深入C++系列

TP312C
A38C
TP313
L126

Generic Programming and the STL

泛型编程与STL

Matthew H. Austern 著
侯 捷 译

A1066234

Addison
Wesley

中国电力出版社

Generic Programming and the STL: Using and Extending the C++ Standard Template Library (ISBN 0-201-30956-4)

Matthew H. Austern

Authorized translation from the English language edition, entitled Generic Programming and the STL, published by Addison Wesley, Copyright©1999

All rights reserved. NO part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2003

本书由美国培生集团授权出版。

北京市版权局著作权合同登记号 图字：01-2001-2338 号

图书在版编目（CIP）数据

泛型编程与STL/（美）奥斯滕著；侯捷译．—北京：中国电力出版社，2003

（深入C++系列）

ISBN 7-5083-1487-5

Ⅰ.泛… Ⅱ.①奥…②侯… Ⅲ.C语言—程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2003）第017798号

责任编辑：姚贵胜

丛 书 名： 深入C++系列

书　　名： 泛型编程与STL

编　　著：（美）Matthew H. Austern

翻　　译： 侯捷

出版发行：中国电力出版社

地址：北京市三里河路6号 邮政编码：100044

电话：（010）88515918 传真：（010）88423191

印　　刷： 北京地矿印刷厂

开　　本： 787×1092 1/16　　**印　　张：** 35.75　　**字　　数：** 806千字

书　　号： 7-5083-1487-5

版　　次： 2003年4月北京第一版

印　　次： 2003年4月第一次印刷

印　　数： 1～6000 册

定　　价： 72.00 元

译序（侯 捷）

泛型编程（generic programming）和其第一份重要实现品 STL（Standard Template Library），对许多人而言并不陌生，但是对更多人而言却非常遥远。

自从被全世界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，作为一个广为人知的面向对象程序语言（Object Oriented Programming Language），C++ 的另一面 — 泛型编程思维 — 被严重地忽略了。不能说面向对象思维和泛型思维有什么主从之分，但是红花绿叶相辅相成，肯定能对程序开发有更大的突破。

泛型思维在 C++ 身上主要以 templates 及相关特性来表现。许多人以为 template 是多么高阶的技巧，**望之俨然**；接触后才发现，**即之也温**。更深入研习泛型思维后又有另一层体会：**其言也厉**。

C++ templates 相关特性不难掌握，然而以此技术为载具所发展出来的一套泛型程序库：STL（现已纳入 C++ 标准程序库），背后却另有一套严密逻辑。本书一语道破 STL 的精神：一个严谨的软件概念（*concepts*）分类学。是的，这些 *concepts* 和任何程序语言无关，但就像 OO 思维需要 OO 语言的协助才得以顺利展现一样，泛型思维需要泛型语言的协助，才得以顺利展现。Ada 和 C++ 都支持泛型语法和语义，Java 也已跟进[1]。

换言之，泛型编程和 C++ templates 和 STL 不该并为一谈。然以现今发展看来，三者又几乎被划上等号。本书即以 C++ 为载具，第一篇讲述泛型思维的演进及 STL 的组织概念，第二篇整理 STL 所涵盖的全部 concepts 的完整规格，第三篇整理 STL 所涵盖的全部 components 的完整规格。

人们对于 STL 的最大误解是**效率**。事实上 STL 提供的是一个不损及效率的抽象性。泛型编程和面向对象编程不同，它并不要求你通过额外的间接层来调用函数；它让你撰写完全一般化并可重复运用的算法，其效率和「针对特定数据型别而设计」的算法旗鼓相当。每一个算法、每一个容器的操作行为，其复杂度都有明确规范 — 通常是最佳效率或极佳效率。在接受规格书明定的复杂度之后，我想你不会认为自己亲手撰写的码，能够更胜通过严格检验的世界通用程序库。

---

[1] 请参考 "GJ : A Generic Java - java may be in for some changes". Philip Wadler, DDJ, Feb., 2000。

人们对 STL 效率的误解，有一大部分是把编译期效率和运行期效率混为一谈了。的确，大量而嵌套地运用 templates，会导致编译器在进行 template 引数推导（argument deduction）及具现化（instantiation）时耗用大量时间。但它绝不影响运行效率。至于对项目开发时程所带来的影响，我要说，和 STL 为我们节省下来的开发时间相比，微不足道。

STL 的严重缺点在于，它尚未支持 persistence（对象持久性）。在良好的解决方案尚未开发出来之前，persistence 必须由用户自行完成。

作者 Austern 在本书「前言」对泛型思维及其技术演化还有更多精辟讨论，我建议你此刻（噢不，至少看完这篇译序之后）立即翻看「前言」，一定能够对整个技术的来龙去脉有所掌握。

永远记住，面对新技术，程序员最大的困难在于心中的怯弱。To be or not to be, that is the question! 不要和哈姆雷特一样犹豫不决，当你面对一个有用的技术，应该果断。

我在台湾元智大学开授一门「泛型编程」课程。本书便是我为同学列出的高阶参考书。本书在 (1) 泛型观念之深入浅出、(2) STL 组织之井然剖析、(3) STL 参考文件之详实整理 三方面有卓越的表现。可以这么说，在泛型技术和 STL 的学习道路上，本书并非万能（至少它不适合初学者），但如果你希望彻彻底底掌握泛型技术和 STL 的深层观念，没有此书万万不能 ☺。

侯捷 2003.03.08 于台湾新竹

jjhou@jjhou.com
http://www.jjhou.com（繁）
http://jjhou.csdn.net（简）

p.s. 本书采页页对译方式，并保留原文索引。简体版系以繁体版为基础。繁体版第一阶段初译工作由黄俊尧先生负责，同挂译者之名；由于俊尧未涉简体版之大小事务，因此未同列简体版译者。但我仍要在此感谢俊尧的贡献。

p.s. 本书根据英文版第四刷（4th printing）制作，并修订前三刷之错误。请上侯捷网站（网址如上）观看后续的讨论、勘误、程序样例。网页所列之 2003/ 02/20 前的繁体版勘误，已于本简体版修正完毕。

本书保留大量简短易读之英文术语；时而中英并陈。以下是我所有书籍（无论著译）的习惯用语，特请读者注意：

英文术语	本书译词	英文术语	本书译词
argument	引数 （i.e. 实参）	instantiated	实体化、具现化
by reference	传址	library	程序库
by value	传值	resolve	决议
dereference	提领 （i.e. 解参考）	parameter	参数 （i.e. 形参）
evaluate	评估、核定	type	型别 （i.e. 类型）
instance	实体		
equality	相等	equivalence	等价

以下是 STL 六大组件（components）之本书译名：

英文术语	本书译词	意义
adapters	配接器	用来修饰其他组件。包括 iterator adapters、function adapters、container adapters 三大类。
allocators	配置器	用来分配空间。空间可来自于内存或磁盘 — 取决于配置器如何实现。主要用来服务容器。
algorithms	算法	就是计算机科学一般意义上的「算法」。例如 sort, bineary search, permutation…。
containers	容器	就是数据结构，用来存放元素。例如 vector, list, set, map, red black tree, hashtable…。
function object	函数对象（意义同下）	同下。
functor	仿函数（意义同上）	一种行为类似函数的东西。具体就是「重载了"function call" 操作符」的 classes。
iterators	迭代器	一种行为类似指针的东西。具体就是「重载了++, --, ->, * 等操作符」的 classes。

前 言

这不是一本讨论面向对象编程（object-oriented programming）的书。

你或许会觉得奇怪。毕竟，你可能在书店的 C++ 分类区看到这本书，也可能听过别人把面向对象与 C++ 当同一件事来讲，但是这并非 C++ 语言的唯一用途。基本上 C++ 支持多种不同的设计思维模型（paradigms），其中最新也最鲜为人知的一项就是泛型编程（generic programming）。

正如许多新的观念一样，泛型编程实际上已有很长一段历史了。早期的泛型编程研究论文大约在 25 年前就已经出现；第一个实验性质的泛型程序库并非以 C++ 写成，而是以 Ada [MS89a, MS89b] 和 Scheme [KMS88] 完成。由于泛型算法太新，所以还没有相关教科书。

第一个走出研究圈的样例显得格外重要，它就是 STL：C++ Standard Template Library。STL 是由 Alexander Stepanov（他后来成为 Hewlett-Packard 实验室的一员）和 Meng Lee 共同发展，于 1994 年被纳入 C++ 标准程序库的一部分。可免费取得之「HP STL 实现品」[SL95] 也于这一年发行，它是展示 STL 强大威力的范本。

当 Standard Template Library 开始成为 C++ 标准规格的一部分，C++ 族群立刻意识到这是一个高品质而且高效率的容器类程序库（container classes library）。要在一堆物品中找出熟悉的东西是最容易的了，而每一位 C++ 程序员都熟悉容器类。每个具相当水准的程序也都需要某种管理对象的方法，甚至每位 C++ 程序员都曾实现过 `strings`、`vectors` 或 `lists`。

C++ 早期就已经有容器类程序库的存在。这个语言加入 template classes（亦即「可参数化的型别」）之后，第一个用途 — 事实上也是引入 template 的主要原因 — 就是将容器类参数化。很多厂商包括 Borland、Microsoft、Rogue Wave 与 IBM，都有自己的程序库，其中包括 `Array<T>` 或其对等物。

容器类的观念是如此根深蒂固，以致于 STL 给人的初步印象似乎不比其他容器类程序库多了些什么。这种印象使大家疏忽了 STL 的独特性。

STL 是一种高效、泛型、可交互操作的软件组件；巨大，而且可以扩充。它包含计算机科学中的许多基本算法和数据结构，而且它把算法和数据结构完全分离开来，互不耦合。STL 不只是一个容器类程序库，更精确地说它是一个泛型算法（generic algorithms）库；容器的存在使这些算法有东西可以操作。

你可以在程序中使用现有的 STL 算法，正如你使用现有的容器一样。举例来说，当你想要使用 C 标准库中的 `qsort` 函数，你也可以使用泛型的 STL `sort`（而且 `sort` 更简单、更弹性、更安全，也更有效率）。很多书籍，包括 David Musser 与 Atul Saini 合著的 *STL Tutorial and Reference Guide* [MS96]，以及 Mark Nelson 的 *C++ Programmer's Guide to the Standard Template Library* [Nel95]，都有说明如何以这种方式使用 STL。

这实际上是非常有用的。重复运用代码总比重新撰写代码来得好，现在你可以在你的程序中重复运用既有的 STL 算法。然而这还只是以某个角度来使用 STL 而已。STL 的设计具有扩充性；也就是说你可以自行撰写一些组件，与 STL 组件交互作用，就像不同的 STL 组件之间可以交互作用一样。有效地运用 STL，意味着对它进行扩充。

泛型编程（Generic Programming）

STL 并非只是一些有用组件的集合。它还有鲜为人知而未被了解的一面：它是描述软件组件抽象需求条件的一个正规而有条理的阶层架构（formal hierarchy）。由于所有 STL 组件都是精确符合某些特定条件而写成，所以 STL 组件可以相互作用，可以扩充，你可以在增加新算法和新容器的同时，对新旧代码之间的协同作用抱持巨大信心。

计算机科学的重要进步，许多是由于发掘了新的抽象性质而促成。一个被当代所有程序语言支持的决定性抽象性质就是副例程 subroutine（又名过程 procedure，或函数 function — 不同的语言使用不同的词汇）。C++ 支持另一种抽象性质：抽象数据型别（abstract data typing，ADT）。是的，在 C++ 中，我们可以定义新的数据型别，以及该型别的基本操作行为。

代码与数据的结合形成了抽象数据型别，它必须通过一个具有明确定义的接口来操作。subroutine 是一种重要的抽象性质，因为一旦使用它，你就不需要仰赖（甚至不必知道）其实际做法；同样道理，你可以使用抽象数据型别，可以用来操作甚至产生新值，而不必在意数据的实际表现方式。唯一重要的是其接口。

C++ 也支持面向对象编程（OOP）[Boo94, Mey97]，此技术涉及「与继承相关」、「由多型（polymorphic）数据型别构成」的阶层体系。面向对象编程（OOP）拥有比抽象数据型别（ADT）更多的间接性，因而达到更进一步的抽象性。某些情况下你可以访问某值并操作之，却不必指明其精确型别。你可以撰写单一函数，用以操作继承阶层体系中的不同型别。

泛型编程（Generic Programming）意味一种新的抽象性质。其中心抽象性比早期如副例程（subroutine）或类（class）或模块（module）的抽象性更难捉摸。它是数据型别的一组需求条件。这很难让人领悟，因为它并非与 C++ 的某个性质系结在一起。C++（甚或任何当代程序语言）并没有任何关键字可用来宣告一组抽象需求条件。

对于这种「一开始令人泄气的含糊情况」，泛型编程的回报是前所未有的弹性，以及不会损及效率的抽象性。泛型编程和面向对象编程不同，它并不要求你通过额外的间接层来调用函数；它让你撰写完全一般化并可重复运用的算法，其效率与「针对某特定数据型别而设计」的算法旗鼓相当。

泛型算法抽离（抽象化）于特定型别和特定数据结构之外，使得以接受尽可能一般化的引数型别。这意味一个泛型算法实际上具有两部分：(1) 用来描述算法步骤的实际指令；(2) 正确指定「其引数型别必须满足之性质」的一组需求条件。

STL 的创新在于认知这些型别条件可以被具体指明并加以系统化。也就是说，我们可以定义一组抽象概念（所谓 *concepts*），只要某个型别满足某组条件，我们就说此型别符合某个 concept。这些 concepts 非常重要，因为算法对于其引数型别的大部分假设，都可以藉由「符合某些 concepts」以及「不同 concepts 之间的关系」来陈述。此外，这些 concepts 形成一个明确定义的阶层架构，这让人联想到传统面向对象编程中的继承（inheritance），只不过它是纯然的抽象。

这个 concepts 阶层架构是 STL 的一个概念性结构，也是 STL 最重要的部分。由于它，使得重复运用和交互操作变得可能。此一概念性结构作为软件组件的正规分类法也十分重要 — 即使没有具体代码。STL 确实包含有具象数据结构如 pair 和 list，但要有效率地运用这些数据结构，你必须了解其所依据的概念结构。

定义抽象的 concepts，并根据抽象的 concepts 来撰写算法与数据结构，是泛型编程的本质。

如何阅读本书

本书把 Standard Template Library 当作抽象概念库（library of abstract concepts）来描述。本书定义出 STL 基本的各个 concepts 与抽象性质，并指出所谓「某个型别模塑出某个 concept」是什么意思，「以某个 concept 的接口写成一个算法」又是什么意思。本书讨论 STL 涵盖的各个类和算法，并阐明如何撰写属于你自己并相容于 STL 的类和算法。此外本书还包含一份所有 STL concepts、classes、算法的完整参考手册。

每个人都应该阅读第一篇，这一部分介绍 STL 与泛型编程的主要概念。说明如何使用以及撰写一个泛型算法，并解释「算法之所以为泛型」的意义。所谓泛型（Genericity），具有「在多种数据型别上皆可操作」的含意。

探究泛型算法，自然而然便导出了 *concept*s、*modeling* 与 *refinement* 的核心观念，这些观念之于泛型编程，就像多型与继承之于面向对象编程，同样地基础，同样地重要。泛型算法作用于一维区间上，导出 STL 的数个基本概念：*iterator*s、*container*s、*function object*s。

第一篇同时也介绍了本书通用的符号及排版习惯：术语 *modeling* 和 *refinement*、*range*s 的非对称标示法，以及 concept 名称的特殊字体。

STL 定义了很多 concepts。某些 concepts 只因技术上的细节而互不相同。第一篇是个概论，概略讨论 STL concepts 的全貌。第二篇是详细的参考手册，包含每个 STL concept 的严格定义。你可能不会想要遍读第二篇；当你需要参考某个 concept 时才到第二篇查询，应该是更好的做法。（每当需要撰写符合某个 STL concept 的新型别时，你都应该参考第二篇。）

第三篇同样是参考手册，提供 STL 预定义的算法和类的说明。这一部分仰赖第二篇的「concept 定义」甚多。STL 所有的算法和几乎所有的具象型别都是 templates，每个 template 的参数都能以某种 concept 的 model 加以描述。第三篇的定义可以和第二篇对应的章节交互参考。

理想状况下，本书到达第三篇就可以结束了。不幸的是现实需求使我必须写出更多章节：一份有关可移植性议题的附录。STL 问世之初并没有可移植性问题，因为只有一份实现品存在。这种情况已不复存在。一旦某种语言或程序库有一个以上的实现品存在，任何关心可移植性的人都必须知道每个实现品之间的差异。

你仍然可以从 anonymous FTP 站台 `butler.hpl.hp.com` 下载早期的 HP 实现品，但此作品已无人维护。Silicon Graphics Computer Systems (SGI) 有一份较新的免费作品，可以从 `http://www.sgi.com/Technology/STL` 下载。至于 SGI STL 在不同编译器上的移植版本，可以从 `http://www.metabyte.com/~fbp/stl` 取得。此外这个世界还存在有其他商业化 STL 实现品。

如果你正在撰写真正的程序，光了解函数库的设计理论是不够的；你还必须知道不同的 STL 实现品以及不同的 C++ 编译器之间的差别。这些不怎么吸引人但却必要的细节，构成了附录 A 的主题。

谁该阅读本书

虽然本书谈的几乎都是以 C++ 写成的算法，但这不是一本算法导入型教科书，也不是一本 C++ 语言教本。本书确实对于上述两方「不为人熟知的某些观点」有所解释。更明确地说，由于 STL 对 templates 的使用方式迥异其他 C++ 程序，所以本书讨论了某些 templates 高阶技术。本书不应该是你的第一本 C++ 书籍，也不应该是你的第一本算法分析入门书。你应该知道如何撰写基本的 C++，同时也应该知道 $O(N)$ 表示法的意义。

算法与数据结构方面，两本标准参考书是 Donand Knuth 的 *The Art of Computer Programming* [Knu97, Knu98a, Knu98b] 和 Cormen, Leiserson, and Rivest 的 *Introduction to Algorithm* [CLR90]。两本最佳的 C++ 入门书籍则是 Bjarne Stroustrup 的 *The C++ Programming Language* [Str97] 和 Stanley Lippman, and Josee Lajoie 的 *C++ Primer* [LL98]。

本书由来

我于 1996 年加入 Silicon Graphics Computer Systems 的编译器研发团队。那时候 Alex Stepanov 已经离开 HP 并加入 SGI 数个月了。当时的 SGI C++ 编译器并不包含 Standard Template Library 实现品。以 HP 的原始作品为基础，Alex、Hans Boehm 和我开始撰写一个新的 STL 版本，用来和 SGI MIPSpro 编译器 7.1（以及后续版本）搭配出货。

SGI Standard Template Library [Aus97] 包含很多崭新的扩充性质，例如高效且「对多线程而言安全（thread-safe）」的内存配置能力、散列表（hash tables），以及某些算法的改良。这些增强功能如果留做私用，对 SGI 的客户而言没有任何价值，所以 SGI STL 把它们全部免费公开。所有源码及文件都放在 `http://www.sgi.com/tech/stl`。

网路上的这份文件以网页形式呈现，将 STL 的概念结构视为核心，描述组成此结构之各个抽象概念（concepts），并以这些 concepts 来说明 STL 的算法和数据结构。我们收到很多要求，希望能够扩充此文件，本书就是对这些要求的回应。本书的「参考手册」部分，也就是二、三两篇，是 SGI STL 网页的分支。

整个网页是为 SGI 而写，其版权属于 SGI 所有。我在我所属的管理部门的宽容许可下使用那些网页内容于本书上。

致谢

首先，也是最重要的，如果没有 Alex Stepanov 的努力，这本书不可能诞生。Alex 参与这本书的每个阶段：他把我带进 SGI，我对泛型编程的每一个知识几乎都是他教导的，他参与 SGI STL 及其网页的开发，并鼓励我将网页的内容写成一本书。我由衷感谢 Alex 的帮助与鼓励。

我也要感谢 Bjarne Stroustrup 和 Andy Koenig 帮助我了解 C++，并感谢 Dave Musser 对于泛型编程、STL 及本书的诸多贡献（其中有些贡献可自参考书目中找到）。Dave 使用早期的 SGI STL 网页内容作为其课程教材，而通过他和他的学生的意见，这个网页有了很大的改进。

同样地，本书通过校阅者的意见而有了很大的改进，校阅者包括 Tom Becker、Steve Clamage、Jay Gischer、Brian Kernighan、Andy Koenig、Angelika Langer、Dave Musser、Sibylla Schupp 和 Alex Stepanov，

他们阅读过本书的每个版本。通过他们的协助，这本书更为清楚，错误也减少了许多。剩下的任何错误都是我的责任。

本书第一刷和第二刷中的许多错误已经更正，我要感谢 Sam Bradsher、Bruce Eckel、Guy Gascoigne、Ed James-Beckham、Jon Jagger、Nate Lewis、Shawn D. Pautz、John Potter、George Reilly、Manos Renieris、Peter Roth、Andreas Scherer 和 Jürgen Zeller，使我注意到这些错误。

我也要感谢 Addison-Wesley 的工作人员，包括 John Fuller、Mike Hendrickson、Marina Lang 与 Genevieve Rajewski，他们在我写作的过程中指导我。感谢 Karen Tongish 细心的编辑。

最后，我要感谢我的未婚妻 Janet Lafler 给我的爱与支持，以及对我在很多夜晚和周末写作的容忍。

我们的猫儿 Randy 与 Oliver，在我的键盘上走来走去试着想帮助我，不过最后我删掉了它们大部分的贡献。

目 录

译序（侯捷） i
前言 xv

第一篇 泛型编程导入 1
第 1 章 STL 巡礼 3
1.1 一个简单的例子 3
1.2 总结 7

第 2 章 算法与区间 9
2.1 线性查找（Linear Search） 9
2.1.1 以 C 完成线性查找 10
2.1.2 Ranges（区间） 12
2.1.3 以 C++ 完成线性查找 13
2.2 Concepts 和 Modeling 16
2.3 Iterators（迭代器，泛型指针） 19
2.3.1 Input Iterators 20
2.3.2 Ouput Iterators 22
2.3.3 Forward Iterators 24
Constant（不变的）Iterators 和 Mutable（可变的）Iterators 27
2.3.4 Bidirectional Iterators 27
2.3.5 Random Access Iterators 28
2.4 Refinement（精炼、强化） 29
2.5 总结 31

第 3 章 再论 Iterators（迭代器 or 泛型指针） 33
3.1 Iterator Traits（迭代器特征）与 Associated Types（相关型别） 33
3.1.1 Value Type（数值型别） 33

3.1.2 Difference Type（差距型别） 36
3.1.3 Reference Type 和 Pointer Type 37
3.1.4 算法的处理与 Iterator Tags 38
3.1.5 把一切统合起来 41
3.1.6 没有 iterator_traits，如何制作 Iterator Traits（迭代器特征） 43
3.2 定义新组件（New Components） 44
3.2.1 Iterator Adapters 46
3.2.2 定义 Iterator 时的建议 47
3.2.3 定义算法时的建议 47
3.3 总结 48

第 4 章 Function Objects（函数对象） 49
4.1 将线性查找一般化 49
4.2 Function Object Concepts（函数对象概念） 52
4.2.1 单参（Unary）与双参（Binary）Function Objects 52
4.2.2 Predicates 和 Binary Predicates 53
4.2.3 相关型别（Assocated Types） 54
4.3 Function Object Adapters（函数对象配接器） 56
4.4 预定义的 Function Objects 58
4.5 总结 58

第 5 章 Containers（容器） 59
5.1 一个简单的 Container 59
5.1.1 一个 Array Class 60
5.1.2 它是如何运作的 63
5.1.3 最后讨论 63
5.2 Container Concepts 67
5.2.1 元素的容纳（Containment of Elements） 68
5.2.2 Iterators 68
5.2.3 Containers 的阶层架构（Hierarchy） 70
5.2.4 最平淡无奇的 Container 71
5.3 大小可变的 Container Concepts 72
5.3.1 Sequences（序列） 73
其他形式的 insert 与 erase 74
安插（Insertion）于开头（Front）和尾端（Back） 74
安插（Insertion）语义和覆盖（Overwrite）语义 75

5.3.2 Associative Containers（关联式容器） 75
5.3.3 Allocators（配置器） 78
5.4 总结 78
5.4.1 我们应该使用什么样的 Container? 78
5.4.2 设计你自己的 Container 79

第二篇 参考手册：STL Concepts 81
第 6 章 基本概念 83
6.1 Assignable 83
6.2 Default Constructible 84
6.3 Equality Comparable 85
6.4 可序性（Ordering） 86
6.4.1 LessThan Comparable 86
6.4.2 Strict Weakly Comparable 88

第 7 章 Iterators（迭代器 or 泛型指针） 91
7.1 Trivial Iterator 91
7.2 Input Iterator 94
7.3 Output Iterator 96
7.4 Forward Iterator 100
7.5 Bidirectional Iterator 102
7.6 Random Access Iterator 103

第 8 章 Function Objects（函数对象） 109
8.1 基本的 Function Objects 110
8.1.1 Generator 110
8.1.2 Unary Function 111
8.1.3 Binary Function 112
8.2 Adaptable Function Objects 113
8.2.1 Adaptable Generator 113
8.2.2 Adaptable Unary Function 114
8.2.3 Adaptable Binary Function 115
8.3 Predicates 116
8.3.1 Predicate 116
8.3.2 Binary Predicate 117
8.3.3 Adaptable Predicate 118
8.3.4 Adaptable Binary Predicate 119
8.3.5 Strict Weak Ordering 119

8.4 特化的 Concept 122
8.4.1 Random Number Generator 122
8.4.2 Hash Function（散列函数） 123

第 9 章 Containers（容器） 125
9.1 General Container Concepts 125
9.1.1 Container 125
9.1.2 Forward Container 131
9.1.3 Reversible Container 133
9.1.4 Random Access Container 135
9.2 Sequence（序列；循序式容器） 136
9.2.1 Sequence 136
9.2.2 Front Insertion Sequence 141
9.2.3 Back Insertion Sequence 143
9.3 Associative Containers（关联式容器） 145
9.3.1 Associative Container 145
9.3.2 Unique Associative Container 149
9.3.3 Multiple Associative Container 152
9.3.4 Simple Associative Container 153
9.3.5 Pair Associative Container 155
9.3.6 Sorted Associative Container 156
9.3.7 Hashed Associative Container 161
9.4 Allocator（空间配置器） 166

第三篇 参考手册：算法与类 **173**
第 10 章 基本组件 175
10.1 `pair` 175
10.2 Iterator 基本要素（Iterator Primitives） 177
10.2.1 `iterator_traits` 177
10.2.2 Iterator Tag Classes 179
10.2.3 `distance` 181
10.2.4 `advance` 183
10.2.5 Iterator Base Class 185
10.3 `allocator` 187
10.4 内存管理基本要素（Memory Management Primitives） 189
10.4.1 `construct` 189
10.4.2 `destroy` 190

10.4.3 uninitialized_copy 192
10.4.4 uninitialized_fill 194
10.4.5 uninitialized_fill_n 195
10.5 临时缓冲区（Temporary Buffers） 196
10.5.1 get_temporary_buffer 197
10.5.2 return_temporary_buffer 198

第 11 章 「不改变操作对象之内容」的算法 199
11.1 线性查找（Linear Search） 199
11.1.1 find 199
11.1.2 find_if 200
11.1.3 adjacent_find 202
11.1.4 find_first_of 204
11.2 子序列匹配（Subsequence Matching） 206
11.2.1 search 206
11.2.2 find_end 209
11.2.3 search_n 211
11.3 计算元素个数（Counting Elements） 214
11.3.1 count 214
11.3.2 count_if 216
11.4 for_each 218
11.5 比较两个 Ranges 220
11.5.1 equal 220
11.5.2 mismatch 222
11.5.3 lexicographical_compare 225
11.6 最大值与最小值 227
11.6.1 min 227
11.6.2 max 228
11.6.3 min_element 229
11.6.4 max_element 231

第 12 章 「会改变操作对象之内容」的算法 233
12.1 拷贝某个区间（Copying Ranges） 233
12.1.1 copy 233
12.1.2 copy_backward 236
12.2 互换元素（Swapping Elements） 237
12.2.1 swap 237

12.2.2 iter_swap 238
12.2.3 swap_ranges 239
12.3 transform 240
12.4 替换元素（Replacing Elements） 243
12.4.1 replace 243
12.4.2 replace_if 244
12.4.3 replace_copy 246
12.4.4 replace_copy_if 248
12.5 充填整个区间（Filling Ranges） 249
12.5.1 fill 249
12.5.2 fill_n 250
12.5.3 generate 251
12.5.4 generate_n 252
12.6 移除元素（Removing Elements） 253
12.6.1 remove 253
12.6.2 remove_if 255
12.6.3 remove_copy 256
12.6.4 remove_copy_if 258
12.6.5 unique 259
12.6.6 unique_copy 262
12.7 排列算法（Permuting Algorithms） 264
12.7.1 reverse 264
12.7.2 reverse_copy 265
12.7.3 rotate 266
12.7.4 rotate_copy 268
12.7.5 next_permutation 269
12.7.6 prev_permutation 271
12.8 分割（Partitions） 273
12.8.1 partition 273
12.8.2 stable_partition 274
12.9 随机重排与抽样（Random Shuffling and Sampling） 275
12.9.1 random_shuffle 276
12.9.2 random_sample 277
12.9.3 random_sample_n 279
12.10 一般化之数值算法（Generalized Numeric Algorithms） 281
12.10.1 accumulate 281
12.10.2 inner_product 283
12.10.3 partial_sum 285

12.10.4 adjacent_difference 287

第 13 章 排序和查找 291
13.1 对某个区间排序（Sorting Ranges） 291
13.1.1 sort 292
13.1.2 stable_sort 294
13.1.3 partial_sort 297
13.1.4 partial_sort_copy 300
13.1.5 nth_element 301
13.1.6 is_sorted 303
13.2 sorted ranges 上的操作行为 305
13.2.1 二分查找法（Binary Search） 305
13.2.1.1 binary_search 306
13.2.1.2 lower_bound 308
13.2.1.3 upper_bound 310
13.2.1.4 equal_range 313
13.2.2 合并（Merging）两个 Sorted Ranges 316
13.2.2.1 merge 316
13.2.2.2 inplace_merge 318
13.2.3 在 Sorted Ranges 身上执行集合（Set）相关操作 320
13.2.3.1 includes 321
13.2.3.2 set_union 324
13.2.3.3 set_intersection 327
13.2.3.4 set_difference 330
13.2.3.5 set_symmetric_difference 333
13.3 堆的相关操作（Heap Operations） 336
13.3.1 make_heap 336
13.3.2 push_heap 338
13.3.3 pop_heap 339
13.3.4 sort_heap 342
13.3.5 is_heap 343

第 14 章 Iterator Classes（迭代器类） 345
14.1 Insert Iterators 345
14.1.1 front_insert_iterator 345
14.1.2 back_insert_iterator 348
14.1.3 insert_iterator 351
14.2 Stream Iterators 354

14.2.1 istream_iterator 354
14.2.2 ostream_iterator 357
14.2.3 istreambuf_iterator 359
14.2.4 ostreambuf_iterator 362
14.3 reverse_iterator 363
14.4 raw_storage_iterator 368

第 15 章 Function Object Classes（函数对象类） 371
15.1 Function Object Base Classes 371
15.1.1 unary_function 371
15.1.2 binary_function 372
15.2 算术运算（Arithmetic Operations） 373
15.2.1 plus 373
15.2.2 minus 375
15.2.3 multiplies 376
15.2.4 divides 378
15.2.5 modulus 379
15.2.6 negate 380
15.3 大小比较（Comparisons） 382
15.3.1 equal_to 382
15.3.2 not_equal_to 383
15.3.3 less 384
15.3.4 greater 386
15.3.5 less_equal 387
15.3.6 greater_equal 388
15.4 逻辑运算（Logical Operations） 390
15.4.1 logical_and 390
15.4.2 logical_or 391
15.4.3 logical_not 393
15.5 证同（Identity）与投射（Projection） 394
15.5.1 identity 394
15.5.2 project1st 395
15.5.3 project2nd 397
15.5.4 select1st 398
15.5.5 select2nd 399
15.6 特殊的 Function Objects 400
15.6.1 hash 400
15.6.2 subtractive_rng 402

15.7 Member Function Adapters 403
15.7.1 mem_fun_t 404
15.7.2 mem_fun_ref_t 406
15.7.3 mem_fun1_t 408
15.7.4 mem_fun1_ref_t 410
15.7.5 const_mem_fun_t 412
15.7.6 const_mem_fun_ref_t 414
15.7.7 const_mem_fun1_t 416
15.7.8 const_mem_fun1_ref_t 418
15.8 其他的 Adapters 421
15.8.1 binder1st 421
15.8.2 binder2nd 422
15.8.3 pointer_to_unary_function 424
15.8.4 pointer_to_binary_function 426
15.8.5 unary_negate 428
15.8.6 binary_negate 429
15.8.7 unary_compose 431
15.8.8 binary_compose 433

第 16 章 Container Classes（容器类） 435
16.1 序列（Sequences） 435
16.1.1 vector 435
16.1.2 list 441
16.1.3 slist 448
16.1.4 deque 455
16.2 Associative Containers（关联式容器） 460
16.2.1 set 461
16.2.2 map 466
16.2.3 multiset 473
16.2.4 multimap 478
16.2.5 hash_set 484
16.2.6 hash_map 488
16.2.7 hash_multiset 494
16.2.8 hash_multimap 499
16.3 Container Adapters 504
16.3.1 stack 505
16.3.2 queue 507
16.3.3 priority_queue 510

附录 A 可移植性与标准化 515
A.1 语言上的变动 516
A.1.1 Template 编译模型（Compilation Model） 516
A.1.2 带缺省值的 Template 参数（Default Template Parameters） 517
A.1.3 Member Templates 518
A.1.4 偏特化（Partial Specialization） 519
A.1.5 新加入的关键字 521
A.2 程序库的变动 524
A.2.1 Allocator 524
A.2.2 Container Adapters 525
A.2.3 次要的程序库变动 526
A.3 命名及包装（Naming and Packaging） 527

参考书目 531
索引 535

第一篇

泛型编程 导入

Introduction to Generic Programming

第 1 章

STL 巡礼

A Tour of the STL

计算机程序设计是一种大量涉及算法及数据结构的工作。不论写什么样的程序，数据的组织及处理都是必备的功夫。

C++ Standard Template Library（标准模板程序库，STL）可让你重复运用既有的算法，而不必在环境类似的情况下一再重新撰写相同代码。STL 算法是泛型的（generic），不与任何特定数据结构或对象型别系缚在一起，但是却像为你量身订做一样，有很高的效率。STL 甚至是可扩充的，是的，就像 STL 组件（components）彼此之间可以相互配合运用一样，STL 组件亦可以和你所写的组件水乳交融地搭配运用。

1.1 一个简单的例子

要见识 STL 的威力，最好的方法就是看一个 STL 简单运用实例。我以一个简化版的 UNIX sort（排序）工具程序作为例子，它由标准输入流（standard input stream）一行行读入数据，然后将排序后的数据一行行写到标准输出流（standard output stream）。

这里需要三个步骤：(1) 从标准输入设备读入文本数据，并予以断行；(2) 对这些数据排序；(3) 将排序后的数据写到标准输出设备上。每个步骤都是一个标准算法，运用 STL 将它们组合起来，非常简单：

```
int main() {
  vector<string> V;
  string tmp;

  while (getline(cin, tmp))
    V.push_back(tmp);

  sort(V.begin(), V.end());
  copy(V.begin(), V.end(), ostream_iterator<string>(cout, "\n"));
}
```

我首先产生一个空的 vector（这是最简单也最常用的一种 STL 容器），每次加入一行文本，然后将 vector 排序，最后再将其内容复制到标准输出设备上。sort 缺省以升幂排序，但是只要这么写：

```
sort(V.begin(), V.end(), greater<string>());
```

它就会改以降幂顺序来为每一行排序。

程序的最后一行特别有趣。我用了 STL 算法 copy，它与「用来将某数组复制到另一数组」的算法别无二样。其引数是所谓的 iterator（迭代器，所有 STL 算法亦都需要这种引数）。iterator 的适用范围可以从普通的 C 指针，到本例所展示之「输出数据流的外层包装（output stream wrapper）」。你可以利用 copy，将任何数据来源端的元素复制到任何数据目标端去（我会在第 2 章给一个更精确的定义），当然你也可以把它应用于输出。

你也可以将 copy 应用于输入，前提是选用正确的 iterator。你需要一个能够「从输入流一次返回一行」的 iterator。STL 并没有提供这样的 iterator，不过动手写一个并非难事，而且一旦你写了这么一个，它不只可以用在 copy，也可以用在其他许多 STL 算法身上。

```
class line_iterator
{
  istream* in;
  string line;
  bool is_valid;
  void read() {
    if (*in)
      getline(*in, line);
    is_valid = (*in) ? true : false;
  }

public:
  typedef input_iterator_tag iterator_category;
  typedef string value_type;
  typedef ptrdiff_t difference_type;
  typedef const string* pointer;
  typedef const string& reference;

  line_iterator() : in(&cin), is_valid(false) {}
  line_iterator(istream& s) : in(&s) { read(); }
  reference operator*() const { return line; }
  pointer operator->() const { return &line; }
  line_iterator operator++() {
    read();
    return *this;
  }
  line_iterator operator++(int) {
    line_iterator tmp = *this;
    read();
    return tmp;
  }
```

```
  bool operator==(const line_iterator& i) const {
    return (in == i.in && is_valid == i.is_valid) ||
           (is_valid == false && i.is_valid == false);
  }

  bool operator!=(const line_iterator& i) const {
    return !(*this == i);
  }
};
```

如果你目前还无法了解 line_iterator 的定义，不必担心，其中有些只是为了技术上的因素才这么做的（7.2 节有 iterator 的完整定义需求）。一旦你有了这么一个 class，我们的排序程序甚至变得更简单了。我们不再需要循环，可以直接以某个区间的 iterators 产出一个 vector：

```
int main() {
  line_iterator iter(cin);
  line_iterator end_of_file;
  vector<string> V(iter, end_of_file);
  sort(V.begin(), V.end());
  copy(V.begin(), V.end(), ostream_iterator<string>(cout, "\n"));
}
```

另一种做法

这是一个挺吸引人的简单方法，用来进行「文本行」的读取和排序，但这是最好的方法吗？或许不是。问题出在对某一区间内的对象进行排序，意味着会引发大量的赋值（assignment）动作。string 是个大而复杂的数据结构，将某个 string 的值赋予另一个 string，速度可能很慢。我们真正想要的应该是对「文本行」的指针做排序，而非「文本行」自身。（你可能会以特别的技巧来加速 string 的赋值动作，然而这不一定有效。不管怎么说，最好别倚赖太多特殊技巧。）

如果你的程序使用大量字符串，最好是不要使用任何字符串 class。比较有效率的做法是将所有字符串放到一个「字符串表格」中，当你要需要用到特定某字符串时，则以「指入该字符串表格内」的一个指针来代替。

这个方法比起单纯使用 vector<string> 来得复杂，但 STL 仍能与之完美兼容。先前我们只使用一个 vector<string> 来存放字符串，现在则需要两个 vectors，一个当作字符串表格使用，另一个用来存放「指向字符串表格的指针」。我们先把字符串表格建立起来，然后产生存放文本行的 vector。考虑到大多数 STL 算法的运作方式，如果我们以一对「指入字符串表格内」的 iterators 来表现一个「文本行」，将会十分方便。其中一个 iterator 指向开头字符，另一个指向最后字符的下一个位置。

由于我们以一对 iterators 来表现一行文本，现在我们必须让 sort 知道如何比较这样的元素 — 因为现在 "<" 操作符不再管用了。这没什么难的，因为我们已经知道，有一个 sort 版本可接受一个额

外引数，此乃正是因为常有需要依据某种特别方式来比较元素之故。我们需要做的便是撰写一个用来进行比较动作的 function object，当作引数传给 sort。我们还可以顺手写出另一个 function object 以协助输出。

```
struct strtab_cmp
{
  typedef vector<char>::iterator strtab_iterator;

  bool operator()(const pair<strtab_iterator, strtab_iterator>& x,
                  const pair<strtab_iterator, strtab_iterator>& y) const {
    return lexicographical_compare(x.first, x.second, y.first, y.second);
  }
};

struct strtab_print
{
  ostream& out;
  strtab_print(ostream& os) : out(os) {}

  typedef vector<char>::iterator strtab_iterator;
  void operator()(const pair<strtab_iterator, strtab_iterator>& s) const {
    copy(s.first, s.second, ostream_iterator<char>(out));
  }
};

int main()
{
  vector<char> strtab;  // 产生字符串表格
  char c;
  while (cin.get(c)) {
    strtab.push_back(c);
  }
                        // 剖析字符串表格，予以断行
  typedef vector<char>::iterator strtab_iterator;
  vector<pair<strtab_iterator, strtab_iterator> > lines;
  strtab_iterator start = strtab.begin();
  while (start != strtab.end()) {
    strtab_iterator next = find(start, strtab.end(), '\n');
    if (next != strtab.end())
      ++next;
    lines.push_back(make_pair(start, next));
    start = next;
  }
                        // 对于文本行所形成的 vector 排序
  sort(lines.begin(), lines.end(), strtab_cmp());

                        // 将排序结果写到标准输出设备上
  for_each(lines.begin(), lines.end(), strtab_print(cout));
}
```

1.2 总结

正如这两个简单例子所示，我们已经勾勒出 STL 的概观了。STL 包含各种泛型算法（***algorithms***）如 sort、find 和 lexicographical_compare；泛型指针（***iterators***）如 istream_iterator 和 ostream_iterator；泛型容器（***containers***）如 vector；以及 ***function objects*** 如 less 和 greater。

这些例子同时也显示运用 STL 时的几个最重要的观念。

- 所谓使用 STL，就是去扩充它。上述两个例子中，我们各自完成了自己的 STL 兼容组件：第一个例子完成的是个新品种的 iterator，第二个例子则完成两个新型的 function objects。（我们甚至可以更进一步写一个一般化的 parsing iterator，取代第二个例子中以循环剖析字符串表格的做法。）
- STL 的算法和容器是独立分离的。任何新增的算法并不要求你改写任何容器，我们可以随心所欲地将任何算法与任何容器匹配混用。假设有 N 个不同的算法，M 种不同的容器，你不需要负担 N ×M 个不同的实现代码。
- 无须继承（inheritance），STL 便具备可扩充、可定制的特性。新的 iterators（如上例的 line_iterator）与新的 function objects（如上例的 strtab_cmp）无须继承自任何特殊基类。它们只需满足某些抽象条件。你无须承担「继承方面之任何实现细节所造成的额外开销」，便可以扩充 STL，并且不必为造成「fragile base class」的问题大伤脑筋。
- 抽象化并不意味效率低。泛型算法如 sort 和 copy，以及泛型容器如 vector，其效率一如为你手工精制一般。

STL 提供了一种新的程序设计思维方式，其中算法与抽象条件组（abstract sets of requirements）居于中心地位。抽象条件是 STL 与这本书的基础所在。

第 2 章

算法与区间

Algorithms and Ranges

第 1 章中我做了些巧妙的事情。我向你显示，扩充 STL 有多么容易，撰写与 STL 组件相互运用的私人组件又是多么容易。只要你知道你需要满足的条件，事情就很好办了。然而，除非你知道什么是 iterator，否则就算你知道 STL 算法正是以 iterators 为基础，也是枉然。

本章介绍「在线性区间（linear range）内运作」的算法的重要概念。所谓线性区间内的元素，意指它们是个一维集合，具有「第一」及「最后」的概念，每个元素（除了最后一个）都有后继者。只要以「一个接一个」的方式，便能遍历整个集合，取得从头至尾的所有元素。这些算法是 STL 的核心成分。

和算法一样「与 *range* 有紧密关系」的，是一整组 iterator 概念。本章也会一并介绍。Iterator 可说是 STL 最重要的一个创新发明，它使「将算法与其相关数据结构之间的关系切割分离」一事变得可能。以算法（*algorithms*）和 *iterators* 为基础，再来了解其他 STL 成分（例如第 4 章的 *function objects* 和第 5 章的 *containers*）就很容易了。

本章讨论算法及 iterators 背后的重要观念，并利用这些观念来阐述所谓的 *concepts*、*modeling* 与 *refinement*。即使你已熟稔 iterators 与泛型算法，你也应该好好地阅读本章，因为本章所介绍的观念会不断出现在往后章节中。

虽然本章介绍算法及 iterators，不过其细部文档则放到本书的 II、III 篇中：*Iterator concepts* 在第 7 章，STL 算法放在 11~13 章。

2.1 线性查找（Linear Search）

线性查找（或 Knuth 所称的「循序查找」[Knu98b]）算是最基本的一种算法了：在线性集合中找寻某个具有特定数值的元素。如果元素是以某种特别的方式组织排列，我们便可以使用更复杂、更特别的算法来查找它。线性查找是最普通的查找算法，它对被查找的数据结构作最少的假设。

2.1.1 以 C 完成线性查找

如何将线性查找写成泛型算法？在此之前，让我们看看一个特殊而熟悉的例子：标准 C 函数库中的 strchr 函数。它的函数形式为：

```
char* strchr(char* s, int c);
```

strchr 会在字符串 s 中查找数值 c。底下是一份相仿的实现代码：

```
char* strchr(char* s, int c)
{
  while (*s != '\0' && *s != c)
    ++s;
  return *s == c ? s : (char*) 0;
}
```

线性查找的一般概念十分简单：从头到尾一步步依序检查某特定数值是否出现在此线性序列中。如果目前元素值与该特定数值相同，此算法便返回目前元素在此序列的位置，否则就移至下一个元素。如果没有下一个元素（也就是说已经到了序列尾端），就返回某个值表示查找失败。

欲实现任何线性查找，你必须有能力回答下列问题：

- 如何指明我们想要查找的序列？
- 如何表示序列中的位置？（我们必须知道这一点，才能标示「当前元素」，也才能返回查找成功时的位置。）
- 如何移动到下一个元素？
- 如何知道我们已经抵达序列尾端了？
- 用来表示查找失败的返回值是什么？

strchr 是以 C 语言写成，我们所要查找的序列是一个字符串：以 null 字符结束的字符数组。我们以指针指明这个字符串，作为传入 strchr 的第一个引数。由于字符串最后字符是 null 字符，所以 strchr 知道哪里是字符串结尾。字符串中的位置自然也是以指针的形式表示；strchr 返回「与 c 等值的第一个元素」的指针，或是代表查找失败的 null 指针。

这个函数虽然有用，却未能如它所应该表现的那么一般化。部分问题出在它只能用来查找 chars 数组中的某个 char（或说某个 unsigned char 或 wchar_t）。另一个问题和型别记号无关：strchr 的接口限制它只能查找一种特定的数据结构。

问题出在：strchr 的引数必须是个指针，指向以 null 结尾的字符数组。这使它无法一般化。C 语

言的字符串常常是以「null 为结尾之字符数组」表示，但即使在 C 语言中，其他（字符串）表示法也是同等重要的。举例来说，如何表示子字符串？以 null 结尾之数组，其子字符串并非也是「以 null 结尾之数组」，所以必须有其他表示法。

两个明显的方法可以修改 strchr 接口。一是提供一个整数计数：在「以 s 为始」之数组的「前 n 个字符」中查找。另一种选择是提供一个指针，指向所要查找的数组尾端。结果显示，后者比较方便，于是我以此形式应用于 find1（strchr 的一般化版本）身上。

为什么「指向序列结尾的指针」比「整数计数法」来得恰当？有许多原因。第一，至少那是可以实现查找算法的做法；累加指针值比维护一个分离的计数值容易多了。第二，指针比较具有一致性。如果序列的开头、结尾以及函数返回值都具有相同表示法，那么「以某一查找结果作为下一查找运算的输入值」就容易多了。第三，也是最重要的一点，当我们将 find1 函数更进一步一般化时，往后章节会证明指针的使用有其必要。

```
char *find1(char* first, char* last, int c)
{
  while (first != last && *first != c)
    ++first;
  return first;
}
```

这份 find1 函数码使用一般 C 语言的习惯，以循环检查数组，以「指针越过数组尾端」作为结束条件。循环一遇到「first 等于 last」的情况就结束，last 自身则从未被提领过（dereferenced）。也就是说，find1 从 first 所指之元素开始一一往下查找，至 last 为止，但不包含 last。

若要走遍整个字符数组来查找目标，你可以这么写：

```
char A[N];
...
char* result = find1(A, A+N, c);
if (result == A + N) {
  // 查找失败
}
else {
  // 查找成功
}
```

我们以指针 A+N 来表示数组结尾，A+N 并不指向任何数组中的元素，因为数组元素是从 A[0] 至 A[N-1]。当循环到达 A+N，表示已经检查了数组中的每一个元素，这时候应该结束循环。

这里导出了一个重要的技术关键。一般而言，C 语言对于指针运算是很挑剔的。举例来说，你不能执行 p<q 这样的比较动作，除非 p 与 q 指向同一个数组。同样道理，你不可对非法指针（例如 null

指针）做数值运算。那么，为什么我们在本例中要使用看似非法的指针 A+N 呢？

C 语言在这里有个特殊规则（C++ 也一样），也因为这个特殊规则，find1 代码正好可以运作得当。典型的循环是由 0 检查至 N-1，所以「past-the-end（越过尾端）」的指针如 A+N 是有效的，但这仅仅适用于特殊情况。每个 C 语言数组都有一个指针会「越过尾端」，你不能提领其值，因为实际上它没有指向任何东西，你也不可累加这个指针，因为每个数组只有这么一个越过尾端的指针，但是你可以利用它来做比对动作和指针运算。（相较之下，数组倒是没有所谓「before-the-beginning（开头之前）」的指针；换句话说表达式 A-1 在 C 语言中并不合法。或许有时候 A-1 不会发生问题，但有时候会引发悲惨的错误。）

因此，C 语言有三种不同种类的指针：(1) 普通而有效的指针如 &A[0]，可对它做提领（dereference）动作；(2) 非法指针如 NULL（又称为 *singular* 指针）；以及 (3)「past-the-end」指针，不可进行提领动作，但可用于指针运算。

2.1.2 Ranges（区间）

在 find1 函数中，我们通过从 first 至 last（但不包含 last）的所有指针，进行查找。这类接口常常出现于 STL 之中，因此有种特别的标记法来表示它们。*range* [first,last) 表示「从 first 至 last（但不包含 last）的所有指针」。这种标记法源于数学领域。非对称形式是为了强调 [first,last) 乃为一个半开区间，包含 first，但不包含 last。

有时 [first,last) 标记法是指 first, ..., last-1 指针，有时是指 *first, ..., *(last-1) 元素。通常从上下文就可以判断它指的是什么，少数必要情况下我们会以 *range of pointers*（指针区间）[1]或 *range of elements*（元素区间）来分辨之。

如果 [first,last) 中的所有指针都可提领，而且从 first 出发后可达 last（也就是将 first 累加有限次，最终会抵达 last），那么 range [first,last) 便是个有效区间。所以 range [A,A+N) 是个有效区间，range [A,A)（空区间）也有效。然而 range [A+N,A) 无效，因为 A+N 落在 A 之后，而非之前，所以提领 A+N 至 A 之间的元素是不合理的。

一般而言，range 满足下列性质：

1. 对任何指针 p 而言，empty range [p,p)（空区间）是有效区间。
2. 如果 [first,last) 有效，而且不为空，那么 [first+1, last) 也必有效。
3. 如果 [first,last) 有效，mid 为满足以下条件之任何指针：first 出发后可达 mid，mid 出发后

[1] 很快我们还会归纳出 *range of iterators*（迭代器区间）。其背后概念是一样的。

可达 last，那么 [first,mid) 和 [mid,last) 亦皆有效。

4. 反方向说，如果 [first,mid) 与 [mid,last) 皆有效，则 [first,last) 亦必有效。

为什么我们使用如此奇怪而不对称的 range 定义，让左端涵盖端点而右端不涵盖其端点呢？会这么定义，其实是为了便利性。「半开区间标记法」或许一开始看起来陌生，但到后来，不论是为了公式化定义，或是为了让算法的撰写和使用更通俗，这样的标记法都更容易些。

在 find1 函数中，我们做了些许扩充。由于 last 并不涵盖于 [first,last) 内，所以 find1 的 while 循环设计相当简单。此外，正如 Andrew Koenig [Koe89] 指出，非对称性 range 有助于避免「相差一个（off-by-one）」的错误：[first,last) 中的元素个数，正如你所期望的是 last-first。这点在 C 和 C++ 中特别方便，因为其数组系由 0（而非 1）开始：N 个元素的数组以 range [A, A+N) 描述之，数组的最后一个元素落在 A[N-1] 身上。以半开区间定义 range，让我们得以这样表示 empty range：[A,A)。这和一般的 range 表示法别无二致，只不过它未含任何元素而已。是故 find1(A, A, c) 理所当然地合法。

原则上，range 依此方式而定义，因为如果某个 range 含有 n 个元素，我们必须能够取用 n+1 个（而不单单只是 n 个）不同的位置。例如 find1 函数必须返回一个特别的值以表示查找失败。另一点也很重要：我们不只需要参考到 range 内实际存在的元素，我们还必须能够参考到「新元素的可安插位置」。在元素个数为 n 的 range 内，会有 n+1 个「新元素可安插位置」：开头、结尾以及 n-1 个元素间隔。

2.1.3 以 C++ 完成线性查找

find1 函数只能用于查找型别为 char 的数组。在 C++ 中，我们可利用 template 将函数的引数型别参数化，这是一个明显而直觉的泛型方法（generalization）。我们可以设计一个类似 find1 的函数，以「任意型别 T」的指针作为参数，代替原来的 char 指针。这个假想函数 find2 可声明如下：

```
template <class T>
T* find2(T* first, T* last, T value);
```

STL 的泛型做法不像上述那般显而易见。STL 的线性查找算法不是 find2，而是 find：

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

为何采用 find，而不采用比较浅显的 find2 呢？这样可以得到什么好处？

好处是，这样的函数可以比 find2 更一般化。find2 要求参数 first 和 last 都必须为指针，find 却没有这样的限制。虽然 find 对那些参数使用了指针运算语法（operator* 用来取出 first 所指之值，operator++ 用来累加 first，operator== 用来验证 first 与 last 是否等值），但这并不表示 Iterator 一定是指针，而是仅仅意味 Iterator 必须支持类似指针的接口。如同 find1 一样，find 仍可对 range [first,last) 执行查找动作，并假设 [first,last) 满足 2.1.2 节所述的所有 range 性质，但 find 并不要求 range [first,last) 中的元素必须是数组元素。

从 find 的身影中我们看到了本章一开头指出的那种所谓的泛型算法：可以查找任何支持「一维序列概念」的数据结构。我们需要的能力是：审视元素、移到下一个元素、检查是否已处理完所有元素。基本操作行为若不以此为前提，[first,last) 大概就不能说是一个线性区间了。

链表（Linked List）的查找

我们可以把 find 用于单向链表的查找，以验证 find 的一般化程度。虽然数组与链表提供不一样的组织方式，但 find 可以适合于两者。

几乎所有稍微复杂些的 C 程序，都会实现以节点（nodes）串接起来的链表，每个节点包含一份数据，以及一个指向下一节点的指针。例如：

```
struct int_node {
  int val;
  int_node* next;
};
```

遍历链表的程序动作如下：

```
int_node* p;
for (p = list_head; p != NULL; p = p->next)
  // 做某些事情
```

（当然这是简化后的样子。实际程序中，节点可能是包含有数个字段的一个 struct 型别。本例的节点单单只含一个整数，实在是杀鸡用牛刀。）

常见的一种链表操作行为是：在链表之中查找某个指定值。换言之，就是做线性查找。既然 int_node 所形成的链表是个线性序列，我们就没有必要再写一个线性查找器；我们应该重复利用 find 函数。怎么做到呢？

我们可以利用 find 查找 range [first,last)。先前的例子中 first 与 last 都是指针，不能用在现在的状况。如果 first 是 int_node 的指针，则 find 会试图以指针运算 ++first 来取得下个元素的位置。这会引发灾难，因为我们此刻所对付的是链表，而非数组。如果 p 是 int_node 的指针，那么下一个节点应该是 p->next，而非 p+1。

这个问题很快就有了答案：C++ 允许操作符重载（operator overloading）。如果 operator++ 的行为不合需要，我们必须重定一个，使 find 可以正确走过链表。

重新定义「参数型别为 int_node*」的 operator++ 操作符是不可能的。C++ 允许你定义操作符的表达式意义，但不允许你变动既有的语法意义（如果可以更改 operator+ 的意义，使两个 int 的「相加」实际上是做「相减」，那不是天下大乱了吗！）。尽管如此，我们所能够做的，还是非常直接易懂。我们可以写一个简单的 wrapper class（外覆类），使它看起来像个 int_node*，而其 operator++ 有更好（更适当）的定义。

既然 int_node 不是唯一具有 next 指针的节点形式，我们不妨稍稍地一般化些。我们可以轻易地定义一个 template wrapper class，使它适用于任何「具有 next 指针之链表节点」，而不局限于 int_node：

```
template <class Node>
struct node_wrap {
  Node* ptr;

  node_wrap(Node* p = 0) : ptr(p) {}
  Node& operator*() const { return *ptr; }
  Node* operator->() const { return ptr; }

  node_wrap& operator++() { ptr = ptr->next; return *this; }
  node_wrap operator++(int) { node_wrap tmp = *this; ++*this; return tmp; }

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

最后，为了方便，我们可以定义一个相等操作符，用来比对 int_node 与 int（译注：如果将以下函数定义为一个 template function，使其参数型别不局限于 int_node 和 int，将更具弹性）：

```
bool operator==(const int_node& node, int n) {
  return node.val == n;
}
```

现在，欲查找哪一个 int_node 具有某特定值，我们再不需要写任何循环了。我们可以重复利用 find，将查找动作写成单一函数调用：

```
find(node_wrap<int_node>(list_head), node_wrap<int_node>(), val)
```

第二个引数 node_wrap<int_node>() 是利用 node_wrap 的缺省构造函数（default constructor）做出。它产生出一个内含 null 指针的 node_wrap。由于链表最后一个节点的 next 指针即为 null，所以我们会从 list_head 查找至链表尾端。

外覆类 node_wrap 看起来简单而普通（事实上也是），但它却很管用。要知道，int_node 并未涉及泛型算法的任何知识；它可能是数年前写成的一个数据结构，甚至在泛型算法（甚或 C++ 语言）出

现之前就存在了。只要使用小小的外覆类，我们就可以让旧的数据结构（int_node 链表）和新的泛型算法（find）之间相互作用。node_wrap class 作为二者的中间媒介而不失效率。是的，node_wrap 的所有 member function 都被定义为 inline，所以面对 node_wrap<int_node> 写出 w++，和面对 int_node* 写出 p = p->next，效果完全相同。

注意，node_wrap 非常紧密地遵循 C 一般指针的性质。它提供基本的指针运算：提领（operator*）、成员访问（operator->）以及前置累加与后置累加（preincrement and postincrement）。是的，两个 operator++ 之中，「无引数」的是前置累加版，「单一 int 引数」的是后置累加版。

2.2 Concepts 和 Modeling

find 有多么一般化呢？或者这样说好了，find 的 template 引数需要什么条件？

观察了「find 对于其 template 引数 Iterator 的运用」之后，我们可以为 Iterator 开出一串条件：必须能够以 operator* 提领 Iterator，必须能够以 operator++ 移至下一个 Iterator ……

由于 find 非常谨慎地希望尽可能一般化，所以它假设其引数可适用于任何 range。这组条件可妥善地运用在其他地方，不仅止于 find。是的，它可运用于任何「在某个 range 上面做动作」的算法。

这组条件是如此地重要，因而有属于自己的名字：find 的 template 参数是个所谓的 ***iterator***（迭代器）。***Iterator*** 是 Standard Template Library 的一个基础成分。

Requirements（需求条件）与 Concepts（概念）

众所周知，计算机科学里头的许多问题，常常借着「增加间接层」的方式获得解决。本例之中，我们的问题可以利用「为 find 之需求加上一层间接性」来解决。最重要的需求（条件）就是，Iterator 必须是一个 Input Iterator。

这里出现的特别字体，意味 Input Iterator 是一个 ***concept***（概念）。在这本书中，所有 concepts 名称都会这样表示。所谓 concept，是一组「描述某个型别」的条件。当某个型别满足所有这样的条件，我们便说它是该 concept 的一个 ***model***。举例来说，char* 与 node_wrap 便是 Input Iterator 的 model。

我以特别的字体来提醒自己，concept 并不是类、变量或是 template 参数；事实上它不是可以直接在 C++ 程序中呈现的东西。然而，在每个运用「泛型编程（generic programming）」的 C++ 程序中，concept 非常重要。

试想想 template 参数 `Iterator` 和 concept Input Iterator 之间的差异性。`Iterator` 是合乎格式的 template 参数。以 C++ 语法而言，它是一种型别，而对 function template `find` 的任何特定实体而言，template 参数 `Iterator` 代表某个特定型别。Input Iterator 这个 concept 并不代表任何型别，代表的是型别必须满足的一组条件。

想像有一种语言可以让你直接声明 concepts。你可以声明一组条件，给它一个名字。然后，就像你可以为某个型别声明变量一样，你将某个型别或某个 template 参数，声明为上述 concept 的一个 model。可惜 C++ 不是这个假想的语言。C++ 无法让你将 template 参数声明为「一定得是某个 concept 的 model」。虽然我们为 `find` 的 template 参数赋予 `Iterator` 这个名字，但 C++ 编译器无法将这个名字与 concept Inputer Iterator 关联在一起，因为 concept 并不是 C++ 语言的成分。

何谓 Concept?

如果 concept 既非 class，也非 function 或 template，那么它是什么？有三种方式可以了解 concepts，这些方式一开始似乎大不相同，但最后证明都是相通的。这三种方法有助于了解泛型算法的一些重要观点。

第一，concept 可以想像是一组型别条件（type requirement）。如果说型别 `T` 是 concept `C` 的一个 model，则 `T` 必须满足 `C` 的所有条件。「描述某个型别所必须具备的性质」几乎是最容易具体指明 concept 的方式了。

第二，concept 可以想成是型别的集合。举例来说，concept Input Iterator 可涵盖 `char*`、`int*`、`node_wrap` 等型别。如果型别 `T` 是 concept `C` 的 model，意思便是说 `T` 隶属于「`C` 所代表的那个型别集合」。由于集合中的所有型别都满足那一系列条件，所以这个定义与前一个定义相通。这个定义涉及集合自身，先前那个定义则涉及集合的形成条件，其实是以不同的角度看待同一件事情。

第三，concept 可以想成是一组合法程序。依此定义，那么像 Input Iterator 这样的 concept，其重要性在于 `find` 以及其他许多算法都会用到它。这个 concept 自身包含「`Iterator` 及那些算法」共有的性质。这种定义似乎比前两者更加抽象，但是却很重要，因为就某种意义来说，这是三种方法中最实用的。这也是新的 concept 的发掘方法。是的，我们并非藉由「写下一组需求条件」来发掘和描述新的 concepts，而是藉由「定义特别的算法」并研究「template 参数如何运用于这些算法身上」的过程来完成。我们研究 `find`，过程中导出了 Input Iterator。本书其他各种 concepts 同样是肇因于算法。

正式规格系统（formal specification system）如 Larch [GHG93] 和 Tecton [KM92, KMS82] 系以数学逻辑来定义 concept。Techton 就定义有一个 concept 为 "a set of many-sorted algebras"。本书并不采用这种正规逻辑。

Concept 的条件（Concept Requirements）

Concept 的需求条件看起来像什么呢？我们决不能说「条件（requirements）是一组 member functions」。是的，我们可以说 `char*` 是 Input Iterator 的一个 model，但 `char*` 并没有任何 member function。无论如何，前述说法并非 function template 的运作方式。只要你能将 `find` 内出现的所有 `Iterator` 都以某个型别取代掉，而导出来的函数仍然合理，你就可以用那个型别来调用 `find`。

我们得到的结论是，所谓条件，是一组有效表达式（valid expressions）。举个例子，如果 `Iterator` 是 Input Iterator 的一个 model，而 `i` 是 `Iterator` 型别的一个对象，那么 `*i` 会是一个有效表达式。

即使这么简单的条件，也已经牵扯到两种不同的型别：(1) `Iterator`（它是 Input Iterator 的 model）；(2) 经由 `*i` 取得的型别。这相当具有代表性。最一般的情况是，concept `C` 内含一组有效表达式，其中每个表达式都牵扯到 `T` 以及 `T` 的「相关型别」— 此处 `T` 必须是 `C` 的一个 model。这些相关型别（*associated types*）的细节往往是 concept 最重要的一组条件。

基本的 Concepts

有些操作行为是如此地基本，几乎每个合理型别都会提供那样的行为。在泛型编程中，这些操作行为相当于一组非常普遍的 concepts。它们是如此普遍，以至于大多数 STL 算法都倚赖这些性质。更确切地说，它们不止运用于 STL，而且对任何泛型程序库都非常基本。

这些基本的 concepts 其中之一便是 Assignable。倘若有办法复制型别 X 的值，或赋值给 X 对象一个新值的话，型别 X 则是 Assignable 的一个 model。这并不像感觉起来的那样简单，因为所有不同形式的赋值与复制都必须相互一致。面对一个 Assignable 型别，你可以确定，以下写法所获得的 `x`：

```
X x(y)
```

和以下写法所获得的 `x` 相同：

```
x = y    或    tmp = y, x = tmp
```

同样道理，你可以假设，「赋予新值给 `x`」并不会造成「其他变量 `y` 有所改变」的副作用。光只是有 copy constructor 和 assignment constructor 是不够的，它们必须互相一致，并且以 `C` 对象模型的基本性质为基础。

其他的基本 concepts 还有 Default Constructible、Equality Comparable 和 LessThan Comparable。

正如 Assignable 之 model 型别一定会有个 copy constructor，Default Constructible 之 model 型别一定会有个 default constructor — 一个不具任何引数的 constructor[2]。也就是说，型别 `T` 成为 Default

[2] 在 C++ 中，如果你不为某个类声明任何 constructor，该类会有一个隐式的 default constructor。

Constructible 的条件是，我们可以藉由以下写法产生一个 T 对象：

```
T()
```

并藉由以下写法声明一个 T 变量：

```
T t;
```

C++ 的所有内建型别如 int 和 void*，都是 Default Constructible。

Equality Comparable 和 LessThan Comparable 这两个 concepts 用来处理两个对象间的比较关系。更明确地说，型别 T 成为 Equality Comparable 的条件是，如果下式能够表现出两个 T 对象的相等与否：

```
x == y
```

或

```
x != y
```

同样地，型别 T 成为 LessThan Comparable 的条件是，如果下式能够表现出两个 T 对象的相对关系：

```
x < y
```

或

```
x > y
```

所谓正规型别（*regular type*），是一种同时满足 Assignable、Default Constructible、Equality Comparable 等条件的型别，亦即上述表达式都能以预期的方式进行。对于正规型别，你可以做这样的假设：如果将 x 赋值给 y，那么 x == y 必为真。

大多数基本的 C++ 型别都是正规型别（int 就是本节提到的所有 concepts 的一个 model），STL 定义的所有型别几乎也都是正规型别。

2.3 Iterators（迭代器，泛型指针）

Iterator 是指针的概括物；它们是「用来指向其他对象」的一种对象，这对于像 find 这样的泛型算法很重要，因为它们可以用来在对象区间内「来回移动（*iterate*）」。假设有一个 iterator 指向某个对象区间内的某个对象，当我们将该 iterator 加 1，它会指向区间内的下一个对象。

Iterator 对于泛型编程之所以重要，原因是它是算法与数据结构之间的接口。类似 find 这样的算法，以 iterator 为引数，便可以作用在许许多多不同的数据结构上 — 即使结构彼此不同（例如链表与 C 数组）。我们仅仅要求：iterator 能够以某种线性顺序遍历某个数据结构，以访问其中所有的元素。

经过 2.2 节讨论之后，iterator 似乎需要定义为一个 concept，而指针便是这个 concept 的一个 model。这差不多是正确的，但却也不尽然。C++ 的指针有许多不同的性质，而泛型算法 find 倚赖的是这些性质中的一个很小的子集。其他的泛型算法可能倚赖不同的子集。换句话说，有许多不同的

方法可以将指针一般化，每一个不同的方法便是一个不同的 concept。Iterator 不单单只是一个 concept，而是五个不同的 concepts：Input Iterator、Output Iterator、Forward Iterator、Bidirectional Iterator 和 Random Access Iterator。（第六个 concept — Trivial Iterator — 的存在目的是为了用来厘清其他 iterator concepts 的定义。）

2.3.1 Input Iterators

我们能够以 `find` 查找链表，是因为 `find` 仅以某种形式来使用其 template 引数 `Iterator`。`Iterator` 必须与指针类似，但不需要支持为指针而定义的所有操作行为。`Iterator` 必须是 Inputer Iterator 的一个 model。或者简略地说，所谓 Input Iterator 就是一个能够满足 `find` 需求的型别。这并不是一个循环定义 — 不完全是 — 因为我们可以精确枚举出那些需求条件。这也不是一个无用的定义。Input Iterator 是一个很重要的 concept，因为不只 `find`，很多算法也都有类似的需求。

本质上，Input Iterator 是某种类似指针的东西，并且在我们谈到 iterator range `[first,last)` 时有意义：

- 当它作为一般 C 指针时，有三种不同价值：它是 ***dereferenceable***（可取值的）、***past the end***（可跨越尾端的）、***singular***（可为 null 的）。当它作为指针，只有在 `first` 和 `last` 都不为 null 指针时，`[first,last)` 才是有意义的 range。
- 我们可以比较型别为 `Iterator` 的两个对象的相等性。`find` 函数内我们便是比较两个 iterators 的相等性，以此判断是否到达 range 尾端。
- Input Iterator 可以被复制或被赋值。例如在 `find` 之中你看到了，`first` 与 `last` 都是以传值方式传入，这会调用 `Iterator` 的 copy constructor。
- 我们可以提领（dereference）一个型别为 `Iterator` 的对象。也就是说，表达式 `*p` 有充分的定义。每一个 Input Iterator 都有一个 ***associated value type***，那是它所指的对象的型别。以 `node_wrap<int_node>` 为例，其 ***value type*** 就是 `int_node`。本章稍后，我们会看到 Input Iterator 还拥有其他的 ***associated types***（相关型别）。
- 我们可以对一个型别为 `Iterator` 的对象进行累加动作。也就是说，表达式 `++p` 和 `p++` 都有充分的定义。如果是一般的 C 指针，`++p` 意味着累加 `p` 后返回新值，而 `p++` 意味着累加 `p` 并返回旧值。

没列出于上的其他东西，其实和列出来的一样重要：

- 我们在 `find` 之中审视 range `[first,last)` 内的值，但不更动它们。Input Iterator 系用来指向某对象，但不需要提供任何更改该对象的方法。你可以提领一个 Input Iterator，但不能对提领后的结果赋予新值，也就是说表达式 `*p = x` 不一定有效。例如你不能更改 `const int*` 指针所指之物。

- Input Iterator 仅支持我们所习惯之指针运算中极小一部分子集。Input Iterator 可以累加，但并非一定需要递减（例如 node_wrap 就没有 operator-- 这样的 member function）。同样道理 p+5 和 p1-p2 这种表达式也并非一定有效。Input Iterator 唯一需要的运算形式是 operator++。
- 正如 Input Iterator 只支持极有限的运算形式一样，它们也只支持有限的比较形式。你可以测试两个 Input Iterators 是否相等，但你不能测试谁在谁之前。举个例子，如果 p1 和 p2 都是指针，那么 p1 < p2 是个合法的布林运算，但如果 p1 与 p2 都是 Input Iterator，上述运算便不一定成立。
- 线性查找是一种「single pass（单回）」算法。它只遍历 range [first,last) 一次，并对 range 中的值最多读取一次。这是使用 Input Iterator 的唯一正确方式。你不可以超过一次地遍历某个「以 Input Iterators 形成的 range」，也不能够以两个 iterators 指向「由 Input Iterators 形成的 range」中的两个不同元素。每个 range 一次只能支持一个有效的 iterator。例如，你不能写 p1=p2++ 然后继续使用 p1 和 p2。同样道理，如果 p == q 成立，不能认定 ++p == ++q 也成立。

务请记住这些限制所代表的意义。这些是「运用了 Input Iterator」的算法所受到的限制，而不是那些「可模塑（model）出 Input Iterator 概念」的型别所需接受的限制。例如，虽然 double* 是 Input Iterator 的一个 model，但是 double* 却能够支持「多回（multipass）」算法。Input Iterator 携带的条件是功能需求之最小集合，而 Input Iterator 的 model 则可能（而且经常）超过这个最小集合。无论如何，一个可以和 Input Iterator 搭配的算法，不能有超越「Input Iterator 所保证之最小功能集」的任何要求。

这些限制非常严格 — 特别是「只有 single pass 算法才能使用 Input Iterator」这一条。你可能会纳闷，如何能够如预期般地遵循这些限制进而实现每个算法呢？答案是你并不被期望做这样的事！

Input Iterator 是两个最「弱」的 iterator concepts 之一。许多算法要求获得的是其他类型的 iterators。许多算法无法单以 Input Iterators 完成，这应该不会令你意外。令人意外的反倒是那些单以 Input Iterators 就能完成的家伙。除了 find（p.199）及其相关算法如 find_if（p.285）之外，Input Iterator 还可以用在如 equal（p.220）、partial_sum（p.285）等算法，甚至如 random_sample（p.277）和 set_intersection（p.327）等复杂算法身上。

Input Iterator 这个名称，暗示着这些需求条件某些看似奇怪的理由。使用 Input Iterator 就像从终端机或网络连线读取数据一样地读进输入值：你可以要求下一个值，但一旦如此，前一个值便消失无踪了。如果你希望再次取得前一个值，你必须自行将取自 input stream 的数据存储下来。换句话说，以 Input

1066204

Iterator 写成的算法，就像面对「由 input stream 读取出来」的数据所能够写成的算法一样。事实上如果你使用 STL 预定义的 Input Iterator class `istream_iterator`（p.354），你真的可以很简单地由 input stream 读进数据。

2.3.2 Ouput Iterators

如果说 Input Iterator 是做「从 input stream 读入」的动作，你或许会期待另有一个 iterator 做「写入 output stream」的动作。是的，这种 iterator 称为 Output Iterators。

欲了解某个 concept，意味着必须了解其使用者（某个算法）。一个「需要执行输出动作」的简单例子是：将某区间的值复制到另一区间去。输入区间可由 Input Iterator 构成，但输出区间却不能如此。因为 Input Iterator 并不提供更改「iterator 所指之物」的方法。然而如果使用 Output Iterator，很简单就能实现一个循序复制的算法：

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
{
  for ( ; first != last; ++result, ++first)
    *result = *first;
  return result;
}
```

这是 STL 的 `copy` 算法（p.233），它和 `find` 一样地使用 Input Iterator，因为它也是单回（signle pass）算法，面对 Input Iterator，在往下移一步之前只会读取一次。

`copy` 对 Output Iterator 的使用方式也很类似，但其中承受了输入与输出之间的差异性。我们以 Output Iterator 写入连续序列的值。写一个值后，将 Output Iterator 累加，再写下一个值。我们绝不会通过同一个 iterator 写值两次，也绝不会回到前一个 output iterator，而且也绝不会跳过任何一个 iterator。输出区间是只写性质（write-only），就好像输入区间是只读（read-only）似的，`copy` 则是与输出输入皆有关的单回（single-pass）算法。

Output Iterator 的需求条件与 Input Iterator 类似：

- 如同 Input Iterator，我们可以复制和赋值 Output Iterator。
- 我们可以使用 Output Iterator 来写值。换言之，表达式 `*p = x` 有良好定义。
- 我们可以累加 output iterator。换言之，表达式 `++p` 与 `p++` 有良好定义。

Output Iterators 和 Input Iterators 共享了某些相同的限制。Output Iterators 必须支持 `++` 操作符，但不必支持其他任何指针运算。任何算法如果使用 Output Iterators，像 `sort` 那样，必定是个 single-pass 算

法，而且，就像 Input Iterators 的情况一样，同一时间内不能有两个不同的 Output Iterators 指到同一个区间内的不同地点。移动中的指针进行写入动作，一旦写完便继续移动。

除此之外，Output Iterator concept 另有其他两个重要限制。第一个很明显：正如 Input Iterator 具有只读性一样，Output Iterator 具有只写性。所以，你可以写 `*p = x`，但不能写 `x = *p`。第二个限制就不是那么显而易见了，但我们可以利用 `copy` 加以说明。

`copy` 内有输入和输出两个区间。每个区间都有开头及结尾，所以共计四个位置。然而，`copy` 的引数列中只有三个 iterator：输入区间 `[first,last)`，以及输出区间的开头 `result`。幸运的是我们可以不用考虑这一点，因为 `copy` 的涂写个数必须与它的读取个数相同；输出区间的尾端不言自明。

因此虽然 `copy` 必须测试 `first != last`，却不需要对 `result` 做任何这类测试。既然 `copy` 及其他类似算法并不需要将某个 Output Iterator 与另一个相比较，所以 Output Iterator 也就不必提供这样的能力。Output Iterators 所形成的「区间」都是以单独一个 iterator 再加上一个计量值来决定。以 `copy` 为例，其输出区间是以 `result` 作为开头，再以 `[first,last)` 所涵盖的元素个数作为区间的长度。

恪遵 Output Iterator concept 的所有限制来撰写算法，是有可能的。`copy` 就是这样的算法，`transform`（p.240）、`merge`（p.316）以及其他某些算法也是。这是必要的吗？是否真的有某些理由使你一定得使用只写性质的 iterator 来支持 single-pass 算法？

答案是肯定的。事实上 Output Iterators 非常普遍也非常有用，其中最重要的一个就是 `insert_iterator`(p.351)，它和其相关的 `front_insert_iterator` 和 `back_insert_iterator` 一样，都可以将新值安插到容器内。Output Iterator 的最简单实例（并且可以阐明这个 concept 的许多特性）就是 `ostream_iterator`（p.357）。

`ostream_iterator` class 是一个与 `istream_iterator` 反相的东西，同时也是一个最直觉的 Output Iterator。如果 `p` 是个 `ostream_iterator`，则 `*p = x` 会使 `x`「格式化输出」至 ostream 身上。`ostream_iterator` 的定义虽然简单，却深具启发性：

```
template <class T> class ostream_iterator {
private:
  ostream* os;
  const char* string;
public:
  ostream_iterator(ostream& s, const char* c = 0) : os(&s), string(c) {}

  ostream_iterator(const ostream_iterator& i)
    : os(i.os), string(i.string) {}

  ostream_iterator& operator=(const ostream_iterator& i) {
    os = i.os;
    string = i.string;
    return *this;
  }
```

```
  ostream_iterator<T>& operator=(const T& value) {
    *os << value;
    if (string) *os << string;
    return *this;
  }

  ostream_iterator<T>& operator*() { return *this; }
  ostream_iterator<T>& operator++() { return *this; }
  ostream_iterator<T>& operator++(int) { return *this; }
};
```

上述定义展示了 Output Iterators 不平常的特色。它显示如何定义一个「只写性质的 iterator」，也显示出为什么 Output Iterators 的使用者必须是与 copy 相同形式的所谓 single-pass 算法。如果你将 ostream_iterator 牢记在心，那么 Output Iterator 的限制就不会令你觉得那么怪异。

ostream_iterator 只能用在 single-pass 算法身上，因为 ostream_iterator class 不会记录 ostream 内的哪个特定位置：它只做输出动作，除此之外便无其他。你不可以藉由复制 ostream_iterator 来存储某个旧位置，因为「对同一 ostream 做写入动作」的任何一个 ostream_iterator 都完全相同。以下的 copy 动作：

```
copy(A, A+N, ostream_iterator<int>(cout, " "))
```

乃是将整数序列复制到标准输出设备。只有这样形式的算法才可以使用 ostream_iterator。

我们也可以看看 ostream_iterator 如何提供「只写访问」，以及这么做的理由。由于 ostream class 自身语义的关系，「只写访问」有其必要，而提供它的最简单做法就是以「定义一个 proxy class（代理类）」这种常见的 C++ 技法来完成。我们要让表达式 *p=x 执行某个行为，所以我们定义 *p 返回一个 proxy object，后者有适当 operator= 来执行该行为。这种情况下，ostream_iterator 唯一稍不寻常的特点是，其 proxy class 便是 ostream_iterator 自身。我们也可以定义另一个 class 取而代之，但没有理由这么做。

花费这么大的力气，似乎只是为了可以将 os << x 替换为 *p = x。但我们其实获得了很重要的东西。泛型编程（generic programming）非常依从语法的约束。这种一致性的语法形式使我们有可能以同一个 copy 算法完成任何「涉及将数值写至任意种类之输出区间」的动作。是的，copy 算法可以将数组元素搬移至另一数组、将元素安插至某个容器内或是将元素写到标准输出设备。

2.3.3 Forward Iterators

很多算法可以单独以 Input Iterator 和 Output Iterator 完成，但也有许多算法无法如此，我想这不会令你感到惊讶：

- Input Iterator 具只读性，而 Output Iterator 具只写性。这意味任何需要「读取并更改」某一区间的算法，无法单就这些 concepts 来运作。你可以利用 Input Iterators 写出某种查找算法，但无法以之写出「查找并置换」的算法。

- 由于运用 Input Iterators 与 Output Iterators 之算法，只能是 single pass（单回）算法，这使得算法的复杂度最多只能与区间内的元素呈线性关系。但我们知道并非所有算法都是线性的。许多有用的排序算法，复杂度是 $O(N \log N)$ 或 $O(N^2)$。
- 任何时刻，某个区间内，只能有一个有效的（作用中的，active）Input Iterator 或 Output Iterator。这使得算法在同一时刻只能对单一元素做动作。对于那种「行为取决于两个或多个不同元素之间的关系」的算法，我们无能为力。

算法如果采用 Forward Iterators，就不会有上述限制。Forward Iterator 支持的指针运算与 Input Iterators 及 Output Iterators 同级（你可以写 `++p`，但不能写 `--p` 或 `p += 5`），其他方面的限制则较为宽松。一个身为 Forward Iterator model 的型别，必然也是 Input Iterator model 和 Output Iterator model，因此我们有可能写出一种算法，让它在同一个区间内做读取动作和更新动作。`replace` 就是这样一种算法，它找出所有的 `old_value` 实体，并以 `new_value` 实体取代之。

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value)
{
  for (; first != last; ++first)
    if (*first == old_value)
      *first = new_value;
}
```

如同那些运用 Input Iterators 与 Output Iterators 的算法一样，`replace` 也是 single pass 算法，一次只作用一个元素。我们之所以需要威力较强的 Forward Iterator concept，是因为我们需要在同一个 iterator 上进行读取和涂写。Forward Iterator 也允许算法使用更一般化的访问模式（access patterns）。

试考虑线性查找的一个变形。这次我们不再查找特定某值，而是查找是否「连续两元素具有相同值」：

```
template <class ForwardIterator>
ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last)
{
  if (first == last)
    return last;
  ForwardIterator next = first;
  while (++next != last) {
    if (*first == *next)
      return first;
    first = next;
  }
  return last;
}
```

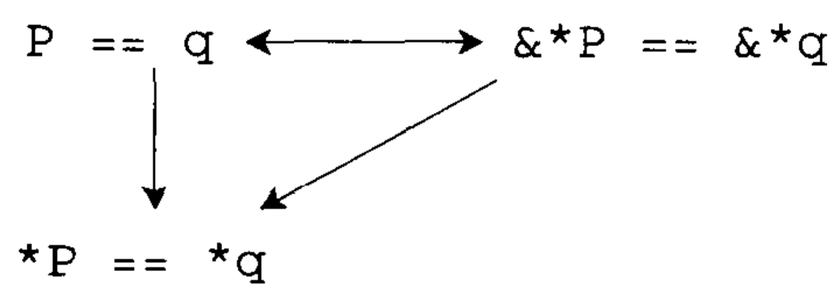

图 2.1 「iterator 相等」的同一性（*identities of iterator equality*）

这个算法使用了 Forward Iterator。虽然 `adjacent_find` 只做只读动作，却不能以 Input Iterator 写成，因为循环的每一次迭代都会比较两个不同的值，`*first` 与 `*next`。如果算法只使用 Input Iterator，无法在同一区间内使用一个以上的 iterator。

在这里，你必须了解很重要的一点，那就是 Input Iterator 和 Output Iterator 有着奇特的性质。如果使用这两种 concepts，算法就必须受限于「简单循环」形式。使用 Forward Iterator 则没有这样的限制。

基本的差异在于，Input Iterator 和 Output Iterator 并未完全遵循一般的 C 内存模式（C memory model），而 Forward Iterator 对之有所遵循。例如 2.1.3 节那个用于链表的 `node_wrap`，就是一个 Forward Iterator。是的，`node_wrap` 对象会指向某个特定地址，就像一般指针一样。

如果 `p` 和 `q` 都是一般的 C 指针，你可以写出以下行为：

```
p = q;
++q;
*p = x;
```

可预期的是 `p`（而非 `q`）所指的值会被更动。这就是 C 内存模式的基本性质：不同的指针指向不同的内存位置。如果 `p` 和 `q` 都是可提领（dereferenceable）指针，那么当且仅当（if and only if）`*p` 与 `*q` 是同一对象，`p == q` 才成立。这同时也可以拿来作为「两指针相等与否」的定义。

Forward Iterator 遵守与上述一样的「同一性（identity）」。此性质不但基本而且重要。如果没有这种性质，我们不知如何让 `sort` 排序，也不知如何让 `reverse` 反转。这是一个强烈条件（strong condition）：如果 iterator `p==q`，则 `*p == *q`。不过下面这个条件更强烈：`p` 与 `q` 不仅指向相同的值，而且指向同一对象；如果你更改 `*p`，你也就同时更改了 `*q`。这个条件可以写为 `&*p == &*q`，因为如果两个对象 `*p` 和 `*q` 拥有相同地址，它们一定是同一个对象[3]。图 2.1 显示了这些同一性（identity）之间的关系。

「两个 iterators 是否相等（equality）」的定义，亦可适用于一般的 C/C++ 指针身上。（这也是 Output

[3] 这并不总是完全正确。一般来说，`*p` 代表 iterator `p` 所指对象，所以 `&*p` 是该对象的地址。但是请记住，C++ 允许你将 `&` 操作符重载（overload）。果真如此则 `&*p` 的意义可能完全不同。不过大部分时候我们可以忽略这种技术。

Iterators 不能比较彼此是否「相等」的一个原因。因为如果你无法验证 iterator 所指之值，前述定义也就没有意义了。）

Const（不变的）Iterators 和 Mutable（可变的）Iterators

replace 和 adjacent_find 算法都使用 Forward Iterator，但它们的使用方式有个重要差异：replace 修改 [first,last) 区间内的值，而 adjacent_find 不然。并非所有使用 Forward Iterator 的算法都会修改 iterator 所指之值。

Forward Iterator 可以是 ***constant***（常量的），这种情况下我们可以取出它所指的对象，但不能经由它赋值；Forward Iterator 也可以是 ***mutable***（可变的），这种情况下我们可以取出其所指之值，或是对它做赋值动作。int* 和 node_wrap<int_node> 都是 mutable Forward Iterator（它同时也是 Input Iterator，并且也是 Output Iterator）。const int* 则是一个 constant Forward Iterator。

constant iterator 这个术语在本书有一贯性用法，但它有点令人困惑：constant iterator 并不一定得是（甚至不常是）const object。constant iterator 是一种「无法用来更改其所指值」的 iterator，但 iterator object 自身可能是 const，也可能不是。例如 const int* 就是一个 constant iterator，因为它不可被用来更改其所指之值。int* const 就不一样，它是 const object，但可用来更改其所指值。

我们当然可以使用两种不同的 concepts 来陈述 constant iterator 与 mutable iterator 的差异，但这会增加非必要的复杂度。

2.3.4 Bidirectional Iterators

和 Forward Iterator 一样，Bidirectional Iterator 允许 multipass 算法。而且它也可以是 constant 或 mutable。

Bidirectional Iterator 的名称暗示：它支持双向移动。Bidirectional Iterator 可以累加以取得下一个元素，或是递减以取得前一个元素。相对之下 Forward Iterator 只支持单向移动。举例来说，像 node_wrap 这样能够遍历单向链表(singly linked list)的便是 Forward Iterator；而能够遍历双向链表(doubly linked list)的则是 Bidirectional Iterator。

凡选择运用 Bidirectional Iterator 之算法，通常都是需要在某个区间内反向移动；也就是说指定某元素之后，该算法需要寻找其先前的元素。举个例子，我们所看过的 copy 算法，会将某个区间内的元素复制到另一区间。如果要以相反方向复制，就必须逆向移动 input 和 output 两区间中的一个：

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result) {
  while (first != last) {
```

```
    --last;
    *result = *last;
    ++result;
  }
  return result;
}
```

我们不需对 input 和 output 两区间都做反向遍历，只需对其中一个进行即可。我们的选择是 input 区间，因此 reverse_copy 的接口很类似 copy。input 区间与 output 区间系以三个 iterators 表示，而非四个，其中 output 区间由 Output Iterator 构成。

reverse_copy 算法是对 last 递减，而非对 first 累加。凡是 Forward Iterator 支持的行为，Bidirectionary Iterator 也都支持，此外它还支持 operator--。注意，reverse_copy 和 copy 并不相同：在 reverse_copy 中我们先递减 last 然后再提领（dereferencing）其值。看起来有点奇怪，事实上完全合理。这样的行为是必要的，因为 [first,last) 是个不对称区间，last 并不属于该区间，而是「尾端的下一个位置」。提领「尾端之后」的 iterator 是不合法的，但是将「尾端之后」的 iterator 递减，是合法的，而且有完善的定义。

2.3.5 Random Access Iterators

在此之前所提的四个 concepts 都只提供指针运算的一小部分子集：Input Iterator、Output Iterator 和 Forward Iterator 只定义 operator++，Bidirectional Iterator 也只不过增加了 operator--。STL 的最后一个 iterator concept — Random Access Iterator — 则涵盖其余所有指针运算：加法与减法（p+n 和 p-n）、下标（p[n]）、两个 iterators 相减（p1-p2）以及前后次序关系（p1 < p2）。

Random Access Iterator 对于 sort（p.292）这类算法很重要，因为排序算法必须有能力比对并交换相隔甚远的元素（而非只是相邻元素）。

一开始你可能会觉得 Random Access Iterator 有其必要。如果 p 是个指针，你可以写 p+n，所以当然也应该有个 iterator concept 能做相同的动作。然而，三思之后，你或许会怀疑是否真有必要定义出 Random Access Iterator，因为其大部分行为皆可由 operator++ 和 operator-- 完成。举例来说，我们可以定义一个名为 advance 的函数（STL 便是如此），让 advance(p,n) 将 Forward Iterator p 累加必要的次数。那么，我们究竟是真的从 Random Access Iterator 身上获得什么新内涵，抑或它只是给我们一些语法上的甜头？如果只是后者，为什么我们只为某些 iterators 提供甜头，其他则没有呢？

问题取决于我们尚未考虑到的某种东西：算法的复杂度（complexity）。如果 p 是个 Forward Iterator，advance(p,n) 会重复执行 operator++。我们有理由预期 advance(p, 500) 所花的时间大约是 advance(p, 100) 的五倍。换句话说，advance 的复杂度是 $O(N)$。

但这并非指针的运作方式。如果 p 是个指针，p+1000000 并不会比 p+1 来得慢。换言之，指针运算

是 *O(1)*。访问数组中的任何一个元素，都像访问其他元素一样快。作用于数组上的算法，必须倚赖这样的事实。以 Quicksort 为例，它是一个作用于数组的排序算法，其快速的原因是它可以在固定时间内访问任何一个数组元素。如果把 quicksort 实现于链表（linked list）数据结构上，就一点也不合理，因为对着链表取其任意元素的方法只有一种：一次一个地在链表中向前移动。

Random Access Iterator 真正独特的性质是它能够以固定时间（constant time）随机访问任意元素。Random Access Iterator 和 Bidirectional Iterator 两者名称上的差异正足以反映出这个事实。

2.4 Refinement（精炼、强化）

Bidirectional Iterator 满足 Forward Iterator 的所有条件，以及其他更多条件；换句话说 Bidirectional Iterator 之任何 model 必然也是 Forward Iterator 的 model。

像 Bidirectional Iterator 和 Forward Iterator 这样的关系十分常见也十分重要，因此拥有一个专属的名称：***refinement***（精炼、强化）。如果 concept C2 提供 concept C1 的所有功能，再加上其他可能的额外功能，我们便说 C2 是 C1 的 refinement。

modeling 与 refinement 满足三个重要性质，如果我们将 concepts 视为型别的集合，便很容易验证这些性质：

1. ***Reflexivity***（自反性）。每个 concept C 都是其自身的一个 refinement。
2. ***Containment***（涵盖性）。如果型别 x 是 concept C2 的一个 model，而 C2 是 concept C1 的一个 refinement，那么 x 必然是 C1 的一个 model。
3. ***Transitivity***（传递性）。如果 C3 是 C2 的 refinement，而 C2 是 C1 的 refinement，那么 C3 一定是 C1 的 refinement。

这些性质所造成的实际影响是，如果 concept C2 是 concept C1 的一个 refinement，而某个算法要求其引数型别必须是 C1 的 model，那么你永远可以喂给它一个 C2 model 的引数型别 — 这是以更前进、更丰富的方式来描述你已经知道的事实。以算法 `find` 为例，它需要 Input Iterator 作为其 template 引数，而你已经知道，你可以将指针传给 `find`，因为指针是 Random Access Iterator 的一个 model（译注：而 Random Access Iterator 是 Input Iterator 的一个 refinement）。

一个型别（type）可以是多个 concepts 的 model，而一个 concept 又可以是多个 concepts 的 refinement。例如 Forward Iterator 便同时是 Input Iterator 和 Output Iterator 的 refinement。一般来说，一堆 concepts 会形成一个复杂的阶层体系（hierarchy）[4]。

---

[4] 严格来讲，不应该称为阶层体系（hierarchy），而是一个「有向无环图（directed acyclic graph）」。不过习惯上我们称之为「阶层体系」，虽然就技术而言这是一种错误的称谓。

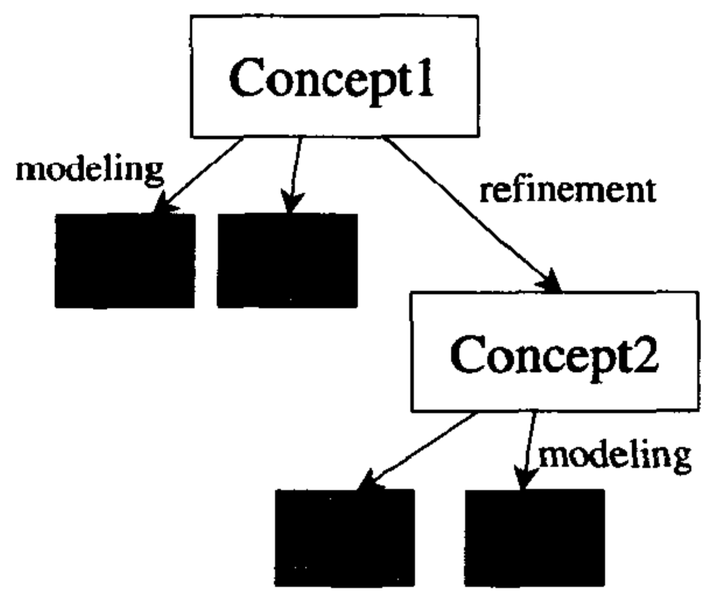

图 2.2 Refinement 与 Modeling 示意

Concepts（概念）与 Inheritance（继承）

Modeling 和 refinement 都是指两种不同事物之间的关系。Modeling 是 type 与 concept 之间的关系（例如 type T 是 concept C 的一个 model），refinement 则是两个 concepts 之间的关系（例如 concept C2 是 concept C1 的一个 refinement）。

这些都是「**is a**（是一种）」的关系，但任何一本 C++ 书籍都会描述另一个「**is a**」的关系：继承（inheritance）。我们常说如果 class D「是一种」class B，则 D 必须继承自 B。这是否表示 modeling 与 refinement 其实只不过是「上下继承」的关系，并不需要 template 的介入？是否只要以继承（inheritance）和多态（polymorphism）就可以模拟泛型编程（generic programming）？

答案是否定的。这项叙述的寓意是，我们应该当心像「**is a**」这样含糊不清的用语。Modeling、refinement 和 inheritance 是事物之间三种不同的关系，我们无法以其中两种关系来模拟第三种关系。

你可以根据其定义，条理分明地了解其中差异。不同的关系作用于不同的层面上。所谓 inheritance（继承）是两个 type（型别）间的关系，所谓 modeling 是一个 type 与一组 types 间的关系，所谓 refinement 则是两组 types 之间的关系。如果 class D 继承自 class B，则 class D 的对象必然也是 class B 的对象。Refinement 定义于型别层次而非对象层次；如果 concept Concept2 是 concept Concept1 的 refinement，那么 Concept2 的每一个 model（type）必然也是 Concept1 的 model。

图 2.2 解释了这样的观念。即使 Concept2 是 Concept1 的一个 refinement，而 T3 与 T4 是 Concept2 的 models，T1 和 T2 是 Concept1 的 models，并不就代表 T3, T4 一定与 T1, T2 之间有什么特别关系。这四个型别之间所共享的，单纯只是个「接口」而已，无关乎任何实现细节。

以 Random Access Iterator 为例，它是 Input Iterator 的一个 refinement，而 char* 是 Random Access Iterator 的一个 model。但这并不代表 char* 继承自「Input Iterator 的任何一个 model」！

欲了解 inheritance 与 modeling 之间并无关联，最简单的方法就是试着以 inheritance 来完成 modeling。那是行不通的。你或许会想像以抽象基类（abstract base class）来表现 concepts，但这样一来你却表现不出其中必要的关系。例如，考虑将 Assignable 以抽象基类来表现，你可能会这么做：

```
class assignable_base {
public:
  virtual ~assignable_base() {}
  virtual assignable_base& operator=(const assignable_base&) = 0;
};
```

我们立刻就遇上了麻烦。assignable_base 的继承者并不会自动获得 Assignable model 的各种操作行为（operations）。多态（polymorphism）在此并无助益，因为 assignable_base 自身就没有提供那些行为。如果某个泛型函数（generic function）的引数型别为 Assignable 的一个 model，那么这个函数可以对该型别之对象进行赋值动作，或是为该型别产生新变量……但如果某个多态函数（polymorphism function）接受一个型别为 assignable_base* 或 assignable_base& 的引数，就无法做到这些事情。如果多态函数接受的是两个型别为 assignable_base* 的引数，它甚至无法确定这两个引数的型别是否相同 — 而如果它们不相同的话，当然也就没有理由认为其中一个可以被赋值给另一个。作为 concept，Assignable 是有用的，但如果作为抽象基类，就没有什么用。

像 Input Iterator 这类「涉及不只一个型别」的 concept，所产生的问题甚至更糟。如何以继承和多态的符号来表示某个 iterator 的 ***value type***？要知道，double* 和 node_wrap<int_node> 这两个型别都是 Input Iterator 的 models，但其 value type 却分别是 double 和 int_node。它们并不能被视为同一个抽象基类的「子类（subclasses）」。

以上叙述并非否定继承与多态的重要性与实用性，而是告诉我们，有必要知道「inheritance（继承）和 modeling 系针对不同的事物起作用」。Inheritance、modeling 和 refinement 是三种完全不同的关系，任何方法论（methodology）如果只尝试运用其中某种关系，都是不完整的。

2.5 总结

「将 iterator 定义为指针的一般化形式」是 2.3 节所揭示的序幕，现在我们终于可以看到它所代表的意义。指针可以被一般化为多种不同的形式 — 取决于我们想要加以抽象化的性质。因此 iterator 不单是一个 concept，而是一群彼此具有 refinement 关系的 concepts 家族。图 2.3 显示了这个家族的关系。

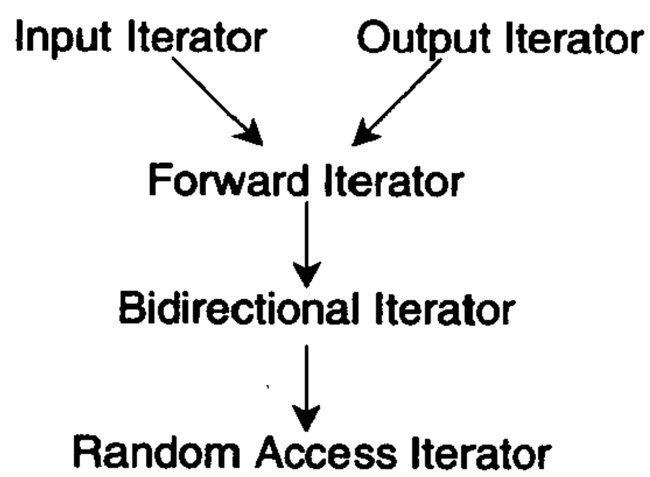

图 2.3 iterator concepts 的阶层关系（hierarchy）

图 2.3 中，Input Iterator 和 Output Iterator 是限制最多的两个 concepts，其中一个有只读性，另一个有只写性；两者都支持「对某个区间进行向前遍历」的行动，而应用于单纯的 single-pass 算法。Forward Iterator 也支持「对某区间的向前遍历动作」，但它支持更一般化的（译注：而非只是 single pass）算法。Forward Iterator 就像普通的 C 指针那样可以「指向」某物。这意味（正式的说法）对 Forward Iterator 做提领（dereferencing）动作，会产生一个 lvalue（左值）。

既然 Bidirectional Iterator 是 Forward Iterator 的 refinement，而 Random Access Iterator 又是 Bidirectional Iterator 的 refinement，所以 Bidirectional Iterator 与 Random Access Iterator 同样支持向前遍历，以及 C 内存模式（C memory model）。不同的是它们还提供其他的遍历形式。Bidirectional Iterator 允许逆向遍历，Random Access Iterator 允许任意距离的跳跃，正如其名称所示。

这些不同的 iterator concepts 提供了泛型算法一个自然分类法则。算法可以以其所采用之 iterator concepts 来分类。在所谓 ***nonmodifying***（不可更改的）算法和所谓 ***mutating***（可变动的）算法之间有一个重大区别，那就是前者使用 constant iterator，后者使用 mutable iterator。

第 7 章将列出所有 iterator concepts 的完整文档，包括五个 iterator concepts 中的任何一个的全部需求条件。是的，虽然至此我们已经讨论过 iterators 的基本性质，但还缺最后一个条件。

第 3 章

再论 Iterators（迭代器 or 泛型指针）

More About Iterators

第 2 章中，我利用 ranges、iterators 以及 STL 的五个 iterator concepts：Input Iterator、Output Iterator、Forward Iterator、Bidirectional Iterator 和 Random Access Iterator，来介绍所谓的泛型算法。不过，那还不够完整，因为尚未提到一些提高主题。当你准备撰写自己的 iterator class 或泛型算法时，这些提高主题是很重要的。

3.1 Iterator Traits（迭代器特征）与 Associated Types（相关型别）

2.2 节曾经提到，concept 可以包含一些 ***associated types***（相关型别）。型别 `T` 的条件是一组涉及 `T` 的表达式，这些表达式可以使用或返回其他某些型别。五个 STL iterator concepts 确实都包含有各自的 ***associated types***，它们是形成 iterator 条件的重要成分。

3.1.1 Value Type（数值型别）

我们已经看到，每个 iterator（除了 Output Iterator）至少会有一个 associated type。是的，iterator 会指向某物，而此物的型别便是这个 iterator 的 ***value type***。

问题在于，我们尚未看到应该如何取用 value type。假设我有一个 iterator type `I`，并需要声明一个变量，其型别为「`I` 的 value type」。这是十分合理的情节，是的，许多算法（例如一个可计算「某数列的平均值及标准差」之泛型算法）都需要一些临时变量，然而「`I` 的 value type」一词并没有告诉我们 C++ 如何表达这样的概念。

这里有个小技巧，用的是 C++ 的 type inference（型别推论）机制。假设有个泛型函数 `f()`，需要一个型别 `I` 的引数，而且需要声明一个临时变量，其型别为「`I` 的 value type」。虽然 C++ 不允许我们写 `typeof(*I)`，但还是有办法绕个弯跨越这项限制。我们可以令 `f()` 成为一个单纯的转递函数（forwarding function），并将所有工作委托给内部函数 `f_impl()`：

```
template <class I, class T>
void f_impl(I iter, T t)
{
  T tmp;              // T 是 I 的 value type
  ...
}

template <class T> inline void f(I iter)
{
  f_impl(iter, *iter);
}
```

于是我们可以在函数 f_impl 之中，声明一个型别为 I's value type 的变量。template 引数的 "type inference" 机制是自动自发的，所以在 f_impl 之中，「template 引数 T」与「*iter 的结果」同义。这个技巧既巧妙又实用。STL 在某种场合下会使用这种技巧的变形。不幸的是，光这样还不够。如果我们只需要声明临时变量，这个技巧尚可满足（虽然有点笨拙），但它却无法用来声明函数的返回型别。问题出在「f() 的返回型别」声明于 f() 自身，而这种 type inference 机制无法让我们取用「需由 f_impl() 自身取得」的型别。（译注：简单地说，C++ template 的引数推导过程 "arguments deduction"，只及于引数身上，不及于函数返回值身上）。

你可能会考虑第二种方法：在 iterator class 内声明其 value type。是的，C++ 允许嵌套式（nested）型别声明；所以，举例来说，我们可以在 node_wrap iterator class（见 2.1.3 节）中利用 typedef 新增一个型别：

```
template <class Node> struct node_wrap {
  typedef Node value_type;
  Node* ptr;
  ...
};
```

如果型别 I 是个 node_wrap，嵌套型别声明允许我们将 I 的 value type 写成 typename I::value_type[5]。这看起来似乎便是我们所需要的。

但其实不是。我们不能要求所有的 iterator 都声明嵌套型别。像 node_wrap 这类 iterator class 倒还好，但对于那些 non-class iterator 呢？例如指针是个 iterator，却不是个 class。如果 I 是 int*，我们无法把 I::value_type 定义为 int。光靠嵌套型别，会破坏 iterator 的原意。指针是 Random Access Iterator 的一个 model，而且我们故意以指针作为其他 iterators 的样本。所以，不论 iterators 采

[5] 此处必须使用看似奇怪的关键字 typename，因为当 I 是个 template 参数时，在此一 template 被具现化（instantiated）之前，编译器完全不知道 I 的任何信息。更明确地说，编译器无法知道 I::value_type 是型别名称，或是 member function，抑或 member variable。细节请看 *The C++ Programming Language* [Str97] 一书的 A.1.5 节与 C.13.5 节。

纳什么条件，都必须满足指针。

我们可以增加一层间接性来解决这个问题。做法是定义一个辅助用的 class，`iterator_traits`：

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::value_type value_type;
};
```

这么一来我们就可以这样的写法取用 iterator `I` 的 value type：

```
typename iterator_traits<I>::value_type
```

这看起来似乎没有任何改进，因为 `iterator_traits` 仍然假设它的 template 引数 `Iterator` 有一个嵌套型别。然而这的确是个改进，因为我们可以利用「对某种 template 引数提供另一种定义」的方法，将 `iterator_traits` class 特化（specialize）。

C++ 允许 template 的全特化（full specialization，亦即为某型别如 `int*` 提供另一种定义）和偏特化（partial specialization，亦即提供另一个定义，其自身亦为 templatized）。本例之中，我们需要针对每一种指针型别做出另一种定义，因此我们需要偏特化：

```
template<class T>
struct iterator_traits<T*> {
  typedef T value_type;
};
```

这已经相当完整了：它允许我们以一致的方法取用所有 iterator（包括指针）的 value type。但是有个小细节必须提出来讨论：constant iterator 的 value type 是什么？考虑下面的例子：

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

根据先前定义的 `iterator_traits`，这个型别会是 `const int` 而非 `int`，因为我们已经针对型别 `T*` 的引数将 `iterator_traits` 特化，而当 `T` 是 `const int` 时，`const int*` 会符合 `T*`。不幸的是这不是我们真正所要的。毕竟我们的目的是声明临时变量，其型别与 iterator 的 value type 相同，而声明一个不能赋值的临时变量，没什么用。iterator（即使是 constant iterator）的 value type 不应该加上 `const` 修饰词。iterator 如果是 `const int*`，其 value type 应该是 `int` 而非 `const int`。`const` 只是用来限制「通过 iterator 所能执行的动作」。

只要另外设计一个偏特化版本（partial specialization），就能轻易解决这个问题：

```
template <class T>
struct iterator_traits<const T*> {
  typedef T value_type;
};
```

现在我们有了三种不同的 `iterator_traits` 版本：一个是针对型别为 `T` 的引数，一个是针对型别为 `T*` 的引数，另一个是针对型别为 `const T*` 的引数。当你写下 `iterator_traits<const int*>`，

原则上它符合任何一个版本，不会有歧义（模棱两可）情况，因为 C++ 编译器总是能够选出最能明确匹配（最符合）的版本。终于，我们有了某种机制，让我们得以使用某个 iterator 的 value type 来撰写算法。举个例子，以下的泛型函数用来计算 nonempty range 内的数值总和：

```
template <class InputIterator>
typename iterator_traits<InputIterator>::value_type
sum_nonempty(InputIterator first, InputIterator last)
{
  typename iterator_traits<InputIterator>::value_type result = *first++;
  for (; first != last; ++first)
    result += *first;
  return result;
}
```

这种写法行得通，因为我们已经建立了一些 iterator_traits 机制。由于特化版本 iterator_traits<const T*> 和 iterator_traits<T*> 的缘故，所以对指针也行得通。对于某些 iterator，其自身以嵌套型别来定义 value_type，也行得通。

如果 I 是 Input Iterator（当然亦包括 Input Iterator 的所有 refinements），那么 iterator_traits<I>::value_type 必须是 I 的 value type。换言之当你定义新的 iterator class 时，你必须确定你的 class 能够支持 iterator_traits。最简单的方法便是：总是在你的 class 中以嵌套型别定义 value_type。这么一来你就不需要明白（explicitly）提到 iterator_traits，而定义于程序库内的 iterator_traits class 仍能正确运作。如果因为某种原因，使得你无法或不方便使用现有的 iterator_traits，你可以针对你的 class，将 iterator_traits 特化，就像先前对指针做法一样。

显然，iterator_traits 机制相当地一般化。我们已经见识了它对于 iterator value type 的作用方式。事实上它还可以运用于「iterator type 与其各种 associated types」之间的任何映对关系。2.3 节的 iterator concepts 隐含有一些 associated type，STL 便是使用 iterator_traits 技术来对付它们。

3.1.2 Difference Type（差距型别）

如果 I 是 Random Access Iterator 的一个 model，p1 与 p2 是型别 I 的值（value），则 p2-p1 便是 p1 到 p2 之间的距离。如果我们会在程序中用到这个距离，我们必须先知道其型别。表达式 p2-p1 的结果是什么型别呢？

如果 I 是指针，答案很简单。指针相减的结果，其型别为 ptrdiff_t[6]。所以，面对所有 Random Access Iterators（而不单单只是指针），可能的解决方法是：令 p2-p1 的返回型别永远是 ptrdiff_t。

[6] C 和 C++ 标准规格都定义有 ptrdiff_t 型别。其本身并非某种新型别，而是以 typedef 定义的带正负号整数型别，通常是 int 或 long。

这样的解决方法虽然简单，却相当受到束缚。如果你有一个 iterator range，其区间太大，以至于 ptrdiff_t 不够存储区间内的元素个数呢？文件便有可能如此。这是个严重的问题。现今诸多处理器中，指针的大小是 32 位元，所以 ptrdiff_t 通常只能表示极大值为 2^{31} 的数，此数（大于 20 亿）看似很大，其实不然。很多操作系统允许你产生比这更大的文件，而所谓「兆字节（terabyte）」数据库更是司空见惯。如果你定义一个 iterator 用来遍历极大文件，ptrdiff_t 应该不够瞧。

对于一般的 Random Access Iterator，p2-p1 不见得型别为 ptrdiff_t。这个表达式会返回某个带正负号的整数型别：iterator 的 ***difference type***（差距型别）。

所有的 Input Iterators 都有 difference type。当我们写下 p2-p1 时，我们所要做的就是计算 range [p1,p2) 内的元素个数，这对于各种 Input Iterators 都是妥当的做法。（不过对于 Output Iterators 就不一样了，这也就是为什么 Output Iterator 不需要 difference type 的原因）

正如 value type 一样，某些算法在操作 iterator ranges 时必须取用 iterator 的 difference type。其中最简单的算法当推 count（p.214），它会计算某值 x 出现于 range [first,last) 中的次数。很显然，返回型别必须是 iterator 的 difference type，因为 count 的返回值可能有 first 至 last 之间的距离那么大。一个明显的解决办法是，让 difference_type 与 value_type 使用相同的机制。我们将 difference_type 定义为 iterator class 的嵌套型别，并把 difference_type 加入 iterator_traits 之中，如此一来便可以完全相同的方法来取用每个 iterator 的 difference type：

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& x) {
  typename iterator_traits<InputIterator>::difference_type n = 0;
  for ( ; first != last; ++first)
    if (*first == x)
      ++n;
    return n;
}
```

在 STL 的历史上，当初导入 iterator_traits 的主要动机之一，就是为了提供一个机制，使得以声明 count 的返回值。

3.1.3 Reference Type 和 Pointer Type

所谓 Forward Iterator，系用来指向某个内存位置。如果 p 是个 Forward Iterator，指向型别为 T 的某对象，那么表达式 *p 不能只是返回型别为 T 的对象，它必须返回一个 lvalue（左值）。所谓 lvalue 便是「用来代表某对象」的一个东西，这是 Stroustrup 的定义 [Str97]。

C++ 函数可以藉由「返回一个 reference」来返回一个 lvalue。「返回一个 reference」，正是 Forward Iterator 得以提供对其所指对象之读/写功能的原因。如果 p 是个 ***mutable*** iterator，而其 value type 为 T，正常情况下 *p 的型别应该是 T&。同样道理，如果 p 是个 ***constant*** iterator，正常情况下 *p 的型别应该是 const T&。一般而言，*p 的型别并不是 p 的 value type，而是 p 的所谓 ***reference type***。

指针与 reference，在 C++ 中的关系相当密切。如果我们能够返回一个 lvalue，代表 p 所指之对象，那么我们一定也能够有个什么东西，可以代表该对象的地址 — 也就是说，我们一定能够返回一个指针，指向该对象。

这些型别已经在 2.1.3 节的 node_wrap class 中出现过了。node_wrap 的 operator* 返回 Node&，operator->[7] 返回 Node*，提供对于 Node 字段的访问能力。换句话说 node_wrap class 的 ***reference type*** 是 Node&，而 ***pointer type*** 则是 Node*。

这些 associated type 同样也定义在 interator_traits 之中。reference 和 pointer 这两个嵌套型别的定义方式和 value_type 和 difference_type 类似。

3.1.4 算法的处理与 Iterator Tags

iterator_traits 最后定义的数个型别，更为抽象。要了解它们，我们不可光着眼于 iterator 自身，而要着眼于算法身上 — 虽然这些算法看起来并没有利用 iterator 的 associated types 做了什么。

常常你会发现，算法对于某种 iterator concept 有个实用定义，对于该 concept 之 refinement 却有另一种不同的定义。举个例子，假设我们有一个算法运用了 Forward Iterators，当然它也可以运用 Random Access Iterators，因为 Random Access Iterator 的每一个 model 必然都是 Forward Iterator 的一个 model。但「可用」并不意味「最佳」。有时候我们可以特别针对 Random Access Iterators 写一个更好的版本：使其 operator< 和 operator+= 的复杂度都是 *O(1)*。

最简单的例子就是我们在 2.3.5 节曾经简短谈论过的 advance。这个函数自身并不非常有趣，不过其他 STL 算法会把它当作基本元素来使用。实际上有三种不同定义的 advance：一个针对 Input Iterators，一个针对 Bidirectional Iterators，另一个针对 Random Access Iterators。

```
template <class InputIterator, class Distance>
void advance_II(InputIterator& i, Distance n)
{
  for ( ; n > 0; --n, ++i) {}
}
```

[7] operator->所提供的功能，不会超越我们在 operator* 上所定义的功能。表达式 p->val 只是 (*p).val 的速写罢了。所有 Input Iterators 都必须同时提供 operator* 和 operator->。

```
template <class BidirectionalIterator, class Distance>
void advance_BI(BidirectionalIterator& i, Distance n)
{
  if (n >= 0)
    for (; n > 0; --n, ++i) {}
  else
    for (; n < 0; ++n, --i) {}
}

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

其中 Input Iterator 版本是将 i 递减 n 次。Bidirectional Iterator 版本的基本做法相同，但允许逆向前进。Random Access Iterator 版本会用到 operator+=。我们没有提供 Forward Iterator 版本，因为没有理由让 Forward Iterator 版本拥有与 Input Iterator 版本不同的定义。

这三个版本之中，我们应该选择哪一个作为 iterator 的基本要素呢？不幸的是三者之中没有真正适合的。如果选 advance_II，面对 Random Access Iterators 将十分没有效率，因为原本应该是 *O(1)* 的行为，却变成了 *O(N)*。如果选择 advance_RAI，那就根本无法接受 Input Iterators。

我们真正需要的是某种可以结合以上三个版本的方法。观念上你可以想像成这样：

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n)
{
  if (is_random_access_iterator(i))
    advance_RAI(i, n);
  else if (is_bidirectional_iterator(i))
    advance_BI(i, n);
  else
    advance_II(i, n);
}
```

当然啦，我们不能够这样实现 advance，因为运行期才做选择实在是太慢了。我们必须在编译期就选对版本。我们真正要尝试的是将某些东西更加一般化，这对每一个 C++ 程序员来说应该都很熟悉：将 advance 函数重载（*overloading*）。

是的，C++ 允许你定义数个不同的函数，它们有相同的名字、不同的引数型别。当你调用某个重载函数，编译器会根据你所指定的引数型别，选择正确的版本。这和我们要做的事情极为类似，只不过现在我们是对 concepts 进行重载，而不是对 types（型别）进行重载。

C++ 语言并未直接支持「对 concepts 进行重载」的语法（因为 concepts 并非 C++ 语言的一部分），但如果我们能够找出一种「以 C++ 型别系统（type system）来表示 concepts」的方法，我们就能够以一般的「函数重载」来模拟「concepts 重载」。

第一步是为每一个 iterator concept 定义一个专属型别，作为 tag（标签）之用。我们不必在乎这些型别实际上是什么，只要它们独一无二就好了。

第二步是将三个 advance 版本加以重载 — 以上述所谓的标签型别（*tag types*）作为重载的识别凭借。这些重载函数用来取代 advance_II、advance_BI 和 advance_RAI。

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n, input_iterator_tag)
{
  // 实现代码同 advance_II。
}

template <class ForwardIterator, class Distance>
inline void advance(ForwardIterator& i, Distance n, forward_iterator_tag)
{
  advance(i, n, input_iterator_tag());
}

template <class BidirectionalIterator, class Distance>
void advance(BidirectionalIterator& i, Distance n, bidirectional_iterator_tag)
{
  // 实现代码同 advance_BI。
}

template <class RandomAccessIterator, class Distance>
void advance(RandomAccessIterator& i, Distance n, randomAccess_iterator_tag)
{
  // 实现代码同 advance_RAI。
}
```

我们并不对 tag 引数作任何动作。它们只是身份裁定者，一种识别凭证而已。它们存在的唯一理由就是为了区分不同版本的 advance。注意，这里也实现 advance 的「Forward Iterator 版本」，其实现代码与「Input Iterator 版」相同。

最后，我们必须写出一个上层函数，由它来调用重载后的 advance。此函数需要两个引数：iterator i 和距离 n。它将这两个引数传给 advance，并加上第三个引数：上述五个 tag types 之一。因此，上层函数必须有能力推导出某个 iterator type 的 tag type。

本节内容到处都是类似上述的问题，它们有着一致的解决办法。是的，我们让每个 iterator 拥有一个相应的 *iterator category* type（迭代器分类型别）。iterator 的 category（分类），指的便是「该 iterator 所隶属之最明确 concepts」的「相应 tag type」。例如，int* 既是 Random Access Iterator 的 model，

又是 Bidirectional Iterator、Forward Iterator、Input Iterator 和 Output Iterator 的 model，所以 int* 的分类是 random_access_iterator_tag。

和其他 associated type 一样，iterator_category 被定义为 iterator_traits 内的嵌套型别。这使我们得以写出「必须依据 iterator 的分类而有不同设计」的 advance 函数及其他算法：

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

STL 把这五个 tag types 定义为空类：

```
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 tag types 都是 classes，而且有继承关系，此点其实无关紧要。重要的是它们独一无二，因此可在函数重载机制中派上用场。继承的运用可以带来方便，因为我们写了四个版本的 advance，但只有三种有所不同 —「Forward Iterator 版本」只是转调用「Input Iterator 版本」而已。如果令 forward_iterator_tag 继承自 input_iterator_tag，我们就可以省略那个平淡无奇而且无关痛痒的转调用函数。

3.1.5 把一切统合起来

iterator_traits 的某些机制所蕴含的意义十分微妙而深远，不过实现起来却相当简单：

```
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;
};

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;
};
```

```
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;
};
```

只有当你需要定义自己的 iterators 或算法，你才需要担心这个机制。当你需要定义自己的算法时，以下两个理由告诉你为什么你可能需要用到 iterator_traits:

- 你必须返回某值，或是声明临时变量，而其型别与 iterator 的 value type 或 difference type 或 reference type 或 pointer type 一致。
- 你的算法类似 advance，必须根据 iterator 的分类而决定不同的实现方法。如果没有 iterator_traits 机制，有时候你只好被迫在「一般化，却没有效率」与「有效率，但过度狭隘」两者之间做一个难以取舍的决定。

有些 STL 算法已经运用这样的机制，所以每当你定义新的 iterators，你必须确保它可以和 iterator_traits 运作得当。幸运的是做法非常简单易行。

每当你定义一个新的 iterator class I，你就必须在这个 class 中定义五个嵌套型别：iterator_category、value_type、difference_type、pointer 和 reference，要不然就得针对你的 class I，明白地令 iterator_traits 为 I 而特化，就好像 iterator_traits 明白地针对指针而特化一样。第一种做法几乎总是比较简单，尤其 STL 内含一个辅助类，base class iterator，让事情更好办：

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

欲确保 iterator_traits 能够针对新的 iterator class I 有适当的定义，最简单的做法就是让 I 继承 iterator。base class iterator 不含任何 member functions 或 member variables，所以继承它应该不会招致任何额外开销。

3.1.6 没有 iterator_traits，如何制作 Iterator Traits（迭代器特征）

iterator_traits class 是颇为晚近的发明，HP STL 原始版本 [SL95] 并未包含它。HP STL 并没有 value types、difference types 和 iterator categories 的概念，但它使用不同（较笨拙）的机制来取用这些型别。那些机制如今已不再是 C++ 标准规格的一部分了，不过许多 STL 实现品为了向下兼容，仍然提供那些机制。

早期的机制是一组 iterator 查询函数：distance_type、value_type 和 iterator_category。每个查询函数只需一个引数，是个 iterator；而该 iterator 之「associated type 相关信息」会由函数返回。

iterator_category 仅需要一个 iterator 引数。此函数会返回该 iterator 的分类标签。也就是说，如果其引数是 Random Access Iterator 的一个 model，便返回型别为 random_access_iterator_tag 的一个值；如果其引数是 Bidirectional Iterator 的一个 model，便返回一个型别为 bidirectional_iterator_tag 的值；依此类推。同样道理，如果 i 是个 iterator，其 value type 为 V 且 difference type 为 D，那么 value_type(i) 返回型别为 V* 的某值，difference_type(i) 返回型别为 D* 的某值。这三个案例中，实际返回值无关紧要，我们关心的是其返回型别。

所有这些函数和 iterator_traits 相比，都称不上方便，因为它们没有直接提供型别信息。例如，使用 iterator_traits，我们可以轻易写出一个函数，将两个 iterator 的值交换：

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 p, ForwardIterator2 q)
{
  typename iterator_traits<ForwardIterator1>::value_type tmp = *p;
  *p = *q;
  *q = tmp;
}
```

如果使用旧机制，就无法直接声明 tmp；必须分两个函数来完成任务：

```
template <class ForwardIterator1, class ForwardIterator2, class T>
void iter_swap_impl(ForwardIterator1 p, ForwardIterator2 q, T*)
{
  T tmp = *p;
  *p = *q;
  *q = tmp;
}

template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 p, ForwardIterator2 q)
{
  iter_swap_impl(p, q, value_type(p));
}
```

在 HP STL 原始版本中，每个 iterator type 都必须提供这三个函数。如果你要定义新的 iterator，确保

这三个函数都有定义的最简单方法，就是令你的 iterator class 继承前述五个 base classes 之一：input_iterator、output_iterator、forward_iterator、bidirection_iterator 和 random_access_iterator。

但是，三个 iterator 查询函数，以及五个 iterator base classes，都已不再是 C++ 标准规格的一部分了。Iterator 查询函数已经被 iterator_traits class 取代，iterator base classes 亦已被单独一个 base class iterator 取代。尽管如此，许多 STL 实现品为了向下兼容，仍然提供旧机制，你可以在年代较早的代码中看到它们。

3.2 定义新组件（New Components）

STL 的设计允许我们加以扩充。你可以使用既有的 STL 算法，搭配新定义的 iterators，或是使用既有的 iterators，搭配新的算法。

关于扩充性，其原因如今应该很明显了。Iterators 与算法都是根据 iterators 的条件而撰写。一个 STL 算法如果作用于 Forward Iterator 所形成的某个区间之上，它就只会使用 Forward Iterator concept 所保证的性质。而当你要模塑一个 Forward Iterator iterator，必须提供 Forward Iterator 要求的所有功能。

由于算法和 iterator types 都以 iterator concept 作为其接口，因此每当你要定义新的算法或新的 iterators 时，应该把 iterator concepts 视为一份清单。这份清单对于新的 iterators 而言，代表「应提供的机能」，对于新算法而言则表示「假设已经存在的机能」。第 7 章会详细介绍 iterator concepts。

基本上，iterator 必须做两件事：(1) 它必须指向某物；(2) 它必须能够遍历任何一个有效区间。一旦你定义了 operator* 与 operator++，通常剩下的 iterator 行为就很简单了。

你必须确定 iterator 被正确地定义为 ***constant***（不变的）或 ***mutable***（可变的）。这是常犯的错误之一。不过 Input Iterator 不在乎这一点，因为 Input Iterator 具有只读性；Output Iterator 也不在乎这一点，因为 Output Iterator 具有只写性。但对于其他类型的 iterators，这一点就很重要。

举个例子，2.1.3 节的 node_wrap class 是个 mutable iterator。它允许你更改其所指之值。「const 的正确性」在 C++ 十分重要。C++ 有所谓的 const 指针，可避免你一不小心更改了不应被更改的数据结构。当你要定义能够遍历某数据结构的 iterators 时，你大概不会只定义一个 iterator。通常你会定义一对 iterators，其中之一是 ***constant***，另一个是 ***mutable***，它们有相同的 value type，并允许从 mutable 转换为 constant（逆向转换并不成立）。

也许你认为这么做没有必要，也许你认为像 node_wrap 这样的 iterator 可以同时承担 constant 与 mutable 双份责任。其实不然。有两个地方是你想像中以为能够放置 const 之处，但这两个地方都是错的。是的，const node_wrap<T> 并不是个 constant iterator。差别在于是否你可以使用这个 iterator

来更改其所指之值。既然 operator* 返回的是 T& 而非 const T&，所以你可以改变其所指之值。node_wrap<const T> 看起来似乎比较接近正确做法（至少 operator* 的确返回 const T&），可惜这还是错的。是的，虽然 node_wrap<const T> 的 value type 是 const T，但 3.1.1 节告诉我们，iterator（即使是 constant iterator）的 value type 不应该修饰为 const。

mutable iterator class node_wrap<T> 的 value type、reference type 和 pointer type 分别是 T、T& 和 T*。至于对应的 constant iterator，上述型别分别是 T、const T& 和 const T*。我们真的需要两个分离的 classes：*constant* 版和 *mutable* 版 — 尽管我们可以利用 template 或继承机制来分解两个 classes 之间相同的部分，

「constant 版本」和「mutable 版本」之间最重要的差异，就是其 pointer type 和 reference type。这似乎暗示我们或许可以只设计单一一个 iterator class，并以上述型别作为 template 参数。两版本间的其他差异还包括：你可以以一个 mutable iterator 构造出一个 constant iterator，反之不成立。然而有个很简单的方法（尽管有点隐微），却可以做到此事：定义一个 constructor 并令其引数为一个 mutable iterator。这么一来，对一个「型别为 template 参数」的值而言，它是个 copy constructor，对其他型别的值而言，它是我们所需要的 conversion constructor。

下面就是 node_wrap 和 const_node_wrap iterator classes 的完整定义。这两个 classes 奉行 Forward Iterator 的所有条件。（我以第 7 章所列规范来检视这两个 classes，确定它们遵照所有条件。当你要定义一个新的 iterator class 时，你也应该像我一样。）

```
template <class Node, class Reference, class Pointer>
struct node_wrap_base
  : public iterator<forward_iterator_tag, Node,
                    ptrdiff_t, Pointer, Reference>
{
  typedef node_wrap_base<Node, Node&, Node*>              iterator;
  typedef node_wrap_base<Node, const Node&, const Node*> const_iterator;
  Pointer ptr;

  node_wrap_base(Pointer p = 0) : ptr(p) {}
  node_wrap_base(const iterator& x) : ptr(x.ptr) {}

  Reference operator*() const { return *ptr; }
  Pointer operator->() const { return ptr; }

  void incr() { ptr = ptr->next; }

  bool operator==(const node_wrap_base& x) const { return ptr == x.ptr; }
  bool operator!=(const node_wrap_base& x) const { return ptr != x.ptr; }
};

template <class Node>
struct node_wrap : public node_wrap_base<Node, Node&, Node*>
{
  typedef node_wrap_base<Node, Node&, Node*> Base;
```

```
  node_wrap(Node* p = 0) : Base(p) {}
  node_wrap(const node_wrap<Node>& x) : Base(x) {}
  node_wrap& operator++()
    { incr(); return *this; }
  node_wrap  operator++(int)
    { node_wrap tmp = *this; incr(); return tmp; }
};

template <class Node>
struct const_node_wrap : public node_wrap_base<Node, const Node&, const Node*>
{
  typedef node_wrap_base<Node, const Node&, const Node*> Base;
  const_node_wrap(const Node* p = 0) : Base(p) {}
  const_node_wrap(const node_wrap<Node>& x) : Base(x) {}
  const_node_wrap& operator++()
    { incr(); return *this; }
  const_node_wrap  operator++(int)
    { const_node_wrap tmp = *this; incr(); return tmp; }
};
```

其实单单一个 base class node_wrap_base 几乎就已足够。我们从 node_wrap_base 衍生出 node_wrap 和 const_node_wrap，而非单单只提供 base class node_wrap_base，唯一真正的理由是，撰写 node_wrap<T> 比撰写 node_wrap_base<T, T&, T*> 方便得多。C++ 并不提供 template typedefs，所以我们必须以继承或内含（containment）替代之。

如果你不在乎方便与否，或如果你定义出其他化名（alias）做法，那就没有必要烦心这些派生类了。

3.2.1 Iterator Adapters

iterator adapter 是一种有用的 iterator。大体上来说，***adapter*** 是「能够将某接口转换成另一种接口」的东西。就某种意义而言，node_wrap class 就是一种 adapter，因为它为链表（linked list）提供了一个 STL 接口，而该链表完全不知道 iterator 或 STL 是什么。

另一种 iterator adapter 则提供不同的接口给既有的 STL 组件。STL 定义了一些这种 iterator adapters，包括 front_insert_iterator、back_insert_iterator 和 reverse_iterator。

其中最有趣的当属 reverse_iterator（p.363）。它大概是最单纯的一种 adapter 了。它只有一个 template 参数，是个 Bidirectional Iterator Iter；reverse_iterator<Iter> 的行为就像 Iter，只不过它是逆向移动而非一般的向前移动，换句话说，reverse_iterator<Iter> 的 operator++ 等价于 Iter 的 operator--。

如果你要定义类似 reverse_iterator 这样的 iterator adapter，不必明白处理 ***constant*** 或 ***mutable*** 的问题。以 reverse_iterator<Iter> 为例，它有着与 Iter 相同的 reference type 和 pointer type，所以如果 Iter 是 mutable，它就是 mutable。而且只有当 Iter 是 mutable，它才是 mutable。

3.2.2 定义 Iterator 时的建议

以第 7 章作为 iterator 的需求清单。

- 注意 iterator 应该是 constant 或 mutable。两者间的主要差别在于其 reference type 和 pointer type 是 `T&` 和 `T*`，或是 `const T&` 和 `const T*`。也许你会想要定义一对 iterators，其中一个是 constant，另一个是 mutable。如果这么做的话，你还应该定义一个「mutable iterator 转换为 constant iterator」的办法，但不必定义其逆向转换。

- 你必须确定 `iterator_traits` 能够正确作用于你的 iterator 身上。这几乎就是意味你必须定义嵌套型别，而最简单的做法就是以 class `iterator` 作为你的 base class。

- 尽可能提供较多的 iterator 操作行为，除非对效率有所影响。如此一来你的 iterator 就比较可能搭配较多的算法。是的，定义一个 Random Access Iterator（如果可以的话）比定义一个 Bidirectional Iterator 更好。如果你想要定义一个能够遍历某数据结构的 iterator（对照反例则是那些用来表现 I/O 的 iterator。译注：例如 `istream_iterator` 和 `ostream_iterator`），那么至少应该是个 Forward Iterator。

- 面对 Forward Iterator，你应该确实遵守所谓「iterator 相等（equality）」的基本原则：只有当两个 iterators 指向相同的对象，它们才被视为相等。

3.2.3 定义算法时的建议

- 尽可能做最少的假设。如果你可以把你的算法设计为与 Input Iterator 或 Output Iterator 搭配运作而不失效率，你就应该这么做。这也意味你的算法可以有最大弹性地与不同种类的 iterators 搭配使用。

- 如果你能够将你的算法写成与 Input Iterator 搭配，但是另一个运用 Forward Iterator 的做法更具效率，那么你不应该试着在效率与一般化之间做选择，因为两者都很重要，你不应该放弃任何一个。你应该利用 3.1.4 节所描述的分派（*dispatching*）技术来解决这个问题。

- 你应该查阅第 7 章所列的 iterator 需求条件，确定自己不会因为疏忽，而对「算法中的 template 参数的性质」做出超越「其所相应之 iterator concept 实际定义的条件」更多的假设。当你测试你的算法时，应该尽可能使用与该 iterator concept 接近的具象型别（concrete type）。举个例子，如果你写一个与 Input Iterator 搭配的算法，那么你应该使用 `istream_iterator` 来测试，而非使用 `int*`；如果你写一个与 Output Iterator 搭配的算法，你就应该使用 `ostream_iterator` 来测试。如果你写一个与 Forward Iterator 搭配的算法，那么「container class `slist`（译注：p.448）所定义的 iterator」就是你进行测试时的最佳选择。

- 特别留意「空区间（empty ranges）」。大多数算法将「空区间」视为合理，而且「空区间」是重要的测试条件。

3.3 总结

STL 的中心特性是「作用于 range 之上的泛型算法」。STL 内含许多事先定义的算法，本书 11~13 章列有其相关文档。这些算法极富变化，涵盖许多操作行为，诸如「子字符串比对」与「随机排列」等等。

所有 STL 算法都会作用在「以 iterators 形成的 range」上；所有 iterators 隶属于五个与指针极为类似的「concepts 家族」。并非所有算法都可以和任何可想像的数据结构连接搭配，例如，想要在单向链表上执行 `quicksort` 就绝对不可能。不过 iterators 自身提供有一种分类方法，可以让使用者知道「何种算法可用于何种数据结构」。我们已经在本章以及第 2 章中看过一些算法，以不同的 iterator concepts 来操作。此外，某些算法会根据它们所接获的 iterator 的种类，分派（*dispatch*）给不同的实现代码处理。

STL 的五个 iterator concepts 是「算法」与「其所作用之数据结构」间的粘胶。由于这些 concepts，我们得以重复利用既有的算法，搭配新的数据结构，或是重复利用既有的数据结构，搭配新的算法。iterators 再加上 function objects（第 4 章主题），使我们得以在非常多样的环境下，重复使用某些算法。

第 4 章

Function Objects

函数对象

如同我们在第 2 章所见，许多作用于 range 身上的算法，可以单单以 iterator 的观点来撰写。然而有时候如此狭隘的观点会受到过度的束缚。

第 2 章所提的线性查找，便是这样一个例子。在 2.1.3 节中我们学到如何写一个名为 `find` 的泛型算法，它会查找 range `[first,last)` 内与引数值相等的第一个元素。到目前为止，这样的做法堪称正确，不过对于未来的需要还嫌不足。下面是 Knuth 对于一般查找问题的定义，载于 *The Art of Computer Programming* [Knu98b] 一书第 6 章：

> 通常我们可以假设有 N 笔已排序的记录，我们的问题就是要找出某笔记录。由于「排列」之故，我们假设每笔数据都包含有一个特别的字段，叫做 *key*……查找算法有一个所谓的「引数 *K*」，它必须在 N 笔排过序的记录中找出某个「以 *K* 为其 *key*」的记录。

先前的 `find` 算法不够一般化，因为它没有完全满足上述定义。从 Knuth 的定义观之，`find` 有一个引数，它会查找某个「与该引数全面相同」的记录，而不是查找某个「*key* 字段等于该引数」的记录。换言之，只有在特殊状况下，也就是记录的 *key* 刚好就是记录自身，`find` 才满足上述定义。

举个例子，假设我们有某种记录的集合，每笔记录代表一个人。我们可能想以「姓氏」或「身份证号」来查找某个人。这种情况下我们不能使用 `find`，因为 `find` 只能用来测试相等性（equality）。是的，我们可以利用 `find` 来寻找第一笔「等于某特定值」的记录，却不能利用它来寻找第一笔「符合较宽松标准」的记录。所以我们无法利用它来测试姓氏相等与否。

4.1 将线性查找一般化

我们已经在 `find` 之中将两种东西参数化了：(1) 查找对象的型别；(2) 该种对象在数据结构中的组织方式。为了全面一般化，我们还需再指明一件事：如何判断找到了正确的元素。这导致另一个新算法 `find_if`：

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred)
{
  while (first != last && !pred(*first))
    ++first;
  return first;
}
```

算法 find_if 非常类似 find。它会查找某个「满足 pred 条件」的元素，而非查找某个「等于引数 T」的元素。很显然，它比 find 更一般化。事实上它甚至比 Knuth 为查找所下的定义更一般化。该定义寻找的是某个「具有特定 *key*」的记录，而 find_if 却可用来寻找「满足某一条件」的记录，和 *key* 全无关联（当然啦，它也可以用于 Knuth 所描述的查找定义）。

在 find_if 中，测试条件 pred 并非被声明为一个函数指针（这是 C 的习惯做法），而是声明为一个 template 参数 Predicate。我们已经在第 1 章看过了某些像 pred 一般的东西，此即所谓的 ***function object***，或称为 ***functor***（二者同义）。换言之，pred 是一种可以像函数一样被调用的东西。

Function objects 在很多不同的场合中都很有用。在 STL 中，它们主要的用途，正如 Stroustrup [Str97] 所说：「以 template 引数来指明所要采行的策略」。以本例而言，传入 find_if 的那个 template 引数即决定了 find_if 要以何种方式（策略）来查找。

最简单的 function object 当推一般的函数指针（function pointer）。举个例子，如果想在整数数组中找出第一个偶数，你可以定义一个简单的函数，让它试验 int 为偶数或奇数：

```
bool is_even(int x) { return (x & 1) == 0; }
```

现在你可以把 is_even 指针当作引数，传入 find_if：

```
find_if(f, L, is_even)
```

返回值将是一个 iterator，指向 range [f,L) 中第一个偶数（如果返回值为 L，表示该 range 中没有任何偶数）。此例之中我们传入一个函数指针给 find_if。函数指针 is_even 的型别是 bool (*)(int)。C++ 会执行正常程序的型别推导（type deduction），将 bool (*)(int) 认许为正规的 template 参数 Predicate。

不一定非得使用函数指针不可。是的，function call 操作符 operator()可以被重载，所以只要某个型别具有妥善定义的 operator()，你便可以将该等对象传给 find_if 当作引数。例如，你可以将测试条件定义为一个 class，而非一个函数：

```
template <class Number> struct even
{
  bool operator()(Number x) const { return (x & 1) == 0); }
};
```

此声明式中的 const 意指：operator() 是个 const member function，不会改变 even 对象内容。此例之中，这其实是个空的保证，因为 even 根本就没有任何 data members 可被变动。然而这是一般规则，也是个好主意，所以请尽可能将 function object 的 operator() 声明为 const member function。当你定义一个像 even 这般平淡无奇的 class 时，可能很容易就忘了这回事，然而这却非常重要。因为如果你要以 by const reference（而非 by value）的方式传递 function object，那么除非 operator() 是个 const member function，否则你就不能使用它。

我们几乎能够像使用函数 is_even 一样地使用 class even。如果要在某个 range 中寻找第一个偶数，可以这么写：

```
find_if(f, l, even<int>());
```

这里必须使用看起来有点奇怪的表达式 even<int>()，理由很简单：even<int> 是个 class，所以我们必须将该种 class object 传给 find_if。所以我们必须调用其 constructor。

class even 并不是什么有趣的 function object。它的确是比函数 is_even 更有弹性（可搭配使用任何整数型别，而非仅仅 int 而已），也更有效率（因为 operator() 被声明为 inline，所以使用 even 时不需任何函数调用），不过除此之外似乎就没什么特别了。它只是个以 class 形式包装的函数罢了。

Function object 并不局限于这般简单形式。它可以像一般 class 一样拥有 member functions 和 member variables。也就是说，你可以利用 function objects 来表现「具有局部状态（local state）之函数」。

我们可以利用「局部状态（local state）」来表示 Knuth 所描述之查找问题。举个例子，我们可以利用这种方式来定义一个 function object，简化「以姓氏查找某个集合」的做法。我们以 find_if 来查找满足此一条件的元素。本例中的测试件就是「元素的 last_name 字段等于某特定值」：

```
struct last_name_is
{
  string value;
  last_name_is(const string& val) : value(val) {}
  bool operator()(const Person& x) const {
    return x.last_name == value;
  }
};
```

下面是使用这个 function object 的方法：

```
find_if(first, last, last_name_is("Smith"))
```

像 last_name_is 这样的 classes，便是一个完全一般化的 function object。它拥有局部状态，以及一个 constructor；而由于具有 operator()，使得其对象看起来就像个函数：接受一个型别为 Person 的引数并返回 bool 值。

很难以 C 语言撰写类似 last_name_is 的东西。最接近的做法或许是一个「用到全局变量」的普通

函数，但仍然有不小的差别，因为 C 语言不允许为了查找不同的姓氏而让这样的函数实体存在一份以上。（某些语言对此可能有更简单的做法，例如 Scheme 和 ML，因为「函数具备局部状态」正是其基本原则。）

4.2 Function Object Concepts（函数对象概念）

4.1 节中我们看到了三个 function objects：`is_even`、`even` 和 `last_name_is`。它们都可以「如函数一般地被调用」，也就是它们都有 `operator()`，除此之外的相同点很少。很显然，任何 function object concept 的基本条件只是：如果 `f` 是个 function object，我们可以将 `operator()` 施行于 `f` 身上。

4.2.1 单参（Unary）与双参（Binary）Function Objects

4.1 节的三个 function objects 都有这样的性质：如果 `f` 是任何一种 function object，针对某个值 `x` 你可以写 `f(x)`。也就是说，`f` 看起来像是个需要单一引数的函数。

但这并非唯一合理的 function object 种类。「单一引数」自身并没有什么特别。我们以「单一引数的 function object」来对 `find` 一般化，同样地我们也能够以「多个引数的 function object」来对 `adjacent_find`（2.3.3 节）一般化：

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last,
              BinaryPredicate pred)
{
  if (first == last)
    return last;
  ForwardIterator next = first;
  while (++next != last) {
    if (pred(*first, *next))
      return first;
    first = next;
  }
  return last;
}
```

STL 内含两个版本的 `adjacent_find`（换句话说它被重载了），严格地说其中只有一个版本才真正用到 function object。这是很常见的一种体制。当算法需要一个函数作为其引数，它便是将某些行为参数化了 — 本例为「相邻元素的测试」。通常，有某种缺省行为是我们常常会用到的，如果真是这样，STL 会为该算法提供一个版本，供应缺省行为。

我们已经看过了，有的算法使用单一引数的 function object，有的算法使用两个引数的 function object。另有一些算法（如 generate，p.251）使用的是不需要引数的 function object。因此，基本的 function object concepts 是 Generator、Unary Function 和 Binary Function。这些 concepts 所描述的是能够以 f()、f(x) 和 f(x,y) 调用的对象（其实还可以扩充为三元或多元，但实际上目前的 STL 算法并未用到两个引数以上的 function objects）。STL 所定义的其他 function object concepts，都是这三个 concepst 的 refinements（加强、精炼）。

这三个 concepts 关系十分密切（都用来描述 function objects），但我们没办法说它们是其他某个 function object concept 的 refinements。Concept 的需求条件会涉及特定的表达式（expressions），而三个表达式 f()、f(x) 和 f(x,y) 本质上就不相同。

4.2.2 Predicates 和 Binary Predicates

目前为止，我们在本章所看到的和 function objects 有关的例子，都是用来测试某个条件成立与否，也就是返回 true 或 false。是否所有的 function objects 都是这种形式呢？

当然不是。Function objects 是非常一般化的概念，它们可以将任何种类的行为参数化。几乎任何算法都能以「将其行为的某一部分抽象化为 function object」的方式来加以一般化。是的，即使是 copy（p.233）这般通俗的算法 — 把某个区间的数值复制到另一个区间去。copy 有一个明显的一般化版本：不直接复制出现于「输入区间」的数值，而是将那些数值经过某种转换后再复制。也就是说，不再执行这样的行为：

$$y_i \leftarrow x_i \qquad (4.1)$$

而改执行这样的行为：

$$y_i \leftarrow f(x_i) \qquad (4.2)$$

注意：当 f 是所谓的证同函数（identity function，*id*）时，式 4.1 便是式 4.2 的特殊状况。

这样一个经过一般化的 copy 算法，STL 称为 transform(p.40)。其实现代码除了使用 function object 之外，看起来非常像 copy：

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction f)
{
  while (first != last) *result++ = f(*first++);
  return result;
}
```

很显然，UnaryFunction 并不限一定得返回 true 或 false。它可以返回任何种类的数值，只要其返回型别可以赋值（*assign*）给 OutputIterator 即可。举个例子，你可以利用 transform 将某个区间内的数值转成负数。

「返回 bool 值」的 function objects 特别重要 — 它们比完全一般化的 function objects 更常被使用。对于涉及测试行为的算法如 find_if，这种 function objects 非常有用。

「单一引数并返回 true 或 false」的 function objects，称为 Predicate，这是 Unary Function 的一个强化（refinement）。同样道理，Binary Predicate 是 Binary Function 的一个强化。所谓 Binary Predicate 是一种「需要两个引数并返回 true 或 false」的 function object。

4.2.3 相关型别（Associated Types）

所谓「function object 的相关型别」，是指其引数与其返回值的型别。Generator 具有一个相关型别（即其返回型别），Unary Function 具有两个相关型别（即其引数型别与返回型别），Binary Function 具有三个相关型别（第一引数与第二引数的型别，以及返回型别）。举个例子，class even<T> 就有一个引数型别 T 和一个返回型别 bool。

和 iterator 一样，有时候取用这些相关型别是很有帮助的。面对 iterators，我们导入所谓的 iterator_traits（3.1 节）来解决问题，面对 function objects，情况类似。Function objects 被定义为 classes，所以就像 iterators（也被定义为 classes）一样，它们可以拥有嵌套型别。我们可以定义 even<T>::argument_type 为 T，even<T>::result_type 为 bool。同时，就像 iterator 一样，我们也定义两个空的基类 unary_function（p.371）和 binary_function（p.372），其中都只有一些 typedefs。

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg    argument_type;
  typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1   first_argument_type;
  typedef Arg2   second_argument_type;
  typedef Result result_type;
};
```

修改 even<T> 的方式有二，一是明白声明出两个相关型别，要不就更简单地继承 unary_function:

```
template <class Number>
struct even : public unary_function<Number, bool>
{
  bool operator()(Number x) const { return (x & 1) == 0; }
}
```

这不是完整的解决方案，因为它不能适用于一种很重要的 function object 身上：函数指针（例如 bool (*)(int)）。函数指针是内建型别，不是 class，所以无法为它定义嵌套型别。

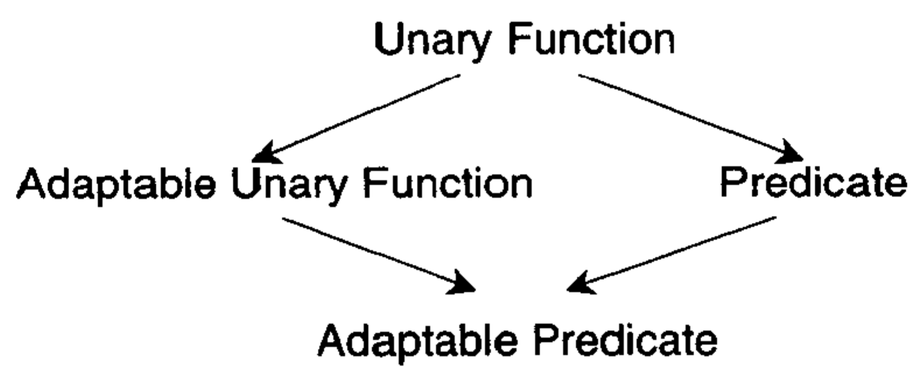

图 4.1 「单一引数之 function object」的各种概念（concepts）

我们可以像解决 iterator 问题一样地解决这个问题。我们可以定义一个 trait class，名为 `unary_function_traits`，然后为 `unary_function_traits` 定义一个「针对函数指针」的局部特殊版（partial specialization）。然而，STL 并没有 `unary_function_traits` class 这样的东西。为什么没有？哦，没有任何人或任何文档告诉我们原因。不过，function objects 虽然很重要，却不像 iterators 那般到处存在。许多运用了 function objects 的算法，并不会明白取用其引数型别与返回型别，所以没有必要对每一个 function object 都提供那些型别的取用方法。

为此，STL 把 ***function objects*** 和 ***adaptable function objects*** 区分开来。一般而言，一个 function object 会有一个引数型别（如果是 Binary Function，就有两个引数型别）与一个返回型别，但程序不需要知道那些型别名称。至于 adaptable function object 则会指明引数与返回型别为何，它会内含嵌套的 typedefs，所以程序中可以指定并使用那些型别。如果型别 `F1` 是 Adaptable Unary Function 的一个 model，则它必须定义有 `F1::argument_type` 和 `F1::result_type`；如果型别 `F2` 是 Adaptable Binary Function 的一个 model，则它必须定义有 `F2::first_argument_type`、`F2::second_argument_type` 和 `F2::result_type`。STL 提供两个基类 `unary_function` 和 `binary_function`，这使得定义 Adaptable Unary Function 和 Adaptable Binary Function 的工作简化许多。

就像 Adaptable Unary Function 是 Unary Function 的一个强化（refinement）一样，Adaptable Predicate 是 Predicate 的一个强化。图 4.1 显示出不同的 Unary Function concepts。Binary Function concepts 的相关观念也很类似。

11~13 章之中没有任何一个算法需要用到 adaptable function objects。例如 `find_if` 用的是 Predicate 而 `adjacent_find` 用的是 Binary Predicate。那又何必自找麻烦地导入未曾用到的 Adaptable Unary Function 和 Adaptable Binary Function 两个概念呢？

答案当然是：它们会被用到 — 只不过预定义好的 STL 算法没有使用它们而已。它们之所以被定义（并命名为「adaptable」），乃因它们用于 function object adapters 之中。

4.3 Function Object Adapters（函数对象配接器）

Adapter 是一种「将某接口转换成另一接口」的组件。例如 3.2 节的 `reverse_iterator` 就是一个 iterator adapter。由于以下因素，function object adapters 格外引人注目：

函数的组合（function composition）是数学与计算机科学的基本操作行为。

定义众多特殊用途的 function objects，是件烦人的工作。只要定义少量精选的 function object adapters，便可以「以简驭繁」。

有很多非常有用的 function object adapters 都很容易定义。（如果你和我一样，你会发现定义一个新的 function object adapter，最困难的便是为它取个好名字！）

举个例子，你可以这样寻找某整数区间内的第一个偶数：

```
find_if(f, l, even<int>())
```

但如果你要找某整数区间内的第一个奇数呢？当然你可以另写一个 function object `odd<T>`，但这似乎有点愚蠢。奇偶数的测试是互补的，所以一旦你定义好其中一个，就不必再定义另一个。我们应该重复运用代码，而非重复撰写代码。

解决之道是利用一个已由 STL 提供的 function object adapter：

```
template <class AdaptablePredicate>
class unary_negate
{
private:
  AdaptablePredicate pred;

public:
  typedef typename AdaptablePredicate::argument_type      argument_type;
  typedef typename AdaptablePredicate::result_type        result_type;

  unary_negate(const AdaptablePredicate& x) : pred(x) {}
  bool operator()(const argument_type& x) const {
    return !pred(x);
  }
};
```

如果 `f` 是型别 `F` 的一个 predicate，则 `unary_negate<F>(f)` 是「`f` 之反相（negation）」的一个 predicate。也就是说，当且仅当 `f(x)` 是 `false`，且 `f_neg` 是 `f` 的 predicate，那么 `f_neg(x)` 必为 `true`。

上述的 template 参数 `AdpatablePredicate`，如其名称所示，必须是 Adaptable Predicate 的一个 model。由于 `unary_negate` 的引数型别与 `AdaptablePredicate` 的引数型别相同，所以我们必须能够取用这个型别。再者，`unary_negate` 自身也是 Adaptable Predicate 的一个 model，因为它包含了

适当的嵌套型别。这对 function object adapter 来说十分典型，这也就是 adaptable function object 被称为「adaptable」的原因。

unary_negate adapter 的唯一问题是，表达式 unary_negate<F>(f) 没必要如此累赘。这的确很啰唆，因为我们必须供应 function object f 与 type F。纯粹为了方便，我们可以定义一个小小的辅助函数，使 unary_negate object 的产生得以简化许多：

```
template <class AdaptablePredicate>
inline unary_negate<AdaptablePredicate>
not1(const AdaptablePredicate& pred) {
  return unary_negate<AdaptablePredicate>(pred);
}
```

我们可以把 not1 想像成「更高层级」的函数，或是作用于函数之上的函数。其引数是个 function object，而其返回值是该 function object 的负值。最后，我们终于能够以如下写法，找寻某整数区间内的第一个奇数：

```
find_if(first, last, not1(even<int>()))
```

将不同的 function object adapters 串连起来，我们可以以预定义的少量简单组件，构筑出复杂的 function objects，而不必撰写新的 function object 或 class。

如果你需要将一般函数指针传递给 function object adapter，可以利用 pointer_to_unary_function（p.424）将函数指针转换为 Adaptable Unary Function。（注意，只有在你需要将函数指针视为一个 Adaptable Unary Function 时，才需动用 pointer_to_unary_function。正如我们在「将函数指针传给 find_if」的例子中所见，就自身条件而言，函数指针已经是个完美的 Unary Function 了。）

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
private:
  Result (*ptr)(Arg);
public:
  pointer_to_unary_function() {}
  pointer_to_unary_function(Result (*x)(Arg)) : ptr(x) {}
  Result operator()(Arg x) const { return ptr(x); }
};

template <class Arg, class Result>
inline pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*x)(Arg) {
  return pointer_to_unary_function<Arg, Result>(x);
}
```

另有一个类似的 adapter，pointer_to_binary_function，供双参函数使用。

STL 还定义有其他数个 function object adapters。其中最重要的是：(1) binder1st 与 binder2nd，可将 Adaptable Binary Function 转换为 Unary Function；(2) mem_fun_t adapter 及其变形，它们除了「作

用于 member function 而非全局函数」之外，极类似 pointer_to_unary_function；(3) unary_compose，能将两个 function objects f 和 g 转换成单一一个 function object h，使得 h(x) 等于 f(g(x))。

4.4 预定义的 Function Objects

除了 adapter 之外，STL 还包含许多基本的 function objects 作为基础素材之用。更明确地说，STL 包含了一些基本数值运算，其中涵盖「比较」运算。

STL 所定义的基本数值运算是：plus、minus、multiplies[8]、divides、modulus 与 negate。其中 netgate 是 Adaptable Unary Function 的一个 model，其他都是 Adaptable Binary Function 的 models。

基本的比较运算包括：equal_to、not_equal_to、greater、less、greater_equal 和 less_equal。它们全都是 Adaptable Binary Function 的 models。在这些用于比较运算的 function objects 中，最重要的显然是 equal_to 与 less，因为很多算法都必须测试两数值相等与否，或测试某数值是否小于另一数值。每当这些操作行为被 function object 参数化，equal_to 和 less 都是对应的缺省值。

「一个对象是否小于另一个对象」的测试行为是如此常见而重要，STL 特别为它规范了一个专属概念：Strick Weak Ordering，这是 Binary Predicate 的一个强化（refinement）。Strick Weak Ordering 是一个 binary predicate，用来表现一种有着良好行为的顺序（ordering）关系。究竟是什么造成某个顺序（ordering）关系「有着良好的行为」呢？其正式规则十分微妙（你可以在 p.119 找到它们），但是要遵守这些规则并不困难。这就好比是将「我们直觉预期的顺序关系」加以形式化一样。

4.5 总结

单独运用 function objects，用途并不大。Function object 通常只是小型的 class，用来执行单一而特定的行为。Function object 常常只有一个 member function operator()，没有任何 member variable。

Function object 之所以有用，因为它们使泛型算法变得更一般化。因为它们的存在，使得「执行策略 参数化」成为可能。一个简单的 find_if 函数，就可以经由任意种类的查找标准，查找任意对象型别所组成的任意线性序列。同样道理，排序算法 sort 不必考虑你所能够想到的任何「得以对某区间进行排序」的合理方法，而可以 function object 取代之（但后者必须是 Strict Weak Ordering 的一个 model），此 function object 用来定义该区间内的排列方式。

function objects，正如 STL 的大部分组件一样，作为「作用于线性区间上的算法」的附属品十分好用。

---

[8] 原本的 HP STL 称之为 times。为避免与 UNIX 函数库中的某个函数名称相冲，遂改名。

第 5 章

Containers
容器

第 2 章中我们看到了如何以 iterator 的观点来撰写泛型算法。Iterator 的抽象化过程让我们可以将算法与其所操作之元素分离（解除耦合关系，decouple），这使我们得以撰写「能操作于任意线性序列之上」的算法。

经过 2~4 章之后，现在我们面临一个重要的问题：那些元素从何而来？2~4 章的例子只涉及普通的 C 数组，但如果只是在数组上使用算法，我们所致力的一般性（generality）实在没有多大用处。

算法凡是作用于「某个 range 的元素身上」，其实就是作用于「某个数据结构的成员（members）身上」。算法作用于 range（区间）身上是很重要的，因为在很多情况下，数据结构可以被（全部或部分）视为某个线性 range。这意味如欲了解作用于 range 之上的泛型算法，我们还得看看其最后一个状况：数据结构（其内包含那些 ranges）。

5.1 一个简单的 Container

「可包含一个 range」的最简化数据结构，同时也是语言自身唯一支持的，就是数组。数组可以存放一系列元素，而且可以遍历那些元素。与数组相关的 iterator 就是指针。

C/C++ 数组有一些明显的优点，而且满足很多用途：

- 数组本质上遵循 range 概念（2.1.2 节）。如果 `A` 是具有 `N` 个元素的数组，则 `A+N` 是一个「past-the-end（越过尾端）」的指针，而 A 内的所有元素都包含在 range `[A, A+N)` 之中。
- 数组可以被分配在栈（stack）之内。大多数数据结构必须以动态方式分配内存（也就是使用 `malloc` 或 `new`），但数组不必。下面这样的声明：

```
int A[10];
```

并不涉及任何动态分配。这是一项优点，因为它比较快，而且不必检查是否分配成功。

- 数组很有效率，因为不必通过多个间接动作就可以访问某一元素。欲取用某个数组元素，只需要极少量机器码即可。
- 数组具有固定大小，而且在编译期就已经知道了。数组的操作不可能导致「重新改变大小」（那会带来一些额外开销），数组的使用者也不必对「数组大小可能有所改变」负责。
- 数组具有方便的初始化语法。你可以这样将数组初始化：

  ```
  int A[6] = {1, 4, 2, 8, 5, 7};
  ```

  但同样的语法不能用来对付任何稍为复杂些的 container classes。

C/C++ 数组也有一些明显的缺点：

- 尽管数组有固定大小，但应用程序必须明白记录数组大小，因为数组并没有 `size()` 这个 member function 可供运用。
- 每个数组都有「指向第一元素」的 iterator，以及「past-the-end（越过尾端）」的 iterator，但没有直接方法可以找到「越过尾端」的那个 iterator。你必须先找到指向开头的 iterator，再利用 iterator 的算术运算来获得数组结尾。
- 无法直接复制数组。数组没有 copy constructor 或 assignment 操作符。如果想要复制数组，必须自己写个循环来处理。
- 无法将数组以 by value 方式传入某个函数。事实上，根本很难将整个数组传入函数。每当你在表达式中取用某个数组时，你真正取用的其实只是「指向该数组第一个元素」的指针。数组是 C/C++ 型别系统的一部分，但数组型别通常「衰退」为指针型别。

5.1.1 一个 Array Class

尽管目前已有更精巧的 container classes，普通的 C 数组仍被广为采用。我们很容易就能定义出一个简单的 `block` class[9]，让它保留 C 数组的所有有用特点，并剔除主要缺点。

```
template <class T, size_t N>
struct block {
  typedef T                    value_type;
```

[9] `block` 这个名称是 Andrew Koenig 建议的。Stroustrup [Str97] 将类似（并不完全相同）的 class 命名为 `c_array`。

```
  typedef value_type*          pointer;
  typedef const value_type*    const_pointer;
  typedef value_type&          reference;
  typedef const value_type&    const_reference;

  typedef ptrdiff_t            difference_type;
  typedef size_t               size_type;

  typedef pointer              iterator;
  typedef const_pointer        const_iterator;

  iterator begin() { return data; }
  iterator end() { return data + N; }

  const_iterator begin() const { return data; }
  const_iterator end() const { return data + N; }

  reference operator[](size_type n) { return data[n]; }
  const_reference operator[](size_type n) const { return data[n]; }

  size_type size() const { return N; }

  T data[N];
};
```

正如 block 这个名称所暗示，它是一个由元素构成的简单而连续的区块。只比 C 数组的一层外包装（wrapper）再多一些东西。你可以这样声明一个容纳 10 个整数的 block：

```
block<int, 10> A;
```

而且可以普通的数组语法（例如 A[1]）取用某个元素。

为了让泛型算法更容易运用于 block，有一些额外机制需要考虑进去。首先，block 比 C 数组更明白支持 iterator 和 range。block 的所有元素都涵盖于 range [A.begin(), A.end()) 之中。其次，block 拥有一些相关型别（associated types），就像 iterator 一样。我们利用嵌套的 typedefs 来提供这些相关型别。

在 C 语言中，面对一个 T 对象构成的数组，你可以利用 T* 遍历之（如果需要改变数组内容的话），或利用 const T* 遍历之（如果不需改变数组内容）。由于 block 是数组的一层外覆物，所以我们能够轻易保存上述两者间的区别。是的，我们定义两个不同的嵌套型别：iterator 和 const_iterator。我们还必须定义两个不同版本的 begin() 与 end()。如果 A 是一个可修改的 block，则 [A.begin(), A.end()) 是一个由 ***mutable iterator*** 组成的区间；如果 A 是个不可修改的 const block，则 [A.begin(), A.end()) 就是一个由 ***constant iterator*** 组成的区间。同样道理，还得有两个不同版本的 operator[] member functions。如果 A 可修改，则 A[n] 返回可修改的 reference，如果 A 不可变，则 A[n] 返回 const reference。

因此，block class 遵循 3.2 节的典型模式，提供了一对 iterators，其中之一为 ***mutable***（可变的），

另一个为 ***constant***（不可变）；并提供一个由 mutable 至 constant 的转换方式。当然，此例之中的 iterators 只是我们所熟悉的 `T*` 和 `const T*`，不过为了一般化起见，我们还是在 `block` 中将它们定义为 `iterator` 和 `const_iterator`。

这是两个最重要的嵌套型别。一旦我们了解 `block` 的 iterator 是什么，任何与型别相关的事务几乎都会机械式地各就各位。是的，另有一些相关型别，尤其是与 `block`'s iterators 相关的型别。

由于 `block` 具备两种 iterators，而每个 iterator 具有 value type、difference type、reference type 与 pointer type，你或许会认为我们应该针对 `block` 定义出八种嵌套型别。这种推理几乎正确，但并不完全正确。第一，这八种型别之中某些是相同的，所以没必要重复定义它们。当两个 iterators 组成一对，它们必须具备相同的 value type 和 difference type（记住，`const T*` 的 value type 是 `T`，不是 `const T`）。第二，最后一个嵌套型别（size type）并非直接与 `block` 的 iterator 有关。正如你使用 difference type 来表示两个 iterators 的差距，你可以使用 size type 来对元素计数。difference type 一定有正负号，而 size type 无正负号。（difference type 几乎总是 `ptrdiff_t`，size type 几乎总是 `size_t`。）

以下说明 `block` 的所有嵌套型别：

1. **iterator types**：`iterator` 和 `const_iterator`。此例之中，iterators 就是指针。
2. **value type**：`value_type`。value type 是 `block` 内的对象型别。`iterator` 和 `const_iterator` 的 value type 是相同的。
3. **difference type**：`difference_type`。`iterator` 和 `const_iterator` 的 difference type 相同。
4. **size type**：`size_type`，用来表示 `block` 中的元素个数，或是某特定元素的索引值。任何型别为 `difference_type` 且非负值者，皆可以用来表示 `size_type`。两者的差别在于 `size_type` 用于计数，`difference_type` 用于减算（subtraction）。
5. **reference types**：`reference` 和 `const_reference`。`reference` 是「型别为 `value_type`」之对象的 reference，它和 `iterator` 的 reference type 同型。它同时也是 `operator[]` non-`const` 版之返回型别。`const_reference` 是「型别为 `value_type`」之对象的 `const` reference，并与 `const_iterator` 的 reference type 同型。它同时也是 `operator[] const` 版之返回型别。
6. **pointer types**：`pointer` 和 `const_pointer`。`pointer` 是「型别为 `value_type`」之对象的指针，它和 `iterator` 的 pointer type 同型。`const_pointer` 是「型别为 `value_type`」之对象的 `const` 指针，并与 `const_iterator` 的 pointer type 同型。

5.1.2 它是如何运作的

block class 的设计相当简单。它只有一个 member variable，就是 data 数组。但从某些方面来看，它是有点不太寻常。

第一，block 的元素个数是其型别的一部分。block<int,10> 与 block<int,12>是完全不同的型别。元素个数在编译期就固定不变了。你不能对 block 增加新元素，将 block<int,12> 赋值给 block<int,10>，或是以 block<int,12> 作为引数、传入预期获得 block<int,10> 的函数。block 的元素个数是该 block 的一个 template 引数，而非可以在 constructor 中指定的值。

这是所谓的「template 非型别参数（nontype template parameter）」，是语言自身的一个特性。是的，block 的第二个 template 参数并非一个型别，而是一个数值。C++ 语言当初加进 template 特性时，「template 非型别参数」就已经成为语言的一部分了，不过却很少被用到。然而它对于本例非常重要。在 block 内部，template 参数 N 被视为编译期常量，可用在编译期常量可运用的任何地方（此例之中，它被当作数组的元素个数）。这使 block 的 member variable data 不只是个指向数组的指针，而且可以代表数组自身。block 的所有元素都分配于一块连续存储空间，因此称为 "block"。

block 之所以不寻常的第二个理由是，它遗漏了很多东西：没有 constructor、没有 destructor、没有 assignment 操作符。这或许不那么令人惊讶，因为由编译器产生的每一个 member functions 的缺省版都不赖。将某个 block 赋值给另一 block，意味赋予其所有元素。较不寻常的是，block 甚至没有任何 public 声明或 private 声明。block 是个 struct，其所有成员，包含数组 data 自身，都是 public。

这似乎违反封装（encapsulation）概念，不过在这个特别的案例中并不会造成任何伤害。令 data 成为 public，可允许客户独立更动 data 内的元素，然而客户本就可以以其他方式更动之（例如通过 iterators 或使用 operator[]）。毕竟，这是整个 block 的要点。

令 data 成为 public，除了「不会造成任何伤害」之外，其实还有个特别而必要的理由。是的，在 C 语言中，你可以像对数组做初始化动作一般地（也就是在一对大括号内放置一串初值），将 struct 初始化。C++ 的情况也一样，不过得符合某种特殊限制：struct 不能拥有使用者声明的任何 constructor，也不能拥有任何 pirvate 或 protected members[10]。block 满足这些限制，所以我们可以像对数组一般地声明一个 block：

```
block<int, 6> A = {1, 4, 2, 8, 5, 7};
```

5.1.3 最后讨论

block 满足我们原先的目标。它是个 container class，可以像普通数组一般地被初始化，效率和数组一

[10] 满足这些限制的型别，在 C++ standard 中称为 aggregate type（聚合型别）。

样，并且仍属于「第一级」数据型别。你可以将某个 block 赋值给另一个 block、以 by value 方式传递、或是取用其起始点以及「past-the-end（越过尾端）」的 iterators。我们还可以对 block 再做一些加强，它们或许并未严格符合原先目标，但可以让 block 更方便用于某些场合。

第一点，同时也是最重要的一点，请回想我们于 2.2 节中定义的一些基础概念（basic concepts），它们被运用于其他概念身上，并且可以模塑出各式各样的型别。我们还定义了一个所谓的 ***regular type***（正规型别），表示满足所有这些概念的型别。

就以我们目前所定义的 block 而言，它已经是上述两个概念（分别为 Assignable 和 Default Constructible）的 model。然而它并不是 Equality Comparable 或 LessThan Comparable 的 model。意思是，如果 x 和 y 是两个型别为 block<T, N> 的对象，你不能够写 x == y 或 x < y。

没有道理遗漏这两个 concepts。operator== 和 operator< 两者都具有完全合理的定义。如果 blocks x 和 y 的元素一一相等，则 x 和 y 相等。也就是说如果 x 的每个元素等于 y 的对应元素，x 与 y 便相等。同样道理，我们可以指定以字母次序（或所谓字典排列方式）来判定 x 是否小于 y。如果 x[0] < y[0]，或是 x[0] == y[0] 而 x[1] < y[1]，或是……依此类推，那么 x 就小于 y[11]。

我们很容易以全局函数的方式实现 operator== 和 operator<:

```
template <class T, size_t N>
bool operator==(const block<T,N>& x, const block<T,N>& y)
{
  for (size_t n = 0; n < N; ++n)
    if (x.data[n] != y.data[n])
      return false;
    return true;
}

template <class T, size_t N>
bool operator<(const block<T,N>& x, const block<T,N>& y)
{
  for (size_t n = 0; n < N; ++n)
    if (x.data[n] < y.data[n])
      return true;
    else if (y.data[n] > x.data[n])
      return false;
    return false;
}
```

严格来讲，这两个函数还不足以保证 block<T,N> 是 Equality Comparable 和 LessThan Comparable 的

[11] 泛型算法 lexicographical_compare（p.225）便是定义这样的次序关系。

model。因为这两个 concepts 都要求能够将 opreator== 和 operator< 施行于型别 T 的对象身上。其真正的意思是，当且仅当 T 是 Equality Comparable 的一个 model，那么 block<T,N> 便是 Equality Comparable 的 model；同时，当且仅当 T 是 LessThan Comparable 的一个 model，那么 block<T,N> 便是 LessThan Comparable 的 model。或者，以比较率直的话来说，你可以以某个「不具有 operator==」的型别来具现化（instantiate）block — 前提是你永远不试图比较这样的两个 blocks。

我们能够做的第二个加强是，提供另一种 iterator。如果 A 是个 block，那么 range [A.begin(),A.end()) 包含 A 的所有元素，并以 A 的顺序来排列。我们在 3.2 节学到一种名为 reverse_iterator 的 iterator adapter（完整描述于 p.363），它会以相反方向（次序）来表现元素。reverse_iterator<Iter> 的行为和 Iter 一样，只不过前者的 operator++ 与后者的 operator-- 意义相同。

现在我们可以多定义两个 member function，rbegin() 和 rend()，返回上述的反向 iterators。那么，[A.rbegin(),A.rend()) 就像 [A.begin(),A.end)) 一样，包含 A 的所有元素，只不过前者是元素反向排列。当然，我们需要两个不同版本的 rbegin()，其原因就像我们需要两个不同版本的 begin() 一样。是的，如果 A 是 const，表达式 A.rbegin() 必须返回 ***constant iterator***，否则返回 ***mutable iterator***。

```
template <class T, size_t N>
struct block
{
  ...
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;

  reverse_iterator rbegin() { return reverse_iterator(end()); }
  reverse_iterator rend() { return reverse_iterator(begin()); }

  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
  }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
  }
};
```

还有一些 member functions 适合放入 block 中，虽然它们并非必要。其中包括：swap()，用来交换两个 blocks 的内容；empty()，当 block 不含任何元素时它会返回 true；max_size()，返回 block 可能容纳的最大元素个数。block 无法成长或缩减，所以 max_size() 并不是很有趣，它总是返回和 size() 一样的值。只有对于可变长度的 container，max_size() 才显得重要。不过我们可以在 block 中定义它，以期未来的泛化：

```
template <class T, size_t N>
struct block {
  ...
  size_type max_size() const { return N; }
  bool empty() const { return N == 0; }

  void swap(block& x) {
    for (size_t n = 0; n < N; ++n)
      std::swap(data[n], x.data[n]);
  }
};
```

终于，我们有了一个完整的 block 版本。虽然带着所有装饰成员（译注：意指并非绝对必要的成员，如 swap()、empty() 和 max_size()）的它，比起精巧而复杂的 STL 数据结构如 deque 和 map 等，显得十分简易，不过它还是很有用的。像 block 这样简易的 class，往往是实用上的最佳选择。

```
template <class T, size_t N>
struct block {
  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 ptrdiff_t               difference_type;
  typedef size_t                  size_type;

  typedef pointer                 iterator;
  typedef const_pointer           const_iterator;

  iterator begin() { return data; }
  iterator end() { return data + N; }

  const_iterator begin() const { return data; }
  const_iterator end() const { return data + N; }

  typedef reverse_iterator<const_iterator>      const_reverse_iterator;
  typedef reverse_iterator<iterator>            reverse_iterator;

  reverse_iterator rbegin() { return reverse_iterator(end()); }
  reverse_iterator rend() { return reverse_iterator(begin()); }

  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
  }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
  }
```

```
  reference operator[](size_type n) { return data[n]; }
  const_reference operator[](size_type n) const { return data[n]; }

  size_type size() const { return N; }
  size_type max_size() const { return N; }
  bool empty() const { return N == 0; }

  void swap(block& x) {
    for (size_t n = 0; n < N; ++n)
      std::swap(data[n], x.data[n]);
  }

  T data[N];
};

template <class T, size_t N>
bool operator==(const block<T,N>&x, const block<T,N>& y)
{
  for (size_t n = 0; n < N; ++n)
    if (x.data[n] != y.data[n])
      return false;
    return true;
}

template <class T, size_t N>
bool operator<(const block<T,N>&x, const block<T,N>& y)
{
  for (size_t n = 0; n < N; ++n)
    if (x.data[n] < y.data[n])
      return true;
    else if (y.data[n] > x.data[n])
      return false;
    return false;
}
```

5.2 Container Concepts

下一个步骤应该很明显。我们已经定义了一个有用的 class 名为 block，而且我们可以合理地想像 block 是某个一般化 concept 的一个 model。下一个步骤就是要定义出这个 concept。或者以不同的角度来看，我们想像要定义其他「类似 block」的 classes；然而它们应该多么类似 block 呢？当我们定义 block 时，我们并没有区别哪些功能是所有 container class 普遍应有，哪些功能是特别设计给 block 用的。现在我们必须区别这两种类型。

block 具有三个主要的功能性：(1) 它包含元素；(2) 它提供访问那些元素的方法；(3) 它支持「让 block 成为 ***regular type*** 」所必须的一些操作行为。

5.2.1 元素的容纳（Containment of Elements）

「container 可以容纳元素」这句话似乎多余，但其中却蕴义深远。两个 blocks 不能重叠，而且一个元素不能同时分属两个 blocks。每个元素都是包含于 block 内的一个子对象（subobject），会在 block 构造时被构造，并在 block 被摧毁时亦被摧毁。block 元素的生命期与 block 自身的生命期一样。你可以更改某个元素的值，但是不能产生新的元素，也不能单单摧毁某元素却不摧毁整个 block。

单就字面意义来说，block 的元素是该 block 的一部分。block 是单一的连续内存区域，每一个元素的地址都位于这段区域内。就字面意义来说，所谓「容纳（containment）」和 block 的特定实现细节有很大关系（我们不希望把 container 定义得太过狭隘，以至于最终必须将任何运用了动态内存分配行为者排除于外），不过我们可以将其中最重要的特性抽象化。

1. 两个 containers 不能重叠，同一个元素不能分属一个以上的 containers。当然，放两个不同的副本于两个不同的 container 中是没有问题的。换句话说前述的限制是针对对象，而不是针对其值。无论如何，拥有权不可共享。

2. 元素的生命期不能超越其所属 container 的生命期。元素不能在其 container 构造之前先产生出来，也不能在其 container 被摧毁之后才被摧毁。

3. container 可以如同 block 一般有固定的大小。但其大小也可以改变，以便你能够在产生 container 之后才产生或摧毁某些元素。即使是可变大小的 container，也拥有其元素，这些元素会被 container 的 destructor 摧毁。

另一种说法是：所有 containers，如同 C 数组或 block 一样，具有数值语义（*value semantics*）而非指针语义（*pointer semantics*）。Container 的元素实际上是对象，并非仅只地址而已。这似乎是个严苛的限制，其实不然。毕竟指针也是一种对象，就像其他对象一样，也可以存储在 container 之中。举例来说，你可以拥有一个 block<char*, 10>；这个 block 仍具有数值语义（因为每个 char* 都是存储于 block 内独一无二的对象），但是它只能「拥有」指针自身，而非指针所指之物。我们曾经在第 1 章看过一个例子，在那里我们产生了一个 vector，其中都是一些指针，指向一个字符串表格。

5.2.2 Iterators

有三种不同的方式可以访问 block 的元素：

1. block class 定义了嵌套型别 iterator 和 const_iterator，而 block A 的所有元素都包含于 range [A.begin(),A.end()) 之中。这种形式对泛型算法格外有用，因为几乎所有 STL 泛型算法都可以作用于 iterators 所形成的 range 上。

2. 相对于 iterator 和 const_iterator，嵌套型别 reverse_iterator 和 const_reverse_iterator 是反向的 iterator 型别。range [A.rbegin(),A.rend()) 所包含的元素与 [A.begin(),A.end()) 相同，但排列次序相反。

3. 如果 n 是一个整数，表达式 A[n] 返回第 n 个元素。

这三种方式中，第一种显然最为基本。第二种使用 adapter；嵌套型别 reverse_iterator 和 const_reverse_iterator 其实只是以下式子的速记法：

```
reverse_iterator<iterator>

reverse_iterator<const_iterator>
```

同样道理，表达式 A[n] 只是 A.begin()[n] 的速记法，或说是 *(A.begin() + n) 的速记法。

block 所提供的元素访问方式中，真正重要的是，它定义了 iterator types iterator 和 const_iterator，以及 member function begin() 和 end()。Iterators 之于 container 的重要性，正如 iterators 之于算法的重要性一样。它们是算法与容器（containers）间的主要接口。

所有 containers 都具有 iterators。block iterators 的某些特性是一般化的，某些特性则专属于 block：

- block 有两种不同的 iterator types：iterator 和 const_iterator。它们之间的差别是：***mutable block*** 提供的是 iterators 所形成的 range，***const block*** 则提供 const_iterators 所形成的 range。所谓 const block，就是其所有元素都不可变动，所以 const_iterator 是一个 constant iterator，不允许你对其所指的值做赋值动作。至于 iterator 则是一个 mutable iterator，允许你对其所指的值做赋值动作。

  事实上，不只 block，所有 containers 都定义有 iterator 和 const_iterator 两型别，以及 member function begin() 与 end()。如其名称所示，型别 const_iterator 一定是个 constant iterator，而且某个 container 的 iterator 一定可以转换为该 container 的 const_iterator。

  如果说 const_iterator 一定是个 constant iterator 的话，很自然地我们会假设 iterator 一定是个 mutable iterator，就像 block 之中一样，而且我们一定能够更改 iterator 所指的值。但这样的假设是错误的。Container 不一定得提供 mutable iterator，而 iterator 自身就可能提供「读写」或「只读」访问方式。有时候，为了维护 class 的不可变动性，有必要限制或禁止对 class 的「写入」动作。这并不只是个假设而已。举例来说，associative container set（p.461）就没有供应 mutable iterators。对 set 而言，iterator 和 const_iterator 可以完全一样。

- 型别 iterator 和 const_iterator 并不仅仅是 iterators 而已。它们是 Random Access Iterator（这表示它们同时也是 Bidirectional Iterator 和 Forward Iterator，因为 Random Access Iterator 是 Bidirectional Iterator 的一个强化，而 Bidirectional Iterator 又是 Forward Iterator 的一个强化）。

「block 的 iterator 是 Random Access Iterator」这件事之所以重要，有两个原因。第一，它影响可施行于 block 元素身上的算法种类：Random Access Iterator 比其他种类的 iterator 支持更多算法。举例来说，你可以将 sort（p.292）施行于 Random Access Iterator 所形成的 range 上，但决不能施行于 Forward Iterator 所形成的 range 上。第二，它会直接影响到 block 所提供的 member functions。

[A.begin(),A.end()) 合理的原因是，block 的 iterators 是一种 Input Iterator（Output Iterator 并不一定支持 range），而 [A.rbegin(),A.rend()) 合理的原因是，block 的 iterators 是一种 Bidirectional Iterator。因为只有在 iterator 支持 operator-- 的情况下，才能在该种 iterators 所形成的 range 中反向移动。最后一点，以 A[n] 来访问 block 元素，就是「随机访问」的意思；由于 block 拥有 Random Access Iterator，因此以这种形式访问元素是合理的。

我们有理由要求所有 containers 的 iterator 型别都至少是 Input Iterator 的一个 model，因为一个无法审视其元素的 container，没什么多大用处。但我们没有理由要求所有 containers 的 iterator 型别都得属于 Random Access Iterator，因为有太多单纯的数据结构，并不适合以随机方式访问元素。

5.2.3 Containers 的阶层架构（Hierarchy）

有些 containers，例如 block，可以并且应该提供 Random Access Iterator。其他 containers 则只需提供 Bindirectional Iterator 或 Forward Iterator 就好。这暗示我们 containers 并不是单一的 concept，而是数个不同的 concepts。我们可以根据 containers 所提供的 iterator 种类，来对 container 分类。

任何一个隶属于 Container 的型别，都定义有嵌套型别 iterator 和 const_iterator，两者都属于 Input Iterator。这两个型别可能相同。任何情况下，const_iterator 一定是个 constant iterator，而且我们一定可以将 iterator 转换为 const_iterator（但除非两型别一致，否则不能做逆向转换。是的，将 constant iterator 转换为 mutable iterator，会严重违反 const 的正确性）。每一个 Container 都拥有 member function begin() 和 end()，而其所有元素都包含于 range [begin(),end()) 之中。

每一个 Container 也拥有一些辅助型别：value_type、pointer_type、const_pointer_type、reference_type、const_reference_type、different_type 和 size_type。它们的定义只是为了方便，并非因为必要。除了 size_type，它们都可以由 iterator 和 const_iterator 推导出来。

所谓 Forward Container，表示其 iterators 隶属于 Forward Iterator；所谓 Reversible Container，是一种 Forward Container，而其 iterators 隶属于 Bidirectional Iterator。Reversible Container 的 model 型别定义了两个额外的嵌套型别：reverse_iterator 和 const_reverse_iterator，以及两个额外的 member functions：rbegin() 和 rend()。所谓 Random Access Container 是一种 Reversible Container，

而其 iterators 隶属于 Random Access Iterator。Random Access Container 的 model 型别拥有 member function `operator[]`。图 5.1 显示了这样的阶层架构。

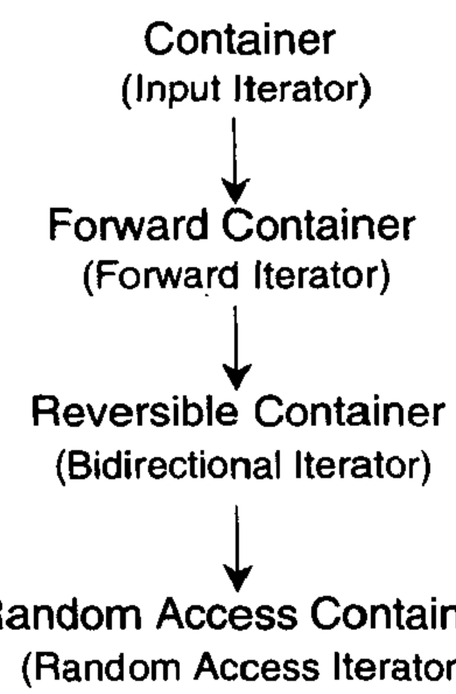

图 5.1 Container concepts 的阶层架构。图中每一个 container concept 之下所列的名称，是该 container 的 iterator 型别

STL 预定义的所有 container classes 都是一种 Forward Container，其中大多数甚至是 Bidirectional Container。有一些像 `block` 之类的 container classes，则属于 Random Access Container。

5.2.4 最平淡无奇的 Container

5.1 节的 `block` class 是一个很简单的 container，但它还不是最简单的 STL container。你知道，最简单的 C++ 程序是一个不做任何事的程序：

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

同样道理，最简单的 STL container 是一个实际上不包含任何东西的 container。

每个 container 都拥有 member functions `begin()` 和 `end()`，它们都返回 iterator。Range `[begin(),end())` 涵盖这个 container 的所有元素。所谓「最平淡无奇的 container」，其 range 永远为空，其大小永远为零：

```
template<class T>
struct trivial_container {
  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 value_type*          iterator;
  typedef const value_type*    const_iterator;
```

```
  typedef ptrdiff_t              difference_type;
  typedef size_t                 size_type;

  const_iterator begin() const { return 0; }
  const_iterator end() const { return 0; }

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

  size_type size() const { return 0; }
  bool empty() const { return true; }
  size_type max_size() const { return 0; }

  void swap(trivial_container&) {}
};
```

这个平淡无奇的 container 是出于好奇心才撰写的，正好展示了所有你必须满足的 Container 条件。不过它也有两个真正的用途：

- 它满足 Container 的所有条件，所以你可以利用它来测试「以 container 作为引数」的泛型算法。
- 如同你在 `trivial_container` class 定义式中所看到的，STL container 涉及大量「照本宣科的行为」。我们很容易遗忘某些实现细节，因此，当你要撰写一个新的 STL container 时，你会发现，以 `trivial_container` 为基础，会比从无到有轻松许多。

5.3 大小可变的 Container Concepts

`block` class 是一个大小固定的 container。`block` 的元素个数是你在撰写程序时给定的常量。这既非优点也不是缺点；这只是个事实。大小固定的 container 有时合用，有时却不合用。是的，在你开始装填之前，你无法知道一个 container 最后会有多少元素。

Container concept（以及 Forward Container、Reversible Container、Random Access Container）允许固定大小和可变大小两种 containers，但是不提供任何新增或删除 container 元素的机制（如果它提供这样的机制，那么像 `block` 这样大小固定的 container 便不能成为 Container 的一个 model）。当然，这意思是说，可变大小的 containers 必须是其他某些 concepts 的 models，而那些 concepts 又都是 Container 的强化。

STL 定义两种大小可变的 containers：Sequence Container[12] 和 Associative Container。

---

[12] Sequence 这个名称有时会造成困扰。在计算机科学或其他领域中，sequence 字义原本是指某些事物的集合，这些事物以某种明确的顺序排列。然而，STL 却使用这个听起来很普遍的字眼当作某种特定 concept 的名称。出现于本书之中的 Sequence，以 concept 的意义而言，蕴含有某些特别的技术意义。

5.3.1 Sequences（序列）

Sequence 是 Forward Container 的一个强化（refinement），是一种最明显的大小可变的 container。就像所有 containers 一样，Sequence 以严格线性顺序的 range 来呈现其元素。此外，你不但可以取用任何元素，也可以在 range 的任意一个地点新增或删除元素。也就是说，Sequence 不会以某种规定来排列元素，它让你依照任何你所希望的顺序进行排列。

Sequence 都必须具有 member functions `insert` 与 `erase`。如果 `s` 是 Sequence 而 `p` 是「指向 `s` 内某个元素」的 iterator，则 `s.erase(p)` 会移除并摧毁该元素。类似情况，`s.insert(p, x)` 会产生一个新对象，此对象是 `x` 的一份副本，然后被安插于 `p` 所指元素之前。为什么在 `p` 之前而非 `p` 之后呢？再一次提醒你，因为 iterator 的 range 是非对称性的。只有所谓「past-the-end（越过尾端）」的元素，没有所谓「before-the-beginning（开头之前）」的元素。你可以在「`s.begin()` 之前」做安插动作，也可以在「`s.end()` 之前」做安插动作，前者将在最前头处安插一个新元素，后者则在最尾端处安插一个新元素。如果 `s.insert(p,x)` 是将 `x` 安插于 `p` 之后而非之前的话，那么上述行为便无法达成了。

尽管这两个 member functions 非常简单，它们却引发两个重要的问题：

- 当你安插或删除元素时，对 Sequence 的其他元素带来什么影响？当然，我们应该预期其他数值不被干扰。如果一开始有数值 (1,2,3)，然后安插 7 于第二个元素之前，结果应为 (1,7,2,3)。否则此等操作行为就不应该获得 `insert` 之美名。然而，「数值」并非唯一的可能。

  要知道，Container 的元素不单只是数值而已：它是一个对象，实际存在于某内存地址上。尽管安插或删除某元素并不会改变 Sequence 中的其他任何元素，但却可能会影响 iterator 所指的那个对象。

  更具体化的说法就是：当我们安插或删除元素，指向 Sequence 的那个 iterator 发生了什么事？举例来说，假设 S 是一个 Sequence 而 `p1` 和 `p2` 指向 `s` 之中两个不同元素的 iterators。现在假设我们安插新元素于 `p1` 之前，或移除 `p1` 所指的元素，对 `p2` 会有什么影响？它仍然指向先前所指的那个元素吗？它真的指向任何东西吗？抑或不论先前指向什么，其所指东西如今已经消失无踪？

- `insert` 与 `erase` 的复杂度如何？它们是简单而快速的操作行为，或者潜在地需要付出昂贵代价呢？举例来说，假设你要在一个拥有上千个元素的 Sequence 身上安插或删除某元素，你预期需要多久时间完成？你是否应该预期这比在拥有百万元素的 Sequence 身上安插或删除元素慢一千倍？

这两个问题的答案相同：视情况而定。面对不同的 Sequence，这些问题有不同的答案。STL 包含三种 Sequence class：vector、list 和 deque，其中主要差异在于它们所提供的 iterator 种类、iterator 的无效语义（译注：意指什么样的操作行为会造成先前取得的 iterator 无效）以及 insert 与 erase 的复杂度。

以 vector 为例，它具有 Random Access Iterator 性质，并将元素存储于单一连续的内存局部内。vector 类似 block，但其大小可变。是否你要安插新元素于 vector 的中间，你必须先将其他元素往尾端移动，以腾出一个空位。这意味「安插」对 vector 而言是十分缓慢的操作行为（其速度与「插入点至尾端的元素个数」成线性关系），而且「安插动作」会使指向这个 vector 的其他 iterators 无效。相较之下，list 是以节点（node）为基础的 container，其 iterators 属于 Bidirectional Iterator。安插新元素到 list 之中并不涉及任何既有元素的搬移，也不会造成任何 iterators 无效。

其他形式的 insert 与 erase

Sequence 的 insert 与 erase 都是重载（*overloaded*）后的 member functions。除了「单元素版本」之外，还有其他版本可以一次安插或删除整个 range。如果你真的必须安插或删除整个 range，使用「多元素版本」会比多次调用「单元素版本」来得快速许多（至少绝对不会比较慢）。举个例子，如果 v 是个 vector，L 是个 list，你可以将 L 的所有元素安插于 v 的开头，动作如下：

```
V.insert(V.begin(), L.begin(), L.end());
```

Sequence 的 constructor 亦具有「多元素版本」。

安插（Insertion）于开头（Front）和尾端（Back）

很多程序并不需要在 Sequence 的任意位置安插或删除元素，只可能会在最开头或最尾端安插或删除元素（例如：从某个缓冲区读进元素）。「安插于开头或尾端」是个很重要的特例。

它之所以是个特例，也因为安插元素于「开头或尾端」常会比安插于「任意位置」快许多。例如，安插元素于 vector 的起头处，复杂度是 $O(N)$，其中 N 是 vector 的大小，但安插元素于尾端则是个快速行为。这导出两个 concepts，都是 Sequence 的强化（refinements）。一个是 Back Insertion Sequence，就像 vector 那样，能够在「分期摊还的常量时间（amortized constant time）」内于尾端安插或删除元素。另一个是 Front Insert Sequence，能够在常量时间（constant time）内于开头安插或删除元素。（译注：amortized time 是一种复杂度分析方式，详见算法专书。）

Front Insert Sequence 具有三个特别的 member functions：(1) front，返回 container 中的第一个元素；(2) push_front，在开头安插新元素；(3) pop_front，删除第一个元素。Back Insertion Sequence 具有三个对应的 member functions：back、push_back 和 pop_back。

安插（Insertion）语义和覆盖（Overwrite）语义

当你初次学习使用 STL，最容易犯的错误便是类似以下的写法：

```
int A[5] = {1, 2, 3, 4, 5};
vector<int> V;
copy(A, A+5, V.begin());
```

这段代码看起来相当自然 — 复制五个元素到 vector 中 — 但其实完全错误。如果你试着执行它，程序可能会完蛋。

好好想一想，很容易便能发现问题所在。copy 算法会从某个 range 复制元素到另一个 range。既然如此，它会复制 range [A,A+5) 到 range [V.begin(),V.begin()+5)。这意味它会执行赋值动作 *(V.begin()) = A[0], *(V.begin()+1) = A[1] .....但这么做不可能成功，因为 vector V 是空的，我们的赋值目标并不存在。

让我们换个角度来看这个程序。copy 没办法将新元素安插于 V 之中，因为 copy 永远看不到 V 的整个本体，它只看到 iterator。也就是说，复制到一个「由 V's iterators 所形成的 range」，其实是一种「覆盖（*overwrite*）」语义。它会将新值赋于 V 的既有元素上，而非安插新的元素。要对 V 安插新元素，你必须调用 V 的 member function（译注：而不是像现在这样调用泛型算法 copy）。

事实上，有办法让你以 copy 算法在 Sequence 中安插新元素：利用所谓的 insert_iterator adapter（p.351），或其他相关的 adapters：front_insert_iterator、back_insert_iterator。insert_iterator 系由一个特定的 Sequence 构造出来，它是个 Output Iterator；通过 insert_iterator 来赋值，等价于将该值安插于 Sequence 之中。

这些 adapters 使你得以通过 iterators 执行安插（*insertion*）语义而非覆盖（*overwrite*）语义。下面是以 copy 将新值安插到 vector 的正确写法：

```
int A[5] = {1, 2, 3, 4, 5};
vector<int> V;
copy(A, A + 5, back_inserter(V));
// 译注：back_inserter() 是一种 insert iterator adapter function,
// 用来产生某个 container 的 back_insert_iterator。
```

5.3.2 Associative Containers（关联式容器）

Sequence 并不强求特定顺序：当你安插元素于 Sequence 之中，你可以指定安插于任意位置。另有一种可变大小的 container，保证其元素总是会根据特定规则来排列。使用这种 container，就不需将元素「安插于某特定位置」，只要「安插」就好了，container 自动会将新元素安排在适当的位置。

什么原因需要这种自行安排顺序的 container 呢？最明显的原因是，它让你得以快速查询某个元素。如果你要在一个拥有 N 个元素的有序（ordered）range 中查询某元素，最好的选择便是使用线性查找：

find 或 find_if，复杂度是 *O(N)*。平均而言，找到目标之前必须先审视过 range 中的一半元素。如果这些元素是以某种方便查找的数据结构组成，复杂度便可从 *O(N)* 降至 *O(logN)*，甚至更好。举个例子，如果这些元素以升幂排列，你可以用二分查找法（lower_bound）取代线性查找。

Associative Container 是一种可变大小的 container，可以高效率撷取「以 key 为基础」的元素。Associative Container 的每个元素都有一个相关联的 key。只要给定一个 key，Associative Container 便有办法快速找寻具有该 key 之元素。

Associative Container 的基本操作行为是查询（lookup）、安插（insertion）和消除（erasure）。你也可以在 Sequence 中执行这些行为，差别在于：(1) Associative Container 提供明白支持这些行为的 member functions；(2) 那些操作行为都具有良好的效率，平均是 *O(log N)*。

Associative Container 比 Sequence 更为复杂。有很多东西在不同的 Associative Container 中互不相同。

- Associative Container 的每个元素都拥有 key，元素系根据该 key 被查询。和所有 container 一样，Associative Container 也有 value type。此外它更多了一个 associated type，亦即其 key type。每一个型别为 value_type 的值，都和某个型别为 key_type 之值相互关联。

  values 和 key 之间存在许多种关联方式。例如 value type 可以是拥有很多字段的记录，而 key 是其中一个字段。STL 定义了两种 Associative Container concepts，表现出两种「value 与 key 的关联方式」。一种是 Simple Associative Container，其 value_type 和 key_type 相同；元素的 key 就是元素自身。另一种是 Pair Associative Container，其 value_type 的形式是 pair<const Key, T>。Pair Associative Container 的 value 是个 pair，而其 key 则是这个 pair 的第一个字段。

  为什么是 pair<const Key, T> 而不是 <Key, T> 呢？这和「不得将新元素安插于 Associative Container 的任意位置」是相同的道理。是的，元素在 Associative Container 内的位置，系取决于其 key，所以更改元素的 key 会造成 Associative Container 前后不一致。在 Pair Associative Container 中，元素的 key 是第一个字段，而这第一个字段必须是不可更改的。在 Simple Associative Container 中，元素的 key 就是元素自身，所有元素的值都不能更改。是的，元素可以被安插以及被删除，但是不能被更改。Simple Associative Container 只提供 constant iterators，不提供 mutable iterators。

- 当你将某个元素安插到 Associative Container 中，而该 container 彼时已有同 key 的元素，会发生什么事？到底 Associative Container 可不可以拥有两个同 key 的不同元素呢？

  如果是 Multiple Associative Container 就可以，如果是 Unique Associative Container 就不行。是的，你可以毫无限制地将元素安插于 Multiple Associative Container 之中，后者并不会检查是否有同 key

的元素已经存在。但是当你将元素安插于 Unique Associative Container 时，如果后者已经拥有同 key 的元素，安插动作便不会生效。

- 每个 Associative Container 内的元素都是根据 key 来排列。Associative Container 至少有两个不同的强化概念，分别以不同方式来组织元素。其一是 Hashed Associative Container，根据 hash table（散列表）来组织元素；它有一个嵌套型别，是个 function object，作为 hash function（散列函数）之用。其二是 Sorted Associative Container，它的元素一定以 key 的升幂顺序排列；也具有一个 function object 嵌套型别 — Binary Predicate，用来决定某个 key 是否小于另一个 key。

以上三种差异具有正交性（orthogonal）。以 STL 的 container class `set` 为例，它隶属于 Sorted Associative Container、Unique Associative Container 和 Simple Associative Container。也就是说，`set` 的所有元素会形成一个「升幂顺序、元素之 key 即为该元素、不会有两个相同元素」的 range。至于 STL container class `multimap`，则隶属于 Sorted Associative Container、Multiple Associative Container 和 Pair Associative Container。

和 Sequence 一样，Associative Container 必须定义 member function `insert` 和 `erase`。再次如同 Sequence，我们可以在 Associative Container 内一次安插（或删除）一个元素或整个 range。真正的差别在于：Sequence 的 member function `insert` 需要一个引数，指定安插位置，而 Associative Container 并不需要。如果想要安插单个元素于 Associative Container `C`，可以这样写：

```
C.insert(x);
```

如果要安插某个 range，可以这样写：

```
C.insert(f, l);
```

让我举一个简单的例子。假设 `L` 是个 `list<int>`，你可以用单一动作，将 `L` 的所有元素安插于一个 `set` 内：

```
set<int> S;
S.insert(L.begin(), L.end());
```

于是 `set S` 会内含 `L` 的所有元素，并以升幂顺序排列，而且移除重复元素。

Associative Container 同样具有「以 key 查找元素」的相关 member functions。例如以下表达式：

```
C.find(k)
```

返回一个 iterator，指向「key 为 `k`」之元素。如果返回 `C.last()`，表示这样的元素不存在。以下表达式：

```
C.count(k)
```

返回「key 为 `k`」之元素个数。如果询问对象是 Unique Associative Container，返回值必为 0 或 1。

5.3.3 Allocators（配置器）

很多 containers 以动态方式分配内存（但非全部如此，例如 block 就不是）。以 list 为例，它是节点的连接链表（linked list），每个节点都靠动态分配获得。

如果能够控制 container 的内存分配方式，有时候很有用。举例来说，你可能希望在一个「thread-safe 分配法」与一个「快速，但只于 single-threaded 程序中安全」的分配法之间做选择。或者你希望使用某些事先分配好的内存，取代自由分配的内存。

这类一般性问题的一个实例是，让 client 控制 class 某些方面的行为。我们已经看过解决之道了：我们可以将这些行为封装成一个 template 参数。这很类似算法（如 find_if 和 sort）所采用的做法，也很类似 container classes（如 set）采用 function object 的做法。

这次我们不能使用 function object，因为内存分配涉及数个不同的操作行为（至少涉及分配函数与归还函数，而这两者必须具有一致性，换句话说你不能以 new 分配内存却以 free 归还之）。然而概念还是一样的：container 的内存管理策略由一个 template 参数定义之，这个参数就是 Allocator（p.166），用来处理所有的内存分配与归还行动。

Allocator 并非 Containre 需求条件的一部分（当然也就不是 Sequence 或 Associative Containre 需求条件的一部分）。然而所有预定义的 STL container classes，都允许以 template 参数指定其 allocator。

5.4 总结

整个 containers 阶层架构颇为复杂，如图 5.2 所示。你可能好奇是否有必要如此复杂，是否真的可以因为「定义出这样的 concepts 阶层架构」或「定义出隶属那些 concepts 的各个具体 container classes」而获得些什么。你的这种好奇情有可原。

定义这个阶层架构让我明了两件事情：第一，它提供 containers 的分类法。例如它使我们更容易了解，为什么 vector 与 list 的接口是那么相像，但却不完全相同的原因。第二，它让你得以撰写能够直接作用于 containers 自身的泛型算法。

5.4.1 我们应该使用什么样的 Container？

由于所有 STL containers 都是同一个 concept 下的一些 models，所以它们提供很相像的功能。你可能会发现，有些 container classes 虽不相同，在你的应用上却都是合理的选择。你甚至可能发现，只要在程序中更改一行（例如 typedef），就能够让程序使用不同的 container class。

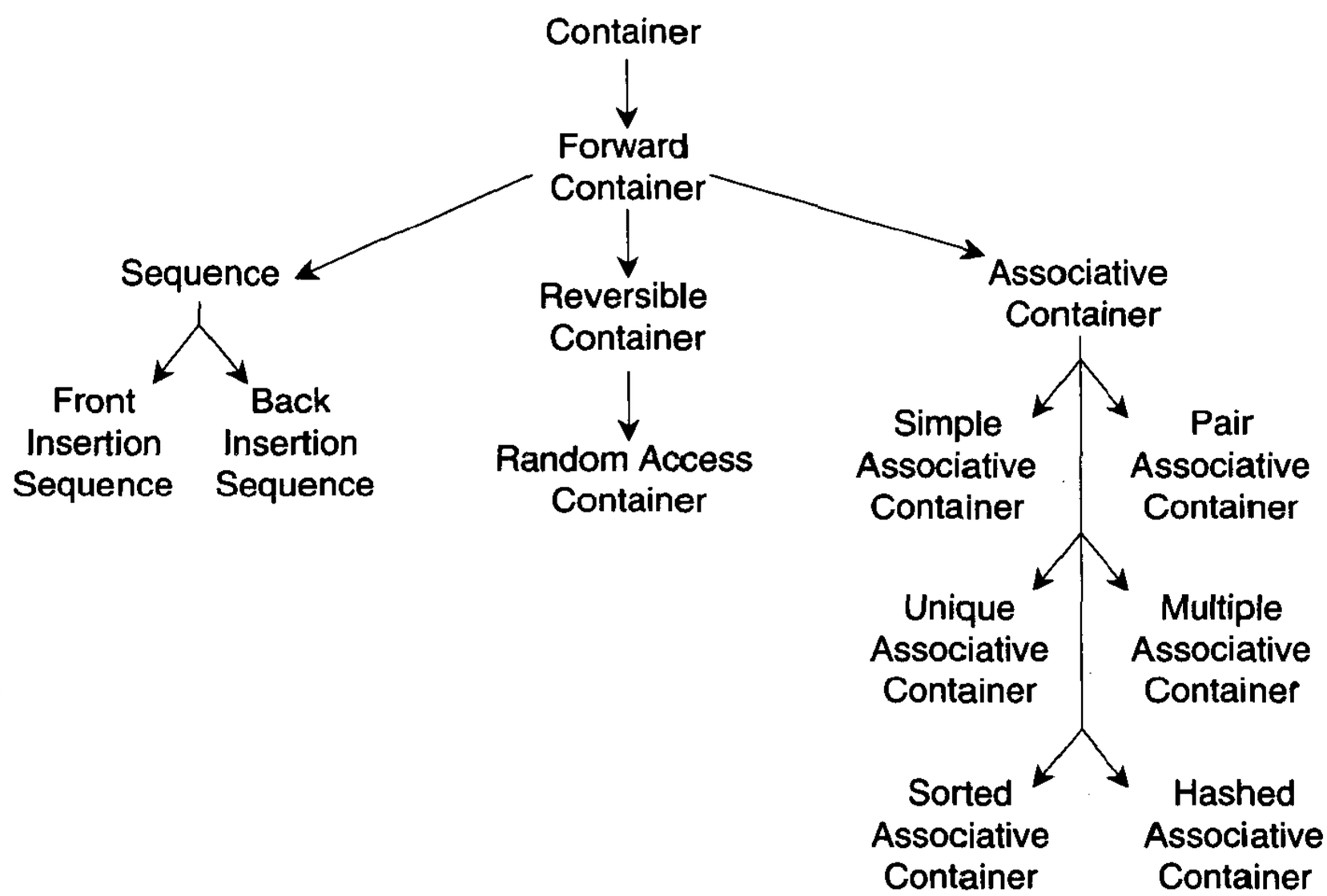

图 5.2 Container concept 的阶层架构，其中包含 Sequence 和 Associative Container

一般的法则是，尽可能使用最简单的 container class。如果你不想在 container 内频繁地做安插动作，`vector` 可能是最有效率的选择。如果你永远不会更动 container 的大小 — 也许你在编译期就知道大小 — 那么甚至可以使用 `block`。然而如果你预期会频繁地执行安插动作，就应该使用诸如 `list` 或 `slist` 之类的 containers，它们是专为「将元素安插于 container 之内」而设计的。

5.4.2 设计你自己的 Container

有时候 STL 预定义好的 container classes 不够用，你必须撰写你自己的 containers。如同我们先前所见的 `block` 一样，container 不必定义得很花俏。是的，Container concept 的需求条件并不多。然而可变大小的 container class，包括 Sequence 和 Associative Container，就稍微有点复杂了。如果你计划撰写一个新的 Sequence 或 Associative Container，你应该先确定你已经了解第 9 章所描述的具体需求。

第二篇

参考手册：
STL Concepts

Reference Manual : STL Concepts

第 6 章

基本概念

Basic Concepts

本章所谈的 concepts 都是非常一般性的概念。它们适用于任何泛型程序库，而非只适用于 STL。大多数 C++ 内建型别以及 STL 定义的许多态别都是这些 concepts 的 models。

所谓正规型别（*regular type*），是同时兼具 Assignable、Default Constructible 和 Equality Comparable 的一个 model。除非有合理的因素，否则任何使用者自定型别（user-defined type）都应该是正规型别。

6.1 Assignable

如果我们能够复制某个型别的 object，并且可以将数值赋予该型别的变量身上，这个型别便是 Assignable。

Object 的复制体必须等价（equivalent）于原来的 object。它们必须具有相同的值。你或许觉得这句话意味「相等（equality）是赋值（assignment）的必然结果」，这几乎正确，却不尽然。问题在于，一个身为 Assignable model 的型别，没有必要同时也是 Equality Comparable 的 model；因为它可以不需要任何 `operator==`。

如果某个型别同时是 Assignable 和 Equality Comparable 的 model，那么 `x` 的复制品和 `x` 自身应该是相等的。

➢ Refinement Of（强化自）

这个 concept 不是其他任何 STL concepts 的强化（refinement）。

➢ Notation（符号）

`X`	某型别，它是 Assignable 的一个 model。
`x, y`	型别 `X` 的 objects。

➢ Valid Expressions（有效表达式）

● Copy constructor

```
X(x)
```

返回型别： X
必然结果： X(x) 是 x 的复制品

● Copy constructor

```
X x(y);
X x = y
```

必然结果： x 是 y 的复制品

● Assignment

```
x = y
```

返回型别： X&
必然结果： x 是 y 的复制品

➢ Models

几乎所有 C++ 内建型别都是 Assignable model，值得注意的是，const T 是个例外。举个例子，int 是个 Assignable model（所以我们能够复制和赋值型别为 int 的值），但 constint 则不然。如果 x 声明为 const int，那么 x = 7 是不合法的。换句话说你可以复制 const int 变量，但你不能对它赋值。

同样地，型别 pair<const int, int> 也不属于 Assignable。

6.2 Default Constructible

如果某型别具有一个 default constructor，也就是说我们不需指定任何初值，就可以构造该型别的 object，那么这个型别便属于 Default Constructible。

➢ Refinement Of（强化自）

这个 concept 不是其他任何 STL concepts 的强化（refinement）。

➢ Notation（符号）

X	某型别，它是 Default Constructible 的一个 model。
x, y	型别 X 的 objects。

➢ Valid Expressions（有效表达式）

● Default constructor

```
X()
```

返回型别： X

● Default constructor

```
X x;
```

注意，以下形式：

```
X x = X();
```

不一定是有效表达式，因为它用到了一个 copy constructor。这种形式对于所有正规型别是有效的，因为正规型别同时是 Default Constructible 和 Assignable 两者的 model，但对于单单只是 Default Constructible model 之型别而言，这种形式并非有效表达式。

➢ Models

几乎所有 C++ 内建型别都是 Default Constructor 的一个 model。举例来说，你可以声明一个型别为 int 的变量，但不必为它指定初值。许多 STL classes 如 vector<int>，同样也属于 Default Constructible。

6.3 Equality Comparable

如果我们可以使用 operator== 对某个型别做相等性比较，而且 operator== 系表示等价关系（equivalence relation）的话，该型别便是 Equality Comparable。稍后我会对「等价关系」加以定义，它们是一般直觉的「相等性（equality）」的正规形式。

➢ Refinement Of（强化自）

这个 concept 不是任何其他 STL concepts 的强化（refinement）。

➢ Notation（符号）

x 某型别，它是 Equality Comparable 的一个 model。

x,y,z 型别 x 的 object。

➢ Valid Expressions（有效表达式）

● 相等性（Equality）

```
x == y
```

返回型别：任何可转换为 bool 的型别。
先决条件：x 和 y 必须在 operator== 的 domain（域）内。此处 domain 一词系采用一般数学观念。

- 不等性（Inequality）

  `x != y`

  返回型别：　任何可转换为 `bool` 的型别。
  先决条件：　`x` 与 `y` 在 `operator==` domain 内。
  语义：　等价于 `!(x == y)`

➢ Invariants（恒长特性）

涉及 Equality Comparable 之表达式必须满足下列各恒长特性，这些表达式定义出所谓的等价关系（equivalence relation）：

- Identity（同一性）

  `&x == &y` 意味 `x == y`。

- Reflexivity（自反性）

  `x == x`。

- Symmetry（对称性）

  `x == y` 意味 `y == x`。

- Transitivity（传递性）

  如果 `x == y` 且 `y == z`，意味 `x == z`。

等价关系（equivalence relation）是数学中的基本概念。所谓等价关系意指「满足 reflexivity（自反性）、symmetry（对称性）和 transitivity（传递性）等恒长特性」的关系。

➢ Models

所有 C++ 内建的数值与指针型别，都是 Equality Comparable model。很多 STL 型别如 `vector<int>` 也是。一般而言，当且仅当 `T` 是 Equality Comparable，则 `vector<T>` 是 Equality Comparable。

6.4 可序性（Ordering）

6.4.1 LessThan Comparable

如果某个型别能够以某种次序排列，它便隶属于 LessThan Comparable。它必须能够以 `operator<` 作为比较动作，而 `operator<` 必须定义出某种一致的可序性。

技术上，`operator<` 一定会导致一种所谓的 partial ordering（偏序、部分可序，稍后定义），不过这不是个严苛条件。LessThan Comparable 的一个 refinement Strict Weakly Comparable，会在「以 `operator<` 导出的可序性」上加入更严格的限制。

➢ Refinement Of（强化自）

这个 concept 并不是任何其他 STL concepts 的强化（refinement）。

➢ Notation（符号）

x　　某型别，它是 LessThan Comparable 的一个 model。

x,y,z　型别 x 的 object。

➢ Valid Expressions（有效表达式）

- 小于（Less）

  `x < y`

  返回型别：　任何可转换为 `bool` 的型别。
  先决条件：　`x` 与 `y` 在 `operator<` domain 内。domain 一词源自一般数学观念。

- 大于（Greater）

  `x > y`

  返回型别：　任何可转换为 `bool` 的型别。
  先决条件：　`x` 与 `y` 在 `operator<` domain 内。
  语义：　等价于 `y < x`。

- 小于等于（Less or equal）

  `x <= y`

  返回型别：　任何可转换为 `bool` 的型别。
  先决条件：　`x` 与 `y` 在 `operator<` domain 内。
  语义：　等价于 `!(y < x)`。

- 大于等于（Greater or equal）

  `x >= y`

  返回型别：　任何可转换为 `bool` 的型别。
  先决条件：　`x` 与 `y` 在 `operator<` domain 内。
  语义：　等价于 `!(x < y)`。

上述四个操作符中，只有一个是真正最基础的原则，其他三个都只是为了方便而设计。我们可以选择任何一个作为这最基础的原则。此处我们选择 `operator<` 来定义其他三者。

➢ Invariants（恒长特性）

涉及 LessThan Comparable 之表达式必须满足下列恒长特性：

- irreflexivity（不自反性）

  `x < x` 必不成立。

- Antisymmetry（非对称性）

  `x < y` 意味 `!(y < x)`。

- Transitivity（传递性）

  如果 `x < y` 且 `y < z`，意味 `x < z`。

这些恒长特性即是所谓「partial ordering（偏序；部分可序）」的定义，此种关系满足 irreflexivity、transitivity 等恒长特性，至于 Antisymmetry 则是定理（theorem）而非公理（axiom），可由 irreflexivity 与 transitivity 导出。

➢ Models

所有 C++ 内建的数值型别都是 LessThan Comparable model。指针型别亦然 — 虽然任意两个指针的比较并不一定具有意义。这就是为什么我们要求 `x` 与 `y` 必须在 `operator<` domain 之内的原因。对指针而言，如果 `x` 和 `y` 是指向同一数组的指针，那么 `x` 和 `y` 便是位于 `operator<` domain 之中。

6.4.2 Strict Weakly Comparable

如果某个型别是 LessThan Comparable，而 `operator<` 除了满足 partial ordering 条件，还满足更严格的 strick weak ordering 条件，那么这个型别便隶属于 Strict Weakly Comparable。

Strick weak ordering 的正式定义如下：如果两元素具有「任何一个都不小于另一个」的性质，那么将它们视为具备「某种程度之等价关系」是合理的。

这个 strict weak ordering 定义，可能看起来既抽象又复杂，甚至你可能会怀疑为何我们要这么麻烦地定义它。两个理由可以说明为何 Strick Weakly Comparable 是一个很重要的 concept。第一，某些涉及次序关系（ordering relation）的算法（例如 `sort`）着实仰赖 strict weak ordering 性质。是的，LessThan Comparable 并未提供足够强烈的保证，保证排序算法能够正确运作。第二，就某个角度来看，任何你能想像的合理次序关系都可能是个 strict weak ordering。

➢ Refinement Of（强化自）

LessThan Comparable（p.86）

➢ Notation（符号）

`X`	某型别，它是 Strict Weakly Comparable 的一个 model。
`x,y,z`	型别 `X` 的 object。

➢ Definitions（定义）

如果 `x < y` 和 `y < x` 皆不为真，则 `x` 与 `y` 便是等价的（equivalent）。注意，由于 irreflexivity 性质，任何 `x` 一定等价于自身。此外，如果 `x` 等价于 `y`，则 `y` 等价于 `x`。

total ordering（全序）是相对于 partial ordering（偏序、部分可序性）的另一种关系。在此性质下，面对 `x` 和 `y`，如果不是 `x < y`，就是 `y < x`，要不就是 `x == y`。

➢ Valid Expressions（有效表达式）

除了定义于 LessThan Comparable 中的那些有效表达式，再没有其他的了。

➢ Invariants（恒长特性）

除了定义于 LessThan Comparable 中的那些恒长特性，还必须满足以下性质：

- Transitivity of equivalence（等价性的传递）

  如果 x 等价于 y，而且 y 等价于 z，则 x 等价于 z。亦即：

  ```
  !(x < y) && !(y < x) && !(y < z) && !(!z < y)
  ```

  意味着

  ```
  !(x < z) && !(z < x)
  ```

这在数学上便是所谓「等价（*equivalence*）」的意义。是的，这就是等价关系。Strict weak ordering 可将许多数值切割为等价类（equivalence class）的集合，一个等价类之内的所有数值，都互相等价。

Total ordering 是一种特别的 strict weak ordering。更明确地说，它的每个等价类（equivalence class）只含单一数值。

➢ Models

所有 C++ 内建的数值型别都是一种 Strict Weakly Comparable model。的确，整数的 `operator<` 是 total ordering，并非只是 strict weak ordering。

「隶属 strict weak ordering 但不属于 total ordering」的情况很常见 — 比你从正式定义来推想的情况还要普遍。任何比较行为如果只着眼于「object 中的某部分」，都可能是个 strict weak ordering。不在乎大小写的字符串比较动作，就是一种 strict weak ordering。同样道理，如果某个 class 具有两个字段 `first_name` 和 `last_name`，而其 `operator<` 只针对 `last_name`，那么这个 class 便是 Strict Weakly Comparable 的一个 model。当两数值 x 和 y 具有相同的 `last_name`，但 `first_name` 不同，它们是等价的（*equivalent*，因为 `x < y` 和 `y < x` 都不成立），但并非同一（*identical*）。

第 7 章

Iterators
迭代器 or 泛型指针

Iterators 是 STL 的核心。它们是泛型化的指针，是「指向其他 objects」的 objects。正如其名称（迭代器）所暗示，iterators 能够遍历由 objects 所形成的 range 之中。

Iterator 相当重要，它让我们得以将容器（containers）与作用其上的算法（algorithms）分离。大多数的算法自身并不直接操作于 container 上，而是操作于 iterators 所形成的 ranges 上。由于所有 STL containers 都具有 iterators，所以单一一个算法便可套用在许多不同的 container types 上。Container 只要能够提供该 iterators 访问元素的方法就行了。

Trivial Iterator concept 表现的是一种抽象概念，它表现出「一个 object，可用来指向其他某个 object」的概念。本章描述的另五种 concepts 则表现出「一个 object，不但能够指向其他 objects，还能够遍历由那些 objects 所形成的 range」。这五个 concepts 之间的主要差别在于它们所提供的迭代（iteration）方式不同。

7.1 Trivial Iterator

Trivial Iterator 是一种「可提领（dereference），使得以取用某个 object」的 concept。它不必支持数值运算如累加（increment）与比较（comparison）。Trivial Iterator 之所以称为 "trivial"（平淡无奇的），乃因它无法提供迭代行为。

➢ Refinement Of（强化自）

Assignable（p.83）与 Equality Comparable（p.85）。

➢ Associated Types（相关型别）

- Value type

  提领（dereferencing）一个 Trivial Iterator 后，所得结果（值）之型别。

➢ Notation（符号）

X	某型别，是 Trivial Iterator 的一个 model。
T	X 之 value type。
x, y	型别 X 的 object。
t	型别 T 的一个 object。

➢ Definitions（定义）

1. 某型别如果是 Trivial Iterator 的一个 model，那么它可以是 ***mutable***（可变的），意味该型别的 object 所指之值可以被更改。它也可以是 ***constant***（不变的），意味该型别的 object 所指之值不能被更改。举例来说，`int*` 是 mutable iterator type，而 `const int*` 是 constant iterator type。如果某个 iterator 是 mutable，则其 value type 必然是 Assignable 的一个 model，反之不成立。

2. Trivial Iterator 可能拥有一个 ***singular*** 值，意味对于大多数操作行为，包括相等性（equality）比较，其结果未有定义。它唯一保证支持的操作行为是「将一个 nonsingular iterator 赋予一个 singular iterator」。

3. Trivial Iterator 可以拥有可提领值（***dereferenceable*** value），意味提领动作将产生出明确定义的结果。可提领的 iterators 都是 nonsingular，反之则不成立。以 past-the-end 指针为例，它紧邻数组的尾端，虽然不可提领，却是 nonsingular（因为它与特定数组产生关联，而且有明确定义的操作行为作用于其上）。

4. 「令某个可提领的 iterator 无效」意指执行某种行为之后，该 iterator 可能变成无法提领，或是变成 singular。举个例子，如果 `p` 是个指针，则 `delete p` 便是令 `p` 不再有效。

➢ Valid Expressions（有效表达式）

除了定义于 Assignable、Equality Comparable 的各个表达式，以下表达式也必须是有效的：

- Default Constructor（缺省构造函数） 注意：并非已删除（请见新节前摘录）

  `X x;`

  必然结果： x 可以是 singular.

- Dereference（提领）

  `*x`

  先决条件： x 是可提领的（dereferenceable）。

返回型别： 任何可转换为 T 的型别[13]。

- Dereference Assignment（提领并赋值）

```
*x = t
```

型别条件： x 是 mutable iterator type。
先决条件： x 是 dereferenceable（可提领的）。
必然结果： *x 是 t 的复制品。

- Member Access（成员访问）

```
x->m
```

型别条件： T 必须是一个对 t.m 有所定义的型别。
先决条件： x 是 dereferenceable（可提领的）。
语义： 等价于 (*x).m。

原本的 HP STL 并没有对 iterators 定义 operator->。C++ standard 虽然定义了这个操作符，但并非目前所有的 STL 实现品皆有提供这个操作符。这个操作符得仰赖 C++ 语言的某部分特性，而这些特性并非广泛地被各家 C++ 编译器所支持。此刻能保证的是，以 (*x).m 取代 x->m 必能适用于任何地方。-> 操作符并未提供任何额外功能，它只是一个缩写。

➢ 复杂度保证（Complexity Guarantees）

所有满足 Trivial Iterator 需求条件的操作行为，其复杂度都必然是 amortized constant time（译注：请参阅数据结构相关书籍）。

➢ Invariants（恒长特性）

涉及 Trivial Iterator 的表达式，必须满足以下特性：

- Identity（同一性）
  当且仅当（if and only if）&*x == &*y，则 x == y。

➢ Models

事实上 C 语言也有 Trivial Iterator，尽管他们在 C 的型别系统中并不显眼。一个指针指向数组以外的某个 object，这就是 Trivial Iterator，因为只有「指向数组内之 object」的指针才可以被累加。

STL 定义的所有 iterator types 都具有累加性，也就是说 STL 的所有 iterator types 不但是 Trivial Iterator 的 models，也是本章所描述之其他 iterator concepts 的 models。

---

13 *x 返回型别的条件是「可转换为 T 的型别」，而非单纯的 T，因为事实上 *x 的返回型别常常不会是 T，而是 T 的 reference 或 const reference。有时候，「iterator 并非返回概念上所指的 object，而是返回某种 proxy object」反倒比较合理。proxy objects 是实现细节而非接口的一部分，所以它们并未出现在 Trivial Iterator 的需求条件上。

7.2 Input Iterator

隶属 Input Iterator 之类的 iterators 可被提领，使能取用某个 object。它也可以累加，以获得序列中的下一个 iterator。

Input Iterator 不一定得是 mutable，也不一定得支持惯常的提领语义（用以容许「多回（multipass）」算法）。例如，它不保证一定能够访问同一个 Input Iterator i 两次。

➢ Refinement Of（强化自）

Trivial Iterator（p.91）。

➢ Associated Types（相关型别）

- Value type

  `typename iterator_traits<X>::value_type`

  提领（dereferencing）一个 Input Iterator 后，所得结果（值）的型别。

- Different type

  `typename iterator_traits<X>::difference_type`

  这是一个有正负号的整数，用以表示某个 iterator 到另一个 iterator 之间的距离，或是某个 range 内的元素个数。

- Reference type

  `typename iterator_traits<X>::reference`

  这是「iterator 的 value type」的 reference。如果 iterator 是 mutable，则此型别为一个 mutable reference，如果 iterator 是 constant，则此型别为一个 `const` reference。如果 value type 以 `T` 表示，则 reference type 通常为 `T&` 或 `const T&`。

- Pointer type

  `typename iterator_traits<X>::pointer`

  这是「iterator 之 value type」的指针。如果 iterator 是 mutable，则此型别为一个 mutable 指针，如果 iterator 是 constant，则此型别为一个 `const` 指针。如果 value type 以 `T` 表示，则 pointer type 通常为 `T*` 或 `const T*`。

- Iterator category

  `typename iterator_traits<X>::iterator_category`

  此型别为以下 iterator tag types 之一：`input_iterator_tag`、`forward_iterator_tag`、`bidirectional_iterator_tag`、`random_access_iterator_tag`。所谓 iterator category（类属），是指对应于该 iterator 之「最具体（most specific）concept」的一个 tag type，而该 iterator 便是此一 concept 的一个 model。例如，`int*` 是 Random Access Iterator 的一个 model，它同样也是 Bidirectional Iterator 和 Forward Iterator 的一个 model。其 category（类属）是 `random_access_iterator_tag`，因为 Random Access Iterator 对 `*int` 而言是最具体（most specific）的 concept。（噢是的，Random Access Iterator 是其他所有 iterator concepts 的强化。）

➢ Notation（符号）

X	某型别，是 Input Iterator 的一个 model。
T	X 的 value type。
i, j	型别 X 的 object。
t	型别 T 的 object。

➢ Definitions（定义）

1. 如果某个 iterator 所指位置超越 container 的最后一个元素，它就是所谓 *past-the-end*。这种 iterator 不但是 nonsingular 而且是 nondereferenceable（不可提领的）。

2. 如果某个 iterator 是 dereferenceable（可提领的）或是 past-the-end，那么它就是有效的（*valid*）。

3. 如果 iterator i 有「下一个」iterator，我们说 i 是可累加的（incrementable）；换句话说如果 ++i 有明确定义，i 就是可累加的。*Past-the-end* iterator 不可累加。

4. 如果将有限次数的 operator++ 套用在 Input Iterator i 身上，便可使 i==j[14]的话，则 Input Iterator j 对于 i 而言是可及的（*reachable*）。

5. 符号 [i,j) 表示「从 i 起始至 j，但不包含 j」的一个 range。如果 i 和 j 都是有效的 iterators，而且 j 对 i 而言是可及的（reachable），则 [i,j) 是一个有效的 range。在有效区间 [i,j) 内的任何一个 iterator 都是可提领的（dereferenceable），而且 j 若不是 dereferenceable 就是 past-the-end。「有效 range 内的每个 iterator 都可提领」这一事实是根据「每个可累加的 iterators 都必须可提领」而来。

➢ Valid Expressions（有效表达式）

除了定义于 Trivial Iterator 的表达式，下列表达式也必须有效：

- Dereference（提领）

  *i

  返回型别：　任何可转换为 T 的型别。
  语义：　返回 i 所指向的元素。
  先决条件：　i 是可提领的（dereferenceable）。

- Preincrement（前置累加）

  ++i

  返回型别：　X&
  先决条件：　i 是可提领的（dereferenceable）。

---

[14] 对 Input Iterator 而言，i==j 并不一定意味 ++i == ++j。Forward Iterator concept 则无此限制。

必然结果： i 是可提领的（dereferenceable）或 past-the-end。++i 执行之后，i 的旧值不再需为可提领的。也不必处于 operator==的 domain（域）中。Input Iterator 之 model type 不需在单一 range 内支持一个以上的现行有效的（active）iterator。

- Postincrement（后置累加）

```
(void) i++
```

先决条件： i 是可提领的（dereferenceable）。
语义： 等价于 (void) ++i。
必然结果： i 是可提领的（dereferenceable）或 past-the-end。

- Postincrement and dereference（后置累加并提领）

```
*i++
```

返回型别： T
先决条件： i 是可提领的（dereferenceable）。
语义： 等价于：

```
T t = *i;
++i;
return t;
```

必然结果： i 是可提领的（dereferenceable）或 past-the-end。

➢ 复杂度保证（Complexity Guarantees）

所有操作行为的复杂度必须是 amortized constant time（译注：请参阅数据结构相关书籍）。

➢ Models

所有 STL iterators 几乎都是 Input Iterator 的 model，传统指针亦然。例如 deque<int>::iterator 既是其他 iterator concepts 的一个 model，同时也是 Input Iterator 的一个 model。

istream_iterator（p.354）是「隶属于 Input Iterator，但却不是其他 iterator concepts 的 model type」的例子之一。此一型别的确具有 Input Iterator 奇妙而受到束缚的语义。

7.3 Output Iterator

Output Iterator 提供一种机制，用来存储数值序列，但不必提供取出数值的方法。Output Iterators 在某些情况下恰与 Input Iterators 相反，不过它们的接口更受束缚。它们不必一定得支持成员访问（member access）或相等比较（equality），也不必拥有相关型别 difference type 或 value type。

除此之外，Output Iterator 与 Input Iterator 具有同样的限制。它们不支持多回（multipass）算法，也不保证你能够访问同一个 Output Iterator 两次。

直觉上你可以把 Output Iterator 想像为磁带：可以写数值到目前位置上，然后前进到下一个位置；但是不能读出，也不能回转或倒带。

➢ Refinement Of（强化自）

Assignable（p.83）。

注意，Output Iterator 并不是 Equality Comparable（p.85）的强化（refinement）：Output Iterator 不必定义 `operator==`。同样请注意，和其他所有 iterator concepts 不同的是，Output Iterator 并非是 Trivial Iterator 的强化（refinement）。

➢ Associated Types（相关型别）

- Iterator category

  `typename iterator_traits<X>::iterator_category`

  此为下列 iterator tag types 其中之一：`output_iterator_tag`、`forward_iterator_tag`、`bidirectional_iterator_tag`、`random_access_iterator_tag`。细节请参考 Input Iterator 的需求条件（p.94）。

注意，Output Iterator 不具有其他众多 iterator concepts 所具有的相关型别。尤其是它并没有定义相关的 value type。其他 iterator types，包括 Input Iterator 和 Trivial Iterator，都定义有所谓的 value type：提领某个 iterator 后的返回值型别。对 Output Iterator 而言，由于提领动作（单参操作符 `operator*`）并不会返回有用的值，所以 Output Iterator 不采用 value type 概念。提领操作符只能被用在如 `*x = t` 的赋值动作上。

尽管大略而言 Input Iterator 和 Output Iterator 是两个对称的 concepts，但在取出数值与存入数值的行为之间，却有很重要的非对称性。对 Input Iterator 而言，`operator*` 必须返回某个独一无二的型别，但对 Output Iterator 而言，在表达式 `*x=t` 中，没有理由让 `operator=` 只接受单一型别。因此 Output Iterator 不必具有任何特定的 "value type"。

➢ Notation（符号）

`X`	某型别，是 Output Iterator 的一个 model。
`x, y`	型别 `X` 的 object。

➢ Definitions（定义）

1. 如果 `x` 是 type `X` 的一个 Output Iterator，则表达式 `*x=t` 会将 `t` 存入 `x` 内。注意，`operator=` 如同其他 C++ 函数一样可被重载（overloaded），事实上它甚至可以是个 function template。因此 `t` 的型别有众多候选者。对一个型别为 `T` 的 object `t`，如果 `*x=t` 有明确定义（well-defined），而且不需任何「并非无关痛痒（nontrivial）」之转换函数作用于 `t` 身上的话，那么 `T` 将是 `X` 的 value types 之一。

2. Output Iterator 可以是 singular — 意即大部分操作行为（包括 copying 和 dereference assignment）的结果都未有定义。singular iterator 的各种操作行为中，唯一需要支持的是将一个 nonsingular iterator 赋值给一个 singular iterator。

3. Output Iterator 可以是 dereferenceable，表示通过它所完成的赋值动作是有定义的。Dereferenceable iterator 总是 nonsingular，但 nonsingular iterator 不一定是 dereferenceable。

➢ Valid Expressions（有效表达式）

除了定义于 Assignable 的各个表达式，以下表达式也必须有效：

- Default constructor（缺省构造函数） **注意：本项已删除（详见原书勘误表）**

```
X x;
X()
```

必然结果： x 可以是 singular。

- Copy constructor（拷贝构造函数）

```
X y(x);
X y = x
```

先决条件： x 是可提领的（dereferenceable）。
必然结果： 表达式 `*y = t` 与表达式 `*x = t` 的行为相同。注意，任何时候对某个特定的 Output Iterator 而言，都应该只有一份有效复制品（active copy）。也就是说使用 Output Iterator x 的复制品 y 后，原来的 iterator x 就不应该再使用了。

- Assignment operator（赋值操作符）

```
y = x
```

返回型别： X&
先决条件： x 是 nonsingular。
必然结果： 表达式 `*y = t` 与表达式 `*x = t` 行为相同。

- Dereference assignment（提领并赋值）

```
*x = t
```

型别条件： t 可转换为 X 之「value types 集合」内的 type。
返回型别： 返回结果不会被使用（或曰不可用）。
先决条件： x 是可提领的（dereferenceable）。若是先前已通过 x 做了赋值动作，那么就会有所谓的 intervening increment。（通过 Output Iterator x 的赋值行为预期将以「累加 x」来替代，而且应该在 x 被累加之前先通过 x 做赋值动作。任何其他顺序的行为结果并无定义。举个例子，`*x = t; ++x; *x = t2; ++x` 是可被接受的操作次序，但 `*x = t; ++x; ++x; *x = t2` 就不是了。）

- Preincrement（前置累加）

  `++x`

  返回型别： `X&`
  先决条件： `x` 是 dereferenceable。在此之前通过 `x` 有一个赋值动作。如果 `x` 先前曾被累加，就会有所谓的 intervening increment 作用于 `x`。
  必然结果： `x` 指向下一个可存入数值的位置。

- Postincrement（后置累加）

  `x++`

  返回型别： void
  先决条件： `x` 是 dereferenceable。在此之前通过 `x` 有一个赋值动作。如果 `x` 先前曾被累加，就会有所谓的 intervening increment 作用于 `x`。
  语义： 等价于 `(void) ++x`。
  必然结果： `x` 指向下一个可存入数值的位置。

- Postincrement and dereference（后置累加并提领）

  `*x++ = t`

  型别条件： `t` 可转换为 `X` 之「value types 集合」内的 type。
  返回型别： 返回结果不会被使用（或曰不可用）。
  先决条件： `x` 是 dereferenceable。如果先前曾通过 `x` 做赋值动作，就会有所谓的 intervening increment。
  语义： 等价于：

  ```
  *x = t;
  ++x;
  ```

  必然结果： `x` 指向下一个可存入数值的位置。

注意，表达式 `*x=t` 必须有定义，但表达式 `*x` 自身不需具备任何有用的值。也就是说 Output Iterator 可以不必指向任何东西；它可以只用来作为赋值动作（assignment）的凭借。

如果你要实现一个 Output Iterator class `X`，那么实现表达式 `*x = t` 的明智做法就是定义 `X::operator*`，让它返回某个 private class `X_proxy` 的 object。并适当定义 member function `X_proxy::operator=`。注意，你可以将 `X_proxy::operator=` 重载，或把它定义为一个 member template。如此一来便可经由一个 Output Iterator class `X` 赋值多种型别。

➢ 复杂度保证（Complexity Guarantees）

所有操作行为的复杂度必须是 amortized constant time（译注：请参阅数据结构相关书籍）。

➢ Models

以下型别都 Output Iterator:

```
front_insert_iterator (p.345)
back_insert_iterator (p.348)
insert_iterator (p.351)
ostream_iterator (p.357)
```

7.4 Forward Iterator

Forward Iterator 符合线性数值序列（linear sequence of values）的直觉概念。Forward Iterator 不同于 Input Iterator 和 Output Iterator，可应用于多回（multipass）算法。然而，顾名思义，Forward Iterator 不允许在 range 中回头移动，只能向前（译注：向 range 尾端）移动。

Forward Iterator 的 model types 可以是 mutable 或 constant，就像 Trivial Iterator 所定义的需求条件一样（p.91）。

➢ Refinement Of（强化自）

Input Iterator（p.94）、Output Iterator（p.96）、Default Constructible（p.84）。

➢ Associated Types（相关型别）

和 Input Iterator 相同。

➢ Notation（符号）

X	某型别，是 Forward Iterator 的一个 model。
T	X 的 value type。
i, j	型别 X 的 object。
t	型别 T 的 object。

➢ Valid Expressions（有效表达式）

Forward Iterator 并没有比 Input Iterator 定义出更多新的表达式。但它开放了 Input Iterator 的某些限制。更明确地说，将一个 Forward Iterator 累加，并不会造成旧值的副本无效。此外如果 `i` 和 `j` 都可提领并且 `i == j`，那么保证 `++i == ++j`。因此我们可以对同一个 Forward Iterator 遍历两次。

- Default constructor（缺省构造函数） **译注：本项乃新增（详见原书勘误表）**

  ```
  X x;
  X()
  ```

  必然结果： `x` 可以是 singular。

- Preincrement（前置累加）

  ```
  ++i
  ```

  返回型别： `X&`
  先决条件： `i` 是可提领的（dereferenceable）。
  语义： `i` 从此指向下一个值。
  必然结果： `i` 是 dereferenceable 或 past the end。`&i == &++i`。如果 `i == j`，则 `++i == ++j`。

- Postincrement（后置累加）

  ```
  i++
  ```

  返回型别： `X`
  先决条件： `i` 是可提领的（dereferenceable）。
  语义： 等价于：

  ```
  X tmp = i;
  ++i;
  return tmp;
  ```

  必然结果： `i` 是 dereferenceable 或 past the end。

- Dereference（提领）

  ```
  *i
  ```

  返回型别： 如果 `X` 是 mutable iterator type，则返回型别是一个可被赋值的 lvalue。如果 `X` 是 constant iterator type，则返回型别是一个 rvalue 或 `const` lvalue。返回型别亦即 `X` 的 reference type，通常是 `T&` 或 `const T&`。
  语义： 返回 `i` 所指的元素。
  先决条件： `i` 是可累加的。

➢ 复杂度保证（Complexity Guarantees）

所有操作行为的复杂度必须是 amortized constant time（译注：请参阅数据结构相关书籍）。

➢ Models

以下型别皆是 Forward Iterator：

```
slist<int>::iterator
slist<double>::const_iterator
hash_set<string>::iterator
char*（它也是一个 Bidirectional Iterator 和 Random Access Iterator）
```

7.5 Bidirectional Iterator

Bidirectional Iterator 是一种可累加（incremented）也可递减（decremented）的 iterator。「可递减」正是 Bidirectional Iterator 与 Forward Iterator 之间的唯一差别。

➢ Refinement Of（强化自）

Forward Iterator（p.100）。

➢ Associated Types（相关型别）

和 Forward Iterator 相同。

➢ Notation（符号）

X	某型别，是 Bidirectional Iterator 的一个 model。
T	X 的 value type。
i, j	型别 X 的 object。
t	型别 T 的 object。

➢ Valid Expressions（有效表达式）

除了定义于 Forward Iterator 的各表达式，以下表达式也必须有效：

- Predecrement（前置递减）

  `--i`

  返回型别： X&

  先决条件： i 是 dereferenceable 或 past the end。存在有一个 dereferenceable iterator j，使得 i == ++j。

  语义： i 从此指向前一个值。

  必然结果： i 是 dereferenceable。&i == &--i。如果 i == j，则 --i == --j。如果 j 是 dereferenceable 并且 i == ++j，则 --i == j。

- Postdecrement（后置递减）

  `i--`

  返回型别： X

  先决条件： i 是 dereferenceable 或 past the end。存在有一个 dereferenceable iterator j，使得 i == ++j。

语义：　　等价于：

```
X tmp = i;
--i;
return tmp;
```

➢ 复杂度保证（Complexity Guarantees）

所有操作行为的复杂度必须是 amortized constant time（译注：请参阅数据结构相关书籍）。

➢ Invariants（恒长特性）

表达式如果涉及 Bidirectional Iterator，必须满足以下性质：

- 累加（increment）与递减（decrement）的对称性

  如果 `i` 是 dereferenceable，则

  `++i; --i;`

  是无用的（无影响的，null）操作。同样道理，如果 `--i` 有明确定义，则

  `--i; ++i;`

  是无用的（无影响的，null）操作。

➢ Models

以下型别都是 Bidirectional Iterator：

```
list<int>::iterator
set<string>::iterator
char*（它同时也是一个 Random Access Iterator）
```

7.6 Random Access Iterator

Random Access Iterator 是一种可累加（incremented）也可递减（decremented）的 iterator（这一点和 Bidirectional Iterator 一样），并在向前或回头移动任意步数时，以及在比较两个 iterators 时，耗用固定时间（constant-time）。

Random Access Iterator 提供了与 C 指针算术（pointer arithmetic）类似的所有操作能力。

➢ Refinement Of（强化自）

Bidirectional Iterator（p.102）、Strict Weakly Comparable（p.88）。

➢ Associated Types（相关型别）

和 Bidirectional Iterator 相同。

➢ Notation（符号）

X	某型别，是 Random Access Iterator 的一个 model。
T	X 的 value type。
D	X 的 difference type。
i, j	型别 X 的 object。
t	型别 T 的 object。
n	型别 D 的 object。

➢ Valid Expressions（有效表达式）

除了定义于 Bidirectional Iterator 的表达式，以下表达式也必须有效：

● Iterator addition（泛型指针加法）

```
i += n
```

返回型别： X&

先决条件： i 之后，包括 i，必须具有 n 个 dereferenceable 或 past-the-end iterators（当 n 为正），否则便是在 i 之前要有 -n 个这样的 iterators（当 n 为负）。

语义： 当 n > 0，等价于执行 n 次 ++i。当 n < 0，等价于执行 |n| 次的 --i。当 n == 0，i += n 无作用（null operation）。注意，所谓「等价」意味 i += n 的结果与累加 i（或递减 i）n 次所产生的结果相同，但并不意味应该如此这般地实现 operator+=；事实上那甚至是不被允许的做法，因为不论 n 是多少，i += n 保证耗用 amortized const time。

必然结果： i 是 dereferenceable 或 past the end。

● Iterator addition（泛型指针加法）

```
i + n
n + i
```

返回型别： X

先决条件： 与 i += n 相同。

语义： 等价于：

```
X tmp = i;
return tmp += n;
```

i + n 和 n + i 两个形式意义完全相同。

必然结果： i 是 dereferenceable 或 past the end。

● Iterator substraction（泛型指针减法）

```
i -= n
```

返回型别： `X&`
先决条件： `i` 之前，包括 `i`，必须有 `n` 个 dereferenceable 或 past-the-end iterators（当 `n` 为正），否则便是在 `i` 之后要有 `-n` 个这样的 iterators（当 `n` 为负）。
语义： 等价于 `i += (-n)`
必然结果： `i` 是 dereferenceable 或 past the end。

- Iterator substraction（泛型指针减法）

```
i - n
```

返回型别： `X`
先决条件： 和 `i -= n` 相同。
语义： 等价于：

```
X tmp = i;
return tmp -= n;
```

必然结果： `i` 是 dereferenceable 或 past the end。

- Difference（差距）

```
i - j
```

返回型别： `D`
先决条件： 如果不是 `j` 可到达 `i`，就是 `i` 可到达 `j`。或者两者皆可。
语义： 返回数值 `n`，使得 `i == j + n`。

- Less（小于）

```
i < j
```

返回型别： 任何可转换为 `bool` 的型别。
先决条件： 如果不是 `j` 可到达 `i`，就是 `i` 可到达 `j`。或者两者皆可。拿这一点和 Strict Weakly Comparable 的先决条件「`i` 和 `j` 都在 `operator<` 的 "domain"（域）中」做比较，很有正面意义。本质上，这就是该 "domain" 的定义：成对的 iterators 所形成的集合，每一对 iterators 中的某一个可以到达另一个。这很符合 C 的原则：两个指针只有在指向同一个数组时，才可以互相比较。
语义： 正如 Strict Weakly Comparable 所叙述，其他各个比较操作符（`>`、`<=` 和 `>=`）都是根据 `operator<` 而定义，所以它们也具有 Strict Weakly Comparable 所描述的语义。

- Element access（元素访问）

```
i[n]
```

返回型别： 任何可转换为 `T` 的型别。
先决条件： `i+n` 必须存在，并且可提领（dereferenceable）。

语义: 等价于 `*(i + n)`[15]。

- Element assignment（元素赋值）

  `i[n] = t`

  型别条件: x 必须是个 mutable iterator type。
  返回型别: 任何可转换为 T 的型别。
  先决条件: i+n 必须存在，并且可提领（dereferenceable）。
  语义: 等价于 `*(i + n) = t`。
  必然结果: i[n] 成为 t 的一个副本。

➢ 复杂度保证（Complexity Guarantees）

所有 Random Access Iterator 操作行为的复杂度都必须是 amortized constant time（译注：请参阅数据结构相关书籍）。这项保证正是 Random Access Iterator 成为独立的一个 concept 的唯一理由。是的，其每一个指针算术运算（iterator arithmetic）都可以针对 Bidirectional Iterator 定义出来 — 这正是算法 distance（p.181）和 advance（p.183）的作为。这两个 iterator concepts 之间的差别仅仅在于，Bidirectional Iterator 乃属线性时间（linear time），而 Random Access Iterator 却必须在 amortized constant time 内完成元素的随机访问。对于那些「两种 iterator types 皆可用」的算法而言，这里提供了一个重要的暗示。

➢ Invariants（恒长特性）

凡表达式涉及 Random Access Iterator 者，都必须满足以下性质:

- 加法和减法的对称性

  如果 i+n 有明确定义，则

  ```
  i += n; i -= n;
  ```

  以及

  ```
  (i + n) - n
  ```

  都是无用的（无影响的，null）行为。同样道理，如果 i-n 有明确定义，则

  ```
  i -= n; i += n
  ```

  以及

  ```
  (i - n) + n
  ```

  都是无用的（无影响的，null）行为。

- 加法（addition）和差距（distance）之间的关系

  如果 `i - j` 有明确定义，则 `i == j + (i - j)`

---

[15] C 语言有一个奇特语法：如果 p 是指针，则 p[n] 等价于 n[p]。然而，一般的 Random Access Iterator 不需保证支持此一性质，它只须支持 i[n] 就好。这项束缚并不重要，因为 p[n] 与 n[p] 的等价性除了用于 obfuscated C contest，其他地方并未采用。（译注：关于 obfuscated C contest，可参考 http://www.ioccc.org/ 网站。）

- 差距（distance）和可及性（reachability）
  如果 `j` 可到达 `i`，则 `i - j >= 0`。

➢ Models

任何指针型别都模塑出（models）Random Access Iterator。指针可算是 Random Access Iterator 中最重要的一个 model 了。此外，下列型别也都是 Random Access Iterator：

```
vector<int>::iterator
vector<int>::const_iterator
deque<int>::iterator
deque<int>::const_iterator
```

第 8 章

Function Objects
函数对象

Function object 或称为 *functor*（两者同义），是一种「能以一般的函数调用语法来调用」的 object。Function pointer（函数指针）是一种 function object，所有具备 `operator()` member function 的 objects 亦然。相形之下，指向 nonstatic member function 的指针就不是 function object。

Generator、Unary Function 和 Binary Function 是基本的 function object concepts：这些 objects 分别能够以 `f()`、`f(x)` 和 `f(x,y)` 的形式被调用（当然，尚可延伸出三个参数或更多参数的函数，但实用上没有任何一个 STL 算法需要两个参数以上的 function objects）。STL 定义的其他所有 function objects 都是这三个 concepts 的强化（refinements）。

Function objects 之中返回 `bool` 者，是一个重要特例。返回 `bool` 之 Unary Function 称为 Predicate，返回 `bool` 之 Binary Function 称为 Binary Predicate。

Function objects 和 adaptable function objects 之间有一个重要而有点微妙的差异。一般而言，function objects 的引数型别受到限制。型别限制不一定是很单纯的，例如 `operator()` 可能被重载，也可能是个 member template，亦或两者都是。对程序而言，决定引数型别的方式不必局限于一种。

`Adaptable` function object 可以指定引数及返回值型别，而且可以包含嵌套型别（nested type）声明，好让这些型别可以被命名以及被程序使用。如果 type `F0` 是 Adaptable Generator 的一个 model，那么它必须定义嵌套型别 `F0::result_type`。同样道理如果 `F1` 是 Adaptable Unary Generator 的一个 model，那么它必须定义嵌套型别 `F1::argument_type` 和 `F1::result_type`；如果 F2 是 Adaptable Binary Generator 的一个 model，那么它必须定义嵌套型别 `F2::first_argument_type`、`F2::second_argument_type` 和 `F2::result_type`。

Adaptable function objects 之所以称为 "adaptable"，是因为它们可用于 function object adapter，它们是「能够被转换为或被用来操作其他 function objects」的一种 function objects。STL 定义了很多不同的 function object adapters，包括 `unary_negate`、`unary_compose`、`bind1st`、`bind2nd`。

8.1 基本的 Function Objects

8.1.1 Generator

Function object 是一种能以一般 C++ 函数之调用方法来「调用」的 object，Generator 则是一种不需任何引数的 function object。

➢ Refinement Of（强化自）

Assignable（p.83）。

➢ Associated Types（相关型别）

Result type

调用 Generator 后，返回值的型别。

➢ Notation（符号）

`F`	某型别，是 Generator 的一个 model。
`Result`	`F` 的 result type。
`f`	型别 `F` 的 object。

➢ Definitions（定义）

所谓 Generator 的 "range"，意指其可能返回值所形成的集合。

➢ Valid Expressions（有效表达式）

Function call（函数调用）

`f()`

返回型别：	`Result`
语义：	返回型别为 `Result` 的某值。注意，调用 `f` 两次可能会返回不同的结果。Generator 可能会参考局部状态（local state）或执行 I/O 动作……表达式 `f()` 甚至可以改变 `f` 的状态 — 也就是说 `opreator()` 不必是个 const member function。例如 `f` 可以扮演「伪乱数生成器（pseudo-random number generator）」的角色。
必然结果：	返回值位于 `f` 的 range 之中。

➢ Models

凡一个函数能够返回某值，而且不需任何引数，它便是一个 Generator。例如型别如下的函数指针便是 Generator 的一个 model。

```
int (*)()
```

8.1.2 Unary Function

Function object 是一种能以一般 C++ 函数调用方法来调用的 object，而 Unary Function 是一种只需单一引数的 function object。

➢ Refinement Of（强化自）

Assignable（p.83）。

➢ Associated Types（相关型别）

Argument type

Unary Function 之引数型别。

Result type

调用 Unary Function 后，返回值的型别。

➢ Notation（符号）

`F`	某型别，是 Unary Function 的一个 model。
`X`	`F` 的 argument type。
`Result`	`F` 的 result type。
`f`	型别 `F` 的一个 object。
`x`	型别 `X` 的一个 object。

➢ Definitions（定义）

所谓 Unary Function 的 "domain"，是指所有「有资格作为其引数」的数值集合。

所谓 Unary Function 的 "range"，是指其所有可能返回值的集合。

➢ Valid Expressions（有效表达式）

Function call（函数调用）

`f(x)`

返回型别：	`Result`
先决条件：	`x` 位于 `f` 的 "domain" 之中。
语义：	以 `x` 为引数，调用 `f`，返回型别为 `Result` 的某值。注意，调用 `f` 两次可能会返回不同的结果 — 即使引数相同。Unary Function 可能会参考局部状态（local state）或执行 I/O 动作。表达式 `f(x)` 甚至可以改变 `f` 的状态 — 也就是说 `opreator()` 不必是个 `const` member function。
必然结果：	返回值位于 `f` 的 range 之中。

➢ Models

凡一个函数能够返回某值，而且需要单一引数，它便是一个 Unary Function。例如型别如下的函数指针便是 Unary Function 的一个 model：

```
double (*)(double)
```

subtractive_rng class（p.402）是个较为有趣的例子。这种类形的 object 拥有局部状态（local state），而且，两个成功的 subtractive_rng 调用通常不会返回相同的结果。

8.1.3 Binary Function

Function object 是一种能以一般 C++ 函数调用方法来调用的 object，而 Binary Function 是一种需要两个引数的 function object。

➢ Refinement Of（强化自）

Assignable（p.83）。

➢ Associated Types（相关型别）

First argument type
: Binary Function 的第一个引数型别。

Second argument type
: Binary Function 的第二个引数型别。

Result type
: 调用 Binary Function 后，返回值的型别。

➢ Notation（符号）

F	某型别，是 Binary Function 的一个 model。
X	F 的 first argument type。
Y	F 的 second argument type。
Result	F 的 result type。
f	型别 F 的 object。
x	型别 X 的 object。
y	型别 Y 的 object。

➢ Definitions（定义）

- 所谓 Binary Function 的 "domain"，是指有资格作为其引数的 (x,y) 集合。
- 所谓 Binary Function 的 "range"，是指其可能返回值所形成的集合。

➢ Valid Expressions（有效表达式）

● Function call（函数调用）

```
f(x,y)
```

返回型别： `Result`
先决条件： 引数 `(x,y)` 位于 `f` 的 "domain" 之中。
语义： 以 `x` 和 `y` 作为引数，调用 `f`，返回型别为 `Result` 的某值。注意，即使是以同一组引数调用 `f` 两次，可能返回不同的结果。Binary Function 可能参考局部状态（local state）或执行 I/O 动作。表达式 `f(x,y)` 可以改变 `f` 的状态 — 也就是说 `opreator()` 不必是个 `const` member function。
必然结果： 返回值位于 `f` 的 range 之中。

➢ Models

凡一个函数能够返回某值，而且需要两个引数，它便是一个 Binary Function。例如型别如下的函数指针便是 Binary Function 的一个 model：

```
double (*)(double, double)
```

8.2 Adaptable Function Objects

8.2.1 Adaptable Generator

所谓 Adaptable Generator，是一种 Generator 并嵌套定义其 result type。这个嵌套定义（nested typedef）使我们得以运用 function object adapter，这是 Adaptable Generator 和其他任何 Generator 之间唯一的差别。举个例子，函数指针 `T(*f)()` 是 Generator 的一个 model，但不是 Adaptable Generator 的一个 model，因为 `f` 虽身为函数指针，但表达式 `f::result_type` 没有意义。

➢ Refinement Of（强化自）

Generator（p.110）。

➢ Associated Types（相关型别）

● Result type

```
F::result_type
```

调用 Adaptable Generator 后，返回值的型别。

➢ Notation（符号）

`F` 某型别，是 Adaptable Generator 的一个 model。

➢ Valid Expressions（有效表达式）

和 Generator 完全相同。

➢ Models

STL 并没有定义出 Adaptable Generator 的任何 model。下面是我自行定义的一个例子：

```
struct counter
{
  typedef int result_type;

  counter(result_type init = 0) : n(init) { }
  result_type operator() { return n++; }

  result_type n;
};
```

8.2.2 Adaptable Unary Function

Adaptable Unary Function 是一种 Unary Function，并嵌套定义了 argument type 和 result type。这些嵌套定义(nested typedef)让我们得以运用 function object adapter，例如 `unary_negate` 和 `unary_compose`，这正是 Adaptable Unary Function 与其他任何 Unary Function 之间唯一的差别。举个例子，函数指针 `T (*f)(X)` 是个 Unary Function，但不是一个 Adaptable Unary Function，因为 `f` 虽身为一个函数指针，但表达式 `f::argument_type` 和 `f::result_type` 皆无意义。

定义一个 Adaptable Unary Function 之 model type 时，你必须定义其嵌套型别。最简单的做法就是让它继承 base class `unary_function`。`unary_function` 是一个空的 class，没有 member function 也没有 member variable，只有型别声明。它存在的唯一理由就是让我们能够更方便地定义 Adaptable Unary Function。`unary_function` 很类似 base class `iterator`。

➢ Refinement Of（强化自）

Unary Function（p.111）。

➢ Associated Types

- Argument type

  `F::argument_type`

  `F` 之引数型别。

- Result type

  `F::result_type`

  调用 Adaptable Unary Function 后，返回值的型别。

➢ **Notation（符号）**

`F` 某型别，是 Adaptable Unary Function 的一个 model。

➢ **Valid Expressions（有效表达式）**

和 Unary Function 完全相同。

➢ **Models**

以下都是 Adaptable Unary Function:

```
negate
identity
pointer_to_unary_function
```

`pointer_to_unary_function` adapter 特别有趣，因为它可以把函数指针当作 Adaptable Unary Function 使用。函数指针 `Result (*f)(Arg)` 不是个 Adaptable Unary Function，但 `ptr_fun(f)` 却是。

8.2.3 Adaptable Binary Function

Adaptable Binary Function 是一种 Binary Function，并嵌套定义其 argument type 和 result type。这些嵌套定义（nested typedef）使我们得以使用 function object adapter 如 `binder1st` 和 `binder2nd`，而这正是 Adaptable Binary Function 与其他任何 Binary Function 之间唯一的差别。举个例子，函数指针 `T (*f)(X,Y)` 是个 Binary Function，但不是一个 Adaptable Binary Function。

当你定义一个 Adaptable Binary Function 的 model type，你必须提供那些嵌套定义。最简单的做法就是继承自 base class `binary_function`，那是一个空的 class，没有 member function 也没有 member variable，只有型别声明。它存在的唯一理由就是让我们更方便地定义 Adaptable Binary Function。`binary_function` 很类似 base class `iterator`。

➢ **Refinement Of（强化自）**

Binary Function（p.112）。

➢ **Associated Types（相关型别）**

- First argument type

  `F::first_argument_type`

  `F` 的第一引数型别

- Second argument type

  `F::second_argument_type`

  `F` 的第二引数型别

- Result type

  `F::result_type`

  调用 Adaptable Binary Function 后，返回值的型别。

➢ Notation（符号）

`F`	某型别，是 Adaptable Binary Function 的一个 model。

➢ Valid Expressions（有效表达式）

和 Binary Function 完全相同。

➢ Models

STL 内含很多预定义的 Adaptable Binary Function，主要扮演数值运算和逻辑运算的角色。下列是 Adaptable Binary Function 的几个例子：

```
plus
multiplies
less
equal_to
pointer_to_binary_function
```

8.3 Predicates

8.3.1 Predicate

所谓 Predicate 是一种 Unary Function，其结果表现出某个条件的真与假。Predicate 是个函数，需要一个 `int` 引数，如果该引数为正值，就返回 `true`。

➢ Refinement Of（强化自）

Unary Function（p.111）。

➢ Associated Types（相关型别）

Predicate 具有和 Unary Function 完全相同的 associated type，但有一个额外限制：其返回型别必须可以转换为 `bool`。

➢ Notation（符号）

`F`	某型别，是 Predicate 的一个 model。
`X`	`F` 的 argument type。
`f`	型别 `F` 的 object。
`x`	型别 `X` 的 object。

➢ Valid Expressions（有效表达式）

● Function call（函数调用）

```
f(x)
```

返回型别：	任何可转换为 bool 的型别。
先决条件：	x 必须位于 f 的 "domain" 内。
语义：	如果满足 f 所代表的条件，就返回 true；反之则返回 false。
必然结果：	返回值不是 true 就是 false。

➢ Models

需要单一引数，而且返回 true 或 false，这样的函数就是 Predicate，C 标准函数库中有关字符分类的函数都是 Predicate。以下的函数指针也是 Predicate 的一个 model：

```
bool (*)(char)
```

8.3.2 Binary Predicate

所谓 Binary Predicate 是一种 Binary Function，其结果表现出某条件的真与假。Binary Predicate 是个函数，接受两个引数并测试它们是否相等。

➢ Refinement Of（强化自）

Binary Function（p.112）。

➢ Associated Types（相关型别）

Binary Predicate 具有和 Binary Function 完全相同的 associated type，但有一个额外限制：其返回型别必须可以转换为 bool。

➢ Notation（符号）

F	某型别，是 Binary Predicate 的一个 model。
X	F 的 first argument type。
Y	F 的 second argument type。
f	型别 F 的 object。
x	型别 X 的 object。
y	型别 Y 的 object。

➢ Valid Expressions（有效表达式）

● Function call（函数调用）

```
f(x,y)
```

返回型别： 任何可转换为 bool 的型别。
先决条件： (x,y) 必须在 f 的 "domain" 之中。
语义： 如果满足 f 所代表的条件，就返回 true，反之则返回 false。
必然结果： 返回值不是 true 就是 false。

➢ Models

以下都是 Binary Predicate 的例子。第一个是函数指针，其他两个是 STL 定义的 function object class：

```
bool (*)(int, int)
equal_to<string>
less_equal<double>
```

8.3.3 Adaptable Predicate

Adaptable Predicate 是一种 Predicate，同时也是一种 Adaptable Unary Function。也就是说，它是一个「返回值型别可转换为 bool」的 Unary Function，并且嵌套定义了 argument type 和 return type。

➢ Refinement Of（强化自）

Predicate（p.116）、Adaptable Unary Function（p.114）。

➢ Associated Types（相关型别）

和定义于 Predicate 以及 Adaptable Unary Function 中的 associated types 相同。Adaptable Predicate 必须具备嵌套型别 result_type，后者必须可转换成 bool。

➢ Valid Expressions（有效表达式）

除了定义于 Predicate 和 Adaptable Unary Function 中的条件，没有别的了。

➢ Models

以下各个 STL function object class 都是 Adaptable Predicate。

```
logical_not<bool>
binder1st<equal_to<int> >
```

注意，函数指针并非 Adaptable Predicate 的一个 model，原因是函数指针并非 class，不具有嵌套型别（nested types）。

8.3.4 Adaptable Binary Predicate

Adaptable Binary Predicate 是一种 Binary Predicate，同时也是一种 Adaptable Binary Function。换句话说，它是一个 Binary Function，其返回值型别可转换为 `bool`，而且嵌套定义有 argument type 与 return type。

➢ Refinement Of（强化自）

Binary Predicate（p.117）、Adaptable Binary Function（p.115）。

➢ Associated Types（相关型别）

和定义于 Binary Predicate 以及 Adaptable Binary Function 中的 associated types 相同。Adaptable Binary Predicate 必须具备嵌套型别 `result_type`，后者必须可转换成 `bool`。

➢ Valid Expressions（有效表达式）

除了定义于 Binary Predicate 和 Adaptable Binary Function 中的条件，没有别的了。

➢ Models

STL 为 Adaptable Binary Predicate 定义了许多个 model classes；这些 classes 能够测试两数之间的某种关系。下面是 STL 所定义的一些 Adaptable Binary Predicate classes：

```
less<T>
greater<T>
equal_to<T>
logical_and<T>
```

8.3.5 Strict Weak Ordering

Strict Weak Ordering 是一种 Binary Predicate，可用来比较两个 objects：如果第一个 object 位于第二个 object 之前，就返回 `true`。这个 predicate 必须满足 *strict weak ordering* 的标准定义。以下正是其精确的条件描述，大略意思是：Strict Weak Ordering 必须满足传统的（例如 less than）次序（ordering）关系。举个例子，如果 `a` 小于 `b`，则 `b` 必定不小于 `a`。

Strict Weak Ordering 和 Strict Weakly Comparable 这两个 concepts 都和 strict weak ordering 有关，只是形式稍微不同。其中一个是「以 `operator<` 定义出来的次序条件」所形成的集合，另一个则是「以

某个 function object 定义出来的次序条件」所形成的集合。

这两个 concepts 互补，因为凡使用次序关系的算法，通常都会成对出现，其中一个版本使用 `operator<` 来比较数值，另一个版本使用外界提供的 function object 作为评量标准。这种情况下，第一版本使用 Strict Weakly Comparable，第二版本使用 Strict Weak Ordering。

➢ Refinement Of（强化自）

Binary Predicate（p.117）。

➢ Associated Types（相关型别）

Strict Weak Ordering 具有与 Binary Predicate 相同的 associated types，但有一个额外限制：first argument type 和 second argument type 必须相同。

➢ Notation（符号）

`F`	某型别，是 Strict Weak Ordering 的一个 model。
`X`	`F` 之 argument type。
`f`	型别 `F` 的 object。
`x`	型别 `X` 的 object。
`y`	型别 `X` 的 object。

➢ Definitions（定义）

- 如果 `f(x,y)` 和 `f(y,x)` 的结果都是 `false`，则 `x` 与 `y` 等价。注意，object 永远等价于其自身（根据稍后出现的恒长特性 irreflexivity）。

➢ Valid Expressions（有效表达式）

- Function call（函数调用）

  `f(x,y)`

返回型别：	任何可转换为 `bool` 的型别。
先决条件：	`(x,y)` 必须在 `f` 的 "domain" 之中。
语义：	如果 `x` 位于 `y` 之前，就返回 `true`，反之返回 `false`。
必然结果：	返回值不是 `true` 就是 `false`。

➢ Invariants（恒长特性）

任何表达式如果涉及 Strict Weak Ordering 数值，必须满足以下性质：

- Irreflexivity（不自反性）
  `f(x, x)` 一定是 `false`。
- Antisymmetry（非对称性）
  `f(x, y)` 意味 `!f(y, x)`。
- Transitivity（传递性）
  `f(x, y)` 且 `f(y, z)`，意味 `f(x, z)`。
- Transitivity of equivalence（等价性的传递）
  等价性（equivalence，如先前所定义）具有传递性质。如果 `x` 等价于 `y` 且 `y` 等价于 `z`，那么 `x` 等价于 `z`。这意味此处的等价性满足数学上的定义。

前三个公理（axioms）：irreflexivity、antisymmetry 和 transitivity，加起来便是 *partial ordering* 的定义。至于 *strict weak ordering*，还需加上 transitivity of equivalence。如果是 *total ordering*，必须满足更严格的条件：等价（equivalence）和相等（equality）一致。

Total ordering 的定义十分简单。但尽管如此，没有任何一个 STL 算法使用 total ordering。Total ordering 的条件过于严格。现实应用中，身为 strict weak ordering 但不为 total ordering 的情况很常见，最常见的便是无大小写之分的字符串比对。

➢ Models

以下型别都是 Strict Weak Ordering 的实例：

```
less<int>
greater<int>
less<string>
greater<string>
```

`less` 和 `greater` 这两个 function obejcts，是 Strict Weak Ordering 和 Strict Weakly Comparable 这两个 concepts 的桥梁（link）。当且仅当（if and only if）`T` 是 Strict Weakly Comparable 的一个 model，那么 `less<T>` 和 `greater<T>` 这两个 classes 便是 Strict Weak Ordering 的 model。

注意，`less_equal<T>` 和 `greater_equal<T>` 并非 Strict Weak Ordering 的 model。strict weak ordering 是一种「类似 less than，而非类似 less than or equal to」的次序关系。Strict Weak Ordering `f` 必须满足「`f(x, x)` 必为 `false`」的性质。

8.4 特化的 Concepts

8.4.1 Random Number Generator

Random Number Generator 是一种 function object，可用来产生整数随机数列。如果 f 是一个 Random Number Generator 并且 N 是个正数，则 f(N) 返回一个小于 N 且大于等于 0 的整数。如果以同一个 N 多次调用 f，产生的数列在 range [0,N) 中将是均匀分布（uniformly distributed）[16]。

random number generator（乱数产生器）是个微妙的主题。一个好的 random number generator 必须满足均匀分布（uniform distribution）以外的许多统计学性质。请参考 Knuth 的著作 [Knu98a] 3.4 节，其中有乱数数列的讨论，该书 3.2 节并提供许多用来撰写 random number generator 的算法。

➢ Refinement Of（强化自）

Unary Function（p.111）。

➢ Associated Types（相关型别）

- Argument type
  Random Number Generator 的引数型别。必须是个整数（integral）型别。
- Result type
  Random Number Generator 被调用后，返回值的型别。必须和 argument type 相同。

➢ Notation（符号）

F	某型别，是 Random Number Generator 的一个 model。
Integer	F 的 argument type。
f	型别 F 的 object。
N	型别 Integer 的 object。

➢ Definitions（定义）

1. Random Number Generator 的 "domain"，亦即有资格作为其引数的所有数值集合，是「大于零且小于某个最大值」的数值集合。
2. Random Number Generator 的 "range"，是小于 Random Number Generator 引数的一个非负整数集合。

---

[16] 均匀分布（uniform distribution）意指 range [0,N) 中的所有数值，其出现频率皆相同。任何特定值被取出的几率是 1/N。

➢ Invariants（恒长特性）

凡表达式涉及 Random Number Generator 数值，必须满足以下性质：

- Uniformity（均匀分布）
  如果以同一引数 N 多次调用 f，range [0,N) 中的每个整数出现次数将会相同。

8.4.2 Hash Function（散列函数）

Hash Function 是一种 Unary Function，用在 Hashed Associative Container（p.161）中。它接受单一引数，并将该引数映射至型别为 `size_t` 的结果。Hash Function 必须具备注定性（deterministic），意味其返回值必须只和其引数有关，而且同一个引数会导致相同的结果。

Hashed Associative Container 的性能主要决定于其 hash function。对 Hash Function 而言，将碰撞（collisions）减到最小是很重要的，而碰撞的定义是，两个不同的引数导出（*hashing*）相同的结果。令 hash value 均匀分布也很重要。Hash Function 返回型别为 `size_t` 之特定值的几率，必须约略等于返回其他值的几率。Hash Function 的执行速度必须快，这一点也很重要。很自然地，这三者之间存在某种紧张关系。

「均匀分布」与「碰撞最小化」，其目的只有在输入值之某种特定分布下才合理。举个例子，假设被 "hashed" 的数值是六个字符串：`"aardvark"`、`"trombone"`、`"history"`、`"diamond"`、`"forthright"`、`"reasonable"`。这种情况下，每个字符串的开头字符将是一个合理（且快速）的 hash function。但如果六个字符串是：`"aaa1001"`、`"aaa1010"`、`"aaa0011"`、`"aaa1100"`、`"aaa0101"`、`"aaa0110"`，这种情况下改用不同的 hash function 会比较妥当。这就是为什么 Hashed Associative Container 要把 hash function 当作参数的原因：没有一个 hash function 可适用于每一种情况。

请注意，Hash Function 并不知道 Hashed Associative Container 所使用的「bucket（桶子）」个数。Hash Function 只负责将其引数对映成某数 — 或许是个很大的数。将那个大数转成特定 bucket，是 Hashed Associative Container 的责任。

➢ Refinement Of（强化自）

Unary Function（p.111）。

➢ Associated Types（相关型别）

和 Unary Function 所定义的相同。只有一个额外限制：result type 必须为 `size_t`。

➢ Invariants（恒长特性）

除了 Unary Function 的恒长特性外，还必须满足以下性质：

- Deterministic function（注定性函数）
  返回值仅取决于引数，与所经历之 Hash Function object 无关。任何特定之 Hash Function object，如果引数相同，返回值便相同。

➢ Models

STL 唯一定义的一个 Hash Function 是 class `hash`（p.400）。

第 9 章

Containers
容器

所谓 container 是一种 object，可以包含并管理其他 objects，并提供 iterators 用以定址其所包含之 objects（元素）。

STL 的 container concepts 可归属为三种广泛类型。第一种是极为一般化的 Container concept，以及另三个稍微不那么一般化的 concepts，它们系以其所提供的 iterators 型别作为分类依据。STL 的所有 containers 都是 Container concept 的 models。第二种是 Sequence concept，这是一种可动态调整大小的 container，可以在任何位置安插或删除元素。第三种是 Associative Container concept，特别针对「以值查询元素」的情况下进行优化。

某些 containers 以其内存分配方式作为参数，其中很多使用 Allocator concept。Allocator 并不是 Container 的强化（refinement），但因为它与 container 的关系密切，所以本章也将它纳入讨论。

9.1 General Container Concepts

9.1.1 Container

所谓 Container 是一种 object，能够存储其他的 objects 作为元素，并拥有访问元素的方法。更明确地说，每一个 Container model 都具有一个相关的 iterator 型别，可用来遍历元素。通过 iterators 所形成的 range，可以访问 Container 中的所有元素。

我们可根据 Container concept 相关的 iterator 型别加以分类。Container 是最一般化的 container concept，它只要求其相关的 iterator 型别必须是 Input Iterator 的一个 model。因此，你不应该对 Container 的 iterators 作出比 Input Iterators 还多的假设。

Container 的元素并不保证以任何明确次序来存储 — 事实上其次序视遍历方式的不同而有所不同。Container 也不保证在同一时间内有一个以上的 iterators 有效（active）。其他的 container concepts 如 Forward Container（p.131）则可能提供更强烈的保证。

Container「拥有」其元素：存储于某 container 内的元素，其生命不会超过 Container 自身的生命。这个限制看似严苛，其实不然。如果你需要更宽松的拥有权（ownership）意义，你可以利用 container 存储指针。毕竟指针如同其他 objects 一样，自身也是一种 object，可以放进 Container 之中。Container 仍然拥有其元素，也就是指针自身，但不拥有其所指之物。

➢ Refinement Of（强化自）

Assignable（p.83）。

➢ Associated Types（相关型别）

- Value type

  `X::value_type`

  存储于 Container 内之 object 型别。此型别必须是 Assignable，但不必是 Default Constructible。

- Reference type

  `X::reference`

  此型别如同「Container 之 value type」的 reference 一样。通常 reference type 就是 `value_type&`。（或许你会想到所谓的 "smart reference"，那是一种使用者定义的 reference 型别，提供某种额外功能。通常这不是一个好的选择，因为使用者自定型别不可能具有与 C++ reference 相同的语义。C++ 并不允许你重新定义成员访问操作符 `operator`。）

- Const reference type

  `X::const_reference`

  此型别如同「Container 之 value type」的 `const` reference。通常它就是 `const value_type&`。

- Pointer type

  `X::pointer`

  此型别如同「Container 之 value type」的指针。pointer type 通常是 `value_type*`。（pointer type 必须具有与一般 C 指针相同的语义，但它不一定得是个指针。和前述的 smart reference 不同，smart pointer 是可行的方案。C++ 允许使用者自定型别自行定义 `operator*` 和 `operator->`，亦即提领操作符和指针成员访问操作符。）

- Const pointer type

  `X::const_pointer`

  此型别如同「Container 之 value type」的 `const` pointer。通常它就是 `const value_type*`。

- Iterator type

  `X::iterator`

  所谓 iterator 系用来指向 Container 的元素。其 value type 应该是 Container 的 value type，而其 reference type 以及 pointer type 也应该和 Container 的 reference type 和 pointer type 相同。Iterator type 必须是 Input Iterator（p.94）的一个 model。

  注意，iterator type 只需要是个 Input Iterator 就行，这提供较弱的保证；更明确地说，所有作用在 Input Iterator 身上的算法必须是单回的（single pass）。Forward Container concept 提供有较强的保证，其 iterator type 为 Forward Iterator 的 model。

- Constant iterator type

  `X::const_iterator`

  所谓 constant iterator 系用来指向 Container 的元素。其 value type 应该是 Container 的 value type，而其 reference type 和 pointer type 也应该和 Container 的 `const` reference type 及 `const` pointer type 相同。和 iterator type 一样，constant iterator type 也必须是 Input Iterator 的一个 model。iterator type 和 constant iterator type 拥有相同的 difference type 与 iterator category，并且必须存在一个转换函数，可以将 iterator type 转换为 constant iterator type。

  注意，Container 的 iterator type 与 constant iterator type 可能相同。因为并非每一个 Container 都需要相关的 mutable iterator type。以 `set` 和 `hash_set` 为例，它们就没有提供 mutable iterator。

  如果 `iterator` 和 `const_iterator` 是相同的型别，那么 `reference` 和 `const_reference` 以及 `pointer` 和 `const_pointer` 也必须是相同的型别。

- Difference type

  `X::difference_type`

  此为一个带正负号的整数（integral）型别，用以表示两个 Container iterators 之间的差距。它必须与 iterator 的 difference type 相同。

- Size type

  `X::size_type`

  此为一个不带正负号的整数（integral）型别，用以表示 Container's difference type 的任何一个非负值。

➢ Notation（符号）

`X`	某型别，是 Container 的一个 model。
`a,b`	型别 `X` 的 object。
`T`	`X` 的 value type。

➢ Definitions（定义）

1. Container 的 "*size*" 是指其元素个数。这是一个非负数。

2. Container 的 "*area*" 是指其所占用的 bytes 总数。更精确地说是指「所有元素所占空间」加上「Container 自身所耗用的额外空间」的 bytes 总数。如果某个 Container 的 value type `T` 是个单纯型别（而非 container type），则这个 Container 的 area 将是某个数值（Container 自身所耗用的额外空间；可能随 container size 变动）加上「container size 和 `sizeof(T)` 的乘积」。也就是说，如果 `a` 这个 container 具有单纯的 value type，那么 `a` 的 area 是 ***O(a.size())***。

3. 所谓「可变大小（variable-sized）」的 container，是一种可以安插和删除元素的 container。其 size 可能会在生命过程中有所变动。所谓「固定大小（fixed-size）」的 container，其 size 在生命过程中不会改变，而是在编译期就定下来了。

➢ Valid Expressions（有效表达式）

除了定义于 Assignable 的表达式外，以下表达式也必须有效：

- Copy Constructor

  `X(a)`

  返回型别： `X`
  必然结果： `X().size() == a.size()`。`X()` 内含 `a` 的每个元素的复制品。

- Copy Constructor

  `X b(a);`

  必然结果： `b.size() == a.size()`。`b` 内含 `a` 的每个元素的复制品。

- Assignment operator

  `b = a`

  型别条件： `b` 必须是 mutable。
  返回型别： `X&`
  必然结果： `b.size() == a.size()`。`b` 内含 `a` 的每个元素的复制品。

- Destructor

  `a.~X()`

  语义： `a` 的每一个元素的 destructor 都会被调用，然后归还内存。

- Range 起始点

  `a.begin()`

返回型别： 如果 a 是可变的（mutable），返回 iterator；否则返回 const_iterator。
语义： 返回一个 iterator，指向 container 的第一个元素。
必然结果： a.begin() 如果不是 dereferencealbe（可提领的）便是 past-the-end，后者只在 a.size() == 0 时才成立。

- Range 尾端

a.end()

返回型别： 如果 a 是可变的（mutable），返回 iterator；否则返回 const_iterator。
语义： 返回一个 iterator，指向 container 的 past-the-end 处。
必然结果： a.end() 是 past-the-end。只有当 a.size()==0 时，a.begin() 和 a.end() 才会相等。

- Size（容器大小）

a.size()

返回型别： size_type
语义： 返回 container a 的大小，也就是 a 所含的元素个数。
必然结果： 0 <= a.size() <= a.max_size()。

- Maximum size（容器的最大可能容量）

a.max_size()

返回型别： size_type
语义： 返回 container 的大小极限。请注意，max_size 会返回某个上限值，但不必然是个精确上限值，也就是说它并不保证你可以产生某个型别为 x 的 container 并令其 size 为 a.max_size()，它只保证你不能开启一个大小超过 a.max_size() 的 container。
必然结果： 0 <= a.size() <= a.max_size()。

- Empty（判断容器是否为空）

a.empty()

返回型别： 任何可转换为 bool 的型别。
语义： 等价于 a.size() == 0，但评估速度有可能更快。此 member function 的存在原因之一是，a.size()的复杂度只需满足 *O(N)* 即可，而 empty 却保证为 *O(1)*。

- Swap（交换）

a.swap(b)

型别条件： a 与 b 皆是可变的（mutable）。

返回型别： void
语义： 等价于：

```
X tmp = a;
a = b;
b = tmp;
```

不过几乎一定比上述做法更快。如果 a 和 b 皆拥有 N 个元素，交换这两个 container 会用到 3N 个 operator= 赋值动作。不过通常我们可以将 Container 的 swap() 复杂度做到只为 *O(1)*，而非 *O(N)*。

➢ 复杂度保证（Complexity Guarantees）

1. copy constructor、assignment operator 和 destructor，其复杂度与 container 的大小成线性关系。
2. begin() 和 end() 耗用 amortized constant time。
3. size() 的复杂度和 container 的大小成线性关系。对很多 containers 如 vector 和 deque 而言，size() 的复杂度为 *O(1)* — 这满足「至少达到 *O(N)*」的条件。
4. max_size() 和 empty() 耗用 amortized constant time。
5. swap() 的复杂度与 container 的大小成线性关系。然而，对大多数 container 而言，它会耗用 amortized constant time。它应该被定义为 *O(1)*，除非有强烈的理由说明为何这样做不可行。

➢ Invariants（恒长特性）

凡表达式涉及 Container 者，必须满足以下性质：

- Valid range（有效区间）

  对于 Container a，[a.begin(),a.end()) 是一个有效区间。然而请注意，唯一条件是这必须是一个由 Input Iterators 形成的区间，而且区间内的元素次序并未指定。Forward Container concept 提供的是 Forward Iterator，并导入其他的次序保证。

- Range size（区间大小）

  a.size() == distance(a.begin(), a.end())。

- Completeness（完全性）

  一个算法如果遍历 [a.begin(),a.end()]，会正好经过每个元素一次。

➢ Models

STL 的每个 container class 都是 Container 的 model。这些 classes 包括 vector、slist、map 等等。

事实上，STL 的所有 container classes 同样也是 Forward Container 的 model。STL 并没有定义任何「仅仅为 Container 之 model」的 class。「身为 Container 之 model 但非 Forward Container 之 model」的实例是所谓「有自我调整能力的数组（self-adjusting array）」。如果以 Knuth 的术语 [Knu98b] "self-organizing file" 来说明便是：一种「会自动地将被频繁需求之元素移至前头」的一种 class。

9.1.2 Forward Container

所谓 Forward Container 是一种 Container，其元素依据明确的次序规则来排列。这个次序规则不会在迭代过程中自行改变。明确的次序规则保证元素逐一相等 — 如果 container 的元素型别是 Equality Comparable 的话，也保证字典排列次序的成功 — 如果 container 之元素型别是 LessThan Comparable 的话。

Forward Container 所提供的 iterators 满足 Forward Iterator 的需求条件。因此 Forward Container 支持多回（multipass）算法，并允许同一个 container 同时拥有多个有效的（active）iterators。

➢ Refinement Of（强化自）

Container（p.125）、Equality Comparable（p.85）、LessThan Comparable（p.86）。

➢ Associated Types（相关型别）

除了定义于 Container 的那些，再没有额外的 associated types 了。不过倒是加强了 iterator type 的条件：iterator type 必须是 Forward Iterator（p.100）的一个 model。

➢ Notation（符号）

X	某型别，是 Forward Container 的一个 model。
a,b	型别 X 之 object。
T	X 的 value type。

➢ Valid Expressions（有效表达式）

除了定义于 Container 的表达式外，以下表达式也必须有效：

- Equality（相等性）

```
a == b
```

型别条件： T 是 Equality Comparable。
返回型别： 任何可转换为 bool 的型别。
语义： 如果 a.size() == b.size() 而且 a 的每一个元素等于 b 的每一个相对元素，就返回 true，否则返回 false。

- Inequality（不相等性）

  `a != b`

  型别条件： T 是 Equality Comparable。
  返回型别： 任何可转换为 bool 的型别。

- Less（小于）

  `a < b`

  型别条件： T 是 LessThan Comparable。
  返回型别： 任何可转换为 bool 的型别。
  语义： 等价于：

  ```
  lexicographical_compare(a.begin(), a.end(),
                          b.begin(), b.end())
  ```

- Greater（大于）

  `a > b`

  型别条件： T 是 LessThan Comparable。
  返回型别： 任何可转换为 bool 的型别。

- Less or equal（小于或等于）

  `a <= b`

  型别条件： T 是 LessThan Comparable。
  返回型别： 任何可转换为 bool 的型别。

- Greater or equal（大于或等于）

  `a >= b`

  型别条件： T 是 LessThan Comparable。
  返回型别： 任何可转换为 bool 的型别。

➢ **复杂度保证**（Complexity Guarantees）

各种比较行为（comparison）均与 container 的大小成线性关系。

➢ Invariants（**恒长特性**）

除了定义于 Container 中的恒长特性外，还必须满足以下性质：

- Ordering（有序性）

  在 Forward Container 中，不同的两个迭代动作（iterations）会以相同的次序访问元素 — 前提是两个迭代动作之间没有执行任何「会造成 container 内容改变」的操作。

➢ Models

所有 STL container classes 都是 Forward Container 的 models。以下是一些例子：

```
vector
list
slist
deque
set
hash_set
map
hash_map
multiset
hash_multiset
multimap
hash_multimap
```

9.1.3 Reversible Container

所谓 Reversible Container 是一种 Forward Container，而且其 iterator 是一种 Bidirectional Iterator。它拥有额外的一些 member functions 和嵌套型别（nested types），不但可以向前遍历（forward iteration），也可以回头遍历（backward iteration）。

➢ Refinement Of（强化自）

Forward Container（p.131）。

➢ Associated Types（相关型别）

这个 concept 导入两个新型别。此外 iterator type 和 constant iterator type 必须满足比 Forward Container 更为严格的条件。Iterator type 和 reverse iterator type 不仅必须是 Forward Iterator，还必须是 Bidirectional Iterator。

- Reverse iterator type

  `X::reverse_iterator`

  一个 base iterator type 为 `X::iterator` 的 reverse iterator adapter（p.363）。将一个型别为 `reverse_iterator` 的 object 做累加动作，便是在该 container 中往回移动。reverse iterator adapter 会将 `operator++` 映射成 `operator--`。

- Constant reverse iterator type

  `X::const_reverse_iterator`

  一个 base iterator type 为 `X::const_iterator` 的 reverse iterator adapter。注意，Container 的 iterator type 和 constant iterator type 可能相同。Container 并不一定得提供 mutable iterators。

同样道理，reverse iterator type 和 constant reverse iterator 也可能相同。

➢ Notation（符号）

`X` 某型别，是 Reversible Container 的一个 model。
`a,b` 型别 `X` 的 object。

➢ Valid Expressions（有效表达式）

除了定义于 Forward Container 的表达式外，以下表达式也必须有效：

- Range 起始点

  `a.rbegin()`

  语义： 等价于 `X::reverse_iterator(a.end())`。
  必然结果： `a.rbegin()` 必然是 dereferenceable 或 past-the-end。当且仅当 `a.size() == 0`，则 `a.rbegin()` 为 past-the-end。

- Range 尾端

  `a.rend()`

  语义： 等价于 `X::reverse_iterator(a.begin())`。
  必然结果： `a.rend()` 是 past-the-end。

➢ 复杂度保证（Complexity Guarantees）

`rbegin()` 和 `rend()` 的运行期复杂度为 amortized constant time。

➢ Invariants（恒长特性）

除了定义于 Forward Container 中的恒长特性外，以下性质也必须满足：

- Valid range（有效区间）

  `[a.rbegin(),a.rend())` 是一个有效区间。

- Equivalence of ranges（区间等价性）

  从 `a.begin()` 到 `a.end()` 的距离和从 `a.rbegin()` 到 `a.rend()` 的距离相等，而且两个区间内的元素都相同。

➢ Models

以下是 Reversible Container 的一些实例：

```
vector<double>
list<string>
set<int>
```

9.1.4 Random Access Container

所谓 Random Access Container 是一种 Reversible Container，而且其 iterator type 是一种 Random Access Iterator。它提供「在 amortized constant time 访问任意元素」的方法，可由 iterator 或直接由 `operator[]` member function 访问。

➢ Refinement Of（强化自）

Reversible Container（p.133）。

➢ Associated Types（相关型别）

除了定义于 Reversible Container 的 associated types 外，就没有别的了，不过 iterator types 的条件更为严格：它必须是 Random Access Iterator。

➢ Notation（符号）

`X`	某型别，是 Random Access Container 的一个 model。
`a,b`	型别 `X` 之 object。
`T`	`X` 之 value type。

➢ Valid Expressions（有效表达式）

● Element access（元素访问）

`a[n]`

型别条件：	任何可转换为 `size_type` 的型别。
返回型别：	若 `a` 可变（mutable），返回 `reference`，反之返回 `const_reference`。
先决条件：	`0 <= n < a.size()`。
语义：	返回 container 开头算起的第 `n` 个元素。`a[n]` 返回的元素与「将 `a.begin()` 累加 `n` 次后」提领而得的元素相同。

➢ 复杂度保证（Complexity Guarantees）

元素访问动作的运行期复杂度是 amortized constant time。

➢ Models

STL containers 中的 `vector` 和 `deque` 都是 Random Access Container 的 models，5.1 节的 `block` class 亦然。相较之下，像 `list` 与 `slist` 这般连接链表并非是 Random Access Container 的 model，它们不具备 `operator[]` member function，其 iterators 也不是 Random Access Iterators。

9.2 Sequence（序列；循序式容器）

9.2.1 Sequence

所谓 Sequence 是一种可变大小（variable-sized）的 Container，其元素系以 strict linear order（严格线性次序）加以排列。Sequence concept 导入许多新的 member functions，用以安插或删除任意位置上的单个元素或某区间内的元素。

➢ Refinement Of（强化自）

Forward Container（p.131）、Default Constructible（p.84）。

➢ Associated Types（相关型别）

仅止于定义于 Forward Container 中的 associated types。

➢ Notation（符号）

`X`	某型别，是 Sequence 的一个 model。
`a,b`	型别 `X` 之 object。
`T`	`X` 之 value type。
`t`	型别 `T` 之 object。
`p,q`	型别 `X::iterator` 之 object。
`n`	可转换成 `X::size_type` 之 object。

➢ Definitions（定义）

1. 如果 `a` 是个 Sequence，并且如果 `p` 是个有效的 iterator 并可由 `a.begin()` 前进到达，那么 `p` 是 `a` 中有效的 iterator。
2. 如果 `a` 是个 Sequence，并且如果 `[p,q)` 是个有效的 range，而且 `p` 是 `a` 中一个有效的 iterator，那么 `[p,q)` 是 `a` 中有效的 range。

➢ Valid Expressions（有效表达式）

除了定义于 Forward Container 的表达式外，以下表达式也必须有效：

- Fill constructor（充填构造函数）

  `X(n, t)`

  返回型别： `X`
  先决条件： `n >= 0`。
  语义： 以 `n` 个「`t` 复制品」产生一个 sequence。
  必然结果： Sequence 大小等于 `n`。Sequence 中的每个元素都是 `t` 的复制品。

● Fill constructor（充填构造函数）

`X(n)`

型别条件： `T` 是 Default Constructible。
返回型别： `X`
先决条件： `n >= 0`。
语义： 等价于 `X(n, T())`，也就是以 `n` 个「初始值为缺省值」的元素产生出一个 Sequence。
必然结果： Sequence 的元素个数等于 `n`。Sequence 中每个元素皆是 `T()` 的复制品。

● Default constructor（缺省构造函数）

`X()`

返回型别： `X`
语义： 等价于 `X(0, t)`。也就是产生一个不含任何元素的 sequence。
必然结果： Sequence 的元素个数为零。

● Range constructor（区间构造函数）

`X(i, j)`

型别条件： `i` 和 `j` 是 Input Iterators，其 value type 可转换为 `T`。
返回型别： `X`
先决条件： `[i,j)` 是有效的 range。
语义： 产生出一个涵盖「range `[i,j)` 之复制品」的 sequence。
必然结果： 产生出来的 sequence 的元素个数等于 `distance(i, j)`。Sequence 中的每个元素等于 input range 内的相对元素。

● Insert（安插）

`a.insert(p, t)`

型别条件： a 是可变的（mutable）。
返回型别： `X::iterator`
先决条件： `p` 是 `a` 中有效的 iterator。`a.size() < a.max_size()`。
语义： 将 `t` 复制品安插于 `p` 之前。有两个重要的特案：`a.insert(a.begin(), t)` 将 `t` 安插于开头（第一个元素之前），`a.insert(a.end(), t)` 将 `t` 附加于尾端。
必然结果： `a.size()` 将增加 1。`*(a.insert(p,t))` 是一个 `t` 复制品。Sequence 中既有元素的相对次序不变。警告：此必然结果并非表示 `a` 之中的某个有效的 iterator 在安插与删除之后还保证有效。举例来说，它就不保证 `p` 仍为有效。某种情况下 iterators 可能仍然有效并继续指向原先的元素。其他情况下，将元素安插于 sequence 内可能会使既有元素重新排列，以便获取空间；先前指向该 sequence 的 iterators 可能会因此指向与原先不同的元素。其间细节根据不同的 sequence class 而有所不同。

● Fill Insert（安插并充填）

```
a.insert(p, n, t)
```

型别条件： a 是可变的（mutable）。
返回型别： void
先决条件： p 是 a 中一个有效的 iterator。n >= 0。a.size() + n <= a.max_size()。
语义： 将 n 个 t 复制品安插于 p 之前。它保证不会比调用 insert(p, t) n 次来得慢，某些情况下可能会快上许多。
必然结果： a.size() 将增加 n。Sequence 内既有元素的相对次序不变。

● Range Insert（安插并充填一个区间）

```
a.insert(p, i, j)
```

型别条件： a 可变。i 和 j 都是 Input Iterators，其 value type 可转换为 T。
返回型别： void
先决条件： p 是 a 中有效的 iterator。[i,j) 是有效的 range。a.size() + distance(i, j) <= a.max_size()。
语义： 将 range [i,j) 之复制品安插于 p 之前。
必然结果： a.size() 将增加 distance(i,j)。Sequence 内既有元素的相对次序不变。

● Erase（抹除）

```
a.erase(p)
```

型别条件： a 是可变的（mutable）。
返回型别： iterator
先决条件： p 是 a 中的一个 dereferenceable iterator。
语义： 将 p 所指的元素析构掉，并将它从 sequence 中移除。返回值是个 iterator，指向被移除元素的下一个紧邻元素 — 如果没有这样的元素则为返回 a.end()。
必然结果： a.size() 将减少 1。Sequence 内其他元素的相对次序不变。注意，就像 insert 一样，erase 可能会造成该 sequence 的 iterators 不再有效。当然，a.erase(p) 会影响 p 自身，但可能还会影响其他 iterators。和 insert 一样，erase 可能涉及 sequence 元素的重新排列。这个事实表示，你不该假设 a.erase(p) 的返回值是 p+1。

● Range erase（抹除某个区间）

`a.erase(p, q)`

型别条件： a 是可变的（mutable）。
返回型别： iterator
先决条件： [p,q) 是 a 中有效的 range。
语义： 将 range [p,q) 中的所有元素析构掉，并将它们从 sequence 中移除。返回值是个 iterator，指向被移除的元素群的下一个紧邻元素，如果没有这样的元素，返回值将是 a.end()。（如同单元素版本的 erase 一样，你不应该假设返回值一定为 q。）
必然结果： a.size() 将减少 distance(p, q)。Sequence 中其他元素的相对顺序不变。

● Front（返回开头元素）

`a.front()`

返回型别： 如果 a 是可变的，返回 reference，否则返回 const_reference。
先决条件： !a.empty()。
语义： 等价于 *(a.begin())。

其中有两个表达式涉及一般性 Input Iterators 所形成的 ranges，所以（举个例子），你可以将 list<> 内的所有元素安插到 vector<> 内，像这样：

```
V.insert(V.begin(), L.begin(), L.end())
```

由于这对于任何 Input Iterator types 都必须成立，所以 insert 必须是个 member template，换句话说它必须是一个 template 形式的 member function。

显然，解决之道是当你定义 Sequence class 时，同时定义两个不同版本的 insert，一个是一般的 member function，可安插 n 个 t 复制品，另一个是 member template，可安插 range [i,j)：

```
template <class T>
class my_sequence {
  ...
public:
  void insert(iterator p, size_type n, const value_type& t);
  template <class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
};
```

基本上这是正确的概念，但技术细节就有点复杂了。不过，当你意图实现新的 Sequence 时，你才需要担心这些细节。

假设你要处理一个「value type 为 int」的 Sequence，例如 vector<int>，现在假设你写：

```
V.insert(V.begin(), 100, 0);
```

意图安插 100 个 0 的复制品。这是 Sequence 允许发生的事，事实上这是 insert 函数的 "fill" 版。不幸的是，如何实现一个 Sequence 使上述行为得以正确无误地运作，却有点棘手。面对先前所定义的两个 insert 版本，C++ 编译器会选择第二个版本而非第一个。

原因是 100 和 0 都属于型别 int。第一版本要求其第二参数的型别为 size_t，本例不符需求。第二版本的两个引数型别必须相同，本例能够符合。基本上问题在于 C++ 编译器不知道所谓的 concepts。insert 的「range 版本」虽然拥有名为 InputIterator 的 template 参数型别，但该名称在此并无特别意义。编译器并不知道 insert 的引数必须是个 Input Iterator。

有两个解决办法。第一，你可以以更多的型别来重载（overload）它。除了下面这个形式：

```
void insert(iterator, size_t, const value_type&);
```

针对各种整数型别，你还可以定义：

```
void insert(iterator, int, const value_type&);
void insert(iterator, long, const value_type&);
```

这很直观，却索然无味。第二个办法是，你可以利用 type dispatching，这也是某些算法所采用的技巧。检查 insert 的引数是否为整数，如果是，就不采用其「range 版本」。

Standard C++ library 中的 numeric_limits class，可让你检查某个型别是否为整数。它会返回一个 bool。藉由所谓的 "dummy class"（而且彼等拥有一个 "nontype" template 参数），你可以将该 bool 值转换为一个型别。

```
template <bool x> struct dummy { };  // 拥有一个 nontype template parameter

template <class T>
class my_sequence {
  ...
private:
  void fill_insert(iterator, size_type, const value_type&);

  template <class InputIter>
  void range_insert(iterator, InputIter, InputIter);

  template <class Number>
  void insert(iterator p, Number n, Number t, dummy<true>) {
    fill_insert(p, n, t);
  }
```

```
  template <class InputIter>
  void insert(iterator p, InputIter first, InputIter last, dummy<false>) {
    range_insert(p, first, last);
  }

public:
  void insert(iterator p, size_type n, const value_type& t) {
    fill_insert(p, n, t);
  }

  template <class InputIter>
  void insert(iterator p, InputIter first, InputIter last) {
    insert(p, first, last,
           dummy<std::numeric_limits<InputIter>::is_integer>());
  }
};
```

➢ **复杂度保证**（Complexity Guarantees）

1. fill constructor、default constructor 和 range constructor 都是线性复杂度。

2. 单元素版的 insert 和 erase，其运行期复杂度取决于 sequence。

3. 多元素版的 insert 与 erase，其复杂度是线性的。

4. front 耗用 amortized constant time。

➢ Models

STL containers 如 vector、deque、list、slist，都是 Sequence 的 model。它们之间的差别主要在于其 iterators（vector 和 deque 提供 Random Access Iterators，list 提供 Bidirectional Iterators，slist 提供 Forward Iterators），以及 insert/erase 的运行期复杂度。

vector 通常是最好的选择。它是 STL constainers 中最为简单的一个，因此对于 iterators 以及元素的访问，具有最小的时间（速度）负担。然而，由于「将元素安插至 vector」的复杂度与「安插点至尾端的元素个数」成线性关系，如果安插动作频率，其他 sequences 可能比较适合。如果你常在开头或尾端执行许多安插动作，deque 可能是个好选择，如果你常在 sequence 两端之间执行许多安插动作，那么 list 或 slist 可能比较理想。

9.2.2 Front Insertion Sequence

所谓 Front Insertion Sequence 是一种 Sequence，可以在 amortized constant time 内将元素安插于起始点或是取得起始位置的元素。Front Insertion Sequence 拥有一些特别的 member functions 以实现上述行为。

➢ Refinement Of（强化自）

Sequence（p.136）。

➢ Associated Types（相关型别）

和定义于 Sequence 的相同。

➢ Notation（符号）

X	某型别，是 Front Insertion Sequence 的一个 model。
a	型别 X 的 object。
T	X 之 value type。
t	型别 T 的 object。

➢ Valid Expressions（有效表达式）

除了定义于 Sequence 的表达式外，以下表达式也必须有效：

- Front（取出起始元素）[17]

  `a.front()`

  返回型别： 如果 a 是可变的（mutable）就返回 reference，否则返回 const_reference。
  先决条件： !a.empty()。
  语义： 等价于 *(a.begin())。

- Push front（前端推入）

  `a.push_front(t)`

  型别条件： a 是可变的（mutable）。
  返回型别： void
  语义： 等价于 a.insert(a.begin(), t)。

- Pop front（前端取出）

  `a.pop_front()`

  型别条件： a 是可变的（mutable）。
  返回型别： void
  先决条件： !a.empty()。
  语义： 等价于 a.erase(a.begin())。

你可能会好奇为什么 pop_front() 返回 void。数据结构教科书中不是往往定义所谓 pop 动作会自该数据结构中移除某值然后返回该值吗？为什么我们不那样定义 pop_front 呢？两个原因：第一是

[17] front 实际上定义于 Sequence 之中，因为它总是能够在 amortized constant time 内完成。这里重复定义并与 push_front 和 pop_front 并列，为的是能够更清楚地表示。

没有必要。如果你想审视你意图移除的值，只要先调用 front 即可。第二个原因是效率考量。是的，pop_front 不能返回一个 reference.（这一点和 front 不同），因为元素被移除后就无法 return by reference 了；而如果要以 return by value 取而代之，会涉及不必要且高代价的复制行为。

➢ 复杂度保证（Complexity Guarantees）

Front Insertion Sequence 所引进的三个操作行为：front、push_front、pop_front，都耗用 amortized constant time。

这正是将 Front Insertion Sequence 独立定义为一个 concept 的唯一原因，因为除了复杂度（效率）因素，这三个动作并没有比 begin、insert、erase 提供更多功能。并非所有 sequences 都必须定义这三个操作行为，但如果它们存在的话，保证会更有效率。

➢ Invariants（恒长特性）

除了 Sequence 所定义的恒长特性外，还必须满足以下性质：

- push 和 pop 的对称性（Symmetry of push and pop）
  push_front 之后紧接着做 pop_front 动作，结果将毫无影响（null operation）。

➢ Models

STL containers 中的 list、slist 和 deque 都是 Front Insertion Sequence 的 model，vector 就不是。vector 拥有 front，但不具备 push_front 和 pop_front。在 vector 的起始处进行安插或删除动作，不是一种 constant time 行为。

9.2.3 Back Insertion Sequence

所谓 Back Insertion Sequence 是一种 Sequence，可以在 amortized constant time 内将元素安插于尾端，或是取出最后元素。Back Insertion Sequence 拥有某些特别的 member functions，用以实现上述行为。

➢ Refinement Of（强化自）

Sequence（p.136）。

➢ Associated Types（相关型别）

仅止于定义于 Sequence 的型别。

➢ Notation（符号）

X	某型别，是 Back Insertion Sequence 的一个 model。
a	型别 X 的 object。
T	X 之 value type。
t	型别 T 的 object。

➢ Valid Expressions（有效表达式）

除了定义于 Sequence 的表达式外，以下表达式也必须有效：

- Back（取出最后一个元素）

  a.back()

  返回型别： 如果 a 可变（mutable）就返回 reference，否则返回 const_reference。
  先决条件： !a.empty()。
  语义： 等价于 *(--a.end())。

- Push back（尾端推入）

  a.push_back(t)

  型别条件： a 是可变的（mutable）。
  返回型别： void
  语义： 等价于 a.insert(a.end(), t)。

- Pop back（尾端取出）

  a.pop_back()

  型别条件： a 是可变的（mutable）。
  返回型别： void
  语义： 等价于 a.erase(a.end())。

如同 Front Insertion Sequence（p.141）中的 pop_front 一样，pop_back 返回 void 而非被移除之元素。如果让 pop_back 返回某值，就得耗用不必要的复制动作。是的，它不能返回一个 reference，因为该 reference 所代表的元素会被移除。

➢ 复杂度保证（Complexity Guarantees）

Back Insertion Sequence 所引进的三个操作行为，back、push_back、pop_back，都耗用 amortized constant time。

这正是 Back Insertion Sequence 独立被定义为一个 concept 的唯一原因，因为除了复杂度（效率）因素，这三个动作并没有比 begin、insert、erase 提供更多功能。并不是所有 sequences 都必须定义这三个操作行为，但如果它们存在的话，保证更有效率。

➢ Invariants（恒长特性）

除了 Sequence 所定义的恒长特性外，还必须满足以下性质：

- push 和 pop 的对称性（Symmetry of push and pop）
  push_back 之后紧接着做 pop_back 动作，结果将毫无影响（null operation）。

➢ Models

STL containers 中的 vector、list、deque 都是 Back Insertion Sequence 的 model。slist 就不是。slist 拥有 back，但不具备 push_back 和 pop_back，它是单向连接链表，供应的是 Forward Iterators 而非 Bidirectional Iterators，而且「访问 slist 的最后一个元素」并不是一种 constant time 行为。

9.3 Associative Containers（关联式容器）

9.3.1 Associative Container

所谓 Associative Container 是一种可变大小的 Container，它支持「以 key（键值）为基础，有效率地取出元素」的方法。它也提供安插与移除动作，但与 Sequence 不同的是，它并不允许将元素安插于特定位置。

不具有上述机制的原因是，在 Associative Container 中，元素的排列方式对 class 而言是一种不变性。以 Sorted Associative Container 为例，元素总是以递增次序来存储，而在 Hashed Associative Container 之中元素总是根据 hash function 来存储。因此，在任意位置上安插元素是不合理的。

如同所有 containers，Associative Container 内含型别为 value_type 的元素。此外每个元素具有一个型别为 key_type 的 key（键值）。在某些 Associative Containers 之中（例如 Simple Associative Containers），其 value type 与 key type 相同，元素自身就是 key。在其他 Associative Containers 中，key（键值）是 value（实值）的特定部分。由于元素的存储与其 key 有关，所以基本上每个元素的 key 是不能改变的。这暗示你一旦将元素放入 Simple Associative Container 内，就再也不能改变它了，而其他的 Associative Containers（例如 Pair Associative Container），其元素自身可变，但作为 key 的那一部分不能更改。这也意味 Associative Container 的 value type 并不一定得是 Assignable。

这个事实有一个重要的结论。由于 Associative Container 的元素不能任意更改，所以 Associative Container 可以不具有 mutable iterators。根据定义，mutable iterators 必须允许赋值动作（亦即支持表达式 *i = t），但这样会侵犯到 key 的不可变动性。

在 Simple Associative Container 之中，元素自身就是它自己的 key，因此嵌套型别 `iterator` 和 `const_iterator` 在功能上完全相同（而且可能有相同的型别）。你可以移除 Simple Associative Container 的某个元素，但无法更改其内容。其他的 Associative Container 具有可改变的元素，而且提供某种 iterator，可通过它来改变元素。你可以改变元素值，只要不要更改其 key 就行了。以 Pair Associative Container 为例，它具有两个不同的嵌套型别：`iterator` 和 `const_iterator`。其实这里的 `iterator` 并非是个 mutable iterator，因为你不能写出 `*i = t`。但它也不完全是 constant iterator，因为如果 `i` 的型别是 `map<int, double>::iterator`，你可以写 `i->second = 3`。

至于所谓 Unique Associative Container，保证没有两个元素具有相同的 key。Multiple Associative Container 则允许多个元素具有相同的（*same*）key。

（所谓 "*same* keys"，其意义取决于 Associative Container 的型别。不一定是指 key 相等（*equal*），因为 Associative Container concept 并不要求 key 一定必须为 Equality Comparable。以 Sorted Associative Container 为例，其元素是以某种 Strict Weak Ordering 为标准来递增排列的，如果两个 keys 互不小于对方，这两个 keys 可视为 "*same*"。）

有很多不同的方法，可以将元素组织于数据结构中，使我们得以简单地藉由 key 来查询某个元素。其中两个特别重要的数据结构是二分查找树（binary search tree）和散列表（hash table）。

➢ Refinement Of（强化自）

Forward Container（p.131）。

➢ Associated Types（相关型别）

除了 Forward Container 所定义的外，还采用了一个新型别。

- Key type

  `X::key_type`

  与 `X::value_type` 关联之 key 的型别。注意，key type 与 value type 可能相同（对 Simple Associative Container 而言，它们总会相同）。

➢ Notation（符号）

`X`	某型别，是 Associative Container 的一个 model。
`a`	型别 `X` 的 object。
`T`	`X` 之 value type。
`t`	`X::value_type` 的 object。
`k`	`X::key_type` 的 object。
`p,q`	`X::iterator` 的 object。

➢ Definitions（定义）

1. 如果 a 是一个 Associative Container，p 是一个有效的 iterator 并且可由 a.begin() 前进到达，那么 p 是 a 的一个有效的 iterator。

2. 如果 a 是一个 Associative Container，[p,q) 是一个有效的 range 而且 p 是 a 的一个有效的 iterator，则 [p,q) 是 a 的一个有效的 range。

➢ Valid Expressions（有效表达式）

● Default Constructor（缺省构造函数）

```
X()
X a;
```

语义：　产生空的 container。
必然结果：　container 的元素个数为零。

● Find（查找）

```
a.find(k)
```

返回型别：　如果 a 可变（mutable）就返回 iterator，否则返回 const_iterator。
语义：　返回一个 iterator，指向一个「键值（key）为 k」的元素。这样的元素可能超过一个，若是如此，返回值所指元素并不明确。如果不存在这样的元素，返回值为 a.end()。
必然结果：　返回值如果不是 a.end()，就是指向「键值（key）为 k」的元素。

● Count（计数）

```
a.count(k)
```

返回型别：　size_type。
语义：　返回「键值（key）为 k」的元素个数。
必然结果：　返回值不为负数，并且一定 <= a.size()。对 Unique Associative Container 而言，返回值非 0 即 1。

● Equal range（找出某个等值区间）

```
a.equal_range(k)
```

返回型别：　如果 a 可变（mutable），返回 pair<iterator, iterator>，否则返回 pair<const_iterator, const_iterator>。
语义：　返回所有「键值(key)为 k」的元素。返回的是一个 pair P，[P.first,P.second) 内含所有「键值（key）为 k」的元素。注意这个 member function 带来的暗示：如果两个元素的 key 相同，它们之间必不夹杂 key 不相同的其他元素。「key 相同的各个元素必须连续存储在一起」这一条件正是 Associative Container 的恒长特性之一。如果 a 并未含有任何符合条件的元素，就返回一个空 range。

必然结果： distance(P.first, P.second) == a.count(k)。如果 p 是 [P.first, P.second) 中的某个可提领的（dereferenceable）iterator，则 p 指向某个「键值(key)为 k」的元素。如果 q是 a 中某个可提领的(dereferenceable)iterator，而且 q 不在 range [P.first,P.second)之中，那么 q 所指元素的键值(key)就不是 k。

- Erase key（删除键值）

a.erase(k)

返回型别： void
语义： 将「键值（key）为 k」的所有元素析构掉，并将它们移出 container。
必然结果： a 的元素个数将减少 a.count(k)。a 之中不再内含任何「键值（key）为 k」的元素。

- Erase element（删除元素）

a.erase(p)

返回型别： void
先决条件： p 是 a 中的一个 dereferenceable iterator。
语义： 析构 p 所指的元素，并将它从 a 中移出。
必然结果： a 的元素个数将减少 1。

- Erase range（删除某个区间）

a.erase(p,q)

返回型别： void
先决条件： [p,q) 是 a 中的一个有效 range。
语义： 将 range [p,q) 中的元素全部析构，并将它们从 a 中移出。
必然结果： a 的元素个数将减少 distance(p,q)。

你或许会好奇，为什么以上包含了元素的查询和删除方法，却不见元素的安插方法。原因是安插动作对 Unique Associative Container 和 Multiple Associative Container 有截然不同的意义。安插法系定义于这两个下层 concepts 之中。

➢ **复杂度保证**（Complexity Guarantees）

1. Find 和 Equal range 的平均复杂度至多是对数等级（logarithmic）。
2. Count 的平均复杂度至多是 $O(log(size()) + count(k))$。

3. Erase key 的平均复杂度至多是 $O(log(size()) + count(k))$。

4. Erase element 的平均复杂度为 amortized constant time。

5. Erase range 的平均复杂度至多是 $O(log(size()) + N)$，N 代表 range 中的元素个数。

注意，这些都是平均复杂度，并非最坏情况。某些 associative container concepts 会将它们强化（*strengthen*）至最坏情况。

➢ Invariants（**恒长特性**）

除了 Forward Container 所定义的恒长特性，还必须满足以下性质：

- 连续存储（Contiguous storage）

  键值（key）相同的所有元素必须互相邻接。也就是说如果 `p` 和 `q` 指向具有相同键值的元素，而且 `p` 在 `q` 之前，那么 range `[p,q)` 中的每一个元素都具有相同键值。

- 键值（key）的不可改变性（Immutability）

  Associative Container 的每一个元素，其键值（key）都不可改变。你可以安插或删除元素，但是无法改变任何元素的键值（key）。

➢ Models

下列 container classes 都是 Associative Container 的 model：

```
set
multiset
hash_set
hash_multiset
map
multimap
hash_map
hash_multimap
```

9.3.2 Unique Associative Container

所谓 Unique Associative Container 是一种 Associative Container，具有「container 中的每一个 key 都独一无二」的性质。Unique Associative Container 之中没有两个元素具有相同的 key。

这意味将元素「安插」于 Unique Associative Container 之中需要某种先决条件：如果 `t` 的 key 已存在于 container 中，`t` 就无法被安插进去。

➢ Refinement Of（强化自）

Associative Container（p.145）。

➢ Associated Types（相关型别）

仅止于定义于 Associative Container 之中的 associated types。

➢ Notation（符号）

X	某型别，是 Unique Associative Container 的一个 model。
a	型别 X 的 object。
T	X 之 value type。
t	X::value_type 的 object。
k	X::key_type 的 object。
p,q	X::iterator 的 object。

➢ Valid Expressions（有效表达式）

除了 Associative Container 所定义的表达式外，以下表达式也必须有效：

● Range Constructor（区间构造函数）

```
X(i, j)
X a(i, j);
```

型别条件： i 和 j 是 Input Iterators，其 value type 可转换成 T。

先决条件： [i,j) 是个有效 range。

语义： 产生一个 associative container，内含 [i,j) 的所有元素，每个元素具有独一无二的 key。

必然结果： size() <= distance(i, j)。注意，是 <= 而非 ==，因为 [i,j) 之中可能有一些元素的 key 相同，果真如此便只会安插这些元素中第一个元素。

● Insert element（安插元素）

```
a.insert(t)
```

返回型别： pair<X::iterator, bool>

语义： 当且仅当（if and only if）a 没有任何一个元素与 t 的键值冲突，则将 t 安插于 a。返回值是 pair P，其第二元素的型别为 bool。如果 a 的某个元素与 t 的键值冲突，则 P.first 指向该元素，P.second 为 false。如果 a 并未含有这样的元素，则 P.first 指向新插入的元素并且 P.second 为 true。也就是说，当 t 实际被安插于 a 之中，P.second 便为 true。

必然结果： P.first 是个 dereferenceable iterator，所指元素的 key 与 t 的 key 相同。当 P.second 为 true，a 的元素个数增加 1。

- Insert range（安插某个区间）

  a.insert(i, j)

  型别条件： i 和 j 都是 Input Iterators，其 value type 可转换为 T。
  返回型别： void
  先决条件： [i,j) 为有效的 range。
  语义： 相当于对 range [i,j) 中的每个元素 t 做 a.insert(t) 动作。也就是说，只要 a 的元素和 range 内的元素不产生键值冲突的情况，整个 range 便都可以安插进入 a。
  必然结果： a 的元素个数至多增加 distance(i, j)。

➢ 复杂度保证（Complexity Guarantees）

单一元素的安插动作，平均复杂度至多是对数等级（logarithmic）。

整个 range 的元素安插动作，平均复杂度至多是 $O(N \log(\texttt{size()} + N))$，其中 N 代表 range 中的元素个数。

➢ Invariants（恒长特性）

除了 Associative Container 所定义的恒长特性外，还必须满足以下性质：

- Uniqueness（唯一性）
  不可能有两个元素具有相同的 key。换句话说对于每个「型别为 key_type」的 k，a.count(k) 非 0 即 1。

➢ Models

下列 STL container classes 都是 Unique Associative Container 的 models：

```
set
map
hash_set
hash_map
```

9.3.3 Multiple Associative Container

所谓 Multiple Associative Container 是一种 Associative Container，可内含具有相同键值（key）的元素。也就是说，它是一种不受 Unique Associative Container 限制的 Associative Container。

将元素安插到 Multiple Associative Container 之中，就像将元素安插到 Sequence 一般，并不需要什么特别条件。安插时，`insert` 不会检查 container 之中是否已具备了键值相同的元素。

➢ Refinement Of（强化自）

Associative Container（p.145）。

➢ Associated Types（相关型别）

仅止于定义于 Associative Container 之中的 associated types。

➢ Notation（符号）

`X`	某型别，是 Multiple Associative Container 的一个 model。
`a`	型别 `X` 的 object。
`T`	`X` 之 value type。
`t`	`X::value_type` 的 object。
`k`	`X::key_type` 的 object。
`p,q`	`X::iterator` 的 object。

➢ Valid Expressions（有效表达式）

除了 Associative Container 所定义的表达式外，以下表达式也必须有效：

- Range Constructor（区间构造函数）

```
X(i, j)
X a(i, j);
```

型别条件： `i` 和 `j` 是 Input Iterators，其 value type 可转换成 `T`。
先决条件： `[i,j)` 是个有效 range。
语义： 产生一个 associative container，其内包含 `[i,j)` 内的所有元素。
必然结果： container 的元素个数等于 `distance(i, j)`。range `[i,j)` 中的每个元素都会出现在 container 之中。

- Insert element（安插元素）

```
a.insert(t)
```

返回型别： `X::iterator`

语义： 无条件安插 t 于 a 中。返回一个 iterator，指向新插入的元素。
必然结果： a 的元素个数加 1。a.count(t) 的值增加 1。

- Insert range（安插某个区间）

```
a.insert(i, j)
```

型别条件： i 和 j 都是 Input Iterators，其 value type 可转换为 T。
返回型别： void
先决条件： [i,j) 是个有效的 range。
语义： 等价于对 range [i,j) 中的每一个元素 t 做 a.insert(t) 动作。也就是说 range 内的每个元素都会被安插于 a。
必然结果： a 的元素个数增加 distance(i, j)。

➢ 复杂度保证（Complexity Guarantees）

单一元素的安插动作，平均复杂度至多是对数等级（logarithmic）。

整个 range 的元素安插动作，平均复杂度至多为 $O(N \log(\texttt{size}() + N))$，其中 N 代表 range 中的元素个数。

➢ Models

下列 STL container classes 都是 Multiple Associative Container 的 models：

```
multiset
multimap
hash_multiset
hash_multimap
```

9.3.4 Simple Associative Container

所谓 Simple Associative Container 是一种 Associative Container，其元素自身就是 key，而非 key 再加上某些额外数据。

将元素安插于 Simple Associative Container 之中，就像将元素安插于 Sequence 一般，并不需要什么特别条件。安插元素时，insert 不会检查 container 之中是否已有相同键值（key）的元素。

➢ Refinement Of（强化自）

Associative Container（p.145）。

➢ Associated Types（相关型别）

仅止于定义于 Associative Container 之中的 associated types。不过 Simple Associative Container 多了两个型别条件：

- Key type

  `X::key_type`

  这是和 `X::value_type` 相关联的 key type。`key_type` 与 `value_type` 两型别必须相同。

- Iterator

  `X::iterator`

  用来「遍历 Simple Associative Container 众元素」的 iterator 的型别。型别 `X::iterator` 与 `X::const_iterator` 必须具有相同行为，甚至两者完全相同。Simple Associative Container 并不提供 mutable iterators，这是因为「Associative Container 的 key 一定不可改变」。是的，key 永远不能被改变，而 Simple Associative Container 的元素值就是它们的 key，所以 Simple Associative Container 中的元素值不能被改变。

➢ Notation（符号）

`X`	某型别，是 Simple Associative Container 的一个 model。
`a`	型别 `X` 的 object。
`k`	`X::key_type` 的 object。
`p,q`	`X::iterator` 的 object。

➢ Valid Expressions（有效表达式）

除了 Associative Container 所定义的表达式外，没有其他更多的了。

➢ Invariants（恒长特性）

元素值不可改变（Immutability of elements）

Simple Associative Container 的元素不可改变。你可以安插或删除元素，但是元素值绝不能被改变。每个 Associative Container 具有不可改变的 key，Simple Associative Container 的元素即是 key，所以元素不能改变。

➢ Models

下列 container classes 都是 Simple Associative Container 的 models：

```
set
multiset
hash_set
hash_multiset
```

9.3.5 Pair Associative Container

所谓 Pair Associative Container 是一种 Associative Container，将 key 与某些 objects 产生关联。其 value type 形式为 `pair<const key_type, mapped_type>`，其中 `key_type` 为 container 的 key type。

➢ Refinement Of（强化自）

Associative Container（p.145）。

➢ Associated Types（相关型别）

除了 Associative Container 所定义的之外，还多增加了一些新型别：

- Key type

  `X::key_type`

  元素的键值（key）型别。

- Mapped type

  `X::mapped_type`

  元素数据（data）的 型别。Pair Associative Container 可被视为一个关联式数组（associated array），存储着型别为 `mapped_type` 的元素，或被视为一种映射（mapping），将型别为 `key_type` 的值映射至型别为 `mapped_type` 的值。mapped type 有时也称为 data type。

- Value type

  `X::value_type`

  container 内所存储之 objects 型别。Value type 必须是 `pair<const key_type, mapped_type>`。value type 之所以为 `pair<const key_type, mapped_type>` 而非 `pair<key_type, mapped_type>`，为的是满足 Associative Container 的一个恒长特性：key 不得改变。在 Pair Associative Container object 中，`mapped_type` 的部分是可以改变的，但 `key_type` 的部分则不行。注意，这透露了一个事实：Pair Associative Container 无法提供 mutable iterator（此物定义于 Trivial Iterator 的需求条件中），因为 mutable iterator 的 value type 必须是 Assignable，但 `pair<const key_type, mapped_type>` 却非 Assignable。不过，Pair Associative Container 的确能够供应非完全的 constant iterators，这种 iterators 允许你撰写诸如 `i->second = d` 的表达式。

➢ Notation（符号）

`X`	某型别，是 Pair Associative Container 的一个 model。
`a`	型别 `X` 的 object。
`t`	`X::value_type` 的 object。
`k`	`X::key_type` 的 object。

d	X::mapped_type 的 object。
p,q	X::iterator 的 object。

➢ Valid Expressions（有效表达式）

仅止于 Associative Container 所定义的表达式。

➢ Models

以下 container classes 都是 Pair Associative Container 的 models：

```
map
multimap
hash_map
hash_multimap
```

9.3.6 Sorted Associative Container

所谓 Sorted Associative Container 是一种 Associative Container，针对 key 做 Strict Weak Ordering 递增排序，来排列各元素。由于 Strict Weak Ordering 导入某种等价（equivalence）关系，所以我们可以这样定义：如果两个 keys 互不小于对方，这两个 keys 便视为相同。

Sorted Associative Container 保证所有的 Associative Container 基本行为最差也不过是对数（logarithmic）复杂度。

➢ Refinement Of（强化自）

Associative Container（p.145）、Reversible Container（p.133）。

➢ Associated Types（相关型别）

除了定义于 Associative Container 和 Reversible Container 中的型别之外，另外多增加了两个新型别：

- Key comparison（键值（key）比较）

  X::key_compare

  这是一个 function object type：以 Strict Weak Ordering 来比较两个 keys。其引数型别为 X::key_type。

- Value comparison（实值比较）

  X::value_compare

  这是一个 Strict Weak Ordering type，用来比较两个实值。其引数型别为 X::value_type。「实值比较」系根据「键值（key）比较」的结果而决定。只要将元素对应键值传给一个型别为 key_compare 的 function object，就可以比较两个元素的 value_type。

➢ Notation（符号）

X	某型别，是 Associative Container 的一个 model。
a	型别 X 的 object。
T	X 之 value type，X::value_type。
t	X::value_type 的 object。
k	X::key_type 的 object。
p,q	X::iterator 的 object。
c	X::key_compare 的 object。

➢ Valid Expressions（有效表达式）

除了 Associative Container 和 Reversible Container 所定义的外，以下表达式也必须有效：

● Default constructor（缺省构造函数）

```
X()
X a;
```

语义：　　产生一个以 key_compare() 作为比较函数的空的 container。

● Default constructor with compare（夹带「比较准则」的缺省构造函数）

```
X(c)
X a(c);
```

语义：　　产生一个以 c 作为比较函数的空的 container。

● Range construct（区间构造函数）

```
X(i, j)
X a(i, j);
```

型别条件：　i 和 j 为 Input Iterators，其 value type 可转换为 T。
先决条件：　[i,j) 是一个有效的 range。
语义：　　等价于：

```
X a;
a.insert(i, j);
```

insert 所带来的影响（亦即它究竟会安插 [i,j) 的所有元素或只是单一元素），取决于 X 为 Multiple Associative Container 或是 Unique Associative Container 的 model。

● Range constructor with compare（夹带「比较准则」的区间构造函数）

```
X(i, j, c)
X a(i, j, c);
```

型别条件：i 和 j 为 Input Iterators，其 value type 可转换为 T。
先决条件：[i,j) 为一个有效的 range。
语义：等价于：

```
X a(c);
a.insert(i, j);
```

- Key comparison（键值比较）

```
a.key_comp()
```

返回型别： key_compare
语义： 返回 a 的 key comparison object。后者在 a 构造时设定，之后便不能改变了。

- Value comparison（实值比较）

```
a.value_comp()
```

返回型别： value_compare
语义： 返回 a 之 key comparison object 所导出的 value comparison object。
必然结果： 如果 t1 和 t2 是 value_type objects，而 k1 和 k2 是其相应键值，那么 a.value_comp()(t1, t2) 等价于 a.key_comp()(k1, k2)

- Lower bound（下界）

```
a.lower_bound(k)
```

返回型别： 如果 a 可变（mutable）就返回 iterator，否则返回 const_iterator。
语义： 返回一个 iterator，指向第一个「键值不小于 k」的元素。若无此等元素存在，就返回 a.end()。
必然结果： 如果 a 内含有键值为 k 的元素，lower_bound 的返回值便指向这般元素中的第一个。

- Upper bound（上界）

```
a.upper_bound(k)
```

返回型别： 如果 a 可变（mutable）就返回 iterator，否则返回 const_iterator。
语义： 返回一个 iterator，指向第一个「键值大于 k」的元素。若无此等元素存在，就返回 a.end()。
必然结果： 如果 a 内含有键值为 k 的元素，upper_bound 的返回值便指向这般元素中的最后元素的下一个。

- Equal range（等值区间）

  `a.equal_range(k)`

  返回型别： 如果 a 可变（mutable），返回 pair<iterator,iterator>，否则返回 pair<const_iterator,const_iterator>。

  语义： 返回 pair P，使得 P.first 为 a.lower_bound(k) 且 P.second 为 a.upper_bound(k)。这个定义符合 Associative Container 条件中所描述的语义，但更严格。如果 a 不包含任何键值为 k 的元素，则 a.equal_range(k) 返回一个 empty range，用来表示「如果这些元素存在，它们应该放置何处」。然而 Associative Container 仅仅要求在这种状况下返回值是任意的 empty range。

  必然结果： 所有与 k 相等的 key 都包含于 range [P.first,P.second) 之中。

- Insert with hint（夹带提示的安插行为）

  `a.insert(p, t)`

  返回型别： iterator

  先决条件： p 是 a 的一个 nonsingular iterator。

  语义： 与 a.insert(t) 相同。如果 X 是 Multiple Associative Container 的一个 model，就做无条件安插动作，返回值（一个 iterator）指向新安插的元素。如果 X 是 Unique Associative Container 的一个 model，那么当尚未存在键值相同的元素时，t 才会被安插。返回值是个 iterator，指向新安插的元素（如果有被安插的话），或是指向「键值与 k 相同」的元素。引数 p 作为提示用，应该指向 t 打算被安插的位置。这个提示不会影响正确性，只会影响性能。

➢ 复杂度保证（Complexity Guarantees）

与 Associative Container 相比，Sorted Associative Container 有十分严格的复杂度保证。Associative Container 的保证只适用于平均复杂度，因此最坏情况可能很糟。然而 Sorted Associative Container 保证了最坏情况下的复杂度上限。

1. range constructor 以及 range constructor with compare 的复杂度一般是 $O(N \log N)$，其中 N 是用来构造该 container 的 range 的大小。然而如果 range 已经根据 value_comp() 作了递增排列的话，复杂度便与 N 成线性关系。

2. `key_comp()` 和 `value_comp()` 耗用 constant time。

3. 一般来说 insert with hint 的复杂度是对数（logarithmic）等级，但如果提示的引数正确的话 — 也就是说如果 `t` 可安插于紧邻 `p` 之前，安插动作会耗用 amortized constant time。

4. insert range 的复杂度一般而言是 ***O(N log N)***，其中 ***N*** 是 range 的大小。然而如果在安插之前 range 已经以 `value_comp()` 作了递增排序，那么复杂度将与 ***N*** 成线性关系。

5. 单一元素的 `erase` 动作耗用 amortized constant time。

6. Erase key 的复杂度是 ***O(log(size()) + count(k))***。

7. Erase range 的复杂度是 ***O(log(size()) + N)***，其中 ***N*** 是欲删除之 range 的大小。

8. Find 的复杂度是对数（algarithmic）等级。

9. Count 的复杂度是 ***O(log(size()) + count(k))***。

10. Lower bound、upper bound 以及 equal range 的复杂度是对数（algarithmic）等级。

➢ Invariants（恒长特性）

除了 Associative Container 所定义的恒长特性外，还必须满足以下性质：

- 实值比较（value compare）的定义

  如果 `t1` 和 `t2` 是型别为 `X::value_type` 的 objects，而 `k1` 和 `k2` 是其相关键值（keys），则 `a.value_comp()` 会返回一个 function object，使 `a.value_comp()(t1, t2)` 等价于 `a.key_comp()(k1, k2)`。

- 递增次序（Ascending order）

  Sorted Associative Container 内的元素一定根据 key 做递增次序排列。如果 `a` 是一个 Sorted Associative Container，则

  ```
  is_sorted(a.begin(), a.end(), a.value_comp())
  ```

  必定为 `true`。这就是为什么它被称为 "sorted associative container" 的原因。

➢ Models

下列 STL container classes 都是 Sorted Associative Container 的 models：

```
set
map
multiset
multimap
```

9.3.7 Hashed Associative Container

所谓 Hashed Associative Container 是一种 Associative Container，以 hash table（散列表）实现完成。Hashed Associative Container 的元素并不保证具有任何富含意义的次序。更明确地说它们并不做排序动作。大部分 Hashed Associative Container 的操作行为，在最坏情况下，其复杂度与 container 的大小成线性关系，平均复杂度则是 constant time。这意味对于「只是单纯地存储与取出数值，次序并不重要」的应用而言，Hashed Associative Container 往往比 Sorted Associative Container 快上许多。

有很多论文对 hash table 有许多讨论。例如 [Knu98b] 的 6.4 节。

➢ Refinement Of（强化自）

Associative Container（p.145）。

➢ Associated Types（相关型别）

除了 Associative Container 所定义的外，还多加了两个新型别：

- Hash function（散列函数）

  `X::hasher`

  这是一种 function object type，是 Hash Function（p.123）的一个 model。`X::hasher` 的引数型别是 `X::key_type`。

- Key equality（键值相等）

  `X::key_equal`

  这是一个 Binary Predicate，其引数型别为 `X::key_type`。一个型别为 `key_equal` 的 object，如果其引数(s) 的键值（key）相同，就返回 `true`，否则返回 `false`。`X::key_equal` 必须是一种等价（equivalence）关系。此外，hash function 与 key equality function 必须一致。如果根据 key equality function 判断出两个键值（keys）相等，则这两个键值必须能够 "hashing" 出相同的实值（values）。

➢ Notation（符号）

`X`	某型别，是 Associative Container 的一个 model。
`a`	型别 `X` 的 object。
`T`	`X` 之 value type，`X::value_type`。
`t`	`X::value_type` 的 object。
`k`	`X::key_type` 的 object。
`p,q`	`X::iterator` 的 object。
`n`	`X::size_type` 的 object。
`h`	`X::hasher` 的 object。
`c`	`X::key_compare` 的 object。

➢ Definitions（定义）

1. Hashed Associative Container x 的 hash function（散列函数）是一种 Unary Function，其引数型别为 `X::key_type`，其返回值型别为 `size_t`。Hash function 必须具备注定性（deterministic），意思是当我们以相同的引数调用它，它一定得返回相同的值。但其返回值应该尽可能均匀（uniform）。理想情况下，不会有两个键值（keys）"hashing" 出相同的实值（values）。

   粗糙的 hash function — 也就是不同的键值（keys）"hashing" 出相同实值（values）— 会折损性能。最坏的情况是每个 key 都 hashing 出相同的 value。若真是这样，Hashed Associative Container 的大部分操作行为的运行期复杂度将是 *O(N)*，而非 *O(1)*。

2. Hashed Associative Container 的元素会分组成一个个的 buckets（桶子）。Hashed Associative Container 利用 hash function 的值来决定将元素赋值给那一个 bucket。

3. Hashed Associative Container 的元素个数除以 buckets 个数，称为 *load factor*（负载因子）。一般来说，load factor 较小者其执行速度比 load factor 较大者快速。

➢ Valid Expressions（有效表达式）

除了 Associative Container 和 Reversible Container 所定义的，以下表达式也必须有效：

- Default constructor（缺省构造函数）

```
X()
X a;
```

语义：　产生一个空的 container，以 `hasher()` 作为 hash function，并以 `key_equal()` 作为 key equality function。

必然结果：　container 大小为 0。buckets 数量是个未明定的缺省值。hash function 是 `hasher()`，而 key equality function 是 `key_equal()`。

- Default constructor with bucket count（「接受 bucket 数量」之缺省构造函数）

```
X(n)
X a(n);
```

语义：　以 `hasher()` 作为 hash function，并以 `key_equal()` 作为 key equality function，产生一个至少具有 `n` 个 buckets 的空 container。

必然结果：　container 大小为 0。buckets 数量大于等于 `n`。hash function 是 `hasher()`，key equality function 是 `key_equal()`。

- Default constructor with hash function（「接受 hash function」之缺省构造函数）

```
X(n, h)
X a(n, h);
```

语义： 以 h 为 hash function 并以 key_equal() 为 key equality function，产生一个最少具有 n 个 buckets 的空 container。

必然结果： container 大小为 0。buckets 数量大于等于 n。hash function 是 h，key equality function 是 key_equal()。

- Default constructor with key equality（「接受 key equality function」之缺省构造函数）

```
X(n, h, k)
X a(n, h, k);
```

语义： 以 h 为 hash function 并以 k 为 key equality function，产生一个最少具有 n 个 buckets 的空 container。

必然结果： container 大小为 0。buckets 数量大于等于 n。hash function 为 h，key equality function 为 k。

- Range construct（「以某区间为初值」的构造函数）

```
X(i, j)
X a(i, j);
```

型别条件： i 和 j 皆为 Input Iterators，其 value type 可转换为 T。

先决条件： [i,j) 是一个有效的 range。

语义： 等价于：

```
X a;
a.insert(i, j);
```

insert 动作带来的影响 — 也就是说它是否会将 [i,j) 的所有元素安插进去，抑或只是单一元素 — 取决于 X 是 Multiple Associative Container 或是 Unique Associative Container 的 model。

- Range constructor with bucket count（「接受 bucket 数量」之区间构造函数）

```
X(i, j, n)
X a(i, j, n);
```

型别条件： i 和 j 皆为 Input Iterators，其 value type 可转换为 T。

先决条件： [i,j) 为有效的 range。

语义： 等价于：

```
X a(n);
a.insert(i, j);
```

- Range constructor with hash function（「接受 hash function」之区间构造函数）

```
X(i, j, n, h)
X a(i, j, n, h);
```

型别条件： i 和 j 皆为 Input Iterators，其 value type 可转换为 T。
先决条件： [i,j) 为有效的 range。
语义： 等价于：

```
X a(n, h);
a.insert(i, j);
```

- Range constructor with key equality（接受 key equality function 之区间构造函数）

```
X(i, j, n, h, k)
X a(i, j, n, h, k);
```

型别条件： i 和 j 皆为 Input Iterator，其 value type 可转换为 T。
先决条件： [i,j) 为有效的 range。
语义： 等价于：

```
X a(n, h, k);
a.insert(i, j);
```

- Hash function（散列函数）

```
a.hash_funct()
```

返回型别： X::hasher
语义： 返回 a 的 hash function。

- Key equality（键值相等检验函数）

```
a.key_eq()
```

返回型别： X::key_equal
语义： 返回 a 的 key equality function。

- Bucket count（Bucket 计数）

```
a.bucket_count()
```

返回型别： X::size_type
语义： 返回 a 的 buckets 数量。

- Resize（重定大小）

```
a.resize(n)
```

语义： 增加 a 的 bucket 数量。
必然结果： a 的 bucket 数量至少为 n。指向 a 内元素的所有 iterators 都仍有效。注意，虽然 resize 并不会造成 iterator 失效，但通常它会改变 iterators 间的次序关系。如果 i 和 j 是指向 Hashed Associative Container 的 iterators，而且 j 在 i 之后，则 resize 完成后并不保证 j 还是在 i 之后。关于次序问题，唯一保证的是连续存储性：键值（key）相同的元素一定紧邻。

作为一个一般性规则，Hashed Associative Container 尝试将其 load factor 维护于某个相当狭窄的区间内。（这必须满足复杂度限制。如果 load factor 允许无限成长，则 Hashed Associative Container 会在元素个数成长时拙劣地扩充。）

当你将元素安插于 Hashed Associative Container 时，你可能会触发 auto resizing 机制。它会像 resize 动作一样地对 iterators 和 ranges 造成影响。它会保留 iterators 的有效性，但不保留 iterators 之间的次序关系。当然，这是 resize 的生存理由之一，也就是为什么会有 constructor 可让你指定 bucket 数量的原因。如果你事先知道一个 Hashed Associative Container 会成长到多大，那么趁着 container 还是空的时候就设定 bucket 数量，会比较有效。

➢ 复杂度保证（Complexity Guarantees）

比起 Associative Container，Hashed Associative Container 拥有更严格的「一般情况下的复杂度保证」。不过其「最坏情况下的复杂度保证」仍然比较弱。最坏情况通常与差劲的 hash function 有很大的关系。

- default constructor、constructor with bucket count、constructor with hash function、constructor with key equality 都耗用 amortized constant time（最坏情况）。
- range constructor、range constructor with bucket count、range construct with hash function、range constructor with key equality 的复杂度都是 $O(N)$，其中 N 是 range 的大小。
- 用来取得 hash function 及 key equality function 的表达式，都耗用 constant time。
- 删除单一元素以及 erase(p) 两个动作，都耗用 amortized constant time（最坏情况）。
- `erase(k)`（删除键值为为 k 的所有元素）的平均复杂度是 $O(n)$，其中 n 为 `count(k)`。最坏情况是 $O(N)$，其中 N 是 container 的大小。
- `erase(p,q)`（删除某个 range）的平均复杂度是 $O(n)$，其中 n 是欲删除之 range 大小。最坏情况是 $O(N)$，其中 N 是 container 的大小。
- `find(k)`（根据 key 来查询）的平均复杂度是 $O(1)$，最坏情况是 $O(N)$，其中 N 是 container 的大小。

- `equal_range(k)`（寻找键值为 `k` 的所有元素）的平均复杂度是 *O(count(k))*，最坏情况是 *O(N)*，其中 *N* 是 container 的大小。
- `count(k)`（计算键值为 `k` 的元素个数）的平均复杂度是 *O(count(k))*，最坏情况是 *O(N)*，其中 *N* 是 container 的大小。
- Bucket 计数函数的复杂度是 *O(1)*（最坏情况）。
- resize 的复杂度最坏情况下是 *O(N)*，其中 *N* 是 container 的大小。

➢ Models

下列 STL container classes 都是 Hashed Associative Container 的 models：

`hash_sethash_map`

9.4 Allocator（空间配置器）

Allocator 是一个 class，用来封装内存分配与归还的某些细节。STL 预定义的所有 containers 都使用 Allocator 来做内存管理。举个例子，当你具现化（instantiate）`vector` template 时，你必须指定使用哪一个 Allocator class。

关于 allocators，有三个事实比任何技术细节都来得重要。

第一（同时也是最重要的），你或许丝毫不必知道 Allocator 的任何事。使用 STL 预定义的 container 时，你不需要知道 Allocator 的存在，因为这些 container classes 使用缺省的 template 参数来隐藏对 Allocator 的使用。例如缺省情况下 `vector<int>` 意味 `vector<int, allocator<int> >`。你甚至不必知道 Allocator 便可以撰写新的 container class。container class 不必使用精致复杂的内存管理技术。事实上，如同我们在 5.1 节所见到的 `block` class，它不必执行任何动态内存分配行为。

第二，allocator 是 STL 之中最不具移植性的一个主题。虽然 C++ standard 的确提到了 Allocator concept，但 1998 年仍然有很多非标准的 allocator 被广泛运用。（部分原因是标准的 allocator 运用许多语言特性，而这些语言特性尚未被完全实现出来。）

第三，虽然 container 以 allocator 作为参数之一，但这并不表示你得以控制每一个想像可及的内存管理层面。Allocator concept 与以下各点毫无关系：`deque` 节点中包含多少元素、container 的元素是否连续存储、`vector` 重新分配时如何增长内存。

基本上，Allocator 将「从自由存储空间中获得或归还内存」的方法封装了起来。如果你通过 Allocator 要求内存，Allocator 可能会使用 malloc 或 new 或你的操作系统的某种特别方法来完成这个动作。由于分配单一大块内存通常比分配许多小块内存更快速，所以 Allocator 可能会维护一个「free list」。然而所有的复杂度都被隐藏起来。一个 Allocator 应该连分配小个头的 objects 也有高效率，而 containers 也没有必要维护其自己的 free list（这是 C++ standard 比原始的 HP STL 更简化的做法之一）。

一般而言，同一个 Allocator class 的两个 objects 不必等价（equivalent）。如果 a1 和 a2 都是 My_Allocator 型别的两个 objects，a1 和 a2 可能不会以相同的方式分配内存。假设你的操作系统可以在不同的 memory pool 上分配内存。那么你可以写一种 Allocator class，让这个 class 所衍生的不同 objects 代表不同的 memory pool。当你想要产生一个型别为 vector<int, My_Allocator<int>> 的 container，你可以指明 vector 使用哪一个 memory pool。所有的标准 containers 都允许你指定使用哪一个 allocator object。

（所有的标准 container 都使用缺省参数，所以你不必为指定何种 allocator 而烦恼，除非你必须使用非缺省值。如果你使用缺省的 Allocator class allocator，那便不打紧了，因为 class allocator 的所有 object 都相同。）

就像 container classes 一样，每一个 Allocator class 也具有 value type，作为其所分配之 object 的型别。如果 Allocator 的 value type 为 T，则 member function allocate(n) 会索求足够的内存空间以包含 n 个 T objects。和 container class 一样，Allocator class 也有嵌套型别 pointer、reference……原则上这些型别可定义成另一种内存模型（memory model），使 pointer 不再是 T* 而是其他某种类似指针的型别。不过，定义另一种内存模型的自由度实际上有所限制，因为除了一般的 T&，C++ 并不支持任何得以定义 reference-like type 的良好方法。

由于 Allocator class 拥有 value types，所以必须有一种方法，可以将某个 value type 转换成另一个 value type。试考虑连接链表（link list）：list<int> 拥有一个 Allocator，其 value type 为 int，但事实上 list 并不分配 int objects，它分配的是 list nodes（节点）。因此 list class 必须有能力将一个「具有某种 value type」的 allocator 转换成「具有另一种 value type」的 allocator，而且它必须将其中一个 class 的 allocator object 转换为第二个 class 的 object。Allocator 的 rebind 机制可以完成第一点，而 Allocator 的 generalized copy constructor 可做到第二点。

最后提醒你。你很可能完全不必担心 allocator 的任何事，甚至根本不必使用「另一种内存模型」以及「不同的 allocator instances」等高级特性。如果你真打算使用这些 allocator 高级特性，应该严密研读你的编译器说明文档，因为 C++ standard [ISO98] 20.1.5 节有一个重要警告：

> 本国际标准文档所描述的 container 实现法，得假设其 Allocator 之 template 参数不仅符合表 32 所列条件，还需符合以下两个额外条件：

1. 某已知 allocator 的所有 instances 都必须可以互换（interchangeable），而且总是可以互相比较是否相等。
2. 各个 typedef members: `pointer`、`const_pointer`、`size_type` 和 `difference_type` 必须分别为 `T*`、`T const*`、`size_t` 和 `ptrdiff_t`。

本标准鼓励实现者在程序库中接受「将更一般化之内存模型封装起来」的 allocator，并支持 "non-equal" instances。如此一来，containers 所课征（impose）的「allocator 条件」将超越表 32 所列条件，而当 allocator instances *compare non-equal* 时，containers 和 algorithms 的语义将由实现品来定义。

这意味尽管 C++ standard 要求 Allocator 对于「不同的内存模型以及不同的 allocator instances」得有语法上的挂勾（syntactic hooks），但不保证你可以在你的实现品所提供的 container 上使用这些特点。

➢ Refinement Of（强化自）

Equality Comparable（p.85）、Default Constructible（p.84）。Allocator 不是 Assignable 的强化（refinement），因为尽管它定义了一个 copy constructor，但并没有定义 assignment operator。

➢ Associated Types（相关型别）

- Value type（数值型别）

  `X::value_type`

  allocator 所管理的型别。

- Pointer type（指针型别）

  `X::pointer`

  一个指针（不必是 C++ 传统指针），指向 `X::value_type`。Pointer type 必须能够转换成 `const` pointer type、`void*` 或是 `value_type*`。

- Const pointer type

  `X::const_pointer`

  这是 `X::value_type` 的 `const` 指针（不必是 C++ 的传统指针）。

- Reference type（参考型别）

  `X::reference`

  这是 `X::value_type` 的 reference。由于 C++ 不可能造出另一种 reference type，所以 `X::reference` 必须是 `X::value_type&`。

- Const reference type

  `X::const_reference`

  这是 `X::value_type` 的 `const` reference。如同 reference type 一般，`const` reference type 必须是 `X::value_type&`。

- Difference type（差距型别）

  `X::difference_type`

  这是一个有正负号的整数型别，表示两个 `X::pointer` objects 之间的差距。

- Size type（大小型别）

  `X::size_type`

  这是一个无正负号的整数型别，用来表示型别为 `X::difference_type` 的任何非负值。

- Alternate allocator type（另一个 allocator types）

  `X::rebind<U>::other`[18]

  对于任何型别 `U`，`X::rebind<U>::other` 是「value type 为 `U`」的一个 allocator。型别 `X::rebind<X::value_type>::other` 等于 `X` 自身。

➢ **Notation（符号）**

`T, U`	任何型别。
`X`	「value type 为 `T`」的 Allocator。
`Y`	「value type 为 `U`」的相应的（corresponding）Allocator。
`a, b`	型别 `X` 的 object。
`y`	型别 `Y` 的 object。
`t`	型别 `T` 的 object。
`k`	`X::key_type` 的 object。
`p`	`X::pointer` 的 object。
`n`	`X::size_type` 的 value。

➢ **Valid Expressions（有效表达式）**

- Default constructor（缺省构造函数）

  ```
  X()
  X a;
  ```

  返回型别： `X`
  语义： 产生 `X` 的一个缺省实体。

- Copy constructor（拷贝构造函数）

  ```
  X b(a);
  X(a);
  ```

  返回型别： `X`
  语义： 产生 `a` 的复制品。
  必然结果： 该复制品与 `a` 做比较，结果相等。

---

[18] 由于 C++ 不支持「template typedef」，所以语法上有点笨拙。`other` 必须写为嵌套的 template class `rebind` 中的一个 `typedef`。

- Generalized copy constructor（一般化拷贝构造函数）

```
X b(y);
X(y);
```

返回型别： X
语义： 产生 y 的复制品，那是一个「value type 为 U」的 Allocator。
必然结果： Y(X(y)) == y。

- Comparison（比较）

```
a == b;
```

返回型别： 任何可转换为 bool 的型别。
语义： 当且仅当（if and only if）以 a 分配的空间可由 b 归还，则返回 true。反之亦然。

- Allocate（分配）

```
a.allocate(n)
```

返回型别： X::pointer
先决条件： 0 < n <= a.max_size()。
语义： 分配 n * sizeof(T) 个 bytes 内存。
必然结果： 返回值是个指针，指向未经初始化的内存块，其中足以包含 n 个 T objects。

- Allocate with hint（带提示的分配行为）

```
a.allocate(n, q)
```

型别条件： q 是型别 Y::const_poiner 的 value（对某些 Y 而言）。
返回型别： X::pointer
先决条件： 0 < n <= a.max_size()。q 如果不是空指针，就是指向先前调用 allocate 后获得的指针。
语义： 分配 n * sizeof(T) 个 bytes 内存。指针引数 q 是个提示。allocator 可能用它来增进局部性（*locality*），或是根本就忽略该提示。
必然结果： 返回值是个指针，指向未经初始化的内存块，其中足以包含 n 个 T objects。

- Deallocate（归还）

```
a.deallocate(p, n)
```

返回型别： void
先决条件： p 是先前调用 allocate 后获得的指针，n 是当初传给该 allocate 的引数。p 不为 null 且 n 不为零。
语义： 归还 p 所指之内存。

- Maximum size（空间上限）

  `a.max_size()`

  返回型别： `X::size_type`
  语义： 返回一个值，此值是 `allocate` 所能接受的极大值。

- Construct（构造）

  `a.construct(p, t)`

  返回型别： `void`
  先决条件： `p` 指向未经初始化的内存局部，其大小至少为 `sizeof(T)`。
  语义： 等价于 `new((void*) p) T(t);`

- Destroy（析构）

  `a.destroy(p)`

  返回型别： `void`
  先决条件： `p` 指向型别为 `T` 的 object。
  语义： 等价于 `((T*) p)->~T();`

- Address（取地址）

  `a.address(r)`

  型别条件： `r` 的型别为 `T&` 或 `const T&`。
  返回型别： 如果 `r` 的型别为 `T&`，就返回 `pointer`，如果 `r` 的型别为 `const T&`，就返回 `const_pointer`。
  先决条件： `r` 是「`a` 所分配之 object」的 reference。
  语义： 返回一个指针，指向 `r`。

➢ Models

`allocator`（p.187）。

第三篇

参考手册：算法与类

Reference Manual : Algorithms and Classes

第 10 章

基本组件

Basic Components

本章所涵盖的型别与函数，是程序库在许多不同地方用到的一些基本素材。大部分都极为简单，不过其中某些十分低阶，你只有在实现一个新的 container 时才会用到它们。

10.1 pair

pair<T1, T2>

Class pair<T1, T2> 是一个可拥有不同成分的「数对」。它持有一个型别为 T1 的 object 以及一个型别为 T2 的 object。pair 很像一个 Container，它「拥有」其元素（当 pair 自身被析构时，元素也会被析构），但它其实并非是 Container 的一个 model，因为它并没有支持 Container 的元素标准访问法（例如 iterators）。

凡是需要返回两个值的函数，通常可以利用 pair。

➢ 样例

```
pair<bool, double> result = do_a_calculation();
if (result.first)
  do_something_more(result.second);
else
  report_error();
```

➢ 何处定义

在早期的 HP 实现品中，pair 定义于头文件 <pair.h>。但在 C++ standard 中它被定义于头文件 <utility>。

➢ Template 参数

T1 pair 的第一元素的型别。
T2 pair 的第二元素的型别。

➢ 隶属何种 concept

当且仅当 T1 与 T2 都是 Assignable，则 pair 为 Assignable（p.83）；当且仅当 T1 与 T2 都是 Default Constructible，则 pair 为 Default Constructible（p.84）；当且仅当 T1 与 T2 都是 Equality Comparable，则 pair 为 Equality Comparable（p.85）；当且仅当 T1 与 T2 都是 LessThan Comparable，则 pair 为 LessThan Comparable（p.86）。

➢ 型别条件

无

➢ Public Base Classes

无

➢ 成员

pair 的某些成员并未定义于 Assignable、Default Constructible、Equality Comparable 或 LessThan Comparable 的需求条件中，而是 pair 所独具。我会在这类成员的前面标上 ※ 符号。

※ **pair::first_type**
pair 第一组件 T1 的型别。

※ **pair::second_type**
pair 第二个组件 T2 的型别。

※ **pair::pair(const T1& x, const T2& y)**
这是 constructor，用来构造一个 pair，其中由 x 构造出 first，由 y 构造出 second。

pair::pair(const pair&)
这是 copy constructor。描述于 Assignable。

※ **template <class U1, class U2>**
pair::pair(const pair<U1, U2>& p)
一般化的 constructor。如果 T1 具有单引数 constructor，其引数型别为 U1，T2 具有单引数 constructor，其引数型别为 U2，则此式能够构造出一个 pair，以 p.first 初始化 first 并以 p.second 初始化 second。

pair& pair::operator=(const pair&)
赋值操作符。描述于 Assignable。

※ **pair::first**
这是一个 public member variable，型别为 first_type，代表 pair 的第一元素。

※ **pair::second**
这是一个 public member variable，型别为 second_type，代表 pair 的第二元素。

bool operator==(const pair&, const pair&)
相等操作符。叙述于 Equality Comparable。

bool operator<(const pair& x, const pair& y)
比较操作符。如果「x.first 小于 y.first」或「x.second 小于 y.second 且 x.first 与 y.first 相等」，则表达式 x < y 为 true。描述于 LessThan Comparable。

※ **template <class T1, class T2>**
pair<T1, T2> make_pair(const T1& x, const T2& y)

辅助函数，用来产生一个 pair。等价于 pair<T1, T2>(x, y)，但较方便。

10.2 Iterator 基本要素（Iterator Primitives）

10.2.1 iterator_traits

iterator_traits<Iterator>

如同 3.1 节所描述，所有 iterators 都具有相关型别（associated types）。value type 便是其中之一，它代表 iterator 所指之 object 的型别。difference type 是其二，用来表示两个 iterators 之间的差距，是一个带正负号的整数（integral）型别。以型别为 int* 的指针为例，其 value type 为 int，difference type 为 ptrdiff_t。

作为 iterator 需求条件的一部分，STL 定义了一个机制，用来取得这些相关型别。例如，假设 I1 是 Input Iterator 的一个 model，其 value type 将是 iterator_traits<I1>::value_type。STL 预定义的 iterator 藉由三个 class template iterator_traits 版本，满足上述需求：(1) 一般版；(2) 指针特殊版；(3) const 指针特殊版。

以下便是 *partial specialization*（偏特化）的一个例子。这两个特殊版本自身就是 templates：

```
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;
};

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;
};
```

```
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;
};
```

当你撰写一个新的 iterator type I 时，你必须确定 iterator_traits<I> 可有适当的定义。有两个方法可以达到这个目标：第一，你可以自行定义你的 iterator，使它具有嵌套型别 I::value_type、I::difference_type 等等；第二，你可以针对你的型别明白地特化 iterator_traits。第一种方式比较方便，一个重要因素是你可以简单地继承自 base class iterator（p.185）。

iterator_traits class 对 STL 而言是相当新的项目。它并非 HP 原始实现品的一部分。事实上 HP 实现品也无法具有这样的东西，因为那时候 partial specialization 还没问世。HP STL 以不同方式来查询 iterator 的相关型别：三个名为 value_type、distance_type 和 iterator_category 的函数（请参考 3.1.6 节）。这些函数如今已不再是 C++ standard 的一部分了。

➢ 样例

下面这个泛型函数用来返回 nonempty range 中的最后一个元素。你不可能根据旧有的 value_type 函数定义一个如此的函数，因为这个函数的返回型别必须声明为 iterator 的 value type：

```
template <class InputIter>
typename iterator_traits<InputIter>::value_type
last_value(InputIter first, InputIter last) {
  typename iterator_traits<InputIter>::value_type result = *first;
  for (++first; first != last; ++first)
    result = *first;
  return result;
}
```

注意，这个例子只是简单说明如何使用 iterator_traits。我们并不建议用它来撰写泛型算法。在 Bidirectional Iterators 或 Random Access Iterators 所形成的 range 中，我们有更好的方法来找寻最后一个元素。

➢ 何处定义

iterator_traits 并不在 HP 的实现中。根据 C++ standard，它声明在头文件 <iterator>。

➢ Template 参数

Iterator　　被我们取相关型别者（一个 iterator type）。

➢ 隶属哪一个 concept

Default Constructible（p.84）、Assignable（p.83）。

➢ 型别条件

- `Iterator` 是 iterator concept Input Iterator（p.94）或 Output Iterator（p.96）的一个 model。

➢ Public Base Classes

无

➢ 成员

`iterator_traits` 的所有成员都是一些嵌套定义（nested typedef）。

`iterator_traits::iterator_category`
返回以下五者之一：`input_iterator_tag`、`output_iterator_tag`、`forward_iterator_tag`、`bidirectional_iterator_tag`、`random_access_iterator_tag`。所谓 iterator category 是指最明确的那个 iterator concept，本 iterator 即为该 concept 的一个 model。

`iterator_traits::value_type`
这是 `Iterator` 的 value type，正如定义于 Input Iterator 需求条件中的一样。注意，只有当 `Iterator` 是 Input Iterator 的一个 model 时才可以使用之；Output Iterator 不必具有 value type。

`iterator_traits::difference_type`
`Iterator` 的 difference type，正如定义于 Input Iterator 需求条件中的一样。注意，Output Iterator 不必具有 difference type。

`iterator_traits::pointer`
`Iterator` 的 pointer type，正如定义于 Input Iterator 需求条件中的一样。注意，Output Iterator 不必具有 pointer type。

`iterator_traits::reference`
`Iterator` 的 reference type，正如定义于 Input Iterator 需求条件中的一样。注意，Output Iterator 不必具有 reference type。

10.2.2 Iterator Tag Classes

```
struct output_iterator_tag { };
struct input_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 { };
```

这五个 ***iterator tag*** classes 完全是空的。它们不具有任何 member functions、member variables 或 nested types。正如其名称所提示，它们仅仅被当作 tag（标签）之用，用来表示 C++ 型别系统中的五种 iterator concepts。

Iterator tag classes 与 `iterator_traits` 有紧密的关系。对于任何一个 iterator type `I`，`iterator_traits<I>::iterator_category` 被定义为上述五个 tag classes 之一。

一般来说，iterator 的 category（类属）指的是其所隶属之最明确的 iterator concept，以 int* 为例，它是 Random Access Iterator 的 model。同时也是 Bidirection Iterator、Forward Iterator 等的 model。然而，这些 concepts 之中最具体而明确的是 Random Access Iterator，那是其他 iterator concepts 的强化（refinement），所以 iterator_traits<int*>::iterator_category 的结果是 random_access_iterator_tag。

➢ 样例

泛型算法 reverse（p.264）可以根据 Bidirectional Iterator 来实现，但其 Random Access Iterator 版更有效率。因此，reverse 利用 iterator_traits 和 tag class，再根据 iterator type 选出最适当的算法。这种布署系在编译期进行，不会招惹运行期的任何损失。

```
template <class BI>
void __reverse(BI first, BI last, bidirectional_iterator_tag) {
 while (true)
   if (first == last || first == --last)
      return;
   else
      iter_swap(first++, last);
}

template <class RAI>
void __reverse(RAI first, RAI last, random_access_iterator_tag) {
   while (first < last)
      iter_swap(first++, --last);
}

template <class BI>
void __reverse(BI first, BI last) {
 __reverse(first, last,
           typename iterator_traits<BI>::iterator_category());
}
```

➢ 何处定义

在 HP 实现品中，iterator tag class 声明于头文件 <iterator.h>。在 C++ standard 中它被声明于头文件 <iterator>。

➢ Template 参数

无

➢ 隶属何种 concept

Default Constructible、Assignable。

➢ **Public Base Classes**

Iterator tag structs 形成一个继承体系：forward_iterator_tag 衍生自 input_iterator_tag，bidirectional_iterator_tag 衍生自 forward_iterator_tag，random_access_iterator_tag 衍生自 bidirectional_iterator_tag。

你可能会好奇为什么 forward_iterator_tag 继承自 input_iterator_tag 而非 output_iterator_tag。基本上答案和「tag classes 运用继承机制」并没有太大关系。Iterator tag classes 的继承体系并不是很重要，它只是一个小小的便利手段而已，使得「根据 iterator category 来进行分派（*dispatch*）」的泛型算法的撰写工作更为容易。在此目的下，由于不需要「继承自 output_iterator_tag」，所以便将它略去。

10.2.3 distance

```
(1) template <class InputIterator>
    typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);

(2) template <class InputIterator, class Distance>
    void distance(InputIterator first, InputItreator last, Distance& n);
```

distance 是其他 STL 算法内部所用的一个 iterator 基本元素。它会找寻介于 first 与 last 之间的距离，也就是「将 first 累加，直到等于 last」所需的次数[19]。[first,last) 内的值无关紧要，因为 distance 只对 iterator 自身起作用 — 没有任何一个 iterator 会被提领其值。

两个版本的 distance 使用不同的接口做相同的事。版本 2 定义于 HP 原始发行的 STL [SL95] 实现品中，但其接口在标准化过程中有了一些改变。C++ standard [ISO98] 只包含版本 1。目前（1998）某些 STL 实现品只包含版本 1，某些只包含版本 2，某些则包含两者并提供旧接口过渡到新接口的方法。版本 1 单纯地返回 first 与 last 间的距离；版本 2 将 n 加上其距离量。

版本 2 的接口十分累赘。要找从 first 到 last 的距离，你需得产生一个局部变量。将这个变量初始化为 0，然后再调用 distance。这既不方便又容易出错，常犯的错误是忘了将局部变量初始为 0。这正是为什么要将接口改成版本 1 的原因。

STL 原始版本使用如此不方便之接口的原因是，distance 返回型别的声明有点棘手。如果我们要计算 [first,last) 之中与 value 相等的元素个数，则返回型别必须是个够大的整数（integral）型别。每个 iterator type 都有一个相关型别：***difference type***，可表示介于该型别之两个 iterators 间的距离。

---

[19] 这就是 distance 不定义在 Output Iterator 之中的原因。因为没有必要比较两个 Output Iterator 是否相等。

对内建指针而言，其 difference type 是 ptrdiff_t。distance 返回值的型别应该是 InputIterator 的 difference type。

声明这个返回型别的机制是 iterator_traits class(3.1 节)。如果 I 是 Input Iterator 的一个 model，则以下的嵌套型别便是 I 的 difference type：

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

HP STL 原始版本使用不同的方式，因为 iterator_traits 彼时尚未问世。因此，版本 2 将返回值存入使用者所提供的变量中。版本 2 中 Distance 型别的大小必须至少与 InputIterator 的 difference type 一样。

➢ 何处定义

在 HP 实现品中，distance 声明于头文件 <algo.h>。C++ standard 则将它声明于头文件 <iterator>。

➢ 型别条件

对于版本 1：

- InputIterator 是 Input Iterator 的一个 model。

对于版本 2：

- InputIterator 是 Input Iterator 的一个 model。
- Distance 是个整数（integral）型别，非负值，大到足以表示 InputIterator 的距离。

➢ 先决条件

对于版本 1：

- [first,last) 是个有效的 range。

对于版本 2：

- [first,last) 是个有效的 range。
- [first,last) 的元素个数不超过 Distance 型别的最大值。

➢ 复杂度

如果 InputIterator 是 Random Access Iterator 的 model，复杂度为常量时间；否则为线性时间。

➢ 样例

以下这个生硬不自然的例子显示如何使用 distance 第一版本：

```
int main() {
  list<int> L;
  L.push_back(0);
  L.push_back(1);

  assert(distance(L.begin(), L.end()) == L.size());
}
```

但 distance 的真正存在原因并不是为了如此琐屑的应用，而是作为其他泛型算法所需要的基本元素。举个例子，lower_bound（p.308）的实现细节中便会使用 distance：

```
template <class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value) {
  iterator_traits<ForwardIterator>::difference_type len
       = distance(first, last);
  iterator_traits<ForwardIterator>::difference_type half;
  ForwardIterator middle;

  while (len > 0) {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (*middle < value) {
      first = middle;
      ++first;
      len = len - half - 1;
    } else
      len = half;
  }
  return first;
}
```

10.2.4 advance

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

advance(i, n) 会将 iterator i 累加 n 步距离。如果 n > 0，那就等价于 ++i 执行 n 次（如果 InputIterator 隶属于 Random Access Iterator，会执行得更快）。如果 n < 0，那就等价于 --i 执行 n 次。如果 n == 0，本次调用等于无效。

当 InputIterator 属于 Bidirectional Iterator 时，n 才允许负值。

➢ 何处定义

HP 实现品中，advance 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <iterator>。

➢ 型别条件

- InputIterator 是 Input Iterator 的一个 model。
- Distance 是整数（integral）型别，可转换成 InputIterator 的 difference type。

➢ 先决条件

- i 是 nonsingular。
- 每个介于 i 与 i + n（含）的 iterators 都是 nonsingular。
- 如果 InputIterator 隶属于 Input Iterator 而非 Bidirection Iterator，n 必须不为负值。如果 InputIterator 隶属于 Bidirection Iterator 或 Random Access Iterator，此一先决条件便不适用。

➢ 复杂度

如果 InputIterator 是 Random Access Iterator 的 model，复杂度为常量时间；否则为线性时间。

➢ 样例

底下这个不甚自然的例子，告诉你如何利用 advance 来将 Forward Iterator 累加 n 次。

```
int main()
{
  slist<int> L;
  L.push_front(0);
  L.push_front(1);

  slist<int>::iterator i = L.begin();
  advance(i, 2);
  assert(i == L.end());
}
```

然而就像 distance（p.181）一样，advance 真正的存在原因是作为其他泛型算法的基本元素。例如 lower_bound 便是根据 distance 和 advance 写成，见 p.183。

10.2.5 Iterator Base Class

iterator<Category, T, Distance, Pointer, Reference>

iterator class 是个空的 class，不具备任何 member function 或 member variable，只有一些 nested typedef。

除非你要定义自己的 iterator class，否则其实不需要在意 iterator class。事实上即使你要定义自己的 iterator class，「运用 iterator」也只是众多途径之一罢了。

当你撰写新的 iterator class，你必须保证它可以适当地与 iterator_traits 合作。最简单的方式就是让你的 class 包含五个嵌套型别：iterator_category、value_type、difference_type、pointer、reference。你可以明白定义出这些型别，或是令你的 iterator class 继承自 iterator，后者会让其 template 引数成为上述五个 typedefs。

注意，HP STL 并不包含 iterator。取而代之的是，它包含五种不同的 iterator base class：input_iterator、output_iterator、forward_iterator、bidirectional_iterator、random_access_iterator。这五个 classes 如今已不再是 C++ standard 的一部分了。在 HP STL 中，你可以定义一个新的 Output Iterator，让它继承自：

```
output_iterator
```

或定义一个新的 Forward Iterator，让它继承自：

```
forward_iterator<T>
```

然而在 standard C++ 之中，欲定义一个新的 Output Iterator，你应该继承自：

```
iterator<output_iterator_tag, void, void, void, void>
```

或者，欲定义一个新的 Forward Iterator，你应该继承自：

```
iterator<forward_iterator_tag, T>。
```

➢ **样例**

```
class my_forward_iterator :
   public std::iterator<forward_iterator_tag, double>
{
  ...
};
```

这会令 my_forward_iterator 为一个 Forward Iterator，其 value type 为 double，difference type 为 ptrdiff_t，而 pointer 和 reference type 分别为 double* 和 double&。

不使用 iterator 一样可以做到相同的事，只要为 my_forward_iterator 定义一些嵌套型别即可：

```
class my_forward_iterator
{
public:
  typedef forward_iterator_tag iterator_category;
  typedef double              value_type;
  typedef ptrdiff_t           difference_type;
  typedef double*             pointer;
  typedef double&             reference;
  ...
};
```

一一明白地定义出嵌套型别，未免有点啰唆，但却会更清楚些。此外，有时可藉由明白的定义，得到更好的效率，因为在很多编译器上，空的 base class 需要某些额外空间（C++ standard 对此并不要求也不禁止）。如果你使用这样的编译器并以 iterator 衍生 my_forward_iterator，那么当你产生 my_forward_iterator 时会浪费一些空间。

➢ 何处定义

HP 原始实现品中并没有定义 iterator class。C++ standard 把它声明于头文件 <iterator>。

➢ Template 参数

Category	Iterator tag struct（p.179）系用来表示该 iterator 的 category（类属）：input_iterator_tag、output_iterator_tag、forward_iterator_tag、bidirectional_iterator_tag 或 random_access_iterator_tag。所谓 iterator category 是指该 iterator 所属之最具体明确的 concept。它又被定义为嵌套型别 iterator_category。
T	这是 iterator 的 value type，也就是 iterator 所指之 object 的型别。它又被定义为嵌套型别 value_type。
Distance	这是 iterator 的 difference type，可以表现两个 iterators 之间距离，是一个带正负号整数型别。它又被定义为嵌套型别 difference_type。**缺省值**为 ptrdiff_t。
Pointer	iterator 的 value type 的指针。pointer type 通常不是 T* 就是 const T*，前者针对 mutable iterator，后者针对 constant iterator。它又被定义为嵌套型别 pointer。**缺省值**为 T*。
Reference	iterator 的 value type 的 reference。reference type 通常不是 T& 就是 const T&，前者针对 mutable iterator，后者针对 constant iterator。它又被定义为嵌套型别 reference。**缺省值**为 T&。

➢ 隶属何种 concept

Default Constructible、Assignable。

➢ Public Base Classes

无

10.3 allocator

allocator<T>

几乎没有什么理由需要直接用到 allocator class。

除非你撰写新的 container class，否则 Allocator（p.166）的唯一出现时机是它被当作 container 的 template 参数 — 而你必须提及该 template 参数的唯一时机是当你想使用 allocator class 之外的某种东西。

STL 事先定义的所有 containers 都具有一个「身为 Allocator model」的 template 参数，缺省值为 allocator。例如 vector<int> 就等于 vector<int, allocator<int> >。

如果你想撰写新的 container class，并且想要将此 class 的动态内存管理方法参数化，你可以遵循上述方法，让该 class 拥有一个 template 参数，其缺省值为 allocator。

➢ 何处定义

在 HP 实现品中，一个差别甚多的 allocator 版本被定义于头文件 <allocator.h>。C++ standard 把它定义于头文件 <memory>。

➢ Template 参数

T　allocator 的 value type，代表 allocator 分配与归还内存空间之对象型别。

➢ 隶属何种 concept

Allocator（p.166）。

➢ 型别条件

无

➢ Public Base Classes

无

➢ 成员

以下是一般情况下 allocator<T> 的成员定义。有一个特化的 allocator<void>，只包含 value_type、pointer、const_pointer、rebind 等成员。

Class allocator 并没有什么特别的成员。所有成员都定义于 Allocator 需求条件中。

allocator::value_type

allocator 的 value type，T （描述于 Allocator 中。）

allocator::pointer

allocator 的 pointer type，T* （描述于 Allocator 中。）

allocator::const_pointer

allocator 的 const pointer type，const T* （描述于 Allocator 中。）

allocator::reference

allocator 的 reference type，T& （描述于 Allocator 中。）

allocator::const_reference

allocator 的 const reference type，const T& （描述于 Allocator 中。）

allocator::size_type

无正负号整数（integral）型别：size_t （描述于 Allocator 中。）

allocator::difference_type

有正负号整数（integral）型别：ptrdiff_t （描述于 Allocator 中。）

allocator::rebind

一个嵌套的（nested）class template。class rebind<U> 有唯一成员 other，那是一个 typedef，代表 allocator<U>。（描述于 Allocator 中。）

allocator::allocator()

Default constructor。（描述于 Allocator 中。）

allocator::allocator(const allocator&)

Copy constructor。由于 class allocator 的每一个 instance 都互相等价（equivalent），所以 default constructor 和 copy constructor 之间纯粹只是语法上的差别。（描述于 Allocator 中。）

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

一般化的 copy constructor。（描述于 Allocator 中。）

allocator::~allocator()

destructor。（描述于 Allocator 中。）

pointer allocator::address(reference x) const

返回某个 object 的地址。表达式 a.address(x) 等价于 &x。（描述于 Allocator 中。）

const_pointer allocator::address(const_reference x) const

返回某个 const object 的地址。表达式 a.address(x) 等价于 &x。（描述于 Allocator 中。）

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

分配内存，足以存储 n 个 T objects。第二引数是个提示。实现上可能会利用它来增进局部性（locality），或完全忽略之。（描述于 Allocator 中。）

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

归还先前以 allocate 分配的内存块。（描述于 Allocator 中。）

size_type allocator::max_size() const

返回 allocate 可成功分配的最大量。（描述于 Allocator 中。）

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

等价于 new((const void*) p) T(x)。（描述于 Allocator 中。）（译注：我认为不该有 const。）

void allocator::destroy(pointer p)

等价于 p->~T()。（描述于 Allocator 中。）

10.4 内存管理基本元素（Memory Management Primitives）

这一节的「算法」和其他所有的 STL 算法不同。它们作用于未初始化的存储空间，而非实际的 C++ objects。这些低阶行为主要用于协助 container classes 的实现。

本节三个最重要的算法是 uninitialized_copy、uninitialized_fill 和 uninitialized_fill_n。它们和算法 copy（p.233）、fill（p.249）、fill_n（p.250）关系十分密切。两者间的唯一真正差别在于，copy、fill 和 fill_n 会赋新值于已存在的 objects 身上，而低阶算法 uninitialized_copy、uninitialized_fill 和 uninitialized_fill_n 会构造出新的 objects。

10.4.1 construct

```
template <class T1, class T2>
void construct(T1* p, const T2& value);
```

在 C++ 中，new 操作符首先分配 object 的内存，然后调用构造函数在该内存位置上构造 object。有时候将这两个行为细分开来，颇有用途，特别是当你要实现 container class 的时候。

如果 p 指向已分配但尚未初始化之内存，则 construct(p, value) 会在 p 所指地方产生一个型别 T1 的 object。引数 value 会传给 T1 的 constructor。

表达式 construct(p, value) 本质上等价于表达式 new(p) T1(value)。

➢ 何处定义

HP 实现品中，construct 声明在头文件 <algo.h>，SGI 实现品将它声明于头文件 <memory>。C++ standard 并不涵盖 construct。

➢ 型别条件

- T1 有一个 constructor，接受一个型别为 T2 的引数。

➢ 先决条件

- p 是有效指针，其所指内存大小至少为 sizeof(T1)。
- p 所指的内存未经初始化。也就是说没有任何 object 构造于 p 所指位置。

➢ 样例

```
string* sptr = (string*) malloc(sizeof(string));
construct(sptr, "test");
assert(strcmp(sptr->c_str(), "test") == 0);
```

10.4.2 destroy

```
(1) template <class T> void destroy(T* pointer);

(2) template <class ForwardIterator>
    void destroy(ForwardIterator first, ForwardIterator last);
```

在 C++ 中，delete 操作符会先调用某个 object 的 destructor，然后归还该 object 所在之内存空间。有时候将这两个行为细分开来，颇有用途（特别是当你要实现 container class 的时候，因为 container 通常要管理它所含有之 C++ objects 的内存空间，以及某些未初始化空间）。你可以利用 destroy 来调用某个 object destructor，而不释放该 object 所占内存。

destroy 第一版会调用 destructor T::~T() 来析构 pointer 所指之 object。该 object 所占内存并不会被归还，也不会被其他 object 重新利用。

destroy 第二版析构 range [first,last) 中的所有 objects。它等于是针对 [first,last) 中的每一个 iterator i 调用 destroy(&*i)。

➢ 何处定义

HP 实现品把 destroy 声明于头文件 <algo.h>，SGI 实现品则是声明于头文件 <memory>。C++ standard 并不涵盖 destroy。

➢ 型别条件

对于第一版本：

- destructor ~T() 必须可取用。

对于第二版本：

- ForwardIterator 是 Forward Iterator 的一个 model。
- ForwardIterator 是可变的（mutable）。
- ForwardIterator 的 value type 具有可取用之 destructor。

➢ 先决条件

对于第一版本：

- pointer 指向型别为 T 之有效 object。

对于第二版本：

- [first,last) 是个有效 range。
- [first,last) 中的每个 iterators 都指向有效的 object。

➢ 复杂度

第二版本的运行期复杂度是线性的；它恰恰调用 destructor (last-first) 次。

➢ 样例

```
class Int {
public:
  Int(int x) : val(x) { }
  int get() { return val; }
private:
  int val;
};

int main()
{
  Int A[] = { Int(1), Int(2), Int(3) };

  destroy(A, A + 3);
  construct(A + 0, Int(10));
```

```
    construct(A + 1, Int(11));
    construct(A + 2, Int(12));
  }
```

10.4.3 uninitialized_copy

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

算法 uninitialized_copy 是一个低阶的 STL 组件，能够将内存的分配与 object 的构造行为分离开来。如果输出区间 [result, result+(last-first)) 中的每一个 iterators 都指向未初始化局部，则 uninitialized_copy 会使用 copy constructor，为[first,last) 内的所有 objects 产生一份复制品，放进输出区间中。换句话说，针对输入区间内的每一个 iterator i，uninitialized_copy 会调用 construct(&*(result+(i-first)),*i)，产生 *i 的复制品，放于输出区间的相对位置上。

如果你需要实现一个 container class，uninitialized_copy 这样的函数对你而言是很有用的。例如 container 的 range constructor 通常需要两个步骤：

1. 分配内存块，足以包含 range 中的所有元素。
2. 使用 uninitialized_copy，在该内存块上产生 container 的元素。

由于 uninitialized_copy 主要用于 container，所以它有一个不常见的负担：必须能够使用 uninitialized_copy 来撰写 "exception-safe" 代码。更明确地说，必须能够撰写一种 container，使得当 uninitialized_copy 调用某个 constructor 时，即使丢出 exception 也不会造成内存漏洞（memory leak）。C++ standard 要求 uninitialized_copy 具有 "*commit*" 或 "*rollback*" 语义：要不就「构造出所有必要元素」，要不就（当有任何一个 copy constructors 失败时）「不构造任何东西」。下面是一份合理的实现：

```
template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
                   ForwardIterator result)
{
  ForwardIterator cur = result;
  try {
    for ( ; first != last; ++first, ++cur)
      contruct(&*cur, *first);
    return cur;
  }
  catch(...) {
    destroy(result, cur);
    throw;
```

```
  }
}
```

➢ 何处定义

HP STL 将 uninitialized_copy 声明于头文件<algo.h>。C++ standard 将它声明于 <memory>。

➢ 型别条件

- InputIterator 是 Input Iterator 的一个 model。
- ForwardIterator 是 Forward Iterator 的一个 model。
- ForwardIterator 是可变的（mutable）。
- ForwardIterator 的 value type 具有单引数 constructor，其引数型别为 InputIterator 的 value type。

➢ 先决条件

- [first,last) 是个有效的 range。
- [result,result + (last - first)) 是个有效的 range。
- [result, result+(last-first)) 中的每一个 iterators 都指向一块未初始化内存，大小足以包含 ForwardIterator value type 的实值。

➢ 复杂度

属于线性复杂度。恰调用 constructor (last-first) 次。

➢ 样例

```
class Int {
public:
  Int(int x) : val(x) { }
  // Int 不具有 default constructor。
  int get() { return val; }
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  uninitialized_copy(A1, A1 + N, A2);
}
```

10.4.4 uninitialized_fill

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

和 uninitialized_copy 一样，uninitialized_fill 也是一个低阶的 STL 组件，能够将内存分配与 object 构造行为分离开来。如果 range [first,last) 中的每个 iterators 都指向未初始化的内存，那么 uninitialized_fill 会在该 range 中产生 x 之复制品。也就是说对于 range 中的每个 iterator i，uninitialized_fill 会调用 construct(&*i, x)，在该位置产生 x 复制品。

如果你希望实现一个 constructor，能够以某个特定区间构造 container，则前述的 uninitialized_copy 很有帮助。如果你希望实现一个 constructor，能够将 container 中的每个元素初始为 x 值，则现在这个 uninitialized_fill 很有帮助。和 uninitialized_copy 一样，uninitialized_fill 必须具备 "*commit*" 或 "*rollback*" 语义，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 exception，uninitialized_fill 必须能够将已产生的所有元素析构掉。

➢ 何处定义

HP STL 将 uninitialized_fill 声明于头文件 <algo.h>。C++ standard 将它声明于 <memory>。

➢ 型别条件

- ForwardIterator 是 Forward Iterator 的一个 model。
- ForwardIterator 是可变的（mutable）。
- ForwardIterator 的 value type 具有单引数 constructor，其引数型别为 T。

➢ 先决条件

- [first,last) 是个有效的 range。
- [first,last) 中的每个 iterator 指向一块未初始化内存，足以包含 ForwardIterator value type 的实值。

➢ 复杂度

线性。恰调用 constructor (last-first) 次。

➢ 样例

```
class Int {
public:
  Int(int x) : val(x) {}
```

```
  // Int 不具有 default constructor。
  int get() { return val; }
private:
  int val;
};

int main()
{
  const int N = 137;

  Int val(46);
  Int* A = (Int*) malloc(N * sizeof(Int));
  uninitialized_fill(A, A + N, val);
}
```

10.4.5 uninitialized_fill_n

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

算法 uninitialized_fill_n 很像 uninitialized_fill，也是一种低阶的 STL 组件，能够将内存分配与 object 构造行为分离开来。它会将 range 中的元素初始为相同的值。两者之间的差别在于 uninitialized_fill 操作于「两个 iterators 表现的 range」身上，而 uninitialized_fill_n 操作于「一个 iterator 加上一个元素个数所表现的 range」身上，这使得 uninitialized_fill_n 可和 Output Iterators 搭配。

如果 [first, first+n) 中的每一个 iterators 都指向未初始化的内存，uninitialized_fill_n 会调用 copy constructor，在该 range 内产生 x 的复制品。也就是说，面对 [first,first + n) 中的每个 iterator i，uninitialized_fill_n 会调用 construct(&*i, x)，在对应位置处产生 x 的复制品。

和 uninitialized_copy 以及 uninitialized_fill 一样，uninitialized_fill_n 具有 "*commit*" 或 "*rollback*" 语义：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 exception，uninitialized_fill_n 必须能够析构已产生的所有元素。

➢ 何处定义

HP STL 将 uninitialized_fill_n 声明于头文件 <algo.h>。C++ standard 将它声明于 <memory>。

➢ 型别条件

- ForwardIterator 是 Forward Iterator 的一个 model。
- ForwardIterator 是可变的（mutable）。

- Size 是整数（integral）型别，可转换成 ForwardIterator 之 difference type。
- ForwardIterator 的 value type 具有单引数 constructor，引数型别为 T。

➢ **先决条件**

- n 是 nonsingular。
- [first,first + n) 是个有效的 range。
- [first,first + n) 中的每一个 iterator 指向一块未初始化内存，足以包含 ForwardIterator value type 之实值。

➢ **复杂度**

线性。恰调用 constructor n 次 。

➢ **样例**

```
class Int {
public:
  Int(int x) : val(x) {}
  // Int 不具有 default constructor。
  int get() { return val; }
private:
  int val;
};

int main()
{
  const int N = 137;

  Int val(46);
  Int* A = (Int*) malloc(N * sizeof(Int));
  uninitialized_fill_n(A, N, val);
}
```

10.5 临时缓冲区（Temporary Buffers）

有些算法，例如 stable_sort 和 inplace_merge，是所谓的 "*adaptive*"（有适应能力的）。它们会尝试分配额外的临时空间来放置中介结果。如果分配成功，这样的 "*adaptive*" 算法便变换为另一种具有较佳运行期复杂度的方法。

当你撰写某个 *adaptive* 算法时，可以好好利用 get_temporary_buffer 和 return_temporary_buffer 这对函数。正如其名称所暗示，get_temporary_buffer 会分配未初始化的内存，而 return_temporary_buffer 会归还该内存块。至于 get_temporary_buffer 应该使用 malloc 或 operator new，并没有任何规定。

这些函数应该只用于内存真正是「临时性的」情况下。如果一个函数以 get_temporary_buffer 分配内存，则在该函数返回前，必得调用 return_temporary_buffer 来归还内存。

不幸的是 get_temporary_buffer 和 return_temporary_buffer 并不像看起来那么有用。这些函数的运用是一个复杂的过程。你从 get_temporary_buffer 手上获得的内存是未初始化的，所以必须以 uninitialized_fill（或其他类似函数）来做初使化动作。同样道理，你必须在利用 return_temporary_buffer 归还内存之前，先将缓冲区中的所有 objects 析构掉。

之所以会有这么一些复杂步骤，部分原因是，我们很难使用 get_temporary_buffer 和 return_temporary_buffer 撰写出 "execption-safe" 代码。当你撰写一个 "*adaptive*" 算法，而 execption safety 对你而言又很重要的话，你几乎一定得用本节所说的临时缓冲区（temporary buffer）class 来取代。constructor 应该以 get_temporary_buffer 分配缓冲区，并在其内进行初始化，而 destructor 必须析构所有元素并归还该缓冲区。

10.5.1 get_temporary_buffer

```
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t len);
```

函数 get_temporary_buffer 可分配临时性内存空间。引数 len 是欲获得之缓冲区大小（元素个数）。例如 get_temporary_buffer<T>(len) 会返回一个缓冲区，此区针对 T object 排列（*aligned*），并足以包含 len 个 T objects。

调用情节颇不寻常：必须明白提供 template 参数 T，因为它无法由 get_temporary_buffer 的 template 引数推导出来。此情节称为「function template arguments 之 explicit specification（显式指定）」，属于比较新的语言特性。在 1998 年之时，并非所有编译器都支持此一特性。

如果你的编译器尚未支持 explicit specification，你可以使用 HP STL 版的 get_temporary_buffer。虽然它已不再是标准的一部分，但通常会为了向下兼容性而存在。这个版本需要两个引数，第二引数是型别为 T* 的挂名引数（dummy argument），纯粹作为 template 参数推导之用。你可以写 get_temporary_buffer(len, (T*) 0) 来获得含有 len 个 T objects 的缓冲区。

不论哪种情况，引数 len 都只是一个索求项目（request），而非一个必要条件。是的，你所获得的缓冲区可能小于你所要求。其用意是让 get_temporary_buffer 返回能力所及的最大缓冲区而不损害性能。

返回值是一个 pair P，其第一元素是个指针，指向分配而得之内存，第二元素表示该缓冲区的大小。P.first 所指向的缓冲区足以包含 P.second 个 T objects。如果 P.second 为 0，表示没有分配任何缓冲区。既然如此，P.first 就是一个 null 指针。

注意 P.first 指向的是未初始化的内存，而非型别为 T 之 object。你可以利用 uninitialized_copy 或 uninitialized_fill 或其他类似方法在该缓冲区产生 objects。

➢ 何处定义

HP STL 将 get_temporary_buffer 定义于头文件 <tempbuf.h>，后者被含入于 <algo.h>。C++ standard 将它声明于头文件 <memory>。

➢ 先决条件

- len 大于 0。

➢ 样例

```
int main()
{
  pair<int*, ptrdiff_t> P = get_temporary_buffer<int>(10000);
  int* buf = P.first;
  ptrdiff_t N = P.second;
  uninitialized_fill_n(buf, N, 42);
  int* result = find_if(buf, buf + N,
                        bind2nd(not_equal_to<int>(), 42));
  assert(result == buf + N);
  return_temporary_buffer(buf);
}
```

10.5.2 return_temporary_buffer

```
template <class T> void return_temporary_buffer(T* p);
```

函数 return_temporary_buffer 负责归还以 get_temporary_buffer 分配来的内存。

➢ 何处定义

HP STL 将 return_temporary_buffer 定义于头文件 <tempbuf.h>，后者被含入于 <algo.h>。C++ standard 将它声明于头文件 <memory>。

➢ 先决条件

- 引数 p 是一个指针，指向 get_temporary_buffer(ptrdiff_t, T*) 分配得来的内存。
- 构造于该内存块中的任何 objects，在 return_temporary_buffer 被调用之前都应该已被析构。

第 11 章

「不改变操作对象之内容」的算法
Nonmutating Algorithms

本章叙述 STL 的基本算法，它们作用于 iterators 所形成的 range 之上，但不会更改这些 iterators 所指元素的内容。这些算法返回 range 所含元素的某种信息。它们主要用于审阅，而非用来更改内容。

11.1 线性查找（Linear Search）

11.1.1 find

```
template <class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

算法 find 会在 iterators range 内执行线性查找，寻找指定之值 value。更明确地说，它会返回在 [first,last) 中第一个满足「*i == value」的 iterator i。如果没有这样的 iterator，就返回 last。

线性查找（Knuth [Knu98b] 的用词是 sequential search，循序查找）是寻找 range 内某个特定值的最直接方法。进行程序是：先测试第一个元素是否具有我们想要的值，如是就停止查找，否则移到下一个元素。重复这些动作，直到没有元素可以测试为止。

有时候线性查找能以稍微不同的方式写成。如果你在尾端放一个等于 value 的「哨兵（sentinel）」，那就保证查找动作一定可以停止，而你可以省略「是否为 range 尾端」的测试动作。然而这种变形并不适合完全泛型的算法，因为彼等需要在「查找程序可以更改其 input range」的前提下完成 — 而 STL 并不做这样的假设。因为或许没有任何元素超过 range [first,last)，亦或许 range 内含 constant iterator。是的，STL 的 find 并不使用这种「哨兵」技术。

➢ 何处定义

HP STL 将 find 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- EqualityComparable 是 Equality Comparable 的 model。
- InputIterator 是 Input Iterator 的 model。
- 相等性（equality）定义于 EqualityComparable object 与 InputIterator value type 之 object 两者间。

➢ 先决条件

[first,last) 是有效的 range。

➢ 复杂度

线性。最多有 (last-first) 次比较行为，测试相等与否。

➢ 样例

```
int main()
{
  list<int> L;
  L.push_back(3);
  L.push_back(1);
  L.push_back(7);

  list<int>::iterator result = find(L.begin(), L.end(), 7);
  assert(result == L.end() || *result == 7);
}
```

11.1.2 find_if

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

如同 find（p.199），算法 find_if 能在 iterators range 内执行线性查找。两者的差别在于 find_if 更为一般化。它并不查找某特定值，而是查找满足某种条件的元素。它会返回 range [first,last) 之中第一个满足「pred(*i) 为 true」的 iterator i。如果没有这样的 iterator 存在，就返回 last。

注意，你可以利用 find_if 来找寻第一个等于某特定值的元素。毕竟 function object pred 可以测试其引数是否等于该值。但如果 find_if 只是这样的用途，就不是那么有价值，因为这只是以更累赘的方式来做等价于 find 的事。更一般化的性质才是它更让人感兴趣的地方。

有一个十分常见的行为是：查询具有某特定键值（key）的元素，例如以「姓」查询某人，或以 process ID 查询操作系统中的某个 process。很多算法书籍，包括 Knuth 的著作 [Knu98b]，将所谓「查找」定义为「查询具有某已知键值(key)之未知元素」。你不能用 find 来做这件事，因为它只能测试 object 的相等性，但你能利用 find_if 来达成目标。你可以使用一个 function object，让它萃取出其引数的键值（key），然后将该键值与查找对象的键值相比较。

（由于 find 和 find_if 是如此相像，你可能会好奇为什么要令它们拥有不同的名称。为什么不做两个重载版本的 find 就好呢？这纯粹是技术上的原因。两个算法都具有相同个数的引数，而且其第三个引数都是 template 参数。因此如果两个算法同名，便无法正确调用它们，因为编译器无法知道你要的是哪一个。）

➢ 何处定义

HP STL 将 find_if 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- Predicate 是 Predicate 的 model。
- InputIterator 是 Input Iterator 的 model。
- InputIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

[first,last) 是有效的 range。

对于 range [first,last) 中的每个 iterator i，*i 属于 Predicate domain 之中。

➢ 复杂度

线性。最多套用 (last-first) 个 Pred。

➢ 样例

以下找寻整数数组中的第一个 0。此处我把 find_if 当作稍不方便的 find 来用。

```
int main()
{
  int A[] = {4, 1, 0, 3, 2, 0, 6};
  const int N = sizeof(A) / sizeof(int);

  int* p = find_if(A, A + N,
                   bind2nd(equal_to<int>(), 0));

  cout << "Index of first zero = " << p - A << endl;
}
```

以下找寻键值为 2 的第一个元素。此例中的元素是 pair，而 pair 的第一元素便视为 key。更一般化的情况下，你可以将任何数据结构的字段视为 key；你所要做的就是让 function object 能够将该字段抽取出来。

```
int main()
{
  typedef pair<int, char*> Pair;

  vector<Pair> V;
  V.push_back(Pair(3, "A"));
  V.push_back(Pair(7, "B"));
  V.push_back(Pair(2, "C"));
  V.push_back(Pair(0, "D"));
  V.push_back(Pair(6, "E"));

  vector<Pair>::iterator p =
    find_if(V.begin(), V.end(),
            compose1(bind2nd(equal_to<int>(), 2), select1st<Pair>()));
  cout << "Found: "
       << "<" << (*p).first << "," << (*p).second << ">"
       << endl;
}
```

以下找寻整数链表中的第一个正数。此例展示 find_if 的一般性。此例并不涉及查询某特定值或是查找具有特定键值（key）的元素。

```
int main()
{
  list<int> L;
  L.push_back(-3);
  L.push_back(0);
  L.push_back(3);
  L.push_back(-2);
  L.push_back(7);

  list<int>::iterator result = find_if(L.begin(), L.end(),
                                       bind2nd(greater<int>(), 0));
  assert(result == L.end() || *result > 0);
}
```

11.1.3 adjacent_find

```
(1) template <class ForwardIterator>
    ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last);
```

```
(2) template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate bindary_pred);
```

如同 find（p.199），adjacent_find 算法会在 iterators range 中执行线性查找。两者间的差异在于 find 是针对单个元素查找，adjacent_find 却是针对两个相邻元素查找。

adjacent_find 有两个版本。版本一查找两个相等的紧邻元素。版本二则更一般化，查找两个紧邻元素是否满足某种条件，该条件系以 Binary Predicate 定义。

adjacent_find 第一版本返回 [first,last) 内第一个满足「i 和 i+1 都有效，且 *i==*(i+1)」的 i。如果没有这样的 iterator 存在，就返回 last。

版本二返回 [first,last) 中第一个满足「binary_pred(*i, *(i+1)) 为 true」的 i。如果没有这样的 iterator 存在，就返回 last。

➢ 何处定义

HP STL 将 adjacent_find 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 是 Forward Iterator 的 model。
- ForwardIterator 之 value type 需为 Equality Comparable。

第二个版本：

- ForwardIterator 是 Forward Iterator 的 model。
- ForwardIterator 之 value type 可转换为 BinaryPredicate 的第一与第二引数型别。

➢ 先决条件

- [first,last) 是有效的 range。

➢ 复杂度

与元素个数成线性关系。更明确地说，如果 range [first,last) 不是空的，则 adjacent_find 最多执行 last-first-1 次比较动作。如果 [first,last) 是空的，不执行任何比较动作。

➢ **样例**

以下找寻第一个重复元素。本例使用第一版本：

```
int main()
{
  int A[] = {1, 2, 3, 3, 4, 5};
  const int N = sizeof(A) / sizeof(int);

  const int* p = adjacent_find(A, A + N);

  if (p != A + N)
    cout << "Duplicate element: " << *p << endl;
}
```

以下找寻第一个大于其后继者的元素 — 也就是寻找 range 之内「不以递增次序排列」的第一个元素位置。本例运用第二版本：

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8};
  const int N = sizeof(A) / sizeof(int);

  const int* p = adjacent_find(A, A + N, greater<int>());

  if (p == A + N)
    cout << "The range is sorted in ascending order." << endl;
  else
    cout << "Element " << p - A << " is out of order: "
         << *p << " > " << *(p + 1) << "." << endl;
}
```

11.1.4 find_first_of

```
(1) template <class InputIterator, class ForwardIterator>
    InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);

(2) template <class InputIterator, class ForwardIterator,
              class BinaryPredicate>
    InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate comp);
```

算法 `find_first_of` 极类似 `find`（p.199）。它会在 Input Iterator range 内执行线性查找。两者间的差异在于 `find` 查找某个特定值，而 `find_first_of` 可查找任何数量的值。更明确地说，`find_first_of` 会在 range `[first1,last1)` 内查找任何出现于 `[first2,last2)` 内的元素。

如果 find 类似 C 标准函数库中的 strchr，那么 find_first_of 就有如 strpbrk。

find_first_of 有两个版本，差别在于它们「比较元素是否相等」的方式。第一版本采用 operator==，第二版本采用外界提供的 function object comp。第一版本返回 [first1,last1) 内的一个 iterator i，它第一个满足「对 [first2,last2) 中的某个 iterator j，*i == *j」。第二版本返回 [first1,last1) 内的一个 iterator i，它第一个满足「对 [first2,last2) 中的某个 iterator j，表达式 comp(*i,*j) 为 true」。

➢ 何处定义

HP STL 将 find_first_of 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 是 Input Iterator 的 model [20]。
- ForwardIterator 为 Forward Iterator 之 model。
- InputIterator 之 value type 为 Equality Comparable，可以与 ForwardIterator 之 value type 作相等性比较。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- ForwardIterator 为 Forward Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- InputIterator 之 value type 可转换为 BinaryPredicate 的第一引数型别。
- ForwardIterator 之 value type 可转换为 BinaryPredicate 的第二引数型别。

➢ 先决条件

- [first1,last1) 是有效的 range。
- [first2,last2) 是有效的 range。

➢ 复杂度

最多执行 (last1-first1) * (last2-first2) 次比较动作。

---

[20] 根据 C++ standard，两个 iterator types 必须都是 Forward Iterator 的 model，但这其实没有必要。find_first_of 允许第二个 iterator 为 Forward Iterator model，而第一个 iterator 为任何类型之 Input Iterator。

➢ **样例**

就像 strpbrk 一样，find_first_of 的某个用途是拿来寻找字符串中的 whitespace 字符。不论 space（空格）、tab（跳格定位）或 newline（新行）都是 whitespace 字符。

```
int main()
{
  const char* WS = " \t\n";
  const int n_WS = strlen(WS);

  char* s1 = "This sentence contains five words.";
  char* s2 = "OneWord";

  char* end1 = find_first_of(s1, s1 + strlen(s1),
                             WS, WS + n_WS);
  char* end2 = find_first_of(s2, s2 + strlen(s2),
                             WS, WS + n_WS);

  printf("First word of s1: %.*s\n", end1 - s1, s1);
  printf("First word of s2: %.*s\n", end2 - s2, s2);
}
```

11.2 子序列匹配（Subsequence Matching）

11.2.1 search

```
(1) template <class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

(2) template <class ForwardIterator1, class ForwardIterator2,
             class BinaryPredicate>
    ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate binary_pred);
```

算法 search 和 find（p.199）、find_if（p.200）类似，可以在某个 range 内做查找动作。不同的是 find 和 find_if 企图查找单个元素，而 search 则是企图查找整个 subrange（子区间）；它试图在 [first1,last1)内寻找 [first2,last2)。也就是说 search 会寻找 [first1,last1) 的某个子序列，使得该子序列与 [first2,last2) 进行元素的一一比较时是相同的。search 返回一个 iterator，指向该子序列的开头，如果这样的子序列不存在，就返回 last1。

版本一会返回 [first1,last1-(last2-first2)] 中的第一个 iterator i，使得以满足「对于 [first2,last2) 中的每个 iterator j，*(i+(j-first2)) == *j」。这个条件看起来非常复杂，但意义很简单：在以 i 开头的 subrange 中，每个元素都必须与 [first2,last2) 内的对应元素相同。

上述所说的是 [first1,last1-(last2-first2)] 而非 [first1,last1) ，原因是我们要寻找的 subrange 必须等于整个 [first2,last2)。subrange 不能比该 range 还长，所以除非 last1-i >= last2-first2，否则 iterator i 不能为此等 subrange 的开头。你可以调用 search，令其引数满足 last1-first1 < last2-first2，其查找结果一定是错的。

除了两元素相等与否同的判断方式外，版本二与版本一相同。版本一利用 operator==，而版本二利用外界提供的 function object binary_pred。版本二返回 [first1,last1-(last2-first2)] 中的第一个 iterator i，使得以满足「对于 [first2,last2) 中的每一个 iterator j，表达式 binary_pred (*(i+(j-first2)), *j) 为 true」。

➢ 何处定义

HP STL 将 search 声明于头文件 <algo.h>。C++ standard 将声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator1 为 Forward Iterator 之 model。
- ForwardIterator2 为 Forward Iterator 之 model。
- ForwardIterator1 之 value type 为 Equality Comparable 之 model。
- ForwardIterator2 之 value type 为 Equality Comparable 之 model。
- ForwardIterator1 value type object 可与 ForwardIterator2 value type object 做相等比较。

第二个版本：

- ForwardIterator1 为 Forward Iterator 之 model。
- ForwardIterator2 为 Forward Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- ForwardIterator1 之 value type 可以转换为 BinaryPredicate 的第一引数型别。
- ForwardIterator2 之 value type 可以转换为 BinaryPredicate 的第二引数型别。

➢ 先决条件

- [first1,last1) 是有效的 range。
- [first2,last2) 是有效的 range。

➢ 复杂度

最坏情况下的复杂度与「本文（text）长度」以及「查找模式（pattern）」成线性关系（最多执行 (last1-first1) * (last2-first2) 次比较动作），但这种情况几乎不可能发生。平均复杂度与 last1-first1 成线性关系。

➢ 样例

以下查找字符串中的某个子字符串。这是 serach 的主要用途之一：

```
int main()
{
  const char S1[] = "Hello, world!";
  const char S2[] = "world";
  const int N1 = strlen(S1);
  const int N2 = strlen(S2);

  const char* p = search(S1, S1 + N1, S2, S2 + N2);
  if (p != S1 + N1)
    printf("Found substring \"%s\" at charactor %d of string \"%s\".\n",
           S2, p - S1, S1);
  else
    printf("Couldn't find substring.\n");
}
```

以下找寻三数形成的子序列，三数的最末一个阿拉伯数字分别为 1, 2, 3。

```
template <class Integer>
struct congruent {
  congruent(Integer mod) : N(mod) {}
  bool operator()(Integer a, Integer b) const {
    return (a - b) % N == 0;
  }

  Integer N;
};

int main()
{
  int A[10] = {23, 46, 81, 2, 43, 19, 14, 98, 72, 51};
  int digits[3] = {1, 2, 3};

  int *seq = search(A, A + 10, digits, digits + 3,
                    congruent<int>(10));
  if (seq != A + 10) {
    cout << "Subsequence: ";
    copy(seq, seq + 3, ostream_iterator<int>(cout, " "));
    cout << endl;
  }
```

```
    else
      cout << "Subsequence not found" << endl;
  }
  // 输出结果为:
  // Subsequence: 81 2 43
```

11.2.2 find_end

```
(1) template <class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

(2) template <class ForwardIterator1, class ForwardIterator2,
              class BinaryPredicate>
    ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate binary_pred);
```

find_end 的名字是个错误。它其实比较像 search（p.206）而非 find，比较精确的名称应该是 search_end。

和 search 一样，find_end 会在 [first1,last1) 之中查找与 [first2,last2) 相同的子序列。它们的差异在于 search 找寻的是第一个这样的子序列，find_end 找寻的是最后一个这样的子序列。它返回一个 iterator，指向该子序列的头。如果不存在这样的子序列，就返回 last1。

版本一会返回 [first1,last1-(last2-first2)] 中的最后一个 iterator i，此 i 满足「对于 [first2,last2) 中的每个 iterator j，*(i+(j-first2)) == *j」。这个条件看起来非常复杂。它的意义其实很简单：在以 i 开头的 subrange 中，每个元素必须与 [first2,last2) 中对应的元素相同。

和 search 一样，上述的 range 为 [first1,last1-(last2-first2)] 而非[first1,last1)，原因是我们要寻找的 subrange 是等于整个 [first2,last2)。subrange 不能比该整个 range 还长，所以除非 last1-i >= last2-first2，否则 iterator i 不能为如此的 subrange 的开头。你可以调用 find_end，令其引数满足 last1-first1 < last2-first2，其返回值一定是 last1。

除了「两元素是否相等」的判断方式外，版本二与版本一相同：版本一采用 operator==，版本二采用外界提供的 function object binary_pred。版本二返回 [first1,last-(last2-first2)] 中的最后一个 iterator i，此 i 满足「对于 [first2,last2) 中的每一个 iterator j，表达式 binary_pred(*(i+(j-first2)), *j) 为 true」。

➢ 何处定义

HP STL 将 find_end 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator1 为 Forward Iterator 之 model。
- ForwardIterator2 为 Forward Iterator 之 model。
- ForwardIterator1 之 value type 为 Equality Comparable 之 model。
- ForwardIterator2 之 value type 为 Equality Comparable 之 model。
- ForwardIterator1 之 value type 可以与 ForwardIterator2 之 value type 做相等测试。

第二个版本：

- ForwardIterator1 为 Forward Iterator 之 model。
- ForwardIterator2 为 Forward Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- ForwardIterator1 之 value type 可以转换为 BinaryPredicate 的第一引数型别。
- ForwardIterator2 之 value type 可以转换为 BinaryPredicate 的第二引数型别。

➢ 先决条件

- [first1,last1) 是有效的 range。
- [first2,last2) 是有效的 range。

➢ 复杂度

比较次数与 (last1-first1)*(last2-first2) 成比例。如果 ForwardIterator1 和 ForwardIterator2 皆是 Bidirectional Iterator 的 model，则平均复杂度是线性的。最坏情况下，复杂度最多为 (last1-first1) * (last2-first2) 次比较动作。

➢ 样例

```
int main()
{
  char* s = "executable.exe";
  char* suffix = "exe";

  const int N = strlen(s);
  const int N_suf = strlen(suffix);
```

```
  char* location = find_end(s, s + N, suffix,
                            suffix + N_suf);

  if (location != s + N) {
    cout << "Found a match for " << suffix << " within " << s
         << endl;
    cout << s << endl;

    int i;
    for (i = 0; i < (location - s); ++i)
      cout << ' ';
    for (i = 0; i < N_suf; ++i)
      cout << '^';
    cout << endl;
  }
  else
    cout << "No match for " << suffix << " within " << s << endl;
}
```

11.2.3 search_n

```
(1) template <class ForwardIterator, class Integer, class T>
    ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                             Integer count, const T& value);

(2) template <class ForwardIterator, class Integer, class T,
              class BinaryPredicate>
    ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                             Integer count, const T& value,
                             BinaryPredicate binary_pred);
```

算法 search_n 查找 [first,last) 之中由 count 个相邻元素形成的子序列，其中所有元素都等于 value。它返回一个 iterator，指向这个子序列的起始点。如果这样的子序列不存在，就返回 last。

注意，count 允许为 0。0 个元素的子序列有其明确定义。不论 value 为何，每个 range 都包含「0 个元素之子序列，其内所有元素等于 value」。如果调用 search_n 时指定 count 为 0，查找动作一定成功且返回值一定是 first。

search_n 有两个版本，其间差异在于两个元素是否相等的判断方法。版本一采用 operator==，版本二采用外界提供的 function object binary_pred。

版本一会返回 [first,last-count] 中的第一个 iterator i，此 i 满足「对于 [i,i+count) 中的每个 iterator j，*j == value」。版本二会返回 [first,last-count] 中的第一个 iterator i，此 i 满足「对于 [i,i+count) 中的每个 iterator j，binary_pred(*j, value) 为 true」。

和 search（p.206）一样，上述 range 为 [first,last-count] 而非只是 [first,last)，原因是我们要寻找的 subrange 长度为 count。subrange 不能比该整个 range 还长，所以除非 last-i >= count，否则 iterator i 不能为如此 subrange 的开头。你可以调用 search_n，令其引数符合 last-first < count，其返回值一定是 last。

➢ 何处定义

HP STL 将 search_n 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- Integer 为整数型别。
- T 为 Equality Comparable 之 model。
- ForwardIterator 之 value type 为 Equality Comparable 之 model。
- ForwardIterator 之 value type 的 object 可以与型别 T 之 object 做相等测试。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- Integer 为整数型别。
- T 为 Equality Comparable 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- ForwardIterator 之 value type 可以转换为 BinaryPredicate 的第一引数型别。
- T 可以转换为 BinaryPredicate 的第二引数型别。

➢ 先决条件

- [first,last) 是有效的 range。
- count >= 0。

➢ 复杂度

线性。search_n 最多执行 last-first 次比较[21]。

---

[21] C++ standard 加上了 count*(last-first) 这样的比较动作上限次数，但这没有必要。不管 count 的值为何，search_n 没有理由审视 [first,last) 中的任何元素一次以上。

➤ **样例**

```
bool eq_nosign(int x, int y) { return abs(x) == abs(y); }

void lookup(int* first, int* last, size_t count, int val) {
  cout << "Searching for a sequence of "
       << count
       << " '" << val << "'"
       << (count != 1 ? "s: " : ": ");
  int* result = search_n(first, last, count, val);
  if (result == last)
    cout << "Not found" << endl;
  else
    cout << "Index = " << result - first << endl;
}

void lookup_nosign(int* first, int* last, size_t count, int val) {
  cout << "Searching for a (sign-insensitive) sequence of "
       << count
       << " '" << val << "'"
       << (count != 1 ? "s: " : ": ");
  int* result = search_n(first, last, count, val, eq_nosign);
  if (result == last)
    cout << "Not found" << endl;
  else
    cout << "Index = " << result - first << endl;
}

int main()
{
  const int N = 10;
  int A[N] = {1, 2, 1, 1, 3, -3, 1, 1, 1, 1};

  lookup(A, A+N, 1, 4);
  lookup(A, A+N, 0, 4);
  lookup(A, A+N, 1, 1);
  lookup(A, A+N, 2, 1);
  lookup(A, A+N, 3, 1);
  lookup(A, A+N, 4, 1);

  lookup(A, A+N, 1, 3);
  lookup(A, A+N, 2, 3);
  lookup_nosign(A, A+N, 1, 3);
  lookup_nosign(A, A+N, 2, 3);
}
```

输出结果：

```
Searching for a sequence of 1 '4': Not found
Searching for a sequence of 0 '4's: Index = 0
Searching for a sequence of 1 '1': Index = 0
Searching for a sequence of 2 '1's: Index = 2
```

```
Searching for a sequence of 3 '1's: Index = 6
Searching for a sequence of 4 '1's: Index = 6
Searching for a sequence of 1 '3':  Index = 4
Searching for a sequence of 2 '3's: Not found
Searching for a (sign-insensitive) sequence of 1 '3':  Index = 4
Searching for a (sign-insensitive) sequence of 2 '3's: Index = 4
```

11.3 计算元素个数（Counting Elements）

11.3.1 count

```
(1) template <class InputIterator, class EqualityComparable>
    typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last,
          const EqualityComparable& value);

(2) template <class InputIterator, class EqualityComparable, class Size>
    void count(InputIterator first, InputIterator last,
               const EqualityComparable& value,
               Size& n);
```

算法 count 可以计算 [first,last) 之中与 value 相等的元素个数。

count 有两个版本，做的事相同，但接口不同。版本二定义于 HP STL [SL95] 中，但在标准化过程中接口有了些许变动。C++ standard [ISO98] 只包含版本一。目前（1998）某些 STL 实现品只包含第一个版本，某些只包含第二个版本，某些则是两个都包含。

第一个版本返回 [first,last) 内与 value 相等的元素个数。第二个版本的接口比较复杂：将 [first,last) 内符合「*i == value」的 iterator i 个数加入 n 之中。

第二个版本的接口很累赘。如果你想要计算与某值相等的元素个数，你必须先产生一个局部变量，将它初始化为 0，然后调用 count。很明显地这不方便又容易出错（常犯的错误是忘了将该局部变量初始化为 0），这也就是为什么要将接口改成第一个版本的原因。

如同 distance（p.181）的情况一样，原始 STL 使用如此不方便接口的原因是：原始的 HP STL 并未包含 iterator_traits（3.1 节）。

➢ 何处定义

HP STL 将 count 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- EqualityComparable 为 Equality Comparable 之 model。
- InputIterator 之 value type 为 Equality Comparable 之 model。
- InputIterator value type 的 object 可以与 EqualityComparable value type 的 object 做相等测试。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- EqualityComparable 为 Equality Comparable 之 model。
- Size 为一整数型别，大到足以表示 InputIterator distance type 的非负数值。
- InputIterator 之 value type 为 Equality Comparable 之 model。
- InputIterator value type 的 object 可以与 EqualityComparable value type 的 object 做相等测试。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。

第二个版本：

- [first,last) 是有效的 range。
- n 加上「与 value 相等」之元素个数，不会超过 Size 型别的最大值。

➢ 复杂度

线性。恰恰执行 last-first 次比较动作。

➢ 样例

以下计算某 range 内元素值为 0 的元素个数，使用版本二。

```
int main()
{
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  int result = 0;
  count(A, A + N, 0, result);
  cout << "Number of zeros: " << result << endl;
}
```

以下使用版本一，重新做一次。

```
int main()
{
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  cout << "Number of zeros: " << count(A, A + N, 0) << endl;
}
```

11.3.2 count_if

```
(1) template <class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last,
             Predicate pred);

(2) template <class InputIterator, class Predicate, class Size>
    void count_if(InputIterator first, InputIterator last,
                  Predicate pred,
                  Size& n);
```

算法 count_if 很类似 count（p.214），但却更一般化。它不再是计算与某特定值相等的元素个数，而是计算在 [first,last) 内满足某种条件的元素个数。该条件系由外界提供的 function object pred 来表现。换句话说，count_if 计算 range 之中满足「pred(*i) 为 true」的 iterator i 个数。

count_if 的两个版本做同样的事，为什么拥有两个接口，原因和 count 一样。版本二定义于 HP STL [SL95] 中，版本一定义于 C++ standard [ISO98] 中。某些 STL 实现品只包含第一版本，某些只包含第二版本，某些则是两者都包含。

版本一返回 [first,last) 内满足 pred 的元素个数，也就是说它返回[first,last) 之中满足「pred(*i) 为 true」的 iterator i 个数。版本二则将 [first,last) 内满足「pred(*i) 为 true」之 iterator i 的数量加到 n 身上。

➢ 何处定义

HP STL 将 count_if 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- Predicate 为 Predicate 之 model。
- InputIterator 之 value type 可转换为 Predicate 之引数型别。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- Predicate 为 Predicate 之 model。
- Size 为一整数型别，可持有 InputIterator distance type 的值。
- InputIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。

第二个版本：

- [first,last) 是有效的 range。
- n 加上符合 pred 条件之元素个数后，不会超过 Size 型别的最大值。

➢ 复杂度

线性。恰恰执行 last-first 次的 pred 动作。

➢ 样例

以下计算 range 之中元素为 0 的个数，使用旧接口（第二版本）。这里没有理由使用 count_if，因为 count 更方便。本例的唯一目的是要显示 count 是 count_if 的一个特例。

```
int main()
{
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  int result = 0;
  count_if(A, A + N, bind2nd(equal_to<int>(), 0), result);
  cout << "Number of zeros: " << result << endl;
}
```

以下使用新接口（第一版本）做相同的事。

```
int main()
{
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  cout << "Number of zeros: "
       << count_if(A, A + N, bind2nd(equal_to<int>(), 0))
       << endl;
}
```

和 find_if（p.200）的情况一样，计算「与某值相等」之元素个数，具有更令人感兴趣的一般性：计算某些「键值与某值相等」的元素个数。下面这个例子的元素是 pair。pair 的第一元素被视为 key。

```
int main()
{
  typedef pair<int, char*> Pair;

  vector<Pair> V;
  V.push_back(Pair(3, "A"));
  V.push_back(Pair(7, "B"));
  V.push_back(Pair(2, "C"));
  V.push_back(Pair(3, "D"));
  V.push_back(Pair(0, "E"));
  V.push_back(Pair(6, "E"));

  cout << "Number of elements with key equal to 3: "
       << count_if(V.begin(), V.end(),
                   compose1(bind2nd(equal_to<int>(), 3),
                            select1st<Pair>()))
       << endl;
}
```

以下计算 range 内的偶元素个数。

```
int main()
{
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  cout << "Number of event element: "
       << count_if(A, A + N,
                   compose1(bind2nd(equal_to<int>(), 0),
                            bind2nd(modulus<int>(), 2)))
       << endl;
}
```

11.4 for_each

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last,
                       UnaryFunction f);
```

算法 for_each 将 function object f 套用于 [first,last) 内的每个元素上，并忽略 f 的返回值（如果有的话）。套用动作是向前进行的，也就是说是从 first 到 last。返回值是套用于每个元素身上的那个 function object。

返回值常被忽略，因为 for_each 通常只是用来执行一连串动作。然而有时候返回值很有用。function object 不必得是单纯的函数指针，它也可以是使用者定义的 class。更明确地说是 function object 可具有局部状态（local state）。这意味着 for_each 不单只是执行一连串的动作，还可以更一般化。例如，function object 可以计算被调用的次数，或者它可以有一个状态标志（status flag），用来表示是否每次调用都成功。

➢ 何处定义

HP STL 将 for_each 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- InputIterator 为 Input Iterator 之 model。
- UnaryFunction 为 Unary Function 之 model。
- InputIterator 之 value type 可转换为 UnaryFunction 之引数型别。

➢ 先决条件

- [first,last) 是有效的 range。

➢ 复杂度

线性。恰恰套用 UnaryFunction 共 last-first 次。

➢ 样例

以下打印一系列元素，用的是一个能够打印其引数的 unary function object。

```
template <class T> struct print : public unary_function<T, void>
{
  print(ostream& out) : os(out), count(0) {}
  void operator() (T x) { os << x << ' '; ++count; }
  ostream& os;
  int count;
};

int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  print<int> P = for_each(A, A + N, print<int>(cout));
  cout << endl << P.count << " object printed." << endl;
}
```

这个例子并不十分有趣，因为使用 ostream_iterator 一样可以达到相同的效果。

稍微复杂一点的例子是利用 C 标准函数库中的 system 函数，依序执行操作系统指令。这种情况下，我们传给 for_each 的 function object 是一个函数指针。

```
int main()
{
  char* commands[] = {"uptime", "pwd", "ls"};
  const int N = sizeof(commands) / sizeof(char*);

  for_each(commands, commands + N, system);
}
```

这个例子几乎可以被无限扩展。例如，我们可以不直接使用 system，而改用一个「可检查 system 之返回值，以测知调用是否成功」的 function object。如果更有野心一些，我们可以使用 function object 来封装不同操作系统间的差异性。

11.5 比较两个 Ranges

11.5.1 equal

```
(1) template <class InputIterator1, class InputIterator2>
    bool equal(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2);

(2) template <class InputIterator1, class InputIterator2,
              class BinaryPredicate>
    bool equal(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, BinaryPredicate binary_pred);
```

如果将 [first1,last1) 和 [first2,first2+(last1-first1)) 内的元素一一比较，结果都相等的话，equal 便返回 true，否则返回 false。

equal 有两个版本，差别在于第一版本采用 operator== 来比较元素，第二版本采用外界提供的 function object binary_pred。

当且仅当（if and only if）[first1,last1) 中的每一个 iterator i 都满足 *i == *(first2 + (i-first1))，则第一个版本返回 true。当且仅当 [first1,last1) 中的每一个 iterator i 都满足 binary_pred(*i, *(first2 + (i-first1)))，则第二个版本返回 true。

➢ 何处定义

HP STL 将 equal 声明于头文件 <algo.h> 并定义于 <algobase.h> 中。C++ standard 将它声明于头文件 <algorithm>。

Generic Programming and the STL

➢ 型别条件

第一个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `InputIterator1` value type 为 Equality Comparable 之 model。
- `InputIterator2` value type 为 Equality Comparable 之 model。
- `InputIterator1` value type 的 object 可与 `InputIterator2` value type 的 object 做相等测试。

第二个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `BinaryPredicate` 为 Binary Predicate 之 model。
- `InputIterator1` 之 value type 可转换为 `BinaryPredicate` 之第一引数型别。
- `InputIterator2` 之 value type 可转换为 `BinaryPredicate` 之第二引数型别。

➢ 先决条件

- `[first1,last1)` 是有效的 range。
- `[first2, first2+(last1-first1))` 是有效的 range。

➢ 复杂度

线性。最多比较 `(last1-first1)` 次。

➢ 样例

以下比较两个 ranges 是否相等。

```
int main()
{
  int A1[] = {3, 1, 4, 1, 5, 9, 3};
  int A2[] = {3, 1, 4, 2, 8, 5, 7};
  const int N = sizeof(A1) / sizeof(int);

  if (equal(A1, A1 + N, A2))
    cout << "Equal" << endl;
  else
    cout << "Not equal" << endl;
}
```

以下比较两个 ranges 是否相等，大小写不论。

```
inline bool eq_nocase(char c1, char c2) {
  return toupper(c1) == toupper(c2);
}

int main()
{
  const char* s1 = "This is a Test";
  const char* s2 = "This is a test";
  const int N = strlen(s1);

  if (equal(s1, s1 + N, s2, eq_nocase))
    cout << "Equal" << endl;
  else
    cout << "Not equal" << endl;
}
```

11.5.2 mismatch

```
(1) template <class InputIterator1, class InputIterator2>
    pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

(2) template <class InputIterator1, class InputIterator2,
              class BinaryPredicate>
    pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2,
             BinaryPredicate binary_pred);
```

算法 mismatch 返回 [first1,last1) 与 [first2,first2+(last1-first1))之间第一个「元素值不同」的位置。该位置可由 [first1,last1) 的一个 iterator 和 [first2,first2+(last1-first1)) 的一个 iterator 描述之，mismatch 返回的便是一个 pair，包含这两个 iterators[22]。

mismatch 有两个版本，其间差异在于元素相等与否的测试方式：第一版本采用 operator==，第二版本采用外界提供的 function object binary_pred。

[22] 注意，这和 equal（p.220）多么类似啊！唯一差异在于当两个 ranges 不同时所发生的事情：equal 只会返回一个 bool，但 mismatch 会告知它们哪里不同。表达式 equal(f1, l1, f2) 等价于表达式 mismatch(f1, l1, f2).first == l1，而这正是 equal 的一个合理实现方式。

第一版本找寻 [first1,last1) 之中第一个满足「*i != *(first2 + (i-first1))」的 iterator i。返回值为 pair，其第一元素为 i，第二元素为 first2+(i-first1)。如果没有这样的 i 存在，返回值是一个 pair，其第一元素为 last1，第二元素为 first2+(last1-first1)。

第二版本找寻在 [first1,last1) 之中第一个满足「binary_pred(*i,*(first2+(i-first1))) 为 false」的 iterator i。返回值为 pair，其第一元素为 i，第二元素为 first2+(i-first1)。如果没有这样的 iterator i 存在，返回值是一个 pair，其第一元素为 last1 而第二元素为 first2+(last1-first1)。

➢ 何处定义

HP STL 将 mismatch 声明于头文件 <algo.h> 并定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- InputIterator1 value type 为 Equality Comparable 之 model。
- InputIterator2 value type 为 Equality Comparable 之 model。
- InputIterator1 value type 的 object 可以与 InputIterator2 value type 的 object 做相等测试。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- InputIterator1 value type 可转换为 BinaryPredicate 之第一引数型别。
- InputIterator2 value type 可转换为 BinaryPredicate 之第二引数型别。

➢ 先决条件

- [first1,last1) 是有效的 range。
- [first2, first2+(last1-first1)) 是有效的 range。

➢ 复杂度

线性。最多比较 (last1-first1) 次。

➢ 样例

以下找出两个 ranges 内第一个「元素值不相同」的位置。

```
int main()
{
  int A1[] = {3, 1, 4, 1, 5, 9, 3};
  int A2[] = {3, 1, 4, 2, 8, 5, 7};
  const int N = sizeof(A1) / sizeof(int);

  pair<int*, int*> result = mismatch(A1, A1 + N, A2);
  if (result.first == A1 + N)
    cout << "The two ranges do not differ." << endl;
  else {
    cout << "First mismatch is in position "
         << result.first - A1 << endl;
    cout << "Values: " << *(result.first) << ", "
                       << *(result.second) << endl;
  }
}
```

以下找寻两个字符串中第一个「字符不相同」的位置，大小写不论。

```
inline bool eq_nocase(char c1, char c2) {
  return toupper(c1) == toupper(c2);
}

int main()
{
  const char* s1 = "This is a Test";
  const char* s2 = "This is a test";
  const int N = strlen(s1);

  pair<const char*, const char*> result =
    mismatch(s1, s1 + N, s2, eq_nocase);

  if (result.first == s1 + N)
    cout << "The two strings do not differ" << endl;
  else {
    cout << "The strings differ starting at character "
         << result.first - s1 << endl;
    cout << "Trailing part of s1: " << result.first << endl;
    cout << "Trailing part of s2: " << result.second << endl;
  }
}
```

11.5.3 lexicographical_compare

```
(1) template <class InputIterator1, class InputIterator2>
    bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

(2) template <class InputIterator1, class InputIterator2,
              class BinaryPredicate>
    bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            BinaryPredicate comp);
```

若以字典排序法做比较，[first1,last1) 小于 [first2,last2) 的话，算法 lexicographical_compare 返回 true，反之返回 false。

字典排序比较方式（lexicographical comparison）意指「字典式」（一个元素接着一个元素）的排序。也就是说如果 *first1 小于 *first2，则 [first1,last1) 小于 [first2,last2)，如果 *first1 大于 *first2，则 [first1,last1) 大于 [first2,last2)。如果两个开头元素相等，则比较下一个元素，依此类推。如果第一 range 内的每个元素都相等于第二 range 内的相对元素，但第二 range 包含更多元素，那么第一 range 被视为小于第二 range。

[first1,last1) 和 [first2,last2) 不必具有相同的元素个数。这很罕见，因为大部分「作用于两个 ranges 身上」的 STL 算法，例如 equal（p.220）和 copy（p.233），都要求两个 ranges 的长度必须相同。大部分算法以三个 iterators 来指示两个 ranges，但 lexicographical_compare 却使用四个 iterators。

lexicographical_compare 有两个版本，其间差别在于如何定义某元素是否小于另一个元素。第一版本采用 operator<，第二版本采用外界提供的 function object comp。

➢ 何处定义

HP STL 将 lexicographical_compare 声明于头文件 <algo.h> 并定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- InputIterator1 之 value type 为 LessThan Comparable 之 model。

- InputIterator2 之 value type 为 LessThan Comparable 之 model。
- 如果 v1 是 InputIterator1 value type 的值，v2 是 InputIterator2 value type 的值，则 v1 < v2 以及 v2 < v1 都必须有定义。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- InputIterator1 value type 可转换为 BinaryPredicate 之第一与第二引数型别。
- InputIterator2 value type 可转换为 BinaryPredicate 之第一与第二引数型别。

➢ 先决条件

- [first1,last1) 是有效的 range。
- [first2,last2) 是有效的 range。

➢ 复杂度

线性。最多比较 2 * min(last1-first1, last2-first2) 次。

➢ 样例

以字典排序法比较两个数字形成的 ranges。

```
int main()
{
  int A1[] = {3, 1, 4, 1, 5, 9, 3};
  int A2[] = {3, 1, 4, 2, 8, 5, 7};
  int A3[] = {1, 2, 3, 4};
  int A4[] = {1, 2, 3, 4, 5};

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3) / sizeof(int);
  const int N4 = sizeof(A4) / sizeof(int);

  bool C12 = lexicographical_compare(A1, A1 + N1,
                                     A2, A2 + N2);

  bool C34 = lexicographical_compare(A3, A3 + N3,
                                     A4, A4 + N4);

  cout << "A1[] < A2[]: " << (C12 ? "true" : "false") << endl;
  cout << "A3[] < A4[]: " << (C34 ? "true" : "false") << endl;
}
```

以字典排序法序比较两个字符串，大小写不论。

```
inline bool lt_nocase(char c1, char c2) {
  return toupper(c1) < toupper(c2);
}

int main()
{
  const char* s1 = "abc";
  const char* s2 = "ABC";
  const char* s3 = "abcDEF";
  const int N1 = 3, N2 = 3, N3 = 6;

  printf("%s < %s : %c\n",
         s1, s2,
         lexicographical_compare(s1, s1 + N1,
                                 s2, s2 + N2,
                                 lt_nocase) ? 't' : 'f');

  printf("%s < %s : %c\n",
         s2, s3,
         lexicographical_compare(s2, s2 + N2,
                                 s3, s3 + N3,
                                 lt_nocase) ? 't' : 'f');
}
// 输出结果:
// abc < ABC : f
// ABC < abcDEF : t
```

11.6 最大值与最小值

11.6.1 min

```
(1) template <class LessThanComparable>
    const LessThanComparable& min(const LessThanComparable& a,
                                  const LessThanComparable& b);

(2) template <class T, class BinaryPredicate>
    const T& min(const T& a, const T& b, BinaryPredicate comp);
```

大多数 STL 算法都作用于元素所形成的 ranges 身上，min 是少数作用在「以引数传入之单一 objects」身上的算法之一。min 返回两个引数中较小的一个。如果比不出大小，就返回第一引数。

min 有两个版本，其差别在于如何定义某元素是否小于另一个。第一版本采用 operator<，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 min 声明于头文件 <algo.h> 并定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- LessThanComparable 为 LessThan Comparable 之 model。

第二个版本：

- BinaryPredicate 为 Binary Predicate 之 model。
- T 可转换为 BinaryPredicate 之第一与第二引数的型别。

➢ 样例

```
int main()
{
  const int x = min(3, 9);
  assert(x == 3);

  int a = 3;
  int b = 3;

  const int& result = min(a, b);
  assert(&result == &a);
}
```

11.6.2 max

```
(1) template <class LessThanComparable>
    const LessThanComparable& max(const LessThanComparable& a,
                                  const LessThanComparable& b);

(2) template <class T, class BinaryPredicate>
    const T& max(const T& a, const T& b, BinaryPredicate comp);
```

大多数 STL 算法都作用于元素所形成的 ranges 身上，max 是少数作用在「以引数传入之单一 objects」身上的算法之一。max 返回两个引数中较大的一个。如果比不出大小，就返回第一引数。

max 有两个版本，其差别在于如何定义某元素是否小于另一个。第一版本采用 operator<，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 max 声明于头文件 <algo.h> 并定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- LessThanComparable 为 LessThan Comparable 之 model。

第二个版本：

- BinaryPredicate 为 Binary Predicate 之 model。
- T 可转换为 BinaryPredicate 之第一与第二引数的型别。

➢ 样例

```
int main()
{
  const int x = max(3, 9);
  assert(x == 9);

  int a = 3;
  int b = 3;

  const int& result = max(a, b);
  assert(&result == &a);
}
```

11.6.3 min_element

```
(1) template <class ForwardIterator>
    ForwardIterator
    min_element(ForwardIterator first, ForwardIterator last);

(2) template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator
    min_element(ForwardIterator first, ForwardIterator last,
                BinaryPredicate comp);
```

算法 min_element 寻找 [first,last) 内的最小元素。它返回 [first,last) 之内第一个满足「range 之内再没有其他 iterator 所指之值小于 *i」[23] 的 iterator i。如果 [first,last) 是空的，返回值为 last。

[23] 我们可不能将条件说成「[first,last) 内指向最小元素的那个 iterator」，因为最小元素可能不只一个。是的，range 内的最小元素可能超过一个以上。举个例子，如果 [first,last) 中的每个元素都相等，min_element 会返回 first。这是 min 函数的一般化结果 — 如果 min 的两个引数相等，则返回第一引数。

min_element 有两个版本，差别在于如何定义某元素是否小于另一个。第一版本采用 operator<，第二版本采用 function object comp。

第一版本返回 [first,last) 中第一个满足「对于 [first,last) 内的的每个 iterator j，*j<*i 为 false」的 iterator i。第二版本返回 [first,last) 中第一个满足「对于 [first,last) 内的每个 iterator j，comp(*j, *i) 为 false」的 iterator i。

➢ 何处定义

HP STL 将 min_element 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 之 value type 为 LessThan Comparable。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- ForwardIterator value type 可转换为 BinaryPredicate 之第一与第二引数的型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。对于非空的 range，恰恰比较 (last-first)- 1 次。

➢ 样例

```
int main()
{
  list<int> L;
  generate_n(front_inserter(L), 1000, rand);

  list<int>::const_iterator it = min_element(L.begin(), L.end());
  cout << "The smallest element is " << *it << endl;
}
```

11.6.4 max_element

```
(1) template <class ForwardIterator>
    ForwardIterator
    max_element(ForwardIterator first, ForwardIterator last);

(2) template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator
    max_element(ForwardIterator first, ForwardIterator last,
                BinaryPredicate comp);
```

算法 max_element 寻找 [first,last) 内的最大元素。它返回 [first,last) 之内第一个满足「range 之内再没有其他 iterator 所指之值大于 *i」[24]的 iterator i。如果 [first,last) 是空的，返回值为 last。

max_element 有两个版本，差别在于如何定义某元素是否小于另一个。第一版本采用 operator<，第二版本采用 function object comp。

第一版本返回 [first,last) 中第一个满足「对于 [first,last) 内的的每个 iterator j，*i<*j 为 false」的 iterator i。第二版本返回 [first,last) 中第一个满足「对于 [first,last) 内的每个 iterator j，comp(*i, *j) 为 false」的 iterator i。

➢ 何处定义

HP STL 将 max_element 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 之 value type 为 LessThan Comparable。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- BinaryPredicate 为 Binary Predicate 之 model。
- ForwardIterator value type 可转换为 BinaryPredicate 之第一与第二引数的型别。

---

[24] 我们可不能将条件说成「[first,last) 内指向最大元素的那个 iterator」，因为最大元素可能不只一个。是的，range 内的最大元素可能超过一个以上。举个例子，如果 [first,last) 中的每个元素都相等，max_element 会返回 first。这是 max 函数的一般化结果 — 如果 max 的两个引数相等，则返回第一引数。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。对于非空的 range，恰恰比较 (last-first)- 1 次。

➢ 样例

```
int main()
{
  list<int> L;
  generate_n(front_inserter(L), 1000, rand);

  list<int>::const_iterator it = max_element(L.begin(), L.end());
  cout << "The largest element is " << *it << endl;
}
```

第 12 章

「会改变操作对象之内容」的算法
Basic Mutating Algorithms

和第 11 章所谈的「审阅式」算法不同，本章所叙述的基本算法作用于一个 iterator 或多个 iterators 所形成的 ranges 上，并更改某些 iterators 所指的元素。这些算法包括：(1) 从某个 range 复制元素至另一个 range；(2) 为 range 中的每个 iterators 赋值；(3) 将某个值覆盖为另一个值。

本章的算法有一个相同的特质：它们更改的是 iterator 所指之值，而非 iterator 自身。或说（以不同观点而言）这些算法会更改元素所形成的 range，而非 iterator range。如果你将 [first,last) 传入本章的算法，即使该算法可能更动 *first，也不能改变 first 自身。它既不能改变「first+1 必紧邻于 first 之后」这个事实，也不能改变 [first,last) 的元素个数。

当你看到这般叙述，此一性质对你或许显而易见，然而在某些算法中，它有时候会带来意料之外的涵意。

本章中的某些算法如 partition，系作用于单一 range 身上。其他算法如 copy，作用于两个不同的 ranges 身上，一个是 input range，其值不会被更动，另一个是 output range，那是算法赋值运算结果的地方。

有时候 STL 对于相同的算法会提供两种形式。例如 reverse 会将某个 range 反转（reverse）后取而代之，reverse_copy 则会将 input range 以相反方向复制到 output range 上。命名原则有其一致性：以 x_copy 为名的算法是以 x 为名的算法的「复制形式（copying form）」。

12.1 拷贝某个区间（Copying Ranges）

12.1.1 copy

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

算法 copy 可将 input range [first,last) 内的元素复制到 output range[result,result+(last-first)) 上。也就是说，它会执行赋值动作*result = *first、*(result+1) = *(first+1)……依此类推。返回值为 result+(last-first)。

copy 对其 template 引数所要求的条件非常宽松。其 input range 只需由 Input Iterators 构成即可，而其 output range 只需由 Output Iterators 构成即可。这意味你几乎可以使用 copy 将任何类型的来源端复制到任何类型的目的端。

对于每个从 0 到 last-first（不含）的整数 n，copy 执行赋值动作 *(result+n) = *(first+n)。赋值动作是向前（亦即累加 n）推进的[25]。

如同本章所有算法一样，copy 只会更改 [result,result+(last-first)) 之中的 iterator 所指值，而非 iterator 自身。它会为 output range 内的元素赋予新值，而不是产生新的元素。它不能改变 output range 的 iterators 个数。此一事实的个中理由是，copy 不能直接用来将元素安插于空的 Container 之中。

如果你想要将元素安插于 Sequence，要不就明白使用其 insert memeber function，要不就使用 copy 并搭配 insert_iterator adapter（p.251）。

➢ 何处定义

HP STL 将 copy 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的型别。

➢ 先决条件

- [first,last) 是有效的 range。
- result 并非 [first,last) 中的 iterator。

---

[25] 如果 input range 和 output range 重叠，赋值顺序需要多加讨论。当 result 位于 [first,last) 之内，也就是说如果 output range 的开头与 input range 重叠，我们便不能使用 copy。但如果 output range 的尾端与 input range 重叠，就可以使用 copy。copy_backward 的限制恰恰相反。如果两个 ranges 完全不重叠，当然毫无问题地两个算法都可以用。如果 result 是个 ostream_iterator（p.357）或其他某种「其语义视赋值顺序而定」的 iterator，那么赋值顺序一样会成为一个需要讨论的问题。

- 有足够的空间容纳所有被复制的元素。更正式地说是：[result,reulst+(last-first)) 是个有效的 range。

➢ 复杂度

线性。恰恰执行 last-first 次赋值（assignments）动作。

➢ 样例

以下将 vector 的元素复制到 slist 身上。在这个例子中，我们明白产生出一个 slist，令它足以容纳 vector 的所有元素。

```
int main()
{
  char A[] = "abcdefgh";
  vector<char> V(A, A + strlen(A));

  slist<char> L(V.size());
  copy(V.begin(), V.end(), L.begin());
  assert(equal(V.begin(), V.end(), L.begin()));
}
```

以下将 vector 的元素填满空的 list。由于不能直接复制到 list（因为这个 list 之中没有任何可以被赋值的元素），因此我们搭配 output iterator adapter 来使用 copy，使元素得以附加于这个 list 身上。

```
int main()
{
  char A[] = "abcdefgh";
  vector<char> V(A, A + strlen(A));

  list<char> L;
  copy(V.begin(), V.end(), back_inserter(L));
  assert(equal(V.begin(), V.end(), L.begin()));
}
```

以下使用 ostream_iterator（p.357），将 list 内的所有元素复制到标准输出设备上。

```
int main()
{
  list<int> L;
  L.push_back(1);
  L.push_back(3);
  L.push_back(5);
  L.push_back(7);

  copy(L.begin(), L.end(),
       ostream_iterator<int>(cout, "\n"));
}
```

12.1.2 copy_backward

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);
```

算法 copy_backward 和 copy（p.233）一样，可将 input range 内的元素复制到 output range。Input range 为 [first,last)，而 output range 为[result-(last-first),result)。它会执行赋值动作 *(result-1) = *(last-1)、*(result-2) = *(last-2)……依此类推。对于从 0 到 last-first（不含）的每个整数 n，copy_backward 会执行赋值动作 *(result-n-1) = *(last-n-1)，由 input range 的尾端至其前端（也就是累加 n）[26]。

返回值为 result-(last-first)。

copy 和 copy_backward 有三个不同点。第一，copy 只要求其 input range 由 Input Iterators 构成而其 output range 由 Output Iterators 构成。相较之下 copy_backward 就严厉多了，其 input range 和 output range 都必须由 Bidirectional Iterators 构成。第二，这两个算法以执行方向相反，copy 由前往后复制，copy_backward 则如其名地由后往前复制。第三，和大多数 STL 算法一样，copy 系以「range 起头之 iterator」表现其 output range，而 copy_backward 则以 iterator result 代表其 output range 的尾端。这种表示法并不常见。在所有 STL 算法中，这是唯一一个以「指向 range 尾端」之单一 iterator 来表示某个 range 者。

➢ 何处定义

HP STL 将 copy_backward 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- BidirectionalIterator1 和 BidirectionalIterator2 皆为 Bidirectional Iterator 之 model。
- BidirectionalIterator1 value type 的值可转换为 BidirectionalIterator2 value type 的值。

---

[26] 如果 input range 和 output range 重叠，赋值顺序需要多加讨论。当 result 位于 [first,last) 之内，也就是说如果 output range 的尾端与 input range 重叠，我们便不能使用 copy_backward。但如果 output range 的开头与 input range 重叠，就可以使用 copy_backward。copy 的限制恰恰相反。如果两个 ranges 完全不重叠，当然毫无问题地两个算法都可以用。

➢ 先决条件

- [first,last) 是有效的 range。
- result 并非 [first,last) 中的 iterator。
- 有足够空间容纳所有被复制的元素。更正式的说是：[reulst-(last-first),result) 是个有效的 range。

➢ 复杂度

线性。恰恰执行 (last-first) 次赋值动作（assignment）。

➢ 样例

copy 和 copy_backward 之间唯一的重要差别在于赋值动作的顺序。如果 input range 与 output range 重叠，那么赋值动作的顺序就成为一个要点了。下例之中，我们将数组的某部分复制到另一部分，而这两部分区间重叠。

```
int main()
{
  int A[15] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
  copy_backward(A, A + 10, A + 15);

  copy(A, A + 15, ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出 "1 2 3 4 5 1 2 3 4 5 6 7 8 9 10"。
}
```

12.2 互换元素（Swapping Elements）

12.2.1 swap

```
template <class Assignable>
void swap(Assignable& a, Assignable& b)
```

swap 是少数作用于个别 objects（藉由引数传入）而非 range 之上的算法。它可将 a 与 b 值互换，将 a 的内容赋值给 b，将 b 的内容赋值给 a。swap 是其他很多算法所使用的基本函数。

欲完全一般化地实现 swap，唯一的做法就是搭配一个临时变量。此法会调用 copy constructor 一次以及 assignment 操作符两次，并预期需要「三次 assignment 动作」的执行时间。然而在很多情况下，我们可以撰写出更具效率的特化 swap 版本。举个例子，假设要交换两个具有 N 个元素的 vector<double>。完全一般化的版本需要三个 vector 赋值动作，一共是 3N 次 double 赋值动作。我们可以为 vector 写个特化版本，使得它只需交换一些指针即可。

这很重要，因为 swap 是其他许多 STL 算法的基础动作，而 container 内含 container（例如 vector<string>）的情况十分常见。STL 对于所有的 container class，都提供特化版本的 swap。因此，使用者自订型别也应该提供 swap 的特化版本。

➢ 何处定义

HP STL 将 swap 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- Assignable 为 Assignable 之 model。

➢ 先决条件

无

➢ 复杂度

Amortized constant time [27]。

➢ 样例

```
int main()
{
  int x = 1;
  int y = 2;
  assert(x == 1 && y == 2);
  swap(x, y);
  assert(x == 2 && y == 1);
}
```

12.2.2 iter_swap

```
template <class ForwardIterator1, class ForwardIterator2>
inline void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

如果 a 和 b 是 iterators，那么 iter_swap(a, b) 等价于 swap(*a, *b)。

很少有正当理由使用 iter_swap。你应该改用 swap。iter_swap 算法如今看起来几乎已完全退化。原始的 HP STL [SL95] 之中之所以会涵盖它，单纯只是为了技术因素（更明确地说是为了支持带有「非标准之 reference type」的 iterator），而它之所以还保留于 C++ standard 中，目的只是为了向下兼容。

➢ 何处定义

HP STL 将 iter_swap 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

---

[27] 以 swap 交换两个 T objects，所需时间显然是取决于型别 T。这里所谓 constant time 并不代表 8-bit char 和 128-bit complex<double> 的 swap 时间一样。

➢ 型别条件

- ForwardIterator1 和 ForwardIterator2 都是 Forward Iterator 的 model。
- ForwardIterator1 和 ForwardIterator2 都是可变的（mutable）。
- ForwardIterator1 和 ForwardIterator2 都具有相同的 value type。

➢ 先决条件

- ForwardIterator1 和 ForwardIterator2 都是可提领的（dereferenceable）。

➢ 复杂度

Amortized constant time。

➢ 样例

```
int main()
{
  int x = 1;
  int y = 2;
  assert(x == 1 && y == 2);
  iter_swap(&x, &y);
  assert(x == 2 && y == 1);
}
```

12.2.3 swap_ranges

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

算法 swap_ranges 可将大小相同的两个 ranges 的内容互换。它会将 [first1,last1) 中的值与 [first2,first2+(last1-first1))中相对的元素互换。也就是说对于 0 <= n < (last1-first1) 区间内的每个整数 n，它会将 *(first1+n)与 *(first2+n)互换。返回值是 first2+(last1-first1)。

➢ 何处定义

HP STL 将 swap_ranges 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator1 和 ForwardIterator2 都是 Forward Iterator 的 model。
- ForwardIterator1 和 ForwardIterator2 的 value type 可互相转换。

➢ 先决条件

- [first1,last1) 为有效的 range。
- [first2,first2+(last1-first1)) 为有效的 range。
- [first1,last1) 与 [first2,first2+(last1-first1)) 不相重叠。

➢ 复杂度

线性。恰恰执行 (last1-first1) 次互换。

➢ 样例

```
int main()
{
  vector<int> V1;
  V1.push_back(1);
  V1.push_back(2);

  vector<int> V2;
  V2.push_back(3);
  V2.push_back(4);

  assert(V1[0] == 1 && V1[1] == 2 && V2[0] == 3 && V2[1] == 4);
  swap_ranges(V1.begin(), V1.end(), V2.begin());
  assert(V1[0] == 3 && V1[1] == 4 && V2[0] == 1 && V2[1] == 2);
}
```

12.3 transform

```
(1) template <class InputIterator, class OutputIterator,
              class UnaryFunction>
    OutputIterator transform(InputIterator first, InputIterator last,
                             OutputIterator result, UnaryFunction op);

(2) template <class InputIterator1, class InputIterator2,
              class OutputIterator,
              class BinaryFunction>
    OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, OutputIterator result,
                             BinaryFunction binary_op);
```

算法 transform 非常类似于 for_each（p.218），可在搭配一个 function object 后对 range 内的 objects 执行某种运算。两者的差别在于 for_each 不在意 function object 的返回值，但 tranform 会

将该返回值复制到另一个 range 中[28]。这么做的理由正如 transform 的名字所暗示：它在 range 内的各元素身上执行某种运算，以便将某个 range「转换」成另一个 range。

transform 有两个版本。一个版本将 Unary Function 套用于单一 input range 身上，另一版本则是将 Binary Function 套用于两个大小相等的 input ranges 身上。

版本一针对 input range 中的每个 iterator i，执行 op(*i)，并将运算结果赋值给 *o，其中 o 表示 i 所对应的 output iterator。也就是说对于 0<=n<last-first 中的每一个 n，此版本执行 *(result+n) = op(*(first+n))。返回值是 result+(last-first)。

版本二以 Binary Function 取代 Unary Function。对于 [first1,last1) 中的每一个 iterator i1，执行 op(*i1, *i2)，并将运算结果赋值给 *o，其中 i2 是第二个 range 中对应的 iterator，o 则是对应的 output iterator。也就是说，对于 0<=n<last1-first1 中的每一个 n，此版本执行 *(result+n) = op(*(first1+n), *(first2+n))。返回值是 result+(last1-first1)。

注意，transform 可以被用来「就地（in place）」更改序列。是的，它允许 first 和 result 是同一个 iterator。但是尽管 input range 和 output range 可视为同一（identical），其他方面它们却不可重叠（overlap），因为除了 first 之外，Output Iterator result 不可与 [first,last) 中的任何一个 Input Iterator 相同。

➢ 何处定义

HP STL 将 transform 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- UnaryFunction 为 Unary Function 之 model。
- InputIterator 之 value type 可转换为 UnaryFunction 之引数型别。
- UnaryFunction result type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

第二个版本：

- InputIterator1 和 InputIterator2 皆为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- BinaryFunction 为 Binary Function 之 model。

---

[28] 如果你是 lisp 程序员，你可能会发现，for_each 与 transform 之间的关系就好像 mapc 与 mapcar 之间的关系。

- InputIterator1 和 InputIterator2 之 value type 各自可转换为 BinaryFunction 之第一与第二引数型别。
- BinaryFunction result type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

第一个版本：

- [first,last) 为有效的 range。
- result 不是 [first+1,last) 中的 iterator。
- 有足够的空间容纳所有被复制的元素。更正式地说：[result,result+(last-first)) 是个有效的 range。

第二个版本：

- [first1,last1) 为有效的 range。
- [first2,first2+(last1-first1)) 为有效的 range。
- result 不是 [first1+1,last1) 或 [first2+1,first2+(last1-first1)) 内的 iterator。
- 有足够的空间容纳所有被复制的元素。更正式地说：[result,result+(last1-first1)) 是有效的 range。

➢ 复杂度

线性。版本一恰恰套用 last-first 次运算行为，版本二恰恰套用 last1-first1 次运算行为。

➢ 样例

transform 的第一个版本是 copy（p.233）的一般化版本。它从一个 range 取值出来，执行某种运算后写入另一个 range。如果这「某种运算」是个 identity function，transform 的作为将与 copy 相同。

```
int main()
{
  const int N = 7;
  double A[N] = {4, 5, 6, 7, 1, 2, 3};
  list<double> L(N);

  transform(A, A + N, L.begin(), identity<double>());
  copy(L.begin(), L.end(), ostream_iterator<double>(cout, "\n"));
}
```

一个比较传统的例子是将 range 中的每个元素以其负值取代之。

```
int main()
{
  const int N = 10;
  double A[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  transform(A, A + N, A, negate<double>());
  copy(A, A + N, ostream_iterator<double>(cout, "\n"));
}
```

transform 的第二个版本作用于两个不同的 input ranges 身上。既然如此，我们可利用它来计算一个 vector 与一个数组的总和。Output range 不一定得是 list 或 vector，可以是任何种类的 Output Iterator，甚至包括根本不与任何 container 相关联的 Output Iterator。

```
int main()
{
  const int N = 10;

  vector<int> V(N);
  fill(V.begin(), V.end(), 75);

  int A[N] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  transform(V.begin(), V.end(), &A[0],
            ostream_iterator<int>(cout, "\n"),
            plus<int>());
}
```

12.4 替换元素（Replacing Elements）

12.4.1 replace

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

算法 replace 会审阅 range 中的每个元素，凡 old_value 出现之处，便改为 new_value。与 old_value 不相等的元素不会受到影响。

更正式地说，针对 [first,last) 中的每个 iterator i，replace 会将*i 与 old_value 做比较，如果 *i == old_value，便执行 *i = new_value。

和 find（p.199）一样，replace 有个明显的一般化版本。此版本不再测试元素是否等于 old_equal，而是让使用者控制测试行为：由他们提供更一般化的 Predicate。由于技术因素，这个一般化版本不能够也命名为 replace，因为两个版本具有相同的引数个数，而且两者都是 template functions。这个一般化版本必须有不同的名字：replace_if。

➢ 何处定义

HP STL 将 replace 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- T 可转换为 ForwardIterator 之 value type。
- T 为 Assignable。
- T 是 Equality Comparable 的 model，并可与 ForwardIterator value type 的 object 做相等测试。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰做 last-first 次比较动作，最多做 last-first 次赋值动作。

➢ 样例

以下将 apples 替换为 oranges。

```
int main()
{
  vector<string> fruits;
  fruits.push_back("cherry");
  fruits.push_back("apple");
  fruits.push_back("peach");
  fruits.push_back("plum");
  fruits.push_back("pear");

  replace(fruits.begin(), fruits.end(),
          string("apple"), string("orange"));
  copy(fruits.begin(), fruits.end(),
       ostream_iterator<string>(cout, "\n"));
}
```

12.4.2 replace_if

```
template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred,
                const T& new_value);
```

算法 replace_if 是 replace 的一般化版本。它会审阅 range 中的每个元素，将每一个「致令 Predicate pred 返回 true」的元素替换为 new_value。「致令 pred 返回 false」的元素则不受影响。

更正式地说，对于 [first,last) 内的每一个 iterator i，replace_if 观察 pred(*i) 的结果。如果 pred(*i) 为 true，replace_if 便执行 *i = new_value。

很多 STL 算法都具有两种版本，版本一执行缺省行为，版本二执行使用者指定之操作行为。通常这样的两个版本具有相同名称。算法 replace 和 replace_if 名称不同，单纯只是技术上的因素：两者都是 template functions，而且具有相同的引数个数，如果两个版本同名，会有歧义（模棱两可，ambiguities）之虞。

➢ 何处定义

HP STL 将 replace_if 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- Predicate 为 Predicate 之 model。
- ForwardIterator 之 value type 可转换为 Predicate 之引数型别。
- T 可转换为 ForwardIterator 之 value type。
- T 为 Assignable。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰套用 pred 共 last-first 次，最多做 last-first 次赋值动作。

➢ 样例

replace_if 的一个常见用途是确保 range 中的所有元素都遵循某种限制。未能遵循的元素则以能够遵循的值取代之。下面这个例子，所有负值都以 0 取代。

```
int main()
{
  vector<double> V;
  V.push_back(1);
```

```
  V.push_back(-3);
  V.push_back(2);
  V.push_back(-1);

  replace_if(V.begin(), V.end(),
             bind2nd(less<double>(), 0),
             0.);
  assert(V[1] == 0 && V[3] == 0);
}
```

以下例子将长度超过 6 的所有字符串以「溢位指示」字符串 "******" 取代之。

```
struct string_length_exceeds
{
  string_length_exceeds(int n) : limit(n) {}
  bool operator()(const string &s) const { return s.size() > limit; }

  int limit;
};

int main()
{
  string A[7] = {"oxygen", "carbon", "nitrogen", "iron",
                 "sodiym", "hydrogen", "silicon"};

  replace_if(A, A + 7,
             string_length_exceeds(6),
             "******");
  copy(A, A + 7, ostream_iterator<string>(cout, "\n"));
}
```

12.4.3 replace_copy

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                            OutputIterator result, const T& old_value,
                            const T& new_value);
```

算法 replace_copy 可将 [first,last) 的元素复制到 [result,result+(last-first)) 中，复制过程中以 new_value 取代 old_value。[first,last) 不会被更动。

更正式地说，对于 0 <= n < last-first 的每个整数 n，如果 *(first+n) == old_value，就执行 *(result+n) = new_value，反之则执行 *(result+n) = *(first+n)。

就像所有以 *_copy 命名的算法一样，replace_copy 的行为就像是「copy 之后，再 replace」。差别之一在于 replace_copy 更有效率，差别之二是 replace_copy 可与 Output Iterators 之 output range 搭配使用。

Generic Programming and the STL

➢ 何处定义

HP STL 将 `replace_copy` 声明于头文件 `<algo.h>`。C++ standard 将它声明于头文件 `<algorithm>`。

➢ 型别条件

- `InputIterator` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `T` 为 Equality Comparable 之 model，且可与 `InputIterator` value type 做相等比较。
- `T` 为 Assignable。
- `T` 可转换为 `OutputIterator` value types 所形成之集合内的某个型别。

➢ 先决条件

- `[first,last)` 为有效的 range。
- output range 有足够的空间容纳被复制的值。也就是说，`[result,result+(last-first))` 是个有效的 range。
- `result` 不是 `[first,last)` 中的某个 iterator。

➢ 复杂度

线性。恰恰进行 `last-first` 次相等比较，恰恰进行 `last-first` 次赋值动作。

➢ 样例

以 99 取代所有的 1。

```
int main()
{
  vector<int> V1;
  V1.push_back(1);
  V1.push_back(2);
  V1.push_back(3);
  V1.push_back(1);

  vector<int> V2(V1.size());
  replace_copy(V1.begin(), V1.end(), V2.begin(), 1, 99);
  assert(V2[1] == V1[1] && V2[2] == V1[2]);
  assert(V2[0] == 99 && V2[3] == 99);
}
```

12.4.4 replace_copy_if

```
template <class InputIterator, class OutputIterator,
          class Predicate, class T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value);
```

算法 replace_copy_if 是 replace_if 的复制式变形，就像算法 replace_copy 是 replace 的复制式变形一般。换句话说，它和 replace_if 很像，唯一不同是它作用于 input range 和 output range，而非以「就地（in place）」方式作用于单一 range。

和其他复制式算法一样，replace_copy_if 的行为类似「copy 之后，再 replace_if」。它可以从 [first,last) 中将元素复制到 [result,result+(last-first)) 中，并将任何「致令 pred 返回 true」的元素取代为 new_value。

更明确地地说，对于 0 <= n < last-first 的每一个整数 n，如果 pred(*(first+n))为 true，就执行 *(result+n) = new_value，反之就执行 *(result+n) = *(first+n)。

➢ 何处定义

HP STL 将 replace_copy_if 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- Predicate 为 Predicate 之 model。
- InputIterator 之 value type 可转换为 Predicate 之引数型别。
- T 为 Assignable。
- T 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

- [first,last) 为有效的 range。
- output range 有足够空间容纳被复制的值。也就是说 [result,result+(last-first)) 是有效的 range。
- result 不是 [first,last) 内的一个 iterator。

➢ 复杂度

线性。恰恰套用 last-first 次的 pred，最多进行 last-first 次赋值动作。

➢ 样例

以下将 vector 内的元素复制到标准输出设备，并将所有负值取代为 0。

```
int main()
{
  vector<double> V;
  V.push_back(1);
  V.push_back(-1);
  V.push_back(-5);
  V.push_back(2);

  replace_copy_if(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"),
                  bind2nd(less<int>(), 0), 0);
}
```

12.5 充填整个区间（Filling Ranges）

12.5.1 fill

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

算法 fill 将某值赋值给 [first,last) 中的每个元素。换句话说对于 [first,last) 中的每个 iterator i，执行 *i = value。

➢ 何处定义

HP STL 将 fill 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model[29]。
- ForwardIterator 是可变的（mutable）。
- T 为 Assignable。
- T 可转换为 ForwardIterator 之 value type。

➢ 先决条件

- [first,last) 为有效的 range。

---

[29] 这个引数必须为 mutable Forward Iterator 而非 Output Iterator，因为 fill 会用到 [first,last)。由于两个 Output Iterators 之间无法做相等测试，所以无法以 Output Iterators 表现一个 range。相对之下，fill_n 就允许使用 Output Iterator。

➢ 复杂度

线性。恰恰进行 last-first 次赋值动作。

➢ 样例

将 137 赋值给 vector 内的每个元素。

```
int main()
{
  vector<double> V(4);
  fill(V.begin(), V.end(), 137);
  assert(V[0] == 137 && V[1] == 137 && V[2] == 137 && V[3] == 137);
}
```

12.5.2 fill_n

```
template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

算法 fill_n 将数值 value 赋值给 [first,first+n) 内的每个元素。亦即对于 [first,first+n) 中的每个 iterator i，执行 *i = value。返回值为 first+n。

➢ 何处定义

HP STL 将 fill_n 声明于头文件 <algo.h>，定义于 <algobase.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- OutputIterator 为 Output Iterator 之 model。
- Size 为整数（integral）型别（可能有正负号，也可能无正负号）。
- T 为 Assignable。
- T 可转换为 OutputIterator 之 value type。

➢ 先决条件

- n >= 0。
- 有足够空间容纳 n 个值。换句话说 [first,first+n) 为有效 range。

➢ 复杂度

线性。恰恰赋值 n 次。

➢ 样例

制作三个 137 副本，添加于 vector 尾端。

```
int main()
{
  vector<double> V(2, 128.);
  fill_n(back_inserter(V), 3, 137.);
  assert(V.size() == 5 &&
         V[2] == 137 && V[3] == 137 && V[4] == 137);
}
```

12.5.3 generate

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
              Generator gen);
```

算法 generate 会调用 gen(一个不需任何引数的 function object)，并将其结果赋值给 [first,last) 中的每个元素。亦即针对 [first,last) 中的每个 iterator i ，执行 *i = gen()。

针对 [first,last) 内的每一个 iterators，gen 都会被调用一次，而非只在循环外调用一次。这一点很重要，因为 Generator 不一定会在每次被调用时都返回相同结果。因此 generate 允许从文件读入，取局部状态的值并更改……

如果你只想调用 function object 一次，然后将其结果赋值给 range 内的每个元素，可以使用 fill。

➢ 何处定义

HP STL 将 generate 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model[30]。
- ForwardIterator 是可变的（mutable）。
- Generator 为 Genreator 之 model。
- Generator 之 result type 可转换为 ForwardIterator 之 value type。

➢ 先决条件

- [first,last) 为有效的 range。

---

[30] 引数必须是一个 mutable Forward Iterator 而非一个 Output Iterator，因为 generate 会产生 iterators range [first,last)。由于两个 Output Iterators 无法做相等测试，所以无法以 Output Iterators 表现一个 range。相对之下，generate_n 允许使用 Output Iterator。

➢ 复杂度

线性。恰恰调用 gen 共 (last-first) 次。

➢ 样例

利用 standard C library rand 函数，以乱数填满 vector。

```
int main()
{
  vector<int> V(100);
  generate(V.begin(), V.end(), rand);
}
```

12.5.4 generate_n

```
template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, const Generator gen);
```

算法 generate_n 会将 gen(一个不需引数的 function object)所得结果赋值给 [first,first+n) 中的每个元素。也就是说针对 [first,first+n) 内的每个 iterator i，执行 *i = gen()。

返回值为 first+n。

Function object gen 会被调用 n 次，针对 [first,first+n) 的每个 iterator 各调用一次，而非只在循环外调用一次。这一点很重要，因为 Generator 每次被调用的结果不尽相同。所以 generate_n 允许从文件读入、取局部状态的值并更改……

➢ 何处定义

HP STL 将 generate_n 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- OutputIterator 为 Output Iterator 之 model。
- Size 为整数（integral）型别（可能有正负号，也可能无正负号）。
- Generator 为 Generator 之 model。
- Generator result type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

- n >= 0。
- 有足够空间容纳 n 个值。换句话说 [first,first+n) 为有效 range。

➢ 复杂度

线性。恰恰赋值 n 次。

➢ 样例

利用 C 标准函数库的 rand 函数，打印 100 个乱数。

```
int main()
{
  generate_n(ostream_iterator<int>(cout, "\n"), 100, rand);
}
```

12.6 移除元素（Removing Elements）

12.6.1 remove

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

算法 remove 会将数值 value 从 [first,last) 中移除。

「移除」的意义有点微妙。其关键重点在于：本章之中任何一个「会改变操作目标物内容」的算法（mutating algorithm），包括现在所谈的 remove，都不可能摧毁任何 iterator 或是改变 first 与 last 之间的距离（见 p.233）。例如，假设 A 是个 int[10] 数组，如果你写 remove(A, A+10, 3)，实际上 remove 不会改变 A 的元素个数，因为 C 语言的数组不能重定大小。表达式 A+10 永远会指向 A 之后第 10 个元素的位置。

由于本章的各个算法并非作用于 range 身上，而是作用于 iterator 身上，所以「移除动作（removal）」实际上并不会改变 container 的大小：它们除了对 iterator 所指元素赋值之外，就无法做什么了。

本书之中，移除动作的实际意思是：remove 会返回一个 iterator new_last，使得 [first,new_last) 内的元素都不等于 value。也就是说，value 在 [first,new_last) 区间内被移除出去了。我们说 remove 动作是稳定的（*stable*），意思是区间左端的元素的相对位置不变。

[new_last,last) 中的 iterators 仍然可提领，但其指向之值未在规范之列（unspecified）— 那是我们不感兴趣的元素，可忽略之。

如果你从一个 Sequence 中移除元素，你可以大大方方地将 new_last 之后的元素抹除掉（erase）。真正能够从 Sequence S 中将元素拿掉（真正抹除它们并改变 Sequence 大小）的一个做法是：

```
S.erase(remove(S.begin(), S.end(), x), S.end());
```

➢ 何处定义

HP STL 将 remove 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- T 为 Equality Comparable。
- 型别为 T 之值可以与 ForwardIterator value type 之值做相等比较。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰执行 last-first 次相等比较（equality comparison）动作。

➢ 样例

将 vector 内其值为 1 的元素移除。注意，调用 remove 之后，vector 的大小仍然不变。在此程序中，紧接着 remove 之后第二次打印 vector，仍有九个元素。不过 [V.begin(),new_end) 内绝不会再有等于 1 的元素。

vector 的大小不会改变，直到明白调用 member function erase()。

```
template <class Container>
void print_container(const Container& C)
{
  cout << C.size() << " elements: ";
  copy(C.begin(), C.end(),
       ostream_iterator<typename Container::value_type>(cout, " "));
  cout << endl;
}

int main()
{
  const int A[9] = {3, 1, 4, 1, 5, 9, 2, 6, 5};

  vector<int> V(A, A + 9);
  print_container(V);

  vector<int>::iterator new_end = remove(V.begin(), V.end(), 1);
  print_container(V);

  V.erase(new_end, V.end());
```

```
    print_container(V);
  }
```

12.6.2 remove_if

```
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

算法 remove_if 会移除每一个「致令 Predicate pred 返回 true」的元素。remove_if之于 remove，就好比 replace_if 之于 replace 一样。remove 和 remove_if 之所以名称不同，纯粹是技术上的因素。因为两者都是 template functions，而且具有相同的引数个数，如果两者同名，会有歧义（模棱两可，ambiguious）之虞。

就像 remove 一样，「移除」的意义有点微妙。实际上本算法无法改变 [first,last) 内的元素个数（见 p.253 的讨论）。

和 remove 一样，remove_if 也会返回一个 iterator new_last，使 [first,new_last) 之内不含「致令 pred 为 true」的元素。也就是说凡是满足 pred 的元素，都会被移除于 [first,new_last) 之外。我们说 remove_if 动作是稳定的（*stable*），意思是[first,new_last) 中各元素的相对位置和 [first,last) 中各元素的相对位置相较之下不曾改变。

[new_last,last) 内的所有 iterators 仍然可提领，但其指向之值未在规范之列（unspecified）。

➢ 何处定义

HP STL 将 remove_if 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- Predicate 为 Predicate 之 model。
- ForwardIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰套用 (last-first) 次 pred 动作。

➢ 样例

以下移除 vector 内的所有偶数。和 remove（p.253）一样，remove_if 并不会改变 vector 的元素个数。在调用 vector member function erase()之前，vector 的大小不变。

```
int main()
{
  vector<int> V;
  V.push_back(1);
  V.push_back(4);
  V.push_back(2);
  V.push_back(8);
  V.push_back(5);
  V.push_back(7);

  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
     // 输出 "1 4 2 8 5 7"

  vector<int>::iterator new_end =
        remove_if(V.begin(), V.end(),
                  compose1(bind2nd(equal_to<int>(), 0),
                           bind2nd(modulus<int>(), 2)));
  V.erase(new_end, V.end());

  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
     // 输出 "1 5 7"
}
```

12.6.3 remove_copy

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

算法 remove_copy 可从 [first,last) 中将元素复制至「以 result 开头的 range」身上，但不会复制与 value 相等的元素。返回值是「用来安放执行结果」之 range 的尾端。如果 [first,last) 内有 n 个元素不等于 value，则返回值为 result+n。

我们说该操作行为是稳定的（stable），意思是被复制元素的相对位置，与原先在 [first,last) 时并没有改变。

➢ 何处定义

HP STL 将 remove_copy 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- `InputIterator` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `InputIterator` value type 可转换为 `OutputIterator` value types 所形成的集合内的某个型别。
- `T` 为 Equality Comparable。
- `T` object 可与 `InputIterator` value type 的 object 做相等测试。

➢ 先决条件

- `[first,last)` 为有效的 range。
- Output range 有足够空间容纳被复制的值。如果 `[first,last)` 中有 `n` 个元素不等于 `value`，则 `[result,result+n)` 是有效的 range。
- `result` 并非 `[first,last)` 内的某个 iterator。

➢ 复杂度

线性。恰恰执行 `last-first` 次相等比较（equality comparison）动作，最多执行 `last-first` 次赋值动作。

➢ 样例

将 `vector` 内的所有非空字符串打印出来。

```
int main()
{
  vector<string> V;

  V.push_back("one");
  V.push_back("");
  V.push_back("four");
  V.push_back("");
  V.push_back("");
  V.push_back("ten");

  remove_copy(V.begin(), V.end(),
              ostream_iterator<string>(cout, "\n"),
              string(""));
}
```

输出为

```
one
four
ten
```

12.6.4 remove_copy_if

```
template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator
remove_copy_if(InputIterator first, InputIterator last,
               OutputIterator result, Predicate pred);
```

算法 remove_copy_if 可将 [first,last) 内的元素复制到以 result 开头的 range 内，但不复制「令 pred 为 true」的元素。返回值是「用来安放执行结果」之 range 的尾端。如果 [first,last) 中有 n 个元素造成 pred 不为 true，则返回值为 result+n。

我们说该操作行为是稳定的（stable），意思是被复制元素的相对位置与原先在 [first,last) 时并没有改变。

➢ 何处定义

HP STL 将 remove_copy_if 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别
- Predicate 为 Predicate 之 model。
- InputIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。
- Output range 有足够空间容纳被复制的值。也就是说如果 [first,last) 有 n 个元素不满足 pred，则 output range [result,result+n) 是有效的 range。
- result 并非 [first,last) 内的某个 iterator。

➢ 复杂度

线性。恰恰套用 last-first 次 pred 动作，最多进行 last-first 次赋值动作。

➢ 样例

将 vector 内的非负元素填到另一个 vector 中。

```
int main()
{
  vector<int> V1;
  V1.push_back(-2);
  V1.push_back(0);
  V1.push_back(-1);
  V1.push_back(0);
  V1.push_back(1);
  V1.push_back(2);

  vector<int> V2;
  remove_copy_if(V1.begin(), V1.end(), back_inserter(V2),
                 bind2nd(less<int>(), 0));
}
```

很多人建议 STL 应该包含算法 copy_if，作为 remove_copy_if 的相反动作。remove_copy_if 用来复制不满足某个条件的所有元素，copy_if 则用来复制满足某个条件的所有元素。

根据 remove_copy_if 实现 copy_if 是很简单的事。

```
template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                       OutputIterator result, Predicate pred)
{
  return remove_copy_if(first, last, result, not1(pred));
}
```

12.6.5 unique

```
(1) template <class ForwardIterator>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last);

(2) template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                           BinaryPredicate binary_pred);
```

算法 unique 能够移除重复的元素。每当在 [first,last) 内遇到相连重复元素群出现，unique 便会移除该群中第一个以外的所有元素。

注意，unique 只移除相邻的重复元素。如果你想要移除所有重复元素，你必须确定所有重复元素都相邻。基于这个理由，unique 和 sort（p.292）搭配特别管用。

正如本章涵盖的所有算法一样，unique 实际上并不会改变 [first,last) 的元素个数。请参考 p.253 的讨论。

unique 会返回一个 iterator new_last，使得 [first,new_last) 内的元素都不含相邻重复元素。我们说 unique 程序是稳定的（stable），意思是未被移除的元素的相对顺序并不改变。

[new_last,last) 内的 iterators 仍然可提领，但其指向之值未在规范之列（unspecified）。

unique 有两个不同版本，因为所谓「相邻元素是否重复」，有两种定义。第一版本使用简单的相等测试：针对 [f,l) 之中除第一个外的每个 iterator i，如果 *i == *(i-1) 就表示该 range 内有元素重复。第二版本使用一个 Binary Predicate binary_pred 作为测试准则：针对 [f,l) 之中除第一个外的每个 iterator i，如果 binary_pred(*i, *(i-1)) 为 true，就表示该 range 内有元素重复。

➢ 何处定义

HP STL 将 unique 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- ForwardIterator 之 value type 为 Equality Comparable。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。
- BinaryPredicate 为 Binary Predicate 之 model[31]。
- ForwardIterator value type 可转换为 BinaryPredicate 之第一与第二引数型别。

➢ 先决条件

[first,last) 为有效的 range。

➢ 复杂度

线性。对于非空的 range，恰恰套用 operator==（对版本一而言）或 binary_pred（对版本二而言）共 (last-first)-1 次。对于空的 range，则不套用任何 operator== 或 binary_pred。

---

[31] BinaryPredicate 并不一定得为等价关系（equivalence relation），但你如果以 unique 搭配不具等价关系的 Binary Predicate，应该小心，因为你很容易获得意外结果。

➢ 样例

将以下相邻而重复的 ints 移除。请注意，和 remove（p.253）一样，vector 的大小不变，除非明白调用其 member function erase()。

```
int main() {
  vector<int> V;
  V.push_back(1);
  V.push_back(3);
  V.push_back(3);
  V.push_back(3);
  V.push_back(2);
  V.push_back(2);
  V.push_back(1);

  vector<int>::iterator new_end = unique(V.begin(), V.end());
  V.erase(new_end, V.end());
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出 "1 3 2 1"
}
```

严格地说，unique 的第一版本有点赘余：如果以 class equal_to 作为第二版本的 Binary Predicate 引数，结果便完全相同。以下我使用 unique 的第二版本完成同一个例子：

```
int main() {
  int A[8] = {7, 7, 1, 4, 6, 6, 6, 3};

  int *new_end = unique(A, A + 8, equal_to<int>());
  copy(A, new_end, ostream_iterator<int>(cout, "\n"));
}
```

第二版本的更大用途是从 vector 中移除重复字符并忽略大小写。首先将 vector 排序，然后从相邻群中移除重复字符。

```
inline bool eq_nocase(char c1, char c2) {
  return tolower(c1) == tolower(c2);
}

inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  const char init[] = "The Standard Template Library";
  vector<char> V(init, init + sizeof(init));
  sort(V.begin(), V.end(), lt_nocase);
  copy(V.begin(), V.end(), ostream_iterator<char>(cout));
```

```
    cout << endl;
    vector<char>::iterator new_end = unique(V.begin(), V.end(), eq_nocase);
    copy(V.begin(), new_end, ostream_iterator<char>(cout));
    cout << endl;
    // 输出为:
    //   aaaabddeeehiLlmnprrrStTtTy
    // abdehiLmnprSty
}
```

12.6.6 unique_copy

```
(1) template <class InputIterator, class OutputIterator>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                               OutputIterator result);

(2) template <class InputIterator, class OutputIterator,
              class BinaryPredicate>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                               OutputIterator result,
                               BinaryPredicate binary_pred);
```

算法 unique_copy 可从 [first,last) 中将元素复制到以 result 开头的 range 身上；如果面对相邻重复元素群，只会复制其中第一个元素。返回值是「用来安放被复制元素」的 range 尾端[32]。

就像其他名为 *_copy 的算法一样，unique_copy 乃是 unique 的一个复制式版本，相当于「copy 完后再做 unique」。和 unique 一样，它只会移除相邻的重复元素，不会移除被其他元素隔开的重复元素。

unique_copy 有两个不同版本，因为所谓「相邻元素是否重复」有两种定义。第一版本使用简单的相等测试：针对 [f,l) 之中除第一个之外的每个 iterator i，如果 *i == *(i-1)，就表示 range 内有元素重复。第二版本使用 Binary Predicate binary_pred 做测试：针对 [f,l) 之中除第一个外的每个 iterator i，如果 binary_pred(*i, *(i-1)) 为 true，就表示该 range 内有元素重复。

➢ 何处定义

HP STL 将 unique_copy 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

---

[32] 其行为类似 UNIX 过滤器（filter）uniq。

➢ 型别条件

第一个版本：

- `InputIterator` 为 Input Iterator 之 model。
- `InputIterator` 之 value type 为 Equality Comparable。
- `OutputIterator` 为 Output Iterator 之 model。
- `InputIterator` value type 可转换为 `OutputIterator` value types 所形成之集合内的某个型别。

第二个版本：

- `InputIterator` 为 Input Iterator 之 model。
- `BianryPredicate` 为 Bianry Predicate 之 model[33]。
- `InputIterator` 之 value type 可转换为 `BinaryPredicate` 之第一与第二引数型别。
- `OutputIterator` 为 Output Iterator 之 model。
- `InputIterator` value type 可转换为 `OutputIterator` value types 所形成之集合内的某个型别。

➢ 先决条件

`[first,last)` 为有效的 range。

有足够空间容纳所有被复制的元素。更正式地说，如果在相邻重复元素被移除之后 `[first,last)` 内还有 `n` 个元素，则 `[result,result+n)` 必须为一个有效的 range。

➢ 复杂度

线性。恰恰套用 `operator==`（对版本一而言）或 `binary_pred`（对版本二而言）`last-first` 次，最多进行 `last-first` 次赋值动作。

➢ 样例

以下打印数组中的所有元素，如果有相邻重复元素，只打印第一个。输出内容仍有重复元素：2 和 8，它们都会出现两次，但输出内容不会有相邻的重复元素 — 这就是 `unique` 和 `unique_copy` 的保证。

```
int main() {
  const int A[] = {2, 7, 7, 7, 1, 1, 8, 8, 8, 2, 8, 8};
  unique_copy(A, A + sizeof(A) / sizeof(int),
              ostream_iterator<int>(cout, " "));
```

---

[33] `BinaryPredicate` 并不一定得为等价关系（equivalence relation），但你如果以 `unique` 搭配不具等价关系的 Binary Predicate，应该小心，因为你很容易获得意外的结果。

```
  cout << endl;
    // 输出 "2 7 1 8 2 8"
}
```

将某数组的内容填入 vector，但区间 0~9 的数只取一个，区间 10~19 的数亦只取一个，依此类推。

```
struct eq_div {
  eq_div(int divisor) : n(divisor) {}
  bool operator()(int x, int y) const { return x / n == y / n; }

  int n;
};

int main() {
  int A[] = {1, 5, 8, 15, 23, 27, 41, 42, 43, 44, 67, 83, 89};
  const int N = sizeof(A) / sizeof(int);

  vector<int> V;
  unique_copy(A, A + N, back_inserter(V), eq_div(10));

  copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
  // 输出 1 15 23 41 67 83
}
```

12.7 排列算法（Permuting Algorithms）

所谓 range 元素的排列（permutation），就是将 range 中的元素重定顺序。它不会删除或新增任何元素，只是将元素值以不同的顺序放入 input range 内。作用于 range 身上的许多重要算法都属于这一类，例如 sorting（排序）就是其一。sorting 极为重要，第 13 章将专门讨论排序算法。

12.7.1 reverse

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

算法 reverse 会将 range 的内容「就地（in place）」反转。也就是说对于 0<=n<=(last-first)/2 区间内的每个 n，它会将 *(first+n) 与 *(last-(n+1)) 互换。

注意，如同本章涵盖的所有算法一样，reverse 所反转的是 range 内的 iterators 所指元素，而非 iterators 自身。

➢ 何处定义

HP STL 将 reverse 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- `BidirectionalIterator` 为 Bidirectional Iterator 之 model。
- `BidirectionalIterator` 是可变的（mutable）。

➢ 先决条件

- `[first,last)` 为有效的 range。

➢ 复杂度

线性。恰恰调用 `swap`（p.237）共 `(last-first)/2` 次。

➢ 样例

```
int main() {
  vector<int> V;
  V.push_back(0);
  V.push_back(1);
  V.push_back(2);
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
              // 输出 "0 1 2"
  reverse(V.begin(), V.end());
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
              // 输出 "2 1 0"
}
```

12.7.2 reverse_copy

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

算法 `reverse_copy` 可从 `[first,last)` 内将元素复制到`[result,result+(last-first))` 内，并使此复制品为原先次序的反转。也就是说对于 `0 <= n < (last-first)` 内的每个 `n`，`reverse_copy` 执行以下动作：`*(result+(last-first)-n-1) = *(first+n)`。

返回值为新产生之 range 的尾端：`result+(last-first)`。

如其名称所示，`reverse_copy` 是 `reverse` 的复制式版本。其效果相同于先 `copy` 再 `reverse`。

➢ 何处定义

HP STL 将 `reverse_copy` 声明于头文件 `<algo.h>`。C++ standard 将它声明于头文件 `<algorithm>`。

➢ **型别条件**

- `BidirectionalIterator` 为 Bidirectional Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `BidirectionalIterator` value type 可转换为 `OutputIterator` value types 所形成之集合内的某个型别。

➢ **先决条件**

- `[first,last)` 为有效的 range。
- 有足够空间容纳所有被复制的元素。更正式地说是 `[result,result+(last-first))` 为一个有效 range。
- `[first,last)` 和 `[result,result+(last-first))` 不重叠。

➢ **复杂度**

线性。恰恰执行 `last-first` 次赋值动作（assignments）。

➢ **样例**

```
int main() {
  vector<int> V;
  V.push_back(0);
  V.push_back(1);
  V.push_back(2);
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
              // 输出 "0 1 2"
  reverse_copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
              // 输出 "2 1 0"
}
```

12.7.3 rotate

```
template <class ForwardIterator>
inline void rotate(ForwardIterator first, ForwardIterator middle,
                   ForwardIterator last);
```

算法 `rotate` 将 range 内的元素加以旋转（rotate）：它会将 `middle` 处的元素移到 `first` 处、将 `middle+1` 处的元素移到 `first+1` 处。更正式地说，对于 `0 <= n < last-first` 区间内的每一个 `n`，它会将元素 `*(first+n)` 赋值给 `*(first+(n+(last-middle)) % (last-first))`。

这份定义令人怯步，你可能因此认为 `rotate` 很晦涩或很特别。事实上有更简单的方式可以了解 `rotate` 的行为：它会将 `[first,middle)` 与 `[middle,last)` 互换。

rotate 的用途之一是以 swap_ranges（p.239）所无法达到的方式来「交换（swap）」两个 ranges。你可以用 swap_ranges 来交换两个长度相同的 ranges，但你可以用 rotate 来交换两个长度不同的相邻 ranges。

➢ 何处定义

HP STL 将 rotate 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 是可变的（mutable）。

➢ 先决条件

- [first,middle) 和 [middle,last) 都是有效的 ranges。这意味 [first,last) 也是一个有效的 range。

➢ 复杂度

线性。rotate 最多调用 swap（p.237）共 last-first 次。赋值动作的确切次数取决于 rotate 的引数是个 Forward Iterator、Bidirectional Iterator 或 Random Access Iterator。这三种情况的复杂度都是线性的。

➢ 样例

本例之中我们旋转整数数组。首先将第一个元素移到数组尾端，也就是说将「只含第一个元素」的 range 和「包含其他元素」的 range 交换。接下来颠倒这样的行为，也就是将「只含最后一个元素」的 range 和「包含其他元素」的 range 交换。最后，将数组中的前三个元素移到尾端。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A) / sizeof(int);

  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
                          // 输出: 1 2 3 4 5 6 7
  rotate(A, A + 1, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
                          // 输出: 2 3 4 5 6 7 1
  rotate(A, A + N - 1, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
```

```
                                    // 输出: 1 2 3 4 5 6 7
    rotate(A, A + 3, A + N);
    copy(A, A + N, ostream_iterator<int>(cout, " "));
    cout << endl;
                                    // 输出: 4 5 6 7 1 2 3
}
```

12.7.4 rotate_copy

```
template <class ForwardIterator, class OutputIterator>
OutputIterator
rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);
```

算法 rotate_copy 是 rotate 的复制式版本。它像是先 copy 然后再 rotate。它并不会「就地（in place）」旋转其 input range，而是会在「以 iterator result 为开头」的 range 中产生一个旋转后的复制品。

就像 rotate 一样，你可以将此行为视为两个 ranges 的交换：从 input range [first,last] 中将元素复制到 output range [result,result+(last-first))，只不过 [first,middle) 和 [middle,last) 是颠倒的。也就是说，*middle 复制到 *result、*(middle+1) 复制到 *(result+1)，依此类推。更正式地说，对于 0 <= n < last - first 区间内的每个 n，rotate_copy 执行以下动作：*(result+(n+(last-middle)) & (last-first)) = *(first+n)。

返回值为 output range 的尾端：result+(last-first)。

➢ 何处定义

HP STL 将 rotate_copy 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- ForwardIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

- [first,middle)与[middle,last)为有效的 range。意味 [first,last) 也是个有效的 range。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+(last-first)) 必须是个有效的 range。

- [first,last) 和 [result,result+(last-first)) 不相重叠。

➢ 复杂度

线性。恰恰赋值 last-first 次。

➢ 样例

本例之中，数组 A 由一连串 1 接着一连串 2 构成。我们将这两部分以相反的次序复制到标准输出设备。

```
int main()
{
  int A[] = {1, 1, 1, 2, 2, 2, 2};
  const int N = sizeof(A) / sizeof(int);

  rotate_copy(A, A + 3, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
                                  // 输出: 2 2 2 2 1 1 1
}
```

12.7.5 next_permutation

```
(1) template <class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);

(2) template <class BidirectionalIterator, class StrictWeakOrdering>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last,
                          StrictWeakOrdering comp);
```

已知任意两个 ranges，其元素皆为 LessThan Comparable，那么我们总是能够利用 lexicographical_compare(p.225)，以字典排序方式来比较它们。更明确地说，我们可以对同一 range 的两个不同排列顺序，例如 (1,2,3) 和 (1,3,2)，进行比较，分出先后次序。

任何 range 如果拥有 n 个元素，其排列顺序的个数有限，最多为 n![34]。概念上你可以想像将所有排列顺序安排于一个表格中，将它们以词汇编纂顺序（lexicographical ordering）进行排序。表格中的每一个排列顺序，其所谓「前一个」与「下一个」排列顺序，都有明确的定义。

算法 next_permutation 会将元素所形成的 range [first,last) 转换为表格中的下一个排列顺序 — 也就是「字典排序方式」中的下一个较大排列顺序。

---

[34] 如果 range 内的所有元素都不相同，则正好有 n! 种排列。然而如果某些元素相同，则排列的数量便比较少。例如 (1,2,2) 只有 (3!/2) 种排列。

若有这样的排列顺序存在，next_permutation 便将 [first,last) 转换为该排列顺序，并返回 true。否则就将 [first,last) 转换成词汇编纂方式下的最小排列顺序（根据定义，那便是「以递增顺序排列」的一种排列顺序），并返回 false。

当且仅当（if and only if）返回值为 true，其必然结果是：元素的新排列顺序在字典排序方式下（以 lexicographical_compare 判断）一定大于旧者。

next_permutation 有两个版本，其差别在于如何定义某个元素小于另一个，并因此决定两个 ranges 中的哪一个在「字典排序」方式下小于另一个。第一版本采用 operator<，第二版本采用外界提供的 function object comp。

➢ 何处定义

HP STL 将 next_permutation 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- BidirectionalIterator 之 value type 为 Strict Weakly Comparable。

第二个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- BidirectionalIterator value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。至多调用 swap（p.237）共 (last-first)/2 次。

➢ 样例

以下枚举元素 (1,2,3,4) 的每一个排列顺序。本程序输出 4! = 24 行。第一个排列顺序为 (1,2,3,4)，最后一个为 (4,3,2,1)。

```
int main() {
  const int N = 4;
  int A[N] = {1, 2, 3, 4};

  do {
    copy(A, A + N, ostream_iterator<int>(cout, " "));
    cout << endl;
  } while (next_permutation(A, A + N));
}
```

「枚举 range 中的所有排列顺序」，这种做法是「已知排序法中最差的一个」。大部分优良排序算法的复杂度为 *O(N log(N))*，即使最糟的也只是 $O(N^2)$。下面这个愚蠢例子展示一个 *O(N!)* 排序算法。

```
template <class BidirectionalIterator, class StrictWeakOrdering>
void snail_sort(BidirectionalIterator first, BidirectionalIterator last,
                StrictWeakOrdering comp)
{
  while (next_permutation(first, last, comp)) { }
}
```

12.7.6 prev_permutation

```
(1) template <class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);

(2) template <class BidirectionalIterator, class StrictWeakOrdering>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last,
                          StrictWeakOrdering comp);
```

元素集合的排列顺序（permutation）可以采用字典排列方式（即词汇编纂（lexicographical）次序）来安排；完整讨论请参见 p.269。由于排列顺序的个数有限，所以词汇编纂方式能够清楚定义出「前一个」与「下一个」排列顺序。

算法 prev_permutation 会将元素所形成的区间 [first,last) 转换为词汇编纂顺序下的「前一个」排列顺序。

如果这样的排列顺序存在，prev_permutation 会将 [first,last) 转换为该排列顺序，并返回 true。如不存在（也就是说如果 [first,last) 的每个排列顺序都大于原本的排列顺序），就将 [first,last) 转换为词汇编纂顺序下最大的排序组合（根据定义，那便是「以递减顺序排列」的一种排列顺序），并返回 false。

当且仅当（if and only if）返回值为 true，其必然结果是：元素的新排列顺序在字典排序方式下（以 lexicographical_compare 判断）一定小于旧者。

prev_permutation 有两个版本，其差别在于如何定义某个元素小于另一个，并因此决定两个 ranges 中的哪一个在「字典排序」方式下小于另一个。第一版本采用 operator<，第二版本采用外界提供的 function object comp。

➢ 何处定义

HP STL 将 prev_permutation 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- BidirectionalIterator 之 value type 为 Strict Weakly Comparable。

第二个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- BidirectionalIterator 之 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。至多调用 swap（p.237）共 (last-first)/2 次。

➢ 样例

```
int main() {
  int A[] = {2, 3, 4, 5, 6, 1};
  const int N = sizeof(A) / sizeof(int);

  cout << "Initially:            ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  prev_permutation(A, A+N);
  cout << "After prev_permutation: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
```

```
  next_permutation(A, A+N);
  cout << "After next_permutation: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

12.8 分割（Partitions）

12.8.1 partition

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

算法 partition 会依据 function object pred 重新排列 [first,last) 的元素，使满足 pred 的元素排在不满足 pred 的元素之前。

partition 的必然结果是：[first,last) 内会存在某个 iterator middle，使得 [first,middle) 内的每个 iterator i 都导致 pred(*i) 为 true，并使得 [middle,last) 内的每个 iterator i 都导致 pred(*i) 为 false。注意，partition 不必保存元素原来的相对位置。它会保证 [first,middle) 中的元素都满足 pred 而 [middle,last) 中的元素都不满足 pred，但不对 [first,middle) 与 [middle,last) 中的元素顺序做任何保证。相较之下，另一个算法 stable_partition（p.274）就会保持原本的相对顺序。

返回值为 iterator middle。

➢ 何处定义

HP STL 将 partition 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- Predicate 为 Predicate 之 model。
- BidirectionalIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰套用 pred 共 (last-first) 次，并至多调用 swap（p.237）共 (last-first)/2 次。

➢ 样例

将某个序列重新排列，使偶数排在奇数之前。注意，尽管初始值已排序妥当，但执行完毕后偶数与奇数都不保证排序好，因为 partition 并不保持元素的相对顺序，这一点和 stable_partition 不同。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  const int N = sizeof(A)/sizeof(int);
  partition(A, A + N,
            compose1(bind2nd(equal_to<int>(), 0),
                     bind2nd(modulus<int>(), 2)));
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  // 输出 "10 2 8 4 6 5 7 3 9 1"
}
```

12.8.2 stable_partition

```
template <class ForwardIterator, class Predicate>
ForwardIterator
stable_partition(ForwardIterator first, ForwardIterator last,
                 Predicate pred);
```

算法 stable_partition 与 partition 类似。它依据 function object pred 重新排列 [first,last) 中的元素，使得满足 pred 的元素排在不满足 pred 的元素前。

和 partition 一样，stable_partition 也有其必然结果：[first,last) 内会存在某个 iterator middle，使得 [first,middle) 内的每个 iterator i 都导致 pred(*i) 为 true，并使得 [middle,last) 内的每个 iterator i 都导致 pred(*i) 为 false。

返回值为 iterator middle。

算法 partition 和 stable_partition 的不同点在于，statble_partition 能够保持元素的相对顺序。如果 x 和 y 是 [first,last) 中造成 pred(x) == pred(y) 的两个元素，而且 x 在 y 之前，那么 stable_partition 执行之后 x 仍然在 y 之前。

「保持相对顺序」是 stable_partition 与 partition 语义上的唯一不同点。由于 partition 执行速度较快，所以除非稳定性（stability）对你而言很重要，否则你不应该使用 stable_partition。

➢ 何处定义

HP STL 将 stable_partition 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

- ForwardIterator 为 Forward Iterator 之 model[35]。
- Predicate 为 Predicate 之 model。
- ForwardIterator 之 value type 可转换为 Predicate 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

和 partition 不同，stable_partition 是一个 ***adaptive*** 算法：它会试图分配临时内存缓冲区，而其运行期复杂度取决于缓冲区中有多少内存。最坏情况下（没有配得任何内存）至多调用 swap（p.237）共 ***N log(N)*** 次，其中 N 为 last-first。最好情况下（缓冲区中有足够的内存）与 N 呈线性关系。这两种情况下，pred 都正好被套用 N 次。

注意，partition 比较快速。它不需任何临时缓冲区，而且总是线性。

➢ 样例

重新排列一个序列，使偶数排在奇数之前。序列初值已经过排序。对它进行分割后，偶数与奇数仍然排过序。这是因为 stable_partition 会保持元素的相对顺序，这一点和 partition（p.273）不同。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  const int N = sizeof(A)/sizeof(int);
  stable_partition(A, A + N, compose1(bind2nd(equal_to<int>(), 0),
                                      bind2nd(modulus<int>(), 2)));
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  // 输出 "2 4 6 8 10 1 3 5 7 9"
}
```

12.9 随机重排与抽样（Random Shuffling and Sampling）

本节所列的第一个算法 random_shuffle，和 12.7 节所列的一样都属于排列算法。它会在「保持原初始值」的情况下重新排列 range 中的数值。本节的另两个算法就不是排列算法，它们并不随机重排 range，其执行动作与「从 input range 中选择一个随机子集」有十分密切的关联。

---

[35] C++ standard 规定，iterator type 须为 Bidirectional Iterator 之 model，但这其实不必要。以任何 Forward Iterator 来实现 stable_partition 都是可行的。

12.9.1 random_shuffle

```
(1) template <class RandomAccessIterator>
    void random_shuffle(RandomAccessIterator first,
                        RandomAccessIterator last);

(2) template <class RandomAccessIterator, class RandomNumberGenerator>
    void random_shuffle(RandomAccessIterator first,
                        RandomAccessIterator last,
                        RandomNumberGenerator& rand);
```

算法 random_shuffle 随机重排 [first,last) 的顺序。也就是说它在 N! 种可能的元素排列顺序中随机选出一种，此处 N 为 last-first。

N 个元素的序列，其排列方式共有 N! 种，random_shuffle 会产生一种均匀分布，因此任何特定排列顺序被选中的几率为 1/N!。这一点相当重要，因为有不少算法的第一阶段必须实现序列的随机重排（random shuffling），但却没有产生出「在 N! 个可能排列顺序上的均匀分布（uniform distribution）」，这很容易造成随机重排的错误。

这个算法描述于 Knuth 书籍 [Knu98a] 的 3.4.2 节。Knuth 将荣耀归功于 Moses 和 Oakford [MO63]，以及 Durstenfeld [Dur64]。

random_shuffle 有两个版本，其差别在于乱数的取得。版本一使用内部乱数产生器，版本二使用 Random Number Generator，那是一种特别的 function object，被当成引数传递进来，传递方式是 by reference 而非 by value，因为 Random Number Generator 的重要特点是具有局部状态，每次被调用时都会改变。

使用乱数时，「能够明白设定该乱数产生器的种子（seed）」有时候是很重要的。如果这样的控制对你的程序的确是重要的，你就必须使用第二版本。

➢ 何处定义

HP STL 将 random_shuffle 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomNumberGenerator 为 Random Number Generator 之 model。
- RandomAccessIterator 之 difference type 可转换为 RandomNumberGenerator 的引数型别。

➢ 先决条件

- [first,last) 为有效的 range。
- last-first 小于 rand 的最大值。

➢ 复杂度

与 last-first 成线性关系。对于非空的 range，random_shuffle 恰恰调用 swap 共 (last-first)-1 次。

➢ 样例

```
int main()
{
  const int N = 8;
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8};
  random_shuffle(A, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
  // 结果可能是 7 1 6 3 2 5 4 8,
  //  或 40,319 种其他可能性之一。
}
```

12.9.2 random_sample

```
(1) template <class InputIterator, class RandomAccessIterator>
    RandomAccessIterator
    random_sample(InputIterator first, InputIterator last,
                  RandomAccessIterator ofirst, RandomAccessIterator olast);

(2) template <class InputIterator, class RandomAccessIterator,
              class RandomNumberGenerator>
    RandomAccessIterator
    random_sample(InputIterator first, InputIterator last,
                  RandomAccessIterator ofirst, RandomAccessIterator olast,
                  RandomNumberGenerator& rand);
```

算法 random_sample 会随机地将 [first,last) 内的一个取样结果复制到 [ofirst, olast) 中。它会复制 n 个元素，此处 n 为 min(last-first, olast-ofirst)。Input range 内的每个元素最多在 output range 中出现一次，该取样结果系以均匀几率的方式选出。

返回值为 ofirst+n。

面对 N 个元素，如果忽略元素的顺序，欲选出 n 个元素时，共有 N!/(n!(N-n)!) 种方法。本算法会产生均匀分布的结果，换句话说任何特定元素被取出的几率为 n/N，而任何特定取样（不考虑元素顺序）的几率为 n!(N-n)!/N!。

Input range 中的相对顺序并不保证会被保留[36]下来，换句话说 output range 中的元素可能以任何顺序出现。

这是 Knuth 书籍 [Knu98a] 3.4.2 节所说的「算法 R」。Knuth 将此归功于 Alan Waterman。

random_sample 有两个版本，其差别在于乱数的取得。第一版本使用内部乱数产生器，第二版本使用 Random Number Generator，那是一种特殊的 function object，被当成引数传递进来。

使用乱数时，「能够明白设定该乱数产生器的种子（seed）」有时候是很重要的。如果这样的控制对你的程序的确是重要的，你就必须使用第二版本。

➢ 何处定义

算法 random_sample 并没有出现在原始的 HP STL 之中，也不存在于 C++ standard 中。SGI STL 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- InputIterator 之 value type 可转换为 RandomAccessIterator 之 value type。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomNumberGenerator 为 Random Number Generator 之 model。
- InputIterator 之 value type 可转换为 RandomAccessIterator 之 value type。
- RandomAccessIterator 之 difference type 可转换为 RandomNumberGenerator 的引数型别。

➢ 先决条件

- [first,last) 为有效的 range。
- [ofirst,olast) 为有效的 range。

---

[36] 如果你的程序需要保持 input range 中的元素相对顺序，你应该改用 random_sample_n（p.279）。random_sample_n 的主要限制是：input range 必须以 Forward Iterators 而非 Input Iterators 构成。

- [first,last) 与 [ofirst,olast) 不相重叠。
- last-first 小于 rand 的最大值。

➢ **复杂度**

与 last-first 成线性关系。最多有 last-first 个元素从 input range 复制到 output range。

➢ **样例**

```
int main()
{
  const int N = 10;
  const int n = 4;
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int B[n];

  random_sample(A, A+N, B, B+n);
  copy(B, B + n, ostream_iterator<int>(cout, " "));
  cout << endl;
  // 打印结果可能是 1 6 3 5,
  //  或 5039 种可能中的任何一种。
}
```

12.9.3 random_sample_n

```
(1) template <class ForwardIterator, class OutputIterator, class Distance>
    OutputIterator
    random_sample_n(ForwardIterator first, ForwardIterator last,
                    OutputIterator out, Distance n);

(2) template <class ForwardIterator, class OutputIterator, class Distance,
              class RandomNumberGenerator>
    OutputIterator
    random_sample_n(ForwardIterator first, ForwardIterator last,
                    OutputIterator out, Distance n,
                    RandomNumberGenerator& rand);
```

算法 random_sample_n 能够随机地将 [first,last) 中的某些元素复制到 [out,out + n) 中。它会复制 m 个元素，此处 m 为 min(last-first,n)。Input range 中的每个元素最多会在 output range 中出现一次，并以均匀几率的方式取样。

返回值为 ofirst+m。

面对 N 个元素，如果忽略元素的顺序，欲选出 n 个元素时，共有 N!/(n!(N-n)!) 种方法。本算法会产生均匀分布的结果，换句话说任何特定元素被取出的几率为 n/N，而任何特定取样（不考虑元素顺序）的几率为 n!(N-n)!/N!。

random_sample 与 random_sample_n 之间一个重要的差别在于，random_sample_n 会保持元素在 input range 中的相对顺序，而 random_sample 不会。这两个算法之间的主要不同点在于：random_sample_n 要求其 input range 必须是 Forward Iterators，其 output range 只要是 Output Iterators 即可，而 random_sample 要求其 input range 必须是 Input Iterators，其 output range 必须是 Random Access Iterators。

这是 Knuth 书籍 [Knu98a] 3.4.2 节所说的「算法 S」。Knuth 将此归功于 Fan、Muller、 Rezucha [FMR62] 以及 T. G. Jones。

random_sample_n 有两个版本，差别在于乱数的取得。第一版本使用内部乱数产生器，第二版本使用 Random Number Generator，那是一种特别的 function object，被当成引数传递进来。

➢ 何处定义

算法 random_sample_n 并没有出现在原始的 HP STL 之中，也不存在于 C++ standard 中。SGI STL 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- ForwardIterator value type 可转换为 OutputIterator value types 所形成之集合内的某型别。
- Distance 是一种整数型别，其大小足以表现数值 last-first。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- RandomNumberGenerator 为 Random Number Generator 之 model。
- Distance 是一种整数型别，其大小足以表现数值 last-first。
- ForwardIterator value type 可转换为 OutputIterator value types 所形成之集合内的某型别。
- Distance 可转换为 RandomNumberGenerator 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。
- n 不为负数。
- [first,last) 与 [out,out+n) 不相重叠。

- 有足够大的空间容纳所有被复制的元素。更正式地说是：[out,out+min(n,last-first)) 为有效的 range。
- last-first 小于 rand 的最大值。

➢ 复杂度

与 last-first 成线性关系。Input range 中最多有 last-first 元素被审阅，而且恰恰有 min(n,last-first) 个元素被复制到 output range 中。

➢ 样例

```
int main()
{
  const int N = 10;
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  random_sample_n(A, A+N, ostream_iterator<int>(cout, " "), 4);
  cout << endl;
  // 打印结果可能是 3 5 6 10,
  //  或 209 种可能性中的任何一种。
}
```

12.10 一般化之数值算法（Generalized Numeric Algorithms）

12.10.1 accumulate

```
(1) template <class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);

(2) template <class InputIterator, class T, class BinaryFunction>
    T accumulate(InputIterator first, InputIterator last, T init,
                 BinaryFunction binary_op);
```

算法 accumulate 为总和运算的一般型。它可以计算 init 和 [first,last) 内的所有元素的总和（或其他双参运算）。

注意，accumulate 的接口有点笨拙。你一定得提供一个初始值 init（这么做的原因之一是当 [first,last) 为空时仍获得一个明确定义的值。）如果你希望计算 [first,last) 中所有数值的总和，你应该将 init 设为 0。

式中的双参操作符不必具备交换性（commutative）和结合性（associative）。所有的 accumulate 运算行为的顺序都有明确设定：首先将 init 初始化，然后针对 [first,last) 之每一个 iterator i，由起头到结尾，执行 result = result + *i（版本一）或 result = binary_op(result, *i)（版本二）。

➢ 何处定义

HP STL 将 accumulate 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <numeric>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- T 为 Assignable 之 model。
- 如果 x 为型别 T 之 object，y 为 InputIterator value type 之 object，则 x + y 是有定义的。
- x + y 的返回型别可转换为 T。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- T 为 Assignable 之 model。
- BinaryFunction 为 Binary Function 之 model。
- T 可转换为 BinaryFunction 的第一引数型别。
- InputIterator 之 value type 可转换为 BinaryFunction 之第二引数型别。
- BinaryFunction 之返回型别可转换为 T。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。恰恰调用双参运算共 last-first 次。

➢ 样例

计算 range 内所有元素的总和以及乘积。本例同时示范 accumulate 两个版本的用法。由于加法与乘法具有不同的证同元素（identity element），所以计算总和时，我提供 0 为初始元素，计算乘积时则以 1 为初始元素。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5};
  const int N = sizeof(A) / sizeof(int);

  cout << "The sum of all elements in A is "
       << accumulate(A, A + N, 0)
```

```
        << endl;

  cout << "The product of all elements in A is "
       << accumulate(A, A + N, 1, multiplies<int>())
       << endl;
}
```

对于非空的 range，很容易就能定义 accumulate 的一个 wrapper（外覆函数），让你不必再提供 init 引数。但如果 [first,last) 是空的，就不能使用这个 wrapper，因为对于返回值没有一个适当选择。

```
template <class InputIterator>
typename iterator_traits<InputIterator>::value_type
accumulate_nonempty(InputIterator first, InputIterator last)
{
  assert(first != last);
  typename iterator_traits<InputIterator>::value_type sum = *first++;
  return accumulate(first, last, sum);
}

template <class InputIterator, BinaryFunction>
typename iterator_traits<InputIterator>::value_type
accumulate_nonempty(InputIterator first, InputIterator last,
                    BinaryFunction binary_op)
{
  assert(first != last);
  typename iterator_traits<InputIterator>::value_type init = *first++;
  return accumulate(first, last, init, binary_op);
}
```

12.10.2 inner_product

```
(1) template <class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2,
                    T init);

(2) template <class InputIterator1, class InputIterator2, class T,
              class BinaryFunction1, class BinaryFunction2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init,
                    BinaryFunction1 binary_op1,
                    BinaryFunction2 binary_op2);
```

算法 inner_product 能够计算 [first1,last1) 和 [first2,first2+(last1-first1))的一般化内积（generalized inner product）。

注意，inner_product 的接口有点笨拙。你一定得提供一个初始值 init。这么做的原因之一是当 [first,last) 为空时，仍可获得一个明确定义的结果。如果你想计算两个 vectors 的一般内积，应该将 init 设为 0。

第一个版本会将两个 ranges 的内积结果加上 init 。也就是说先将结果初始化为 init，然后针对 [first1,last1) 的每一个 iterator i，由起头到结尾，执行 result = result + (*i) * *(first2+(i-first1))。

第二版本与第一版本的唯一差异是以外界提供之 function object 来取代 operator+ 和 operator*。也就是说，首先将结果初始化为 init，然后针对 [first1,last1) 的每一个 iterator i，由起头到结尾，执行 result = binary_op1(result, binary_op2(*i, *(first2+(i-first1)))。

式中所用的双参操作符不必具备交换性（commutative）与结合性（associative）。inner_product 所有运算行为的顺序都有明确设定。

➢ 何处定义

HP STL 将 inner_product 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <numeric>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- T 为 Assignable 之 model。
- 如果 x 为型别 T 之 object，y 为 InputIterator1 value type 之 object，z 为 InputIterator2 value type 之 object，则 x + y * z 是有定义的。
- x + y * z 的型别可转换为 T。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- T 为 Assignable 之 model。
- BinaryFunction1 为 Binary Function 之 model。
- BinaryFunction2 为 Binary Function 之 model。
- InputIterator1 之 value type 可转换为 BinaryFunction2 之第一引数型别。
- InputIterator2 之 value type 可转换为 BinaryFunction2 之第二引数型别。
- T 可转换为 BinaryFunction1 的第一引数型别。

- BinaryFunction2 之返回型别可转换为 BinaryFunction1 之第二引数型别。
- BinaryFunction1 之返回型别可转换为 T。

➢ 先决条件

- [first1,last1) 为有效的 range。
- [first2,first2+(last1-first1)) 为有效的 range。

➢ 复杂度

线性。恰恰套用每个双参运算共 last1-first1 次。

➢ 样例

```
int main()
{
  int A1[] = {1, 2, 3};
  int A2[] = {4, 1, -2};
  const int N1 = sizeof(A1) / sizeof(int);

  cout << "The inner product of A1 and A2 is "
       << inner_product(A1, A1 + N1, A2, 0)
       << endl;
}
```

12.10.3 partial_sum

```
(1) template <class InputIterator, class OutputIterator>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result);

(2) template <class InputIterator, class OutputIterator,
              class BinaryFunction>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result,
                               BinaryFunction binary_op);
```

算法 partial_sum 用来计算部分总和（partial sum）。它会将 *first 赋值给 *result，将 *first 和 *(first+1) 的和赋值给 *(result+1)，依此类推。注意，result 可以等于 iterator first。这对于就地（in place）计算部分总和很有帮助。

运算中的总和首先初始为 *first，然后赋值给 *result。对于 [first+1,last) 中每个 iterator i，从起头到结尾，执行 sum = sum + *i（第一版本）或 sum = binary_op(sum, *i)（第二版本），然后再将 sum 赋值给 *(result + (i - first))。此式所用之双参操作符不必具备交换性（commutative）与结合性（associative）。所有运算行为的顺序都有明确设定。

本算法返回 output range 的尾端，result+(last-first)。

➢ 何处定义

HP STL 将 partial_sum 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <numeric>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- 如果 x 和 y 为 InputIterator value type 的 object，则 x + y 是有定义的。
- x + y 的返回型别可转换为 InputIterator 的 value type。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- BinaryFunction 为 Binary Function 之 model。
- InputIterator 之 value type 可转换为 BinaryFunction 之第一与第二引数型别。
- BinaryFunction 之 result type 可转换为 InputIterator 之 value type。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

- [first,last) 为有效的 range。
- [result,result+(last-first)) 为有效的 range。

➢ 复杂度

线性。如果 [first,last) 为空，双参运算行为将执行 0 次，否则恰恰执行 (last-first)-1 次。

➢ 样例

```
int main()
{
  const int N = 10;
```

```
  int A[N];

  fill(A, A+N, 1);
  cout << "A:                ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Partial sums of A: ";
  partial_sum(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

12.10.4 adjacent_difference

```
(1) template <class InputIterator, class OutputIterator>
    OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result);

(2) template <class InputIterator, class OutputIterator,
              class BinaryFunction>
    OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result,
                        BinaryFunction binary_op);
```

算法 adjacent_difference 可用来计算 [first,last) 中相邻元素的差额。也就是说它将 *first 赋值给 *result，并针对 [first+1,last) 内的每个 iterator i，将 *i 与 *(i-1) 之差值赋值给 *(result+(i-first))。

注意，result 可以等于 iterator first。因此你可以利用 adjacent_difference 就地（in place）计算差额。

「存储第一元素之值，然后存储后继元素之差值」这种做法很有用，因为这么一来便有足够的信息可以重建 input range。如果加法与减法拥有常用的算术定义，那么 adjacent_difference 与 partial_sum 互为相反运算。

第一版本使用 operator- 来计算差额，第二版本采用外界提供之双参函数。第一个版本针对 [first+1,last) 中的每个 iterator i，将 *i - *(i-1) 赋值给 *(result+(i-first))，第二个版本则是将 binary_op(*i, *(i-1)) 的运算结果赋值给 *(result+(i-first))。

➢ 何处定义

HP STL 将 adjacent_difference 声明于头文件 <algo.h>。C++ standard 将它声明于 <numeric>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- 如果 x 和 y 为 InputIterator value type 的 object，则 x - y 是有定义的。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。
- x - y 的返回型别可转换为 OutputIterator value types 所形成之集合内的某个型别。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- BinaryFunction 为 Binary Function 之 model。
- InputIterator 之 value type 可转换为 BinaryFunction 之第一与第二引数型别。
- InputIterator value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。
- BinaryFunction result type 可转换为 OutputIterator value types 所形成之集合内的某型别。

➢ 先决条件

- [first,last) 为有效的 range。
- [result,result+(last-first)) 为有效的 range。

➢ 复杂度

线性。如果 [first,last) 为空，双参运算的执行次数为 0，否则恰恰执行 (last-first)-1 次。

➢ 样例

这个例子首先产生整数数组的部分差额，然后再以 partial_sum 重建原来的数组。

```
int main()
{
  int A[] = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100};
  const int N = sizeof(A) / sizeof(int);
  int B[N];

  cout << "A:          ";
  copy(A, A + N, ostream_iterator<int>(cout, " "));
```

```
  cout << endl;

  adjacent_difference(A, A + N, B);
  cout << "Differences: ";
  copy(B, B + N, ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Reconstruct: ";
  partial_sum(B, B + N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

第 13 章

排序和查找
Sorting and Searching

在计算机科学领域中，某些最重要的算法系用来处理一般的数值排序问题。本章将描述 STL 的排序算法（可作用于整个 range 或部分 range），以及作用于已序（sorted）ranges 身上的算法。

本章所提的所谓「已序（sorted）range」，是一种以递增顺序排列的 range。由于 range 可能内含重复（duplicate）元素，所以更精确地说它是一种以非递减顺序排列的 range。不过本章的所有算法都已完全一般化，都具有一个「接受 function objects 作为参数」的版本，所以你一定可以根据你的应用，完成适当的排序方式。

甚至我刚刚所说「range 可能内含重复（*duplicate*）元素」也是不够精确的，应该说「range 可能包含一些等价（*equivalent*）元素」。

本章所有的算法都以 strict weak ordering 来比较元素，其原因和「等价关系（equivalence）之所以重要」以及「等价（equivalence）并非就是相等（equality）」的原因相同。一如 Strict Weakly Comparable 需求条件（p.88）中所叙述，当你定义一个 strict weak ordering，你同时也隐然地定义了一个等价概念。如果两个元素之间没有任何一个小于另一个的话，这两个元素就是等价的。举例来说，如果你以「姓氏」来对人名排序，当两个人的姓氏相同，这两个人就是等价的。

「等价（equivalent）」一词在本章的意思如上所述。本章没有任何一个算法是以元素相等性（equality）作为比较依据。

13.1 对某个区间排序（Sorting Ranges）

本节涵盖 STL 的排序算法，也就是能够对 ranges 进行排序的算法。排序算法是一种排列算法（permuting algorithms），它和 12.7 节的算法一样，会重新安排 range 中的数值，但不引入新数值。

除了本章所列的算法，STL 还包含三种其他形式的排序法：第一，containers `list`（p.441）和 `slist`（p.448）都拥有 member function `sort`，可用来对整个 container 排序；第二，Sorted Associative Containers（p.156）的元素一定（自动）处于已序状态；第三，你可以使用 heap sort 算法来对任何 range

排序：首先调用 make_heap（p.336）构造一个 heap，然后调用 sort_heap（p.342）将这个 heap 转换成一个已序的 range。

没有任何一种排序算法（在各种情况下）永远是最好的。由于排序是一种极重要而又潜在性地极耗时间的行为，所以你应该小心选择你的排序算法；通常 sort 是个好选择，但有时候其他方法也许更恰当。

13.1.1 sort

```
(1) template <class RandomAccessIterator>
    void sort(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void sort(RandomAccessIterator first, RandomAccessIterator last,
              StrictWeakOrdering comp);
```

算法 sort 可以将 [first,last) 中的元素以递增顺序（或更严格地说是非递减顺序）排序，意思是如果 i 和 j 是 [first,last) 中的任两个可提领的 iterators，而且 i 在 j 之前，那么保证 *j 不小于 *i。

注意，sort 不一定是个稳定（*stable*）排序算法。假设 *i 和 *j 等价，也就是说没有任何一个小于另一个，sort 并不保证这两个元素的相对位置在排序后还会一样。

稳定性（stability）通常是个无关紧要的准则。只有当 range 内含「等价（equivalent）但不完全相同（identical）[37]」的元素，而且排序前后这些元素的相对次序必须不变时，稳定性才显得重要。

当你要对多字段数据进行排序的时候，稳定性（stability）会有举足轻重的影响。你可能想要先以某个字段排序，再以另一个字段排序，这种情况下第二个排序行为是否稳定（stable）就相当重要了。算法 stable_sort（p.294）如其名称所暗示，能够保持等价元素间的相对顺序。

sort 有两个版本，其间差异在于如何定义某元素是否小于另一元素。第一版本采用 operator< 来比较，第二版本采用 function object comp 做比较。第一版本必导致 is_sorted(first, last)（p.303）为 true，第二版本必导致 is_sorted(first, last, comp) 为 true。

➢ 何处定义

HP STL 将 sort 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

---

[37] 以「姓氏」对人名排序便是一个好例子。就排序的意义而言，同姓氏的两笔数据被视为等价，但由于「名字」可能不一样，所以等价的两笔数据并不一定完全相同。

➢ 型别条件

第一个版本：

- `RandomAccessIterator` 为 Random Access Iterator 之 model。
- `RandomAccessIterator` 是可变的（mutable）。
- `RandomAccessIterator` 的 value type 为 Strict Weakly Comparable。

第二个版本：

- `RandomAccessIterator` 为 Random Access Iterator 之 model。
- `RandomAccessIterator` 是可变的（mutable）。
- `StrictWeakOrdering` 为 Strict Weak Ordering 之 model。
- `RandomAccessIterator` 的 value type 可转换为 `StrictWeakOrdering` 的引数型别。

➢ 先决条件

`[first,last)` 是有效的 range。

➢ 复杂度

根据 C++ standard 的要求，`sort` 平均执行 $O(N \log N)$ 个比较动作，最坏情况下的复杂度为 $O(N^2)$，其中 `N` 是 `last-first`。

然而，允许如此宽松的复杂度并没有必要。我们可以将 `sort` 实现成不论平均状态或最坏情况下，都只执行 $O(N \log N)$ 个比较动作。SGI STL 以及其他产品都可以做到这样的保证。

早期的 `sort` 版本之所以有二次方（quadratic）最坏情况，因为它们通常都使用 quicksort 算法 [Hoa62]，这个算法选择三者中点（median-of-three）当作枢轴（pivot）[Sin69]。Quicksort 的平均复杂度为 $O(N \log N)$，最坏情况之复杂度则为 $O(N^2)$。参见 Knuth 书籍 [Knu98b] 5.2.2 节。

近来的 `sort` 版本使用 introsort 算法 [Mus97]，使其最坏情况下的复杂度进步到 $O(N \log N)$。Introsort 类似 median-of-three quicksort，平均而言它至少像 quicksort 一样快。

➢ 样例

试将整数 range 以递增顺序排序。这是 `sort` 的最简单使用方式。

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);
  sort(A, A + N);
  assert(is_sorted(A, A + N));
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
```

```
  // 输出为 "1 2 4 5 7 8"
}
```

上个例子中，我们将一个整数 range 做递增顺序排序。以递减顺序排序也是一样简单：我们可以传入适当的 function object 给 sort 的第二版本。Function object greater（p.386）是一种 Strict Weak Ordering，因此它是个合理的选择。根据定义，如果某 range 系根据 greater 来排序，它就会以递减顺序来排序。

```
int main()
{
  vector<string> fruits;
  fruits.push_back("apple");
  fruits.push_back("banana");
  fruits.push_back("pear");
  fruits.push_back("grapefruit");
  fruits.push_back("cherry");
  fruits.push_back("orange");
  fruits.push_back("watermelon");
  fruits.push_back("mango");

  sort(fruits.begin(), fruits.end(),
       greater<string>());
  assert(is_sorted(fruits.begin(), fruits.end(),
        greater<string>()));

  copy(fruits.begin(), fruits.end(),
       ostream_iterator<string>(cout, "\n"));
}
```

13.1.2 stable_sort

```
(1) template <class RandomAccessIterator>
    void
    stable_sort(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                     StrictWeakOrdering comp);
```

算法 stable_sort 十分类似 sort（p.292）。它可以将 [first,last) 中的元素以递增顺序（或更严格地说是以非递减顺序）排序。

stable_sort 和 sort 之间有两个重要的相异处。第一，它们的运行期复杂度不同。通常 sort 比较快。第二，正如其名称所暗示，stable_sort 是稳定的（*stable*），能够保持等价元素间的相对顺序。如果 x 和 y 都是 [first,last) 内的元素，x 在 y 之前，而且这两个元素是等价的（没有任何一个小于另一个），那么 stable_sort 执行之后 x 仍然在 y 之前。sort 就不保证如此。

稳定性的保证很重要，因为在某种顺序下，两元素可能等价（equivalent）但不完全一致（identical）。常见的例子便是以姓氏来对人名排序。如果两个人同姓但不同名，在这种比较之下（没有任何一个小于另一个），虽然他们并不完全一致（identical），但被视为等价（equivalent）。这是 stable_sort 可以发挥之处。如果你要对多字段数据进行排序，你可能希望以某个字段排序，但不希望破坏先前以另一字段排序后的结果。例如你可能会先以姓氏排序，然后再以名字来进行稳定排序。

stable_sort 有两个版本，其间差异在于如何定义某个元素小于另一个元素。第一版本使用 operator< 进行比较，第二版本使用 function object comp 进行比较。

第一版本必然导致 is_stable_sorted(first, last) 为 true 的结果，第二版本必然导致 is_stable_sorted(first, last, comp) 为 true 的结果。两个版本都能够维持 stable_sort 排序前的相对顺序。

stable_sort 内部系以 merge sort 算法实现；请参考 p.316 以及 Knuth 书籍 [Knu98b] 5.2.4 节。

➢ 何处定义

HP STL 将 stable_sort 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ 先决条件

[first,last) 是有效的 range。

➢ 复杂度

和 sort 不同，stable_sort 是一种 ***adaptive*** 算法。它会试图分配临时内存缓冲区，而其运行期复杂度取决于究竟分配了多少内存。在没有分配到任何辅助内存的最坏情况下，将执行 $O(N(\log N)^2)$ 个比较动作，此处的 N 为 last-first。最好情况下（分配了足够内存）为 $O(N(\log N))$。sort 和 stable_sort 两者都是 $O(N(\log N))$，但 sort 会快上某个常量倍。

➢ 样例

试排序一个字符序列，并忽略其中的大小写。注意，字符的相对顺序只有在维持大小写时才有差别。

```
inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  char A[] = "fdBeACFDbEac";
  const int N = sizeof(A) - 1;
  stable_sort(A, A+N, lt_nocase);
  cout << A << endl;
  // 输出结果为 "AaBbCcdDeEfF"
}
```

另一个例子，假设某个 vector 容纳了一系列数据项，每笔数据项都有其优先权。我们希望以此优先权来对这些数据项排序，如果优先权相同，位置在前的数据项仍应在前。这个例子可用来表示整批队列（batch queue）中的工作，或模拟进入窗口系统中的各消息。

```
class job {
public:
  enum priority_code {standby, normal, high, urgent};
  job(string id, priority_code p = normal) : nam(id), pri(p) {}

  string name() const { return nam; }
  priority_code priority() const { return pri; }

private:
  string nam;
  priority_code pri;
};

ostream& operator<<(ostream& os, const job& j) {
  os << j.name() << " (" << j.priority() << ")";
  return os;
}
```

```
bool operator<(const job& j1, const job& j2) {
  return j1.priority() > j2.priority();
}

int main() {
  vector<job> jobs;

  jobs.push_back(job("Long computation", job::standby));
  jobs.push_back(job("System reboot", job::urgent));
  jobs.push_back(job("Print"));
  jobs.push_back(job("Another long computation", job::standby));
  jobs.push_back(job("Copy file"));

  stable_sort(jobs.begin(), jobs.end());
  copy(jobs.begin(), jobs.end(), ostream_iterator<job>(cout, "\n"));
}
```

13.1.3 partial_sort

```
(1) template <class RandomAccessIterator>
    void partial_sort(RandomAccessIterator first,
                      RandomAccessIterator middle,
                      RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void partial_sort(RandomAccessIterator first,
                      RandomAccessIterator middle,
                      RandomAccessIterator last,
                      StrictWeakOrdering comp);
```

算法 partial_sort 可重新安排 [first,last)，使其中 middle-first 个最小元素以递增顺序排序。

partial_sort 会导致 input range 中最小的 middle-first 个元素以递增顺序排序，并包含于 [first,middle) 内。其余的 last-middle 个元素安置于 [middle,last) 中，无任何特定顺序。

注意，如果你调用 sort，同样也保证较小的 N 个元素以递增顺序包含于[first,first+N)。之所以选择 partial_sort 而非 sort，唯一理由是效率。如果只是挑出前 N 个最小元素来排序，会比对整个 range 排序快上许多。

partial_sort 有两个版本，差异在于如何定义某个元素小于另一个元素。第一版本使用 operator< 进行比较，第二版本使用 function object comp 进行比较。

正式地说，第一版本的必然结果如下。如果 i 和 j 是 [first,middle) 之中任意两个有效的 iterators，i 在 j 之前，而 k 是 [middle,last) 之中一个有效的 iterator，那么 *j<*i 和 *k<*i

两者皆不为真。第二版本的必然结果则是：comp(*j, *i) 和 comp(*k, *i) 都不为真。非正式的说法是，此一必然结果意味前 middle-first 个元素是以递增顺序排序，而且 [middle,last) 之中没有任一个元素会小于 [first,middle) 中的任一个元素。

partial_sort 内部系采用 heapsort 来实现；请参见 Williams [Wil64] 以及 Knuth [Knu98b] 5.2.3 节。

➢ 何处定义

HP STL 将 partial_sort 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ 先决条件

- [first,middle) 是有效的 range。
- [middle,last) 是有效的 range。

由这两个条件可推导出 [first,last) 也是有效的 range。

➢ 复杂度

大约执行 (last-first) log(middle-first) 次比较动作。

➢ 样例

试找寻整数数组中的前五个最小值。A[0]~A[4] 五个元素应以递增顺序排序，其余元素不保证任何顺序。

```
int main()
{
  int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
  const int N = sizeof(A) / sizeof(int);
```

```
  partial_sort(A, A + 5, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出结果为 "1 2 3 4 5 11 12 10 9 8 7 6"
}
```

一般来说，partial_sort(first,middle,last) 会从 [first,last) 中取出前 middle-first 个最小元素，将它们放入 [first,middle)，然后以递增顺序排序 [first,middle)。不过 middle == first 和 middle == last 是两个明显特例。

如果 middle == first，partial_sort 不会做什么有意义的事。它将 0 个最小元素放入（空的）[first,first) 之中，然后将其余元素（也就是所有元素）以未定顺序放入 [first,last)。partial_sort(first, first, last) 唯一保证的便是将 [first,last) 以未定顺序重新排列过。

middle == last 这个特例就有趣多了。它将 [first,last) 中的前 last-first 个最小元素取出，将它们放入 [first,last)，然后再以递增顺序排序 [first,last)。换句话说 partial_sort(first, last, last) 会对整个 [first,last) 重新排序。

```
int main()
{
  int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
  const int N = sizeof(A) / sizeof(int);

  partial_sort(A, A + N, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出结果为 "1 2 3 4 5 6 7 8 9 10 11 12"
}
```

什么理由让你非得以这样的方式使用 partial_sort 呢？如果你使用某个版本的 STL sort(p.292)，而该 sort 系以 introsort 实现出来，那么没有任何理由这么做。如果你要排序一个具有 N 个元素的 range，sort 与 partial_sort 都是 $O(N \log N)$，但通常 sort 会快上两倍。

但如果你使用旧版本的 STL sort，而该 sort 系以 quicksort 实现出来，那么有些情况你应该考虑以 partial_sort 取代 sort 来对整个 range 排序。因为虽然 quicksort 通常是 $O(N \log N)$，但在极少情况下会是 $O(N^2)$。

13.1.4 partial_sort_copy

```
(1) template <class InputIterator, class RandomAccessIterator>
    void partial_sort_copy(InputIterator first, InputIterator last,
                           RandomAccessIterator result_first,
                           RandomAccessIterator result_last);

(2) template <class InputIterator, class RandomAccessIterator,
              class StrictWeakOrdering>
    void partial_sort_copy(InputIterator first, InputIterator last,
                           RandomAccessIterator result_first,
                           RandomAccessIterator result_last,
                           StrictWeakOrdering comp);
```

如其名称所示，算法 partial_sort_copy 是 partial_sort 的复制式版本。然而它并没有遵照 STL 普遍的复制式版本做法，它并非直接在 partial_sort 之后调用 copy，而是像 partial_sort 运用 [first,middle) 那样地运用 [result_first,result_last)。

partial_sort_copy 会从 [first,last) 内复制前 N 个最小元素到 [result_first, result_first+N) 之中，这里的 N 是 last-first 和 result_last-result_first 两者中的最小值。被复制过去的元素系以递增顺序排序。

返回值为 result_first+N。

partial_sort_copy 有两个版本，其差异在于如何定义某个元素小于另一个元素。第一版本使用 operator< 来进行比较，第二版本使用 function object comp 来进行比较。第一版本必然导致 is_sorted(result_first,result_last) 为 true，第二版本必然导致 is_sorted(result_first, result_last, comp) 为 true。

➢ 何处定义

HP STL 将 partial_sort_copy 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator 为 Input Iterator 之 model。
- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- InputIterator 与 RandomAccessIterator 的 value types 是相同的。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable。

第二个版本：

- InputIterator 为 Input Iterator 之 model。
- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- InputIterator 与 RandomAccessIterator 的 value types 是相同的。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ 先决条件

- [first,last) 是有效的 range。
- [result_first,result_last) 是有效的 range。
- [first,last) 与 [result_first,result_last) 不可重叠。

➢ 复杂度

大约执行 (last-first) log(N) 个比较动作，这里的 N 是 last-first 和 result_last-result_first 之间的最小值。

➢ 样例

试将整数数组中的前 4 个最小值复制到另一个 vector 内。不可使用 partial_sort_copy 将结果直接复制到标准输出设备上，因为 partial_sort_copy 要求其 output range 必须由 Random Access Iterators 组成。

```
int main()
{
  int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
  const int N = sizeof(A) / sizeof(int);

  vector<int> V(4);
  partial_sort_copy(A, A + N, V.begin(), V.end());
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出结果为 "1 2 3 4"
}
```

13.1.5 nth_element

```
(1) template <class RandomAccessIterator>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last);
```

```
(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                     RandomAccessIterator last, StrictWeakOrdering comp);
```

就像 partial_sort（p.297）一样，算法 nth_element 也会对 range 内的元素进行部分排序。它会重新排列 [first,last)，使 iterator nth 所指元素与整个 [first,last) 排序后的的同一位置的元素相同。此外 iterator nth 亦可作为分割（partition）之用：它保证 [nth,last) 内没有任何一个元素小于 [first,nth) 内的元素。

与 partial_sort 不同之处在于，nth_element 并不保证 [first,nth) 或 [nth,last)都会被排序。它只保证 [first,nth) 内的元素都小于（或者更精确地说是不大于）[nth,last) 内的元素。以此点观之，nth_element 比较类似 partition（p.273）而非 sort 或 partial_sort。

由于 nth_element 比 partial_sort 的保证更少，所以它当然应该比较快。这也是 nth_element 得以取代 partial_sort 的主要考量之一。

nth_element 有两个版本，其差异在于如何定义某个元素小于另一个元素。第一版本使用 operator< 进行比较，第二个版本使用 function object comp 进行比较。

正式地说，第一版本的必然结果是：[first,nth) 内不存有任何一个 iterator i，使得 *nth < *i，[nth - 1,last) 中亦不存有任何一个 iterator j，使得 *j < *nth。

同样地，第二版本的必然结果是：[first,nth) 内不存在任何一个 iterator i，使得 comp(*nth, *i) 为 true，[nth+1,last) 内亦不存在任何一个 iterator j，使得 comp(*j, *nth) 为 true。

➢ 何处定义

HP STL 将 nth_element 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。

- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ **先决条件**

- [first,nth) 是有效的 range。
- [nth,last) 是有效的 range。

这两个先决条件可推导出 [first,last) 为有效的 range。

➢ **复杂度**

平均复杂度与 last-first 成线性关系。

注意，此复杂度大大低于 partial_sort 的运行期复杂度。除非 partial_sort 的其他保证对于你的应用程序很重要，否则你应该尽量以 nth_element 取代 partial_sort。

➢ **样例**

已知具有 12 个整数的数组 A，试排序之以达到以下要求：

1.元素 A[6] 的值与「整个数组重新排序后」同一位置的值相同。

2.A[0]~A[6] 的元素都小于 A[6]~A[11] 的元素。

```
int main()
{
  int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
  const int N = sizeof(A) / sizeof(int);

  nth_element(A, A + 6, A + N);
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

输出结果可能是：

```
5 2 6 1 4 3 7 8 9 10 11 12
```

或其他排列方式。这里唯一保证的是数组会被分成两段，第一段内的任何元素都不大于第二段内的元素。[A,A+6) 与 [A+7, A+12) 的内部顺序并非重点。

13.1.6 is_sorted

```
(1) template <class ForwardIterator>
    bool is_sorted(ForwardIterator first, ForwardIterator last);
```

```
(2) template <class ForwardIterator, class StrictWeakOrdering>
    bool is_sorted(ForwardIterator first, ForwardIterator last,
                   StrictWeakOrdering comp);
```

is_sorted 不会对 range 进行排序，只是用来测试某个 range 是否已经排序过了。如果 [first,last) 已排序，is_sorted 返回 true，否则返回 false。它并不会更动其 input range。

is_sorted 的先决条件、必然结果（或两者兼具）都如同本章许多算法一样。

is_sorted 有两个版本，其差异在于如何定义某个元素小于另一个元素。第一版本使用 operator< 来进行比较，第二版本使用 function object comp 来进行比较。

正式地说，当且仅当（if and only if）[first,last) 内的任意两个 iterators i, j，当 i 在 j 之前时导致 *j < *i 为 false（针对第一版本）或导致 comp(*j, *i) 为 false（针对第二版本），那么 is_sort 便会返回 true。

如果 [first,last) 是空的（也就是说当 first == last），两个 is_sorted 版本都返回 true。是的，空的 range 被视为已排序。

➢ 何处定义

is_sorted 算法并未出现于原始的 HP STL 内或 C++ standard 之中。SGI STL 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- ForwardIterator 的 value type 为 Strict Weakly Comparable。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- ForwardIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ 先决条件

- [first,last) 是有效的 range。

➢ 复杂度

线性。对于非空的 range，至多执行 (last-first)-1 次比较动作。

➢ **样例**

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  assert(is_sorted(A, A));
  assert(is_sorted(A, A, greater<int>()));

  assert(!is_sorted(A, A + N));
  assert(!is_sorted(A, A + N, greater<int>()));

  sort(A, A + N);
  assert(is_sorted(A, A + N));
  assert(!is_sorted(A, A + N, greater<int>()));

  sort(A, A + N, greater<int>());
  assert(!is_sorted(A, A + N));
  assert(is_sorted(A, A + N, greater<int>()));
}
```

13.2 sorted ranges 上的操作行为

13.1 节的排序算法之所以重要，原因之一是有很多算法必须输入已排序的 ranges。range 一旦排序，查找特定值、与其他已排序之 ranges 结合等等动作便会简单许多。

13.2.1 二分查找法（Binary Search）

每个曾经用过字典的人都了解排序的重要性：可以轻易在 range 内寻找某物。如果具有 N 个元素的 range 未经排序，那么元素找寻动作是一种线性操作。如同 find（p.199），你可能得审阅该 range 中的每个元素。如果这个 range 已经排序，那么其元素找寻动作就只是对数（logarithmic）操作。二分查找法（binary search）」的运作方式就是一再地将一个已序 range 分为两半。

STL 包含四种不同的二分查找算法，因为执行二分查找法时你可能想要解决不同的问题。其中最基本（但事实上不是最有用）的问题是：某元素是否包含于某个 range 之中。这个问题对应到 binary_search 算法。

但是通常解决这个问题还不够。如果该元素已确定包含于某 range 之中，你可能希望知道它的位置。如果它不在该 range 中，你可能想要知道如果它在其中的话，它应该在哪里。

这便是 lower_bound、upper_bound、equal_range 三个算法的一般基本。之所以有三个（而非一个）算法，原因是你要找寻的元素在某个 range 中可能不只存在一份。如果你要找寻数字 17，而数

字列中有四个 17，算法该给你哪一个呢？第一个？最后一个？或给你整个 17 所形成的 range？这三者分别相应于 lower_bound、upper_bound、equal_range。大部分情况下，lower_bound 似乎最方便。

13.2.1.1 binary_search

```
(1) template <class ForwardIterator, class StrictWeaklyComparable>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                       const StrictWeaklyComparable& value);

(2) template <class ForwardIterator, class T, class StrictWeakOrdering>
    bool binary_search(ForwardIterator first, ForwardIterator last,
                       const T& value,
                       StrictWeakOrdering comp);
```

算法 binary_search 是二分查找法的一个版本。它试图在已排序的[first,last) 中寻找元素 value。如果 [first,last) 内有等价于 value 的元素，它会返回 true，否则返回 false。

返回单纯的 bool 或许不能满足你。13.2.1 节的其他算法，包括 lower_bound、upper_bound、equal_range，能够提供额外的信息。

binary_search 的第一版本采用 operator< 进行比较，第二版本则采用 function object comp。

正式地说，当且仅当（if and only if）[first,last) 中存在一个 iterator i 使得「*i < value 和 value < *i 皆不为真」，则第一版本返回 true。当且仅当在 [first,last) 中存在一个 iterator i 使得「comp(*i, value) 和 comp(value, *i) 皆不为真」，则第二版本返回 true。

➢ 何处定义

HP STL 将 binary_search 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeaklyComparable 为 Strict Weakly Comparable 之 model。
- ForwardIterator 的 value type 与 StrickWeaklyComparable 隶属相同型别。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。

- ForwardIterator 的 value type 与 T 相同。
- ForwardIterator 的 value type 可转换为 StrickWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。
- [first,last) 系根据 operator< 递增排列。也就是说 is_sorted(first,last) 为真。

第二个版本：

- [first,last) 是有效的 range。
- [first,last) 系根据 function object comp 递增排列。也就是说 is_sorted(first, last, comp) 为真。

➢ 复杂度

比较次数是对数等级：至多 log(last-first)+2 次。如果 ForwardIterator 隶属于 Random Access Iterator，则遍历该 range 的次数也是对数等级；否则遍历的次数与 last-fisrt 成比例。

针对 Random Access Iterators 和其他的 iterators，运行期复杂度有所不同，因为 advance（p.183）之于 Random Access Iterators 为常量时间，之于 Forward Iterators 为线性时间。

➢ 样例

试在已排序的整数数组中查找元素。

```
int main() {
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    cout << "Searching for " << i << ": "
         << (binary_search(A, A + N, i) ? "present" : "not present")
         << endl;
  }
}
```

输出：

```
Searching for 1: present
Searching for 2: present
Searching for 3: present
Searching for 4: not present
```

```
Searching for 5: present
Searching for 6: not present
Searching for 7: not present
Searching for 8: present
Searching for 9: not present
Searching for 10: not present
```

13.2.1.2 lower_bound

```
(1) template <class ForwardIterator, class StrictWeaklyComparable>
    ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const StrictWeaklyComparable& value);

(2) template <class ForwardIterator, class T, class StrictWeakOrdering>
    ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, StrictWeakOrdering comp);
```

算法 lower_bound 是二分查找法的一种版本。它试图在已排序的[first,last) 中寻找元素 value。如果 [first,last) 具有等价于 value 的元素，lower_bound 返回一个 iterator 指向其中第一个元素。如果没有这样的元素存在，它便返回「假设这样的元素存在的话，会出现的位置」。也就是说它会返回一个 iterator，指向第一个「不小于 value」的元素。如果 value 大于 [first,last) 的任何一个元素，则返回 last。以稍许不同的观点来看 lower_bound，其返回值是「在不破坏顺序的原则下，第一个可安插 value 的位置」。

第一版本采用 operator< 进行比较，第二版本采用 function object comp。

更正式地说，第一版本返回 [first,last) 中最远的 iterator i，使得[first,i) 中的每个 iterator j 都满足「*j < value」。同样地，第二版本返回 [first,last) 中最远的 iterator i，使得[first,i) 中的每个 iterator j 都满足「comp(*j, value) 为真」。

➢ 何处定义

HP STL 将 lower_bound 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeaklyComparable 为 Strict Weakly Comparable 之 model。
- ForwardIterator 的 value type 与 StrickWeaklyComparable 隶属相同型别。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- ForwardIterator 的 value type 与 T 相同。
- ForwardIterator 的 value type 可转换为 StrickWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。
- [first,last) 已根据 operator< 递增排列。也就是说 is_sorted(first, last) 为 true。

第二个版本：

- [first,last) 是有效的 range。
- [first,last) 已根据 function object comp 递增排列。也就是说 is_sorted(first, last, comp) 为 true。

➢ 复杂度

比较动作的执行次数呈对数等级：至多 log(last-first)+1 次。如果 ForwardIterator 隶属于 Random Access Iterator，那么 range 的遍历次数也是对数等级，否则其次数与 last-fisrt 成比例。

面对 Random Access Iterator 和其他类型之 iterators，运行期复杂度有所不同，因为 advance（p.183）对 Random Access Iterators 而言为常量时间，对 Forward Iterators 则为线性时间。

➢ 样例

```
int main() {
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    int* p = lower_bound(A, A + N, i);
    cout << "Searching for " << i << ".  ";
    cout << "Result: index = " << p - A << ", ";
    if (p != A + N)
       cout << "A[" << p - A << "] == " << *p << endl;
```

```
    else
        cout << "which is off-the-end." << endl;
  }
}
```

输出为：

```
Searching for 1.  Result: index = 0, A[0] == 1
Searching for 2.  Result: index = 1, A[1] == 2
Searching for 3.  Result: index = 2, A[2] == 3
Searching for 4.  Result: index = 5, A[5] == 5
Searching for 5.  Result: index = 5, A[5] == 5
Searching for 6.  Result: index = 6, A[6] == 8
Searching for 7.  Result: index = 6, A[6] == 8
Searching for 8.  Result: index = 6, A[6] == 8
Searching for 9.  Result: index = 7, which is off-the-end.
Searching for 10.  Result: index = 7, which is off-the-end.
```

算法 lower_bound 很类似 C 函数库的 bsearch 函数。其中最主要的差别在于，当你查找某值却找不到时，lower_bound 返回「如果该值存在，会出现在哪个位置」，而 bsearch 返回 null 指针，表示找不到。

如果你所要查找的值 val 没有出现，lower_bound 会返回 last，或是某个 iterator i 致使 val < *i。这表示根据 lower_bound 撰写一个与 bsearch 类似接口的外覆函数（wrapper）是很简单的事。

```
template <class ForwardIterator, class StrictWeaklyComparable>
ForwardIterator stl_bsearch(ForwardIterator first, ForwardIterator last,
                            const StrictWeaklyComparable& value) {
  ForwardIterator loc = lower_bound(first, last, value);
  return loc == last || value < *loc ? last : loc;
}

template <class ForwardIterator, class T, class StrictWeakOrdering>
ForwardIterator stl_bsearch(ForwardIterator first, ForwardIterator last,
                            const T& value,
                            StrictWeakOrdering& comp) {
  ForwardIterator loc = lower_bound(first, last, value, comp);
  return loc == last || comp(value, *loc) ? last : loc;
}
```

13.2.1.3 upper_bound

```
(1) template <class ForwardIterator, class StrictWeaklyComparable>
    ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const StrictWeaklyComparable& value);
```

```
(2) template <class ForwardIterator, class T, class StrictWeakOrdering>
    ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, StrictWeakOrdering comp);
```

算法 upper_bound 是二分查找法的一个版本。它试图在已排序的[first,last) 中寻找 value。更明确地说，它会返回「在不破坏顺序的情况下，可安插 value 的最后一个合适位置」。

由于 ranges 的开头和尾端并不对称（[first,last) 包含 first 但不包含 last），所以 upper_bound 与 lower_bound 的返回值意义大有不同。如果你查找某值而它的确出现在 range 之中，lower_bound 返回的是指向该元素的 iterator。相较之下 upper_bound 并不这么做。因为 upper_bound 所返回的是在不破坏顺序的情况下 value 可被安插的「最后一个」合适位置。如果 value 存在，那么它返回的 iterator 将指向 value 的下一位置，而非 value 自身。

第一版本采用 operator< 进行比较，第二版本采用 function object comp。

更正式地说，第一版本返回 [first,last) 中最远的 iterator i，使得 [first,i) 中的每个 iterator j 都满足「value < *j 不为真」。类似情况，第二版本返回 [first,last) 中最远的 iterator i，使得 [first,i) 中的每个 iterator j 都满足「comp(value, *j) 不为真」。

➢ 何处定义

HP STL 将 upper_bound 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeaklyComparable 为 Strict Weakly Comparable 之 model。
- ForwardIterator 的 value type 和 StrickWeaklyComparable 隶属相同型别。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- ForwardIterator 的 value type 与 T 相同。
- ForwardIterator 的 value type 可转换为 StrickWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。
- [first,last) 已根据 operator< 递增排列。也就是说 is_sorted(first, last) 为 true。

第二个版本：

- [first,last) 是有效的 range。
- [first,last) 已根据 function object comp 递增排列。也就是说 is_sorted(first,last,comp) 为 true。

➢ 复杂度

比较动作的执行次数呈对数等级，至多 log(last-first)+1 次。如果 ForwardIterator 属于 Random Access Iterator，则遍历该 range 的次数也是对数等级，否则遍历次数与 last-fisrt 成比例。

面对 Random Access Iterator 和其他类型之 iterators，运行期复杂度有所不同，因为 advance（p.183）对 Random Access Iterators 而言为常量时间，对 Forward Iterators 则为线性时间。

➢ 样例

```
int main() {
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    int* p = upper_bound(A, A + N, i);
    cout << "Searching for " << i << ".  ";
    cout << "Result: index = " << p - A << ", ";
    if (p != A + N)
      cout << "A[" << p - A << "] == " << *p << endl;
    else
      cout << "which is off-the-end." << endl;
  }
}
```

输出为：

```
Searching for 1.  Result: index = 1, A[1] == 2
Searching for 2.  Result: index = 2, A[2] == 3
Searching for 3.  Result: index = 5, A[5] == 5
Searching for 4.  Result: index = 5, A[5] == 5
Searching for 5.  Result: index = 6, A[6] == 8
Searching for 6.  Result: index = 6, A[6] == 8
```

```
Searching for 7.  Result: index = 6, A[6] == 8
Searching for 8.  Result: index = 7, which is off-the-end.
Searching for 9.  Result: index = 7, which is off-the-end.
Searching for 10.  Result: index = 7, which is off-the-end.
```

13.2.1.4 equal_range

```
(1) template <class ForwardIterator, class StrictWeaklyComparable>
    pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const StrictWeaklyComparable& value);

(2) template <class ForwardIterator, class T, class StrictWeakOrdering>
    pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value,
                StrictWeakOrdering comp);
```

算法 equal_range 是二分查找法的一个版本。它试图在已排序的 [first,last)中寻找 value。

equal_range 的返回值本质上结合了 lower_bound 和 upper_bound 两者的返回值。其返回值是一对 iterators i 和 j，其中 i 是在不破坏顺序的前提下 value 可安插的第一个位置，j 则是在不破坏顺序的前提下 value 可安插的最后一个位置。

这也是可推演出：[i,j) 中的每个元素都等价于（equivalent）value，而且 [i,j) 是 [first,last) 之中符合上述性质的一个最大子区间。

如果以稍许不同的角度来思考 equal_range，我们可把它想成是[first,last) 内「与 value 等价」之所有元素所形成的 range（由于 [first,last) 已排序，所以我们知道「与 value 等价」之所有元素一定都相邻）。算法 lower_bound 返回该 range 的第一个 iterator，算法 upper_bound 返回该 range 的 past-the-end iterator，算法 equal_range 则是以 pair 的形式将两者都返回。

即使 [first,last) 并未含有「与 value 等价」之任何元素，以上叙述仍然合理。这种情况下「与 value 等价」之所有元素所形成的，其实是个空的 range。在不破坏 range 顺序的前提下只有一个位置可以安插 value，而 equal_range 所返回的 pair，其第一与第二元素（都是 iterators）皆指向该位置。

第一版本采用 operator< 进行比较，第二版本采用 function object comp。

更正式地说，第一版本返回一个由 iterators i, j 构成的 pair，其中 i 为 [first,last) 中最远的 iterator，使得 [first,i) 中的每个 iterator k 都满足 *k < value；j 则是 [first,last) 中最远的 iterator，使得 [first,j) 中的每个 iterator k 都满足「value < *k 不为真」。对于 [i,j) 中的每一个 iterator k，value < *k 和 *k < value 都不为真。

类似情况，第二版本返回一个由 iterators i, j 构成的 pair，其中 i 为 [first,last) 中最远的

iterator，使得 [first,i) 中的每个 iterator k 都满足「comp(*k,value) 为真」；j 则是 [first,last) 中最远的 iterator，使得 [first,j) 中的每个 iterator k 都满足「comp(value, *k) 不为真」。对于 [i,j) 中的每一个 iterator k，comp(*k, value)和 comp(value, *k) 都不为真。

➢ 何处定义

HP STL 将 equal_range 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeaklyComparable 为 Strict Weakly Comparable 之 model。
- ForwardIterator 的 value type 和 StrickWeaklyComparable 隶属相同的型别。

第二个版本：

- ForwardIterator 为 Forward Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- ForwardIterator 的 value type 与 T 相同。
- ForwardIterator 的 value type 可转换为 StrickWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 是有效的 range。
- [first,last) 系根据 operator< 递增排列。也就是说 is_sorted(first, last) 为 true。

第二个版本：

- [first,last) 是有效的 range。
- [first,last) 系根据 function object comp 递增排列。也就是说 is_sorted(first,last,comp) 为 true。

➢ 复杂度

比较动作的执行次数呈对数等级：至多 2log(last-first)+1 次。如果 ForwardIterator 属于 Random Access Iterator，则 range 的遍历次数也呈对数等级，否则遍历次数与 last-fisrt 成比例。

面对 Random Access Iterator 和其他类型之 iterators，运行期复杂度有所不同，因为 advance（p.183）对 Random Access Iterators 而言为常量时间，对 Forward Iterators 则为线性时间。

Generic Programming and the STL

➢ **样例**

```
int main() {
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 2; i <= 5; ++i) {
    pair<int*, int*> result = equal_range(A, A + N, i);

    cout << endl;
    cout << "Searching for " << i << endl;

    cout << "  First position where " << i << " could be inserted: "
         << result.first - A << endl;
    cout << "  Last position where " << i << " could be inserted: "
         << result.second - A << endl;

    if (result.first < A + N)
      cout << "  *result.first = " << *result.first << endl;
    if (result.second < A + N)
      cout << "  *result.second = " << *result.second << endl;
  }
}
```

输出为:

```
Searching for 2
  First position where 2 could be inserted: 1
  Last position where 2 could be inserted: 2
  *result.first = 2
  *result.second = 3

Searching for 3
  First position where 3 could be inserted: 2
  Last position where 3 could be inserted: 5
  *result.first = 3
  *result.second = 5

Searching for 4
  First position where 4 could be inserted: 5
  Last position where 4 could be inserted: 5
  *result.first = 5
  *result.second = 5

Searching for 5
  First position where 5 could be inserted: 5
  Last position where 5 could be inserted: 6
  *result.first = 5
  *result.second = 8
```

13.2.2 合并（Merging）两个 Sorted Ranges

13.2.2.1 merge

```
(1) template <class InputIterator1, class InputIterator2,
              class OutputIterator>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

(2) template <class InputIterator1, class InputIterator2,
              class OutputIterator,
              class StrictWeakOrdering>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, StrictWeakOrdering comp);
```

算法 merge 可将两个已排序的 ranges 合并成单一一个已排序的 range。它会从 [first1,last1) 和 [first2,last2) 将元素复制到[result,result+(last1-first1)+(last2-first2))，使其结果以递增顺序排序。返回值为 result+(last1-first1)+(last2-first2)。

这个动作属于稳定（*stable*）行为，意思是能够保持 input ranges 中每个元素的相对顺序。面对 input ranges 中等价的元素，从第一 range 来的元素会排在第二 range 来的元素之前。

merge 有两个版本，其差别在于如何比较元素。第一版本使用 operator< 进行比较，第二版本使用 function object comp。

请注意，merge 和 set_union（p.324）十分类似，两个算法都可以由两个已排序之 ranges 所含元素中构造出一个已排序的 range。差别在于「两个 input ranges 含有相同的值」时的处理方式：set_union 会剔除重复元素，而 merge 不会。因此，merge 导致以下必然结果：output range 的长度等于两个 input ranges 的长度总和。相较之下 set_union 只保证 output range 的长度小于或等于 input ranges 的长度总和。

➢ 何处定义

HP STL 将 merge 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。

- InputIterator1 的 value type 与 InputIterator2 的 value type 相同。
- InputIterator1 的 value type 为 Strick Weakly Comparable。
- InputIterator1 value types 可转换为 OutputIterator value types 所形成之集合内的某个型别。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- InputIterator1 的 value type 与 InputIterator2 的 value type 相同。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- InputIterator1 的 value type 可转换为 StrickWeakOrdering 的引数型别。
- InputIterator1 value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

第一个版本：

- [first1,last1) 为有效的 range。
- [first1,last1) 已以 operator< 做递增排序。也就是说 is_sorted(first1, last1) 为 true。
- [first2,last2) 为有效的 range。
- [first2,last2) 已以 operator< 做递增排序。也就是说 is_sorted(first2, last2) 为 true。
- 有足够空间容纳所有被复制的元素。更正式地说 [result, result+(last1-first1)+(last2-first2)) 为有效的 range。
- [first1,last1) 和 [result, result+(last1-first1)+(last2-first2)) 不能重叠。
- [first2,last2) 和 [result, result+(last1-first1)+(last2-first2)) 不能重叠。

第二个版本：

- [first1,last1) 为有效的 range。
- [first1,last1) 已以 function object comp 做递增排序。也就是说 is_sorted(first1, last1, comp) 为 true。
- [first2,last2) 为有效的 range。
- [first2,last2) 已以 function object comp 做递增排序。也就是说 is_sorted(first2, last2, comp) 为 true。

- 有足够空间容纳所有被复制的元素。更正式地说，[result, result+(last1-first1)+(last2-first2)) 为有效的 range。
- [first1,last1) 和 [result, result+(last1-first1)+(last2-first2)) 不能重叠。
- [first2,last2) 和 [result, result+(last1-first1)+(last2-first2)) 不能重叠。

➢ **复杂度**

线性。对于非空的 range，至多执行 (last1-first1)+(last2-first2)- 1 次比较。

➢ **样例**

```
int main()
{
  int A1[] = { 1, 3, 5, 7 };
  int A2[] = { 2, 4, 6, 8 };
  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);

  merge(A1, A1 + N1, A2, A2 + N2,
       ostream_iterator<int>(cout, " "));
  cout << endl;
  // 输出为 "1 2 3 4 5 6 7 8"
}
```

13.2.2.2 inplace_merge

```
(1) template <class BidirectionalIterator>
    inline void inplace_merge(BidirectionalIterator first,
                              BidirectionalIterator middle,
                              BidirectionalIterator last);

(2) template <class BidirectionalIterator, class StrictWeakOrdering>
    inline void inplace_merge(BidirectionalIterator first,
                              BidirectionalIterator middle,
                              BidirectionalIterator last,
                              StrictWeakOrdering comp);
```

如果两个连接一起的 ranges [first,middle) 和 [middle,last) 都已排序，那么 inplace_merge 可将它们结合成单一已序 range。也就是说其结果由两段递增排序的 ranges 构成，重新排列后使整个 range 以递增排序。

和 merge 一样，inplace_merge 也是稳定（*stable*）操作。每个 input ranges 中的元素相对顺序不会变动，而两 input ranges 内如果有等价元素，从第一 range 来的元素会被排在从第二 range 来的元素之前。

inplace_merge 有两个版本，其差别在于如何定义某元素小于另一个元素。第一版本使用 operator< 进行比较，第二版本使用 function object comp。

➢ 何处定义

HP STL 将 inplace_merge 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- BidirectionalIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- BidirectionalIterator 为 Bidirectional Iterator 之 model。
- BidirectionalIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- BidirectionalIterator 的 value type 可转换为 StrictWeakOrdering 的引数型别。

➢ 先决条件

第一个版本：

- [first,middle) 为有效的 range。
- [middle,last) 为有效的 range。
- [first,middle) 以 operator< 递增排序。也就是说 is_sorted(first, middle) 为 true。
- [middle,last) 系以 operator< 递增排序。也就是说 is_sorted(middle, last) 为 true。

第二个版本：

- [first,middle) 为有效的 range。
- [middle,last) 为有效的 range。
- [first,middle) 以 function object comp 递增排序。也就是说 is_sorted(first,middle,comp) 为真。
- [middle,last) 以 function object comp 递增排序。也就是说 is_sorted(middle,last,comp) 为真。

➢ 复杂度

算法 inplace_merge 是一种 ***adaptive*** 算法。它会试图分配临时的内存缓冲区，而且它的运行期复杂度取决于有多少可用的内存。对于非空的 ranges，在最坏的情况下（若没有辅助的内存可用的话），其复杂度为 ***O(N log N)***，其中 N 为 last-first，而在最好的情况下（若有足够大的辅助内存可用），最多需要 N-1 次比较动作。

➢ 样例

试着将两个相邻且已排序的整数 ranges 合并为一个新的 range。

```
int main()
{
  int A[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

  inplace_merge(A, A + 4, A + 8);
  copy(A, A + 8, ostream_iterator<int>(cout, " "));
  // 输出为 "1 2 3 4 5 6 7 8"
}
```

由于「将两个连续且已排序的 ranges 合并」的效率相当好，所以我们可以利用「先分剖再征服（divide and conquer）」的方法来对一个 range 排序：先将 range 对半分开，分别对各半部排序，然后再利用 inplace_merge 构造出一个已排序的 range。

```
template <class BidirectionalIter>
void mergesort(BidirectionalIter first, BidirectionalIter last) {
  typename iterator_traits<BidirectionalIter>::difference_type n
    = distance(first, last);
  if (n == 0 || n == 1)
    return;
  else {
    BidirectionalIter mid = first + n / 2;
    mergesort(first, mid);
    mergesort(mid, last);
    inplace_merge(first, mid, last);
  }
}
```

这就是所谓的 ***merge sort*** 算法。事实上，stable_sort（p.294）正是以这种方式实现出来（搭配某些小小的修改以增加效率）。

13.2.3 在 Sorted Ranges 身上执行集合（Set）相关操作

集合（set）是一种基本数学概念，是未经排序的元素的聚集。集合理论的基本运算包括联集（union）、交集（intersection）、差集（difference）。

如果集合无序可言，本章中的集合运算对于已排序的（sorted）range 会怎么处理？答案取决于纯抽象数学中之集合与计算机程序所表示之集合的差别。在抽象数学中，集合无序可言，例如 {1,2,3} 和 {3,2,1} 是相同的集合。然而在计算机程序中，我们必须选择某种方式，将集合表示为一种数据结构，这种表示法可让联集和交集的运算更为简单。

「sorted range」可成为集合的一个恰当表示法。它可让集合理论中的所有基本运算在 sorted ranges 身上更有效率地执行，而且那些运算所产生的 ranges 也自然而然地排好了序。本节涵盖的算法将集合运算行为实现成 sorted range 上的操作行为。

除了这些算法，STL 还包含一种 container class，称作 set（p.461）。如果你希望在本节执行各种集合算法，set 是一种十分合宜的元素聚集方式。set 内的元素独一无二，并且自动形成一个 sorted range。

由于本节算法所操作的集合是以 sorted ranges 表示，而非以抽象数学定义表示，所以它们以一个重要的方式将数学上的集合运算行为一般化：某特定值可以不只一次地出现于 sorted range 中。由于不再有「每个元素在每个 input range 中至多出现一次」的先决条件，这些算法于是成功地将数学上的集合运算行为一般化，使这些运算行为在面对具有重复数值之 input ranges 时仍有明确的定义。这些一般化后的运算行为仍然满足一般的集合运算，而且，当 input ranges 不含重复数值，这些算法的行为正是根据数学而定义。

13.2.3.1 includes

```
(1) template <class InputIterator1, class InputIterator2>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2);

(2) template <class InputIterator1, class InputIterator2,
              class StrictWeakOrdering>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  StrictWeakOrdering comp);
```

算法 includes 可测试某个集合是否为另一集合的子集合，此处两个集合都以 sorted ranges 来表示。也就是说，当且仅当 [first2,last2) 中的每个元素在 [first1,last2) 中存在等价元素，那么 includes 便返回 true。

和 STL 的所有集合算法一样，includes 将集合运算的数学定义一般化。使 [first1,last1) 或 [first2,last2) 中的每个元素不必独一无二。这种情况下，所谓「含有一个子集合」的一般化定义是：[first2,last2) 中的每个元素，其相应的等价元素必须也出现于 [first1,last1) 中。换句话说假设某个元素在 [first1,last1) 出现 m 次，在 [first2,last2) 出现 n 次，那么如果 m < n，includes 会返回 false。

includes 有两个版本，差别在于如何定义某个元素小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 include 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- InputIterator1 的 value type 为 Strick Weakly Comparable 之 model。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- InputIterator1 的 value type 可转换为 StrickWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。
- [first1,last1) 以 operator< 递增排序。也就是说 is_sorted(first1, last1) 为 true。
- [first2,last2) 以 operator< 递增排序。也就是说 is_sorted(first2, last2) 为 true。

第二个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。

- [first1,last1) 以 function object comp 递增排序。也就是说 is_sorted(first1, last1, comp) 为 true。
- [first2,last2) 以 function object comp 递增排序。也就是说 is_sorted(first2, last2, comp) 为 true。

➢ 复杂度

线性。如果 [first1,last1) 或 [first2,last2) 是空的，就没有任何比较动作。否则执行至多 2((last1-first1)+(last2-first2))-1 次比较。

➢ 样例

```
int main()
{
  int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
  int A2[] = { 1, 4, 7 };
  int A3[] = { 2, 7, 9 };
  int A4[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
  int A5[] = { 1, 2, 13, 13 };
  int A6[] = { 1, 1, 3, 21 };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3) / sizeof(int);
  const int N4 = sizeof(A4) / sizeof(int);
  const int N5 = sizeof(A5) / sizeof(int);
  const int N6 = sizeof(A6) / sizeof(int);

  cout << "A2 contained in A1: "
       << (includes(A1, A1 + N1, A2, A2 + N2) ? "true" : "false")
       << endl;
  cout << "A3 contained in A1: "
       << (includes(A1, A1 + N1, A3, A3 + N3) ? "true" : "false")
       << endl;
  cout << "A5 contained in A4: "
       << (includes(A4, A4 + N4, A5, A5 + N5) ? "true" : "false")
       << endl;
  cout << "A6 contained in A4: "
       << (includes(A4, A4 + N4, A6, A6 + N6) ? "true" : "false")
       << endl;
}
```

输出为:

```
A2 contained in A1: true
A3 contained in A1: false
A5 contained in A4: false
A6 contained in A4: true
```

13.2.3.2 set_union

```
(1) template <class InputIterator1, class InputIterator2,
             class OutputIterator>
    OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);

(2) template <class InputIterator1, class InputIterator2,
             class OutputIterator,
             class StrictWeakOrdering>
    OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result,
                             StrictWeakOrdering comp);
```

算法 set_union 可构造出两集合之联集。也就是说它能构造出集合 S1∪S2，此集合内含 S1 或 S2 内的每一个元素。S1、S2 及其联集都是以 sorted ranges 表示。返回值为 output range 的尾端。

如同 STL 的所有集合算法，set_union 将集合运算的数学定义给一般化了：[first1,last1) 或 [first2,last2) 内的每个元素并不需要唯一。在此情况下，联集的定义也被一般化了：如果某个值在 [first1,last1) 出现 n 次，同样的值在 [first2,last2) 出现 m 次，那么该值在 output range 中会出现 max(m, n) 次。（联集的惯常定义是在 n <= 1, m <= 1 的情况下进行。）联集属于稳定（stable）操作，意思是 input ranges 中每个元素的相对顺序都不会被改变，而且如果是两个 input ranges 皆出现的元素，一定从第一个（而非第二个）input range 复制过来。

当我说「某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次」，这种表示法其实不够精确。更精确的说法是：[first1,last1) 内含一个「拥有 n 个彼此等价之元素」所形成的 range，[first2,last2) 内含一个「拥有 m 个彼此等价之元素」所形成的 range。output range 则内含 max(n,m) 个彼此等价的元素。output range 之中有 n 个元素是从 [first1,last1) 复制而来，其他 max(m-n, 0) 个元素从 [first2,last2) 复制而来。

set_union 有两个版本，差别在于如何定义某个元素是否小于另一个元素。第一版本使用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 set_union 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `InputIterator1` 与 `InputIterator2` 具有相同的 value type。
- `InputIterator1` 的 value type 为 Strick Weakly Comparable 之 model。
- `InputIterator1` value type 可转换为 `OutputIterator` value types 所形成集合内的某个型别。

第二个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `StrictWeakOrdering` 为 Strict Weak Ordering 之 model。
- `InputIterator1` 与 `InputIterator2` 具有相同的 value type。
- `InputIterator1` 的 value type 可转换为 `StrickWeakOrdering` 之引数型别。
- `InputIterator1` value type 可转换为 `OutputIterator` value types 所形成集合内的某个型别。

➢ 先决条件

第一个版本：

- `[first1,last1)` 为有效的 range。
- `[first2,last2)` 为有效的 range。
- `[first1,last1)` 以 `operator<` 递增排序。也就是说 `is_sorted(first1, last1)` 为真。
- `[first2,last2)` 以 `operator<` 递增排序。也就是说 `is_sorted(first2, last2)` 为真。
- 有足够空间容纳所有被复制的元素。更正式地说，`[result,result+n)` 是有效的 range，其中 `n` 是两个 input ranges 之联集的元素个数。
- `[first1,last1)` 与 `[result,result+n)` 不重叠。
- `[first2,last2)` 与 `[result,result+n)` 不重叠。

第二个版本：

- `[first1,last1)` 为有效的 range。
- `[first2,last2)` 为有效的 range。

- [first1,last1) 以 function object comp 递增排序。也就是说 is_sorted(first1,last1,comp) 为真。
- [first2,last2) 以 function object comp 递增排序。也就是说 is_sorted(first2,last2,comp) 为真。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，其中 n 是两个 input ranges 之联集的元素个数。
- [first1,last1) 与 [result,result+n) 不重叠。
- [first2,last2) 与 [result,result+n) 不重叠。

➢ 复杂度

线性。对于非空的 ranges，至多执行 2((last1-first1)+(last2-first2))-1 次比较。

➢ 样例

```
inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'B', 'B', 'f', 'H'};
  char A4[] = {'A', 'B', 'b', 'C', 'D', 'F', 'F', 'h', 'h'};

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Union of A1 and A2: ";
  set_union(A1, A1 + N1, A2, A2 + N2,
          ostream_iterator<int>(cout, " "));
  cout << endl
       << "Union of A3 and A4: ";
  set_union(A3, A3 + N3, A4, A4 + N4,
          ostream_iterator<char>(cout, " "),
          lt_nocase);
  cout << endl;
}
```

输出为：

```
Union of A1 and A2: 1 1 2 3 5 7 8 9 11 13
Union of A3 and A4: a b B B C D f F H h
```

13.2.3.3 set_intersection

```
(1) template <class InputIterator1, class InputIterator2,
             class OutputIterator>
    OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

(2) template <class InputIterator1, class InputIterator2,
             class OutputIterator,
             class StrictWeakOrdering>
    OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result,
                     StrictWeakOrdering comp);
```

算法 set_intersection 可以构造出两集合的交集。也就是说，它能构造出集合 S1 ∩ S2，包含每一个在 S1 且在 S2 中出现的元素。S1、S2 及其交集都以 sorted ranges 表示。返回值为 output range 的尾端。

如同 STL 的所有集合算法，set_intersection 将集合运算的数学定义一般化了：[first1,last1) 或 [first2,last2) 内的元素并不一定得独一无二。交集的定义也因此被一般化了：如果某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次，那么该值会在 output range 中出现 min(m,n) 次。（交集的惯常定义是在 n <= 1, m <= 1 的情况下进行。）交集属于稳定（*stable*）操作，意思是元素都从第一个 ranges 复制过来，而且 output range 内的元素相对顺序一定和第一个 input range 的相对顺序相同。

当我说「某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次」，这种表示法其实不够精确。更精确的说法是：[first1,last1) 内含一个「拥有 n 个彼此等价之元素」所形成的 range，[first2,last2) 内含一个「拥有 m 个彼此等价之元素」所形成的 range。output range 则内含 min(n,m) 个彼此等价的元素。所有元素都复制于 [first1,last1)。

set_intersection 有两个版本，差别在于如何定义某个元素是否小于另一个元素。第一版本使用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 set_intersection 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `InputIterator1` 与 `InputIterator2` 具有相同的 value type。
- `InputIterator1` 的 value type 为 Strick Weakly Comparable 之 model。
- `InputIterator1` value type 可转换为 `OutputIterator` value types 所形成之集合内的某型别。

第二个版本：

- `InputIterator1` 为 Input Iterator 之 model。
- `InputIterator2` 为 Input Iterator 之 model。
- `OutputIterator` 为 Output Iterator 之 model。
- `StrictWeakOrdering` 为 Strict Weak Ordering 之 model。
- `InputIterator1` 与 `InputIterator2` 具有相同的 value type。
- `InputIterator1` 的 value type 可转换为 `StrickWeakOrdering` 之引数型别。
- `InputIterator1` value type 可转换为 `OutputIterator` value types 所形成之集合内的某型别。

➢ 先决条件

第一个版本：

- `[first1,last1)` 为有效的 range。
- `[first2,last2)` 为有效的 range。
- `[first1,last1)` 以 `operator<` 递增排序。也就是说 `is_sorted(first1, last1)` 为 `true`。
- `[first2,last2)` 以 `operator<` 递增排序。也就是说 `is_sorted(first2, last2)` 为 `true`。
- 有足够空间容纳所有被复制的元素。更正式地说，`[result,result+n)` 是有效的 range，其中 `n` 为两 input ranges 之交集的元素个数。
- `[first1,last1)` 与 `[result,result+n)` 不重叠。
- `[first2,last2)` 与 `[result,result+n)` 不重叠。

第二个版本：

- `[first1,last1)` 为有效的 range。
- `[first2,last2)` 为有效的 range。

- [first1,last1) 以 function object comp 递增排序。也就是说，is_sorted(first1, last1, comp) 为真。
- [first2,last2) 以 function object comp 递增排序。也就是说，is_sorted(first2, last2, comp) 为真。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，其中 n 是两个 input ranges 之交集的元素个数。
- [first1,last1) 与 [result,result+n) 不重叠。
- [first2,last2) 与 [result,result+n) 不重叠。

➢ 复杂度

线性。如果 [first1,last1) 或 [first2,last2) 为空，就没有任何比较动作，否则最多执行 2((last1-first1)+(last2-first2))-1 次比较。

➢ 样例

```
inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Intersection of A1 and A2: ";
  set_intersection(A1, A1 + N1, A2, A2 + N2,
                   ostream_iterator<int>(cout, " "));
  cout << endl
       << "Intersection of A3 and A4: ";
  set_intersection(A3, A3 + N3, A4, A4 + N4,
                   ostream_iterator<char>(cout, " "),
                   lt_nocase);
  cout << endl;
}
```

输出为：

```
Intersection of A1 and A2: 1 3 5
Intersection of A3 and A4: a b b f h
```

13.2.3.4 set_difference

```
(1) template <class InputIterator1, class InputIterator2,
              class OutputIterator>
    OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);

(2) template <class InputIterator1, class InputIterator2,
              class OutputIterator,
              class StrictWeakOrdering>
    OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result,
                   StrictWeakOrdering comp);
```

算法 set_difference 可构造出两集合之差。也就是说，它能构造出集合 S1 — S2，其中包含「出现于 S1 但不出现于 S2 内」的每个元素。S1、S2 及其差集都以 sorted ranges 表示。返回值为 output range 的尾端。

如同 STL 的所有集合算法（set algorithms），set_difference 将集合运算的数学定义一般化了：[first1,last1) 或 [first2,last2) 内的每一个元素不需要独一无二。差集的定义也因此一般化了：如果某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次，那么该值在 output range 中会出现 max(n-m, 0) 次。此运算属于稳定（*stable*）操作，意味元素都复制自第一个 range，而非第二个 range，而且 output range 的元素相对位置与第一个 input range 一样。（惯常的集合理论中，差集运算的定义是在 n <= 1, m <= 1 的情况下进行。）

当我说「某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次」，这种表示法其实不够精确。更精确的说法是：[first1,last1) 内含一个「拥有 n 个彼此等价之元素」所形成的 range，[first2,last2) 内含一个「拥有 m 个彼此等价之元素」所形成的 range。output range 则内含 max(n-m, 0) 个彼此等价的元素。所有元素都复制于 [first1,last1)。

set_difference 有两个版本，差别在于如何定义某个元素小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 set_difference 声明于头文件 <algo.h>。C++ standard 将它声明于 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- InputIterator1 的 value type 为 Strick Weakly Comparable 之 model。
- InputIterator1 value type 可转换为 OutputIterator value types 所形成之集合内的某型别。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- InputIterator1 的 value type 可转换为 StrickWeakOrdering 之引数型别。
- InputIterator1 value type 可转换为 OutputIterator value types 所形成之集合内的某型别。

➢ 先决条件

第一个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。
- [first1,last1) 以 operator< 递增排序。也就是说 is_sorted(first1, last1) 为真。
- [first2,last2) 以 operator< 递增排序。也就是说 is_sorted(first2, last2) 为真。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，其中 n 是两个 input ranges 之差集的元素个数。
- [first1,last1) 与 [result,result + n) 不重叠。
- [first2,last2) 与 [result,result + n) 不重叠。

第二个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。
- [first1,last1) 以 function object comp 递增排序。也就是说 is_sorted(first1,last1,comp) 为 true。
- [first2,last2) 以 function object comp 递增排序。也就是说 is_sorted(first2,last2,comp) 为 true。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，这里的 n 是两个 input ranges 之差集的元素个数。
- [first1,last1) 与 [result,result + n) 不重叠。
- [first2,last2) 与 [result,result + n) 不重叠。

➢ 复杂度

线性。对于非空的 ranges，至多执行 2((last1-first1)+(last2-first2))-1 次比较。

➢ 样例

```
inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'g', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Difference of A1 and A2: ";
  set_difference(A1, A1 + N1, A2, A2 + N2,
                 ostream_iterator<int>(cout, " "));
  cout << endl
       << "Defference of A3 and A4: ";
  set_difference(A3, A3 + N3, A4, A4 + N4,
                 ostream_iterator<char>(cout, " "),
                 lt_nocase);
  cout << endl;
}
```

输出为：

```
Difference of A1 and A2: 7 9 11
Defference of A3 and A4: B B g H
```

13.2.3.5 set_symmetric_difference

```
(1) template <class InputIterator1, class InputIterator2,
              class OutputIterator>
    OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);

(2) template <class InputIterator1, class InputIterator2,
              class OutputIterator,
              class StrictWeakOrdering>
    OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result,
                             StrictWeakOrdering comp);
```

算法 set_symmetric_difference 可构造出两集合之对称差。也就是说它所构造的集合包含「出现于 S1 但不出现于 S2 内」的所有元素以及「出现于 S2 但不出现于 S1 内」的所有元素，数学上我们可以写成 (S1-S2) ∪ (S2-S1)。S1、S2 及其对称差集都以 sorted ranges 表现，返回值为 output range 的尾端。

如同 STL 的所有集合算法，set_symmetric_difference 将集合运算的数学定义一般化了。[first1,last1) 或 [first2,last2) 内的每个元素不必一定得独一无二。集合之对称差因此亦有了一般化的定义：如果某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次，那么该值在 output range 中会出现 |n-m| 次（一般集合理论中，此一运算系在 n <= 1, m <= 1 的情况下进行）。这是一种稳定的（*stable*）运算行为，意味每个 input range 的元素相对位置不会被改变。

当我说「某值在 [first1,last1) 出现 n 次，在 [first2,last2) 出现 m 次」，这种表示法其实不够精确。更精确的说法是：[first1,last1) 内含一个「拥有 n 个彼此等价之元素」所形成的 range，[first2,last2) 内含一个「拥有 m 个彼此等价之元素」所形成的 range。output range 则内含 |n-m| 个彼此等价的元素。如果 n>m，output range 元素中的最后 n-m 个将由 input range [first1,last1) 复制而来，如果 n < m 则 output range 元素中的最后 m-n 个将由 input range [first2,last2) 复制而来。

set_symmetric_difference 有两个版本，差别在于如何定义某个元素小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 set_symmetric_difference 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- InputIterator1 的 value type 为 Strick Weakly Comparable 之 model。
- InputIterator1 value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

第二个版本：

- InputIterator1 为 Input Iterator 之 model。
- InputIterator2 为 Input Iterator 之 model。
- OutputIterator 为 Output Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- InputIterator1 与 InputIterator2 具有相同的 value type。
- InputIterator1 的 value type 可转换为 StrickWeakOrdering 之引数型别。
- InputIterator1 value type 可转换为 OutputIterator value types 所形成之集合内的某个型别。

➢ 先决条件

第一个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。
- [first1,last1) 以 operator< 递增排序。也就是说 is_sorted(first1, last1) 为真。

- [first2,last2) 以 operator< 递增排序。也就是说 is_sorted(first2, last2) 为真。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，其中 n 为两个 input ranges 之对称差的元素个数。
- [first1,last1) 与 [result,result + n) 不重叠。
- [first2,last2) 与 [result,result + n) 不重叠。

第二个版本：

- [first1,last1) 为有效的 range。
- [first2,last2) 为有效的 range。
- [first1,last1) 以 function object comp 递增排序。也就是说 is_sorted(first1,last1,comp) 为 true。
- [first2,last2) 以 function object comp 递增排序。也就是说 is_sorted(first2,last2,comp) 为 true。
- 有足够空间容纳所有被复制的元素。更正式地说，[result,result+n) 是有效的 range，其中 n 为两个 input ranges 之对称差的元素个数。
- [first1,last1) 与 [result,result + n) 不重叠。
- [first2,last2) 与 [result,result + n) 不重叠。

➢ 复杂度

线性。对于非空的 ranges，至多执行 2((last1-first1)+(last2-first2))-1 次比较。

➢ 样例

```
inline bool lt_nocase(char c1, char c2) {
  return tolower(c1) < tolower(c2);
}

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'g', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);
```

```
  cout << "Symmetric difference of A1 and A2: ";
  set_symmetric_difference(A1, A1 + N1, A2, A2 + N2,
                           ostream_iterator<int>(cout, " "));
  cout << endl
       << "Symmetric defference of A3 and A4: ";
  set_symmetric_difference(A3, A3 + N3, A4, A4 + N4,
                           ostream_iterator<char>(cout, " "),
                           lt_nocase);
  cout << endl;
}
```

输出为:

```
Symmetric difference of A1 and A2: 1 2 7 8 9 11 13
Symmetric defference of A3 and A4: B B C D F g H
```

13.3 堆的相关操作（Heap Operations）

堆（heaps）并不是 sorted ranges。堆中的元素并不会以递增顺序排列，而是以一种更复杂的方式排列。堆内部事实上是以树状方式来表现一个 sequential range，该树状结构系以「每个节点小于或等于其父节点」的方式构造。

由于三个理由，使得堆和 sorted ranges 有密切关系。第一，和 sorted ranges 一样，堆提供高效率的方式来取得其最大元素。如果 `[first,last)` 是一个堆，那么 `*first` 便是堆中的最大元素。第二，我们可以在对数（logarithmic）时间内以 `push_heap` 为堆增加新元素，或以 `pop_heap` 为堆移除某元素。第三，有一种简单而有效率的算法，称作 `sort_heap`，可将堆转成 sorted range。

堆是一种表现 priority queues 的方便法门，后者的每一个元素都以任意顺序安插，但移除的顺序是由最大到最小。例如 STL 有一个 container adapter `priority_queue` 就是以堆实现出来。

13.3.1 make_heap

```
(1) template <class RandomAccessIterator>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                   StrictWeakOrdering comp);
```

算法 `make_heap` 可将任意的 range `[first,last)` 转换成一个堆。

`make_heap` 有两个版本，差别在于如何定义某个元素小于另一个元素。第一版本采用 `operator<` 进行比较，第二版本采用 function object `comp`。

第一版本必然导致 is_heap(first, last) 为 true，第二版本必然导致 is_heap(first, last, comp) 为 true。

➢ 何处定义

HP STL 将 make_heap 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

- [first,last) 为有效的 range。

➢ 复杂度

线性。最多执行 3(last-first) 次比较。

➢ 样例

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  assert(is_heap(A, A+N));
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  sort_heap(A, A+N);
  assert(is_sorted(A, A+N));
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

13.3.2 push_heap

```
(1) template <class RandomAccessIterator>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                   StrictWeakOrdering comp);
```

算法 push_heap 可以为堆（heap）增加新元素。其接口有点特别：堆以 [first,last-1) 表示，新增的元素为 *(last-1)。

push_heap 有两个版本，差别在于如何定义某个元素是否小于另一个。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

第一版本的必然结果是，is_heap(first,last) 为 true；第二版本的必然结果是，is_heap(first, last, comp) 为 true。

➢ 何处定义

HP STL 将 push_heap 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 为有效的 range。
- [first,last-1) 为有效的 range。也就是说 [first,last) 不为空。
- [first,last-1) 是一个堆。也就是说 is_heap(first, last-1) 为 true。

第二个版本：

- [first,last) 为有效的 range。
- [first,last-1) 为有效的 range。也就是说 [first,last) 不为空。
- [first,last-1) 是一个堆。也就是说 is_heap(first, last-1, comp) 为 true。

➢ 复杂度

对数等级。最多执行 log(last-first) 次比较。

➢ 样例

```
int main()
{
  int A[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  make_heap(A, A + 9);
  cout << "[A, A + 9) = ";
  copy(A, A + 9, ostream_iterator<int>(cout, " "));
  cout << endl;

  push_heap(A, A + 10);
  cout << "[A, A + 10) = ";
  copy(A, A + 10, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

输出为：

```
[A, A + 9) = 8 7 6 3 4 5 2 1 0
[A, A + 10) = 9 8 6 3 7 5 2 1 0 4
```

13.3.3 pop_heap

```
(1) template <class RandomAccessIterator>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                  StrictWeakOrdering comp);
```

算法 pop_heap 可移除堆 [first,last) 内的最大元素（亦即移除 *first）。pop_heap 有两个版本，差别在于如何定义某个元素是否小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

pop_heap 的两个版本都有一个必然结果：「从堆中移除的元素是 *(last-1)」。此外，第一版本必然导致 is_heap(first, last-1) 为 true，第二版本必然导致 is_heap(first, last-1, comp) 为 true。

➢ 何处定义

HP STL 将 pop_heap 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 为有效的 range。
- [first,last-1) 为有效的 range。也就是说 [first,last) 不是空的。
- [first,last) 是一个堆。也就是说 is_heap(first, last) 为真。

第二个版本：

- [first,last) 为有效的 range。
- [first,last-1) 为有效的 range。也就是说 [first,last) 不是空的。
- [first,last) 是一个堆。也就是说 is_heap(first, last, comp) 为真。

➢ 复杂度

对数等级。最多执行 2log(last-first) 次比较。

➢ 样例

移除堆内的最大元素。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  cout << "Before pop: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));

  pop_heap(A, A+N);
  cout << endl << "After pop: ";
  copy(A, A+N-1, ostream_iterator<int>(cout, " "));
  cout << endl << "A[N-1] = " << A[N-1] << endl;
}
```

输出为：

```
Before pop: 6 5 3 4 2 1
After pop: 5 4 3 1 2
A[N-1] = 6
```

当你调用 pop_heap，它可从堆中移除最大元素，然后缩小该堆。这意味如果你持续调用 pop_heap，直到堆之中只剩一个元素，你将会得到一个 sorted range，如同堆原本的内容。

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  int n = N;

  while (n > 1) {
    copy(A, A + N, ostream_iterator<int>(cout, " "));
    cout << endl;
    pop_heap(A, A + n);
    --n;
  }

  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

这正是 sort_heap 的实现法。

13.3.4 sort_heap

```
(1) template <class RandomAccessIterator>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                   StrictWeakOrdering comp);
```

算法 sort_heap 可将堆 [first,last) 转为 sorted range。请注意，它并不是个稳定（*stable*）排序法，也就是说它不一定会保持等价元素的相对顺序。（p.294 有稳定性方面的讨论。）

sort_heap 有两个版本，差别在于如何定义某个元素是否小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

HP STL 将 sort_heap 声明于头文件 <algo.h>。C++ standard 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 是可变的（mutable）。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

第一个版本：

- [first,last) 为有效的 range。
- [first,last) 是一个堆。也就是说 is_heap(first, last) 为 true。

第二个版本：

- [first,last) 为有效的 range。
- [first,last) 是一个堆。也就是说 is_heap(first, last, comp) 为 true。

➢ 复杂度

最多执行 N log N 次比较，其中 N 为 last-first。

➢ 样例

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  sort_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

13.3.5 is_heap

```
(1) template <class RandomAccessIterator>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

(2) template <class RandomAccessIterator, class StrictWeakOrdering>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                 StrictWeakOrdering comp);
```

如果 [first,last) 是一个堆，算法 is_heap 返回 true，否则返回 false。is_heap 有两个版本，差别在于如何定义某个元素是否小于另一个元素。第一版本采用 operator< 进行比较，第二版本采用 function object comp。

➢ 何处定义

is_heap 并未出现于原始的 HP STL 之中，也没有出现于 C++ standard。SGI STL 将它声明于头文件 <algorithm>。

➢ 型别条件

第一个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- RandomAccessIterator 的 value type 为 Strict Weakly Comparable 之 model。

第二个版本：

- RandomAccessIterator 为 Random Access Iterator 之 model。
- StrictWeakOrdering 为 Strict Weak Ordering 之 model。
- RandomAccessIterator 的 value type 可转换为 StrictWeakOrdering 之引数型别。

➢ 先决条件

[first,last) 为有效的 range。

➢ 复杂度

线性。对于非空的 range，最多执行 (last-first)-1 次比较动作。

➢ 样例

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A) / sizeof(int);

  assert(!is_heap(A, A+N));
  make_heap(A, A+N);
  assert(is_heap(A, A+N));
}
```

第 14 章

Iterator Classes
迭代器类

每个 STL container 都拥有嵌套的（nested）iterator classes（这是它之所以被称为 "container" 的部分原因）。此外，STL 亦包含一些独立运作的 iterator classes，其中大部分是 iterator adapters。

C++ standard 规定，所有标准的 iterator classes 都得继承自 base class `iterator`(p.185)，本书遵守 C++ standard 的此一规定。但你应该记住，这纯粹只是实现上的细节。真正重要的是，如果一个 iterator class 希望内含 `iterator_traits`（p.177）所要求之各种嵌套型别，「继承自 iterator」只不过是可行方法之一，而且并不一定是最简单的方法。从 template base classes 身上继承而来的型别声明，有着某种复杂的技术限制；明白提供嵌套的 typedefs 通常才是比较简单的做法。

14.1 Insert Iterators

14.1.1 front_insert_iterator

`front_insert_iterator<FrontInsertionSequence>`

Class `front_insert_iterator` 是具有 Output Iterator 功能的一个 iterator adapter。通过 `front_insert_iterator` 完成的赋值动作，可将 object 安插于 Front Insertion Sequence（p.141）中的第一个元素之前。

注意，Front Insertion Sequence 自身已经具备 iterators，毕竟它是一个 Container，而所有的 Containers 都定义有它们自己的 iterators。那么我们为什么还需要为 Front Insertion Sequence 定义另一种 iterator 呢？

是的，我们必须这么做，因为其行为和 Sequence's iterators 有很大的差别。假设 `Seq` 是个 Front Insertion Sequence，那么 `Seq::iterator` 就具有覆盖（overwrite）语义，而 `front_insert_iterator<Seq>` 具有安插（insertion）语义。如果 `i` 是一个有效的 `Seq::iterator`，指向某序列 `s` 中的某个元素，那么表达式 `*i = t` 会将该元素以 `t` 取代，但不改变 `s` 的元素总个数。如果 `ii` 是一个有效

的 front_insert_iterator<Seq>，那么 *ii = t 就相当于 seq.push_front(t)，会为 S 增加新元素，而非覆盖 S 中的任何既有元素。

➢ 样例

```
int main()
{
  list<int> L;

  L.push_front(3);
  front_insert_iterator<list<int> > ii(L);
  *ii++ = 0;
  *ii++ = 1;
  *ii++ = 2;
  copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
  // 打印结果为 2 1 0 3
}
```

注意，此例中出现于 L 内的元素顺序与新增元素时的顺序相反。每个通过 ii 的赋值动作会将元素新增于 L 开头处。这暗示我们有一个简单方法可以产生出一个与现有 Sequence 反向的崭新 Seqence：将 Sequence 现有的全部元素复制到一个 front_insert_iterator 身上。

```
int main()
{
  vector<int> V(100);
  for (int i = 0; i < 100; ++i)
    V[i] = i;

  list<int> L;
  copy(V.begin(), V.end(), front_inserter(L));
  // L 现在以反向顺序容纳 V 中所有元素
}
```

此例之中，我们并不明显运用 front_insert_interator 的 constructor，而是改用辅助函数 front_inserter。这么做会比较方便，因为它使我们不必精确写出 front_insert_iterator 这个型别。由于本例之中我们将这个 iterator 传给泛型算法，而非将它产生于栈（stack）之上，所以不必明白地提到其型别。

➢ 何处定义

HP STL 将 front_insert_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

FrontInsertionSequence 隶属 Front Insertion Sequence 型别，新值安插于此。

➢ 隶属何种 concept

Output Iterator（p.96）。

➢ 型别条件

- template 参数 `FrontInsertionSequence` 必须是个 Front Insertion Sequence。

➢ Public Base Classes

`iterator<output_iterator_tag, void, void, void, void>`

➢ 成员

`front_insert_iterator` 的某些成员并未定义于 Output Iterator 需求条件中，它们是 `front_insert_iterator` 特有的。这些特别的成员以 ※ 符号标示之。

※ **`front_insert_iterator::front_insert_iterator(FrontInsertionSequence& S)`**

这是 constructor。产生一个 `front_insert_iterator`，可将元素安插于 `S` 的第一个元素之前。

`front_insert_iterator::front_insert_iterator(const front_insert_iterator&)`

这是 copy constructor。（描述于 Assignable。）

`front_insert_iterator::~front_insert_iterator()`

这是 destructor。（描述于 Assignable。）

`front_insert_iterator&`
`front_insert_iterator::operator= (const front_insert_iterator&)`

这是赋值操作符（assignment operator）。（描述于 Assignable。）

`front_insert_iterator& front_insert_iterator::operator* ()`

此式用来实现 Output Iterator 相关表达式 `*i = x`。

`front_insert_iterator&`
`front_insert_iterator::operator= (`
`  const typename FrontInsertionSequence::value_type&)`

此式用来实现 Output Iterator 相关表达式 `*i = x`。

`front_insert_iterator& front_insert_iterator::operator++()`

前置累加（preincrement）。（描述于 Output Iterator。）

`front_insert_iterator front_insert_iterator::operator++(int)`

后置累加（postincrement）。（描述于 Output Iterator。）

`front_insert_iterator::iterator_category`

这是一个 tag type，用来表现 iterator 的类属（category）：`output_iterator_tag`。（描述于 Output Iterator。）

```
※ template <class FrontInsertionSequence>
  front_insert_iterator<FrontInsertionSequence>
  front_inserter(FrontInsertionSequence& S)
```

等价于 front_insert_iterator<FrontInsertionSequence>(S)。这个辅助函数之所以存在，主要是为了便利性：它是一个非成员函数（nonmember function），所以 template 参数可被推导出来，而且无须明白声明型别 front_insert_iterator。

注意，Output Iterator 的定义只要求「*ii = t 需为有效表达式」，并未论及 operator* 或 operator=。针对 operator*，最明显的做法就是让它返回某个 proxy object，后者具备适当的 operator=。而在此情况下，proxy object 就是 front_insert_iterator 自身。这纯粹是实现细节，因此你不应该倚赖这种做法。就像描述于 Output Iterator 需求条件中的一样，你应该总是写成 *ii = t。

14.1.2 back_insert_iterator

back_insert_iterator<BackInsertionSequence>

Class back_insert_iterator 是一种 iterator adapter，具有和 Output Iterator 一样的功能：通过 back_insert_iterator，赋值动作可将 object 安插于 Back Insertion Sequence（p.143）的最后一个元素之后。

Back Insertion Sequence 已经具有 iterators，为什么还需要 back_insert_iterator？其存在原因是：back_insert_iterator 提供安插语义而非覆盖语义。如果 i 是一个有效的 Seq::iterator，指向序列 S 之中的某个元素；表达式 *i = t 会将该元素替换为 t，但不改变 S 的元素总个数。但如果 ii 是个 back_insert_iterator<Seq>，*ii = t 就相当于 seq.push_back(t)，它会在 S 之中新增元素，而非覆盖（overwirte）S 内的任何既有元素。

➢ **样例**

```
int main()
{
  list<int> L;
  L.push_front(3);
  back_insert_iterator<list<int> > ii(L);
  *ii++ = 0;
  *ii++ = 1;
  *ii++ = 2;
  copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
  // 打印结果为 3 0 1 2
}
```

请注意此例与 front_insert_itreator 中的对应例子（p.345）间的差异。本例之中，L 内的元素顺序与新增顺序相同。每个赋值动作会将元素新增于 L 尾端。这意味「产生一个 Sequence，使其成为

另一个 Sequence 之复制品」的方法之一就是「通过 back_insert_iterator 复制一个 range」。

```
int main()
{
  list<int> L;
  for (int i = 0; i < 100; ++i)
    L.push_back(i);

  vector<int> v1(L.begin(), L.end());
  vector<int> v2;
  copy(L.begin(), L.end(), back_inserter(v2));

  assert(v1 == v2);
}
```

上例中的两个版本动作大体上等价，但第一版本可能比较快。是的，采用 vector construct，一次可安插所有元素，采用 copy 则一次只能安插一个元素。如果你的编译器尚未支持 member templates，你就无法使用第一版本。这时候你可以使用 copy 加上 back_insert_iterator，因为它们无须倚赖 member templates。

➢ 何处定义

HP STL 将 back_insert_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

BackInsertionSequence　隶属于 Back Insertion Sequence 型别，新值安插于此中。

➢ 隶属何种 concept

Output Iterator（p.96）。

➢ 型别条件

- template 参数 BackInsertionSequence 必须是一个 Back Insertion Sequence。

➢ Public Base Classes

iterator<output_iterator_tag, void, void, void, void>

➢ 成员

back_insert_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 back_insert_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **back_insert_iterator::back_insert_iterator(BackInsertionSequence& S)**

这是 constructor。产生一个 back_insert_iterator，可将元素安插于 S 的最后一个元素之后。

back_insert_iterator::back_insert_iterator(const back_insert_iterator&)

这是 copy constructor。（描述于 Assignable。）

back_insert_iterator::~back_insert_iterator()

这是 destructor。（描述于 Assignable。）

back_insert_iterator&
back_insert_iterator::operator= (const back_insert_iterator&)

这是赋值操作符（assignment operator）。（描述于 Assignable。）

back_insert_iterator& back_insert_iterator::operator* ()

此式用来实现 Output Iterator 相关表达式 *i = x。

back_insert_iterator&
back_insert_iterator::operator= (
const typename BackInsertionSequence::value_type&)

此式用来实现 Output Iterator 相关表达式 *i = x。

back_insert_iterator& back_insert_iterator::operator++()

前置累加（preincrement）。（描述于 Output Iterator。）

back_insert_iterator back_insert_iterator::operator++(int)

后置累加（postincrement）。（描述于 Output Iterator。）

back_insert_iterator::iterator_category

这是一个 tag type，用来表现 iterator 的类属（category）：output_iterator_tag。（描述于 Output Iterator。）

※ **template <class BackInsertionSequence>**
back_insert_iterator<BackInsertionSequence>
back_inserter(BackInsertionSequence& S)

等价于 back_insert_iterator<BackInsertionSequence>(S)。这个辅助函数之所以存在，主要是为了便利性：它是一个非成员函数（nonmember function），所以 template 参数可被推导出来，而且无须明白声明型别 back_insert_iterator。

注意，Output Iterator 的定义只要求「*ii=t 需为有效表达式」，并未论及 operator* 或 operator=。针对 operator*，最明显的做法就是让它返回某个 proxy object，后者具备适当的 operator=。而在此情况下，proxy object 就是 back_insert_iterator 自身。这纯粹是实现细节，因此你不应该倚赖

这种做法。就像描述于 Output Iterator 需求条件中的一样，你应该总是写成 `*ii = t`。

14.1.3 insert_iterator

`insert_iterator<Container>`

Class `insert_iterator` 是一种 iterator adapter，功能如同 Output Iterator。通过 `insert_iterator`，赋值动作可将 object 安插于 Container 之中。如果 `ii` 是个 `insert_iterator`，那么 `ii` 会持续追踪一个 Container `c` 和一个安插点 `p`。表达式 `*ii = t` 将执行安插动作 `c.insert(p, t)`。

在每个 Container 都已拥有 iterators 的情况下，`insert_iterator` 之所以存在，原因是 `insert_iterator` 提供安插语义而非覆盖语义。如果 `i` 是一个有效的 `C::iterator`，指向 Container `c` 内的某个元素，表达式 `*i = t` 会将该元素以 `t` 取代，但不改变 `c` 的元素总个数。但如果 `ii` 是个 `insert_iterator<C>`，`*ii = t` 就相当于 `c.insert(p, t)`，它会为 `c` 新增元素，而非覆盖（overwrite）`c` 内的任何既有元素。

你可以拿 `insert_iterator` 搭配任何 Container，只要后者对表达式 `C.insert(p, t)` 有所定义。Sequence(p.136)和 Sorted Associative Container(p.156)都定义有这种表达式。两者都藉由 `c.insert(p, x)` 来定义 container 的元素安插动作，但其语义在两种情况中截然不同。

对 Sequence `S` 而言，表达式 `S.insert(p, x)` 表示将数值 `x` 安插于紧邻 iterator `p` 之前。也就是说带有两个引数的 `insert` 版本，可让你控制新元素被安插的位置。对 Sorted Associative Container 而言，并没有这样的控制方式。Sorted Associative Container 的元素一定以 key 的递增顺序出现。Sorted Associative Container 虽定义有带两个引数的 `insert` 版本，但只是作为优化之用：第一个引数只是个提示，指向查找起始位置。

如果你经由 `insert_iterator` 执行多次赋值（assign）动作，你会将多个元素安插于 container 中。面对 Sequence，这些元素会以安插顺序出现在该序列的某个特定位置上。`insert_iterator` constructor 有一个引数为 iterator `p`，新安插的 range 便是被安插于紧邻 `p` 之前。

相较之下，面对 Sorted Associative Container，`insert_iterator` constructor 所收到的 iterator `p` 几乎无关紧要。新增元素不一定会形成相邻的 range，而是会根据 key 的递增顺序出现在 container 的适当位置上。元素的安插顺序只会影响效率：将一个 sorted range 安插到一个 Sorted Associative Container 上，是一种 $O(N)$ 行为。

➢ **样例**

将一整个区间的元素安插到 `list` 中。

```
int main()
{
  list<int> L;
  L.push_front(3);
  insert_iterator<list<int> > ii(L, L.begin());
  *ii++ = 0;
  *ii++ = 1;
  *ii++ = 2;
  copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
  // 打印结果为 0 1 2 3
}
```

将两个 sorted lists 合并，再将其结果安插到一个 set 之中。注意，set 不可含有重复元素。

```
int main()
{
  const int N = 6;

  int A1[N] = {1, 3, 5, 7, 9, 11};
  int A2[N] = {1, 2, 3, 4, 5, 6};
  set<int> result;

  merge(A1, A1 + N, A2, A2 + N,
        inserter(result, result.begin()));

  copy(result.begin(), result.end(), ostream_iterator<int>(cout, " "));
  cout << endl;

  // 输出为 "1 2 3 4 5 6 7 9 11"。
}
```

➢ 何处定义

HP STL 将 insert_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

Container　　Container 的型别；新值安插于此之中。

➢ 隶属何种 concept

Output Iterator（p.96）。

➢ 型别条件

- template 参数 Container 是 Container 的一个 model。
- Container 的大小可改变，如同 Container 需求条件所描述的那样。

- Container 有一个需要两个引数的 insert member function。更明确地说，如果 c 是个 Container object，p 是 Container::iterator object，v 是 Container::value_type object，那么 c.insert(p, v) 是一个有效表达式，其返回型别为 Container::iterator。

➢ Public Base Classes

iterator<output_iterator_tag, void, void, void, void>

➢ 成员

insert_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 insert_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **insert_iterator::insert_iterator(Container& c, Container::iterator i)**

构造一个「可将 object 安插于 c 之中」的 insert_iterator。如果 c 是个 Sequence，那么 object 将被安插于 i 所指元素之前一位置。如果 c 是个 Sorted Associative Container，那么第一个安插动作将以 i 作为查找起始点的提示。i 必须是 c 的一个有效的 iterator（必须可提领，或是 past-the-end）。

insert_iterator::insert_iterator(const insert_iterator&)

这是 copy constructor。（描述于 Assignable。）

insert_iterator::~insert_iterator()

这是 destructor。（描述于 Assignable。）

insert_iterator& insert_iterator::operator=(const insert_iterator&)

这是赋值操作符（assignment operator）。（描述于 Assignable。）

insert_iterator& insert_iterator::operator* ()

此式用来实现 Output Iterator 的相关表达式 *i = x。

insert_iterator&insert_iterator::operator= (const typename Container::value_type&)

此式用来实现 Output Iterator 的相关表达式 *i = x。

insert_iterator& insert_iterator::operator++()

前置累加（preincrement）。（描述于 Output Iterator。）

insert_iterator insert_iterator::operator++(int)

后置累加（postincrement）。（描述于 Output Iterator。）

insert_iterator::iterator_category

这是一个 tag type，用来表现 iterator 的类属（category）：output_iterator_tag。（描述于 Output Iterator。）

※ **template <class Container, class Iterator>**
insert_iterator<Container> inserter(Container& c, Iterator p)

等价于 insert_iterator<Container>(c, i)，这意味 i 必须可转换为 Container::iterator。这个辅助函数之所以存在，主要是为了便利性：它是一个非成员函数（nonmember function），所以 template 参数可被推导出来，而且无须明白声明型别 insert_iterator。

注意，Output Iterator 的定义只要求「*ii = t 需为有效表达式」，并未论及 operator* 或 operator=。针对 operator*，最明显的做法就是让它返回某个 proxy object，后者具备适当的 operator=。而在此情况下，proxy object 就是 insert_iterator 自身。这纯粹是实现细节，因此你不应该倚赖这种做法。就像描述于 Output Iterator 需求条件中的一样，你应该总是写成 *ii = t。

14.2 Stream Iterators

STL 事先定义的 iterators 大多都能迭代于 container 元素之中，但这并非必要条件。Iterator 的概念极为一般化，它们可用来表现「以任何次序安排」之数值（而非只是 container 中已排序好之数值）身上的迭代动作。这种一般性的例子之一就是 STL 的 stream iterators，它能藉由 C++ 的 stream I/O library，执行输入与输出动作。

14.2.1 istream_iterator

```
istream_iterator<T, charT, traits, Distance>
```

istream_iterator 是一种 Input Iterator，它能为「来自某个 basic_istream」的 objects 执行格式化输入动作。一旦 stream 结束，istream_iterator 便呈现一个特别的「stream 终结值」，此值为 past-the-end iterator。Input Iterator 的所有限制都必须被遵守，包括 operator* 和 operator++ 操作顺序上的限制。

原始版的 HP STL 所定义的 istream_iterator，如今已有了改变，因为今天的 C++ 已经具备比以前更一般化的 stream I/O 设施。除非你想使用它，否则没有理由去烦恼这些额外的复杂性。是的，新的 template 参数具有缺省值，而 istream_iterator<int> 仍具有与原先相同的行为。

在最初的 C++ stream I/O 版本中，istream 是一个「由某输入设备读入 char」的 class。然而 C++ standard 中的 istream 并非是个 class，而是一个 typedef。它是 basic_istream<char, char_traits<char> > 的别名，此处的 class basic_istream 是个 template，可为一般化的字符执行格式化输入。HP STL 与 C++ standard 两者的 istream_iterator 有个差别：后者拥有额外的 template 参数 charT 和 traits。一个型别为 istream_iterator<T, charT, traits, Distance> 的 iterator，可自 basic_istream<charT, traits> 读取数据。

1998 年时，并非所有 C++ 编译器都提供新版（也就是说被 templatized）的 stream I/O。为了达到最大可移植性，你最好尽量使用这些缺省值。是的，如果你使用一个 istream_iterator<T>，不管新旧版的 stream I/O 都能正确运作。

➢ 样例

```
int main()
{
  vector<int> V;
  copy(istream_iterator<int>(cin), istream_iterator<int>(),
      back_inserter(V));
}
```

➢ 何处定义

HP STL 将 istream_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

T	istream_iterator 的 value type。Iterator 的 operator* 拥有返回值型别 const T&。
CharT	Input stream 的 char type。缺省为 char。
traits	Input stream 的 char traits class。缺省为 char_traits<charT>。
Distance	Iterator 的 difference type。缺省为 ptrdiff_t。

➢ 隶属何种 concept

Input Iterator（p.94）。

➢ 型别条件

- Value type T 是 Default Constructible 的一个 model。
- Value type T 必须满足：针对一个 T object t 与一个型别为 basic_istream<charT, traits> 的 stream i，表达式 i >> t 有明确定义。
- 型别 charT 与 traits 必须满足「stream 之 char type 以及 char traits type」的需求条件[38]。
- Distance 型别如同 Input Iterator 的需求一样，是一个带正负号的整数（integral）型别。

➢ Public Base Classes

```
iterator<input_iterator_tag, T, Distance, const T*, const T&>
```

[38] char type 和 char traits type 都是 C++ I/O library 的一部分，而非 STL 的一部分，所以本书没有提到这些型别的需求条件。

➢ 成员

istream_iterator 的某些成员并未定义于 Input Iterator 需求条件中，它们是 istream_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **istream_iterator::char_type**
这是 iterator 的 char type：`charT`

※ **istream_iterator::traits_type**
这是 iterator 的 char traits type：`traits`

※ **istream_iterator::istream_type**
这是 iterator 的 stream type：`basic_istream<charT, traits>`

istream_iterator::iterator_category
这是一个 tag type，用来表现 iterator 的类属（category）：`input_iterator_tag`。（描述于 Input Iterator。）

istream_iterator::value_type
这是 iterator 的 value type：`T`（描述于 Input Iterator。）

istream_iterator::difference_type
这是 iterator 的 difference type：`Distance`（描述于 Input Iterator。）

istream_iterator::pointer
这是 iterator 的 pointer type：`const T*`（描述于 Input Iterator。）

istream_iterator::reference
这是 iterator 的 reference type：`const T&`（描述于 Input Iterator。）

※ **istream_iterator::istream_iterator(istream_type& s)**
这是 constructor。产生一个「可从 input stream 读取数据」的 `istream_iterator`。当 `s` 到达 stream 终点，这个 iterator 和 end-of-stream iterator 相比较的结果会相等。end-of-stream iterator 系由 default constructor 产生出来。

※ **istream_iterator::istream_iterator()**
这是 default constructor。产生一个 end-of-stream iterator，那是个 past-the-end（最后元素的下一位置）iterator，可于构造某个区间（range）时派上用场。

istream_iterator::istream_iterator(const istream_iterator&)
这是 copy constructor。（描述于 Input Iterator。）

istream_iterator& istream_iterator::operator=(const istream_iterator&)
这是赋值操作符（assignment operator）。（描述于 Input Iterator。）

const T& istream_iterator::operator*() const
返回 stream 的下一个 object。（描述于 Input Iterator。）

const T* istream_iterator::operator->() const
`i->m` 等价于 `(*i).m`。（描述于 Input Iterator。）

istream_iterator& istream_iterator::operator++()
前置累加（preincrement）。（描述于 Input Iterator。）

istream_iterator istream_iterator::operator++(int)
后置累加（postincrement）。（描述于 Input Iterator。）

bool operator==(const istream_iterator&, const istream_iterator&)
相等（equality）操作符。（描述于 Input Iterator。）

14.2.2 ostream_iterator

ostream_iterator<T, charT, traits>

ostream_iterator 是一种 Output Iterator，它能将一个 T objects 格式化输出到某个特定的 basic_ostream 中。Output Iterator 的所有限制都必须被遵守，包括 operator* 和 operator++ 操作顺序上的限制。

原始版的 HP STL 所定义的 ostream_iterator，如今已有了改变，理由和 istream_iterator 的情况一样：今天的 C++ 已经具备比以前更一般化的 stream I/O 设施。除非你想使用它，否则没有理由去烦恼这些额外的复杂性。是的，新的 template 参数具有缺省值，而 ostream_iterator<int> 仍具有与原先相同的行为。

新版的 stream I/O 中，ostream 并非是个型别，而是一个 typedef。它是 basic_ostream<char, char_traits<char> > 的别名。ostream_iterator 的新版本具有额外的 template 参数，这些参数会被转递（forwarded）为 basic_ostream 的 template 参数。

1998 年时，并非所有 C++ 编译器都提供新版（也就是说被 templatized）的 stream I/O。为了达到最大可移植性，你最好尽量使用这些缺省值。是的，如果你使用一个 ostream_iterator<T>，不管新旧版的 stream I/O 都能正确运作。

➢ 样例

以相反的顺序，将一个 vector 复制到到标准输出设备上，一个元素占一行。此例之中我们所使用的 ostream_iterator constructor 需要两个参数。第二参数是个字符串，会在每个元素打印之后被打印。

```
int main()
{
  vector<int> V;
  for (int i = 0; i < 20; ++i)
      V.push_back(i);
  reverse_copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
}
```

➢ 何处定义

HP STL 将 ostream_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

T 被写至 output stream 的那些数值的型别。ostream_iterator value types 所形成的集合只含有单一型别 T。

charT Output stream 的 char type。**缺省为** char。

traits Output stream 的 char traits class。**缺省为** char_traits<charT>。

➢ 隶属何种 concept

Output Iterator（p.96）。

➢ 型别条件

- 型别 T 必须满足：针对一个 T object t 与一个型别为 basic_ostream<charT, traits> 的 stream out 而言，表达式 out << t 有明确定义。
- 型别 charT 与 traits 必须满足「stream 之 char type 以及 char traits type」的需求条件[39]。

➢ Public Base Classes

iterator<output_iterator_tag, void, void, void, void>

➢ 成员

ostream_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 ostream_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **ostream_iterator::char_type**
这是 iterator 的 char type：charT

※ **ostream_iterator::traits_type**
这是 iterator 的 char traits type：traits

※ **ostream_iterator::ostream_type**
这是 iterator 的 stream type：basic_ostream<charT, traits>

---

[39] char type 和 char traits types 都是 C++ I/O library 的一部分，而非 STL 的一部分，所以本书没有提到这些型别的需求条件。

ostream_iterator::iterator_category

这是一个 tag type，用来表现 iterator 的类属（category）：output_iterator_tag。（描述于 Output Iterator。）

※ **ostream_iterator::ostream_iterator(ostream_type& s)**

这是 constructor。此式产生一个 ostream_iterator，使得通过它所进行的赋值动作犹如 s<<t。

※ **ostream_iterator::ostream_iterator(ostream_type& s, const charT* delim)**

这是具定界符号（delimiter）的 constructor。此式产生一个 ostream_iterator，使得通过它所进行的赋值动作犹如 s << t << delim。

ostream_iterator::ostream_iterator(const ostream_iterator&)

这是 copy constructor。（描述于 Output Iterator。）

ostream_iterator& ostream_iterator::operator=(const ostream_iterator&)

这是赋值操作符（assignment operator）。（描述于 Output Iterator。）

ostream_iterator& ostream_iterator::operator=(const T&)

此式用来实现 Output Iterator 的相关表达式 *i = t。（描述于 Output Iterator。）

ostream_iterator& ostream_iterator::operator*()

此式用来实现 Output Iterator 的相关表达式 *i = t。（描述于 Output Iterator。）

ostream_iterator& ostream_iterator::operator++()

前置累加（preincrement）。（描述于 Output Iterator。）

ostream_iterator ostream_iterator::operator++(int)

后置累加（postincrement）。（描述于 Output Iterator。）

注意，Output Iterator 的定义只要求「*ii=t 需为有效表达式」，并未论及 operator* 或 operator=。针对 operator*，最明显的做法就是让它返回某个 proxy object，后者具备适当的 operator=。而在此情况下，proxy object 就是 ostream_iterator 自身。这纯粹是实现细节，因此你不应该倚赖这种做法。就像描述于 Output Iterator 需求条件中的一样，你应该总是写成 *ii = t。

14.2.3 istreambuf_iterator

istreambuf_iterator<charT, traits>

istreambuf_iterator 和 istream_iterator 非常相似，但它并不执行任意型别的一般格式化输入，而是从 input stream 读入单个字符。它是一种 Input Iterator，并且就像 istream_iterator 一样在遇到 stream 终结时，会以一个特别的 past-the-end 值表示之。

➢ 样例

将标准的 input stream 中剩余的所有字符读进一个缓冲区内。

```
int main()
{
  istreambuf_iterator<char> first(cin);
  istreambuf_iterator<char> end_of_stream;
  vector<char> buffer(first, end_of_stream);
  ...
}
```

➢ 何处定义

HP STL 并未涵盖 istreambuf_iterator。C++ standard 将它声明于 <iterator>。

➢ Template 参数

charT　　istreambuf_iterator 的 value type。这个 iterator 会读进型别为 charT 的字符。

traits　　Input stream 的 char traits class。**缺省为** char_traits<charT>。

➢ 隶属何种 concept

Input Iterator（p.94）。

➢ 型别条件

- 型别 charT 和 traits 必须满足 stream 的 char type 和 char traits type 的需求条件[40]。

➢ Public Base Classes

iterator<input_iterator_tag, charT, traits::off_type, charT*, charT&>

➢ 成员

istreambuf_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 istreambuf_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **istreambuf_iterator::char_type**
　这是 iterator 的 char type：charT

※ **istreambuf_iterator::traits_type**
　这是 iterator 的 char traits type：traits

※ **istreambuf_iterator::int_type**
　等价于 traits::int_type

---

[40] char type 和 char trait types 都是 C++ I/O library 的一部分，不是 STL 的一部分，所以本书并未提到这些型别的需求条件。

※ **istreambuf_iterator::streambuf_type**
等价于 basic_streambuf<charT, traits>

※ **istreambuf_iterator::istream_type**
等价于 basic_istream<charT, traits>

istreambuf_iterator::iterator_category
这是一个 tag type，用来表现 iterator 的类属（category）：input_iterator_tag。（描述于 Input Iterator。）

istreambuf_iterator::value_type
这是 iterator 的 value type：charT（描述于 Input Iterator。）

istreambuf_iterator::difference_type
这是 iterator 的 difference type：traits::off_type（描述于 Input Iterator。）

istreambuf_iterator::pointer
这是 iterator 的 pointer type：charT*（描述于 Input Iterator。）

istreambuf_iterator::reference
这是 iterator 的 reference type：charT&（描述于 Input Iterator。）

※ **istreambuf_iterator::istreambuf_iterator(istream_type& s)**
这是 constructor。它会产生一个 istreambuf_iterator，可藉此从 input stream s 中读值。一旦遇到 stream 终结点，这个 iterator 与「default constructor 所产生之 end-of-stream iterator」相比较的结果必为相等。

※ **istreambuf_iterator::istreambuf_iterator(streambuf_type* s)**
这是 constructor。它会产生一个 istreambuf_iterator，可藉此从 streambuf *s 中读值。一旦遇到 stream 终结，这个 iterator 与「default constructor 产生出来的 end-of-stream iterator」相比较的结果必为相等。如果 s 是 null 指针，这个 constructor 等价于 default constructor。

※ **istreambuf_iterator::istreambuf_iterator()**
这是 default constructor。它会构造出一个 end-of-stream iterator。这是一个 past-the-end iterator，当我们需要构造一个 range 时，它很管用。

charT istreambuf_iterator::operator*() const
返回 stream 的下一个字符。（描述于 Input Iterator。）

istreambuf_iterator& istreambuf_iterator::operator++()
前置累加（preincrement）。（描述于 Input Iterator。）

istreambuf_iterator istreambuf_iterator::operator++(int)
后置累加（postincrement）。注意，其实现有点棘手，因为它必须将 streambuf 累加（也就是移到下一个字符）并在此时返回一个 iterator，仍旧指向目前字符。其典型做法是返回一个 proxy class，并可转换为一个 istreambuf_iterator。（描述于 Input Iterator。）

bool operator==(const istreambuf_iterator&, const istreambuf_iterator&)
相等（equality）操作符。（描述于 Input Iterator。）

※ **bool istreambuf_iterator::equal(const istreambuf_iterator&) const**
表达式 i.equal(j) 等价于表达式 i == j。

14.2.4 ostreambuf_iterator

ostreambuf_iterator<charT, traits>

ostreambuf_iterator 是一种 Output Iterator，可以将字符写入一个 output stream 之中。它和 ostream_iterator 很相似，但不会执行任意型别的一般格式化输出，而是利用 streambuf class 的 sputc member function 输出单个字符。

➢ 样例

使用 ostreambuf_iterator，将字符串写到标准输出设备上。

```
int main()
{
  string s = "This is a test.\n";
  copy(s.begin(), s.end(), ostreambuf_iterator<char>(cout));
}
```

➢ 何处定义

HP STL 并未涵盖 ostreambuf_iterator。C++ standard 将它声明于 <iterator>。

➢ Template 参数

charT　　ostreambuf_iterator 的 value type。此 iterator 能够输出型别为 charT 的字符。

traits　　Output stream 的 char traits class。缺省为 char_traits<charT>。

➢ 隶属何种 concept

Output Iterator（p.96）。

➢ 型别条件

- 型别 charT 与 traits 必须满足 stream 的 char type 与 char traits type 的需求条件[41]。

➢ Public Base Classes

iterator<output_iterator_tag, void, void, void, void>

[41] char type 和 char trait types 都是 C++ I/O library 的一部分，不是 STL 的一部分，所以本书并未提到这些型别的需求条件。

➢ 成员

ostreambuf_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 ostreambuf_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **ostreambuf_iterator::char_type**
等价于 charT

※ **ostreambuf_iterator::traits_type**
等价于 traits

※ **ostreambuf_iterator::streambuf_type**
等价于 basic_streambuf<charT, traits>

※ **ostreambuf_iterator::ostream_type**
等价于 basic_ostream<charT, traits>

※ **ostreambuf_iterator::ostreambuf_iterator(ostream_type& s)**
这是 constructor。产生一个 ostreambuf_iterator，使得通过它对 c 进行的赋值动作等价于 s.rdbuf()->sputc(c)。

ostreambuf_iterator::iterator_category
这是一个 tag type，用来表现 iterator 的类属（category）：output_iterator_tag。（描述于 Output Iterator。）

※ **ostreambuf_iterator::ostreambuf_iterator(streambuf_type* s)**
这是 constructor。产生一个 ostreambuf_iterator，使得通过它对 c 进行的赋值动作等价于 s->sputc(c)。

ostreambuf_iterator& ostreambuf_iterator::operator=(const charT&)
此式用来实现 Output Iterator 的相关表达式 *i=c。如果 i 是一个型别为 ostreambuf_iterator 的 iterator，则表达式 *i = c 等价于 s->sputc(c)，其中 s 是 i 的 streambuf。

ostreambuf_iterator& ostreambuf_iterator::operator*()
此式用来实现 Output Iterator 的相关表达式 *i = c

ostreambuf_iterator& ostreambuf_iterator::operator++()
前置累加（preincrement）。（描述于 Output Iterator。）

ostreambuf_iterator ostreambuf_iterator::operator++(int)
后置累加（postincrement）。（描述于 Output Iterator。）

※ **bool ostreambuf_iterator::failed() const**
当且仅当（if and only if）前一次调用 sputc 时返回 eof，此函数返回 true。

14.3 reverse_iterator

reverse_iterator<Iterator>

Class reverse_iterator 是一种 iterator adapter，能够在 range 上逆向移动。「在 reverse_iterator<Iter> object 上执行 operator++」和「在 Iter object 上执行 operator--」

的结果是相同的，反之亦然。给定一个 [f,l)，则 [reverse_iterator<Iter>(l), reverse_iterator<Iter>(f)) 恰恰包含相同的元素，但次序相反。

这项性质有一个很重要的结果。f 和 l 的角色颠倒了：l 表示 range 尾端，reverse_iterator(l) 则表示被颠倒之 range 的开头。然而，range 的开头与尾端并不对称，因为 f 是 [f,l) 的第一个 iterator，而 l 是最后一个 iterator 的「下一位置」。

这意味 reverse_iterator(l) 并不会指向 l 所指的元素。（就此而言，l 可能根本就不指向任何元素；它可能是一个 past-the-end iterator。）取而代之的是 reverse_iterator(l) 必须指向 *(l-1) 元素。Revese iterator 的同一性（identity）为：

```
&*(reverse_iterator(i)) == &*(i-1)
```

reverse_iterator<Iter> 会维护其底部（underlying）iterator（亦即 Iter）的值，你可以利用 base() member function 取得其底部 iterator。因此你也可以把 reverse iterators 的同一性（identity）写成这样：

```
&*(ri) == &*(ri.base() - 1)。
```

原始的 HP STL 拥有两个不同的 classes：(1) reverse_iterator，只可用于 Random Access Iterators；(2) reverse_bidirectional_iterator，可用于其他的 Bidirectional Iterators。这样的差异之所以存在，纯粹是技术上的因素[42]，而且 reverse_bidirectional_iterator 现在也已经完全不需要了；某些实现品为了向下兼容，仍然提供这个 class，但事实上它已经从 C++ standard 中移除了。

➢ 样例

```
int main()
{
  const int N = 10;
  int A[N] = {2, 5, 7, 8, 1, 5, 3, 6, 9, 1};
  list<int> L(A, A+N);
  // 以向前顺序（forward order）打印 L 的元素。
  copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  // 以颠倒顺序（reverse order）打印 L 的元素。
  copy(reverse_iterator<list<int>::iterator>(L.end()),
      reverse_iterator<list<int>::iterator>(L.begin()),
      ostream_iterator<int>(cout, " "));
  cout << endl;
  // 找出 '5' 的第一次出现地点
  list<int>::iterator i1 = find(L.begin(), L.end(), 5);
```

[42] 因为原始的 HP STL 并未拥有 iterator_traits 技术（p.177）。

```
  // 找出 '5' 的最后一次出现地点
  list<int>::iterator i2 =
    (find(reverse_iterator<list<int>::iterator>(L.end()),
          reverse_iterator<list<int>::iterator>(L.begin()),
          5)).base();
  --i2;
  assert(*i1 == 5 && *i2 == 5 && distance(i1, i2) == 4);
}
```

实用上，当你希望在 list 中逆向移动，你永远不会写出上面那样的代码。因为就像其他 Reversible Containers 一样，list 所含的 member functions 和嵌套型别，可让你更轻易使用 reverse iterators。你应该会以：

```
list<int>::reverse_iterator ri = L.rbegin();
```

取代

```
reverse_iterator<list<int>::iterator> ri(L.rbegin());
```

如果你需要自行定义一个 Reversible Container，你必须为你的 container 提供上一段所说的那些 typedefs 和 member functions。或许你会使用以下的程序片段：

```
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;

reverse_iterator rbegin() { return reverse_iterator(end()); }
reverse_iterator rend() { return reverse_iterator(begin()); }

const_reverse_iterator rbegin() const
  { return const_reverse_iterator(end()); }
const_reverse_iterator rend() const
  { return const_reverse_iterator(begin()); }
```

➢ 何处定义

HP STL 将 reverse_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <iterator>。

➢ Template 参数

Iterator　　此为 reverse iterator 的 base iterator type，是一个 Bidirectional Iterator。

➢ 隶属何种 concept

Bidirectional Iterator（p.102）。如果 Iterator 是 Random Access Iterator（p.103）的一个 model，则 reverse_iterator<Iterator> 同样也是 Random Access Iterator 的一个 model。

➢ 型别条件

- template 参数 `Iterator` 是 Bidirectional Iterator 的一个 model。注意，`reverse_iterator` 的某些 member functions 只有在 `Iterator` 同样也是 Random Access Iterator 时才可用。

➢ Public Base Classes

```
iterator<typename iterator_traits<Iterator>::iterator_category,
         typename iterator_traits<Iterator>::value_type,
         typename iterator_traits<Iterator>::difference_type,
         typename iterator_traits<Iterator>::pointer,
         typename iterator_traits<Iterator>::reference>
```

➢ 成员

`reverse_iterator` 的某些成员并未定义于 Bidirectional Iterator 或Random Access Iterator 的需求条件中，它们是 `reverse_iterator` 特有的。这些特别的成员以 ※ 符号标示之。

`reverse_iterator::iterator_category`

这是一个 tag type，用来表现 iterator 的类属（category）。它和 `Iterator` 的 category tag 相同。（描述于 Input Iterator。）

`reverse_iterator::value_type`

这是 reverse iterator 的 value type，和 `Iterator` 的 value type 相同。（描述于 Input Iterator。）

`reverse_iterator::difference_type`

这是 reverse iterator 的 difference type，与 `Iterator` 的 difference type 相同。（描述于 Input Iterator。）

`reverse_iterator::pointer`

这是 reverse iterator 的 pointer type，与 `Iterator` 的 pointer type 相同。（描述于 Input Iterator。）

`reverse_iterator::reference`

这是 reverse iterator 的 reference type，与 `Iterator` 的 reference type 相同。（描述于 Input Iterator。）

※ **`reverse_iterator::iterator_type`**

与 `Iterator` 相同。

※ **`explicit reverse_iterator::reverse_iterator(Iterator i)`**

这是 constructor。构造出一个 `reverse_iterator`，以 `i` 为底部（underlying）iterator。

`reverse_iterator::reverse_iterator()`

这是 default constructor。（描述于 Input Iterator。）

`reverse_iterator::reverse_iterator(const reverse_iterator&)`

这是 copy constructor。（描述于 Input Iterator。）

※ **`template<class I>reverse_iterator::reverse_iterator(const reverse_iterator<I>& i)`**

这是一般化的 copy constructor。构造出一个 `reverse_iterator`，其底部（underlying）iterator 为 `i.base()`。

这个 constructor 的效用在于：当且仅当 X 可转换为 Y，则 reverse_iterator<X>可转换为 reverse_iterator<Y>。它存在的主要原因是为了让 Reversible Constainer 的 reverse_iterator type 可转换为其 const_reverse_iterator type，正如每个 Container 的 iterator type 可转换为其 const_iterator type 一样。

reverse_iterator& reverse_iterator::operator=(const reverse_iterator&)
这是赋值操作符（assignment operator）。（描述于 Input Iterator。）

※ **Iterator reverse_iterator::base() const**
返回其底部（underlying）iterator。

reference reverse_iterator::operator*() const
返回 iterator 所指元素。（描述于 Forward Iterator。）

reverse_iterator& reverse_iterator::operator++()
前置累加（preincrement）。（描述于 Forward Iterator。）

reverse_iterator reverse_iterator::operator++(int)
后置累加（postincrement）。（描述于 Forward Iterator。）

reverse_iterator& reverse_iterator::operator--()
前置递减（predecrement）。（描述于 Bidirectional Iterator。）

reverse_iterator reverse_iterator::operator--(int)
后置递减（postdecrement）。（描述于 Bidirectional Iterator。）

reverse_iterator reverse_iterator::operator+(difference_type)
Iterator 相加。当 Iterator 是一个 Random Access Iterator 时，这个 member function 才有用。（描述于 Random Access Iterator。）

reverse_iterator reverse_iterator::operator+=(difference_type)
Iterator 相加。当 Iterator 是一个 Random Access Iterator 时，这个 member function 才有用。（描述于 Random Access Iterator。）

reverse_iterator reverse_iterator::operator-(difference_type)
Iterator 相减。当 Iterator 是一个 Random Access Iterator 时，这个 member function 才有用。（描述于 Random Access Iterator。）

reverse_iterator reverse_iterator::operator-=(difference_type)
Iterator 相减。当 Iterator 是一个 Random Access Iterator 时，这个 member function 才有用。（描述于 Random Access Iterator。）

reference& reverse_iterator::operator[](difference_type) const
元素访问。当 Iterator 是一个 Random Access Iterator 时，这个 member function 才有用。（描述于 Random Access Iterator。）

reverse_iterator operator+(difference_type, reverse_iterator)

Iterator 加法。当 Iterator 是一个 Random Access Iterator 时，这个函数才有用。（描述于 Random Access Iterator。）

difference_type operator-(const reverse_iterator&, const reverse_iterator&)

返回两个 iterators 的差距。当 Iterator 是一个 Random Access Iterator 时，这个函数才有用。（描述于 Random Access Iterator。）

bool operator=(const reverse_iterator&, const reverse_iterator&)

相等（equality）操作符。（描述于 Input Iterator。）

bool operator<(const reverse_iterator&, const reverse_iterator&)

当且仅当第一引数在第二引数之前，则此函数返回 true。当 Iterator 是一个 Random Access Iterator 时，这个函数才有用。（描述于 Random Access Iterator。）

14.4 raw_storage_iterator

raw_storage_iterator<ForwardIterator, T>

class raw_storage_iterator 是一种 adapter，可以让 STL 算法与低阶内存操作彼此结合起来。当你有必要将内存的分配与对象的构造分开处理时，它就可以派上用场。

如果 i 是一个 iterator，指向一块未经初始化的内存，你可以利用 construct(p.189)或 placement new 操作符，在 i 所指位置上产生一个 object。raw_storage_iterator adapter 可以让这个程序变得更方便，它是一种 Output Iterator，在 ForwardIterator iterator 所指的内存上构造 objects。

raw_storage_iterator<Iter>r 与其底部（underlying）iteartor i（隶属 Iter 型别）指向同一块内存。表达式 *r = x 等价于表达式 construct(&*i, x)。

除非你打算撰写一个 container 或是一个 ***adaptive*** 算法，否则不应该用到 raw_storage_iterator adapter。事实上你或许根本就不该用它；直接使用 construct（或其他「得以将某个 range 中的所有成员都初始化」的算法）几乎总是比较好的选择。

以 raw_storage_iteratorg 撰写 "exception-safe" 代码是有可能的，但并不容易。如果你关心 "exception safety"（异常情况下的安全性）问题，更简单的做法是采用算法 uninitialized_copy（p.192）、uninitialized_fill（p.194）和 uninitialized_fill_n（p.195）。

➢ 样例

```
class Int {
public:
  Int(int x) : val(x) {}        // Int 不具有 default constructor
  int get() { return val; }
```

```
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  transform(A1, A1 + N,
            raw_storage_iterator<Int*, int>(A2),
            negate<int>());
}
```

➢ 何处定义

HP STL 将 raw_storage_iterator 定义于头文件 <iterator.h>。C++ standard 将它声明于 <memory>。

➢ Template 参数

ForwardIterator	raw_storage_iterator 底部（underlying）iterator 的型别。
T	ForwardIterator::value_type 的 constructor 的引数型别。

➢ 型别条件

- ForwardIterator 是 Forward Iterator 的 model。
- ForwardIterator 是可变的（mutable）。
- ForwardIterator value type 具有单一引数（型别为 T）的 constructor。

➢ Public Base Classes

iterator<output_iterator_tag, void, void, void, void>

➢ 成员

raw_storage_iterator 的某些成员并未定义于 Output Iterator 需求条件中，它们是 raw_storage_iterator 特有的。这些特别的成员以 ※ 符号标示之。

※ **raw_storage_iterator::raw_storage_iterator(ForwardIterator i)**
这是 constructor。产生一个 raw_storage_iterator，以 i 为底部（underlying）iterator。

raw_storage_iterator::raw_storage_iterator(const raw_storage_iterator&)
这是 copy constructor。（描述于 Output Iterator。）

raw_storage_iterator&
raw_storage_iterator::operator=(const raw_storage_iterator&)
这是赋值操作符（assignment operator）。（描述于 Output Iterator。）

raw_storage_iterator& raw_storage_iterator::operator*()
此式用来实现 Output Iterator 的相关表达式 `*i = t`。（描述于 Output Iterator。）

raw_storage_iterator& raw_storage_iterator::operator=(const T& val)
此式用来实现 Output Iterator 的相关表达式 `*i = t`。以 `val` 作为 constructor 的引数，在 iterator 所指位置处构造出一个 object。新的 object 的型别为 `ForwardIterator` 的 value type。

raw_storage_iterator& raw_storage_iterator::operator++()
前置累加（preincrement）。（描述于 Output Iterator。）

raw_storage_iterator raw_storage_iterator::operator++(int)
后置累加（postincrement）。（描述于 Output Iterator。）

raw_storage_iterator::iterator_category
这是一个 tag type，用来表现 iterator 的类属（category）：`output_iterator_tag`。（描述于 Output Iterator。）

注意，Output Iterator 的定义只要求「`*ii = t` 需为有效表达式」，并未论及 `operator*` 或 `operator=`。针对 `operator*`，最明显的做法就是让它返回某个 proxy object，后者具备适当的 `operator=`。而在此情况下，proxy object 就是 `raw_storage_iterator` 自身。这纯粹是实现细节，因此你不应该倚赖这种做法。就像描述于 Output Iterator 需求条件中的一样，你应该总是写成 `*ii = t`。

第 15 章

Function Object Classes
函数对象类

就像第 8 章所讨论过的，很多 STL 算法会利用 *function objects* 将执行动作「参数化」。STL 含有大量事先定义的 function objects，可执行基本的数值运算和逻辑运算。

STL 事先定义的所有 function objects 都隶属于 Adaptable Unary Function（p.114）或 Adaptable Binary Function（p.115），意味它们都声明有嵌套型别，用以描述其引数型别和返回值型别。如果你要自行实现一个 function object，有一种方法可以提供这些嵌套型别（但此法并非唯一，也不见得最好），就是继承自一个空的 base classes `unary_function`（稍后叙述）或 `binary_function`（p.372）。有时你可能会发现，直接以 typedefs 定义这些嵌套型别会比较简单，因为在 C++ 语言中，从一个 template base class 继承型别的声明，有相当复杂的技术限制。

C++ standard 规定所有标准的 function object classes 都必须继承自 `unary_function` 或 `binary_function`，本书遵循这个规则，但你应该记住这纯粹只是实现细节。真正的重点是，它们必须具备 Adaptable Unary Function 与 Adaptable Binary Function 这两个 concepts 所要求的嵌套型别。

15.1 Function Object Base Classes

15.1.1 unary_function

`unary_function<Arg, Result>`

`unary_function` 是一个空的 base class，其中没有任何 member functions 或 member variables，只有型别信息。它的存在是为了让 Adaptable Unary Function(p.114)models 的定义更方便些。是的，Adaptable Unary Function 的 models 都必须内含嵌套型别的声明，而继承 `unary_function` 则是获得这些嵌套型别的方便法门。

➢ 样例

```
struct sine : public unary_function<double, double> {
  double operator()(double x) const { return sin(x); }
};
```

➢ 何处定义

HP STL 将 unary_function 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

Arg　　此 function object 之引数型别（argument type）。

Result　　此 function object 之结果型别（result type）。

➢ 隶属何种 concept

Assignable（p.83）、Default Constructible（p.84）。

➢ Public Base Classes

无

➢ 成员

unary_function 的所有成员都是嵌套型别（nested types）：

unary_function::argument_type

function object 的引数型别。这是针对 template 引数 Arg 而设计的一个 typedef。

unary_function::result_type

function object 的 result type。这是针对 template 引数 Result 而设计的一个 typedef。

15.1.2 binary_function

binary_function<Arg1, Arg2, Result>

binary_function class 是一个空的 base class，不含任何 member functions 或 member variables，只含型别信息。它的存在让 Adaptable Binary Function models 的定义变得更方便些。Adaptable Binary Function 的任何 models 都必须包含嵌套型别声明，而继承 base class binary_function 就是获得这些 typedefs 的一种方便法门。

➢ 样例

```
struct exponentiate : public binary_function<double, double, double>
{
  double operator()(double x, double y) const { return pow(x, y); }
};
```

➢ 何处定义

HP STL 将 binary_function 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

Arg1 function object 之第一引数的型别。

Arg2 function object 之第二引数的型别。

Result function object 的结果型别。

➢ 隶属何种 concept

Assignable（p.83）、Default Constructible（p.84）。

➢ Public Base Classes

无

➢ 成员

binary_function 所有的成员都是嵌套型别（nested types）：

binary_function::first_argument_type

function object 的第一引数型别。这是针对 template 引数 Arg1 而设计的一个 typedef。

binary_function::second_argument_type

function object 的第二引数型别。这是针对 template 引数 Arg2 而设计的一个 typedef。

binary_function::result_type

function object 的 result type。这是针对 template 引数 Result 而设计的一个 typedef。

15.2 算术运算（Arithmetic Operations）

15.2.1 plus

plus<T>

Class plus<T> 是一种 Adaptable Binary Function。如果 f 是 class plus<T> 的一个 object 而且 x 和 y 都是型别为 T 的值，则 f(x,y) 返回 x + y。

➢ 样例

令 v3 内的每个元素是 v1 和 v2 内对应元素之和。

```
int main()
{
  const int N = 1000;
  vector<double> v1(N);
  vector<double> v2(N);
  vector<double> v3(N);

  generate(v1.begin(), v1.end(), rand);
  fill(v2.begin(), v2.end(), -RAND_MAX / 2.);

  transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),
            plus<double>());

  for (int i = 0; i < N; ++i)
    assert(v3[i] == v1[i] + v2[i]);
}
```

➢ 何处定义

HP STL 将 `plus` 声明于头文件 `<function.h>`。C++ standard 将它声明于 `<functional>`。

➢ Template 参数

`T` function object 的引数型别和结果型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- `T` 为 Assignable。
- `T` 为数值型别（numeric type）。如果 `x` 和 `y` 都属于 `T` 型别，则 `x + y` 有定义，且其返回型别可转换为 `T`。

➢ Public Base Classes

`binary_function<T, T, T>`

➢ 成员

`plus::first_argument_type`

第一引数之型别：`T`（描述于 Adaptable Binary Function。）

`plus::second_argument_type`

第二引数之型别：`T`（描述于 Adaptable Binary Function。）

plus::result_type

执行结果之型别：T（描述于 Adaptable Binary Function。）

T plus::operator()(const T& x, const T& y) const

function call 操作符。执行结果为 x + y。（描述于 Adaptable Binary Function。）

plus::plus()

default constructor。（描述于 Default Constructible。）

15.2.2 minus

minus<T>

Class minus<T> 是一种 Adaptable Binary Function。如果 f 是 class minus<T> 的一个 object 而且 x 和 y 都属于 T 型别，则 f(x,y) 返回 x - y。

➢ 样例

v3 内的每个元素都是 v1 和 v2 内对应元素之差。

```
int main()
{
  const int N = 1000;
  vector<double> v1(N);
  vector<double> v2(N);
  vector<double> v3(N);

  generate(v1.begin(), v1.end(), rand);
  fill(v2.begin(), v2.end(), -RAND_MAX / 2.);

  transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),
            minus<double>());

  for (int i = 0; i < N; ++i)
    assert(v3[i] == v1[i] - v2[i]);
}
```

➢ 何处定义

HP STL 将 minus 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T function object 的引数型别与结果型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Assignable。
- T 为数值型别（numeric type）。如果 x 和 y 都属于 T 型别，则 x - y 有定义，且其结果可转换为 T。

➢ Public Base Classes

binary_function<T, T, T>

➢ 成员

minus::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Function。）

minus::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Function。）

minus::result_type

执行结果之型别：T （描述于 Adaptable Binary Function。）

T minus::operator()(const T& x, const T& y) const

function call 操作符。执行结果为 x - y。（描述于 Adaptable Binary Function。）

minus::minus()

default constructor。（描述于 Default Constructible。）

15.2.3 multiplies

multiplies<T>

Class multiplies<T> 是一种 Adaptable Binary Function[43]。如果 f 是 class multiplies<T> 的一个 object 且 x 和 y 都属于 T 型别，则 f(x,y) 返回 x * y。

➢ 样例

将 1! 至 20! 的值填入一个表格中。

---

[43] HP STL 将这个 function object 称为 times。C++ standard 将它易名为 multiplies，因为 times 这个名字与 UNIX 头文件 <sys/times.h>中定义的函数相冲。

```
int main() {
  const int N = 20;
  vector<double> V(N);
  for (int i = 0; i < N; ++i)
    V[i] = i + 1;

  partial_sum(V.begin(), V.end(), V.begin(), multiplies<double>());
  copy(V.begin(), V.end(), ostream_iterator<double>(cout, "\n"));
}
```

➢ 何处定义

HP STL 将 `multiplies` 声明于头文件 `<function.h>`。C++ standard 将它声明于 `<functional>`。

➢ Template 参数

`T` function object 的引数型别与结果型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- `T` 为 Assignable。
- `T` 为数值型别（numeric type）。如果 `x` 和 `y` 都属于型别 `T`，则 `x * y` 有定义，其结果可转换为 `T`。

➢ Public Base Classes

`binary_function<T, T, T>`

➢ 成员

`multiplies::first_argument_type`

第一引数之型别：`T`（描述于 Adaptable Binary Function。）

`multiplies::second_argument_type`

第二引数之型别：`T`（描述于 Adaptable Binary Function。）

`multiplies::result_type`

执行结果之型别：`T`（描述于 Adaptable Binary Function。）

`T multiplies::operator()(const T& x, const T& y) const`

function call 操作符。执行结果为 `x * y`。（描述于 Adaptable Binary Function。）

`multiplies::multiplies()`

default constructor。（描述于 Default Constructible。）

15.2.4 divides

divides<T>

Class divides<T> 是一种 Adaptable Binary Function。如果 f 是 class divides<T> 的一个 object 且 x 和 y 都属于型别 T，则 f(x,y) 返回 x / y。

➢ 样例

将 vector 内的每个元素都除以某个常量值。

```
int main() {
  const int N = 1000;
  vector<double> v(N);

  generate(v.begin(), v.end(), rand);
  transform(v.begin(), v.end(), v.begin(),
            bind2nd(divides<double>(), double(RAND_MAX)));

  for (int i = 0; i < N; ++i)
    assert(0 <= v[1] && v[i] <= 1.);
}
```

➢ 何处定义

HP STL 将 divies 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　function object 的引数型别与结果型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Assignable。
- T 为数值型别（numeric type）。如果 x 和 y 都属于型别 T，则 x / y 有定义，且其结果可转换为 T。

➢ Public Base Classes

binary_function<T, T, T>

➢ 成员

divides::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Function。）

divides::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Function。）

divides::result_type

执行结果之型别：T （描述于 Adaptable Binary Function。）

T divides::operator()(const T& x, const T& y) const

function call 操作符。执行结果为 x / y。（描述于 Adaptable Binary Function。）

divides::divides()

default constructor。 （描述于 Default Constructible。）

15.2.5 modulus

modulus<T>

Class modulus<T> 是一种 Adaptable Binary Function。如果 f 是 class modulus<T> 的一个 object 且 x 和 y 都属于型别 T，则 f(x,y) 返回 x % y。

➢ 样例

将 vector 内的每一个元素都以其最后一个阿拉伯数字取代之。

```
int main() {
  const int N = 1000;
  vector<int> v(N);

  generate(v.begin(), v.end(), rand);
  transform(v.begin(), v.end(), v.begin(),
            bind2nd(modulus<int>(), 10));

  for (int i = 0; i < N; ++i)
    assert(0 <= v[1] && v[i] <= 10);
}
```

➢ 何处定义

HP STL 将 modulus 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T function object 的引数型别与结果型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Assignable。
- T 为数值型别（numeric type）。如果 x 和 y 都属于型别 T，则 x % y 有定义，且其结果可转换为 T。

➢ Public Base Classes

binary_function<T, T, T>

➢ 成员

modulus::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Function。）

modulus::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Function。）

modulus::result_type

执行结果之型别：T （描述于 Adaptable Binary Function。）

T modulus::operator()(const T& x, const T& y) const

function call 操作符。执行结果为 x % y。（描述于 Adaptable Binary Function。）

modulus::modulus()

default constructor。（描述于 Default Constructible。）

15.2.6 negate

negate<T>

Class negate<T> 是一种 Adaptable Unary Function，是一个单引数的 function object。如果 f 是 class negate<T> 的一个 object 且 x 属于型别 T，则 f(x) 返回 -x。

➢ 样例

构造一个 vector v2，使其每个元素都是 v1 相对元素的负值。

```
int main() {
  const int N = 1000;
```

```
  vector<int> v1(N);
  vector<int> v2(N);

  generate(v1.begin(), v1.end(), rand);
  transform(v1.begin(), v1.end(), v2.begin(),
          negate<int>());

  for (int i = 0; i < N; ++i)
    assert(v1[i] + v2[i] == 0);
}
```

➢ 何处定义

HP STL 将 negate 声明于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　function object 的引数型别与结果型别。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Assignable。
- T 为数值型别（numeric type）。如果 x 隶属型别 T，则 -x 有定义，而且其结果可转换为 T。

➢ Public Base Classes

unary_function<T, T>

➢ 成员

negate::argument_type

引数型别：T （描述于 Adaptable Unary Function。）

negate::result_type

执行结果之型别：T （描述于 Adaptable Unary Function。）

T negate::operator()(const T& x) const

function call 操作符。执行结果为 -x。 （描述于 Adaptable Unary Function。）

negate::negate()

default constructor。（描述于 Default Constructible。）

15.3 大小比较（Comparisons）

15.3.1 equal_to

equal_to<T>

Class equal_to<T> 是一种 Adaptable Binary Predicate，这表示它可用来验证某条件的真或伪。如果 f 是 class equal_to<T> 的一个 object 且 x 与 y 为型别 T 之值，则只有在 x == y 时 f(x,y) 才会返回 true。

➢ 样例

重新安排一个数组，将等于零的所有元素排在不等于零的所有元素之前。

```
int main()
{
  const int N = 10;
  int A[N] = {1, 3, 0, 2, 5, 9, 0, 0, 6, 0};

  partition(A, A+N, bind2nd(equal_to<int>(), 0));
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 equal_to 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　equal_to 之引数型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Equality Comparable 的一个 model。

➢ Public Base Classes

binary_function<T, T, bool>

➢ 成员

equal_to::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Predicate。）

equal_to::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Predicate。）

equal_to::result_type

执行结果之型别：bool （描述于 Adaptable Binary Predicate。）

bool equal_to::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x == y。（描述于 Binary Predicate。）

equal_to::equal_to()

default constructor。（描述于 Default Constructible。）

15.3.2 not_equal_to

not_equal_to<T>

Class not_equal_to<T> 是一种 Adaptable Binary Predicate，这表示它可用来验证某条件之真伪。如果 f 是 class not_equal_to<T> 的一个 object 而且 x 和 y 都为型别 T 之值，则只有在 x != y 的时候 f(x,y) 才返回 true。

➢ 样例

在 list 之中找寻第一个非零元素。

```
int main()
{
  const int N = 9;
  int A[N] = {0, 0, 0, 0, 1, 2, 4, 8, 0};
  list<int> L(A, A+N);

  list<int>::iterator i = find_if(L.begin(), L.end(),
                                  bind2nd(not_equal_to<int>(), 0));
  cout << "Elements after initial zeros (if any): ";
  copy(i, L.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 not_equal_to 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ **Template 参数**

T　　not_equal_to 之引数型别。

➢ **隶属何种 concept**

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

➢ **型别条件**

- T 为 Equality Comparable 的一个 model。

➢ **Public Base Classes**

binary_function<T, T, bool>

➢ **成员**

not_equal_to::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Predicate。）

not_equal_to::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Predicate。）

not_equal_to::result_type

执行结果之型别：bool （描述于 Adaptable Binary Predicate。）

bool not_equal_to::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x != y。 （描述于 Binary Predicate。）

not_equal_to::not_equal_to()

default constructor。（描述于 Default Constructible。）

15.3.3 less

less<T>

Class less<T> 是一种 Adaptable Binary Predicate，这表示它可以用来验证某条件之真伪。如果 f 是 class less<T> 的一个 object 而且 x 和 y 都是型别 T 之值，则只有当 x < y 时 f(x,y) 才返回 true。

STL 的许多 classes 及 algorithms 都需要用到比较函数，例如 sort、set、map。less 是其典型的缺省值。

➢ 样例

重新安排一个数组，使所有负数排在非负数之前。

```
int main()
{
  const int N = 10;
  int A[N] = {1, -3, -7, 2, 5, -9, -2, 1, 6, -8};

  partition(A, A+N, bind2nd(less<int>(), 0));
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 `less` 定义于头文件 `<function.h>`。C++ standard 将它声明于 `<functional>`。

➢ Template 参数

`T` `less` 之引数型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。此外，当且仅当（if and only if）T 是 Strict Weakly Comparable（p.88）的一个 model，则 `less<T>` 是 Strict Weak Ordering（p.119）的一个 model。

➢ 型别条件

- `T` 为 LessThan Comparable 的一个 model。

➢ Public Base Classes

```
binary_function<T, T, bool>
```

➢ 成员

`less::first_argument_type`

第一引数之型别：`T`（描述于 Adaptable Binary Predicate。）

`less::second_argument_type`

第二引数之型别：`T`（描述于 Adaptable Binary Predicate。）

`less::result_type`

执行结果之型别：`bool`（描述于 Adaptable Binary Predicate。）

bool less::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x < y。（描述于 Binary Predicate。）

less::less()

default constructor。（描述于 Default Constructible。）

15.3.4 greater

greater<T>

Class greater<T> 是一种 Adaptable Binary Predicate，这表示它可用来验证某条件之真伪。如果 f 是 class greater<T> 的一个 object 而且 x 和 y 都是型别 T 之值，则只有当 x > y 时 f(x,y) 才返回 true。

➢ 样例

将一个 vector 以「递减顺序而非递增顺序」做排序。

```
int main()
{
  const int N = 10;
  int A[N] = {1, -3, -7, 2, 5, -9, -2, 1, 6, -8};
  vector<int> V(A, A+N);

  sort(V.begin(), V.end(), greater<int>());
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 greater 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　greater 之引数型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。此外，当且仅当（if and only if）T 是 Strict Weakly Comparable（p.88）的一个 model，则 greater<T> 是 Strict Weak Ordering（p.119）的一个 model。

➢ 型别条件

- T 为 LessThan Comparable 的一个 model。

➢ **Public Base Classes**

binary_function<T, T, bool>

➢ **成员**

greater::first_argument_type

第一引数之型别：T。（描述于 Adaptable Binary Predicate。）

greater::second_argument_type

第二引数之型别：T。（描述于 Adaptable Binary Predicate。）

greater::result_type

执行结果之型别：bool。（描述于 Adaptable Binary Predicate。）

bool *greater::operator()*(const T& x, const T& y) const

function call 操作符。其返回值为 x > y。（描述于 Binary Predicate。）

greater::greater()

default constructor。（描述于 Default Constructible。）

15.3.5 less_equal

less_equal<T>

Class less_equal<T> 是一种 Adaptable Binary Predicate，这表示它可以用来验证某条件之真伪。如果 f 是 class less_equal<T> 的一个 object 而且 x 和 y 都是型别 T 之值，则只有当 x <= y 时 f(x,y) 才返回 true。

➢ **样例**

产生一个 list，令其元素为某个 range 之内的所有正数值。

```
int main()
{
  const int N = 10;
  int A[N] = {3, -7, 0, 6, 5, -1, -3, 0, 4, -2};

  list<int> L;

  remove_copy_if(A, A+N, back_inserter(L),
                 bind2nd(less_equal<int>(), 0));

  cout << "Elements in list: ";
  copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 less_equal 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T less_equal 之引数型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

注意，less_equal 并非 Strict Weak Ordering 的一个 model。例如你不能以 less_equal 作为 sort 的比较函数。它甚至不是一个 partial ordering，更遑论是个 strict weak ordering。面对 partial ordering，f(x, x) 一定总是 false。

➢ 型别条件

- T 为 LessThan Comparable 的一个 model。

➢ Public Base Classes

binary_function<T, T, bool>

➢ 成员

less_equal::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Predicate。）

less_equal::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Predicate。）

less_equal::result_type

执行结果之型别：bool （描述于 Adaptable Binary Predicate。）

bool less_equal::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x <= y。（描述于 Binary Predicate。）

less_equal::less_equal()

default constructor。 （描述于 Default Constructible。）

15.3.6 greater_equal

greater_equal<T>

Class greater_equal<T> 是一种 Adaptable Binary Predicate，这表示它可用来验证某条件之真伪。如果 f 是 class greater_equal<T> 的一个 object 而且 x 和 y 都是型别 T 之值，则只有当 x >= y

时 f(x,y) 才返回 true。

➢ 样例

找寻 vector 中第一个不为负的值。

```
int main()
{
  const int N = 10;
  int A[N] = {-4, -3, 0, -6, 5, -1, -3, 0, 4, -2};
  vector<int> v(A, A+N);

  vector<int>::iterator i = find_if(v.begin(), v.end(),
                                    bind2nd(greater_equal<int>(), 0));
  assert(i == v.end() || *i >= 0);
}
```

➢ 何处定义

HP STL 将 greater_equal 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　greater_equal 之引数的型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

注意，greater_equal 并非 Strict Weak Ordering 的一个 model。它甚至不是一个 partial ordering，更遑论是个 strict weak ordering。面对 partial ordering，f(x, x) 一定总是 false。

➢ 型别条件

- T 为 LessThan Comparable 的一个 model。

➢ Public Base Classes

binary_function<T, T, bool>

➢ 成员

greater_equal::first_argument_type

第一引数之型别：T　（描述于 Adaptable Binary Predicate。）

greater_equal::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Predicate。）

greater_equal::result_type

执行结果之型别：bool （描述于 Adaptable Binary Predicate。）

bool greater_equal::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x >= y。（描述于 Binary Predicate。）

greater_equal::greater_equal()

default constructor。 （描述于 Default Constructible。）

15.4 逻辑运算（Logical Operations）

逻辑运算（logical）function objects 和算术运算（arithmetic）function objects 很相似；它们执行的是诸如 x || y 和 x && y 这类操作。

logical_and 和 logical_or 自身并不很有用，它们主要的用途在于与 function object ***adapter*** binary_compose（p.433）结合，那个时候它们便可以在其他 function objects 身上执行逻辑运算。

15.4.1 logical_and

logical_and<T>

Class logical_and<T> 是一种 Adaptable Binary Predicate，这表示它可以用来验证某条件之真伪。如果 f 是 class logical_and<T> 的一个 object 而且 x 和 y 都是型别 T 之值，T 可转换为 bool，则只有当 x 和 y 皆为 true 时 f(x,y) 才返回 true。

➢ **样例**

找寻 list 之内第一个介于 0...10 的元素。

```
int main()
{
  list<int> L;
  generate_n(back_inserter(L), 10000, rand);

  list<int>::iterator i =
    find_if(L.begin(), L.end(),
            compose2(logical_and<bool>();
                     bind2nd(greater_equal<int>(), 1),
                     bind2nd(less_equal<int>(), 10)));
  assert(i == L.end() || (*i >= 1 && *i <= 10));
}
```

➢ **何处定义**

HP STL 将 `logical_and` 定义于头文件 `<function.h>`。C++ standard 将它声明于 `<functional>`。

➢ **Template 参数**

`T` `logical_and` 之引数型别。

➢ **隶属何种 concept**

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

➢ **型别条件**

- `T` 可转换为 `bool`。

➢ **Public Base Classes**

`binary_function<T, T, bool>`

➢ **成员**

`logical_and::first_argument_type`

第一引数之型别：`T` （描述于 Adaptable Binary Predicate。）

`logical_and::second_argument_type`

第二引数之型别：`T` （描述于 Adaptable Binary Predicate。）

`logical_and::result_type`

执行结果之型别：`bool` （描述于 Adaptable Binary Predicate。）

`bool logical_and::operator()(const T& x, const T& y) const`

function call 操作符。其返回值为 `x && y`。（描述于 Binary Predicate。）

`logical_and::logical_and()`

default constructor。（描述于 Default Constructible。）

15.4.2 logical_or

`logical_or<T>`

Class `logical_or<T>` 是一种 Adaptable Binary Predicate，这表示它可用来验证某条件之真伪。如果 `f` 是 class `logical_or<T>` 的一个 object 而且 `x` 和 `y` 都是型别 `T` 之值，`T` 可转换为 `bool`，则只有当 `x` 或 `y` 有一个为 `true` 时 `f(x,y)` 才返回 `true`。

➢ 样例

找寻字符串中第一个 ' ' 字符或 '\n' 字符。

```
int main()
{
  char str[] = "The first line\nThe second line";
  int len = strlen(str);

  const char* wptr = find_if(str, str + len,
                             compose2(logical_or<bool>(),
                                      bind2nd(equal_to<char>(), ' '),
                                      bind2nd(equal_to<char>(), '\n')));
  assert(wptr == str + len || *wptr == ' ' || *wptr == '\n');
}
```

➢ 何处定义

HP STL 将 logical_or 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　logical_or 之引数型别。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）、Default Constructible（p.84）。

➢ 型别条件

- T 可转换为 bool。

➢ Public Base Classes

binary_function<T, T, bool>

➢ 成员

logical_or::first_argument_type

第一引数之型别：T （描述于 Adaptable Binary Predicate。）

logical_or::second_argument_type

第二引数之型别：T （描述于 Adaptable Binary Predicate。）

logical_or::result_type

执行结果之型别：bool （描述于 Adaptable Binary Predicate。）

bool logical_or::operator()(const T& x, const T& y) const

function call 操作符。其返回值为 x || y。（描述于 Binary Predicate。）

logical_or::logical_or()

default constructor。（描述于 Default Constructible。）

15.4.3 logical_not

logical_not<T>

Class logical_not<T> 是一种 Adaptable Predicate，这表示它可以用来验证某条件的真或伪，而且只需单一引数。如果 f 是 class logical_not<T> 的一个 object 而且 x 是型别 T 之值，T 可转换为 bool，则只有当 x 为 false 时 f(x) 才返回 true。

➢ 样例

将一个 bit vector 转换为其逻辑补数（logical complement）。

```
int main()
{
  const int N = 1000;

  vector<bool> v1;
  for (int i = 0; i < N; ++i)
    v1.push_back(rand() > (RAND_MAX / 2));

  vector<bool> v2;
  transform(v1.begin(), v1.end(), back_inserter(v2),
            logical_not<bool>());

  for (int i = 0; i < N; ++i)
    assert(v1[i] == !v2[i]);
}
```

➢ 何处定义

HP STL 将 logical_not 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

T　　logical_not 之引数型别。

➢ 隶属何种 concept

Adaptable Predicate（p.118）、Default Constructible（p.84）。

➢ **型别条件**

- T 可转换为 bool。

➢ **Public Base Classes**

unary_function<T, bool>

➢ **成员**

logical_not::argument_type

引数型别：T （描述于 Adaptable Predicate。）

logical_not::result_type

执行结果之型别：bool （描述于 Adaptable Predicate。）

bool logical_not::operator()(const T& x) const

function call 操作符。其返回值为 !x。 （描述于 Predicate。）

logical_not::logical_not()

default constructor。 （描述于 Default Constructible。）

15.5 证同（Identity）与投射（Projection）

本节所有的 function objects，以某种观点来看，都只是将其引数值原封不动地返回。基本的 identity function object 为 identity，其他 function objects 都是其一般化形式。

C++ standard 并未涵盖任何 identity 和 projection 操作行为，不过它们常常存在于各个实现品中，作为一种扩充性产品。

15.5.1 identity

identity<T>

Class identity 是一种 Unary Function，表示证同函数（identity function）。它需要一个引数 x，返回的是未经任何改变的 x。

➢ **样例**

```
int main()
{
  int x = 137;
  identity<int> id;
  assert(x == id(x));
}
```

➢ 何处定义

HP STL 将 identity 定义于头文件 <projectn.h>。SGI STL 将它定义于头文件 <functional>。C++ standard 并未涵盖这个 function object。

➢ Template 参数

T　　identity 之引数型别与返回值型别[44]。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）、Default Constructible（p.84）。

➢ Public Base Classes

unary_function<T, T>

➢ 成员

identity::argument_type

引数型别：T　（描述于 Adaptable Unary Function。）

identity::result_type

执行结果之型别：T　（描述于 Adaptable Unary Function。）

const T& identity::operator()(const T& x) const

function call 操作符。其返回值为 x。　（描述于 Adaptable Unary Function。）

identity::identity()

default constructor。　（描述于 Default Constructible。）

15.5.2 project1st

project1st<Arg1, Arg2>

Class project1st 接受两个引数，返回第一引数并忽略第二引数。本质上它是 Binary Function 情况下的 identity 一般化形式。

➢ 样例

```
int main()
{
  vector<int> v1(10, 137);
```

[44] 基本上引数与型别返回值型别是相同的。「将 identity 一般化，使两个型别得以不同」是没有用的，因为 identity 返回的是其引数的 const reference，而非其引数的复制品。如果 identity 能做到型别转换，则其返回值将会成为一个 dangling（空悬的）reference。

```
  vector<char*> v2(10, (char*) 0);
  vector<int> result(10);

  transform(v1.begin(), v1.end(), v2.begin(), result.begin(),
            project1st<int, char*>());
  assert(equal(v1.begin(), v1.end(), result.begin()));
}
```

➢ 何处定义

project1st class 并未出现于原始的 HP STL 中，也没有出现在 C++ standard 中。SGI STL 将定义于头文件 <functional>。

➢ Template 参数

Arg1　　project1st 之第一引数型别以及返回值型别。

Arg2　　project1st 之第二引数型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- Arg1 是 Assignable 的一个 model。

➢ Public Base Classes

binary_function<Arg1, Arg2, Arg1>

➢ 成员

project1st::first_argument_type

第一引数之型别：Arg1　（描述于 Adaptable Binary Function。）

project1st::second_argument_type

第二引数之型别：Arg2　（描述于 Adaptable Binary Function。）

project1st::result_type

执行结果之型别：Arg1　（描述于 Adaptable Binary Function。）

Arg1 project1st::operator()(const Arg1& x, const Arg2& y) const

function call 操作符。其返回值为 x。　（描述于 Adaptable Binary Function。）

project1st::project1st()

default constructor。　（描述于 Default Constructible。）

15.5.3 project2nd

project2nd<Arg1, Arg2>

Class project2nd 接受两个引数，返回第二引数并忽略第一引数。本质上它是 Binary Function 情况下的 identity 一般化形式。

➢ 样例

```
int main()
{
  vector<char*> v1(10, (char*) 0);
  vector<int> v2(10, 137);
  vector<int> result(10);

  transform(v1.begin(), v1.end(), v2.begin(), result.begin(),
            project2nd<char*, int>());
  assert(equal(v2.begin(), v2.end(), result.begin()));
}
```

➢ 何处定义

project2nd class 并未出现于原始的 HP STL 中，也没有出现在 C++ standard 中。SGI STL 将定义于头文件 <functional>。

➢ Template 参数

Arg1　　project2nd 之第一引数的型别。

Arg2　　project2nd 之第二引数型别以及返回值型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）、Default Constructible（p.84）。

➢ 型别条件

- Arg2 是 Assignable 的一个 model。

➢ Public Base Classes

binary_function<Arg1, Arg2, Arg2>

➢ 成员

project2nd::first_argument_type

第一引数之型别：Arg1　（描述于 Adaptable Binary Function。）

project2nd::second_argument_type

第二引数之型别：Arg2 （描述于 Adaptable Binary Function。）

project2nd::result_type

执行结果之型别：Arg2 （描述于 Adaptable Binary Function。）

Arg2 project2nd::operator()(const Arg1& x, const Arg2& y) const

function call 操作符。其返回值为 y。 （描述于 Adaptable Binary Function。）

project2nd::project2nd()

default constructor。 （描述于 Default Constructible。）

15.5.4 select1st

select1st<Pair>

Class select1st 接受单一引数，此引数是个 pair（或与 pair 相同接口的 class），并返回该 pair 的第一个元素。

➢ 样例

map 的元素都是 pairs，所以我们可以利用 select1st 取出及打印 map 的所有键值（keys）。

```
int main()
{
  map<int, double> M;
  M[1] = 0.3;
  M[47] = 0.8;
  M[33] = 0.1;

  transform(M.begin(), M.end(), ostream_iterator<int>(cout, " "),
            select1st<map<int, double>::value_type>());
  cout << endl;
  // 输出为 1 33 47。
}
```

➢ 何处定义

原始的 HP STL 将 select1st 定义于头文件 <projectn.h>。SGI STL 将它定义于头文件 <functional>。这个 class 并没有出现在 C++ standard 中。

➢ Template 参数

Pair 这个 function object 的引数型别。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）、Default Constructible（p.84）。

➢ 型别条件

- Pair 具有一个型别为 Pair::first_type 的 public member variable，Pair::first。

➢ Public Base Classes

unary_function<Pair, typename Pair::first_type>

➢ 成员

select1st::argument_type

引数型别：Pair （描述于 Adaptable Unary Function。）

select1st::result_type

执行结果之型别：Pair::first_type （描述于 Adaptable Unary Function。）

const typename Pair::first_type&select1st::operator()(const Pair& p) const

function call 操作符。其返回值为 p.first。 （描述于 Adaptable Unary Function。）

select1st::select1st()

default constructor。 （描述于 Default Constructible。）

15.5.5 select2nd

select2nd<Pair>

Class select2nd 接受单一引数，此引数是个 pair（或与 pair 相同接口的 class），并返回该 pair 的第二个元素。

➢ 样例

map 的元素都是 pairs，所以我们可以利用 select2nd 取出及打印 map 的所有实值（values）。

```
int main()
{
  map<int, double> M;
  M[1] = 0.3;
  M[47] = 0.8;
  M[33] = 0.1;

  transform(M.begin(), M.end(), ostream_iterator<double>(cout, " "),
            select2nd<map<int, double>::value_type>());
  cout << endl;
  // 输出为 0.3 0.1 0.8。
}
```

➢ 何处定义

原始的 HP STL 将 select2nd 定义于头文件 <projectn.h>。SGI STL 将它定义于头文件 <functional>。这个 class 并没有出现在 C++ standard 中。

➢ Template 参数

Pair　　这个 function object 的引数型别。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）、Default Constructible（p.84）。

➢ 型别条件

- Pair 具有一个型别为 Pair::second_type 的 public member variable Pair::second。

➢ Public Base Classes

unary_function<Pair, typename Pair::second_type>

➢ 成员

select2nd::argument_type

引数型别：Pair （描述于 Adaptable Unary Function。）

select2nd::result_type

执行结果之型别：Pair::second_type （描述于 Adaptable Unary Function。）

const typename Pair::second_type&select2nd::operator()(const Pair& p) const

function call 操作符。其返回值为 p.second。 （描述于 Adaptable Unary Function。）

select2nd::select2nd()

default constructor。 （描述于 Default Constructible。）

15.6 特殊的 Function Objects

15.6.1 hash

hash<T>

Class hash<T> 是一种 Hash Function。STL 内所有的 Hashed Associative Containers 把它拿来当作缺省的 hash function。

Template hash<T> 只针对 template 引数型别为 char*、const char*、string 以及内建整数(integral)而定义[45]。如果你要搭配不同引数型别之 Hash Function，你就必须提供自己的 template specialization，或是提供一个新的 Hash Function class。

➢ 样例

```
int main()
{
  hash<const char*> H;
  cout << "foo -> " << H("foo") << endl;
  cout << "bar -> " << H("bar") << endl;
}
```

➢ 何处定义

hash class 并未出现于原始的 HP STL 中，也没有出现在 C++ standard 中。然而它却是最常见的一份扩充。SGI STL 把它定义于头文件 <hash_set> 和 <hash_map>。

➢ Template 参数

T　　引数型别。也就是被 "hashed" 之值的型别。

➢ 隶属何种 concept

Hash Function（p.123）。

➢ 型别条件

型别 T 必须有一个对应的、已特化的 hash 版本。STL 已针对下列型别定义了对应的特化版本：

```
char*
const char*
string
char
signed char
unsigned char
short
unsigned short
int
unsigned int
long
unsigned long
```

[45] Tempate hash<T> 实际上是个空的 class。其 member function operator() 定义于不同的特化版本（specializations）之中。

➢ Public Base Classes

无

➢ **成员**

size_t hash::operator()(const T& x) cosnt

返回 x 的 hash 值。（描述于 Hash Function。）

15.6.2 subtractive_rng

subtractive_rng

Class subtractive_rng<T> 是一种 Random Number Generator，使用「减去法（subtractive method）」来产生拟真乱数（pseudo-random number）[46]。它是一种 Unary Function，需要一个型别为 unsignedint 的引数 N，返回一个小于 N 的无正负号整数。如果连续调用同一个 subtractive_rng object，会产生一个拟真乱数数列。

注意，subtractive_rng 产生出来的数列完全是可决定的（deterministic），由两个不同的 subtractive_rng objects 所产生出来的数列则互不相干。也就是说如果 R1 是一个 subtractive_rng，R1 产生的值只取决于 R1 的乱数种子（seed）以及 R1 先前被调用的次数。对其他 subtractive_rng objects 的调用将毫不相干。以实现方面的术语来说，这是因为 class subtractive_rng 并不包含任何 static member variables。

➢ **样例**

```
int main()
{
  subtractive_rng R;
  for (int i = 0; i < 20; ++i)
    cout << R(5) << ' ';
  cout << endl;
}
// 输出为 3 2 3 2 4 3 1 1 2 2 0 3 4 4 4 4 2 1 0 0
```

➢ **何处定义**

subtractive_rng 并非 C++ standard 的一部分。SGI STL 将它定义于头文件 <functional>。

[46] Knuth 书籍 [Knu98a] 3.6 节曾以 FORTRAN 实现 subtractive method，其 3.2.2 节并分析此一算法的种类。

➢ **Template 参数**

无

➢ **隶属何种 concept**

Random Number Generator（p.122）、Adaptable Unary Function（p.114）。

➢ **Public Base Classes**

`unary_function<unsigned int, unsigned int>`

➢ **成员**

`subtractive_rng` 的某些成员并未定义于 Random Number Generator 或 Adaptable Unary Function 的需求条件中，这些成员是 `subtractive_rng` 所特有。这些特别的成员以 ※ 符号标示之。

`subtractive_rng::argument_type`

引数型别：`unsigned int`（描述于 Adaptable Unary Function。）

`subtractive_rng::result_type`

执行结果的型别：`unsigned int`（描述于 Adaptable Unary Function。）

※ **`subtractive_rng::subtractive_rng(unsigned int seed)`**

constructor。产生一个 `subtractive_rng` 并以 `seed` 初始化其内部状态。

※ **`subtractive_rng::subtractive_rng()`**

default constructor。产生一个 `subtractive_rng` 并以缺省的 seed 初始化其内部状态。

`unsigned int subtractive_rng::operator()(unsigned int n)`

function call 操作符。返回区间 `[0,N)` 内的一个拟真乱数。

※ **`void subtractive_rng::initialize(unsigned int seed)`**

重新设定乱数产生器，使其内部状态恰与「以 `seed` 值构造之」的方式相同。

15.7 Member Function Adapters

所谓 ***member function adapters*** 是一群小型的 classes，让你能够将 member functions 当作 function objects 来调用。每一个 adapter 需要一个型别为 `X*` 或 `X&` 的引数，可通过该引数调用 `X` 的一个 member function — 如果那是一个 virtual member function，那么便是一个多态函数调用（polymorphic function call）。因此，member function adapters 是面向对象编程与泛型编程之间的桥梁。

Member function adapters 数量之多令人却步，但它们都遵循系统化的命名格式。三个不同的特点形成了共 $2^3 = 8$ 种 adapters：

1. 这个 adapter 接受型别为 x* 或 x& 的引数吗？如果是后者，其名称有 "_ref" 后缀词。
2. 这个 adapter 封装的是无引数的或单一引数的 member function？如果是后者，其名称有 '1' 后缀词。
3. 这个 adapter 封装的是 non-const 或 const member function？（C++ 的型别声明语法无法以同一个 class 处理两种情况。）如果是后者，其名称会冠上 "const_" 前缀词。

以上的复杂度在一般使用情况下多隐匿无形。通常构造一个 function object adapter 的方式是使用辅助函数（helper function）而非直接调用其 constructor。由于辅助函数被重载（overloaded），所以只需记住两个名称就好：mem_fun 和 mem_fun_ref。

15.7.1 mem_fun_t

mem_fun_t<R, X>

Class mem_fun_t 是一种 member function adapter。如果 X 是个 class，具有 member function RX::f()（亦即无任何引数，返回型别为 R[47]），那么 mem_fun_t<R, X> 便成为一个 function object adapter，使得以一般函数（而非 member function）的方式来调用 f()。

Constructor 需要一个指针，指向 X 的一个 member function。和所有的 function objects 一样，mem_fun_t 拥有一个 operator()，使 mem_fun_t 得以一般的函数调用语法被调用。在这种情况下，mem_fun_t 的 operator() 接受一个型别为 X* 的引数。

如果 F 是一个 mem_fun_t，以 member function X::f 构造出来，并且 x 是一个指针，型别为 X*，那么表达式 F(x) 等价于表达式 x->f()。两者的差别在于语法层面：F 支持 Adaptable Unary Function 的接口。

如同其他众多 adapters 一样，直接使用 mem_fun_t constructor 是很不方便的。使用辅助函数 mem_fun 来代替往往是比较好的做法。

➢ 样例

```
struct B {
  virtual void print() = 0;
};

struct D1 : public B {
  void print() { cout << "I'm a D1" << endl; }
};
```

[47] 型别 R 可以是 void。

```
struct D2 : public B {
  void print() { cout << "I'm a D2" << endl; }
};

int main()
{
  vector<B*> V;

  V.push_back(new D1);
  V.push_back(new D2);
  V.push_back(new D2);
  V.push_back(new D1);

  for_each(V.begin(), V.end(), mem_fun(&B::print));
}
```

➢ 何处定义

HP STL 并不包含 mem_fun_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　mem_fun_t 所调用之 member function 的所属 class。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 non-const member function，此 member function 不需引数且其返回值型别为 R。

➢ Public Base Classes

unary_function<X*, R>

➢ 成员

mem_fun_t 的某些成员并未定义于 Adaptable Unary Function 的需求条件中，它们是 mem_fun_t 所特有，皆以 ※ 符号标示之。

mem_fun_t::argument_type

引数型别：X* （描述于 Adaptable Unary Function。）

mem_fun_t::result_type

执行结果的型别：R （描述于 Adaptable Unary Function。）

R mem_fun_t::operator()(X* x) const

function call 操作符。它会调用 x->f()，f 为其 constructor 所接受之 member function。（描述于 Adaptable Unary Function。）

※ **explicit mem_fun_t::mem_fun_t(R (X::*f)())**

Constructor。产生一个 mem_fun_t，得以藉它调用 member function f。

※ **template <class R, class X> mem_fun_t<R, X> mem_fun(R (X::*f)())**

一个辅助函数，用来产生 mem_fun_t。如果 f 的型别为 R (X::*)()，则 mem_fun(f) 与 mem_fun_t<R, X>(f) 相同。

15.7.2 mem_fun_ref_t

mem_fun_ref_t<R, X>

Class mem_fun_ref_t 是一种 member function adapter。如果 X 是个 class，具有 member function R X::f()（亦即无引数，返回型别为 R[48]），那么 mem_fun_ref_t<R, X> 为一个 function object adapter，使得以一般函数（而非 member function）的方式来调用 f()。

Constructor 需要一个指针，指向 X 的一个 member function。和所有的 function objects 一样，mem_fun_ref_t 拥有一个 operator()，使得以一般的函数调用语法被调用。在这种情况下，mem_fun_ref_t 的 operator() 接受一个型别为 X& 的引数。

如果 F 是一个 mem_fun_ref_t，以 member function X::f 构造出来，并且 x 是一个型别为 X 的 object，那么表达式 F(x) 等价于表达式 x.f()。两者的差别在于语法层面：F 支持 Adaptable Unary Function 的接口。

如同其他众多 adapters 一样，直接使用 mem_fun_ref_t constructor 是很不方便的。使用辅助函数 mem_fun_ref 来代替往往是比较好的做法。

➢ **样例**

```
struct B {
  virtual void print() = 0;
};
```

[48] 型别 R 可以是 void。

```
struct D1 : public B {
  void print() { cout << "I'm a D1" << endl; }
};

struct D2 : public B {
  void print() { cout << "I'm a D2" << endl; }
};

int main()
{
  vector<D1> V;

  V.push_back(D1());
  V.push_back(D1());

  for_each(V.begin(), V.end(), mem_fun_ref(&B::print));
}
```

➢ 何处定义

HP STL 并不包含 mem_fun_ref_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　mem_fun_ref_t 调用之 member function 的所属 class。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 nono-const member function，此 member function 不需引数，返回值型别为 R。

➢ Public Base Classes

unary_function<X, R>

➢ 成员

mem_fun_ref_t 的某些成员并未定义于 Adaptable Unary Function 的需求条件之中，这些成员是 mem_fun_ref_t 所特有，皆以 ※ 符号标示之。

mem_fun_ref_t::argument_type

引数型别：X （描述于 Adaptable Unary Function。）

mem_fun_ref_t::result_type

执行结果的型别：R （描述于 Adaptable Unary Function。）

R mem_fun_ref_t::operator()(X& x) const

function call 操作符。它会调用 x.f()，f 为 constructor 所接受之 member function。（描述于 Adaptable Unary Function。）

※ **explicit mem_fun_ref_t::mem_fun_ref_t(R (X::*f)())**

constructor。产生一个 mem_fun_ref_t，能够藉此调用 member function f。

※ **template <class R, class X> mem_fun_ref_t<R, X> mem_fun_ref(R (X::*f)())**

一个辅助函数，用来产生 mem_fun_ref_t。如果 f 的型别为 R (X::*)()，则 mem_fun_ref(f) 与 mem_fun_ref_t<R, X>(f) 相同。

15.7.3 mem_fun1_t

mem_fun1_t<R, X, A>

Class mem_fun1_t 是一种 member function adapter。如果 X 是个 class，具有 member function RX::f(A)（亦即引数型别为 A，返回型别为 R[49]），那么 mem_fun1_t<R, X, A> 是一个 function object adapter，使得以一般函数（而非 member function）的方式来调用 f。

constructor 需要一个指针，指向 X 的一个 member function。就像所有的 function objects 一样，mem_fun1_t 具有一个 operator()，使 mem_fun1_t 得以一般的函数调用语法被调用。这种情况下 mem_fun1_t 的 operator() 需要两个引数，第一引数的型别为 X*，第二引数的型别为 A。

如果 F 是一个 mem_fun1_t，并以 member function X::f 构造出来，x 是个指针，型别为 X*，a 是型别 A 的值，那么表达式 F(x, a) 等价于表达式 x->f(a)。两者差别在于语法层面上：F 支持 Adaptable Binary Function 的接口。

和其他众多 adapters 一样，直接使用 mem_fun1_t 的 constructor 并不方便。以辅助函数 mem_fun 代替之通常是比较好的做法。

➢ **样例**

```
struct Operation {
  virtual double eval(double) = 0;
};

struct Square : public Operation {
  double eval(double x) { return x * x; }
};
```

[49] 型别 R 可以是 void。

```
struct Negate : public Operation {
  double eval(double x) { return -x; }
};

int main()
{
  vector<Operation*> operations;
  vector<double> operands;

  operations.push_back(new Square);
  operations.push_back(new Square);
  operations.push_back(new Negate);
  operations.push_back(new Negate);
  operations.push_back(new Square);

  operands.push_back(1);
  operands.push_back(2);
  operands.push_back(3);
  operands.push_back(4);
  operands.push_back(5);

  transform(operations.begin(), operations.end(),
            operands.begin(),
            ostream_iterator<double>(cout, "\n"),
            mem_fun(&Operation::eval));
}
```

➢ 何处定义

HP STL 并不包含 `mem_fun1_t`。C++ standard 将它声明于头文件 `<functional>`。

➢ Template 参数

R	member function 的返回型别。
X	`mem_fun1_t` 所调用之 member function 的所属 class。
A	member function 的引数型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 non-const member function，此 member function 需要一个型别为 A 的引数，返回值型别为 R。
- A 为 Assignable。

➢ **Public Base Classes**

binary_function<X*, A, R>

➢ **成员**

mem_fun1_t 的某些成员并未定义于 Adaptable Binary Function 的需求条件中，这些成员是 mem_fun1_t 所特有，皆以 ※ 符号标示之。

mem_fun1_t::first_argument_type

第一引数型别：X* （描述于 Adaptable Binary Function。）

mem_fun1_t::second_argument_type

第二引数型别：A （描述于 Adaptable Binary Function。）

mem_fun1_t::result_type

执行结果的型别：R （描述于 Adaptable Binary Function。）

R mem_fun1_t::operator()(X* x, A a) const

function call 操作符。它会调用 x->f(a)，f 为其 constructor 所接受之 member function。（描述于 Adaptable Binary Function。）

※ **explicit mem_fun1_t::mem_fun1_t(R (X::*f)(A))**

constructor。产生一个 mem_fun1_t，能够藉此调用 member function f。

※ **template <class R, class X, class A> mem_fun1_t<R, X, A> mem_fun(R (X::*f)(A))**

一个辅助函数，用来产生 mem_fun1_t。如果 f 的型别为 R (X::*)(A)，则 mem_fun(f) 与 mem_fun1_t<R, X, A>(f) 相同。

15.7.4 mem_fun1_ref_t

mem_fun1_ref_t<R, X, A>

Class mem_fun1_ref_t 是一种 member function adapter。如果 X 是个 class，具有 member function R X::f(A)（亦即引数型别为 A，返回型别为 R[50]），那么 mem_fun1_ref_t<R, X, A> 是一个 function object adapter，使得以一般函数（而非 member function）的方式来调用 f。

constructor 需要一个指针，指向 X 的一个 member function。如同所有 function objects 一样，mem_fun1_ref_t 具有一个 operator()，使 mem_fun1_ref_t 得以一般的函数调用语法被调用。这种情况下 mem_fun1_ref_t 的 operator() 需要两个引数。第一引数的型别为 X&，第二引数的型别为 A。

---

[50] 型别 R 可以是 void。

如果 F 是一个 mem_fun1_ref_t，并以 member function X::f 构造出来，x 是型别为 X 的 object，a 是型别为 A 的值，则表达式 F(x, a) 等价于表达式 x->f(a)。两者差别在于语法层面：F 支持 Adaptable Binary Function 的接口。

和其他众多 adapters 一样，直接使用 mem_fun1_ref_t 的 constructor 是很不方便的。以辅助函数 mem_fun_ref 代替通常是比较好的做法。

➢ 样例

```
int main() {
  int A1[5] = {1, 2, 3, 4, 5};
  int A2[5] = {1, 1, 2, 3, 5};
  int A3[5] = {1, 4, 1, 5, 9};

  vector<vector<int> > V;
  V.push_back(vector<int>(A1, A1 + 5));
  V.push_back(vector<int>(A2, A2 + 5));
  V.push_back(vector<int>(A3, A3 + 5));

  int indices[3] = {0, 2, 4};

  int& (vector<int>::*extract)(vector<int>::size_type);
  extract = &vector<int>::operator[];
  transform(V.begin(), V.end(), indices,
          ostream_iterator<int>(cout, " "),
          mem_fun_ref(extract));
  cout << endl;
}
```

输出为：

```
1 2 9
```

➢ 何处定义

HP STL 并不包含 mem_fun1_ref_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　mem_fun1_ref_t 调用之 member function 的所属 class。

A　　member function 的引数型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 non-const member function，此 member function 需要一个型别为 A 的引数，返回值型别为 R。
- A 为 Assignable。

➢ Public Base Classes

binary_function<X, A, R>

➢ 成员

mem_fun1_ref_t 的某些成员并未定义于 Adaptable Binary Function 的需求条件中，这些成员是 mem_fun1_ref_t 所特有，皆以 ※ 符号标示之。

mem_fun1_ref_t::first_argument_type

第一引数型别：X （描述于 Adaptable Binary Function。）

mem_fun1_ref_t::second_argument_type

第二引数型别：A （描述于 Adaptable Binary Function。）

mem_fun1_ref_t::result_type

执行结果的型别：R （描述于 Adaptable Binary Function。）

R mem_fun1_ref_t::operator()(X& x, A a) const

function call 操作符。它会调用 x.f(a)，f 为其 constructor 所接受的 member function。（描述于 Adaptable Binary Function。）

※ **explicit mem_fun1_ref_t::mem_fun1_ref_t(R (X::*f)(A))**

constructor。产生一个 mem_fun1_ref_t，能藉此调用 member function f。

※ **template <class R, class X, class A>**
mem_fun1_ref_t<R, X, A> mem_fun_ref(R (X::*f)(A))

一个辅助函数，用来产生 mem_fun1_ref_t。如果 f 的型别为 R(X::*)(A)，则 mem_fun_ref(f) 与 mem_fun1_ref_t<R, X, A>(f) 相同。

15.7.5 const_mem_fun_t

const_mem_fun_t<R, X>

Class const_mem_fun_t 是一种 member function adapter。如果 X 是个 class，具有 member function R X::f() const（亦即无引数、返回型别为 R[51]），则 const_mem_fun_t<R, X> 为一个 function object

[51] 型别 R 可以是 void。

adapter，使得以一般函数（而非 member function）的方式调用 f()。

constructor 需要一个指针，指向 X 的一个 member function。如同所有的 function objects 一样，const_mem_fun_t 具有一个 operator()，使 const_mem_fun_t 得以一般函数调用语法被调用。在这种情况下，const_mem_fun_t 的 operator() 需要一个型别为 const X* 的引数。

如果 F 是一个 const_mem_fun_t，并且是以 member function X::f 构造出来，x 是个指针，型别为 const X*，则表达式 F(x) 等价于表达式 x->f()。两者差别在于语法层面：F 支持 Adaptable Unary Function 的接口。

如同其他很多的 adapters 一样，直接使用 const_mem_fun_t 的 constructor 是很不方便的。以辅助函数 mem_fun 代替通常是比较好的做法。

➢ 样例

已知一个 vector，其每个元素亦都是一个 vector。试打印每个元素的大小。

```
int main()
{
  vector<vector<int>*> v;
  v.push_back(new vector<int>(5));
  v.push_back(new vector<int>(3));
  v.push_back(new vector<int>(4));

  transform(v.begin(), v.end(),
            ostream_iterator<int>(cout, " "),
            mem_fun(&vector<int>::size));
  cout << endl;
  // 输出为 "5 3 4"
}
```

➢ 何处定义

HP STL 并不包含 const_mem_fun_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　const_mem_fun_t 调用之 member function 的所属 class。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 const member function，此 member function 不需引数，返回值型别为 R。

➢ Public Base Classes

unary_function<const X*, R>

➢ 成员

const_mem_fun_t 的某些成员并未定义于 Adaptable Unary Function 的需求条件中，这些成员是 const_mem_fun_t 所特有，皆以 ※ 符号标示之。

const_mem_fun_t::argument_type

引数型别：const X* （描述于 Adaptable Unary Function。）

const_mem_fun_t::result_type

执行结果的型别：R （描述于 Adaptable Unary Function。）

R const_mem_fun_t::operator()(const X* x) const

function call 操作符。它会调用 x->f()，其中 f 为 constructor 所接受的 member function。（描述于 Adaptable Unary Function。）

※ **explicit const_mem_fun_t::const_mem_fun_t(R (X::*f)() const)**

constructor。产生一个 const_mem_fun_t，能藉此调用 member function f。

※ **template <class R, class X> const_mem_fun_t<R, X> mem_fun(R (X::*f)() const)**

一个辅助函数，用来产生 const_mem_fun_t。如果 f 的型别为 R(X::*)()const，则 mem_fun(f) 与 const_mem_fun_t<R, X>(f) 相同。

15.7.6 const_mem_fun_ref_t

const_mem_fun_ref_t<R, X>

Class const_mem_fun_ref_t 是一种 member function adapter。如果 X 是个 class，具有 member function R X::f() const（亦即无引数、返回型别为 R[52]），则 const_mem_fun_ref_t<R, X> 为一个 function object adapter，使得以一般函数（而非 member function）的方式调用 f()。

---

[52] 型别 R 可以是 void。

const_mem_fun_ref_t<R, X> 的 constructor 需要一个指针，指向 X 的一个 member function。如同所有的 function objects，const_mem_fun_ref_t adapter 具有一个 operator()，使 const_mem_fun_ref_t 能以一般的函数调用语法被调用。在这种情况下，const_mem_fun_ref_t 的 operator() 需要一个型别为 const X& 的引数。

如果 F 是一个 const_mem_fun_ref_t，并且是以 member function X::f 构造出来，x 是一个型别为 X 的 object，则表达式 F(x) 等价于表达式 x.f()。两者差别在于语法层面：F 支持 Adaptable Unary Function 的接口。

和其他众多的 adapters 一样，直接使用 const_mem_fun_ref_t 的 constructor 是很不方便的。使用辅助函数 mem_fun_ref 代替通常是比较好的做法。

➢ 样例

```
int main()
{
  vector<vector<int> > v;
  v.push_back(vector<int>(2));
  v.push_back(vector<int>(7));
  v.push_back(vector<int>(3));

  transform(v.begin(), v.end(),
            ostream_iterator<int>(cout, " "),
            mem_fun_ref(&vector<int>::size));
  cout << endl;
  // 输出为 "2 7 3"
}
```

➢ 何处定义

HP STL 并不包含 const_mem_fun_ref_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　const_mem_fun_ref_t 调用之 member function 的所属 class。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 const member function，此 member function 不需引数，返回值型别为 R。

➢ **Public Base Classes**

unary_function<X, R>

➢ **成员**

const_mem_fun_ref_t 的某些成员并未定义于 Adaptable Unary Function 需求条件中，这些成员是 const_mem_fun_ref_t 所特有，皆以 ※ 符号标示之。

const_mem_fun_ref_t::argument_type

引数型别：X （描述于 Adaptable Unary Function。）

const_mem_fun_ref_t::result_type

执行结果的型别：R （描述于 Adaptable Unary Function。）

R const_mem_fun_ref_t::operator()(const X& x) const

function call 操作符。它会调用 x.f()，f 为其 constructor 所接受的 member function。（描述于 Adaptable Unary Function。）

※ **explicit const_mem_fun_ref_t::const_mem_fun_ref_t(R (X::*f)() const)**

constructor。产生一个 const_mem_fun_ref_t，能藉此调用 member function f。

※ **template <class R, class X>**
const_mem_fun_ref_t<R, X> mem_fun_ref(R (X::*f)() const)

一个辅助函数，用来产生 const_mem_fun_ref_t。如果 f 的型别为 R (X::*)() const，则 mem_fun_ref(f) 与 const_mem_fun_ref_t<R, X>(f) 相同。

15.7.7 const_mem_fun1_t

const_mem_fun1_t<R, X, A>

Class const_mem_fun1_t 是一种 member function adapter。如果 X 是个 class，具有 member function R X::f(A) const（亦即引数型别为 A、返回型别为 R[53]），则 const_mem_fun1_t<R, X, A> 为一个 function object adapter，使得以一般函数（而非 member function）的方式调用 f。

constructor 需要一个指针，指向 X 的一个 member function。如同所有的 function objects 一样，const_mem_fun1_t 具有一个 operator()，使 const_mem_fun1_t 得以一般的函数调用语法被调用。这种情况下 const_mem_fun1_t 的 operator() 需要两个引数。第一引数的型别为 const X*，第二引数的型别为 A。

如果 F 是一个 const_mem_fun1_t，并且是以 member function X::f 构造出来，x 是一个指针，型

[53] 型别 R 可以是 void。

别为 const X*，a 是型别 A 的值，则表达式 F(x, a) 等价于表达式 x->f(a)。两者差别在于语法层面：F 支持 Adaptable Binary Function 的接口。

如同其他很多的 adapters，直接使用 const_mem_fun1_t 的 constructor 是很不方便的。使用辅助函数 mem_fun 来代替通常是比较好的做法。

➢ 样例

```
struct B {
  virtual int f(int x) const = 0;
};

struct D : public B {
  int val;

  D(int x) : val(x) {}
  int f(int x) const { return val + x; }
};

int main()
{
  vector<B*> v;

  v.push_back(new D(3));
  v.push_back(new D(4));
  v.push_back(new D(5));

  int A[3] = {7, 8, 9};
  transform(v.begin(), v.end(), A, ostream_iterator<int>(cout, " "),
            mem_fun(&B::f));
  cout << endl;
  // 输出为 "10 12 14"
}
```

➢ 何处定义

HP STL 并不包含 const_mem_fun1_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　const_mem_fun1_t 调用之 member function 的所属 class。

A　　member function 的引数型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 const member function，此 member function 需要一个型别为 A 的引数，返回值型别为 R。
- A 为 Assignable。

➢ Public Base Classes

binary_function<const X*, A, R>

➢ 成员

const_mem_fun1_t 的某些成员并未定义于 Adaptable Binary Function 需求条件中，这些成员是 const_mem_fun1_t 所特有，皆以 ※ 符号标示之。

const_mem_fun1_t::first_argument_type

第一引数型别：const X* （描述于 Adaptable Binary Function。）

const_mem_fun1_t::second_argument_type

第二引数型别：A （描述于 Adaptable Binary Function。）

const_mem_fun1_t::result_type

执行结果的型别：R （描述于 Adaptable Binary Function。）

R const_mem_fun1_t::operator()(const X* x, A a) const

function call 操作符。它会调用 x->f(a)，f 为其 constructor 所接受的 member function。（描述于 Adaptable Binary Function。）

※ **explicit const_mem_fun1_t::const_mem_fun1_t(R (X::*f)(A) const)**

constructor。产生一个 const_mem_fun1_t，可藉此调用 member function f。

※ **template <class R, class X, class A>**
const_mem_fun1_t<R, X, A> mem_fun(R (X::*f)(A) const)

一个辅助函数，用来产生 const_mem_fun1_t。如果 f 的型别为 R(X::*)(A)const，则 mem_fun(f) 与 const_mem_fun1_t<R, X, A>(f) 相同。

15.7.8 const_mem_fun1_ref_t

const_mem_fun1_ref_t<R, X, A>

Class const_mem_fun1_ref_t 是一种 member function adapter。如果 X 是个 class，具有 member function RX::f(A)const（亦即单一引数型别为 A，返回型别为 R[54]），则 const_mem_fun1_ref_t<R,

[54] 型别 R 可以是 void。

X, A> 为一个 function object adapter，使得以一般函数（而非 member function）的方式调用 f。

constructor 需要一个指针，指向 X 的一个 member function。如同所有的 function objects 一样，const_mem_fun1_ref_t 具有一个 operator()，使 const_mem_fun1_ref_t 得以一般的函数调用语法被调用。这种情况下 const_mem_fun1_ref_t 的 operator() 需要两个引数。第一引数的型别为 const X&，第二引数的型别为 A。

如果 F 是一个 const_mem_fun1_ref_t，并且是以 member function X::f 构造出来，x 是型别 X 的一个 object，a 是型别 A 的一个值，则表达式 F(x, a) 等价于表达式 x.f(a)。两者差别在于语法层面：F 支持 Adaptable Binary Function 的接口。

如同其他很多的 adapters，直接使用 const_mem_fun1_ref_t 的 constructor 是很不方便的。使用辅助函数 mem_fun_ref 代替之通常是比较好的做法。

➢ 样例

```
struct B {
  virtual int f(int x) const = 0;
};

struct D : public B {
  int val;

  D(int x) :  val(x) {}
  int f(int x) const { return val + x; }
};

int main()
{
  vector<D> v;

  v.push_back(D(3));
  v.push_back(D(4));
  v.push_back(D(5));

  int A[3] = {7, 8, 9};
  transform(v.begin(), v.end(), A,
            ostream_iterator<int>(cout, " "),
            mem_fun_ref(&B::f));
  cout << endl;
  // 输出为 "10 12 14"
}
```

➢ 何处定义

HP STL 并不包含 const_mem_fun1_ref_t。C++ standard 将它声明于头文件 <functional>。

➢ Template 参数

R　　member function 的返回型别。

X　　const_mem_fun1_ref_t 调用之 member function 的所属 class。

A　　member function 的引数型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- R 为 Assignable 或 void。
- X 至少具有一个 const member function，此 member function 需要一个型别为 A 的引数，返回值型别为 R。
- A 为 Assignable。

➢ Public Base Classes

binary_function<X, A, R>

➢ 成员

const_mem_fun1_ref_t 的某些成员并未定义于 Adaptable Binary Function 需求条件中，这些成员是 const_mem_fun1_ref_t 所特有，皆以 ※ 符号标示之。

const_mem_fun1_ref_t::first_argument_type

第一引数型别：X （描述于 Adaptable Binary Function。）

const_mem_fun1_ref_t::second_argument_type

第二引数型别：A （描述于 Adaptable Binary Function。）

const_mem_fun1_ref_t::result_type

执行结果的型别：R （描述于 Adaptable Binary Function。）

R const_mem_fun1_ref_t::operator()(const X& x, A a) const

function call 操作符。它会调用 x.f(a)，f 为其 constructor 所接受的 member function。（描述于 Adaptable Binary Function。）

※ **explicit const_mem_fun1_ref_t::const_mem_fun1_ref_t(R (X::*f)(A) const)**

constructor。产生一个 const_mem_fun1_ref_t，能藉此调用 member function f。

※ **template <class R, class X, class A>**
const_mem_fun1_ref_t<R, X, A> mem_fun(R (X::*f)(A) const)

一个辅助函数，用来产生 const_mem_fun1_ref_t。如果 f 的型别是 R (X::*)(A) const，则 mem_fun_ref(f) 与 const_mem_fun1_ref_t<R, X, A>(f) 相同。

15.8 其他的 Adapters

15.8.1 binder1st

binder1st<BinaryFun>

Class binder1st 是一种 function object adapter。可用来将 Adaptable Binary Function 转换成 Adaptable Unary Function。如果 f 是 class binder1st<BinaryFun> 的 object，则 f(x) 返回 F(c, x)，其中 F 为 BinaryFun object，c 为一常量。F 和 c 都会被当作引数，传给 binder1st 的 constructor。

直观上你可以将此动作想成是将一个双参函数的第一引数设定为某个常量，因而形成一个单参函数。

要产生一个 binder1st，最简单的方式不是直接调用其 constructor，而是使用辅助函数 bind1st。

➢ 样例

在链表之中寻找第一个不为零的元素。

```
int main()
{
  list<int> L;
  for (int i = 0; i < 20; ++i)
    L.push_back(rand() % 3);

  list<int>::iterator first_nonzero =
        find_if(L.begin(), L.end(),
                bind1st(not_equal_to<int>(), 0));
  assert(first_nonzero == L.end() || *first_nonzero != 0);
}
```

➢ 何处定义

HP STL 将 binder1st 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

BinaryFun　　双参函数的型别，其第一引数将与某个常量系结在一起。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- BinaryFun 必须为 Adaptable Binary Function 的一个 model。

➢ Public Base Classes

```
unary_function<typename BinaryFun::second_argument_type,
               typename BinaryFun::result_type>
```

➢ **成员**

binder1st 的某些成员并未定义于 Adaptable Unary Function 需求条件中，这些成员是 binder1st 所特有，皆以 ※ 符号标示之。

binder1st::argument_type

引数型别：BinaryFun::second_argument_type（描述于 Adaptable Unary Function。）

binder1st::result_type

执行结果的型别：BinaryFun::result_type （描述于 Adaptable Unary Function。）

result_type binder1st::operator()(const argument_type& x) const

function call 操作符。返回 F(c, x)，其中 F 和 c 是 binder1st object 构造时的引数。（描述于 Adaptable Unary Function。）

※ **binder1st::binder1st(const BinaryFun& F,**
typename BinaryFun::first_argument_type c)

constructor。产生一个 binder1st，致使当我们调用它并带引数 x(其型别为 BinaryFun::second_argument_type）时，就像调用 F(c, x) 一样。

※ **template <class BinaryFun, class T>**
binder1st<BinaryFun> bind1st(const BinaryFun& F, const T& c)

辅助函数，用来产生一个 binder1st object。如果 F 的型别为 BinaryFun，则 bind1st(F,c) 不但等价于 binder1st<BinaryFun>(F,c)，而且更为方便。型别 T 必须可以转换为 BinaryFun 的第一引数型别。

15.8.2 binder2nd

binder2nd<BinaryFun>

Class binder2nd 是一种 function object adapter。可用来将 Adaptable Binary Function 转换成 Adaptable Unary Function。如果 f 是 class binder2nd<BinaryFun> 的 object，则 f(x) 返回 F(x,c)，其中 F 为 BinaryFun object，c 为一常量。F 和 c 都会被当作引数，传给 binder2nd 的 constructor。

直观上你可以将此动作想成是将一个双参函数的第二引数设定为某个常量，因而形成一个单参函数。

要产生一个 binder2nd，最简单的方式不是直接调用其 constructor，而是使用辅助函数 bind2nd。

➢ **样例**

在链表之中寻找第一个正数。

```
int main()
{
  list<int> L;
  for (int i = 0; i < 20; ++i)
    L.push_back(rand() % 4 - 3);

  list<int>::iterator first_positive =
        find_if(L.begin(), L.end(),
                bind2nd(greater<int>(), 0));
  assert(first_positive == L.end() || *first_positive > 0);
}
```

➢ 何处定义

HP STL 将 binder2nd 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

BinaryFun　　双参函数的型别，其第二引数将与某个常量系结在一起。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- BinaryFun 必须为 Adaptable Binary Function 的一个 model。

➢ Public Base Classes

```
unary_function<typename BinaryFun::first_argument_type,
               typename BinaryFun::result_type>
```

➢ 成员

binder2nd 的某些成员并未定义于 Adaptable Unary Function 的需求条件中，这些成员是 binder2nd 所特有，皆以 ※ 符号标示之。

binder2nd::argument_type

引数型别：BinaryFun::first_argument_type （描述于 Adaptable Unary Function。）

binder2nd::result_type

执行结果的型别：BinaryFun::result_type （描述于 Adaptable Unary Function。）

result_type binder2nd::operator()(const argument_type& x) const

function call 操作符。返回 F(x, c)，其中 F 和 c 是 binder2nd object 构造时的引数。（描述于 *Adaptable Unary Function*。）

※ **binder2nd::binder2nd(const BinaryFun& F,**
typename BinaryFun::second_argument_type c)

constructor。产生一个 binder2nd，致使当我们调用它并带引数 x（其型别为 BinaryFun::first_argument_type）时，就像调用 F(x, c) 一样。

※ **template <class BinaryFun, class T>**
binder2nd<BinaryFun> bind2nd(const BinaryFun& F, const T& c)

辅助函数，用来产生一个 binder2nd object。如果 F 的型别为 BinaryFun，则 bind2nd(F,c) 不但等价于 binder2nd<BinaryFun>(F,c)，而且更为方便。型别 T 必须可以转换为 BinaryFun 的第二引数型别。

15.8.3 pointer_to_unary_function

pointer_to_unary_function<Arg, Result>

Class pointer_to_unary_function 是一种 function object adapter，允许将函数指针 Result (*f)(Arg) 视为一个 Adaptable Unary Function。如果 F 是一个 pointer_to_unary_function<Arg, Result> object，而且以一个型别为 Result (*)(Arg) 的函数指针 f 作为初值，那么 F(x) 会调用函数 f(x)。f 与 F 之间的差别在于 pointer_to_unary_function 是一个 Adaptable Unary Function；也就是说它定义有嵌套型别 argument_type 和 result_type。

型别为 Result (*)(Arg) 的那个函数指针，就自身条件而言已经是一个相当不错的 Unary Function object，它可以传入任何「需要以 Unary Function 为引数」的 STL 算法中。pointer_to_unary_function 的唯一使用时机，是当你希望在「需要 Adaptable Unary Function」的情况下使用一般函数指针（例如，以它作为某个 function object adapter 的引数）。

大多数情况下，你不应该直接使用 pointer_to_unary_function 的 constructor。使用辅助函数 ptr_fun 几乎总是比较简单的方式。

➢ 样例

以标准函数库中的 fabs，将某个 range 内的数值以其绝对值取代。这不需要用到 pointer_to_unary_function adapter，因为我们只要将 fabs 直接传给 transform 即可：

```
transform(first, last, first, fabs);
```

底下这段代码则是将某个 range 内的数值以其绝对值的负数取代。我们需要将 fabs 与 negate 组合起来，因此 fabs 必须被视为一个 Adaptable Unary Function，此时我们必须使用 pointer_to_unary_function：

```
transform(first, last, first, compose1(negate<double>(), ptr_fun(fabs)));
```

➢ 何处定义

HP STL 将 pointer_to_unary_function 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

Arg　　此 function object 的引数型别。

Result　　此 function object 执行结果的型别。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- Arg 为 Assignable。
- Result 为 Assignable。

➢ Public Base Classes

unary_function<Arg, Result>

➢ 成员

pointer_to_unary_function 的某些成员并未定义于 Adaptable Unary Function 需求条件中，这些成员是 pointer_to_unary_function 所特有，皆以 ※ 符号标示之。

pointer_to_unary_function::argument_type

引数型别：Arg（描述于 Adaptable Unary Function。）

pointer_to_unary_function::result_type

执行结果的型别：Result（描述于 Adaptable Unary Function。）

Result pointer_to_unary_function::operator()(Arg) const

function call 操作符。（描述于 Adaptable Unary Function。）

※ **pointer_to_unary_function::pointer_to_unary_function(Result (*f)(Arg))**

constructor。产生一个 pointer_to_unary_function，其底部（underlying）的函数指针为 f。

pointer_to_unary_function::pointer_to_unary_function()

default constructor。产生一个 pointer_to_unary_function，其底部（underlying）的函数指针为 null。

```
※ template <class Arg, class Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*x)(Arg))
```

一个辅助函数，用来产生 pointer_to_unary_function object。如果 f 的型别为 Result (*)(Arg)，则表达式 ptr_fun(f) 等价（而且更方便）于明白调用 constructor pointer_to_unary_function<Arg, Result>(f)。

15.8.4 pointer_to_binary_function

pointer_to_binary_function<Arg1, Arg2, Result>

Class pointer_to_binary_function 是一种 function object adapter，允许将函数指针 Result (*f)(Arg1, Arg2) 视为一个 Adaptable Binary Function。如果 F 是一个 pointer_to_binary_function<Arg1, Arg2, Result> object，而且以一个型别为 Result (*)(Arg1, Arg2) 的函数指针 f 作为初值，那么 F(x, y)会调用函数 f(x, y)。f 与 F 之间的唯一重要差别在于 pointer_to_binary_function 是一个 Adaptable Binary Function；也就是说它定义有嵌套型别 first_argument_type、second_argument_type、result_type。

型别为 Result (*)(Arg1, Arg2) 的那个函数指针，就自身条件而言已经是一个相当不错的 Binary Function object，它可以传入任何「需要以 Binary Function 为引数」的 STL 算法中。pointer_to_binary_function 的唯一使用时机，是当你希望在「需要 Adaptable Binary Function」的情况下使用一般函数指针（例如，以它作为某个 function object adapter 的引数）。

大多数情况下，你不应该直接使用 pointer_to_binary_function 的 constructor。使用辅助函数 ptr_fun 几乎总是比较简单的方式。

➢ 样例

以下程序片段能在链表之中找寻与 "OK" 相等的字符串。由于欲使用标准函数库函数 strcmp 作为 function object adapter 的引数，所以必须先使用 pointer_to_binary_function adapter 以提供 Adaptable Binary Function 的接口给 strcmp。

```
list<char*>::iterator item =
            find_if(L.begin(), L.end(),
                    not1(binder2nd(ptr_fun(strcmp), "OK")));
```

➢ 何处定义

HP STL 将 pointer_to_binary_function 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

Arg1　　此 function object 的第一引数型别。

Arg2　　此 function object 的第二引数型别。

Result　　此 function object 执行结果的型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- Arg1 为 Assignable。
- Arg2 为 Assignable。
- Result 为 Assignable。

➢ Public Base Classes

binary_function<Arg1, Arg2, Result>

➢ 成员

pointer_to_binary_function 的某些成员并未定义于 Adaptable Binary Function 需求条件中，这些成员是 pointer_to_binary_function 所特有，皆以 ※ 符号标示之。

pointer_to_binary_function::first_argument_type

第一引数型别：Arg1（描述于 Adaptable Binary Function。）

pointer_to_binary_function::second_argument_type

第二引数型别：Arg2（描述于 Adaptable Binary Function。）

pointer_to_binary_function::result_type

执行结果的型别：Result（描述于 Adaptable Binary Function。）

Resultpointer_to_binary_function::operator()(Arg1, Arg2) const

function call 操作符。（描述于 Adaptable Binary Function。）

※ **pointer_to_binary_function**
::pointer_to_binary_function(Result (*f)(Arg1, Arg2))

constructor。产生一个 pointer_to_binary_function，其底部（underlying）的函数指针为 f。

pointer_to_binary_function::pointer_to_binary_function()

default constructor。产生一个 pointer_to_binary_function，其底部（underlying）的函数指针为 null。

※ **template <class Arg1, class Arg2, class Result>**
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*x)(Arg1, Arg2))

一个辅助函数，用来产生 pointer_to_binary_function object。如果 f 的型别为 Result (*)(Arg1, Arg2)，则表达式 ptr_fun(f) 等价（而且更方便）于明白调用 constructor pointer_to_binary_function<Arg1, Arg2, Result>(f)。

15.8.5 unary_negate

unary_negate<Predicate>

Class unary_negate 是一种 Adaptable Predicate，用来表示其他某个 Adaptable Predicate 的逻辑负值（logical negation）。如果 f 是一个 unary_negate<Predicate> object，构造时以 pred 作为其底部的一个 function object，那么 f(x) 会返回 !pred(x)。严格来说 unary_nagate 有点多余，因为它可以用 function object logical_not 与 adapter unary_compose 构造出来。

很少有理由需要直接写出 unary_negate 的 constructor。辅助函数 not1 几乎成为最简单的使用形式。例如你通常不需要写 unary_negate<Pred>(pred)，而应该写成 not1(pred)。

➢ 样例

试着在链表中找出第一个落于 1...10 以外的元素。

```
int main()
{
  list<int> L;
  generate_n(back_inserter(L), 10000, rand);

  list<int>::iterator i =
    find_if(L.begin(), L.end(),
            not1(compose2(logical_and<bool>(),
                          bind2nd(greater_equal<int>(), 1),
                          bind2nd(less_equal<int>(), 10))));
  assert(i == L.end() || !(*i >= 1 && *i <= 10));
}
```

➢ 何处定义

HP STL 将 unary_negate 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

Predicate　　一个 function object 型别。unary_negate 会获得其逻辑负值。

➢ 隶属何种 concept

Adaptable Predicate（p.118）。

➢ 型别条件

- Predicate 为 Adaptable Predicate 的一个 model。

➢ **Public Base Classes**

unary_function<typename Predicate::argument_type, bool>

➢ **成员**

unary_negate 的某些成员并未定义于 Adaptable Predicate 需求条件中，这些成员是 unary_negate 所特有，皆以 ※ 符号标示之。

unary_negate::argument_type

引数型别：Predicate::argument_type （描述于 Adaptable Predicate。）

unary_negate::result_type

执行结果的型别：bool （描述于 Adaptable Predicate。）

bool unary_negate::operator()(argument_type) const

function call 操作符。 （描述于 Adaptable Predicate。）

※ **unary_negate::unary_negate(const Predicate&)**

constructor。产生一个 unary_negate<Predicate>，其底部（underlying）的 predicate 为 p。

※ **template <class Predicate> unary_negate<Predicate> not1(const Predicate& p)**

辅助函数，用来产生一个 unary_negate。如果 p 为 Predicate 型别，则表达式 not1(p) 等价（但使用上更方便）于 unary_negate<Predicate>(p)。

15.8.6 binary_negate

binary_negate<BinaryPredicate>

Class binary_negate 是一种 Adaptable Binary Predicate，用来表示其他某个 Adaptable Binary Predicate 的逻辑负值（logical negation）。如果 f 是一个 binary_negate<Predicate> object，构造时以 pred 作为其底部的一个 function object，那么 f(x, y) 会返回 !pred(x, y)。

很少有理由需要直接写出 binary_negate 的 constructor。辅助函数 not2 几乎成为最简单的使用形式。例如你通常不需要写 binary_negate<Pred>(pred)，而应该写成 not2(pred)。

➢ **样例**

在字符串中找出第一个不为 ' ' 或 '\n' 的字符。

```
char str[MAXLEN];
...
const char* wptr = find_if(str, str + MAXLEN,
                           compose2(not2(logical_or<bool>()),
                                    bind2nd(equal_to<char>(), ' '),
                                    bind2nd(equal_to<char>(), '\n')));
assert(wptr == str + MAXLEN || !(*wptr == ' ' || *wptr == '\n'));
```

➢ 何处定义

HP STL 将 binary_negate 定义于头文件 <function.h>。C++ standard 将它声明于 <functional>。

➢ Template 参数

BinaryPredicate function object 的型别，该 function object 是 binary_negate 的动作目标。

➢ 隶属何种 concept

Adaptable Binary Predicate（p.119）。

➢ 型别条件

- BinaryPredicate 为 Adaptable Binary Predicate 的一个 model。

➢ Public Base Classes

```
binary_function<typename BinaryPredicate::first_argument_type,
                typename BinaryPredicate::second_argument_type,
                bool>
```

➢ 成员

binary_negate 的某些成员并未定义于 Adaptable Binary Predicate 需求条件中，这些成员是 binary_negate 所特有，皆以 ※ 符号标示之。

binary_negate::first_argument_type

第一引数型别：BinaryPredicate::first_argument_type（描述于 Adaptable Binary Predicate。）

binary_negate::second_argument_type

第二引数型别：BinaryPredicate::second_argument_type（描述于 Adaptable Binary Predicate。）

binary_negate::result_type

执行结果的型别：bool （描述于 Adaptable Binary Predicate。）

bool binary_negate::operator()(first_argument_type, second_argument_type) const

function call 操作符。（描述于 Adaptable Binary Predicate。）

※ **binary_negate::binary_negate(const BinaryPredicate&)**

constructor。产生一个 binary_negate<BinaryPredicate>，其底部（underlying）的 predicate 为 p。

※ **template <class BinaryPredicate>**
binary_negate<BinaryPredicate> not2(const BinaryPredicate& p)

辅助函数，用来产生一个 binary_negate。如果 p 属于 BinaryPredicate 型别，则表达式 not2(p) 等价（但使用上更方便）于 binary_negate<BinaryPredicate>(p)。

15.8.7 unary_compose

unary_compose<Function1,Function2>

Class unary_compose 是一种 function object adapter。如果 f 和 g 都是 Adaptable Unary Functions 而且 g 的返回型别可转换为 f 的引数型别，那么 unary_compose 可被用来产生一个 function object h，使 h(x) 相同于 f(g(x))。

这样的行为称为函数合成（function composition），也因此才有 unary_compose这个名称。通常它用来表示数学运算 f o g，也就是产出一个函数，使 (f o g)(x) 即为 f(g(x))。函数合成是代数上的一个重要观念，此一观念对于「利用其他组件来构造组件」的方法也相当重要，因为这使我们得以单纯的 function objects 任意构造出复杂的 function objects。

如同其他的 function object adapters，产生一个 unary_compose 的最简单方式就是使用辅助函数 compose1。虽然直接调用 unary_compose constructor 亦可行，但通常没有理由这么做。

➢ 样例

计算 vector 中的所有元素之正弦值（sines）的负数。每个元素代表一个角度（degrees）。由于 C 函数库的 sin 需要的引数是以弧度（radians）为单位，所以整个运算由三个行为合成：角度转换为弧度、取正弦值、取负值。

```
vector<double> angles;
vector<double> sines;
const double pi = 3.14159265358979323846;
...
assert(sines.size() >= angles.size());
transform(angles.begin(), angles.end(), sines.begin(),
          compose1(negate<double>(),
                   compose1(ptr_fun(sin),
                            bind2nd(multiplies<double>(), pi / 180.))));
```

➢ 何处定义

HP STL 将 unary_compose 定义于头文件 <function.h>。SGI STL 将它定义于头文件 <functional>。unary_compose 并未涵盖于 C++ standard，但却是常见的一个扩充物。

➢ Template 参数

Function1 「函数合成」操作行为中第一操作数的型别。换句话说如果该合成行为写成 f o g，则 Function1 是 function object f 的型别。

Function2 「函数合成」操作行为中第二操作数的型别。换句话说如果该合成行为写成 f o g，则 Function2 是 function object g 的型别。

➢ 隶属何种 concept

Adaptable Unary Function（p.114）。

➢ 型别条件

- Function1 为 Adaptable Unary Function 的一个 model。
- Function2 为 Adaptable Unary Function 的一个 model。
- Function2::result_type 可转换为 Function1::argument_type。

➢ Public Base Classes

```
unary_function<typename Function2::argument_type,
               typename Function1::result_type>
```

➢ 成员

unary_compose 的某些成员并未定义于 Adaptable Unary Function 需求条件中，这些成员是 unary_compose 所特有，皆以 ※ 符号标示之。

unary_compose::argument_type

function object 的引数型别：Function2::argument_type （描述于 Adaptable Unary Function。）

unary_compose::result_type

function object 执行结果的型别：Function1::result_type （描述于 Adaptable Unary Function。）

※ **unary_compose::unary_compose(const Function1& f, const Function2& g)**

constructor。产生一个 unary_compose object，用来表示 function object f o g。

※ **template <class Function1, class Function2>**
unary_compose<Function1, Function2>
compose1(const Function1& op1, const Function2& op2);

辅助函数，用来产生一个 unary_compose object。如果 f 和 g 分别为 classes Function1 和 Function2 的 function objects，则 compose1(f,g) 等价(但使用上更方便)于 unary_compose<Function1, Function2>(f, g) 。

15.8.8 binary_compose

binary_compose<BinaryFunction, UnaryFunction1, UnaryFunction2>

Class binary_compose 是一种 function object adapter。如果 f 是 Adaptable Binary Functions，g1 和 g2 都是 Adaptable Unary Functions，而且 g1 和 g2 的返回型别皆可转换为 f 的引数型别，那么 binary_compose 可被用来产生一个 function object h，使 h(x) 相同于 f(g1(x), g2(x))。

➢ 样例

找出链表之中第一个落在 1...10 区间内的元素。

```
list<int> L;
...
list<int>::iterator in_range =
    find_if(L.begin(), L.end(),
            compose2(logical_and<bool>(),
                     bind2nd(greater_equal<int>(), 1),
                     bind2nd(less_equal<int>(), 10)));
assert(in_range == L.end() || (*in_range >= 1 && *in_range <= 10));
```

以下试针对某区间内的每个元素 x，计算 sin(x)/(x+DBL_MIN)。

```
transform(first, last, first,
          compose2(divides<double>(),
                   ptr_fun(sin),
                   bind2nd(plus<double>(), DBL_MIN)));
```

➢ 何处定义

HP STL 将 binary_compose 定义于头文件 <function.h>。SGI STL 将它定义于头文件 <functional>。binary_compose 并未涵盖于 C++ standard，但却是常见的一个扩充物。

➢ Template 参数

BinaryFunction	函数合成运算中的「外层(outer)」函数型别。也就是说，如果 binary_compose adapter 表示一个 function object h，使得 h(x) == f(g1(x), g2(x))，则 BinaryFunction 便是 f 的型别。
UnaryFunction1	函数合成运算中第一个「内层（inner）」函数的型别。也就是说，如果 binary_compose adapter 表示一个 function object h，使得 h(x) == f(g1(x), g2(x))，则 UnaryFunction1 便是 g1 的型别。
UnaryFunction2	函数合成运算中第二个「内层（inner）」函数的型别。也就是说，如果 binary_compose adapter 表示一个 function object h，使得 h(x) == f(g1(x), g2(x))，则 UnaryFunction2 便是 g2 的型别。

➢ 隶属何种 concept

Adaptable Binary Function（p.115）。

➢ 型别条件

- BinaryFunction 为 Adaptable Binary Function 的一个 model。
- UnaryFunction1 与 UnaryFunction2 为 Adaptable Unary Function 的 models。
- UnaryFunction1 与 UnaryFunction2 的结果型别可分别转换为 BinaryFunction 的第一与第二引数型别。

➢ Public Base Classes

```
unary_function<UnaryFunction1::argument_type,
               BinaryFunction::result_type>
```

➢ 成员

binary_compose 的某些成员并未定义于 Adaptable Unary Function 需求条件中，这些成员是 binary_compose 所特有，皆以 ※ 符号标示之。

binary_compose::argument_type

function object 的引数型别：UnaryFunction1::argument_type（描述于 Adaptable Unary Function。）

binary_compose::result_type

function object 执行结果的型别：BinaryFunction::result_type （描述于 Adaptable Unary Function。）

```
※ binary_compose::binary_compose(const BinaryFunction& f,
                                 const UnaryFunction1& g1, const UnaryFunction2& g2)
```

constructor。产生一个 binary_compose object，使得当我们以引数 x 调用时，会返回 f(g1(x), g2(x))。

```
※ template <class BinaryFunction,
            class UnaryFunction1, class UnaryFunction2>
  binary_compose<BinaryFunction, UnaryFunction1, UnaryFunction2>
  compose2(const BinaryFunction&,
           const UnaryFunction1&, const UnaryFunction2&);
```

辅助函数，用来产生一个 binary_compose object。如果 f、g1、g2 分别隶属于型别 BinaryFunction、UnaryFunction1、UnaryFunction2，则表达式 compose2(f, g1, g2) 等价于对 constructor 进行明白的（explicit）调用，动作如下：

```
binary_compose<BinaryFunction,
               UnaryFunction1, UnaryFunction2>(f, g1, g2)
```

但使用上更为方便。

第 16 章

Container Classes
容器类

STL 定义的所有 container classes 都是 Sequence 或 Associative Container 的 models。它们都是 templates，可被具现化以包含任何型别的 objects。例如，你可以像使用普通 C 数组一样地使用 vector<int>，它使你得以免除内存管理之类的琐事。

除了序列式 containers 和关联式 containers，STL 还定义了三种 container adapters。它们自身并不是 containers，它们不是 Container 的 models，它们刻意提供受限的功能。

16.1 序列（Sequences）

C++ standard 定义了三种序列：(1) vector（一种简单的数据结构，提供元素的快速随机访问能力）；(2) deque（一种较为复杂的数据结构，允许高效率地在 containers 两端安插及移除元素）；(3) list（双向连接链表）。vector 是 STL container classes 当中最简单的一个，通常也是最好的选择。

C++ standard 并未提供单向连接链表，不过那是常见的一种扩充物。SGI STL 就实现有一个单向连接链表 slist，本章涵盖该 class 的说明。

16.1.1 vector

vector<T, Allocator>

vector 是一种 Sequence，支持随机访问元素、常量时间内在尾端安插和移除元素，以及线性时间内在开头或中间处安插或移除元素。vector 的元素个数能够动态变更，其内存管理行为完全自动。

vector 是 STL container classes 中最简单的一个，很多情况下它也是最有效率的。通常它的实现方式是把元素安排在连续的存储块中，这使得 vector 的 iterators 可以是一般指针。

vector 的大小（size，其所包含的元素个数）与容量（capacity，其内存所能容纳的元素个数）之间有很重要的差别。要了解这一点，最简单的方式就是以典型的实现角度来看。vector 会管理其内存，并在内存开头处构造元素。典型的 vector 有三个 member variables，三个都是指针：start、finish 和 end_of_storage。vector 的所有元素位于 [start,finish) 之中，而 [finish, end_of_storage) 则包含了未初始化的存储空间。vecotr 的大小是 finish-start，容量为 end_of_storage-start。当然，容量一定大于等于其大小（元素个数）。

当你要将元素安插于 vector 内，大小与容量之间的差别就变得格外重要。如果 vector 的大小等于其容量（亦即这个 vector 已经没有未初始化的存储空间了），安插新元素的唯一方法就是增加这个 vector 的内存总量[55]。这意味得分配一块新的而且更大的内存，再将旧内存块的内容复制到新内存块中，然后归还旧内存块。

一旦 vector 的内存重新分配，其 iterators 便失效了。此外，在 vector 中间安插或删除元素，也会使指向该安插点或删除点之后元素的所有 iterators 都失效。这表示如果你以 member function reserve 来预先分配内存，或如果所有安插或删除动作都在 vector 尾端进行，那么你就可以使 vector 的 iterators 不失效。

➢ 样例

产生一个空的 vector，然后安插一个元素进去。

```
int main()
{
  vector<int> v;
  v.insert(v.begin(), 3);
  assert(v.size() == 1 && v.capacity() >= 1 && v[0] == 3);
}
```

以 vector 作为临时存储体，从标准输入设备读入一些数值，然后打印这串数值的中间值。（vector 的大小可以依需要延展。）这个例子倚赖一个与 vector constructor 有关的重要事实：vector 可以由任何种类的 Input Iterators range 构造出来，本例是由型别为 istream_iterator 的 range 构造出来。这个例子也仰赖一个事实：vector 提供 Random Access Iterators。

---

[55] vector 自动执行内存重新分配动作时，通常是以 2 为因数（factor）来增加容量。因此容量的增加与目前的容量成比例，而不是一个固定常量。 这一点很重要。前述第一种做法，会使一串元素的安插动作呈现线性时间的操作行为，而第二种做法，会使一串元素的安插动作呈现二次方（quadratic）复杂度。

```
int main()
{
  istream_iterator<double> first(cin);
  istream_iterator<double> end_of_file;
  vector<double> buf(first, end_of_file);

  nth_element(buf,begin(), buf.begin() + buf.size() / 2, buf.end());
  cout << "Median: " << buf[buf.size() / 2] << endl;
}
```

你可以使用 member function `reserve` 来增加 `vector` 的容量，但没有任何一个 member function 可以为我们缩小容量，因为不需要任何特别的 member functions 我们就可以缩小 `vector` 的容量。底下就是一种方法：

```
template <class T, class Allocator>
void shrink_to_fit(vector<T, Allocator>& v)
{
  vector<T, Allocator> tmp(v.begin(), v.end());
  tmp.swap(v);
}
```

➢ 何处定义

HP STL 将 `vector` 定义于头文件 `<vector.h>`。C++ standard 将它声明于 `<vector>`。

➢ Template 参数

`T`	`vector` 的 value type，也就是存储在此 `vector` 内的 object 型别。
`Allocator`	`vector`'s allocator，用来管理所有内部内存。**缺省值**：`allocator<T>`

➢ 隶属何种 concept

Random Access Container（p.135）、Back Insertion Sequence（p.143）。

➢ 型别条件

- `T` 为 Assignable 的 model。
- `Allocator` 是 Allocator 之 model，且其 value type 为 `T`。

➢ Public Base Classes

无

➢ 成员

vector 的某些成员并未定义于 Random Access Container 和 Back Insertion Sequence 的需求条件中，它们都是 vector 所特有，皆以 ※ 符号标示之。

vector::value_type
存储于此 vector 之内的 object 型别，T。（描述于 Container。）

vector::pointer
T 的指针。（描述于 Container。）

vector::const_pointer
const T 的指针。（描述于 Container。）

vector::reference
T 的 reference。（描述于 Container。）

vector::const_reference
const T 的 reference。（描述于 Container。）

vector::size_type
一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

vector::difference_type
一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

vector::iterator
vector 的 iterator type，是一个可变的（mutable）Random Access Iterator。（描述于 Container。）

vector::const_iterator
vector 的 constant iterator type，是一个不可变的（constant）Random Access Iterator。（描述于 Container。）

vector::reverse_iterator
用于逆向遍历的一种 vector iterator，是一个可变的（mutable）Random Access Iterator。（描述于 Reversible Container。）

vector::const_reverse_iterator
用于逆向遍历的一种 vector iterator，是一个不可变的（constant）Random Access Iterator。（描述于 Reversible Container。）

iterator vector::begin()
返回一个 iterator，指向此 vector 的开头。（描述于 Container。）

iterator vector::end()
返回一个 iterator，指向此 vector 的尾端。（描述于 Container。）

const_iterator vector::begin() const
返回一个 const_iterator，指向此 vector 的开头。（描述于 Container。）

const_iterator vector::end() const

返回一个 const_iterator，指向此 vector 的尾端。（描述于 Container。）

reverse_iterator vector::rbegin()

返回一个 reverse_iterator，指向被倒转之 vector 的开头。（描述于 Reversible Container。）

reverse_iterator vector::rend()

返回一个 reverse_iterator，指向被倒转之 vector 的尾端。（描述于 Reversible Container。）

const_reverse_iterator vector::rbegin() const

返回一个 const_reverse_iterator，指向被倒转之 vector 的开头。（描述于 Reversible Container。）

const_reverse_iterator vector::rend() const

返回一个 const_reverse_iterator，指向被倒转之 vector 的尾端。（描述于 Reversible Container。）

size_type vector::size() const

返回 vector 的元素个数。（描述于 Container。）

size_type vector::max_size() const

返回 vector 的最大可能元素个数。（描述于 Container。）

※ **size_type vector::capacity() const**

返回 vector 的容量，亦即分配获得之内存所能容纳的元素个数。容量总是大于等于其大小。（如果安插的元素多过 capacity()-size()，这个 vector 的内存会自动重新分配，此重新分配动作并不会增加 vector 的大小，也不会改变 vector 的任何元素内容，它只会增加 capacity()，并且造成任何指向此 vector 的 iterators 失效。

bool vector::empty() const

当且仅当（if and only if）vector 的大小为零，就返回 true。（描述于 Container。）

reference vector::operator[](size_type n)

返回第 n 个元素。（描述于 Random Asscess Container。）

const_reference vector::operator[](size_type n) const

返回第 n 个元素。（描述于 Random Asscess Container。）

expilcit vector::vector(const Allocator& A = Allocator())

使用指定的 allocator 产生一个空的 vector。（描述于 Sequence。）

```
expilcit vector::vector(size_type n, const T& x = T(),
                        const Allocator& A = Allocator())
```

使用指定的 allocator 产生一个具有 n 个 x 复制品的 vector。（描述于 Sequence。）

vector::vector(const vector& v)

copy constructor。（描述于 Container。）

```
template <class InputIterator>
vector::vector(InputIterator f, InputIterator l,
               const Allocator& A = Allocator())
```

使用指定之 allocator，产生一个 vector，使其拥有 [f,l) 的复制内容。（描述于 Sequence。）

```
vector::~vector()
```

destructor。（描述于 Container。）

```
vector& vector::operator=(const vector&)
```

assignment 操作符。（描述于 Container。）

※ `allocator_type vector::get_allocator() const`

返回 vector 构造时所搭配之 allocator 的一份复制品。

```
void vector::swap(vector&)
```

将两个 vectors 的内容互换。（描述于 Container。）

※ `void vector::reserve(size_type n)`

增加 vector 的容量。如果 n 小于等于 capacity()，则此函数不带来任何影响。否则它会要求分配额外内存。如果成功，capacity() 会大于或等于 n，而指向此 vector 的任何 iterators 都将失效。vector 的大小以及元素内容都维持不变。

使用 reverse() 的理由主要是为了效率。如果你知道 vector 最后会增长到多大的容量，那么一次就把所有内存分配好，可比倚赖内存自动分配机制更有效率多了。使用 reserve() 也可以让你控制 iterators 的无效化时机。

```
reference vector::front()
```

返回第一个元素。（描述于 Sequence。）

```
const_reference vector::front() const
```

返回第一个元素。（描述于 Sequence。）

```
reference vector::back()
```

返回最后一个元素。（描述于 Back Insertion Sequence。）

```
const_reference vector::back() const
```

返回最后一个元素。（描述于 Back Insertion Sequence。）

```
void vector::push_back(const T& x)
```

在 vector 尾端添加 x。（描述于 Back Insertion Sequence。）

```
void vector::pop_back()
```

移除最后一个元素。（描述于 Back Insertion Sequence。）

```
iterator vector::insert(iterator pos, const T& x)
```

在 pos 之前一个位置安插 x。指向此 vector 的所有 iterators 都可能因此失效。（描述于 Sequence。）

```
template <class InputIterator>
void vector::insert(iterator pos, InputIterator f, InputIterator l)
```

在 pos 之前一个位置安插 [f,l)。指向此 vector 的所有 iterators 都可能因此失效。（描述于 Sequence。）

```
void vector::insert(iterator pos, size_type n, const T& x)
```

在 pos 之前一个位置安插 n 个 x 复制品。指向此 vector 的所有 iterators 都可能因此失效。（描述于 Sequence。）

```
iterator vector::erase(iterator pos)
```

删除 pos 所指的元素。pos 之后的任何 iterators 都将因此失效。（描述于 Sequence。）

```
iterator vector::erase(iterator first, iterator last)
```

删除 [first,last) 内的元素。此 range 之后的任何 iterators 都将因此失效。（描述于 Sequence。）

```
void vector::clear()
```

删除 vector 的所有元素。（描述于 Sequence。）

```
void vector::resize(size_type n, const T& t = T())
```

将 vector 的大小改变为 n。（描述于 Sequence。）

```
※ template <class InputIterator>
void vector::assign(InputIterator first, InputIterator last)
```

相当于删除 *this 中的所有元素，再以 [first,last) 内的元素取而代之。

```
※ void vector::assign(size_type n, const T& x)
```

相当于删除 *this 中的所有元素，再以 n 个 x 复制品取而代之。

```
bool operator==(const vector&, const vector&)
```

验证两个 vectors 是否相等。（描述于 Forward Container。）

```
bool operator<(const vector&, const vector&)
```

以字典排列顺序比较两个 vectors。（描述于 Forward Container。）

16.1.2 list

```
list<T, Allocator>
```

list 是一种双向连接链表，其中每个元素都有一个前承元素（predecessor）和一个后继元素（successor）。也就是说它是一种 Sequence，支持前后两种移动方向，并能以 amortized const time 在 list 开头、尾端、中间处安插及移除元素。list 有一个重要性质：安插与接合（splicing）动作不会造成指向 list 的 iterators 失效，即使是删除动作，也只会令指向被删除元素的那个 iteator 失效。是的，这些 iterators 的顺序可能会被改变（也就是说在链表操作之前与之后，一个 list<T>::iterator 可能拥有不同的前承元素与后继元素），但 iterators 自身不会被无效化或因此指向不同元素，除非明白执行无效化动作或目标物变换动作。

list 和 vector 的比较深具启发意义。假设 i 是型别为 vector<T>::iterator 的一个有效 iterator，当我们在 i 之前一个位置安插或移除一个元素，i 会因此指向不同的元素，或者 i 根本完全失效。另一方面，假设 i 和 j 都是指向 vector 的 iterators，存在某个整数 n，使得 i == j + n。这个情况下即使元素被安插在 vector 之中且 i 和 j 指向不同元素，两个指针的相互关系仍然维持不变。list 正好相反，其 iterators 不会被无效化，也不会指向不同元素，但 iterators 的前承元素与后继元素的关系不会维持不变。

一般而言 list 会被实现成一些节点（nodes）的集合，每个节点包含一个元素，并且带有指针，指向一个前承元素和一个后继元素。

有时候「只允许前进（forward）移动」的单向串链链表也是很有用的。如果你不需要往回（backward）移动，slist（p.448）可能会比 list 更有效率。

➢ 样例

产生两个空的 lists，为它们新增元素，然后对它们排序，然后将这两个 lists 合并成单个 list。

```
int main()
{
  list<int> L1;
  L1.push_back(0);
  L1.push_front(1);
  L1.insert(++L1.begin(), 3);

  list<int> L2;
  L2.push_back(4);
  L2.push_front(2);

  L1.sort();
  L2.sort();
  L1.merge(L2);
  assert(L1.size() == 5);
  assert(L2.size() == 0);
  L1.reverse();
  copy(L1.begin(), L1.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

输出为：

```
4 3 2 1 0
```

➢ 何处定义

HP STL 将 list 定义于头文件 <list.h>。C++ standard 将它声明于 <list>。

➢ Template 参数

T	list 的 value type：存储于 list 内的 object 型别。
Allocator	list 的 allocator，用来管理所有内部内存。**缺省值**：allocator<T>

➢ 隶属何种 concept

Reversible Container（p.133）、Front Insertion Sequence（p.141）、Back Insertion Sequence（p.143）。

➢ 型别条件

- T 为 Assignable 的 model。
- Allocator 是 Allocator 之 model，且其 value type 为 T。

➢ Public Base Classes

无

➢ 成员

list 的某些成员并未定义于 Reversible Container、Front Insertion Sequence 和 Back Insertion Sequence 需求条件中，它们是 list 所特有，皆以 ※ 符号标示之。

list::value_type
存储于此 list 之内的 object 型别，T。（描述于 Container。）

list::pointer
T 的指针。（描述于 Container。）

list::const_pointer
const T 的指针。（描述于 Container。）

list::reference
T 的 reference。（描述于 Container。）

list::const_reference
const T 的 reference。（描述于 Container。）

list::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

list::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

list::iterator

list 的 iterator type，是一个可变的（mutable）Bidirectional Iterator。（描述于 Container。）

list::const_iterator

list 的 constant iterator type，是一个不可变的（constant）Bidirectional Iterator。（描述于 Container。）

list::reverse_iterator

用于逆向遍历的一种 list iterator，是一个可变的（mutable）Bidirectional Iterator。（描述于 Reversible Container。）

list::const_reverse_iterator

用于逆向遍历的一种 list iterator，是一个不可变的（constant）Bidirectional Iterator。（描述于 Reversible Container。）

iterator list::begin()

返回一个 iterator，指向此 list 的开头。（描述于 Container。）

iterator list::end()

返回一个 iterator，指向此 list 的尾端。（描述于 Container。）

const_iterator list::begin() const

返回一个 const_iterator，指向此 list 的开头。（描述于 Container。）

const_iterator list::end() const

返回一个 const_iterator，指向此 list 的尾端。（描述于 Container。）

reverse_iterator list::rbegin()

返回一个 reverse_iterator，指向被倒转之 list 的开头。（描述于 Reversible Container。）

reverse_iterator list::rend()

返回一个 reverse_iterator，指向被倒转之 list 的尾端。（描述于 Reversible Container。）

const_reverse_iterator list::rbegin() const

返回一个 const_reverse_iterator，指向被倒转之 list 的开头。（描述于 Reversible Container。）

const_reverse_iterator list::rend() const

返回一个 const_reverse_iterator，指向被倒转之 list 的尾端。（描述于 Reversible Container。）

size_type list::size() const

返回 list 的元素个数。注意，size 的运行期复杂度只为 *O(N)*。你不应该预期它会是 *O(1)*。（描述于 Container。）

size_type list::max_size() const

返回 list 的最大可能元素个数。（描述于 Container。）

bool list::empty() const

当且仅当（if and only if）list 的大小为零，就返回 true。（描述于 Container。）

expilcit list::list(const Allocator& A = Allocator())

使用指定的 allocator 产生一个空的 list。（描述于 Sequence。）

expilcit list::list(size_type n, const T& x = T(),
const Allocator& A = Allocator())

使用指定的 allocator 产生一个具有 n 个 x 复制品的 list。（描述于 Sequence。）

list::list(const list& l)

copy constructor。（描述于 Container。）

template <class InputIterator>
list::list(InputIterator f, InputIterator l,
const Allocator& A = Allocator())

使用指定之 allocator，产生一个 list，使其拥有 [f,l) 的复制内容。（描述于 Sequence。）

list::~list()

destructor。（描述于 Container。）

list& list::operator=(const list&)

assignment 操作符。（描述于 Container。）

※ **allocator_type list::get_allocator() const**

返回 list 构造时所搭配之 allocator 的一份复制品。

void list::swap(list&)

将两个 lists 的内容互换。（描述于 Container。）

reference list::front()

返回第一个元素。（描述于 Sequence。）

const_reference list::front() const

返回第一个元素。（描述于 Sequence。）

reference list::back()

返回最后一个元素。（描述于 Back Insertion Sequence。）

const_reference list::back() const

返回最后一个元素。（描述于 Back Insertion Sequence。）

void list::push_front(const T& t)

将 t 安插于开头。（描述于 Front Insertion Sequence。）

void list::pop_front()

移除（remove）第一个元素。（描述于 Front Insertion Sequence。）

void list::push_back(const T& t)

将 t 安插于尾端。（描述于 Back Insertion Sequence。）

void list::pop_back()

移除（remove）最后一个元素。（描述于 Back Insertion Sequence。）

iterator list::insert(iterator pos, const T& x)

在 pos 之前一个位置安插 x。不会有任何一个 list iterators 因此而失效。（描述于 Sequence。）

template <class InputIterator>
void list::insert(iterator pos, InputIterator f, InputIterator l)

在 pos 之前一个位置安插 [f,l)。不会有任何一个 list iterators 因此而失效。（描述于 Sequence。）

void list::insert(iterator pos, size_type n, const T& x)

在 pos 之前一个位置安插 n 个 x 复制品。不会有任何一个 list iterators 因此而失效。（描述于 Sequence。）

iterator list::erase(iterator pos)

删除 pos 所指的元素。除了 pos，没有任何一个 list iterators 会因此而失效。（描述于 Sequence。）

iterator list::erase(iterator first, iterator last)

删除 [first,last) 内的元素。除了指向被删除元素的那些 iterators 外，没有任何 list iterators 会因此而失效。（描述于 Sequence。）

void list::clear()

删除 list 的所有元素。（描述于 Sequence。）

void list::resize(size_type n, const T& t = T())

将 list 的大小改变为 n。（描述于 Sequence。）

※ **template <class InputIterator>**
void list::assign(InputIterator first, InputIterator last)

相当于删除 *this 中的所有元素，再以 [first,last) 内的元素取而代之。

※ **void list::assign(size_type n, const T& x)**

相当于删除 *this 中的所有元素，再以 n 个 x 复制品取而代之。

※ **void list::splice(iterator pos, list& x)**

移除 x 的所有元素并将这些元素安插于 pos 的前一个位置。Iterator pos 必须是 *this 中一个有效的 iterator，x 必须是不同于 *this 的一个 list。所有 iterators 都将继续有效，包括指向 x 元素的那些 iterators。复杂度为 *O(1)*。

※ **void list::splice(iterator pos, list& x, iterator i)**

将 i 所指的元素由 x 移往 *this，并将它安插于 pos 的前一个位置。Iterator pos 必须是 *this 中一个有效的 iterator，i 必须是 x 中的一个可提领（dereferenceable）iterator。*this 与 x 不必

不同。所有 iterators 都将继续有效，包括那些指向 x 元素的 iterators。如果 pos == i 或 pos == ++i，此函数将不带来任何影响（所谓 null operation）。复杂度：*O(1)*。

※ **void list::splice(iterator pos, list& x, iterator f, iterator l)**

将 [f,l) 由 x 移往 *this，并安插于 pos 的前一个位置。Iterator pos 必须是 *this 中一个有效的 iterator，[f,l) 必须是 x 中一个有效的 range。*this 与 x 不必不同，但 pos 不能是 [f,l) 之中的 iterator。所有 iterators 都继续有效，包括那些指向 x 元素的 iterators。复杂度：*O(1)*。

※ **void list::remove(const T& val)**

移除所有与 val 相等的元素。未被移除的元素其相对顺序不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size() 次相等测试。

※ **template <class Predicate>**
void list::remove_if(Predicate p)

移除所有「使得 p(*i) 为 true」的元素 *i。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size() 次的 p 动作。

※ **void list::unique()**

移除等值元素所形成的相邻群组中第一个以外的元素。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size()-1 次相等测试。

※ **template <class BinaryPredicate> void list::unique(BinaryPredicate p)**

移除「等价元素」所形成的相邻群组中第一个以外的元素。所谓「等价元素」是指如果有两个元素 *i 和 *j 使得 p(*i, *j) 为 true，这两个元素便被视为等价（equivalent）。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size()-1 次等价测试（以 p 为之）。

※ **void list::merge(list& x)**

将两个 sorted lists 合并，亦即移除 x 的所有元素，然后将它们安插于 *this 之中。*this 和 x 都必须已经根据 operator< 排过序，而且这两个链表不能是同一个。合并动作是稳定的（stable）。如果 *this 的某元素等价于 x 的某元素，那么来自 *this 的元素会被排在来自 x 的元素之前。所有指向 *this 和 x 的 iterators 都维持有效。此函数之复杂度为 *O(N)*：它至多执行 size()+x.size()-1 次比较动作。

※ **template <class StrictWeakOrdering>**
void list::merge(list& x, StrictWeakOrdering Comp)

将两个 sorted lists 合并，亦即移除 x 的所有元素，然后将它们安插于 *this 之中，其中顺序关系系由 Comp 来定义。作为比较用的函数 Comp 必须是个 Strict Weak Ordering，其引数型别为 T，而 *this 和 x 都必须根据这种顺序关系来排序。x 和 *this 不能是同一个链表。合并动作是稳定的（stable）。如果 *this 的某元素等价于 x 的某元素，那么来自 *this 的元素将会被排在来自 x 的元素之前。所有指向 *this 与 x 之中的 iterators 都维持有效。此函数之复杂度为 *O(N)*：它至多执行 Comp 动作共 size()+x.size()-1 次。

※ **void list::reverse()**

逆转链表各元素的顺序。所有 iterators 都维持有效，并继续指向原元素。此函数耗用线性时间。

注意，另有一个全局算法 reverse。如果 L 是一个 list，则 L.reverse 和 reverse(L.begin(), L.end()) 都是逆转 L 的正确方式。差别在于 L.reverse() 会保留指向 L 之每个 iterator 的值，但不维护其前承元素与后继元素的相对关系。reverse(L.begin(), L.end()) 不会保留每个指向 L 之 iterator 的值，但能维持此等 iterator 的前承元素与后继元素的相对关系。此外，reverse 算法运用 T 的 assignment 操作符，但 member function reverse 并不如此。

※ **void list::sort()**

根据 operator<，将 *this 排序。此排序动作是稳定的（stable）。等价元素的相对顺序都能够获得保留。所有 iterators 也都维持有效并持续指向原元素。比较动作的执行次数大约为 N log N，此处的 N 为 list 的大小。

※ **template <class StrictWeakOrdering>**
void list::sort(StrictWeakOrdering Comp)

根据 Strict Weak Ordering Comp，将 *this 排序。此排序动作是稳定的（stable）。等价元素的相对顺序都能够获得保留。所有 iterators 也都维持有效并持续指向原本的元素。比较动作的执行次数大约为 N log N，此处的 N 为 list 的大小。

bool operator==(const list&, const list&)

验证两个 lists 是否相等。（描述于 Forward Container。）

bool operator<(const list&, const list&)

以字典排列顺序比较两个 lists。（描述于 Forward Container。）

16.1.3 slist

slist<T, Allocator>

slist 是一种单向连接链表，链表中每个元素都链结至下一个元素，但不链结前一个元素。也就是说 slist 是一种 Sequence，支持前进（forward，但不支持回头，backward）移动，并能够在 amortized constant time 内安插及移除元素。单向连接链表在诸如 Common Lisp、Schemem、ML 等程序语言中是很常见的。某些语言甚至将所有数据结构都以单向连接链表来表示。

就像 list 一样，slist 具有一个重要性质：安插与接合（splicing）动作不会造成 slist iterators 失效。甚至 erase 也只会令「指向被删除元素」的那个 iterator 失效。这些 iterators 的顺序可能会被改变（也就是说 slist<T>::iterator 在链表操作前后可能具有不同的前承元素或后继元素），但 iterators 自身不会被无效化或指向不同元素，除非明白地执行了无效化动作或目标物变更动作。

slist 与 list 之间主要的差别在于：list 的 iterators 是一种 Bidirectional Iterators，而 slist 的 iterators 是一种 Forward Iterators。这表示 slist 的功能比 list 少。然而 Bidirectional Iterators 的一些额外功能往往是没有必要的。除非你需要这些额外功能，否则你应该使用 slist，因为单向连接链表比起双向连接链表来得更小且更快。（译注：使用 list 而不使用 slist 的另一个原因是，毕竟 C++ standard 只支持 list。）

然而，我要提一个重要的警告。如同其他 Sequences 一样，slist 定义了 member functions insert 与

erase。如果不是很谨慎地使用这些 member functions，你的程序执行起来可能会严重变慢。问题出在 insert 的第一引数是个 iterator pos，它会将新元素安插于 pos 之前（而非之后）。这表示 insert 必须找到 pos 之前的那个 iterator。由于 Bidirectional Iterators 的缘故，list 只需耗用常量时间就可以完成了，但 slist 就必须从链表头部开始移动。换句话说，除了 slist 起头附近，insert 和 erase 作用在任何位置上都是缓慢的行为。

slist class 提供 member function insert_after 和 erase_after，这是常量时间的操作行为。你应该尽可能以这些函数取代 insert 和 erase。如果你发现 insert_after 和 erase_after 不能满足你的需求，而常常必须使用 insert 和 erase 来取代，那么你可能应该使用 list 而非 slist。

➢ 样例

产生一个 slist，并安插一些元素进去。

```
int main()
{
  slist<int> L;
  L.push_front(0);
  L.push_front(1);
  L.insert_after(L.begin(), 2);
  copy(L.begin(), L.end(),
       ostream_iterator<int>(cout, " "));
  cout << endl;
}
// 输出为: 1 2 0
```

如果能够一次一个元素地将链表构造起来，往往是很方便的。面对 slist，最明显的方法就是使用 push_front（但是并没有 push_back；别忘了 slist 隶属于 Front Insertion Sequence 而非 Back Insertion Sequence）。当然，这会造成 slist 中的元素以「相反于新增元素的顺序」排列。

如果你想要让元素次序和新增元素的顺序相同，有两种选择。第一，你可以在元素全部加入之后，使用 member function reverse。第二，你可以运用 LISP 程序普遍使用的技巧：维护一个 iterator，指向最后元素。下例展示了这种做法。

```
int main() {
  slist<double> L;
  double x = 1;
  L.push_front(x);
  slist<double>::iterator back = L.begin();

  while (x < 1000000.)
    back = L.insert_after(back, x *= 2);
  copy(L.begin(), L.end(),
```

```
        ostream_iterator<double>(cout, "\n"));
}
// 输出 1~1048576 之间的 2 的幂次方（powers of 2）。
```

➢ 何处定义

slist 并非 HP STL 的一部分，也不是 C++ standard 的一部分，但却是很普遍的扩充功能。SGI STL 把它定义于头文件 <slist>。

➢ Template 参数

T	slist 的 value type：存储于此 slist 内的 object 型别。
Allocator	slist 的 allocator，用来管理所有内部内存。**缺省值**：allocator<T>

➢ 隶属何种 concept

Front Insertion Sequence（p.141）。

➢ 型别条件

- T 为 Assignable 的 model。
- Allocator 是 Allocator 的 model，其 value type 为 T。

➢ Public Base Classes

无

➢ 成员

slist 的某些成员并未定义于 Front Insertion Sequence 需求条件中，它们是 slist 所特有，皆以 ※ 符号标示之。

slist::value_type

存储于此 slist 之内的 object 型别，T。（描述于 Container。）

slist::pointer

T 的指针。（描述于 Container。）

slist::const_pointer

const T 的指针。（描述于 Container。）

slist::reference

T 的 reference。（描述于 Container。）

slist::const_reference

const T 的 reference。（描述于 Container。）

slist::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

slist::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

slist::iterator

slist 的 iterator type，是一个可变的（mutable）Forward Iterator。（描述于 Container。）

slist::const_iterator

slist 的 constant iterator type，是一个不可变的（constant）Forward Iterator。（描述于 Container。）

iterator slist::begin()

返回一个 iterator，指向此 slist 的开头。（描述于 Container。）

iterator slist::end()

返回一个 iterator，指向此 slist 的尾端。（描述于 Container。）

const_iterator slist::begin() const

返回一个 const_iterator，指向此 slist 的开头。（描述于 Container。）

const_iterator slist::end() const

返回一个 const_iterator，指向此 slist 的尾端。（描述于 Container。）

size_type slist::size() const

返回 slist 的元素个数。注意，size 的运行期复杂度只为 *O(N)*。你不应该预期它会是 *O(1)*。（描述于 Container。）

size_type slist::max_size() const

返回 slist 的最大可能元素个数。（描述于 Container。）

bool slist::empty() const

当且仅当（if and only if）slist 的大小为零，就返回 true。（描述于 Container。）

expilcit slist::slist(const Allocator& A = Allocator())

使用指定的 allocator 产生一个空的 slist。（描述于 Sequence。）

expilcit slist::slist(size_type n, const T& x = T(),
const Allocator& A = Allocator())

使用指定的 allocator 产生一个具有 n 个 x 复制品的 slist。（描述于 Sequence。）

slist::slist(const slist& l)

copy constructor。（描述于 Container。）

```
template <class InputIterator>
slist::slist(InputIterator f, InputIterator l,
             const Allocator& A = Allocator())
```

使用指定之 allocator，产生一个 slist，使其拥有 [f,l) 的复制内容。（描述于 Sequence。）

```
slist::~slist()
```

destructor。（描述于 Container。）

```
slist& slist::operator=(const slist&)
```

assignment 操作符。（描述于 Container。）

※ `allocator_type slist::get_allocator() const`

返回 slist 构造时所搭配之 allocator 的一份复制品。

```
void slist::swap(slist&)
```

将两个 slists 的内容互换。（描述于 Container。）

```
reference slist::front()
```

返回第一个元素。（描述于 Sequence。）

```
const_reference slist::front() const
```

返回第一个元素。（描述于 Sequence。）

```
void slist::push_front(const T& t)
```

将 t 安插于开头。（描述于 Front Insertion Sequence。）

```
void slist::pop_front()
```

移除（remove）第一个元素。（描述于 Front Insertion Sequence。）

※ `iterator slist::previous(iterator pos)`

返回 pos 的前一个元素。引数 pos 必须是 *this 中的一个有效 iterator，返回值是一个 iterator prev，使得 ++prev == pos。复杂度：与 pos 之前的 iterator 个数成线性关系。

这个 member function 显示出 list 与 slist 之间本质上的不同。它是线性时间而非常量时间，因为 slist 提供的是 Forward Iterators 而非 Bidirectional Iterators。同样道理 insert 和 erase 是线性的，因为它们必须调用 previous。

※ `const_iterator slist::previous(const_iterator pos)`

返回 pos 的前一个元素。引数 pos 必须是 *this 中的一个有效 iterator，返回值是一个 iterator prev，使得 ++prev == pos。复杂度：与 pos 之前的 iterator 个数成线性关系。

```
iterator slist::insert(iterator pos, const T& x)
```

在 pos 之前一个位置安插 x。不会有任何一个 slist iterators 因此而失效。（描述于 Sequence。）

```
template <class InputIterator>

void slist::insert(iterator pos, InputIterator f, InputIterator l)
```

在 pos 之前一个位置安插 [f,l)。不会有任何一个 slist iterators 因此而失效。（描述于 Sequence。）

void slist::insert(iterator pos, size_type n, const T& x)

在pos之前一个位置安插n个x复制品。不会有任何slist iterators 因此失效。（描述于Sequence。）

iterator slist::erase(iterator pos)

删除pos所指的元素。除了 pos，没有任何一个 slist iterators 会因此而失效。（描述于Sequence。）

iterator slist::erase(iterator first, iterator last)

删除 [first,last) 内的元素。除了指向被删除元素的那些 iterators 外，没有任何 slist iterators 会因此而失效。（描述于 Sequence。）

void slist::clear()

删除 slist 的所有元素。（描述于 Sequence。）

void slist::resize(size_type n, const T& t = T())

将 slist 的大小改变为 n。（描述于 Sequence。）

※ **template <class InputIterator>**
void slist::assign(InputIterator first, InputIterator last)

相当于删除 *this 中的所有元素，再以 [first,last) 内的元素取而代之。

※ **void slist::assign(size_type n, const T& x)**

相当于删除 *this 中的所有元素，再以 n 个 x 复制品取而代之。

※ **iterator slist::insert_after(iterator pos, const T& x)**

在 pos 之后安插 x，pos 必须是 *this 中的一个可提领（dereferenceable）iterator（亦即不能为 end()）。返回值是个 iterator，指向新元素。没有任何 slist iterators 会因此失效。复杂度：*O(1)*。

※ **template <class InputIterator>**
void slist::insert_after(iterator pos, InputIterator f, InputIterator l)

将 [f,l) 内的元素安插于 *this，使它们紧邻于 pos 之后。没有任何 slist iterators 会因此而失效。复杂度：与 l-f 成线性关系。

※ **void slist::insert_after(iterator pos, size_type n, const T& x)**

紧邻 pos 之后安插 n 个 x 复制品。没有任何 slist iterators 会因此而失效。复杂度：与 n 成线性关系。

※ **iterator slist::erase_after(iterator pos)**

移除 pos 之后的那个 iterator 所指元素。复杂度：常量时间。

※ **iterator slist::erase_after(iterator before_first, iterator last)**

移除 [before_first+1, last) 内的所有元素。复杂度：与该 range 内的元素个数成线性关系。

※ **void slist::splice(iterator pos, slist& x)**

移除 x 的所有元素并将这些元素安插于 pos 的前一个位置。Iterator pos 必须是 *this 中一个有效的 iterator，x 必须是不同于 *this 的一个 slist。所有 iterators 都将继续有效，包括指向 x 元素的那些 iterators。复杂度：与 pos 之前的元素个数成线性关系。

※ **void slist::splice(iterator pos, slist& x, iterator i)**

将 i 所指的元素由 x 移往 *this，并将它安插于 pos 的前一个位置。Iterator pos 必须是 *this

中一个有效的 iterator，i 必须是 x 中的一个可提领（dereferenceable）iterator。*this 与 x 不必不同。所有 iterators 都将继续有效，包括那些指向 x 元素的 iterators。如果 pos == i 或 pos == ++i，此函数将不带来任何影响（所谓 null operation）。复杂度：与 c1(pos-begin()) + c2(i-x.begin()) 成比例，此处 c1 和 c2 是未知常量。

※ **void slist::splice(iterator pos, slist& x, iterator f, iterator l)**

将 [f,l) 由 x 移往 *this，并安插于 pos 的前一个位置。Iterator pos 必须是 *this 中一个有效的 iterator，[f,l) 必须是 x 中一个有效的 range。*this 与 x 不必不同，但 pos 不能是 [f,l) 之中的 iterator。所有 iterators 都继续有效，包括那些指向 x 元素的 iterators。复杂度：与 c1(pos-begin()) + c2(f-x.begin()) + c3(l-f) 成比例，此处的 c1、c2、c3 是未知常量。

※ **void slist::splice_after(iterator pos, iterator prev)**

将 prev 之后的那个元素移到 *this 中，并安插于 pos 之后。Iterator pos 必须是 *this 中的一个有效 iterator，prev 必须是*this 或其他某个 slist 中的一个可提领的（dereferenceable）iterator。（由于 pos 与 prev 都必须可提领，所以它们都不可以是 end()）。复杂度：*O(1)*。

※ **void slist::splice_after(iterator pos,**
iterator before_first, iterator before_last)

将 range [before_first+1, before_last+1) 移到 *this 中，并安插于紧邻 pos 之后。Iterator pos 必须是 *this 中的一个有效 iterator，before_first 和 before_last 必须是 *this 或其他某个 slist 中的一个可提领的（dereferenceable）iterator。复杂度：*O(1)*。

※ **void slist::remove(const T& val)**

移除所有与 val 相等的元素。未被移除的元素其相对顺序不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size() 次相等测试。

※ **template <class Predicate>**
void slist::remove_if(Predicate p)

移除所有「使得 p(*i) 为 true」的元素 *i。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size() 次的 p 动作。

※ **void slist::unique()**

移除等值元素所形成的相邻群组中第一个以外的元素。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size()-1 次相等测试。

※ **template <class BinaryPredicate>**
void slist::unique(BinaryPredicate p)

移除「等价元素」所形成的相邻群组中第一个以外的元素。所谓「等价元素」是指如果有两个元素 *i 和 *j 使得 p(*i, *j) 为 true，这两个元素便被视为等价（equivalent）。未被移除的元素其相对顺序维持不变，且其 iterators 也都维持有效。此函数之复杂度为 *O(N)*。它恰恰执行 size()-1 次等价测试（以 p 为之）。

※ **void slist::merge(slist& x)**

将两个 sorted slists 合并，亦即移除 x 的所有元素，然后将它们安插于 *this。*this 和 x 都必须根据 operator< 排序，而且这两个链表不能是同一个。合并动作是稳定的（stable）。如果 *this 的某元素等价于 x 的某元素，那么来自 *this 的元素会被排在来自 x 的元素之前。所

有指向 *this 和 x 的 iterators 都维持有效。此函数之复杂度为 *O(N)*：它至多执行 size()+x.size()-1 次比较动作。

※ **template <class StrictWeakOrdering>**
void slist::merge(slist& x, StrictWeakOrdering Comp)

将两个 sorted slists 合并，亦即移除 x 的所有元素，然后将它们安插于 *this 之中，其中顺序关系系由 Comp 来定义。作为比较用的函数 Comp 必须是个 Strict Weak Ordering，其引数型别为 T，而 *this 和 x 都必须根据这种顺序关系来排序。x 和 *this 不能是同一个链表。合并动作是稳定的（stable）。如果 *this 的某元素等价于 x 的某元素，那么来自 *this 的元素将会被排在来自 x 的元素之前。所有指向 *this 与 x 之中的 iterators 都维持有效。此函数之复杂度为 *O(N)*：它至多执行 Comp 动作共 size()+x.size()-1 次。

※ **void slist::reverse()**

逆转链表中各个元素的顺序。所有 iterators 都维持有效，并继续指向同一个元素。复杂度为 *O(N)*。

※ **void slist::sort()**

根据 operator<，将 *this 排序。此排序动作是稳定的（stable）。等价元素的相对顺序都能够获得保留。所有 iterators 也都维持有效并持续指向同一个元素。比较动作的执行次数大约为 N log N，此处的 N 为 slist 的大小。

※ **template <class StrictWeakOrdering>**
void slist::sort(StrictWeakOrdering Comp)

根据 Strict Weak Ordering Comp，将 *this 排序。此排序动作是稳定的（stable）。等价元素的相对顺序都能够获得保留。所有 iterators 也都维持有效并持续指向同一个元素。比较动作的执行次数大约为 N log N，此处的 N 为 slist 的大小。

bool operator==(const slist&, const slist&)

验证两个 slists 是否相等。（描述于 Forward Container。）

bool operator<(const slist&, const slist&)

以字典排列顺序比较两个 slists。（描述于 Forward Container。）

16.1.4 deque

deque<T, Allocator>

deque[56]类似 vector（p.435）。和 vector 一样，它也是一种 Sequence，支持元素随机访问、常量时间内于尾端安插或移除元素、线性时间内于中间处安插或移除元素。

deque 和 vector 的差异在于，deque 另外还提供常量时间内于序列开头新增或移除元素。在 deque 开头或尾端安插一个元素，需要 amortized constant time，在中间处安插元素的复杂度则与 n 成线性关系，此处 n 是安插点距离 deque 开头与尾端的最短距离。

---

[56] deque 这个名称念作「deck」，代表「双向队列」。Knuth [Knu97] 书上记载，这个名称由 E. J. Schweppe 所创。进一步信息请参考 Knuth 书籍的 2.2.1 节。

另一个差异是 deque 不具备类似 vector 的 capacity() 和 reserve() 之类的 member functions，也不提供任何与这些 member functions 相关之 iterator 有效性保证。取而代之的是，它保证「将元素安插于 deque 开头或尾端」不会造成 deque 中现存的任何元素被复制。对 deque 而言，类似 vector 内存重新分配的行为是不会发生的。

一般来说，insert（包括 push_front 与 push_back）会造成指向 deque 的所有 iterators 失效。对 deque 中段调用 erase 亦会造成指向 deque 的所有 iterators 失效。至于对着 deque 的开头或尾端调用 erase（包括 pop_front 与 pop_back），则只会造成被移除元素的 iterator 失效。

通常，deque 系以动态区段化（dynamic segmented）的数组实现出来。deque 通常内含一个表头，指向一组节点（nodes），每个节点包含固定数量并且连续存储的元素。当 deque 增长时便增加新的节点。

deque 的内部细节对你并不重要。重要的是能够认知到 deque 远比 vector 复杂。对一个 deque iterator 做累加动作，并非只是单纯的指针累加；必须包括至少一次比较动作。所以，尽管 vector 与 deque 都提供 Random Access Iterators，你应该能够预期，deque iterator 上的任何操作行为都要比 vector iterator 上的相同行为慢许多。

除非你需要的功能只有 deque 才提供（例如在常量时间内于开头做安插或移除动作），否则应该尽量使用 vector。同样道理，对 deque 排序几乎不会是个好主意，你应该将此 deque 的元素复制到一个 vector，然后再对此 vector 排序，再将所有元素复制回 deque，这样会比较快。

➢ 样例

```
int main()
{
  deque<int> Q;
  Q.push_back(3);
  Q.push_front(1);
  Q.insert(Q.begin() + 1, 2);
  Q[2] = 0;
  copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
}
```

输出为：

```
1 2 0
```

➢ 何处定义

HP STL 将 deque 定义于头文件 <deque.h>。C++ standard 将它声明于 <deque>。

➢ Template 参数

T	deque 的 value type：存储在此 deque 内的 object 型别。
Allocator	deque 的 allocator，用来管理所有内部内存。**缺省值**：allocator<T>

➢ 型别条件

- T 为 Assignable 的 model。
- Allocator 是 Allocator 的 model。其 value type 为 T。

➢ 隶属何种 concept

Random Access Container（p.135）、Front Insertion Sequence（p.141）、Back Insertion Sequence（p.143）。

➢ 成员

deque 的某些成员并未定义于 Random Access Container、Front Insertion Sequence 与 Back Insertion Sequence 的需求条件中，它们都是 deque 所特有，皆以 ※ 符号标示之。

deque::value_type

存储于此 deque 之内的 object 型别，T。（描述于 Container。）

deque::pointer

T 的指针。（描述于 Container。）

deque::const_pointer

const T 的指针。（描述于 Container。）

deque::reference

T 的 reference。（描述于 Container。）

deque::const_reference

const T 的 reference。（描述于 Container。）

deque::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

deque::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

deque::iterator

deque 的 iterator type，是一个可变的（mutable）Random Access Iterator。（描述于 Container。）

deque::const_iterator

deque 的 constant iterator type，是一个不可变的（constant）Random Access Iterator。（描述于 Container。）

deque::reverse_iterator

用于逆向遍历的一种 deque iterator，是一个可变的（mutable）Random Access Iterator。（描述于 Reversible Container。）

deque::const_reverse_iterator

用于逆向遍历的一种 deque iterator，是一个不可变的（constant）Random Access Iterator。（描述于 Reversible Container。）

iterator deque::begin()

返回一个 iterator，指向此 deque 的开头。（描述于 Container。）

iterator deque::end()

返回一个 iterator，指向此 deque 的尾端。（描述于 Container。）

const_iterator deque::begin() const

返回一个 const_iterator，指向此 deque 的开头。（描述于 Container。）

const_iterator deque::end() const

返回一个 const_iterator，指向此 deque 的尾端。（描述于 Container。）

reverse_iterator deque::rbegin()

返回一个 reverse_iterator，指向被倒转之 deque 的开头。（描述于 Reversible Container。）

reverse_iterator deque::rend()

返回一个 reverse_iterator，指向被倒转之 deque 的尾端。（描述于 Reversible Container。）

const_reverse_iterator deque::rbegin() const

返回一个 const_reverse_iterator，指向被倒转之 deque 的开头。（描述于 Reversible Container。）

const_reverse_iterator deque::rend() const

返回一个 const_reverse_iterator，指向被倒转之 deque 的尾端。（描述于 Reversible Container。）

size_type deque::size() const

返回 deque 的元素个数。（描述于 Container。）

size_type deque::max_size() const

返回 deque 的最大可能元素个数。（描述于 Container。）

bool deque::empty() const

当且仅当（if and only if）deque 的大小为零，就返回 true。（描述于 Container。）

reference deque::operator[](size_type n)

返回第 n 个元素。（描述于 Random Asscess Container。）

const_reference deque::operator[](size_type n) const

返回第 n 个元素。（描述于 Random Asscess Container。）

expilcit deque::deque(const Allocator& A = Allocator())

使用指定的 allocator 产生一个空的 deque。（描述于 Sequence。）

expilcit deque::deque(size_type n, const T& x = T(),
const Allocator& A = Allocator())

使用指定的 allocator 产生一个具有 n 个 x 复制品的 deque。（描述于 Sequence。）

deque::deque(const deque& d)

copy constructor。（描述于 Container。）

template <class InputIterator>
deque::deque(InputIterator f, InputIterator l,
const Allocator& A = Allocator())

使用指定之 allocator，产生一个 deque，使其拥有 [f,l) 的复制内容。（描述于 Sequence。）

deque::~deque()

destructor。（描述于 Container。）

deque& deque::operator=(const deque&)

assignment 操作符。（描述于 Container。）

※ **allocator_type deque::get_allocator() const**

返回 deque 构造时所搭配之 allocator 的一份复制品。

void deque::swap(deque&)

将两个 deques 的内容互换。（描述于 Container。）

reference deque::front()

返回第一个元素。（描述于 Sequence。）

const_reference deque::front() const

返回第一个元素。（描述于 Sequence。）

reference deque::back()

返回最后一个元素。（描述于 Back Insertion Sequence。）

const_reference deque::back() const

返回最后一个元素。（描述于 Back Insertion Sequence。）

void deque::push_front(const T& t)

将 t 安插于开头处。（描述于 Front Insertion Sequence。）

void deque::pop_front()

移除第一个元素。（描述于 Front Insertion Sequence。）

void deque::push_back(const T& x)

在 deque 尾端添加 x。（描述于 Back Insertion Sequence。）

void deque::pop_back()

移除最后一个元素。（描述于 Back Insertion Sequence。）

iterator deque::insert(iterator pos, const T& x)

在 pos 之前一个位置安插 x。（描述于 Sequence。）

template <class InputIterator>
void deque::insert(iterator pos, InputIterator f, InputIterator l)

在 pos 之前一个位置安插 [f,l)。（描述于 Sequence。）

void deque::insert(iterator pos, size_type n, const T& x)

在 pos 之前一个位置安插 n 个 x 复制品。（描述于 Sequence。）

iterator deque::erase(iterator pos)

删除 pos 所指的元素。（描述于 Sequence。）

iterator deque::erase(iterator first, iterator last)

删除 [first,last) 内的元素。（描述于 Sequence。）

void deque::clear()

删除 deque 的所有元素。（描述于 Sequence。）

void deque::resize(size_type n, const T& t = T())

将 deque 的大小改变为 n。（描述于 Sequence。）

※ **template <class InputIterator>**
void deque::assign(InputIterator first, InputIterator last)

相当于删除 *this 中的所有元素，再以 [first,last) 内的元素取而代之。

※ **void deque::assign(size_type n, const T& x)**

相当于删除 *this 中的所有元素，再以 n 个 x 复制品取而代之。

bool operator==(const deque&, const deque&)

验证两个 deques 是否相等。（描述于 Forward Container。）

bool operator<(const deque&, const deque&)

以字典排列顺序比较两个 deques。（描述于 Forward Container。）

16.2 Associative Containers（关联式容器）

C++ standard 提供四种 Associative Container classes：set、map、multiset 和 multimap。这四个 classes 都是 Sorted Associative Container 的 models。

C++ standard 并未包含 Hashed Associative Containers，然而 hash tables（散列表）却是很常见的一种扩充功能。早期的 STL hash table 版本是由 Javier Barreiro 与 David Musser 撰写，另一个版本由 Bob Fraley 撰写。这些版本在 "Hash Tables for the Standard Template Library" [BFM95] 文档中有说明。

目前最广为流行的 STL hash table 是 SGI 的版本。本章提供了 C++ standard 所定义的四种 Sorted Associated Containers 的说明，以及 SGI STL 所定义的四种 Hashed Associative Containers 的说明。

16.2.1 set

set<Key, Compare, Allocator>

set 是一种 Sorted Associative Container，能够存储型别为 Key 的 objects。它是一种 Simple Associative Container，意指其 value type 与其 key type 都是 Key。它同时也是 Unique Associative Container，表示没有两个元素相同。

由于 set 是一个 Sorted Associative Container，其元素一定会以递增顺序排序。Template 参数 Compare 定义了这种顺序关系。

set 与 multiset 特别满足泛型的「集合算法」（set_union、set_intersection 等等）。原因有二：第一，集合算法要求其引数为 sorted ranges，而 Sorted Associative Container 保证 set 和 multiset 一定满足这项条件；第二，这些算法的 output range 一定是排好序的，而且安插一个 sorted range 到 set 或 multiset 内是十分快速的行为。Sorted Associative Container 保证安插一个 sorted range 只需线性时间即可。insert_iterator adapter 可以使「集合算法的输出结果安插至 set 之中」的动作变得格外方便。

和 list（p.441）一样，set 具有数个重要性质：新元素的安插并不会造成既有元素的 iterators 失效，从 set 中删除元素也不会令任何 iterators 失效 — 当然被移除元素的 iterator 除外。

➢ **样例**

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  const int N = 6;
  const char* a[N] = {"isomer", "ephemeral", "prosaic",
                      "nugatory", "artichoke", "serif"};
```

```
  const char* b[N] = {"flat", "this", "artichoke",
                      "frigate", "prosaic", "isomer"};

  set<const char*, ltstr> A(a, a + N);
  set<const char*, ltstr> B(b, b + N);
  set<const char*, ltstr> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<const char*>(cout, " "),
            ltstr());
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<const char*>(cout, " "),
                   ltstr());
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()),
                 ltstr());
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 set 定义于头文件 <set.h>。C++ standard 将它声明于 <set>。

➢ Template 参数

Key	set 的 key type 和 value type。这个参数也被定义为 set::key_type 和 set::value_type。
Compare	键值的比较函数，是一个 Strict Weak Ordering（p.119），其引数型别为 key_type。如果其第一引数小于第二引数，便返回 true，否则返回 false。此参数也被定义为 set::key_compare 和 set::value_compare。**缺省值：** less<Key>

`Allocator` `set` 的 allocator，用来管理所有内部内存。**缺省值：** `allocator<Key>`

➢ 隶属何种 concept

Sorted Associative Container（p.156）、Simple Associative Container（p.153）、Unique Associative Container（p.149）。

➢ 型别条件

- `Key` 为 Assignable 的 model。
- `Compare` 为 Strict Weak Ordering 的 model。
- `Compare` 的 value type 为 `Key`。
- `Allocator` 是一个 Allocator model，其 value type 为 `Key`。

➢ Public Base Classes

无

➢ 成员

`set` 的某些成员并未定义于 Sorted Associative Container、Simple Associative Container、Unique Associative Container 需求条件中，它们都是 `set` 所特有，皆以 ※ 符号标示之。

`set::value_type`

存储于此 `set` 之内的 object 型别，亦即 `Key`。（描述于 Container。）

`set::key_type`

与 `value_type` object 相应的键值型别。亦即 `Key`。（描述于 Associative Container。）

`set::key_compare`

一个 function object，用来比较两个键值（keys）顺序，是一个 Strict Weak Ordering。（描述于 Sorted Associative Container。）

`set::value_compare`

一个 function object，用来比较两个实值（values）顺序，与 `key_compare` 型别相同。（描述于 Sorted Associative Container。）

`set::pointer`

`Key` 的指针。（描述于 Container。）

`set::const_pointer`

`const Key` 的指针。（描述于 Container。）

`set::reference`

`Key` 的 reference。（描述于 Container。）

set::const_reference

const Key 的 reference。（描述于 Container。）

set::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

set::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

set::iterator

set 的 iterator type，是一个不可变的（constant）Bidirectional Iterator。set 不具有可变的（mutable）iterator type。虽然你可以将元素安插或移除于 set，但这些元素都不能被更改。（描述于 Container。）

set::const_iterator

set 的 constant iterator type，同样是不可变的（constant）Bidirectional Iterator。可能和 set::iterator 相同。（描述于 Container。）

set::reverse_iterator

用于逆向遍历的一种 set iterator，是一个不可变的（constant）Bidirectional Iterator。（描述于 Reversible Container。）

set::const_reverse_iterator

用于逆向遍历的一种 set iterator，也是一个不可变的（constant）Bidirectional Iterator。（描述于 Reversible Container。）

iterator set::begin() const

返回一个 iterator，指向此 set 的开头。（描述于 Container。）

iterator set::end() const

返回一个 iterator，指向此 set 的尾端。（描述于 Container。）

reverse_iterator set::rbegin() const

返回一个 reverse_iterator，指向被倒转之 set 的开头。（描述于 Reversible Container。）

reverse_iterator set::rend() const

返回一个 reverse_iterator，指向被倒转之 set 的尾端。（描述于 Reversible Container。）

size_type set::size() const

返回 set 的元素个数。（描述于 Container。）

size_type set::max_size() const

返回 set 的最大可能元素个数。（描述于 Container。）

bool set::empty() const

当且仅当（if and only if）set 的大小为零，就返回 true。（描述于 Container。）

key_compare set::key_comp() const

返回 set 所使用的 key_compare object。（描述于 Sorted Associative Container。）

value_compare set::value_comp() const

返回 set 所使用的 value_compare object。（描述于 Sorted Associative Container。）

expilcit set::set(const key_compare& comp = key_compare(),
const Allocator& A = Allocator())

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个空的 set。（描述于 Sorted Associative Container。）

template <class InputIterator>
set::set(InputIterator f, InputIterator l,
const key_compare& comp = key_compare(),
const Allocator& A = Allocator())

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个 set，使它拥有 range[f,l) 的一份复制品。（描述于 Sorted Associative Container。）

set::set(const set&)

copy constructor。（描述于 Container。）

set::~set()

destructor。（描述于 Container。）

set& set::operator=(const set&)

assignment 操作符。（描述于 Container。）

※ **allocator_type set::get_allocator() const**

返回 set 构造时所搭配之 allocator 的一份复制品。

void set::swap(set&)

将两个 sets 的内容互换。（描述于 Container。）

pair<iterator, bool>set::insert(const value_type& x)

将 x 安插到 set 之中。如果 x 的确被安插了，返回值（pair）的第二部分为 true，如果 x 已存在所以没有被安插，返回值(pair)的第二部分为 false。（描述于 Unique Associative Container。）

iterator set::insert(iterator pos, const value_type& x)

以 pos 当作插入点，安插 x 于 set 之中。（描述于 Sorted Associative Container。）

template <class InputIterator>void set::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 set 之中。（描述于 Unique Associative Container。）

void set::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type set::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数 — 非 0 即 1。（描述于 Associative Container。）

void set::erase(iterator f, iterator l)

删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator set::find(const key_type& k) const

寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type set::count(const key_type& k) const

计算「键值为 k」的元素个数。返回值非 0 即 1。（描述于 Associative Container。）

iterator set::lower_bound(const key_type& k) const

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

iterator set::upper_bound(const key_type& k) const

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

pair<iterator, iterator> set::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

bool operator==(const set&, const set&)

验证两个 sets 的相等性。（描述于 Forward Container。）

bool operator<(const set&, const set&)

以字典排列顺序比较两个 sets。（描述于 Forward Container。）

16.2.2 map

map<Key, T, Compare, Allocator>

map 是一种 Sorted Associative Container，可将型别为 Key 的 objects 与型别为 T 的 objects 系结在一起。它是一种 Pair Associative Container，表示其 value type 为 pair<const Key, T>。它同时也是 Unique Associative Container，意指没有两个元素具有相同的 key。

map 与 set 很类似。主要的差别在于 set 是一种 Simple Associative Container（其 value type 与 key type 相同），而 map 是一种 Pair Associative Container（其 value type 是「key type 加上其他相关数据」）。这种差异造成 set 无法区分 iterator 与 const_iterator，而 map 却能够。

和 list（p.441）一样，map 具有一个重要性质：新元素的安插不会造成既有元素的 iterators 的失效。自 map 中删除元素也不会造成任何 iterators 失效 — 除了被移除元素的 iterator 外。

➢ **样例**

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  map<const char*, int, ltstr> days;

  days["january"] = 31;
  days["february"] = 28;
  days["march"] = 31;
  days["april"] = 30;
  days["may"] = 31;
  days["june"] = 30;
  days["july"] = 31;
  days["august"] = 31;
  days["september"] = 30;
  days["october"] = 31;
  days["november"] = 30;
  days["december"] = 31;

  cout << "june -> " << days["june"] << endl;
  map<const char*, int, ltstr>::iterator cur = days.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is "
       << (*prev).first << endl;
  cout << "Next (in alphabetical order) is "
       << (*next).first << endl;
}
```

当你以 map 作为一个关联式数组（associated array）时，很自然地会使用 operator[] 来检索元素，因为这样看起来就很类似一般数组的索引法。请你牢牢记住，operator[] 只是一种捷径写法。没有什么是 operatr[] 可以完成而 find 加上 insert 却办不到的。反之则成立：operator[] 无法让你知道你是否正在「更改一个既有值」或「安插一个新值」，但 find 却办得到。

```
int main() {
  map<string, int> M;
```

```
M.insert(make_pair("A", 17));
M.insert(make_pair("B", 74));

if (M.find("Z") == M.end())
  cout << "Not found: Z" << endl;

                // 安插新值于此 map 中
pair<map<string, int>::iterator, bool> p = M.insert(make_pair("C", 4));
assert(p.second);

                // 试着再安插一个新值。这次不会成功，
                // 因为 map 已经拥有一个元素其键值为 "B"。
p = M.insert(make_pair("B", 3));
assert(!p.second);

                // 改变与 B 关联的值
  cout << "Value associated with B: " << p.first->second << endl;
  p.first->second = 7;
  cout << "Value associated with B: " << p.first->second << endl;
}
```

➢ 何处定义

HP STL 将 map 定义于头文件 <map.h>。C++ standard 将它声明于 <map>。

➢ Template 参数

Key	map 的 key type。此参数也被定义为 map::key_type。
T	map 的 mapped type。此参数也被定义为 map::mapped_type。
Compare	键值比较函数，是一个 Strict Weak Ordering（p.119），其引数型别为 key_type。如果其第一引数小于第二引数，便返回 true，否则返回 false。此参数也被定义为嵌套型别 map::key_compare。**缺省值：** less<Key>
Allocator	map 的 allocator，用来管理所有内部内存。**缺省值：** allocator<pair<const Key, T> >

➢ 隶属何种 concept

Sorted Associative Container（p.156）Pair Associative Container（p.155）、Unique Associative Container（p.149）。

➢ 型别条件

- Key 为 Assignable 的 model。
- T 为 Assignable 的 model。

- Compare 为 Strict Weak Ordering 的 model。
- Compare 的 value type 为 Key。
- Allocator 是一个 Allocator model，其 value type 与 map 的 value type 相同。

➢ Public Base Classes

无

➢ 成员

map 的某些成员并未定义于 Sorted Associative Container、Pair Associative Container、Unique Associative Container 需求条件中，它们都是 map 所特有，皆以 ※ 符号标示之。

map::key_type

map 的 key type，Key。（描述于 Associative Container。）

map::mapped_type

「与键值相关联之 object」的型别。这个 mapped type 是为 T。（描述于 Pair Associative Container。）

map::value_type

存储于此 map 之内的 object 型别，亦即 pair<const Key, T>。（描述于 Pair Associative Container。）

map::key_compare

一个 function object，用来比较两个键值（keys）顺序，是一个 Strict Weak Ordering。（描述于 Sorted Associative Container。）

map::value_compare

一个 function object，用来比较两个实值（values）顺序。（相当于「先取出实值（value）的 keys，然后再比较那些 keys」）（描述于 Sorted Associative Container。）

map::pointer

map::value_type 的指针。（描述于 Container。）

map::const_pointer

const map::value_type 的指针。（描述于 Container。）

map::reference

map::value_type 的 reference。（描述于 Container。）

map::const_reference

const map::value_type 的 reference。（描述于 Container。）

map::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

map::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

map::iterator

map 的 iterator type，这是一个 Bidirectional Iterator，但它并非可变的（mutable），因为 map::value_type 并不隶属于 Assignable（如果 i 的型别为 map::iterator，你不能写 *i = x）。但它也不是完全的 constant iterator，因为它可用来更改它所指的 object。是的，你可以这样写：i->second = x。（描述于 Container。）

map::const_iterator

map 的 constant iterator type，是一个不可变的（constant）Bidirectional Iterator。可能和 iterator 相同。（描述于 Container 中。）

map::reverse_iterator

用于逆向遍历的一种 map iterator。（描述于 Reversible Container。）

map::const_reverse_iterator

用于逆向遍历的一种 map constant iterator。（描述于 Reversible Container。）

iterator map::begin()

返回一个 iterator，指向 map 的开头。（描述于 Container。）

iterator map::end()

返回一个 iterator，指向 map 的尾端。（描述于 Container。）

const_iterator map::begin() const

返回一个 const_iterator，指向 map 的开头。（描述于 Container。）

const_iterator map::end() const

返回一个 const_iterator，指向 map 的尾端。（描述于 Container。）

reverse_iterator map::rbegin()

返回一个 reverse_iterator，指向被倒转之 map 的开头。（描述于 Reversible Container。）

reverse_iterator map::rend()

返回一个 reverse_iterator，指向被倒转之 map 的尾端。（描述于 Reversible Container。）

const_reverse_iterator map::rbegin() const

返回一个 const_reverse_iterator，指向被倒转之 map 的开头。（描述于 Reversible Container。）

const_reverse_iterator map::rend() const

返回一个 const_reverse_iterator，指向被倒转之 map 的尾端。（描述于 Reversible Container。）

size_type map::size() const

返回 map 的元素个数。（描述于 Container。）

size_type map::max_size() const

返回 map 的最大可能元素个数。（描述于 Container。）

bool map::empty() const

当且仅当（if and only if）map 的大小为零，就返回 true。（描述于 Container。）

key_compare map::key_comp() const

返回 map 所使用的 key_compare object。（描述于 Sorted Associative Container。）

value_compare map::value_comp() const

返回 map 所使用的 value_compare object。（描述于 Sorted Associative Container。）

expilcit map::map(const key_compare& comp = key_compare(),
const Allocator& A = Allocator())

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个空的 map。（描述于 Sorted Associative Container。）

```
template <class InputIterator>
map::map(InputIterator f, InputIterator l,
         const key_compare& comp = key_compare(),
         const Allocator& A = Allocator())
```

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个 map，使它拥有 range[f,l) 的一份复制品。（描述于 Sorted Associative Container。）

map::map(const map&)

copy constructor。（描述于 Container。）

map::~map()

destructor。（描述于 Container。）

map& map::operator=(const map&)

assignment 操作符。（描述于 Container。）

※ **allocator_type map::get_allocator() const**

返回 map 构造时所搭配之 allocator 的一份复制品。

void map::swap(map&)

将两个 maps 的内容互换。（描述于 Container。）

pair<iterator, bool> map::insert(const value_type& x)

将 x 安插到 map 之中。如果 x 的确被安插了，返回值（pair）的第二部分为 true，如果 x 已存在所以没有被安插，返回值(pair)的第二部分为 false。（描述于 Unique Associative Container。）

iterator map::insert(iterator pos, const value_type& x)

以 pos 当作插入点，安插 x 于 map 之中。（描述于 Sorted Associative Container。）

template <class InputIterator>
void map::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 map 之中。（描述于 Unique Associative Container。）

void map::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type map::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数 — 非 0 即 1。（描述于 Associative Container。）

void map::erase(iterator f, iterator l)

删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator map::find(const key_type& k)

寻找「键值为 k」的元素。（描述于 Associative Container。）

const_iterator map::find(const key_type& k) const

寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type map::count(const key_type& k) const

计算「键值为 k」的元素个数。返回值非 0 即 1。（描述于 Associative Container。）

iterator map::lower_bound(const key_type& k)

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

const_iterator map::lower_bound(const key_type& k) const

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

iterator map::upper_bound(const key_type& k)

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

const_iterator map::upper_bound(const key_type& k) const

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

pair<iterator, iterator> map::equal_range(const key_type& k)

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

pair<const_iterator, const_iterator> map::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

※ **mapped_type& map::operator[](const key_type&)**

由于 map 是一个 Unique Associative Container，故同一键值在 map 中最多出现一次。operator[] 即是返回该键值所关联的 object 的 reference。如果 map 未含这样的 object，operator[] 会将缺省对象 mapped_type() 安插进去。（注意，返回型别为 mapped_type& 而非 value_type&。也就是说operator[] 并非返回 map 元素的 reference，而是返回该元素的某一部分的 reference。）

你可能会奇怪为什么 operator[] 没有 const 版本。这并不是意外的疏忽。问题是，当 m 未含「键值为 k」的任何元素时，m[k] 会有什么作为呢？由于 operator[] 在这样的状况下会安插新元素，所以它不能被定义为 const。

严格地说 operator[] 是多余的。它除了表现一种缩写状态，就没有别的了。表达式 m[k] 等价于 m.insert(value_type(k, mapped_type())).first->second

bool operator==(const map&, const map&)

验证两个 maps 的相等性。（描述于 Forward Container。）

bool operator<(const map&, const map&)

以字典排列顺序比较两个 maps。（描述于 Forward Container。）

16.2.3 multiset

multiset<Key, Compare, Allocator>

multiset 是一种 Sorted Associative Container，能够存储型别为 Key 的 objects。它是一种 Simple Associative Container，意指其 value type 与其 key type 都是 Key。它同时也是 Multiple Associative Container，表示可以容纳两个或更多个相同元素，这是 set 与 multiset 唯一不同之处。

由于 multiset 是一个 Sorted Associative Container，其元素一定会以递增顺序排序。Template 参数 Compare 定义了这种顺序关系。

和 list（p.441）一样，multiset 具有数个重要性质：新元素的安插并不会造成既有元素的 iterators 失效，从 set 中删除元素也不会令任何 iterators 失效 — 当然被移除元素的 iterator 除外。

➢ **样例**

```
int main()
{
    const int N = 10;
    int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
    int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};
```

```
  multiset<int> A(a, a + N);
  multiset<int> B(b, b + N);
  multiset<int> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<int>(cout, " "));
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()));
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

➢ 何处定义

HP STL 将 multiset 定义于头文件 <multiset.h>。C++ standard 将它声明于 <set>。

➢ Template 参数

Key	multiset 的 key type 和 value type。这个参数也被定义为嵌套型别 multiset::key_type 和 multiset::value_type。
Compare	key 的比较函数，是一个 Strict Weak Ordering（p.119），其引数型别为 key_type。如果其第一引数小于第二引数，便返回 true，否则返回 false。此参数也被定义为嵌套型别 multiset::key_compare 和 multiset::value_compare。**缺省值**：less<Key>
Allocator	multiset 的 allocator，用来管理所有内部内存。**缺省值**：allocator<Key>

➢ 隶属何种 concept

Sorted Associative Container（p.156）、Simple Associative Container（p.153）、Multiple Associative Container（p.152）。

➢ 型别条件

- Key 为 Assignable 的 model。
- Compare 为 Strict Weak Ordering 的 model。
- Compare 的 value type 为 Key。
- Allocator 是一个 Allocator model，其 value type 为 Key。

➢ Public Base Classes

无

➢ 成员

multiset 的某些成员并未定义于 Sorted Associative Container、Simple Associative Container、Multiple Associative Container 需求条件中，它们都是 multiset 所特有，皆以 ※ 符号标示之。

multiset::value_type

存储于此 multiset 之内的 object 型别，亦即 Key。（描述于 Container。）

multiset::key_type

与 value_type object 相应的键值型别。亦即 Key。（描述于 Associative Container。）

multiset::key_compare

一个 function object，用来比较两个键值（keys）顺序，是一个 Strict Weak Ordering。（描述于 Sorted Associative Container。）

multiset::value_compare

一个 function object，用来比较两个实值（values）顺序，与 key_compare 型别相同。（描述于 Sorted Associative Container。）

multiset::pointer

Key 的指针。（描述于 Container。）

multiset::const_pointer

const Key 的指针。（描述于 Container。）

multiset::reference

Key 的 reference。（描述于 Container。）

multiset::const_reference

const Key 的 reference。（描述于 Container。）

multiset::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

multiset::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

multiset::iterator

multiset 的 iterator type，是一个不可变的（constant）Bidirectional Iterator。（multiset 不具有可变的（mutable）iterator type。虽然你可以将元素安插或移除于 multiset，但这些元素都不能被更改。）（描述于 Container。）

multiset::const_iterator

multiset 的 constant iterator type，同样是不可变的（constant）Bidirectional Iterator。可能和 multiset::iterator 相同。（描述于 Container。）

multiset::reverse_iterator

用于逆向遍历的一种 multiset iterator，是一个不可变的（constant）Bidirectional Iterator。（描述于 Reversible Container。）

multiset::const_reverse_iterator

用于逆向遍历的一种 multiset iterator，也是一个不可变的（constant）Bidirectional Iterator。（描述于 Reversible Container。）

iterator multiset::begin() const

返回一个 iterator，指向此 multiset 的开头。（描述于 Container。）

iterator multiset::end() const

返回一个 iterator，指向此 multiset 的尾端。（描述于 Container。）

reverse_iterator multiset::rbegin() const

返回一个 reverse_iterator，指向被倒转之 multiset 的开头。（描述于 Reversible Container。）

reverse_iterator multiset::rend() const

返回一个 reverse_iterator，指向被倒转之 multiset 的尾端。（描述于 Reversible Container。）

size_type multiset::size() const

返回 multiset 的元素个数。（描述于 Container。）

size_type multiset::max_size() const

返回 multiset 的最大可能元素个数。（描述于 Container。）

bool multiset::empty() const

当且仅当（if and only if）multiset 的大小为零，就返回 true。（描述于 Container。）

key_compare multiset::key_comp() const

返回 multiset 所使用的 key_compare object。（描述于 Sorted Associative Container。）

value_compare multiset::value_comp() const

返回 multiset 所使用的 value_compare object。（描述于 Sorted Associative Container。）

expilcit multiset::multiset(const key_compare& comp = key_compare(),
const Allocator& A = Allocator())

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个空的 multiset。（描述于 Sorted Associative Container。）

template <class InputIterator>
multiset::multiset(InputIterator f, InputIterator l,
const key_compare& comp = key_compare(),
const Allocator& A = Allocator())

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个 multiset，使它拥有 range[f,l) 的一份复制品。（描述于 Sorted Associative Container。）

multiset::multiset(const multiset&)

copy constructor。（描述于 Container。）

multiset::~multiset()

destructor。（描述于 Container。）

multiset& multiset::operator=(const multiset&)

assignment 操作符。（描述于 Container。）

※ **allocator_type multiset::get_allocator() const**

返回 multiset 构造时所搭配之 allocator 的一份复制品。

void multiset::swap(multiset&)

将两个 multisets 的内容互换。（描述于 Container。）

iterator multiset::insert(const value_type& x)

将 x 安插到 multiset 之中。（描述于 Multiple Associative Container。）

iterator multiset::insert(iterator pos, const value_type& x)

以 pos 当作插入点，安插 x 于 multiset 之中。（描述于 Sorted Associative Container。）

template <class InputIterator>

void multiset::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 multiset 之中。（描述于 Multiple Associative Container。）

void multiset::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type multiset::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数。（描述于 Associative Container。）

void multiset::erase(iterator f, iterator l)

删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator multiset::find(const key_type& k) const

寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type multiset::count(const key_type& k) const

计算「键值为 k」的元素个数。（描述于 Associative Container。）

iterator multiset::lower_bound(const key_type& k) const

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

iterator multiset::upper_bound(const key_type& k) const

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

pair<iterator, iterator> multiset::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

bool operator==(const multiset&, const multiset&)

验证两个 multisets 的相等性。（描述于 Forward Container。）

bool operator<(const multiset&, const multiset&)

以字典排列顺序比较两个 multisets。（描述于 Forward Container。）

16.2.4 multimap

multimap<Key, T, Compare, Allocator>

multimap 是一种 Sorted Associative Container，可将型别为 Key 的 objects 与型别为 T 的 objects 系结在一起。它是一种 Pair Associative Container，表示其 value type 为 pair<const Key, T>。它同时也是 Multiple Associative Container，意指可容纳两个或更多的相同元素（这是 map 与 multimap 唯一不同之处）。也就是说，multimap 并非将型别为 Key 的键值对映到单一的 T object，而是对映到一个或多个 T objects 身上。

由于 multimap 是一个 Sorted Associative Container，其元素一定以递增顺序排列。Template 参数 Compare 负责定义此一顺序关系。

和 list（p.441）一样，multimap 具有一个重要性质：新元素的安插不会造成既有元素的 iterators 的失效。自 multimap 中删除元素也不会造成任何 iterators 失效 — 除了被移除元素的 iterator 外。

➢ 样例

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
```

```
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  multimap<const char*, int, ltstr> m;

  m.insert(make_pair("a", 1));
  m.insert(make_pair("c", 2));
  m.insert(make_pair("b", 3));
  m.insert(make_pair("b", 4));
  m.insert(make_pair("a", 5));
  m.insert(make_pair("b", 6));

  cout << "Number of elements with key a: " << m.count("a") << endl;
  cout << "Number of elements with key b: " << m.count("b") << endl;
  cout << "Number of elements with key c: " << m.count("c") << endl;

  cout << "Elements in m: " << endl;
  for (multimap<const char*, int, ltstr>::iterator it = m.begin();
       it != m.end();
       ++it)
    cout << "  [" << (*it).first << ", " << (*it).second << "]" << endl;
}
```

➢ 何处定义

HP STL 将 `multimap` 定义于头文件 `<multimap.h>`。C++ standard 将它声明于 `<map>`。

➢ Template 参数

`Key`	`multimap` 的 key type。此参数也被定义为 `multimap::key_type`。
`T`	`multimap` 的 mapped type。此参数也被定义为 `multimap::mapped_type`。
`Compare`	键值比较函数，是一个 Strict Weak Ordering（p.119），其引数型别为 `key_type`。如果其第一引数小于第二引数，便返回 `true`，否则返回 `false`。此参数也被定义为嵌套型别 `multimap::key_compare`。**缺省值：** `less<Key>`
`Allocator`	`multimap` 的 allocator，用来管理所有内部内存。**缺省值：** `allocator<pair<const Key, T> >`

➢ 隶属何种 concept

Sorted Associative Container（p.156）Pair Associative Container（p.155）、Multiple Associative Container（p.152）。

➢ 型别条件

- Key 为 Assignable 的 model。
- T 为 Assignable 的 model。
- Compare 为 Strict Weak Ordering 的 model。
- Compare 的 value type 为 Key。
- Allocator 是一个 Allocator model，其 value type 与 multimap 的 value type 相同。

➢ Public Base Classes

无

➢ 成员

multimap 的某些成员并未定义于 Sorted Associative Container、Pair Associative Container、Multiple Associative Container 需求条件中，它们都是 multimap 所特有，皆以 ※ 符号标示之。

multimap::key_type

multimap 的 key type，Key。（描述于 Associative Container。）

multimap::mapped_type

「与键值相关联之 object」的型别。这个 mapped type 是为 T。（描述于 Pair Associative Container。）

multimap::value_type

存储于此 multimap 之内的 object 型别，亦即 pair<const Key, T>。（描述于 Pair Associative Container。）

multimap::key_compare

一个 function object，用来比较两个键值（keys）顺序，是一个 Strict Weak Ordering。（描述于 Sorted Associative Container。）

multimap::value_compare

一个 function object，用来比较两个实值（values）顺序。（相当于「先取出实值（value）的 keys，然后再比较那些 keys」）（描述于 Sorted Associative Container。）

multimap::pointer

multimap::value_type 的指针。（描述于 Container。）

multimap::const_pointer

const multimap::value_type 的指针。（描述于 Container。）

multimap::reference

multimap::value_type 的 reference。（描述于 Container。）

multimap::const_reference

const multimap::value_type 的 reference。（描述于 Container。）

multimap::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

multimap::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

multimap::iterator

multimap 的 iterator type，这是一个 Bidirectional Iterator，但它并非可变的（mutable），因为 multimap:: value_type 并不隶属于 Assignable（如果 i 的型别为 multimap::iterator，你不能写 *i = x）。但它也不是完全的 constant iterator，因为它可用来更改它所指的 object。是的，你可以这样写： i->second = x。（描述于 Container。）

multimap::const_iterator

multimap 的 constant iterator type，是一个不可变的（constant）Bidirectional Iterator。可能和 iterator 相同。（描述于 Container 中。）

multimap::reverse_iterator

用于逆向遍历的一种 multimap iterator。（描述于 Reversible Container。）

multimap::const_reverse_iterator

用于逆向遍历的一种 multimap constant iterator。（描述于 Reversible Container。）

iterator multimap::begin()

返回一个 iterator，指向 multimap 的开头。（描述于 Container。）

iterator multimap::end()

返回一个 iterator，指向 multimap 的尾端。（描述于 Container。）

const_iterator multimap::begin() const

返回一个 const_iterator，指向 multimap 的开头。（描述于 Container。）

const_iterator multimap::end() const

返回一个 const_iterator，指向 multimap 的尾端。（描述于 Container。）

reverse_iterator multimap::rbegin()

返回一个 reverse_iterator，指向被倒转之 multimap 的开头。（描述于 Reversible Container。）

reverse_iterator multimap::rend()

返回一个 reverse_iterator，指向被倒转之 multimap 的尾端。（描述于 Reversible Container。）

const_reverse_iterator multimap::rbegin() const

返回一个 const_reverse_iterator，指向被倒转之 multimap 的开头。（描述于 Reversible Container。）

const_reverse_iterator multimap::rend() const

返回一个 const_reverse_iterator，指向被倒转之 multimap 的尾端。（描述于 Reversible Container。）

size_type multimap::size() const

返回 multimap 的元素个数。（描述于 Container。）

size_type multimap::max_size() const

返回 multimap 的最大可能元素个数。（描述于 Container。）

bool multimap::empty() const

当且仅当（if and only if）multimap 的大小为零，就返回 true。（描述于 Container。）

key_compare multimap::key_comp() const

返回 multimap 所使用的 key_compare object。（描述于 Sorted Associative Container。）

value_compare multimap::value_comp() const

返回 multimap 所使用的 value_compare object。（描述于 Sorted Associative Container。）

```
expilcit multimap::multimap(const key_compare& comp = key_compare(),
                            const Allocator& A = Allocator())
```

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个空的 multimap。（描述于 Sorted Associative Container。）

```
template <class InputIterator>
multimap::multimap(InputIterator f, InputIterator l,
                   const key_compare& comp = key_compare(),
                   const Allocator& A = Allocator())
```

以 comp 作为 key 之比较准则，以 A 作为内存分配准则，产生一个 multimap，使它拥有 range[f,l) 的一份复制品。（描述于 Sorted Associative Container。）

multimap::multimap(const multimap&)

copy constructor。（描述于 Container。）

multimap::~multimap()

destructor。（描述于 Container。）

multimap& multimap::operator=(const multimap&)

assignment 操作符。（描述于 Container。）

※ **allocator_type multimap::get_allocator() const**

返回 multimap 构造时所搭配之 allocator 的一份复制品。

void multimap::swap(multimap&)

将两个 multimaps 的内容互换。（描述于 Container。）

iterator multimap::insert(const value_type& x)

将 x 安插到 multimap 之中。（描述于 Multiple Associative Container。）

iterator multimap::insert(iterator pos, const value_type& x)

以 pos 当作插入点，安插 x 于 multimap 之中。（描述于 Sorted Associative Container。）

template <class InputIterator>
void multimap::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 multimap 之中。（描述于 Multiple Associative Container。）

void multimap::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type multimap::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数。（描述于 Associative Container。）

void multimap::erase(iterator f, iterator l)

删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator multimap::find(const key_type& k)

寻找「键值为 k」的元素。（描述于 Associative Container。）

const_iterator multimap::find(const key_type& k) const

寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type multimap::count(const key_type& k) const

计算「键值为 k」的元素个数。（描述于 Associative Container。）

iterator multimap::lower_bound(const key_type& k)

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

const_iterator multimap::lower_bound(const key_type& k) const

寻找第一个「键值不小于 k」的元素。（描述于 Sorted Associative Container。）

iterator multimap::upper_bound(const key_type& k)

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

const_iterator multimap::upper_bound(const key_type& k) const

寻找第一个「键值大于 k」的元素。（描述于 Sorted Associative Container。）

pair<iterator, iterator> multimap::equal_range(const key_type& k)

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

pair<const_iterator, const_iterator>multimap::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Sorted Associative Container。）

bool operator==(const multimap&, const multimap&)

验证两个 multimaps 的相等性。（描述于 Forward Container。）

bool operator<(const multimap&, const multimap&)

以字典排列顺序比较两个 multimaps。（描述于 Forward Container。）

16.2.5 hash_set

hash_set<Key, HashFun, EqualKey, Allocator>

hash_set 是一种 Hashed Associative Container，存储着型别为 Key 的 objects。它是一种 Simple Associative Container，意指其 value type 和 key type 皆为 Key。它同时也是 Unique Associative Container，表示如果以 Binary Predicate EqualKey 进行比较，没有任何两个元素会相同。

在重视快速查找的应用程序中，hash_set class 很有用。然而如果元素必须以特定顺序排列的话，那么 set（p.461）比较适合。

➢ 样例

```
struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

void lookup(const hash_set<const char*, hash<const char*>, eqstr>& Set,
            const char* word)
{
  hash_set<const char*, hash<const char*>, eqstr>::const_iterator it
    = Set.find(word);
  cout << "  " << word << ": "
       << (it != Set.end() ? "present" : "not present")
       << endl;
}

int main()
{
  hash_set<const char*, hash<const char*>, eqstr> Set;
  Set.insert("kiwi");
  Set.insert("plum");
  Set.insert("apple");
  Set.insert("mango");
  Set.insert("apricot");
  Set.insert("banana");

  lookup(Set, "mango");
  lookup(Set, "apple");
  lookup(Set, "durian");
```

```
  // 输出为:
  //   mango: present
  //   apple: present
  //   durian: not present
}
```

➢ 何处定义

hash_set 并未出现于原始的 HP STL 中，也不在 C++ standard 规范之列。SGI STL 将它声明于头文件 <hash_set>。

➢ Template 参数

Key	hash_set 的 key type 和 value type。此参数也被定义成嵌套型别 hash_set::key_type 和 hash_set::value_type。
HashFun	hash_set 所使用的 Hash Function。此参数也被定义成嵌套型别 hash_set::hasher。**缺省值**: hash<Key>
EqualKey	此式用来检测 hash_set 的 key 是否相等。这是一个 Binary Predicate，此参数也被定义成嵌套型别 hash_set::key_equal。**缺省值**: equal_to<Key>
Allocator	hash_set 的 allocator，用来管理所有内部内存。**缺省值**: allocator<Key>

➢ 隶属何种 concept

Hashed Associative Container（p.161）、Simple Associative Container（p.153）、Unique Associative Container（p.149）。

➢ 型别条件

- Key 为 Assignable 的 model。
- HashFun 为 Hash Function 的 model。
- HashFun 的 value type 为 Key。
- EqualKey 为 Binary Predicate 的 model。
- EqualKey 是一种等价性（equivalence）关系。
- EqualKey 的 value type 为 Key。
- Allocator 是 Allocator 的 model，其 value type 为 Key。

➢ Public Base Classes

无

➢ 成员

hash_set 的某些成员并未定义于 Hashed Associative Container、Simple Associative Container、Unique Associative Container 需求条件中，它们是 hash_set 所特有，皆以 ※ 符号标示之。

hash_set::value_type

存储于此 hase_set 之内的 object 型别，亦即 Key。（描述于 Container。）

hash_set::key_type

与 value_type object 相应的键值型别。亦即 Key。（描述于 Associative Container。）

hash_set::hasher

hash_set 的 Hash Function。（描述于 Hashed Associative Container。）

hash_set::key_equal

一个 Binary Predicate，用来比较 key 是否相等(equality)。（描述于 Hashed Associative Container。）

hash_set::pointer

Key 的指针。（描述于 Container。）

hash_set::const_pointer

const Key 的指针。（描述于 Container。）

hash_set::reference

Key 的 reference。（描述于 Container。）

hash_set::const_reference

const Key 的 reference。（描述于 Container。）

hash_set::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

hash_set::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

hash_set::iterator

hash_set 的 iterator type，是一个不可变的（constant）Forward Iterator。（hash_set 不具有可变的（mutable）iterator type。（描述于 Container。）

hash_set::const_iterator

hash_set 的 constant iterator type，同样是不可变的（constant）Forward Iterator。可能和 hash_set::iterator 相同。（描述于 Container。）

iterator hash_set::begin() const

返回一个 iterator，指向此 hash_set 的开头。（描述于 Container。）

iterator hash_set::end() const

返回一个 iterator，指向此 hash_set 的尾端。（描述于 Container。）

size_type hash_set::size() const

返回 hash_set 的元素个数。（描述于 Container。）

size_type hash_set::max_size() const

返回 hash_set 的最大可能元素个数。（描述于 Container。）

bool hash_set::empty() const

当且仅当（if and only if）hash_set 的大小为零，就返回 true。（描述于 Container。）

size_type hash_set::bucket_count() const

返回 hash table 中的 buckets 个数。（描述于 Hashed Associative Container。）

void hash_set::resize(size_type n)

增加 bucket 的个数，使它至少为 n。（描述于 Hashed Associative Container。）

hasher hash_set::hash_funct() const

返回 hash_set 所使用之 hasher object。（描述于 Hashed Associative Container。）

key_equal hash_set::key_eq() const

返回 hash_set 所使用之 key_equal object。（描述于 Hashed Associative Container。）

```
expilcit hash_set::hash_set(size_type n = 0,
                            const hasher& h = hasher(),
                            const key_equal& k = key_equal(),
                            const Allocator& A = Allocator())
```

以指定之 hash function、key equality function 和 allocator，产生一个空的 hash_set，并至少具备 n 个 buckets。（描述于 Hashed Associative Container。）

```
template <class InputIterator>
hash_set::hash_set(InputIterator f, InputIterator l,
                   size_type n = 0,
                   const hasher& h = hasher(),
                   const key_equal& k = key_equal(),
                   const Allocator& A = Allocator())
```

以指定之 hash function、key equality function 和 allocator，产生一个 hash_set，使之拥有 [f,l) 的复制内容，并至少具备 n 个 buckets。（描述于 Hashed Associative Container。）

hash_set::hash_set(const hash_set&)

copy constructor。（描述于 Container。）

hash_set::~hash_set()

destructor。（描述于 Container。）

hash_set& hash_set::operator=(const hash_set&)

assignment 操作符。（描述于 Container。）

※ **allocator_type hash_set::get_allocator() const**

返回 hash_set 构造时所搭配之 allocator 的一份副本。

void hash_set::swap(hash_set&)

将两个 hash_sets 的内容互换。（描述于 Container。）

pair<iterator, bool> hash_set::insert(const value_type& x)

将 x 安插到 hash_set 之中。如果 x 的确被安插了，返回值（pair）的第二部分为 true，如果 x 已存在所以没有被安插，返回值（pair）的第二部分为 false。（描述于 Unique Associative Container。）

template <class InputIterator>
void hash_set::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 hash_set 中。（描述于 Unique Associative Container。）

void hash_set::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type hash_set::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数 — 非 0 即 1。（描述于 Associative Container。）

void hash_set::erase(iterator f, iterator l)

删除 range [f,l) 内的元素。（描述于 Associative Container。）

iterator hash_set::find(const key_type& k) const

寻找键值为 k 的元素。（描述于 Associative Container。）

size_type hash_set::count(const key_type& k) const

返回键值为 k 的元素个数。返回值非 0 即 1。（描述于 Associative Container。）

pair<iterator, iterator> hash_set::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

bool operator==(const hash_set&, const hash_set&)

验证两个 hash_set objects 是否相等。（描述于 Forward Container。）

16.2.6 hash_map

hash_map<Key, T, HashFun, EqualKey, Allocator>

hash_map 是一种 Hashed Associative Container，将 Key 型别的 objects 与 T 型别的 objects 联系在

一起。它是一种 Pair Associative Container，意指其 value type 为 pair<const Key, T>。它同时也是 Unique Associative Container，意指当我们以 EqualKey 作为比较标准时，没有任何两个元素拥有相同的 key。

在 hash_map 中以 key 来查询元素，效率很高，所以 hash_map 有助于「字典」或关联式数组的实现，其中 Key object 与 T object 之间相互有关联，但顺序无关紧要。如果元素必须以特定顺序排列，那么 map（p.466）比较适合。

➢ 样例

```
struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

int main()
{
  hash_map<const char*, int, hash<const char*>, eqstr> days;

  days["january"] = 31;
  days["february"] = 28;
  days["march"] = 31;
  days["april"] = 30;
  days["may"] = 31;
  days["june"] = 30;
  days["july"] = 31;
  days["august"] = 31;
  days["september"] = 30;
  days["october"] = 31;
  days["november"] = 30;
  days["december"] = 31;

  cout << "september -> " << days["september"] << endl;
  cout << "april     -> " << days["april"] << endl;
  cout << "june      -> " << days["june"] << endl;
  cout << "november  -> " << days["november"] << endl;
}
```

➢ 何处定义

hash_map 并未定义于原始 HP STL 中，也不在 C++ standard 规范之列。SGI STL 将它声明于头文件 <hash_map>。

➢ Template 参数

Key	hash_map 的 key type。此参数也被定义为 hash_map::key_type。
T	hash_map 的 mapped type（对映型别）。此参数也被定义成 hash_map::mapped_type。
HashFun	hash_map 所使用的 Hash Function。此参数也被定义成嵌套型别 hash_map::hasher。**缺省值**：hash<Key>
EqualKey	一个函数，被 hash_map 用来比较两个 keys 是否相等。这是一个 Binary Predicate，此参数也被定义成嵌套型别 hash_map::key_equal。 **缺省值**：equal_to<Key>
Allocator	hash_map 的 allocator，用来管理所有内部内存。 **缺省值**：allocator<pair<const Key, T> >

➢ 隶属何种 concept

Hashed Associative Container（p.161）Pair Associative Container（p.155）、Unique Associative Container（p.149）。

➢ 型别条件

- Key 为 Assignable 的 model。
- T 为 Assignable 的 model。
- HashFun 为 Hash Function 的 model。
- HashFun 的 value type 为 Key。
- EqualKey 为 Binary Predicate 的 model。
- EqualKey 是一种等价性（equivalence）关系。
- EqualKey 的 value type 为 Key。
- Allocator 为 Allocator 的 model。
- Allocator 的 value type 与 hash_map 的 value_type 相同。

➢ Public Base Classes

无

➢ 成员

hash_map 的某些成员并未定义于 Hashed Associative Container、Pair Associative Container、Unique Associative Container 需求条件中，它们是 hash_map 所特有，皆以 ※ 符号标示之。

hash_map::key_type

hash_map 的 key type，Key。（描述于 Associative Container。）

hash_map::mapped_type

与键值相关联之 object 的型别。这个 mapped type 是为 T。（描述于 Pair Associative Container。）

hash_map::value_type

存储于此 hash_map 之内的 object 型别，亦即 pair<const Key, T>。（描述于 Pair Associative Container。）

hash_map::hasher

hash_map 的 Hash Function。（描述于 Hashed Associative Container。）

hash_map::key_equal

一个 Binary Predicate，用来比较 key 是否相等（equality）。

hash_map::pointer

hash_map::value_type 的指针。（描述于 Container。）

hash_map::const_pointer

const hash_map::value_type 的指针。（描述于 Container。）

hash_map::reference

hash_map::value_type 的 reference。（描述于 Container。）

hash_map::const_reference

const hash_map::value_type 的 reference。（描述于 Container。）

hash_map::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

hash_map::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

hash_map::iterator

hash_map 的 iterator type，这是一个 Forward Iterator，但它并非可变的（mutable），因为 hash_map::value_type 并不隶属于 Assignable（如果 i 的型别为 hash_map::iterator，你不能写 *i=x）。但它也不是完全的 constant iterator，因为它可用来更改它所指的 object。是的，你可以这样写：i->second = x。（描述于 Container。）

hash_map::const_iterator

hash_map 的 constant iterator type，是一个不可变的（constant）Forward Iterator。（描述于 Container。）

iterator hash_map::begin()

返回一个 iterator，指向 hash_map 的开头。（描述于 Container。）

iterator hash_map::end()

返回一个 iterator，指向 hash_map 的尾端。（描述于 Container。）

const_iterator hash_map::begin() const

返回一个 const_iterator，指向 hash_map 的开头。（描述于 Container。）

const_iterator hash_map::end() const

返回一个 const_iterator，指向 hash_map 的尾端。（描述于 Container。）

size_type hash_map::size() const

返回 hash_map 的元素个数。（描述于 Container。）

size_type hash_map::max_size() const

返回 hash_map 的最大可能元素个数。（描述于 Container。）

bool hash_map::empty() const

当且仅当（if and only if）hash_map 的大小为零，就返回 true。（描述于 Container。）

size_type hash_map::bucket_count() const

返回 hash table 中的 buckets 个数。（描述于 Hashed Associative Container。）

void hash_map::resize(size_type n)

增加 bucket 的个数，使之至少为 n。（描述于 Hashed Associative Container。）

hasher hash_map::hash_funct() const

返回 hash_map 所使用的 hasher object。（描述于 Hashed Associative Container。）

key_equal hash_map::key_eq() const

返回 hash_map 所使用的 key_equal object。（描述于 Hashed Associative Container。）

```
expilcit hash_map::hash_map(size_type n = 0,
                            const hasher& h = hasher(),
                            const key_equal& k = key_equal(),
                            const Allocator& A = Allocator())
```

使用指定之 hash function、key equality function、allocator，产生一个至少具备 n 个 buckets 的空 hash_map。（描述于 Hashed Associative Container。）

```
template <class InputIterator>
hash_map::hash_map(InputIterator f, InputIterator l,
                   size_type n = 0,
                   const hasher& h = hasher(),
                   const key_equal& k = key_equal(),
                   const Allocator& A = Allocator())
```

使用指定之 hash function、key equality function、allocator，产生一个 hash_map，具有 range [f,l) 的复制内容且 bucket 数目至少为 n。（描述于 Hashed Associative Container。）

hash_map::hash_map(const hash_map&)

copy constructor。（描述于 Container。）

hash_map::~hash_map()

destructor。（描述于 Container。）

hash_map& hash_map::operator=(const hash_map&)

assignment 操作符。（描述于 Container。）

※ **allocator_type hash_map::get_allocator() const**

返回 hash_map 构造时所搭配之 allocator 的一份副本。

void hash_map::swap(hash_map&)

将两个 hash_maps 的内容互换。（描述于 Container。）

pair<iterator, bool> hash_map::insert(const value_type& x)

将 x 安插到 hash_map 之中。如果 x 的确被安插了，返回值（pair）的第二部分为 true，如果 x 已存在所以没有被安插，返回值（pair）的第二部分为 false。（描述于 Unique Associative Container。）

template <class InputIterator>
void hash_map::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 hash_map 中。（描述于 Unique Associative Container。）

void hash_map::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type hash_map::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数 — 非 0 即 1。（描述于 Associative Container。）

void hash_map::erase(iterator f, iterator l)

删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator hash_map::find(const key_type& k)

寻找「键值为 k」的元素。（描述于 Associative Container。）

const_iterator hash_map::find(const key_type& k) const

寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type hash_map::count(const key_type& k) const

计算「键值为 k」的元素个数。返回值非 0 即 1。（描述于 Associative Container。）

pair<iterator, iterator> hash_map::equal_range(const key_type& k)

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

pair<const_iterator, const_iterator>hash_map::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

※ **mapped_type& hash_map::operator[](const key_type&)**

由于 hash_map 是一个 Unique Associative Container，故同一键值在 hash_map 中最多出现一次。operator[] 即是返回该键值所关联的 object 的 reference。如果 hash_map 未含这样的 object，operator[] 会将缺省对象 mapped_type() 安插进去。（注意，返回型别为 mapped_type& 而非 value_type&。也就是说 operator[] 并非返回 hash_map 元素的 reference，而是返回该元素的某一部分的 reference。）

由于 operator[] 可能会将新元素安插于 hash_map，所以它没有 const 版本。

严格来说 operator[] 是多余的。它除了表现一种缩写状态，就没有别的了。表达式 m[k] 等价于 m.insert(value_type(k, mapped_type())).first->second

bool operator==(const hash_map&, const hash_map&)

验证两个 hash_maps 的相等性。（描述于 Forward Container。）

16.2.7 hash_multiset

hash_multiset<Key, HashFun, EqualKey, Allocator>

hash_multiset 是一种 Hashed Associative Container，能够存储型别为 Key 的 objects。它是一种 Simple Associative Container，意指其 value type 与 key type 都是 Key。它同时也是 Multiple Associative Container，表示可容纳两个或多个相同元素。（这是 hash_set 与 hash_multiset 唯一不同之处。）

在重视快速查找的应用程序中，hash_multiset 很有用。然而，如果元素需以特定顺序排列，那么 multiset（p.473）比较适合。

➢ 样例

```
struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

typedef hash_multiset<const char*, hash<const char*>, eqstr>
      Hash_Multiset;

void lookup(Hash_Multiset& Set, const char* word)
{
  int n_found = Set.count(word);
  cout << "  " << word << ": "
       << n_found << " "
       << (n_found == 1 ? "instance" : "instances")
       << endl;
}
```

```
int main()
{
  Hash_Multiset Set;

  Set.insert("mango");
  Set.insert("kiwi");
  Set.insert("apple");
  Set.insert("pear");
  Set.insert("kiwi");
  Set.insert("mango");
  Set.insert("pear");
  Set.insert("mango");
  Set.insert("apricot");
  Set.insert("banana");
  Set.insert("mango");

  lookup(Set, "mango");
  lookup(Set, "apple");
  lookup(Set, "durian");

  // 输出为:
  //   mango: 4 instances
  //   apple: 1 instance
  //   durian: 0 instances
}
```

➢ 何处定义

hash_multiset 并未定义于原始的 HP STL 中，也不在 C++ standard 规范之列。SGI STL 将它定义于头文件 <hash_set>。

➢ Template 参数

Key	hash_multiset 的 key type 和 value type。此参数也被定义成嵌套型别 hash_multiset::key_type 和 hash_multiset::value_type。
HashFun	hash_multiset 所使用的 Hash Function。此参数也被定义成嵌套型别 hash_multiset::hasher。**缺省值**：hash<Key>
EqualKey	这是一个 Binary Predicate，用来比较 hash_multiset 的 keys 是否相等。此参数也被定义成嵌套型别 hash_multiset::key_equal。**缺省值**：equal_to<Key>
Allocator	hash_multiset 的 allocator，用来管理所有内部内存。**缺省值**：allocator<Key>

➢ 隶属何种 concept

Hashed Associative Container（p.161）、Simple Associative Container（p.153）、Multiple Associative Container（p.152）。

➢ 型别条件

- Key 为 Assignable 的 model。
- HashFun 为 Hash Function 的 model。
- HashFun 的 value type 为 Key。
- EqualKey 为 Binary Predicate 的 model。
- EqualKey 是一种等价性性（equivalence）关系。
- EqualKey 的 value type 为 Key。
- Allocator 是 Allocator 的 model，其 value type 为 Key。

➢ Public Base Classes

无

➢ 成员

hash_multiset 的某些成员并未定义于 Hashed Associative Container、Simple Associative Container、Multiple Associative Container 需求条件中，它们是 hash_multiset 所特有，皆以 ※ 符号标示之。

hash_multiset::value_type

存储于此 hase_multiset 之内的 object 型别，亦即 Key。（描述于 Container。）

hash_multiset::key_type

与 value_type object 相应的键值型别。亦即 Key。（描述于 Associative Container。）

hash_multiset::hasher

hash_multiset 的 Hash Function。（描述于 Hashed Associative Container。）

hash_multiset::key_equal

一个 Binary Predicate，用来比较 key 是否相等(equality)。（描述于 Hashed Associative Container。）

hash_multiset::pointer

Key 的指针。（描述于 Container。）

hash_multiset::const_pointer

const Key 的指针。（描述于 Container。）

hash_multiset::reference

Key 的 reference。（描述于 Container。）

hash_multiset::const_reference

const Key 的 reference。（描述于 Container。）

hash_multiset::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

hash_multiset::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

hash_multiset::iterator

hash_multiset 的 iterator type，是一个不可变的（constant）Forward Iterator。（hash_multiset 不具有可变的（mutable）iterator type。（描述于 Container。）

hash_multiset::const_iterator

hash_multiset 的 constant iterator type，同样是不可变的（constant）Forward Iterator。可能和 hash_multiset::iterator 相同。（描述于 Container。）

iterator hash_multiset::begin() const

返回一个 iterator，指向此 hash_multiset 的开头。（描述于 Container。）

iterator hash_multiset::end() const

返回一个 iterator，指向此 hash_multiset 的尾端。（描述于 Container。）

size_type hash_multiset::size() const

返回 hash_multiset 的元素个数。（描述于 Container。）

size_type hash_multiset::max_size() const

返回 hash_multiset 的最大可能元素个数。（描述于 Container。）

bool hash_multiset::empty() const

当且仅当（if and only if）hash_multiset 的大小为零，就返回 true。（描述于 Container。）

size_type hash_multiset::bucket_count() const

返回 hash table 中的 buckets 个数。（描述于 Hashed Associative Container。）

void hash_multiset::resize(size_type n)

增加 buckets 的个数，使它至少为 n。（描述于 Hashed Associative Container。）

hasher hash_multiset::hash_funct() const

返回 hash_multiset 所使用之 hasher object。（描述于 Hashed Associative Container。）

key_equal hash_multiset::key_eq() const

返回 hash_multiset 所使用之 key_equal object。（描述于 Hashed Associative Container。）

```
expilcit hash_multiset::hash_multiset(size_type n = 0,
                                      const hasher& h = hasher(),
                                      const key_equal& k = key_equal(),
                                      const Allocator& A = Allocator())
```

以指定之 hash function、key equality function 和 allocator，产生一个空的 hash_multiset，并至少具备 n 个 buckets。（描述于 Hashed Associative Container。）

```
template <class InputIterator>
hash_multiset::hash_multiset(InputIterator f, InputIterator l,
                             size_type n = 0,
                             const hasher& h = hasher(),
                             const key_equal& k = key_equal(),
                             const Allocator& A = Allocator())
```

以指定之 hash function、key equality function 和 allocator，产生一个 hash_multiset，使之拥有 [f,l) 的复制内容，并至少具备 n 个 buckets。（描述于 Hashed Associative Container。）

hash_multiset::hash_multiset(const hash_multiset&)

copy constructor。（描述于 Container。）

hash_multiset::~hash_multiset()

destructor。（描述于 Container。）

hash_multiset& hash_multiset::operator=(const hash_multiset&)

assignment 操作符。（描述于 Container。）

※ **allocator_type hash_multiset::get_allocator() const**

返回 hash_multiset 构造时所搭配之 allocator 的一份副本。

void hash_multiset::swap(hash_multiset&)

将两个 hash_multisets 的内容互换。（描述于 Container。）

iteratr hash_multiset::insert(const value_type& x)

将 x 安插到 hash_multiset 之中。（描述于 Multiple Associative Container。）

```
template <class InputIterator>
```

void hash_multiset::insert(InputIterator f, InputIterator l)

将 range [f,l) 安插到 hash_multiset 中。（描述于 Multiple Associative Container。）

void hash_multiset::erase(iterator pos)

删除 pos 所指元素。（描述于 Associative Container。）

size_type hash_multiset::erase(const key_type& k)

删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数。（描述于 Associative Container。）

void hash_multiset::erase(iterator f, iterator l)

删除 range [f,l) 内的元素。（描述于 Associative Container。）

iterator hash_multiset::find(const key_type& k) const

寻找键值为 k 的元素。（描述于 Associative Container。）

size_type hash_multiset::count(const key_type& k) const

返回键值为 k 的元素个数。（描述于 Associative Container。）

pair<iterator, iterator>hash_multiset::equal_range(const key_type& k) const

找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

bool operator==(const hash_multiset&, const hash_multiset&)

验证两个 hash_multiset objects 是否相等。（描述于 Forward Container。）

16.2.8 hash_multimap

hash_multimap<Key, T, HashFun, EqualKey, Allocator>

hash_multimap 是一种 Hashed Associative Container，将型别为 Key 之 objects 与型别为 T 之 objects 系结在一起。它是一种 Pair Associative Container，意指其 value type 为 pair<const Key, T>。它同时也是 Multiple Associative Container，表示可容纳两个或多个相同元素（这是 hash_map 与 hash_multimap 唯一不同之处）。也就是说 hash_multimap 并非将 Key object 对映到单一一个 T object，而是对映到一个或多个 T objects。

在 hash_multimap 中以 key 来查询元素，速度很快，所以 hash_multimap 有助于「字典」或关联式数组的实现，其中 Key object 与 T object 之间相互有关联，但顺序无关紧要。如果元素必须以特定顺序排列，那么 multimap（p.466）比较适合。

➢ 样例

```
struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

typedef hash_multimap<const char*, int, hash<const char*>, eqstr>
      map_type;

vcid lookup(const map_type& Map, const char* str)
{
  cout << "  " << str << ": ";
  pair<map_type::const_iterator, map_type::const_iterator> p
     = Map.equal_range(str);
```

```
  for (map_type::const_iterator i = p.first; i != p.second; ++i)
     cout << (*i).second << " ";
  cout << endl;
}

int main()
{
  map_type M;

  M.insert(map_type::value_type("H", 1));
  M.insert(map_type::value_type("H", 2));
  M.insert(map_type::value_type("C", 12));
  M.insert(map_type::value_type("C", 13));
  M.insert(map_type::value_type("O", 16));
  M.insert(map_type::value_type("O", 17));
  M.insert(map_type::value_type("O", 18));
  M.insert(map_type::value_type("I", 127));

  lookup(M, "I");
  lookup(M, "O");
  lookup(M, "Rn");

  // 输出为:
  //   I: 127
  //   O: 16 18 17
  //   Rn:
}
```

➢ 何处定义

hash_multimap 并未定义于原始的 HP STL 之中，也不在 C++ standard 规范之列。SGI STL 将它声明于头文件 <hash_map>。

➢ Template 参数

Key	hash_multimap 的 key type。此参数也被定义成 hash_multimap::key_type。
T	hash_multimap 的 mapped type（对映型别）。此参数也被定义成 hash_multimap::mapped_type。
HashFun	hash_multimap 所使用的 Hash Function。此参数也被定义成嵌套型别 hash_multimap::hasher。**缺省值**：hash<Key>
EqualKey	一个函数，被 hash_multimap 用来比较两个 keys 是否相等。这是一个 Binary Predicate，此参数也被定义成嵌套型别 hash_multimap::key_equal。 **缺省值**：equal_to<Key>
Allocator	hash_multimap 的 allocator，用来管理所有内部内存。 **缺省值**：allocator<pair<const Key, T> >

➢ 隶属何种 concept

Hashed Associative Container（p.161）Pair Associative Container（p.155）、Multiple Associative Container（p.152）。

➢ 型别条件

- Key 为 Assignable 的 model。
- HashFun 为 Hash Function 的 model。
- HashFun 的 value type 为 Key。
- EqualKey 为 Binary Predicate 的 model。
- EqualKey 是一种等价性（equivalence）关系。
- EqualKey 的 value type 为 Key。
- Allocator 为 Allocator 的 model。
- Allocator 的 value type 与 hash_multimap 的 value_type 相同。

➢ Public Base Classes

无

➢ 成员

hash_multimap 的某些成员并未定义于 Hashed Associative Container、Pair Associative Container、Multiple Associative Container 需求条件中，它们都是 hash_multimap 所特有，皆以 ※ 符号标示之。

hash_multimap::key_type

hash_multimap 的 key type，Key。（描述于 Associative Container。）

hash_multimap::mapped_type

与键值相关联之 object 的型别。这个 mapped type 是为 T。（描述于 Pair Associative Container。）

hash_multimap::value_type

存储于此 hash_multimap 之内的 object 型别，亦即 pair<const Key, T>。（描述于 Pair Associative Container。）

hash_multimap::hasher

hash_multimap 的 Hash Function。（描述于 Hashed Associative Container。）

hash_multimap::key_equal

一个 Binary Predicate，用来比较 key 是否相等（equality）。

hash_multimap::pointer

hash_multimap::value_type 的指针。（描述于 Container。）

hash_multimap::const_pointer

const hash_multimap::value_type 的指针。（描述于 Container。）

hash_multimap::reference

hash_multimap::value_type 的 reference。（描述于 Container。）

hash_multimap::const_reference

const hash_multimap::value_type 的 reference。（描述于 Container。）

hash_multimap::size_type

一个无正负号的整数（integral）型别，通常是 size_t。（描述于 Container。）

hash_multimap::difference_type

一个有正负号的整数（integral）型别，通常是 ptrdiff_t。（描述于 Container。）

hash_multimap::iterator

hash_multimap 的 iterator type，这是一个 Forward Iterator，但它并非可变的（mutable），因为 hash_multimap::value_type 并不隶属于 Assignable（如果 i 的型别为 hash_multimap::iterator，你不能写 *i = x）。但它也不是完全的 constant iterator，因为它可用来更改它所指的 object。是的，你可以这样写：i->second = x。（描述于 Container。）

hash_multimap::const_iterator

hash_multimap 的 constant iterator type，是一个不可变的（constant）Forward Iterator。（描述于 Container。）

iterator hash_multimap::begin()

返回一个 iterator，指向 hash_multimap 的开头。（描述于 Container。）

iterator hash_multimap::end()

返回一个 iterator，指向 hash_multimap 的尾端。（描述于 Container。）

const_iterator hash_multimap::begin() const

返回一个 const_iterator，指向 hash_multimap 的开头。（描述于 Container。）

const_iterator hash_multimap::end() const

返回一个 const_iterator，指向 hash_multimap 的尾端。（描述于 Container。）

size_type hash_multimap::size() const

返回 hash_multimap 的元素个数。（描述于 Container。）

size_type hash_multimap::max_size() const

返回 hash_multimap 的最大可能元素个数。（描述于 Container。）

bool hash_multimap::empty() const

当且仅当（if and only if）hash_multimap 的大小为零，就返回 true。（描述于 Container。）

size_type hash_multimap::bucket_count() const

返回 hash table 中的 buckets 个数。（描述于 Hashed Associative Container。）

void hash_multimap::resize(size_type n)

增加 buckets 的个数，使之至少为 n。（描述于 Hashed Associative Container。）

hasher hash_multimap::hash_funct() const

返回 hash_multimap 所使用的 hasher object。（描述于 Hashed Associative Container。）

key_equal hash_multimap::key_eq() const

返回 hash_multimap 所使用的 key_equal object。（描述于 Hashed Associative Container。）

```
expilcit hash_multimap::hash_multimap(size_type n = 0,
                                      const hasher& h = hasher(),
                                      const key_equal& k = key_equal(),
                                      const Allocator& A = Allocator())
```

使用指定之 hash function、key equality function、allocator，产生一个至少具备 n 个 buckets 的空 hash_multimap。（描述于 Hashed Associative Container。）

```
template <class InputIterator>
hash_multimap::hash_multimap(InputIterator f, InputIterator l,
                             size_type n = 0,
                             const hasher& h = hasher(),
                             const key_equal& k = key_equal(),
                             const Allocator& A = Allocator())
```

使用指定之 hash function、key equality function、allocator，产生一个 hash_multimap，具有 range [f,l) 的复制内容且 buckets 数目至少为 n。（描述于 Hashed Associative Container。）

hash_multimap::hash_multimap(const hash_multimap&)

copy constructor。（描述于 Container。）

hash_multimap::~hash_multimap()

destructor。（描述于 Container。）

hash_multimap& hash_multimap::operator=(const hash_multimap&)

assignment 操作符。（描述于 Container。）

※ **allocator_type hash_multimap::get_allocator() const**

返回 hash_multimap 构造时所搭配之 allocator 的一份副本。

void hash_multimap::swap(hash_multimap&)

将两个 hash_multimaps 的内容互换。（描述于 Container。）

iterator hash_multimap::insert(const value_type& x)

将 x 安插到 hash_multimap 之中。（描述于 Multiple Associative Container。）

template <class InputIterator>
void hash_multimap::insert(InputIterator f, InputIterator l)
将 range [f,l) 安插到 hash_multimap 中。（描述于 Multiple Associative Container。）

void hash_multimap::erase(iterator pos)
删除 pos 所指元素。（描述于 Associative Container。）

size_type hash_multimap::erase(const key_type& k)
删除所有「键值为 k」的元素（如果有的话）。返回被删除的元素个数。（描述于 Associative Container。）

void hash_multimap::erase(iterator f, iterator l)
删除 range [f,l) 中的所有元素。（描述于 Associative Container。）

iterator hash_multimap::find(const key_type& k)
寻找「键值为 k」的元素。（描述于 Associative Container。）

const_iterator hash_multimap::find(const key_type& k) const
寻找「键值为 k」的元素。（描述于 Associative Container。）

size_type hash_multimap::count(const key_type& k) const
计算「键值为 k」的元素个数。（描述于 Associative Container。）

pair<iterator, iterator> hash_multimap::equal_range(const key_type& k)
找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

pair<const_iterator, const_iterator>

hash_multimap::equal_range(const key_type& k) const
找出「键值等价于 k」的所有元素所形成的 range。（描述于 Associative Container。）

bool operator==(const hash_multimap&, const hash_multimap&)
验证两个 hash_multimaps 的相等性。（描述于 Forward Container。）

16.3 Container Adapters

stack、queue 和 priority_queue 等 classes 并非 containers，它们只提供 container 操作行为的有限子集。例如 stack 只允许你安插、移除或检查栈(stack)顶端的元素。这些 classes 被称为 ***adapters***，因为它们系根据底部的（underlying）container 为实现基础。

Stacks（栈）、queues（队列）和 priority queues（带优先权的队列）都是常见的数据结构，但相应的 container adapters 的接口可能令人感到陌生。这三种 classes 都具有 member function pop，可移除最顶端元素，而且不返回任何值。你可能会奇怪，为何 pop() 不返回被移除的值呢？

如果 pop 必须返回被移除的值，那么必须以 by value 而非 by reference 的方式返回（因为元素已被移除，没有留下任何东西了）。然而，return by value 很没有效率，因为它至少会调用一次不必要的 copy constructor。由于对 pop() 而言无法有效率而又正确地将值返回，所以明智的做法就是不让它返回任何东西。如果你想要检查最顶端元素，可以调用其他 member function。

16.3.1 stack

stack<T, Sequence>

stack 是一种 ***adapter***，提供 Container 功能子集。它允许安插、移除以及审查 stack 最顶端元素。这是一种「后进先出」（last in first out, LIFO）的数据结构，stack 最顶端元素即是最后新增元素。

除了最顶端元素外，没有任何方法能够访问栈（stack）的其他元素；stack 不允许遍历其元素。这项限制是 stack 存在的唯一理由，因为任何 Front Insertion Sequence 或 Back Insertion Sequence 都足以胜任栈（stack）的要求。例如，以 vector 而言，栈的行为是 member functions back、push_back、pop_back。使用 container adapter stack 而不直接使用 Sequence 的唯一理由是，让你清楚知道你正在执行栈的行为。

由于 stack 是一个 container adapter，它实现于某个底部的 container 之上。缺省的底部型别是 deque，但也可以明确指定不同的底部型别。

栈（stacks）是一种标准数据结构，所有算法书籍都会讨论它。请参考 [Knu97] 2.2.1 节。

➢ 样例

```
int main() {
  stack<int> S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

➢ 何处定义

HP STL 将 stack 定义于头文件 <stack.h>。C++ standard 将它声明于 <stack>。

➢ Template 参数

T	stack 内所存储之 objects 的型别。
Sequence	stack 底部 container 的型别。**缺省值**：deque<T>

➢ 隶属何种 concept

Assignable（p.83）、Default Constructible（p.84）。此外如果 T 是 Equality Comparable，则 stack 同时也是 Equality Comparable（p.85）；如果 T 是 LessThan Comparable，则 stack 同时也是 lessThan Comparable（p.85）。

➢ 型别条件

- T 为 Assignable 的 model。
- Sequence 为 Back Insertion Sequence 的 model。
- Sequence 的 value type 为 T。
- 如果用到 operator==，则 T 是 Equality Comparable 的 model。
- 如果用到 operator<，则 T 是 LessThan Comparable 的 model。

➢ Public Base Classes

无

➢ 成员

stack 的某些成员并未定义于 Assignable、Default Constructible、Equality Comparable 或 LessThan Comparable 的需求条件中，它们都是 stack 所特有，皆以 ※ 符号标示之。

※ **stack::value_type**

存储于此 stack 之内的 object 型别，它和 T、Sequence::value_type 相同。

※ **stack::size_type**

一个无正负号的整数（integral）型别，相当于 Sequence::size_t。

stack::stack()

default constructor。（描述于 Default Constructible。）

※ **stack::stack(const Sequence& S)**

构造一个新的 stack，其内容与 S 相同。这个 constructor 以 S 作为初值，将 stack 底部的 Sequence object 初始化。

stack::stack(const stack&)

copy constructor。（描述于 Assignable。）

stack::~stack()

destructor。析构 stack 内的所有元素。

stack& stack::operator=(const stack&)

assignment 操作符。（描述于 Assignable。）

※ **size_type stack::size() const**

返回 stack 的元素个数。

※ **bool stack::empty() const**

当且仅当（if and only if）stack 不具任何元素，便返回 true。S.empty() 相当于 S.size()==0。

※ **value_type& stack::top()**

返回一个可变的（mutable）reference，代表 stack 的最顶端元素。先决条件：empty() 为 false。

※ **const value_type& stack::top() const**

返回一个 const reference，代表 stack 的最顶端元素。先决条件：empty() 为 false。

※ **void stack::push(const value_type& x)**

将 x 安插于 stack 最顶端。必然结果：size() 增加 1，且 top() 取得 x 的副本。

※ **void stack::pop()**

移除 stack 最顶端元素。先决条件：empty() 为 false。必然结果：size() 减少 1。

bool operator==(const stack&, const stack&)

Equality 操作符。如果两个 stacks 的元素个数相同，且每个元素都相等，则这两个 stacks 相等。

bool operator<(const stack&, const stack&)

以字典排列顺序对每个元素一一比较。

16.3.2 queue

queue<T, Sequence>

queue 是一种 *adapter*，提供 Container 功能子集。它是一种「先进先出」（first in first out, FIFO）式的数据结构。也就是说允许在 queue 的尾端新增元素，并在 queue 的前端将该元素移除。Q.front() 返回的是最早被加入 queue 内的元素。

除了 queue 的前端和尾端，没有任何方式能够访问 queue 的其他元素；queue 并不具有 iterators。这项限制是 queue 存在的唯一理由，因为任何 container 如果隶属于 Front Insertion Sequence 且为 Back Insertion Sequence，便足以胜任队列（queue）的要求。例如 deque 和 list 都拥有 member functions front、back、push_front、push_back、pop_front、pop_back。使用 container adapter queue 而不使用 container deque 的唯一理由是，让你清楚知道你正在执行队列（queue）的行为。

由于 queue 是一个 container adapter，它实现于某个底部的 container 之上。缺省的底部型别是 deque，但也可以明确指定不同的底部型别。

队列（queues）是一种标准数据结构，所有算法书籍都会讨论它。请参考 Knuth 书籍 [Knu97] 2.2.1 节。

➢ 样例

```
int main() {
  queue<int> Q;
  Q.push(8);
  Q.push(7);
  Q.push(6);
  Q.push(2);

  assert(Q.size() == 4);
  assert(Q.back() == 2);

  assert(Q.front() == 8);
  Q.pop();

  assert(Q.front() == 7);
  Q.pop();

  assert(Q.front() == 6);
  Q.pop();

  assert(Q.front() == 2);
  Q.pop();

  assert(Q.empty());
}
```

➢ 何处定义

HP STL 将 queue 定义于头文件 <queue.h>。C++ standard 将它声明于 <queue>。

➢ Template 参数

T	queue 内所存储之 objects 的型别。
Sequence	queue 底部 container 的型别。**缺省值**：deque<T>

➢ 隶属何种 concept

Assignable（p.83）、Default Constructible（p.84）。此外如果 T 是 Equality Comparable，则 queue 同时也是 Equality Comparable（p.85）；如果 T 是 LessThan Comparable，则 queue 同时也是 lessThan Comparable（p.85）。

➢ **型别条件**

- T 是 Assignable 的 model。
- Sequence 是 Back Insertion Sequence 的 model。
- Sequence 是 Front Insertion Sequence 的 model。
- Sequence 的 value type 是 T。
- 如果用到 operator==，则 T 是 Equality Comparable 的 model。
- 如果用到 operator<，则 T 是 LessThan Comparable 的 model。

➢ **Public Base Classes**

无

➢ **成员**

queue 的某些成员并未定义于 Assignable、Default Constructible、Equality Comparable 或 LessThan Comparable 的需求条件中，它们都是 queue 所特有，皆以 ※ 符号标示之。

※ **queue::value_type**

存储于此 queue 之内的 object 型别，它和 T、Sequence::value_type 相同。

※ **queue::size_type**

一个无正负号的整数（integral）型别，相当于 Sequence::size_t。

queue::queue()

default constructor。（描述于 Default Constructible。）

※ **queue::queue(const Sequence& S)**

构造一个新的 queue，其内容与 S 相同。这个 constructor 以 S 作为初值，将 queue 底部的 Sequence object 初始化。

queue::queue(const queue&)

copy constructor。（描述于 Assignable。）

queue::~queue()

destructor。析构 queue 内的所有元素。

queue& queue::operator=(const queue&)

assignment 操作符。（描述于 Assignable。）

※ **size_type queue::size() const**

返回 queue 的元素个数。

※ **bool queue::empty() const**

当且仅当（if and only if）queue 不具任何元素，便返回 true。S.empty() 相当于 S.size() == 0。

※ **value_type& queue::front()**

返回一个可变的（mutable）reference，代表 queue 最前端元素，也就是最早被安插的元素。先决条件：empty() 为 false。

※ **const value_type& queue::front() const**

返回一个 const reference，代表 queue 最前端元素，也就是最早被安插的元素。先决条件：empty() 为 false。

※ **value_type& queue::back()**

返回一个可变的（mutable）reference，代表 queue 最尾端元素，也就是最新被安插的元素。先决条件：empty() 为 false。

※ **const value_type& queue::back() const**

返回一个 const reference，代表 queue 最尾端元素，也就是最新被安插的元素。先决条件：empty() 为 false。

※ **void queue::push(const value_type& x)**

将 x 安插于 queue 最尾端。必然结果：size() 增加 1，且 back() 取得 x 的副本。

※ **void queue::pop()**

移除 queue 最顶端元素。先决条件：empty() 为 false。必然结果：size() 减少 1。

bool operator==(const queue&, const queue&)

Equality 操作符。如果两个 queues 的元素个数相同，且每个元素都相等，则这两个 queues 相等。

bool operator<(const queue&, const queue&)

以字典排列顺序对每个元素一一比较。

16.3.3 priority_queue

priority_queue<T, Sequence, Compare>

priority_queue 是一种 ***adapter***，提供 Container 功能子集。它提供安插、审视、移除最顶端元素的功能。没有任何机制可以更改 priority_queue 的任何元素，或是遍历这些元素。

priority_queue 的最顶端元素一定是元素中的最大值，function object Compare 用来定义元素间的大小次序关系。这正是 priority_queue 不提供任何元素访问方法的原因。「最大元素位于最顶端」这个事实是此等 class 的一个恒长特性（class invariant）。

由于 priority_queue 是一个 container adapter，它实现于某个底部的 container 之上。缺省的底部型别是 vector，但也可以明确指定不同的底部型别。

带优先权的队列（priority queues）是标准的数据结构，可以不同方式实现出来。通常 priority_queue 是以 heap 来实现，并以算法 make_heap、push_heap 和 pop_heap 来维护 heap 的状态。Priority queues 在所有算法书籍都会讨论到，请参考 Knuth 书籍 [Knu98b] 5.2.3 节。

➢ 样例

```
int main() {
  priority_queue<int> Q;
  Q.push(1);
```

```
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);
  while (!Q.empty()) {
    cout << Q.top() << " ";
    Q.pop();
  }
  cout << endl;
  // 输出为 8 7 5 4 2 1
}
```

上个例子中的 priority_queue，其最顶端元素一定是数值最大的元素。元素次序关系可由指定之 function object 决定，所以轻易就能产生一个 priority_queue，使其最顶端元素为数值最小之元素。

```
int main() {
  priority_queue<int, vector<int>, greater<int> > Q;
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);
  while (!Q.empty()) {
    cout << Q.top() << " ";
    Q.pop();
  }
  cout << endl;
  // 输出为 1 2 4 5 7 8
}
```

➢ 何处定义

HP STL 将 priority_queue 定义于头文件 <stack.h>。C++ standard 将它声明于 <queue>。

➢ Template 参数

T	priority_queue 内所存储之 objects 的型别。
Sequence	priority_queue 底部 container 的型别。**缺省值**：vector<T>
Compare	一个函数，用来测试某元素是否小于另一元素。**缺省值**：less<T>

➢ 隶属何种 concept

Assignable（p.83）、Default Constructible（p.84）。

➢ 型别条件

- T 为 Assignable 的 model。
- Sequence 为 Sequence 的 model。
- Sequence 为 Random Access Container 的 model。
- Sequence 的 value type 为 T。
- Compare 为 Strict Weak Ordering 的 model。
- Compare 的 value type 为 T。

➢ Public Base Classes

无

➢ 成员

priority_queue 的某些成员并未定义于 Assignable 或 Default Constructible 的需求条件中，它们都是 priority_queue 所特有，皆以 ※ 符号标示之。

※ **priority_queue::value_type**

存储于此 priority_queue 之内的 object 型别，它和 T、Sequence::value_type 相同。

※ **priority_queue::size_type**

一个无正负号的整数（integral）型别，相当于 Sequence::size_t。

```
explicit priority_queue::priority_queue(const Compare& c = Compare())
```

产生一个空的 priority_queue，并以 c 作为两个 objects 的比较标准。c 的缺省值是 Compare()。（描述于 Default Constructible。）

```
※ template <class InputIterator>
  priority_queue::priority_queue(InputIterator f, InputIterator l,
                                 const Compare& c = Compare())
```

产生一个 priority_queue，内含 range [f,l) 的元素。并以 c 作为比较标准。

※ **priority_queue::priority_queue(const Compare& c, const Sequence& S)**

以 c 为比较标准，产生一个 priority_queue，使它包含与 S 相同的元素。换句话说它以 S 为初值，将 priority_queue 底部的 Sequence object 初始化，然后重新将这些元素排序，使它们满足 priority_queue 的恒长特性。

※ **template <class InputIterator>**
priority_queue::priority_queue(InputIterator f, InputIterator l,
const Compare& c,
const Sequence& S)

以 c 为比较标准，产生一个 priority_queue，使它包含 [f,l) 以及 S 的所有元素。换句话说它以 S 为初值，将 priority_queue 底部的 Sequence object 初始化，并增入 [f,l) 内的元素，然后再重新对这些元素排序。

priority_queue::~priority_queue()

destructor。析构 priority_queue 的所有元素。

priority_queue&priority_queue::operator=(const priority_queue&)

assignment 操作符。（描述于 Assignable。）

※ **size_type priority_queue::size() const**

返回 priority_queue 的元素个数。

※ **bool priority_queue::empty() const**

当且仅当（if and only if）priority_queue 不具任何元素，便返回 true。S.empty() 相当于 S.size() == 0。

※ **const value_type& priority_queue::top() const**

返回一个 const reference，代表 priority_queue 的最顶端元素。最顶端元素保证是以 Comp 决定出来的最大元素，亦即对于 priority_queue 的其他任何元素 x，Comp(Q.top(), x) 必为 false。先决条件：!empty()。

※ **void priority_queue::push(const value_type& x)**

将 x 安插于 priority_queue 中。必然结果：size() 将增加 1。注意，x 通常不会变成最顶端元素，除非 x 大于 priority_queue 内的任何元素。

※ **void priority_queue::pop()**

移除 priority_queue 的最顶端元素，亦即 priority_queue 的最大元素。先决条件：!empty()。必然结果：size() 减少 1。

附录 A

可移植性与标准化

Portability and Standarization

C++ 语言正处于一个过渡状态。

这个语言早期唯一的实现是 AT&T Cfront 编译器，最可靠的参考手册则是 Bjarne Stroustrup 所着的 ***The C++ Programming Language*** [Str86] 第一版。此后，当其他厂商的编译器纷纷出现，Margaret Ellis 与 Stroustrup 合著的 ***Anotated C++ Reference Manual*** [ES90]（被 C++ 迷昵称为 ***ARM***）便成为实际上的标准。

C++ 目前正处于国际标准化的阶段（国际标准最后草稿于 1997 年 11 月 14 日通过），该标准规格目前在国际标准组织(International Standards Organization, ISO)与美国国家标准机构(American National Standard Institute, ANSI）联合委员会的控制之下。在标准化过程中，这个语言有了些许改变。C++ standard [ISO98] 与 ***ARM*** 之间的差异，就像 ANSI C standard 与 ***The C Programming Language*** [KR78] 之间的差异一样，极富戏剧性。很多新特色被加入这个语言之中，很多方面则做了净化。

这些枝枝节节对于一般程序员似乎无关紧要，但它却造成不同的 C++ 实现品（编译器）之间不再那么理所当然地具备移植性。目前还没有一个编译器实现完整的 C++ standard 规格，也很少有编译器将自己局限于 ***ARM*** 所描述的特色。几乎所有 C++ 编译器所实现出来的，都介于 ***ARM*** C++ 与 standard C++ 之间。

STL 的情况也很类似。原始的 STL 定义于 Alexander Stepanov 和 Meng Lee 所撰写的技术报告内 [SL95]。然而 STL 现在已成为 standard C++ library 的一部分了。C++ 标准委员会对 STL 的定义做了些微改变 — 就像它也曾经对 C++ 核心语言定义做了些微改变一样。

事实上即使 STL 原始的 HP 版本是由设计该程序库正式规格的人所撰写，那份实现品也没有完全遵循正规定义。事实上那个时候也没办法遵循正规定义，因为 STL 的定义之中所需要的语言特性在当时还没有能够实现出来。HP 版本是「正规设计」与「1994 年的编译器技术」之间的一个妥协方案。

到了 1998 年，仍然没有一个 STL 实现品完全符合 Stepanov-Lee 的原始定义，或是符合 C++ standard 的定义。技术上的一个原因是，那份标准规格仍然是份草稿，而且一直有所变动。试图跟上变动中的规格脚步，就像爱丽丝（Alice）与红心皇后（Red Queen）之间的竞赛：「在这场赛跑中你所能够做的，就是保持相同的距离。」大部分程序库实现者在 C++ standard 正式定案之前，会保持谨慎观望的态度。

第二个问题是，即使到了今天，也没有任何一个 C++ 编译器支持 STL 所用到的每一个语言特性。（是的，编译器厂商也同样谨慎得很！）在尚未获得某些语言特性之前，STL 的某些特性不是暂时空着，就是通过某种非标准的迂回方式实现出来。不幸的是，这些限制及迂回法往往罕有文档介绍，或甚至根本没有文档。

STL 中的绝大部分组件已经完全稳定下来了，这一部分不受编译器的束缚，标准委员会也未曾更动过它。你可以预期这些部分在每个实现版本中都是一样的。至于其他部分就比较不稳定了。有时候不同实现版本间的差异性会带来重要的影响。

幸运的是，这些混乱状况只出现于边缘地带。STL 之中没有任何变化或任何实现上的困难点会影响到该程序库的核心设计。泛型编程的基本原则以及 STL 代码主体在任何编译器以及任何 STL 实现版本中都是有效的，实现版本之间的差异只集中于某几个问题。

这一份附录谈的是你必须知道的可移植性议题。包括：(1) 不同实现品中某些组件的不同定义方式；(2) 编译器带来的限制（以至无法以正规定义方式来实现 STL 程序库）；(3) 定义有所更动的某些组件；(4) 某些会影响 STL 运用方式的 C++ 语言变动。

A.1 语言上的变动

A.1.1 Template 编译模型（Compilation Model）

自从 C++ 增加 template 特性之后，templates 分离式编译（separate compilation）就成了语言的一部分。其目的在于让 function templates 和 class templates 可以如同一般 functions 和 classes 一样地被使用。也就是说，你可以在某个源代码文件中声明一个函数，然后在不同的源代码文件中使用这个函数。这两个文件可以分开编译，只要最后两个目标文件（object files）被连接（linked）起来即可。

ARM 要求 templates 分离式编译，C++ standard 亦然 — 尽管某些细节有了些微变动。AT&T Cfront 3.0 编译器包含受限的 templates 分离式编译，其他很多编译器亦具有部分受限的分离式编译能力。

不幸的是，没有任何两种分离式编译完全相同。某些涉及了代码中特别的 #pragma 声明，某些要求代码文件名需符合特定命名规则，甚至于很多编译器根本就不具有任何 templates 分离式编译能力。

你的编译器可能允许某种分离式编译，但如果你使用那种方式，当移植到其他编译器时，可能会遇到一些麻烦。或许多年之后，每个编译器才会支持 C++ standard 所定义的分离式编译。

目前唯一具可移植性的解决方案就是，遵循一个规则：「function template 的定义式，不只有其函数原型(prototype)，还必须在该函数被使用之前可见(visible)」。这意味 function templates 和 class templates 必须定义在头文件（*.h）中，而你必须含入适当的头文件，才能使用某个 template。这就是为什么现有的所有 STL 实现品大部分（或甚至全部）都是一些头文件的原因。

A.1.2 带缺省值的 Template 参数（Default Template Parameters）

就像 C++ 函数的参数可具有缺省值，class templates 也一样。假设你声明一个 class template 如下：

```
template <class A, class B = A>
class X {
  ...
};
```

那么当你具现化（instantiated）X 时，可以不指定第二个 template 参数。这么一来便会采用缺省值 A。是的，型别 X<int> 与型别 X<int, int> 完全相同。

所有的 STL container classes 都使用带有缺省值的 template 参数。举个例子，vector 就具有两个 template 参数，第一个表现出 vector 的元素型别，第二个表现 allocator，用来参数化「vector 的内存分配策略」。第二个 template 参数便带有缺省值，因为大部分的 vector 使用时机都没有理由使用非标准的内存分配策略。

目前，并非所有的 C++ 编译器都完全支持带缺省值的 template 参数。某些编译器对此有某种限制（例如要求缺省值不能与其他 template 参数相依），某些编译器则根本不允许。由于 STL 极度仰赖带缺省值的 template 参数，所以它不能够在那样的编译器上精确地依据标准设计实现出来。

当一个编译器不具备这个语言特性，基本上有两种方法可以实现带缺省值的 template 参数：要不就要求使用者必须提供该参数值，要不就全然移除该参数。以 vector 为例，第一种方法要求你总得写成 vector<int, allocator<int> >，即使你用的是缺省的 allocator。第二种方法允许你简单地写成 vector<int>，但你却也因此无法使用他种 allocator。这些妥协办法都不完全令人满意。

HP STL 完成之时，尚未有任何编译器支持带缺省值的 template 参数。当时它使用上述第二种方法。两种方法在后继的 STL 实现品中相继都被用过。

A.1.3 Member Templates

templates 最初加入 C++ 语言时，只有两种东西可以是 template：全局的 classes 和全局的 functions。也就是说，template 声明式不能在 class scope 内发生。这意味 member functions 不能是 function template。

这个限制是没有必要的。如今 C++ standard 已允许 non-virtual member functions[57]可以成为 function templates。Member function templates 的调用方式和一般的 function templates 一样。编译器可以由函数调用所给定的引数推导出 template 参数，并自动产生该 member function 的适当实体（instance）。

Class's constructor 是一个 member function，所以如同其他 member functions 一样，它可以成为一个 template。这个结果导致 member templates 最重要的用途之一。例如 class `pair` 便拥有一个经过泛化的 copy constructor：

```
template <class T1, T2> template <class U1, class U2>
pair<T1, T2>::pair(const pair<U1, U2>&);
```

此式允许以任何 `pair<U1, U2>` 构造出任何一个 `pair<T1, T2>`，只要 `U1` 可转换为 `T1` 且 `U2` 可转换为 `T2`。这种语法看起来很奇怪，但它是正确的。关键字 `template` 必须出现两次，因为此处有两个 template 参数列，一个针对 `pair` template 自身，另一个针对 member template constructor。

STL container classes 大量使用 member templates。让我再说一次，其主要用途之一是在 constructors。你可以根据一个 iterators range 构造出一个 container，member templates 允许该 range 所含之型别为任何 Input Iterator 之 model。例如你可以这样产生一个 `vector V`，让它和 `list L` 包含相同的元素：

```
vector<T> V(L.begin(), L.end());
```

只要 `L` 的 value type 可转换为 `T`，这种写法就有效。同样地，containers 的 `insert` member functions 也可以藉由 member templates 技术，接受一个 iterators range。

在编译器支持 member templates 之前，这种功能无法达成。如果你的编译器尚未支持 member templates，便无法撰写一个完全一般化的 constructor，允许由某个 Input Iterators range 构造出一个 `vector`。另一个可行办法是针对某些挑选出来的 iterator types，将 `vector` constructor 重载。这里所谓挑选出来的 iterator types 是程序库实现者预期最有用的几个 iterator types。通常你应该至少能够根据一个 pointers range 以及一个 container-A iterators range 构造出一个 container-A。

class `pair` 的 member template constructor 不完全是方便性的问题。举个例子，标准的 container `map` 是一个 Associative Container，其元素型别为 `pair<const Key, Data>`。和其他 Associative Container 一样，`map` 有一个 member function 可让你将单个元素安插到 container 内。当你写：

```
M.insert(p)
```

---

[57] 根据 *The Design and Evolution of C++* [Str94] 所描述，由于某种技术原因，virtual member functions 目前仍然不可为 templates。

此处 p 的型别相同于 M 的元素型别，也就是说 p 的型别是 pair<const Key, Data>。问题在于如何构造出 object p。

如果使用 member template constructor，你可以不必写出 pair constructor 就将一个元素安插到 map：

```
M.insert(make_pair(k, d))
```

这会产生 pair<Key, Data> object，然后将该 object 转为 pair<const Key, Data>，然后将它安插到 map 中。如果没有 member template constructor，便没有这种转换行为。一旦缺少 member templates 的支持，你就必须使用较不方便的形式：

```
M.insert(pair<const Key, Data>(k, d))
```

A.1.4 偏特化（Partial Specialization）

当 template 成为 C++ 语言性质的那一天起，我们就已经可以将一个 class template 特化了。假设你有一个 class X<T>，由于某种原因，X 的一般定义不能套用在某个特定型别上，那么你可以针对该型别给予 X 不同的定义。例如你或许能够将 X<int> 设计得比一般版本更有效率。X<int> 可以和一般版的 X<T> 完全无关，它可以拥有一组完全不同的 member functions。

Class templates 的特化如今进展到了所谓的「偏特化（partial specialization）」。就拿以下的例子来说，你可以针对「整个型别类属（entire category of types）」，而非只针对单一型别，给予 template 一份不同的定义。

```
template <class T> class X
{
  // 版本 1：最一般化（泛化）。
};

template <class T> class X<T*>
{
  // 版本 2：针对普通指针。
};

template <> class X<void*>
{
  // 版本 3：针对某种特定指针。
};
```

全特化与偏特化并不使用相同的语法。这是 C++ 标准化过程中的一项改变。如果你所使用的编译器遵循 C++ standard，你必须写

```
template <> class X<void*>
```

如果你使用的编译器比较旧，你得这么写：

```
class X<void*>
```

这个式子并未使用关键字 template。除了使用预处理器（pre-processor）所提供的宏，否则再没有其他方法可以写出符合新旧规格的全特化语法。

偏特化对于「优化」常常很有用途。例如有时候你可以针对指针写出某个 class 的特别版本，此一特别版本拥有最佳性能。此外，偏特化使得某种程序设计技术变得可能，而 STL 正是架构于这些技术之上。

第一，为了存储效率，STL 内含一个特化版本的 vector container class：vector<bool>。乍见之下这不像是一个偏特化的例子，但它的确是：就像所有的 STL 序列一样，vector 具有两个 template 参数：一个是 value type，一个是 allocator。vector<bool> class 是一个偏特化版本，因为它给予第一个 template 参数一个特别型别，而让 allocator 完全泛化。在不支持偏特化的编译器上，STL 实现品通常会声明一个 class bit_vector 取而代之。bit_vector 只是一个过渡时期的迂回做法，并非 C++ standard 的一部分，而且一旦偏特化被广泛支持之后，它就可能消失。

更重要的是，STL 基本要素之一的 iterator_traits，非常关键性地倚赖偏特化技术。

3.1 节曾描述的 iterator_traits class，是「访问 iterator 相关型别信息」的一个重要机制。对于任何一个 iterator type I，

```
iterator_traits<I>::value_type
```

便是 I 的 value type，而

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

便是 I 的 difference type。这须得倚赖偏特化，因为面对一般指针，我们无法像面对「以 classes 形态出现的」iterators 那样地定义 iterator_traits。

你可以在自己的程序中使用 iterator_traits。它也被使用于 STL 的其他部分。

- Iterator adapter reverse_iterator 接受单一一个 template 参数 Iter，这是一个 Bidirectional Iterator。这个 adapter 使用 iterator_traits 机制，使 reverse_iterator<Iter> 具有与 Iter 相同的 difference type 和 value type。
- 算法 distance（p.181）、count（p.214）和 count_if（p.216）都在 input iterators range 上头动作。这些算法都返回型别 typename iterator_traits<InputIter>::difference_type 的值。

原始的 HP STL 并不仰赖偏特化，也不具备 iterator_traits class。它提供较粗陋的机制来访问 iterator 型别信息，此机制涉及查询函数 distance_type、value_type、iterator_category。它们并未被纳入 C++ standard。大多数 STL 实现品仍然支持这些函数，但它们最终会被移除。

如果你的编译器不支持偏特化，你便不能使用 iterator_traits 的完全一般化性质。这时候你可能

得以 HP STL 的旧式机制来取代，也可能得使用旧式的 count、count_if、reverse_iterator。

函数的偏特化（partial specialization of functions）

C++ standard 有另一项特色，和偏特化非常类似：那就是 function template 的偏序（partial ordering）关系。

Template 函数，如同其他函数一样，是可被重载的（overloaded）。这提高了面对一个函数调用时有多个版本能够精确吻合（exact match）的几率。假设你声明了两个 function templates：

```
template <class T> void f(T)
template <class U> void f(U*)
```

当你写下 f((int*)0)，你会调用哪个版本？该调用动作可吻合第一版本（当 T 为 int*）或第二版本（当 U 为 int）。

在 template 初被引进之际，这类函数调用是不合法的，因为它精确吻合了一个以上的函数。但是新规则已经放宽了这项限制。每当某个函数调用动作可匹配多个重载的 function templates 时，编译器会选择其中最特化（most specialized）的一个。此例之中 f 的第二版本比第一版本更特化。是的，第一版本能匹配任何型别，但第二版本只能匹配指针。

STL 只在一个地方使用函数的这种偏序（partial ordering）关系。它有一个 swap 函数（p.237），可交换任何两个变量内容。STL 针对所有的 container classes 定义了特化的 swap 版本。当你写下 swap(v1, v2)，如果 v1 和 v2 都是 vectors，你会调用以下函数：

```
template <class T, class Allocator>
void swap(vector<T, Allocator>&, vector<T, Allocator>&)
```

而非更一般化的版本：

```
template <class T> void swap(T&, T&)
```

这很重要，因为 swap 的一般化版本必须以赋值行为（assignment）来运作，而将一个 vector 赋值给另一个 vector 是很慢的动作（因为它必须复制该 vector 的所有引数）。特化版本直接在 vector 的内部数据上起作用，非常快速。

在不支持 function templates 之偏序（partial ordering）性质的编译器上，STL 并未定义出特化的 swap 版本。但这不会影响你所能撰写的代码 — 你仍然可以使用一般版的 swap 来交换两个 vectors，这意味某些程序的执行速度会比它们所应该表现的慢许多。

A.1.5 新加入的关键字

C++ 语言新增了许多关键字，这些关键字是 *ARM* 发行之时所没有的。对于 STL 以及运用 templates 的程序而言，有两个关键字特别重要，那就是 explicit 和 typename。

关键字 explicit

explicit 用来抑制某种「自动型别转换」功能。通常，如果 class x 具有一个 constructor，可接受型别为 T 的单一引数，C++ 编译器便会自动以该 constructor 将型别为 T 的值转换成型别为 x 的值。因此假设你有一个函数 f，需要一个型别为 x 的引数，而 t 的型别为 T，你可以直接写 f(t)，不必手动完成转换。

有时候这是你所想要的。有时候这种自动转换会导致非常不可预期的结果。举个例子，如果 f 预期其引数为 vector<string>，难道你希望 f(3) 有效吗？但 vector<string>(3) 却是一个完全合理的 constructor call。

将单引数 constructor 声明成 explicit，就可以抑制这种自动转换行为。噢是的，你还是可以使用该 constructor，但必须明确写出才行。C++ standard 将大多数 containers 的单引数 constructors 都声明为 explicit。

由于 explicit 的目的在抑制某些自动发生的事，意味可能有某些程序在旧规则中原本合法，在使用标准编译器与标准程序库后便不合法了。如果你真的想要传入一个新构造的 vector，并使它拥有三个空的 strings，你得明确写成：f(vector<string>(3))。这个事实亦套用于其他不同的声明上。不要写成：

```
vector<string> v = 3;
```

你必须写为：

```
vector<string> v(3);
```

或

```
vector<string> v = vector<string>(3);
```

关键字 typename

explicit 关键字只影响少数含糊不清的构造行为 — 它禁止了某些「如今已不再是个好概念」的行为。但 typename 就不一样，它出现在 C++ 泛型编程的很多基础地点。如果你曾经使用 templates 写过任何象样的 C++ 程序，你必须了解如何使用 typename。

基本议题属于技术层面，但不很复杂。当编译器看到某些表达式涉及 template 参数，它应该将该表达式视为「取用某型别」，或视为「取用其他某物（例如 member function 或 member variable）」呢？

举例来说，在这个函数中：

```
template <class X> void f(X) {
  X::T(x);
}
```

编译器究竟应该将 T 解释为一个型别，致使 X::T 成为一个 constructor call（也就是构造出一个匿名

的临时对象，然后再将它丢掉）呢？或者应该将 T 解释为一个 static member function 或 static member variable 呢？同样的情况，以下函数：

```
template <class X> void g(X) {
  X::T t;
}
```

编译器究竟应该将 T 解释为一个型别，致使这个函数使用 T 的 default constructor 来产生 object t 呢？或者应该将 T 解释为一个 static member function 或 static member variable，致使 X::Tt 形成一个语法错误呢？

后一种情况特别麻烦。一旦 g 被具现化（instantiated），编译器当然可以辨别 X::T 究竟是一个 class、一个 member function、一个 member variable 或其他某物。但如果无法在尚未具现化的时候便检查出该 template 语法的正确性，实在令人失望。果真如此，template 的分离式编译也就毫无指望了。

C++ 语言的设计使得 template 的语法检验可行。C++ 有一个非常简单的规则来决定是否要将 X::T 视为型别：如果某个名称可能代表型别或代表其他某物，则「除非你以别种方式明确告知编译器，否则编译器总是假设该名称不代表某个型别」。

typename 关键字就是用来告知编译器应该将某特定名称视为一个型别。函数 g 的正确写法是：

```
template <class X> void g(X) {
  typename X::T t;
}
```

在 X::T 之前冠以关键字 typename，就是告知编译器应该将 X::T 解释为型别。注意，每当要代表一个型别时，你都必须使用 typename。也就是说你必须写出：

```
template <class X> void g1(X) {
  typename X::T t;
  typename X::T* pt;
}
```

第二个 typename 也是必要的。

基本规则如下：如果没有 typename 的修饰编译器就无法将某物视为一个型别，那么你就必须使用 typename。也就是说，如果某个名称符合以下两条件，你就必须使用 typename：

1. 它是一个资格受限的名称（qualified name），亦即其他型别中的嵌套型别。
2. 它取决于 template 参数，也就是取决于 template 被具现化后的 template 引数。

例如，函数 g 内的 X::T 是一个资格受限的名称（以 X:: 加以限制），并取决于 template 参数 X。同样道理，表达式 Y<X>::T 也有一个资格受限的名称，取决于 template 参数 X。

这对 STL 应用程序象征了什么意义？唔，第一件事是不管你自己正在撰写新的 STL 组件，或者使用

别人所定义的组件，你得使用很多 typename — 甚至当你正在做某件似乎完全无害的事时，例如撰写一个作用于 vector<T> 身上的 template function 而且必须取用该 vector 的 iterator：

```
template <class T>
void f(vector<T>& v) {
  typename vector<T>::iterator i = v.begin();
  ...
}
```

这虽然看起来很奇怪，但此地的确需要 typename。对身为人类的你看来，很显然 vector<T>::iterator 一定是取用 vector 的嵌套的 iterator type，但对编译器而言，那只是一种取决于 template 参数的资格受限名称。除非你明白告知编译器 vector<T>::iterator 是个型别名称，否则编译器无法知道。

这个道理也适用于 iterator_traits。在 3.1.5 节中我们看到，定义 iterator_traits class 时，我们必须使用 typename；同样道理当我们使用 iterator_traits 时，几乎亦总得使用 typename。以 count 算法（p.214）为例，它具有这样的形式：

```
template <class InputIterator, class EqualityComparable>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const EqualityComparable& value)
```

不用说，typename 造成了可移植性的问题。符合 C++ standard 之编译器需要它，但其他更早（尚未加入关键字 typename）之前的编译器却不认识它。时值 1998 之际，这两种编译器仍然都很常见。

唯一可以通吃上述两种编译器的做法，就是利用预处理器技巧。你可以定义一个宏 TYPENAME，让它在某些编译器中展开为 typename，在另一些编译器下则什么也没展开。

关键字 typename 是一个技术细节，是为了某种特殊技术目的才导入的。但不幸的是你无法忽视这个细节。好消息是，如果你错用了 typename，编译器一定会给你某些错误消息。是的，万一不小心遗漏 typename，并不会让一个原本做某事情的有效程序摇身变为一个做不同事情的有效程序。

A.2 程序库的变动

A.2.1 Allocators

就像 9.4 节所说，STL containers 都将其内存分配方法参数化了。这种做法有时候很有用，但从历史的角度来看，它却成为 STL 最不稳定的一部分。目前大家常用的有四种不同的 allocator 设计：HP STL allocators、SGI STL allocators，以及从 C++ standard 草案衍化而来的两个不同的 allocators 设计。

由于两个进一步的理由，使可移植性更成为问题所在。第一，关于「container 如何使用 allocators」，

C++ standard 给予程序库实现者良好的自由度。第二，许多 STL 实现品并未实际提供这四种设计中的任何一种。HP allocator 在缺乏支持所谓 "template template parameters"[58]的环境下无法实现出来，草案版的 allocators 在缺乏支持 member templates 的环境下也不可行。如果你的编译器不支持 member templates，你便不能使用 C++ standard 所描述的 allocators 版本，顶多只能使用某家厂商所提供的折衷版本。

如果你关心可移植性议题，最安全的做法是根本就不要使用 allocators。啊，应该说是使用缺省的 allocator。Allocator 是一个带有缺省值的 template 参数，所以你无须指明。如果你确实必须撰写自己的 allocator，你应该详读文档，并找出你的程序库所使用的 allocator 设计法。更明确地说，如果你的编译器不支持 member templates，你应该确定对于必要的迂回做法与限制都有足够的了解。每当使用不同的编译器或程序库，你也应该预期必须做某些改变。

定义于 C++ standard 中的 allocators，最显而易见的创新之处是 allocator *instances*。Member functions `allocate` 和 `deallocate` 与特定的 allocator objects 相应。如果你通过一个 allocator object `a` 来分配内存，你必须通过一个相等于 `a` 的 allocator 来释放该内存，因为不同的 allocators 可能参考到不同的内存局部。

只让 allocator 成为 container 型别的一部分是不够的。每个 container object 必须包含一个特别的 allocator instance。即使两个 containers 具有相同的型别，它们不见得要以相同的方式分配内存。

container 被构造出来后才去改变其 allocator，这种做法是不合理的，必须有某种方式来控制该 container 使用哪一个 allocator instance。如果 container 以 allocators 作为参数之一，那么该 container 的所有 constructors 就都必须拥有一个 allocator 参数。

C++ standard 定义的所有 STL containers（`vector`、`list`、`deque`、`set`、`map`、`multiset`、`multimap`）都以这种方式来定义其 constructors。这些 containers 的 constructors 都拥有一个型别为 `Allocator` 的参数，大部分情况下它都是 constructor 的最后一个参数，并具有缺省值 `Allocator()`。你可以不提到它，以避免使用 allocator instances。因此如果我们写：

```
vector<int> V(100, 0);
```

就相当于写：

```
vector<int, allocator<int> > V(100, 0, allocator<int>());
```

A.2.2 Container Adapters

STL 定义三种 container adapters：`stack`、`queue` 和 `priority_queue`。它们并非 Containers，它们不提供 Container 的整个接口，只提供该接口的有限子集。Container adapter 只是对其底部（underlying）

[58] 这项语言特性并未在本书任何地方提及，因为 STL 完全没有用到它。请参考 *The C++ Programming Language* [Str97] C.13.3 节。

containers 的某种外包装（wrapper）而已，该底部 container 是一个 template 参数。此一参数化的精确形式目前已有所改变。

在原始的 HP STL 中，stack adapter 需要单一一个 template 参数，作为该 stack 底部数据的型别。如果你想要产生一个由 ints 组成的 stack 并以 deque 作为其底部数据组织，你得这么声明：

```
stack< deque<int> >
```

这虽然不至于含糊不清，但却令人困惑。看起来像是说「stack 内的值就是 deques，而非 ints」。此外，它迫使你必须明白声明某些东西（也就是其底部数据组织），而那应该只归属于实现细节。

C++ standard 已经改变这一点。stack 如今需要两个 template 参数。第一个是存储于该 stack 内的 object 型别，第二个才是该 stack 的底部数据组织。当然后者是一个带缺省值的 template 参数。对 stack 而言，该缺省值为 deque。因此，由 ints 组成的 stack 可以声明为 stack<int>。

这项改变造成了两个可移植性问题。第一，虽然大多数 STL 实现品都已改变其 container adapters 以遵循 C++ standard 的规定，但毕竟不是全部都如此。因此就着你手上的程序库实现品，你可能得使用较旧的 container adapters 版本。第二，如果你使用的是不支持 default template parameters 的编译器，那你便不能准确使用 C++ standard 所规范的 container adapters。你应该详读文档，找出你的程序库实现者所使用的迂回做法。一种常见的迂回做法是要求你提供两个 template 参数，例如你可能得把由 ints 组成的 stack 声明为：

```
stack<int, deque<int> >
```

A.2.3 次要的程序库变动

标准化过程中，STL 的接口于某些次要部分亦有些微的改变及扩充。

新的算法

C++ standard 内含三个并未定义于原始 HP STL 中的算法：(1) find_first_of，它很类似 C 函数库函数 strpbrk；(2) find_end，它很类似 STL 算法 search；(3) search_n。

Standard 还包含了三个泛型算法：uninitialized_copy、uninitialized_fill 和 uninitialized_fill_n，以及两个临时内存分配函数：get_temporary_buffer 和 return_temporary_buffer，这些函数都曾出现于 HP 版本中，但未见诸文档。这些特别函数主要用于自行撰写自己的 container 或 adaptive 算法。

Iterator 接口改变

比起原始的 HP STL，C++ standard 对于 iterators 多定义了一个次要功能。所有 iterators（Output Iterator

除外，因为以下叙述对于它并不合理）如今都定义了 member function operator->。如同你对指针的预期，i->m 和 (*i).m 应该代表相同的事情（动作）。

这对于 map 和 multimap 特别方便，因为这些 containers 的元素都是 pairs。如果 i 是一个 iterator，其 value type 是 pair<const T, X>，你可以写 i->first 或 i->second，就好像 i 就是一般指针一样。

Sequence 接口改变

所有的 Sequences 都包含了并未列于原始 HP STL 中的三个新的 member functions：(1) resize，它会删除或附加元素于尾端，使 Sequence 变成所指定的大小；(2) assign，它是一种一般化的赋值操作行为；(3) clear，它是 erase(begin(), end()) 的简略写法。此外，erase member function 的返回型别也有了改变。在 HP STL 中其返回型别为 void，但 C++ standard 将它改变为 iterator。是的，erase 的返回值如今是一个 iterator，指向被移除元素的下一个紧邻位置。

multiplies Function Object

HP STL 内含一个 function obejct times，可用来计算两个引数的乘积。C++ standard 将这函数改名为 multiplies。

不幸的是没有任何实现品有什么好方法可以同时提供新旧名称。这个名称之所以被改变，是因为 times 与 UNIX 的某个头文件名称互相冲突，如果同时保留新旧两个名称，改变就没有意义了。如果你正在将代码由旧的 STL 版本移植到新的版本，你得留意这个 function object 的名称。

A.3 命名及包装（Naming and Packaging）

> *What's in a name? That which we call a rose*
> *By any other word would smell as sweet.*
>
> 名字有什么意义？我们称呼玫瑰为其他名字，并不会改变其芬芳本质。

命名议题或许很琐细，但它们是 C++ standard STL 与原始的 HP STL 最明显的差别。例如在原始的 STL 中，下面是一个完整而正确的程序（虽然不是一个非常有趣的程序）：

```
#include <vector.h>

int main() {
  vector<int> v;
}
```

以 C++ standard 的观点，这不是一个合法程序。两个原因：

1. C++ 语言如今具备一个 namespace（命名空间）系统。根据 standard 规范，vector 并未定

义于 global namespace 中，而是定义于 namespace std 中。或者，以另一个角度来说，C++ standard 并不拥有一个名为 vector 的 class，而是拥有一个名为 std::vector 的 class。

2. 根据 standard 规范，class std::vector 不是声明于头文件 <vector.h>，而是声明于头文件 <vector>。

有很多方法可以将这个程序修改为符合 C++ standard 的所有规范。其中一种方法是：

```
#include <vector>

int main() {
  std::vector<int> v;
}
```

另一种做法是：

```
#include <vector>

int main() {
  using std::vector;
  vector<int> v;
}
```

第一个版本明白地以全名参考 std::vector。第二个版本中的这一行：

```
using std::vector;   // 此为一个 "using declaration"
```

意思是编译器应该将 vector 视为 std::vector 的缩写。第三种方式不采用 using declaration，而是采用所谓的 "using directive"：

```
using namespace std;
```

这样会将 std namespace 内的所有名称导入（import）。但是我并不鼓励此法。

除了少许例外，C++ 标准程序库的所有组件都定义于 namespace std 之中。每当你要使用 containers、algorithms，或标准程序库中的任何一个名称，都必须明白冠以修饰词 std::，否则就得使用 using declaration 将它导入（import）。

以操作符 == 和 < 为基础，定义出的一组操作符 !=、>、<= 和 >=（都是 templates），是以上规则的例外情况。它们自身并非声明于 namespace std 中，而是声明于名为 std::rel_ops 的一个嵌套命名空间中。

头文件的命名方式也比过去稍微复杂了些。C++ standard 将来自 HP STL 的所有头文件重新命名。新的头文件名称如 <vector>，不再具有文件延伸名 ".h"。新旧名称之间并没有一对一的映射关系，新组织出来的文件简直和重新命名的一样多。甚至 <vector> 就是这种情况。在 HP STL 中，template vector<> 声明于头文件 <vector.h>，class bit_vector（等价于特化的 vector<bool>）声明于 <bvector.h>。C++ standard 则将以上两者都声明于头文件 <vector>。

表 A.1：C++ Standard 和 HP STL 之间的头文件对映关系

C++ Standard 头文件	相应的 HP STL 头文件
<algorithm>	<algo.h>（大部分）
<deque>	<deque.h> （全部）
<functional>	<function.h>（大部分）
<iterator>	<iterator.h>（全部）
<list>	<list.h>（全部）
<memory>	<defalloc.h>（全部）
	<tempbuf.h>（全部）
	<iterator.h>（部分）
	<algo.h>（部分）
<numeric>	<algo.h>（部分）
<queue>	<stack.h>（部分）
<utility>	<pair.h>（全部）
	<function.h>（部分）
<stack>	<stack.h>（部分）
<vector>	<vector.h>（全部）
	<bvector.h>（全部）
<map>	<map.h>（全部）
	<multimap.h>（全部）
<set>	<set.h>（全部）
	<multiset.h>（全部）

表 A.1 是旧式头文件与 C++ standard 头文件的概略指引。更详细的信息应该查阅本书第三篇。篇中每个 functions 及每个 class 的文档都明确指出它被声明于哪个 standard 头文件及哪个旧式头文件。

「C++ standard 头文件与旧式名称全然不同」这一事实，证明对于可移植性是帮助而非阻碍。它给予实现者一个向下兼容的管道。

C++ standard 并未提及诸如 <algo.h> 和 <list.h> 等旧式名称。程序库实现者没有必要提供这类头文件，但也没有被禁止这么做。是的，实现者可以提供旧式的头文件名（而且很多人也的确这么做），并将其内容封装在 global namespace 而非 namespace std 之中。通常在这种情况下，头文件 <vector> 会包含实际的实现代码：

```
namespace std {
  template <class T, class Allocator>
  class vector {
    ...
  };
}
```

而头文件 <vector.h> 只包含以下两行：

```
#include <vector>
using std::vector;
```

这种做法并不属于 C++ standard 的规范，但它是被允许的，而且相当常见。从历史的角度来看，将 STL 头文件重新命名，其动机便是希望这种方式行得通。它容许你逐渐过渡到新式头文件名称，并明白地（explicit）使用 namespaces。

参考书目
Bibliography

[Amm97] Leen Ammeraal. ***STL for C++ Programmers***. John Wiley, Chichester, UK, 1997.

[Aus97] Matthew H. Austern. The SGI Standard Template Library. ***Dr. Dobb's Journal***, 22(8):18-27, Aug 1997.

[BFM95] Javier Barreiro, Robert Fraley, andDavid R. Musser. Hash tables for the Standard Template Library.Technical Report X3J16/94-0218 and WG21/N0605,International Standards Organization, Feb 1995.

[Boo94] Grady Booch. ***Object-Oriented Analysis and Design with Applications***. Benjamin Cummings, Redwood City, CA, Second edition, 1994.

[Bre98] Ulrich Breymann. ***Designing Components with the C++ STL: A New Approach to Programming.*** Addison Wesley Longman, Harlow, UK, 1998.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. ***Introduction to Algorithms***. MIT Press, Cambridge, MA, 1990.

[Dur64] Richard Durstenfeld. Algorithm 235, random permutation. ***Communications of the ACM***, 7(7):420, Jul 1964.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. ***The Annotated C++ Reference Manual***.Addison-Wesley, Reading, MA, 1990.

[FMR62] C. T. Fan, Mervin E. Muller, and Ivan Rezucha. Development of sampling plansby using sequential (item by item) selection techniques and digital computers.***Journal of the American Statistical Association,*** 57:387-402, 1962.

[GHG93] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. ***Larch: Languages and Tools for Formal Specification***. Springer-Verlag,New York, NY, 1993.

[GS96] Graham Glass and Brett Schuchert. ***The STL Primer***. Prentice-Hall, Englewood Cliffs, NJ, 1996. 551552

[Hoa62] C. A. R. Hoare. Quicksort. ***Computer Journal***, 5(1):10-15, 1962.

[ISO98] International Organization for Standardization (ISO), 1 rue de Varemb[e'],Case postale 56, CH-1211 Gen[e`]ve 20, Switzerland. ***ISO/IEC Final Draft International Standard 14882: Programming Language C++***, 1998.

[KM92] Deepak Kapur and David R. Musser. Tecton: A framework for specifying andverifying generic system components. Computer Science Technical Report 92-20,Rensselaer Polytechnic Institute, Jul 1992. Available viahttp://www.cs.rpi.edu/~musser/Tecton.

[KM97] Andrew Koenig and Barbara Moo. ***Ruminations on C++: A Decade of Programming Insight and Experience***. Addison Wesley Longman, Reading, MA, 1997.

[KMS82] Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Tecton: A languagefor manipulating generic objects. In J. Staunstrup, editor, ***Program Specification:Proceedings of Workshop, Aarhus, Denmark***, volume 134 of ***Lecture Notes in Computer Science***. Springer-Verlag, Berlin, 1982.

[KMS88] Aaron Kershenbaum, David R. Musser, and Alexander A. Stepanov. Higher-order imperative programming. Computer Science Technical Report 88-10, Renssalaer Polytechnic Institute, Apr 1988. Available via http://www.cs.edu/ ~musser/genprog.html.

[Knu97] Donald E. Knuth. ***Fundamental Algorithms***, volume I of ***The Art of Computer Programming***. Addison-Wesley, Reading, MA, Third edition, 1997.

[Knu98a] Donald E. Knuth. ***Seminumerical Algorithms***, volume II of ***The Art of Computer Programming***. Addison-Wesley, Reading, MA, Third edition, 1998.

[Knu98b] Donald E. Knuth. ***Sorting and Searching***, volume III of ***The Art of Computer Programming***. Addison-Wesley, Reading, MA, Second edition, 1998.

[Koe89] Andrew Koenig. ***C Traps and Pitfalls***. Addison-Wesley, Reading, MA, 1989.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. ***The C Programming Language***. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Lip91] Stanley Lippman. ***C++ Primer***. Addison-Wesley, Reading, MA, Second edition, 1991.

[LL98] Stanley Lippman and Josee Lajoie. ***C++ Primer***. Addison-Wesley, Reading, MA, Third edition, 1998.

[Mey97] Bertrand Meyer. ***Object-Oriented Software Construction***. Prentice-Hall, NewYork, Second edition, 1997.

[MO63] Lincoln E. Moses and Robert V. Oakford. ***Tables of Random Permutations***. Stanford University Press, Stanford, CA, 1963.

[MS89a] David R. Musser and Alexander A. Stepanov. ***The Ada Generic Library: Linear List Processing Packages***. Springer-Verlag, New York, NY, 1989.

[MS89b] David R. Musser and Alexander A. Stepanov. Generic programming. In P. Gianni,editor, ***Symbolic and Algebraic Computation: International Symposium ISSAC*** 1988,volume 358 of Lecture Notes in Computer Science, pages 13-25. Springer-Verlag, Berlin, 1989.

[MS96] David R. Musser and Atul Saini. ***STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library***. Addison-Wesley, Reading, MA, 1996.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. ***Software Practice and Experience***, 27(8):983-993, 1997.

[Nel95] Mark Nelson. ***C++ Programer's Guide to the Standard Template Library***. IDG Books Worldwide, Foster City, CA, 1995.

[Rob98] Robert Robson. ***Using the STL***. Springer-Verlag, New York, 1998.

[Sin69] Richard C. Singleton. Algorithm 347, an eAcient algorithm for sorting with minimal storage. ***Communications of the ACM***, 12:185-187, Mar 1969.

[SL95] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11 (R.1), Hewlett-Packard Laboratories, Nov 1995. Available by anonymous FTP from butler.hpl.hp.com.

[Str86] Bjarne Stroustrup. ***The C++ Programming Language***. Addison-Wesley, Reading, MA, 1986.

[Str94] Bjarne Stroustrup. ***The Design and Evolution of C++***. Addison-Wesley, Reading, MA, 1994.

[Str97] Bjarne Stroustrup. ***The C++ Programming Language***. Addison-Wesley, Reading, MA, Third edition, 1997.

[Vil94] Michael J. Vilot. An introduction to the Standard Template Library. ***C++ Report***,6(8):22-29, Oct 1994.

[Wil64] J. W. J. Williams. Algorithm 232, heapsort. ***Communications of the ACM***,7:347-348, 1964.

索引
Index

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Access expressions
  in Random Access Container, 135
  in Random Access Iterator, 105-106
accumulate algorithm, 281-283
Adaptable Binary Function concept, 55, 115-116
  function composition for, 433-434
  with function pointers, 426-427
  transforming into Adaptable Unary Function, 421-424
Adaptable Binary Predicate concept, 119, 429-431
Adaptable function objects, 55, 109
  Adaptable Binary Function, 115-116
  Adaptable Generator, 113-114
  Adaptable Unary Function, 114-115
Adaptable Generator concept, 113-114
Adaptable Predicate concept, 55-57, 118, 428-429
Adaptable Unary Function concept, 55, 57, 114-115
  function composition for, 431-432
  with function pointers, 424-426
  transforming Adaptable Binary Function into, 421-424
Adaptive algorithms, 196, 296
Adaptors
  container, 504-513
  function object, 56-58
  iterator, 46-47
  member function. *See* Member function adaptors
Addition in Random Access Iterator, 28, 104, 106
Address expressions in Allocator, 171
adjacent_difference algorithm, 287-289
adjacent_find algorithm, 25-26, 52-53, 202-204
advance algorithm, 38-40, 183-184
Algorithms, 3
  complexity of, 28-29
  concepts and modeling in, 16-19
  decoupling from containers, 7
  iterators in, 19-29
  linear search, 9-16
  mutating. *See* Mutating algorithms
  new, 526
  nonmutating. *See* Nonmutating algorithms
Allocate expressions, 170
allocator class, 187-189
Allocator concept, 166-171
Allocators and memory allocation, 78
  allocator class for, 187-189
  Allocator concept for, 166-171
  library changes for, 524-525
  for vector, 436
Annotated C++ Reference Manual (Ellis and Stroustrup), 515
Antisymmetry invariant
  in LessThan Comparable, 87
  in Strict Weak Ordering, 121
Area of Containers, 128
Argument type
  in Adaptable Unary Function, 114
  in Random Number Generator, 122
  in Unary Function
Arithmetic operations, 58
  divides, 378-379
  minus, 375-376
  modulus, 379-380
  multiplies, 376-377
  negate, 380-381
  plus, 373-375
  times, 376-377
Arrays, 59-60
  class for, 60-67
  past-the-end pointers for, 12
Art of Computer Programming, The (Knuth), 49
Ascending order invariants, 160
assign member function
  in deque, 460
  in list, 446

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

assign member function (*cont.*)
  in Sequence, 527
  in slist, 453
  in vector, 441
assignable_base class, 31
Assignable concept, 18-19, 83-84
Assignment expressions
  in Assignable, 84
  in Container, 128
  in Output Iterator, 98
  in Random Access Iterator, 106
  in Trivial Iterator, 93
Associated types, 18
  for Container, 126-127
  for function objects, 54-55
  for iterators, 33-41
Associative Container concept, 75-77, 145-149, 499
Associative containers, 460-461
  Associative Container, 145-149
  hash_map, 488-494
  hash_multimap, 499-504
  hash_multiset, 494-499
  hash_set, 484-488
  Hashed Associative Container, 161-166
  map, 466-473
  multimap, 478-484
  Multiple Associative Container, 152-153
  multiset, 473-478
  Pair Associative Container, 155-156
  set, 461-466
  Simple Associative Container, 153-154
  Sorted Associative Container, 156-160
  Unique Associative Container, 149-151
Asymmetric ranges, 13
Automatic type conversions, suppressing, 522

Back Insertion Sequence concept, 74, 143-145
back member function
  in Back Insertion Sequence, 74, 144
  in queue, 510
back_insert_iterator class, 348
  as adaptor, 46
  for Sequence, 75
Barreiro, Javier, 461
base member, 367
Basic function objects
  Binary Function, 112-113
  Generator, 110
  Unary Function, 111-112
begin member function
  for block, 61
  in Container, 128-129
  for containers, 68-69
Bidirectional Iterator concept, 27-28, 102-103
  for containers, 70
  refinement of, 29-30
bidirectional_iterator_tag class, 41, 179-181
binary_compose class, 433-434
binary_function class, 54-55, 372-373
Binary Function concept, 53-54, 112-113
binary_negate class, 429-431
Binary Predicate concept, 53-54, 117-118
binary_search algorithm, 306-308
Binary searches, 305-306
  binary_search, 306-308
  equal_range, 313-315
  lower_bound, 308-310
  upper_bound, 310-313
bind1st function, 421
bind2nd function, 422
binder1st class, 57, 421-422
binder2nd class, 57, 422-424
block class, 60-62
bool type with function objects, 54
Boolean operations. *See* Logical operations
Buckets for hash functions, 162
  Hash Function, 123
  Hashed Associative Container, 164
Buffers, temporary, 196-198

C++ language
  books about, xix, 515
  lack of support for concepts, 17, 40
  new keywords in, 521-524
  standardization of, 515
  templates in, xv, 13, 516-521
capacity member, 439
Capacity of vectors, 436-437
char_type member
  in istream_iterator, 356
  in istreambuf_iterator, 360
  in ostream_iterator, 358
  in ostreambuf_iterator, 363
Class templates, partial specialization of, 519-521
Classes
  container. *See* Container classes
  equivalence. *See* Equivalence classes
  inheritance of, 30-31
  wrapper, 15-16
clear function, 527
Collisions in Hash Function, 123
Commit or rollback semantics, 192
Comparing ranges algorithms
  equal, 220-222
  lexicographic_compare, 225-227

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Comparing ranges algorithms (*cont.*)
  mismatch, 222-224
Comparison expressions in Allocator, 170
Comparison operations, 58
  equal_to, 382-383
  greater, 386-387
  greater_equal, 388-390
  less, 384-386
  less_equal, 387-388
  not_equal_to, 383-384
Compilation model for templates, 516-517
Completeness invariants, 130
Complexity
  of algorithms, 28-29
  of hash functions, 162, 165-166
compose1 function, 431-432
compose2 function, 433-434
Concepts
  adaptable function objects, 113-116
  as abstraction, xvii-xviii, 7
  associative containers, 145-166
  basic, 83-89
  basic function objects, 110-113
  definition of, 17-18
  general container, 125-135
  and inheritance, 30-31
  iterators, 91-107
  and modeling, 16-17
  overloading, 39-40
  predicates, 116-121
  refinement of, 29-30
  requirements in, 16-19
  sequences, 136-145
  specialized, 122-124
const type with Assignable, 84
const_iterator type
  for block, 62
  for containers, 68-71
const_mem_fun_ref_t class, 414-416
const_mem_fun_t class, 412-414
const_mem_fun1_ref_t class, 418-420
const_mem_fun1_t class, 416-418
const_node_wrap class, 45-46
const_pointer type
  for Allocator, 168
  for block, 62
  for Container, 71, 126
const_reference type
  for Allocator, 168
  for block, 62
  for Container, 71, 126
const_reverse_iterator type, 69, 133-134
Constant iterators, 27, 44
  in Container, 127
  Forward Iterators, 100
  Trivial Iterators, 92
construct algorithm, 189-190
Constructors
  in Allocator, 171
  default, 84-85
  member templates for, 518
Container classes, xv, 59-80, 435-514
  adaptors, 504-513
  associative, 460-504
  deque, 455-460
  hash_map, 488-494
  hash_multimap, 499-504
  hash_multiset, 494-499
  hash_set, 484-488
  library changes for, 525-526
  list, 441-448
  map, 466-473
  multimap, 478-484
  multiset, 473-478
  priority_queue, 510-513
  queue, 507-510
  sequences, 435-460
  set, 461-466
  slist, 448-455
  stack, 505-507
  templates for, 518
  vector, 435-441
Container concept, 70-71, 125-130
Containers, 4, 125. *See also* Container classes; Container concept
  Allocator, 166-171
  arrays, 59-67
  Associative Container, 145-149
  Back Insertion Sequence, 143-145
  Container, 125-131
  decoupling from algorithms, 7
  defining, 79
  elements in, 67-68
  Forward Container, 131-133
  Front Insertion Sequence, 141-143
  Hashed Associative Container, 161-166
  hierarchy of, 70-71, 79
  iterators for, 68-70
  Multiple Associative Container, 152-153
  Pair Associative Container, 155-156
  Random Access Container, 135
  Reversible Container, 133-134
  selecting, 78-79
  Sequence, 136-141
  Simple Associative Container, 153-154
  Sorted Associative Container, 156-160
  trivial, 71-72
  Unique Associative Container, 149-151

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Containers (*cont.*)
  variable size, 72-78
Containment in modeling and refinement, 29
Contiguous storage invariants, 149
copy algorithm, 4, 53, 233-235
copy_backward algorithm, 236-237
Copy constructor expressions
  in Allocator, 169-170
  in Assignable, 84
  in Container, 128
  in Output Iterator, 98
Copying
  arrays, 60
  with copy, 233-235
  with copy_backward, 236-237
  Input Iterators, 20
  Output Iterators, 22
  with partial_sort_copy, 300-301
  with random_sample_n, 279-281
  ranges, 233-237
  with remove_copy, 256-257
  with remove_copy_if, 258-259
  with replace_copy, 246-247
  with replace_copy_if, 248-249
  with reverse_copy, 265-266
  with rotate_copy, 268-269
  with uninitialized_copy, 192-193
  with unique_copy, 262-264
count algorithm, 77, 147, 214-216
count_if algorithm, 216-218
Customizability of STL, 7

Deallocate expressions, 170
Decoupling of algorithms from containers, 7
Decrement expressions, 102
Default allocators, 525
Default Constructible concept, 18-19, 84-85
Default constructors, 84-85
  in Allocator, 169
  in Associative Container, 147
  in Hashed Associative Container, 162-163
  in Output Iterator, 98
  in Sequence, 137
  in Sorted Associative Container, 157
  in Trivial Iterator, 92
Default template parameters, 517-518
Defining
  containers, 79
  iterators, 47
  templates, 517
delete operator, 190
Deleting elements. *See* Removing elements
deque class, 455-456
Dereference expressions
  in Forward Iterator, 101
  in Input Iterator, 95
  in Output Iterator, 98-99
  in Trivial Iterator, 92-93
Dereferenceable iterators, 20
  Input Iterators, 20
  Output Iterators, 98
  Trivial Iterators, 92
destroy algorithm, 171, 190-192
Destructor expressions, 128
Dictionaries
  hash_map for, 489
  hash_multimap for, 499
Dictionary order comparisons, 225-227
Difference of adjacent elements, 287-289
Difference of sets
  set_difference for, 330-333
  set_symmetric_difference for, 333-336
difference_type type
  in Allocator, 169
  for block, 62
  in Container, 71, 127
  in Input Iterator, 94
  for iterators, 36-37
Dispatching algorithms, 38-41
distance algorithm, 181-183
distance_type, 43
Divide and conquer algorithm, 320
divides class, 58, 378-379
Dodgson, Charles Lutwidge, 516
Domains
  in Binary Function, 112
  in Random Number Generator, 122
  in Unary Function, 111
Doubly linked lists, 441-448
Duplicate elements
  unique for, 259-262
  unique_copy for, 262-264
Durstenfeld, Richard, 276

Efficiency of STL, 7
Element access expressions
  in Random Access Container, 135
  in Random Access Iterator, 105-106
Elements
  in containers, 67-68
  counting, 214-218
  partitions for, 273-275
  permutations of, 269-272
  random shuffling and sampling, 275-281
  removing, 253-264
  replacing, 243-249
  reversing, 264-266
  rotating, 266-269

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Elements (*cont.*)
  sorting, 291-305
  swapping, 237-240
Ellis, Margaret, 515
empty member function
  for block, 65
  in Container, 129
  in priority_queue, 513
  in queue, 509
  in stack, 507
Empty ranges, 12
end member function
  for block, 61
  in Container, 129
  for containers, 68-69
end_of_storage pointers, 436
End of stream value, 354
equal algorithm, 220-222, 361
equal_range algorithm, 313-315
  in Associative Container, 147-148
  in Sorted Associative Container, 159
equal_to class, 58, 382-383
Equality
  of Forward Iterators, 26-27, 47
  of Input Iterators, 20
  of pointers, 26
Equality Comparable concept, 18-19, 64, 85-86
Equality expressions
  in Equality Comparable, 85
  in Forward Container, 131
Equivalence classes, 89, 120-121
Equivalence invariants
  in Reversible Container, 134
  in Strict Weak Ordering, 121
Equivalence relations, 85-86, 89
Equivalent elements, 291
Equivalent objects, 88
erase member function, 254
  in Associative Container, 77, 148
  in Sequence, 73-74, 138-139
erase_after member function, 449, 453
even class, 50-51, 54
Exception safety, 192
explicit keyword, 522
Explicit specification of function template arguments, 197
Expressions in concepts, 18
Extensibility of STL, 7, 44

FIFO data structure, 507
fill algorithm, 249-250
fill_n algorithm, 250-251
Filling
  fill for, 249-250
  fill_n for, 250-251
  generate for, 251-252
  generate_n for, 252-253
  in Sequence, 136-138
  uninitialized_fill for, 194-195
  uninitialized_fill_n for, 195-196
find algorithm, 13, 199-200
  limitations of, 49
find member in Associative Container, 147
find_end algorithm, 209-211, 526
find_first_of algorithm, 204-206, 526
find_if algorithm, 50-51, 56-57, 200-202
finish pointers, 436
First argument type
  in Adaptable Binary Function, 115
  in Binary Function, 112
First in first out data structure. *See* FIFO data structure
Fixed size
  for arrays, 60
  for containers, 68, 128
for_each algorithm, 218-220
Formatted input, istream_iterator for, 354-357
Formatted output, ostream_iterator for, 357-359
Forward Container concept, 71, 131-133
Forward Iterator concept, 24-27, 100-102
  for containers, 70
  refinement of, 29-30
forward_iterator_tag class, 41, 179-181
Fraley, Bob, 461
Free lists in Allocator, 167
Front Insertion Sequence concept, 74, 141-143
front member function
  in Front Insertion Sequence, 74, 142
  in queue, 509-510
  in Sequence, 139
front_insert_iterator class, 345-346
  as adaptor, 46
  for Sequence, 75
Full specialization of templates, 35
Function call expressions
  in Binary Function, 113
  in Binary Predicate, 118
  in Generator, 110
  in Predicate, 117
  in Strict Weak Ordering, 120
  in Unary Function, 111
Function composition
  binary_compose for, 433-434
  unary_compose for, 431-432
Function object classes
  arithmetic operations, 373-382

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Function object classes (*cont.*)
binary_compose, 433-434
binary_function, 372-373
binary_negate, 429-431
binder1st, 421-422
binder2nd, 422-424
comparisons, 382-390
identity and projection, 394-400
logical operations, 390-394
member function adaptors, 403-420
pointer_to_binary_function, 426-427
pointer_to_unary_function, 424-426
unary_compose, 431-432
unary_function, 371-372
unary_negate, 428-429
Function object concepts
Adaptable Binary Function, 115-116
Adaptable Binary Predicate, 119
Adaptable Generator, 113-114
Adaptable Predicate, 118
Adaptable Unary Function, 114-115
Binary Function, 52-53, 112-113
Binary Predicate, 53-54, 117-118
Generator, 110
Hash Function, 123-124
Predicate, 53-54, 116-117
Random Number Generator, 122-123
Strict Weak Ordering, 119-121
Unary Function, 52-53, 111-112
Function objects, 6, 50-52, 109. *See also* Function object classes; Function object concepts
adaptors for, 56-58
associated types for, 54-55
predefined, 58
predicates and binary predicates, 53-54
unary and binary, 52-53
Function overloading, 39-41
Function pointers
with Adaptable Binary Function, 426-427
with Adaptable Unary Function, 424-426
Function template arguments, 197
Function templates, 521
Functors. *See* Function object classes; Function object concepts; Function objects

General container concepts
Container, 125-131
Forward Container, 131-133
Random Access Container, 135
Reversible Container, 133-134
Generalized copy constructor expressions, 170
Generalized numeric algorithms
accumulate, 281-283
adjacent_difference, 287-289
inner_product, 283-285
partial_sum, 285-287
generate algorithm, 53, 251-252
generate_n algorithm, 252-253
Generator concept, 53-54, 110
get_allocator member function
in deque, 459
in hash_map, 493
in hash_multimap, 503
in hash_multiset, 498
in hash_set, 488
in list, 445
in map, 471
in multimap, 482
in multiset, 477
in set, 465
in slist, 452
in vector, 440
get_temporary_buffer algorithm, 197-198, 526
greater class, 58, 386-387
greater_equal class, 58, 388-390

hash class, 400-402
Hash Function concept, 123-124
Hash functions, 161-164
hash_map class, 488-494
hash_multimap class, 499-504
hash_multiset class, 494-499
hash_set class, 484
Hashed Associative Container concept, 161-166
Header names, 528-530
Heap operations
is_heap, 343-344
make_heap, 336-337
pop_heap, 339-341
push_heap, 338-339
sort_heap, 342-343
Heap sort algorithm, 292
Hierarchy of containers, 70-71, 79

Identity and projection classes
identity, 394-395
project1st, 395-396
project2nd, 397-398
select1st, 398-399
select2nd, 399-400
identity class, 394-395
Identity of Forward Iterators, 26-27, 47
Identity invariants
in Equality Comparable, 86
in Trivial Iterator, 93
Immutability invariants
in Associative Container, 149

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Immutability invariants (*cont.*)
  in Simple Associative Container, 154
includes algorithm, 321-323
Incrementable iterators, 95
incrementing iterators, 183-184
  Forward Iterator, 101
  Input Iterator, 20-21, 96
  Output Iterator, 22, 99
Inference mechanism for iterator value types, 33-34
Inheritance
  compared to modeling and refinement, 30-31
  of iterator tags, 41, 181
  in STL, 7
Initializing arrays, 60
inner_product algorithm, 283-285
inplace_merge algorithm, 318-320
Input
  istream_iterator for, 354-357
  istreambuf_iterator for, 359-361
Input Iterator concept, 16-17, 20-22, 70, 94-96
input_iterator_tag class, 41, 179-181
Insert iterators
  back_insert_iterator, 348-351
  front_insert_iterator, 345-348
  insert_iterator, 351-354
insert member function
  in Associative Container, 77
  in Multiple Associative Container, 152-153
  in Sequence, 73-74, 137-141
  in Sorted Associative Container, 159
  in Unique Associative Container, 150-151
insert_after member function, 449, 453
insert_iterator class, 23, 351-354
  for Sequence, 75
Inserting
  in Associative Containers, 76-77
  in Sequences, 73-75, 139-141
int_node structure, 14-16
int_type member, 360
Intersection of sets, 327-330
Introsort algorithm, 293
Invalid pointers, 12
Invalidating dereferenceable iterators, 73-74, 92
Irreflexivity invariant
  in LessThan Comparable, 87
  in Strict Weak Ordering, 121
"Is a" relationships, 30
is_even function, 50-51
is_heap algorithm, 343-344
is_sorted algorithm, 303-305
istream_iterator class, 354-357
istream_type member
  in istream_iterator, 356
  in istreambuf_iterator, 361
istreambuf_iterator class, 359-361
iter_swap algorithm, 43, 238-239
iter_swap_impl algorithm, 43
iterator_category type, 41, 42-43
  in Input Iterator, 94
  in Output Iterator, 97
iterator class, 42, 185-186
Iterator classes
  back_insert_iterator, 348-351
  front_insert_iterator, 345-348
  insert, 345-354
  insert_iterator, 351-354
  istream_iterator, 354-357
  istreambuf_iterator, 359-361
  iterator, 42, 185-186
  iterator_traits, 177-179
  library changes to, 526-527
  ostream_iterator, 357-359
  ostreambuf_iterator, 362-363
  raw_storage_iterator, 368-370
  reverse_iterator, 363-368
  stream, 354-363
Iterator concepts
  Bidirectional Iterator, 102-103
  Forward Iterator, 100-102
  Input Iterator, 94-96
  Output Iterator, 96-100
  Random Access Iterator, 103-107
  Trivial Iterator, 91-93
iterator_traits class, 35-36, 42-43, 177-179
  partial specialization for, 520
  typename for, 524
iterator type
  for block, 62
  in Container, 127
  for containers, 68-69
  in Simple Associative Container, 154
Iterators, 4, 9, 16, 19. *See also* Iterator classes; Iterator concepts
  adaptors for, 46-47
  advance, 183-184
  for arrays, 60
  bidirectional, 27-28
  for block, 61-62
  constant and mutable, 27, 44-45
  for containers, 68-70
  defining, 46-47
  difference types for, 36-37
  dispatching algorithms and tags for, 38-41
  distance, 181-183
  forward, 24-27
  input, 20-22
  output, 22-24

本索引 ***concepts*** 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Iterators (*cont.*)
  primitives for, 177-186
  query functions for, 43
  random access, 28-29
  reference and pointer types for, 37-38, 45
  in Sequence, 136
  swapping, 238-239
  value types for, 33-36

Keys
  in Associative Containers, 76-77, 145-146
  in find_if, 201
  in Pair Associative Container, 155
  in searches, 49
  in Simple Associative Container, 154
Keywords, new, 521-524
Knuth, Donald E., xix, 49
Koenig, Andrew, 13, 60

Language changes
  default template parameters, 517-518
  member templates, 518-519
  new keywords, 521-524
  partial specialization, 519-521
  template compilation model, 516-517
Last in first out data structure. *See* LIFO data structure
Lee, Meng, 515-516
less class, 384-386
less_equal class, 58, 387-388
LessThan Comparable concept, 18-19, 64-65, 86-88
lexicographic_compare algorithm, 225-227
Library changes
  allocators, 524-525
  container adaptors, 525-526
  iterator interface, 526-527
  multiplies, 527
  new algorithms, 526
  Sequence, 527
  times, 527
Lifetime of Container elements, 68, 126
LIFO data structure, 505
line_iterator class, 4-5
Linear search, 9
  adjacent_find for, 202-204
  in C, 10-12
  in C++, 13-16
  description of, 49
  find for, 199-200
  find_first_of for, 204-206
  find_if for, 200-202
  generalizing, 49-52
  operation of, 14-16
  ranges in, 12-13
Linked lists
  list for, 441-448
  searches in, 14-16
  slist for, 448-455
list class, 441-448
  inserting elements in, 74
Load factors for hash functions, 162, 165
logical_and class, 390-391
logical_not class, 393-394
Logical operations
  and, 390-391
  not, 393-394
  or, 391-393
logical_or class, 391-393
Lookup operations, 76
lower_bound algorithm, 158, 308-310
lvalues, 37

make_heap algorithm, 336-337
make_pair function, 177
map class, 466-473
Mapped type, 155
max algorithm, 228-229
max_element algorithm, 231-232
max_size member function, 65
  in Allocator, 170, 171
  in Container, 129
Maximum algorithms
  max, 228-229
  max_element, 231-232
mem_fun function, 404
mem_fun_ref_t class, 406-408
mem_fun_t class, 57-58, 404-406
mem_fun1_ref_t class, 410-412
mem_fun1_t class, 408-410
Member access expressions, 93
Member function adaptors, 403-404
  const_mem_fun_ref_t, 414-416
  const_mem_fun_t, 412-414
  const_mem_fun1_ref_t, 418-420
  const_mem_fun1_t, 416-418
  mem_fun_ref_t, 406-408
  mem_fun_t, 404-406
  mem_fun1_ref_t, 410-412
  mem_fun1_t, 408-410
Member templates
  in Sequence, 139-140
  uses of, 518-519
Memory, 78
  allocator class for, 187-189
  Allocator concept for, 166-171
  construct for, 189-190

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Memory (*cont.*)
  destroy for, 190-192
  with Forward Iterator, 26
  library changes for, 524-525
  raw_storage_iterator for, 368-370
  temporary, 196-198
  uninitialized_copy for, 192-193
  uninitialized_fill for, 194-195
  uninitialized_fill_n for, 195-196
  for vector, 436
merge algorithm, 316-318
merge member function
  in list, 447
  in slist, 454-455
Merge sort algorithm, 320
Merging sorted ranges
  with inplace_merge, 318-320
  with merge, 316-318
min algorithm, 227-228
min_element algorithm, 229-230
Minimization of Hash Function collisions, 123
Minimum algorithms
  min, 227-228
  min_element, 229-230
minus class, 58, 375-376
mismatch algorithm, 222-224
Modeling, 16-19, 29-31
modulus class, 379-380
Moses, Lincoln E., 276
multimap class, 478-484
Multipass algorithms, 27
Multiple Associative Container concept, 76, 152-153
  hash_multimap as, 499
  hash_multiset as, 494
  multimap as, 478
  multiset as, 473
multiplies class, 58, 376-377, 527
multiset class, 473-478
Musser, David, 461
Mutable iterators, 27, 44
  Forward Iterator, 100
  Trivial Iterator, 92
Mutating algorithms, 32
  copying ranges, 233-237
  filling ranges, 249-253
  generalized numeric algorithms, 281-289
  partitions, 273-275
  permuting, 264-272
  random shuffling and sampling, 275-281
  removing elements, 253-264
  replacing elements, 243-249
  swapping elements, 237-240
  transform, 240-243
Names
  portability and standardization in, 527-530
  typename for, 523-524
Namespace system, 527-528
negate class, 58, 380-381
Negation
  of Adaptable Binary Predicate, 429-431
  of Adaptable Predicate, 428-429
New algorithms, 526
New keywords, 521
  explicit, 522
  typename, 522-524
new operator, 189
next_permutation algorithm, 269-271
node_wrap class, 15-16, 34
  as adaptor, 46
  as mutable iterator, 44-46
node_wrap_base class, 44-46
Nodes
  in list, 167, 442
  in trees, 336
Nonmutating algorithms, 32
  comparing ranges, 220-227
  counting elements, 214-218
  for_each, 218-220
  linear search, 199-206
  minimum and maximum, 227-232
  subsequence matching, 206-214
Nontype template parameters, 63
not_equal_to class, 58, 383-384
nth_element algorithm, 301-303
NULL pointers, 12
Numeric algorithms
  accumulate, 281-283
  adjacent_difference, 287-289
  inner_product, 283-285
  partial_sum, 285-287
numeric_limits class, 140

Oakford, Robert V., 276
'Omar Khayyām, 23
operator-> member, 527
operator<
  in LessThan Comparable, 86
  in Strict Weakly Comparable, 86
operator(), overloading, 50-51
operator[] member
  in block, 61, 69
  in hash_map, 493-494
  in map, 472-473
  in Random Access Container, 135
Operators, overloading, 15
Optimization, partial specialization for, 520

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Ordering
  in Forward Container, 132
  LessThan Comparable in, 86-88
  partial, 88
  in Random Access Iterator, 28
  in Sorted Associative Container, 160
  in stable sorts, 294-295
  Strict Weak Ordering in, 119-121
  Strict Weakly Comparable in, 88-89
  total, 88-89
ostream_iterator class, 23-24, 357-359
ostream_type member
  in ostream_iterator, 358
  in ostreambuf_iterator, 363
ostreambuf_iterator class, 362-363
Output
  ostream_iterator for, 357-359
  ostreambuf_iterator for, 362-363
Output Iterator concept, 22-24, 96-100
output_iterator_tag class, 41, 179-181
Overloading
  function templates, 521
  functions, 39-41
  operators, 15

Packaging, portability and standardization in, 527-530
Pair Associative Container concept, 76-77, 155-156
  hash_map as, 489
  hash_multimap as, 499
  map as, 466
  multimap as, 478
pair class, 175-177
Partial ordering, invariants in, 88
Partial specialization
  of class templates, 519-521
  of function templates, 521
  of types, 35, 177
partial_sort algorithm, 297-299
partial_sort_copy algorithm, 300-301
partial_sum algorithm, 285-287
partition algorithm, 273-274
Partition algorithms
  partition, 273-274
  stable_partition, 274-275
Past-the-end iterators, 20, 95, 354, 359
Past-the-end pointers, 12
Permutations
  next_permutation for, 269-271
  prev_permutation for, 271-272
Permuting algorithms, 264
  next_permutation, 269-271
  prev_permutation, 271-272
  reverse, 264-265
  reverse_copy, 265-266
  rotate, 266-268
  rotate_copy, 268-269
plus class, 58, 373-375
pointer type
  in Allocator, 168
  in block, 62
  in Container, 71, 126
  in Input Iterator, 94
  for iterators, 37-38, 45
pointer_to_binary_function class, 426-427
pointer_to_unary_function class, 57-58, 424-426
Pointers. *See also* Iterators
  in C, 11-12, 26
  in containers, 68, 126
  in Random Access Iterator, 28
Polymorphic function calls, 403
Polymorphism, compared to modeling and refinement, 31
pop member function
  in priority_queue, 513
  in queue, 510
  in stack, 504-505, 507
pop_back member function
  in Back Insertion Sequence, 74, 144
  in stack, 505
pop_front member function, 74, 142-143
pop_heap algorithm, 339-341
Portability, 515-516
  of allocators, 166, 524-525
  of container adaptors, 526
  with istream_iterator, 354
  and language changes, 516-524
  and library changes, 524-527
  naming and packaging in, 527-530
  with ostream_iterator, 354
Postdecrement expressions, 102-103
Postincrement expressions
  in Forward Iterator, 101
  in Input Iterator, 96
  in Output Iterator, 99
Predecessor element
  in list, 441
  in slist, 449
Predecrement expressions, 102
Predicate concept, 53-54, 116-117
Predicate function objects
  Adaptable Binary Predicate, 119
  Adaptable Predicate, 118
  Binary Predicate, 117-118
  Predicate, 116-117
  Strict Weak Ordering, 119-121

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Preincrement expressions
  in Forward Iterator, 101
  in Input Iterator, 95-96
  in Output Iterator, 99
prev_permutation algorithm, 271-272
previous member function in slist, 452
priority_queue class, 510-513
project1st class, 395-396
project2nd class, 397-398
ptrdiff_t type, 36-37
ptr_fun function, 57, 424-426, 426-427
push member function
  in priority_queue, 513
  in queue, 510
  in stack, 507
push_back member function
  in Back Insertion Sequence, 74, 144
  in stack, 505
push_front member function in Front Insertion Sequence, 74, 142
push_heap algorithm, 338-339

queue class, 507-510
Quicksort algorithm, 293

Random Access Container concept, 71, 135
Random Access Iterator concept, 28-29, 103-107
  for containers, 70
  difference types for, 36-37
Random Number Generator concept, 122-123, 402-403
random_access_iterator_tag class, 40-41, 179-181
random_sample algorithm, 277-279
random_sample_n algorithm, 279-281
random_shuffle algorithm, 276-277
Range constructor expressions
  in Hashed Associative Container, 163-164
  in Multiple Associative Container, 152
  in Sequence, 137
  in Sorted Associative Container, 157-158
  in Unique Associative Container, 150
Range erase expressions, 139
Range insert expressions, 138
Range invariants
  in Container, 130
  in Reversible Container, 134
Ranges
  for arrays, 59
  asymmetric, 13
  in Binary Function, 112
  comparing, 220-227
  copying, 233-237
  filling, 249-253
  in Generator, 110
  heaps from, 336-337
  of iterators, 95
  in linear search, 12-13
  for Output Iterators, 23
  in Random Number Generator, 122
  in Sequence, 136
  sorted. *See* Sorted ranges
  sorting, 291-305
  swapping, 239-240
  in Unary Function, 111
raw_storage_iterator class, 368-370
rbegin member function, 65
  for Reversible Container, 134
Reachability invariants, 106
Reachable iterators, 95
Reachable ranges, 12
reference type
  in Allocator, 168
  for block, 62
  in Container, 71, 126
  in Input Iterator, 94
  for iterators, 37-38, 44-45
Refinement of concepts, 29-31
Reflexivity
  in Equality Comparable, 86
  in modeling and refinement, 29
Regular types, 19, 83
rel_ops namespace, 528
Relative ordering in stable sorts, 294-295
remove algorithm, 253-255
remove member function
  in list, 447
  in slist, 454
remove_copy algorithm, 256-257
remove_copy_if algorithm, 258-259
remove_if algorithm, 255-256
remove_if member function
  in list, 447
  in slist, 454
Removing elements
  in Associative Containers, 76-77
  pop_heap for, 339-341
  remove for, 253-255
  remove_copy for, 256-257
  remove_copy_if for, 258-259
  remove_if for, 255-256
  in Sequences, 73-74
  unique for, 259-262
  unique_copy for, 262-264
rend member function, 65
  for Reversible Container, 134
replace algorithm, 25, 243-244

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

replace_copy algorithm, 246-247
replace_copy_if algorithm, 248-249
replace_if algorithm, 244-246
Requirements in concepts, 16-19
reserve member, 437, 440
resize function
  in Hashed Associative Container, 164-165
  in Sequence, 527
Result type
  in Adaptable Binary Function, 116
  in Adaptable Generator, 113
  in Adaptable Unary Function, 114
  in Binary Function, 112
  in Generator, 110
  in Random Number Generator, 122
  in Unary Function, 111
return_temporary_buffer algorithm, 198, 526
reverse algorithm, 264-265
reverse member function
  in list, 447-448
  in slist, 455
reverse_copy algorithm, 27-28, 265-266
reverse_iterator class, 56, 363-368
  as adaptor, 46
  for containers, 68-69
  in Reversible Container, 133
Reversible Container concept, 71, 133-134
rotate algorithm, 266-268
rotate_copy algorithm, 268-269

Sampling algorithms
  random_sample, 277-279
  random_sample_n, 279-281
Schweppe, E. J., 455
search algorithm, 206-209
search_n algorithm, 211-214, 526
Searches
  in Associative Containers, 77
  binary. *See* Binary searches
  linear search. *See* Linear searches
Second argument type
  in Adaptable Binary Function, 115
  in Binary Function, 112
select1st class, 398-399
select2nd class, 399-400
Sentinel elements, 199
Sequence concept, 73-75, 136-141, 527
Sequences
  Back Insertion Sequence, 143-145
  deque, 455-460
  Front Insertion Sequence, 141-143
  list, 441-448
  Sequence, 136-141
  slist, 448-455
  vector, 435-441
set class, 77, 321, 461-466
set_difference algorithm, 330-333
set_intersection algorithm, 327-330
Set operations, 320-321
  includes, 321-323
  set_difference, 330-333
  set_intersection, 327-330
  set_symmetric_difference, 333-336
  set_union, 324-327
set_symmetric_difference algorithm, 333-336
set_union algorithm, 324-327
Shakespeare, William, 527
Shuffling and sampling algorithms
  random_sample, 277-279
  random_sample_n, 279-281
  random_shuffle, 276-277
Simple Associative Container concept, 76-77, 153-154
  hash_multiset as, 494
  hash_set as, 484
  multiset as, 473
  set as, 461
Single pass algorithms, 21
Singly linked lists
  searches in, 14-16
  slist for, 448-455
Singular iterators, 20
  Output Iterator, 98
  Trivial Iterator, 92
Singular pointers, 12
Size
  of arrays, 60
  of containers, 68, 128
  of vectors, 436
size member function
  in Container, 129
  in priority_queue, 513
  in queue, 509
  in stack, 507
size_type member
  in Allocator, 169
  in block, 62
  in Container, 71, 127
  in priority_queue, 512
  in queue, 509
  in stack, 506
slist class, 448-455
snail_sort algorithm, 271
sort algorithm, 4, 292-294
sort member function
  in list, 448
  in slist, 455
sort_heap algorithm, 342-343

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

Sorted Associative Container concept, 77, 156-160
  map as, 466
  multimap as, 478
  multiset as, 473
  set as, 461
Sorted ranges
  binary searches in, 305-315
  from heaps, 342-343
  merging, 316-320
  set operations on, 320-336
Sorting, 4-6, 291-292
  is_sorted for, 303-305
  nth_element for, 301-303
  partial_sort for, 297-299
  partial_sort_copy for, 300-301
  sort for, 292-294
  sort member function for, 448, 455
  Sorted Associative Containers, 77, 156
  stable_sort for, 294-297
Specialization, partial
  of class templates, 519-521
  of function templates, 521
  of types, 35, 177
Specialized function objects
  hash, 400-402
  subtractive_rng, 402-403
splice member function
  in list, 446-447
  in slist, 453-454
splice_after member function, 454
Stability
  in partitions, 273-274, 274-275
  in sort, 292
  in stable_sort, 294-295
stable_partition algorithm, 274-275
stable_sort algorithm, 294-297
stack class, 505-507
Standardization, 515-516
  and language changes, 516-524
  and library changes, 524-527
  naming and packaging in, 527-530
start pointers, 436
std namespace, 527-528
Stepanov, Alexander, xv, xix, 515-516
strchr function, 10-11
Stream iterators
  istream_iterator, 354-357
  istreambuf_iterator, 359-361
  ostream_iterator, 357-359
  ostreambuf_iterator, 362-363
streambuf_type member
  in istreambuf_iterator, 360
  in ostreambuf_iterator, 363
Strict weak ordering, 88
Strict Weak Ordering concept, 58, 119-121
Strict Weakly Comparable concept, 88-89
Stroustrup, Bjarne, xix, 37, 60, 515
strtab_cmp class, 6
strtab_print class, 6
Subscripting in Random Access Iterator, 28
Subsequence matching algorithms
  find_end, 209-211
  search, 206-209
  search_n, 211-214
subsets, includes for, 321-323
Subtraction of iterators, 28, 104-105, 106
subtractive_rng class, 402-403
Successor elements in lists, 441
Summation
  accumulate for, 281-283
  partial_sum for, 285-287
swap algorithm, 237-238
  for block, 65-66
  in Container, 129-130
  partial specialization of, 521
swap_ranges algorithm, 239-240
Swapping elements
  iter_swap for, 238-239
  swap for, 237-238
  swap_ranges for, 239-240
Symmetric difference of sets, 333-336
Symmetry invariants
  in Back Insertion Sequence, 145
  in Bidirectional Iterator, 103
  in Equality Comparable, 86
  in Front Insertion Sequence, 143
  in Random Access Iterator, 106
system function, 220

Tags, iterator, 38-41, 179-181
Template parameters
  explicit specification of, 197
  nontype, 63
Template wrapper classes, 15-16
Templates
  compilation model for, 516-517
  default parameters for, 517-518
  member, 518-519
  partial specialization of, 519-521
  typename for, 522-524
Temporary buffers, 196-197
  get_temporary_buffer for, 197-198
  return_temporary_buffer for, 198
top member function
  in priority_queue, 513
  in stack, 507
Total ordering of objects, 88-89

本索引 *concepts* 皆以 sans serif 字形表现，classes 及其他 C++ 名称皆以 typewriter 字形表现。

traits_type member
  in istream_iterator, 356
  in istreambuf_iterator, 360
  in ostream_iterator, 358
  in ostreambuf_iterator, 363
transform algorithm, 53, 240-243
Transitivity in modeling and refinement, 29
Transitivity invariants
  in Equality Comparable, 86
  in LessThan Comparable, 88
  in Strict Weak Ordering, 121
  in Strict Weakly Comparable, 89
Traversal of doubly linked lists, 441
Trees, 336
Trivial Iterator concept, 91-93
trivial_container class, 71-72
typename keyword, 522-524
Types
  with concepts, 18-19
  for function objects, 54-55
  suppressing conversions of, 522

Unary Function concept, 53-54, 111-112
Unary function objects, 52-53
unary_compose class, 431-432
unary_function class, 54-55, 371-372
unary_negate class, 56-57, 428-429
Uniformity invariants in Random Number Generator, 123
Uniformity of Hash Function, 123
uninitialized_copy algorithm, 192-193, 526
uninitialized_fill algorithm, 194-195, 526
uninitialized_fill_n algorithm, 195-196, 526
Union of sets, 324-327
unique algorithm, 259-262
Unique Associative Container concept, 76-77, 149-151
  hash_map as, 489
  hash_set as, 484
  map as, 466
  set as, 461
unique member function
  in list, 447
  in slist, 454
unique_copy algorithm, 262-264
Uniqueness invariants, 151
upper_bound algorithm, 158, 310-313
Using declarations, 528

Valid expressions in concepts, 18
Valid iterators, 95, 136
Valid pointers, 12
Valid programs, concepts as, 17
Valid range invariants
  in Container, 130
  in Reversible Container, 134
Valid ranges, 12-13
  in iterators, 95
  in Sequence, 136
Value types in Trivial Iterator, 91
value_type, 42
  in Allocator, 168
  in Associative Container, 76, 145
  for block, 62
  in Container, 71, 126
  in Input Iterator, 94
  for iterators, 33-37
  in Pair Associative Container, 155
  in priority_queue, 512
  in queue, 509
  in stack, 506
Variable size containers, 68, 72
  allocators, 78
  Associative Container, 75-77
  Container, 128
  Sequence, 73-75
vector class, 4, 435-441
  vs. deque, 455-456
  inserting elements in, 74

Waterman, Alan, 278
Wrapper classes, 15-16
Write-only iterators, 23-24