



STL 源碼剖析

侯捷

基峰

無限延伸你的視野

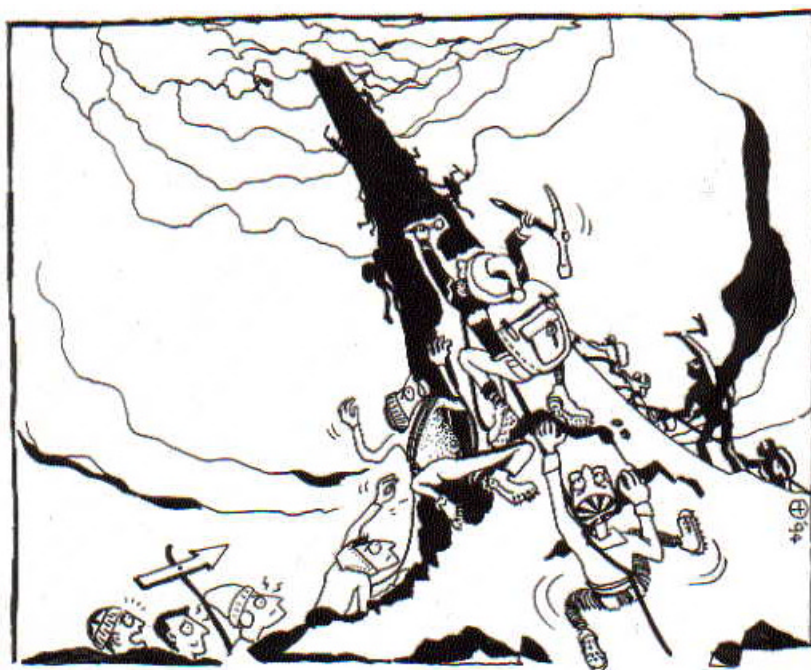
The Annotated STL Source

庖丁解牛 恢恢乎游刃有餘

STL源碼剖析

The Annotated STL Source (using SGI STL)

侯捷



基峰腦圖書資料股份有限公司

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，我開始為計劃中的《泛型思維》一書陸續準備並熱身。為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。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，價值不菲；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《泛型思維》相呼應。於是我便一頭栽進去了。

我選擇 SGI STL 做為剖析對象。這份實作版本的可讀性極佳，運用極廣，被選為 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。不僅最為人廣泛運用的各種資料結構（data structures）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，那些蛰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）。從技術研究與本質提昇的角度看，深究細節可以让你徹底掌握一切；不論是爲了重溫資料結構和演算法，或是想要扮演輪子角色，或是想要進一步擴張別人的輪子，都可因此獲得深厚紮實的基礎。

天下大事，必作於細！

但是別忘了，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。然而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。

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，做為呈堂作業，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 — 我的確認為 99.99 % 的程式員所寫的程式，在 SGI STL 面前都是三流水準 ☺。

侯捷 2001/05/30 新竹 ■ 臺灣

<http://www.jjhou.com> (繁體)

<http://jjhou.csdn.net> (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。為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。
- 《泛型思維》，內容涵括：語言層次（C++ templates 語法、Java generic 語法、C++ 運算子重載），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，泛型思考。
- 《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解說。

目 錄

庖丁解牛（侯捷自序）	i
目錄	v
前言	xvii
本書定位	xvii
合適的讀者	xviii
最佳閱讀方式	xviii
我所選擇的剖析對象	xix
各章主題	xx
編譯工具	xx
中英術語的運用風格	xxi
英文術語採用原則	xxii
版面字形風格	xxiii
源碼形式與下載	xxiv
線上服務	xxvi
推薦讀物	xxvi
第 1 章 STL 概論與版本簡介	001
1.1 STL 概論	001
1.1.1 STL 的歷史	003
1.1.2 STL 與 C++ 標準程式庫	003

1.2 STL 六大組件 — 功能與運用	004
1.3 GNU 源碼開放精神	007
1.4 HP STL 實作版本	009
1.5 P.J. Plauger STL 實作版本	010
1.6 Rouge Wave STL 實作版本	011
1.7 STLport 實作版本	012
1.8 SGI STL 實作版本 總覽	013
1.8.1 GNU C++ header 檔案分佈	014
1.8.2 SGI STL 檔案分佈與簡介	016
STL 標準表頭檔（無副檔名）	017
C++ 標準規格定案前，HP 規範的 STL 表頭檔（副檔名 .h）	017
SGI STL 內部檔案（SGI STL 真正實作於此）	018
1.8.3 SGI STL 的組態設定（configuration）	019
1.9 可能令你困惑的 C++ 語法	026
1.9.1 <code>stl_config.h</code> 中的各種組態	027
組態 3：static template member	027
組態 5：class template partial specialization	028
組態 6：function template partial order	028
組態 7：explicit function template arguments	029
組態 8：member templates	029
組態 10：default template argument depend on previous template parameters	030
組態 11：non-type template parameters	031
組態：bound friend template function	032
組態：class template explicit specialization	034
1.9.2 暫時物件的產生與運用	036
1.9.3 靜態常數整數成員在 class 內部直接初始化 in-class static const <i>integral</i> data member initialization	037

1.9.4 increment/decrement/dereference 運算子	037
1.9.5 「前閉後開」區間表示法 [)	039
1.9.6 function call 運算子 (operator())	040
第 2 章 空間配置器 (allocator)	043
2.1 空間配置器的標準介面	043
2.1.1 設計一個陽春的空間配置器，JJ::allocator	044
2.2 具備次配置力 (sub-allocation) 的 SGI 空間配置器	047
2.2.1 SGI 標準的空間配置器，std::allocator	047
2.2.2 SGI 特殊的空間配置器，std::alloc	049
2.2.3 建構和解構基本工具：construct() 和 destroy()	051
2.2.4 空間的配置與釋放，std::alloc	053
2.2.5 第一級配置器 __malloc_alloc_template 剖析	056
2.2.6 第二級配置器 __default_alloc_template 剖析	059
2.2.7 空間配置函式 allocate()	062
2.2.8 空間釋放函式 deallocate()	064
2.2.9 重新充填 <i>free-lists</i>	065
2.2.10 記憶池 (memory pool)	066
2.3 記憶體基本處理工具	070
2.3.1 uninitialized_copy	070
2.3.2 uninitialized_fill	071
2.3.3 uninitialized_fill_n	071
第 3 章 迭代器 (iterators) 概念與 traits 編程技法	079
3.1 迭代器設計思維 — STL 關鍵所在	079
3.2 迭代器是一種 smart pointer	080
3.3 迭代器相應型別 (associated types)	084
3.4 Traits 編程技法 — STL 源碼門鑰	085

Partial Specialization (偏特化) 的意義	086
3.4.1 迭代器相應型別之一 <i>value_type</i>	090
3.4.2 迭代器相應型別之二 <i>difference_type</i>	090
3.4.3 迭代器相應型別之三 <i>pointer_type</i>	091
3.4.4 迭代器相應型別之四 <i>reference_type</i>	091
3.4.5 迭代器相應型別之五 <i>iterator_category</i>	092
3.5 <code>std::iterator</code> class 的保證	099
3.6 <code>iterator</code> 相關源碼整理重列	101
3.7 SGI STL 的私房菜： <code>__type_traits</code>	103
第 4 章 序列式容器 (sequence containers)	113
4.1 容器概觀與分類	113
4.1.1 序列式容器 (sequence containers)	114
4.2 <code>vector</code>	115
4.2.1 <code>vector</code> 概述	115
4.2.2 <code>vector</code> 定義式摘要	115
4.2.3 <code>vector</code> 的迭代器	117
4.2.4 <code>vector</code> 的資料結構	118
4.2.5 <code>vector</code> 的建構與記憶體管理： <code>constructor</code> , <code>push_back</code>	119
4.2.6 <code>vector</code> 的元素操作： <code>pop_back</code> , <code>erase</code> , <code>clear</code> , <code>insert</code>	123
4.3 <code>list</code>	128
4.3.1 <code>list</code> 概述	128
4.3.2 <code>list</code> 的節點 (node)	129
4.3.3 <code>list</code> 的迭代器	129
4.3.4 <code>list</code> 的資料結構	131
4.3.5 <code>list</code> 的建構與記憶體管理： <code>constructor</code> , <code>push_back</code> , <code>insert</code>	132
4.3.6 <code>list</code> 的元素操作： <code>push_front</code> , <code>push_back</code> , <code>erase</code> , <code>pop_front</code> , <code>pop_back</code> , <code>clear</code> , <code>remove</code> , <code>unique</code> , <code>splice</code> , <code>merge</code> , <code>reverse</code> , <code>sort</code>	136

4.4 deque	143
4.4.1 deque 概述	143
4.4.2 deque 的中控器	144
4.4.3 deque 的迭代器	146
4.4.4 deque 的資料結構	150
4.4.5 deque 的建構與記憶體管理：ctor, push_back, push_front	152
4.4.6 deque 的元素操作：pop_back, pop_front, clear, erase, insert	161
4.5 stack	167
4.5.1 stack 概述	167
4.5.2 stack 定義式完整列表	167
4.5.3 stack 沒有迭代器	168
4.5.4 以 list 做為 stack 的底層容器	168
4.6 queue	169
4.6.1 queue 概述	169
4.6.2 queue 定義式完整列表	170
4.6.3 queue 沒有迭代器	171
4.6.4 以 list 做為 queue 的底層容器	171
4.7 heap（隱性表述，implicit representation）	172
4.7.1 heap 概述	172
4.7.2 heap 演算法	174
push_heap	174
pop_heap	176
sort_heap	178
make_heap	180
4.7.3 heap 沒有迭代器	181
4.7.4 heap 測試實例	181
4.8 priority-queue	183

4.8.1 priority-queue 概述	183
4.8.2 priority-queue 定義式完整列表	183
4.8.3 priority-queue 沒有迭代器	185
4.8.4 priority-queue 測試實例	185
4.9 slist	186
4.9.1 slist 概述	186
4.9.2 slist 的節點	186
4.9.3 slist 的迭代器	188
4.9.4 slist 的資料結構	190
4.9.5 slist 的元素操作	191
第 5 章 關聯式容器 (associated containers)	197
5.1 樹的導覽	199
5.1.1 二元搜尋樹 (binary search tree)	200
5.1.2 平衡二元搜尋樹 (balanced binary search tree)	203
5.1.3 AVL tree (Adelson-Velskii-Landis tree)	203
5.1.4 單旋轉 (Single Rotation)	205
5.1.5 雙旋轉 (Double Rotation)	206
5.2 RB-tree (紅黑樹)	208
5.2.1 安插節點	209
5.2.2 一個由上而下的程序	212
5.2.3 RB-tree 的節點設計	213
5.2.4 RB-tree 的迭代器	214
5.2.5 RB-tree 的資料結構	218
5.2.6 RB-tree 的建構與記憶體管理	221
5.2.7 RB-tree 的元素操作	223
元素安插動作 insert_equal	223
元素安插動作 insert_unique	224

真正的安插執行程序 <code>__insert</code>	224
調整 RB-tree （旋轉及改變顏色）	225
元素的搜尋 <code>find</code>	229
5.3 <code>set</code>	233
5.4 <code>map</code>	237
5.5 <code>multiset</code>	245
5.6 <code>multimap</code>	246
5.7 <code>hashtable</code>	247
5.7.1 <code>hashtable</code> 概述	247
5.7.2 <code>hashtable</code> 的桶子（buckets）與節點（nodes）	253
5.7.3 <code>hashtable</code> 的迭代器	254
5.7.4 <code>hashtable</code> 的資料結構	256
5.7.5 <code>hashtable</code> 的建構與記憶體管理	258
安插動作（ <code>insert</code> ）與表格重整（ <code>resize</code> ）	259
判知元素的落腳處（ <code>bkt_num</code> ）	262
複製（ <code>copy_from</code> ）和整體刪除（ <code>clear</code> ）	263
5.7.6 <code>hashtable</code> 運用實例（ <code>find</code> , <code>count</code> ）	264
5.7.7 <code>hash functions</code>	268
5.8 <code>hash_set</code>	270
5.9 <code>hash_map</code>	275
5.10 <code>hash_multiset</code>	279
5.11 <code>hash_multimap</code>	282
第 6 章 演算法（ <code>algorithms</code> ）	285
6.1 演算法概觀	285
6.1.1 演算法分析與複雜度表示 $O()$	286
6.1.2 STL 演算法總覽	288
6.1.3 mutating algorithms — 會改變操作對象之值	291

6.1.4 nonmutating algorithms — 不改變操作對象之值	292
6.1.5 STL 演算法的一般型式	292
6.2 演算法的泛化過程	294
6.3 數值演算法 <stl_numeric.h>	298
6.3.1 運用實例	298
6.3.2 accumulate	299
6.3.3 adjacent_difference	300
6.3.4 inner_product	301
6.3.5 partial_sum	303
6.3.6 power	304
6.3.7 itoa	305
6.4 基本演算法 <stl_algobase.h>	305
6.4.1 運用實例	305
6.4.2 equal	307
fill	308
fill_n	308
iter_swap	309
lexicographical_compare	310
max, min	312
mismatch	313
swap	314
6.4.3 copy, 強化效率無所不用其極	314
6.4.4 copy_backward	326
6.5 Set 相關演算法 (應用於已序區間)	328
6.5.1 set_union	331
6.5.2 set_intersection	333
6.5.3 set_difference	334
6.5.4 set_symmetric_difference	336
6.6 heap 演算法: make_heap, pop_heap, push_heap, sort_heap	338
6.7 其他演算法	338

6.7.1 單純的資料處理	338
adjacent_find	343
count	344
count_if	344
find	345
find_if	345
find_end	345
find_first_of	348
for_each	348
generate	349
generate_n	349
includes (應用於已序區間)	349
max_element	352
merge (應用於已序區間)	352
min_element	354
partition	354
remove	357
remove_copy	357
remove_if	357
remove_copy_if	358
replace	359
replace_copy	359
replace_if	359
replace_copy_if	360
reverse	360
reverse_copy	361
rotate	361
rotate_copy	365
search	365
search_n	366
swap_ranges	369
transform	369
unique	370
unique_copy	371
6.7.2 lower_bound (應用於已序區間)	375
6.7.3 upper_bound (應用於已序區間)	377
6.7.4 binary_search (應用於已序區間)	379
6.7.5 next_permutation	380
6.7.6 prev_permutation	382
6.7.7 random_shuffle	383

6.7.8 <code>partial_sort / partial_sort_copy</code>	386
6.7.9 <code>sort</code>	389
6.7.10 <code>equal_range</code> (應用於已序區間)	400
6.7.11 <code>inplace_merge</code> (應用於已序區間)	403
6.7.12 <code>nth_element</code>	409
6.7.13 <code>merge sort</code>	411
第 7 章 仿函式 (functor, 另名 函式物件 function objects)	413
7.1 仿函式 (functor) 概觀	413
7.2 可配接 (adaptable) 的關鍵	415
7.2.1 <code>unary_function</code>	416
7.2.2 <code>binary_function</code>	417
7.3 算術類 (Arithmetic) 仿函式	418
plus, minus, multiplies, divides, modulus, negate, identity_element	
7.4 相對關係類 (Relational) 仿函式	420
equal_to, not_equal_to, greater, greater_equal, less, less_equal	
7.5 邏輯運算類 (Logical) 仿函式	422
logical_and, logical_or, logical_not	
7.6 證同 (identity)、選擇 (select)、投射 (project)	423
identity, select1st, select2nd, project1st, project2nd	
第 8 章 配接器 (adapter)	425
8.1 配接器之概觀與分類	425
8.1.1 應用於容器, container adapters	425
8.1.2 應用於迭代器, iterator adapters	425
運用實例	427
8.1.3 應用於仿函式, functor adapters	428
運用實例	429

8.2 container adapters	434
8.2.1 stack	434
8.2.1 queue	434
8.3 iterator adapters	435
8.3.1 insert iterators	435
8.3.2 reverse iterators	437
8.3.3 stream iterators (<code>istream_iterator</code> , <code>ostream_iterator</code>)	442
8.4 function adapters	448
8.4.1 對傳回值進行邏輯否定： <code>not1</code> , <code>not2</code>	450
8.4.2 對參數進行繫結（綁定）： <code>bind1st</code> , <code>bind2nd</code>	451
8.4.3 用於函式合成： <code>compose1</code> , <code>compose2</code> （未納入標準）	453
8.4.4 用於函式指標： <code>ptr_fun</code>	454
8.4.5 用於成員函式指標： <code>mem_fun</code> , <code>mem_fun_ref</code>	456
附錄 A 參考資料與推薦讀物（Bibliography）	461
附錄 B 侯捷網站簡介	471
附錄 C STLport 的移植經驗（by 孟岩）	473
索引	481

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。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 80% 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，是一個符合開放性封閉（Open-Closed）原則的程式庫。

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笑話。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，並提升泛型編程技法的人，「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《*The C++ Standard Library*》by Nicolai M. Josuttis；想要嚴謹認識 STL 的整體架構和設計思維，以及 STL 的詳細規格，我建議你看《*Generic Programming and the STL*》by Matthew H. Austern；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《泛型思維》by 侯捷；想要深入 STL 實作技法，一窺大家風範，提昇自己的編程功力，我建議你看你手上這本《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，採循序漸進方式，遵循書中安排的章節先行瀏覽一遍。視個人功力的深淺，你可以或快或慢並依個人興趣或需要，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，以及定義於

² *The C++ Standard Template Library*, by P.J.Plauger, Alexander Al. Stepanov, Meng Lee, David R. Musser, Prentice Hall 2001/03，與本書定位相近，但在表現方式上大有不同。

`<stl_uninitialized.h>` 之中用於**記憶體管理**的基本函式，以及定義於 `<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

本書名為《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。選擇一份好的實作版本，就學習而言當然是極為重要的。我選擇的剖析對象是聲名最著，也是我個人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*》作者 Matthew H. Austern、STL 耆宿 David Musser 等人投注心力的 STL 實作版本，不論在技術層次、源碼組織、源碼可讀性上，均有卓越的表現。這份產品被納為 GNU C++ 標準程式庫，任何人皆可從國際網路上下載 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 cygnus³ 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讀者對於 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，因為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。

網絡上有個 STLport (<http://www.stlport.org>) 站點，提供一份以 SGI STL 為藍本的高度可攜性實作版本。本書附錄 C 列有孟岩先生所寫的文章，是一份 STLport 移植到 Visual C++ 和 C++ Builder 的經驗談。

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。

各章主題

本書假設你對 STL 已有基本認識和某種程度的運用經驗。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）為章節之進行依據。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努力。

- 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（adapter）

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 等編譯平台上分別測試它們。

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，我也就只使用 GNU C++ 來編譯⁴。

⁴ SGI STL 事實上是個高度可攜產品，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 C。

中英術語的運用風格

我曾經發表過一篇《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。文章收錄於侯捷網站 <http://www.jjhhou.com/article99-14.htm>。以下簡單敘述我的想法。

爲了學術界和業界的習慣，也爲了與全球科技接軌，並且也因為我所撰寫的是供專業人士閱讀的書籍而非科普讀物，我決定適量保留專業領域中被朗朗上口的英文術語。朗朗上口與否，見仁見智，我以個人閱歷做爲抉擇依據。

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並不在單字，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「完全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。但極少數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^②，例如 *insert, delete, sort*。因此視狀況而定，我可能會選擇使用英文。
- 如果某些術語關係到語言關鍵字，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，無法避免造成版面的「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。我將儘量以版面手法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。如果把英文術語視為一種符號，這些中英夾雜但帶有特殊字形的版面，並不會比市面上琳琅滿目的許多應用軟體圖解使用手冊來得突兀（而後者不是普遍為大眾所喜愛嗎☺）。我所採用的版面，都已經過一再試鍊，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；否則業界多半採用英文詞。

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」或「重載」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。請注意，並沒有絕對的實踐，有時候要看上下文情況。同時，容我再強調一次，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（adapter）、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。

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

<http://www.jjhou.com/terms.txt> 列有我個人整理的一份英中繁簡術語對照表。

版面字型風格

中 文

- 本文：細明 9.5pt
- 標題：華康粗體
- 視覺加強：華康中黑

英 文

- 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，只是為了讓您在閱讀時有比較好的效果，其本身並不具其他意義。局部的一致性更重於全體的一致性。

源碼形式與下載

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`）視同源碼處理，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 名稱大多以粗體表示。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 為配置器
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盡量設法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，多看幾眼你就會喜歡它☺。這些經過註解的 SGI STL 源碼以 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，考慮到書籍的篇幅，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，可自行上網查閱這些經過整理的源碼檔案。

⁵ 下載這些檔案並不會引發版權問題。詳見 1.3 節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。

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
- 電子檔下載

附錄 B 對侯捷網站有一些導引介紹。

推薦讀物

詳見附錄 A。這些精選讀物可為你建立紮實的泛型（Genericity）思維理論基礎與紮實的 STL 實務應用能力。

1

STL 概論 與 存本簡介

STL，雖然是一套程式庫（library），卻不只是一般印象中的程式庫，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，說是軟體組件技術發展史上的一個大突破點，它也當之無愧。

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，以及一種得以製造出「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（components），從結構化設計、模組化設計、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。

為的就是復用性（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，竟是為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 後兩者是科技到達不了的幽暗世界。就算 STL, COM, CORBA, OO, Patterns...也無能為力☹。

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，直接而明朗，令大多數人有最立即明顯的感受。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）…，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*（傳回真假值的一元判斷式）、*Binary Predicate*（傳回真假值的二元判斷式）…，更高階的概念包括 *sequence container*（序列式容器）、*associative container*（關聯式容器）…。

STL 的創新價值便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，形成了一個嚴謹的介面標準。在此介面之下，任何組件有最大的獨立性，並以所謂迭代器（iterator）膠合起來，或以所謂配接器（adapter）互相配接，或以所謂仿函式（functor）動態選擇某種策略（policy 或 strategy）。

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

型別參數化，藉由兩者結合並透過 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 的起源

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，但 Alexander 卻已經意識到，C++ 允許程式員透過指標以極佳彈性處理記憶體，這一點正是既要求一般化（泛型）又不失效能的一個重要關鍵。

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，成為 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為 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。因為 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 不是威靈頓公爵擊敗拿破崙的那個地方，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,但很受歡迎。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 成為 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為應該給予這份提案一個機會,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,以及他們的辛苦成果而言,標示的卻是巨大的成功。投票結果,80% 贊成,20% 反對,於是 STL 進入了 C++ 標準化的正式流程,並終於成為 1998/09 定案的 C++ 標準規格中的 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 刊出一篇名為 *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,均有詳細敘述,本處不再贅述。侯捷網站(見附錄 B)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 功能與運用

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*,

The Annotated STL Sources

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 STL 容器都附帶有自己專屬的迭代器 — 是的，只有容器設計者才知道如何巡訪自己的元素。原生指標 (`native pointer`) 也是一種迭代器。
4. 仿函式 (`functors`)：行為類似函式，可做為演算法的某種策略 (`policy`)，詳見第 7 章。從實作的角度看，仿函式是一種重載了 `operator()` 的 `class` 或 `class template`。一般函式指標可視為狹義的仿函式。
5. 配接器 (`adapters`)：一種用來修飾容器 (`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，其實只能算是一種容器配接器，因為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，稱為 `function adapter`，改變 `container` 介面者，稱為 `container adapter`，改變 `iterator` 介面者，稱為 `iterator adapter`。配接器的實作技術很難一言以蔽之，必須逐一分析，詳見第 8 章。
6. 配置器 (`allocators`)：負責空間配置與管理，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 已成為 C++ 標準程式庫的大脈系，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 並非以二進位碼 (`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再有副檔名，但或許是爲了回溯相容，或許是爲了內部組織規劃，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 **Dinkumware** 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 版本不僅有一線裝配，還有二線裝配，例如 GNU C++ 的 SGI 版本不但有一線的 `<vector.h>` 和 `<vector>`，還有二線的 `<stl_vector.h>`。

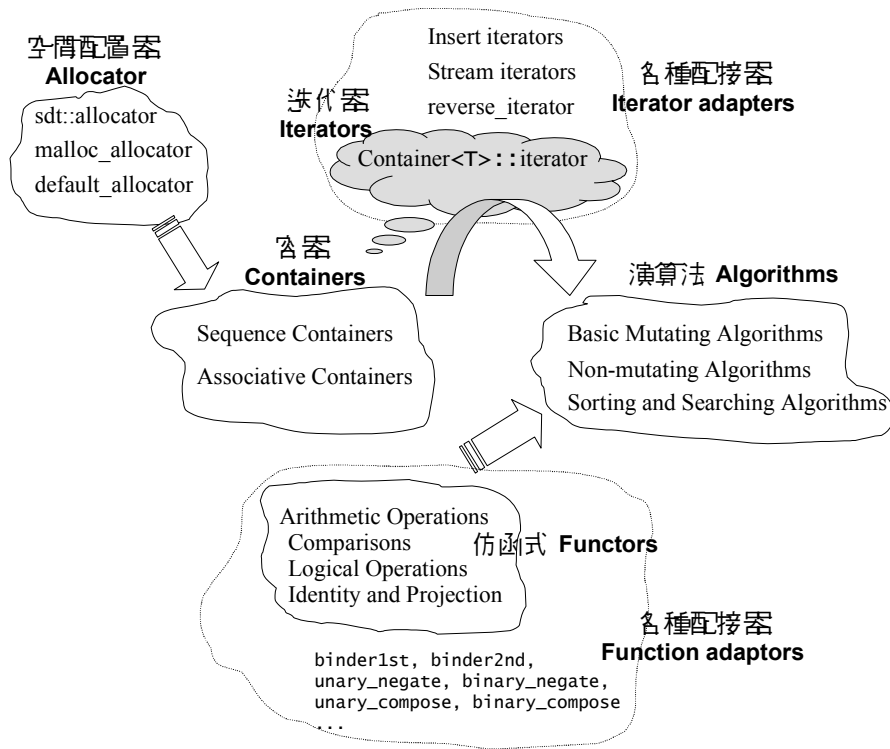


圖 1-1 STL 六大組件的交互關係：Container 透過 Allocator 取得資料儲存空間，Algorithm 透過 Iterator 存取 Container 內容，Functor 可以協助 Algorithm 完成不同的策略變化，Adapter 可以修飾或套接 Functor。

如果只是應用 STL，請各位讀者務必從此養成良好習慣，遵照 C++ 規範，使用無副檔名的表頭檔⁴。如果進入本書層次，探究 STL 源碼，就得清楚所有這些表頭檔的組織分佈。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。這對使用者而言是透通的。

1.3 GNU 源碼開放精神

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，一般統稱為 **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 Stallman⁵（理察·史托曼）。他認為私藏源碼是一種違反人性的罪惡行為。他認為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 Foundation⁶，簡稱 **FSF**），寫下著名的 GNU 宣言（GNU Manifesto），開始進行名為 GNU 的開放改革計劃。

GNU⁷ 這個名稱是電腦族的幽默展現，代表 **GNU is Not Unix**。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。這原本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，並對從中獲利抱以預期。於是開始要求大學校園內的相關研究人員簽約，要求他們不得公開或透露 UNIX 源碼，並贊助 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、NetBSD⁸...

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，或譯為「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 的這種行為視為思想箝制，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 認為 AT&T 對大學的贊助，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，並稱之為 GNU：GNU is Not Unix。

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 License⁹，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 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。得之於人，捨於人。

GPL 對於版權（copyright）觀念帶來巨大的挑戰，甚至被稱為「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，導致種種不同的反對意見，包括可能造成經濟競爭力薄弱等等。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。其中最甚者為 GNU GPL 和 BSD License 的擁護者。

1998 年，自由軟體社群企圖創造出一個新名詞 **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 的定義共有 9 條¹⁰，任何軟體只要符合這 9 條，就可稱呼自己為 open source 軟體。

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

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

本書所採用的 GCC 套件是 **Cygnus C++2.91 for Windows**，又稱為 **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，並因此成為 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，為了統一事權，GCC 指導委員會開始考慮整合 1997 成立的 EGCS（**Experimental/Enhanced GNU Compiler System**）計劃。這個計劃採用比較開放的開發態度，比標準 GCC 涵蓋更多優化技術和更多 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 一樣的開發方式。自 1999 年起，EGCS 正式成為唯一的 GCC 官方維護機構。

1.4 HP 實作版本

HP 版本是所有 STL 實作版本的濫觴。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。這種授權並不屬於 GNU GPL 範疇，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
 * 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 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 版本承繼 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，因為 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，所以當然你可以在 Visual C++ 的 "include" 子目錄下（例如 C:\msdev\VC98\Include）找到所有 STL 表頭檔，但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，主要因為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 PJ 版本的表現也受影響。

這項產品目前由 Dinkumware¹² 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 詳見 <http://www.dinkumware.com>

1.6 Rouge Wave 實作版本

RogueWave 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 版本承繼 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，因為 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。

Rogue Wave 版本被 C++Builder 採用，所以當然你可以在 C++Builder 的 "include" 子目錄下（例如 C:\Inprise\CBuilder4\Include）找到所有 STL 表頭檔，但是

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 RW 版本正面的影響。

1.7 STLport 實作版本

網絡上有個 STLport 站點，提供一個以 SGI STL 為藍本的高度可攜性實作版本。本書附錄 C 列有孟岩先生所寫的一篇文章，介紹 STLport 移植到 Visual C++ 和 C++ Builder 的經驗。SGI STL（下節介紹）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 Silicon Graphics Computer Systems, Inc. 公司發展，承繼 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;
}
```

下面是對應於先前所列之 RW 版本的源碼實例（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 C++ 主流編譯器中表現耀眼，連帶地給予了 SGI STL 正面影響。事實上 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 也採用某些 GPL（廣泛性開放授權）檔案，例如 `<std\complex.h>`，`<std\complex.cc>`，`<std\bastring.h>`，`<std\bastring.cc>`。這些檔案都有如下的聲明：

```
// 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，773,042 bytes：

<code>algo.h</code>	<code>algotbase.h</code>	<code>algorithm</code>
<code>alloc.h</code>	<code>builtinbuf.h</code>	<code>bvector.h</code>
<code>cassert</code>	<code>cctype</code>	<code>cerrno</code>
<code>cfloat</code>	<code>ciso646</code>	<code>climits</code>

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	iosdio.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

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

(2) C++ Standard 定義前, 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` 取代,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>
algbase.h	2,086		ref. <stl_algbase.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_algbase.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 hash-table 完成之 map, multimap
stl_hash_set.h	12,990	5.7	以 hash-table 完成之 set, multiset
stl_hashtable.h	26,922	5.6	hash-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	5.4	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 templated 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 C++ 語言的支援程度不盡相同。做為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> 檔案起始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' 定義為一個 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 (亦稱為
//     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' 為一個 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 定義為一個
//      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
#   if !defined(_PARTIAL_SPECIALIZATION_OF_CLASS_TEMPLATES)
#       define __STL_CLASS_PARTIAL_SPECIALIZATION
#   endif
#   if !defined(_MEMBER_TEMPLATES)
#       define __STL_MEMBER_TEMPLATES
#   endif
#   if !defined(_EXPLICIT_IS_KEYWORD)
#       define __STL_NEED_EXPLICIT
#   endif
#   if !defined(_EXCEPTIONS)
#       define __STL_USE_EXCEPTIONS
#   endif
#   if (_COMPILER_VERSION >= 721) && defined(_NAMESPACES)

```

```

#   define __STL_USE_NAMESPACES
# endif
# if !defined(_NO_THREADS) && !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(_NO_THREADS) && __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》第 4 章。

```

```

#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: lconfig.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 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: lconfig3.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
// 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: lconfig5.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
// 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 C++ 語法書籍。

```
// file: lconfig6.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 // 只為說明。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 // 只為說明。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: lconfig8.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
// 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: lconfig10.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
// vc6[o] cb4[o] gcc[o]

#include <iostream>
#include <cstdint> // 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 的預設值為 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: lconfig11.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
// 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: lconfig-null-template-arguments.cpp
// test __STL_NULL_TMPL_ARGS in <stl_config.h>
// ref. C++ Primer 3/e, p.834: bound friend function template
// 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: lconfig-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
// vc6[o] cb4[x] gcc[o]

#include <iostream>
using namespace std;

// 將 __STL_TEMPLATE_NULL 定義為 template<>，可以。
// 若定義為 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 暫時物件的產生與運用

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: lconfig-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《More Effective C++》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: lconfig-inclass-init.cpp
// test in-class initialization of static const integral members
// ref. C++ Primer 3/e, p.643
// 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，因為任何一個迭代器都必須實作出前進（*increment*, `operator++`）和取值（*dereference*, `operator*`）功能，前者還分為前置式（*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《*More Effective C++*》條款 6：Distinguish between prefix and postfix forms of increment and decrement operators

```
// file: lconfig-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 轉為 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為錯誤
    }
};
```

```

    }

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 前閉後開區間標示法 [)

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，或說 *pass the end*）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：

```

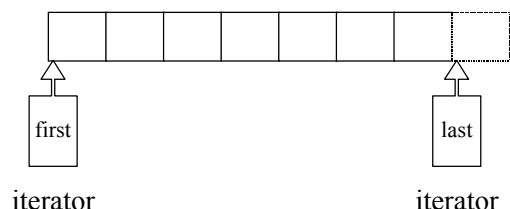
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}

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）、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: lqsort.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。

為此，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，它就成為一個仿函式。至於要成為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: lfunctor.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，因為整個 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。

為什麼不說 **allocator** 是**記憶體**配置器而說它是**空間**配置器呢？因為，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，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, const 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 設計一個陽春的空間配置器，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
#ifdef _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」是因為，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」則是因為，RW STL 在很多容器身上運用了緩衝區，情況複雜得多，`JJ::allocator` 無法與之相容。至於完全無法應用於 SGI STL 是因為，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為 `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::allocator<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，因為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為 `alloc`。例如下面的 `vector` 宣告：

```

template <class T, class Alloc = alloc> // 預設使用 alloc 為配置器
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、名為 `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 <algbase.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，為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/解放行為（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 第 1,3 章。

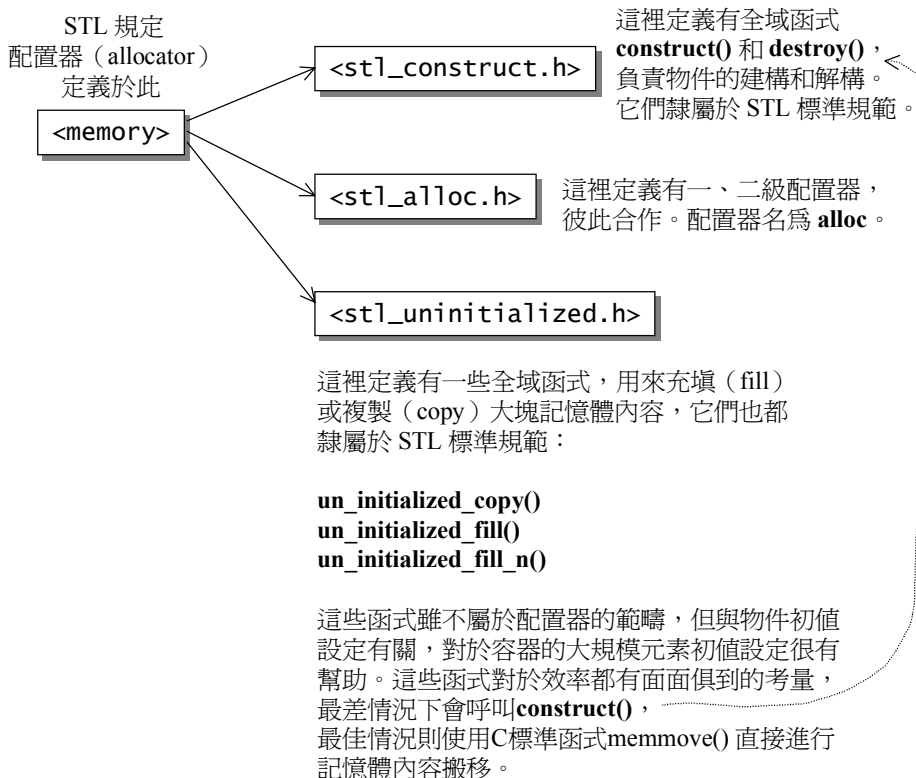
`Foo::~Foo()` 將物件解構，(2) 呼叫 `::operator delete` 釋放記憶體。

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 T1::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為 char* 和 wchar_t* 的特化版
inline void destroy(char*, char*) {}
inline void destroy(wchar_t*, wchar_t*) {}
```


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，對效率是一種斬傷。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「物件解構式是否為 *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為，和記憶體釋放前的物件解構行為，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，視之為「足夠大」，便呼叫第一級配置器；當配置區塊小於 128bytes，視之為「過小」，為了降低額外負擔 (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 為第一級配置器
# else
...
// 令 alloc 為第二級配置器
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` 被定義為第一級或第二級配置器，SGI 還為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，因為無論如何，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 轉為個別元素的大小（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 為配置器
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。

SGI STL 第一級配置器

```
template<int inst>
class __malloc_alloc_template { ... };
其中：
```

1. allocate() 直接使用 malloc()，deallocate() 直接使用 free()。
2. 模擬 C++ 的 set_new_handler() 以處理記憶體不足的狀況

SGI STL 第二級配置器

```
template <bool threads, int inst>
class __default_alloc_template { ... };
其中：
```

1. 維護16個自由串列（free lists），負責16種小型區塊的次配置能力。記憶池（memory pool）以 malloc() 配置而得。如果記憶體不足，轉呼叫第一級配置器（那兒有處理程序）。
2. 如果需求區塊大於 128bytes，就轉呼叫第一級配置器。

圖 2-2a 第一級配置器與第二級配置器

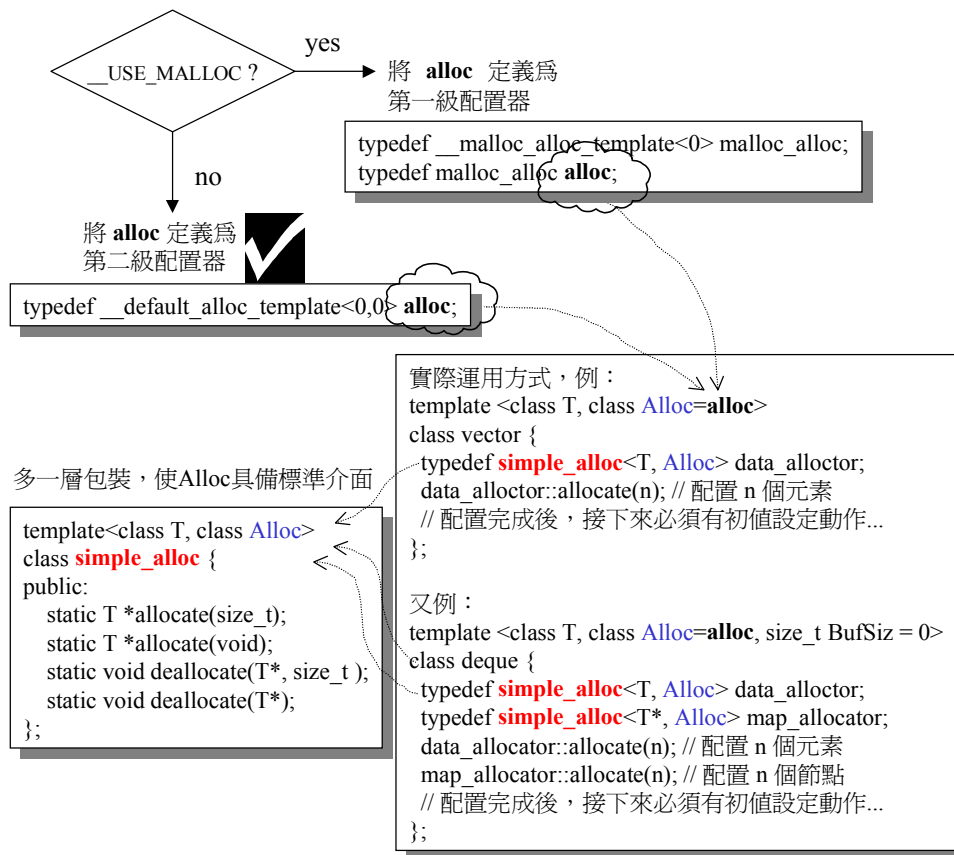


圖 2-2b 第一級配置器與第二級配置器，其包裝介面和運用方式

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
// 初值為 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 指定為 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，因為它並非使用 `::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

⁷ 詳見《Effective C++》2e, 條款 7：Be prepared for out-of-memory conditions.

出 `std::bad_alloc` 異常狀態之前，會先呼叫由客端指定的處理常式。此處理常式通常即被稱為 `new-handler`。`new-handler` 解決記憶體不足的作法有特定的模式，請參考《Effective C++》2e 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*。

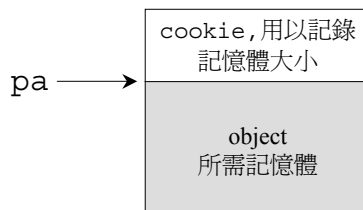


圖 2-3 索取任何一塊記憶體，都得有一些「稅」要繳給系統。

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，此法又稱為次層配置 (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。為了方便管理，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，就自動調整為 32 bytes)，並維護 16 個 *free-lists*，各自管理大小分別為 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，為了維護串列 (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* 可被視為一個指標，指向相同形式的另一個 *obj*。從其第二欄位觀之，*obj* 可被視為一個指標，指向實際區塊，如圖 2-4。一物二用的結果是，不會為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。

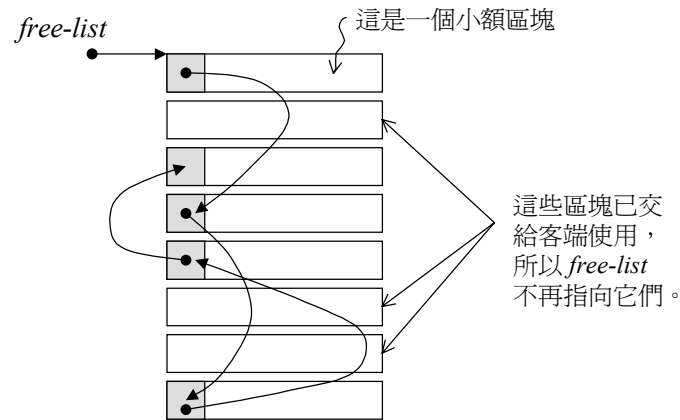


圖 2-4 自由串列 (free-list) 的實作技巧

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);
    }
}
```

2.2.7 空間配置 関数 allocate()

```
// 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。

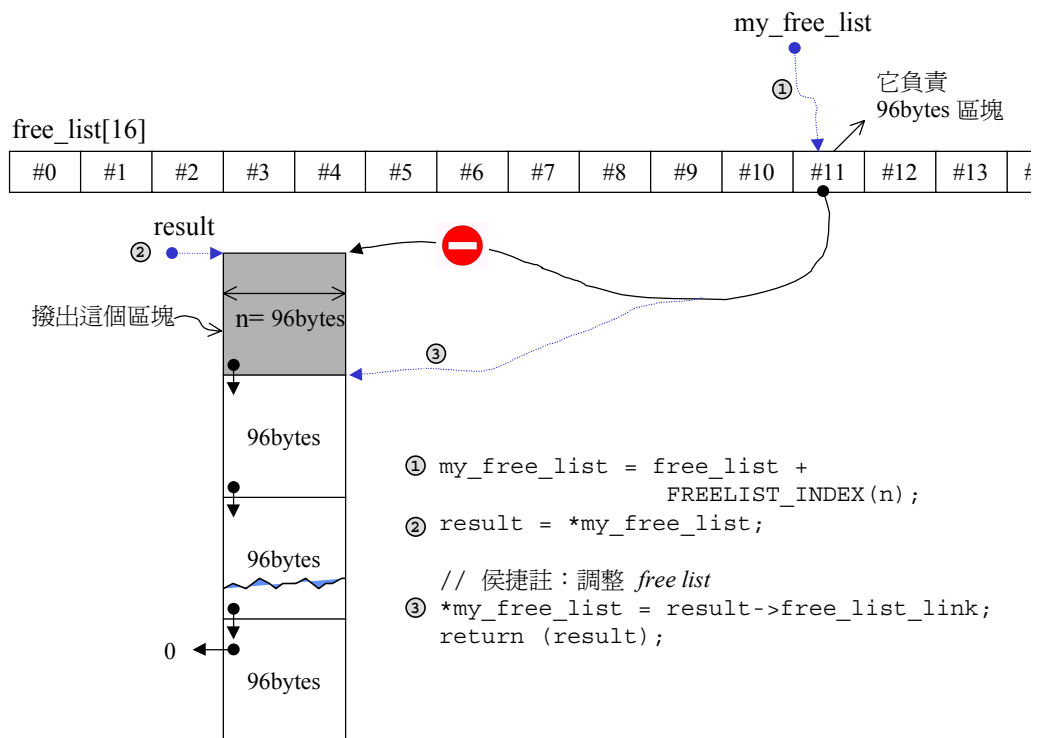


圖 2-5 區塊自 *free list* 撥出。閱讀次序請循圖中編號。

2.2.8 空間釋還函式 `deallocate()`

身為一個配置器，`__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。

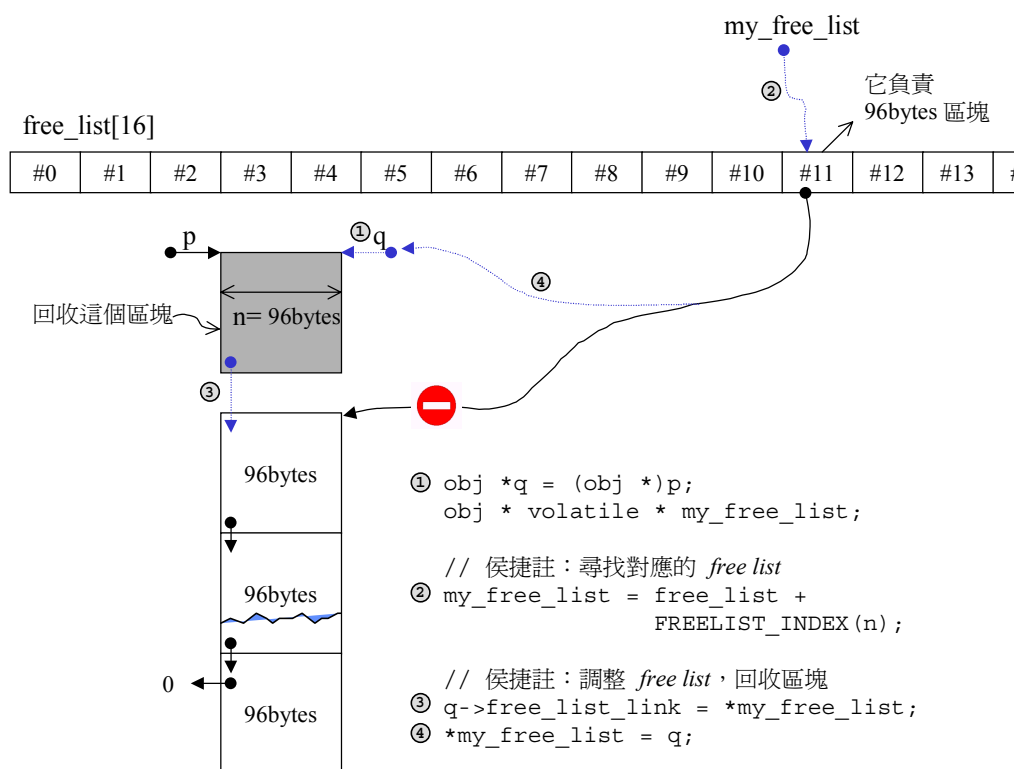


圖 2-6 區塊回收，納入 *free list*。閱讀次序請循圖中編號。

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()` 準備為 *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：

```
// 傳回一個大小為 n 的物件，並且有時候會為適當的 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 個區塊做為 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，因為第 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) {
    // 記憶體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 {
    // 記憶體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);
    // 以下試著讓記憶體池中的殘餘零頭還有利用價值。
    if (bytes_left > 0) {
        // 記憶體池內還有一些零頭，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，將記憶體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，用來挹注記憶體池。
    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，因為那在多行程 (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，為了修正 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，此時必需利用 `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。

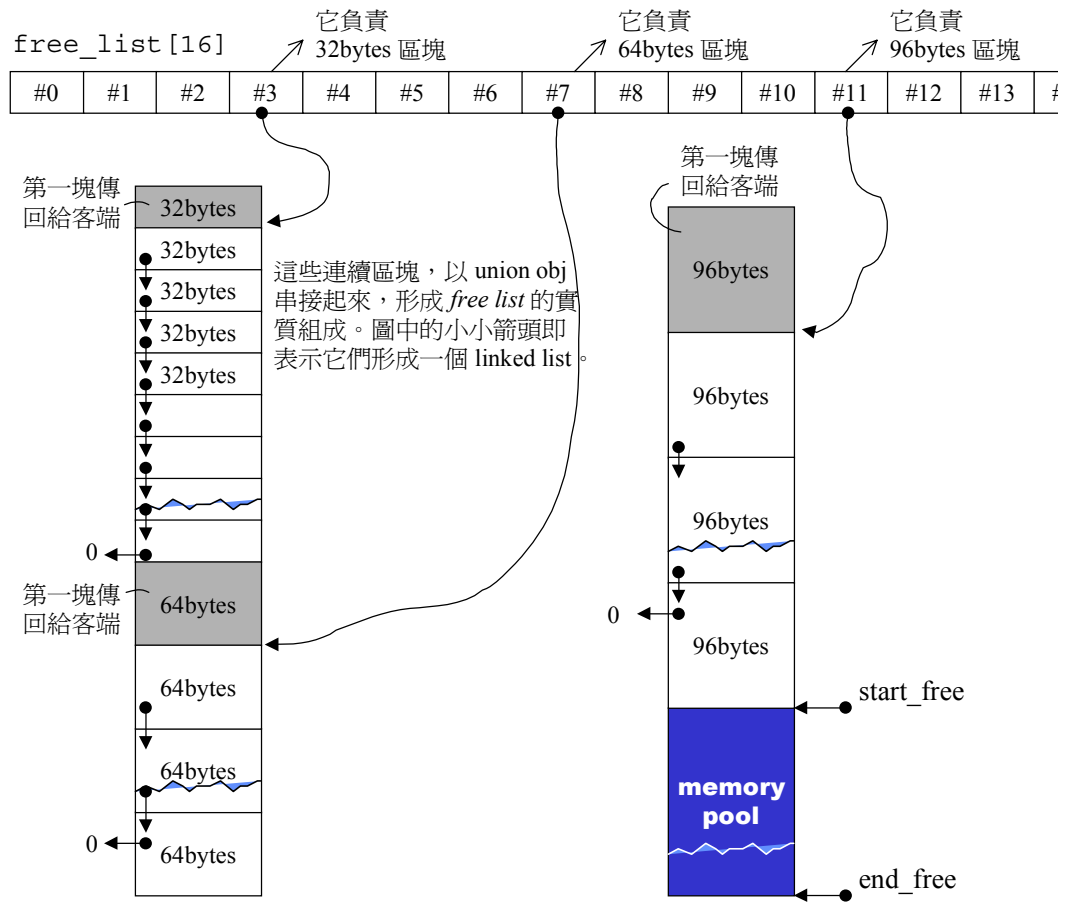


圖 2-7 記憶池 (memory pool) 實際操練結果

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 為配置器
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 已經把它設為第二級配置器，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為分離開來。如果做為輸出目的地的 `[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`，為身為輸入來源之 `[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()` 這樣的函式會為你帶來很大的幫助，因為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為分離開來。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為分離開來。它會為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為 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;
    // 為求閱讀順暢，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為 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;
    // 為求閱讀順暢，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）來執行複製行為。因此 SGI 得以為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為 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;
    // 為求閱讀順暢, 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。

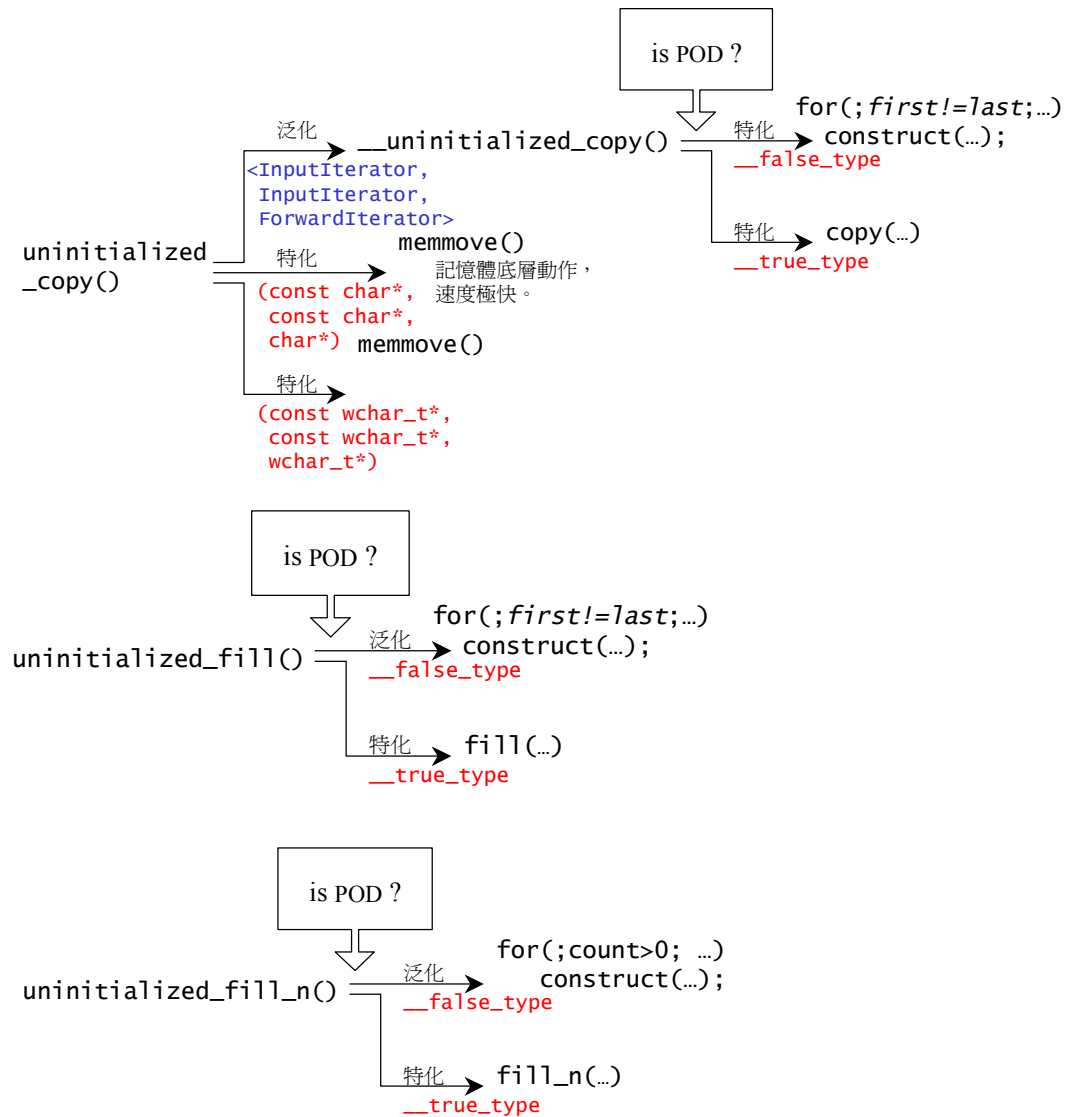


圖 2-8 三個記憶體基本函式的泛型版本與特化版本。

3

迭代器 (iterators) 概念、與 traits 編程技法

迭代器 (iterators) 是一種抽象的設計概念，現實程式語言中並沒有直接對映於這個概念的實物。《Design Patterns》一書提供有 23 個設計樣式 (design patterns) 的完整描述，其中 *iterator* 樣式定義如下：提供一種方法，俾得依序訪問某個聚合物 (容器) 所含的各個元素，而且無需暴露該聚合物的內部表示方式。

3.1 迭代器設計思維 — STL 關鍵所在

不論是泛型思維或 STL 的實際運用，迭代器 (iterators) 都扮演重要角色。STL 的中心思想在於，將資料容器 (containers) 和演算法 (algorithms) 分開，彼此獨立設計，最後再以一帖膠著劑將它們撮合在一起。容器和演算法的泛型化，從技術角度來看並不困難，C++ 的 class templates 和 function templates 可分別達成目標。如何設計出兩者之間的良好膠著劑，才是大難題。

以下是容器、演算法、迭代器 (iterator, 扮演黏膠角色) 的合作展示。以演算法 `find()` 為例，它接受兩個迭代器和一個「搜尋標的」：

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

只要給予不同的迭代器，find() 便能夠對不同的容器做搜尋動作：

```
// file : 3find.cpp
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    const int arraySize = 7;
    int ia[arraySize] = { 0,1,2,3,4,5,6 };

    vector<int> ivect(ia, ia+arraySize);
    list<int> ilist(ia, ia+arraySize);
    deque<int> ideque(ia, ia+arraySize); // 注意：VC6 [x]，未符合標準

    vector<int>::iterator it1 = find(ivect.begin(), ivect.end(), 4);
    if (it1 == ivect.end())
        cout << "4 not found." << endl;
    else
        cout << "4 found. " << *it1 << endl;
    // 執行結果：4 found. 4

    list<int>::iterator it2 = find(ilist.begin(), ilist.end(), 6);
    if (it2 == ilist.end())
        cout << "6 not found." << endl;
    else
        cout << "6 found. " << *it2 << endl;
    // 執行結果：6 found. 6

    deque<int>::iterator it3 = find(ideque.begin(), ideque.end(), 8);
    if (it3 == ideque.end())
        cout << "8 not found." << endl;
    else
        cout << "8 found. " << *it3 << endl;
    // 執行結果：8 not found.
}
```

從這個例子看來，迭代器似乎依附在容器之下。是嗎？有沒有獨立而泛用的迭代器？我們又該如何自行設計特殊的迭代器？

3.2 迭代器 (iterator) 是 - 種 smart pointer

迭代器是一種行為類似指標的物件，而指標的各種行為中最常見也最重要的便是

內容提領 (*dereference*) 和成員取用 (*member access*)，因此迭代器最重要的編程工作就是對 `operator*` 和 `operator->` 進行多載化 (*overloading*) 工程。關於這一點，C++ 標準程式庫有一個 `auto_ptr` 可供我們參考。任何一本詳盡的 C++ 語法書籍都應該談到 `auto_ptr` (如果沒有，扔了它☹)，這是一個用來包裝原生指標 (*native pointer*) 的物件，聲名狼藉的記憶體漏洞 (*memory leak*) 問題可藉此獲得解決。`auto_ptr` 用法如下，和原生指標一模一樣：

```
void func()
{
    auto_ptr<string> ps(new string("jjhou"));

    cout << *ps << endl;           // 輸出: jjhou
    cout << ps->size() << endl;     // 輸出: 5
    // 離開前不需 delete, auto_ptr 會自動釋放記憶體
}
```

函式第一行的意思是，以算式 `new` 動態配置一個初值為 "jjhou" 的 `string` 物件，並將所得結果 (一個原生指標) 做為 `auto_ptr<string>` 物件的初值。注意，`auto_ptr` 角括號內放的是「原生指標所指物件」的型別，而不是原生指標的型別。

`auto_ptr` 的源碼在表頭檔 `<memory>` 中，根據它，我模擬了一份陽春版，可具體說明 `auto_ptr` 的行為與能力：

```
// file: 3autoptr.cpp
template<class T>
class auto_ptr {
public:
    explicit auto_ptr(T *p = 0): pointee(p) {}
    template<class U>
    auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}
    ~auto_ptr() { delete pointee; }

    template<class U>
    auto_ptr<T>& operator=(auto_ptr<U>& rhs) {
        if (this != &rhs) reset(rhs.release());
        return *this;
    }
    T& operator*() const { return *pointee; }
    T* operator->() const { return pointee; }
    T* get() const { return pointee; }
    // ...
private:
    T *pointee;
```



```
};
```

其中關鍵字 `explicit` 和 `member template` 等編程技法，並不是這裡的講述重點，相關語法和語意請參閱 [Lippman98] 或 [Stroustrup97]。

有了模倣對象，現在我們來為 `list`（串列）設計一個迭代器¹。假設 `list` 及其節點的結構如下：

```
// file: 3mylist.h
template <typename T>
class List
{
    void insert_front(T value);
    void insert_end(T value);
    void display(std::ostream &os = std::cout) const;
    // ...
private:
    ListItem<T>* _end;
    ListItem<T>* _front;
    long _size;
};

template <typename T>
class ListItem
{
public:
    T value() const { return _value; }
    ListItem* next() const { return _next; }
    ...
private:
    T _value;
    ListItem* _next; // 單向串列 (single linked list)
};
```

如何將這個 `List` 套用到先前所說的 `find()` 呢？我們需要為它設計一個行為類似指標的外衣，也就是一個迭代器。當我們提領（*dereference*）此一迭代器，傳回的應該是個 `ListItem` 物件；當我們累加該迭代器，它應該指向下一個 `ListItem` 物件。為了讓此迭代器適用於任何型態的節點，而不只限於 `ListItem`，我們可以將它設計為一個 `class template`：

¹ [Lippman98] 5.11 節有一個非泛型的 `list` 實例可以參考。《泛型思維》書中有一份泛型版本的 `list` 的完整設計與說明。

```

// file : 3mylist-iter.h
#include "3mylist.h"

template <class Item> // Item 可以是單向串列節點或雙向串列節點。
struct ListIter      // 此處這個迭代器特定只為串列服務，因為其
{                    // 獨特的 operator++ 之故。
    Item* ptr; // 保持與容器之間的一個聯繫 (keep a reference to Container)

    ListIter(Item* p = 0) // default ctor
    : ptr(p) { }

    // 不必實作 copy ctor，因為編譯器提供的預設行為已足夠。
    // 不必實作 operator=，因為編譯器提供的預設行為已足夠。

    Item& operator*() const { return *ptr; }
    Item* operator->() const { return ptr; }

    // 以下兩個 operator++ 遵循標準作法，參見 [Meyers96] 條款 6
    // (1) pre-increment operator
    ListIter& operator++()
    { ptr = ptr->next(); return *this; }

    // (2) post-increment operator
    ListIter operator++(int)
    { ListIter tmp = *this; ++*this; return tmp; }

    bool operator==(const ListIter& i) const
    { return ptr == i.ptr; }
    bool operator!=(const ListIter& i) const
    { return ptr != i.ptr; }
};

```

現在我們可以這樣子將 `List` 和 `find()` 藉由 `ListIter` 黏合起來：

```

// 3mylist-iter-test.cpp
void main()
{
    List<int> mylist;

    for(int i=0; i<5; ++i) {
        mylist.insert_front(i);
        mylist.insert_end(i+2);
    }
    mylist.display();    // 10 ( 4 3 2 1 0 2 3 4 5 6 )

    ListIter<ListItem<int> > begin(mylist.front());
    ListIter<ListItem<int> > end;    // default 0, null
    ListIter<ListItem<int> > iter; // default 0, null

```

```

    iter = find(begin, end, 3);
    if (iter == end)
        cout << "not found" << endl;
    else
        cout << "found. " << iter->value() << endl;
    // 執行結果: found. 3

    iter = find(begin, end, 7);
    if (iter == end)
        cout << "not found" << endl;
    else
        cout << "found. " << iter->value() << endl;
    // 執行結果: not found
}

```

注意，由於 `find()` 函式內以 `*iter != value` 來檢查元素值是否吻合，而本例之中 `value` 的型別是 `int`，`iter` 的型別是 `ListItem<int>`，兩者之間並無可供使用的 `operator!=`，所以我必須另外寫一個全域的 `operator!=` 多載函式，並以 `int` 和 `ListItem<int>` 做為它的兩個參數型別：

```

template <typename T>
bool operator!=(const ListItem<T>& item, T n)
{ return item.value() != n; }

```

從以上實作可以看出，為了完成一個針對 `List` 而設計的迭代器，我們無可避免地曝露了太多 `List` 實作細節：在 `main()` 之中為了製作 `begin` 和 `end` 兩個迭代器，我們曝露了 `ListItem`；在 `ListIter` class 之中為了達成 `operator++` 的目的，我們曝露了 `ListItem` 的操作函式 `next()`。如果不是為了迭代器，`ListItem` 原本應該完全隱藏起來不曝光的。換句話說，要設計出 `ListIter`，首先必須對 `List` 的實作細節有非常豐富的瞭解。既然這無可避免，乾脆就把迭代器的開發工作交給 `List` 的設計者好了，如此一來所有實作細節反而得以封裝起來不被使用者看到。這正是為什麼每一種 STL 容器都提供有專屬迭代器的緣故。

3.3 迭代器相態型別 (associated types)

上述的 `ListIter` 提供了一個迭代器雛形。如果將思想拉得更高遠一些，我們便會發現，演算法之中運用迭代器時，很可能會用到其相應型別 (associated type)。什麼是相應型別？迭代器所指之物的型別便是其一。假設演算法中有必要宣告一個變數，以「迭代器所指物件的型別」為型別，如何是好？畢竟 C++ 只支援

`sizeof()`，並未支援 `typeof()`！即便動用 RTTI 性質中的 `typeid()`，獲得的也只是型別名稱，不能拿來做變數宣告之用。

解決辦法是有：利用 `function template` 的引數推導（`argument deduction`）機制。

例如：

```

> template <class I, class T>
void func_impl(I iter, T t)
{
    T tmp; // 這裡解決了問題。T 就是迭代器所指之物的型別，本例為 int

    // ... 這裡做原本 func() 應該做的全部工作
};

template <class I>
inline
void func(I iter)
{
    func_impl(iter, *iter); // func 的工作全部移往 func_impl
}

int main()
{
    int i;
    func(&i);
}

```

我們以 `func()` 為對外介面，卻把實際動作全部置於 `func_impl()` 之中。由於 `func_impl()` 是一個 `function template`，一旦被呼叫，編譯器會自動進行 `template` 引數推導。於是導出型別 `T`，順利解決了問題。

迭代器相應型別（`associated types`）不只是「迭代器所指物件的型別」一種而已。根據經驗，最常用的相應型別有五種，然而並非任何情況下任何一種都可利用上述的 `template` 引數推導機制來取得。我們需要更全面的解法。

3.4 Traits 編程技法 — STL 源碼門鑰

迭代器所指物件的型別，稱為該迭代器的 *value type*。上述的引數型別推導技巧雖然可用於 *value type*，卻非全面可用：萬一 *value type* 必須用於函式的傳回值，就束手無策了，畢竟函式的「`template` 引數推導機制」推而導之的只是引數，無

法推導函式的回返值型別。

我們需要其他方法。宣告巢狀型別似乎是個好主意，像這樣：

```
template <class T>
struct MyIter
{
    typedef T value_type; // 巢狀型別宣告 (nested type)
    T* ptr;
    MyIter(T* p=0) : ptr(p) { }
    T& operator*() const { return *ptr; }
    // ...
};

template <class I>
typename I::value_type // 這一整行是 func 的回返值型別
func(I ite)
{ return *ite; }

// ...
MyIter<int> ite(new int(8));
cout << func(ite); // 輸出：8
```

注意，`func()` 的回返型別必須加上關鍵字 `typename`，因為 `T` 是一個 `template` 參數，在它被編譯器具現化之前，編譯器對 `T` 一無所悉，換句話說編譯器此時並不知道 `MyIter<T>::value_type` 代表的是一個型別或是一個 `member function` 或是一個 `data member`。關鍵字 `typename` 的用意在告訴編譯器說這是一個型別，如此才能順利通過編譯。

看起來不錯。但是有個隱晦的陷阱：並不是所有迭代器都是 `class type`。原生指標就不是！如果不是 `class type`，就無法為它定義巢狀型別。但 STL（以及整個泛型思維）絕對必須接受原生指標做為一種迭代器，所以上面這樣還不夠。有沒有辦法可以讓上述的一般化概念針對特定情況（例如針對原生指標）做特殊化處理呢？

是的，`template partial specialization` 可以做到。

Partial Specialization（偏特化）的意義

任何完整的 C++ 語法書籍都應該對 `template partial specialization` 有所說明（如果沒有，扔了它☹）。大致的意義是：如果 `class template` 擁有一個以上的 `template` 參數，我們可以針對其中某個（或數個，但非全部）`template` 參數進行特化工作。

換句話說我們可以在泛化設計中提供一個特化版本（也就是將泛化版本中的某些 `template` 參數賦予明確的指定）。

假設有一個 `class template` 如下：

```
template<typename U, typename V, typename T>
class C { ... };
```

`partial specialization` 的字面意義容易誤導我們以為，所謂「偏特化版」一定是對 `template` 參數 `U` 或 `V` 或 `T`（或某種組合）指定某個引數值。事實不然，[Austern99] 對於 `partial specialization` 的意義說得十分得體：「所謂 `partial specialization` 的意思是提供另一份 `template` 定義式，而其本身仍為 `templated`」。《泛型技術》一書對 `partial specialization` 的定義是：「針對（任何）`template` 參數更進一步的條件限制，所設計出來的一個特化版本」。由此，面對以下這麼一個 `class template`：

```
template<typename T>
class C { ... }; // 這個泛化版本允許（接受）T 為任何型別
```

我們便很容易接受它有一個型式如下的 `partial specialization`：

```
template<typename T>
class C<T*> { ... }; // 這個特化版本僅適用於「T 為原生指標」的情況
// 「T 為原生指標」便是「T 為任何型別」的一個更進一步的條件限制
```

有了這項利器，我們便可以解決前述「巢狀型別」未能解決的問題。先前的問題是，原生指標並非 `class`，因此無法為它們定義巢狀型別。現在，我們可以針對「迭代器之 `template` 引數為指標」者，設計特化版的迭代器。

提高警覺，我們進入關鍵地帶了。下面這個 `class template` 專門用來「萃取」迭代器的特性，而 *value type* 正是迭代器的特性之一：

```
template <class I>
struct iterator_traits { // traits 意為「特性」
    typedef typename I::value_type value_type;
};
```

這個所謂的 **traits**，其意義是，如果 `I` 定義有自己的 *value type*，那麼透過這個 **traits** 的作用，萃取出來的 `value_type` 就是 `I::value_type`。換句話說如果 `I` 定義有自己的 *value type*，先前那個 `func()` 可以改寫成這樣：

```
template <class I>
typename iterator_traits<I>::value_type // 這一整行是函式回返型別
func(I ite)
{ return *ite; }
```

但這除了多一層間接性，又帶來什麼好處？好處是 **traits** 可以擁有特化版本。現在，我們令 `iterator_traits` 擁有一個 `partial specializations` 如下：

```
template <class T>
struct iterator_traits<T*> { // 偏特化版 — 迭代器是個原生指標
    typedef T value_type;
};
```

於是，原生指標 `int*` 雖然不是一種 `class type`，亦可透過 **traits** 取其 *value type*。這就解決了先前的問題。

但是請注意，針對「指向常數物件的指標 (`pointer-to-const`)」，下面這個式子得到什麼結果：

```
iterator_traits<const int*>::value_type
```

獲得的是 `const int` 而非 `int`。這是我們期望的嗎？我們希望利用這種機制來宣告一個暫時變數，使其型別與迭代器的 *value type* 相同，而現在，宣告一個無法賦值（因 `const` 之故）的暫時變數，沒什麼用！因此，如果迭代器是個 `pointer-to-const`，我們應該設法令其 *value type* 為一個 `non-const` 型別。沒問題，只要另外設計一個特化版本，就能解決這個問題：

```
template <class T>
struct iterator_traits<const T*> { // 偏特化版 — 當迭代器是個 pointer-to-const
    typedef T value_type;          // 萃取出來的型別應該是 T 而非 const T
};
```

現在，不論面對的是迭代器 `MyIter`，或是原生指標 `int*` 或 `const int*`，都可以透過 **traits** 取出正確的（我們所期望的）*value type*。

圖 3-1 說明 **traits** 所扮演的「特性萃取機」角色，萃取各個迭代器的特性。這裡所謂的迭代器特性，指的是迭代器的相應型別 (`associated types`)。當然，若要這個「特性萃取機」**traits** 能夠有效運作，每一個迭代器必須遵循約定，自行以巢狀型別定義 (`nested typedef`) 的方式定義出相應型別 (`associated types`)。這種一個約定，誰不遵守這個約定，誰就不能相容於 STL 這個大家庭。

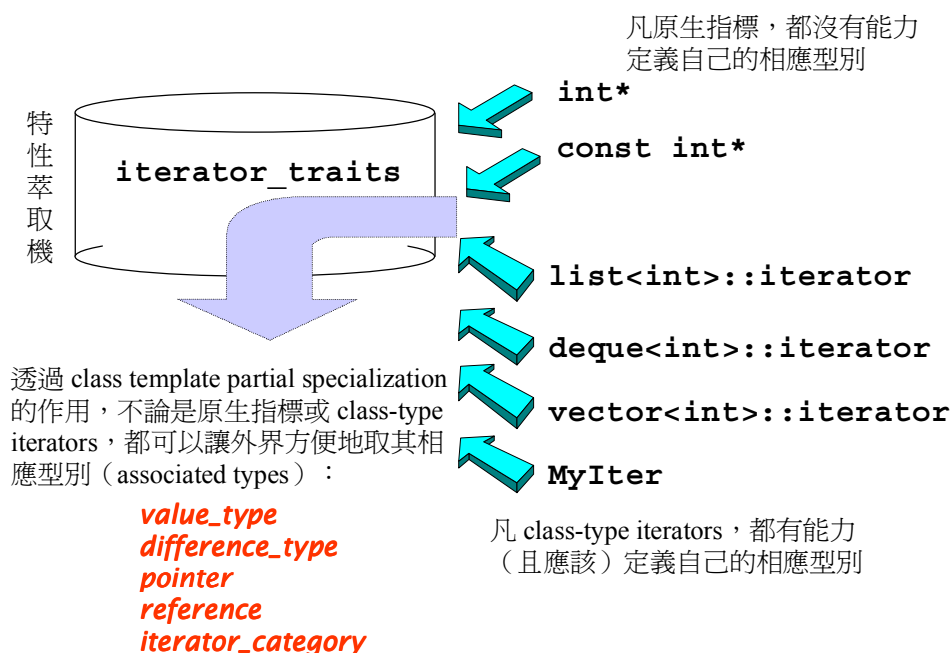


圖 3-1 traits 就像一台「特性萃取機」，榨取各個迭代器的特性（相應型別）。

根據經驗，最常用到的迭代器相應型別有五種：*value type*, *difference type*, *pointer*, *reference*, *iterator catagoly*。如果你希望你所開發的容器能與 STL 水乳交融，一定要為你的容器的迭代器定義這五種相應型別。「特性萃取機」traits 會很忠實地將原汁原味榨取出來：

```
template <class I>
struct iterator_traits {
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type value_type;
    typedef typename I::difference_type difference_type;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
};
```

iterator_traits 必須針對傳入之型別為 pointer 及 pointer-to-const 者，設計特化版本，稍後數節為你展示如何進行。

3.4.1 迭代器相係型別之一：value type

所謂 *value type*，是指迭代器所指物件的型別。任何一個打算與 STL 演算法有完美搭配的 class，都應該定義自己的 *value type* 巢狀型別，作法就像上節所述。

3.4.2 迭代器相係型別之二：difference type

difference type 用來表示兩個迭代器之間的距離，也因此，它可以用來表示一個容器的最大容量，因為對於連續空間的容器而言，頭尾之間的距離就是其最大容量。如果一個泛型演算法提供計數功能，例如 STL 的 `count()`，其傳回值就必須使用迭代器的 *difference type*：

```
template <class I, class T>
typename iterator_traits<I>::difference_type // 這一整行是函式回返型別
count(I first, I last, const T& value) {
    typename iterator_traits<I>::difference_type n = 0;
    for ( ; first != last; ++first)
        if (*first == value)
            ++n;
    return n;
}
```

針對相應型別 *difference type*，traits 的兩個（針對原生指標而寫的）特化版本如下，以 C++ 內建的 `ptrdiff_t`（定義於 `<cstdint>` 表頭檔）做為原生指標的 *difference type*：

```
template <class I>
struct iterator_traits {
    ...
    typedef typename I::difference_type difference_type;
};

// 針對原生指標而設計的「偏特化 (partial specialization)」版
template <class T>
struct iterator_traits<T*> {
    ...
    typedef ptrdiff_t difference_type;
};

// 針對原生的 pointer-to-const 而設計的「偏特化 (partial specialization)」版
template <class T>
struct iterator_traits<const T*> {
    ...
}
```

```
typedef ptrdiff_t    difference_type;
};
```

現在，任何時候當我們需要任何迭代器 *I* 的 *difference type*，可以這麼寫：

```
typename iterator_traits<I>::difference_type
```

3.4.3 迭代器相係型別之三：*reference type*

從「迭代器所指之物的內容是否允許改變」的角度觀之，迭代器分為兩種：不允許改變「所指物件之內容」者，稱為 *constant iterators*，例如 `const int* pci`；允許改變「所指物件之內容」者，稱為 *mutable iterators*，例如 `int* pi`。當我們對一個 *mutable iterators* 做提領動作時，獲得的不應該是個右值（*rvalue*），應該是個左值（*lvalue*），因為右值不允許賦值動作（*assignment*），左值才允許：

```
int* pi = new int(5);
const int* pci = new int(9);
*pi = 7; // 對 mutable iterator 做提領動作時，獲得的應該是個左值，允許賦值。
*pci = 1; // 這個動作不允許，因為 pci 是個 constant iterator，
          // 提領 pci 所得結果，是個右值，不允許被賦值。
```

在 C++ 中，函式如果要傳回左值，都是以 *by reference* 的方式進行，所以當 *p* 是個 *mutable iterators* 時，如果其 *value type* 是 *T*，那麼 **p* 的型別不應該是 *T*，應該是 *T&*。將此道理擴充，如果 *p* 是一個 *constant iterators*，其 *value type* 是 *T*，那麼 **p* 的型別不應該是 `const T`，而應該是 `const T&`。這裡所討論的 **p* 的型別，即所謂的 *reference type*。實作細節將在下一小節一併展示。

3.4.4 迭代器相係型別之四：*pointer type*

pointers 和 *references* 在 C++ 中有非常密切的關連。如果「傳回一個左值，令它代表 *p* 所指之物」是可能的，那麼「傳回一個左值，令它代表 *p* 所指之物的位址」也一定可以。也就是說我們能夠傳回一個 *pointer*，指向迭代器所指之物。

這些相應型別已在先前的 `ListIter` class 中出現過：

```
Item& operator*() const { return *ptr; }
Item* operator->() const { return ptr; }
```

`Item&` 便是 `ListIter` 的 *reference type* 而 `Item*` 便是其 *pointer type*。

現在我們把 *reference type* 和 *pointer type* 這兩個相應型別加入 **traits** 內：

```
template <class I>
struct iterator_traits {
    ...
    typedef typename I::pointer      pointer;
    typedef typename I::reference    reference;
};

// 針對原生指標而設計的「偏特化版 (partial specialization)」
template <class T>
struct iterator_traits<T*> {
    ...
    typedef T*      pointer;
    typedef T&      reference;
};

// 針對原生的 pointer-to-const 而設計的「偏特化版 (partial specialization)」
template <class T>
struct iterator_traits<const T*> {
    ...
    typedef const T*      pointer;
    typedef const T&      reference;
};
```

3.4.5 迭代器相應型別之五：iterator_category

最後一個（第五個）迭代器相應型別會引發較大規模的寫碼工程。在那之前，我必須先討論迭代器的分類。

根據移動特性與施行動作，迭代器被分為五類：

- *Input Iterator*：這種迭代器所指物件，不允許外界改變。唯讀（read only）。
- *Output Iterator*：唯寫（write only）。
- *Forward Iterator*：允許「寫入型」演算法（例如 `replace()`）在此種迭代器所形成的區間上做讀寫動作。
- *Bidirectional Iterator*：可雙向移動。某些演算法需要逆向走訪某個迭代器區間（例如逆向拷貝某範圍內的元素），就可以使用 *Bidirectional Iterators*。
- *Random Access Iterator*：前四種迭代器都只供應一部份指標算術能力（前三種支援 `operator++`，第四種再加上 `operator--`），第五種則涵蓋所有指標算術能力，包括 `p+n`，`p-n`，`p[n]`，`p1-p2`，`p1<p2`。

這些迭代器的分類與從屬關係，可以圖 3-2 表示。直線與箭頭代表的並非 C++ 的繼承關係，而是所謂 **concept**（概念）與 **refinement**（強化）的關係²。

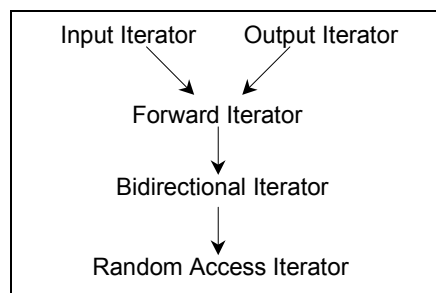


圖 3-2 迭代器的分類與從屬關係

設計演算法時，如果可能，我們儘量針對圖 3-2 中的某種迭代器提供一個明確定義，並針對更強化的某種迭代器提供另一種定義，這樣才能在不同情況下提供最大效率。研究 STL 的過程中，每一分每一秒我們都要念茲在茲，效率是個重要課題。假設有個演算法可接受 *Forward Iterator*，你以 *Random Access Iterator* 餵給它，它當然也會接受，因為一個 *Random Access Iterator* 必然是一個 *Forward Iterator*（見圖 3-2）。但是可用並不代表最佳！

以 `advanced()` 為例

拿 `advance()` 來說（這是許多演算法內部常用的一個函式），此函式有兩個參數，迭代器 `p` 和數值 `n`；函式內部將 `p` 累進 `n` 次（前進 `n` 距離）。下面有三份定義，一份針對 *Input Iterator*，一份針對 *Bidirectional Iterator*，另一份針對 *Random Access Iterator*。倒是沒有針對 *Forward Iterator* 而設計的版本，因為那和針對 *Input Iterator* 而設計的版本完全一致。

```

template <class InputIterator, class Distance>
void advance_II(InputIterator& i, Distance n)
{
    // 單向，逐一前進
    while (n--) ++i;           // 或寫 for ( ; n > 0; --n, ++i );
}
  
```

² **concept**（概念）與 **refinement**（強化），是架構 STL 的重要觀念，詳見 [Austern98]。

```

    }

    template <class BidirectionalIterator, class Distance>
    void advance_BI(BidirectionalIterator& i, Distance n)
    {
        // 雙向，逐一前進
        if (n >= 0)
            while (n--) ++i;          // 或寫 for ( ; n > 0; --n, ++i );
        else
            while (n++) --i;          // 或寫 for ( ; n < 0; ++n, --i );
    }

    template <class RandomAccessIterator, class Distance>
    void advance_RAI(RandomAccessIterator& i, Distance n)
    {
        // 雙向，跳躍前進
        i += n;
    }

```

現在，當程式呼叫 `advance()`，應該選用（呼叫）哪一份函式定義呢？如果選擇 `advance_II()`，對 *Random Access Iterator* 而言極度缺乏效率，原本 $O(1)$ 的操作竟成為 $O(N)$ 。如果選擇 `advance_RAI()`，則它無法接受 *Input Iterator*。我們需要將三者合一，下面是一種作法：

```

    template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n)
    {
        if (is_random_access_iterator(i))          // 此函式有待設計
            advance_RAI(i, n);
        else if (is_bidirectional_iterator(i))    // 此函式有待設計
            advance_BI(i, n);
        else
            advance_II(i, n);
    }

```

但是像這樣在執行時期才決定使用哪一個版本，會影響程式效率。最好能夠在編譯期就選擇正確的版本。多載化函式機制可以達成這個目標。

前述三個 `advance_xx()` 都有兩個函式參數，型別都未定（因為都是 `template` 參數）。為了令其同名，形成多載化函式，我們必須加上一個型別已確定的函式參數，使函式多載化機制得以有效運作起來。

設計考量如下：如果 **traits** 有能力萃取出迭代器的種類，我們便可利用這個「迭代器類型」相應型別做為 `advanced()` 的第三參數。這個相應型別一定必須是個

class type，不能只是數值號碼類的東西，因為編譯器需仰賴它（一個型別）來進行多載化決議程序（overloaded resolution）。下面定義五個 classes，代表五種迭代器類型：

```
// 五個 做為標記用的型別 (tag types)
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

這些 classes 只做為標記用，所以不需要任何成員。至於為什麼運用繼承機制，稍後再解釋。現在重新設計 __advance()（由於只在內部使用，所以函式名稱加上特定的前置字元），並加上第三參數，使它們形成多載化：

```
template <class InputIterator, class Distance>
inline void __advance(InputIterator& i, Distance n,
                     input_iterator_tag)
{
    // 單向，逐一前進
    while (n-- > 0) ++i;
}

// 這是一個單純的轉呼叫函式 (trivial forwarding function)。稍後討論如何免除之。
template <class ForwardIterator, class Distance>
inline void __advance(ForwardIterator& i, Distance n,
                     forward_iterator_tag)
{
    // 單純地進行轉呼叫 (forwarding)
    advance(i, n, input_iterator_tag());
}

template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator& i, Distance n,
                     bidirectional_iterator_tag)
{
    // 雙向，逐一前進
    if (n >= 0)
        while (n-- > 0) ++i;
    else
        while (n++ < 0) --i;
}

template <class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator& i, Distance n,
                     random_access_iterator_tag)
```

```
{
    // 雙向，跳躍前進
    i += n;
}
```

注意上述語法，每個 `__advance()` 的最後一個參數都只宣告型別，並未指定參數名稱，因為它純粹只是用來啟動多載化機制，函式之中根本不使用該參數。如果硬要加上參數名稱也可以，畫蛇添足罷了。

行進至此，還需要一個對外開放的上層控制介面，呼叫上述各個多載化的 `__advance()`。此一上層介面只需兩個參數，當它準備將工作轉給上述的 `__advance()` 時，才自行加上第三引數：迭代器類型。因此，這個上層函式必須有能力從它所獲得的迭代器中推導出其類型 — 這份工作自然是交給 **traits** 機制：

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n)
{
    __advance(i, n,
               iterator_traits<InputIterator>::iterator_category() );3
}
```

注意上述語法，`iterator_traits<Iterator>::iterator_category()` 將產生一個暫時物件（道理就像 `int()` 會產生一個 `int` 暫時物件一樣），其型別應該隸屬前述五個迭代器類型之一。然後，根據這個型別，編譯器才決定呼叫哪一個 `__advance()` 多載函式。

³ 關於此行，SGI STL `<stl_iterator.h>` 的源碼是：

```
__advance(i, n, iterator_category(i));
並另定義函式 iterator_category() 如下：
template <class I>
inline typename iterator_traits<I>::iterator_category
iterator_category(const I&) {
    typedef typename iterator_traits<I>::iterator_category category;
    return category();
}
```

綜合整理後原式即為：

```
__advance(i, n,
          iterator_traits<InputIterator>::iterator_category() );
```

因此，爲了滿足上述行爲，**traits** 必須再增加一個相應型別：

```
template <class I>
struct iterator_traits {
    ...
    typedef typename I::iterator_category    iterator_category;
};

// 針對原生指標而設計的「偏特化版 (partial specialization)」
template <class T>
struct iterator_traits<T*> {
    ...
    // 注意，原生指標是一種 Random Access Iterator
    typedef random_access_iterator_tag    iterator_category;
};

// 針對原生的 pointer-to-const 而設計的「偏特化版 (partial specialization)」
template <class T>
struct iterator_traits<const T*>
{
    ...
    // 注意，原生的 pointer-to-const 是一種 Random Access Iterator
    typedef random_access_iterator_tag    iterator_category;
};
```

任何一個迭代器，其類型永遠應該落在「該迭代器所隸屬之各種類型中，最強化的那個」。例如 `int*` 既是 *Random Access Iterator* 又是 *Bidirectional Iterator*，同時也是 *Forward Iterator*，而且也是 *Input Iterator*，那麼，其類型應該歸屬爲 *random_access_iterator_tag*。

你是否注意到 `advance()` 的 `template` 參數名稱取得好像不怎麼理想：

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n);
```

按說 `advanced()` 既然可以接受各種類型的迭代器，就不應將其型別參數命名爲 `InputIterator`。這其實是 STL 演算法的一個命名規則：以演算法所能接受之最低階迭代器類型，來爲其迭代器型別參數命名。

消除「單純轉呼叫函式」

以 `class` 來定義迭代器的各種分類標籤，不唯可以促成多載化機制的成功運作（使編譯器得以正確執行多載化決議程序，`overloaded resolution`），另一個好處是，

透過繼承，我們可以不必再寫「單純只做轉呼叫」的函式（例如前述的 `advance()` *ForwardIterator* 版）。為什麼能夠如此？考慮下面這個小例子，從其輸出結果可以看出端倪：

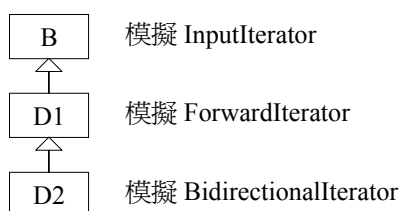


圖 3-3 類別繼承關係

```

// file: 3tag-test.cpp
// 模擬測試 tag types 繼承關係所帶來的影響。
#include <iostream>
using namespace std;

struct B { }; // B 可比擬為 InputIterator
struct D1 : public B { }; // D1 可比擬為 ForwardIterator
struct D2 : public D1 { }; // D2 可比擬為 BidirectionalIterator

template <class I>
func(I& p, B)
{ cout << "B version" << endl; }

template <class I>
func(I& p, D2)
{ cout << "D2 version" << endl; }

int main()
{
    int* p;
    func(p, B()); // 參數與引數完全吻合。輸出: "B version"
    func(p, D1()); // 參數與引數未能完全吻合；因繼承關係而自動轉呼叫。
                  // 輸出: "B version"
    func(p, D2()); // 參數與引數完全吻合。輸出: "D2 version"
}

```

以 `distance()` 為例

關於「迭代器類型標籤」的應用，以下再舉一例。`distance()` 也是常用的一個

迭代器操作函式，用來計算兩個迭代器之間的距離。針對不同的迭代器類型，它可以有不同的計算方式，帶來不同的效率。整個設計模式和前述的 `advance()` 如出一轍：

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
    iterator_traits<InputIterator>::difference_type n = 0;
    // 逐一累計距離
    while (first != last) {
        ++first; ++n;
    }
    return n;
}

template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
    // 直接計算差距
    return last - first;
}

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
    typedef typename iterator_traits<InputIterator>::iterator_category category;
    return __distance(first, last, category());
}
```

注意，`distance()` 可接受任何類型的迭代器；其 `template` 型別參數之所以命名為 `InputIterator`，是為了遵循 STL 演算法的命名規則：以演算法所能接受之最初級類型來為其迭代器型別參數命名。此外也請注意，由於迭代器類型之間存在著繼承關係，「轉呼叫 (*forwarding*)」的行為模式因此自然存在 — 這一點我已在前一節討論過。換句話說，當客端呼叫 `distance()` 並使用 *Output Iterators* 或 *Forward Iterators* 或 *Bidirectional Iterators*，統統都會轉呼叫 *Input Iterator* 版的那個 `__distance()` 函式。

3.5 std::iterator 的保證

為了符合規範，任何迭代器都應該提供五個巢狀相應型別，以利 **traits** 萃取，否

則便是自外於整個 STL 架構，可能無法與其他 STL 組件順利搭配。然而寫碼難免掛一漏萬，誰也不能保證不會有粗心大意的時候。如果能夠將事情簡化，就好多了。STL 提供了一個 **iterators class** 如下，如果每個新設計的迭代器都繼承自它，就保證符合 STL 所需之規範：

```
template <class Category,
          class T,
          class Distance = ptrdiff_t,
          class Pointer = T*,
          class Reference = T&>
struct iterator {
    typedef Category      iterator_category;
    typedef T            value_type;
    typedef Distance     difference_type;
    typedef Pointer      pointer;
    typedef Reference    reference;
};
```

iterator class 不含任何成員，純粹只是型別定義，所以繼承它並不會招致任何額外負擔。由於後三個參數皆有預設值，新的迭代器只需提供前兩個參數即可。先前 3.2 節土法煉鋼的 **ListIter**，如果改用正式規格，應該這麼寫：

```
template <class Item>
struct ListIter :
    public std::iterator<std::forward_iterator_tag, Item>
{ ... }
```

總結

設計適當的相應型別 (associated types)，是迭代器的責任。設計適當的迭代器，則是容器的責任。唯容器本身，才知道該設計出怎樣的迭代器來走訪自己，並執行迭代器該有的各種行為（前進、後退、取值、取用成員...）。至於演算法，完全可以獨立於容器和迭代器之外自行發展，只要設計時以迭代器為對外介面就行。

traits 編程技法，大量運用於 STL 實作品中。它利用「巢狀型別」的寫碼技巧與編譯器的 **template** 引數推導功能，補強 C++ 未能提供的關於型別認證方面的能力，補強 C++ 不為強型 (strong typed) 語言的遺憾。瞭解 **traits** 編程技法，就像獲得「芝麻開門」口訣一樣，從此得以一窺 STL 源碼堂奧。

3.6 iterator 源碼完整重列

由於討論次序的緣故，先前所列的源碼切割散落，有點凌亂。以下重新列出 SGI STL `<stl_iterator.h>` 表頭檔內與本章相關的程式碼。該表頭檔還有其他內容，是關於 `iostream iterators`、`inserter iterators` 以及 `reverse iterators` 的實作，將於第 8 章討論。

```
// 節錄自 SGI STL <stl_iterator.h>
// 五種迭代器類型
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};

// 為避免寫碼時掛一漏萬，自行開發的迭代器最好繼承自下面這個 std::iterator
template <class Category, class T, class Distance = ptrdiff_t,
         class Pointer = T*, class Reference = T&>
struct iterator {
    typedef Category      iterator_category;
    typedef T             value_type;
    typedef Distance      difference_type;
    typedef Pointer       pointer;
    typedef Reference     reference;
};

// 「榨汁機」traits
template <class Iterator>
struct iterator_traits {
    typedef typename Iterator::iterator_category  iterator_category;
    typedef typename Iterator::value_type         value_type;
    typedef typename Iterator::difference_type    difference_type;
    typedef typename Iterator::pointer            pointer;
    typedef typename Iterator::reference          reference;
};

// 針對原生指標（native pointer）而設計的 traits 偏特化版。
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag  iterator_category;
    typedef T                           value_type;
    typedef ptrdiff_t                   difference_type;
    typedef T*                          pointer;
    typedef T&                          reference;
};
```

// 針對原生之 `pointer-to-const` 而設計的 traits 偏特化版。

```
template <class T>
struct iterator_traits<const T*> {
    typedef random_access_iterator_tag    iterator_category;
    typedef T                            value_type;
    typedef ptrdiff_t                    difference_type;
    typedef const T*                    pointer;
    typedef const T&                    reference;
};
```

// 這個函式可以很方便地決定某個迭代器的類型 (category)

```
template <class Iterator>
inline typename iterator_traits<Iterator>::iterator_category
iterator_category(const Iterator&) {
    typedef typename iterator_traits<Iterator>::iterator_category category;
    return category();
}
```

// 這個函式可以很方便地決定某個迭代器的 distance type

```
template <class Iterator>
inline typename iterator_traits<Iterator>::difference_type*
distance_type(const Iterator&) {
    return static_cast<typename iterator_traits<Iterator>::difference_type*>(0);
}
```

// 這個函式可以很方便地決定某個迭代器的 value type

```
template <class Iterator>
inline typename iterator_traits<Iterator>::value_type*
value_type(const Iterator&) {
    return static_cast<typename iterator_traits<Iterator>::value_type*>(0);
}
```

// 以下是整組 distance 函式

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
    iterator_traits<InputIterator>::difference_type n = 0;
    while (first != last) {
        ++first; ++n;
    }
    return n;
}
```

```
template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
    return last - first;
}
```

```

}

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
    typedef typename
        iterator_traits<InputIterator>::iterator_category category;
    return __distance(first, last, category());
}

// 以下是整組 advance 函式
template <class InputIterator, class Distance>
inline void __advance(InputIterator& i, Distance n,
                     input_iterator_tag) {
    while (n-->0) ++i;
}

template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator& i, Distance n,
                     bidirectional_iterator_tag) {
    if (n >= 0)
        while (n-->0) ++i;
    else
        while (n++<0) --i;
}

template <class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator& i, Distance n,
                     random_access_iterator_tag) {
    i += n;
}

template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n) {
    __advance(i, n, iterator_category(i));
}

```

3.7 SGI STL 的私房菜：__type_traits

traits 編程技法很棒，適度彌補了 C++ 語言本身的不足。STL 只對迭代器加以規範，制定出 **iterator_traits** 這樣的東西。SGI 把這種技法進一步擴大到迭代器以外的世界，於是有了所謂的 **__type_traits**。雙底線前綴詞意指這是 SGI STL 內部所用的東西，不在 STL 標準規範之內。

iterator_traits 負責萃取迭代器的特性，**__type_traits** 則負責萃取型別

(type) 的特性。此處我們所關注的型別特性是指：這個型別是否具備 non-trivial default ctor？是否具備 non-trivial copy ctor？是否具備 non-trivial assignment operator？是否具備 non-trivial dtor？如果答案是否定的，我們在對這個型別進行建構、解構、拷貝、賦值等動作時，就可以採用最有效率的措施（例如根本不喚起尸位素餐的那些 constructor, destructor），而採用記憶體直接處理動作如 malloc()、memcpy() 等等，獲得最高效率。這對於大規模而動作頻繁的容器，有著顯著的效率提昇⁴。

定義於 SGI <type_traits.h> 中的 `__type_traits`，提供了一種機制，允許針對不同的型別屬性 (type attributes)，在編譯時期完成函式派送決定 (function dispatch)。這對於撰寫 template 很有幫助，例如，當我們準備對一個「元素型別未知」的陣列執行 copy 動作時，如果我們能事先知道其元素型別是否有一個 trivial copy constructor，便能夠幫助我們決定是否可使用快速的 memcpy() 或 memmove()。

從 `iterator_traits` 得來的經驗，我們希望，程式之中可以這樣運用 `__type_traits<T>`，T 代表任意型別：

```
__type_traits<T>::has_trivial_default_constructor
__type_traits<T>::has_trivial_copy_constructor
__type_traits<T>::has_trivial_assignment_operator
__type_traits<T>::has_trivial_destructor
__type_traits<T>::is_POD_type           // POD : Plain Old Data
```

我們希望上述式子回應我們「真」或「假」（以便我們決定採取什麼策略），但其結果不應該只是個 bool 值，應該是個有著真/假性質的「物件」，因為我們希望利用其回應結果來進行引數推導，而編譯器只有面對 class object 形式的引數，才會做引數推導。為此，上述式子應該傳回這樣的東西：

```
struct __true_type { };
struct __false_type { };
```

這兩個空白 classes 沒有任何成員，不會帶來額外負擔，卻又能夠標示真假，滿足我們所需。

⁴ C++ *Type Traits*, by John Maddock and Steve Cleary, DDJ 2000/10 提了一些測試數據。

爲了達成上述五個式子，__type_traits 內必須定義一些 typedefs，其值不是 __true_type 就是 __false_type。下面是 SGI 的作法：

```
template <class type>
struct __type_traits
{
    typedef __true_type      this_dummy_member_must_be_first;
    /* 不要移除這個成員。它通知「有能力自動將 __type_traits 特化」
       的編譯器說，我們現在所看到的這個 __type_traits template 是特
       殊的。這是爲了確保萬一編譯器也使用一個名爲 __type_traits 而其
       實與此處定義並無任何關聯的 template 時，所有事情都仍將順利運作。
       */

    /* 以下條件應被遵守，因爲編譯器有可能自動爲各型別產生專屬的 __type_traits
       特化版本：
       - 你可以重新排列以下的成員次序
       - 你可以移除以下任何成員
       - 絕對不可以將以下成員重新命名而卻沒有改變編譯器中的對應名稱
       - 新加入的成員會被視爲一般成員，除非你在編譯器中加上適當支援。*/

    typedef __false_type     has_trivial_default_constructor;
    typedef __false_type     has_trivial_copy_constructor;
    typedef __false_type     has_trivial_assignment_operator;
    typedef __false_type     has_trivial_destructor;
    typedef __false_type     is_POD_type;
};
```

爲什麼 SGI 把所有巢狀型別都定義爲 __false_type 呢？是的，SGI 定義出最保守的值，然後（稍後可見）再針對每一個純量型別（scalar types）設計適當的 __type_traits 特化版本，這樣就解決了問題。

上述 __type_traits 可以接受任何型別的引數，五個 typedefs 將經由以下管道獲得實值：

- 一般具現體（general instantiation），內含對所有型別都必定有效的保守值。上述各個 has_trivial_xxx 型別都被定義爲 __false_type，就是對所有型別都必定有效的保守值。
- 經過宣告的特化版本，例如 <type_traits.h> 內對所有 C++ 純量型別（scalar types）提供了對映的特化宣告。稍後展示。
- 某些編譯器（如 Silicon Graphics N32 和 N64 編譯器）會自動爲所有型別提供適當的特化版本。（這真是了不起的技術。不過我對其精確程度存疑）

以下便是 `<type_traits.h>` 對所有 C++ 純量型別所定義的 `__type_traits` 特化版本。這些定義對於內建有 `__types_traits` 支援能力的編譯器（例如 Silicon Graphics N32 和 N64）並無傷害，對於無該等支援能力的編譯器而言，則屬必要。

```
/* 以下針對 C++ 基本型別 char, signed char, unsigned char, short,
unsigned short, int, unsigned int, long, unsigned long, float, double,
long double 提供特化版本。注意，每一個成員的值都是 __true_type，表示這些
型別都可採用最快速方式（例如 memcpy）來進行拷貝（copy）或賦值（assign）動
作。*/

// 注意，SGI STL <stl_config.h> 將以下出現的 __STL_TEMPLATE_NULL
// 定義為 template<>，見 1.9.1 節，是所謂的
// class template explicit specialization

__STL_TEMPLATE_NULL struct __type_traits<char> {
    typedef __true_type      has_trivial_default_constructor;
    typedef __true_type      has_trivial_copy_constructor;
    typedef __true_type      has_trivial_assignment_operator;
    typedef __true_type      has_trivial_destructor;
    typedef __true_type      is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<signed char> {
    typedef __true_type      has_trivial_default_constructor;
    typedef __true_type      has_trivial_copy_constructor;
    typedef __true_type      has_trivial_assignment_operator;
    typedef __true_type      has_trivial_destructor;
    typedef __true_type      is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned char> {
    typedef __true_type      has_trivial_default_constructor;
    typedef __true_type      has_trivial_copy_constructor;
    typedef __true_type      has_trivial_assignment_operator;
    typedef __true_type      has_trivial_destructor;
    typedef __true_type      is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<short> {
    typedef __true_type      has_trivial_default_constructor;
    typedef __true_type      has_trivial_copy_constructor;
    typedef __true_type      has_trivial_assignment_operator;
    typedef __true_type      has_trivial_destructor;
    typedef __true_type      is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned short> {
    typedef __true_type      has_trivial_default_constructor;
```

```

    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<int> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned int> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<long> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned long> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<float> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<double> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;

```

```

typedef __true_type    has_trivial_assignment_operator;
typedef __true_type    has_trivial_destructor;
typedef __true_type    is_POD_type;
};

```

```

__STL_TEMPLATE_NULL struct __type_traits<long double> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

```

// 注意，以下針對原生指標設計 __type_traits 偏特化版本。

// 原生指標亦被視為一種純量型別。

```

template <class T>
struct __type_traits<T*> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type    has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};

```

__types_traits 在 SGI STL 中的應用很廣。下面我舉幾個實例。第一個例子是出現於本書 2.3.3 節的 uninitialized_fill_n() 全域函式：

```

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));
}

```

此函式以 x 為藍本，自迭代器 first 開始建構 n 個元素。為求取最大效率，首先以 value_type() (3.6 節) 萃取出迭代器 first 的 value type，再利用 __type_traits 判斷該型別是否為 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*)
{
    typedef typename __type_traits<T1>::is_POD_type is_POD;
    return __uninitialized_fill_n_aux(first, n, x, is_POD());
}

```

以下就「是否為 POD 型別」採取最適當的措施：

```

// 如果不是 POD 型別，就會派送 (dispatch) 到這裡
template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                          const T& x, __false_type) {
    ForwardIterator cur = first;
    // 為求閱讀順暢簡化，以下將原本有的異常處理 (exception handling) 去除。
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x);    // 見 2.2.3 節
    return cur;
}

// 如果是 POD 型別，就會派送 (dispatch) 到這裡。下兩行是原檔所附註解。
// 如果 copy construction 等同於 assignment，而且有 trivial destructor，
// 以下就有效。
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，如下所示。
}

// 以下是定義於 <stl_algobase.h> 中的 fill_n()
template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value) {
    for ( ; n > 0; --n, ++first)
        *first = value;
    return first;
}

```

第二個例子是負責物件解構的 `destroy()` 全域函式。此函式之源碼及解說在 2.2.3 節有完整的說明。

第三個例子是出現於本書第 6 章的 `copy()` 全域函式（泛型演算法之一）。這個函式有非常多的特化（specialization）與強化（refinement）版本，殫精竭慮，全都是為了效率考慮，希望在適當的情況下採用最「雷霆萬鈞」的手段。最基本的想法是這樣：

```

// 拷貝一個陣列，其元素為任意型別，視情況採用最有效率的拷貝手段。
template <class T> inline void copy(T* source, T* destination, int n) {
    copy(source, destination, n,
         typename __type_traits<T>::has_trivial_copy_constructor());
}

// 拷貝一個陣列，其元素型別擁有 non-trivial copy constructors。

```

```

template <class T> void copy(T* source, T* destination, int n,
                           __false_type)
{ ... }

// 拷貝一個陣列，其元素型別擁有 trivial copy constructors。
// 可借助 memcpy() 完成工作
template <class T> void copy(T* source, T* destination, int n,
                           __true_type)
{ ... }

```

以上只是針對「函式參數為原生指標」的情況而做的設計。第 6 章的 `copy()` 演算法是個泛型版本，情況又複雜許多。詳見 6.4.3 節。

請注意，`<type_traits.h>` 並未像其他許多 SGI STL 表頭檔有這樣的聲明：

```

/* NOTE: This is an internal header file, included by other STL headers.
 * You should not attempt to use it directly.
 */

```

因此如果你是 SGI STL 的使用者，你可以在自己的程式中充份運用這個 `__type_traits`。假設我自行定義了一個 `Shape` class，`__type_traits` 會對它產生什麼效應？如果編譯器夠厲害（例如 Silicon Graphics 的 N32 和 N64 編譯器），你會發現，`__type_traits` 針對 `Shape` 萃取出來的每一個特性，其結果將取決於我的 `Shape` 是否有 `trivial default ctor` 或 `trivial copy ctor` 或 `trivial assignment operator` 或 `trivial dtor` 而定。但對大部份缺乏這種特異功能的編譯器而言，`__type_traits` 針對 `Shape` 萃取出來的每一個特性都是 `__false_type`，即使 `Shape` 是個 POD 型別。這樣的結果當然過於保守，但是別無選擇，除非我針對 `Shape`，自行設計一個 `__type_traits` 特化版本，明白地告訴編譯器以下事實（舉例）：

```

template<> struct __type_traits<Shape> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __false_type   has_trivial_copy_constructor;
    typedef __false_type   has_trivial_assignment_operator;
    typedef __false_type   has_trivial_destructor;
    typedef __false_type   is_POD_type;
};

```

究竟一個 class 什麼時候該有自己的 `non-trivial default constructor`, `non-trivial copy constructor`, `non-trivial assignment operator`, `non-trivial destructor` 呢？一個簡單的判

斷準則是：如果 `class` 內含指標成員，並且對它進行記憶體動態配置，那麼這個 `class` 就需要實作出自己的 `non-trivial-xxx`⁵。

即使你無法全面針對你自己定義的型別，設計 `__type_traits` 特化版本，無論如何，至少，有了這個 `__type_traits` 之後，當我們設計新的泛型演算法時，面對 C++ 純量型別，便有足夠的資訊決定採用最有效的拷貝動作或賦值動作 — 因為每一個純量型別都有對應的 `__type_traits` 特化版本，其中每一個 `typedef` 的值都是 `__true_type`。

⁵ 請參考 [Meyers98] 條款 11: *Declare a copy constructor and an assignment operator for classes with dynamically allocated memory*，以及條款 45: *Know what functions C++ silently writes and calls*.

4

序列式容器
sequence containers

4.1 容器的概觀與分類

容器，置物之所也。

研究資料的特定排列方式，以利搜尋或排序或其他特殊目的，這一專門學科我們稱為資料結構（Data Structures）。大學資訊相關教育裡頭，與編程最有直接關係的科目，首推資料結構與演算法（Algorithms）。幾乎可以說，任何特定的資料結構都是為了實現某種特定的演算法。STL 容器即是將運用最廣的一些資料結構實作出來（圖 4-1）。未來，在每五年召開一次的 C++ 標準委員會中，STL 容器的數量還有可能增加。

眾所周知，常用的資料結構不外乎 array（陣列）、list（串列）、tree（樹）、stack（堆疊）、queue（佇列）、hash table（雜湊表）、set（集合）、map（映射表）……等等。根據「資料在容器中的排列」特性，這些資料結構分為序列式（sequence）和關聯式（associative）兩種。本章探討序列式容器，下一章探討關聯式容器。

容器是大多數人對 STL 的第一印象，這說明了容器的好用與受歡迎。容器也是許多人對 STL 的唯一印象，這說明了還有多少人利器（STL）在手而未能善用。

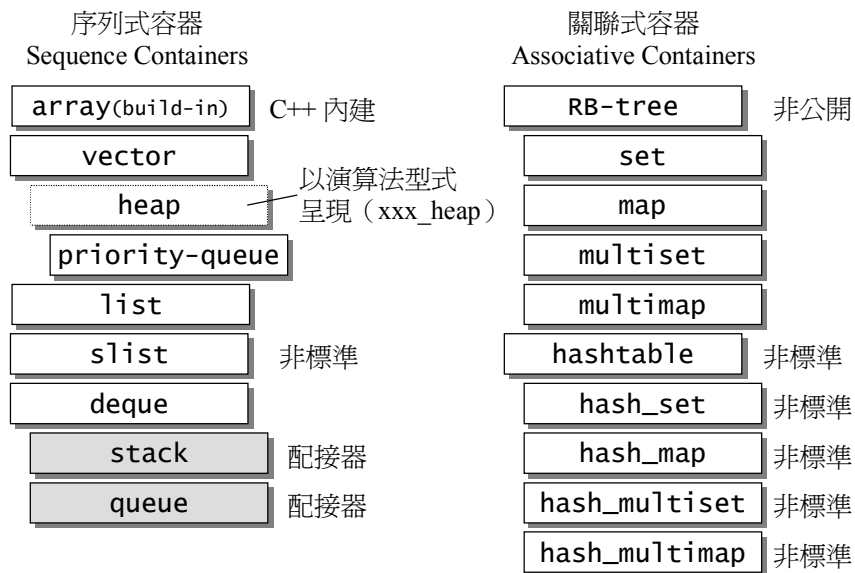


圖 4-1 SGI STL 的各種容器。本圖以內縮方式來表達基層與衍生層的關係。這裡所謂的衍生，並非繼承 (inheritance) 關係，而是內含 (containment) 關係。例如 heap 內含一個 vector，priority-queue 內含一個 heap，stack 和 queue 都含一個 deque，set/map/multiset/multimap 都內含一個 RB-tree，hash_x 都內含一個 hashtable。

4.1.1 序列式容器 (sequential containers)

所謂序列式容器，其中的元素都可序 (ordered)，但未排序 (sorted)。C++ 語言本身提供了一個序列式容器 array，STL 另外再提供 vector, list, deque, stack, queue, priority-queue 等等序列式容器。其中 stack 和 queue 由於只是將 deque 改頭換面而成，技術上被歸類為一種配接器 (adapter)，但我仍把它們放在本章討論。

本章將帶你仔細看過各種序列式容器的關鍵實作細節。

4.2 vector

4.2.1 vector 概述

vector 的資料安排以及操作方式，與 **array** 非常像似。兩者的唯一差別在於空間的運用彈性。**array** 是靜態空間，一旦配置了就不能改變；要換個大（或小）一點的房子，可以，一切細瑣得由客端自己來：首先配置一塊新空間，然後將元素從舊址一一搬往新址，然後再把原來的空間釋還給系統。**vector** 是動態空間，隨著元素的加入，它的內部機制會自行擴充空間以容納新元素。因此，**vector** 的運用對於記憶體體的樽節與運用彈性有很大的幫助，我們再也不必因為害怕空間不足而一開始就要求一個大塊頭 **array** 了，我們可以安心使用 **vector**，吃多少用多少。

vector 的實作技術，關鍵在於其對大小的控制以及重新配置時的資料搬移效率。一旦 **vector** 舊有空間滿載，如果客端每新增一個元素，**vector** 內部只是擴充一個元素的空間，實為不智，因為所謂擴充空間（不論多大），一如稍早所說，是「配置新空間 / 資料搬移 / 釋還舊空間」的大工程，時間成本很高，應該加入某種未雨綢繆的考量。稍後我們便可看到 SGI **vector** 的空間配置策略。

4.2.2 vector 定義式擄取

以下是 **vector** 定義式的源碼摘錄。雖然 STL 規定，欲使用 **vector** 者必須先含入 `<vector>`，但 SGI STL 將 **vector** 實作於更底層的 `<stl_vector.h>`。

```
// alloc 是 SGI STL 的空間配置器，見第二章。
template <class T, class Alloc = alloc>
class vector {
public:
    // vector 的巢狀型別定義
    typedef T          value_type;
    typedef value_type* pointer;
    typedef value_type* iterator;
    typedef value_type& reference;
    typedef size_t      size_type;
    typedef ptrdiff_t   difference_type;

protected:
```

```

// 以下，simple_alloc 是 SGI STL 的空間配置器，見 2.2.4 節。
typedef simple_alloc<value_type, Alloc> data_allocator;

iterator start;           // 表示目前使用空間的頭
iterator finish;          // 表示目前使用空間的尾
iterator end_of_storage;  // 表示目前可用空間的尾

void insert_aux(iterator position, const T& x);
void deallocate() {
    if (start)
        data_allocator::deallocate(start, end_of_storage - start);
}

void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);
    finish = start + n;
    end_of_storage = finish;
}

public:
    iterator begin() { return start; }
    iterator end() { return finish; }
    size_type size() const { return size_type(end() - begin()); }
    size_type capacity() const {
        return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }
    reference operator[](size_type n) { return *(begin() + n); }

    vector() : start(0), finish(0), end_of_storage(0) {}
    vector(size_type n, const T& value) { fill_initialize(n, value); }
    vector(int n, const T& value) { fill_initialize(n, value); }
    vector(long n, const T& value) { fill_initialize(n, value); }
    explicit vector(size_type n) { fill_initialize(n, T()); }

    ~vector()
    {
        destroy(start, finish);           // 全域函式，見 2.2.3 節。
        deallocate();                     // 這是 vector 的一個 member function
    }
    reference front() { return *begin(); } // 第一個元素
    reference back() { return *(end() - 1); } // 最後一個元素
    void push_back(const T& x) {           // 將元素安插至最尾端
        if (finish != end_of_storage) {
            construct(finish, x);         // 全域函式，見 2.2.3 節。
            ++finish;
        }
        else
            insert_aux(end(), x);         // 這是 vector 的一個 member function
    }

    void pop_back() {                     // 將最尾端元素取出

```

```

    --finish;
    destroy(finish);           // 全域函式，見 2.2.3 節。
}

iterator erase(iterator position) {    // 清除某位置上的元素
    if (position + 1 != end())
        copy(position + 1, finish, position);    // 後續元素往前搬移
    --finish;
    destroy(finish);           // 全域函式，見 2.2.3 節。
    return position;
}

void resize(size_type new_size, const T& x) {
    if (new_size < size())
        erase(begin() + new_size, end());
    else
        insert(end(), new_size - size(), x);
}

void resize(size_type new_size) { resize(new_size, T()); }
void clear() { erase(begin(), end()); }

protected:
    // 配置空間並填滿內容
    iterator allocate_and_fill(size_type n, const T& x) {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x); // 全域函式，見 2.3 節
        return result;
    }
}

```

4.2.3 vector 的迭代器

vector 維護的是一個連續線性空間，所以不論其元素型別為何，原生指標都可以做為 **vector** 的迭代器而滿足所有必要條件，因為 **vector** 迭代器所需要的操作行為如 `operator*`, `operator->`, `operator++`, `operator--`, `operator+`, `operator-`, `operator+=`, `operator-=`，原生指標天生就具備。**vector** 支援隨機存取，而原生指標正有著這樣的能力。所以，**vector** 提供的是 *Random Access Iterators*。

```

template <class T, class Alloc = alloc>
class vector {
public:
    typedef T          value_type;
    typedef value_type* iterator;    // vector 的迭代器是原生指標
    ...
};

```

根據上述定義，如果客端寫出這樣的碼：

```
vector<int>::iterator ivite;
vector<Shape>::iterator svite;
```

ivite 的型別其實就是 `int*`，svite 的型別其實就是 `Shape*`。

4.2.4 vector 的資料結構

`vector` 所採用的資料結構非常簡單：線性連續空間。它以兩個迭代器 `start` 和 `finish` 分別指向配置得來的連續空間中目前已被使用的範圍，並以迭代器 `end_of_storage` 指向整塊連續空間（含備用空間）的尾端：

```
template <class T, class Alloc = alloc>
class vector {
...
protected:
    iterator start;           // 表示目前使用空間的頭
    iterator finish;          // 表示目前使用空間的尾
    iterator end_of_storage;  // 表示目前可用空間的尾
...
};
```

爲了降低空間配置時的速度成本，`vector` 實際配置的大小可能比客端需求量更大一些，以備將來可能的擴充。這便是容量 (capacity) 的觀念。換句話說一個 `vector` 的容量永遠大於或等於其大小。一旦容量等於大小，便是滿載，下次再有新增元素，整個 `vector` 就得另覓居所。見圖 4-2。

運用 `start`、`finish`、`end_of_storage` 三個迭代器，便可輕易提供首尾標示、大小、容量、空容器判斷、註標 (`[]`) 運算子、最前端元素值、最後端元素值…等機能：

```
template <class T, class Alloc = alloc>
class vector {
...
public:
    iterator begin() { return start; }
    iterator end() { return finish; }
    size_type size() const { return size_type(end() - begin()); }
    size_type capacity() const {
        return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }
    reference operator[](size_type n) { return *(begin() + n); }

    reference front() { return *begin(); }
```

```
reference back() { return *(end() - 1); }
...
};
```

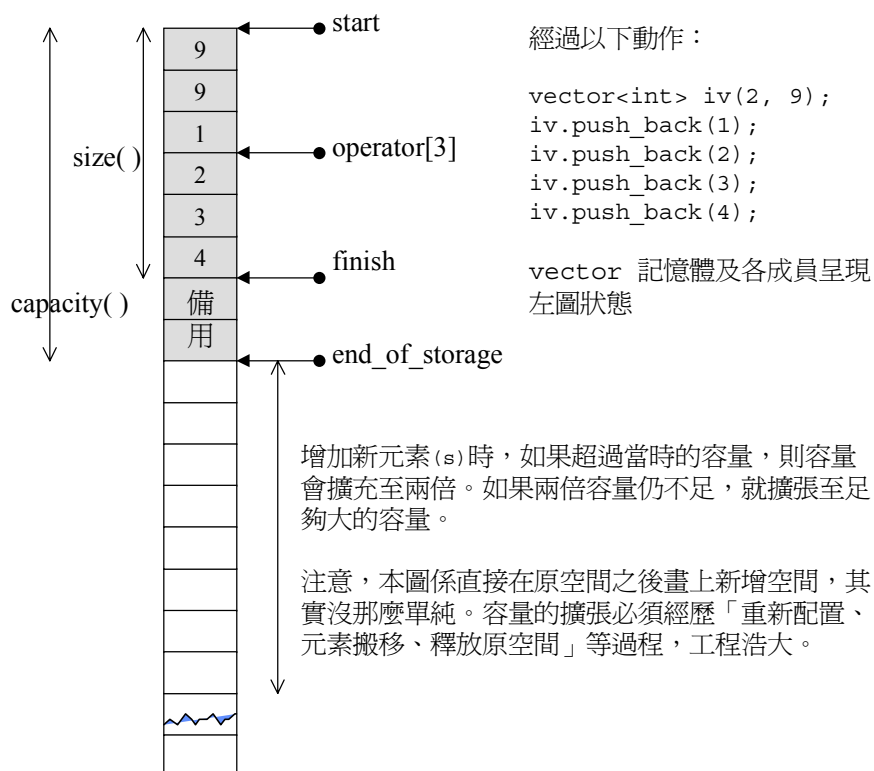


圖 4-2 vector 示意圖

4.2.5 vector 的建構與記憶體管理：constructor, push_back

千頭萬緒該如何說起？以客端程式碼為引導，觀察其所得結果並實證源碼，是個良好的學習路徑。下面是個小小的測試程式，我的觀察重點在建構的方式、元素的添加，以及大小、容量的變化：

```
// filename : 4vector-test.cpp
#include <vector>
#include <iostream>
#include <algorithm>
```

```

using namespace std;

int main()
{
    int i;
    vector<int> iv(2,9);
    cout << "size=" << iv.size() << endl;           // size=2
    cout << "capacity=" << iv.capacity() << endl;     // capacity=2

    iv.push_back(1);
    cout << "size=" << iv.size() << endl;           // size=3
    cout << "capacity=" << iv.capacity() << endl;     // capacity=4

    iv.push_back(2);
    cout << "size=" << iv.size() << endl;           // size=4
    cout << "capacity=" << iv.capacity() << endl;     // capacity=4

    iv.push_back(3);
    cout << "size=" << iv.size() << endl;           // size=5
    cout << "capacity=" << iv.capacity() << endl;     // capacity=8

    iv.push_back(4);
    cout << "size=" << iv.size() << endl;           // size=6
    cout << "capacity=" << iv.capacity() << endl;     // capacity=8

    for(i=0; i<iv.size(); ++i)
        cout << iv[i] << ' ';                     // 9 9 1 2 3 4
    cout << endl;

    iv.push_back(5);

    cout << "size=" << iv.size() << endl;           // size=7
    cout << "capacity=" << iv.capacity() << endl;     // capacity=8
    for(i=0; i<iv.size(); ++i)
        cout << iv[i] << ' ';                     // 9 9 1 2 3 4 5
    cout << endl;

    iv.pop_back();
    iv.pop_back();
    cout << "size=" << iv.size() << endl;           // size=5
    cout << "capacity=" << iv.capacity() << endl;     // capacity=8

    iv.pop_back();
    cout << "size=" << iv.size() << endl;           // size=4
    cout << "capacity=" << iv.capacity() << endl;     // capacity=8

    vector<int>::iterator ivite = find(iv.begin(), iv.end(), 1);
    if (ivite) iv.erase(ivite);

```

```

cout << "size=" << iv.size() << endl;           // size=3
cout << "capacity=" << iv.capacity() << endl;    // capacity=8
for(i=0; i<iv.size(); ++i)
    cout << iv[i] << ' ';                       // 9 9 2
cout << endl;

ite = find(ivec.begin(), ivec.end(), 2);
if (ite) ivec.insert(ite,3,7);

cout << "size=" << iv.size() << endl;           // size=6
cout << "capacity=" << iv.capacity() << endl;    // capacity=8
for(int i=0; i<ivec.size(); ++i)
    cout << ivec[i] << ' ';                     // 9 9 7 7 7 2
cout << endl;

iv.clear();
cout << "size=" << iv.size() << endl;           // size=0
cout << "capacity=" << iv.capacity() << endl;    // capacity=8
}

```

vector 預設使用 `alloc` (第二章) 做為空間配置器，並據此另外定義了一個 `data_allocator`，為的是更方便以元素大小為配置單位：

```

template <class T, class Alloc = alloc>
class vector {
protected:
    // simple\_alloc<> 見 2.2.4 節
    typedef simple_alloc<value_type, Alloc> data_allocator;
    ...
};

```

於是，`data_allocator::allocate(n)` 表示配置 `n` 個元素空間。

vector 提供許多 constructors，其中一個允許我們指定空間大小及初值：

```

// 建構式，允許指定 vector 大小 n 和初值 value
vector(size_type n, const T& value) { fill_initialize(n, value); }

// 充填並予初始化
void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);
    finish = start + n;
    end_of_storage = finish;
}

// 配置而後充填
iterator allocate_and_fill(size_type n, const T& x) {

```



```

        iterator result = data_allocator::allocate(n); // 配置 n 個元素空間
        uninitialized_fill_n(result, n, x); // 全域函式，見 2.3 節
        return result;
    }

```

`uninitialized_fill_n()` 會根據第一參數的型別特性 (`type traits`, 3.7 節)，決定使用演算法 `fill_n()` 或反覆呼叫 `construct()` 來完成任務 (見 2.3 節描述)。

當我們以 `push_back()` 將新元素安插於 `vector` 尾端，該函式首先檢查是否還有備用空間？如果有就直接在備用空間上建構元素，並調整迭代器 `finish`，使 `vector` 變大。如果沒有備用空間了，就擴充空間 (重新配置、搬移資料、釋放原空間)：

```

void push_back(const T& x) {
    if (finish != end_of_storage) { // 還有備用空間
        construct(finish, x); // 全域函式，見 2.2.3 節。
        ++finish; // 調整水位高度
    }
    else // 已無備用空間
        insert_aux(end(), x); // vector member function，見以下列表
}

template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
    if (finish != end_of_storage) { // 還有備用空間
        // 在備用空間起始處建構一個元素，並以 vector 最後一個元素值為其初值。
        construct(finish, *(finish - 1));
        // 調整水位。
        ++finish;
        T x_copy = x;
        copy_backward(position, finish - 2, finish - 1);
        *position = x_copy;
    }
    else { // 已無備用空間
        const size_type old_size = size();
        const size_type len = old_size != 0 ? 2 * old_size : 1;
        // 以上配置原則：如果原大小為 0，則配置 1 (個元素大小)；
        // 如果原大小不為 0，則配置原大小的兩倍，
        // 前半段用來放置原資料，後半段準備用來放置新資料。

        iterator new_start = data_allocator::allocate(len); // 實際配置
        iterator new_finish = new_start;
        try {
            // 將原 vector 的內容拷貝到新 vector。
            new_finish = uninitialized_copy(start, position, new_start);
            // 為新元素設定初值 x
            construct(new_finish, x);
            // 調整水位。

```

```

    ++new_finish;
    // 將原 vector 的備用空間中的內容也忠實拷貝過來（侯捷疑惑：啥用途？）
    new_finish = uninitialized_copy(position, finish, new_finish);
}
catch(...) {
    // "commit or rollback" semantics.
    destroy(new_start, new_finish);
    data_allocator::deallocate(new_start, len);
    throw;
}

// 解構並釋放原 vector
destroy(begin(), end());
deallocate();

// 調整迭代器，指向新 vector
start = new_start;
finish = new_finish;
end_of_storage = new_start + len;
}
}

```

注意，所謂動態增加大小，並不是在原空間之後接續新空間（因為無法保證原空間之後尚有可供配置的空間），而是以原大小的兩倍另外配置一塊較大空間，然後將原內容拷貝過來，然後才開始在原內容之後建構新元素，並釋放原空間。因此，對 `vector` 的任何操作，一旦引起空間重新配置，指向原 `vector` 的所有迭代器就都失效了。這是程式員易犯的一個錯誤，務需小心。

4.2.6 vector 的元素操作：pop_back, erase, clear, insert

`vector` 所提供的元素操作動作很多，無法在有限篇幅中一一講解——其實也沒有這種必要。為搭配先前對空間配置的討論，我挑選數個相關函式做為解說對象。這些函式也出現在先前的測試程式中。

```

// 將尾端元素拿掉，並調整大小。
void pop_back() {
    --finish;                // 將尾端標記往前移一格，表示將放棄尾端元素。
    destroy(finish);         // destroy 是全域函式，見第 2 章
}

// 清除 [first,last) 中的所有元素
iterator erase(iterator first, iterator last) {
    iterator i = copy(last, finish, first); // copy 是全域函式，第 6 章
    destroy(i, finish);                 // destroy 是全域函式，第 2 章
}

```

```

    finish = finish - (last - first);
    return first;
}

// 清除某個位置上的元素
iterator erase(iterator position) {
    if (position + 1 != end())
        copy(position + 1, finish, position); // copy 是全域函式，第 6 章
    --finish;
    destroy(finish); // destroy 是全域函式，2.2.3 節
    return position;
}

void clear() { erase(begin(), end()); } // erase() 就定義在上面

```

圖 4-3a 展示 `erase(first, last)` 的動作。

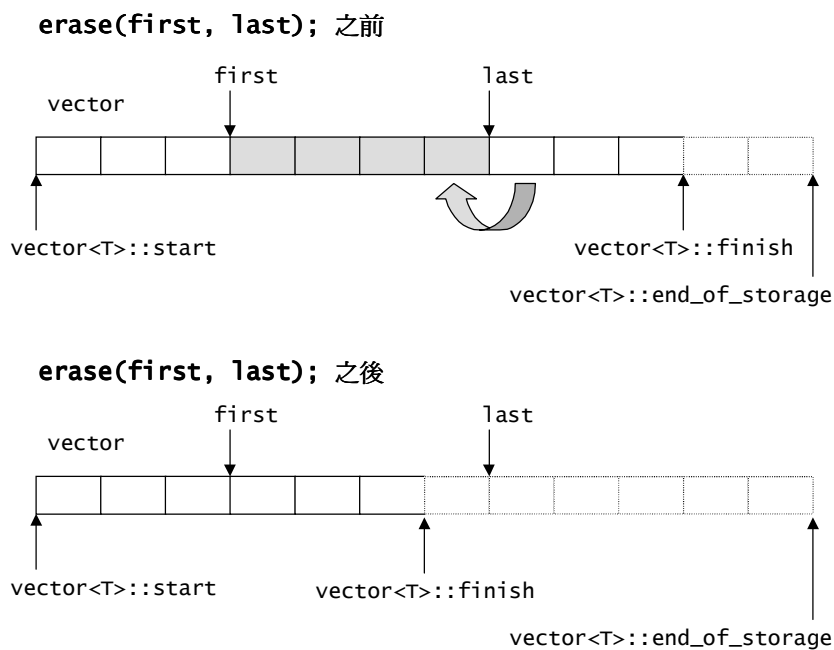


圖 4-3a 局部區間的清除動作：`erase(first, last)`

下面是 `vector::insert()` 實作內容：

```

// 從 position 開始，安插 n 個元素，元素初值為 x
template <class T, class Alloc>

```

```

void vector<T, Alloc>::insert(iterator position, size_type n, const T& x)
{
    if (n != 0) { // 當 n != 0 才進行以下所有動作
        if (size_type(end_of_storage - finish) >= n)
            // 備用空間大於等於「新增元素個數」
            T x_copy = x;
            // 以下計算安插點之後的現有元素個數
            const size_type elems_after = finish - position;
            iterator old_finish = finish;
            if (elems_after > n)
                // 「安插點之後的現有元素個數」大於「新增元素個數」
                uninitialized_copy(finish - n, finish, finish);
                finish += n; // 將 vector 尾端標記後移
                copy_backward(position, old_finish - n, old_finish);
                fill(position, position + n, x_copy); // 從安插點開始填入新值
            }
            else {
                // 「安插點之後的現有元素個數」小於等於「新增元素個數」
                uninitialized_fill_n(finish, n - elems_after, x_copy);
                finish += n - elems_after;
                uninitialized_copy(position, old_finish, finish);
                finish += elems_after;
                fill(position, old_finish, x_copy);
            }
        }
        else {
            // 備用空間小於「新增元素個數」（那就必須配置額外的記憶體）
            // 首先決定新長度：舊長度的兩倍，或舊長度+新增元素個數。
            const size_type old_size = size();
            const size_type len = old_size + max(old_size, n);
            // 以下配置新的 vector 空間
            iterator new_start = data_allocator::allocate(len);
            iterator new_finish = new_start;
            __STL_TRY {
                // 以下首先將舊 vector 的安插點之前的元素複製到新空間。
                new_finish = uninitialized_copy(start, position, new_start);
                // 以下再將新增元素（初值皆為 n）填入新空間。
                new_finish = uninitialized_fill_n(new_finish, n, x);
                // 以下再將舊 vector 的安插點之後的元素複製到新空間。
                new_finish = uninitialized_copy(position, finish, new_finish);
            }
        }
        # ifdef __STL_USE_EXCEPTIONS
        catch(...) {
            // 如有異常發生，實現 "commit or rollback" semantics.
            destroy(new_start, new_finish);
            data_allocator::deallocate(new_start, len);
            throw;
        }
    }
    # endif /* __STL_USE_EXCEPTIONS */
}

```

```

// 以下清除並釋放舊的 vector
destroy(start, finish);
deallocate();
// 以下調整水位標記
start = new_start;
finish = new_finish;
end_of_storage = new_start + len;
}
}
}

```

注意，安插完成後，新節點將位於標兵迭代器（上例之 `position`，標示出安插點）所指之節點的前方 — 這是 STL 對於「安插動作」的標準規範。圖 4-3b 展示 `insert(position, n, x)` 的動作。

`insert(position, n, x);`

(1) 備用空間 $2 \geq$ 新增元素個數 2

例：下圖， $n=2$

(1-1) 安插點之後的現有元素個數 3 > 新增元素個數 2

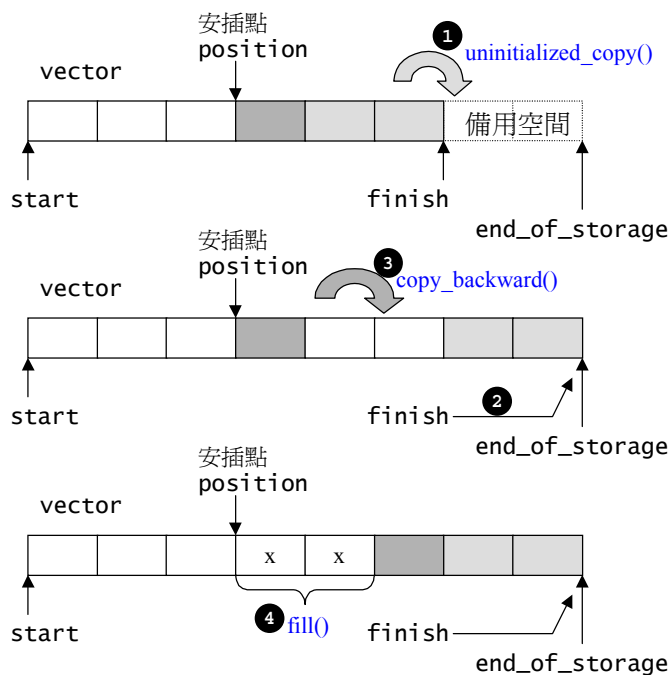


圖 4-3b-1 `insert(position, n, x)` 狀況 1

(1-2) 安插點之後的現有元素個數 $2 \leq$ 新增元素個數 3

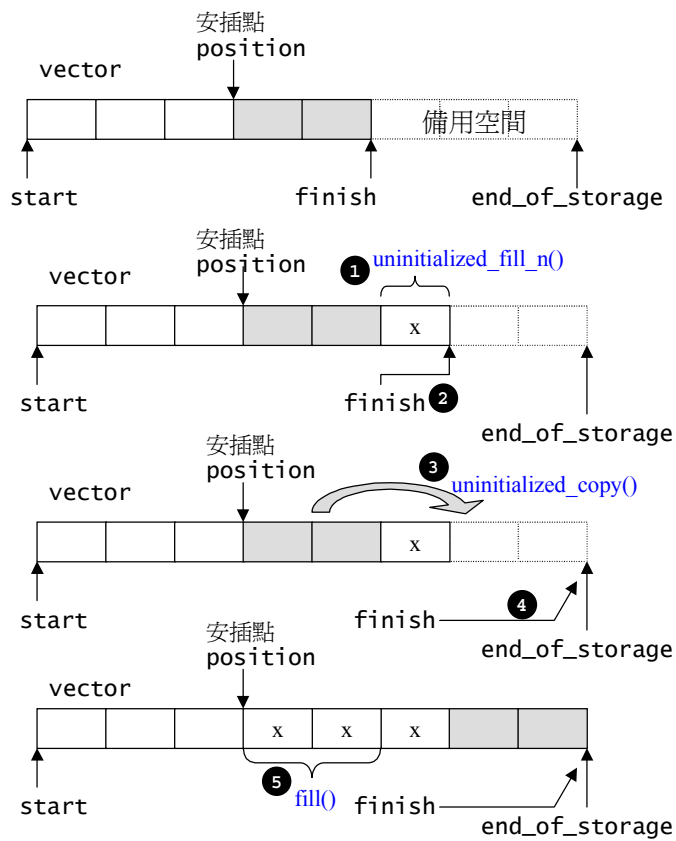


圖 4-3b-2 `insert(position, n, x)` 狀況 2

```
insert(position,n,x);
```

(2) 備用空間 < 新增元素個數

例：下圖， $n=3$

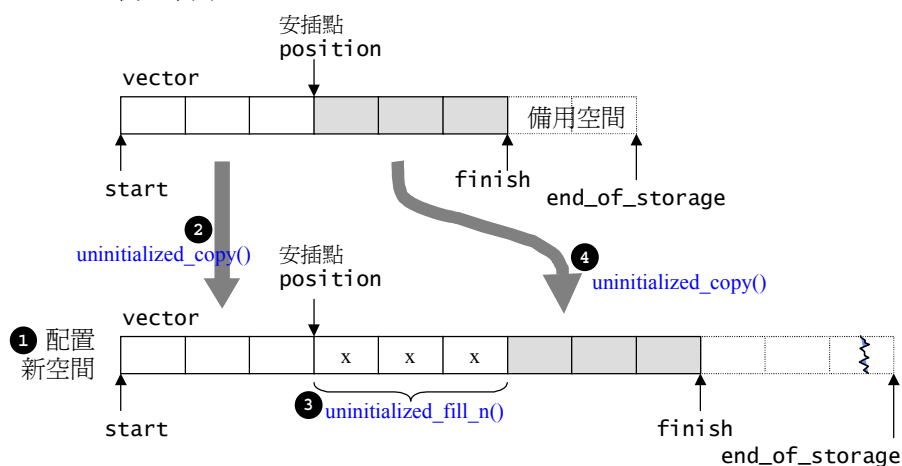


圖 4-3b-3 `insert(position,n,x)` 狀況 3

4.3 list

4.3.1 list 概述

相較於 `vector` 的連續線性空間，`list` 就顯得複雜許多，它的好處是每次安插或刪除一個元素，就配置或釋放一個元素空間。因此，`list` 對於空間的運用有絕對的精準，一點也不浪費。而且，對於任何位置的元素安插或元素移除，`list` 永遠是常數時間。

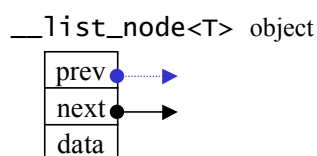
`list` 和 `vector` 是兩個最常被使用的容器。什麼時機下最適合使用哪一種容器，必須視元素的多寡、元素的構造複雜度（有無 `non-trivial copy constructor`, `non-trivial copy assignment operator`）、元素存取行為的特性而定。[Lippman 98] 6.3 節對這兩種容器提出了一份測試報告。

4.3.2 list 的節點 (node)

每一個設計過 `list` 的人都知道，`list` 本身和 `list` 的節點是不同的結構，需要分開設計。以下是 STL `list` 的節點 (node) 結構：

```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    void_pointer prev; // 型別為 void*。其實可設為 __list_node<T>*
    void_pointer next;
    T data;
};
```

顯然這是一個雙向串列¹。



4.3.3 list 的迭代器

`list` 不再能夠像 `vector` 一樣以原生指標做為迭代器，因為其節點不保證在儲存空間中連續存在。`list` 迭代器必須有能力指向 `list` 的節點，並有能力做正確的遞增、遞減、取值、成員存取…等動作。所謂「`list` 迭代器正確的遞增、遞減、取值、成員取用」動作是指，遞增時指向下一個節點，遞減時指向上一個節點，取值時取的是節點的資料值，成員取用時取用的是節點的成員，如圖 4-4。

由於 STL `list` 是一個雙向串列 (double linked-list)，迭代器必須具備前移、後移的能力。所以 `list` 提供的是 *Bidirectional Iterators*。

`list` 有一個重要性質：安插動作 (insert) 和接合動作 (splice) 都不會造成原有的 `list` 迭代器失效。這在 `vector` 是不成立的，因為 `vector` 的安插動作可能造成記憶體重新配置，導致原有的迭代器全部失效。甚至 `list` 的元素刪除動作

¹ SGI STL 另有一個單向串列 `slist`，我將在 4.9 節介紹它。

(erase)，也只有「指向被刪除元素」的那個迭代器失效，其他迭代器不受任何影響。

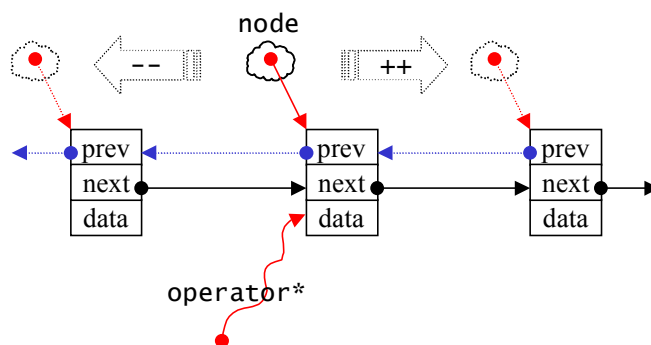


圖 4-4 list 的節點與 list 的迭代器

以下是 list 迭代器的設計：

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    typedef __list_iterator<T, T&, T*>      iterator;
    typedef __list_iterator<T, Ref, Ptr>    self;

    typedef bidirectional_iterator_tag iterator_category;
    typedef T value_type;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef __list_node<T>* link_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    link_type node; // 迭代器內部當然要有一個原生指標，指向 list 的節點

    // constructor
    __list_iterator(link_type x) : node(x) {}
    __list_iterator() {}
    __list_iterator(const iterator& x) : node(x.node) {}

    bool operator==(const self& x) const { return node == x.node; }
    bool operator!=(const self& x) const { return node != x.node; }
    // 以下對迭代器取值 (dereference)，取的是節點的資料值。
    reference operator*() const { return (*node).data; }

    // 以下是迭代器的成員存取 (member access) 運算子的標準作法。
```

```

pointer operator->() const { return &(operator*()); }

// 對迭代器累加 1，就是前進一個節點
self& operator++()
{
    node = (link_type)((*node).next);
    return *this;
}
self operator++(int)
{
    self tmp = *this;
    ++*this;
    return tmp;
}

// 對迭代器遞減 1，就是後退一個節點
self& operator--()
{
    node = (link_type)((*node).prev);
    return *this;
}
self operator--(int)
{
    self tmp = *this;
    --*this;
    return tmp;
}
};

```

4.3.4 list 的資料結構

SGI `list` 不僅是一個雙向串列，而且還是一個環狀雙向串列。所以它只需要一個指標，便可以完整表現整個串列：

```

template <class T, class Alloc = alloc> // 預設使用 alloc 為配置器
class list {
protected:
    typedef __list_node<T> list_node;
public:
    typedef list_node* link_type;

protected:
    link_type node; // 只要一個指標，便可表示整個環狀雙向串列
    ...
};

```

如果讓指標 `node` 指向刻意置於尾端的一個空白節點，`node` 便能符合 STL 對於「前閉後開」區間的要求，成為 `last` 迭代器，如圖 4-5。這麼一來，以下幾個函式便都可以輕易完成：

```

iterator begin() { return (link_type)((*node).next); }
iterator end() { return node; }
bool empty() const { return node->next == node; }
size_type size() const {
    size_type result = 0;
    distance(begin(), end(), result); // 全域函式，第 3 章。
    return result;
}
// 取頭節點的內容 (元素值)。
reference front() { return *begin(); }
// 取尾節點的內容 (元素值)。
reference back() { return *(--end()); }

```

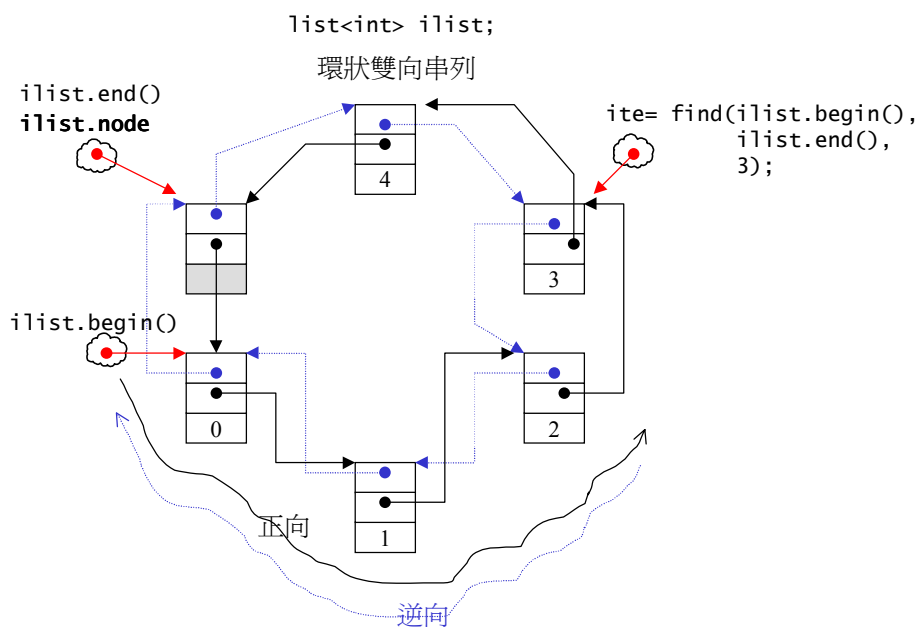


圖 4-5 list 示意圖。是環狀串列只需一個標記，即可完全表示整個串列。只要刻意在環狀串列的尾端加上一個空白節點，便符合 STL 規範之「前閉後開」區間。

4.3.5 list 的建構與記憶體管理：

constructor, push_back, insert

千頭萬緒該如何說起？以客端程式碼為引導，觀察其所得結果並實證源碼，是個

良好的學習路徑。下面是一個測試程式，我的觀察重點在建構的方式以及大小的變化：

```
// filename : 4list-test.cpp
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    int i;
    list<int> ilist;
    cout << "size=" << ilist.size() << endl;    // size=0

    ilist.push_back(0);
    ilist.push_back(1);
    ilist.push_back(2);
    ilist.push_back(3);
    ilist.push_back(4);
    cout << "size=" << ilist.size() << endl;    // size=5

    list<int>::iterator ite;
    for(ite = ilist.begin(); ite != ilist.end(); ++ite)
        cout << *ite << ' ';                // 0 1 2 3 4
    cout << endl;

    ite = find(ilist.begin(), ilist.end(), 3);
    if (ite!=0)
        ilist.insert(ite, 99);

    cout << "size=" << ilist.size() << endl;    // size=6
    cout << *ite << endl;                      // 3

    for(ite = ilist.begin(); ite != ilist.end(); ++ite)
        cout << *ite << ' ';                // 0 1 2 99 3 4
    cout << endl;

    ite = find(ilist.begin(), ilist.end(), 1);
    if (ite!=0)
        cout << *(ilist.erase(ite)) << endl;    // 2

    for(ite = ilist.begin(); ite != ilist.end(); ++ite)
        cout << *ite << ' ';                // 0 2 99 3 4
    cout << endl;
}
```

`list` 預設使用 `alloc` (2.2.4 節) 做為空間配置器，並據此另外定義了一個 `list_node_allocator`，為的是更方便地以節點大小為配置單位：

```
template <class T, class Alloc = alloc> // 預設使用 alloc 為配置器
class list {
protected:
    typedef __list_node<T> list_node;
    // 專屬之空間配置器，每次配置一個節點大小：
    typedef simple_alloc<list_node, Alloc> list_node_allocator;
    ...
};
```

於是，`list_node_allocator(n)` 表示配置 `n` 個節點空間。以下四個函式，分別用來配置、釋放、建構、摧毀一個節點：

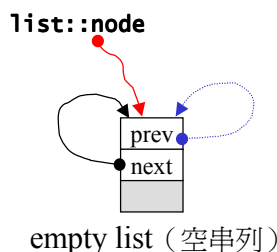
```
protected:
    // 配置一個節點並傳回
    link_type get_node() { return list_node_allocator::allocate(); }
    // 釋放一個節點
    void put_node(link_type p) { list_node_allocator::deallocate(p); }

    // 產生 (配置並建構) 一個節點，帶有元素值
    link_type create_node(const T& x) {
        link_type p = get_node();
        construct(&p->data, x); // 全域函式，建構/解構基本工具。
        return p;
    }
    // 摧毀 (解構並釋放) 一個節點
    void destroy_node(link_type p) {
        destroy(&p->data); // 全域函式，建構/解構基本工具。
        put_node(p);
    }
```

`list` 提供有許多 constructors，其中一個是 default constructor，允許我們不指定任何參數做出一個空的 `list` 出來：

```
public:
    list() { empty_initialize(); } // 產生一個空串列。

protected:
    void empty_initialize()
    {
        node = get_node(); // 配置一個節點空間，令 node 指向它。
        node->next = node; // 令 node 頭尾都指向自己，不設元素值。
        node->prev = node;
    }
```



當我們以 `push_back()` 將新元素安插於 `list` 尾端，此函式內部呼叫 `insert()`：

```
void push_back(const T& x) { insert(end(), x); }
```

`insert()` 是一個多載化函式，有多種型式，其中最簡單的一種如下，符合以上所需。首先配置並建構一個節點，然後在尾端做適當的指標動作，將新節點安插進去：

```
// 函式目的：在迭代器 position 所指位置安插一個節點，內容為 x。
iterator insert(iterator position, const T& x) {
    link_type tmp = create_node(x); // 產生一個節點（設妥內容為 x）
    // 調整雙向指標，使 tmp 安插進去。
    tmp->next = position.node;
    tmp->prev = position.node->prev;
    (link_type(position.node->prev))->next = tmp;
    position.node->prev = tmp;
    return tmp;
}
```

於是，先前測試程式連續安插了五個節點（其值為 0 1 2 3 4）之後，`list` 的狀態如圖 4-5。如果我們希望在 `list` 內的某處安插新節點，首先必須確定安插位置，例如我希望在資料值為 3 的節點處安插一個資料值為 99 的節點，可以這麼做：

```
ilite = find(il.begin(), il.end(), 3);
if (ilite!=0)
    il.insert(ilite, 99);
```

`find()` 動作稍後再做說明。安插之後的 `list` 狀態如圖 4-6。注意，安插完成後，新節點將位於標兵迭代器（標示出安插點）所指之節點的前方 — 這是 STL 對於「安插動作」的標準規範。由於 `list` 不像 `vector` 那樣有可能在空間不足時做重新配置、資料搬移的動作，所以安插前的所有迭代器在安插動作之後都仍然有效。

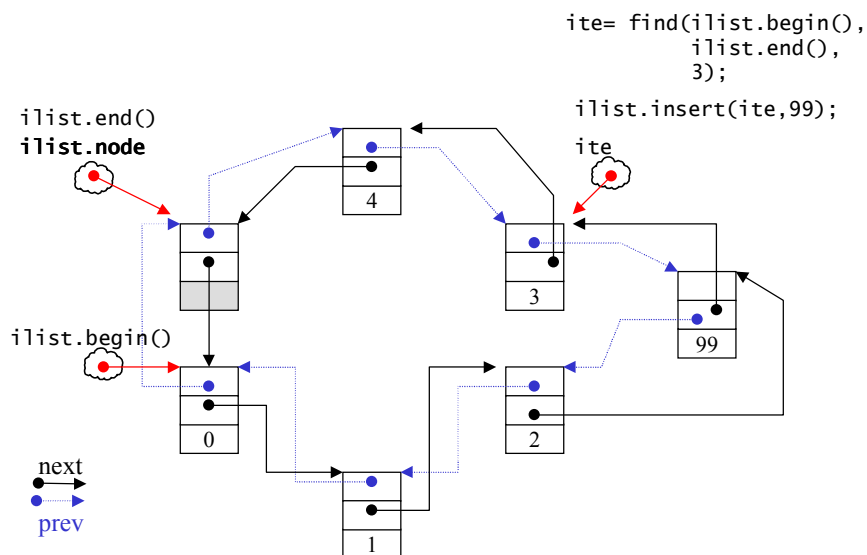


圖 4-6 安插新節點 99 於節點 3 的位置上（所謂安插是指「安插在...之前」）

4.3.6 list 的元素操作：

`push_front`, `push_back`, `erase`, `pop_front`, `pop_back`,
`clear`, `remove`, `unique`, `splice`, `merge`, `reverse`, `sort`

`list` 所提供的元素操作動作很多，無法在有限的篇幅中一一講解 — 其實也沒有這種必要。為搭配先前對空間配置的討論，我挑選數個相關函式做為解說對象。先前示例中出現有尾部安插動作（`push_back`），現在我們來看看其他的安插動作和移除動作。

```
// 安插一個節點，做為頭節點
void push_front(const T& x) { insert(begin(), x); }
// 安插一個節點，做為尾節點（上一小節才介紹過）
void push_back(const T& x) { insert(end(), x); }

// 移除迭代器 position 所指節點
iterator erase(iterator position) {
    link_type next_node = link_type(position.node->next);
    link_type prev_node = link_type(position.node->prev);
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
}
```

```

    }

    // 移除頭節點
    void pop_front() { erase(begin()); }
    // 移除尾節點
    void pop_back()
    {
        iterator tmp = end();
        erase(--tmp);
    }

    // 清除所有節點（整個串列）
    template <class T, class Alloc>
    void list<T, Alloc>::clear()
    {
        link_type cur = (link_type) node->next; // begin()
        while (cur != node) { // 巡訪每一個節點
            link_type tmp = cur;
            cur = (link_type) cur->next;
            destroy_node(tmp); // 摧毀（解構並釋放）一個節點
        }
        // 恢復 node 原始狀態
        node->next = node;
        node->prev = node;
    }

    // 將數值為 value 之所有元素移除
    template <class T, class Alloc>
    void list<T, Alloc>::remove(const T& value) {
        iterator first = begin();
        iterator last = end();
        while (first != last) { // 巡訪每一個節點
            iterator next = first;
            ++next;
            if (*first == value) erase(first); // 找到就移除
            first = next;
        }
    }

    // 移除數值相同的連續元素。注意，只有「連續而相同的元素」，才會被移除剩一個。
    template <class T, class Alloc>
    void list<T, Alloc>::unique() {
        iterator first = begin();
        iterator last = end();
        if (first == last) return; // 空串列，什麼都不必做。
        iterator next = first;
        while (++next != last) { // 巡訪每一個節點
            if (*first == *next) // 如果在此區段中有相同的元素
                erase(next); // 移除之
            else

```



```

        first = next;           // 調整指標
        next = first;          // 修正區段範圍
    }
}

```

由於 `list` 是一個雙向環狀串列，只要我們把邊際條件處理好，那麼，在頭部或尾部安插元素 (`push_front` 和 `push_back`)，動作幾乎是一樣的，在頭部或尾部移除元素 (`pop_front` 和 `pop_back`)，動作也幾乎是一樣的。移除 (`erase`) 某個迭代器所指元素，只是做一些指標搬移動作而已，並不複雜。如果圖 4-6 再經以下搜尋並移除的動作，狀況將如圖 4-7。

```

ite = find(ilist.begin(), ilist.end(), 1);
if (ite!=0)
    cout << *(ilist.erase(ite)) << endl;

```

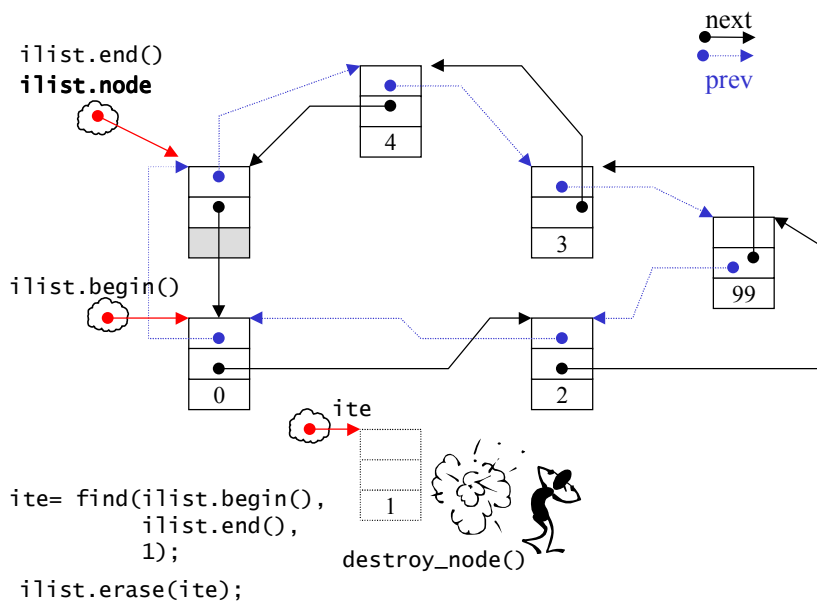


圖 4-7 移除「元素值為 1」的節點

`list` 內部提供一個所謂的遷移動作 (`transfer`)：將某連續範圍的元素遷移到某個特定位置之前。技術上很簡單，節點間的指標移動而已。這個動作為其他的複雜動作如 `splice`, `sort`, `merge` 等奠定良好的基礎。下面是 `transfer` 的源碼：

```
protected:
    // 將 [first,last) 內的所有元素搬移到 position 之前。
    void transfer(iterator position, iterator first, iterator last) {
        if (position != last) {
            (*(link_type((*last.node).prev))).next = position.node;    // (1)
            (*(link_type((*first.node).prev))).next = last.node;      // (2)
            (*(link_type((*position.node).prev))).next = first.node;   // (3)
            link_type tmp = link_type((*position.node).prev);         // (4)
            (*position.node).prev = (*last.node).prev;                // (5)
            (*last.node).prev = (*first.node).prev;                   // (6)
            (*first.node).prev = tmp;                                  // (7)
        }
    }
}
```

以上七個動作，一步一步地顯示於圖 4-8a。

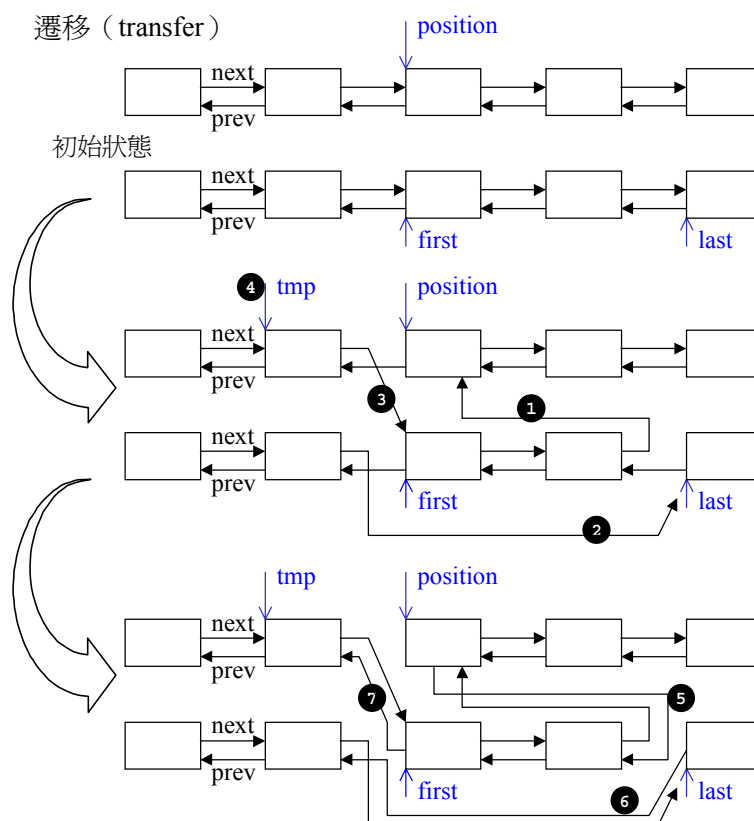


圖 4-8a list<T>::transfer 的動作示意

`transfer` 所接受的 `[first,last)` 區間，是否可以在同一個 `list` 之中呢？答案是可以。你只要想像圖 4-8a 所畫的兩條 `lists` 其實是同一個 `list` 的兩個區段，就不難得到答案了。

上述的 `transfer` 並非公開介面。`list` 公開提供的是所謂的接合動作(`splice`)：將某連續範圍的元素從一個 `list` 搬移到另一個（或同一個）`list` 的某個定點。如果接續先前 `4list-test.cpp` 程式的最後執行點，繼續執行以下 `splice` 動作：

```
int iv[5] = { 5,6,7,8,9 };
list<int> ilist2(iv, iv+5);

// 目前，ilist 的內容為 0 2 99 3 4
ite = find(ilist.begin(), ilist.end(), 99);
ilist.splice(ite, ilist2);           // 0 2 5 6 7 8 9 99 3 4
ilist.reverse();                     // 4 3 99 9 8 7 6 5 2 0
ilist.sort();                         // 0 2 3 4 5 6 7 8 9 99
```

很容易便可看出效果。圖 4-8b 顯示接合動作。技術上很簡單，只是節點間的指標移動而已，這些動作已完全由 `transfer()` 做掉了。

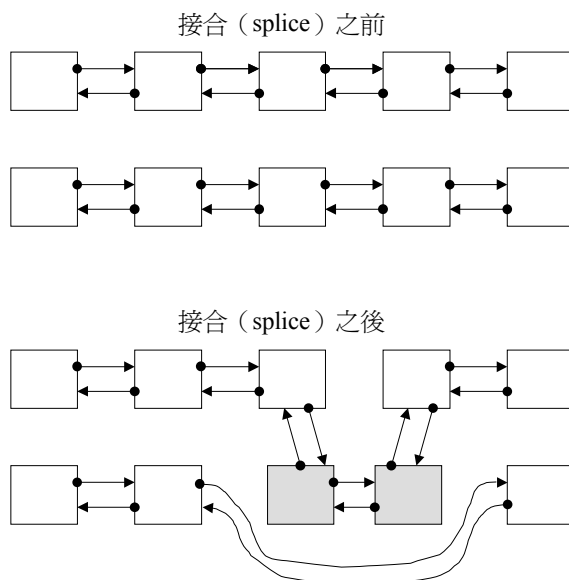


圖 4-8b `list` 的接合 (`splice`) 動作

爲了提供各種介面彈性，`list<T>::splice` 有許多版本：

```
public:
    // 將 x 接合於 position 所指位置之前。x 必須不同於 *this。
    void splice(iterator position, list& x) {
        if (!x.empty())
            transfer(position, x.begin(), x.end());
    }

    // 將 i 所指元素接合於 position 所指位置之前。position 和 i 可指向同一個 list。
    void splice(iterator position, list&, iterator i) {
        iterator j = i;
        ++j;
        if (position == i || position == j) return;
        transfer(position, i, j);
    }

    // 將 [first,last) 內的所有元素接合於 position 所指位置之前。
    // position 和 [first,last) 可指向同一個 list，
    // 但 position 不能位於 [first,last) 之內。
    void splice(iterator position, list&, iterator first, iterator last) {
        if (first != last)
            transfer(position, first, last);
    }
```

以下是 `merge()`，`reverse()`，`sort()` 的源碼。有了 `transfer()` 在手，這些動作都不難完成。

```
// merge() 將 x 合併到 *this 身上。兩個 lists 的內容都必須先經過遞增排序。
template <class T, class Alloc>
void list<T, Alloc>::merge(list<T, Alloc>& x) {
    iterator first1 = begin();
    iterator last1 = end();
    iterator first2 = x.begin();
    iterator last2 = x.end();

    // 注意：前提是，兩個 lists 都已經過遞增排序，
    while (first1 != last1 && first2 != last2)
        if (*first2 < *first1) {
            iterator next = first2;
            transfer(first1, first2, ++next);
            first2 = next;
        }
        else
            ++first1;
    if (first2 != last2) transfer(last1, first2, last2);
}
```

```

// reverse() 將 *this 的內容逆向重置
template <class T, class Alloc>
void list<T, Alloc>::reverse() {
    // 以下判斷，如果是空白串列，或僅有一個元素，就不做任何動作。
    // 使用 size() == 0 || size() == 1 來判斷，雖然也可以，但是比較慢。
    if (node->next == node || link_type(node->next)->next == node)
        return;
    iterator first = begin();
    ++first;
    while (first != end()) {
        iterator old = first;
        ++first;
        transfer(begin(), old, first);
    }
}

// list 不能使用 STL 演算法 sort()，必須使用自己的 sort() member function，
// 因為 STL 演算法 sort() 只接受 RandomAccessIterator。
// 本函式採用 quick sort。
template <class T, class Alloc>
void list<T, Alloc>::sort() {
    // 以下判斷，如果是空白串列，或僅有一個元素，就不做任何動作。
    // 使用 size() == 0 || size() == 1 來判斷，雖然也可以，但是比較慢。
    if (node->next == node || link_type(node->next)->next == node)
        return;

    // 一些新的 lists，做為中介資料存放區
    list<T, Alloc> carry;
    list<T, Alloc> counter[64];
    int fill = 0;
    while (!empty()) {
        carry.splice(carry.begin(), *this, begin());
        int i = 0;
        while(i < fill && !counter[i].empty()) {
            counter[i].merge(carry);
            carry.swap(counter[i++]);
        }
        carry.swap(counter[i]);
        if (i == fill) ++fill;
    }

    for (int i = 1; i < fill; ++i)
        counter[i].merge(counter[i-1]);
    swap(counter[fill-1]);
}

```

4.4 deque

4.4.1 deque 概述

vector 是單向開口的連續線性空間，**deque** 則是一種雙向開口的連續線性空間。所謂雙向開口，意思是在頭尾兩端分別做元素的安插和刪除動作，如圖 4-9。**vector** 當然也可以在頭尾兩端做動作（從技術觀點），但是其頭部動作效率奇差，無法被接受。

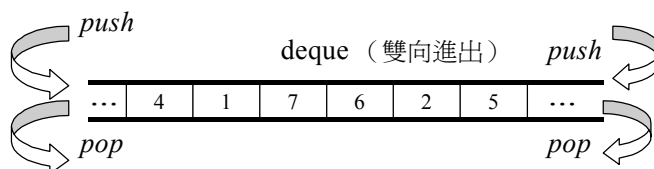


圖 4-9 deque 示意

deque 和 **vector** 的最大差異，一在於 **deque** 允許於常數時間內對起頭端進行元素的安插或移除動作，二在於 **deque** 沒有所謂容量 (capacity) 觀念，因為它是動態地以分段連續空間組合而成，隨時可以增加一段新的空間並鏈接起來。換句話說，像 **vector** 那樣「因舊空間不足而重新配置一塊更大空間，然後複製元素，再釋放舊空間」這樣的事情在 **deque** 是不會發生的。也因此，**deque** 沒有必要提供所謂的空間保留 (reserve) 功能。

雖然 **deque** 也提供 *Random Access Iterator*，但它的迭代器並不是原生指標，其複雜度和 **vector** 不可以道里計（稍後看到源碼，你便知道），這當然在在影響了各個運算層面。因此，除非必要，我們應儘可能選擇使用 **vector** 而非 **deque**。對 **deque** 進行的排序動作，爲了最高效率，可將 **deque** 先完整複製到一個 **vector** 身上，將 **vector** 排序後（利用 STL sort 演算法），再複製回 **deque**。

4.4.2 deque 的控制器

`deque` 是連續空間（至少邏輯看來如此），連續線性空間總令我們聯想到 `array` 或 `vector`。`array` 無法成長，`vector` 雖可成長，卻只能向尾端成長，而且其所謂成長原是個假象，事實上是 (1) 另覓更大空間、(2) 將原資料複製過去、(3) 釋放原空間 三部曲。如果不是 `vector` 每次配置新空間時都有留下一些餘裕，其「成長」假象所帶來的代價將是相當高昂。

`deque` 係由一段一段的定量連續空間構成。一旦有必要在 `deque` 的前端或尾端增加新空間，便配置一段定量連續空間，串接在整個 `deque` 的頭端或尾端。`deque` 的最大任務，便是在這些分段的定量連續空間上，維護其整體連續的假象，並提供隨機存取的介面。避開了「重新配置、複製、釋放」的輪迴，代價則是複雜的迭代器架構。

受到分段連續線性空間的字面影響，我們可能以為 `deque` 的實作複雜度和 `vector` 相比雖不中亦不遠矣，其實不然。主要因為，既曰分段連續線性空間，就必須有中央控制，而為了維護整體連續的假象，資料結構的設計及迭代器前進後退等動作都頗為繁瑣。`deque` 的實作碼份量遠比 `vector` 或 `list` 都多得多。

`deque` 採用一塊所謂的 *map*（注意，不是 STL 的 `map` 容器）做為主控。這裡所謂 *map* 是一小塊連續空間，其中每個元素（此處稱為一個節點，*node*）都是指標，指向另一段（較大的）連續線性空間，稱為緩衝區。緩衝區才是 `deque` 的儲存空間主體。SGI STL 允許我們指定緩衝區大小，預設值 0 表示將使用 512 bytes 緩衝區。

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:
    // Basic types
    typedef T value_type;
    typedef value_type* pointer;
    ...
protected:
    // Internal typedefs
    // 元素的指標的指標 (pointer of pointer of T)
    typedef pointer* map_pointer;

protected:
    // Data members
```

```

map_pointer map;    // 指向 map，map 是塊連續空間，其內的每個元素
                    // 都是一個指標（稱為節點），指向一塊緩衝區。
size_type map_size; // map 內可容納多少指標。
...
};

```

把令人頭皮發麻的各種型別定義（爲了型別安全，那其實是有必要的）整理一下，我們便可發現，`map` 其實是一個 `T**`，也就是說它是一個指標，所指之物又是一個指標，指向型別爲 `T` 的一塊空間，如圖 4-10。

稍後在 `deque` 的建構過程中，我會詳細解釋 `map` 的配置及維護。

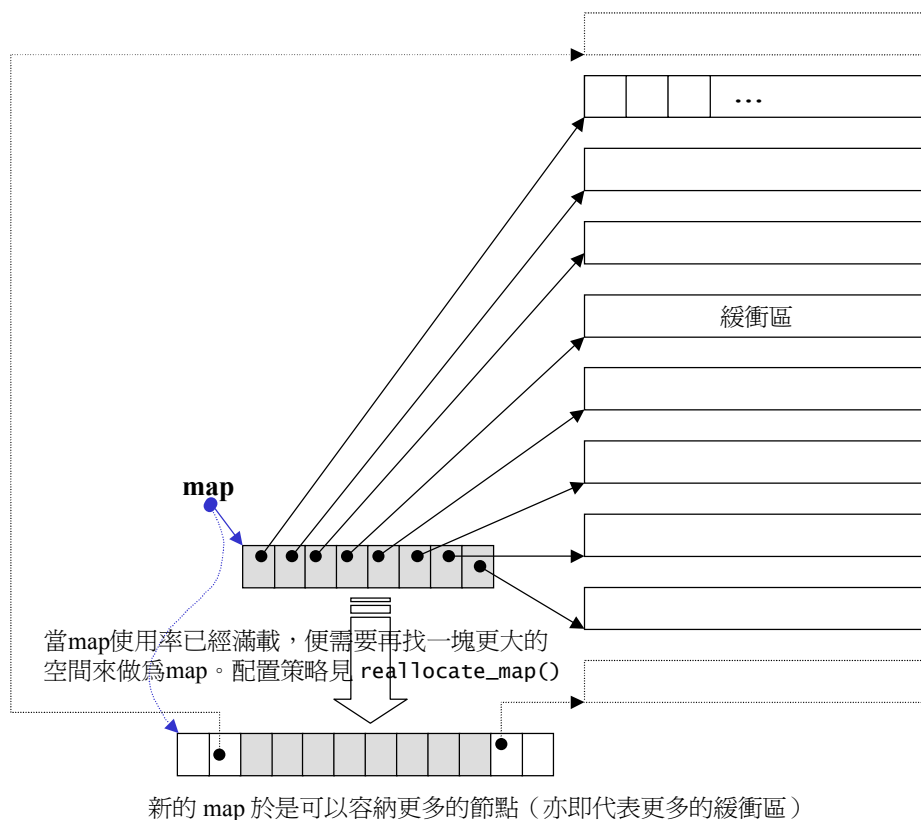


圖 4-10 deque 的結構設計中，`map` 和 node-buffer（節點-緩衝區）的關係。

4.4.3 deque 的迭代器

`deque` 是分段連續空間。維護其「整體連續」假象的任務，著落在迭代器的 `operator++` 和 `operator--` 兩個運算子身上。

讓我們思考一下，`deque` 迭代器應該具備什麼結構。首先，它必須能夠指出分段連續空間（亦即緩衝區）在哪裡，其次它必須能夠判斷自己是否已經處於其所在緩衝區的邊緣，如果是，一旦前進或後退時就必須跳躍至下一個或上一個緩衝區。為了能夠正確跳躍，`deque` 必須隨時掌握管控中心（*map*）。下面這種實作方式符合需求：

```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator { // 未繼承 std::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)); }

    // 未繼承 std::iterator，所以必須自行撰寫五個必要的迭代器相應型別（第 3 章）
    typedef random_access_iterator_tag iterator_category; // (1)
    typedef T value_type;                                // (2)
    typedef Ptr pointer;                                  // (3)
    typedef Ref reference;                                // (4)
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;                    // (5)
    typedef T** map_pointer;

    typedef __deque_iterator self;

    // 保持與容器的聯結
    T* cur;        // 此迭代器所指之緩衝區中的現行（current）元素
    T* first;      // 此迭代器所指之緩衝區的頭
    T* last;       // 此迭代器所指之緩衝區的尾（含備用空間）
    map_pointer node; // 指向管控中心
    ...
};
```

其中用來決定緩衝區大小的函式 `buffer_size()`，呼叫 `__deque_buf_size()`，後者是個全域函式，定義如下：

```
// 如果 n 不為 0，傳回 n，表示 buffer size 由使用者自定。
// 如果 n 為 0，表示 buffer size 使用預設值，那麼
// 如果 sz（元素大小，sizeof(value_type)）小於 512，傳回 512/sz，
// 如果 sz 不小於 512，傳回 1。
```

```

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));
}

```

圖 4-11 是 deque 的中控器、緩衝區、迭代器的相互關係。

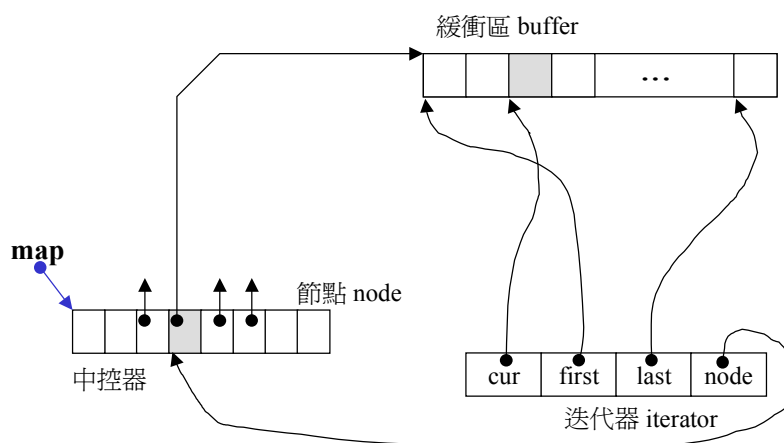


圖 4-11 deque 的中控器、緩衝區、迭代器的相互關係

假設現在我們產生一個 `deque<int>`，並令其緩衝區大小為 32，於是每個緩衝區可容納 $32/\text{sizeof}(\text{int})=4$ 個元素。經過某些操作之後，`deque` 擁有 20 個元素，那麼其 `begin()` 和 `end()` 所傳回的兩個迭代器應該如圖 4-12。這兩個迭代器事實上一直保持在 `deque` 內，名為 `start` 和 `finish`，稍後在 `deque` 資料結構中便可看到）。

20 個元素需要 $20/8=3$ 個緩衝區，所以 `map` 之內運用了三個節點。迭代器 `start` 內的 `cur` 指標當然指向緩衝區的第一個元素，迭代器 `finish` 內的 `cur` 指標當然指向緩衝區的最後元素（的下一位置）。注意，最後一個緩衝區尚有備用空間。稍後如果有新元素要安插於尾端，可直接拿此備用空間來使用。

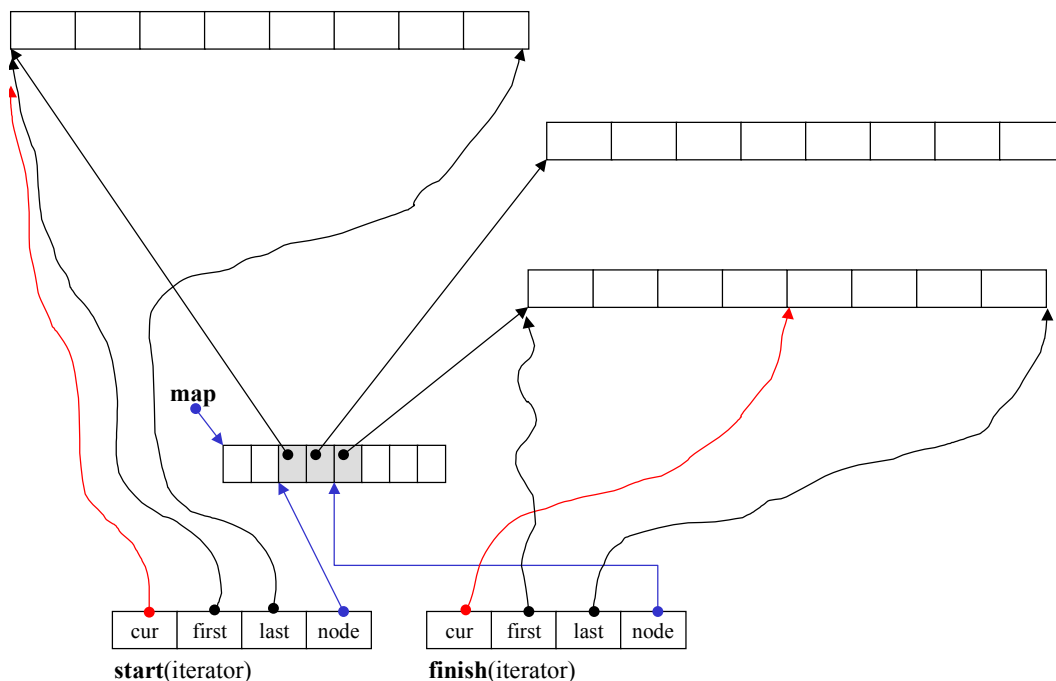


圖 4-12 `deque::begin()` 傳回迭代器 `start`，`deque::end()` 傳回迭代器 `finish`。這兩個迭代器都是 `deque` 的 data members。圖中所示的這個 `deque` 擁有 20 個 `int` 元素，以 3 個緩衝區儲存之。每個緩衝區 32 bytes，可儲存 8 個 `int` 元素。`map` 大小為 8 (起始值)，目前用了 3 個節點。

下面是 `deque` 迭代器的幾個關鍵行為。由於迭代器內對各種指標運算都做了多載化動作，所以各種指標運算如加、減、前進、後退...都不能直觀視之。其中最重點的關鍵就是：一旦行進時遇到緩衝區邊緣，要特別當心，視前進或後退而定，可能需要呼叫 `set_node()` 跳一個緩衝區：

```
void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first + difference_type(buffer_size());
}
```

// 以下各個多載化運算子是 `__deque_iterator<>` 成功運作的關鍵。

```
reference operator*() const { return *cur; }
pointer operator->() const { return &(operator*()); }
```

```

difference_type operator-(const self& x) const {
    return difference_type(buffer_size()) * (node - x.node - 1) +
        (cur - first) + (x.last - x.cur);
}

// 參考 More Effective C++, item6: Distinguish between prefix and
// postfix forms of increment and decrement operators.
self& operator++() {
    ++cur; // 切換至下一個元素。
    if (cur == last) { // 如果已達所在緩衝區的尾端，
        set_node(node + 1); // 就切換至下一節點（亦即緩衝區）
        cur = first; // 的第一個元素。
    }
    return *this;
}
self operator++(int) { // 後置式，標準寫法
    self tmp = *this;
    ++*this;
    return tmp;
}
self& operator--() {
    if (cur == first) { // 如果已達所在緩衝區的頭端，
        set_node(node - 1); // 就切換至前一節點（亦即緩衝區）
        cur = last; // 的最後一個元素。
    }
    --cur; // 切換至前一個元素。
    return *this;
}
self operator--(int) { // 後置式，標準寫法
    self tmp = *this;
    --*this;
    return tmp;
}

// 以下實現隨機存取。迭代器可以直接跳躍 n 個距離。
self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
        // 標的位置在同一緩衝區內
        cur += n;
    else {
        // 標的位置不在同一緩衝區內
        difference_type node_offset =
            offset > 0 ? offset / difference_type(buffer_size())
                : -difference_type((-offset - 1) / buffer_size()) - 1;
        // 切換至正確的節點（亦即緩衝區）
        set_node(node + node_offset);
        // 切換至正確的元素

```

```

        cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
}

// 參考 More Effective C++, item22: Consider using op= instead of
// stand-alone op.
self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n; // 喚起 operator+=
}

self& operator-=(difference_type n) { return *this += -n; }
// 以上利用 operator+= 來完成 operator-=

// 參考 More Effective C++, item22: Consider using op= instead of
// stand-alone op.
self operator-(difference_type n) const {
    self tmp = *this;
    return tmp -= n; // 喚起 operator-=
}

// 以下實現隨機存取。迭代器可以直接跳躍 n 個距離。
reference operator[](difference_type n) const { return *(*this + n); }
// 以上喚起 operator*, operator+

bool operator==(const self& x) const { return cur == x.cur; }
bool operator!=(const self& x) const { return !(*this == x); }
bool operator<(const self& x) const {
    return (node == x.node) ? (cur < x.cur) : (node < x.node);
}

```

4.4.4 deque 的資料結構

deque 除了維護一個先前說過的指向 *map* 的指標外，也維護 *start*, *finish* 兩個迭代器，分別指向第一緩衝區的第一個元素和最後緩衝區的最後一個元素（的下一位置）。此外它當然也必須記住目前的 *map* 大小。因為一旦 *map* 所提供的節點不足，就必須重新配置更大的一塊 *map*。

```

// 見 __deque_buf_size()。BufSize 預設值為 0 的唯一理由是爲了閃避某些
// 編譯器在處理常數算式 (constant expressions) 時的臭蟲。
// 預設使用 alloc 爲配置器。
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:
    // Basic types
    typedef T value_type;

```

```

typedef value_type* pointer;
typedef size_t size_type;

public:                                     // Iterators
    typedef __deque_iterator<T, T&, T*, BufSiz> iterator;

protected:                               // Internal typedefs
    // 元素的指標的指標 (pointer of pointer of T)
    typedef pointer* map_pointer;

protected:                               // Data members
    iterator start;                        // 表現第一個節點。
    iterator finish;                       // 表現最後一個節點。

    map_pointer map;                       // 指向 map，map 是塊連續空間，
                                           // 其每個元素都是個指標，指向一個節點（緩衝區）。
    size_type map_size;                   // map 內有多少指標。
    ...
};

```

有了上述結構，以下數個機能便可輕易完成：

```

public:                                     // Basic accessors
    iterator begin() { return start; }
    iterator end() { return finish; }

    reference operator[](size_type n) {
        return start[difference_type(n)]; // 喚起 __deque_iterator<>::operator[]
    }

    reference front() { return *start; } // 喚起 __deque_iterator<>::operator*
    reference back() {
        iterator tmp = finish;
        --tmp; // 喚起 __deque_iterator<>::operator--
        return *tmp; // 喚起 __deque_iterator<>::operator*
        // 以上三行何不改為：return *(finish-1);
        // 因為 __deque_iterator<> 沒有為 (finish-1) 定義運算子?!
    }

    // 下行最後有兩個 `;'，雖奇怪但合乎語法。
    size_type size() const { return finish - start; }
    // 以上喚起 iterator::operator-
    size_type max_size() const { return size_type(-1); }
    bool empty() const { return finish == start; }

```

4.4.5 deque 的建構與記憶體管理 ctor, push_back, push_front

千頭萬緒該如何說起？以客端程式碼為引導，觀察其所得結果並實證源碼，是個良好的學習路徑。下面是一個測試程式，我的觀察重點在建構的方式以及大小的變化，以及容器最前端的安插功能：

```
// filename : 4deque-test.cpp
#include <deque>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    deque<int, alloc, 32> ideq(20, 9);           // 注意，alloc 只適用於 G++
    cout << "size=" << ideq.size() << endl;      // size=20
    // 現在，應該已經建構了一個 deque，有 20 個 int 元素，初值皆為 9。
    // 緩衝區大小為 32bytes。

    // 為每一個元素設定新值。
    for(int i=0; i<ideq.size(); ++i)
        ideq[i] = i;

    for(int i=0; i<ideq.size(); ++i)
        cout << ideq[i] << ' ';                 // 0 1 2 3 4 5 6...19
    cout << endl;

    // 在最尾端增加 3 個元素，其值為 0,1,2
    for(int i=0; i<3; i++)
        ideq.push_back(i);

    for(int i=0; i<ideq.size(); ++i)
        cout << ideq[i] << ' ';                 // 0 1 2 3 ... 19 0 1 2
    cout << endl;
    cout << "size=" << ideq.size() << endl;      // size=23

    // 在最尾端增加 1 個元素，其值為 3
    ideq.push_back(3);
    for(int i=0; i<ideq.size(); ++i)
        cout << ideq[i] << ' ';                 // 0 1 2 3 ... 19 0 1 2 3
    cout << endl;
    cout << "size=" << ideq.size() << endl;      // size=24

    // 在最前端增加 1 個元素，其值為 99
    ideq.push_front(99);
    for(int i=0; i<ideq.size(); ++i)
```

```

        cout << ideq[i] << ' ';           // 99 0 1 2 3...19 0 1 2 3
    cout << endl;
    cout << "size=" << ideq.size() << endl;    // size=25

    // 在最前端增加 2 個元素，其值分別為 98,97
    ideq.push_front(98);
    ideq.push_front(97);
    for(int i=0; i<ideq.size(); ++i)
        cout << ideq[i] << ' ';           // 97 98 99 0 1 2 3...19 0 1 2 3
    cout << endl;
    cout << "size=" << ideq.size() << endl;    // size=27

    // 搜尋數值為 99 的元素，並列印出來。
    deque<int,alloc,32>::iterator itr;
    itr = find(ideq.begin(), ideq.end(), 99);
    cout << *itr << endl;                   // 99
    cout << *(itr.cur) << endl;             // 99
}

```

`deque` 的緩衝區擴充動作相當瑣碎繁雜，以下將以分解動作的方式一步一步圖解說明。程式一開始宣告了一個 `deque`：

```
deque<int,alloc,32> ideq(20,9);
```

其緩衝區大小為 32 bytes，並令其保留 20 個元素空間，每個元素初值為 9。為了指定 `deque` 的第三個 `template` 參數（緩衝區大小），我們必須將前兩個參數都指明出來（這是 C++ 語法規則），因此必須明確指定 `alloc`（第二章）為空間配置器。現在，`deque` 的情況如圖 4-12（該圖並未顯示每個元素的初值為 9）。

`deque` 自行定義了兩個專屬的空間配置器：

```

protected:           // Internal typedefs
    // 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;
    // 專屬之空間配置器，每次配置一個指標大小
    typedef simple_alloc<pointer, Alloc> map_allocator;

```

並提供有一個 `constructor` 如下：

```

deque(int n, const value_type& value)
: start(), finish(), map(0), map_size(0)
{
    fill_initialize(n, value);
}

```

其內所呼叫的 `fill_initialize()` 負責產生並安排好 `deque` 的結構，並將元素

的初值設定妥當：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n,
                                              const value_type& value) {
    create_map_and_nodes(n); // 把 deque 的結構都產生並安排好
    map_pointer cur;
    __STL_TRY {
        // 為每個節點的緩衝區設定初值
        for (cur = start.node; cur < finish.node; ++cur)
            uninitialized_fill(*cur, *cur + buffer_size(), value);
        // 最後一個節點的設定稍有不同 (因為尾端可能有備用空間，不必設初值)
        uninitialized_fill(finish.first, finish.cur, value);
    }
    catch(...) {
        ...
    }
}
```

其中 create_map_and_nodes() 負責產生並安排好 deque 的結構：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements)
{
    // 需要節點數=(元素個數/每個緩衝區可容納的元素個數)+1
    // 如果剛好整除，會多配一個節點。
    size_type num_nodes = num_elements / buffer_size() + 1;

    // 一個 map 要管理幾個節點。最少 8 個，最多是 "所需節點數加 2"
    // (前後各預留一個，擴充時可用)。
    map_size = max(initial_map_size(), num_nodes + 2);
    map = map_allocator::allocate(map_size);
    // 以上配置出一個 "具有 map_size 個節點" 的 map。

    // 以下令 nstart 和 nfinish 指向 map 所擁有之全部節點的最中央區段。
    // 保持在最中央，可使頭尾兩端的擴充能量一樣大。每個節點可對應一個緩衝區。
    map_pointer nstart = map + (map_size - num_nodes) / 2;
    map_pointer nfinish = nstart + num_nodes - 1;

    map_pointer cur;
    __STL_TRY {
        // 為 map 內的每個現用節點配置緩衝區。所有緩衝區加起來就是 deque 的
        // 可用空間 (最後一個緩衝區可能留有一些餘裕)。
        for (cur = nstart; cur <= nfinish; ++cur)
            *cur = allocate_node();
    }
    catch(...) {
        // "commit or rollback" 語意：若非全部成功，就一個不留。
        ...
    }
}
```

```

    }

    // 為 deque 內的兩個迭代器 start 和 end 設定正確內容。
    start.set_node(nstart);
    finish.set_node(nfinish);
    start.cur = start.first;    // first, cur 都是 public
    finish.cur = finish.first + num_elements % buffer_size();
    // 前面說過，如果剛好整除，會多配一個節點。
    // 此時即令 cur 指向這多配的一個節點（所對映之緩衝區）的起頭處。
}

```

接下來範例程式以註標運算子為每個元素重新設值，然後在尾端安插三個新元素：

```

for(int i=0; i<ideq.size(); ++i)
    ideq[i] = i;

for(int i=0; i<3; i++)
    ideq.push_back(i);

```

由於此時最後一個緩衝區仍有 4 個備用元素空間，所以不會引起緩衝區的再配置。

此時的 deque 狀態如圖 4-13。

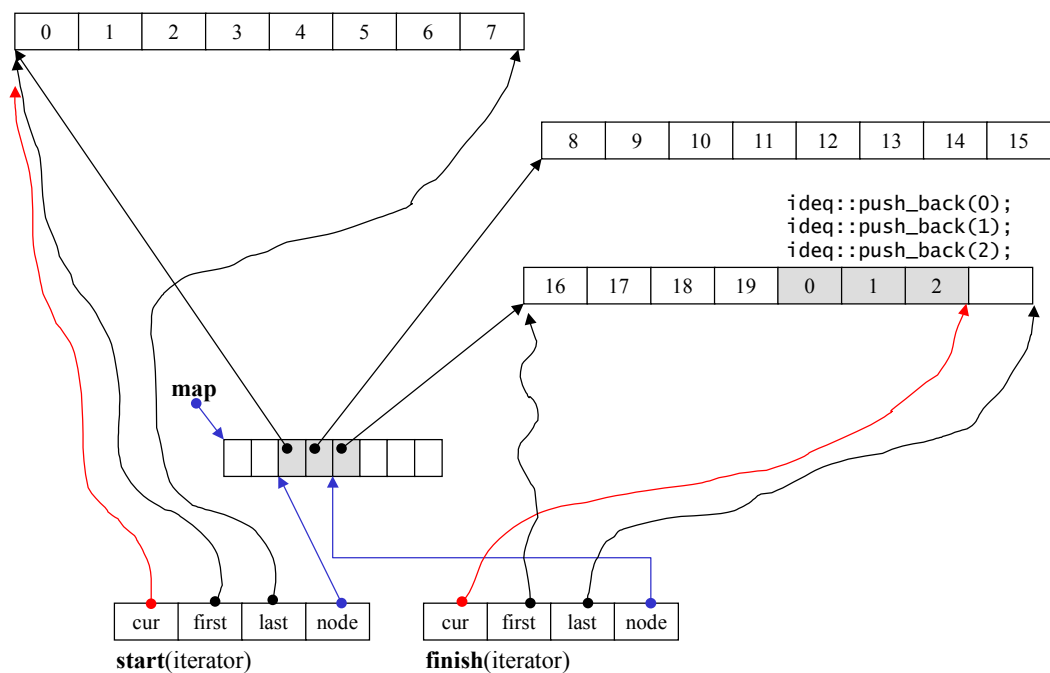


圖 4-13 延續圖 4-12 的狀態，將每個元素重新設值，並在尾端新增 3 個元素。

以下是 `push_back()` 函式內容：

```
public:                                // push_* and pop_*
void push_back(const value_type& t) {
    if (finish.cur != finish.last - 1)
        // 最後緩衝區尚有一個以上的備用空間
        construct(finish.cur, t); // 直接在備用空間上建構元素
        ++finish.cur;           // 調整最後緩衝區的使用狀態
    }
    else // 最後緩衝區已無（或只剩一個）元素備用空間。
        push_back_aux(t);
}
```

現在，如果再新增一個新元素於尾端：

```
ideq.push_back(3);
```

由於尾端只剩一個元素備用空間，於是 `push_back()` 呼叫 `push_back_aux()`，先配置一整塊新的緩衝區，再設定新元素內容，然後更改迭代器 `finish` 的狀態：

```
// 只有當 finish.cur == finish.last - 1 時才會被呼叫。
// 也就是說只有當最後一個緩衝區只剩一個備用元素空間時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_back_aux(const value_type& t) {
    value_type t_copy = t;
    reserve_map_at_back(); // 若符合某種條件則必須重換一個 map
    *(finish.node + 1) = allocate_node(); // 配置一個新節點（緩衝區）
    __STL_TRY {
        construct(finish.cur, t_copy); // 針對標的元素設值
        finish.set_node(finish.node + 1); // 改變 finish，令其指向新節點
        finish.cur = finish.first; // 設定 finish 的狀態
    }
    __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}
```

現在，`deque` 的狀態如圖 4-14。

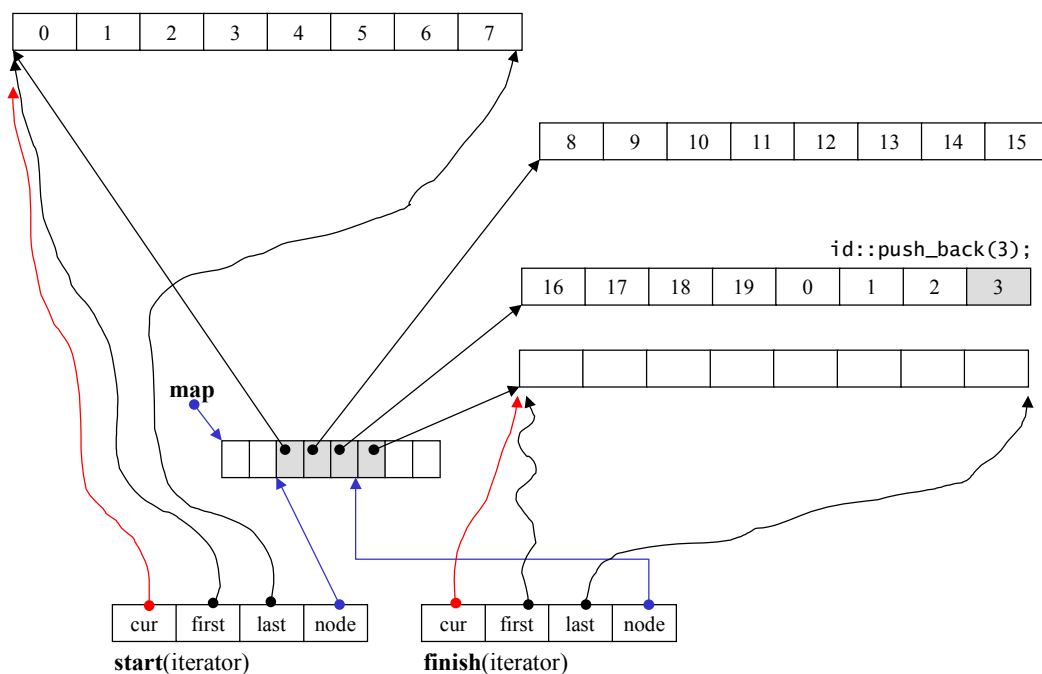


圖 4-14 延續圖 4-13 的狀態，在尾端再加一個元素，於是引發新緩衝區的配置，同時也造成迭代器 `finish` 的狀態改變。`map` 大小為 8（初始值），目前用了 4 個節點。

接下來範例程式在 `deque` 的前端安插一個新元素：

```
ideq.push_front(99);
```

`push_front()` 函式動作如下：

```
public:                                     // push_* and pop_*
void push_front(const value_type& t) {
    if (start.cur != start.first) { // 第一緩衝區尚有備用空間
        construct(start.cur - 1, t); // 直接在備用空間上建構元素
        --start.cur;                // 調整第一緩衝區的使用狀態
    }
    else // 第一緩衝區已無備用空間
        push_front_aux(t);
}
```

由於目前狀態下，第一緩衝區並無備用空間，所以呼叫 `push_front_aux()`：

```
// 只有當 start.cur == start.first 時才會被呼叫。
// 也就是說只有當第一個緩衝區沒有任何備用元素時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_front_aux(const value_type& t)
{
    value_type t_copy = t;
    reserve_map_at_front();           // 若符合某種條件則必須重換一個 map
    *(start.node - 1) = allocate_node(); // 配置一個新節點 (緩衝區)
    __STL_TRY {
        start.set_node(start.node - 1); // 改變 start，令其指向新節點
        start.cur = start.last - 1;    // 設定 start 的狀態
        construct(start.cur, t_copy);  // 針對標的元素設值
    }
    catch(...) {
        // "commit or rollback" 語意：若非全部成功，就一個不留。
        start.set_node(start.node + 1);
        start.cur = start.first;
        deallocate_node(*(start.node - 1));
        throw;
    }
}
```

此函式一開始即呼叫 `reserve_map_at_front()`，後者用來判斷是否需要擴充 `map`，如有需要就付諸行動。稍後我會呈現 `reserve_map_at_front()` 的函式內容。目前的狀態不需要重新整治 `map`，所以後繼流程便配置了一塊新緩衝區並直接將節點安置於現有的 `map` 上，然後設定新元素，然後改變迭代器 `start` 的狀態，如圖 4-15。

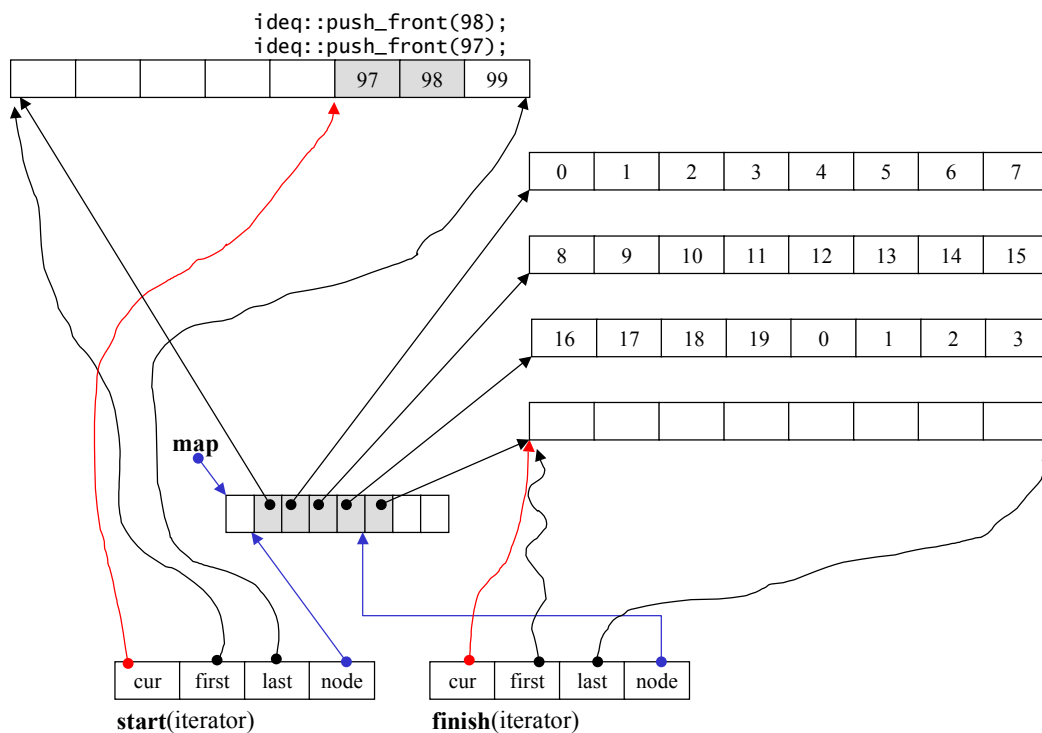


圖 4-16 延續圖 4-15 的狀態，在最前端再加兩個元素。由於第一緩衝區尚有備用空間，因此直接取用備用空間來建構新元素即可。

圖 4-12 至圖 4-16 的連環圖解，已經充份展示了 `deque` 容器的空間運用策略。讓我們回頭看看一個懸而未解的問題：什麼時候 `map` 需要重新整治？這個問題的判斷由 `reserve_map_at_back()` 和 `reserve_map_at_front()` 進行，實際動作則由 `reallocate_map()` 執行：

```
void reserve_map_at_back (size_type nodes_to_add = 1) {
    if (nodes_to_add + 1 > map_size - (finish.node - map))
        // 如果 map 尾端的節點備用空間不足
        // 符合以上條件則必須重換一個 map (配置更大的，拷貝原來的，釋放原來的)
        reallocate_map(nodes_to_add, false);
}

void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
        // 如果 map 前端的節點備用空間不足
```

```

        // 符合以上條件則必須重換一個 map (配置更大的, 拷貝原來的, 釋放原來的)
        reallocate_map(nodes_to_add, true);
    }

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::reallocate_map(size_type nodes_to_add,
                                              bool add_at_front) {
    size_type old_num_nodes = finish.node - start.node + 1;
    size_type new_num_nodes = old_num_nodes + nodes_to_add;

    map_pointer new_nstart;
    if (map_size > 2 * new_num_nodes) {
        new_nstart = map + (map_size - new_num_nodes) / 2
            + (add_at_front ? nodes_to_add : 0);
        if (new_nstart < start.node)
            copy(start.node, finish.node + 1, new_nstart);
        else
            copy_backward(start.node, finish.node + 1, new_nstart + old_num_nodes);
    }
    else {
        size_type new_map_size = map_size + max(map_size, nodes_to_add) + 2;
        // 配置一塊空間, 準備給新 map 使用。
        map_pointer new_map = map_allocator::allocate(new_map_size);
        new_nstart = new_map + (new_map_size - new_num_nodes) / 2
            + (add_at_front ? nodes_to_add : 0);
        // 把原 map 內容拷貝過來。
        copy(start.node, finish.node + 1, new_nstart);
        // 釋放原 map
        map_allocator::deallocate(map, map_size);
        // 設定新 map 的起始位址與大小
        map = new_map;
        map_size = new_map_size;
    }

    // 重新設定迭代器 start 和 finish
    start.set_node(new_nstart);
    finish.set_node(new_nstart + old_num_nodes - 1);
}

```

4.4.6 deque 的元素操作

pop_back, pop_front, clear, erase, insert

deque 所提供的元素操作動作很多，無法在有限的篇幅中一一講解 — 其實也沒有這種必要。以下我只挑選幾個 member functions 做為示範說明。

前述測試程式曾經以泛型演算法 find() 尋找 deque 的某個元素：


```
deque<int,alloc,32>::iterator itr;
itr = find(ideq.begin(), ideq.end(), 99);
```

當 `find()` 動作完成，迭代器 `itr` 狀態如圖 4-17 所示。下面這兩個動作輸出相同的結果，印證我們對 `deque` 迭代器的認識。

```
cout << *itr << endl;           // 99
cout << *(itr.cur) << endl;     // 99
```

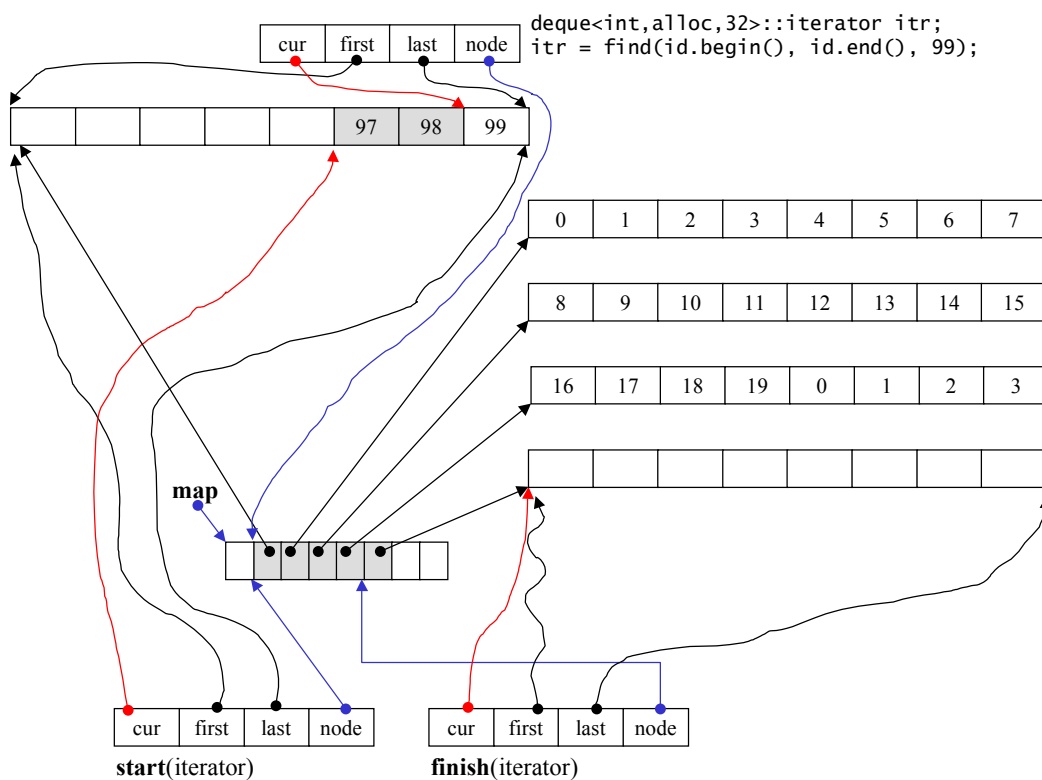


圖 4-17 延續圖 4-16 的狀態，以 `find()` 尋找數值為 99 的元素。此函式將傳回一個迭代器，指向第一個符合條件的元素。注意，該迭代器的四個欄位都必須有正確的設定。

前一節已經展示過 `push_back()` 和 `push_front()` 的實作內容，現在我舉對應的 `pop_back()` 和 `pop_front()` 為例。所謂 `pop`，是將元素拿掉。無論從 `deque` 的最前端或最尾端取元素，都需考量在某種條件下，將緩衝區釋放掉：

```
void pop_back() {
    if (finish.cur != finish.first) {
        // 最後緩衝區有一個（或更多）元素
        --finish.cur;           // 調整指標，相當於排除了最後元素
        destroy(finish.cur);    // 將最後元素解構
    }
    else
        // 最後緩衝區沒有任何元素
        pop_back_aux();         // 這裡將進行緩衝區的釋放工作
}

// 只有當 finish.cur == finish.first 時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_back_aux() {
    deallocate_node(finish.first); // 釋放最後一個緩衝區
    finish.set_node(finish.node - 1); // 調整 finish 的狀態，使指向
    finish.cur = finish.last - 1;    // 上一個緩衝區的最後一個元素
    destroy(finish.cur);             // 將該元素解構。
}

void pop_front() {
    if (start.cur != start.last - 1) {
        // 第一緩衝區有一個（或更多）元素
        destroy(start.cur); // 將第一元素解構
        ++start.cur;        // 調整指標，相當於排除了第一元素
    }
    else
        // 第一緩衝區僅有一個元素
        pop_front_aux();    // 這裡將進行緩衝區的釋放工作
}

// 只有當 start.cur == start.last - 1 時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
    destroy(start.cur); // 將第一緩衝區的第一個元素解構。
    deallocate_node(start.first); // 釋放第一緩衝區。
    start.set_node(start.node + 1); // 調整 start 的狀態，使指向
    start.cur = start.first;        // 下一個緩衝區的第一個元素。
}
```

下面這個例子是 `clear()`，用來清除整個 `deque`。請注意，`deque` 的最初狀態（無任何元素時）保有一個緩衝區，因此 `clear()` 完成之後回復初始狀態，也一樣要保留一個緩衝區：

```
// 注意，最終需要保留一個緩衝區。這是 deque 的策略，也是 deque 的初始狀態。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
    // 以下針對頭尾以外的每一個緩衝區（它們一定都是飽滿的）
    for (map_pointer node = start.node + 1; node < finish.node; ++node) {
        // 將緩衝區內的所有元素解構。注意，呼叫的是 destroy() 第二版本，見 2.2.3 節
        destroy(*node, *node + buffer_size());
        // 釋放緩衝區記憶體
        data_allocator::deallocate(*node, buffer_size());
    }

    if (start.node != finish.node) { // 至少有頭尾兩個緩衝區
        destroy(start.cur, start.last); // 將頭緩衝區的目前所有元素解構
        destroy(finish.first, finish.cur); // 將尾緩衝區的目前所有元素解構
        // 以下釋放尾緩衝區。注意，頭緩衝區保留。
        data_allocator::deallocate(finish.first, buffer_size());
    }
    else // 只有一個緩衝區
        destroy(start.cur, finish.cur); // 將此唯一緩衝區內的所有元素解構
        // 注意，並不釋放緩衝區空間。這唯一的緩衝區將保留。

    finish = start; // 調整狀態
}
```

下面這個例子是 `erase()`，用來清除某個元素：

```
// 清除 pos 所指的元素。pos 為清除點。
iterator erase(iterator pos) {
    iterator next = pos;
    ++next;
    difference_type index = pos - start; // 清除點之前的元素個數
    if (index < (size() >> 1)) { // 如果清除點之前的元素比較少，
        copy_backward(start, pos, next); // 就搬移清除點之前的元素
        pop_front(); // 搬移完畢，最前一個元素贅餘，去除之
    }
    else { // 清除點之後的元素比較少，
        copy(next, finish, pos); // 就搬移清除點之後的元素
        pop_back(); // 搬移完畢，最後一個元素贅餘，去除之
    }
    return start + index;
}
```

下面這個例子是 `erase()`，用來清除 `[first, last)` 區間內的所有元素：

```
template <class T, class Alloc, size_t BufSize>
deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::erase(iterator first, iterator last) {
    if (first == start && last == finish) { // 如果清除區間就是整個 deque
        clear(); // 直接呼叫 clear() 即可
        return finish;
    }
    else {
        difference_type n = last - first; // 清除區間的長度
        difference_type elems_before = first - start; // 清除區間前方的元素個數
        if (elems_before < (size() - n) / 2) { // 如果前方的元素比較少，
            copy_backward(start, first, last); // 向後搬移前方元素（覆蓋清除區間）
            iterator new_start = start + n; // 標記 deque 的新起點
            destroy(start, new_start); // 搬移完畢，將贅餘的元素解構
            // 以下將贅餘的緩衝區釋放
            for (map_pointer cur = start.node; cur < new_start.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            start = new_start; // 設定 deque 的新起點
        }
        else { // 如果清除區間後方的元素比較少
            copy(last, finish, first); // 向前搬移後方元素（覆蓋清除區間）
            iterator new_finish = finish - n; // 標記 deque 的新尾點
            destroy(new_finish, finish); // 搬移完畢，將贅餘的元素解構
            // 以下將贅餘的緩衝區釋放
            for (map_pointer cur = new_finish.node + 1; cur <= finish.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            finish = new_finish; // 設定 deque 的新尾點
        }
        return start + elems_before;
    }
}
```

本節要說明的最後一個例子是 `insert`。`deque` 為這個功能提供了許多版本，最基礎最重要的是以下版本，允許在某個點（之前）安插一個元素，並設定其值。

```
// 在 position 處安插一個元素，其值為 x
iterator insert(iterator position, const value_type& x) {
    if (position.cur == start.cur) { // 如果安插點是 deque 最前端
        push_front(x); // 交給 push_front 去做
        return start;
    }
    else if (position.cur == finish.cur) { // 如果安插點是 deque 最尾端
        push_back(x); // 交給 push_back 去做
        iterator tmp = finish;
        --tmp;
    }
}
```

```

        return tmp;
    }
    else {
        return insert_aux(position, x);    // 交給 insert_aux 去做
    }
}

template <class T, class Alloc, size_t BufSize>
typename deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::insert_aux(iterator pos, const value_type& x) {
    difference_type index = pos - start;    // 安插點之前的元素個數
    value_type x_copy = x;
    if (index < size() / 2) {                // 如果安插點之前的元素個數比較少
        push_front(front());                // 在最前端加入與第一元素同值的元素。
        iterator front1 = start;            // 以下標示記號，然後進行元素搬移...
        ++front1;
        iterator front2 = front1;
        ++front2;
        pos = start + index;
        iterator pos1 = pos;
        ++pos1;
        copy(front2, pos1, front1);        // 元素搬移
    }
    else {
        // 安插點之後的元素個數比較少
        push_back(back());                // 在最尾端加入與最後元素同值的元素。
        iterator back1 = finish;            // 以下標示記號，然後進行元素搬移...
        --back1;
        iterator back2 = back1;
        --back2;
        pos = start + index;
        copy_backward(pos, back2, back1);    // 元素搬移
    }
    *pos = x_copy;    // 在安插點上設定新值
    return pos;
}

```

4.5 stack

4.5.1 stack 概述

stack 是一種先進後出（First In Last Out，FILO）的資料結構。它只有一個出口，型式如圖 4-18。**stack** 允許新增元素、移除元素、取得最頂端元素。但除了最頂端外，沒有任何其他方法可以存取 **stack** 的其他元素。換言之 **stack** 不允許有走訪行為。

將元素推入 **stack** 的動作稱為 *push*，將元素推出 **stack** 的動作稱為 *pop*。

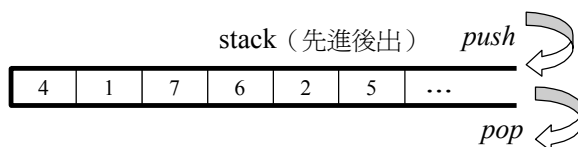


圖 4-18 stack 的結構

4.5.2 stack 定義式完整列表

以某種既有容器做為底部結構，將其介面改變，使符合「先進後出」的特性，形成一個 **stack**，是很容易做到的。**deque** 是雙向開口的資料結構，若以 **deque** 為底部結構並封閉其頭端開口，便輕而易舉地形成了一個 **stack**。因此，SGI STL 便以 **deque** 做為預設情況下的 **stack** 底部結構，**stack** 的實作因而非常簡單，源碼十分簡短，本處完整列出。

由於 **stack** 係以底部容器完成其所有工作，而具有這種「修改某物介面，形成另一種風貌」之性質者，稱為 **adapter**（配接器），因此 STL **stack** 往往不被歸類為 **container**（容器），而被歸類為 **container adapter**。

```
template <class T, class Sequence = deque<T> >
class stack {
    // 以下的 __STL_NULL_TMPL_ARGS 會開展為 <>，見 1.9.1 節
    friend bool operator== __STL_NULL_TMPL_ARGS (const stack&, const stack&);
    friend bool operator< __STL_NULL_TMPL_ARGS (const stack&, const stack&);
public:
```

```

typedef typename Sequence::value_type value_type;
typedef typename Sequence::size_type size_type;
typedef typename Sequence::reference reference;
typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c;    // 底層容器
public:
    // 以下完全利用 Sequence c 的操作，完成 stack 的操作。
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    // deque 是兩頭可進出，stack 是末端進，末端出（所以後進者先出）。
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};

template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y)
{
    return x.c == y.c;
}

template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x, const stack<T, Sequence>& y)
{
    return x.c < y.c;
}

```

4.5.3 stack 沒有迭代器

stack 所有元素的進出都必須符合「先進後出」的條件，只有 **stack** 頂端的元素，才有機會被外界取用。**stack** 不提供走訪功能，也不提供迭代器。

4.5.4 以 list 做為 stack 的底層容器

除了 **deque** 之外，**list** 也是雙向開口的資料結構。上述 **stack** 源碼中使用的底層容器的函式有 **empty**, **size**, **back**, **push_back**, **pop_back**，凡此種種 **list** 都具備。因此若以 **list** 為底部結構並封閉其頭端開口，一樣能夠輕易形成一個 **stack**。下面是作法示範。

```

// file : 4stack-test.cpp
#include <stack>
#include <list>
#include <iostream>

```

```

#include <algorithm>
using namespace std;

int main()
{
    stack<int, list<int> > istack;
    istack.push(1);
    istack.push(3);
    istack.push(5);
    istack.push(7);

    cout << istack.size() << endl;    // 4
    cout << istack.top() << endl;    // 7

    istack.pop(); cout << istack.top() << endl;    // 5
    istack.pop(); cout << istack.top() << endl;    // 3
    istack.pop(); cout << istack.top() << endl;    // 1
    cout << istack.size() << endl;    // 1
}

```

4.6 queue

4.6.1 queue 概述

queue 是一種先進先出（First In First Out, FIFO）的資料結構。它有兩個出口，型式如圖 4-19。**queue** 允許新增元素、移除元素、從最底端加入元素、取得最頂端元素。但除了最底端可以加入、最頂端可以取出，沒有任何其他方法可以存取 **queue** 的其他元素。換言之 **queue** 不允許有走訪行為。

將元素推入 **queue** 的動作稱為 *push*，將元素推出 **queue** 的動作稱為 *pop*。

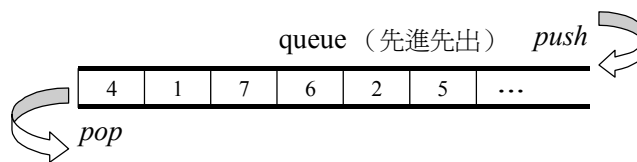


圖 4-19 queue 的結構

4.6.2 queue 定義式完整列表

以某種既有容器為底部結構，將其介面改變，使符合「先進先出」的特性，形成一個 `queue`，是很容易做到的。`deque` 是雙向開口的資料結構，若以 `deque` 為底部結構並封閉其底端的出口和前端的入口，便輕而易舉地形成了一個 `queue`。因此，SGI STL 便以 `deque` 做為預設情況下的 `queue` 底部結構，`queue` 的實作因而非常簡單，源碼十分簡短，本處完整列出。

由於 `queue` 係以底部容器完成其所有工作，而具有這種「修改某物介面，形成另一種風貌」之性質者，稱為 `adapter`（配接器），因此 STL `queue` 往往不被歸類為 `container`（容器），而被歸類為 `container adapter`。

```
template <class T, class Sequence = deque<T> >
class queue {
    // 以下的 __STL_NULL_TMPL_ARGS 會開展為 <>, 見 1.9.1 節
    friend bool operator== __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);
    friend bool operator< __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c;    // 底層容器
public:
    // 以下完全利用 Sequence c 的操作，完成 queue 的操作。
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    const_reference front() const { return c.front(); }
    reference back() { return c.back(); }
    const_reference back() const { return c.back(); }
    // deque 是兩頭可進出，queue 是末端進，前端出（所以先進者先出）。
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};

template <class T, class Sequence>
bool operator==(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
    return x.c == y.c;
}
```

```
template <class T, class Sequence>
bool operator<(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
    return x.c < y.c;
}
```

4.6.3 queue 沒有迭代器

`queue` 所有元素的進出都必須符合「先進先出」的條件，只有 `queue` 頂端的元素，才有機會被外界取用。`queue` 不提供走訪功能，也不提供迭代器。

4.6.4 以 `list` 做為 `queue` 的基層容器

除了 `deque` 之外，`list` 也是雙向開口的資料結構。上述 `queue` 源碼中使用的基層容器的函式有 `empty`, `size`, `back`, `push_back`, `pop_back`，凡此種種 `list` 都具備。因此若以 `list` 為底部結構並封閉其頭端開口，一樣能夠輕易形成一個 `queue`。下面是作法示範。

```
// file : 4queue-test.cpp
#include <queue>
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    queue<int, list<int> > iqueue;
    iqueue.push(1);
    iqueue.push(3);
    iqueue.push(5);
    iqueue.push(7);

    cout << iqueue.size() << endl;    // 4
    cout << iqueue.front() << endl;  // 1

    iqueue.pop(); cout << iqueue.front() << endl; // 3
    iqueue.pop(); cout << iqueue.front() << endl; // 5
    iqueue.pop(); cout << iqueue.front() << endl; // 7
    cout << iqueue.size() << endl;    // 1
}
```

4.7 heap (隱性表述, implicit representation)

4.7.1 heap 概述

heap 並不歸屬於 STL 容器組件，它是個幕後英雄，扮演 `priority queue` (4.8 節) 的推手。顧名思義，`priority queue` 允許使用者以任何次序將任何元素推入容器內，但取出時一定是從優先權最高（也就是數值最高）之元素開始取。`binary max heap` 正是具有這樣的特性，適合做為 `priority queue` 的底層機制。

讓我們做點分析。如果使用 4.3 節的 `list` 做為 `priority queue` 的底層機制，元素安插動作可享常數時間。但是要找到 `list` 中的極值，卻需要對整個 `list` 進行線性掃描。我們也可以改個作法，讓元素安插前先經過排序這一關，使得 `list` 的元素值總是由小到大（或由大到小），但這麼一來，收之東隅卻失之桑榆：雖然取得極值以及元素刪除動作達到最高效率，元素的安插卻只有線性表現。

比較麻辣的作法是以 `binary search tree` (如 5.1 節的 `RB-tree`) 做為 `priority queue` 的底層機制。這麼一來元素的安插和極值的取得就有 $O(\log N)$ 的表現。但殺雞用牛刀，未免小題大作，一來 `binary search tree` 的輸入需要足夠的隨機性，二來 `binary search tree` 並不容易實作。`priority queue` 的複雜度，最好介於 `queue` 和 `binary search tree` 之間，才算適得其所。`binary heap` 便是這種條件下的適當候選人。

所謂 `binary heap` 就是一種 `complete binary tree` (完全二元樹)²，也就是說，整棵 `binary tree` 除了最底層的葉節點(s) 之外，是填滿的，而最底層的葉節點(s) 由左至右又不得有空隙。圖 4-20 是一個 `complete binary tree`。

² 關於 `tree` 的種種，5.1 節會有更多介紹。

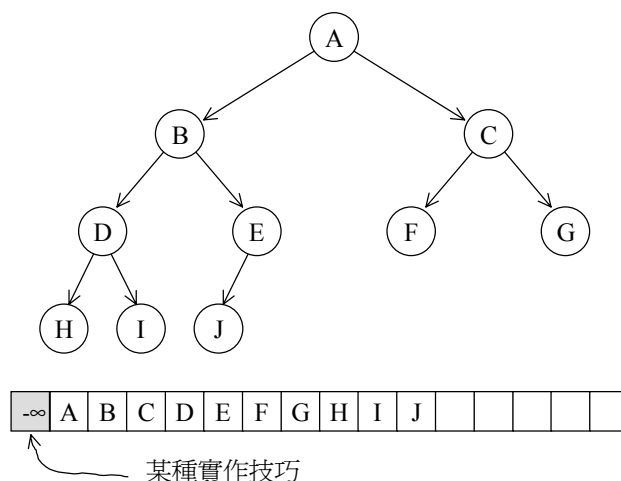


圖 4-20 一個完全二元樹（complete binary tree），及其 array 表述式

complete binary tree 整棵樹內沒有任何節點漏洞，這帶來一個極大好處：我們可以利用 **array** 來儲存所有節點。假設動用一個小技巧³，將 **array** 的 #0 元素保留（或設為無限大值或無限小值），那麼當 **complete binary tree** 中的某個節點位於 **array** 的 i 處，其左子節點必位於 **array** 的 $2i$ 處，其右子節點必位於 **array** 的 $2i+1$ 處，其父節點必位於「 $i/2$ 」處（此處的「 $\lceil$ 」權且代表高斯符號，取其整數）。通過這麼簡單的位置規則，**array** 可以輕易實作出 **complete binary tree**。這種以 **array** 表述 **tree** 的方式，我們稱為隱式表述法（implicit representation）。

這麼一來，我們需要的工具就很簡單了：一個 **array** 和一組 **heap** 演算法（用來安插元素、刪除元素、取極值、將某一整組數據排列成一個 **heap**）。**array** 的缺點是無法動態改變大小，而 **heap** 卻需要這項功能，因此以 **vector**（4.2 節）代替 **array** 是更好的選擇。

根據元素排列方式，**heap** 可分為 **max-heap** 和 **min-heap** 兩種，前者每個節點的鍵值（*key*）都大於或等於其子節點鍵值，後者的每個節點鍵值（*key*）都小於或等

³ SGI STL 提供的 **heap** 並未使用此一小技巧。計算左右子節點以及父節點的方式，因而略有不同。詳見稍後的源碼及解說。

於其子節點鍵值。因此，**max-heap** 的最大值在根節點，並總是位於底層 **array** 或 **vector** 的起頭處；**min-heap** 的最小值在根節點，亦總是位於底層 **array** 或 **vector** 的起頭處。STL 供應的是 **max-heap**，因此以下我說 **heap** 時，指的是 **max-heap**。

4.7.2 heap 演算法

push_heap 演算法

圖 4-21 是 **push_heap** 演算法的實際操演情況。為了滿足 **complete binary tree** 的條件，新加入的元素一定要放在最下一層做為葉節點，並填補在由左至右的第一個空格，也就是把新元素安插在底層 **vector** 的 **end()** 處。

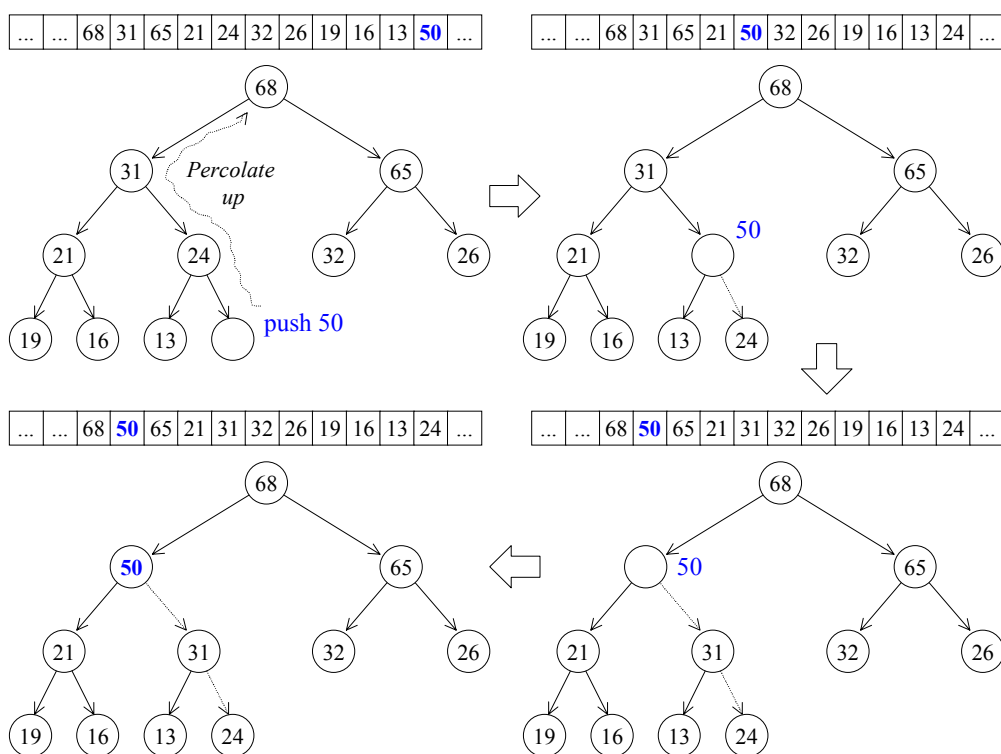


圖 4-21 push_heap 演算法

新元素是否適合於其現有位置呢？為滿足 max-heap 的條件（每個節點的鍵值都大於或等於其子節點鍵值），我們執行一個所謂的 **percolate up**（上溯）程序：將新節點拿來與其父節點比較，如果其鍵值（*key*）比父節點大，就父子對換位置。如此一直上溯，直到不需對換或直到根節點為止。

下面便是 `push_heap` 演算法的實作細節。此函式接受兩個迭代器，用來表現一個 heap 底部容器（**vector**）的頭尾，新元素並且已經安插到底部容器的最尾端。如果不符合這兩個條件，`push_heap` 的執行結果未可預期。

```
template <class RandomAccessIterator>
inline void push_heap(RandomAccessIterator first,
                      RandomAccessIterator last) {
    // 注意，此函式被呼叫時，新元素應已置於底部容器的最尾端。
    __push_heap_aux(first, last, distance_type(first),
                    value_type(first));
}

template <class RandomAccessIterator, class Distance, class T>
inline void __push_heap_aux(RandomAccessIterator first,
                           RandomAccessIterator last, Distance*, T*) {
    __push_heap(first, Distance((last - first) - 1), Distance(0),
                T(*(last - 1)));
    // 以上係根據 implicit representation heap 的結構特性：新值必置於底部
    // 容器的最尾端，此即第一個洞號：(last-first)-1。
}

// 以下這組 push_back() 不允許指定「大小比較標準」
template <class RandomAccessIterator, class Distance, class T>
void __push_heap(RandomAccessIterator first, Distance holeIndex,
                 Distance topIndex, T value) {
    Distance parent = (holeIndex - 1) / 2; // 找出父節點
    while (holeIndex > topIndex && *(first + parent) < value) {
        // 當尚未到達頂端，且父節點小於新值（於是不符合 heap 的次序特性）
        // 由於以上使用 operator<，可知 STL heap 是一種 max-heap（大者為父）。
        *(first + holeIndex) = *(first + parent); // 令洞值為父值
        holeIndex = parent; // percolate up：調整洞號，向上提昇至父節點。
        parent = (holeIndex - 1) / 2; // 新洞的父節點
    } // 持續至頂端，或滿足 heap 的次序特性為止。
    *(first + holeIndex) = value; // 令洞值為新值，完成安插動作。
}
```

pop_heap 演算法

圖 4-22 是 pop_heap 演算法的實際操演情況。既然身為 max-heap，最大值必然在根節點。pop 動作取走根節點（其實是移至底部容器 vector 的最後一個元素）之後，為了滿足 complete binary tree 的條件，必須將最下一層最右邊的葉節點拿掉，現在我們的任務是為這個被拿掉的節點找一個適當的位置。

為滿足 max-heap 的條件（每個節點的鍵值都大於或等於其子節點鍵值），我們執行一個所謂的 percolate down（下放）程序：將根節點（最大值被取走後，形成一個「洞」）填入上述那個失去生存空間的葉節點值，再將它拿來和其兩個子節點比較鍵值（key），並與較子節點對調位置。如此一直下放，直到這個「洞」的鍵值大於左右兩個子節點，或直到下放至葉節點為止。

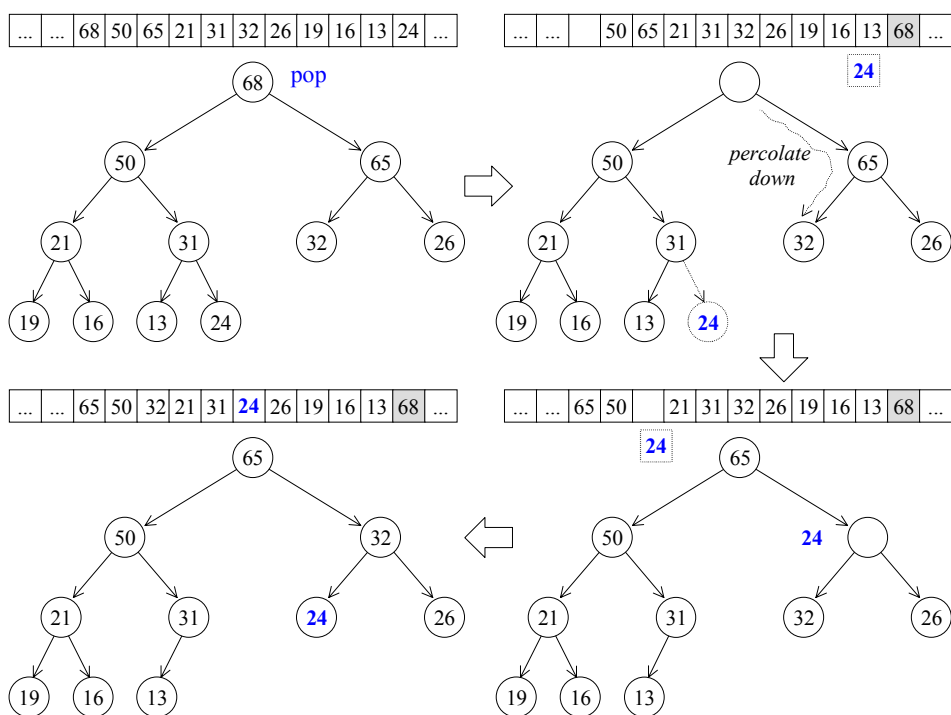


圖 4-22 pop_heap 演算法

下面便是 `pop_heap` 演算法的實作細節。此函式接受兩個迭代器，用來表現一個 `heap` 底部容器 (`vector`) 的頭尾。如果不符合這個條件，`pop_heap` 的執行結果未可預期。

```
template <class RandomAccessIterator>
inline void pop_heap(RandomAccessIterator first,
                    RandomAccessIterator last) {
    __pop_heap_aux(first, last, value_type(first));
}

template <class RandomAccessIterator, class T>
inline void __pop_heap_aux(RandomAccessIterator first,
                        RandomAccessIterator last, T*) {
    __pop_heap(first, last-1, last-1, T(*(last-1)),
              distance_type(first));
    // 以上，根據 implicit representation heap 的次序特性，pop 動作的結果
    // 應為底部容器的第一個元素。因此，首先設定欲調整值為尾值，然後將首值調至
    // 尾節點（所以以上將迭代器 result 設為 last-1）。然後重整 [first, last-1)，
    // 使之重新成一個合格的 heap。
}

// 以下這組 __pop_heap() 不允許指定「大小比較標準」
template <class RandomAccessIterator, class T, class Distance>
inline void __pop_heap(RandomAccessIterator first,
                    RandomAccessIterator last,
                    RandomAccessIterator result,
                    T value, Distance*) {
    *result = *first; // 設定尾值為首值，於是尾值即為欲求結果，
    // 可由客端稍後再以底層容器之 pop_back() 取出尾值。
    __adjust_heap(first, Distance(0), Distance(last - first), value);
    // 以上欲重新調整 heap，洞號為 0（亦即樹根處），欲調整值為 value（原尾值）。
}

// 以下這個 __adjust_heap() 不允許指定「大小比較標準」
template <class RandomAccessIterator, class Distance, class T>
void __adjust_heap(RandomAccessIterator first, Distance holeIndex,
                  Distance len, T value) {
    Distance topIndex = holeIndex;
    Distance secondChild = 2 * holeIndex + 2; // 洞節點之右子節點
    while (secondChild < len) {
        // 比較洞節點之左右兩個子值，然後以 secondChild 代表較大子節點。
        if (*(first + secondChild) < *(first + (secondChild - 1)))
            secondChild--;
        // Percolate down: 令較大子值為洞值，再令洞號下移至較大子節點處。
        *(first + holeIndex) = *(first + secondChild);
        holeIndex = secondChild;
        // 找出新洞節點的右子節點
    }
}
```



```

        secondChild = 2 * (secondChild + 1);
    }
    if (secondChild == len) { // 沒有右子節點，只有左子節點
        // Percolate down: 令左子值為洞值，再令洞號下移至左子節點處。
        *(first + holeIndex) = *(first + (secondChild - 1));
        holeIndex = secondChild - 1;
    }
    // 將欲調整值填入目前的洞號內。注意，此時肯定滿足次序特性。
    // 依侯捷之見，下面直接改為 *(first + holeIndex) = value; 應該可以。
    __push_heap(first, holeIndex, topIndex, value);
}

```

注意，`pop_heap` 之後，最大元素只是被置放於底部容器的最尾端，尚未被取走。如果要取其值，可使用底部容器 (`vector`) 所提供的 `back()` 操作函式。如果要移除它，可使用底部容器 (`vector`) 所提供的 `pop_back()` 操作函式。

sort_heap 演算法

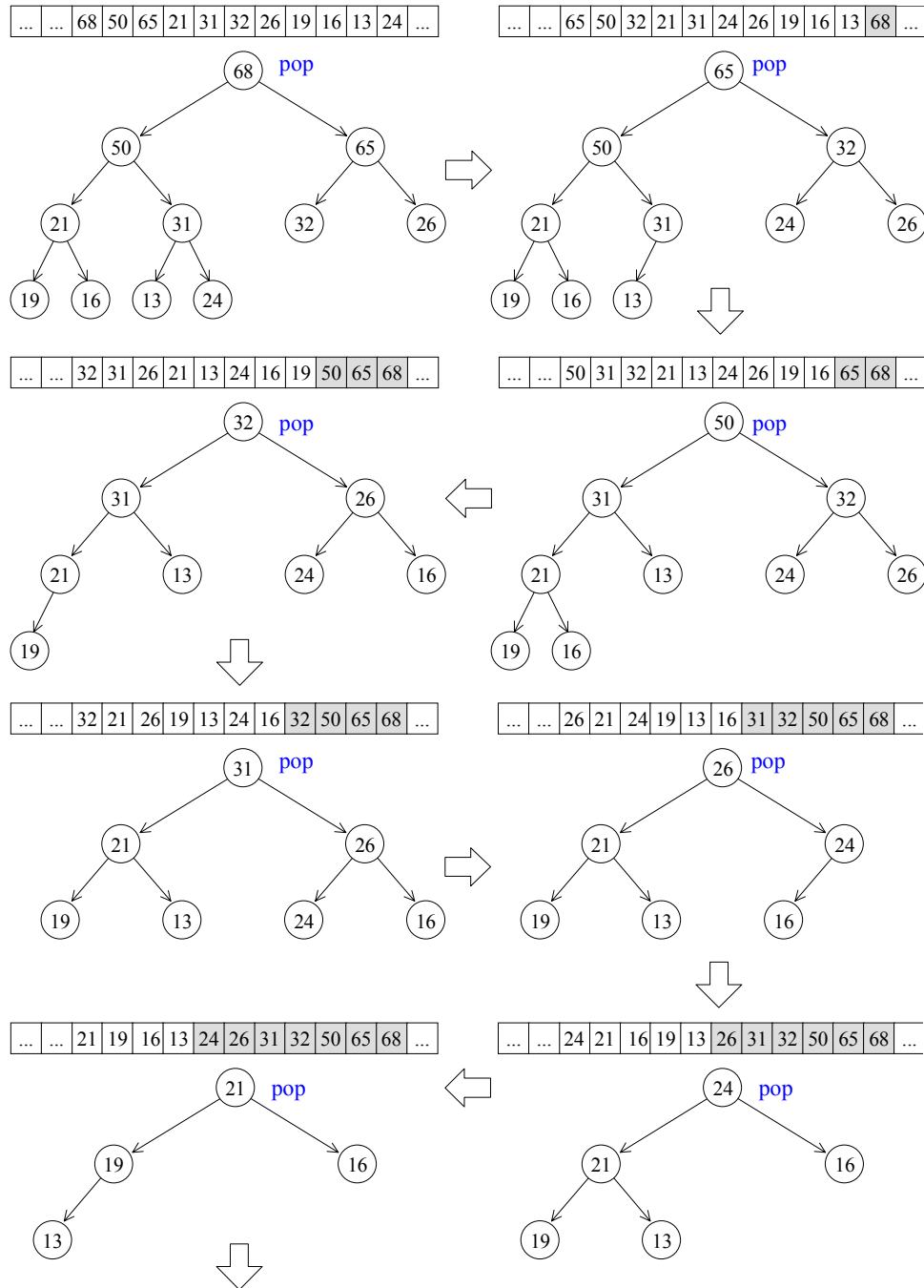
既然每次 `pop_heap` 可獲得 `heap` 之中鍵值最大的元素，如果持續對整個 `heap` 做 `pop_heap` 動作，每次將操作範圍從後向前縮減一個元素（因為 `pop_heap` 會把鍵值最大的元素放在底部容器的最尾端），當整個程序執行完畢，我們便有了一個遞增序列。圖 4-23 是 `sort_heap` 的實際操演情況。

下面是 `sort_heap` 演算法的實作細節。此函式接受兩個迭代器，用來表現一個 `heap` 底部容器 (`vector`) 的頭尾。如果不符合這個條件，`sort_heap` 的執行結果未可預期。注意，排序過後，原來的 `heap` 就不再是個合法的 `heap` 了。

```

// 以下這個 sort_heap() 不允許指定「大小比較標準」
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last) {
    // 以下，每執行一次 pop_heap()，極值（在 STL heap 中為極大值）即被放在尾端。
    // 扣除尾端再執行一次 pop_heap()，次極值又被放在新尾端。一直下去，最後即得
    // 排序結果。
    while (last - first > 1)
        pop_heap(first, last--); // 每執行 pop_heap() 一次，操作範圍即退縮一格。
}

```



續下頁

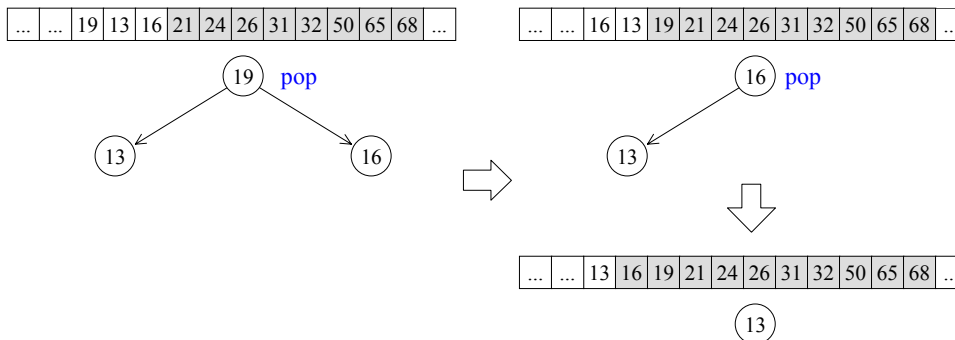


圖 4-23 `sort_heap` 演算法：不斷對 `heap` 做 `pop` 動作，便可達到排序效果

`make_heap` 演算法

這個演算法用來將一段現有的資料轉化為一個 `heap`。其主要依據就是 4.7.1 節提到的 `complete binary tree` 的隱式表述 (`implicit representation`)。

```
// 將 [first,last) 排列為一個 heap。
template <class RandomAccessIterator>
inline void make_heap(RandomAccessIterator first,
                      RandomAccessIterator last) {
    __make_heap(first, last, value_type(first), distance_type(first));
}

// 以下這組 make_heap() 不允許指定「大小比較標準」。
template <class RandomAccessIterator, class T, class Distance>
void __make_heap(RandomAccessIterator first,
                 RandomAccessIterator last, T*,
                 Distance*) {
    if (last - first < 2) return; // 如果長度為 0 或 1，不必重新排列。
    Distance len = last - first;
    // 找出第一個需要重排的子樹頭部，以 parent 標示出。由於任何葉節點都不需執行
    // perlocate down，所以有以下計算。parent 命名不佳，名為 holeIndex 更好。
    Distance parent = (len - 2)/2;

    while (true) {
        // 重排以 parent 為首的子樹。len 是為讓 __adjust_heap() 判斷操作範圍
        __adjust_heap(first, parent, len, T(*(first + parent)));
        if (parent == 0) return; // 走完根節點，就結束。
        parent--; // (即將重排之子樹的) 頭部向前一個節點
    }
}
```

4.7.3 heap 沒有迭代器

heap 的所有元素都必須遵循特別的 (complete binary tree) 排列規則, 所以 heap 不提供走訪功能, 也不提供迭代器。

4.7.4 heap 測試實例

```
// file: 4heap-test.cpp
#include <vector>
#include <iostream>
#include <algorithm> // heap algorithms
using namespace std;

int main()
{
    {
        // test heap (底層以 vector 完成)
        int ia[9] = {0,1,2,3,4,8,9,3,5};
        vector<int> ivec(ia, ia+9);

        make_heap(ivec.begin(), ivec.end());
        for(int i=0; i<ivec.size(); ++i)
            cout << ivec[i] << ' ';          // 9 5 8 3 4 0 2 3 1
        cout << endl;

        ivec.push_back(7);
        push_heap(ivec.begin(), ivec.end());
        for(int i=0; i<ivec.size(); ++i)
            cout << ivec[i] << ' ';          // 9 7 8 3 5 0 2 3 1 4
        cout << endl;

        pop_heap(ivec.begin(), ivec.end());
        cout << ivec.back() << endl;          // 9. return but no remove.
        ivec.pop_back();                     // remove last elem and no return

        for(int i=0; i<ivec.size(); ++i)
            cout << ivec[i] << ' ';          // 8 7 4 3 5 0 2 3 1
        cout << endl;

        sort_heap(ivec.begin(), ivec.end());
        for(int i=0; i<ivec.size(); ++i)
            cout << ivec[i] << ' ';          // 0 1 2 3 3 4 5 7 8
        cout << endl;
    }
}
```

```
{
// test heap (底層以 array 完成)
int ia[9] = {0,1,2,3,4,8,9,3,5};
make_heap(ia, ia+9);
// array 無法動態改變大小，因此不可以對滿載的 array 做 push_heap() 動作。
// 因為那得先在 array 尾端增加一個元素。

sort_heap(ia, ia+9);
for(int i=0; i<9; ++i)
    cout << ia[i] << ' ';           // 0 1 2 3 3 4 5 8 9
cout << endl;
// 經過排序之後的 heap，不再是個合法的 heap

// 重新再做一個 heap
make_heap(ia, ia+9);
pop_heap(ia, ia+9);
cout << ia[8] << endl;    // 9
}

{
// test heap (底層以 array 完成)
int ia[6] = {4,1,7,6,2,5};
make_heap(ia, ia+6);
for(int i=0; i<6; ++i)
    cout << ia[i] << ' ';           // 7 6 5 1 2 4
cout << endl;
}
}
```

4.8 priority_queue

4.8.1 priority_queue 概述

顧名思義，`priority_queue` 是一個擁有權值觀念的 `queue`，它允許加入新元素、移除舊元素，審視元素值等功能。由於這是一個 `queue`，所以只允許在底端加入元素，並從頂端取出元素，除此之外別無其他存取元素的途徑。

`priority_queue` 帶有權值觀念，其內的元素並非依照被推入的次序排列，而是自動依照元素的權值排列（通常權值以實值表示）。權值最高者，排在最前面。

預設情況下 `priority_queue` 係利用一個 `max-heap` 完成，後者是一個以 `vector` 表現的 `complete binary tree`（4.7 節）。`max-heap` 可以滿足 `priority_queue` 所需要的「依權值高低自動遞增排序」的特性。

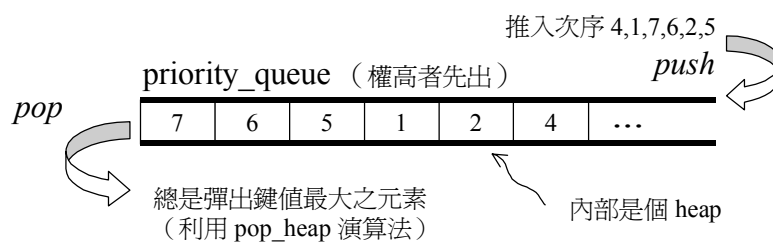


圖 4-24 priority queue

4.8.2 priority_queue 定義式完整列表

由於 `priority_queue` 完全以底部容器為根據，再加上 `heap` 處理規則，所以其實作非常簡單。預設情況下是以 `vector` 為底部容器。源碼很簡短，此處完整列出。

`queue` 以底部容器完成其所有工作。具有這種「修改某物介面，形成另一種風貌」之性質者，稱為 `adapter`（配接器），因此 STL `priority_queue` 往往不被歸類為 `container`（容器），而被歸類為 `container adapter`。

```

template <class T, class Sequence = vector<T>,
          class Compare = less<typename Sequence::value_type> >
class priority_queue {
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c;          // 底層容器
    Compare comp;         // 元素大小比較標準
public:
    priority_queue() : c() {}
    explicit priority_queue(const Compare& x) : c(), comp(x) {}

    // 以下用到的 make_heap(), push_heap(), pop_heap() 都是泛型演算法
    // 注意，任一個建構式都立刻於底層容器內產生一個 implicit representation heap。
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last, const Compare& x)
        : c(first, last), comp(x) { make_heap(c.begin(), c.end(), comp); }
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last)
        : c(first, last) { make_heap(c.begin(), c.end(), comp); }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    const_reference top() const { return c.front(); }
    void push(const value_type& x) {
        __STL_TRY {
            // push_heap 是泛型演算法，先利用底層容器的 push_back() 將新元素
            // 推入末端，再重排 heap。見 C++ Primer p.1195。
            c.push_back(x);
            push_heap(c.begin(), c.end(), comp); // push_heap 是泛型演算法
        }
        __STL_UNWIND(c.clear());
    }
    void pop() {
        __STL_TRY {
            // pop_heap 是泛型演算法，從 heap 內取出一個元素。它並不是真正將元素
            // 彈出，而是重排 heap，然後再以底層容器的 pop_back() 取得被彈出
            // 的元素。見 C++ Primer p.1195。
            pop_heap(c.begin(), c.end(), comp);
            c.pop_back();
        }
        __STL_UNWIND(c.clear());
    }
};

```

4.8.3 priority_queue 沒有迭代器

`priority_queue` 的所有元素，進出都有一定的規則，只有 `queue` 頂端的元素（權值最高者），才有機會被外界取用。`priority_queue` 不提供走訪功能，也不提供迭代器。

4.8.4 priority_queue 測試實例

```
// file: 4pqueue-test.cpp
#include <queue>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    // test priority queue...
    int ia[9] = {0,1,2,3,4,8,9,3,5};
    priority_queue<int> ipq(ia, ia+9);
    cout << "size=" << ipq.size() << endl;           // size=9

    for(int i=0; i<ipq.size(); ++i)
        cout << ipq.top() << ' ';                   // 9 9 9 9 9 9 9 9 9
    cout << endl;

    while(!ipq.empty()) {
        cout << ipq.top() << ' ';                     // 9 8 5 4 3 3 2 1 0
        ipq.pop();
    }
    cout << endl;
}
```


4.9 `slist`

4.9.1 `slist` 概述

STL `list` 是個雙向串列 (double linked list)。SGI STL 另提供了一個單向串列 (single linked list)，名為 `slist`。這個容器並不在標準規格之內，不過多做一些剖析，多看多學一些實作技巧也不錯，所以我把它納入本書範圍。

`slist` 和 `list` 的主要差別在於，前者的迭代器屬於單向的 *Forward Iterator*，後者的迭代器屬於雙向的 *Bidirectional Iterator*。為此，`slist` 的功能自然也就受到許多限制。不過，單向串列所耗用的空間更小，某些動作更快，不失為另一種選擇。

`slist` 和 `list` 共同具有的一個相同特色是，它們的安插 (`insert`)、移除 (`erase`)、接合 (`splice`) 等動作並不會造成原有的迭代器失效 (當然啦，指向被移除元素的那個迭代器，在移除動作發生之後肯定是會失效的)。

注意，根據 STL 的習慣，安插動作會將新元素安插於指定位置之前，而非之後。然而做為一個單向串列，`slist` 沒有任何方便的辦法可以回頭定出前一個位置，因此它必須從頭找起。換句話說，除了 `slist` 起始處附近的區域之外，在其他位置上採用 `insert` 或 `erase` 操作函式，都是不智之舉。這便是 `slist` 相較於 `list` 之下的大缺點。為此，`slist` 特別提供了 `insert_after()` 和 `erase_after()` 供彈性運用。

基於同樣的 (效率) 考量，`slist` 不提供 `push_back()`，只提供 `push_front()`。因此 `slist` 的元素次序會和元素安插進來的次序相反。

4.9.2 `slist` 的節點

`slist` 節點和其迭代器的設計，架構上比 `list` 複雜許多，運用了繼承關係，因此在型別轉換上有複雜的表現。這種設計方式在第 5 章 `RB-tree` 將再一次出現。圖 4-25 概述了 `slist` 節點和其迭代器的設計架構。

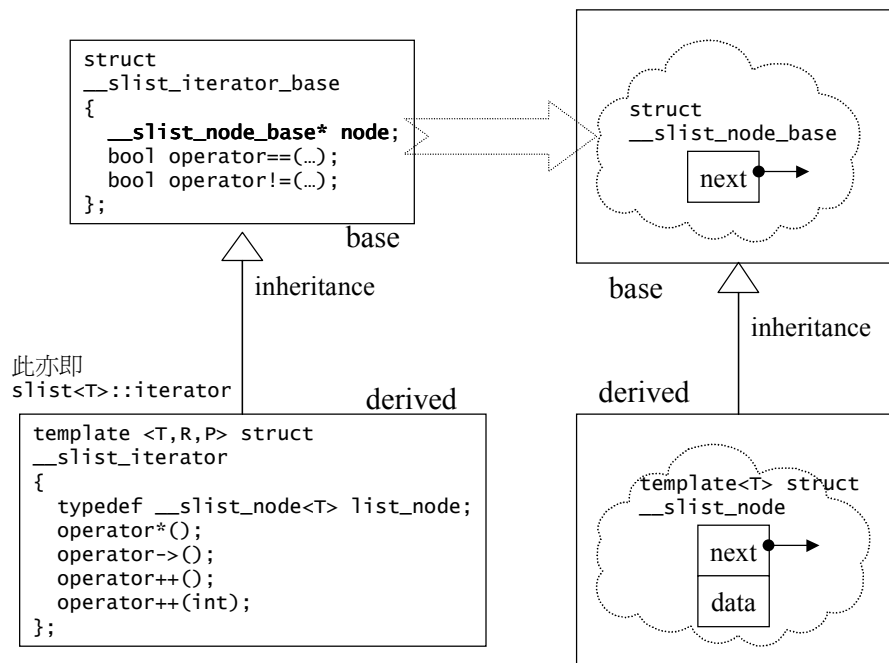


圖 4-25 slist 的節點和迭代器的設計架構

```

// 單向串列的節點基本結構
struct __slist_node_base
{
    __slist_node_base* next;
};

// 單向串列的節點結構
template <class T>
struct __slist_node : public __slist_node_base
{
    T data;
};

// 全域函式：已知某一節點，安插新節點於其後。
inline __slist_node_base* __slist_make_link(
    __slist_node_base* prev_node,
    __slist_node_base* new_node)
{
    // 令 new 節點的下一節點為 prev 節點的下一節點
    new_node->next = prev_node->next;
    prev_node->next = new_node;    // 令 prev 節點的下一節點指向 new 節點
}
  
```

```

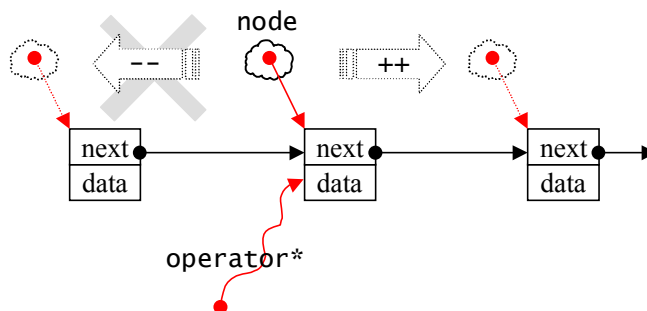
    return new_node;
}

// 全域函式：單向串列的大小（元素個數）
inline size_t __slist_size(__slist_node_base* node)
{
    size_t result = 0;
    for ( ; node != 0; node = node->next)
        ++result;    // 一個一個累計
    return result;
}

```

4.9.3 slist 的迭代器

slist 迭代器可以下圖表示：



實際構造如下。請注意它和節點的關係（見圖 4-25）。

```

// 單向串列的迭代器基本結構
struct __slist_iterator_base
{
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef forward_iterator_tag iterator_category; // 注意，單向

    __slist_node_base* node; // 指向節點基本結構

    __slist_iterator_base(__slist_node_base* x) : node(x) {}

    void incr() { node = node->next; } // 前進一個節點

    bool operator==(const __slist_iterator_base& x) const {
        return node == x.node;
    }
    bool operator!=(const __slist_iterator_base& x) const {

```

```

        return node != x.node;
    }
};

// 單向串列的迭代器結構
template <class T, class Ref, class Ptr>
struct __slist_iterator : public __slist_iterator_base
{
    typedef __slist_iterator<T, T&, T*>          iterator;
    typedef __slist_iterator<T, const T&, const T*> const_iterator;
    typedef __slist_iterator<T, Ref, Ptr>         self;

    typedef T value_type;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef __slist_node<T> list_node;

    __slist_iterator(list_node* x) : __slist_iterator_base(x) {}
    // 呼叫 slist<T>::end() 時會造成 __slist_iterator(0)，於是喚起上述函式。
    __slist_iterator() : __slist_iterator_base(0) {}
    __slist_iterator(const iterator& x) : __slist_iterator_base(x.node) {}

    reference operator*() const { return ((list_node*) node)->data; }
    pointer operator->() const { return &(operator*()); }

    self& operator++()
    {
        incr(); // 前進一個節點
        return *this;
    }
    self& operator++(int)
    {
        self tmp = *this;
        incr(); // 前進一個節點
        return tmp;
    }

    // 沒有實作 operator--，因為這是一個 forward iterator
};

```

注意，比較兩個 `slist` 迭代器是否等同時（例如我們常在迴圈中比較某個迭代器是否等同於 `slist.end()`），由於 `__slist_iterator` 並未對 `operator==` 實施多載化，所以會喚起 `__slist_iterator_base::operator==`。根據其中之定義，我們知道，兩個 `slist` 迭代器是否等同，視其 `__slist_node_base* node` 是否等同而定。

4.9.4 slist 的資料結構

下面是 `slist` 源碼摘要，我把焦點放在「單向串列之形成」的一些關鍵點上。

```
template <class T, class Alloc = alloc>
class slist
{
public:
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef __slist_iterator<T, T&, T*> iterator;
    typedef __slist_iterator<T, const T&, const T*> const_iterator;

private:
    typedef __slist_node<T> list_node;
    typedef __slist_node_base list_node_base;
    typedef __slist_iterator_base iterator_base;
    typedef simple_alloc<list_node, Alloc> list_node_allocator;

    static list_node* create_node(const value_type& x) {
        list_node* node = list_node_allocator::allocate(); // 配置空間
        __STL_TRY {
            construct(&node->data, x); // 建構元素
            node->next = 0;
        }
        __STL_UNWIND(list_node_allocator::deallocate(node));
        return node;
    }

    static void destroy_node(list_node* node) {
        destroy(&node->data); // 將元素解構
        list_node_allocator::deallocate(node); // 釋還空間
    }

private:
    list_node_base head; // 頭部。注意，它不是指標，是實物。

public:
    slist() { head.next = 0; }
    ~slist() { clear(); }

public:
```

```

iterator begin() { return iterator((list_node*)head.next); }
iterator end() { return iterator(0); }
size_type size() const { return __slist_size(head.next); }
bool empty() const { return head.next == 0; }

// 兩個 slist 互換：只要將 head 交換互指即可。
void swap(slist& L)
{
    list_node_base* tmp = head.next;
    head.next = L.head.next;
    L.head.next = tmp;
}

public:
    // 取頭部元素
    reference front() { return ((list_node*) head.next)->data; }

    // 從頭部安插元素（新元素成為 slist 的第一個元素）
    void push_front(const value_type& x) {
        __slist_make_link(&head, create_node(x));
    }

    // 注意，沒有 push_back()

    // 從頭部取走元素（刪除之）。修改 head。
    void pop_front() {
        list_node* node = (list_node*) head.next;
        head.next = node->next;
        destroy_node(node);
    }
    ...
};

```

4.9.5 slist 的元素操作

下面是一個小小練習：

```

// file: 4slist-test.cpp
#include <slist>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    int i;
    slist<int> islist;
    cout << "size=" << islist.size() << endl;    // size=0
}

```

```

islist.push_front(9);
islist.push_front(1);
islist.push_front(2);
islist.push_front(3);
islist.push_front(4);
cout << "size=" << islist.size() << endl;    // size=5

slist<int>::iterator ite =islist.begin();
slist<int>::iterator ite2=islist.end();
for(; ite != ite2; ++ite)
    cout << *ite << ' ';                    // 4 3 2 1 9
cout << endl;

ite = find(islist.begin(), islist.end(), 1);
if (ite!=0)
    islist.insert(ite, 99);

cout << "size=" << islist.size() << endl;    // size=6
cout << *ite << endl;                        // 1

ite =islist.begin();
ite2=islist.end();
for(; ite != ite2; ++ite)
    cout << *ite << ' ';                    // 4 3 2 99 1 9
cout << endl;

ite = find(islist.begin(), islist.end(), 3);
if (ite!=0)
    cout << *(islist.erase(ite)) << endl;    // 2

ite =islist.begin();
ite2=islist.end();
for(; ite != ite2; ++ite)
    cout << *ite << ' ';                    // 4 2 99 1 9
cout << endl;
}

```

首先依次序把元素 9,1,2,3,4 安插到 `slist`，實際結構呈現如圖 4-26。

接下來搜尋元素 1，並將新元素 99 安插進去，如圖 4-27。注意，新元素被安插在插入點（元素 1）的前面而不是後面。

接下來搜尋元素 3，並將該元素移除，如圖 4-28。

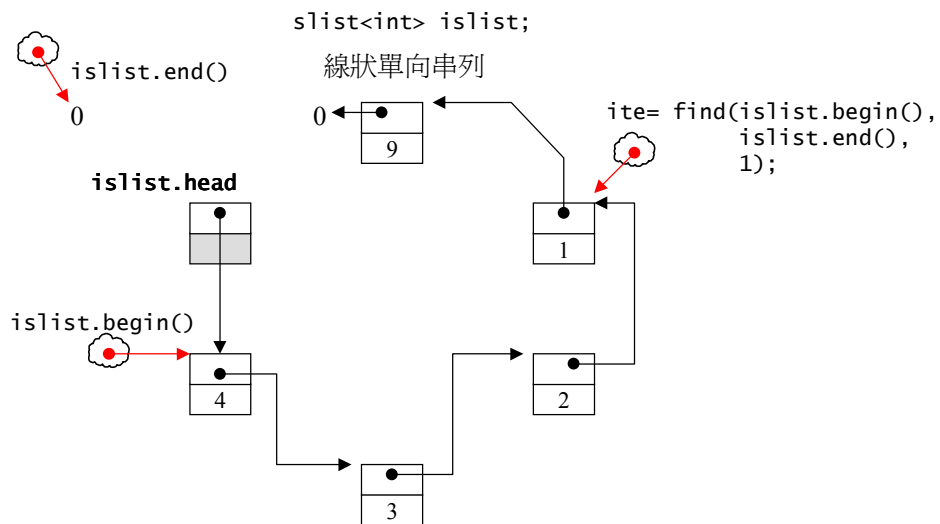


圖 4-26 元素 9,1,2,3,4 依序安插到 `slist` 之後所形成的結構

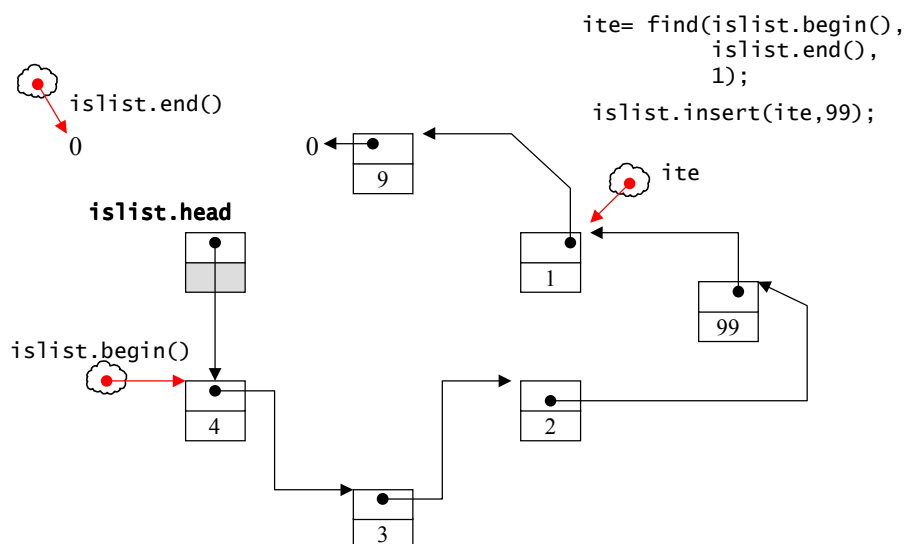


圖 4-27 元素 9,1,2,3,4 依序安插到 `slist` 之後的實際結構

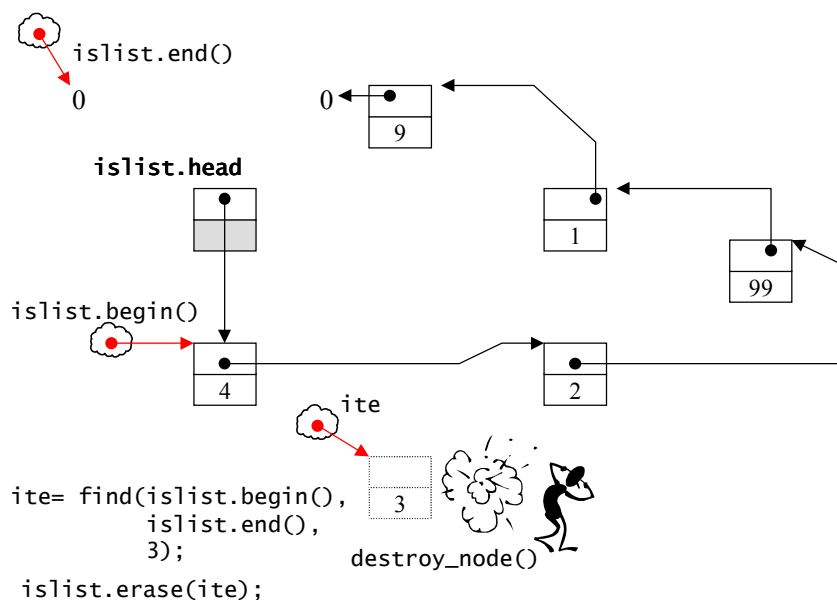


圖 4-28 搜尋元素 3，並將該元素移除

如果你對於圖 4-26、圖 4-27、圖 4-28 中的 `end()` 的畫法感到奇怪，這裡我要做一些說明。請注意，練習程式中一再以迴圈巡訪整個 `slist`，並以迭代器是否等於 `slist.end()` 做為迴圈結束條件，這其中有一些容易疏忽的地方，我必須特別提醒你。當我們呼叫 `end()` 企圖做出一個指向尾端（下一位置）的迭代器，STL 源碼是這麼進行的：

```
iterator end() { return iterator(0); }
```

這會因為源碼中如下的定義：

```
typedef __slist_iterator<T, T&, T*> iterator;
```

而形成這樣的結果：

```
__slist_iterator<T, T&, T*>(0); // 產生一個暫時物件，引發 ctor
```

從而因為源碼中如下的定義：

```
__slist_iterator(list_node* x) : __slist_iterator_base(x) {}
```

而導致基礎類別的建構：

```
__slist_iterator_base(0);
```

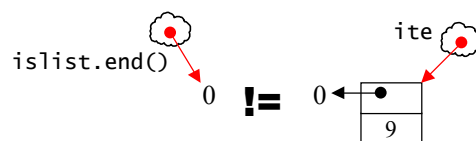
並因為源碼中這樣的定義：

```
struct __slist_iterator_base
{
    __slist_node_base* node; // 指向節點基本結構
    __slist_iterator_base(__slist_node_base* x) : node(x) {}
    ...
};
```

而導致：

```
node(0);
```

因此我在圖 4-26、圖 4-27、圖 4-28 中皆以下圖左側的方式表現 `end()`，它和下圖右側的迭代器截然不同。



A

參考書籍與推薦讀物

Bibliography

Genericity/STL 領域裡頭，已經產生了一些經典作品。

我曾經書寫一篇文章，介紹 Genericity/STL 經典好書，分別刊載於台北《Run!PC》雜誌和北京《程序員》雜誌。該文討論 Genericity/STL 的數個學習層次，文中所列書籍不但是我的推薦，也是本書《STL 源碼剖析》寫作的部分參考。以下摘錄該文中關於書籍的介紹。全文見 <http://www.jjhou.com/programmer-2-stl.htm>。

侯捷觀點《Genericity/STL 大系》——泛型技術的三個學習階段

自從被全球軟體界廣泛運用以來，C++ 有了許多演化與變革。然而就像人們總是把目光放在豔麗的牡丹而忽略了花旁的綠葉，做為一個廣為人知的物件導向程式語言（Object Oriented Programming Language），C++ 所支援的另一種思維——泛型編程——被嚴重忽略了。說什麼紅花綠葉，好似主觀上劃分了主從，其實物件導向思維和泛型思維兩者之間無分主從。兩者相輔相成，肯定對程式開發有更大的突破。

面對新技術，我們的最大障礙在於心中的怯弱和遲疑。To be or not to be, that is the question! 不要和哈姆雷特一樣猶豫不決，當你面對一項有用的技術，必須果敢。

王國維說大事業大學問者的人生有三個境界。依我看，泛型技術的學習也有三個境界，第一個境界是淨解 STL。對程式員而言，諸多抽象描述，不如實象的程式

The Annotated STL Sources

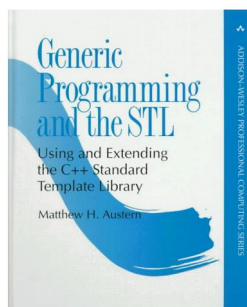
碼直指人心。**第二個境界是理解泛型技術的內涵與 STL 的學理。**不但要理解 STL 的概念分類學（concepts taxonomy）和抽象概念庫（library of abstract concepts），最好再對數個 STL 組件（不必太多，但最好涵蓋各類型）做一番深刻追蹤。STL 源碼都在手上（就是相應的那些 header files 嘛），好好做幾個個案研究，便能夠對泛型技術以及 STL 的學理有深刻的掌握。

第三個境界是擴充 STL。當 STL 不能滿足我們的需求，我們必須有能力動手寫一個可融入 STL 體系中的軟體組件。要到達這個境界之前，得先徹底瞭解 STL，也就是先通過第二境界的痛苦折磨。

也許還應該加上所謂**第一境界：C++ template 機制**。這是學習泛型技術及 STL 的第一道門檻，包括諸如 class templates, function templates, member templates, specialization, partial specialization。更往基礎看去，由於 STL 大量運用了 operator overloading（運算子多載化），所以這個技法也必須熟捻。

以下，我便為各位介紹多本相關書籍，涵蓋不同的切入角度，也涵蓋上述各個學習層次。另有一些則為本書之參考依據。為求方便，以下皆以學術界慣用法標示書籍代名，並按英文字母排序。凡有中文版者，我會特別加註。

[Austern98]: *Generic Programming and the STL - Using and Extending the C++ Standard Template Library*, by Matthew H. Austern, Addison Wesley 1998. 548 pages
繁體中文版：《泛型程式設計與 STL》，侯捷/黃俊堯合譯，基峰 2000, 548 頁。



這是一本艱深的書。沒有三兩三，別想過梁山，你必須對 C++ template 技法、STL 的運用、泛型設計的基本精神都有相當基礎了，才得一窺此書堂奧。

此書第一篇對 STL 的設計哲學有很好的導入，第二篇是詳盡的 STL concepts 完整規格，第三篇則是詳盡的 STL components 完整規格，並附運用範例：

Part I : Introduction to Generic Programming
1. A Tour of the STL
2. Algorithms and Ranges
3. More about Iterators
4. Function Objects
5. Containers
Part II : Reference Manual: STL Concepts
6. Basic Concepts
7. Iterators
8. Function Objects
9. Containers
Part III : Reference Manual : Algorithms and Classes
10. Basic Components
11. Nonmutating Algorithms
12. Basic Mutating Algorithms
13. Sorting and Searching
14. Iterator Classes
15. Function Object Classes
16. Container Classes
Appendix A. Portability and Standardization
Bibliography
Index

此書通篇強調 STL 的泛型理論基礎，以及 STL 的實作規格。你會看到諸如 *concept*, *model*, *refinement*, *range*, *iterator* 等字詞的意義，也會看到諸如 *Assignable*, *Default Constructible*, *Equality Comparable*, *Strict Weakly Comparable* 等觀念的嚴謹定義。雖然一本既富學術性又帶長遠參考價值的工具書，給人嚴肅又艱澀的表象，但此書第二章及第三章解釋 *iterator* 和 *iterator traits* 時的表現，卻不由令人擊節讚賞大嘆精彩。一旦你有能力徹底解放 *traits* 編程技術，你才有能力觀看 STL 源碼（STL 幾乎無所不在地運用 *traits* 技術）並進一步撰寫符合規格的 STL 相容組件。就像其他任何 *framework* 一樣，STL 以開放源碼的方式呈現市場，這種白盒子方式使我們在更深入剖析技術時（可能是為了透徹，可能是為了擴充），有一個終極依恃。因此，觀看 STL 源碼的能力，我認為對技術的養成與掌握，極為重要。

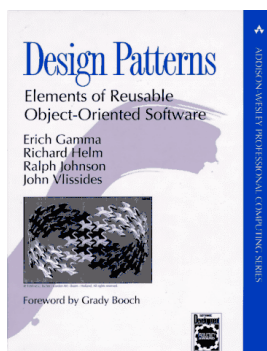
總的來說，此書在 STL 規格及 STL 學理概念的資料及說明方面，目前無出其右者。不論在 (1) 泛型觀念之深入淺出、(2) STL 架構組織之井然剖析、(3) STL 參考文件之詳實整理 三方面，此書均有卓越表現。可以這麼說，在泛型技術和 STL 的學

習道路上，此書並非萬能（至少它不適合初學者），但如果你希望徹底掌握泛型技術與 STL，沒有此書萬萬不能。

[Gamma95]: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. 395 pages

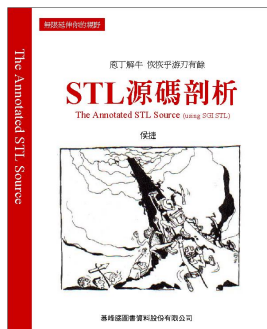
繁體中文版：

《物件導向設計模式—可再利用物件導向軟體之要素》葉秉哲譯，培生 2001, 458 頁



此書與泛型或 STL 並沒有直接關係。但是 STL 的兩大類型組件：*Iterator* 和 *Adapter*，被收錄於此書 23 個設計樣式（design patterns）中。此書所談的其他設計樣式在 STL 之中也有發揮。兩相比照，尤其是看過 STL 的源碼之後，對於這些設計樣式會有更深的體會，迴映過來對 STL 本身架構也會有更深一層的體會。

[Hou02a]: 《STL 源碼剖析 — 向專家取經，學習 STL 實作技術、強型檢驗、記憶體管理、演算法、資料結構之高階編程技法》，侯捷著，基峰 2002，??? 頁。

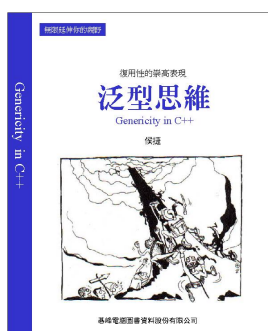


這便是你手上這本書。揭示 SGI STL 實作版本的關鍵源碼，涵蓋 STL 六大組件之

The Annotated STL Sources

實作技術和原理解說。是學習泛型編程、資料結構、演算法、記憶體管理…等高階編程技術的終極性讀物，畢竟…唔…源碼之前了無秘密。

[Hou02b]:《泛型思維 — Genericity in C++》，侯捷著（計劃中）

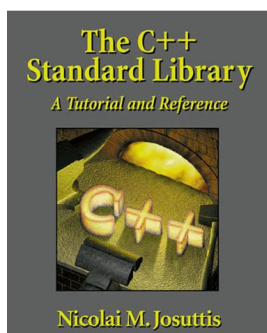


下筆此刻，此書尚在撰寫當中，內容涵蓋語言層次(C++ templates 語法、Java generic 語法、C++ 運算子重載)，STL 原理介紹與架構分析，STL 現場重建，STL 深度應用，STL 擴充示範，泛型思考。並附一個微型、高度可移植的 STLLite，讓讀者得以最精簡的方式和時間一窺 STL 全貌，一探泛型之宏觀與微觀。

[Josuttis99]: *The C++ Standard Library - A Tutorial and Reference*, by Nicolai M. Josuttis, Addison Wesley 1999. 799 pages

繁體中文版：

《C++ 標準程式庫 — 學習教本與參考工具》，侯捷/孟岩譯，碁峰 2002, 800 頁。



一旦你開始學習 STL，乃至實際運用 STL，這本書可以為你節省大量的翻查、參考、錯誤嘗試的時間。此書各章如下：

The Annotated STL Sources

- 1. About the Book
- 2. Introduction to C++ and the Standard Library
- 3. General Concepts
- 4. Utilities
- 5. The Standard Template Library
- 6. STL Containers
- 7. STL Iterators
- 8. STL Function Objects
- 9. STL Algorithms
- 10. Special Containers
- 11. Strings
- 12. Numerics
- 13. Input/Output Using Stream Classes
- 14. Internationalization
- 15. Allocators
- Internet Resources
- Bibliography
- Index

此書涵蓋面廣，不僅止於 STL，而且是整個 C++ 標準程式庫，詳細介紹每個組件的規格及運用方式，並佐以範例。作者的整理功夫做得非常紮實，並大量運用圖表做為解說工具。此書的另一個特色是涵蓋了 STL 相關各種異常 (exceptions)，這很少見。

此書不僅介紹 STL 組件的運用，也導入關鍵性 STL 源碼。這些源碼都經過作者的節錄整理，砍去枝節，留下主幹，容易入目。這是我特別激賞的一部份。繁中取簡，百萬軍中取敵首級，不是容易的任務，首先得對龐大的源碼有清晰的認識，再有堅定而正確的詮釋主軸，知道什麼要砍，什麼要留，什麼要註解。

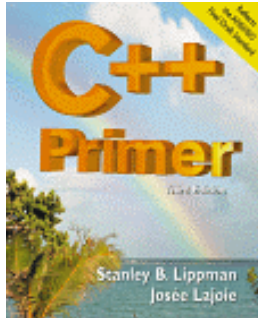
閱讀此書，不但得以進入我所謂的第一學習境界，甚且由於關鍵源碼的提供，得以進入第二境界。此書也適度介紹某些 STL 擴充技術。例如 6.8 節介紹如何以 smart pointer 使 STL 容器具有 "reference semantics" (STL 容器原本只支援 "value semantics")，7.5.2 節介紹一個訂製型 iterator，10.1.4 節介紹一個訂製型 stack，10.2.4 節介紹一個訂製型 queue，15.4 節介紹一個訂製型 allocator。雖然篇幅都不長，只列出基本技法，但對於想要擴充 STL 的程式員而言，有個起始終究是一種實質上的莫大幫助。就這點而言，此書又進入了我所謂的第三學習境界。

正如其副標所示，本書兼具學習用途及參考價值。在國際書市及國際 STL 相關研討會上，此書都是首選。盛名之下無虛士，誠不欺也。

The Annotated STL Sources

[Lippman98] : *C++ Primer*, 3rd Edition, by Stanley Lippman and Josée Lajoie, Addison Wesley Longman, 1998. 1237 pages.

繁體中文版：《C++ Primer 中文版》，侯捷譯，基峰 1999, 1237 頁。



這是一本 C++ 百科經典，向以內容廣泛說明詳盡著稱。其中與 `template` 及 `STL` 直接相關的章節有：

```
chap6: Abstract Container Types
chap10: Function Templates
chap12: The Generic Algorithms
chap16: Class Templates
appendix: The Generic Algorithms Alphabetically
```

與 `STL` 實作技術間接相關的章節有：

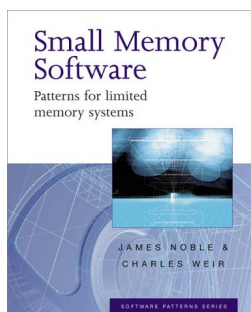
```
chap15: Overloaded Operators and User Defined Conversions
```

書中有大量範例，尤其附錄列出所有 `STL` 泛型演算法的規格、說明、實例，是極佳的學習資料。不過書上有少數例子，由於作者疏忽，未能完全遵循 C++ 標準，仍沿用舊式寫法，修改方式可見 www.jjhou.com/errata-cpp-primer-appendix.htm。

這本 C++ 百科全書並非以介紹泛型技術的角度出發，而是因為 C++ 涵蓋了 `template` 和 `STL`，所以才介紹它。因此在相關組織上，稍嫌凌亂。不過我想，沒有人會因此對它求全責備。

[Noble01]: *Small Memory Software – Patterns for systems with limited memory*, by James Noble & Charles Weir, Addison Wesley, 2001. 333 pages.

繁體中文版：《記憶體受限系統 之設計樣式》，侯捷/王飛譯，碁峰 2002，333 頁。



此書和泛型技術、STL 沒有任何關連。然而由於 SGI STL allocator（空間配置器）在記憶體配置方面運用了 memory pool 手法，如果能夠參考此書所整理的一些記憶體管理經典手法，頗有助益，並收觸類旁通之效。

[Stroustrup97]: *The C++ Programming Language*, 3rd Editoin, by Bjarne Stroustrup, Addison Wesley Longman, 1997. 910 pages

繁體中文版：《C++ 程式語言經典本》，葉秉哲譯，儒林 1999，總頁數未錄。



這是一本 C++ 百科經典，向以學術權威（以及口感艱澀☹）著稱。本書內容直接與 template 及 STL 相關的章節有：

chap3: A Tour of the Standard Library
chap13: Templates
chap16: Library Organization and Containers
chap17: Standard Containers
chap18: Algorithms and Function Objects
chap19: Iterators and Allocators

與 STL 實作技術間接相關的章節有：

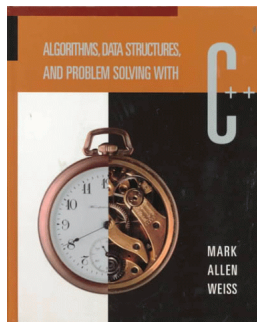
The Annotated STL Sources

chap11: Operator Overloading

其中第 19 章對 Iterators Traits 技術的介紹，在 C++ 語法書中難得一見，不過蜻蜓點水不易引發閱讀興趣。關於 Traits 技術，[Austern98] 表現極佳。

這本 C++ 百科全書並非以介紹泛型技術的角度出發，而是因為 C++ 涵蓋了 template 和 STL，所以才介紹它。因此在相關組織上，稍嫌凌亂。不過我想，沒有人會因此對它求全責備。

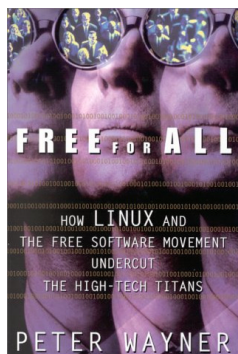
[Weiss95]: *Algorithms, Data Structures, and Problem Solving With C++*, by Mark Allen Weiss, Addison Wesley, 1995, 820 pages



此書和泛型技術、STL 沒有任何關連。但是在你認識 STL 容器和 STL 演算法之前，一定需要某些資料結構（如 red black tree, hash table, heap, set...）和演算法（如 quick sort, heap sort, merge sort, binary search...）以及 Big-Oh 複雜度標記法的學理基礎。本書在學理敘述方面表現不俗，用字用例淺顯易懂，頗獲好評。

[Wayner00]: *Free For All – How Linux and the Free Software Movement Undercut the High-Tech Titans*, by Peter Wayner, HarperBusiness, 2000. 340 pages

繁體中文版：《開放原始碼—Linux 與自由軟體運動對抗軟體巨人的故事》，蔡憶懷譯，商周 2000，393 頁。



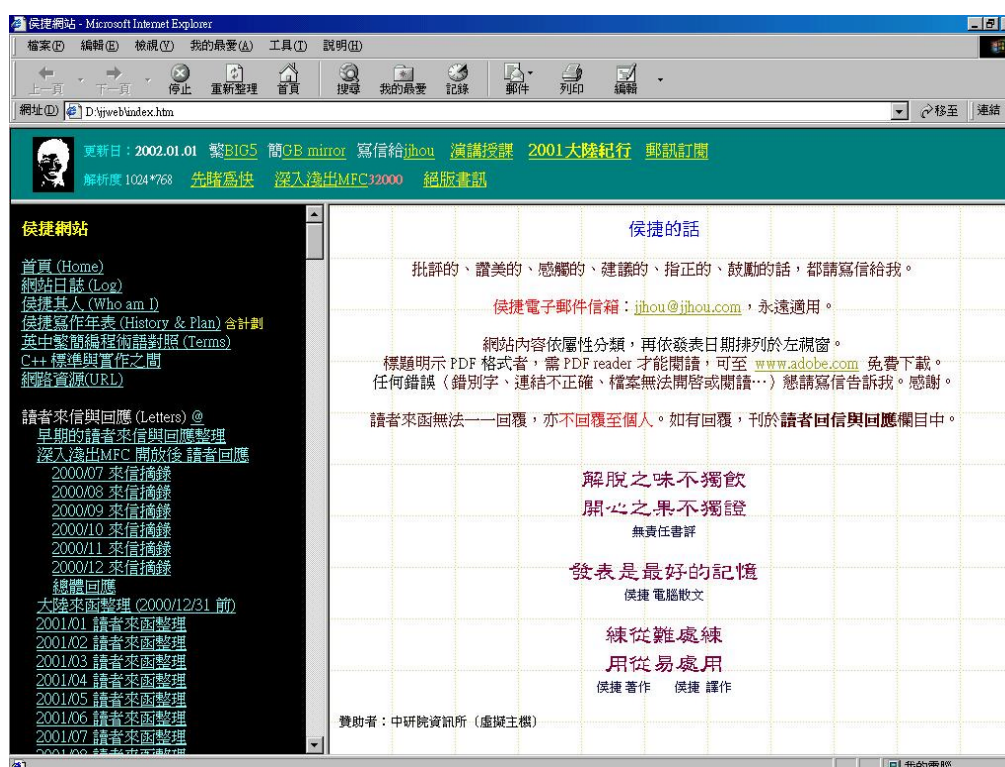
《STL 源碼剖析》一書採用 SGI STL 實作版本為解說對象，而 SGI 版本屬於源碼開放架構下的一員，因此《STL 源碼剖析》第一章對於 **open source**（源碼開放精神）、**GNU**（由 Richard Stallman 創先領導的開放改革計劃）、**FSF**（Free Software Foundation，自由軟體基金會）、**GPL**（General Public License，廣泛開放授權）等等有概要性的說明，皆以此書為參考依據。

B

侯捷網站

本書支援站點簡介

侯捷網站是我的個人站點，收錄寫作教育生涯的所有足跡。我的所有作品的勘誤、討論、源碼下載、電子書下載等服務都在這個站點上進行。永久網址是 <http://www.jjhou.com>（中文繁體），中文簡體鏡像站點為 <http://jjhou.csdn.com>。下面是進入侯捷網站後的畫面，其中上視窗是變動主題，提供最新動態。左視窗是目錄，右視窗是主畫面：



左視窗之主目錄，內容包括：

- 首頁
- 網站日誌
- 侯捷其人
- 侯捷寫作年表 (History) 含計劃
- [英中繁簡編程術語對照 \(Terms\)](#)
- [C++ 標準與實作之間](#)
- [網路資源 \(URL\)](#)

- 讀者來信與回應
- 課程
- [電子書開放](#)
- [程式源碼下載](#)
- [答客問 \(Q/A\)](#)
- 作品勘誤 (errata)
- 無責任書評 1 1993.01~1994.04
- 無責任書評 2 1994.07~1995.12
- 無責任書評 3 1996.08~1997.11
- 侯捷散文 1998
- 侯捷散文 1999
- 侯捷散文 2000
- 侯捷散文 2001
- 侯捷散文 2002
- STL 系列文章 (PDF)
- 《程序員》雜誌文章
- 侯捷著作
- [侯捷譯作](#)
- 作序推薦

本書《STL 源碼剖析》出版後之相關服務，皆可於以上各相關欄位中獲得。

C

STLPort 的移植經驗

by 孟岩¹

STL 是一個標準，各商家根據這個標準開發了各自的 STL 版本。而在這形形色色的 STL 版本中，SGI STL 無疑是最引人矚目的一個。這當然是因為這個 STL 產品系出名門，其設計和編寫者名單中，Alexander Stepanov 和 Matt Austern 赫然在內，有兩位大師坐鎮，其代碼水平自然有了最高保證。SGI STL 不但在效率上一直名列前茅，而且完全依照 ISO C++ 之規範設計，使用者儘可放心。此外，SGI STL 做到了 thread-safe，還體貼地為用戶增設數種組件，如 hash, hash_map, hash_multimap, slist 和 rope 容器等等。因此無論在學習或實用上，SGI STL 應是首選。

無奈，SGI STL 本質上是為了配合 SGI 公司自身的 UNIX 變體 IRIX 而量身定做，其他平台上的 C++ 編譯器想使用 SGI STL，都需要一番週折。著名的 GNU C++ 雖然也使用 SGI STL，但在發行前已經過調試整合。一般用戶，特別是 Windows 平台上的 BCB/VC 用戶要想使自己的 C++ 編譯器與 SGI STL 共同工作，可不是一件容易的事情。俄國人 Boris Fomitchev 注意到這個問題之後，建立了一個免費提供服務的專案，稱為 STLport，旨在將 SGI STL 的基本代碼移植到各主流編譯環境中，使各種編譯器的用戶都能夠享受到 SGI STL 帶來的先進機能。STLport 發展過程中曾接受 Matt Austern 的指導，發展到今天，已經比較成熟。最新的 STLport 4.0，可以從 www.stlport.org 免費下載，zip 檔案體積約 1.2M，可支持各主流 C++ 編譯環境的移植。BCB 及 VC 當然算是主流編譯環境，所以當然也得到了 STLport 的關照。但據筆者實踐經驗看來，配置過程中還有一些障礙需要跨越，本文詳細

¹ 感謝孟岩先生同意將他的寶貴經驗與本書讀者分享。原文以大陸術語寫就，為遷就臺灣讀者方便，特以臺灣術語略做修改。

指導讀者如何在 Borland C++Builder 5.5 及 Visual C++ 6.0 環境中配置 STLport。

首先請從 www.stlport.org 下載 STLport 4.0 的 ZIP 檔案，檔名 `stlport-4.0.zip`。然後利用 WinZip 等工具展開。生成 `stlport-4.0` 目錄，該目錄中有（而且僅有）一個子目錄，名稱亦為 `stlport-4.0`，不妨將整目錄拷貝到你的合適位置，然後改一個合適的名字，例如配合 BCB 者可名為 `STL4BC...` 等等。

下面分成 BCB 和 VC 兩種情形來描述具體過程。

Borland C++Builder 5

Borland C++Builder5 所攜帶的 C++ 編譯器是 5.5 版本，在當前主流的 Windows 平台編譯器中，對 ISO C++ *Standard* 的支持是最完善的。以它來配合 SGI STL 相當方便，也是筆者推薦之選。手上無此開發工具的讀者，可以到 www.borland.com 免費下載 Borland C++ 5.5 編譯器的一個精簡版，該精簡版體積為 8.54M，名為 `freecommandlinetools1.exe`，乃一自我解壓縮安裝檔案，在 Windows 中執行它便可安裝到你選定的目錄中。展開後體積 50M。

以下假設你使用的 Windows 安裝於 `C:\Windows` 目錄。如果你有 BCB5，假設安裝於 `C:\Program Files\Borland\CBUILDER5`；如果你沒有 BCB5，而是使用上述的精簡版 BCC，則假設安裝於 `C:\BCC55` 目錄。STLport 原包置於 `C:\STL4BC`，其中應有以下內容：

- <目錄> `doc`
- <目錄> `lib`
- <目錄> `src`
- <目錄> `stlport`
- <目錄> `test`
- 檔案 `ChangLog`
- 檔案 `Install`
- 檔案 `Readme`
- 檔案 `Todo`

請確保 `C:\Program Files\Borland\CBUILDER5\Bin` 或 `C:\BCC55\Bin` 已登記於你的 Path 環境變數中。

筆者推薦你在安裝之前讀一讀 `Install` 檔案，其中講到如何避免使用 SGI 提供的

iostream。如果你不願意使用 SGI iostream，STLport 會在原本編譯器自帶的 iostream 外加一個 wrapper，使之能與 SGI STL 共同合作。不過 SGI 提供的 iostream 標準化程度好，和本家的 STL 代碼配合起來速度也快些，所以筆者想不出什麼理由不使用它，在這裡假定大家也都樂於使用 SGI iostream。有不同看法者儘可按照 Install 檔案的說法調整。

下面是逐一步驟（本任務均在 DOS 命令狀態下完成，請先打開一個 DOS 視窗）：

1. 移至 C:\Program Files\Borland\CBUILDER5\bin，使用任何文字編輯器修改以下兩個檔案。

檔案一 bcc32.cfg 改為：

```
-I"C:\STL4BC\stlport";\  
"C:\Program Files\Borland\CBUILDER5\Include";\  
"C:\Program Files\Borland\CBUILDER5\Include\vcl"  
-L"C:\STL4BC\LIB";\  
"C:\Program Files\Borland\CBUILDER5\Lib";\  
"C:\Program Files\Borland\CBUILDER5\Lib\obj";\  
"C:\Program Files\Borland\CBUILDER5\Lib\release"
```

以上爲了方便閱讀，以 "\" 符號將很長的一行折行。本文以下皆如此。

檔案二 ilink32.cfg 改為：

```
-L"C:\STL4BC\LIB";\  
"C:\Program Files\Borland\CBUILDER5\Lib";\  
"C:\Program Files\Borland\CBUILDER5\Lib\obj";\  
"C:\Program Files\Borland\CBUILDER5\Lib\release"
```

C:\BCC55\BIN 目錄中並不存在這兩個檔案，請你自己用文字編輯器手工做出這兩個檔案來，內容與上述有所不同，如下。

檔案一 bcc32.cfg 內容：

```
-I"C:\STL4BC\stlport"; "C:\BCC55\Include";  
-L"C:\STL4BC\LIB"; "C:\BCC55\Lib";
```

檔案二 ilink32.cfg 內容：

```
-L"C:\STL4BC\LIB"; "C:\BCC55\Lib";
```

2. 進入 C:\STL4BC\SRC 目錄。
3. 執行命令 `copy bcb5.mak Makefile`
4. 執行命令 `make clean all`

這個命令會執行很長時間，尤其在老舊機器上，可能運行 30 分鐘以上。螢幕

不斷顯示工作情況，有時你會看到好像在反覆做同樣幾件事，請保持耐心，這其實是在以不同編譯開關建立不同性質的標的程式庫。

5. 經過一段漫長的編譯之後，終於結束了。現在再執行命令 `make install`。這次需要的時間不長。
6. 來到 `C:\STL4BC\LIB` 目錄，執行：


```
copy *.dll c:\windows\system;
```
7. 大功告成。下面一步進行檢驗。`rope` 是 SGI STL 提供的一個特有容器，專門用來對付超大規模的字串。`string` 是細弦，而 `rope` 是粗繩，可以想見 `rope` 的威力。下面這個程式有點暴殄天物，不過倒也還足以做個小試驗：

```
//issgistl.cpp
#include <iostream>
#include <rope>

using namespace std;

int main()
{
    // crope 就是容納 char-type string 的 rope 容器
    crope bigstr1("It took me about one hour ");
    crope bigstr2("to plug the STLport into Borland C++!");
    crope story = bigstr1 + bigstr2;
    cout << story << endl;
    return 0;
}
//~issgistl.cpp
```

現在，針對上述程式進行編譯：`bcc32 issgistl.cpp`。咦，怪哉，`linker` 報告說找不到 `stlport_bcc_static.lib`，到 `C:\STL4BC\LIB` 看個究竟，確實沒有這個檔案，倒是有一個 `stlport_bcb55_static.lib`。筆者發現這是 STLport 的一個小問題，需要將程式庫檔案名稱做一點改動：

```
copy stlport_bcb55_static.lib stlport_bcc_static.lib
```

這個做法頗為穩妥，原本的 `stlport_bcb55_static.lib` 也保留了下來。以其他選項進行編譯時，如果遇到類似問題，只要照葫蘆畫瓢改變檔案名稱就沒問題了。

現在再次編譯，應該沒問題了。可能有一些警告訊息，沒關係。只要能運行，就表示 `rope` 容器起作用了，也就是說你的 SGI STL 開始工作了。

Microsoft Visual C++ 6.0:

Microsoft Visual C++ 6.0 是當今 Windows 下 C++ 編譯器主流中的主流，但是對於 ISO C++ 的支持不盡如人意。其所配送的 STL 性能也比較差。不過既然是主流，STLport 自然不敢怠慢，下面介紹 VC 中的 STLport 安裝方法。

以下假設你使用的 Windows 系統安裝於 C:\Windows 目錄，VC 安裝於 C:\Program Files\Microsoft Visual Studio\VC98；而 STLport 原包置於 C:\STL4VC，其中應有以下內容：

```
<目錄> doc
<目錄> lib
<目錄> src
<目錄> stlport
<目錄> test
檔案 ChangLog
檔案 Install
檔案 Readme
檔案 Todo
```

請確保 C:\Program Files\Microsoft Visual Studio\VC98\bin 已設定在你的 Path 環境變數中。

下面是逐一步驟（本任務均在 DOS 命令狀態下完成，請先打開一個 DOS 視窗）：

1. 移至 C:\Program Files\Microsoft Visual Studio\VC98 中，使用任何文字編輯器修改檔案 vcvars32.bat。將其中原本的兩行：

```
set
INCLUDE=%MSVCDir%\ATL\INCLUDE;%MSVCDir%\INCLUDE;%MSVCDir%\MFC\I
NCLUDE;%INCLUDE%
set LIB=%MSVCDir%\LIB;%MSVCDir%\MFC\LIB;%LIB%
```

改成：

```
set
INCLUDE=C:\STL4VC\stlport;%MSVCDir%\ATL\INCLUDE;%MSVCDir%\INCLUDE;\
%MSVCDir%\MFC\INCLUDE;%INCLUDE%
set LIB=C:\STL4VC\lib;%MSVCDir%\LIB;%MSVCDir%\MFC\LIB;%LIB%
```

以上爲了方便閱讀，以 "\" 符號將很長的一行折行。

修改完畢後存檔，然後執行之。一切順利的話應該給出一行結果：

```
Setting environment for using Microsoft Visual C++ tools.
```

如果你預設的 DOS 環境空間不足，這個 BAT 檔執行過程中可能導致環境空間不足，此時應該在 DOS 視窗的「內容」對話框中找到「記憶體」附頁，將「起始環境」（下拉式選單）改一個較大的值，例如 1280 或 2048。然後再開一個 DOS 視窗，重新執行 `vcvars32.bat`。

2. 進入 `C:\STL4VC\SRC` 目錄。
3. 執行命令 `copy vc6.mak Makefile`
4. 執行命令 `make clean all`

如果說 BCB 編譯 STLport 的時間很長，那麼 VC 編譯 STLport 的過程就更加漫長了。螢幕反反覆覆地顯示似乎相同的內容，請務必保持耐心，這其實是在以不同編譯開關建立不同性質的標的程式庫。

5. 經過一段漫長的編譯之後，終於結束了。現在你執行命令 `make install`。這次需要的時間不那麼長，但也要有點耐心。
6. 大功告成。下一步應該檢驗是不是真的用上了 SGI STL。和前述的 BCB 過程差不多，找一個運用了 SGI STL 特性的程式，例如運用了 `rope`, `slist`, `hash_set`, `hash_map` 等容器的程式來編譯。注意，編譯時務必使用以下格式：

```
cl /GX /MT program.cpp
```

這是因為 SGI STL 大量使用了 `try...throw...catch`，而 VC 預設情況下並不支持此一語法特性。`/GX` 要求 VC++ 編譯器打開對異常處理的語法支持。`/MT` 則是要求 VC linker 將本程式的 `obj` 檔和 `libcmtd.lib` 連結在一起 — 因為 SGI STL 是 `thread-safe`，必須以 `multi-thread` 的形式運行。

如果想要在圖形介面中使用 SGI STL，可在 VC 整合環境內調整 `Project | Setting(Alt+F7)`，設置編譯選項，請注意一定要選用 `/MT` 和 `/GX`，並引入選項 `/Ic:\stl4vc\stlport` 及 `/libpath:c:\stl4vc\lib`。

整個過程在筆者的老式 Pentium 150 機器上耗時超過 3 小時，雖然你的機器想必快得多，但也必然會花去出乎你意料的時間。全部完成後，`C:\STL4VC` 這個目錄的體積由原本區區 4.4M 膨脹到可怕的 333M，當然這其中有 300M 是編譯過程產生的 `.obj` 檔，如果你確信自己的 STLport 工作正常的話，可以刪掉它們，空出硬碟空間。不過這麼一來若再進行一次安裝程序，就只好再等很長時間。

另外，據筆者勘察，STLport 4.0 所使用的 SGI STL 並非最新問世的 SGI STL3.3 版

本，不知道把 SGI STL3.3 的代碼導入 STLport 會有何效果，有興趣的讀者不妨一試。

大致情形就是這樣，現在，套用 STLport 自帶檔案的結束語：享受這一切吧（Have fun!）

孟岩
2001-3-11

索引 I

請注意，本書並未探討所有的 STL 源碼（那需要數千頁篇幅），所以
請不要將此一索引視為完整的 STL 組件索引

```
( )
  as operator 414
*
  for auto_ptr 81
  for deque iterator 148
  for hash table iterator 254
  for list iterator 130
  for red black tree iterator 217
  for slist iterator 189
  for vector iterator 117
+
  for deque iterator 150
  for vector iterator 117
++
  for deque iterator 149
  for hash table iterator 254
  for list iterator 131
  for red black tree iterator 217
  for slist iterator 189
  for vector iterator 117
+=
  for deque iterator 149
  for vector iterator 117
-
  for deque iterator 150
  for vector iterator 117
--
  for deque iterator 149
  for list iterator 131
  for red black tree iterator 217
  for vector iterator 117

-=
  for deque iterator 150
  for vector iterator 117
->
  for auto_ptr 81
  for deque iterator 148
  for hash table iterator 254
  for list iterator 131
  for red black tree iterator 217
  for slist iterator 189
  for vector iterator 117
[ ]
  for deques 151
  for deque iterators 150
  for maps 240
  for vectors 116,118
```

A

```
accumulate() 299
adapter
  for containers 425
  for functions
    see function adapter
  for member functions
    see member function adapter
address()
  for allocators 44
adjacent_difference() 300
adjacent_find() 343
advance() 93, 94, 96
```


- algorithm 285
 - accumulate() 299
 - adjacent_difference() 300
 - adjacent_find() 343
 - binary_search() 379
 - complexity 286
 - copy() 314
 - copy_backward() 326
 - count() 344
 - count_if() 344
 - equal() 307
 - equal_range() 400
 - fill() 308
 - fill_n() 308
 - find() 345
 - find_end() 345
 - find_first_of() 348
 - find_if() 345
 - for_each() 349
 - generate() 349
 - generate_n() 349
 - header file 288
 - heap 174
 - includes() 349
 - inner_product() 301
 - inplace_merge() 403
 - iterator_swap() 309
 - itoa() 305
 - lexicographical_compare() 310
 - lower_bound() 375
 - make_heap() 180
 - max_element() 352
 - maximum 312
 - merge() 352
 - min_element() 354
 - minimum 312
 - mismatch() 313
 - next_permutation() 380
 - nth_element() 409
 - numeric 298
 - overview 285
 - partial_sort() 386
 - partial_sort_copy() 386
 - partial_sum() 303
 - partition() 354
 - pop_heap() 176
 - power() 304
 - prev_permutation() 382
 - push_heap() 174
 - random_shuffle() 383
 - ranges 39
 - remove() 357
 - remove_copy() 357
 - remove_copy_if() 358
 - remove_if() 357
 - replace() 359
 - replace_copy() 359
 - replace_copy_if() 360
 - replace_if() 359
 - reverse() 360
 - reverse_copy() 361
 - rotate() 361
 - rotate_copy() 365
 - search() 365
 - search_n() 366
 - set_difference() 334
 - set_intersection() 333
 - set_symmetric_difference() 336
 - set_union() 331
 - sort() 389
 - sort_heap() 178
 - suffix_copy 293
 - suffix_if 293
 - swap_ranges() 369
 - transform() 369
 - unique() 370
 - unique_copy() 371
 - upper_bound() 377
- <algorithm> 294
- alloc 47
- allocate()
 - for allocators 62
- allocator 43
 - address() 44, 46
 - allocate() 44, 46
 - const_pointer 43, 45
 - const_reference 43, 45
 - construct() 44, 46
 - constructor 44
 - deallocate() 44, 46
 - destroy() 44, 46
 - destructor 44
 - difference_type 43, 45
 - max_size() 44, 46
 - pointer 43, 45
 - rebind 44, 46
 - reference 43, 45

- size_type 43, 45
- standard interface 43
- user-defined, JJ version 44
- value_type 43, 45
- allocator 48
- argument_type 416, 417
- arithmetic
 - of iterators 92
- array 114, 115, 144, 173
- associative container 197
- auto pointer
 - see auto_ptr
- auto_ptr 81
 - * 81
 - > 81
 - = 81
 - constructor 81
 - destructor 81
 - get() 81
 - header file 81
 - implementation 81

B

- back()
 - for deque 151
 - for lists 132
 - for queues 170
 - for vectors 116
- back inserter 426
- back_inserter 436
- bad_alloc 56, 58, 59, 68
- base() 440
- begin()
 - for deque 151
 - for lists 131
 - for maps 240
 - for red-black tree 221
 - for sets 235
 - for slist 191
 - for vectors 116
- bibliography 461
- bidirectional iterator 92, 95, 101
- Big-O notation 286
- binary_function 417
- binary_negate 451
- binary predicate 450
- binary_search() 379
- bind1st 433, 449, 452
- bind2nd 433, 449, 453
- binder1st 452
- binder2nd 452

C

- capacity
 - of vectors 118
- capacity()
 - for vectors 116
- category
 - of container iterators 92, 95
 - of iterators 92, 97
- class
 - auto_ptr 81
 - deque
 - see deque
 - hash_map 275
 - hash_multimap 282
 - hash_multiset 279
 - hash_set 270
 - list
 - see list
 - map
 - see map
 - multimap
 - see multimap
 - multiset
 - see multiset
 - priority_queue 184
 - queue 170
 - set
 - see set
 - stack 167
 - vector
 - see vector
- clear()
 - for hash table 263
 - for sets 235
 - for vectors 117, 124
- commit-or-rollback 71, 72, 123, 125, 154, 158
- compare
 - lexicographical 310
- complexity 286
- compose1 453
- compose1 433, 453
- compose2 453
- compose2 433, 454
- compose function object 453
- const_mem_fun1_ref_t 433, 449, 459
- const_mem_fun1_t 433, 449, 459

`const_mem_fun_ref_t` 433,449,458
`const_mem_fun_t` 433,449,458
`construct()`
 for allocators 51
`constructor`
 for deques 153
 for hash table 258
 for lists 134
 for maps 240
 for multimaps 246
 for multisets 245
 for priority queues 184
 for red black tree 220,222
 for sets 234
 for slists 190
 for vectors 116
`copy()`
 algorithm 314
`copy_backward()` 326
`copy_from()`
 for hash table 263
`count()` 344
 for hash table 267
 for maps 241
 for sets 235
`count_if()` 344
 Cygnus 9

D

`deallocate()`
 for allocators 64
`__default_alloc_template` 59, 61
`deque` 143, 144, 150
 see container
 [] 151
 `back()` 151
 `begin()` 151
 `clear()` 164
 `constructor` 153
 `empty()` 151
 `end()` 151
 `erase()` 164, 165
 example 152
 `front()` 151
 `insert()` 165
 iterators 146
 `max_size()` 151
 `pop_back()` 163
 `pop_front()` 157, 163

`push_back()` 156
 `size()` 151
`destroy()`
 for allocators 51
`destructor`
 for red black tree 220
 for vectors 116
`dictionary` 198,247
`distance()` 98
`dynamic array container` 115

E

EGCS 9
`empty()`
 for deques 151
 for lists 131
 for maps 240
 for priority queues 184
 for queues 170
 for red black tree 221
 for sets 235
 for stacks 168
 for vectors 116
`end()`
 for deques 151
 for lists 131
 for maps 240
 for red black tree 221
 for sets 1235
 for vectors 116
`equal()` 307
`equal_range()` 400
 for maps 241
 for sets 236
`equal_to` 420
`erase()`
 for deques 164,165
 for lists 136
 for maps 241
 for sets 235
 for vectors 117,123

F

`fill_n()` 308
`find()`
 algorithm 345
 for hash table 267
 for maps 241

- for red black tree 229
 - for sets 235
- find_end() 345
- find_first_of()
 - algorithm 348
- find_if() 345
- first
 - for pairs 237
- first_argument_type 417
- first_type
 - for pairs 237
- for_each() 349
- forward iterator 92,95
- Free Software Foundation 7
- front()
 - for deque 151
 - for lists 131
 - for vectors 116
- front_inserter 426
- front_inserter 426, 436
- FSF see also Free Software Foundation
- function adapter 448
 - bind1st 452
 - bind2nd 453
 - binder1st 452
 - binder2nd 452
 - compose1 453
 - compose1 433,453
 - compose2 453
 - compose2 433,454
 - mem_fun 431,433,449,456,459
 - mem_fun_ref 431,433,449,456,460
 - not1 433, 449, 451
 - not2 433, 449, 451
 - ptr_fun 431,433,449,454,455
- <functional> 415
- functional composition 453
- function object 413
 - as sorting criterion 413
 - bind1st 433,449,452
 - bind2nd 433,449,453
 - compose1 453
 - compose1 433,453
 - compose2 453
 - compose2 453,454
 - divides 418
 - equal_to 420
 - example for arithmetic functor 419
 - example for logical functor 423

- example for rational functor 421
- greater 421
- greater_equal 421
- header file 415
- identity 424
- identity_element 420
- less 421
- less_equal 421
- logical_and 422
- logical_not 422
- logical_or 422
- mem_fun 431,433,449,456,459
- mem_fun_ref 431,433,449,456,460
- minus 418
- modulus 418
- multiplies 418
- negate 418
- not1 433, 449, 451
- not2 433, 449, 451
- not_equal_to 420
- plus 418
- project1st 424
- project2nd 424
- ptr_fun 431,433,449,454,455
- select1st 424
- select2nd 424
- functor 413
 - see also function objects

G

- GCC 8
- General Public License 8
- generate() 349
- generate_n() 349
- get()
 - for auto_ptrs 81
- GPL see also General Public License
- greater 421
- greater_equal 421

H

- half-open range 39
- hash_map 275
 - example 278
- hash_multimap 282
- hash_multiset 279
- hash_set 270
 - example 273

- hash table 247, 256
 - buckets 253
 - clear() 263
 - constructors 258
 - copy_from() 263
 - count() 267
 - example 264, 269
 - find() 267
 - hash functions 268
 - iterators 254
 - linear probing 249
 - loading factor 249
 - quadratic probing 251
 - separate chaining 253
- header file
 - for SGI STL, see section 1.8.2
- heap 172
 - example 181
- heap algorithms 174
 - make_heap 180
 - pop_heap 176
 - push_heap 174
 - sort_heap 178
- heapsort 178

I

- include file
 - see header file
- index operator
 - for maps 240, 242
- inner_product() 301
- inplace_merge() 403
- input iterator 92
- input_iterator 95
- input stream
 - iterator 426, 442
 - read() 443
- insert()
 - called by inserters 435
 - for deques 165
 - for lists 135
 - for maps 240, 241
 - for multimaps 246
 - for multisets 245
 - for sets 235
 - for vectors 124
- insert_equal()
 - for red black tree 221
- insert_unique()

- for red black tree 221
- inserter 426, 428
- insert iterator 426, 428
- introsort 392
- istream iterator 426
- iterator 79
 - adapters 425
 - advance() 93, 94, 96
 - back_inserter 426, 436
 - back inserters 426, 435, 436
 - bidirectional 92, 95
 - categories 92
 - convert into reverse iterator 426, 437, 439
 - distance() 98
 - end-of-stream 428, 443
 - for hash tables 254
 - for lists 129
 - for maps 239
 - for red black trees 214
 - for sets 234
 - for slists 188
 - for streams 426, 442
 - for vectors 117
 - forward 92, 95
 - front_inserter 426, 436
 - front inserters 426, 436
 - input 92, 95
 - inserter 426, 436
 - iterator tags 95
 - iterator traits 85, 87
 - iter_swap() 309
 - output 92, 95
 - past-the-end 39
 - random access 92, 95
 - ranges 39
 - reverse 426, 437, 440
- iterator adapter 425, 435
 - for streams 426, 442
 - inserter 426, 436
 - reverse 426, 437, 440
- iterator tag 95
- iterator traits 85, 87
 - for pointers 87
- iter_swap() 309

K

- key_comp() 221, 235, 240
- key_compare 219, 234, 239
- key_type 218, 234, 239

L

- less 421
- less_equal 421
- lexicographical_compare() 310
- linear complexity 286,287
- list 131
 - see container
 - back() 131
 - begin() 131
 - clear() 137
 - constructor 134
 - empty() 131
 - end() 131
 - erase() 136
 - example 133
 - front() 131
 - insert() 135
 - iterators 129, 130
 - merge() 141
 - pop_back() 137
 - pop_front() 137
 - push_back() 135, 136
 - push_front() 136,
 - remove() 137
 - reverse() 142
 - size() 131
 - sort() 142
 - splice() 140, 141
 - splice functions 141
 - unique() 137
- logarithmic complexity 287
- logical_and 422
- logical_not 422
- logical_or 422
- lower_bound() 375
 - for maps 241
 - for sets 235

M

- make_heap() 180
- malloc_alloc
 - for allocators 54
- __malloc_alloc_template 56, 57
- map 237, 239
 - see container
 - < 241
 - == 241

- [] 240, 242
- begin() 240
- clear() 241
- constructors 240
- count() 241
- empty() 240
- end() 240
- equal_range() 241
- erase() 241
- example 242
- find() 241
- insert() 240, 241
- iterators 239
- lower_bound() 241
- max_size() 240
- rbegin() 240
- rend() 240
- size() 240
- sorting criterion 238
- swap() 240
- upper_bound() 241
- max() 312
- max_element() 352
- max_size()
 - for deque 151
 - for maps 240
 - for red black tree 221
 - for sets 235
- member function adapter 456
 - mem_fun 431,433,449,456,459
 - mem_fun_ref 431,433,449,456,460
- mem_fun 431,433,449,456,459
- mem_fun1_ref_t 433,449,459
- mem_fun1_t 433,449,459
- mem_fun_ref 431,433,449,456,460
- mem_fun_ref_t 433,449,458
- mem_fun_t 433,449,458
- memmove() 75
- <memory> 50,70
- memory pool 54,60,66,69
- merge() 352
 - for lists 141
- merge sort 412
- min() 312
- min_element() 354
- minus 418
- mismatch() 313
- modulus 418
- multimap 246

- constructors 246
- insert() 246
- multiplies 418
- multiset 245
 - constructor 245
 - insert() 245
- mutating algorithms 291

N

- negate 418
- new 44,45,48,49,53,58
- next_permutation() 380
- n-log-n complexity 392,396,412
- not1 433, 449, 451
- not2 433, 449, 451
- not_equal_to 420
- nth_element() 409
- numeric
 - algorithms 298
- <numeric> 298

O

- O(n) 286
- Open Closed Principle xvii
- Open Source 8
- ostream iterator 426
- output iterator 92,95
- output_iterator 95

P

- pair 237
 - constructor 237
 - first 237
 - first_type 237
 - second 237
 - second_type 237
- partial_sort() 386
- partial_sort_copy() 386
- partial_sum() 303
- partition() 354
- past-the-end iterator 39
- plus 418
- POD 73
- pop()
 - for priority queues 184
 - for queues 170
 - for stacks 168
- pop_back()

- for dequeues 163
- for lists 137
- for vectors 116,123
- pop_front()
 - for dequeues 157,163
 - for lists 137
- pop_heap() 176
- predicate 450
- prev_permutation() 382
- priority queue 183,184
 - constructor 184
 - empty() 184
 - example 185
 - pop() 184
 - push() 184
 - size() 184
 - size_type 184
 - top() 184
 - value_type 184
- priority_queue 184
- ptrdiff_t type 90
- ptr_fun 431,433,449,454,455
- push()
 - for priority queues 184
 - for queues 170
 - for stacks 168
- push_back()
 - called by inserters 435,
 - for dequeues 156
 - for lists 135,136
 - for vectors 116
- push_front()
 - called by inserters 435
 - for lists 136
- push_heap() 174

Q

- quadratic complexity 286
- queue 169, 170
 - < 170, 171
 - == 170
 - back() 170
 - empty() 170
 - example 171
 - front() 170
 - pop() 170
 - push() 170
 - size() 170
 - size_type 170

value_type 170
quicksort 392

R

rand() 384
random access iterator 92,95
random_access_iterator 95
random_shuffle() 383
rbegin()
 for maps 240
 for sets 235
read()
 for input streams 443
reallocation
 for vectors 115,122,123
rebind
 for allocators 44,46
red black tree 208, 218
 begin() 221
 constructor 220,222
 destructor 220
 empty() 221
 end() 221
 example 227
 find() 229
 insert_equal() 221
 insert_unique() 221
 iterators 214
 max_size() 221
 member access 223
 rebalance 225
 rotate left 226
 rotate right 227
 size() 221
release()
 for auto_ptrs 81
remove() 357
 for lists 137
remove_copy() 357
remove_copy_if() 358
remove_if() 357
rend()
 for maps 240
 for sets 235
replace_copy() 359
replace_copy_if() 360
replace_if() 359
reserve() 360
reset()

 for auto_ptrs 81
resize()
 for vectors 117
result_type 416,417
reverse() 360
 for lists 142
reverse_copy() 361
reverse iterator 425,426,437
 base() 440
reverse_iterator 440
Richard Stallman 7
rotate() 361
rotate_copy() 365

S

search() 365
search_n() 366
second
 for pairs 237
second_argument_type 417
second_type
 for pairs 237
sequence container 113
set 233
 see container
 < 236
 == 236
 begin() 235
 clear() 235
 constructors 234
 count() 235
 empty() 235
 end() 235
 equal_range() 236
 erase() 235
 example 236
 find() 235
 insert() 235
 iterators 234
 lower_bound() 235
 max_size() 235
 rbegin() 235
 rend() 235
 size() 235
 sorting criterion 233
 swap() 235
 upper_bound() 236
set_difference() 334
set_intersection() 333

- `set_symmetric_difference()` 336
- `set_union()` 331
- `simple_alloc`
 - for allocators 54
- `size()`
 - for deques 151
 - for lists 131
 - for maps 240
 - for priority queues 184
 - for queues 170
 - for red black tree 221
 - for sets 235
 - for stacks 168
 - for vectors 116
- `size_type`
 - for priority queues 184
 - for queues 170
 - for stacks 168
- `slist` 186, 190
 - see container
 - `begin()` 191
 - constructor 190
 - destructor 190
 - difference with list 186
 - `empty()` 191
 - `end()` 191, 194
 - example 191
 - `front()` 191
 - iterators 188
 - `pop_front()` 191
 - `push_front()` 191
 - `size()` 191
 - `swap()` 191
- smart pointer
 - `auto_ptr` 81
- `sort()` 389
 - for lists 142
- `sort_heap()` 178
- `splice()`
 - for lists 140, 141
- stack 167
 - `<` 167, 168
 - `==` 167, 168
 - `empty()` 168
 - example 168
 - `pop()` 168
 - `push()` 168
 - `size()` 168
 - `size_type` 168

- `top()` 168
- `value_type` 168
- standard template library 73
 - see STL
- `<stl_config.h>` 20
- STL implementation
 - HP implementation 9
 - PJ implementation 10
 - RW implementation 11
 - SGI implementation 13
 - STLport implementation 12
- stream iterator 426, 442
 - end-of-stream 428, 443
- subscript operator
 - for deques 151
 - for maps 240, 242
 - for vectors 116
- suffix
 - `_copy` 293
 - `_if` 293
- `swap()` 67
 - for maps 240
 - for sets 235

T

- tags
 - for iterators 95
- `top()`
 - for priority queues 184
 - for stacks 168
- traits
 - for iterators 87
 - for types 103
- `transform()` 369
- tree
 - AVL trees 203
 - balanced binary (search) trees 203
 - binary (search) trees 200
 - Red Black trees 208

U

- `unary_function` 416
- `unary_negate` 451
- `unary_predicate` 450
- `uninitialized_copy()` 70, 73
- `uninitialized_fill()` 71, 75
- `uninitialized_fill_n()` 71, 72
- `unique()` 370

for lists 137
unique_copy() 371
upper_bound() 377
for maps 241
for sets 236

V

value_comp() 235,240
value_compare 234,239
value_type
for allocators 45
for priority queues 184
for queues 170
for stacks 168
for vectors 115
vector 115
see container
[] 116
allocate_and_fill() 117
as dynamic array 115
back() 116
begin() 116
capacity 118
capacity() 116
clear() 117,124
constructor 116
destructor 116
difference_type 115
empty() 116
end() 116
erase() 117, 123
example 119
front() 116
header file 115
insert() 124
iterator operator ++ 117
iterator operator -- 117
iterator 115
iterators 117
pointer 115
pop_back() 116, 123
push_back() 116
reallocation 115,123
reference 115
resize() 117
size() 116
size_type 115
value_type 115
<vector> 115

W

X

Y

