

侯捷



— 繁華似錦 —

Frameworks in C++BuilderX

— CBX 的多樣編譯環境與框架 —

台北《微電腦傳真》2004/02

北京《程序员》2004/02

作者簡介：侯捷，資訊工作者、專欄執筆、元智大學教師。常著文章自娛，頗示己志。
侯捷網站：<http://www.jjhou.com>（繁體）
北京鏡站：<http://jjhou.csdn.net>（簡體）
永久郵箱：(1) jjhou@jjhou.com (2) jjhou@ccca.nctu.edu.tw

- 讀者基礎：無需任何基礎。
- 本文適用工具：Windows XP, C++BuilderX (personal or Enterprise),
C++BuilderX preview
- 關鍵縮語（按字母順序排列）：
 - ACE**：Adaptive Communication Environment，一個專注於網絡的框架。
 - Boost**：高階 C++ Templates 程式庫，涵蓋面廣，包括 Regular Expression, Graph, Type Traits。
 - CBX Enttrial**：C++ BuilderX 企業試用版。
 - CBX Personal**：C++ BuilderX 個人版。
 - CBX Tech. Preview**：CBX 新一代編譯器先覽版本。搭配企業版使用。
 - Loki**：高階 C++ Templates 程式庫，專注於 Metaprogramming 和 Design Patterns。
 - STL**：Standard Template Library，C++標準模板程式庫，專注於資料結構和演算法。
 - STLPort**：一個可移植的 STL 版本。
 - TFB**：Together for C++BuilderX（只適用於 CBX 企業版）
 - wxWindows**：一個源碼開放、可移植的 GUI 框架。

全文提要

CBX 是 Borland 新推出的 C++ 開發平台，目前為 1.0 版（其中某些工具為先覽版本）。CBX 帶有眾多的 C++ 編譯器、框架（Frameworks），以及一個開放源碼的應用程式框架（App.Framework）。這種大熔爐式的開發環境對程式員是比較陌生的。本文將為讀者介紹 CBX 各類專案的試用、各編譯器的設定、各個框架的檔案組織與試用。由於 CBX 的下載和安裝也可能引起小小的混亂或遲疑，為節省讀者在這方面的時間花費，本文一開始也談談下載與安裝經驗。

C++ 開發工具已經很長一段時間波瀾不興了！真好，我們可以靜下心來好好學習並沉墊屬於「萬古長空」的東西。然而在維持了這麼幾年的「好光景」後，繼 Microsoft 於 2002 年初發表 Visual C++.NET（亦即 VC++7）新版本，Borland 也於 2003 年末推出 C++BuilderX，看來我們又得花點時間在「一朝風月」上頭了。

任何新版編譯工具總是會在速度、效率、團隊開發方便性上有許多表現，但這不是本文的焦點。CBX 宣稱支援多種 C++ 編譯器、可生成多種平台上的可執行檔碼、帶有多個框架（大型程式庫）以及一個用以構築 GUI 應用程式的新框架（這裡「新」的意思是指和 C++Builder 過去所帶的 OWL 完全不相容），這些宣稱倒是第一時間引起了我的興趣，因為我正是一個對框架特別感興趣的人，而剛好 CBX「收納」的框架我都略有涉獵。我想知道，眾多編譯器在 CBX 中是怎麼組織起來的？眾多框架又是怎麼組織的？如何在程式中用上它們？是否需要什麼設定？尋找答案的過程促成了這篇文章。我將以此文帶您理解 CBX 的輪廓，包括其下載、安裝、所附框架的源碼位置和編譯設定方式。本文的目的是讓您在親手安裝並測試之前，便能夠臥遊寰宇，相當程度地掌握這個嶄新的 C++ 開發環境的（某些方面的）特徵。本文所談是我個人在 Windows XP 上的經驗，其他平台或可類推。

CBX 及搭配工具之下載

只要連線 Borland 公司網站 (<http://www.borland.com>) 便可在其中找到該公司各種產品的名稱、簡介和下載網頁。本文涉及的工具如下（表格中的灰底文字代表網頁上並無記載）：

產品名稱	說明	下載檔案	大小	Key
CBuilderX personal edition	個人版	cbx1_personal_windows.zip	331MB	需要
CBuilderX Enterprise edition	企業試用版 30 天有效	cbx1_enttrial_windows.zip	374MB	需要
CBuilderX Preview	(1) 新編譯器 (2) GUI framework 以上兩者先覽版本	cbx1_preview_cd_windows.zip	118MB	不需要
Together for CBuilderX, Windows edition	試用版 15 天有效 僅搭配企業版	tcbx61_win.zip	76MB	需要

看看這些肥胖的傢伙，沒寬頻可千萬別嘗試下載呀，電話費會要人命！官方網頁上沒有說明 CBX 企業版大小，因此我很自制地下載個人版。一切都很好，直到我又下載 Together 工具打算安裝，才發現其文件上寫著只能搭配企業版使用，於是只好又下載一次幾乎 400MB 的大傢伙，並且忐忑不安地卸除（uninstall）所有原先安裝（卸除動作總讓我忐忑不安），所幸一切順利。

CBX 企業版有 30 天試用限制，個人版無此限制，因此您或有需要在同一台電腦上安裝個人版和企業版。能不能夠如此呢？大概可以，我沒嘗試（30 天試用期夠我寫這篇文章了）。這兩個版本都會在安裝前詢問您安裝位置（預設都是 \CBuilderX），我猜想也許錯開它們就可以同時並存。

每次下載，螢幕上都會出現一個要您註冊或驗證的網頁，您必須確實動作，才能於稍後獲得一封來自 Borland 的電子郵件，其中附有一個 reg_.txt 檔案（約 3KB），可於稍後安裝時做為 key 使用（電子郵件內有詳細的使用說明）。如果您因為某種原因遺失了這些 keys，也可隨時再進入 CBX 網頁，再次取得 keys。

CBX 及搭配工具之安裝

雖然 WinZip 允許我們直接執行壓縮檔內的可執行檔（我指的是在 WinZip 中直接雙擊壓縮檔內的安裝程式 install.exe），但由於 CBX 個人版或企業版解壓縮後有一些很重要的東西（主要就是它所帶的三個框架：ACE, Boost, Loki 的源碼）並不

放在 CBX 安裝目錄下，因此如果我們不解開其壓縮檔，會遺漏不少財富。

下載檔案	說明	解壓縮目錄（可自設）	解壓後大小
cbx1_personal_windows.zip	個人版	\cbx1-personal	376MB
cbx1_enttrial_windows.zip	企業版	\cbx1-enttrial	423MB
cbx1_preview_cd_windows.zip	先覽工具。 似乎可以不必解壓縮，直接在 WinZip 中執行壓縮檔內的 Windows\install.exe。	\cbx1-preview-cd	114MB
tcbx61_win.zip	Together 工具	無需解壓縮，可直接在 WinZip 中執行壓縮檔內的 tcb6.1win.exe。	

接下來，執行上述每一個解壓縮目錄下的 Windows\install.exe，便可一一安裝。安裝次序應該是：CBX 企業版+Preview+Together（本文不談 Together）。

※此刻起，除非特別聲明，文中所說的 CBX 指的是企業版。

安裝過程中需要 key。稍早已談過您手上應該握有一些 reg_.txt。請將這些檔案拷貝到 Windows XP 的 **Documents and Settings\<user>** 目錄下，那麼上述軟體安裝過程中就不會向您要鑰匙了（也可以在被詢問時才拷貝進去）。

請注意，有時候您或許會從某些文件中感覺 CBX 好像可以安裝在 95/98 上，例如上述那封帶有 key 的電子郵件這麼寫著，要我們把 reg_.txt 放進：

```
Windows 95/98 (single-user) : C:\Windows
Windows 95/98 (multi-user) : C:\Windows\Profiles\<username>
Windows NT: C:\WINNT\Profiles\<username>
Windows 2000/XP: C:\Documents and Settings\<username>
```

如果您這麼做，雖可正常安裝，卻無法執行其 IDE（整合環境）。

安裝後的目錄是：

CBX	安裝目錄（預設）	安裝總量（硬碟空間）
企業版	\CBuilderX	989MB
先覽版	\CBuilderX	含於上述 989MB 中
Together	\Borland\TogetherCBX	161MB

至此，不考慮個人版，從下載之壓縮檔，到解壓縮，到安裝，CBX+Preview+Together 共吃掉 2.25GB 硬碟空間。如果刪除下載之壓縮檔，也還用掉 1.69GB；再刪除可能非必要之解壓縮檔，也還用掉 1.58GB。現代軟體真是吃硬碟的妖怪（神獸？）。

圖 1 是 CBX 安裝畫面之一。



圖 1 / CBX 安裝畫面之一（此畫面為個人版安裝畫面，所以列出的選項比較少）

下面是安裝過程中列出的清單（都是選擇 "FULL 安裝" 的結果），有助於理解我們所獲得的東西：

Product Name: Borland C++BuilderX
Install Folder:

```
D:\CBuilderX
Product Components:
  C++BuilderX,
  Samples,                               (註：個人版無)
  InstallShield Plugin,                 (註：個人版無)
  Mobile,                                (註：個人版無)
  CORBA,                                 (註：個人版無)
  Enable Installed Toolsets,
  Borland(R) C++,                         Microsoft(R) C++,          (註：個人版無)
  MinGW,                                 Intel(R) Compilers and Libraries, (註：個人版無)
  Forte C++,                             Enable Additional Toolsets,
  GNU C++,                               MetroWerks CodeWarrior(TM)      (註：個人版無)
Disk Space Information (for Installation Target):
  Required: 350,256,101 bytes
```

※ 請注意，這裡並未列出 ACE, Boost, Loki。正如稍早所說，它們被放在他處。

```
Product Name:
  Borland C++ Technology Preview
Install Folder:
  D:\CBuilderX
Disk Space Information (for Installation Target):
  Required: 357,987,667 bytes
```

```
Product Name:
  Borland® Together® Edition for C++BuilderX(TM) (version 6.1)
Install Folder:
  D:\Borland\TogetherCBX
Shortcut Folder:
  D:\Documents and Settings\user\「開始」功能表\程式集\TogetherCBX
Java VM Installation Folder:
  D:\Borland\TogetherCBX\jdk
Disk Space Information (for Installation Target):
  Required: 192,127,665 bytes
```

CBX 內有幾種類

想在 CBX 下寫個程式，第一個步驟就是點選 **File ▶ New...** 啓動 **Object Gallery**，畫面如圖 2。這個工具提供多種專案種類，可以為我們自動生成程式檔案。熟用現代化整合開發環境（IDE）的讀者對這種作業模式必然已不陌生。

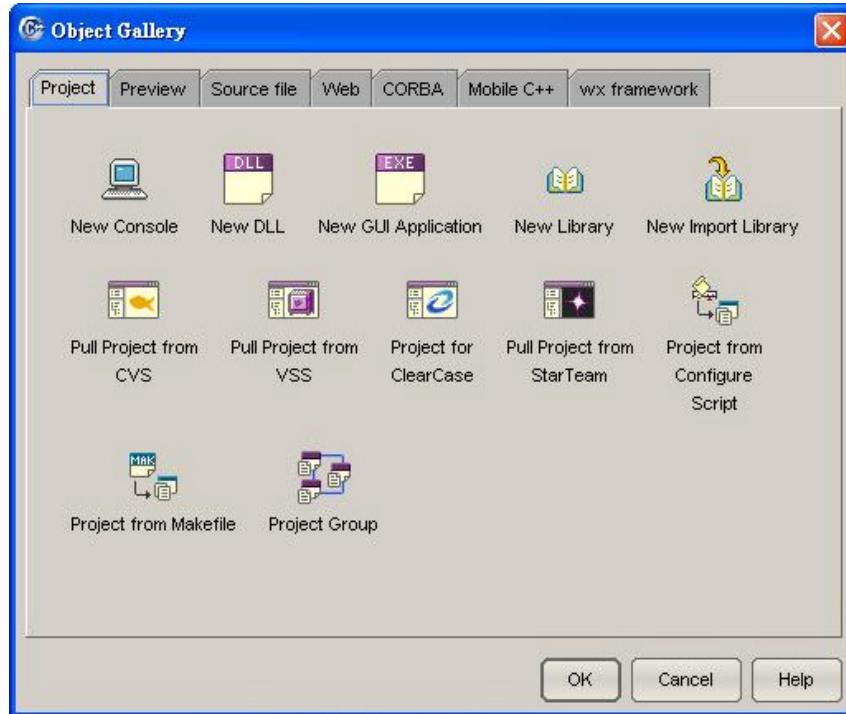


圖 2 / Object Gallery，提供多種專案種類，並自動為您生成相應的程式碼骨幹。注意，安裝 CBX Preview 之後本圖的 Preview 頁籤和 wx framework 頁籤才有作用。

為了在 CBX 中實際寫點小東西執行看看，我從 Object Gallery 中挑選了一些 EXE（而不是 DLL, LIB 之類）專案試試。以下列出五種專案的特性：

■種類一 **Project▶New Console**：這種專案的程式進入點為 main()，在 console/text 模式中執行。下面是產出的程式骨幹（專案名稱為 ConsoleApp1）：

檔名	說明	內容
ConsoleApp1.cbx	專案維護檔	N/A
untitled1.cpp	程式主檔案	main() 函式空殼

我把這個專案的檔名做點改變，並在檔案中加上一些對 STL vector 容器的運用，畫面如圖 3。而後點選 **Project▶Make...**自動完成編譯聯結。此時採用的是預設編譯器和聯結器（詳見稍後介紹）。您可以從訊息窗口看到所有的編譯聯結過程。

再點選 **Run ▶ Run ...** 即可執行。執行結果（文字輸出）也會顯示於訊息窗口。

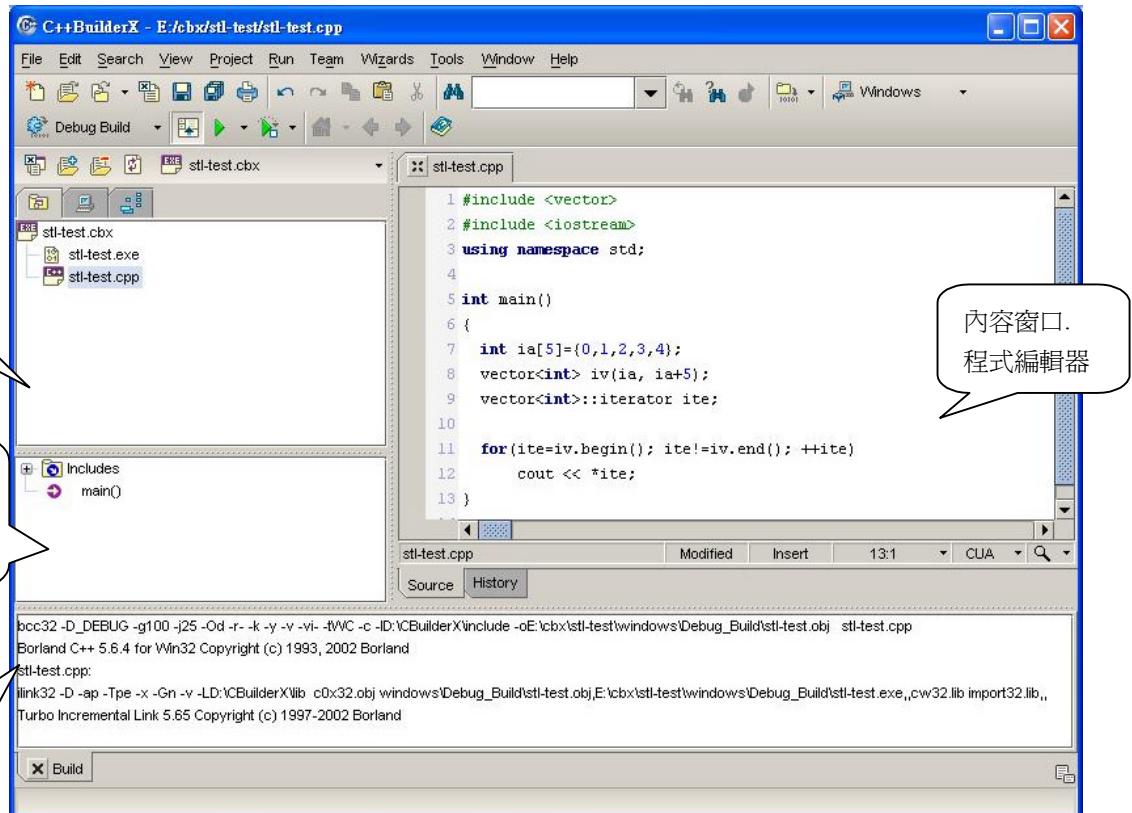


圖 3 / CBX 執行畫面全覽

■種類二 **Preview ▶ New Preview Console Application**：程式的組成和性質與種類一
同。請注意，本類專案與上一類專案都可設定採用 CBX 提供之任何編譯器（稍後
詳述編譯器），而非「**Preview**」頁籤生成者只能使用 Preview 編譯器」。既然如此
為何要區分本類（Preview）與上一類專案，目前我還不能明白。

■種類三 **Project ▶ New GUI Application**：程式進入點為 WinMain()，在 Windows
圖形模式中執行；相當於一般所謂的 Win32 SDK 程式。下面是產生出來的程式骨
幹（下例之專案名稱為 Application1）：

檔名	說明	內容簡介
Application1.cbx	專案維護檔	N/A
untitled.cpp	程式主檔案	WinMain() 函式空殼

■種類四 [Preview ► New Preview GUI Application](#)：程式的組成和性質與種類三同。

請注意，本類專案與上一類專案都可設定採用 CBX 提供之任何編譯器（稍後詳述編譯器），而非「[Preview](#)」頁籤所生成者只能使用 Preview 編譯器」。既然如此為何要區分本類（Preview）與上一類專案兩者，目前我還不能明白。

■種類五 [wx framework ► New wx framework project](#)：點選這種專案，會喚起 [New wx framework project wizard](#)，此精靈工具有兩個執行步驟，第一步（[圖 4](#)）詢問專案名稱、路徑以及窗體（form）名稱，第二步（[圖 5](#)）詢問窗體之上是否需要功能表（file, edit, help）和狀態列。這種專案的背後有巨大的 GUI 框架（定位上類似 MFC 或 OWL 或 VCL）支撐著。CBX 所採用的 GUI 框架名為 "wxWindows"（稍後介紹）。下面是這種專案的程式骨幹（下例之專案名稱為 JJHOU，窗體名稱為 wxFrame1）：

檔名	內容簡介	說明
JJHOU.cpp	程式主檔，以 wxWindows 提供的 APIs 建立一個基本的 GUI 程式。	有 wxJJHOU class 的四個成員函式 OnPaste(), OnCopy(), OnAbout(), OnQuit() 分別對應目前的四個功能表項目。
wxFrame1.cpp	IMPLEMENT_APP() wxwxFrame1::OnInit()	wxWindows 程式啓動點
wxFrame1.rc	#include JJHOU.xrc	使用 brc32.exe (Borland Resource Compiler v5.40) 進行編譯
JJHOU.xrc	xml，用以描述程式的 GUI。	

製作出來的 wxFrame1.exe 可在 Windows XP 執行，畫面如[圖 6](#)。如果把它拿到 Win98 執行，會發出「找不到 CC3260MT.DLL」的訊息。但如果您設定路徑讓 Win98 找

到該 DLL，這個程式也可順利執行，同時 UI 會自動轉換為 Win98 風格。這種 wxWindows 專案雖是 preview CD 安裝後才能製作，但亦可使用 bcc32, bccx, mingw 等任何編譯器來編譯，並不是非得以 preview CD 提供的 bccx 編譯器才能編譯。

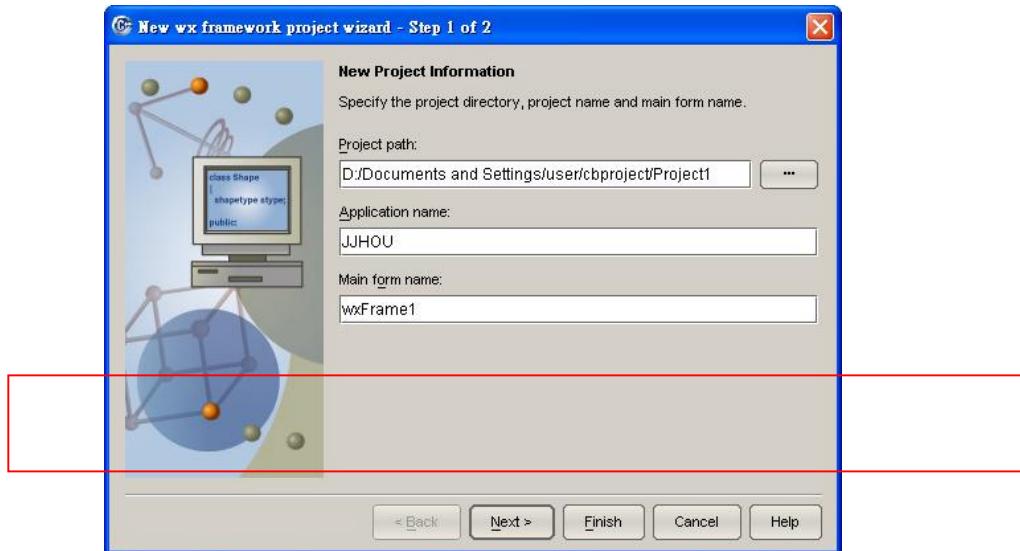


圖 4 / New wx framework project wizard 第一步驟

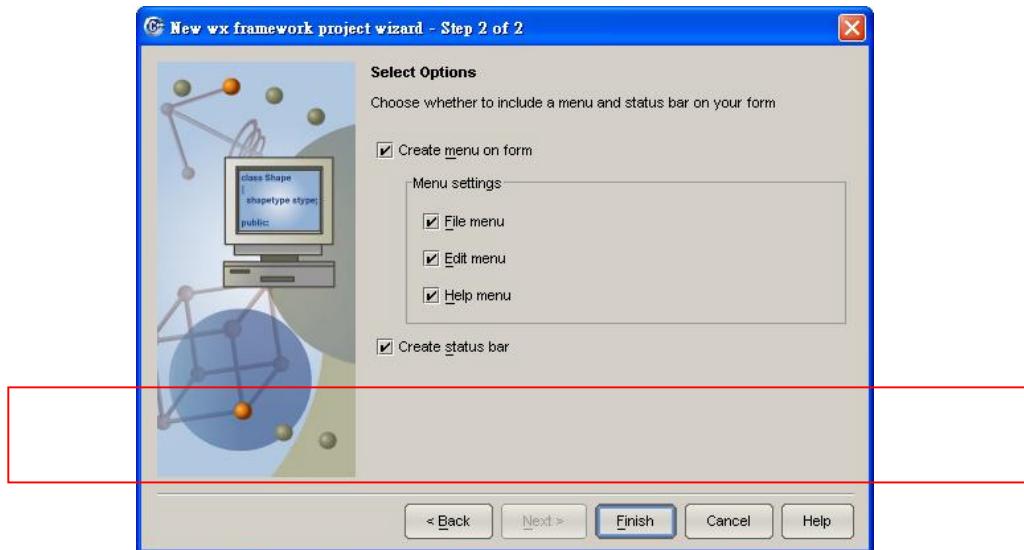


圖 5 / New wx framework project wizard 第二步驟

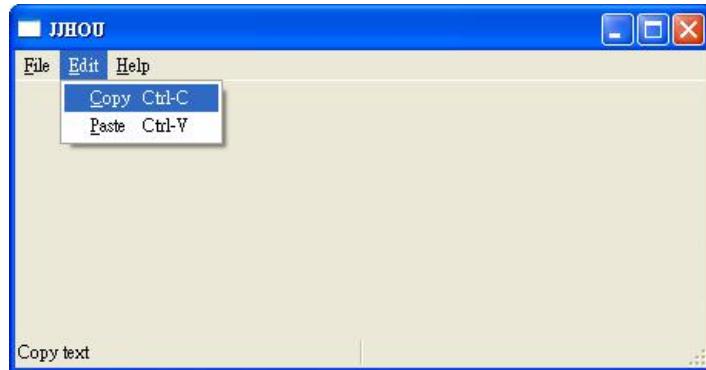


圖 6 / wxWindows 應用程式骨幹的執行畫面

CBX 搭配之編譯器及其設定

CBX 的特徵之一是，配備多種編譯器，並可產出多種平台上的可執行碼。CBX 的特徵之二是，配備多個可適用多種編譯器的框架。由於這兩個特徵，也就難怪其身軀特別龐大。下面是 CBX 支援的編譯器種類：

C++BuilderX Preview,	(以下或稱 bccx 編譯器)
Borland(R) C++,	(以下或稱 bcc32 編譯器)
MinGW,	
Microsoft(R) C++,	(個人版無)
Intel(R) Compilers and Libraries,	(個人版無)
Forte C++,	(個人版無)
GNU C++,	

當您有一個 CBX project 在手，選擇編譯器的方式是：點選 **Project▶Project Properties...** 獲得圖 7 畫面，再點選其中的 **Platform** 頁籤（畫面最右）。畫面上出現的所有選項（所有平台以及所有編譯器）都可勾選，但每個平台上只能有一個 "Active" 編譯器 — 可使用圖 7 右下角之 **Set as active toolset** 按鈕切換之。

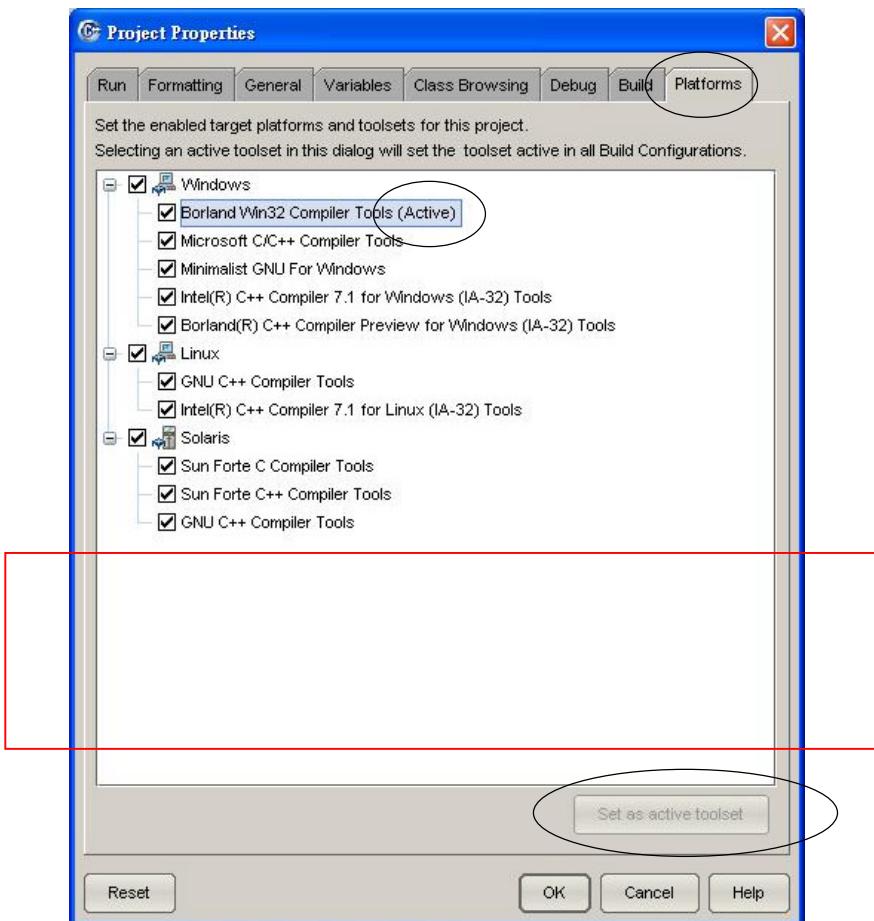


圖 7 / Project Properties 的 Platform 附頁供您挑選各種編譯器。

注意，任何時候當您選用某一種編譯器，CBX 便會在專案目錄下產生一個相應於作業平台(Windows 或 Linux 或 Solaris)的子目錄，但不會產生相應於編譯器(Borland 或 Microsoft 或 Intel 或 GNU 或 MingWG 或 Sun-Force)的子目錄。換句話說在同一專案下，您可以維護「不同作業平台」的版本，但無法維護「相同作業平台、不同編譯器」的版本。下面是 IDE 可能在專案目錄下建立的子目錄及其功用：

子目錄名稱	用途	備註
bak\...	檔案備份區	由 CBX IDE 自動維護
windows\...	只有在選用相應之編譯器後，才會出現這個目錄。	其下還分兩個子目錄 Debug_Build 和 Release_Build，分別存放除錯版和發行版之 .exe, .tds, .obj
linux\...	同上	同上
solaris\...	同上	同上

選定一個編譯器後，點選 **Project ▶ Build Objects Explorer...** 獲得圖 8 畫面，便可觀察（但不能修改）編譯器和聯結器的選項設定。

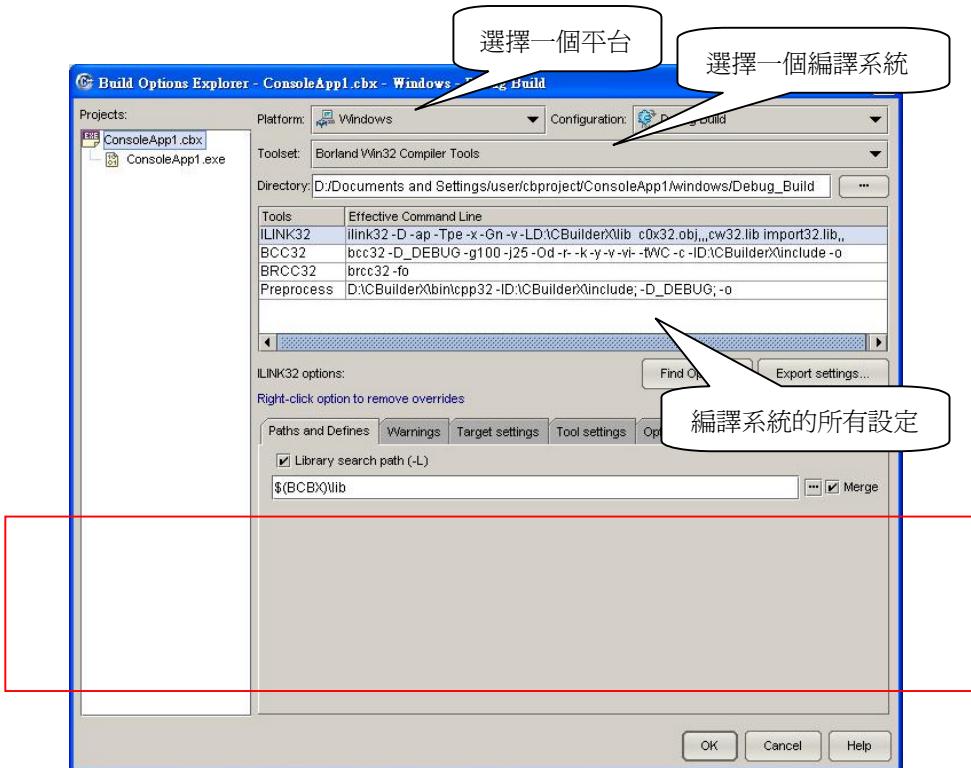


圖 8 / Build Objects Explorer 讓我們觀察編譯器和聯結器的所有設定

如果想要修改編譯系統的各項設定，可點選 **[Wizards ▶ Build Options...]**。它執行兩個步驟，第一步（圖 9）詢問稍後的設定將影響哪些專案（因為目前可能有多個專案同時被您開啟），我們可從所列之已開啟專案清單中勾選。第二步（圖 10）允許我們增加、移除或改變編譯系統（包括編譯器、聯結器、程式庫收集工具、文件、預處理器...）的各種選項，其中包括編譯器的 Include、Lib 環境變數 — 這正是稍後談到運用 Loki、Boost 等框架之前必須做的設定。

實際編譯聯結時，我們可以從 CBX 的訊息窗口（見圖 3）清楚看出哪一個編譯器被喚起，以及它帶有哪些設定（哪些編譯選項）。

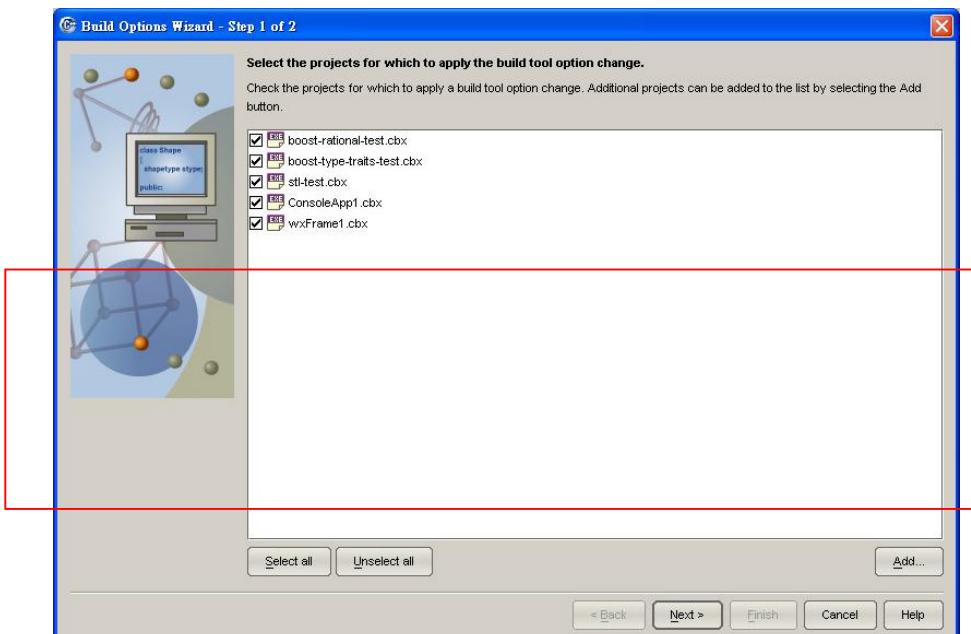


圖 9 / **Build Objects Wizard** 步驟 1，允許我們勾選影響層面（哪些專案）。本圖表示目前 CBX IDE 開啓了五個專案，而我打算稍後（步驟二）設定能夠影響所有這五個專案，所以我統統勾選了它們。

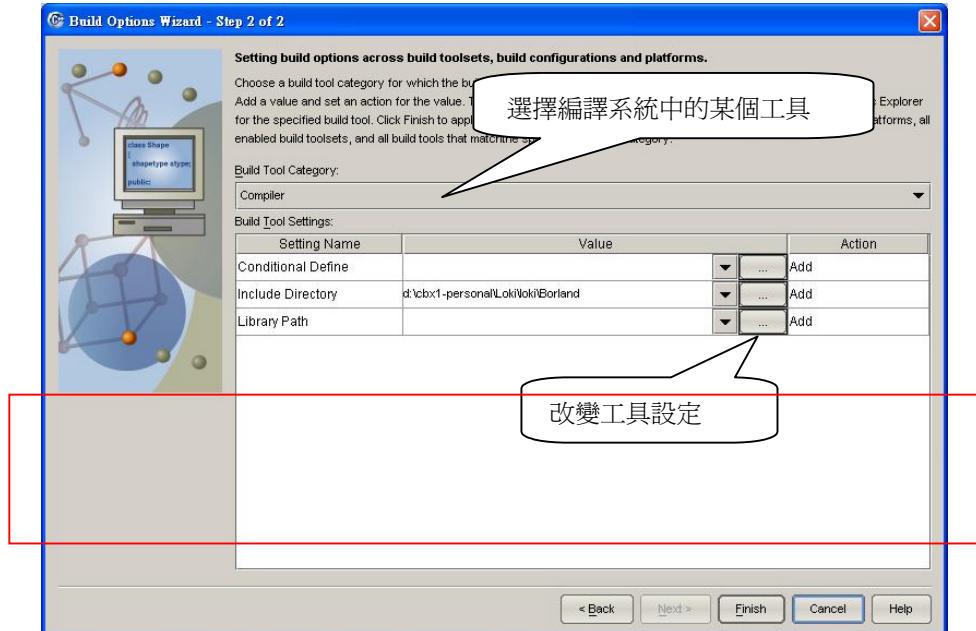


圖 10 / Build Objects Wizard 步驟 2，允許我們修改編譯系統的各種設定

二、入框架 (Frameworks) 坐落何處？

用得上就好了，何必知道位置？也許這是您的第一個想法。

容我告訴各位，CBX 所帶的這五大框架 ACE, Boost, Loki, STL, wxWindows 都是開放源碼（Open Source）社群的產物。雖然，Boost, Loki, STL 以源碼形式直接被含入您的應用程式中，而 ACE, wxWindows 以二進制碼形式和您的應用程式聯結在一起，但無論哪種情況，非常可能有一天您需要觀察它們的源碼 — 或許是為了技術學習，或許是為了除錯、或許是為了確認某些事物或細節。這種經驗對於 STL, Boost, Loki 使用者應該一點也不陌生（因為它們的規模相對較小，比較容易加入訂製型組件，也比較容易誘發觀看源碼的動機或產生觀看源碼的需要）。CBX 對框架（例如 STL）提供了相當多版本（以適應不同的平台和不同的編譯器），因此惟有清楚知道手上專案所用的框架源碼位於何處，才能避免將來想要觀看或修改程式庫源碼時亂點鴛鴦譜。

框架 (Frameworks) 是什麼？

人類不斷在軟體技術與工程上追求層次更高的復用性 (reusability)。從早期不帶參數的 subroutines，進化到帶參數的 functions/procedures，再到用戶自定的 structures，乃至資料和操作封裝於一體的 classes，以及型別參數化的 templates。這些進步固然都標示著某種里程，但都還是在極小的範圍內施展。

欲將軟體復用從小範圍物體擴展至更大範圍，一條可行之道是令程式主體反主為客，並令輔助元件（不論是 classes libraries 或 functions libraries）反客為主。惟有當這種「控制的反轉」發生，客端 (clients) 才能夠把「肩負應用程式主要運轉機能」的重擔卸下來交給程式庫，專注於解決領域問題 (domain problems)。當某些領域問題成熟到可以成為程式庫的一環，或程式庫技術發展到可以彈性抽換各種工作策略 (policies)，程式庫便又有更深層次的提昇。

在這種主客反轉的環境中，等於是把整個應用程式賴以運作的骨幹演算法全部由程式庫這一端寫好，而由復用這個骨幹的客戶端（應用程式端）自行填寫/衍生細節部分。所有物件導向語言都有適當支援此一性質的語言構件（在 C++ 中就是繼承機制與虛擬函式）。這種設計範式稱為 *Template Methods*（這裡的 *Template* 不是指 C++ templates，而是一種概念，請參考《Design Patterns》）。

如果某個程式庫負擔了這樣的主控功能，它本身其實已經是一個應用程式半成品 (semi-complete application)，因為其中有主動的事件迴圈、事件處理機制、控制流程。這樣的程式庫我們稱之為框架 (Framework)，例如 MFC, OWL, VCL, wxWindows。上述所舉這些框架都是負責架構 GUI 應用程式，所以又稱為 GUI 框架。但也有不以 GUI 為目標的框架，例如 ACE。

有些人主張「主動控制」並不成為框架的必要條件。他們認為只要是一組整合的、針對特定領域的結構和功能，並且能夠隨著客端的需求而由客端自行適配(調整)，這樣的程式庫也可以稱為框架。在這種條件下，Loki, Boost, STL 也都可以算是框架，而有別於此種框架的前述那種主動控制的「應用程式半成品」就被稱為「應用程式框架」 (Application Frameworks)。本文有時會稱它們為 App 框架。

C++標準程式庫在那兒？

第一個最被關切的應該是 C++ 標準程式庫 — 除非你是摩登原始人，否則寫作 C++ 程式少不了標準程式庫。這個程式庫以 STL 為骨幹，所以有時候被誤稱（或簡稱）為 STL (Standard Template Library)。為求簡便以下便以 STL 名之。它主要提供大量現成而高品質的資料結構（容器）與演算法；配接器（adapters）、迭代器（iterators）和仿函式（functors）則是用以烘托資料結構（容器）與演算法兩大明星的小雫組件（見圖 11）。

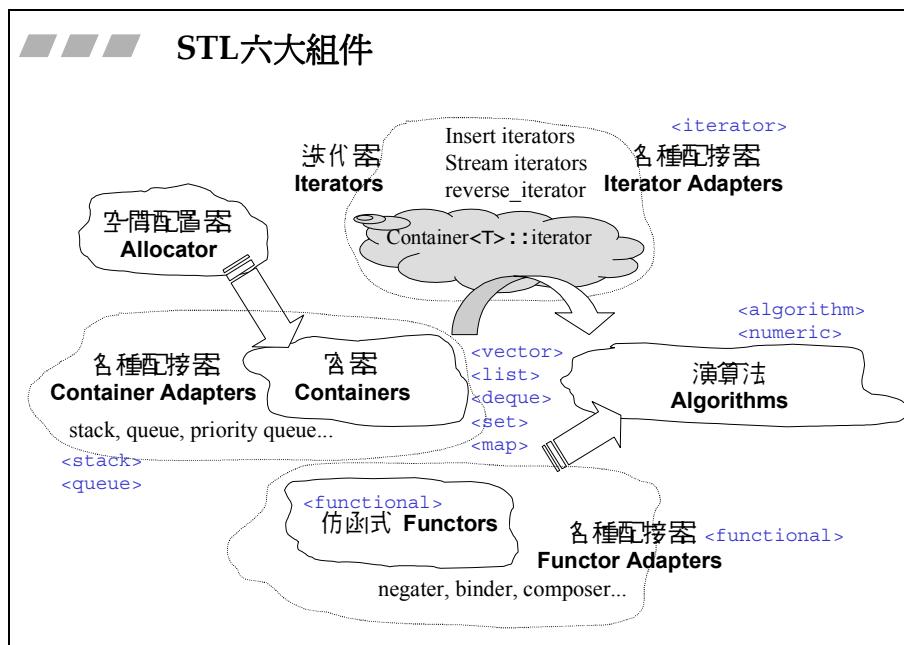


圖 11 / STL 基本示意圖（摘自《STL 源碼剖析》by 侯捷, 茉峰 2002）

STL 有各種版本，都源於 HP 的 GPL 授權，目前較知名的有 SGI, RW, PJ, STLPort 等版本。一旦您在 CBX 中產生任何專案，便可直接在其中使用 STL：只要 #include 相關表頭檔便是了，不需任何額外的編譯設定（圖 3 展示了這種情況）。由此可見，STL 一定位於 \CBuilderX\include 之中（圖 12）。

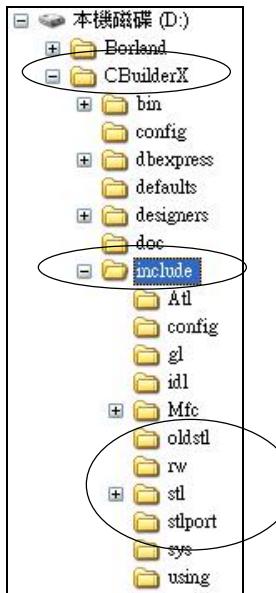


圖 12 / CBX 提供了這麼多 STL

為了解這麼多 STL 相關子目錄的組織與相關關係，讓我們進行抽絲剝繭的工作。雖然 C++ Standard 規定，標準表頭檔就像圖 3 例中所含入的`<vector>` 那樣，沒有副檔名，但 \CbuilderX\include 之內卻找不到任何一個「無副檔名」的表頭檔。那麼程式到底含入了什麼東西呢？這其中並不是發生靈異現象，而是 bcc32 編譯器自動做了預處理，一看見程式中使用`<vector>`，便視之為`<vector.h>`。於是，我打開 \CbuilderX\include\vector.h 瞧瞧，發現其中有這樣的預處理判斷式：

```
#ifdef _USE_OLD_RW_STL
#include <oldstd\vector.h>           (說明：使用 RW STL)
#else
#include <stlport\vector>            (說明：使用 STLPort)
#endif

#if !defined (__USING_STD_NAMES__) && defined (__cplusplus)
using namespace std;    (說明：這就是為什麼您在 CBX 或 BCB 中不必寫此行的原因)
#endif                 (續上：這種代勞有可能抹殺初學者正確的編程習慣及背後的正確觀念)
```

而`<stlport\vector>`（內容很短）又定義：

```
#ifndef _STLP_INTERNAL_VECTOR_H
#include <stl/_vector.h>   (說明：這裡才有真正的 vector 容器的宣告和定義)
#endif
```

這就水落石出了：如果您的程式沒有定義 _USE_OLD_RW_STL（預設情況下的確沒有定義它），那麼便使用 STLPort，否則使用 RW STL。事實上這種安排從 BCB6 以來便如此。[圖 13](#) 對眾多可能出現 STL 源碼的地點做一個整理。

STL 相關目錄	內容簡介	說明
\CBuilderX\include\	這裡有很多 STL 表頭檔，都帶有副檔名.h，例如 vector.h	只是扮演橋樑，分派至 oldstl 或 stlport 。
→ \CBuilderX\include\oldstl		RW STL 實際內容所在
→ \CBuilderX\include\stlport	絕大多數表頭檔都沒有副檔名，例如 vector。	STLPort 的橋樑，用以含入 stl 內的表頭檔。
→ \CBuilderX\include\stl	絕大多數表頭檔都有副檔名.h，以及前綴底線，例如 _vector.h。	STLPort 實際正內容所在
\CBuilderX\include\rw	檔案數量不多，例如沒有 vector.h 或 vector。	名稱上看起來是 RW STL，但實際上不知做何用途。

圖 13 / CBX 中可能出現 STL 源碼的眾多地點。請注意，雖然 CBX 用的是 STLPort 或 RW STL，但求助文件用的卻是 Dinkum C++ Library Reference（所謂 Dinkum 版本就是前面曾經提過的 PJ 版本）。

如果用的是 MinGW 編譯器，故事又有點不同了，見[圖 14](#) 的整理。

STL 相關目錄	內容簡介	說明
\CBuilderX\mingw\include	這裡並沒有 STL 表頭檔	
→ \CBuilderX\mingw\include\c++\3.2	有許多 STL 標準表頭檔，但只是扮演橋樑，用以含入真正內容（見下欄）	經驗：我個人尚未弄清楚 mingw 如何將 STL 的 include 路徑設到這兒。
→ \CBuilderX\mingw\include\c++\3.2 之下的四個子目錄：backward, bits, ext, mingw32	stl_xxx.h。例如 stl_vector.h (SGI 版標準命名方式)	SGI STL 實際內容所在。

圖 14 / 在 CBX 中使用 MinGW 編譯器時，STL 的組織情況。

如果使用 preview (bccx) 編譯器，故事又有點不同，見圖 15 的整理。

STL 相關目錄	內容簡介	說明
\CBuilderX\Preview\include	這裡有很多 STL 表頭檔，區分為帶有副檔名 .h 如 vector.h 以及不帶副檔名如 vector 兩種。	帶.h 副檔名者，只是扮演橋樑，分派至 oldstl 或 stlport 。但由於 Preview 目錄之下並沒有子目錄 oldstl 和 stlport ，難道連至 CBX 目錄下（圖 13）的 oldstl 和 stlport 去嗎？如何連接過去的呢？ 不帶.h 副檔名者，有完整 STL 內容但無 GPL 聲明。 經驗：測試程式可成功編譯，但執行時失敗。

圖 15 / 在 CBX 中使用 bccx 編譯器時，STL 的組織情況。

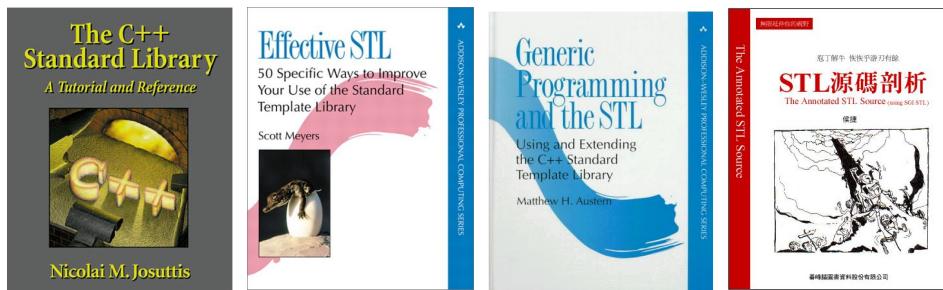
如果您使用其他編譯器，故事可能又不相同。本文不在這個題目上繼續討論下去，讀者可依個人需要自行探索。如果您想學習 STL 的使用乃至其實作，下面是四本不同層次的好書：

《The C++ Standard Library》 by Nicolai M. Josuttis, AW 1999 (有繁體中文版)

《Effective STL》 by Scott Meyers, AW 2001 (目前無繁體中文版)

《Generic Programming and the STL》 by Matthew H. Austern, AW 1998 (有繁體中文版)

《STL 源碼剖析》 by 侯捷, 暮峰 2002 (本身就是繁體中文版)



Loki 在那兒？

Loki 是一個高階 templates（模板）程式庫（就其格局而言，我們也可以說 Loki 是個框架），由 Andrei Alexandrescu 創造開發，主要提供多個設計範式（design patterns）之具體實現。本文論及的所有框架中以 Loki 規模最小，源碼只有 20 個 .h 和 2 個 .cpp，但其技術承載量非常高，在 C++ Templates 技術領域上是十分前衛的作品。Loki 極繁重地運用 policy（classes）技術，實作品涵蓋一個極重要的核心工具 Typelists（大量運用編譯期計算技術達到 metaprogramming）、一個專門負責小塊記憶體配置的 SmallObjAllocation、以及一個超迷您多緒程庫（只有空殼介面，並無太多實作內容），剩餘便是各自獨立的設計範式：Generalized Functors（泛化仿函式），Singletons（單件），Smart Pointers（靈巧指標），Object Factories（物件工廠），Abstract Factory（抽象工廠），Visitor（訪問者），Multimethods（多重方法），如圖 16 所示。

我們所面臨的問題是，Loki 被安放於「CBX 解壓縮目錄」而不是「CBX 安裝目錄」內（見圖 17），和 CBX IDE 之間並沒有任何關聯。做為應用程式的開發者，CBX IDE 的用戶，我們該如何讓 Loki 與 CBX 產生關聯呢？

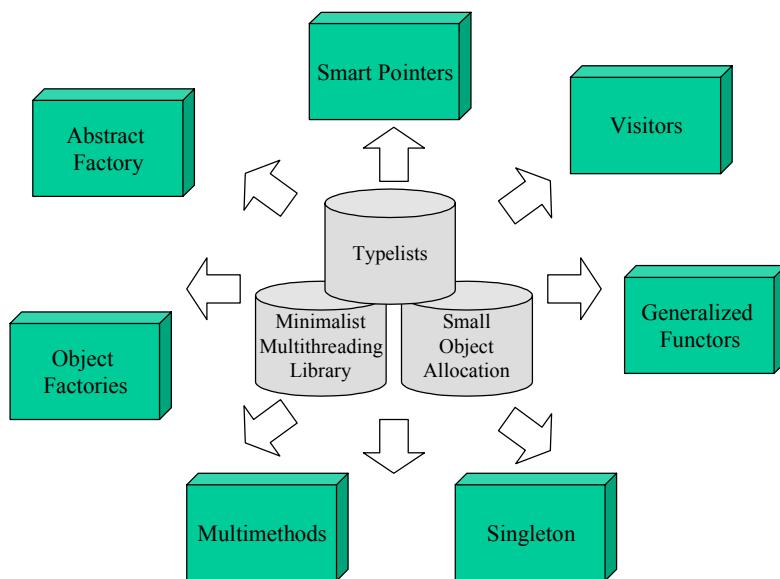


圖 16 / Loki 涵蓋圖中組件。灰色部分是核心（基礎）建設，四周綠色部分是對外介面（含實作）。

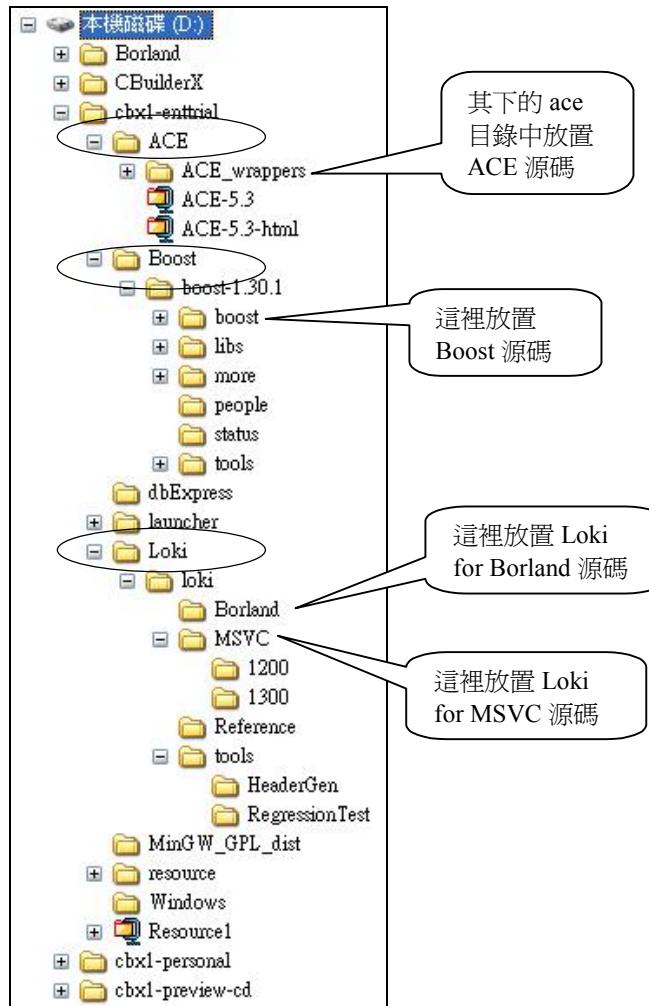


圖 17 / CBX 企業版的解壓縮目錄下有 Loki, ACE, Boost 等框架（的源碼）

產生關聯的作法非常簡單：只要在圖 10 的 Build Objects Wizard 步驟 2 中為編譯器的 Include 路徑加上一條：**D:\cbx1-personal\Loki\loki\Borland**（若是針對 MSVC 編譯器就要設為其他目錄，參考圖 17），編譯器就會自動尋找該目錄下的 Loki 檔案。我寫了一個用來測試 Loki SmallObj 的小程式如圖 18，其中所用的 Loki::SmallObjAllocator 專門負責配置小塊記憶體，比::operator new 執行速度更快，空間運用率更高，技術可與 SGI STL allocator 媲美。為了得知 Loki::SmallObjAllocator 內部對記憶體區塊的管理，我特地改動 Loki 的 SmallObj.h

源碼（如果您也想試著這麼做，請記得先備份），將其中許多成員變數的存取權限由 `private` 改為 `public`，這樣一來我才能夠在測試程式中列印出 `SmallObjAllocator` 的內部結構變化，進而畫出圖 19。限於篇幅，本文完全沒有機會談論圖 19 的結構，但對於《Modern C++ Design》的讀者，這張圖很可能使你對書中第四章有更深刻的體會。

```
#include "SmallObj.h"           // link with Loki's SmallObj.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
using namespace Loki;

// jjhou: 我改變了 "SmallObj.h" 中的成員存取層級，才得以在此觀察它們
void LokiAllocator_dissecting(SmallObjAllocator& myAlloc)
{
    cout << myAlloc.chunkSize_ << ' '
        << myAlloc.pool_.size() << endl;

    SmallObjAllocator::Pool::iterator ite1 = myAlloc.pool_.begin();
    SmallObjAllocator::Pool::iterator ite2 = myAlloc.pool_.end();
    for (; ite1!=ite2; ++ite1) {
        cout << ite1->blockSize_ << ' '
            << (int)ite1->numBlocks_ << ' '
            << ite1->chunks_.size() << ' ';

        cout << "allocChunk:" << ite1->allocChunk_ << ' ';
        cout << "deallocChunk:" << ite1->deallocChunk_ << ' ';

        FixedAllocator::Chunks::iterator ite3 =
                        ite1->chunks_.begin();
        FixedAllocator::Chunks::iterator ite4 =
                        ite1->chunks_.end();
        for (; ite3!=ite4; ++ite3) {
            cout << "current:" << &(*ite3) << ' ';
            cout << (int)ite3->firstAvailableBlock_ << ' '
                << (int)ite3->blocksAvailable_ << ' '
                << endl;
        }
    }

    class C : public SmallObject<>
    {
private:
    int x,y;
```

```
};

int main()
{
    SmallObjAllocator myAlloc(2048,256);

    void* p1 = (void*)myAlloc.Allocate(20);
    void* p2 = (void*)myAlloc.Allocate(40);
    void* p3 = (void*)myAlloc.Allocate(300); // > 256
    void* p4 = (void*)myAlloc.Allocate(64);
    void* p5 = (void*)myAlloc.Allocate(64);
    void* p6 = (void*)myAlloc.Allocate(64);
    void* p7 = (void*)myAlloc.Allocate(64);
    LokiAllocator_dissecting(myAlloc);

    void* ptr[100];
    for (int i=0; i< 65; ++i)
        ptr[i] = (void*)myAlloc.Allocate(64);
    LokiAllocator_dissecting(myAlloc);

    myAlloc.Deallocate(p5,64);
    LokiAllocator_dissecting(myAlloc);
    myAlloc.Deallocate(p6,64);
    LokiAllocator_dissecting(myAlloc);
    myAlloc.Deallocate(p4,64);
    LokiAllocator_dissecting(myAlloc);
    myAlloc.Deallocate(p7,64);
    LokiAllocator_dissecting(myAlloc);

    myAlloc.Deallocate(p1,20);
    myAlloc.Deallocate(p2,40);
    myAlloc.Deallocate(p3,300);
    for (int i=0; i< 65; ++i)
        myAlloc.Deallocate(ptr[i], 64);
    LokiAllocator_dissecting(myAlloc);

    cout << sizeof(SmallObject<>) << endl;
    cout << sizeof(C) << endl;

    C c1,c2,c3;
    C* pc1 = new C;      // SmallObject::operator new
    C* pc2 = new C;      // SmallObject::operator new

    LokiAllocator_dissecting(
        SingletonHolder<SmallObject<>::MySmallObjAllocator,
        CreateStatic,
        PhoenixSingleton
        >::Instance()
    );
}
```

```

    delete pc1;          // SmallObject::operator delete
    delete pc2;          // SmallObject::operator delete
}

```

圖 18 / Loki::SmallObjAllocator 測試程式

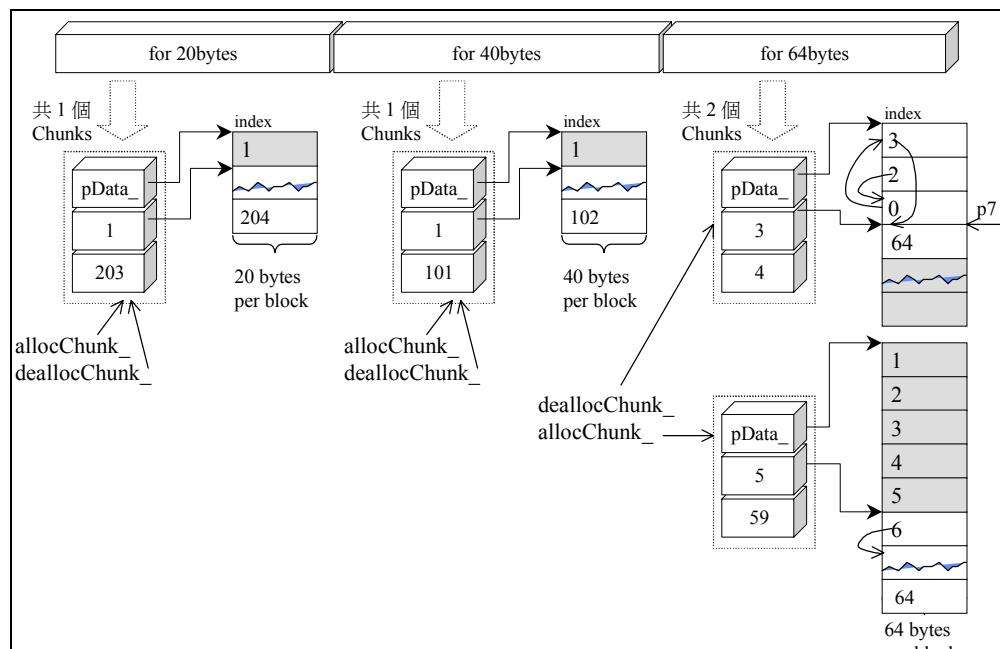
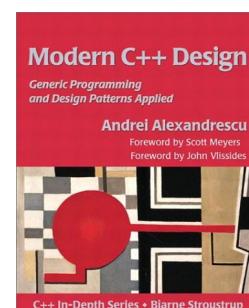


圖 19 / Loki's SmallObjectAllocator 的內部結構。Loki 的最高目標是以參數化的 classes（亦即 class templates）實現設計範式（design patterns），不過本圖的 SmallObjAllocator 並非某個設計範式，而是一個提供底層服務（小塊記憶體配置）的 class。

Loki 創造者 Andrei Alexandrescu 寫了一本名為《Modern C++ Design》的書籍，其中對 Loki 有十分詳細的介紹和技術剖析。

《Modern C++ Design》by Andrei Alexandrescu, AW 2001
(有繁體中文版)



Boost 在那兒？

Boost.org 由一群 C++ 標準委員會成員組織而成，目標在於開發出「可與 C++ 標準程式庫搭配」的可移植程式庫，進而希望爭取成為新版 C++ 標準程式庫的一部分。Boost 對 templates 的運用雖不比 Loki 那麼前衛，但也非常深刻而厚實，規模之大遠非 Loki 能夠望其項背。單就其中的 BGL (Boost Graph Library) 便已單獨出書（稍後介紹），Regular Express 也絕對具備一本書的份量。我甚至在 Boost-1.21.1 版看到 Persistence 的文件和子目錄（但尚未實現），那也是個大工程；只不過在 CBX 所帶的 Boost-1.30.1 版中它卻消失了。Boost 的整個涵蓋範圍可見其官方網站上的 Libraries 頁籤清單（如圖 24）

作為 Boost 用戶的我們，面臨的問題是：Boost 被安放於「CBX 解壓縮目錄」而不是「CBX 安裝目錄」下，和 CBX IDE 之間沒有產生任何關聯。該如何讓兩者產生關聯呢？

只要在圖 10 為編譯器的 Include 路徑加上一條：**D:\cbx1-personal\Boost\boost-1.30.1**（參考圖 17），並於應用程式中含入 `<boost/xxx.hpp>`，編譯器便會搜尋該目錄內的 "boost" 子目錄下的所有 Boost 表頭檔 *.hpp。請注意，所有 Boost 表頭檔 (*.hpp) 在 Boost-1.21.1 中全部預設為「唯讀」（點選右鍵目錄之「內容」項可改變之），但在 CBX 附帶的 Boost-1.30.1 中，其預設屬性全部取消了「唯讀」狀態。

圖 20 取自 Boost 自帶測試程式 rational_example.cpp，用以測試 boost::rational，那是負責處理有理數的一個簡單的 class（其實是處理由分母和分子構成的 "分數"；命名為 boost::fractional 應該是比較好的選擇）。圖 21 是其執行結果。

```
// rational number example program -----
// 這裡有一些注釋，限於文章篇幅，略而不列。

// 這裡有一些#include 和 using 指令，限於文章篇幅，略而不列。
#include <boost/rational.hpp>
using boost::rational;

#ifndef BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP
// 這裡有一些注釋，限於文章篇幅，略而不列。
```

```
using boost::abs;
#endif

int main ()
{
    rational<int> half(1,2);
    rational<int> one(1);
    rational<int> two(2);

    // Some basic checks
    assert(half.numerator() == 1);
    assert(half.denominator() == 2);
    assert(boost::rational_cast<double>(half) == 0.5);

    // Arithmetic
    assert(half + half == one);
    assert(one - half == half);
    assert(two * half == one);
    assert(one / half == two);

    // With conversions to integer
    assert(half+half == 1);
    assert(2 * half == one);
    assert(2 * half == 1);
    assert(one / half == 2);
    assert(1 / half == 2);

    // Sign handling
    rational<int> minus_half(-1,2);
    assert(-half == minus_half);
    assert(abs(minus_half) == half);

    // Do we avoid overflow?
#ifndef BOOST_NO_LIMITS
    int maxint = std::numeric_limits<int>::max();
#else
    int maxint = INT_MAX;
#endif
    rational<int> big(maxint, 2);
    assert(2 * big == maxint);

    // Print some of the above results
    cout << half << "+" << half << "=" << one << endl;
    cout << one << "-" << half << "=" << half << endl;
    cout << two << "*" << half << "=" << one << endl;
    cout << one << "/" << half << "=" << two << endl;
    cout << "abs(" << minus_half << ")=" << half << endl;
    cout << "2 * " << big << "=" << maxint
        << " (rational: " << rational<int>(maxint) << ")" << endl;
```

```

// Some extras
rational<int> pi(22,7);
cout << "pi = " << boost::rational_cast<double>(pi)
    << " (nearly)" << endl;

// Exception handling
try {
    rational<int> r;           // Forgot to initialise - set to 0
    r = 1/r;                  // Boom!
}
catch (const boost::bad_rational &e) {
    cout << "Bad rational, as expected: " << e.what() << endl;
}
catch (...) {
    cout << "Wrong exception raised!" << endl;
}

return 0;
}

```

圖 20 / Boost::Rational 測試程式

```

1/2+1/2=1/1
1/1-1/2=1/2
2/1*1/2=1/1
1/1/1/2=2/1
abs(-1/2)=1/2
2 * 2147483647/2=2147483647 (rational: 2147483647/1)
pi = 3.14286 (nearly)
Bad rational, as expected: bad rational: zero denominator

```

圖 21 / 圖 20 測試程式的執行結果。從中可看到分數（fractional）運算。

再舉一個例子。[圖 22](#) 是 Boost 自帶的一個 Type Traits 測試程式 copy_example.cpp；本例篇幅過大，我只節錄一小部分，揭示相關的運用語法。[圖 23](#) 是其執行結果。Type Traits 這種東西及其概念非常值得 C++ 程式員多多注意，否則將來可能哪一天你既看不懂高級程式員寫的程式碼，也沒有能力善用各種 C++ 框架的強大功能。所謂 Type Traits 是用來「在編譯期萃取 types（型別）特徵」的一種機制，例如用以測知某個 type 是否為 pointer？或是 class？或是 function pointer？或是 enum？是否有 non-trivial copy ctor？是否有 non-trivial dtor？...。其功能有點像 `typeof()` 運算子（當然我們都知道 C++ 沒有這麼一個運算子，也因此才需要動

用各種高階技術造出類似的東西加以彌補）。

```
/*
 * 這裡有一些聲明。限於文章篇幅，略而不列。
 * opt::copy - optimised for trivial copy (cf std::copy)
 */
// 這裡有一些 #include 和 using 指令。限於文章篇幅，略而不列。
#include <boost/timer.hpp>
#include <boost/type_traits.hpp>

namespace opt{
// opt::copy
// same semantics as std::copy
// calls memcpy where appropriate.

namespace detail{

template<typename I1, typename I2>
I2 copy_imp(I1 first, I1 last, I2 out) {
    // 註：限於文章篇幅，實作碼略而不列。
}

template <bool b>
struct copier {
    // 註：限於文章篇幅，實作碼略而不列。
};

template <>
struct copier<true> {
    // 註：限於文章篇幅，實作碼略而不列。
};

} // namespace detail

#ifndef BOOST_NO_STD_ITERATOR_TRAITS

template<typename I1, typename I2>
inline I2 copy(I1 first, I1 last, I2 out)
{
    typedef typename boost::remove_cv<
        typename std::iterator_traits<I1>::value_type>::type v1_t;
    typedef typename boost::remove_cv<
        typename std::iterator_traits<I2>::value_type>::type v2_t;
    return detail::copier
        <::boost::type_traits::ice_and
            < ::boost::is_same<v1_t, v2_t>::value,
            ::boost::is_pointer<I1>::value,
            ::boost::is_pointer<I2>::value,
            ::boost::has_trivial_assign<v1_t>::value
        >::value
    >::do_copy(first, last, out);
}
```

```
}

#ifndef BOOST_NO_STD_ITERATOR_TRAITS
    // 註：限於文章篇幅，此處略而不列。
#endif // BOOST_NO_STD_ITERATOR_TRAITS
}; // namespace opt

// define some global data:
const int array_size = 1000;
int i_array_[array_size] = {0,};
const int ci_array_[array_size] = {0,};
char c_array_[array_size] = {0,};
const char cc_array_[array_size] = {0,};
// 由於 arrays 不是 iterators 所以我們定義一組 pointer aliases 指向 arrays，否則編譯器
// 有權將被傳遞給 template functions 的型別推導為 T(&) [N] 而不是 T*。
int* i_array = i_array_;
const int* ci_array = ci_array_;
char* c_array = c_array_;
const char* cc_array = cc_array_;
const int iter_count = 1000000;

int main()
{
    boost::timer t;
    double result;
    int i;
    cout << "Measuring times in micro-seconds per 1000 elements
processed" << endl << endl;
    cout << "testing copy...\n"
    "[Some standard library versions may already perform this
optimisation.]" << endl;

    // 註：以下進行六種 copy 動作，觀察其花費時間。
    // (1). cache load:
    opt::copy(ci_array, ci_array + array_size, i_array);
    // time optimised version:
    t.restart();
    for(i = 0; i < iter_count; ++i) {
        opt::copy(ci_array, ci_array + array_size, i_array);
    }
    result = t.elapsed();
    cout << "opt::copy<const int*, int*>: " << result << endl;

    // (2). cache load:
    std::copy(ci_array, ci_array + array_size, i_array);
    // time standard version:
    t.restart();
    for(i = 0; i < iter_count; ++i) {
        std::copy(ci_array, ci_array + array_size, i_array);
```

```
}

result = t.elapsed();
cout << "std::copy<const int*, int*>: " << result << endl;

// (3). cache load:
opt::detail::copier<false>::do_copy(ci_array, ci_array +
                                         array_size, i_array);

// time unoptimised version:
t.restart();
for(i = 0; i < iter_count; ++i) {
    opt::detail::copier<false>::do_copy(ci_array, ci_array +
                                         array_size, i_array);
}
result = t.elapsed();
cout << "standard \"unoptimised\" copy: " << result << endl;

// (4). cache load:
opt::copy(cc_array, cc_array + array_size, c_array);
// time optimised version:
t.restart();
for(i = 0; i < iter_count; ++i) {
    opt::copy(cc_array, cc_array + array_size, c_array);
}
result = t.elapsed();
cout << "opt::copy<const char*, char*>: " << result << endl;

// (5). cache load:
std::copy(cc_array, cc_array + array_size, c_array);
// time standard version:
t.restart();
for(i = 0; i < iter_count; ++i) {
    std::copy(cc_array, cc_array + array_size, c_array);
}
result = t.elapsed();
cout << "std::copy<const char*, char*>: " << result << endl;

// (6). cache load:
opt::detail::copier<false>::do_copy(cc_array, cc_array +
                                         array_size, c_array);

// time unoptimised version:
t.restart();
for(i = 0; i < iter_count; ++i) {
    opt::detail::copier<false>::do_copy(cc_array, cc_array +
                                         array_size, c_array);
}
result = t.elapsed();
cout << "standard \"unoptimised\" copy: " << result << endl;
return 0;
}
```

圖 22 / Boost's TypeTraits 測試程式，節錄自 copy_example.cpp。

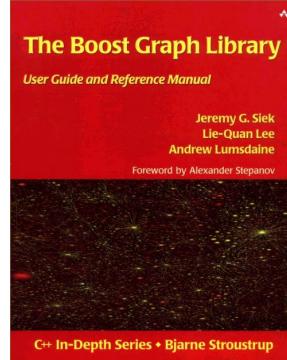
```
testing copy...
[Some standard library versions may already perform this
optimisation.]
opt::copy<const int*, int*>: 2.413
std::copy<const int*, int*>: 8.553
standard "unoptimised" copy: 8.241

opt::copy<const char*, char*>: 1.753
std::copy<const char*, char*>: 8.532
standard "unoptimised" copy: 8.162
```

圖 23 / Boost's TypeTraits 測試程式執行結果(測試平台:Windows XP, 1400MHz CPU, 512MB RAM)。從中可見使用一般作法 (std::) 和優化作法 (opt::) 所產生的效率差異。這在小資料量或操作不頻繁的情況下是無所謂的，但對於大公司或大型工廠的巨大數據，就有十分顯著的影響。

如果您要學習 Boost，目前並沒有全面描述的書籍，惟有 Boost 網頁（[圖 23](#)）是最全面而深入的文件。至於特殊主題如 BGL 已有專書出現（如下）。但願像 Regular Expression 這樣的主題也能很快出現專書。

《The Boost Graph Library》 by Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine, AW
2001 (目前無繁體中文版)



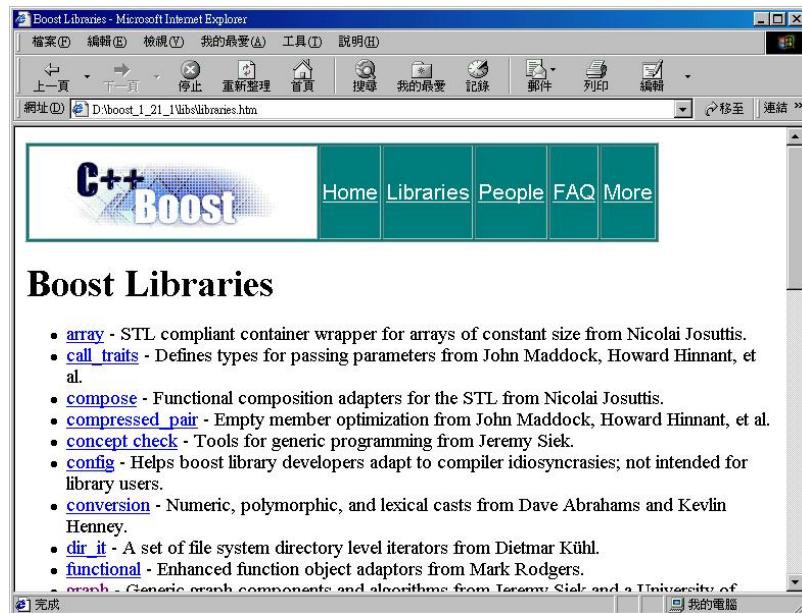


圖 24 / Boost 網站首頁

ACE 在那兒？

ACE 的全名是 ADAPTIVE Communication Environment，是一個「適配式（可調整）之通信軟體開發環境」。它是 Douglas C. Schmidt 於 1991 年啟動的一個源碼開放專案，原本是為了解決其博士論文實作過程中遇到的偶發複雜性。目前已發展成一個用於開發通信軟體的核心模型，可跨越多種平台完成通用通信軟體的各種應有任務，包括基本網絡組件如各式各樣的 wrapper facade，及高階的 Event Demultiplexing and Dispatching, Connection Management, Service Configuration, Concurrency, Stream-Processing...等等。ACE 是大量網絡編程專家的經驗的具體化實現，而且是「開放源碼」社群的一員，內含大約 25 萬行 C++ 程式碼，500 個 classes，規模非常龐大。它本身其實已經是一個應用程式半成品（semi-complete application），因為其中有主動的事件迴圈（event loops）、事件處理機制、控制流程。CBX 所帶之 ACE 為 5.3 版。

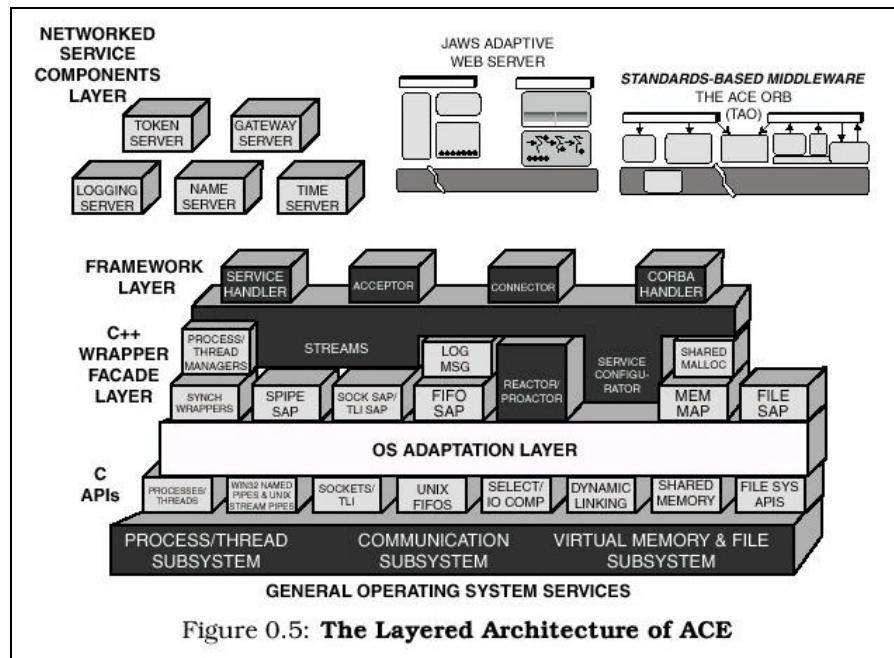


圖 25 / ACE 的分層架構。本圖取自《C++ Network Programming》v2, chap1, p15。

我們面臨的問題是，ACE 被安放於「CBX 解壓縮目錄」而不是「CBX 安裝目錄」中，和 CBX IDE 之間沒有產生任何關聯。該如何讓兩者產生關聯呢？

首先得清楚一點，ACE 非常龐大，它不是以表頭檔（源碼）形式來協助 ACE 應用程式開發，而是以二進制碼的形態來協助。因此我們需要做出（編譯出）一個 ACE 程式庫。請參考圖 17 列出的 ACE 主目錄，其中的 ACE\ACE_wrappers（內含 ACE 源碼）內有個 ACE_INSTALL.HTM，其內容很清楚地告訴我們三個步驟：

(1) 首先必須在 ACE\ACE_wrappers\ace 內添加一個 config.h，內含單獨一行：

```
#include "ace/config-win32.h"
```

(2) 啓動一個 DOS 視窗，在其中設定環境變數：

```
set ACE_ROOT = d:\cbx1-enttrial\ACE\ACE_wrappers
set BCBVER=5
```

(3) 進入 ACE_ROOTS\ace 目錄，執行以下命令建立起 release DLLs for ACE (命令中所用的 make.exe 位於 D:\CBuilderX\bin)：

```
make -f Makefile.bor -D... (可選) -DINSTALL_DIR=d:\ACETAO install
```

(預設產生 Dynamic Link 版本，除非定義 -D STATIC=1)

(預設產生 non-Debug 版本，除非定義 -D DEBUG=1)

請注意，由於是在 DOS 視窗而非整合環境中進行編譯，因此需要先在 DOS 視窗中設好編譯器所需的 Lib, Include, Path 等環境變數。編譯過程會新建許多.obj 檔：

```
d:\cbx1-enttrial\ACE\ACE_wrappers\ace\obj\ace\Dynamic\Release\*.obj
```

並新產出：

```
d:\cbx1-enttrial\ACE\ACE_wrappers\ace\Qos\obj\*.obj (暫態檔，可刪除)  
d:\cbx1-enttrial\ACE\ACE_wrappers\ace\RMCast\obj\*.obj (暫態檔，可刪除)  
d:\ACETAO\bin 有 ace_b.dll, ... (另兩個 dlls for Qos 和 RMCast)  
    \lib 有 ace_b.lib, ... (另兩個 libs for Qos 和 RMCast)  
    \include (都是舊檔案，由 ACE 的表頭檔複製過來)
```

注意：如果先前 make 時沒有設定 -DINSTALL_DIR=...，那麼上述的 dlls 和 libs 便都會被置於 d:\cbx1-enttrial\ACE\ACE_wrappers\bin\Dynamic\Release。

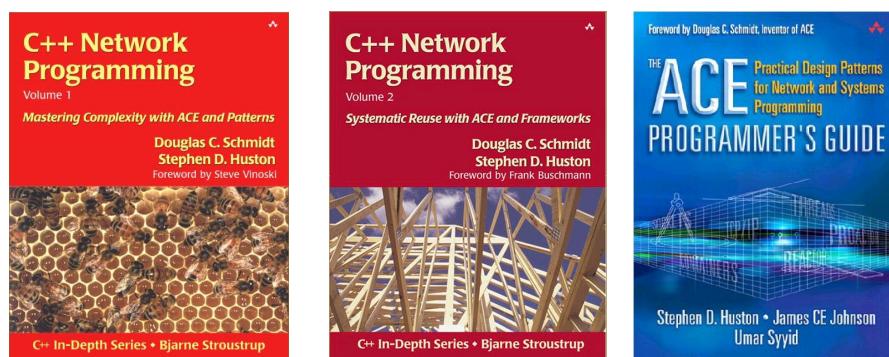
建立起 ACE 二進制程式庫後，欲以它為環境基礎，開發你自己的應用程式時，只需在編譯聯結之前為 "Include" 環境變數增設 `=$ACE_ROOT` (當然啦這麼一來應用程式欲含入 ACE 表頭檔時必須寫為 `#include "ace/xxx.h"` 而不能只是寫成 `#include "xxx.h"`)，並為 "Lib" 環境變數增設 `=d:\ACETAO\lib`。ACE 應用程式執行時，環境變數 "Path" 必須涵蓋路徑 `d:\ACETAO\bin` 以便讓系統找到 DLLs。

下面三本專書可幫助您學習 ACE 的架構及運用，其中前兩冊是 ACE 創造者 Douglas C. Schmidt 參與的作品。

《C++ Networking Programming - Mastering Complexity with ACE and Patterns》volume1,
by Douglas C. Schmidt, Stephen D. Huston, AW 2001 (目前無繁體中文版)

《C++ Networking Programming - Systematic Reuse with ACE and Frameworks》volume2,
by Douglas C. Schmidt, Stephen D. Huston, AW 2002 (目前無繁體中文版)

《ACE Programmer's Guide, The Practical Design Patterns for Network and Systems Programming》, by Stephen D. Huston, James Johnson, Umar Syyid, AW 2003 (目前無繁體中文版)



wxWindows 在那兒？

wxWindows 是個 GUI 框架，同時也是個 App 框架，地位相當於大家耳熟能詳的 MFC 或 OWL 或 VCL。這是一個開放源碼專案，也是 CBX 捨棄 C++Builder 所帶的（與 Delphi 共用的）VCL 後的新選擇。CBX 所附的是 wxWindows Beta 2.5 版，不過本稿撰寫期間，wxWindows 官方網站（[圖 26](#)）已提供正式 2.5 版。

眾所周知，換一個 App 框架便是換整個 App 骨幹。因此新工具無法回溯相容於老產品，在在困擾著老客戶。VC++身上新式的.NET Framework 之於老式的 MFC，CBX 身上新式的 wxWindows 之於老式的 VCL，莫不如此。站在老客戶立場，其實已經無所謂慣用的 App 框架是否缺陷重重、其他 App 框架是否可帶來氣象一新的技術或廣告宣稱的驚人效益，因為老程式員首先考慮維護問題，其次他們已經把心力目光放在框架之外的領域競爭力上；框架的一點點技術落後（如果真是這樣的話）已經不會繫絆他們的思維或實作。但畢竟軟體開發世界永遠有新手進來，新手是沒有包袱的一群人。就這個角度去想，開發工具廠商每隔數年就來一次組成大換血，也就不無可以理解的地方了。再者 CBX 把自己的競爭力定位於「可生成多種平台原生碼」（因為它有多種編譯器可供選擇），因此非需要換上一個跨

平台的 App/GUI 框架不可。

無論如何，世界又一次改朝換代，現在（在 CBX 中）是 wxWindows 的世界了。



圖 26 / wxWindows 官方網站 www.wxwindows.org。

欲實作一個 wxWindows 應用程式，最理想的方式是進入圖 2 畫面，點選其後所說的「種類五」專案。當你產生出這種專案，建置（*build*，亦即編譯聯結）時可從圖 3 的訊息窗口中看到 IDE（整合環境）自動為編譯器設定了以下的 Include 路徑：

```
-ID:\CBuilderX\include;
D:\CBuilderX\wxWindows\include;
D:\CBuilderX\wxWindows\lib\bcc_lib\mswd
```

並自動為聯結器設定以下的 Lib 路徑：

```
-LD:\CBuilderX\lib;
D:\CBuilderX\wxWindows\lib\bcc_lib
```

以及自動聯結以下程式庫（全都位於圖 26 所示的 wxWindows\lib\bcc_lib 中）：

```
wxmsw25d.lib (42,990KB)
wxexpatd.lib (312KB)
wxregexd.lib (75KB)
```

圖 27 是位於 CBX 安裝目錄下的 wxWindows 目錄組織。

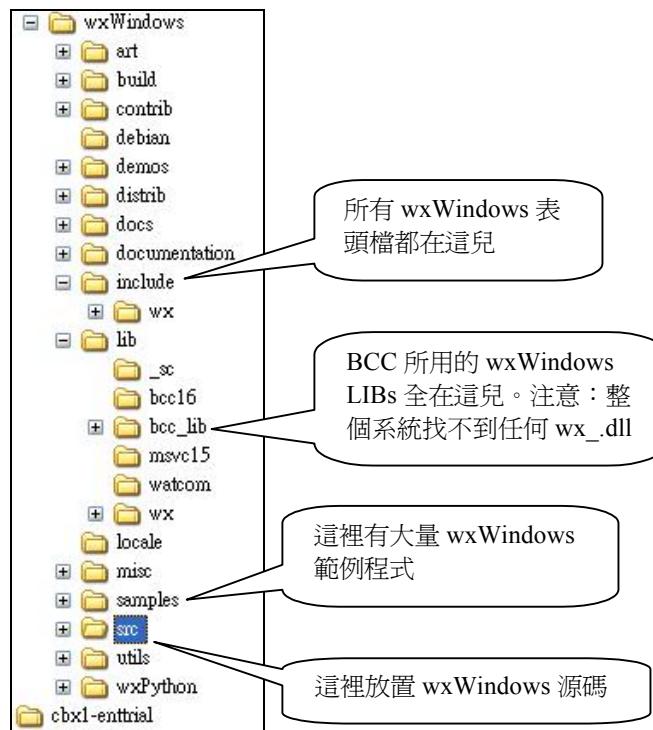


圖 27 / CBX 安裝目錄下的 wxWindows 目錄組織

本篇文章沒有任何篇幅空間允許我們談談 wxWindows 應用程式的編程模型，因為其中的任何一點點都是大工程。我在 amazon.com 網站上沒有發現任何一本討論 wxWindows 的書，或許直接從官方網站的文件開始你的學習之路，是惟一可行的辦法。`\CBuilderX\wxWindows\docs\html\index.htm` 列有許多範例程式（實際程式碼放在`\CBuilderX\wxWindows\samples`中），也是相當好的學習資源。

回顧

本文介紹了 CBX 的下載與安裝、CBX 支援之多編譯器組態與設定、CBX 所附五大框架的檔案組織與編譯選項設定。並推薦各個框架的學習資源（書籍與網站）。

-- the end