YYYYYYYYYYYYYYYYYYYYYYYYYYYYYY

STL 源码剖析

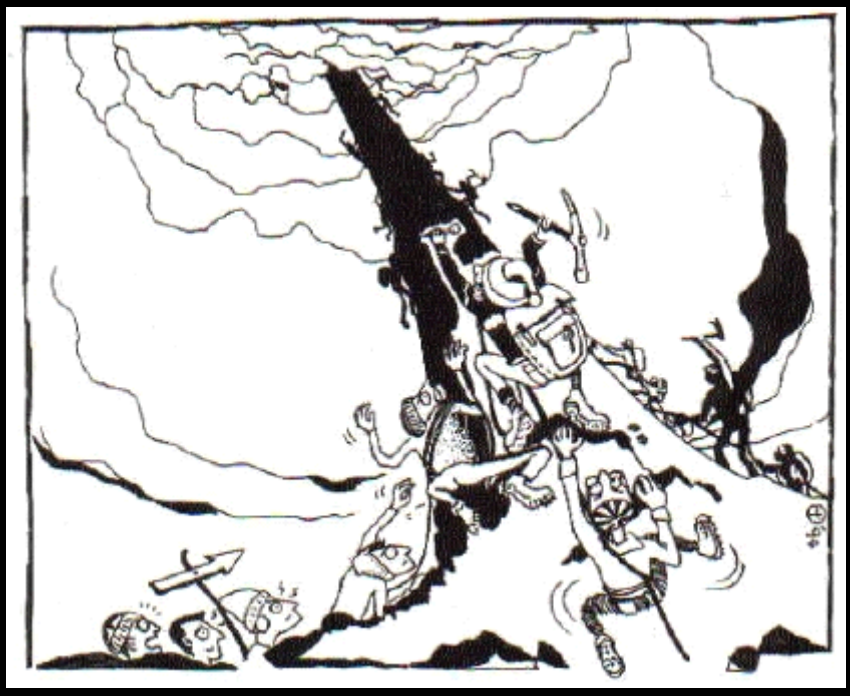
侯捷

The Annotated STL Source

无限延伸你的视野

STL源码剖析

The Annotated STL Source (using SGI STL)



SGI STL 源码剖析

The Annotated STL Sources

向专家学习
强型检验、内存管理、算法、数据结构、
及 STL 各类组件之实作技术

侯 捷 着

源码之前
了无秘密

献给每一位对 GP/STL 有所渴望的人

天下大事 必作于细

– 侯 捷 –

庖丁解牛[1]

侯捷自序

这本书的写作动机，纯属偶然。

2000 年下半，我开始为计划中的《泛型思维》一书陆续准备并热身。为了对泛型编程技术以及 STL 实作技术有更深的体会，以便在讲述整个 STL 的架构与应用时更能虎虎生风，我常常深入到 STL 源码去刨根究底。2001/02 的某一天，我突然有所感触：既然花了大把精力看过 STL 源码，写了眉批，做了整理，何不把它再加一点功夫，形成一个更完善的面貌后出版？对我个人而言，一份批注详尽的 STL 源码，价值不扉；如果我从中获益，一定也有许多人能够从中获益。

这样的念头使我极度兴奋。剖析大架构本是侯捷的拿手，这个主题又可以和《泛型思维》相呼应。于是我便一头栽进去了。

我选择 SGI STL 做为剖析对象。这份实作版本的可读性极佳，运用极广，被选为 GNU C++ 的标准链接库，又开放自由运用。愈是细读 SGI STL 源码，愈令我震惊抽象思考层次的落实、泛型编程的奥妙、及其效率考虑的绵密。不仅最为人广泛运用的各种数据结构（data structures）和算法（algorithms）在 STL 中有良好的实现，连内存配置与管理也都重重考虑了最佳效能。一切的一切，除了实现软件积木的高度复用性，让各种组件（components）得以灵活搭配运用，更考虑了实用上的关键议题：效率。

---

[1] 庄子养生主：「彼节间有间，而刀刃者无厚；以无厚入有间，恢恢乎其于游刃必有余地矣。」侯捷不让，以此自况。

这本书不适合 C++ 初学者，不适合 Genericity（泛型技术）初学者，或 STL 初学者。这本书也不适合带领你学习对象导向（Object Oriented）技术 — 是的，STL 与对象导向没有太多关连。本书前言清楚说明了书籍的定位和合适的读者，以及各类基础读物。如果你的 Generic Programming/STL 实力足以阅读本书所呈现的源码，那么，恭喜，你踏上了基度山岛，这儿有一座大宝库等着你。源码之前了无秘密，你将看到 vector 的实作、list 的实作、heap 的实作、deque 的实作、RB-tree 的实作、hash-table 的实作、set/map 的实作；你将看到各种算法（排序、搜寻、排列组合、数据搬移与复制...）的实作；你甚至将看到底层的 memory pool 和高阶抽象的 traits 机制的实作。那些数据结构、那些算法、那些重要观念、那些编程实务中最重要最根本的珍宝，那些蜇伏已久彷佛已经还给老师的记忆，将重新在你的脑中闪闪发光。

人们常说，不要从轮子重新造起，要站在巨人的肩膀上。面对扮演轮子角色的这些 STL 组件，我们是否有必要深究其设计原理或实作细节呢？答案因人而异。从应用的角度思考，你不需要探索实作细节（然而相当程度地认识底层实作，对实务运用有绝对的帮助）。从技术研究与本质提升的角度看，深究细节可以让你彻底掌握一切；不论是为了重温数据结构和算法，或是想要扮演轮子角色，或是想要进一步扩张别人的轮子，都可因此获得深厚扎实的基础。

天下大事，必作于细！

但是别忘了，参观飞机工厂不能让你学得流体力学，也不能让你学会开飞机。然而如果你会开飞机又懂流体力学，参观飞机工厂可以带给你最大的乐趣和价值。

The Annotated STL Sources

我开玩笑地对朋友说，这本书出版，给大学课程中的「数据结构」和「算法」两门授课老师出了个难题。几乎所有可能的作业题目（复杂度证明题除外），本书都有了详尽的解答。然而，如果学生能够从庞大的 SGI STL 源码中干净抽出某一部份，加上自己的包装，做为呈堂作业，也足以证明你有资格获得学分和高分。事实上，追踪一流作品并于其中吸取养份，远比自己关起门来写个三流作品，价值高得多 — 我的确认为 99.99 % 的程序员所写的程序，在 SGI STL 面前都是三流水平。

侯捷 2001/05/30 新竹　台湾
http://www.jjhou.com　(繁体)
http://jjhou.csdn.net　(简体)
jjhou@jjhou.com

p.s. 以下三书互有定位，互有关联，彼此亦相呼应。为了不重复讲述相同的内容，我会在适当时候提醒读者在哪本书上获得更多数据：

> 《多型与虚拟》，内容涵括：C++ 语法、语意、对象模型，对象导向精神，小型 framework 实作，OOP 专家经验，设计样式（design patterns）导入。
>
> 《泛型思维》，内容涵括：语言层次（C++ templates 语法、Java generic 语法、C++ 运算子重载），STL 原理介绍与架构分析，STL 现场重建，STL 深度应用，STL 扩充示范，泛型思考。
>
> 《STL 源码剖析》，内容涵括：STL 所有组件之实作技术和其背后原理解说。

目录

庖[丁] 解牛（侯捷自序） i
目录 v
前言 xvii
本书定位 xvii
合适的读者 xviii
最佳阅读方式 xviii
我所选择的剖析对象 xix
各章主题 xx
编译工具 xx
[中] 英术语的运用风格 xxi
英文术语采用原则 xxii
版面字形风格 xxiii
源码形式与[下] 载 xxiv
线[上] 服务 xxvi
推荐读物 xxvi
第 1 章 STL 概论与版本简介 001
1.1 STL 概论 001
1.1.1 STL 的历史 003
1.1.2 STL 与 C++ 标准链接库 003

1.2 STL 六大组件 — 功能与运用 004
1.3 GNU 源码开放精神 007
1.4 HP STL 实作版本 009
1.5 P.J. Plauger STL 实作版本 010
1.6 Rouge Wave STL 实作版本 011
1.7 STLport 实作版本 012
1.8 SGI STL 实作版本 总览 013
1.8.1 GNU C++ header 档案分布 014
1.8.2 SGI STL 档案分布与简介 016
STL 标准表头档（无扩展名） 017
C++ 标准规格定案前，HP 规范的 STL 表头档（扩展名 .h） 017
SGI STL 内部档案（SGI STL 真正实作于此） 018
1.8.3 SGI STL 的组态设定（configuration） 019
1.9 可能令你困惑的 C++ 语法 026
1.9.1 stl_config.h # 的各种组态 027
组态 3：static template member 027
组态 5：class template partial specialization 028
组态 6：function template partial order 028
组态 7：explicit function template arguments 029
组态 8：member templates 029
组态 10：default template argument depend on previous template parameters 030
组态 11：non-type template parameters 031
组态：bound friend template function 032
组态：class template explicit specialization 034
1.9.2 暂时对象的产生与运用 036
1.9.3 静态常数整数成员在 class 内部直接初始化 037
in-class static const *integral* data member initialization

1.9.4 increment/decrement/dereference 运算子 037
1.9.5 「前闭后开」区间表示法 [ ) 039
1.9.6 function call 运算子 (operator()) 040

第 2 章 空间配置器 (allocator) 043
2.1 空间配置器的标准接口 043
2.1.1 设计一个阳春的空间配置器，JJ::allocator 044
2.2 具备次配置力 (sub-allocation) 的 SGI 空间配置器 047
2.2.1 SGI 标准的空间配置器，std::allocator 047
2.2.2 SGI 特殊的空间配置器，std::alloc 049
2.2.3 建构和解构基本工具：construct() 和 destroy() 051
2.2.4 空间的配置与释放，std::alloc 053
2.2.5 第一级配置器 __malloc_alloc_template 剖析 056
2.2.6 第二级配置器 __default_alloc_template 剖析 059
2.2.7 空间配置函式 allocate() 062
2.2.8 空间释放函式 deallocate() 064
2.2.9 重新充填 *free-lists* 065
2.2.10 记忆池 (memory pool) 066
2.3 内存基本处理工具 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) 概念与 traits 编程技法 079
3.1 迭代器设计思维 — STL 关键所在 079
3.2 迭代器是一种 smart pointer 080
3.3 迭代器相应型别 (associated types) 084
3.4 Traits 编程技法 — STL 源码门钥 085

Partial Specialzation（偏特化）的意义 086
3.4.1 迭代器相应型别之一 *value_type* 090
3.4.2 迭代器相应型别之二 *difference_type* 090
3.4.3 迭代器相应型别之三 *pointer_type* 091
3.4.4 迭代器相应型别之四 *reference_type* 091
3.4.5 迭代器相应型别之五 *iterator_category* 092
3.5 std::iterator class 的保证 099
3.6 iterator 相关源码整理重列 101
3.7 SGI STL 的私房菜：__type_traits 103

第 4 章 序列式容器（sequence containers） 113
4.1 容器概观与分类 113
4.1.1 序列式容器（sequence containers） 114
4.2 vector 115
4.2.1 vector 概述 115
4.2.2 vector 定义式摘要 115
4.2.3 vector 的迭代器 117
4.2.4 vector 的数据结构 118
4.2.5 vector 的建构与内存管理：constructor, push_back 119
4.2.6 vector 的元素操作：pop_back, erase, clear, 123
insert 128
4.3 list 128
4.3.1 list 概述 129
4.3.2 list 的节点（node） 129
4.3.3 list 的迭代器 131
4.3.4 list 的数据结构
4.3.5 list 的建构与内存管理：constructor, push_back, insert 132
4.3.6 list 的元素操作：push_front, push_back, erase, 136
pop_front,
pop_back, clear, remove, unique, splice, merge, reverse, sort

4.4 deque 143
4.4.1 deque 概述 143
4.4.2 deque 的* 控器 144
4.4.3 deque 的迭代器 146
4.4.4 deque 的数据结构 150
4.4.5 deque 的建构与内存管理 ：ctor, push_back, push_front 152
4.4.6 deque 的元素操作：pop_back, pop_front, clear, erase, 161
insert 167
4.5 stack 167
4.5.1 stack 概述 167
4.5.2 stack 定义式完整列表 168
4.5.3 stack 没有迭代器 168
4.5.4 以 list 做为 stack 的底层容器 169
4.6 queue 169
4.6.1 queue 概述 170
4.6.2 queue 定义式完整列表 171
4.6.3 queue 没有迭代器 171
4.6.4 以 list 做为 queue 的底层容器 172
4.7 heap（隐性表述，implicit representation） 172
4.7.1 heap 概述 174
4.7.2 heap 算法 174
push_heap 176
pop_heap 178
sort_heap 180
make_heap 181
4.7.3 heap 没有迭代器 181
4.7.4 heap 测试实例 183
4.8 priority-queue

4.8.1 priority-queue 概述 183
4.8.2 priority-queue 定义式完整列表 183
4.8.3 priority-queue 没有迭代器 185
4.8.4 priority-queue 测试实例 185
4.9 slist 186
4.9.1 slist 概述 186
4.9.2 slist 的节点 186
4.9.3 slist 的迭代器 188
4.9.4 slist 的数据结构 190
4.9.5 slist 的元素操作 191
第 5 章 关系型容器 (associated containers) 197
5.1 树的导览 199
5.1.1 二元搜寻树 (binary search tree) 200
5.1.2 平衡二元搜寻树 (balanced binary search tree) 203
5.1.3 AVL tree (Adelson-Velskii-Landis tree) 203
5.1.4 单旋转 (Single Rotation) 205
5.1.5 双旋转 (Double Rotation) 206
5.2 RB-tree (红黑树) 208
5.2.1 安插节点 209
5.2.2 一个由上而下的程序 212
5.2.3 RB-tree 的节点设计 213
5.2.4 RB-tree 的迭代器 214
5.2.5 RB-tree 的数据结构 218
5.2.6 RB-tree 的建构与内存管理 221
5.2.7 RB-tree 的元素操作 223
元素安插动作 insert_equal 223
元素安插动作 insert_unique 224

真正的安插执行程序 __insert 224
调整 RB-tree（旋转及改变颜色） 225
元素的搜寻 find 229
5.3 set 233
5.4 map 237
5.5 multiset 245
5.6 multimap 246
5.7 hashtable 247
5.7.1 hashtable 概述 247
5.7.2 hashtable 的桶子（buckets）与节点（nodes） 253
5.7.3 hashtable 的迭代器 254
5.7.4 hashtable 的数据结构 256
5.7.5 hashtable 的建构与内存管理 258
安插动作（insert）与表格重整（resize） 259
判知元素的落脚处（bkt_num） 262
复制（copy_from）和整体删除（clear） 263
5.7.6 hashtable 运用实例（find, count） 264
5.7.7 hash functions 268
5.8 hash_set 270
5.9 hash_map 275
5.10 hash_multiset 279
5.11 hash_multimap 282
第 6 章 算法（algorithms） 285
6.1 算法概观 285
6.1.1 算法分析与复杂度表示 *O*( ) 286
6.1.2 STL 算法总览 288
6.1.3 mutating algorithms — 会改变操作对象之值 291

6.1.4 nonmutating algorithms — 不改变操作对象之值 292
6.1.5 STL 算法的一般型式 292
6.2 算法的泛化过程 294
6.3 数值算法 <stl_numeric.h> 298
6.3.1 运用实例 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 基本算法 <stl_algobase.h> 305
6.4.1 运用实例 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，强化效率无所不用其极 314
6.4.4 copy_backward 326
6.5 Set 相关算法（应用于已序区间） 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 算法：make_heap, pop_heap, push_heap, sort_heap 338
6.7 其它算法 338

6.7.1 单纯的数据处理 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 （应用于已序区间） 349
max_element 352
merge （应用于已序区间） 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 （应用于已序区间） 375

6.7.3 upper_bound （应用于已序区间） 377

6.7.4 binary_search （应用于已序区间） 379

6.7.5 next_permutation 380

6.7.6 prev_permutation 382

6.7.7 random_shuffle 383

6.7.8 partial_sort / partial_sort_copy 386
6.7.9 sort 389
6.7.10 equal_range（应用于已序区间） 400
6.7.11 inplace_merge（应用于已序区间） 403
6.7.12 nth_element 409
6.7.13 merge sort 411
第 7 章 仿函式（functor，另名 函式物件 function objects） 413
7.1 仿函式（functor）概观 413
7.2 可配接（adaptable）的关键 415
7.1.1 unary_function 416
7.1.2 binary_function 417
7.3 算术类（Arithmetic）仿函式 418
plus, minus, multiplies, divides, modulus, negate, identity_element
7.4 相对关系类（Relational）仿函式 420
equal_to, not_equal_to, greater, greater_equal, less, less_equal
7.5 逻辑运算类（Logical）仿函式 422
logical_and, logical_or, logical_not
7.6 证同（identity）、选择（select）、投射（project） 423
identity, select1st, select2nd, project1st, project2nd
第 8 章 配接器（adapter） 425
8.1 配接器之概观与分类 425
8.1.1 应用于容器，container adapters 425
8.1.2 应用于迭代器，iterator adapters 427
运用实例 428
8.1.3 应用于仿函式，functor adapters 429
运用实例

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 (istream_iterator, ostream_iterator) 442
8.4 function adapters 448
8.4.1 对传回值进行逻辑否定：not1, not2 450
8.4.2 对参数进行系结（绑定）：bind1st, bind2nd 451
8.4.3 用于函式合成：compose1, compose2（未纳入标准） 453
8.4.4 用于函式指标：ptr_fun 454
8.4.5 用于成员函式指标：mem_fun, mem_fun_ref 456

附录 A 参考数据与推荐读物（Bibliography） 461
附录 B 侯捷网站简介 471
附录 C STLport 的移植经验（by 孟岩） 473
索引 481

前言

本书定位

C++ 标准链接库是个伟大的作品。它的出现，相当程度地改变了 C++ 程序的风貌以及学习模式[1]。纳入 STL（Standard Template Library）的同时，标准链接库的所有组件，包括大家早已熟悉的 string、stream 等等，亦全部以 template 改写过。整个标准链接库没有太多的 OO（Object Oriented），倒是无处不存在 GP（Generic Programming）。

C++ 标准链接库中隶属 STL 范围者，粗估当在 80% 以上。对软件开发而言，STL 是尖端利兵，可以节省你许多时间。对编程技术而言，STL 是金柜石室 — 所有与编程工作最有直接密切关联的一些最被广泛运用的数据结构和算法，STL 都有实作，并符合最佳（或极佳）效率。不仅如此，STL 的设计思维，把我们提升到另一个思想高点，在那里，对象的耦合性（coupling）极低，复用性（reusability）极高，各种组件可以独立设计又可以灵活无罅地结合在一起。是的，STL 不仅仅是链接库，它其实具备 framework 格局，允许使用者加上自己的组件，与之融合并用，是一个符合开放性封闭（Open-Closed）原则的链接库。

从应用角度来说，任何一位 C++ 程序员都不应该舍弃现成、设计良好而又效率极佳的标准链接库，却「入太庙每事问」地事事物物从轮子造起 — 那对组件技术及软件工程是一大嘲讽。然而对于一个想要深度钻研 STL 以便拥有扩充能力的人，

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

相当程度地追踪 STL 源码是必要的功课。是的，对于一个想要充实数据结构与演算法等固有知识，并提升泛型编程技法的人，「入太庙每事问」是必要的态度，追踪 STL 源码则是提升功力的极佳路线。

想要良好运用 STL，我建议你看《*The C++ Standard Library*》by Nicolai M. Josuttis；想要严谨认识 STL 的整体架构和设计思维，以及 STL 的详细规格，我建议你看《*Generic Programming and the STL*》by Matthew H. Austern；想要从语法层面开始，学理与应用得兼，宏观与微观齐备，我建议你看《泛型思维》by 侯捷；想要深入 STL 实作技法，一窥大家风范，提升自己的编程功力，我建议你看你手上这本《STL 源码剖析》—— 事实上就在下笔此刻，你也找不到任何一本相同定位的书[2]。

合适的读者

本书不适合 STL 初学者（当然更不适合 C++ 初学者）。本书不是对象导向（Object Oriented）相关书籍。本书不适合用来学习 STL 的各种应用。

对于那些希望深刻了解 STL 实作细节，俾得以提升对 STL 的扩充能力，或是希望藉由观察 STL 源码，学习世界一流程式员身手，并藉此彻底了解各种被广泛运用之数据结构和算法的人，本书最适合你。

最佳阅读方式

无论你对 STL 认识了多少，我都建议你第一次阅读本书时，采循序渐进方式，遵循书中安排的章节先行浏览一遍。视个人功力的深浅，你可以或快或慢并依个人兴趣或需要，深入其中。初次阅读最好循序渐进，理由是，举个例子，所有容器（containers）的定义式一开头都会出现空间配置器（allocator）的运用，我可以在最初数次提醒你空间配置器于第 2 章介绍过，但我无法遍及全书一再一再提醒你。又例如，源码之中时而会出现一些全域函式呼叫动作，尤其是定义于 <stl_construct.h> 之中用于物件建构与解构的基本函式，以及定义于

The C++ Standard Template Library, by P.J.Plauger, Alexander A1. Stepanov, Meng Lee, David R. Musser, Prentice Hall 2001/03，与本书定位相近，但在表现方式上大有不同。

<stl_uninitialized.h> 之中 用于记忆体管理的基本函式，以及定义于 <stl_algobase.h> 之中 的各种基本算法。如果那些全域函式已经在先前章节介绍过，我很难保证每次都提醒你 — 那是一 种顾此失彼、苦不堪言的劳役，并且容易造成阅读上 的累赘。

我所选择的剖析对象

本书名为《STL 源码剖析》，然而 STL 实作品百花齐放，不论就技术面或可读性，皆有高下 之分。选择一 份好的实作版本，就学习而言当然是极为重要的。我选择的剖析对象是声名最着，也是我个人 评价最高的一 个产品：SGI（Silicon Graphics Computer Systems, Inc.）版本。这份由 STL 之父 Alexander Stepanov、经典书籍《*Generic Programming and the STL*》作者 Matthew H. Austern、STL 耆宿 David Musser 等人 投注心力的 STL 实作版本，不论在技术层次、源码组织、源码可读性上 ，均有卓越的表现。这份产品被纳为 GNU C++ 标准链接库，任何人 皆可从网际网络上 下 载 GNU C++ 编译器，从而获得整份 STL 源码，并获得自由运用的权力（详见 1.8 节）。

我所选用的是 cygnus[3] C++ 2.91.57 for Windows 版本。我并未刻意追求最新版本，一 来书籍不可能永远呈现最新的软件版本 — 软件更新永远比书籍改版快速，二 来本书的根本目的在建立读者对于 STL 巨观架构和微观技术的掌握，以及源码的阅读能力，这种核心知识的形成与源码版本的关系不是那么唇齿相依，三 来 SGI STL 实作品自从搭配 GNU C++2.8 以来已经十分稳固，变异极微，而我所选择的 2.91 版本，表现相当良好；四 来这个版本的源码比后来的版本更容易阅读，因为许多内部变量名称并不采用下 划线（underscore） — 下 划线在变数命名规范上 有其价值，但到处都是下 划线则对大量阅读相当不利。

网络上 有个 STLport（http://www.stlport.org）站点，提供一 份以 SGI STL 为蓝本的高度可移植性实作版本。本书附录 C 列有孟岩先生所写的文章，是一 份 STLport 移植到 Visual C++ 和 C++ Builder 的经验谈。

关于 cygnus、GNU 源码开放精神、以及自由软件基金会（FSF），请见 1.3 节介绍。

各章主题

本书假设你对 STL 已有基本认识和某种程度的运用经验。因此除了第一章略作介绍之外，立刻深入 STL 技术核心，并以 STL 六大组件（components）为章节之进行依据。以下是各章名称，这样的次序安排大抵可使每一章所剖析的主题能够于先前章节中获得充份的基础。当然，技术之间的关连错综复杂，不可能存在单纯的线性关系，这样的安排也只能说是尽最大努力。

第 1 章 STL 概论与实作版本简介

第 2 章 空间配置器（allocator）

第 3 章 迭代器（iterators）概念与 traits 编程技法

第 4 章 序列式容器（sequence containers）

第 5 章 关系型容器（associated containers）

第 6 章 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）

编译工具

本书主要探索 SGI STL 源码，并提供少量测试程序。如果测试程序只做标准的 STL 动作，不涉及 SGI STL 实作细节，那么我会在 VC6、CB4、cygnus 2.91 for Windows 等编译平台上分别测试它们。

随着对 SGI STL 源码的掌握程度增加，我们可以大胆做些练习，将 SGI STL 内部接口打开，或是修改某些 STL 组件，加上少量输出动作，以观察组件的运作过程。这种情况下，操练的对象既然是 SGI STL，我也就只使用 GNU C++ 来编译[4]。

[4] SGI STL 事实上是个高度可携产品，不限使用于 GNU C++。从它对各种编译器的环境组态设定（1.8.3 节）便可略知一二。网络上有一个 STLport 组织，不遗余力地将 SGI STL 移植到各种编译平台上。请参阅本书附录 C。

中英术语的运用风格

我曾经发表过一篇《技术引导乎 文化传承乎》的文章，阐述我对专业计算机书籍的中英术语运用态度。文章收录于侯捷网站 http://www.jjhou.com/article99-14.htm。以下简单叙述我的想法。

为了学术界和业界的习惯，也为了与全球科技接轨，并且也因为我所撰写的是供专业人士阅读的书籍而非科普读物，我决定适量保留专业领域中被朗朗上口的英文术语。朗朗上口与否，见仁见智，我以个人阅历做为抉择依据。

做为一个并非以英语为母语的族裔，我们对英文的阅读困难并不在单字，而在整句整段的文意。做为一项技术的学习者，我们的困难并不在术语本身（那只是个符号），而在术语背后的技术意义。

熟悉并使用原文术语，至为重要。原因很简单，在科技领域里，你必须与全世界接轨。中文技术书籍的价值不在于「建立本国文化」或「让它成为一本道地的中文书」或「完全扫除英汉字典的需要」。中文技术书籍的重要价值，在于引进技术、引导学习、扫平阅读障碍、增加学习效率。

绝大部份我所采用的英文术语都是名词。但极少数动词或形容词也有必要让读者知道原文（我会时而中英并列，并使用斜体英文），原因是：

> C++ 编译器的错误讯息并未中文化，万一错误讯息中出现以下字眼：*unresolved*, *instantiated*, *ambiguous*, *override*，而编写程序的你却不熟悉或不懂这些动词或形容词的技术意义，就不妙了。
> 有些动作关系到 library functions，而 library functions 的名称并未中文化，例如 *insert*, *delete*, *sort*。因此视状况而定，我可能会选择使用英文。
> 如果某些术语关系到语言关键词，为了让读者有最直接的感受与联想，我会采用原文，例如 static, private, protected, public, friend, inline, extern。

版面像一张破碎的脸？

大量中英夹杂的结果，无法避免造成版面的「破碎」。但为了实现合宜的表达方

式，牺牲版面的「全半文化」在所难免。我将尽量以版面手法来达到视觉上的顺畅，换言之我将采用不同的字形来代表不同属性的术语。如果把英文术语视为一种符号，这些半英夹杂但带有特殊字形的版面，并不会比市面上琳琅满目的许多应用软件图解使用手册来得突兀（而后者不是普遍为大众所喜爱吗 ）。我所采用的版面，都已经过一再试炼，过去以来获得许多读者的赞同。

英文术语采用原则

就我的观察，人们对于英文词或半文词的采用，隐隐有一个习惯：如果半文词发音简短（或至少不比英文词繁长）并且意义良好，那么就比较有可能被业界用于日常沟通；否则业界多半采用英文词。

例如，polymorphism 音节过多，所以意义良好的半文词「多型」就比较有机会被采用。例如，虚拟函式的发音不比 virtual function 繁长，所以使用这个半文词的人也不少。「多载」或「重载」的发音比 overloaded 短得多，意义又正确，用的人也不少。

但此并非绝对法则，否则就不会有绝大多数工程师说 data member 而不说「数据成员」、说 member function 而不说「成员函式」的情况了。

以下是本书采用原文术语的几个简单原则。请注意，并没有绝对的实践，有时候要看上下文情况。同时，容我再强调一次，这些都是基于我与业界和学界的接触经验而做的选择。

> 编程基础术语，采用半文。例如：函式、指针、变量、常数。本书的英文术语绝大部份都与 C++/OOP/GP（Generic Programming）相关。
>
> 简单而朗朗上口的词，视情况可能直接使用英文：input，output，lvalue，rvalue...
>
> 读者有必要认识的英文名词，不译：template，class，object，exception，scope，namespace。
>
> 长串、有特定意义、半译名称拗口者，不译：explicit specialization，partial specialization，using declaration，using directive，exception specialization。
>
> 运算子名称，不译：copy assignment 运算子，member access 运算子，arrow 运算子，dot 运算子，address of 运算子，dereference 运算子...

The Annotated STL Sources

业界惯用词，不译：constructor, destructor, data member, member function, reference。

涉及 C++ 关键词者，不译：public, private, protected, friend, static,

意义良好，发音简短，流传颇众的译词，用之：多型（polymorphism），虚拟函式（virtual function）、泛型（genericity）...

译后可能失掉原味而无法完全彰显原味者，中英并列。

重要的动词、形容词，时而中英并列：模棱两可（*ambiguous*），决议（*resolve*），改写（*override*），自变量推导（*argument deduced*），具现化（*instantiated*）。

STL 专用术语：采用中文，如迭代器（iterator）、容器（container）、仿函式（functor）、配接器（adapter）、空间配置器（allocator）。

数据结构专用术语：尽量采用英文，如 vector, list, deque, queue, stack, set, map, heap, binary search tree, RB-tree, AVL-tree, priority queue.

援用英文词，或不厌其烦地中英并列，获得的一项重要福利是：本书得以英文做为索引凭借。

http://www.jjhou.com/terms.txt 列有我个人整理的一份英中繁简术语对照表。

版面字型风格

中文

本文：细明 9.5pt

标题：**华康粗圆**

视觉加强：华康中黑

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

动词或形容词，*Times New Roman* 斜体 9.5pt，例如：*resolve, ambiguous, override, instantiated*

class 名称，Lucida Console 8.5pt，例如：stack, list, map

程序代码识别符号，Courier New 8.5pt，例如：int, min(SmallInt*, int)

长串术语，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*

运算子名称及某些特殊动作，Footlight MT Light 9.5pt，例如：copy assignment 运算子，dereference 运算子，address of 运算子，equality 运算子，function call 运算子，constructor, destructor, default constructor, copy constructor, virtual destructor, memberwise assignment, memberwise initialization

程序代码，Courier New 8.5pt，例如：

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

要在整本书中维护一贯的字形风格而没有任何疏漏，很不容易，许多时候不同类型的术语搭配起来，就形成了不知该用哪种字形的困扰。排版者顾此失彼的可能也不是没有。因此，请注意，各种字形的运用，只是为了让您阅读时有比较好的效果，其本身并不具其它意义。局部的一致性更重于全体的一致性。

源码形式与下载

SGI STL 虽然是可读性最高的一份 STL 源码，但其中并没有对实作程序乃至于实作技巧有什么文字批注，只偶而在档案最前面有一点点总体说明。虽然其符号名称有不错的规划，真要仔细追踪源码，仍然旷日费时。因此本书不但在正文之中解说其设计原则或实作技术，也直接在源码中加上许多批注。这些批注皆以蓝色标示。条件式编译（#ifdef）视同源码处理，函式呼叫动作以红色表示，巢状定义亦以红色表示。classes 名称、data members 名称和 member functions 名称大多以粗体表示。特别需要提醒的地方（包括 template 预设自变量、长度很长的巢状式定义）则加上灰阶底纹。例如：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class vector {
public:
  typedef T             value_type;
```

```
    typedef value_type* iterator;
  ...
  protected:
    // vector 采用简单的线性连续空间。以两个迭代器 start 和 end 分别指向头尾，
    // 并以迭代器 end_of_storage 指向容量尾端。容量可能比(尾-头)还大，
    // 多余的空间即备用空间。
    iterator start;
    iterator finish;
    iterator end_of_storage;

    void fill_initialize(size_type n, const T& value) {
      start = allocate_and_fill(n, value); // 配置空间并设初值
      finish = start + n;                          // 调整水位
      end_of_storage = finish;                     // 调整水位
    }
  ...
  };

  #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：

```
  // 以下配接器用来表示某个 Adaptable Binary Predicate 的逻辑负值
  template <class Predicate>
  class binary_negate
    : public binary_function<typename Predicate::first_argument_type        ,
                             typename Predicate::second_argument_type        ,
                             bool> {
  ...
  };
```

这些作法可能在某些地方有少许例外（或遗漏），唯一不变的原则就是尽量设法让读者一眼抓住源码重点。花花绿绿的颜色乍见之下或许不习惯，多看几眼你就会喜欢它 。这些经过批注的 SGI STL 源码以 Microsoft Word 97 档案格式，连同 SGI STL 源码，置于侯捷网站供自由下载。噢，是的，STL 涵盖面积广大，源码浩繁，考虑到书籍的篇幅，本书仅能就具代表性者加以剖析，如果你感兴趣的某些细节未涵盖于书中 ，可自行上网查阅这些经过整理的源码档案。

下 载这些档案并不会引发版权问题。详见 1.3 节关于自由软件基金会（FSF）、源码开放（open source）精神以及各种授权声明。

在线服务

侯捷网站（网址见于封底）是我的个人网站。我的所有作品，包括本书，都在此网站上提供服务，包括：

勘误和补充
技术讨论
程序代码下载
电子文件下载

附录 B 对侯捷网站有一些导引介绍。

推荐读物

详见附录 A。这些精选读物可为你建立扎实的泛型（Genericity）思维理论基础与扎实的 STL 实务应用能力。

1

STL 概论与版本简介

STL，虽然是一套链接库（library），却不只是一般印象中的链接库，而是一个有着划时代意义，背后拥有先进技术与深厚理论的产品。说它是产品也可以，说它是规格也可以，说是软件组件技术发展史上的一个大突破点，它也当之无愧。

1.1 STL 概论

长久以来，软件界一直希望建立一种可重复运用的东西，以及一种得以制造出「可重复运用的东西」的方法，让工程师 / 程序员的心血不致于随时间迁移、人事异动、私心欲念、人谋不臧[1]而烟消云散。从子程序（subroutines）、程序（procedures）、函式（functions）、类别（classes），到函式库（function libraries）、类别库（class libraries）、各种组件（components），从结构化设计、模块化设计、对象导向（object oriented）设计，到样式（patterns）的归纳整理，无一不是软件工程的漫漫奋斗史。

为的就是复用性（reusability）的提升。

复用性必须建立在某种标准之上 — 不论是语言层次的标准，或数据交换的标准，或通讯协议的标准。但是，许多工作环境下，就连软件开发最基本的数据结构（data structures）和算法（algorithms）都还迟迟未能有一套标准。大量程序员被迫从事大量重复的工作，竟是为了完成前人早已完成而自己手上并未拥有的程序代码。这不仅是人力资源的浪费，也是挫折与错误的来源。

---

[1] 后两者是科技到达不了的幽暗世界。就算 STL, COM, CORBA, OO, Patterns...也无能为力 。

为了建立数据结构和算法的一套标准，并且降低其间的耦合（coupling）关系以提升各自的独立性、弹性、交互操作性（相互合作性，interoperability），C++ 社群里诞生了 STL。

STL 的价值在两方面。低层次而言，STL 带给我们一套极具实用价值的零组件，以及一个整合的组织。这种价值就像 MFC 或 VCL 之于 Windows 软件开发过程所带来的价值一样，直接而明朗，令大多数人有最立即明显的感受。除此之外 STL 还带给我们一个高层次的、以泛型思维（Generic Paradigm）为基础的、系统化的、条理分明的「软件组件分类学（components taxonomy）」。从这个角度来看，STL 是个抽象概念库（library of abstract concepts），这些「抽象概念」包括最基础的 *Assignable*（可被赋值）、*Default Constructible*（不需任何自变量就可建构）、*Equality Comparable*（可判断是否等同）、*LessThan Comparable*（可比较大小）、*Regular*（正规）…，高阶一点的概念则包括 *Input Iterator*（具输入功能的迭代器）、*Output Iterator*（具输出功能的迭代器）、*Forward Iterator*（单向迭代器）、*Bidirectional Iterator*（双向迭代器）、*Random Access Iterator*（随机存取迭代器）、*Unary Function*（一元函式）、*Binary Function*（二元函式）、*Predicate*（传回真假值的一元判断式）、*Binary Predicate*（传回真假值的二元判断式）…，更高阶的概念包括 *sequence container*（序列式容器）、*associative container*（关系型容器）…。

STL 的创新价值便在于具体叙述了上述这些抽象概念，并加以系统化。

换句话说，STL 所实现的，是依据泛型思维架设起来的一个概念结构。这个以抽象概念（abstract concepts）为主体而非以实际类别（classes）为主体的结构，形成了一个严谨的接口标准。在此接口之下，任何组件有最大的独立性，并以所谓迭代器（iterator）胶合起来，或以所谓配接器（adapter）互相配接，或以所谓仿函式（functor）动态选择某种策略（policy 或 strategy）。

目前没有任何一种程序语言提供任何关键词（keyword）可以实质对应上述所谓的抽象概念。但是 C++ classes 允许我们自行定义型别，C++ templates 允许我们将

型别参数化，藉由两者结合并透过 traits 编程技法，形成了 STL 的绝佳温床[2]。

关于 STL 的所谓软件组件分类学，以及所谓的抽象概念库，请参考 [Austern98] ——没有任何一本书籍在这方面说得比它更好，更完善。

1.1.1 STL 的历史

STL 系由 Alexander Stepanov 创造于 1979 年前后，这也正是 Bjarne Stroustrup 创造 C++ 的年代。虽然 David R. Musser 于 1971 开始即在计算器几何领域中发展并倡导某些泛型程序设计观念，但早期并没有任何程序语言支持泛型编程。第一个支持泛型概念的语言是 Ada。Alexander 和 Musser 曾于 1987 开发出一套相关的 Ada library。然而 Ada 在美国国防工业以外并未被广泛接受，C++ 却如星火燎原般地在程序设计领域中攻城略地。当时的 C++ 尚未导入 template 性质，但 Alexander 却已经意识到，C++ 允许程序员透过指针以极佳弹性处理内存，这一点正是既要求一般化（泛型）又不失效能的一个重要关键。

更重要的是，必须研究并实验出一个「立基于泛型编程之上」的组件库完整架构。Alexander 在 AT&T 实验室以及惠普公司的帕罗奥图（Hewlett-Packard Palo Alto）实验室，分别实验了多种架构和算法公式，先以 C 完成，而后再以 C++ 完成。1992 年 Meng Lee 加入 Alex 的项目，成为 STL 的另一位主要贡献者。

贝尔（Bell）实验室的 Andrew Koenig 于 1993 年知道这个研究计划后，邀请 Alexander 于是年 11 月的 ANSI/ISO C++ 标准委员会会议上展示其观念。获得热烈回应。Alexander 于是再接再励于次年夏天的 Waterloo（滑铁卢[3]）会议开幕前，完成正式提案，并以压倒性多数一举让这个巨大的计划成为 C++ 标准规格的一部份。

1.1.2 STL 与 C++ 标准链接库

1993/09，Alexander Stepanov 和他一手创建的 STL，与 C++ 标准委员会有了第一

这么说有点因果混沌。因为 STL 的成形过程中也获得了 C++ 的一些重大修改支持，例如 template partial specialization。

不是威灵顿公爵击败拿破仑的那个地方，是加拿大安大略湖畔的滑铁卢市。

次接触。

当时 Alexander 在硅谷（圣荷西）给了 C++ 标准委员会一个演讲，讲题是：*The Science of C++ Programming*。题目很理论，但很受欢迎。1994/01/06 Alexander 收到 Andy Koenig（C++ 标准委员会成员，当时的 *C++ Standard* 文件审核编辑）来信，言明如果希望 STL 成为 C++ 标准链接库的一部份，可于 1994/01/25 前送交一份提案报告到委员会。Alexander 和 Lee 于是拼命赶工完成了那份提案。

然后是 1994/03 的圣地牙哥会议。STL 在会议上获得了很好的回响，但也有许多反对意见。主要的反对意见是，C++ 即将完成最终草案，而 STL 却是如此庞大，似乎有点时不我予。投票结果压倒性地认为应该给予这份提案一个机会，并把决定性投票延到下次会议。

下次会议到来之前，STL 做了几番重大的改善，并获得诸如 Bjarne Stroustrup、Andy Koenig 等人的强力支持。

然后便是滑铁卢会议。这个名称对拿破仑而言，标示的是失败，对 Alexander 和 Lee，以及他们的辛苦成果而言，标示的却是巨大的成功。投票结果，80 % 赞成，20 % 反对，于是 STL 进入了 C++ 标准化的正式流程，并终于成为 1998/09 定案的 C++ 标准规格的 C++ 标准链接库的大脉系。影响所及，原本就有的 C++ 链接库如 stream, string 等也都以 template 重新写过。到处都是 templates！ 整个 C++ 标准链接库呈现「春城无处不飞花」的场面。

Dr Dobb's Journal 曾于 1995/03 刊出一篇名为 *Alexander Stepanov and STL* 的访谈文章，对于 STL 的发展历史、Alexander 的思路历程、STL 纳入 C++ 标准链接库的过程，均有详细叙述，本处不再赘述。侯捷网站（见附录 B）上有孟岩先生的译稿「STL 之父访谈录」，欢迎观访。

1.2 STL 六大组件 功能与运用

STL 提供六大组件，彼此可以组合套用：

1. 容器（containers）：各种数据结构，如 vector, list, deque, set, map，

用来存放数据，详见本书 4，5 两章。从实作的角度看，STL 容器是一种 class template。就体积而言，这一部份很像冰山在海面下的比率。

2. 算法（algorithms）：各种常用算法如 sort，search，copy，erase…，详见第 6 章。从实作的角度看，STL 算法是一种 function template。
3. 迭代器（iterators）：扮演容器与算法之间的胶着剂，是所谓的「泛型指标」，详见第 3 章。共有五种类型，以及其它衍生变化。从实作的角度看，迭代器是一种将 operator*，operator->，operator++，operator-- 等指标相关操作予以多载化的 class template。所有 STL 容器都附带有自己专属的迭代器 —— 是的，只有容器设计者才知道如何巡访自己的元素。原生指标（native pointer）也是一种迭代器。
4. 仿函式（functors）：行为类似函式，可做为算法的某种策略（policy），详见第 7 章。从实作的角度看，仿函式是一种重载了 operator()的 class 或 class template。一般函式指标可视为狭义的仿函式。
5. 配接器（adapters）：一种用来修饰容器（containers）或仿函式（functors）或迭代器（iterators）接口的东西，详见第 8 章。例如 STL 提供的 queue 和 stack，虽然看似容器，其实只能算是一种容器配接器，因为它们的底部完全借重 deque，所有动作都由底层的 deque 供应。改变 functor 接口者，称为 function adapter，改变 container 接口者，称为 container adapter，改变 iterator 界面者，称为 iterator adapter。配接器的实作技术很难一言以蔽之，必须逐一分析，详见第 8 章。
6. 配置器（allocators）：负责空间配置与管理，详见第 2 章。从实作的角度看，配置器是一个实现了动态空间配置、空间管理、空间释放的 class template。

图 1-1 显示 STL 六大组件的交互关系。

由于 STL 已成为 C++ 标准链接库的大脉系，因此目前所有的 C++ 编译器一定支援有一份 STL。在哪里？就在相应的各个 C++ 表头档（headers）。是的，STL 并非以二进位码（binary code）面貌出现，而是以原始码面貌供应。按 *C++ Standard* 的规定，所有标准表头档都不再有扩展名，但或许是为了回溯相容，或许是为了内部组织规划，某些 STL 版本同时存在具扩展名和无扩展名的两份档案，例如 Visual C++ 的 **Dinkumware** 版本同时具备 <vector.h> 和 <vector>；某些 STL 版本只存在具副檔名的表头檔，例如 C++Builder 的 **RaugeWave** 版本只有

<vector.h>。某些 STL 版本不仅有¯ 线装配，还有= 线装配，例如 GNU C++ 的 **SGI** 版本不但有¯ 线的<vector.h> 和<vector>，还有= 线的<stl_vector.h>。

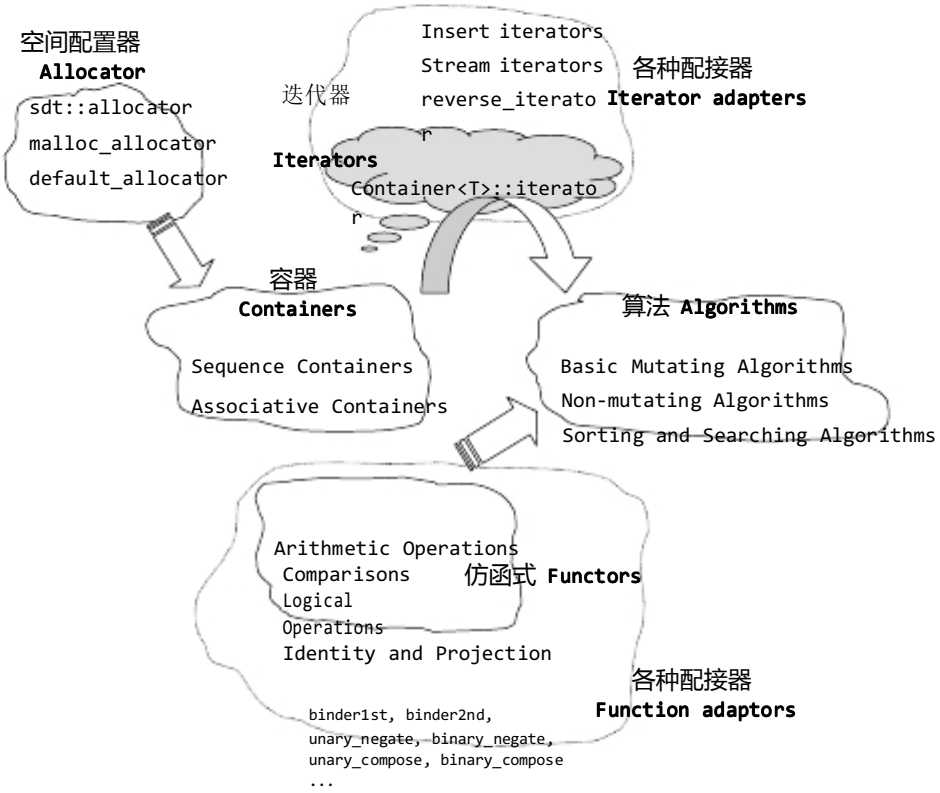


图 1-1 STL 六大组件的交互关系：Container 透过 Allocator 取得数据储存空间，Algorithm 透过 Iterator 存取 Container 内容，Functor 可以协助 Algorithm 完成不同的策略变化，Adapter 可以修饰或套接 Functor。

如果只是应用 STL，请各位读者务必从此养成良好习惯，遵照 C++ 规范，使用无扩展名的表头档[4]。如果进入本书层次，探究 STL 源码，就得清楚所有这些表头档的组织分布。1.8.2 节将介绍 GNU C++ 所附的 SGI STL 各个表头档。

[4] 某些编译器（例如 C++Builder）会在「前处理器」* 动手脚，使无扩展名的表头档名实际对应到有扩展名的表头档。这对使用者而言是透通的。

1.3 GNU 源码开放精神

全世界所有的 STL 实品，都源于 Alexander Stepanov 和 Meng Lee 完成的原始版本，这份原始版本属于 Hewlett-Packard Company（惠普公司）拥有。每一个表头檔都有一份声明，允许任何人任意运用、拷贝、修改、传布、贩卖这些码，无需付费，唯一的条件是必须将该份声明置于使用者新开发的档案内。

这种开放源码的精神，一般统称为 **open source**。本书既然使用这些免费开放的源码，也有义务对这种精神及其相关历史与组织，做一个简介。

开放源码的观念源自美国人 Richard Stallman[5]（理察 史托曼）。他认为私藏源码是一种违反人性的罪恶行为。他认为如果与他人分享源码，便可以让其它人从中学习，并回馈给原始创作者。封锁源码虽然可以程度不一地保障「智慧所可能衍生的财富」，却阻碍了使用者从中学习和修正错误的机会。Stallman 于 1984 离开麻省理工学院，创立自由软件基金会（Free Software Foundation[6]，简称 **FSF**），写下著名的 GNU 宣言（GNU Manifesto），开始进行名为 GNU 的开放改革计划。

GNU[7] 这个名称是计算机族的幽默展现，代表 **G**NU is **N**ot **U**nix。当时 Unix 是计算机界的主流操作系统，由 AT&T Bell 实验室的 Ken Thompson 和 Dennis Ritchie 创造。这原本只是一个学术上的练习产品，AT&T 将它分享给许多研究人员。但是当所有研究与分享使这个产品愈变愈美好时，AT&T 开始思考是否应该更加投资，并对从中获利抱以预期。于是开始要求大学校园内的相关研究人员签约，要求他们不得公开或透露 UNIX 源码，并赞助 Berkeley （柏克莱）大学继续强化 UNIX，导致后来发展出 **BSD**（Berkeley Software Distribution）版本，以及更后来的 FreeBSD、OpenBSD、NetBSD[8]...。

Richard Stallman 的个人网页见 http://www.stallman.org。

自由软件基金会 Free Software Foundation，见 http://www.gnu.org/fsf/fsf.html。

根据 GNU 的发音，或译为「革奴」，意思是从此革去被奴役的命运。音义俱佳。

FreeBSD 见 http://www.freebsd.org，OpenBSD 见 http://www.openbsd.org，NetBSD 见 http://www.netbsd.org。

Stallman 将 AT&T 的这种行为视为思想箝制，以及一种伟大传统的沦丧。在此之前，计算机界的氛围是大家无限制地共享各人成果（当然是指最根本的源码）。Stallman 认为 AT&T 对大学的暂助，只是一种微薄的施舍，拥有高权力的人才能吃到牛排和龙虾。于是他进行了他的反奴役计划，并称之为 GNU：**G**NU is **N**ot **U**nix。

GNU 计划中，早期最著名的软件包括 **Emacs** 和 **GCC**。前者是 Stallman 开发的一个极具弹性的文字编辑器，允许使用者自行增加各种新功能。后者是个 C/C++ 编译器，对所有 GNU 软件提供了平台的一致性与可移植性，是 GNU 计划的重要基石。GNU 计划晚近的著名软件则是 1991 年由芬兰人 Linus Torvalds 开发的 **Linux** 作业系统。这些软件当然都领受了许多使用者的心力回馈，才能更强固稳健。

GNU 以所谓的 **GPL**（General Public License[9]，广泛开放授权）来保护（或说控制）其成员：使用者可以自由阅读与修改 GPL 软件的源码，但如果使用者要传布借助 GPL 软件而完成的软件，他们必须也同意 GPL 规范。这种精神主要是强迫人们分享并回馈他们对 GPL 软件的改善。得之于人，舍于人。

GPL 对于版权（copyright）观念带来巨大的挑战，甚至被称为「反版权」（**copyleft**，又一个属于计算机族群的幽默）。GPL 带给使用者强大的道德束缚力量，「黏」性甚强，导致种种不同的反对意见，包括可能造成经济竞争力薄弱等等。于是其后又衍生出各种不同精义的授权，包括 Library GPL, Lesser GPL, Apache License, Artistic License, BSD License, Mozilla Public License, Netscape Public License。这些授权的共同原则就是「开放源码」。然而各种授权的拥护群众所渗杂的本位主义，加上精英份子难以妥协的个性，使「开放源码」阵营中的各个分支，意见纷歧甚至互相对立。其中最甚者为 GNU GPL 和 BSD License 的拥护者。

1998 年，自由软件社群企图创造出一个新名词 **open source** 来整合各方。他们组成了一个非财团法人的组织，注册一个标记，并设立网站。open source 的定义共有 9 条[10]，任何软件只要符合这 9 条，就可称呼自己为 open source 软件。

---

[9] GPL 的详细内容见 http://www.opensource.org/licenses/gpl-license.html。

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

本书所采用的 GCC 套件是 **Cygnus** C++2.91 for Windows，又称为 **EGCS** 1.1。GCC 和 Cygnus、EGCS 之间的关系常常令人混淆。Cygnus 是一家商业公司，包装并出售自由软件基金会所建构的软件工具，并贩卖各种服务。他们协助芯片厂商调整 GCC，在 GPL 的精神和规范下将 GCC 原始码的修正公布于世；他们提供 GCC 运作信息，并提升其运作效率，并因此成为 GCC 技术领域的最佳咨询对象。Cygnus 公司之于 GCC，地位就像 Red Hat（红帽）公司之于 Linux。虽然 Cygnus 持续地技术回馈并经济赞助 GCC，他们并不控制 GCC。GCC 的最终控制权仍然在 GCC 指导委员会（GCC Steering Committee）身上。

当 GCC 的发展进入第二版时，为了统一事权，GCC 指导委员会开始考虑整合 1997 成立的 EGCS（**E**xperimental/Enhanced **G**NU **C**ompiler **S**ystem）计划。这个计划采用比较开放的开发态度，比标准 GCC 涵盖更多优化技术和更多 C++ 语言性质。实验结果非常成功，因此 GCC 2.95 版反过头接纳了 EGCS 码。从那个时候开始，GCC 决定采用和 EGCS 一样的开发方式。自 1999 年起，EGCS 正式成为唯一的 GCC 官方维护机构。

1.4 HP 实作版本

HP 版本是所有 STL 实作版本的滥觞。每一个 HP STL 表头档都有如下一份声明，允许任何人免费使用、拷贝、修改、传布、贩卖这份软件及其说明文件，唯一需要遵守的是，必须在所有档案中加上 HP 的版本声明和运用权限声明。这种授权并不属于 GNU GPL 范畴，但属于 open source 范畴。

```
/*
   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 实作版本

P.J. Plauger 版本由 P.J. Plauger 发展，本书后继章节皆以 **PJ STL** 称呼此一版本。PJ 版本承继 HP 版本，所以它的每一个表头档都有 HP 的版本声明，此外还加上 P.J. Plauger 的个人版权声明：

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

这个产品既不属于 open source 范畴，更不是 GNU GPL。这么做是合法的，因为 HP 的版权声明并非 GPL，并没有强迫其衍生产品必须开放源码。

P.J. Plauger 版本被 Visual C++ 采用，所以当然你可以在 Visual C++ 的 "include" 子目录下（例如 C:\msdev\VC98\Include）找到所有 STL 表头档，但是不能公开它或修改它或甚至贩卖它。以我个人的阅读经验及测试经验，我对这个版本的可读性评价极低，主要因为其中的符号命名极不讲究，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); }
```

由于 Visual C++ 对 C++ 语言特性的支持不甚理想[11]，导致 PJ 版本的表现也受影响。

这项产品目前由 Dinkumware [12] 公司提供服务。

[11] 我个人对此有一份经验整理：http://www.jjhou.com/qa-cpp-primer-27.txt

[12] 详见 http://www.dinkumware.com

1.6 Rouge Wave 实作版本

RougeWave 版本由 Rouge Wave 公司开发，本书后继章节皆以 **RW STL** 称呼此一版本。RW 版本承继 HP 版本，所以它的每一个表头档都有 HP 的版本声明，此外还加上 Rouge Wave 的公司版权声明：

```
/*************************************************************************
 * (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.
 *
 *************************************************************************/
```

这份产品既不属于 open source 范畴，更不是 GNU GPL。这么做是合法的，因为 HP 的版权声明并非 GPL，并没有强迫其衍生产品必须开放源码。

Rouge Wave 版本被 C++Builder 采用，所以当然你可以在 C++Builder 的 "include" 子目录下（例如 C:\Inprise\CBuilder4\Include）找到所有 STL 表头档，但是

不能公开它或修改它或甚至贩卖它。就我个人 的阅读经验及测试经验，我要说，这个版本的可读性还不错，例如：

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

  return first;
}
```

但是像这个例子（class vector 的内部定义），源码中 夹杂特殊的常数，对阅读的顺畅性是一 大考验：

```
#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 述定义方式也不够清爽（请与稍后的 SGI STL 比较）。

C++Builder 对 C++ 语言特性的支持相当不错，连带地 给予了 RW 版本正面的影响。

1.7 STLport 实作版本

网络上 有个 STLport 站点，提供一 个以 SGI STL 为蓝本的高度可移植性实作版本。本书附录 C 列有孟岩先生所写的一 篇文章，介绍 STLport 移植到 Visual C++ 和 C++ Builder 的经验。SGI STL（下 节介绍）属于开放源码组织的一 员，所以 STLport 有权利那么做。

1.8 SGI STL 实作版本

SGI 版本由 Silicon Graphics Computer Systems, Inc. 公司发展，承继 HP 版本。所以它的每一个表头档也都有 HP 的版本声明。此外还加上 SGI 的公司版权声明。从其声明可知，它属于 open source 的一员，但不属于 GNU GPL（广泛开放授权）。

```
/*
 * 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 采用。你可以在 GCC 的 "include" 子目录下（例如 C:\cygnus\cygwin-b20\include\g++）找到所有 STL 表头档，并获准自由公开它或修改它或甚至贩卖它。就我个人的阅读经验及测试经验，我要说，不论是在符号命名或编程风格上，这个版本的可读性非常高，例如：

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

下面是对应于先前所列之 RW 版本的源码实例（class vector 的内部定义），也显得十分干净：

```
#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 对 C++ 语言特性的支持相当良好，在 C++ 主流编译器[中] 表现耀眼，连带[地] 给予了 SGI STL 正面影响。事实[上] SGI STL 为了高度移植性，已经考虑了不同编译器的不同的编译能力，详见 1.9.1 节。

SGI STL 也采用某些 GPL（广泛性开放授权）档案，例如 <std\complext.h>, <std\complext.cc>, <std\bastring.h>, <std\bastring.cc> 。这些檔案都有如[下] 的声明：

```
//   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 档案分布（按字母排序）

我手[上] 的 Cygnus C++ 2.91 for Windows 安装于磁盘目录 C:\cygnus。图 1-2 是这个版本的所有表头档，置于 C:\cygnus\cygwin-b20\include\g++，共 128 个档案，773,042 bytes：

```
algo.h                  algobase.h              algorithm
alloc.h                 builtinbuf.h            bvector.h
cassert                 cctype                  cerrno
cfloat                  ciso646                 climits
```

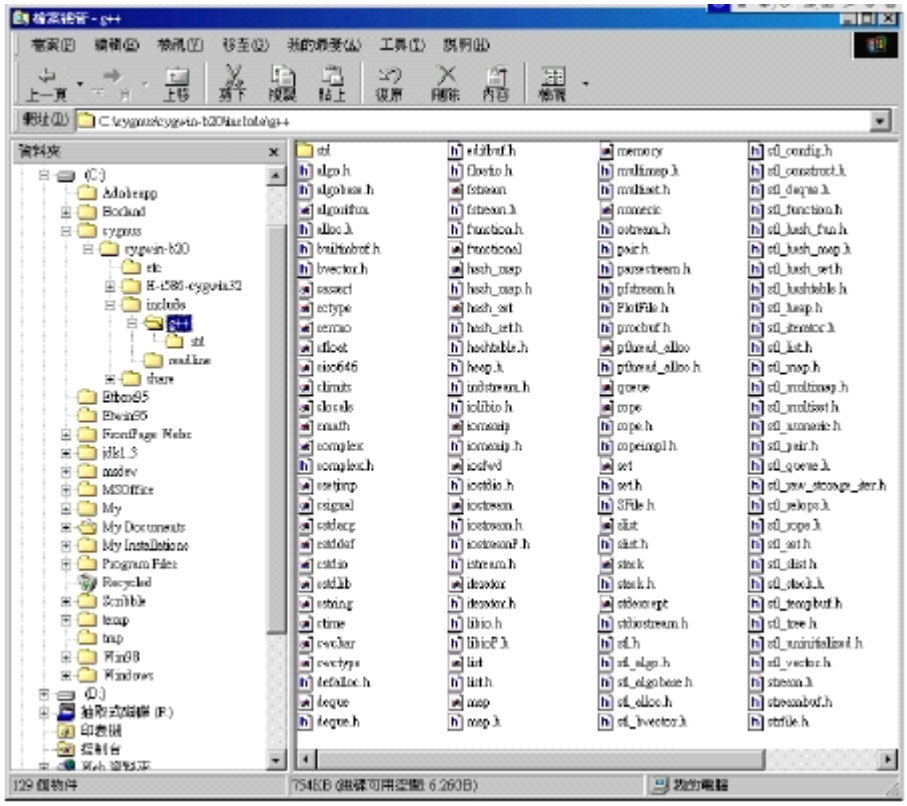
```
clocale                 cmath                   complex
complex.h               csetjmp                 csignal
cstdarg                 cstddef                 cstdio
cstdlib                 cstring                 ctime
cwchar                  cwctype                 defalloc.h
deque           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

子目录 **[std]** 内有 8 个档案，70,669 bytes：

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

图 1-2 Cygnus C++ 2.91 for Windows 的所有表头档

下面是以档案总管观察 Cygnus C++ 的档案分布：



1.8.2 SGI STL 档案分布与简介

上一小节所呈现的众多表头档，概略可分为五组：

C++ 标准规范的 C 表头档（无扩展名），例如 cstdio, cstdlib, cstring...

C++ 标准链接库不属于 STL 范畴者，例如 stream, string...相关档案。

STL 标准表头档（无扩展名），例如 vector, deque,list, map, algorithm, functional ...

C++ Standard 定案前，HP 所规范的 STL 表头档，例如 vector.h, deque.h, list.h, map.h, algo.h, function.h ...

SGI STL 内部档案（STL 真正实作于此），例如 stl_vector.h, stl_deque.h, stl_list.h, stl_map.h, stl_algo.h, stl_function.h …

其中前两组不在本书讨论范围内。后三组表头档详细列表于下。

(1) STL 标准表头档（无扩展名）

请注意，各档案之「本书章节」栏如未列出章节号码，表示其实际功能由「说明」栏中的 stl_xxx 取代，因此实际剖析内容应观察对应之 stl_xxx 档案所在章节，见稍后之第三列表。

文件名（按字母排序）	bytes	本书章节	说明
algorithm	1,337		ref. <stl_algorithm.h>
deque	1,350		ref. <stl_deque.h>
functional	762		ref. <stl_function.h>
hash_map	1,330		ref. <stl_hash_map.h>
hash_set	1,330		ref. <stl_hash_set.h>
iterator	1,350		ref. <stl_iterator.h>
list	1,351		ref. <stl_list.h>
map	1,329		ref. <stl_map.h>
memory	2,340	3.2	定义 auto_ptr，并含入 <stl_algobase.h>, <stl_alloc.h>, <stl_construct.h>, <stl_tempbuf.h>, <stl_uninitialized.h>, <stl_raw_storage_iter.h>
numeric	1,398		ref. <stl_numeric.h>
pthread_alloc	9,817	N/A	与 Pthread 相关的 node allocator
queue	1,475		ref. <stl_queue.h>
rope	920		ref. <stl_rope.h>
set	1,329		ref. <stl_set.h>
slist	807		ref. <stl_slist.h>
stack	1,378		ref. <stl_stack.h>
utility	1,301		含入 <stl_relops.h>, <stl_pair.h>
vector	1,379		ref. <stl_vector.h>

(2) *C++ Standard* 定案前，HP 规范的 STL 表头档（扩展名 .h）

请注意，各档案之「本书章节」栏如未列出章节号码，表示其实际功能由「说明」栏中的 stl_xxx 取代，因此实际剖析内容应观察对应之 stl_xxx 档案所在章节，见稍后之第三列表。

文件名（按字母排序）	bytes	本书章节	说明
complex.h	141	N/A	复数，含入 <complex>
stl.h	305		含入 STL 标准表头档 <algorithm>, <deque>, <functional>, <iterator>, <list>, <map>, <memory>, <numeric>, <set>, <stack>, <utility>, <vector>
type_traits.h	8,888	3.7	SGI 独特的 type-traits 技法
algo.h	3,182		ref. <stl_algo.h>
algobase.h	2,086		ref. <stl_algobase.h>
alloc.h	1,216		ref. <stl_alloc.h>
bvector.h	1,467		ref. <stl_bvector.h>
defalloc.h	2,360	2.2.1	标准空间配置器 std::allocator，不建议使用。
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 的功能实作
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 内部私用檔案（SGI STL 真正实作于此）

文件名（按字母排序）	bytes	本书章节	说明
stl_algo.h	86,156	6	算法（数值类除外）
stl_algobase.h	14,105	6.4	基本算法 swap, min, max, copy, copy_backward, copy_n, fill, fill_n, mismatch, equal, lexicographical_compare
stl_alloc.h	21,333	2	空间配置器 std::alloc。
stl_bvector.h	18,205	N/A	bit_vector（类似标准的 bitset）
stl_config.h	8,057	1.9.1	针对各家编译器特性定义各种环境常数
stl_construct.h	2,402	2.2.3	建构/解构基本工具

			(construct(), destroy())
stl_deque.h	41,514	4.4	deque (双向开口的 queue)
stl_function.h	18,653	7	函式物件 (function object) 或称仿函式 (functor)
stl_hash_fun.h	2,752	5.6.7	hash function (杂凑函数，用于 hash-table)
stl_hash_map.h	13,552	5.8	以 hast-table 完成之 map, multimap
stl_hash_set.h	12,990	5.7	以 hast-table 完成之 set, multiset
stl_hashtable.h	26,922	5.6	hast-table (杂凑表)
stl_heap.h	8,212	4.7	heap 算法：push_heap, pop_heap, make_heap, sort_heap
stl_iterator.h	26,249	3, 8.4, 8.5	迭代器及其相关配接器。并定义迭代器 常用函式 advance(),distance()
stl_list.h	17,678	4.3	list (串行，双向)
stl_map.h	7,428	5.3	map (映射表)
stl_multimap.h	7,554	5.5	multi-map (多键映射表)
stl_multiset.h	6,850	5.4	multi-set (多键集合)
stl_numeric.h	6,331	6.3	数值类算法: accumulate, inner_product, partial_sum, adjacent_difference, power, iota.
stl_pair.h	2,246	5.4	pair (成对组合)
stl_queue.h	4,427	4.6	queue (队列), priority_queue (高权先行队列)
stl_raw_storage_iter.h	2,588	N/A	定义 raw_storage_iterator (一种 OutputIterator)
stl_relops.h	1,772	N/A	定义四个 templatized operators： operator!=, operator>, operator<=, operator>=
stl_rope.h	62,538	N/A	大型 (巨量规模) 的字符串
stl_set.h	6,769	5.2	set (集合)
stl_slist.h	20,524	4.9	single list (单向串行)
stl_stack.h	2,517	4.5	stack (堆栈)
stl_tempbuf.h	3,328	N/A	定义 temporary_buffer class， 应用于 <stl_algo.h>
stl_tree.h	35,451	5.1	Red Black tree (红黑树)
stl_uninitialized.h	8,592	2.3	内存管理基本工具： uninitialized_copy, uninitialized_fill, uninitialized_fill_n.
stl_vector.h	17,392	4.2	vector (向量)

1.8.3 SGI STL 的编译器组态设定 (configuration)

不同的编译器对 C++ 语言的支持程度不尽相同。做为一个希望具备广泛移植能力的链接库，SGI STL 准备了一个环境组态档 <stl_config.h>，其中定义许多常

数，标示某些状态的成立与否。所有 STL 表头档都会直接或间接含入这个组态档，并以条件式写法，让前处理器（pre-processor）根据各个常数决定取舍哪一段程序码。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 // 前处理器的条件判断式
template <class T>
struct __copy_dispatch<T*, T*>
{
  ...
};
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
```

<stl_config.h> 档案起始处有一份常数定义说明，然后即针对各家不同的编译器以及可能的不同版本，给予常数设定。从这里我们可以一窥各家编译器对标准 C++ 的支持程度。当然，随着版本的演进，这些状态都有可能改变。其中的状态 (3),(5),(6),(7),(8),(10),(11)，各于 1.9 节中分别在 VC6，CB4，GCC 三家编译器上测试过。

以下是 GNU C++ 2.91.57 <stl_config.h> 的完整内容：

```
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) 如果编译器没有定义 bool, true, false，就定义它们
//    (2) 如果编译器的标准链接库未支持 drand48() 函式，就定义 __STL_NO_DRAND48
//    (3) 如果编译器无法处理 static members of template classes，就定义
//          __STL_STATIC_TEMPLATE_MEMBER_BUG
//    (4) 如果编译器未支持关键词 typename，就将'typename' 定义为一个 null macro.
//    (5) 如果编译器支持 partial specialization of class templates，就定义
//          __STL_CLASS_PARTIAL_SPECIALIZATION.
//    (6) 如果编译器支持 partial ordering of function templates（亦称为
//          partial specialization of function templates），就定义
//          __STL_FUNCTION_TMPL_PARTIAL_ORDER
```

```
//   (7) 如果编译器允许我们在呼叫一个 function template 时可以明白指定其
//       template arguments，就定义 __STL_EXPLICIT_FUNCTION_TMPL_ARGS
//   (8) 如果编译器支持 template members of classes，就定义
//       __STL_MEMBER_TEMPLATES.
//   (9) 如果编译器不支持关键词 explicit，就定义'explicit' 为一个 null macro.
//   (10) 如果编译器无法根据前一个 template parameters 设定下一个 template
//        parameters 的默认值，就定义 __STL_LIMITED_DEFAULT_TEMPLATES
//   (11) 如果编译器针对 non-type template parameters 执行 function template
//        的自变量推导（argument deduction）时有问题，就定义
//        __STL_NON_TYPE_TMPL_PARAM_BUG.
//   (12) 如果编译器无法支持迭代器的 operator->，就定义
//        __SGI_STL_NO_ARROW_OPERATOR
//   (13) 如果编译器（在你所选择的模式中）支持 exceptions，就定义
//        __STL_USE_EXCEPTIONS
//   (14) 定义 __STL_USE_NAMESPACES 可使我们自动获得 using std::list; 之类的叙句
//   (15) 如果本链接库由 SGI 编译器来编译，而且使用者并未选择 pthreads
//        或其它 threads，就定义 __STL_SGI_THREADS.
//   (16) 如果本链接库由一个 WIN32 编译器编译，并且在多绪模式下，就定义
//        __STL_WIN32THREADS
//   (17) 适当地定义与 namespace 相关的 macros 如 __STD, __STL_BEGIN_NAMESPACE。
//   (18) 适当地定义 exception 相关的 macros 如 __STL_TRY, __STL_UNWIND。
//   (19) 根据 __STL_ASSERTIONS 是否定义，将 __stl_assert 定义为一个
//        测试动作或一个 null macro。

#ifdef _PTHREADS
# define __STL_PTHREADS
#endif

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

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

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

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

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

// 侯捷注：VC6 的版本号码是 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

// 侯捷注：Inprise Borland C++builder 也定义有此常数。
// C++Builder 的表现岂有如下所示这般差劲？
# 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：难道不该 #define typename class 吗？
# 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(); }
    // 侯捷注：以上 使用 stringizing operator # , 详见《多型与虚拟》第 4 章。
```

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

#endif /* __STL_CONFIG_H */

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

下面这个小程序，用来测试 GCC 的常数设定：

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

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

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

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

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

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

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

// case 6
...以下写法类似。详见档案 config.cpp（可自侯捷网站下载）。
}
```

执行结果如下，由此可窥见 GCC 对各种 C++ 特性的支持程度：

```
__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++ 语法

1.8 节所列出的几个状态常数，用来区分编译器对 C++ Standard 的支持程度。这几个状态，也正是许多程序员对于 C++ 语法最为困扰之所在。以下我便一一测试 GCC 在这几个状态上的表现。有些测试程序直接取材（并剪裁）自 SGI STL 源码，因此你可以看到最贴近 SGI STL 真面貌的实例。由于这几个状态所关系的，都是 template 自变量推导（argument deduction）、偏特化（partial specialization）之类的问题，所以测试程序只需完成 classes 或 functions 的接口，便足以测试状态是否成立。

本节所涵盖的内容属于 C++ 语言层次，不在本书范围之内。因此本节各范例程序只做测试，不做太多说明。每个程序最前面都会有一个批注，告诉你在《*C++ Primer*》

3/e 哪些章节有相关的语法介绍。

1.9.1 stl_config.h 中的各种组态（configurations）

以下所列组态编号与上一节所列的 <stl_config.h> 档案起头的批注编号相同。

组态 3： __STL_STATIC_TEMPLATE_MEMBER_BUG

```
//   file: 1config3.cpp
//   测试在 class template 中拥有 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:        // 纯粹为了方便测试，使用 public
  static int _data;
};

// 为 static data members 进行定义（配置内存），并设初值。
int testClass<int>::_data = 1;
int testClass<char>::_data = 2;

int main()
{
  // 以下，CB4 表现不佳，没有接受初值设定。
  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
}
```

组态 5： __STL_CLASS_PARTIAL_SPECIALIZATION.

```
//   file: 1config5.cpp
//   测试 class template partial specialization — 在 class template 的
//     一般化设计之外，特别针对某些 template 参数做特殊设计。
//   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;

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

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

// 特殊化设计
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*
}
```

组态 6： __STL_FUNCTION_TMPL_PARTIAL_ORDER

请注意，虽然 <stl_config.h> 檔案中声明，这个常数的意义就是 partial

specialization of function templates，但其实两者并不相同。前者意义如下所示，后者的实际意义请参考 C++ 语法书籍。

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

#include <iostream>
using namespace std;

class alloc {
};

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

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER // 只为说明。非本程序内容。
template <class T, class Alloc>
inline void swap(vector<T, Alloc>& x, vector<T, Alloc>& y) {
  x.swap(y);
}
#endif // 只为说明。非本程序内容。

// 以上节录自 stl_vector.h，灰色部份系源码中的条件编译，非本测试程序内容。

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

组态 7： __STL_EXPLICIT_FUNCTION_TMPL_ARGS

整个 SGI STL 内都没有用到此一常数定义。

状态 8： __STL_MEMBER_TEMPLATES

```
//   file: 1config8.cpp
//   测试 class template 之内可否再有 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()
}
```

组态 10： __STL_LIMITED_DEFAULT_TEMPLATES

```
//   file: 1config10.cpp
//   测试 template 参数可否根据前一个 template 参数而设定默认值。
//   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 <cstddef>              // for size_t
using namespace std;

class alloc {
};

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

// 根据前一个参数值 T，设定下一个参数 Sequence 的默认值为 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
}
```

组态 11： __STL_NON_TYPE_TMPL_PARAM_BUG

```
//   file: 1config11.cpp
//   测试 class template 可否拥有 non-type template 参数。
//   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
}
```

以下组态常数虽不在前列编号之内，却也是 <stl_config.h> 内的定义，并使用于整个 SGI STL 之中。有认识的必要。

组态： __STL_NULL_TMPL_ARGS (bound friend template friend)

<stl_config.h> 定义 __STL_NULL_TMPL_ARGS 如下：

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

这个组态常数常常出现在类似这样的场合 (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&);
...
};
```

展开后就变成了：

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

这种奇特的语法是为了实现所谓的 bound friend templates，也就是说 class template 的某个具现体 (instantiation) 与其 friend function template 的某个具现体有一对一的关系。下面是个测试程序：

```
//   file: 1config-null-template-arguments.cpp
//   test __STL_NULL_TMPL_ARGS in <stl_config.h>
//   ref. C++ Primer 3/e, p.834: bound friend function template
//   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 {
  // 写成这样是可以的
  friend bool operator== <T> (const stack<T>&, const stack<T>&);
  friend bool operator<  <T> (const stack<T>&, const stack<T>&);
  // 写成这样也是可以的
  friend bool operator== <T> (const stack&, const stack&);
  friend bool operator<  <T> (const stack&, const stack&);
  // 写成这样也是可以的
  friend bool operator== <> (const stack&, const stack&);
  friend bool operator<  <> (const stack&, const stack&);
  // 写成这样就不可以
  // 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...
}
```

组态： __STL_TEMPLATE_NULL (class template explicit specialization)

<stl_config.h> 定义了一个 __STL_TEMPLATE_NULL 如下：

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

这个组态常数常常出现在类似这样的场合：

```
// 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> { ... };
```

展开后就变成了：

```
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> { ... };
```

这是所谓的 class template explicit specialization。下面这个例子适用于 GCC 和 VC6，允许使用者不指定 template<> 就完成 explicit specialization。C++Builder 则是非常严格地要求必须完全遵照 C++ 标准规格，也就是必须明白写出 template<>。

```
//   file: 1config-template-exp-special.cpp
//   以下测试 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;

// 将 __STL_TEMPLATE_NULL 定义为 template<>，可以。
// 若定义为 blank，如下，则只适用于 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 暂时对象的产生与运用

所谓暂时对象，就是一种无名对象（unnamed objects）。它的出现如果不在程序员的预期之下（例如任何 pass by value 动作都会引发 copy 动作，于是形成一个暂时对象），往往造成效率上的负担[13]。但有时候刻意制造一些暂时对象，却又是使程序干净清爽的技巧。刻意制造暂时对象的方法是，在型别名称之后直接加一对小括号，并可指定初值，例如 Shape(3,5) 或 int(8)，其意义相当于唤起相应的 constructor 且不指定物件名称。STL 最常将此技巧应用于仿函式（functor）与算法的搭配上，例如：

```
// file: 1config-temporary-object.cpp
// 本例测试仿函式用于 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() 多载化。见 1.9.6 节
     { cout << elem << ' '; }
};

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

  // print<int>() 是一个暂时对象，不是一个函式呼叫动作。
  for_each(iv.begin(), iv.end(), print<int>());
}
```

最后一行便是产生「function template 具现体」print<int> 的一个暂时对象。这个对象将被传入 for_each() 之中起作用。当 for_each() 结束，这个暂时对象也就结束了它的生命。

[13] 请参考《*More Effective C++*》条款 19: Understand the origin of temporary objects.

1.9.3 静态常数整数成员在 class 内部直接初始化 in-class static constant integer initialization

如果 class 内含 const static *integral* data member，那么根据 C++ 标准规格，我们可以在 class 之内直接给予初值。所谓 *integral* 泛指所有整数型别，不单只是指 int。下面是个例子：

```
// file: 1config-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 运算子

increment/dereference 运算子在迭代器的实作上占有非常重要的地位，因为任何一个迭代器都必须实作出前进（*increment*，operator++）和取值（*dereference*，operator*）功能，前者还分为前置式（prefix）和后置式（postfix）两种，有非常规律的写法[14]。有些迭代器具备双向移动功能，那么就必须再提供 decrement 运算子（也分前置式和后置式两种）。下面是个范例：

[14] 请参考《*More Effective C++*》条款 6：Distinguish between prefix and postfix forms of increment and decrement operators

```
// file: 1config-operator-overloading.cpp
// vc6[x] cb4[o] gcc[o]
// vc6 的 friend 机制搭配 C++ 标准链接库，有臭虫。
#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); // 随着 class 的不同，此行应该有不同的动作。
    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); // 随着 class 的不同，此行应该有不同的动作。
    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± 转换动作告诉编译器，你确实要将 const int 转为 non-const lvalue.
    // 如果没有这样明白❀ 转型，有些编译器会给你警告，有些更严格的编译器会视为错误
```

```
  }

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
}
```

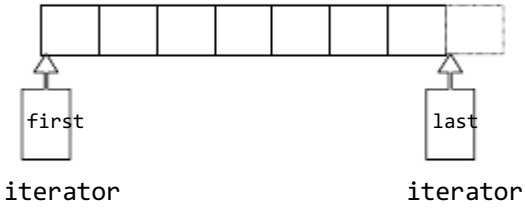
1.9.5 前闭后开区间表示法 [ )

任何一个 STL 算法，都需要获得由一对迭代器（泛型指标）所标示的区间，用以表示操作范围。这一对迭代器所标示的是个所谓的前闭后开区间[15]，以 [first, last) 表示。也就是说，整个实际范围从 first 开始，直到 last-1。迭代器 last 所指的是「最后一个元素的下一位置」。这种 *off by one*（偏移一格，或说 *pass the end*）的标示法，带来许多方便，例如下面两个 STL 算法的循环设计，就显得干净利落：

```
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] 这是一种半开（half-open）、后开（open-ended）区间。

前闭后开区间图示如下（注意，元素之间无需占用连续内存空间）：



1.9.6 function call 运算子（operator()）

很少人注意到，函式呼叫动作（C++ 语法中的左右小括号）也可以被多载化。

许多 STL 算法都提供两个版本，一个用于一般状况（例如排序时以递增方式排列），一个用于特殊状况（例如排序时由使用者指定以何种特殊关系进行排列）。像这种情况，需要使用者指定某个条件或某个策略，而条件或策略的背后由一整组动作构成，便需要某种特殊的东西来代表这「一整组动作」。

代表「一整组动作」的，当然是函式。过去 C 语言时代，欲将函式当做参数传递，唯有透过函式指标（pointer to function，或称 function pointer）才能达成，例如：

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

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

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

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

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

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

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

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

但是函式指标有缺点，最重要的是它无法持有自己的状态（所谓区域状态，local states），也无法达到组件技术[4] 的可配接性（adaptability）— 也就是无法再将某些修饰条件加诸于其上而改变其状态。

为此，STL 算法的特殊版本所接受的所谓「条件」或「策略」或「一整组动作」，都以仿函式形式呈现。所谓仿函式（functor）就是使用起来像函式一样的东西。如果你针对某个 class 进行 operator() 多载化，它就成为一个仿函式。至于要成为一个可配接的仿函式，还需要一些额外的努力（详见第 8 章）。

下面是一个将 operator() 多载化的例子：

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

// 由于将 operator() 多载化了，因此 plus 成了一个仿函式
template <class T>
struct plus {
    T operator()(const T& x, const T& y) const { return x + y; }
};

// 由于将 operator() 多载化了，因此 minus 成了一个仿函式
template <class T>
struct minus {
    T operator()(const T& x, const T& y) const { return x - y; }
};

int main()
{
  // 以下产生仿函式对象。
  plus<int> plusobj;
```

```
    minus<int> minusobj;

    // 以下 使用仿函式，就像使用一般函式一样。
    cout << plusobj(3,5) << endl;                    // 8
    cout << minusobj(3,5) << endl;                   // -2

    // 以下 直接产生仿函式的暂时对象（第一对小括号），并呼叫之（第二对小括号）。
    cout << plus<int>()(43,50) << endl;              // 93
    cout << minus<int>()(43,50) << endl;             // -7
}
```

上述的 plus<T> 和 minus<T> 已经非常接近 STL 的实作了，唯一差别在于它缺乏「可配接能力」。关于「可配接能力」，将在第 8 章详述。

2

空间配置器

allocator

以 STL 的运用角度而言，空间配置器是最不需要介绍的东西，它总是隐藏在一切组件（更具体地说是指容器，container）的背后，默默工作默默付出。但若以 STL 的实作角度而言，第一个需要介绍的就是空间配置器，因为整个 STL 的操作对象（所有的数值）都存放在容器之内，而容器一定需要配置空间以置放数据。不先掌握空间配置器的原理，难免在观察其它 STL 组件的实作时处处遇到挡路石。

为什么不说 allocator 是内存配置器而说它是空间配置器呢？因为，空间不一定是内存，空间也可以是磁盘或其它辅助储存媒体。是的，你可以写一个 allocator，直接向硬盘取空间[1]。以下介绍的是 SGI STL 提供的配置器，配置的对象，呃，是的，是内存 。

2.1 空间配置器的标准接口

根据 STL 的规范，以下是 allocator 的必要界面[2]：

```
// 以下各种 type 的设计原由，第三章详述。
allocator::value_type
allocator::pointer
allocator::const_pointer
allocator::reference
allocator::const_reference
allocator::size_type
allocator::difference_type
```

请参考 *Disk-Based Container Objects*, by Tom Nelson, *C/C++ Users Journal*, 1998/04

请参考 [Austern98], 10.3 节。

allocator::rebind

一个巢状的（nested）class template。class rebind<U> 拥有唯一成员 other，那是一个 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

传回某个对象的地址。算式 a.address(x) 等同于 &x。

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

传回某个 const 对象的地址。算式 a.address(x) 等同于 &x。

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

配置空间，足以储存 n 个 T 对象。第二自变量是个提示。实作上可能会利用它来增进区域性（locality），或完全忽略之。

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

归还先前配置的空间。

size_type allocator::**max_size**() const

传回可成功配置的最大量。

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

等同于 new(const void*) p) T(x)。

void allocator::**destroy**(pointer p)

等同于 p->~T()。

2.1.1 设计一个阳春的空间配置器，JJ::allocator

根据前述的标准介面，我们可以自行完成一个功能阳春、介面不怎么齐全的 allocator 如下：

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

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

namespace JJ
{

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

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

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

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

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

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

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

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

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

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

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

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

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

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

}  // end of namespace JJ

#endif // _JJALLOC_
```

将 JJ::allocator 应用于程序之[#] ，我们发现，它只能有限度[*] 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」是因为，PJ STL 未完全遵循 STL 规格，其所供应的许多容器都需要一个非标准的空间配置器接口 allocator::_Charalloc()。「只能有限度搭配 RW STL」则是因为，RW STL 在很多容器身上运用了缓冲区，情况复杂得多，JJ::allocator 无法与之兼容。至于完全无法应用于 SGI STL 是因为，SGI STL 在这个项目上根本上就逸脱了 STL 标准规格，使用一个专属的、拥有次层配置（sub-allocation）能力的、效率优越的特殊配置器，稍后有详细介绍。

我想我可以提前先做一点说明。事实上 SGI STL 仍然提供了一个标准的配置器介面，只是把它做了一层隐藏。这个标准接口的配置器名为 simple_alloc，稍后便会提到。

2.2 具备次配置力（sub-allocation）的 SGI 空间配置器

SGI STL 的配置器与众不同，也与标准规范不同，其名称是 alloc 而非 allocator，而且不接受任何自变量。换句话说如果你要在程序中明白采用 SGI 配置器，不能采用标准写法：

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

必须这么写：

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

SGI STL allocator 未能符合标准规格，这个事实通常不会对我们带来困扰，因为通常我们使用预设的空间配置器，很少需要自行指定配置器名称，而 SGI STL 的每一个容器都已经指定其预设的空间配置器为 alloc。例如下面的 vector 宣告：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class vector { ... };
```

2.2.1 SGI 标准的空间配置器，std::allocator

虽然 SGI 也定义有一个符合部份标准、名为 allocator 的配置器，但 SGI 自己

从未用过它，也不建议我们使用。主要原因是效率不彰，只把 C++ 的 ::operator new 和 ::operator delete 做一层薄薄的包装而已。下面是 SGI 的 std::allocator 全貌：

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

```
// 我们不赞成含入此档。这是原始的 HP default allocator。提供它只是为了
// 回溯相容。
//
// DO NOT USE THIS FILE 不要使用这个档案，除非你手上的容器是以旧式作法
// 完成 — 那就需要一个拥有 HP-style interface 的空间配置器。SGI STL 使用
// 不同的 allocator 界面。SGI-style allocators 不带有任何与对象型别相关
// 的参数；它们只回应 void* 指标（侯捷注：如果是标准接口，就会响应一个
// 「指向对象型别」的指针，T*）。此档并不含入于其它任何 SGI STL 表头档。

#ifndef DEFALLOC_H
#define DEFALLOC_H

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

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

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

template <class T>
class allocator {
public:
    // 以下各种 type 的设计原由，第三章详述。
    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，为什么最前面不需加上 template<>？
// 见 1.9.1 节的组态测试。注意，只适用于 GCC。
class allocator<void>
public:
    typedef void* pointer;
};

#endif
```

2.2.2 SGI 特殊的空间配置器，std::alloc

上一节所说的 allocator 只是基层内存配置/解放行为（也就是 ::operator new 和 ::operator delete）的一层薄薄包装，并没有考虑到任何效率上的强化。SGI 另有法宝供本身内部使用。

一般而言，我们所习惯的 C++ 内存配置动作和释放动作是这样：

```
class Foo { ... };
Foo* pf = new Foo;              // 配置内存，然后建构对象
delete pf;                      // 将对象解构，然后释放内存
```

这其中的 new 算式内含两阶段动作[3]：(1) 呼叫 ::operatornew 配置内存，(2) 呼叫 Foo::Foo() 建构物件内容。delete 算式也内含两阶段动作：(1) 呼叫

[3] 详见《多型与虚拟》2/e 第 1,3 章。

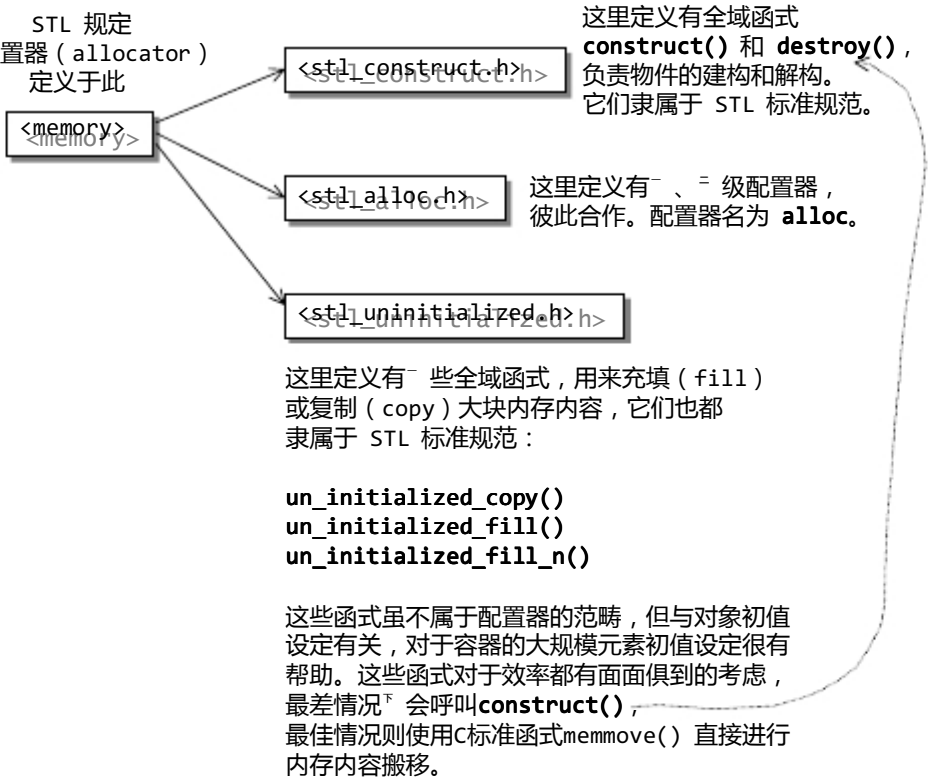
Foo::~Foo() 将对象解构，(2) 呼叫 ::operator delete 释放内存。

为了精密分工，STL allocator 决定将这两阶段动作区分开来。内存配置动作由 alloc:allocate() 负责，内存释放动作由 alloc::deallocate() 负责；对象建构动作由 ::construct() 负责，对象解构动作由 ::destroy() 负责。

STL 标准规格告诉我们，配置器定义于 <memory> 之中，SGI <memory> 内含以下两个档案：

```
#include <stl_alloc.h>                  // 负责内存空间的配置与释放
#include <stl_construct.h>              // 负责对象内容的建构与解构
```

内存空间的配置/释放与对象内容的建构/解构，分别着落在这两个档案身上。其中 <stl_construct.h> 定义有两个基本函式：建构用的 construct() 和解构用的 destroy()。一头栽进复杂的内存动态配置与释放之前，让我们先看清楚这两个函式如何完成对象的建构和解构。



2.2.3 建构和解构基本工具：construct() 和 destroy()

下面是 <stl_construct.h> 的部份内容（阅读程序代码的同时，请参考图 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; 唤起 T1::T1(value);
}

// 以下是 destroy() 第一版本，接受一个指标。
template <class T>
inline void destroy(T* pointer) {
    pointer->~T();                  // 唤起 dtor ~T()
}

// 以下是 destroy() 第二版本，接受两个迭代器。此函式设法找出元素的数值型别，
// 进而利用 __type_traits<> 求取最适当措施。
template <class ForwardIterator>
inline void destroy(ForwardIterator first, ForwardIterator last) {
  __destroy(first, last, value_type(first));
}

// 判断元素的数值型别（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());
}

// 如果元素的数值型别（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);
}

// 如果元素的数值型别（value type）有 trivial destructor...
template <class ForwardIterator>
inline void __destroy_aux(ForwardIterator, ForwardIterator, __true_type) {}

// 以下是 destroy() 第二版本针对迭代器为 char* 和 wchar_t* 的特化版
inline void destroy(char*, char*) {}
inline void destroy(wchar_t*, wchar_t*) {}
```

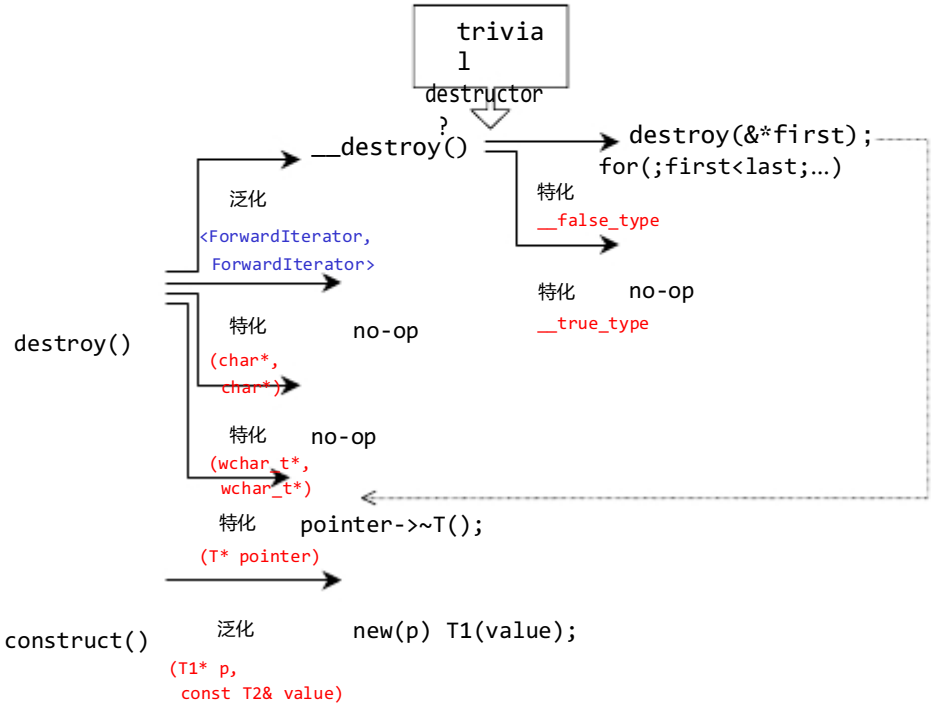


图 2-1 construct() 和 drestroy() 示意。

这两个做为建构、解构之用的函式被设计为全域函式，符合 STL 的规范[4]。此外 STL 还规定配置器必须拥有名为construct() 和 destroy() 的两个成员函式（见 2.1 节），然而真正在 SGI STL 中大显身手的那个名为 std::alloc 的配置器并未遵守此一规则（稍后可见）。

上述 construct() 接受一个指标 p 和一个初值 value，此函式的用途就是将初值设定到指标所指的空间上。C++ 的 placement new 运算子[5] 可用来完成此一任务。

destroy() 有两个版本，第一版本接受一个指标，准备将该指标所指之物解构掉。

请参考 [Austern98] 10.4.1 节。
请参考 [Lippman98] 8.4.5 节。

这很简单，直接呼叫该对象的解构式即可。第二版本接受 first 和 last 两个迭代器（所谓迭代器，第二章有详细介绍），准备将 [first,last) 范围内的所有物件解构掉。我们不知道这个范围有多大，万一很大，而每个物件的解构式都无关痛痒（所谓 *trivial* destructor），那么一次次呼叫这些无关痛痒的解构式，对效率是一种蕞伤。因此，这里首先利用 value_type() 获得迭代器所指物件的型别，再利用 __type_traits<T> 判别该型别的解构式是否无关痛痒。若是（__true_type），什么也不做就结束；若否（__false_type），这才以循环方式巡访整个范围，并在循环中每经历一个对象就呼叫第一个版本的 destroy()。

这样的观念很好，但 C++ 本身并不直接支持对「指标所指之物」的型别判断，也不支持对「对象解构式是否为 *trivial*」的判断，因此，上述的 value_type() 和 __type_traits<> 该如何实作呢？3.7 节有详细介绍。

2.2.4 空间的配置与释放，std::alloc

看完了内存配置后的对象建构行为，和内存释放前的对象解构行为，现在我们来看看内存的配置和释放。

对象建构前的空间配置，和对象解构后的空间释放，由 <stl_alloc.h> 负责，SGI 对此的设计哲学如下：

> 向 system heap 要求空间。
> 考虑多绪（multi-threads）状态。
> 考虑内存不足时的应变措施。
> 考虑过多「小型区块」可能造成的内存破碎（fragment）问题。

为了将问题控制在一定的复杂度内，以下的讨论以及所摘录的源码，皆排除多绪状态的处理。

C++ 的记忆体配置基本动作是 ::operator new()，记忆体释放基本动作是 ::operator delete()。这两个全域函式相当于 C 的 malloc() 和 free() 函式。是的，正是如此，SGI 正是以 malloc() 和 free() 完成内存的配置与释放。

考虑小型区块所可能造成的内存破碎问题，SGI 设计了双层级配置器，第一级配置器直接使用 malloc() 和 free()，第二级配置器则视情况采用不同的策略：当配置区块超过 128bytes，视之为「足够大」，便呼叫第一级配置器；当配置区块小于 128bytes，视之为「过小」，为了降低额外负担（overhead，见 2.2.6 节），便采用复杂的 memory pool 整理方式，而不再求助于第一级配置器。整个设计究竟只开放第一级配置器，或是同时开放第二级配置器，取决于 __USE_MALLOC[6] 是否被定义（唔，我们可以轻易测试出来，SGI STL 并未定义 __USE_MALLOC）：

```
# ifdef __USE_MALLOC
...
typedef __malloc_alloc_template<0> malloc_alloc;
typedef malloc_alloc alloc;          // 令 alloc 为第一级配置器
# else
...
// 令 alloc 为第二级配置器
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
#endif /* ! __USE_MALLOC */
```

其中 __malloc_alloc_template 就是第一级配置器，__default_alloc_template 就是第二级配置器。稍后分别有详细介绍。再次提醒你注意，**alloc** 并不接受任何 template 型别参数。

无论 **alloc** 被定义为第一级或第二级配置器，SGI 还为它再包装一个接口如下，使配置器的接口能够符合 STL 规格：

```
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)); }
};
```

其内部四个成员函式其实都是单纯的转呼叫，呼叫传入之配置器（可能是第一级

__USE_MALLOC 这个名称取得不甚理想，因为无论如何，最终总是使用 malloc().

也可能是第二级）的成员函式。这个接口使配置器的配置单位从 bytes 转为个别元素的大小（sizeof(T)）。SGI STL 容器全都使用这个 simple_alloc 接口，例如：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class vector {
protected:
  // 专属之空间配置器，每次配置一个元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;

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

一、二级配置器的关系，接口包装，及实际运用方式，可于图 2-2 略见端倪。

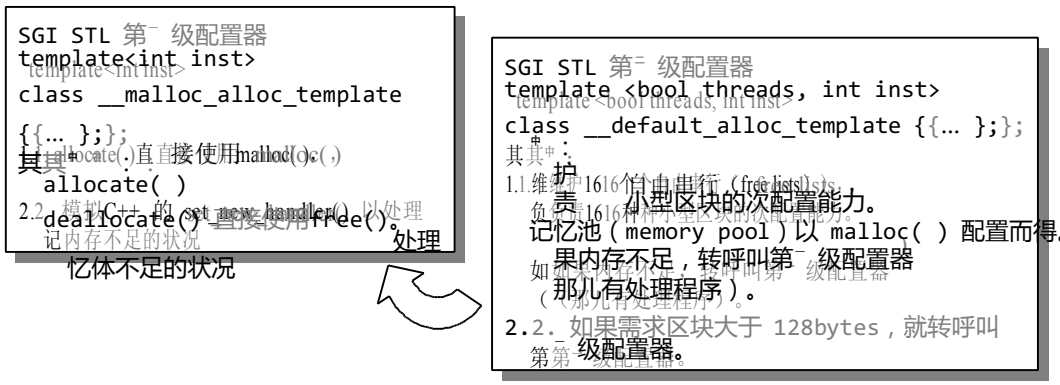


图 2-2a 第一级配置器与第二级配置器

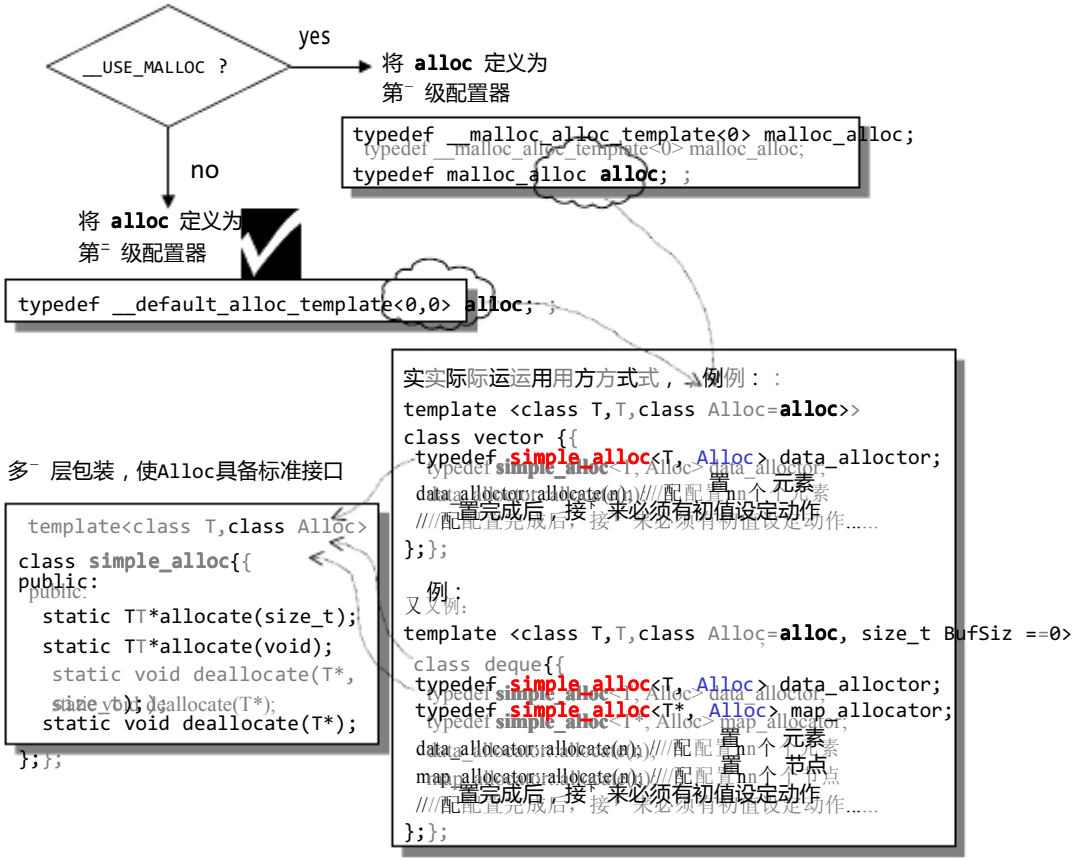


图 2-2b 第一级配置器与第二级配置器，其包装接口和运用方式

2.2.5 第一级配置器 __malloc_alloc_template 剖析

首先我们观察第一级配置器：

```
#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. 通常比稍后介绍的 default alloc 速度慢，
```

```
// 一般而言是 thread-safe，并且对于空间的运用比较高效（efficient）。
// 以下是第一级配置器。
// 注意，无「template 型别参数」。至于「非型别参数」inst，完全没派上用场。
template <int inst>
class __malloc_alloc_template {

private:
// 以下都是函式指标，所代表的函式将用来处理内存不足的情况。
// 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);                  // 第一级配置器直接使用 malloc()
    // 以下，无法满足需求时，改用 oom_malloc()
    if (0 == result) result = oom_malloc(n);
    return result;
}

static void deallocate(void *p, size_t /* n */)
{
    free(p); // 第一级配置器直接使用 free()
}

static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
    void * result = realloc(p, new_sz);                        // 第一级配置器直接使用 realloc()
    // 以下，无法满足需求时，改用 oom_realloc()
    if (0 == result) result = oom_realloc(p, new_sz);
    return result;
}

// 以下模拟 C++ 的 set_new_handler()。换句话说，你可以透过它，
// 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
// 初值为 0。有待客端设定。
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 (;;) {            // 不断尝试释放、配置、再释放、再配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();                // 呼叫处理例程，企图释放内存。
        result = malloc(n);                    // 再次尝试配置内存。
        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 (;;) {            // 不断尝试释放、配置、再释放、再配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();                // 呼叫处理例程，企图释放内存。
        result = realloc(p, n);                // 再次尝试配置内存。
        if (result) return(result);
    }
}

// 注意，以下直接将参数 inst 指定为 0。
typedef __malloc_alloc_template<0> malloc_alloc;
```

第一级配置器以 `malloc()`, `free()`, `realloc()` 等 C 函式执行实际的内存配置、释放、重配置动作，并实作出类似 C++ new-handler[7] 的机制。是的，它不能直接运用 C++ new-handler 机制，因为它并非使用 `::operator new` 来配置记忆体。

所谓 C++ new handler 机制是，你可以要求系统在内存配置需求无法被满足时，唤起一个你所指定的函式。换句话说一旦 `::operator new` 无法达成任务，在丢

[7] 详见《*Effective C++*》2e，条款 7：*Be prepared for out-of-memory conditions.*

出 *std::bad_alloc* 异常状态之前，会先呼叫由客端指定的处理例程。此处理例程通常即被称为 new-handler。new-handler 解决内存不足的作法有特定的模式，请参考《*Effective C++*》2e 条款 7。

注意，SGI 以 malloc 而非 ::operator new 来配置内存（我所能够想象的一个原因是历史因素，另一个原因是 C++ 并未提供相应于 realloc() 的内存配置动作），因此 SGI 不能直接使用 C++ 的 set_new_handler()，必须模拟一个类似的 set_malloc_handler()。

请注意，SGI 第一级配置器的 allocate() 和 realloc() 都是在呼叫 malloc() 和 realloc() 不成功后，改呼叫 oom_malloc() 和 oom_realloc()。后两者都有内循环，不断呼叫「内存不足处理例程」，期望在某次呼叫之后，获得足够的内存而圆满达成任务。但如果「内存不 足处理例程」并未被客端设定，oom_malloc() 和 oom_realloc() 便老实不客气[6] 呼叫 __THROW_BAD_ALLOC，丢出 *bad_alloc* 异常讯息，或利用 exit(1) 硬生生[7] 止程序。

记住，设计「内存不足处理例程」是客端的责任，设定「内存不足处理例程」也是客端的责任。再一次提醒你，「内存不足处理例程」解决问题的作法有着特定的模式，请参考 [Meyers98] 条款 7。

2.2.6 第二级配置器 __default_alloc_template 剖析

第二级配置器多了一些机制，避免太多小额区块造成内存的破碎。小额区块带来的其实不仅是内存破碎而已，配置时的额外负担（overhead）也是一大问题[8]。额外负担永远无法避免，毕竟系统要靠这多出来的空间来管理内存，如图 2-3。但是区块愈小，额外负担所占的比例就愈大、愈显得浪费。

[6] 请参考 [Meyers98] 条款 10：*write operator delete if you write operator new.*

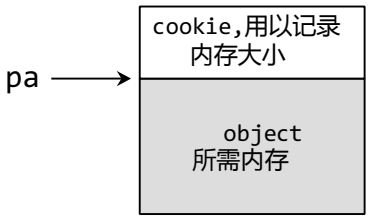


图 2-3 索求任何一块内存，都得有一些「税」要缴给系统。

SGI 第二级配置器的作法是，如果区块够大，超过 128 bytes，就移交第一级配置器处理。当区块小于 128 bytes，则以记忆池（memory pool）管理，此法又称为次层配置（sub-allocation）：每次配置一大块内存，并维护对应之自由串行（*free-list*）。下次若再有相同大小的内存需求，就直接从 *free-lists* 中拨出。如果客端释还小额区块，就由配置器回收到 *free-lists* 中 —— 是的，别忘了，配置器除了负责配置，也负责回收。为了方便管理，SGI 第二级配置器会主动将任何小额区块的记忆体需求量上调至 8 的倍数（例如客端要求 30 bytes，就自动调整为 32 bytes），并维护 16 个 *free-lists*，各自管理大小分别为 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 bytes 的小额区块。*free-lists* 的节点结构如下：

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

诸君或许会想，为了维护串行（lists），每个节点需要额外的指标（指向下一个节点），这不又造成另一种额外负担吗？你的顾虑是对的，但早已有好的解决办法。

注意，上述 obj 所用的是 union，由于 union 之故，从其第一字段观之，obj 可被视为一个指标，指向相同形式的另一个 obj。从其第二字段观之，obj 可被视为一个指标，指向实际区块，如图 2-4。一物二用的结果是，不会为了维护串行所必须的指针而造成内存的另一种浪费（我们正在努力樽节内存的开销呢）。

这种技巧在强型（strongly typed）语言如 Java 中行不通，但是在非强型语言如 C++ 中十分普遍[9]。

请参考 [Lippman98] p840 及 [Noble] p254。

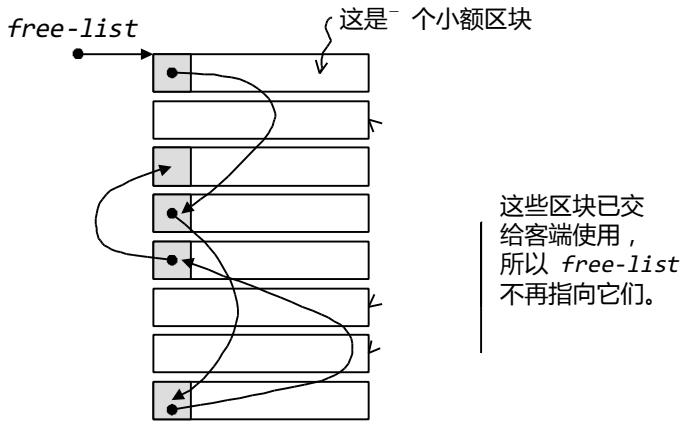


图 2-4 自由串行（*free-list*）的实作技巧

下面是第二级配置器的部份实作内容：

```
enum {__ALIGN = 8};              // 小型区块的上调边界
enum {__MAX_BYTES = 128}; // 小型区块的上限
enum {__NFREELISTS = __MAX_BYTES/__ALIGN}; // free-lists 个数

// 以下是第二级配置器。
// 注意，无「template 型别参数」，且第二参数完全没派上用场。
// 第一参数用于多绪环境下。本书不讨论多绪环境。
template <bool threads, int inst>
class __default_alloc_template {

private:
  // ROUND_UP() 将 bytes 上调至 8 的倍数。
  static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
  }
private:
  union obj {              // free-lists 的节点构造
        union obj * free_list_link;
        char client_data[1]; /* The client sees this. */
  };
private:
    // 16 个 free-lists
    static obj * volatile free_list[__NFREELISTS];
    // 以下函式根据区块大小，决定使用第 n 号 free-list。n 从 1 起算。
    static size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
  }
```

```
  // 传回一个大小为 n 的对象，并可能加入大小为 n 的其它区块到 free list.
  static void *refill(size_t n);
  // 配置一大块空间，可容纳 nobjs 个大小为 "size" 的区块。
  // 如果配置 nobjs 个区块有所不便，nobjs 可能会降低。
  static char *chunk_alloc(size_t size, int &nobjs);

  // Chunk allocation state.
  static char *start_free; // 记忆池起始位置。只在 chunk_alloc()中变化
  static char *end_free;               // 记忆池结束位置。只在 chunk_alloc()中变化
  static size_t heap_size;

public:
  static void * allocate(size_t n) { /* 详述于后 */ }
  static void deallocate(void *p, size_t n) { /* 详述于后 */ }
  static void * reallocate(void *p, size_t old_sz, size_t new_sz);
};

// 以下是 static data member 的定义与初值设定
template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

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

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

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

2.2.7 空间配置函式 allocate()

身为一个配置器，__default_alloc_template 拥有配置器的标准介面函式 allocate()。此函式首先判断区块大小，大于 128 bytes 就呼叫第一级配置器，小于 128 bytes 就检查对应的 *free list*。如果 *free list* 之内有可用的区块，就直接拿来用，如果没有可用区块，就将区块大小上调至 8 倍数边界，然后呼叫 refill()，准备为 *free list* 重新填充空间。refill() 将于稍后介绍。

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

```
    obj * result;

    // 大于 128 就呼叫第一级配置器
    if (n > (size_t) __MAX_BYTES)
        return(malloc_alloc::allocate(n));
    }
    // 寻找 16 个 free lists 中适当的一个
    my_free_list = free_list + FREELIST_INDEX(n);
    result = *my_free_list;
    if (result == 0) {
        // 没找到可用的 free list，准备重新填充 free list
        void *r = refill(ROUND_UP(n));                    // 下节详述
        return r;
    }
    // 调整 free list
    *my_free_list = result -> free_list_link;
    return (result);
};
```

区块自 *free list* 拨出的动作，如图 2-5。

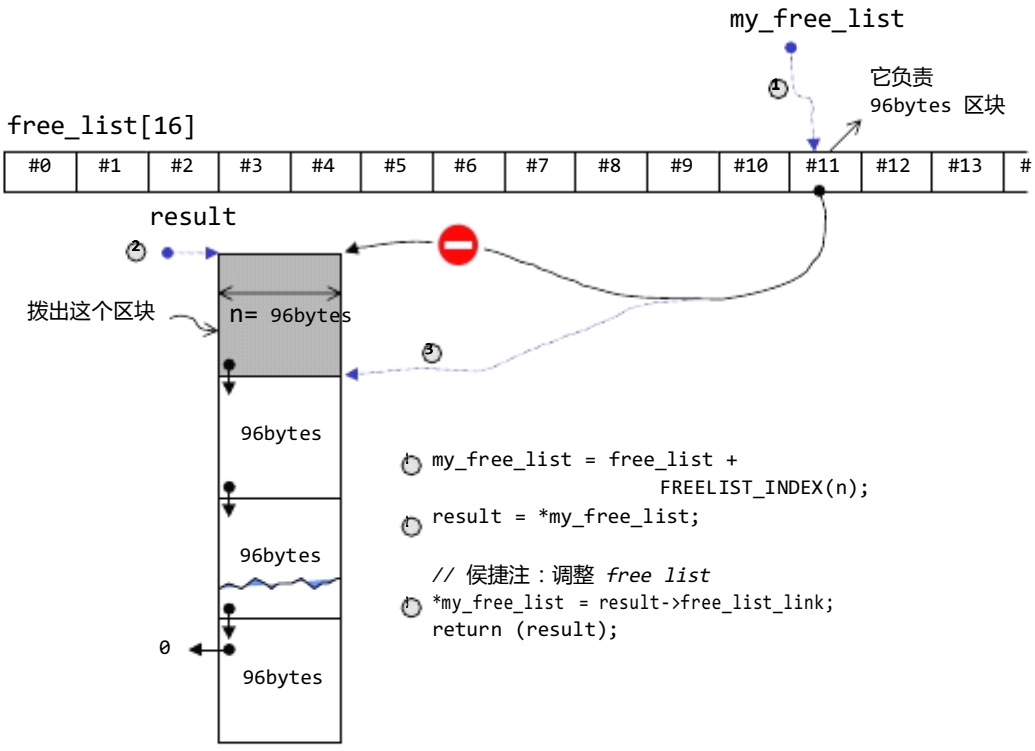


图 2-5 区块自 *free list* 拨出。阅读次序请循图中编号。

2.2.8 空间释还函式 deallocate()

身为一个配置器，__default_alloc_template 拥有配置器标准介面函式 deallocate()。此函式首先判断区块大小，大于 128 bytes 就呼叫第一级配置器，小于 128 bytes 就找出对应的 *free list*，将区块回收。

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

  // 大于 128 就呼叫第一级配置器
  if (n > (size_t) __MAX_BYTES) {
      malloc_alloc::deallocate(p, n);
      return;
  }
  // 寻找对应的 free list
  my_free_list = free_list + FREELIST_INDEX(n);
  // 调整 free list，回收区块
  q -> free_list_link = *my_free_list;
  *my_free_list = q;
}
```

区块回收纳入 *free list* 的动作，如图 2-6。

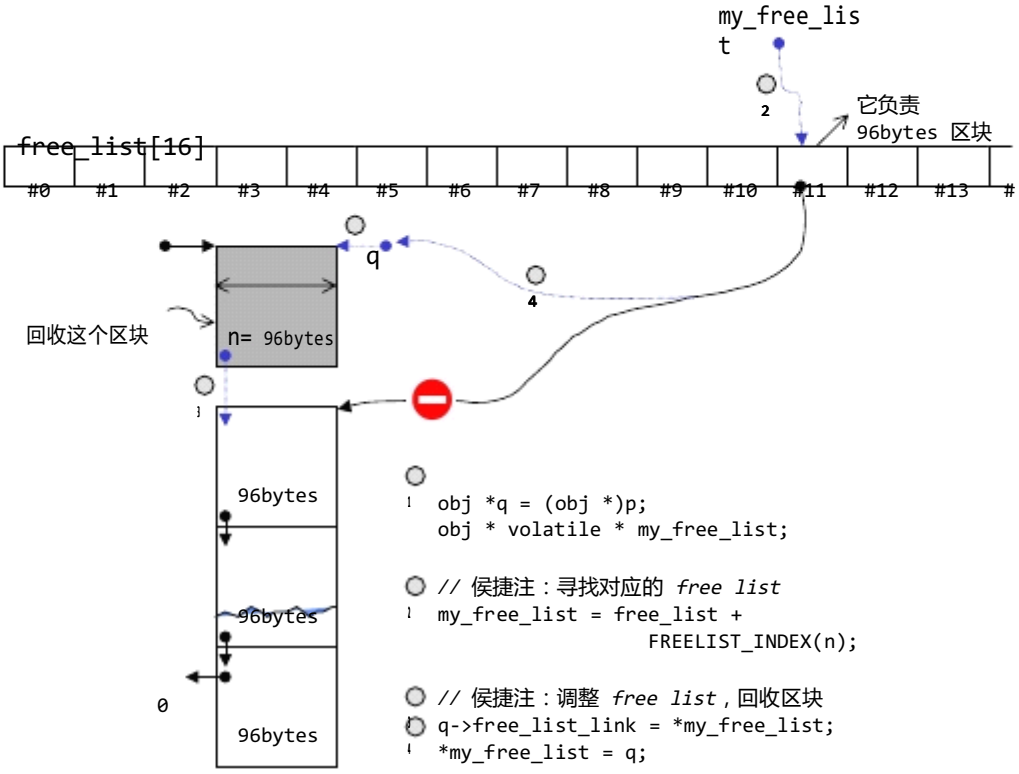


图 2-6 区块回收，纳入 *free list*。阅读次序请循图中编号。

2.2.9 重新充填 *free lists*

回头讨论先前说过的 allocate()。当它发现 *free list* 中没有可用区块了，就呼叫 refill() 准备为 *free list* 重新填充空间。新的空间将取自记忆池（经由 chunk_alloc() 完成）。预设取得 20 个新节点（新区块），但万一记忆池空间不足，获得的节点数（区块数）可能小于 20：

```
// 传回一个大小为 n 的对象，并且有时候会为适当的 free list 增加节点.
// 假设 n 已经适当上调至 8 的倍数。
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    int nobjs = 20;
    // 呼叫 chunk_alloc()，尝试取得 nobjs 个区块做为 free list 的新节点。
    // 注意参数 nobjs 是 pass by reference。
    char * chunk = chunk_alloc(n, nobjs); // 下节详述
```

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

    // 如果只获得一个区块，这个区块就拨给呼叫者用，free list 无新节点。
    if (1 == nobjs) return(chunk);
    // 否则准备调整 free list，纳入新节点。
    my_free_list = free_list + FREELIST_INDEX(n);

    // 以下在 chunk 空间内建立 free list
      result = (obj *)chunk;                // 这一块准备传回给客端
      // 以下导引 free list 指向新配置的空间（取自记忆池）
      *my_free_list = next_obj = (obj *)(chunk + n);
      // 以下将 free list 的各节点串接起来。
      for (i = 1; ; i++) { // 从 1 开始，因为第 0 个将传回给客端
        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 记忆池（memory pool）

从记忆池中取空间给 *free list* 使用，是 chunk_alloc() 的工作：

```
// 假设 size 已经适当上调至 8 的倍数。
// 注意参数 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; // 记忆池剩余空间

    if (bytes_left >= total_bytes) {
        // 记忆池剩余空间完全满足需求量。
        result = start_free;
        start_free += total_bytes;
```

```
        return(result);
    } else if (bytes_left >= size) {
        // 记忆池剩余空间不能完全满足需求量，但足够供应一个（含）以上的区块。
        nobjs = bytes_left/size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else {
        // 记忆池剩余空间连一个区块的大小都无法提供。
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        // 以下试着让记忆池中的残余零头还有利用价值。
        if (bytes_left > 0) {
            // 记忆池内还有一些零头，先配给适当的 free list。
            // 首先寻找适当的 free list。
            obj * volatile * my_free_list =
                        free_list + FREELIST_INDEX(bytes_left);
            // 调整 free list，将记忆池中的残余空间编入。
            ((obj *)start_free) -> free_list_link = *my_free_list;
            *my_free_list = (obj *)start_free;
        }

        // 配置 heap 空间，用来挹注记忆池。
        start_free = (char *)malloc(bytes_to_get);
        if (0 == start_free) {
            // heap 空间不足，malloc() 失败。
            int i;
            obj * volatile * my_free_list, *p;
            // 试着检视我们手上拥有的东西。这不会造成伤害。我们不打算尝试配置
            // 较小的区块，因为那在多行程（multi-process）机器上容易导致灾难
            // 以下搜寻适当的 free list，
            // 所谓适当是指「尚有未用区块，且区块够大」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 内尚有未用区块。
                    // 调整 free list 以释出未用区块
                    *my_free_list = p -> free_list_link;
                    start_free = (char *)p;
                    end_free = start_free + i;
                    // 递归呼叫自己，为了修正 nobjs。
                    return(chunk_alloc(size, nobjs));
                    // 注意，任何残余零头终将被编入适当的 free-list 中备用。
                }
            }
        end_free = 0; // 如果出现意外（山穷水尽，到处都没内存可用了）
            // 呼叫第一级配置器，看看 out-of-memory 机制能否尽点力
            start_free = (char *)malloc_alloc::allocate(bytes_to_get);
            // 这会导致掷出异常（exception），或内存不足的情况获得改善。
```

```
            }
            heap_size += bytes_to_get;
            end_free = start_free + bytes_to_get;
            // 递归呼叫自己，为了修正 nobjs。
            return(chunk_alloc(size, nobjs));
        }
    }
```

上述的 chunk_alloc() 函式以 end_free - start_free 来判断记忆池的水量。如果水量充足，就直接拨出 20 个区块传回给 *free list*。如果水量不足以提供 20 个区块，但还足够供应一个以上的区块，就拨出这不足 20 个区块的空间出去。这时候其 pass by reference 的 nobjs 参数将被修改为实际能够供应的区块数。如果记忆池连一个区块空间都无法供应，对客端显然无法交待，此时便需利用 malloc() 从 heap 中配置内存，为记忆池注入活水源头以应付需求。新水量的大小为需求量的两倍，再加上一个随着配置次数增加而愈来愈大的附加量。

举个例子，见图 2-7，假设程序一开始，客端就呼叫 chunk_alloc(32,20)，于是 malloc() 配置 40 个 32bytes 区块，其中第 1 个交出，另 19 个交给 free_list[3] 维护，余 20 个留给记忆池。接下来客端呼叫 chunk_alloc(64,20)，此时 free_list[7] 空空如也，必须向记忆池要求支持。记忆池只够供应 (32*20)/64=10 个 64bytes 区块，就把这 10 个区块传回，第 1 个交给客端，余 9 个由 free_list[7] 维护。此时记忆池全空。接下来再呼叫 chunk_alloc(96, 20)，此时free_list[11] 空空如也，必须向记忆池要求支持，而记忆池此时也是空的，于是以 malloc() 配置 40+n（附加量）个 96bytes 区块，其中第 1 个交出，另 19 个交给 free_list[11] 维护，余 20+n（附加量）个区块留给记忆池……。

万一山穷水尽，整个 system heap 空间都不够了（以至无法为记忆池注入活水源头），malloc() 行动失败，chunk_alloc() 就四处寻找有无「尚有未用区块，且区块够大」之 *free lists*。找到的话就挖一块交出，找不到的话就呼叫第一级配置器。第一级配置器其实也是使用 malloc() 来配置内存，但它有 out-of-memory 处理机制（类似 new-handler 机制），或许有机会释放其它的内存拿来此处使用。如果可以，就成功，否则发出 *bad_alloc* 异常。

以上便是整个第二级空间配置器的设计。

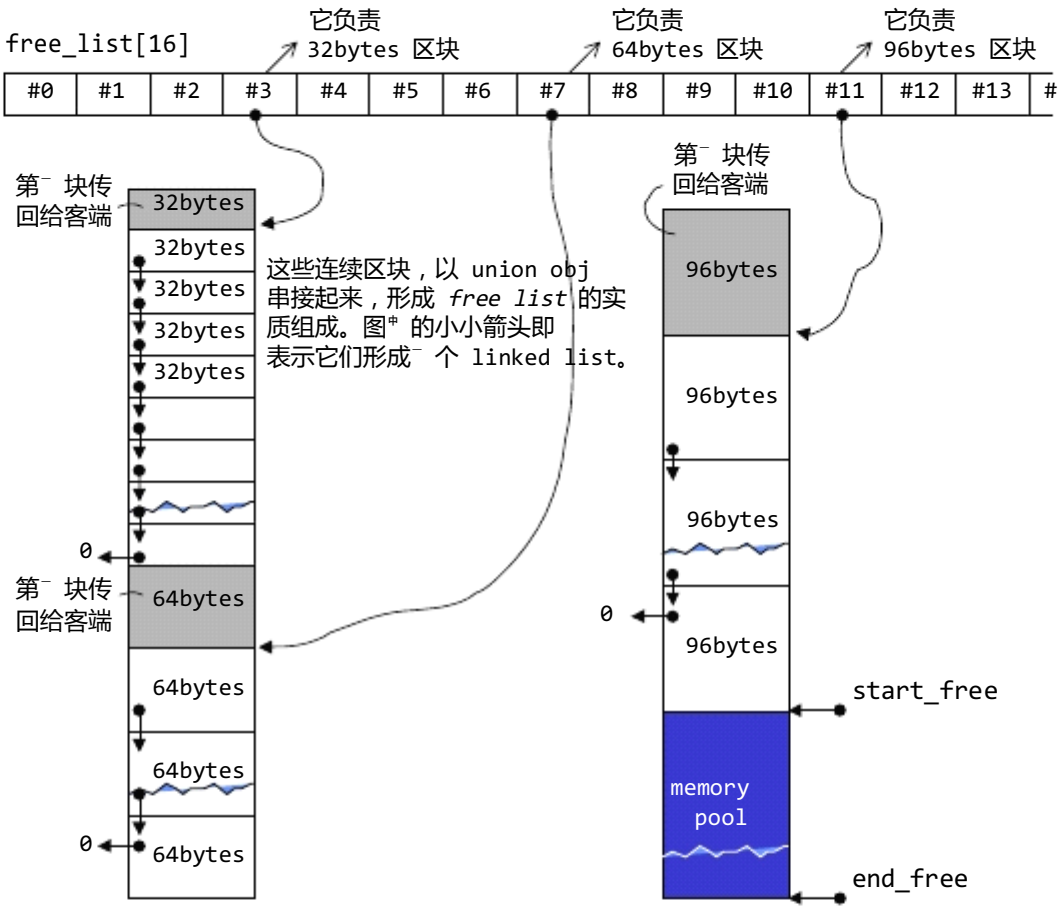


图 2-7 记忆池（memory pool）实际操练结果

回想一下 2.2.4 节最后提到的那个提供配置器标准接口的 simple_alloc：

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

SGI 容器通常以这种方式来使用配置器：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class vector {
public:
  typedef T value_type;
...
```

```
protected:
  // 专属之空间配置器，每次配置一个元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;
...
};
```

其中第二个 template 参数所接受的预设自变量 **alloc**，可以是第一级配置器，也可以是第二级配置器。不过，SGI STL 已经把它设为第二级配置器，见 2.2.4 节及图 2-2b。

2.3 内存基本处理工具

STL 定义有五个全域函式，作用于未初始化空间上。这样的功能对于容器的实作很有帮助，我们会在第四章容器实作码中，看到它们的吃重演出。前两个函式是 2.2.3 节说过，用于建构的 construct() 和用于解构的 destroy()，另三个函式是 uninitialized_copy(),uninitialized_fill(),uninitialized_fill_n()[10]，分别对应于高阶函式 copy()、fill()、fill_n() — 这些都是 STL 算法，将在第六章介绍。如果你要使用本节的三个低阶函式，应该含入 <memory>，不过 SGI 把它们实际定义于 <stl_uninitialized>。

2.3.1 uninitialized_copy

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

uninitialized_copy() 使我们能够将记忆体的配置与物件的建构行为分离开来。如果做为输出目的的 [result, result+(last-first)) 范围内的每一个迭代器都指向未初始化区域，则 uninitialized_copy() 会使用 copy constructor，为身为输入来源之 [first,last) 范围内的每一个对象产生一份复制品，放进输出范围中。换句话说，针对输入范围内的每一个迭代器 i，此函式会呼叫 construct(&*(result+(i-first)),*i)，产生 *i 的复制品，放置于输

[1] [Austern98] 10.4 节对于这三个低阶函式有详细的介绍。

出范围的相对位置上 。式中 的 construct() 已于 2.2.3 节讨论过。

如果你有需要实作一 个容器，uninitialized_copy() 这样的函式会为你带来很大的帮助，因为容器的全范围建构式（range constructor）通常以两个步骤完成：

配置内存区块，足以包含范围内的所有元素。
使用 uninitialized_copy()，在该内存区块上 建构元素。

C++ 标准规格书要求 uninitialized_copy() 具有 "*commit* or *rollback*" 语意，意思是要不就「建构出所有必要元素」，要不就（当有任何一 个 copy constructor 失败时）「不建构任何东西」。

2.3.2 uninitialized_fill

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

uninitialized_fill() 也能够使我们将记忆体配置与物件的建构行为分离开来。如果 [first,last) 范围内的每个迭代器都指向未初始化的内存，那么 uninitialized_fill() 会在该范围内产生 x（上 式第三 参数）的复制品。换句话说 uninitialized_fill() 会针对操作范围内的每个迭代器 i，呼叫 construct(&*i, x)，在 i 所指之处产生 x 的复制品。式中 的 construct() 已于 2.2.3 节讨论过。

和 uninitialized_copy() 一 样，uninitialized_fill() 必须具备 "*commit* or *rollback*" 语意，换句话说它要不就产生出所有必要元素，要不就不产生任何元素。如果有任何一 个 copy constructor 丢出异常（exception）uninitialized_fill() 必须能够将已产生之所有元素解构掉。

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() 能够使我们将内存配置与对象建构行为分离开来。
它会为指定范围内的所有元素设定相同的初值。

如果 [first, first+n) 范围内的每一个迭代器都指向未初始化的内存，那么 uninitialized_fill_n() 会呼叫 copy constructor，在该范围内产生 x（上式第三参数）的复制品。也就是说面对 [first,first+n) 范围内的每个迭代器 i，uninitialized_fill_n() 会呼叫 construct(&*i, x)，在对应位置处产生 x 的复制品。式中的 construct() 已于 2.2.3 节讨论过。

uninitialized_fill_n() 也具有 "*commit* or *rollback*" 语意：要不就产生所有必要的元素，否则就不产生任何元素。如果任何一个 copy constructor 丢出异常（exception），uninitialized_fill_n() 必须解构已产生的所有元素。

以下分别介绍这三个函式的实作法。其中所呈现的 iterators（迭代器）、value_type()、__type_traits、__true_type、__false_type、is_POD_type 等实作技术，都将于第三章介绍。

(1) uninitialized_fill_n

首先是 uninitialized_fill_n() 的源码。本函式接受三个参数：

1. 迭代器 first 指向欲初始化空间的起始处
2. n 表示欲初始化空间的大小
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.
}
```

这个函式的进行逻辑是，首先萃取出迭代器 first 的 *value type*（详见第三章），然后判断该型别是否为 POD 型别：

```
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，详见 3.7 节
  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**，也就是纯量型别（scalar types）或传统的 C struct 型别。POD 型别必然拥有 *trivial* ctor/dtor/copy/assignment 函式，因此，我们可以对 POD 型别采取最有效率的初值填写手法，而对 non-POD 型别采取最保险安全的作法：

```
// 如果 copy construction 等同于 assignment，而且
// destructor 是 trivial，以下 就有效。
// 如果是 POD 型别，执行流程就会转进到以下 函式。这是藉由 function template
// 的自变量推导机制而得。
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);                  // 交由高阶函式执行。见 6.4.2 节。
}

// 如果不是 POD 型别，执行流程就会转进到以下 函式。这是藉由 function template
// 的自变量推导机制而得。
template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {
  ForwardIterator cur = first;
    // 为求阅读顺畅，以下 将原本该有的异常处理（exception handling）省略。
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x);                   // 见 2.2.3 节
    return cur;
}
```

(2) uninitialized_copy

下 面列出 uninitialized_copy() 的源码。本函式接受三 个参数：

迭代器 first 指向输入端的起始位置

迭代器 last 指向输入端的结束位置（前闭后开区间）

迭代器 result 指向输出端（欲初始化空间）的起始处

```
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.
}
```

这个函式的进行逻辑是，首先萃取出迭代器 result 的 *value type*（详见第三章），然后判断该型别是否为 POD 型别：

```
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，企图利用 is_POD() 所获得的结果，让编译器做自变量推导。
}
```

POD 意指 Plain Old Data，也就是纯量型别（scalar types）或传统的 C struct 型别。POD 型别必然拥有 *trivial* ctor/dtor/copy/assignment 函式，因此，我们可以对 POD 型别采取最有效率的复制手法，而对 non-POD 型别采取最保险安全的作法：

```
// 如果 copy construction 等同于 assignment，而且
// destructor 是 trivial，以下就有效。
// 如果是 POD 型别，执行流程就会转进到以下函式。这是藉由 function template
// 的自变量推导机制而得。
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 算法 copy()
}

// 如果是 non-POD 型别，执行流程就会转进到以下函式。这是藉由function template
// 的自变量推导机制而得。
template <class InputIterator, class ForwardIterator>
ForwardIterator
__uninitialized_copy_aux(InputIterator first, InputIterator last,
                         ForwardIterator result,
                         __false_type) {
  ForwardIterator cur = result;
    // 为求阅读顺畅，以下将原本该有的异常处理（exception handling）省略。
    for ( ; first != last; ++first, ++cur)
      construct(&*cur, *first); // 必须一个一个元素建构，无法批量进行
    return cur;
  }
}
```

针对 char* 和 wchar_t* 两种型别，可以最具效率的作法 memmove（直接搬移内存内容）来执行复制行为。因此 SGI 得以为这两种型别设计一份特化版本。

```
// 以下是针对 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);
}

// 以下是针对 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() 的源码。本函式接受三个参数：

迭代器 first 指向输出端（欲初始化空间）的起始处

迭代器 last 指向输出端（欲初始化空间）的结束处（前闭后开区间）

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));
}
```

这个函式的进行逻辑是，首先萃取出迭代器 first 的 *value type*（详见第三章），然后判断该型别是否为 POD 型别：

```
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，也就是纯量型别（scalar types）或传统的 C struct 型别。POD 型别必然拥有 *trivial* ctor/dtor/copy/assignment 函式，因此，我们可以对 POD 型别采取最有效率的初值填写手法，而对 non-POD 型别采取最保险安全的

作法：

```
// 如果 copy construction 等同于 assignment，而且
// destructor 是 trivial，以下就有效。
// 如果是 POD 型别，执行流程就会转进到以下函式。这是藉由 function template
// 的自变量推导机制而得。
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 算法 fill()
}

// 如果是 non-POD 型别，执行流程就会转进到以下函式。这是藉由function template
// 的自变量推导机制而得。
template <class ForwardIterator, class T>
void
__uninitialized_fill_aux(ForwardIterator first, ForwardIterator last,
                         const T& x, __false_type)
{
  ForwardIterator cur = first;
    // 为求阅读顺畅，以下将原本该有的异常处理（exception handling）省略。
    for ( ; cur != last; ++cur)
      construct(&*cur, x); // 必须一个一个元素建构，无法批量进行
  }
}
```

图 2-8 将本节三个函式对效率的特殊考虑，以图形显示。

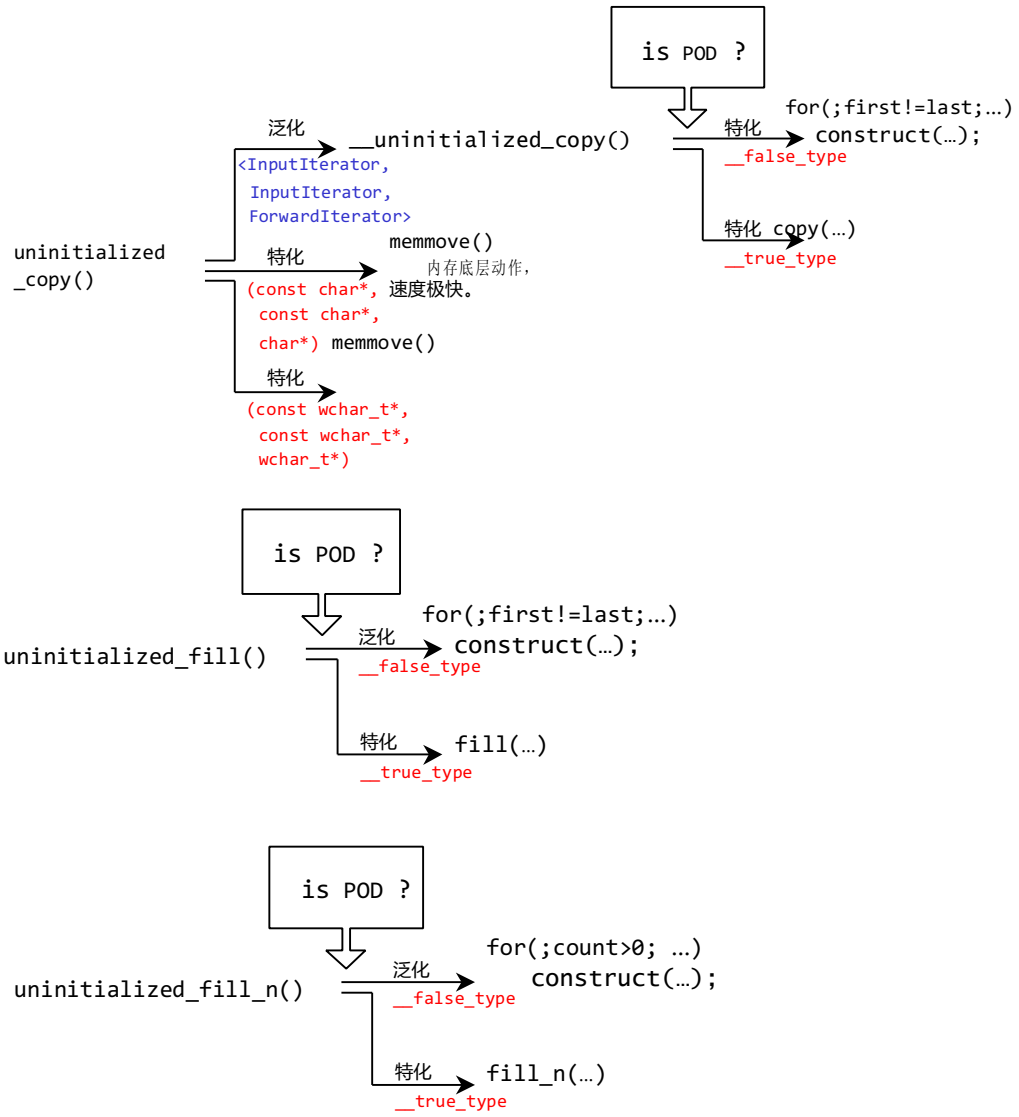


图 2-8 三个内存基本函式的泛型版本与特化版本。

3

迭代器（iterators）概念与 traits 编程技法

迭代器（iterators）是一种抽象的设计概念，现实程序语言并没有直接对映于这个概念的实物。《*Design Patterns*》一书提供有 23 个设计样式（design patterns）的完整描述，其 *iterator* 样式定义如下：提供一种方法，俾得依序巡访某个聚合物（容器）所含的各个元素，而又无需曝露该聚合物的内部表述方式。

3.1 迭代器设计思维 — STL 关键所在

不论是泛型思维或 STL 的实际运用，迭代器（iterators）都扮演重要角色。STL 的心思想在于，将数据容器（containers）和算法（algorithms）分开，彼此独立设计，最后再以一帖胶着剂将它们撮合在一起。容器和算法的泛型化，从技术角度来看并不困难，C++ 的 class templates 和 function templates 可分别达成目标。如何设计出两者之间的良好胶着剂，才是大难题。

以下是容器、算法、迭代器（iterator，扮演黏胶角色）的合作展示。以算法 find() 为例，它接受两个迭代器和一个「搜寻标的」：

```
// 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;
}
```

The Annotated STL Sources

只要给予不同的迭代器，find() 便能够对不同的容器做搜寻动作：

```
// 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]，未符合标准

   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;
   // 执行结果：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;
   // 执行结果：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;
   // 执行结果：8 not found.
}
```

从这个例子看来，迭代器似乎依附在容器之下。是吗？有没有独立而泛用的迭代器？我们又该如何自行设计特殊的迭代器？

3.2 迭代器（iterator）是一种 smart pointer

迭代器是一种行为类似指针的对象，而指针的各种行为中最常见也最重要的便是

内容提领（*dereference*）和成员取用（member access），因此迭代器最重要的编程工作就是对 operator* 和 operator-> 进行多载化（*overloading*）工程。关于这一点，C++ 标准链接库有一个 auto_ptr 可供我们参考。任何一本详尽的 C++ 语法书籍都应该谈到 auto_ptr（如果没有，扔了它），这是一个用来包装原生指标（native pointer）的对象，声名狼藉的内存漏洞（memory leak）问题可藉此获得解决。auto_ptr 用法如下，和原生指标一模一样：

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

   cout << *ps << endl;                    // 输出：jjhou
   cout << ps->size() << endl;             // 输出：5
   // 离开前不需 delete，auto_ptr 会自动释放内存
}
```

函式第一行的意思是，以算式 new 动态配置一个初值为 "jjhou" 的 string 物件，并将所得结果（一个原生指针）做为 auto_ptr<string> 对象的初值。注意，auto_ptr 角括号内放的是「原生指针所指对象」的型别，而不是原生指标的型别。

auto_ptr 的源码在表头档 <memory> 中，根据它，我模拟了一份阳春版，可具体说明 auto_ptr 的行为与能力：

```
// 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;
```

```
};
```

其中关键词 explicit 和 member template 等编程技法，并不是这里的讲述重点，相关语法和语意请参阅 [Lippman98] 或 [Stroustrup97]。

有了模仿对象，现在我们来为 list（串行）设计一个迭代器[1]。假设 list 及其节点的结构如下：

```
// 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; // 单向串行（single linked list）
};
```

如何将这个 List 套用到先前所说的 find() 呢？我们需要为它设计一个行为类似指标的外衣，也就是一个迭代器。当我们提领（*dereference*）此一迭代器，传回的应该是个 ListItem 对象；当我们累加该迭代器，它应该指向下一个 ListItem 物件。为了让此迭代器适用于任何型态的节点，而不只限于 ListItem，我们可以将它设计为一个 class template：

[Lippman98] 5.11 节有一个非泛型的 list 实例可以参考。《泛型思维》书中有一份泛型版本的 list 的完整设计与说明。

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

template <class Item> // Item 可以是单向串行节点或双向串行节点。
struct ListIter              // 此处这个迭代器特定只为串行服务，因为其
{                            // 独特的 operator++ 之故。
  Item* ptr; // 保持与容器之间的一个联系 ( keep a reference to Container )

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

  // 不必实作 copy ctor，因为编译器提供的预设行为已足够。
  // 不必实作 operator=，因为编译器提供的预设行为已足够。

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

  // 以下两个 operator++ 遵循标准作法，参见 [Meyers96] 条款 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; }
};
```

现在我们可以这样子将 List 和 find() 藉由 ListIter 黏合起来：

```
// 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;
    // 执行结果：found. 3

    iter = find(begin, end, 7);
    if (iter == end)
        cout << "not found" << endl;
    else
        cout << "found. " << iter->value() << endl;
    // 执行结果：not found
}
```

注意，由于 find() 函式内以 *iter != value 来检查元素值是否吻合，而本例之中 value 的型别是 int，iter 的型别是 ListItem<int>，两者之间并无可供使用的 operator!=，所以我必须另外写一个全域的 operator!= 多载函式，并以 int 和 ListItem<int> 做为它的两个参数型别：

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

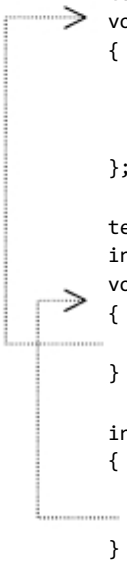
从以上实作可以看出，为了完成一个针对 List 而设计的迭代器，我们无可避免地曝露了太多 List 实作细节：在 main() 之中为了制作 begin 和 end 两个迭代器，我们曝露了 ListItem；在 ListIter class 之中为了达成 operator++ 的目的，我们曝露了 ListItem 的操作函式 next()。如果不是为了迭代器，ListItem 原本应该完全隐藏起来不曝光的。换句话说，要设计出 ListIter，首先必须对 List 的实作细节有非常丰富的了解。既然这无可避免，干脆就把迭代器的开发工作交给 List 的设计者好了，如此一来所有实作细节反而得以封装起来不被使用者看到。这正是为什么每一种 STL 容器都提供有专属迭代器的缘故。

3.3 迭代器相应型别（associated types）

上述的 ListIter 提供了一个迭代器雏形。如果将思想拉得更高远一些，我们便会发现，算法之中运用迭代器时，很可能会用到其相应型别（associated type）。什么是相应型别？迭代器所指之物的型别便是其一。假设算法中有必要宣告一个变数，以「迭代器所指物件的型别」为型别，如何是好？毕竟 C++ 只支援

sizeof()，并未支持 typeof()！即便动用 RTTI 性质[中] 的 typeid()，获得的也只是型别名称，不能拿来做变量宣告之用。

解决办法是有：利用 function template 的自变量推导（argument deducation）机制。例如：



```
template <class I, class T>
void func_impl(I iter, T t)
{
  T tmp; // 这里解决了问题。T 就是迭代器所指之物的型别，本例为 int

  // ... 这里做原本 func() 应该做的全部工作
};

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);
}
```

我们以 func() 为对外界面，却把实际动作全部置于 func_impl() 之[中]。由于 func_impl() 是[一]个 function template，[一]旦被呼叫，编译器会自动进行 template 自变量推导。于是导出型别 T，顺利解决了问题。

迭代器相应型别（associated types）不只是「迭代器所指对象的型别」[一]种而已。根据经验，最常用的相应型别有五种，然而并非任何情况[下]任何[一]种都可利用[上]述的 template 自变量推导机制来取得。我们需要更全面的解法。

3.4 Traits 编程技法 — STL 源码门钥法

迭代器所指物件的型别，称为该迭代器的 *value type*。[上]述的自变量型别推导技巧虽然可用于 *value type*，却非全面可用：万[一] *value type* 必须用于函式的传回值，就束手无策了，毕竟函式的「template 自变量推导机制」推而导之的只是自变量，无

法推导函式的回返值型别。

我们需要其它方法。宣告巢状型别似乎是个好主意，像这样：

```
template <class T>
struct MyIter
  typedef T value_type;          // 巢状型别宣告（nested type）
  T* ptr;
  MyIter(T* p=0) : ptr(p) { }
  T& operator*() const { return *ptr; }
  // ...
};

template <class I>
typename I::value_type           // 这一整行是 func 的回返值型别
func(I ite)
{ return *ite; }

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

注意，func() 的回返型别必须加上关键词 typename，因为 T 是一个 template 参数，在它被编译器具现化之前，编译器对 T 一无所悉，换句话说编译器此时并不知道 MyIter<T>::value_type 代表的是一个型别或是一个 member function 或是一个 data member。关键词 typename 的用意在告诉编译器说这是一个型别，如此才能顺利通过编译。

看起来不错。但是有个隐晦的陷阱：并不是所有迭代器都是 class type。原生指标就不是！如果不是 class type，就无法为它定义巢状型别。但 STL（以及整个泛型思维）绝对必须接受原生指标做为一种迭代器，所以上面这样还不够。有没有办法可以让上述的一般化概念针对特定情况（例如针对原生指标）做特殊化处理呢？

是的，template partial specialization 可以做到。

Partial Specialization（偏特化）的意义

任何完整的 C++ 语法书籍都应该对 template partial specialization 有所说明（如果没有，扔了它 ）。大致的意义是：如果 class template 拥有一个以上的 template 参数，我们可以针对其中某个（或数个，但非全部）template 参数进行特化工作。

换句话说我们可以在泛化设计中提供一个特化版本（也就是将泛化版本中的某些 template 参数赋予明确的指定）。

假设有一个 class template 如下：

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

partial specialization 的字面意义容易误导我们以为，所谓「偏特化版」一定是对 template 参数 U 或 V 或 T（或某种组合）指定某个自变量值。事实不然，**[Austern99]** 对于 partial specialization 的意义说得十分得体：「所谓 partial specialization 的意思是提供另一份 template 定义式，而其本身仍为 templatized」。《泛型技术》一书对 partial specialization 的定义是：「针对（任何）template 参数更进一步的条件限制，所设计出来的一个特化版本」。由此，面对以下这么一个 class template：

```
template<typename T>
class C { ... };              // 这个泛化版本允许（接受）T 为任何型别
```

我们便很容易接受它有一个型式如下的 partial specialization：

```
template<typename T>
class C<T*> { ... }; // 这个特化版本仅适用于「T 为原生指标」的情况
                     // 「T 为原生指标」便是「T 为任何型别」的一个更进一步的条件限制
```

有了这项利器，我们便可以解决前述「巢状型别」未能解决的问题。先前的问题是，原生指标并非 class，因此无法为它们定义巢状型别。现在，我们可以针对「迭代器之 template 自变量为指标」者，设计特化版的迭代器。

提高警觉，我们进入关键地带了。下面这个 class template 专门用来「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 意为「特性」
  typedef typename I::value_type                  value_type;
};
```

这个所谓的 **traits**，其意义是，如果 I 定义有自己的 *value type*，那么透过这个 **traits** 的作用，萃取出来的 value_type 就是 I::value_type。换句话说如果 I 定义有自己的 *value type*，先前那个 func() 可以改写成这样：

```
template <class I>
typename iterator_traits<I>::value_type // 这一整行是函式回返型别
func(I ite)
{ return *ite; }
```

但这除了多一层间接性，又带来什么好处？好处是 **traits** 可以拥有特化版本。现在，我们令 iterator_traites 拥有一个 partial specializations 如下：

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

于是，原生指标 int* 虽然不是一种 class type，亦可透过 **traits** 取其 *value type*。这就解决了先前的问题。

但是请注意，针对「指向常数对象的指针（pointer-to-const）」，下面这个式子得到什么结果：

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

获得的是 const int 而非 int。这是我们期望的吗？我们希望利用这种机制来宣告一个暂时变量，使其型别与迭代器的 *value type* 相同，而现在，宣告一个无法赋值（因 const 之故）的暂时变数，没什么用！因此，如果迭代器是个 pointer-to-const，我们应该设法令其 *value type* 为一个 non-const 型别。没问题，只要另外设计一个特化版本，就能解决这个问题：

```
template <class T>
struct iterator_traits<const T*> { // 偏特化版 — 当迭代器是个 pointer-to-const
  typedef T value_type;                    // 萃取出来的型别应该是 T      而非 const T
};
```

现在，不论面对的是迭代器 MyIter，或是原生指标 int* 或 const int*，都可以透过 **traits** 取出正确的（我们所期望的）*value type*。

图 3-1 说明 **traits** 所扮演的「特性萃取机」角色，萃取各个迭代器的特性。这里所谓的迭代器特性，指的是迭代器的相应型别（associated types）。当然，若要这个「特性萃取机」**traits** 能够有效运作，每一个迭代器必须遵循约定，自行以巢状型别定义（nested typedef）的方式定义出相应型别（associated types）。这种一个约定，谁不遵守这个约定，谁就不能相容于 STL 这个大家庭。

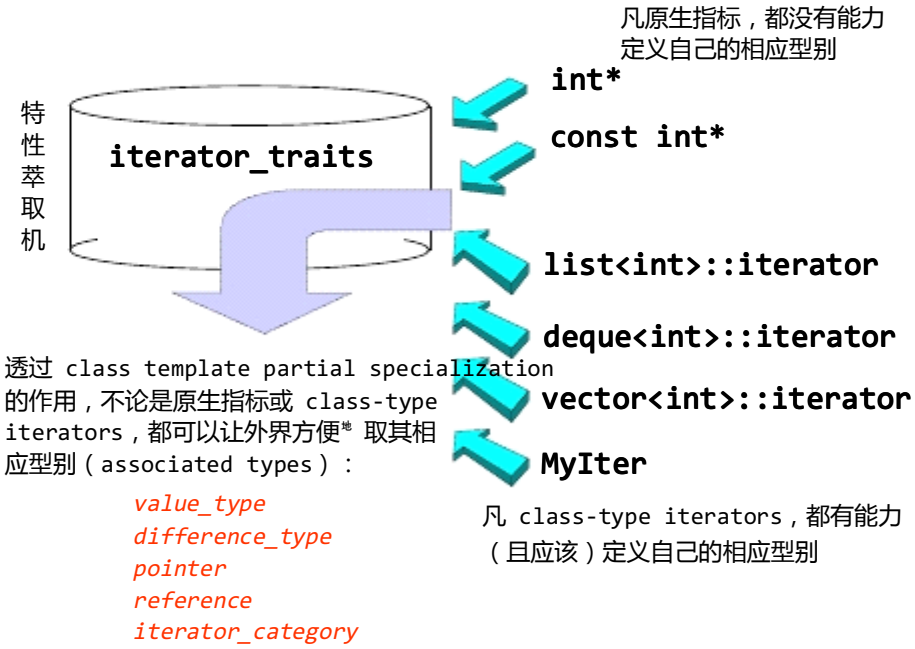


图 3-1 **traits** 就像一台「特性萃取机」，榨取各个迭代器的特性（相应型别）。

根据经验，最常用到的迭代器相应型别有五种：*value type*，*difference type*，*pointer*，*reference*，*iterator catagoly*。如果你希望你所开发的容器能与 STL 水乳交融，一定要为你的容器的迭代器定义这五种相应型别。「特性萃取机」**traits** 会很忠实[1]将原汁原味榨取出来：

```
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 必须针对传入之型别为 pointer 及 pointer-to-const 者，设计特化版本，稍后数节为你展示如何进行。

3.4.1 迭代器相应型别之一：*value type*

所谓 *value type*，是指迭代器所指对象的型别。任何一个打算与 STL 算法有完美搭配的 class，都应该定义自己的 *value type* 巢状型别，作法就像上一节所述。

3.4.2 迭代器相应型别之二：*difference type*

difference type 用来表示两个迭代器之间的距离，也因此，它可以用来表示一个容器的最大容量，因为对于连续空间的容器而言，头尾之间的距离就是其最大容量。

如果一个泛型算法提供计数功能，例如 STL 的 count()，其传回值就必须使用迭代器的 *difference type*：

```
template <class I, class T>
typename iterator_traits<I>::difference_type // 这一整行是函式回返型别
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;
}
```

针对相应型别 *difference type*，**traits** 的两个（针对原生指标而写的）特化版本如下，以 C++ 内建的 ptrdiff_t（定义于 <cstddef> 表头档）做为原生指标的 *difference type*：

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

// 针对原生指标而设计的「偏特化（partial specialization）」版
template <class T>
struct iterator_traits<T*> {
  ...
  typedef ptrdiff_t difference_type;
};

// 针对原生的 pointer-to-const 而设计的「偏特化（partial specialization）」版
template <class T>
struct iterator_traits<const T*> {
  ...
```

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

现在，任何时候当我们需要任何迭代器 I 的 *difference type* ，可以这么写：

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

3.4.3 迭代器相应型别之三：*reference type*

从「迭代器所指之物的内容是否允许改变」的角度观之，迭代器分为两种：不允许改变「所指对象之内容」者，称为 constant iterators，例如 const int* pic；允许改变「所指对象之内容」者，称为 mutable iterators，例如 int* pi。当我们对一个 mutable iterators 做提领动作时，获得的不应该是个右值（rvalue），应该是个左值（lvalue），因为右值不允许赋值动作（assignment），左值才允许：

```
int* pi = new int(5);
const int* pci = new int(9);
*pi = 7;       // 对 mutable iterator 做提领动作时，获得的应该是个左值，允许赋值。
*pci = 1; // 这个动作不允许，因为 pci 是个 constant iterator，
            // 提领 pci 所得结果，是个右值，不允许被赋值。
```

在 C++ 中，函式如果要传回左值，都是以 by reference 的方式进行，所以当 p 是个 mutable iterators 时，如果其 *value type* 是 T，那么 *p 的型别不应该是 T，应该是 T&。将此道理扩充，如果 p 是一个 constant iterators，其 *value type* 是 T，那么 *p 的型别不应该是 const T，而应该是 const T&。这里所讨论的 *p 的型别，即所谓的 *reference type*。实作细节将在下一小节一并展示。

3.4.4 迭代器相应型别之四：*pointer type*

pointers 和 references 在 C++ 中有非常密切的关连。如果「传回一个左值，令它代表 p 所指之物」是可能的，那么「传回一个左值，令它代表 p 所指之物的位址」也一定可以。也就是说我们能够传回一个 pointer，指向迭代器所指之物。

这些相应型别已在先前的 ListIter class 中出现过：

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

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

现在我们把 *reference type* 和 *pointer type* 这两个相应型别加入 **traits** 内：

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

// 针对原生指标而设计的「偏特化版（partial specialization）」
template <class T>
struct iterator_traits<T*> {
  ...
  typedef T* pointer;
  typedef T& reference;
};

// 针对原生的 pointer-to-const 而设计的「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 迭代器相应型别之五：*iterator_category*

最后一个（第五个）迭代器相应型别会引发较大规模的写码工程。在那之前，我必须先讨论迭代器的分类。

根据移动特性与施行动作，迭代器被分为五类：

Input Iterator：这种迭代器所指对象，不允许外界改变。只读（read only）。

Output Iterator：唯写（write only）。

Forward Iterator：允许「写入型」算法（例如 replace()）在此种迭代器所形成的区间上做读写动作。

Bidirectional Iterator：可双向移动。某些算法需要逆向走访某个迭代器区间（例如逆向拷贝某范围内的元素），就可以使用 *Bidirectional Iterators*。

Random Access Iterator：前四种迭代器都只供应一部份指标算术能力（前三种支持 operator++，第四种再加上 operator--），第五种则涵盖所有指标算术能力，包括 p+n, p-n, p[n], p1-p2, p1<p2。

这些迭代器的分类与从属关系，可以图 3-2 表示。直线与箭头代表的并非 C++ 的继承关系，而是所谓 concept（概念）与 refinement（强化）的关系[2]。

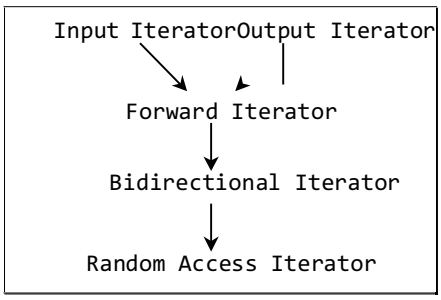


图 3-2 迭代器的分类与从属关系

设计算法时，如果可能，我们尽量针对图 3-2 中的某种迭代器提供一个明确定义，并针对更强化的某种迭代器提供另一种定义，这样才能在不同情况下提供最大效率。研究 STL 的过程中，每一分每一秒我们都要念兹在兹，效率是个重要课题。假设有个算法可接受 *Forward Iterator*，你以 *Random Access Iterator* 喂给它，它当然也会接受，因为一个 *Random Access Iterator* 必然是一个 *Forward Iterator*（见图 3-2）。但是可用并不代表最佳！

以 advanced() 为 例

拿 advance() 来说（这是许多算法内部常用的一个函式），此函式有两个参数，迭代器 p 和数值 n；函式内部将 p 累进 n 次（前进 n 距离）。下面有三份定义，一份针对 *Input Iterator*，一份针对 *Bidirectional Iterator*，另一份针对 *Random Access Iterator*。倒是没有针对 *ForwardIterator* 而设计的版本，因为那和针对 *InputIterator* 而设计的版本完全一致。

```
template <class InputIterator, class Distance>
void advance_II(InputIterator& i, Distance n)
{
  // 单向，逐一前进
  while (n--) ++i;              // 或写 for ( ; n > 0; --n, ++i );
```

---

[2] concept（概念）与 refinement（强化），是架构 STL 的重要观念，详见 [Austern98]。

```
}

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

template <class RandomAccessIterator, class Distance>
void advance_RAI(RandomAccessIterator& i, Distance n)
{
  // 双向，跳跃前进
  i += n;
}
```

现在，当程序呼叫 advance()，应该选用（呼叫）哪一份函式定义呢？如果选择 advance_II()，对 *Random Access Iterator* 而言极度缺乏效率，原本 *O(1)* 的操作竟成为 *O(N)*。如果选择 advance_RAI()，则它无法接受 *Input Iterator*。我们需要将三者合一，下面是一种作法：

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

但是像这样在执行时期才决定使用哪一个版本，会影响程序效率。最好能够在编译期就选择正确的版本。多载化函式机制可以达成这个目标。

前述三个 advance_xx() 都有两个函式参数，型别都未定（因为都是 template 参数）。为了令其同名，形成多载化函式，我们必须加上一个型别已确定的函式参数，使函式多载化机制得以有效运作起来。

设计考虑如下：如果 **traits** 有能力萃取出迭代器的种类，我们便可利用这个「迭代器类型」相应型别做为 advanced() 的第三参数。这个相应型别一定必须是个

class type，不能只是数值号码类的东西，因为编译器需仰赖它（一个型别）来进行多载化决议程序（overloaded resolution）。下面定义五个 classes，代表五种迭代器类型：

```
// 五个 做为标记用的型别（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 { };
```

这些 classes 只做为标记用，所以不需要任何成员。至于为什么运用继承机制，稍后再解释。现在重新设计 __advance()（由于只在内部使用，所以函式名称加上特定的前导符），并加上第三参数，使它们形成多载化：

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

// 这是一个单纯的转呼叫函式（trivial forwarding function）。稍后讨论如何免除之。
template <class ForwardIterator, class Distance>
inline void __advance(ForwardIterator& i, Distance n,
                      forward_iterator_tag)
{
  // 单纯地进行转呼叫（forwarding）
  advance(i, n, input_iterator_tag());
}

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

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

```
{
  // 双向，跳跃前进
  i += n;
}
```

注意上述语法，每个 __advance() 的最后一个参数都只宣告型别，并未指定参数名称，因为它纯粹只是用来启动多载化机制，函式之中根本不使用该参数。如果硬要加上参数名称也可以，画蛇添足罢了。

行进至此，还需要一个对外开放的上层控制接口，呼叫上述各个多载化的 __advance()。此一上层介面只需两个参数，当它准备将工作转给上述的 __advance() 时，才自行加上第三自变量：迭代器类型。因此，这个上层函式必须有能力从它所获得的迭代器中推导出其类型 — 这份工作自然是交给 **traits** 机制：

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

[3]

注意上述语法，iterator_traits<Iterator>::iterator_category() 将产生一个暂时对象（道理就像 int() 会产生一个 int 暂时对象一样），其型别应该隶属前述五个迭代器类型之一。然后，根据这个型别，编译器才决定呼叫哪一个 __advance() 多载函式。

关于此行，SGI STL <stl_iterator.h> 的源码是：

```
  __advance(i, n, iterator_category(i));
```

并另定义函式 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();
}
```

综合整理后原式即为：

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

因此，为了满足上 述行为，**traits** 必须再增加一 个相应型别：

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

// 针对原生指标而设计的「偏特化版（partial specialization）」
template <class T>
struct iterator_traits<T*> {
  ...
  // 注意，原生指标是一 种 Random Access Iterator
  typedef random_access_iterator_tag                       iterator_category;
};

// 针对原生的 pointer-to-const 而设计的「偏特化版（partial specialization）」
template <class T>
struct iterator_traits<const T*>
  ...
  // 注意，原生的 pointer-to-const 是一 种 Random Access Iterator
  typedef random_access_iterator_tag                       iterator_category;
};
```

任何一 个迭代器，其类型永远应该落在「该迭代器所隶属之各种类型* ，最强化的那个」。例如 int* 既是 *Random Access Iterator* 又是 *Bidirectional Iterator*，同时也是 *Forward Iterator*，而且也是 *Input Iterator*，那么，其类型应该归属为 *random_access_iterator_tag*。

你是否注意到 advance() 的 template 参数名称取得好像不怎么理想：

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

按说 advanced() 既然可以接受各种类型的迭代器，就不应将其型别参数命名为 InputIterator。这其实是 STL 算法的一 个命名规则：以算法所能接受之最低阶迭代器类型，来为其迭代器型别参数命名。

消除「单纯转呼叫函式」

以 class 来定义迭代器的各种分类标签，不唯可以促成多载化机制的成功运作（使编译器得以正确执行多载化决议程序，overloaded resolution），另一 个好处是，

透过继承，我们可以不必再写「单纯只做转呼叫」的函式（例如前述的 advance() *ForwardIterator* 版）。为什么能够如此？考虑下面这个小例子，从其输出结果可以看出端倪：

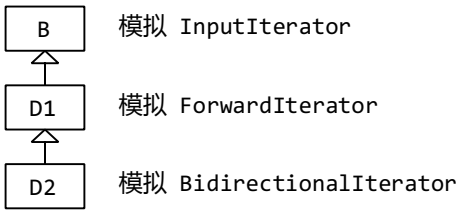


图 3-3 类别继承关系

```
// file: 3tag-test.cpp
// 模拟测试 tag types 继承关系所带来的影响。
#include <iostream>
using namespace std;

struct B { };                          // B 可比拟为 InputIterator
struct D1 : public B { };              // D1 可比拟为 ForwardIterator
struct D2 : public D1 { };             // D2 可比拟为 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());          // 参数与自变量完全吻合。输出：  "B version"
  func(p, D1()); // 参数与自变量未能完全吻合；因继承关系而自动转呼叫。
                 // 输出:"B version"
  func(p, D2()); // 参数与自变量完全吻合。输出：  "D2 version"
}
```

以 distance() 为 例

关于「迭代器类型标签」的应用，以下再举一例。distance() 也是常用的一个

迭代器操作函式，用来计算两个迭代器之间的距离。针对不同的迭代器类型，它可以有不同的计算方式，带来不同的效率。整个设计模式和前述的 advance() 如出一辙：

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
  iterator_traits<InputIterator>::difference_type n = 0;
  // 逐一累计距离
  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) {
  // 直接计算差距
  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() 可接受任何类型的迭代器；其 template 型别参数之所以命名为 InputIterator，是为了遵循 STL 算法的命名规则：以算法所能接受之最初级类型来为其迭代器型别参数命名。此外也请注意，由于迭代器类型之间存在着继承关系，「转呼叫（*forwarding*）」的行为模式因此自然存在 — 这一点我已在前一节讨论过。换句话说，当客端呼叫 distance() 并使用 *Output Iterator*s 或 *Forward Iterator*s 或 *Bidirectional Iterator*s，统统都会转呼叫 *Input Iterator* 版的那个 __distance() 函式。

3.5 std::iterator 的保证

为了符合规范，任何迭代器都应该提供五个巢状相应型别，以利 **traits** 萃取，否

则便是自外于整个 STL 架构，可能无法与其它 STL 组件顺利搭配。然而写码难免挂一漏万，谁也不能保证不会有粗心大意的时候。如果能够将事情简化，就好多了。STL 提供了一个 iterators class 如下，如果每个新设计的迭代器都继承自它，就保证符合 STL 所需之规范：

```
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 不含任何成员，纯粹只是型别定义，所以继承它并不会招致任何额外负担。由于后三个参数皆有默认值，新的迭代器只需提供前两个参数即可。先前 3.2 节土法炼钢的 ListIter，如果改用正式规格，应该这么写：

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

总结

设计适当的相应型别（associated types），是迭代器的责任。设计适当的迭代器，则是容器的责任。唯容器本身，才知道该设计出怎样的迭代器来走访自己，并执行迭代器该有的各种行为（前进、后退、取值、取用成员...）。至于算法，完全可以独立于容器和迭代器之外自行发展，只要设计时以迭代器为对外接口就行。

traits 编程技法，大量运用于 STL 实作品[#]。它利用「巢状型别」的写码技巧与编译器的 template 自变量推导功能，补强 C++ 未能提供的关于型别认证方面的能力，补强 C++ 不为强型（strong typed）语言的遗憾。了解 **traits** 编程技法，就像获得「芝麻开门」口诀一样，从此得以一窥 STL 源码堂奥。

3.6 iterator 源码完整重列

由于讨论次序的缘故，先前所列的源码切割散落，有点凌乱。以下重新列出 SGI STL <stl_iterator.h> 表头档内与本章相关的程序代码。该表头档还有其它内容，是关于 iostream iterators、inserter iterators 以及 reverse iterators 的实作，将于第 8 章讨论。

```
// 节录自 SGI STL <stl_iterator.h>
// 五种迭代器类型
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 {};

// 为避免写码时挂一漏万，自行开发的迭代器最好继承自下面这个 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;
};

// 「榨汁机」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;
};

// 针对原生指标（native pointer）而设计的 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;
};
```

```
// 针对原生之 pointer-to-const 而设计的 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;
};

// 这个函式可以很方便地决定某个迭代器的类型 ( 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();
}

// 这个函式可以很方便地决定某个迭代器的 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);
}

// 这个函式可以很方便地决定某个迭代器的 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);
}

// 以下是整组 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());
}

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

template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator& i, Distance n,
                      bidirectional_iterator_tag) {
  if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --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 编程技法很棒，适度弥补了 C++ 语言本身的不足。STL 只对迭代器加以规范，制定出 iterator_traits 这样的东西。SGI 把这种技法进一步扩大到迭代器以外的世界，于是有了所谓的 __type_traits。双底线前缀词意指这是 SGI STL 内部所用的东西，不在 STL 标准规范之内。

iterator_traits 负责萃取迭代器的特性，__type_traits 则负责萃取型别

（type）的特性。此处我们所关注的型别特性是指：这个型别是否具备 non-trivial defalt ctor ？是否具备 non-trivial copy ctor ？是否具备 non-trivial assignment operator？是否具备 non-trivial dtor？如果答案是否定的，我们在对这个型别进行建构、解构、拷贝、赋值等动作时，就可以采用最有效率的措施（例如根本不唤起尸位素餐的那些 constructor，destructor），而采用记忆体直接处理动作如 malloc()、memcpy() 等等，获得最高效率。这对于大规模而动作频繁的容器，有着显著的效率提升[4]。

定义于 SGI <type_traits.h> 中的 __type_traits，提供了一种机制，允许针对不同的型别属性（type attributes），在编译时期完成函式派送决定（function dispatch）。这对于撰写 template 很有帮助，例如，当我们准备对一个「元素型别未知」的数组执行 copy 动作时，如果我们能事先知道其元素型别是否有一个 trivial copy constructor，便能够帮助我们决定是否可使用快速的 memcpy() 或 memmove()。

从 iterator_traits 得来的经验，我们希望，程式之中可以这样运用 __type_traits<T>，T 代表任意型别：

```
__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
```

我们希望上述式子响应我们「真」或「假」（以便我们决定采取什么策略），但其结果不应该只是个 bool 值，应该是个有着真/假性质的「对象」，因为我们希望利用其响应结果来进行自变量推导，而编译器只有面对 class object 形式的自变量，才会做自变量推导。为此，上述式子应该传回这样的东西：

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

这两个空白 classes 没有任何成员，不会带来额外负担，却又能够标示真假，满足我们所需。

C++ Type Traits, by John Maddock and Steve Cleary, DDJ 2000/10 提了一些测试数据。

为了达成上述五个式子，__type_traits 内必须定义一些 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;
           /* 不要移除这个成员。它通知「有能力自动将 __type_traits 特化」
            的编译器说，我们现在所看到的这个 __type_traits template 是特
            殊的。这是为了确保万一编译器也使用一个名为 __type_traits 而其
            实与此处定义并无任何关联的 template 时，所有事情都仍将顺利运作。
            */

   /* 以下条件应被遵守，因为编译器有可能自动为各型别产生专属的 __type_traits
      特化版本：
          - 你可以重新排列以下的成员次序
          - 你可以移除以下任何成员
          - 绝对不可以将以下成员重新命名而却没有改变编译器的对应名称
          - 新加入的成员会被视为一般成员，除非你在编译器加上适当支持。*/

   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;
};
```

为什么 SGI 把所有巢状型别都定义为 __false_type 呢？是的，SGI 定义出最保守的值，然后（稍后可见）再针对每一个纯量型别（scalar types）设计适当的 __type_traits 特化版本，这样就解决了问题。

上述 __type_traits 可以接受任何型别的自变量，五个 typedefs 将经由以下管道获得实值：

一般具现体（general instantiation），内含对所有型别都必定有效的保守值。

上述各个 has_trivial_xxx 型别都被定义为 __false_type，就是对所有型别都必定有效的保守值。

经过宣告的特化版本，例如 <type_traits.h> 内对所有 C++ 纯量型别（scalar types）提供了对映的特化宣告。稍后展示。

某些编译器（如 Silicon Graphics N32 和 N64 编译器）会自动为所有型别提供适当的特化版本。（这真是了不起的技术。不过我对其精确程度存疑）

以下便是 <type_traits.h> 对所有 C++ 纯量型别所定义的 __type_traits 特化版本。这些定义对于内建有 __types_traits 支持能力的编译器（例如 Silicon Graphics N32 和 N64）并无伤害，对于无该等支持能力的编译器而言，则属必要。

```
/* 以下针对 C++ 基本型别 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，每一个成员的值都是 __true_type，表示这些
型别都可采用最快速方式（例如 memcpy）来进行拷贝（copy）或赋值（assign）动
作。*/

// 注意，SGI STL <stl_config.h> 将以下出现的 __STL_TEMPLATE_NULL
// 定义为 template<>，见 1.9.1 节，是所谓的
// 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，以下针对原生指标设计 __type_traits 偏特化版本。
// 原生指标亦被视为一种纯量型别。
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 中的应用很广。下面我举几个实例。第一个例子是出现于本书 2.3.3 节的 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 为蓝本，自迭代器 first 开始建构 n 个元素。为求取最大效率，首先以 value_type()（3.6 节）萃取出迭代器 first 的 *value type*，再利用 __type_traits 判断该型别是否为 POD 型别：

```
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「是否为 POD 型别」采取最适当的措施：

```
// 如果不是 POD 型別，就会派送（dispatch）到这里
template <class ForwardIterator, class Size, class T>
ForwardIterator
__uninitialized_fill_n_aux(ForwardIterator first, Size n,
                           const T& x, __false_type) {
  ForwardIterator cur = first;
    // 为求阅读顺畅简化，以下将原本有的异常处理（exception handling）去除。
    for ( ; n > 0; --n, ++cur)
        construct(&*cur, x);              // 见 2.2.3 节
    return cur;
}

// 如果是 POD 型別，就会派送（dispatch）到这里。下两行是原档所附注解。
// 如果 copy construction 等同于 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);             // 交由高阶函式执行，如下所示。
}

// 以下是定义于 <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;
}
```

第三个例子是负责对象解构的 destroy() 全域函式。此函式之源码及解说在 2.2.3 节有完整的说明。

第四个例子是出现于本书第 6 章的 copy() 全域函式（泛型算法之一）。这个函式有非常多的特化（specialization）与强化（refinement）版本，殚精竭虑，全都是为了效率考虑，希望在适当的情况下采用最「雷霆万钧」的手段。最基本的想法是这样：

```
// 拷贝一个数组，其元素为任意型別，视情况采用最有效率的拷贝手段。
template <class T> inline void copy(T* source,T* destination,int n) {
   copy(source,destination,n,
        typename __type_traits<T>::has_trivial_copy_constructor() );
}

// 拷贝一个数组，其元素型别拥有 non-trivial copy constructors。
```

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

// 拷贝一个数组，其元素型别拥有 trivial copy constructors。
// 可借助 memcpy() 完成工作
template <class T> void copy(T* source,T* destination,int n,
                             __true_type)
{ ... }
```

以上只是针对「函式参数为原生指标」的情况而做的设计。第 6 章的 copy() 演算法是个泛型版本，情况又复杂许多。详见 6.4.3 节。

请注意，<type_traits.h> 并未像其它许多 SGI STL 表头档有这样的声明：

```
/* 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运用这个 __type_traits。假设我自行定义了一个 Shape class，__type_traits 会对它产生什么效应？如果编译器够厉害（例如 Silicon Graphics 的 N32 和 N64 编译器），你会发现，__type_traits 针对 Shape 萃取出来的每一个特性，其结果将取决于我的 Shape 是否有 trivial defalt ctor 或 trivial copy ctor 或 trivial assignment operator 或 trivial dtor 而定。但对大部份缺乏这种特异功能的编译器而言，__type_traits 针对 Shape 萃取出来的每一个特性都是 __false_type，即使 Shape 是个 POD 型别。这样的结果当然过于保守，但是别无选择，除非我针对 Shape，自行设计一个 __type_traits 特化版本，明白地告诉编译器以下事实（举例）：

```
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;
};
```

究竟一个 class 什么时候该有自己的 non-trivial default constructor, non-trivial copy constructor, non-trivial assignment operator, non-trivial destructor 呢？一个简单的判

断准则是：如果 class 内含指标成员，并且对它进行内存动态配置，那么这个 class 就需要实作出自己的 non-trivial-xxx[5]。

即使你无法全面针对你自己定义的型别，设计__type_traits 特化版本，无论如何， 至少，有了这个 __type_traits 之后，当我们设计新的泛型算法时，面对 C++ 纯量型别，便有足够的信息决定采用最有效的拷贝动作或赋值动作 — 因为每一个纯量型别都有对应的 __type_traits 特化版本，其中每一个 typedef 的值都是 __true_type。

请参考 [Meyers98] 条款 11: *Declare a copy constructor and an assignment operator for classes with dynamically allocated memory*，以及条款 45：*Know what functions C++ silently writes and calls*.

4

序列式容器

sequence containers

4.1 容器的概观与分类

容器，置物之所也。

研究数据的特定排列方式，以利搜寻或排序或其它特殊目的，这一门学科我们称为数据结构（Data Structures）。大学信息相关教育里头，与编程最有直接关系的科目，首推数据结构与算法（Algorithms）。几乎可以说，任何特定的数据结构都是为了实现某种特定的算法。STL 容器即是将运用最广的一些数据结构实作出来（图 4-1）。未来，在每五年召开一次的 C++ 标准委员会[#]，STL 容器的数量还有可能增加。

众所周知，常用的数据结构不外乎 array（数组）、list（串行）、tree（树）、stack（堆栈）、queue（队列）、hash table（杂凑表）、set（集合）、map（映像表）…等等。根据「资料在容器[#]的排列」特性，这些数据结构分为序列式（sequence）和关系型（associative）两种。本章探讨序列式容器，下一章探讨关系型容器。

容器是大多数人对 STL 的第一印象，这说明了容器的好用与受欢迎。容器也是许多人对 STL 的唯一印象，这说明了还有多少人利器（STL）在手而未能善用。

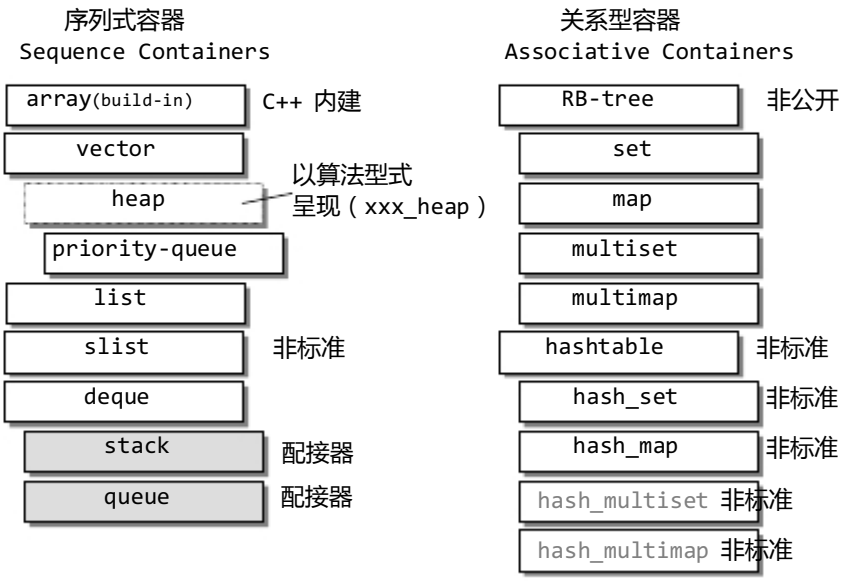


图 4-1 SGI STL 的各种容器。本图以内缩方式来表达基层与衍生层的关系。
这里所谓的衍生，并非继承（inheritance）关系，而是内含（containment）关系。
例如 heap 内含一个 vector，priority-queue 内含一个 heap，stack 和 queue 都含一个 deque，set/map/multiset/multimap 都内含一个 RB-tree，hast_x 都内含一个 hashtable。

4.1.1 序列式容器（sequential containers）

所谓序列式容器，其中的元素都可序（*ordered*），但未排序（*sorted*）。C++ 语言本身提供了一个序列式容器 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 由于只是将 deque 改头换面而成，技术上被归类为一种配接器（adapter），但我仍把它们放在本章讨论。

本章将带你仔细看过各种序列式容器的关键实作细节。

4.2 vector

4.2.1 vector 概述

vector 的数据安排以及操作方式，与 array 非常像似。两者的唯一差别在于空间的运用弹性。array 是静态空间，一旦配置了就不能改变；要换个大（或小）一点的房子，可以，一切细琐得由客端自己来：首先配置一块新空间，然后将元素从旧址一一搬往新址，然后再把原来的空间释还给系统。vector 是动态空间，随着元素的加入，它的内部机制会自行扩充空间以容纳新元素。因此，vector 的运用对于内存的樽节与运用弹性有很大的帮助，我们再也不必因为害怕空间不足而一开始就要求一个大块头 array 了，我们可以安心使用 vector，吃多少用多少。

vector 的实作技术，关键在于其对大小的控制以及重新配置时的数据搬移效率。一旦 vector 旧有空间满载，如果客端每新增一个元素，vector 内部只是扩充一个元素的空间，实为不智，因为所谓扩充空间（不论多大），一如稍早所说，是「配置新空间 / 数据搬移 / 释还旧空间」的大工程，时间成本很高，应该加入某种未雨绸缪的考虑。稍后我们便可看到 SGI vector 的空间配置策略。

4.2.2 vector 定义式摘要

以下是 vector 定义式的源码摘录。虽然 STL 规定，欲使用 vector 者必须先含入 <vector>，但 SGI STL 将 vector 实作于更底层的 <stl_vector.h>。

```
// alloc 是 SGI STL 的空间配置器，见第二章。
template <class T, class Alloc = alloc>
class vector {
public:
  // vector 的巢状型别定义
  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 的空间配置器，见 2.2.4 节。
  typedef simple_alloc<value_type, Alloc> data_allocator;

  iterator start;               // 表示目前使用空间的头
  iterator finish;              // 表示目前使用空间的尾
  iterator end_of_storage; // 表示目前可用空间的尾

  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，见 2.2.3 节。
    deallocate();                           // 这是 vector 的一个 member function

  reference front() { return *begin(); }                          // 第一个元素
  reference back() { return *(end() - 1); }                       // 最后一个元素
  void push_back(const T& x) {                        // 将元素安插至最尾端
    if (finish != end_of_storage) {
      construct(finish, x);                 // 全域函式，见 2.2.3 节。
      ++finish;
    }
    else
      insert_aux(end(), x);                 // 这是 vector 的一个 member function
  }

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

```
    --finish;
    destroy(finish);                    // 全域函式，见 2.2.3 节。
  }

  iterator erase(iterator position) {                    // 清除某位置上的元素
    if (position + 1 != end())
        copy(position + 1, finish, position);                    // 后续元素往前搬移
    --finish;
    destroy(finish);                    // 全域函式，见 2.2.3 节。
    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:
  // 配置空间并填满内容
  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，见 2.3 节
      return result;
  }
```

4.2.3 vector 的迭代器

vector 维护的是一个连续线性空间，所以不论其元素型别为何，原生指标都可以做为 vector 的迭代器而满足所有必要条件，因为 vector 迭代器所需要的操作行为如operator*,operator->,operator++,operator--,operator+, operator-, operator+=,operator-=，原生指标是一种 生就具备。vector 支援随机存取，而原生指标正有着这样的能力。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 的迭代器是原生指标
...
};
```

根据上述定义，如果客端写出这样的码：

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

ivite 的型別其实就是 int*，svite 的型別其实就是 Shape*。

4.2.4 vector 的数据结构

vector 所采用的数据结构非常简单：线性连续空间。它以两个迭代器 start 和 finish 分别指向配置得来的连续空间中目前已被使用的范围，并以迭代器 end_of_storage 指向整块连续空间（含备用空间）的尾端：

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

为了降低空间配置时的速度成本，vector 实际配置的大小可能比客端需求量更大一些，以备将来可能的扩充。这便是容量（capacity）的观念。换句话说一个 vector 的容量永远大于或等于其大小。一旦容量等于大小，便是满载，下次再有新增元素，整个 vector 就得另觅居所。见图 4-2。

运用 start, finish, end_of_storage 三个迭代器，便可轻易提供首尾标示、大小、容量、空容器判断、注标（[ ]）运算子、最前端元素值、最后端元素值...等机能：

```
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); }
...
};
```

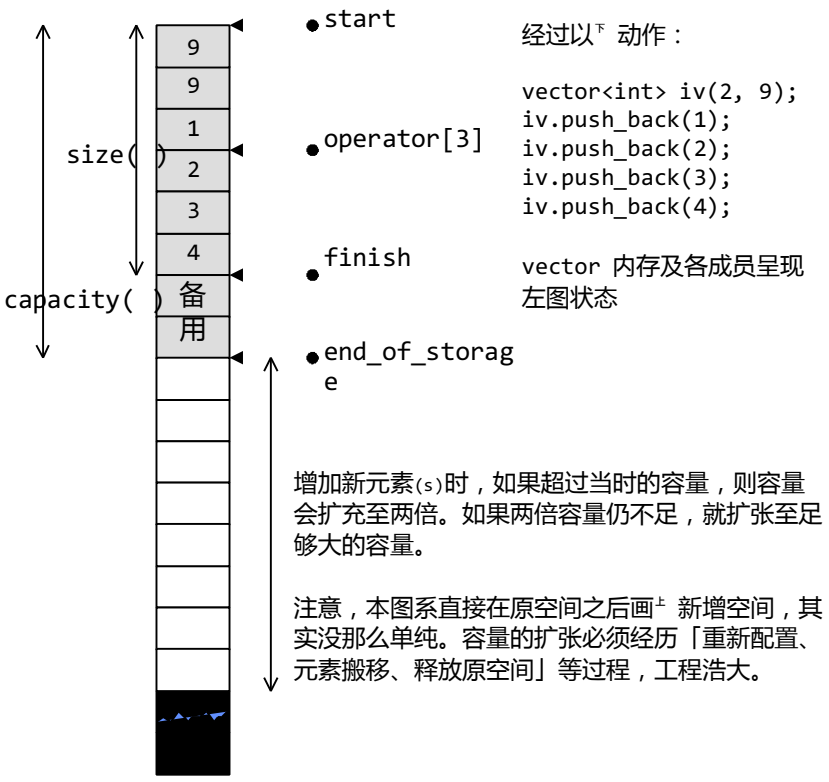


图 4-2 vector 示意图

4.2.5 vector 的建构与内存管理：constructor, push_back

千头万绪该如何说起？以客端程序代码为引导，观察其所得结果并实证源码，是个良好的学习路径。下面是个小小的测试程序，我的观察重点在建构的方式、元素的添加，以及大小、容量的变化：

```
// 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 预设使用 alloc（第二章）做为空间配置器，并据此另外定义了一个 data_allocator，为的是更方便以元素大小为配置单位：

```
template <class T, class Alloc = alloc>
class vector {
protected:
  // simple_alloc<> 见 2.2.4 节
  typedef simple_alloc<value_type, Alloc> data_allocator;
...
};
```

于是，data_allocator::allocate(n) 表示配置 n 个元素空间。

vector 提供许多 constructors，其中一个允许我们指定空间大小及初值：

```
    // 建构式，允许指定 vector 大小 n 和初值 value
    vector(size_type n, const T& value) { fill_initialize(n, value); }

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

    // 配置而后充填
    iterator allocate_and_fill(size_type n, const T& x) {
```

```
    iterator result = data_allocator::allocate(n); // 配置 n 个元素空间
    uninitialized_fill_n(result, n, x); // 全域函式，见 2.3 节
    return result;
  }
```

uninitialized_fill_n() 会根据第二参数的型别特性（type traits，3.7 节），决定使用算法 fill_n() 或反复呼叫 construct() 来完成任务（见 2.3 节描述）。

当我们以 push_back() 将新元素安插于 vector 尾端，该函式首先检查是否还有备用空间？如果有就直接在备用空间上建构元素，并调整迭代器finish，使 vector 变大。如果没有备用空间了，就扩充空间（重新配置、搬移数据、释放原空间）：

```
  void push_back(const T& x) {
    if (finish != end_of_storage) {                    // 还有备用空间
      construct(finish, x);                            // 全域函式，见 2.2.3 节。
      ++finish;                                        // 调整水位高度
    }
    else                                               // 已无备用空间
      insert_aux(end(), x); // vector member function，见以下 列表
  }

template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
  if (finish != end_of_storage) {                      // 还有备用空间
    // 在备用空间起始处建构一个元素，并以 vector 最后一个元素值为其初值。
    construct(finish, *(finish - 1));
    // 调整水位。
    ++finish;
    T x_copy = x;
    copy_backward(position, finish - 2, finish - 1);
    *position = x_copy;
  }
  else {        // 已无备用空间
    const size_type old_size = size();
    const size_type len = old_size != 0 ? 2 * old_size : 1;
    // 以上 配置原则：如果原大小为 0，则配置 1（个元素大小）；
    // 如果原大小不为 0，则配置原大小的两倍，
    // 前半段用来放置原资料，后半段准备用来放置新资料。

    iterator new_start = data_allocator::allocate(len); // 实际配置
    iterator new_finish = new_start;
    try {
      // 将原 vector 的内容拷贝到新 vector。
      new_finish = uninitialized_copy(start, position, new_start);
      // 为新元素设定初值 x
      construct(new_finish, x);
      // 调整水位。
```

```
      ++new_finish;
      // 将原 vector 的备用空间中的内容也忠实拷贝过来（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;
    }

    // 解构并释放原 vector
    destroy(begin(), end());
    deallocate();

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

注意，所谓动态增加大小，并不是在原空间之后接续新空间（因为无法保证原空间之后尚有可供配置的空间），而是以原大小的两倍另外配置一块较大空间，然后将原内容拷贝过来，然后才开始在原内容之后建构新元素，并释放原空间。因此，对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了。这是程序员易犯的一个错误，务需小心。

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

vector 所提供的元素操作动作很多，无法在有限篇幅中一一讲解——其实也没有这种必要。为搭配先前对空间配置的讨论，我挑选数个相关函式做为解说对象。这些函式也出现在先前的测试程序中。

```
// 将尾端元素拿掉，并调整大小。
void pop_back() {
  --finish;                  // 将尾端标记往前移一格，表示将放弃尾端元素。
  destroy(finish);           // destroy 是全域函式，见第 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;
  }

  // 清除某个位置上的元素
  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 节
    return position;
  }

  void clear() { erase(begin(), end()); } // erase() 就定义在上面
```

图 4-3a 展示 erase(first, last) 的动作。

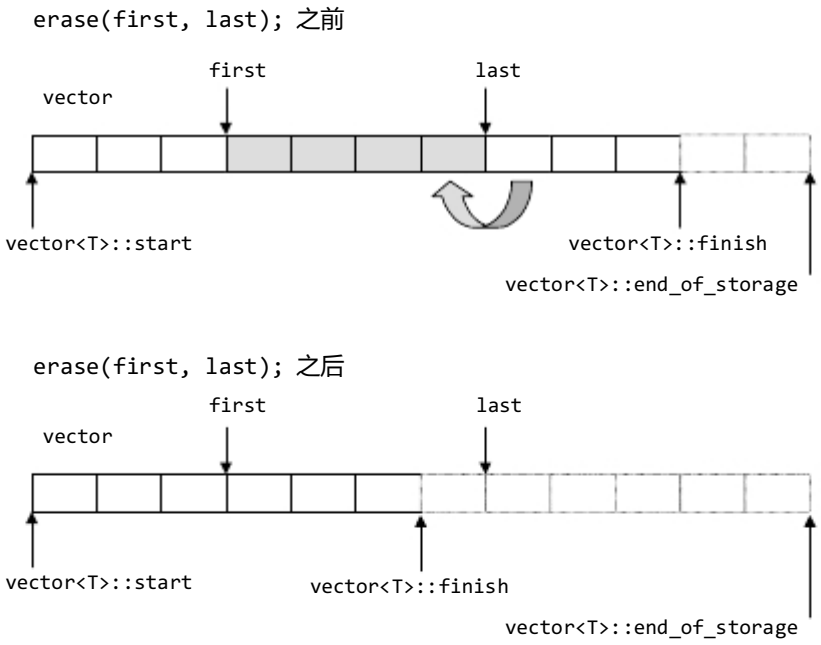


图 4-3a 局部区间的清除动作：erase(first,last)

下面是 vector::insert() 实作内容：

```
// 从 position 开始，安插 n 个元素，元素初值为 x
template <class T, class Alloc>
```

```
void vector<T, Alloc>::insert(iterator position, size_type n, const T& x)
{
  if (n != 0) { // 当 n != 0 才进行以下所有动作
    if (size_type(end_of_storage - finish) >= n)
      // 备用空间大于等于「新增元素个数」
      T x_copy = x;
      // 以下计算安插点之后的现有元素个数
      const size_type elems_after = finish - position;
      iterator old_finish = finish;
      if (elems_after > n)
        // 「安插点之后的现有元素个数」大于「新增元素个数」
        uninitialized_copy(finish - n, finish, finish);
        finish += n;          // 将 vector 尾端标记后移
        copy_backward(position, old_finish - n, old_finish);
        fill(position, position + n, x_copy);                         // 从安插点开始填入新值
      }
      else {
        // 「安插点之后的现有元素个数」小于等于「新增元素个数」
        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 {
      // 备用空间小于「新增元素个数」（那就必须配置额外的内存）
      // 首先决定新长度：旧长度的两倍，或旧长度+新增元素个数。
      const size_type old_size = size();
      const size_type len = old_size + max(old_size, n);
      // 以下配置新的 vector 空间
      iterator new_start = data_allocator::allocate(len);
      iterator new_finish = new_start;
      __STL_TRY {
        // 以下首先将旧 vector 的安插点之前的元素复制到新空间。
        new_finish = uninitialized_copy(start, position, new_start);
        // 以下再将新增元素（初值皆为 n）填入新空间。
        new_finish = uninitialized_fill_n(new_finish, n, x);
        // 以下再将旧 vector 的安插点之后的元素复制到新空间。
        new_finish = uninitialized_copy(position, finish, new_finish);
      }
# ifdef __STL_USE_EXCEPTIONS
      catch(...) {
        // 如有异常发生，实现 "commit or rollback" semantics.
        destroy(new_start, new_finish);
        data_allocator::deallocate(new_start, len);
        throw;
      }
# endif /* __STL_USE_EXCEPTIONS */
```

```
        // 以下 清除并释放旧的 vector
        destroy(start, finish);
        deallocate();
        // 以下 调整水位标记
        start = new_start;
        finish = new_finish;
        end_of_storage = new_start + len;
      }
    }
  }
```

注意，安插完成后，新节点将位于标兵迭代器（上 例之 position，标示出安插点）所指之节点的前方 — 这是 STL 对于「安插动作」的标准规范。图 4-3b 展 示 insert(position,n,x) 的动作。

insert(position,n,x);

(1) 备用空间 2 ≥ 新增元素个数 2
例：下 图，n==2

(1-1)安插点之后的现有元素个数 3 > 新增元素个数 2

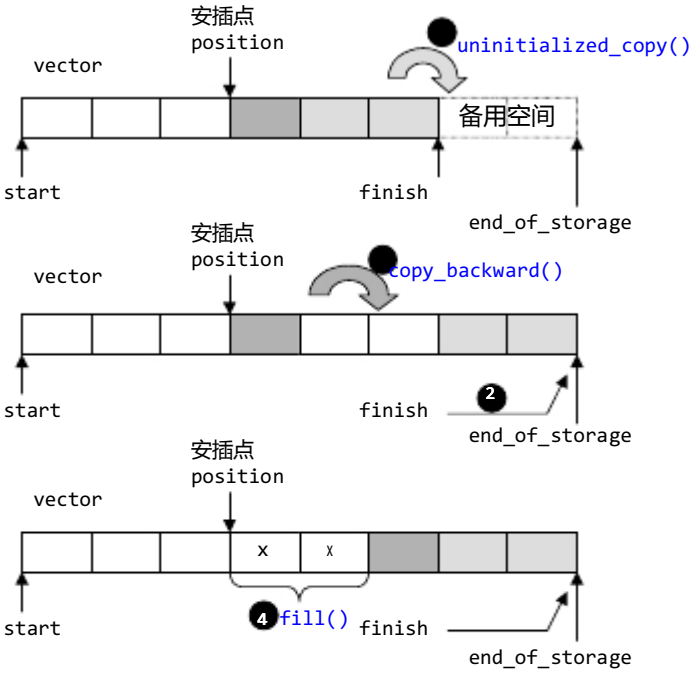


图 4-3b-1 insert(position,n,x) 状况 1

(1-2)安插点之后的现有元素个数 2 ≤ 新增元素个数 3

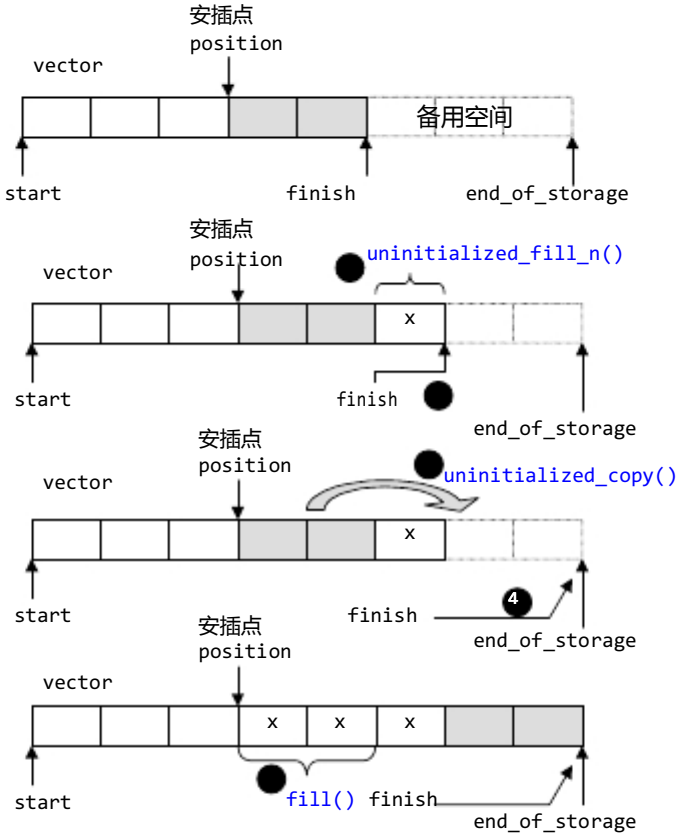


图 4-3b-2 insert(position,n,x) 状况 2

insert(position,n,x);

(2) 备用空间 < 新增元素个数
例：下图，n==3

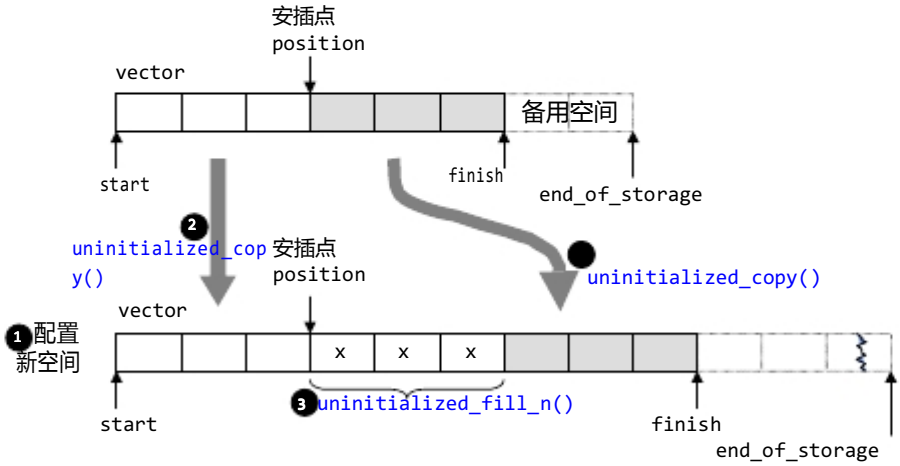


图 4-3b-3 insert(position,n,x) 状况 3

4.3 list

4.3. list 概述

1

相较于 vector 的连续线性空间，list 就显得复杂许多，它的好处是每次安插或删除一个元素，就配置或释放一个元素空间。因此，list 对于空间的运用有绝对的精准，一点也不浪费。而且，对于任何位置的元素安插或元素移除，list 永远是常数时间。

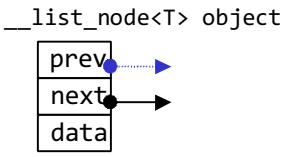
list 和 vector 是两个最常被使用的容器。什么时机下最适合使用哪一种容器，必须视元素的多寡、元素的构造复杂度（有无 non-trivial copy constructor, non-trivial copy assignmen operator）、元素存取行为的特性而定。[Lippman 98] 6.3 节对这两种容器提出了一份测试报告。

4.3.2 list 的节点（node）

每一个设计过 list 的人都知道，list 本身和 list 的节点是不同的结构，需要分开设计。以下是 STL list 的节点（node）结构：

```
template <class T>
struct __list_node {
  typedef void* void_pointer;
  void_pointer prev;            // 型别为 void*。其实可设为 __list_node<T>*
  void_pointer next;
  T data;
};
```

显然这是一个双向串行[1]。



4.3.3 list 的迭代器

list 不再能够像 vector 一样以原生指标做为迭代器，因为其节点不保证在储存空间中连续存在。list 迭代器必须有能力指向 list 的节点，并有能力做正确的递增、递减、取值、成员存取...等动作。所谓「list 迭代器正确的递增、递减、取值、成员取用」动作是指，递增时指向下一个节点，递减时指向上一个节点，取值时取的是节点的资料值，成员取用时取用的是节点的成员，如图 4-4。

由于 STL list 是一个双向串行（double linked-list），迭代器必须具备前移、后移的能力。所以 list 提供的是 *Bidirectional Iterators*。

list 有一个重要性质：安插动作（insert）和接合动作（splice）都不会造成原有的 list 迭代器失效。这在 vector 是不成立的，因为 vector 的安插动作可能造成记忆体重新配置，导致原有的迭代器全部失效。甚至 list 的元素删除动作

---

[1] SGI STL 另有一个单向串行 slist，我将在 4.9 节介绍它。

（erase），也只有「指向被删除元素」的那个迭代器失效，其它迭代器不受任何影响。

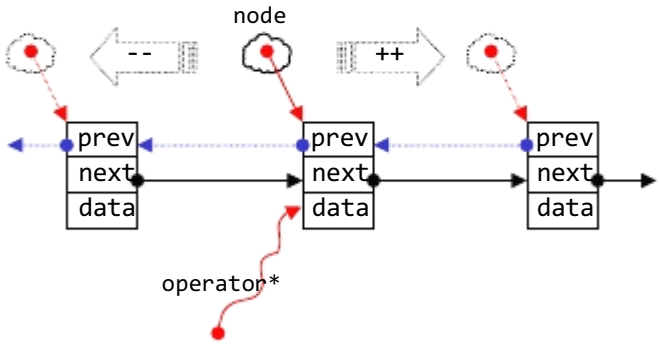


图 4-4 list 的节点与 list 的迭代器

以下是 list 迭代器的设计：

```
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; // 迭代器内部当然要有一个原生指标，指向 list 的节点

  // 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; }
  // 以下对迭代器取值（dereference），取的是节点的资料值。
  reference operator*() const { return (*node).data; }

  // 以下是迭代器的成员存取（member access）运算子的标准作法。
```

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

    // 对迭代器累加 1，就是前进一个节点
    self& operator++()
      node = (link_type)((*node).next);
      return *this;
    }
    self operator++(int)
      self tmp = *this;
      ++*this;
      return tmp;
    }

    // 对迭代器递减 1，就是后退一个节点
    self& operator--()
      node = (link_type)((*node).prev);
      return *this;
    }
    self operator--(int)
      self tmp = *this;
      --*this;
      return tmp;
    }
  };
```

4.3.4 list 的数据结构

SGI list 不仅是一个双向串行，而且还是一个环状双向串行。所以它只需要一个指标，便可以完整表现整个串行：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class list {
protected:
  typedef __list_node<T> list_node;
public:
  typedef list_node* link_type;

protected:
  link_type node; // 只要一个指标，便可表示整个环状双向串行
...
};
```

如果让指标 node 指向刻意置于尾端的一个空白节点，node 便能符合 STL 对于「前闭后开」区间的要求，成为 last 迭代器，如图 4-5。这么一来，以下几个函式便都可以轻易完成：

```
  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;
  }
  // 取头节点的内容（元素值）。
  reference front() { return *begin(); }
  // 取尾节点的内容（元素值）。
  reference back() { return *(--end()); }
```

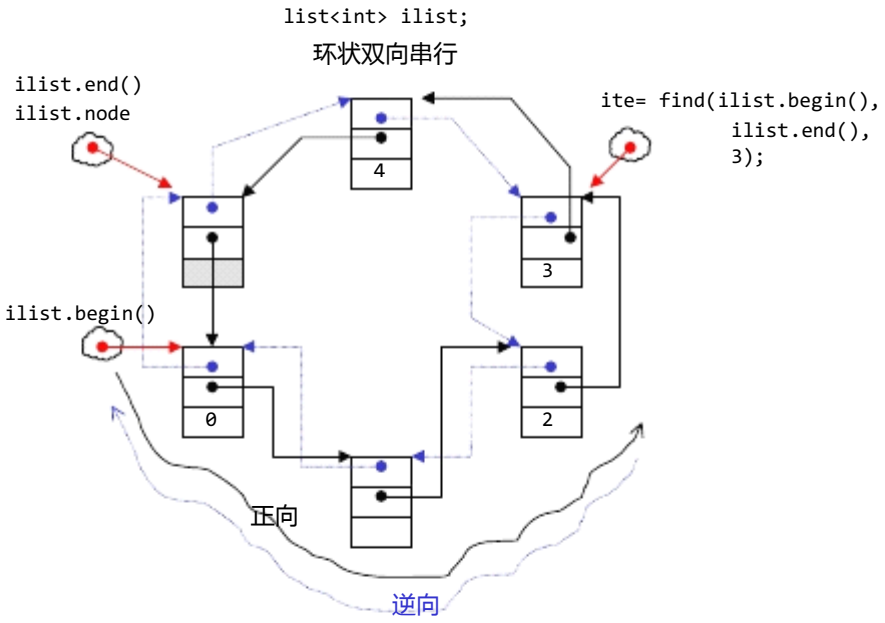


图 4-5 list 示意图。是环状串行只需一个标记，即可完全表示整个串行。只要刻意在环状串行的尾端加上一个空白节点，便符合 STL 规范之「前闭后开」区间。

4.3.5 list 的建构与内存管理：

constructor, push_back, insert

千头万绪该如何说起？以客端程序代码为引导，观察其所得结果并实证源码，是个

良好的学习路径。下面是一个测试程序，我的观察重点在建构的方式以及大小的变化：

```
// 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 预设使用 alloc（2.2.4 节）做为空间配置器，并据此另外定义了一个 list_node_allocator，为的是更方便地以节点大小为配置单位：

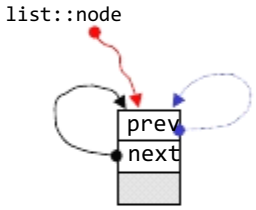
```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class list {
protected:
  typedef __list_node<T> list_node;
  // 专属之空间配置器，每次配置一个节点大小：
  typedef simple_alloc<list_node, Alloc> list_node_allocator;
...
};
```

于是，list_node_allocator(n) 表示配置 n 个节点空间。以下四个函式，分别用来配置、释放、建构、摧毁一个节点：

```
protected:
  // 配置一个节点并传回
  link_type get_node() { return list_node_allocator::allocate(); }
  // 释放一个节点
  void put_node(link_type p) { list_node_allocator::deallocate(p); }

  // 产生（配置并建构）一个节点，带有元素值
  link_type create_node(const T& x) {
    link_type p = get_node();
    construct(&p->data, x);// 全域函式，建构/解构基本工具。
    return p;
  }
  // 摧毁（解构并释放）一个节点
  void destroy_node(link_type p) {
    destroy(&p->data);                  // 全域函式，建构/解构基本工具。
    put_node(p);
  }
```

list 提供有许多 constructors，其中一个是 default constructor，允许我们不指定任何参数做出一个空的 list 出来：

```
public:
  list() { empty_initialize(); }                    // 产生一个空串行。

protected:
  void empty_initialize()
    node = get_node(); // 配置一个节点空间，令 node 指向它。
    node->next = node; // 令 node 头尾都指向自己，不设元素值。
    node->prev = node;
  }
```



empty list（空串行）

当我们以 push_back() 将新元素安插于 list 尾端，此函式内部呼叫 insert()：

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

insert() 是一个多载化函式，有多种型式，其中最简单的一种如下，符合以上所需。首先配置并建构一个节点，然后在尾端做适当的指标动作，将新节点安插进去：

```
// 函式目的：在迭代器 position 所指位置安插一个节点，内容为 x。
iterator insert(iterator position, const T& x) {
  link_type tmp = create_node(x); // 产生一个节点（设妥内容为 x）
  // 调整双向指标，使 tmp 安插进去。
  tmp->next = position.node;
  tmp->prev = position.node->prev;
  (link_type(position.node->prev))->next = tmp;
  position.node->prev = tmp;
  return tmp;
}
```

于是，先前测试程序连续安插了五个节点（其值为 0 1 2 3 4）之后，list 的状态如图 4-5。如果我们希望在 list 内的某处安插新节点，首先必须确定安插位置，例如我希望在资料值为 3 的节点处安插一个数据值为 99 的节点，可以这么做：

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

find() 动作稍后再做说明。安插之后的 list 状态如图 4-6。注意，安插完成后，新节点将位于标兵迭代器（标示出安插点）所指之节点的前方 — 这是 STL 对于「安插动作」的标准规范。由于 list 不像 vector 那样有可能在空间不足时做重新配置、数据搬移的动作，所以安插前的所有迭代器在安插动作之后都仍然有效。

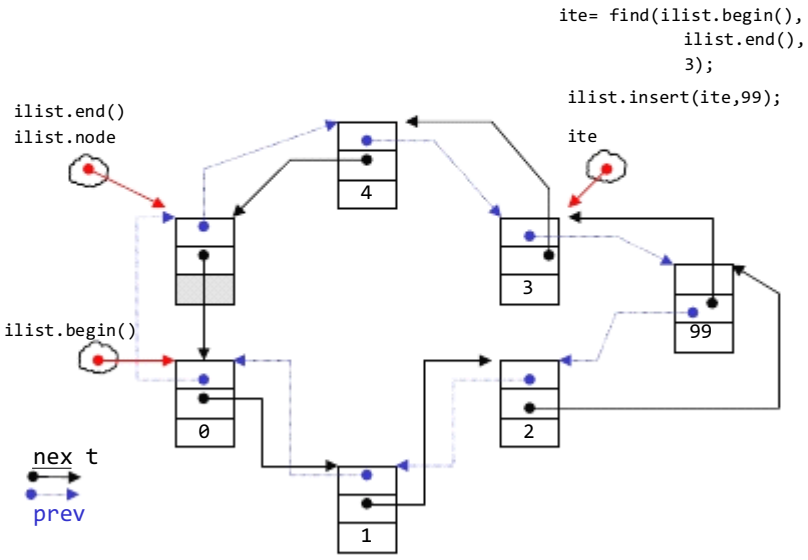


图 4-6 安插新节点 99 于节点 3 的位置[1]（所谓安插是指「安插在...之前」）

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 所提供的元素操作动作很多，无法在有限的篇幅[4]一一讲解 — 其实也没有这种必要。为搭配先前对空间配置的讨论，我挑选数个相关函式做为解说对象。先前示例[4]出现有尾部安插动作（push_back），现在我们来看看其它的安插动作和移除动作。

```
// 安插一个节点，做为头节点
void push_front(const T& x) { insert(begin(), x); }
// 安插一个节点，做为尾节点（上一小节才介绍过）
void push_back(const T& x) { insert(end(), x); }

// 移除迭代器 position 所指节点
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);
```

```
  }

  // 移除头节点
  void pop_front() { erase(begin()); }
  // 移除尾节点
  void pop_back()
    iterator tmp = end();
    erase(--tmp);
  }

// 清除所有节点（整个串行）
template <class T, class Alloc>
void list<T, Alloc>::clear()
{
  link_type cur = (link_type) node->next; // begin()
  while (cur != node) {                // 巡访每一个节点
    link_type tmp = cur;
    cur = (link_type) cur->next;
    destroy_node(tmp);                 // 摧毁（解构并释放）一个节点
  }
  // 恢复 node 原始状态
  node->next = node;
  node->prev = node;
}

// 将数值为 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) {              // 巡访每一个节点
    iterator next = first;
    ++next;
    if (*first == value) erase(first);                    // 找到就移除
    first = next;
  }
}

// 移除数值相同的连续元素。注意，只有「连续而相同的元素」，才会被移除剩一个。
template <class T, class Alloc>
void list<T, Alloc>::unique() {
  iterator first = begin();
  iterator last = end();
  if (first == last) return;                  // 空串行，什么都不必做。
  iterator next = first;
  while (++next != last) {                    // 巡访每一个节点
    if (*first == *next)                      // 如果在此区段中有相同的元素
      erase(next);                            // 移除之
    else
```

```
        first = next;                // 调整指标
      next = first;                  // 修正区段范围
    }
  }
```

由于 list 是一个双向环状串行，只要我们把边际条件处理好，那么，在头部或尾部安插元素（push_front 和 push_back），动作几乎是一样的，在头部或尾部移除元素（pop_front 和 pop_back），动作也几乎是一样的。移除（erase）某个迭代器所指元素，只是做一些指标搬移动作而已，并不复杂。如果图 4-6 再经以下搜寻并移除的动作，状况将如图 4-7。

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

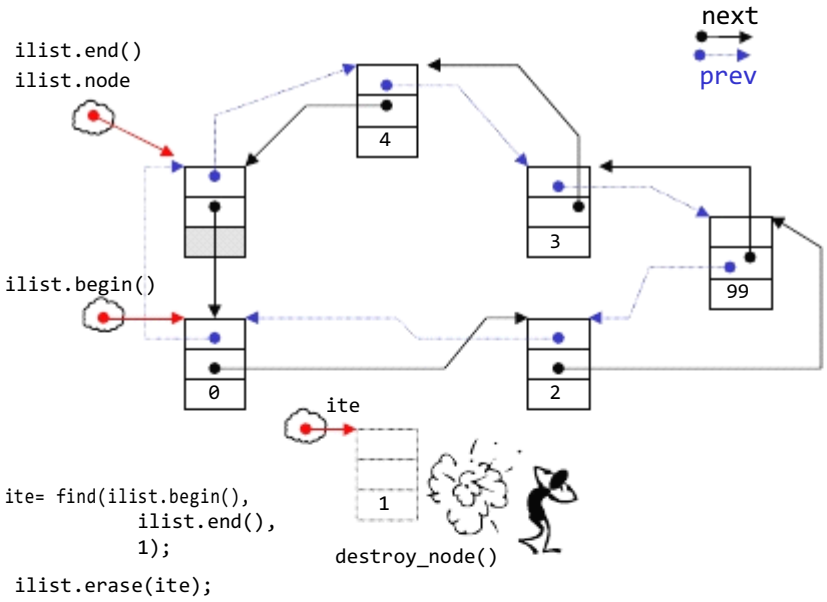


图 4-7 移除「元素值为 1」的节点

list 内部提供一个所谓的迁移动作（transfer）：将某连续范围的元素迁移到某个特定位置之前。技术上很简单，节点间的指标移动而已。这个动作为其它的复杂动作如 splice, sort, merge 等奠定良好的基础。下面是 transfer 的源码：

```
protected:
  // 将 [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)
    }
  }
```

以上七个动作，一步一步地显示于图 4-8a。

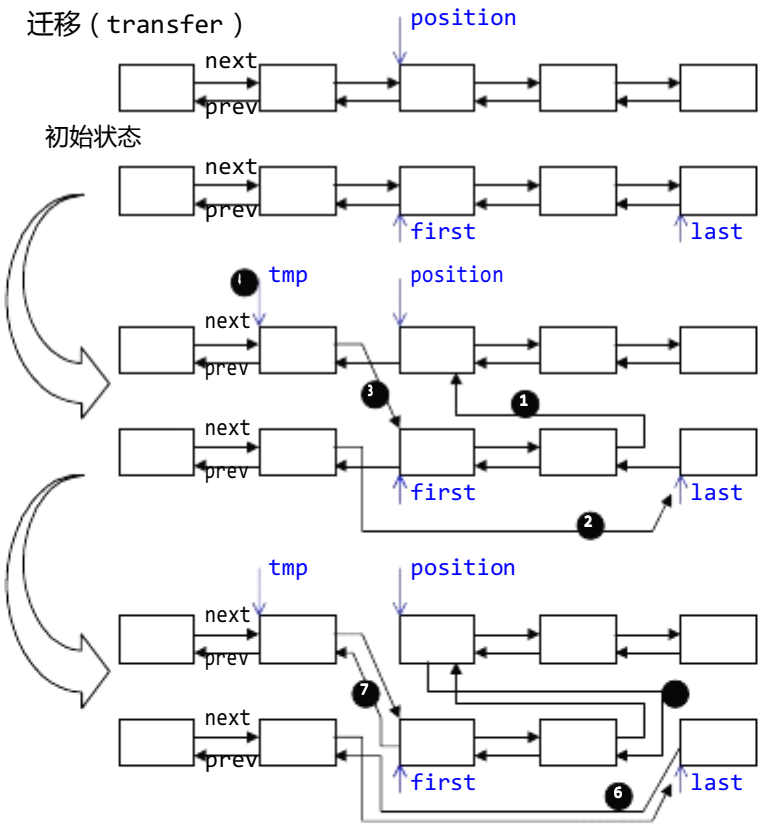


图 4-8a list<T>::transfer 的动作示意

transfer 所接受的 [first,last) 区间，是否可以在同一个 list 之中呢？答案是可以。你只要想象图 4-8a 所画的两条 lists 其实是同一个 list 的两个区段，就不难得到答案了。

上述的 transfer 并非公开界面。list 公开提供的是所谓的接合动作（splice）：将某连续范围的元素从一个 list 搬移到另一个（或同一个）list 的某个定点。如果接续先前 4list-test.cpp 程序的最后执行点，继续执行以下 splice 动作：

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

// 目前，ilist 的内容为 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。图 4-8b 显示接合动作。技术上很简单，只是节点间的指标移动而已，这些动作已完全由 transfer() 做掉了。

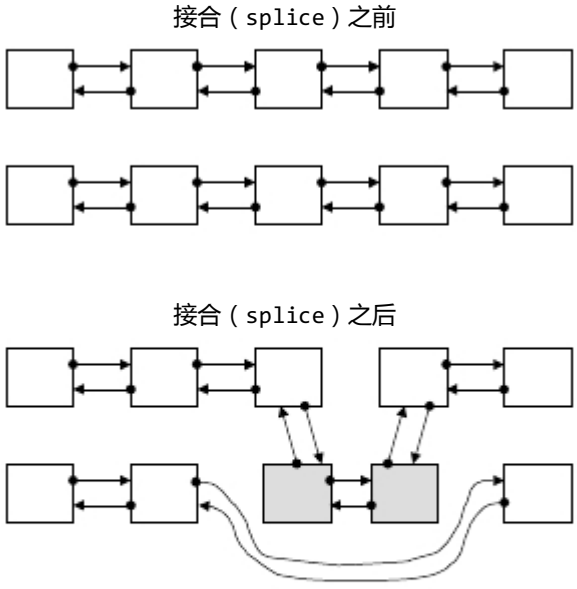


图 4-8b list 的接合（splice）动作

为了提供各种接口弹性，list<T>::splice 有许多版本：

```
public:
  // 将 x 接合于 position 所指位置之前。x 必须不同于 *this。
  void splice(iterator position, list& x) {
    if (!x.empty())
      transfer(position, x.begin(), x.end());
  }

  // 将 i 所指元素接合于 position 所指位置之前。position 和 i 可指向同一个 list。
  void splice(iterator position, list&, iterator i) {
    iterator j = i;
    ++j;
    if (position == i || position == j) return;
    transfer(position, i, j);
  }

  // 将 [first,last) 内的所有元素接合于 position 所指位置之前。
  // position 和[first,last)可指向同一个 list，
  // 但 position 不能位于[first,last)之内。
  void splice(iterator position, list&, iterator first, iterator last) {
    if (first != last)
      transfer(position, first, last);
  }
```

以下是 merge(), reverse(), sort() 的源码。有了 transfer() 在手，这些动作都不难完成。

```
// merge() 将 x 合并到 *this 身上。两个 lists 的内容都必须先经过递增排序。
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，两个 lists 都已经过递增排序，
  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，如果是空白串行，或仅有一個元素，就不做任何动作。
  // 使用 size() == 0 || size() == 1 來判斷，虽然也可以，但是比较慢。
  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 算法 sort()，必須使用自己的 sort() member function，
// 因為 STL 算法 sort() 只接受 RamdonAccessIterator.
// 本函式采用 quick sort.
template <class T, class Alloc>
void list<T, Alloc>::sort() {
  // 以下判斷，如果是空白串行，或仅有一個元素，就不做任何动作。
  // 使用 size() == 0 || size() == 1 來判斷，虽然也可以，但是比较慢。
  if (node->next == node || link_type(node->next)->next  == node)
       return;

  // 一些新的 lists，做為中介數据存放区
  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. deque 概述

1

vector 是单向开口的连续线性空间，deque 则是一种双向开口的连续线性空间。所谓双向开口，意思是可以在头尾两端分别做元素的安插和删除动作，如图 4-9。vector 当然也可以在头尾两端做动作（从技术观点），但是其头部动作效率奇差，无法被接受。

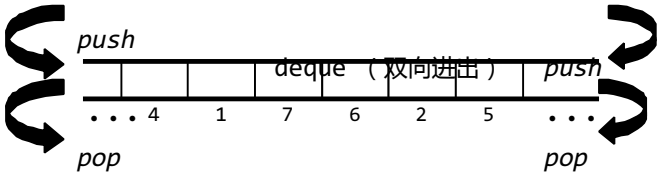


图 4-9 deque 示意

deque 和 vector 的最大差异，一在于 deque 允许于常数时间内对起头端进行元素的安插或移除动作，二在于 deque 没有所谓容量（capacity）观念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来。换句话说，像 vector 那样「因旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间」这样的事情在 deque 是不会发生的。也因此，deque 没有必要提供所谓的空间保留（reserve）功能。

虽然 deque 也提供 *Ramdon Access Iterator*，但它的迭代器并不是原生指标，其复杂度和 vector 不可以道里计（稍后看到源码，你便知道），这当然在在影响了各个运算层面。因此，除非必要，我们应尽可能选择使用 vector 而非 deque。对 deque 进行的排序动作，为了最高效率，可将 deque 先完整复制到一个 vector 身上，将 vector 排序后（利用 STL sort 算法），再复制回 deque。

4.4.2 deque 的中控器

deque 是连续空间（至少逻辑看来如此），连续线性空间总令我们联想到 array 或 vector。array 无法成长，vector 虽可成长，却只能向尾端成长，而且其所谓成长原是个假象，事实上是（1）另觅更大空间、（2）将原数据复制过去、（3）释放原空间三部曲。如果不是 vector 每次配置新空间时都有留下一些余裕，其「成长」假象所带来的代价将是相当高昂。

deque 系由一段一段的定量连续空间构成。一旦有必要在 deque 的前端或尾端增加新空间，便配置一段定量连续空间，串接在整个 deque 的头端或尾端。deque 的最大任务，便是在这些分段的定量连续空间上，维护其整体连续的假象，并提供随机存取的界面。避开了「重新配置、复制、释放」的轮回，代价则是复杂的迭代器架构。

受到分段连续线性空间的字面影响，我们可能以为 deque 的实作复杂度和 vector 相比虽不中亦不远矣，其实不然。主要因为，既曰分段连续线性空间，就必须有中央控制，而为了维护整体连续的假象，数据结构的设计及迭代器前进后退等动作都颇为繁琐。deque 的实作码份量远比 vector 或 list 都多得多。

deque 采用一块所谓的 *map*（注意，不是 STL 的 map 容器）做为主控。这里所谓 *map* 是一小块连续空间，其中每个元素（此处称为一个节点，node）都是指标，指向另一段（较大的）连续线性空间，称为缓冲区。缓冲区才是 deque 的储存空间主体。SGI STL 允许我们指定缓冲区大小，默认值 0 表示将使用 512 bytes 缓冲区。

```
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
  // 元素的指针的指针（pointer of pointer of T）
  typedef pointer* map_pointer;

protected:                  // Data members
```

```
    map_pointer map;                // 指向 map，map 是块连续空间，其内的每个元素
                                    // 都是一个指标（称为节点），指向一块缓冲区。
    size_type map_size; // map 内可容纳多少指标。
  ...
  };
```

把令人头皮发麻的各种型别定义（为了型别安全，那其实是有必要的）整理一下，我们便可发现，*map* 其实是一个 T**，也就是说它是一个指标，所指之物又是一个指标，指向型别为 T 的一块空间，如图 4-10。

稍后在 deque 的建构过程中，我会详细解释 *map* 的配置及维护。

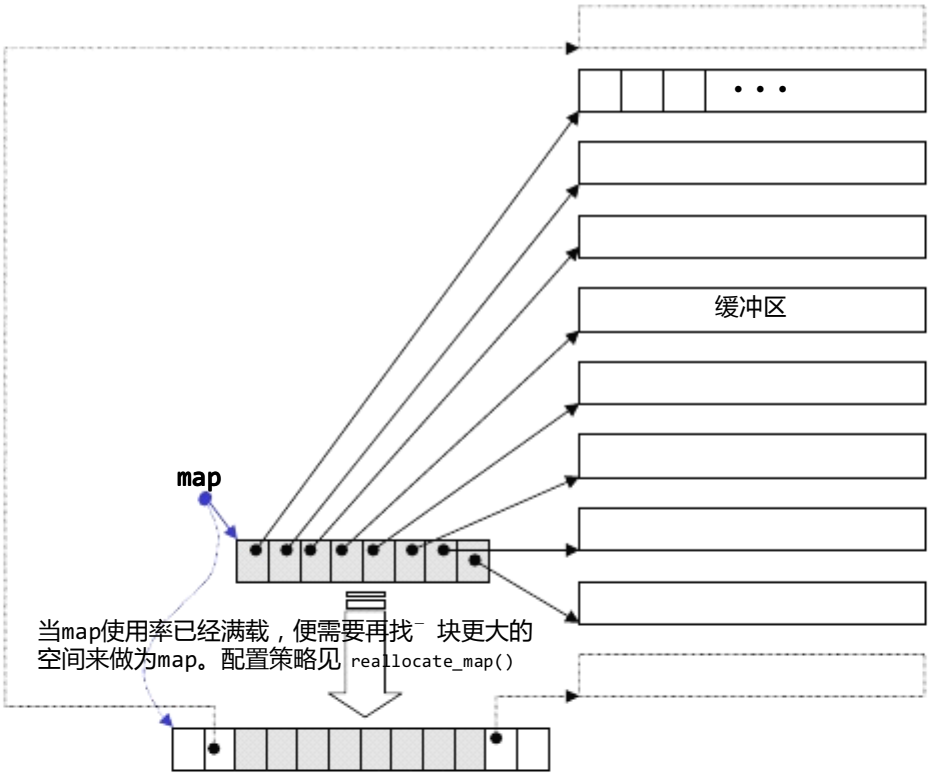


图 4-10 deque 的结构设计中，*map* 和 node-buffer（节点-缓冲区）的关系。

4.4.3 deque 的迭代器

deque 是分段连续空间。维护其「整体连续」假象的任务，着落在迭代器的 operator++ 和 operator-- 两个运算子身上。

让我们思考一下，deque 迭代器应该具备什么结构。首先，它必须能够指出分段连续空间（亦即缓冲区）在哪里，其次它必须能够判断自己是否已经处于其所在缓冲区的边缘，如果是，一旦前进或后退时就必须跳跃至下一个或上一个缓冲区。为了能够正确跳跃，deque 必须随时掌握管控中心（*map*）。下面这种实作方式符合需求：

```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator { // 未继承 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)); }

  // 未继承 std::iterator，所以必须自行撰写五个必要的迭代器相应型别（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;

  // 保持与容器的联结
  T* cur;       // 此迭代器所指之缓冲区中的现行（current）元素
  T* first;     // 此迭代器所指之缓冲区的头
  T* last;      // 此迭代器所指之缓冲区的尾（含备用空间）
  map_pointer node;             // 指向管控中心
...
};
```

其中用来决定缓冲区大小的函式 buffer_size()，呼叫 __deque_buf_size()，后者是个全域函式，定义如下：

```
// 如果 n     不为 0，传回 n，表示 buffer size 由使用者自定。
// 如果 n     为 0，表示 buffer size 使用默认值，那么
//     如果   sz（元素大小，sizeof(value_type)）小于 512，传回 512/sz，
//     如果   sz 不小于 512，传回 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));
}
```

图 4-11 是 deque 的中控器、缓冲区、迭代器的相互关系。

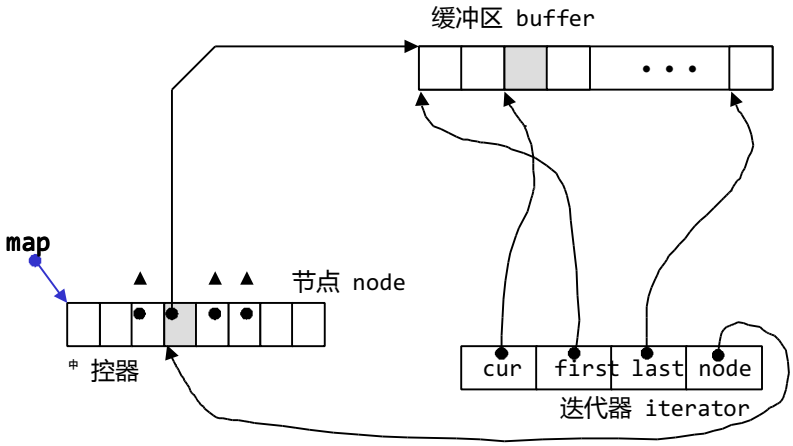


图 4-11 deque 的中控器、缓冲区、迭代器的相互关系

假设现在我们产生一个 deque<int>，并令其缓冲区大小为 32，于是每个缓冲区可容纳 32/sizeof(int)=4 个元素。经过某些操作之后，deque 拥有 20 个元素，那么其 begin() 和 end() 所传回的两个迭代器应该如图 4-12。这两个迭代器事实上一直保持在 deque 内，名为 start 和 finish，稍后在 deque 数据结构中便可看到）。

20 个元素需要 20/8 = 3 个缓冲区，所以 *map* 之内运用了三个节点。迭代器 start 内的 cur 指标当然指向缓冲区的第一个元素，迭代器 finish 内的 cur 指标当然指向缓冲区的最后元素（的下一位置）。注意，最后一个缓冲区尚有备用空间。稍后如果有新元素要安插于尾端，可直接拿此备用空间来使用。

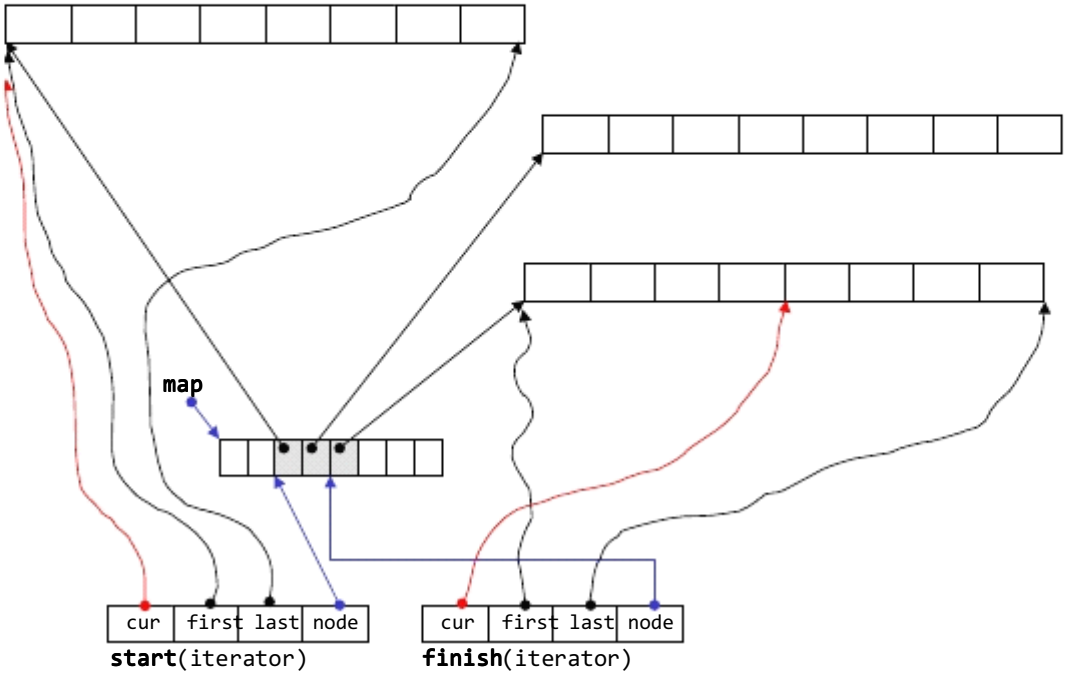


图 4-12 deque::begin() 传回迭代器 start，deque::end() 传回迭代器 finish。这两个迭代器都是 deque 的 data members。图中所示的这个 deque 拥有 20 个 int 元素，以 3 个缓冲区储存之。每个缓冲区 32 bytes，可储存 8 个 int 元素。map 大小为 8（起始值），目前用了 3 个节点。

下面是 deque 迭代器的几个关键行为。由于迭代器内对各种指标运算都做了多载化动作，所以各种指标运算如加、减、前进、后退...都不能直观视之。其中最重点的关键就是：一旦行进时遇到缓冲区边缘，要特别当心，视前进或后退而定，可能需要呼叫 set_node() 跳一个缓冲区：

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

// 以下各个多载化运算子是 __deque_iterator<> 成功运作的关键。

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);
}

// 参考 More Effective C++, item6: Distinguish between prefix and
// postfix forms of increment and decrement operators.
self& operator++() {
  ++cur;                          // 切换至下一个元素。
  if (cur == last) {              // 如果已达所在缓冲区的尾端，
    set_node(node + 1);           // 就切换至下一节点（亦即缓冲区）
    cur = first;                  //    的第一个元素。
  }
  return *this;

self operator++(int) {            // 后置式，标准写法
  self tmp = *this;
  ++*this;
  return tmp;
}
self& operator--() {
  if (cur == first) {             // 如果已达所在缓冲区的头端，
    set_node(node - 1);           // 就切换至前一节点（亦即缓冲区）
    cur = last;                   //    的最后一个元素。

  --cur;                          // 切换至前一个元素。
  return *this;
}
self operator--(int) {            // 后置式，标准写法
  self tmp = *this;
  --*this;
  return tmp;
}

// 以下实现随机存取。迭代器可以直接跳跃 n 个距离。
self& operator+=(difference_type n) {
  difference_type offset = n + (cur - first);
  if (offset >= 0 && offset < difference_type(buffer_size()))
    // 标的位置在同一缓冲区内
    cur += n;
  else {
    // 标的位置不在同一缓冲区内
    difference_type node_offset =
      offset > 0 ? offset / difference_type(buffer_size())
                 : -difference_type((-offset - 1) / buffer_size()) - 1;
    // 切换至正确的节点（亦即缓冲区）
    set_node(node + node_offset);
    // 切换至正确的元素
```

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

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

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

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

  // 以下实现随机存取。迭代器可以直接跳跃 n 个距离。
  reference operator[](difference_type n) const { return *(*this + n); }
  // 以上唤起 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 的数据结构

deque 除了维护一个先前说过的指向 *map* 的指标外，也维护 start, finish 两个迭代器，分别指向第一缓冲区的第一个元素和最后缓冲区的最后一个元素（的下一位置）。此外它当然也必须记住目前的 *map* 大小。因为一旦 *map* 所提供的节点不足，就必须重新配置更大的一块 *map*。

```
// 见 __deque_buf_size()。BufSize 默认值为 0 的唯一理由是为了闪避某些
// 编译器在处理常数算式（constant expressions）时的臭虫。
// 预设使用 alloc 为配置器。
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
  // 元素的指针的指针（pointer of pointer of T）
  typedef pointer* map_pointer;

protected:                       // Data members
  iterator start;             // 表现第一个节点。
  iterator finish;            // 表现最后一个节点。

  map_pointer map;            // 指向 map，map 是块连续空间，
                              // 其每个元素都是个指针，指向一个节点（缓冲区）。
  size_type map_size; // map 内有多少指标。
...
};
```

有了上述结构，以下数个机能便可轻易完成：

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

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

  reference front() { return *start; } // 唤起 __deque_iterator<>::operator*
  reference back() {
    iterator tmp = finish;
    --tmp;      // 唤起 __deque_iterator<>::operator--
    return *tmp;        // 唤起 __deque_iterator<>::operator*
    // 以上三行何不改为：return *(finish-1);
    // 因为 __deque_iterator<> 没有为 (finish-1) 定义运算子?!
  }

  // 下行最后有两个 ';' ，虽奇怪但合乎语法。
  size_type size() const { return finish - start;; }
  // 以上唤起 iterator::operator-
  size_type max_size() const { return size_type(-1); }
  bool empty() const { return finish == start; }
```

4.4.5 deque 的建构与内存管理 ctor, push_back, push_front

千头万绪该如何说起？以客端程序代码为引导，观察其所得结果并实证源码，是个良好的学习路径。下面是一个测试程序，我的观察重点在建构的方式以及大小的变化，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 只适用于 G++
  cout << "size=" << ideq.size() << endl;                    // size=20
  // 现在，应该已经建构了一个 deque，有 20 个 int 元素，初值皆为 9。
  // 缓冲区大小为 32bytes。

  // 为每一个元素设定新值。
  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 个元素，其值为 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 个元素，其值为 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 个元素，其值为 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 个元素，其值分别为 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

    // 搜寻数值为 99 的元素，并打印出来。
    deque<int,alloc,32>::iterator itr;
    itr = find(ideq.begin(), ideq.end(), 99);
    cout << *itr << endl;                                     // 99
    cout << *(itr.cur) << endl;                               // 99
}
```

deque 的缓冲区扩充动作相当琐碎繁杂，以下将以分解动作的方式一步一步图解说明。程序一开始宣告了一个 deque：

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

其缓冲区大小为 32 bytes，并令其保留 20 个元素空间，每个元素初值为 9。为了指定 deque 的第三个 template 参数（缓冲区大小），我们必须将前两个参数都指明出来（这是 C++ 语法规则），因此必须明确指定 alloc（第二章）为空间配置器。现在，deque 的情况如图 4-12（该图并未显示每个元素的初值为 9）。

deque 自行定义了两个专属的空间配置器：

```
protected:                          // Internal typedefs
  // 专属之空间配置器，每次配置一个元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;
  // 专属之空间配置器，每次配置一个指针大小
  typedef simple_alloc<pointer, Alloc> map_allocator;
```

并提供有一个 constructor 如下：

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

其内所呼叫的 fill_initialize() 负责产生并安排好 deque 的结构，并将元素

的初值设定妥当：

```
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 的结构都产生并安排好
  map_pointer cur;
  __STL_TRY {
    // 为每个节点的缓冲区设定初值
    for (cur = start.node; cur < finish.node; ++cur)
      uninitialized_fill(*cur, *cur + buffer_size(), value);
    // 最后一个节点的设定稍有不同（因为尾端可能有备用空间，不必设初值）
    uninitialized_fill(finish.first, finish.cur, value);
  }
  catch(...) {
     ...
  }
}
```

其中 create_map_and_nodes() 负责产生并安排好 deque 的结构：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements)
{
  // 需要节点数=(元素个数/每个缓冲区可容纳的元素个数)+1
  // 如果刚好整除，会多配一个节点。
  size_type num_nodes = num_elements / buffer_size() + 1;

  // 一个 map 要管理几个节点。最少 8 个，最多是 “所需节点数加 2”
  // （前后各预留一个，扩充时可用）。
  map_size = max(initial_map_size(), num_nodes + 2);
  map = map_allocator::allocate(map_size);
  // 以上配置出一个 “具有 map_size 个节点” 的 map。

  // 以下令 nstart 和 nfinish 指向 map 所拥有之全部节点的最中央区段。
  // 保持在最中央，可使头尾两端的扩充能量一样大。每个节点可对应一个缓冲区。
  map_pointer nstart = map + (map_size - num_nodes) / 2;
  map_pointer nfinish = nstart + num_nodes - 1;

  map_pointer cur;
  __STL_TRY {
    // 为 map 内的每个现用节点配置缓冲区。所有缓冲区加起来就是 deque 的
    // 可用空间（最后一个缓冲区可能留有一些余裕）。
    for (cur = nstart; cur <= nfinish; ++cur)
      *cur = allocate_node();
  }
  catch(...) {
    // "commit or rollback" 语意：若非全部成功，就一个不留。
    ...
```

```
    }

    // 为 deque 内的两个迭代器 start 和 end 设定正确内容。
    start.set_node(nstart);
    finish.set_node(nfinish);
                                        // first, cur 都是 public
    start.cur = start.first;
    finish.cur = finish.first + num_elements % buffer_size();
    // 前面说过，如果刚好整除，会多配一个节点。
    // 此时即令 cur 指向这多配的一个节点（所对映之缓冲区）的起头处。
}
```

接下来范例程序以注标运算子为每个元素重新设值，然后在尾端安插三个新元素：

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

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

由于此时最后一个缓冲区仍有 4 个备用元素空间，所以不会引起缓冲区的再配置。此时的 deque 状态如图 4-13。

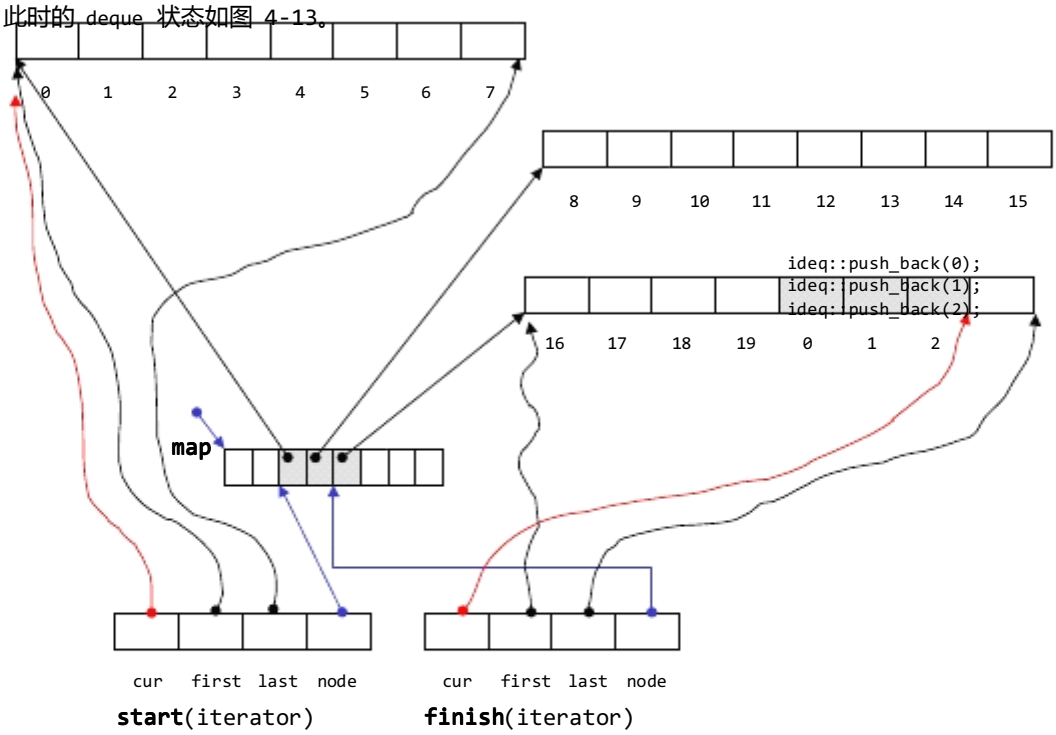


图 4-13 延续图 4-12 的状态，将每个元素重新设值，并在尾端新增 3 个元素。

以下是 push_back() 函式内容：

```
public:                          // push_* and pop_*
  void push_back(const value_type& t) {
    if (finish.cur != finish.last - 1)
      // 最后缓冲区尚有一个以上的备用空间
      construct(finish.cur, t); // 直接在备用空间上建构元素
      ++finish.cur;             // 调整最后缓冲区的使用状态
    }
    else // 最后缓冲区已无（或只剩一个）元素备用空间。
      push_back_aux(t);
  }
```

现在，如果再新增加一个新元素于尾端：

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

由于尾端只剩一个元素备用空间，于是 push_back() 呼叫 push_back_aux()，先配置一整块新的缓冲区，再设妥新元素内容，然后更改迭代器 finish 的状态：

```
// 只有当 finish.cur == finish.last – 1 时才会被呼叫。
// 也就是说只有当最后一个缓冲区只剩一个备用元素空间时才会被呼叫。
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();                  // 若符合某种条件则必须重换一个 map
  *(finish.node + 1) = allocate_node();                 // 配置一个新节点（缓冲区）
  __STL_TRY {
    construct(finish.cur, t_copy);                      // 针对标的元素设值
    finish.set_node(finish.node + 1);                   // 改变 finish，令其指向新节点
    finish.cur = finish.first;                          // 设定 finish 的状态
  }
  __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}
```

现在，deque 的状态如图 4-14。

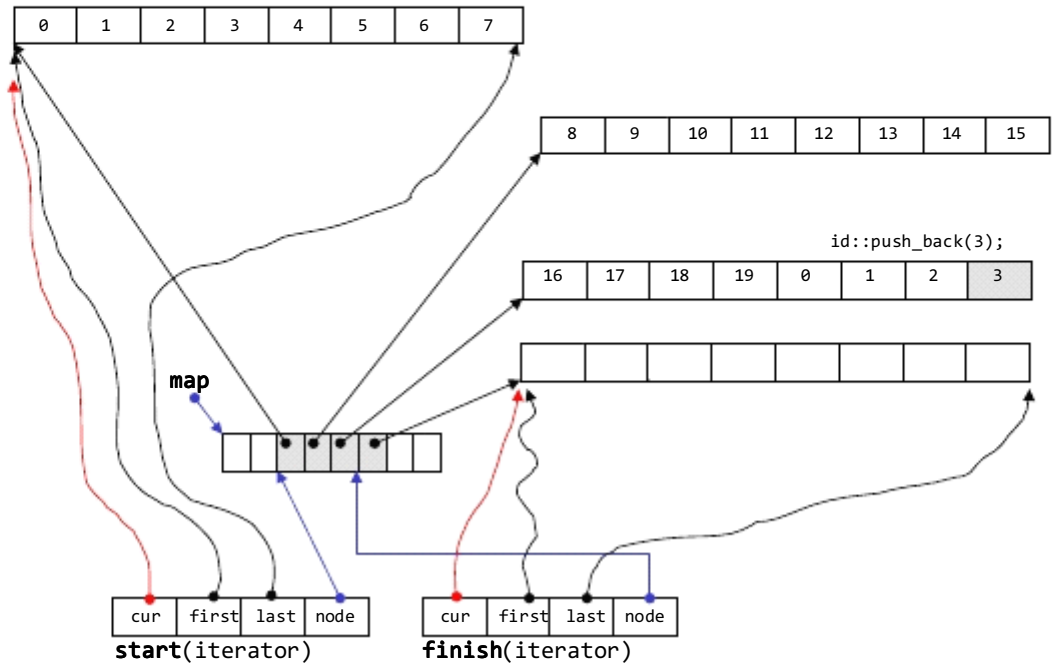


图 4-14 延续图 4-13 的状态，在尾端再加一个元素，于是引发新缓冲区的配置，同时也造成迭代器 finish 的状态改变。*map* 大小为 8（初始值），目前用了 4 个节点。

接下来范例程序在 deque 的前端安插一个新元素：

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

push_front() 函式动作如下：

```
public:                          // push_* and pop_*
  void push_front(const value_type& t) {
    if (start.cur != start.first) {                  // 第一缓冲区尚有备用空间
      construct(start.cur - 1, t); // 直接在备用空间上建构元素
      --start.cur;               // 调整第一缓冲区的使用状态
    }
    else // 第一缓冲区已无备用空间
      push_front_aux(t);
  }
```

由于目前状态下，第一缓冲区并无备用空间，所以呼叫 push_front_aux()：

```
// 只有当 start.cur == start.first 时才会被呼叫。
// 也就是说只有当第一个缓冲区没有任何备用元素时才会被呼叫。
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();                  // 若符合某种条件则必须重换一个 map
  *(start.node - 1) = allocate_node();                  // 配置一个新节点（缓冲区）
  __STL_TRY {
    start.set_node(start.node - 1);                     // 改变 start，令其指向新节点
    start.cur = start.last - 1;                         // 设定 start 的状态
    construct(start.cur, t_copy);                       // 针对标的元素设值
  }
  catch(...) {
    // "commit or rollback" 语意：若非全部成功，就一个不留。
    start.set_node(start.node + 1);
    start.cur = start.first;
    deallocate_node(*(start.node - 1));
    throw;
  }
}
```

此函式一开始即呼叫 reserve_map_at_front()，后者用来判断是否需要扩充 *map*，如有需要就付诸行动。稍后我会呈现 reserve_map_at_front() 的函式内容。目前的状态不需要重新整治 *map*，所以后继流程便配置了一块新缓冲区并直接将节点安置于现有的 *map* 上，然后设定新元素，然后改变迭代器 start 的状态，如图 4-15。

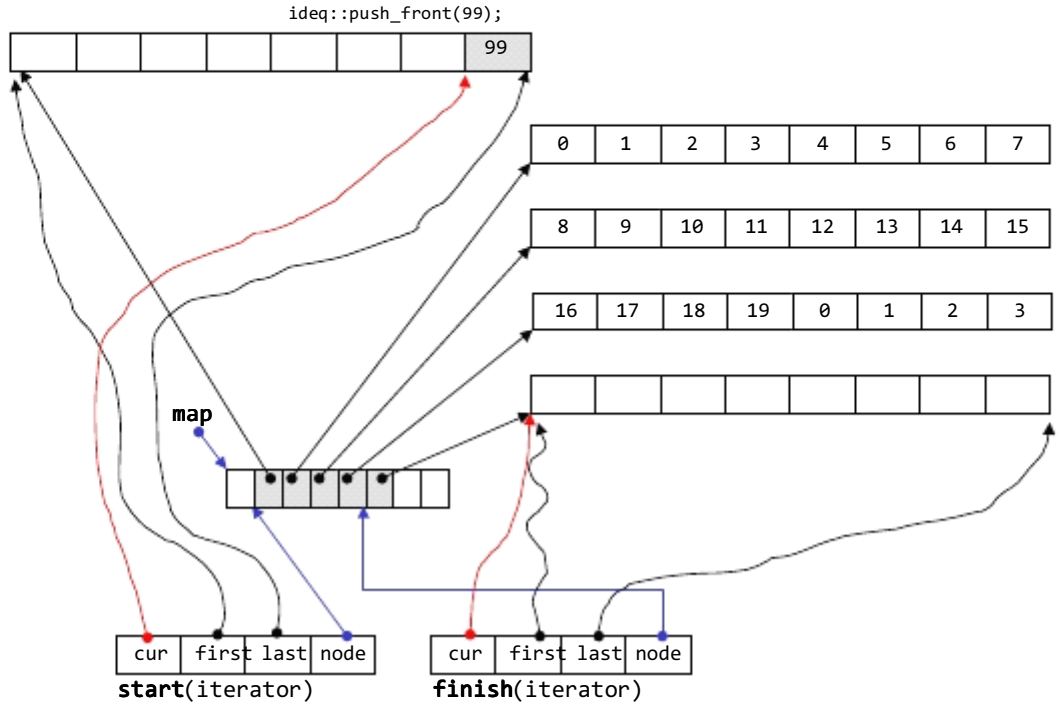


图 4-15 延续图 4-14 的状态，在最前端加上一个元素。引发新缓冲区的配置，同时也造成迭代器 start 状态改变。*map* 大小为 8（初始值），目前用掉 5 个节点。

接下来范例程序又在 deque 的最前端安插两个新元素：

```
ideq.push_front(98);
ideq.push_front(97);
```

这一次，由于第一缓冲区有备用空间，push_front() 可以直接在备用空间上建构新元素，如图 4-16。

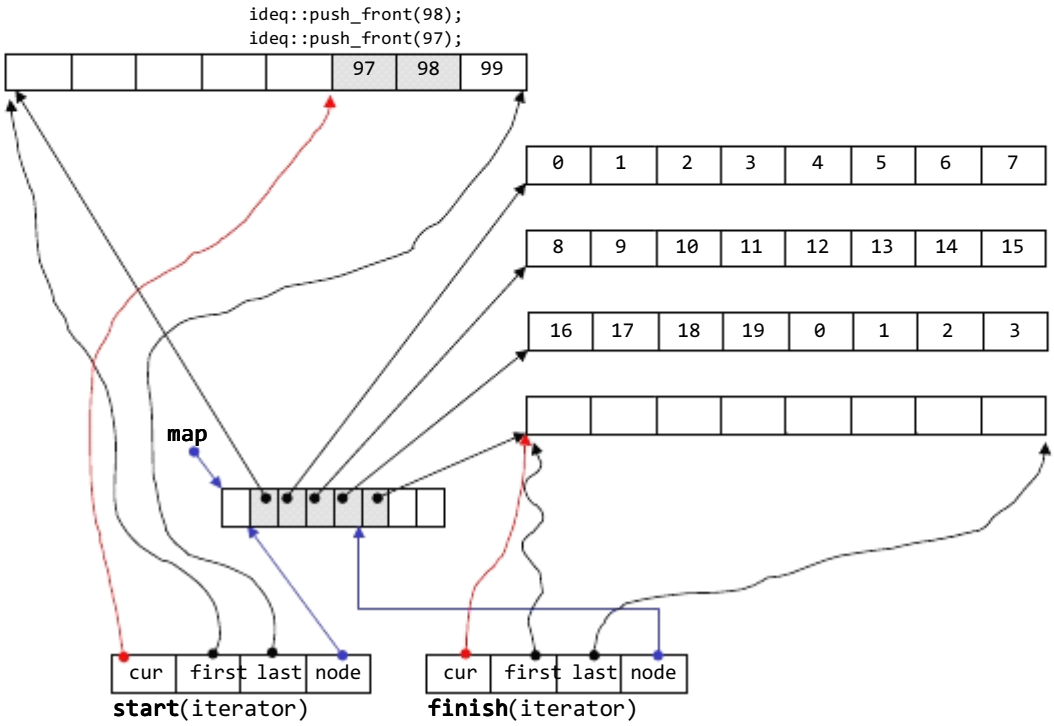


图 4-16　延续图 4-15 的状态，在最前端再加两个元素。由于第一缓冲区尚有备用空间，因此直接取用备用空间来建构新元素即可。

图 4-12 至图 4-16 的连环图解，已经充份展示了 deque 容器的空间运用策略。让我们回头看看一个悬而未解的问题：什么时候 *map* 需要重新整治？这个问题的判断由 reserve_map_at_back() 和 reserve_map_at_front() 进行，实际动作则由 reallocate_map() 执行：

```
void reserve_map_at_back (size_type nodes_to_add = 1) {
  if (nodes_to_add + 1 > map_size - (finish.node - map))
    // 如果 map 尾端的节点备用空间不足
    // 符合以上条件则必须重换一个 map（配置更大的，拷贝原来的，释放原来的）
    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 前端的节点备用空间不足
```

```
    // 符合以上条件则必须重换一个 map（配置更大的，拷贝原来的，释放原来的）
    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;
    // 配置一块空间，准备给新 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 内容拷贝过来。
    copy(start.node, finish.node + 1, new_nstart);
    // 释放原 map
    map_allocator::deallocate(map, map_size);
    // 设定新 map 的起始地址与大小
    map = new_map;
    map_size = new_map_size;
  }

  // 重新设定迭代器 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 所提供的元素操作动作很多，无法在有限的篇幅中一一讲解 — 其实也没有这种必要。以下我只挑选几个 member functions 做为示范说明。

前述测试程序曾经以泛型算法 find() 寻找 deque 的某个元素：

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

当 find() 动作完成，迭代器 itr 状态如图 4-17 所示。下面这两个动作输出相同的结果，印证我们对 deque 迭代器的认识。

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

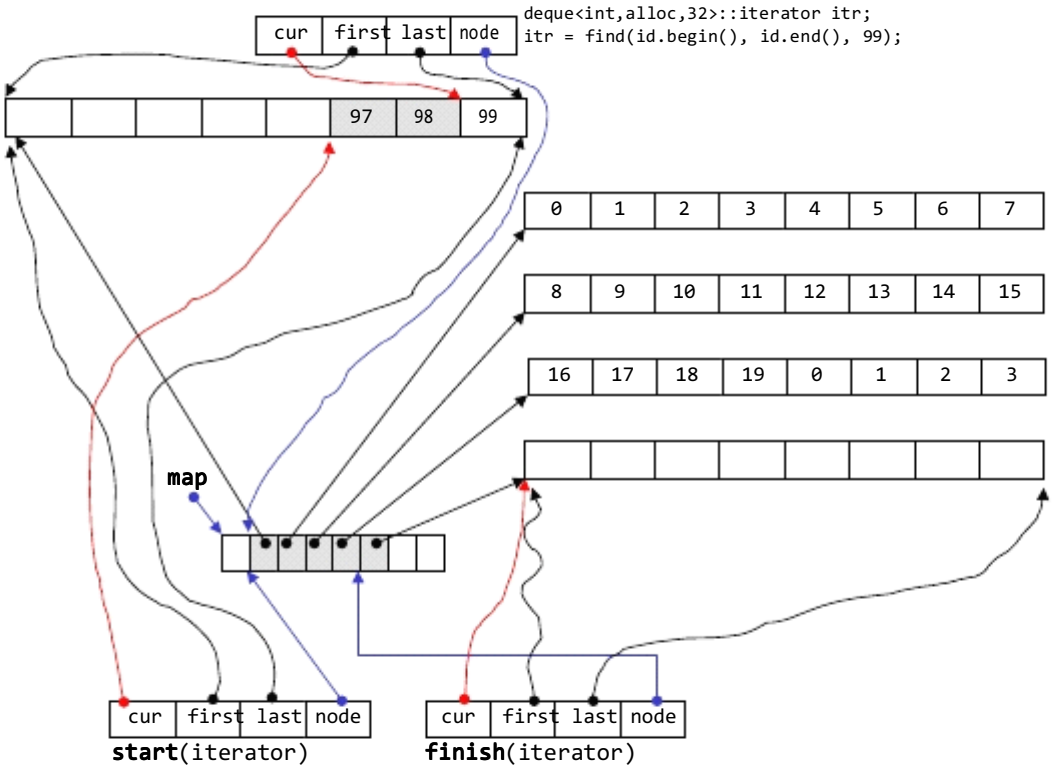


图 4-17 延续图 4-16 的状态，以 find() 寻找数值为 99 的元素。此函式将传回一个迭代器，指向第一个符合条件的元素。注意，该迭代器的四个字段都必须有正确的设定。

前一节已经展示过 push_back() 和 push_front() 的实作内容，现在我举对应的 pop_back() 和 pop_front() 为例。所谓 pop，是将元素拿掉。无论从 deque 的最前端或最尾端取元素，都需考虑在某种条件下，将缓冲区释放掉：

```
  void pop_back() {
    if (finish.cur != finish.first) {
      // 最后缓冲区有一个（或更多）元素
      --finish.cur;             // 调整指针，相当于排除了最后元素
      destroy(finish.cur); // 将最后元素解构
    }
    else
      // 最后缓冲区没有任何元素
      pop_back_aux ();          // 这里将进行缓冲区的释放工作
  }

// 只有当 finish.cur == finish.first 时才会被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_back_aux() {
  deallocate_node(finish.first);                  // 释放最后一个缓冲区
  finish.set_node(finish.node - 1); // 调整 finish 的状态，使指向
  finish.cur = finish.last - 1;                   // 上一个缓冲区的最后一个元素
  destroy(finish.cur);                            // 将该元素解构。
}


  void pop_front() {
    if (start.cur != start.last - 1) {
      // 第一缓冲区有一个（或更多）元素
      destroy(start.cur);           // 将第一元素解构
      ++start.cur;                  // 调整指针，相当于排除了第一元素
    }
    else
      // 第一缓冲区仅有一个元素
      pop_front_aux();              // 这里将进行缓冲区的释放工作
  }

// 只有当 start.cur == start.last - 1 时才会被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
  destroy(start.cur);                             // 将第一缓冲区的第一个元素解构。
  deallocate_node(start.first);                   // 释放第一缓冲区。
  start.set_node(start.node + 1);                 // 调整 start 的状态，使指向
  start.cur = start.first;                        // 下一个缓冲区的第一个元素。
}
```

下面这个例子是 clear()，用来清除整个 deque。请注意，deque 的最初状态（无任何元素时）保有一个缓冲区，因此 clear() 完成之后回复初始状态，也一样要保留一个缓冲区：

```
// 注意，最终需要保留一个缓冲区。这是 deque 的策略，也是 deque 的初始状态。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
  // 以下针对头尾以外的每一个缓冲区（它们一定都是饱满的）
  for (map_pointer node = start.node + 1; node < finish.node; ++node) {
    // 将缓冲区内的所有元素解构。注意，呼叫的是 destroy() 第二版本，见 2.2.3 节
    destroy(*node, *node + buffer_size());
    // 释放缓冲区内存
    data_allocator::deallocate(*node, buffer_size());
  }

  if (start.node != finish.node) {                    // 至少有头尾两个缓冲区
    destroy(start.cur, start.last);                   // 将头缓冲区的目前所有元素解构
    destroy(finish.first, finish.cur); // 将尾缓冲区的目前所有元素解构
    // 以下释放尾缓冲区。注意，头缓冲区保留。
    data_allocator::deallocate(finish.first, buffer_size());
  }
  else      // 只有一个缓冲区
    destroy(start.cur, finish.cur);                   // 将此唯一缓冲区内的所有元素解构
    // 注意，并不释放缓冲区空间。这唯一的缓冲区将保留。

  finish = start; // 调整状态
}
```

下面这个例子是 erase()，用来清除某个元素：

```
// 清除 pos 所指的元素。pos 为清除点。
iterator erase(iterator pos) {
  iterator next = pos;
  ++next;
  difference_type index = pos - start; // 清除点之前的元素个数
  if (index < (size() >> 1)) {                        // 如果清除点之前的元素比较少，
    copy_backward(start, pos, next);                  // 就搬移清除点之前的元素
    pop_front();                    // 搬移完毕，最前一个元素赘余，去除之
  }
  else {                            // 清除点之后的元素比较少，
    copy(next, finish, pos);        // 就搬移清除点之后的元素
    pop_back();                     // 搬移完毕，最后一个元素赘余，去除之
  }
  return start + index;
}
```

下面这个例子是 erase()，用来清除 [first,last) 区间内的所有元素：

```
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) { // 如果清除区间就是整个 deque
    clear();                              // 直接呼叫 clear() 即可
    return finish;
  }
  else {
    difference_type n = last - first;                    // 清除区间的长度
    difference_type elems_before = first - start;        // 清除区间前方的元素个数
    if (elems_before < (size() - n) / 2) {               // 如果前方的元素比较少，
      copy_backward(start, first, last);                 // 向后搬移前方元素（覆盖清除区间）
      iterator new_start = start + n;                    // 标记 deque 的新起点
      destroy(start, new_start);                         // 搬移完毕，将赘余的元素解构
      // 以下将赘余的缓冲区释放
      for (map_pointer cur = start.node; cur < new_start.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      start = new_start;              // 设定 deque 的新起点

    else {       // 如果清除区间后方的元素比较少
      copy(last, finish, first);                         // 向前搬移后方元素（覆盖清除区间）
      iterator new_finish = finish - n;                  // 标记 deque 的新尾点
      destroy(new_finish, finish);                       // 搬移完毕，将赘余的元素解构
      // 以下将赘余的缓冲区释放
      for (map_pointer cur = new_finish.node + 1; cur <= finish.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      finish = new_finish; // 设定 deque 的新尾点
    }
    return start + elems_before;
  }
}
```

本节要说明的最后一个例子是 insert。deque 为这个功能提供了许多版本，最基础最重要的是以下版本，允许在某个点（之前）安插一个元素，并设定其值。

```
// 在 position 处安插一个元素，其值为 x
iterator insert(iterator position, const value_type& x) {
  if (position.cur == start.cur) { // 如果安插点是 deque 最前端
    push_front(x);                 // 交给 push_front 去做
    return start;
  }
  else if (position.cur == finish.cur) { // 如果安插点是 deque 最尾端
    push_back(x);                        // 交给 push_back 去做
    iterator tmp = finish;
    --tmp;
```

```
    return tmp;
  }
  else {
    return insert_aux(position, x);                    // 交给 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;                 // 安插点之前的元素个数
  value_type x_copy = x;
  if (index < size() / 2) {                            // 如果安插点之前的元素个数比较少
    push_front(front());                               // 在最前端加入与第一元素同值的元素。
    iterator front1 = start;                           // 以下标示记号，然后进行元素搬移...
    ++front1;
    iterator front2 = front1;
    ++front2;
    pos = start + index;
    iterator pos1 = pos;
    ++pos1;
    copy(front2, pos1, front1);                        // 元素搬移
  }
  else {                                               // 安插点之后的元素个数比较少
    push_back(back());                                 // 在最尾端加入与最后元素同值的元素。
    iterator back1 = finish;                           // 以下标示记号，然后进行元素搬移...
    --back1;
    iterator back2 = back1;
    --back2;
    pos = start + index;
    copy_backward(pos, back2, back1);                  // 元素搬移
  }
  *pos = x_copy; // 在安插点上设定新值
  return pos;
}
```

4.5 stack

4.5. stack 概述
1

stack 是一种先进后出（First In Last Out，FILO）的数据结构。它只有一个出口，型式如图 4-18。stack 允许新增元素、移除元素、取得最顶端元素。但除了最顶端外，没有任何其它方法可以存取 stack 的其它元素。换言之 stack 不允许有走访行为。

将元素推入 stack 的动作称为 *push*，将元素推出 stack 的动作称为 *pop*。

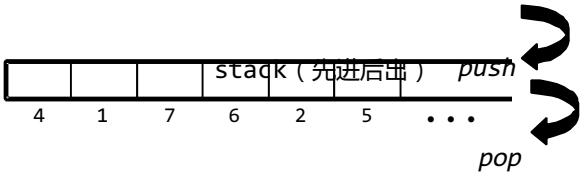


图 4-18 stack 的结构

4.5.2 stack 定义式完整列表

以某种既有容器做为底部结构，将其接口改变，使符合「先进后出」的特性，形成一个 stack，是很容易做到的。deque 是双向开口的数据结构，若以 deque 为底部结构并封闭其头端开口，便轻而易举[1] 形成了一个 stack。因此，SGI STL 便以 deque 做为预设情况[2] 的 stack 底部结构，stack 的实作因而非常简单，源码十分简短，本处完整列出。

由于 stack 系以底部容器完成其所有工作，而具有这种「修改某物接口，形成另一种风貌」之性质者，称为 adapter（配接器），因此 STL stack 往往不被归类为 container（容器），而被归类为 container adapter。

```
template <class T, class Sequence = deque<T> >
class stack {
  // 以下 的 __STL_NULL_TMPL_ARGS 会开展为 <>，见 1.9.1 节
  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;          // 底层容器
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 是两头可进出，stack 是末端进，末端出（所以后进者先出）。
  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 没有迭代器

stack 所有元素的进出都必须符合「先进后出」的条件，只有 stack 顶端的元素，才有机会被外界取用。stack 不提供走访功能，也不提供迭代器。

4.5.4 以 list 做为 stack 的底层容器

除了 deque 之外，list 也是双向开口的数据结构。上述 stack 源码中使用的底层容器的函式有 empty, size, back, push_back, pop_back，凡此种种 list 都具备。因此若以 list 为底部结构并封闭其头端开口，一样能够轻易形成一个 stack。下面是作法示范。

```
// 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 是一种先进先出（First In First Out，FIFO）的数据结构。它有两个出口，型式如图 4-19。queue 允许新增元素、移除元素、从最底端加入元素、取得最顶端元素。但除了最底端可以加入、最顶端可以取出，没有任何其它方法可以存取 queue 的其它元素。换言之 queue 不允许有走访行为。

将元素推入 queue 的动作称为 *push*，将元素推出 queue 的动作称为 *pop*。

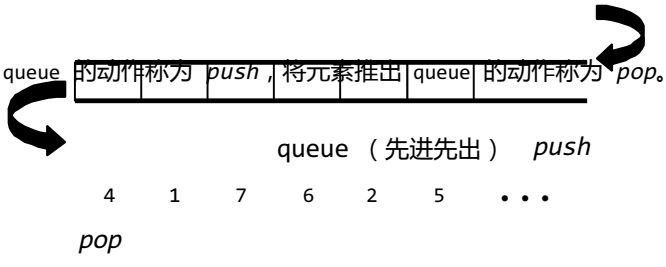


图 4-19 queue 的结构

4.6.2 queue 定义式完整列表

以某种既有容器为底部结构，将其接口改变，使符合「先进先出」的特性，形成一个 queue，是很容易做到的。deque 是双向开口的数据结构，若以 deque 为底部结构并封闭其底端的出口和前端的入口，便轻而易举[地]形成了一个 queue。因此，SGI STL 便以 deque 做为预设情况[下]的 queue 底部结构，queue 的实作因而非常简单，源码十分简短，本处完整列出。

由于 queue 系以底部容器完成其所有工作，而具有这种「修改某物接口，形成另一种风貌」之性质者，称为 adapter（配接器），因此 STL queue 往往不被归类为 container（容器），而被归类为 container adapter。

```
template <class T, class Sequence = deque<T> >
class queue {
  // 以下的 __STL_NULL_TMPL_ARGS 会开展为 <>，见 1.9.1 节
  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;          // 底层容器
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 是两头可进出，queue 是末端进，前端出（所以先进者先出）。
  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 没有迭代器

queue 所有元素的进出都必须符合「先进先出」的条件，只有 queue 顶端的元素，才有机会被外界取用。queue 不提供走访功能，也不提供迭代器。

4.6.4 以 list 做为 queue 的底层容器

除了 deque 之外，list 也是双向开口的数据结构。上述 queue 源码中使用的底层容器的函式有 empty, size, back, push_back, pop_back，凡此种种 list 都具备。因此若以 list 为底部结构并封闭其头端开口，一样能够轻易形成一个 queue。下面是作法示范。

```
// 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（隐性表述，implicit representation）

4.7. heap 概述
1

heap 并不归属于 STL 容器组件，它是个幕后英雄，扮演 priority queue（4.8 节）的推手。顾名思义，priority queue 允许使用者以任何次序将任何元素推入容器内，但取出时一定是从优先权最高（也就是数值最高）之元素开始取。binary max heap 正是具有这样的特性，适合做为 priority queue 的底层机制。

让我们做点分析。如果使用 4.3 节的 list 做为 priority queue 的底层机制，元素安插动作可享常数时间。但是要找到 list 中的极值，却需要对整个 list 进行线性扫描。我们也可以改个作法，让元素安插前先经过排序这一关，使得 list 的元素值总是由小到大（或由大到小），但这么一来，收之东隅却失之桑榆：虽然取得极值以及元素删除动作达到最高效率，元素的安插却只有线性表现。

比较麻辣的作法是以 binary search tree（如 5.1 节的 RB-tree）做为 priority queue 的底层机制。这么一来元素的安插和极值的取得就有 *O(logN)* 的表现。但杀鸡用牛刀，未免小题大作，一来 binary search tree 的输入需要足够的随机性，二来 binary search tree 并不容易实作。priority queue 的复杂度，最好介于 queue 和 binary search tree 之间，才算适得其所。binary heap 便是这种条件下的适当候选人。

所谓 binary heap 就是一种 complete binary tree（完全二元树）[2]，也就是说，整棵 binary tree 除了最底层的叶节点(s) 之外，是填满的，而最底层的叶节点(s) 由左至右又不得有空隙。图 4-20 是一个 complete binary tree。

关于 tree 的种种，5.1 节会有更多介绍。

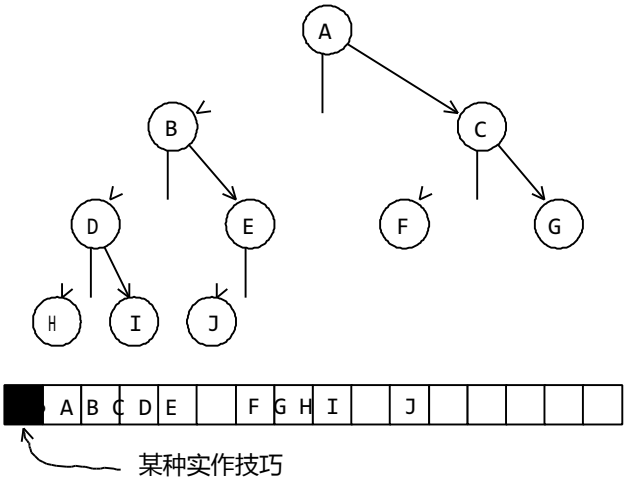


图 4-20 一个完全二元树（complete binary tree），及其 array 表述式

complete binary tree 整棵树内没有任何节点漏洞，这带来一个极大好处：我们可以利用 array 来储存所有节点。假设动用一个小技巧[3]，将 array 的 #0 元素保留（或设为无限大值或无限小值），那么当 complete binary tree 中的某个节点位于 array 的i 处，其左子节点必位于 array 的2i 处，其右子节点必位于 array 的 2i+1 处，其父节点必位于「i/2」处（此处的「」权且代表高斯符号，取其整数）。通过这么简单的位置规则，array 可以轻易实作出 complete binary tree。这种以 array 表述 tree 的方式，我们称为隐式表述法（implicit representation）。

这么一来，我们需要的工具就很简单了：一个 array 和一组 heap 算法（用来安插元素、删除元素、取极值、将某一整组数据排列成一个 heap）。array 的缺点是无法动态改变大小，而 heap 却需要这项功能，因此以 vector（4.2 节）代替 array 是更好的选择。

根据元素排列方式，heap 可分为 max-heap 和 min-heap 两种，前者每个节点的键值（*key*）都大于或等于其子节点键值，后者的每个节点键值（*key*）都小于或等

---

[3] SGI STL 提供的 heap 并未使用此一小技巧。计算左右子节点以及父节点的方式，因而略有不同。详见稍后的源码及解说。

于其子节点键值。因此，max-heap 的最大值在根节点，并总是位于底层 array 或 vector 的起头处；min-heap 的最小值在根节点，亦总是位于底层 array 或 vector 的起头处。STL 供应的是 max-heap，因此以下我说 heap 时，指的是 max-heap。

4.7.2 heap 算法

push_heap 演算法

图 4-21 是 push_heap 算法的实际操演情况。为了满足 complete binary tree 的条件，新加入的元素一定要放在最下一层做为叶节点，并填补在由左至右的第一个空格，也就是把新元素安插在底层 vector 的 end() 处。

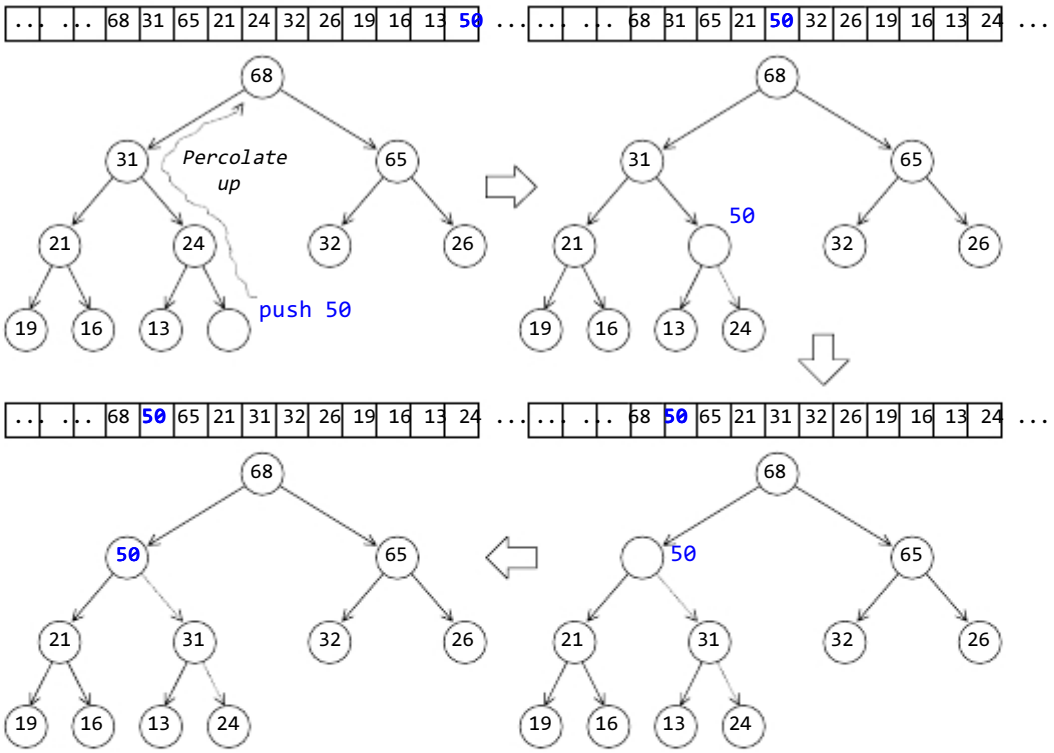


图 4-21 push_heap 算法

新元素是否适合于其现有位置呢？为满足 max-heap 的条件（每个节点的键值都大于或等于其子节点键值），我们执行一个所谓的 percolate up（上溯）程序：将新节点拿来与其父节点比较，如果其键值（*key*）比父节点大，就父子对换位置。如此一直上溯，直到不需对换或直到根节点为止。

下面便是 push_heap 算法的实作细节。此函式接受两个迭代器，用来表现一个 heap 底部容器（vector）的头尾，新元素并且已经安插到底部容器的最尾端。如果不符合这两个条件，push_heap 的执行结果未可预期。

```
template <class RandomAccessIterator>
inline void push_heap(RandomAccessIterator first,
                      RandomAccessIterator last) {
  // 注意，此函式被呼叫时，新元素应已置于底部容器的最尾端。
  __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)));
  // 以上系根据 implicit representation heap 的结构特性：新值必置于底部
  // 容器的最尾端，此即第一个洞号：(last-first)-1。
}

// 以下这组 push_back()不允许指定「大小比较标准」
template <class RandomAccessIterator, class Distance, class T>
void __push_heap(RandomAccessIterator first, Distance holeIndex,
                 Distance topIndex, T value) {
  Distance parent = (holeIndex - 1) / 2; // 找出父节点
  while (holeIndex > topIndex && *(first + parent) < value) {
    // 当尚未到达顶端，且父节点小于新值（于是不符合 heap 的次序特性）
    // 由于以上使用 operator<，可知 STL heap 是一种 max-heap（大者为父）。
    *(first + holeIndex) = *(first + parent); // 令洞值为父值
    holeIndex = parent; // percolate up：调整洞号，向上提升至父节点。
    parent = (holeIndex - 1) / 2;                    // 新洞的父节点
  }    // 持续至顶端，或满足 heap 的次序特性为止。
  *(first + holeIndex) = value; // 令洞值为新值，完成安插动作。
}
```

pop_heap 算法

图 4-22 是 pop_heap 算法的实际操演情况。既然身为 max-heap，最大值必然在根节点。pop 动作取走根节点（其实是移至底部容器 vector 的最后一个元素）之后，为了满足 complete binary tree 的条件，必须将最下一层最右边的叶节点拿掉，现在我们的任务是为这个被拿掉的节点找一个适当的位置。

为满足 max-heap 的条件（每个节点的键值都大于或等于其子节点键值），我们执行一个所谓的 percolate down（下放）程序：将根节点（最大值被取走后，形成一个「洞」）填入上述那个失去生存空间的叶节点值，再将它拿来和其两个子节点比较键值（*key*），并与较大子节点对调位置。如此一直下放，直到这个「洞」的键值大于左右两个子节点，或直到下放至叶节点为止。

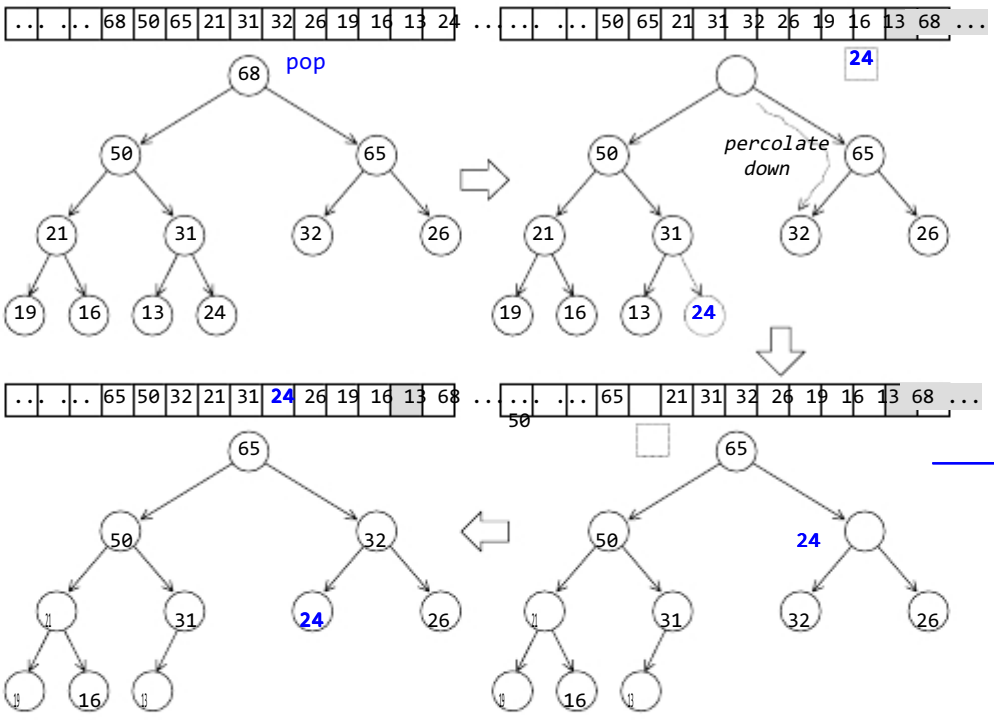


图 4-22 pop_heap 算法

下面便是 pop_heap 算法的实作细节。此函式接受两个迭代器，用来表现一个 heap 底部容器（vector）的头尾。如果不符合这个条件，pop_heap 的执行结果未可预期。

```
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，根据 implicit representation heap 的次序特性，pop 动作的结果
  // 应为底部容器的第一个元素。因此，首先设定欲调整值为尾值，然后将首值调至
  // 尾节点（所以以上将迭代器 result 设为 last-1）。然后重整 [first, last-1)，
  // 使之重新成一个合格的 heap。
}

// 以下这组 __pop_heap() 不允许指定「大小比较标准」
template <class RandomAccessIterator, class T, class Distance>
inline void __pop_heap(RandomAccessIterator first,
                       RandomAccessIterator last,
                       RandomAccessIterator result,
                       T value, Distance*) {
  *result = *first; // 设定尾值为首值，于是尾值即为欲求结果，
                    // 可由客端稍后再以底层容器之 pop_back() 取出尾值。
  __adjust_heap(first, Distance(0), Distance(last - first), value);
  // 以上欲重新调整 heap，洞号为 0（亦即树根处），欲调整值为 value（原尾值）。
}

// 以下这个 __adjust_heap() 不允许指定「大小比较标准」
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;                    // 洞节点之右子节点
  while (secondChild < len) {
    // 比较洞节点之左右两个子值，然后以 secondChild 代表较大子节点。
    if (*(first + secondChild) < *(first + (secondChild - 1)))
      secondChild--;
    // Percolate down：令较大子值为洞值，再令洞号下移至较大子节点处。
    *(first + holeIndex) = *(first + secondChild);
    holeIndex = secondChild;
    // 找出新洞节点的右子节点
```

```
    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);
}
```

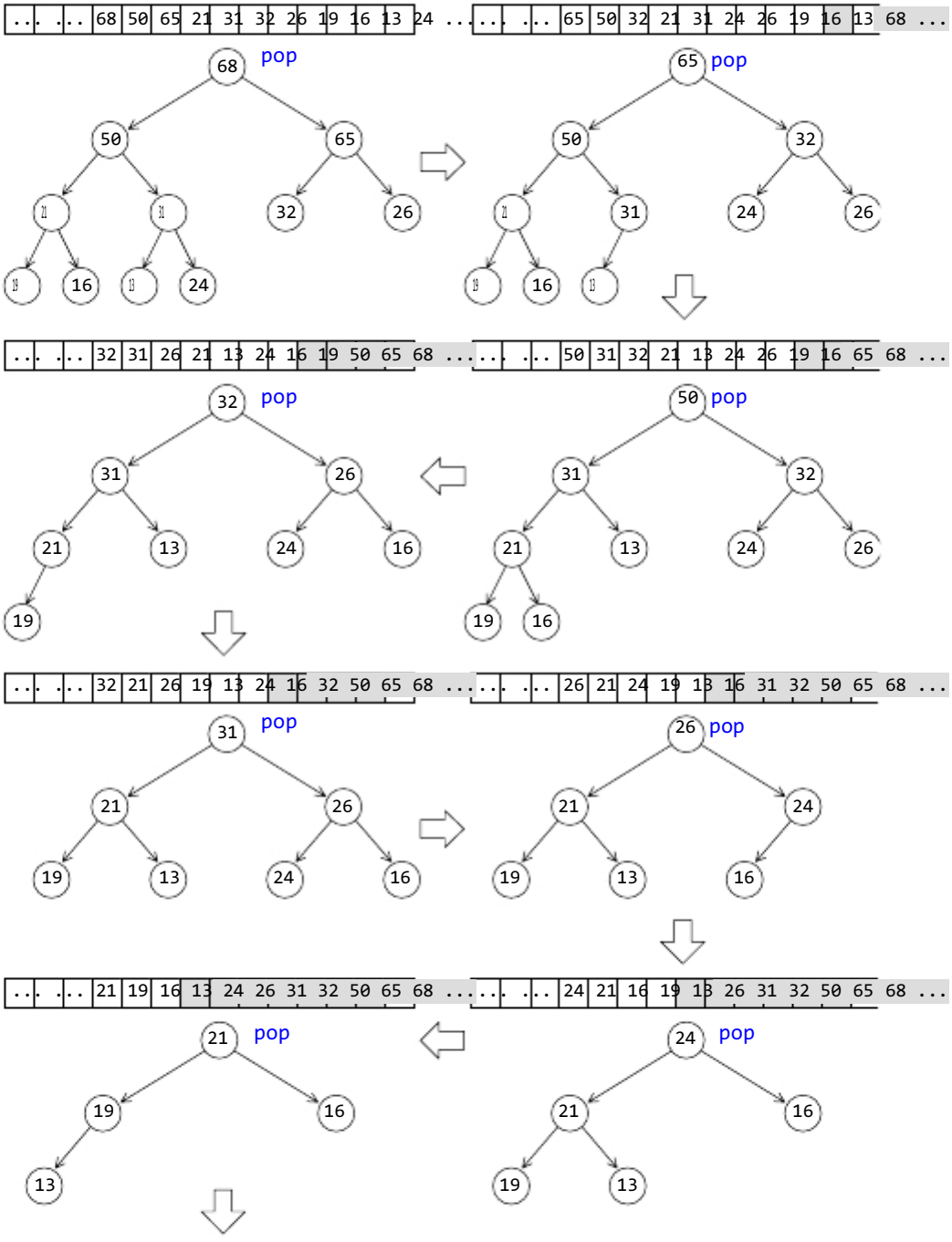
注意，pop_heap 之後，最大元素只是被置放於底部容器的最尾端，尚未被取走。如果要取其值，可使用底部容器（vector）所提供的 back() 操作函式。如果要移除它，可使用底部容器（vector）所提供的 pop_back() 操作函式。

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，每次將操作範圍從后向前縮減一個元素（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 算法的實作細节。此函式接受两个迭代器，用来表现一个 heap 底部容器（vector）的头尾。如果不符合这个条件，sort_heap 的执行结果未可预期。注意，排序过后，原来的 heap 就不再是个合法的 heap 了。

```
// 以下这个 sort_heap() 不允许指定「大小比较标准」
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last) {
  // 以下，每执行一次 pop_heap()，极值（在 STL heap 中为极大值）即被放在尾端。
  // 扣除尾端再执行一次 pop_heap()，次极值又被放在新尾端。一直下去，最后即得
  // 排序结果。
  while (last - first > 1)
    pop_heap(first, last--); // 每执行 pop_heap() 一次，操作范围即退缩一格。
}
```



续下页

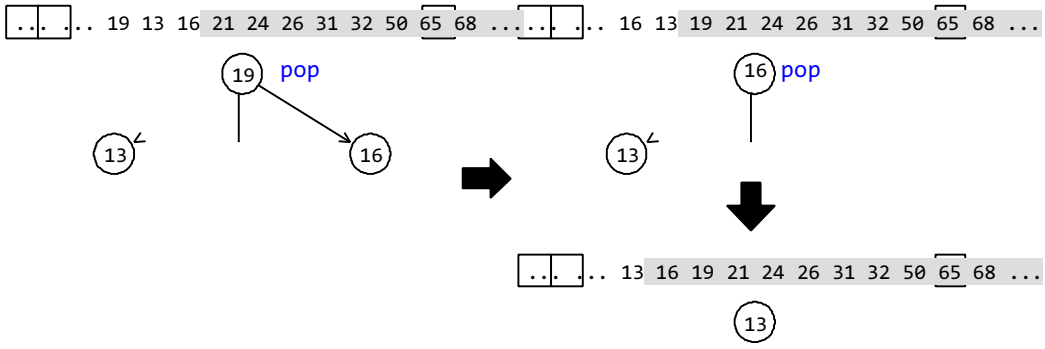


图 4-23 sort_heap 算法：不断对 heap 做 pop 动作，便可达到排序效果

make_heap 演算法

这个算法用来将一段现有的数据转化为一个 heap。其主要依据就是 4.7.1 节提到的 complete binary tree 的隐式表述（implicit representation）。

```
// 将 [first,last) 排列为一个 heap。
template <class RandomAccessIterator>
inline void make_heap(RandomAccessIterator first,
                      RandomAccessIterator last) {
  __make_heap(first, last, value_type(first), distance_type(first));
}

// 以下这组 make_heap() 不允许指定「大小比较标准」。
template <class RandomAccessIterator, class T, class Distance>
void __make_heap(RandomAccessIterator first,
                 RandomAccessIterator last, T*,
                 Distance*) {
  if (last - first < 2) return; // 如果长度为 0 或 1，不必重新排列。
  Distance len = last - first;
  // 找出第一个需要重排的子树头部，以 parent 标示出。由于任何叶节点都不需执行
  // perlocate down，所以有以下计算。parent 命名不佳，名为 holeIndex 更好。
  Distance parent = (len - 2)/2;

  while (true) {
    // 重排以 parent 为首的子树。len 是为了让 __adjust_heap() 判断操作范围
    __adjust_heap(first, parent, len, T(*(first + parent)));
    if (parent == 0) return;                // 走完根节点，就结束。
    parent--;                               // （即将重排之子树的）头部向前一个节点
  }
}
```

4.7.3 heap 没有迭代器

heap 的所有元素都必须遵循特别的（complete binary tree）排列规则，所以 heap 不提供走访功能，也不提供迭代器。

4.7.4 heap 测试实例

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

int main()
{
  {
  // test heap (底层以 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 (底层以 array 完成)
  int ia[9] = {0,1,2,3,4,8,9,3,5};
  make_heap(ia, ia+9);
  // array 无法动态改变大小，因此不可以对满载的 array 做 push_heap() 动作。
  // 因为那得先在 array 尾端增加一个元素。

  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;
  // 经过排序之后的 heap，不再是个合法的 heap

  // 重新再做一个 heap
  make_heap(ia, ia+9);
  pop_heap(ia, ia+9);
  cout << ia[8] << endl;                 // 9
  }

  {
  // test heap (底层以 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. priority_queue 概述
1

顾名思义，priority_queue 是一个拥有权值观念的 queue，它允许加入新元素、移除旧元素，审视元素值等功能。由于这是一个 queue，所以只允许在底端加入元素，并从顶端取出元素，除此之外别无其它存取元素的途径。

priority_queue 带有权值观念，其内的元素并非依照被推入的次序排列，而是自动依照元素的权值排列（通常权值以实值表示）。权值最高者，排在最前面。

预设情况下 priority_queue 系利用一个 max-heap 完成，后者是一个以 vector 表现的 complete binary tree（4.7 节）。max-heap 可以满足 priority_queue 所需要的「依权值高低自动递增排序」的特性。

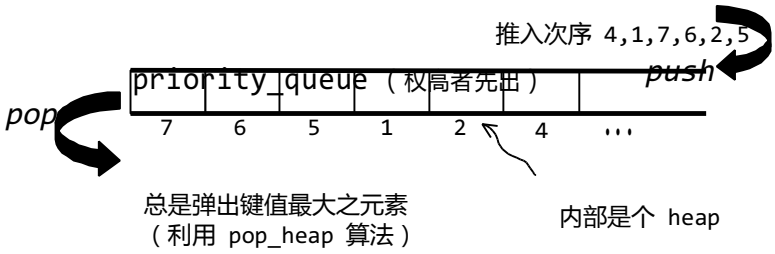


图 4-24 priority queue

4.8.2 priority_queue 定义式完整列表

由于 priority_queue 完全以底部容器为根据，再加上 heap 处理规则，所以其实作非常简单。预设情况下是以 vector 为底部容器。源码很简短，此处完整列出。

queue 以底部容器完成其所有工作。具有这种「修改某物接口，形成另一种风貌」之性质者，称为 adapter（配接器），因此 STL priority_queue 往往不被归类为 container（容器），而被归类为 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;          // 底层容器
  Compare comp;        // 元素大小比较标准
public:
  priority_queue() : c() {}
  explicit priority_queue(const Compare& x) : c(), comp(x) {}

// 以下用到的 make_heap(), push_heap(), pop_heap()都是泛型算法
// 注意，任一个建构式都立刻于底层容器内产生一个 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 是泛型算法，先利用底层容器的 push_back() 将新元素
      // 推入末端，再重排 heap。见 C++ Primer p.1195。
      c.push_back(x);
      push_heap(c.begin(), c.end(), comp); // push_heap 是泛型算法
    }
    __STL_UNWIND(c.clear());
  }
  void pop() {
    __STL_TRY {
      // pop_heap 是泛型算法，从 heap 内取出一个元素。它并不是真正将元素
      // 弹出，而是重排 heap，然后再以底层容器的 pop_back() 取得被弹出
      // 的元素。见 C++ Primer p.1195。
      pop_heap(c.begin(), c.end(), comp);
      c.pop_back();
    }
    __STL_UNWIND(c.clear());
  }
};
```

4.8.3 priority_queue 没有迭代器

priority_queue 的所有元素，进出都有一定的规则，只有 queue 顶端的元素（权值最高者），才有机会被外界取用。priority_queue 不提供走访功能，也不提供迭代器。

4.8.4 priority_queue 测试实例

```
// 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. slist 概述
1

STL list 是个双向串行（double linked list）。SGI STL 另提供了一个单向串行（single linked list），名为 slist。这个容器并不在标准规格之内，不过多做一些剖析，多看多学一些实作技巧也不错，所以我把它纳入本书范围。

slist 和 list 的主要差别在于，前者的迭代器属于单向的 *Forward Iterator*，后者的迭代器属于双向的 *Bidirectional Iterator*。为此，slist 的功能自然也就受到许多限制。不过，单向串行所耗用的空间更小，某些动作更快，不失为另一种选择。

slist 和 list 共同具有的一个相同特色是，它们的安插（insert）、移除（erase）、接合（splice）等动作并不会造成原有的迭代器失效（当然啦，指向被移除元素的那个迭代器，在移除动作发生之后肯定是会失效的）。

注意，根据 STL 的习惯，安插动作会将新元素安插于指定位置之前，而非之后。然而做为一个单向串行，slist 没有任何方便的办法可以回头定出前一个位置，因此它必须从头找起。换句话说，除了 slist 起始处附近的区域之外，在其它位置上采用 insert 或 erase 操作函式，都是不智之举。这便是 slist 相较于 list 之下的大缺点。为此，slist 特别提供了 insert_after() 和 erase_after() 供弹性运用。

基于同样的（效率）考虑，slist 不提供 push_back()，只提供 push_front()。因此 slist 的元素次序会和元素安插进来的次序相反。

4.9.2 slist 的节点

slist 节点和其迭代器的设计，架构上比 list 复杂许多，运用了继承关系，因此在型别转换上有复杂的表现。这种设计方式在第 5 章 RB-tree 将再一次出现。图 4-25 概述了 slist 节点和其迭代器的设计架构。

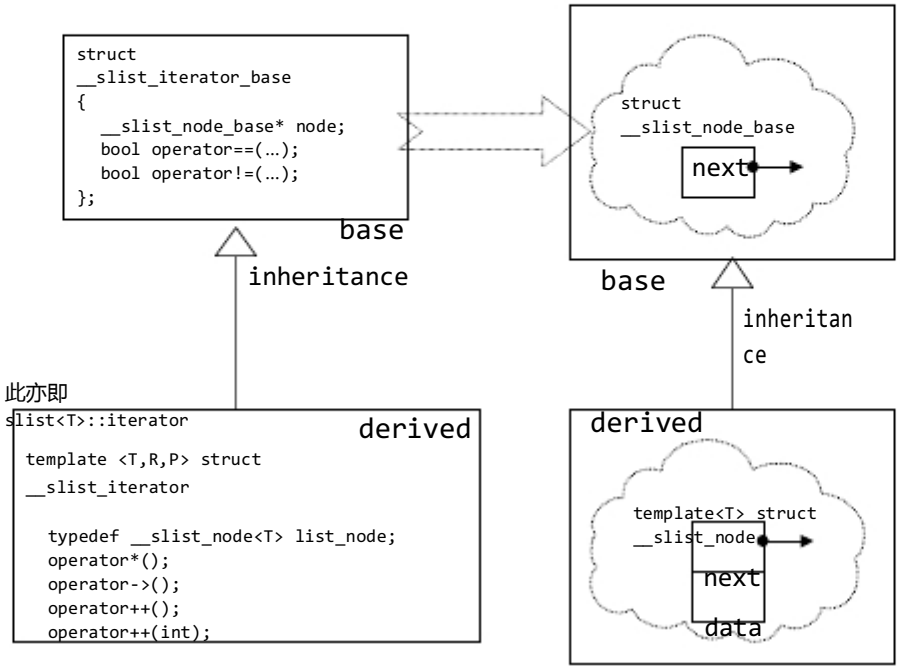


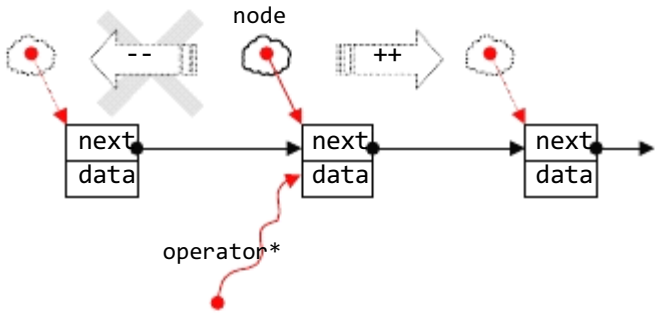
图 4-25 slist 的节点和迭代器的设计架构

```
// 单向串行的节点基本结构
struct __slist_node_base
{
  __slist_node_base* next;
};

// 单向串行的节点结构
template <class T>
struct __slist_node : public __slist_node_base
{
  T data;
};

// 全域函式：已知某一节点，安插新节点于其后。
inline __slist_node_base* __slist_make_link(
            __slist_node_base* prev_node,
            __slist_node_base* new_node)
{
  // 令 new 节点的下一节点为 prev 节点的下一节点
  new_node->next = prev_node->next;
  prev_node->next = new_node;                  // 令 prev 节点的下一节点指向 new 节点
```

```
  return new_node;
}

// 全域函式：单向串行的大小（元素个数）
inline size_t __slist_size(__slist_node_base* node)
{
  size_t result = 0;
  for ( ; node != 0; node = node->next)
    ++result;          // 一个一个累计
  return result;
```

4.9.3 slist 的迭代器

slist 迭代器可以下图表示：



实际构造如下。请注意它和节点的关系（见图 4-25）。

```
// 单向串行的迭代器基本结构
struct __slist_iterator_base
{
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef forward_iterator_tag iterator_category; // 注意，单向

  __slist_node_base* node; // 指向节点基本结构

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

  void incr() { node = node->next; } // 前进一个节点

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

```
    return node != x.node;
  }
};

// 单向串行的迭代器结构
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() 时会造成 __slist_iterator(0)，于是唤起上述函式。
  __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();      // 前进一个节点
    return *this;
  }
  self operator++(int)
  {
    self tmp = *this;
    incr();      // 前进一个节点
    return tmp;
  }

// 没有实作 operator--，因为这是一个 forward iterator
};
```

注意，比较两个 slist 迭代器是否等同时（例如我们常在循环中比较某个迭代器是否等同于 slist.end()），由于 __slist_iterator 并未对 operator== 实施多载化，所以会唤起 __slist_iterator_base::operator==。根据其中之定义，我们知道，两个 slist 迭代器是否等同，视其 __slist_node_base* node 是否等同而定。

4.9.4 slist 的数据结构

下面是 slist 源码摘要，我把焦点放在「单向串行之形成」的一些关键点上。

```
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();                    // 配置空间
    __STL_TRY {
      construct(&node->data, x);                    // 建构元素
      node->next = 0;
    }
    __STL_UNWIND(list_node_allocator::deallocate(node));
    return node;
  }

  static void destroy_node(list_node* node) {
    destroy(&node->data);                  // 将元素解构
    list_node_allocator::deallocate(node);                          // 释还空间
  }

private:
  list_node_base head; // 头部。注意，它不是指标，是实物。

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; }

  // 两个 slist 互换：只要将 head 交换互指即可。
  void swap(slist& L)
  {
    list_node_base* tmp = head.next;
    head.next = L.head.next;
    L.head.next = tmp;
  }

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

  // 从头部安插元素（新元素成为 slist 的第一个元素）
  void push_front(const value_type& x) {
    __slist_make_link(&head, create_node(x));
  }

  // 注意，没有 push_back()

  // 从头部取走元素（删除之）。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

下面是一个小小练习：

```
// 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，实际结构呈现如图 4-26。

接下来搜寻元素 1，并将新元素 99 安插进去，如图 4-27。注意，新元素被安插在插入点（元素 1）的前面而不是后面。

接下来搜寻元素 3，并将该元素移除，如图 4-28。

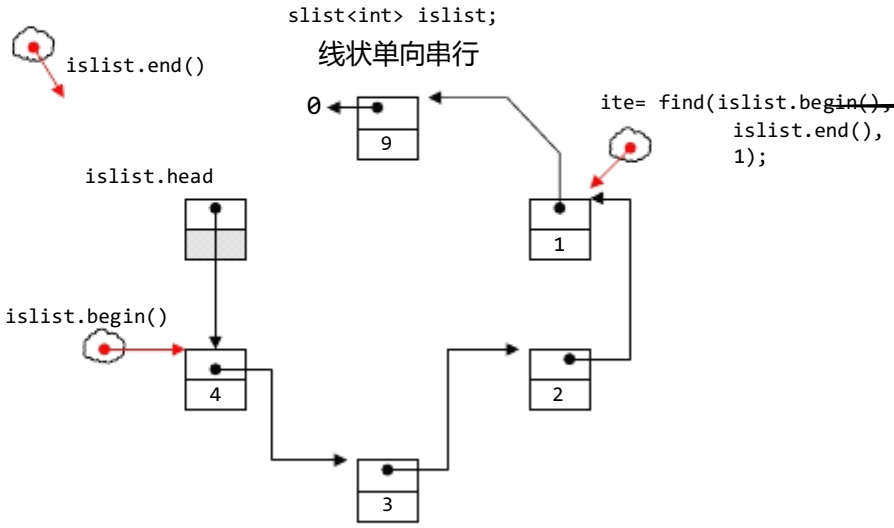


图 4-26 元素 9,1,2,3,4 依序安插到 slist 之后所形成的结构

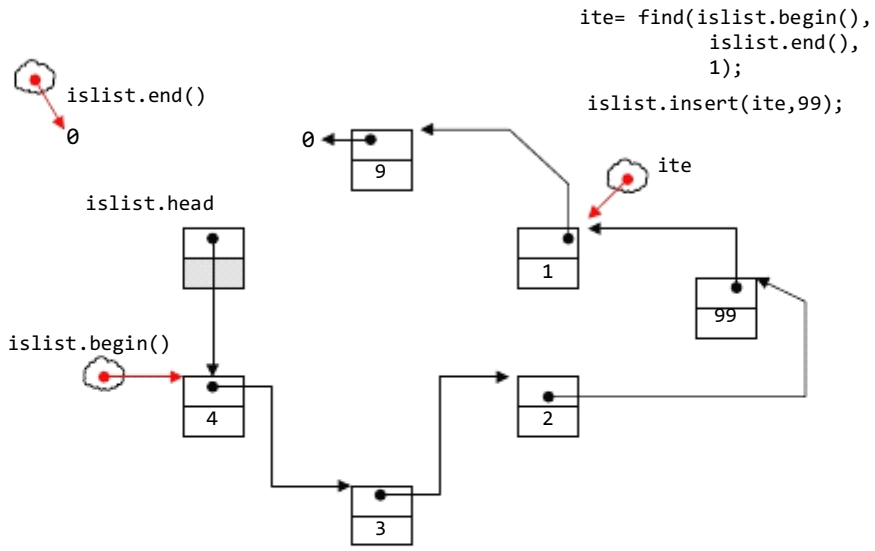


图 4-27 元素 9,1,2,3,4 依序安插到 slist 之后的实际结构

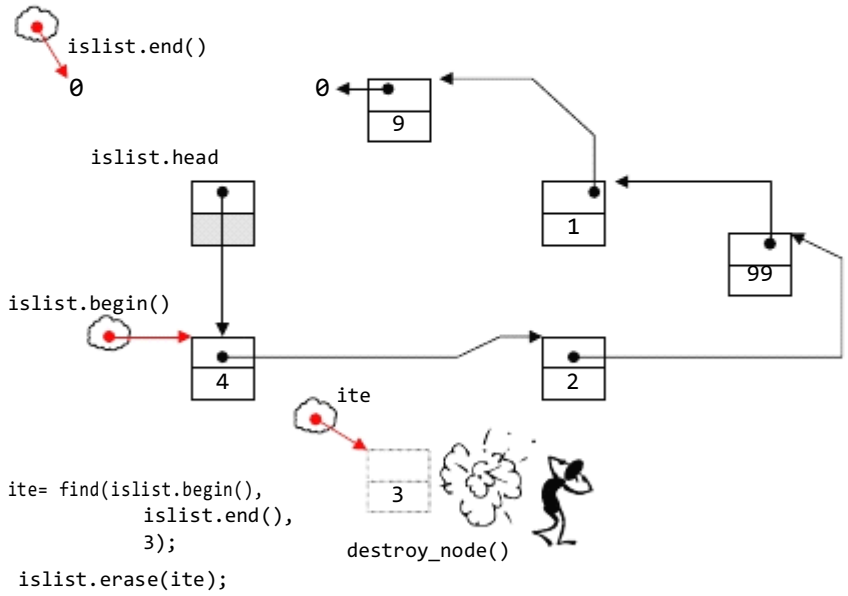


图 4-28 搜寻元素 3，并将该元素移除

如果你对于图 4-26、图 4-27、图 4-28 中的 end() 的画法感到奇怪，这里我要做一些说明。请注意，练习程序中一再以循环巡访整个 slist，并以迭代器是否等于 slist.end() 做为循环结束条件，这其中有一些容易疏忽的地方，我必须特别提醒你。当我们呼叫 end() 企图做出一个指向尾端（下一位置）的迭代器，STL 源码是这么进行的：

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

这会因为源码中如下的定义：

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

而形成这样的结果：

```
__slist_iterator<T, T&, T*>(0);                    // 产生一个暂时对象，引发 ctor
```

从而因为源码中如下的定义：

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

而导致基础类别的建构：

```
__slist_iterator_base(0);
```

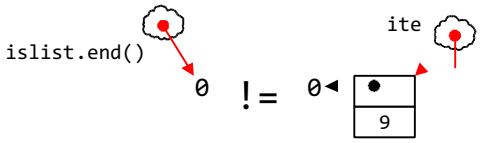
并因为源码[中] 这样的定义：

```
struct __slist_iterator_base
{
  __slist_node_base* node; // 指向节点基本结构
  __slist_iterator_base(__slist_node_base* x) : node(x) {}
  ...
};
```

而导致：

```
node(0);
```

因此我在图 4-26、图 4-27、图 4-28 [中] 皆以[下] 图左侧的方式表现 end()，它和[下] 图右侧的迭代器截然不同。



A

参考书籍与推荐读物

Bibliography

Genericity/STL 领域里头，已经产生了一些经典作品。

我曾经书写一篇文章，介绍 Genericity/STL 经典好书，分别刊载于台北《*Run!PC*》杂志和北京《程序员》杂志。该文讨论 Genericity/STL 的数个学习层次，文中所列书籍不但是我的推荐，也是本书《STL 源码剖析》写作的部分参考。以下摘录该文中关于书籍的介绍。全文见 http://www.jjhou.com/programmer-2-stl.htm。

侯捷观点《Genericity/STL 大系》— 泛型技术的三个学习阶段

自从被全球软件界广泛运用以来，C++ 有了许多演化与变革。然而就像人们总是把目光放在艳丽的牡丹而忽略了花旁的绿叶，做为一个广为人知的对象导向程序语言（Object Oriented Programming Language），C++ 所支持的另一种思维 — 泛型编程 — 被严重忽略了。说什么红花绿叶，好似主观上划分了主从，其实对象导向思维和泛型思维两者之间无分主从。两者相辅相成，肯定对程序开发有更大的突破。

面对新技术，我们的最大障碍在于心中的怯弱和迟疑。To be or not to be, that is the question! 不要和哈姆雷特一样犹豫不决，当你面对一项有用的技术，必须果敢。

王国维说大事业大学问者的人生有三个境界。依我看，泛型技术的学习也有三个境界，第一个境界是运用 **STL**。对程序员而言，诸多抽象描述，不如实象的程序

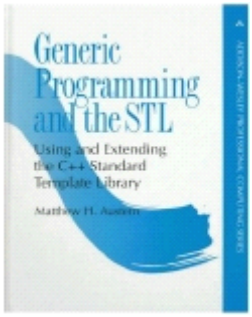
码直指人心。第二个境界是了解泛型技术的内涵与 **STL** 的学理。不但要理解 STL 的概念分类学（concepts taxonomy）和抽象概念库（library of abstract concepts），最好再对数个 STL 组件（不必太多，但最好涵盖各类型）做一番深刻追踪。STL 源码都在手上（就是相应的那些 header files 嘛），好好做几个个案研究，便能够对泛型技术以及 STL 的学理有深刻的掌握。

第三个境界是扩充 **STL**。当 STL 不能满足我们的需求，我们必须有能力动手写一个可融入 STL 体系中的软件组件。要到达这个境界之前，得先彻底了解 STL，也就是先通过第二境界的痛苦折磨。

也许还应该加上所谓第零境界：**C++ template** 机制。这是学习泛型技术及 STL 的第一道门槛，包括诸如 class templates, function templates, member templates, specialization, partial specialization。更往基础看去，由于 STL 大量运用了 operator overloading（运算子多载化），所以这个技法也必须熟捻。

以下，我便为各位介绍多本相关书籍，涵盖不同的切入角度，也涵盖上述各个学习层次。另有一些则为本书之参考依据。为求方便，以下皆以学术界惯用法标示书籍代名，并按英文字母排序。凡有中文版者，我会特别加注。

[Austern98]: *Generic Programming and the STL - Using and Extending the C++ Standard Template Library*, by Matthew H. Austern, Addison Wesley 1998. 548 pages
繁体中文版：《泛型程序设计与 STL》，侯捷/黄俊尧合译，碁峰 2000, 548 页。



这是一本艰深的书。没有二两二，别想过梁山，你必须对 C++ template 技法、STL 的运用、泛型设计的基本精神都有相当基础了，才得一窥此书堂奥。

The Annotated STL Sources

此书第一篇对 STL 的设计哲学有很好的导入，第二篇是详尽的 STL concepts 完整规格，第三篇则是详尽的 STL components 完整规格，并附运用范例：

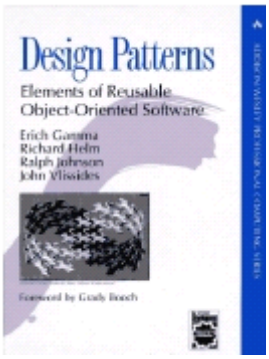
```
PartI : Introduction to Generic Programming
1. A Tour of the STL
2. Algorithms and Ranges
3. More about Iterators
4. Function Objects
5. Containers
PartII : Reference Manual: STL Concepts
6. Basic Concepts
7. Iterators
8. Function Objects
9. Containers
PartIII : Reference Manual : Algorithms and 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
```

此书通篇强调 STL 的泛型理论基础，以及 STL 的实作规格。你会看到诸如 concept，model，refinement，range，iterator 等字词的意义，也会看到诸如 *Assignable, Default Constructible, Equality Comparable, Strict Weakly Comparable* 等观念的严谨定义。虽然一本既富学术性又带长远参考价值的工具书，给人严肃又艰涩的表象，但此书第二章及第三章解释 iterator 和 iterator traits 时的表现，却不由令人击节赞赏大叹精彩。一旦你有能力彻底解放 traits 编程技术，你才有能力观看 STL 源码（STL 几乎无所不在[46] 运用 traits 技术）并进一步撰写符合规格的 STL 兼容组件。就像其他任何 framework 一样，STL 以开放源码的方式呈现市场，这种白盒子方式使我们在更深入剖析技术时（可能是为了透彻，可能是为了扩充），有一个终极依恃。因此，观看 STL 源码的能力，我认为对技术的养成与掌握，极为重要。

总的来说，此书在 STL 规格及 STL 学理概念的资料及说明方面，目前无出其右者。不论在（1）泛型观念之深入浅出、（2）STL 架构组织之井然剖析、（3）STL 参考文件之详实整理三方面，此书均有卓越表现。可以这么说，在泛型技术和 STL 的学

习道路[上]，此书并非万能（至少它不适合初学者），但如果你希望彻底掌握泛型技术与 STL，没有此书万万不能。

[Gamma95]: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. 395 pages
繁体[中] 文版：
《对象导向设计模式—可再利用对象导向软件之要素》叶秉哲译，培生 2001，458 页



此书与泛型或 STL 并没有直接关系。但是 STL 的两大类型组件：*Iterator* 和 *Adapter*，被收录于此书 23 个设计样式（design patterns）[中]。此书所谈的其它设计样式在 STL 之[中]也有发挥。两相比照，尤其是看过 STL 的源码之后，对于这些设计样式会有更深的体会，回映过来对 STL 本身架构也会有更深[一]层的体会。

[Hou02a]: 《STL 源码剖析 — 向专家取经，学习 STL 实作技术、强型检验、记忆体管理、算法、数据结构之高阶编程技法》，侯捷着，碁峰 2002，??? 页。

这便是你手[上]这本书。揭示 SGI STL 实作版本的关键源码，涵盖 STL 六大组件之

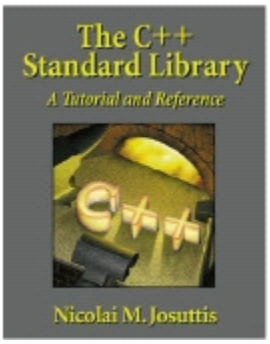
The Annotated STL Sources

实作技术和原理解说。是学习泛型编程、数据结构、算法、内存管理...等高阶编程技术的终极性读物，毕竟...唔...源码之前了无秘密。

[Hou02b]:《泛型思维 — Genericity in C++》，侯捷着（计划中）

下笔此刻，此书尚在撰写当中，内容涵盖语言层次（C++ templates 语法、Java generic 语法、C++ 运算子重载），STL 原理介绍与架构分析，STL 现场重建，STL 深度应用，STL 扩充示范，泛型思考。并附一个微型、高度可移植的 STLLite，让读者得以最精简的方式和时间一窥 STL 全貌，一探泛型之宏观与微观。

[Josuttis99]: *The C++ Standard Library - A Tutorial and Reference*, by Nicolai M. Josuttis, Addison Wesley 1999. 799 pages
繁体中文版：
《C++ 标准链接库 — 学习教本与参考工具》，侯捷/孟岩译，碁峰 2002，800 页。



一旦你开始学习 STL，乃至实际运用 STL，这本书可以为你节省大量的翻查、参考、错误尝试的时间。此书各章如下：

```
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
```

此书涵盖面广，不仅止于 STL，而且是整个 C++ 标准链接库，详细介绍每个组件的规格及运用方式，并佐以范例。作者的整理功夫做得非常扎实，并大量运用图表做为解说工具。此书的另一个特色是涵盖了 STL 相关各种异常（exceptions），这很少见。

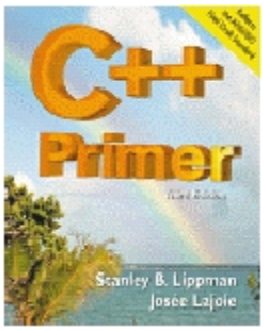
此书不仅介绍 STL 组件的运用，也导入关键性 STL 源码。这些源码都经过作者的节录整理，砍去枝节，留下主干，容易入目。这是我特别激赏的一部份。繁中取简，百万军中取敌首级，不是容易的任务，首先得对庞大的源码有清晰的认识，再有坚定而正确的诠释主轴，知道什么要砍，什么要留，什么要批注。

阅读此书，不但得以进入我所谓的第一学习境界，甚且由于关键源码的提供，得以进入第二境界。此书也适度介绍某些 STL 扩充技术。例如 6.8 节介绍如何以 smart pointer 使 STL 容器具有 "reference semantics"（STL 容器原本只支持 "value semantics"），7.5.2 节介绍一个订制型 iterator，10.1.4 节介绍一个订制型 stack，10.2.4 节介绍一个订制型 queue，15.4 节介绍一个订制型 allocator。虽然篇幅都不长，只列出基本技法，但对于想要扩充 STL 的程序员而言，有个起始终究是一种实质上的莫大帮助。就这点而言，此书又进入了我所谓的第三学习境界。

正如其副标所示，本书兼具学习用途及参考价值。在国际书市及国际 STL 相关研讨会上，此书都是首选。盛名之下无虚士，诚不欺也。

The Annotated STL Sources

[Lippman98] : *C++ Primer*, 3rd Edition, by Stanley Lippman and Josée Lajoie, Addison Wesley Longman, 1998. 1237 pages.
繁体中文版：《C++ Primer 中文版》，侯捷译，碁峰 1999，1237 页。



这是一本 C++ 百科经典，向以内容广泛说明详尽著称。其中与 template 及 STL 直接相关的章节有：

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

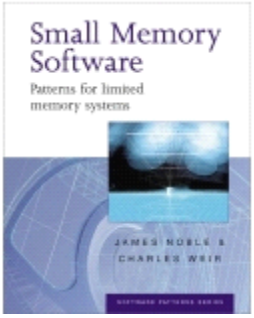
与 STL 实作技术间接相关的章节有：

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

书中有大量范例，尤其附录列出所有 STL 泛型算法的规格、说明、实例，是极佳的学习数据。不过书上有少数例子，由于作者疏忽，未能完全遵循 C++ 标准，仍沿用旧式写法，修改方式可见 www.jjhou.com/errata-cpp-primer-appendix.htm。

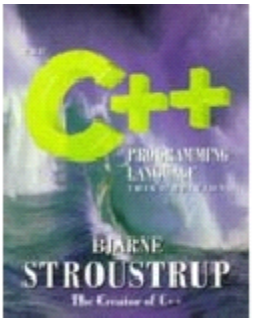
这本 C++ 百科全书并非以介绍泛型技术的角度出发，而是因为 C++ 涵盖了 template 和 STL，所以才介绍它。因此在相关组织上，稍嫌凌乱。不过我想，没有人会因此对它求全责备。

[Noble01]: *Small Memory Software – Patterns for systems with limited memory*, by James Noble & Charles Weir, Addison Wesley, 2001. 333 pages.
繁体* 文版：《内存受限系统 之设计样式》，侯捷/王飞译，碁峰 2002，333 页。



此书和泛型技术、STL 没有任何关连。然而由于 SGI STL allocator（空间配置器）在内存配置方面运用了 memory pool 手法，如果能够参考此书所整理的一些记忆体管理经典手法，颇有帮助，并收触类旁通之效。

[Struostrup97]: *The C++ Programming Language*, 3rd Editoin, by Bjarne Stroustrup, Addison Wesley Longman, 1997. 910 pages
繁体* 文版：《C++ 程序语言经典本》，叶秉哲译，儒林 1999，总页数未录。



这是一本 C++ 百科经典，向以学术权威（以及口感艰涩 ）著称。本书内容直接与 template 及 STL 相关的章节有：

```
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
```

与 STL 实作技术间接相关的章节有：

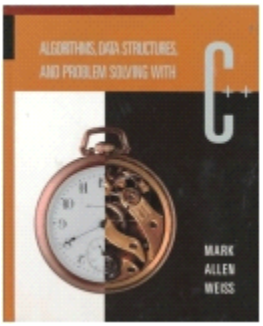
The Annotated STL Sources

chap11: Operator Overloading

其中第 19 章对 Iterators Traits 技术的介绍，在 C++ 语法书中难得一见，不过蜻蜓点水不易引发阅读兴趣。关于 Traits 技术，[Austern98] 表现极佳。

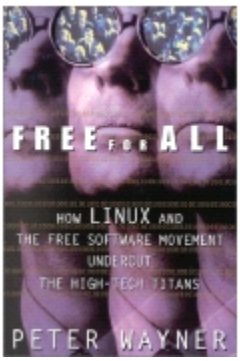
这本 C++ 百科全书并非以介绍泛型技术的角度出发，而是因为 C++ 涵盖了 template 和 STL，所以才介绍它。因此在相关组织上，稍嫌凌乱。不过我想，没有人会因此对它求全责备。

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



此书和泛型技术、STL 没有任何关连。但是在你认识 STL 容器和 STL 算法之前，一定需要某些数据结构（如 red black tree, hash table, heap, set...）和算法（如 quick sort, heap sort, merge sort, binary search...）以及 Big-Oh 复杂度标记法的学理基础。本书在学理叙述方面表现不俗，用字用例浅显易懂，颇获好评。

[Wayner00]: *Free For All — How Linux and the Free Software Movement Undercut the High-Tech Titans*, by Peter Wayner, HarperBusiness, 2000. 340 pages
繁体中文版：《开放原始码—Linux 与自由软件运动对抗软件巨人的故事》，
蔡忆怀译，商周 2000，393 页。



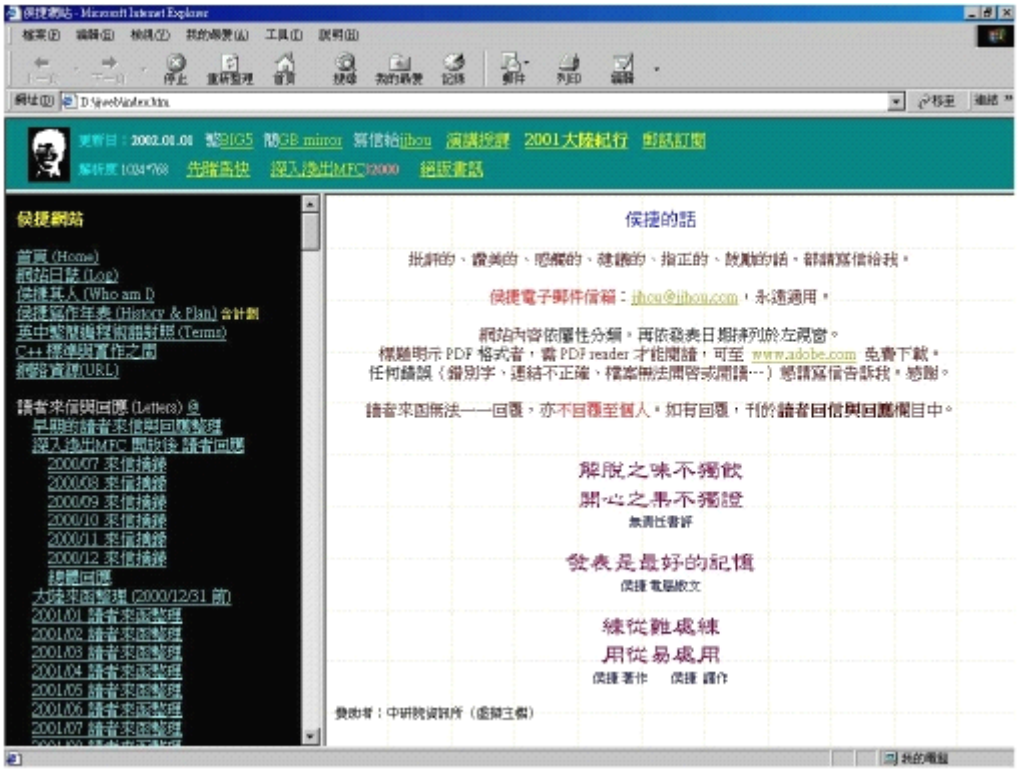
《STL 源码剖析》一书采用 SGI STL 实作版本为解说对象，而 SGI 版本属于源码开放架构下的一员，因此《STL 源码剖析》第一章对于 **open source**（源码开放精神）、**GNU**（由 Richard Stallman 创先领导的开放改革计划）、**FSF**（Free Software Foundation，自由软件基金会）、**GPL**（General Public License，广泛开放授权）等等有概要性的说明，皆以此书为参考依据。

B

侯捷网站

本书支援站点简介

侯捷网站是我的个人站点，收录写作教育生涯的所有足迹。我的所有作品的勘误、讨论、源码下载、电子书下载等服务都在这个站点上进行。永久网址是 http://www.jjhou.com（繁体），简体镜像站点为 http://jjhou.csdn.com。下面是进入侯捷网站后的画面，其上窗口是变动主题，提供最新动态。左窗口是目录，右窗口是主画面：



左窗口之主目录，内容包括：

首页
网站日志
侯捷其人
侯捷写作年表（History）含计划
英中繁简编程术语对照（Terms）
C++ 标准与实作之间
网络资源(URL)

读者来信与响应
课程
电子书开放
程序源码下载
答客问（Q/A）
作品勘误(errata)
无责任书评 1 1993.01~1994.04
无责任书评 2 1994.07~1995.12
无责任书评 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）
《程序员》杂志文章
侯捷著作
侯捷译作
作序推荐

本书《STL 源码剖析》出版后之相关服务，皆可于以上各相关字段中获得。

C

STLPort 的移植经验

by 孟岩 [1]

STL 是一个标准，各商家根据这个标准开发了各自的 STL 版本。而在这形形色色的 STL 版本中，SGI STL 无疑是最引人瞩目的一个。这当然是因为这个 STL 产品系出名门，其设计和编写者名单中，Alexander Stepanov 和 Matt Austern 赫然在内，有两位大师坐镇，其代码水平自然有了最高保证。SGI STL 不但在效率上一直名列前茅，而且完全依照 ISO C++ 之规范设计，使用者尽可放心。此外，SGI STL 做到了 thread-safe，还体贴地为用户增设数种组件，如 hash，hash_map，hash_multimap，slist 和 rope 容器等等。因此无论在学习或实用上，SGI STL 应是首选。

无奈，SGI STL 本质上是为了配合 SGI 公司自身的 UNIX 变体 IRIX 而量身定做，其它平台上的 C++ 编译器想使用 SGI STL，都需要一番周折。著名的 GNU C++ 虽然也使用 SGI STL，但在发行前已经过调试整合。一般用户，特别是 Windows 平台上的 BCB/VC 用户要想使自己的 C++ 编译器与 SGI STL 共同工作，可不是一件容易的事情。俄国人 Boris Fomitchev 注意到这个问题之后，建立了一个免费提供服务的项目，称为 STLport，旨在将 SGI STL 的基本代码移植到各主流编译环境中，使各种编译器的用户都能够享受到 SGI STL 带来的先进机能。STLport 发展过程中曾接受 Matt Austern 的指导，发展到今天，已经比较成熟。最新的 STLport 4.0，可以从 www.stlport.org 免费下载，zip 档案体积约 1.2M，可支持各主流 C++ 编译环境的移植。BCB 及 VC 当然算是主流编译环境，所以当然也得到了 STLport 的关照。但据笔者实践经验看来，配置过程中还有一些障碍需要跨越，本文详细

感谢孟岩先生同意将他的宝贵经验与本书读者分享。原文以大陆术语写就，为迁就台湾读者方便，特以台湾术语略做修改。

指导读者如何在 Borland C++Builder 5.5 及 Visual C++ 6.0 环境中配置 STLport。

首先请从 www.stlport.org 下载 STLport 4.0 的 ZIP 档案，档名 stlport-4.0.zip。然后利用 WinZip 等工具展开。生成 stlport-4.0 目录，该目录中有（而且仅有）一个子目录，名称亦为 stlport-4.0，不妨将整目录拷贝到你的合适位置，然后改一个合适的名字，例如配合 BCB 者可名为 STL4BC...等等。

下面分成 BCB 和 VC 两种情形来描述具体过程。

Borland C++Builder 5

Borland C++Builder5 所携带的 C++ 编译器是 5.5 版本，在当前主流的 Windows 平台编译器中，对 ISO *C++ Standard* 的支持是最完善的。以它来配合 SGI STL 相当方便，也是笔者推荐之选。手上无此开发工具的读者，可以到 www.borland.com 免费下载 Borland C++ 5.5 编译器的一个精简版，该精简版体积为 8.54M，名为 freecommandlinetools1.exe，乃一自我解压缩安装档案，在 Windows 中执行它便可安装到你选定的目录中。展开后体积 50M。

以下假设你使用的 Windows 安装于 C:\Windows 目录。如果你有 BCB5，假设安装于 C:\Program Files\Borland\CBuilder5；如果你没有 BCB5，而是使用上述的精简版 BCC，则假设安装于 C:\BCC55 目录。STLport 原包置于 C:\STL4BC，其中应有以下内容：

```
<目录> doc
<目录> lib
<目录> src
<目录> stlport
<目录> test
档案 ChangLog
档案 Install
档案 Readme
档案 Todo
```

请确保 C:\Program Files\Borland\CBuilder5\Bin 或 C:\BCC55\Bin 已登记于你的 Path 环境变量中。

笔者推荐你在安装之前读一读 Install 档案，其中讲到如何避免使用 SGI 提供的

iostream。如果你不愿意使用 SGI iostream，STLport 会在原本编译器自带的 iostream 外加一个 wrapper，使之能与 SGI STL 共同合作。不过 SGI 提供的 iostream 标准化程度好，和本家的 STL 代码配合起来速度也快些，所以笔者想不出什么理由不使用它，在这里假定大家也都乐于使用 SGI iostream。有不同看法者尽可按照 Install 档案的说法调整。

下面是逐一步骤（本任务均在 DOS 命令状态下完成，请先打开一个 DOS 窗口）：

1. 移至 C:\Program Files\Borland\CBuilder5\bin，使用任何文字编辑器修改以下两个档案。

   档案一 bcc32.cfg 改为：

   ```
   -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"
   ```

   以上为了方便阅读，以 "\" 符号将很长的一行折行。本文以下皆如此。

   档案二 ilink32.cfg 改为：

   ```
   -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 目录中并不存在这两个档案，请你自己用文字编辑器手工做出这两个档案来，内容与上述有所不同，如下。

   档案一 bcc32.cfg 内容：

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

   档案二 ilink32.cfg 内容：

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

2. 进入 C:\STL4BC\SRC 目录。
3. 执行命令 `copy bcb5.mak Makefile`
4. 执行命令 `make clean all`

   这个命令会执行很长时间，尤其在老旧机器上，可能运行 30 分钟以上。屏幕

不断显示工作情况，有时你会看到好像在反复做同样几件事，请保持耐心，这其实是在以不同编译开关建立不同性质的标的链接库。

5. 经过一段漫长的编译之后，终于结束了。现在再执行命令 make install。这次需要的时间不长。
6. 来到 C:\STL4BC\LIB 目录，执行：

```
copy *.dll c:\windows\system;
```

7. 大功告成。下面一步进行检验。rope 是 SGI STL 提供的一个特有容器，专门用来对付超大规模的字符串。string 是细弦，而 rope 是粗绳，可以想见 rope 的威力。下面这个程序有点暴殄天物，不过倒也还足以做个小试验：

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

using namespace std;

int main()
{
    // crope 就是容纳 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
```

现在，针对上述程序进行编译：bcc32 issgistl.cpp。咦，怪哉，linker 报告说找不到 stlport_bcc_static.lib，到 C:\STL4BC\LIB 看个究竟，确实没有这个档案，倒是有一个 stlport_bcb55_static.lib。笔者发现这是 STLport 的一个小问题，需要将链接库文件名称做一点改动：

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

这个做法颇为稳妥，原本的 stlport_bcb55_static.lib 也保留了下来。以其它选项进行编译时，如果遇到类似问题，只要照葫芦画瓢改变文件名称就没问题了。

现在再次编译，应该没问题了。可能有一些警告讯息，没关系。只要能运行，就表示 rope 容器起作用了，也就是说你的 SGI STL 开始工作了。

Microsoft Visual C++ 6.0:

Microsoft Visual C++ 6.0 是当今 Windows 下 C++ 编译器主流中的主流，但是对于 ISO C++ 的支持不尽如人意。其所配送的 STL 性能也比较差。不过既然是主流，STLport 自然不敢怠慢，下面介绍 VC 中的 STLport 安装方法。

以下假设你使用的 Windows 系统安装于 C:\Windows 目录，VC 安装于 C:\Program Files\Microsoft Visual Studio\VC98；而 STLport 原包置于 C:\STL4VC，其中应有以下内容：

```
<目录> doc
<目录> lib
<目录> src
<目录> stlport
<目录> test
档案 ChangLog
档案 Install
档案 Readme
档案 Todo
```

请确保 C:\Program Files\Microsoft Visual Studio\VC98\bin 已设定在你的 Path 环境变数中。

下面是逐一步骤（本任务均在 DOS 命令状态下完成，请先打开一个 DOS 窗口）：

1. 移至 C:\Program Files\Microsoft Visual Studio\VC98 中，使用任何文字编辑器修改档案 vcvars32.bat。将其中原本的两行：

   ```
   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%
   ```

   以上为了方便阅读，以 "\" 符号将很长的一行折行。

   修改完毕后存盘，然后执行之。一切顺利的话应该给出一行结果：

   ```
   Setting environment for using Microsoft Visual C++ tools.
   ```

如果你预设的 DOS 环境空间不足，这个 BAT 档执行过程中可能导致环境空间不足，此时应该在 DOS 窗口的「内容」对话框中找到「内存」附页，将「起始环境」（下拉式选单）改一个较大的值，例如 1280 或 2048。然后再开一个 DOS 窗口，重新执行 vcvars32.bat。

2. 进入 C:\STL4VC\SRC 目录。
3. 执行命令 `copy vc6.mak Makefile`
4. 执行命令 `make clean all`

   如果说 BCB 编译 STLport 的时间很长，那么 VC 编译 STLport 的过程就更加漫长了。屏幕反反复覆地显示似乎相同的内容，请务必保持耐心，这其实是在以不同编译开关建立不同性质的标的链接库。
5. 经过一段漫长的编译之后，终于结束了。现在你执行命令 `make install`。这次需要的时间不那么长，但也要有点耐心。
6. 大功告成。下一步应该检验是不是真的用上了 SGI STL。和前述的 BCB 过程差不多，找一个运用了 SGI STL 特性的程序，例如运用了 rope，slist，hash_set，hash_map 等容器的程序来编译。注意，编译时务必使用以下格式：

   ```
   cl /GX /MT program.cpp
   ```

   这是因为 SGI STL 大量使用了 try…throw…catch，而 VC 预设情况下并不支持此一语法特性。/GX 要求 VC++ 编译器打开对异常处理的语法支持。/MT 则是要求 VC linker 将本程序的 obj 文件和 libcmt.lib 连结在一起 — 因为 SGI STL 是 thread-safe，必须以 multi-thread 的形式运行。

如果想要在图形介面中使用 SGI STL，可在 VC 整合环境内调整 Project | Setting(Alt+F7)，设置编译选项，请注意一定要选用 /MT 和 /GX，并引入选项 /Ic:\stl4vc\stlport 及 /libpath:c:\stl4vc\lib。

整个过程在笔者的老式 Pentium 150 机器上耗时超过 3 小时，虽然你的机器想必快得多，但也必然会花去出乎你意料的时间。全部完成后，C:\STL4VC 这个目录的体积由原本区区 4.4M 膨胀到可怕的 333M，当然这其中有 300M 是编译过程产生的.obj 档，如果你确信自己的 STLport 工作正常的话，可以删掉它们，空出硬盘空间。不过这么一来若再进行一次安装程序，就只好再等很长时间。

另外，据笔者勘察，STLport 4.0 所使用的 SGI STL 并非最新问世的 SGI STL3.3 版

本，不知道把 SGI STL3.3 的代码导入 STLport 会有何效果，有兴趣的读者不妨一试。

大致情形就是这样，现在，套用 STLport 自带档案的结束语：享受这一切吧（Have fun!）

孟岩
2001-3-11

索引

请注意，本书并未探讨所有的 STL 源码（那需要数千页篇幅），所以
请不要将此¯ 索引视为完整的 STL 组件索引

()
  as operator 414
*
  for auto_ptrs 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_ptrs 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 deques 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 deques 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 Foundation7
front()
for deques 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 deques 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 deques 163
  for lists 137
  for vectors 116,123
pop_front()
  for deques 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 deques 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
iterator 214
s
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