外花錢，還是更有趣、更有成就感的活動呢。

知而後行

到底「知難行易」與「知易行難」哪個合理？反正總有兩派說法，公說公有理，婆說婆有理，我們就不打破沙鍋問到底。重點是，不論知行難易，必須先知而後能行，才能知行合一、心手相應、手腦並用，進而格物致知、數往知來...哇！緩下來，講到哪去了...

嗯...請將目光移向右邊，噢，不對，再下面一點，^，再稍微左邊一點就行了...沒錯，就是這兒，看到了嗎？在你前方的正是今天的主角－螢幕保護程式。

螢幕保護程式的構成

雙擊控制台的「顯示器」元件，或在桌面上單按滑鼠右鍵選擇「內容」選項，都可以開啓「顯示器內容」對話盒。通常我們經由其中的「螢幕保護裝置」頁次來選擇螢幕保護程式、設定「等候啟動時間」、「密碼保護」等選項。下拉選單中有許多螢幕保護程式可供選擇，或多或少，視電腦上安裝的螢幕保護程式數量而定。按下【設定】按鈕可開啓螢幕保護本身的設定視窗，按下【預覽】鈕立即啓動螢幕保護程式，而在對話盒中央的小螢幕裡，可以看到運作中的縮小版螢幕保護程式，如下圖：



圖 7-1 / 螢幕保護裝置頁次，預覽畫面正上演著「小青蛙螢幕保護程式」

安裝螢幕保護程式的方式是，將螢幕保護程式檔案（副檔名為 .SCR）置於 Windows 目錄或系統目錄¹，下一次開啓「顯示器內容」對話盒時，它就會自動出現在螢幕保護裝置的下拉選單，供使用者選用。

¹ 系統目錄指的是利用 *GetSystemDirectory* API 函式所取得的路徑。

這個 .SCR 檔究竟是什麼玩意？這麼大牌，也不掛個號說一聲，只管擺在 Windows 目錄或系統目錄，讓「顯示器內容」對話盒自己來尋找。關於這個傢伙，讓我提一份報告來說明：

- 它是不折不扣的執行檔（.EXE），只不過副檔名改為 .SCR。
- 它的視窗是一個底色全黑、最大化的無邊框視窗，所以執行時能涵蓋整個螢幕，像是畫面上什麼東西都沒有似的。
- 執行時，若收到任何滑鼠、鍵盤或特定視窗訊息時，自動結束程式。
- 它必須能夠分析命令參數，根據參數指示進入「螢幕保護」、「功能設定」或「預視」三種模式。

只要擁有以上這些特性，任何 Windows 程式都可以搖身一變成為螢幕保護程式。

相關的系統登錄設定

位於登錄資料庫的 HKEY_CURRENT_USER\Control Panel\Desktop 機碼下，有四個螢幕保護裝置相關設定：

表 7-2 / 登錄資料庫內的螢幕保護相關設定

鍵值名稱	含意
ScreenSaveActive	是否啟動螢幕保護功能。
ScreenSaverIsSecure ²	是否啟動密碼保護。
ScreenSaveTimeOut	等待啟動時間，單位為秒。系統在這段時間內若無任何鍵盤或滑鼠訊息，就會執行螢幕保護程式。
SCRNSAVE.EXE ³	螢幕保護程式所在路徑。修改這個值可使你的螢幕保護程式擺在任何地方，不一定要在 Windows 或系統目錄下。

² 在 Windows 95/98 下，此字串值名稱為 ScreenSaveUsePassword。

³ 在 Windows 95/98 下，此設定值存在於 SYSTEM.INI 的“BOOT”區段。

由此處可看出，雖然「顯示器內容」對話盒只列出置於 Windows 目錄或系統目錄下的 .SCR 檔案，但是若直接修改此 *SCRNSAVE.EXE* 鍵值，其實可將螢幕保護程式指向位於任何目錄下的 .SCR 檔案。

禁！螢幕保護退散

對於一些需要花長時間來運作、運算的軟體，例如磁碟重組、圖形繪製或者自動展示軟體等等，雖然沒有使用者的操作及互動，但並不希望每每執行個三五分鐘後，就有螢幕保護程式不識好歹地蹦出來。也許這是要擺在資訊展會場供民眾參觀的展示程式，也許使用者想要有事沒事瞄一下螢幕，看看進度如何啊，是否出現錯誤訊息啦等等，螢幕保護程式雖然有趣，但是在重要的場合上就應該乖乖看家，別出來搗蛋，否則很容易讓人抓狂的。

不過沒關係，若程式有這樣的需求，只要攔截處理一道視窗訊息，就可以防止螢幕保護程式的揭竿而起。作業系統在許多時候，例如關閉、移動視窗、按下【ALT - TAB】鍵切換工作，叫出「工作管理員」、啟動螢幕保護程式時，都會送出 *WM_SYSCOMMAND* 視窗訊息給目前的前景視窗。

WM_SYSCOMMAND 為「請求型」的視窗訊息，言下之意，表示前景視窗掌有該事件能否發生的絕對權力。如果你默許該事件發生，那麼什麼事都不必做，當訊息層層傳遞至 *DefWindowProc* 時，被請求的事件（例如啟動螢幕保護程式）就會發生；如果想投否決票，只要將訊息攔截下來，不要讓 *DefWindowProc* 函式知道此消息（就像攔截成績單那樣），該事件就不會發生。訊息截下後，記得將訊息結構的 *Result* 欄位設為零，表示該訊息已經處理完畢。

我們可以利用此項特性，攔截傳送給 form 的 *WM_SYSCOMMAND* 視窗訊息。當 *TWMSysCommand* 訊息結構的 *CmdType* 欄位與 *0xFFFF0* 進行 AND 運算後的結果等於 *SC_SCREENSAVE* 時，表示螢幕保護程式即將啟動，此時先將 *Msg.Result* 設為 0，表示此訊息已經完成所有處理動作，然後直接跳離，不依照正常程序將訊息交給父代類別的

訊息處理函式。

Info

由於 *CmdType* 欄位的最低四個位元另有用途（Windows 內部使用），所以必須先與 0xFFFF 數值進行 AND 運算後才能與動作代碼判斷比較。

如下做法，就可防止螢幕保護程式的啟動：

```
#0001 class TForm1 : public TForm
#0002 {
#0003 private:// User declarations
#0004     void __fastcall WMSysCommand(TWMSysCommand& Message);
#0005 BEGIN_MESSAGE_MAP
#0006     VCL_MESSAGE_HANDLER(WM_SYSCOMMAND, TWMSysCommand, WMSysCommand);
#0007 END_MESSAGE_MAP(TForm);
#0008 };
#0009
#0010 void __fastcall TForm1::WMSysCommand(TWMSysCommand& Message)
#0011 {
#0012     if (Message.CmdType & 0xffff == SC_SCREENSAVE)
#0013         Message.Result = 0;
#0014     else
#0015         TForm::Dispatch(&Message);
#0016 }
```

除了螢幕保護程式啟動事件，還有許多事件都是依賴 *WM_SYSCOMMAND* 視窗訊息來運作的，是可以好好利用的訊息哦。

不過需要注意的是，攔截 *WM_SYSCOMMAND* 訊息來阻止螢幕保護程式啟動的方法有個超級大罩門，即當程式的視窗不是前景視窗時就無效，因為 *WM_SYSCOMMAND* 視窗訊息只送給前景視窗。

如果你希望即使程式在背景工作時，還是能夠防止螢幕保護程式的啟動，那麼至少有下列兩種方法可以做到：

1. 呼叫 *SystemParametersInfo* API 函式，傳入 *SPI_SETSCREENSAVEACTIVE* 代碼，將螢幕保護裝置暫時取消。

2. 掛上 system-wide 的 *WH_CBT* hook。當 hook 函式收到 *HCBT_SYSCOMMAND* 代碼時，表示某個視窗即將收到 *WM_SYSCOMMAND* 訊息；經由其 *CmdType* 欄位，可以將啟動螢幕保護的事件攔下，只要不呼叫 *CallNextHookEx* 函式繼續傳遞事件，就可以防止螢幕保護程式啟動。

呵呵，又提到 hook 了！在 Win32 下，若希望進行什麼奇怪的舉動，不是原本系統支援的作法，那麼就來個 hook 吧，通常能夠幫助我們順利地搞定問題，它真是搞怪程式員的超級法寶！

啟動螢幕保護

前頭討論了好久的 *WM_SYSCOMMAND* 訊息，簡言之，只要將 *WM_SYSCOMMAND* 訊息攔截下來，不讓它送至 *DefWindowProc* 函式手中，螢幕保護程式就不會啟動。

那麼，反過來說，若要主動地使螢幕保護運作，最簡單的方式就是：將帶有 *SC_SCREENSAVE* 代碼的 *WM_SYSCOMMAND* 訊息送給 *DefWindowProc* 函式，就像這樣：

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
    DefWindowProc(Handle, WM_SYSCOMMAND, SC_SCREENSAVE, 0);
}
```

哈哈，很暴力吧。完全不給任何行程攔截螢幕保護啟動事件的機會：直接請 *DefWindowProc* 函式立即啟動螢幕保護程式。以這種做法來說，連前頭介紹的 *WH_CBT* hook 都無法攔截下來，唯一阻止這種行為的做法⁴ 只有將登錄資料庫裡的 *ScreenSaveActive* 鍵值設為零才行。

如果你崇尚和平，希望在事情真正發生前還有挽救的餘地，那麼，送自己一個 *WM_SYSCOMMAND* 訊息會是比较優雅的做法：

⁴ 這是指正常狀況而言。若是遇到超級搞怪駭客，把你的 *DefWindowProc* 函式偷天換日，過濾螢幕保護啟動事件，也不是不可能的事。

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
    Perform(WM_SYSCOMMAND, SC_SCREENSAVE, 0);
}
```

Perform 函式會將視窗訊息送給 form 本身，接著 *WM_SYSCOMMAND* 訊息就會和平常一樣，傳遞給 form 本身的視窗函式，經過 VCL 複雜的訊息繞送機制，若一直沒有被攔截下來，最後會交給 *DefWindowProc* 函式，啟動螢幕保護程式。

實作預備課程

尚未接觸 Windows 程式設計之前，我對螢幕保護程式的印象一直是，嗯，十分有趣，應該也是十分困難的主題。

大部分的螢幕保護程式會以黑色鋪滿整個畫面，執行時無論將畫面塗抹得七彩繽紛、五光十色亦或一團糟，只要動一下滑鼠或敲一下鍵盤，一切馬上回復原狀，靜悄悄地，好似什麼事都沒發生過。

當時的想法很直覺，螢幕保護程式啟動時要負責將整個桌面儲存起來，結束前再將桌面復原，玩具玩完要物歸原位嘛，從小媽媽就這樣教我的。

直到學習 Windows 程式設計，知道訊息／事件驅動架構後，才認識 *WM_PAINT* 視窗訊息：系統會適時地將 *WM_PAINT* 訊息傳遞至需要重繪的視窗，由每一個視窗負責自己的繪製動作。這時的想法變成：螢幕保護程式並不擁有任何視窗，它只是任意地、完全不負責任地將桌面佔為己有，在上頭胡天胡地，反正螢幕保護結束時，系統會將 *WM_PAINT* 訊息傳遞給桌面上所有視窗，讓它們全部重繪，恢復畫面原有的模樣。

這個想法已經十分接近標準答案，只差那麼一點點...

原來是個窗

其實，在畫面上看到的，只是個放大到全螢幕、黑底、無邊框且將滑鼠指標隱藏起來的視窗。

要將 form 變成所謂的「放大到全螢幕、黑底、無邊框」的視窗，請按下【F11】叫出物件檢視器，為它做個小小的整型手術：

- 將 *BorderStyle* 屬性設為 *bsNone*，使視窗沒有標題列及邊框。
- 將 *Color* 屬性設為 *clBlack*，成為黑底視窗。
- 將 *WindowState* 屬性設為 *wsMaximized*，使視窗建立後自動最大化。

如果此時按下【F9】執行程式，螢幕立刻變為全黑，幾乎就像是一個「空白螢幕」的螢幕保護程式，只差在沒有將滑鼠指標隱藏起來。不過呢，面子重要，裏子更可貴，目前它還只是個空心湯糰，成為夠格的螢幕保護程式視窗之前，內部還有得大肆翻修呢。

事件處理

若要成為螢幕保護程式的主視窗，必須再攔截下列事件，進行對應動作：

表 7-3 / 需要攔截的事件及對應動作

事件名稱	對應動作
<i>OnActivate</i>	啟動螢幕保護運作。
<i>OnKeyDown</i> 、 <i>OnMouseDown</i>	中止螢幕保護運作，並關閉視窗，結束程式。
<i>OnMouseMove</i>	檢查使用者是否移動滑鼠，若移動超過容許程度，同樣地中止螢幕保護運作，結束程式。

OnActivate 事件發生在視窗即將變成前景視窗前，我們利用這個事件來啟動螢幕保護運作，進入螢幕保護模式。

OnKeyDown 及 *OnMouseDown* 事件分別代表使用者按下鍵盤或滑鼠鍵，這表示使用者想要離開螢幕保護狀態，所以必須立刻結束程式。

比較特別的是 *OnMouseMove* 事件的處理動作，每當滑鼠指標移動時，*OnMouseMove* 事件就會觸發。在此可以多花點功夫，讓滑鼠變得不那麼「靈敏」，才不會一個不小心碰到電腦桌，滑鼠稍稍「滑」了一下，就讓螢幕保護程式結束了。你可以自訂滑鼠的「靈敏」度判定規則，比如說一秒鐘內觸發 *OnMouseMove* 事件超過六次或是一秒鐘內滑鼠指標位移量超過十個像素時，才判定使用者真的想要結束螢幕保護程式，否則就忽略不理。

訊息攔截

除了上述的事件，螢幕保護程式的主視窗還必須攔截處理下列視窗訊息：

表 7-4 / 需要攔截的視窗訊息及對應動作

視窗訊息	對應動作
<i>WM_SYSCOMMAND</i>	若 <i>CmdType & 0xFFFF0</i> 為 <i>SC_SCREENSAVE</i> (啓動螢幕保護程式請求)，則將此視窗訊息攔截起來，不讓事件發生。
<i>WM_ACTIVATEAPP</i>	若 (<i>TWMActiveApp</i>) <i>Message.Active</i> 為 <i>false</i> ，結束程式。
<i>WM_ACTIVATE</i>	若 (<i>TWMActivate</i>) <i>Message.Active</i> 為 <i>WM_INACTIVE</i> ，表示前景視窗即將更替為其它視窗，此時應關閉程式。
<i>WM_NCACTIVATE</i>	若 (<i>TWMNCActive</i>) <i>Message.Active</i> 為 <i>false</i> ，結束程式。

帶有 *SC_SCREENSAVE* 代碼的 *WM_SYSCOMMAND* 訊息是啓動螢幕保護程式的請求訊息。如果螢幕保護程式由系統啓動執行，那麼在運行過程中，將不會收到帶有 *SC_SCREENSAVE* 代碼的 *WM_SYSCOMMAND* 視窗訊息；但是假使螢幕保護程式是由使用者或程式手動執行 (直接執行 *.SCR* 檔案)，系統不認為進入螢幕保護模式，所以當螢幕保護等候時間一到，就會再送出 *WM_SYSCOMMAND* 訊息來啓動螢幕保護。

所以，螢幕保護程式執行時，若收到這樣的訊息，必須將它攔截，不使 *DefWindowProc* 函式再一次啓動螢幕保護程式，否則執行中的螢幕保護程式將被中止，由系統啓動的使

用者指定的螢幕保護程式取代。

攔截 `WM_ACTIVATEAPP`、`WM_ACTIVATE` 及 `WM_NCACTIVATE` 三個視窗訊息的目的是：當主視窗即將變成非前景視窗時，表示有其它視窗想要取而代之成為前景視窗，螢幕保護程式此時不必再繼續執行，所以立即結束程式。

剖析參數

一個完整的螢幕保護程式必須支援「螢幕保護」、「功能設定」及「預視」三種模式，模式的選擇是由啟動程式時傳入的參數而定，參數及對應模式如下：

表 7-5 / 需要支援的參數及對應模式

參數	對應模式
沒有參數	啟動功能設定對話盒。
<code>/C</code> ⁵	啟動功能設定對話盒。
<code>/P</code> 視窗 handle	預覽螢幕保護，參數為提供預視區域的視窗 handle。
<code>/S</code>	啟動螢幕保護。

運作核心

上述的介面修改、事件處理及訊息攔截等等完全都是制式的動作，每個螢幕保護程式皆大同小異，但是運作核心可就毫無限制、規則可言。你可以在螢幕保護程式視窗上進行任何動作，包括常見的圖形展示、影像撥放或簡單的動畫等等，寫個腦力激盪的益智遊戲、即時的線上瀏覽或是來個電視節目表撥放也行，還有人不甘心電腦的運算能力白白浪費，撰寫以暴力法嘗試密碼核對的螢幕保護程式，並且能夠自動上網回報結果，透過網路聯合成千上萬部電腦一塊破解密碼。簡單地說，螢幕保護程式的行為只受限於程

⁵ Windows NT下為 `/C:視窗handle`，所以判斷參數時要格外小心。

式設計師的創意及程式功力。

在 Windows 3.1 那個年代，由於 Win16 並不支援先佔式多工（preemptive multitasking），因此螢幕保護核心的運作時機通常伴隨 *Application* 物件的 *OnIdle* 事件而起：每當 *OnIdle* 事件觸發時，趕快進行螢幕保護的核心運作，而且不能佔著不放，做了一些事情後馬上離開，才不會妨礙訊息迴圈的處理。另外一個作法是，完全將控制權奪下，在迴圈內進行螢幕保護核心的運作，但是必須同時兼任訊息迴圈的任務，才能夠處理送達螢幕保護程式視窗的視窗訊息。

直到 Win32 的降臨，多執行緒機制隨之出現，螢幕保護程式的撰寫方式也有了革命性的轉變。多執行緒的程式撰寫是那麼地方便，發明電燈後當然不希望天天拿著火把看書囉。所以雖然還是可以使用從前的運作方式，不過我偏好另外建立一個執行緒來負責螢幕保護運作行為—稱之為「螢幕保護運作核心執行緒」，簡稱「核心執行緒」。主執行緒負責接收、回應視窗訊息、核心執行緒的生滅，而核心執行緒只要專心地繪製、更新畫面即可。

預視功能

螢幕保護程式的預視功能並不是必要的支援，就算沒有，螢幕保護也能運作良好。話雖如此，這可關係到使用者對螢幕保護程式的印象分數。想想看，在「顯示器內容」對話盒中，從列示盒拉出一排的螢幕保護程式，十個有八個都支援預視功能，可直接在對話盒上的小螢幕裏看到縮小版的螢幕保護執行狀況，而剩下那兩個不支援預視的傢伙，硬要使用者按下【預視】鈕才能看到，我想這兩個螢幕保護程式會立刻給人「不專業」、「草率」、「實作不完全」的感覺。

若要提供預視功能，撰寫螢幕保護模式的運作核心時要格外小心：**不要假設螢幕保護程式總在全螢幕大小的視窗中運作**。欲進行螢幕保護程式的預視時，「顯示器內容」對話盒會執行該程式的執行檔，並傳入兩個參數：第一個參數為「/P」，第二個參數會是個十進位的視窗 handle 字串。接下來，螢幕保護程式必須在這個小小的預視視窗上運作，

視窗位置及大小都不一定，唯一能依賴的資訊就只有視窗 `handle` 而已。

當然囉，若你的螢幕保護程式不適合在小小的預視視窗內顯示，也可以簡單地在預視視窗上秀出一張圖形檔，或展示程式的版權說明，或畫幾條快速晃動的彩色曲線，至少讓使用者感覺你的心意已到。

功能設定

通常我們不會將螢幕保護程式使用的參數值寫死，總是希望在程式撰寫不會太過複雜的前提下，多留給使用者一些自由選擇的空間。以各版本 Windows 皆有的「留言設定」（或 Scrolling Marquee）螢幕保護程式為例，雖然功能十分簡單，單純地將一行字串由右而左不停捲動而已；但是它十分細緻地提供了四種設定功能，幾乎可讓使用者自訂全部的參數。請見下圖，留言位置、背景色彩、捲動速度、顯示字型、顯示文字，通通可以設定：



圖 7-6 / 「留言設定」螢幕保護程式的設定對話盒

撰寫螢幕保護程式時，盡量不要將參數定死，最好通通可經由設定對話盒來更動這些參數；讓使用者的感覺由「用」螢幕保護程式，轉變為「玩」螢幕保護程式，那麼你的螢幕保護程式就可說是成功了。

螢幕保護程式的功能規劃完畢，並且將運作核心撰寫完成後，整理出所有可供使用者調整的參數或資料，接著就必須設計它的設定對話盒。這可是 C++Builder 的 RAD 特性擅場的時刻，請好好利用 C++Builder 的視覺化設計工具，恣意揮灑，盡情發揮，設計出美觀友善的設定對話盒。

雖然位於同一個程式內，但是「螢幕保護」及「功能設定」兩個模式不可能同時執行。所以，設定對話盒取得的使用者設定參數，無法直接交由螢幕保護運作核心使用。因此設定對話盒必須將使用者的設定值寫入登錄資料庫，下回螢幕保護程式執行時，再從登錄資料庫將設定值讀出，供螢幕保護運作核心使用。

取個響噹噹的好名字

這篇文章並不是教你如何為辛苦撰寫出來的螢幕保護程式命名，我只知道中國人依五行八卦來命名，不曉得軟體要如何依它的生辰八字來取名，市面上也還未出現「軟體命名學」這類的書籍，也許快要有了也說不定。

在「顯示器內容」對話盒中，我們可以利用 `Listbox` 控制項來選取螢幕保護程式。如果你的螢幕保護程式檔案叫做“GOOSE.SCR”，那麼你將會在下拉列表中看到一行有點蠢的「GOOSE」。不像別的螢幕保護程式，不但包含中文，而且還拉得很長，例如「立體飛行物體（OpenGL）」、「小青蛙螢幕保護程式」等等，可是，在 Windows 目錄及系統目錄下怎麼找也找不到“立體飛行物體（OpenGL）.SCR”檔和“小青蛙螢幕保護程式.SCR”呀，它們是如何辦到的？

難道，「顯示器內容」對話盒所列出的螢幕保護程式名稱不一定得是螢幕保護程式的檔名嗎？答案是，算你好運，撞對了。

「顯示器內容」對話盒會從螢幕保護程式取出編號為 1 的資源字串作為它的名稱，如果沒有編號為 1 的資源字串，才使用程式的檔名為名稱。

瞭解招數後，見招拆招，想法子在執行檔中加入編號為 1 的資源字串就行了！

建立資源字串的方法很多，我偏好的作法是：先用文書編輯器建立 RC 資源描述檔，再使用 C++Builder 附的 Resource Compiler（BRCC32.EXE）將它編譯為 RES 檔。RC 檔是這樣寫的：

SCRSAVER.RC

```
STRINGTABLE
{
  1, "It's my first screen saver, and I wrote it with Borland C++Builder !!"
}
```

接著打開命令列視窗，鍵入：

```
BRCC32 SCRSAVER.RC
```

即可將它編譯為 RES 檔。

RES 檔並不是最終目的，最終的目的是將它與執行檔連結在一塊。所以別忘了，打開專案的任一個單元，加入 {\$R SCRSAVER.RES} 資源編譯指示，告訴連結程式說，請將 SCRSAVER.RES 的內容放進結果檔案的資源區段。

Info

雖然微軟提供的技術文件是這樣寫的，而且經過觀察，大部分的螢幕保護程式也都含有指示螢幕保護程式名稱的編號 1 資源字串，但是，Windows 95/98 竟然完全不理會這回事，直接以檔名做為螢幕保護程式名稱。

可能是由於長檔名的支援，認為不需要再這樣做，但是，支援長檔名的 Windows NT 4 也認識此編號為 1 的資源字串呀！嘿，又是實作、文件不協調的一例。

XEssay Screen Saver

原本的想法十分簡單，只想寫一個螢幕保護程式，能夠靜態地顯示自己的計劃表就好。不過程式寫著寫著，除了將靜態的顯示功能升級為動態的捲動功能外，發現它其實也十分適合顯示短文、歌詞、名片檔等等，再加上我平日搜集的一拖拉庫名言佳話及勵志箴言，正好將螢幕保護程式當成模特兒走秀用的伸展台。只要螢幕保護程式一啟動，就將一篇篇短文自動展示出來，並以捲動、拉頁、閉幕效果來進行，不但達成保護顯示器的目的，也讓我每天複習秀在畫面上的那些佳話、歌詞、情書⁶，實用極了。

這就是“XEssay Screen Saver”的命名原因：Essay 者，短文、小品、隨筆也；而前頭的大寫字母 X 是我的軟體命名習慣，可以解釋為 Xshadow（筆者過去在 BBS 上慣用的代號），也可以解釋為 eXcellent、eXtra、eXtreme、酷斃了、帥透了、棒極了之意...，呵呵。

程式功能規劃

我為 XEssay Screen Saver 的功能規劃如下：

- 以捲動方式顯示純文字檔內容。
- 可以設定多個文字檔，以亂數輪流交替顯示。
- 所有的參數皆可由使用者設定，如捲動速度、移動間隔時間、過場延遲時間以及顯示的字型、顏色、大小等等。
- 支援預視功能，能在任何視窗內進行螢幕保護運作。

⁶ 情書時時拿出來複習，可以增進男女朋友之間的感情哦。

Main Form 的設計

這個程式的 main form 介面可說是宇宙超級無敵沒力的簡單，只要將 *Color* 屬性設為 *clBlack*、*WindowState* 屬性設為 *wsMaximized*、再把 *BorderStyle* 屬性改為 *bsNone*，介面設計動作就全部完工，一個元件都用不著。

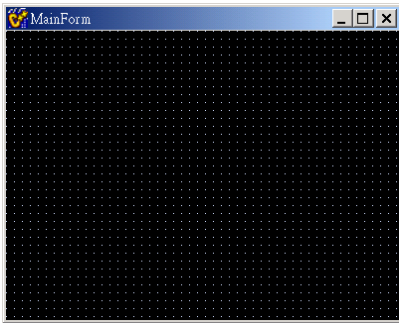


圖 7-7 / Main form 的設計時期畫面

參照前面的說明，請分別為 main form 撰寫 *OnKeyDown*、*OnMouseDown*、*OnMouseMove* 事件處理函式，並且攔截 *WM_SYSCOMMAND*、*WM_ACTIVATE*、*WM_NCACTIVATE* 及 *WM_ACTIVATEAPP* 視窗訊息，在使用者有任何動作或視窗即將成為非前景視窗時結束程式。

由於我們採用執行緒運作方式，所以最重要的是，分別在 *OnActivate* 及 *OnClose* 事件觸發時，啟動及中止螢幕保護核心執行緒。還有，進入螢幕保護模式時，記得將滑鼠指標隱藏起來，要不然進行螢幕保護模式後，滑鼠指標還留在畫面上，太不像話了！

```
#0001 void __fastcall TMainForm::FormActivate(TObject *Sender)
#0002 {
#0003     ShowCursor(false); // 將滑鼠指標隱藏起來
#0004
#0005     FDrawThread = new TDrawThread(Handle); // 建立核心執行緒
#0006 }
#0007
#0008 void __fastcall TMainForm::FormClose(TObject *Sender,
#0009     TCloseAction &Action)
#0010 {
```

```
#0011  FDrawThread->Terminate(); // 中止核心執行緒
#0012  }
```

建立核心執行緒

TThread 類別的使用方式較為特別，與其它類別不同的是，一定要由它衍生出新的類別才能使用，因此我建立一個 *TDrawThread* 作為螢幕保護程式的核心執行緒：

```
#0001  class TDrawThread : public TThread
#0002  {
#0003  private:
#0004      HWND FWnd; // 螢幕保護運作視窗
#0005
#0006      AnsiString FCurEssayFile; // 目前的短文檔名
#0007      TString* FEssay; // 目前的短文內容
#0008
#0009      int FWindowWidth, FWindowHeight, FWindowMiddleY;
#0010      TRect FWindowRect; // 視窗的矩形區域
#0011
#0012      int FMoveSpeed; // 移動速度
#0013      int FDelayTime, FClearDelayTime, FChangeDelayTime;
#0014
#0015      void __fastcall ChooseAndLoadEssay(); // 選擇並載入一篇短文
#0016      void __fastcall DisplayEssay(); // 播放目前短文
#0017  protected:
#0018      void __fastcall Execute();
#0019  public:
#0020      __fastcall TDrawThread(HWND Wnd);
#0021      __fastcall virtual ~TDrawThread();
#0022  };
```

0020 列的建構函式接收一個視窗 *handle* 參數，代表螢幕保護動作運行的目標視窗，先將它指派給 *FWnd* 變數，此後執行緒所有的動作皆根據 *FWnd* 視窗來進行。0009 ~ 0010 列宣告儲存目標視窗寬度、高度及矩形範圍等資訊的 *FWindowWidth*、*FWindowHeight*、*FWindowRect* 等變數，這些資訊完全由 *FWnd* 視窗取得，而短文的顯示、捲動的座標和範圍又完全遵照這些資訊。最終的結果是，不論視窗的位置及大小為何，螢幕保護行為都可以在其中正常地運作。

另外還有一點要注意的事，若視窗不是全螢幕大小時，除了座標要按照視窗位置及尺寸

來修改，文字的字型大小及移動速度也必須按照視窗比例縮小，運作步調才能保持一致。

DisplayEssay 函式內，進行繪製動作以前，就有這麼一段程式碼：

```
// 按照視窗比例縮小字型及移動速度
if (FWindowHeight != Screen->Height) {
    Canvas->Font->Size = MAX(
        (FWindowHeight * Canvas->Font->Size) / Screen->Height, 1);
    FMoveSpeed = MAX(
        (FWindowHeight * FMoveSpeed) / Screen->Height, 1);
}
```

所以，即使在「顯示器內容」對話盒的預視視窗上，還是可以看到 *XEssay Saver* 十分正常地運作哦，只是原本斗大的字變成一丁點，除了比例縮小外，所有的行為都相同。

TThread 後代類別都必須改寫 *Execute* 函式，將執行緒所有的運作程式置於其中。它通常包含一個不斷循環執行的迴圈，只是時時檢查 *Terminated* 屬性，若 *Terminated* 屬性值為 *true* 時，就跳離迴圈。

```
#0001 void __fastcall TDrawThread::Execute()
#0002 {
#0003     do {
#0004         ChooseAndLoadEssay(); // 選擇並載入一篇短文
#0005         DisplayEssay(); // 播放目前短文
#0006     } while (!Terminated);
#0007 }
```

不斷檢查 *Terminated* 屬性的原因是，當其它執行緒呼叫某個執行緒物件的 *Terminate* 函式，欲結束該執行緒時，*Terminate* 函式並不直接強行結束執行緒，它唯一的動作只是將 *Terminated* 屬性設為 *true* 而已。所以我們必須自行檢查 *Terminated* 屬性，一旦為 *true* 就應該中斷執行，結束執行緒。

0004 列的 *ChooseAndLoadEssay* 函式正如其名，每次呼叫時就從短文檔案列表選擇並載入一篇與目前短文不同的短文檔案，並將所選定的短文內容載入到 *TStringList* 物件 *FEssay*。而負責播放短文的 *DisplayEssay* 函式將進行如下動作：

1. 由視窗下緣緩緩出現昇起一列文字。

2. 如果是第一列，該列文字上升到一定高度時，就停住不動；如果非第一列，則該列文字上升至即將撞上之前文字時，也停住不動。
3. 畫面上所有文字依文字正常的行進速度緩緩往上捲動一行。
4. 回到步驟一，接著浮出下一列文字，直到全部文字播放完畢。
5. 整篇短文顯示結束後，延遲一段時間（使用者自訂）。
6. 黑色線條緩緩行進，由左右兩邊往視窗中央，毫不留情地將文字「刷」掉。

這就是每篇短文的播放動作，你可以在本章最末頁的執行畫面中看到這些步驟的行進過程。在 *DisplayEssay* 函式中特別要注意的是，只要進入迴圈中，可能佔用相當時間的動作，都必須時時查看 *Terminated* 屬性，確認 *Terminate* 函式是否已被呼叫。

短文的捲動由 *ScrollWindow* API 函式達成：傳入視窗 *handle*、欲捲動的矩形區域及 X、Y 軸位移量，*ScrollWindow* 函式就可以快速無閃爍地捲動視窗影像，使用起來十分簡單，而且完全符合我們的需求。例如，步驟三「畫面上所有文字依文字正常的行進速度緩緩往上捲動一行」只要下列數行程式碼就辦得到：

```
#0001 // 畫面上所有文字依文字正常的行進速度緩緩往上捲動一行
#0002 while (R.Top > ScrollFinalY && !Terminated) {
#0003     ScrollWindow(FWnd, 0, -FMoveSpeed, &R, &FWindowRect);
#0004     R.Top -= FMoveSpeed;
#0005     R.Bottom -= FMoveSpeed;
#0006     if (FDelayTime) Sleep(FDelayTime);
#0007 }
```

提供預視功能

Windows 對於螢幕保護程式的預視支援有如下要求：

螢幕保護程式**不應該**在預視視窗上直接運行；螢幕保護程式**應該**自行建立一個和預視視窗位置、大小一模一樣的子視窗，讓它在上頭運行。

因此，獲得預視視窗 *handle* 後，呼叫 *GetWindowRect* 取得預視視窗的座標及大小，接著

按照此矩形區域建立一個新視窗，使它成為預視視窗的子視窗，而螢幕保護程式就在新視窗上頭運作。

所以說，「螢幕保護程式必須在預視視窗上運作」這句話大有問題，應該說：「螢幕保護程式就必須在一個和預視視窗大小、位置皆相同的子視窗上運作，使之看起來就像在預視視窗上運作」才對。

避免混淆起見，接下來我以「預視視窗」稱呼由命令列參數傳入的視窗 `handle`，以「自建預視視窗」表示我們自行建立的新視窗，這才是螢幕保護真正運作其上的視窗。

註冊視窗類別

建立自建預視視窗之前，必須為它註冊新的視窗類別，登錄它的視窗風格、類別名稱及視窗函式等等。這是一個十分簡單的視窗類別，除了視窗函式、instance `handle` 及類別名稱外，其它欄位都直接填零即可。

```
#0001 WndClass.style = 0;
#0002 WndClass.lpfnWndProc = (WNDPROC)PreviewWndProc;
#0003 WndClass.cbClsExtra = 0;
#0004 WndClass.cbWndExtra = 0;
#0005 WndClass.hIcon = 0;
#0006 WndClass.hInstance = HInstance;
#0007 WndClass.hCursor = 0;
#0008 WndClass.hbrBackground = 0;
#0009 WndClass.lpszMenuName = NULL;
#0010 WndClass.lpszClassName = "XEssayScreenSaverPreview";
#0011
#0012 RegisterClass(&WndClass);
```

將 `WndClass` 結構欄位填好之後，呼叫 `RegisterClass` API 函式註冊此視窗類別。

而此視窗類別的視窗函式 – `PreviewWndProc`，也是最簡單的視窗函式了。它只須處理 `WM_DESTROY` 訊息：收到 `WM_DESTROY` 訊息時，呼叫 `PostQuitMessage` 函式結束程式，其它的通通留給 `DefWindowProc` 函式來處理。`PreviewWndProc` 函式的程式碼只有短短數行：

```
#0001 LRESULT _stdcall PreviewWndProc(HWND Wnd, UINT Msg, Longint WPARAM,  
#0002     Longint LPARAM)  
#0003 {  
#0004     if (Msg == WM_DESTROY) {  
#0005         PostQuitMessage(0);  
#0006         return 0;  
#0007     }  
#0008  
#0009     return DefWindowProc(Wnd, Msg, WPARAM, LPARAM);  
#0010 }
```

Talk

你可能覺得麻煩，就爲了這道 *WM_DESTROY* 訊息，就得特別撰寫一個視窗函式。爲什麼不讓 *WM_DESTROY* 訊息的預設處理行爲變成 *PreviewWndProc* 函式所做的，呼叫 *PostQuitMessage* API 函式以結束程式呢？

這是因爲，雖然每個程式的主視窗（我指的是，一旦關閉就代表程式結束的那個視窗）都必須和 *PreviewWndProc* 函式一樣，收到 *WM_DESTROY* 訊息時就結束程式。但是程式裡其餘的所有視窗並不都是如此呀，除了主視窗，摧毀這些視窗都不能代表程式結束，要不然，在程式中隨便摧毀一個按鈕元件就讓程式結束了，多可怕呀！

所以，視窗的預設行爲並不處理 *WM_DESTROY* 訊息，有必要（擔任程式主視窗的視窗）時，再由自訂的視窗函式來處理它。

建立視窗

鯛魚燒的模子買回來後要幹嘛？當然是用它做出一堆好吃的鯛魚燒來享用。那麼，上頭建立 *XEssayScreenSaverPreview* 視窗類別後，馬上就用它來建立自建預視視窗，好讓螢護保護預視動作早點進行。

呼叫 *CreateWindow* 函式建立視窗時，有幾點注意事項：

- 將新視窗的座標及大小設定爲與預視視窗完全相同。

- 加入 `WS_CHILD`、`WS_DISABLED` 及 `WS_VISIBLE` 視窗風格旗標。
- 指定預視視窗為其父視窗。

務必以預視視窗擔任自建預視視窗的父視窗，同時含入 `WS_CHILD` 視窗風格旗標。原因是，若遵照這個規則來建立視窗，螢幕保護預視功能的呼叫者便可以下列呼叫來取得自建預視視窗 `handle`：

```
自建預視視窗 handle = GetWindow(預視視窗 handle, GWL_CHILD);
```

這個式子的成立十分重要，因為螢幕保護預視功能的啟動程式（如「顯示器」控制台元件）必須藉此取得自建預視視窗，並在預視結束時摧毀此視窗，螢幕保護程式才能夠順利結束。如果不這麼做，你就會看到即使預視功能早已結束，但是螢幕保護程式全然不知，還會繼續在畫面上不斷地繪製，十分不合作。

Tips

就是因為這個原因（螢幕保護預視功能啟動程式假設自建預視視窗為預視視窗第一個子視窗），所以我們才不能直接在預視視窗上進行螢幕保護預視動作，不然預視功能的支援就簡單多了。

附帶一提的是，在 Windows NT 下，即使直接在預視視窗上進行預視動作，系統還是有辦法成功地將被啟動的螢幕保護程式結束掉；但是在 Windows 95/98 上，錯誤的設計將輕易地使系統當掉。這兩種作業系統外觀雖然類似，但骨子裡可是差很多的呢！

遵照上述的遊戲規則，以下列程序建立自建預視視窗：

```
#0001 GetWindowRect(ParamHandle, &R);  
#0002  
#0003 MyWnd = CreateWindow("XEssayScreenSaverPreview", "XEssaySaver",  
#0004     WS_CHILD | WS_DISABLED | WS_VISIBLE, 0, 0,  
#0005     R.Right - R.Left, R.Bottom - R.Top,  
#0006     ParamHandle, 0, HInstance, NULL);
```

進行預視動作

此時東風既備，仗豈有不打的道理。建立自建預視視窗後，建立 *TDrawThread*，傳入由 *CreateWindow* 函式傳回自建預視視窗 *handle*，然後進入自備的訊息迴圈工作。

```
#0001 // 建立運作核心執行緒
#0002 TDrawThread* DrawThread = new TDrawThread(MyWnd);
#0003 // 訊息迴圈
#0004 while (GetMessage(&Msg, 0, 0, 0)) {
#0005     TranslateMessage(&Msg);
#0006     DispatchMessage(&Msg);
#0007 }
#0008 DrawThread->Terminate(); // 結束運作核心執行緒
```

此時，訊息迴圈不斷地取得並分派訊息，而運作核心執行緒正賣力地在我們的自建預視視窗中運行。這個狀態會一直持續著，直到預視功能啟動者（通常是「顯示器」控制台元件）呼叫 *DestroyWindow* API 函式摧毀自建預視視窗，預視動作才告結束。

DestroyWindow 呼叫會遞送 *WM_DESTROY* 訊息給 *PreviewWndProc* 視窗函式，而 *PreviewWndProc* 函式的反應是呼叫 *PostQuitMessage* 函式，將 *WM_QUIT* 訊息置於訊息佇列。當 *GetMessage* 遇上 *WM_QUIT* 訊息時，就會回傳 *False*，離開訊息迴圈，結束程式。

函式離去前，雖然明知此時程式將要關閉，但還是盡量培養用後歸還的好習慣，呼叫 *UnregisterClass* 函式將先前註冊的 *XEssayScreenSaverPreview* 視窗類別取消。

巧妙地使用 TForm

經過上述的說明，我想你已經十分明白自建預視視窗建立及預視功能提供的所有細節。也許你會懷疑，VCL 辦不到這些事嗎？為什麼非得呼叫一大堆 API 函式，如此麻煩地註冊視窗類別、建立視窗、撰寫視窗函式及視窗迴圈咧？

其實，只要巧妙地使用，VCL 的 *TForm* 視窗類別也可以達成相同的效果。方才我們先由 SDK 的角度看過一遍，現在再介紹使用 *TForm* 類別的解決方案，相信可幫助你更瞭解 *TForm* 的特性及行爲。並且，比較看看，只要能夠徹底瞭解及靈活使用 VCL，它可爲我們省下多少功夫？

首先，建立另外一個取代自建預視視窗的 form，命名爲 *TPreviewForm*。請將它的 *Color* 屬性設爲 *clBlack*，使預視畫面呈現黑色。

TPreviewForm 的叫用方式不同於正常的使用途徑，整段預視功能的程式碼列表如下：

```
#0001 TPreviewForm* frm = new TPreviewForm(ParamHandle);
#0002 try {
#0003     TRect R;
#0004
#0005     // 設定座標
#0006     GetWindowRect(ParamHandle, &R);
#0007     OffsetRect(&R, -R.Left, -R.Top);
#0008     frm->BoundsRect = R;
#0009
#0010     // 建立運作核心執行緒
#0011     TDrawThread* thrd = new TDrawThread(frm->Handle);
#0012     frm->ShowModal(); // 秀出 PreviewForm 視窗
#0013     thrd->Terminate(); // 結束運作核心執行緒
#0014 } __finally {
#0015     delete frm;
#0016 }
```

0001 列建立 *TPreviewForm* 時，傳入預視視窗 handle，這使得新建立的 *TPreviewForm* 成爲預視視窗的子視窗；緊接著，0006 ~ 0008 列設定 *TPreviewForm* 視窗的位置及大小。

0011 列建立 *TDrawThread* 運作核心執行緒，傳入 *TPreviewForm* 視窗 handle，讓螢幕保護動作於 *TPreviewForm* 視窗上運行。0012 列是運作關鍵，*ShowModal* 函式內部含有訊息迴圈，因此主執行緒進入 *ShowModal* 函式後，不會立即返回，等到 *TPreviewForm* 視窗關閉後才返回；而 *TDrawThread* 執行緒一旦產生，就不斷地努力運作。

最後，當 *TPreviewForm* 視窗關閉，自 *ShowModal* 函式返回後，主執行緒接著呼叫作核心執行緒的 *Terminate* 函式，中止預視動作的運行。

重點有兩處，一是接受視窗 handle 的建構函式，它可使 VCL 控制項以系統內任何視窗為父視窗，不限制為 VCL 元件；二是 *TCustomForm::ShowModal* 函式，它內含訊息迴圈，於 form 關閉時返回，在妥善的運用下，可以省下撰寫訊息迴圈及視窗函式的麻煩。

你瞧，只要能夠掌握架構精良的 VCL，就算無關介面設計的機制，VCL 總是可將事情簡化，為我們省下撰寫大量 SDK 程式碼的時間，很棒吧。

設定對話盒

設定對話盒的功能無它，主要就是提供設定螢幕保護程式參數的使用者介面，並將使用者設定的結果寫入登錄資料庫。直到螢幕保護程式進行螢幕保護動作時，才將這些設定值自登錄資料庫中讀出使用。XEssay Saver 設定對話盒的設計時期畫面如下，兩個頁面分別設定短文的顯示參數及短文檔案列表：

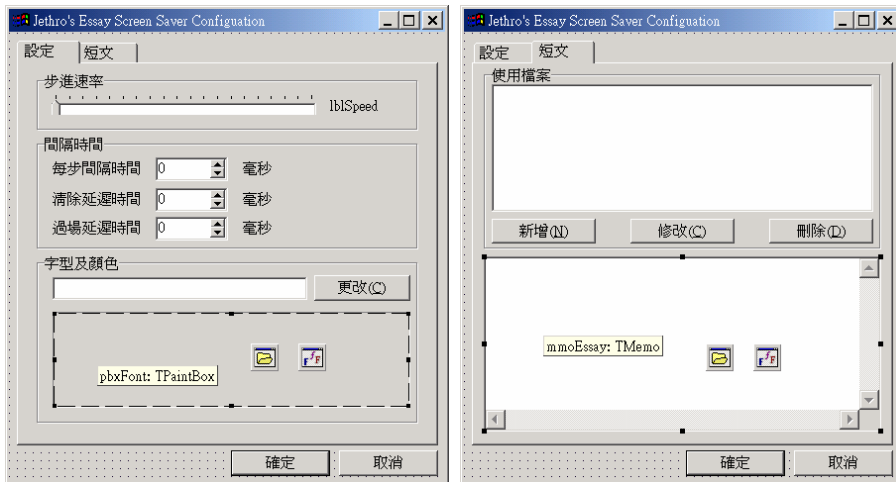


圖 7-8 / 設定對話盒的設計時期畫面

剖析命令列參數

先定義螢幕保護程式的執行模式列舉型別：

```
// 螢幕保護、功能設定、預視三種模式
enum TSaverMode {smSaver, smConfig, smPreview};
```

我將所有剖析命令列參數的動作置於專案原始碼中，由 *ParseParameter* 函式負責：

```
#0001 // 將第二個參數轉成整數存入 ParamHandle 變數
#0002 bool SetParamHandle()
#0003 {
#0004     bool r = ParamCount() > 1;
#0005     try {
#0006         if (r) ParamHandle = (HWND)StrToInt(ParamStr(2));
#0007     } catch (...) {
#0008         return false;
#0009     }
#0010     return true;
#0011 }
#0012
#0013 void ParseParameter()
#0014 {
#0015     AnsiString Param;
#0016
#0017     ParentHandle = 0;
#0018
#0019     if (!ParamCount()) { // 沒有參數 => 設定模式
#0020         SaverMode = smConfig;
#0021         return;
#0022     }
#0023
#0024     Param = UpperCase(ParamStr(1));
#0025
#0026     // 若第一個字元不是英文字元，就刪除掉
#0027     if (Param[1] < 'A' || Param[1] > 'Z') Param.Delete(1, 1);
#0028
#0029     switch (Param[1]) {
#0030     case 'C':
#0031         SaverMode = smConfig;
#0032         ParentHandle = GetForegroundWindow();
#0033         break;
#0034
```

```
#0035     case 'P': if (SetParamHandle()) // 若成功取得預視視窗 handle
#0036         SaverMode = smPreview; // 進入預視模式
#0037         break;
#0038
#0039     default:
#0040         SaverMode = smSaver; // 否則進入螢幕保護模式
#0041         break;
#0042     }
#0043 }
```

接著修改專案原始碼中的主函式，使得程式執行後，立即呼叫 *ParseParameter* 函式剖析參數，得知欲進入的模式，存入 *SaverMode* 全域變數。再根據分析結果分別進入不同的執行模式：

```
#0001 WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0002 {
#0003     // 首先分析參數，根據參數修改 SaverMode
#0004     ParseParameter();
#0005
#0006     TConfigDlg* ConfigDlg;
#0007
#0008     try
#0009     {
#0010         switch (SaverMode) {
#0011             case smSaver:
#0012                 Application->Initialize();
#0013                 Application->CreateForm(__classid(TMainForm), &MainForm);
#0014                 Application->Run();
#0015                 break;
#0016
#0017             case smConfig:
#0018                 ConfigDlg = new TConfigDlg(NULL);
#0019                 try {
#0020                     ConfigDlg->ShowModal();
#0021                 } __finally {
#0022                     delete ConfigDlg;
#0023                 }
#0024                 break;
#0025
#0026             case smPreview:
#0027                 PreviewSaver();
#0028                 break;
#0029         }
#0030     } catch (Exception &exception)
#0031     {
```



```
#0032     Application->ShowException(&exception);
#0033     }
#0034     return 0;
#0035 }
```

三種執行模式分別進入三個完全不同的流程：

□ 螢幕保護模式

由 *Application* 物件建立 *TMainForm* 視窗，因為它會成為 main form，與標準 VCL 程式的運行架構相同。*MainForm* 產生後自動最大化，並啟動運作核心執行緒來進行螢幕保護模式運作。

□ 功能設定模式

建立 *TConfigForm* 視窗，在此完全跳離平日的流程，完全不理會 *Application* 物件。我直接呼叫 *TConfigForm* 類別的建構函式來建立新的 *TConfigForm* 視窗，接著呼叫 *ShowModal* 函式，因為 *ShowModal* 函式自備訊息迴圈，所以直到使用者關閉設定對話盒後，*ShowModal* 函式才會返回，程式也隨之結束。

□ 預視模式

呼叫 *PreviewSaver* 函式。*PreviewSaver* 函式會先建立自建預視視窗，再產生運作核心執行緒，在自己的預視視窗上進行螢幕保護動作。

你瞧，我們已經將主函式改得面目全非，三種狀況中只有一種是按照正常的流程，其它兩種皆自行建立視窗且自備訊息迴圈，所以不必依賴 *Application* 物件。打開「Project Options」對話盒，只要其中的「Forms」頁面顯示正常，看得到專案擁有的全部 forms，就表示這樣的修改沒有問題。

編譯及執行

雖然目的十分特別，不過螢幕保護程式還是正常的 Win32 執行檔，所以可以直接在 C++Builder 整合環境內測試、除錯、執行。

螢幕保護程式執行檔的副檔名為 .SCR，因此必須告訴連結程式，製作出 .SCR 副檔名的執行檔。選擇【Project / Options】功能表選項叫出對話盒，切換至「Application」頁面，

在 Target file extension 欄位填入 “.SCR”。

測試螢幕保護程式時，必須同時指定執行模式。你可以選取【Run / Parameters】選項叫出對話盒，指定執行時傳入的參數，或者直接在程式碼中指定 *SaverMode* 變數，都是十分方便的作法。

安裝

一切就緒，只差臨門一腳了。即使是臨門一腳，也得小心點踢，否則為山九仞，功虧一簣，豈不痛哉？

安裝方法十分簡單，將編譯連結完成的 .SCR 檔拷貝至 Windows 目錄即可。下面是個簡單的批次檔：

INSTALL.BAT

```
copy xessay.scr c:\winnt
```

將 .SCR 檔複製過去，下回打開「顯示器」控制台元件時，就可以看到 XEssay Saver 螢幕保護程式也在列表中參一腳了。

當然囉，若你的對象是廣大的使用者，寫個周詳點、帶有圖形使用者介面的安裝程式，或者直接利用安裝程式套件打包起來，會比較好的作法。

成果賞玩

下圖是在「顯示器內容」對話盒中，選擇 XEssay Saver 後的情況。你可以在預視視窗中看到一些小白點，那些白點可不是亂畫的，而是 XEssay Saver 的預視功能正在運作呢。



圖 7-9 / 「顯示器內容」對話盒正預視著 XEssay Saver 螢幕保護程式

按下【設定】鈕，蹦出 XEssay Saver 的功能設定對話盒：

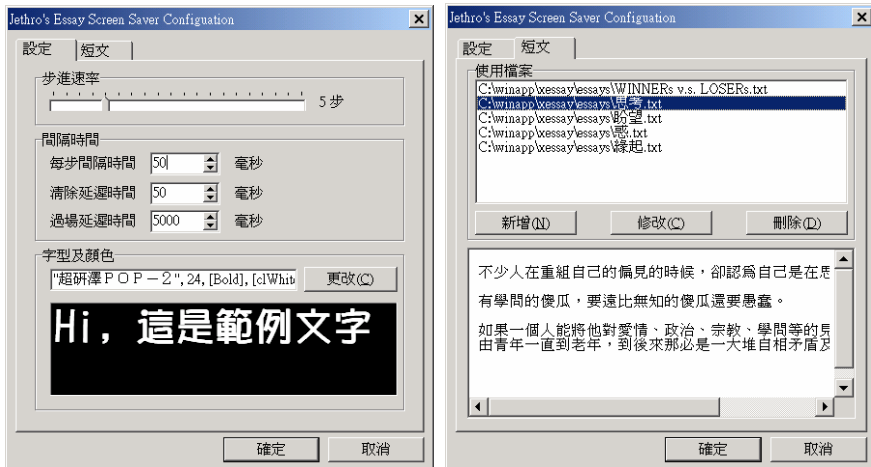


圖 7-10 / XEssay Saver 設定對話盒

按下【預覽】鈕，XEssay Saver 立即進入螢幕保護模式。從以下四張圖你可以看到，本書的章節概要，一行一行緩緩地從視窗下方升起，全部出現後，再由視窗兩端將視窗清除，準備進入下一篇短文的 show time。

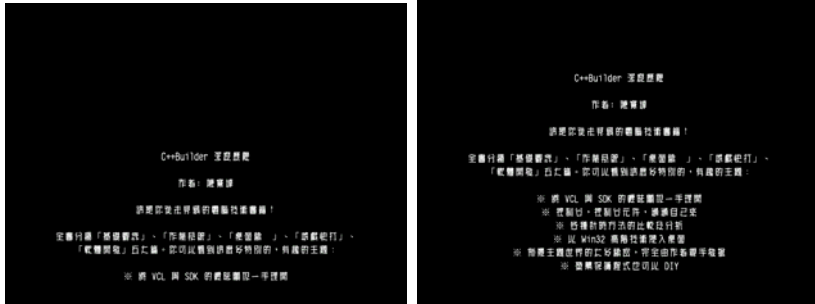


圖 7-11 / XEssay Saver 螢幕保護程式執行畫面

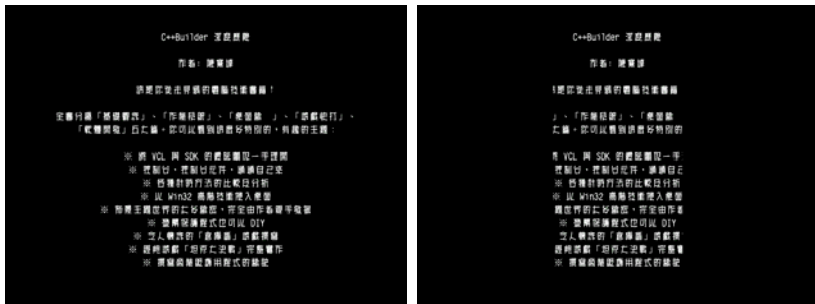


圖 7-12 / XEssay Saver 螢幕保護程式執行畫面

看過完整的 XEssay Saver 螢幕保護程式實例，你發現了嗎？所有的螢幕保護程式都採用幾乎相同的架構。因此，若想要自行製作一個與眾不同的螢幕保護程式，只要更改兩個地方：一個是設定對話盒的介面，另一個是重新撰寫 *TDrawThread* 運作核心執行緒程式碼。只要就著正確的架構，在其中修修改改塗塗寫寫，你也可以很快地寫出一個聲光俱佳的螢幕保護程式。

第四篇

遊戲快打



第八章

足球番

許多人應該都很熟悉「倉庫番」這個小遊戲，
在各個平臺及各式電視遊樂器上皆曾見它的蹤跡。

我的回味方式比較奇特，不是好好地玩玩它，
而是以 C++Builder 從頭到尾撰寫一套足球版的「足球番」。



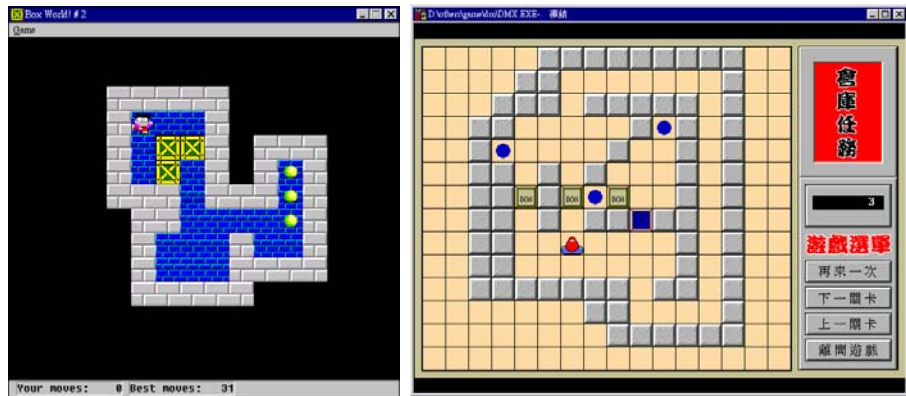


圖 8-1 / 兩個從 Internet 上取得的倉庫番遊戲

上圖是兩個從 Internet 上取得的倉庫番遊戲，皆是國人的作品。前者似乎以純 Windows SDK 開發，後者以 QBasic 4.5 英文版撰寫。

你一定也很熟悉這個小遊戲吧！在各個平臺及各式電視遊樂器上皆曾見它的蹤跡。規則十分簡單，只要操縱主角將箱子一一推至地圖上的標示點，就算過關。雖然規則就這麼簡單，但難度可不低哨，在細心安排設計下，有些關卡總要讓人傷透腦筋，嘗試個十來次甚至百來次才能過關¹。

這個再簡單也不過的老遊戲，正是今天的主題。我們將以 C++Builder 從頭到尾撰寫一套足球版的「倉庫番」，就稱它為「足球番」好了。它的特性十分符合本章的教學目的：

1. 畫面處理 GDI 即可輕鬆解決，背景不用捲動，也不算即時遊戲。
2. 除遊戲主程式外，必須另有地圖／關卡編輯器的搭配才算完整，另外還配有圖庫編輯器，這兩支工具可推廣使用於許多益智、角色扮演甚至動作遊戲上頭。
3. 圖形使用不多，但是遊戲本身耐玩，不是只靠華麗畫面吸引人的遊戲類型。

先來訂立我們這套足球番的功能及特色：

¹ 好吧，我承認，上頭那兩個遊戲我就有些關卡過不了。:P

1. 遊戲規則與倉庫番皆相同，將所有目的地形利用覆蓋用物品掩住即可過關。
2. 地圖大小即是可視範圍，因此不用支援地圖捲動。
3. 所有圖片，包括角色圖形尺寸大小皆相同。
4. 角色的移動以圖片大小為單位，因此沒有小碎步移動所帶來的碰撞處理問題。
5. 所有圖片，包括角色圖形皆是外掛方式，可以在不修改程式碼的情形下更動圖形。
6. 採用關卡制度，可供使用者自行編輯關卡。
7. 具有動作重播功能。

看起來，除了有些新鮮有點無聊的重播功能外，只不過是另一套簡單的倉庫番類型遊戲，連圖形、角色移動等方式都還一切從簡哩。希望你不會太失望，功夫從易處練，雖然一點都不炫，至少是自己寫出來的嘛。

系統規劃

全套遊戲除了主程式外，另設計兩支工具程式－圖庫編輯器及地圖編輯器。因為重覆性高，有些遊戲將地圖編輯器及主程式合併放在同一支程式內，但是，分開為兩支程式有下列優點：

- 減少程式設計的複雜度。同一支程式提供的功能越多，程式邏輯必定越複雜。
- 不必要的 overhead。地圖編輯功能不一定會提供給 end user，若將這些功能置於主程式中而不去使用，徒然增加檔案大小而已。

好，那表示我們要分別撰寫三支應用程式。在動手之前，別急，讓我們先將系統必要的幾個重要類別切割出來。

TTiles 類別

用來定義一個「圖庫」（Tile Archive），一個圖庫包含多張圖片，畫面上除了角色圖形外的所有圖片都是由圖庫中取得。

因為提供給「地形」及「物品」兩層地圖的圖片屬性完全不同，因此必須為這兩個圖庫分別設計不同的圖庫類別，所以我將 *TTiles* 設定為抽象類別，但做好大部分的工作，如載入／儲存圖庫，圖片管理及繪製圖片等等，再分別由 *TTerrTiles* 及 *TItemTiles* 兩個類別繼承，分別提供設定及讀取圖片屬性的功能。

- 「地形」圖片有「可以通過」及「目的地形」兩種屬性。
- 「物品」圖片有「可以移動」及「覆蓋用物品」兩種屬性。

TMap 類別

用來定義一張地圖，或說，一個關卡的佈局。本遊戲的地圖設計為兩層，底下一層是地形，上面一層是物品，兩層獨立操作／貼圖互不相干，貼圖用的圖片分別由兩個圖庫提供，因此分別需要一個二維陣列來儲存圖片編號。

為求視覺逼真效果及操作編輯方便，現在的地圖往往都不只分為兩層，如 *StarCraft* 就分為五層（請見圖 8-2），*英雄無敵 III* 也至少分為四層（請見圖 8-3），分別是土地、河流、道路及地形物。我們的足球番由於地圖簡單，因此分為地形及物品兩層即足夠。

TMap 類別不但負責關卡的載入及儲存，另外也會儲存角色的初始位置。



圖 8-2 / StarCraft 的地圖編輯器，它有五層地形可供編輯。



圖 8-3 / 英雄無敵 III 的地圖編輯器，它有四層地形可供編輯。

TRole 類別

角色類別，負責角色的圖形載入及繪製，移動本身的位置計算及搬動物品。另外重播功也由此類別自行紀錄移動資訊，再交由主程式來播放。

總共才需五個類別，而且除了 *TTiles* 類別是抽象類別，不用來產生物件，其它四個類別在遊戲中都只要產生一個物件就夠了，為什麼？因為我們同一時間只可能使用一張地圖，一個「地形」圖庫，一個「物件」圖庫以及一個角色。

類別實作

先將類別的功能及需求定義出來，切割清楚後，一一將這些重要類別實作出來，到時候用「兜」的，便可輕易兜出三個程式來了，信不信?:)

首先得先定義一些到處用得著的常數（定義於 *Util.pas*）：

```
#define TILE_WIDTH 32 // 圖片寬度點數
#define TILE_HEIGHT 32 // 圖片高度點數

#define TILE_NUM_X 10 // 畫面橫軸格數
#define TILE_NUM_Y 10 // 畫面縱軸格數

const char* FN_TERR_ARCHIVE "TERR.TIA" // 地形圖庫
const char* FN_ITEM_ARCHIVE "ITEM.TIA" // 物品圖庫
const char* FN_ROLEBITS "ROLEBITS.BMP" // 角色圖檔

const char* FN_MAP_PREFIX "MAP" // 關卡圖檔檔名 (MAP??.DAT)
const char* FN_MAP_EXT ".DAT" // 關卡圖檔副檔名

const char* SIG_MYFILE "Xshadow_Stock" // 圖庫及地圖檔案的檔頭標籤
```

圖片大小為 32 x 32，是十分常見且有效率的大小設定，因為我們的 CPU 通用暫存器寬度也是 32 bit，在進行記憶體區塊搬移時，不會有不符合 *DWORD alignment* 的情況發生。

畫面橫軸及縱軸格數是隨手定義的，在我的 1280 x 1024 解析度畫面下看起來小小的，但

後來才發現 10 x 10 格設計不出複雜的關卡。:P 反正只要將這兒的常數更動，重新編譯主程式及地圖編輯器後，即可以新的畫面大小進行關卡設計及遊戲，覺得地圖範圍過小的讀者請自己修改。:p

SIG_MYFILE 用來作為檔頭標籤，在讀取圖庫及地圖檔案時，先確認檔案開頭有沒有此字串，以確定讀取的是我們自己的檔案，不會有誤讀的情況發生。此外，我還在圖庫的檔頭標籤後頭加上「副檔頭標籤」，用以辨別、確認「地形」及「物品」圖庫。

TTiles 圖庫類別及子類別

TTiles 是第一個實作的類別，這是它的類別宣告（定義於 *TileUnit.h*）：

```
#0001 class TTiles {
#0002 private:
#0003     int FTileNum; // 圖片數量
#0004     Graphics::TBitmap* FBits; // 存放圖庫所有圖片的 bitmap
#0005
#0006     int GetTilePos_Left(int index);
#0007     int GetTilePos_Top(int index);
#0008
#0009     int GetTileNumPerRow();
#0010     int GetTileRowCount();
#0011 protected:
#0012     virtual void SetTileNum(int Value);
#0013
#0014     // 讀取及設定屬性都是虛擬函式，讓後代類別來改寫
#0015     virtual void LoadAttrs(TReader* filer) = 0;
#0016     virtual void WriteAttrs(TWriter* filer) = 0;
#0017 public:
#0018     TTiles();
#0019     virtual ~TTiles();
#0020
#0021     // 載入及儲存圖庫
#0022     void LoadFromFile(AnsiString Filename);
#0023     void SaveToFile(AnsiString Filename);
#0024
#0025     void ImportPicture(AnsiString Filename); // 匯入圖形檔
#0026
#0027     __property int TileNum = {read = FTileNum, write = SetTileNum};
#0028
```

```

#0029 // 每列的圖片數目
#0030 __property int TileNumPerRow = {read = GetTileNumPerRow};
#0031 // 共有幾列圖片
#0032 __property int TileRowCount = {read = GetTileRowCount};
#0033
#0034 __property Graphics::TBitmap* Bitmap = {read = FBits};
#0035
#0036 // 根據圖片編號就可取得圖片在圖形中的位置
#0037 __property int TilePos_Left[int index] = {read = GetTilePos_Left};
#0038 __property int TilePos_Top[int index] = {read = GetTilePos_Top};
#0039 };
#0040
#0041 const int TILE_BITS_NUM_X = 10; // 圖庫中每列圖片的數目
    
```

取巧的圖片儲存方式

圖庫的實作採用了一個很偷懶很取巧的函式：將所有的圖片擺在同一個 bitmap 中，同時定義了一個 `TILE_BITS_NUM_X` 常數，表示圖庫中每列圖片的數目。我將此常數設定為 10，表示圖庫 bitmap 為寬 = 320 點，高 = 圖片數目 / 10 x 32，所以編號為 0..9 的圖片會依序擺在第一列、編號為 10..19 的圖片會擺在第二列等等。

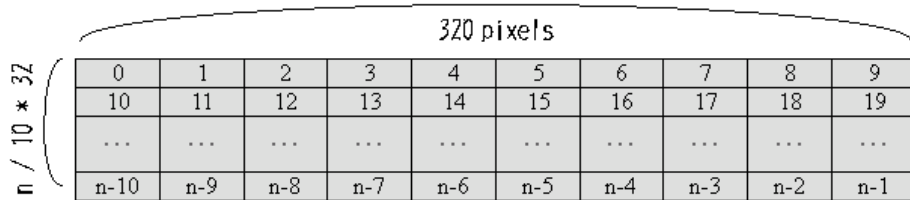


圖 8-4 / 圖庫中圖片的儲存方式

這種偷懶的圖片儲存函式讓設計者在畫好圖片後必須開啓影像編輯軟體，手動將圖片擺在正確的位置上，麻煩極了。例如，我畫好 No.13 圖片，這時必須打開原本儲存圖庫的圖形檔，將它擺在 (96, 32) 的位置上，差一點都不行。呵，我知道你們都會罵我很笨，饒了我吧，下不爲例，爲了程式簡單，我只好出此下策，下回我一定將它改成比較聰明的儲存方式。

爲了這種圖片儲存法，*TTiles* 類別提供 *TilePos_Left* 及 *TilePos_Top* 兩個陣列型態屬性，來讓外界依圖片編號就可取得圖片在圖檔中的 X 軸及 Y 軸位置。這兩個屬性皆屬於一維陣列屬性，使用時必須傳入索引值，而它們對應的 *GetTilePos_Left* 及 *GetTilePos_Top* 兩個函式才會根據此索引值去計算圖片位置。

另外圖片數目也一定爲 *TILE_BITS_NUM_X* 的倍數，是呼叫 *ImportPicture* 函式後根據此圖形檔的長寬大小自動計算而得，*ImportPicture* 函式同時會裁切載入的圖形檔，使圖形檔的寬度變成 *TILE_WIDTH * TILE_BITS_NUM_X*，高度變成 *TILE_HEIGHT* 的整數倍：

```
#0001 void TTiles::ImportPicture(AnsiString Filename)
#0002 {
#0003     FBits->LoadFromFile(Filename);
#0004     TileNum = FBits->Height / TILE_HEIGHT * TILE_BITS_NUM_X;
#0005
#0006     // 將 bitmap 裁減爲所需的大小
#0007     FBits->Width = TILE_WIDTH * TILE_BITS_NUM_X;
#0008     FBits->Height = TileNum / TILE_BITS_NUM_X * TILE_HEIGHT;
#0009 }
```

載入圖檔的 *TTiles.LoadFromFile* 函式使用 VCL 的 *TFileStream* 及 *TReader* 類別來開啓檔案及讀取資料，0018 行還呼叫了 *LoadAttrs* 函式供後代類別 *TTerrTiles* 及 *TItemTiles* 來改寫，讀取它們自己的屬性資料：

```
#0001 void TTiles::LoadFromFile(AnsiString Filename)
#0002 {
#0003     TFileStream* fs;
#0004     TReader* reader;
#0005
#0006     fs = new TFileStream(Filename, fmOpenRead);
#0007     reader = new TReader(fs, 2048);
#0008     try {
#0009         CheckSignature(reader, SIG_MYFILE); // 檢查檔頭標籤
#0010         CheckSignature(reader, typeid(*this).name()); // 檢查副檔頭標籤
#0011
#0012         // 爲了觸發 property 的 SetXXX method
#0013         TileNum = reader->ReadInteger();
#0014         reader->FlushBuffer();
#0015
#0016         // 這是虛擬函式，因爲 TTiles 本身根本沒有定義屬性，
#0017         // 讓後代類別決定該如何讀取屬性
#0018         LoadAttrs(reader);
```

```
#0019
#0020  FBits->LoadFromStream(fs); // 讀入存放所有圖片的圖形
#0021
#0022  // 檢查圖形的長寬不符合標準
#0023  if (FBits->Width < TILE_WIDTH * TILE_BITS_NUM_X)
#0024      throw Exception("Width of tile bitmap is invalid");
#0025
#0026  if (FBits->Height / TILE_HEIGHT * TILE_BITS_NUM_X < FTileNum)
#0027      throw Exception("Height of tile bitmap is invalid");
#0028  } __finally {
#0029      delete reader;
#0030      delete fs;
#0031  }
#0032 }
```

你可以在類別宣告中注意到，*LoadAttrs* 及 *WriteAttrs* 兩個函式不但為虛擬函式，同時還宣告為抽象函式，這是因為 *TTiles* 類別根本沒有圖片屬性的概念，只是宣告好這兩個函式，完全不實作，留待後代改寫使用。

地形及物品圖庫類別

TTiles 類別撰寫完成後，接著就可以從它衍生出談論已久，呼之欲出的 *TTerrTiles* 及 *TItemTiles* 類別了（與 *TTiles* 類別同樣定義於 *TileUnit.h*）。

```
#0001 // 地形的屬性
#0002 typedef enum {taCanPass, taTarget} TTerrAttrElement;
#0003 typedef Set<TTerrAttrElement, taCanPass, taTarget> TTerrAttr;
#0004
#0005 // 物品的屬性
#0006 typedef enum {iaCanMove, iaSource} TItemAttrElement;
#0007 typedef Set<TItemAttrElement, iaCanMove, iaSource> TItemAttr;
#0008
#0009 class TTerrTiles : public TTiles {
#0010 private:
#0011     std::vector<TTerrAttr> FAttrs;
#0012     TTerrAttr GetAttrs(int index);
#0013     void SetAttrs(int index, const TTerrAttr& Value);
#0014 protected:
#0015     virtual void SetTileNum(int Value);
#0016
#0017     virtual void LoadAttrs(TReader* filer);
```



```

#0018     virtual void WriteAttrs(TWriter* filer);
#0019 public:
#0020     __property TTerrAttr Attrs[int index] =
#0021         {read = GetAttrs, write = SetAttrs};
#0022 };
#0023
#0024 class TItemTiles : public TTiles {
#0025 private:
#0026     std::vector<TItemAttr> FAttrs;
#0027     TItemAttr GetAttrs(int index);
#0028     void SetAttrs(int index, const TItemAttr& Value);
#0029 protected:
#0030     virtual void SetTileNum(int Value);
#0031
#0032     virtual void LoadAttrs(TReader* filer);
#0033     virtual void WriteAttrs(TWriter* filer);
#0034 public:
#0035     __property TItemAttr Attrs[int index] =
#0036         {read = GetAttrs, write = SetAttrs};
#0037 };
#0038
#0039 TTerrTiles Terrs;
#0040 TItemTiles Items;

```

第 0003、0007 行是這兩個新類別的重點，兩個圖庫類別分別擁有不同的圖片屬性。0011 及 0026 分別宣告兩個類別用來儲存屬性的容器物件，在此我使用 C++ Standard Library 所提供的 *vector* 類別。

兩個類別都改寫了 *TTiles* 類別的 *SetTileNum* 及 *LoadAttrs*、*WriteAttrs* 函式，目的十分簡單，改寫 *SetTileNum* 函式是爲了在圖片數目改變時同時變更 *FAttrs* 屬性陣列的長度，而 *LoadAttrs*/*WriteAttrs* 則經由 *TFileStream* 物件來讀取及寫入屬性陣列。以 *TTerrTiles* 類別的程式碼爲例：

```

#0001 void TTerrTiles::SetTileNum(int Value)
#0002 {
#0003     TTiles::SetTileNum(Value);
#0004     FAttrs.resize(TileNum);
#0005 }
#0006
#0007 void TTerrTiles::LoadAttrs(TReader* filer)
#0008 {
#0009     for (int i = 0; i < TileNum; i++)
#0010         for (TTerrAttrElement x = taCanPass; x <= taTarget;

```

```

#0011         x = (TTerrAttrElement)(x + 1)) {
#0012         bool b = filer->ReadBoolean();
#0013         if (b) FAttrs[i] = FAttrs[i] << x;
#0014     }
#0015
#0016     filer->FlushBuffer();
#0017 }
#0018
#0019 void TTerrTiles::WriteAttrs(TWriter* filer)
#0020 {
#0021     for (int i = 0; i < TileNum; i++)
#0022         for (TTerrAttrElement x = taCanPass; x <= taTarget;
#0023             x = (TTerrAttrElement)(x + 1))
#0024             filer->WriteBoolean(FAttrs[i].Contains(x));
#0025
#0026     filer->FlushBuffer();
#0027 }

```

儲存屬性的步驟有些麻煩：尋訪每個圖片的屬性，將 *TTerrAttrElement* 集合型態的值儲存起來，但因為 *TTerrAttrElement* 類別本身未提供輸出入介面，也未將內部的資料佈局公開給外界存取（這是當然的），所以我們只能以一個個可能的值來詢問，此集合是否包括某某值，如果是的話，就寫入 *true*，否則寫入 *false*。讀取屬性的步驟當然也一樣囉，讀入一個個布林值，根據它們的真偽值來還原圖片的屬性。

0039 及 0040 列分別為 *TTerrTiles* 及 *TItemTiles* 類別各宣告一個物件，因為它們只使用一個物件便已足夠，因此在此宣告，在三支程式中使用時就不用分別再為它們宣告了。

好了，完成了三個類別，它們皆置於 *TileUnit* 單元。

TMap 地圖類別

嘿，接下來輪到 *TMap* 類別了（好像在幸羊圈裏的待幸綿羊似的:p），*TMap* 類別定義於 *MapUnit.h*：

```

#0001 typedef Byte TMapArray[TILE_NUM_Y][TILE_NUM_X];
#0002
#0003 class TMap {
#0004     private:

```

```
#0005   TMapArray FTerrMap, FItemMap; // 地形及物品地圖
#0006
#0007   int FLevelNo; // 目前載入的關卡編號
#0008   Graphics::TBitmap* FInvBitmap; // 用於物品的透明貼圖
#0009
#0010   int FRole_X, FRole_Y; // 角色的起始位置
#0011
#0012   void SetLevelNo(int Value);
#0013
#0014   bool GetCanPass(int x, int y);
#0015   bool GetIsTarget(int x, int y);
#0016
#0017   bool GetCanMove(int x, int y);
#0018   bool GetIsSource(int x, int y);
#0019
#0020   TTerrAttr GetTerrAttr(int x, int y);
#0021   void SetTerrAttr(int x, int y, TTerrAttr Value);
#0022   TItemAttr GetItemAttr(int x, int y);
#0023   void SetItemAttr(int x, int y, TItemAttr Value);
#0024
#0025   Byte GetTerrMap(int x, int y);
#0026   Byte GetItemMap(int x, int y);
#0027   void SetTerrMap(int x, int y, Byte Value);
#0028   void SetItemMap(int x, int y, Byte Value);
#0029   void SetRole_X(int Value);
#0030   void SetRole_Y(int Value);
#0031   protected:
#0032   public:
#0033       TMap();
#0034       ~TMap();
#0035
#0036   AnsiString GetFileName(); // 根據關卡編號，傳回對應的檔名
#0037
#0038   void LoadFromFile();
#0039   void SaveToFile();
#0040
#0041   void DrawTerrMap(TCanvas* Canvas); // 將地形畫在 Canvas 上
#0042   void DrawItemMap(TCanvas* Canvas); // 將物品畫在 Canvas 上
#0043
#0044   // 重設整張地圖，或只重設地形或物品
#0045   void ResetMap();
#0046   void ResetTerrs();
#0047   void ResetItems();
#0048
#0049   __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0050
```

```

#0051 // 取得某格的地形及物品圖片編號
#0052 __property Byte TerrMap[int x][int y] =
#0053     {read = GetTerrMap, write = SetTerrMap};
#0054 __property Byte ItemMap[int x][int y] =
#0055     {read = GetItemMap, write = SetItemMap};
#0056
#0057 // 取得初始的角色位置
#0058 __property int Role_X = {read = FRole_X, write = SetRole_X};
#0059 __property int Role_Y = {read = FRole_Y, write = SetRole_Y};
#0060
#0061 // 取得某格的地形屬性
#0062 __property TTerrAttr TerrAttr[int x][int y] =
#0063     {read = GetTerrAttr, write = SetTerrAttr};
#0064
#0065 __property bool CanPass[int x][int y] = {read = GetCanPass};
#0066 __property bool IsTarget[int x][int y] = {read = GetIsTarget};
#0067
#0068 // 取得某格的物品屬性
#0069 __property TItemAttr ItemAttr[int x][int y] =
#0070     {read = GetItemAttr, write = SetItemAttr};
#0071
#0072 __property bool CanMove[int x][int y] = {read = GetCanMove};
#0073 __property bool IsSource[int x][int y] = {read = GetIsSource};
#0074 };
#0075
#0076 TMap Map;

```

光是類別宣告就洋洋灑灑的七十六行，其實絕大部分是屬性宣告及對應的讀取／寫入函式，讓 *TMap* 類別使用起來更方便罷了。

首先宣告 *TMapArray* 型別，它是 *TILE_NUM_X * TILE_NUM_Y* 個元素的 *Byte* 陣列，*Byte* 型態的範圍為 0 ~ 255，表示圖片最多只能有 256 種，我知道這時又有人想 K 我了，不急，換個角度想，現在限制越多，表示改良空間越大，以後領到最佳進步獎的機會也越高說。不過 256 個圖片對於足球番這種小遊戲來講再怎麼說也夠多了，不夠的話隨時再將 *Byte* 改為 *Word* 或 *unsigned long*（四位元組的無號正整數）也成。

0005 列宣告 *FTerrMap* 及 *FItemMap* 兩個型態為 *TMapArray* 的地形及物品地圖。在這兒我做了特殊的設定：「0 號地形為預設地形，而 0 號物品表示此處無物品」。因為地形一定遍布整張地圖，但物品不是，所以一定得設定一個數字表示此處沒有東西，而 0 號是最佳選擇。

處理透明貼圖

0008 列所宣告的 *FInvBitmap* 物件是專門用來供物品圖片做透明貼圖用的。透明貼圖指的是將物品「貼」到背景上時，周圍不屬於物品本身之處，就應該讓背景顯現出來，如圖 8-5。

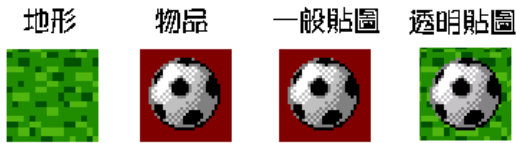


圖 8-5 / 一般貼圖及透明貼圖的比較

透明貼圖一直是 GDI 繪圖中十分傷腦筋的問題。從前 DOS 時代在 X mode 下，連貼圖動作都是自己寫的，利用兩層迴圈將像素「塞」進顯示卡的視訊記憶體，完全掌握著最低階的動作，因此達成透明貼圖的效果只是舉手之勞。而 DirectDraw 的 *ISurface* 也提供了 *SetColorKey* 等函式提供透明貼圖的解決方案。

但在 GDI 中要達成透明貼圖的效果可就麻煩了，一來我們對它的掌控能力不若 DOS 下直接填寫視訊記憶體那麼完整，二來 GDI 本身動作就慢，不容許我們花太多額外時間在處理透明貼圖。於是，最早最早，在 Windows 95 那個時代，GDI API 還沒有提供任何關於透明貼圖的解決方案前，達成透明貼圖通常必須經由下列的九大步驟：

實驗器材：

原始影像一，遮罩顏色一，遊戲畫面一。

實驗目的：

將原始影像貼到遊戲畫面上，但不貼原始影像中遮罩顏色的部分，使得那一部分仍然呈現原本的遊戲畫面。

實驗步驟：

1. 建立一個 DC 來放置原始影像，稱之為「影像 DC」。

2. 將原始影像選擇至 DC 上。
3. 另外建立一個 memory DC 來存放最後的影像，暫且稱它為「目的 DC」。
4. 將畫面上將要貼圖的矩形區域圖形拷貝到目的 DC。
5. 建立一個「AND 遮罩」，讓原始影像所有以遮罩顏色繪製的部分通通變成透明的，建立「AND 遮罩」有下列三小步驟：
 - 將「影像 DC」的背景顏色設定為遮罩顏色。
 - 建立一個相同大小的單色 DC。
 - 將原始影像不管三七二十一貼到此單色 DC 上。這使得此單色 DC 變成原始影像透明貼圖用的「AND 遮罩」，此遮罩於原始影像為遮罩顏色部分呈現 1，而非遮罩顏色部分呈現 0。
6. 呼叫 *BitBlt* 函式，搭配 *SRCAND* 貼圖模式將「AND 遮罩」貼到「目的 DC」上。
7. 呼叫 *BitBlt* 函式，搭配 *SRCAND* 貼圖模式將經過反相的「AND 遮罩」貼到「影像 DC」上。
8. 呼叫 *BitBlt* 函式，搭配 *SRCPAINT* 貼圖模式將「影像 DC」貼到「目的 DC」上。
9. 最後將「影像 DC」貼到畫面上適當的位置。

呼，看得都很累了，這是 MSDN 裏建議使用的方法，當然不是唯一做法囉，只是方法大同小異，簡單不到哪去。

Tips

你可以發現這九道步驟完全沒有涉及調色盤的處理，所以這方法只適用於不使用調色盤的顯示模式，不適合 256 色或 16 色模式使用。

但是，若你的程式只想在 Windows NT 上執行，Windows NT 提供一道新的 *MaskBlt* 函式，讓我們可以很簡單地達成透明貼圖，只是這一點都不實用，有誰想要寫只能在 Windows NT 執行的遊戲呢？

雖然微軟知錯能改，亡羊補牢，早已提出 *TransparentBlt*、*AlphaBlend* 透明及半透明貼圖

等 GDI 函式，但只在 Windows 2000 才提供，遠水救不了近火，我們還是自求多福，自己動手做透明貼圖囉。

很幸運地，VCL 的 *TCanvas* 及 *TBitmap* 類別提供透明貼圖的機制，只要將 *TBitmap* 的 *Transparent* 屬性設為 *true*，將 *TransparentColor* 設為遮罩顏色，再呼叫 *TCanvas::Draw* 函式就行了：

```
#0001 void __fastcall TForm1::btnVCLClick(TObject *Sender)
#0002 {
#0003     Graphics::TBitmap* Bits;
#0004
#0005     Bits = new Graphics::TBitmap; // 建立暫時的 TBitmap
#0006     Bits->Assign(imgSrc->Picture->Bitmap); // 將原始影像拷貝過來
#0007     Bits->Transparent = True;
#0008     Bits->TransparentColor = RGB(128, 0, 0); // 設定遮罩顏色
#0009
#0010     imgDst->Canvas->Brush->Color = clBtnFace;
#0011     imgDst->Canvas->FillRect(imgDst->Canvas->ClipRect);
#0012     imgDst->Canvas->Draw(0, 0, Bits); // 複製到目的畫布上
#0013     delete Bits;
#0014 }
```

要注意的是，*TBitmap* 的這幾個 *TransparentXXXX* 屬性只針對 *TCanvas* 的 *Draw* 及 *BrushCopy* 函式有效，對於 *TCanvas* 的其它函式或 GDI 函式皆無效用，所以若你將 0012 行改成 *BitBlt* API 函式或 *TCanvas* 的 *StretchBlt* 函式時，會發現完全沒有透明貼圖的效果。你可以在 *Graphics* 單元中找到實作透明貼圖的 *TransparentStretchBlt* 函式，若在 Windows NT 下而且原始及目的影像大小一樣時，它就直接呼叫 *MaskBlt* GDI 函式；對於其它版本的 Windows，就仿前頭的九大步驟，先製作「AND 遮罩」，經過幾道邏輯貼圖，再加上調色盤的處理，以支援 256 色或 16 色模式下的透明貼圖效果。

Info

十分遺憾的是，*TCanvas* 的透明貼圖能力在 Windows 98 竟然出問題，無論 C++Builder 5 或新出爐的 C++Builder 6，這個問題依舊存在，Borland 真該打。

真是氣煞人也，沒關係，幸好我們有那九陽真經..哦，不，是透明貼圖九步心訣，大不了自己做一個就是，沒什麼了不起。我寫了一個小範例程式，分別使用 VCL 及 API 來達成

透明貼圖，執行結果請見下頁圖 8-6。



圖 8-6 / 分別使用 VCL 及 API 來達成透明貼圖的範例程式

好，回到正題，我們提到，*FInvBitmap* 物件是專門用來供物品圖片做透明貼圖用的，因此在 *TMap* 的建構函式中，可以看到 *FInvBitmap* 的屬性設定：

```
TMap::TMap()  
{  
    ...  
  
    FInvBitmap = new Graphics::TBitmap;  
    FInvBitmap->Width = TILE_WIDTH;  
    FInvBitmap->Height = TILE_HEIGHT;  
    FInvBitmap->Transparent = true;  
    FInvBitmap->TransparentColor = (TColor)RGB(128, 0, 0);  
}
```

待會便拿它來達成貼物品圖片的透明貼圖效果。而遮罩顏色 *RGB(128, 0, 0)* 是我任意選定的，但選定就不要更動，因為遊戲中所有的物品及角色圖片都必須嚴格遵照這個遮罩顏色來繪製，若有更動便要大肆修改，十分麻煩。

關卡檔案的載入儲存

TMap 類別的載入及儲存函式會分別將兩層地圖，及初始角色位置放到檔案或從檔案讀出。這裏的改進空間極大，也可將其它與關卡相關的設定一併儲存，例如關卡描述啦、

過關提示啦、過關美女圖啦，都一起放到關卡檔案中。*TMap::LoadFromFile* 函式如下：

```
#0001 void TMap::LoadFromFile()
#0002 {
#0003     TFileStream* fs;
#0004     TReader* reader;
#0005
#0006     fs = new TFileStream(GetFileName(), fmOpenRead);
#0007     reader = new TReader(fs, 2048);
#0008     try {
#0009         CheckSignature(reader, SIG_MYFILE);
#0010         CheckSignature(reader, typeid(*this).name());
#0011
#0012         FRole_X = reader->ReadInteger();
#0013         FRole_Y = reader->ReadInteger();
#0014
#0015         reader->Read(FTerrMap, sizeof(TMapArray));
#0016         reader->Read(FItemMap, sizeof(TMapArray));
#0017     } __finally {
#0018         delete reader;
#0019         delete fs;
#0020     }
#0021 }
```

0009 及 0010 列同樣地檢查檔頭標籤及副檔頭標籤，接下來讀取角色位置，最後才是兩張固定大小的地形及物品地圖，過程平鋪直述，十分簡單直覺。

TMap 類別的重頭戲是 *DrawTerrMap* 及 *DrawItemMap* 兩個分別繪製地形層及物品層畫面的函式，裏頭同樣地都是兩層迴圈，一一尋訪所有的圖格，根據那一格的地形或物品圖片編號，將圖片畫在 *Canvas* 上頭：

```
#0001 void TMap::DrawTerrMap(TCanvas* Canvas)
#0002 {
#0003     for (int y = 0; y < TILE_NUM_Y; y++)
#0004         for (int x = 0; x < TILE_NUM_X; x++)
#0005             BitBlt(Canvas->Handle, x * TILE_WIDTH, y * TILE_HEIGHT,
#0006                 TILE_WIDTH, TILE_HEIGHT, Terrs.Bitmap->Canvas->Handle,
#0007                 Terrs.TilePos_Left[FTerrMap[y][x]],
#0008                 Terrs.TilePos_Top[FTerrMap[y][x]], SRCCOPY);
#0009 }
#0010
#0011 void TMap::DrawItemMap(TCanvas* Canvas)
#0012 {
#0013     // 統一 FInvBitmap 及 Items.Bitmap 的格式
```

```

#0014   FInvBitmap->PixelFormat = Items.Bitmap->PixelFormat;
#0015
#0016   for (int y = 0; y < TILE_NUM_Y; y++)
#0017     for (int x = 0; x < TILE_NUM_X; x++)
#0018       if (FItemMap[y][x]) {
#0019         int Left = Items.TilePos_Left[FItemMap[y][x]];
#0020         int Top = Items.TilePos_Top[FItemMap[y][x]];
#0021
#0022         FInvBitmap->Canvas->CopyRect(Rect(0, 0, TILE_WIDTH,
#0023           TILE_HEIGHT), Items.Bitmap->Canvas,
#0024           Rect(Left, Top, Left + TILE_WIDTH, Top + TILE_HEIGHT));
#0025
#0026         // 解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround
#0027         if (IsWindows98())
#0028           DrawTransparentBitmap(Canvas->Handle, FInvBitmap->Handle,
#0029             x * TILE_WIDTH, y * TILE_HEIGHT, RGB(128, 0, 0));
#0030         else
#0031           Canvas->Draw(x * TILE_WIDTH, y * TILE_HEIGHT, FInvBitmap);
#0032       }
#0033   }

```

0005 列繪製地形層比較單純，呼叫 *BitBlt* 函式，將對應的地形圖片拷貝到 *Canvas* 上的對應位置。但繪製物品層時，首先要檢查該格是否有物品，若 *FItemMap[y][x]* 為零，表示沒有物品，就不畫；另外要繪製時，先將圖片拷貝至 *FInvBitmap* 上，再呼叫 *TCanvas::Draw* 函式貼上 *FInvBitmap*，目的就是為了先前討論已久的透明貼圖。理論上是如此，但我們先前提過 VCL 的透明貼圖能力在 Windows 98 下失效，所以 0028 行就專為解決 VCL 在 Windows 98 下透明貼圖 bug 而特別將 Windows 98 平臺下的貼圖動作獨立出來，先以 *IsWindows98* 函式（Util 單元）判斷目前是否處於 Windows 98 下，若是的話，再呼叫我們自己的 *DrawTransparentBitmap* 函式來達成透明貼圖。

大一統的 Win32 平臺？！

經常撰寫 Win32 程式的程式員，一定會對微軟吶喊的「統一的 Win32 平臺」這句好聽的口號印象深刻吶！這...真的只是口號罷了，我很少寫出一套不需針對不同 Windows 版本修正問題的軟體。判斷系統是否為 Windows 98 並不難，檢查 VCL 提供的三個全域變數 *Win32Platform*、*Win32MajorVersion* 及 *Win32MinorVersion* 即可，詳情請查閱 *GetVersionEx*

API 函式。而 Windows 98 即等於版本為 4.10 以上的 Windows：

```
bool IsWindows98()
{
    return (Win32Platform == VER_PLATFORM_WIN32_WINDOWS) &&
        ((Win32MajorVersion > 4) ||
         ((Win32MajorVersion == 4) && (Win32MinorVersion >= 10)));
}
```

若系統為 Windows NT，則 *Win32Platform* 變數值為 *VER_PLATFORM_WIN32_NT*，而 Windows 2000 的 *Win32MajorVersion* == 5，因為其實就是 NT 5.0 嘛。

二維陣列屬性

最後再實作 *TMap* 所提供的一大票屬性，以 *CanPass* 屬性為例，它是二維陣列屬性，宣告為：

```
__property bool CanPass[int x][int y] = {read = GetCanPass};
```

它有一個對應的屬性讀取函式，因此就要另外實作此函式：

```
bool TMap::GetCanPass(int x, int y)
{
    return Terrs.Attrs[FTerrMap[y][x]].Contains(taCanPass);
}
```

於是 *TMap* 的物件使用者就可以輕易地使用此屬性，如經由 *Map.CanPass[2][3]* 取得一個布林值，判斷此地圖座標為 (2, 3) 的格點是否能夠讓角色通過；由 *Map.CanMove[8][2]* 來判斷座標為 (8, 2) 處是否有物品，若有物品，能不能推動等等。

TMap 類別宣告的 0076 列，宣告一個 *TMap* 類別的全域物件 *Map*，原因也跟 *TTerrTiles* 及 *TItemTiles* 類別一樣，因為 *TMap* 物件只需要一個，因此宣告在這兒最清楚也最方便。

TRole 主角類別

終於剩下最後一個類別—*TRole*（定義於 *Role.h*）：

```

#0001 enum TDirection {drUp, drDown, drLeft, drRight};
#0002 typedef std::vector<TDirection> TDirectionArray;
#0003
#0004 class TRole {
#0005 private:
#0006     int FX, FY; // 角色座標
#0007     TDirection FDirection; // 行進方向
#0008     Graphics::TBitmap *FBits, *FInvBitmap; // 角色圖片及透明貼圖用圖片
#0009
#0010     TDirectionArray FPlayBackList, FMoveList; // 重播功能用的動作記錄
#0011
#0012     TDirectionArray& GetPlayBackList() {return FPlayBackList;}
#0013 public:
#0014     TRole();
#0015     ~TRole();
#0016
#0017     void LoadBits(); // 載入角色的 bitmap
#0018     void Draw(TCanvas* Canvas); // 繪製角色
#0019     void Move(TDirection Dir); // 移動角色 (碰撞處理, 移動物品)
#0020
#0021     void SavePlayBack(); // 將動作記錄移至重播紀錄
#0022     void CleanMoveList(); // 清除動作記錄
#0023
#0024     // 位置及方向
#0025     __property int X = {read = FX, write = FX};
#0026     __property int Y = {read = FY, write = FY};
#0027     __property TDirection Direction =
#0028         {read = FDirection, write = FDirection};
#0029
#0030     __property TDirectionArray& PlayBackList =
#0031         {read = GetPlayBackList};
#0032 };

```

0001 列先宣告 *TDirection* 列舉型態，定義出角色可能面對及移動的方向。*FInvBitmap* 變數的目的及用法與 *TMap* 的 *FInvBitmap* 一模一樣，同時是為了透明貼圖功能而存在的。

TRole 的設定比較偷懶，因為畫面上只有唯一一個角色，因此圖片也只要一份，上下左右各一張圖片，總共才需一張 128 x 32 的點陣圖。若遊戲中可同時出現多種角色，每種角色又有不同的圖形及行為模式時，*TRole* 的設計可就沒這麼簡單。一般來說，走動時的圖片都常會一個方向提供三張，分別是站立不動、提起左腳及邁開右腳，若再加上蹲姿或射擊、中彈等其它動作表情等等，所需的圖片數量還真多，同樣地，留待日後再來加強角色的功能。

讀取角色圖形的函式很簡單，呼叫 *TBitmap::LoadFromFile* 從 *bitmap* 檔案中讀出即可：

```
#0001 void TRole::LoadBits()
#0002 {
#0003     // 讀取 BMP 檔，內含四個方向的圖形
#0004     FBits->LoadFromFile(FN_ROLEBITS);
#0005
#0006     if (FBits->Width < TILE_WIDTH * 4) // 四個方向
#0007         throw Exception("Width of role bits is invalid");
#0008
#0009     // 統一 FInvBitmap 及 FBits 的格式
#0010     FInvBitmap->PixelFormat = FBits->PixelFormat;
#0011 }
```

接下來是畫出角色的 *Draw* 函式，與 *TMap::DrawItemMap* 函式類似，先將對應的圖片拷貝到 *FInvBitmap* 上，再呼叫 *TCanvas::Draw* 將 *FInvBitmap* 以透明貼圖的方式貼上去。這裏也同樣有解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround。

```
#0001 void TRole::Draw(TCanvas* Canvas)
#0002 {
#0003     // 先將要秀出的區域拷至 FInvBitmap，再畫出 FInvBitmap
#0004     FInvBitmap->Canvas->CopyRect(Rect(0, 0, TILE_WIDTH, TILE_HEIGHT),
#0005     FBits->Canvas, Rect((int)FDirection * TILE_WIDTH, 0,
#0006     ((int)Direction + 1) * TILE_WIDTH, TILE_HEIGHT));
#0007
#0008     // 解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround
#0009     if (IsWindows98())
#0010         DrawTransparentBitmap(Canvas->Handle, FInvBitmap->Handle,
#0011         FX * TILE_WIDTH, FY * TILE_HEIGHT, RGB(128, 0, 0));
#0012     else
#0013         Canvas->Draw(FX * TILE_WIDTH, FY * TILE_HEIGHT, FInvBitmap);
#0014 }
```

移動及推動

TRole 類別最重要的函式，也是與使用者操作最直接相關的，就屬 *Move* 函式了。它決定當使用者按上下左右方向鍵後，角色的移動及搬動物品的處理：

```
#0001 // small but useful function, update x & y location of role
#0002 void MoveAhead(int& X, int& Y, TDirection Dir)
#0003 {
#0004     switch (Dir) {
```

```
#0005     case drUp: Y--; break;
#0006     case drDown: Y++; break;
#0007     case drLeft: X--; break;
#0008     case drRight: X++; break;
#0009     }
#0010 }
#0011
#0012 void TRole::Move(TDirection Dir)
#0013 {
#0014     int X, Y, MX, MY;
#0015
#0016     X = FX; Y = FY;
#0017     MoveAhead(X, Y, Dir); // 計算角色的下一位置
#0018     FDirection = Dir;
#0019
#0020     // 超出畫面範圍了...
#0021     if (Y < 0 || X < 0 || X >= TILE_NUM_X || Y >= TILE_NUM_Y) return;
#0022
#0023     if (!Map.CanPass[X][Y]) return; // 不能走的地形
#0024
#0025     if (Map.ItemMap[X][Y] != 0) { // 如果要移動過去的位置有物品
#0026         if (!Map.CanMove[X][Y]) return; // 不能搬動耶
#0027
#0028         MX = X; MY = Y;
#0029         MoveAhead(MX, MY, Dir); // 計算物品的新位置
#0030         if (MY < 0 || MX < 0 || MX >= TILE_NUM_X || MY >= TILE_NUM_Y)
#0031             return; // 不可將物品移到範圍外
#0032
#0033         // 要搬過去的新位置上不能有東西，也必須是可以走動的地形
#0034         if (Map.ItemMap[MX][MY] != 0 || !Map.CanPass[MX][MY]) return;
#0035
#0036         Map.ItemMap[MX][MY] = Map.ItemMap[X][Y]; // 搬動過去
#0037         Map.ItemMap[X][Y] = 0; // 原來的地方沒有物品了
#0038     }
#0039
#0040     // 成功走出，將移動方向記錄下來
#0041     FMoveList.push_back(Dir);
#0042
#0043     // 更新角色位置
#0044     FX = X;
#0045     FY = Y;
#0046 }
```

先設計一個小小的 *MoveAhead* 函式來輔助位置的處理，根據傳入的 *Dir* 方向，更改 *X* 或 *Y* 軸位置，雖然簡單，但很有用。

上面處理角色移動的邏輯並不難，可歸納如下：

1. 0017 列先計算下一步的位置。
2. 0021 列判斷是否超出畫面範圍了，是的話就跳離處理函式。
3. 0023 列判斷是否將走到不能穿過的地形，是的話就跳離處理函式。
4. 0025 列判斷是否將走到有物品的圖格，是的話繼續步驟 5，否則進行步驟 9。
5. 0026 列判斷該物品能否搬動，否的話就跳離處理函式。
6. 0030 列判斷是否會將物品搬出畫面範圍了，是的話就跳離處理函式。
7. 0034 列判斷物品的新位置上是否有東西，是否為可以穿越的地形？若沒有擺置物品且地形可以穿越就繼續，否則跳離處理函式。
8. 0036、0037 列將物品搬動到新位置。
9. 0041 列將移動方向記錄在 *FMoveList* 物件中，留待重播功能使用。
10. 0044、0045 列更新角色位置，大功告成。

程式乍看之下可能不怎麼懂，但若用文字敘述出來，就變成很簡單的幾道判斷敘述而已，而這已是整個遊戲中最麻煩重要的邏輯處理呢。所以我說的沒錯吧，撰寫遊戲一點都不難，難的通常是顯示技術、速度及程式複雜度，而非程式邏輯。讓我們繼續往下看。

TRole 類別宣告的 0009 列中，將重播動作記錄用的兩個變數 *FPlayBackList* 及 *FMoveList* 宣告為 *TDirectionArray* 類別，

在這兒，我想要將角色的每一步移動方向都記錄下來，但是角色的移動步數未知，可能只有兩三步，也可能是兩三千步，所以不適合靜態地配置記憶體空間。但自己呼叫 *malloc*、*realloc*、*free* 等函式來管理移動記錄所需的記憶體空間又稍嫌麻煩，*vector* 物件在此是最適宜的容器。

角色動作錄影支援

TRole 的 *SavePlayback* 及 *CleanMoveList* 函式分別將 *FMoveList* 的記錄移至 *FPlaybackList* 中，以及將 *FMoveList* 的內容清除：

```
#0001 void TRole::SavePlayBack()  
#0002 {  
#0003     // 將行動記錄放到 FPlayBackList 中，以便重播  
#0004     FPlayBackList = FMoveList;  
#0005 }  
#0006  
#0007 void TRole::CleanMoveList()  
#0008 {  
#0009     FMoveList.clear(); // 行動記錄清除以方便下次使用  
#0010 }
```

不知不覺間，我們已將 *TTiles*、*TMap* 及 *TRole* 等三個核心類別實作完成，好的開始是成功的一半，緊接著，就要利用上頭介紹的這些類別來架構遊戲的三支程式囉。

圖庫編輯器、地圖編輯器，以及遊戲主程式的實作也有一定的順序。圖庫編輯器還未完成前，沒有圖庫，就無法編輯地圖；而地圖編輯器還未完成前，沒有地圖，就無法進行遊戲。所以看來非從圖庫編輯器下手不可。

圖庫編輯器

RAD 開發工具的好處是，可以在開始撰寫第一行程式碼前，先透過「所視即所得」的整合環境編輯方式，將使用者介面，所有的控制項，元件，選單及視覺佈局擺好在視窗上，待程式一執行，嘿，就是設計時期擺放的那個模樣。

我個人十分偏愛這樣的設計方式，先將使用者介面設計好，再下手來撰寫程式碼。只要介面制訂後不再更動，程式碼就好寫；若是介面甚至元件種類還變來變去，那程式碼肯定就得改來改去，越修越複雜，麻煩的不得了。

好，選取【New / Application】開啓一個新專案順便建立 main form，在 form 上將介面佈局擺好，如圖 8-7，看起來很陽春，我知道，是我的錯，以後改進。圖 8-8 是其選單設計畫面，可由其得知圖庫編輯器準備提供的功能。

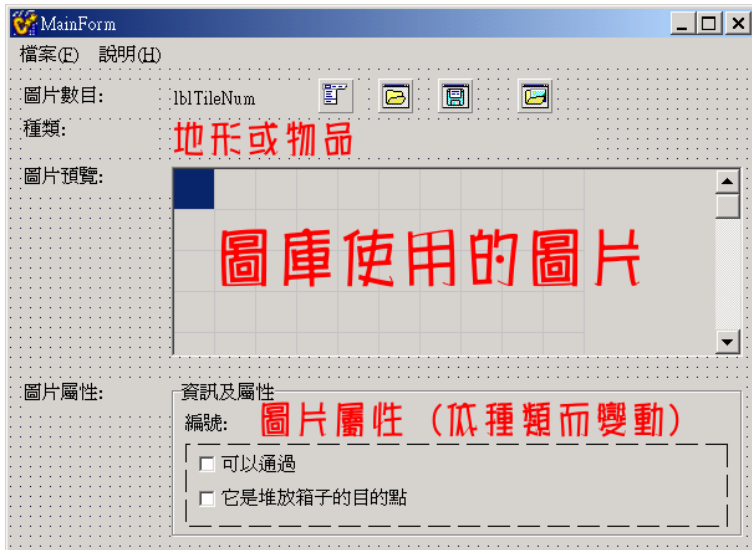


圖 8-7 / 圖庫編輯器的設計畫面



圖 8-8 / 圖庫編輯器的選單設計畫面

假設我準備好 16 張地形圖片（大小皆是 32 x 32），想要納入遊戲使用。典型的操作方式是這樣的：

1. 請先在影像編輯軟體中編輯 320 x 64（因為每列 10 格，16 張需佔用兩列）大小的影像，分別將 16 張圖片放到適當的位置，第一張擺在（0, 0）、第二張擺在（32, 0）、第三張擺在（64, 0）...，最後儲存為 BMP 檔案。

2. 接著執行圖庫編輯器，選擇【檔案 / 開新檔案】，圖庫類型選擇為「地形」，建立新的圖庫。
3. 匯入事先準備好的 BMP 圖檔。
4. 針對每張圖片一一設定它們的屬性，決定角色是否可通過、是否為目的地形等等。
5. 最後，存檔成 *FN_TERR_ARCHIVE* 常數所指定的檔名，就可以順利供地圖編輯器及遊戲主程式載入使用。
6. 物品圖庫也依上述步驟如法泡製。

我打算利用同一套程式，相同的介面，幾乎相同的程式碼來進行地形圖庫及物品圖庫的製作及處理。

雙重「物」格的 FTiles

要怎麼做呢？地形圖庫及物品圖庫分別屬於 *TTerrTiles* 及 *TItemTiles* 類別，而不同型態，不同類別的物件通常要分別撰寫程式碼處理才行。別忘了，它們來自同一個父類別—*FTiles*，藉著多型機制的幫忙，我們可以讓 *TTerrTiles* 及 *TItemTiles* 物件共享同一段程式碼，甚至共用同一個變數：

```
#0001 class TMainForm : public TForm
#0002 {
#0003 ...
#0004 private:
#0005     FTiles* FTiles;
#0006
#0007     void __fastcall CreateTiles(const std::type_info&);
#0008 };
#0009
#0010 void __fastcall TMainForm::CreateTiles(const std::type_info&
#0011     typeinfo)
#0012 {
#0013     if (FTiles) delete FTiles;
#0014
#0015     if (typeinfo == typeid(TTerrTiles))
#0016         FTiles = new TTerrTiles();
#0017     else if (typeinfo == typeid(TItemTiles))
```

```
#0018     FTiles = new TItemTiles();
#0019     else // 不是預期的類別
#0020         throw Exception("Invalid class type info");
#0021 }
```

你瞧，0005 列宣告著 *TTiles* 類別的 *FTiles* 物件，因為同一時間只能編輯地形或物品圖庫其中之一，因此我使用同一個 *FTiles* 變數來儲存這兩種物件。

我們希望 *FTiles* 變數有時指向 *TTerrTiles* 物件，有時指向 *TItemTiles* 物件，並使用一道專門的函式 *CreateTiles* 來負責 *FTiles* 物件的建立及釋放。那麼，*CreateTiles* 函式一定需要接收參數，來得知應該建立哪一個類別的物件。在 C++ 裡頭，類別只是個抽象的描述，類別的作用就是拿來建立物件，以及衍生新的資料型別，從來無法把它當作參數傳遞，所以我們不能這麼宣告：

```
void CreateTiles(const ????? class) { ... }
...
CreateTiles(TTerrTiles); // 建立 TTerrTiles 物件
CreateTiles(TItemTiles); // 建立 TItemTiles 物件
```

每個變數及物件都有它的資料型別，但類別屬於什麼資料型別？我們可以說 *MainForm* 的資料型別為 *TMainForm*、變數 *i* 的資料型別為 *int*，但 *TMainForm* 以及 *int* 的資料型別是什麼？

答案是 C++ 並沒有規範「資料型別」的「資料型別」。所以我們不能將「資料型別」直接當成函式參數傳遞。在這兒，我們可以依賴 C++ 的 RTTI (Run-Time Type Information，執行時期型別資訊) 機制來間接傳遞「資料型別」。

可別被這串陌生的英文縮寫嚇到了，事實上它一點都不困難，簡單地說，RTTI 就是把「資料型別」的各項資訊，也編譯入目的碼，讓我們可在程式執行時期，取得「資料型別」的各種資訊。這些資訊其實就是我們（人類）可以由原始程式碼輕易得到的訊息及定義，但由於 C++ 是靜態型別語言，所以在編譯之後，許多在程式執行時（CPU）用不上的資料型別定義就失去了，不會帶進目的碼裡頭，這是很直覺的作法，也兼顧目的碼大小與執行效率。例如以下宣告的列舉型別變數 *Alignment*：

```
enum {Evil, Neutral, Good} Alignment;  
if (Alignment == Evil) ...  
else if (Alignment == Good) ...
```

經過編譯之後，編譯器會將每一個列舉型別的值編號，例如 *Evil* 為 0、*Neutral* 為 1、*Good* 為 2，而 *Alignment* 只不過是數值可能為 0~2 的無號整數。每一個在程式碼中使用 *Evil*、*Neutral* 及 *Good* 之處，都會自動以 0、1、2 來看待。基本上，跟下列寫法無異，只不過以整數值來表示三種陣營，容易讓程式員搞混，大大降低程式維護性而已：

```
unsigned int Alignment; // possible values: 0 ~ 2  
if (Alignment == 0) ...  
else if (Alignment == 1) ...
```

CPU 執行程式時只需要這些資訊即足夠，但相對的我們已失去了資訊型別的資訊。例如我們已經無法從 *Alignment* 的數值 0，取得“Evil”這個字串，因為“Evil”這個字串只在原始碼中有效，並沒有編入目的碼中。

C++ 的 RTTI 就是為了解決這類問題而出現，此機制會將各種資料型別的資訊記錄下來，一併編譯入目的碼中，讓我們在執行時期時可透過程式取用。事實上，雖然它看起來只是很簡單的機制，C++ 的某些語言設施（如多型）以及 C++Builder 的 RAD 開發環境，皆是由於 RTTI 在背後支援，才能達到。

C++ 的 RTTI 支援，以 *typeid* 保留字為起始點，由此我們可以取得 *std::type_info* 類別，對於描述資料型別的資訊進行比較、比對及取得型別名稱等動作。例如：

```
int x;  
ShowMessage(typeid(x).name()); // 顯示 int  
  
TComponent* b = new TButton(NULL);  
ShowMessage(typeid(b).name()); // 顯示 TComponent*  
ShowMessage(typeid(*b).name()); // 顯示 TButton (支援多型, see ?)  
  
if (typeid(x) == typeid(y)) ... // 檢查兩個變數的資料型別是否相等
```

對於 *std::type_info* 類別，請參閱 C++Builder 線上說明的 *typeid* 及 *type_info* 等主題。

於是，藉由 *typeid* 保留字及 *std::type_info* 類別，我們可讓 *CreateTiles* 接收 *std::type_info&* 參數，傳入的參數再與 *typeid(TTerrTiles)* 及 *typeid(TItemTiles)* 比對，就

可得知程式現在想要建立何種類別的物件了。*CreateTiles* 函式的呼叫方式為：

```
CreateTiles(typeid(TTerrTiles)); // 建立 TTerrTiles 物件
CreateTiles(typeid(TItemTiles)); // 建立 TItemTiles 物件
CreateTiles(typeid(TButton)); // 來鬧場的？CreateTiles 會丟出例外
```

以 RTTI 驗明「物」格

現在我們能確定的是，*FTiles* 指向的不是 *TTerrTiles* 物件就是 *TItemTiles* 物件，那麼要如何辨別目前指向的物件究竟是哪一種呢？這又得依賴 RTTI 機制了。我們必須使用 *dynamic_cast* 運算子來驗證物件的實際類別。它的使用方式為：

```
dynamic_cast<T>(ptr)
```

T 為類別指標或參考型別，或是 *void**；*ptr* 必須是型態為指標或參考的 *expression*。若轉型成功，*dynamic_cast* 會回傳型別為 *T* 的指標或參考；若轉型失敗，會傳回 *NULL* 或引發 *Bad_cast* 例外。詳細的轉型規則請見線上說明的「*dynamic_cast*」主題。

例如以下這道 *expression*：

```
dynamic_cast<TBarClass*>(FooObjectPtr)
```

如果 *FooObjectPtr* 指向 *TBarClass* 類別或衍生類別的物件，會得到型別的 *TBarClass** 的指標，否則得到 *NULL*。因此，我們可以使用 *dynamic_cast* 運算子來作保證安全的物件向下轉型：

```
// 如果 FooObjectPtr 指向 TBarClass 類別及衍生類別物件的話...
if (TBarClass* p = dynamic_cast<TBarClass*>(FooObjectPtr)) {
    ... // 使用物件指標 p
}
```

你可以在範例程式中看到我大量地用 *dynamic_cast* 運算子，尤其在事件處理函式內：

```
#0001 void __fastcall TMainForm::mnuSaveClick(TObject *Sender)
#0002 {
#0003     // 若是"另存新檔" 或還未指定檔名，就先問使用者檔名
#0004     if (dynamic_cast<TComponent*>(Sender)->Tag == 1 ||
#0005         FFileName == "") {
#0006         dlgSave->Filter = dlgOpen->Filter;
#0007         // 詢問使用者檔名，若按取消就離開
```

```
#0008     if (!dlgSave->Execute()) return;
#0009     FFileName = dlgSave->FileName; // 將檔名記起來
#0010     }
#0011
#0012     FFiles->SaveToFile(FFileName); // 儲存圖庫
#0013     UpdateControlStatus(); // 更新視窗標題
#0014     }
```

VCL 中絕大部分的事件處理函式都帶有一個 *Sender* 參數，代表觸發此函式的物件，因為不一定是什麼類別，所以 *Sender* 宣告為指向所有 VCL 類別的始祖—*TObject* 的指標，但其實它可能是一個 *TButton* 元件、一個 *TImage* 元件甚至一個 *Timer* 計時器元件。只要我們確定它應該是什麼類別的物件，就可以放心地將它轉型，然後呼叫函式或使用屬性。

以上面的 *mnuSaveClick* 函式為例，由於我只將這個事件處理函式指派給 *mnuSave* 及 *mnuSaveAs* 兩個 *TMenuItem* 物件，所以當 *mnuSaveClick* 函式被觸發時，理論上 *Sender* 參數必定是兩者其中之一（除非程式有其它地方呼此函式），於是我便可放心地將 *Sender* 參數轉型為 *TComponent* 類別，取得它的 *Tag* 屬性來使用。為何轉型為 *TComponent* 類別，而不轉為 *TMenuItem* 類別呢？其實兩者都行。只是我個人習慣讓程式碼擁有較大的彈性，若將它轉型為 *TMenuItem* 類別，萬一日後我又想將 *mnuSaveClick* 函式指派給另一個 *TButton* 物件時，型別轉換部分勢必得修改。因此，我個人的習慣是，若要存取某個屬性或呼叫函式，就將它轉型為該屬性或函式出現的類別。此處來說，*Tag* 屬性是 *TComponent* 類別介紹的新屬性，因此我就將 *Sender* 參數轉為 *TComponent* 類別。

提到 *Tag* 屬性，它是所有 VCL 元件都擁有的一個屬性，為四個位元組的長整數，我們可以任意地使用它。如上述的例子，我將【儲存】及【另儲新檔】兩個元件指派同一個事件處理函式，但代表【儲存】的 *TMenuItem* 元件的 *Tag* 屬性為 0，而代表【另儲新檔】的 *TMenuItem* 元件的 *Tag* 屬性則設為 1，以此來區分兩者的不同，進行對應的動作。若我不這樣做，而以傳統的做法分別為兩個 *TMenuItem* 元件撰寫不同的事件處理函式，就會有很多重覆的程式碼，佔空間，維護起來也較麻煩。

Ouch，離題似乎又遠了~~。於是呢，藉由 *dynamic_cast* 運算子之助，我們可以輕易判別出，目前 *FFiles* 變數指向的究竟是地形或者物品圖庫。

```

if (dynamic_cast<TTerrTiles*>(FTiles))
    ... // 是地形圖庫唷！
else if (dynamic_cast<TItemTiles*>(FTiles))
    ... // 是物品圖庫耶～

```

雙重「物」格變換

什麼情況下，*FTiles* 會指向不同類別的物件呢？只有兩個地方，一是開啓新檔時，詢問使用者準備建立的圖庫類型，二是開啓舊檔時：

```

#0001 void __fastcall TMainForm::mmuNewClick(TObject *Sender)
#0002 {
#0003     // 建立詢問對話盒
#0004     TTileKindDlg* dlg = new TTileKindDlg(this);
#0005     try {
#0006         // 秀出對話盒，使用者按下"確定"後會傳回 mrOK
#0007         if (dlg->ShowModal() == mrOk) {
#0008             if (dlg->rgpTileKind->ItemIndex == 0)
#0009                 CreateTiles(typeid(TTerrTiles)); // 地形圖庫
#0010             else
#0011                 CreateTiles(typeid(TItemTiles)); // 物品圖庫
#0012
#0013             FOldSelection = -1;
#0014             UpdateControlStatus();
#0015         }
#0016     } __finally {
#0017         delete dlg; // 無論如何，摧毀詢問對話盒
#0018     }
#0019 }

```

開新檔時比較簡單，我另外設計一個詢問使用者圖庫類型的對話盒，當使用者按下「確定」後會傳回 *mrOK*，再根據它的選擇呼叫 *ChangeTileClass* 函式建立地形或物品圖庫物件。



圖 8-9 / 詢問使用者圖庫類型的對話盒

開啓舊檔就比較有趣了，還記得之前設計 *TTiles* 類別時，在 *TTiles::SaveToFile* 函式中有寫入檔頭標籤及副檔頭標籤嗎？我們就靠著這個及 Object Pascal 的例外捕捉機制（`try .. except .. end`）來判斷讀入的是何種圖庫，同時建立對應的 *FTiles* 物件：

```
#0001 void __fastcall TMainForm::mnuOpenClick(TObject *Sender)
#0002 {
#0003     if (dlgOpen->Execute()) {
#0004         try {
#0005             // 先試試看是否為地形圖庫
#0006             // 讀取有錯的話，就會產生例外，讓我們的例外捕捉機制處理
#0007             CreateTiles(typeid(TTerrTiles));
#0008             FTiles->LoadFromFile(dlgOpen->FileName);
#0009             FFileName = dlgOpen->FileName; // 讀取地形圖庫成功
#0010         } catch (...) {
#0011             // 如果不是地形圖庫，檢查副檔頭標籤時會產生例外，
#0012             // 所以跳到這兒來執行
#0013             // 再試試看是否為物品圖庫，否則就什麼都不是
#0014             CreateTiles(typeid(TItemTiles));
#0015             // 有錯的話，也會產生例外，我們就不處理了
#0016             FTiles->LoadFromFile(dlgOpen->FileName);
#0017             FFileName = dlgOpen->FileName;
#0018         }
#0019
#0020         FoldSelection = -1;
#0021         UpdateControlStatus();
#0022     }
#0023 }
```

讀取圖庫時採取「試誤法」，先試試看是否為地形圖庫，不是的話，再試試看是否為物品圖庫，否則就不理它。我憑藉的是 *TTiles::LoadFromFile* 函式中會呼叫 *CheckSignature* 函式來檢查檔頭標籤，而 *CheckSignature* 函式會在檔頭標籤與預期不符時丟出一個例外：


```
#0001 void CheckSignature(TReader* reader, const AnsiString Sig)
#0002 {
#0003     AnsiString S = reader->ReadString();
#0004     if (Sig != S)
#0005         throw Exception("File signature not match");
#0006 }
```

這個看起來很 *dirty* 又很 *smart* 的「試誤法」，其實是懶得另外撰寫檢查檔頭標籤函式的結果，不然正常的寫法會像是這樣，你喜歡哪個呢？

```
if (IsTerrArchive(dlgOpen->FileName))
    CreateTileClass(typeid(TTerrTiles));
else
    CreateTileClass(typeid(TItemTiles));

FTiles->LoadFromFile(dlgOpen->FileName);
```

繪製圖庫圖片

視窗中央那個 *TDrawGrid* 物件 *grdPreview* 會根據目前的 *FTiles* 內容將圖片顯示出來，*TDrawGrid* 元件本身並不儲存任何資訊，顯示的結果端視我們如何處理它的 *OnDrawCell* 事件而定。以下是 *grdPreview* 的 *OnDrawCell* 事件處理函式：

```
#0001 void __fastcall TMainForm::grdPreviewDrawCell(TObject *Sender, int
#0002     ACol,
#0003     int ARow, TRect &Rect, TGridDrawState State)
#0004 {
#0005     // 由行及列換算圖片編號
#0006     int No = ARow * grdPreview->ColCount + ACol;
#0007
#0008     if (No < FTiles->TileNum) // 將對應的圖形畫出來
#0009         BitBlt(grdPreview->Canvas->Handle, Rect.Left, Rect.Top,
#0010             TILE_WIDTH, TILE_HEIGHT, FTiles->Bitmap->Canvas->Handle,
#0011             FTiles->TilePos_Left[No], FTiles->TilePos_Top[No], SRCCOPY);
#0012     else {
#0013         grdPreview->Canvas->Brush->Color = clBlack; // 編號大於圖片數目
#0014         grdPreview->Canvas->FillRect(Rect);
#0015     }
#0016 }
```

同樣地，還是經由 *TTiles* 的 *TilePos_Left* 及 *TilePos_Top* 兩個屬性來取得圖片在圖庫中的座標，再利用 *BitBlt* API 將該圖片貼到 *grdPreview* 的對應位置上；對於沒有圖片的那些

圖格，就塗黑。

而每當使用者選擇 *grdPreview* 上的某一格時，*grdPreview* 的 *OnSelectCell* 事件處理函式 *grdPreviewSelectCell* 就必須先將目前設定的屬性寫回 *FTiles* 物件中，接著再讀出即將選擇的圖片屬性，依屬性更新 *TCheckBox* 元件狀態：

```
#0001 void __fastcall TMainForm::grdPreviewSelectCell(TObject *Sender,
#0002     int ACol, int ARow, bool &CanSelect)
#0003 {
#0004     int No, OldNo = FOldSelection;
#0005
#0006     // OldNo 為原本選擇的圖片編號
#0007     if (OldNo < FTiles->TileNum) { // 將使用者設定的圖片屬性寫回去
#0008         // 若是地形圖庫的話...
#0009         if (TTerrTiles* TerrTiles = dynamic_cast<TTerrTiles*>(FTiles)){
#0010             TTerrAttr& Attr = TerrTiles->Attrs[OldNo];
#0011             // 根據 cbxCanPass 及 cbxTarget 兩個 checkbox 來設定圖片屬性
#0012             Attr = TTerrAttr();
#0013
#0014             // 可以穿越的地形
#0015             if (cbxCanPass->Checked) Attr = Attr << taCanPass;
#0016             // 它是目的地形
#0017             if (cbxTarget->Checked) Attr = Attr << taTarget;
#0018         } else if (TItemTiles* ItemTiles =
#0019             dynamic_cast<TItemTiles*>(FTiles)) {
#0020             // 若是物品圖庫的話...
#0021             TItemAttr& Attr = ItemTiles->Attrs[OldNo];
#0022             Attr = TItemAttr();
#0023
#0024             // 可以搬動的物品
#0025             if (cbxCanMove->Checked) Attr = Attr << iaCanMove;
#0026             // 覆蓋用物品
#0027             if (cbxSource->Checked) Attr = Attr << iaSource;
#0028         }
#0029     }
#0030
#0031     No = ARow * grdPreview->ColCount + ACol; // 計算即將選擇的圖片編號
#0032     if (No < FTiles->TileNum) { // 秀出目前所選的圖片屬性
#0033         lblTileNo->Caption = "編號: " + IntToStr(No);
#0034
#0035         // 將所選擇的圖片屬性由 checkbox 元件表現出來
#0036         if (TTerrTiles* TerrTiles = dynamic_cast<TTerrTiles*>(FTiles)){
#0037             TTerrAttr& Attr = TerrTiles->Attrs[No];
#0038             cbxCanPass->Checked = Attr.Contains(taCanPass);
```

```

#0039     cbxTarget->Checked = Attr.Contains(taTarget);
#0040     } else if (TItemTiles* ItemTiles =
#0041     dynamic_cast<TItemTiles*>(FTiles)) {
#0042     TItemAttr& Attr = ItemTiles->Attrs[No];
#0043     cbxCanMove->Checked = Attr.Contains(iaCanMove);
#0044     cbxSource->Checked = Attr.Contains(iaSource);
#0045     }
#0046
#0047     // 更新選擇的 tile no
#0048     FOldSelection = No;
#0049     } else
#0050     CanSelect = false; // 選擇不合法的圖格，不給選

```

嗯，到此為止，也許你不相信，但是圖庫編輯器一不小心就這樣完工了，圖 8-10 及圖 8-11 分別是圖庫編輯器在製作地形及物品圖庫時的執行畫面。從畫面中可以看到，地形我只提供四張圖片，編號 0 號為草地，以它做為預設地形；物品圖片更少，只有一顆足球，放在編號 1 的位置上，因為編號 0 具有特殊意義，代表「此地無任何物品」。

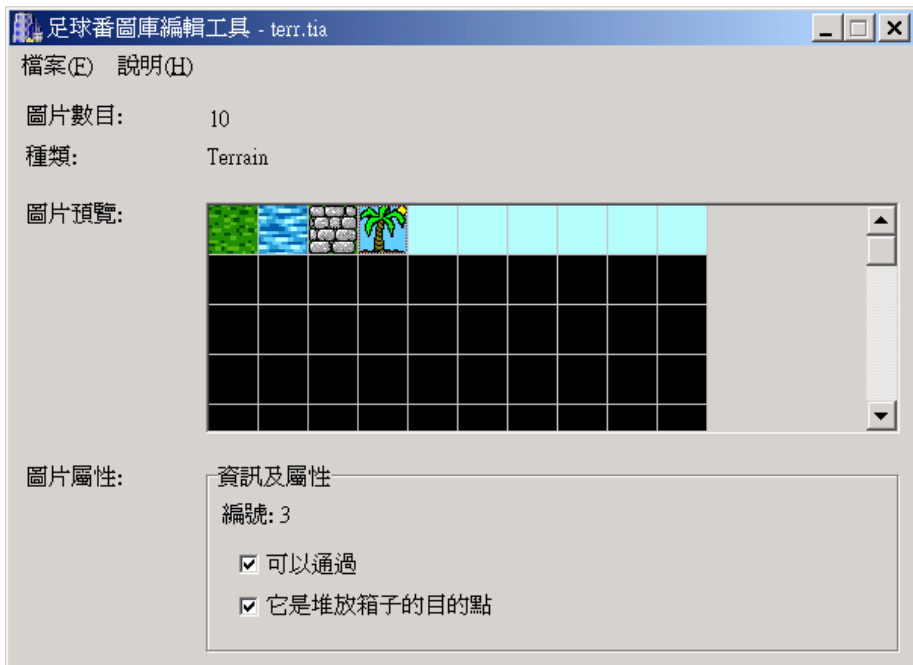


圖 8-10 / 圖庫編輯器的執行畫面（地形圖庫）



圖 8-11 / 圖庫編輯器的執行畫面（物品圖庫）

事實上這大概是全世界最陽春的圖庫編輯器了，跟圖 8-2 及圖 8-3 列出的 StarCraft 及英雄無敵 III 的地圖編輯器一比，咱們的圖庫編輯器羞得無地自容，差點離家出走，還是我好說歹說才將它留住。我想，就算是第二陽春的圖庫編輯器至少也有圖片拉曳、更換位置編號等功能，再者圖片群組及物件的概念也該支援，可以發揮的地方還多得是，讓我們以後慢慢玩吧。

地圖編輯器

有了圖庫編輯器，製作出圖庫後，接著就可以編輯地圖。地圖編輯器的目的很簡單－提供一個 WYSIWYG 的介面讓使用者可以方便地編輯存放那兩層地圖的二維陣列元素值。呵，很繞口吧。

說得清楚點，*TMap* 不是擁有兩個 *TMapArray* 型態的 *TerrMap* 及 *ItemMap* 變數嗎？*TMapArray* 型態是 $TILE_NUM_X * TILE_NUM_Y$ 大小的二維 *Byte* 陣列，而地圖編輯器的目的就是提供與遊戲進行時相同的圖片及畫面以及方便的操作介面，供關卡設計者編輯陣列內容。

目的很簡單，說來只有幾句話，但寫來比我們那個世界第一笨的圖庫編輯器還稍微複雜一滴滴（以程式碼行數比較的話:p）。還是先將介面設計出來：

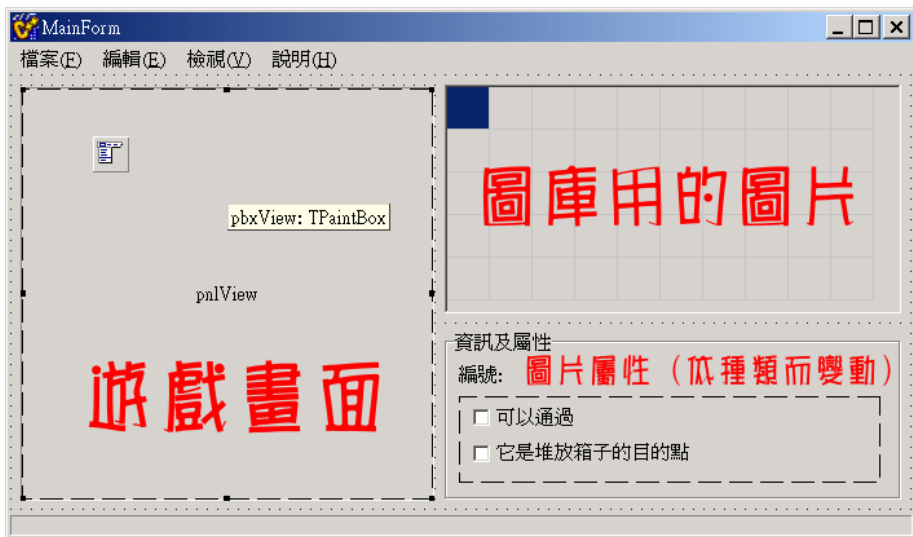


圖 8-12 / 地圖編輯器的設計畫面

程式規劃如下：

1. 有三種編輯模式，分別是地形，物品及角色編輯模式。
2. 與遊戲畫面不同的是，可以顯示格線及特殊區域以利編輯。
3. 能夠檢查關卡地形及物品擺設是否合法。

首先宣告編輯模式型別及視窗類別 *TMainForm*：

```

#0001 // 三種編輯模式：地形，物品及主角
#0002 enum TEditKind {ekTerrs, ekItems, ekRole};
#0003
#0004 class TMainForm : public TForm
#0005 {
#0006 ...
#0007 private:
#0008     TEditKind FEditKind; // 目前編輯模式
#0009     int FLevelNo; // 目前編輯的關卡
#0010     bool FModified; // 載入關卡後是否更動過
#0011
#0012     TRole FRole; // 角色物件
#0013     // double-buffering 用的背景 bitmap
#0014     Graphics::TBitmap* FBackBitmap;
#0015
#0016     int FCursorX, FCursorY; // 滑鼠游標位置
#0017     TMouseButton FButtonPressed; // 滑鼠按鍵狀態
#0018
#0019     void __fastcall DrawBackBitmap(); // 繪製背景 bitmap
#0020     void __fastcall UpdateView(); // 更新編輯畫面
#0021     void __fastcall UpdateControlStatus();
#0022
#0023     void __fastcall SetLevelNo(int Value);
#0024
#0025     void __fastcall MySaveMap(); // 儲存地圖檔案
#0026     bool __fastcall AskSaveMap(); // 確認是否儲存地圖
#0027     bool __fastcall ValidateMap(); // 檢查地圖是否合法
#0028
#0029     __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0030 ...
#0031 };

```

0012 列宣告了 *TRole* 物件 *FRole*，這是 *TRole* 類別第一回派上用場呢。不過在此只是虛晃幾招，只用它來做顯示及定位角色的用途而已，那複雜的移動邏輯仍派不上用場。

0014 列的 *FBackBitmap* 就是文章前頭談到 double-buffering 時所說的背景 bitmap。在 *UpdateView* 函式中，會先呼叫 *DrawBackBitmap* 將應該出現在畫面上的所有東東繪製於 *FBackBitmap*，再呼叫 *TCanvas::Draw* 函式一口氣將 *FBackBitmap* 畫上 *pbxView*，也就是代表編輯畫面的 *TPaintBox* 畫布上頭。所以 *UpdateView* 函式只有簡單的短短兩行：

```

#0001 void __fastcall TMainForm::UpdateView()
#0002 {
#0003     DrawBackBitmap(); // 將畫面放到 FBackBitmap

```

```
#0004  pbxView->Canvas->Draw(0, 0, FBackBitmap); // 複製到 pbxView
#0005  }
```

在地圖編輯器中，因為必須同時使用地形圖庫及物品圖庫，因此就不像圖庫編輯器那樣，兩者共用一個 *TTiles* 變數，而直接使用宣告在 *TileUnit* 單元中的全域變數 *Terrs* 及 *Items*。

程式初始化

在 *OnCreate* 事件處理函式中，初始化所有的物件，並載入角色圖片及地形、物品圖庫，另外還設定好 *FBackBitmap*，讓它跟編輯畫面一樣大：

```
#0001  void __fastcall TMainForm::FormCreate(TObject *Sender)
#0002  {
#0003      FEditKind = ekTerrs; // 預設為地形編輯模式
#0004      FButtonPressed = mbMiddle; // 表示目前滑鼠鍵沒有按著
#0005
#0006      try {
#0007          FRole.LoadBits(); // 讀入主角的圖形
#0008
#0009          Terrs.LoadFromFile(AppDir + FN_TERR_ARCHIVE); // 讀入地形
#0010          Items.LoadFromFile(AppDir + FN_ITEM_ARCHIVE); // 讀入物品
#0011      } catch (Exception& E) {
#0012          ShowMessage(E.Message);
#0013          // 以 asynchronous 方式關閉視窗
#0014          PostMessage(Handle, WM_CLOSE, 0, 0);
#0015      }
#0016
#0017      // According tile num and size, adjust dimension of pnlView
#0018      pnlView->ClientWidth = TILE_WIDTH * TILE_NUM_X;
#0019      pnlView->ClientHeight = TILE_HEIGHT * TILE_NUM_Y;
#0020
#0021      FBackBitmap = new Graphics::TBitmap; // 緩衝用 bitmap
#0022      FBackBitmap->Width = TILE_WIDTH * TILE_NUM_X;
#0023      FBackBitmap->Height = TILE_HEIGHT * TILE_NUM_Y;
#0024
#0025      LevelNo = 1;
#0026      UpdateControlStatus();
#0027
#0028      // 滑鼠游標位置
#0029      FCursorX = -1;
#0030      FCursorY = -1;
#0031  }
```

0009 及 0010 列載入圖庫檔案時所用的 *AppDir* 字串記錄著程式執行檔所在的目錄，由程式庫的 *xFiles* 單元提供。

0021 ~ 0023 列建立 *double-buffering* 用的背景 *bitmap* *FBackBitmap*，並將大小設定與遊戲畫面相同，這使得 *UpdateView* 函式中，將 *FBackBitmap* 內容複製到編輯畫面 *pbxView* 的動作省事許多。

奇妙的 *LevelNo* 屬性

LevelNo 是整數型態的屬性，每當設定新值時，就會呼叫 *SetLevelNo* 函式去設定（見類別宣告 0029 列）：

```
#0001 void __fastcall TMainForm::SetLevelNo(int Value)
#0002 {
#0003     try {
#0004         // 讀取地圖檔及角色位置
#0005         Map.LevelNo = Value;
#0006     } catch(...) {
#0007         // 若讀取地圖檔失敗，清除整張地圖
#0008         Map.ResetMap();
#0009     }
#0010
#0011     FRole.X = Map.Role_X; // 將角色位置由 Map 物件中抄出來
#0012     FRole.Y = Map.Role_Y;
#0013
#0014     FLevelNo = Value;
#0015     FModified = False; // 地圖尚未更動（當然:p）
#0016     UpdateControlStatus();
#0017     UpdateView(); // 別忘了更新遊戲畫面
#0018 }
```

這就是我為什麼喜歡使用屬性的原因。瞧 *FormCreate* 函式中簡簡單單的一道敘述，將 *LevelNo* 設為 1，事實上它的屬性寫入函式—*SetLevelNo* 就會為我讀取第一關的地圖出來，同時取得角色位置，設定其它變數，並且更新編輯畫面等等，多麼優雅的撰寫方式呀。屬性機制讓所有讀／寫的邊際效應都漂亮地隱藏在屬性值讀／取的簡單敘述背後。

LevelNo 屬性甚至可以這樣使用：


```
#0001 void __fastcall TMainForm::mnuRestoreLevelClick(TObject *Sender)
#0002 {
#0003     // 會觸發 property write method (SetLevelNo)
#0004     LevelNo = LevelNo;
#0005 }
```

mnuRestoreLevelClick 是選擇【恢復此關卡原狀】選項的事件處理函式，看到上面那行程式碼，不曉得 *LevelNo* 是屬性的傢伙一定會認為我花轟了，竟然把一個變數指派給自己，只是浪費 CPU 時間的無意義動作呀。但是別忽略隱藏在 *LevelNo* 屬性背後的 *SetLevelNo* 函式，將 *LevelNo* 指派給 *LevelNo* 的結果是，*SetLevelNo* 函式會去重新載入目前的關卡地圖，達成「恢復此關卡原狀」的目的，很特別吧。

繪製編輯畫面

前頭剛提過，呈現編輯畫面的核心函式只有一個—*DrawBackBitmap*，它先繪製地形層，接著物品層，最後是角色。可以想像出，待會才要進行的遊戲主程式的 *DrawBackBitmap* 函式，似乎也只需要這三個動作就夠了。但在地圖編輯器中，還要能夠顯示格線及特殊區域，所以繪製物品層後，繪製角色前，另外加上三段程式碼，分別將格線，目的地形及覆蓋用物品標示出來：

```
#0001 void __fastcall TMainForm::DrawBackBitmap()
#0002 {
#0003     Map.DrawTerrMap(FBackBitmap->Canvas); // 繪製地形層
#0004     Map.DrawItemMap(FBackBitmap->Canvas); // 繪製物品層
#0005
#0006     // 如果使用者想看格線，就將格線畫出來
#0007     if (mnuShowGrid->Checked) {
#0008         FBackBitmap->Canvas->Pen->Color = clBlack;
#0009         FBackBitmap->Canvas->Pen->Style = psSolid;
#0010         FBackBitmap->Canvas->Pen->Width = 1;
#0011
#0012         // 先畫橫線
#0013         for (int y = 0; y < TILE_NUM_Y; y++) {
#0014             FBackBitmap->Canvas->MoveTo(0, y * TILE_HEIGHT);
#0015             FBackBitmap->Canvas->LineTo(TILE_NUM_X * TILE_WIDTH,
#0016                 y * TILE_HEIGHT);
#0017         }
#0018 }
```

```

#0019 // 再畫直線
#0020 for (int x = 0; x < TILE_NUM_X; x++) {
#0021     FBackBitmap->Canvas->MoveTo(x * TILE_WIDTH, 0);
#0022     FBackBitmap->Canvas->LineTo(x * TILE_WIDTH,
#0023         TILE_NUM_Y * TILE_HEIGHT);
#0024 }
#0025 }
#0026
#0027 // 如果使用者想看特殊區域，將目的地地形標出來
#0028 if (mnuShowSpeicalArea->Checked) {
#0029     FBackBitmap->Canvas->Pen->Color = clRed;
#0030     FBackBitmap->Canvas->Pen->Width = 1;
#0031     FBackBitmap->Canvas->Pen->Style = psDash;
#0032     FBackBitmap->Canvas->Brush->Style = bsClear;
#0033
#0034     for (int y = 0; y < TILE_NUM_Y; y++) // 逐一檢查
#0035         for (int x = 0; x < TILE_NUM_X; x++)
#0036             if (Map.IsTarget[x][y]) { // 外框加上右上畫到左下的紅色虛線
#0037                 FBackBitmap->Canvas->Rectangle(x * TILE_WIDTH,
#0038                     y * TILE_HEIGHT, (x + 1) * TILE_WIDTH - 1,
#0039                     (y + 1) * TILE_HEIGHT - 1);
#0040                 FBackBitmap->Canvas->MoveTo((x + 1) * TILE_WIDTH,
#0041                     y * TILE_HEIGHT);
#0042                 FBackBitmap->Canvas->LineTo(x * TILE_WIDTH,
#0043                     (y + 1) * TILE_HEIGHT);
#0044             }
#0045     }
#0046
#0047 // 如果使用者想看特殊物品，將覆蓋用物品標出來
#0048 if (mnuShowSpeicalItems->Checked) {
#0049     FBackBitmap->Canvas->Pen->Color = clLime;
#0050     FBackBitmap->Canvas->Pen->Width = 1;
#0051     FBackBitmap->Canvas->Pen->Style = psDash;
#0052     FBackBitmap->Canvas->Brush->Style = bsClear;
#0053
#0054     for (int y = 0; y < TILE_NUM_Y; y++) // 逐一檢查
#0055         for (int x = 0; x < TILE_NUM_X; x++)
#0056             if (Map.IsSource[x][y]) { //
#0057                 外框加上左上畫到右下的亮綠色虛線
#0058                 FBackBitmap->Canvas->Rectangle(x * TILE_WIDTH,
#0059                     y * TILE_HEIGHT, (x + 1) * TILE_WIDTH - 1,
#0060                     (y + 1) * TILE_HEIGHT - 1);
#0061                 FBackBitmap->Canvas->MoveTo(x * TILE_WIDTH,
#0062                     y * TILE_HEIGHT);
#0063                 FBackBitmap->Canvas->LineTo((x + 1) * TILE_WIDTH,
#0064                     (y + 1) * TILE_HEIGHT);

```

```

#0065      }
#0066    }
#0067
#0068    FRole.Draw(FBackBitmap->Canvas); // 最後畫出角色圖案
#0069  }

```

你可以先偷偷看一下圖 8-13，看看這段程式碼繪出的畫面長得什麼樣子。當然囉，可以獨立開關這三個顯示選項，預設值為關，所以你自己執行時不會看到格線及特殊區域，選取功能表將它們打開後才看得到。

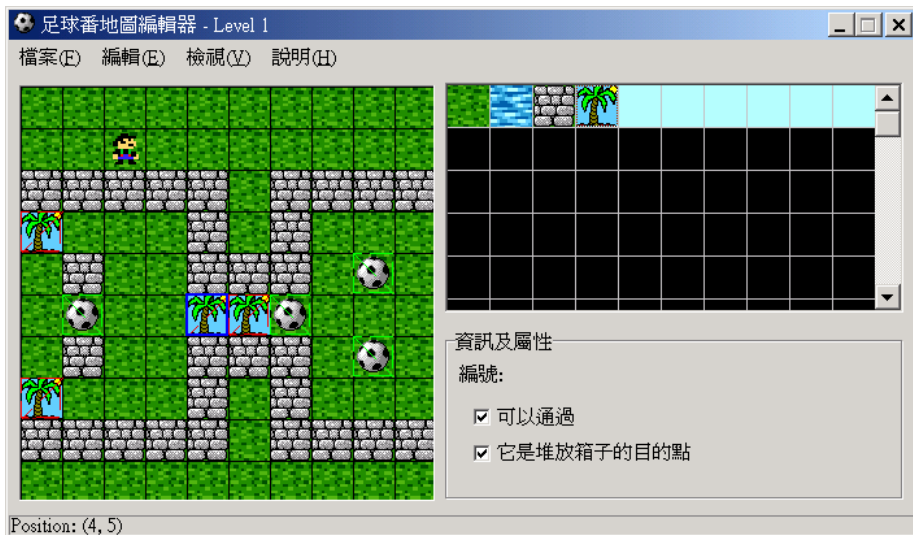


圖 8-13 / 地圖編輯器的執行畫面（將三個顯示選項都打開了）

右上方顯示圖庫圖片的 *grdView* 及右下角顯示圖片屬性的 *TCheckBox* 都跟圖庫編輯器中一樣，因此不再贅述。

選取【編輯 / (地形、物品、人物)】時，會觸發 *mnuEditRoleClick* 函式，首先將觸發此函式的 *TMenuItem* 元件打勾（即 *Checked* 屬性設為 *true*），設定 *FEditKind* 的新值，此處我又使用前述的函式，將三個 *TMenuItem* 元件的 *OnClick* 事件全指派到同一個事件處理函式，以 *Tag* 屬性區分彼此。所以 0009 列只要輕鬆的一行指派敘述，到 *Kinds* 陣列中查表，就可以將 *FEditKind* 設定為對應的編輯模式。此函式的其它部分都在處理切換編輯模式時控制項的介面差異問題（顯示顏色，控制項是否致能等等）。

```
#0001 void __fastcall TMainForm::mnuEditKindClick(TObject *Sender)
#0002 {
#0003     const TEditKind Kinds[3] = {ekTerrs, ekItems, ekRole};
#0004     const TColor Colors[2] = {clBtnFace, clWindow};
#0005
#0006     TMenuItem* item = dynamic_cast<TMenuItem*>(Sender);
#0007
#0008     item->Checked = true;
#0009     FEditKind = Kinds[item->Tag]; // 設定選取的編輯模式
#0010
#0011     UpdateControlStatus();
#0012     UpdateView();
#0013
#0014     ...
#0015 }
```

滑鼠的拉曳及物品置放

真正的好菜現在上桌，不論在何種編輯模式下，當滑鼠游標在編輯畫面範圍內時，游標底下對映的圖格就會有個水藍色的框框跟著它跑，這是怎麼做到的呢？這效果來自於 *grdView* 的 *OnMouseMove* 事件處理函式 *grdViewMouseMove*：

```
#0001 void __fastcall TMainForm::pbxViewMouseMove(TObject *Sender,
#0002     TShiftState Shift, int X, int Y)
#0003 {
#0004     // 已經跑出範圍外了...
#0005     if (X < 0 || X >= TILE_WIDTH * TILE_NUM_X || Y < 0 ||
#0006         Y >= TILE_HEIGHT * TILE_NUM_Y) return;
#0007
#0008     FCursorX = X / TILE_WIDTH; // 換算座標，以圖格為單位
#0009     FCursorY = Y / TILE_HEIGHT;
#0010
#0011     // 一邊拉曳一邊置放物品的效果
#0012     pbxView->OnMouseDown(Sender, FButtonPressed, Shift, X, Y);
#0013
#0014     UpdateView(); // 重繪遊戲畫面
#0015
#0016     // 畫出目前所選取的區域外框
#0017     pbxView->Canvas->Pen->Width = 2;
#0018     pbxView->Canvas->Pen->Color = clBlue;
#0019     pbxView->Canvas->Brush->Style = bsClear;
#0020     pbxView->Canvas->Rectangle(FCursorX * TILE_WIDTH,
#0021         FCursorY * TILE_HEIGHT, (FCursorX + 1) * TILE_WIDTH,
```

```

#0022     (FCursorY + 1) * TILE_HEIGHT);
#0023
#0024     stbMain->SimpleText = Format("Position: (%d, %d)",
#0025     OPENARRAY(TVarRec, (FCursorX, FCursorY)));
#0026 }

```

每當滑鼠指標在控制項上移動時，就會不斷產生 *OnMouseMove* 事件。那麼，你也許會問，既然如此，那麼滑鼠指標一定是在控制項範圍內呀，又何必要有 0005 ~ 0006 列的範圍檢查碼呢？原因是，若使用者在控制項內按下滑鼠任一鍵然後「拉曳」的話，此控制項就會不斷地收到 *OnMouseMove* 事件，不論滑鼠指標是否早已移出控制項範圍，直到滑鼠鍵放開，所以 0005 ~ 0006 列的範圍檢查是必要的。

0008 ~ 0009 列將座標值由以像素為單位換算為以圖格為單位。0012 列是爲了達成一邊拉曳滑鼠一邊置放物品的效果，主動觸發 *pbxView* 的 *OnMouseDown* 事件處理函式以設定或清除圖格。接著，畫出所選取區域的外框。藍色外框是仿英雄無敵 III 地圖編輯器而來的，我覺得挺顯眼好看。

接著介紹最重要的一個，也就是達成地圖編輯器目的的重要函式－修改地圖內容的 *pbxView OnMouseDown* 事件處理函式：

```

#0001 void __fastcall TMainForm::pbxViewMouseDown(TObject *Sender,
#0002     TMouseButton Button, TShiftState Shift, int X, int Y)
#0003 {
#0004     // 將按下的按鍵記錄起來，配合 OnMouseMove event handler
#0005     // 產生拉曳設定效果
#0006     FButtonPressed = Button;
#0007
#0008     if (Button == mbMiddle) return; // 滑鼠中鍵不做任何事
#0009
#0010     // 計算目前選擇的圖片編號
#0011     int No = grdPreview->Row * grdPreview->ColCount + grdPreview->Col;
#0012
#0013
#0014     if (Button == mbLeft) { // 左鍵是設定
#0015         switch (FEditKind) { // 根據編輯模式不同進行不同的設定動作
#0016             case ekTerrs:
#0017                 Map.TerrMap[FCursorX][FCursorY] = No; // 將新地形擺上
#0018                 break;
#0019
#0020             case ekItems:

```

```

#0021         // 物品不可擺在角色身上
#0022         if (FRole.X == FCursorX && FRole.Y == FCursorY) return;
#0023
#0024         Map.ItemMap[FCursorX][FCursorY] = No; // 將新物件擺上
#0025         break;
#0026
#0027     case ekRole:
#0028         // 角色不可以擺在物品上
#0029         if (Map.ItemMap[FCursorX][FCursorY] != 0) return;
#0030         // 角色不可以擺在不可走動的地形上
#0031         if (!Map.CanPass[FCursorX][FCursorY]) return;
#0032
#0033         FRole.X = FCursorX; // 設定角色位置
#0034         FRole.Y = FCursorY;
#0035         break;
#0036     }
#0037 } else { // 右鍵是清除
#0038     switch (FEditKind) {
#0039     case ekTerrs: Map.TerrMap[FCursorX][FCursorY] = 0; // 清除地形
#0040                 break;
#0041
#0042     case ekItems: Map.ItemMap[FCursorX][FCursorY] = 0; // 清除物品
#0043                 break;
#0044
#0045     case ekRole: // 角色不能清掉, so do nothing
#0046                 break;
#0047     }
#0048 }
#0049
#0050 FModified = true;
#0051 UpdateView();
#0052 }

```

這段程式碼的註解相當清楚，邏輯也十分簡單，首先判斷使用者按下的是左鍵或右鍵，左鍵代表置放，右鍵代表清除。接著再根據目前的編輯模式，置放地形、物品、角色或是清除地形或物品。

唯一要注意的就是置放物品及角色前，要小心會不會讓地圖產生不合法，或是遊戲無法進行的窘況，例如角色不可擺在物品上，也不可擺在無法穿越的地形上這類的合法性檢查。

FButtonPressed 變數是用來實作拉曳設定效果的關鍵處，即是你可以按著滑鼠左鍵隨意

在遊戲畫面上游走，經過之處就會擺上目前選擇的地形或物品，編輯起來爽快多了。*FButtonPressed* 是 *TMouseButton* 列舉型態，其值可能為 *mbLeft*、*mbRight*、*mbMiddle* 三者之一，分別代表滑鼠左鍵、右鍵及中鍵。在此利用 *FButtonPressed* 記錄著使用者目前按下的滑鼠鍵，原本還須用一個布林變數來記錄目前是否真正按著滑鼠鍵，但因為我們沒用到中鍵，所以設定當 *FButtonPressed* 為 *mbMiddle* 時，就代表滑鼠鍵沒有按著，其值為 *mbLeft* 或 *mbRight* 時，才代表滑鼠左鍵或右鍵正被按壓著。

擁有 *FButtonPressed* 資訊後，就可以在 *OnMouseMove* 事件處理函式中，主動呼叫 *OnMouseDown* 的事件處理函式，設定或清除對應的圖格。哦對了，別忘了撰寫 *OnMouseUp* 事件處理函式，在滑鼠鍵放開時，將 *FButtonPressed* 設為 *mbMiddle*：

```
void __fastcall TMainForm::pbxViewMouseUp(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    // 設定為 mbMiddle, 表示使用者已放開滑鼠鍵
    FButtonPressed = mbMiddle;
}
```

關卡合法性檢查

地圖編輯器還有一項重要的功能，就是在關卡設計完成後，儲存入檔案前，檢查地圖的合法性，覆蓋用物品及目的地形數目，角色是否位於不能移動的地形上等等，合法的關卡才能順利地進行遊戲。檢查地圖合法性的 *ValidateMap* 函式程式碼如下：

```
#0001 // 檢查地圖是否合法
#0002 bool __fastcall TMainForm::ValidateMap()
#0003 {
#0004     int x, y;
#0005     int SourceCount, TargetCount;
#0006
#0007     SourceCount = 0; // 覆蓋用物品數目
#0008     TargetCount = 0; // 目的地形數目
#0009     for (int y = 0; y < TILE_NUM_Y; y++)
#0010         for (int x = 0; x < TILE_NUM_X; x++) {
#0011
#0012             // 計算覆蓋用物品及目的地形數目
#0013             if (Map.IsSource[x][y]) SourceCount++;
```

```
#0014     if (Map.IsTarget[x][y]) TargetCount++;
#0015
#0016     // 此格地圖中記錄的圖片編號是不是大於圖庫的圖片數目？
#0017     // 是的話就調回預設值
#0018     if (Map.TerrMap[x][y] > Terrs.TileNum) Map.TerrMap[x][y] = 0;
#0019     if (Map.ItemMap[x][y] > Items.TileNum) Map.ItemMap[x][y] = 0;
#0020     }
#0021
#0022     // 是否沒有目的地形？（無法過關）
#0023     if (TargetCount == 0 &&
#0024         !YesNoBox("沒有目的地形，是否繼續？", MB_DEFBUTTON1))
#0025         return false;
#0026
#0027     // 是否覆蓋用物品少於目的地形？（無法過關）
#0028     if (TargetCount > SourceCount &&
#0029         !YesNoBox("覆蓋用物品少於目的地形，是否繼續？", MB_DEFBUTTON1))
#0030         return false;
#0031
#0032     // 角色位於不能移動的地形上
#0033     if (!Map.CanPass[Map.Role_X][Map.Role_Y] &&
#0034         !YesNoBox("主角位於不能移動的地形上，是否繼續？", MB_DEFBUTTON1))
#0035         return false;
#0036
#0037     return true;
#0038 }
```

至此，地圖編輯器也順利完工，讓我們再看一次執行畫面，這回三個特殊顯示選項沒有打開。嗯，遊戲畫面也可從這兒的編輯畫面看出大概了，是不是對即將完成的遊戲更充滿期待呢？



圖 8-14 / 地圖編輯器的執行畫面

明明是倉庫番類型的遊戲，不過物品圖庫竟然沒有箱子的蹤影，嘻，這是因為我找不到箱子的圖片，只好以足球代替，這也是這套遊戲之所以稱為「足球番」的原因。:p 無論如何，這套「足球番」已呼之欲出，加把勁就要完成了，休息一下，咱們繼續。

「足球番」主程式

一路過關斬將，砍了圖庫編輯器，宰掉地圖編輯器，最後來到大魔王－「足球番」主程式前...

老法子，先在將視窗介面設計好。在設計時期看起來，這遊戲主程式比前兩支程式都還簡單，因為只有一個 *TPaintBox* 元件 *pbxView*，上面再放著一個 *TMainMenu* 元件及三個計時器元件而已。



圖 8-15 / 「足球番」主程式的設計畫面

在下列的 *TMainForm* 類別宣告中，0002 列首先宣告 *TGameStatus* 型態，定義四種狀態，分別是「歡迎畫面」、「遊戲進行中」、「關卡完成後的慶祝畫面」及「過關動作回顧」（即重播功能）。有許多變數，物件及函式的命名及含意都與地圖編輯器一模一樣，如代表角色的 *FRole*，擔任 double-buffering 緩衝區的 *FBackBitmap*，繪製背景 bitmap 的 *DrawBackBitmap* 函式及更新遊戲畫面的 *UpdateView* 函式等等。比較新鮮的是存放目前遊戲狀態的 *FGameStatus* 變數，記錄可以播放過關回顧關卡編號的 *FPlayBackLevelNo* 變數，可在遊戲畫面中央或正上方畫出字串及方框的 *DrawStatusBox* 函式，以及檢查是否已完成任務的 *CheckFinished* 函式等。類別宣告的原始程式碼列表如下：

```
#0001 // 有四種狀態, Title 畫面, 遊戲中, 完成某關卡及過關回顧
#0002 enum TGameStatus {gsTitle, gsPlaying, gsSuccess, gsPlayBack};
#0003
#0004 class TMainForm : public TForm
#0005 {
#0006 ...
#0007 private:
#0008     TGameStatus FGameStatus; // 遊戲狀態
#0009     int FLevelNo, FPlayingTime; // 目前關卡及遊戲進行時間
#0010     TRole FRole; // 角色物件
#0011
#0012     // double-buffering 用的背景 bitmap
```

```

#0013     Graphics::TBitmap* FBackBitmap;
#0014     int FPlayBackLevelNo; // 可以 playback 的 LevelNo
#0015
#0016     void __fastcall DrawBackBitmap(); // 繪製背景 bitmap
#0017     void __fastcall UpdateView(); // 更新編輯畫面
#0018     void __fastcall UpdateControlStatus();
#0019
#0020     // 在遊戲畫面中央或正上面繪出字串及外圍方框
#0021     void __fastcall DrawStatusBox(AnsiString S, bool TopOrCenter);
#0022     bool __fastcall CheckFinished(); // 檢查是否已完成任務
#0023
#0024     void __fastcall SetGameStatus(TGameStatus Value);
#0025     void __fastcall SetLevelNo(int Value);
#0026
#0027     __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0028     __property TGameStatus GameStatus =
#0029         {read = FGameStatus, write = SetGameStatus};
#0030     ...
#0031 };

```

TMainForm 的 *OnCreate* 事件處理函式與地圖編輯器中幾乎完全一樣: 同樣地建立及初始化 *TRole*、*TMap*、*TTiles* 物件及緩衝用 *bitmap* 等等, 並沒有新的工作。

三個小時鐘

而丟在 *form* 上的三個 *TTimer* 計時器元件, 它們的任務是什麼呢?

- *tmrTitle*
負責在歡迎畫面時, 讓角色飛快地胡亂移動, 帶來爆笑效果, 觸發間隔設為 50 毫秒。
- *tmrTimer*
為遊戲進行中的計時器, 每 1000 毫秒, 即一秒鐘觸發一次, 同時更新螢幕上的時鐘, 讓使用者曉得此關卡已花費多少時間。
- *tmrPlayback*
重播過關記錄時, 每 500 毫秒觸發一次, 讓角色每半秒鐘根據先前的動作記錄移動到下一步, 若不使用計時器來放慢速度, 直接用迴圈來播放, 幾百步驟的動作記錄可能一眨眼就播完了:p

它們的觸發事件處理函式分別如下：

```
#0001 void __fastcall TMainForm::tmrTimerTimer(TObject *Sender)
#0002 {
#0003     FPlayingTime++; // 遊戲進行時間加一秒
#0004     UpdateView(); // 強制更新畫面
#0005 }
#0006
#0007 void __fastcall TMainForm::tmrTitleTimer(TObject *Sender)
#0008 {
#0009     FRole.Move((TDirection)random(4)); // 讓主角任意走動
#0010     UpdateView(); // 更新畫面
#0011 }
#0012
#0013 void __fastcall TMainForm::tmrPlayBackTimer(TObject *Sender)
#0014 {
#0015     // 按照之前的動作循序走動，使用 tmrPlayBack 的 Tag
#0016     // 屬性來記錄目前走到第幾步
#0017     TDirectionArray& plist = FRole.PlayBackList;
#0018     FRole.Move(plist[tmrPlayBack->Tag]);
#0019
#0020     // 這一步走過了，遞增至下一步
#0021     tmrPlayBack->Tag = tmrPlayBack->Tag + 1;
#0022
#0023     // 全部播完了，過關畫面
#0024     if ((unsigned int)tmrPlayBack->Tag == FRole.PlayBackList.size())
#0025         GameStatus = gsSuccess;
#0026
#0027     UpdateView(); // 更新畫面
#0028 }
```

tmrTimer 觸發時只要遞增 *FPlayingTime*，並且強制更新畫面，*DrawBackBitmap* 函式就會根據 *FPlayingTime* 的值將此關卡已花費時間畫在右上角。

tmrTitle 觸發時十分放心地呼叫 *random* 函式取得上下左右任一方向，接著呼叫 *TRole::Move* 函式將角色移向亂數取得的方向，有點不知死活的樣子，不過這樣子不會出問題，因為我們的移動碰撞檢查碼都放在 *Move* 函式裏，所以若檢查為不合法的移動，角色就會留在原地不動，這樣一來，因為移動方向有時合法有時不合法，還可模擬出時走時停的效果呢。

tmrPlayBackTimer 的觸發事件處理函式中，將已播放的步數存在它本身的 *Tag* 屬性，然

後經由 *TDirectionArray* 物件 *PlaybackList* 查得目前這一步的走法。0022 列檢查，若是全部移動記錄播放完畢，則立即進入過關畫面，反正我們只有使用者過關後才會將移動記錄保留下來，因此播放的一定是過關走法，所以移動記錄全部播完畢時一定正好過關。

遊戲狀態的初始化

這些計時器由 *GameStatus* 屬性的屬性寫入函式來控制啟動狀態，*SetGameStatus* 任務重大，負責在進入各個遊戲狀態時，開關這三個計時器，並分別將該狀態所需的變數或物件初始化：

```
#0001 void __fastcall TMainForm::SetGameStatus(TGameStatus Value)
#0002 {
#0003     FGameStatus = Value;
#0004
#0005     // 根據新的遊戲狀態開關三個計時器
#0006     tmrTimer->Enabled = FGameStatus == gsPlaying;
#0007     tmrPlayBack->Enabled = FGameStatus == gsPlayBack;
#0008     tmrTitle->Enabled = FGameStatus == gsTitle;
#0009
#0010     switch (FGameStatus) {
#0011     case gsTitle: LevelNo = 1; // 歡迎畫面顯示第一關地圖
#0012         break;
#0013
#0014     case gsPlaying:
#0015         FPlayingTime = 0; // 計時歸零
#0016         FRole.CleanMoveList(); // play back 歸零
#0017         break;
#0018
#0019     case gsSuccess:
#0020         // 過關了，將關卡記錄起來，表示要重播時就回此關卡
#0021         FPlayBackLevelNo = FLevelNo;
#0022         break;
#0023
#0024     case gsPlayBack:
#0025         LevelNo = FPlayBackLevelNo; // 切換到記錄重播回顧的關卡
#0026         tmrPlayBack->Tag = 0; // 從第一步開始播放
#0027         break;
#0028     }
#0029
#0030     UpdateControlStatus(); // 更新標題列及其它控制項
```

```
#0031  UpdateView(); // 更新遊戲畫面
#0032  }
```

因為三個計時器只有分別在歡迎畫面，遊戲中，及重播過關回顧時時才須啟動，因此 0006 ~ 0008 列根據新的遊戲狀態設定它們的 *Enabled* 屬性，同一時間最多只有一個計時器為啟動狀態。0025 列，進入重播過關回顧狀態時，必須主動載入記錄重播回顧的關卡（同時會將角色擺在初始位置），如此才可順利進行重播，要不然若目前處於第一關地圖，而動作記錄是第二關記下來的，就會看到主角到處碰壁，亂走一通的蠢模樣。

設定 *LevelNo* 屬性，也就是間接呼叫 *SetLevelNo* 函式時，裏頭再呼叫 *TMap::LevelNo* 來載入關卡：

```
#0001  void __fastcall TMainForm::SetLevelNo(int Value)
#0002  {
#0003      Map.LevelNo = Value;
#0004      FRole.X = Map.Role_X;
#0005      FRole.Y = Map.Role_Y;
#0006
#0007      FLevelNo = Value;
#0008      UpdateControlStatus();
#0009      UpdateView();
#0010  }
```

唯一要特別注意的是，載入關卡後，千萬別忘了將角色的初始位置由 *Map* 物件的 *Role_X* 及 *Role_Y* 屬性中讀出，更新 *FRole* 的位置。

繪製遊戲畫面

有了這些幕後工作人員控制著流程，在分工清楚的前提下，前景的演員只要盡守本分，依照模式好好地繪製遊戲畫面就夠了。下列是畫出遊戲畫面的 *DrawBackBitmap* 函式：

```
#0001  void __fastcall TMainForm::DrawBackBitmap()
#0002  {
#0003      Map.DrawTerrMap(FBackBitmap->Canvas); // 繪製地形層
#0004      Map.DrawItemMap(FBackBitmap->Canvas); // 繪製物品層
#0005      FRole.Draw(FBackBitmap->Canvas); // 畫出角色圖案
#0006
#0007      int M = FPlayingTime / 60; // 已花費時間的分鐘數
```

```

#0008     int S = FPlayingTime % 60; // 已花費時間的秒鐘數
#0009
#0010     FBackBitmap->Canvas->Font->Color = clWhite;
#0011     FBackBitmap->Canvas->Font->Name = "FixedSys";
#0012     FBackBitmap->Canvas->Font->Style = TFontStyles() << fsBold;
#0013     FBackBitmap->Canvas->Font->Size = 14;
#0014     FBackBitmap->Canvas->Brush->Style = bsClear;
#0015
#0016     TRect R;
#0017     switch (FGameStatus) {
#0018     case gsTitle:
#0019         // 畫出上面的標題大字及下方的作者名稱
#0020         R = Rect(0, 0, TILE_WIDTH * TILE_NUM_X, TILE_HEIGHT *
#0021             TILE_NUM_Y - FBackBitmap->Canvas->TextHeight("我") / 2);
#0022         DrawText(FBackBitmap->Canvas->Handle, "作者: 陳寬達", - 1, &R,
#0023
#0024             DT_BOTTOM | DT_CENTER | DT_SINGLELINE);
#0025         DrawStatusBox("歡迎光臨 足球番", true);
#0026         break;
#0027
#0028     case gsPlayBack: FBackBitmap->Canvas->TextOut(5, 5,
#0029         Format("過關回顧 (LEVEL %d) ...",
#0030             OPENARRAY(TVarRec, (FLevelNo))));
#0031         break;
#0032
#0033     default:
#0034         // 在右上角顯示時間, 以 XX:XX 的格式顯示
#0035         AnsiString Str = Format("%.2d:%.2d",
#0036             OPENARRAY(TVarRec, (M, S)));
#0037         FBackBitmap->Canvas->TextOut(TILE_WIDTH * TILE_NUM_X -
#0038             FBackBitmap->Canvas->TextWidth(Str) - 5, 5, Str);
#0039
#0040         // 在左上角顯示關卡
#0041         Str = Format("LEVEL %d", OPENARRAY(TVarRec, (FLevelNo)));
#0042         FBackBitmap->Canvas->TextOut(5, 5, Str);
#0043         break;
#0044     }
#0045
#0046     // 這是過關畫面
#0047     if (FGameStatus == gsSuccess)
#0048         DrawStatusBox("哇, 成功了 !!", false);
#0049 }

```

0003 ~ 0005 列分別繪出地形、物品及角色，剩下的程式碼則根據目前的遊戲狀態，在畫面的不同區域畫出標題，時間，關卡及祝賀訊息等等。你會發現只要當初遊戲的狀態區

分的清楚無模糊地帶，顯示畫面的程式邏輯就會簡單的不得了，根據狀態顯示不同的訊息就一切 OK。

嗯，其實已經可以看到遊戲畫面，九牛只差一毛了。還差什麼？原來居十分關鍵地位的使用者輸入處理，不能讓玩家控制的遊戲就不能叫做遊戲，叫 DEMO 版本。:p

處理使用者輸入

第一個步驟，先將 *TMainForm* 的 *KeyPreview* 屬性設定為 *true*，這樣可以保證不論鍵盤輸入焦點位於哪個控制項上，*MainForm* 本身一定會第一個收到鍵盤事件，並且還可以進行過濾處理，讓控制項本身看不到鍵盤事件哩。接著為 *TMainForm* 的 *OnKeyDown* 事件撰寫處理函式，以上下左右四個方向鍵來控制角色的移動：

```
#0001 void __fastcall TMainForm::FormKeyDown(TObject *Sender, WORD &Key,
#0002     TShiftState Shift)
#0003 {
#0004     // 注意，只有遊戲中狀態，鍵盤控制才有效
#0005     if (FGameStatus == gsPlaying) {
#0006         switch (Key) {
#0007             case VK_UP: FRole.Move(drUp); break; // 向上走
#0008
#0009             case VK_DOWN: FRole.Move(drDown); break; // 向下走
#0010
#0011             case VK_LEFT: FRole.Move(drLeft); break; // 向左走
#0012
#0013             case VK_RIGHT: FRole.Move(drRight); break; // 向右走
#0014
#0015             default: return; // 亂按一通，不理它
#0016         }
#0017
#0018         UpdateView(); // 更新畫面
#0019
#0020         if (CheckFinished()) { // 是否完成任務 ??
#0021             FRole.SavePlayBack(); // 將行動記錄存起來以便重播
#0022             GameStatus = gsSuccess; // 進入過關狀態
#0023         }
#0024     }
#0025 }
```


首先注意只有在遊戲中狀態，鍵盤控制才有效。接下來就簡單啦，檢查按鍵是否為方向鍵，讓角色往對應的方向移動，如果不是方向鍵就離開，省得麻煩。角色更新後，呼叫 *CheckFinished* 函式檢查是否完成任務，是否已將所有目的地形利用覆蓋用物品掩住了？若是的話，就將行動記錄存起來以便重播，並且進入過關狀態。*CheckFinished* 檢查函式是這樣的：

```
#0001 bool __fastcall TMainForm::CheckFinished()
#0002 {
#0003     // 逐一檢查每個目的地形是否已被覆蓋用物品覆蓋住了？
#0004     for (int y = 0; y < TILE_NUM_Y; y++)
#0005         for (int x = 0; x < TILE_NUM_X; x++)
#0006             if (Map.IsTarget[x][y] && !Map.IsSource[x][y])
#0007                 return false; // 哦，有一個目的地形還沒被掩住，失敗 !!
#0008
#0009     return true; // Yeah, 任務達成
#0010 }
```

哇哈，它成功了，我們也成功了。再補上操作介面上的一些其它功能，遊戲主程式也完成了，迫不及待看看它的執行畫面：



圖 8-16 / 足球番的歡迎畫面



圖 8-17 / 足球番的遊戲中畫面一



圖 8-18 / 足球番的遊戲中畫面二



圖 8-19 / 足球番的過關畫面



圖 8-20 / 足球番的過關回顧畫面

圖片確實單調了些，關卡也是我隨手拉出來的，不要又打我呀，這些不是重點，遊戲寫出來才是重點咩。

你是否已有幾分感覺，撇開技術層面不談，遊戲設計與一般的程式設計其實沒有太大的

差異。只要別因為「撰寫遊戲」這四個字而興奮過頭，好好地訂立企劃，將程式中的類別、型態、模組規劃出來，再一步步慢慢兜，你將發現，遊戲程式的撰寫不但沒有想像中那麼難，所獲得的成就感還不是一般程式比得上的喲。

第九章

坦克大決戰

懷念古早時代的坦克大決戰遊戲嗎？
這紅白機時代經典遊戲的魅力，似乎至今未減。
既是人人愛玩的遊戲，又沒有太多的聲光特效，
玩得累了，手癢了，自己寫一套吧。



研一的暑假，可能也是學生生涯的最後一個暑假，帶著有假可玩直須玩的心理，期末考才剛結束就夥同幾位好友到南台灣度假遊玩。

停留墾丁的那幾天，當然不例外地來到凱撒大飯店地下一樓的星際碼頭玩玩虛擬實境遊戲及「360 度腳踏車」。沒想到，就在星際碼頭入口處的電玩遊樂場，赫然發現某部電玩主機提供的是 BATTLE CITY 遊戲，也就是「坦克大決戰」。如同幾十年沒見的老朋友，一看到它，我就愣愣地定在主機旁，兩眼直盯遊戲畫面，直到朋友們合力把我架走為止...

對於即將實作的第二個遊戲，原本心中是以「炸彈超人」為底的，因為它曾在我心中佔有極重要的地位，這留待後話。不料徵詢女友大人的意見時，她卻提出「坦克大決戰」的點子，沒想到她也在弟弟任天堂主機的 42 合 1 卡帶中玩過這遊戲，且還記憶頗深呢。再加上前陣子的一面之緣，充分感受到這紅白機時代經典遊戲的魅力，題目就這樣定下來了。

可是，這麼久以前的遊戲，細節規則忘得差不多，遊戲圖片也沒著落，得再想法找來玩玩，順便抓取圖形才行。於是我立刻到網路上下載任天堂模擬器，心想著只要再找到遊戲的 ROM 檔案即可。萬萬沒想到，我老早忘了它的英文名字叫「BATTLE CITY」，用「Tank」、「Tank War」、「坦克」、「坦克大決戰」等等字串為關鍵字找了老半天找不著，氣死我了。最後，花了一天一夜的時間，好不容易在一個 100 合 1 的 ROM 中找到它，好辛苦哪。

怕有讀者未曾玩過此遊戲或是跟我一樣，年紀一大就把古早的遊戲忘光光了，在這先大略介紹一下好了。

任天堂版坦克大決戰

下圖是任天堂版坦克大決戰執行畫面，很令人懷念吧。後來我才知道，原來住我樓上房

間的同学就是個現成的坦克大決戰高手，可以一隻玩到最後一關（三十五關），接著繼續進攻第二輪。一看到我抓下來的執行畫面，就立即嚷著「我知道，這是第三關，這一關要怎麼怎麼打就可以輕易過關...」，真是敗給他了。

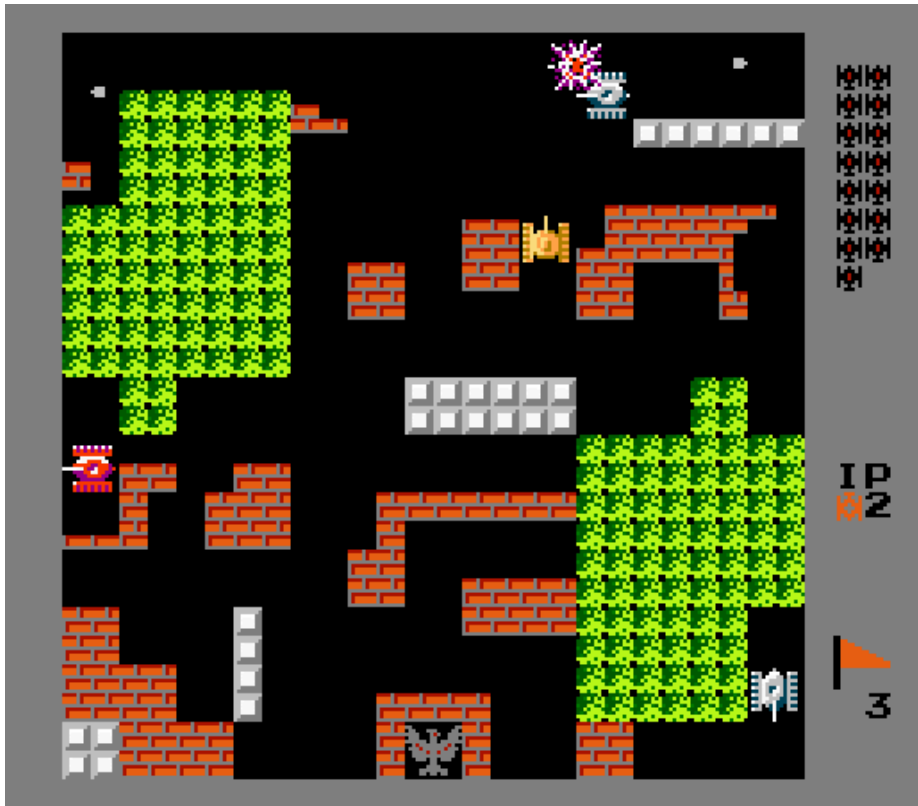


圖 9-1 / 坦克大決戰執行畫面，很懷念嗎？

圖中，黃色坦克為主角，其餘皆是敵方坦克，會依序從畫面的左上角，正上方，右上角三個出生點出現，畫面上最多同時有四輛敵方坦克。每一關卡的目標只有一個，「**全力保護我方軍旗，並殲滅所有敵方坦克**」。每一關卡有一定的敵方坦克數量，似乎皆為二十隻，全部打完就過關。不論我方坦克剩餘隻數，只要軍旗被打到，就立即 GAME OVER（我同學又說話了，因為旗幟為老鷹形狀，所以若軍旗被打壞可稱為「烤小鳥」），好殘酷。

坦克大決戰的圖形很簡單，地形總共才五種：

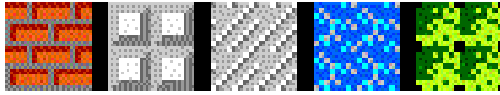


圖 9-2 / 坦克大決戰的地形

- 磚牆
最常見的地形，不能通過，可被子彈擊毀。依擊中位置不同，一發子彈可打壞八分之一或四分之一。
- 鐵牆
坦克無法通過，也不能被子彈擊毀，除非吃到三顆星星以上。
- 鋼板
可在上面走動，但會打滑，每走一步會滑動好幾步。
- 海洋
坦克無法通過，但子彈可通過。
- 樹林
可以通過，但會遮住坦克，成爲絕佳的隱蔽場所。

坦克大決戰的好玩之處就是能夠善用這五種地形，設計成有趣且富挑戰性的關卡，地形的排列方式通常決定此關卡的難易度及平衡度，可說是一門大學問呢。

敵方坦克會不斷在畫面上走來走去，同時胡亂射擊，請小心，務必在槍林彈雨下殲滅它們。敵方坦克有四種：



圖 9-3 / 四種敵方坦克

- 第一種
最常見，一切屬性普通。
- 第二種
六個輪子，跑得很快，一不小心就會被它溜到軍旗旁偷打。
- 第三種
第一種敵方坦克的改良版，只有炮管及尾端有些許不同，子彈飛行速度極快。
- 第四種
體積最大，裝甲最厚，要擊中四次才會翹辮子。

摧毀四種坦克的得分依序是一百、兩百、三百及四百分，但對於以手榴彈（可摧毀所有敵方坦克的寶物）炸掉的坦克，則不計分。

每一關最多會出現四次寶物，每當轟掉閃著紅光的坦克後，寶物就會立即出現，寶物位置及種類依亂數決定。寶物有下列六種：



圖 9-4 / 六種寶物

- 弓箭
將我方軍旗暫時以鐵牆圍住，沒有被攻破之虞。
- 時鐘
看樣子大概也猜得到，可暫停敵方坦克的行動一段時間。
- 手榴彈
將畫面上所有敵方坦克摧毀。
- 鋼盔
我方坦克暫時無敵。
- 坦克
生命數加一，但無論生命數多少，只要軍旗被攻破，遊戲即結束。
- 星星
這是最有用的寶物，若沒有此寶物，要順利玩上十關我想也很難。吃到第一顆星星後，子彈速度會加速，如此才能與敵方的高級坦克匹敵；吃到第二顆星星後，可以連發兩發子彈；吃到第三顆星星後，可以無堅不摧，轟破鐵牆。這時我同學又說了「傳說中，吃到三十顆星星後，坦克就可以自由地行走海洋」。不過傳說歸傳說，沒試過誰也不曉得，依我的功力，往往還沒吃到三顆星星就掛點了，你可以幫我驗證這個「傳說」嗎？：)

寶物出現後，在畫面上閃呀閃的，若不去吃，待下一個閃著紅光的坦克出現後，寶物就會消失，所以吃寶物的動作要快，迅速確實才行。

設計自己的坦克大決戰

詳細介紹任天堂版本的坦克大決戰後，希望能讓你徹底瞭解這個經典級遊戲。接下來，讓我們也來寫一套坦克大決戰，希望能實作出其大部分的功能，甚至加入新的改良及功能。

對於前一章的「足球番」遊戲還滿意嗎？不知道你覺得太簡單，還是太難了？撰寫此章的同時，我已將程式全部撰寫完成了，完成後才發現，糟糕，不曉得是上一章的「足球番」太簡單還是這一章的坦克大決戰太難，「足球番」三支程式的程式碼林林總總加起來約兩千行，而這回坦克大決戰卻是四千行，正好是兩倍，哇 ~~ 好多啊，讀者你要有心理準備哦。

選擇坦克大決戰，也因為它的幾個特性恰巧符合本章的教學目的：

1. 畫面處理 GDI 「勉強夠用」，不需用到 `DirectDraw`，背景不用捲動。
2. 除遊戲主程式外，必須另有地圖／關卡編輯器的搭配才算完整，另外還配有圖庫編輯器，這兩支工具可推廣使用於許多益智、角色扮演甚至動作遊戲上頭。
3. 圖形使用不多，但是遊戲本身耐玩，不是須靠畫面才能吸引人的遊戲類型。
4. 可擴充的地方極多，例如可為它加上網路連線功能，讓我可跟住台北的大哥合作破敵等等。

使用 GDI 來撰寫倉庫番可說是遊刃有餘，但今天對於坦克大決戰這樣的即時動作遊戲，在圖形不多且地圖不大的情況下，我只能說，勉強夠用。在 Pentium II 450、Windows NT 4.0 下十分順暢，在 IBM ThinkPad 570、Pentium II 300、Windows 98 下執行速度也還不錯，但這大概就是極限了。要不是有畫面上最多四輛敵方坦克的限制，光是重繪所有坦克及漫天飛舞的子彈我想就可能讓遊戲一頓一頓的，更別提即時處理使用者的輸入了。

先來訂立我們這套坦克大決戰的功能及特色：

1. 遊戲規則與坦克大決戰大致相同，主角必須保護軍旗，並將所有敵方坦克摧毀即可過

關。

2. 可視範圍即是地圖大小，因此不必支援地圖捲動。
3. 支援由多張地形圖片拼湊為一個圖片群組，編輯地圖時可直接對整個或部分圖片群組進行操縱。
4. 角色大小沒有限制，亦即，哪天心血來潮，弄了一部半個畫面大的巨型坦克想加入遊戲成為敵方坦克也不是問題。
5. 角色的移動以「點」為單位，可以平滑移動，因此碰撞處理十分麻煩。這也是此程式的程式碼為「足球番」程式碼兩倍行數的最大因素。
6. 所有圖片，包括角色圖形皆採外掛方式，可在不修改程式碼的情形下更動圖形。
7. 採用關卡制度，可讓使用者自行編輯關卡及遊戲。
8. 多層貼圖，因此可造出高度及層次感，也可設計出較真實的地形。

與正宗的「坦克大決戰」相較，最大的改進是**支援圖片群組**，**任意尺寸的角色**以及**多層貼圖**。其實若只想單純地實作「任天堂版本坦克大決戰」，這三項功能根本不必要，只要依賴「足球番」那種超陽春型圖庫編輯器及地圖編輯器，即可達成坦克大決戰所需的效果。

但是程式寫著寫著，我還是決定多留點空間給這支程式，使得它的發揮空間極大。而這些空間就留給讀者盡情發揮囉。

系統規劃

全套遊戲除了主程式外，另設計兩支工具程式－圖庫編輯器及地圖編輯器，理由是減少程式設計的複雜度及不必要的 overhead。

程式中，我大量使用類別及物件，亦即，遊戲畫面中所有看得到的地圖、圖格、坦克、子彈、寶物等等，通通都是物件。在物件導向程式設計中，好好地規劃類別以及類別間

式只是遲早的事，端視實作能力而定；但若老在技術及程式碼上斤斤計較，分析及規劃能力窒礙不進，整體的程式設計能力是很難升級的。

分爲兩個子系統來介紹遊戲的類別規劃。

地圖子系統

圖庫處理

TTile 類別

- 管理一張圖片（待會常數定義就會看到，圖片大小定爲 32 x 32）。
- 記錄圖片屬性，如坦克能否通過、子彈能否通過、子彈能否破壞等等。
- 支援圖片群組。每個圖片群組爲最大 5 x 5 張圖片的集合，例如你可以設計大小爲 128 x 96 的城堡圖案，使用 4 x 3 大小的圖片群組，使用 12 個 *TTile* 物件，也就是 12 張圖片。

TTiles 類別

- 擔任圖庫總管，管理（包括產生及摧毀、新增、刪除）旗下所有 *TTile* 物件。
- 負責儲存／載入整個圖庫。
- 提供圖片給地圖類別使用，才能順利地繪製遊戲畫面。

地圖處理

TCell 類別

- 管理地圖上一個圖格（與圖片大小一致，爲 32 x 32）。
- 記錄此圖格所使用的圖片編號，即 *TTile* 物件在圖庫中的編號。

- 管理圖格的破碎情形，將圖格區分為 4 x 4 個小圖格，其中每個小圖格皆呈存在或不存在的狀態，繪製時只繪出存在狀態的小圖格，如此便可達到圖格破碎的效果（用於被子彈擊毀的磚牆）。
- 計算此圖格所佔用的矩型區域（考慮破碎狀態）。

TMap 類別

- 管理一道關卡的所有地圖，每個關卡擁有四張地圖（分別為地形、地形物、物品及高地形物四層）。
- 負責此四層地圖的載入／儲存，還有使用最頻繁的繪製圖層工作。
- 繪製圖層時，會將此圖層所有圖格以對應的圖片繪製出來，其中地形層採直接繪製，而其它圖層使用透明貼圖。

角色子系統

TSprite 類別

角色子系統完全由 *TSprite* 類別來擔綱，所有角色子系統中的其它類別皆是 *TSprite* 的子類別，由此可見它的重要性。*TSprite* 類別的任務為：

- 管理角色圖形。角色面對每個方向時各自有不同的代表圖形，而在同一個方向時也有多張動畫可以輪替使用，以達成走動、游動等效果。
- 記錄角色狀態，如座標、方向、移動速度、是否可見、是否在空中、佔用的矩型區域等等。
- 記錄角色屬性，如圖形是否具方向性、是否與地形物碰撞、是否與坦克碰撞、走動時是否自動切齊地形物等等。
- 更新角色動作。每進行一步動作，就要更新動畫、移動座標、進行碰撞處理。
- 碰撞處理，分別檢查與邊界、地形及其它角色的碰撞情況，再根據屬性及狀態來決

定碰撞後的反應。

- 繪製角色。將正確的角色圖形繪製在畫面上的對應位置。

工作很多吧。它實在太重要了，此類別要是沒有好好設計，保證遊戲繼續往下寫，需要進行碰撞處理，如子彈打到坦克、坦克吃到寶物等部分時，會後悔地叫苦連天。爲什麼我這麼確信呢？因爲...這是...經驗談。:P

坦克類別

所有的坦克皆是 *TTank* 類別的後代類別，*TTank* 類別的任務如下：

- 記錄坦克的狀態，如生命力、目前子彈數目、子彈爆炸威力、是否無敵模式等等。
- 處理與坦克碰撞相關的碰撞處理。
- 發射子彈，將子彈依坦克方向發射出去。

TTank 類別有兩個後代，分別是我方坦克專用的 *TMyTank* 及敵方坦克使用的 *TETank*：

- 我方坦克 *TMyTank*
 - 進行與寶物物件的碰撞處理，若碰到了即吃掉寶物。
- 敵方坦克 *TETank*
 - 負責在出現前發出金光閃閃的特殊效果。
 - 除非被幹掉，否則從不停止走動。
 - 以亂數自由走動、轉向、發射子彈，若連續碰撞超過某個數目，就一定轉向。

遊戲中會出現五種敵方坦克，在此我們爲五種坦克分別設計新的類別，而使用 *Prototype* 樣式¹：先產生 *TETank* 類別物件，再根據五種敵方坦克的特性，分別設定它們的移動速度、子彈速度、爆炸威力、移動方式等等屬性。

¹ 請參考附錄 C 「參考書目」所列的 *Design Patterns* 一書。

咦？任天堂版的坦克大決戰不是只有四種敵方坦克嗎？嘻，因為我偷偷多加了一種可「在天上飛」的超強敵方「坦克」，我同學又在身邊唸了：「破壞遊戲平衡度的傢伙...！」，呵呵，反正程式是我寫的，不用他。這五個類別只要負責設定好它們各自對應的屬性即可。

金光閃閃的 TStar

敵方坦克出生前，同一地點會出現一顆星星閃呀閃的，接著敵方坦克才出現。這顆星星是由 *TStar* 類別負責繪製，*TStar* 類別任務最簡單了，只有一項：

- 原地不動，將星光閃動的幾張動畫秀完就行了。

TStar 類別只是被動地由製造效果的 *TETank* 使用，因此主控權完全操在 *TETank* 手上，待會我們就可看到 *TETank* 如何地控制 *TStar* 來達成金光閃閃的特殊效果。

子彈類別 TBullet

TTank 類別不是會發射子彈嗎？此子彈為 *TBullet* 類別，子彈雖小，但 *TBullet* 類別可不簡單，它必須負責：

- 記錄發射子彈本身的坦克。
- 依坦克的方向及設定好的速度移動。
- 最重要的是碰撞處理：
 - 有些地形，雖然一般角色會撞上，但子彈不會撞到（如海洋），須特別處理。
 - 撞上邊界時，引發**小爆炸**。
 - 撞上地形物時，引發**小爆炸**，並損毀地形物（若該地形物可被損毀）。
 - 撞上坦克時，判斷是不是我方打到敵方或是敵方打到我方（若敵方打到敵方，則當

做沒事，子彈將穿越坦克而過），然後引發**大爆炸**。接著減少坦克生命力，若坦克掛了，則將坦克摧毀。

- 撞上別的子彈時，判斷是不是我方打到敵方或是敵方打到我方，若是，則將兩發子彈**摧毀**。
- 管理及繪製爆炸效果物件（*TExplosion*）。

爆炸效果類別

Oh，沒想到小小一顆子彈，任務這麼繁重。子彈爆炸時，它會產生 *TExplosion* 物件來製造爆炸效果，*TBullet* 及 *TExplosion* 物件的關係就如同 *TETank* 與 *TStar* 物件的關係一般，*TExplosion* 只負責繪製效果，它的生滅以及所有活動皆被產生它的 *TBullet* 物件控制。不過，*TExplosion* 類別還多了一項任務：

- 原地不動，將爆炸效果的幾張動畫秀完就行了。
- 若爆炸導致遊戲結束，則在爆炸結束後通報遊戲主迴圈：此局必須結束了。

至於 *TExplosion* 的兩個子類別：*TSmallExplosion* 及 *TBigExplosion*，分別代表小爆炸及大爆炸，沒有動作上的不同，只有圖形的不同而已。

TGem 寶物類別

最後，剩下 *TGem* 寶物類別，它的任務為：

- 隨意找個地方擺，接著原地不動，等著我方坦克來吃。

就這樣，哇啊，簡單吧。角色子系統的類別也介紹完畢。

兩個子系統中所有類別的任務皆分派後，請再回頭看看圖 9-5，對於整個遊戲的架構，你是否已瞭然於胸了呢？那麼，理論上，目標已在眼前，接下來的路途，唯「實作」二字

而已。

地圖子系統

不用我說，你一定也知道，地圖子系統一定比角色子系統好寫多了。因為柿子總先挑軟的吃，所以我才選擇先撰寫地圖子系統，對不對？

呵，才不是呢，我們不是要分別撰寫圖庫編輯器、地圖編輯器及遊戲主程式三支程式嗎？地圖子系統於三個程式中都會用到，而角色子系統只在遊戲主程式用得上而已。再加上地圖子系統裡所有程式碼並不依賴任何角色，但角色必須依賴地圖來進行碰撞處理，因此無論如何我們也得先從地圖子系統著手。

首先定義地圖子系統所需的常數（定義於 `util.h`）：

```
#define TILE_NUM_X      13           // 畫面橫軸格數
#define TILE_NUM_Y      13           // 畫面縱軸格數

#define TILE_WIDTH      32           // 圖片寬度點數
#define TILE_HEIGHT     32           // 圖片高度點數

#define SM_TILE_NUM_X   4            // 小碎片寬度點數
#define SM_TILE_NUM_Y   4            // 小碎片高度點數

#define SM_TILE_WIDTH   (TILE_WIDTH / SM_TILE_NUM_X) // 小碎片寬度點數
#define SM_TILE_HEIGHT  (TILE_HEIGHT / SM_TILE_NUM_Y) // 小碎片高度點數

#define WORLD_WIDTH     (TILE_WIDTH * TILE_NUM_X)     // 畫面寬度
#define WORLD_HEIGHT    (TILE_HEIGHT * TILE_NUM_Y)    // 畫面高度

const char* SIG_MYFILE = "Xshadow_Stock"; // 圖庫及地圖檔案的檔頭標籤

const char* FN_TILE_ARCHIVE = "TILES.TIA"; // 圖庫檔案

const char* FN_MAP_PREFIX = "MAP"; // 關卡圖檔檔名 (MAP???.DAT)
const char* FN_MAP_EXT = ".DAT"; // 關卡圖檔副檔名

const int LAYER_TERR = 0; // 地形層
const int LAYER_TERRITEM = 1; // 地形物層
```

```
const int LAYER_ITEM          = 2;           // 物品層
const int LAYER_HITERRITEM    = 3;           // 高地形物層

#define LAYER_MAX              3             // 最多到高地形物層
```

圖片高及寬度為 32 x 32，是十分常見且有效率的大小設定，因為我們的 CPU 通用暫存器寬度也是 32 bit，在進行記憶體區塊搬移時，不會有不符合 **DWORD alignment** 的情況發生。

畫面橫軸及縱軸格數，13 x 13，是根據任天堂版坦克大決戰而訂，這樣一來，連地圖都可以照抄，享受一下不動大腦的悠閒舒適。:p

SIG_MYFILE 為檔頭標籤，在讀取圖庫及地圖檔案時，先確認檔案開頭有沒有這個字串，以確定讀取的是我們自己的檔案，不會有誤讀的情況發生。

與原版坦克大決戰相比，多個圖層是一大改良，我定義了四個圖層：

- 地形層
鋪在最底端，如沙地、水泥地、海洋等等，採不透明貼圖。因此不論如何，我們的畫面上一定有「地板」，不會讓玩者看到黑黑的圖格，與任天堂版坦克大決戰不同，請參看圖 9-1。
- 地形物層
擺設磚牆、鐵牆的圖層，採透明貼圖，因此若磚牆有半塊打破了，可以看到下面的地形層。為求效率起見，這是唯一進行碰撞判斷的圖層。因此，假設有一塊屬性設定為不能通過的海洋圖片，若將它擺在地形層，坦克依然可以通過，因為地形層並不進行碰撞處理；但若擺在地形物層，因為碰撞處理的緣故，坦克就不能通過，這樣的設定使得關卡設計的自由度大增。
- 物品層
意義與地形物層一樣，也採透明貼圖，只差在它並不進行碰撞處理。
- 高地形物層
意義與物品層一樣，唯一的差別是繪製的順序。此圖層繪製順序在角色之後，所以看起來會在角色上方。例如擺上一些花棚，坦克可從其下通過。

四個圖層以及角色的貼圖順序為：地形層、地形物層、物品層、地上的角色、高地形物層、天上的角色。

四個圖層各有其功能及特色，妥善地安排圖層，既可使程式好寫不少，又可造出更好看的佈景。

不過圖層數目一多，缺點也跟著衍生，每多一層圖層，貼圖部分就多了一份工作，它必須一一檢查 *TILE_NUM_X* 乘上 *TILE_NUM_Y* 個圖格的圖片編號，若該圖格有圖片就把圖片貼上，因此若分為太多層，每層又擺一大堆東西時，很容易就大幅拉下遊戲的執行速度，因此必須謹慎考量。

圖庫處理

TTile 圖片類別

圖庫處理部分只有兩個類別：*TTile* 及 *TTiles*。而 *TTile* 圖片物件由 *TTiles* 圖庫物件管理，因此第一個就從管理單一圖片的 *TTile* 類別下手（定義於 *TileUnit.h*）：

```
#0001 // 圖片屬性：可以通過，圖片可被打破，子彈可以通過，此圖片是軍旗
#0002 enum TTileAttrElement {taCanPass, taCanBreak, taBulletCanPass,
#0003     taFlag};
#0004 typedef Set<TTileAttrElement, taCanPass, taFlag> TTileAttr;
#0005
#0006 class TTile { // 單一圖片
#0007 private:
#0008     Graphics::TBitmap* FBits; // 存放圖片的 bitmap
#0009     TTileAttr FAttr; // 屬性
#0010     bool FDisposed; // 是否已棄置不用
#0011
#0012     bool FFirstTile; // 是不是圖片群組裡的第一張圖片
#0013     Byte FXNum, FYNum; // 圖片群組的橫向及縱向圖片數目
#0014 protected:
#0015 public:
#0016     TTile();
#0017     ~TTile();
```

```
#0018
#0019 void init();
#0020
#0021 // assignment constructor
#0022 TTile& operator=(const TTile& t);
#0023 // copy constructor
#0024 TTile(const TTile& t);
#0025
#0026 // 載入及儲存圖片
#0027 void LoadFromStream(TStream* Stream);
#0028 void SaveToStream(TStream* Stream);
#0029
#0030 // 提供給外界存取的屬性
#0031 __property Graphics::TBitmap* Bitmap = {read = FBits};
#0032 __property TTileAttr Attr = {read = FAttr, write = FAttr};
#0033 __property bool Disposed = {read = FDisposed, write = FDisposed};
#0034
#0035 __property bool FirstTile =
#0036     {read = FFirstTile, write = FFirstTile};
#0037 __property Byte XNum = {read = FXNum, write = FXNum};
#0038 __property Byte YNum = {read = FYNum, write = FYNum};
#0039 };
```

與前一章足球番程式管理方式不同的是，不再很愚蠢地將圖庫中所有圖片置於一個 BMP 圖檔中，現在每張圖片分別管理，每張都是獨立的 `bitmap`，存放於 `TBitmap` 物件 `FBits`。

無法任意刪除的圖片

`FDisposed` 布林變數指的是此圖片是否已廢棄不用。這樣做的理由是，為求方便，我直接使用陣列索引來做為圖片編號，這很好。不過當刪除一張圖片時，原來陣列索引大於它的圖片的陣列索引就會減一，那麼原本編輯好的地圖（每個圖格皆儲存對應的圖片編號）就會混亂，應該秀出 4 號的圖格結果秀出 5 號，應該秀出 100 號的圖格結果秀出 101 號... 結果每當圖片刪除時，所有的地圖檔都必須隨之修改，這是無法接受的情形。因此，若 `FDisposed` 為 `true`，表示此圖片事實上已刪除，只是我們不將它從陣列中拿走，免得影響其它圖片的編號。

那你可能會抗議，若刪除的圖片都不拿掉，豈不白白浪費記憶體及磁碟空間，配置不必

要的 *TBitmap* 來存放不必要的圖形？唔，只要在載入及儲存圖片的函式中動點手腳，就可讓已棄置的圖片不再浪費資源來存放 *bitmap* 及其它資料。至於 *TTile* 物件則沒有辦法不建構，合理的解決方案是在遊戲設計時期先不去管它，等到要將遊戲移交給別人使用前，再撰寫一支工具程式來移除已棄置的圖片，並同時修正所有的地圖檔。

避免浪費記憶體及磁碟空間的手腳是這樣做的：

```
#0001 void TTile::LoadFromStream(TStream* Stream)
#0002 {
#0003     TReader* reader = new TReader(Stream, 2048);
#0004     try {
#0005         FDisposed = reader->ReadBoolean();
#0006         reader->FlushBuffer();
#0007         // 是否已棄置不用 ?? 是的話就不再讀取其它屬性
#0008         if (FDisposed) return;
#0009
#0010         FBits->LoadFromStream(Stream); // 圖形
#0011         reader->FlushBuffer();
#0012
#0013         // 屬性
#0014         for (TTileAttrElement x = taCanPass; x <= taFlag;
#0015             x = (TTileAttrElement)(x + 1)) {
#0016             bool b = reader->ReadBoolean();
#0017             if (b) FAttr = FAttr << x;
#0018         }
#0019
#0020         FFirstTile = reader->ReadBoolean();
#0021         if (FFirstTile) { // 若是群組頭頭，則讀取群組長寬
#0022             FXNum = reader->ReadInteger();
#0023             FYNum = reader->ReadInteger();
#0024         }
#0025     } __finally {
#0026         delete reader;
#0027     }
#0028 }
#0029
#0030 void TTile::SaveToStream(TStream* Stream)
#0031 {
#0032     TWriter* writer = new TWriter(Stream, 2048);
#0033     try {
#0034         writer->WriteBoolean(FDisposed);
#0035         writer->FlushBuffer();
#0036         // 是否已棄置不用 ?? 是的話就不再寫入其它屬性
#0037         if (FDisposed) return;
```

```
#0038
#0039  FBits->SaveToStream(Stream); // 圖形
#0040
#0041  writer->FlushBuffer();
#0042
#0043  // 屬性
#0044  for (TTileAttrElement x = taCanPass; x <= taFlag;
#0045       x = (TTileAttrElement)(x + 1))
#0046     writer->WriteBoolean(FAttr.Contains(x));
#0047
#0048  writer->WriteBoolean(FFirstTile);
#0049  if (FFirstTile) { // 若是群組頭頭, 則寫入群組長寬
#0050     writer->WriteInteger(FXNum);
#0051     writer->WriteInteger(FYNum);
#0052  }
#0053  } __finally {
#0054     delete writer;
#0055  }
#0056 }
```

0010 及 0039 列分別讀出及寫入圖片影像，只要一個呼叫就了，這正是物件永續（object persistence）機制最快樂的應用。

圖片群組支援

下圖分別從兩個地圖編輯器執行畫面取得，左圖的圖片一團混亂，硬生生地將河流、小橋、流水，拆成好多圖片，散落在圖片堆內，讓人邊找邊拼，設計地圖時同時玩拼圖遊戲。右圖是支援圖片群組的圖片選取視窗，可以一次將整個單位的圖片框選，扔到地圖上；也可以只選取任何一部分，完全隨心所欲。



圖 9-6 / 有無支援圖片群組的比較

光看這兩個地圖編輯器的畫面，即刻的反應是：「天啊，不支援圖片群組的地圖編輯器對於關卡設計者真是太殘忍了」。於是，稟著仁民愛物的精神，*TTile* 類別也不落人後，提供圖片群組的支援。

若 *TTile* 物件本身為「群組頭頭」，也就是圖片群組的最左上角那一張，則只有它知道圖片群組的寬與高資訊，所以圖片群組的大小由它來維護、管理，「群組頭頭」以外的圖片並不曉得自己究竟屬於哪個群組，又此群組的大小為何。圖片群組對遊戲主程式沒有任何影響，它只是讓地圖編輯器使用起來方便且人性化多了。如果你曾讓不支援群組的地圖編輯器虐待過，那麼你一定會喜歡此設計，雖然圖片群組的支援用不著多少程式碼。

另外還有一點的好處是，這兒設計的圖片群組的唯一功能只有輔助地圖設計，而不會帶來其它困擾。後頭實作地圖編輯器時你將會看到，可以直接將整個群組以一張大圖的方式來張貼，也可以將它拆開張貼，如同古早的地圖編輯器那樣。

TTiles 圖庫類別

下面是 *TTiles* 類別的定義，它擔任圖庫總管的角色，管理（包括產生及摧毀、新增、刪除）旗下所有 *TTile* 物件，並負責儲存／載入整個圖庫。

```

#0001 class Ttiles { // 圖庫
#0002 // Singleton Pattern
#0003 private:
#0004     static Ttiles* FInstance;
#0005 public:
#0006     static Ttiles& Instance();
#0007 private:
#0008     typedef std::vector<Ttile*> TtileArray;
#0009     TtileArray* FTiles; // 所擁有的圖片陣列
#0010
#0011     int GetTileNum(); // 圖片數目
#0012
#0013     Ttile& GetTile(int No); // 利用索引取得圖片
#0014 protected:
#0015     Ttiles();
#0016     ~Ttiles();
#0017 public:
#0018     int AddTile(Ttile* NewTile); // 加入新的圖片
#0019     void FreeTiles(); // 釋放所有圖片
#0020
#0021     // 載入及儲存圖庫
#0022     void LoadFromFile(AnsiString FileName);
#0023     void SaveToFile(AnsiString FileName);
#0024
#0025     __property int TileNum = {read = GetTileNum};
#0026     __property Ttile Tile[int No] = {read = GetTile};
#0027 };

```

0009 列宣告存放所有 *Ttile* 物件的 *FTiles* 動態陣列，在此我以 C++ Standard Library 提供的 *vector* 來存放不定數目的 *Ttile* 物件。

Ttiles 類別沒幹啥大事，反正需要圖片時找它要就對了。因為整個遊戲中，只用到一個圖庫，多了也沒用，所以我實作了 Singleton 樣式，強制整個系統最多只能有一個 *Ttiles* 物件，且可於任何地方存取。

地圖處理

圖庫及圖片準備好後，接著才能撰寫地圖部分，因為地圖必須依賴圖庫才能操作及顯示。地圖處理的最小單位為圖格，用一個 *TCell* 物件來表示。除了存放此圖格所使用的圖片編號，另外還負責管理圖格的破碎情形，*TCell* 類別宣告如下（定義於 MapUnit.h）：

```
#0001 class TCell {
#0002 private:
#0003     int FTileNo; // 圖格所放置的圖片編號
#0004     int FLayer, FX, FY; // 圖層, 座標
#0005
#0006     // 圖格破碎表格
#0007     // SM_TILE_NUM_X x SM_TILE_NUM_Y 個小圖格, "破" 或 "沒破"
#0008     typedef bool TBreakMap[SM_TILE_NUM_X][SM_TILE_NUM_Y];
#0009     typedef TBreakMap* PBreakMap;
#0010
#0011     PBreakMap FBreakPtr; // 圖格破碎表格
#0012     TRect FRect; // 圖格所佔區域, 會隨圖格破碎而變更
#0013
#0014     // 用於破碎圖格的貼圖動作
#0015     Graphics::TBitmap* FCellBitmap, *FSMTileBitmap;
#0016
#0017     TTileAttr GetTileAttr();
#0018     void SetTileAttr(const TTileAttr Value);
#0019
#0020     bool GetCanPass(); // 這個圖格能否通過 ?
#0021     void SetTileNo(int Value);
#0022
#0023     // 重新計算圖格所佔區域
#0024     void CalcRect();
#0025     // 建立圖格圖形 (可能是破碎的)
#0026     void BuildCellBitmap();
#0027
#0028     void BreakMapChanged();
#0029
#0030     bool IsBroken(int x, int y);
#0031
#0032     Graphics::TBitmap* GetTileBitmap(); // 取得圖片 bitmap
#0033
#0034     __property Graphics::TBitmap* TileBitmap = {read = GetTileBitmap};
#0035 protected:
#0036 public:
#0037     TCell();
#0038     ~TCell();
#0039
#0040     void init(int Layer, int x, int y);
#0041
#0042     // assignment constructor
#0043     TCell& operator=(const TCell& c);
#0044
#0045     // 載入及儲存
#0046     void LoadFromStream(TStream* Stream);
```

```
#0047 void SaveToStream(TStream* Stream);
#0048
#0049 // 圖格破碎處理函式
#0050 void AllocBreakMap(); // 建立圖格破碎表格
#0051 void DisposeBreakMap(); // 釋放圖格破碎表格
#0052 void BreakBy(TRect ARect); // 依矩形區域設定圖格破碎表格
#0053
#0054 void Draw(TCanvas* Canvas, bool bTransparent);
#0055
#0056 __property int TileNo = {read = FTileNo, write = SetTileNo};
#0057 __property TTileAttr TileAttr =
#0058     {read = GetTileAttr, write = SetTileAttr};
#0059
#0060 __property TRect Rect = {read = FRect};
#0061
#0062 __property bool CanPass = {read = GetCanPass};
#0063 };
```

0003 列為最重要的，記錄圖格所使用的圖片編號。除此之外，0004 列還記錄此圖格位於哪個圖層的哪個位置上，位置資訊於計算圖格佔用的矩形區域時派上用場，而圖層編號也是繪製圖格時所需的重要資訊。

0008 列為其圖格破碎表格的宣告，此表格是四乘四的布林陣列，分別記錄每個小圖格「破」或「沒破」，以達成磚牆被擊毀的碎裂效果。

破碎圖格處理

圖格最吃重的任務大概就屬破碎圖格的處理了。0011 列將 *FBreakPtr* 宣告為指向圖格破碎表格的指標，而不直接宣告指向圖格破碎表格是有其用意的。因為可能呈破碎情況的圖格算是少數，只有位於地形物層（此層才有碰撞處理）且屬性帶有 *taCanBreak* 的圖片才可能破碎，更何況有時磚牆還沒破幾個，就過關了（若是我的話，大概是磚牆還沒破幾個，就被敵人幹掉了），所以以指標來指向破碎表格。正常圖格的 *FBreakPtr* 指標為 *NULL*，只有在圖格被打破時，才動態地建立破碎表格來使用，如此便可以節省不少記憶體空間的使用。

設定圖格破碎表格內容，也就是圖格破碎情形的函式有二，一是呼叫 *DisposeBreakMap* 函式，釋放目前的圖格破碎表格，代表此圖格完全沒有破碎情形；二是呼叫 *BreakBy* 函式，指定一個 *TRect* 矩形區域，將圖格與此矩形區域交集的所有小圖格設定為破碎（不存在）：

```
#0001 // 依矩形區域設定圖格破碎表格
#0002 void TCell::BreakBy(TRect ARect)
#0003 {
#0004 // 若此時還沒有破碎表格，就建一個
#0005 if (!FBreakPtr) AllocBreakMap();
#0006
#0007 TRect R, R1;
#0008
#0009 // 與 ARect 取交集，有交集的小格則設為破掉
#0010 for (int y = 0; y < SM_TILE_NUM_Y; y++)
#0011     for (int x = 0; x < SM_TILE_NUM_X; x++) {
#0012         // 計算小格的矩形區域
#0013         R1 = SM_TILE_RECT;
#0014         OffsetRect(&R1, SM_TILEWIDTH[x], SM_TILEHEIGHT[y]);
#0015         OffsetRect(&R1, TILEWIDTH[FX], TILEHEIGHT[FY]);
#0016
#0017         // 若有交集，則讓它破掉
#0018         if (IntersectRect(&R, &ARect, &R1))
#0019             (*FBreakPtr)[x][y] = true;
#0020     }
#0021
#0022     BreakMapChanged();
#0023 }
```

因為 $n * TILE_WIDTH$ 及 $n * TILE_HEIGHT$ 兩個乘法運算使用頻率極高，因此 Util 單元特別提供 *TILEWIDTH* 及 *TILEHEIGHT* 兩個一維陣列作為查表用途。

設定破碎表格的函式是，針對四乘四個小圖格一一測試，呼叫 *IntersectRect* API 函式，傳入兩個矩形區域，它會傳回一布林值代表這兩矩形是否重疊，一旦有交集，就將此小圖格設定為破碎，繪製圖格時就不會畫出來了。

BreakMapChanged 函式用來重新計算圖格所佔用的矩形區域。若 *FBreakPtr* 為 *NULL*，表示此圖格沒有任何破碎，就按照正常程序計算圖格佔有的矩形區域；但若 *FBreakPtr* 不為 *NULL*，表示此圖格已有破碎發生，此時必須再分別從四個方向去檢查破碎表格，最

上方、最下方、最左方及最右方的未破碎小圖格，才能得到圖格的真正矩形區域。此矩形區域供碰撞處理函式使用，但你也許已經發現，以矩形來測試碰撞，寫出來的遊戲一定會有不合理的狀況發生，例如，若某圖格除了最左邊一排及最下邊一排的小圖格外，通通都被打破了，但它佔用的矩形區域還是跟原本一樣大，所以從右上方來的坦克還是無法進入它的空缺處。這個缺點只有更詳細的碰撞處理方式才能解決，目前我們暫且先將此問題擱下。

儲存圖格時，也會將目前的破碎表格一併寫入資料流中，因此，只要地圖編輯器支援，設計關卡時就可設定圖格的破碎情況，可以藉此設計一道殘破不堪的廢墟關卡，或是拼湊出字形、圖案等等，都是不錯的應用。

圖格的繪製

圖格的繪製工作由 *Draw* 函式負責。繪製過程中，有兩個要素需要考量：是否需要採用透明貼圖，以及破碎圖格的外觀。

是否需要採用透明貼圖呢？這個問題十分好作答，只要是地形層，就採不透明貼圖；只要不是地形層，就採透明貼圖。

至於破碎圖格的外觀，的確是比較棘手的問題，因為圖格一旦呈破碎狀態，我們就無法直接使用圖庫所提供的圖片來繪製—因為圖格的外觀改變了嘛。那麼，可否在建立圖庫時，就為每一種破碎情況準備一張圖片呢？這方法不可行。因為每個圖格有 16 個小圖格，在每個小圖格都可能破／沒破的情形下，可能有 $2^{16} = 65536$ 種破碎情形，數目太大了，建立這種多個圖片只會造成資源浪費。於是，我們只剩下一種方法：在圖格破碎情形改變時，立即為此圖格準備一張呈現其破碎情形的圖片。首先，在 *TCell* 類別裡加入 *TBitmap* 物件 *FCellBitmap*，包含呈現圖格外觀的影像。然後撰寫 *BuildCellBitmap* 函式：

```
#0001 // 繪製破碎的小圖格
#0002 void TCell::BuildCellBitmap()
#0003 {
#0004     if (!FBreakPtr) { // 未破碎，直接取用圖庫圖片
```

```

#0005   FCellBitmap->Assign(GetTileBitmap());
#0006       return;
#0007   }
#0008
#0009   // 先把 FTileBitmap 設成全部透明
#0010   FCellBitmap->Canvas->Brush->Color = TRANSPARENT_COLOR;
#0011   FCellBitmap->Canvas->Brush->Style = bsSolid;
#0012   FCellBitmap->Canvas->FillRect(TILE_RECT);
#0013
#0014   // 若為破碎圖格，則一一將仍存在的小圖格貼上
#0015   for (int n = 0; n <= 3; n++)
#0016       for (int m = 0; m <= 3; m++) {
#0017
#0018       // 若此小圖格破掉了就不用畫
#0019       if (IsBroken(m, n)) continue;
#0020
#0021       // 先複製到另一個 bitmap
#0022       FSMTileBitmap->Canvas->CopyRect(
#0023           SM_TILE_RECT,
#0024           GetTileBitmap()->Canvas,
#0025           Classes::Rect(SM_TILEWIDTH[m], SM_TILEHEIGHT[n],
#0026               SM_TILEWIDTH[m + 1], SM_TILEHEIGHT[n + 1]));
#0027
#0028       // 再貼到畫布上，以達成透明貼圖效果
#0029       FCellBitmap->Canvas->Draw(SM_TILEWIDTH[m], SM_TILEHEIGHT[n],
#0030           FSMTileBitmap);
#0031   }
#0032 }

```

若圖格沒有破碎，*FCellBitmap* 的影像只要直接由圖庫取得，呼叫 *TBitmap::Assign* 函式複製過來即可。若圖格是破碎的，沒有更聰明的方法，必須依序一個個檢查小圖格的破碎與否，將還存在的小圖格畫到 *FCellBitmap* 上頭。這也是為什麼，每一次破碎表格更動了，就必須呼叫 *BreakMapChanged* 函式的原因：*FCellBitmap* 必須重新繪製。

```

#0001 void TCell::BreakMapChanged()
#0002 {
#0003     CalcRect(); // 重新計算所佔用區域
#0004     BuildCellBitmap(); // 建立圖格圖形 (可能是破碎的)
#0005 }

```

麻煩的 *FCellBitmap* 準備好之後，任何時候需要繪製圖格時，只消呼叫 *TCell::Draw* 函式，將 *FCellBitmap* 貼到畫布上即可。必須注意的是，在非地形層的其他圖層中，編號 0 號的圖片表示此圖格是空的，沒有放置圖片，所以不畫。

```
#0001 // 將指定的地圖層畫在 Canvas 上
#0002 void TCell::Draw(TCanvas* Canvas)
#0003 {
#0004     if (FLayer != LAYER_TERR) { // 非地形層
#0005         if (FTileNo == 0) return; // 沒有設定物品
#0006
#0007         Canvas->Draw(TILEWIDTH[FX], TILEHEIGHT[FY], FCellBitmap);
#0008     } else {
#0009         // 地形層不需要透明貼圖
#0010         BitBlt(Canvas->Handle, TILEWIDTH[FX], TILEHEIGHT[FY],
#0011             TILE_WIDTH, TILE_HEIGHT, FCellBitmap->Canvas->Handle,
#0012             0, 0, SRCCOPY);
#0013     }
#0014 }
```

地圖總管 TMap

接下來是地圖總管—*TMap* 類別，類別宣告如下（定義於 MapUnit.h）：

```
#0001 class TMap {
#0002 private:
#0003     static TMap* FInstance;
#0004 public:
#0005     static TMap& Instance();
#0006 private:
#0007     typedef TCell TMapArray[TILE_NUM_X][TILE_NUM_Y];
#0008
#0009     TMapArray FMaps[LAYER_MAX + 1]; // (0 ~ LAYER_MAX) 層地圖
#0010
#0011     int FLevelNo; // 目前載入的關卡編號
#0012
#0013     int FRole_X, FRole_Y; // 角色的起始位置
#0014
#0015     void SetLevelNo(int Value);
#0016
#0017     void SetRole_X(int Value);
#0018     void SetRole_Y(int Value);
#0019 protected:
#0020     TMap();
#0021     virtual ~TMap();
#0022
#0023     AnsiString GetFileName(); // 根據關卡編號，傳回對應的檔名
#0024 public:
#0025     void LoadFromFile();
```



```

#0026 void SaveToFile();
#0027
#0028 // 將某地圖層畫在 Canvas 上
#0029 void Draw(TCanvas* Canvas, int Layer);
#0030
#0031 TCell& GetCell(int Layer, int x, int y);
#0032
#0033 // 重設整張地圖, 或只重設某一層
#0034 void ResetAllLayers();
#0035 void ResetLayer(int Layer);
#0036
#0037 __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0038
#0039 // 取得初始的角色位置
#0040 __property int Role_X = {read = FRole_X, write = SetRole_X};
#0041 __property int Role_Y = {read = FRole_Y, write = SetRole_Y};
#0042 };

```

0007 列宣告存放每層地圖的 *TCell* 二維陣列型態 *TMapArray*，它包含 *TILE_NUM_X* * *TILE_NUM_Y* 個圖格。而 0009 列宣告 *FMaps* 一維陣列，它包含 *LAYER_MAX* + 1 個 *TMapArray* 陣列，也就是每一關卡所需的所有地圖層。除了地圖層、關卡編號，0013 列還記錄著此道關卡中主角的初始位置。這裡應該還要加入其它關卡資訊，例如每道關卡的敵方坦克種類及出現順序等等。不過目前沒有這樣做，敵方坦克的出現時機及種類由亂數決定。

TMap 類別最常用的函式為 *GetCell* 屬性，它需要三個參數，分別傳入圖層及座標，來取得指定圖層指定座標上的 *TCell* 圖格物件參考。

除了寫入地圖檔及讀出地圖檔兩個函式外，最重要的是繪製地圖層的 *Draw* 函式。此函式會將 *Layer* 參數所指定的圖層畫在 *Canvas* 上頭：

```

#0001 // 將指定的地圖層畫在 Canvas 上
#0002 void TMap::Draw(TCanvas* Canvas, int Layer)
#0003 {
#0004     for (int y = 0; y < TILE_NUM_Y; y++) // 對於每一圖格
#0005         for (int x = 0; x < TILE_NUM_X; x++)
#0006             FMaps[Layer][x][y].Draw(Canvas);
#0007 }

```

因為所有的透明貼圖、破碎圖格的判斷、處理已包含於 *TCell* 類別，所以 *TMap::Draw* 函

式只需進入兩層迴圈，針對畫面上所有的圖格，呼叫 `TCell::Draw` 函式即可，把繪製整張地圖的複製處理分散各處，自個擊破了。

這就是地圖的繪製函式，在遊戲中，每次更新畫面時，會呼叫此函式四次，分別傳入地形、地形物、物品及高地形物等四個圖層編號。而在遊戲進行中，至少每秒鐘會有十二次以上的重繪動作，才不致讓使用者覺得畫面延滯，正因為它是影響遊戲進行效率的重大關鍵，所以此段程式碼必須越精簡越好。

圖庫編輯器

目前為止，我們已將地圖子系統中的四個類別撰寫完成，算是好的開始。不過，由於角色子系統的類別既多且複雜（複雜度主要來自碰撞處理），繼續撰寫角色子系統之前，讓我們先將比較簡單的圖庫編輯器及地圖編輯器完成，程式寫得筋疲力盡後，這兩支程式應該可帶來較為「具體」的成就感。:p

圖庫編輯器的功能純粹為管理圖片及圖片群組，只要提供新增／修改／刪除圖片及圖片群組的功能即足夠，絕對是三支程式中最簡單的，好，那我們就先從它下手。

在程式的撰寫步驟上，我還是延續先介面後程式的習慣，先在 `C++Builder` 整合環境中將使用者介面全部完成，再開始撰寫第一行程式碼。事實上，我本身平日開發大小程式時，也盡量依照這個準則。將使用者介面清楚制定下來後，才可嚴謹地定義出各使用者介面元件之間的互動關係及訊息傳遞流程，最後才在裡頭填寫程式碼，只要介面規劃沒有問題，骨架都搭好了磚頭要擺錯地方也很難。下圖是圖庫編輯器的設計時期畫面：

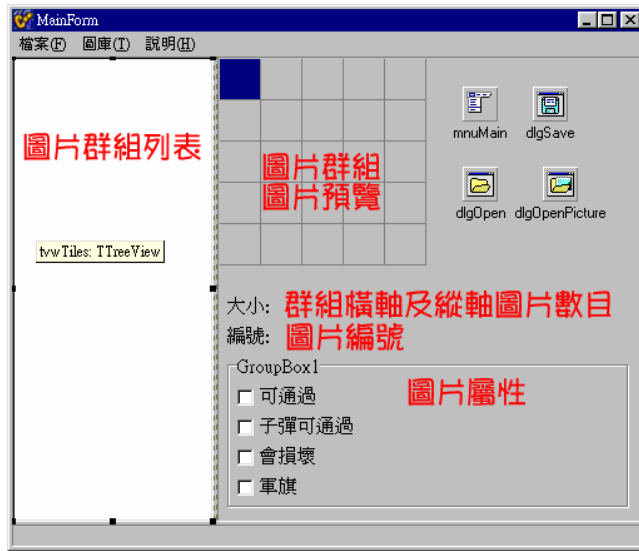


圖 9-7 / 圖庫編輯器主視窗的設計時期畫面

與陽春型圖庫編輯器最大的不同處是，以往以圖片為操作單位，而現在以圖片群組為操作單位，但仍可以個別設定每張圖片的屬性。

視窗左邊是一個樹狀檢視元件，使用這個元件原本的用意是，可以做到階層式的圖片群組分類，例如可將旱地、沙地、草地等歸一類，海洋、河流、湖泊等圖片群組歸一類。但你將可在執行畫面圖 9-8 看到，所有圖片群組描述筆直地排成一行，並沒有所謂階層觀念，這是怎麼回事咧？沒事，只是我懶得撰寫新增、刪除群組類別及樹狀檢視元件的節點拖曳搬移等功能，這種「一塊蛋糕」²等級的功能，相信你也可以在吃掉一塊蛋糕內的時間內寫出；再說，反正目前圖片少得可憐，也沒有分類的必要，就偷懶沒做這部分了。

² Err, I mean the phrase “a piece of cake” !!



圖 9-8 / 圖庫編輯器執行畫面

圖 9-8 中可以清楚地看到，「草怪」這個圖片群組包含 3 x 3 共九張圖片，但在右上角的 *TDrawGrid* 元件中，一次只能選擇一張圖片，再由右下角的四個 *TCheckBox* 元件來設定圖片屬性。並不是每個圖片群組都必須包含多張圖片，事實上，「草怪」是我為了示範圖片群組功能才特別找來加入的，其它的群組，如草地、磚牆、海洋等，都只包含一張圖片而已。

新增及移除圖片群組

【圖庫】功能表下有【新增】、【移除】兩個功能，分別新增及移除一個圖片群組。新增圖片群組時，由使用者指定一個包含整個圖片群組圖形的 **BMP** 檔案，由程式計算此圖片群組的橫軸及縱軸圖片數目，接著一一將此群組的圖片切割取出，加入圖庫。

移除圖片群組時，先由使用者在樹狀檢視元件中選定一個群組描述，程式會一一將此群組所有圖片的 *FDisposed* 布林變數設為 *true*，表示這些圖片已成孤兒，沒人要了，再將該群組描述移除，連身分都毀掉，徹底地讓世人遺忘它們。

新增及移除圖片群組的動作分別由 `mnuAddClick` 及 `mnuRemoveClick` 兩個事件處理函式來負責，程式碼如下：

```
#0001 void __fastcall TMainForm::mnuAddClick(TObject *Sender)
#0002 {
#0003     if (dlgOpenPicture->Execute()) {
#0004
#0005         // 產生及載入欲加入圖庫的 bitmap
#0006         Graphics::TBitmap* Bits = new Graphics::TBitmap;
#0007
#0008         try {
#0009             Bits->LoadFromFile(dlgOpenPicture->FileName);
#0010
#0011             // 若 bitmap 比單張圖片的尺寸還小，無法處理
#0012             if (Bits->Width < TILE_WIDTH || Bits->Height < TILE_HEIGHT)
#0013                 throw Exception("Bitmap is too small");
#0014
#0015             // 最大是 5 x 5 的圖片群組
#0016             // 圖形切割後的橫軸及縱軸圖片數目
#0017             int XNum = MIN(5, Bits->Width / TILE_WIDTH);
#0018             int YNum = MIN(5, Bits->Height / TILE_HEIGHT);
#0019
#0020             int FirstNo;
#0021             // 依序切割出 XNum * YNum 個圖片
#0022             for (int y = 0; y < YNum; y++)
#0023                 for (int x = 0; x < XNum; x++) {
#0024                     TTile* Tile = new TTile; // 產生圖片物件
#0025
#0026                     // 將對應的圖形複製到圖片的 bitmap 上
#0027                     Tile->Bitmap->Canvas->CopyRect(TILE_RECT, Bits->Canvas,
#0028                         Rect(x * TILE_WIDTH, y * TILE_HEIGHT,
#0029                         (x + 1) * TILE_WIDTH, (y + 1) * TILE_HEIGHT));
#0030
#0031                     Tile->FirstTile = (x == 0 && y == 0); // 是不是群組頭頭
#0032                     Tile->XNum = XNum; // 是群組頭頭的話，負責記錄
#0033                     Tile->YNum = YNum; // 圖片群組的橫軸及縱軸圖片數目
#0034
#0035                     // 將產生的新圖片加入圖庫中
#0036                     if (Tile->FirstTile)
#0037                         // 取得此群組的頭頭編號
#0038                         FirstNo = TTiles::Instance().AddTile(Tile);
#0039                     else
#0040                         TTiles::Instance().AddTile(Tile);
#0041                 }
#0042
#0043             // 將群組描述加入樹狀檢視元件
```

```

#0044     AddTreeNode(dlgOpenPicture->FileName, FirstNo);
#0045
#0046     UpdateControlStatus();
#0047     } __finally {
#0048     delete Bits; // 原始影像沒有用了, 釋放掉
#0049     }
#0050     }
#0051 }
#0052
#0053 void __fastcall TMainForm::mnuRemoveClick(TObject *Sender)
#0054 {
#0055     if (!tvwTiles->Selected) return; // 一定要選定某個群組才行
#0056
#0057     // 取得目前圖片群組首張圖片編號
#0058     int No = (int)tvwTiles->Selected->Data;
#0059
#0060     TTiles& Tiles = TTiles::Instance();
#0061     // 將整個圖片群組的圖片都設為"棄置"
#0062     for (int i = 0; i < Tiles.Tile[No].XNum *Tiles.Tile[No].YNum; i++)
#0063         Tiles.Tile[No + i].Disposed = true;
#0064
#0065     tvwTiles->Selected->Delete(); // 將圖片群組描述砍掉
#0066     UpdateControlStatus();
#0067 }
#0068
#0069 void __fastcall TMainForm::AddTreeNode(AnsiString FileName, int
#0070     FirstNo)
#0071 {
#0072     // 在樹狀檢視元件中加入此圖片群組的節點 (描述)
#0073     // 圖片群組描述預設值為加入的 BMP 圖形檔檔名
#0074     TTreeNode* node = tvwTiles->Items->Add(NULL,
#0075         ExtractFileNameNoExt(FileName));
#0076     node->Data = (void*)FirstNo; // 記錄此群組對應的第一張圖片編號
#0077
#0078     // 若未選擇任何群組, 則幫他選擇第一個節點
#0079     if (tvwTiles->Selected == NULL)
#0080         tvwTiles->Selected = tvwTiles->Items->GetFirstNode();
#0081 }

```

0075 列的 *ExtractFileNameNoExt* 函式由 *xFiles* 單元提供, 傳入一個檔案名稱, 它會傳回除去副檔名後的結果, 我用它來作為群組描述的預設名稱。例如若匯入「綠油油的草地.BMP」, 則此圖片群組的描述就會被設定為「綠油油的草地」。此後你可以隨時經由樹狀檢視元件的修改節點文字功能來改變群組描述。

圖片與圖片群組的關聯

由於樹狀檢視元件的每個節點 (*TTreeNode* 物件) 擁有一個可供使用者自行應用的 *Data* 屬性，因此拿它來儲存此節點所對應的圖片群組再理想也不過了。雖然 *Data* 屬性的資料型態為 *Pointer*，但在 Win32 下，指標為 4 bytes，而整數也為 4 bytes，同樣是那 32 bits，誰也管不著你怎麼用它。所以我直接拿它來儲存所對應圖片群組的首張圖片編號，只不過在指定及讀取時必須進行轉型，如 0058 及 0076 列。

儲存群組頭頭的圖片編號就夠了嗎？是的，因為循著此編號找到首張圖片後，即可取得此圖片群組的橫軸及縱軸圖片數目，那程式就有足夠的資訊來顯示或使用此群組了。如 *mnuRemoveClick* 函式中 0057 ~ 0063 列的動作，由首張圖片取得圖片群組資訊後，就可一一將此群組裡所有圖片設為「棄置」，讓整個圖片群組從此消失。

圖片群組描述的永續性

不過呢，也許你已發現，那些顯示在 *twvTiles* 樹狀檢視元件的圖片群組描述似乎沒有記錄下來，*TTile* 及 *TTiles* 類別似乎也找不著與群組描述相關的欄位及程式碼，那麼這些群組描述是如何保存下來並維持資料永續性呢？答案在這：

```
#0001 void __fastcall TMainForm::mnuOpenClick(TObject *Sender)
#0002 {
#0003     if (dlgOpen->Execute()) // 開啓舊檔對話盒
#0004         try {
#0005             mnuNew->OnClick(NULL); // 先釋放圖庫內容
#0006
#0007             // 載入圖庫
#0008             TTiles::Instance().LoadFromFile(dlgOpen->FileName);
#0009             // 讀取圖片群組描述
#0010             ReadComponentResFile(
#0011                 ChangeFileExt(dlgOpen->FileName, ".TVW"), twvTiles);
#0012
#0013             FFileName = dlgOpen->FileName; // 讀取地形圖庫成功
#0014
#0015             // 若未選擇任何群組，則幫他選擇第一個節點
```

```

#0016         if (!tvwTiles->Selected)
#0017             tvwTiles->Selected = tvwTiles->Items->GetFirstNode();
#0018     } __finally {
#0019         UpdateControlStatus();
#0020     }
#0021 }
#0022
#0023 void __fastcall TMainForm::mnuSaveClick(TObject *Sender)
#0024 {
#0025     // 若是"另存新檔" 或還未指定檔名, 就先問使用者檔名
#0026     if (dynamic_cast<TComponent*>(Sender)->Tag == 1 ||
#0027         FFileName == "") {
#0028         dlgSave->Filter = dlgOpen->Filter;
#0029         // 詢問使用者檔名, 若按取消就離開
#0030         if (!dlgSave->Execute()) return;
#0031         FFileName = dlgSave->FileName; // 將檔名記起來
#0032     }
#0033
#0034     BackupAttr(grdTile->Col, grdTile->Row); // 儲存目前圖片屬性
#0035
#0036     TTiles::Instance().SaveToFile(FFileName); // 儲存圖庫
#0037     // 儲存圖片群組描述
#0038     WriteComponentResFile(
#0039         ChangeFileExt(FFileName, ".TVW"), tvwTiles);
#0040     UpdateControlStatus(); // 更新視窗標題
#0041 }

```

0010 列的 *ReadComponentResFile* 函式及 0038 列的 *WriteComponentResFile* 函式即是關鍵所在，就是這短短的兩行呼叫，解決了群組描述的資料永續需求。這兩個強力函式的原型如下：

```

TComponent* __fastcall ReadComponentResFile(const AnsiString FileName,
    TComponent* Instance);

void __fastcall WriteComponentResFile(const AnsiString FileName,
    TComponent* Instance);

```

兩者參數相同，皆需傳入一個檔案名稱及一個元件。*WriteComponentResFile* 會將 *Instance* 元件的 *__published* 區段屬性值寫入 *FileName* 檔案；而 *ReadComponentResFile* 的功能恰恰相反，將由 *WriteComponentResFile* 寫入的元件資訊讀取回來，回復 *Instance* 元件的原來狀態。藉由這兩支函式，可以很輕鬆、很偷懶地將 VCL 元件的狀態、資料儲存起來，供日後讀取，回復為元件原來的狀態。

Info

這兩支函式內部依賴的正是 VCL 的 streaming 機制，所以儲存格式與 DFM 檔相同。*WriteComponentResFile* 函式並不真正將所有的 *__published* 區段屬性值寫入檔案，而只存入與預設屬性值不同的屬性；除此之外，元件也可自由決定是否儲存額外的內部資訊。

藉著 VCL 的永續機制，這些辛辛苦苦由 *mnuAddClick* 函式建立的群組描述，只要將儲放群組描述節點的 *twvTiles* 整個備份起來，存到檔案中，下次需要的時候再整批還原即可，連帶記錄著群組頭頭圖片編號的 *TTreeNode::Data* 屬性也一併儲存還原，如同啥事都沒發生過，方便極了。

偷懶法的利與弊

這當然不是最理想的辦法，正規的方法是將群組描述隨著首張圖片與其它群組資訊一塊存放，並在讀取圖庫後尋訪所有圖片以重新建立樹狀檢視元件節點，這必須多寫一些程式碼。而我們的偷懶法雖然簡單省事，但天底下總沒有那麼完美的事，這方法的弊端是，將圖庫和群組描述分為兩個獨立的檔案存放，若是不小心刪除其中一個，整套圖庫就毀了，無法使用。而且於架構上，於資料封裝的觀點看來，群組描述由圖片頭頭自行處理才是正道。

不過意外的好處是，因為圖片群組只對圖庫編輯器及地圖編輯器有用，對遊戲主程式則完全沒有意義，因為遊戲中完全不需圖片群組的概念，只消將正確的圖片繪出即可。因此移交遊戲給外界時，只須附上圖庫檔案，群組描述檔案自己留著，可以減少遊戲所佔用的磁碟空間。

預視圖片群組

視窗右上角提供即時預視圖片群組能力的是 *TDrawGrid* 元件，欲讓它可以正確地顯示圖片群組，只要撰寫 *OnDrawCell* 事件處理函式即可：

```
#0001 void __fastcall TMainForm::grdTileDrawCell(TObject *Sender,
#0002     int ACol, int ARow, TRect &Rect, TGridDrawState State)
#0003 {
#0004     if (!tvwTiles->Selected) return; // 一定要選擇某群組才行
#0005
#0006     // 此格所對應的圖片編號 = 首張圖片編號 + ARow * 橫軸數目 + ACol
#0007     int No = (int)tvwTiles->Selected->Data +
#0008         ARow * grdTile->ColCount + ACol;
#0009     if (No >= TTiles::Instance().TileNum) return; // 是否為合法編號？
#0010
#0011     // 畫出對應的圖片
#0012     grdTile->Canvas->Draw(Rect.Left, Rect.Top,
#0013         TTiles::Instance().Tile[No].Bitmap);
#0014 }
```

是否覺得圖庫編輯器的挑戰性不高呢？沒關係，將遊戲所需的圖片群組通通加入圖庫，儲存為 *TILES.TIA* 檔案後，將它擱置一旁。緊接著要將地圖編輯器也一口氣拼出來，小陳飛刀既出，請接招囉。

地圖編輯器

地圖編輯器，按照定義，可稱呼為「以所見即所得方式編輯地圖圖片編號的編輯器」。所以呢，只要忠實地將目前的地圖畫出來，並配合使用者的輸入改變地圖資訊，讓使用者可以輕鬆地編輯地圖，就是成功的地圖編輯器。聽起來一點也不難，那就開始拉元件吧！

哦不，忘了說明一件事，*TMap* 類別除了存放關卡的四層地圖外，另外也儲存每道關卡我們的初始位置，但是我們還沒有寫出角色子系統呀，所以主角位置設計這部分得先略過，為它預留空間，以後待 *TMyTank* 類別完成後再補進去即可。

圖 9-9 是地圖編輯器主視窗的設計時期畫面，看起來，嗯，很沮喪，空洞洞的，只有三個元件，左方的地圖畫面，右上方用來列出圖片群組的樹狀檢視元件及右下角可供預視及選擇圖片的 *TDrawGrid* 元件。

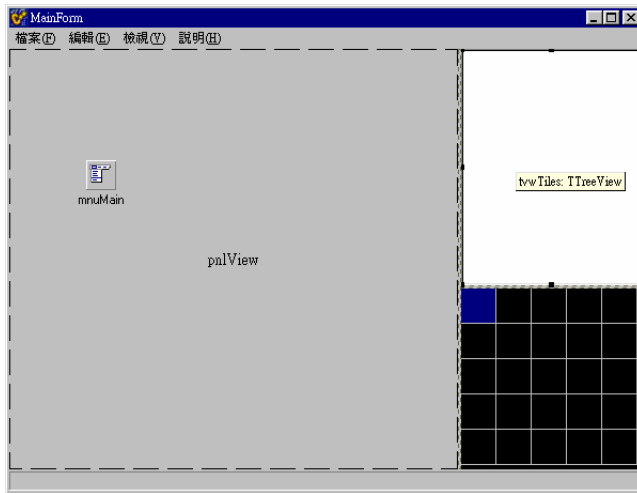


圖 9-9 / 地圖編輯器的設計畫面

地圖編輯器中所使用的技巧，包括使用 *double-buffering* 來繪出地圖、利用 *OnMouseMove* 事件來達成小藍框跟著滑鼠指標跑的效果、繪製特殊區域等等，都與前一章「足球番」的地圖編輯器一模一樣，在此就不重覆介紹。

程式一開始，在 *TMainForm* 的 *OnCreate* 事件處理函式中，首先由全域物件變數 *Tiles* 載入圖庫，接著再呼叫 *ReadComponentResFile* 函式將搭配圖庫使用的樹狀檢視元件資料還原至右上角的 *twTiles*，使圖片群組描述重現；而右下角的 *TDrawGrid*，也和圖庫編輯器中的 *grdTiles* 負責相同的任務，當使用者選擇某圖片群組時，繪出此圖片群組。

靈活的圖片群組操作功能

TDrawGrid 還提供一個功能，讓使用者能在其上拉曳一塊矩形區域，只取圖片群組的某

部分貼到地圖上， \setminus ，十分難表達，請看看圖 9-10 的執行畫面。你瞧，雖然咱們可愛的「草怪」佔有 3×3 張圖片，但我可以只取其左上角 2×2 張，貼在地圖上，很靈活吧。地圖上跟隨著滑鼠指標移動的藍色矩形區域，會隨著你在 *grdTile* 上拉曳的矩形區域大小而變，這能力是由 *TDrawGrid* 的 *Selection* 屬性而得：

```
#0001 // 取得選擇區域的寬及高
#0002 int __fastcall TMainForm::GetSelectionWidth()
#0003 {
#0004     return grdTile->Selection.Right -
#0005         grdTile->Selection.Left + 1;
#0006 }
#0007
#0008 int __fastcall TMainForm::GetSelectionHeight()
#0009 {
#0010     return grdTile->Selection.Bottom -
#0011         grdTile->Selection.Top + 1;
#0012 }
```

GetSelectionWidth 及 *GetSelectionHeight* 函式分別是 *SelectionWidth* 及 *SelectionHeight* 屬性的屬性存取函式，因此在程式中隨時取用這兩個屬性，都能夠正確地回傳目前 *grdTile* 元件使用者選擇的區域大小。



圖 9-10 / 「草怪」群組佔有 3×3 張圖片，但可只取其左上角 2×2 張圖片

地圖編輯模式

編輯模式共有五種，分別是四層地圖層及角色位置設定，如圖 9-11。程式中使用 *FEditLayer* 變數來記錄，若 *FEditLayer* 為 0 ~ *LAYER_MAX*，表示正在編輯對應的圖層，否則為我方坦克位置設定模式。



圖 9-11 / 可選定任一種編輯模式來進行設計工作

地圖圖層的資料設定

程式中最重要的動作就屬按下滑鼠鍵時，對於地圖圖層的資料設定及清除工作了，這份工作是由 *pbxView* 的 *OnMouseDown* 事件處理函式來擔綱：

```
#0001 void __fastcall TMainForm::pbxViewMouseDown(TObject *Sender,
#0002         TMouseButton Button, TShiftState Shift, int X, int Y)
#0003 {
#0004     // 記錄按下的滑鼠鍵，配合 OnMouseMove 事件處理函式
#0005     // 產生拉曳設定效果
#0006     FButtonPressed = Button;
```

```

#0007
#0008   if (Button == mbMiddle) return; // 滑鼠中鍵不做任何事
#0009
#0010   TMap& Map = TMap::Instance();
#0011
#0012   if (Button == mbLeft) { // 左鍵是設定
#0013       if (!tvwTiles->Selected) return; // 沒有選定任何群組
#0014
#0015       int No = (int)tvwTiles->Selected->Data; // 圖片群組的頭頭編號
#0016
#0017       // 左上角, 正上方及右上角三處是敵方坦克的出生點, 不能放東西
#0018       if (FEditLayer != LAYER_TERR && FSelectionY == 0 &&
#0019           (FSelectionX == 0 || FSelectionX == TILE_NUM_X / 2 + 1 ||
#0020            FSelectionX == TILE_NUM_X - 1)) return;
#0021
#0022       if (FEditLayer <= LAYER_MAX) {
#0023           // 將新地形擺上
#0024           for (int MY = 0; MY < GetSelectionHeight(); MY++)
#0025               for (int MX = 0; MX < GetSelectionWidth(); MX++)
#0026               Map.GetCell(FEditLayer, FSelectionX + MX,
#0027                           FSelectionY + MY).TileNo =
#0028                   No + TTiles::Instance().Tile[No].XNum *
#0029                       (MY + grdTile->Selection.Top) +
#0030                       (MX + grdTile->Selection.Left);
#0031       } else {
#0032           // 角色不可以擺在不可走動的地形上
#0033           if (!Map.GetCell(LAYER_TERRITEM, FSelectionX,
#0034                           FSelectionY).CanPass) return;
#0035
#0036           FTank->PosX = TILEWIDTH[FSelectionX]; // 設定主角初始位置
#0037           FTank->PosY = TILEHEIGHT[FSelectionY];
#0038       }
#0039   } else { // 右鍵是清除
#0040       if (FEditLayer > LAYER_MAX) return; // 角色不用清除
#0041
#0042       for (int MY = 0; MY < GetSelectionHeight(); MY++) // 清除此地形
#0043           for (int MX = 0; MX < GetSelectionWidth(); MX++)
#0044               Map.GetCell(FEditLayer, FSelectionX + MX,
#0045                           FSelectionY + MY).TileNo = 0;
#0046   }
#0047
#0048   FModified = true; // 此地圖已更改
#0049   UpdateView(); // 更新地圖畫面
#0050 }

```

0024 ~ 0030 列將新地形擺上時，不但要利用兩層迴圈——設定 *FSelectionX* x *FSelectionY*

個圖格，取得圖片編號時還得小心圖片選擇區域不見得由左上角開始，可能由群組中任一張圖片開始拉曳，所以必須考慮 *grdTile->Selection->Top* 及 *grdTile->Selection->Left* 兩屬性。至於清除地形時，就不用考慮這麼多，通通指定為零就成了。

破碎圖格的編輯能力

有一點沒跟上任天堂版坦克大決戰的地方是，它的地圖編輯器提供破掉一半的磚牆可供編輯，而我們的版本則沒有。其實加入此功能的空間早已預留，你在前頭已經看過，若某個 *TCell* 圖格物件呈破碎狀態，呼叫它的 *SaveToStream* 函式儲存時，*FBreakPtr* 指向的圖格破碎表格也會一併寫入資料流，並且也可從資料流完整地重現圖格破碎表格。依著這樣的設計，只要再加上編輯圖格破碎表格的能力，就可以使用地圖編輯器設計出包含破碎圖格的關卡。這一點都不難，我想差不多也是一塊蛋糕的等級，留待日後再發揮補強囉。嗚，你看，圖 9-13 及圖 9-14 分別是任天堂版及我們的第一道關卡地圖，地圖編輯器才少了一個破碎圖格編輯功能，關卡看起來就遜多了，怨嘆啊。

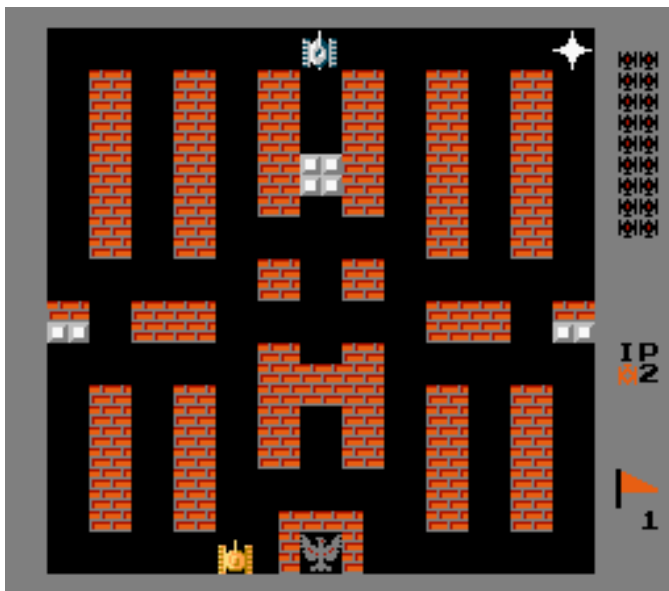


圖 9-13 / 任天堂版第一道關卡



圖 9-14 / 我們的比較遜的第一道關卡

圖層檢視選擇

在操縱地圖編輯器的時候，你可以即時見到對地圖所做的任何更動，不過這樣的操作介面仍有些不便。例如，很可能在某些圖格上放置不必要的圖片，雖然被上層地圖的圖片蓋住了，看不見，但繪製地圖時仍會為它耗費執行時間。所以我加入圖層檢視選擇功能，讓使用者可選擇是否只想見到目前編輯的那層地圖，如圖 9-12，這就是單獨檢視地形物層的結果。



圖 9-12 / 單獨檢視地形物層

就這樣，說著說著地圖編輯器也開始能讓我編些關卡來「看」了。不過只能「看」，不能玩哪，連角色子系統的影子都還沒見著，辛辛苦苦「按」出來的關卡不曉得何時才能身歷其境地玩它一場。唉，還是認命點，左腳都踏入火坑了，右腳豈有不進來溫暖一下的道理，歡迎進入角色子系統的實作。

角色子系統

由圖 9-5 的類別大局觀看來，*TSprite* 類別是角色子系統所有類別的祖先類別，因此只要好好地設計 *TSprite* 類別，處處預留合理的擴充性，讓繼承它的後代類別們做起事來不至綁手綁腳，這是十分重要的設計原則。虛擬及抽象函式也必須運用得當，有許多情況並不是後代類別單純地改寫（*override*）幾道虛擬函式就可達成的；函式的切割、類別之間的任務分派等等規劃事宜，絕對會大大影響實作困難度及成品品質。

對於像 *TSprite* 這麼重要的類別，正是訓練自己類別規劃能力的大好時機，而訓練的方法通常是「設計、實作、驗證、思考、改良、再來一次」這幾個步驟的循環。對於程式架

構的規劃能力，重新撰寫似乎是設計上、實作上絕佳的訓練方法。

還記得高一時，初得知創世紀的作者Richard當初將創世紀一還是創世紀三重寫了二十多遍，只為求得更高竿的規劃能力、更全面俱到的設計觀點、更靈活的程式撰寫技巧時，心中的驚訝及感動，真是無可言喻。我自己也曾嘗試重寫自己以往的作品，每每翻閱幾個月前還自鳴得意寫出的程式碼，幾個月後卻為之臉紅，心想「我怎麼會寫出這種狗屁不通、盤根錯結、旗正飄飄³的程式碼？天啊，我不敢承認這是我寫的！」時，大略可以感受到自己在程式設計方面的進展。

系上有位大我一屆，對於程式設計的概念、思考、經驗、實作上都是駭客級的學長，時常對我說，他哪時又把自己的某某程式或程式庫重寫了一遍，又獲得了好多好多感覺、好多好多感動云云。我只有想著，啊，這真是步入程式設計涅槃境界的絕佳法門呀。

呵，老毛病，又離題遠了，送你一支蒼蠅拍，下回再離題時請拍我回來，謝謝。

以下列出角色子系統所需的常數（定義於 Util.h 中）：

```
const char* DR_IMAGES           = "img\\"; // 角色影像檔的存放目錄
const int SPRITE_DEFAULT_SPEED  = 4;     // 角色的預設速度
const int SMOOTH_MOVE_THRESHOLD = SPRITE_DEFAULT_SPEED * 3 - 1; // 平滑移動的門檻值
const int MYTANK_DEFAULT_SPEED  = 5;     // 我方坦克預設速度
const int BULLET_DEFAULT_SPEED  = 10;    // 子彈預設速度
const int BULLET_DEFAULT_BLOW_RANGE = 4; // 子彈預設爆炸範圍
const int MAX_ETANK_PER_SCENARIO = 20;   // 每道關卡的敵方坦克數目
const int MAX_TANK_ON_SCREEN    = 5;     // 畫面上最多坦克數目
const int MAX_ETANK_COLLISION_COUNT = 5; // 敵方坦克連續碰撞次數上限
const float PROBAB_ETANK_SHOT_BULLET = 0.1; // 敵方坦克射擊子彈機率
const float PROBAB_ETANK_RANDOM_TURN  = 0.02; // 敵方坦克隨意轉彎機率
const float PROBAB_ETANK_BORN         = 0.02; // 敵方坦克出生機率
const float PROBAB_GEM_BORN           = 0.005; // 寶物出現機率
```

³ 當程式邏輯不順時，最直覺也最dirty的方法就是：再加一個flag變數進去！

```

#define TIMER_ID_GEM          1          // 寶物生滅所使用的 Timer 編號
#define TIMER_ID_GEM_CLOCK    2          // 時鐘寶物效果所使用的 Timer 編號
#define TIMER_ID_GEM_HAT      3          // 帽子寶物效果所使用的 Timer 編號
#define TIMER_ID_GEM_ARROW    4          // 弓箭寶物效果所使用的 Timer 編號

// 欲傳遞給遊戲主迴圈的視窗訊息編號
#define WM_DESTROY_OBJECT     (WM_USER + 0) // 摧毀物件指令
#define WM_GAMEOVER           (WM_USER + 1) // 遊戲結束訊息
#define WM_SPECIAL_CONDITION   (WM_USER + 2) // 吃到寶物時產生效果指令
#define WM_INIT_LEVEL         (WM_USER + 3) // 關卡重新開始

```

TSprite 類別

廢話不多說，請進入戰戰兢兢、冷汗直冒的備戰狀態，我們面對的是整套遊戲最艱難的 *TSprite* 類別（定義於 *Sprite.h*）：

```

#0001 enum TDirection {drUp, drDown, drLeft, drRight}; // 方向
#0002
#0003 // 屬性
#0004 enum TSpriteAttrElement {
#0005     saUndirectionalBitmap, // 圖形不具方向性
#0006     saNoCellCollision, // 不跟圖格碰撞
#0007     saNoTankCollision, // 不跟坦克碰撞
#0008     saNoBulletCollision, // 不跟子彈碰撞
#0009     saAlignWithTerrItem} ; // 走動時會自動對齊地形物，走起來比較順
#0010
#0011 typedef Set<TSpriteAttrElement, saUndirectionalBitmap,
#0012     saAlignWithTerrItem> TSpriteAttr;
#0013
#0014 // 動畫結束後是否停止或重頭開始
#0015 enum TAdvanceFrameMode {afWrap, afStop} ;
#0016
#0017 // 傳回碰撞結果的陣列型態
#0018 typedef std::vector<TCell*> TCellArray;
#0019 typedef std::vector<TSprite*> TSpriteArray;
#0020
#0021 // 在畫面上行走活動的角色物件
#0022 class TSprite {
#0023 private:
#0024     int FX, FY; // 座標
#0025     int FFrameNo; // 目前顯示的 frame 編號
#0026     int FCollisionCount; // 連續碰撞次數

```

```

#0027
#0028     bool FActive; // 是否進行動作
#0029     bool FVisible; // 是否可見
#0030     bool FOnAir; // 是否在天空
#0031     int FSpeed; // 行進速度
#0032
#0033     TRect FRect; // 佔用矩形區域
#0034     TDirection FDirection; // 行進方向
#0035     TSpriteAttr FAttr; // 屬性
#0036
#0037     bool FPostToDead; // 是否已登記要摧毀
#0038
#0039     Graphics::TBitmap *FBits, *FInvBitmap; // 圖片及貼圖用圖片
#0040
#0041     // 角色中心點所在的圖格位置
#0042     int GetTile_X();
#0043     int GetTile_Y();
#0044
#0045     // 取得角色的寬及高度
#0046     int GetObjectWidth();
#0047     int GetObjectHeight();
#0048
#0049     void SetDirection(TDirection Value);
#0050
#0051     // helper functions for CheckCollisions()
#0052     bool IsRealCellCollision(int Layer, int CellXPos, int CellYPos,
#0053         int& x, int& y);
#0054     void CheckCellCollisionsPos(TCellArray& Collisions, const TRect&
#0055         SpriteRect, int Layer, int CellXPos, int CellYPos,
#0056         int& x, int& y);
#0057     protected:
#0058     // 不同角色有不同的資訊
#0059     TRect FObjectRect; // 角色尺寸
#0060     AnsiString FFileName; // 角色圖形檔名
#0061     int FFrameMax; // 動畫框數
#0062
#0063     int FMoveDelay, FMoveDelayCount; // 下次動作前的延遲次數
#0064     TAdvanceFrameMode FAdvanceFrameMode; // 動畫結束後處理方式
#0065
#0066     virtual void ResetStatus(); // 重設角色狀態
#0067
#0068     // 碰撞檢查觸發函式，負責呼叫所有的碰撞檢查函式
#0069     virtual bool CheckCollisions(int& x, int& y);
#0070
#0071     // 邊界碰撞檢查
#0072     virtual bool CheckBoundCollisions(int& x, int& y);

```

```
#0073 // 地形物碰撞檢查
#0074 virtual bool CheckCellCollisions(int Layer, int& x, int& y,
#0075     TCellArray& Collisions);
#0076 // 角色碰撞檢查
#0077 virtual bool CheckSpriteCollisions(TSpriteArray& Sprites,
#0078     int& x, int& y, TSpriteArray& Collisions);
#0079 public:
#0080     TSprite();
#0081     virtual ~TSprite();
#0082
#0083     void LoadBits(); // 載入角色圖形
#0084
#0085     virtual void Draw(TCanvas* Canvas); // 繪製角色
#0086
#0087     virtual void Move(); // 進行下一步動作
#0088
#0089     void PostToDie(WPARAM Param = 0); // 登記欲摧毀本身
#0090
#0091     void RandomDirection(); // 隨意選擇方向
#0092     void RandomPosition(); // 隨意擺置
#0093     void CenterWith(TSprite& ASprite); // 與另一角色置中對齊
#0094     void CenterBy(int x, int y); // 使中心點為 (X, Y)
#0095
#0096     __property int PosX = {read = FX, write = FX};
#0097     __property int PosY = {read = FY, write = FY};
#0098
#0099     __property int Tile_X = {read = GetTile_X};
#0100     __property int Tile_Y = {read = GetTile_Y};
#0101
#0102     __property TDirection Direction =
#0103         {read = FDirection, write = SetDirection};
#0104     __property int Speed = {read = FSpeed, write = FSpeed};
#0105     __property bool Active = {read = FActive, write = FActive};
#0106     __property bool Visible = {read = FVisible, write = FVisible};
#0107     __property bool OnAir = {read = FOnAir, write = FOnAir};
#0108     __property bool PostToDead = {read = FPostToDead};
#0109     __property int CollisionCount = {read = FCollisionCount};
#0110
#0111     __property TRect Rect = {read = FRect};
#0112     __property TSpriteAttr Attr = {read = FATtr, write = FATtr};
#0113
#0114     __property TRect ObjectRect = {read = FObjectRect};
#0115     __property int ObjectWidth = {read = GetObjectWidth};
#0116     __property int ObjectHeight = {read = GetObjectHeight};
#0117 };
```

大部分的變數定義及函式宣告都已加上詳細的註解，請多瀏覽幾回。這兒的 *TSprite* 類別和你心目中的 *TSprite* 類別有哪些相異處？爲什麼？孰優孰劣？爲什麼？多多問自己類似的問題，多多思考，有助於規劃能力的增長。

0026 列宣告的 *FCollisionCount* 變數用來偵測角色的「碰壁」狀況，每當碰到東西時就加一，若順利行走，什麼東西都沒碰到則歸零，改變方向時也歸零。如此一來，此變數就可視爲「連續碰壁計數器」，對於亂數控制的角色而言，若此計數值大於某個上限時，就必須請它轉彎，否則一直卡在牆壁旁或邊界，很難看的耶！

0030 列 *FOnAir* 設定此角色「是否在空中」，這是配合本遊戲的四層地圖層而設。畫面重繪的步驟是，先繪出地形層、地形物層及物品層，接著才畫出角色，最後畫出高地形物層。但是這樣一來就無法將飛行器加入遊戲中，哪有飛機會飛在樹棚底下的呀。所以我再加上 *OnAir* 屬性，若其值爲 *true*，則此角色會在高地形層之後才繪出，*OnAir* 屬性爲 *False* 的角色則按照原設定，在高地形物層之前繪出（也就是在下面）。待會你就會看到，爲了示範這個屬性，我真的很在咱們的坦克大決戰中加入飛行器了！:p

0033 列定義 *FRect*，記錄著此角色所佔用的矩形區域，會隨著角色移動自動更新。我們必須使用此矩形區域來測試碰撞情形，還記得嗎？*TCell* 圖格類別也有個 *Rect* 屬性，將角色的 *Rect* 及圖格的 *Rect* 做交集運算，便可得知角色有沒有撞到地形物了。不過使用矩形做爲碰撞運算單位是極危險的事，萬一你的坦克長的像十字架那種形狀，也就是在其佔用矩形區域內，真正佔用面積極少的情況，就很容易發生子彈明明距離坦克本體還好幾步，但是莫名其妙就爆炸了，這不是因爲坦克有替身使者⁴，是因爲它們的矩形區域已經產生碰撞了。但是以矩形區域來測試碰撞是最有效率的辦法，這是速度與品質的交易，也留待日後改善。

將 *FObjectRect*、*FFileName*、*FFrameMax* 三個變數宣告於 *protected* 區段的用意是，讓後代類別去改變它們，以取得該類別使用的圖形檔名、動畫框數及角色尺寸。*FObjectRect*

⁴ 日本漫畫「Jo Jo冒險野郎」裏許多角色擁有的特殊能力，例如承太郎的「白金之星」可使全世界暫停十幾秒。十分好看的漫畫，筆者強力推薦。:P

預設值設為地圖圖格大小的矩形區域（即 *TILE_WIDTH* x *TILE_HEIGHT*），後代類別的角色若與圖格同樣大小的話就不必改變它。

0063 列 *FMoveDelay* 及 *FMoveDelayCount* 是針對動作特別遲緩的角色而設，每當正常角色行動 *FMoveDelay* 次，它才行動一次。可用於播放爆炸、閃爍效果，讓一個動畫在畫面上持續久一點，而不是一閃而逝。

碰撞處理總管

0069 列的 *CheckCollisions* 函式極為重要，每一步行動時都會呼叫它來測試角色的碰撞情況。*CheckCollisions* 函式本身並不進行碰撞測試，只負責呼叫所有的碰撞處理函式。*TSprite* 類別提供三種碰撞處理函式，分別是測試邊界碰撞的 *CheckBoundCollisions* 函式、測試地形物碰撞的 *CheckCellCollisions* 函式及測試角色相互碰撞的 *CheckSpriteCollisions* 函式。

CheckCollisions 是個虛擬函式，所以後代類別可以改寫它，讓它呼叫更多種類的碰撞處理函式，例如主角的 *TMyTank* 類別就改寫它以呼叫 *TMyTank* 的寶物碰撞處理函式；而 *TBullet* 類別也改寫它加入子彈碰撞處理函式等等。

這種處理方式可使碰撞測試處理的開放性大增，不必將所有的碰撞測試函式通通放在 *TSprite* 類別，只要提供一個嵌入點（*CheckCollisions* 函式），有需要的後代類別便可將它專屬的碰撞測試函式納入。

所有的碰撞處理函式都宣告為虛擬函式，讓後代類別有機會改寫。它們都提供傳回值，傳回「判定為碰撞對象的物件陣列」，這使後代類別改寫碰撞處理函式時，可對父類別的判定結果重新翻案，本來被打入黑五類的傢伙也可來個鹹魚大翻身。舉例而言，*TTile* 類別有個 *taBulletCanPass* 屬性，指這張圖片無論是否能被角色通過，子彈一定可以通過。假設地形物層的某個圖格是不能通過的（不含 *taCanPass* 屬性），當角色經過此圖格時，*TSprite::CheckCellCollisions* 地形物碰撞處理函式會將此圖格納入碰撞名單內，所以坦克

及其它角色都會卡住，無法通過。但是子彈類別 *TBullet*，改寫 *CheckCellCollisions* 函式，取得重新審核的機會，若被判定為碰撞的圖格包含 *taBulletCanPass* 屬性的話，就無條件釋放，改判無碰撞。這種比蕭蕾腿上絲襪更具彈性的「多審制度」，賦予各種物件完全宰制行動的能力，完全避免類別設計不良所帶來的挖東牆補西牆或旗正飄飄的可能性。

角色圖形

角色的圖形採用很簡單的處理方式：角色圖形可為無方向性（例如圓形的飛碟，怎麼轉都是同一個模樣）或上下左右四個方向；而每個方向可有任意張數的輪替動畫。這些圖形通通放在同一張 BMP 圖形，繪製角色時再依需求將對應的區域拷貝出來。

以遊戲中的主角坦克為例，它有四個方向的圖形，每個方向兩張，一律為 32 x 32 大小。

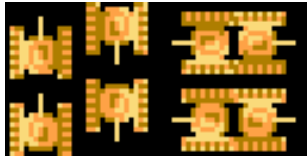


圖 9-15 / 主角坦克的角色圖形檔

從主角坦克的角色圖形可看出，四個方向上下左右的圖形依序由左至右排列，而每個方向的動畫則由上至下置放。只要按照規矩設計角色圖形，並適當地改寫虛擬函式宣告類別使用的角色圖形檔名、角色尺寸及動畫張數，就可讓角色擁有正確的外觀。

天經地義，繪製角色圖形的工作由 *TSprite::Draw* 函式來負責：

```
#0001 void TSprite::Draw(TCanvas* Canvas)
#0002 {
#0003     //若不可見或已登記摧毀則不畫
#0004     if (!FVisible || FPostToDead) return;
#0005
#0006     // 動畫框編號不合法則不畫
#0007     if (FFrameNo > FFrameMax || FFrameNo < 0) return;
#0008
#0009     // 先將要秀出的區域拷至 FInvBitmap
```



```

#0010 // 若只有一張圖形，不分方向性
#0011 if (Attr.Contains(saUndirectionalBitmap))
#0012     FInvBitmap->Canvas->CopyRect(ObjectRect, FBits->Canvas,
#0013     Classes::Rect(0, FFrameNo * ObjectHeight, ObjectWidth,
#0014     (FFrameNo + 1) * ObjectHeight));
#0015 else // 依目前方向
#0016     FInvBitmap->Canvas->CopyRect(ObjectRect, FBits->Canvas,
#0017     Classes::Rect((int)FDirection * ObjectWidth,
#0018     FFrameNo * ObjectHeight, ((int)FDirection + 1) * ObjectWidth,
#0019     (FFrameNo + 1) * ObjectHeight));
#0020
#0021 // 再畫出 FInvBitmap
#0022 Canvas->Draw(FX, FY, FInvBitmap);
#0023 }

```

角色圖形繪製一律採用透明貼圖，依它的圖形是否具有方向性而有不同的座標計算公式，這些座標計算看似複雜，實則簡單，不信你細看便知，皆是十分直覺的倍數運算罷了。

角色的運作

角色一定要會活動才叫角色，否則就成了盆栽。*TSprite* 類別裡的運作核心為 *Move* 函式，幾乎可說 *TSprite* 類別所有的變數、狀態、屬性及函式都與 *Move* 函式有直接或間接的關係。

每次呼叫 *Move* 函式，該角色就進行「一動」，這「一動」，包括：

- 若角色已登記準備被摧毀 (*FPostToDead* 為 *true*) 或停止運作 (*FActive* 為 *false*)，則角色全部活動停止。
- 若有行動延遲設定，則待行動延遲次數達到設定值後才真正行動。
- 若有多張動畫，則換下一張。若動畫已播到最後一張且 *FAdvanceFrameMode* 為 *afStop*，則此角色停止運作 (*FActive* 變成 *false*)。
- 計算下一步的位置，然後針對此新座標進行各種碰撞測試，碰撞測試處理可能帶來各種影響，可能是修正座標，也可能引發爆炸等等。

□ 更新角色座標。

遊戲進行中會不斷呼叫所有角色的 *Move* 函式，每個輪迴各呼叫一次，每秒鐘至少十二個輪迴，這可使得遊戲中的角色們持續活動，栩栩如生，像真的有生命一樣。

TSprite::Move 函式程式碼如下：

```
#0001 // 最重要的動作函式，控制角色的動作
#0002 void TSprite::Move()
#0003 {
#0004     if (FPostToDead) return; // 已經登記準備被摧毀了
#0005     if (!FActive) return; // 不進行任何動作
#0006
#0007     // 每次動作前的延遲次數
#0008     if (FMoveDelay != 0) {
#0009         // 每 FMoveDelay 次才真正 Move 一次
#0010         FMoveDelayCount++;
#0011         if (FMoveDelayCount != FMoveDelay) return;
#0012         FMoveDelayCount = 0; // 歸零
#0013     }
#0014
#0015     if (FFrameMax > 0) { // 若有動畫的話
#0016         FFrameNo++; // 遞增動畫編號
#0017         if (FFrameNo > FFrameMax) { // 播放一個輪迴了
#0018             if (FAdvanceFrameMode == afWrap)
#0019                 FFrameNo = 0; // 重新再來
#0020             else {
#0021                 FFrameNo = FFrameMax; // 維持在最後一張動畫
#0022                 FActive = false; // 停止動作
#0023             }
#0024         }
#0025     }
#0026
#0027     if (FSpeed == 0) return; // 若速度為零，不必移動
#0028
#0029     // 取得目前位置
#0030     int x = FX;
#0031     int y = FY;
#0032
#0033     // 計算下一步的位置
#0034     switch (FDirection) {
#0035         case drUp: y -= FSpeed; break; // 往上走
#0036         case drDown: y += FSpeed; break; // 往下走
#0037         case drLeft: x -= FSpeed; break; // 往左走
#0038         case drRight: x += FSpeed; break; // 往右走
```

```

#0039 }
#0040
#0041 // 檢查所有碰撞
#0042 if (CheckCollisions(x, y))
#0043     FCollisionCount++; // 撞到什麼東西了...
#0044 else
#0045     FCollisionCount = 0; // 什麼都沒撞到, 連續碰撞計數器歸零
#0046
#0047 // 更新位置
#0048 FX = x;
#0049 FY = y;
#0050
#0051 // 更新佔用的矩形區域
#0052 FRect = ObjectRect;
#0053 OffsetRect(&FRect, FX, FY);
#0054 }

```

0042 列呼叫的 *CheckCollisions* 函式是 *Move* 函式的關鍵，也是將具有活動力的角色與位於背景的地圖子系統連結起來的繩引。此處若未進行碰撞處理，則一堆角色在畫面上只是愚蠢地行動著，絲毫不受腳下地形的影響，跋山涉水，如履平地。這還不算什麼，若是於角色之間也沒有碰撞處理，就可以看到一堆坦克疊在一起滑動，太拙了。

麻煩的碰撞檢查

CheckCollisions 函式並不真正擔任碰撞檢查者的工作，它的角色像是「碰撞檢查部門」的主管，負責發號司令，所有的碰撞檢查工作都必須透過它來實行。

TSprite::CheckCollisions 程式碼如下：

```

#0001 // (X, y) 是欲使用的新座標, 傳回值表示是否發生任何碰撞
#0002 bool TSprite::CheckCollisions(int& x, int& y)
#0003 {
#0004     if (FPostToDead) return false; // 已經登記準備被摧毀了
#0005
#0006     bool bCollision = false;
#0007
#0008     // 檢查是否撞到地形物
#0009     if (!bCollision && !FAttr.Contains(saNoCellCollision)) {
#0010         TCellArray CellCollisions;
#0011         bCollision = CheckCellCollisions(LAYER_TERRITEM, x, y,

```

```
#0012     CellCollisions);  
#0013 }  
#0014  
#0015 // 檢查是否撞到邊界  
#0016 if (!bCollision)  
#0017     bCollision = CheckBoundCollisions(x, y);  
#0018  
#0019     return bCollision;  
#0020 }
```

0002 列中的參數 x 及 y 指角色的新座標，也就是用來測試碰撞的座標。所有碰撞處理函式皆需要 x 、 y 這兩個參數，並且以傳址方式傳遞，使碰撞處理函式擁有修正新座標的能力，待會你就可以看到，座標將如何被修正。

此函式傳回一布林值，表示這一動是否發生任何碰撞。在 *Move* 函式中，若 *CheckCollisions* 傳回 *true*，則遞增連續碰撞計數值 *FCollisionCount*；若為 *false*，則將 *FollisionCount* 重置為零。

TSprite::CheckCollisions 函式中，依序進行兩種基本的碰撞測試，分別是角色與地形物、角色與邊界的碰撞測試。因為 *CheckCollisions* 是虛擬函式，所以後代類別可以改寫它，加入新的碰撞測試。

每當完成一項碰撞測試，若此項碰撞測試得知有碰撞發生，該碰撞處理函式就會將傳入的布林參數設為 *true*，然後返回。而呼叫碰撞處理函式的 *CheckCollisions* 函式得知後，就會省略其它的測試工作，直接跳離函式。這種做法的特性是，一個角色在「一動」中只會發生一種碰撞，亦即，它不是碰到牆壁，就是撞上其它角色，要不然就是超出邊界，絕不會有兩者以上同時發生的情況。好處是可以省下碰撞處理的時間，其次是簡化碰撞結果的處理。否則，當一顆子彈同時打到磚牆及坦克時，要在哪邊爆炸？還是兩者皆爆？何況若兩者皆爆的話，就得為每顆子彈建立一個爆炸物件陣列，多麻煩哪。

邊界碰撞測試

TSprite::CheckCollisions 所呼叫的三種碰撞測試中，以邊界測試最為簡單：

```
#0001 bool TSprite::CheckBoundCollisions(int& x, int& y)  
#0002 {
```

```
#0003   int OrgX, OrgY;
#0004
#0005   // 邊界檢查
#0006   OrgX = x; OrgY = y;
#0007
#0008   if (y < 0) y = 0; // 是否超出上方
#0009   if (x < 0) x = 0; // 是否超出左方
#0010   if (x + ObjectWidth >= WORLD_WIDTH) // 是否超出右方
#0011       x = WORLD_WIDTH - ObjectWidth;
#0012   if (y + ObjectHeight >= WORLD_HEIGHT) // 是否超出下方
#0013       y = WORLD_HEIGHT - ObjectHeight;
#0014
#0015   // 是否撞到邊界 ?
#0016   return (OrgX != x || OrgY != y);
#0017 }
```

這個函式中，分別檢查角色的四邊是否超出畫面的範圍了，若是的話，則直接修正其座標，讓角色由超出畫面邊緣成為緊貼著畫面邊緣，這樣一來，無論再怎麼走，角色都不可能離開畫面範圍，十分簡單又有力的邊界檢查。

地形物碰撞測試

檢查地形物碰撞的 *CheckCellCollisions* 函式，由於程式碼實在太長，礙於篇幅無法列出，不過它的檢查方式可以一言蔽之，就是針對以角色為中心的九個圖格（角色所在圖格加上周圍環繞的八個圖格），一一檢查圖格的矩形區域與角色的新矩形區域是否產生碰撞（呼叫 *IntersectRect* API 函式來判斷），函式十分直覺簡單，只是一堆座標的處理使程式碼寫來十分冗長而已。

另外，*CheckCellCollisions* 函式還包括走動時自動切齊地形物的座標修正功能。此功能是針對擁有 *saAlignWithTerrItem* 屬性的角色設計，本遊戲中所有坦克都具有此屬性，這個功能會讓我們在操作坦克時，若要走進或彎入一條巷道中時，即使坦克位置不是對得很準，只要差距小於 *SMOOTH_MOVE_THRESHOLD* 點數，此功能就會自動幫你修正座標，讓玩者不必為了讓坦克走某條巷道就得瞄準個好半天。

角色碰撞測試

CheckSpriteCollisions 會檢查角色與其它角色的碰撞情形，並傳回與該角色發生碰撞的角

色陣列（因為可能同時撞上好多個角色）：

```
#0001 bool TSprite::CheckSpriteCollisions(TSpriteArray& Sprites,
#0002     int& x, int& y, TSpriteArray& Collisions)
#0003 {
#0004     TRect SpriteRect, R;
#0005
#0006     // 計算新的矩形區域
#0007     SpriteRect = ObjectRect;
#0008     OffsetRect(&SpriteRect, x, y);
#0009
#0010     // 一一尋訪傳入的角色
#0011     for (TSpriteArray::iterator p = Sprites.begin();
#0012         p != Sprites.end(); p++) {
#0013         TSprite* &S = *p;
#0014         TRect SRect = S->Rect;
#0015
#0016         if (this != S && !S->PostToDead && S->Visible &&
#0017             IntersectRect(&R, &SpriteRect, &SRect)) {
#0018
#0019             // 將撞到的角色加入碰撞結果陣列
#0020             Collisions.push_back(S);
#0021         }
#0022     }
#0023
#0024     return Collisions.size() > 0; // 表示有碰撞發生
#0025 }
```

CheckSpriteCollisions 函式需要傳入一個 *TSpriteArray* 陣列 *Sprites* 參數，包含所有要拿來進行碰撞測試的角色，此函式會將碰撞到的角色加入同樣是 *TSpriteArray* 陣列型別的 *Collisions* 參數。由於並不是所有角色物件都需要主動進行與其它角色的碰撞測試，因此 *CheckSpriteCollisions* 函式目前只是備而不用，若 *TSprite* 的子類別需要進行角色的碰撞測試，只要改寫 *CheckCollisions* 函式，在於其中呼叫 *CheckSpriteCollisions* 函式即可。

至於 *TSprite* 其它的變數、函式及屬性皆屬座標、面積、屬性管理之類，註解說明得十分清楚，不再一一介紹。

TSprite 類別至此可說是大功告成，因為它已預留許多空間，供後代類別改寫發揮，角色子系統可說完成了一半。接下來的工作就簡單了，依序由 *TSprite* 類別衍生出角色子系統的其他類別，按圖索驥，將圖 9-5 類別大局觀的類別依階層關係一個個實作出來便成。

TTank 坦克抽象類別

最早的構想是，從 *TSprite* 類別衍生出一個坦克類別，供遊戲中所有坦克使用。不過再仔細想想，我方坦克及敵方坦克差異頗大，例如說我方坦克可吃寶物；敵方坦克可由亂數決定其行動、在出生前會先發出金光閃閃效果等等。這些功能全部由同一個類別負責容易使類別十分複雜，難以處理，所以決定先建立一個擁有坦克功能及特性的抽象類別 *TTank*，我方坦克及敵方坦克再分別由 *TTank* 建立自己的類別。

TTank 類別宣告如下（定義於 *Sprite.h* 中）：

```
#0001 // 坦克類別，我方及敵方坦克皆從此類別繼承
#0002 class TTank : public TSprite {
#0003 private:
#0004     int FHP; // 裝甲（生命力）
#0005     int FBulletNum; // 目前子彈數
#0006     bool FSuperMode; // 無敵模式
#0007
#0008     friend class TBullet;
#0009 protected:
#0010     // 每種角色不同
#0011     int FScore; // 被摧毀後，主角的得分
#0012     int FMaxBulletNum; // 同一時間子彈數上限
#0013     int FBulletBlowRange; // 子彈爆炸威力（範圍）
#0014
#0015     // 碰撞檢查觸發函式，負責呼叫所有的碰撞檢查函式
#0016     virtual bool CheckCollisions(int& x, int& y);
#0017
#0018     // 新增坦克碰撞檢查
#0019     virtual bool CheckTankCollisions(int& x, int& y,
#0020         TSpriteArray& Collisions);
#0021 public:
#0022     TTank();
#0023     virtual ~TTank();
#0024
#0025     virtual void ResetStatus(); // 重設坦克狀態
#0026     virtual TBullet* FireBullet(); // 發射子彈
#0027
#0028     __property int HP = {read = FHP, write = FHP};
#0029     __property int MaxBulletNum =
#0030         {read = FMaxBulletNum, write = FMaxBulletNum};
#0031     __property bool SuperMode =
```

```
#0032     {read = FSuperMode, write = FSuperMode};  
#0033  };
```

0013 列宣告 *FBulletBlowRange* 變數，記錄此坦克所發射子彈的爆炸威力（範圍），這是不同於任天堂版坦克大決戰的一點改良，因我們完全使用矩形區域來進行碰撞測試，包括子彈爆炸範圍及可爆破地形物的矩形交集測試。透過此爆炸威力的設定，遊戲中的子彈不再只能呆呆地永遠只破壞八分之一或四分之一塊磚牆，威力弱一點的子彈可能只損壞十六分之一個圖格（因為 *TCell* 的圖格破碎表格為 4 x 4 大小），威力強大的子彈甚至能夠一口氣毀壞九個圖格（因為地形物的碰撞測試只針對圍繞角色的八塊圖格及角色本身所在圖格進行測試），威力相差 144 倍。

咻～子彈發射

TTank 類別除了加入一堆坦克資訊，如生命力、子彈數目、爆炸範圍外，最有趣的功能就屬 0026 列宣告的 *FireBullet* 子彈發射函式了。想想，若沒有發射子彈的能力，坦克大決戰就不再是坦克大決戰，而是一拖拉庫坦克互相輾來輾去的「碰碰坦克」，多無趣啊。

發射子彈是典型的「由角色產生其它角色」的例子，*FireBullet* 函式程式碼如下：

```
#0001  // 發射子彈，傳回值為發射出去的子彈物件  
#0002  TBullet* TTank::FireBullet()  
#0003  {  
#0004  // 目前子彈數若已達上限則不容許再發射  
#0005  if (FBulletNum >= FMaxBulletNum) return NULL;  
#0006  
#0007  TBullet* b = new TBullet(this); // 建立子彈物件  
#0008  b->LoadBits(); // 載入子彈圖形  
#0009  b->Direction = Direction; // 子彈與坦克本身同樣方向  
#0010  
#0011  b->CenterWith(*this); // 先將子彈座標設定與坦克置中對齊  
#0012  
#0013  switch (Direction) { // 根據行進方向來調整子彈座標  
#0014  case drUp: b->PosY = Rect.Top; break;  
#0015  case drDown: b->PosY = Rect.Bottom - b->ObjectHeight; break;  
#0016  case drLeft: b->PosX = Rect.Left; break;  
#0017  case drRight: b->PosX = Rect.Right - b->ObjectWidth; break;  
#0018  }
```



```
#0019
#0020  FBulletNum++; // 遞增坦克的目前子彈數
#0021  return b;
#0022  }
```

這段程式碼十分易懂，首先檢查目前子彈數目是否已達上限，然後建立子彈物件，適當地設定它的屬性，最後再遞增坦克的目前子彈數目。這顆子彈會自動加入遊戲所維護的子彈陣列中，和其它角色一樣朝著自己眼前的方向一步一步行進，直到擊中什麼東西為止。

FireBullet 函式會傳回剛發射出去的子彈物件，而且宣告為虛擬函式，這與碰撞處理函式有異曲同工之妙—留給後代類別不反既定事實的機會。後代類別若有需要，即可改寫 *FireBullet* 函式，取得 *TTank::FireBullet* 所發射的子彈物件，將此子彈物件修改為更合適的狀態，才放行。

坦克與坦克的碰撞行為

TTank 類別加入 *CheckTankCollisions* 函式來處理坦克與坦克的碰撞行為。*CheckTankCollisions* 函式藉由 *TSprite::CheckSpriteCollisions* 函式的輔助，將畫面上所有坦克傳入，取得碰撞角色列表後，一一重新審查，若發現碰撞的坦克擁有 *saNoTankCollision* 屬性，則此坦克不應該与其它坦克發生碰撞，就將它自碰撞結果陣列移除。程式碼列表如下：

```
#0001  bool TTank::CheckTankCollisions(int& x, int& y, TSpriteArray&
#0002  Collisions)
#0003  {
#0004  // 首先呼叫 CheckSpriteCollisions 函式取得碰撞坦克列表
#0005  TSpriteArray& r_Tanks = *Tanks(); // r_Tanks 指向所有坦克列表
#0006  bool bCollision = TSprite::CheckSpriteCollisions(r_Tanks, x, y,
#0007  Collisions);
#0008
#0009  if (bCollision) { // 若與任何坦克發生碰撞
#0010  for (TSpriteArray::iterator p = Collisions.begin();
#0011  p != Collisions.end(); ) {
#0012  TSprite* &T = *p;
#0013
```

```

#0014     if (T->Attr.Contains(saNoTankCollision))
#0015         Collisions.erase(p);
#0016     else
#0017         p++;
#0018     }
#0019
#0020     bCollision = Collisions.size() > 0; // 重新裁決是否發生碰撞
#0021     // 亡羊補牢, 猶未晚也. 其實根本不算有碰撞的..
#0022     if (!bCollision) return false;
#0023
#0024     // 根據碰撞到的坦克及自己的方向來調整座標
#0025     for (TSpriteArray::iterator p = Collisions.begin();
#0026         p != Collisions.end(); p++) {
#0027         TSprite* &T = *p;
#0028
#0029         switch (Direction) {
#0030             // 把自己放在對方的下方
#0031             case drUp: y = T->Rect.Bottom; break;
#0032             case drDown: y = T->Rect.Top - ObjectHeight; break;
#0033             // 把自己放在對方的右方
#0034             case drLeft: x = T->Rect.Right; break;
#0035             case drRight: x = T->Rect.Left - ObjectWidth; break;
#0036         }
#0037     }
#0038 }
#0039 return bCollision;
#0040 }

```

我方坦克

坦克的抽象類別也定義好後，只要再由其分別衍生我方坦克及敵方坦克，稍加改寫即可。

我方坦克 *TMyTank* 繼承自 *TTank* 類別，改寫 *CheckCollisions* 碰撞觸發函式，呼叫 *TMyTank::CheckGemCollisions* 寶物碰撞處理函式來檢查我方坦克與寶物的碰撞情形。

寶物為 *TSprite* 的子類別 *TGem* 物件，由於遊戲中任一時間最多只可能有一個寶物存在，所以我宣告一個全域的 *TGem* 物件，*Gem*。當它為 *NULL* 時，表示目前畫面上沒有寶物，否則指向該寶物物件。

我方坦克加入吃掉寶物能力的程式碼為：

```
#0001 bool TMyTank::CheckGemCollisions(int& x, int& y)
#0002 {
#0003     if (!Gem) return false; // 寶物不存在, 不可能碰到
#0004
#0005     // 根據新座標計算坦克所佔用的矩形區域
#0006     TRect SpriteRect = ObjectRect;
#0007     OffsetRect(&SpriteRect, x, y);
#0008
#0009     // 坦克是否與寶物所佔矩形區域產生交集
#0010     TRect R, R1 = Gem->Rect;
#0011     if (IntersectRect(&R, &SpriteRect, &R1))
#0012         return true; // 吃到寶物了
#0013
#0014     return false;
#0015 }
#0016
#0017 bool TMyTank::CheckCollisions(int& x, int& y)
#0018 {
#0019     bool bCollision = TTank::CheckCollisions(x, y);
#0020
#0021     // 如果吃到寶物的話 (不影響 bCollision) ...
#0022     if (CheckGemCollisions(x, y)) {
#0023
#0024         // 根據寶物的種類, 傳送訊息給遊戲迴圈處理, 或自己處理掉
#0025         switch (Gem->GemKind) {
#0026             case gkClock:
#0027                 PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_CLOCK, 0);
#0028                 break;
#0029
#0030             case gkHat:
#0031                 PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_HAT, 0);
#0032                 break;
#0033
#0034             case gkArrow:
#0035                 PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_ARROW, 0);
#0036                 break;
#0037
#0038             case gkStar: FMaxBulletNum = 10; // 能夠連發子彈
#0039                 break;
#0040
#0041             case gkBlow: FBulletBlowRange = 32; // 超強子彈爆炸威力
#0042                 break;
#0043
#0044             case gkApple: Speed = 2 * MYTANK_DEFAULT_SPEED; // 兩倍速度
#0045                 break;
#0046         }
```

```
#0047
#0048 // 將寶物摧毀(被吃掉了)
#0049     delete Gem;
#0050 }
#0051
#0052 return bCollision;
#0053 }
```

若 *CheckGemCollision* 函式回傳 *true*，確定吃到寶物後，0025 列會根據寶物的型態做出適當的回應。有些變更只純粹發生在我方坦克上，例如能夠連發子彈、超強子彈爆炸威力、兩倍行走速度的改變，只要更動 *TMyTank* 本身變數即可；但有些寶物的能力，例如暫停敵方坦克動作、為軍旗加上防護罩等等，必須依賴遊戲系統的支援才能達成。此處你可以先看到我以呼叫 *PostMessage* API 函式將自訂的視窗訊息丟到目前執行緒的訊息佇列中，這些自訂訊息會由誰來處理，為我們達成特殊效果的請求呢？先賣個關子，後頭再敘。

敵方坦克

所有敵方坦克類別的父類別—*TETank*，也和 *TTank* 一樣都是抽象類別，而所有敵方坦克類別皆繼承它。*TETank* 類別最大的貢獻是，出現時產生金光閃閃效果。

金光閃閃效果是由另一個直接衍生自 *TSprite* 的 *TStar* 類別所負責，*TETank* 負責產生及摧毀 *TStar* 物件，並在適當時候（出生前）不繪出自己，而繪出 *TStar* 物件，即可達成先出現金光閃閃而後敵方坦克才出現的效果。*TStar* 的建構函式如下，它的角色圖形如圖 9-16。

```
#0001 // 敵方坦克出生時金光閃閃的效果類別
#0002 TStar::TStar()
#0003 {
#0004     Speed = 0; // 不動
#0005     Attr = TSpriteAttr() << saUndirectionalBitmap; // 沒有方向性
#0006     FMoveDelay = 8; // 每八個 frame 才變一次
#0007     FAdvanceFrameMode = afStop; // 動畫播完後就停止
#0008
#0009     FFileName = "star.bmp";
#0010     FFrameMax = 2; // 三張動畫
#0011     FObjectRect = Classes::Rect(0, 0, 38, 32);
#0012 }
```



圖 9-16 TStar

由此可知，*TStar* 類別即是個不行動、圖形無方向性、緩慢播放動畫，且三格動畫播完即功成身退的角色。而 *TETank* 敵方坦克以如下方式使用 *TStar* 來達成出生效果的目的：

```
#0001 // 敵方坦克的行動函式
#0002 void TETank::Move()
#0003 {
#0004     if (FStar) { // 還在出生中..
#0005         FStar->Move(); // 金光閃閃物件進行下一動
#0006
#0007         if (!FStar->Active) { // 若金光閃閃動畫秀完了..
#0008             delete FStar; // 摧毀金光閃閃物件，下一動敵方坦克就會出現了..
#0009             FStar = NULL;
#0010         }
#0011     } else {
#0012         // 依亂數"可能"發射子彈
#0013         if (random(100) < PROBAB_ETANK_SHOT_BULLET * 100)
#0014             FireBullet();
#0015
#0016         // 若連續碰撞次數過多，或亂數許可，則轉向
#0017         if (CollisionCount > MAX_ETANK_COLLISION_COUNT ||
#0018             random(100) < PROBAB_ETANK_RANDOM_TURN * 100)
#0019             RandomDirection();
#0020
#0021         TTank::Move(); // 進行下一動
#0022     }
#0023 }
#0024
#0025 void TETank::Draw(TCanvas* Canvas)
#0026 {
#0027     if (!FStar) // 金光閃閃物件是否存在？
#0028         TTank::Draw(Canvas); // 正常的繪製動作
#0029     else
#0030         FStar->Draw(Canvas); // 畫出金光閃閃的星星
#0031 }
```

可以看出，*TETank* 類別除了提供金光閃閃出生效果外，還以亂數控制，主動發射子彈及轉向，當然囉，若連續碰撞次數過多也會轉向。

既然 *TETank* 已具有所有敵方坦克該具備的功能及特色，五種不同的敵方坦克只要在建立 *TETank* 物件及進行某些特定動作時時，按照各自的特徵修改對應的變數即可。哦，還有設定個別的 *FScore* 變數，代表其被主角摧毀後可獲得的分數。

- 第一種敵方坦克
一切正常，改都沒改。
- 第二種敵方坦克
將 *FSpeed* 設為 *SPRITE_DEFAULT_SPEED * 2*，足足比別的敵方坦克快一倍。
- 第三種敵方坦克
在 *FireBullet* 函式中，將子彈的速度設定為正常速度的兩倍。
- 第四種敵方坦克
將 *FSpeed* 設為 *SPRITE_DEFAULT_SPEED - 1*，*FHP* 設為 4，跑得慢一點，但打四次才會死。
- 第五種敵方坦克
嘻，這是我自行新增的「飛行」坦克，狀似蝴蝶。將 *FOnAir* 設為 *true*，表示在空中飛行；將 *FSpeed* 設為 *SPRITE_DEFAULT_SPEED / 2*，只有一般敵方坦克的一半速度，但皮實在厚，*FHP* 為 10。

子彈及爆炸

坦克之外，坦克所發射出的子彈也是戰場上最耀眼的明星之一。它的實作重心在於碰撞處理，不論碰到邊界、地形物、坦克甚至其它子彈時，都要視情況啟動爆炸效果及產生其它邊際效應，如設定地形物破碎表格、遞減坦克生命力、摧毀其它子彈等等。

子彈類別 *TBullet* 的類別宣告如下（定義於 *Sprite.h*）：

```
#0001 class TBullet : public TSprite {  
#0002 private:  
#0003     TTank* FTank; // 產生此子彈的坦克  
#0004     TExplosion* FExplosion; // 此子彈所產生的爆炸物件  
#0005     bool FTankBulletNum_Dropped; // 是否已將坦克的子彈數目遞減  
#0006
```

```

#0007 // 啓動爆炸效果 (建立爆炸物件)
#0008 void FireExplosion(const std::type_info& typeinfo);
#0009 void LocateExplosion(TRect R); // 將爆炸置於矩形區域中心
#0010 protected:
#0011 // 改寫碰撞檢查觸發函式
#0012 virtual bool CheckCollisions(int& x, int& y);
#0013
#0014 // 改寫所有的碰撞處理函式, 同時新增與子彈的碰撞處理函式
#0015 virtual bool CheckBoundCollisions(int& x, int& y);
#0016 virtual bool CheckCellCollisions(int Layer, int& x, int& y,
#0017     TCellArray& Collisions);
#0018 virtual bool CheckTankCollisions(int& x, int& y,
#0019     TSpriteArray& Collisions);
#0020 virtual bool CheckBulletCollisions(int& x, int& y,
#0021     TSpriteArray& Collisions);
#0022 public:
#0023 TBullet(TTank* ATank);
#0024 virtual ~TBullet();
#0025
#0026 virtual void Draw(TCanvas* Canvas);
#0027 virtual void Move();
#0028
#0029 __property TTank* Tank = {read = FTank, write = FTank};
#0030 };

```

子彈所引發的爆炸效果由 *TExplosion* 類別提供, 它的功用如同 *TStar* 類別, 也只是擺在那邊好看, 慢速播放效果動畫的物件。而 *TExplosion* 又是個抽象類別, 真正的爆炸由它的兩個子類別 *TSmallExplosion* 及 *TBigExplosion* 提供。這兩個類別唯一的不同就是圖形大小及動畫數目罷了, 分別使用不同的角色圖形, 如圖 9-17。

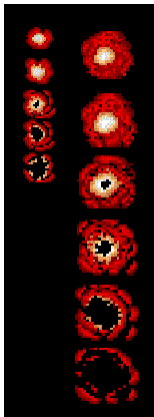


圖 9-17 / 大小爆炸所用的圖形, 分別有六張及五張動畫

而 *TBullet* 使用 *TExplosion* 物件的方式也和 *TETank* 使用 *TStar* 物件的方式無二，不再重覆說明。

TBullet 的建構及解構函式中，會將本身加入 *Bullets* 全域陣列及將本身自陣列中移除。這也就是為什麼在 *TTank::FireBullet* 函式中，產生子彈物件後，什麼記錄都不必留，也不怕遺失對於子彈物件的參考，因為子彈會負責自己的登錄工作。

TBullet 類別的重頭戲在於個碰撞處理函式的改寫：

- 改寫 *CheckBoundCollisions* 函式
撞上任一邊界時，引發小爆炸。
- 改寫 *CheckCellCollisions* 函式
去除帶有 *taBulletCanPass* 屬性的圖格，引發小爆炸，並針對每個碰撞到的圖格計算破碎程度，更新圖格的破碎表格。若炸到的是帶有 *taFlag* 屬性的圖格，表示此圖格放置軍旗，就發個訊息告知遊戲結束。
- 新增 *CheckTankCollisions* 函式
若撞到的坦克是發射子彈本身的坦克當然沒事（不過坦克要被自己發射的子彈 **K** 中倒也不容易）；若撞到的坦克及主人皆屬敵方坦克，則也沒事，讓子彈繼續飛行；否則的話，表示主角子彈打到敵方坦克或敵方子彈打到主角，先製造一場大爆炸，同時遞減被炸到坦克的生命力，若生命力等於零，則摧毀該坦克。
- 新增 *CheckBulletCollisions* 函式
將子彈與畫面上所有其它子彈皆進行碰撞測試。若子彈主人互為敵我，則兩子彈抵消掉，將自己及對方子彈皆申報作廢；否則，當作沒事，子彈繼續飛行。

好囉，重點在於瞭解類別及函式的任務分派，所以只有簡述，沒有程式碼，本文的重點在於如何架構一個完整的遊戲程式，而非撰寫細密繁雜的程式碼。就這樣，快刀亂麻、秋風落葉般地将角色子系統完成。好累吧～該是將它們全部兜在一塊的時刻了。

遊戲的誕生

下圖是遊戲主視窗的設計時期畫面，哇，夠簡單的了。比足球賽程式用的元件還少，不過別擔心，因為這是遊戲寫作，而不是資料庫程式設計課程，遊戲畫面與元件的使用無關，只要給我一個視窗，一張畫布，任是什麼樣的遊戲畫面也生得出來。

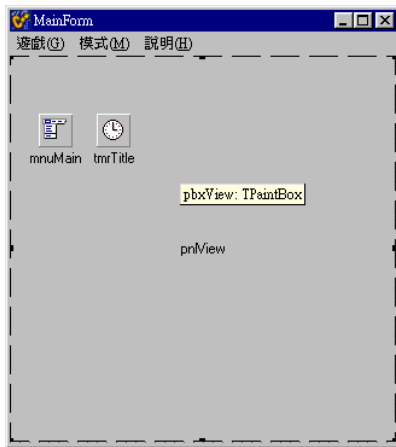


圖 9-18 / 遊戲主視窗的設計時期畫面

有了先前設計的兩大子系統為靠山，遊戲主程式的撰寫真的可以「用兜的」，不必再煩惱什麼碰撞處理，也不必在意圖片、圖層及圖格之間曖昧不明的三角關係。身處直接面對遊戲玩家的第一線上，遊戲主程式只要負責將**畫面**、**使用者輸入**及**遊戲邏輯**三部分打點好，其餘的交給背後的地圖及角色子系統去操心吧。

繪製遊戲畫面

無可避免地，由於速度上的考量，我們不直接在畫布上進行貼圖動作，而先暗地裡在另一個相同大小的 `bitmap` 上作畫，等到全部繪製完成後再一口氣複製到視窗上。

`DrawBackBitmap` 做的就是這些動作，首先依序畫出地形層、地形物層、物品層、寶物、

地面坦克、子彈、高地形物層及空中坦克。接著才根據目前的遊戲狀態繪出玩家坦克生命、得分等等。*DrawBackBitmap* 函式如下：

```
#0001 // 先將畫面繪製在緩衝用 bitmap, 再複製到視窗上
#0002 void __fastcall TMainForm::DrawBackBitmap()
#0003 {
#0004     TMap& Map = TMap::Instance();
#0005
#0006     if (mnuDrawLayer1->Checked)
#0007         Map.Draw(FBackBitmap->Canvas, LAYER_TERR); // 繪製地形層
#0008     else { // 若沒畫地形層, 就塗黑
#0009         FBackBitmap->Canvas->Brush->Color = clBlack;
#0010         FBackBitmap->Canvas->FillRect(FBackBitmap->Canvas->ClipRect);
#0011     }
#0012
#0013     if (mnuDrawLayer2->Checked) // 繪製地形物層
#0014         Map.Draw(FBackBitmap->Canvas, LAYER_TERRITEM);
#0015
#0016     if (mnuDrawLayer3->Checked) // 繪製物品層
#0017         Map.Draw(FBackBitmap->Canvas, LAYER_ITEM);
#0018
#0019     // 若有寶物, 畫出寶物
#0020     if (Gem) Gem->Draw(FBackBitmap->Canvas);
#0021
#0022     DrawTanks(FBackBitmap->Canvas, false); // 畫出地面上所有坦克
#0023
#0024     DrawBullets(FBackBitmap->Canvas); // 畫出子彈
#0025
#0026     if (mnuDrawLayer4->Checked) // 繪製高地形物層
#0027         Map.Draw(FBackBitmap->Canvas, LAYER_HITERRITEM);
#0028
#0029     // 畫出"空中"所有坦克
#0030     DrawTanks(FBackBitmap->Canvas, true);
#0031
#0032     FBackBitmap->Canvas->Font->Color = clWhite;
#0033     FBackBitmap->Canvas->Font->Name = "FixedSys";
#0034     FBackBitmap->Canvas->Font->Style = TFontStyles() << fsBold;
#0035     FBackBitmap->Canvas->Font->Size = 14;
#0036     FBackBitmap->Canvas->Brush->Style = bsClear;
#0037
#0038     TRect R;
#0039     switch (FGameStatus) {
#0040         case gsTitle:
#0041             // 畫出上面的標題大字及下方的作者名稱
#0042             R = Rect(0, 0, TILE_WIDTH * TILE_NUM_X, TILE_HEIGHT *
#0043                 TILE_NUM_Y - FBackBitmap->Canvas->TextHeight("我") / 2);
```

```

#0044     DrawText(FBackBitmap->Canvas->Handle, "作者: 陳寬達", -1, &R,
#0045         DT_BOTTOM | DT_CENTER | DT_SINGLELINE);
#0046     DrawStatusBox("歡迎光臨 坦克大決戰", true);
#0047     break;
#0048
#0049     case gsOver: // Ouch, 軍旗被幹掉或主角死掉了
#0050         DrawStatusBox("任務失敗", false);
#0051         break;
#0052
#0053     default:
#0054         // 在右上角顯示生命力及分數
#0055         AnsiString Str = Format("裝甲 %.2d 得分 %.4d",
#0056             OPENARRAY(TVarRec, (FTank->HP, FScore)));
#0057         FBackBitmap->Canvas->TextOut(WORLD_WIDTH -
#0058             FBackBitmap->Canvas->TextWidth(Str) - 5, 5, Str);
#0059
#0060         // 在左上角顯示關卡
#0061         Str = Format("LEVEL %d", OPENARRAY(TVarRec, (FLevelNo)));
#0062         FBackBitmap->Canvas->TextOut(5, 5, Str);
#0063
#0064         break;
#0065     }
#0066
#0067     // 這是過關畫面
#0068     if (FGameStatus == gsSuccess)
#0069         DrawStatusBox("任務成功 !!", false);
#0070 }

```

程式共分為五種狀態，分別為：

```

enum TGameStatus {gsTitle,      // 歡迎畫面
                  gsPlaying,    // 遊戲進行中
                  gsSuccess,    // 過關畫面
                  gsOver,       // GAME OVER
                  gsTerminate}; // 程式即將關閉

```

我希望能於歡迎畫面時，一邊秀出第一關的地圖，一邊有一群坦克們在裡頭忙碌地移動及破壞地形物，所以進入 *gsTitle* 狀態時，隨機產生 *MAX_TANK_ON_SCREEN* 輛敵方坦克，並將我方坦克藏起來，免得使用者與遊戲中畫面搞混，煞有其事地拿起搖桿來砰砰。另一方面利用 *tmrTitle* 計時器，每 20 毫秒進行一動，便可模擬遊戲進行時敵方坦克的行為。

設定 *GameStatus* 屬性時，會呼叫到它的屬性存取函式 *SetGameStatus*：

```
#0001 void __fastcall TMainForm::SetGameStatus(TGameStatus Value)
#0002 {
#0003     FGameStatus = Value;
#0004
#0005     // 根據新的遊戲狀態開關計時器
#0006     tmrTitle->Enabled = FGameStatus == gsTitle;
#0007
#0008     switch (FGameStatus) {
#0009         case gsTitle:
#0010             LevelNo = 1; // 歡迎畫面顯示第一關地圖
#0011
#0012             // 隨機產生 MAX_TANK_ON_SCREEN 輛敵方坦克
#0013             for (int i = 1; i <= MAX_TANK_ON_SCREEN; i++)
#0014                 CreateETank(random(5) + 1);
#0015
#0016             FTank->Visible = false; // 將自己坦克藏起來
#0017             FTank->Active = false; // 自己坦克不要動
#0018             break;
#0019
#0020         case gsPlaying:
#0021             UpdateControlStatus(); // 更新標題列及其它控制項
#0022             UpdateView(); // 更新遊戲畫面
#0023
#0024             GameLoop(); // 進入遊戲主迴圈 (遊戲進行中都一直在此迴圈內)
#0025             break;
#0026
#0027         case gsSuccess:
#0028             break;
#0029     }
#0030
#0031     if (FGameStatus != gsTerminate) { // 使用者是否欲關閉視窗 ??
#0032         UpdateControlStatus(); // 更新標題列及其它控制項
#0033         UpdateView(); // 更新遊戲畫面
#0034     } else
#0035         Close(); // 關閉視窗
#0036 }
```

0024 列呼叫 *GameLoop* 函式，這個函式可說是集天下之大成，整個遊戲進行中都會一直處於這個函式內執行。爲什麼要這麼麻煩？怎麼不像上回足球番程式那樣，使用者按一下，行動一步，畫面重繪一次，不是很簡單嗎？

差別在於：足球番是非即時遊戲，而坦克大決戰是即時遊戲。所謂即時遊戲意指遊戲不

論是否有玩者輸入事件（按鍵或扳動搖桿等等），都會持續不間斷地進行份內該做的事。例如，射出子彈後，不論玩者跑去上廁所或不斷地敲擊空白鍵，子彈還是得繼續飛，敵方坦克也得繼續忙碌著亂跑。

你可能會想到使用計時器，就像我們的歡迎畫面那樣，每隔很短的一段時間就觸發一次，讓所有角色進行一動，再更新畫面。

聽來很合理，不過現實總是不同。重點是 Windows 裡的計時器功能並不值得依賴，這組計時器功能依賴 *WM_TIMER* 視窗訊息來觸發事件或呼叫回呼函式，而 *WM_TIMER* 視窗訊息又是幾百個標準 Windows 視窗訊息裡優先權最低的幾個，只要有其它事發生，*WM_TIMER* 一定會遲到甚至不到，這樣沒有時間觀念的傢伙你還敢讓它擔任維持遊戲時程的重責大任嗎？（請參閱第四章「分秒必爭，細說計時器」，對計時器機制有十分詳盡的解說。）

因此我們通常選擇取得完全主控權的方法，換句話說，讓遊戲進行中，不論有無任何事件發生，執行的仍是我們為它準備好的程式碼。靠著 *application framework* 的幫忙，平日撰寫應用程式時，我們絲毫不必知道當程式處於閒置狀態時，它在做什麼？反正只要有人傳遞訊息過來，將感興趣的訊息攔截下來，做出適當回應就好了，其它就不用管了。

說得具體一點，VCL的*TApplication*類別不但背負整個應用程式的生滅，也包括訊息的處理。Win32 架構中，一個執行緒只要擁有視窗，就同時擁有一個訊息佇列，此執行緒必須常常去詢問取得佇列中的視窗訊息，並分派給訊息的目的視窗的視窗函式。而負責取得分派視窗訊息的迴圈就稱為訊息迴圈，對於擁有視窗的執行緒而言，終其一生皆在訊息迴圈內度過。打開專案原始碼，看到那行熟悉的*Application->Run*呼叫了沒？裏頭包含的正是主執行緒的訊息迴圈⁵。

說得再具體一點，我們必須另開新局，自行建立一個訊息迴圈，在遊戲進行中，暫時將

⁵ 三言兩語要道盡訊息、訊息佇列、訊息迴圈及VCL的訊息處理流程是不可能的任務，若希望能有徹底認識，請閱讀錢達智先生的Delphi學習筆記Win32 基礎篇。

TApplication 類別的任務接下，處理即時遊戲中所有的行進、重繪及訊息處理等工作。

遊戲主迴圈

GameLoop 函式中，有一個 *while* 迴圈，我稱之為「遊戲主迴圈」，遊戲開始後、結束前，執行緒會一直處於此 *while* 迴圈內執行，不會離開。事實上，它就是接替 *TApplication* 工作的新訊息迴圈，負責視窗訊息的分派處理。不過不只這些，它還有額外的任務：

- 若訊息佇列內尚有訊息，則取出訊息來處理。
- 若訊息佇列內沒有訊息，且遊戲視窗保有輸入焦點，則讓所有角色進行一動，並且重繪畫面。
- 若遊戲視窗未取得輸入焦點，則呼叫 *WaitMessage* API 函式將控制權轉讓系統中其它執行緒（反正沒事幹），直到有新訊息進來為止。

TMainForm::GameLoop 函式程式碼列表如下：

```
#0001 void __fastcall TMainForm::GameLoop()
#0002 {
#0003     InitLevel(); // 初始化
#0004
#0005     TMsg Msg;
#0006     int iStopTime;
#0007     TSprite* Sprite;
#0008     TMap& Map = TMap::Instance();
#0009
#0010     // 遊戲迴圈
#0011     while (FGameStatus == gsPlaying) {
#0012         if (PeekMessage(&Msg, 0, 0, 0, PM_REMOVE)) { // 取得訊息
#0013             switch (Msg.message) {
#0014                 case WM_QUIT: // 程式結束，離開遊戲迴圈
#0015                     FGameStatus = gsTerminate;
#0016                     goto OutGameLoop;
#0017
#0018                 case WM_DESTROY_OBJECT: // 要求釋放某物件
#0019                     Sprite = (TSprite*)Msg.wParam;
#0020
#0021                 if (Sprite == FTank) { // 若死掉的是主角
#0022                     FTank->Visible = false;
```

```
#0023         FGameStatus = gsOver; // Game over 囉 ~~
#0024         goto OutGameLoop; // 離開遊戲迴圈
#0025     }
#0026
#0027         // 坦克摧毀的話要做特別處理
#0028     if (dynamic_cast<TTank*>(Sprite)) {
#0029
#0030         // 若被摧毀的是敵方坦克，則 Msg.LPARAM 代表其分數
#0031         FScore += Msg.lParam;
#0032
#0033         // 將坦克與與其有關係的子彈解除關係
#0034         FreeBulletsForTank(dynamic_cast<TTank*>(Sprite));
#0035
#0036         // 若敵方坦克全部出現且死光光了，則到下一關去
#0037         if (FTankUsed == MAX_ETANK_PER_SCENARIO &&
#0038             Tanks()->size() == 1) {
#0039             // 這裡應該加一些過場畫面...
#0040             LevelNo = LevelNo + 1;
#0041             InitLevel();
#0042         }
#0043     }
#0044
#0045         // 釋放物件
#0046     delete Sprite;
#0047
#0048     break;
#0049
#0050     case WM_GAMEOVER: // 軍旗被打掉了，遊戲結束
#0051         FGameStatus = gsOver;
#0052         goto OutGameLoop; // 離開遊戲迴圈
#0053
#0054     case WM_SPECIAL_CONDITION: // 進入特殊狀態
#0055         iStopTime = 10; // 特殊狀態預設有效時間十秒鐘
#0056
#0057     switch (Msg.wParam) {
#0058         // 暫停敵人行動
#0059         case TIMER_ID_GEM_CLOCK: FEnemyStopped = true;
#0060
#0061         break;
#0062
#0063         // 無敵
#0064         case TIMER_ID_GEM_HAT: FTank->SuperMode = true;
#0065         break;
#0066
#0067         case TIMER_ID_GEM_ARROW:
#0068             // 在軍旗周圍擺上打不破的鐵牆，且修復它
```

```

#0069         Map.GetCell(LAYER_TERRITEM, 5, 11).TileNo = 26;
#0070         Map.GetCell(LAYER_TERRITEM, 5, 11).DisposeBreakMap();
#0071         Map.GetCell(LAYER_TERRITEM, 6, 11).TileNo = 26;
#0072         Map.GetCell(LAYER_TERRITEM, 6, 11).DisposeBreakMap();
#0073         Map.GetCell(LAYER_TERRITEM, 7, 11).TileNo = 26;
#0074         Map.GetCell(LAYER_TERRITEM, 7, 11).DisposeBreakMap();
#0075         Map.GetCell(LAYER_TERRITEM, 5, 12).TileNo = 26;
#0076         Map.GetCell(LAYER_TERRITEM, 5, 12).DisposeBreakMap();
#0077         Map.GetCell(LAYER_TERRITEM, 7, 12).TileNo = 26;
#0078         Map.GetCell(LAYER_TERRITEM, 7, 12).DisposeBreakMap();
#0079
#0080         iStopTime = 20; // 鐵牆持續二十秒
#0081         break;
#0082     }
#0083     // 設定計時器
#0084     SetTimer(Handle, Msg.wParam, iStopTime * 1000, NULL);
#0085     break;
#0086
#0087     case WM_TIMER: // 計時器時間到 (特殊狀態時間到)
#0088         KillTimer(Handle, Msg.wParam); // 取消計時器
#0089
#0090     switch (Msg.wParam) {
#0091         case TIMER_ID_GEM: // 寶物擺夠久了, 還不吃, 拿掉
#0092             if (Gem) delete Gem;
#0093             break;
#0094
#0095         case TIMER_ID_GEM_CLOCK: FEnemyStopped = false;
#0096             break;
#0097
#0098         case TIMER_ID_GEM_HAT:
#0099             FTank->SuperMode = mnuSuperMode->Checked;
#0100             break;
#0101
#0102         case TIMER_ID_GEM_ARROW: // 在軍旗周圍擺回磚牆
#0103             Map.GetCell(LAYER_TERRITEM, 5, 11).TileNo = 10;
#0104             Map.GetCell(LAYER_TERRITEM, 6, 11).TileNo = 10;
#0105             Map.GetCell(LAYER_TERRITEM, 7, 11).TileNo = 10;
#0106             Map.GetCell(LAYER_TERRITEM, 5, 12).TileNo = 10;
#0107             Map.GetCell(LAYER_TERRITEM, 7, 12).TileNo = 10;
#0108             break;
#0109     }
#0110     break;
#0111
#0112     case WM_INIT_LEVEL:
#0113         InitLevel(); // 關卡重新開始
#0114     break;

```



```

#0115     }
#0116
#0117     // 正常的訊息處理程序
#0118     TranslateMessage(&Msg);
#0119     DispatchMessage(&Msg);
#0120 } else if (Focused()) { // 若視窗擁有輸入焦點才動作
#0121     // 若此關卡及目前敵方坦克都沒達到上限,
#0122     // 則按照亂數"可能"出現敵方坦克
#0123
#0124     if (FTankUsed < MAX_ETANK_PER_SCENARIO &&
#0125         (int)Tanks()->size() < MAX_TANK_ON_SCREEN &&
#0126         random(100) < 100 * PROBAB_ETANK_BORN)
#0127         CreateETank(random(5) + 1); // 五種坦克任選一種
#0128
#0129     // 若目前沒有寶物, 則按照亂數"可能"出現寶物
#0130     if (!Gem && random(100) < 100 * PROBAB_GEM_BORN)
#0131         CreateGem(random(6)); // 六種寶物任選一種
#0132
#0133     // 移動我方坦克及/或敵方坦克
#0134     if (FEnemyStopped)
#0135         FTank->Move();
#0136     else
#0137         MoveTanks();
#0138
#0139     // 移動子彈
#0140     MoveBullets();
#0141
#0142     // 更新畫面
#0143     UpdateView();
#0144 } else {
#0145     // 若視窗沒有取得輸入焦點, 則將控制權交給其它執行緒,
#0146     // 直到有訊息進來
#0147     WaitMessage();
#0148 }
#0149 }
#0150
#0151 OutGameLoop: ;
#0152 }

```

0012 列呼叫的 *PeekMessage* API 函式, 配合 *PM_REMOVE* 參數, 效果與平日常用的 *GetMessage* API 函式幾乎一樣。唯一的差別是, *PeekMessage* 的傳回值為布林值, 表示是否取得訊息, 若沒有訊息等待, 函式立即返回; 而 *GetMessage* 的傳回值代表取得的訊息是否為 *WM_QUIT*, 函式會等待取得訊息後才返回。這就是遊戲迴圈中, 一邊能讓程式正常運作, 一面能保持遊戲即時性的關鍵: 「不斷呼叫 *PeekMessage* 函式, 若有訊息,

則處理訊息；若沒訊息，則進行遊戲狀態的更新」。

不過，當遊戲視窗沒有取得輸入焦點時，遊戲狀態就不再需要不斷更新。一方面玩家可能只是想切換到 BBS 連線程式回個熱訊，沒想到切換回遊戲視窗時，才發現主角坦克已經被轟爛了，人家打到三十關了誰賠他呀？所以必須防止此情況的發生。另一個理由是，不讓遊戲繼續佔用大量 CPU 時間，遊戲狀態不斷更新會耗掉大量 CPU 時間，所以必須在視窗未取得輸入焦點時暫停動作。因此 0120 列會先呼叫 *Focused*，得知目前是否擁有輸入焦點，若有的話，更新遊戲狀態；沒有的話，則呼叫 *WaitMessage* API 函式。此函式會將控制權交給其它執行緒使用，若此次執行周期尚未結束前又有新的視窗訊息進入，控制權會自動再交還我們。

視窗訊息的處理

0013 ~ 0115 列是視窗訊息的處理邏輯，一一處理感興趣的訊息：

WM_QUIT 訊息

將 *FGameStatus* 設為 *gsTerminate*，程式將跳離遊戲迴圈，並於離開 *GameLoop* 函式後立即結束程式。

WM_DESTROY_OBJECT 訊息

這是由 *TSprite* 物件的 *PostToDie* 函式所丟出的訊息，表示此角色「申請」自毀。訊息裡的 *wParam* 參數事實上是指向申請的物件本身的指標，*lParam* 參數則記錄玩者所得的分數。若 *wParam* 參數指向主角坦克 *FTank*，表示主角被幹掉了，則將 *FGameStatus* 設為 *gsOver*，離開遊戲迴圈。

在此將 *wParam* 參數轉型為 *TSprite*，以 *delete* 保留字摧毀此物件。由於 *wParam* 參數必定指向遊戲中的某個 *TSprite* 物件，在多型機制的輔助下，物件摧毀時一定會呼叫到正確的物件解構函式。

最後檢查是否所有該出現的敵方坦克已全部被摧毀，若是的話，遞增關卡編號，並將遊

戲狀態初始化，直接進入下一關繼續遊戲。理論上，關卡與關卡之間會有一些過場畫面，如上一關的得分、下一關的任務說明等等，請原諒我的偷懶。

WM_GAMEOVER 訊息

表示軍旗被打爛了，將 *FGameStatus* 設為 *gsOver*，離開遊戲迴圈。

WM_SPECIAL_CONDITION 訊息

這是我方坦克吃到寶物時對於特殊效果的請求，由 *TMyTank::CheckCollisions* 函式發出。訊息裡的 *wParam* 參數為特殊效果代碼，遊戲迴圈必須根據代碼提供正確的服務：

- *TIMER_ID_GEM_CLOCK* 暫停敵人行動
- *TIMER_ID_GEM_HAT* 將我方坦克加上防護罩，變成無敵狀態
- *TIMER_ID_GEM_ARROW* 在軍旗周圍擺上打不破的鐵牆

最後的步驟是設定計時器，讓特殊效果只持續一段時間後即恢復。

WM_TIMER 訊息

當我們設定的計時器時間到時便會收到此訊息，此時再根據計時器代碼來進行適當的動作，我們也利用它來進行特殊效果的還原：

- *TIMER_ID_GEM* 表示寶物擺夠久了，主角還不吃，拿掉
- *TIMER_ID_GEM_CLOCK* 恢復敵人行動
- *TIMER_ID_GEM_HAT* 將我方坦克恢復原來非無敵狀態
- *TIMER_ID_GEM_ARROW* 在軍旗周圍擺回磚牆

WM_INITLEVEL 訊息

將遊戲狀態重設為目前關卡的初始狀態。

最後，對於其它尚未處理的訊息，一律採用最標準的訊息處理方式，呼叫 *TranslateMessage* API 函式剖析鍵盤訊息以及呼叫 *DispatchMessage* API 函式將訊息分派給適當的視窗程序。

例行的遊戲狀態更新

但如果 *PeekMessage* 未取得任何視窗訊息呢？0121 ~ 0143 列就進行例行的遊戲狀態更新動作：

1. 若此關卡及目前敵方坦克都沒達到上限，則按照亂數「可能」出現敵方坦克。
2. 若目前沒有寶物，則按照亂數「可能」出現寶物。
3. 移動我方坦克及／或敵方坦克。
4. 移動子彈。
5. 更新畫面。

如此一來，遊戲就會即時不斷地更新狀態，你可以看到敵方坦克不停地走動，子彈不斷地發射，不停地飛行、碰撞、爆炸，一幕幕活生生的坦克大戰景象不停地在遊戲視窗中上演著。

上面程式所呼叫的 *CreateETank* 及 *CreateGem* 函式分別依參數產生對應的敵方坦克及寶物。建立寶物時，必須同時呼叫 *SetTimer* API 函式來設定計時器，以使寶物能自動在三十秒後消失。兩個函式的程式碼列表如下：

```
#0001 // 產生敵方坦克
#0002 void __fastcall TMainForm::CreateETank(int Kind)
#0003 {
#0004     FETankUsed++; // 遞增此關卡已產生的敵方坦克
#0005
#0006     TETank* T;
#0007     T = new TETank(Kind);
#0008     T->LoadBits();
#0009 }
#0010
#0011 // 產生寶物
#0012 void __fastcall TMainForm::CreateGem(int Kind)
#0013 {
#0014     Gem = new TGem(TGemKind(Kind));
#0015     Gem->LoadBits();
#0016     Gem->RandomPosition(); // 隨意擺置
```

```
#0017 // 寶物出現 30 秒後自動消失
#0018 SetTimer(Handle, TIMER_ID_GEM, 30 * 1000, NULL);
#0019 }
```

處理使用者輸入

最後的最後，讓我們來畫龍點睛，加上使用者控制部分。先將 *TMainForm* 的 *KeyPreview* 屬性設為 *true*，讓它無時無刻都能優先收到鍵盤訊息。再分別撰寫它的 *OnKeyDown* 及 *OnKeyUp* 事件處理函式：

```
#0001 void __fastcall TMainForm::FormKeyDown(TObject *Sender, WORD &Key,
#0002     TShiftState Shift)
#0003 {
#0004     // 注意，只有遊戲中狀態，鍵盤控制才有效
#0005     if (FGameStatus == gsPlaying) {
#0006         switch (Key) {
#0007             case VK_UP: case VK_DOWN: case VK_LEFT: case VK_RIGHT:
#0008                 switch (Key) {
#0009                     case VK_UP: FTank->Direction = drUp; break; // 向上走
#0010
#0011                     case VK_DOWN: FTank->Direction = drDown; break; // 向下走
#0012
#0013                     case VK_LEFT: FTank->Direction = drLeft; break; // 向左走
#0014
#0015                     case VK_RIGHT: FTank->Direction = drRight; break; // 向右走
#0016
#0017                 }
#0018
#0019                 FTank->Active = true; // 主角開始"動"
#0020                 break;
#0021
#0022             case VK_SPACE:
#0023                 FTank->FireBullet(); // 發射子彈，咻 ~~
#0024                 break;
#0025             }
#0026         }
#0027     }
#0028
#0029 void __fastcall TMainForm::FormKeyUp(TObject *Sender, WORD &Key,
#0030     TShiftState Shift)
#0031 {
#0032     // 注意，只有遊戲中狀態，鍵盤控制才有效
#0033     if (FGameStatus == gsPlaying)
```

```
#0034     switch (Key) { // 放開鍵盤，我方坦克就停止動作
#0035         case VK_UP: case VK_DOWN: case VK_LEFT: case VK_RIGHT:
#0036             FTank->Active = false;
#0037         break;
#0038     }
#0039 }
```

與足球番的使用者輸入最大的不同是，這兒希望做到讓我方坦克連續行走的效果。也就是，若按下任一方向鍵，坦克開始移動後，只要在該鍵放開前，坦克都會不斷地移動，即使另外按下空白鍵發射子彈也一樣。

因此我們不再採用，每收到 *WM_KEYDOWN* 視窗訊息，就更新主角座標一次的方式；而是收到 *WM_KEYDOWN* 視窗訊息時，讓 *FTank->Moving* 為 *true*，而收到 *WM_KEYUP* 時，讓 *FTank->Moving* 為 *false*，讓遊戲主迴圈時更新坦克的位置，如此便可達成上述效果。

熬呀熬出頭

終於，好不容易，千辛萬苦，披荆斬棘，排除萬難，我們來到最令人興奮的一刻！儲存整個專案，按下【F9】執行鍵，讓下面幾張圖來道感言吧。



圖 9-19 / 歡迎畫面



圖 9-20 / 第一關遊戲畫面



圖 9-21 / 第二關遊戲畫面



圖 9-22 / 軍旗被烤焦，遊戲結束畫面

第一關遊戲畫面中，你可以看到我新增的第五種坦克——一隻狀似蝴蝶的飛行器在天上慢慢移動，不斷轟炸地面的可怕景象。第二關遊戲畫面中，兩隻可愛的草怪用磚牆圍住，一輛敵方坦克正穿過樹棚自裡頭跑出來，而主角的子彈正越過海洋快要偷襲到它呢。

而任務失敗的畫面中，我方坦克跑到畫面中央想將剛出現的大蝴蝶轟掉，太興奮了，沒注意到下方什麼時候偷跑來兩輛敵方坦克，狠心將小鳥軍旗烤焦，唉。

雖然速度慢了點，聲音單調了點（由於聲道的限制，目前只播放大爆炸音效），平衡度差了點（所有的坦克參數皆未經過精心考慮，隨手設定的結果），不過還是很好玩，好像重回任天堂版坦克大決戰時代。

坦克大決戰實作，成功！

第五篇

軟體開發



第十章

Fancy 軟體撰寫手則

在 RAD 開發工具的火力支援下，花俏的程式隨處可見。

但是，功能強大與使用者界面的好壞是兩回事，
本章將與你分享從使用者層面出發的細節及技術。



軟體人人會寫，尤其在 RAD 的火力支援下，花俏的程式隨處可見，一個稍有經驗的程式設計師往往可在短短的時間內架構出程式的外框原型。但長久與冷冰冰的電腦硬體、開發工具及漫無止境的程式碼為伍，程式設計師的思考模式似乎與一般使用者截然不同，別說日夜顛倒的作息、街頭浪人般的行頭、沒事大呼小叫直抓頭髮的習慣，就連最基本的軟體操作觀點也與使用者大異其趣。

舉個例說，技術部門的工程師們評估軟體時總是以它們的功能是否強大、軟體支援是否完備、使用的技術是否新穎等技術觀點為最大依據，而使用者最需要的往往只是簡潔方便的介面及處處體貼的設計，而這常是程式設計師最容易忽略的部分。辛辛苦苦寫完最富挑戰性的程式核心後，還叫我慢慢小心地調整元件位置大小顏色字型，殺了我吧¹。不曉得各位有沒有用過 IBM Visual Age C++ 3.5 這套開發工具，稍微把玩後你應該就可明瞭，原來有些功力強得不得了的程式設計師是完完全全不適合設計軟體操作介面的。

挾著撰寫多套免費及共享軟體的經驗，我試著將一些從使用者層面出發的細節及技術分享出來，順便介紹解決這些問題必須用到的技巧及觀念，希望有助於你在應用軟體方面的設計及思考。

與系統字型起舞

仔細觀察別人撰寫的應用程式，可發現不論系統設定的字型如何更動，TreeView、ListView 及 StatusBar 等控制項使用的字型皆與系統設定保持一致。從【開始】鈕選擇【設定 / 控制台 / 顯示】，開啓「顯示」對話盒後，再切換到「外觀」頁次，就可以看到「圖示」及「工具提示」兩種字型設定。TreeView 及 ListView 元件通常使用「圖示」的字型，而 StatusBar 通常使用「工具提示」的字型。例如，你可以試著更動這兩個字型設定，再切換為檔案總管視窗，看看檔案總管所使用的字型是否也跟著改變。

¹ 呵呵，我就有這個壞習慣。往往將程式最富挑戰性的部分完成後，即扔到一旁懶得繼續撰寫了。我覺得，有挑戰性的程式設計才好玩，沒挑戰性的程式設計叫苦刑。

TStatusBar::UseSystemFont 屬性

使用這些元件時，若能使字型隨著這些能在控制台設定的使用者自訂字型更動，提供使用者一致的視覺感受，會使程式更具親和力。因此，VCL 的 *TStatusBar* 類別就提供了 *UseSystemFont* 屬性，若將此屬性設為 *true*，當元件建立或系統字型更動時，會自動呼叫 *TStatusBar::SyncToSystemFont* 函式：

```
#0001 procedure TStatusBar.SyncToSystemFont;
#0002 var
#0003   NonClientMetrics: TNonClientMetrics;
#0004 begin
#0005   if FUseSystemFont then
#0006     begin
#0007       NonClientMetrics.cbSize := sizeof(NonClientMetrics);
#0008       if SystemParametersInfo(SPI_GETNONCLIENTMETRICS, 0,
#0009         @NonClientMetrics, 0) then
#0010         Font.Handle :=
#0011           CreateFontIndirect(NonClientMetrics.lfStatusFont)
#0012     end;
#0013 end;
```

0008 列呼叫 *SystemParametersInfo* API 函式並傳入 *SPI_GETNONCLIENTMETRICS* 代碼，可取得目前的視窗外觀設定值。除了狀態列字型外，*TNonClientMetrics* 結構還包含其它四種字型：

```
typedef struct tagNONCLIENTMETRICS
{
    ...
    LOGFONTA lfCaptionFont;           // 視窗標題列
    LOGFONTA lfSmCaptionFont;        // 色板標題
    LOGFONTA lfMenuFont;             // 功能表
    LOGFONTA lfStatusFont;           // 狀態列
    LOGFONTA lfMessageFont;          // 訊息視窗
    ...
} NONCLIENTMETRICS;

typedef tagNONCLIENTMETRICS TNonClientMetrics;
```

0011 列呼叫 *CreateFontIndirect* API 函式，傳入取得的 *TLogFont* 結構，取回該字型的 font

handle，指定給 *Font.Handle* 屬性。因此只要 *UseSystemFont* 屬性為 *true*，*TStatusBar* 元件字型就會永遠與使用者設定的「狀態列」字型一致。

程式如何得知系統字型更動而主動 *SyncToSystemFont* 函式呢？來源是每當有全域性的系統設定更動時，作業系統會對系統中每個視窗進行廣播，送出 *WM_SETTINGCHANGE* 或 *WM_WININICHANGE* 視窗訊息，而 *TStatusBar* 元件會攔截這些視窗訊息，呼叫 *SyncToSystemFont* 函式以維持字型同步。

可憐沒人愛的 TTreeView 及 TListView 元件

可惜的是，*TTreeView* 及 *TListView* 類別並沒有這項機制，缺乏 *UseSystemFont* 這樣方便的屬性供我們使用，所以只好捲起袖子自己動手做了。正要依樣畫葫蘆時，赫然發現，唔，方才取得的 *TNonClientMetrics* 結構雖然包含五種字型設定，卻沒有準備讓 *TreeView* 及 *ListView* 元件使用的「圖示」字型...幾經尋訪，原來它暗藏在另一個結構內：

```
typedef struct tagICONMETRICS {
    ...
    LOGFONTA lFont;
} ICONMETRICS;

typedef tagICONMETRICS TIconMetrics;
```

此 *TIconMetrics* 結構的取得方法與 *TNonClientMetrics* 結構的取得方法相同，只不過在呼叫 *SystemParametersInfo* 函式時，傳入 *SPI_GETICONMETRICS* 代碼即可。有了足夠的資訊問題就簡單多了，參考 *TStatusBar::SyncToSystemFont* 的作法，將配合系統字型設定的功能也加入 *TTreeView* 及 *TListView* 元件中，動手吧。

喂，等等，剛剛傳來線報，有明確來源的消息指出，*VCL* 對於「圖示」字型厚愛有加，使得「圖示」字型可隨時由 *TScreen::IconFont* 屬性取得，所以只需簡單的一行程式碼：

```
TreeView1->Font = Screen->IconFont;
```

就可使 *TreeView1* 元件套用「圖示」字型。ㄟ...請再等等，又有新的消息進來了...

TControl::DesktopFont 屬性

快快追查 *TControl* 類別的 *DesktopFont* 屬性：

```
__property bool DesktopFont = {read = FDesktopFont, write = SetDesktopFont,
    default = false};
```

與 *DesktopFont* 屬性相關的函式如下：

```
#0001 procedure TControl.SetDesktopFont(Value: Boolean);
#0002 begin
#0003     if FDesktopFont <> Value then
#0004     begin
#0005         FDesktopFont := Value;
#0006         Perform(CM_SYSFONTCHANGED, 0, 0);
#0007     end;
#0008 end;
#0009
#0010 procedure TControl.CMSysFontChanged(var Message: TMessage);
#0011 begin
#0012     if FDesktopFont then
#0013     begin
#0014         SetFont(Screen.IconFont); // 將字型設定為「圖示」字型
#0015         FDesktopFont := True;
#0016     end;
#0017 end;
```

哦～繞了一大圈，原來 VCL 早已在 *TControl* 類別偷偷預留出路，任何 *TControl* 的後代類別（即所有視覺化元件），若有「將字型維持與「圖示」字型相同」的需求，只要將 *DesktopFont* 屬性設成 *true* 即可，真是方便。

不過十分惱人的是，*TControl::DesktopFont* 屬性宣告於 *protected* 區段，若要使用它，必須自行撰寫新的元件，將 *DesktopFont* 屬性開放出來才行。VCL 雖然大架構頗有可取之處，不過像這一類的小瑕疵其實還不少。

字型的設定及維持

許多較偏向使用者層面或注重介面的應用程式，如文字編輯器、郵件軟體、個人資訊管理軟體等等，通常將顯示字型的選擇權留給使用者，不會將畫面上的字型鎖死，例如我拿來打草稿用的 UltraEdit 文書編輯器及鼎鼎有名的 Eudora 郵件軟體等等。

提供字型選擇的功能十分簡單，建立一個 *TFontDialog* 元件，設定好屬性，呼叫 *Execute* 函式，若傳回值為 *true*，表示使用者按下【確定】鈕，此時就可由 *Font* 屬性取得被選定的字型。

TFontDialog::Options 集合屬性有太多選項可供切換，較常用且常被忽略的是 *fdFixedPitchOnly* 選項。Fixed-Pitch 字型指的是該字型中每個字元所佔用的寬度皆相同，不論字元的形狀如何，而 Non Fixed-Pitch 字型就不同了，有些字元極窄，有些字元極寬，佔用的寬度可能相差好多倍。請試著開啓文字編輯軟體，先以「Arial」字型鍵入英文字母小寫 *i* 及大寫 *W*，比較兩者的寬度，再將兩字元的字型改爲「Courier New」，觀察字型更改前後的差異，Fixed-Pitch 及 Non Fixed-Pitch 字型的差別將不言自明。

這兩種字型的使用通常視使用者的習慣而定，沒有一定的規範。不過對於需要垂直對齊檢視的場合，尤其是程式碼編輯軟體，非用 Non Fixed-Pitch 字型不同，否則顯示效果簡直不堪入目。哪些字型屬於 Fixed-Pitch 家族？又哪些字型屬於 Non Fixed-Pitch 家族呢？舉些常見的字型作爲範例：

表 10-1 / 常見的 Fixed-Pitch 及 Non Fixed-Pitch 字型

	Non Fixed-Pitch	Fixed-Pitched
英文字型	Arial、MS Sans Serif、Times New Roman、System	Courier New、Fixedsys
中文字型	新細明體	細明體、標楷體

Info

歸類為 Non Fixed-Pitch 家族的中文字型（如新細明體）指的只是其英文字元為 Non Fixed-Pitch，中文字必定為 Fixed-Pitch。想想看，若一個區塊的中文字擺在一起，卻連上下左右都無法標齊，活脫脫像是不停扭動的蚯蚓大集合，能看嗎？

選定合適的字型後，想當然必須將使用者選定的字型儲存起來，下次程式開啓時直接讀入使用，省得每次執行都要再選擇字型，會出人命的。所謂的「字型」，除了最重要的字型名稱外，字型高度、大小、顏色、粗體／斜體／底線也都屬於字型資訊的一部分，要如何將字型資訊十分方便地儲存起來也是麻煩事一樁。在此提供幾個適合不同情況下使用的方法：

將字型資訊轉換為字串

若程式資訊原本即儲存在系統登錄或 INI 檔案，那麼可呼叫 xFonts 單元所提供的兩道將 *TFont* 字型物件與字串兩者互相轉換的函式。

FontToString 函式將傳入的 *TFont* 物件轉為字串，而 *StringToFont* 函式取得字串表示，轉換為 *TFont* 物件。例如：

```
FontToString(Form1->Font);
// 傳回的值會是 "MS Sans Serif", 8, [fsBold], [clWindowText]' 這種格式
StringToFont("\超研澤POP-2", 16, [fsItalic], [clRed], Font);
// 將字型更改為紅色斜體大小為 16 的超研澤POP-2字體
```

使用 TFont 物件的永續機制

想一想，幾乎每個視覺化元件都擁有 *TFont* 屬性，所以若我們在設計時期更改某個元件的字型，此元件的字型資訊也會隨著元件的其它資訊儲存在 DFM 檔案內。此行為依賴的是元件及 *TFont* 物件本身的資料永續機制，由 *TPersistent* 類別提供。隨手拿個例子，在整合環境中，叫出 Form Designer 的快捷選單選取【View As Text】，以文字表示方式

來檢視 *Form1* :

```
object Form1: TForm1
  ...
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = 32
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ...
end
```

瞧，*Form1*->*Font* 屬性被處理得好好的，由於 *TFont* 為 *TPersistent* 的子類別，所以它具有寫入資料流及自資料流讀回資訊的能力，那我們又何苦擺著現成的機制不用，花上半天功夫自行處理 *TFont* 的永續能力呢？

不幸的是，VCL 雖然擁有全套的物件永續機制，但是這些能力完全隱藏在 RAD 整合環境、Form Designer、連結程式之後，與程式設計師幾乎無緣相見。Borland 似乎認為物件永續機制除了 RAD 的支援外，不太有其它用途，所以支援這些機制的類別鮮為人知，就算有心者想切入取用，往往也綁手縛腳的，難以取得所需的功能。

TStream 類別稍微地將物件永續機制功能開放出來，它提供 *WriteComponent* 函式將元件寫入資料流，還有 *ReadComponent* 函式將元件從資料流讀回，但是對於不是元件的其它 VCL 物件就完全棄之不顧。若我想將某個元件寫入資料流，沒有問題；但若我只想單純地將元件的某個屬性寫入資料流，門都沒有。這真是十分不周到的支援，你見過哪家賣排骨飯的餐廳不能單買排骨？

只好自力救濟想法子了。首先，我們知道物件屬性永續能力的讀寫功能分別由 *TWriter* 及 *TReader* 類別支援，往裏頭找，可找到負責寫入屬性的 *TWriter::WriteProperty* 函式及負責讀取屬性的 *TReader::ReadProperty* 函式，這就是了。使用這兩個函式前還會遇到一些問題，因為這兩個函式皆宣告於 *protected* 區段，外界無法直接使用，所以必須衍生新的類別，將函式開放出來才行。經過些許努力，xStreams 單元又加入兩位生力軍－

SavePropertyToStream 函式及 *LoadPropertyFromStream* 函式²。只要正確地傳入物件及屬性名稱，就可以將屬性資料寫入資料流，或從資料流將屬性資料讀取回來。

使用這兩道函式，就可很方便地操縱、維持元件的字型資訊或其它屬性資料。例如，下列程式碼十分方便地將 *Mem01* 及 *Panel1* 元件的字型設定寫入 FONT.CFG 檔案：

```
TStream* Stream;

Stream = new TFileStream("FONT.CFG", fmCreate | fmOpenWrite);
try {
    SavePropertyToStream(Stream, Mem01, "Font");
    SavePropertyToStream(Stream, Panel1, "Font");
} __finally {
    delete Stream;
}
```

讀取字型設定也一樣地簡單，依照 *Font* 屬性的寫入順序，呼叫 *LoadPropertyFromStream* 函式時，傳入接收該 *Font* 屬性的元件，即可成功還原元件的字型資訊。這兩支函式可與任何資料流處理函式合作使用，絲毫不會干擾資料流裏其它資料的處理。

處理 SDK 提供的 TLogFont 結構

有些情況下，直接處理 SDK 提供的 *TLogFont* 結構並且單純地操縱 font handle 會比使用 VCL 提供的 *TFont* 類別來得適合³，那麼直接儲存包含字型資訊的 *TLogFont* 結構會是最直接可行的辦法。

現在，假設你手中只有 *hFont* 這個 font handle，首先你必須取得它的 *TLogFont* 結構，再將結構寫入資料流（或儲存到系統登錄、INI 檔案等等）：

² 關於這兩個函式的宣告及使用說明，請參閱附錄 A 「我的程式庫」。

³ 例如，第六章「佈景主題工具實戰」中，我們就必須直接以 *TLogFont* 結構來處理字型，而不是慣用的 *TFont* 類別。

```
TLogFont lf;  
  
GetObject(hFont, sizeof(TLogFont), &lf); // 取得 TLogFont 結構  
Stream->Write(&lf, sizeof(TLogFont));
```

關於 *TLogFont* 結構、*TFont* 類別及 font handle 三者之間的關係及轉換方式，請參閱第六章「佈景主題工具實戰」的說明。

帶著字型走

常可看到一些愛搞怪或悶騷型的程式設計師，喜歡在自己的程式裏使用很可愛、很有特色、以製造氣氛爲目的的字型，例如 X-Files 造型、耶誕樹造型或血淋淋的噁心字體等等。如果字型和程式外觀搭配得好，就像女孩的耳環、項鍊般，能有畫龍點睛之妙，立刻爲程式介面加個二十分，這是介面設計中十分可以發揮的空間。

唯一的問題是，雖然程式設計師、美術人員以及測試工程師的電腦上皆裝有這些特別的字型，但使用者的電腦可不一定有。若利用安裝程式於軟體安裝過程中，順便將這些特別的字型安裝到使用者的電腦，就可以解決此問題。不過有兩個缺點：一、這等於你同時將字型檔案贈送給軟體使用者。二、有些使用者不希望安裝軟體時又額外偷偷安裝些什麼，此類的程式專屬字型原則上只有一套應用程式會使用，而字型安裝的數目與系統的開機速度又有很大的關係（我個人覺得，這方面是 Windows 本身設計不良，字型安裝數目與系統開機速度不見得要有相依關係），易使使用者感到不快。

動態安裝及卸除字型

因此，比較好的解決方式是動態地安裝及卸除字型，程式啓動時才即時載入字型，並在程式結束前將字型釋放，其餘時間，字型單純地只是字型檔案，完全不佔用任何系統資源。動態安裝及卸除字型需要呼叫下列兩道 API 函式：

```
int AddFontResource (
    LPCTSTR    szFileName
);
```

```
int RemoveFontResource (
    LPCTSTR    szFileName
);
```

參數

szFileName 任何合法的字型檔名，可為FON、FNT、TTF或FOT形式。

回返回值

安裝：如果成功，傳回正整數，代表成功安裝的字型數目；如果失敗，傳回零。

移除：如果成功，傳回非零整數，否則傳回零。

AddFontResource 函式負責安裝字型，*RemoveFontResource* 函式則負責卸除字型，兩者皆傳入字型檔名即可。呼叫這兩個函式的最佳落腳處當然就是程式的初始化以及即將結束時。假設我想在程式中使用 X-FILES.TTF 提供的 X-Files 字型，那麼我只需在 main form 的 *OnCreate* 及 *OnClose* 事件處理函式中，加上這段程式碼：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    AddFontResource("X-FILES.TTF");
}

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
&Action)
{
    RemoveFontResource("X-FILES.TTF");
}
```

這樣一來，就可以十分放心地在程式中使用 X-Files 字型。這個作法雖然方便，不過呢，必須帶著一堆字型檔跟著執行檔跑，一來麻煩，二來難看，三來萬一不想被使用者輕易地將我心愛的字型檔拿去私用或者流傳出去，以後人手一份我就沒得現寶了。其實不只是字型檔，不論是圖示、影像、滑鼠游標等等，都有同樣的困擾。最常見的解決方法是，將檔案塞入執行檔或 DLL 內，也就是以 Windows 的資源型式挾帶在執行檔或 DLL 裏頭，不會有一堆拖油瓶唏哩嘩啦不勝其煩又易曝光的困擾。

下圖是範例程式分別於設計時期及執行時期的視窗外觀。雖然早已指定為 X-Files 字型，但由於電腦尚未安裝此字型，所以文字以預設字型呈現。程式執行後，字型被動態地加入系統，出現在眼前的即是風格獨特的 X-Files 字型。

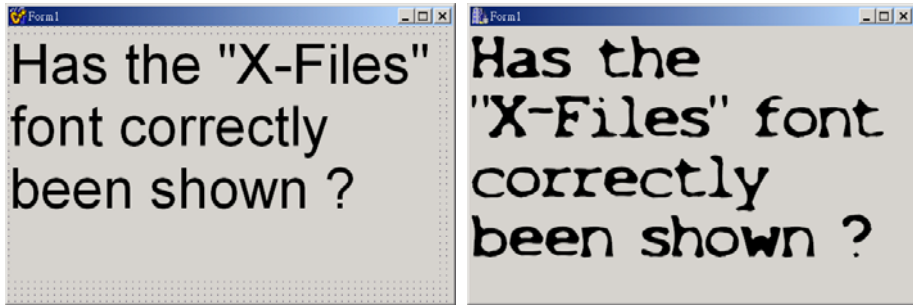


圖 10-2 / 設計時期及執行時期的視窗外觀

藏起拖油瓶

就沿用上述的範例，首先，撰寫一個 RC 檔，檔名取為 MYFONT.RC，內文指明欲轉換為資源型式的檔案名稱，如下所列：

MYFONT.RC

```
X_FILES FONTDATA "X-FILES.TTF"
```

X_FILES 是資源名稱，*FONTDATA* 是資源型態，都隨你高興可以任意自訂。

RC 檔編寫完成後，記得將 X-FILES.TTF 檔案置於同一個目錄（或者你也可以直接在 RC 檔內指明字型檔的絕對路徑），在命令列模式下達指令：

```
BRCC32 MYFONT.RC
```

BRCC32.EXE 是 Borland 的 32bit Resource Compiler，我們用它來將字型檔案包裝起來，成為 RES 檔案。經過編譯後，結果產生 MYFONT.RES，也就是所謂的資源檔案。此時在程式的任一單元加上資源含入編譯指示：

```
#pragma resource "MYFONT.RES"
```


這行編譯指示告訴連結程式，製作執行檔時，記得將 MYFONT.RES 一併連結進去加入執行檔的資源區段。通常我們會將資源含入編譯指示放在單元的程式碼之前，緊接在含入 *.DFM 檔案的編譯指示之後。

此後，就不必將 X-FILES.TTF 隨著執行檔送給使用者，因為它已經躲在執行檔內部。但是程式執行後需要安裝字型時，*AddFontResource* API 函式該傳入什麼參數呢？這裏必須進行不小的修繕工作：

```
#0001 AnsiString FontFileName;
#0002
#0003 void __fastcall TForm1::FormCreate(TObject *Sender)
#0004 {
#0005     TResourceStream* rs = new TResourceStream((int)HInstance,
#0006         "X_FILES", "FONTDATA");
#0007     try {
#0008         FontFileName = ChangeFileExt(GetTemporaryFileName(), ".TTF");
#0009         rs->SaveToFile(FontFileName);
#0010         AddFontResource(FontFileName.c_str());
#0011     } __finally {
#0012         delete rs;
#0013     }
#0014 }
#0015
#0016 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
#0017     &Action)
#0018 {
#0019     RemoveFontResource(FontFileName.c_str());
#0020     DeleteFile(FontFileName.c_str());
#0021 }
```

哇，這些步驟看起來複雜多了。首先建立一個指向字型檔資源的 *TResourceStream* 物件，呼叫 *xFiles* 單元提供的 *GetTemporaryFileName* 函式取得一個暫存檔名，並將副檔名由 .TMP 改為 .TTF 後，再呼叫 *TResourceStream::SaveToFile* 函式將字型檔取出，寫入 *FontFileName* 指向的暫時檔案。接下來就與平常一樣，呼叫 *AddFontResource* 函式來安裝字型，呼叫 *RemoveFontResource* 函式來移除字型，只是字型移除後記得刪除 *FontFileName* 指向的暫時字型檔。

這是因為 *AddFontResource* 函式只接受字型檔名為參數，無法直接處理置於記憶體內的

字型資料，所以程序才會如此麻煩。API 的缺陷及設計不良即是程式員無盡噩夢的來源，希望 Windows 在這方面能夠繼續補強。

狀態列小圖示

在TANet打混這麼多年以來，無論是牛屎鋪⁴也好，Mailing List也好，「狀態列小圖示」的問題大可榮登「Borland C++Builder常問問題金榜」，而且穩坐前三名寶座無疑。

何謂「狀態列小圖示」？大家都知道的，就是工作列最右方平常顯示著時間、日期的那塊區域上的圖示，若裝有輸入法，這兒會有「En」、「注」、「倉」等圖示表示目前使用的輸入法；若裝有防毒軟體，大概就會顯示一個紅紅的P；若音效卡支援音量調節，則可能出現一個小喇叭，供我們調節音效...越來越多的常駐型軟體就愛在此擺置一個小圖示，程式本身則不出現於工作列，若需操作軟體時請由小圖示直接選擇功能或將程式本身叫出。瞧，下圖是我的 ThinkPad 裏 Windows 98 的狀態列模樣，右方一整排的狀態列小圖示，數一數共有十二個，佔掉超過狀態列三分之一的寬度，很誇張吧！



圖 10-3 / 我的 Windows 98 狀態列，右方即是我指的「狀態列小圖示」

英文文件中，通常把這些小圖示稱為 TrayIcon。不過，TrayIcon 這字可就難翻成中文了，Tray 是盤子、托盤、文件盒的意思，Icon 當然是圖示、圖像之意，但兩個單字湊成一塊卻不好翻譯，總不能翻成「托盤圖示」之類怪異的詞彙吧。

Win32 文件中也沒有訂出正式的名稱，只說是“an icon in the taskbar status area”，這段話的中譯為「工作列狀態區域的圖示」。這些小傢伙連正名問題都十分難解，真讓人頭痛，我們就姑且稱之 TrayIcon 吧。

⁴ News group的諧音，又稱Usenet news，中文常翻為「新聞討論群組」。

雖然 `TrayIcon` 是個常被詢問的問題，VCL 元件搜集站台也總有一拖拉庫的 `TrayIcon` 元件可供下載使用。但其實，追根究底，`TrayIcon` 只是 `SHELL32.DLL` 提供的一項服務，而這項服務，出人意料的只包含一道函式及一個結構而已，簡單得嚇人吧！

WINSHELLAPI BOOL WINAPI Shell_NotifyIcon (

DWORD dwMessage,
PNotifyIconData lpData

);

參數

dwMessage *NIM_ADD* 加入新的TrayIcon
 NIM_DELETE 刪除TrayIcon
 NIM_MODIFY 修改TrayIcon的屬性或資料

lpData 指向*TNotifyIconData*結構的指標。

回返回值

如果成功，傳回非零值；如果失敗，傳回零。

唯一的資料結構 *TNotifyIconData* 為：

```
typedef struct _NOTIFYICONDATA { // nid
    DWORD cbSize; // 結構大小，設為 sizeof(TNotifyIconData)
    HWND hWnd; // 欲接收視窗訊息的 window handle
    UINT uID; // 程式自訂的 trayicon 編號
    UINT uFlags; // 若包含 NIF_ICON，表示 hIcon 可用
                // 若包含 NIF_MESSAGE，表示 uCallbackMessage 可用
                // 若包含 NIF_TIP 時，表示 szTip 可用
    UINT uCallbackMessage; // 回呼所使用的視窗訊息
    HICON hIcon; // 顯示出來的圖示
    char szTip[64];
} NOTIFYICONDATA, *PNOTIFYICONDATA;

typedef NOTIFYICONDATA TNotifyIconData;
```

使用方法很簡單，只要將 *TNotifyIconData* 結構填入適當資訊，呼叫 *Shell_NotifyIcon* API 函式，傳入適當的 *dwMessage* 參數及指向該結構的指標即可。

管理 TrayIcon

接下來是一個簡單但極具參考價值的範例。首先定義 *WM_TRAYICON* 自訂訊息，這是應用程式自訂的視窗訊息編號，一般由 *WM_APP* 訊息起跳：

```
const int WM_TRAYICON = WM_APP + 0;
```

接著是集新增、修改、刪除 TrayIcon 能力於一身的 *ModifyTrayIcon* 函式：

```
#0001 void __fastcall TForm1::ModifyTrayIcon(DWORD Action)
#0002 {
#0003     TNotifyIconData NIData;
#0004
#0005     NIData.cbSize = sizeof(TNotifyIconData);
#0006     NIData.uID = 0;
#0007     NIData.uFlags = NIF_MESSAGE | NIF_ICON | NIF_TIP;
#0008     NIData.hWnd = Handle;
#0009     // 若發生任何事件，以此訊息傳遞給 Wnd 視窗
#0010     NIData.uCallbackMessage = WM_TRAYICON;
#0011     // 與程式本身使用同樣的圖示
#0012     NIData.hIcon = Application->Icon->Handle;
#0013     // 提示文字與程式標題相同
#0014     StrPCopy(NIData.szTip, Application->Title);
#0015
#0016     // 依據 Action 去新增，修改或刪除 TrayIcon
#0017     Shell_NotifyIcon(Action, &NIData);
#0018 }
```

以 *WM_TRAYICON* 自訂訊息向 *Shell_NotifyIcon* 函式註冊後，每當有任何滑鼠訊息（*WM_MOUSEFIRST...WM_MOUSELAST*）產生，且作用於 TrayIcon 上時，就會以 *WM_TRAYICON* 視窗訊息通知視窗 *Wnd*（由 *TNotifyIconData::Wnd* 欄位指定）。你必須處理接收到的 *WM_TRAYICON* 訊息，檢查它的 *lParam* 參數以辨別實際發生的視窗訊息。

對於程式的 *main form*，在 *OnCreate* 事件處理函式中，呼叫 *ModifyTrayIcon* 函式，傳入 *NIM_ADD* 代碼即可建立新的 TrayIcon；在 *OnClose* 事件處理函式中，呼叫 *ModifyTrayIcon* 函式，傳入 *NIM_DELETE* 代碼來刪除 TrayIcon。最後，攔截 *WM_TRAYICON* 訊息，若使用者按下滑鼠右鍵，就彈出快捷選單。

```

#0001 void __fastcall TForm1::FormCreate(TObject *Sender)
#0002 {
#0003   ModifyTrayIcon(NIM_ADD);
#0004 }
#0005
#0006 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
#0007   &Action)
#0008 {
#0009   ModifyTrayIcon(NIM_DELETE);
#0010 }
#0011
#0012 void __fastcall TForm1::WMTrayIcon(TMessage& Message)
#0013 {
#0014   TPoint MousePos;
#0015
#0016   if (Message.LParam == WM_RBUTTONDOWN) {
#0017     SetActiveWindow(Handle);
#0018     GetCursorPos(&MousePos);
#0019     PopupMenu->Popup(MousePos.x, MousePos.y);
#0020   }
#0021 }

```

下圖為範例程式的執行結果：



圖 10-4 / TrayIcon 範例程式執行結果

留下 TrayIcon，其餘的都不要

而在 C++Builder 常問問題排行榜上，也有兩個問題緊咬著 TrayIcon 不放，分別是：

- 程式啟動時如何不讓 main form 出現？
- 如何讓應用程式不出現在工作列？

這兩個問題可視為 `TrayIcon` 的衍生問題，因為只有擁有 `TrayIcon` 的程式才能安心地將應用程式及 `main form` 隱藏起來的嘛。

程式啟動時如何不讓 Main Form 出現？

第一個問題，無論在 `main form` 的 `OnCreate` 事件處理函式中呼叫 `Hide` 函式、設定 `Visible` 屬性或透過 API 函式來設定視窗屬性，都無法徹底解決，還是會看到 `main form` 在畫面上閃即逝。其實，VCL 早已留下一個切換的開關，就是 `TApplication::ShowMainForm` 屬性。請打開專案原始碼，在 `form` 建立之前，將 `Application->ShowMainForm` 改為 `false` 即可：

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    Application->Initialize();
    Application->ShowMainForm = false;
    Application->CreateForm(__classid(TForm1), &Form1);
    Application->Run();
    ...
}
```

如何讓應用程式不出現在工作列？

關於第二個問題，首先，我們必須先知道哪些視窗會出現在工作列上，哪些視窗不會？答案是，最上層的視窗（表示沒有父視窗），而且沒有被其它視窗擁有的視窗（沒有 `owner`），就會出現在工作列。請跟隨我以 `SoftICE` 瞧瞧，一個正常的，包含一個 `TForm1` 視窗的 VCL 應用程式 `Project1`，它的 `Application` 視窗及 `Form1` 視窗到底擁有什麼曖昧不明的關係：

:HWND Project1					
Handle	Class	WinProc	TID	Module	
010202	IME	77E952BA	BA	00000000	
010206	TForm1	00254477	BA	00010100	
010214	TEdit	00DE0F94	BA	0200:0000	
010212	TButton	00DE0FA1	BA	00000000	

010210	TButton	00DE0FAE	BA	00000000
01020C	TMemo	00DE0FBB	BA	0200:0000
01020E	TButton	00DE0FC8	BA	00000000
010200	TApplication	00DE0FEF	BA	0100:0000
:HWND -x TApplication				
Hwnd	:	010200	(A0912430)	
Class Name	:	TApplication		
Module	:	0100:0000		
Window Proc	:	00DE0FEF	(SuperClassed from: 004063D8)	
Win Version	:	4.00		
Title	:	Project1		
Owner	:	0	// 沒有擁有者, 所以會出現在工作列	
Parent	:	01001E	(A08C0618) // 父視窗是桌面視窗	
Next	:	01018E	(A08FEB90)	
Style	:	80		
Ex. Style	:	TOPMOST STATICEDGE	20880000	
System Menu	:	010204		
Property List	:	A0912618		
Window Rect	:	640, 512, 640, 512	(0 x 0)	
Client Rect	:	3, 22, 3, 22	(0 x 0)	
:HWND -c TForm1				
Hwnd	:	010206	(A0912BE8)	
Class Name	:	TForm1		
Module	:	00010100		
Window Proc	:	00254477	(SuperClassed from: 0041FDAC)	
Win Version	:	4.00		
Title	:	Form1		
Owner	:	10200	(A0912430) // 擁有者是 Application 視窗	
Parent	:	01001E	(A08C0618) // 父視窗是桌面視窗	
Next	:	010200	(A0912430)	
1st Child	:	010214	(A0913848)	
Style	:	80		
Ex. Style	:	TOPMOST STATICEDGE	20800000	
System Menu	:	020207		
Property List	:	A0913258		
Window Rect	:	224, 173, 1094, 813	(870 x 640)	
Client Rect	:	4, 23, 866, 636	(862 x 613)	

經由查詢所得的資訊，可以很清楚地看到：form 的擁有者是 *Application* 視窗，而 *Application* 視窗沒有擁有者。這正是為什麼 *Application* 視窗會出現在工作列，而 form 卻不會的原因。

所以，工作列上的程式狀態按鈕即代表 *Application* 視窗的狀態，由於這個視窗的大小為零，所以即使它是可見視窗，但我們從來不會看見它的存在。經由 *TApplication::Handle* 屬性可取得 *Application* 視窗的視窗 handle，因此，若要隱藏或顯示工作列上的按鈕，只要針對 *TApplication::Handle* 視窗操作即可：

- 隱藏工作列按鈕

```
ShowWindow(Application->Handle, SW_HIDE);
```

- 顯示工作列按鈕

```
ShowWindow(Application->Handle, SW_SHOW);
```

至於 *Application* 物件所扮演的角色、在 VCL 應用程式中佔有的地位，以及它與 forms 之間的互動關係，值得花上專篇討論，有機會咱們再聊。

檔案捷徑管理

雖說大部分的安裝程式都具有在程式集及桌面上建立程式群組及程式捷徑的功能，但也許不小心把捷徑刪掉了，或是重灌系統後懶得重新安裝軟體，此時若應用程式本身能提供新增捷徑的能力，懶人一族會愛死你了。舉個例，佔有率極高的 MP3 撥放程式 WinAMP 就提供了這樣的功能，在「Preferences」對話盒有兩個按鈕，「Add Start Menu Items」及「Add Desktop Icon」，可分別在開始功能表及桌面上新增程式捷徑。



圖 10-5 / WinAMP 提供的新增捷徑功能

以 WinAMP 為榜樣，現在讓我們也來寫支具備建立檔案捷徑能力的小程式。

COM 物件及介面

在 Win32 內，建立捷徑最正規的方式就是透過 SHELL32.DLL 所提供的 *IShellLink* 介面來達成。「介面」是 COM⁵ 物件與外界溝通的唯一管道。當我們建立一個 COM 物件後，不能直接要求它進行任何動作，必須取得它的介面後，再透過介面來驅使物件。

COM 物件有點像一般人無法私下接觸的大明星，凡事得透過經紀人才行。必須提醒大家的是，COM 物件都擁有多個介面，而擁有多個介面的好處是可以很清楚容易地將同一類別所提供的眾多功能分類，每組相異的功能由不同介面提供。例如要設計一個影像處理類別時，就可將檔案存取及畫面控制的功能分別提供為 *IFileAccess* 及 *IUIControl* 兩個不同的介面，分別提供不同類型的成員函式。

簡短幾句話介紹 COM 物件及介面，再加上以下的範例程式及說明，希望能幫助對 COM 尚為陌生的朋友們儘早熟悉 COM 的使用。COM 不是三言兩語就說得完的，有興趣的朋友請參考附錄 C「參考書目」，選擇合適的書籍來閱讀。

ShellLink 物件及 IShellLink 介面

一步一步來，建立檔案捷徑的步驟為：

1. 建立 *ShellLink* 物件 A 並取得其 *IShellLink* 介面 B。
2. 呼叫介面 B 的成員函式設定檔案捷徑的屬性。
3. 經由介面 B 取得 *ShellLink* 物件 A 的 *IPersistFile* 介面 C（看，有多重介面）。
4. 由系統登錄查得捷徑檔案應該放置的目錄。

⁵ COM，為 Component Object Model 的頭字語。

5. 呼叫介面 C 的 *Save* 成員函式建立 LNK 檔案。

接著立即閱讀程式碼，印證上述的捷徑建立五大步驟：

```
#0001 enum TShellFolder {sfDesktop, sfFavorites, sfFonts, sfPersonal,
#0002     sfPrograms, sfRecent, sfSendTo, sfStartMenu, sfStartup,
#0003     sfTemplates};
#0004
#0005 const AnsiString ShellFolderKeys[10 /* TShellFolder */] =
#0006     {"Desktop", "Favorites", "Fonts", "Personal", "Programs",
#0007     "Recent", "SendTo", "Start Menu", "Startup", "Templates"};
#0008
#0009 void __fastcall TForm1::btnCreateShellLinkClick(TObject *Sender)
#0010 {
#0011     const int WindowStates[3 /* TWindowState */] =
#0012         {SW_SHOWNORMAL, SW_SHOWMINNOACTIVE, SW_SHOWMAXIMIZED};
#0013
#0014     if (txtFilePath->Text == "") return;
#0015
#0016     // 步驟 1
#0017     IShellLink* Psl;
#0018     if FAILED(CoCreateInstance(CLSID_ShellLink, NULL,
#0019         CLSCTX_INPROC_SERVER, IID_IShellLinkA, (void*)&Psl))
#0020         throw new Exception("Error in create instance");
#0021
#0022     // 步驟 2
#0023     Psl->SetPath(txtFilePath->Text.c_str());
#0024     Psl->SetDescription(txtDescription->Text.c_str());
#0025     Psl->SetWorkingDirectory(txtWorkingDirectory->Text.c_str());
#0026     Psl->SetArguments(txtArguments->Text.c_str());
#0027     Psl->SetHotkey(HotKey->HotKey);
#0028     Psl->SetShowCmd(WindowStates[cbxWindowStates->ItemIndex]);
#0029     Psl->SetIconLocation(txtIconLocation->Text.c_str(), 0);
#0030
#0031     // 步驟 3
#0032     IPersistFile* Ppf;
#0033     if FAILED(Psl->QueryInterface(IID_IPersistFile, (void*)&Ppf))
#0034         throw new Exception("Error in query instance");
#0035
#0036     // 步驟 4
#0037     AnsiString sFileName = ChangeFileExt(
#0038         ExtractFileName(txtFilePath->Text), ".LNK");
#0039     TRegIniFile* r = new TRegIniFile(
#0040         "Software\\Microsoft\\Windows\\CurrentVersion\\Explorer");
#0041     try {
#0042         sFileName = r->ReadString("Shell Folders",
```

```

#0043     ShellFolderKeys[cbxFolders->ItemIndex], "") + "\\\" + sFileName;
#0044     } __finally {
#0045         delete r;
#0046     }
#0047
#0048     // 步驟 5
#0049     WideChar wFileName[MAX_PATH];
#0050     // convert AnsiString to Unicode string
#0051     sFileName.WideChar(wFileName, sizeof(wFileName));
#0052     if FAILED(Ppf->Save(wFileName, true))
#0053         throw new Exception("Error in save LNK file");
#0054 }

```

btnCreateShellLinkClick 函式中，大部分皆照著上述的捷徑建立五大步驟來運作。不過在呼叫 *IPersistFile* 介面的 *Save* 函式前，必須將檔案路徑由平日使用的 *AnsiString* 型態轉為 *Unicode* 的 *WideString* 型態，在此我使用 *AnsiString* 類別的 *WideChar* 函式來轉換。

系統資料夾的真正位置

0001 列的 *TShellFolder* 列舉型態定義十種系統資料夾，*ShellFolderKeys* 陣列則記錄每個資料夾於系統登錄中對應的名稱。各個系統資料夾的目錄位置其實存放於系統登錄的 *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders* 機碼下，因此，雖然程式很短，但是卻能根據選擇很方便地將捷徑建立在「桌面」、「我的最愛」、「字型」、「個人資料夾」、「程式集」、「文件」、「傳送」、「開始」、「啟動」、「範本」等十個系統資料夾。只要查詢到資料夾所對應的磁碟目錄，就可以將 LNK 檔案擺放到正確的目錄下。

底下是在我的電腦上，以 *REGDUMP* 傾印 *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders* 機碼的結果：

```

d:\util>REGDUMP HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders

Key: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders

```

```
SendTo is REG_SZ: C:\WINNT\Profiles\jethro\SendTo
Recent is REG_SZ: C:\WINNT\Profiles\jethro\Recent
Desktop is REG_SZ: C:\WINNT\Profiles\jethro\桌面
Favorites is REG_SZ: C:\WINNT\Profiles\jethro\Favorites
Programs is REG_SZ: C:\WINNT\Profiles\jethro\「開始」功能表\程式集
Start Menu is REG_SZ: C:\WINNT\Profiles\jethro\「開始」功能表
Startup is REG_SZ: C:\WINNT\Profiles\jethro\「開始」功能表\程式集\啓動
Fonts is REG_SZ: C:\WINNT\Fonts
Personal is REG_SZ: C:\WINNT\Profiles\jethro\Personal
NetHood is REG_SZ: C:\WINNT\Profiles\jethro\NetHood
PrintHood is REG_SZ: C:\WINNT\Profiles\jethro\PrintHood
Templates is REG_SZ: C:\WINNT\ShellNew
AppData is REG_SZ: C:\WINNT\Profiles\jethro\Application Data
AltStartup is REG_SZ: C:\WINNT\Profiles\jethro\「開始」功能表\程式集
\Startup
Cache is REG_SZ: C:\WINNT\Profiles\jethro\Temporary Internet Files
Cookies is REG_SZ: C:\WINNT\Profiles\jethro\Cookies
History is REG_SZ: C:\WINNT\Profiles\jethro\History
```

檔案捷徑管理這主題就拿範例程式的執行畫面做為結尾囉。

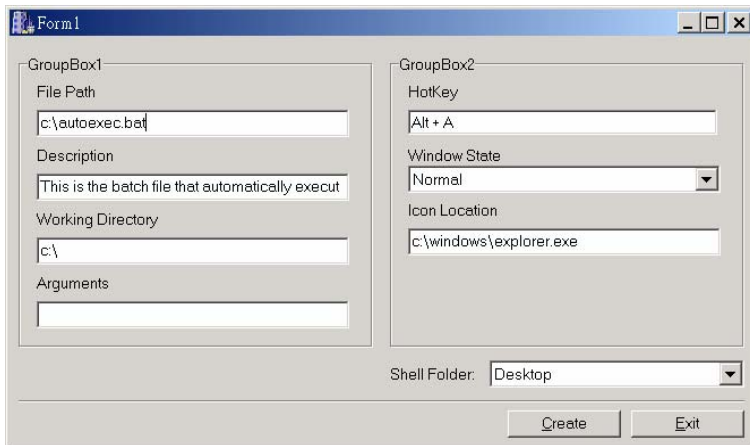


圖 10-6 / 建立 Shell Link 範例程式執行畫面

維持視窗屬性

對於使用頻繁的應用軟體，最痛恨的就是不會自動儲存視窗位置、大小及狀態的軟體。上次使用時明明已經配合桌布、桌面圖示及工具列的位置將視窗的尺寸、座標調整好，結果程式竟然沒有維持視窗屬性的功能，每次啟動時都要重新調整，實在太不體貼使用者了。

幸好，許多的應用程式都已考慮這一點，本身即具備儲存及回復視窗位置尺寸及狀態的能力。實作起來相當簡單，我們可以分別在 form 的 *OnCreate* 及 *OnClose* 事件觸發時回復及儲存視窗資訊：

```
#0001 const char* KEY_REGISTRY = "\\Software\\Jethro\\PosSize";
#0002 const char* SEC_SAVEFORM = "SaveForm";
#0003
#0004 void __fastcall TForm1::FormCreate(TObject *Sender)
#0005 {
#0006     TRegIniFile* r = new TRegIniFile(KEY_REGISTRY);
#0007     try {
#0008         Width = r->ReadInteger(SEC_SAVEFORM,
#0009             AnsiString(ClassName()) + "_Width", Width);
#0010         Height = r->ReadInteger(SEC_SAVEFORM,
#0011             AnsiString(ClassName()) + "_Height", Height);
#0012
#0013         Top = r->ReadInteger(SEC_SAVEFORM,
#0014             AnsiString(ClassName()) + "_Top", Top);
#0015         Left = r->ReadInteger(SEC_SAVEFORM,
#0016             AnsiString(ClassName()) + "_Left", Left);
#0017
#0018         WindowState = TWindowState(r->ReadInteger(SEC_SAVEFORM,
#0019             AnsiString(ClassName()) + "_WindowState", WindowState));
#0020
#0021         Visible = r->ReadBool(SEC_SAVEFORM,
#0022             AnsiString(ClassName()) + "_Visible", True);
#0023     } __finally {
#0024         delete r;
#0025     }
#0026 }
#0027
#0028 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
#0029     &Action)
```

```
#0030 {
#0031   TRegIniFile* r = new TRegIniFile(KEY_REGISTRY);
#0032   try {
#0033     r->WriteInteger(SEC_SAVEFORM,
#0034       AnsiString(ClassName()) + "_Width", Width);
#0035     r->WriteInteger(SEC_SAVEFORM,
#0036       AnsiString(ClassName()) + "_Height", Height);
#0037
#0038     r->WriteInteger(SEC_SAVEFORM,
#0039       AnsiString(ClassName()) + "_Top", Top);
#0040     r->WriteInteger(SEC_SAVEFORM,
#0041       AnsiString(ClassName()) + "_Left", Left);
#0042
#0043     r->WriteInteger(SEC_SAVEFORM,
#0044       AnsiString(ClassName()) + "_WindowState", WindowState);
#0045
#0046     r->WriteBool(SEC_SAVEFORM,
#0047       AnsiString(ClassName()) + "_Visible", Visible);
#0048   } __finally {
#0049     delete r;
#0050   }
#0051 }
```

這段程式碼使用 VCL 提供的 *TRegIniFile* 類別來存取系統登錄。*TRegIniFile* 類別衍生自 *TRegistry* 類別，因此同樣具有存取系統登錄的能力，但使用方法與用來讀寫 INI 檔案的 *TIniFile* 類別幾乎完全相同，呼叫起來也乾淨方便的多。因此，大部分情況下我都以 *TRegIniFile* 類別來存取系統登錄。

上述的方法雖然方便，但是一個程式通常包括不只一個視窗，再加上工具列、對話盒等等，若每個視窗都得寫一段程式碼來維持視窗屬性豈不麻煩透了。元件在此是最佳的選擇，你可以嘗試筆者撰寫的 *TWinSaver* 元件或至 Delphi 深度歷險網站下載其它的元件來比較使用。以 *TWinSaver* 元件為例，只要將它扔到 form 上，它就會自動攔截 form 的 *OnCreate* 及 *OnClose* 事件來進行上述的視窗狀態記錄及維持工作，而且不限於 main form，它會使用 form 的類別名稱做為識別字來管理多個 form 的屬性維持。

執行一份足矣

有許多軟體具有只能執行一份的特性，例如佔用音效輸出裝置的 MP3 撥放程式、使用同一個 TCP 連接埠的伺服器端軟體或全螢幕模式進行的遊戲等等；也有些軟體雖然不具這些特性，但以使用方式而言就算開啓多份也沒有意義，尤其是 MDI 型式的應用程式，例如郵件軟體、文字編輯器、字典工具等等。拿我日常常用的軟體來說，管理檔案的 Windows Commander、撰寫文字的 UltraEdit、閱讀電子郵件的 Eudora 以及即時翻譯軟體 Dr.Eye 譯典通都屬於這一類，當它們執行的時候，即使我另外再執行一次，新的副本不會出現，而會將視窗焦點轉移至原來執行中的那一份程式，讓使用者曉得程式已經執行了。

尋找前一份副本

程式啓動時，如何得知同一支程式是否已經執行呢？方法很多很多，幾乎只要是能夠跨越行程藩籬的機制就可使用，舉例如下：

- **FindWindow**
尋找應用程式視窗是否存在，同時得到其視窗 handle。
- **Atom**
將某個特定字串加入全域的字串表格（global atom table），後來的副本藉由尋找此特定字串來判定前一副本是否存在。
- **Window Property**
將某個數值加入視窗的 property list，後來的副本必須搜尋每個視窗的 property list 來判定前一副本是否存在。
- **Mutex、Semaphore、Event**
以上三種為 Win32 的執行緒同步物件，使用方式相同。先建立一個具名的核心物件，後來的副本藉由開啓此具名核心物件是否成功來判定前一副本是否存在。
- **File Mapping**
建立跨行程的共享記憶區域，可將視窗 handle 或其它資訊放入。後來的副本藉由開啓此具名的 file mapping 物件是否成功來判定前一副本是否存在，若前一副本存在，可同時取得其視窗 handle 及其它相關資訊。

能夠跨越行程藩籬的機制不少，全憑程式設計師如何活用。尋找前一份副本是否存在是

最重要的課題，但是在確定前一副本早已存在後（即目前的行程必須結束，並將控制權交給前一份副本），最好能夠同時取得前一份副本行程編號、執行緒編號或視窗 `handle` 等等資訊，以便與前一副本進行溝通。下面舉出三種簡單且實用的解決方案：

單純的視窗尋找

此方法完全不需要額外的機制或資源來輔助，可說是最簡單最輕量級的解決方案。此方案依賴的是 `FindWindow` API 函式，此函式依照視窗標題及視窗類別名稱搜尋系統中的最上層視窗，並回傳符合條件的第一個視窗 `handle`。假設專案的 `main form` 類別名稱為 `TForm1`，標題為 “This is a test program”，那麼，請打開專案原始碼，於主函式 `WinMain` 加入以下程式碼：

```
#0001 WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0002 {
#0003     HWND Wnd;
#0004
#0005     // 找尋最上層視窗中，類別名稱爲 TForm1
#0006     // 標題爲 "This is a test program" 的視窗
#0007     Wnd = FindWindow("TForm1", "This is a test program");
#0008     if (Wnd) { // 找到沒？
#0009         // 將上一份執行副本的 main form 拉到前面來
#0010         SetForegroundWindow(Wnd);
#0011         // 然後結束新執行的這份程式
#0012         return 0;
#0013     }
#0014
#0015     ...
#0016 }
```

此方法雖然簡單，但有兩個問題亟待解決：

Main Form 的標題可能更動

Main form 的標題並不是恆久不變，通常會隨著軟體目前的使用狀態或情形動態變更。若將視窗尋找的條件放寬，只根據視窗類別名稱來尋找，這種方式又太不保險了，因為很有可能另一套軟體的 `main form` 類別名稱也叫 `TForm1` 或 `TMainForm`，視窗類別名稱萬一

重覆，就會因為此套軟體的存在導致誤判，這是無法接受的情形。

C++Builder 整合環境內的雙胞胎弟兄

在 C++Builder 整合環境中所看到的每個 form，都是貨真價實的視窗。因此，若在整合環境開啓 main form 來進行設計時，按下【F9】執行程式，*FindWindow* API 函式就會產生誤判，取得整合環境內的 main form 視窗而認為是前份執行副本。為了預防此情形，我通常會在重覆執行檢查碼的前後各加上 `#ifdef RELEASE` 及 `#endif` 條件編譯指示，只有在程式完成後，才加入 `#define RELEASE` 編譯指示，啓動這段程式碼。

改良型視窗尋找

針對上一個方案的兩個問題，我將 *FindWindow* 函式的尋找對象，由程式的 main form 轉移到程式的 *Application* 視窗⁶。

與 main form 不同的是，*Application* 視窗的標題，亦即在 C++Builder 整合環境選擇

【Project / Options / Application / Title】所填的字串，也就是工作列上所出現的應用程式標題，通常不會在程式執行時動態更動，所以可以放心地依賴 *FindWindow* 函式去尋找它。

另一方面來看，*Application* 視窗是由 *TApplication* 物件產生，而 *TApplication* 物件只有在程式執行時才建立，所以不會和 main form 一樣，被整合環境內的設計視窗干擾。

利用這些特性，我們可以將專案原始碼改成這樣：

```
#0001 WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0002 {
#0003 // 找尋最上層視窗中，類別名稱爲 'TApplication'
#0004 // 標題爲 "This is my project title !!" 的視窗
#0005 HWND AppWnd = FindWindow("TApplication", "My Project Title!");
#0006 if (AppWnd) { // 找到沒？
```

⁶ 此指由 *TApplication* 物件所建立的視窗，視窗大小爲 0 x 0。

```
#0007     if (IsIconic(AppWnd)) // 若處於最小化狀態，將它復原
#0008         ShowWindow(AppWnd, SW_RESTORE);
#0009     else // 拉到前面來
#0010         SetForegroundWindow(AppWnd);
#0011
#0012     return 0;
#0013 }
#0014
#0015     try
#0016     {
#0017         Application->Initialize();
#0018         Application->Title = "My Project Title!";
#0019         Application->CreateForm(__classid(TForm1), &Form1);
#0020         Application->Run();
#0021     }
#0022     catch (Exception &exception)
#0023     {
#0024         Application->ShowException(&exception);
#0025     }
#0026     return 0;
#0027 }
```

簡而言之，此方法就是以 *Application* 視窗代替 *main form* 視窗，雖然程式碼幾乎沒有變動，但是就可因此避免前一個方法所衍生的兩個問題。

共享記憶區域

利用 *xMemory* 單元提供的 *MapGlobalData*、*ReleaseGlobalData* 及 *IsGlobalDataExistent* 三道共享記憶區域管理函式，也可以作為判定程式是否重覆執行以及傳遞前一副本資訊的解決方案。使用的步驟大致如下：

1. 呼叫 *IsGlobalDataExistent* 函式，檢查是否已存在名稱為 *MapName* 的共享區域？如果存在，表示此程式的前一副本正在執行，進行步驟二；否則沒有前一副本，進行步驟三。
2. 若前一副本存在，則開啓共享記憶區域，取得前一副本的 *Application* 視窗 *handle*，將此視窗提升為前景視窗，然後結束執行（行程本身）。
3. 若沒有前一副本，則開啓共享記憶區域，將本身的 *Application* 視窗 *handle* 填入。

4. 記得在程式結束前，關閉共享記憶區域。

根據以上步驟，將專案原始碼改寫成：

```
#0001 WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
#0002 {
#0003     typedef struct { // 共享記憶區域的資料結構
#0004         HWND AppWnd; // Application 視窗 handle
#0005     } TSharedStruct, *PSharedStruct;
#0006
#0007     const char* MapName = "MYMAP"; // 共享記憶區域名稱
#0008
#0009     PSharedStruct GlobalPointer;
#0010     THandle hMap;
#0011
#0012     if (IsGlobalDataExistent(MapName)) { // 共享區域是否已建立？
#0013         // 開啓共享記憶區域
#0014         hMap = MapGlobalData(MapName, sizeof(TSharedStruct),
#0015             Pointer(GlobalPointer));
#0016
#0017         // 若處於最小化狀態，將它復原
#0018         if (IsIconic(GlobalPointer->AppWnd))
#0019             ShowWindow(GlobalPointer->AppWnd, SW_RESTORE);
#0020         else // 拉到前面來
#0021             SetForegroundWindow(GlobalPointer->AppWnd);
#0022
#0023         // 釋放共享記憶區域
#0024         ReleaseGlobalData(hMap, Pointer(GlobalPointer));
#0025
#0026         // 然後結束新執行的這份程式
#0027         return 0;
#0028     }
#0029
#0030     // 建立共享記憶區域
#0031     hMap = MapGlobalData(MapName, sizeof(TSharedStruct),
#0032         Pointer(GlobalPointer));
#0033     // 填入本身的 Application 視窗 handle
#0034     GlobalPointer->AppWnd = Application->Handle;
#0035
#0036     try
#0037     {
#0038         Application->Initialize();
#0039         Application->CreateForm(__classid(TForm1), &Form1);
#0040         Application->Run();
#0041     }
#0042     catch (Exception &exception)
```

```
#0043 {  
#0044     Application->ShowException(&exception);  
#0045 }  
#0046  
#0047 // 釋放共享記憶區域  
#0048 ReleaseGlobalData(hMap, Pointer(GlobalPointer));  
#0049  
#0050     return 0;  
#0051 }
```

這個方法的好處是，若有新副本傳遞參數或資訊給前一副本的需求，你很快就可以看到，藉著記憶共享區域的特性，只需將參數或其它資訊放入 *TSharedStruct* 結構，加上一點點的修改，立即擁有資訊傳遞的能力。

傳遞參數及資訊

程式啟動時，若發現有另一份副本正在執行，就將自己關閉，並且將視窗焦點轉移至前一份副本。除此之外，若此程式可接受執行參數，那麼最體貼使用者的設計是，新的副本在結束自己之前，將它所接收的程式參數傳遞給前一副本，交由它來處理使用者的需求。

假設我撰寫了一個支援WMA⁷格式音效檔案的播放程式，取名為WMAPLAY。執行它時，可同時傳入一個檔名作為參數，它會立即開啓此檔案來播放。

此時，在阿達的電腦上，WMAPLAY 正播放著「心動.WMA」，這首歌阿達今天已經聽了好幾十遍了...「換一首吧！」他想。

於是，糊里糊塗的他，選擇【開始 / 執行】，鍵入這行命令：

```
WMAPLAY c:\music\壁花.WMA
```

另一份 WMAPLAY 程式隨之而起，首先啟動重覆執行檢查碼，發現早有另一份 WMAPLAY 程式正在執行。於是：

⁷ WMA為Microsoft所推出，欲取代MP3 成為新一代音效檔案的語音壓縮格式。

□ 狀況 A

新的 WMAPLAY 程式將視窗焦點轉移給原來的 WMAPLAY 程式，隨後立即結束執行。阿達覺得很奇怪，怎麼命令一點作用都沒有咧，再試一次，結果仍相同——一點反應都沒有。這時他突然抬起頭來看看牆上的掛鐘，發現今天正好是十三號星期五，而現在的時間正好是 13:05AM，一股涼意快速地從背脊竄了上來...

□ 狀況 B

新的 WMAPLAY 程式接收到的檔案參數 “c:\music\壁花.WMA” 傳遞給原來的 WMAPLAY 程式，並將視窗焦點轉移給它，結束執行。林曉培的「心動」正唱到一半，立刻換上阿雅唱起「壁花」，阿達愣了一下，搞清楚是怎麼一回事後，立刻回到 BBS 上繼續跟網友哈拉...

呵呵，別替阿達擔心，這當然只是假想情況。不過，身為程式設計師的你，希望看到狀況 A 的出現，還是狀況 B 的發生呢？

所以，對於能夠接收執行參數的程式，找到前一份副本後，新的副本還要盡責地將資訊傳達之後，才能夠功成身退，離開沙場。由於尋找前一副本方法的性質不同，最適合的傳遞資訊的方法也有所不同。

WM_COPYDATA 視窗訊息

取得前一副本的視窗 handle 後，可以利用視窗訊息將參數傳遞過去，再由對方將視窗訊息攔截下來，取得參數進行處理。有兩點考量：

□ 必須使用 WM_COPYDATA 視窗訊息

因為只有 WM_COPYDATA 視窗訊息才能跨行程地傳遞大量資料，其它的視窗訊息只能傳遞 wParam 及 lParam 參數⁸，在此不敷使用，因為參數可能包含一長串文字或是一大段數值資料。

⁸ WM_SETTEXT 視窗訊息是個例外，由於作業系統的支援，它能夠傳遞字串給其它行程的視窗函式。

- 必須取得 main form 的視窗 handle

WM_COPYDATA 視窗訊息傳送過去後，前一副本必須將訊息攔截下來處理才行。而 *Application* 視窗並不好攔截，因為 *Application* 視窗由 *Application* 物件建立，而 *Application* 物件由 VCL 的 Controls 單元建立，並非 C++Builder 程式員所能掌控。當然我們也可以經由 subclass *Application* 視窗的方法來攔截訊息，不過這太麻煩了。比較簡單的方式是，將訊息傳遞給程式的 main form，讓 main form 負責攔截及處理 *WM_COPYDATA* 視窗訊息。

Tips

也許你注意到了 *TApplication* 的 *OnMessage* 事件，不過在這兒它並不適用。因為 *WM_COPYDATA* 訊息必須呼叫 *SendMessage* 函式以直接傳遞的方式交給視窗函式處理，而 *TApplication::OnMessage* 事件只能攔截投遞到訊息佇列中的視窗訊息。

再者，每當主執行緒有任何視窗訊息由訊息佇列取出處理時，*TApplication::OnMessage* 事件都會觸發，不論該訊息傳送的目的是哪個視窗。所以除非必要，不要輕易攔截 *TApplication::OnMessage* 事件，一旦沒寫好，可能對程式效率造成不小的影響。

經由上述的尋找前一份副本的第二個解決方案，我們只取得前一副本的 *Application* 視窗 handle，那要如何才能取得 main form 的視窗 handle 呢？方法也有好多好多，例如：

- 呼叫 *EnumWindows* API 函式，尋訪所有最上層視窗。對於每一個視窗類別名稱與 main form 類別名稱相同的視窗，若視窗的 owner 是前一副本的 *Application* 視窗，那麼就可以確定該視窗就為前一副本的 main form。
- 將 main form 的視窗 handle 加入 *Application* 視窗的 property list，取得前一副本的 *Application* 視窗 handle 後，就可經由 property list 直接取得前一副本的 main form 視窗 handle。

第一個方法需要尋訪系統所有最上層視窗，效率當然不如第二個直接取得結果的方法來得高。所以我選擇第二種方法來做示範。

處理 property list

Property list 是項雖不起眼，但時常能幫上大忙的機制。每個視窗維護一個 property list，可將它視為一個以字串為索引的 *Handle* 型態陣列，將某個 handle 加入 property list 後，任何行程皆可以同樣的字串索引取出該 handle。

在這兒，我們以“Main Form Handle”字串做為索引，將 main form 的視窗 handle 加入 *Application* 視窗的 property list。由於必須等到 main form 建立之後才有視窗 handle 可取用，所以在它的 *OnCreate* 事件處理函式中，呼叫 *SetProp* API 函式來加入 handle：

```
#0001 void __fastcall TForm1::FormCreate(TObject *Sender)
#0002 {
#0003     // 將 main form 視窗 handle 加入 Application 視窗的 property list
#0004     SetProp(Application->Handle, "Main Form Handle", Handle);
#0005 }
```

然後，在專案原始碼中，取得前一副本的 *Application* 視窗 handle 後，就可呼叫 *GetProp* API 函式，傳入相同的字串索引，取得 main form 的視窗 handle：

```
// 取得前一副本的 main form handle
MainWnd = GetProp(AppWnd, "Main Form Handle");
```

傳遞參數

接著，使用 *ParamCount* 及 *ParamStr* 兩道函式，取得傳給本行程的所有參數，將它們組成參數字串後，再呼叫 *SendMessage* 函式，藉由 *WM_COPYDATA* 視窗訊息將參數字串傳遞給前一副本的 main form—*MainWnd*。

```
#0001     // 將參數組合起來
#0002     AnsiString ParamString = ParamStr(0);
#0003     for (int i = 1; i <= ParamCount(); i++)
#0004         ParamString = ParamString + " " + ParamStr(i);
#0005
#0006     // 利用 WM_COPYDATA window message 將新副本所下的參數傳遞過去
#0007     TCopyDataStruct CopyDataStruct;
#0008     CopyDataStruct.cbData = ParamString.Length() + 1;
#0009     CopyDataStruct.lpData = ParamString.c_str();
#0010     SendMessage(MainWnd, WM_COPYDATA, 0, LPARAM(&CopyDataStruct));
```

接收新副本傳來的參數字串

參數字串傳遞出去後，新的副本任務結束，功成身退。而前一副本呢？它的 `main form` 必須攔截 `WM_COPYDATA` 視窗訊息，將參數字串取出處理：

```
#0001 void __fastcall TForm1::WMCopyData(TWmCopyData& Message)
#0002 {
#0003     AnsiString ParamString;
#0004
#0005     ParamString = AnsiString((char*)Message.CopyDataStruct->lpData);
#0006
#0007     Mem1->Lines->Add("收到執行請求:");
#0008     Mem1->Lines->Add(ParamString);
#0009     Mem1->Lines->Add("");
#0010 }
```

下圖是此範例程式的執行畫面，它只將收到的新副本參數列在 `Mem1` 元件中。真正的應用程式當然不能這樣做，必須妥善分析收到的參數字串，做出適當的回應才行。

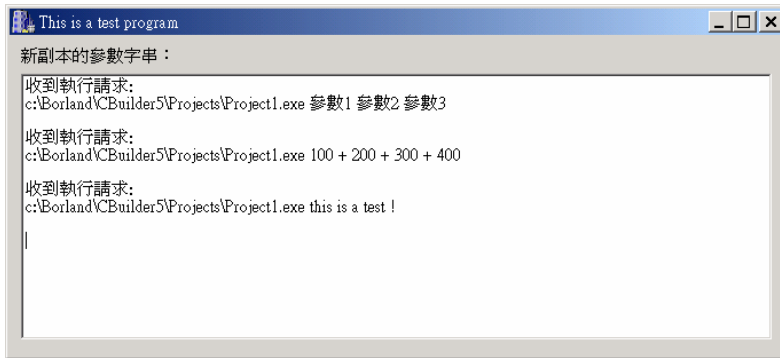


圖 10-7 / 能夠接收新副本傳來的參數字串

共享記憶區域

如果你使用共享記憶區域的方式來尋找前一副本的存在，那麼，加入傳遞參數或其它程式啟動資訊的功能真如反掌折枝般的輕鬆愉快。

只要將需要傳遞的資訊加入 `TSharedStruct` 結構，當新副本找到前一副本時，傳遞一道自

訂視窗訊息（例如 *WM_APP*）過去，通知前一副本已經將資料寫入共享記憶區域了，新副本就可結束執行。而原有副本接收到此自訂視窗訊息後，只消將參數或其它資訊自 *TSharedStruct* 結構取出使用即可，行程間的資料傳遞動作完全交給共享記憶區域機制負責。

站在使用者的角度思考，你的應用程式是否適合只容許單一副本的形式？設計為單一副本的形式是否可增加操作上的便利性？能為使用者省下操作時間或是增加使用者的困擾呢？只要讓使用者覺得自然一點、便利一點，程式設計師多寫幾百行程式也值得，畢竟一個程式設計師付出，就可有幾十、幾百、幾千甚至幾十萬使用者受益，何樂不為？單一副本執行的支援到底重不重要還是其次，可是連這點小地方都不放過，程式員的用心及體貼可見一斑。

檔案拖曳支援

在 GUI 及滑鼠運動大行其道的今日，若一個與檔案相關的軟體不支援檔案的拖曳功能，就像是沒有提供解除馬賽克效果的影像處理軟體，或是沒有提供定點放大功能的影片播放程式，只有一句話來形容它們，遜斃了！

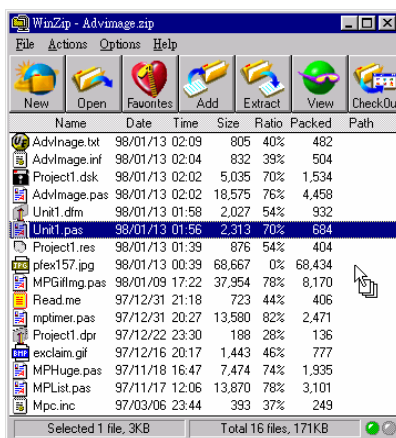


圖 10-10 / WinZIP 可接受從檔案總管拖曳過來的檔案

上圖左方是知名的壓縮／解壓縮工具 WinZIP 的執行畫面，我正從檔案總管拖曳一堆檔案過去，打算將它們加入原有的 ZIP 檔。拖曳的來源處不一定要是檔案總管哦，也可以從「我的電腦」或桌面，直接將檔案或捷徑抓進 WinZIP。這真是太神奇了，傑克，它是怎麼辦到的呢？

對於一項從未接觸的技術，最佳解答就隱藏在問題的來源本身。於是我搬出工具箱，利用C++Builder所附的TDUMP⁹工具來檢視WinZIP32.EXE的import table：

```
c:\winapp\winzip>TDUMP WINZIP32.EXE
...
Imports from SHELL32.dll
  ShellExecuteA(hint = 005e)
  DragAcceptFiles(hint = 000c)
  DragQueryFileA(hint = 000e)
  DragFinish(hint = 000d)
  DragQueryPoint(hint = 0011)
  FindExecutableA(hint = 0020)
...
```

當然，WinZIP32.EXE的import table不只這些，根據我統計的結果，它總共使用了七個DLL所提供的 346 道函式¹⁰。由函式名稱可十分清楚地看出，WinZIP的檔案拖曳功能的確是由SHELL32.DLL中四個以*Drag*字串開頭的函式所提供，是它們讓程式擁有接收從檔案總管及桌面（嚴格地說，應指EXPLORER.EXE + SHELL32.DLL所提供的Shell Space）拖曳過來檔案的能力。

使用檔案拖曳支援函式

上述的四道檔案拖曳支援函式其函式宣告如下：

⁹ 關於檔案檢視及刺探工具，請參閱附錄 B 「我的工具箱」。

¹⁰ 從import table只能得到被此程式implicitly linked的函式，無法看出explicitly linked的函式。

```

VOID DragAcceptFiles(HWND hWnd, BOOL fAccept);
UINT DragQueryFile(HDROP hDrop, UINT iFile, LPTSTR lpszFile,
    UINT cch);
BOOL DragQueryPoint(HDROP hDrop, LPPOINT lppt);
VOID DragFinish(HDROP hDrop);

```

身為一個預備接受外來檔案的視窗，它的行為必須是：

- 欲進入可接受拖曳檔案的狀態時，呼叫 *DragAcceptFiles* 函式，傳入其視窗 handle，*Accept* 參數為 *true*。
- 收到 *WM_DROPFILES* 視窗訊息時，取得隨訊息帶來的 drop handle，接著可呼叫 *DragQuery* 系列函式以取得檔案及拖檔落點座標資訊。當必要的資訊全部取得後，呼叫 *DragFinish* 函式，傳入 drop handle，告知此次的檔案拖曳動作已經處理完成。
- 若想回復正常狀態（無法接受拖曳檔案），呼叫 *DragAcceptFiles* 函式，傳入其視窗 handle，*Accept* 參數為 *false*。

簡單來說，這四道函式的使用次序如下圖：

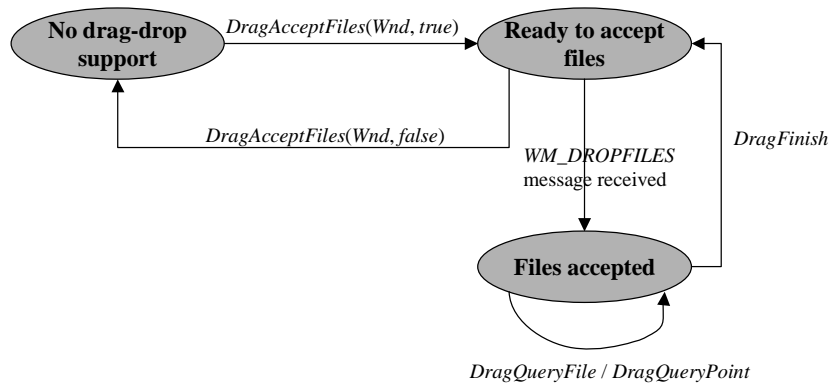


圖 10-11 / 檔案拖曳支援函式的使用次序圖

Info

呼叫 *DragAcceptFiles* 函式可啟動及結束視窗的檔案拖曳支援。不過還有另外一種控制檔案拖曳支援機制方式，就是更改視窗的延伸風格。

只要視窗的延伸風格包含 *WS_EX_ACCEPTFILES* 旗標，它就可接收拖曳而來的檔案，反之則否。你可以藉由改寫 *TWinControl::CreateParams* 函式或呼叫 *GetWindowLong*、*SetWindowLong* API 函式來修改視窗的延伸風格。

取得檔案拖曳資訊

收到 *WM_DROPFILES* 視窗訊息後，只有兩種動作可進行：呼叫 *DragQueryFile* 函式取得每一個被拖曳檔案的檔名及路徑，以及呼叫 *DragQueryPoint* 取得滑鼠鍵放開的落點座標，此座標為接收拖曳檔案視窗客戶端區域的相對座標。

```
UINT DragQueryFile (
    HDROP          Drop,
    UINT           FileIndex,
    LPTSTR         FileName,
    UINT           cb
);
```

參數

<i>Drop</i>	由 <i>WM_DROPFILES</i> 訊息取得的 drop handle。
<i>FileIndex</i>	檔案編號，從 0 ~ 被拖曳檔案數目 - 1。若傳入 0xFFFFFFFF，將傳回被拖曳檔案的數目。
<i>FileName</i>	指向字元陣列的指標，用來取得指定檔案的絕對路徑。
<i>cb</i>	字元陣列的大小。

回返回值

若成功取得指定檔案的檔名，傳回取得的路徑長度。

若 *FileIndex* 參數為 0xFFFFFFFF，傳回被拖曳檔案的數目。

若檔案編號合法，但 *FileName* 參數為 *NULL*，傳回檔案路徑的長度。

若有任何錯誤，傳回零。

DragDrop 範例程式

在 *DragDrop* 範例程式中，我希望使整個 *Form1* 皆可接受來自 *Shell Space* 的檔案，所以分別在它的 *OnCreate* 及 *OnClose* 事件處理函式中呼叫 *DragAcceptFiles* 函式，以啟動及取消檔案拖曳功能的支援：

```
#0001 void __fastcall TForm1::FormCreate(TObject *Sender)
#0002 {
#0003     DragAcceptFiles(Handle, true); // 進入可接收狀態
#0004 }
#0005
#0006 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
#0007     &Action)
#0008 {
#0009     DragAcceptFiles(Handle, false); // 結束可接收狀態
#0010 }
```

接下來唯一的動作便是攔截 *WM_DROPFILES* 視窗訊息，在裡頭一一查詢被拖曳的檔案路徑，列在 *ListBox1* 元件中。

```
#0001 void __fastcall TForm1::WMDropFiles(TWMDropFiles& Message)
#0002 {
#0003     // 取得被拖曳檔案數目
#0004     int num = DragQueryFile((HDROP)Message.Drop, 0xFFFFFFFF, NULL, 0);
#0005     lblCount->Caption = Format("接收到 %d 個檔案",
#0006         OPENARRAY(TVarRec, (num)));
#0007
#0008     ListBox1->Items->Clear();
#0009     for (int i = 0; i < num; i++) { // 一一查詢檔名，加入 ListBox1
#0010         char buf[256];
#0011         DragQueryFile((HDROP)Message.Drop, i, buf, sizeof(buf));
#0012         ListBox1->Items->Add(buf); // 加入檔案列表
#0013     }
#0014
#0015     DragFinish((HDROP)Message.Drop); // 結束 WM_DROPFILES 處理動作
#0016 }
```

0009 ~ 0012 列只是很簡單地查詢每個檔案的絕對路徑，然後將它們塞進 *ListBox1* 元件，

就這麼簡單。



圖 10-12 / DragDrop 範例程式執行畫面

行程的最後一刻

行程的結束有各種原因，也許是下列這些：

- 使用者下拉視窗左上角的系統選單，選擇關閉。
- 使用者以滑鼠左鍵單擊視窗右上角的關閉按鈕。
- 使用者按下【ALT - F4】熱鍵。
- 使用者從功能表選擇【檔案 / 離開】。
- 應用程式本身呼叫 `main form` 的 `Close` 函式。
- 應用程式本身呼叫 `TApplication::Terminate` 函式。
- 應用程式發生無法收拾的錯誤或例外，強制結束。
- 其它應用程式傳遞視窗訊息要求 `main form` 關閉。
- 其它應用程式呼叫 API 函式強制行程結束。

當然，還有其它可能的方式，這些只是常見的情況。不論如何，只要程式依正常程序結束，`main form` 就會依序觸發下列的事件：

1. `OnCloseQuery` 事件

此事件用來確定視窗是否可以關閉。如果不希望關閉，將 `CanClose` 參數設為 `false` 即可。通常我們在此檢查工作是否完成、文件是否存檔等等，並提供使用者一個最後確

認的機會，依檢查的結果及使用者的回應來決定是否接受程式結束。

2. *OnClose* 事件

大部分程式在這裏進行資料儲存及資源清理的動作。雖然名為 *OnClose* 事件，但事實上在這個階段尚可翻身，只要改變 *Action* 參數（型態為 *TCloseAction*），將它從預設值 *caFree* 改為 *caNone*，視窗關閉動作就不會繼續。

3. *OnDestroy* 事件

在 *form* 物件摧毀之前所觸發的事件，我們也可利用此事件進行善後工作。

但是，當程式執行中，使用者欲簽出、關閉或重新啟動系統時，*main form* 卻只有下列事件被觸發：

1. *OnCloseQuery*事件

由 *WM_QUERYENDSESSION* 視窗訊息驅動，系統關閉前，會將此訊息廣播給所有的視窗，徵得所有視窗的同意後才關閉系統¹¹。若將 *CanClose* 參數設為 *false*，表示程式拒絕結束，那麼系統就不會關閉，程式就可以苟活下去了。

2. *OnDestroy* 事件

在 *form* 物件摧毀之前所觸發的事件，我們也可利用此事件進行善後工作。

可以發現到，一向信用良好的 *TForm::OnClose* 事件竟然未觸發，真是出人意料。嗚呼，難怪有些程式平時行為十分良好，但若放心地讓它背景執行，直到系統關閉時才結束，反而會出現奇怪的行為，現在知道答案了。：)

萬無一失的善後工作

每當系統發出 *WM_QUERYENDSESSION* 訊息——詢問所有視窗是否可以關閉系統之後，緊接著系統會再廣播 *WM_ENDSESSION* 訊息，告知所有視窗方才的調查結果。我們可從 *TWMEndSession::EndSession* 欄位得知系統是否確定要關閉，如果是的話，就趁此機

¹¹ 這是指「優雅的」系統關閉行為而言。當然，也可以強迫系統關閉，不論其它程式是否同意。

會趕緊進行最後的善後工作。

利用此特性，我們可以提出上述問題的解決方案，藉由 `WM_ENDSESSION` 視窗訊息來進行系統關閉前的善後工作，代替平日的 `OnClose` 事件。

```
#0001 class TForm1 : public TForm
#0002 {
#0003 ...
#0004 BEGIN_MESSAGE_MAP
#0005     VCL_MESSAGE_HANDLER(WM_ENDSESSION, TWmEndSession, WmEndSession);
#0006 END_MESSAGE_MAP(TForm);
#0007 };
#0008
#0009 void __fastcall TForm1::MyCleanupRoutine()
#0010 {
#0011     SaveIni();
#0012     SaveOpenHistory();
#0013
#0014     ... // 所有的善後工作
#0015 }
#0016
#0017 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction
#0018     &Action)
#0019 {
#0020     MyCleanUpRoutine();
#0021 }
#0022
#0023 void __fastcall TForm1::WmEndSession(TWmEndSession& Msg)
#0024 {
#0025     TForm::Dispatch(&Msg);
#0026
#0027     // 若真的要關閉了，快做善後工作
#0028     if (Msg.EndSession) MyCleanUpRoutine();
#0029 }
```

說穿了很簡單，只是將所有的善後工作獨立為 `MyCleanupRoutine` 函式，並且在 `OnClose` 事件處理函式以及收到 `WM_ENDSESSION` 訊息且 `Msg.EndSession` 為 `true` 時呼叫它，就可以確保程式不論是由使用者關閉或因系統關閉而皆結束，都能夠萬無一失地儲存重要資料及清理善後了。

寫封伊媚兒

一位 SOHO 程式員的來信

寬達兄：

您好。

閉門數日，好不容易完成一套嘔心瀝血、完美無瑕的軟體，集自己十年功力之大成，作品的每一個視窗、每一個按鈕、每一張圖示，都搭配得恰到好處、無可挑剔，差點想仿效 Knuth，貼出一字百金的臭蟲懸賞公告了。將作品放上網路，丟到各個軟體介紹網站，好好地睡一覺，心想，醒來之後，大概網路上已經充斥著討論我的作品的文章，而電子郵件信箱也應該很快就塞滿仰慕者的信件...

沒想到，這一等就是幾個禮拜，明明各個軟體站台都有介紹，都提供下載 URL；明明有不少人已經把我的程式下載回去使用了，為什麼...為什麼連一封回應作者的信件都沒收到呢？到底是怎麼一回事？

左思右想，嗯，我知道了。一定是使用者都很懶，懶得自己開啓郵件程式，懶得將我的電子郵件地址鍵入，懶得打信件標題，雖然很仰慕我，但是因為大家都懶得寫封伊媚兒，所以才害我沒收到信。

怎麼辦？怎麼樣才能使我的愛慕者勤勞一點，寫信給我呢？請救救我，我的春天全靠它了。

Talk

信中的Knuth，指的是資訊科學界的大師Donald Knuth，他的中文名字叫高德納。在他所寫的*T_EX: The Program*一書的前言裡，這麼寫道：

「我相信T_EX的最後一個錯誤已在 1985 年 11 月 27 日被我發現，並且已經修正。日後只要有人發現程式還有任何錯誤，歡迎通知本人，我很樂意付給第一個發現錯誤的人 20.48 美金（這個獎金已經比去年提高一倍了，而且我願意每年提高一倍的獎金，賞給第一個找到錯誤的人）。相信大家都能明白，我對自己非常有信心」。

這行為跟咱們中國的呂不韋先生極為相似。史記呂不韋傳：「不韋乃使其客，人人著所聞集論，．．．，號曰呂氏春秋。布咸陽市門，懸千金其上，延諸侯游士賓客，有能增損一字者，予千金」。

不過，Knuth 面對的是客觀嚴酷的程式碼，呂不韋面對的是主觀認知的著作，我總覺得高先生的勇氣遠比呂大叔要高多了！:P

嗯，看起來是共享／免費軟體作者易患的典型「使用者回應缺乏癥候群」，尤其這位寂寞老兄格外嚴重，想藉著作品一炮而紅，在網路上找尋美麗的愛慕者當女友，我看除非哪天台北市有足夠的停車位，或是哪天新竹科學園區突然不塞車了，否則想藉著寫軟體找尋春天，難哪。

雖然不看好，忙還是要幫的，嗯，讓我們來研究研究如何為程式加入方便的「與作者連繫」功能。雖然無法強迫使用者寫信給他，可是至少讓步驟簡化一點，寫信的機率就會高一點，盡人事聽天命囉。

Mailto URL Scheme

編寫過 HTML 檔案的朋友一定都十分熟悉 URL (Uniform Resource Locator) 這東東，藉著短短的字串，簡單清楚地表示連結的通訊協定、動作、位址及參數等等。例如：

- `http://www.vclxx.org/cgi.html`
- `ftp://ftp.vclxx.org/pub/patch/d4patchcs.exe`
- `news://forums.inprise.com/borland.public.announce`
- `gopher://gopher.nthu.edu.tw/`

這些都是標準的 URL 字串，從冒號之前的第一個單字就可讀出此 URL 所代表的通訊協定，接著才是主機或目錄或其它含意。

由於電子郵件的便利性及普及，現在幾乎沒有找不到郵件程式或不連接 Internet 的電腦了。因此，我們可以利用 URL 簡化使用者撰寫新的電子郵件的步驟。

怎麼做呢？關鍵在於 `mailto` URL scheme。`mailto` URL 在網頁上其實十分常見，大部分都只是單純的 `mailto:someone's_email_address` 格式，其實這並不是 `mailto` URL 唯一的用法，其中大有文章，待咱慢慢瞧來。

先砸出一顆大號威而剛，噢，不不，是大補丸才對。以下是 `mailto` URL 的語法：

```
mailtoURL = "mailto:" [ to ] [ headers ]
to        = #mailbox
headers   = "?" header *( "&" header )
header    = hname "=" hvalue
hname     = *urlc
hvalue    = *urlc
```

其中 `mailbox` 規範於 RFC 822 文件，而 `hname`、`hvalue` 的定義也同樣在 RFC 822，與 `Message Header` 的定義相同。

由以上的語法，很容易就可看出，原來 `mailto` URL 也跟使用最廣泛的 HTTP URL 一樣，可以容許參數對（即「Name = Value」這樣的字串）存在。其中 `hname` 指的是信件的標頭，例如 **From**、**To**、**Date**、**Subject**、**CC**、**BCC**、**Reply-To** 等等皆屬常用的信件標頭；`hname` 也可以是 **Body**，指信件內文。

Info

若想對 `mailto` URL 有更進一步的瞭解，請自行閱讀以下的 RFC：

- RFC 822 "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES"
- RFC 1738 "Uniform Resource Locators"
- RFC 2045 ~ RFC 2049 "MIME (Multipurpose Internet Mail Extensions) Part One ~ Part Five"
- RFC 2368 "The `mailto` URL scheme"

Mailto URL 的應用

例如，可以為訂閱 C++BuilderChat 郵件討論群組的動作設置一個 `mailto` URL：

```
mailto:request@vclxx.org?body=subscribe%20News
```

將這個 URL 放在網頁上讓使用者點按即可。一旦啟動此 URL，會立刻開啓郵件程式並且自動撰寫一封新信件，收信人為 `request@vclxx.org`，內文為 `subscribeNews`。細心一點的讀者也許會發現，空白字元以“%20”代替，這即是一般的 URL 編碼法則。

範例程式裏頭列出五種應用方式：

- `mailto:kuan@ilife.cx`
寄信給 `kuan@ilife.cx`。
- `mailto:kuan@ilife.cx?to=jethro@iis.sinica.edu.tw&to=MichaelJordan@NBA.com`
同時寄信給 `kuan@ilife.cx`、`jethro@iis.sinica.edu.tw` 及 `MichaelJordan@NBA.com`。
- `mailto:kuan@ilife.cx?subject=Hello&body=C%2B%2BBuilder%20is%20great`
寄信給 `kuan@ilife.cx`，標題是 `Hello`，內文是 `C++Builder is great`。
- `mailto:kuan@ilife.cx?cc=jethro@iis.sinica.edu.tw&subject=Hi!`
寄信給 `kuan@ilife.cx`，副本送給 `jethro@iis.sinica.edu.tw`，標題為

Hi!。

- **"mailto:kuan@ilife.cx?body="** + Memo1->Lines->Text
寄信給 kuan@ilife.cx，內文是 Memo1 的內容。

範例程式中，我使用 *TLabel* 元件，調整為斜體藍字加底線的「Times New Roman」字型，讓它看起來挺像瀏覽器裏的 URL。:P 以滑鼠左鍵單擊這些 *TLabel* 元件時，程式會呼叫 *ShellExecute* API 函式來開啓 mailto URL：

```
ShellExecute(Handle, "open", URL, NULL, NULL, SW_SHOW);
```

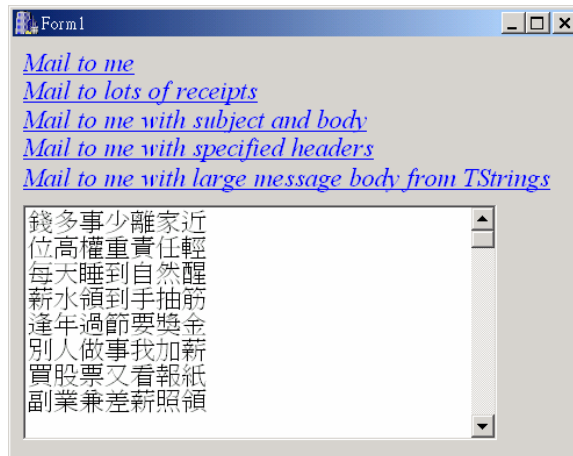


圖 10-13 / 「跟作者連繫」範例程式執行畫面

附錄



附錄 A

我的程式庫

從本書的範例程式中，你可以發現有許多範例程式使用了非 C++Builder 提供的標準單元，且單元名稱皆以小寫 x 字母開頭，背後似乎隱藏著一套功能眾多的程式庫？

這是我自己撰寫維護的程式庫，它的特點是，依功能詳細分類，且**不是類別程式庫**。

Info

要與各位讀者說聲抱歉：我的程式庫完全以 Object Pascal 撰寫。

主要是筆者使用 Delphi 的時間比 C++Builder 來得長，再加上 C++Builder 可以編譯、連結 Object Pascal 單元，省去維護兩套程式庫的麻煩。因此不論撰寫 Delphi 或 C++Builder 程式，我同樣使用這裡所介紹的程式庫。

特地耗費一番精神將 Object Pascal 原始碼轉換為 C++，不但得不到任何好處，日後的版本控制／維護也將是個大問題。因此，請見諒。

看書時，最喜歡看到作者整理出一大堆與主題相關，可直接套用在自己程式內的現成原始碼；最不喜歡看到的是，這堆原始碼通通以類別形式存在，既不好閱讀，使用起來也麻煩。

書籍程式庫的形式

以 *Secrets of Delphi 2* 這本書為例，作者 Ray Lischner 十足是位實戰經驗豐富的駭客級高手，對於任何主題，總能搬出一堆輔助使用的類別。

但是，對於只需兩三道函式，亦不必全域變數、自訂型別就能解決的工作而言，將它包裝為類別，真如同拿屠龍刀來斬雞了。書上所制訂的類別通常功能陽春，無法直接套用在實際的應用程式內，若要勉強套用就勢必得修改類別宣告及實作程式碼，那麼類別存在的原意就抵消了，倒不如只介紹精髓或技術難度最高部分的函式或程式碼片斷來得有用。

事實上，若書籍的範例程式每個皆能自我滿足，除了標準單元外，所用到的每道函式皆由程式本身提供，這種範例程式讀起來最為順暢，不必每每東翻西找只為了看看一道知名不知義的函式究竟在幹嘛，好不容易從書附光碟找到原始碼後，才發現原來它只是簡單的範例檢查函式、或是某道 API 函式的包裝，嘔死人了。

原本我是希望朝著這個方向努力，不過程式寫著寫著，才發現說來容易做來難。我才發現，對程式重用性十分注重的我，無法忍受許多程式單元塞著太多重覆程式碼的情形，真是叫人寫來直冒冷汗、直打哆嗦，自己的程式不敢再看第二遍。

於是心念一轉，開始放心地在書內範例程式加入自己平日使用的程式庫。一來程式碼無謂重覆的情況減至最低，再者順便驗證程式庫的功能及適用性。規範這麼一套完整的程式庫後，範例程式將不使用其它的外來函式，只要是程式中呼叫的非 C++Builder 標準單元提供的函式，就可以在此找到解說，盡量不增加閱讀範例程式時的困難。當然囉，既然我不愛拿倚天劍砍泥鰍，這套程式庫就不會有任何類別或物件存在，只有一道道功能簡單、分類詳明的函式，喜歡的話，直接取著用，一點也不麻煩。

下表是使用於本書內的程式庫單元，以及個別單元所支援的功能對照列表：

表 A-1 / 書附程式庫的單元及函式分類

單元名稱	函式分類
xCONTROLS	VCL 控制項的輔助操作函式。
xDARRAY	Object Pascal 雖然支援動態陣列，但程式員能使用的操作極為有限，此為動態陣列的操作補強函式。
xDESKTOP	桌面、圖示、螢幕保護程式等相關函式。
xFILES	檔名、路徑的字串處理及操作，以及執行、複製檔案等相關函式。
xFONTS	字型資訊的儲存及載入支援。
xKERNEL	除錯支援、行程列表、模組名稱等相關函式。
xMEMORY	記憶體讀寫權限測試、記憶體映射檔案等相關函式。
xREGISTRY	系統登錄資料庫的輔助操作函式。
xSTREAMS	資料流及物件永續機制支援函式。
xSTRINGS	字串處理相關函式，補 SysUtils 單元之不足。
xTIMES	時間、日期等相關函式。
xUTILS	最常用的工具函式，如顯示訊息、舉發 API 函式例外等等。
xWINDOWS	視窗處理、找尋、判斷等相關函式。

茲將程式庫各單元列表解說如下，所有原始碼皆置於書附光碟的 BCB5\UNIT 及 BCB6\UNIT 目錄。

xCONTROLS.PAS

```
procedure EnableControl(AControl: TControl; Enable: Boolean);
procedure EnableChildControls(AControl: TControl; Enable: Boolean);
procedure EnableClassControl(AControl: TControl; Enable: Boolean;
    ControlClass: TControlClass);
```

將控制項本身、控制項的子控制項以及某個特定類別的控制項致能或除能。

有時必須視使用者的選擇將某個控制項及其子控制項全部致能或除能，例如 Delphi 整合環境【Project / Options】對話盒的 Version Info 頁面，當「Include version information in

project」核取項目打勾時，同樣頁面的其它控制項皆進入致能狀態，否則為除能狀態，你可以用 *EnableChildControls* 函式輕易達成這點。而 *EnableClassControl* 函式可選擇其中的某種元件進行致能／除能動作。

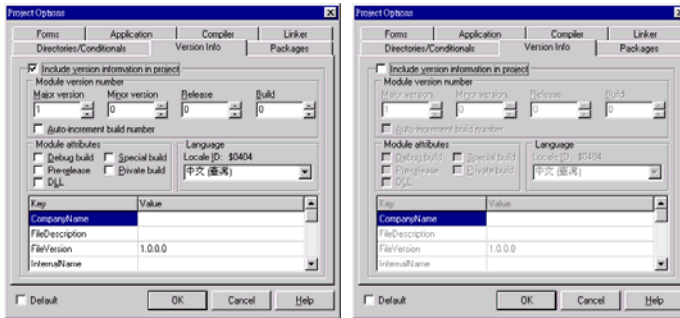


圖 A-1 / 【 Project / Options 】對話盒的 Version Info 頁面

```
procedure SelectPageIndex(const PC: TPageControl; const iIndex: Integer;
const Animated: Boolean);
```

選取 *TPageControl* 元件的某個頁面，若 *Animated* 參數為 *True*，則會從目前頁面一頁一頁地翻到目的頁面。

```
procedure MakeSurePageVisible(const PC: TPageControl);
```

若 *TPageControl* 元件的頁面可能動態隱藏或顯示時，必須小心若被隱藏的正巧是目前頁面，則 *TPageControl* 元件將陷入沒有目前頁面的尷尬情形，此時必須切換到下一個可見的頁面（即 *TTabSheet* 物件）才行。隱藏頁面後呼叫 *MakeSurePageVisible* 函式可確保 *PageControl* 遠離此困境。

```
// Filename : 儲存或載入的檔案路徑
procedure LoadTreeViewFromTextFile(Nodes: TTreeNode; Filename: string);
procedure SaveTreeViewToTextFile(Nodes: TTreeNode; Filename: string);
```

分別將 *TTreeView* 元件的節點（即 *TTreeNode* 物件）文字以類似 WinHelp CNT 的檔案

格式儲存至檔案以及從檔案讀回的函式。

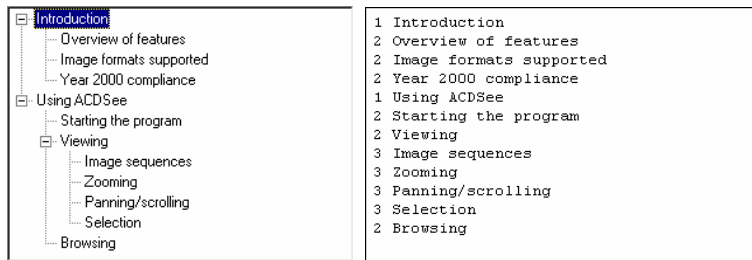


圖 A-2 / 將 TreeView 元件節點文字以仿 CNT 檔案格式儲存，左邊為 TreeView 元件，右邊為檔案內容

xDARRAY.PAS

Delphi 4 起，內建各種型別的動態陣列支援，程式員一陣狂喜，從此拋棄自行動態配置及管理記憶體之負擔。但，依我看動態陣列支援似乎只做了一半，因為竟然連刪除、搬移連續元素的函式都沒提供，使用起來總覺得它雖是能助我斬棘破路的好劍，劍柄卻只做一半，另一半師傅偷工減料沒做出來，難以全面掌握啊。

動態陣列宣告之後，只有下列幾個函式可供對其操作：

- 設定陣列大小，可任意縮減或增加陣列大小。

```
procedure SetLength(var S; NewLength: Integer);
```

- 取出陣列連續元素，複製給另一個陣列變數。

```
function Copy(S; Index, Count: Integer): array;
```

- 取得陣列大小及上下限

```
function Length(S): Integer;  
function High(X);  
function Low(X);
```

瞧，區區五個函式的支援，如果你想要刪除動態陣列的某段元素、插入一段新的元素或者將陣列的一部分複製到另一個陣列內，不管是什麼動作，上述五道函式未提供的，就只能委屈使用一個個元素分別指派的笨方法囉。

因此，絕對有必要建立程式庫來補強對於動態陣列的操作處理，以下是我研究 Delphi 動態陣列的實作方式後，自行撰寫的幾道使用起來雖不方便，但十分有用的函式。

```
// A      : 動態陣列變數
// lSize  : 元素所佔位元組
// Index  : 元素索引
// Count  : 元素數目
procedure DynArrayDelete(var A; elSize: Longint; Index, Count: Integer);
procedure DynArrayInsert(var A; elSize: Longint; Index, Count: Integer);
```

這幾道處理動態陣列的函式十分麻煩的地方就是每道函式都必須傳入 *elSize* 參數，也就是元素所佔位元組才行。原因是雖然曉得動態陣列的指標、儲存方式以及所有細節，但陣列元素的型態並未與陣列本身置於同處，而是由編譯器來記錄管理。所以對於一個動態陣列變數 *A*，編譯器可以由它本身的記錄得知此陣列元素佔用的空間，而我們不行，只能要求程式員將陣列元素所佔空間作為參數傳入，很不公平對吧。

DynArrayDelete 函式將動態陣列從索引值為 *Index* 的元素算起，刪除 *Count* 個元素，後面元件向前推進；*DynArrayInsert* 函式在 *Index* 索引後插入 *Count* 個數值為零的元素。你大概可以這樣用：

```
#0001 procedure TForm1.Button1Click(Sender: TObject);
#0002 var
#0003   A: array of char;
#0004   I: Integer;
#0005 begin
#0006   SetLength(A, 4);
#0007   // A = ('', '', '', '')
#0008   for I := 0 to 3 do
#0009     A[I] := Chr(Ord('a') + I);
#0010   // A = ('a', 'b', 'c', 'd')
#0011   DynArrayDelete(A, sizeof(char), 1, 2); // 刪除 'b' 及 'c'
#0012   // A = ('a', 'd')
#0013   DynArrayInsert(A, sizeof(char), 0, 2); // 在 'a' 之後插入兩個
#0014   // A = ('a', '', '', 'd')
#0015 end;
```

```
// ADst   : 目的動態陣列
// ASrc   : 來源動態陣列
```

```

// elSize   : 元素所佔位元組 (兩陣列相同)
// IndexDst  : 目的元素索引
// IndexSrc  : 來源元素索引
// Count    : 元素數目
procedure DynArrayCopy(var ADst; const ASrc; elSize: Longint; IndexDst,
  IndexSrc, Count: Integer);
procedure DynArrayAppend(var ADst; const ASrc; elSize: Longint; IndexSrc,
  Count: Integer);

```

DynArrayCopy 將來源陣列 *ASrc* 的一部分元素複製到目的陣列 *ADst* 上頭，注意只有複製動作，所以呼叫時務必確認目的陣列 *ADst* 的大小足夠放置來自 *ASrc* 的 *Count* 個元素。

DynArrayAppend 的動作十分類似，不同的是它將 *ASrc* 陣列的一部分接續在 *ADst* 陣列後頭，會自動為目的陣列拉長 *Count* 個元素大小。看看範例：

```

#0001 procedure TForm1.Button4Click(Sender: TObject);
#0002 var
#0003   A, B: array of Integer;
#0004   I: Integer;
#0005 begin
#0006   SetLength(A, 4);
#0007   // A = (0, 0, 0, 0)
#0008   for I := 0 to 3 do
#0009     A[I] := I;
#0010   // A = (0, 1, 2, 3)
#0011   SetLength(B, 6);
#0012   // B = (0, 0, 0, 0, 0, 0)
#0013   DynArrayCopy(B, A, sizeof(Integer), 1, 0, 4); // 從 A 拷貝至 B
#0014   // B = (0, 0, 1, 2, 3, 0)
#0015   DynArrayAppend(B, A, sizeof(Integer), 0, 4); // 將 A 接到 B
#0016   // B = (0, 0, 1, 2, 3, 0, 0, 1, 2, 3)
#0017 end;

```

對於 Delphi 動態陣列內部實作細節有興趣瞭解的讀者們，可將這裏提供的幾道函式作為敲門磚，其它的內部細節通通置於 System 單元，淨往裡頭挖，絕不會空手而回。

xDESKTOP.PAS

```

function GetDesktopDefView: HWND;
function GetDesktopListView: HWND;
function GetActiveDesktopWindow: HWND;

```

取得桌面上特殊視窗的視窗 handle。

- *GetDesktopDefView* 取得 *ProgMan* 視窗的第一個子視窗，視窗類別為 *SHELLDLL_DefView*。
- *GetDesktopListView* 取得 *DefView* 視窗的第一個子視窗，視窗類別為 *SysListView32*，即負責顯示桌面圖示的視窗。如果要對桌面進行特殊處理或訊息攔截，通常由此視窗下手。
- *GetActiveDesktopWindow* 函式取得 *ListView* 視窗的下一個視窗，視窗類別為 *Internet Explorer_Server*，若 Active Desktop 功能啟動中，就可找到此視窗，否則不會有此視窗的存在，此函式可用來判斷 Active Desktop 的存在與否。

```
procedure RebuildIconCache;
```

重建所謂的圖示快取 (icon cache)，由於圖示的重繪如此頻繁，因此 Windows 使用圖示快取機制，減低圖示重覆讀取建立的工作。此函式會強迫所有正顯示在畫面上的圖示重新載入，這個動作大概只有兩個用處：

- 有時圖示因為不知名原因 (顯示卡驅動程式或不當應用程式) 產生影像錯誤或破碎現象，重建圖示快取可修復此狀況。
- 若欲更新桌面圖示，先將新的圖示寫入系統登錄資料庫，再呼叫 *RebuildIconCache* 函式，即可強迫 shell 重新讀取圖示檔，更新桌面圖示。

xFILES.PAS

```
var  
  AppDir: string;
```

指向程式執行檔所在的目錄，以反斜線結尾。

```
function PathWithoutSlash(const Path: string; PathD: Char = '\'): string;  
function PathWithSlash(const Path: string; PathD: Char = '\'): string;
```


呼叫 *GetDir* 函式、*GetCurrentDir* 函式、*GetCurrentDirectory* API 函式取得目前工作目錄或經由 *ExtractFilePath(ExpandFileName(Application.ExeName))* 取得執行檔置放目錄時，同一個路徑至少有兩種表示法，如：

```
d:\delphi\works\mailer
d:\delphi\works\mailer\
```

雖然我們一眼可看出兩字串指向同一個目錄，但字串處理函式可不曉得，因此才有這兩道路徑格式處理函式來確認使用的路徑格式。*PathWithoutSlash* 傳回不帶最後頭反斜線的路徑，而 *PathWithSlash* 正巧相反。有了這兩支函式，隨時可確保路徑字串的格式：

```
Filename = PathWithSlash(GetCurrentDir) + 'tree.bmp';
FindFirst(PathWithoutSlash(sDir) + '*.*', faAnyFile, SearchRec);
```

Tips

比如說，同樣是 *GetCurrentDirectory* API 函式，但在微軟所宣告大一統的 Win32 平臺上（例如 Windows 95 及 Windows NT），傳回的路徑字串格式就不同，這個惱人的相容性問題就交給在此介紹的函式解決吧。

這兩個函式以及接下來所介紹的一系統檔案路徑操作函式，都帶有一個可有可無的字串參數 *PathDelimiter*，預設值為反斜線符號 \，你可以視情況傳入斜線符號 /，此參數使得這一系列函式同時支援以斜線符號及反斜線符號作為目錄分隔子元的檔案系統（例如 UN*X 及 URL 使用斜線符號，DOS/Windows 使用反斜線符號）。

```
// BaseDir    : 基底路徑
// FilePath   : 檔案的絕對路徑
function RelativePath(BaseDir, FilePath: string; PathD: Char = '\'): string;
```

根據基底路徑，將絕對路徑轉換為相對路徑：

```
RelativePath('c:\winnt', 'c:\winnt\system32\kernel32.dll');
// 傳回 'system32\kernel32.dll'
```

```
// sAction    : 動作, 如 'open', 'edit', 'play' 等等
// sFileName  : 檔名或 URL
// sPara      : 動作執行參數
function MyShellExecute(const sAction, sFileName, sPara: string): Boolean;
```

常常可在網路上的新聞討論群組見到「如何程式開啓瀏覽器，指向某個URL？」「請問要怎麼啓動預設的信件程式，並指定新信件的收信人及標題？」等等問題。事實上，執行URL的任務，交給SHELL32.DLL提供的*ShellExecute* API函式就行了¹。由於*ShellExecute*函式的呼叫稍嫌麻煩，因此才有此道*MyShellExecute*函式的包裝。

```
MyShellExecute('open', 'http://www.vclxx.org', '');
MyShellExecute('open', 'mailto:kuan@ilife.cx', '');
```

```
// Command    : 執行檔名及參數
// bWaitExecute : 是否等待程式執行結束
// bShowWindow : 是否顯示程式視窗
// PI          : 指向 TProcessInformation 結構, 用以取回執行後的行程資訊
function Execute(const Command: string; bWaitExecute: Boolean;
    bShowWindow: Boolean; PI: PProcessInformation): Boolean;
```

CreateProcess API 函式大概是 API 函式裏叫用最麻煩的一個了，它必須傳入十個參數，其中還有兩個參數指向結構。因此 *Execute* 函式只是 *CreateProcess* 函式的包裝，讓執行檔案的工作輕鬆點。若傳入的 *bWaitExecute* 參數為 *true*，函式會等到程式執行結束後才返回。若需要新行程、執行緒的編號或 *handle*，可將 *PI* 參數指向 *TProcessInformation* 結構，函式會將行程建立後的結果傳入。

例如，可在程式裏偷偷呼叫外部程式來幫忙處理：

```
// 呼叫 PKUNZIP 程式將 data.zip 解壓縮，解壓縮完成前函式不會返回
Execute('pkunzip c:\temp\data.zip', true, false, NULL);
```

```
function ExtractFileNameNoExt(Filename: string): string;
```

傳入任意檔案（可包括路徑），傳回主檔名（除去副檔名後的檔名）。

¹ 關於URL的指定方式及*ShellExecute*函式，請參考第十章「Fancy軟體撰寫手則」。

```
ExtractFileNameNoExt('c:\windows\Jethro.dat') // 傳回 'Jethro'
```

```
function MyGetFileSize(const Filename: string): DWORD;
```

傳入檔名，取得檔案大小。

```
procedure MyCopyFile(const sSrcFile, sDstFile: string);
```

CopyFile API 函式的包裝，將 *sSrcFile* 檔案複製為 *sDstFile* 檔案。

```
function TruncateTrailNumber(var S: string): Integer;  
function TruncateTrailIfNotDLL(S: string): string;  
function FileExistsAfterTruncate(Filename: string): Boolean;
```

用來處理檔案內圖示的指示字串，例如若要指向 FOO.DLL 的第二個圖示，會以 “FOO.DLL,1” 來表示（編號由零開始）。*TruncateTrailNumber* 函式會將逗號及逗號之後的編號去掉；*TruncateTrailIfNotDLL* 函式會判斷此檔案是否為 EXE、DLL、ICL 等可能帶有圖示的檔案，若是，才保留圖示編號；而 *FileExistsAfterTruncate* 則用來判斷包含此圖示的檔案是否存在。

```
function TruncateDirSpecifier(const Path: string): string;
```

在許多場合（如系統登錄資料庫、INI 檔或佈景描述檔），路徑名稱可能包含代表某些系統磁碟或目錄的特殊字串，皆以「%」字元包夾。如代表 Windows 目錄的 “%WinDir%”、代表系統目錄的 “%SysDir%” 等等。此函式用來去除這些特殊字串。

```
TruncateDirSpecifier('%WinDir%\explorer.exe');  
// 傳回 'explorer.exe'
```

```
function ComparePath(const Path1, Path2: string): Boolean;
```

比較兩個路徑，不論兩者的字母大小寫方式及路徑格式。

```
function ParentDirectory(Path: string): string;
```

傳回指定目錄的上一層目錄，若已是最高層級，則傳回本身。支援UNC命名方式²。

```
ParentDirectory('c:\windows\system\'); // 傳回 'c:\windows'
ParentDirectory('c:\'); // 傳回 'c:\'
ParentDirectory('\\\\xshadow\c\fonts'); // 傳回 '\\xshadow\c\'
ParentDirectory('\\\\xshadow\c\'); // 傳回 '\\xshadow\c\'
```

```
function SystemDirFile(const Filename: string): string;
function WindowsDirFile(const Filename: string): string;
function SystemDriveFile(const Filename: string): string;
```

傳入檔名，分別取回此檔名於系統目錄、Windows 目錄及系統磁碟根目錄下的完整路徑。

```
type
  // Filename      : 取得的檔名
  // Attr          : 檔案屬性
  // UserData      : 呼叫 EnumDirectoryFiles 時傳入的整數值
  // bContinue     : 是否繼續列舉
  TEnumDirectoryFileProc = procedure (Filename: string; Attr: Integer;
    UserData: Integer; var bContinue: Boolean) of object;
  // sDir          : 欲列舉檔案的路徑
  // sMask         : 欲列舉的檔名遮罩
  // Attr         : 欲列舉的檔案屬性
  // bRecursive   : 是否列舉子目錄內的檔案
  // UserData     : 自行定義的使用者資料
  // EnumDirectoryFileProc : 回呼函式
  procedure EnumDirectoryFiles (sDir, sMask: string; Attr: Integer;
    bRecursive: Boolean; UserData: Integer;
    EnumDirectoryFileProc: TEnumDirectoryFileProc);
```

透過回呼函式的輔助，可快速地針對某個目錄下的所有或部分檔案進行列舉，並可在回呼函式中決定是否繼續列舉或者中斷列舉動作。由於採用遞迴呼叫，所以包括子目錄下的所有檔案也一併列舉。

² UNC, Universal Naming Convention, 可跨網路直接指向另一部電腦的某一個檔案，如 “\\MyWorkstation\Samples\Northwind.mdb” 此類的定位方式。

```
procedure TForm1.EnumFileProc (Filename: string; Attr: Integer;
  UserData: Integer; var bContinue: Boolean);
begin
  ListBox1.Items.Add(Filename); // 將檔名加入列示盒
  bContinue := true; // 繼續列舉
end;

// 列舉 c:\delphi 目錄下所有執行檔
EnumDirectoryFiles('c:\delphi', '*.exe', faArchive, EnumFileProc);
```

```
procedure CleanDirectory(sDir: string);
```

將 *sDir* 目錄下所有目錄、檔案悉數刪除。

```
procedure CopyDirectory(sDir, tDir: string; bRecursive: Boolean);
```

複製目錄，將 *sDir* 目錄所有檔案拷貝至 *tDir* 目錄，若 *bRecursive* 參數為 *true*，則連同子目錄一併複製。

```
function GetUniqueFileName(const Path: string; Filename: string): string;
```

傳入路徑及嘗試使用的檔名，傳回保證不與現有檔案衝突的檔名。

```
GetUniqueFileName('c:\delphi\temp', 'movie.dat');
// 假設目前沒有同名檔案，傳回 movie.dat
GetUniqueFileName('c:\delphi\temp', 'movie.dat');
// 產生衝突，傳回 movie1.dat
GetUniqueFileName('c:\delphi\temp', 'movie.dat');
// 產生衝突，傳回 movie2.dat
```

```
function GetTemporaryFileName: string;
```

傳回位於系統臨時目錄³下的臨時檔名，保證不與現有檔案衝突。

³ 系統臨時目錄由TEMP環境變數所指定。

```
function GetSystemPath: string;  
function GetWindowsPath: string;
```

取得系統及 Windows 目錄。

```
function GetRootDir(var sPath: string): string;
```

取得指定路徑的磁碟代號，*sPath* 參數將成為失去磁碟代碼的路徑字串。

```
function GetLeafDir(var sPath: string): string;
```

取得指定路徑的最下層目錄名稱，*sPath* 參數將成為指定路徑的上一層。

```
S := 'c:\windows\media';  
GetLeafDir(S); // 傳回 'media'  
// S 變成 'c:\windows\'
```

xFONTS.PAS

```
procedure StringToFont(sFont: string; Font: TFont;  
  bIncludeColor: Boolean = True);  
function FontToString(Font: TFont; bIncludeColor: Boolean = True): string;
```

將 *TFont* 字型物件與字串兩者互相轉換的函式，便於將選用的字型資訊寫入系統登錄或 INI 檔。 *bIncludeColor* 參數指示轉換時是否包含字型顏色。

```
FontToString(Form1.Font);  
// 傳回值會是 'MS Sans Serif', 8, [], [clWindowText]' 這種格式  
FontToString(Form1.Font, False); // 不包含字型顏色  
// 傳回值會是 '新細明體', 11, [fsBold]' 這種格式
```

xGRAPHICS.PAS

```
function ColorToRGBString(AColor: TColor): string;
function RGBStringToColor(RGBStr: string): TColor;
```

將紅、綠、藍三原色的十進位字串表示與 *TColor* 型別互相轉換，用於處理系統登錄及佈景描述檔等等。

```
ColorToRGBString(Canvas.Brush.Color);
// 傳回 'xxx yyy zzz' 格式的字串，其中 xxx, yyy, zzz 為 0 ~ 255 的整數

RGBStringToColor('255 0 0'); // 傳回紅色
RGBStringToColor('128 0 128'); // 傳回紫色
RGBStringToColor('255 255 255'); // 傳回白色
```

```
function MyLoadIcon(const Filename: string; DefaultIcon: HICON = 0): HICON;
```

傳入圖示檔名或圖示資源表示字串，取回包含該圖示的 *HICON*。

```
MyLoadIcon('ultima.ico'); // 傳回 ultima.ico 包含的圖示
MyLoadIcon('c:\winnt\system32\shell32.dll,15');
// 傳回 shell32.dll 包含的第 16 個圖示
```

```
function CursorToIcon(const hc: HCURSOR; bDestroyCursor: Boolean = False):
HICON;
```

傳入滑鼠指標 *hc*（型別為 *HCURSOR*），將此指標圖案轉為圖示，傳回圖示的 *HICON*。若 *bDestroyCursor* 參數為 *true*，則同時將 *hc* 滑鼠指標摧毀。

```
CursorToIcon(LoadCursor(hInstance, IDC_WAIT), True); // 傳回沙漏指標的圖示
```

xKERNEL.PAS

```
procedure DebugStr(S: string);
procedure DebugStrFmt(const Format: string; const Args: array of const);
```

這兩個函式是 *OutputDebugString* API 函式的包裝，可以更輕鬆地輸出除錯訊息。若程式

在 Delphi / C++Builder 整合環境內執行，除錯訊息由 Event Log 視窗攔截；若直接由 shell 執行，則除錯訊息將由系統除錯器攔截（如果系統除錯器存在）。*DebugStrFmt* 函式的兩個參數與 *SysUtils::Format* 函式完全相同，只差在它會直接將結果字串輸出至除錯器，且會自動換行。

```
DebugStr('Server successfully started');
DebugStrFmt('%d %s', [ResponseNo, ReceiveMessage]);
```

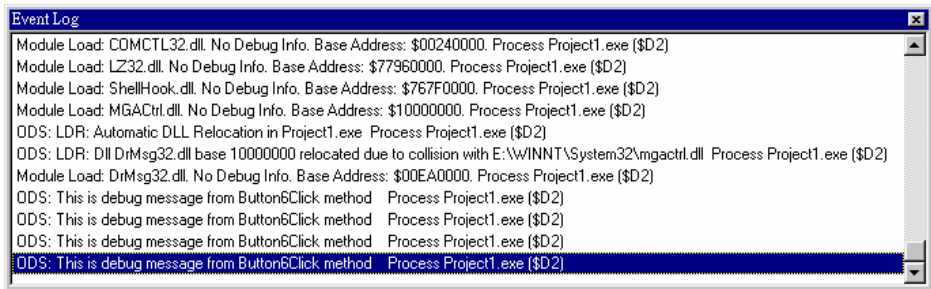


圖 A-3 / Event Log 視窗，最後四行由 *DebugStr* 函式輸出。開頭的 ODS 為 *OutputDebugString* 的頭字語

```
function GetWindowModuleFileName(Wnd: HWND): string;
```

從指定的視窗 handle 取得建立此視窗的模組檔名，SDK 未提供如此功能的函式，必須自行列舉行程資訊來取得。

```
GetWindowModuleFileName(Form1.Handle);
// 傳回執行檔名，如'project1.exe'
```

xMEMORY.PAS

```
// P : 指向某段記憶體開頭的指標
// Size : 欲測試的記憶體範圍大小
function IsMemAddressReadable(P: Pointer; Size: DWORD = 4): Boolean;
function IsMemAddressWritable(P: Pointer; Size: DWORD = 4): Boolean;
```

測試某段記憶體範圍是否可讀、可寫，可用於偵測可能出錯的指標值。

```
procedure TryReadMemAddress(P: Pointer; Size: DWORD = 4);
```



```
procedure TryWriteMemAddress(P: Pointer; Size: DWORD = 4);
```

與前兩個函式功能相同，也是測試某段記憶體範圍是否可讀或可寫，若結果是「不可以哦」，將舉發 *EAccessViolation* 例外。

```
// MapName : 記憶體映射名稱
// Size    : 記憶體映射範圍大小
// Ptr     : 取回映射記憶體指標
// 傳回值  : file mapping handle
function MapGlobalData(const MapName: string; Size: Integer;
  var Ptr: Pointer): THandle;
// Handle  : file mapping handle
// Ptr     : 映射記憶體指標
procedure ReleaseGlobalData(Handle: THandle; var Ptr: Pointer);
function IsGlobalDataExistent(const MapName: string): Boolean;
```

建立及釋放記憶體映射區塊，可用來快速建立可被所有行程、模組分享的全域資料結構。

傳入記憶體映射名稱，*IsGlobalDataExistent* 函式可用來判斷指定的記憶體映射物件是否已經建立。

```
var
  GlobalPointer: ^TSharedStruct;
  hMap: THandle;

hMap := MapGlobalData('MAPNAME', sizeof(TSharedStruct),
  Pointer(GlobalPointer));
// 與其它模組共享 GlobalPointer 指向的 TSharedStruct 結構
// 可任意讀、寫此結構資料
ReleaseGlobalData(hMap, Pointer(GlobalPointer));
```

xREGISTRY.PAS

```
function ReadKeyDefaultValue(REG: TRegistry; const sKey: string): string;
procedure WriteKeyDefaultValue(REG: TRegistry; const sKey, sValue: string);
```

TRegistry 類別在操作上比起 *TRegIniFile* 或 *TRegistryIniFile* 等類別較為不便，若只需簡單地讀取或寫入某機碼下的預設值時，使用這兩道函式可以省下開啓、關閉機碼的函式呼叫。

xSTREAMS.PAS

```
procedure SavePropertyToStream(Stream: TStream; Instance: TPersistent;  
  PropName: string);  
procedure LoadPropertyFromStream(Stream: TStream; Instance: TPersistent);
```

SavePropertyToStream 函式將 *Instance* 物件的 *PropName* 屬性寫入 *Stream* 資料流，此屬性必須是公開屬性；*LoadPropertyFromStream* 函式則將屬性由資料流讀出，兩者必須搭配使用。

藉著 VCL 提供的元件永續機制，這兩支函式以類似 DFM 檔案格式來記錄元件的屬性資料，十分適合擔任字型資訊、節點資料或元件特性的管理儲存函式。例如，下列程式碼即可十分方便地將 *TreeView1* 元件的字型設定及節點資訊寫入 TREEVIEW.DAT 檔案：

```
var  
  Stream: TStream;  
begin  
  Stream := TFileStream.Create('treeview.dat', fmCreate or fmOpenWrite);  
  with Stream do  
    try  
      SavePropertyToStream(Stream, TreeView1, 'Font'); // 寫入字型設定  
      SavePropertyToStream(Stream, TreeView1, 'Nodes'); // 寫入節點  
    finally  
      Free;  
    end;
```

這兩支函式可與任何資料流處理函式合作使用，絲毫不會干擾資料流裏其它資料的處理。

```
procedure ReadFormAsText(AExeName, AClassName: string; Strings: TStrings);  
procedure WriteFormAsBinary(AExeName, AClassName: string;  
  Strings: TStrings);
```

連結 VCL 應用程式時，會將可能使用的 DFM 資料放入執行檔的 *RC_DATA* 資源區段，資源名稱爲 *form* 的類別名稱。*ReadFormAsText* 函式由 *AExeName* 執行檔讀出類別名稱爲 *AClassName* 的 *form*，並將其文字表示寫入 *Strings* 物件。而 *WriteFormAsBinary* 函式則進行相反的動作，將以文字表示的 *form* 寫回執行檔的資源區段。

Info

不過，由於作業系統的支援度問題，*WriteFormAsBinary* 函式只在 Windows NT 上生效，Windows 95/98 只能讀出資源資料，無法寫回。

```
function VisualizeForm(Strings: TStrings): TForm;
procedure TextizeForm(AForm: TForm; Strings: TStrings);
```

VisualizeForm 函式可將 form 的文字表示轉為活生生的 form 物件；相反的，*TextizeForm* 函式可由 form 物件取得其文字表示。

xSTRINGS.PAS

```
const
  DEFAULT_DELIMITERS = [' ', #9, #10, #13]; // 預設分隔字元

// S          : 欲分析的字串
// Index      : 欲取得的 token 編號
// bTrail     : 是否連同後頭的 token 一併取回
// Delimiters  : 分隔字元集合
function GetToken(const S: string; Index: Integer; bTrail: Boolean = False;
  Delimiters: TSysCharSet = DEFAULT_DELIMITERS): string;
function CountWords(S: string; Delimiters: TSysCharSet =
  DEFAULT_DELIMITERS): Integer;
```

GetToken 是個十分強力、有用的字串分析函式，只要是固定使用某些分隔字元來做間隔的字串，都可以透過 *GetToken* 函式將其一一分解取得。*CountWords* 函式傳回指定字串總共包含幾個 token。

```
S := 'I am a boy.';
GetToken(S, 1); // 傳回 'I'
GetToken(S, 4); // 傳回 'boy.'
GetToken(S, 2, True); // 傳回 'am a boy.'
CountWords(S); // 傳回 4
S := 'You said: I am a good boy.';
GetToken(S, 1, False, [':']); // 傳回 'You said'
GetToken(S, 2, False, [':']); // 傳回 ' I am a good boy.'
```

```
CountWords(S); // 傳回 7
CountWords(S, [':']); // 傳回 2
```

```
procedure TruncateCRLF(var S: string);
function IsContainingCRLF(const S: string): Boolean;
```

去除字串結尾的換行字元以及判斷字串結尾是否為換行字元。

```
// S           : 欲處理的字串
// Token       : 將被取代的 token
// NewToken    : 取代別人的新 token
// bCaseSenitive : 是否區分大小寫
function ReplaceString(var S: string; const Token, NewToken: string;
    bCaseSenitive: Boolean): Boolean;
```

將字串中的某些字串以別的字串取代，不限字串的出現／取代次數。

```
S := 'Mary said that John likes Mary.';
ReplaceString(S, 'Mary', 'Jennifer', True);
// S 變成 'Jennifer said that John likes Jennifer.'
```

```
// S           : 欲處理的字串
// SubStr      : 取代別人的新字串
// Index       : 被取代的字元起始位置
// Count       : 被取代的字元數目
procedure Simple_ReplaceString(var S: string; const Substr: string;
    Index, Count: Integer);
```

這是簡易版的字串替代函式，必須指定欲取代的字元起始位置及字元數目，以另一個字串來取代這段字串。

```
S := '1234567';
Simple_ReplaceString(S, 'Piggy', 2, 4);
// 原字串有 4 個字元被取代，至於用幾個字元來取代則視那個用來取代別人的字串的長度來決定
// S 變成 '1Piggy67'
```

```
// S           : 欲處理的字串
// Delimiter   : 分隔字串
// Remove      : 是否去除第一個 token
function FirstToken(var S: string; const Delimiter: string;
    Remove: Boolean): string;
```

依照 *Delimiter* 字串來分隔字串 *S*，傳回 *S* 的第一個 *token*，若 *Remove* 參數為 *True*，則刪去 *S* 的第一個 *token* 與其後的分隔字元。

```
S := 'It's morning now.';
FirstToken(S, ' ', True); // 傳回 'It's', S 變成 'morning now.'
FirstToken(S, ' ', True); // 傳回 'morning', S 變成 'now.'
FirstToken(S, ' ', True); // 傳回 'now.', S 變成 ''
```

```
function AddTimeStamp(const S: string): string;
```

為字串 *S* 加上時間戳記，時間戳記由 *DateTimeToStr* 函式取得，便於除錯及事件記錄。

```
// SL      : 欲尋訪的 TStrings 物件
// S      : 欲找尋的字串
// StartIndex : 開始尋訪的項目編號
// bForward  : 往後找或往前找
function PartialIndexOf(SL: TStrings; S: string; StartIndex: Integer;
  bForward: Boolean): Integer;
```

尋訪 *TStrings* 的每個項目，只要項目文字與字串 *S* 有部分相同即傳回該項目編號。

```
function CompositeStrings(SL: TStrings; const Delimiter: string): string;
```

將 *TStrings* 的每個項目文字接續起來，兩兩以 *Delimiter* 字串間隔。

```
function SafeLoadStrings(SL: TStrings; const Filename: string): Boolean;
procedure SafeSaveStrings(SL: TStrings; const Filename: string);
```

強迫載入或儲存 *TStrings* 物件的文字，在仍有可能完成任務的前提下，不達成目的函式就不返回。

```
procedure RemoveDuplicates(SL: TStrings);
```

刪除 *TStrings* 物件中字串重覆的項目。

```
function ParseRPLNo(var Msg: string): Integer;
```

有許多的TCP上層協定，當客戶端向伺服器提出一個命令請求時，伺服器會以三碼數字做為每道回應的開端，讓客戶端程式便於分析請求的執行結果。處理信件傳遞的SMTP⁴就是一例：

```
#6 [6:17am] cs24)~% telnet mbox smtp
Trying 140.114.87.20...
Connected to dns.
Escape character is '^]'.
220 dns.cs.nthu.edu.tw ESMTP Sendmail 8.9.3/8.9.3; Wed, 27 Oct 1999
    06:17:49 +0800 (CST)
HELP
214-This is Sendmail version 8.9.3
214-Topics:
214-  HELO    EHLO    MAIL    RCPT    DATA
214-  RSET    NOOP    QUIT    HELP    VRFY
214-  EXPN    VERB    ETRN    DSN
214-For more info use "HELP <topic>".
214-To report bugs in the implementation send email to
214-  sendmail-bugs@sendmail.org.
214-For local information send email to Postmaster at your site.
214 End of HELP info
QUIT
221 dns.cs.nthu.edu.tw closing connection
Connection closed by foreign host.
```

你可以看到伺服器的每行回應前頭皆帶著三個數字字元，當然囉，若要曉得每個字元所代表的涵意，請參考規範SMTP的RFC 821。不過這不是正題，*ParseRPLNo* 函式的設計由此而來，實作這些帶有回應碼的TCP上層協定時，就可輕鬆地將前頭的狀態代碼分離出來，交給狀態碼判斷程式過濾處理。函式回返後，*Msg* 參數的狀態碼會被刪除，方便後續處理。

```
S := Recv; // 假設讀入 '214-This is Sendmail version 8.9.3' 字串
ParseRPLNo(S); // 傳回整數 214
```

⁴ Simple Mail Transfer Protocol，雖然名為Simple，雖然是西元1982年就設計出來的協定，但目前為止全世界Internet上的電子郵件仍然全依賴著它傳遞。

```
// S 變成 '-This is Sendmail version 8.9.3'
```

```
function RPos(const C: Char; const S: string): Integer;
function AnsiPos(const Substr, S: string): Integer;
```

RPos 功能同 *StrRScan* 函式，不過是為 *AnsiString* 型別制定的版本。*AnsiPos* 則是不區分大小寫的 *AnsiPos* 函式。

```
// S      : 找尋 token 的字串
// SubS   : 欲找尋的 token
// Options : 字串比對選項
function MatchString(S, SubS: string; Options: TFindOptions): Integer;
```

TFindOptions 集合型態是從 *TFindDialog* 類別借過來用的，事實上，本函式只用到其中的 *frWholeWord* 及 *frMatchCase* 兩個成員。若 *Options* 包含 *frWholeWord*，表示必須全字符合；若包含 *frMatchCase*，表示區分字母大小寫。若依照規則成功地在字串 *S* 內找到 *SubS*，則傳回 *SubS* 出現在字串 *S* 的位置，否則傳回零。

```
MatchString('I am a good boy', 'Goo', []); // 傳回 8
MatchString('I am a good boy', 'Goo', [frWholeWord]); // 傳回 0
MatchString('I am a good boy', 'Goo', [frMatchCase]); // 傳回 0
```

xTIMES.PAS

```
function TimeT_To_DateTime(TimeT: Longint): TDateTime;
```

在 C 的世界裏，標準的時間儲存格式是 *time_t*；在 VCL 的世界中，*TDateTime* 好用至極。不過兩者的儲存方式天差地遠，這兒提供由 *time_t* 型別轉換至 *TDateTime* 型別的函式。

```
function TimeToSecond(const H, M, S: Integer): Integer;
procedure SecondToTime(const secs: Integer; var H, M, S: Word);
```

將秒數與時、分、秒互相轉換的函式。

xUTILS.PAS

```
procedure MsgBox(const Msg: string);  
procedure ErrMsg(const Msg: string);  
function YesNoBox(const Msg: string): Boolean;  
function YesNoCancelBox(const Msg: string): Integer;
```

這幾道是最常用的函式，分別秀出一般訊息盒、錯誤訊息盒、選擇「是」、「否」訊息盒以及選擇「是」、「否」、「取消」的訊息盒。

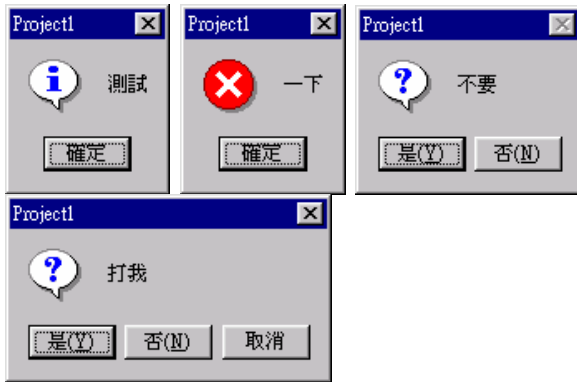


圖 A-4 / 四種常用的訊息盒。

```
procedure DoBusy(Busy: Boolean);
```

進行必須讓使用者暫時等待、無法操作的長時間動作前，先呼叫 *DoBusy* 函式，傳入 *True*，可使滑鼠指標換成忙碌中指標（通常是沙漏）；待動作完成後再呼叫一次，傳入 *False* 來還原。

```
DoBusy(True);  
try  
    // 進行花費較久的工作...  
finally  
    DoBusy(False);  
end;
```

```
procedure ShowLastError(const Msg: string = 'API Error');
procedure RaiseLastError(const Msg: string = 'API Error');
```

呼叫 API 函式時，若傳回值表示發生錯誤，即可叫用 *ShowLastError* 顯示錯誤訊息，或呼叫 *RaiseLastError* 舉發例外。舉個例子好了：

```
if not CloseWindow(123456) then // 嘗試關閉不存在的視窗
  ShowLastError; // 畫面顯示：「API Error: 無效的視窗 handle」
```

```
procedure FreeStringsObjects(SL: TStrings);
```

我時常會將 *TStrings.Objects* 屬性的個別項目填進指標(呼叫 *TStrings.AddObject* 新增項目時將指標強迫轉型為 *TObject*)，每個指標皆指向各自的資料結構。*FreeStringsObjects* 函式正是用來收拾殘局的清潔工，它會尋訪 *TStrings* 物件 *Objects* 屬性的每個項目，若其值不為 *nil*，表示指向一塊記憶體，就呼叫 *Dispose* 函式歸還記憶體。

xWINDOWS.PAS

```
const
  DEFAULT_FOCUSRECT_WIDTH = 3; // 預設的視窗虛線外框寬度

function FindControl(Handle: HWND; ProcessID: DWORD = 0): TWinControl;
```

系統中有許多視窗存在，有些屬於 VCL 元件，有些則來自 MFC、OWL 等其它 application framework，有些則直接呼叫 API 建立。傳入視窗 handle 及行程編號（這表示不限於行程本身的視窗），*FindControl* 函式將會傳回擁有此視窗的 *TWinControl* 元件，若此視窗不屬於 VCL 元件，傳回值為 *nil*。

```
function IsDelphiHandle(Handle: HWND; ProcessID: DWORD = 0): Boolean;
```

與 *FindControl* 函式類似，不過它的傳回值是布林變數，表示指定視窗是否為 VCL 元件。

```
// ParentWnd : 欲搜尋最上層子視窗的視窗 handle  
// Pt       : 相對於該視窗客戶端區域的座標  
function FindTopmostWindow(ParentWnd: HWND; Pt: TPoint): HWND;
```

傳入視窗及座標，*FindTopmostWindow* 函式將傳回位於該座標上，最上層的可視子視窗。

```
procedure MyDrawFocusRect(DC: HDC; R: TRect; Width: Integer =  
    DEFAULT_FOCUSRECT_WIDTH);  
procedure DrawWindowFocusRect(Wnd: HWND; Width: Integer =  
    DEFAULT_FOCUSRECT_WIDTH);  
procedure DrawControlFocusRect(Control: TControl; Width: Integer =  
    DEFAULT_FOCUSRECT_WIDTH);
```

用來為視窗或 *TControl* 元件繪製寬度為 *Width* 的虛線外框，此虛線外框以 XOR 方式和原有背景結合，所以再呼叫一次便可將虛線外框拭去，很適合於快速移動的焦點視窗選定介面。

附錄 B

我的工具箱

面對越趨複雜的電腦系統，程式員越是無法不藉助工具程式而徹底地瞭解底層的作業系統及各式各樣的軟硬體。如同偵探風衣口袋中的放大鏡，程式設計師也總有一個裝滿各式利器的工具箱，才能從容地面對未知的挑戰。

工欲善其事，必先利其器，話不多說，亮出我的工具箱，請多多指教。

檔案分析／解譯

DUMPBIN

前些時日收到一位朋友的來信，主題為“Pretty Park.exe”，沒有內文，但夾帶一個名為 Pretty Park.exe 的執行檔。對於這類執行檔夾帶信件十分敏感的我，雖然知道寄信人是交情還不錯的朋友，想了一會兒，還是決定打開工具箱，選一把稱手的檔案剖析工具，檢查一下這個檔案：

```
C:\TEMP>DUMPBIN /imports "pretty park.exe"

Microsoft (R) COFF Binary File Dumper Version 5.00.7022
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file pretty park.exe
```

```
File Type: EXECUTABLE IMAGE

Section contains the following Imports

user32.dll
    0 PostMessageA
    0 MessageBoxA

kernel32.dll
    0 RemoveDirectoryA
    0 MoveFileA
    0 GetProcAddress
    0 GetModuleHandleA
    ...

wsock32.dll
    0 socket
    0 sendto
    0 send
    0 recvfrom
    0 recv
    0 connect
    0 closesocket
    ...

SHELL32.DLL
    0 ShellExecuteA
```

嘿嘿，果然被我抓到了。雖然“Pretty Park”這名稱聽起來像是很漂亮、很好玩的桌面程式，不過竟然完全沒有使用 GDI32.DLL，這表示它根本沒有視窗及繪圖動作；更扯的是，它使用了 KERNEL32.DLL 提供的 *RemoveDirectoryA* API 函式，這個函式可用來刪除目錄，太可怕了！另外，它還使用 WSOCK32.DLL 提供的 WinSock API 函式，看來企圖想做什麼網路連線及傳遞資料的動作。

我想這一定是該程式自動寄出的信件，於是我立刻發出信件通知那位絲毫不知情的朋友，可憐的他，只能一一地跟收到這封信的朋友們道歉了。

這就是擁有工具、善用工具的優勢，當別人只能以訛傳訛地談論傳遞著病毒、特洛伊木馬、軟體臭蟲的信件等等謠言或消息時，深諳系統及底層技術的程式設計師們卻能拿出

工具，藉著鐵證如山的分析及追蹤過程，證明消息的虛實，事情的真相，不但能為自己避掉災禍難題，還可用以輔助解決平日期式設計上的大小問題。

上列的檔案傾印內容是以 DUMPBIN 來做到的，將 Pretty Park.exe 的 import table 列出，就可看到所有被 Pretty Park.exe implicitly linked 的 DLL 函式。

- 工具名稱：DUMPBIN
- 發行方式：包含於開發套件
- 發行公司或作者：Microsoft
- URL：http://www.microsoft.com/
- 用途：Win32 PE 格式檔案剖析

除了 DUMPBIN，另外還有兩套常見的檔案剖析工具，分別是 TDUMP 及 PEDUMP。

TDUMP可見於Borland C++Builder/C++/C++Builder開發工具；PEDUMP為Windows系統核心挖掘大師Matt Pietrek所撰寫¹，有趣的是，Matt Pietrek也曾任Borland TDUMP工具開發小組的一員。這些常見的檔案剖析工具及能力列表如下：

表 B-1 / 常見的檔案剖析工具能力比較

支援型態	DUMPBIN	TDUMP	PEDUMP
MZ 檔案 (DOS)		○	
NE 檔案 (Win16)		○	
PE 檔案 (Win32)	○	○	○
除錯資訊	○	○	○
反組譯	○		
OBJ	○	○	○
資源	○	○	○

¹ 你可在Matt Pietrek的*Windows 95 System Programming SECRETS*書內找到此程式。

關於 Pretty Park 特洛伊木馬

最近你如果有收到一封信，標題為“c:*****”，信件內容為“TEST: Pretty Park.exe”，並且附帶一個名為 Pretty Park.exe 的檔案，請切勿執行該檔案，它是有問題的特洛伊木馬！

它會將你的連線帳號、密碼、ICQ、郵件等私人資料傳送至作者手中，進而控制你的電腦及使用你的個人資料等等。若你是微軟 Outlook 郵件軟體的使用者，當你執行 Pretty Park.EXE 時，它會偷偷為你將夾帶 Pretty Park.EXE 的信件寄給通訊錄裡的所有朋友，引發連鎖效應。

如果你曾執行過它，請尋找電腦裡是否有 FILES32.VXD 這個檔，如果有的話，恭喜你，你中毒了。解除方法為，執行登錄編輯器，打開 HKEY_LOCAL_MACHINE\Software\Classes\exefile\shell\open\command 機碼，可以找到“FILES32.VXD %1" %*”字串值，刪除它後，重新啟動電腦，再刪除系統目錄下的 FILES32.VXD 檔案，最後記得將始作俑者—Pretty Park.EXE 刪除就行了。

W32Dasm

有時候，當我很好奇某某軟體是如何做到某項功能時，我會直接拿出 W32Dasm 反組譯程式，將執行檔反組譯為組合語言來研究。

幾十萬行的組合語言當然很難看得懂，幸好 W32Dasm 會將反組譯的結果加上詳細的呼叫／參考說明，並且做適當的排版及分段，看起來還不算太辛苦。若真的無法以人腦來運行一大堆組合語言時，也可將它當成除錯器來使用。W32Dasm 具備基本的除錯功能，可讓你直接 attach 運行中的程式，也可以啟動新的行程來進行除錯。

- 工具名稱：W32Dasm
- 發行方式：共享軟體
- 發行公司或作者：URSoft

- URL : <http://www.expage.com/page/w32dasm/>
- 用途 : Win32 反組譯 / 除錯器

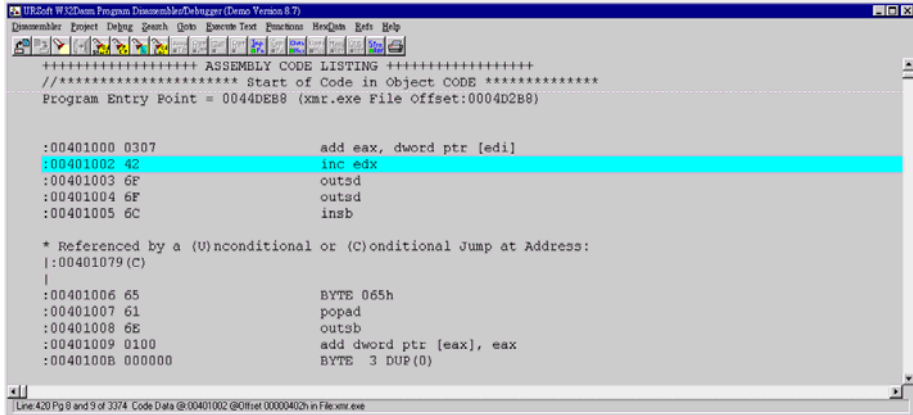


圖 B-2 / 完整地反組譯 Win32 執行檔，且加上詳細說明

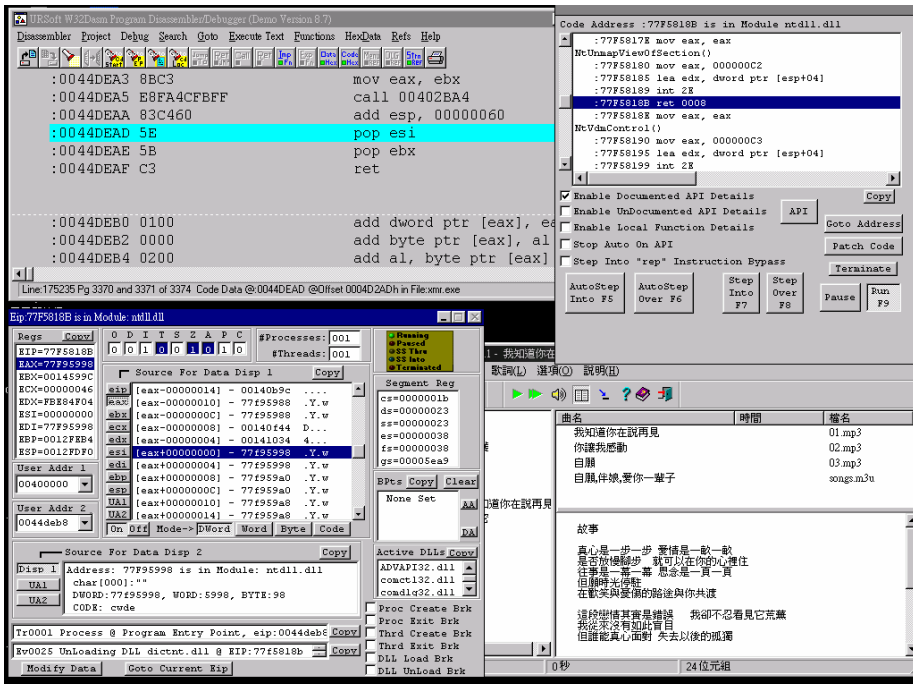


圖 B-3 / W32Dasm 也是除錯器，可在組合語言層級除錯 Win32 程式

行程／視窗行為刺探

Spy++

Spy++ 是我最常用的視窗刺探工具，我通常用它來研究某個軟體的視窗介面配置，或者瞭解特定視窗的訊息傳遞流程。

另外，若要瞭解某某軟體是哪套開發工具的作品時，觀察程式中的視窗類別名稱可說是最快的方法。若程式以 MFC 撰寫，視窗類別名稱通常會包含“Afx”字串；若程式以 VCL 撰寫，該行程通常會擁有一個 *TApplication* 視窗，且大部分的視窗類別名稱以字母“T”開頭。

- 工具名稱：Spy++
- 發行方式：包含於開發套件
- 發行公司或作者：Microsoft
- URL：<http://www.microsoft.com/>
- 用途：系統監看，包括行程、執行緒、視窗及視窗訊息

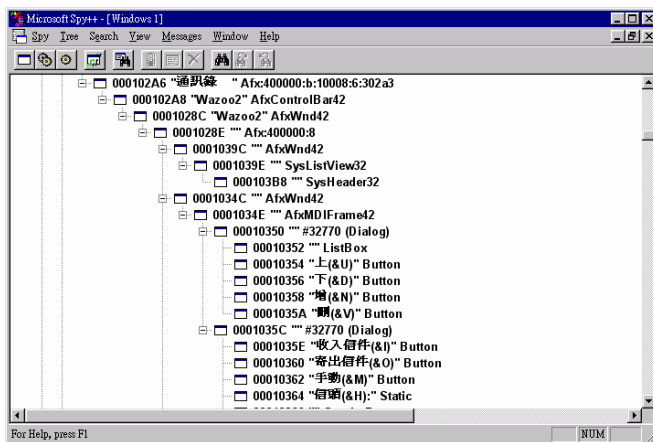


圖 B-4 / 查看系統所有視窗階層列表

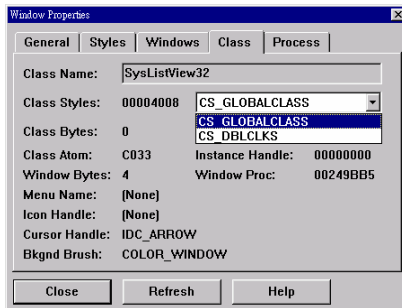


圖 B-5 / 檢視背景視窗屬性

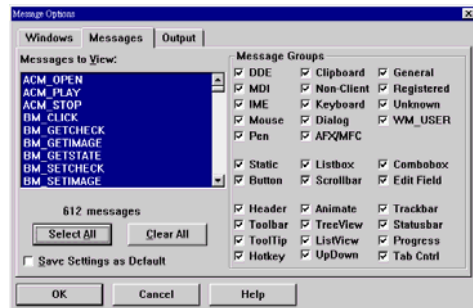


圖 B-6 / 選擇想要監看的視窗訊息種類

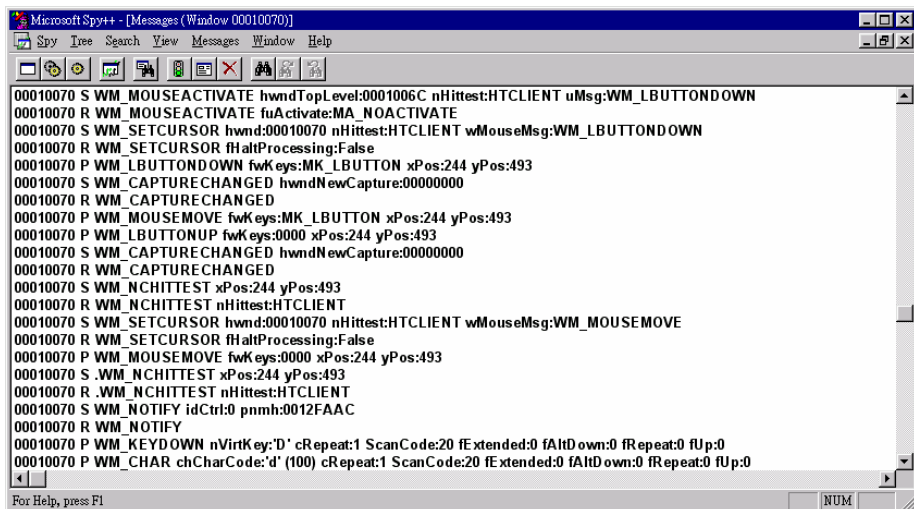


圖 B-7 / 記錄傳送 (send 或 post) 給特定視窗的視窗訊息

BoundsChecker

對於某些軟體獨有的「神奇功能」，即使翻遍 Win32 技術文件也找不到正確的做法，那時，我就會拿出 BoundsChecker。在 BoundsChecker 中啟動該程式，將程式的所有 API 呼叫行為，包括參數、傳回值等等，全部記錄下來。接著再慢慢瀏覽這些呼叫歷程，看看它到底呼叫了哪些 DLL 的哪些函式，參數如何，次序如何等等，通常可以很快地找出該「神奇功能」的正確做法，將別人的技術很快地化為自己的知識（這讓我想起「化功大法」:p）。

- 工具名稱：BoundsChecker
- 發行方式：商業軟體
- 發行公司或作者：NuMega
- URL：http://www.numega.com/
- 用途：檢查程式錯誤、分析程式行爲

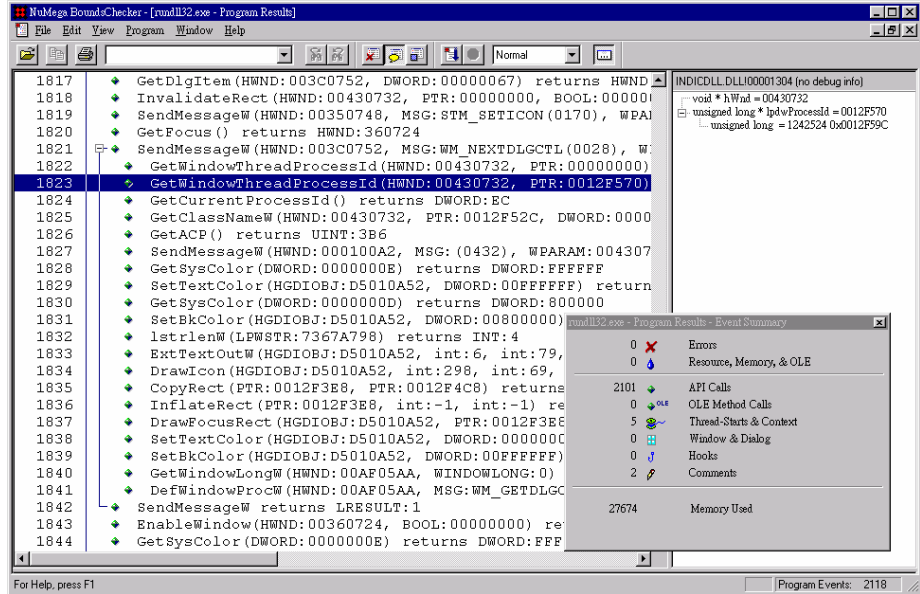


圖 B-8 / 記錄應用程式的 API 呼叫歷程，並以清楚的階層方式呈現

APISPY32

APISPY32 的能力與上述的 BoundsChecker 類似，也可記錄程式的 API 呼叫歷程。

- 工具名稱：APISPY32
- 發行方式：隨書（*Windows 95 System Programming SECRETS*）附送
- 發行公司或作者：Matt Pietrek
- URL：71773.362@compuserve.com
- 用途：刺探其它程式的 API 呼叫行爲

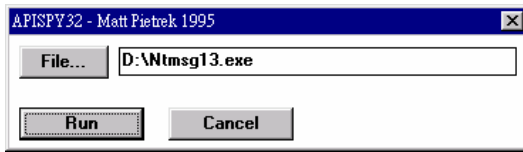


圖 B-9 / APISPY32 的載入程式

舉個例子。某天我在網路上取得一個叫做 NT messenger 的應用程式，能夠傳送訊息給網路上的其它電腦，類似 Windows 95/98 中的 WinPopup 程式，如下圖。

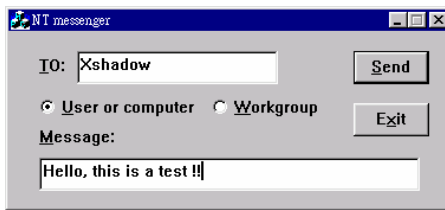


圖 B-10 / NT messenger 程式

我十分好奇它的運作原理，於是使用 APISPY32 來觀察它的 API 呼叫情形，如下：

```

GetStartupInfoA(LPDATA:0012FF38)
GetStartupInfoA returns: 12FFB0
GetTempPathA(DWORD:00000050,LPSTR:0012F880:" ")
GetTempPathA returns: 8
GetTempFileNameA(LPSTR:0012F880:"E:\TEMP\",LPSTR:0040307C:"goody",D
WORD:00000000,LPSTR:0012F8D0)
GetTempFileNameA returns: C
WinExec(LPSTR:00351320:"cmd /c net",DWORD:00000000)
WinExec returns: 21
Sleep(DWORD:00000BB8)
Sleep returns: 0
lstrlenA(LPSTR:00351320:"The messag")
lstrlenA returns: 30

```

天啊，原來它只是暗中呼叫 Windows NT 內建的 net 指令，並不是直接使用 mailslot API 函式與網路上其它電腦溝通，真令人失望，本來還以為是一個好的研究對象呢。

Socket Spy/32

若想要得知網路軟體的通訊流程，或者研究軟體使用的未公開通訊協定（例如 ICQ 的通訊協定等等），這套 Socket Spy/32 絕對是最佳幫手。

自行發展 WinSock 應用程式時，我也經常以 Socket Spy/32 來觀察／記錄客戶端及伺服器之間的溝通流程，尤其在程式不定時地發生錯誤時。Socket Spy/32 能夠忠實地記錄軟體的 WinSock API 函式呼叫，並將參數及資料以一目瞭然的方式呈現，通常可以幫助我很快地找到問題的根源。下圖是我使用抓信軟體經由 POP3 連接埠連線到郵件伺服器（vcl.vclxx.com）時的 WinSock API 呼叫情形。

- 工具名稱：Socket Spy/32
- 發行方式：共享軟體
- 發行公司或作者：WinTECH
- URL：http://www.win-tech.com
- 用途：刺探網路應用程式的 WinSock API 呼叫行爲

```

Socket Spy/32 - Professional Edition
File Edit View Setup Help
[1] 12:01:38:002 (0x7A) socket (af=PF_INET, type=SOCK_STREAM, protocol=0) returns (SOCKET=856)
[2] 12:01:38:002 (0x7A) WSAAsyncSelect (SOCKET=856, hWnd=0x0041043E, wParam=0x0464, lEvent=0x00000010)
[3] 12:01:38:002 (0x7A) connect (SOCKET=856, SOCKADDR.Length=16,
                             .family=AF_INET
                             .port=110
                             .address=140.114.89.40) returns (WSAEWOULDBLOCK)
[4] 12:01:38:002 (0x7A) WSAGetLastError () returns (WSAEWOULDBLOCK)
[5] 12:01:38:493 (0x7A) WSAAsyncSelect (SOCKET=856, hWnd=0x0041043E, wParam=0x0464, lEvent=0x00000023)
[6] 12:01:38:493 (0x7A) recv (SOCKET=856, flags=0x0000) returns (WSAEWOULDBLOCK)
[7] 12:01:38:493 (0x7A) WSAGetLastError () returns (WSAEWOULDBLOCK)
[8] 12:01:40:386 (0x7A) recv (SOCKET=856, flags=0x0000) returns (56 bytes)
0000: 2B 4F 4B 20 76 63 6C 78 78 2E 63 6F 6D 20 50 4F +OK.vclxx.com.PO
0010: 50 20 73 65 72 76 69 63 65 20 72 65 61 64 79 20 P.service.ready.
0020: 5B 33 5D 20 4D 44 61 65 6D 6F 6E 20 76 32 2E 38 [3].MDaemon.v2.8
0030: 2E 35 2E 30 20 52 0D 0A .5.0.R..
For Help, press F1 Trap Buffer Empty Trigger Off 1: 28

```

圖 B-11 / 記錄抓信軟體的 WinSock API 呼叫歷程

Registry Monitor

我常常會好奇某些軟體究竟從哪邊取得很特別的系統資訊（又來了，我玩電腦的主要動機幾乎可說是「好奇」兩個字），是從登錄資料庫呢？還是直接跟作業系統要來的？這時候就可拿出 Registry Monitor，監看系統所有的登錄資料庫存取動作，藉著設定適當的過濾條件，就可以清楚地看到該軟體的行為。

- 工具名稱：NT Registry Monitor
- 發行方式：免費軟體
- 發行公司或作者：Mark Russinovich、Bryce Cogswell
- URL：<http://www3.ncr.com/support/nt/tools/ntregmon.htm>
- 用途：記錄系統的登錄資料庫存取動作

- 工具名稱：Win95 Registry Monitor
- 發行方式：隨書附送（*Inside Windows 95 Registry*）
- 發行公司或作者：Mark Russinovich、Bryce Cogswell
- URL：mark@osr.com、cogswell@cs.uoregon.edu
- 用途：記錄系統的登錄資料庫存取動作

The screenshot shows the NT Registry Monitor application window with a menu bar (File, Events, Help) and a table of registry operations. The table has columns for #, Process, Request, Path, and Result. The operations listed include queries and modifications to the font substitutes registry path and the HKCU\Control Panel\Input Method path.

#	Process	Request	Path	Result
2765	UEDIT32.EXE	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS
2766	UEDIT32.EXE	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes新細明體	NOTFOUND
2767	UEDIT32.EXE	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS
2768	EXPLORER.EXE	OpenKey	0xE19A57C0\Control Panel\Input Method	SUCCESS
2769	EXPLORER.EXE	QueryValue	0xE19A57C0\Control Panel\Input Method\show status	SUCCESS
2770	EXPLORER.EXE	CloseKey	0xE19A57C0\Control Panel\Input Method	SUCCESS
2771	UEDIT32.EXE	OpenKey	HKCU\Control Panel\Input Method	SUCCESS
2772	UEDIT32.EXE	QueryValue	HKCU\Control Panel\Input Method\show status	SUCCESS
2773	UEDIT32.EXE	CloseKey	HKCU\Control Panel\Input Method	SUCCESS
2774	UEDIT32.EXE	OpenKey	HKCU\Control Panel\Input Method	SUCCESS
2775	UEDIT32.EXE	QueryValue	HKCU\Control Panel\Input Method\show status	SUCCESS
2776	UEDIT32.EXE	CloseKey	HKCU\Control Panel\Input Method	SUCCESS
2777	UEDIT32.EXE	OpenKey	HKCU\Control Panel\Input Method	SUCCESS
2778	UEDIT32.EXE	QueryValue	HKCU\Control Panel\Input Method\show status	SUCCESS
2779	UEDIT32.EXE	CloseKey	HKCU\Control Panel\Input Method	SUCCESS
2780	UEDIT32.EXE	OpenKey	HKCU\Control Panel\Input Method	SUCCESS
2781	UEDIT32.EXE	QueryValue	HKCU\Control Panel\Input Method\show status	SUCCESS
2782	UEDIT32.EXE	CloseKey	HKCU\Control Panel\Input Method	SUCCESS
2783	UEDIT32.EXE	OpenKey	HKCU\Control Panel\Input Method	SUCCESS
2784	UEDIT32.EXE	QueryValue	HKCU\Control Panel\Input Method\show status	SUCCESS
2785	UEDIT32.EXE	CloseKey	HKCU\Control Panel\Input Method	SUCCESS
2786	UEDIT32.EXE	OpenKey	HKLM\SYSTEM\CurrentControlSet\Services\MGACtrlPowerDesk\Current Settings	SUCCESS
2787	UEDIT32.EXE	QueryValue	HKLM\SYSTEM\CurrentControlSet\Services\MGACtrlPowerDesk\Current Settings	SUCCESS

圖 B-12 / 記錄應用程式的登錄資料庫存取動作

即時偵錯／除錯

SoftICE

我常覺得好笑，哪有 C++Builder 程式設計師像我一樣整天把 SoftICE 掛著，時時拿出來研究把玩的呀？

SoftICE 是系統層級的超強除錯器，它的能力真的很強，很強，無敵地強，在它的掌控下，只要想做的，沒有做不到的事。啟動時，它會將整個系統的中斷停住，甚至連系統時間都停止更新，把整個系統控制權交給你－SoftICE 使用者，你說它到底強不強。正因如此，只要長時間處於 SoftICE 使用模式，系統時間就會明顯延遲，害得我跟朋友約會時經常遲到：～

它能做什麼呢？以我來說，經常在下列數種場合使用它：

- 有時不滿足 C++Builder 整合環境提供的除錯器，便打開專案的「Include TD32 Debug Info」選項，以 SoftICE 為 C++Builder 程式除錯。
- 以 SoftICE 充當反組譯器，反組譯小範圍的機械碼。
- 對於不瞭解的軟體功能，以 SoftICE 研究它的實作。
- 我愛在程式中加入除錯訊息來輔助偵錯，而 SoftICE 會自動將不是被除錯狀態的應用軟體所吐出的除錯訊息記錄下來。
- 嫌某個共享軟體實在太貴，或者試用期間太短時，以 SoftICE 將它破解。:P
- 開發驅動程式（當然不是用 C++Builder 寫的 :P）時，由於載入順序的關係²，非得使用 SoftICE 為除錯器。

² 由於載入順序的關係，有些驅動程式（例如磁碟驅動程式、檔案系統驅動程式）無法以任何除錯器來偵錯，包括 SoftICE。

- 工具名稱：SoftICE
- 發行方式：商業軟體
- 發行公司或作者：NuMega
- URL：http://www.numega.com/
- 用途：系統層級除錯器

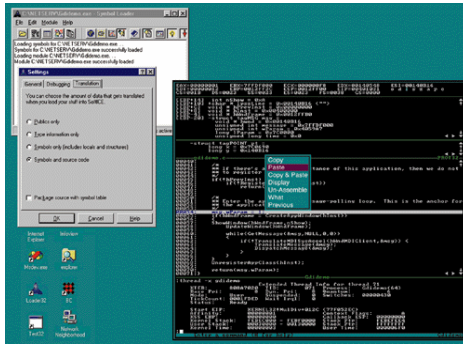


圖 B-13 / 在畫面上的 SoftICE 視窗

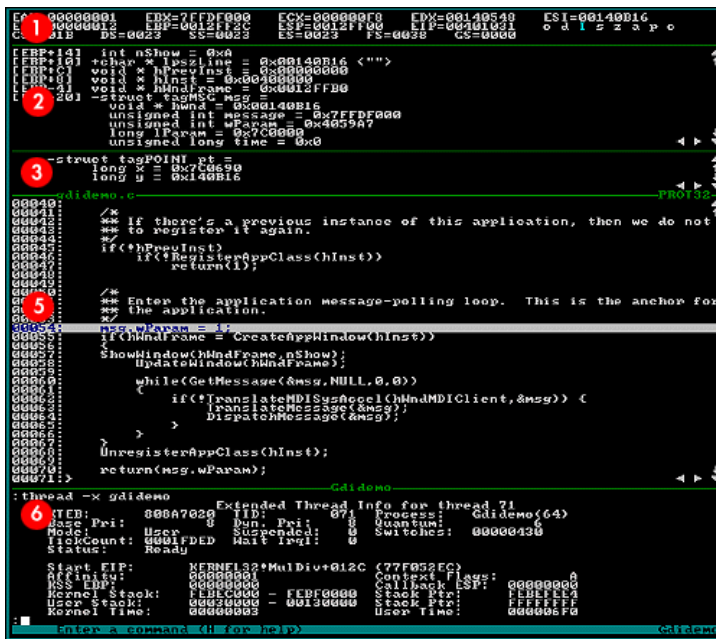


圖 B-14 / SoftICE 視窗：1.暫存器 2.區域變數 3.觀看特定變數 5.原始碼 6.命令視窗

DebugView

Win32 API 提供一道 *OutputDebugString* 函式，用來吐出除錯訊息，它的作用是：

- 若程式為被除錯狀態（除錯器存在），則將除錯訊息交由除錯器印出（在 C++Builder 整合環境中，除錯訊息列於 Event Log 視窗）。
- 若程式不是被除錯狀態，則將除錯訊息交由系統除錯器印出。
- 若系統除錯器（如 SoftICE）不存在，則什麼都不做。

DebugView 的作用是，你不必為了接收所有應用程式的除錯訊息就特地安裝系統除錯器，只要使用 DebugView，它可以隨時啟動／結束，攔截任何未被應用程式除錯器處理的除錯訊息。

使用方便的 DebugView，就可以大量地藉助除錯訊息來輔助程式偵錯。

- 工具名稱：DebugView
- 發行方式：免費軟體
- 發行公司或作者：Mark Russinovich
- URL：<http://www.sysinternals.com/>
- 用途：除錯訊息記錄工具

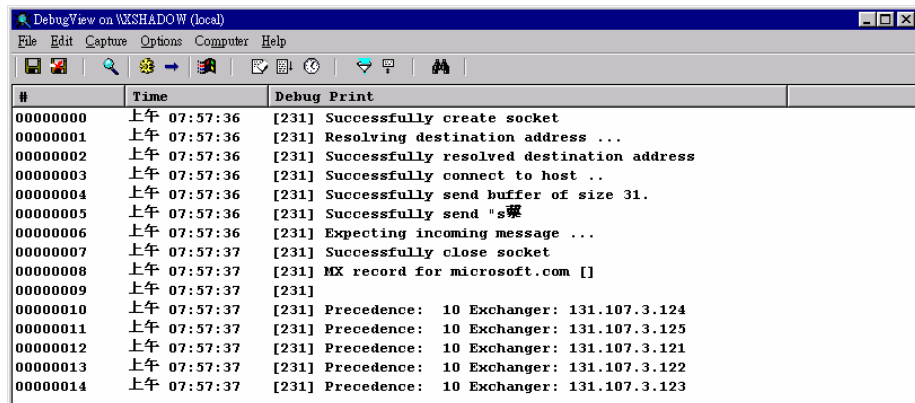


圖 B-15 / 接收 MX records 查詢元件所吐出的除錯訊息

資源檢視／修改

Resource Workshop

Borland 公司出品的資源檢視／修改工具，支援各種資源類型，除了常見的 bitmap、滑鼠指標及圖示外，還支援對話盒、字型、功能表、字串、版本資源等等。除此之外，它還可將資源轉換為 RC 資源描述檔案。

- 工具名稱：Resource Workshop
- 發行方式：附於開發套件（C++Builder 4、C++Builder 4 後的版本開始附送）
- 發行公司或作者：Borland
- URL：<http://www.borland.com/>
- 用途：資源編輯工具

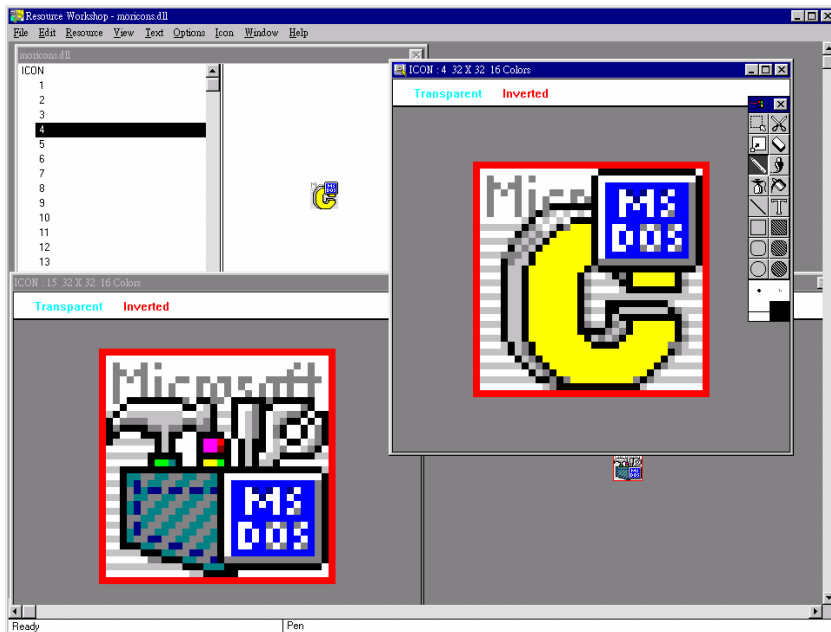


圖 B-16 / 以 Resource Workshop 編輯 DLL 檔裡的圖示資源

Microsoft Developer Studio

這是 Microsoft 公司出品的程式開發整合環境。不過對我來說，最大的好處卻是將它當成功能完整的資源檢視／修改工具來用。

Developer Studio 支援各種資源類型，使用者介面設計良好，方便易用。唯一的缺憾是，在 Windows 95/98 下，無法將資源寫入執行檔或 DLL 的資源區段。

- 工具名稱：Developer Studio
- 發行方式：商業軟體
- 發行公司或作者：Microsoft
- URL：<http://www.microsoft.com/>
- 用途：整合型開發環境

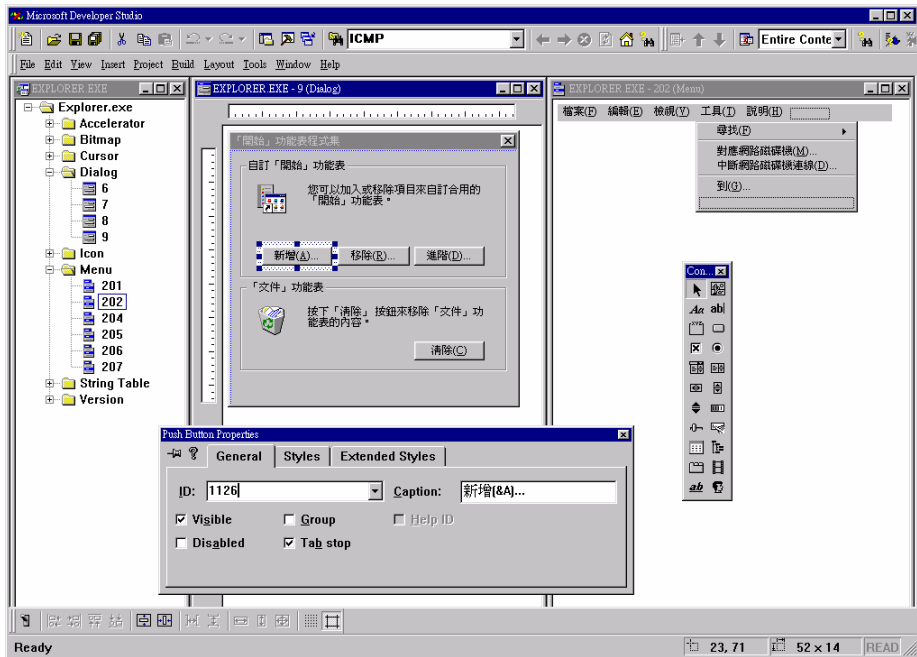


圖 B-17 / 正在編輯 EXE 檔裡的對話盒及功能表資源

Language Localizer

C++Builder 應用程式的多國語文化一直是十分傷腦筋的問題，因為 form 及元件的資料皆儲存於 RCDATA 資源區段，以 DFM 格式儲存，無法被標準的資源編輯器辨認／處理，所以無法像以 SDK、Visual C++ 等開發工具撰寫的程式那樣，能夠直接以資源編輯器進行多國語文化工作。

不少廠商為 C++Builder 程式設計師提出各自的解決方案，大部分的解決方案都要求程式設計師在執行時期就將他們提供的 VCL 元件加入專案中一併編譯連結，十分麻煩。後來，經過比較試用後，我決定 Language Localizer 這套工具。

Language Localizer 的特性是，不必對執行檔進行任何更動，也不需要原始碼的配合，只要是正常的 Delphi 或 C++Builder 寫成的執行檔或 DLL，就可以直接進行多國語文化。

它的使用方法是，載入執行檔（或 DLL），讓它取出 form 上所有元件的文字以及資源區段裡的資源字串，以手動或自動的方式來更改文字。例如，你可以餵給它一個英漢對照的詞庫，就可以很快地將英文版軟體改為中文版軟體。

唯一需要程式設計師配合的是，必須將原始碼使用到的字串放到資源區段中。換句話說，每當你想在程式中使用以單引號括起來的字串時，就請使用任何資源編輯程式，或直接撰寫資料描述檔案 (.RC)，再以資源編譯器 brcc32 來進行編譯。最後，再使用 `#pragma` 編譯指示將資源檔含入執行檔內即可。

- 工具名稱：Language Localizator
- 發行方式：免費軟體
- 發行公司或作者：HLINKA-SOFT
- URL：http://www.clexpert.cz/software/localize/english/
- 用途：C++Builder / C++Buidler 程式多國語文化

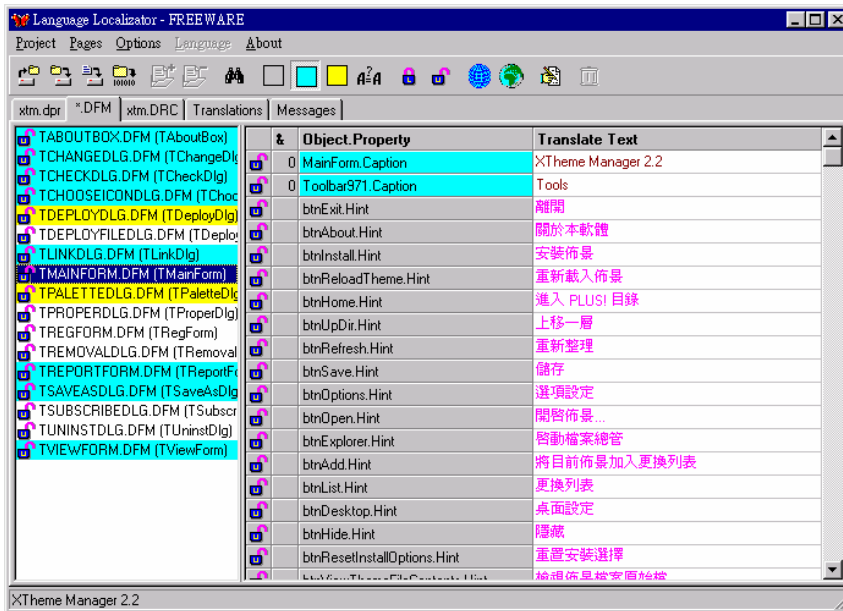


圖 B-18 / 列出 form 上所有元件的文字來進行修改

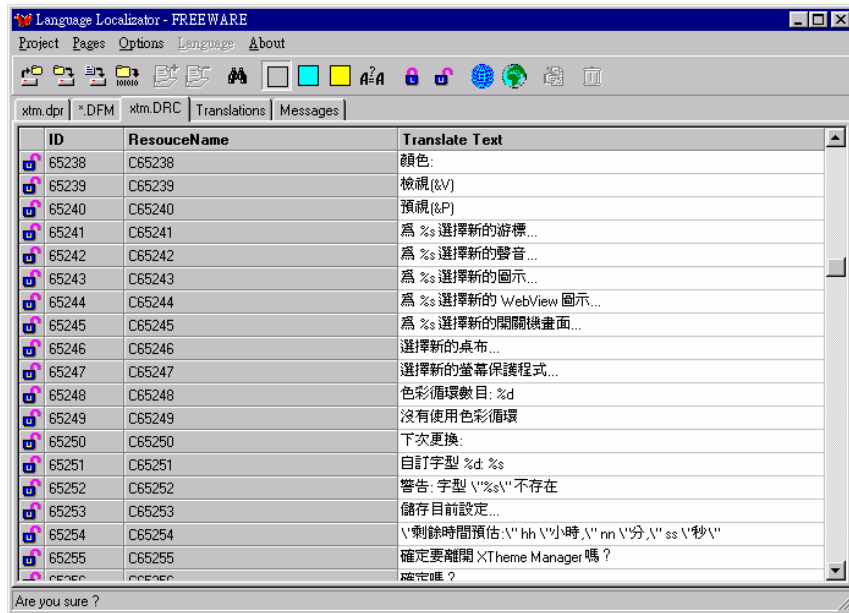


圖 B-19 / 列出所有字串資源以進行修改

系統資訊觀察

OLE/COM Object Viewer

檢視 COM 物件、介面、Type Libraries 的工具，讓你不必在充斥著一大串 GUID 及複雜交互參考關係的登錄資料庫內找尋這些 COM 相關資訊。

- 工具名稱：OLE/COM Object Viewer
- 發行方式：附於開發套件，也可從 Microsoft 網站免費下載
- 發行公司或作者：Microsoft
- URL：http://www.microsoft.com
- 用途：檢視 COM 物件、介面、Type Libraries

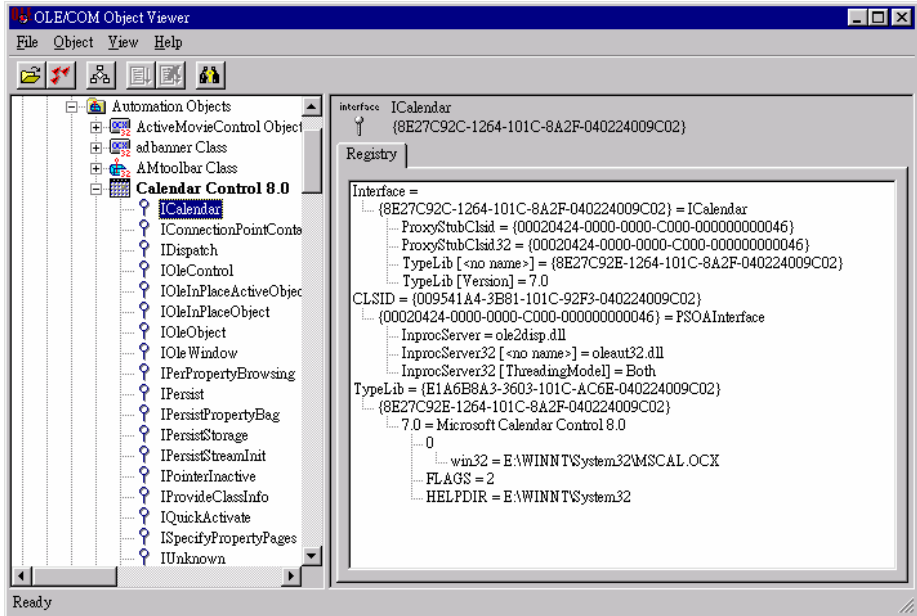


圖 B-20 / 檢視 Calendar Control 的 ICalendar 介面

Process Viewer

觀察行程、執行緒及行程記憶體使用情形的工具，通常我拿它來查看行程所包含的執行緒數目，以及各個執行緒所佔用的 CPU 時間是否正常等等。

- 工具名稱：Process Viewer
- 發行方式：附於開發套件
- 發行公司或作者：Microsoft
- URL：http://www.microsoft.com/
- 用途：觀察行程及記憶體使用情形

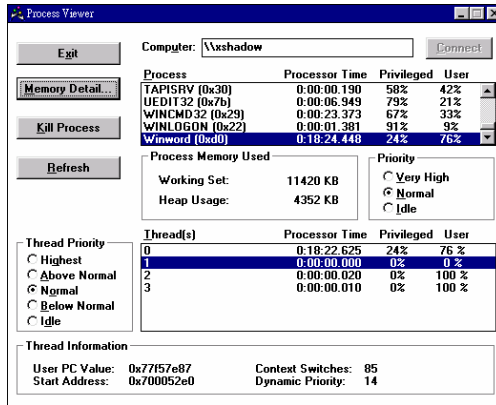


圖 B-21 / 觀察 Winword 行程中的執行緒

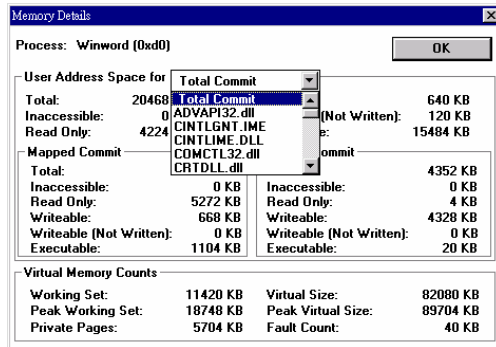


圖 B-22 / 查看 Winword 行程裡各模組的記憶體使用情況

Dependency Walker

DUMPBIN 之類的檔案剖析工具可以檢視檔案的 import table、export table 等等資訊，而 Dependency Walker 可說是這些資訊的最佳分析檢視工具。它針對某個執行檔（或 DLL）進行分析，列出該檔案參考使用的其它 DLL，有助於 DLL 之間循環參考錯綜關係的分析檢視。

- 工具名稱：Dependency Walker
- 發行方式：附於開發套件
- 發行公司或作者：Microsoft

- URL : <http://www.microsoft.com/>
- 用途：察看 DLL 檔的參考使用情形

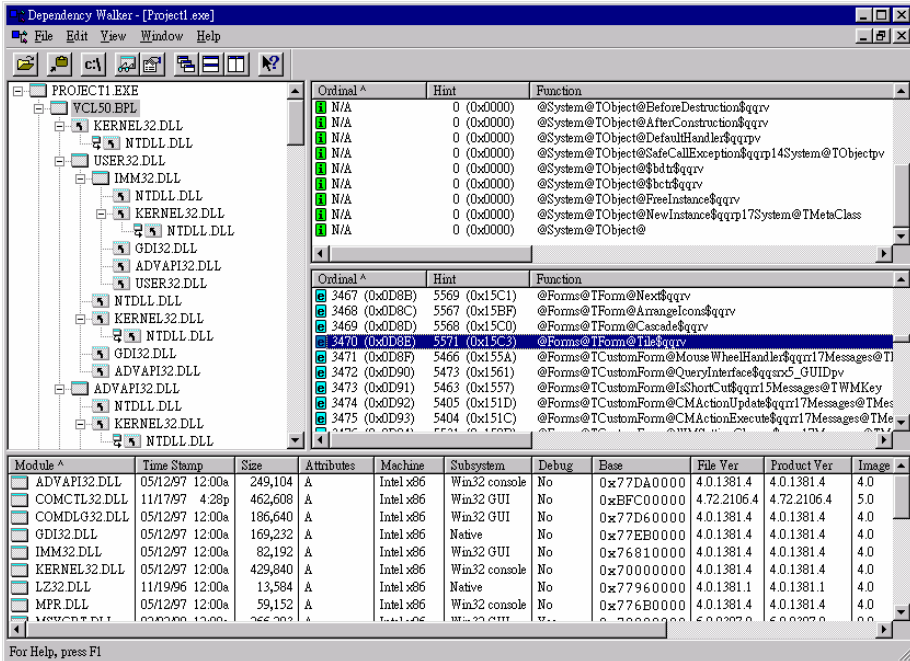


圖 B-23 / 檢視 Project1.exe 的 DLL 使用情況

RegDump

以文字格式傾印登錄資料庫內容。

- 工具名稱：RegDump
- 發行方式：隨書附送 (*Inside the Windows 95 Registry*)
- 發行公司或作者：Andrew Schulman
- URL : andrew@ora.com
- 用途：傾印登錄資料庫內容


```
d:\util>regdump HKEY_CURRENT_USER\Software\Xshadow\XViewer

Key: HKEY_CURRENT_USER\Software\Xshadow\XViewer

Key: HKEY_CURRENT_USER\Software\Xshadow\XViewer\General

TreeViewWidth is REG_SZ: 279
MemoFont is REG_SZ: "Fixedsys", 12, [], [clBlack]

Key: HKEY_CURRENT_USER\Software\Xshadow\XViewer\WinSaver

TMainForm_Width is REG_SZ: 1280
TMainForm_Height is REG_SZ: 996
TMainForm_Top is REG_SZ: 0
TMainForm_Left is REG_SZ: 0
TMainForm_WindowState is REG_SZ: 3
```

TCPView

它的功能相當於“netstat -a”指令，能將目前的 TCP/UDP 各連接埠狀況列出，可藉此得知系統的連線情形。不過一來是視窗程式，二來狀態更新的速度較快，所以我通常拿它來取代 netstat 指令。

- 工具名稱：TCPView for Windows NT
- 發行方式：免費軟體
- 發行公司或作者：Mark Russinovich
- URL：http://www.sysinternals.com/
- 用途：觀察系統上的 TCP/UDP 連接埠使用情況

The screenshot shows the TCPView application window with a menu bar (File, View, Help) and a toolbar. The main area contains a table of network connections with the following columns: Protocol, Local Address, Remote Address, and State.

Protocol	Local Address	Remote Address	State
TCP	xshadow: 1026	0.0.0.0	LISTENING
TCP	xshadow: 1096	0.0.0.0	LISTENING
TCP	xshadow: 135	0.0.0.0	LISTENING
TCP	xshadow: 135	0.0.0.0	LISTENING
TCP	localhost: 1025	0.0.0.0	LISTENING
TCP	localhost: 1025	localhost: 1026	ESTABLISHED
TCP	localhost: 1026	localhost: 1025	ESTABLISHED
TCP	localhost: 1042	0.0.0.0	LISTENING
TCP	dial8-170.Eden.nthu.edu.tw: 1088	cs.nthu.edu.tw: pop3	TIME_WAIT
TCP	dial8-170.Eden.nthu.edu.tw: 1089	bbs.cs.nthu.edu.tw: pop3	TIME_WAIT
TCP	dial8-170.Eden.nthu.edu.tw: 1090	vc1.cs.nthu.edu.tw: pop3	TIME_WAIT
TCP	dial8-170.Eden.nthu.edu.tw: 1091	vc1.cs.nthu.edu.tw: pop3	TIME_WAIT
TCP	dial8-170.Eden.nthu.edu.tw: 1096	vc1.cs.nthu.edu.tw: 100	ESTABLISHED
TCP	dial8-170.Eden.nthu.edu.tw: 137	0.0.0.0	LISTENING
TCP	dial8-170.Eden.nthu.edu.tw: 138	0.0.0.0	LISTENING
TCP	dial8-170.Eden.nthu.edu.tw: nbsession	0.0.0.0	LISTENING
TCP	XSHADOW: 1092	0.0.0.0	LISTENING
TCP	XSHADOW: 1092	COOKIE: nbsession	ESTABLISHED
TCP	XSHADOW: 137	0.0.0.0	LISTENING
TCP	XSHADOW: 138	0.0.0.0	LISTENING
TCP	XSHADOW: nbsession	0.0.0.0	LISTENING
TCP	XSHADOW: nbsession	COOKIE: 1025	ESTABLISHED

The status bar at the bottom of the window displays "Ready."

圖 B-24 / 目前電腦上的 TCP/UDP 連接埠狀況

OSR Driver and Device Explorer

檢視系統上的驅動程式以及它們各自負責的裝置資訊。

- 工具名稱：OSR's Driver and Device Exploration Utility
- 發行方式：免費軟體
- 發行公司或作者：OSR Open Systems Resources
- URL：http://www.osr.com
- 用途：檢視驅動程式及裝置資訊

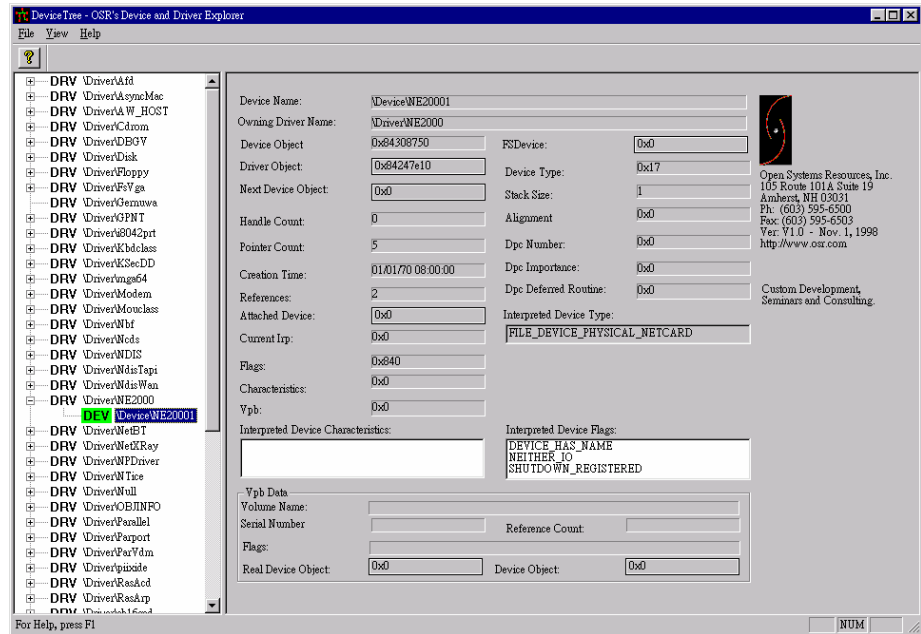


圖 B-25 / 檢視 NE2000 網路卡裝置的狀態

雜項

Hex Workshop

十六進位形式的檔案編輯／計算工具。

- 工具名稱：Hex Workshop—The Professional Hex Editor
- 發行方式：共享軟體
- 發行公司或作者：BreakPoint Software
- URL：http://www.bpssoft.com/
- 用途：十六進位形式的檔案編輯／計算工具

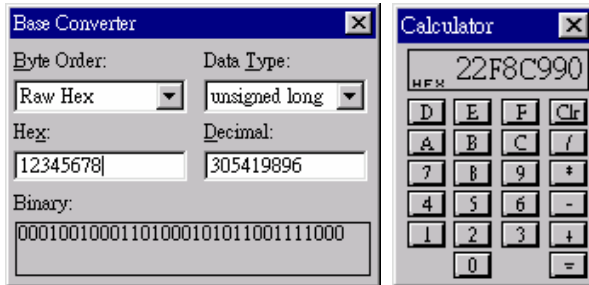


圖 B-27 / 進位轉換工具及十六進位計算機

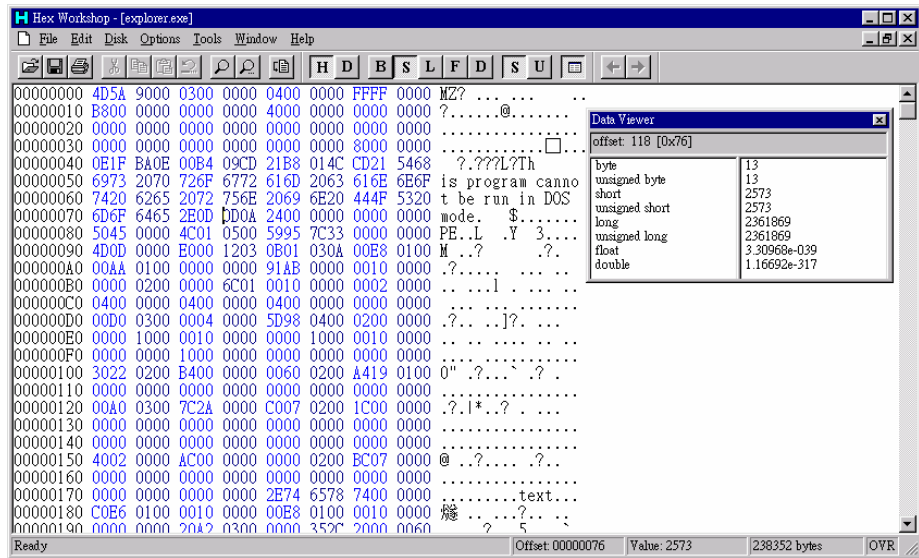
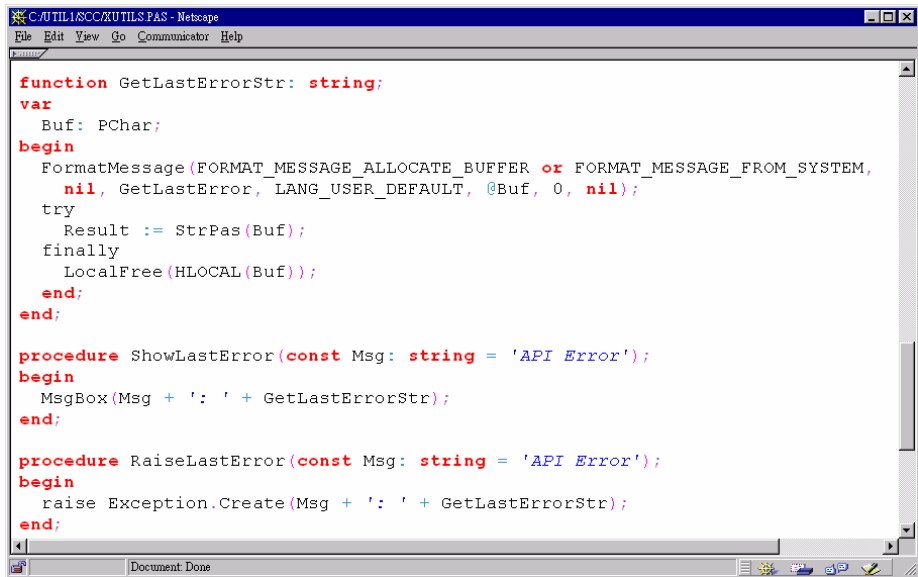


圖 B-28 / 功能齊全的十六進位模式檔案編輯環境

Source Code Colorizer

程式寫好了，想將它放上網站供 C++Builder 同好們欣賞，但是直接擺上去很醜，所以最好使用 Source Code Colorizer 這類工具將程式碼轉為 syntax highlighted HTML。所謂 syntax highlighted 就是說，對於保留字、關鍵字、字串、整數、浮點數等等程式中不同種類的 token，予以不同的字型、顏色來顯示，會使原始碼看起來更為賞心悅目。

- 工具名稱：SCC Source Code Colorizer
- 發行方式：共享軟體
- 發行公司或作者：Juergen Mueller
- URL：juergen.mueller@isw.uni-stuttgart.de
- 用途：將程式碼轉為 syntax highlighted HTML

The image shows a screenshot of a web browser window titled "C:\UTILS\CC\UTILS\PAS - Netzape". The browser's address bar shows "Document Done". The main content area displays Pascal code for a unit named "xUtils.pas". The code includes three procedures: "GetLastErrorStr", "ShowLastError", and "RaiseLastError".

```
function GetLastErrorStr: string;
var
  Buf: PChar;
begin
  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER or FORMAT_MESSAGE_FROM_SYSTEM,
    nil, GetLastError, LANG_USER_DEFAULT, @Buf, 0, nil);
  try
    Result := StrPas(Buf);
  finally
    LocalFree(HLOCAL(Buf));
  end;
end;

procedure ShowLastError(const Msg: string = 'API Error');
begin
  MsgBox(Msg + ': ' + GetLastErrorStr);
end;

procedure RaiseLastError(const Msg: string = 'API Error');
begin
  raise Exception.Create(Msg + ': ' + GetLastErrorStr);
end;
```

圖 B-29 / 以瀏覽器檢視 xUtils.pas 單元的轉換結果

WinDiff

檔案／目錄比對工具。它的檔案比對工具介面不但炫，而且十分實用，對於版本不同，不曉得修改過什麼地方的程式碼，只要拿 WinDiff 來比對，它會為你建立漂亮的區塊對映圖，並且在每個不同的地方標示清楚，檔案 A 這裡是怎麼寫的，檔案 B 這裡又是怎麼寫的。不只是原始碼，有時我也拿它來比對兩份不同、但十分接近的文件。對了，拿它來比對學生作業，看看是否抄襲，我想也是很好的應用。:P

- 工具名稱：WinDiff
- 發行方式：附於開發套件
- 發行公司或作者：Microsoft
- URL：http://www.microsoft.com/
- 用途：檔案／目錄比對工具

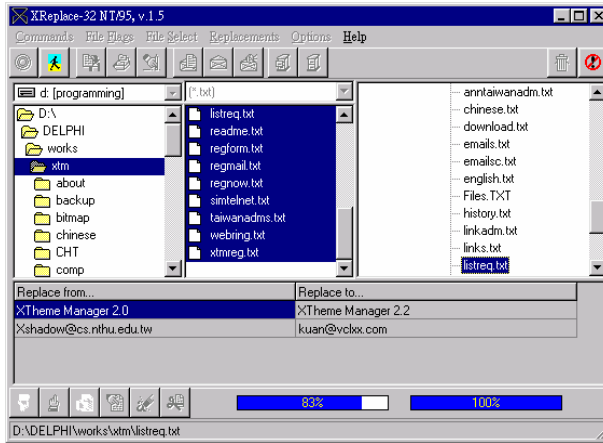


圖 B-31 / 一口氣進行多個檔案、多字串的取代動作

Windows Help Designer

Windows 說明檔案編輯／製作工具，雖然在編輯時，中文的顯示有些問題（幾乎大部分的 Windows 說明檔案編輯工具都與 DBCS 字元不相容），但是編譯過程及成品沒有問題。它是在眾多的說明檔案編輯軟體中，操作介面及功能強度都還不錯的選擇。

- 工具名稱：Windows Help Designer
- 發行方式：共享軟體
- 發行公司或作者：Nick Ameladiotis
- URL：http://www.devgr.com/
- 用途：Windows 說明檔案編輯／製作工具

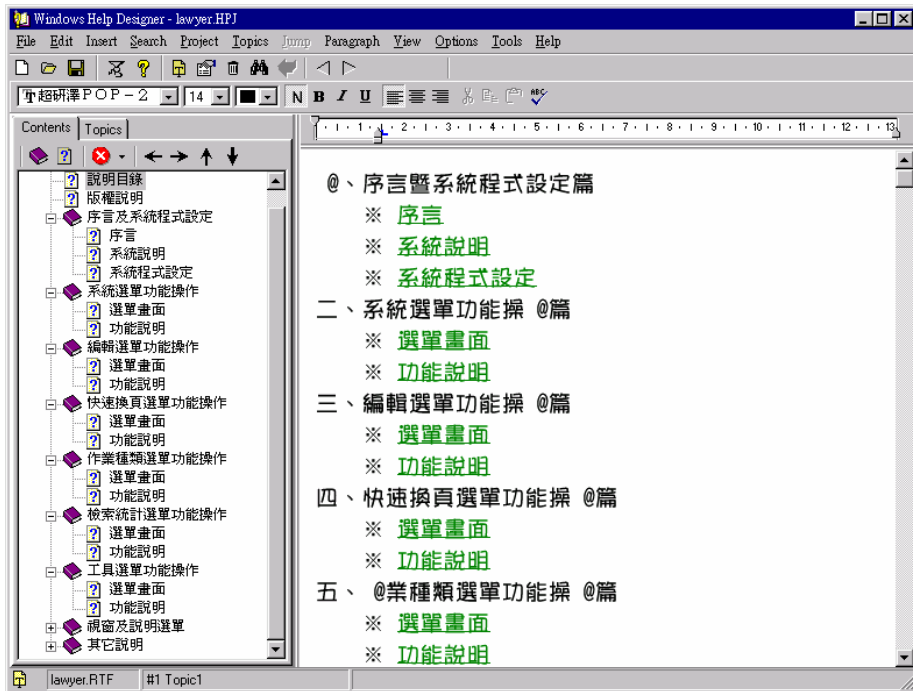


圖 B-32 / WYSIWYG 的編輯畫面

附錄 C

參考書目

以下列出我認為相當值得一讀的相關書籍（排列順序不代表任何意義）。

C / C++

- The C Programming Language
Brian W. Kernighan, Dennis M. Ritchie / Prentice Hall
- The C++ Programming Language
Bjarne Stroustrup / Addison Wesley
C++ 程式語言經典本 / 葉秉哲 / 儒林
- C++ Primer
Stanley B. Lippman, Josee Lajoie, Jose Lajoie / Prentice Hall
C++ Primer 中文版 / 侯捷 / 碁峰

Borland C++Builder

- C++Builder 5 Developer's Guide
Jarrod Hollingworth / SAMS
- C++Builder 4 Unleashed
Kent Reisdorph / SAMS

Borland Delphi

- Delphi 5 Developer's Guide
Xavier Pacheco, Steve Teixeira / SAMS
- Delphi 5 Unleashed
Charlie Calvert / SAMS
- Secrets of Delphi 2 : Exposing Undocumented Features of Delphi
Ray Lischner / Waite Group
- Delphi 32-Bit Programming Secrets
Tom Swan, Jeff Cogswell, Jeffrey M. Cogswell / IDG Books
- Delphi Component Design
Danny Thorpe / Addison Wesley
- Delphi 學習筆記 Win32 基礎篇
錢達智 / 碁峰
- Delphi 深度歷險
陳寬達 / 碁峰

Windows Programming

- Programming Windows, The Definitive Guide to the Win32 API
Charles Petzold / Microsoft Press
- Windows 95 System Programming SECRETS
Matt Pietrek / IDG Books
Windows 95 系統程式設計大奧秘 / 侯俊傑譯 / 旗標
- Programming Applications for Microsoft Windows
Jeffrey Richter / Microsoft Press
- Windows 95: A Developer's Guide
Jeffrey Richter, Jonathan Locke / M&T Books
Windows 95 程式設計指南 / 李書良譯 / 碁峰

- Multithreading Applications in Win32
James E. Beveridge, Robert Wiener / Addison Wesley
Win32 多緒程式設計 / 侯俊傑譯 / 基峰

OOA / OOD / OOP

- Object-Oriented Analysis and Design with Applications, 2nd Ed
Grady Booch / Benjamin/Cummings Publishing
- Object-Oriented Modeling And Design
James Rumbaugh, etc. / Prentice Hall
- Object-Oriented Software Construction 2nd Ed.
Bertrand Meyer / Prentice Hall
- Design Patterns, Elements of Reusable Object-Oriented Software
GoF / Addison Wesley
物件導向設計模式 / 葉秉哲 / 培生 Win32 Programming
Brent E. Rector, Joseph M. Newcomer / Addison Wesley
- 世紀末軟體革命
劉燈、賴明宗、賀元 / 傳徵

Practical Programming

- Programming Pearls 2nd Ed.
Jon Bentley / Addison Wesley
- The Practice of Programming
Brian W.Kernighan, Rob Pike / Addison Wesley
- Write Solid Code
Steve Maguire / Microsoft Press
如何撰寫 0 錯誤程式 / 施威銘研究室 / 旗標
- Code Complete : A Practical Handbook of Software Construction
Steve C McConnell / Microsoft Press

- Debugging the Development Process : Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams
Steve Maguire / Microsoft Process

工具書類

- Win32 Programming
Brent E. Rector, Joseph M. Newcomer / Addison Wesley
- Delphi Programming Problem Solver
Neil Rubenking / IDG Books
Delphi 2 高階技巧 / 劉剛樑譯 / 松格

其它值得一讀的書籍

- History of Programming Languages
Thomas J., Jr Bergin, Richard G., Jr Gibson / Addison Wesley
- 無責任書評 1 ~ 3
侯捷 / 旗標, 資迅人, 松崗