

導讀

侯 捷

這一部份闡述我（譯者）對於《C++ Primer 中文版》的製作理念。由於此書採用許多原文術語，因此本篇後段整理出各術語的中英文對照意義，並利用簡短的程式片段，解釋各術語的技術意義。

翻譯風格

首先讓我援引一篇不久前發表的文章，說明我對此書的翻譯態度。此文發表於網路 CompBook 論壇，並收錄於【侯捷網站/電腦散文 1999】之中。文中的時態與所有論述主題皆予保留，以為紀念。

【技術引導乎 文化傳承乎】 1999.06.28 第一次發表

C++ Primer 3/e 的翻譯工作已達 92%。預計 1999.07.05 完成初稿。完成後，我要把它雷射輸出，雙面影印，裁切裝訂成一本書的樣子。做好樣書之後，打算每天帶著它到光明新村湖邊，校對修潤。不想待在書房做這件事，眼睛真的需要休息了。湖光翠影，正好撫慰我可憐的雙眸。

校潤、重製、再校潤、再重製...，大約三次輪迴，這本書就可以付梓。打算在這上面花 25 天的時間。如果都如預期，那麼或許 08/01 便可交稿，或許 08/20 左右便可看到它上市，but no promise ☺

一個大困擾是：書這麼厚，怎麼辦？原書 1238+22 頁，一頁對譯一頁的結果，也是 1238+22 頁，再加上譯序和導讀，總共大約要到 1300 頁。若採用一般紙張，全書厚度肯定不低於 6.0 公分。用進口紙，時間內買不買得到是個問題（畢竟國內電腦出版界沒有先例，沒有經驗），成本也是個問題。我強力企圖說服出版公司用進口紙，但結果如何，實難預言。

很苦惱一本 4.0 公分的原文書被我做成巨無霸，而這還是一頁對譯一頁，內文用 9 號字體，程式碼用 8 號字體的結果。

真不希望看到它成為一塊破記錄的磚頭。

我很苦惱！

第一次嘗試

爲什麼要把書稿裁切裝訂做成樣書？首次嘗試這麼做，原因有三：

1. 這樣子才能夠感覺並掌握書籍出版後的真實模樣。左頁的確在左側，右頁的確在右側；邊緣留白是否合宜，天地是否足夠，都能夠因此有精準的感覺。
2. 自從電腦打字排版盛行，再沒有手稿這種東西了。自行裝訂的這三版初稿，上面將有我的修潤、圈點、眉批，算是給自己留個紀念。畢竟 *C++ Primer* 是最重要的譯作之一，是我自認爲譯筆逐漸成熟之後的重要作品，也將是我影響最廣的譯作。
3. 最重要的一點。經由最終形式的呈現，我要試驗如何在滿篇中英夾雜的文字上，以版面技巧讓讀者有良好的接受度。

第 3 點其實是我寫這篇文章的真正動機。

換句話說，這本《C++ Primer 中文版》，有許多英文術語出現。數量之多，前無古人！這些術語，不但包括名詞，還包括形容詞，以及極少數的動詞。只要是這個領域裡被朗朗上口的英文術語，我都儘量保留或中英並陳。（註：朗朗上口與否，其實見仁見智。後述）

最大的痛苦

若問我，翻譯 *C++ Primer* 的最大困難點與痛苦在哪裡，我說，對我而言，不是語文上的隔閡，不是技術上的難度，是「術語的處理」。

1993.12 年我在《Windows DDE 動態資料交換》一書序言中寫道：

關於電腦書籍中無法避免的英文字，一方面我希望專有名詞和術語儘量使用原文（英文），一方面卻不希望太多的英文充斥在字裡行間。兩難！本書的衝突更是明顯，原因是 DDE 的許多術語以及 DDE 函式名稱都十分冗長。我盡了我的能力去達到一個自己還算滿意的平衡程度。

六年過去了，這種兩難的情況在侯捷的寫作風格中益形擴大。

原文術語

通常一個人的做事風格，反應出他的成長經歷。我早年在研究單位、近年在高專教育、長年與資訊業界朋友互動，接觸的技術人員都慣用原文術語來溝通與思考，這是我希望儘量在書中保留原文術語的因素之一（當然，這就涉及一本書的定位，後述）。

習慣使用原文術語的技術人，我敢說，絕對不是不食人間煙火的少數族群。事實上，堅持把所有（或絕大部份）原文術語都轉爲中文來使用的人，在資訊業界恐怕才是不食人間煙火的少數族群。

我希望保留原文術語，並且保留區日益擴大，另一個因素是，在技術領域裡頭，沒有哪個資訊人不需要接觸原文（英文）資訊。你看了這本中文書、看了那本中文書，你早晚總還需要接觸英文書的。那麼，早早習慣世界一統的原文術語，有絕對必要性。

侯捷決定用個人的影響力來引領讀者。

初學者最好都不要接觸到英文？

我在其他的 web site 陸陸續續看到對我這種「保留原文術語」作法的評論。對我的任何批評，我都很歡迎。其中有一點很引起我的興趣：

目前在 [Run!PC](#) 上看到的 [C++ Primer](#) 片斷文章似乎也是如此處理法，就令人有些匪夷所思了，畢竟這是本入門書啊！看來英文不好真是罪惡。我心目中理想的譯本是，手邊不需要英漢字典一樣能看，英文程度不好一樣能懂的「中文」翻譯書。

我想談一談，書籍定位與原文術語的引用，有沒有什麼必然的關係。

我也想請大家思考一下，在專業領域中，原文術語的接受度和英文程度好不好有沒有什麼必然的關係。

一位初學者，踏入一個新的領域，他的困難不在英文單字，而在整句、整段文字所要表達的技術觀念。而且，一位初學者，並不一定是個大一學生，或專一學生，或高一學生，他也非常非常可能是另一個領域中經驗豐富的工程師。

不管初學者是什麼年齡，什麼身份，當他準備開始學習 C++，我絕對認為，不論是名詞如 `class`, `object`, `scope`，形容詞如 `virtual`, `local`, `global`，這些原文單字對他們都不構成困難 -- 只要他們知道這些名詞真正代表什麼意義。

我正在教導小學六年級的姪子學習 C++ programming，他對上述單字一點困難都沒有。`polymorphism`、`inheritance`、`encapsulation`、`reference`、`overloading`、`instantiated`、`exception`、`dereference`、`template`... 這些長單字對他也沒有困難。他並非英文資優，他是個平常的 12 歲小男孩，識得幾個寫程式時必要的英文單字，談不上具備怎樣的英文程度。遇上不懂的生字，我告訴他字面意義與技術意義，他就懂了；我告訴他這個字很重要，要用原文表達，他也就記下來了，下回朗朗上口。

是的，英文單字完全不會造成閱讀的困難 -- 只要讀者懂得單字背後的意義！

事實上我認為，C++ 初學者或許更需要在一開始就熟悉原文術語，就接觸行內話，並習慣使用原文術語。原因很簡單，剛才我說了，在技術領域裡，沒有哪個資訊人不需要接觸原文（最大宗是英文）資訊。你看了這本中文書、看了那本中文書，總有一天你還需要接觸英文書（除非你很混，或是不想混了）。那麼，早早習慣並使用原文術語，有絕對必要性。

此外，關於 *C++ Primer* 的書籍定位，我也要提出我的看法：此書由於用詞多數還算淺顯，切入面也還算親和，所以初學者可以看；但許多 C++ 老手也都把它放在一旁當重要參考。所以我絕不會爲了初學者可能需要什麼樣特殊的對待（如果真有的話），而讓 C++ 老手看此書看得很彆扭。*C++ Primer* 的讀者群中，初學者一定比老手多嗎？我不這麼認爲。

新手有一天會變成老手，有一天會需要自己找原文書看。學生有一天會成爲工程師，有一天會需要和別人用「行內話」溝通。那時候再讓自己重新習慣原文術語？改變習慣是很痛苦的！我不希望如此，我也不希望我的讀者如此！

我永遠記得初次看到 `member function` 被譯爲「成員函式」時的錯愕與不適。這種譯法你覺得有錯嗎？沒有，它四平八穩。你有更好的譯法嗎？沒有，它幾乎無可取代。那麼你錯愕什麼？不適什麼？

我錯愕是因爲，把中文慣用的兩個名詞合在一起用，唸起來卻拗口。我不適是因爲，我的業界朋友沒有人這樣唸。

很多年過去了，甚至在我自己所著所譯的書中也不敢造次，蕭規曹隨地延用「成員函式」；甚至我已經能夠在閱讀到「成員函式」時不加思索地聯想「就是 `member function` 啦」，但是我時時思考解放的那一天。

我漸次地解放。從著作《多型與虛擬》，到譯作《COM 本質論》，到最新譯作《C++ Primer 中文版》，我漸次地解放。網友曾謂：

侯先生在保留原文字方面可真是不遺餘力。甚至連 `static`, `global` 這兩個平凡的單字都要保留。我的感覺是霧裡看花，有點不忍卒讀，看起來像是一堆殘破的英文字和中文字，組成一段奇怪的程式碼。我覺得很不妥，也許侯先生另有用意，可是我總以爲既然是中譯本何不來個大解放呢？讓中譯本看起來像中文書呢（而不是像程式碼）。

我是要解放，只是解放的方向與這位網友所想的恰恰相反 ☹。至於殘破與否，奇怪與否，恐怕是見仁見智。後述。

閱讀原文書的最大障礙

單字不是障礙，長句子才是！句子背後隱喻的技術觀念才是！

這便是一本技術書籍中譯的價值所在：

1. 將長句子轉化為本國語文的習慣語法(短句子當然也要轉。特別強調長句子,因為它難度高)。讓讀者快速掌握字面意義,全付心思得以放在技術層面的思考,而不是用在拆解句型或翻查字典。
2. 將上下文突兀處(對本國人而言)加以修潤、磨平。由於語文習慣的因素,有許多時候我們覺得外文累贅,嘮嘮叨叨,有許多時候我們又覺得外文的上下段連接性不夠,明明在講同一件事情,乍見之下卻以為是另一個起點。換句話說,外文用來牽引上下文脈絡的習慣,和中文有許多不同。此外,英文的 *which, one, the other, it, that...* 讓人乍見之下不知所指,那麼亦該明確修潤。
3. 以譯者的技術能力來撫平可能出現的閱讀上的坎坷崎嶇。上述兩點都有賴這一點(專業技術)為基礎。任何一個人如果不懂 C++, 無論如何也譯不出一本好的 C++ 書籍。就算他是翻譯博士,亦然。

我在翻譯 *C++ Primer* 的過程中,興之所至,隨手記錄了幾則例句。正好此處派上用場。摘錄幾則,請大家一觀。

C++ Primer 3/e p.443 (這一段是這裡面最簡單的)

A typedef name provides an alternative name for an existing data type; it does not create a new data type. Therefore, two function parameter lists that differ only in that one uses a typedef and the other uses the type to which the typedef corresponds are not different parameter lists.

C++ Primer 3/e p.559 (這一段需要的技術背景多些)

The process by which compound statements and function definitions exit because of a thrown exception in the search for a catch clause to handle the exception is called stack unwinding. As the stack is unwound, the lifetime of local objects declared in the compound statements and in function definitions that are exited ends.

C++ Primer 3/e p.688 (這一段很長)

The resolution of names used in the body of the member functions of a local class are first looked up in the complete scope of the class before the enclosing scopes are searched.

C++ Primer 3/e p.730 (這一句可得好好分析語法)

This is particularly inappropriate in a copy assignment operator that first frees a resource currently associated with the class in order to assign the resource associated with the class being copied.

C++ Primer 3/e p.849 (三個 *in which*, 真不好看)

As is the case with class types, in which a class definition must be provided in every file in which the class members are used, the compiler instantiates a class template for a particular type in every file in which this instantiation is used in a context that requires a complete class definition.

C++ *Primer 3e* p955（代名詞一堆，each, one, the other, it..., 不知誰指誰）

Here are the various eval operations. We've defined each to be inline. The evalAnd() and evalOr() operations execute the following steps. Each pops `_query_stack` (this takes two operations with the standard library stack class, recall: `top()` to get the element and `pop()` to remove it from the stack.) One or the other allocates either an `AndQuery` or an `OrQuery` object from the heap, passing it the object retrieved from `_query_stack`. Each passes the `AndQuery` or `OrQuery` object the count of any left and right parentheses the operator needs to output on displaying itself. Finally, each pushes the incomplete operator onto `_current_op` :

C++ *Primer 3/e* p1021（那麼多 'to'，真累人）

RTTI allow programs that manipulate objects as pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

這裡沒有呈現上下文，也沒有特殊字形的輔助，單單閱讀孤段，很不好做正確的理解。但是大家多少能夠體會一下這些句子對中國人的閱讀難度。有的是句子過長，倒裝句、連接子句過多；有的是代名詞一大堆，不知誰指誰，有的是隱含的技術過於深澀，沒有內行人指點迷津，不知伊於胡底。

所以，我說，閱讀的困難，不在單字，在於長句子和技術基礎。翻譯一本電腦技術書，最重要的工作，以我的理念而言，在於掃平這些障礙，不在於「建立本國文化」或是「讓它成為一本道地的中文書」或是「掃除英漢字典的需要」。

如果術語本身不是問題，為什麼不用中文術語？

我也想用中文術語，如果：

1. 中文術語統一
2. 業界人士都（或大部份）以這些中文術語溝通

可是這兩個條件都不成立！

另一個重要因素，讓我不厭其煩地再強調一次：

在技術領域裡，沒有哪個資訊人不需要接觸原文（最大宗是英文）資訊。你看了這本中文書、看了那本中文書，你早晚還要接觸到英文書的。那麼，早早習慣並使用原文術語，免得做二次翻譯。

化身（reference）、定字（literal）、執行所（apartment）、公寓式（apartment）、斷言（assertion）、迭代子（iterator）、配置者（allocator）、樣板（template）、模板（template）、範本（template）、資料成員（data member）、成員函式（member function）、成員樣板（member template）、類別樣板部份特製化（class template partial specialization）...這些術語其實譯得都有點道理。但當工程師都以原詞溝通，誰會在乎它們被譯成什麼呢？誰又會在乎有沒有更文，更雅，更貼切，更傳神的譯詞呢？

術語該怎樣譯，才文，才雅，才貼切，才傳神？某些時候這種討論是必要的（尤其是在文史藝術等方面），某些時候是不必要的（尤其是在給專業人士看的技術專業讀物上面）。以上所列術語，我都以為不必要。至於什麼時候是必要的，什麼時候是不必要的，其實也是見仁見智。後述。

一本書如果要給入門者看，它會在書中某處（或多處）對術語做出解釋，不必一定得靠「中文」術語才能達意。如果書是要給進階者看，我相信它的讀者應該更希望讀到原文術語，不必玩「我猜 我猜 我猜猜猜」的遊戲。

另一些原因

切莫以為我提倡使用原文術語，我就是滿口英文。不，除了專業術語之外，我不是一個脫口英語的人（我不喜歡如此，也沒這樣的能力）。

日常生活中我會上口的英文詞，大約只有名詞（特別是專有名詞）。我不喜歡把 `complain`, `ridiculous` 等不少人習用的動詞或形容詞穿插在日常用語中，那很怪異，我覺得（但是我有尊重別人習慣的習慣，所以我勉力去聽懂這些中英夾雜的日常對話）。

絕大部份我所採用的原文術語，也都是名詞。但是有些動詞或形容詞有必要讓讀者知道（這時候我會三不五時地讓它中英並列），原因是：

1. C++ 編譯器的錯誤訊息並沒有中文化。所以，萬一錯誤訊息中出現以下字眼：`unresolved`, `instantiated`, `ambiguous`, `override`，而寫程式的你卻不熟悉或不懂這些動詞或形容詞的技術意義，就不妙了。
2. 有些動作關係到 `library functions` 的名稱，而 `library functions` 並沒有被中文化（當然！）。例如 `insert`, `delete`, `sort`。當這些字眼出現，視出現時與 `library functions` 的關係強度而定，我可能會選擇使用英文術語。這些術語會突兀嗎？噢，我們不都隨口使用這些原文術語的嗎？
3. 如果術語關係到 C++ 關鍵字，為了讓讀者有最直接的感受與聯想，我一定用原文，例如 `static`, `private`, `protected`, `public`, `friend`, `inline`, `extern`。

版面像一張破碎的臉

無法避免地，大量中英夾雜的結果，就是版面的破碎。在一行 20 公分長的文字中，看到 10 公分以上的英文，而這本書卻又是一本中文書，恐怕任何人乍見之下都覺得怪怪的。

我的態度有二：

一. 莫可奈何。

為了實現我以為合宜的資訊技術翻譯方式，我必須犧牲版面的「全中文化」或「絕大部份中文化」。版面的破碎是否會造成生硬呢？見仁見智！譬如你唸這句（我臨時隨便想的）：

(a). `class A` 的 `private data members` 只能在 `class A` (及其 `friends`) 的 `member functions` 中處理，至於其 `protected data members` 則不僅是如此，還可以在 `class A` (及其 `friends`) 的 `derived classes` 中處理。

然後唸這句：

(b). 類別 `A` 的私有資料成員只能在類別 `A` 及其夥伴的成員函式中處理，至於其受保護的資料成員則不僅如此，還可以在類別 `A` 及其夥伴的衍生類別中處理。

感覺如何？或是你唸這句：

(a). `function template` 可被宣告為 `inline` 或 `extern`，宣告方式和一般函式相同。

然後唸這句：

(b). 函式樣板可以被宣告為行內或外部，宣告方式和一般函式相同。

誰比較生硬呢？業內人士會怎麼說這兩句？我相信很多人（至少我週遭的人）都會說 (a) 而不是 (b)。看書，不就是在心中默唸書中的句子嗎？如果你平常習慣以 (a) 的方式溝通，我相信你看到 (a) 句不會認為它生硬。

如果你是初學者，我要引導你用 (a) 的方式來說。如果你本就已經習慣 (a)，我的作法你大概會喜歡。如果你一向習慣 (b)，又不願接受 (a)，那麼，容我說聲抱歉，此書在敘述風格上不適合你。

二. 儘量以版面技術來控制視覺上的順暢。

我打算採用不同的字形，代表不同屬性的英文術語。語言關鍵字用一種字形、程式碼出現過的符號名稱用一種字形、運算子名稱用一種字形（是的，幾乎所有的運算子名稱我也都保留原文），特殊長術語用一種字形（例如 `function template`, `stack unwinding...`）。

這便是為什麼要製作樣書的原因，我要在書籍出版之前看到它出版之後的樣子，從而體會一下閱讀的感受。如果我說服不了我自己，我就說服不了任何人。（但我也知道，當我說服了我自己，我也無法說服所有的人）

以己度人?!（度，音 `ㄉㄨˋ`，四聲）

自從學到了「以己度人」這句成語，我就很喜歡拿來反省自己，也檢視別人。

以己度人，意思是拿自己的想法去衡量別人。

把自己的鞋子硬套到別人腳上，尺寸剛好是運氣（別人的運氣）；尺寸不合可就難過了（別人很難過）。

我們每個人或多或少都有以己度人的習慣。

以己度人是好是壞呢？有時候它是好的，有時候它是壞的。

寫書/授課，就是以自己的想法和經驗來揣度他人（讀者/學生）的學習狀況，以自己的學習經歷來分享給別人或供做借鏡。要講多深，要講多淺，講什麼例子，用什麼說明方式，全都是「以己度人」。用的好，字字句句敲到心坎裡，棒棒打中要害。讀者/學生如沐春風，大嘆何其幸哉，得聞斯言，得見斯人。

「以己度人」用過了頭，它就壞了。全然以自己的需求為需求，以自己的思考為思考，以自己的意志為意志，完全用自己的觀點來看待別人的作為，這就壞了。

前面我曾說過，只要是 C++ 程式設計領域裡被朗朗上口的英文術語，我都在《C++ Primer 中文版》儘量保留或中英並陳。問題是，什麼才是朗朗上口的英文術語？誰認定的？

所以，我的選擇跳脫不了「以己度人」的原始方式。我把我所習慣的和我週遭所接觸的，當成了選擇的標準。英文術語的取捨成了我翻譯此書的最大痛苦來源。

本文至此一共用了四個「見仁見智」，顯然，「原文術語的採用」這件事被我認定是不可能客觀的答案了，恐怕也沒有辦法察納雅言，皆大歡喜了。畢竟公有公的道理，婆有婆的道理！

沒有人能夠說出全國有：

百分之多少的人說 "object"，	百分之多少的人說「物件」；
百分之多少的人說 "class"，	百分之多少的人說「類別」；
百分之多少的人說 "scope"，	百分之多少的人說「範圍」；
百分之多少的人說 "global 變數"，	百分之多少的人說「全域變數」；
百分之多少的人說 "local 變數"，	百分之多少的人說「區域變數」；
百分之多少的人說 "friend"，	百分之多少的人說「夥伴」；
百分之多少的人說 "data member"，	百分之多少的人說「資料成員」；
百分之多少的人說 "member function"，	百分之多少的人說「成員函式」；
百分之多少的人說 "virtual function"，	百分之多少的人說「虛擬函式」；

侯捷批評別人為什麼用「配置者 (allocator)」，別人也一樣會批評侯捷何必用「具現化 (instantiated)」。

侯捷批評別人為什麼不用 `reference`，別人也一樣會批評侯捷為什麼捨 `pointer` 而用「指標」。

雖然我建議使用原文術語，但是其間的取捨（哪個該用原文，哪個已經普遍化到可以用中文），沒有辦法得到客觀的標準。我能夠做的，就是以己度人。我曾在最近密集地詢問朋友們對於術語的採用有什麼看法，後來也就放棄詢問了。為什麼？一樣習性的人才會聚在一塊兒，我問來問去，問到的還不就是差不多的看法。我如果問到一個人，他告訴我『喔，在班上（or 辦公室），我們都說成員函式而不說 `member function`』，我又會因此改變我對原文術語的執著嗎？

不會！

很重要的原因是，我不只爲了原文術語被廣泛使用而終於大膽地大幅度採用它，還有一個很主要的因素（我要說第四次了）：

在技術領域裡，沒有哪個資訊人不需要接觸原文（最大宗是英文）資訊。你看了這本中文書、看了那本中文書，你早晚還要接觸到英文書的（除非你很混，或是不想混了）。那麼，早早習慣並使用原文術語，免得做二次翻譯。

我在這個議題上的「以己度人」，是好是壞？我相信以我在業界與學界的經驗，以我的接觸面的廣泛程度，它會是好的，合宜的。

最後，只能說，在施（譯者）與受（讀者）之間：

1. 磁場要對（風格要合）
2. 施者要經驗豐富，才能做出合宜的引導
3. 受者願意接納施者的經驗或理念，彼此才能相看兩不厭。

我想影響誰

我有尊重別人習慣的習慣。

早說過，使用中文術語或原文術語，沒有對錯的問題，是風格和理念的不同而已。

所以，別人專注於別人認爲好的事情，侯捷專注於侯捷認爲好的事情。我不企圖說服整個電腦翻譯界，也不企圖說服任何一位從事技術寫作的朋友。

我只企圖說服我的讀者，接受這種習慣。

文化議題

有些朋友談翻譯，談呀談呀就談上了文化傳承的議題。

有些朋友對侯捷很愛護，愛呀護呀就希望侯捷能夠以中華文化爲己任。

科技翻譯和文化傳承有沒有什麼關係？我持保留態度。

不管科技翻譯和文化傳承有沒有關係，侯捷都不定位自己要在這方面「繼往聖，開絕學」，或是要「以天下興亡爲己任，繼個人死生於度外」。雖然我很喜歡中國古典文學，我也努力在比較輕鬆的文章中加上一些很中國味（甚至有章回小說味道）的用語，可是我對自己的定位很清楚。在科技翻譯上該扮演什麼角色，在科技著作上該扮演什麼角色，在技術書評方面該扮演什麼角色，我都胸有成竹。

我以爲，翻譯一本專業科技書，最重要的工作是省下讀者看原文書時拆解句型或翻查字典的時間，俾


使他們得以專心致志地研究文中的技術內涵。讓他們快速地（比閱讀原文快個五倍十倍地）進入書中領域並吸收其中知識。是為科技翻譯之功。

至於科技「生根」嘛（可能有人會提到這個詞），對，讓科技的內涵完全被國人吸收，就是生根。科技生根和科技術語中文化，怕是完全兩碼事兒。

譯無定法，法無定論

翻譯如果沒有變通，就不是再創作。那末，用機器來瓜代就可以了。

同樣的科技，同樣的主題，不同的讀者群，應該有不同的翻譯作法。像侯捷這種大幅度採用原文術語的作法，就不適合科普文章，也不一定適合其他技術層面或其他讀者群。

我因為（被動地）知道自己所著所譯的讀者對象，或者說，我因為（主動地）清楚界定了我的讀者對象，所以大膽地做了本文所述的決定。 1999.06.28 

原文術語的採用原則

我觀察人們對於英文詞或中文詞的採用，似乎有一個習慣：如果中文詞發音簡短（或至少不比英文詞繁長）並且意義良好，那麼就比較有可能被業界用於日常溝通；否則業界多採用英文詞。

例如，polymorphism 音節過多，所以意義良好的中文詞「多型」就比較有機會被採用。例如，虛擬函式的發音不比 virtual function 繁長，所以使用這個中文詞的人也不少。「多載」的發音比 overloaded 短得多，意義又正確，用的人也不少。

但此並非絕對法則，否則就不會有絕大多數工程師說 data member 而不說「資料成員」、說 member function 而不說「成員函式」的情況了。

以下是本書採用原文術語的幾個簡單原則，但是沒有絕對的實踐，因為有時候要看上下文情況。同時，請容我再謙卑地強調一次，這些都是基於我與業界和學界的接觸經驗，而做的選擇。

- 程式設計基礎術語，採用中文。例如：函式、指標、變數、常數。本書採用的英文術語，絕大部份都是與 C++/OOP 相關的英文術語。
- 簡單且朗朗上口的詞，不譯：input, output, lvalue, rvalue...
- 讀者應該認識的原文名詞，不譯：template, class, object, exception, scope, namespace...
- 長串、有特定意義、中譯名稱拗口者，不譯：explicit specialization, partial specialization, using declaration, using directive, exception specialization...

- 運算子名稱，不譯：`copy assignment` 運算子，`member access` 運算子，`arrow` 運算子，`dot` 運算子，`address of` 運算子，`dereference` 運算子...
- 業界慣用語，不譯：`constructor`, `destructor`, `data member`, `member function`, `reference`...
- 6, 17 兩章為範例而取的資料結構名稱，不譯：*word location map*, *solution set*, *location pair*...
- 牽涉到 C++ 關鍵字者，不譯：`public`, `private`, `protected`, `friend`, `static`,
- 有良好譯詞，發音簡短，用者頗眾的詞，譯之：多型 (*polymorphism*)，虛擬函式 (*virtual function*)
- 譯後可能失掉原味而無法完全彰顯原味者，時而中英並列。
- 重要的動詞、形容詞，時而中英並列：模稜兩可 (*ambiguous*)，決議 (*resolve*)，改寫 (*override*)

援用原文詞，或不厭其煩地中英並列，獲得的一項重要福利是：搭配著一頁對譯一頁的作法，本書得以最經濟方式保留原書索引。索引對於一本工具書的重要性，不言可喻。

中英術語對照（按字母順序排列）

以下術語，在本書之中，某些以英文呈現，某些以中文呈現，某些則中英並列。

<code>abstract</code>	抽象的
<code>abstraction</code>	抽象體、抽象物、抽象性
<code>access</code>	存取、取用
<code>access function</code>	存取函式
<code>address-of operator</code>	取址運算子 <code>&</code>
<code>algorithm</code>	演算法
<code>argument</code>	引數（傳遞給函式的值）。參見 <code>parameter</code>
<code>array</code>	陣列
<code>arrow operator</code>	<code>arrow</code> 運算子 <code>-></code>
<code>assignment</code>	指派
<code>assignment operator</code>	指派運算子 <code>=</code>
<code>associated</code>	相應的、相關的
<code>associative container</code>	聯合容器（對應於 <code>sequential container</code> ）
<code>base class</code>	基礎類別
<code>best viable function</code>	最佳可行函式（從 <code>viable functions</code> 中挑出的最佳吻合者）
<code>binding</code>	繫結
<code>bit</code>	位元
<code>bitwise</code>	「以 <code>bit</code> 為單元的...」。例 <code>bitwise copy</code>
<code>block</code>	區塊
<code>boolean</code>	布林值（真假值， <code>true</code> 或 <code>false</code> ）
<code>byte</code>	位元組（8 <code>bits</code> 所組成的一個單元）
<code>call operator</code>	<code>call</code> 運算子 <code>()</code> （與 <code>function call operator</code> 同）
<code>chain</code>	串鏈（例 <code>chain of function calls</code> ）
<code>child class</code>	子類別（或稱為 <code>derived class</code> , <code>subtype</code> ）

class	類別
class body	類別本體
class declaration	類別宣告、類別宣告式
class definition	類別定義、類別定義式
class derivation list	類別衍化列
class head	類別表頭
class template	類別範本
class template partial specializations	類別範本局部特製體、類別範本局部特殊化
class template specializations	類別範本特製體、類別範本特殊化
cleanup	清理、善後
candidate function	候選函式（在函式多載決議程序中出現的候選函式）
command line	命令列（在作業系統文字模式中，於系統提示號之後所下的整行命令）
compiler	編譯器
component	組件
concrete	具象的
container	容器（可存放資料的一種結構，例如 <code>list</code> , <code>map</code> , <code>set</code> ）
context	背景關係、週遭環境、上下脈絡
const	常數（ <code>constant</code> 的縮寫，相對於變數 <code>variable</code> ）
constructor (ctor)	建構式（與 <code>class</code> 同名的一種 <code>member functions</code> ）
data member	資料成員、成員變數
declaration	宣告、宣告式
deduction	推導（例： <code>template argument deduction</code> ）
definition	定義、定義區、定義式
dereference	提領（取出指標所指物體的內容）
dereference operator	提領運算子 <code>*</code>
derived class	衍生類別
destructor (dtor)	解構式
directive	指令（例： <code>using directive</code> ）
dot operator	<code>dot</code> 運算子 <code>.</code>
dynamic binding	動態繫結
entity	物體
encapsulation	封裝
enclosing class	外圍類別（與巢狀類別 <code>nested class</code> 有關）
enum (enumeration)	列舉（一種 C++ 資料型別）
enumerators	列舉元（ <code>enum</code> 型別中的成員）
equality operator	等號運算子 <code>==</code>
evaluate	評估、求值、核定
exception	異常情況
exception declaration	異常宣告（請參考 11.3 節）
exception handling	異常處理、異常處理機制
exception specification	異常規格（請參考 11.4 節）
exit	退離（指離開函式時的那一個執行點）
explicit	明白的、明顯的、顯式
export	匯出
expression	算式
facility	設施、設備
flush	清理、掃清
formal parameter	形式參數
forward declaration	前置宣告

function call operator	與 call operator 同
function object	函式物件 (12.3 節)
function overloaded resolution	函式多載決議程序
function template	函式範本
generic	泛型、一般化的
generic algorithm	泛型演算法
global	全域性的
global scope resolution operator	全域生存空間 (範圍決議) 運算子 ::
handler	處理常式
header file	表頭檔 (放置各種型別定義、資料結構、函式宣告的檔案)
hierarchy	階層體系 (base class 和 derived class 所組成)
identifier	識別符號
immediate base	直接的 (緊臨的) 上一層 base class。第 18 章
immediate derived	直接的 (緊臨的) 下一層 derived class。第 18 章
implement	實作
implementation	實作品、實作物、實作體、實作碼; 編譯器。
implicit	隱喻的、暗自的、隱式
increment operator	累加運算子 ++
inheritance	繼承、繼承機制
inline	行內
inline expansion	行內擴展
initialization	初始化 (動作)
initialization list	初值列
initialize	初始化
instance	實體 (常指根據 class 而產生出來的 object)
instantiated	具現化 (應用於 template)
instantiation	具現體、具現化實體 (應用於 template)
invoke	喚起
iterate	迭代 (迴圈一個輪迴一個輪迴地進行, 謂之)
iterator	迭代器 (一種泛型指標)
iteration	迭代 (迴圈中的每一次輪迴稱為一個 iteration)
lifetime	生命期、生命週期、壽命
linker	連結器
literal constant	字面常數 (例如 3.14159 或 12 這樣的常數值)
list	串列 (linked-list)
local	區域性的
lvalue	左值
manipulator	操縱器 (iostream 預先定義的一種東西. ref p.1119)
mechanism	機制
member	成員
member access operator	成員取用運算子 (有 dot 和 arrow 兩種)
member function	成員函式
member initialization list	成員初值列
memberwise	「以 member 為單元的...」。例 memberwise copy
most derived class	最末層的 derived class
mutable	易變的
namespace	命名空間
nested class	巢狀類別
object	物件

operand	運算元
operation	操作行為
operator	運算子
option	選項
overflow	上限溢位 (相對於 underflow)
overhead	額外負擔
overload	多載, 重載
overloaded function	多載化函式
overloaded operator	多載化運算子
overloaded set	多載集合
override	改寫。意指在 derived class 中重新定義 virtual function。
parameter	參數 (函式參數列上的變數)
parameter list	參數列
parent class	父類別 (或稱 base class)
parse	解析
partial specialization	局部特殊化, 局部特殊化定義, 局部特殊化宣告。16.10 節
pass by address	傳址 (函式引數的傳遞方式之一)
pass by reference	傳址 (函式引數的傳遞方式之一)
pass by value	傳值 (函式引數的傳遞方式之一)
placement delete	15.8.2 節
placement new	15.8.2 節
platform	平台
pointer	指標
polymorphism	多型
preprocessor	前置處理器、前處理器
programming	編程、程式設計、程式化
project	專案
qualified	經過資格修飾 (例如加上 class scope 運算子)
qualifier	資格修飾詞
raise	發生 (常用來表示發出一個 exception)
rank	等級、分等 (chap09, chap15)
raw	生鮮的、未經處理的
reference	C++ 之中類似 pointer (指標) 的東西。意義上相當於「化身」
represent	表述, 表現
resolve	決議。為算式中的符號名稱尋找對應宣告的過程。
resolution	決議程序、決議過程
rvalue	右值
scope	生存空間
scope operator	生存空間運算子 ::
scope resolution operator	生存空間決議運算子 (與 scope operator 同)
sequential container	循序容器 (對應於 associative container)
specialization	特殊化、特殊化定義、特殊化宣告 (16.9 節)
stack	堆疊
stack unwinding	堆疊輾轉開解 (此詞用於 exception 主題)
statement	述句
stream	資料流
string	字串
subscript operator	下標運算子 []
subtype	子型別 (亦即 derived class)

target	標的 (例 target pointer: 標的指標)
template	範本、模板、樣板
template argument deduction	範本引數推導
template explicit specialization	範本明白特製體、範本明白特殊化
template parameter	範本參數
text file	程式本文檔 (放置程式原始碼的檔案)
throw	丟出 (常指發出一個 exception)
token	語彙單元
type	型別
underflow	下限溢位 (相對於 overflow)
unqualified	未經資格修飾 (而直接取用)
unwinding	(請參考 stack unwinding)
variable	變數 (相對於常數 const)
vector	向量 (類似陣列 array)
viable	可實行的 (chap15, chap09)
viable function	可行函式 (從 candidate functions 中挑出者)
volatile	易揮發的

原書勘誤和修正

《C++ Primer 中文版》係根據 *C++ Primer* 第三版第一刷翻譯，並根據原書網站的勘誤資料加以修正 (截至 1999.08.10 為止)。最後再以 *C++ Primer* 第三版第二刷為本，重新檢閱。所有勘誤皆已直接修定於文內，並以譯註方式明白告示，以避免持有原文書之讀者錯愕。若只第一刷有誤，則註以「原書一刷有誤…」，若是一、二刷皆有誤 (印象中只發生於 p.228)，則註以「原書 1, 2 刷皆有誤…」。

如有新的中英文版勘誤資料，將於網上公佈，《C++ Primer 中文版》網址請見封底。

版面字形風格

- 一般文字：class, object, member function, data member, base class, derived class, private, protected, public, reference, template, namespace, function template, class template, local, global...
- 斜體文字 (常用於動詞或形容詞)：*resolve*, *ambiguous*, *override*, *instantiated*
- class 名稱、變數名稱、函式名稱：min(SmallInt*, int), Query, int, string, map...
- 長串術語：member initialization list, name return value, using directive, using declaration, pass by value, function try block, exception declaration, exception specification, stack unwinding, function object, function adaptor, class template specialization, class template partial specialization,...
- 特殊意義詞：範例程式資料結構名稱如 *word location map*, *location vector*, *location pair*, *solution set*, C++ standard library 內建的 *exceptions types*, *manipulators*, *iterator types*...

- 運算子名稱及某些特殊動作：copy assignment 運算子, dereference 運算子, address of 運算子, equality 運算子, call 運算子, constructor, destructor, default constructor, copy constructor, virtual destructor, memberwise assignment, memberwise initialization,...

在一本這麼厚的書籍中要維護一貫的字形風格，並不是太容易，許多時候不同類型的術語搭配起來，就形成了不知該用哪種字形的困擾。排版者顧此失彼的可能也不是沒有。因此，請注意，**各種字形的運用**，只是爲了讓您閱讀時有比較好的效果，其本身並不具任何其他意義。

範例程式注意事項

我曾以本書繁多的各個小型範例，在不同的 C++ 編譯器上試驗，發現各編譯器仍然不同程度地不相容於 Standard C++（其間細節瑣碎，不及備載）。讀者在練習書中範例時，宜有心理準備。

本書範例程式碼可自支援網站（網址附於封底）下載。欲編譯這些程式碼，請注意以下數點：

1. 程式檔的副檔名皆爲 .C。在大部份編譯器下可能需要改爲 .CPP。
2. 您可能需要爲許多程式加上一行 `using namespace std;`（其意義請參考 8.6.4 節），可加在所有 `#include` 指令之後。
3. 在我下筆此刻，幾乎所有編譯器都或多或少尚未完全支援 Standard C++，尤其在 `template` 進階部份。所以如果您在相關領域遇到編譯上的挫折，不足爲奇。不過，閱讀本書之後，您可試著改寫那些有問題的碼，使能夠在現有的編譯器上順利通過。往好處想，這或許成爲檢驗功力的一個練習。

關於索引

本書和英文版頁次完全對應，並保留大量 C++/OOP 英文術語，所以英文版索引可直接應用於本書。本書索引以英文爲主，中文爲輔；中文的出現是爲了說明某些術語在本書中的譯名。此份索引完全未減去任何文字，其內所有中文皆爲增補，而非改寫。

由於英文索引皆以極簡潔之用詞如 `for`, `as`, `with`, `by`, `of` ... 表達上下層索引關係，不適合譯爲中文，故本份索引的中文補註絕大部份僅及於上層索引名詞。或有不夠週延之處，實爲多方考量下的權宜結果。

本書英文版的前兩個刷次（1st printing 和 2nd printing），在索引上有一些變化。然而兩個刷次的版面安排事實上完全沒有改變，索引的變化只是反應出索引結構的小量調整。本書以原書第一刷之索引爲編寫對象。另，作者 Lippman 的個人網站（<http://people.wmediaone.net/stanlipp/index.html>）又提供一份新的索引，有興趣的讀者可自行上網下載。

重要術語的技術意義（各章重點整理與導讀）

初次面對本書英文術語而欲瞭解其意義，一個方法是查閱書後索引，一個方法是查閱本節。此處以各章出現次序，將能夠以簡短程式碼形式表現意義的重要術語，列出於後。內容挑選標準以 C++/OOP 的新術語為主。本處之選擇及解說或未臻完備，純粹只是為了提供一個快速導引。

讓我從第 6 章開始。

第 6 章 Abstract Container Types（抽象容器型別）

這一章發展出一個不小的程式，運用由 C++ Standard Library 供應的數個 container 型別，完成一個文字檢索系統。這個系統將在第 17 章接續發展。由於程式份量頗重，此處以圖形呈現其資料結構，可協助你快速掌握重點。這些資料結構亦通用於第 17 章的文字檢索系統。

整個程式被包裝在一個大型 class `TextQuery` 之中（這是一種 *object-based* 風格，不是 *object-oriented* 風格）。其中的主流程是 `doit()` 函式：

```
void TextQuery::doit()
{
    retrieve_text();
    separate_words();
    filter_text();
    suffix_text();
    strip_caps();
    build_word_map();
}
```

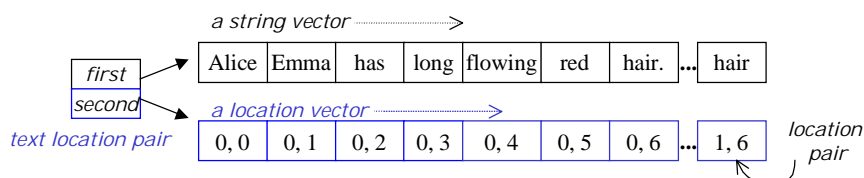
這六大動作是：

- `retrieve_text()`：讀檔案，將每一文字行讀進一個 *string* 內，形成一個 *string vector*。

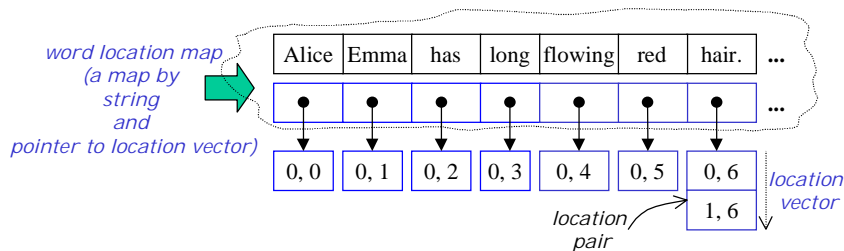
string vector

Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

- `separate_words()`：將上述所有文字行拆解成一個個單字，統統放進一個 *string vector* 內，每一個 *string* 容納一個單字。並計算每個單字對應的位置（行號與列號），放進所謂的 *location vector* 內。最後再以一個所謂的 *text location pair* 指向這兩個 *vector*。



- `filter_text()`：過濾單字中的標點符號。
- `suffix_text()`：過濾長度小於 4 的單字、處理單字尾辭。
- `strip_caps()`：單字全部轉為小寫。
- `build_word_map()`：產生一個 *word location map*，以單字為鍵值，以單字對應的 *location vector* 的指標為實值。處理過程中去除語意中性的單字（如 *the, but, and, its, into...*），並將重複單字的位置記錄於對應的 *location vector* 內（內放行號與列號，也就是一個 *location pair*）。



例如，下面這些於文件中重複出現的單字，其對應的 *location vector* 內容是：

bird	((2,3),(2,9))
daddy	((0,8),(3,3),(5,5))
fiery	((2,2),(2,8))
hair	((0,6),(1,6))
her	((0,7),(1,5),(2,12),(4,11))
him	((4,2),(4,8))
she	((4,0),(5,1))
tell	((2,11),(4,1),(4,10))

有了這份 *word location map* 之後，任何單字的檢索動作就可以輕鬆完成了。只要搜尋此一 *map* 的 *string vector*，找到目標後即可根據其對應的 *location vector*，列出其中記錄的各個位置（單字出現位置）。以下是 `class TextQuery` 的運用模式：

```
int main()
{
    TextQuery tq;
    tq.doit();           // 處理整份文件，建立起必要的資料結構。
    tq.query_text();     // 詢問單字，進行檢索
```

```
    tq.display_map_text();    // 顯示檢索結果
    return 0;
}
```

第 7 章 函式 (functions)

```
// 以下是參數原型 (function prototype)，亦是所謂函式宣告 (function declaration)
// 亦或稱為識別型式 (function signature)，其中包含回返值型別、函式名稱、參數列 (parameter list)
// 小括弧內即是參數列 (parameter list)，其中第三個參數有預設引數 (default argument)
int foo( int& a, int* b, int c = 3 );

// 以下是所謂參數定義 (function definition)，小括弧內稱為函式本體 (function body)
int foo( int& a, int* b, int c)
{ /* ... */ return 0; }

// 以下是一個函式呼叫動作 (function call)，小括弧內即是引數列 (argument list)
// 引數 a 是 pass by reference (通常亦譯為傳址，與 pass by address 混用)
// 引數 b 是 pass by address (通常譯為傳址，與 pass by reference 混用)
// 引數 c 是 pass by value (傳值)
int a, b, c;
int i = foo( a, &b, c );
```

第 8 章 生存空間與生命週期 (scope and lifetime)

```
// 所謂 scope 係用來區分符號名稱的意義及其前後背景關係。
// C++ 支援三種 scopes: local scope, namespace scope, class scope
// 分別由函式內的局部區塊 (以 { } 標示)、namespace、class 形成。

// 以下是 namespace (命名空間) 的定義形式，其中可以放置任何述句 (statements)
namespace International_Business_Machines
{
    void func();
    // ...
}

namespace IBM = International_Business_Machines; // 這是一個 using alias (別名)
using International_Business_Machines::func;    // 這是一個 using declaration
using namespace International_Business_Machines; // 這是一個 using directive
```

第 9 章 多載化函式 (overloaded functions)

```
// 相同名稱、不同參數列的各個函式，稱為多載化函式 (overloaded functions)
// 以下是四個多載化函式
void f();
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );
```

```

void main() {
    f( 5.6 ); // 此一呼叫須透過函式多載決議程序 (function overload resolution) 才能決定喚起誰
}

// 所謂「函式多載決議程序」，是將一個函式呼叫動作，關聯到多載化函式集
// 中的某個確切函式身上。這個程序的主要過程就是挑選多載集合中的
// 各個同名函式，找出其中哪一個最能夠吻合函式呼叫時所指定之引數型別。
//
// 函式多載決議程序有三個步驟，分別挑選以下各組函式：
// 1. 候選函式 (candidate functions)：與被呼叫函式同名，且於呼叫點可見。
// 2. 可行函式 (viable functions)：根據引數挑選出。
// 3. 最佳可行函式 (best viable function)：可行函式中等級 (rank) 最高者。

// 在第二步驟中，編譯器鑑定出可施行於函式呼叫所夾帶之每一個引數身上的型別轉換行為，
// 並予分等。此一分等導至三個可能的結果：
//
// 1. 完全吻合。不過引數並不一定得精確吻合參數型別；某些輔助轉換動作可以施行於引數身上：
//     a. lvalue-to-rvalue 轉換
//     b. array-to-pointer 轉換
//     c. function-to-pointer 轉換
//     d. qualification (資格修飾詞) 轉換
// 2. 因型別轉換而吻合。可能的轉換動作分為三類：晉升轉換、標準轉換、使用者自定轉換。
//     使用者自定的轉換為行係由兩種函式來執行：(1) 轉換函式 (2) member function
// 3. 不吻合。

```

第 10 章 Function Templates (函式範本)

```

// 所謂 template (範本) 是一種帶參數的一般化預先描述體，描述對象包括 function 和 class。
// 描述 function 者稱為 function template，描述 class 者稱為 class template (第 16 章)。
// 下面是一個 function template 形式。
template <typename T> // <typename T> 稱為 template parameter list
T func( T p1, T p2, int p3 ) // 其中 T 是 template type parameter (型別參數)
{ /* ... */ } // 另有所謂 template nontype parameter (非型別參數)，
// 本例未出現。

// 給予 template 適當的引數，編譯器會自動產生出對應的 template 實體 (instance)。
// 這個過程稱為 instantiated (具現化)，產生出來的實體稱為 template instantiation (具現體)
// 以下造成上述 function template 產生出函式實體 double func( double, double, int );
func( 6.2, 1.5, 6 );

unsigned int ui;
func( ui, 1, 6 ); // 這動作會因引數推導 (argument deduction) 失敗而造成編譯錯誤

// 以下動作明白指定 template 引數 (所謂 explicit template argument)，可強迫引數推導成功
func<unsigned int>(ui, 1, 6);

```

```
// 欲針對某引數型別，訂製特別的函式定義，可使用所謂的 template explicit specialization
template<> double func<double>(double p1, double p2, int p3)
{ /* ... */ }

// 為強迫編譯器在某個定點產生具現體，可採用 explicit instantiation declaration（明白的具現宣告）
template double func<double>(double p1, double p2, int p3);
```

第 11 章 異常處理（exception handling）

```
// 所謂 exceptions 是一種在程式執行期間可偵測到的不正常情況。我們以及我們所使用的 library
// 通常將它設計為 classes。例如：
class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };

// 下面是 exception 的丟出（throw）和捕捉（catch）
int fool()
{
    try { // 大括弧內稱為一個 try block。裡面可能發生 exceptions：或直接以
        // throw 算式丟出，或由這裡所呼叫的函式（或再呼叫下去的函式）丟出。
        throw popOnEmpty(); // 丟出一個 exception object
    }
    catch ( pushOnFull ) { // 小括號內，稱為一個 exception declaration
        // ... 「catch 子句」又稱為 exception handler（異常處理常式）
    }
    catch ( popOnEmpty eobj ) { // 收到一個 exception object
        // ...
    }

    return 0;
}

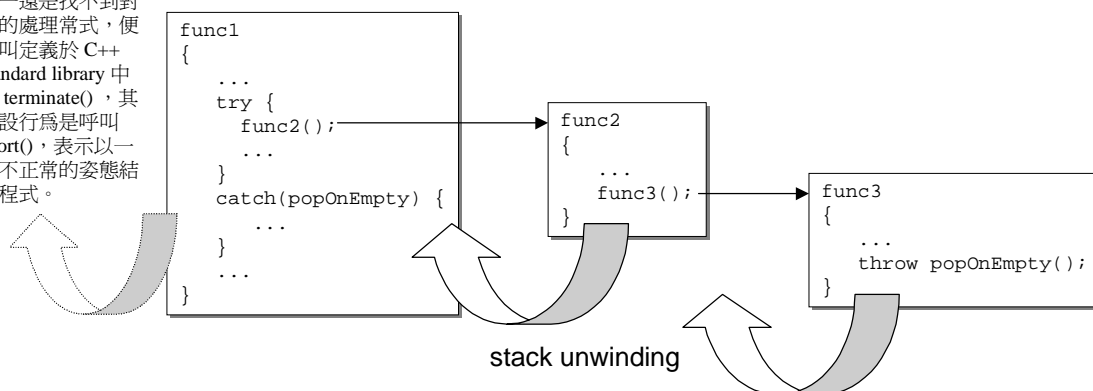
// 也可以宣告一個函式，讓整個函式本體內含於一個 try block 內。
// 也就是把函式本體封包於一個所謂的 function try block 內。
// 此為最新之 C++ 特性，支援之編譯器相對較少。
int foo2()
try {
    // 這裡放置 foo2() 的函式本體
}
catch ( pushOnFull ) {
    // ...
}
catch ( popOnEmpty ) {
    // ...
}

// 我們可以利用所謂的 exception specification 列出某個函式可能丟出的 exceptions 種類
int foo3( int i ) throw(double, int); // 保證不發出 double 和 int 型別以外的 exceptions
```



```
// stack unwinding (堆疊輾轉開解) :
// 「尋找適當的 catch 子句以處理某個被丟出的 exception」過程中，搜尋過程會持續
// 往上一層呼叫端函式 (calling function) 進行。這樣的過程持續沿著「巢狀的 (nested)
// 函式呼叫串鏈 (chain)」上溯，直至找到一個與此 exception 相應的 catch 子句為止。
// 當複合述句和函式定義因為一個 exception 而退離 (exit)，並因此開始搜尋該
// exception 所關聯的 catch 子句，整個過程我們稱為 stack unwinding (堆疊輾轉開解)。
// 一旦如此，那些退離的複合述句和函式定義所宣告的 local object 的生命就會結束。
// C++ 保證，一旦發生 stack unwinding，雖然上述那些 local class objects 的生命結束
// 是由於某個被丟出的 exception 而造成，但它們的 class destructors 會被喚起。
// 請參考 11.3.2 節和 19.2.5 節。
```

萬一還是找不到對應的處理常式，便呼叫定義於 C++ standard library 中的 `terminate()`，其預設行為是呼叫 `abort()`，表示以一種不正常的姿態結束程式。



第 12 章 泛型演算法 (Generic Algorithms)

```
// 所謂泛型演算法，實作出共同的、常用的動作，如 min()，max()，find()，和 sort()。
// 稱其為泛型則是因為它們的動作跨越多種 container (容器) 型別。Container 藉著一對
// iterators (迭代器) 與泛型演算法產生關聯，這一對 iterators 標示出來回巡訪的元素範圍。
// container (容器)：一種可容納眾多同型物件的資料結構。C++ Standard Library 定義有
// list, vector, map, set, deque...等各色容器，它們都是 class template。
// iterator：一種泛型指標。
```

```
#include <algorithm> // 運用 generic algorithms 前需先含入此檔
#include <vector>     // 運用 vector container 前需先含入此檔
#include <iostream>
using namespace std; // 這是 using directive。std 是 C++ Standard Library 的 namespace 名稱

int main()
{
    int search_value;
    int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
    vector<int> vec( ia, ia+6 ); // 使用一個 vector container，名為 vec

    cout << "enter search value: ";
    cin >> search_value;
```

```

vector<int>::iterator presult; // 定義一個使用於 vector 的 iterator
// 以下 vec.begin() 和 vec.end() 分別傳回指向頭尾的兩個 iterators，用以界定巡曳範圍
presult = find( vec.begin(), vec.end(), search_value );

cout << "The value " << search_value
      << ( presult == vec.end()
          ? " is not present" : " is present" )
      << endl;

return 0;
}

// function object：這是一種 class，其中將 call 運算子多載化。call 運算子用來
// 將「正常情況下應該被實作為函式」的特徵封裝起來。function object 通常會被當做
// 泛型演算法的引數，其作用有點類似函式指標。以下是個實例：
class LessThan {
public:
    bool operator()( const string & s1, const string & s2 )
    { return s1.size() < s2.size(); }
};

// function adaptors：由 Standard library 提供，可用來將一元和二元的 function objects
// 加以特殊化或擴充。主要區分為兩大類型：(1) Binder (2) Negator，請參考 12.3.5 節。例如：
count_if( vec.begin(), vec.end(),
          bind2nd( less_equal<int>(), 10 ) );

// 其中 count_if 是個泛型演算法；vec.begin() 和 vec.end() 分別傳回指向頭尾的 iterators；
// less_equal 是預先定義的 function object；bind2nd 是個 Binder function adaptor，用來
// 修改 less_equal 的意義。又例如：
count_if( vec.begin(), vec.end(),
          not1( bind2nd( less_equal<int>(), 10 ) ) );

// 其中 not1 是個 Negator function adaptor，用來修改其後的 function object 的意義。

```

第 13 章、14 章 Class 相關主題

```

// 以下是 class Demo 的宣告式 (declaration)，又稱為定義式 (definition)。
class Demo // 這是 class header
{ // 左右大括號所夾範圍稱為 class body
    friend foo(); // friend function (此種函式有特殊存取權限)
public: // public access level (「開放」存取層級)
    // class 內宣告的函式是所謂的 class member functions (成員函式)
    // 可加以各種修飾詞：friend, inline, const, volatile, mutable, static
    void func1();
    void func2( int, int );
    Demo( int a1, int a2); // 與 class 同名，稱為 constructor (建構式)
    Demo() { }; // 無參數之 constructor，稱為 default constructor
    Demo& operator=(const Demo &rhs); // 這種型式稱為 copy assignment 運算子

```

```

~Demo() { } ; // 與 class 同名且加 ~ 符號，稱為 destructor (解構式)

private: // private access level (「私有」存取層級)
    // class 之中定義的資料是所謂的 class data members (資料成員)
    int i1, i2;

protected: // protected access level (「受保護」存取層級)
    int i3;
};

inline Demo:: // 函式名稱修飾，表示這是個 Demo member function，且為 inline 性質。
Demo(int a1, int a2)
    : i1(a1), i2(a2) // 這是所謂的 member initialization list (成員初值列)
{ i3 = i1 + i2; }

Demo& Demo::
operator=(const Demo &rhs) // 這是 copy assignment 運算子
{
    if (this != &rhs) {
        i1 = rhs.i1; // 原本編譯器會自動提供一個 default copy assignment 運算子，
        i2 = rhs.i2; // 其內實現 memberwise copy (成員逐一拷貝) 語意。
        i3 = 10; // 為改變此一預設行為，可自行撰寫 copy assignment 運算子，如本例。
    }
    return *this;
}

Demo obj1(3, 5); // 這便產生出一個 Demo class object
Demo obj2 = obj1; // 這便產生出一個 Demo class object 並以 obj1 為初值

// 以下 Node 是一個 nested class (巢狀類別)
class Tree {
public:
    class Node { /* ... */ };
};

// 以下 Bar 是一個 local class (區域類別)
void foo( int val )
{
    class Bar { /* ... */ };
}

```

第 15 章 多載化運算子 (overloaded operator) 與使用者自定轉換 (user-defined conversion)

// 運算子其實也是一種函式。函式可以被多載化，所以運算子也可以被多載化。程式員為求
 // 對設計出來的 classes 提供最直接最習慣的使用形式，俾與基本資料型別 (如 int) 的
 // 操作習慣相同，往往會在 class 內對運算子做多載化處理。其主要步驟與形貌皆與
 // 函式的多載化相同。運算子的函式名稱是 "operator" 加上運算子符號。例：

```

class Demo
{
public:
    Demo operator+( const Demo& rhs)    // 多載化運算子 (overloaded operator)
    {
        Demo result; // 許多編譯器對此做了 name return value 最佳化處理以提昇效率 (14.8 節)
        result.i1 = this->i1 + rhs.i1;
        result.i2 = this->i2 + rhs.i2;
        result.i3 = this->i3 + rhs.i3;
        return result;
    }
    // ...
};

// 於是便可以這麼做：
Demo obj1, obj2;
Demo obj3 = obj1 + obj2; // 喚起上述的 operator+ 運算子

// 所謂轉換函式 (conversion function) 是一種特殊的 class member function。它定義出一種使用者
// 自定的轉換行為，將一個 class object 轉換為某種型別。轉換函式被宣告於 class body 內，以
// 關鍵字 operator 加上一個型別名稱 (此型別即其轉換目標)。下例完成後，凡需要 int 型別的
// 地點，如給予 Token object，編譯器便自動喚起轉換函式，將 Token object 轉為 int object

class SmallInt;
class Token {
public:
    Token( char*, int );
    operator SmallInt() { return val; } // 這是一個轉換函式 (conversion function)
    operator int()      { return val; } // 這是一個轉換函式 (conversion function)
    // other public members
private:
    SmallInt val;
    char *name;
};

```

第 16 章 Class Templates (類別範本)

// 本章使用的術語，在第 13 章 Class 和第 10 章 Template 中都已出現過。
 // 以下是一個 class template 的基本形式示範：

```

template <class T> class QueueItem; // 這只是一個前置宣告 (forward declaration)
template <typename Type>           // Type 是所謂的 template parameter (範本參數)，
class Queue {                      // 此例中 Type 是一個型別參數 (type parameter)，
public:                             // 另有所謂的非型別參數 (nontype parameter)
    Queue() : front( 0 ), back ( 0 ) { } // constructor，含 member initialization list
    ~Queue() { }                       // 這是 destructor

    Type max();
    Type& remove();

```

```

        void add( const Type & );
        bool is_empty() const {
            return front == 0;
        }
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};

// 以下產生一個 template instantiation (範本具現體)
Queue<int> qi;    // 本例的 template argument (範本引數) 為 int

// 於是編譯器相當於為我們做出一個 class Queue<int> 如下。注意，
// 每一個 Type (本例之 template 型別參數) 出現點都被 int (本例之 template 引數) 取代。
class Queue<int> {
public:
    Queue<int>() : front( 0 ), back ( 0 ) { }
    ~Queue<int>() { }

    int max();
    int& remove();
    void add( const int & );
    bool is_empty() const {
        return front == 0;
    }
private:
    QueueItem<int> *front;
    QueueItem<int> *back;
};

// 當編譯器測知程式需要一個 template 的完全定義時，才會將該 template 具現化，產生出一個
// 具現體 (instantiation)。template 應用程式對檔案的編組方式，視編譯器支援哪一種
// 編譯模型而定：(1) Inclusion (含入式) 編譯模型 (2) Separation (分離式) 編譯模型

// 若要確實掌握 class template 的具現點，可使用「明白指定的具現宣告」
// (explicit instantiation declarations)：
template class Queue<int>;

// 如果 class template 的 member function 有必要針對某個型別做出特殊的定義，
// 可使用所謂的 class template specialization (特製體)。例如假設上述 Queue
// 的函式 max() 有必要為 LongDouble 型別做特殊定義，可以這麼做 (詳見 16.9 節)：
template<> LongDouble Queue<LongDouble>::max( )
{
    // 針對 LongDouble 型別做特殊定義，
    // 與「泛型定義」所產生出來的具現體不同。
}

// 如果 class template 擁有一個以上的 template 參數，而我們打算為針對某個 (或某一組)
// 特定的 template 引數做特殊化，而不是針對所有的 template 參數做特殊化，可使用
// 所謂的 class template partial specialization (局部特製體)，詳見 16.10 節

```

第 17 章 Class 的繼承與子型別的建立

```
// 本章引入繼承機制 (inheritance)，以及多型性質 (polymorphism)。本書至此之前
// 的 classes 都是獨立出現，class 和 class 之間沒有上下屬的關連稱為所謂的
// object-based programming。導入繼承與多型之後，才稱得上是 object-oriented programming。
// 擁有虛擬函式的 class，稱為 polymorphic class (具備多型性質的類別)。
// 下面是 base class (基底類別)、derived class (衍生類別)、虛擬函式 (virtual functions) 的形式：

class Query { // 擁有純虛擬函式之 class，稱為抽象類別 (abstract class)，
public:      // 否則稱為具象類別 (concrete class)
    virtual eval() = 0; // 這是純虛擬函式 (pure virtual function)
    virtual display() const; // 這是虛擬函式 (virtual function)
    // ...
protected: // 宣告為 protected 之 data member，可被 derived class 直接取用。
    // ...
};

class NameQuery : public Query // NameQuery 是個 derived class，Query 是個 base class
{
public:
    void eval(); // 重新定義虛擬函式，此動作稱為改寫 (override)
    // ...
};

// 於是可以產生以下的物件：
NameQuery q1;

// derived class object 體中可分為兩部份，一是自 base class 繼承而來的 base subobject，
// 一是衍生的嶄新部份 derived part。這個主題其實屬於 object model 領域，
// Inside the C++ Object Model 《中譯 深度探索 C++ 物件模型》或《多型與虛擬》講解更為細微精密。
// 由 base class 與 derived classes 架構起來的階層體系，稱為 classes hierarchy。
// 如果 class 階層體系中的每一個 derived class 只有一個 base class，這個階層體系
// 稱為單一繼承 (single inheritance) 體系
```

第 17 章利用 *object-oriented programming* 方式，接續開發第 6 章的文字檢索系統。該系統所需的幾個重要資料結構，曾在先前第 6 章導讀中展現，可有效幫助你快速掌握此一大型程式架構。

相對於第 6 章的單字檢索，本章之檢索系統提供一種「檢索語言」，可容許比較複雜的檢索句型。此一檢索語言提供 **And (&&)**、**Or (||)**、**Not (!)**、小括號 **()** 等四種運算子，可讓使用者組合出複式檢索句型。例如：

```
bird || ( daddy && hair ) || ( ! she )
```

為支援這種能力，本章以一個 `Query class` 階層體系 (p.888) 完成任務，其中每一個 `class` 實作出一種運算子語意。分析複式運算句型中的一元運算子、二元運算子、小括號時，如果讀者曾修習「資料結構」這門課，比較能夠輕鬆以對。

注意，如果某個單字重複出現於同一行多次，本系統希望只顯示該行一次就好。如此一來第 6 章完成的 *word location map* 中的 *location vector* 不敷運用。我們有必要從 *location vector* 身上計算出一組不重複的行號集合，作法之一就是以 *location vector* 內的所有行號組成一個 *solution set*，自動以遞增次序存放一組絕無重複的值。例如，下面這些重複出現的單字（請參考前述第 6 章導讀）：

```
bird      ((2,3),(2,9))
daddy     ((0,8),(3,3),(5,5))
fiery     ((2,2),(2,8))
hair      ((0,6),(1,6))
her        ((0,7),(1,5),(2,12),(4,11))
him        ((4,2),(4,8))
she        ((4,0),(5,1))
tell      ((2,11),(4,1),(4,10))
```

其 *location vector* 所對應的 *solution set*（由不重複的行號組成）是：

```
bird      (2)
daddy     (0,3,5)
fiery     (2)
hair      (0,1)
her        (0,1,2,4)
him        (4)
she        (4,5)
tell      (2,4)
```

第 18 章 多重繼承與虛擬繼承

```
// 所謂多重繼承（multiple inheritance），形式如下。其中的繼承型式可為
// public 或 protected 或 private，各有用途（18.3 節）
class Panda
: public Bear, public Endangered // 這一行稱為 class derivation list
{ ... };

// 所謂組合（composition），是指 class member 的型別亦為 class。繼承表現出
// 是一種（is-a）的關係，組合則表現有一個（has-a）的關係。例如：
class Wheel { /* ... */ };
class Car
{
    //...
    Wheel w[4]; // 汽車有四個輪子
};

// 組合有兩種形式：composition by value 和 composition by reference（18.3.4 節）

// C++ 一般的繼承，是 composition by value 的特殊型式。所謂虛擬繼承（virtual inheritance），
// 則是一種 composition by reference 機制。其形式如下：
class ZooAnimal { /* ... */ };
class Bear : public virtual ZooAnimal { /* ... */ };
```



```
class Raccoon : virtual public ZooAnimal { /* ... */ };
class Panda : public Bear, public Raccoon
{ /* ... */ };    // 於是 ZooAnimal 成為 Panda 的 virtual base class
```

第 19 章 在 C++ 中使用繼承機制

// C++ 程式運用繼承機制時，常常需要在執行時期得知某個 object 的真正型別，這個性質稱為執行時期型別辨識（Runtime Type Information，RTTI）。C++ 提供兩個運算子：支援 RTTI，一是 `dynamic_cast` 運算子，一是 `typeid` 運算子。它們都必須在 `polymorphic class`（具備多型條件的類別）身上才有作用。以上述 `ZooAnimal` 繼承體系為例：

```
ZooAnimal* animal1 = new Panda;    // 這是一種向上轉型（up cast），是安全的轉型
cout << typeid(*animal1).name() << endl;
// typeid() 獲得一個 typeid 的 class object，name() 是其中的一個 member function。
```

```
ZooAnimal* animal2 = new Bear;      // 這是一種向上轉型（up cast），是安全的轉型
// 以下是一種向下轉型（down cast），是不安全的轉型。藉 dynamic_cast 運算子之助可挑出錯誤。
Panda* jjhou = dynamic_cast<Panda*>(animal2);
assert(jjhou);
```

第 20 章 The iostream library

iostream library 是 C++ Standard Library 的一個組件，幾乎可以說是最為 C++ 族群耳熟能詳的一個程式庫。它不但供應對標準輸入/輸出裝置的支援，也支援對檔案的輸入/輸出，並允許任何 class 將 `output` 運算子（`operator<<`）和 `input` 運算子（`operator>>`）多載化，使 class object 的輸入/輸出動作更為簡明乾淨。將 class object 儲存到檔案，這個性質即是所謂 `object persistence`（物件的永續生存）。

iostream library 是虛擬繼承和多重繼承的成功案例。