SGI STL 源碼剖析

The Annotated STL Sources

向專家學習
強型檢驗、記憶體管理、演算法、資料結構、
及 STL 各類組件之實作技術

侯 捷 著

源碼之前
了無秘密

獻給每一位對 GP/STL 有所渴望的人

天下大事 必作於細

– 侯 捷 –

庖丁解牛[1]

侯捷自序

這本書的寫作動機，純屬偶然。

2000 年下半，我開始爲寫作計劃中的《泛型技術》陸續做準備並熱身。爲了對泛型編程技術以及 STL 實作技術有更深的體會，以便在講述整個 STL 的架構與應用時更能虎虎生風，我常常深入到 STL 源碼去刨根究底一番。2001/02 的某一天，我突然有所感觸：既然花了大把精力看過 STL 源碼，寫了眉批，做了整理，何不把它再加一點功夫，形成一個更完善的面貌後出版？對我個人而言，一份註解詳盡的 STL 源碼，價值不扉；如果我從中獲益，一定也有許多人能夠從中獲益。

這樣的念頭使我極度興奮。剖析大架構本是侯捷的拿手，這個主題又可以和《泛型技術》相呼應。於是我便一頭栽進去了。

我選擇 SGI STL 做爲剖析對象。這份實作版本的可讀性極佳，運用極廣，被選爲 GNU C++ 的標準程式庫，又開放自由運用。愈是細讀 SGI STL 源碼，愈令我震驚抽象思考層次的落實、泛型編程的奧妙、及其效率考量的綿密。不僅最爲人廣泛運用的各種資料結構（data structurs）和演算法（algorithms）在 STL 中有良好的實現，連記憶體配置與管理也都重重考慮了最佳效能。一切的一切，除了實現軟體積木的高度復用性，讓各種組件（components）得以靈活搭配運用，更考量了實用上的關鍵議題：效率。

[1] 莊子養生主：「彼節間有間，而刀刃者無厚；以無厚入有間，恢恢乎其於游刃必有餘地矣。」侯捷不讓，以此自況。

這本書不適合 C++ 初學者，不適合 Genericity（泛型技術）初學者，或 STL 初學者。這本書也不適合帶領你學習物件導向（Object Oriented）技術 — 是的，STL 與物件導向沒有太多關連。本書前言清楚說明了書籍的定位和合適的讀者，以及各類基礎讀物。如果你的 Generic Programming/STL 實力足以閱讀本書所呈現的源碼，那麼，恭喜，你踏上了基度山島，這兒有一座大寶庫等著你。源碼之前了無秘密，你將看到 `vector` 的實作、`list` 的實作、`heap` 的實作、`deque` 的實作、`RB-tree` 的實作、`hash-table` 的實作、`set/map` 的實作；你將看到各種演算法（排序、搜尋、排列組合、資料搬移與複製…）的實作；你甚至將看到底層的 memory pool 和高階抽象的 traits 機制的實作。那些資料結構、那些演算法、那些重要觀念、那些編程實務中最重要最根本的珍寶，那些蜇伏已久彷彿已經還給老師的記憶，將重新在你的腦中閃閃發光。

人們常說，不要從輪子重新造起，要站在巨人的肩膀上。面對扮演輪子、肩膀角色的這些 STL 組件，我們是否有必要深究其設計原理或實作細節呢？答案因人而異。從應用的角度思考，你不需要探索實作細節（然而相當程度地認識底層實作，對實務運用有絕對的幫助）。從技術研究與提昇的角度看，深究細節可以讓你徹底掌握一切；不論是爲了重溫資料結構和演算法，或是想要扮演別人的輪子、肩膀，都可以因此獲得深厚紮實的基礎。

天下大事，必作於細！

別忘了，參觀飛機工廠不能讓你學得流體力學，也不能讓你學會開飛機。倒是，如果你會開飛機又懂流體力學，參觀飛機工廠可以帶給你最大的樂趣和價值。

The Annotated STL Sources

我開玩笑地對朋友說，這本書出版，給大學課程中的「資料結構」和「演算法」兩門授課老師出了個難題。幾乎所有可能的作業題目（複雜度證明題除外），本書都有了詳盡的解答。然而，如果學生能夠從龐大的 SGI STL 源碼中乾淨抽出某一部份，加上自己的包裝，做爲呈堂作業，也足以證明你有資格獲得學分和高分。事實上，追蹤一流作品並於其中吸取養份，遠比自己關起門來寫個三流作品，價值高得多 — 我的確認爲 99.99 % 的程式員所寫的程式，在 SGI STL 面前都是三流程度 ☺。

侯捷 2001/05/30 新竹 ▪ 臺灣

www.jjhou.com (繁體)
www.csdn.net/expert/jjhou (簡體)
jjhou@jjhou.com

p.s. 以下三書互有定位，互有關聯，彼此亦相呼應。爲了不重複講述相同的內容，我會在適當時候提醒讀者在哪本書上獲得更多資料：

- 《多型與虛擬》，內容涵括：C++ 語法、語意、物件模型，物件導向精神，小型 framework 實作，OOP 專家經驗，設計樣式（design patterns）導入。
- 《泛型技術》，內容涵括：語言層次（C++ templates 語法、Java generic 語法、C++ 運算子多載化），STL 原理介紹與架構分析，STL 現場重建，STL 深度應用，STL 擴充示範。
- 《STL 源碼剖析》，內容涵括：STL 所有組件之實作技術和其背後原理。

目 錄

前言

本書定位

C++ 標準程式庫是個偉大的作品。它的出現，相當程度地改變了 C++ 程式的風貌以及學習模式[1]。在納入 STL（Standard Template Library）的同時，標準程式庫的所有組件，包括大家早已熟悉的 `string`、`stream` 等等，亦全部以 template 改寫過。整個標準程式庫沒有太多的 OO（Object Oriented），倒是無處不存在 GP（Generic Programming）。

C++ 標準程式庫中隸屬 STL 範圍者，粗估當在 90% 以上。對軟體開發而言，STL 是尖甲利兵，可以節省你許多時間。對編程技術而言，STL 是金櫃石室 — 所有與編程工作最有直接密切關聯的一些最被廣泛運用的資料結構和演算法，STL 都有實作，並符合最佳（或極佳）效率。不僅如此，STL 的設計思維，把我們提昇到另一個思想高點，在那裡，物件的耦合性（coupling）極低，復用性（reusability）極高，各種組件可以獨立設計而又可以靈活無罅地結合在一起。是的，STL 不僅僅是程式庫，它其實具備 framework 格局，允許使用者加上自己的組件，與之融合並用。

從應用角度來說，任何一位 C++ 程式員都不應該捨棄現成、設計良好而又效率極佳的標準程式庫，卻「入太廟每事問」地事事物物從輪子造起 — 那對組件技術及軟體工程是一大嘲諷。然而對於一個想要深度鑽研 STL 以便擁有擴充能力的人，

[1] 請參考 *Learning Standard C++ as a New Language*, by Bjarne Stroustrup, C/C++ Users Journal 1999/05。中譯文 http://www.jjhou.com/programmer-4-learning-standard-cpp.htm

相當程度地追蹤 STL 源碼是必要的功課。是的，對於一個想要充實資料結構與演算法等固有知識，並提昇泛型編程技法的人，「入太廟每事問」是必要的態度，追蹤 STL 源碼則是提昇功力的極佳路線。

想要良好運用 STL，我建議你看 [Josuttis99]；想要認識 STL 的架構和設計思維，以及 STL 的詳細規格，我建議你看 [Austern98]；想要從語法層面開始，學理與應用得兼，宏觀與微觀齊備，我建議你看《泛型技術》；想要深入 STL 實作技法，一窺大家風範，提昇自己的編程功力，我建議你看手上這本《STL 源碼剖析》— 事實上就在下筆此刻，你也找不到任何一本相同定位的書[2]。

合適的讀者

本書不適合 STL 初學者（當然更不適合 C++ 初學者）。本書不是物件導向（Object Oriented）相關書籍。本書不適合用來學習 STL 的各種應用。

對於那些希望深刻瞭解 STL 實作細節，俾得以提昇對 STL 的擴充能力，或是希望藉由觀察 STL 源碼，學習世界一流程式員身手，並藉此徹底瞭解各種被廣泛運用之資料結構和演算法的人，本書最適合你。

最佳閱讀方式

無論你對 STL 認識了多少，我都建議你第一次閱讀本書時，採循序漸進的方式，按著我所安排的章節進行。隨著個人功力的深淺，你可以或快或慢地將全書瀏覽一遍，並依個人的興趣或需要，深入其中。初次閱讀最好循序漸進，理由是，舉個例子，所有容器（containers）的定義式一開頭都會出現空間配置器（allocator）的運用，我可以在最初數次提醒你空間配置器於第 2 章介紹過，但我無法遍及全書一再一再提醒你。又例如，源碼之中時而會出現一些全域函式呼叫動作，尤其是定義於 `<stl_construct.h>` 之中用於物件建構與解構的基本函式，定義於 `<stl_uninitialized.h>` 之中用於記憶體管理的基本函式，以及定義於

[2] 最新消息：*The C++ Standard Template Library*, by P.J.Plauger, Alexander Al. Stepanov, Meng Lee, David R. Musser, Prentice Hall 2001/03，是一本與本書定位相近的書，但在表現方式上有相當大的不同。

<stl_algobase.h> 之中的各種基本演算法。如果那些全域函式已經在先前章節介紹過，我很難保證每次都提醒你 — 那是一種顧此失彼、苦不堪言的勞役，並且容易造成閱讀上的累贅。

我所選擇的剖析對象

本書名爲《STL 源碼剖析》，然而 STL 實作品百花齊放，不論就技術面或可讀性，皆有高下之分。選擇一份好的產品，就學習而言，當然是極爲重要的。我選擇的剖析對象是聲名最著，也是經過我個人比較之後評價最高的一個產品：SGI（Silicon Graphics Computer Systems, Inc.）版本。這份由 STL 之父 Alexander Stepanov、經典書籍 *Generic Programming and the STL* 的作者 Matthew H. Austern、STL 耆宿 David Musser 等人投注心力的 STL 實作品，不論在技術層次、源碼組織、源碼可讀性上，均有卓越的表現。這份產品被納爲 GNU C++ 標準程式庫，任何人皆可從網際網路上下載 GNU C++ 編譯器，從而獲得整份 STL 源碼，並獲得自由運用的權力（詳見 1.8 節）。

我所選用的是 cygnus[3] C++ 2.91.57 for Windows 版本。我並未刻意追求最新版本，一來書籍不可能永遠呈現最新的軟體版本 — 軟體更新永遠比書籍改版快速，二來本書的根本目的在建立各位對 STL 巨觀架構和微觀技術的掌握，以及源碼的閱讀能力，這種核心知識的形成與源碼版本的關係不是那麼唇齒相依，三來 SGI STL 實作品自從搭配 GNU C++2.8 以來已經十分穩固，變異極微，而我所選擇的 2.91 版本，表現相當良好；四來這個版本的源碼比後來的版本更容易閱讀，因爲許多內部變數名稱並不採用下劃線（underscore） — 下劃線在變數命名規範上有其價值，但到處都是下劃線則對大量閱讀相當不利。當然，這個版本有一部份未能跟上 C++ 標準化的腳步，例如 `bitset`, `valarray`, `auto_ptr`，這一部份我將集中在第 9 章，以 GNU C++ 2.95 for Solaris 所附的符合 C++ 標準的 SGI STL 爲解說對象，並明確提示讀者。

網路上有個 STLport（http://www.stlport.org）站點，提供一份以 SGI STL 爲藍本

[3] 關於 cygnus，GNU 源碼開放精神，以及自由軟體基金會（FSF），請見 1.3 節介紹。

的高度可攜性實作品。本書附錄 D 列有孟岩先生所寫的文章，介紹 STLport 移植到 Visual C++ 和 C++ Builder 的經驗。

各章主題

本書假設你對於 STL 已有基本認識與某種程度之運用經驗。因此除了第一章略作介紹之外，立刻深入 STL 技術核心，並以 STL 六大組件（components）爲依據，安排章節之進行。以下是各章名稱，這樣的次序安排大抵可使每一章所剖析的主題能夠於先前章節中獲得充份的基礎。當然，技術之間的關連錯綜複雜，不可能存在單純的線性關係，這樣的安排也只能說是盡最大的努力。這種安排亦呼應《泛型技術》的章節，使兩本書有更強烈的對應關係。

第 1 章　STL 概論與實作版本簡介

第 2 章　空間配置器（allocator）

第 3 章　迭代器（iterators）概念與 traits 編程技法

第 4 章　序列式容器（sequence containers）

第 5 章　關聯式容器（associated containers）

第 6 章　演算法（algorithms）

第 7 章　仿函式 or 函式物件（functors, or function objects）

第 8 章　配接器（adaptor）

編譯工具

本書主要探索 SGI STL 源碼，並提供少量測試程式。如果測試程式只做標準的 STL 動作，不涉及 SGI STL 實作細節，那麼我會在 VC6、CB4、cygnus 2.91 for Windows 上測試它們。

隨著對 SGI STL 源碼的掌握程度增加，我們可以大膽做些練習，將 SGI STL 內部介面打開，或是修改某些 STL 組件，加上少量輸出動作，以觀察組件的運作過程。

這種情況下，操練的對象既然是 SGI STL，當然我也就使用 GNU C++ 來編譯[4]。

中英術語的運用風格

我曾經發表過一篇名爲「技術引導乎 文化傳承乎」的文章，闡述我對專業電腦書籍的中英術語運用態度。文章發表於網路上的 CompBook 論壇，並收錄於侯捷網站 http://www.jjhou.com/article99-14.htm。以下簡單敘述我的想法。

爲了學術界與業界的習慣，也爲了與全球科技接軌，並且也因爲我所撰寫的是供專業人士閱讀的書籍而非科普讀物，我決定大量保留專業領域中被朗朗上口的英文術語。朗朗上口與否，是見仁見智的問題，我以個人閱歷做爲抉擇的依據。

做爲一個並非以英語爲母語的族裔，我們對英文的閱讀困難其實不在單字，而在整句整段的文意。做爲一項技術的學習者，我們的困難並不在術語本身（那只是個符號），而在術語背後的技術意義。

熟悉並使用原文術語，至爲重要。原因很簡單，在資訊科技領域裡，你必須與全世界接軌。中文技術書籍的價值不在於「建立本國文化」或「讓它成爲一本道地的中文書」或「掃除英漢字典的需要」。中文技術書籍的重要價值，在於引進技術、引導學習、掃平閱讀障礙、增加學習效率。

絕大部份我所採用的英文術語都是名詞。但某些動詞或形容詞也有必要讓讀者知道原文（我會時而中英並列，並使用斜體英文），原因是：

- C++ 編譯器的錯誤訊息並未中文化，萬一錯誤訊息中出現以下字眼：*unresolved*, *instantiated*, *ambiguous*, *override*，而編寫程式的你卻不熟悉或不懂這些動詞或形容詞的技術意義，就不妙了。
- 有些動作關係到 library functions，而 library functions 的名稱並未中文化☺，例如 *insert*, *delete*, *sort*。因此視狀況而定，我可能會選擇使用英文。

[4] SGI STL 事實上是個高度可攜的產品，不限使用於 GNU C++。從它對各種編譯器的環境組態設定（1.8.3 節）便可略知一二。網路上有一個 STLport 組織，不遺餘力地將 SGI STL 移植到各種編譯平台上。請參閱本書附錄 D。

- 如果某些術語關係到程式語言關鍵字（keyword），爲了讓讀者有最直接的感受與聯想，我會採用原文，例如 static, private, protected, public, friend, inline, extern。

版面像一張破碎的臉？

大量中英夾雜的結果，無法避免地就是版面的「破碎」。但，爲了實現合宜的表達方式，犧牲版面的「全中文化」在所難免。我將儘量以版面技術來達到視覺上的順暢，換言之我將採用不同的字形來代表不同屬性的術語。如果把英文術語視爲一種符號，這些中英夾雜但帶有特殊字形的版面，並不會比市面上琳瑯滿目的許多圖解軟體使用手冊來得突兀。我慣用的版面都已經過一再試鍊，並且過去以來獲得許多讀者的贊同。

英文術語採用原則

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

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

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

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

- 編程基礎術語，採用中文。例如：函式、指標、變數、常數。本書的英文術語絕大部份都與 C++/OOP/GP（Generic Programming）相關。
- 簡單而朗朗上口的詞，視情況可能直接使用英文：input, output, lvalue, rvalue...
- 讀者有必要認識的英文名詞，不譯：template, class, object, exception, scope,

namespace。

- 長串、有特定意義、中譯名稱拗口者，不譯：explicit specialization, partial specialization, using declaration, using directive, exception specialization。
- 運算子名稱，不譯：copy assignment 運算子，member access 運算子，arrow 運算子，dot 運算子，address of 運算子，dereference 運算子...
- 業界慣用詞，不譯：constructor, destructor, data member, member function, reference。
- 涉及 C++ 關鍵字者，不譯：public, private, protected, friend, static,
- 意義良好，發音簡短，流傳頗眾的譯詞，用之：多型（polymorphism），虛擬函式（virtual function）、泛型（genericity）…
- 譯後可能失掉原味而無法完全彰顯原味者，中英並列。
- 重要的動詞、形容詞，時而中英並列：模稜兩可（*ambiguous*），決議（*resolve*），改寫（*override*），引數推導（*argument deduced*），具現化（*instantiated*）。
- STL 專用術語：採用中文，如迭代器（iterator）、容器（container）、仿函式（functor）、配接器（adaptor）、空間配置器（allocator）。
- 資料結構專用術語：盡量採用英文，如 `vector`, `list`, `deque`, `queue`, `stack`, `set`, `map`, `heap`, `binary search tree`, `RB-tree`, `AVL-tree`, `priority queue`.

援用原文詞，或不厭其煩地中英並列，獲得的一項重要福利是：本書得以英文做爲索引憑藉。

本書附錄 B 列有英中繁簡術語對照表。

關於索引

本書索引以英文爲主，中文爲輔。中文的出現是爲了說明某些術語在本書的譯名。上下層關係皆以簡潔之英文如 for, as, with, by, of … 連結。此或有不夠週延之處，實爲多方考量下的權宜結果。

版面字型格式

中文

- 本文：細明 9.5pt
- 標題：[illegible]

- 視覺加強：

- 一般文字，Times New Roman, 9.5pt，例如：class, object, member function, data member, base class, derived class, private, protected, public, reference, template, namespace, function template, class template, local, global
- 動詞或形容詞，*Times New Roman* 斜體 9.5pt，例如：*resolve, ambiguous, override, instantiated*
- class 名稱，`Lucida Console` 8.5pt，例如：`stack, list, map`
- 程式碼識別符號，`Courier New` 8.5pt，例如：`int, min(SmallInt*, int)`
- 長串術語，Arial 9pt，例如：member initialization list, name return value, using directive, using declaration, pass by value, pass by reference, function try block, exception declaration, exception specification, stack unwinding, function object, class template specialization, class template partial specialization…
- exception types 或 *iterator types* 或 *iostream manipulators*，*Lucida Sans* 9pt，例如： *bad_alloc, back_inserter, boolalpha*
- 運算子名稱及某些特殊動作，Footlight MT Light 9.5pt，例如：copy assignment 運算子，dereference 運算子，address of 運算子，equality 運算子，function call 運算子，constructor, destructor, default constructor, copy constructor, virtual destructor, memberwise assignment, memberwise initialization
- 程式碼，`Courier New` 8.5pt，例如：

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

要在整本書中維護一貫的字形風格而沒有任何疏漏，很不容易，許多時候不同類型的術語搭配起來，就形成了不知該用哪種字形的困擾。排版者顧此失彼的可能也不是沒有。因此，請注意，**各種字形的運用，只是爲了讓您閱讀時有比較好的效果，其本身並不具任何其他意義**。局部的一致性更重於全體的一致性。

SGI STL 雖然是可讀性最高的一份 STL 源碼，但其中並沒有對實作程序乃至於實作技巧有什麼文字註解，只偶而在檔案最前面有一點點總體說明。雖然其符號名

稱有不錯的規劃，真要仔細追蹤源碼，仍然曠日費時。因此本書不但在正文之中解說其設計原則或實作技術，也直接在源碼中加上許多註解。這些註解皆以藍色標示。條件式編譯（#ifdef）亦以藍色標示。函式呼叫動作以紅色表示，巢狀定義亦以紅色表示。classes 名稱、data members 名稱和 member functions 名稱大部份皆以**粗體**表示。特別需要提醒的地方（包括 template 預設引數、長度很長的巢狀式定義）則加上灰階網底。例如：

```
template <class T, class Alloc = alloc>  // 預設使用 alloc 爲配置器
class vector {
public:
  typedef T           value_type;
  typedef value_type* iterator;
...
protected:
  // vector 採用簡單的線性連續空間。以兩個迭代器 start 和 end 分別指向頭尾，
  // 並以迭代器 end_of_storage 指向容量尾端。容量可能比(尾-頭)還大，
  // 多餘的空間即備用空間。
  iterator start;
  iterator finish;
  iterator end_of_storage;

  void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);  // 配置空間並設初值
    finish = start + n;                   // 調整水位
    end_of_storage = finish;              // 調整水位
  }
...
};

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER
template <class T, class Alloc>
inline void swap(vector<T, Alloc>& x, vector<T, Alloc>& y) {
  x.swap(y);
}
#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */
```

又如：

```
// 以下配接器用來表示某個 Adaptable Binary Predicate 的邏輯負值
template <class Predicate>
class binary_negate
  : public binary_function<typename Predicate::first_argument_type,
                           typename Predicate::second_argument_type,
                           bool> {
...
};
```

這些作法可能在某些地方有少許例外（或遺漏），唯一不變的原則就是儘量讓讀者一眼抓住源碼重點。花花綠綠的顏色乍見之下或許不習慣。啊，多看幾眼你就會喜歡它 ☺。這些經過註解的 STL 源碼並以 Microsoft Word 97 檔案格式，連同 SGI STL 源碼，置於侯捷網站供自由下載[5]。噢，是的，STL 涵蓋面積廣大，源碼浩繁，本書僅能就具代表性者加以剖析。如果你感興趣的某些細節未涵蓋於書中，或可自行上網查閱這些經過整理的源碼檔案。

線上服務

侯捷網站（網址見於封底）是我的個人網站。我的所有作品，包括本書，都在此網站上提供服務，包括：

- 勘誤和補充
- 技術討論
- 程式碼下載
- 部份電子檔下載

附錄 C 對侯捷網站有一些導引介紹。

推荐讀物

詳見附錄 A。這些精選讀物可爲你建立紮實的泛型思維（Generic Paradigm）理論基礎與紮實的 STL 實務應用能力。

[5] 下載這些檔案不會發生版權問題。詳見 1.3 節有關於自由軟體基金會（FSF）、源碼開放（open source）精神以及各種授權聲明。

1

STL 概論與技術簡介

STL，雖然是一套程式庫（library），卻不是一般人印象中的程式庫，而是一個有著劃時代意義，背後擁有先進技術與深厚理論的產品。說它是產品也可以，說它是規格也可以，說它是軟體組件技術發展史上的一個大突破點，也可以。

1.1 STL 概論

長久以來，軟體界一直希望建立一種可重複運用的東西，以及一種可以製造出「可重複運用的東西」的方法，讓工程師 / 程式員的心血不致於隨時間遷移、人事異動、私心欲念、人謀不臧[1]而煙消雲散。從副程式（subroutines）、程序（procedures）、函式（functions）、類別（classes），到函式庫（function libraries）、類別庫（class libraries）、各種組件技術（component tech.），從模組化（modulized）設計、物件導向（object oriented）設計，到樣式（patterns）的歸納整理，無一不是軟體工程的漫漫奮鬥史。

爲的就是復用性（reusability）的提昇。

復用性必須建立在某種標準之上 — 不論是語言層次的標準，或資料交換的標準，或通訊協定的標準。但是，許多工作環境下，就連軟體開發最基本的資料結構（data structures）和演算法（algorithms）都還遲遲未能有一套標準。大量程式員被迫從事大量重複的工作，竟是爲了完成前人早已完成而自己手上並未擁有的程式碼。這不僅是人力資源的浪費，也是挫折與錯誤的來源。

[1] 後兩者其實是科技到達不了的幽暗世界。就算 STL, COM, CORBA, OO, Patterns 也無能爲力。

爲了建立資料結構和演算法的一套標準，並且降低其間的耦合（coupling）關係以提昇各自的獨立性、彈性、交互操作性（相互合作性，interoperability），C++ 社群裡誕生了 STL。

STL 的價值在兩方面。低層次而言，STL 帶給我們一套極具實用價值的零組件，以及一個整合的組織。這種價值就像 MFC 或 VCL 之於 Windows 軟體開發過程所帶來的價值一樣，直接而明瞭，令大部份人有最直接明顯的感受。除此之外 STL 還帶給我們一個高層次的、以泛型思維（Generic Paradigm）爲基礎的、系統化的、條理分明的「軟體組件分類學（components taxonomy）」。從這個角度來看，STL 是個抽象概念庫（library of abstract concepts），這些「抽象概念」包括最基礎的 *Assignable*（可被賦值）、*Default Constructible*（不需任何引數就可建構）、*Equality Comparable*（可判斷是否等同）、*LessThan Comparable*（可比較大小）、*Regular*（正規）…，稍微高階一點的概念則包括 *Input Iterator*（具輸入功能的迭代器）、*Output Iterator*（具輸出功能的迭代器）、*Forward Iterator*（單向迭代器）、*Bidirectional Iterator*（雙向迭代器）、*Random Access Iterator*（隨機存取迭代器）、*Unary Function*（一元函式）、*Binary Function*（二元函式）、*Predicate*（傳回真假值的一元函式）、*Binary Predicate*（傳回真假值的二元函式）…，更高階的概念包括 *sequence container*（序列式容器）、*associative container*（關聯式容器）…。

STL 的創新價值在於具體敘述了上述這些抽象概念，並加以系統化。

換句話說，STL 所實現的，是依據泛型思維架設起來的一個概念結構。這個以抽象概念（abstract concepts）爲主體而非以實際類別（classes）爲主體的結構，嚴謹地形成了一個介面標準。在此介面之下，任何組件有最大的獨立性，並以所謂的迭代器（iterator）膠合起來，或以所謂的配接器（adaptor）互相配接，或以所謂的仿函式（functor）做爲策略選擇。

目前沒有任何一種程式語言提供任何關鍵字（keyword）實質對應上述所謂的抽象概念。C++ classes 允許我們自行定義型別，C++ templates 允許我們將型別參數化，

兩者結合並透過 traits 編程技法，形成了 STL 的絕佳溫床[2]。

關於 STL 的所謂軟體組件分類學，以及所謂的抽象概念庫，請參考 [Austern98] —沒有任何一本書籍在這方面說得比它更好，更完善。

1.1.1 STL 的誕生

STL 係由 Alexander Stepanov 創造於 1979 年前後，這也正是 Bjarne Stroustrup 創造 C++ 的年代。雖然 David R. Musser 於 1971 開始即在計算機幾何領域中發展並倡導某些泛型程式設計觀念，但早期並沒有任何程式語言支援泛型編程。第一個支援泛型概念的語言是 Ada。Alexander 和 Musser 曾於 1987 開發出一套相關的 Ada library。然而 Ada 在美國國防工業以外並未被廣泛接受，C++ 卻如星火燎原般地在程式設計領域中攻城略地。當時的 C++ 尚未導入 template 性質，然而 Alexander 已經意識到，C++ 允許程式員經由指標以極佳彈性處理記憶體，這正是既要求一般化（泛型）又不能失去效率的一個重要關鍵。

更重要的是，必須研究並實驗出一個「立基於泛型編程之上」的組件庫的完整架構。Alexander 在 AT&T 實驗室以及惠普公司的帕羅奧圖（Hewlett-Packard Palo Alto）實驗室，分別實驗了許多種架構和演算法公式，先以 C 完成，而後再以 C++ 完成。1992 年 Meng Lee 加入 Alex 的專案，成爲 STL 的另一位主要貢獻者。

貝爾(Bell)實驗室的 Andrew Koenig 於 1993 年知道這個研究計劃後，邀請 Alexander 於是年 11 月的 ANSI/ISO C++ 標準委員會會議上展示其觀念。獲得熱烈迴應。Alexander 於是再接再勵於次年夏天的 Waterloo（滑鐵盧[3]）會議開幕前，完成正式提案，並以壓倒性多數一舉讓這個巨大的計劃成爲 C++ 標準規格的一部份。

1.1.2 STL 與 C++ 標準程式庫

1993/09，Alexander Stepanov 和他一手創建的 STL，與 C++ 標準委員會有了第一

[2] 這麼說有點因果混沌。因爲 STL 的成形過程中也獲得了 C++ 的一些重大修改支援，例如 template partial specialization。

[3] 不是拿破崙被威靈頓公爵擊敗的那個地方，而是加拿大安大略湖畔的滑鐵盧市。

次接觸。

當時 Alexander 在矽谷（聖荷西）給了 C++ 標準委員會一個演講，講題是：The Science of C++ Programming。題目很理論，卻非常受歡迎。1994/01/06 Alexander 收到 Andy Koenig（C++ 標準委員會成員，當時的 *C++ Standard* 文件審核編輯）來信，言明如果希望 STL 成爲 C++ 標準程式庫的一部份，可於 1994/01/25 前送交一份提案報告到委員會。Alexander 和 Lee 於是拼命趕工完成了那份提案。

然後是 1994/03 的聖地牙哥會議。STL 在會議上獲得了很好的迴響，但也有許多反對意見。主要的反對意見是，C++ 即將完成最終草案，而 STL 卻是如此龐大，似乎有點時不我予。投票結果壓倒性地認爲應對這份提案給予機會，並把決定性投票延到下次會議。

下次會議到來之前，STL 做了幾番重大的改善，並獲得諸如 Bjarne Stroustrup、Andy Koenig 等人的強力支持。

然後便是滑鐵盧會議。這個名稱對拿破崙而言，標示的是失敗，對 Alexander 和 Lee，以及他們的辛苦作品 STL 而言，標示的卻是巨大的成功。會議投票結果，80 % 贊成，20 % 反對，於是 STL 進入了 C++ 標準化的正式文件與流程之中，並於 1998/09 定案的 C++ 標準規格裡，成爲 C++ 標準程式庫的一大脈系。影響所及，原本就有的 C++ 程式庫如 stream, string 等也都以 template 重新寫過。到處都是 templates！整個 C++ 標準程式庫呈現「春城無處不飛花」的場面。

Dr Dobb's Journal 曾於 1995/03 刊出一篇名爲 *Alexander Stepanov and STL* 的訪談文章，對於 STL 的發展歷史、Alexander 的思路歷程、STL 納入 C++ 標準程式庫的過程，均有詳細的敘述。本處不再贅述。侯捷網站（見附錄 C）上有孟岩先生的譯稿「STL 之父訪談錄」，歡迎觀訪。

1.2 STL 六大組件 功能與運作

STL 提供六大組件，彼此可以組合套用：

1. 容器（containers）：各種資料結構，如 `vector`, `list`, `deque`, `set`, `map`，

用來存放資料，詳見本書 4, 5 兩章。從實作的角度看，STL 容器是一種 class template。就體積而言，這一部份很像冰山在海面下的比率。

2. 演算法（algorithms）：各種常用演算法如 `sort, search, copy, erase…`，詳見第 6 章。從實作的角度看，STL 演算法是一種 function template。
3. 迭代器（iterators）：扮演容器與演算法之間的膠著劑，是所謂的「泛型指標」，詳見第 3 章。共有五種類型，以及其他衍生變化。從實作的角度看，迭代器是一種將 `operator*, operator->, operator++, operator--` 等指標相關操作予以多載化的 class template。所有的 STL 容器都附帶有自己專屬的迭代器 — 是的，因爲只有容器設計者才知道如何走訪自己的元素。原生指標（native pointer）也是一種迭代器。
4. 仿函式（functors）：行爲類似函式，可做爲演算法的某種策略（policy）選擇，詳見第 7 章。從實作的角度看，仿函式是一種改寫（多載化）`operator()` 的 class 或 class template。函式指標可視爲仿函式使用。
5. 配接器（adaptors）：一種用來修飾改變容器（containers）或仿函式（functors）或迭代器（iterators）介面的東西，詳見第 8 章。例如 STL 提供的 `queue` 和 `stack`，雖然看似容器，其實只能算是一種配接器，因爲它們的底部完全借重 `deque`，所有動作都由底層的 `deque` 供應。改變 functor 介面者，稱爲 function adaptor，改變 container 介面者，稱爲 container adaptor，改變 iterator 介面者，稱爲 iterator adaptor。配接器的實作技術很難一言以敝之，必須逐一分析。
6. 配置器（allocators）：空間配置與管理系統，詳見第 2 章。從實作的角度看，配置器是一個實現了動態空間配置、空間管理、空間釋放的 class template。

圖 1-1 顯示 STL 六大組件的交互關係。

由於 STL 已成爲 C++ 標準程式庫的大動脈，因此目前所有的 C++ 編譯器一定支援有一份 STL。在哪裡？就在相應的各個 C++ 表頭檔（headers）。是的，STL 並不以二進位碼（binary code）面貌出現，而是以原始碼面貌供應。按 C++ Standard 的規定，所有標準表頭檔都不應該有副檔名，但或許爲了回溯相容，或許爲了內部組織規劃，某些 STL 版本同時存在具副檔名和無副檔名的兩份檔案，例如 Visual C++ 的 **Dindumware** 版本同時具備 `<vector.h>` 和 `<vector>`；某些 STL 版本只存在具副檔名的表頭檔，例如 C++Builder 的 **RaugeWave** 版本只有

<vector.h>。某些 STL 版本不僅有第一線表頭檔，還有第二線表頭檔，例如 GNU C++ 的 **SGI** 版本不但有第一線的 <vector.h> 和 <vector>，還有第二線的 <stl_vector.h>。

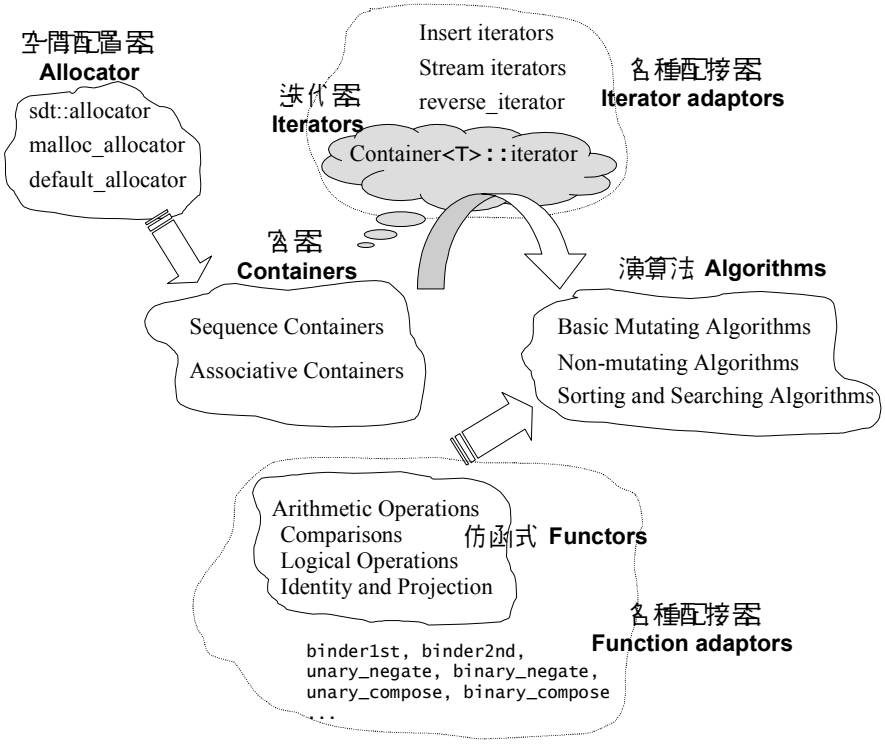


圖 1-1 STL 六大組件的交互關係：Container 透過 Allocator 取得資料儲存空間，Algorithm 透過 Iterator 存取 Container 內容，Functor 可以協助 Algorithm 完成不同的策略變化，Adaptor 可以修飾或套接 Functor。

如果只是應用 STL，請各位務必遵照 C++ 規範，使用無副檔名的表頭檔[4]。如果進入本書層次，要探究 STL 源碼，就得清楚所有這些表頭檔的分佈組織。1.8.2 節將介紹 GNU C++ 所附的 SGI STL 各個表頭檔。

[4] 某些編譯器（例如 C++Builder）會在「前處理器」中動手腳，使無副檔名的表頭檔名實際對應到有副檔名的表頭檔。

1.3 GNU 原碼開放精神

全世界所有的 STL 實品，都源於 Alexander Stepanov 和 Meng Lee 完成的原始版本，這份原始版本屬於 Hewlett-Packard Company（惠普公司）擁有。每一個表頭檔都有一份聲明，允許任何人任意運用、拷貝、修改、傳佈、販賣這些碼，無需付費，唯一的條件是必須將該份聲明置於使用者新開發的檔案內。

這種開放源碼的精神，一般統稱爲 **open source**。本書既然使用這些免費開放的源碼，也有義務對這種精神及其相關歷史與組織，做一個簡介。

開放源碼的觀念源自美國人 Richard Stallman[5]（理察▪史托曼）。他認爲私藏源碼是一種違反人性的罪惡行爲。他認爲如果與他人分享源碼，便可以讓其他人從中學習，並回饋給原始創作者。封鎖源碼雖然可以程度不一地保障智慧所可能衍生的財富，卻阻礙了使用者從中學習和修正錯誤的機會。Stallman 於 1984 離開麻省理工學院，創立自由軟體基金會（Free Software Foundation[6]，簡稱 **FSF**），寫下著名的 GNU 宣言（GNU Manifesto），開始進行名爲 GNU 的開放改革計劃。

GNU[7] 這個名稱是電腦族的幽默展現，代表 **G**NU is **N**ot **U**nix。當時 Unix 是電腦界的主流作業系統，由 AT&T Bell 實驗室的 Ken Thompson 和 Dennis Ritchie 創造。原本這只是一個學術上的練習產品，AT&T 將它分享給許多研究人員。但是當所有研究與分享使這個產品愈變愈美好時，AT&T 開始思考是否應該更加投資並對於從中獲利抱以預期。於是開始要求大學中的相關研究人員簽約，要求他們不得公開或透露 UNIX 源碼，並暫助 Berkeley （柏克萊）大學繼續強化 UNIX，導致後來發展出 **BSD**（Berkeley Software Distribution）版本，以及更後來的 FreeBSD、OpenBSD、NetBSD[8]…。

---

[5] Richard Stallman 的個人網頁見 http://www.stallman.org。

[6] 自由軟體基金會 Free Software Foundation，見 http://www.gnu.org/fsf/fsf.html。

[7] 根據 GNU 的發音，或譯爲「革奴」，意思是從此革去被奴役的命運。音義俱佳。

[8] FreeBSD 見 http://www.freebsd.org，OpenBSD 見 http://www.openbsd.org，NetBSD 見 http://www.netbsd.org。

Stallman 將 AT&T 的這種行爲視爲對思想的控制，以及一種偉大傳統的淪喪。在此之前，電腦界的氛圍是大家無限制地共享各人成果（當然是指最根本的源碼）。Stallman 認爲 AT&T 對大學的暫助，只是一種微薄的施捨，擁有高權力的人才能吃到牛排和龍蝦。於是他進行了他的反奴役計劃，並稱之爲 GNU：**G**NU is **N**ot **U**nix。

GNU 計劃中，早期最著名的軟體包括 **Emacs** 和 **GCC**。前者是 Stallman 開發的一個極具彈性的文字編輯器，允許使用者自行增加各種新功能。後者是個 C/C++ 編譯器，對所有 GNU 軟體提供了平台的一致性與可攜性，是 GNU 計劃的重要基石。GNU 計劃晚近的最著名軟體則是 1991 年由芬蘭人 Linus Torvalds 開發的 **Linux** 作業系統。這些軟體當然都領受了許多使用者的心力回饋，才能更強固穩健。

GNU 以所謂的 **GPL**（General Public License[9]，廣泛開放授權）來保護（或說控制）其成員：使用者可以自由閱讀與修改 GPL 軟體的源碼，但如果使用者要傳佈借助 GPL 軟體而完成的軟體，他們必須也同意 GPL 的規範。這種精神主要是強迫人們分享並回饋他們對 GPL 軟體的改善。

GPL 對於版權（copyright）觀念帶來巨大的挑戰，甚至被稱爲「反版權」（**copyleft**，又一個屬於電腦族群的幽默）。GPL 帶給使用者強大的道德束縛力量，「黏」性甚強，導致某些人產生種種不同的意見，包括可能造成經濟競爭力的薄弱等等。於是其後又衍生出各種不同精義的授權，包括 Library GPL, Lesser GPL, Apache License, Artistic License, BSD License, Mozilla Public License, Netscape Public License。這些授權的共同原則就是，開放源碼。然而各種授權的擁護群眾所滲雜的本位主義，加上精英份子難以妥協的個性，使「開放源碼」陣營中的各個分支，意見紛歧甚至互相對立。其中最甚者爲 GNU GPL 和 BSD License 的擁護者。

1998 年，自由軟體社群企圖藉由創造出一個新名詞 **open source** 來整合各方。他們組成了一個非財團法人的組織，註冊一個標記，並設立網站。open source 的定

[9] GPL 的詳細內容見 http://www.opensource.org/licenses/gpl-license.html。

義共有 9 條[10]，任何軟體只要符合這 9 條，就可稱呼自己爲 open source 軟體。

本書所採用的 GCC 套件是 **Cygnus** C++2.91 for Windows，又稱爲 **EGCS** 1.1。GCC 和 Cygnus、EGCS 之間的關係常常令人混淆。Cygnus 是一家商業公司，包裝並出售自由軟體基金會所建構的軟體工具，並販售各種服務。他們協助晶片廠商調整 GCC，在 GPL 的精神和規範下將 GCC 原始碼的修正公佈於世；他們提供 GCC 運作資訊，並提昇其運作效率，並因此成爲 GCC 技術領域的最佳諮詢對象。Cygnus 公司之於 GCC，地位就像 Red Hat（紅帽）公司之於 Linux。雖然 Cygnus 持續地技術回饋並經濟贊助 GCC，他們並不控制 GCC。GCC 的最終控制權仍然在 GCC 指導委員會（GCC Steering Committee）身上。

當 GCC 的發展進入第二版時，爲了統一事權，GCC 指導委員會開始考慮整合 1997 成立的 EGCS（**E**xperimental/Enhanced **GNU** **C**ompiler **S**ystem）計劃。這個計劃採用比較開放的開發態度，比標準 GCC 涵蓋更多最佳化技術和更多 C++ 語言性質。實驗結果非常成功，因此 GCC 2.95 版反過頭接納了 EGCS 碼。從那個時候開始，GCC 決定採用和 EGCS 一樣的開發方法。自 1999 年起，EGCS 正式成爲唯一的 GCC 官方維護機構。

1.4 HP STL 實作版本

HP 版本是所有 STL 實作品的濫觴。每一個 HP STL 表頭檔都有如下一份聲明，允許任何人免費使用、拷貝、修改、傳佈、販賣這份軟體及其說明文件，唯一需要遵守的是，必須在所有檔案中加上 HP 的版本聲明和運用權限聲明。這種授權並不是 GNU GPL 授權，但屬於 open source 範疇。

```
/*
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
```

[10] 見 http://www.opensource.org/docs/definition_plain.html。

```
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
*/
```

1.5 P. J. Plauger STL 實作版本

P.J. Plauger 版本由 P.J. Plauger 發展，本書後繼章節皆以 **PJ STL** 稱呼此一版本。PJ STL 繼承 HP 版本，所以它的每一個表頭檔都有 HP 的版本聲明，此外還加上 P.J. Plauger 的個人版權聲明：

```
/*
 * Copyright (c) 1995 by P.J. Plauger.  ALL RIGHTS RESERVED.
 * Consult your license regarding permissions and restrictions.
 */
```

這個產品既不屬於 open source 範疇，更不是 GNU GPL。這麼做是合法的，因爲 HP 的版權聲明並非 GPL，並沒有強迫其衍生產品必須開放源碼。

P.J. Plauger 版本被 Visual C++ 採用。當然你可以在 Visual C++ 的 include 子目錄下（例如 `C:\msdev\VC98\Include`）找到所有 STL 表頭檔，只是不能公開它或修改它或甚至販售它。以我個人的閱讀經驗及測試經驗，我對這個版本的可讀性評價極低，主要因爲其中的符號命名極不講究，例如：

```
// TEMPLATE FUNCTION find
template<class _II, class _Ty> inline
   _II find(_II _F, _II _L, const _Ty& _V)
   {for (; _F != _L; ++_F)
        if (*_F == _V)
             break;
   return (_F); }
```

由於 Visual C++ 對 C++ 語言特性的支援不甚理想[11]，導致 P.J. Plauger STL 的表現也受影響。

這項產品目前由 Dinkumware [12]公司提供服務。

[11] 我個人對此有一份經驗整理：http://www.jjhou.com/qa-cpp-primer-27.txt

[12] http://www.dinkumware.com

1.6 Rouge Wave STL 實作版本

RougeWave 版本由 Rouge Wave 公司開發，本書後繼章節皆以 **RW STL** 稱呼此一版本。RW STL 繼承 HP 版本，所以它的每一個表頭檔都有 HP 的版本聲明，此外還加上 Rouge Wave 的公司版權聲明：

```
/*************************************************************************
 * (c) Copyright 1994, 1998 Rogue Wave Software, Inc.
 * ALL RIGHTS RESERVED
 *
 * The software and information contained herein are proprietary to, and
 * comprise valuable trade secrets of, Rogue Wave Software, Inc., which
 * intends to preserve as trade secrets such software and information.
 * This software is furnished pursuant to a written license agreement and
 * may be used, copied, transmitted, and stored only in accordance with
 * the terms of such license and with the inclusion of the above copyright
 * notice.  This software and information or any other copies thereof may
 * not be provided or otherwise made available to any other person.
 *
 * Notwithstanding any other lease or license that may pertain to, or
 * accompany the delivery of, this computer software and information, the
 * rights of the Government regarding its use, reproduction and disclosure
 * are as set forth in Section 52.227-19 of the FARS Computer
 * Software-Restricted Rights clause.
 *
 * Use, duplication, or disclosure by the Government is subject to
 * restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
 * Technical Data and Computer Software clause at DFARS 252.227-7013.
 * Contractor/Manufacturer is Rogue Wave Software, Inc.,
 * P.O. Box 2328, Corvallis, Oregon 97339.
 *
 * This computer software and information is distributed with "restricted
 * rights."  Use, duplication or disclosure is subject to restrictions as
 * set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial
 * Computer Software-Restricted Rights (April 1985)."  If the Clause at
 * 18-52.227-74 "Rights in Data General" is specified in the contract,
 * then the "Alternate III" clause applies.
 *
 *************************************************************************
/
```

這份產品既不屬於 open source 範疇，更不是 GNU GPL。這麼做是合法的，因爲 HP 的版權聲明並非 GPL，並沒有強迫其衍生產品必須開放源碼。

Rouge Wave 版本被 C++Builder 採用。當然你可以在 C++Builder 的 include 子目

錄下（例如 C:\Inprise\CBuilder4\Include）找到所有 STL 表頭檔，只是不能公開它或修改它或甚至販售它。就我個人的閱讀經驗及測試經驗，我要說，這個版本的可讀性還不錯，例如：

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
                    InputIterator last,
                    const T& value)
{
  while (first != last && *first != value)
    ++first;

  return first;
}
```

但是像這個例子（class vector 的內部定義），源碼中夾雜特殊的常數，對閱讀的順暢性是一大考驗：

```
#ifndef _RWSTD_NO_CLASS_PARTIAL_SPEC
    typedef _RW_STD::reverse_iterator<const_iterator>
            const_reverse_iterator;
    typedef _RW_STD::reverse_iterator<iterator>  reverse_iterator;
#else
    typedef _RW_STD::reverse_iterator<const_iterator,
      random_access_iterator_tag, value_type,
      const_reference, const_pointer, difference_type>
      const_reverse_iterator;
    typedef _RW_STD::reverse_iterator<iterator,
      random_access_iterator_tag, value_type,
      reference, pointer, difference_type>
      reverse_iterator;
#endif
```

此外，上述定義方式也不夠清爽（請與稍後的 SGI STL 比較）。

C++Builder 對 C++ 語言特性的支援相當不錯，連帶地給予了 Rouge Wave STL 正面的影響。

1.7 STLport 實作版本

網路上有個 STLport 站點，提供一個以 SGI STL 爲藍本的高度可攜性實作品。本書附錄 D 列有孟岩先生所寫的文章，介紹 STLport 移植到 Visual C++ 和 C++ Builder 的經驗。SGI STL 於下一節介紹，屬於開放源碼組織的一員，所以 STLport 有權

利那麼做。

1.8 SGI STL 實作版本

SGI 版本由 **S**ilicon **G**raphics Computer Systems, **I**nc. 公司發展，繼承 HP 版本。所以它的每一個表頭檔也都有 HP 的版本聲明。此外還加上 SGI 的公司版權聲明。從其聲明可知，它屬於 open source 的一員，但不屬於 GNU GPL（廣泛開放授權）。

```
/*
 * Copyright (c) 1996-1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */
```

SGI 版本被 GCC 採用。你可以在 GCC 的 include 子目錄下（例如 `C:\cygnus\cygwin-b20\include\g++`）找到所有 STL 表頭檔，並獲准自由公開它或修改它或甚至販售它。就我個人的閱讀經驗及測試經驗，我要說，不論是在符號命名或編程風格上，這個版本的可讀性非常高，例如：

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}
```

下面是對應於先前 RaugeWave STL 的源碼實例（class `vector` 的內部定義），也顯得十分乾淨：

```
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  typedef reverse_iterator<const_iterator, value_type, const_reference,
                           difference_type>  const_reverse_iterator;
  typedef reverse_iterator<iterator, value_type, reference, difference_type>
```

```
        reverse_iterator;
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
```

GCC 對 C++ 語言特性的支援相當良好，在主流的 C++ 編譯器中表現耀眼，連帶地給予了 SGI STL 正面的影響。（其實 SGI STL 爲了高度移植性，已經考量了不同編譯器的不同的編譯能力，詳見 1.9.1 節）

SGI STL 也採用了某些 GPL（廣泛性開放授權）檔案，例如 <std\complext.h>, <std\complext.cc>, <std\bastring.h>, <std\bastring.cc>。這些檔案都有這樣的聲明：

```
// This file is part of the GNU ANSI C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
// Free Software Foundation; either version 2, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License
// along with this library; see the file COPYING.  If not, write to the Free
// Software Foundation, 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.

// As a special exception, if you link this library with files
// compiled with a GNU compiler to produce an executable, this does not cause
// the resulting executable to be covered by the GNU General Public License.
// This exception does not however invalidate any other reasons why
// the executable file might be covered by the GNU General Public License.

// Written by Jason Merrill based upon the specification in the 27 May 1994
// C++ working paper, ANSI document X3J16/94-0098.
```

1.8.1 GNU C++ headers 檔案分佈（按字母排序）

我手上的 Cygnus C++ 2.91 for Windows 安裝於磁碟目錄 C:\cygnus。圖 1-2 是這個版本的所有表頭檔，置於 C:\cygnus\cygwin-b20\include\g++，共 128 個檔案。

```
algo.h                  algobase.h          algorithm
alloc.h                 builtinbuf.h        bvector.h
```

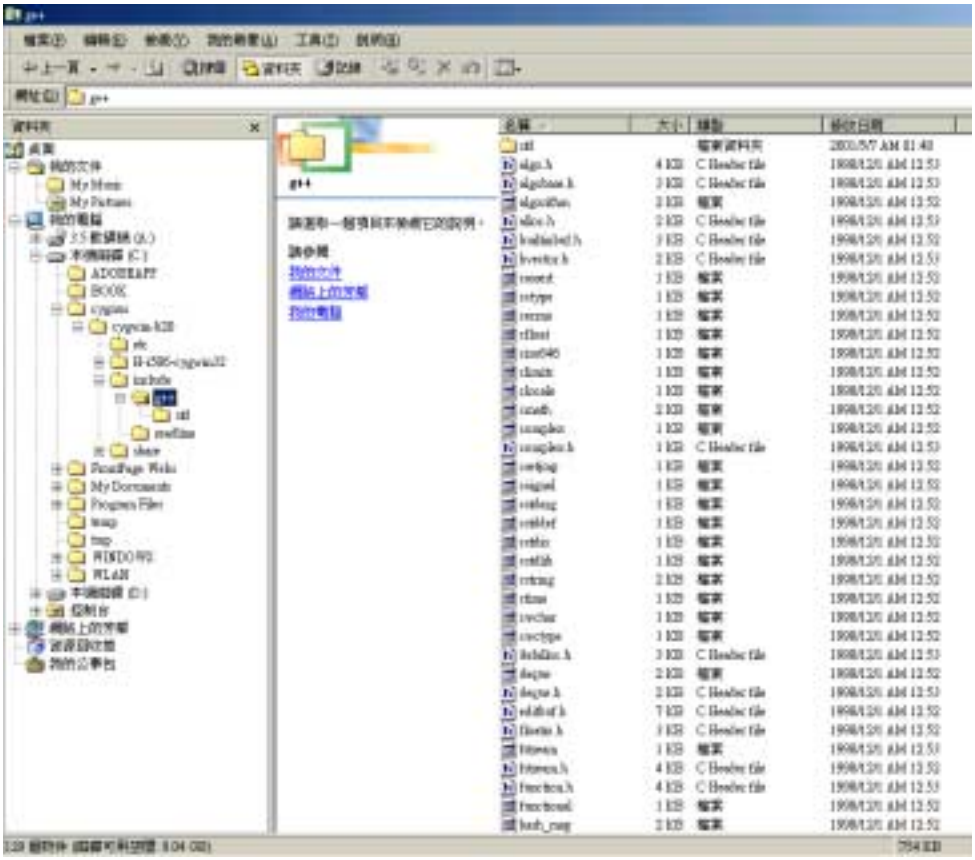
```
cassert              cctype               cerrno
cfloat               ciso646              climits
clocale              cmath                complex
complex.h            csetjmp              csignal
cstdarg              cstddef              cstdio
cstdlib              cstring              ctime
cwchar               cwctype              defalloc.h
deque                deque.h              editbuf.h
floatio.h            fstream              fstream.h
function.h           functional           hashtable.h
hash_map             hash_map.h           hash_set
hash_set.h           heap.h               indstream.h
iolibio.h            iomanip              iomanip.h
iosfwd               iostdio.h            iostream
iostream.h           iostreamP.h          istream.h
iterator             iterator.h           libio.h
libioP.h             list                 list.h
map                  map.h                memory
multimap.h           multiset.h           numeric
ostream.h            pair.h               parsestream.h
pfstream.h           PlotFile.h           procbuf.h
pthread_alloc        pthread_alloc.h      queue
rope                 rope.h               ropeimpl.h
set                  set.h                SFile.h
slist                slist.h              stack
stack.h              [std]                stdexcept
stdiostream.h        stl.h                stl_algo.h
stl_algobase.h       stl_alloc.h          stl_bvector.h
stl_config.h         stl_construct.h      stl_deque.h
stl_function.h       stl_hashtable.h      stl_hash_fun.h
stl_hash_map.h       stl_hash_set.h       stl_heap.h
stl_iterator.h       stl_list.h           stl_map.h
stl_multimap.h       stl_multiset.h       stl_numeric.h
stl_pair.h           stl_queue.h          stl_raw_storage_iter.h
stl_relops.h         stl_rope.h           stl_set.h
stl_slist.h          stl_stack.h          stl_tempbuf.h
stl_tree.h           stl_uninitialized.h  stl_vector.h
stream.h             streambuf.h          strfile.h
string               strstream            strstream.h
tempbuf.h            tree.h               type_traits.h
utility              vector               vector.h
```

子目錄 **[std]** 內有 8 個檔案，70,669 bytes：

```
bastring.cc          bastring.h           complext.cc
complext.h           dcomplex.h           fcomplex.h
ldcomplex.h          straits.h
```

圖 **1-2** Cygnus C++ 2.91 for Windows 的所有表頭檔

下面是以檔案總管觀察 Cygnus C++ 的檔案分佈：



1.8.2 SGI STL 檔案分佈與簡介

上一小節所呈現的眾多表頭檔中，概略可分爲五組：

- C++ 標準規範下的 C 表頭檔（無副檔名），例如 `cstdio`, `cstdlib`, `cstring`…
- C++ 標準程式庫中不屬於 STL 範疇者，例如 stream, string…相關檔案。
- STL 標準表頭檔（無副檔名），例如 `vector`, `deque`, `list`, `map`, `algorithm`, `functional` …
- C++ 標準規格定案前，HP 所規範的 STL 表頭檔，例如 `vector.h`, `deque.h`, `list.h`, `map.h`, `algo.h`, `function.h` …

- SGI STL 內部檔案(STL 真正實作於此),例如 `stl_vector.h, stl_deque.h, stl_list.h, stl_map.h, stl_algo.h, stl_function.h` …

其中前兩組不在本書討論範圍內。後三組表頭檔詳細列表於下。

(1) STL 標準表頭檔(無副檔名)

請注意,各檔案之「本書章節」欄如未列出章節號碼,表示其實際功能由「說明」欄中的 stl_xxx 瓜代,因此實際剖析內容應觀察對應之 stl_xxx 檔案所在章節,見稍後之第三列表。

檔名(按字母排序)	bytes	本書章節	說明
algorithm	1,337		ref. <stl_algorithm.h>
deque	1,350		ref. <stl_deque.h>
functional	762		ref. <stl_function.h>
hash_map	1,330		ref. <stl_hash_map.h>
hash_set	1,330		ref. <stl_hash_set.h>
iterator	1,350		ref. <stl_iterator.h>
list	1,351		ref. <stl_list.h>
map	1,329		ref. <stl_map.h>
memory	2,340	3.2	定義 auto_ptr,並含入 <stl_algobase.h>, <stl_alloc.h>, <stl_construct.h>, <stl_tempbuf.h>, <stl_uninitialized.h>, <stl_raw_storage_iter.h>
numeric	1,398		ref. <stl_numeric.h>
pthread_alloc	9,817	N/A	與 Pthread 相關的 node allocator
queue	1,475		ref. <stl_queue.h>
rope	920		ref. <stl_rope.h>
set	1,329		ref. <stl_set.h>
slist	807		ref. <stl_slist.h>
stack	1,378		ref. <stl_stack.h>
utility	1,301		含入 <stl_relops.h>, <stl_pair.h>
vector	1,379		ref. <stl_vector.h>

(2) C++標準規格定案前,HP 規範的 STL 表頭檔(副檔名 .h)

請注意,各檔案之「本書章節」欄如未列出章節號碼,表示其實際功能由「說明」欄中的 stl_xxx 瓜代,因此實際剖析內容應觀察對應之 stl_xxx 檔案所在章節,見稍後之第三列表。

檔名（按字母排序）	bytes	本書章節	說明
complex.h	141	N/A	複數，含入 <complex>
stl.h	305		含入 STL 標準表頭檔 <algorithm>, <deque>, <functional>, <iterator>, <list>, <map>, <memory>, <numeric>, <set>, <stack>, <utility>, <vector>
type_traits.h	8,888	3.7	SGI 獨特的 type-traits 技法
algo.h	3,182		ref. <stl_algo.h>
algobase.h	2,086		ref. <stl_algobase.h>
alloc.h	1,216		ref. <stl_alloc.h>
bvector.h	1,467		ref. <stl_bvector.h>
defalloc.h	2,360	2.2.1	標準空間配置器 std::allocator，不建議使用。
deque.h	1,373		ref. <stl_deque.h>
function.h	3,327		ref. <stl_function.h>
hash_map.h	1,494		ref. <stl_hash_map.h>
hash_set.h	1,452		ref. <stl_hash_set.h>
hashtable.h	1,559		ref. <stl_hashtable.h>
heap.h	1,427		ref. <stl_heap.h>
iterator.h	2,792		ref. <stl_iterator.h>
list.h	1,373		ref. <stl_list.h>
map.h	1,345		ref. <stl_map.h>
multimap.h	1,370		ref. <stl_multimap.h>
multiset.h	1,370		ref. <stl_multiset.h>
pair.h	1,518		ref. <stl_pair.h>
pthread_alloc.h	867	N/A	#include <pthread_alloc>
rope.h	909		ref. <stl_rope.h>
ropeimpl.h	43,183	N/A	rope 的功能實作
set.h	1,345		ref. <stl_set.h>
slist.h	830		ref. <stl_slist.h>
stack.h	1,466		ref. <stl_stack.h>
tempbuf.h	1,709		ref. <stl_tempbuf.h>
tree.h	1,423		ref. <stl_tree.h>
vector.h	1,378		ref. <stl_vector.h>

(3) SGI STL 內部私用檔案（SGI STL 真正實作於此）

檔名（按字母排序）	bytes	本書章節	說明
stl_algo.h	86,156	6	演算法（數值類除外）
stl_algobase.h	14,105	6.4	基本演算法 swap, min, max, copy, copy_backward, copy_n, fill, fill_n, mismatch, equal, lexicographical_compare
stl_alloc.h	21,333	2	空間配置器 std::alloc。
stl_bvector.h	18,205	N/A	bit_vector（類似標準的 bitset）
stl_config.h	8,057	1.9.1	針對各家編譯器特性定義各種環境常數
stl_construct.h	2,402	2.2.3	建構/解構基本工具

			（construct(), destroy()）
stl_deque.h	41,514	4.4	deque（雙向開口的 queue）
stl_function.h	18,653	7	函式物件（function object） 或稱仿函式（functor）
stl_hash_fun.h	2,752	5.6.7	hash function （雜湊函數，用於 hash-table）
stl_hash_map.h	13,552	5.8	以 hast-table 完成之 map, multimap
stl_hash_set.h	12,990	5.7	以 hast-table 完成之 set, multiset
stl_hashtable.h	26,922	5.6	hast-table（雜湊表）
stl_heap.h	8,212	4.7	heap 演算法：push_heap, pop_heap, make_heap, sort_heap
stl_iterator.h	26,249	3, 8.4, 8.5	迭代器及其相關配接器。並定義迭代器 常用函式 advance(),distance()
stl_list.h	17,678	4.3	list（串列，雙向）
stl_map.h	7,428	5.3	map（映射表）
stl_multimap.h	7,554	5.5	multi-map（多鍵映射表）
stl_multiset.h	6,850	5.4	multi-set（多鍵集合）
stl_numeric.h	6,331	6.3	數值類演算法: accumulate, inner_product, partial_sum, adjacent_difference, power, iota.
stl_pair.h	2,246	9.4	pair（成對組合）
stl_queue.h	4,427	4.6	queue（佇列）, priority_queue（高權先行佇列）
stl_raw_storage_iter.h	2,588	N/A	定義 raw_storage_iterator （一種 OutputIterator）
stl_relops.h	1,772	N/A	定義四個 templatized operators： operator!=, operator>, operator<=, operator>=
stl_rope.h	62,538	N/A	大型（巨量規模）的字串
stl_set.h	6,769	5.2	set（集合）
stl_slist.h	20,524	4.9	single list（單向串列）
stl_stack.h	2,517	4.5	stack（堆疊）
stl_tempbuf.h	3,328	N/A	定義 temporary_buffer class， 應用於 <stl_algo.h>
stl_tree.h	35,451	5.1	Red Black tree（紅黑樹）
stl_uninitialized.h	8,592	2.3	記憶體管理基本工具： uninitialized_copy, uninitialized_fill, uninitialized_fill_n.
stl_vector.h	17,392	4.2	vector（向量）

1.8.3 SGI STL 的編譯器組態設定（configuration）

不同的編譯器對於 C++ 語言性質的支援程度不同。做爲一個希望具備廣泛移植能力的程式庫，SGI STL 準備了一個環境組態檔 <stl_config.h>，其中定義許多

常數，標示某些狀態的成立與否。所有 STL 表頭檔都會直接或間接含入這個組態檔，並以條件式寫法，讓前處理器（pre-processor）根據各個常數決定取捨哪一段程式碼。例如：

```
// in client
#include <vector>

// in <vector>
#include <stl_algobase.h>

// in <stl_algobase.h>
#include <stl_config.h>
...
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION    // 前處理器的條件判斷式
template <class T>
struct __copy_dispatch<T*, T*>
{
  ...
};
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
```

<stl_config.h> 檔案起頭處有一份常數定義說明，然後即針對各家不同的編譯器以及可能的不同版本，給予常數設定。從這裡我們可以一窺各家編譯器對標準 C++ 的支援程度。當然，隨著版本的演進，這些狀態都有可能改變。其中的狀態 (3),(5),(6),(7),(8),(10),(11)，各於 1.9 節中分別在 VC6, CB4, GCC 三家編譯器上測試過。

以下是 GNU C++ 2.91.57 <stl_config.h> 的完整內容：

```
G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_config.h 完整列表
#ifndef __STL_CONFIG_H
# define __STL_CONFIG_H

// 本檔所做的事情：
//  (1)  如果編譯器沒有定義 bool, true, false，就定義它們
//  (2)  如果編譯器的標準程式庫未支援 drand48() 函式，就定義 __STL_NO_DRAND48
//  (3)  如果編譯器無法處理 static members of template classes，就定義
//       __STL_STATIC_TEMPLATE_MEMBER_BUG
//  (4)  如果編譯器未支援關鍵字 typename，就將'typename' 定義爲一個 null macro.
//  (5)  如果編譯器支援 partial specialization of class templates，就定義
//       __STL_CLASS_PARTIAL_SPECIALIZATION.
//  (6)  如果編譯器支援 partial ordering of function templates（亦稱爲
//       partial specialization of function templates），就定義
//       __STL_FUNCTION_TMPL_PARTIAL_ORDER
```

```
// (7)  如果編譯器允許我們在呼叫一個 function template 時可以明白指定其
//      template arguments，就定義 __STL_EXPLICIT_FUNCTION_TMPL_ARGS
// (8)  如果編譯器支援 template members of classes，就定義
//      __STL_MEMBER_TEMPLATES.
// (9)  如果編譯器不支援關鍵字 explicit，就定義'explicit' 爲一個 null macro.
// (10) 如果編譯器無法根據前一個 template parameters 設定下一個 template
//      parameters 的預設值，就定義 __STL_LIMITED_DEFAULT_TEMPLATES
// (11) 如果編譯器針對 non-type template parameters 執行 function template
//      的引數推導（argument deduction）時有問題，就定義
//      __STL_NON_TYPE_TMPL_PARAM_BUG.
// (12) 如果編譯器無法支援迭代器的 operator->，就定義
//      __SGI_STL_NO_ARROW_OPERATOR
// (13) 如果編譯器（在你所選擇的模式中）支援 exceptions，就定義
//      __STL_USE_EXCEPTIONS
// (14) 定義 __STL_USE_NAMESPACES 可使我們自動獲得 using std::list; 之類的敘句
// (15) 如果本程式庫由 SGI 編譯器來編譯，而且使用者並未選擇 pthreads
//      或其他 threads，就定義 __STL_SGI_THREADS.
// (16) 如果本程式庫由一個 WIN32 編譯器編譯，並且在多緒模式下，就定義
//      __STL_WIN32THREADS
// (17) 適當地定義與 namespace 相關的 macros 如 __STD, __STL_BEGIN_NAMESPACE。
// (18) 適當地定義 exception 相關的 macros 如 __STL_TRY, __STL_UNWIND。
// (19) 根據 __STL_ASSERTIONS 是否定義，將 __stl_assert 定義爲一個
//      測試動作或一個 null macro。

#ifdef _PTHREADS
#   define __STL_PTHREADS
#endif

# if defined(__sgi) && !defined(__GNUC__)
// 使用 SGI STL 但卻不是使用 GNU C++
#   if !defined(_BOOL)
#     define __STL_NEED_BOOL
#   endif
#   if !defined(_TYPENAME_IS_KEYWORD)
#     define __STL_NEED_TYPENAME
#   endif
#   ifdef _PARTIAL_SPECIALIZATION_OF_CLASS_TEMPLATES
#     define __STL_CLASS_PARTIAL_SPECIALIZATION
#   endif
#   ifdef _MEMBER_TEMPLATES
#     define __STL_MEMBER_TEMPLATES
#   endif
#   if !defined(_EXPLICIT_IS_KEYWORD)
#     define __STL_NEED_EXPLICIT
#   endif
#   ifdef __EXCEPTIONS
#     define __STL_USE_EXCEPTIONS
#   endif
#   if (_COMPILER_VERSION >= 721) && defined(_NAMESPACES)
```

```
#    define __STL_USE_NAMESPACES
#  endif
#  if !defined(_NOTHREADS) && !defined(__STL_PTHREADS)
#    define __STL_SGI_THREADS
#  endif
# endif

# ifdef __GNUC__
#  include <_G_config.h>
#  if __GNUC__ < 2 || (__GNUC__ == 2 && __GNUC_MINOR__ < 8)
#    define __STL_STATIC_TEMPLATE_MEMBER_BUG
#    define __STL_NEED_TYPENAME
#    define __STL_NEED_EXPLICIT
#  else // 這裡可看出 GNUC 2.8+ 的能力
#    define __STL_CLASS_PARTIAL_SPECIALIZATION
#    define __STL_FUNCTION_TMPL_PARTIAL_ORDER
#    define __STL_EXPLICIT_FUNCTION_TMPL_ARGS
#    define __STL_MEMBER_TEMPLATES
#  endif
   /* glibc pre 2.0 is very buggy. We have to disable thread for it.
      It should be upgraded to glibc 2.0 or later. */
#  if !defined(_NOTHREADS) && __GLIBC__ >= 2 && defined(_G_USING_THUNKS)
#    define __STL_PTHREADS
#  endif
#  ifdef __EXCEPTIONS
#    define __STL_USE_EXCEPTIONS
#  endif
# endif

# if defined(__SUNPRO_CC)
#  define __STL_NEED_BOOL
#  define __STL_NEED_TYPENAME
#  define __STL_NEED_EXPLICIT
#  define __STL_USE_EXCEPTIONS
# endif

# if defined(__COMO__)
#  define __STL_MEMBER_TEMPLATES
#  define __STL_CLASS_PARTIAL_SPECIALIZATION
#  define __STL_USE_EXCEPTIONS
#  define __STL_USE_NAMESPACES
# endif

// 侯捷註：VC6 的版本號碼是 1200
# if defined(_MSC_VER)
#  if _MSC_VER > 1000
#    include <yvals.h>     // 此檔在 MSDEV\VC98\INCLUDE
#  else
#    define __STL_NEED_BOOL
```

```
#   endif
#   define __STL_NO_DRAND48
#   define __STL_NEED_TYPENAME
#   if _MSC_VER < 1100
#     define __STL_NEED_EXPLICIT
#   endif
#   define __STL_NON_TYPE_TMPL_PARAM_BUG
#   define __SGI_STL_NO_ARROW_OPERATOR
#   ifdef _CPPUNWIND
#     define __STL_USE_EXCEPTIONS
#   endif
#   ifdef _MT
#     define __STL_WIN32THREADS
#   endif
# endif

// 侯捷註：Inprise Borland C++builder 也定義有此常數。
// C++Builder 的表現豈有如下所示這般差勁？
# if defined(__BORLANDC__)
#   define __STL_NO_DRAND48
#   define __STL_NEED_TYPENAME
#   define __STL_LIMITED_DEFAULT_TEMPLATES
#   define __SGI_STL_NO_ARROW_OPERATOR
#   define __STL_NON_TYPE_TMPL_PARAM_BUG
#   ifdef _CPPUNWIND
#     define __STL_USE_EXCEPTIONS
#   endif
#   ifdef __MT__
#     define __STL_WIN32THREADS
#   endif
# endif

# if defined(__STL_NEED_BOOL)
    typedef int bool;
#   define true 1
#   define false 0
# endif

# ifdef __STL_NEED_TYPENAME
#   define typename    // 侯捷：難道不該 #define typename class 嗎？
# endif

# ifdef __STL_NEED_EXPLICIT
#   define explicit
# endif

# ifdef __STL_EXPLICIT_FUNCTION_TMPL_ARGS
#   define __STL_NULL_TMPL_ARGS <>
# else
```

```
#   define __STL_NULL_TMPL_ARGS
# endif

# ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
#   define __STL_TEMPLATE_NULL template<>
# else
#   define __STL_TEMPLATE_NULL
# endif

// __STL_NO_NAMESPACES is a hook so that users can disable namespaces
// without having to edit library headers.
# if defined(__STL_USE_NAMESPACES) && !defined(__STL_NO_NAMESPACES)
#   define __STD std
#   define __STL_BEGIN_NAMESPACE namespace std {
#   define __STL_END_NAMESPACE }
#   define  __STL_USE_NAMESPACE_FOR_RELOPS
#   define __STL_BEGIN_RELOPS_NAMESPACE namespace std {
#   define __STL_END_RELOPS_NAMESPACE }
#   define __STD_RELOPS std
# else
#   define __STD
#   define __STL_BEGIN_NAMESPACE
#   define __STL_END_NAMESPACE
#   undef  __STL_USE_NAMESPACE_FOR_RELOPS
#   define __STL_BEGIN_RELOPS_NAMESPACE
#   define __STL_END_RELOPS_NAMESPACE
#   define __STD_RELOPS
# endif

# ifdef __STL_USE_EXCEPTIONS
#   define __STL_TRY try
#   define __STL_CATCH_ALL catch(...)
#   define __STL_RETHROW throw
#   define __STL_NOTHROW throw()
#   define __STL_UNWIND(action) catch(...) { action; throw; }
# else
#   define __STL_TRY
#   define __STL_CATCH_ALL if (false)
#   define __STL_RETHROW
#   define __STL_NOTHROW
#   define __STL_UNWIND(action)
# endif

#ifdef __STL_ASSERTIONS
# include <stdio.h>
# define __stl_assert(expr) \
   if (!(expr)) { fprintf(stderr, "%s:%d STL assertion failure: %s\n", \
               __FILE__, __LINE__, # expr); abort(); }
   // 侯捷註：以上使用 stringizing operator #，詳見《多型與虛擬》x.x 節。
```

```
#else
# define __stl_assert(expr)
#endif

#endif /* __STL_CONFIG_H */

// Local Variables:
// mode:C++
// End:
```

下面這個小程式，用來測試 GCC 的常數設定：

```
// file: 1config.cpp
// test configurations defined in <stl_config.h>
#include <vector>     // which included <stl_algobase.h>,
                      //   and then <stl_config.h>
#include <iostream>
using namespace std;

int main()
{
# if defined(__sgi)
  cout << "__sgi" << endl;           // none!
# endif

# if defined(__GNUC__)
  cout << "__GNUC__" << endl;        // __GNUC__
  cout << __GNUC__ << ' ' << __GNUC_MINOR__ << endl;   // 2 91
  // cout << __GLIBC__ << endl;      // __GLIBC__ undeclared
# endif

// case 2
#ifdef __STL_NO_DRAND48
  cout << "__STL_NO_DRAND48 defined" << endl;
#else
  cout << "__STL_NO_DRAND48 undefined" << endl;
#endif

// case 3
#ifdef __STL_STATIC_TEMPLATE_MEMBER_BUG
  cout << "__STL_STATIC_TEMPLATE_MEMBER_BUG defined" << endl;
#else
  cout << "__STL_STATIC_TEMPLATE_MEMBER_BUG undefined" << endl;
#endif

// case 5
#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  cout << "__STL_CLASS_PARTIAL_SPECIALIZATION defined" << endl;
#else
```

```
  cout << "__STL_CLASS_PARTIAL_SPECIALIZATION undefined" << endl;
#endif

// case 6
...以下寫法類似。詳見檔案 config.cpp（可自侯捷網站下載）。
}
```

執行結果如下，由此可窺見 GCC 對各種 C++ 特性的支援程度：

```
__GNUC__
2 91
__STL_NO_DRAND48 undefined
__STL_STATIC_TEMPLATE_MEMBER_BUG undefined
__STL_CLASS_PARTIAL_SPECIALIZATION defined
__STL_FUNCTION_TMPL_PARTIAL_ORDER defined
__STL_EXPLICIT_FUNCTION_TMPL_ARGS defined
__STL_MEMBER_TEMPLATES defined
__STL_LIMITED_DEFAULT_TEMPLATES undefined
__STL_NON_TYPE_TMPL_PARAM_BUG undefined
__SGI_STL_NO_ARROW_OPERATOR undefined
__STL_USE_EXCEPTIONS defined
__STL_USE_NAMESPACES undefined
__STL_SGI_THREADS undefined
__STL_WIN32THREADS undefined

__STL_NO_NAMESPACES undefined
__STL_NEED_TYPENAME undefined
__STL_NEED_BOOL undefined
__STL_NEED_EXPLICIT undefined
__STL_ASSERTIONS undefined
```

1.9 可能令你困惑的 C++ 語法

1.8 節所列出的幾個狀態常數，用來區分編譯器對 C++ Standard 的支援程度。這幾個狀態，也正是許多程式員對於 C++ 語法最爲困擾之所在。以下我便一一測試 GCC 在這幾個狀態上的表現。有些測試程式直接取材（並剪裁）自 SGI STL 源碼，因此你可以看到最貼近 SGI STL 真面貌的實例。由於這幾個狀態所關係的，都是 template 引數推導（argument deduction）、偏特化（partial specialization）之類的問題，所以測試程式只需完成 classes 或 functions 的介面，便足以測試狀態是否成立。

本節所涵蓋的內容屬於 C++ 語言層次，不在本書範圍之內。因此本節各範例程式只做測試，不做太多說明。每個程式最前面都會有一個註解，告訴你在《C++ Primer

3/e》或《泛型技術》的哪些章節有相關的語法介紹。

1.9.1 stl_config.h 中的各種組態（configurations）

以下所列組態編號與上一節所列的 <stl_config.h> 檔案起頭的註解編號相同。

組態 3：__STL_STATIC_TEMPLATE_MEMBER_BUG

```
// file: 1config3.cpp
// 測試在 class template 中擁有 static data members.
// test __STL_STATIC_TEMPLATE_MEMBER_BUG, defined in <stl_config.h>
// ref. C++ Primer 3/e, p.839, or《泛型技術》2.x
// vc6[o] cb4[x] gcc[o]
// cb4 does not support static data member initialization.

#include <iostream>
using namespace std;

template <typename T>
class testClass {
public:      // 純粹爲了方便測試，使用 public
  static int _data;
};

// 爲 static data members 進行定義（配置記憶體），並設初值。
int testClass<int>::_data  = 1;
int testClass<char>::_data = 2;

int main()
{
  // 以下，CB4 表現不佳，沒有接受初值設定。
  cout << testClass<int>::_data  << endl;  // GCC, VC6:1  CB4:0
  cout << testClass<char>::_data << endl;  // GCC, VC6:2  CB4:0

  testClass<int>  obji1, obji2;
  testClass<char> objc1, objc2;

  cout << obji1._data << endl;  // GCC, VC6:1  CB4:0
  cout << obji2._data << endl;  // GCC, VC6:1  CB4:0
  cout << objc1._data << endl;  // GCC, VC6:2  CB4:0
  cout << objc2._data << endl;  // GCC, VC6:2  CB4:0

  obji1._data = 3;
  objc2._data = 4;

  cout << obji1._data << endl;  // GCC, VC6:3  CB4:3
  cout << obji2._data << endl;  // GCC, VC6:3  CB4:3
```

```
  cout << objc1._data << endl;  // GCC, VC6:4  CB4:4
  cout << objc2._data << endl;  // GCC, VC6:4  CB4:4
}
```

組態 5：__STL_CLASS_PARTIAL_SPECIALIZATION.

```
// file: 1config5.cpp
// 測試 class template partial specialization — 在 class template 的
//   一般化設計之外，特別針對某些 template 參數做特殊設計。
// test __STL_CLASS_PARTIAL_SPECIALIZATION in <stl_config.h>
// ref. C++ Primer 3/e, p.860, or《泛型技術》2.x
// vc6[x] cb4[o] gcc[o]

#include <iostream>
using namespace std;

// 一般化設計
template <class I, class O>
struct testClass
{
  testClass() { cout << "I, O" << endl; }
};

// 特殊化設計
template <class T>
struct testClass<T*, T*>
{
  testClass() { cout << "T*, T*" << endl; }
};

// 特殊化設計
template <class T>
struct testClass<const T*, T*>
{
  testClass() { cout << "const T*, T*" << endl; }
};

int main()
{
  testClass<int, char> obj1;             // I, O
  testClass<int*, int*> obj2;            // T*, T*
  testClass<const int*, int*> obj3;      // const T*, T*
}
```

組態 6：__STL_FUNCTION_TMPL_PARTIAL_ORDER

請注意，雖然 <stl_config.h> 檔案中聲明，這個常數的意義就是 partial

specialization of function templates，但其實兩者並不相同。前者意義如下所示，後者的實際意義請參考《泛型技術》第 2 章或其他 C++ 語法書籍。

```
// file: 1config6.cpp
// test __STL_FUNCTION_TMPL_PARTIAL_ORDER in <stl_config.h>
// vc6[x] cb4[o] gcc[o]

#include <iostream>
using namespace std;

class alloc {
};

template <class T, class Alloc = alloc>
class vector {
public:
  void swap(vector<T, Alloc>&) { cout << "swap()" << endl; }
};

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER  // 只爲說明。非本程式內容。
template <class T, class Alloc>
inline void swap(vector<T, Alloc>& x, vector<T, Alloc>& y) {
  x.swap(y);
}
#endif   // 只爲說明。非本程式內容。

// 以上節錄自 stl_vector.h，灰色部份係源碼中的條件編譯，非本測試程式內容。

int main()
{
  vector<int> x,y;
  swap(x, y);      // swap()
}
```

組態 7：__STL_EXPLICIT_FUNCTION_TMPL_ARGS

整個 SGI STL 內都沒有用到此一常數定義。

狀態 8：__STL_MEMBER_TEMPLATES

```
// file: 1config8.cpp
// 測試 class template 之內可否再有 template (members).
// test __STL_MEMBER_TEMPLATES in <stl_config.h>
// ref. C++ Primer 3/e, p.844, or《泛型技術》2.x
// vc6[o] cb4[o] gcc[o]

#include <iostream>
```

```
using namespace std;

class alloc {
};

template <class T, class Alloc = alloc>
class vector {
public:
  typedef T value_type;
  typedef value_type* iterator;

  template <class I>
  void insert(iterator position, I first, I last) {
    cout << "insert()" << endl;
  }
};

int main()
{
  int ia[5] = {0,1,2,3,4};

  vector<int> x;
  vector<int>::iterator ite;
  x.insert(ite, ia, ia+5);   // insert()
}
```

組態 10：__STL_LIMITED_DEFAULT_TEMPLATES

```
// file: 1config10.cpp
// 測試 template 參數可否根據前一個 template 參數而設定預設值。
// test __STL_LIMITED_DEFAULT_TEMPLATES in <stl_config.h>
// ref. C++ Primer 3/e, p.816, or《泛型技術》2.x
// vc6[o] cb4[o] gcc[o]

#include <iostream>
#include <cstddef>    // for size_t
using namespace std;

class alloc {
};

template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:
  deque() { cout << "deque" << endl; }
};

// 根據前一個參數值 T，設定下一個參數 Sequence 的預設值爲 deque<T>
```

```
template <class T, class Sequence = deque<T> >
class stack {
public:
  stack() { cout << "stack" << endl; }
private:
  Sequence c;
};

int main()
{
  stack<int> x;   // deque
                  // stack
}
```

組態 11：__STL_NON_TYPE_TMPL_PARAM_BUG

```
// file: 1config11.cpp
// 測試 class template 可否擁有 non-type template 參數。
// test __STL_NON_TYPE_TMPL_PARAM_BUG in <stl_config.h>
// ref. C++ Primer 3/e, p.825, or《泛型技術》2.x
// vc6[o] cb4[o] gcc[o]

#include <iostream>
#include <cstddef>    // for size_t
using namespace std;

class alloc {
};

inline size_t __deque_buf_size(size_t n, size_t sz)
{
  return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}

template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator {
  typedef __deque_iterator<T, T&, T*, BufSiz>  iterator;
  typedef __deque_iterator<T, const T&, const T*, BufSiz>
const_iterator;
  static size_t buffer_size() {return __deque_buf_size(BufSiz, sizeof(T)); }
};

template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:                         // Iterators
  typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
};
```

```
int main()
{
  cout << deque<int>::iterator::buffer_size() << endl;       // 128
  cout << deque<int,alloc,64>::iterator::buffer_size() << endl; // 64
}
```

以下組態常數雖不在前列編號之內，卻也是 <stl_config.h> 內的定義，並使用於整個 SGI STL 之中。有認識的必要。

組態：__STL_NULL_TMPL_ARGS（bound friend template friend）

<stl_config.h> 定義 __STL_NULL_TMPL_ARGS 如下：

```
# ifdef __STL_EXPLICIT_FUNCTION_TMPL_ARGS
#   define __STL_NULL_TMPL_ARGS <>
# else
#   define __STL_NULL_TMPL_ARGS
# endif
```

這個組態常數常常出現在類似這樣的場合（class template 的 friend 函式宣告）：

```
// in <stl_stack.h>
template <class T, class Sequence = deque<T> >
class stack {
  friend bool operator== __STL_NULL_TMPL_ARGS (const stack&, const stack&);
  friend bool operator< __STL_NULL_TMPL_ARGS (const stack&, const stack&);
...
};
```

展開後就變成了：

```
template <class T, class Sequence = deque<T> >
class stack {
  friend bool operator== <> (const stack&, const stack&);
  friend bool operator< <> (const stack&, const stack&);
...
};
```

這種奇特的語法是爲了實現所謂的 bound friend templates，也就是說 class template 的某個具現體（instantiation）與其 friend function template 的某個具現體有一對一的關係。下面是個測試程式：

```
// file: 1config-null-template-arguments.cpp
// test __STL_NULL_TMPL_ARGS in <stl_config.h>
// ref. C++ Primer 3/e, p.834: bound friend function template
// or《泛型技術》2.x
// vc6[x] cb4[x] gcc[o]
```

```
#include <iostream>
#include <cstddef>     // for size_t
using namespace std;

class alloc {
};

template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:
  deque() { cout << "deque" << ' '; }
};

// 以下宣告如果不出現，GCC 也可以通過。如果出現，GCC 也可以通過。這一點和
// C++ Primer 3/e p.834 的說法有出入。書上說一定要有這些前置宣告。
/*
template <class T, class Sequence>
class stack;

template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x,
                const stack<T, Sequence>& y);

template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x,
               const stack<T, Sequence>& y);
*/

template <class T, class Sequence = deque<T> >
class stack {
  // 寫成這樣是可以的
  friend bool operator== <T> (const stack<T>&, const stack<T>&);
  friend bool operator<  <T> (const stack<T>&, const stack<T>&);
  // 寫成這樣也是可以的
  friend bool operator== <T> (const stack&, const stack&);
  friend bool operator<  <T> (const stack&, const stack&);
  // 寫成這樣也是可以的
  friend bool operator== <> (const stack&, const stack&);
  friend bool operator<  <> (const stack&, const stack&);
  // 寫成這樣就不可以
  // friend bool operator== (const stack&, const stack&);
  // friend bool operator<  (const stack&, const stack&);

public:
  stack() { cout << "stack" << endl; }
private:
  Sequence c;
};
```

```
template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x,
                const stack<T, Sequence>& y) {
  return cout << "operator==" << '\t';
}

template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x,
               const stack<T, Sequence>& y) {
  return cout << "operator<" << '\t';
}

int main()
{
  stack<int> x;             // deque stack
  stack<int> y;             // deque stack

  cout << (x == y) << endl;   // operator==   1
  cout << (x < y) << endl;    // operator<    1

  stack<char> y1;           // deque stack
//  cout << (x == y1) << endl;  // error: no match for...
//  cout << (x < y1) << endl;   // error: no match for...
}
```

組態：__STL_TEMPLATE_NULL（class template explicit specialization）

<stl_config.h> 定義了一個 __STL_TEMPLATE_NULL 如下：

```
# ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
#   define __STL_TEMPLATE_NULL template<>
# else
#   define __STL_TEMPLATE_NULL
# endif
```

這個組態常數常常出現在類似這樣的場合：

```
// in <type_traits.h>
template <class type> struct __type_traits {   ... };
__STL_TEMPLATE_NULL struct __type_traits<char> {  ... };

// in <stl_hash_fun.h>
template <class Key> struct hash { };
__STL_TEMPLATE_NULL struct hash<char> {  ... };
__STL_TEMPLATE_NULL struct hash<unsigned char> {  ... };
```

展開後就變成了：

```
template <class type> struct __type_traits {   ... };
template<> struct __type_traits<char> {  ... };

template <class Key> struct hash { };
template<> struct hash<char> {  ... };
template<> struct hash<unsigned char> {  ... };
```

這是所謂的 class template explicit specialization。下面這個例子適用於 GCC 和 VC6，允許使用者不指定 `template<>` 就完成 explicit specialization。C++Builder 則是非常嚴格地要求必須完全遵照 C++ 標準規格，也就是必須明白寫出 `template<>`。

```
// file: 1config-template-exp-special.cpp
// 以下測試 class template explicit specialization
// test __STL_TEMPLATE_NULL in <stl_config.h>
// ref. C++ Primer 3/e, p.858, or《泛型技術》2.x
// vc6[o] cb4[x] gcc[o]

#include <iostream>
using namespace std;

// 將 __STL_TEMPLATE_NULL 定義爲 template<>，可以。
// 若定義爲 blank，如下，則只適用於 GCC.
#define __STL_TEMPLATE_NULL   /* blank */

template <class Key> struct hash {
  void operator()() { cout << "hash<T>" << endl; }
};

// explicit specialization
__STL_TEMPLATE_NULL struct hash<char> {
  void operator()() { cout << "hash<char>" << endl; }
};

__STL_TEMPLATE_NULL struct hash<unsigned char> {
  void operator()() { cout << "hash<unsigned char>" << endl; }
};

int main()
{
  hash<long> t1;
  hash<char> t2;
  hash<unsigned char>t3;

  t1();  // hash<T>
  t2();  // hash<char>
  t3();  // hash<unsigned char>
}
```

1.9.2 暫時物件的生命與運用

所謂暫時物件，就是一種無名物件（unnamed objects）。它的出現如果不在程式員的預期之下（例如任何 pass by value 動作都會引發 copy 動作，於是形成一個暫時物件），往往造成效率上的負擔[13]。但有時候刻意製造一些暫時物件，卻又是使程式乾淨清爽的技巧。刻意製造暫時物件的方法是，在型別名稱之後直接加一對小括號，並可指定初值，例如 `Shape(3,5)` 或 `int(8)`，其意義相當於喚起相應的 constructor 且不指定物件名稱。STL 最常將此技巧應用於仿函式（functor）與演算法的搭配上，例如：

```
// file: 1config-temporary-object.cpp
// 本例測試仿函式用於 for_each() 的情形
// vc6[o] cb4[o] gcc[o]
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

template <typename T>
class print
{
public:
  void operator()(const T& elem)   // operator() 多載化。見 1.9.6 節
     {  cout << elem << ' ';  }
};

int main()
{
  int ia[6] = { 0,1,2,3,4,5 };
  vector< int > iv(ia, ia+6);

  // print<int>() 是一個暫時物件，不是一個函式呼叫動作。
  for_each(iv.begin(), iv.end(), print<int>());
}
```

最後一行便是產生「function template 具現體」`print<int>` 的一個暫時物件。這個物件將被傳入 `for_each()` 之中起作用。當 `for_each()` 結束，這個暫時物件也就結束了它的生命。

[13] 請參考 [Meyers96] 條款 19: *Understand the origin of temporary objects.*

1.9.3 靜態常數整數成員在 class 內部直接初始化 in-class static constant integer initialization

如果 class 內含 const static *integral* data member，那麼根據 C++ 標準規格，我們可以在 class 之內直接給予初值。所謂 *integral* 泛指所有整數型別，不單只是指 int。下面是個例子：

```
// file: 1config-inclass-init.cpp
// test in-class initialization of static const integral members
// ref. C++ Primer 3/e, p.??? , or《泛型技術》2.x
// vc6[x] cb4[o] gcc[o]

#include <iostream>
using namespace std;

template <typename T>
class testClass {
public:   // expedient
  static const int _datai = 5;
  static const long _datal = 3L;
  static const char _datac = 'c';
};

int main()
{
  cout << testClass<int>::_datai << endl;  // 5
  cout << testClass<int>::_datal << endl;  // 3
  cout << testClass<int>::_datac << endl;  // c
}
```

1.9.4 increment/decrement/dereference 運算子

increment/dereference 運算子在迭代器的實作上佔有非常重要的地位，因爲任何一個迭代器都必須實作出前進（*increment*, operator++）和取值（*dereference*, operator*）功能，前者還分爲前置式（prefix）和後置式（postfix）兩種，有非常規律的寫法[14]。有些迭代器具備雙向移動功能，那麼就必須再提供 decrement 運算子（也分前置式和後置式兩種）。下面是個範例：

[14] 請參考 [meyers96] item6: distinguish between prefix and postfix forms of increment and decrement operators

```
// file: 1config-operator-overloading.cpp
// vc6[x] cb4[o] gcc[o]
// vc6 的 friend 機制搭配 C++ 標準程式庫，有臭蟲。
#include <iostream>
using namespace std;

class INT
{
friend ostream& operator<<(ostream& os, const INT& i);

public:
  INT(int i) : m_i(i) { };

  // prefix : increment and then fetch
  INT& operator++()
  {
    ++(this->m_i);  // 隨著 class 的不同，此行應該有不同的動作。
    return *this;
  }

  // postfix : fetch and then increment
  const INT operator++(int)
  {
    INT temp = *this;
    ++(*this);
    return temp;
  }

  // prefix : decrement and then fetch
  INT& operator--()
  {
    --(this->m_i);  // 隨著 class 的不同，此行應該有不同的動作。
    return *this;
  }

  // postfix : fetch and then decrement
  const INT operator--(int)
  {
    INT temp = *this;
    --(*this);
    return temp;
  }

  // dereference
  int& operator*() const
  {
    return (int&)m_i;
    // 以上轉換動作告訴編譯器，你確實要將 const int 轉爲 non-const lvalue.
    // 如果沒有這樣明白地轉型，有些編譯器會給你警告，有些更嚴格的編譯器會視爲錯誤
```

```
  }

private:
  int m_i;
};

ostream& operator<<(ostream& os, const INT& i)
{
  os << '[' << i.m_i << ']';
  return os;
}

int main()
{
  INT I(5);
  cout << I++;    // [5]
  cout << ++I;    // [7]
  cout << I--;    // [7]
  cout << --I;    // [5]
  cout << *I;     // 5
}
```

1.9.5 前閉後開區間表示法 [ )

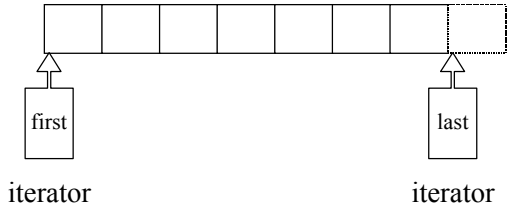
任何一個 STL 演算法，都需要獲得由一對迭代器（泛型指標）所標示的區間，用以表示操作範圍。這一對迭代器所標示的是個所謂的前閉後開區間[15]，以 `[first, last)` 表示。也就是說，整個實際範圍從 `first` 開始，直到 `last-1`。迭代器 `last` 所指的是「最後一個元素的下一位置」。這種 *off by one*（偏移一格）的標示法，帶來許多方便，例如下面兩個 STL 演算法的迴圈設計，就顯得乾淨俐落：

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}

template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first)
    f(*first);
  return f;
}
```

[15] 這是一種半開（half-open）區間，一種後開（open-ended）區間。

前閉後開區間圖示如下（注意，元素之間無需佔用連續記憶體空間）：



1.9.6 function call 運算子 (operator())

很少人注意到，函式呼叫動作（C++ 語法中的左右小括號）也可以被多載化。

許多 STL 演算法都提供兩個版本，一個用於一般狀況（例如排序時以遞增方式排列），一個用於特殊狀況（例如排序時由使用者指定以何種特殊關係進行排列）。像這種情況，需要使用者指定某個條件或某個策略，而條件或策略的背後由一整組動作構成，便需要某種特殊的東西來代表這「一整組動作」。

代表「一整組動作」的，當然是函式。過去 C 語言時代，欲將函式當做參數傳遞，唯有透過函式指標（pointer to function，或稱 function pointer）才能達成，例如：

```
// file: 1qsort.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

int fcmp( const void* elem1, const void* elem2);

void main()
{
  int ia[10] = {32,92,67,58,10,4,25,52,59,54};

  for(int i = 0; i < 10; i++)
    cout << ia[i] << " ";      // 32 92 67 58 10 4 25 52 59 54

  qsort(ia,sizeof(ia)/sizeof(int),sizeof(int), fcmp);

  for(int i = 0; i < 10; i++)
    cout << ia[i] << " ";      // 4 10 25 32 52 54 58 59 67 92
}
```

```
int fcmp( const void* elem1, const void* elem2)
{
const int* i1 = (const int*)elem1;
const int* i2 = (const int*)elem2;

   if( *i1 < *i2)
      return -1;
   else if( *i1 == *i2)
      return 0;
   else if( *i1 > *i2)
      return 1;
}
```

但是函式指標有缺點，最重要的是它無法持有自己的狀態（所謂區域狀態，local states），也無法達到組件技術中的可配接性（adaptability）— 也就是無法再將某些修飾條件加諸於其上而改變其狀態。

爲此，STL 演算法的特殊版本所接受的所謂「條件」或「策略」或「一整組動作」，都以仿函式形式呈現。所謂仿函式（functor）就是使用起來像函式一樣的東西。如果你針對某個 class 進行 operator() 多載化，它就成爲一個仿函式。至於要成爲一個可配接的仿函式，還需要一些額外的努力（詳見第 8 章）。

下面是一個將 operator() 多載化的例子：

```
// file: 1functor.cpp
#include <iostream>
using namespace std;

// 由於將 operator() 多載化了，因此 plus 成了一個仿函式
template <class T>
struct plus {
    T operator()(const T& x, const T& y) const { return x + y; }
};

// 由於將 operator() 多載化了，因此 minus 成了一個仿函式
template <class T>
struct minus {
    T operator()(const T& x, const T& y) const { return x - y; }
};

int main()
{
  // 以下產生仿函式物件。
  plus<int> plusobj;
```

```
    minus<int> minusobj;

    // 以下使用仿函式，就像使用一般函式一樣。
    cout << plusobj(3,5) << endl;           // 8
    cout << minusobj(3,5) << endl;          // -2

    // 以下直接產生仿函式的暫時物件（第一對小括號），並呼叫之（第二對小括號）。
    cout << plus<int>()(43,50) << endl;     // 93
    cout << minus<int>()(43,50) << endl;    // -7
}
```

上述的 `plus<T>` 和 `minus<T>` 已經非常接近 STL 的實作了，唯一差別在於它缺乏「可配接能力」。關於「可配接能力」，將在第 8 章詳述。

2

空間配置器
allocator

以 STL 的運用角度而言，空間配置器是最不需要介紹的東西，它總是隱藏在一切組件（更具體地說是指容器，container）的背後，默默工作默默付出。但若以 STL 的實作角度而言，第一個需要介紹的就是空間配置器，因爲整個 STL 的操作對象（所有的數值）都存放在容器之內，而容器一定需要配置空間以置放資料。不先掌握空間配置器的原理，難免在觀察其他 STL 組件的實作時處處遇到擋路石。

爲什麼不說 allocator 是記憶體配置器而說它是空間配置器呢？因爲，空間不一定是記憶體，空間也可以是磁碟或其他輔助儲存媒體。是的，你可以寫一個 allocator，直接向硬碟取空間[1]。以下介紹的是 SGI STL 提供的配置器，配置的對象，呃，是的，當然是記憶體 ☺。

2.1 空間配置器的標準介面

根據 STL 的規範，以下是 allocator 的必要介面[2]：

```
// 以下各種 type 的設計原由，第三章詳述。
allocator::value_type
allocator::pointer
allocator::const_pointer
allocator::reference
allocator::const_reference
allocator::size_type
allocator::difference_type
```

[1] 請參考 *Disk-Based Container Objects*, by Tom Nelson, *C/C++ Users Journal*, 1998/04

[2] 請參考 [Austern98], 10.3 節。

`allocator::rebind`

一個巢狀的（nested）class template。class `rebind<U>` 擁有唯一成員 `other`，那是一個 typedef，代表 `allocator<U>`。

`allocator::allocator()`

default constructor。

`allocator::allocator(const allocator&)`

copy constructor。

`template <class U>allocator::allocator(const allocator<U>&)`

泛化的 copy constructor。

`allocator::~allocator()`

default constructor。

`pointer allocator::address(reference x) const`

傳回某個物件的位址。算式 `a.address(x)` 等同於 `&x`。

`const_pointer allocator::address(const_reference x) const`

傳回某個 `const` 物件的位址。算式 `a.address(x)` 等同於 `&x`。

`pointer allocator::allocate(size_type n, cosnt void* = 0)`

配置空間，足以儲存 `n` 個 `T` 物件。第二引數是個提示。實作上可能會利用它來增進區域性（locality），或完全忽略之。

`void allocator::deallocate(pointer p, size_type n)`

歸還先前配置的空間。

`size_type allocator::max_size() const`

傳回可成功配置的最大量。

`void allocator::construct(pointer p, const T& x)`

等同於 `new(const void*) p) T(x)`。

`void allocator::destroy(pointer p)`

等同於 `p->~T()`。

2.1.1 設計一個場者的空間配置器，JJ::allocator

根據前述的標準介面，我們可以自行完成一個功能陽春、介面不怎麼齊全的 allocator 如下：

```
// file: 2jjalloc.h
#ifndef _JJALLOC_
#define _JJALLOC_
```

```
#include <new>        // for placement new.
#include <cstddef>    // for ptrdiff_t, size_t
#include <cstdlib>    // for exit()
#include <climits>    // for UINT_MAX
#include <iostream>   // for cerr

namespace JJ
{

template <class T>
inline T* _allocate(ptrdiff_t size, T*) {
   set_new_handler(0);
   T* tmp = (T*)(::operator new((size_t)(size * sizeof(T))));
   if (tmp == 0) {
      cerr << "out of memory" << endl;
      exit(1);
   }
   return tmp;
}

template <class T>
inline void _deallocate(T* buffer) {
   ::operator delete(buffer);
}

template <class T1, class T2>
inline void _construct(T1* p, const T2& value) {
   new(p) T1(value);      // placement new. invoke ctor of T1.
}

template <class T>
inline void _destroy(T* ptr) {
   ptr->~T();
}

template <class T>
class allocator {
public:
   typedef T          value_type;
   typedef T*         pointer;
   typedef const T*   const_pointer;
   typedef T&         reference;
   typedef const T&   const_reference;
   typedef size_t     size_type;
   typedef ptrdiff_t  difference_type;

   // rebind allocator of type U
   template <class U>
```

```
    struct rebind {
        typedef allocator<U> other;
    };

    // hint used for locality. ref.[Austern],p189
    pointer allocate(size_type n, const void* hint=0) {
        return _allocate((difference_type)n, (pointer)0);
    }

    void deallocate(pointer p, size_type n) { _deallocate(p); }

    void construct(pointer p, const T& value) {
        _construct(p, value);
    }

    void destroy(pointer p) { _destroy(p); }

    pointer address(reference x) { return (pointer)&x; }

    const_pointer const_address(const_reference x) {
        return (const_pointer)&x;
    }

    size_type max_size() const {
        return size_type(UINT_MAX/sizeof(T));
    }
};

} // end of namespace JJ

#endif // _JJALLOC_
```

將 JJ::allocator 應用於程式之中，我們發現，它只能有限度地搭配 PJ STL 和 RW STL，例如：

```
// file: 2jjalloc.cpp
// VC6[o], BCB4[o], GCC2.9[x].
#include "jjalloc.h"
#include <vector>
#include <iostream>
using namespace std;

int main()
{
  int ia[5] = {0,1,2,3,4};
  unsigned int i;

  vector<int, JJ::allocator<int> > iv(ia, ia+5);
```

```
  for(i=0; i<iv.size(); i++)
    cout << iv[i] << ' ';
  cout << endl;
}
```

「只能有限度搭配 PJ STL」是因爲，PJ STL 未完全遵循 STL 規格，其所供應的許多容器都需要一個非標準的空間配置器介面 `allocator::_Charalloc()`。「只能有限度搭配 RW STL」則是因爲，RW STL 在許多容器身上運用了緩衝區，情況複雜許多，`JJ::allocator` 無法與之相容。至於完全無法應用於 SGI STL 是因爲，SGI STL 在這個項目上根本上就逸脫了 STL 標準規格，使用一個專屬的、擁有次層配置（sub-allocation）能力的、效率優越的特殊配置器，稍後有詳細介紹。

我想我可以提前先做一點說明。事實上 SGI STL 仍然提供了一個標準的配置器介面，只是把它做了一層隱藏。這個標準介面的配置器名爲 `simple_alloc`，稍後便會提到。

2.2 具備次配置力（sub-allocation）的 SGI 空間配置器

SGI STL 的配置器與眾不同，也與標準規範不同，其名稱是 `alloc` 而非 `allocator`，而且不接受任何引數。換句話說如果你要在程式中明白採用 SGI 配置器，不能採用標準寫法：

```
vector<int, std::allocate<int> > iv;    // in VC or CB
```

必須這麼寫：

```
vector<int, std::alloc> iv;             // in GCC
```

SGI STL allocator 未能符合標準規格，這個事實通常不會對我們帶來困擾，因爲通常我們使用預設的空間配置器，很少需要自行指定配置器名稱，而 SGI STL 的每一個容器都已經指定其預設的空間配置器爲 `alloc`。例如下面的 `vector` 宣告：

```
template <class T, class Alloc = alloc>  // 預設使用 alloc 爲配置器
class vector { ... };
```

2.2.1 SGI 標準的空間配置器，std::allocator

雖然 SGI 也定義有一個符合部份標準、名爲 `allocator` 的配置器，但 SGI 自己

從未用過它，也不建議我們使用。主要原因是效率不彰，只把 C++ 的 ::operator new 和 ::operator delete 做一層薄薄的包裝而已。下面是 SGI 的 std::allocator 全貌：

G++ 2.91.57，cygnus\cygwin-b20\include\g++**defalloc.h 完整列表**

```
// 我們不贊成含入此檔。這是原始的 HP default allocator。提供它只是爲了
// 回溯相容。
//
// DO NOT USE THIS FILE  不要使用這個檔案，除非你手上的容器是以舊式作法
// 完成 — 那就需要一個擁有 HP-style interface 的空間配置器。SGI STL 使用
// 不同的 allocator 介面。SGI-style allocators 不帶有任何與物件型別相關
// 的參數；它們只回應 void* 指標（侯捷註：如果是標準介面，就會回應一個
// 「指向物件型別」的指標，T*）。此檔並不含入於其他任何 SGI STL 表頭檔。

#ifndef DEFALLOC_H
#define DEFALLOC_H

#include <new.h>
#include <stddef.h>
#include <stdlib.h>
#include <limits.h>
#include <iostream.h>
#include <algobase.h>

template <class T>
inline T* allocate(ptrdiff_t size, T*) {
    set_new_handler(0);
    T* tmp = (T*)(::operator new((size_t)(size * sizeof(T))));
    if (tmp == 0) {
        cerr << "out of memory" << endl;
        exit(1);
    }
    return tmp;
}

template <class T>
inline void deallocate(T* buffer) {
    ::operator delete(buffer);
}

template <class T>
class allocator {
public:
    // 以下各種 type 的設計原由，第三章詳述。
    typedef T value_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
```

```
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    pointer allocate(size_type n)
       return ::allocate((difference_type)n, (pointer)0);
    }
    void deallocate(pointer p) { ::deallocate(p); }
    pointer address(reference x) { return (pointer)&x; }
    const_pointer const_address(const_reference x)
       return (const_pointer)&x;
    }
    size_type init_page_size()
       return max(size_type(1), size_type(4096/sizeof(T)));
    }
    size_type max_size() const
       return max(size_type(1), size_type(UINT_MAX/sizeof(T)));
    }
};

// 特化版本（specialization）。注意，爲什麼最前面不需加上 template<>？
// 見 1.9.1 節的組態測試。注意，只適用於 GCC。
class allocator<void>
public:
    typedef void* pointer;
};

#endif
```

2.2.2 SGI 特殊的空間配置器，std::alloc

上一節所說的 allocator 只是基層記憶體配置/解放行爲（也就是 ::operator new 和 ::operator delete）的一層薄薄包裝，並沒有考量到任何效率上的強化。SGI 另有法寶供本身內部使用。

一般而言，我們所習慣的 C++ 記憶體配置動作和釋放動作是這樣：

```
class Foo { ... };
Foo* pf = new Foo;     // 配置記憶體，然後建構物件
delete pf;             // 將物件解構，然後釋放記憶體
```

這其中的 new 算式內含兩階段動作[3]：(1) 呼叫 ::operator new 配置記憶體，(2) 呼叫 Foo::Foo() 建構物件內容。delete 算式也內含兩階段動作：(1) 呼叫

[3] 詳見《多型與虛擬》2/e x..x 節。

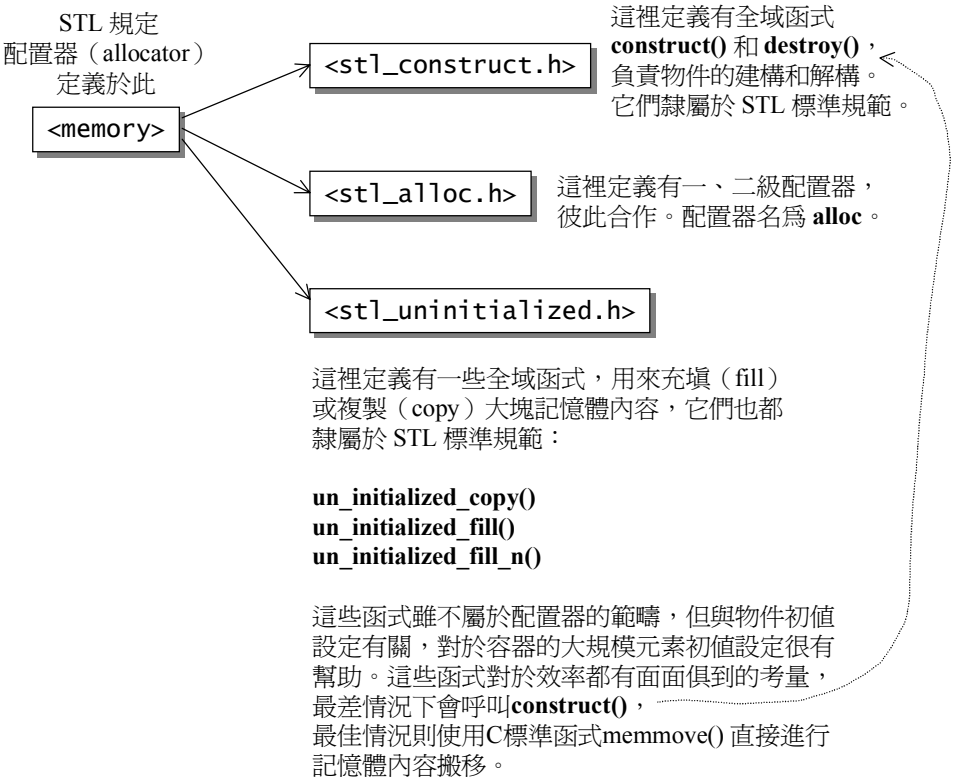
`Foo::~Foo()` 將物件解構，(2) 呼叫 `::operator delete` 釋放記憶體。

爲了精密分工，STL `allocator` 決定將這兩階段動作區分開來。記憶體配置動作由 `alloc:allocate()` 負責，記憶體釋放動作由 `alloc::deallocate()` 負責；物件建構動作由 `::construct()` 負責，物件解構動作由 `::destroy()` 負責。

STL 標準規格告訴我們，配置器定義於 `<memory>` 之中，SGI `<memory>` 內含以下兩個檔案：

```
#include <stl_alloc.h>          // 負責記憶體空間的配置與釋放
#include <stl_construct.h>      // 負責物件內容的建構與解構
```

記憶體空間的配置/釋放與物件內容的建構/解構，分別著落在這兩個檔案身上。其中 `<stl_construct.h>` 定義有兩個基本函式：建構用的 `construct()` 和解構用的 `destroy()`。一頭栽進複雜的記憶體動態配置與釋放之前，讓我們先看清楚這兩個函式如何完成物件的建構和解構。



2.2.3 建構和解構基本工具：construct() 和 destroy()

下面是 <stl_construct.h> 的部份內容（閱讀程式碼的同時，請參考圖 2-1）：

```
#include <new.h>        // 欲使用 placement new，需先含入此檔

template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
  new (p) T1(value);   // placement new; 喚起 ctor T1(value);
}

// 以下是 destroy() 第一版本，接受一個指標。
template <class T>
inline void destroy(T* pointer) {
    pointer->~T();     // 喚起 dtor ~T()
}

// 以下是 destroy() 第二版本，接受兩個迭代器。此函式設法找出元素的數值型別，
// 進而利用 __type_traits<> 求取最適當措施。
template <class ForwardIterator>
inline void destroy(ForwardIterator first, ForwardIterator last) {
  __destroy(first, last, value_type(first));
}

// 判斷元素的數值型別（value type）是否有 trivial destructor
template <class ForwardIterator, class T>
inline void __destroy(ForwardIterator first, ForwardIterator last, T*)
{
  typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
  __destroy_aux(first, last, trivial_destructor());
}

// 如果元素的數值型別（value type）有 non-trivial destructor…
template <class ForwardIterator>
inline void
__destroy_aux(ForwardIterator first, ForwardIterator last, __false_type) {
  for ( ; first < last; ++first)
    destroy(&*first);
}

// 如果元素的數值型別（value type）有 trivial destructor…
template <class ForwardIterator>
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}

// 以下是 destroy() 第二版本針對迭代器爲 char* 和 wchar_t* 的特化版
inline void destroy(char*, char*) {}
inline void destroy(wchar_t*, wchar_t*) {}
```

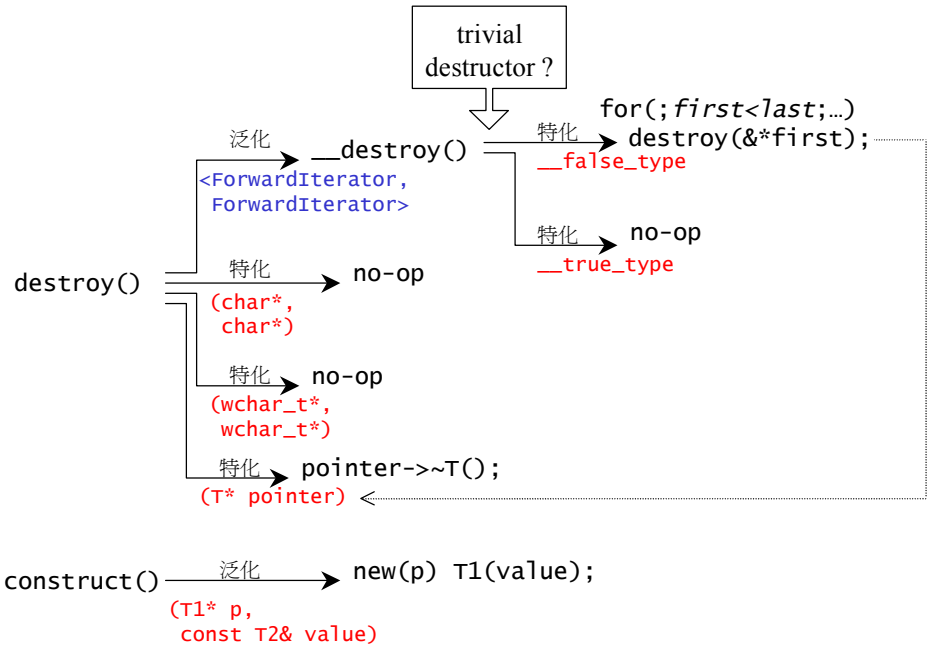


圖 2-1 construct() 和 drestroy() 示意。

這兩個做爲建構、解構之用的函式被設計爲全域函式，符合 STL 的規範[4]。此外 STL 還規定配置器必須擁有名爲 construct() 和 destroy() 的兩個成員函式（見 2.1 節），然而真正在 SGI STL 中大顯身手的那個名爲 std::alloc 的配置器並未遵守此一規則（稍後可見）。

上述 construct() 接受一個指標 p 和一個初值 value，此函式的用途就是將初值設定到指標所指的空間上。C++ 的 placement new 運算子[5] 可用來完成此一任務。

destroy() 有兩個版本，第一版本接受一個指標，準備將該指標所指之物解構掉。

[4] 請參考 [Austern98] 10.4.1 節。

[5] 請參考 [Lippman98] 8.4.5 節。

這很簡單，直接呼叫該物件的解構式即可。第二版本接受 first 和 last 兩個迭代器（所謂迭代器，第三章有詳細介紹），準備將 [first,last) 範圍內的所有物件解構掉。我們不知道這個範圍有多大，萬一很大，而每個物件的解構式都無關痛癢（所謂 *trivial* destructor），那麼一次次呼叫這些無關痛癢的解構式，對效率是一種蘄傷。因此，這裡首先利用 value_type() 獲得迭代器所指物件的型別，再利用 __type_traits<T> 判別該型別的解構式是否無關痛癢。若是（__true_type），什麼也不做就結束；若否（__false_type），這才以迴圈方式巡訪整個範圍，並在迴圈中每經歷一個物件就呼叫第一個版本的 destroy()。

這樣的觀念很好，但 C++ 本身並不直接支援對「指標所指之物」的型別判斷，也不支援對「物件解構式是否爲 *trivial*」的判斷，因此，上述的 value_type() 和 __type_traits<> 該如何實作呢？3.7 節有詳細介紹。

2.2.4 空間的配置與釋放，std::alloc

看完了記憶體配置後的物件建構行爲，和記憶體釋放前的物件解構行爲，現在我們來看看記憶體的配置和釋放。

物件建構前的空間配置，和物件解構後的空間釋放，由 <stl_alloc.h> 負責，SGI 對此的設計哲學如下：

- 向 system heap 要求空間。
- 考慮多緒（multi-threads）狀態。
- 考慮記憶體不足時的應變措施。
- 考慮過多「小型區塊」可能造成的記憶體破碎（fragment）問題。

爲了將問題控制在一定的複雜度內，以下的討論以及所摘錄的源碼，皆排除多緒狀態的處理。

C++ 的記憶體配置基本動作是 ::operator new()，記憶體釋放基本動作是 ::operator delete()。這兩個全域函式相當於 C 的 malloc() 和 free() 函式。是的，正是如此，SGI 正是以 malloc() 和 free() 完成記憶體的配置與釋放。

考量小型區塊所可能造成的記憶體破碎問題，SGI 設計了雙層級配置器，第一級配置器直接使用 malloc() 和 free()，第二級配置器則視情況採用不同的策略：當配置區塊超過 128bytes，視之爲「足夠大」，便呼叫第一級配置器；當配置區塊小於 128bytes，視之爲「過小」，爲了降低額外負擔（overhead，見 2.2.6 節），便採用複雜的 memory pool 整理方式，而不再求助於第一級配置器。整個設計究竟只開放第一級配置器，或是同時開放第二級配置器，取決於 __USE_MALLOC[6] 是否被定義（唔，我們可以輕易測試出來，SGI STL 並未定義 __USE_MALLOC）：

```
# ifdef __USE_MALLOC
...
typedef __malloc_alloc_template<0> malloc_alloc;
typedef malloc_alloc alloc;     // 令 alloc 爲第一級配置器
# else
...
// 令 alloc 爲第二級配置器
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
#endif /* ! __USE_MALLOC */
```

其中 __malloc_alloc_template 就是第一級配置器，__default_alloc_template 就是第二級配置器。稍後分別有詳細介紹。再次提醒你注意，**alloc** 並不接受任何 template 型別參數。

無論 **alloc** 被定義爲第一級或第二級配置器，SGI 還爲它再包裝一個介面如下，使配置器的介面能夠符合 STL 規格：

```
template<class T, class Alloc>
class simple_alloc {
public:
    static T *allocate(size_t n)
              { return 0 == n? 0 : (T*) Alloc::allocate(n * sizeof (T)); }
    static T *allocate(void)
              { return (T*) Alloc::allocate(sizeof (T)); }
    static void deallocate(T *p, size_t n)
              { if (0 != n) Alloc::deallocate(p, n * sizeof (T)); }
    static void deallocate(T *p)
              { Alloc::deallocate(p, sizeof (T)); }
};
```

其內部四個成員函式其實都是單純的轉呼叫，呼叫傳入之配置器（可能是第一級

[6] __USE_MALLOC 這個名稱取得不甚理想，因爲無論如何，最終總是使用 malloc().

也可能是第二級）的成員函式。這個介面使配置器的配置單位從 bytes 轉爲個別元素的大小（sizeof(T)）。SGI STL 容器全都使用這個 simple_alloc 介面，例如：

```
template <class T, class Alloc = alloc>  // 預設使用 alloc 爲配置器
class vector {
protected:
  // 專屬之空間配置器，每次配置一個元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;

  void deallocate() {
    if (...)
        data_allocator::deallocate(start, end_of_storage - start);
  }
...
};
```

一、二級配置器的關係，介面包裝，及實際運用方式，可於圖 2-2 略見端倪。

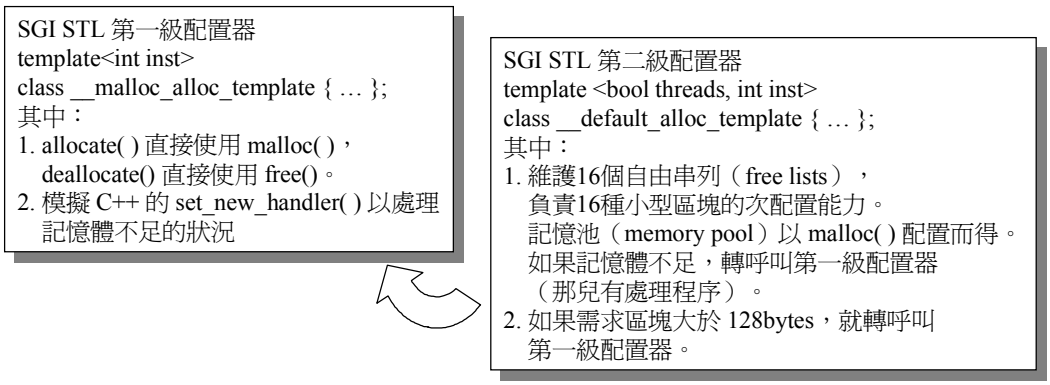


圖 2-2a 第一級配置器與第二級配置器

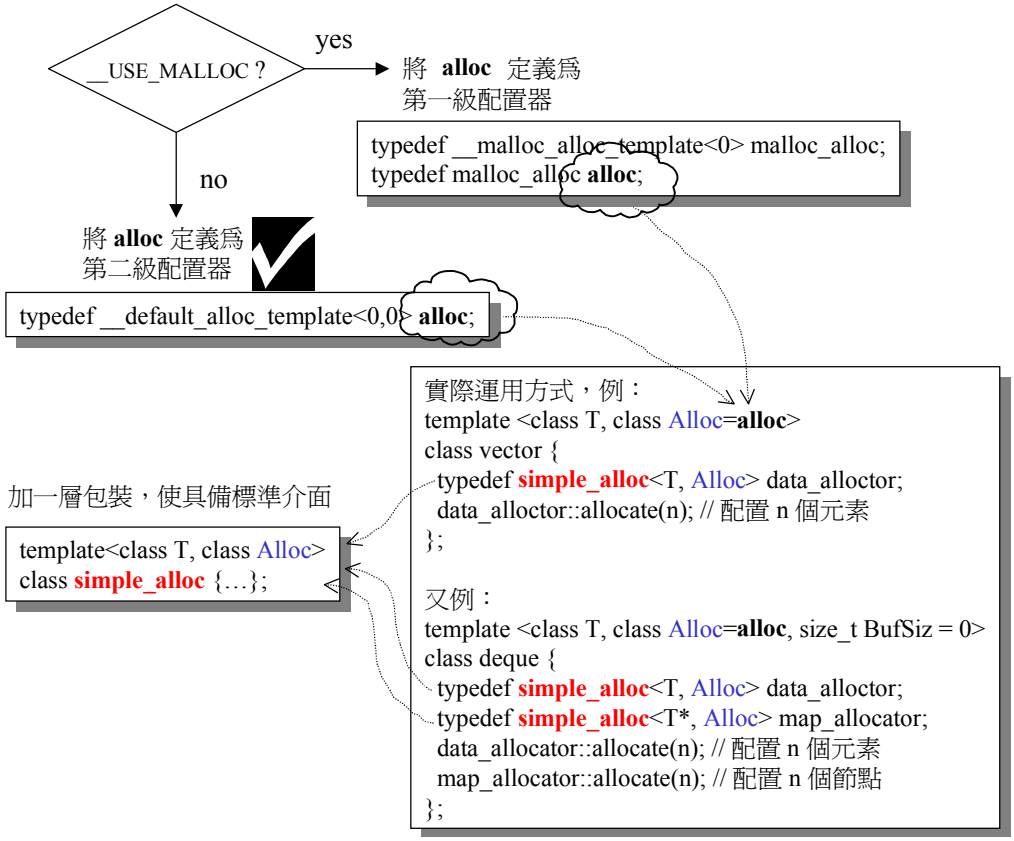


圖 2-2b 第一級配置器與第二級配置器，其包裝介面和運用方式

2.2.5 第一級配置器 __malloc_alloc_template 剖析

首先我們觀察第一級配置器：

```
#if 0
#   include <new>
#   define __THROW_BAD_ALLOC throw bad_alloc
#elif !defined(__THROW_BAD_ALLOC)
#   include <iostream.h>
#   define __THROW_BAD_ALLOC cerr << "out of memory" << endl; exit(1)
#endif

// malloc-based allocator. 通常比稍後介紹的 default alloc 速度慢，
// 一般而言是 thread-safe，並且對於空間的運用比較高效（efficient）。
// 以下是第一級配置器。
```

```
// 注意，無「template 型別參數」。至於「非型別參數」inst，完全沒派上用場。
template <int inst>
class __malloc_alloc_template {

private:
// 以下都是函式指標，所代表的函式將用來處理記憶體不足的情況。
// oom : out of memory.
static void *oom_malloc(size_t);
static void *oom_realloc(void *, size_t);
static void (* __malloc_alloc_oom_handler)();

public:

static void * allocate(size_t n)
{
   void *result = malloc(n);   // 第一級配置器直接使用 malloc()
   // 以下，無法滿足需求時，改用 oom_malloc()
   if (0 == result) result = oom_malloc(n);
   return result;
}

static void deallocate(void *p, size_t /* n */)
{
   free(p); // 第一級配置器直接使用 free()
}

static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
   void * result = realloc(p, new_sz);   // 第一級配置器直接使用 realloc()
   // 以下，無法滿足需求時，改用 oom_realloc()
   if (0 == result) result = oom_realloc(p, new_sz);
   return result;
}

// 以下模擬 C++ 的 set_new_handler(). 換句話說，你可以透過它，
// 指定你自己的 out-of-memory handler
static void (* set_malloc_handler(void (*f)()))()
{
   void (* old)() = __malloc_alloc_oom_handler;
   __malloc_alloc_oom_handler = f;
   return(old);
}
};

// malloc_alloc out-of-memory handling
// 初值爲 0。有待客端設定。
template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;
```

```
template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {          // 不斷嘗試釋放、配置、再釋放、再配置…
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();  // 呼叫處理常式，企圖釋放記憶體。
        result = malloc(n);      // 再次嘗試配置記憶體。
        if (result) return(result);
    }
}

template <int inst>
void * __malloc_alloc_template<inst>::oom_realloc(void *p, size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {          // 不斷嘗試釋放、配置、再釋放、再配置…
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();  // 呼叫處理常式，企圖釋放記憶體。
        result = realloc(p, n);  // 再次嘗試配置記憶體。
        if (result) return(result);
    }
}

// 注意，以下直接將參數 inst 指定爲 0。
typedef __malloc_alloc_template<0> malloc_alloc;
```

第一級配置器以 `malloc()`, `free()`, `realloc()` 等 C 函式執行實際的記憶體配置、釋放、重配置動作，並實作出類似 C++ new-handler[7] 的機制。是的，它不能直接運用 C++ new-handler 機制，因爲它並非使用 `::operator new` 來配置記憶體。

所謂 C++ new handler 機制是，你可以要求系統在記憶體配置需求無法被滿足時，喚起一個你所指定的函式。換句話說一旦 `::operator new` 無法達成任務，在丟出 *std::bad_alloc* 異常狀態之前，會先呼叫由客端指定的處理常式。此處理常式

[7] 詳見 [Meyers98] 條款 7：*Be prepared for out-of-memory conditions.*

通常即被稱爲 new-handler。new-handler 解決記憶體不足的作法有特定的模式，請參考 [Meyers98] 條款 7。

注意，SGI 以 `malloc` 而非 `::operator new` 來配置記憶體（我所能夠想像的一個原因是歷史因素，另一個原因是 C++ 並未提供相應於 `realloc()` 的記憶體配置動作），因此 SGI 不能直接使用 C++ 的 `set_new_handler()`，必須模擬一個類似的 `set_malloc_handler()`。

請注意，SGI 第一級配置器的 `allocate()` 和 `realloc()` 都是在呼叫 `malloc()` 和 `realloc()` 不成功後，改呼叫 `oom_malloc()` 和 `oom_realloc()`。後兩者都有內迴圈，不斷呼叫「記憶體不足處理常式」，期望在某次呼叫之後，獲得足夠的記憶體而圓滿達成任務。但如果「記憶體不足處理常式」並未被客端設定，`oom_malloc()` 和 `oom_realloc()` 便老實不客氣地呼叫 `__THROW_BAD_ALLOC`，丟出 *bad_alloc* 異常訊息，或利用 `exit(1)` 硬生生中止程式。

記住，設計「記憶體不足處理常式」是客端的責任，設定「記憶體不足處理常式」也是客端的責任。再一次提醒你，「記憶體不足處理常式」解決問題的作法有著特定的模式，請參考 [Meyers98] 條款 7。

2.2.6 第二級配置器 __default_alloc_template 剖析

第二級配置器多了一些機制，避免太多小額區塊造成記憶體的破碎。小額區塊帶來的其實不僅是記憶體破碎而已，配置時的額外負擔（overhead）也是一大問題[8]。額外負擔永遠無法避免，畢竟系統要靠這多出來的空間來管理記憶體，如圖 2-3。但是區塊愈小，額外負擔所佔的比例就愈大、愈顯得浪費。

[8] 請參考 [Meyers98] 條款 10：*write operator delete if you write operator new.*

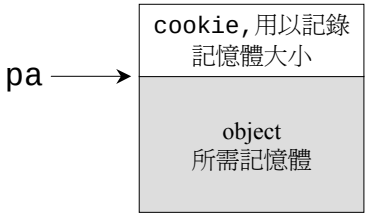


圖 2-3 索求任何一塊記憶體，都得有一些「稅」要繳給系統。

SGI 第二級配置器的作法是，如果區塊夠大，超過 128 bytes，就移交第一級配置器處理。當區塊小於 128 bytes，則以記憶池（memory pool）管理，此法又稱爲次層配置（sub-allocation）：每次配置一大塊記憶體，並維護對應之自由串列（*free-list*）。下次若再有相同大小的記憶體需求，就直接從 *free-lists* 中撥出。如果客端釋還小額區塊，就由配置器回收到 *free-lists* 中 — 是的，別忘了，配置器除了負責配置，也負責回收。爲了方便管理，SGI 第二級配置器會主動將任何小額區塊的記憶體需求量上調至 8 的倍數（例如客端要求 30 bytes，就自動調整爲 32 bytes），並維護 16 個 *free-lists*，各自管理大小分別爲 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 bytes 的小額區塊。*free-lists* 的節點結構如下：

```
union obj {
     union obj * free_list_link;
     char client_data[1];    /* The client sees this. */
};
```

諸君或許會想，爲了維護串列（lists），每個節點需要額外的指標（指向下一個節點），這不又造成另一種額外負擔嗎？你的顧慮是對的，但早已有好的解決辦法。注意，上述 `obj` 所用的是 `union`，由於 `union` 之故，從其第一欄位觀之，`obj` 可被視爲一個指標，指向相同形式的另一個 `obj`。從其第二欄位觀之，`obj` 可被視爲一個指標，指向實際區塊，如圖 2-4。一物二用的結果是，不會爲了維護串列所必須的指標而造成記憶體的另一種浪費（我們正在努力樽節記憶體的開銷呢）。這種技巧在強型（strongly typed）語言如 Java 中行不通，但是在非強型語言如 C++ 中十分普遍[9]。

[9] 請參考 [Lippman98] p840 及 [Noble] p254。

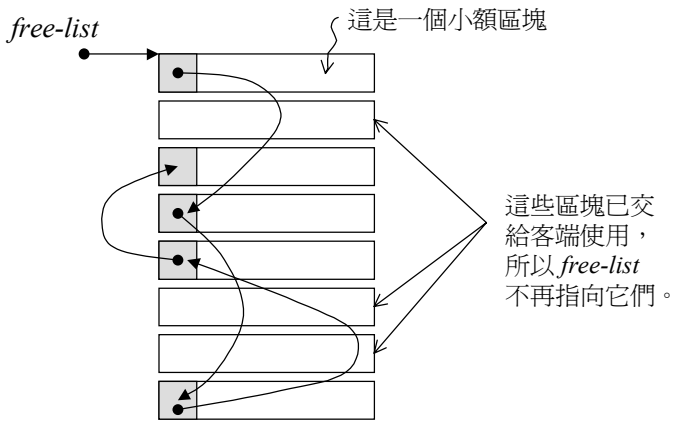


圖 2-4 自由串列（*free-list*）的實作技巧

下面是第二級配置器的部份實作內容：

```
enum {__ALIGN = 8};    // 小型區塊的上調邊界
enum {__MAX_BYTES = 128};  // 小型區塊的上限
enum {__NFREELISTS = __MAX_BYTES/__ALIGN}; // free-lists 個數

// 以下是第二級配置器。
// 注意，無「template 型別參數」，且第二參數完全沒派上用場。
// 第一參數用於多緒環境下。本書不討論多緒環境。
template <bool threads, int inst>
class __default_alloc_template {

private:
  // ROUND_UP() 將 bytes 上調至 8 的倍數。
  static size_t ROUND_UP(size_t bytes) {
       return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
  }
private:
  union obj {          // free-lists 的節點構造
       union obj * free_list_link;
       char client_data[1];    /* The client sees this. */
  };
private:
    // 16 個 free-lists
    static obj * volatile free_list[__NFREELISTS];
    // 以下函式根據區塊大小，決定使用第 n 號 free-list。n 從 1 起算。
    static  size_t FREELIST_INDEX(size_t bytes) {
       return (((bytes) + __ALIGN-1)/__ALIGN - 1);
  }
```

```
  // 傳回一個大小爲 n 的物件，並可能加入大小爲 n 的其他區塊到 free list.
  static void *refill(size_t n);
  // 配置一大塊空間，可容納 nobjs 個大小爲 "size" 的區塊。
  // 如果配置 nobjs 個區塊有所不便，nobjs 可能會降低。
  static char *chunk_alloc(size_t size, int &nobjs);

  // Chunk allocation state.
  static char *start_free; // 記憶池起始位置。只在 chunk_alloc()中變化
  static char *end_free;   // 記憶池結束位置。只在 chunk_alloc()中變化
  static size_t heap_size;

public:
  static void * allocate(size_t n) { /* 詳述於後 */ }
  static void deallocate(void *p, size_t n) { /* 詳述於後 */ }
  static void * reallocate(void *p, size_t old_sz, size_t new_sz);
};

// 以下是 static data member 的定義與初值設定
template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::end_free = 0;

template <bool threads, int inst>
size_t __default_alloc_template<threads, inst>::heap_size = 0;

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj * volatile
__default_alloc_template<threads, inst>::free_list[__NFREELISTS] =
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, };
```

2.2.7 空間配置函式 allocate()

身爲一個配置器，__default_alloc_template 擁有配置器的標準介面函式 allocate()。此函式首先判斷區塊大小，大於 128 bytes 就呼叫第一級配置器，小於 128 bytes 就檢查對應的 *free list*。如果 *free list* 之內有可用的區塊，就直接拿來用，如果沒有可用區塊，就將區塊大小上調至 8 倍數邊界，然後呼叫 refill()，準備爲 *free list* 重新填充空間。refill() 將於稍後介紹。

```
  // n must be > 0
  static void * allocate(size_t n)
  {
    obj * volatile * my_free_list;
```

```
    obj * result;

    // 大於 128 就呼叫第一級配置器
    if (n > (size_t) __MAX_BYTES)
        return(malloc_alloc::allocate(n));
    }
    // 尋找 16 個 free lists 中適當的一個
    my_free_list = free_list + FREELIST_INDEX(n);
    result = *my_free_list;
    if (result == 0) {
        // 沒找到可用的 free list，準備重新填充 free list
        void *r = refill(ROUND_UP(n));     // 下節詳述
        return r;
    }
    // 調整 free list
    *my_free_list = result -> free_list_link;
    return (result);
  };
```

區塊自 *free list* 撥出的動作，如圖 2-5。

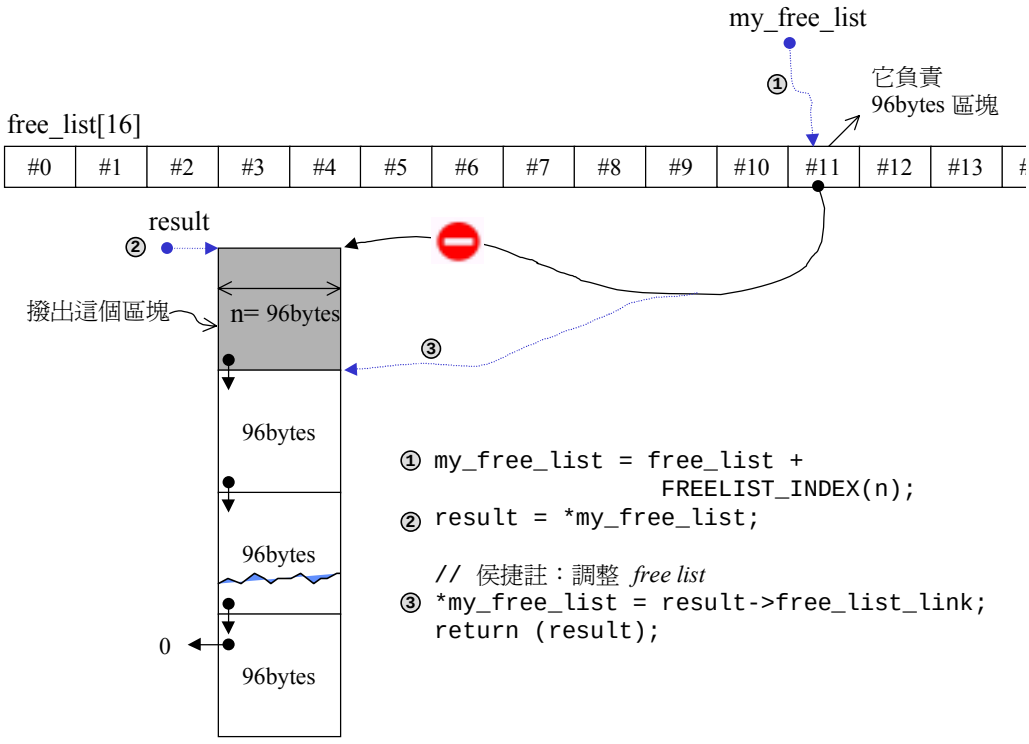


圖 2-5 區塊自 *free list* 撥出。閱讀次序請循圖中編號。

2.2.8 空間釋還函式 deallocate()

身爲一個配置器，__default_alloc_template 擁有配置器標準介面函式 deallocate()。此函式首先判斷區塊大小，大於 128 bytes 就呼叫第一級配置器，小於 128 bytes 就找出對應的 *free list*，將區塊回收。

```
// p 不可以是 0
static void deallocate(void *p, size_t n)
{
  obj *q = (obj *)p;
  obj * volatile * my_free_list;

  // 大於 128 就呼叫第一級配置器
  if (n > (size_t) __MAX_BYTES) {
     malloc_alloc::deallocate(p, n);
     return;
  }
  // 尋找對應的 free list
  my_free_list = free_list + FREELIST_INDEX(n);
  // 調整 free list，回收區塊
  q -> free_list_link = *my_free_list;
  *my_free_list = q;
}
```

區塊回收納入 *free list* 的動作，如圖 2-6。

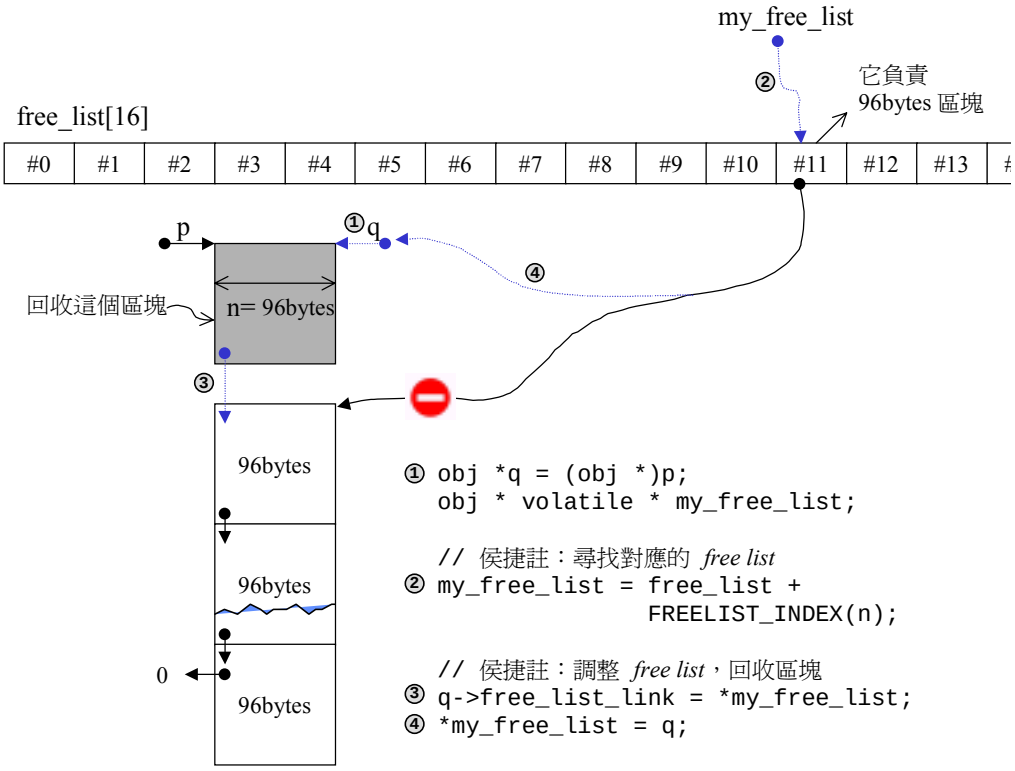


圖 2-6 區塊回收，納入 *free list*。閱讀次序請循圖中編號。

2.2.9 重新填充 *free lists*

回頭討論先前說過的 `allocate()`。當它發現 *free list* 中沒有可用區塊了，就呼叫 `refill()`準備爲 *free list* 重新填充空間。新的空間將取自記憶池（經由 `chunk_alloc()` 完成）。預設取得 20 個新節點（新區塊），但萬一記憶池空間不足，獲得的節點數（區塊數）可能小於 20：

```
// 傳回一個大小爲 n 的物件，並且有時候會爲適當的 free list 增加節點.
// 假設 n 已經適當上調至 8 的倍數。
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    int nobjs = 20;
    // 呼叫 chunk_alloc()，嘗試取得 nobjs 個區塊做爲 free list 的新節點。
    // 注意參數 nobjs 是 pass by reference。
    char * chunk = chunk_alloc(n, nobjs); // 下節詳述
```

```
    obj * volatile * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    // 如果只獲得一個區塊，這個區塊就撥給呼叫者用，free list 無新節點。
    if (1 == nobjs) return(chunk);
    // 否則準備調整 free list，納入新節點。
    my_free_list = free_list + FREELIST_INDEX(n);

    // 以下在 chunk 空間內建立 free list
      result = (obj *)chunk;     // 這一塊準備傳回給客端
      // 以下導引 free list 指向新配置的空間（取自記憶池）
      *my_free_list = next_obj = (obj *)(chunk + n);
      // 以下將 free list 的各節點串接起來。
      for (i = 1; ; i++) {  // 從 1 開始，因爲第 0 個將傳回給客端
        current_obj = next_obj;
        next_obj = (obj *)((char *)next_obj + n);
        if (nobjs - 1 == i) {
            current_obj -> free_list_link = 0;
            break;
        } else {
            current_obj -> free_list_link = next_obj;
        }
      }
    return(result);
}
```

2.2.10 記憶池（memory pool）

從記憶池中取空間給 *free list* 使用，是 `chunk_alloc()` 的工作：

```
// 假設 size 已經適當上調至 8 的倍數。
// 注意參數 nobjs 是 pass by reference。
template <bool threads, int inst>
char*
__default_alloc_template<threads, inst>::
chunk_alloc(size_t size, int& nobjs)
{
    char * result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free; // 記憶池剩餘空間

    if (bytes_left >= total_bytes) {
        // 記憶池剩餘空間完全滿足需求量。
        result = start_free;
        start_free += total_bytes;
```

```
        return(result);
    } else if (bytes_left >= size) {
        // 記憶池剩餘空間不能完全滿足需求量，但足夠供應一個（含）以上的區塊。
        nobjs = bytes_left/size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else {
        // 記憶池剩餘空間連一個區塊的大小都無法提供。
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        // 以下試著讓記憶池中的殘餘零頭還有利用價值。
        if (bytes_left > 0) {
            // 記憶池內還有一些零頭，先配給適當的free list。
            // 首先尋找適當的free list。
            obj * volatile * my_free_list =
                        free_list + FREELIST_INDEX(bytes_left);
            // 調整free list，將記憶池中的殘餘空間編入。
            ((obj *)start_free) -> free_list_link = *my_free_list;
            *my_free_list = (obj *)start_free;
        }

        // 配置 heap 空間，用來挹注記憶池。
        start_free = (char *)malloc(bytes_to_get);
        if (0 == start_free) {
            // heap 空間不足，malloc() 失敗。
            int i;
            obj * volatile * my_free_list, *p;
            // 試著檢視我們手上擁有的東西。這不會造成傷害。我們不打算嘗試配置
            // 較小的區塊，因爲那在多行程（multi-process）機器上容易導致災難
            // 以下搜尋適當的free list，
            // 所謂適當是指「尚有未用區塊，且區塊夠大」之free list。
            for (i = size; i <= __MAX_BYTES; i += __ALIGN) {
                my_free_list = free_list + FREELIST_INDEX(i);
                p = *my_free_list;
                if (0 != p) { // free list內尚有未用區塊。
                    // 調整free list 以釋出未用區塊
                    *my_free_list = p -> free_list_link;
                    start_free = (char *)p;
                    end_free = start_free + i;
                    // 遞迴呼叫自己，爲了修正 nobjs。
                    return(chunk_alloc(size, nobjs));
                    // 注意，任何殘餘零頭終將被編入適當的free-list 中備用。
                }
            }
        end_free = 0; // 如果出現意外（山窮水盡，到處都沒記憶體可用了）
            // 呼叫第一級配置器，看看 out-of-memory 機制能否盡點力
            start_free = (char *)malloc_alloc::allocate(bytes_to_get);
            // 這會導致擲出異常（exception），或記憶體不足的情況獲得改善。
```

```
        }
        heap_size += bytes_to_get;
        end_free = start_free + bytes_to_get;
        // 遞迴呼叫自己，爲了修正 nobjs。
        return(chunk_alloc(size, nobjs));
    }
}
```

上述的 chunk_alloc() 函式以 end_free - start_free 來判斷記憶池的水量。如果水量充足，就直接撥出 20 個區塊傳回給 *free list*。如果水量不足以提供 20 個區塊，但還足夠供應一個以上的區塊，就撥出這不足 20 個區塊的空間出去。這時候其 pass by reference 的 nobjs 參數將被修改爲實際能夠供應的區塊數。如果記憶池連一個區塊空間都無法供應，對客端顯然無法交待，此時便需利用 malloc() 從 heap 中配置記憶體，爲記憶池注入活水源頭以應付需求。新水量的大小爲需求量的兩倍，再加上一個隨著配置次數增加而愈來愈大的附加量。

舉個例子，見圖 2-7，假設程式一開始，客端就呼叫 chunk_alloc(32,20)，於是 malloc() 配置 40 個 32bytes 區塊，其中第 1 個交出，另 19 個交給 free_list[3] 維護，餘 20 個留給記憶池。接下來客端呼叫 chunk_alloc(64,20)，此時 free_list[7] 空空如也，必須向記憶池要求支援。記憶池只夠供應 (32*20)/64=10 個 64bytes 區塊，就把這 10 個區塊傳回，第 1 個交給客端，餘 9 個由 free_list[7] 維護。此時記憶池全空。接下來再呼叫 chunk_alloc(96, 20)，此時 free_list[11] 空空如也，必須向記憶池要求支援，而記憶池此時也是空的，於是以 malloc() 配置 40+n（附加量）個 96bytes 區塊，其中第 1 個交出，另 19 個交給 free_list[11] 維護，餘 20+n（附加量）個區塊留給記憶池……。

萬一山窮水盡，整個 system heap 空間都不夠了（以至無法爲記憶池注入活水源頭），malloc() 行動失敗，chunk_alloc() 就四處尋找有無「尚有未用區塊，且區塊夠大」之 *free lists*。找到的話就挖一塊交出，找不到的話就呼叫第一級配置器。第一級配置器其實也是使用 malloc() 來配置記憶體，但它有 out-of-memory 處理機制（類似 new-handler 機制），或許有機會釋放其他的記憶體拿來此處使用。如果可以，就成功，否則發出 *bad_alloc* 異常。

以上便是整個第二級空間配置器的設計。

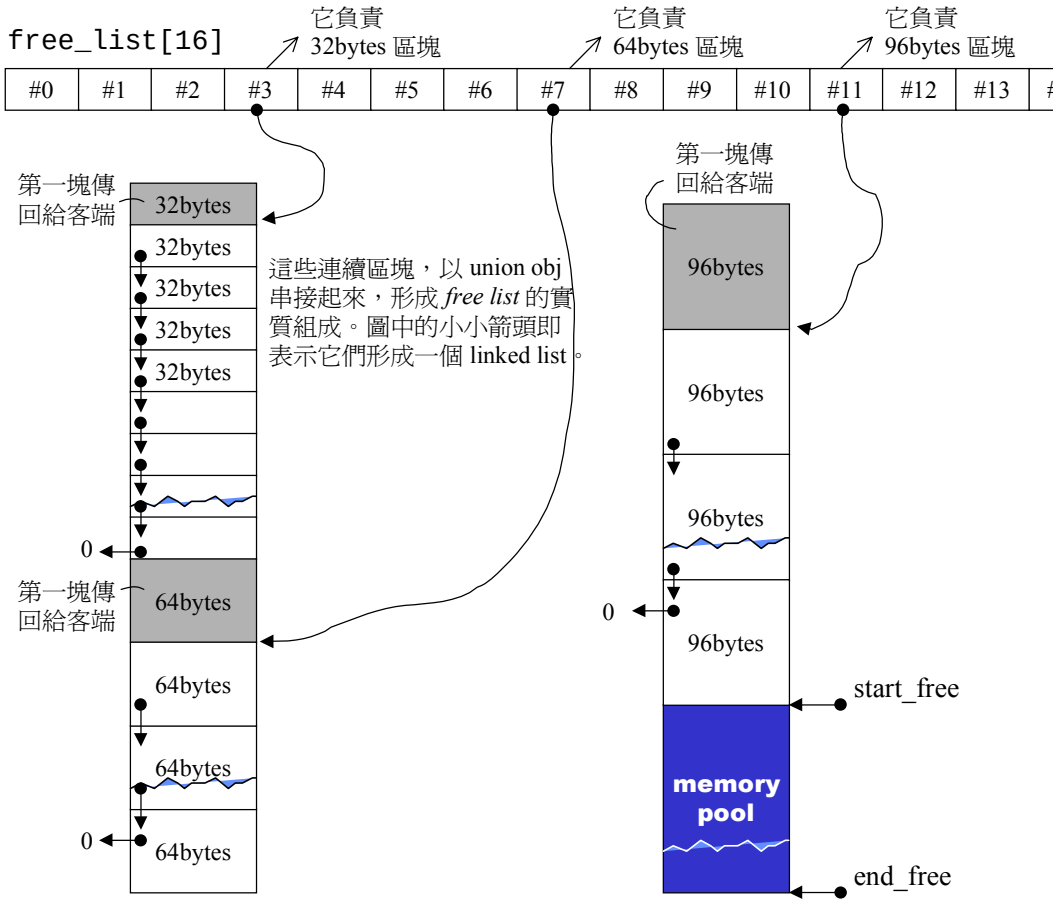


圖 2-7 記憶池（memory pool）實際操練結果

回想一下 2.2.4 節最後提到的那個提供配置器標準介面的 `simple_alloc`：

```
template<class T, class Alloc>
class simple_alloc {
...
};
```

SGI 容器通常以這種方式來使用配置器：

```
template <class T, class Alloc = alloc>  // 預設使用 alloc 爲配置器
class vector {
public:
  typedef T value_type;
...
```

```
protected:
  // 專屬之空間配置器，每次配置一個元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;
...
};
```

其中第二個 template 參數所接受的預設引數 **alloc**，可以是第一級配置器，也可以是第二級配置器。不過，SGI STL 已經把它設爲第二級配置器，見 2.2.4 節及圖 2-2b。

2.3 記憶體基本處理工具

STL 定義有五個全域函式，作用於未初始化空間上。這樣的功能對於容器的實作很有幫助，我們會在第四章容器實作碼中，看到它們的吃重演出。前兩個函式是 2.2.3 節說過，用於建構的 construct() 和用於解構的 destroy()，另三個函式是 uninitialized_copy(),uninitialized_fill(),uninitialized_fill_n()[10]，分別對應於高階函式 copy()、fill()、fill_n() — 這些都是 STL 演算法，將在第六章介紹。如果你要使用本節的三個低階函式，應該含入 <memory>，不過 SGI 把它們實際定義於 <stl_uninitialized>。

2.3.1 uninitialized_copy

```
template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
                   ForwardIterator result);
```

uninitialized_copy() 使我們能夠將記憶體的配置與物件的建構行爲分離開來。如果做爲輸出目的地的 [result, result+(last-first)) 範圍內的每一個迭代器都指向未初始化區域，則 uninitialized_copy() 會使用 copy constructor，爲身爲輸入來源之 [first,last) 範圍內的每一個物件產生一份複製品，放進輸出範圍中。換句話說，針對輸入範圍內的每一個迭代器 i，此函式會呼叫 construct(&*(result+(i-first)),*i)，產生 *i 的複製品，放置於輸

[10] [Austern98] 10.4 節對於這三個低階函式有詳細的介紹。

出範圍的相對位置上。式中的 `construct()` 已於 2.2.3 節討論過。

如果你有需要實作一個容器，`uninitialized_copy()` 這樣的函式會爲你帶來很大的幫助，因爲容器的全範圍建構式（range constructor）通常以兩個步驟完成：

- 配置記憶體區塊，足以包含範圍內的所有元素。
- 使用 `uninitialized_copy()`，在該記憶體區塊上建構元素。

C++ 標準規格書要求 `uninitialized_copy()` 具有 "*commit* or *rollback*" 語意，意思是要不就「建構出所有必要元素」，要不就（當有任何一個 copy constructor 失敗時）「不建構任何東西」。

2.3.2 uninitialized_fill

```
template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                        const T& x);
```

`uninitialized_fill()` 也能夠使我們將記憶體配置與物件的建構行爲分離開來。如果 `[first,last)` 範圍內的每個迭代器都指向未初始化的記憶體，那麼 `uninitialized_fill()` 會在該範圍內產生 `x`（上式第三參數）的複製品。換句話說 `uninitialized_fill()` 會針對操作範圍內的每個迭代器 `i`，呼叫 `construct(&*i, x)`，在 `i` 所指之處產生 `x` 的複製品。式中的 `construct()` 已於 2.2.3 節討論過。

和 `uninitialized_copy()` 一樣，`uninitialized_fill()` 必須具備 "*commit* or *rollback*" 語意，換句話說它要不就產生出所有必要元素，要不就不產生任何元素。如果有任何一個 copy constructor 丟出異常（exception），`uninitialized_fill()` 必須能夠將已產生之所有元素解構掉。

2.3.3 uninitialized_fill_n

```
template <class ForwardIterator, class Size, class T>
ForwardIterator
uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

`uninitialized_fill_n()` 能夠使我們將記憶體配置與物件建構行爲分離開來。它會爲指定範圍內的所有元素設定相同的初值。

如果 `[first, first+n)` 範圍內的每一個迭代器都指向未初始化的記憶體，那麼 `uninitialized_fill_n()` 會呼叫 copy constructor，在該範圍內產生 `x`（上式第三參數）的複製品。也就是說面對 `[first,first+n)` 範圍內的每個迭代器 `i`，`uninitialized_fill_n()` 會呼叫 `construct(&*i, x)`，在對應位置處產生 `x` 的複製品。式中的 `construct()` 已於 2.2.3 節討論過。

`uninitialized_fill_n()` 也具有 "*commit* or *rollback*" 語意：要不就產生所有必要的元素，否則就不產生任何元素。如果任何一個 copy constructor 丟出異常（exception），`uninitialized_fill_n()` 必須解構已產生的所有元素。

以下分別介紹這三個函式的實作法。其中所呈現的 iterators（迭代器）、`value_type()`、`__type_traits`、`__true_type`、`__false_type`、`is_POD_type` 等實作技術，都將於第三章介紹。

(1) uninitialized_fill_n

首先是 `uninitialized_fill_n()` 的源碼。本函式接受三個參數：

1. 迭代器 `first` 指向欲初始化空間的起始處
2. `n` 表示欲初始化空間的大小
3. `x` 表示初值

```
template <class ForwardIterator, class Size, class T>
inline ForwardIterator uninitialized_fill_n(ForwardIterator first,
                                            Size n, const T& x) {
  return __uninitialized_fill_n(first, n, x, value_type(first));
  // 以上，利用 value_type() 取出 first 的 value type.
}
```

這個函式的進行邏輯是，首先萃取出迭代器 `first` 的 *value type*（詳見第三章），然後判斷該型別是否爲 POD 型別：

```
template <class ForwardIterator, class Size, class T, class T1>
inline ForwardIterator __uninitialized_fill_n(ForwardIterator first,
                                              Size n, const T& x, T1*)
```

```
{
  // 以下 __type_traits<> 技法，詳見 3.7 節
  typedef typename __type_traits<T1>::is_POD_type is_POD;
  return __uninitialized_fill_n_aux(first, n, x, is_POD());
}
```

POD 意指 Plain Old Data，也就是純量型別（scalar types）或傳統的 C struct 型別。POD 型別必然擁有 *trivial* ctor/dtor/copy/assignment 函式，因此，我們可以對 POD 型別採取最有效率的初值填寫手法，而對 non-POD 型別採取最保險安全的作法：

```
// 如果 copy construction 等同於 assignment, 而且
//  destructor 是 trivial，以下就有效。
// 如果是 POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class ForwardIterator, class Size, class T>
inline ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __true_type) {
  return fill_n(first, n, x);   // 交由高階函式執行。見 6.4.2 節。
}

// 如果不是 POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {
  ForwardIterator cur = first;
    // 爲求閱讀順暢，以下將原本該有的異常處理（exception handling）省略。
    for ( ; n > 0; --n, ++cur)
       construct(&*cur, x);     // 見 2.2.3 節
    return cur;
}
```

(2) uninitialized_copy

下面列出 `uninitialized_copy()` 的源碼。本函式接受三個參數：

- 迭代器 `first` 指向輸入端的起始位置
- 迭代器 `last` 指向輸入端的結束位置（前閉後開區間）
- 迭代器 `result` 指向輸出端（欲初始化空間）的起始處

```
template <class InputIterator, class ForwardIterator>
inline ForwardIterator
  uninitialized_copy(InputIterator first, InputIterator last,
```

```
                    ForwardIterator result) {
  return __uninitialized_copy(first, last, result, value_type(result));
  // 以上，利用 value_type() 取出 first 的 value type.
}
```

這個函式的進行邏輯是，首先萃取出迭代器 result 的 *value type*（詳見第三章），然後判斷該型別是否爲 POD 型別：

```
template <class InputIterator, class ForwardIterator, class T>
inline ForwardIterator
__uninitialized_copy(InputIterator first, InputIterator last,
                     ForwardIterator result, T*) {
  typedef typename __type_traits<T>::is_POD_type is_POD;
  return __uninitialized_copy_aux(first, last, result, is_POD());
  // 以上，企圖利用 is_POD() 所獲得的結果，讓編譯器做引數推導。
}
```

POD 意指 Plain Old Data，也就是純量型別（scalar types）或傳統的 C struct 型別。POD 型別必然擁有 *trivial* ctor/dtor/copy/assignment 函式，因此，我們可以對 POD 型別採取最有效率的複製手法，而對 non-POD 型別採取最保險安全的作法：

```
// 如果 copy construction 等同於 assignment, 而且
//  destructor 是 trivial，以下就有效。
// 如果是 POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class InputIterator, class ForwardIterator>
inline ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                         ForwardIterator result,
                         __true_type) {
  return copy(first, last, result);  // 呼叫 STL 演算法 copy()
}

//如果是 non-POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class InputIterator, class ForwardIterator>
ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                         ForwardIterator result,
                         __false_type) {
  ForwardIterator cur = result;
    // 爲求閱讀順暢，以下將原本該有的異常處理（exception handling）省略。
    for ( ; first != last; ++first, ++cur)
      construct(&*cur, *first);  // 必須一個一個元素地建構，無法批量進行
    return cur;
  }
}
```

針對 char* 和 wchar_t* 兩種型別，可以最具效率的作法 memmove（直接搬移記憶體內容）來執行複製行爲。因此 SGI 得以爲這兩種型別設計一份特化版本。

```
// 以下是針對 const char* 的特化版本
inline char* uninitialized_copy(const char* first, const char* last,
                                char* result) {
  memmove(result, first, last - first);
  return result + (last - first);
}

// 以下是針對 const wchar_t* 的特化版本
inline wchar_t* uninitialized_copy(const wchar_t* first, const wchar_t* last,
                                   wchar_t* result) {
  memmove(result, first, sizeof(wchar_t) * (last - first));
  return result + (last - first);
}
```

(3) uninitialized_fill

下面列出 uninitialized_fill() 的源碼。本函式接受三個參數：

- 迭代器 first 指向輸出端（欲初始化空間）的起始處
- 迭代器 last 指向輸出端（欲初始化空間）的結束處（前閉後開區間）
- x 表示初值

```
template <class ForwardIterator, class T>
inline void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                               const T& x) {
  __uninitialized_fill(first, last, x, value_type(first));
}
```

這個函式的進行邏輯是，首先萃取出迭代器 first 的 *value type*（詳見第三章），然後判斷該型別是否爲 POD 型別：

```
template <class ForwardIterator, class T, class T1>
inline void __uninitialized_fill(ForwardIterator first, ForwardIterator last,
                                 const T& x, T1*) {
  typedef typename __type_traits<T1>::is_POD_type is_POD;
  __uninitialized_fill_aux(first, last, x, is_POD());
}
```

POD 意指 Plain Old Data，也就是純量型別（scalar types）或傳統的 C struct 型別。POD 型別必然擁有 *trivial* ctor/dtor/copy/assignment 函式，因此，我們可以對 POD 型別採取最有效率的初值填寫手法，而對 non-POD 型別採取最保險安全的

作法：

```
// 如果 copy construction 等同於 assignment, 而且
//  destructor 是 trivial，以下就有效。
// 如果是 POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class ForwardIterator, class T>
inline void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator last,
                         const T& x, __true_type)
{
  fill(first, last, x);    // 呼叫 STL 演算法 fill()
}

//如果是 non-POD 型別，執行流程就會轉進到以下函式。這是藉由 function template
// 的引數推導機制而得。
template <class ForwardIterator, class T>
void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator last,
                         const T& x, __false_type)
{
  ForwardIterator cur = first;
    // 爲求閱讀順暢，以下將原本該有的異常處理（exception handling）省略。
    for ( ; cur != last; ++cur)
      construct(&*cur, x);  // 必須一個一個元素地建構，無法批量進行
  }
}
```

圖 2-8 將本節三個函式對效率的特殊考量，以圖形顯示。

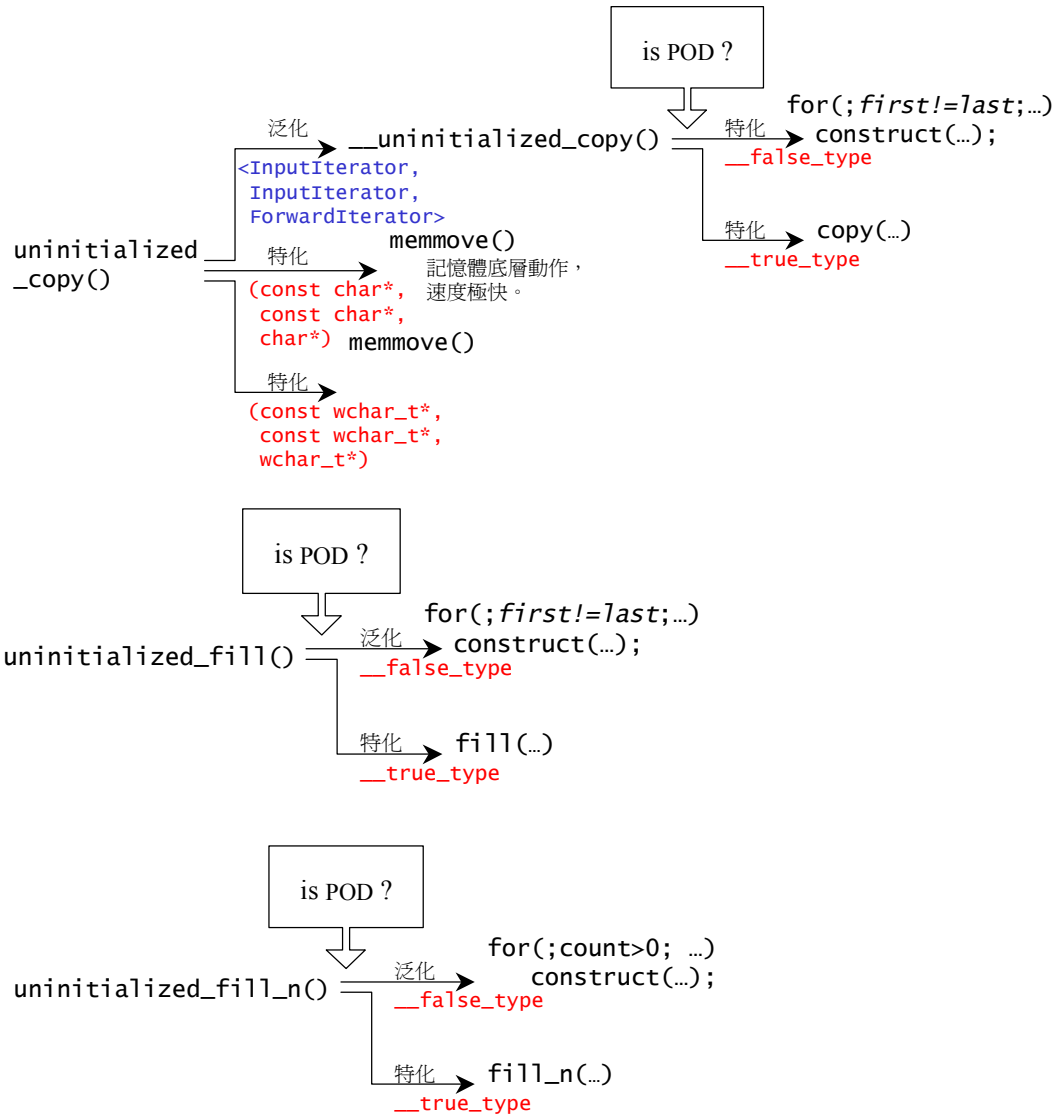


圖 2-8 三個記憶體基本函式的泛型版本與特化版本。