C++ 標準程式庫

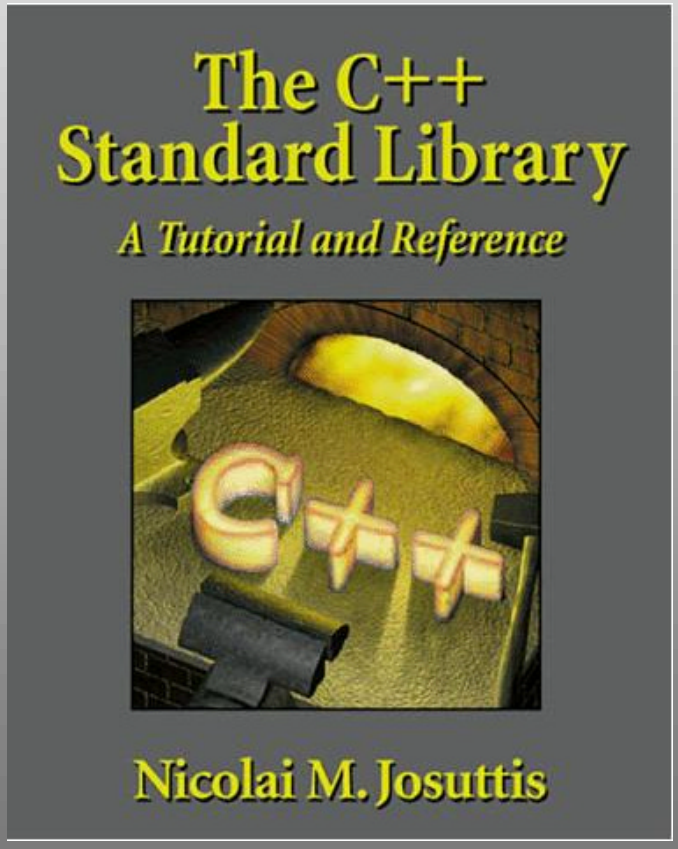
The C++ Standard Library, by Josuttis

侯捷 孟岩 譯

碁峰

C++ 標準程式庫

— 自修教本與參考工具 —



侯捷/孟岩 譯

碁峰腦圖書資料股份有限公司

C++ 標準程式庫

C++ Standard Library

教本與手冊（A Tutorial and Reference）

Nicolai M. Josuttis 著

侯捷 / 孟岩 合譯

巨細靡遺 井然有序

（侯捷譯序）

自從 1998 年 *C++ Standard* 定案以後，C++ 程式庫便有了大幅擴充。原先爲大家所熟知、標準規格定案前蘊釀已久的 STL（Standard Template Library，標準模板程式庫），不再被單獨對待，而是被納入整個 C++ 標準程式庫（Standard Library）。同時，原有的程式庫（如 iostream）也根據泛型技術（generics）在內部上做了很大的修改。可以說，*C++ Standard* 的發佈對 C++ 社群帶來了翻天覆地的大變動 — 不是來自語言本身，而是來自標準程式庫。這個變動，影響 C++ 程式編寫風格至鉅，C++之父 Bjarne Stroustrup 並因此寫了一篇文章：*Learning Standard C++ as a New Language*（載於 C/C++ User's Journal, 1999/05）。

我個人於 1998 年開始潛心研究泛型技術和 STL，本書英文版《***The C++ Standard Library***》甫一出版便成爲我學習 C++ 標準程式庫的最重要案頭工具之一。小有心得之後，我寫過數篇相關技術文章，從來離不開本書的影響和幫助。我曾經把 STL（代表泛型技術目前最被廣泛運用的一個成熟產品，也是 C++ 標準程式庫的絕大成份）的學習比喻爲三個階段（或層次）：

- 第一境界：熟用 STL
- 第二境界：瞭解泛型技術的內涵與 STL 的學理乃至實作
- 第三境界：擴充 STL

不論哪一個階段，你都能夠從本書獲得不同程度的幫助。

第一階段（對最大多數程式員有立竿見影之效），我們需要一本全面而詳盡的教本，附帶多量而設計良好的範例，帶領我們認識十數個 STL 容器（containers）、數十個 STL 演算法（algorithms）、許許多多的迭代器（iteartors）、配接器（adapters）、仿函式（functors）…的各種特性和用途。這些爲數繁多的組件必須經過良好的編排組織和索引，才能成就一本效果良好、富教育性又可供長久查閱的案頭工具書。

在這一階段裡，本書表現極爲優異。書中運用許多圖表，對所有 STL 組件的成員做了極其詳盡的整理。更值得稱道的是書中交叉參考（cross reference）做得非常好，在許多關鍵地點告訴讀者當下可參見哪一章哪一節哪一頁，對於閱讀和學習帶來很大的幫助（本中文版以頁頁對譯方式保留了所有交叉參考和索引）。

第二階段（從 STL 的運用晉升至泛型技術的學習），我們需要一些關鍵的 STL 源碼（或僞碼, pseudo code），幫助我們理解關鍵的資料結構、關鍵的編程技術。認識這些關鍵源碼（或僞碼）同時也有助提昇第一階段的運用深度（學會使用一樣東西，卻不知道它的道理，不高明☺[1]）。

本書很多地方都提供了 C++ 標準程式庫的關鍵源碼。不全面，但很關鍵。

第三階段（成爲一位泛型技術專家；打造自己的 STL 相容組件），我們需要深入了解 STL 的設計理念和組織架構[2]，並深入（且全面地）了解 STL 實作手法[3]。是的，不入虎穴，不能得虎子；徹底了解 STL 如何被打造出來之後，你才能寫出和 STL 水乳交融、完美整合的自定組件（user-defined components）。

本書對第三階段的學習也有相當幫助。雖然沒能提供全面的 STL 源碼並分析其技術（那需要另外 800 頁☺），卻提供了爲數不少的訂製型組件實作範例：p191, p213 提供了一個執行期指定排序準則並運用不同排序準則的實例，p219 提供一個自定容器（雖然只是個簡單的包覆類別），p222 提供一個「*reference* 語意」示範作法，p285 提供一個針對迭代器而設計的泛型演算法，p288 提供一個用於關聯式容器的訂製型 *inserter*，p294 有一個自定的排序準則，p441 有一個自定的（安全的）stack，p450 有一個自定的（安全的）queue，p504 有一個自定的 traits class for string， p614 有一個自定的 stream 操控器，p663 有一個自定的 stream 緩衝區，p735 有一個自定的記憶體配置器（allocator）。

---

[1] 乍見之下令人錯愕的一句話。看電視需要先了解電視的原理嗎？呵呵，話講白就沒意思了。這句話當然是對技術人員說的。

[2] 這方面我推薦你看《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》, by Matthew H. Austern, Addison Wesley 1998。詳見稍後說明。中譯本《泛型程式設計與 STL》,侯捷/黃俊堯合譯, 碁峰, 2001。

[3] 這方面我推薦你看《STL 源碼剖析, *The Annotated STL Sources*》by 侯捷, 碁峰, 2002。詳見稍後說明。

除了眾所矚目的 STL，本書也涵蓋一般不被歸類爲 STL 的 String 程式庫，以及一般不被視爲關鍵的 IOStream 和 Locale 程式庫[4]。三部分互有關連，以 IOStream 爲主幹。在 GUI（圖形使用介面）和 application framework（應用程式框架）當道的今天，IOStream 提供的輸出輸入可能對大部份人失去了價值，但如果你希望開拓 OO 技術視野，IOStream 是一顆沉睡的珠寶。

❖ ❖ ❖

泛型技術不僅在 C++ 被發揚光大，在 Java 上也有發展[5]，在 C# 上亦被眾人期待。從目前的勢頭看，泛型技術（Generics）或許是物件導向（Object Oriented）技術以來程式編寫方面的又一個巨大衝擊。新一代 C++ 標準程式庫[6]將採用更多更複雜更具威力的泛型技術，提供給 C++ 社群更多更好更具復用價值的組件。

不論你要不要、想不想、有沒有興趣在你的程式編寫過程中直接用上泛型技術，至少，在 C++ 程式編寫過程中你已經不可或缺於泛型技術帶來的成熟產品：C++ 標準程式庫。只要你具備 C++ 語言基礎，本書便可以帶領你漂亮地運用 C++ 標準程式庫，漂亮地提昇你的編程效率和程式品質。

面對陌生，程式員最大的障礙在於心中的怯弱。To be or not to be, that is the question! 不要像哈姆雷特一樣猶豫不決。面對光明的技術，必須果敢。

❖ ❖ ❖

關於術語的處理，本書大致原則如下：

1. STL 各種資料結構名稱皆不譯，例如 array, vector, list, deque, hast table, map, set, stack, queue, tree…。雖然其中某些已有約定俗成的中文術語，但另一些沒有既標準又被普遍運用的中文名稱，強譯之讀者瞠目以對，部分譯部分不譯則閱讀時詞性平衡感不佳（例如「面對向量和 **deque** 兩種容器…」就不如「面對 **vector** 和 **deque** 兩種容器…」讀起來順暢）。因此，資料結構名稱全部不譯。直接呈現這些簡短的英文術語，可能營造更突出的視覺效果，反而有利

[4] 這方面我見過的唯一專著是《*Standard C++ IOStreams and Locales - Advanced Programmer's and Reference*》, by Angelika Langer and Klaus Kreft, Addison Wesley 2000。

[5] (1) *GJ : A Generic Java*, by Philip Wadler, Dr. Dobb's Journal February 2000.
(2) *JSR- 000014 : Adding Generics to the Java Programming Language*, http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

[6] 請參考 http://www.boost.org/，這個程式庫據稱將成爲下一代 C++ 標準。

閱讀。技術書籍的翻譯不是爲了建立全中文化閱讀環境，我們的讀者水平也不可能受制於這些英文單字。

2. STL 六大組件的英文名稱原打算全部保留，但由於處處出現，對版面的中英文比例形成視覺威脅，因此全部採用以下譯名：**container 容器**，**algorithm 演算法**，**iterator 迭代器**，**adapter 配接器**，**functor 仿函式**[7]，**allocator 配置器**。
3. 任何一個被保留的英文關鍵術語，其第一次（或前數次）出現時儘可能帶上中文名稱。同樣地，任何關鍵的中文術語，我也會時而讓它中英並陳。

❖ ❖ ❖

關於編排，本書原則如下：

1. 全書按英文版頁次編排，並因而得以保留原書索引。索引詞條皆不譯。
2. 中文版採用之程式碼字體（Courier New 8.5）比文本字體（細明體 9.5）小，英文版之程式碼字體卻比其文本字體大，且行距寬。因此中文版遇有大篇幅程式列表，爲保持和英文版頁次相應，便會出現較多留白。根據我個人對書籍的經驗，去除這些留白的最後結果亦不能爲全書節省五頁十頁；填滿每一處空白卻喪失許多立即可享的好處，智者不取☺。

❖ ❖ ❖

一旦你從本書獲得了對 C++ **標準程式庫**運用層面的全盤掌握與實踐經驗之後，可能希望對 STL 原理乃至實作技術做更深的研究，或甚至對泛型編程（Generic Programming）產生無比狂熱。在眾多相關書籍之中，下面是我認爲非常值得繼續進修的四本書：

1. 《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》, by Matthew H. Austern, Addison Wesley 1998。本書第一篇（前五章）談論 STL 的設計哲學、程式庫背後的嚴密架構和嚴謹定義。其中對於 STL 之異於一般程式庫，有許多重要立論。其餘部分（第二篇、第三篇）是 STL 的完整規格（分別從 concepts 的角度和 components 的角度來闡述），並附範例程式。

[7] 原書大部份時候使用 **function object 函式物件**一詞，爲求精簡及突出，中文版全部改用其另一個名稱 **functor 仿函式**（見第 8 章譯註）。

2. 《STL 源碼剖析, *The Annotated STL Sources*》by 侯捷, 碁峰, 2002。本書剖析 STL 實作技法，詳實揭示並註解 STL 六大組件的底層實作，並以公認最嚴謹的 SGI（Silicon Graphics Inc.）STL 版本爲剖析對象。附許多精彩分析圖，對於高度精巧的記憶體配置策略、各種資料結構、各種演算法、乃至極爲「不可思議」的配接器（adapter）實作手法，都有深入的剖析。
3. 《*Effective STL*》, by Scott Meyers, Addison Wesley 2001。本書定位爲 STL 的深層運用。在深層運用的過程中，你會遇到一些難解的問題和效率的考量，你需要知道什麼該做、什麼該避免。本書提供 50 個專家條款。請注意，深層運用和效率調校，可能需要讀者先對底部機制有相當程度的了解。
4. 《*Modren C++ Design*》by Andrei Alexandrescu, Addison Wesley 2001。將泛型技術發揮到淋漓盡致、令人目瞪口獃的一本書籍。企圖將泛型技術和設計樣式（design patterns）結合在一起。領先時代開創先河的一本書。

❖ ❖ ❖

本書由我和孟岩先生共同完成。孟岩在大陸技術論壇以 C++/OO/Generics 馳名，見解深雋文筆不凡。我很高興和他共同完成這部作品。所謂合譯，我們兩人對全書都有完整的參與（而非你一半我一半的對拆法），最終由我定稿。本書同時發行繁體版和簡體版，基於兩岸計算機術語的岐異性，簡體版由孟岩負責必要轉換。

侯捷 2002/05/23 于新竹

http://www.jjhou.com（繁體網站）
http://jjhou.csdn.net（簡體網站）
jjhou@jjhou.com（個人電子郵箱）

孟岩譯序

IT 技術書籍市場，歷來是春秋戰國。一般來說，同一個技術領域裏總會有那麼數本、十數本、甚至數十本定位相似的書籍相互激烈競爭。其中會有一些大師之作脫穎而出，面南背北，黃袍加身。通常還會有後來者不斷挑戰，企圖以獨到特色贏得自己的一片天地。比如說在演算法與資料結構領域，D. E. Knuth 的那套《*The Art of Computer Programming*》一至三卷，當然是日出東方惟我獨尊。但是他老人家的學生 Robert Sedgewick 憑著一套更貼近實用的《*Algorithms in C*》系列，也打出自己一片天下，成爲很多推薦列表上的首選。就 C++ 運用經驗類書籍來說，Scott Meyers 的《*Effective C++*》稱王稱霸已經多年，不過其好友 Herb Sutter 也能用一本《*Exceptional C++*》獲得幾乎並駕齊驅的地位。嗨，這不是很正常的事嗎？技術類書籍畢竟不是詩詞歌賦。蘇軾一首「明月幾時有，把酒問青天」，可以達到「詠中秋者，自東坡西江月後，餘詞盡廢」的程度，但怎麼可能想像一本技術著作達到「我欲乘風歸去，又恐瓊樓禦宇，高處不勝寒」的境界！誰能夠寫出一本技術書，讓同一領域後來者望而卻步乾脆死心，那才是大大的奇跡！

然而，您手上這本《*The C++ Standard Library*》，作爲 C++ 標準程式庫教學和參考類書籍的定音之作，已經將這個奇跡維持了三年之久。按照 IT 出版界時鐘，三年的時間幾乎就是半個世紀，足以錘煉又一傳世經典！

1998 年 *C++ Standard* 通過之後，整個 C++ 社群面臨的最緊迫任務，就是學習和理解這份標準給我們帶來的新觀念和新技術。而其中對於 C++ 標準程式庫的學習需求，最爲迫切。C++ 第二號人物 Andrew Koenig 曾經就 C++ 的特點指出：『語言設計就是程式庫設計，程式庫設計就是語言設計』[1]。*C++ Standard* 對程式庫所做的巨大擴充和明確規範，實際上即相當於對 C++ 語言的能力做了全面提升與擴展，意味著你可以站在無數超一流專家的肩上，於輕敲鍵盤間，將最出色的思想、設計與技術納入囊中，讓經過千錘百煉的精美代碼成爲自己軟體大廈的堅實基礎。

[1] "Language design is library design, library design is language design"，參見 Andrew Koenig, Barbara Moo 合著《Ruminations on C++》第 25, 26 章。

可以說，對於大多數程式師來說，標準 C++ 較諸「ARM 時代」之最大進步，不是語言本身，而恰恰是標準程式庫。因此，我們可以想像當時人們對 C++ 標準程式庫教學類書籍的企盼，是何等熱切！

一方面是已經標準化了的成熟技術，另一方面是萬衆期待的眼神，我們完全有理由認爲，歷史上理應爆發一場魚龍混雜的圖書大戰。它的典型過程應該是這樣：先是一批快刀手以迅雷不及掩耳盜鈴之勢[2]推出一堆斂財廢紙，然後在漫長的唾罵與期待中，大師之作漸漸脫穎而出。大浪淘沙，最後產生數本經典被人傳頌。後雖偶有新作面世，波光點點已是波瀾不興。

然而，這一幕終究沒有在「C++ 標準程式庫教學與參考書籍」領域內出現。時至今日，中外技術書籍市場上這一領域內的書籍爲數寥寥，與堆積如山的 C++ 語言教學類書籍形成鮮明對比。究其原因，大概有二，一是這個領域裏的東西畢竟份量太重，快刀手雖然善斬亂麻，對於 C++ 標準程式庫這樣嚴整而精致的目標，一時也難以下手。更重要的原因則恐怕是 1999 年 8 月《*The C++ Standard Library*》問世，直如橫刀立馬，震懾天下。自推出之日起至今，本書在所有關於 C++ 標準程式庫的評論與推薦列表上，始終高居榜首，在 Amazon 的銷量排行榜上名列所有 C++ 相關書籍之最前列。作者僅憑一書而爲天下知，成爲號召力可與 Stan Lippman, Hurb Sutter 等「經典」C++ 作家比肩的人物。此書之後，雖然仍有不少著作，或深入探討標準程式庫的某些組件，或極力擴展標準庫倡導的思想與技術，但是與《*The C++ Standard Library*》持同一路線的書籍，再沒有出現過。所謂泰山北斗已現，後來者已然無心戀戰。

於是有了這樣的評論：『如果你只需要一本講述 C++ 標準程式庫和 STL 的書籍，我推薦 Nicolai Josuttis 的《*The C++ Standard Library*》。它是你能得到的唯一一本全面講述 C++ 標準程式庫的書，也是你能想像的最好的一本書。』這種奇異情形在當今技術書壇，雖然不是絕無僅有，也是極爲罕見。

究竟這本書好到什麼程度，可以獲得這麼高的評價？

我正是帶著這分疑問，接受侯捷先生的邀請，著手翻譯這本經典之作。隨著翻譯過程的推進，我也逐漸解開了心中的疑惑。在我看來，這本書的特點有四：內容詳實，組織嚴密，態度誠懇，深入淺出。

首先，作爲一本程式庫參考手冊，內容詳實全面是一項基本要求。但是，本書在

[2] 此處非筆誤，而是大陸流行的一句“新俚語”，意思十分明顯，就是“迅雷不及掩耳”地“掩耳盜鈴”。

這方面所達到的高度可以說樹立了一個典範。本書作者一開始就提出一個極高的目標，要幫助讀者解決「使用 C++ 標準程式庫過程中所遇到的所有問題」。衆所周知，C++ 標準程式庫是大傢伙，每一部分又有很精深的思想和技術，既不能有所遺漏，又不能漫無邊際地深入下去，何取何舍，何去何從，難度之大可想而知！作者在大局上涵蓋了 C++ 標準程式庫的全部內容，在此基礎上又對所有組件都進行細緻的、立體式的講解。所謂立體式講解，就是對於一個具體組件，作者首先從概念上講解其道理，然後通過漂亮的範例說明其用法，申明其要點，最後再以圖表或詳解方式給出參考描述。有如錢塘江潮，層層疊疊，反反復復，不厭其煩。讀完此書，我想您會和我一樣感受衝擊，並且完全認可作者付出的巨大心血。

C++ 標準程式庫本身就是一個巨大的有機整體，加上這本書的立體講解方式，前後組織和對應的工作如果不做好，很容易會使整部書顯得散亂。令人欽佩的是，這本書在組織方面極其嚴密，幾無漏洞。相關內容的照應、交叉索引、前後對應，無一不處理得妥善曼妙。整體上看，整本書就像一張大網，各部分內容之間組織嚴謹，契合密切，卻又頭緒清晰，脈絡分明，著實難能可貴。我在閱讀和翻譯過程中，常常詫異於其內容組織的精密程度，簡直像德國精密機械一樣分毫不差 — 後來才想到，本書作者 Nicolai Josuttis 就是德國人，精密是德意志民族的性格烙印，名不虛傳！

說起德意志民族，他們的另一個典型性格就是誠實坦率。這一點在這本書同樣有精彩的展現。身爲 C++ 標準程式庫次委員會成員，作者對於 C++ 標準程式庫的理解至深，不但清楚知道其優點何在，更對其缺陷、不足、不完備和不一致的地方瞭如指掌。可貴的是，在這些地方，作者全不避諱，開誠佈公，直言不諱，事實是什麼樣就是什麼樣，絕不文過飾非，絕不含混過關。作爲讀者，我們不僅得以學到好東西，而且學到如何繞開陷阱和障礙。一個最典型的例子就是對於 valarray 的介紹，作者先是清清楚楚地告訴讀者，由於負責該組件設計的人中途退場，這個組件沒有經過細緻的設計，最好不要使用。然後作者一如既往，詳細介紹 valarray 的使用，完全沒有因爲前面的話而稍微有所懈怠。並且在必要的地方將 valarray 的設計缺陷原原本本地指出來，讓讀者口服心服。讀到這些地方，將心比心，我不禁感歎作者的坦誠與無私，專精與嚴謹。

本書最具特色之處，就是其內容選取上獨具匠心，可謂深入淺出。本書的目的除了作爲手冊使用，更是一本供學習者閱讀學習的 "tutorial"（自學教本）。也就是說，除了當手冊查閱，你也可以捧著它一篇一篇地閱讀學習，獲得系統化的堅實知識。一本書兼作 "tutorial" 和 "reference"，就好像一本字典兼作「作文指南」，沒有極高的組織能力和精當的內容選擇，簡直難以想像會搞成什麼樣子。了不起的是本書不僅做到了，而且讓你感覺，學習時它是一本最好的 "tutorial"，查閱時

它是一本最好的 "reference"，我要說，這是個奇跡！單從學習角度來說，本書極爲實用，通過大量鮮明的例子和切中要害的講解讓你迅速入門，而且絕不僅僅淺嘗輒止，而是不失時機地深入進去，把組件的實作技術和擴展方法都展現給讀者。單以 STL 而論，我經常以侯捷先生提出的「STL 學習三境界」來描述一本書的定位層次，這本書就像一座金字塔，紮根於實用，尖鋒直達「明理」和「擴展」層次。從中你可以學到「*reference* 語意」的 STL 容器、smart pointer（靈巧指標）的數種實現、擴充的組合型仿函式（composing function object）、STL 和 IOStream 的擴展方法、訂製型的配置器（allocator）設計思路等等高級技術，也能學到大量的實踐經驗，比如 vector 的使用技巧，STL 容器的選擇，`basic_string<>` 作爲容器的注意事項等等。可以這麼說，這本書足以將你從入門帶到高手層次，可謂深入淺出，精彩至極！

我很高興自己第一次進行技術書籍翻譯，就能夠碰到這樣一本好書，這裏要深深感謝侯捷先生給我一輩子都慶幸的機會。翻譯過程出乎意料地艱辛，前後持續將近 10 個月。我逐字逐句地閱讀原文，消化理解，譯成自以爲合適的中文，然後交給侯先生。侯先生接手後再逐字逐句地閱讀原文，對照我的粗糙譯文，進行修訂和潤色，反復品味形成最終譯稿。作爲譯者，侯先生和我所追求的是，原書技術的忠實呈現加上完全中文化、中國式的表達。我們爲此花費了巨大的心力，對我來說，付出的心血遠遠超過一般翻譯一本書的範疇。雖然最終結果需要廣大讀者評論，但今天面對這厚厚的書稿，我問心無愧地享受這份滿足感。我最大的希望是，每一位讀者在學習和查閱這本中文版的時候，完全忘掉譯者曾經的存在，感覺不到語言的隔閡，自由地獲取知識和技術。對於一個初涉技術翻譯的人來說，這個目標未免太貪心了，可是這始終會是我心裏的願望。一個譯者應該是爲了被忽略而努力的。

最後，感謝侯先生一直以來對我的欣賞和幫助，感謝您給我的機會，我十分榮幸！感謝華中科技大學出版社的周筠老師，您始終友好地關注著我，鼓勵著我。感謝 CSDN 的蔣濤先生，您的熱情鼓勵始終是我的動力。感謝我的父母，弟弟，你們是我最愛的人，是我最堅強的支柱！感謝曾經幫助過我，曾經關心過我的每一個人，無論你現在怎樣，我爲曾經擁有過的，仍然擁有著的每一片快樂和成果，衷心地感謝你！

祝各位讀書快樂！

侯 捷　2002 年 5 月於北京

目 錄

前言
Preface

一開始，我只不過想寫一本篇幅不大的有關於 C++ 標準程式庫的德文書（也就 400 多頁吧）。萌生這個想法是在 1993 年。而在 1999 年的今天，您看到了這個想法的成果：一本英文書，厚達 800 多頁，其中包含大量的文字描述、圖片和範例。我的目標是，詳盡講解 C++ 標準程式庫，使你的所有（或幾乎所有）編程問題都能夠在你遇到之前就先給你解答。然而，請注意，這不是一種完整描述 C++ 標準程式庫的所有面向的書籍，我透過「在 C++ 中利用標準程式庫進行學習和程式編寫」的形式，表現出最重要的主題。

每一個主題都是以一般性概念爲基礎而開展，然後導入日常程式編寫工作所必須了解的具體細節。爲了幫助你理解這些概念和細節，書中提供詳盡的範例程式。

這就是我的前言 — 言簡意賅！撰寫此書的過程中，我得到了很多樂趣，希望你閱讀本書時，能夠像我一樣快樂。請享用！

致謝

Acknowledgments

這本書表達的觀點、概念、解決方案和範例，來源十分廣泛。從這個意義上講，封面只列我一個人的名字，未免不公平。所以我願在此向過去數年來幫助和支持我的人和公司，表示誠摯的謝意。

我第一個要感謝的是 Dietmar Kühl。Dietmar 是一位 C++專家，尤其精通 I/O streams（資料串流）和國際化（他曾經僅僅爲了好玩而寫了一個 I/O stream library）。他不僅將本書的大部分從德文譯爲英文，還親自動筆，發揮專長，爲本書撰寫了數節內容。除此之外，過去的數年裡，他向我提供了很多寶貴的回饋意見。

其次，我要感謝所有檢閱者和那些向我表達過意見的人。他們的努力使本書的品質獲得巨大提昇。由於名單太長，以下如有任何疏漏，還請見諒。英文版的檢閱者包括Chuck Allison, Greg Comeau, James A. Crotinger, Gabriel Dos Reis, Alan Ezust, Nathan Myers, Werner Mossner, Todd Veldhuizen, Chichiang Wan, Judy Ward, Thomas Wike-hult。德文版的檢閱者包括 Ralf Boecker, Dirk Herrmann, Dietmar Kühl, Edda Lörke, Herbert Scheubner, Dominik Strasser, Martin Weitzel。其他投入者包括 Matt Austern, Valentin Bonnard, Greg Colvin, Beman Dawes, Bill Gibbons, Lois Goldthwaite, Andrew Koenig, Steve Rumsby, Bjarne Stroustrup, 和 David Vandevoorde。

我要特別感謝 Dave Abrahams, Janet Cocker, Catherine Ohala 和 Maureen Willard，他們對全書進行了非常細緻的檢閱和編輯。他們的回饋意見讓本書的品質獲得了難以置信的提昇。

我也要特別感謝我的「活字典」Herb Sutter，他是著名的 "Guru of the Week" 的創始人，這是一個常態性的 C++ 難題講解專欄，播出於 comp.std.c++.moderated。

我還要感謝一些公司和個人，他們的幫助使我有機會在各個不同的平台上，使用各種不同的編譯器來測試自己的範例程式。非常感謝來自 EDG 的 Steve Adamczyk，Mike Anderson 和 John Spicer，他們的編譯器真是太棒了，在 C++ 標準化過程和

本書寫作過程中，提供了巨大的支援。感謝 P. J. Plauger 和 Dinkumware, Ltd，他們很早以來就持續進行與 C++ 標準規格相容的 C++ 標準程式庫實作工作。感謝 Andreas Hommel 和 Metrowerks，他們完成了深具價值的 CodeWarrior 程式開發環境。感謝 GNU 和 egcs 編譯器的所有開發者。感謝 Microsoft，他們完成了深具價值的 Visual C++。感謝 Siemens Nixdorf Informations Systems AG 的 Roland Hartinger，他提供了一份他們的 C++ 編譯器測試版本。感謝 Topjects GmbH，爲了他那一份深具價值的 ObjectSpace library 實作品。

感謝 Addison Wesley Longman 公司裡頭與我共同工作過的每一個人。包括 Janet Cocker, Mike Hendrickson, Debbie Lafferty, Marina Lang, Chanda Leary, Catherine Ohala, Marty Rabinowitz, Susanne Spitzer, 和 Maureen Willard 等等。這項工作真是太有趣了。

此外，我還要感謝 BREDEX GmbH 的人們，感謝 C++ 社群中的所有人，特別是那些參與標準化過程的人，感謝他們的支持和耐心（有時候我問的問題確實挺傻）。

最後，也是最重要的，我要將我的感謝（附上一個親吻）送給我的家人：Ulli, Lucas, Anica, 和 Frederic。爲了這本書，我很長時間沒有好好陪他們了。

但願各位能從這本書獲得樂趣，另外，請保持寬厚。
（譯註：上句原文爲 Have fun and be human!。請抱歉我對其精確意思毫無把握）

1

關於本書

1.1 緣起

C++ 問世後不久，就成爲物件導向程式設計領域的實質標準（de facto standard），因此正式標準化的呼聲也就浮上了檯面。一旦我們有了一個可以依循的標準規格，我們才可能寫出跨越 PC 乃至大型主機各種不同平台的程式。此外，如果能夠建立起一個標準程式庫，程式員便得以運用可移植的通用組件（general components）和更高層次的抽象性，而不必從頭創造世界。

C++ 標準化過程始於 1989 年，由國際性的 ANSI/ISO 委員會負責。標準化工作以 Bjarne Stroustrup 的兩本書 *The C++ Programming Language* 和 *The Annotated C++ Reference Manual* 爲根基。這份標準規格於 1997 年通過後，又在數個國家進行了一些正式程序，最後於 1998 年成爲國際性的 ISO/ANSI 標準。標準化過程本身包含一個任務：建立 C++ 標準程式庫。作爲核心語言的拓展，標準程式庫提供了一些通用組件。藉由大量運用 C++ 新的抽象能力和泛型（generic types）特性，標準程式庫提供了一系列共同的類別和介面。程式員藉此獲得了更高層次的抽象能力。這個標準程式庫提供了以下組件：

- String 型別
- 各種資料結構（例如 dynamic array、linked lists、binary trees）
- 各種演算法（例如各種排序演算法）
- 數值類別（numeric classes）
- 輸入/輸出（I/O）類別
- 國際化支援（internationalization support）類別

所有這些組件的介面都十分簡單。這些組件在很多程式中都很重要。如今的資料處理工作，通常就是意味著輸入、計算、處理和輸出大量資料（通常是字串）。

這個標準程式庫的用法並非不言自明。想要從其強大能力中受惠，你需要一本好書；不能夠僅僅列出每一個類別和其函式了事，而是必須詳細解釋各組件的概念和重要細節。本書正是以此爲目標。本書首先從概念上介紹標準程式庫及其所有

組件，然後描述實際編程（programming）所需了解的細節。爲了展示組件的確切用法，書中還包括了大量實例。因此，這本書不論對初學者或是編程老手，都是極爲詳盡的 C++ 標準程式庫文件。以本書所提供的資料來武裝自己的頭腦，你就能從 C++ 標準程式庫中獲得最大利益。

注意，我不擔保本書所有內容都容易理解。標準程式庫非常靈活，但這種非同尋常的靈活性是有代價的。標準程式庫中有一些陷阱和缺陷，你必須小心應對。碰到它們時我會爲你指出，並提出一些建議，幫助你迴避問題。

1.2 閱讀前的必要基礎

要想讀懂本書的大部分內容，你需要先了解 C++。本書講述 C++標準組件，而不是語言本身。你應該熟悉類別（classes）、繼承（inheritance）、模板（templates）和異常處理（exception handling）的概念，但不必熟知語言的每一個細節。某些重要的細節本書也會講解（至於次要細節對程式庫實作者可能很重要，對程式庫使用者就不那麼重要了）。注意，在標準化過程中，C++ 語言發生了很大的變化，也許你的知識已經過時了。2.2 節簡單介紹了一些「使用標準程式庫時，你需要了解的最新語言特性」。如果你不確定自己是否了解 C++ 的新特性（例如關鍵字 `typename` 以及 namespace 概念），請先閱讀該節。

1.3 本書風格與結構

標準程式庫內的組件有相當程度的獨立性，但彼此又存在關聯，所以很難在描述某一部分時全然不提其他部分。我爲這本書考慮了幾種不同的組織方式。一是按照 *C++ standard* 的次序，但這並非完整介紹 C++ 標準程式庫的最佳選擇。另一種方式是，首先縱覽所有組件，再逐章詳細介紹。第三個方式則是，由我依照「組件之交叉參考」程度高低，從最低者開始介紹，逐次介紹最複雜的組件。最終，我綜合了三種方式：首先簡短介紹標準程式庫所涉及的總體概念和工具，然後分章詳述各個組件，每個組件一章或數章。首當其衝的便是 STL（Standard Template Library，標準模板庫）。STL 無疑是標準程式庫中最強大、最複雜、最激動人心的部分，其設計深刻影響了其他組件。接下來我再講解較易理解的組件，例如特殊容器、strings 和數值類別。再來是你或許已經使用多時的老朋友：IOStream 程式庫。最後是國際化議題的討論，這部分對於 IOStream 程式庫有些影響。

講述每個組件時，我首先給出該組件的目的、設計和範例，然後藉由各種使用方法和注意事項的描述，講解組件的細節。最後是個參考章節，你可以在其中找到組件類別和其函式的確切標記型式（exact signature）。

以下是本書內容。最前面的四章總體介紹了本書及 C++ 標準程式庫：

- 第 1 章：關於本書
  這一章（也就是你此刻正閱讀的）介紹本書的主題和內容。
- 第 2 章： C++ 和其標準程式庫簡介
  這一章對於 C++ 標準程式庫的歷史和背景進行簡短綜覽。此章也包括了本書及標準程式庫的技術背景的大致介紹，例如新的語言特性和所謂複雜度概念。
- 第 3 章 一般性概念（General Concepts）
  本章描述標準函式庫的基本概念，這些概念對於你理解和使用函式庫中的所有組件都是必須的。明確地說，本章介紹了 namaspace `std`、表頭檔（headers）格式、錯誤和異常處理（exception handling）的一般性支援。
- 第 4 章 通用工具（Utilities）
  本章描述數種提供給程式庫用戶和程式庫本身運用的小工具，更明確地說是 `max(),min(),swap()`等輔助函式，和 `pair,auto_ptr,numeric_limits` 等型別。上述最後一個型別提供了與實作品相依的數值型別相關資訊。

第 5 章至第 9 章分別從各個面向描述 STL：

- 第 5 章：Standard Template Library（STL，標準模板庫）
  STL 提供了用於處理資料的容器和演算法。本章詳細介紹 STL 的概念，並逐步解釋其中的概念、問題、特殊編程技術，以及它們所扮演的角色。
- 第 6 章：STL 容器（Containers）
  本章解釋 STL 容器類別的概念和能力。首先透過詳盡的例子，分別講解 vectors、deques、lists、sets 和 maps，然後介紹它們的共通能力。最後以簡明的形式列出並描述所有容器所提供的函式，做爲一份上手的參考資料。
- 第 7 章：STL 迭代器（Iterators）
  本章具體介紹了 STL 迭代器。解釋迭代器的分類和輔助函式，以及相應的配接器（iterator adapter）如 stream iterators、reverse iterators、insert iterators。
- 第 8 章：STL 仿函式（Functors，又名 Function Objects）
  本章詳細講解 STL 仿函式。
- 第 9 章：STL 演算法（Algorithms）
  本章羅列並描述 STL 演算法。在簡單介紹和比較這些演算法後，藉由一個（或多個）範例，對每個演算法進行詳細描述。

第 10 章至 12 章描述了一些「簡單的」標準類別：

- 第 10 章：特殊容器（Special Containers）
  本章描述 C++ 標準程式庫的各種特殊容器，並涵蓋容器配接器（container adapters）queues, stacks，以及 class `bitset`，後者可管理任意數量的 bits 或 flags。
- 第 11 章：Strings（字串）
  本章描述 C++ 標準程式庫的 string 型別（不止一種哦）。C++ *standard* 將 strings 設計爲一種「能夠處理不同字元類型」的基本資料型別，而且簡明易用。
- 第 12 章：數值（Numerics）
  本章描述 C++ 標準程式庫中的數值組件，包括複數（complex），以及一些用來處理數值陣列的類別（可用於矩陣 matrices、向量 vectors 和方程式 equations）。

第 13 章和第 14 章的主題是 I/O 和國際化（兩者緊密相關）：

- 第 13 章：以 Stream Classes 完成輸入和輸出
  本章涵蓋 C++ 的 I/O 組件。該組件是眾所周知的 IOStream 程式庫的標準化形式。本章也講述了一些對程式員而言可能很重要、但又鮮爲人知的細節。例如如何定義及整合特殊 I/O 通道，這是一個在實務過程中經常被搞錯的題目。
- 第 14 章：國際化（Internationalization, i18n）
  本章涵蓋「將程式國際化」的概念和類別，包括對不同字元集（character sets）的處理，以及如何使用不同格式的浮點數和日期。

剩餘部分包括：

- 第 15 章：配置器（Allocators）
  本章描述 C++ 標準程式庫中記憶體模型（memory model）的概念。
- 附錄，包括：
  — 網際網上的資源（Internet Resources）
  — 參考書目（Bibliography）
  — 索引（Index）

1.4 如何閱讀本書

本書既是介紹性的使用指南，又是 C++ 標準程式庫的結構化參考手冊。C++ 標準程式庫的各個組件在相當程度上是彼此獨立的，所以讀完第 2 章至第 4 章後，你可以按任意順序閱讀其他各章節。不過切記，第 5 章至第 9 章講述的是同一組東西。要理解 STL，應該從介紹性的第 5 章開始。

如果你是一位想總體認識 C++ 標準程式庫概念和其各個面向的程式員，可以簡略瀏覽這本書，並略過其參考章節（譯註：就是完整條列各個介面的那些小節）。當你需要運用某個 C++ 標準程式庫組件時，最好的辦法就是透過索引（index），找出相關資料。我已盡力把索引做得詳盡，希望能夠節省你的搜尋時間。

以我的經驗來看，學習新東西的最佳方式就是閱讀範例。因此，你會發現本書通篇有大量的範例，可能是幾行代碼，也可能是完整程式。如果是後者，註解第一行列有其檔案名稱。你可以在我的網站 `http://www.josuttis.com/libbook/` 找到這些檔案。

1.5 目前發展情勢

C++ standard 在我撰寫本書期間完成。請記住，有些編譯器可能還無法與之相容。這一點很可能在不久的將來得到很大的改善。但是現在，你可能會發現，本書所談的東西並非一定能夠在你的系統上有相同的表現，或許得稍做修改才能在你的環境中正常運作。我所使用的 EGCS 編譯器 2.8 版，及其更高版本，能夠編譯書中所有範例程式。該編譯器幾乎適用於所有計算機平台，你可以從網際網路（`http://egcs.cygnus.com/`）和某些軟體光碟中獲得。

1.6 範例程式碼及額外資訊

在我的網站 `http://www.josuttis.com/libbook/` 上，你可以獲得所有範例程式碼、本書及 C++ 標準程式庫的其他相關資訊。你也可以在網際網路上找到許多其他相關資訊，詳見 p743。

1.7 回饋

歡迎你對本書的任何回應（不論好的或壞的）。我已盡力而爲，但我畢竟是凡人，而且總有結稿的時候，所以難免有一些錯誤或前後不一的地方，需要進一步改善。你的回應將使我的新版本有機會進步。與我聯繫的最佳方式是透過電子郵件：

`libbook@josuttis.com`。

也可以透過電話、傳真、或「蝸牛般的」信件與我聯繫：

```
Nicolai M. Josuttis
Berggarten 9
D??38108 Braunschweig
Germany
Phone: +49 5309 5747
Fax: +49 5309 5774
```

非常感謝。

2

C++ 及其標準程式庫介紹

2.1 沿革

C++ 的標準化過程始於 1989 年，於 1997 年底完成。不過，由於某些原因，最終的標準規格遲至 1998 年 9 月才公佈，整個努力的成果就是國際標準化組織（ISO，International Organization for Standardization）發布的一份長達 750 頁的標準規格參考手冊。這份標準被命名爲 "Information Technology — Programming Language — C++"，文件編號 ISO/IEC 14882-1998，由 ISO 的各國成員機構發佈。例如美國的 ANSI 機構[1]。

標準規格的建立，是 C++ 的一個重要里程碑，它準確定義了 C++ 的內容和行爲，簡化了 C++ 的教學、使用、以及在不同平台之間的移植工作，給予用戶選擇不同 C++ 實作品（編譯器）的自由。它的穩定性和可移植性，對於程式庫、工具庫的供應者和實現者都是一樁佳音。因此，這份標準規格幫助 C++ 應用程式開發人員更快更好地創建應用程式，減輕維護上的精力。

標準程式庫是 C++ 標準規格的一部分，提供一系列核心組件，用以支援 I/O、字串（strings）、容器（資料結構）、演算法（排序、搜索、合併等等）、數值計算、國別（例如不同的字元集, character sets）等主題。

這個標準化過程竟花費了近 10 年的時間，未免讓人奇怪。如果你了解其中一些細節，更會奇怪爲什麼這麼久之後此一標準仍然未臻完美。十年其實不夠！儘管從標準化的歷史和內容來看，確實完成了許多東西，也形成了可在實務中應用的成果，但是距離完美尚遠（畢竟世間無完美之物）。

[1] 本書寫作之時，你可以花 18 美元從 ANSI Electronics Standard Store 獲得這份 C++ 標準文件（見 http://www.ansi.org/）。

這份標準並非某個公司花費大把鈔票和大量時間後的產物。那些爲制定標準而辛勤工作的人們，幾乎沒有從標準化組織那兒獲取任何報酬。對他們而言，如果他們所處的公司對 C++ 標準化漠不關心，那麼他們就只能以興趣爲全部的動力了。感天謝地，有這麼多勇於奉獻的人，能夠拿出時間和財力參與其中。

這份 C++ 標準規格並非從零開始，它以 C++ 創始人 Bjame Stroustrup 對於這個語言所作的描述爲基礎。然而標準程式庫並非基於某本書或某一個現成的函式庫，而是將各種不同的類別（classes）整合而成[2]，因此其結果並非十分地同質同種。你會發現不同組件背後有不同的設計原則，string class 和 STL 之間的差別就是很好的例子，後者是一個資料結構和演算法框架（framework）。

- string classes 被設計爲安全易用的組件，其介面幾乎不言自明，並能對許多可能的錯誤作檢驗。
- STL 的設計目標，是將不同的演算法和資料結構結合在一起，並獲取最佳效率，所以 STL 並不非常便利，也不檢驗許多可能的邏輯錯誤。要運用 STL 強大的框架和優異的效率，你就必須通曉其概念並小心運用。

標準程式庫中有一個組件，在標準化之前就已經作爲「準」標準而存在，那就是 IOStream 程式庫。這個東西於 1984 年開發，1989 年重做過一次，部分內容重新設計。由於很多程式員早就已經使用 IOStream，所以 IOStream 程式庫的概念沒有改變，保持回溯相容。

總體來說，整份標準規格（語言和程式庫）是在來自全球各地數百位人士的大量討論和影響下誕生的，例如日本人就是國際化（internationalization）組件的重要支持者。當然，我們曾經犯下錯誤，曾經改弦更張，曾經意見不一。到了 1994 年，當人們認爲這個標準接近完成時，STL 又被加了進來，此舉從根本上改變了整個程式庫。然而事情總要有個結束，終於有那麼一天，人們決定不再考慮任何重大擴張，無論這個擴張多麼有價值。就因爲這樣，hash tables 沒有被納入標準 — 盡管它作爲一種常用的資料結構，理應在 STL 中享有一席之地。

現有的標準並不是終極產品，每五年會有一個新版本，修正錯誤和不一致的地方。然而，起碼在接下來的數年中，C++ 程式員終於有了一個標準，有機會編寫功能強大並可移植到各種平台上的 C++ 程式了。

---

[2] 你可能會奇怪，爲什麼標準化過程中不從頭設計一個新的程式庫。要知道，標準化的主要目的不是爲了發明新東西或發展出某些東西，而是爲了讓既有的東西調和共處。

2.2 新的語言特性

C++ 語言核心和 C++ 程式庫是同時被標準化的，這麼一來，程式庫可以從語言的發展中受益，語言的發展也可以從程式庫的實作經驗中得到啓發。事實上，在標準化過程中，程式庫經常用到已規劃但當時尚未被實現出來的語言特性。

今天的 C++ 可不是五年前的 C++ 了。如果你沒有緊跟其發展，可能會對程式庫所使用的語言新特性大感驚訝。本節對這些新特性進行簡單的概括說明，至於細節，請查閱語言相關書籍。

我撰寫本書的時候（1998 年），並非所有編譯器都提供所有的語言新特性，這就限制了程式庫的使用。我希望（並預期）這種情況能很快獲得改善（絕大多數編譯器廠商都參與了標準化過程）。實作一份可移植的程式庫時，通常都要考慮你的電算環境是否支持你所用到的特性 — 通常我們會使用一些測試程式，檢查哪些語言特性獲得支援，然後根據檢驗結果設置前處理器指令（preprocessor directives）。我會在書中以註腳方式指出所有典型而重要的限制。

接下來數個小節描述和 C++ 標準程式庫有關的幾個最重要的語言新特性。

2.2.1 template（模板）

程式庫中幾乎所有東西都被設計爲 template 形式。不支援 template，就不能使用標準程式庫。此外程式庫還需要一些新的、特殊的 template 特性，我將在簡介之後詳細說明。

所謂 templates，是針對「一個或多個尚未明確的型別」所撰寫的函式或類別。使用 template 時，可以明白地（explicitly）或隱喻地（implicitly）將型別當作引數來傳遞。下面是一個典型例子，傳回兩數之中的較大數：

```
template <class T>
inline const T& max (const T& a, const T& b)
{
    // if a < b then use b else use a
    return a < b ? b : a;
}
```

在這裡，第一行將 `T` 定義爲任意資料型別，於函式被呼叫時由呼叫者指定。任何合法的識別符號都可以拿來作爲參數名稱，但通常以 `T` 表示，這差不多成了一個「準」標準。這個型別由關鍵字 `class` 引導，但型別本身不一定得是 class — 任何資料型別只要提供 template 定義式內所用到的操作，都可適用於此 template[3]。

[3]這裡使用關鍵字 `class`，原是爲了避免增加新的關鍵字。然而最終還是不得不引入一個新的關鍵字 `typename`。此處亦可使用關鍵字 `typename`（詳見 p11）。

遵循同樣原則，你可以將 class 參數化，並以任意型別做爲實際引數。這一點對容器類別非常有用。你可以實作出「有能力操控任意型別之元素」的容器。C++ 標準程式庫提供了許多 template container classes（詳見第 6 章和第 10 章）。標準函式庫對於 template classes 的使用，還有其他原因。例如可以以字元型別（character type）和字元集（character set）屬性，將 `string` class 參數化（詳見第 11 章）。

Template 並非一次編譯便生出適合所有型別的代碼，而是針對被使用的某個（或某組）型別進行編譯。這導致一個重要的問題：實際處理 template 時，面對 template function，你必須先提供它的某個實作品，然後才能呼叫，如此方可通過編譯。所以目前唯一能夠讓「template 的運用」具有可攜性的方式，就是在表頭檔中以 inline function 實現 template function[4]。

欲實現 C++ 標準程式庫的完整功能，編譯器不僅需要提供一般的 template 支援，還需要很多新的 template 標準特性，以下分別探討。

Nontype Templates 參數（非型別模板參數）

型別（type）可作爲 template 參數，非型別（nontype）也可以作爲 template 參數。非型別參數因而可被看作是整個 template 型別的一部分。例如可以把標準類別 `bitset<>`（本書 10.4 節, p460 會介紹它）的 bits 數量以 template 參數指定之。以下述句定義了兩個由 bits 構成的容器，分別爲 32 個 bits 空間和 50 個 bits 空間：

```
bitset<32> flags32;   // bitset with 32 bits
bitset<50> flags50;   // bitset with 50 bits
```

這些 bitsets 由於使用不同的 template 參數，所以有不同的型別，不能互相賦值（assign）或比較（除非提供了相應的型別轉換機制）。

Default Template Parameters（預設模板參數）

Template classes 可以有預設引數。例如以下宣告，允許你使用一個或兩個 template 引數來宣告 `MyClass` 物件[5]：

```
template <class T, class container = vector<T> >
class MyClass;
```

如果只傳給它一個引數，那麼預設參數可作爲第二引數使用：

```
MyClass<int> x1; // equivalent to: MyClass<int,vector<int> >
```

---

[4] 目前 template 必須定義於表頭檔中。爲了消除這個限制，標準規格導入了一個 template compilation model（模板編譯模型）和一個關鍵字 `export`。可惜據我所知，目前尚無任何編譯器實現出這一特性。

[5] 注意，兩個 ">" 之間必須有一個空格，如果你沒寫空格，">>" 會被解讀爲移位運算子（shift operator），導致語法錯誤。

注意，template 預設引數可根據前一個（或前一些）引數而定義。

關鍵字 typename

關鍵字 typename 被用來做爲型別之前的標識符號。考慮下面例子：

```
template <class T>
class MyClass
    typename T::SubType * ptr;
    ...
};
```

這裡，typename 指出 SubType 是 class T 中定義的一個型別，因此 ptr 是一個指向 T::SubType 型別的指標。如果沒有關鍵字 typename，SubType 會被當成一個 static 成員，於是：

```
T::SubType * ptr
```

會被解釋爲型別 T 內的數值 SubType 與 ptr 的乘積。

SubType 成爲一個型別的條件是，任何一個用來取代 T 的型別，其內都必須提供一個內隱型別（inner type）SubType 的定義。例如，將型別 Q 當做 template 參數：

```
MyClass<Q> x;
```

必要條件是型別 Q 有如下的內隱型別定義：

```
class Q
    typedef int SubType;
    ...
};
```

此時，MyClass<Q> 的 ptr 成員應該變成一個指向 int 型別的指標。子型別（subtype）也可以成爲抽象資料型別（例如 class）：

```
class Q
    class SubType;
    ...
};
```

注意，如果要把一個 template 中的某個標識符號指定爲一種型別，就算意圖顯而易見，關鍵字 typename 也不可或缺，因此 C++ 的一般規則是，除了以 typename 修飾之外，template 內的任何標識符號都被視爲一個實值（value）而非一個型別。

typename 還可在 template 宣告式中用來替換關鍵字 class：

```
template <typename T> class MyClass;
```

Member Template（成員模板）

class member function 可以是個 template，但這樣的 member template 既不能是 virtual，也不能有預設參數。例如：

```
calss MyClass
    ...
    template <class T>
    void f(T);
};
```

在這裡，MyClass::f 宣告了一個成員函式集，適用任何型別參數。只要某個型別提供有 f()用到的所有操作，它就可以被當做引數傳遞進去。這個特性通常用來爲 template classes 中的成員提供自動型別轉換。例如以下定義式中，assign()的參數 x，其型別必須和呼叫端所提供的物件的型別完全吻合：

```
template <class T>
class MyClass
    private:
      T value;
    public:
      void assign (const MyClass<T>& x) { // x must have same type as *this
          value = x.value;
      }
    ...
};
```

即使兩個型別之間可以自動轉換，如果我們對 assign() 使用不同的 template 型別，也會出錯：

```
void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // ERROR: i is MyClass<int>
                 //        but MyClass<double> is required
}
```

如果 C++ 允許我們爲 member function 提供不同（一個以上）的 template 型別，就可以放寬「必須精確吻合」這條規則；只要型別可被賦值（*assignable*），就可以被當做上述 member template function 的引數。

```
template <class T>
class MyClass
  private:
    T value;
```

```
  public:
    template <class X> // member template
    void assign (const MyClass<X>& x) { // allows different template types
      value = x.getValue();
    }
    T getValue () const
      return value;
  }
  ...
};

void f()
{
   MyClass<double> d;
   MyClass<int> i;

   d.assign(d);  // OK
   d.assign(i);  // OK (int is assignable to double)
}
```

請注意，現在，`assign()`的參數 `x` 和`*this` 的型別並不相同，所以你不再能夠直接存取 `MyClass<>` 的 private 成員和 protected 成員，取而代之的是，此例中你必須使用類似 `getValue()` 之類的東西。

Template constructor 是 member template 的一種特殊形式。Template constructor 通常用於「在複製物件時實現隱式型別轉換」。注意，template constructor 並不遮蔽（*hide*）implicit ***copy*** constructor。如果型別完全吻合，implicit ***copy*** constructor 就會被產生出來並被喚起。舉個例子：

```
template <class T>
class MyClass
   public:
     //copy constructor with implicit type conversion
     // - does not hide implicit copy constructor
     template <class U>
     MyClass (const MyClass<U>& x);
     ...
};

void f()
{
   MyClass<double> xd;
```

```
    ...
    MyClass<double> xd2(xd);    // calls built-in copy constructor
    MyClass<int> xi(xd);        // calls template constructor
    ...
}
```

在這裡，xd2 和 xd 的型別完全一致，所以它被內建的 *copy* ctor 初始化。xi 的型別和 xd 不同，所以它使用 template ctor 進行初始化。因此，撰寫 template ctor 時，如果 default *copy* ctor 不符合你的需要，別忘了自己提供一個 *copy* ctor。member template 的另一個例子詳見 p33, 4.1 節。

Nested Template Classes

巢狀的（*nested*）classes 本身也可以是個 template：

```
template <class T>
class MyClass
    ...
    template <class T2>
    class NestedClass;
    ...
};
```

2.2.2 基本型別的顯式初始化（explicit initialization）

如果採用不含參數的、明確的 constructor（建構式）呼叫語法，基本型別會被初始化爲零：

```
int i1;          // undefined value
int i2 = int();  // initialized with zero
```

這個特性可以確保我們在撰寫 template 程式碼時，任何型別都有一個確切的初值。例如下面這個函式中，x 保証被初始化爲零。

```
template <class T>
void f()
{
    T x = T();
    ...
}
```

2.2.3 異常處理（Exception Handling）

藉由異常處理，C++ 標準程式庫可以在不「污染」函式介面（亦即參數和回返值）的情況下處理異常。如果你遇到一個意外情況，可以藉由「丟出一個異常」來停止一般的（正常的）處理過程：

```
class Error;
void f()
{
  ...
  if (exception-condition)
      throw Error(); // create object of class Error and throw it as exception
  }
  ...
}
```

述句 `throw` 開始了 *stack unwinding*（堆疊輾轉開解）過程，也就是說，它將導致退離任何函式區段，就像遭遇 `return` 述句一般，然而程式卻不會跳轉到任何地點。對於所有被宣告於某區段 — 而該區段卻因程式異常而退離 — 的區域物件而言，其 destructor（解構式）會被喚起。*Stack unwinding* 的動作會持續直到退出 `main()` 或直到有某個 `catch` 子句捕捉並處理了該異常爲止。如果是第一種情況，程式會結束：

```
int main()
{
    try
        ...
        f();
        ...
    }
    catch (const Error&)
        ...   // handle exception
    }
    ...
}
```

在這裡， `try` 區段中任何「型別爲 `Error` 的異常」都將在 `catch` 子句獲得處理[6]。

[6] 異常（exception）會終結某個函式呼叫動作，而該函式正是異常發生之處。異常處理機制有能力將某個物件當作引數，回傳給函式呼叫者。但這並非反向函式回呼（call back）。所謂反向是指從「問題被發現的地方」到「問題被處理的地方」，也就是從下往上的方向。以此觀之，異常處理和信號處理（signal handling）完全是兩碼事。

異常物件（exception objects）其實就是一般類別或基本型別的物件，可以是 ints、strings，也可以是類別體系中的某個 template classes。通常你會設計一個特殊的 error 類別體系。你可以運用異常物件的狀態（state），將任何資訊從錯誤被偵測到的地點帶往錯誤被處理的地點。

注意，這個概念叫作「異常處理（*exception handling*）」，而不是「錯誤處理（*error handling*）」，兩者未必相同。舉個例子，許多時候用戶的無效輸入（這經常發生）並非一種異常；這時候最好是在區域範圍內採用一般的錯誤處理技術來處理。

你可以運用所謂的異常規格（*exception specification*）來指明某個函式可能拋出哪些異常，例如：

```
void f() throw(bad_alloc);  // f()只可能丟出 bad_alloc 異常
```

如果宣告一個空白異常規格，那就表明該函式不會丟擲任何異常：

```
void f() throw();       // f()不丟擲任何異常
```

違反異常規格將會導致特殊行爲，詳見 p26 關於異常類別 *bad_exception* 的描述。

C++ 標準程式庫提供了一些通用的異常處理特性，例如標準異常類別（standard exception classes）和 auto_ptr 類別（詳見 p25, 3.3 節和 p38, 4.2 節）。

2.2.4 命名空間（Namespaces）

越來越多的軟體以程式庫、模組（modules）和組件拼湊完成。各種不同事物的結合，可能導致一場名稱大衝突。Namespaces 正是用來解決此一問題。

Namespaces 將不同的標識符號集合在一個具名範圍內。如果你在 namespace 之內定義所有標識符號，則 namespace 本身名稱就成了唯一可能與其他全域符號衝突的標識符號。你必須在標識符號前加上 namespace 名字，才能援引該 namespace 內的符號，這和 class 處理方式雷同。namespace 的名字和標識符號之間以 :: 分隔開來（譯註：這個符號及其意義和 class 與其 members 之間的聯繫有點類似）。

```
// defining identifiers in namespace josuttis
namespace josuttis
    class File;
    void myGlobalFunc();
    ...
}
...
// using a namespace identifier
josuttis::File obj;
...
josuttis::myGlobalFunc();
```

不同於 class 的是，namespaces 是開放的，你可以在不同模組（modules）中定義

和擴展 namespace。因此你可以使用 namespace 來定義模組、程式庫或組件，甚至在多個檔案之間完成。Namespace 定義的是邏輯模組，而非實質模組。請注意，在 UML 及其他建模表示法（modeling notations）中，模組又被稱爲 *package*。

如果某個函式的一個或多個引數型別，乃定義於函式所處的 namespace 中，那麼你不必爲該函式指定 namespace。這個規則稱爲 *Koenig lookup*（譯注：Andrew Koenig 是 C++ 社群代表人物之一，對 C++ 的形成和發展有重要貢獻）。例如：

```
// defining identifiers in namespace josuttis
namespace josuttis
    class File;
    void myGlobalFunc(const File&);
    ...
}
...

josuttis::File obj;
...
myGlobalFunc(obj); // OK, lookup finds josuttis::myGlobalFunc()
```

藉由 *using declaration*，我們可以避免一再寫出冗長的 namespace 名稱。例如以下宣告式：

```
using josuttis::File;       // 譯註：這就是一個 using declaration
```

會使 `File` 成爲當前範圍（current scope）內代表 `josuttis::File` 的一個同義字。

using directive 會使 namespace 內的所有名字曝光。*using directive* 等於將這些名字宣告於 namespace 之外。但這麼一來，名稱衝突問題就可能死灰復燃。例如：

```
using namespace josuttis; // 譯註：這就是一個 using directive
```

會使 `File` 和 `MyGlobalFunc()` 在當前範圍內完全曝光。如果全域範圍內已存在同名的 `File` 或 `MyGlobalFunc()`，而且使用者不加任何資格飾詞（qualification）地使用這兩個名字，編譯器將東西難辨。

注意，如果脈絡環境（context）不甚清楚（例如不清楚究竟在表頭檔、模組或程式庫中），你不應該使用 *using directive*。這個指令可能會改變 namespace 的作用範圍，從而使程式碼被包含或使用於另一模組中，導致意外行爲的發生。事實上在表頭檔中使用 *using directive* 相當不智。

C++ 標準程式庫在 namespace `std` 中定義了它的所有標識符號，詳見 p23, 3.1 節。

2.2.5 bool 型別

爲了支持布林值（真假值），C++ 增加了 bool 型別。 bool 可增加程式的可讀性，並允許你對布林值實現重載（*overloaded*）動作。兩個常數 true 和 false 同時亦被引入 C++。此外 C++ 還提供布林值與整數值之間的自動轉換。0 值相當於 false，非 0 值相當於 true。

2.2.6 關鍵字 explicit

藉由關鍵字 explicit 的作用，我們可以禁止「單引數建構式（single argument constructor）」被用於自動型別轉換。典型的例子便是群集類別（collection classes），你可以將初始長度作爲引數傳給建構式，例如你可以宣告一個建構式，以 stack 的初始大小爲參數：

```
class Stack
   explicit Stack(int size); // create stack with initial size
   ...
};
```

在這裡，explicit 的應用非常重要。如果沒有 explicit，這個建構式有能力將一個 int 自動轉型爲 Stack。一旦這種情況發生，你甚至可以給 Stack 指派一個整數值而不會引起任何問題：

```
Stack s;
...
s = 40; // Oops, creates a new Stack for 40 elements and assigns it to s
```

「自動型別轉換」動作會把 40 轉換爲有 40 個元素的 stack，並指派給 s，這幾乎肯定不是我們所要的結果。如果我們將建構式宣告爲 explicit，上述賦值（指派，assign）動作就會導致編譯錯誤（那很好）。

注意，explicit 同樣也能阻絕「以賦值動作進行帶有轉型現象之初始化行爲」：

```
Stack s1(40);     // OK
Stack s2 = 40;    // ERROR
```

這是因爲以下兩組動作：

```
X x;
Y y(x);      // explicit conversion
```

和

```
X x;
Y y = x;     //implicit conversion
```

存在一個小差異。前者透過顯式轉換，根據型別 X 產生了一個型別 Y 的新物件；後者透過隱式轉換，產生了一個型別 Y 的新物件。

2.2.7 新的型別轉換運算子（Type Conversion Operators）

爲了讓你對「引數的顯式型別轉換」更透徹，C++ 引入以下四個新的運算子：

1. `static_cast`

   將一個值以合邏輯的方式轉型。這可看做是「利用原值重建一個臨時物件，並在設立初值時使用型別轉換」。唯有當上述的型別轉換有所定義，整個轉換才會成功。所謂的「有所定義」，可以是語言內建規則，也可以是程式員自定的轉換動作。例如：

   ```
   float x;
   ...
   cout << static_cast<int>(x);   // print x as int
   ...
   f(static_cast<string>("hello")); // call f() for string instead of char*
   ```

2. `dynamic_cast`

   將多形型別（polymorphic type）向下轉型（downcast）爲其實際靜態型別（real static type）。這是唯一在執行期進行檢驗的轉型動作。你可以用它來檢驗某個多形實值（polymorphic value）的型別，例如：

   ```
   class Car; // abstract base class (has at least one virtual function)

   class Cabriolet : public Car
   ...
   };

   class Limousine : public Car
   ...
   };

   void f(Car* cp)
   {
      Cabriolet* p = dynamic_cast<Cabriolet*>(cp);
      if (p == NULL)
         // did not refer to an object of type Cabriolet
         ...
      }
   }
   ```

   在這個例子中，面對實際靜態型別爲 `Cabriolet` 的物件，`f()` 有特殊應對行爲。當引數是個 reference，而且型別轉換失敗時，`dynamic_cast` 丟出一個 ***bad_cast*** 異常（***bad_cast*** 的描述見 p26）。注意，以設計角度而言，你應該在運用多型技術的程式中，避免這種「程式行爲取決於具體型別」的寫法。

3. const_cast

   設定或去除型別的常數性（constness），亦可去除 volatile 飾詞。除此之外不允許任何轉換。

4. reinterpret_cast

   此運算子的行爲由實作品（編譯器）定義。可能重新解釋 bits 意義，但也不一定如此。使用此一轉型動作通常帶來不可移植性。

這些運算子取代了以往小圓括號所代表的舊式轉型，能夠清楚闡明轉型的目的。小圓括號轉型可替換 dynamic_cast 之外的其他三種轉型，也因此當你運用它時，你無法明確顯示使用它的確切理由。這些新式轉型運算子給了編譯器更多資訊，讓編譯器清楚知道轉型的理由，並在轉型失敗時釋出一份錯誤報告。

注意，這些運算子都只接受一個引數。試看以下例子：

```
static_cast<Fraction>(15,100) // Oops, creates Fraction(100)
```

在這個例子中你得不到你想要的結果。它只用一個數值 100，將 Fraction 暫時物件設定初值，而非設定分子 15、分母 100。逗號在這裡並不起分隔作用，而是形成一個 comma（逗號）運算子，將兩個算式組合爲一個算式，並傳回第二算式做爲最終結果。將 15 和 100「轉換」爲分數的正確做法是：

```
Fraction(15,100) // fine, creates Fraction(15,100)
```

2.2.8 常數靜態成員（Constant Static Members）的初始化

如今，我們終於能夠在 class 宣告式中對「整數型（integral）常數靜態成員」直接賦予初值。初始化後，這個常數便可用於 class 之中，例如：

```
class MyClass
    static const int num = 100;
    int elems[num];
    ...
};
```

注意，你還必須爲 class 之中宣告的常數靜態成員，定義一個空間：

```
const int MyClass::num;    // no initialization here
```

2.2.9 main() 的定義式

我樂於澄清這個語言中一個重要而又常被誤解的問題，那就是正確而可移植的 main() 的唯一寫法。根據 C++標準規格，只有兩種 main() 是可移植的：

```
int main()
{
    ...
}
```

和

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

這裡 argv（命令列參數陣列）也可定義爲 char**。請注意，由於不允許「不言而喻」的回返型別 int，所以回返型別必須明白寫爲 int。你可以使用 return 述句來結束 main()，但不必一定如此。這一點和 C 不同，換句話說，C++ 在 main() 的尾端定義了一個隱喻的：

```
return 0;
```

這意味如果你不採用 return 述句離開 main()，實際上就表示成功退出（傳回任何一個非零值都代表某種失敗）。出於這個原因，本書範例在 main() 尾端都沒有 return 述句。有些編譯器可能會對此發出警告（譯註：例如 Visual C++），有的甚至認爲這是錯誤的。唔，那正是標準制定前的黑暗日子。

2.3 複雜度和 Big-O 表示法

對於 C++ 標準程式庫的某些部分（特別是 STL），演算法和成員函式的效率需要嚴肅考慮，因此需要動用「複雜度」的概念。計算機科學家運用特定符號，比較演算法的相對複雜度，如此便可以很快依據演算法的運行時間加以分類，進行演算法之間的定性比較。這種衡量方法叫做 Big-O 表示法。

Big-O 表示法係將一個演算法的運行時間以輸入量 n 的函數表示。例如，當運行時間隨元素個數成線性增長時（亦即如果元素個數呈倍數增長，運行時間亦呈倍數增長），複雜度爲 O(n)；如果運行時間獨立於輸入量，複雜度爲 O(1)。表 2.1 列出典型的複雜度和其 Big-O 表示法。

請注意，Big-O 表示法隱藏了（忽略了）指數較小的因子（例如常數因子），這一點十分重要，更明確地說，它不關心演算法到底耗用多長時間。根據這種量測法則，任何兩個線性演算法都被視爲具有相同的接受度。甚至可能發生一種情況：帶有巨大常數的線性演算法竟然比帶有小常數的指數演算法受到歡迎（譯註：因

爲 Big-O 複雜度表示法無法顯現真實的運算時間）。這是對 Big-O 表示法的一種合理批評。記住，這只是一種量度規則。具有最佳（最低）複雜度的演算法，不一定就是最好（最快）的演算法。

型別	表示法	含義
常數	O(1)	運行時間與元素個數無關。
對數	O(log(n))	運行時間隨元素個數的增加呈對數增長。
線性	O(n)	運行時間隨元素個數的增加呈線性增長。
n-log-n	O(n*log(n))	運行時間隨元素個數的增加呈「線性和對數的乘積」增長。
二次	$O(n^2)$	運行時間隨元素個數的增加呈平方增長。

表 2.1 典型的五種複雜度

表 2.2 列出所有複雜度分類，並以某些元素個數來說明運行時間隨元素個數增長的程度。一如你所看到，當元素較少時，運行時間的差別很小，此時 Big-O 表示法所隱藏的常數因子可能會帶來很大影響。但是當元素個數愈多，運行時間差別愈大，常數因子也就變得無關緊要了。當你考慮複雜度時，請記住，輸入量必須夠大才有意義。

複雜度		元素數目						
型別	表示法	1	2	5	10	50	100	1000
常數	O(1)	1	1	1	1	1	1	1
對數	O(log(n))	1	2	3	4	6	7	10
線性	O(n)	1	2	5	10	50	100	1,000
n-log-n	O(n*log(n))	1	4	15	40	300	700	10,000
二次方	$O(n^2)$	1	4	25	100	2,500	10,000	1,000,000

表 2.2 運行時間、複雜度、元素個數 對照表

C++ 標準手冊中的某些複雜度被稱爲 *amortized*（分期攤還），意思是，長期而言，大量操作將如上述描述般進行，但單一操作卻可能花費比平均值更長的時間。舉個例子，如果你爲一個 dynamic array 追加元素，運行時間將取決於 array 是否尚有備用記憶體。如果記憶體足夠，就屬於常數複雜度，因爲在尾端加入一個新元素，總是花費相同時間。如果備用記憶體不足，那麼就是線性複雜度，因爲你必須配置足夠的記憶體並搬動（複製）它們，實際耗用時間取決於當時的元素個數。記憶體重新配置動作並不常發生（譯註：STL 的 dynamic array 容器會以某種哲學來保持備用記憶體），所以任何「長度充份」的序列（sequence），元素附加動作幾乎可說是常數複雜度。這種複雜度我們便稱爲 *amortized*（分期攤還）常數時間。

3

一般概念

General Concepts

本章講述 C++ 標準程式庫中的基本概念。幾乎所有 C++ 標準程式庫組件都需要這些概念。

- 命名空間（namespace）std
- 表頭檔（headers）的名稱與格式
- 錯誤（error）和異常（exception）處理的一般概念
- 配置器（allocator）的簡單介紹

3.1 命名空間（namespace）std

當你採用不同的模組和程式庫時，經常會出現名稱衝突現象，這是因爲不同的模組和程式庫可能針對不同的物件使用相同的標識符號。namespaces（參見 p16, 2.2.4 節的介紹）用來解決這個問題。所謂 namespace，是指標識符號的某種可見範圍。和 class 不同，namespace 具有擴展開放性，可以發生於任何源碼檔案上。因此你可以利用一個 namespace 來定義一些組件，而它們可散佈於多個實質模組上。這類組件的典型例子就是 C++ 標準程式庫，因爲 C++ 標準程式庫使用了一個 namespace。事實上，C++ 標準程式庫中的所有標識符號都被定義於一個名爲 std 的 namespace 中。

由於 namespace 的概念，使用 C++ 標準程式庫的任何標識符號時，你有三種選擇：

1. 直接指定標識符號。例如 std::ostream 而不是 ostream。完整語句類似這樣：
   ```
   std::cout << std::hex << 3.4 << std::endl
   ```
2. 使用 *using declaration*（詳見 p17）。例如以下程式片段使我們不必再寫出範圍修飾符號 std::，而可直接使用 cout 和 endl：
   ```
   using std::cout;
   using std::endl;
   ```
   於是先前的例子可以寫成這樣：

```
cout << std::hex << 3.4 << endl;
```

3．使用 *using directive*（詳見 p17），這是最簡便的選擇。如果對 namespace `std` 採用 *using directive*，便可以讓 `std` 內定義的所有標識符號都有效（曝光），就好像它們被宣告爲全域標識符號一樣。因此，寫下：

```
using namespace std;
```

之後，就可以直接寫：

```
cout << hex << 3.4 << endl;
```

但請注意，由於某些晦澀的重載（overloading）規則，在複雜的程式中，這種方式可能導致意外的命名衝突，更糟的是甚至導致不一樣的行爲。如果場合不夠明確（例如在表頭檔、模組或程式庫中），就應避免使用 *using directive*。

本書的例子都很小，所以，爲了我自己方便，書中範例程式通常採用最後一種手法。

3.2 表頭檔（Header Files）

將 C++ 標準程式庫中所有標識符號都定義於 namespace `std` 裡頭，這種做法是標準化過程中引入的。這個作法不具回溯相容性，因爲原先的 C/C++ 表頭檔都將 C++ 標準程式庫的標識符號定義於全域範圍（global scope）。此外標準化過程中有些 classes 的介面也有了更動（當然啦，儘可能以回溯相容爲目標）。爲此，特別引入了一套新的表頭檔命名風格，這麼一來組件供應商得以藉由「提供舊的表頭檔」來達到回溯相容目的。

既然有必要重新定義標準表頭檔的名稱，正好藉此機會把表頭檔副檔名做個規範。以往，表頭檔副檔名五花八門，包括 .h, .hpp, .hxx。相較之下如今的標準表頭檔副檔名簡潔得令人吃驚：根本就沒有副檔名。於是標準表頭檔的 `#include` 如下：

```
#include <iostream>
#include <string>
```

這種寫法也適用於 C 標準表頭檔。但必須採用前綴字元 `c`，而不再是副檔名 `.h`：

```
#include <cstdlib>    // was: < stdlib.h>
#include <cstring>    // was: < string.h>
```

在這些表頭檔中，每一個標識符號都被宣告於 namespace `std`。

這種命名方式的優點之一是可以區分舊表頭檔中的 `char*` C 函式，和新表頭檔中的標準 C++ `string` class：

```
#include <string>     // C++ class string
#include <cstring>    // char* functions from C
```

注意，以作業系統角度觀之，新表頭檔命名方式並非意味標準表頭檔沒有副檔名。標準表頭檔的 `#include` 該如何處理，由編譯器決定。C++ 系統可以自動添加一個副檔名，甚至可以使用內建宣告，不讀入任何檔案。不過實際上大多數系統只是簡單含入一個「名稱與 `#include` 句中的檔名完全相同」的檔案。所以，在大部份系統中，C++ 標準表頭檔都沒有副檔名。注意，「無副檔名」這一條件只適用於標準表頭檔。一般而言，爲你自己所寫的表頭檔加上一個良好的副檔名，仍然是個好主意，有助於輕易識別出這些檔案的性質。

爲了回溯相容於 C，舊式的 C 標準表頭檔仍然有效，如果需要，你還是可以使用它們，例如：

```
#include <stdlib.h>
```

此時，標識符號同時宣告於全域範圍和 namespace `std` 中。事實上這些表頭檔的行爲類似於先在 `std` 中宣告所有標識符號，再悄悄使用 *using declaration* 把這些標識符號引入全域範圍（參見 p17）。

至於 `<iostream.h>` 這一類 C++ 舊式表頭檔，標準規格中並未加以規範（這一點在標準化過程中曾經多次改變），意味不再支持這些表頭檔。不過目前大多數廠商都會提供它們，以求回溯相容。

注意，除了引入 namespace `std`，表頭檔還有很多改變。所以，你要嘛就採用表頭檔舊名，要嘛就應該完全改用新的標準名稱。

3.3 錯誤（Error）處理和異常（Exception）處理

C++ 標準程式庫由不同的成份構成。來源不同，設計與實現風格迥異。而錯誤處理和異常處理正是這種差異的典型表徵。標準程式庫中有一部分，例如 string classes，支援具體的錯誤處理，它們檢查所有可能發生的錯誤，並於錯誤發生時丟出異常。至於其它部分如 STL 和 valarrays，效率重於安全，因此幾乎不檢驗邏輯錯誤，並且只在執行期（runtime）發生錯誤時才丟擲異常。

3.3.1 標準異常類別（Standard Exception Classes）

語言本身或標準程式庫所丟擲的所有異常，都衍生自基礎類別 `exception`。這是其他數個標準異常類別的基礎類別，他們共同構成一個類別體系，如圖 3.1。這些標準異常類別可分爲三組：

1. 語言本身支援的異常
2. C++ 標準程式庫發出的異常
3. 程式作用域（scope of a program）之外發出的異常

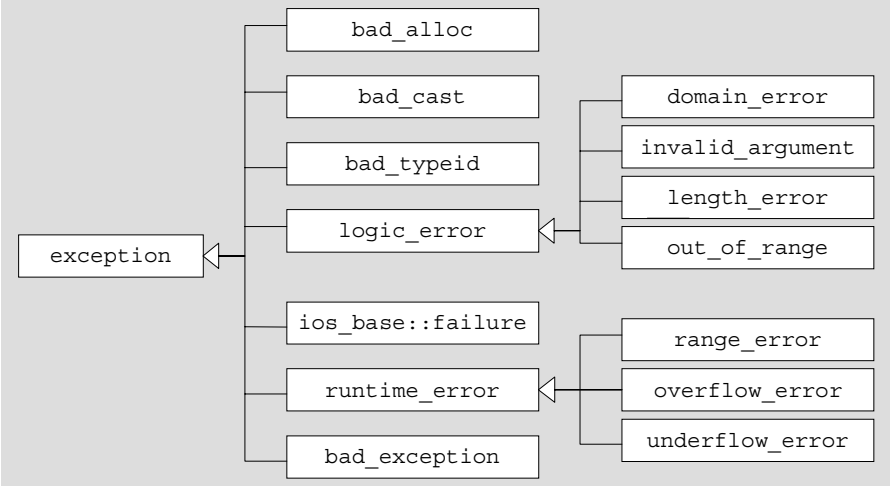


圖 3.1 標準異常（Standard Exceptions）階層體系

語言本身所支援的異常

此類異常用以支撐某些語言特性，所以，從某種角度來說它們不是標準程式庫的一部分，而是核心語言的一部分。如果以下操作失敗，就會丟擲這一類異常。

- 全域運算子 `new` 操作失敗，會丟擲 *bad_alloc* 異常（若採用 `new` 的 `nothrow` 版本，另當別論）。由於這個異常可能於任何時間在任何較複雜的程式中發生，所以可說是最重要的一個異常。
- 執行期間，當一個加諸於 reference 身上的「動態型別轉換動作」失敗時，`dynamic_cast` 會丟擲 *bad_cast* 異常。p19 對於 `dynamic_cast` 運算子有些描述。
- 執行期型別辨識（RTTI）過程中，如果交給 `typeid` 的引數爲零或空指標，`typeid` 運算子會丟擲 *bad_typeid* 異常。
- 如果發生非預期的異常，*bad_exception* 異常會接手處理，方式如下：當函式擲出異常規格（exception specification，p16 介紹）以外的異常，*bad_exception* 就會喚起 `unexpected()`。例如：

```
class E1;
class E2; // not derived from E1
```

```
void f() throw(E1) // throws only exceptions of type E1
{
    ...
    throw E1(); // throws exception of type E1
    ...
    throw E2(); // calls unexpected(), which calls terminate()
}
```

`f()`之中丟擲出「型別爲 *E2*」的異常，這種動作違反了異常規格（exception specification）的設定，於是喚起`unexpected()`，後者通常會喚起`terminate()`終止程式。

然而如果你在你的異常規格中列出 *bad_exception*，那麼`unexpected()` 總是會重新擲出（rethrows）*bad_exception* 異常。

```
class E1;
class E2;    // not derived from E1

void f() throw(E1, std::bad_exception)
                // throws exception of type E1 or
                // bad_exception for any other exception type
{
    ...
    throw E1();   // throws exception of type E1
    ...
    throw E2();   // calls unexpected(),which throws bad_exception
}
```

因此，如果異常規格羅列了 *bad_exception*，那麼任何未列於規格的異常，都將在函式`unexpected()`中被代之以 *bad_exception*[1]。

C++ 標準程式庫所發生的異常

C++ 標準程式庫異常總是衍生自 *logic_error*。理論而言，我們能夠透過一些手段，在程式中避免邏輯錯誤 — 例如對函式引數進行額外測試等等。所謂邏輯錯誤包括違背邏輯前提或違反 class 的不變性。C++ 標準程式庫提供以下邏輯錯誤類別：

- *invalid_argument* 表示無效引數，例如將`bitset`（array of bits）以`char`而非 '0' 或 '1' 進行初始化。
- *length_error* 指出某個行爲「可能超越了最大極限」，例如對著某個字串附加太多字元。

[1] 你可以修改`unexpected()`的具體操作。然而只要宣告有異常規格，函式就絕不會擲出規格中未列的異常。

- *out_of_range* 指出引數值「不在預期範圍內」，例如在諸如 array 的群集（collection）或字串（string）中採用一個錯誤索引。
- *domain_error* 指出專業領域範疇內的錯誤。

此外，標準程式庫的 I/O 部分提供了一個名爲 *ios_base::failure* 的特殊異常。當資料串流（data stream）由於錯誤或由於到達檔案尾端而發生狀態改變時，就可能丟擲這個異常。此一異常的具體行爲見 p602, 13.4.4 節。

程式作用域（scope of a program）之外發生的異常

衍生自 *runtime_error* 的異常，用來指出「不在程式範圍內，且不容易迴避」的事件。C++ 標準程式庫針對執行期錯誤提供了以下三個 classes：

- *range_error* 指出內部計算時發生區間錯誤（range error）。
- *overflow_error* 指出算術運算發生上溢位（overflow）。
- *underflow_error* 指出算術運算發生下溢位（underflow）。

標準程式庫所丟擲的異常

C++ 標準程式庫自身可能丟擲 *range_error*、*out_of_range* 和 *invalid_argument* 異常。然而由於標準程式庫會用到語言特性及客戶所寫的程式碼，所以也可能間接擲出任何異常。尤其是，無論何時配置儲存空間，都有可能擲出 *bad_alloc* 異常。

標準程式庫的任何具體實作品，都可能提供額外的異常類別（或作爲兄弟類別，或衍生爲子類別）。使用這些非標準類別將導致程式難以移植，因爲一旦你想採用其他標準程式庫實作版本，就不得不痛苦地修改你的程式。所以最好只使用標準異常。

異常類別的表頭檔

基礎類別 *exception* 和 *bad_exception* 定義於 `<exception>`。*bad_alloc* 定義於 `<new>`。*bad_cast* 和 *bad_typeid* 定義於 `<typeinfo>`。*ios_base::failure* 定義於 `<ios>`。其他異常類別都定義於 `<stdexcept>`。

3.3.2 異常類別（Exception Classes）的成員

爲了在 `catch` 子句中處理異常，你必須採用異常所提供的介面。所有標準異常的介面只含一個成員函式：`what()`，用以獲取「型別本身以外的附加資訊」。它傳回一個以 null 結束的字串：

```
namespace std {
   class exception {
      public:
```

```
        virtual const char* what() const throw();
        ...
    };
}
```

被傳回的字串，其內容由實作廠商定義。它很大程度（但非必然）決定了幫助的級別和資訊的詳細度。注意，該字串有可能是個以 null 結尾的 "multibyte" 字串，可被輕鬆轉換爲 `wstring`（詳見 p480, 11.2.1 節）並顯示出來。`what()` 傳回的 C-string 在其所屬的異常物件被摧毀後，就不再有效了[2]。

標準異常中的其他成員，用來處理生成、複製、賦值、摧毀等動作。要注意的是，除了 `what()`，再沒有任何異常提供任何其他成員函式，能夠描述異常的種類。例如，沒有可找出異常脈絡（context）的一致性方法，或找出區間錯誤（range error）發生時的錯誤索引值。因此，唯一通用的異常評估手段，大概只有列印一途了：

```
try {
    ...
}
catch (const std::exception& error) {
    // print implementation-defined error message
    std::cerr << error.what() << std::endl;
    ...
}
```

唯一可能實現的另一個異常評估手段是，根據異常的精確型別，自己得出推論。例如，如果 *bad_alloc* 異常被擲出，可能是因爲程式企圖獲得更多記憶體。

3.3.3 ﾖ､擲標準異常

你可以在自己的程式庫或程式內部丟擲某些標準異常。允許你這般運用的各個標準異常，生成時都只需要一個 `string` 參數（第 11 章對 `string` class 有所描述），它將成爲被 `what()` 傳回的描述字串。例如 *logic_error* 定義如下：

```
namespace std {
    class logic_error : public exception {
        public:
        explicit logic_error (const string& whatString);
    };
}
```

[2] C++ 標準規格並未對 `what()` 回返值的壽命加以規範，這裡所講的是被提出的一種建議解決方案。

提供這種功能的標準異常有：*logic_error* 及其衍生類別、*runtime_error* 及其衍生類別、*ios_base::failure*。你不能丟擲 *exception*，也不能丟擲任何用以支援語言核心性質的異常。

想要丟擲一個標準異常，只需生成一個描述該異常的字串，並將它初始化，交給異常物件：

```
std::string s;
...
throw std::out_of_range(s);
```

由於 `char*` 可被隱晦轉換爲 `string`，所以你可以直接使用字串字面常數：

```
throw std::out_of_range("out_of_range (somewhere, somehow)");
```

3.3.4 從標準異常類別（Exception Classes）衍生新類別

另一個在程式中採用標準異常類別的可能情況是，定義一個直接或間接衍生自 *exception* 的特定異常類別。要這麼做，首先必須確保 `what()` 機制正常運作。`what()` 是個虛擬函式，所以提供 `what()` 的方法之一就是自己實現 `what()`：

```
namespace MyLib {
  /* user-defined exception class
  * derived from a standard class for exceptions
  */
  class MyProblem : public std::exception {
    public:
    ...
    MyProblem(...) { // special constructor
    }
    virtual const char* what() const throw() { // what() function
      ...
    }
  };
  ...

  void f() {
    ...
    // create an exception object and throw it
```

```
    throw MyProblem(...);
    ...
  }
}
```

提供 what()函式的另一種方法是，令你的異常類別衍生自 3.3.3 節所描述的標準異常：

```
namespace MyLib {
  /* user-defined exception class
  * - derived from a standard class for exceptions
  * that has a constructor for the what() argument
  */

  class MyRangeProblem : public std::out_of_range {
    public:
    MyRangeProblem (const string& whatString)
      : out_of_range(whatString) {
    }
  };
  ...

  void f() {
  ...
  // create an exception object by using a string constructor and throw it
  throw MyRangeProblem("here is my special range problem");
  ...
  }
}
```

完整程式見 p441 的 Stack 和 p450 的 Queue。

3.4 配置器（Allocators）

C++ 標準程式庫在許多地方採用特殊物件來處理記憶體配置和定址，這樣的物件稱爲配置器（allocator）。配置器表現出一種特定的記憶體模型（memory model），成爲一個抽象表徵，表現出「記憶體需求」至「記憶體低階呼叫」的轉換。如果運用多個不同的配置器物件，你便可以在同一個程式中採用不同的記憶體模型。

配置器最初是作爲 STL 的一部份而引進，用於處理諸如 PC 上不同指標型別（例如 near, far, huge 指標）這一類亂七八糟的問題；現在則作爲一種技術方案的基礎，

使得諸如共享記憶體（shared memory）、垃圾回收（garbage collection）、物件導向資料庫（object oriented databases）等特定記憶體模型，能夠保持一致的介面。但是這種用法還相當新穎，尚未獲得廣泛的接受（情況正在改變中）。

C++ 標準程式庫定義了一個預設配置器（default allocator）如下：

```
namespace std {
    template <class T>
    class allocator;
}
```

預設配置器可在任何「配置器得以被當作引數使用」的地方擔任預設值。預設配置器會執行記憶體配置和回收的一般性手法，也就是呼叫 `new` 和 `delete` 運算子。但是 C++ 並沒有對於「在什麼時候以什麼方式調用這些運算子」給予明確規定。所以，預設配置器甚至可能對已配置之記憶體施行「內部快取（internal cache）」手法。

絕大多數程式都採用預設配置器，但有時候其他程式庫也可能提供某些配置器以滿足特定需求。這種情況下只需簡單地將它們當作引數即可。自行設計並實作配置器的實際意義不大。實際生活中最典型的方式還是直接採用預設配置器，所以我將遲至第 15 章才詳細探討配置器。（譯註：舉個例子，SGI STL 對外呈現一個透通的配置器介面，內部卻維護有一、二兩級配置器，製作出十分精巧繁複的 memory pool 機制，對於體積小而數量極大的物件需求而言，可帶來極好的時間和空間效率。詳見《STL 源碼剖析》第 2 章）

4

通用工具

Utilities

本章講解 C++ 標準程式庫中的通用工具。它們由短小精幹的類別和函式構成，執行最一般性的工作。這些工具包括：

- 數種通用型別（general types）
- 一些重要的 C 函式
- 數值極值[1]（numeric limits）

大部分通用工具在 C++ 標準規格書第 20 款（clause）描述，定義於標準表頭檔 `<utility>` 內。其餘工具則與標準程式庫中一些比較主要的組件一起描述，其原因可能是該類工具主要便是和那些組件共同使用，抑或因爲歷史因素。例如某些通用輔助函式被定義於 `<algorithm>` 表頭檔中，但按照 STL 的定義，它們不算是演算法（參見第 5 章）。

這些工具中的一部分也被運用於 C++ 標準程式庫中。特別是型別 `pair`，凡需要將兩個值視爲一個單元的場合（例如必須「回傳兩個值」的某函式），就必須用到它。

4.1 Pairs（對組）

class `pair` 可以將兩個值視爲一個單元。C++ 標準程式庫內多處使用了這個 class。尤其容器類別 `map` 和 `multimap`，就是使用 `pairs` 來管理其鍵值/實值（value/key）的成對元素（詳見 6.6 節，p194）。任何函式需回傳兩個值，也需要 `pair`。

[1] 可能有些人認爲數值極值應該屬於第 12 章，也就是專門講解數值的那一章，但這些數值極值在程式庫的其他部分也會被用到，所以我決定把它放在這裏。

Structure pair 定義於<utility>：

```
namespace std {
    template <class T1, class T2>
    struct pair {
        // type names for the values
        typedef T1 first_type;
        typedef T2 second_type;

        // member
        T1 first;
        T2 second;

        /* default constructor
        *-T1() and T2() force initialization for built-in types
        */
        pair()
          : first(T1()), second(T2()) {
        }

        // constructor for two values
        pair(const T1& a, const T2& b)
        : first(a), second(b) {
        }

        // copy constructor with implicit conversions
        template<class U, class V>
        pair(const pair<U,V>& p)
          : first(p.first), second(p.second) {
        }
    };

    // comparisons
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    ... // similar: !=, < =, > , > =
```

```
    // convenience function to create a pair
    template <class T1, class T2>
    pair<T1,T2> make_pair (const T1&, const T2&);
}
```

注意，pair 被定義爲 struct，而不是 class，這麼一來，所有成員都是 public，我們因此可以直接存取 pair 中的個別值。

上述 *default* 建構式生成一個 pair 時，以兩個「被該 *default* 建構式個別初始化」的值做爲初值。根據語言規則，基本型別（如 int）的 *default* 建構式也可以引起適當的初始化動作，所以：

```
std::pair<int,float> p; // initialize p.first and p.second with zero
```

就是以 int() 和 float() 來初始化 p。這兩個建構式都傳回零值。p14 曾經討論過基本型別的顯式初始化動作。

這裏之所以使用 template 形式的 *copy* 建構式，乃是因爲建構過程中可能需要隱式型別轉換。如果 pair 物件被複製，喚起的是由系統隱喻合成的那個 *copy* 建構式[2]。例如：

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo {
    std::pair<int,const char*> p(42,"hello");
    f(p);     // OK: calls built-in default copy constructor
    g(p);     // OK: calls template constructor
}
```

Pair 之間的比較

爲了比較兩個 pair 物件，C++ 標準程式庫提供了大家慣用的運算子。如果兩個 pair 物件內的所有元素都相等，這兩個 pair 物件就被視爲相等（equal）：

```
namespace std {
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>& x, const pair<T1,T2>& y) {
        return x.first == y.first && x.second == y.second;
    }
}
```

[2] template 形式的建構式並不會遮掩（由編譯器）隱喻合成的 *default* 建構式。詳見 p13。

兩個 pairs 互相比較時，第一元素具有較高的優先序。所以如果兩個 pairs 的第一元素不相等，其比較結果就成爲整個比較行爲的結果。如果第一元素相等，才繼續比較第二元素，並把比較結果當作整體比較結果。

```
namespace std {
  template <class T1, class T2>
  bool operator< (const pair<T1,T2>& x, const pair<T1,T2>& y) {
    return x.first < y.first ||
      (!(y.first < x.first) && x.second < y.second);
  }
}
```

其他的比較運算子（comparison operators）也如法炮製。

4.1.1 便捷函式 make_pair()

template 函式 make_pair() 使你無需寫出型別，就可以生成一個 pair 物件[3]：

```
namespace std {
    // create value pair only by providing the values
    template <class T1, class T2>
    pair<T1,T2> make_pair (const T1& x, const T2& y) {
        return pair<T1,T2>(x, y);
    }
}
```

例如，你可以這樣使用 make_pair()：

```
std::make_pair(42,'@')
```

而不必費力地這麼寫：

```
std::pair<int,char>(42,'@')
```

當我們有必要對一個接受 pair 引數的函式傳遞兩個值時，make_pair()尤其顯得方便，請看下例：

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo {
    f(std::make_pair(42,"hello")); // pass two values as pair
    g(std::make_pair(42,"hello")); // pass two values as pair
    // with type conversions
}
```

[3] 使用 make_pair 並不會多花你任何執行時間，編譯器應該會將此一動作最佳化。

從例子中可以看出，make_pair()使得「將兩個值當作一個 pair 引數來傳遞」的動作更容易。即使兩個值的型別並不準確符合要求，也能在 template 建構式提供的支援下順利工作。當你使用 map 和 multimap，你會經常用到這個特點（詳見 p203）。

注意，一個算式如果明白指出型別，便帶有一個優勢：產生出來的 pair 將有絕對明確的型別。例如：

```
std::pair<int,float>(42,7.77)
```

其結果與：

```
std::make_pair(42,7.77)
```

不同。後者所生成的 pair，第二元素的型別是 double（因爲「無任何飾詞的浮點字面常數」，其型別被視爲 double）。當我們使用重載函式（overloaded function）或 template，確切的型別非常重要。例如，爲了提高效率，程式員可能同時提供分別針對 float 和 double 的 function 或 template，這時候確切的型別就非常重要了。

4.1.2 Pair 運用實例

C++ 標準程式庫大量運用了 pair。例如 map 和 multimap 容器的元素型別便是 pair，也就是一組鍵值/實值（key/value）。關於 maps 和 multimaps 的一般性描述，詳見 p194, 6.6 節。p91 有一個 pair 型別的運用實例。C++ 標準程式庫中凡是「必須傳回兩個值」的函式，也都會利用 pair 物件（實例請見 p183）。

4.2 Class auto_ptr

本節描述 auto_ptr 型別。C++ 標準程式庫提供的 auto_ptr 是一種智慧型指標（smart pointer），幫助程式員防止「異常被擲出時發生資源洩漏」。注意我說的是「一種」智慧型指標，現實生活中還有其他許多有用的智慧型指標，auto_ptr 只是針對某個特定問題而設計，對於其他問題，auto_ptr 無能爲力。所以，請謹慎閱讀以下內容。

4.2.1 auto_ptr 的發展動機

函式的操作經常依以下模式進行[4]：

1. 獲取一些資源。
2. 執行一些動作。
3. 釋放所獲取的資源。

如果一開始獲取的資源，被繫結於區域物件（local objects）身上，當函式退出時，它們的解構式（destructor）被喚起，從而自動釋放這些資源。然而事情並不總是如此順利，如果資源是以顯式手法（explicitly）獲得，而且未被繫結於任何物件身上，那就必須以顯式手法釋放。這種情形常常發生在指標身上。

一個典型的例子就是運用 new 和 delete 來產生和銷毀物件:

```
void f()
{
      ClassA* ptr = new ClassA;  // create an object explicitly
      ...                        // perform some operations
      delete ptr;                // clean up (destroy the object explicitly)
}
```

也許你尚未意識到，這個函式其實是一系列麻煩的根源。一個顯而易見的問題是，我們經常忘掉 delete 動作，特別是當函式中間存在 return 述句時更是如此。然而真正的麻煩發生於更隱晦之處，那就是當異常發生時我們所要面對的災難。異常一旦出現，函式將立刻退離，根本不會呼叫函式尾端的 delete 述句。結果可能是記憶體遺失，或更一般地說是資源遺失。防止這種資源遺失的常見辦法就是捕捉所有異常，例如：

---

[4] class auto_ptr 的推動，是以 Scott Meyers 所著《*More Effective C++*》書中的相關資料爲基礎（並獲得他的允許）。這個問題的一般性技術最早描述於 Bjarne Stroustrup 的《*The C++ Programming Language*》2nd Edtion 和《*The Design and Evolution of C++*》，當時的主題是 "resource allocation is initialization"。auto_ptr 被加入 C++ 標準之中，正是爲了支援此一技術。

```
void f()
{
    ClassA* ptr = new ClassA;    // create an object explicitly

    try {
        ...              // perform some operations
    }
    catch (...) {        // for any exception
        delete ptr;      // - clean up
        throw;           // - rethrow the exception
    }

    delete ptr;          // clean up on normal end
}
```

你看，爲了在異常發生時處理物件的刪除工作，程式碼變得多麼複雜和累贅！如果還有第二個物件，如果還要比照辦理，如果還需要更多的 catch 子句，那簡直是一場惡夢。這不是優良的編程風格，複雜而且容易出錯，必須盡力避免。

如果使用智慧型指標，情形就會大不相同。這個智慧型指標應該保證，無論在何種情形下，只要自己被摧毀，就一定連帶釋放其所指資源。而由於智慧型指標本身就是區域變數，所以無論是正常退出，還是異常退出，只要函式退出，它就一定會被銷毀。auto_ptr 正是這種智慧型指標。

auto_ptr 是這樣一種指標：它是「它所指向的物件」的擁有者（owner）。所以，當身爲物件擁有者的 auto_ptr 被摧毀時，該物件也將遭到摧毀。auto_ptr 要求，一個物件只能有一個擁有者，嚴禁一物二主。

下面是上例改寫後的版本：

```
// header file for auto_ptr
#include <memory>

void f()
{
    // create and initialize an auto_ptr
    std::auto_ptr<ClassA> ptr(new ClassA);

    ...              // perform some operations
}
```

不再需要 delete，也不再需要 catch 了。auto_ptr 的介面與一般指標非常相似：operator* 用來提領其所指物件，operator-> 用來指向物件中的成員。然而，所有指標算術（包括 ++）都沒有定義（這可能是件好事，因爲指標算術是一大麻煩根源）。

注意，auto_ptr<> 不允許你使用一般指標慣用的賦值（assign）初始化方式。你必須直接使用數值來完成初始化[5]：

```
std::auto_ptr<ClassA> ptr1(new ClassA);       // OK
std::auto_ptr<ClassA> ptr2 = new ClassA;      // ERROR
```

4.2.2 auto_ptr 擁有權（Ownership）的轉移

auto_ptr 所界定的乃是一種嚴格的擁有權觀念。也就是說，由於一個 auto_ptr 會刪除其所指物件，所以這個物件絕對不能同時被其他物件「擁有」。絕對不應該出現多個 auto_ptrs 同時擁有一個物件的情況。不幸的是，這種事情可能會發生（如果你以同一個物件爲初值，將兩個 auto_ptrs 初始化，就會出現這種事）。程式員必須負責防範這種錯誤。

這個條件導致了一個問題：auto_ptr 的 *copy* 建構式和 *assignment* 運算子應當如何運作？此類操作往往是將此處資料拷貝到彼處。然而這種操作恰恰會導致上面所提的情形。解決辦法很簡單，但意義深遠：令 auto_ptr 的 *copy* 建構式和 *assignment* 運算子將物件擁有權交出去。試看下例 *copy* 建構式的運用：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);

// copy the auto_ptr
// - transfers ownership from ptr1 to ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

第一個述句中，ptr1 擁有了那個 new 出來的物件。第二個述句中，擁有權由 ptr1 轉交給 ptr2。此後 ptr2 就擁有了那個 new 出來的物件，而 ptr1 不再擁有它。這樣，物件就只會被 delete 一次 — 在 ptr2 被銷毀的時候。

---

[5] 下面兩種情況實際上是有分別的：

```
X x;
Y y(x);    // 顯式轉換（explicit conversion）
```

和：

```
X x;
Y y = x;   // 隱式轉換（implicit conversion）
```

前者使用顯式轉換，以型別 X 建構型別 Y 的一個新物件，後者使用隱式轉換。

賦值（指派, *assign*）動作也差不多：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
std::auto_ptr<ClassA> ptr2;     // create another auto_ptr
ptr2 = ptr1; // assign the auto_ptr
             // - transfers ownership from ptr1 to ptr2
```

在這裡，賦值（*assign*）動作將擁有權從 `ptr1` 轉移至 `ptr2`。於是，`ptr2` 擁有了先前被 `ptr1` 所擁有的那個物件。

如果 `ptr2` 被賦值之前正擁有另一個物件，賦值動作發生時會呼叫 `delete`，將該物件刪除：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
// initialize another auto_ptr with a new object
std::auto_ptr<ClassA> ptr2(new ClassA);

ptr2 = ptr1;          // assign the auto_ptr
                      // - delete object owned by ptr2
                      // - transfers ownership from ptr1 to ptr2
```

注意，擁有權的轉移，意味實值並非只是被簡單拷貝而已。只要發生了擁有權轉移，先前的擁有者（本例爲 `ptr1`）就失去了擁有權，結果，擁有者一旦交出擁有權，就兩手空空，只剩一個 null 指標在手了。在這裡，*copy* 建構式更動了「用以初始化新物件」的原物件，而賦值操作也修改了右側物件，這和程式語言中慣常的初始化動作和賦值動作可說大相徑庭。那麼誰來保證那個「失去了所有權、只剩一個 null 指標」的原 `auto_ptr` 不會再次進行提領動作呢？是的，還是程式員的責任。

只有 `auto_ptr` 可以拿來當做另一個 `auto_ptr` 的初值，普通指標是不行的：

```
std::auto_ptr<ClassA> ptr;              // create an auto_ptr

ptr = new ClassA;                       // ERROR
ptr = std::auto_ptr<ClassA>(new ClassA); // OK, delete old object
                                        // and own new
```

起點和終站（source and sink）

擁有權的移轉，使得 `auto_ptrs` 產生一種特殊用法：某個函式可以利用 `auto_ptr` 將擁有權轉交給另一個函式。這種事情可能在兩種情形下出現：

1. 某函式是資料的終站。如果 auto_ptr 以 *by value*（傳值）方式被當做一個引數傳遞給某函式，就是這種情況。此時被呼叫端的參數獲得了這個 auto_ptr 的擁有權，如果函式不再將它傳遞出去，它所指的物件就會在函式退出時被刪除：

```
void sink(std::auto_ptr<ClassA>);     // sink() gets ownership
```

2. 某函式是資料的起點。當一個 auto_ptr 被傳回，其擁有權便被轉交給呼叫端了。見下例：

```
std::auto_ptr<ClassA> f()
{
  std::auto_ptr<ClassA> ptr(new ClassA); // ptr owns the new object
  ...
  return ptr; // transfer ownership to calling function
}

void g()
{
  std::auto_ptr<ClassA> p;

  for (int i=0; i<10; ++i) {
    p = f(); // p gets ownership of the returned object
             // (previously returned object of f() gets deleted)
    ...
  }
}    // last-owned object of p gets deleted
```

每當 f() 被呼叫，它都 new 一個新物件，然後把該物件連同其擁有權一起傳回給呼叫端。將回返值指派（*assign*）給 p，同時也完成了擁有權的移轉。一旦迴圈再次執行這個指派動作，p 原先擁有的物件將被刪除。離開 g()時，p 也會被銷毀，這樣就刪除了 p 所擁有的最後一個物件。無論如何都不會有資源遺失之虞。即使有異常被擲出，擁有資料的 auto_ptr 也會盡職地將自己的資料刪除。

缺陷

auto_ptr 的語義本身就涵蓋擁有權，所以如果你無意轉交你的擁有權，就不要在參數列中使用 auto_ptr，也不要以它作爲回返值。下面例子是一個幼稚的作法，原本是想將 auto_ptr 所指物件列印出來，實際上卻引發一場災難：

```
// this is a bad example
template <class T>
void bad_print(std::auto_ptr<T> p) // p gets ownership of passed argument
{
```

```
    // does p own an object ?
    if (p.get() == NULL) {
        std::cout << "NULL";
    }
    else {
        std::cout << *p;
    }
}  // Oops, exiting deletes the object to which p refers
```

只要有一個 auto_ptr 被當做引數，放進這個 bad_print()函式，它所擁有的物件（如果有的話）就一定會被刪除。因爲作爲引數的 auto_ptr 會將擁有權轉交給參數 p，而當函式退出時，會刪除 p 所擁有的物件。這恐怕不是程式員所希望的，最終必然會引起致命的執行期錯誤：

```
std::auto_ptr<int> p(new int);
*p = 42;          // change value to which p refers
bad_print(p);     // Oops, deletes the memory to which p refers
*p = 18;          // RUNTIME ERROR
```

你可能會認爲，將 auto_ptrs 以 *pass by reference* 方式傳遞就萬事大吉。然而這種行爲卻會使「擁有權」的概念變得難以捉摸，因爲面對一個「透過 reference 而獲得 auto_ptr」的函式，你根本無法預知擁有權是否被轉交。所以以 *by reference* 方式傳遞 auto_ptr 是非常糟糕的設計，應該全力避免。

考慮到 auto_ptr 的概念，我們倒是可以運用 constant reference，向函式傳遞擁有權。然而這十分危險，因爲當你傳遞一個 constant reference 時，通常預期該物件不會被更動。幸好 auto_ptrs 的一個晚期設計降低了此一危險性。藉由某些實作技巧，我們可以令 constant reference 無法交出擁有權。事實上，你無法變更任何 constant reference 的擁有權：

```
const std::auto_ptr<int> p(new int);
*p = 42;          // change value to which p refers
bad_print(p);     // COMPILE-TIME ERROR
*p = 18;          // OK
```

這一方案使得 auto_ptrs 比以前顯得更安全一些。很多介面在需要內部拷貝時，都藉由 constant reference 獲得原值。事實上，C++ 標準程式庫的所有容器（例見第 6 章或第 10 章）都如此，大致像這樣：

```
template <class T>
void container::insert (const T& value)
{
    ...
    x = value; // assign or copy value internally
    ...
}
```

如果這一類賦值動作對 auto_ptr 有效，那麼擁有權就會被轉交給容器。然而正由於 auto_ptrs 的實際設計，這種行爲必然會導致編譯錯誤：

```
container<std::auto_ptr<int> > c;
const std::auto_ptr<int> p(new int);
...
c.insert(p); // ERROR
...
```

總而言之，常數型 auto_ptr 減小了「不經意轉移擁有權」所帶來的危險。只要一個物件藉由 auto_ptr 傳遞，就可以使用常數型 auto_ptr 來終結擁有權移轉鏈，此後擁有權將不能再進行移轉。

在這裡，關鍵字 const 並非意味你不能更改 auto_ptr 所擁有的物件，而是意味你不能更改 auto_ptr 的擁有權。例如：

```
std::auto_ptr<int> f()
{
  const std::auto_ptr<int> p(new int); // no ownership transfer possible
  std::auto_ptr<int> q(new int); // ownership transfer possible

  *p = 42;        // OK, change value to which p refers
  bad_print(p);   // COMPILE-TIME ERROR
  *p = *q;        // OK, change value to which p refers
  p = q;          // COMPILE-TIME ERROR
  return p;       // COMPILE-TIME ERROR
}
```

如果使用 const auto_ptr 作爲引數，對新物件的任何賦值（*assign*）動作都將導致編譯期錯誤。就常數特性而言，const auto_ptr 比較類似常數指標（T* const p），而非指向常數的指標（const T* p）— 儘管其語法看上去比較像後者。

4.2.3 auto_ptrs 作 [illegible]

在 class 中使用 auto_ptrs，你可以因而避免遺失資源。如果你以 auto_ptr 而非一般指標作爲成員，當物件被刪除時，auto_ptr 會自動刪除其所指的成員物件，於是你也就不再需要解構式了。此外，即使在初始化期間丟擲異常，auto_ptr 也可以幫助避免資源遺失。注意，只有當物件被完整建構成功，才有可能於將來呼叫其解構式。這造成了資源遺失的隱憂：如果第一個 new 成功了，第二個 new 卻失敗了，就會造成資源遺失。例如：

```
class ClassB {
  private:
    ClassA* ptr1; // pointer members
    ClassA* ptr2;
  public:
    // constructor that initializes the pointers
    // - will cause resource leak if second new throws
    ClassB (ClassA val1, ClassA val2)
      : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - might cause resource leak if second new throws
    ClassB (const ClassB& x)
      : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
      *ptr1 = *x.ptr1;
      *ptr2 = *x.ptr2;
      return *this;
    }

    ~ClassB () {
      delete ptr1;
      delete ptr2;
    }
    ...
};
```

使用 auto_ptr，你就可以輕鬆避免這場悲劇，：

```
class ClassB {
  private:
    const std::auto_ptr<ClassA> ptr1; // auto_ptr members
    const std::auto_ptr<ClassA> ptr2;
  public:
    // constructor that initializes the auto_ptrs
    // - no resource leak possible
```

```
    ClassB (ClassA val1, ClassA val2)
      : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - no resource leak possible
    ClassB (const ClassB& x)
      : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
      *ptr1 = *x.ptr1;
      *ptr2 = *x.ptr2;
      return *this;
    }

    // no destructor necessary
    // (default destructor lets ptr1 and ptr2 delete their objects)
    ...
};
```

然而請注意，儘管你可以略過解構式，卻還是不得不親自撰寫 *copy* 建構式和 *assignment* 運算子。預設狀況下，這兩個操作都會轉交擁有權，這恐怕並非你所願。正如 p42 所說，爲了避免擁有權的意外轉交，如果你的 auto_ptr 在整個生命期內都不必改變其所指物件的擁有權，你可以使用 const auto_ptrs。

4.2.4 auto_ptrs 的錯誤運用

auto_ptr 確實解決了一個特定問題，那就是在異常處理過程中的資源遺失問題。不幸的是由於 auto_ptr 的具體行爲方式曾經三番五次地改動，而且 C++ 標準程式庫中只此一個智慧型指標（smart pointer），別無分號，所以人們總是會誤用 auto_ptrs。爲了幫助你正確使用它，這裏給出一些要點：

1. auto_ptrs 之間不能共享擁有權

   一個 auto_ptr 千萬不能指向另一個 auto_ptr（或其他物件）所擁有的物件。否則，當第一個指標刪除該物件後，另一個指標突然間指向了一個已被銷毀的物件，那麼，如果再透過那個指標進行讀寫操作，就會引發一場災難。

2. 並不存在針對 array 而設計的 auto_ptrs

   auto_ptr 不可以指向 array，因爲 auto_ptr 是透過 delete 而非 delete[] 來釋放其所擁有的物件。注意，C++ 標準程式庫並未提供針對 array 而設計的 auto_ptr。標準程式庫另提供了數個容器類別，用來管理資料群（參見第 5 章）。

3. auto_ptrs 決非一個「四海通用」的智慧型指標

   並非任何適用智慧型指標的地方，都適用 auto_ptr。特別請注意的是，它不是參用計數（*reference counting*）型指標 — 這種指標保證，如果有一組智慧型指標指向同一個物件，那麼若且唯若（if and only if）最後一個智慧型指標被銷毀時，該物件才會被銷毀。

4. auto_ptrs 不滿足 STL 容器對其元素的要求

   auto_ptr 並不滿足 STL 標準容器對於元素的最基本要求，因爲在拷貝（*copy*）和賦值（*assign*）動作之後，原本的 auto_ptr 和新產生的 auto_ptr 並不相等。是的，拷貝和賦值之後，原本的 auto_ptr 會交出擁有權，而不是拷貝給新的 auto_ptr。因此請絕對不要將 auto_ptr 作爲標準容器的元素。幸好語言和程式庫的設計本身就可以防止這種誤用，如果你的工作環境符合標準，這類誤用應該無法通過編譯。

不幸的是，某些時候，即使誤用 auto_ptr，程式仍然能夠順利運作。就此點而言，使用一個非常數（nonconstant）auto_ptr，並不比使用一個一般指標更安全。如果你的誤用行爲沒有導致全盤崩潰，你或許會暗自慶幸，而這其實是真正的不幸，因爲你或許根本就沒有意識到你已經犯了錯誤。關於「參用計數型（reference counting）智慧指標」的討論請見 p135, 5.10.2 節，p222, 6.8 節有一份實作碼。當我們有必要在不同容器之間共用元素時，這種指標非常有用。

4.2.5 auto_ptr 運用實例

下面第一個例子展示 auto_ptrs 移轉擁有權的行爲：

```
// util/autoptr1.cpp
#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
* - print object value or NULL
*/
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
   // does p own an object ?
   if (p.get() == NULL) {
```

```
        strm << "NULL";      // NO: print NULL
    }
    else {
        strm << *p;          // YES: print the object
    }
    return strm;
}

int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    q = p;
    cout << "after assigning auto pointers:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    *q += 13;      // change value of the object q owns
    p = q;
    cout << "after change and reassignment:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
}
```

程式輸出如下：

```
after initialization:
  p: 42
  q: NULL
after assigning auto pointers:
  p: NULL
  q: 42
after change and reassignment:
  p: 55
  q: NULL
```

注意，output 運算子的第二個參數是一個 const reference，所以並沒有發生擁有權的移轉。

正如我在 p40 所說，請時刻銘記於心，你不能以一般指標的賦值手法來初始化一個 auto_ptr：

```
std::auto_ptr<int> p(new int(42));       // OK
std::auto_ptr<int> p = new int(42);      // ERROR

p = std::auto_ptr<int>(new int(42));     // OK
p = new int(42);                         // ERROR
```

這是因爲，「根據一般指標生成一個 auto_ptr」的那個建構式，被宣告爲 explicit（關於 explicit，詳見 p18, 2.2.6 節）。

下面這個例子展示 const auto_ptr 的特性：

```
// util/autoptr2.cpp

#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
* - print object value or NULL
*/
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
  // does p own an object ?
  if (p.get() == NULL) {
    strm << "NULL";    // NO: print NULL
  }
  else {
    strm << *p;        // YES: print the object
  }
  return strm;
}

int main()
{
  const auto_ptr<int> p(new int(42));
  const auto_ptr<int> q(new int(0));
  const auto_ptr<int> r;
```

```
    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

    *q = *p;
// *r = *p;    // ERROR: undefined behavior
    *p = -77;
    cout << "after assigning values:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

// q = p;      // ERROR at compile time
// r = p;      // ERROR at compile time
}
```

程式輸出如下：

```
after initialization:
  p: 42
  q: 0
  r: NULL
after assigning values:
  p: -77
  q: 42
  r: NULL
```

這個例子爲 auto_ptrs 定義了一個 output 運算子，其中將 auto_ptr 以 const reference 的方式傳遞。根據 p43 的討論，你不應該以任何形式傳遞 auto_ptr，但此處是個例外。

注意下列賦值（指派）動作是錯誤的：

```
*r = *p;
```

這個句子對於一個「未指向任何物件」的 auto_ptr 進行提領（*dereference*）動作。C++ 標準規格宣稱，這會導致未定義行爲，比如說導致程式的崩潰。從這個例子可以看出，你可以操作 const auto_ptrs 所指物件本身，但「它所擁有的究竟是哪個物件」這一事實無法改變。就算 r 不具常數性，最後一個述句也會失敗，因爲 p 具有常數性，其擁有權不得被更改。

4.2.6 auto_ptr 實作細目

class auto_ptr 宣告於 <memory>：

```
#include <memory>
```

auto_ptr 定義於 namespace std 中，是「可用於任何型別身上」的一個 template class。下面是 auto_ptr 的確切宣告[6]：

```
namespace std {
  // auxiliary type to enable copies and assignments
  template <class Y> struct auto_ptr_ref {};

  template<class T>
  class auto_ptr {
    public:
      // type names for the value
      typedef T element_type;

      // constructor
      explicit auto_ptr(T* ptr = 0) throw();

      // copy constructors (with implicit conversion)
      // - note: nonconstant parameter
      auto_ptr(auto_ptr&) throw();
      template<class U> auto_ptr(auto_ptr<U>&) throw();

      // assignments (with implicit conversion)
      // - note: nonconstant parameter
      auto_ptr& operator= (auto_ptr&) throw();
      template<class U>
        auto_ptr& operator= (auto_ptr<U>&) throw();

      // destructor
      ~auto_ptr() throw();

      // value access
      T* get() const throw();
      T& operator*() const throw();
      T* operator->() const throw();
```

[6] 這裡所給的版本較之 C++ 標準程式庫的版本作了一些小小改進，修正了幾個小問題（這裡的 auto_ptr_ref 是全域的，這裡並且設定了從 auto_ptr_ref 到 auto_ptr 的 *assignment* 運算子，參見 p55）。

```
    // release ownership
    T* release() throw();

    // reset value
    void reset(T* ptr = 0) throw();

  // special conversions to enable copies and assignments
  public:
    auto_ptr(auto_ptr_ref<T>) throw();
    auto_ptr& operator= (auto_ptr_ref<T> rhs) throw();
    template<class U> operator auto_ptr_ref<U>() throw();
    template<class U> operator auto_ptr<U>() throw();
 };
}
```

個別成員的詳細描述將在後續數節中進行。討論過程中我把 auto_ptr<T> 簡寫爲 auto_ptr。p56 有一份完整的 auto_ptr 實作範例。

型別定義

auto_ptr::**element_type**

- auto_ptr 所擁有之物件的型別

建構式（constructor）、賦值運算子（assign operator）、解構式（destructor）

auto_ptr::**auto_ptr()** throw()

- *default* 建構式。
- 生成一個不擁有任何物件的 auto_ptr。
- 將 auto_ptr 的值初始化爲零。

explicit *auto_ptr*::**auto_ptr**(T* *ptr*) throw()

- 生成一個 auto_ptr，並擁有 ptr 所指物件。
- 此一動作完成後，*this 成爲 ptr 所指物件的唯一擁有者。不允許再有其他擁有者。

- 如果 ptr 本身不是 null 指標，那就必須是個 new 回返值，因爲 auto_ptr 解構式會對其所擁有的物件自動呼叫 delete。
- 不能用 new[] 所生成的 array 作爲初值。當你需要 array，請考慮使用 STL 容器，p75, 5.2 節對此有些介紹。

```
auto_ptr::auto_ptr(auto_ptr& ap) throw()
template<class U> auto_ptr::auto_ptr(auto_ptr<U>& ap) throw()
```

- 針對 non-const values 而設計的一個 *copy* 建構式。
- 生成一個 auto_ptr，在入口處將 ap 所擁有的物件（如果有的話）的擁有權奪取過來。
- 此操作完畢之後，ap 不再擁有任何物件，其值變爲 null 指標。所以，和一般 *copy* 建構式不同，這個操作改變了原物件。
- 注意，此函式有一個多載化的 member template（請參考 p11），使得 ap 可藉由型別自動轉換，構造出合適的 auto_ptr。例如，根據一個「衍生類別的物件」，建構出一個基礎類別物件的 auto_ptr。
- 擁有權移轉問題，請參考 p40, 4.2.2 節。

```
auto_ptr& auto_ptr::operator= (auto_ptr& ap) throw()
template<class U> auto_ptr& auto_ptr::operator= (auto_ptr<U>& ap) throw()
```

- 針對 non-const value 而設計的一個 *assignment*（賦值）運算子
- 如果自身原本擁有物件，進入本動作時將被刪除，然後獲得 ap 所擁有的物件。於是，原本 ap 所擁有的物件，其擁有權就移轉給了 *this。
- 此一動作完成後，ap 不再擁有任何物件。其值變爲 null 指標。與一般賦值動作不同，此處這個動作改變了原物件。
- 左手邊的 auto_ptr 原本所指物件將被刪除（deleted）。
- 注意，此函式有一個多載化的 member template（請參考 p11）。這使得 ap 可藉由「型別自動轉換」指派給合適的 auto_ptr。例如，將一個「衍生類別的物件」，指派給一個基礎類別物件的 auto_ptr。
- 擁有權移轉問題，請參考 p40, 4.2.2 節。

```
auto_ptr::~auto_ptr() throw()
```

- 解構式
- 如果 auto_ptr 擁有某個物件，此處將呼叫 delete 刪除之。

數值存取（value access）

```
T* auto_ptr::get() const throw()
```

- 傳回 auto_ptr 所指物件的位址。
- 如果 auto_ptr 未指向任何物件，傳回 null 指標。
- 這個動作並不改變擁有權。退出此函式時，auto_ptr 仍然保有對物件（如果有的話）的擁有權。

```
T& auto_ptr::operator*() const throw()
```

- *dereference*（提領）運算子
- 傳回 auto_ptr 所擁有的物件。
- 如果 auto_ptr 並未擁有任何物件，此呼叫導致未定義行爲（可能導致崩潰）。

```
T* auto_ptr::operator->() const throw()
```

- *member access*（成員存取）運算子
- 傳回 auto_ptr 所擁有的物件中的一個成員。
- 如果 auto_ptr 並未擁有任何物件，此呼叫將導致未定義行爲（可能導致崩潰）。

數值操作

```
T* auto_ptr::release() throw()
```

- 放棄 auto_ptr 原先所擁有之物件的擁有權。
- 傳回 auto_ptr 原先擁有物件（如果有的話）的位址。
- 如果 auto_ptr 原先並未擁有任何物件，傳回 null 指標。

```
void auto_ptr::reset(T* ptr = 0) throw()
```

- 以 ptr 重新初始化 auto_ptr。
- 如果 auto_ptr 原本擁有物件，則此動作開始前先刪除之。
- 呼叫結束後，*this 成爲 ptr 所指物件的擁有者。注意，不應該有任何其他擁有者。
- 如果 ptr 不是 null 指標，應當是一個由 new 傳回的值，因爲 auto_ptr 的解構式會呼叫 delete 來刪除其所擁有的物件。
- 注意，不得將透過 new[] 生成的 array 當作引數傳進來。如果需要使用 array，請考慮使用 STL 容器類別，詳見 p75, 5.2 節。

Conversions（轉型操作）

auto_ptr 中剩餘的內容（輔助型別 auto_ptr_ref 及其相關函式）涉及非常精緻的技巧，使我們得以拷貝和指派 non-const auto_ptrs，卻不能拷貝和指派 const auto_ptrs（詳見 p44）。下面是一份扼要解釋[7]。我們有兩個需求：

1. 我們需要將 auto_ptr 作爲右值（rvalue）傳遞到函式去，或由函式中傳回[8]。由於 auto_ptr 是個類別，所以這些工作應當由建構式完成。
2. 拷貝 auto_ptr 時，原指標務必放棄擁有權。這就要求拷貝動作必須修改原本的那個 auto_ptr。

一般的 *copy* 建構式當然可以拷貝右值，但爲了做到這點，它必須將其參數型別宣告爲一個 *reference to const object*。如果在 auto_ptr 中使用一般的 *copy* 建構式，我們恐怕不得不將 auto_ptr 內含的實際指標宣告爲 mutable，只有這樣，才能在 *copy* 建構式中更改它。你以爲萬事大吉了嗎？錯，這種做法將允許用戶拷貝那些宣告爲 const 的物件，將其擁有權轉交他人，這與其原本的常數性背道而馳。

變通作法是找出一種機制，能夠將右值轉化爲左值。「直接轉型爲 reference」的那種簡單的轉型運算子派不上用場，因爲當你實際上是把一個物件轉化爲自己原本的型別時，不會有任何轉型操作被喚起（切記，reference 屬性並非型別的一部份）。爲此才有了 auto_ptr_ref 類別的引進，協助我們將右值轉化爲左值。這一機制的理論基礎是「多載化」和「template 引數推導規則」之間一個細微的不同處。這個差別實在太細微了，不大可能成爲一般程式編寫技巧而用於別處，但卻足以在這裡讓 auto_ptr 正確運作。這就夠了。

如果你的編譯器對於 non-const 和 const auto_ptrs 之間的區別尚不能做出很好的闡釋，請不必驚訝。但是請你保持清醒的頭腦，如果你的編譯器尚未達到這一水平，那麼 auto_ptr 的使用就會變得更加危險。因爲這種情況下很容易意外地將擁有權旁落他人之手。

---

[7] 感謝 Bill Gibbons 指出這一點。

[8] rvalue（右值）和 lvalue（左值）的名稱由來，是從賦值運算 expr1 = expr2 得來。在這種運算式中，左運算元 expr1 必須是一個（可更改的）lvalue。不過或許更貼切的描述是：lvalue 代表 *l*ocator *value*。也就是說，這個算式藉由名字和參考值（pointer 或 reference）來指定一個物件。lvalue 並非一定「可被更改」。例如常數物件的名字就是一個不可被改動的 lvalue。所有 non-lvalues 物件，都是 rvalues。尤其顯式生成（T()）的暫時物件和函式回返值，都是 rvalue。

類別 auto_ptr 的實作範例

以下源碼展示了一個符合標準的 auto_ptr 類別的實作示範[9]：

```
// util/autoptr.hpp

/* class auto_ptr
* - improved standard conforming implementation
*/
namespace std {
  // auxiliary type to enable copies and assignments (now global)
  template<class Y>
  struct auto_ptr_ref {
    Y* yp;
    auto_ptr_ref (Y* rhs)
      : yp(rhs) {
    }
  };

  template<class T>
  class auto_ptr {
    private:
      T* ap; // refers to the actual owned object (if any)
    public:
      typedef T element_type;

      // constructor
      explicit auto_ptr (T* ptr = 0) throw()
        : ap(ptr) {
      }

      // copy constructors (with implicit conversion)
      // - note: nonconstant parameter
      auto_ptr (auto_ptr& rhs) throw()
        : ap(rhs.release()) {
      }
```

[9] 感謝 Greg Colvin 提供的這份 auto_ptr 實作內容。注意，這個實作版本並不完全契合 C++ 標準規範。事實證明，C++ 標準所規定的形式中，當利用 auto_ptr_ref 進行轉型時，在某種特殊情況下仍會出現小小瑕疵。這裡所給的方案很有可能徹底解決所有問題。不過，撰寫本書的時候，仍有一些相關討論正在進行。

```
template<class Y>
auto_ptr (auto_ptr<Y>& rhs) throw()
  : ap(rhs.release()) {
}

// assignments (with implicit conversion)
// - note: nonconstant parameter
auto_ptr& operator= (auto_ptr& rhs) throw() {
  reset(rhs.release());
  return *this;
}
template<class Y>
auto_ptr& operator= (auto_ptr<Y>& rhs) throw() {
  reset(rhs.release());
  return *this;
}

// destructor
~auto_ptr() throw() {
  delete ap;
}

// value access
T* get() const throw() {
  return ap;
}
T& operator*() const throw() {
  return *ap;
}
T* operator->() const throw() {
  return ap;
}

// release ownership
T* release() throw() {
  T* tmp(ap);
  ap = 0;
  return tmp;
}
```

```
    // reset value
    void reset (T* ptr=0) throw() {
      if (ap != ptr) {
        delete ap;
        ap = ptr;
      }
    }

    /* special conversions with auxiliary type to enable copies and
    assignments
    */
    auto_ptr(auto_ptr_ref<T> rhs) throw()
      : ap(rhs.yp) {
    }
    auto_ptr& operator= (auto_ptr_ref<T> rhs) throw() { // new
      reset(rhs.yp);
      return *this;
    }
    template<class Y>
    operator auto_ptr_ref<Y>() throw() {
      return auto_ptr_ref<Y>(release());
    }
    template<class Y>
    operator auto_ptr<Y>() throw() {
      return auto_ptr<Y>(release());
    }
  };
}
```

4.3 數值極限（Numeric Limits）

一般說來，數值型別（Numeric types）的極值是一個與平台相依的特性。C++ 標準程式庫藉由 template `numeric_limits` 提供這些極值，取代傳統 C 語言所採用的預處理器常數（preprocessor constants）。你仍然可以使用後者，其中整數常數定義於 `<climits>` 和 `<limits.h>`，浮點常數定義於 `<cfloat>` 和 `<float.h>`。新的極值概念有兩個優點，第一是提供了更好的型別安全性，第二是程式員可藉此寫出一些 templates 以核定（*evaluate*）這些極值。

本節的剩餘部份專門討論極值問題。注意，*C++ Standard* 規定了各種型別必須保證的最小精度，如果你能夠注意並運用這些極值，就比較容易寫出與平台無關的程式。這些最小值列於表 4.1。

型別	最小長度
`char`	1 byte (8 bits)
`short int`	2 bytes
`int`	2 bytes
`long int`	4 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes

表 4.1 內建型別的最小長度

Class `numeric_limits<>`

使用 template，通常是爲了對所有型別一次性地撰寫出一個通用解決方案。除此之外，你還可以在必要時候以 template 爲每個型別提供共同介面。方法是：不但提供通用性的 template，還提供其特化（specialization）版本。`numeric_limits` 就是這項技術的一個典型例子，作法如下：

- 通用性的 template，爲所有型別提供預設極值：

```
namespace std {
    /* general numeric limits as default for any type
    */
    template <class T>
    class numeric_limits {
       public:
       // no specialization for numeric limits exist
       static const bool is_specialized = false;
       ... // other members that are meaningless for the general numeric limits
    };
}
```

這個通用性 template 將成員 `is_specialized` 設爲 `false`，意思是，對型別 `T` 而言，無所謂極值的存在。

- 各具體型別的極值，由特化版本（specialization）提供：

```
namespace std {
    /* numeric limits for int
    * - implementation defined
    */
    template<> class numeric_limits<int> {
    public:
        // yes, a specialization for numeric limits of int does exist
        static const bool is_specialized = true;

        static T min() throw() {
            return -2147483648;
        }
        static T max() throw() {
            return 2147483647;
        }
        static const int digits = 31;
        ...
    };
}
```

這裡把 `is_specialized` 設爲 `true`，所有其他成員都根據特定型別的具體極值加以設定。

通用性的 `numeric_limits` template，及其特化版本都被放在`<limits>`表頭檔中。*C++ Standard* 所囊括的特化版本，涵蓋了所有數值基本型別，包括：`bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`。你可以輕易爲你自定的數值型別加上補充。

表 4.2 和表 4.3 列出 class `numeric_limits<>` 的所有成員及其意義。最右一列顯示對應的 C 常數，它們分別定義於 `<climits>`, `<limits.h>`, `<cfloat>`, `<float.h>` 內。

成員	意義	對應的 C 常數
`is_specialized`	型別是否有極值	
`is_signed`	型別帶有正負號	
`is_integer`	整數型別	
`is_exact`	計算結果不產生捨/入誤差（此成員對所有整數型別而言均爲 true）	
`is_bounded`	數值集的個數有限（對所有內建型別而言，此成員均爲 true）	
`is_modulo`	兩正值相加，其結果可能因溢位而回繞爲較小的值。	
`is_iec559`	遵從 IEC 559 及 IEEE 754 標準	
`min()`	最小值（對浮點數而言，是標準化後的值；只有當 `is_bounded || !is_signed` 成立時才有意義)	`INT_MIN`, `FLT_MIN`, `CHAR_MIN`, ...
`max()`	最大值（只有當 `is_bounded` 成立時才有意義）	`INT_MAX`, `FLT_MAX`, ...
`digits`	字元和整數：不帶正負號之位元個數 浮點數：尾數中之 `radix`（見下）位元個數。	`CHAR_BIT` `FLT_MANT_DIG`, ...
`digits10`	十進位數的個數（只有當 `is_bounded` 成立時才有意義）	`FLT_DIG`, ...
`radix`	整數：表示式的基底（base），幾乎總是 2。 浮點數：指數表示式的基底（base）	 `FLT_RADIX`
`min_exponent`	基底 `radix` 的最小負整數指數	`FLT_MIN_EXP`, ...
`max_exponent`	基底 `radix` 的最大正整數指數	`FLT_MAX_EXP`, ...
`min_exponent10`	基底 10 的最小負整數指數	`FLT_MIN_10_EXP`, ...
`max_exponent10`	基底 10 的最大正整數指數	`FLT_MAX_10_EXP`, ...
`epsilon()`	1 和最接近 1 的值之間的差距	`FLT_EPSILON`, ...
`round_style`	捨/入（rounding）風格（見 p63）	
`round_error()`	最大捨/入誤差量測（根據 ISO/IEC 10967-1 標準）	
`has_infinity`	有「正無窮大」表示式	
`infinity()`	表現出「正無窮大」（如有的話）	
`has_quiet_NaN`	本型別有不發訊號（nonsignaling）的「非數值」表述式。	
`quiet_NaN()`	如果可以，安靜地（nonsignaling）表述出「這不是個數值」。	
`has_signaling_NaN`	本型別有會發出訊號（signaling）的「非數值」表述式。	
`signaling_NaN()`	如果可以，發出訊號地（signaling）表述出「這不是個數值」。	
`has_denorm`	本型別是否允許非標準化數值（denormalized values，也就是 variable members of exponent bits, 見 p63）	

`has_denorm_loss`	準確度的遺失是以一個非標準化值（denormalization）而不是以一個不精密的結果被偵測出來。
`denorm_min()`	最小的非標準化（denormalized）正值。
`traps`	已實作出 Trapping
`tinyness_before`	在捨/入（rounding）之前可偵測出 tinyness

表 4.2. class `numeric_limits<>` 的所有成員

譯註 1：本表格「意義」欄中的諸多解釋，涉及數位表示法專業術語。譯者這方面能力有限，沒把握正確譯出。對於沒有把握的名詞，皆保留英文，望諒。

譯註 2：本表格在英文版中因分頁關係而被切割爲表 4.2 和表 4.3。此處合而爲一。後續表格從表 4.4 開始繼續編號。

下面是對於 `float` 型別的數值限定模板特殊化的一個完全實作版，當然是和平台相依的。這裏同時還給出了各成員的確切標記（signatures）：

```
namespace std {
  template<> class numeric_limits<float> {
    public:
      // yes, a specialization for numeric limits of float does exist
      static const bool is_specialized = true;

      inline static float min() throw() {
        return 1.17549435E-38F;
      }
      inline static float max() throw() {
        return 3.40282347E+38F;
      }

      static const int digits = 24;
      static const int digits10 = 6;

      static const bool is_signed = true;
      static const bool is_integer = false;
      static const bool is_exact = false;
      static const bool is_bounded = true;
      static const bool is_modulo = false;
      static const bool is_iec559 = true;

      static const int radix = 2;

      inline static float epsilon() throw() {
```

```
      return 1.19209290E-07F;
    }

    static const float_round_style round_style
      = round_to_nearest;
    inline static float round_error() throw() {
      return 0.5F;
    }

    static const int min_exponent = -125;
    static const int max_exponent = +128;
    static const int min_exponent10 = -37;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    inline static float infinity() throw() { return ...; }
    static const bool has_quiet_NaN = true;
    inline static float quiet_NaN() throw() { return ...; }
    static const bool has_signaling_NaN = true;
    inline static float signaling_NaN() throw() { return ...; }
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    inline static float denorm_min() throw() { return min(); }

    static const bool traps = true;
    static const bool tinyness_before = true;
  };
}
```

注意，所有資料成員如果不是 const，便是 static，這麼一來其值便可在編譯期間確定。至於由函式所定義的成員，在某些編譯器中恐怕無法在編譯期間確定其值。因此，同一份目的碼（object code）雖然可以在不同的處理器上執行，可能會得出不同的浮點值。

round_style 的值列於表 4.4，has_denorm 的值列於表 4.5。has_denorm 其實也許應該稱爲 denorm_style 更貼切，可惜並非如此。這是因爲 C++ 標準化後期才決定將其原本的 bool 型別改變爲列舉值（enumerative value）之故。不過你還是可以把 has_denorm 當成 bool 值來用；*C++ Standard* 保證，如果 denorm_absent 爲 0，就等於 false，如果 denorm_present 爲 1 而且 denorm_indeterminate 爲 -1，那麼兩者都等於 true。因此你可以把 has_denorm 視爲一個 bool 值，用以判定某個型別是否允許所謂的 "denormalized values"。

捨/入（round）風格	意義
round_toward_zero	向零捨/入
round_to_nearest	向最接近的可表示值捨/入
round_toward_infinity	向正無限值捨/入
round_toward_neg_infinity	向負無限值捨/入
round_indeterminate	無法確定

表 4.4 numeric_limits<> 的捨入（round）風格

捨入風格	意義
denorm_absent	此型別不允許 "denormalized values"
denorm_present	此型別允許向最接近的可表示值做 denormalized values
denorm_indeterminate	無法確定

表 4.5 numeric_limits<> 的 "denormalization style"

numeric_limits<> 使用範例

下面的例子展示某些型別極值的可能運用，例如用來瞭解某個型別的最大值，或確定 char 是否帶正負號：

```
// util/limits1.cpp

#include <iostream>
#include <limits>
#include <string>
using namespace std;
int main()
{
  // use textual representation for bool
  cout << boolalpha;

  // print maximum of integral types
  cout << "max(short): " << numeric_limits<short>::max() << endl;
  cout << "max(int): " << numeric_limits<int>::max() << endl;
  cout << "max(long): " << numeric_limits<long>::max() << endl;
  cout << endl;
```

```
    // print maximum of floating-point types
    cout << "max(float): "
         << numeric_limits<float>::max() << endl;
    cout << "max(double): "
         << numeric_limits<double>::max() << endl;
    cout << "max(long double): "
         << numeric_limits<long double>::max() << endl;
    cout << endl;

    // print whether char is signed
    cout << "is_signed(char): "
         << numeric_limits<char>::is_signed << endl;
    cout << endl;

    // print whether numeric limits for type string exist
    cout << "is_specialized(string): "
         << numeric_limits<string>::is_specialized << endl;
}
```

程式的輸出結果和執行平台有關，下面是其中一種可能：

```
max(short): 32767
max(int): 2147483647
max(long): 2147483647

max(float): 3.40282e+38
max(double): 1.79769e+308
max(long double): 1.79769e+308

is_signed(char): false

is_specialized(string): false
```

最後一行表示，型別 `string` 並沒有定義數值極限。這是理所當然的，因爲 `strings` 並非數值型別。正如本例所示，你可以對任何型別進行詢問，無論它是否定義了極值。

4.4 輔助函式

演算法程式庫（定義於表頭檔 <algorithm>）內含三個輔助函式，一個用來在兩值之中挑選較大者，另一個用來在兩值之中挑選較小者，第三個用來交換兩值。

4.4.1 挑選較小值和較大值

「在兩物之間選擇較大值和較小值」的函式，定義於 <algorithm>，如下所示：

```
namespace std {
  template <class T>
  inline const T& min (const T& a, const T& b) {
    return b < a ? b : a;
  }

  template <class T>
  inline const T& max (const T& a, const T& b) {
    return a < b ? b : a;
  }
}
```

如果兩值相等，通常會傳回第一值。不過你的程式最好不要依賴這一點。

上述兩個函式還有另一個版本，接受一個額外的 template 引數作爲「比較準則」：

```
namespace std {
  template <class T, class Compare>
  inline const T& min (const T& a, const T& b, Compare comp) {
    return comp(b,a) ? b : a;
  }

  template <class T, class Compare>
  inline const T& max (const T& a, const T& b, Compare comp) {
    return comp(a,b) ? b : a;
  }
}
```

作爲「比較準則」的那個引數應該是個函式或仿函式（functor，將於 5.9 節, p124 介紹），接受兩個引數並進行比較：在某個指定規則下，判斷第一引數是否小於第二引數，並傳回判斷結果。

下面這個例子示範如何傳入特定的比較函式作爲引數，以此方式來運用 max()：

```
// util/minmax1.cpp

#include <algorithm>
using namespace std;

/* function that compares two pointers by comparing the values to which
they point
*/
bool int_ptr_less (int* a, int* b)
{
  return *a < *b;
}

int main()
{
  int x = 17;
  int y = 42;
  int* px = &x;
  int* py = &y;
  int* pmax;

  // call max() with special comparison function
  pmax = max (px, py, int_ptr_less);
  ...
}
```

注意，min()和 max()都要求它們所接受的兩個引數的型別必須一致。如果不一致，你將無法正確呼叫之：

```
int i;
long l;
...
l = std::max(i,l); // ERROR: argument types don't match
```

不過你倒是可以明白地宣告引數型別（這樣也就確定了回返值的型別）：

```
l = std::max<long>(i,l);   // OK
```

4.4.2 swap()：值互換

函式 swap()用來交換兩物件的值。其泛用型實作碼定義於<algorithm>：

```
namespace std {
    template<class T>
    inline void swap(T& a, T& b) {
       T tmp(a);
       a = b;
       b = tmp;
    }
}
```

運用這個函式，你可以如此交換任意變數 x 和 y 的值：

```
std::swap(x,y);
```

當然啦，只有當 swap() 所依賴的 *copy* 建構動作和 *assignment* 操作行爲存在時，這個呼叫才可能有效。

swap() 的最大優勢在於，透過 template specialization（模板特化）或 function overloading（函式多載化），我們可以爲更複雜的型別提供特殊的實作版本；我們可以交換物件內部成員，不必勞師動衆地反復指派（賦值），這無疑將大大地節約時間。標準程式庫中的所有容器（6.1.2 節, p147）以及 strings（11.2.8 節, p490）都運用了這項技術。舉個例子，有個簡單容器，僅僅內含一個 array 和一個成員（用來指示 array 元素個數量），那麼爲它特別實作的 swap() 可以是這樣：

```
class MyContainer {
    private:
      int* elems;       // dynamic array of elements
      int numElems;     // number of elements
    public:
      ...
      // implementation of swap()
      void swap(MyContainer& x) {
        std::swap(elems,x.elems);
        std::swap(numElems,x.numElems);
      }
      ...
};

// overloaded global swap() for this type
inline void swap (MyContainer& c1, MyContainer& c2)
{
    c1.swap(c2); // calls implementation of swap()
}
```

你瞧，呼叫 swap()而非透過反覆指派（賦值）動作來交換兩容器的值，會帶來效率上的提升。對於你自己定義的型別，如果確實能夠帶來效率上的改善，你就應該義不容辭地爲它提供 swap()特化版本。

4.5 輔助性的「比較運算子」（Comparison Operators）

有四個 template functions，分別定義了 !=, >, <=, >= 四個比較運算子。它們都是利用運算子 == 和 < 完成的。這四個函式定義於 <utility>：

```
namespace std {
    namespace rel_ops {
        template <class T>
        inline bool operator!= (const T& x, const T& y) {
            return !(x == y);
        }

        template <class T>
        inline bool operator> (const T& x, const T& y) {
            return y < x;
        }

        template <class T>
        inline bool operator<= (const T& x, const T& y) {
            return !(y < x);
        }

        template <class T>
        inline bool operator>= (const T& x, const T& y) {
            return !(x < y);
        }
    }
}
```

只需定義 < 和 == 運算子，你就可以使用它們。只要加上 using namespace std::rel_ops，上述四個比較運算子就自動獲得了定義。例如：

```
#include <utility>

class X {
    ...
    public:
```

```
        bool operator== (const X& x) const;
        bool operator< (const X& x) const;
        ...
};

void foo()
{
    using namespace std::rel_ops; // make !=, >, etc., available
    X x1, x2;
    ...

    if (x1 != x2) {
        ...
    }
    ...

    if (x1 > x2) {
        ...
    }
    ...
}
```

注意，這些運算子都定義於 std 的次命名空間（sub-namespace）rel_ops 中。之所以如此安排，是爲了防止和用戶（可能）定義的全域命名空間中的同類形運算子發生衝突。於是，就算你這樣使用 using directive：

```
using namespace std;  // operators are not in global scope
```

因而把 std 的全部識別字引入全域命名空間，也沒問題。

另一方面，那些想向 rel_ops 借一臂之力的用戶可以這麼做：

```
using namespace std::rel_ops; // operators are in global scope
```

於是四個新的運算子就輕鬆到手了，無需使用複雜的搜尋規則來引用它們。

某些實作版本採用兩個不同的引數型別來定義上述 template：

```
namespace std {
    template <class T1, class T2>
    inline bool operator!=(const T1& x, const T2& y) {
        return !(x == y);
    }
    ...
}
```

這麼做的好處是，兩個運算元的型別可以不同（只要它們之間「可以比較」就行）。但這並非 C++ 標準程式庫所支援的作法。所以，如果想占這個便宜，就得付出可攜性方面的代價。

4.6 表頭檔 <cstddef> 和 <cstdlib>

表頭檔 <cstddef> 和 <cstdlib> 和其 C 對應版本相容，在 C++ 程式中經常用到。它們是 C 表頭檔 <stddef.h> 和 <stdlib.h> 的較新版本，定義了一些常用的常數、巨集、型別和函式。

4.6.1 <cstddef> 內的各種定義

表 4.6 列出表頭檔 <cstddef> 的各個定義項。NULL 通常用來表明一個不指向任何物件的指標，其實就是 0（其型別可以是 int，也可以是 long）。注意，C 語言中的 NULL 通常定義爲(void*)0。在 C++ 中這並不正確，NULL 的型別必須是個整數型別，否則你無法將 NULL 指派給一個指標。這是因爲 C++ 並沒有定義從 void* 到任何其他型別的自動轉型操作[10]。NULL 同時也定義於表頭檔 <cstdio>, <cstdlib>, <cstring>, <ctime>, <cwchar>, <clocale> 內。

識別字	意義
NULL	指標值，用來表示「未定義」或「無值」。
size_t	一種無正負號的型別，用來表示大小（例如元素個數）。
ptrdiff_t	一種帶有正負號的型別，用來表示指標之間的距離。
offsetof	表示一個成員在 struct 或 union 中的偏移量。

表 4.6 <cstddef> 中的定義項

4.6.2 <cstdlib> 內的各種定義

表 4.7 列出表頭檔 <cstdlib> 內最重要的一些定義。常數 EXIT_SUCCESS 和 EXIT_FAILURE 用來當做 exit() 的引數，也可以當做 main()的回返值。

經由 atexit()登錄的函式，在程式正常退出時會依登錄的相反次序被一一呼叫起來。無論是透過 exit()退出或從 main()尾部退出，都會如此，不傳遞任何引數。

[10] 鑒於 NULL 型別有這些晦澀的問題，有人建議 C++ 程式中最好不要使用 NULL，最好直接使用 0 或用戶自定的（例如）NIL 常數。不過我還是使用 NULL，所以本書範例程式中你還是可以看到它的蹤跡。

定義	意義
`exit(int status)`	退出（離開，exit）程式（並清理 static 物件）
`EXIT_SUCCESS`	程式正常結束。
`EXIT_FAILURE`	程式不正常結束。
`abort()`	退出程式（在某些系統上可能導致崩潰）。
`atexit (void (*function)())`	退出（exit）程式時呼叫某些函式。

表 4.7 `<cstdlib>` 中的定義項

函式 `exit()`和 `abort()`可用來在任意地點終止程式運行，無需返回 `main()`：

- `exit()` 會銷毀所有 static 物件，將所有緩衝區（buffer）清空（flushes），關閉所有 I/O 通道（channels），然後終止程式（之前會先呼叫經由 `atexit()`登錄的函式）。如果 `atexit()`登錄的函式擲出異常，就會喚起 `terminate()`。
- `abort()`會立刻終止函式，不做任何清理（clean up）工作。

這兩個函式都不會銷毀區域物件（local objects），因爲堆疊輾轉開展動作（stack unwinding）不會被執行起來。爲確保所有區域物件的解構式獲得呼叫，你應該運用異常（exceptions）或正常回返機制，然後再由 `main()`離開。

5

Standard Template Library
標準模板庫

STL（標準模板庫）是 C++ 標準程式庫的核心，它深刻影響了標準程式庫的整體結構。STL 是一個泛型（generic）程式庫，提供一系列軟體方案，利用先進、高效的演算法來管理資料。程式員無需了解 STL 的原理，便可享受資料結構和演算法領域中的這一革新成果。從程式員的角度看來，STL 是由一些可適應不同需求的群集類別（collection classes），和一些能夠在這些資料群集上運作的演算法構成。STL 內的所有組件都由 templates（模板）構成，所以其元素可以是任意型別。更妙的是，STL 建立了一個架構，在此架構下，你可以提供其他群集類別或演算法，與現有的組件搭配，共同運作。總之，STL 賦予 C++ 新的抽象層次。把 dynamic arrays（動態陣列）、linked list（串列）、binary trees（二元樹）之類的東西拋開吧，也不用再操心不同的搜尋演算法了。你只需使用恰當的群集類別，然後呼叫其成員函式和（或）演算法來處理資料，就萬事大吉。當然，如此的靈活性並非免費午餐，代價總是有的。首要的一點是，STL 並不好懂。也正因爲如此，本書傾注了好幾章篇幅，爲你講解 STL 的內容。這一章介紹 STL 的總體概念，探討其使用技術。第一個範例展示如何使用 STL，以及運用過程中有何考量。第 6 章至第 9 章詳細討論 STL 的各個組件（包括容器 containers、迭代器 iterators、仿函式 functors、演算法 algorithms），並提供更多範例。

5.1 STL 組件（STL Components）

若干精心勾畫的組件共同合作，構築起 STL 的基礎。這些組件中最關鍵的是容器、迭代器和演算法。

- 容器 **Containers**，用來管理某類物件的集合。每一種容器都有其優點和缺點，所以，爲了應付程式中的不同需求，STL 準備了不同的容器類型。容器可以是 arrays 或是 linked lists，或者每個元素有一個特別的鍵值（key）。
- 迭代器 **Iterators**，用來在一個物件群集（collection of objects）的元素上進行巡

訪動作。這個物件群集或許是個容器，或許是容器的一部分。迭代器的主要好處是，爲所有容器提供了一組很小的公共介面。利用這個介面，某個操作（operations）就可以行進至群集內的下一個元素。至於如何做到，當然取決於群集的內部結構。不論這個群集是 array 或 tree，此一行進動作都能成功。爲什麼？因爲每一種容器都提供了自己的迭代器，而這些迭代器了解該種容器的內部結構，所以能夠知道如何正確行進。

迭代器的介面和一般指標差不多，以 `operator++` 累進，以 `operator*` 提領所指之值。所以，你可以把迭代器視爲一種 *smart pointer*，能夠把「前進至下一個元素」的意圖轉換成合適的操作。

- 演算法 **Algorithms**，用來處理群集內的元素。它們可以出於不同的目的而搜尋、排序、修改、使用那些元素。透過迭代器的協助，我們只需撰寫一次演算法，就可以將它應用於任意容器之上，這是因爲所有容器的迭代器都提供一致的介面。你還可以提供一些特殊的輔助性函式供演算法呼叫，從而獲取更佳的靈活性。這樣你就可以一方面運用標準演算法，一方面適應自己特殊或複雜的需求。例如，你可以提供自己的搜尋準則或元素合併時的特殊操作。

STL 的基本觀念就是將資料和操作分離。資料由容器類別加以管理，操作則由可定制（configurable）的演算法定義之。迭代器在兩者之間充當粘合劑，使任何演算法都可以和任何容器交互運作（圖 5-1）。

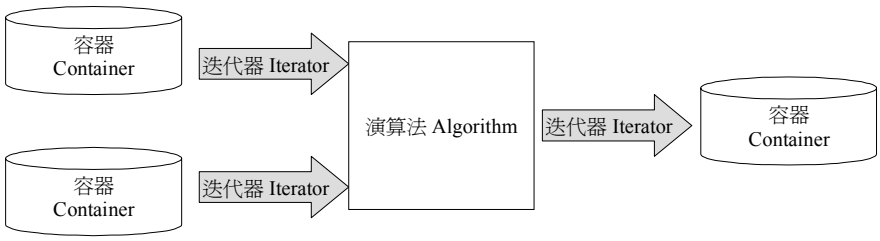


圖 5.1 STL 組件之間的合作

STL 將資料和演算法分開對待，而不是合併考慮。因此從某種意義上說，STL 的概念和物件導向程式編寫（OOP）的最初思想是矛盾的。然而這麼做有著很重要的原因。首先，你可以將各種容器與各種演算法結合起來，在很小的框架（framework）內達成非常大的彈性。

STL 的一個根本特性是，所有組件都可以針對任意型別（types）運作。顧名思義，所謂 standard **template** libary 意味其內的所有組件都是「可接受任意型別」的 templates，前提是這些型別必須能夠執行必要操作。因此 STL 成了泛型編程（*generic*

programming）概念下的一個出色範例。容器和演算法對任意型別（types）和類別（classes）而言，都已經被一般化了。

STL 甚至提供更泛型化的組件。藉由特定的配接器（*adapters*）和仿函式（*functors*，或稱 *function objects*），你可以補充、約束或訂製演算法，以滿足特別需求。目前說這些似乎爲時太早，眼下我還是先透過實例，循序漸進地講解概念，這才是理解並熟悉 STL 的最佳方法。

5.2 容器（Containers）

容器類別（簡稱容器）用來管理一組元素。爲了適應不同需要，STL 提供了不同的容器，如圖 5.2。

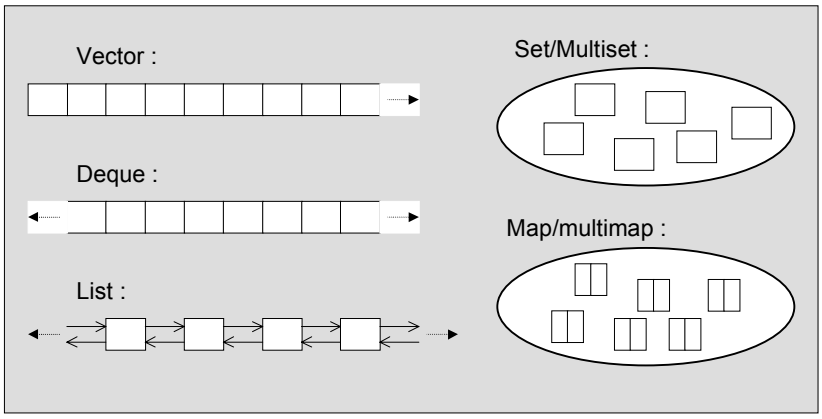


圖 5.2 STL 的容器種類

總的來說，容器可分爲兩類：

1. 序列式容器 **Sequence containers**，此乃可序（*ordered*）群集，其中每個元素均有固定位置 — 取決於插入時機和地點，和元素值無關。如果你以尾附方式對一個群集置入六個元素，它們的排列次序將和置入次序一致。STL 提供三個定義好的序列式容器：`vector`, `deque`, `list`。
2. 關聯式容器 **Associative containers**，此乃已序（*sorted*）群集，元素位置取決於特定的排序準則。如果你將六個元素置入這樣的群集中，它們的位置取決於元素值，和插入次序無關。STL 提供了四個關聯式容器：`set`, `multiset`, `map`, `multimap`。

關聯式容器也可被視爲特殊的序列式容器，因爲已序（*sorted*）群集正是根據某個排序準則排列（*ordered*）而成。如果你曾經用過其他群集庫，例如 Smalltalk 和

NIHCL[1]所提供者，你可能已經估計到這一點。在那些程式庫中，*sorted collections* 由 *ordered collections* 衍生而來。不過請注意，STL 所提供的群集型別（collection types）彼此獨立，各自實現，毫無關聯（譯註：意指其間並無 classes 繼承關係）。

關聯式容器自動對其元素排序，這並不意味它們就是用來排序的。你也可以對序列式容器的元素加以手動排序。自動排序帶來的主要優點是，當你搜尋元素時，可獲得更佳效率。更明確地說你可以放心使用二分搜尋法（binary search）。該演算法具有對數（logarithmic）複雜度，而非線性複雜度。這什麼意思呢？如果你想在 1000 個元素中搜尋某個元素，平均而言只需 10 次比較，而非 500 次比較（參見 2.3 節, p21）。因此自動排序只不過是關聯式容器的一個（有用的）副作用而已。

下面各小節詳細討論各種容器類別。其中特別講解了容器的典型實作法。嚴格說來，*C++ Standard* 並未定義某一種容器的具體實作法。然而 *C++ Standard* 所規定的行爲和其對複雜度的要求，讓實作者沒有太多變化餘地。所以實際上各個實作版本之間只在細節上有所差異。第 6 章會談到容器類別的確切行爲、描述它們共有和特有的能力、並詳細分析其成員函式。

5.2.1 序列式容器（Sequence Containers）

STL 內部預先定義好以下三個序列式容器：

- Vectors
- Deques
- Lists

此外你也可以將 strings 和 array 當作一種序列式容器。

Vectors

Vector 將其元素置於一個 dynamic array 中加以管理。它允許隨機存取，也就是說你可以利用索引直接存取任何一個元素。在 array 尾部附加元素或移除元素均非常快速[2]，但是在 array 中部或頭部安插元素就比較費時，因爲，爲了保持原本的相對次序，安插點之後的所有元素都必須移動，挪出位子來。

---

[1] The National Institute of Health's Class Library，是最早的 C++ 類別庫之一。

[2] 嚴格說來，元素尾附動作是一種「分攤後的（*amortized*）」高速。單一附加動作可能是緩慢的，因爲 vector 可能需要重新分配記憶體，並將現有元素拷貝到新位置。不過這種事情不常發生，所以總體看來這個操作十分迅速。見 p22 的複雜度討論。

以下例子針對整數型別定義了一個 vector，插入 6 個元素，然後列印所有元素：

```
// stl/vector1.cpp

#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> coll;     // vector container for integer elements

  // append elements with values 1 to 6
  for (int i=1; i<=6; ++i) {
     coll.push_back(i);
  }

  // print all elements followed by a space
  for (int i=0; i<coll.size(); ++i) {
     cout << coll[i] << ' ';
  }
  cout << endl;
}
```

其中的：

```
#include <vector>
```

含入 vectors 的表頭檔。

以下宣告式：

```
vector<int> coll;
```

生成一個「元素型別爲 `int`」的 vector。由於沒有任何初始化參數，***default*** 建構式就將它建構爲空群集。

`push_back()`函式可爲容器附加元素:

```
coll.push_back(i);
```

所有序列式容器都提供有這個成員函式。

`size()`成員函式回傳容器中的元素個數：

```
for (int i=0; i<coll.size(); ++i) {
   ...
}
```

所有容器類別都提供有這個函式。

你可以通過 *subscript*（下標）運算子[]，存取 vector 內的某個元素：

```
cout << coll[i] << ' ';
```

在這裡，元素被寫至標準輸出裝置，所以整個程式的輸出是：

```
1 2 3 4 5 6
```

Deques

所謂 deque（發音類似 "check"[3]），是 "double-ended queue" 的縮寫。它是一個 dynamic array，可以向兩端發展，因此不論在尾部或頭部安插元素都十分迅速。在中間部份安插元素則比較費時，因爲必須移動其他元素。

以下例子宣告了一個浮點數型別的 deque，並在容器頭部安插 1.1 至 6.6 共 6 個元素，最後列印出所有元素。

```
// stl/deque1.cpp

#include <iostream>
#include <deque>
using namespace std;

int main()
{
  deque<float> coll; // deque container for floating-point elements

  // insert elements from 1.1 to 6.6 each at the front
  for (int i=1; i<=6; ++i) {
     coll.push_front(i*1.1); // insert at the front
  }

  // print all elements followed by a space
  for (int i=0; i<coll.size(); ++i) {
     cout << coll[i] << ' ';
  }
  cout << endl;
}
```

[3] 有時候 "deque" 聽起來頗爲類似 "hack"，不過這純屬巧合 ☺

本例之中，

```
#include <deque>
```

含入 deques 的表頭檔。

下面這一句：

```
deque<float> coll;
```

會產生一個空的浮點數群集。

`push_front()`函式可以用來安插元素：

```
coll.push_front(i*1.1);
```

它會將元素安插於群集前端。注意，這種安插方式造成的結果是，元素排放次序與安插次序恰好相反，因爲每個元素都安插於上一個元素的前面。因此，程式輸出如下：

```
6.6 5.5 4.4 3.3 2.2 1.1
```

你也可以使用成員函式 `push_back()`在 deque 尾端附加元素。vector 並未提供 `push_front()`，因爲其時間效能很差（在 vector 頭端安插一個元素，需要移動全部元素）。一般而言，STL 容器只提供通常具備良好時間效能的成員函式（所謂「良好」的時間效能，通常意味具有常數複雜度或對數複雜度），這可以防止程式員呼叫性能很差的函式。

Lists

List 由雙向串列（doubly linked list）實作而成。這意味 list 內的每個元素都以一部分記憶體指示其前導元素和後繼元素。List 不提供隨機存取，因此如果你要存取第 10 個元素，你必須沿著串鏈依次走過前 9 個元素。不過，移動至下一個元素或前一個元素的行爲，可以在常數時間內完成。因此一般的元素存取動作會花費線性時間（平均距離和元素數量成比例）。這比 vector 和 deque 提供的「分攤性（*amoritzed*）」常數時間，性能差很多。

List 的優勢是：在任何位置上執行安插或刪除動作都非常迅速，因爲只需改變連結（links）就好。這表示在 list 中間位置移動元素比在 vector 和 deque 快得多。

以下例子產生一個空 list，準備放置字元，然後將 `'a'` 至 `'z'` 的所有字元插入其中，利用迴圈每次列印並移除群集的第一個元素，從而印出所有元素：

```
// stl/list1.cpp

#include <iostream>
#include <list>
```

```
using namespace std;

int main()
{
  list<char> coll; // list container for character elements

  // append elements from 'a' to 'z'
  for (char c='a'; c<='z'; ++c) {
     coll.push_back(c);
  }

  /* print all elements
  * - while there are elements
  * - print and remove the first element
  */
  while (! coll.empty()) {
     cout << coll.front() << ' ';
  coll.pop_front();
  }
  cout << endl;
}
```

就像先前的例子一樣，表頭檔 `<list>` 內含 lists 的宣告。以下定義一個「元素型別爲字元」的 list：

```
list<char> coll;
```

成員函式 `empty()`的回傳值告訴我們容器中是否還有元素。只要這個函式回傳 false（也就是說容器內還有元素），迴圈就繼續進行：

```
while (! coll.empty()) {
    ...
}
```

迴圈之內，成員函式 `front()`會傳回第一個元素：

```
cout << coll.front() << ' ';
```

`pop_front()`函式會刪除第一個元素:

```
coll.pop_front();
```

注意，`pop_front()` 並不會傳回被刪除的元素，所以你無法將上述兩個語句合而爲一。

程式的輸出結果取決於所用字集。如果是 ASCII 字集，輸出如下[4]：

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

當然，爲了列印 list 的所有元素而「採用迴圈輸出並刪除第一個元素」的做法實在是很奇怪。通常你只需走訪所有元素即可。lists 並沒有提供以 `operator[]` 直接存取元素的能力，因爲 lists 不支持隨機存取，如果採用 `operator[]` 會導致不良效能。運用迭代器也可以走訪並列印所有元素。介紹過迭代器後我會給一個例子。如果你等不及，請跳到 p84。

Strings

你也可以將 string 當作 STL 容器來使用。這裡的 strings 是指 C++ string 類別族系（`basic_string<>, string, wstring`）的物件，第 11 章對此有所介紹。Strings 跟 vectors 很相似，只不過其元素爲字元。11.2.13 節, p497 對此有詳細解說。

Arrays

另一種容器並非是個類別（class），而是 C/C++ 語言核心所支持的一個型別（type）：具有靜態大小或動態大小的 array。但 array 並非 STL 容器，它們並沒有類似 `size()` 和 `empty()` 等成員函式。儘管如此，STL 的設計允許你針對 array 呼叫 STL 演算法。當我們以 static arrays 作爲初值列（initializer list）時，這一點特別有用。

Array 的運用並無新意，面對 arrays 使用演算法，才是新的議題。這個議題將在 6.7.2 節, p218 討論。

值得注意的是，我們沒有必要再直接編寫 dynamic array 了。Vectors 已經具備了 dynamic array 的全部性質，並提供更安全更便捷的介面。詳見 6.2.3 節, p155。

5.2.2 關聯式容器（Associative Containers）

關聯式容器依據特定的排序準則，自動爲其元素排序。排序準則以函式形式呈現，用來比較元素值（value）或元素鍵（key）。預設情況下以 `operator<` 進行比較，不過你也可以提供自己的比較函式，定義出不同的排序準則。

通常關聯式容器由二元樹（binary tree）實作出來。在二元樹中，每個元素（節點）都有一個父節點和兩個子節點；左子樹的所有元素都比自己小，右子樹的所有元素都比自己大。關聯式容器的差別主要在於元素的類型以及處理重複元素時的方

[4] 如果是 ASCII 以外的字集，輸出結果可能包含非字母字元，甚至可能什麼都沒有（如果 'z' 不大於 'a' 的話）。

式（態度）。

下面是 STL 中預先定義好的關聯容器。由於訪問其中元素需要用到迭代器（iterator），所以我推遲至 p87 討論過迭代器後再舉例子。

- **Sets**
  Set 的內部元素依據其值自動排序，每個元素值只能出現一次，不允許重複。
- **Multisets**
  Multiset 和 set 相同，只不過它允許重複元素，也就是說 multiset 可包括多個數值相同的元素。
- **Maps**
  Map 的元素都是「實值/鍵值」所形成的一個對組（*key/value* pairs）。每個元素有一個鍵，是排序準則的基礎。每一個鍵只能出現一次，不允許重複。Map 可被視爲關聯式陣列（associative array），也就是具有任意索引型別（index type）的陣列（詳見 p91）。
- **Multimaps**
  Multimap 和 map 相同，但允許重複元素，也就是說 multimap 可包含多個鍵值（*key*）相同的元素。Multimap 可被當作「字典」（譯註：*dictionary*，某種資料結構）使用。p209 有個範例。

所有關聯式容器都有一個可供選擇的 template 引數，指明排序準則。預設採用 `operator<`。排序準則同時也用來測試互等性（equality）：如果兩個元素都不小於對方，則兩者被視爲相等。

你可以將 set 視爲一種特殊的 map：其元素實值就是鍵值。實際產品中，所有這些關聯式容器通常都由二元樹（binary tree）實作而成。

5.2.3 容器配接器（Container Adapters）

除了以上數個根本的容器類別，爲滿足特殊需求，C++ 標準程式庫還提供了一些特別的（並且預先定義好的）容器配接器，根據基本容器類別實作而成。包括：

- **Stacks**
  名字說明了一切。Stack 容器對元素採取 LIFO（後進先出）管理策略。
- **Queues**
  Queue 容器對元素採取 FIFO（先進先出）管理策略。也就是說，它是個普通的緩衝區（buffer）。
- **Priority Queues**
  Priority Queue 容器中的元素可以擁有不同的優先權。所謂優先權，乃是基於程

式員提供的排序準則（預設使用 operator<）而定義。Priority queue 的效果相當於這樣一個 buffer：「下一元素永遠是 queue 中優先權最高的元素」。如果同時有多個元素具備最高優先權，則其次序無明確定義。

5.3 迭代器（Iterators）

迭代器是一個「可巡訪 STL 容器內全部或部分元素」的物件。一個迭代器用來指出容器中的一個特定位置。基本操作如下：

- **Operator ***
  傳回當前位置上的元素值。如果該元素擁有成員，你可以透過迭代器，直接以 operator-> 取用它們[5]。
- **Operator ++**
  將迭代器前進至下一元素。大多數迭代器還可使用 operator-- 退回到前一個元素。
- **Operators == 和 Operator != **
  判斷兩個迭代器是否指向同一位置。
- **Operator =**
  爲迭代器賦值（將其所指元素的位置指派過去）。

這些操作和 C/C++「操作 array 元素」時的指標介面一致。不同之處在於，迭代器是個所謂的 *smart pointer*s，具有走訪複雜資料結構的能力。其下層運作機制取決於其所走訪的資料結構。因此，每一種容器型別都必須提供自己的迭代器。事實上每一種容器都將其迭代器以巢狀（nested）方式定義於內部。因此各種迭代器的介面相同，型別卻不同。這直接導出了泛型程式設計的概念：所有操作行爲都使用相同介面，雖然它們的型別不同。因此，你可以使用 templates 將泛型操作公式化，使之得以順利運作那些「能夠滿足介面需求」的任何型別。

所有容器類別都提供有一些成員函式，使我們得以獲得迭代器並以之遍訪所有元素。這些函式中最重要的是：

- **begin()**
  傳回一個迭代器，指向容器起始點，也就是第一元素（如果有的話）的位置。
- **end()**
  傳回一個迭代器，指向容器結束點。結束點在最後一個元素之後，這樣的迭代器又稱作「逾尾（*past-the-end*）」迭代器。

[5] 某些老舊的 STL 環境並不對迭代器支援 operator->。

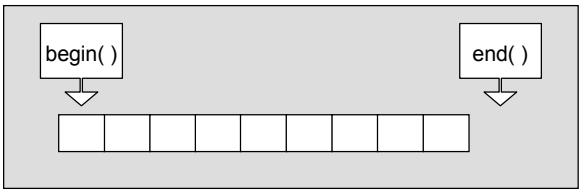


圖 5.3 容器的 begin() 和 end() 成員函式

於是，begin()和 end()形成了一個半開區間（half-open range），從第一個元素開始，到最後一個元素的下一位置結束（圖 5.3）。半開區間有兩個優點：

1. 爲「走訪元素時，迴圈的結束時機」提供了一個簡單的判斷依據。只要尚未到達 end()，迴圈就可以繼續進行。
2. 不必對空區間採取特殊處理手法。空區間的 begin()就等於 end()。

下面這個例子展現了迭代器的用法，將 list 容器內的所有元素列印出來（這就是 p79 那個 list 實例的改進版）。

```
// stl/list2.cpp

#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> coll; // list container for character elements

  // append elements from 'a' to 'z'
  for (char c='a'; c<='z'; ++c) {
    coll.push_back(c);
  }

  /* print all elements
   * - iterate over all elements
  */
  list<char>::const_iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
  }
  cout << endl;
}
```

首先產生一個 list，然後填入'a' ~ 'z'字元，然後在 for 迴圈中印出所有元素：

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
   cout << *pos << ' ';
}
```

迭代器 pos 宣告於迴圈之前，其型別是「指向容器中的不可變元素」的迭代器：

```
list<char>::const_iterator pos;
```

任何一種容器都定義有兩種迭代器型別：

1. *container*::iterator
   這種迭代器以「讀/寫」模式走訪元素。
2. *container*::const_iterator
   這種迭代器以「唯讀」模式走訪元素。

例如，在 class list 之中，它們的定義可能是這樣：

```
namespace std {
  template <class T>
  class list {
  public:
    typedef ... iterator;
    typedef ... const_iterator;
    ...
  };
}
```

至於其中 iterator 和 const_iterator 的確切型別，則於實作中定義。

在迴圈中，迭代器 pos 以容器的第一個元素位置爲初值：

```
pos = coll.begin()
```

迴圈不斷進行，直到 pos 到達容器的結束點：

```
pos != coll.end()
```

在這裡，pos 是在和「逾尾（*past-the-end*）」迭代器作比較。當迴圈內部執行 ++pos 述句，迭代器 pos 就會前進到下一個元素。

總而言之，pos 從第一個元素開始，逐一訪問了每一個元素，直到抵達結束點爲止（圖 5.4）。如果容器內沒有任何元素，coll.begin()等於 coll.end()，迴圈根本不會執行。

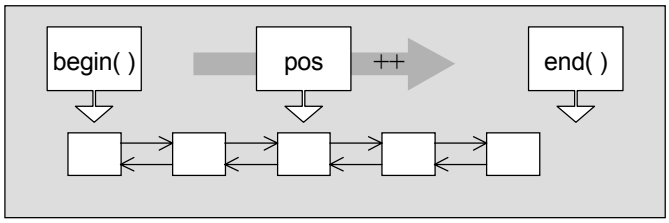


圖 5.4 迭代器 pos 走訪 list 的每一個元素

在迴圈內部，述句 *pos 代表當前（current）元素。本例將它輸出之後，又接著輸出了一個空格。你不能改變元素內容，因爲 pos 是個 const_iterator，從迭代器的觀點看去，元素是常量，不能更改。不過如果你採用非常量（nonconstant）迭代器，而且元素本身的型別也是非常量（nonconstant），那麼就可以透過迭代器來改變元素值。例如：

```
// make all characters in the list uppercase
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    *pos = toupper(*pos);
}
```

注意，這裏使用「前置式遞增（preincrement）」++pos，因爲它比「後置式遞增（postincrement）」pos++ 效率高。後者需要一個額外的臨時物件，因爲它必須存放迭代器的原本位置並將它回傳，所以一般情況下最好使用 ++pos，不要用 pos++。也就是說，不要這麼寫：

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                              ^^^^^ // OK, but slower
    ...
}
```

爲了這個理由，我建議優先採用前置式遞增（pre-increment）或前置式遞減（pre-decrement）運算子。

5.3.1 關聯式容器的運用實例

上個例子中的迭代器迴圈可應用於任何容器，只需調整迭代器型別即可。現在你知道如何列印關聯式容器內的元素了吧。下面是使用關聯式容器的一些例子。

Sets 和 Multisets 運用實例

第一個例子展示如何在 set 之中安插元素，並使用迭代器來列印它們。

```
// stl/set1.cpp

#include <iostream>
#include <set>

int main()
{
  // type of the collection
  typedef std::set<int> IntSet;

  IntSet coll; // set container for int values

  /* insert elements from 1 to 6 in arbitrary order
  *-value1 gets inserted twice
  */
  coll.insert(3);
  coll.insert(1);
  coll.insert(5);
  coll.insert(4);
  coll.insert(1);
  coll.insert(6);
  coll.insert(2);

  /* print all elements
  * - iterate over all elements
  */
  IntSet::const_iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << *pos << ' ';
  }
  std::cout << std::endl;
}
```

一如以往，include 指令：

```
#include <set>
```

定義了 sets 的所有必要型別和操作。

既然容器的型別要用到好幾次，不妨先定義一個短一點的名字：

```
typedef set<int> IntSet;
```

這個述句定義的 IntSet 型別，其實就是「元素型別爲 int 的一個 set」。這種型別有預設的排序準則，以 operator< 爲依據，對元素進行排序。這意味元素將以遞增方式排列。如果希望以遞減方式排列，或是希望使用一個完全不同的排序準則，你可以將該準則傳入做爲第二個 template 參數。下面例子即是將元素以遞減方式排列[6]：

```
typedef set<int,greater<int> > IntSet;
```

以上所用的 greater<> 是一個預先定義的仿函式（functor, or function object），我將在 5.9.2 節, p131 討論它。8.1.1 節, p294 另有一個例子，僅使用元素的部分資料（例如 ID）進行排序。

所有關聯式容器都提供有一個 insert()成員函式，用以安插新元素：

```
coll.insert(3);
coll.insert(1);
...
```

新元素會根據排序準則自動安插到正確位置。注意，你不能使用序列式容器的 push_back()和 push_front()函式，它們在這裏毫無意義，因爲你沒有權力指定新元素的位置。

所有元素（不論以任何次序）安插完畢後，容器的狀態如圖 5.5。元素以已序狀態（*sorted*）存放於內部 tree 結構。任何一個元素（節點）的左子樹的所有元素，永遠小於右子樹的所有元素（這裏的「小於」是指就當前排序準則而言）。Sets 不允許存在重複元素，所以容器裡頭只有一個 "1"。

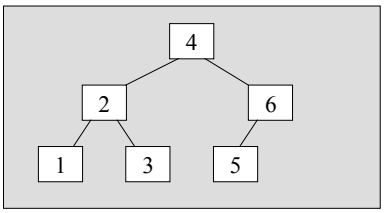


圖 5.5 一個 Set，擁有 6 個元素

[6] 注意，兩個 ">" 符號之間一定要有一個空格。">>" 會被編譯器視爲一個右移（right-shift）運算子，從而導致語法錯誤。

現在，我們可以運用先前 list 例中所用的相同迴圈來列印 set 內的元素。以一個迭代器走訪全部元素，並逐一列印出來：

```
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
  cout << *pos << ' ';
}
```

再一次我要提醒你，由於迭代器是容器定義的，所以無論容器內部結構如何複雜，它都知道如何行事。舉個例子，如果迭代器指向第三個元素，運算子++ 便會將它移動到上端的第四個元素，再一次++，便會將它移動到下方第五個元素（圖 5.6）。

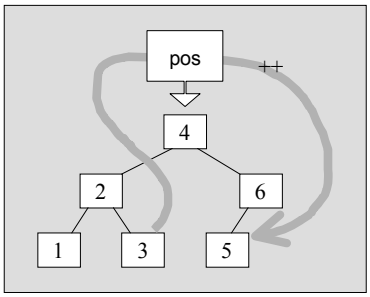


圖 5.6 迭代器 pos 走訪 Set 內的元素

以下是輸出結果：

```
1 2 3 4 5 6
```

如果你想使用 multiset 而不是 set，唯一需要改變的就是容器的型別（set 和 multiset 的定義被置於同一個表頭檔）：

```
typedef multiset<int> IntSet;
```

由於 multiset 允許元素重複存在，因此其中可包含兩個數值皆爲 1 的元素。輸出結果如下：

```
1 1 2 3 4 5 6
```

Maps 和 Multimaps 的運用實例

Map 的元素是成對的鍵值/實值（key/value）。因此其宣告、元素安插、元素存取皆和 set 有所不同。下面是一個 multimap 運用實例：

```
// stl/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  // type of the collection
  typedef multimap<int,string> IntStringMMap;

  IntStringMMap coll; // container for int/string values

  // insert some elements in arbitrary order
  // - a value with key 1 gets inserted twice
  coll.insert(make_pair(5,"tagged"));
  coll.insert(make_pair(2,"a"));
  coll.insert(make_pair(1,"this"));
  coll.insert(make_pair(4,"of"));
  coll.insert(make_pair(6,"strings"));
  coll.insert(make_pair(1,"is"));
  coll.insert(make_pair(3,"multimap"));

  /* print all element values
  * - iterate over all elements
  * - element member second is the value
  */
  IntStringMMap::iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
       cout << pos->second << ' ';
  }
  cout << endl;
}
```

程式的輸出結果可能是這樣：

```
this is a multimap of tagged strings
```

不過由於 "this" 和 "is" 的鍵值相同，兩者的出現順序也可能反過來。

拿這個例子和 p87 的 set 實例作比較，你會發現以下兩點不同：

1. 這裡的元素是成對的鍵值/實值(*key/value* pair)，所以你必須首先生成這個 pair，再將它插入群集內部。輔助函式 `make_pair()` 正是爲了這個目的而打造。這個問題的細節，以及其他安插方法，請見 p203，
2. 迭代器所指的是「鍵值/實值」對組（*key/value* pair），因此你無法一口氣列印它們，你必須取出 `pair` 的成員，亦即所謂的 `first` 和 `second`（`pair` 型別在 4.1 節, p33 介紹過）。因此，以下述句：

   ```
   pos->second
   ```

   便取得了「鍵值/實值」對組中的第二部分，也就是 multimap 元素的實值(*value*)。和一般指標的情形一樣，上述述句就是以下述句的簡寫方案[7] ：

   ```
   (*pos).second
   ```

   同樣道理，以下述句：

   ```
   pos->first
   ```

   取得「鍵值/實值」對組中的第一部分，也就是 multimap 元素的鍵值（*key*）。

Multimaps 也可以用來作爲 *dictionaries*，詳見 p209 實例。

將 Maps 當作關聯式陣列（associative arrays）

如果上述例子中以 map 取代 multimap，輸出結果就不會有重複鍵值（*key*s），實值（*value*s）則和上述結果一樣。一個「鍵值/實值」對組所形成的群集中，如果所有鍵值都是獨一無二的，我們可將它視爲一個關聯式陣列（associative array）。考慮以下例子：

```
// stl/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  /* type of the container:
  *-map: elements key/value pairs
  *-string: keys have type string
  *-float: values have type float
  */
```

[7] 某些老舊環境並沒有實現出 `iterator->`，這時候你就只能使用第二個表述式了。

```
    typedef map<string,float> StringFloatMap;
    StringFloatMap coll;

    // insert some elements into the collection
    coll["VAT"] = 0.15;
    coll["Pi"] = 3.1415;
    coll["an arbitrary number"] = 4983.223;
    coll["Null"] = 0;

    /* print all elements
    * - iterate over all elements
    * - element member first is the key
    * - element member second is the value
    */
    StringFloatMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
         cout << "key: \"" << pos->first << "\" "
              << "value: " << pos->second << endl;
    }
}
```

當我們宣告容器型別的時候，必須同時指定鍵值（*key*）和實值（*value*）的型別：

```
typedef map<string,float> StringFloatMap;
```

Maps 允許你使用 `operator[]` 安插元素：

```
coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;
```

在這裡，以鍵值爲索引，鍵值可爲任意型別。這正是關聯式陣列的介面。所謂關聯式陣列就是：索引可以採用任何型別。

注意這裏的 ***subscript***（下標）運算子和一般 array 所用的行爲有些不同：在這裡，索引可以不對應於任何元素。如果你指定了一個新索引（新鍵值），會導致產生一個對應的新元素，並被安插於 map。也就是說，沒有任何索引是「錯誤」的。因此，以下述句：

```
coll["Null"] = 0;
```

其中的式子：

```
coll["Null"]
```

會產生一個新元素，鍵值爲 "Null"。然後 ***assignment***（賦值）運算子再將該元素的實值設爲 0（並轉化爲 `float`）。6.6.3 節, p205 更詳細地討論了如何將 maps 當作關聯式陣列。

Multimaps 不允許我們使用 *subscript*（下標）運算子，因爲 multimaps 允許單一索引對應到多個不同元素，而下標運算子卻只能處理單一實值。你必須先產生一個「鍵值/實值」對組，然後再插入 multimap，見 p90。當然，對於 maps 也可以這麼做，細節請參考 p202。

存取 multimaps 或 maps 的元素時，你必須透過 `pair` 結構的 `first` 成員和 `second` 成員，才能取得鍵值（*key*）和實值（*value*）。上述程式的輸出如下：

```
key: "Null" value: 0
key: "Pi" value: 3.1415
key: "VAT" value: 0.15
key: "an arbitrary number" value: 4983.22
```

5.3.2 迭代器分類（Iterator Categories）

除了基本操作之外，迭代器還有其他能力。這些能力取決於容器的內部結構。STL 總是只提供效率上比較出色的操作，因此，如果容器允許隨機存取（例如 vectors 或 deques），那麼它們的迭代器也能進行隨機操作（例如直接讓迭代器指向第五元素）。

根據能力的不同，迭代器被劃分爲五種不同類屬。STL 預先定義好的所有容器，其迭代器均屬於以下兩種分類：

1. 雙向迭代器（**Bidirectional iterator**）
   顧名思義，雙向迭代器可以雙向行進：以遞增（increment）運算前進或以遞減（decrement）運算後退。list、set、multiset、map 和 multimap 這些容器所提供的迭代器都屬此類。
2. 隨機存取迭代器（**Random access iterator**）
   隨機存取迭代器不但具備雙向迭代器的所有屬性，還具備隨機訪問能力。更明確地說，它們提供了「迭代器算術運算」必要的運算子（和「一般指標的算術運算」完全對應）。你可以對迭代器增加或減少一個偏移量、處理迭代器之間的距離、或是使用 `<` 和 `>` 之類的 *relational*（相對關係）運算子來比較兩個迭代器。vector, deque 和 strings 所提供的迭代器都屬此類。

其他迭代器類型在 7.2 節, p251 介紹。

爲了撰寫儘可能與容器型別無關的泛型程式碼，你最好不要使用隨機存取迭代器（*random access iterator*s）的特有操作。例如以下例子，可以在任何容器上運作：

```
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

而下面這樣的程式碼就不是所有容器都適用了：

```
for (pos = coll.begin(); pos < coll.end(); ++pos) {
    ...
}
```

兩者的唯一區別在於測試迴圈條件時，第二例使用 operator<，第一例使用 operator!=。要知道，只有 *random access iterator*s 才支援 operator<，所以第二例中的迴圈對於 lists、sets 和 maps 無法運作。爲了寫出適用於任何容器的泛型程式碼，你應該使用 operator!= 而非 operator<。不過如此一來，程式碼的安全性可能有損，因爲如果 pos 的位置在 end()的後面，你未必便能發現（關於 STL 使用上的可能錯誤，請見 5.11 節, p136）。究竟使用哪種方式，取決於當時情況，取決於個人經驗，取決於你。

爲了避免誤解，我再強調一句。注意，我說的是類屬、分類（categories），不是類別（classes）。所謂類屬，只是定義迭代器的能力，無關乎迭代器的型別（type）。STL 的泛型概念可以以純抽象形式工作，也就是說，任何東西只要行爲「像」一個雙向迭代器，那麼它就是一個雙向迭代器。

5.4 演算法（Algorithms）

爲了處理容器內的元素，STL 提供了一些標準演算法，包括搜尋、排序、拷貝、重新排序、修改、數值運算等十分基本而普遍的演算法。

演算法並非容器類別的成員函式，而是一種搭配迭代器使用的全域函式。這麼做（譯註：意指搭配迭代器來使用）有一個重要優勢：所有演算法只需實作出一份，就可以對所有容器運作，不必爲每一種容器量身訂製。演算法甚至可以操作不同型別（types）之容器內的元素，也可以與用戶定義的容器搭配。這個概念大幅降低了程式碼的體積，提高了程式庫的能力和彈性。

注意，這裏所闡述的並非物件導向思維模式（OOP paradigm），而是泛型函式編程思維模式（generic functional programming paradigm）。在物件導向編程（OOP）概念裡，資料與操作合爲一體，在這裏則被明確劃分開來，再透過特定的介面彼此互動。當然這需要付出代價：首先，用法有失直觀，其次，某些資料結構和演算法之間並不相容。更有甚者，某些容器和演算法雖然勉強相容，卻毫無用處（也許導致很糟的效能）。因此，深入學習 STL 的概念並了解其缺陷，顯得十分重要，惟其如此，方能取其利而避其害。我將在本章剩餘篇幅中，透過實例詳細介紹它們。讓我們從簡單的 STL 演算法的運用入手。以下實例展現了某些演算法的使用方式：

```
// stl/algo1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll;
   vector<int>::iterator pos;

   // insert elements from 1 to 6 in arbitrary order
   coll.push_back(2);
   coll.push_back(5);
   coll.push_back(4);
   coll.push_back(1);
   coll.push_back(6);
   coll.push_back(3);

   // find and print minimum and maximum elements
   pos = min_element (coll.begin(), coll.end());
   cout << "min: " << *pos << endl;
   pos = max_element (coll.begin(), coll.end());
   cout << "max: " << *pos << endl;

   // sort all elements
   sort (coll.begin(), coll.end());

   // find the first element with value 3
   pos = find (coll.begin(), coll.end(),   // range
               3);                         // value
   // reverse the order of the found element with value 3 and
   // all following elements
   reverse (pos, coll.end());

   // print all elements
   for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
   }
   cout << endl;
}
```

爲了呼叫演算法，首先你必須含入表頭檔 <algorithm>：

```
#include <algorithm>
```

最先出現的是演算法 min_element()和 max_element()。呼叫它們時，你必須傳入兩個引數，定義出欲處理的元素範圍。如果想要處理容器內的所有元素，可以使用 begin()和 end()。兩個演算法都傳回一個迭代器，分別指向最小或最大元素。因此，以下述句：

```
pos = min_element (coll.begin(), coll.end());
```

演算法 min_element()回傳最小元素的位置（如果最小元素不只一個，則回傳第一個最小元素的位置）。以下述句印出該元素：

```
cout << "min: " << *pos << endl;
```

當然，你也可以合併上述兩個動作於單一述句：

```
cout << *max_element(coll.begin(),coll.end()) << endl;
```

接下來的演算法是 sort()。顧名思義，它將「由兩個引數設定出來」的區間內的所有元素加以排序。你可以（選擇性地）傳入一個排序準則：預設的是 operator <。因此，本例容器內的所有元素以遞增方式排列。

```
sort (coll.begin(), coll.end());
```

排序後的容器元素如下排列：

```
1 2 3 4 5 6
```

再來便是演算法 find()。它在給定範圍中搜尋某個值。本例在整個容器內尋找第一個數值爲 3 的元素。

```
pos = find (coll.begin(), coll.end(),       // range
            3);                             // value
```

如果 find()成功了，便回傳一個迭代器，指向目標元素。如果失敗，回傳一個「逾尾（past-the-end）」迭代器，亦即 find()所接受的第二引數。本例在第三個元素位置上發現數值 3，因此完成後 pos 指向 coll 的第三個位置。

本例所展示的最後一個演算法是 reverse()，將區間內的元素反轉放置：

```
reverse (pos, coll.end());
```

於是第三個至最後一個元素之間的所有元素都被反轉置放。整個程式輸出如下：

```
min: 1
max: 6
1 2 6 5 4 3
```

5.4.1 區間（Ranges）

所有演算法都用來處理一個或多個區間內的元素。這樣的區間可以（但非強行要求）涵蓋容器內的全部元素。因此，爲了得以操作容器元素的某個子集，我們必須將區間首尾當做兩個引數（arguments）傳給演算法，而不是一口氣把整個容器傳遞進去。

這樣的介面靈活又危險。呼叫者必須確保經由兩引數定義出來的區間是有效的（valid）。所謂有效就是，從起點出發，逐一前進，能夠到達終點。也就是說，程式員自己必須確保兩個迭代器隸屬同一容器，而且前後放置正確。否則結果難料，可能會引起無限迴圈，也可能會存取到記憶體禁區。就此點而言，迭代器就像一般指標一樣危險。不過請注意，所謂「結果難料」（或說行爲未有定義，*undefined behavior*）意味任何 STL 實作品均可自由選擇合適的方式來處理此類錯誤。稍後你會發現，確保區間的有效性並不像聽起來那麼簡單。與此相關的一些細節請參見 5.11 節, p136。所有演算法處理的都是半開區間（half-open ranges） —含括起始元素位置但不含括結尾元素位置。傳統的數學表示方式爲：

```
[begin, end)
```

或

```
[begin, end[
```

本書採用第一種表示法。

半開區間的優點已於 p84 介紹過（主要是單純，可避免對空群集做另外特殊處理）。當然，金無足赤，世上亦沒有完美的設計。請看下面的例子：

```
// stl/find1.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;
    list<int>::iterator pos;

    // insert elements from 20 to 40
    for (int i=20; i<=40; ++i)
         coll.push_back(i);
    }
```

```
    /* find position of element with value 3
     * - there is none, so pos gets coll.end()
     */
    pos = find (coll.begin(), coll.end(),   // range
                3);                         // value

    /* reverse the order of elements between found element and the
    * end - because pos is coll.end() it reverses an empty range
    */
    reverse (pos, coll.end());

    // find positions of values 25 and 35
    list<int>::iterator pos25, pos35;
    pos25 = find (coll.begin(), coll.end(),      // range
                  25);                           // value
    pos35 = find (coll.begin(), coll.end(),      // range
                  35);                           // value

    /* print the maximum of the corresponding range
    * - note: including pos25 but excluding pos35
    */
    cout << "max: " << *max_element (pos25, pos35) << endl;

    // process the elements including the last position
    cout << "max: " << *max_element (pos25, ++pos35) << endl;
}
```

本例首先以 20 至 40 的整數做爲容器初值。當搜尋元素值 3 的任務失敗後，`find()` 傳回區間的結束位置（本例爲 `coll.end()`）並指派給 `pos`。以此 `pos` 作爲稍後呼叫 `reverse()`時的區間起點，純粹是空擺架子，因爲其結果相當於：

```
reverse (coll.end(), coll.end());
```

這其實就是在逆轉一個空區間，當然毫無效果了（亦即所謂的 "no-op"）。

如果使用 `find()`來獲取某個子集的第一個和最後一個元素，你必須考慮一點：半開區間並不包含最後一個元素。所以上述例子第一次呼叫 `max_element()`：

```
max_element (pos25, pos35)
```

傳回的是 34，而不是 35：

```
max: 34
```

爲了處理最後一個元素，你必須把該元素的下一個位置傳遞給演算法：

```
max_element (pos25, ++pos35)
```

這樣才能得到正確的結果：

```
max: 35
```

注意，本例使用的是 list 容器，所以你只能以 ++ 取得 pos35 的下一個位置。如果面對的是 vectors 或 deques 的隨機存取迭代器（*random access iterator*s），你可以寫 pos35 + 1。這是因爲隨機存取迭代器允許「迭代器算術運算」（參見 p93, 5.3.2 節，p255, 7.2.5 節）。

當然，你可以使用 pos25 和 pos35 來搜尋其間的任何東西。記住，爲了讓搜尋動作及於 pos35，必須將元素 35 的下一位置傳入，例如：

```
// increment pos35 to search with its value included
++pos35;
pos30 = find(pos25,pos35,    // range
             30);            // value
if (pos30 == pos35) {
     cout << "30 is NOT in the subrange" << endl;
}
else {
     cout << "30 is in the subrange" << endl;
}
```

本節中的所有例子都可以正常運作，但那完全是因爲你很清楚 pos25 一定在 pos35 之前。否則，[pos25;pos35) 就不是個有效區間。如果你對於「哪個元素在前，哪個元素在後」心中沒譜兒，事情可就麻煩了，說不定會導致未定義行爲。

現在假設你並不知道元素 25 和元素 35 的前後關係，甚至連它們是否存在也心存疑慮。如果你手上用的是隨機存取迭代器（*random access iterator*s），你可以使用 operator< 進行檢查：

```
if (pos25 < pos35) {
    // only [pos25; pos35) is valid
    ...
}
else if (pos35 < pos25) {
    // only [pos35; pos25) is valid
    ...
}
else {
    // both are equal, so both must be end()
    ...
}
```

如果你手上的並非隨機存取迭代器，那還真的沒什麼直截了當的辦法可確定哪個迭代器在前。你只能在「起點和某個迭代器」之間，以及「該迭代器和終點」之間，尋找另外那個迭代器。此時你的解決方法需要一些變化：不是一口氣在整個區間中搜尋兩個值，而是了解哪個值先找到，哪個值後找到。例如：

```
pos25 = find (coll.begin(), coll.end(),     // range
              25);                          // value
pos35 = find (coll.begin(), pos25,          // range
              35);                          // value
if (pos35 != pos25) {
   /* pos35 is in front of pos25
   * so, only [pos35; pos25) is valid
   */
   ...
}
else {
   pos35 = find (pos25, coll.end(),         // range
                 35);                       // value
   if (pos35 != pos25) {
      /* pos25 is in front of pos35
      * so, only [pos25; pos35) is valid
      */
      ...
    }
   else {
      // both are equal, so both must be end()
      ...
   }
}
```

和前例不同的是，本例並非在 coll 的整個區間內搜尋 35，而是先在起點和 pos25 之間尋找，如果一無所獲，再在 pos25 之後的區間尋找。其結果當然使你得以完全掌握哪個位置在前面、哪個子區間有效。

這麼做並不是很有效率。當然還有其他高招，可以直接找到 25 或 35 第一次出現的位置，不過那就需要用到目前還未介紹的一些 STL 技術了：

```
pos = find_if (coll.begin(), coll.end(),              // range
               compose_f_gx_hx(logical_or<bool>(),    // criterion
                               bind2nd(equal_to<int>(),25),
                               bind2nd(equal_to<int>(),35)));
```

```
switch (*pos) {
   case 25:
      // element with value 25 comes first
      pos25 = pos;
      pos35 = find (++pos, coll.end(),     // range
                    35);                   // value
      ...
      break;
   case 35:
      // element with value 35 comes first
      pos35 = pos;
      pos25 = find (++pos, coll.end(),     // range
                    25);                   // value
      ...
      break;
   default:
      // no element with value 25 or 35 found
      ...
      break;
}
```

這裡使用了一個特別的運算式作爲搜尋規則，其目的是找到數值 25 或數值 35 第一次出現的位置。這個運算式由好幾個預先定義的仿函式（functors，或名 function objects）組成，我將在 5.9.2 節, p131 和 8.2 節, p305 介紹所有預先定義的仿函式。`compose_f_gx_hx` 是個靈巧的輔助型仿函式，我將在 8.3.1 節, p316 介紹它。

5.4.2 處理多個區間

有數個演算法可以（或說需要）同時處理多個區間。通常你必須設定第一個區間的起點和終點，至於其他區間，你只需設定起點即可，終點通常可由第一區間的元素數量推導出來。下面例子中，`equal()` 從頭開始逐一比較 `coll1` 和 `coll2` 的所有元素：

```
if (equal (coll1.begin(), coll1.end(),
           coll2.begin())) {
   ...
}
```

因此，`coll2` 之中參與比較的元素數量，間接取決於 `coll1` 內的元素數量。

這使我們導出一個重要心得：如果某個演算法是來處理多個區間，那麼當你呼叫它時，務必確保第二（以及其他）區間所擁有的元素個數，至少和第一區間的元素個數相同。特別是，執行塗寫動作時，務必確保目標區間（destination ranges）夠大。

考慮下面這個程式：

```
// stl/copy1.cpp

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll1.push_back(i);
    }

    // RUNTIME ERROR:
    // - overwrites nonexisting elements in the destination
    copy (coll1.begin(), coll1.end(),  // source
          coll2.begin());              // destination
    ...
}
```

這裏呼叫了 copy() 演算法，將第一區間內的全部元素拷貝至目標區間。如上所述，第一區間的起點和終點都已指定，第二區間只指出起點。然而，由於該演算法執行的是覆寫動作（overwrites）而非安插動作（inserts），所以目標區間必須擁有足夠的元素來被覆寫，否則就會像這個例子一樣，導致未定義的行爲。如果目標區間內沒有足夠的元素供覆寫，通常意味你會覆寫 coll2.end() 之後的任何東西，幸運的話你的程式立即崩潰 — 這起碼還能讓你知道出錯了。你可以強制自己獲得這種幸運：使用 STL 安全版本。在這個安全版本中，所有未定義的行爲都會被導向一個錯誤處理程序（error handling procedure）。請參考 5.11.1 節，p138。

要想避免上述錯誤，你可以 (1) 確認目標區間內有足夠的元素空間，或是 (2) 採用 *insert iterator*s。*Insert iterator*s 將在 5.5.1 節, p104 介紹。我首先解釋如何修改目標區間，俾使它有足夠的空間。

要想讓目標區間夠大，你要不一開始就給它一個正確大小，要不就明白地改變其大小。這兩個辦法都只適用於序列式容器（vectors, deques, lists）。關聯式容器根本不會有此問題，因爲關聯式容器不可能被當作覆寫式演算法的操作目標（原因見 5.6.2 節，p115）。以下例子展示如何增加容器的大小：

```
// stl/copy2.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll1.push_back(i);
    }

    // resize destination to have enough room for the
    // overwriting algorithm
    coll2.resize (coll1.size());

    /* copy elements from first into second collection
    * - overwrites existing elements in destination
    */
    copy (coll1.begin(), coll1.end(),  // source
          coll2.begin());              // destination

    /* create third collection with enough room
    * - initial size is passed as parameter
    */
    deque<int> coll3(coll1.size());
```

```
    // copy elements from first into third collection
    copy (coll1.begin(), coll1.end(),       // source
          coll3.begin());                   // destination
}
```

在這裡，resize()的作用是改變 coll2 的元素個數：

```
coll2.resize (coll1.size());
```

coll3 則是在初始化時就指明要有足夠空間，以容納 coll1 中的全部元素：

```
deque<int> coll3(coll1.size());
```

注意，這兩種方法都會產出新元素並賦予初值。這些元素由 *default* 建構式初始化，沒有任何引數。你可以傳遞額外的引數給建構式和 resize()，這樣就可以按你的意願將新元素初始化。

5.5 迭代器之配接器（Iterator Adapters）

迭代器（Iterators）是一個純粹抽象概念：任何東西，只要其行爲類似迭代器，它就是一個迭代器。因此，你可以撰寫一些類別（classes），具備迭代器介面，但有著各不相同的行爲。C++ 標準程式庫提供了數個預先定義的特殊迭代器，亦即所謂迭代器配接器（iterator adapters）。它們不僅是輔助性質而已，它們賦予整個迭代器抽象概念更強大的能力。

以下數小節簡介三種迭代器配接器（iterator adapters）：

1. *Insert iterator*s（安插型迭代器）
2. *Stream iterator*s（串流迭代器）
3. *Reverse iterator*s（逆向迭代器）

第 7.4 節, p264 會對它們做更詳實的講解。

5.5.1 Insert Iterators（安插型迭代器）

迭代器配接器的第一個例子是 *insert iterator*s，或稱爲 *inserter*s。*Inserter*s 可以使演算法以安插（insert）方式而非覆寫（overwrite）方式運作。使用它，可以解決演算法的「目標空間不足」問題。是的，它會促使目標區間的大小按需要成長。

*Insert iterator*s 內部將介面做了新的定義：

- 如果你對某個元素設值（assign），會引發「對其所屬群集的安插（insert）動作」。至於插入位置是在容器的最前或最後，或是於某特定位置上，端視三種不同的 *insert iterator*s 而定。
- 「單步前進（step forward）」不會造成任何動靜（是一個 no-op）。

現在請看下面這個例子：

```
// stl/copy3.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll1;

   // insert elements from 1 to 9 into the first collection
   for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
   }

   // copy the elements of coll1 into coll2 by appending them
   vector<int> coll2;
   copy (coll1.begin(), coll1.end(),       // source
         back_inserter(coll2));            // destination

   // copy the elements of coll1 into coll3 by inserting them at the
   // front - reverses the order of the elements
   deque<int> coll3;
   copy (coll1.begin(), coll1.end(),       // source
         front_inserter(coll3));           // destination

   // copy elements of coll1 into coll4
   // - only inserter that works for associative collections
   set<int> coll4;
   copy (coll1.begin(), coll1.end(),       // source
         inserter(coll4,coll4.begin()));   // destination
}
```

此例運用了三種預先定義的 *insert iterator*s：

1. **Back inserters**（安插於容器最尾端）
   *Back inserter*s 的內部呼叫 `push_back()`，在容器尾端插入元素（此即「尾附」動作）。以下述句完成之後，`coll1` 的所有元素都會被附加到 `coll2` 中：
   ```
   copy (coll1.begin(), coll1.end(),  // source
         back_inserter(coll2));       // destination
   ```
   當然，只有在提供有 `push_back()`成員函式的容器中，*back inserter*s 才能派上用場。在 C++ 標準程式庫中，這樣的容器有三：vector, deque, list。

2. **Front inserters**（安插於容器最前端）
   *Front inserter*s 的內部呼叫 `push_front()`，將元素安插於容器最前端。以下述句將 `coll1` 的所有元素插入 `coll3`：
   ```
   copy (coll1.begin(), coll1.end(),  // source
         front_inserter(coll3));      // destination
   ```
   注意，這種動作逆轉了被安插元素的次序。如果你先安插 1，再向前安插 2，那麼 1 會排列在 2 的後面。

   *Front inserter*s 只能用於提供有 `push_front()`成員函式的容器，在標準程式庫中，這樣的容器是 deque 和 list。

3. **General inserters**（一般性安插器）
   這種一般性的 inserters，簡稱就叫 *inserter*s，它的作用是將元素插入「初始化時接受之第二引數」所指位置的前方。*Inserter*s 內部呼叫成員函式 `insert()`，並以新值和新位置做爲引數。所有 STL 容器都提供有 `insert()`成員函式，因此，這是唯一可用於關聯式容器身上的一種預先定義好的 *inserter*。

   等等，我不是說過，在關聯式容器身上安插新元素時，不能指定其位置嗎？它們的位置是由它們的值決定的啊！好，我解釋一下，很簡單：在關聯式容器中，你所給的位置只是一個提示，幫助它確定從什麼地方開始搜尋正確的安插位置。如果提示不正確，效率上的表現會比「沒有提示」更糟糕。7.5.2 節, p288 介紹了一個用戶自定的 *inserter*，對關聯式容器特別有用。

表 5.1 列出 *insert iterator*s 的功能。7.4.2 節, p271 還會介紹更多細節。

算式（expression）	Inserter 種類
`back_inserter(container)`	使用 `push_back()`在容器尾端安插元素，元素排列次序和安插次序相同。
`front_inserter(container)`	使用 `push_front()`在容器前端安插元素，元素排列次序和安插次序相反。
`inserter(container, pos)`	使用 `insert()`在 `pos` 位置上安插元素，元素排列次序和安插次序相同。

表 5.1 預先定義的三種 *Insert Iterator*s

5.5.2 Stream Iterators（串流迭代器）

另一種非常有用的迭代器配接器（iterator adapter）是 *stream iterator*，這是一種用來讀寫 stream[8]的迭代器。它們提供了必要的抽象性，使得來自鍵盤的輸入像是個群集（collection），你能夠從中讀取內容。同樣道理，你也可以把一個演算法的輸出結果重新導向到某個檔案或螢幕上。

下面是展示 STL 威力的一個典型例子。和一般 C 或 C++ 程式相比，本例僅用數條述句，就完成了大量複雜工作：

```
// stl/ioiter1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> coll;

    /* read all words from the standard input
    * - source: all strings until end-of-file (or error)
    * - destination: coll (inserting)
    */
    copy (istream_iterator<string>(cin),    // start of source
          istream_iterator<string>(),       // end of source
          back_inserter(coll));             // destination

    // sort elements
    sort (coll.begin(), coll.end());

    /* print all elements without duplicates
    * - source: coll
    * - destination: standard output (with newline between elements)
    */
    unique_copy (coll.begin(), coll.end(),              // source
                 ostream_iterator<string>(cout,"\n")); // destination
}
```

[8] Stream（串流）是一個用來表現 I/O 通道的物件（詳見第 13 章）。

這個程式只用三個述句就完成一系列工作：從標準輸入裝置讀取所有輸入文字、排序、將它們列印於螢幕。讓我們逐一思考這三個述句。下面這個述句：

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

用到兩個 *input stream iterator*s：

1. `istream_iterator<string>(cin)`
這會產生一個可從「標準輸入串流（standard input stream）`cin`」讀取資料的 ***stream iterator***[9]。其中的 template 引數 `string` 表示，這個 ***stream iterator*** 專司讀取該種型別的元素（`string` 型別將在第 11 章介紹）。這些元素透過一般的 `operator>>` 被讀取進來。因此每當演算法企圖處理下一個元素時，***istream iterator*** 就會將這種企圖轉化爲以下行動：

```
cin >> string
```

針對 string 而執行的 ***input*** 運算子通常讀取以空白分隔的文字（參見 p492），因此上述演算法的行爲將是「逐詞讀取（word-by-word）」。

2. `istream_iterator<string>()`
呼叫 ***istream iterator***s 的 ***default*** 建構式，產生一個代表「串流結束符號」（end-of-stream）的迭代器，它代表的意義是：你不能再從中讀取任何東西。

只要不斷逐一前進的那個第一引數不同於第二引數，演算法 `copy()` 就持續動作。這裏的 end-of-stream 迭代器正是作爲區間終點之用，因此這個演算法便從 `cin` 讀取所有 strings，直到讀無可讀爲止（可能是因爲到達了 end-of-stream，也可能是因爲讀入過程發生錯誤）。總而言之，演算法的資料來源是「來自 `cin` 的所有文字」。在 ***back inserter*** 的協助下，這些文字被拷貝並插入 `coll` 中。

接下來的 `sort()` 演算法對所有元素進行排序：

```
sort (coll.begin(), coll.end());
```

最後，下面這個述句：

```
unique_copy (coll.begin(), coll.end(),
             ostream_iterator<string>(cout,"\n"));
```

將其中所有元素拷貝到目的端 `cout`。處理過程中演算法 `unique_copy()` 會消除毗鄰的重複值。其中的運算式：

```
ostream_iterator<string>(cout,"\n")
```

---

[9] 在某些老舊系統中，你必須使用 `ptrdiff_t` 作爲第二個模板引數，才能產生出一個 *istream iterator*（參見 7.4.3 節, p280）。

會產生一個 *output stream iterator*，透過 operator<< 向 cout 寫入 strings。cout 之後的第二引數（可有可無）被用來作爲元素之間的分隔符號。本例指定爲一個換行符號，因此每個元素都被列印於獨立的一行。

這個程式內的所有組件都是 templates，所以你可以輕易改變程式，對其他型別如整數或更複雜的物件進行排序。7.4.3 節, p277 對於 *iostream iterator*s 進行了更詳細的介紹，並附帶更多實例。

本例使用一個宣告和三個述句，對來自標準輸入裝置的所有文字（單詞）進行排序。你還可以更進一步，只用一個宣告和一個述句就搞定一切。詳見 p228。

5.5.3 Reverse Iterators（逆向迭代器）

第三種預先定義的迭代器配接器（iterator adapters）就是 *reverse iterator*s，此物像是倒轉筋脈似地以逆向方式進行所有操作。它將 *increment*（遞增）運算轉換爲 *decrement*（遞減）運算，反之亦然。所有容器都可以透過成員函式 rbegin() 和 rend() 產生出 *reverse iterator*s。例如：

```
// stl/riter1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll.push_back(i);
    }

    // print all element in reverse order
    copy (coll.rbegin(), coll.rend(),              // source
          ostream_iterator<int>(cout," "));        // destination
    cout << endl;
}
```

其中的運算式：

```
coll.rbegin()
```

傳回一個由 coll 定義的 *reverse iterator*。這個迭代器可作爲「對群集 coll 的元素逆向走訪」的起點。它指向群集的結尾位置（也就是最後元素的下一位置）。因此，運算式：

```
*coll.rbegin()
```

傳回的是最後一個元素的值。

對應地，運算式：

```
coll.rend()
```

傳回的 *reverse iterator*，可作爲「對群集 coll 的元素逆向走訪」的終點。它也是指向「逾尾」（past-the-end）位置，只不過方向相反，指的是容器內第一個元素的前一個位置。

以下運算式沒有定義：

```
*coll.rend()
```

同樣情況，以下運算式也沒有定義：

```
*coll.end()
```

注意，當某個位置上並無合法元素時，永遠不要使用 operator* 或 operator->。

如果採用 *reverse iterator*s，所有演算法便可以不需特殊處理就以相反方向操作容器，這自然是美事一樁。使用 operator++ 前進至下一元素，被轉化爲使用 operator-- 後退至前一元素。本例中的 copy()，「從尾到頭」地走訪所有 coll 元素。程式輸出如下：

```
9 8 7 6 5 4 3 2 1
```

你可以將一般迭代器轉換爲 *reverse iterator*s，反之亦可。然而，對於具體某個迭代器而言，這樣的轉換會改變其所指對象。這些細節在第 7.4.1 節, p264 介紹。

5.6 更易型演算法（Manipulating Algorithms）

譯註：根據實質意義，我不把 manipulating algorithms 譯爲「操控型」演算法。manipulating algorithms 是指會「刪除或重排或修改元素」的演算法，見 p115。該頁亦出現另一個相同意義的術語：modifying algorithms。有些書籍（例如 *Generic Programming and the STL*）採用 mutating algorithms 一詞。爲此，我將這些相同意義的術語都譯爲「更易型」或「變動型」演算法。我亦曾在某些書中採用「質變演算法」一詞。

某些演算法會變更目標區間的內容，甚至會刪除元素。一旦這種情況出現，請務必注意幾個特殊問題。本節將對此做出解釋。它們確實令人訝異，並體現了 STL「爲了將容器和演算法分離，以獲取彈性」而付出的代價。

5.6.1 移除（Removing）元素

演算法 `remove()` 自某個區間刪除元素。然而如果你用它來刪除容器中的所有元素，其行爲肯定會讓你吃驚。例如：

```
// stl/remove1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
         coll.push_front(i);
         coll.push_back(i);
    }

    // print all elements of the collection
    cout << "pre: ";
    copy (coll.begin(), coll.end(),                 // source
          ostream_iterator<int>(cout," "));         // destination
    cout << endl;

    // remove all elements with value 3
    remove (coll.begin(), coll.end(),           // range
            3);                                 // value
```

```
    // print all elements of the collection
    cout << "post: ";
    copy (coll.begin(), coll.end(),          // source
          ostream_iterator<int>(cout," ")); // destination
    cout << endl;
}
```

缺乏 STL 深層認識的人，看了這程式，必然認爲所有數值爲 3 的元素都會從群集中被移除。然而，程式的輸出卻是這樣：

```
pre: 6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6
```

啊呀，`remove()`並沒有改變群集中的元素數量。`end()`傳回的還是當初那個終點，`size()`傳回的還是當初那個大小。不過某些事情還是有了變化：元素的次序改變了，有些元素被刪除掉了。數值爲 3 的元素被其後的元素覆蓋了（圖 5.7）。至於群集尾端那些未被覆蓋的元素，原封不動 — 但是從邏輯角度來說，那些元素已經不屬於這個群集了。

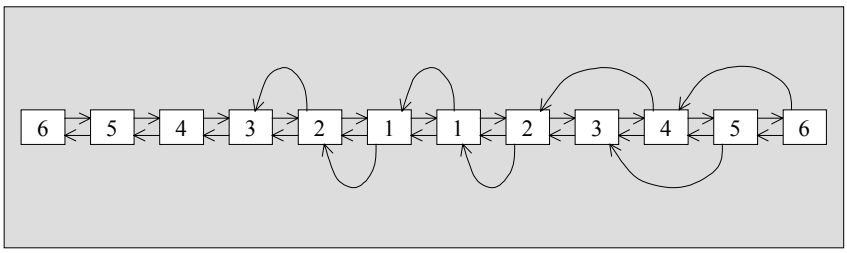


圖 5.7 `remove()` 如何運作

事實上，這個演算法傳回了一個新的終點。你可以利用該終點獲得新區間、縮減後的容器大小，或是獲得被刪除元素的個數。看看下面這個改進版本：

```
// stl/remove2.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
```

```
    for (int i=1; i<=6; ++i) {
         coll.push_front(i);
         coll.push_back(i);
    }

    // print all elements of the collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements with value 3
    // - retain new end
    list<int>::iterator end = remove (coll.begin(), coll.end(),
                                      3);

    // print resulting elements of the collection
    copy (coll.begin(), end,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // print number of resulting elements
    cout << "number of removed elements: "
         << distance(end,coll.end()) << endl;

    // remove "removed" elements
    coll.erase (end, coll.end());

    // print all elements of the modified collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

在這個版本中，`remove()`的回傳值被設定給 `end` 迭代器：

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                  3);
```

這個 `end` 正是「被修改之群集」經過元素移除動作後，邏輯上的新終點。接下來你便可以拿它當作新的終點使用：

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout," "));
```

另一種可能用法是，藉由測定群集之「邏輯」終點和實際終點間的距離，獲得「被刪除元素」的數量：

```
cout << "number of removed elements: "
     << distance(end,coll.end()) << endl;
```

在這裡，針對迭代器而設計的輔助函式 `distance()`發揮了作用。它的功用是傳回兩個迭代器之間的距離。如果這兩個迭代器都是隨機存取迭代器（*random access iterators*），你可以使用 `operator-` 直接計算其距離。不過本例所用的容器是 list，只提供雙向迭代器（*bidirectional iterators*）。關於 `distance()`的細節，詳見 7.3.2 節, p261[10]。

如果真想把那些被刪除的元素斬草除根，你必須呼叫該容器的相應成員函式。容器所提供的成員函式 `erase()`，正適用於此目的。`erase()`可以刪除「引數所指示之區間」內的全部元素：

```
coll.erase (end, coll.end());
```

下面是整個程式的完整輸出：

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

如果你需要以單一述句來刪除元素，可以如此這般：

```
coll.erase (remove(coll.begin(),coll.end(),
                   3),
            coll.end());
```

爲何演算法不自己呼叫 `erase()`呢？哎，這個問題正好點出 STL 爲了獲取彈性而付出的代價。透過「以迭代器爲介面」，STL 將資料結構和演算法分離開來。然而，迭代器只不過是「容器中某一位置」的抽象概念而已。一般來說，迭代器對自己所屬的容器一無所知。任何「以迭代器訪問容器元素」的演算法，都不得（無法）透過迭代器呼叫容器類別所提供的任何成員函式。

這個設計導致一個重要結果：演算法的操作對象不一定得是「容器內的全部元素」所形成的區間，而可以是那些元素的子集。甚至演算法可運作於一個「並未提供成員函式 `erase()`」的容器上（array 就是個例子）。所以，爲了達成演算法的最大彈性，不要求「迭代器必須能解其容器細節」還是很有道理的。

---

[10] `distance()`的定義有些變化。在 STL 舊式版本中，爲了使用它，你必須含入 `distance.hpp`，見 p263。

注意，通常並無必要刪除那些「已被移除」的元素。通常，以邏輯終點來取代容器的實際終點，就足以應對。你可以以這個邏輯終點搭配任何演算法演出。

5.6.2 更易型演算法和關聯式容器

更易型演算法（指那些會移除 *remove*、重排 *resort*、修改 *modify* 元素的演算法）用於關聯式容器身上會出問題。關聯式容器不能被當作操作目標，原因很簡單：如果更易型演算法用於關聯式容器身上，會改變某位置上的值，進而破壞其已序（*sorted*）特性，那就推翻了關聯式容器的基本原則：容器內的元素總是根據某個排序準則自動排序。因此，爲了保證這個原則，關聯式容器的所有迭代器均被宣告爲指向常量（不變量）。如果你更易關聯式容器中的元素，會導致編譯錯誤[11]。

注意，這使你無法在關聯式容器身上運用移除性（*removing*）演算法，因爲這類演算法實際上悄悄更易了元素：「被移除元素」被其後的「未被移除元素」覆蓋。

現在問題來了，如何從關聯容器中刪除元素？唔，很簡單：呼叫它們的成員函式！每一種關聯式容器都提供用以移除元素的成員函式。例如你可以呼叫 `erase()` 來移除元素：

```
// stl/remove3.cpp

#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
    set<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll.insert(i);
    }

    // print all elements of the collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
```

[11] 糟糕的是，有些系統提供的錯誤處理能力令人不敢恭維；面對錯誤，你無法找出原因。有些編譯器甚至連出錯的源碼都不列出來。希望這種狀況在不久的將來獲得改善。

```
    cout << endl;

    /* Remove all elements with value 3
    * - algorithm remove() does not work
    * - instead member function erase() works
    */
    int num = coll.erase(3);

    // print number of removed elements
    cout << "number of removed elements: " << num << endl;

    // print all elements of the modified collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

注意，容器類別提供了多個不同的 `erase()`成員函式。其中一種形式是以「待刪除之元素值」爲唯一引數，它會傳回被刪除的元素個數（第 242 頁）— 當然，在禁止元素重複的容器中（例如 sets 和 maps），其傳回值永遠只能是 0 或 1。

本節範例程式輸出如下：

```
1 2 3 4 5 6 7 8 9
number of removed elements: 1
1 2 4 5 6 7 8 9
```

5.6.3 演算法 vs. 成員函式

就算我們符合種種條件，得以使用某個演算法，那也未必就一定是好。容器本身可能提供功能相似而性能更佳的成員函式。

一個極佳例子便是對 list 的元素呼叫 `remove()`。演算法本身並不知道它工作於 list 身上，因此它在任何容器中都一樣，做些四平八穩的工作：改變元素值，從而重新排列元素。如果它移除第一個元素，後面所有元素就會分別被設給各自的前一個元素。這就違反了 lists 的主要優點 — 藉由修改鏈結（links）而非實值（values）來安插、移動、移除元素。

爲了避免這麼糟糕的表現，list 針對所有「更易型」演算法提供了一些對應的成員函式。是的，如果你使用 list，你就應該使用這些成員函式。此外請注意，這些成員函式真的移除了「被移除」的元素（譯註：而不像先前所說只是某種搬移而已），一如下例所示：

```
// stl/remove4.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
         coll.push_front(i);
         coll.push_back(i);
    }

    // remove all elements with value 3
    // - poor performance
    coll.erase (remove(coll.begin(),coll.end(),
                       3),
                coll.end());

    // remove all elements with value 4
    // - good performance
    coll.remove (4);
}
```

如果高效率是你的最高目標，你應該永遠優先選用成員函式。問題是你必須先知道，某個容器確實存在有效率上明顯突出的成員函式。面對 list 卻使用 `remove()` 演算法，你決不會收到任何警告訊息或錯誤通告。然而如果你決定使用成員函式，一旦換用另一種容器，就不得不更動程式碼。第 9 章的演算法參考章節中，如果某個成員函式的性能優於某個演算法，我會明白指出。

5.7 使用者自定之泛型函式（User-Defined Generic Functions）

STL 乃是一個可擴展的框架（framework）。這意味你可以撰寫自己的函式和演算法，處理群集內的元素。當然，這些操作函式本身也可以是泛型的（generic）。

爲了在這些操作之中宣告有效的迭代器，你必須使用容器提供的型別，因爲每一種容器都有自己的迭代器。爲了讓我們方便寫出真正的泛型函式，每一種容器都提供了一些內部的型別定義。請看下面的例子：

```
// stl/print.hpp

#include <iostream>

/* PRINT_ELEMENTS()
* - prints optional C-string optcstr followed by
* - all elements of the collection coll
* - separated by spaces
*/
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
         std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

本例定義出一個泛型函式，可列印一個字串（也可以不指定），然後列印容器的全部元素。以下宣告式：

```
typename T::const_iterator pos;
```

其中的 pos 被宣告爲「傳入之容器型別」內的迭代器型別，關鍵字 typename 在此不可或缺，用以表明 const_iterator 是型別 T 所定義的一個型別，而不是一個型別爲 T 的值（請見 p11 對 typename 的介紹）。

除了 iterator 和 const_iterator，容器還提供了其他（內部定義的）型別，幫助你寫出泛型函式。例如它提供了元素型別（譯註：即所謂 *value type*），以便在元素暫時拷貝場合中派上用場。詳見 7.5.1 節, p285。

PRINT_ELEMENTS 的第二引數是個可有可無的前綴字，用來在列印時放於所有元素之前。你可以這樣使用 PRINT_ELEMENTS()：

```
PRINT_ELEMENTS (coll, "all elements: ");
```

我之所以介紹這個函式，因爲本書剩餘部份會大量運用它來列印容器的所有元素。

5.8 以函式做爲演算法的引數

一些演算法可以接受用戶定義的輔助性函式，由此提高其彈性和能力。這些函式將在演算法內部被呼叫。

5.8.1 「以函式做爲演算法的引數」實例示範

最簡單的例子莫過於 `for_each()`演算法了。它針對區間內的每一個元素，呼叫一個由用戶指定的函式。下面是個例子：

```
// stl/foreach1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// function that prints the passed argument
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll.push_back(i);
    }

    // print all elements
    for_each (coll.begin(), coll.end(),     // range
              print);                       // operation
    cout << endl;
}
```

這裏的 `for_each()`函式針對`[coll.begin(), coll.end())`區間內的每個元素呼叫`print()`函式。輸出如下：

```
1 2 3 4 5 6 7 8 9
```

演算法以數種態度來面對這些輔助函式：有的視之爲可有可無，有的視之爲必要。你可以利用它們來指定搜尋準則、排序準則、或定義某種操作，以便將某個容器內的元素轉換至另一個容器。

下面是個運用實例：

```
// stl/transform1.cpp

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include "print.hpp"

int square (int value)
{
   return value*value;
}

int main()
{
   std::set<int> coll1;
   std::vector<int> coll2;

   // insert elements from 1 to 9 into coll1
   for (int i=1; i<=9; ++i) {
        coll1.insert(i);
   }
   PRINT_ELEMENTS(coll1,"initialized: ");

   // transform each element from coll1 to coll2
   // - square transformed values
   std::transform (coll1.begin(),coll1.end(),   // source
                   std::back_inserter(coll2),   // destination
                   square);                     // operation

   PRINT_ELEMENTS(coll2,"squared: ");
}
```

此例之中，square()的作用是將 coll1 內的每一個元素予以平方運算，然後轉移到 coll2（圖 5.8）。輸出如下：

```
initialized: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
```

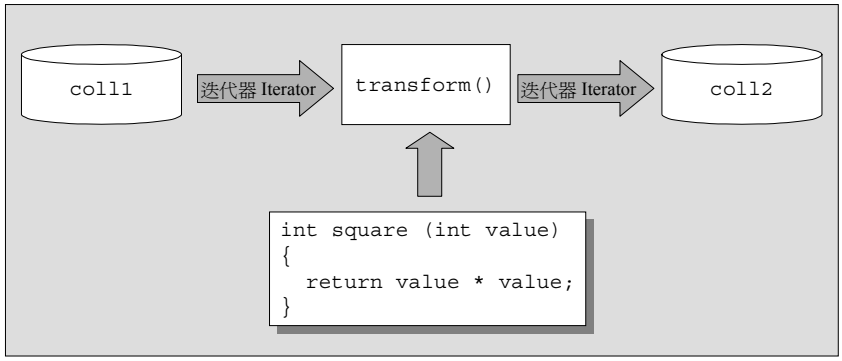


圖 5.8 transform() 的運作方式

5.8.2 判斷式（Predicates）

演算法有一種特殊的輔助函式叫做 predicates（判斷式）。所謂 predicates，就是回傳布林值（boolean）的函式。它們通常被用來指定排序準則和搜尋準則。Predicates 可能有一個或兩個運算元，視具體情形而定。注意，並非任何傳回布林值的一元函式或二元函式就是合法的 predicate。STL 要求，面對相同的值，predicates 必須得出相同的結果。這條戒律將那些「被呼叫時，會改變自己內部狀態」的函式清除出場。細節請見 8.1.4 節, p302。

Unary Predicates（一元判斷式）

Unary predicates 會檢查唯一引數的某項特性。典型例子是像下面這樣的函式，用來搜尋第一個質數：

```
// stl/prime1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <cstdlib>      // for abs()
using namespace std;
```

```
// predicate, which returns whether an integer is a prime number
bool isPrime (int number)
{
   // ignore negative sign
   number = abs(number);

   // 0 and 1 are prime numbers
   if (number == 0 || number == 1) {
      return true;
   }

   // find divisor that divides without a remainder
   int divisor;
   for (divisor = number/2; number%divisor != 0; --divisor) {
        ;
   }

   // if no divisor greater than 1 is found, it is a prime number
   return divisor == 1;
}

int main()
{
   list<int> coll;

   // insert elements from 24 to 30
   for (int i=24; i<=30; ++i) {
        coll.push_back(i);
   }

   // search for prime number
   list<int>::iterator pos;
   pos = find_if (coll.begin(), coll.end(), // range
                  isPrime); // predicate
   if (pos != coll.end()) {
      // found
      cout << *pos << " is first prime number found" << endl;
   }
   else {
```

```
        // not found
        cout << "no prime number found" << endl;
    }
}
```

在這個例子中，find_if()演算法在給定區間內尋找使「被傳入之一元判斷式（unary predicate）」運算結果爲 true 的第一個元素。本例中的 predicate 是 isPrime()函式，它會檢查某數是否爲質數。透過它，這個演算法可以傳回給定區間內的第一個質數。如果沒有任何元素能夠匹配這個（質數）條件，find_if()演算法就傳回區間終點（也就是函式的第二引數）。本例中，24 到 30 之間確實存在一個質數，所以程式輸出：

```
29 is first prime number found
```

Binary Predicates（二元判斷式）

Binary predicates 的典型用途是，比較兩個引數的特定屬性。例如，爲了依照你自己的原則對元素排序，你必須以一個簡單的 predicate 形式提供這項原則。如果元素本身不支持 operator<，或如果你想使用不同的排序原則，這就派上用場了。

下面這個例子，根據每個人的姓名，對一組元素進行排序：

```
// stl/sort1.cpp

#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
using namespace std;

class Person {
  public:
    string firstname() const;
    string lastname() const;
    ...
};

/* binary function predicate:
* - returns whether a person is less than another person
*/
```

```
bool personSortCriterion (const Person& p1, const Person& p2)
{
   /* a person is less than another person
   * - if the last name is less
   * - if the last name is equal and the first name is less
   */
   return p1.lastname()<p2.lastname() ||
          (!(p2.lastname()<p1.lastname()) &&
          p1.firstname()<p2.firstname());
}

int main()
{
   deque<Person> coll;
   ...
   sort(coll.begin(),coll.end(), // range
   personSortCriterion);         // sort criterion
   ...
}
```

注意，你也可以使用仿函式（functor，或名 function object）來實作一個排序準則。這種作法的優點是，製作出來的準則將是一個型別（type），可用來做爲諸如「宣告一個 set，以某種型別爲排序準則」之類的事情。詳見 8.1.1 節, p294。

5.9 仿函式（Functors, Function Objects）

譯註：本書英文版通篇採用的術語是 **function object**，對應之譯名爲「函式物件」。此物在 STL 發展初期曾經名爲 **functor**，取其音義，我譯爲「仿函式」。考量 STL 六大組件之譯名整體性，以及「術語最好具備獨特性，且不與其他名詞混淆」的原則，再考慮上下文閱讀的順暢性，我認爲「仿函式」較「函式物件」爲佳。爲此，本中文版將 function object 全以 functor 取代，並譯爲「仿函式」。

傳遞給演算法的「函式型引數」（functional arguments），並不一定得是函式，可以是行爲類似函式的物件。這種物件稱爲 **function object**（函式物件），或稱 **functor**（仿函式）。當一般函式使不上勁時，你可以使用仿函式。STL 大量運用仿函式，也提供（預先定義）了一些很有用的仿函式。

5.9.1 什麼是仿函式

仿函式是泛型編程強大威力和純粹抽象概念的又一個例証。你可以說，任何東西，只要其行爲像函式，它就是個函式。因此如果你定義了一個物件，行爲像函式，它就可以被當作函式來用。

好，那麼，什麼才算是具備函式行爲（也就是行爲像個函式）？所謂函式行爲，是指可以「使用小括號傳遞引數，藉以呼叫某個東西」。例如：

```
function(arg1,arg2); // a function call
```

如果你指望物件也可以如此這般，就必須讓它們也有可能被「呼叫」 — 透過小括號的運用和引數的傳遞。沒錯，這是可能的（在 C++ 中，很少有什麼是不可能的）。你只需定義 operator()，並給予合適的參數型別：

```
class X {
public:
  // define ''function call'' operator
  return-value operator() (arguments) const;
  ...
};
```

現在，你可以把這個類別的物件當作函式來呼叫了：

```
X fo;
...
fo(arg1,arg2); // call operator () for function object fo
```

上述呼叫等同於：

```
fo.operator()(arg1,arg2); // call operator () for function object fo
```

下面是個完整例子，是先前 p119 範例的一個仿函式版本，其行爲和使用一般函式（非仿函式）完全相同：

```
// stl/foreach2.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// simple function object that prints the passed argument
class PrintInt {
  public:
    void operator() (int elem) const {
        cout << elem << ' ';
    }
};

int main()
{
  vector<int> coll;
```

```
  // insert elements from 1 to 9
  for (int i=1; i<=9; ++i) {
       coll.push_back(i);
  }

  // print all elements
  for_each (coll.begin(), coll.end(),  // range
            PrintInt());               // operation
  cout << endl;
}
```

PrintInt 所做的定義顯示，你可以對它的物件呼叫 operator()，並傳入一個 int 引數。至於述句：

```
for_each (coll.begin(), coll.end(),
          PrintInt());
```

其中的運算式：

```
PrintInt()
```

產生出此類別的一個臨時物件，當作 for_each() 演算法的一個引數。for_each() 演算法大致如下：

```
namespace std {
  template <class Iterator, class Operation>
  Operation for_each (Iterator act, Iterator end, Operation op)
  {
    while (act != end) {      // as long as not reached the end
       op(*act);              // - call op() for actual element
       ++act;                 // - move iterator to the next element
    }
    return op;
  }
}
```

for_each() 使用暫時物件 op（一個仿函式），針對每個元素呼叫 op(*act)。如果第三引數是個一般函式，就以*act 爲引數呼叫之。如果第三引數是個仿函式，則以*act 爲引數，呼叫仿函式 op 的 operator()。因此，本例之中，for_each() 呼叫：

```
PrintInt::operator()(*act)
```

你也許不以爲然，你也許認爲仿函式看起來怪異、令人討厭、甚或毫無意義。的確，它們帶來更復雜的程式碼，然而仿函式有其過人之處，比起一般函式，它們有以下優點：

1. 仿函式是 "smart functions"（精靈函式，智慧型函式）

「行爲類似指標」的物件，我們稱爲 "smart pointers"。「行爲類似函式」的物件呢？同樣道理，我們可以稱之爲 "smart functions"，因爲它們的能力可以超越 `operator()`。仿函式可擁有成員函式和成員變數，這意味仿函式擁有狀態 (state)。事實上，在同一時間裡，由某個仿函式所代表的單一函式，可能有不同的狀態。這在一般函式中是不可能的。另一個好處是，你可以在執行期（runtime）初始化它們 — 當然必須在它們被使用（被呼叫）之前。

2. 每個仿函式都有自己的型別

一般函式，唯有在它們的標記式（signatures）不同時，才算型別不同。而仿函式即使標記式相同，也可以有不同的型別。事實上，由仿函式定義的每一個函式行爲都有其自己的型別。這對於「利用 template 實現泛型程式編寫」乃是一個卓越貢獻，因爲如此一來，我們便可以將函式行爲當做 template 參數來運用。這使得不同型別的容器可以使用同類型的仿函式作爲排序準則。這可以確保你不會在排序準則不同的群集（collections）之間賦值、合併或比較。你甚至可以設計仿函式繼承體系（functors hierarchies），以此完成某些特別事情，例如在一個總體原則下確立某些特殊情況。

3. 仿函式通常比一般函式速度快

就 template 概念而言，由於更多細節在編譯期就已確定，所以通常可能進行更好的最佳化。所以，傳入一個仿函式（而非一般函式），可能獲得更好的性能。

這一小節的剩餘部分，我會給出數個例子，展示仿函式較之於一般函式的優勢所在。第 8 章專攻仿函式，有更多例子和細節。尤其該章爲你展示「以函式行爲作爲 template 參數」這一技術帶給我們的利益。

假設你需要對群集（collection）中的每個元素加上一個固定值。如果你在編譯期便確切知道這個固定數，你可以使用一般函式：

```
void add10 (int& elem)
{
   elem += 10;
}

void f1()
{
   vector<int> coll;
   ...
   for_each (coll.begin(), coll.end(),      // range
             add10);                        // operation
}
```

如果你需要數個不同的固定值，而它們在編譯期都已確切，你可以使用 template:

```
template <int theValue>
void add (int& elem)
{
   elem += theValue;
}

void f1()
{
   vector<int> coll;
   ...
   for_each (coll.begin(), coll.end(),     // range
             add<10>);                     // operation
}
```

如果你必須在執行時期才處理這個數值，那可就麻煩了。你必須在函式被呼叫之前先將這個數值傳給該函式。這通常會導致產生一些全域變數，「演算法的呼叫者」和「演算法所呼叫的函式」都會用到它們。真是一團糟。

如果你兩次用到該函式，每次加數不同，而都是在執行時期才處理，那麼一般函式根本就無能爲力。你要嘛傳入一個標記（tag），要嘛乾脆寫兩個函式。你是否有過這樣的經歷：握有一個函式，它有個 static 變數用以記錄狀態（state），而你需要這個函式在同一時間內有另一個不同狀態（state）？於是你只好拷貝整份函式定義，化爲兩個不同的函式。這正是先前所說的問題。

如果使用仿函式，你就可以寫出「更機靈」的函式，遂你所願。物件可以有自己的狀態，可以被正確初始化。下面是一個完整例子[12]：

```
// stl/add1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

// function object that adds the value with which it is initialized
class AddValue {
```

[12] 輔助函式 PRINT_ELEMENTS() 已於 p118 介紹過。

```
    private:
       int theValue; // the value to add
    public:
       // constructor initializes the value to add
       AddValue(int v) : theValue(v) {
       }

    // the ''function call'' for the element adds the value
    void operator() (int& elem) const {
       elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
       coll.push_back(i);
    }

    PRINT_ELEMENTS(coll,"initialized: ");

    // add value 10 to each element
    for_each (coll.begin(), coll.end(),     // range
              AddValue(10));                // operation

    PRINT_ELEMENTS(coll,"after adding 10: ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),     // range
              AddValue(*coll.begin()));     // operation

    PRINT_ELEMENTS(coll,"after adding first element: ");
}
```

初始化之後，群集內含數值 1 至 9：

```
initialized: 1 2 3 4 5 6 7 8 9
```

第一次呼叫 for_each()，將每個數值加 10：

```
for_each (coll.begin(), coll.end(),    // range
          AddValue(10));               // operation
```

這裡，運算式 AddValue(10) 生出一個 AddValue 物件，並以 10 爲初值。AddValue 建構式將這個值保存在成員 theValue 中。而在 for_each() 之內，針對 coll 的每一個元素呼叫 "()"，實際上就是對傳入的那個 AddValue 暫時物件呼叫 operator()，並以容器元素作爲引數。仿函式（AddValue 物件）將每個元素加 10。結果如下：

```
after adding 10: 11 12 13 14 15 16 17 18 19
```

第二次呼叫 for_each() 亦採用相同機能，將第一元素值加到每個元素身上。首先使用第一元素值做爲仿函式暫時物件的初值：

```
AddValue(*coll.begin())
```

最後結果如下：

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

p335 有這個例子的改進版，其中 AddValue 仿函式的型別被改爲一個 template，可接納不同的加數。

運用此項技術，先前所說的「一個函式、兩個狀態」的問題就可以用「兩個不同的仿函式」加以解決。例如，你可以宣告兩個仿函式，然後各自運用：

```
AddValue addx(x); // function object that adds value x
AddValue addy(y); // function object that adds value y

for_each (coll.begin(),coll.end(), // add value x to each element
          addx);
...
for_each (coll.begin(),coll.end(), // add value y to each element
          addy);
...
for_each (coll.begin(),coll.end(), // add value x to each element
          addx);
```

同樣道理，你也可以提供一些成員函式，在仿函式生命期間查詢或改變物件狀態。

注意，C++ 標準程式庫並未限制演算法「對著一個容器元素」呼叫仿函式的次數，因此可能導致同一個仿函式有若干副本被傳給元素。如果把仿函式當做判斷式（predicates）使用，這個問題會惹來一身麻煩。8.1.4 節, p302 討論了這個問題。

5.9.2 預先定義的仿函式

C++ 標準程式庫包含了一些預先定義的仿函式，涵蓋許多基礎運算。有了它們，很多時候你就不必費心自己去寫仿函式了。一個典型的例子是作爲排序準則的仿函式。operator< 之預設排序準則乃是 less<>，所以，如果你宣告：

```
set<int> coll;
```

會被擴展爲[13]：

```
set<int,less<int> > coll;    // sort elements with <
```

既然如此，想必你能猜到，反向排列這些元素將不是什麼難事[14] ：

```
set<int,greater<int> > coll; // sort elements with >
```

類似情況，還有許多仿函式用於數值處理。下例是將群集中的全部元素都設爲反相（負值）：

```
transform (coll.begin(), coll.end(),   // source
           coll.begin(),               // destination
           negate<int>());             // operation
```

其中運算式：

```
negate<int>()
```

根據預先定義好的 template class negate 生成一個仿函式，將傳進來的 int 值設定爲負。transform()演算法使用此一運算，將第一群集的所有元素處理之後轉移到第二群集。如果轉移目的地就是自己，那麼這段程式碼就是「對群集內的每一個元素取負值」。

同樣道理，你也可以對群集內的所有元素求平方（二次方）。

```
// process the square of all elements
transform (coll.begin(), coll.end(),   // first source
           coll.begin(),               // second source
           coll.begin(),               // destination
           multiplies<int>());         // operation
```

這裡運用了 transform()演算法的另一種形式，以某種特定運算，將兩群集內的元素處理後的結果寫入第三群集。由於本例的三個群集實際上是同一個，所以其

[13] 有些系統並不支持 default template arguments，那麼你只能使用後一種形式。

[14] 注意，兩個 ">" 之間必須保留一個空格，否則 ">>" 會被解析爲右移（right shift）運算子，因而發生語法錯誤。

內的每個元素都被計算了平方值，並寫進群集內，改寫原有值[15]。

透過一些特殊的函式配接器（function adaptors），你還可以將預先定義的仿函式和其他數值組合在一起，或使用特殊狀況。下面是一個完整範例：

```
// stl/fo1.cpp

#include <iostream>
#include <set>
#include <deque>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    set<int,greater<int> > coll1;
    deque<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
         coll1.insert(i);
    }

    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform all elements into coll2 by multiplying 10
    transform (coll1.begin(),coll1.end(),        // source
               back_inserter(coll2),             // destination
               bind2nd(multiplies<int>(),10));   // operation

    PRINT_ELEMENTS(coll2,"transformed: ");

    // replace value equal to 70 with 42
    replace_if (coll2.begin(),coll2.end(),  // range
               bind2nd(equal_to<int>(),70), // replace criterion
               42);                         // new value

    PRINT_ELEMENTS(coll2,"replaced: ");
```

[15] STL 早期版本中，乘法運算的仿函式名爲 `times`。但這和某些作業系統（POSIX, X/Open）中用以計算時間的函式名稱衝突了，所以後來改爲更清楚的名稱：`multiplies`。

```
    // remove all elements with values less than 50
    coll2.erase(remove_if(coll2.begin(),coll2.end(), // range
                          bind2nd(less<int>(),50)), // remove criterion
                coll2.end());

    PRINT_ELEMENTS(coll2,"removed: ");
}
```

其中的述句：

```
transform (coll1.begin(),coll1.end(),       // source
           back_inserter(coll2),            // destination
           bind2nd(multiplies<int>(),10)); // operation
```

將 coll1 內的所有元素乘以 10 後轉移（安插）到 coll2 中。這裡使用配接器 bind2nd，使得進行 multiples<int>運算時，以源群集（source collection）的元素作爲第一引數，10 作爲第二引數。

配接器 bind2nd 的工作方式如下：transform()期望它自己的第四引數是個能接納單一引數(也就是容器實際元素)的運算式，然而我們卻希望先把該元素乘以 10，再傳給 transfor()。所以我們必須構造出一個運算式，接受兩個引數，並以數值 10 作爲第二引數，以此產生一個「只需單引數」的運算式。bind2nd()正好勝任這項工作。它會把運算式保存起來，把第二引數當作內部數值也保存起來。當演算法以實際群集元素爲引數，呼叫 bind2nd 時，bind2nd 把該元素當作第一引數，把原先保存下來的那個內部數值作爲第二引數，呼叫保留下來的那個運算式，並傳回結果。（譯註：這段繁複的文字說明可能解釋效果不甚差，實際情況（源碼運作）請看《STL 源碼剖析》第 8 章, 侯捷著, 碁峰出版 2002）

類似情況，以下的：

```
replace_if (coll2.begin(),coll2.end(),      // range
            bind2nd(equal_to<int>(),70),    // replace criterion
            42);
```

其中的運算式：

```
bind2nd(equal_to<int>(),70)
```

被用來當作一項準則，判斷哪些元素將被 42 替代。bind2nd以 70 作爲第二引數，呼叫二元判斷式（binary predicate）equal_to，從而定義出一個一元判斷式（unary predicate），處理群集內的每一個元素。

最後一個述句也一樣：

```
bind2nd(less<int>(),50)
```

它被用來判斷群集內的哪些元素應當被掃地出門。所有小於 50 的元素都被移除。程式輸出如下：

```
initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced: 90 80 42 60 50 40 30 20 10
removed: 90 80 60 50
```

此種方式的程式編寫，導致函式的組合。有趣的是，所有這些仿函式通常都宣告爲 inline。如此一來，你一方面使用類似函式的表示法或抽象性，一方面又能獲得出色的效能。

另外還有一些仿函式。某些仿函式可用來呼叫群集內每個元素的成員函式：

```
for_each (coll.begin(), coll.end(),          // range
          mem_fun_ref(&Person::save));       // operation
```

仿函式 `mem_fun_ref` 用來呼叫它所作用的元素的某個成員函式。因此上例就是針對 `coll` 內的每個元素呼叫 `Person::save()`。當然啦，唯有當這些元素的型別是 `Person`，或 `Person` 的衍生類別，以上程式碼才能有效運作。

8.2 節, p305 對於 STL 預先定義的仿函式、函式配接器、以及各類函式組合，有更詳盡的討論，並告訴你如何撰寫你自己的仿函式。

5.10 容器內的元素

容器內的元素必須符合特定條件，因爲容器乃是以一種特別方式來操作它們。本節討論這些條件。此外，容器會在內部對其元素進行複製，我也會討論這種行爲的後果。

5.10.1 容器元素的條件

STL 的容器、迭代器、演算法，都是 templates，因此可以操作任何型別 — 不論 STL 預先定義好的或用戶自行定義的，都可以。然而，由於某些加諸於元素身上的操作行爲，某些需求條件也就相應出現了。STL 容器元素必須滿足以下三個基本要求：

1. 必須可透過 *copy* 建構式進行複製。副本與原本必須相等（equivalent），亦即所有相等測試（equality test）的結果都必須顯示，原本與副本行爲一致。

   所有容器都會在內部生成一個元素副本，並傳回該暫時性副本，因此 *copy* 建構式會被頻繁地叫用。所以 *copy* 建構式的性能應該儘可能優化（這雖然不是條件之一，但可視爲獲得良好效能的訣竅）。如果物件的拷貝必須耗費大量時間，你可以選用「reference 語義」來使用容器，因而避免拷貝任何物件。詳見 6.8 節, p222。

2. 必須可以透過 *assignment* 運算子完成賦值動作。容器和演算法都使用 *assignment* 運算子，才能以新元素改寫（取代）舊元素。
3. 必須可以透過解構式完成銷毀動作。當容器元素被移除（*removed*），它在容器內的副本將被銷毀。因此解構式絕不能被設計爲 `private`。此外，依 C++ 慣例，解構式絕不能丟擲異常（throw exceptions），否則沒戲唱了。

這三個條件對任何 class 而言其實都是隱喻成立的。如果某個 class 既沒有爲上述動作定義特殊版本，也沒有定義任何「可能破壞這些動作之健全性」的特殊成員，那麼它自然而然也就滿足了上述條件。

下面幾個條件，也應當獲得滿足[16]：

- 對序列式容器而言，元素的 *default* 建構式必須可用。
  我們可以在沒有給予任何初值的情況下，創建一個非空容器，或增加容器的元素個數。這些元素都將以 *default* 建構式完成。
- 對於某些動作，必須定義 `operator==` 以執行相等測試。如果你有搜尋需求，這一點特別重要。
- 在關聯式容器中，元素必須定義出排序準則。預設情況下是 `operator<`，透過仿函式 `less<>` 被叫用。

5.10.2 Value 語意 vs. Reference 語意

所有容器都會建立元素副本，並回傳該副本。這意味容器內的元素與你放進去的物件「相等（equal）」但非「同一（identical）」。如果你修改容器中的元素，實際上改變的是副本而不是原先物件。這意味 STL 容器所提供的是「*value* 語意」。它們所容納的是你所安插的物件值，而不是物件本身。然而實用上你也許需要用到「*reference* 語意」，讓容器容納元素的 reference。

STL 只支援 *value* 語意，不支援 *reference* 語意。這當然是利弊參半。好處是：

- 元素的拷貝很簡單。
- 使用 references 時容易導致錯誤。你必須確保 reference 所指向的物件仍然健在，並需小心對付偶爾出現的循環引用（circular references）狀態。

缺點是：

- 「拷貝元素」可能導致不好的效能；有時甚至無法拷貝。

[16] 在某些老式系統中，即使你未用到這些額外條件，也必須滿足它們。例如某些 vector 實作版本無論如何用到元素的 *default* 建構式。另一些實作版本則要求 *comparison*（比較）運算子必須存在。然而根據標準，這些要求是錯誤的，所以它們終將逐漸被取消。

- 無法在數個不同的容器中管理同一份物件。

實用上你同時需要兩種作法。你不但需要一份獨立（於原先物件）的拷貝（此乃 *value* 語意），也需要一份代表原資料、俾能相應改變原值的拷貝（此乃 *reference* 語意）。不幸的是，C++ 標準程式庫不支援 *reference* 語意。不過我們可以利用 *value* 語意來實現 *reference* 語意。

一個顯而易見的方法是以指標作爲元素[17]。然而一般指標有些常見問題。例如它們指向的物件也許不復存在，「比較」行爲也未必如你所預期，因爲實際比較的是指標而非指標所指物件。所以使用一般指標作爲容器元素，必須非常謹慎。

好一點的辦法是使用某種智慧型指標（smart pointers），所謂智慧型指標，是一種物件，有著類似指標的介面，但內部作了一些額外檢查和處理工作。這裡有一個重要的問題：它們需要多麼智慧？C++ 標準程式庫確實提供了一個智慧型指標，名爲 `auto_ptr`（詳見 4.2 節, p38），乍見之下用於此處似乎頗爲合適。然而，你可千萬別使用 `auto_ptrs`，因爲它們不符合作爲容器元素所需的基本要求。當 `auto_ptr` 執行了拷貝（*copy*）或賦值（*assign*）動作後，標的物與原物並不相等：原來的那個 `auto_ptr` 發生了變化，其值並不是被拷貝了，而是被移轉了（見 p43 和 p47）。這意味即使對容器中的元素進行排序和列印，也會摧毀它們！所以，千萬別在容器內放置 `auto_ptrs`（如果你的 C++ 系統符合標準規範，當你企圖將 `auto_ptr` 當作容器元素，你應該會收到錯誤訊息）。詳見 p43。

想要獲得適用於 STL 容器的 *reference* 語意，你必須自己寫個合適的智慧型指標。但請注意：就算你使用帶有參用計數（reference counting）功能的智慧型指標（譯註：可參考《*More Effective C++*》條款 28），也就是那種「當最後一個指向物件的 reference 不復存在後，能夠自動摧毀物件」的智慧型指標，仍然很麻煩。舉個例子，如果你擁有直接存取元素的能力，你就可以更改元素值，而這在關聯式容器中卻會打破元素順序關係。你肯定不想那樣是吧！6.8 節 p222 更細緻地探討了容器的 *reference* 語意。尤其棒的是該處展示了一種作法，通過「參用計數」智慧型指標，實現 STL 容器的 *reference* 語意。

5.11 STL 內部的錯誤處理和異常處理

錯誤是無可避免的，可能是程式（程式員）引起的邏輯性錯誤（logical error），也可能是程式運行時的環境或背景（例如記憶體不足）所引起的執行期錯誤（runtime error）。這兩種錯誤都能夠被異常機制（exceptions）處理（p15 有一個關於異常的簡短介紹）。本節討論 STL 內部如何處理錯誤（error）和異常（exceptions）。

[17] C 程式員或許很能認可「以指標實現 reference 語意」的手法。因爲在 C 語言中函式引數只能 *passed by value*（傳值），因此需要藉由指標才能實現所謂的 *call by reference*。

5.11.1 錯誤處理（Error Handling）

STL 的設計原則是效率優先，安全次之。錯誤檢查相當花時間，所以幾乎沒有。如果你能正確無誤地編寫程式，自然很好。如果你不行，那就大難臨頭了。C++ 標準程式庫接納 STL 之前，對於是否應該加入更多的錯誤檢驗，曾有過一些討論。大部分人決定不加入，原因有二：

1. 錯誤檢驗會降低效率，而速度始終是程式的總體目標。剛剛提過，良好的效率是 STL 的設計目標之一。
2. 如果你認爲安全重於效率，你還是可以如願：或增加一層包裝（wrapper），或使用 STL 特殊版本。但是，一旦錯誤檢驗被放進所有基本動作內，再想消除它們以獲得高效率，可就沒門了。舉個例子，如果每一個 *subscript*（下標）運算子都對索引範圍進行合法性檢驗，你就無法撰寫不作檢驗的版本。反過來則可以。

所以，錯誤檢驗是可行的，但並不是 STL 的內在條件。

C++ 標準程式庫指出，對於 STL 的任何運用，如果違反規則，將會導致未定義的行爲。因此，如果索引、迭代器、或區間範圍不合法，結果將未有定義。如果你使用的 STL 並非安全版本，就會導致未定義的記憶體存取，這可能導致難纏的副作用，甚至導致全盤崩潰。從這個意義上說，STL 和 C 指標一樣容易引發錯誤。尋找這樣的錯誤是非常困難的，尤其當你缺乏一個 STL 安全版本時，更是如此。

具體地說，使用 STL，必須滿足以下要求：

- 迭代器務必合法而有效。例如你必須在使用它們之前先將它們初始化。注意，迭代器可能會因爲其他動作的副效應而變得無效。例如當 vectors 和 deques 發生元素的安插、刪除或重新配置時，迭代器可能因此失效。
- 一個迭代器如果指向「逾尾（past-the-end）」位置，它並不指向任何物件，因此不能對它呼叫 `operator*` 或 `operator->`。這一點適用於任何容器的 `end()` 和 `rend()`所傳回的迭代器。
- 區間（range）必須是合法的：
  - 用以指出某個區間的前後兩迭代器，必須指向同一個容器。
  - 從第一個迭代器出發，必須可以到達第二個迭代器所指位置。
- 如果涉及的區間不只一個，第二區間及後繼各區間必須擁有「至少和第一區間一樣多」的元素。
- 覆蓋（overwritten）動作中的「標的區間」（destination ranges）必須擁有足夠元素，否則就必須採用 *insert iterator*s（插入型迭代器）。

以下實例展示了一些可能的錯誤：

```
// stl/iterbug1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll1;       // empty collection
   vector<int> coll2;       // empty collection

   /* RUNTIME ERROR:
   * - beginning is behind the end of the range
   */
   vector<int>::iterator pos = coll1.begin();
   reverse (++pos, coll1.end());

   // insert elements from 1 to 9 into coll2
   for (int i=1; i<=9; ++i) {
        coll2.push_back (i);
   }

   /* RUNTIME ERROR:
   * - overwriting nonexisting elements
   */
   copy (coll2.begin(), coll2.end(),  // source
         coll1.begin());              // destination

   /* RUNTIME ERROR:
   * - collections mistaken
   *-begin() and end() mistaken
   */
   copy (coll1.begin(), coll2.end(),  // source
         coll1.end());                // destination
}
```

注意，這些錯誤發生在執行期間而非編譯期間，因而導致未定義的行爲。

誤用 STL 的方法百百種，STL 沒有義務預防你的各種可能不慎。因此，在軟體開發階段使用「安全版本」的 STL 是個好主意。第一個 STL 安全版本由 Cay Horstmann 開發[18]。不幸的是大部分 STL 開發廠商所供應的 STL，都是植基於 STL 最原始版本，其中並未包含錯誤處理。但是情況正在好轉，有一個帶有警戒能力的 STL 版本，名爲 "STLport"，幾乎適用於任何平台，可自 http://www.stlport.org/ 免費下載。

[18] 你可以從 www.horstmann.com/safestl.html 獲得一份由 Cay Horstmann 開發的 "safe STL"。

5.11.2 異常處理（Exception Handling）

STL 幾乎不檢驗邏輯錯誤。所以邏輯問題幾乎不會引發 STL 產生異常。事實上 *C++ Standard* 只要求唯一一個函式呼叫動作必要時直接引發異常：`vector` 和 `deque` 的成員函式 `at()`（它是下標運算子的受驗版本）。此外，*C++ Standard* 要求，只有一般的（標準的）異常才可以發生，像是因記憶體不足而引發的 *bad_alloc* 或是因客戶自定之操作行爲而引發的異常。

異常何時發生？異常一旦發生對 STL 組件有何影響？在標準化過程中，很長一段時間裡，並未對此問題定義出相關的行爲規範。事實上每一個異常都會引發未定義的行爲。如果執行某項動作的過程中丟擲出異常，那麼即使容器馬上解構，也會導致未定義行爲，例如程式整個崩解。因此如果你需要的是有擔保的、確定的行爲，STL 無能爲力，它甚至不可能正確地將堆疊輾轉開解（所謂 stack unwinding）。

如何處理異常，這是標準化過程中最晚的幾個討論議題之一。找到好的解決方法可不容易，而且花了很長時間，因爲：

1. 很難確定 C++ 標準程式庫究竟應該提供怎樣的安全程度。你大概認爲應該儘可能提供最佳安全性。例如你可能覺得，對著 `vector` 中的任何位置插入一個新元素，要嘛成功，要嘛應該不生任何效果。然而把後繼元素向後移動以空出位置容納新元素，這種行爲通常會導致異常，而且無法復原。如果想要達成上述提出的目標，安插動作就必須把 `vector` 的每一個元素拷貝到新位置去，這對效率是莫大的折損！如果優異效能是設計目標之一（就像 STL），你絕對無法完美處理所有異常狀況，你必須在效率和安全之間尋求某種妥協。
2. 還有一種考量：處理異常的程式碼本身，也會對效能帶來負面影響。這與「儘可能獲得最佳效能」的設計目標抵觸。然而編譯器實作者指出，原則上，異常處理的實作方案應該可以免除任何明顯的效能負荷（許多編譯器也確實做到了這一點）。毫無疑問，如果效能沒有明顯損耗，又能在異常發生時擁有確定、有保障的行爲（而非當機了事），那當然比較好。

經過種種討論，C++ 標準程式庫就「異常處理問題」提供了以下基本保証[19]：C++ 標準程式庫在面對異常時，保証不會發生資源洩漏（resources leak），也不會與容器的恆常特性（container invariants）發生牴觸。

遺憾的是很多時候這還不夠，你需要更強的保證，保證當異常被丟出時，進行中的操作不產生任何影響。以異常的觀點來看，這種操作可被視爲「不可切割的」（**atomic**）。借用資料庫領域的一個術語，這些操作支援所謂「交付或回復，二擇一」（**commit-or-rollback**）行爲，又稱爲「安全交易行爲」（**transaction safe**）。

[19] 特別感謝 Dave Abrahams 和 Greg Colvin 對於 C++ 標準程式庫的異常安全問題所作的貢獻，以及在這個主題上對我的幫助。

考慮到這種強烈需求，C++ 標準程式庫如今做出以下保証：

- 對於所有「以節點爲構造基礎」（node-based）的容器如 `lists`, `sets`, `multisets`, `maps` 和 `multimaps`，如果節點建構失敗，容器保持不變。移除節點的動作保證不會失敗（當然你得保証解構式不得丟出異常）。然而，如果是對關聯式容器插入多個元素，爲保証已序性（*sorted*），失敗時無法完全恢復原狀。所有對關聯式容器「插入單一元素」的操作，支持 **commit-or-rollback** 行爲。也就是說，要不成功，要不沒有任何影響。此外，所有擦拭（erase）操作，無論是針對單一元素或針對多重元素，肯定會成功。

  面對 lists，就算同時插入多個元素，這個操作也是屬於「安全交易行爲」（**transaction-safe**）。事實上 `list` 的所有操作，除了 `remove()`, `remove_if()`, `merge()`, `sort()` 和 `unique()` 之外，要不成功，要不沒有任何影響（也就是 **commit-or-rollback**）。至於上述各函式，C++ 標準程式庫也提供了有條件的保証（見 p172）。所以如果你需要一個 **transaction-safe** 容器，就用 `list` 吧。

- 所有「以 array 爲構造基礎」（array-based）的容器如 `vectors` 和 `deques`，安插元素時如果失敗，都不可能做到完全回復。要達到完全回復，就必須在安插動作之前拷貝所有（安插點之後的）後繼元素。而且爲了實現拷貝動作的完全回復性，需要耗費大量時間。不過由於 `push` 和 `pop` 這兩個動作在容器尾端執行，不需拷貝任何既有元素，所以萬一發生異常，這兩個動作可以保証容器會回復原狀。此外，如果元素的型別能夠保証拷貝動作（也就是 *copy* 建構式和 *assignment* 運算子）不丟出異常，則所有加諸於該種元素身上的操作，都能夠保証「要不成功，要不毫無影響」的行爲。

6.10.10 節, p248 有一份詳細整理，讓你對「異常發生時，擁有較強烈的保證」的各種容器操作，有一份了解。

注意，所有這些保証都有一個前提：解構式不得丟擲異常（C++ 中通常如此）。C++ 標準程式庫做了這個承諾，身爲應用程式員的你，也得做出相同承諾。

如果你需要具備「完全 **commit-or-rollback** 能力」的容器，你應當使用 `list`（但不要呼叫它的 `sort` 和 `unique`），或使用任何關聯式容器（但不要對它安插多個元素）。當你使用它們，可以確保資料不會損失，也確保不會在任何「修改動作」之前先拷貝元素 — 要知道，對一個容器而言，拷貝動作極可能代價高昂。

如果你不使用「以節點爲構造基礎」（node-based）的容器，但又希望獲得「完全 **commit-or-rollback** 能力」，只好自己動手爲每一個關鍵操作提供一份包裝（wrapper）了。舉個例子，以下函式對任何容器而言，幾乎都可以安全地將元素安插於某個特定位置上：

```
template <class T, class Cont, class Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
```

```
    Cont tmp(coll);           // copy container and all elements
    tmp.insert(pos,value);    // modify the copy
    coll.swap(tmp);      // use copy (in case no exception was thrown)
}
```

注意我的用詞，我說「幾乎」，因爲這個函式仍然未臻完美。這是因爲，當 `swap()` 針對關聯性容器複製「比較準則（comparison criterion）」時如果發生異常，那麼 `swap()` 便會丟擲異常。這下你明白了吧，想完美處理異常是多麼不容易！

5.12 擴展 STL

STL 被設計成一個框架（framework），可以向任何方向擴展。你可以提供自己的容器、迭代器、演算法、仿函式…，只要你滿足條件即可。事實上很多有用的擴展都沒有出現在 C++ 標準程式庫中。不能非難他們，C++ 標準委員會必須在某個時刻停止加入新特性，將精力集中於現有特性的完善上，否則標準化工作永無完結之日。STL 遺漏的最重要組件是 hash table（容器類）。這完全是因爲它太晚被提出之故。新的標準程式庫很可能包含數種不同形式的 hash table。大部分 C++ 標準程式庫實作版本已經提供了 hash table，只可惜彼此之間有些差異。詳見 6.7.3 節, p221。

另一些有用的擴展是額外的仿函式（8.3 節, p313）、迭代器（7.5.2 節, p288）、容器（6.7 節, p217）和演算法（7.5.1 節, p285）。

6

STL 容器

STL container

本章延續第 5 章以來的討論，詳細講解 STL 容器。首先對所有容器共通的能力和操作進行巡禮，然後詳細講解每一個容器，包括內部資料結構、操作（operations）、性能，以及各種操作的運用。如果某些操作值得深述，我還會給出相應的實例。每個容器的講解都以一個典型運用實例作爲結束。本章還討論一個有趣的問題：各種容器的使用時機。比較各種容器的能力、優點、缺點之後，你便會了解如何選擇最符合需求的容器。最後，本章詳細介紹了每一個容器的所有成員。這一部分可視爲參考手冊，你可以在其中找到容器介面的細節和容器操作的確切標記式（signature）。必要的時候我會列出交叉索引，幫助你了解相似或互補的演算法。

C++ 標準程式庫還提供了一些特殊的容器類別 — 所謂的「容器配接器」（container adapters，包括 stack, queue, priority queue），以及 bitsets 和 valarrays。這些容器都有一些特殊介面，並不滿足 STL 容器的一般要求，所以本書把它們放在其他章節講解[1]。容器配接器和 bitsets 安排在第 10 章，valarrays 安排在 12.2 節, p547。

[1] 從歷史沿革來說，容器配接器是 STL 的一部分。然而，從概念角度觀之，它們並不屬於 STL framework，它們只不過是「使用」STL。

6.1 容器的共通能力和共通操作

6.1.1 容器的共通能力

本節講述 STL 容器的共通能力。其中大部分都是必要條件，所有 STL 容器都必須滿足那些條件。三個最核心的能力是：

1. 所有容器提供的都是「*value* 語意」而非「*reference* 語意」。容器進行元素的安插動作時，內部實施的是拷貝動作，置於容器內。因此 STL 容器的每一個元素都必須能夠被拷貝。如果你意圖存放的物件不具有 public *copy* 建構式，或者你要的不是副本（例如你要的是被多個容器共同容納的元素），那麼容器元素就只能是指標（指向物件）。5.10.2 節, p135 對此有所描述。
2. 總體而言，所有元素形成一個次序（order）。也就是說，你可以依相同次序一次或多次巡訪每個元素。每個容器都提供「可傳回迭代器」的函式，運用那些迭代器你就可以巡訪元素。這是 STL 演算法賴以生存的關鍵介面。
3. 一般而言，各項操作並非絕對安全。呼叫者必須確保傳給操作函式的引數符合需求。違反這些需求（例如使用非法索引）會導致未定義的行爲。通常 STL 自己不會丟擲異常。如果 STL 容器所呼叫的使用者自定操作（user-defined operations）擲出異常，會導致各不相同的行爲。參見 5.11.2 節, p139。

6.1.2 容器的共通操作

以下操作爲所有容器共有，它們均滿足上述核心能力。表 6.1 列出這些操作。後續各小節分別探討這些共通操作。

初始化（initialization）

每個容器類別都提供了一個 *default* 建構式，一個 *copy* 建構式和一個解構式。你可以以某個已知區間的內容做爲容器初值 — 是的，負責此一行爲的建構式專門用來從另一個容器或 array 或標準輸入裝置（standard input）得到元素並建構出容器。這些建構式都是 member templates（p11），所以如果提供了從「來源端」到「標的端」的元素型別自動轉換，那麼不光是容器型別可以不同，元素型別也可以不同[2]。下面是個實例：

[2] 如果系統本身不支援 member templates，那就只能接受相同型別。你可以換用 `copy()` 演算法，參見 p188 範例。

操作	效果
ContType c	產生一個未含任何元素的空容器
ContType c1(c2)	產生一個同型容器
ContType c(beg,end)	複製 [beg;end) 區間內的元素，做爲容器初值
c.~ContType()	刪除所有元素，釋放記憶體
c.size()	傳回容器中的元素數量
c.empty()	判斷容器是否爲空（相當於 size()==0, 但可能更快）
c.max_size()	傳回元素的最大可能數量
c1 == c2	判斷是否 c1 等於 c2
c1 != c2	判斷是否 c1 不等於 c2，相當於 !(c1 == c2)
c1 < c2	判斷是否 c1 小於 c2
c1 > c2	判斷是否 c1 大於 c2，相當於 c2 < c1
c1 <= c2	判斷是否 c1 小於等於 c2，相當於 !(c2 < c1)
c1 >= c2	判斷是否 c1 大於等於 c2，相當於 !(c1 < c2)
c1 = c2	將 c2 的所有元素指派給 c1
c1.swap(c2)	交換 c1 和 c2 的資料
swap(c1,c2)	同上，是個全域函式
c.begin()	傳回一個迭代器，指向第一元素
c.end()	傳回一個迭代器，指向最後元素的下一位置
c.rbegin()	傳回一個逆向迭代器，指向逆向巡訪時的第一元素
c.rend()	傳回一個逆向迭代器，指向逆向巡訪時的最後元素的下一位置
c.insert(pos,elem)	將 elem 的一份副本安插於 pos 處。回返值和 pos 的意義並不相同。
c.erase(beg,end)	移除 [beg;end) 區間內的所有元素。某些容器會傳回未被移除的第一個接續元素。
c.clear()	移除所有元素，令容器爲空
c.get_allocator()	傳回容器的記憶體模型（memory model）

表 6.1 容器類別（Container Classes）的共通操作函式

- 以另一個容器的元素爲初值，完成初始化動作：

```
std::list<int> l; // l is a linked list of ints
...
// copy all elements of the list as floats into a vector
std::vector<float> c(l.begin(),l.end());
```

- 以某個 array 的元素爲初值，完成初始化動作：

```
int array[] = { 2, 3, 17, 33, 45, 77 };
...
// copy all elements of the array into a set
std::set<int> c(array,array+sizeof(array)/sizeof(array[0]));
```

- 以標準輸入裝置完成初始化動作：

```
// read all integer elements of the deque from standard input
std::deque<int> c((std::istream_iterator<int>(std::cin)),
                  (std::istream_iterator<int>()));
```

注意，不要遺漏了涵括「初始化引數」的那對「多餘的」括弧，否則這個算式的意義會迥然不同，肯定讓你匪夷所思，你會得到一堆奇怪的警告或錯誤。看看不寫括弧的情形：

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

這種情況下 `c` 被視爲一個函式，回返值是 `deque<int>`，第一參數的型別是 `istream_iterator<int>`，參數名爲 `cin`，第二參數無名稱，型別是「一個函式，不接受任何引數，回返值型別爲 `istream_iterator<int>`」。以上結構不論作爲宣告式或算式，語法上都正確。根據 C++ 規則它被視爲宣告式。只要加上一對括弧，便可使引數 `(std::istream_iterator<int>(std::cin))` 不再符合宣告式語法[3]，也就消除了歧義。

原則上還有一些操作，可支援從另一區間獲取資料、指派、插入元素。不過這些操作的確切介面在各容器中彼此不同，有不同的附加引數。

與大小相關的操作函式（Size Operations）

所有容器都提供了三個和大小相關的操作函式：

1. **size()**
   傳回當前容器的元素數量。
2. **empty()**
   這是 `size()==0` 算式的一個快捷形式。`empty()` 的實作可能比 `size()==0` 更有效率，所以你應該儘可能使用它。
3. **max_size()**
   傳回容器所能容納的最大元素數量。其值因實作版本的不同而異。例如 vector 通常保有一個記憶體區塊的全部元素，所以在 PCs 上可能會有相關限定。`max_size()` 通常傳回索引型別的最大值。

[3] 感謝 EDG 的 John H. Spicer 給予的說明。

比較（Comparisons）

包括常用的比較運算子==, !=, <, <=, >, >=。它們的定義依據以下三個規則：

1. 比較動作的兩端（兩個容器）必須屬於同一型別。
2. 如果兩個容器的所有元素依序相等，那麼這兩個容器相等。採用 operator== 檢查元素是否相等。
3. 採用字典式（lexicographical）順序比較原則來判斷某個容器是否小於另一個容器。參見 p360。

比較兩個不同型別的容器，必須使用「比較」演算法，參見 9.5.4 節, p356。

指派（Assignments）和swap()

當你對著容器指派元素時，源容器的所有元素被拷貝到標的容器內，後者原本的所有元素全被移除。所以，容器的指派（賦值）動作代價比較高昂。

如果兩個容器型別相同，而且拷貝後源容器不再被使用，那麼我們可以使用一個簡單的優化方法：swap()。swap()的性能比上述優異得多，因爲它只交換容器的內部資料。事實上它只交換某些內部指標（指向實際資料如元素、配置器、排序準則 — 如果有的話），所以時間複雜度是「常數」，不像實際指派（賦值）動作的複雜度爲「線性」。

6.2 Vectors

vector 模塑出一個 dynamic array。因此，它本身是「將元素置於 dynamic array 中加以管理」的一個抽象概念（圖 6.1）。不過請注意，*C++ Standard* 並未要求必須以 dynamic array 實作 vector，只是規定了相應條件和操作複雜度。

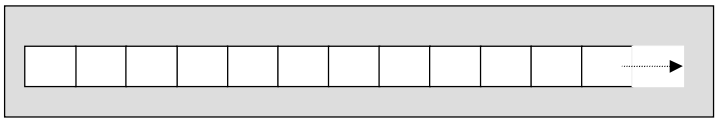


圖 6.1 vector 的結構

使用 vector 之前，必須含入表頭檔 `<vector>`[4]

```
#include <vector>
```

其中，型別 `vector` 是一個定義於 namespace `std` 內的 template：

```
namespace std {
  template <class T,
            class Allocator = allocator<T> >
  class vector;
}
```

vector 的元素可以是任意型別 `T`，但必須具備 *assignable* 和 *copyable* 兩個性質。第二個 template 參數可有可無，用來定義記憶體模型（memory model，參見 15 章）。預設的記憶體模型是 C++ 標準程式庫提供的 `allocator`[5]。

6.2.1 Vectors 的能力

vectors 將其元素複製到內部的 dynamic array 中。元素之間總是存在某種順序，所以 vectors 是一種有序群集（ordered collection）。vector 支援隨機存取，因此只要知道位置，你可以在常數時間內存取任何一個元素。vector 的迭代器是隨機存取迭代器，所以對任何一個 STL 演算法都可以奏效。

在末端附加或刪除元素時，vector 的性能相當好。可是如果你在前端或中部安插或刪除元素，性能就不怎麼樣了，因爲動作點之後的每一個元素都必須移到另一個位置，而每一次移動都得呼叫 *assignment*（賦值）運算子。

---

[4] 早期的 STL 中，vectors 的定義表頭檔是 `<vector.h>`

[5] 在不支援 default template parameters 的系統中，第二參數通常就沒有了。

大小（Size）和容量（Capacity）

vector 優異性能的密訣之一，就是配置比其所容納的元素所需的更多記憶體。爲了能夠高效運用 vectors，你應該瞭解大小和容量之間的關係。

vectors 之中用於操作大小的函式有 `size()`,`empty()`,`max_size()`（6.1.2 節, p144）。另一個與大小有關的函式是 `capacity()`，傳回 vector 實際能夠容納的元素數量。如果超越這個數量，vector 就有必要重新配置內部記憶體。

vector 的容量之所以很重要，有以下兩個原因：

1. 一旦記憶體重新配置，和 vector 元素相關的所有 references、pointers、iterators 都會失效。
2. 記憶體重新配置很耗時間。

所以如果你的程式管理了和 vector 元素相關的 references、pointers、iterators，或如果執行速度對你而言至關重要，那麼就必須考慮容量問題。

你可以使用 `reserve()`保留適當容量，避免一再重新配置記憶體。如此一來，只要保留的容量尚有餘裕，就不必擔心 references 失效。

```
std::vector<int> v;    // create an empty vector
v.reserve(80);         // reserve memory for 80 elements
```

另一種避免重新配置記憶體的方法是，初始化期間就向建構式傳遞附加引數，建構出足夠的空間。如果你的引數是個數值，它將成爲 vector 的起始大小。

```
std::vector<T> v(5); // creates a vector and initializes it with five values
                     // (calls five times the default constructor of type T)
```

當然，要獲得這種能力，此種元素型別必須提供一個 ***default*** 建構式。請注意，如果型別很複雜，就算提供了 ***default*** 建構式，初始化動作也很耗時。如果你這麼做只不過是爲了保留足夠的記憶體，那倒不如使用 `reserve()`。

vectors 的容量，概念上和 strings 類似（參見 11.2.5 節, p485）。不過有一個大不同點：vector 不能使用 `reserve()`來縮減容量，這一點和 strings 不同。如果呼叫 `reserve()`所給的引數比當前 vector 的容量還小，不會引發任何舉動。此外，如何達到時間和空間的最佳效率，係由實作版本決定。因此具體實作版本中，容量的增長幅度可能比你我料想的還大。事實上爲了防止記憶體破碎，在許多實作方案中即使你不呼叫 `reserve()`，當你第一次安插元素時也會一口氣分配整塊記憶體（例如 2K）。如果你有一大堆 vectors，每個 vector 的實際元素卻寥寥無幾，那麼浪費的記憶體相當可觀。

既然 vectors 的容量不會縮減，我們便可確定，即使刪除元素，其 references、pointers、iterators 也會繼續有效，繼續指向動作發生前的位置。然而安插動作卻可能使 references、pointers、iterators 失效（譯註：因爲安插可能導致 vector 重新配置）。

這裏有一個間接縮減 vector 容量的小竅門。注意，兩個 vectors 交換內容後，兩者的容量也會互換，因此下面的例子雖然保留了元素，卻縮減了容量：

```
template <class T>
void shrinkCapacity(std::vector<T>& v)
{
    std::vector<T> tmp(v); // copy elements into a new vector
    v.swap(tmp);           // swap internal vector data
}
```

你甚至可以利用下面的述句直接縮減容量[6]：

```
// shrink capacity of vector v for type T
std::vector<T>(v).swap(v);
```

不過請注意，`swap()`之後原先所有的 references、pointers、iterators 都換了指涉對象；它們仍然指向原本位置。換句話說上述的 `shrinkCapacity()` 使所有 references、pointers、iterators 失效。

6.2.2 Vector 的操作函式

建構、拷貝和解構

表 6.2 列出 vectors 的所有建構式和解構式。你可以在建構時提供元素，也可以不。如果只指定大小，系統便會呼叫元素的 *default* 建構式一一製造新元素。記住，即使對基本型別如 `int`，顯式呼叫 *default* 建構式進行初始化，也是一樣可行（這個特性在 p14 介紹過）。請參考 6.1.2 節, p144 對於初始化的介紹。

操作	效果
`vector<Elem> c`	產生一個空 vector，其中沒有任何元素
`vector<Elem> c1(c2)`	產生另一個同型 vector 的副本（所有元素都被拷貝）
`vector<Elem> c(n)`	利用元素的 *default* 建構式生成一個大小爲 `n` 的 vector
`vector<Elem> c(n,elem)`	產生一個大小爲 `n` 的 vector，每個元素值都是 `elem`
`vector<Elem> c(beg,end)`	產生一個 vector，以區間 `[beg;end)` 做爲元素初值
`c.~vector<Elem>()`	銷毀所有元素，並釋放記憶體

表 6.2 Vectors 的建構式和解構式

[6] 你（或你的編譯器）大概會認爲這個述句實在荒謬，居然針對一個暫時物件呼叫一個 non-const 成員函式。然而標準 C++確實允許我們這麼做。

非變動性操作（Nonmodifying Operations）

表 6.3 列出 vectors 的所有非變動性操作。參見 6.1.2 節, p14 附註和 6.2.1 節, p149。

操作	效果
`c.size()`	傳回當前的元素數量
`c.empty()`	判斷大小是否爲零。等同於 `size()==0`，但可能更快
`c.max_size()`	傳回可容納的元素最大數量
`capacity()`	傳回重新分配空間前所能容納的元素最大數量
`reserve()`	如果容量不足，擴大之[7]
`c1 == c2`	判斷 `c1` 是否等於 `c2`
`c1 != c2`	判斷 `c1` 是否不等於 `c2`，等同於 `!(c1==c2)`。
`c1 < c2`	判斷 `c1` 是否小於 `c2`
`c1 > c2`	判斷 `c1` 是否大於 `c2`，等同於 `c2<c1`。
`c1 <= c2`	判斷 `c1` 是否小於等於 `c2`，等同於 `!(c2<c1)`。
`c1 >= c2`	判斷 `c1` 是否大於等於 `c2`，等同於 `!(c1<c2)`。

表 6.3　Vectors 的非變動性操作

賦值（指派, Assignments）

操作	效果
`c1 = c2`	將 `c2` 的全部元素指派給 `c1`
`c.assign(n,elem)`	複製 `n` 個 `elem`，指派給 `c`
`c.assign(beg,end)`	將區間 `[beg;end)` 內的元素指派給 `c`
`c1.swap(c2)`	將 `c1` 和 `c2` 元素互換
`swap(c1,c2)`	同上。此爲全域函式。

表 6.4　Vectors 的指派（賦值）操作

表 6.4 列出「將新元素指派給 vectors，並將舊元素全部移除」的方法。一系列 `assign()` 函式和建構式一一對應。你可以採用不同的內容指派方式（來自容器、array 和標準輸入裝置），這和 p144 的建構式情況類似。所有指派（賦值）操作都可能會呼叫元素型別的 *default* 建構式、*copy* 建構式、*assignment* 運算子和/或解構式，視元素數量的變化而定。例如：

```
std::list<Elem> l;
std::vector<Elem> coll;
```

[7] `reserve()` 的確會更易（變動，*modify*）vector。因爲它造成所有 references、pointers 和 iterators 失效。但是從邏輯內容來說，容器並沒有變化，所以還是把它列在這裏。

```
...
// make coll be a copy of the contents of l
coll.assign(l.begin(),l.end());
```

元素存取（Element Access）

表 6.5 列出用來直接存取 vector 元素的全部操作函式。按照 C 和 C++ 的慣例，第一元素的索引爲 0，最後元素的索引爲 size()-1。所以第 n 個元素的索引是 n-1。對於 non-const vectors，這些函式都傳回元素的 reference。也就是說你可以使用這些操作函式來更改元素內容（如果沒有其他妨礙因素的話）。

操作	效果
c.at(idx)	傳回索引 idx 所標示的元素。如果 idx 越界，丟擲 *out_of_range*
c[idx]	傳回索引 idx 所標示的元素。不進行範圍檢查。
c.front()	傳回第一元素。不檢查第一個元素是否存在。
c.back()	傳回最後一個元素。不檢查最後一個元素是否存在。

表 6.5 直接用來存取 vectors 元素的各項操作

對呼叫者來說，最重要的事情莫過於搞清楚這些操作是否進行範圍檢查。只有 at() 會那麼做。如果索引越界，at()會丟擲一個 *out_of_range* 異常（詳 3.3 節, p25）。其他函式都不作檢查。如果發生越界錯誤，會引發未定義行爲。對著一個空 vector 呼叫 operator[],front(),back()，都會引發未定義行爲。

```
std::vector<Elem> coll;    // empty!

coll[5] = elem;            // RUNTIME ERROR → undefined behavior
std::cout << coll.front(); // RUNTIME ERROR → undefined behavior
```

所以，呼叫 operator[]時，你必須心裏有數，確定索引有效；呼叫 front()或 back()時必須確定容器不空：

```
std::vector<Elem> coll;    // empty!
if (coll.size() > 5) {
   coll[5] = elem;         // OK
}
if (!coll.empty()) {
   cout << coll.front();   // OK
}
coll.at(5) = elem;         // throws out_of_range exception
```

迭代器相關函式（Iterator Functions）

vectors 提供了一些常規函式來獲取迭代器，如表 6.6。vector 迭代器是 *random access iterator*s（隨機存取迭代器；關於迭代器分類詳見 7.2 節, p251），因此從理論上講，你可以藉此迭代器操作所有 STL 演算法。

操作	效果
`c.begin()`	傳回一個隨機存取迭代器，指向第一元素。
`c.end()`	傳回一個隨機存取迭代器，指向最後元素的下一位置。
`c.rbegin()`	傳回一個逆向迭代器，指向逆向迭代的第一元素。
`c.rend()`	傳回一個逆向迭代器，指向逆向迭代的最後元素的下一位置。

表 6.6　Vectors 的迭代器相關函式

這些迭代器的確切型別由實作版本決定。對 vectors 來說，通常就是一般指標。一般指標就是隨機存取迭代器，而 vector 內部結構通常也就是個 array，所以指標行爲可以適用。不過你可不能仰仗這一點。例如也許有個 STL 安全版本，對所有區間範圍和其他潛在錯誤實施檢查，那麼其 vector 迭代器可能就是個輔助類別。7.2.6 節, p258 展示了「以指標實作迭代器」和「以類別實作迭代器」之間的差異所引起的麻煩問題。

vector 迭代器持續有效，除非發生兩種情況：(1) 使用者在一個較小索引位置上安插或移除元素，(2) 由於容量變化而引起記憶體重新分配（詳見 6.2.1 節, p149）。

安插（insert）和移除（remove）元素

表 6.7 列出 vector 元素的安插、移除操作函式。依 STL 慣例，你必須保證傳入的引數合法：(1) 迭代器必須指向一個合法位置、(2) 區間的起始位置不能在結束位置之後、(3) 絕不能從空容器中移除元素。

關於性能，以下情況你可以預期安插動作和移除動作會比較快些：

- 在容器尾部安插或移除元素
- 容量一開始就夠大
- 安插多個元素時，「呼叫一次」當然比「呼叫多次」來得快

安插元素和移除元素，都會使「作用點」之後的各元素的 references、pointers、iterators 失效。如果安插動作甚至引發記憶體重新分配，那麼該容器身上的所有 references、pointers、iterators 都會失效。

操作	效果
`c.insert(pos,elem)`	在 pos 位置上插入一個 elem 副本，並傳回新元素位置
`c.insert(pos,n,elem)`	在 pos 位置上插入 n 個 elem 副本。無回傳值。
`c.insert(pos,beg, end)`	在 pos 位置上插入區間 [beg;end) 內的所有元素的副本。無回傳值。
`c.push_back(elem)`	在尾部添加一個 elem 副本。
`c.pop_back()`	移除最後一個元素（但不回傳）。
`c.erase(pos)`	移除 pos 位置上的元素，傳回下一元素的位置。
`c.erase(beg,end)`	移除 [beg, end) 區間內的所有元素，傳回下一元素的位置。
`c.resize(num)`	將元素數量改爲 num（如果 size() 變大了，多出來的新元素都需以 *default* 建構式建構完成）
`c.resize(num,elem)`	將元素數量改爲 num（如果 size() 變大了，多出來的新元素都是 elem 的副本）
`c.clear()`	移除所有元素，將容器清空。

表 6.7 vector 的安插、移除相關操作

vectors 並未提供任何函式可以直接移除「與某值相等」的所有元素。這是演算法發揮威力的時候。以下述句可將所有其值爲 val 的元素移除：

```
std::vector<Elem> coll;
...
// remove all elements with value val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
           coll.end());
```

具體解釋詳見 5.6.1 節, p111。

如果只是要移除「與某值相等」的第一個元素，可以這麼做：

```
std::vector<Elem> coll;
...
// remove first element with value val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(),
           val);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

6.2.3 將 Vectors 當作一般 Arrays 使用

C++ 標準程式庫並未明確要求 vector 的元素必須分佈於連續空間中。但是一份標準規格缺陷報告顯示，這個缺點將獲得彌補，標準規格書中將明確保證上述論點。如此一來你可以確定，對於 vector `v` 中任意一個合法索引 `i`，以下算式肯定爲 true：

```
&v[i] == &v[0] + i
```

保證了這一點，就可推導出一系列重要結果。簡單地說，任何地點只要你需要一個 dynamic array，你就可以使用 vector。例如你可以利用 vector 來存放常規的 C 字串（型別爲 `char*` 或 `const char*`）：

```
std::vector<char> v;        // create vector as dynamic array of chars
v.resize(41);          // make room for 41 characters (including '\0')
strcpy(&v[0],"hello, world");   // copy a C-string into the vector
printf("%s\n", &v[0]); // print contents of the vector as C-string
```

不過，這麼運用 vector 你可得小心（和使用 dynamic array 一樣小心），例如你必須確保上述 vector 的大小足以容納所有資料，如果你用的是 C-String，記住最後有個 `'\0'` 元素。這個例子說明，不管出於什麼原因（例如爲了和既有的 C 程式庫打交道），只要你需要一個元素型別爲 `T` 的 array，就可以採用 `vector<T>`，然後傳遞第一元素的位址給它。

注意，千萬不要把迭代器當作第一元素的位址來傳遞。vector 迭代器是由實作版本定義的，也許並不是個一般指標。

```
printf("%s\n", v.begin()); // ERROR (might work, but not portable)
printf("%s\n", &v[0]);     // OK
```

6.2.4 異常處理（Exception Handling）

vector 只支援最低限度的邏輯錯誤檢查。*subscript*（下標）運算子的安全版本 `at()`，是唯一被標準規格書要求可能丟擲異常的一個函式（p152）。此外標準規格書也規定，只有一般標準異常（例如記憶體不足時丟擲 *bad_alloc*），或用戶自定操作函式的異常，才可能發生。

如果 vector 呼叫的函式（元素型別所提供的函式，或使用者提供的函式）擲出異常，C++ 標準程式庫作出如下保證：

1. 如果 `push_back()` 安插元素時發生異常，該函式不生效用。
2. 如果元素的拷貝動作（包括 *copy* 建構式和 *assignment* 運算子）不丟擲異常，那麼 `insert()` 要嘛成功，要嘛不生效用。
3. `pop_back()` 決不會丟擲任何異常。

4. 如果元素拷貝動作(包括 *copy* 建構式和 *assignment* 運算子)不丟擲異常，`erase()` 和 `clear()` 就不丟擲異常。
5. `swap()` 不丟擲異常。
6. 如果元素拷貝動作（包括 *copy* 建構式和 *assignment* 運算子）絕對不會丟擲異常，那麼所有操作不是成功，就是不生效用。這類元素可被稱爲 **POD**（plain old data，簡樸的老式資料）。POD 泛指那些無 C++ 特性的型別，例如 C structure 便是。

所有這些保證都基於一個條件：解構式不得丟擲異常。參見 5.11.2 節, p139 對於 STL 異常處理的一般性討論。6.10.10 節, p249 列出對於異常給予特別保證的所有容器操作函式。

6.2.5 Vectors 運用實例

下面例子展示了 vectors 的簡單用法：

```
// cont/vector1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  // create empty vector for strings
  vector<string> sentence;

  // reserve memory for five elements to avoid reallocation
  sentence.reserve(5);

  // append some elements
  sentence.push_back("Hello,");
  sentence.push_back("how");
  sentence.push_back("are");
  sentence.push_back("you");
  sentence.push_back("?");

  // print elements separated with spaces
  copy (sentence.begin(), sentence.end(),
        ostream_iterator<string>(cout," "));
  cout << endl;
```

```
  // print "technical data"
  cout << " max_size(): " << sentence.max_size() << endl;
  cout << " size(): " << sentence.size() << endl;
  cout << " capacity(): " << sentence.capacity() << endl;

  // swap second and fourth element
  swap (sentence[1], sentence[3]);

  // insert element "always" before element "?"
  sentence.insert (find(sentence.begin(),sentence.end(),"?"),
                   "always");

  // assign "!" to the last element
  sentence.back() = "!";

  // print elements separated with spaces
  copy (sentence.begin(), sentence.end(),
        ostream_iterator<string>(cout," "));
  cout << endl;

  // print "technical data" again
  cout << " max_size(): " << sentence.max_size() << endl;
  cout << " size(): " << sentence.size() << endl;
  cout << " capacity(): " << sentence.capacity() << endl;
}
```

程式的輸出可能像這樣：

```
Hello, how are you ?
  max_size(): 268435455
  size(): 5
  capacity(): 5
Hello, you are how always !
  max_size(): 268435455
  size(): 6
  capacity(): 10
```

注意我說「可能」。是的，`max_size()`和 `capacity()`的結果由實作版本決定。從這個例子中你可以看到，當容量不足時，此一實作版本將容量擴充一倍。

6.2.6 Class vector<bool>

C++ 標準程式庫專門針對元素型別爲 bool 的 vector 設計了一個特殊版本，目的是獲取一個優化的 vector。其耗用空間遠遠小於以一般 vector 實作出來的 bool vector。一般 vector 的實作版本會爲每個元素至少分配一個 byte 空間，而 vector<bool>特殊版本內部只用一個 bit 來存儲一個元素。所以通常小 8 倍之多。不過這裏有個小麻煩：C++ 的最小可定址值通常以 byte 爲單位。所以上述的 vector 特殊版本需針對 references 和 iterators 做特殊考慮。

考量結果是，vector<bool> 無法滿足其他 vectors 必須的所有條件（例如 vector<bool>::reference 並不傳回真正的 lvalue，vector<bool>::iterator 不是個真正的隨機存取迭代器）。所以某些 template 程式碼可能適用於任何型別的 vector，唯獨無法應付 vector<bool>。此外 vector<bool> 可能比一般 vectors 慢一些，因爲所有元素操作都必須轉化爲 bit 操作。不過 vector<bool> 的具體方案也是由實作版本決定，所以性能（包括速度和空間消耗）也可能都有不同。

注意，vector<bool> 不僅僅是個特殊的 bool 版本，它還提供某些特殊的 bit 操作。你可以利用它們更方便操作 bit 或旗標（flags），而且由於 vector<bool> 的大小可動態改變，你還可以把它當成動態大小的 bitfield（位元場）。如此一來你便可以添加或移除 bits。如果你需要靜態大小的 bitfield，應當使用 bitset，而不是 vector<bool>。bitset 詳見 10.4 節, p460。

操作	效果
c.flip()	將所有 bool 元素值取反值，亦即求補數。
m[idx].flip()	將索引 idx 的 bit 元素取反值
m[idx] = val	令索引 idx 的 bit 元素值爲 val（指定單一 bit）
m[idx1] = m[idx2]	令索引 idx1 的 bit 元素值爲索引 idx2 的 bit 元素值

表 6.8 vector<bool> 的特殊操作

表 6.8 列出 vector<bool> 的特殊操作。flip()對 vector 中的每一個 bit 取補數。注意，你竟然可以對單一 bool 元素呼叫 flip()。是不是很驚訝？也許你覺得讓 *subscript* 運算子傳回 bool，再對如此基本型別呼叫 flip()是不可能的。然而這裏 vector<bool>用了一個常見技巧，稱作 **proxy**[8]，對於 vector<bool>，*subscript* 運算子（及其他傳回單一元素的運算子）的回返型別實際上是個輔助類別，一旦

[8] proxy 可讓你控制一般無法控制的東西，通常用來獲取更好的安全性。上述情形中，此技術施行某種控制，使某種操作成爲可能。原則上其傳回的物件行爲類似 bool。

你要求回返值爲 bool，便會觸發一個自動型別轉換函式。表 6.8 的其他操作由成員函式支援。vector<bool> 的相關宣告如下：

```
namespace std {
class vector<bool> {
   public:
      // auxiliary type for subscript operator
      class reference {
         ...
         public:
            // automatic type conversion to bool
            operator bool() const;

            // assignments
            reference& operator= (const bool);
            reference& operator= (const reference&);

            // bit complement
            void flip();
      }
      ...

      // operations for element access
      // - return type is reference instead of bool
      reference operator[](size_type n);
      reference at(size_type n);
      reference front();
      reference back();
      ...
   };
}
```

你會發現，所有用於元素存取的函式，傳回的都是 reference 型別。所以，你可以使用以下述句：

```
c.front().flip(); // negate first Boolean element
c[5] = c.back();  // assign last element to element with index 5
```

一如往常，爲了避免未定義的行爲，呼叫者必須確保第一、第六和最後一個元素存在。

只有在 non-const vector<bool> 容器中才會用到內部型別 reference。存取元素用的 const member function 會傳回型別爲 bool 的普通數值。

6.3 Deques

容器 deque（發音爲 "deck"）和 vector 非常相似。它也採用 dynamic array 來管理元素，提供隨機存取，並有著和 vector 幾乎一模一樣的介面。不同的是 deque 的 dynamic array 頭尾都開放，因此能在頭尾兩端進行快速安插和刪除（圖 6.2）。

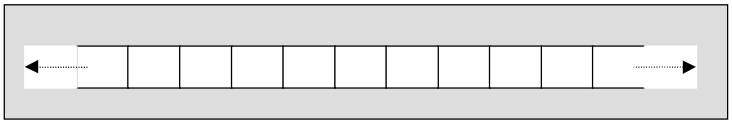


圖 6.2 deque 的邏輯結構

爲了獲取這種能力，deque 通常實作爲一組獨立區塊，第一區塊朝某方向擴展，最後一個區塊朝另一方向擴展，如圖 6.3。

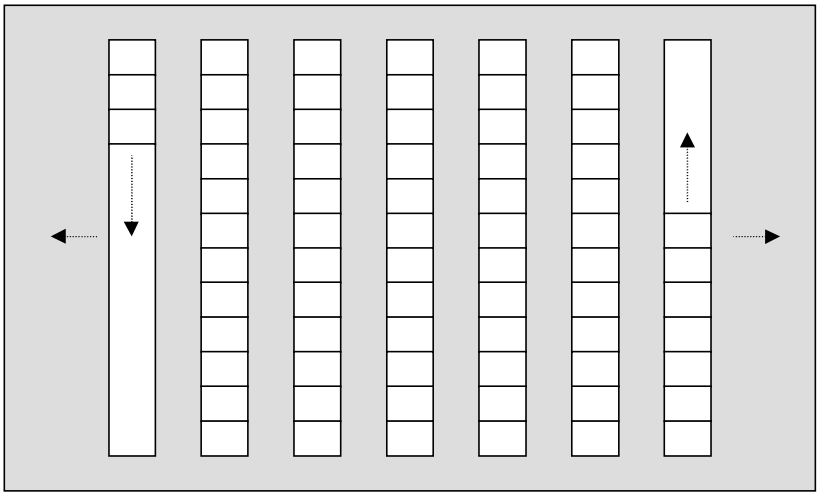


圖 6.3 deque 的內部結構

使用 deque 之前，必須先含入表頭檔 `<deque>`[9]：

```
#include <deque>
```

在其中，`deque` 型別是定義於命名空間 `std` 內的一個 class template：

```
namespace std {
    template <class T,
```

[9] 早期的 STL 中，deque 的表頭檔是 `<deque.h>`。

```
                class Allocator = allocator<T> >
    class deque;
}
```

和 vector 相同，第一個 template 參數用來表明元素型別 — 只要是 *assignable* 和 *copyable* 都可以勝任。第二個 template 參數可有可無，用來指定記憶體模型（memory model），預設爲 `allocator`（詳見第 15 章）[10]。

6.3.1 Deques 的能力

與 vectors 相比，deques 功能上的不同處在於：

- 兩端都能快速安插元素和移除元素（vector 只在尾端逞威風）。這些操作可以在分期攤還的常數時間（amortized constant time）內完成。
- 存取元素時，deque 的內部結構會多一個間接過程，所以元素的存取和迭代器的動作會稍稍慢一些。
- 迭代器需要在不同區塊間跳轉，所以必須是特殊的智慧型指標，非一般指標。
- 在對記憶體區塊有所限制的系統中（例如 PC 系統），deque 可以內含更多元素，因爲它使用不止一塊記憶體。因此 deque 的 `max_size()` 可能更大。
- deque 不支援對容量和記憶體重分配時機的控制。特別要注意的是，除了頭尾兩端，在任何地方安插或刪除元素，都將導致指向 deque 元素的任何 pointers、references、iterators 失效。不過，deque 的記憶體重分配優於 vectors，因爲其內部結構顯示，deques 不必在記憶體重分配時複製所有元素。
- deque 的記憶體區塊不再被使用時，會被釋放。deque 的記憶體大小是可縮減的。不過，是不是這麼做，以及究竟怎麼做，由實作版本定義之。

deques 的下述特性跟 vectors 差不多：

- 在中段部分安插、移除元素的速度相對較慢，因爲所有元素都需移動以騰出或填補空間。
- 迭代器屬於 *random access iterator*（隨機存取迭代器）。

總之，如果是以下情形，最好採用 deque：

- 你需要在兩端安插和移除元素（這是 deque 的拿手好戲）。
- 無需引用（*refer to*）容器內的元素。
- 要求容器釋放不再使用的元素（不過，標準規格上並沒有保證這一點）。

vectors 和 deques 的介面幾乎一樣，所以如果無需什麼特殊性質，兩者都可試試。

---

[10] 在尚未支援 default template parameters 的系統中，第二參數通常會被省略。

6.3.2 Deque 的操作函式

表 6.9 至表 6.11 列出了 deque 的所有操作函式：

操作	效果
`deque<Elem> c`	產生一個空的 deque
`deque<Elem> c1(c2)`	針對某個 deque 產生同型副本（所有元素都被拷貝）
`deque<Elem> c(n)`	產生一個 deque，含有 n 個元素，這些元素均以 *default* 建構式產生出來。
`deque<Elem> c(n,elem)`	產生一個 deque，含有 n 個元素，這些元素均是 elem 的副本。
`deque<Elem> c(beg,end)`	產生一個 deque，以區間 [beg;end) 內的元素爲初值
`c.~deque<Elem>()`	銷毀所有元素，釋放記憶體

表 6.9 deques 的建構式和解構式

操作	效果
`c.size()`	傳回容器的實際元素個數
`c.empty()`	判斷容器大小是否爲零。等同於 `size()==0`，但可能更快。
`c.max_size()`	傳回可容納的最大元素數量。
`c1 == c2`	判斷是否 c1 等於 c2
`c1 != c2`	判斷是否 c1 不等於 c2。等同於 `!(c1 == c2)`。
`c1 < c2`	判斷是否 c1 小於 c2。
`c1 > c2`	判斷是否 c1 大於 c2。等同於 `c2 < c1`。
`c1 <= c2`	判斷是否 c1 小於等於 c2 。等同於 `!(c2 < c1)` 。
`c1 >= c2`	判斷 c1 是否大於等於 c2 。等同於 `!(c1 < c2)` 。
`c.at(idx)`	傳回索引 idx 所標示的元素。如果 idx 越界，丟擲 *out_of_range*
`c[idx]`	傳回索引 idx 所標示的元素，不進行範圍檢查。
`c.front()`	傳回第一個元素。不檢查元素是否存在。
`c.back()`	傳回最後一個元素。不檢查元素是否存在。
`c.begin()`	傳回一個隨機迭代器，指向第一元素。
`c.end()`	傳回一個隨機迭代器，指向最後元素的下一位置。
`c.rbegin()`	傳回一個逆向迭代器，指向逆向迭代時的第一個元素。
`c.rend()`	傳回一個逆向迭代器，指向逆向迭代時的最後元素的下一位置。

表 6.10 deques 的非變動性操作（nonmodifying operations）

操作	效果
`c1 = c2`	將 c2 的所有元素指派給 c1
`c.assign(n,elem)`	將 n 個 elem 副本指派給 c
`c.assign(beg,end)`	將區間 [beg;end) 中的元素指派給 c
`c1.swap(c2)`	將 c1 和 c2 的元素互換
`swap(c1,c2)`	同上。此爲全域函式。
`c.insert(pos,elem)`	在 pos 位置插入一個 elem 副本，並傳回新元素的位置
`c.insert(pos,n, elem)`	在 pos 位置插入 elem 的 n 個副本，無回返值。
`c.insert(pos,beg, end)`	在 pos 位置插入在區間 [beg;end) 所有元素的副本，無回返值。
`c.push_back(elem)`	在尾部添加 elem 的一個副本
`c.pop_back()`	移除最後一個元素（但不回傳）
`c.push_front(elem)`	在頭部插入 elem 的一個副本
`c.pop_front()`	移除頭部元素（但不回傳）
`c.erase(pos)`	移除 pos 位置上的元素，傳回下一元素位置
`c.erase(beg,end)`	移除 [beg, end) 區間內的所有元素，傳回下一元素位置
`c.resize(num)`	將大小（元素個數）改爲 num。如果 size() 增長了，新增元素都以 *default* 建構式產生出來。
`c.resize(num,elem)`	將大小（元素個數）改爲 num。如果 size() 增長了，新增元素都是 elem 的副本。
`c.clear()`	移除所有元素，將容器清空。

表 6.11 deques 的變動性操作（modifying operations）

deques 的各項操作只在以下數點和 vectors 不同：

1. deques 不提供容量操作（capacity() 和 reserve()）。
2. deque 直接提供函式，用以完成頭部元素的安插和刪除（push_front() 和 pop_front()）。

其他操作都相同，所以這裏不重複。它們的具體描述請見 p150, 6.2.2 節。

還有一些值得考慮的事情：

1. 除了 at()，沒有任何成員函式會檢查索引或迭代器是否有效。
2. 元素的插入或刪除可能導致記憶體重新分配，所以任何插入或刪除動作都會使所有指向 deque 元素的 pointers、references 和 iterators 失效。唯一例外是在頭部或尾部插入元素，動作之後，pointers 和 references 仍然有效（但 iterators 就沒這麼幸運）。

6.3.3 異常處理（Exception Handling）

原則上 deque 提供的異常處理和 vectors 提供的一樣（p155）。新增的操作函式 `push_front()`和 `pop_front()`分別對應於 `push_back()`和 `pop_back()`。因此，C++ 標準程式庫保證下列行爲：

- 如果以 `push_back()`或 `push_front()`安插元素時發生異常，則該操作不帶來任何效應。
- `pop_back()`和`pop_front()`不會丟擲任何異常。

STL 的異常處理一般原則請見 p139, 5.11.2 節。異常發生時，提供特殊保障的所有容器操作函式均列於 p248, 6.10.10 節。

6.3.4 Deques 運用實例

以下程式以簡單的例子說明 deque 的功用：

```
// cont/deque1.cpp

#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    // create empty deque of strings
    deque<string> coll;

    // insert several elements
    coll.assign (3, string("string"));
    coll.push_back ("last string");
    coll.push_front ("first string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout,"\n"));
    cout << endl;

    // remove first and last element
    coll.pop_front();
    coll.pop_back();
```

```
    // insert "another" into every element but the first
    for (int i=1; i<coll.size(); ++i) {
         coll[i] = "another " + coll[i];
    }

    // change size to four elements
    coll.resize (4, "resized string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout,"\n"));
}
```

程式輸出如下：

```
first string
string
string
string
last string

string
another string
another string
resized string
```

6.4 Lists

List 使用一個 doubly linked list（雙向串列）來管理元素，如圖 6.4。按慣例，C++ 標準程式庫並未明定實作方式，只是遵守 list 的名稱、限制和規格。

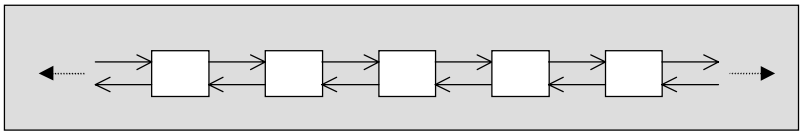


圖 6.4 List 的結構

使用 list 時必須含入表頭檔 `<list>`[11]：

```
#include <list>
```

其中 list 型別係定義於 namespace `std` 中，是個 class template：

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class list;
}
```

任何型別 `T` 只要具備 *assignable* 和 *copyable* 兩性質，就可以作爲 list 的元素。第二個 template 引數可有可無，用來指定記憶體模型（詳見第 15 章）。預設的記憶體模型是 C++ 標準程式庫所提供的 `allocator`[12]。

6.4.1 Lists 的能力

List 的內部結構和 vector 或 deque 截然不同，所以在幾個主要方面與前述二者存在明顯區別：

- List 不支援隨機存取。如果你要存取第 5 個元素，就得順著串鏈一一爬過前 4 個元素。所以，在 list 中隨機巡訪任意元素，是很緩慢的行爲。
- 任何位置上（不只是兩端）執行元素的安插和移除都非常快，始終都是常數時間內完成，因爲無需移動任何其他元素。實際上內部只是進行了一些指標操作而已。

[11] 早期 STL 中，list 的定義表頭檔是 `<list.h>`

[12] 如果系統不支援 default template parameters，通常省略第二引數。

- 安插和刪除動作並不會造成指向其他元素的各個 pointers、references、iterators 失效。
- List 對於異常有著這樣的處理方式：要嘛操作成功，要嘛什麼都不發生。你決不會陷入「只成功一半」這種前不著村後不巴店的尷尬境地。

Lists 所提供的成員函式反映出它和 vectors 以及 deques 的不同：

- 由於不支援隨機存取，lists 既不提供 *subscript*（下標）運算子，也不提供 `at()`。
- Lists 並未提供容量、空間重新分配等操作函式，因爲全無必要。每個元素都有自己的記憶體，在被刪除之前一直有效。
- Lists 提供了不少特殊的成員函式，專門用於移動元素。較之同名的 STL 通用演算法，這些函式執行起來更快，因爲它們無需拷貝或移動，只需調整若干指標即可。

6.4.2 List 的操作

生成（Creation），複製（Copy）和銷毀（Destroy）

lists 的生成、複製和銷毀動作，和所有序列式容器相同，詳見表 6.12。關於初值來源（initialization sources）的若干注意事項，可參考 6.1.2 節, p144。

操作	效果
`list<Elem> c`	產生一個空的 list
`list<Elem> c1(c2)`	產生一個與 `c2` 同型的 list（每個元素都被複製）
`list<Elem> c(n)`	產生擁有 `n` 個元素的 list，這些元素都以 *default* 建構式初始化
`list<Elem> c(n,elem)`	產生擁有 `n` 個元素的 list，每個元素都是 `elem` 的副本。
`list<Elem> c(beg,end)`	產生一個 list 並以 `[beg;end)` 區間內的元素爲初值
`c.~list<Elem>()`	銷毀所有元素，釋放記憶體

表 6.12　Lists 的建構式和解構式

非變動性操作（Nonmodifying Operations）

Lists 也提供諸如「詢問大小」和「兩相比較」等等一般性操作。詳見表 6.13 和 6.1.2 節, p144。

操作	效果
`c.size()`	傳回元素個數
`c.empty()`	判斷容器大小是否爲零。等同於 `size()==0`，但可能更快。
`c.max_size()`	傳回元素的最大可能數量
`c1 == c2`	判斷是否 `c1` 等於 `c2`
`c1 != c2`	判斷是否 `c1` 不等於 `c2`。等同於`!(c1 == c2)`。
`c1 < c2`	判斷是否 `c1` 小於 `c2`
`c1 > c2`	判斷是否 `c1` 大於 `c2`。等同於與 `c2 < c1` 相同。
`c1 <= c2`	判斷是否 `c1` 小於等於 `c2`。等同於 `!(c2 < c1)`。
`c1 >= c2`	判斷是否 `c1` 大於等於 `c2`。等同於 `!(c1 < c2)`。

表 6.13 Lists 的非變動性操作（Nonmodifying Operations）

賦值（指派, Assignment）

和其他序列式容器一樣，lists 也提供了一般常用的賦值（指派）動作，如表 6.14。

操作	效果
`c1 = c2`	將 `c2` 的全部元素指派給 `c1`
`c.assign(n,elem)`	將 `elem` 的 `n` 個拷貝指派給 `c`
`c.assign(beg,end)`	將區間`[beg;end)`的元素指派給 `c`
`c1.swap(c2)`	將 `c1` 和 `c2` 的元素互換
`swap(c1,c2)`	同上。此爲全域函式。

表 6.14 Lists 的 ***assignment***（指派, 賦值）操作函式

一如往常，安插動作和建構式一一匹配，如此一來就有能力提供不同的初值來源（initialization sources），詳見 6.1.2 節, p144。

元素存取（Element Access）

list 不支援隨機存取，只有 `front()`和 `back()`能夠直接存取元素，如表 6.15。

操作	效果
`c.front()`	傳回第一個元素。不檢查元素存在與否。
`c.back()`	傳回最後一個元素。不檢查元素存在與否。

表 6.15 Lists 元素的直接存取

一如以往，這些操作並不檢查容器是否爲空。對著空容器執行任何操作，都會導致未定義的行爲。所以呼叫者必須確保容器至少含有一個元素。例如：

```
std::list<Elem> coll;       // empty!

std::cout << coll.front(); // RUNTIME ERROR ➔ undefined behavior

if (!coll.empty()) {
    std::cout << coll.back();  // OK
}
```

迭代器相關函式（Iterator Functions）

只有運用迭代器，才能夠存取 list 中的各個元素。Lists 提供的迭代器函式如表 6.16。然而由於 list 不能隨機存取，這些迭代器只是雙向（而非隨機）迭代器。所以凡是用到隨機存取迭代器的演算法（所有用來操作元素順序的演算法 — 特別是排序演算法 — 都歸此類）你都不能呼叫。不過你可以拿 list 的特殊成員函式 `sort()` 取而代之，詳見 p245。

操作	效果
`c.begin()`	傳回一個雙向迭代器，指向第一元素。
`c.end()`	傳回一個雙向迭代器，指向最後元素的下一位置。
`c.rbegin()`	傳回一個逆向迭代器，指向逆向迭代的第一個元素。
`c.rend()`	傳回一個逆向迭代器，指向逆向迭代的最後元素的下一位置。

表 6.16　Lists 的迭代器相關函式

元素的安插（Inserting）和移除（Removing）

表 6.17 列出 lists 元素的安插和移除操作。Lists 提供 deques 的所有功能，還增加了 `remove()`和 `remove_if()`演算法應用於 list 身上的特殊版本。

和一般運用 STL 時相似，你必須確保引數的正確。迭代器必須指向合法位置，區間終點不能位於區間起點的前頭；還有，你不能從空容器中移除元素。

如果想要安插或移除多個元素，你可以對它們進行單一呼叫，比多次呼叫來得快。

爲了移除元素，Lists 特別配備了 `remove()`演算法（9.7.1 節, p378）的特別版本。這些成員函式比 `remove()`演算法的速度更快，因爲它們只進行內部指標的操作，無需顧及元素。所以，面對 list，你應該呼叫成員函式 `remove()`，而不是像面對 vectors 和 deques 那樣呼叫 STL 演算法（如 p154 所示）。

操作	效果
c.insert(pos,elem)	在迭代器 pos 所指位置上安插一個 elem 副本，並傳回新元素的位置。
c.insert(pos,n,elem)	在迭代器 pos 所指位置上安插 n 個 elem 副本，無回返值。
c.insert(pos,beg,end)	在迭代器 pos 所指位置上安插 [beg;end) 區間內的所有元素的副本，無回返值。
c.push_back(elem)	在尾部追加一個 elem 副本。
c.pop_back()	移除最後一個元素（但不傳回）。
c.push_front(elem)	在頭部安插一個 elem 副本。
c.pop_front()	移除第一元素（但不傳回）。
c.remove(val)	移除所有其值爲 val 的元素。
c.remove_if(op)	移除所有「造成 op(elem) 結果爲 true」的元素。
c.erase(pos)	移除迭代器 pos 所指元素，傳回下一元素位置。
c.erase(beg,end)	移除區間 [beg;end) 內的所有元素，傳回下一元素位置
c.resize(num)	將元素容量變爲 num。如果 size() 變大，則以 *default* 建構式建構所有新增元素。
c.resize(num,elem)	將元素容量變爲 num。如果 size() 變大，則以 elem 副本做爲新增元素的初值。
c.clear()	移除全部元素，將整個容器清空。

表 6.17 Lists 的安插、移除操作函式

想要將所有「與某值相等」的元素移除，可以這麼做（進一步細節詳見 5.6.3 節, p116）：

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

如果只是想移除「與某值相等」的第一個元素，你得使用諸如 p154 中針對 vectors 所用的演算法。

如果使用 remove_if()，你可以藉由一個函式或仿函式[13]來定義元素移除原則；它可以將每一個「令傳入之操作結果爲 true」的元素移除：

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

如果你對以上述句感到頭暈，別急，前進 p306 看看詳細解釋。關於 remove() 和 remove_if() 的其他例子，詳見 p378。

[13] 一個不支援 member templates 的系統，通常不會提供 remove_if() 成員函式。

Splice 函式

Linked lists 的一大好處就是不論在任何位置，元素的安插和移除都只需要常數時間。如果你有必要將若干元素從 A 容器轉放到 B 容器，那麼上述好處就更見其效了，因爲你只需要重新定向某些指標即可，如圖 6.5。

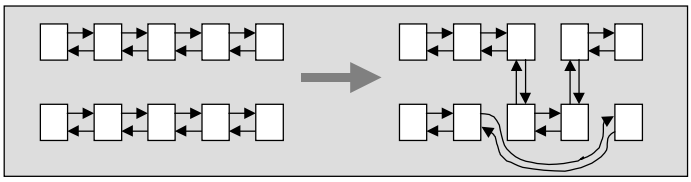


圖 6.5 Splice 操作函式用以改變 list 元素的次序

爲了利用這個優勢，lists 不僅提供 `remove()`，還提供其他一些成員函式，用來改變元素和區間次序，或是用來重新串鏈。我們不僅可以呼叫這些函式，移動單一 list 內的元素，也可以移動兩個 lists 之間的元素 — 只要 lists 的型別一致即可。表 6.18 列舉出這些函式。6.10.8 節, p244 另有詳細說明，實例可見 6.4.4 節, p172。

操作	效果
`c.unique()`	如果存在若干相鄰而數值相同的元素，就移除重複元素，只留下一個。
`c.unique(op)`	如果存在若干相鄰元素，都使 `op()`的結果爲 `true`，則移除重複元素，只留下一個。
`c1.splice(pos,c2)`	將 `c2` 內的所有元素轉移到 `c1` 之內、迭代器 `pos` 之前。
`c1.splice(pos,c2, c2pos)`	將 `c2` 內的 `c2pos` 所指元素轉移到 `c1` 內的 `pos` 所指位置上（`c1` 和 `c2` 可相同）。
`c1.splice(pos,c2, c2beg,c2end)`	將 `c2` 內的`[c2beg;c2end)`區間內所有元素轉移到 `c1` 內的 `pos` 之前（`c1` 和 `c2` 可相同）。
`c.sort()`	以 `operator<` 爲準則，對所有元素排序。
`c.sort(op)`	以 `op()`爲準則，對所有元素排序。
`c1.merge(c2)`	假設 `c1` 和 `c2` 容器都包含已序（*sorted*）元素，將 `c2` 的全部元素轉移到 `c1`，並保證合併後的 list 仍爲已序。
`c1.merge(c2,op)`	假設 `c1` 和 `c2` 容器都包含 `op()`原則下的已序（*sorted*）元素，將 `c2` 的全部元素轉移到 `c1`，並保證合併後的 list 在 `op()`原則下仍爲已序。
`c.reverse()`	將所有元素反序（reverse the order）。

表 6.18 Lists 的特殊變動性操作（Special Modifying Operations）

6.4.3 異常處理（Exception Handling）

所有 STL 標準容器中，lists 對於異常安全性（**exception safety**）提供了最佳支援。幾乎所有操作都是不成功便成仁：要嘛成功，要嘛無效。僅有少數幾個操作沒有如此保證，包括指派（賦值）運算和成員函式 `sort()`，不過它們也有基本保證：異常發生時不會洩漏資源，也不會與容器恆常特性（invariants）發生衝突。`merge()`, `remove()`,`remove_if()`,`unique()` 提供的保證是有前提的，那就是元素間的比較動作（採用 `operator==` 或判斷式 *predicate*）並不會丟擲異常。用資料庫編程術語來說，只要你不呼叫賦值操作或 `sort()`，並保證元素相互比較時不丟擲異常，那麼 lists 便可說是「交易安全（**transaction safe**）」。表 6.19 列出異常狀況下提供特殊保證的所有操作函式。STL 異常處理的一般性討論，請見 5.11.2 節, p139。

操作	保證
`push_back()`	如果不成功，就是無任何作用。
`push_front()`	如果不成功，就是無任何作用。
`insert()`	如果不成功，就是無任何作用。
`pop_back()`	不丟擲異常。
`pop_front()`	不丟擲異常。
`erase()`	不丟擲異常。
`clear()`	不丟擲異常。
`resize()`	如果不成功，就是無任何作用。
`remove()`	只要元素比較動作不丟擲異常，它就不丟擲異常。
`remove_if()`	只要判斷式 *predicate* 不丟擲異常，它就不丟擲異常。
`unique()`	只要元素比較動作不丟擲異常，它就不丟擲異常。
`splice()`	不丟擲異常。
`merge()`	只要元素比較時不丟擲異常，它便保證「要不成功，要不無任何作用」。
`reverse()`	不丟擲異常。
`swap()`	不丟擲異常。

表 6.19 Lists 的各種操作在異常發生時提供的特殊保證

6.4.4 Lists 運用實例

下面這個例子突顯出 list 特殊成員函式的用法:

```
// cont/list1.cpp

#include <iostream>
#include <list>
#include <algorithm>
```

```
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout," "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout," "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
         list1.push_back(i);
         list2.push_front(i);
    }
    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(),list2.end(), // destination position
                 3),
                 list1);             // source list
    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(),        // destination position
                 list2,              // source list
                 list2.begin());     // source position
    printLists(list1, list2);

    // sort second list, assign to list1 and remove duplicates
    list2.sort();
    list1 = list2;
```

```
    list2.unique();
    printLists(list1, list2);

    // merge both sorted lists into the first list
    list1.merge(list2);
    printLists(list1, list2);
}
```

程式輸出如下:

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

6.5 Sets 和 Multisets

Set 和 multiset 會根據特定的排序準則，自動將元素排序。兩者不同處在於 multisets 允許元素重複而 sets 不允許（請參考 6.6 節和第 5 章關於本主題的討論）。

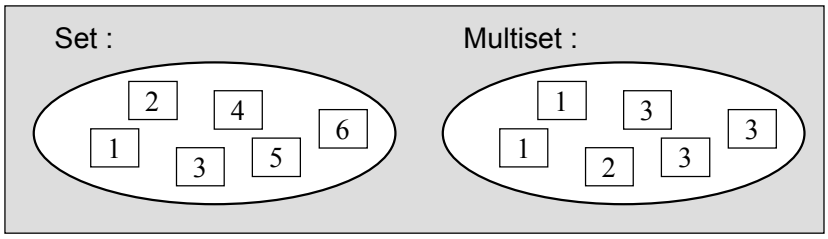


圖 6.6 Sets 和 Multisets

使用 set 或 multiset 之前，必須先含入表頭檔 `<set>`[14]：

```
#include <set>
```

在這個表頭檔中，上述兩個型別都被定義爲命名空間 `std` 內的 class templates：

```
namespace std {
    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class set;

    template <class T,
           class Compare = less<T>,
           class Allocator = allocator<T> >
    class multiset;
}
```

只要是 *assignable*、*copyable*、*comparable*（根據某個排序準則）的型別 `T`，都可以成爲 set 或 multiset 的元素型別。可有可無的第二個 template 引數用來定義排序準則。如果沒有傳入特別的排序準則，就採用預設準則 `less`— 這是一個仿函式，以 `operator<` 對元素進行比較，以便完成排序（`less`[15]的內容詳見 p305）。可有可無的第三引數用來定義記憶體模型（參見第 15 章）。預設的記憶體模型是

[14] 早期 STL 中，sets 的表頭檔是 `<set.h>`，multisets 的表頭檔是 `<mulitiset.h>`

[15] 在不支援 default template arguments 的系統中，第二引數通常會被省略。

allocator，由 C++ 標準程式庫提供[16]。

所謂「排序準則」，必須定義 **strict weak ordering**，其意義如下：

1. 必須是「非對稱的（**antisymmetric**）」。
   對 operator< 而言，如果 x < y 爲真，則 y < x 爲假。
   對判斷式 *predicate* op() 而言，如果 op(x,y) 爲真，則 op(y,x) 爲假。
2. 必須是「可遞移的（**transitive**）」。
   對 operator< 而言，如果 x < y 爲真且 y < z 爲真，則 x < z 爲真。
   對判斷式 op() 而言，如果 op(x,y) 爲真且 op(y,z) 爲真，則 op(x,z) 爲真。
3. 必須是「非反身的（**irreflexive**）」。
   對 operator< 而言，x < x 永遠爲假。
   對判斷式 *predicate* op() 而言，op(x,x) 永遠爲假。

基於這些特性，排序準則也可用於相等性檢驗，也就是說，如果兩個元素都不小於對方（或說 op(x,y) 和 op(y,x) 都爲假），則兩個元素相等。

譯註：以上種種性質（及其他各種相關性質）是 STL 學術理論的一部分。STL 先建立起一個抽象概念階層體系，形成一個軟體組件分類學，最後再以實際工具（C++ template）將各個概念實作出來。這些理論架構的最佳描述書籍是《*Generic Programming and the STL*》,by Matthew H. Austern, Addison Wesley 1998；中譯本《泛型程式設計與 STL》，侯捷/黃俊堯合譯，碁峰 2000。

6.5.1 Sets 和 Multisets 的能力

和所有標準關聯式容器類似，sets 和 multisets 通常以平衡二元樹（balanced binary tree，圖 6.7）完成。C++ 標準規格書並未明定之，但由 sets 和 multisets 各項操作的複雜度可以得出這樣的結論[17]。

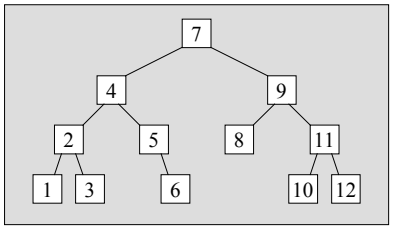


圖 6.7 Sets 和 Multisets 的內部結構

[16] 在不支援 default template arguments 的系統中，第三引數通常會被省略。

[17] 事實上 sets 和 multisets 通常以紅黑樹（red-black tree）實作而成。紅黑樹在改變元素數量和元素搜尋方面都很出色，它保證節點安插時最多只會作兩個重新鏈結（relink）動作，而且到達某一元素的最長路徑深度，最多只是最短路徑深度的兩倍。

自動排序的主要優點在於使二元樹於搜尋元素時具有良好性能。其搜尋函式演算法具有對數（logarithmic）複雜度。在擁有 1000 個元素的 sets 或 multisets 中搜尋元素，二元樹搜尋動作（由成員函式執行）的平均時間爲線性搜尋（由 STL 演算法執行）的 1/50。關於複雜度的討論，詳見 p21, 2.3 節。

但是，自動排序造成 sets 和 multisets 的一個重要限制：你不能直接改變元素值，因爲這樣會打亂原本正確的順序。因此，要改變元素值，必須先刪除舊元素，再插入新元素。這裡提供的介面正反映了這種行爲：

- sets 和 multisets 不提供用來直接存取元素的任何操作函式。
- 通過迭代器進行元素間接存取，有一個限制：從迭代器的角度來看，元素值是常數。

6.5.2 Sets 和 multisets 的操作函式

生成（Create）、複製（Copy）和銷毀（Destroy）

表 6.20 列出 sets 和 multisets 的建構式和解構式。

操作	效果
set `c`	產生一個空的 set/multiset，其中不含任何元素。
set `c(op)`	以 `op` 爲排序準則，產生一個空的 set/multiset。
set `c1(c2)`	產生某個 set/multiset 的副本，所有元素均被複製。
set `c(beg,end)`	以區間 `[beg;end)` 內的元素產生一個 set/multiset。
set `c(beg,end,op)`	以 `op` 爲排序準則，利用 `[beg;end)` 內的元素生成一個 set/multiset
`c.~`***set***`()`	銷毀所有元素，釋放記憶體。

其中 ***set*** 可爲下列形式：

set	效果
`set<Elem>`	一個 set，以 `less<>`（`operator<`）爲排序準則
`set<Elem,Op>`	一個 set，以 `op` 爲排序準則
`multiset<Elem>`	一個 multiset，以 `less<>`（`operator<`）爲排序準則
`multiset<Elem,Op>`	一個 multiset，以 `op` 爲排序準則

表 6.20　Sets 和 Multisets 的建構式和解構式

有兩種方式可以定義排序準則：

1. 以 template 參數定義之。
   例如[18]：
   ```
   std::set<int, std::greater<int> > coll;
   ```
   這種情況下，排序準則就是型別的一部分。因此型別系統確保「只有排序準則相同的容器才能被合併」。這是排序準則的通常指定法。更精確地說，第二參數是排序準則的**型別**，實際的排序準則是容器所產生的函式物件（*function object*，或稱 *functor*）。爲了產生它，容器建構式會呼叫「排序準則型別」的 ***default*** 建構式。p294 有一個「使用者自定之排序準則」的運用實例。

2. 以建構式參數定義之。
   這種情況下，同一個型別可以運用不同的排序準則，而排序準則的初始值或狀態也可以不同。如果執行期才獲得排序準則，而且需要用到不同的排序準則（但資料型別必須相同），此一方式可派上用場。p191 有個完整例子。

如果使用者沒有提供特定的排序準則，那麼就採用預設準則 — 仿函式 `less<>`。`less<>` 係透過 `operator<` 對元素進行排序[19]。

請注意，排序準則也被用於元素相等性檢驗工作。當採用預設排序準則時，兩元素的相等性檢驗語句如下：

```
if (! (elem1 < elem2 || elem2 < elem1))
```

這樣做有三點好處：

1. 只需傳遞一個引數作爲排序準則。
2. 不必針對元素型別提供 `operator==`。
3. 你可以對「相等性」有截然相反的定義（即使算式中 `operator==` 的行爲有所不同，也無關緊要）。不過當心造成混淆。

這種相等性檢驗方式會花費比較長的時間，因爲評估上述算式可能需要兩次比較。注意，如果第一次比較結果爲 true，就不用進行第二次比較了。

看到這裡，如果容器型別名稱讓你很煩，採用「型別定義式」不失爲一個好辦法。在使用容器型別（以及迭代器型別）的任何地方都可以採用這種便捷之道，例如：

---

[18] 注意，兩個 `>` 之間需加上一個空格，因爲 `>>` 會被編譯器視爲移位運算子，導致本處語法錯誤。

[19] 在不支援 default template parameters 的系統中，通常必須這麼設定排序準則：
```
set<int,less<int> > coll;
```

```
typedef std::set<int,std::greater<int> > IntSet;
...
IntSet coll
IntSet::iterator pos;
```

「利用區間的起點和終點來建構容器」的建構式，可以從其他型別的容器中，或是從 array、或是從標準輸入裝置（standard input）中接受元素來源。詳見 6.1.2 節, p144。

非變動性操作（Nonmodifying Operations）

Sets 和 multisets 提供常見的非變動性操作，用來查詢大小、相互比較。

操作	效果
`c.size()`	傳回容器的大小
`c.empty()`	判斷容器大小是否爲零。等同於 `size()==0`，但可能更快
`c.max_size()`	傳回可容納的最大元素數量
`c1 == c2`	判斷是否 `c1` 等於 `c2`
`c1 != c2`	判斷是否 `c1` 不等於 `c2`。等同於 `!(c1 == c2)`。
`c1 < c2`	判斷是否 `c1` 小於 `c2`
`c1 > c2`	判斷是否 `c1` 大於 `c2`。等同於 `c2 < c1`。
`c1 <= c2`	判斷是否 `c1` 小於等於 `c2`。等同於 `!(c2 < c1)`。
`c1 >= c2`	判斷是否 `c1` 大於等於 `c2`。等同於 `!(c1 < c2)`。

表 6.21　Sets 和 Multisets 的非變動性操作（Nonmodifying Operations）

元素比較動作只能用於型別相同的容器。換言之，元素和排序準則必須有相同的型別，否則編譯時期會產生型別方面的錯誤。

```
std::set<float> c1;   // sorting criterion: std::less<>
std::set<float,std::greater<float> > c2;
...
if (c1 == c2) {       // ERROR: different types
   ...
}
```

比較動作係以「字典（lexicographical）順序」來檢查某個容器是否小於另一個容器。如果要比較不同型別（擁有不同排序準則）的容器，你必須採用 p356, 9.5.4 節的「比較演算法（comparing algorithms）」。

特殊的搜尋函式（Special Search Operations）

Sets 和 multisets 在元素快速搜尋方面有最佳化設計，所以提供了特殊的搜尋函式，如表 6.22。這些函式是同名的 STL 演算法的特殊版本。面對 sets 和 multisets，你應該優先採用這些最佳化演算法，如此可獲得對數複雜度，而非 STL 演算法的線性複雜度。舉個例子，在 1,000 個元素中搜尋，平均 10 次比較之後便可得出結果，如果是線性複雜度，平均 500 次比較才能有結果（參見 2.3 節, p21）。

操作	效果
`count(elem)`	傳回「元素值爲 `elem`」的元素個數。
`find(elem)`	傳回「元素值爲 `elem`」的第一個元素，如果找不到就傳回 `end()`。
`lower_bound(elem)`	傳回 `elem` 的第一個可安插位置，也就是「元素值 `>= elem`」的第一個元素位置。
`upper_bound(elem)`	傳回 `elem` 的最後一個可安插位置，也就是「元素值 `> elem`」的第一個元素位置。
`equal_range(elem)`	傳回 `elem` 可安插的第一個位置和最後一個位置，也就是「元素值 `== elem`」的元素區間。

表 6.22　Sets 和 Multisets 的搜尋操作函式

成員函式 `find()` 搜尋出與引數值相同的第一個元素，並傳回一個迭代器，指向該位置。如果沒找到這樣的元素，就傳回容器的 `end()`。

`lower_bound()` 和 `upper_bound()` 分別傳回元素可安插點的第一個和最後一個位置。換言之，`lower_bound()` 傳回大於等於引數值的第一個元素所處位置，`upper_bound()` 傳回大於引數值的第一個元素位置。`equal_range()` 則是將 `lower_bound()` 和 `upper_bound()` 的回返值做成一個 `pair` 傳回（型別 `pair` 在 p33, 4.1 節介紹），所以它傳回的是「與引數值相等」的元素所形成的區間。如果 `lower_bound()` 或「`equal_range()` 的第一值」等於「`equal_range()` 的第二值」或 `upper_bound()`，則此 sets 或 multisets 內不存在相同數值的元素。這是當然啦，同值區間中至少也得包含一個元素嘛！

下面例子說明如何使用 `lower_bound()`, `upper_bound()` 和 `equal_range()`：

```
// cont/set2.cpp

#include <iostream>
#include <set>
using namespace std;
```

```
int main ()
{
  set<int> c;

  c.insert(1);
  c.insert(2);
  c.insert(4);
  c.insert(5);
  c.insert(6);

  cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
  cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
  cout << "equal_range(3): " << *c.equal_range(3).first << " "
                             << *c.equal_range(3).second << endl;

  cout << endl;
  cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
  cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
  cout << "equal_range(5): " << *c.equal_range(5).first << " "
                             << *c.equal_range(5).second << endl;
}
```

程式輸入如下：

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4

lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

上例如果使用 multisets 而不是 sets，程式輸出相同。

指派（賦值, Assignments）

sets 和 multisets 只提供所有容器都提供的基本賦值操作（表 6.23），詳見 p147。

這些操作函式中，賦值操作的兩端容器必須具有相同型別。儘管「比較準則」本身可能不同，但其型別必須相同。p191 列出一個「排序準則不同，但型別相同」的例子。如果準則不同，準則本身也會被指派（assigned）或交換（swapped）。

操作	效果
`c1 = c2`	將 `c2` 中所有元素指派給 `c1`
`c1.swap(c2)`	將 `c1` 和 `c2` 的元素互換。
`swap(c1,c2)`	同上。此爲全域函式。

表 6.23 Sets 和 Multisets 的指派（賦值）操作

迭代器相關函式（Iterator Functions）

Sets 和 multisets 不提供元素直接存取，所以只能採用迭代器。Sets 和 multisets 也提供了一些常見的迭代器函式（表 6.24）。

操作	效果
`c.begin()`	傳回一個雙向迭代器（將元素視爲常數），指向第一元素
`c.end()`	傳回一個雙向迭代器（將元素視爲常數），指向最後元素的下一位置
`c.rbegin()`	傳回一個逆向迭代器，指向逆向巡訪時的第一個元素
`c.rend()`	傳回一個逆向迭代器，指向逆向巡訪時的最後元素的下一位置

表 6.24 Sets 和 multisets 的迭代器相關操作函式

和其他所有關聯式容器類似，這裏的迭代器是雙向迭代器（參見 p255, 7.2.4 節）。所以，對於只能用於隨機存取迭代器的 STL 演算法（例如排序或隨機亂序 *random shuffling* 演算法），sets 和 multisets 就無福消受了。

更重要的是，對迭代器操作而言，所有元素都被視爲常數，這可確保你不會人爲改變元素值，從而打亂既定順序。然而這也使得你無法對 sets 或 multisets 元素呼叫任何變動性演算法（modifying algorithms）。例如你不能對它們呼叫 `remove()`，因爲 `remove()` 演算法實際上是以一個引數值覆蓋被移除的元素（詳細討論見 p115, 5.6.2 節）。如果要移除 sets 和 multisets 的元素，你只能使用它們所提供的成員函式。

元素的安插（Inserting）和移除（Removing）

表 6.25 列出 sets 和 multisets 的元素安插和刪除函式。

按 STL 慣例，你必須保證引數有效：迭代器必須指向有效位置、序列起點不能位於終點之後、不能從空容器中刪除元素。

安插和移除多個元素時，單一呼叫（一次處理）比多次呼叫（逐一處理）快得多。

操作	效果
`c.insert(elem)`	安插一份 `elem` 副本，傳回新元素位置（不論是否成功 — 對 sets 而言）
`c.insert(pos,elem)`	安插一份 `elem` 副本，傳回新元素位置（`pos` 是個提示，指出安插動作的搜尋起點。如果提示恰當，可大大加快速度）
`c.insert(beg,end)`	將區間 `[beg;end)` 內所有元素的副本安插到 `c`（無回返值）
`c.erase(elem)`	移除「與 `elem` 相等」的所有元素，傳回被移除的元素個數
`c.erase(pos)`	移除迭代器 `pos` 所指位置上的元素，無回返值。
`c.erase(beg,end)`	移除區間 `[beg;end)` 內的所有元素，無回返值。
`c.clear()`	移除全部元素，將整個容器清空。

表 6.25　Sets 和 Multisets 的元素安插和移除

注意，安插函式的回返值型別不盡相同：

- Sets 提供如下介面：

```
pair<iterator,bool> insert(const value_type& elem);
iterator            insert(iterator pos_hint,
                           const value_type& elem);
```

- Multisets 提供如下介面：

```
iterator insert(const value_type& elem);
iterator insert(iterator pos_hint,
                const value_type& elem);
```

回返值型別不同的原因是：multisets 允許元素重複，而 sets 不允許。因此如果將某元素安插至一個 set 內，而該 set 已經內含同值元素，則安插動作將告失敗。所以 set 的回返值型別是以 `pair` 組織起來的兩個值（關於 `pair` 詳見 p33, 4.1 節）：

1. pair 結構中的 `second` 成員表示安插是否成功。
2. pair 結構中的 `first` 成員傳回新元素的位置，或傳回現存的同值元素的位置。

其他任何情況下，函式都傳回新元素位置（如果 sets 已經內含同值元素，則傳回同值元素的位置）。

以下例子把數值 3.3 的元素安插到 set `c` 中，藉此說明如何使用上述介面：

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```

如果你還想處理新位置或舊位置，程式碼得更複雜些：

```
// define variable for return value of insert()
std::pair<std::set<float>::iterator,bool> status;

// insert value and assign return value
status = c.insert(value);

// process return value
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1
          << std::endl;
```

對於此一序列的兩次呼叫結果可能如下：

```
8.9 inserted as element 4
7.7 already exists as element 3
```

注意，所有擁有「位置提示參數」的安插函式，其回返值型別都一樣，不論是 sets 或 multisets，這些函式都只傳回一個迭代器。這些函式的效果與「無位置提示參數」的函式一樣，只不過性能略有差異。你可以傳進一個迭代器，該位置將作爲一個提示，用來提昇性能。事實上如果被安插元素的位置恰好緊貼於提示位置之後，那麼時間複雜度就會從「對數」一變而爲「分期攤還常數（amortized constant）」（複雜度的介紹請見 p21, 2.3 節）。和「單引數安插函式」不同的是，帶有「額外提示位置」的若干安插函式，都具有相同的回返值型別，這就確保你至少有了一個通用型安插函式，在各種容器中有共同介面。事實上通用型安插迭代器（*general inserter*s）就是靠這個介面的支援才得以實現。

要刪除「與某值相等」的元素，只需呼叫 `erase()`：

```
std::set<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

和 lists 不同的是，`erase()`並非取名爲 `remove()`（後者的討論請見 p170）。是的，它的行爲不同，它傳回被刪除元素的個數，用在 sets 身上，回返值非 0 即 1。

如果 multisets 內含重複元素，你不能使用 `erase()`來刪除這些重複元素中的第一個。你可以這麼做：

```
std::multiset<Elem> coll;
...
// remove first element with passed value
std::multiset<Elem>::iterator pos;
pos = coll.find (elem);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

這裏應該採用成員函式 `find()`，而非 STL 演算法 `find()`，因爲前者速度更快（參見 p154 的例子）。

注意，還有一個回返值不一致的情況。作用於序列式容器和關聯式容器的 `erase()` 函式，其回返值有以下不同：

1. 序列式容器提供下面的 `erase()`成員函式：
   ```
   iterator erase(iterator pos);
   iterator erase(iterator beg, iterator end);
   ```
2. 關聯式容器提供下面的 `erase()`成員函式：
   ```
   void erase(iterator pos);
   void erase(iterator beg, iterator end);
   ```

存在這種差別，完全是爲了性能。在關聯式容器中「搜尋某元素並傳回後繼元素」可能頗爲耗時，因爲這種容器的底部是以二元樹完成，所以如果你想編寫對所有容器都適用的程式碼，你必須忽略回返值。

6.5.3 異常處理（Exception Handling）

Sets 和 multisets 是「以節點（nodes）爲基礎」的容器。如果節點建構失敗，容器仍保持原樣。此外，由於解構式通常並不拋擲異常，所以節點的移除不可能失敗。

然而，面對多重元素安插動作，「保持元素次序」這一條件會造成「異常拋出時能夠完全復原」這一需求變得不切實際。因此只有「單一元素安插動作」才支援「成功，否則無效」的操作原則。至於「多元素刪除動作」總是能夠成功。如果排序準則之複製/賦值動作會丟擲異常，則 `swap()`也會丟擲異常。

STL 異常處理的一般性討論見於 p139, 5.11.2 節。p248 的 6.10.10 節列出「異常出現時會給予特殊保證」的所有容器操作函式。

6.5.4 Sets 和 multisets 運用實例

以下程式展示 sets 的一些能力[20]：

```
// cont/set1.cpp

#include <iostream>
#include <set>
using namespace std;

int main()
{
    /* type of the collection:sets：
    * - no duplicates
    * - elements are integral values
    * - descending order
    */
    typedef set<int,greater<int> > IntSet;

    IntSet coll1; // empty set container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
         cout << *pos << ' ';
    }
    cout << endl;
```

[20] distance()的定義已有改變。早期 STL 版本中，你必須含入 distance.hpp（見 p263）

```
    // insert 4 again and process return value
    pair<IntSet::iterator,bool> status = coll1.insert(4);
    if (status.second) {
       cout << "4 inserted as element "
            << distance(coll1.begin(),status.first) + 1
            << endl;
    }
    else {
       cout << "4 already exists" << endl;
    }

    // assign elements to another set with ascending order
    set<int> coll2(coll1.begin(),
             coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

首先，以下的型別定義：

```
typedef set<int,greater<int> > IntSet;
```

定義了一個 set，其中容納降冪（遞減）排列的 ints。產生一個空的 sets 之後，首先利用 insert()安插數個元素：

```
IntSet coll1;

coll1.insert(4);
...
```

注意數值爲 5 的元素被安插兩次，但第二次安插動作會被程式忽略，因爲 sets 不允許數值重複的元素。

列印所有元素後，程式再次安插元素 4。這次按照 p183 的方法來處理 `insert()`回返值。

以下述句：

```
set<int> coll2(coll1.begin(),coll1.end());
```

產生一個新的 set，其中容納昇冪（遞增）排列的 `ints`，並以原本那個 sets 的元素做爲初值[21]。

兩個容器有不同的排序準則，所以它們的型別不同，不能直接相互指派（賦值）或比較。但只要元素型別相同或彼此可以轉型，你就可以使用某些「有能力處理不同容器型別」的演算法來達成目的。

以下述句：

```
coll2.erase (coll2.begin(), coll2.find(3));
```

移除了數值爲 3 的元素之前的所有元素。注意數值爲 3 的元素位於序列尾端，所以沒被移除。

最後，所有數值爲 5 的元素都被移除：

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

程式輸出如下：

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

[21] 這行述句需要兩個語言新性質：member templates 和 default template arguments。如果系統不支援，你必須改成這樣：

```
set<int,less<int> > coll2;
copy (coll1.begin(), coll1.end(),
      inserter(coll2,coll2.begin()));
```

對於 multisets，上一個程式需要些微改變，並產生不同結果：

```
// cont/mset1.cpp

#include <iostream>
#include <set>
using namespace std;

int main()
{
    /* type of the collection: sets
    * - duplicates allowed
    * - elements are integral values
    * - descending order
    */
    typedef multiset<int,greater<int> > IntSet;

    IntSet coll1; // empty multiset container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
         cout << *pos << ' ';
    }
    cout << endl;

    // insert 4 again and process return value
    IntSet::iterator ipos = coll1.insert(4);
    cout << "4 inserted as element "
         << distance(coll1.begin(),ipos) + 1
         << endl;
```

```
    // assign elements to another multiset with ascending order
    multiset<int> coll2(coll1.begin(),
                        coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
    ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

這個程式把所有的 set 都改爲 multiset，此外 `insert()`回返值的處理也有所不同：

```
IntSet::iterator ipos = coll1.insert(4);
cout << "4 inserted as element "
     << distance(coll1.begin(),ipos) + 1
     << endl;
```

由於 multisets 可能包含重複元素，所以安插動作只在異常被拋出時才失敗。因此，回返值型別只是一個迭代器，指向新元素位置。

程式輸出如下：

```
6 5 5 4 3 2 1
4 inserted as element 5
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6
```

6.5.5 執行期指定排序準則

無論是將排序準則作爲第二個 template 引數傳入，或是採用預設的排序準則 `less<>`，通常你都會將排序準則定義爲型別的一部分。但有時必須在執行期處理排序準則，或者你可能需要對同一種資料型別採用不同的排序準則。此時你就需要一個「用來表現排序準則」的特殊型別，使你能夠在執行期間傳遞某個準則。以下範例程式說明了這種做法：

```
// cont/setcmp.cpp

#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// type for sorting criterion
template <class T>
class RuntimeCmp {
    public:
        enum cmp_mode {normal, reverse};

    private:
        cmp_mode mode;

    public:
        // constructor for sorting criterion
        // - default criterion uses value normal
        RuntimeCmp (cmp_mode m=normal) : mode(m) {
        }

        // comparision of elements
        bool operator() (const T& t1, const T& t2) const {
            return mode == normal ? t1 < t2 : t2 < t1;
        }

        // comparision of sorting criteria
        bool operator== (const RuntimeCmp& rc) {
            return mode == rc.mode;
    }
};

// type of a set that uses this sorting criterion
typedef set<int,RuntimeCmp<int> > IntSet;
```

```
// forward declaration
void fill (IntSet& set);

int main()
{
   // create, fill, and print set with normal element order
   // - uses default sorting criterion
   IntSet coll1;
   fill(coll1);
   PRINT_ELEMENTS (coll1, "coll1: ");

   // create sorting criterion with reverse element order
   RuntimeCmp<int> reverse_order(RuntimeCmp<int>::reverse);

   // create, fill, and print set with reverse element order
   IntSet coll2(reverse_order);
   fill(coll2);
   PRINT_ELEMENTS (coll2, "coll2: ");

   // assign elements AND sorting criterion
   coll1 = coll2;
   coll1.insert(3);
   PRINT_ELEMENTS (coll1, "coll1: ");

   // just to make sure...
   if (coll1.value_comp() == coll2.value_comp()) {
      cout << "coll1 and coll2 have same sorting criterion"
           << endl;
   }
   else {
      cout << "coll1 and coll2 have different sorting criterion"
           << endl;
   }
}

void fill (IntSet& set)
{
   // fill insert elements in random order
   set.insert(4);
   set.insert(7);
   set.insert(5);
   set.insert(1);
   set.insert(6);
   set.insert(2);
   set.insert(5);
}
```

在這個程式中，RuntimeCmp<> 是一個簡單的 template，提供「執行期間面對任意型別定義一個排序準則」的泛化能力。其 *default* 建構式採用預設值 normal，按升冪排序；你也可以將 RuntimeCmp<>::reverse 傳遞給建構式，便能按降冪排序。

程式輸出入下：

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have same sorting criterion
```

注意，coll1 和 coll2 擁有相同型別，該型別即 fill() 函式的參數型別。再請注意，*assignment* 運算子不僅指派了元素，也指派了排序準則（否則任何一個指派動作豈不輕易危及排序準則！）

6.6 Maps 和 Multimaps

Map 和 multimap 將 *key/value* pair（鍵值/實值 對組）當作元素，進行管理。它們可根據 *key* 的排序準則自動將元素排序。multimaps 允許重複元素，maps 不允許，見圖 6.8。

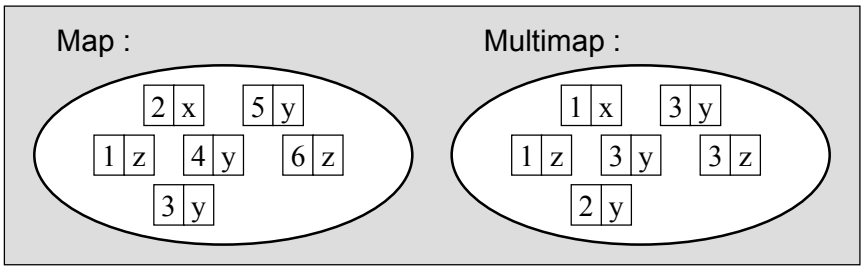


圖 6.8 Maps 和 Multimaps

使用 map 和 multimap 之前，你必須先含入表頭檔 `<map>`[22]:

```
#include <map>
```

在其中，map 和 multimap 被定義爲命名空間 `std` 內的 class templates：

```
namespace std {
    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class map;

    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class multimap;
}
```

第一個 template 引數被當做元素的 *key*，第二個 template 引數被當做元素的 *value*。Map 和 multimap 的元素型別 `Key` 和 `T`，必須滿足以下兩個條件：

[22] 在早期 STL 中，maps 被定義於 `<map.h>` 而 mltimaps 被定義於 `<multimap.h>`。

1. *key/value* 必須具備 ***assignable***（可賦值的）和 ***copyable***（可複製的）性質。
2. 對排序準則而言，*key* 必須是 ***comparable***（可比較的）。

第三個 template 引數可有可無，用來定義排序準則。和 sets 一樣，這個排序準則必須定義爲 **strict weak ordering**（參見 p176）。元素的次序由它們的 *key* 決定，和 *value* 無關。排序準則也可以用來檢查相等性：如果兩個元素的 *key* 彼此都不小於對方，則兩個元素被視爲相等。如果使用者未傳入特定排序準則，就使用預設的 `less` 排序準則 — 以 `operator<` 來進行比較[23]（`less` 的詳細資料請見 p305）。

第四個 template 引數也是可有可無，用來定義記憶體模型（詳見第 15 章）。預設的記憶體模型是 `allocator`，由 C++ 標準程式庫提供[24]。

6.6.1 Maps 和 Multimaps 的能力

和所有標準的關聯式容器一樣，maps/multimpas 通常以平衡二元樹完成，如圖 6.9。

標準規格書並未明定這一點，但是從 map 和 multimap 各項操作的複雜度可以得出這一結論。典型情況下，set, multisets, map, multimaps 使用相同的內部資料結構。因此你可以把 set 和 multisets 分別視爲特殊的 map 和 multimaps，只不過 sets 元素的 *value* 和 *key* 是指同一物件。因此 map 和 multimaps 擁有 set 和 multisets 的所有能力和所有操作函式。當然某些細微差異還是有的：首先，它們的元素是 *key/value* pair，其次，map 可作爲關聯式陣列來運用。

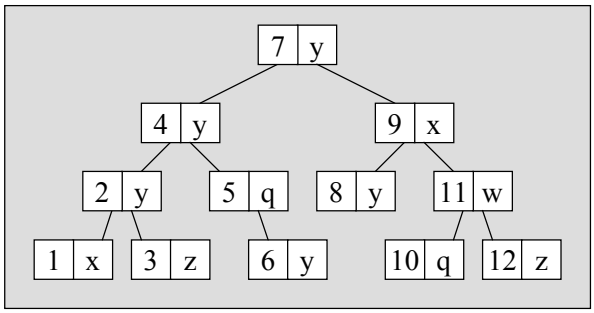


圖 6.9　Maps 和 Multimaps 的內部結構

[23] 在不支援 default template parameters 的系統中，第三個參數通常會被省略。

[24] 在不支援 default template parameters 的系統中，第四個參數通常會被省略。

Map 和 multimaps 根據元素的 *key* 自動對元素進行排序。這麼一來，根據已知的 *key* 搜尋某個元素時，就能夠有很好的性能，而根據已知 *value* 搜尋元素時，性能就很糟糕。「自動排序」這一性質使得 map 和 multimaps 身上有了一條重要的限制：你不可以直接改變元素的 *key*，因爲這會破壞正確次序。要修改元素的 *key*，你必須先移除擁有該 *key* 的元素，然後插入擁有新的 *key*/*value* 的元素（詳見 p201）。從迭代器的觀點來看，元素的 *key* 是常數。至於元素的 *value* 倒是可以直接修改，當然，前提是 *value* 並非常數型態。

6.6.2 Map 和 Multimap 的操作

生成（Create）、複製（Copy）和銷毀（Destroy）

表 6.26 列出 maps 和 multimaps 的生成、複製、銷毀等各項操作。

操作	效果
map `c`	產生一個空的 map/multimap，其中不含任何元素。
map `c(op)`	以 `op` 爲排序準則，產生一個空的 map/multimap。
map `c1(c2)`	產生某個 map/multimap 的副本，所有元素均被複製。
map `c(beg,end)`	以區間 `[beg;end)` 內的元素產生一個 map/multimap。
map `c(beg,end,op)`	以 `op` 爲排序準則，利用 `[beg;end)` 內的元素生成一個 map/multimap。
`c.~`***map***`()`	銷毀所有元素，釋放記憶體。

其中，***map*** 可爲下列型式：

map	效果
`map<Key,Elem>`	一個 map，以 `less<>`（`operator<`）爲排序準則。
`map<Key,Elem,Op>`	一個 map，以 `op` 爲排序準則。
`multimap<Key,Elem>`	一個 multimap，以 `less<>`（`operator<`）爲排序準則。
`multimap<Key,Elem,Op>`	一個 multimap，以 `op` 爲排序準則。

表 6.26　Maps 和 Multimaps 的建構式和解構式

有兩種方式可以定義排序準則：

1. 以 template 引數定義之。

   例如[25]：

   ```
   std::map<float, std::string, std::greater<float> > coll;
   ```

   這種情況下，排序準則就是型別的一部分。因此型別系統確保「只有排序準則相同的容器才能被合併」。這是比較常見的排序準則指定法。更精確地說，第三參數是排序準則的**型別**。實際的排序準則是容器所產生的函式物件（*function object*，或稱 *functor*）。爲了產生它，容器建構式會呼叫「排序準則型別」的 ***default*** 建構式。p294 有一個「使用者自定之排序準則」的運用實例。

2. 以建構式參數定義之。

   在這種情況下，你可以有一個「排序準則型別」並爲它指定不同的排序準則（也就是說讓該型別所產生出來的物件（代表一個排序準則）的初值或狀態不同）。如果執行期才獲得排序準則，而且程式需要用到不同的排序準則（但其資料型別必須相同），此一方式可派上用場。一個典型的例子是在執行期指定「*key* 的型別爲 `string`」的排序準則。完整例子見 p213。

如果使用者沒有提供特定排序準則，就採用預設準則 — 仿函式 `less<>`。`less<>` 係透過 `operator<` 對元素進行排序[26]。

你應當做一些型別定義（typedef），從而簡化繁瑣的型別表示式：

```
typedef std::map<std::string,float,std::greater<std::string> >
                 StringFloatMap;
...
StringFloatMap coll;
```

某些建構式使用區間起點和終點作爲引數，它們可以使用不同型別的容器、array、標準輸入裝置（standard input）來進行初始化，詳見 6.1.2 節, p144。然而由於元素是 *key/value* pair，因此你必須確定來自源區間的元素型別也是 `pair<`*key, value* `>`，或至少可轉化成 `pair<`*key, value*`>`。

非變動性操作（Nonmodifying Operations）

map 和 multimaps 提供常見的非變動性操作，用來查詢大小、相互比較。如表 6.27。

---

[25] 注意，兩個 `>` 之間需加上一個空格，因爲 `>>` 會被編譯器視爲移位運算子，導致本處語法錯誤。

[26] 在不支援 default template parameters 的系統中，通常必須這麼設定排序準則：

```
map<float, string, less<float> > coll;
```

操作	效果
`c.size()`	傳回容器的大小。
`c.empty()`	判斷容器大小是否爲零。等同於 `size()==0`，但可能更快。
`c.max_size()`	傳回可容納的最大元素數量。
`c1 == c2`	判斷是否 `c1` 等於 `c2`。
`c1 != c2`	判斷是否 `c1` 不等於 `c2`。等同於 `!(c1 == c2)`。
`c1 < c2`	判斷是否 `c1` 小於 `c2`。
`c1 > c2`	判斷是否 `c1` 大於 `c2`。等同於 `c2 < c1`。
`c1 <= c2`	判斷是否 `c1` 小於等於 `c2`。等同於 `!(c2 < c1)`。
`c1 >= c2`	判斷是否 `c1` 大於等於 `c2`。等同於 `!(c1 < c2)`。

表 6.27 Maps 和 Multimaps 的非變動性操作（Nonmodifying Operations）

元素比較動作只能用於型別相同的容器。換言之，容器的 *key*、*value*、排序準則都必須有相同的型別，否則編譯期會產生型別方面的錯誤。例如：

```
std::map<float,std::string> c1;      // sorting criterion: less<>
std::map<float,std::string,std::greater<float> > c2;
...
if (c1 == c2) { // ERROR: different types
    ...
}
```

比較動作係以「字典（lexicographical）順序」來檢查某個容器是否小於另一個容器（詳見 p360）。如果要比較不同型別（擁有不同排序準則）的容器，你必須採用 p356, 9.5.4 節的「比較演算法（comparing algorithms）」。

特殊的搜尋動作（Special Search Operations）

就像 set 和 multisets 一樣，map 和 multimaps 也提供特殊的搜尋函式，以便利用內部樹狀結構獲取較好的性能，見表 6.28。

成員函式 `find()` 用來搜尋擁有某個 *key* 的第一個元素，並傳回一個迭代器，指向該位置。如果沒找到這樣的元素，就傳回容器的 `end()`。你不能以 `find()` 搜尋擁有某特定 *value* 的元素，你必須改用通用演算法如 `find_if()`，或乾脆寫一個顯式迴圈。下面這個例子便是利用一個簡單迴圈，對擁有特定 *value* 的所有元素進行某項操作:

```
std::multimap<std::string,float> coll;
...
// do something with all elements having a certain value
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
```

操作	效果
`count(key)`	傳回「鍵值等於 `key`」的元素個數
`find(key)`	傳回「鍵值等於 `key`」的第一個元素，找不到就傳回 `end()`。
`lower_bound(key)`	傳回「鍵值爲 `key`」之元素的第一個可安插位置，也就是「鍵值 `>= key`」的第一個元素位置。
`upper_bound(key)`	傳回「鍵值爲 `key`」之元素的最後一個可安插位置，也就是「鍵值 `> key`」的第一個元素位置。
`equal_range(key)`	傳回「鍵值爲 `key`」之元素的第一個可安插位置和最後一個可安插位置，也就是「鍵值 `== key`」的元素區間。

表 6.28　Maps 和 Multimaps 的特殊搜尋操作函式

```
    if (pos->second == value) {
       do_something();
    }
 }
```

當你要以此類迴圈來移除元素時，請特別當心。因爲可能會發生一些意料之外的事。細節詳見 p204。如果使用 `find_if()`演算法做類似的搜尋動作，會比寫一個迴圈更複雜，因爲你必須提供仿函式（*functor*，亦即函式物件 *function object*），將元素的 *value* 拿來和某個 *value* 比較。詳見 p211 實例。

至於 `lower_bound()`, `upper_bound()`, `equal_range()`，其行爲和 sets（見 p180）的相應函式十分相似，唯一的不同就是：元素是個 *key*/*value* pair。

指派（賦值, Assignments）

Maps 和 multimaps 只支援所有容器都提供的基本指派操作（表 6.29），詳見 p147。

操作	效果
`c1 = c2`	將 `c2` 中所有元素指派給 `c1`
`c1.swap(c2)`	將 `c1` 和 `c2` 的元素互換。
`swap(c1,c2)`	同上。此爲全域函式。

表 6.29　Maps 和 Multimaps 的指派（賦值）操作

這些操作函式中，賦值動作的兩端容器必須具有相同型別。儘管「比較準則」本身可能不同，但其型別必須相同。p213 列出一個「排序準則不同，但型別相同」的例子。如果準則不同，準則本身也會被指派（assigned）或交換（swapped）。

迭代器函式（Iterator Functions）和元素存取（Element Access）

Map 和 multimaps 不支援元素直接存取，因此元素的存取通常是經由迭代器進行。不過有個例外：map 提供 *subscript*（下標）運算子，可直接存取元素，詳見 6.6.3 節, p205。表 6.30 列出 maps 和 multimaps 所支援的迭代器相關函式。

操作	效果
`c.begin()`	傳回一個雙向迭代器（*key* 被視爲常數），指向第一元素
`c.end()`	傳回一個雙向迭代器（*key* 被視爲常數），指向最後元素的下一位置
`c.rbegin()`	傳回一個逆向迭代器，指向逆向巡訪時的第一個元素
`c.rend()`	傳回一個逆向迭代器，指向逆向巡訪時的最後元素的下一位置

表 6.30 Maps 和 Multimaps 的迭代器相關操作函式

和其他所有關聯式容器類似，這裏的迭代器是雙向迭代器（參見 p255, 7.2.4 節）。所以，對於只能用於隨機存取迭代器的 STL 演算法（例如排序或隨機亂序演算法 *random shuffling*），maps 和 multimaps 就無福消受了。

更重要的是，在 map 和 multimap 中，所有元素的 *key* 都被視爲常數。因此元素的實質型別是 `pair<const key,T>`。這個限制是爲了確保你不會因爲變更元素的 *key* 而破壞業已排好的元素次序。所以你不能針對 map 或 multimap 呼叫任何變動性演算法（modifying algorithms）。例如你不能對它們呼叫 `remove()`，因爲 `remove()` 演算法實際上是以一個引數值覆蓋被移除的元素（詳細討論見 p115，5.6.2 節）。如果要移除 map 和 multimaps 的元素，你只能使用它們所提供的成員函式。

下面是 map 迭代器運用實例：

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

其中迭代器 `pos` 巡訪了 `string/float` pair 所組成的序列。以下算式：

```
pos->first
```

獲得元素的 *key*，而以下算式：

```
pos->second
```

獲得元素的 *value*[27]。

如果你嘗試改變元素的 *key*，會引發錯誤：

```
pos->first = "hello"; // ERROR at compile time
```

不過如果 *value* 本身的型別並非 const，改變 *value* 沒有問題：

```
pos->second = 13.5; // OK
```

如果你一定得改變元素的 *key*，只有一條路：以一個「*value* 相同」的新元素替換掉舊元素。下面是個泛化函式：

```
// cont/newkey.hpp

namespace MyLib {
    template <class Cont>
    inline
    bool replace_key (Cont& c,
                      const typename Cont::key_type& old_key,
                      const typename Cont::key_type& new_key)
    {
        typename Cont::iterator pos;
        pos = c.find(old_key);

        if (pos != c.end()) {
            // insert new element with value of old element
            c.insert(typename Cont::value_type(new_key,
                                               pos->second));
            // remove old element
            c.erase(pos);
            return true;
        }
        else {
            // key not found
            return false;
        }
    }
}
```

關於 insert()和 erase()成員函式，請見下一節討論。

[27] pos->first 是(*pos).first 的簡寫形式。有些程式庫只支援後一種形式。

這個泛型函式的用法很簡單，把舊的 *key* 和新的 *key* 傳遞進去就行。例如：

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

如果你面對的是 multimaps，情況也一樣。

注意，maps 提供了一種非常方便的手法，讓你改變元素的 *key*。只需如此這般：

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

關於 maps 的 ***subscript***（下標）運算子使用細節，詳見 6.6.3 節, p205。

元素的安插（Inserting）和移除（Removing）

表 6.31 列出 maps 和 multimaps 所支援的元素安插和刪除函式。

操作	效果
`c.insert(elem)`	安插一份 `elem` 副本，傳回新元素位置（不論是否成功 — 對 maps 而言）。
`c.insert(pos,elem)`	安插一份 `elem` 副本，傳回新元素位置（`pos` 是個提示，指出安插動作的搜尋起點。如果提示恰當，可大大加快速度）
`c.insert(beg,end)`	將區間 `[beg;end)` 內所有元素的副本安插到 `c`（無回返值）
`c.erase(elem)`	移除「實值（*value*）與 `elem` 相等」的所有元素，傳回被移除的元素個數。
`c.erase(pos)`	移除迭代器 `pos` 所指位置上的元素，無回返值。
`c.erase(beg,end)`	移除區間 `[beg;end)` 內的所有元素，無回返值。
`c.clear()`	移除全部元素，將整個容器清空。

表 6.31　Maps 和 Multimaps 的元素安插和移除

p182 之中關於 set 和 multisets 的說明，此處依然適用。上述操作函式的回返值型別有些差異，其情況與 set 和 multisets 的情況完全相同。當然，這裡的元素是 *key*/*value* pair。所以這裡的用法更複雜些。

安插一個 *key*/*value* pair 的時候，你一定要記住，在 map 和 multimaps 內部，*key* 被視爲常數。你要不得提供正確型別，要不就得提供隱式或顯式型別轉換。有三個不同的方法可以將 *value* 傳入 map：

1. 運用 value_type

   爲了避免隱式型別轉換，你可以利用 value_type 明白傳遞正確型別。value_type 是容器本身提供的型別定義。例如：

```
std::map<std::string,float> coll;
...
coll.insert(std::map<std::string,float>::value_type("otto",
                                                    22.3));
```

2. 運用 pair<>

   另一個作法是直接運用 pair<>。例如：

```
std::map<std::string,float> coll;
...
// use implicit conversion:
coll.insert(std::pair<std::string,float>("otto",22.3));
// use no implicit conversion:
coll.insert(std::pair<const std::string,float>("otto",22.3));
```

   上述第一個 insert() 述句內的型別並不正確，所以會被轉換成真正的元素型別。爲了做到這一點，insert() 成員函式被定義爲 member template[28]。

3. 運用 make_pair()

   最方便的辦法是運用 make_pair() 函式（詳見 p36）。這個函式根據傳入的兩個引數建構出一個 pair 物件：

```
std::map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

   和作法 2 一樣，也是利用 member template insert() 來執行必要的型別轉換。

下面是個簡單例子，對著 map 安插一個元素，然後檢查是否成功：

```
std::map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

[28] 如果你的系統不支援 member template，你必須傳遞型別正確的元素，你通常必須因此進行顯式型別轉換（explicit conversions）。

關於 insert() 回返值的討論，請見 p182，那兒有更多例子，也適用於 maps。注意此處仍然透過 map 的 ***subscript***（下標）運算子提供較爲方便的元素安插和設定動作。這一點將在 6.6.3 節, p205 討論。

如果要移除「擁有某個 *value*」的元素，呼叫 erase() 即可辦到：

```
std::map<std::string,float> coll;
...
// remove all elements with the passed key
coll.erase(key);
```

erase() 傳回移除元素的個數。對 maps 而言其回返值非 0 即 1。

如果 multimap 內含重複元素，你不能使用 erase() 來刪除這些重複元素中的第一個。你可以這麼做：

```
typedef std::multimap<std::string,float> StringFloatMMap;
StringFloatMMap coll;
...
// remove first element with passed key
StringFloatMMap::iterator pos;
pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

這裏應該採用成員函式 find()，而非 STL 演算法 find()，因爲前者速度更快（參見 p154 的例子）。然而你不能使用成員函式 find() 來移除「擁有某個 *value*（而非某個 *key*）」的元素。詳細討論請見 p198。

移除元素時，當心發生意外狀況。當你移除迭代器所指物件時，有一個很大的危險，看看這個例子：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos); // RUNTIME ERROR !!!
    }
}
```

對 pos 所指元素實施 erase()，會使 pos 不再成爲一個有效的 coll 迭代器。如果此後你未對 pos 重新設值就逕行使用 pos，前途未卜！事實上只要一個 ++pos 動作就會導致未定義的行爲。

如果 erase() 總是傳回下一元素的位置，那就好辦了：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        pos = coll.erase(pos); // would be fine, but COMPILE TIME ERROR
    }
    else {
        ++pos;
    }
}
```

可惜 STL 設計過程中否決了這種想法，因爲萬一用戶並不需要這一特性，就會耗費不必要的執行時間。我個人不太贊成這項決定，因爲這麼一來代碼會變得更複雜，更容易出錯，就時間而言，恐怕得不償失！

下面是移除「迭代器所指元素」的正確作法：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// remove all elements having a certain value
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        coll.erase(pos++);
    }
    else {
        ++pos;
    }
}
```

注意，pos++ 會將 pos 移向下一元素，但傳回其原始值（指向原位置）的一個副本。因此，當 erase() 被喚起，pos 已經不再指向那個即將被移除的元素了。

6.6.3 將 Maps 視爲關聯式陣列（Associated Arrays）

通常，關聯式容器並不提供元素的直接存取，你必須依靠迭代器。不過 maps 是個例外。Non-const maps 提供下標運算子，支援元素的直接存取，如表 6.32。不過，

下標運算子的索引值並非元素整數位置，而是元素的 *key*。也就是說，索引可以是任意型別，而非侷限爲整數型別。這種介面正是我們所說的關聯式陣列（associative array）。

操作	效果
`m[key]`	傳回一個 reference，指向鍵值爲 `key` 的元素。如果該元素尚未存在，就安插該元素。

表 6.32 Maps 的直接元素存取（透過 `operator[]`）

和一般 array 之間的區別還不僅僅在於索引型別。其他的區別包括：你不可能用上一個錯誤索引。如果你使用某個 *key* 作爲索引，而容器之中尚未存在對應元素，那麼就會自動安插該元素。新元素的 *value* 由 ***default*** 建構式建構。如果元素的 *value* 型別沒有提供 ***default*** 建構式，你就沒這個福分了。再次提醒你，所有基本資料型別都提供有 ***default*** 建構式，以零爲初值（見 p14）。

關聯式陣列的行爲方式可說是毀譽參半：

- 優點是你可以透過更方便的介面對著 map 安插新元素。例如：

```
std::map<std::string,float> coll; // empty collection
/* insert "otto"/7.7 as key/value pair
* - first it inserts "otto"/float()
* - then it assigns 7.7
*/
coll["otto"] = 7.7;
```

其中的述句：

```
coll["otto"] = 7.7;
```

處理如下：

1. 處理 `coll["otto"]`：
   - 如果存在鍵值爲 `"otto"` 的元素，以上式子傳回該元素的 reference。
   - 如果沒有任何元素的鍵值是 `"otto"`，以上式子便爲 map 自動安插一個新元素，鍵值 *key* 爲 `"otto"`，實值 *value* 則以 ***default*** 建構式完成，並傳回一個 reference 指向新元素。
2. 將 7.7 指派給 *value*：
   - 緊接著，將 7.7 指派給上述剛剛誕生的新元素。

這樣，map 之內就包含了一個鍵值（*key*）爲 `"otto"` 的元素，其實值（*value*）爲 7.7。

- 缺點是你可能會不小心誤置新元素。例如下面的述句可能會做出一些意想不到的事情：

  ```
  std::cout << coll["ottto"];
  ```

  它會安插一個鍵值爲 "otto" 的新元素，然後列印其實值，預設情況下是 0。然而，按道理它應該產生一條錯誤訊息，告訴你你把 "otto" 拼寫錯了。

  同時亦請注意，這種元素安插方式比一般的 maps 安插方式來得慢，p202 曾經談過這個主題。原因是新元素必須先使用 ***default*** 建構式將實值（*value*）初始化，而這個初值馬上又被真正的 *value* 給覆蓋了。

6.6.4 異常處理（Exception Handling）

就異常處理而言，Maps 和 multimaps 的行爲與 sets 和 multisets 一樣。參見 p185。

6.6.5 Maps 和 Multimaps 運用實例

將 Map 當作關聯式陣列

下面這個例子將 map 當成一個關聯式陣列來使用。這個 map 用來反映股票行情。元素的鍵值（*key*）是股票名稱，實值（*value*）是股票價格：

```
// cont/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   /* create map / associative array
   * - keys are strings
   * - values are floats
   */
   typedef map<string,float> StringFloatMap;

   StringFloatMap stocks; // create empty container

   // insert some elements
   stocks["BASF"] = 369.50;
```

```
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
    stocks["Siemens"] = 842.20;

    // print all elements
    StringFloatMap::iterator pos;
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
         cout << "stock: " << pos->first << "\t"
              << "price: " << pos->second << endl;
    }
    cout << endl;

    // boom (all prices doubled)
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
         pos->second *= 2;
    }

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
         << "price: " << pos->second << endl;
    }
    cout << endl;

    /* rename key from "VW" to "Volkswagen"
    * - only provided by exchanging element
    */
    stocks["Volkswagen"] = stocks["VW"];
    stocks.erase("VW");

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
         << "price: " << pos->second << endl;
    }
}
```

程式輸出如下:

```
stock: BASF price: 369.5
stock: BMW price: 834
stock: Daimler price: 819
stock: Siemens price: 842.2
stock: VW price: 413.5

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: VW price: 827

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: Volkswagen price: 827
```

將 Multimap 當作字典

下面例子展示如何將 multimap 當成一個字典來使用：

```
// cont/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
#include <iomanip>
using namespace std;

int main()
{
    // define multimap type as string/string dictionary
    typedef multimap<string,string> StrStrMMap;

    // create empty dictionary
    StrStrMMap dict;

    // insert some elements in random order
    dict.insert(make_pair("day","Tag"));
```

```
    dict.insert(make_pair("strange","fremd"));
    dict.insert(make_pair("car","Auto"));
    dict.insert(make_pair("smart","elegant"));
    dict.insert(make_pair("trait","Merkmal"));
    dict.insert(make_pair("strange","seltsam"));
    dict.insert(make_pair("smart","raffiniert"));
    dict.insert(make_pair("smart","klug"));
    dict.insert(make_pair("clever","raffiniert"));

    // print all elements
    StrStrMMap::iterator pos;
    cout.setf (ios::left, ios::adjustfield);
    cout << ' ' << setw(10) << "english "
         << "german " << endl;
    cout << setfill('-') << setw(20) << ""
         << setfill(' ') << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         cout << ' ' << setw(10) << pos->first.c_str()
              << pos->second << endl;
    }
    cout << endl;

    // print all values for key "smart"
    string word("smart");
    cout << word << ": " << endl;
    for (pos = dict.lower_bound(word);
         pos != dict.upper_bound(word); ++pos) {
         cout << "     " << pos->second << endl;
    }

    // print all keys for value "raffiniert"
    word = ("raffiniert");
    cout << word << ": " << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         if (pos->second == word) {
              cout << " " << pos->first << endl;
         }
    }
}
```

程式輸出如下:

```
  english    german
-------------------------
  car        Auto
  clever     raffiniert
  day        Tag
  smart      elegant
  smart      raffiniert
  smart      klug
  strange    fremd
  strange    seltsam
  trait      Merkmal

smart:
   elegant
   raffiniert
   klug
raffiniert:
   clever
   smart
```

搜尋具有某特定實值（*values*）的元素

下面例子展示如何使用全域的 `find_if()` 演算法來搜尋具有某特定 *value* 的元素:

```
// cont/mapfind.cpp

#include <iostream>
#include <algorithm>
#include <map>
using namespace std;

/* function object to check the value of a map element
*/
template <class K, class V>
class value_equals {
   private:
   V value;

   public:
   // constructor (initialize value to compare with)
```

```
    value_equals (const V& v)
      : value(v) {
    }

    // comparison
    bool operator() (pair<const K, V> elem) {
    return elem.second == value;
    }
};

int main()
{
    typedef map<float,float> FloatFloatMap;
    FloatFloatMap coll;
    FloatFloatMap::iterator pos;

    // fill container
    coll[1]=7;
    coll[2]=4;
    coll[3]=2;
    coll[4]=3;
    coll[5]=6;
    coll[6]=1;
    coll[7]=3;

    // search an element with key 3.0
    pos = coll.find(3.0); // logarithmic complexity
    if (pos != coll.end()) {
       cout << pos->first << ": "
            << pos->second << endl;
    }

    // search an element with value 3.0
    pos = find_if(coll.begin(),coll.end(), // linear complexity
                  value_equals<float,float>(3.0));
    if (pos != coll.end()) {
         cout << pos->first << ": "
              << pos->second << endl;
    }
}
```

程式輸出如下：

```
3: 2
4: 3
```

6.6.6 綜合實例：運用 Maps, Strings 並於執行期指定排序準則

這裏再示範一個例子。此例針對高級程式員而非 STL 初學者。你可以把它視爲展現 STL 威力與障礙的一個範例。更明確地說，這個例子展現了以下技巧：

- 如何使用 maps
- 如何撰寫和使用仿函式（functor, 或名 function object）
- 如何在執行期定義排序準則
- 如何在「不在乎大小寫」的情況下比較字串（strings）

```
// cont/mapcmp.cpp

#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
using namespace std;

/* function object to compare strings
* - allows you to set the comparison criterion at runtime
* - allows you to compare case insensitive
*/
class RuntimeStringCmp {
  public:
    // constants for the comparison criterion
    enum cmp_mode {normal, nocase};

  private:
    // actual comparison mode
    const cmp_mode mode;

    // auxiliary function to compare case insensitive
    static bool nocase_compare (char c1, char c2)
    {
        return toupper(c1) < toupper(c2);
    }
```

```
  public:
    // constructor: initializes the comparison criterion
    RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
    }

    // the comparison
    bool operator() (const string& s1, const string& s2) const {
        if (mode == normal) {
            return s1 < s2;
        }
        else {
            return lexicographical_compare(s1.begin(), s1.end(),
                                           s2.begin(), s2.end(),
                                           nocase_compare);
        }
    }
};

/* container type:
* - map with
* -string keys
* -string values
* - the special comparison object type
*/
typedef map<string, string, RuntimeStringCmp> StringStringMap;

// function that fills and prints such containers
void fillAndPrint(StringStringMap& coll);

int main()
{
    // create a container with the default comparison criterion
    StringStringMap coll1;
    fillAndPrint(coll1);
```

```
    // create an object for case-insensitive comparisons
    RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

    // create a container with the case-insensitive
    // comparisons criterion
    StringStringMap coll2(ignorecase);
    fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
    // fill insert elements in random order
    coll["Deutschland"] = "Germany";
    coll["deutsch"] = "German";
    coll["Haken"] = "snag";
    coll["arbeiten"] = "work";
    coll["Hund"] = "dog";
    coll["gehen"] = "go";
    coll["Unternehmen"] = "enterprise";
    coll["unternehmen"] = "undertake";
    coll["gehen"] = "walk";
    coll["Bestatter"] = "undertaker";

    // print elements
    StringStringMap::iterator pos;
    cout.setf(ios::left, ios::adjustfield);
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << setw(15) << pos->first.c_str() << " "
             << pos->second << endl;
    }
    cout << endl;
}
```

`main()`建構出兩個容器，並對它們呼叫 `fillAndPrint()`。這個函式以相同的元素值填充上述兩個容器，然後列印其內容。兩個容器的排序準則不同：

1. `coll1` 使用一個型別爲 `RuntimeStringCmp` 的預設仿函式。這個仿函式以元素的 `operator<` 來執行比較動作。
2. `coll2` 使用一個型別爲 `RuntimeStringCmp` 的仿函式，並以 `nocase` 爲初值。`nocase` 會令這個仿函式以「大小寫無關」模式來完成字串的比較和排序。

程式輸出如下：

```
Bestatter    undertaker
Deutschland  Germany
Haken        snag
Hund         dog
Unternehmen  enterprise
arbeiten     work
deutsch      German
gehen        walk
unternehmen  undertake

arbeiten     work
Bestatter    undertaker
deutsch      German
Deutschland  Germany
gehen        walk
Haken        snag
Hund         dog
Unternehmen  undertake
```

第一部分列印第一個容器的內容，該容器以 `operator<` 進行排序。首先輸出所有鍵值爲大寫的字串，然後是鍵值爲小寫的字串。

第二部分以「大小寫無關」模式列印所有字串，次序和第一部分不同。請注意，第二部分少列了一個元素，因爲在大小寫無關的情況下 `"Unternehmen"` 和 `"unternehmen"` 被視爲兩個相同字串[29]，而我們使用的 map 並不接納重複元素。很不幸，列印結果亂七八糟。原本「*value* 應爲 `"enterprise"`」的那個 *key*（是個德文字），其 *value* 卻變成 `"undertake"`。看來這裡應該使用 multimap。沒錯，multimap 的確是用來表現字典的一個典型容器。

[29] 德語中的所有名詞，第一個字母皆大寫。動詞全部小寫。

6.7 其他 STL 容器

STL 是個框架，除了提供標準容器，它也允許你使用其他資料結構作爲容器。你可以使用 string 或 array 作爲 STL 容器，也可以自行撰寫特殊容器以滿足特殊需求。如果你自行撰寫容器，仍可從諸如排序、合併等演算法中受益。這樣的框架正是「開放性封閉（*Open-Closed*）」原則的極佳範例[30]：允許擴展，謝絕修改。

下面是使你的容器「STL 化」的三種不同方法：

1. **The invasive approach**[31]（侵入性作法）
   直接提供 STL 容器所需介面。特別是諸如 `begin()`和 `end()`之類的常用函式。這種作法需以某種特定方式編寫容器，所以是侵入性的。
2. **The noninvasive approach** [31]（非侵入性作法）
   由你撰寫或提供特殊迭代器，作爲演算法和特殊容器間的介面。此一作法是非侵入性的，它所需要的只是「巡訪容器所有元素」的能力 — 這是任何容器都能以某種形式展現的能力。
3. **The wrapper approach**（包裝法）
   將上述兩種方法加以組合，我們可以寫一個外套類別（wrapper class）來包裝任何資料結構，並顯暴出與 STL 容器相似的介面。

本節首先將 string 視爲標準容器來討論，當作侵入性作法的一個例子，然後再以非侵入性作法討論重要的標準容器：array。當然你也可以使用包裝法來存取 array 的資料。本節最後概略討論了一個目前尚未被涵蓋於標準規格中的容器：**hash table**。

任何 STL 容器都應該能夠以不同的配置器（allocator）加以參數化。C++ 標準程式庫提供了一些特殊函式和類別，幫助你撰寫配置器並對付尚未初始化的記憶體。詳見 15.2 節, p728。

6.7.1 Strings 可被視爲一種 STL 容器

C++ 標準程式庫的 `string` 類別，乃是「以侵入性作法編寫 STL 容器」的一個好例子（關於 `string` 類別的詳盡討論，請見第 11 章）。Strings 可被視爲以字元（characters）爲元素的一種容器；字元構成序列，你可以在序列上來回移動巡訪。因此，標準的 `string` 類別提供了 STL 容器介面。Strings 也提供成員函式 `begin()` 和 `end()`，傳回隨機存取迭代器，可用來巡訪整個 string。同時，爲了支援迭代器和迭代器配接器(iterator adapters)，strings 也提供了一些操作函式，例如 `push_back()` 用以支援 *back inserter*s。

---

[30] 我從 Robert C. Martin 那兒頭一次聽說這個名稱，他是從 Bertrand Meyer 那兒聽來的。

[31] 有時也說成 **intrusive** 和 **nonintrusive**

從 STL 角度來思考，string 的處理有點不尋常，因爲我們通常將 string 當作一個物件來處理（我們可以傳遞、複製或設定 string）。但如果要對單個字元進行處理，採用 STL 演算法將大有裨益。例如可以採用 istream 迭代器讀取字元，或轉換 string 內的字元（譬如轉成大寫或小寫）。此外，透過 STL 演算法，可以對 string 採取特殊的比較規則 — 標準 string 介面並不提供這種能力。

p497, 11.2.13 節是 string 完整章節的一部分，在那裏我詳細討論了 string 的 STL 相關特性，並給出一些實例。

6.7.2 Arrays 可被視爲一種 STL 容器

我們也可以把 array 當成 STL 容器來使用，但 array 並不是類別，所以不提供 begin() 和 end() 等成員函式，也不允許存在任何成員函式。在這裡，我們只能採用非侵入性作法或包裝法。

直接運用 array

採取非侵入性作法很簡單，你只需要一個物件，它能夠透過 STL 迭代器介面，巡訪 array 的所有元素。事實上這樣的物件早就恭候多時了，就是一般指標。STL 設計之初就決定讓迭代器擁有和一般指標相同的介面，於是你可以將一般指標當成迭代器來使。這又一次展示了純粹抽象的泛化概念：「行爲類似迭代器」的任何東西就是一種迭代器。事實上指標正是一個隨機存取迭代器（參見 p255, 7.2.5 節）。以下例子示範如何以 array 作爲 STL 容器：

```
// cont/array1.cpp

#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    int coll[] = { 5, 6, 2, 4, 1, 3 };

    // square all elements
    transform (coll, coll+6,          // first source
               coll,                  // second source
               coll,                  // destination
               multiplies<int>());    // operation
```

```
    // sort beginning with the second element
    sort (coll+1, coll+6);

    // print all elements
    copy (coll, coll+6,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

千萬注意，一定要正確傳遞 array 尾部位置，這裡是 `coll+6`。記住，一定要確保區間尾端是最後元素的下一個位置。

程式輸出如下：

```
25 1 4 9 16 36
```

p382 和 p421 還有一些例子。

一個 array 的包裝

Bjarne Stroustrup 的《*The C++ Programming Language*》第三版中，介紹了一個很有用的 array 包裝類別，性能不輸一般的 array，而且更安全。這是「使用者自行定義 STL 容器」的一個好例子。該容器所使用的，就是包裝法：在 array 之外包裝一層常用的容器介面。

Class `carray`（這是 "C array" 或 "constant size array" 的縮寫）定義如下[32]：

```
// cont/carray.hpp

#include <cstddef>

template<class T, std::size_t thesize>
class carray {
    private:
    T v[thesize]; // fixed-size array of elements of type T

    public:
    // type definitions
    typedef T     value_type;
    typedef T*    iterator;
```

[32] 原始例子名爲 `c_array`，定義於 Bjarne Stroustrup 的《*The C++ Programming Language*》第三版 17.5.4 節。這裡我做了一些改動。

```
    typedef const T*       const_iterator;
    typedef T&             reference;
    typedef const T&       const_reference;
    typedef std::size_t    size_type;
    typedef std::ptrdiff_t difference_type;

    // iterator support
    iterator begin() { return v; }
    const_iterator begin() const { return v; }
    iterator end() { return v + thesize; }
    const_iterator end() const { return v+thesize; }

    // direct element access
    reference operator[](std::size_t i) { return v[i]; }
    const_reference operator[](std::size_t i) const { return v[i]; }

    // size is constant
    size_type size() const { return thesize; }
    size_type max_size() const { return thesize; }

    // conversion to ordinary array
    T* as_array() { return v; }
};
```

下面是 carray 的一個運用實例：

```
// cont/carray1.cpp

#include <algorithm>
#include <functional>
#include "carray.hpp"
#include "print.hpp"
using namespace std;

int main()
{
    carray<int,10> a;

    for (unsigned i=0; i<a.size(); ++i) {
        a[i] = i+1;
    }
```

```
    PRINT_ELEMENTS(a);

    reverse(a.begin(),a.end());
    PRINT_ELEMENTS(a);

    transform(a.begin(),a.end(),     // source
              a.begin(),             // destination
              negate<int>());        // operation
    PRINT_ELEMENTS(a);
}
```

如你所見，你可以使用一般容器介面（`begin()`, `end()`, `operator[]`）來直接操作這個容器。這麼一來你也就可以使用那些需要呼叫 `begin()`和 `end()`的各項操作了，例如某些 STL 演算法，以及 p118 所介紹的輔助函式 `PRINT_ELEMENTS()`。

程式輸出如下：

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

（譯註：關於這個 class，更細緻的實作手法請參考 http://www.boost.org/ 的 Boost 程式庫）

6.7.3 Hash Tables

有一個資料結構可用於群集（collection）身上，非常重要，卻未包含於 C++ 標準程式庫內，那就是 hash table。最初的 STL 並未涵蓋 hash table，然而確實曾有提案要求，將 hash table 併入標準規格。但是標準委員會覺得這份提議來得太晚，沒有採納。（我們必須在某個時間點中止引入新功能，開始關注細節，否則工作永無止境）

不過，C++ 社群早已經有了數種可用的 hash table 實作版本。一般而言程式庫會提供四種 hash table：hash_set, hash_multiset, hash_map, hash_multimap。和其他關聯式容器一樣，"multi" 版允許元素重複，"map" 版的元素是個 *key/value* pair。Bjarne Stroustrup 在其《*The C++ Programming Language*》第三版 17.6 節中曾經實作一個 `hash_map` 容器作爲示範，並進行詳細討論。關於 hash table 的具體實現，可參見 STLport（http://www.stlport.org）。當然，由於 hash table 尚未正規化，所以不同的實作版本可能在細節上有所不同。（譯註：如果你對 hash table 的運用和設計原理感興趣，請參考《STL 源碼剖析》by 侯捷, 碁峰 2002，5.7 節~5.11 節。該處討論的是 SGI STL 實作版本，涵括底層 hash table 和外顯介面 hash_set, hash_multiset, hash_map, hash_multimap）

6.8 動手實現 Reference 語意

通常，STL 容器提供的是「*value* 語意」而非「*reference* 語意」，後者在內部建構了元素副本，任何操作傳回的也是這些副本。p135, 5.10.2 節討論了這種作法的優劣，並分析了產生後果。總之，要在 STL 容器中用到「*reference* 語意」（不論是因爲元素的複製代價太大，或因爲需要在不同群集中共用同一個元素），就要採用智慧型指標，避免可能的錯誤。這裏有一個解決辦法：對指標所指之物件採用 **reference counting**（參用計數）智慧型指標[33]。

```
// cont/countptr.hpp

#ifndef COUNTED_PTR_HPP
#define COUNTED_PTR_HPP

/* class for counted reference semantics
* - deletes the object to which it refers when the last CountedPtr
* that refers to it is destroyed
*/
template <class T>
class CountedPtr {
  private:
    T* ptr;       // pointer to the value
    long* count;  // shared number of owners

  public:
    // initialize pointer with existing pointer
    // - requires that the pointer p is a return value of new
    explicit CountedPtr (T* p=0)
      : ptr(p), count(new long(1)) {
    }

    // copy pointer (one more owner)
    CountedPtr (const CountedPtr<T>& p) throw()
      : ptr(p.ptr), count(p.count) {
        ++*count;
    }
```

[33] 感謝 Greg Colvin 和 Beman Dawes 對這個 class 的實作內容所給予的回應。

```
    // destructor (delete value if this was the last owner)
    ~CountedPtr () throw() {
       dispose();
    }

    // assignment (unshare old and share new value)
    CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
       if (this != &p) {
          dispose();
          ptr = p.ptr;
          count = p.count;
          ++*count;
       }
       return *this;
    }

    // access the value to which the pointer refers
    T& operator*() const throw() {
       return *ptr;
    }
    T* operator->() const throw() {
       return ptr;
    }

  private:
    void dispose() {
       if (--*count == 0) {
          delete count;
          delete ptr;
       }
    }

};

#endif /*COUNTED_PTR_HPP*/
```

這個 class 有點類似標準規格書所提供的 `auto_ptr` class（參見 p38, 4.2 節）。用來初始化智慧型指標的值，應當是 `operator new` 的回返值。但是和 `auto_ptr` 不同的是，這種智慧型指標一旦被複製，原指標和新的副本指標都是有效的。只有當指向同一物件的最後一個智慧型指標被摧毀，其所指物件才會被刪除。

你可以改善這個 class，例如你可以爲它實現自動型別轉換，或是提供「將擁有權由智慧型指標交至呼叫端手上」的能力。

以下程式說明如何使用這個 class：

```
// cont/refsem1.cpp

#include <iostream>
#include <list>
#include <deque>
#include <algorithm>
#include "countptr.hpp"
using namespace std;

void printCountedPtr (CountedPtr<int> elem)
{
    cout << *elem << ' ';
}

int main()
{
    // array of integers (to share in different containers)
    static int values[] = { 3, 5, 9, 1, 6, 4 };

    // two different collections
    typedef CountedPtr<int> IntPtr;
    deque<IntPtr> coll1;
    list<IntPtr> coll2;

    /* insert shared objects into the collections
    * - same order in coll1  coll1
    * - reverse order in coll2 coll2
    */
    for (int i=0; i<sizeof(values)/sizeof(values[0]); ++i) {
         IntPtr ptr(new int(values[i]));
         coll1.push_back(ptr);
         coll2.push_front(ptr);
    }
```

```
    // print contents of both collections
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl << endl;

    /* modify values at different places
    * - square third value in coll1
    * - negate first value in coll1
    * - set first value in coll2 to 0
    */
    *coll1[2] *= *coll1[2];
    (**coll1.begin()) *= -1;
    (**coll2.begin()) = 0;

    // print contents of both collections again
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl;
}
```

程式輸出如下：

```
3 5 9 1 6 4
4 6 1 9 5 3

-3 5 81 1 6 0
0 6 1 81 5 -3
```

注意，如果你呼叫一個輔助函式，而它在某處保存了群集（collection）內的某個元素（一個 `IntPtr`），那麼即使群集被銷毀，或其元素全被刪除，那個智慧型指標所指的元素依然有效。

關於其他智慧型指標類別，請參考 http://www.boost.org/ 的 Boost 程式庫，該程式庫是 C++ 標準程式庫的擴充（在那兒 `CountedPtr<>` 名爲 `shared_ptr<>`）。

6.9 各種容器的運用時機

C++ 標準程式庫提供了各具特長的不同容器。現在的問題是：該如何選擇最佳的容器類別？表 6.33 作了一番概述，但其中有些描述可能不一定實際。例如，如果你需要處理的元素數量很少，可以忽略複雜度，因爲線性演算法通常對元素本身的處理過程比較快，這種情況下，「線性複雜度搭配快速的元素處理」要比「對數複雜度搭配緩慢的元素處理」來得划算。

以下規則做爲表 6.33 的補充，可能對你有所幫助：

- 預設情況下應該使用 vector。vector 的內部結構最簡單，並允許隨機存取，所以資料的存取十分方便靈活，資料的處理也夠快。
- 如果經常要在序列頭部和尾部安插和移除元素，應該採用 deque。如果你希望元素被移除時，容器能夠自動縮減記憶體，那麼你也應該採用 deque。此外，由於 vectors 通常採用一個記憶體區塊來存放元素，而 deque 採用多個區塊，所以後者可內含更多元素。
- 如果需要經常在容器的中段執行元素的安插、移除和移動，可考慮使用 list。List 提供特殊的成員函式，可以在常數時間內將元素從 A 容器轉移到 B 容器。但由於 list 不支援隨機存取，所以如果只知道 list 的頭部卻要造訪 list 的中段元素，性能會大打折扣。

  和所有「以節點爲基礎」的容器相似，只要元素還是容器的一部分，list 就不會令指向那些元素的迭代器失效。vectors 則不然，一旦超過其容量，它的所有 iterators、pointers、references 都會失效；執行安插或移除動作時，也會令一部分 iterators、pointers、references 失效。至於 deque，當它的大小改變，所有 iterators、pointers、references 都會失效。
- 如果你要的容器是這種性質：「每次操作若不成功，便無效用」（並以此態度來處理異常），那麼你應該選用 list（但是不保證其 ***assignment*** 運算子和 `sort()`；而且如果元素比較過程中會擲出異常，那就不要呼叫 `merge()`,`remove()`, `remove_if()`,`unique()`，參見 p172），或是採用關聯式容器（但是不保證多元素安插動作，而且如果比較準則（comparision criterion）的複製/指派動作都可能丟擲異常，那麼也不保證 `swap()`）。STL 的異常處理通論請見 p139，5.11.2 節。6.10.10 節, p249 提供了一個表，列舉出「異常發生時提供特別保障」的所有容器操作函式。
- 如果你經常需要根據某個準則來搜尋元素，那麼應當使用「以該排序準則對元素進行排序」的 set 或 multiset。記住，理論上，面對 1,000 個元素的排序，對數複雜度比線性複雜度好 10 倍。此時正是二元樹拿手好戲的發揮時機。

	Vector	Deque	List	Set	Multiset	Map	Multimap
典型內部結構	dynamic array	array of arrays	doubly linked list	binary tree	biary tree	binary tree	binary tree
元素	value	value	value	value	value	*key*/*value* pair	*key*/*value* pair
元素可重複	是	是	是	否	是	對 *key* 而言否	是
可隨機存取	是	是	否	否	否	對 *key* 而言是	否
迭代器類型	隨機存取	隨機存取	雙向	雙向 元素被視爲常數	雙向 元素被視爲常數	雙向 *key* 被視爲常數	雙向 *key* 被視爲常數
元素搜尋速度	慢	慢	非常慢	快	快	對 *key* 而言快	對 *key* 而言快
快速安插移除	尾端	頭尾兩端	任何位置	—	—	—	—
安插移除導致除效 iterators, pointers, references	重新配置時	總是如此	絕不會	絕不會	絕不會	絕不會	絕不會
釋放被移除元素之記憶體	絕不會	有時會	總是如此	總是如此	總是如此	總是如此	總是如此
允許保留記憶體	是	否	—	—	—	—	—
交易安全若失敗不帶來任何影響	尾端 push/pop 時	頭尾兩端 push/pop 時	任何時候除了排序和賦值	任何時候除了多元素安插	任何時候除了多元素安插	任何時候除了多元素安插	任何時候除了多元素安插

表 6.33　STL 容器能力一覽表

就搜尋速度而言，hash table 通常比二元樹還要快 5~10 倍。所以如果有 hash table 可用，就算它尚未標準化，也應該考慮使用。但是 hash table 的元素並未排序，所以如果元素必須排序，它就用不上了。由於 hash table 不是 C++ 標準程式庫的一員，如果你要保證可攜性，就必須擁有其源碼。

- 如想處理 *key*/*value* pair，請採用 map 或 multimap（可以的話請採用 hash table）。
- 如果需要關聯式陣列，應採用 map。
- 如果需要字典結構，應採用 multimap。

有一個問題比較棘手：如何根據兩種不同的排序準則對元素進行排序？例如存放元素時，你希望採用客戶提供的排序準則，搜尋元素時，希望使用另一個排序準則。這和資料庫的情況相同，你需要在數種不同的排序準則下進行快速存取。這時候你可能需要兩個 sets 或 maps，各自擁有不同的排序準則，但共用相同的元素。注意，數個群集共用相同的元素，乃是一項特殊技術，6.8 節, p222 對此有所闡述。

關聯式容器擁有自動排序能力，並不意味它們在排序方面的執行效率更高。事實上由於關聯式容器每安插一個新元素，都要進行一次排序，所以速度反而不及序列式容器經常採用的手法：先安插所有元素，然後呼叫 9.2.2 節, p328 介紹的排序演算法進行一次完全排序。

下面兩個簡單的程式分別使用不同的容器，從標準輸入讀取字串，進行排序，然後列印所有元素（去掉重複字串）：

1. 運用 **set**：

```
// cont/sortset.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
   /* create a string set
   * - initialized by all words from standard input
   */
   set<string> coll((istream_iterator<string>(cin)),
                    (istream_iterator<string>()));

   // print all elements
   copy (coll.begin(), coll.end(),
         ostream_iterator<string>(cout, "\n"));
}
```

2. 運用 vector：

```
// cont/sortvec.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    /* create a string vector
    * - initialized by all words from standard input
    */
    vector<string> coll((istream_iterator<string>(cin)),
                        (istream_iterator<string>()));

    // sort elements
    sort (coll.begin(), coll.end());

    // print all elements ignoring subsequent duplicates
    unique_copy (coll.begin(), coll.end(),
                 ostream_iterator<string>(cout, "\n"));
}
```

我在我的系統上使用大約 150,000 個字串來測試這兩個程式，我發現 vectors 版本快 10% 左右。如果使用 `reserve()`，vectors 版本還可以再快將近 5%。如果允許重複元素（改用 multiset 取代 set，呼叫 `copy()` 取代 `unique_copy()`），則情況發生劇烈變化：vectors 版本領先超過 40%！這些比較雖然不具代表性，但至少證實了一點：對各種不同的元素處理方法多加嘗試是值得的。

現實中預測哪種容器最好，往往相當困難。STL 的一大優點就是你可以輕而易舉地嘗試各種版本。主要工作 — 各種資料結構和演算法 — 已經就位，你只需依照對自己最有利的方式將它們組合運用就行了。

6.10 細說容器的型別和成員

本節討論各種 STL 容器，闡述 STL 容器所支援的一切操作函式。型別和成員一律按功能分組。針對每一種型別定義和操作，本節描述其標記式（signature）、行爲、支援者（容器）。本節涉及的容器包括 vectors、deques、lists、sets、multisets、maps、multimaps 和 strings。後續數節中 *container* 指的是「支援該成員」的某容器型別。

6.10.1 容器的型別

container::**value_type**

- 元素型別。
- 用於 sets 和 multisets 時是常數。
- 用於 maps 和 multimaps 時是 `pair <const` *key-type*, *value-type*`>`。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 之中都有定義。

container::**reference**

- 元素的引用型別（reference type）。
- 典型定義：*container*`::value_type&`。
- 在 `vector<bool>` 中其實是個輔助類別（參見 p158）。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**const_reference**

- 常數元素的引用型別（reference type）。
- 典型定義：`const` *container*`::value_type&`。
- 在 `vector<bool>` 中是 `bool`。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**iterator**

- 迭代器型別。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**const_iterator**

- 常數迭代器的型別。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**reverse_iterator**

- 反向迭代器型別。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps 中都有定義。

container::**const_reverse_iterator**

- 常數反向迭代器的型別。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**size_type**

- 無正負號整數型別，用以定義容器大小。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**difference_type**

- 有正負號整數型別，用以定義距離。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

container::**key_type**

- 用以定義關聯式容器的元素內的 *key* 型別。
- 用於 sets 和 multisets 時，相當於 `value_type`。
- 在 sets, multisets, maps, multimaps 中都有定義。

container::**mapped_type**

- 用以定義關聯式容器的元素內的 *value* 型別。
- 在 maps 和 multimaps 中都有定義。

container::**key_compare**

- 關聯式容器內的「比較準則」的型別。
- 在 sets, multisets, maps, multimaps 中都有定義。

container::**value_compare**

- 用於整個元素之「比較準則」的型別。
- 用於 sets 和 multisets 時，相當於 `key_compare`。
- 在 maps 和 multimaps 中，它是「比較準則」的輔助類別，僅比較兩元素的 *key*。
- 由 sets, multisets, map, multimap 中都有定義。

container::**allocator_type**

- 配置器型別。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定義。

6.10.2 生成（Create）、複製（Copy）、銷毀（Destroy）

STL 容器支援下列建構式和解構式，並且大多數建構式允許將配置器作爲一個附加引數傳遞（參見第 6.10.9 節, p246）。

container::*container* ()

- *default* 建構式
- 產生一個新的空容器
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支援

`explicit container::container (const CompFunc& op)`

- 以 op 爲排序準則，產生一個空容器（參見 p191 和 p213 實例）。
- 排序準則必須定義一個 **strict weak ordering**（參見 p176）。
- sets, multisets, maps, multimaps 支援。

`explicit container::container (const container& c)`

- *copy* 建構式。
- 產生既有容器的一個副本。
- 針對 c 中的每一個元素呼叫 *copy* 建構式。
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支援。

`explicit container::container (size_type num)`

- 產生一個容器，可含 *num* 個元素。
- 元素由其 *default* 建構式創建。
- vectors, deques, lists 都支援。

`container::container (size_type num, const T& value)`

- 產生一個容器，可含 *num* 個元素。
- 所有元素都是 *value* 的副本。
- T 是元素型別。
- 對於 strings，*value* 並非 *pass by reference*。
- vectors、deques、lists 和 strings 都支援。

`container::container (InputIterator beg, InputIterator end)`

- 產生容器，並以區間 [beg;end) 內的所有元素爲初值。
- 此函式爲一個 member template（參見 p11）。因此只要源區間的元素型別可轉換爲容器元素型別，此函式即可派上用場。
- vectors、deques、lists、sets、multisets、maps、multimaps、strings 都支援。

```
container::container (InputIterator beg, InputIterator end,
                      const CompFunc& op)
```

- 產生一個排序準則爲 *op* 的容器，並以區間 [beg;end) 內的所有元素進行初始化。
- 此函式爲一個 member template（參見 p11）。因此只要源區間的元素型別可轉換爲容器元素型別，此函式即可派上用場。
- 排序準則必須定義一個 **strict weak ordering**（參見 p176）。
- sets、multisets、maps 和 multimaps 都支援。

container::~*container* ()

- 解構式。
- 移除所有元素，並釋放記憶體。
- 對每個元素呼叫其解構式。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.3 非變動性操作（Nonmodifying Operations）

大小相關操作（Size Operations）

size_type *container*::**size** () const

- 傳回現有元素的數目。
- 欲檢查容器是否爲空，應使用 empty()，因爲 empty() 可能更快。
- vectors、deques、lists、sets、multisets、maps、multimasp 和 strings 都支援。

bool *container*::**empty** () const

- 檢驗容器是否爲空，並傳回檢查結果。
- 相當於 *container*::size()==0，但是可能更快（尤其對 lists 而言）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

size_type *container*::**max_size** () const

- 傳回容器可包含的最大元素個數。
- 這是一個技術層次的數值，可能取決於容器的記憶體模型。尤其 vectors 通常使用一個記憶體區段（segment），所以 vector 的這個值往往小於其他容器。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

容量操作（Capacity Operations）

size_type *container*::**capacity** () const

- 傳回重配置記憶體之前所能容納的最多元素個數。
- vectors 和 strings 都支援。

void *container*::**reserve** (size_type *num*)

- 在內部保留若干記憶體，至少能夠容納 num 個元素。
- 如果 *num* 小於實際容量，對 vectors 無效，對 strings 則是一個非固定的縮減請求（nonbinding shrink request）。

- vectors 的容量如何縮小，請見 p149 例子。
- 每次重新配置都會耗用相當時間，並造成所有 references、pointers、iterators 失效。因此 `reserve()`可以提高速度，保持 references、pointers、iterators 的有效性。詳見 p149。
- vectors 和 strings 都支援。

元素間的比較（Comparison Operations）

```
bool comparison (const container& c1, const container& c2)
```

- 傳回兩個同型容器的比較結果。
- *comparison* 可以是下面之一：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 如果兩個容器擁有相同數量的元素，且元素順序相同，而且所有相應元素兩兩相比之結果爲 true，我們便說這兩個容器相等。
- 要檢驗 A 容器是否小於 B 容器，需使用「字典順序」來比較。關於「字典順序」，請見 p360 `lexicographical_compare()`的描述。
- vectors、deques、lists、sets、multisets、maps、multimaps, strings 都支援。

關聯式容器特有的非變動性操作

這裏所介紹的成員函式都是對應於 p338, 9.5 節和 p397, 9.9 節所討論的 STL 演算法的特殊實作版本。這些函式利用了關聯式容器的元素已序性，提供更好的性能。例如在 1,000 個元素中進行搜尋，所需的比較平均不超過 10 次（參見 p21, 2.3 節）。

```
size_type container::count (const T& value) const
```

- 傳回與 *value* 相等的元素個數。
- 這是 p338 所討論的 `count()`演算法的特殊版本。
- `T` 是被排序值的型別
  - — 在 sets 和 multisets 中，`T` 是元素型別。
  - — 在 maps 和 multimaps 中，`T` 是 *key* 的型別。
- 複雜度：線性。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::find (const T& value)
const_iterator container::find (const T& value) const
```

- 傳回「實值等於 *value*」的第一個元素位置。
- 如果找不到元素就傳回 `end()`。
- 這是 p341 所討論的 `find()` 演算法的特殊版本。
- `T` 是被排序值的型別：
  - — 在 sets 和 multisets 中，`T` 是元素型別。
  - — 在 maps 和 multimaps 中，`T` 是 *key* 的型別。
- 複雜度：對數。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::lower_bound (const T& value)
const_iterator container::lower_bound (const T& value) const
```

- 傳回一個迭代器，指向「根據排序準則，可安插 *value* 副本的第一個位置」。
- 傳回之迭代器指向「實值大於等於 *value* 的第一個元素」（有可能是 `end()`）。
- 如果找不到就傳回 `end()`。
- 這是 p413 所討論的 `lower_bound()` 演算法的特殊版本。
- `T` 是被排序值的型別：
  - — 在 sets 和 multisets 中，`T` 是元素型別。
  - — 在 map 和 multimap 中，`T` 是 *key* 的型別。
- 複雜度：對數。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::upper_bound (const T& value)
const_iterator container::upper_bound (const T& value) const
```

- 傳回一個迭代器，指向「根據排序準則，可安插 *value* 副本的最後一個位置」。
- 傳回之迭代器指向「實值大於 *value* 的第一個元素」（有可能是 `end()`）。
- 如果找不到就傳回 `end()`。
- 這是 p413 所討論的 `upper_bound()` 演算法的特殊版本。
- `T` 是被排序值的型別：
  - — 在 sets 和 multisets 中，`T` 是元素型別。
  - — 在 map 和 multimap 中，`T` 是 *key* 的型別。
- 複雜度：對數。
- sets、multisets、maps 和 multimaps 都支援。

```
pair<iterator,iterator> container::equal_range (const T& value)
pair<const_iterator,const_iterator>
                        container::equal_range (const T& value) const
```

- 傳回一個區間（一對迭代器），指向「根據排序準則，可安插 *value* 副本的第一個位置和最後一個位置」。
- 傳回一個區間，其內的元素實值皆等於 *value*。
- 相當於：
  `make_pair(lower_bound(value),upper_bound(value))`
- 這是 p415 所討論的 `equal_range()` 演算法的特殊版本。
- `T` 是被排序值的型別：
  — 在 sets 和 multisets 中，`T` 是元素型別。
  — 在 map 和 multimap 中，`T` 是 *key* 的型別。
- 複雜度：對數。
- sets、multisets、maps 和 multimaps 都支援。

```
key_compare container::key_comp ()
```

- 傳回一個「比較準則」。
- sets、multisets、maps 和 multimaps 都支援。

```
value_compare container::value_comp ()
```

- 傳回一個作爲比較準則的物件。
- 在 sets 和 multisets 中，它相當於 `key_comp()`。
- 在 maps 和 multimaps 中，它是一個輔助類別，用來比較兩元素的 *key*。
- sets、multisets、maps 和 multimaps 都支援。

6.10.4 賦值（指派，Assignments）

```
container& container::operator = (const container& c)
```

- 將 `c` 的所有元素指派給現有容器，亦即以 `c` 的元素替換所有現有元素。
- 這個運算子會針對被覆蓋的元素呼叫其 ***assignment*** 運算子，針對被附加的元素呼叫其 ***copy*** 建構式，針對被移除的元素呼叫其解構式。
- vectors、deques、lists、sets、multisets、maps 和 multimaps 都支援。

```
void container::assign (size_type num, const T& value)
```

- 指派 *num* 個 *value*，亦即以 *num* 個 *value* 副本替換掉所有現有元素。
- `T` 必須是元素型別。
- sets、multisets、maps 和 multimaps 都支援。

```
void container::assign (InputIterator beg, InputIterator end)
```

- 指派區間 [beg;end) 內的所有元素，亦即以 [beg;end) 內的元素副本替換掉所有現有元素。
- 此函式爲一個 member template（參見 p11）。因此只要源區間的元素型別可轉換爲容器元素型別，此函式即可派上用場。
- vectors、deques、lists 和 strings 都支援。

```
void container::swap (container& c)
```

- 和 c 交換內容。
- 兩個容器*互換*：
  - 元素
  - 排序準則（如果有的話）。
- 此函式擁有常數複雜度。如果不再需要容器中的老舊元素，則應使用本函式來取代賦值動作（參見 p147, 6.1.2 節）。
- 對於關聯式容器，只要「比較準則」進行複製或指派時不丟擲異常，本函式就不丟擲異常。對於其他所有容器，此函式一律不丟擲異常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
void swap (container& c1, container& c2)
```

- 相當於 c1.swap(c2)（參見稍早前的描述）。
- 對於關聯式容器，只要「比較準則」進行複製或賦值時，不丟擲異常，本函式就不丟擲異常。對於其他所有容器，此函式一律不丟擲異常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.5 直接元素存取

```
reference container::at (size_type idx)
const_reference container::at (size_type idx) const
```

- 二者都傳回索引 *idx* 所代表的元素（第一個元素的索引爲 0）。
- 如果傳入一個無效索引（< 0 或 >= size()），會導致 *out_of_range* 異常。
- 後續的修改或記憶體重新配置，可能會導致傳回的 reference 無效。
- 如果呼叫者保證索引有效，那麼最好使用速度更快的 operator[]。
- vectors、deques 和 strings 都支援。

```
reference container::operator[] (size_type idx)
const_reference container::operator[] (size_type idx) const
```

- 二者都傳回索引 *idx* 所代表的元素（第一個元素的索引爲 0）。
- 如果傳入一個無效索引（`< 0` 或 `>= size()`），會導致未定義的行爲。所以呼叫者必須確保索引有效，否則應該使用 `at()`。
- (1) 修改 strings 或 (2) 記憶體重新配置，可能會導致 non-`const` strings 傳回的 reference 失效（詳見 p487）。
- vectors、deques 和 strings 都支援。

```
T& map::operator[] (const key_type& key)
```

- 關聯式陣列的 `operator[]`。
- 在 map 中，會傳回 *key* 所對應的 *value*。
- 注意：如果不存在「鍵值爲 *key*」的元素，則本操作會自動生成一個新元素，其初值由 *value* 型別的 ***default*** 建構式給定。所以不存在所謂的無效索引。例如：

```
map<int,string> coll;
coll[77] = "hello";  // insert key 77 with value "hello"
cout << coll[42];    // Oops, inserts key 42 with value "" and
                     // prints the value
```

  詳見 p205, 6.6.3 節。

- `T` 是元素的 *value* 型別。
- 相當於：

```
(*((insert(make_pair(x,T()))).first)).second
```

- 只有 map 支援此一操作。

```
reference container::front ()
const_reference container::front () const
```

- 都傳回第一個元素（第一個元素的索引爲 0）。
- 呼叫者必須確保容器內有元素（`size()>0`），否則會導致未定義的行爲。
- vectors、deques 和 lists 都支援。

```
reference container::back ()
const_reference container::back () const
```

- 都傳回最後一個元素（索引爲 `size()-1`）。
- 呼叫者必須確保容器內擁有元素（`size()>0`）；否則會導致未定義的行爲。
- vectors、deques 和 lists 都支援。

6.10.6「會產出迭代器」的各項操作

本節各個成員函式都會傳回迭代器，憑藉這些迭代器你可以巡訪容器中的所有元素。表 6.34 列出各種容器所提供的迭代器類型（參見 p251, 7.2 節）。

容器	迭代器類型（iterator category）
Vector	隨機存取
Deque	隨機存取
List	雙向
Set	雙向，元素爲常量
Multiset	雙向，元素爲常量
Map	雙向，*key* 爲常量
Multimap	雙向，*key* 爲常量
String	隨機存取

表 6.34　各種容器提供的迭代器類型

```
iterator container::begin ()
const_iterator container::begin () const
```

- 傳回一個迭代器，指向容器起始處（第一元素的位置）。
- 如果容器爲空，則此動作相當於 `container::end()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
iterator container::end ()
const_iterator container::end () const
```

- 傳回一個迭代器，指向容器尾端（最後元素的下一位置）。
- 如果容器爲空，則此動作相當於 `container::begin()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
reverse_iterator container::rbegin ()
const_reverse_iterator container::rbegin () const
```

- 傳回一個逆向迭代器，指向逆向迭代時巡訪的第一個元素。
- 如果容器爲空，則此動作相當於 `container::rend()`。
- 關於逆向迭代器，詳見 p264, 7.4.1 節。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
reverse_iterator container::rend ()
const_reverse_iterator container::rend () const
```

- 傳回一個逆向迭代器，指向逆向迭代時巡訪的最後一個元素的下一位置。
- 如果容器爲空，則此動作相當於 `container::rbegin()`。
- 關於逆向迭代器，詳見 p264, 7.4.1 節。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.7 元素的安插（Inserting）和移除（Removing）

```
iterator container::insert (const T& value)
pair<iterator,bool> container::insert (const T& value)
```

- 安插一個 *value* 副本於關聯式容器。
- 元素可重複者（multisets 和 multimap）採用第一形式。傳回新元素的位置。
- 元素不可重複者（sets 和 map）採用第二形式。如果有「具備相同 *key*」的元素已經存在，導致無法安插，會傳回現有元素的位置和一個 `false`。如果安插成功，傳回新元素的位置和一個 `true`。
- `T` 是容器元素的型別，對 map 和 multimap 而言那是一個 *key*/*value* pair。
- 函式如果不成功，不帶來任何影響。
- sets、multisets、maps、multimaps 和 strings 都支援。

```
iterator container::insert (iterator pos, const T& value)
```

- 在迭代器 *pos* 的位置上安插一個 *value* 副本。
- 傳回新元素的位置。
- 對於關聯式容器（sets、multisets、maps 和 multimaps），*pos* 只作爲一個提示，指向安插時必要的搜尋動作的起始建議位置。如果 *value* 剛好可安插於 *pos* 之後，則此函式具有「分期攤還之常數時間」複雜度，否則具有對數複雜度。
- 如果容器是 sets 或 maps，並且已內含一個「實值等於 *value*（意即兩者的 *key* 相等）」的元素，則此呼叫無效，並傳回現有元素的位置。
- 對於 vectors 和 deques，這個操作可能導致指向其他元素的某些 iterators 和 references 無效。
- `T` 是容器元素的型別，在 maps 和 multimaps 中是一個 *key*/*value* pair。
- 對於 strings，*value* 並不採用 ***pass by reference***。
- 對於 vectors 和 deques，如果元素的複製動作（***copy*** 建構式和 `operator=`）不丟擲異常，則此函式一旦失敗並不會帶來任何影響。對於所有其他容器，函式一旦失敗並不會帶來任何影響。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::insert (iterator pos, size_type num, const T& value)`

- 在迭代器 *pos* 的位置上安插 *num* 個 *value* 副本。
- 對於 vectors 和 deques，此操作可能導致指向其他元素的 iterators 和 references 失效
- T 是容器元素的型別，在 maps 和 multimaps 中是一個 *key/value* pair。
- 對於 strings，*value* 並不採用 ***pass by reference***。
- 對於 vectors 和 deques，如果元素複製動作（***copy*** 建構式和 operator=）不丟擲異常，則函式失敗亦不會帶來任何影響。對於 lists，函式若失敗不會帶來任何影響。
- vectors、deques、lists 和 strings 都支援。

`void container::insert (InputIterator beg, InputIterator end)`

- 將區間 [*beg*,*end*) 內所有元素的副本安插於關聯式容器內。
- 此函式是個 member template（參見 p11），因此只要源區間的元素可轉換爲容器元素的型別，本函式就可派上用場。
- sets、multisets、maps、multimaps 和 strings 都支援。

```
void container::insert (iterator pos, InputIterator beg,
                                      InputIterator end)
```

- 將區間 [*beg*,*end*) 內所有元素的副本安插於迭代器 *pos* 所指的位置上。
- 此函式是個 member template（參見 p11），因此只要源區間的元素可轉換爲容器元素的型別，本函式就可派上用場。
- 對於 vectors 和 deques，此操作可能導致指向其他元素的 iterators 和 references 失效。
- 對於 lists，此函式若失敗不會帶來任何影響。
- vectors、deques、lists 和 strings 都支援。

`void container::push_front (const T& value)`

- 安插 *value* 的副本，使成爲第一個元素。
- T 是容器元素的型別。
- 相當於 `insert(begin(),value)`。
- 對於 deques，此一操作會造成「指向其他元素」的 iterators 失效，而「指向其他元素」的 references 仍保持有效。
- 此函式若失敗不會帶來任何影響。
- deques 和 lists 都支援。

`void container::push_back (const T& value)`

- 安插 *value* 的副本，使成爲最後一個元素。
- T 是容器元素的型別。
- 相當於 `insert(end(),value)`。
- 對於 vectors，如果造成記憶體重新配置，此操作會造成「指向其他元素」的 iterators 和 references 失效。

- 對於 deques，此一操作造成「指向其他元素」的 iterators 失效，而「指向（或說代表）其他元素」的 reference 始終有效。
- 此函式若失敗不會帶來任何影響。
- vectors、deques、lists 和 strings 都支援。

```
void list::remove (const T& value)
void list::remove_if (UnaryPredicate op)
```

- `remove()`會移除所有「實值等於 *value*」的元素。
- `remove_if()`會移除所有「使判斷式 *op*(*elem*)結果爲 true」的元素。
- 注意在函式呼叫過程中，*op* 不應改變狀態。詳見 p302, 8.14 節。
- 兩者都會呼叫被移除元素的解構式。
- 剩餘元素的相對次序保持不變（stable）。
- 這是 p378 所討論的 `remove()`演算法的特殊版本。
- `T` 是容器元素的型別。
- 細節和範例見 p170。
- 只要元素的比較動作不丟擲異常，此函式也不丟擲異常。
- 只有 lists 支援這個成員函式。

```
size_type container::erase (const T& value)
```

- 從關聯式容器中移除所有和 *value* 相等的元素。
- 傳回被移除的元素個數。
- 呼叫被移除元素的解構式。
- `T` 是已序（*sorted*）元素的型別。
  - 在 sets 和 multisets 中，`T` 是元素型別。
  - 在 map 和 multimap 中，`T` 是 *key* 的型別。
- 此函式不丟擲異常。
- sets、multisets、maps 和 multimaps 都支援。

```
void container::erase (iterator pos)
iterator container::erase (iterator pos)
```

- 將迭代器 *pos* 所指位置上的元素移除。
- 序列式容器（vectors、deques、lists 和 strings）採用第二形式，傳回後繼元素的位置（或傳回 `end()`）。
- 關聯式容器（sets、multisets、maps 和 multimaps）採用第一形式，無回返值。
- 兩者都呼叫被移除元素的解構式。

- 注意，呼叫者必須確保迭代器 *pos* 有效。例如：
  `coll.erase(coll.end()); // ERROR ➔ undefined behavior`
- 對於 vectors 和 deques，此操作可能造成「指向其他元素」的 iterators 和 references 無效。
- 對於 vectors 和 deques，只要元素複製動作（*copy* 建構式和 `operator=`）不丟擲異常，此函式就不丟擲異常。對於其他容器，此函式不丟擲異常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::erase (iterator beg, iterator end)`
`iterator container::erase (iterator beg, iterator end)`

- 移除區間 [*beg*, *end*) 內的所有元素。
- 序列式容器（vectors、deques、lists 和 strings）採用第二形式，傳回被移除的最後一個元素的下一位置（或傳回 `end()`）。
- 關聯式容器（sets、multisets、maps 和 multimaps）採用第一形式，無回返值。
- 一如區間慣例，始於 *beg*（含）終於 *end*（不含）的所有元素都被移除。
- 呼叫被移除元素的解構式。
- 呼叫者必須確保 *beg* 和 *end* 形成一個有效序列，並且該序列是容器的一部分。
- 對於 vectors 和 deques，此操作可能導致「指向其他元素」的 iterators 和 references 失效。
- 對於 vectors 和 deques，只要元素複製動作（*copy* 建構式和 `operator=`）不丟擲異常，此函式就不丟擲異常。對於其他容器，此函式不丟擲異常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::pop_front ()`

- 將容器的第一個元素移除。
- 相當於 *container*`.erase(`*container*`.begin())`。
- 注意：如果容器是空的，會導致未定義行爲。因此呼叫者必須確保容器至少有一個元素，也就是 `size()>0`。
- 此函式不丟擲異常。
- deques 和 lists 都支援。

`void container::pop_back ()`

- 將容器的最後一個元素移除。
- 相當於 *container*`.erase(--`*container*`.end())`，前提是其中的算式有效。在 vector 中此算式不一定有效（參見 p258）。
- 注意，如果容器爲空，會導致未定義行爲。因此呼叫者必須確保容器至少包含一個元素，也就是 `size()>0`。
- 此函式不丟擲異常。
- vectors、deques 和 lists 都支援。

```
void container::resize (size_type num)
void container::resize (size_type num, T value)
```

- 兩者都將容器大小改爲 *num*。
- 如果 size() 原本就是 *num*，則兩者皆不生效用。
- 如果 *num* 大於 size()，則在容器尾端產生並附加額外元素。第一形式透過 ***default*** 建構式來建構新元素，第二形式則以 *value* 的副本作爲新元素。
- 如果 *num* 小於 size()，則移除尾端元素，直到大小爲 size()。每個被移除元素的解構式都會被呼叫。
- 對於 vectors 和 deques，這些函式可能導致「指向其他元素」的 iterators 和 references 失效。
- 對於 vectors 和 deques，只要元素複製動作（***copy*** 建構式和 operator=）不丟擲異常，這些函式就不丟擲異常。對於 lists，函如果失敗不會帶來任何影響。
- vectors、deques、lists 和 strings 都支援。

```
void container::clear ()
```

- 移除所有元素（將容器清空）。
- 呼叫被移除元素的解構式。
- 此一容器的所有 iterators 和 references 都將失效。
- 對於 vectors 和 deques，只要元素複製動作（***copy*** 建構式和 operator=）不丟擲異常，此函式就不丟擲異常。對於其他容器，此函式不丟擲異常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.8 Lists 的特殊成員函式

```
void list::unique ()
void list::unique (BinaryPredicate op)
```

- 移除 lists 之內相鄰而重複的元素，使每一個元素都不同於下一個元素。
- 第一形式會將所有「和前一元素相等」的元素移除。
- 第二形式的意義是：任何一個元素 *elem*，如果其前一元素是 *e*，而 *elem* 和 *e* 造成二元判斷式 *op(elem, e)* 獲得 true 值，那麼就移除 *elem*[34]。換言之，這個判斷式並非拿元素和其目前的前一緊臨元素比較，而是拿元素和其未被移除的前一元素比較。

[34] 第二版的 unique() 僅在支援 member templates 的系統中可用（參見 p11）。

- 注意，*op* 不應在函式呼叫過程中改變狀態，詳見 p302, 8.1.4 節。
- 被移除元素的解構式會被喚起。
- 這是 p381 `unique()` 演算法的 lists 特別版本。
- 如果「元素比較動作」中不丟擲異常，則此函式亦不丟擲異常。

```
void list::splice (iterator pos, list& source)
```

- 將 *source* 的所有元素搬移到`*this`，並安插到迭代器 *pos* 所指位置。
- 呼叫之後，*source* 清空。
- 如果 *source* 和`*this` 相同，會導致未定義的行爲。所以呼叫端必須確定 *source* 和`*this` 是不同的 lists。如果要移動同一個 lists 內的元素，應該使用稍後提及的其他 `splice()` 形式。
- 呼叫者必須確定 *pos* 是`*this` 的一個有效位置；否則會導致未定義的行爲。
- 本函式不丟擲異常。

```
void list::splice (iterator pos, list& source, iterator sourcePos)
```

- 從 *source* list 中，將位於 *sourcePos* 位置上的元素搬移至 `*this`，並安插於迭代器 *pos* 所指位置。
- *source* 和 `*this` 可以相同。這種情況下，元素將在 lists 內部被搬移。
- 如果 *source* 和 `*this` 不是同一個 list，在此操作之後，其元素個數少 1。
- 呼叫者必須確保 *pos* 是 `*this` 的一個有效位置、*sourcePos* 是 *source* 的一個有效迭代器，而且 *sourcePos* 不是 `source.end()`；否則會導致未定義行爲。
- 此函式不丟擲異常。

```
void list::splice (iterator pos, list& source,
                   iterator sourceBeg, iterator sourceEnd)
```

- 從 *source* list 中，將位於`[`*sourceBeg*`,`*sourceEnd*`)` 區間內的所有元素搬移到`*this`，並安插於迭代器 *pos* 所指位置。
- *source* 和 `*this` 可以相同。這種情況下，*pos* 不得爲被移動序列的一部分，而元素將在 lists 內部移動。
- 如果 *source* 和 `*this` 不是同一個 list，在此操作之後，其元素個數將減少。
- 呼叫者必須確保 *pos* 是 `*this` 的一個有效位置、*sourceBeg* 和 *sourceEnd* 形成一個有效區間，該區間是 *source* 的一部分；否則會導致未定義的行爲。
- 本函式不丟擲異常。

```
void list::sort ()
void list::sort (CompFunc op)
```

- 對 lists 內的所有元素進行排序。
- 第一型式以 `operator<` 對 lists 中的所有元素進行排序。

- 第二型式透過如下的 *op* 動作來比較兩元素，進而對 lists 中的所有元素排序[35]：*op*(*elem1*,*elem2*)。
- 實值相同的元素，其順序保持不變（除非有異常被丟出）。
- 這是 p397 所討論的 sort() 和 stable_sort()演算法的「list 特殊版本」。

```
void list::merge (list& source)
void list::merge (list& source, CompFunc op)
```

- 將 lists *source* 內的所有元素併入*this。
- 呼叫後 *source* 變成空容器。
- 如果*this 和 *source* 在排序準則 operator< 或 op 之下已序（*sorted*），則新產生的 lists 也是已序。嚴格地說，標準規格書要求兩個 lists 必須已序，但實際上對無序的 lists 進行合併也是可能的，不過使用前最好先確認一下。
- 第一形式採用 operator< 作爲排序準則。
- 第二形式採用以下的 op 動作作爲可有可無的排序準則，以此比較兩個元素的大小[36]：op(*elem*,*sourceElem*)
- 這是 p416 所討論的 merge()演算法的 list 特殊版本。
- 只要元素的比較動作不丟擲異常，此函式萬一失敗也不會造成任何影響。

```
void list::reverse ()
```

- 將 lists 中的元素顛倒次序。
- 這是 p386 所討論的 reverse()演算法的「list 特殊版本」。
- 本函式不丟擲異常。

6.10.9 對配置器（Allocator）支援

所有 STL 容器都能夠與某個配置器物件（allocator object）所定義的某種特定記憶體模型（memory model）搭配合作（詳見第 15 章）。本節討論的是支援配置器的各個成員。標準容器要求：配置器（型別）的每一個實體都必須是可互換的（interchangeable），所以某一容器的空間，可透過另一同型容器釋放之。因此，元素（及其儲存空間）在同型的兩個容器之間移動，並不會出現問題。

[35] sort()的第二形式僅在支援 member templates 的系統中可用（參見 p11）

[36] merge()的第二形式僅在支援 member templates 的系統中可用（參見 p11）

基本的配置器相關成員（Fundamental Allocator Members）

container::**allocator_type**

- 配置器型別。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

allocator_type *container*::**get_allocator** () const

- 傳回容器的記憶體模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

帶有「可選擇之配置器參數」的建構式

explicit *container*::*container* (const Allocator& *alloc*)

- 產生一個新的空白容器，使用 *alloc* 作爲記憶體模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

container::*container* (const CompFunc& *op*, const Allocator& *alloc*)

- 產生一個新的空白容器，使用 *alloc* 作爲記憶體模型，並以 *op* 爲排序準則。
- *op* 排序準則必須定義 **strict weak ordering**（參見 p176）。
- sets、multisets、maps 和 multimaps 都支援。

container::*container* (size_type *num*, const T& *value*,
const Allocator& *alloc*)

- 產生一個擁有 *num* 個元素的容器，使用 *alloc* 作爲記憶體模型。
- 所生成的元素都是 *value* 的副本。
- T 是容器元素的型別。注意，對於 strings，*value* 採用 ***by value*** 的型式傳遞。
- vectors、deques、lists 和 strings 都支援。

container::*container* (InputIterator *beg*, InputIterator *end*,
const Allocator& *alloc*)

- 產生一個容器，以區間 [*beg*,*end*) 內的所有元素爲初值，並使用 *alloc* 作爲記憶體模型。
- 此函式是一個 member template（參見 p11）。所以只要源序列的元素能夠轉換爲容器元素的型別，此函式就可執行。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

container::*container* (InputIterator *beg*, InputIterator *end*,
const CompFunc& *op*, const Allocator& *alloc*)

- 產生一個以 *op* 爲排序準則的容器，以區間 [*beg*,*end*) 中的所有元素爲初值，並使用 *alloc* 作爲記憶體模型。

- 本函式是一個 member template（參見 p11）。所以只要源序列的元素能夠轉換爲容器元素的型別，本函式就可執行。
- 排序準則 *op* 必須定義 **strict weak ordering**（參見 p176）
- sets、multisets、map 和 multimap 都支援。

6.10.10 綜觀 STL 容器的異常處理

p139, 5.11.2 節曾指出，不同的容器在異常發生時，給予不同程度的保證。通常 C++ 標準程式庫在異常發生時並不會洩漏資源或破壞容器的恆常特性（invariants）。有些操作提供更強的保證（前提是其引數必須滿足某些條件）：它們可以保證 **commit-or-rollback**（意思是「要嘛成功，要嘛不帶來任何影響」），甚至可以保證絕不丟擲異常。表 6.35 列出所有支援更強烈保證的操作函式[37]。

對於 vectors、deques 和 lists 而言，`resize()`也提供特別保證。其行爲或許相當於 `erase()`，或許相當於 `insert()`，或許相當於什麼也沒做。

```
void container::resize (size_type num, T value = T())
{
    if (num > size()) {
        insert (end(), num-size(), value);
    }
    else if (num < size()) {
        erase (begin()+num, end());
    }
}
```

因此，它所提供的保證就是「`erase()`和 `insert()`兩者所提供的保證」的組合（參見 p244）。

[37] 感謝 Greg Colvin 和 Dave Abrahams 提供這個表格。

操作	頁次	保證
`vector::push_back()`	241	要不成功，要不無任何影響。
`vector::insert()`	240	要不成功，要不無任何影響 — 前提是元素的複製/賦值動作不丟擲異常
`vector::pop_back()`	243	不丟擲異常
`vector::erase()`	242	不丟擲異常 — 前提是元素的複製/賦值動作不丟擲異常
`vector::clear()`	244	不丟擲異常 — 前提是元素的複製/賦值動作不丟擲異常
`vector::swap()`	237	不丟擲異常
`deque::push_back()`	241	要不成功，要不無任何影響。
`deque::push_front()`	241	要不成功，要不無任何影響。
`deque::insert()`	240	要不成功，要不無任何影響 — 前提是元素的複製/賦值動作不丟擲異常
`deque::pop_back()`	243	不丟擲異常
`deque::pop_front()`	243	不丟擲異常
`deque::erase()`	242	不丟擲異常 — 前提是元素的複製/賦值動作不丟擲異常
`deque::clear()`	244	不丟擲異常 — 前提是元素的複製/賦值動作不丟擲異常
`deque::swap()`	237	不丟擲異常
`list::push_back()`	241	要不成功，要不無任何影響。
`list::push_front()`	241	要不成功，要不無任何影響。
`list::insert()`	240	要不成功，要不無任何影響。
`list::pop_back()`	243	不丟擲異常
`list::pop_front()`	243	不丟擲異常
`list::erase()`	242	不丟擲異常
`list::clear()`	244	不丟擲異常
`list::remove()`	242	不丟擲異常 — 前提是元素的比較動作不丟擲異常
`list::remove_if()`	242	不丟擲異常 — 前提是判斷式 *predicate* 不丟擲異常
`list::unique()`	244	不丟擲異常 — 前提是元素的比較動作不丟擲異常
`list::splice()`	245	不丟擲異常
`list::merge()`	246	要不成功，要不無任何影響 — 前提是元素的比較動作不丟擲異常
`list::reverse()`	246	不丟擲異常
`list::swap()`	237	不丟擲異常
`[multi]set::insert()`	240	要不成功，要不無任何影響 — 對單個元素而言
`[multi]set::erase()`	242	不丟擲異常
`[multi]set::clear()`	244	不丟擲異常
`[multi]set::swap()`	237	不丟擲異常 — 前提是對「比較準則」執行複製/賦值動作時不丟擲異常
`[multi]map::insert()`	240	要不成功，要不無任何影響 — 對單個元素而言
`[multi]map::erase()`	242	不丟擲異常
`[multi]map::clear()`	244	不丟擲異常
`[multi]map::swap()`	237	不丟擲異常 — 前提是對「比較準則」執行複製/賦值動作時不丟擲異常

表 6.35 「異常發生時帶有特殊保證」的各個容器操作函式