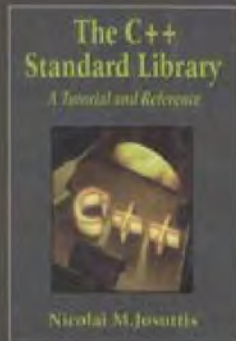


HUSTP



C++ 标准程序库

— 自修教程与参考手册 —



华中科技大学出版社
http://www.hustp.cn

侯捷/孟岩 译

以下为Genericity/STL经典好书
循序渐进，可收宏效

C++标准程序库之百
科全书：

内容全面 组织严明
举例广泛 索引清晰

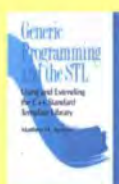
侯捷 / 孟岩 译
华中科技大学出版社



在STL深层运用过程
中，你会遇到一些难解的
问题和效率上的考虑。你
需要知道什么该做，什么
该避免。本书提供50个专
家条款。

既富学术价值又带有
长远参考价值之工具书。
STL学理概念及STL规格
之说明无人能出其右。完
备之STL规格文件与范例。

侯捷 / 黄俊尧 译
(繁体中文版)



向专家取经，学习内
存管理、算法、数据结构
之高阶泛型编程技法。让
你知其然并知其所以然。
精美的示意图，关键源码，
丰富注解，将给你带来最
佳的学习效果。

侯捷 著
华中科技大学出版社

泛型技术淋漓尽致，
令人目瞪口呆。试图将泛
型技术和设计模式结合在
一起。开创先河，独领风
骚。

侯捷 译
华中科技大学出版社



侯捷观点 —— 泛型技术的三个学习阶段

就像人们总是把目光放在艳丽的牡丹上而忽视了花旁的绿叶，作为一个广为人知的面向对象程序语言（OOP），C++所支持的另一种思维模式：泛型，被严重忽略了。说起红花绿叶，好似主观上划分了主从，其实面向对象思维和泛型思维两者之间无分主从。两者相辅相成，对程序开发将带来更大的突破。

面对新技术，我们的障碍在于心中的怯懦和迟疑。To be or not to be, that is the question! 不要像哈姆雷特一样犹豫不决，面对一项有用而且前途灿烂的技术，必须果敢。

泛型技术的学习有三个境界，**第一境界是运用STL**。对程序员而言，诸多抽象描述不如实象代码直指人心。**第二境界是了解泛型技术的内涵与STL的学理**。不但要理解STL的概念分类学（concept taxonomy）和抽象概念库（library of abstract concepts），最好再对各类STL组件抽样做点深刻追踪。STL源码都在手上，个案研究能使你对泛型技术以及STL学理有深刻的掌握。**第三个境界是扩充STL**。当STL不能满足我们的需求时，我们必须有能力动手写一个可融入STL体系的组件。要达到这个境界，先得彻底了解STL，也就必得先通过第二境界。

也许还应该加上**第零境界，C++template机制**。这是学习泛型技术及STL的第一道门槛，包括诸如 class templates, function templates, member templates, specialization, partial specialization……更往基础看去，STL大量运用了operator overloading，所以这一技法也必须熟稔。

C++ 标准程序库

The C++ Standard Library

自修教程与参考手册 (A Tutorial and Reference)

Nicolai M. Josuttis 著

侯捷 / 孟岩 译

C++ 标准程序库

The C++ Standard Library

Nicolai M. Josuttis

Copyright ©1999 by Addison Wesley Longman, Inc.

Simplified Chinese Copyright 2002 by Huazhong Science and Technology University Press and Pearson Education North Asia Limited.

All rights Reserved.

Published by arrangement with Pearson Education North Asia Limited, a Pearson Education Company.

版权所有,翻印必究。

本书封面贴有华中科技大学出版社(原华中理工大学出版社)激光防伪标签,封底贴有“Pearson Education”激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

C++标准程序库/(德)Nicolai M. Josuttis 著;侯捷/孟岩译

武汉:华中科技大学出版社,2002年9月

ISBN 7-5609-2782-3

I. C…

I. ①N… ②侯… ③孟…

Ⅱ. C++-程序设计

Ⅳ. TP312

责任编辑:周 筠(<http://yeka.xilubbs.com> junzhou@public.wh.hb.cn)

技术编辑:孟 岩

出版发行:华中科技大学出版社 (武昌喻家山 邮编:430074)

录排:华中科技大学惠友科技文印中心

印刷:湖北新华印务有限公司

开本:787×1092 1/16

印张:51.75

字数:800 000

版次:2002年9月第1版

印次:2002年9月第1次印刷

印数:1—12 000

定价:108.00元

ISBN 7-5609-2782-3/TP·478

巨细靡遗 井然有序

（侯捷译序）

自从 1998 年 C++ *Standard* 定案以后，C++ 程序库便有了大幅扩充。原先为大家所熟知、标准规格定案前酝酿已久的 STL（Standard Template Library，标准模板程序库），不再被单独对待，而是被纳入整个 C++ 标准程序库（Standard Library）。同时，原有的程序库（如 *iostream*）也根据泛型技术（*generics*）在内部做了很大的修改。可以说，C++ *Standard* 的发布对 C++ 社群带来了翻天覆地的大变动——不是来自语言本身，而是来自标准程序库。这个变动，影响 C++ 程序编写风格至巨，C++ 之父 Bjarne Stroustrup 并因此写了一篇文章：*Learning Standard C++ as a New Language*（载于 *C/C++ User's Journal*, 1999/05）。

我个人于 1998 年开始潜心研究泛型技术和 STL，本书英文版《*The C++ Standard Library*》甫一出版便成为我学习 C++ 标准程序库的最重要案头工具书之一。小有心得之后，我写过数篇相关技术文章，从来离不开本书的影响和帮助。我曾经把 STL（代表泛型技术目前最被广泛运用的一个成熟产品，也是 C++ 标准程序库的绝大成分）的学习比喻为三个境界（或层次）：

- 第一境界：熟用 STL
- 第二境界：了解泛型技术的内涵与 STL 的学理乃至实作
- 第三境界：扩充 STL

不论哪一个阶段，你都能够从本书获得不同程度的帮助。

第一阶段（对大多数程序员有立竿见影之效），我们需要一本全面而详尽的教程，附带大量设计良好的范例，带领我们认识十数个 STL 容器（*containers*）、数十个 STL 算法（*algorithms*）、许许多多的迭代器（*iteartors*）、配接器（*adapters*）、仿函数（*functors*）……的各种特性和用途。这些为数繁多的组件必须经过良好的编排组织和索引，才能成就一本效果良好、富教育性又可供长久查阅的案头工具书。

在这一阶段里，本书表现极为优异。书中运用许多图表，对所有 STL 组件的成员做了极其详尽的整理。更值得称道的是书中交叉参考（cross reference）做得非常好，在许多关键地点告诉读者当下可参见哪一章哪一节哪一页，对于阅读和学习带来很大的帮助（本中文版以页页对译方式保留了所有交叉参考和索引）。

第二阶段（从 STL 的运用晋升至泛型技术的学习），我们需要一些关键的 STL 源代码（或伪码，pseudo code），帮助我们理解关键的数据结构、关键的编程技术。认识这些关键源代码（或伪码）同时也有助提升第一阶段的运用深度（学会使用一样东西，却不知道它的道理，不高明^①）。

本书很多地方都提供了 C++ 标准程序库的关键源代码。不全面，但很关键。

第三阶段（成为一位泛型技术专家；打造自己的 STL 兼容组件），我们需要深入了解 STL 的设计理念和组织架构^②，并深入（且全面地）了解 STL 实作手法^③。是的，不入虎穴，不能得虎子；彻底了解 STL 如何被打造出来之后，你才能写出和 STL 水乳交融、完美整合的自定义组件（user-defined components）。

本书对第三阶段的学习也有相当帮助。虽然没能提供全面的 STL 源码并分析其技术（那需要另外 800 页^④），却提供了为数不少的订制型组件实作范例：p191, p213 提供了一个执行期指定排序准则并运用不同排序准则的实例，p219 提供了一个自定义容器（虽然只是个简单的包装类别），p222 提供了一个“reference 语义”示范作法，p285 提供了一个针对迭代器而设计的泛型算法，p288 提供了一个用于关联式容器的定制型 inserter，p294 有一个自定的排序准则，p441 有一个自定的（安全的）stack，p450 有一个自定的（安全的）queue，p504 有一个自定的 traits class for string，p614 有一个自定的 stream 操控器，p663 有一个自定的 stream 缓冲区，p735 有一个自定的内存配置器（allocator）。

^① 这是乍见之下令人错愕的一句话。看电视需要先了解电视的原理吗？呵呵，话讲白了就没意思了。这句话当然是对技术人员说的。

^② 这方面我推荐你看《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》，by Matthew H. Austern, Addison Wesley 1998。详见稍后说明。中译本《泛型程序设计与 STL》，侯捷/黄俊尧合译，碁峰，2001。

^③ 这方面我推荐你看《STL 源码剖析，*The Annotated STL Sources*》by 侯捷，碁峰，2002。详见稍后说明。

除了众所瞩目的 STL，本书也涵盖了一般不被归类为 STL 的 String 程序库，以及一般不被视为关键的 IOSTream 和 Locale 程序库⁴。三部分互有关连，以 IOSTream 为主干。在 GUI（图形使用接口）和 application framework（应用程序框架）当道的今天，IOStream 提供的输出输入可能对大部分人失去了价值，但如果你希望开拓 OO 技术视野，IOStream 是一颗沉睡的珠宝。



泛型技术不仅在 C++ 被发扬光大，在 Java 上也有发展⁵，在 C# 上亦被众人期待。从目前的势头看，泛型技术（Generics）或许是面向对象（Object Oriented）技术以来程序编写方面的又一个巨大冲击。新一代 C++ 标准程序库⁶将采用更多更复杂更具威力的泛型技术，提供给 C++ 社群更多更好更具复用价值的组件。

不论你要不要、想不想、有没有兴趣在你的程序编写过程中直接用上泛型技术，至少，在 C++ 程序编写过程中你已经不可或缺于泛型技术带来的成熟产品：C++ 标准程序库。只要你具备 C++ 语言基础，本书便可以带领你漂亮地运用 C++ 标准程序库，漂亮地提升你的编程效率和程序品质。

面对陌生，程序员最大的障碍在于心中的怯懦。To be or not to be, that is the question! 不要像哈姆雷特一样犹豫不决。面对光明的技术，必须果敢。



关于术语的处理，本书大致原则如下：

1. STL 各种数据结构名称皆不译，例如 array, vector, list, deque, hast table, map, set, stack, queue, tree...。虽然其中某些已有约定俗成的中文术语，但另一些没有既标准又被普遍运用的中文名称，强译之会令读者瞠目以对，部分译部分不译则阅读时词性平衡感不佳（例如“面对向量和 deque 两种容器...”就不如“面对 vector 和 deque 两种容器...”读起来顺畅）。因此，数据结构名称全部不译。直接呈现这些简短的英文术语，可能营造更突出的视觉效果，反而有利阅读。技术书籍的翻译不是为了建立全中文化阅读环境，我们的读者水

⁴ 这方面我见过的唯一专著是《Standard C++ IOStreams and Locales - Advanced Programmer's and Reference》，by Angelika Langer and Klaus Krefl, Addison Wesley 2000.

⁵ (1) GJ: A Generic Java, by Philip Wadler, Dr. Dobb's Journal February 2000.

(2) JSR- 000014: Adding Generics to the Java Programming Language, <http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html>.

⁶ 请参考 <http://www.boost.org/>，这个程序库据称将成为下一代 C++ 标准。

平也不可能受制于这些英文单词。

2. STL 六大组件的英文名称原打算全部保留，但由于处处出现，对版面的中英文比例形成视觉威胁，因此全部采用以下译名：**container** 容器，**algorithm** 算法，**iterator** 迭代器，**adapter** 配接器，**functor** 仿函数⁷，**allocator** 配置器。
3. 任何一个被保留的英文关键术语，其第一次（或前数次）出现时尽可能带上中文名称。同样地，任何关键的中文术语，我也会时而让它中英并陈。



关于编排，本书原则如下：

1. 全书按英文版页次编排，并因而得以保留原书索引。索引词条皆不译。
2. 中文版采用之程序代码字体（Courier New 8.5）比文本字体（细明体 9.5）小，英文版之程序代码字体却比其文本字体大，且行距宽。因此中文版遇有大篇幅程序行表时，为保持和英文版页次相应，便会出现较多留白。根据我个人对书籍的经验，去除这些留白的最后结果亦不能为全书节省五页十页；填满每一处空白却丧失许多立即可享的好处，为智者不取^⑤。



一旦你从本书获得了对 C++ 标准程序库运用层面的全盘掌握与实践经验之后，可能希望对 STL 原理乃至实作技术做更深的研究，或甚至对泛型编程（Generic Programming）产生无比狂热。在众多相关书籍之中，下面是我认为非常值得继续进修的四本书：

1. 《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》，by Matthew H. Austern, Addison Wesley 1998。本书第一篇（前五章）谈论 STL 的设计哲学、程序库背后的严密架构和严谨定义。其中对于 STL 之异于一般程序库，有许多重要立论。其余部分（第二篇、第三篇）是 STL 的完整规格（分别从 concepts 的角度和 components 的角度来阐述），并附范例程序。

⁷ 原书大部分时候使用 **function object**（函数对象）一词，为求精简及突出，中文版全部改用其另一个名称 **functor**（仿函数）（见第 8 章译注）。

2. 《STL 源码剖析, *The Annotated STL Sources*》by 侯捷, 暮峰, 2002。本书剖析 STL 实作技法, 详实揭示并注释 STL 六大组件的底层实作, 并以公认最严谨的 SGI (Silicon Graphics Inc.) STL 版本为剖析对象。附许多精彩分析图, 对于高度精巧的内存配置策略、各种数据结构、各种算法、乃至极为“不可思议”的配接器 (adapter) 实作手法, 都有深入的剖析。
3. 《*Effective STL*》, by Scott Meyers, Addison Wesley 2001。本书定位为 STL 的深层运用。在深层运用的过程中, 你会遇到一些难解的问题和效率的考虑, 你需要知道什么该做、什么该避免。本书提供 50 个专家条款。请注意, 深层运用和效率调校, 可能需要读者先对底部机制有相当程度的了解。
4. 《*Modern C++ Design*》by Andrei Alexandrescu, Addison Wesley 2001。将泛型技术发挥到淋漓尽致、令人目瞪口呆的一本书籍。企图将泛型技术和设计模式 (design patterns) 结合在一起。是领先时代开创先河的一本书。



本书由我和孟岩先生共同完成。孟岩在大陆技术论坛以 C++/OO/Generics 驰名, 见解深雋, 文笔不凡。我很高兴和他共同完成这部作品。所谓合译, 我们两人对全书都有完整的参与 (而非你一半我一半的对拆法), 最终由我定稿。本书同时发行繁体版和简体版, 基于两岸计算机术语的差异性, 简体版由孟岩负责必要转换。

侯捷 2002/05/23 于新竹

<http://www.jjhou.com> (繁体网站)

<http://jjhou.csdn.net> (简体网站)

jjhou@jjhou.com (个人电子邮箱)

孟岩译序

IT 技术书籍市场，历来是春秋战国。一般来说，同一个技术领域里总会有那么数本、十数本、甚至数十本定位相似的书籍相互激烈竞争。其中会有一些大师之作脱颖而出，面南背北，黄袍加身。通常还会有后来者不断挑战，企图以独到特色赢得自己的一片天地。比如说在算法与数据结构领域，D. E. Knuth 的那套《*The Art of Computer Programming*》一至三卷，当然是日出东方，惟我独尊。但是他老人家的学生 Robert Sedgewick 凭着一套更贴近实用的《*Algorithms in C*》系列，也打出了自己的一片天下，成为很多推荐列表上的首选。就 C++ 应用经验类书籍来说，Scott Meyers 的《*Effective C++*》称王称霸已经多年，不过其好友 Herb Sutter 也能用一本《*Exceptional C++*》获得几乎并驾齐驱的地位。嗨，这不是很正常的事吗？技术类书籍毕竟不是诗词歌赋。苏轼一首“明月几时有，把酒问青天”，可以达到“咏中秋者，自东坡西江月后，余词尽废”的程度，但怎么可能想象一本技术著作达到“我欲乘风归去，又恐琼楼玉宇，高处不胜寒”的境界！谁能够写出一本技术书，让同一领域后来者望而却步，叹为观止，那才是大大的奇迹！

然而，您手上这本《*The C++ Standard Library*》，作为 C++ 标准程序库教学和参考类书籍的定音之作，已经将这个奇迹维持了三年之久。按照 IT 出版界时钟，三年的时间几乎就是半个世纪，足以锤炼又一传世经典！

1998 年 C++ Standard 通过之后，整个 C++ 社群面临的最紧迫任务，就是学习和理解这份标准给我们带来的新观念和新技术。而其中对于 C++ 标准程序库的学习需求，最为迫切。C++ 第二号人物 Andrew Koenig 曾经就 C++ 的特点指出：“语言设计就是程序库设计，程序库设计就是语言设计”¹。C++ Standard 对程序库所作的巨大扩充和明确规范，实际上即相当于对 C++ 语言的能力作了全面提升与扩展，意味着你可以站在无数超一流专家的肩上，将最出色的思想、设计与技术纳入囊中，

¹ “Language design is library design, library design is language design”，参见 Andrew Koenig, Barbara Moo 合著《*Ruminations on C++*》第 25, 26 章标题。

让经过千锤百炼的精美代码成为自己软件大厦的坚实基础。可以说，对于大多数程序员来说，标准 C++ 较之于“ARM 时代”之最大进步，不是语言本身，而恰恰是标准程序库。因此，我们可以想象当时人们对 C++ 标准程序库教学类书籍的企盼，是何等热切！

一方面是已经标准化了的成熟技术，另一方面是万众期待的眼神，我们完全有理由认为，历史上理应爆发一场鱼龙混杂的图书大战。它的典型过程应该是这样：先是一批快刀手以迅雷不及掩耳盗铃之势²推出一堆敛财废纸，然后在漫长的唾骂与期待中，大师之作渐渐脱颖而出。大浪淘沙，最后产生数本经典被人传颂。后虽偶有新作面世，但波光点点已是波澜不兴。

然而，这一幕终究没有在“C++ 标准程序库教学与参考书籍”领域内出现。时至今日，中外技术书籍市场上这一领域内的书籍为数寥寥，与堆积如山的 C++ 语言教学类书籍形成鲜明对比。究其原因，大概有二，一是这个领域里的东西毕竟份量太重，快刀手虽然善斩乱麻，对于 C++ 标准程序库这样严整而精致的目标，一时也难以下手。更重要的原因则恐怕是 1999 年 8 月《The C++ Standard Library》问世，直如横刀立马，震慑天下。自推出之日起至今，本书在所有关于 C++ 标准程序库的评论与推荐列表上，始终高居榜首，在 Amazon 的销量排行榜上名列所有 C++ 相关书籍之最前列。作者仅凭一书而为天下知，成为号召力可与 Stan Lippman, Hurb Sutter 等“经典”C++ 作家比肩的人物。此书之后，虽然仍有不少著作，或深入探讨标准程序库的某些组件，或极力扩展标准库倡导的思想与技术，但是与《The C++ Standard Library》持同一路线的书籍，再没有出现过。所谓泰山北斗已现，后来者已然无心恋战。

于是有了这样的评论：“如果你只需要一本讲述 C++ 标准程序库和 STL 的书籍，我推荐 Nicolai Josuttis 的《The C++ Standard Library》¹。它是你能得到的唯一一本全面讲述 C++ 标准程序库的书，也是你能想象的最好的一本书。”这种奇异情形在当今技术书坛，虽然不是绝无仅有，也是极为罕见。

究竟这本书好到什么程度，可以获得这么高的评价？

我正是带着这份疑问，接受侯捷先生的邀请，着手翻译这本经典之作。随着翻译过程的推进，我也逐渐解开了心中的疑惑。在我看来，这本书的特点有四：内容详实，组织严密，态度诚恳，深入浅出。

² 此处非笔误，而是大陆流行的一句“新俚语”，意思十分明显，就是“迅雷不及掩耳”地“掩耳盗铃”。

首先，作为一本程序库参考手册，内容详实全面是一项基本要求。但是，本书在这方面所达到的高度可以说树立了一个典范。本书作者一开始就提出一个极高的目标，要帮助读者解决“使用 C++ 标准程序库过程中所遇到的所有问题”。众所周知，C++ 标准程序库是庞然大物，每一部分又有很精深的思想和技术，既不能有所遗漏，又不能漫无边际地深入下去，何取何舍，何去何从，难度之大可想而知！作者在大局上涵盖了 C++ 标准程序库的全部内容，在此基础上又对所有组件都进行细致的、立体式的讲解。所谓立体式讲解，就是对于一个具体组件，作者首先从概念上讲解其道理，然后通过漂亮的范例说明其用法，申明其要点，最后再以图表或详解方式给出参考描述。有如钱塘江潮，层层叠叠，反反复复，不厌其烦。读完此书，我想您会和我一样感受冲击，并且完全体认作者付出的巨大心血。

C++ 标准程序库本身就是一个巨大的有机整体，加上这本书的立体讲解方式，前后组织和对应的工作如果做不好，很容易会使整部书显得散乱。令人钦佩的是，这本书在组织方面极其严密，几无漏洞。相关内容的照应、交叉索引、前后对应，无一不处理得妥善曼妙。整体上看，整本书就像一张大网，各部分内容之间组织严谨，契合密切，却又头绪清晰，脉络分明，着实难能可贵。我在阅读和翻译过程中，常常诧异于其内容组织的精密程度，简直像德国精密机械一样分毫不差——后来才想到，本书作者 Nicolai Josuttis 就是德国人，精密是德意志民族的性格烙印，真是名不虚传！

说起德意志民族，他们的另一个典型性格就是诚实坦率。这一点在这本书同样有精彩的展现。身为 C++ 标准程序库委员会成员，作者对于 C++ 标准程序库的理解至深，不但清楚知道其优点何在，更对其缺陷、不足、不完备和不一致的地方了如指掌。可贵的是，在这些地方，作者全不避讳，开诚布公，直言不讳，事实是什么样就是什么样，绝不文过饰非，绝不含混过关。作为读者，我们不仅得以学到好东西，而且学到如何绕开陷阱和障碍。一个最典型的例子就是对于 `valarray` 的介绍，作者先是清清楚楚地告诉读者，由于负责该组件设计的人中途退场，这个组件没有经过细致的设计，最好不要使用。然后作者一如既往，详细介绍 `valarray` 的使用，完全没有因为前面的话而稍微有所懈怠。并且在必要的地方将 `valarray` 的设计缺陷原原本本地指出来，让读者口服心服。读到这些地方，将心比心，我不禁感叹作者的坦诚与无私，专精与严谨。

本书最具特色之处，就是其内容选取上独具匠心，可谓深入浅出。本书的目的除了作为手册使用，更是一本供学习者阅读学习的“tutorial”（自学教本）。也就是说，除了当手册查阅，你也可以捧着它一篇一篇地阅读学习，获得系统化的坚实知识。一本书兼作“tutorial”和“reference”，就好像一本字典兼作“作文指南”，没

有极高的组织能力和精当的内容选择，简直难以想象会搞成什么样子。了不起的是本书不仅做到了，而且让你感觉，学习时它是一本最好的“tutorial”，查阅时它是一本最好的“reference”，我要说，这是个奇迹！单从学习角度来说，本书极为实用，通过大量鲜明的例子和切中要害的讲解让你迅速入门，而且绝不仅仅浅尝辄止，而是不失时机地深入进去，把组件的实作技术和扩展方法都展现给读者。单以 STL 而论，我经常以侯捷先生提出的“STL 学习三境界”来描述一本书的定位层次，这本书就像一座金字塔，扎根于实用，尖锋直达“明理”和“扩展”层次。从中你可以学到“reference 语意”的 STL 容器、smart pointer（智能指针）的数种实现、扩充的组合型仿函数（composing function object）、STL 和 IOStream 的扩展方法、定制型的配置器（allocator）设计思路等等高级技术，也能学到大量的实践经验，比如 vector 的使用技巧，STL 容器的选择，basic_string<> 作为容器的注意事项等等。可以这么说，这本书足以将你从入门带到高手层次，可谓深入浅出，精彩至极！

我很高兴自己第一次进行技术书籍翻译，就能够碰到这样一本好书，这里要深深感谢侯捷先生给我一辈子都庆幸的机会。翻译过程出乎意料地艰辛，前后持续将近 10 个月。我逐字逐句地阅读原文，消化理解，译成自以为合适的中文，然后再由侯先生逐字逐句地阅读原文，对照我的粗糙译文，进行修订和润色，反复品味形成最终译稿。作为译者，侯先生和我所追求的是，原书技术的忠实呈现加上中文化、中国式的表达。我们为此花费了巨大的心力，对我来说，付出的心血远远超过一般翻译一本书的范畴。虽然最终结果需要广大读者评论，但今天面对这厚厚的书稿，我问心无愧地享受这份满足感。我最大的希望是，每一位读者在学习和查阅这本中文版的时候，完全忘掉译者曾经的存在，感觉不到语言的隔阂，自由地获取知识和技术。对于一个初涉技术翻译的人来说，这个目标未免太贪心了，可是这始终会是我心里的愿望。一个译者应该就是为了被忽略而努力的。

最后，感谢侯先生一直以来对我的欣赏和帮助，感谢您给我的机会，我十分荣幸！感谢华中科技大学出版社的周筠老师，您始终友好地关注着我，鼓励着我。感谢 CSDN 的蒋涛先生，您的热情鼓励始终是我的动力。感谢我的父母、弟弟，你们是最爱我的人，是最坚强的支柱！感谢曾经帮助过我，曾经关心过我的每一个人，无论你现在怎样，我为曾经拥有过的，仍然拥有着的每一片快乐和成果，衷心地感谢你！

祝各位读书快乐！

孟岩 2002 年 5 月于北京

目录

巨细靡遗·井然有序（侯捷译序）	a
孟岩译序	g
目录（Contents）	v
前言（Preface）	xvii
致谢（Acknowledgments）	xix
1 关于本书	1
1.1 缘起	1
1.2 阅读前的必要基础	2
1.3 本书风格与结构	2
1.4 如何阅读本书	5
1.5 目前发展形势	5
1.6 范例程序代码及额外信息	5
1.7 响应	5
2 C++ 及其标准程序库简介	7
2.1 沿革	7
2.2 新的语言特性	9
2.2.1 template（模板）	9
2.2.2 基本型别的显式初始化（Explicit Initialization）	14
2.2.3 异常处理（Exception Handling）	15
2.2.4 命名空间（Namespaces）	16
2.2.5 bool 型别	18
2.2.6 关键字 explicit	18
2.2.7 新的型别转换操作符（Type Conversion Operators）	19
2.2.8 常数静态成员（Constant Static Members）的初始化	20
2.2.9 main() 的定义式	21
2.3 复杂度和 Big-O 表示法	21

3 一般概念 (General Concepts)	23
3.1 命名空间 (namespace) std	23
3.2 头文件 (Header Files)	24
3.3 错误 (Error) 处理和异常 (Exception) 处理	25
3.3.1 标准异常类别 (Standard Exception Classes)	25
3.3.2 异常类别 (Exception Classes) 的成员	28
3.3.3 抛出标准异常	29
3.3.4 从标准异常类别 (Exception Classes) 中派生新类别	30
3.4 配置器 (Allocators)	31
4 通用工具 (Utilities)	33
4.1 Pairs (对组)	33
4.1.1 便捷函数 make_pair()	36
4.1.2 Pair 运用实例	37
4.2 Class auto_ptr	38
4.2.1 auto_ptr 的设计动机	38
4.2.2 auto_ptr 拥有权 (Ownership) 的转移	40
4.2.3 auto_ptrs 作为成员之一	44
4.2.4 auto_ptrs 的错误运用	46
4.2.5 auto_ptr 运用实例	47
4.2.6 auto_ptr 实作细目	51
4.3 数值极限 (Numeric Limits)	59
4.4 辅助函数	66
4.4.1 挑选较小值和较大值	66
4.4.2 两值互换	67
4.5 辅助性的“比较操作符” (Comparison Operators)	69
4.6 头文件 <cstdint> 和 <cstdlib>	71
4.6.1 <cstdint> 内的各种定义	71
4.6.2 <cstdlib> 内的各种定义	71
5 Standard Template Library (STL, 标准模板库)	73
5.1 STL 组件 (STL Components)	73
5.2 容器 (Containers)	75
5.2.1 序列式容器 (Sequence Containers)	76
5.2.2 关联式容器 (Associative Containers)	81
5.2.3 容器配接器 (Container Adapters)	82

5.3 迭代器 (Iterators)	83
5.3.1 关联式容器的运用实例	86
5.3.2 迭代器分类 (Iterator Categories)	93
5.4 算法 (Algorithms)	94
5.4.1 区间 (Ranges)	97
5.4.2 处理多个区间	101
5.5 迭代器之配接器 (Iterator Adapters)	104
5.5.1 Insert Iterators (安插型迭代器)	104
5.5.2 Stream Iterators (流迭代器)	107
5.5.3 Reverse Iterators (逆向迭代器)	109
5.6 更易型算法 (Manipulating Algorithms)	111
5.6.1 移除 (Removing) 元素	111
5.6.2 更易型算法和关联式容器	115
5.6.3 算法 vs. 成员函数	116
5.7 使用者自定义之泛型函数 (User-Defined Generic Functions)	117
5.8 以函数作为算法的参数	119
5.8.1 “以函数作为算法的参数” 实例示范	119
5.8.2 判断式 (Predicates)	121
5.9 仿函数 (Functors, Function Objects)	124
5.9.1 什么是仿函数	124
5.9.2 预先定义的仿函数	131
5.10 容器内的元素	134
5.10.1 容器元素的条件	134
5.10.2 Value 语意 vs. Reference 语意	135
5.11 STL 内部的错误处理和异常处理	136
5.11.1 错误处理 (Error Handling)	137
5.11.2 异常处理 (Exception Handling)	139
5.12 扩展 STL	141
6 STL 容器 (STL Container)	143
6.1 容器的共通能力和共通操作	144
6.1.1 容器的共通能力	144
6.1.2 容器的共通操作	144
6.2 Vectors	148
6.2.1 Vectors 的能力	148
6.2.2 Vector 的操作函数	150

6.2.3 将 Vectors 当做一般 Arrays 使用	155
6.2.4 异常处理 (Exception Handling)	155
6.2.5 Vectors 运用实例	156
6.2.6 Class <code>vector<bool></code>	158
6.3 Deques	160
6.3.1 Deques 的能力	161
6.3.2 Deques 的操作函数	162
6.3.3 异常处理 (Exception Handling)	164
6.3.4 Deques 运用实例	164
6.4 Lists	166
6.4.1 Lists 的能力	166
6.4.2 Lists 的操作函数	167
6.4.3 异常处理 (Exception Handling)	172
6.4.4 Lists 运用实例	172
6.5 Sets 和 Multisets	175
6.5.1 Sets 和 Multisets 的能力	176
6.5.2 Sets 和 Multisets 的操作函数	177
6.5.3 异常处理 (Exception Handling)	185
6.5.4 Sets 和 Multisets 运用实例	186
6.5.5 执行期指定排序准则	191
6.6 Maps 和 Multimaps	194
6.6.1 Maps 和 Multimaps 的能力	195
6.6.2 Maps 和 Multimaps 的操作函数	196
6.6.3 将 Maps 视为关联式数组 (Associated Arrays)	205
6.6.4 异常处理 (Exception Handling)	207
6.6.5 Maps 和 Multimaps 运用实例	207
6.6.6 综合实例: 运用 Maps, Strings 并于执行期指定排序准则	213
6.7 其它 STL 容器	217
6.7.1 Strings 可被视为一种 STL 容器	217
6.7.2 Arrays 可被视为一种 STL 容器	218
6.7.3 Hash Tables	221
6.8 动手实现 Reference 语义	222
6.9 各种容器的运用时机	226
6.10 细说容器内的型别和成员	230
6.10.1 容器内的型别	230

6.10.2	生成 (Create)、复制 (Copy)、销毁 (Destroy)	231
6.10.3	非变动性操作 (Nonmodifying Operations)	233
6.10.4	赋值 (Assignments)	236
6.10.5	直接元素存取	237
6.10.6	“可返回迭代器”的各项操作	239
6.10.7	元素的安插 (Inserting) 和移除 (Removing)	240
6.10.8	Lists 的特殊成员函数	244
6.10.9	对配置器 (Allocator) 的支持	246
6.10.10	综观 STL 容器的异常处理	248
7	STL 迭代器 (STL Iterators)	251
7.1	迭代器头文件	251
7.2	迭代器类型 (Iterator Categories)	251
7.2.1	Input (输入) 迭代器	252
7.2.2	Output (输出) 迭代器	253
7.2.3	Forward (前向) 迭代器	254
7.2.4	Bidirectional (双向) 迭代器	255
7.2.5	Random Access (随机存取) 迭代器	255
7.2.6	Vector 迭代器的递增 (Increment) 和递减 (Decrement)	258
7.3	迭代器相关辅助函数	259
7.3.1	advance() 可令迭代器前进	259
7.3.2	distance() 可处理迭代器之间的距离	261
7.3.3	iter_swap() 可交换两个迭代器所指内容	263
7.4	迭代器配接器 (Iterator Adapters)	264
7.4.1	Reverse (逆向) 迭代器	264
7.4.2	Insert (安插型) 迭代器	271
7.4.3	Stream (流) 迭代器	277
7.5	迭代器特性 (Iterator Traits)	283
7.5.1	为迭代器编写泛型函数	285
7.5.2	使用者自定义 (User-Defined) 的迭代器	288
8	STL 仿函数 (functors) (又名函数对象, function objects)	293
8.1	仿函数 (functor) 的概念	293
8.1.1	仿函数可当做排序准则 (Sort Criteria)	294
8.1.2	拥有内部状态 (Internal State) 的仿函数	296
8.1.3	for_each() 的回返值	300
8.1.4	判断式 (Predicates) 和仿函数 (Functors)	302

8.2 预定义的仿函数	305
8.2.1 函数配接器 (Function Adapters)	306
8.2.2 针对成员函数而设计的函数配接器	307
8.2.3 针对一般函数 (非成员函数) 而设计的函数配接器	309
8.2.4 让自定义仿函数也可以使用函数配接器	310
8.3 辅助用 (组合型) 仿函数	313
8.3.1 一元组合函数配接器 (Unary Compose Function Object Adapters)	314
8.3.2 二元组合函数配接器 (Binary Compose Function Object Adapters)	318
9 STL 算法 (STL Algorithms)	321
9.1 算法头文件 (header files)	321
9.2 算法概览	322
9.2.1 简介	322
9.2.2 算法分门别类	323
9.3 辅助函数	332
9.4 <code>for_each()</code> 算法	334
9.5 非变动性算法 (Nonmodifying Algorithms)	338
9.5.1 元素计数	338
9.5.2 最小值和最大值	339
9.5.3 搜寻元素	341
9.5.4 区间的比较	356
9.6 变动性算法 (Modifying Algorithms)	363
9.6.1 复制 (Copying) 元素	363
9.6.2 转换 (Transforming) 和结合 (Combining) 元素	366
9.6.3 互换 (Swapping) 元素内容	370
9.6.4 赋予 (Assigning) 新值	372
9.6.5 替换 (Replacing) 元素	375
9.7 移除性算法 (Removing Algorithms)	378
9.7.1 移除某些特定元素	378
9.7.2 移除重复元素	381
9.8 变序性算法 (Mutating Algorithms)	386
9.8.1 逆转 (Reversing) 元素次序	386
9.8.2 旋转 (Rotating) 元素次序	388
9.8.3 排列 (Permuting) 元素	391
9.8.4 重排元素 (Shuffling, 搅乱次序)	393
9.8.5 将元素向前搬移	395

9.9 排序算法 (Sorting Algorithms)	397
9.9.1 对所有元素排序	397
9.9.2 局部排序 (Partial Sorting)	400
9.9.3 根据第 n 个元素排序	404
9.9.4 Heap 算法	406
9.10 已序区间算法 (Sorted Range Algorithms)	409
9.10.1 搜寻元素 (Searching)	410
9.10.2 合并元素 (Merging)	416
9.11 数值算法 (Numeric Algorithms)	425
9.11.1 加工运算后产生结果	425
9.11.2 相对值和绝对值之间的转换	429
10 特殊容器 (Special Containers)	435
10.1 Stacks (堆栈)	435
10.1.1 核心接口	436
10.1.2 Stacks 运用实例	437
10.1.3 Class stack<> 细部讨论	438
10.1.4 一个使用者自定义的 Stack Class	441
10.2 Queues (队列)	444
10.2.1 核心接口	445
10.2.2 Queues 运用实例	446
10.2.3 Class queue<> 细部讨论	447
10.2.4 一个使用者自定义的 Queue Class	450
10.3 Priority Queues (优先队列)	453
10.3.1 核心接口	455
10.3.2 Priority Queues 运用实例	455
10.3.3 Class priority_queue<> 细部讨论	456
10.4 Bitsets	460
10.4.1 Bitsets 运用实例	460
10.4.2 Class bitset 细部讨论	463
11 Strings (字符串)	471
11.1 动机	471
11.1.1 例一: 引出一个临时文件名	472
11.1.2 例二: 引出一段文字并逆向打印	476
11.2 String Classes 细部描述	479
11.2.1 String 的各种相关型别	479

11.2.2	操作函数 (Operations) 综览	481
11.2.3	构造函数和析构函数 (Constructors and Destructors)	483
11.2.4	Strings 和 C-Strings	484
11.2.5	大小 (Size) 和容量 (Capacity)	485
11.2.6	元素存取 (Element Access)	487
11.2.7	比较 (Comparisons)	488
11.2.8	更改内容 (Modifiers)	489
11.2.9	子串和字符串接合 (concatenation)	492
11.2.10	I/O 操作符	492
11.2.11	搜索和查找 (Searching and Finding)	493
11.2.12	数值 npos 的意义	495
11.2.13	Strings 对迭代器的支持	497
11.2.14	国际化 (Internationalization)	503
11.2.15	效率 (Performance)	506
11.2.16	Strings 和 Vectors	506
11.3	细说 String Class	507
11.3.1	内部的型别定义和静态值	507
11.3.2	生成 (Create)、拷贝 (Copy)、销毁 (Destroy)	508
11.3.3	大小 (Size) 和容量 (Capacity)	510
11.3.4	比较 (Comparisons)	511
11.3.5	字符存取 (Character Access)	512
11.3.6	产生 C-String 和字符数组 (Character Arrays)	513
11.3.7	更改内容	514
11.3.8	搜寻 (Searching and Finding)	520
11.3.9	子字符串及字符串接合 (String Concatenation)	524
11.3.10	I/O 函数	524
11.3.11	产生迭代器	525
11.3.12	对配置器 (allocator) 的支持	526
12	数值 (Numerics)	529
12.1	复数 (Complex Numbers)	529
12.1.1	Class Complex 运用实例	530
12.1.2	复数的各种操作	533
12.1.3	Class complex<> 细部讨论	541
12.2	Valarrays	547
12.2.1	认识 Valarrays	547

12.2.2	Valarray 的子集 (Subsets)	553
12.2.3	Class valarray 细部讨论	569
12.2.4	Valarray 子集类别 (Subset Classes) 细部讨论	575
12.3	全局性的数值函数	581
13	以 Stream Classes 完成输入和输出	583
13.1	IOStreams 基本概念	584
13.1.1	Stream 对象	584
13.1.2	Stream 类别	584
13.1.3	全局性的 Stream 对象	585
13.1.4	Stream 操作符	586
13.1.5	操控器 (Manipulators)	586
13.1.6	一个简单的例子	587
13.2	基本的 Stream 类别和 Stream 对象	588
13.2.1	相关类别及其阶层体系	588
13.2.2	全局性的 Stream 对象	591
13.2.3	头文件 (Headers)	592
13.3	标准的 Stream 操作符 << 和 >>	593
13.3.1	output 操作符 <<	593
13.3.2	input 操作符 >>	594
13.3.3	特殊型别的 I/O	595
13.4	Streams 的状态 (state)	597
13.4.1	用来表示 Streams 状态的一些常数	597
13.4.2	用来处理 Streams 状态的一些成员函数	598
13.4.3	Stream 状态与布尔条件测试	600
13.4.4	Stream 的状态和异常	602
13.5	标准 I/O 函数	607
13.5.1	输入用的成员函数	607
13.5.2	输出用的成员函数	610
13.5.3	运用实例	611
13.6	操控器 (Manipulators)	612
13.6.1	操控器如何运作	612
13.6.2	使用者自定义操控器	614
13.7	格式化 (Formatting)	615
13.7.1	格式标志 (Format Flags)	615
13.7.2	布尔值 (Boolean Values) 的 I/O 格式	617

13.7.3	字段宽度、充填字符、位置调整	618
13.7.4	正记号与大写字符	620
13.7.5	数值进制 (Numeric Base)	621
13.7.6	浮点数 (Floating-Point) 表示法	623
13.7.7	一般性的格式定义	625
13.8	国际化 (Internationalization)	625
13.9	文件存取 (File Access)	627
13.9.1	文件标志 (File Flags)	631
13.9.2	随机存取	634
13.9.3	使用文件描述符 (File Descriptors)	637
13.10	连接 Input Streams 和 Output Streams	637
13.10.1	以 tie() 完成“松耦合” (Loose Coupling)	637
13.10.2	以 stream 缓冲区完成“紧耦合” (Tight Coupling)	638
13.10.3	将标准 Streams 重新定向 (Redirecting)	641
13.10.4	用于读写的 Streams	643
13.11	String Stream Classes	645
13.11.1	String Stream Classes	645
13.11.2	char* Stream Classes	649
13.12	“使用者自定义型别”之 I/O 操作符	652
13.12.1	实作一个 output 操作符	652
13.12.2	实作一个 input 操作符	654
13.12.3	以辅助函数完成 I/O	656
13.12.4	以非格式化函数完成使用者自定义的操作符	658
13.12.5	使用者自定义的格式标志 (Format Flags)	659
13.12.6	使用者自定义之 I/O 操作符的数个依循惯例	662
13.13	Stream Buffer Classes	663
13.13.1	从使用者的角度看 Stream 缓冲区	663
13.13.2	Stream 缓冲区迭代器 (Buffer Iterators)	665
13.13.3	使用者自定义的 Stream 缓冲区	668
13.14	关于性能 (Performance)	681
13.14.1	与 C 标准串流 (Standard Streams) 同步	682
13.14.2	Stream 缓冲区内的缓冲机制	682
13.14.3	直接使用 Stream 缓冲区	683

14 国际化 (Internationalization, i18n)	685
14.1 不同的字符编码 (Character Encodings)	686
14.1.1 宽字符 (Wide-Character) 和多字节 (Multibyte) 文本	686
14.1.2 字符特性 (Character Traits)	687
14.1.3 特殊字符国际化	691
14.2 Locales 的概念	692
14.2.1 运用 Locales	693
14.2.2 Locale Facets	698
14.3 Locales 细部讨论	700
14.4 Facets 细部讨论	704
14.4.1 数值格式化	705
14.4.2 时间和日期的格式化	708
14.4.3 货币符号格式化	711
14.4.4 字符的分类和转换	715
14.4.5 字符串校勘 (String Collation)	724
14.4.6 信息国际化	725
15 空间配置器 (Allocator)	727
15.1 应用程序开发者如何使用配置器	727
15.2 程序库开发者如何使用配置器	728
15.3 C++ 标准程序库的缺省配置器	732
15.4 使用者自行定义的配置器	735
15.5 配置器细部讨论	737
15.5.1 型别定义	737
15.5.2 各项操作	739
15.6 “未初始化内存”之处理工具细部讨论	740
 网络上的资源 (Internet Resources)	 743
参考书目 (Bibliography)	745
索引 (Index)	747

前言

Preface

一开始，我只不过想写一本篇幅不大的有关于 C++ 标准程序库的德文书（也就 400 多页吧）。萌生这个想法是在 1993 年。而在 1999 年的今天，您看到了这个想法的成果：一本英文书，厚达 800 多页，其中包含大量的文字描述、图片和范例。我的目标是，详尽讲解 C++ 标准程序库，使你的所有（或几乎所有）编程问题都能够在你遇到之前就先给你解答。然而，请注意，这不是一种完整描述 C++ 标准程序库的所有方面的书籍，我通过“在 C++ 中利用标准程序库进行学习和程序编写”的形式，表现出最重要的主题。

每一个主题都是以一般性概念为基础而开展，然后导入日常程序编写工作所必须了解的具体细节。为了帮助你理解这些概念和细节，书中提供详尽的范例程序。

这就是我的前言——言简意赅！撰写此书的过程中，我得到了很多乐趣，希望你阅读本书时，能够像我一样快乐。请享用！

致 谢

Acknowledgments

这本书表达的观点、概念、解决方案和范例，来源十分广泛。从这个意义上讲，封面只列我一个人的名字，未免不公平。所以我愿在此向过去数年来帮助和支持我的人和公司，表示诚挚的谢意。

我第一个要感谢的是 Dietmar Kühl。Dietmar 是一位 C++ 专家，尤其精通 I/O streams（输入输出流）和国际化（他曾经仅仅为了好玩而写了一个 I/O stream library）。他不仅将本书的大部分从德文译为英文，还亲自动笔，发挥专长，为本书撰写了数节内容。除此之外，在过去的数年里，他还向我提供了很多宝贵的反馈意见。

其次，我要感谢所有审阅者和那些向我表达过意见的人。他们的努力使本书的品质获得巨大提升。由于名单太长，以下如有任何疏漏，还请见谅。英文版的检阅者包括：Chuck Allison, Greg Comeau, James A. Crotinger, Gabriel Dos Reis, Alan Ezust, Nathan Myers, Werner Mossner, Todd Veldhuizen, Chichiang Wan, Judy Ward, Thomas Wike-hult。德文版的检阅者包括：Ralf Boecker, Dirk Herrmann, Dietmar Kühl, Edda Lörke, Herbert Scheubner, Dominik Strasser, Martin Weitzel。其它人包括：Matt Austern, Valentin Bonnard, Greg Colvin, Beman Dawes, Bill Gibbons, Lois Goldthwaite, Andrew Koenig, Steve Rumsby, Bjarne Stroustrup 和 David Vandevoorde。

我要特别感谢 Dave Abrahams, Janet Cocker, Catherine Ohala 和 Maureen Willard，他们对全书进行了非常细致的审阅和编辑。他们的反馈意见让本书的品质获得了难以置信的提升。

我也要特别感谢我的“活字典”Herb Sutter，他是著名的“Guru of the Week”的创始人，这是一个常设的 C++ 难题讲解专栏，在 `comp.std.c++.moderated` 上发表。

我还要感谢一些公司和个人，他们的帮助使我有机会在各个不同的平台上，使用各种不同的编译器来测试自己的范例程序。非常感谢来自 EDG 的 Steve Adamczyk, Mike Anderson 和 John Spicer，他们的编译器真是太棒了，在 C++ 标

准化过程和本书写作过程中，提供了巨大的支持。感谢 P. J. Plauger 和 Dinkumware, Ltd, 他们很早以来就持续进行与 C++ 标准规格兼容的 C++ 标准程序库开发工作。感谢 Andreas Hommel 和 Metrowerks, 他们完成了深具价值的 CodeWarrior 程序开发环境。感谢 GNU 和 egcs 编译器的所有开发者。感谢 Microsoft, 他们完成了深具价值的 Visual C++。感谢 Siemens Nixdorf Informations Systems AG 的 Roland Hartinger, 他提供了一份他们的 C++ 编译器测试版本。感谢 Topjects GmbH, 为了他那一深具价值的 ObjectSpace library 实作品。

感谢 Addison Wesley Longman 公司里头与我共同工作过的每一个人, 包括 Janet Cocker, Mike Hendrickson, Debbie Lafferty, Marina Lang, Chanda Leary, Catherine Ohala, Marty Rabinowitz, Susanne Spitzer, 和 Maureen Willard 等等。这项工作真是太有趣了。

此外, 我还要感谢 BREDEX GmbH 的人们, 感谢 C++ 社群中的所有人, 特别是那些参与标准化过程的人, 感谢他们的支持和耐心 (有时候我问的问题确实挺傻)。

最后, 也是最重要的, 我要将我的感谢 (附上一个亲吻) 送给我的家人: Ulli, Lucas, Anica 和 Frederic。为了这本书, 我很长时间没有好好陪他们了。

但愿各位能从这本书获得乐趣, 另外, 请保持宽厚。

(译注: 上句原文为 Have fun and be human!。抱歉, 译者对其精确含义并无十足把握)

1

关于本书

1.1 缘 起

C++ 问世后不久，就成为面向对象程序设计领域的实质标准（*de facto standard*），因此，正式标准化的呼声也就浮上了台面。一旦有了一个可以依循的标准规格，我们便可能写出跨越 PC，乃至大型主机等各种不同平台的程序。此外，如果能够建立起一个标准程序库，程序员便得以运用可移植的通用组件（*general components*）和更高层次的抽象性，而不必从头创造世界。

C++ 标准化过程始于 1989 年，由国际性的 ANSI/ISO 委员会负责。标准化工作以 Bjarne Stroustrup 的两本书 *The C++ Programming Language* 和 *The Annotated C++ Reference Manual* 为根基。这份标准规格于 1997 年通过后，又在数个国家进行了一些正式程序，最后于 1998 年成为国际性的 ISO/ANSI 标准。标准化过程本身包含一个任务：建立 C++ 标准程序库。作为核心语言的拓展，标准程序库提供了一些通用组件。通过大量运用 C++ 新的抽象能力和泛型（*generic types*）特性，标准程序库提供了一系列共同的类别和接口。程序员藉此获得了更高层次的抽象能力。这个标准程序库提供了以下组件：

- String 型别
- 各种数据结构（例如 *dynamic array*、*linked lists*、*binary trees*）
- 各种算法（例如各种排序算法）
- 数值类别（*numeric classes*）
- 输入/输出（*I/O*）类别
- 国际化支持（*internationalization support*）类别

所有这些组件的接口都十分简单。这些组件在很多程序中都很重要。如今的数据处理工作，通常就意味着输入、计算、处理和输出大量数据（通常是字符串）。

这个标准程序库的用法并非不言自明。要想从其强大能力中受惠，你需要一本好书；不能够仅仅列出每一个类别和其函数，而是必须详细解释各组件的概念和重

要细节。本书正是以此为目标。本书首先从概念上介绍标准程序库及其所有组件，然后描述实际编程（programming）所需了解的细节。为了展示组件的确切用法，书中还包括了大量实例。因此，这本书不论对初学者或是编程老手，都是极为详尽的 C++ 标准程序库文件。以本书所提供的数据来武装自己的头脑，你就能从 C++ 标准程序库中获得最大助益。

注意，我不担保本书所有内容都易于理解。标准程序库非常灵活，但这种非同寻常的灵活性是有代价的。标准程序库中有一些陷阱和缺陷，你必须小心应对。碰到它们时我会为你指出，并提出一些建议，帮助你回避问题。

1.2 阅读前的必要基础

要想读懂本书的大部分内容，你需要先了解 C++。本书讲述 C++ 标准组件，而不是语言本身。你应该熟悉类别（classes）、继承（inheritance）、模板（templates）和异常处理（exception handling）的概念，但不必熟知语言的每一个细节。某些重要的细节本书也会讲解（至于次要细节对程序库实现者可能很重要，对程序库使用者就不那么重要了）。注意，在标准化过程中，C++ 语言发生了很大的变化，也许你的知识已经过时了。2.2 节简单介绍了一些“使用标准程序库时，你需要了解的最新语言特性”。如果你不确定自己是否了解 C++ 的新特性（例如关键词 `typename` 以及 `namespace` 概念），请先阅读该节。

1.3 本书风格与结构

标准程序库内的组件有相当程度的独立性，但彼此又存在关联，所以很难在描述某一部分时全然不提及其它部分。我为这本书考虑了几种不同的组织方式。一是按照 C++ *standard* 的次序，但这并非完整介绍 C++ 标准程序库的最佳选择。另一种方式是，首先纵览所有组件，再逐章详细介绍。第三个方式则是，由我依照“组件之交叉参考”程度的高低，从最低者开始介绍，逐步介绍最复杂的组件。最终，我综合了三种方式：首先简短介绍标准程序库所涉及的总体概念和工具，然后分章详述各个组件，每个组件一章或数章。首当其冲的便是 STL（Standard Template Library，标准模板库）。STL 无疑是标准程序库中最强大、最复杂、最激动人心的部分，其设计深刻影响了其它组件。接下来我再讲解较易理解的组件，例如特殊容器、strings 和数值类别。再就是你或许已经使用多时的老朋友：IOStream 程序库。最后是国际化议题的讨论，这一部分对于 IOStream 程序库有些影响。

讲述每个组件时，我首先给出该组件的目的、设计和范例，然后通过各种使用方法和注意事项的描述，讲解组件的细节。最后是一个参考章节，你可以在其中找到组件类别和其函数的确切标记形式（exact signature）。

以下是本书内容。最前面的四章总体介绍了本书及 C++ 标准程序库：

- 第 1 章：关于本书

这一章（也就是你此刻正阅读的）介绍本书的主题和内容。

- 第 2 章：C++ 及其标准程序库简介

这一章对于 C++ 标准程序库的历史和背景进行简短综览。此章也包括了本书及标准程序库的技术背景的大致介绍，例如新的语言特性和所谓复杂度的概念。

- 第 3 章：一般概念（General Concepts）

本章描述标准程序库的基本概念，这些概念对于你理解和使用程序库中的所有组件都是必须的。明确地说，本章介绍了 `namespace std`、头文件（headers）格式、错误和异常处理（exception handling）的一般性支持。

- 第 4 章：通用工具（Utilities）

本章描述了数种提供给程序库用户和程序库本身运用的小工具，更明确地说是 `max()`、`min()`、`swap()` 等辅助函数，以及 `pair`、`auto_ptr`、`numeric_limits` 等型别。上述最后一个型别提供了与由实作决定的数值型别相关信息。

第 5 章至第 9 章分别从各个方面描述 STL。

- 第 5 章：Standard Template Library（STL，标准模板库）

STL 提供了用于处理数据的容器和算法。本章详细介绍 STL 的概念，并逐步解释其中的概念、问题、特殊编程技术，以及它们所扮演的角色。

- 第 6 章：STL 容器（Containers）

本章解释 STL 容器类别的概念和能力。首先透过详尽的例子，分别讲解 `vectors`、`deques`、`lists`、`sets` 和 `maps`，然后介绍它们的共同能力。最后以简明的形式列出并描述所有容器所提供的函数，作为一份方便的参考数据。

- 第 7 章：STL 迭代器（Iterators）

本章具体介绍了 STL 迭代器。解释迭代器的分类和辅助函数，以及相应的配接器（iterator adapter），如 `stream iterators`、`reverse iterators` 和 `insert iterators`。

- 第 8 章：STL 仿函数（Functors，又名 Function Objects）

本章详细讲解 STL 仿函数。

- 第9章：STL 算法 (Algorithms)

本章罗列并描述 STL 算法。在简单介绍和比较这些算法后，通过一个（或多个）范例，对每个算法进行详细描述。

第10章至12章描述了一些“简单的”标准类别：

- 第10章：特殊容器 (Special Containers)

本章描述 C++ 标准程序库的各种特殊容器，并涵盖容器配接器 (container adapters) `queues`, `stacks`，以及 `class bitset`，后者可管理任意数量的 bits 或 flags。

- 第11章：Strings (字符串)

本章描述 C++ 标准程序库的 `string` 型别（不止一种哦）。C++ *standard* 将 strings 设计为一种“能够处理不同字符类型”的基本数据类型，而且简明易用。

- 第12章：数值 (Numerics)

本章描述 C++ 标准程序库中的数值组件，包括复数 (`complex`)，以及一些用来处理数值数组的类别（可用于矩阵 `matrices`、向量 `vectors` 和方程式 `equations`）。

第13章和第14章的主题是 I/O 和国际化（两者紧密相关）。

- 第13章：以 Stream Classes 完成输入和输出

本章涵盖 C++ 的 I/O 组件。该组件是众所周知的 `IOStream` 程序库的标准化形式。本章也讲述了一些对程序员而言可能很重要，但又鲜为人知的细节。例如，如何定义及整合特殊 I/O 通道，这是一个在实务过程中经常被搞错的题目。

- 第14章：国际化 (Internationalization, `il8n`)

本章涵盖“将程序国际化”的概念和类别，包括对不同字符集 (`character sets`) 的处理，以及如何使用不同格式的浮点数和日期。

剩余部分包括以下内容。

- 第15章：空间配置器 (Allocators)

本章描述 C++ 标准程序库中内存模型 (`memory model`) 的概念。

- 附录，包括：

- 因特网上的资源 (Internet Resources)
- 参考书目 (Bibliography)
- 索引 (Index)

1.4 如何阅读本书

本书既是介绍性的使用指南, 又是 C++ 标准程序库的结构化参考手册。C++ 标准程序库的各个组件在相当程度上是彼此独立的, 所以读完第 2 章至第 4 章后, 你可以按任意顺序阅读其它各章节。不过切记, 第 5 章至第 9 章讲述的是同一组东西。要理解 STL, 应该从介绍性的第 5 章开始。

如果你是一位想总体认识 C++ 标准程序库概念和其各个方面的程序员, 可以简略浏览这本书, 并略过其参考章节 (译注: 就是完整列出各个接口的那些小节)。当你需要运用某个 C++ 标准程序库组件时, 最好的办法就是通过索引 (index), 找出相关数据。我已尽力把索引做得详尽, 希望能够节省你的搜寻时间。

以我的经验来看, 学习新东西的最佳方式就是阅读范例。因此, 你会发现本书通篇有大量的范例, 可能是几行代码, 也可能是完整程序。如果是后者, 则批注的第一行列有其文件名称。你可以在我的网站 <http://www.josuttis.com/libbook/> 找到这些文件。

1.5 目前发展形势

C++ *standard* 在我撰写本书期间完成。请记住, 有些编译器可能还无法与之兼容。这一点很可能在不久的将来得到很大的改善。但是现在, 你可能会发现, 本书所谈的东西并非一定能够在你的系统上有相同的表现, 或许得稍作修改后才能在你的环境中正常运行。我所使用的 EGCS 编译器 2.8 版, 及其更高版本, 能够编译书中所有范例程序。该编译器几乎适用于所有计算机平台, 你可以从因特网 (<http://egcs.cygnum.com/>) 和某些软件光盘中获得。

1.6 范例程序代码及额外信息

在我的网站 <http://www.josuttis.com/libbook/> 上, 你可以获得所有范例程序代码、本书及 C++ 标准程序库的其它相关信息。你也可以在因特网上找到许多其它相关信息, 详见 p743。

1.7 响应

欢迎你对本书提出意见 (不论肯定还是批评)。我已尽力而为, 但我毕竟是凡人, 而且总有结稿的时候, 所以难免有一些错误或前后不一的地方, 需要进一步改善。你的响应将使我的新版本有机会改进。与我联系的最佳方式是通过电子邮件:

libbook@josuttis.com.

也可以通过电话、传真或“蜗牛般的”信件与我联系:

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

非常感谢!

2

C++ 及其标准程序库简介

2.1 沿革

C++ 的标准化过程始于 1989 年，于 1997 年底完成。不过，由于某些原因，最终的标准规格迟至 1998 年 9 月才公布，整个努力的成果就是国际标准化组织 (ISO, International Organization for Standardization) 发布的一份长达 750 页的标准规格参考手册。这份标准被命名为 “Information Technology — Programming Language — C++”，文件编号为 ISO/IEC 14882-1998，由 ISO 的各国成员机构发布，例如美国的 ANSI 机构¹。

标准规格的建立，是 C++ 的一个重要里程碑，它准确定义了 C++ 的内容和行为，简化了 C++ 的教学、使用，以及在不同平台之间的移植工作，给予用户选择不同 C++ 实作品（编译器）的自由。它的稳定性和可移植性，对于程序库、工具库的供应者和实现者都是一个福音。因此，这份标准规格能帮助 C++ 应用程序开发人员更快更好地创建应用程序，减少维护上的精力。

标准程序库是 C++ 标准规格的一部分，提供一系列核心组件，用以支持 I/O、字符串 (strings)、容器 (数据结构)、算法 (排序、搜索、合并等等)、数值计算、国别 (例如不同的字符集, character sets) 等主题。

这个标准化过程竟花费了近 10 年的时间，未免让人奇怪。如果你了解其中一些细节，更会奇怪为什么这么久之后这一标准仍然未臻完美。十年其实不够！尽管从标准化的历史和内容来看，确实完成了许多东西，也形成了可在实务中应用成果，但是距离完美尚远（毕竟世间无完美之物）。

¹ 花 18 美元可以从 ANSI Electronics Standard Store 获得这份 C++ 标准文件（见 <http://www.ansi.org/>）。

这份标准并非某个公司花费大把钞票和大量时间后的产物。那些为制定标准而辛勤工作的人们，几乎没有从标准化组织那儿获取任何报酬。对他们而言，如果他们所处的公司对 C++ 标准化漠不关心，那么他们就只能以兴趣为全部的动力了。谢天谢地，有这么多勇于奉献的人，能够拿出时间和财力参与其中。

这份 C++ 标准规格并非从零开始，它以 C++ 创始人 Bjarne Stroustrup 对于这个语言所作的描述为基础。然而标准程序库并非基于某本书或某一个现成的函数库，而是将各种不同的类别（classes）整合而成²，因此，其结果并非十分地同质同种。你会发现不同组件的背后有着不同的设计原则，string class 和 STL 之间的差别就是很好的例子，后者是一个数据结构和算法框架（framework）。

- string classes 被设计为安全易用的组件，其接口几乎不言自明，并能对许多可能的错误作检验。
- STL 的设计目标，是将不同的算法和数据结构结合在一起，并获取最佳效率，所以 STL 并不非常便利，也不检验许多可能的逻辑错误。要运用 STL 强大的框架和优异的效率，你就必须通晓其概念并小心运用。

标准程序库中有一个组件，在标准化之前就已经作为“准”标准而存在，那就是 IOSTream 程序库。这个东西开发于 1984 年，1989 年重做过一次，部分内容重新设计。由于很多程序员早就已经使用 IOSTream，所以 IOSTream 程序库的概念没有改变，保持向下兼容。

总体来说，整份标准规格（语言和程序库）是在来自全球各地数百位人士的大量讨论和影响下诞生的，例如日本人就是国际化（internationalization）组件的重要支持者。当然，我们曾经犯下错误，曾经改弦更张，曾经意见不一。到了 1994 年，当人们认为这个标准接近完成时，STL 又被加了进来，此举从根本上改变了整个程序库。然而事情总要有个结束，终于有那么一天，人们决定不再考虑任何重大扩张，无论这个扩张多么有价值。就因为这样，hash tables 没有被纳入标准——尽管它作为一种常用的数据结构，理应在 STL 中享有一席之地。

现有的标准并不是终极产品，每五年会有一个新版本，修正错误和不一致的地方。然而，起码在接下来的数年中，C++ 程序员终于有了一个标准，有机会编写功能强大并可移植到各种平台上的 C++ 程序了。

² 你可能会奇怪，为什么标准化过程中不从头设计一个新的程序库。要知道，标准化的主要目的不是为了发明新东西或发展出某些东西，而是为了让已有的东西协调共处。

2.2 新的语言特性

C++ 语言核心和 C++ 程序库是同时被标准化的，这么一来，程序库可以从语言的发展中受益，语言的发展也可以从程序库的实践经验中得到启发。事实上，在标准化过程中，程序库经常用到已规划但当时尚未被实现出来的语言特性。

今天的 C++ 可不是五年前的 C++ 了。如果你没有紧跟其发展，可能会对程序库所使用的语言新特性大感惊讶。本节将对这些新特性进行简单的概括说明，至于细节，请查阅语言相关书籍。

我撰写本书的时候（1998 年），并非所有编译器都提供所有的语言新特性，这就限制了程序库的使用。我希望（并预期）这种情况能很快获得改善（绝大多数编译器厂商都参与了标准化过程）。实作一份可移植的程序库时，通常都要考虑你的计算环境是否支持你所用到的特性——通常会使用一些测试程序，检查哪些语言特性获得支持，然后根据检验结果设置预处理器指令（preprocessor directives）。我会在书中以脚注方式指出所有典型而重要的限制。

接下来数个小节将描述与 C++ 标准程序库有关的几个最重要的语言新特性。

2.2.1 template（模板）

程序库中几乎所有东西都被设计为 template 形式。不支持 template，就不能使用标准程序库。此外程序库还需要一些新的、特殊的 template 特性，我将在简介之后详细说明。

所谓 templates，是针对“一个或多个尚未明确的型别”所撰写的函数或类别。使用 template 时，可以显式地（explicitly）或隐式地（implicitly）将型别当做参数来传递。下面是一个典型例子，传回两数之中的较大数：

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

在这里，第一行将 `T` 定义为任意数据类型别，于函数被调用时由调用者指定。任何合法的识别符号都可以拿来作为参数名称，但通常以 `T` 表示，这差不多成了一个“准”标准。这个型别由关键字 `class` 引导，但型别本身不一定得是 `class`——任何数据类型别只要提供 template 定义式内所用到的操作，都可适用于此 template³。

³ 这里使用关键词 `class`，原是为了避免增加新的关键词。然而最终还是不得不引入一个新的关键词 `typename`。此处亦可使用关键词 `typename`（详见 p11）。

遵循同样原则，你可以将 `class` 参数化，并以任意型别作为实际参数。这一点对容器类别非常有用。你可以实作出“有能力操控任意型别元素”的容器。C++ 标准程序库提供了许多 `template container classes`（详见第6章和第10章）。标准程序库对于 `template classes` 的使用，还有其它原因。例如可以以字符型别（`character type`）和字符集（`character set`）属性，将 `string class` 参数化（详见第11章）。

`Template` 并非一次编译便生出适合所有型别的代码，而是针对被使用的某个（或某组）型别进行编译。这导致一个重要的问题：实际处理 `template` 时，面对 `template function`，你必须先提供它的某个实作品，然后才能调用，如此方可通过编译。所以目前唯一能够让“`template` 的运用”具有可移植性的方式，就是在头文件中以 `inline function` 实现 `template function`⁴。

欲实现 C++ 标准程序库的完整功能，编译器不仅需要提供一般的 `template` 支持，还需要很多新的 `template` 标准特性，以下分别探讨。

Nontype Templates 参数（非型别模板参数）

型别（`type`）可作为 `template` 参数，非型别（`nontype`）也可以作为 `template` 参数。非型别参数因而可被看做是整个 `template` 型别的一部分。例如可以把标准类别 `bitset<>`（本书 10.4 节，p460 会介绍它）的 `bits` 数量以 `template` 参数指定之。以下述句定义了两个由 `bits` 构成的容器，分别为 32 个 `bits` 空间和 50 个 `bits` 空间：

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

这些 `bitsets` 由于使用不同的 `template` 参数，所以有不同的型别，不能互相赋值（`assign`）或比较（除非提供了相应的型别转换机制）。

Default Template Parameters（缺省模板参数）

`Template classes` 可以有缺省参数。例如以下声明，允许你使用一个或两个 `template` 参数来声明 `MyClass` 对象⁵：

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

如果只传给它一个参数，那么缺省参数可作为第二参数使用：

⁴ 目前 `template` 必须定义于头文件中。为了消除这个限制，标准规格导入了一个 `template compilation model`（模板编译模型）和一个关键词 `export`。可惜据我所知，目前尚无任何编译器实现出这一特性。

⁵ 注意，两个 “>” 之间必须有一个空格，如果你没写空格，“>>” 会被解读为移位运算符（`shift operator`），导致语法错误。

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

注意，`template` 缺省参数可根据前一个（或前一些）参数而定义。

关键字 `typename`

关键字 `typename` 被用来做为型别之前的标识符号。考虑下面例子：

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

这里，`typename` 指出 `SubType` 是 `class T` 中定义的一个型别，因此 `ptr` 是一个指向 `T::SubType` 型别的指针。如果没有关键字 `typename`，`SubType` 会被当成一个 `static` 成员，于是：

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

会被解释为型别 `T` 内的数值 `SubType` 与 `ptr` 的乘积。

`SubType` 成为一个型别的条件是，任何一个用来取代 `T` 的型别，其内部都必须提供一个内部型别（inner type）`SubType` 的定义。例如，将型别 `Q` 当做 `template` 参数：

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

必要条件是型别 `Q` 有如下的内部型别定义：

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

此时，`MyClass<Q>` 的 `ptr` 成员应该变成一个指向 `int` 型别的指针。子型别（subtype）也可以成为抽象数据类型（例如 `class`）：

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

注意，如果要把一个 `template` 中的某个标识符号指定为一种型别，就算意图显而易见，关键字 `typename` 也不可或缺，因此 C++ 的一般规则是，除了以 `typename` 修饰之外，`template` 内的任何标识符号都被视为一个值（value）而非一个型别。

`typename` 还可在 `template` 声明中用来替换关键字 `class`：

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

Member Template (成员模板)

class member function 可以是 `template`，但这样的 `member template` 既不能是 `virtual`，也不能有缺省参数。例如：

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

在这里，`MyClass::f` 声明了一个成员函数集，适用任何型别参数。只要某个型别提供有 `f()` 用到的所有操作，它就可以被当做参数传递进去。这个特性通常用来为 `template classes` 中的成员提供自动型别转换。例如以下定义式中，`assign()` 的参数 `x`，其型别必须和调用端所提供的对象的型别完全吻合：

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

即使两个型别之间可以自动转换，如果我们对 `assign()` 使用不同的 `template` 型别，也会出错：

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

如果 C++ 允许我们为 `member function` 提供不同(一个以上)的 `template` 型别，就可以放宽“必须精确吻合”这条规则；只要型别可被赋值 (*assignable*)，就可以被当做上述 `member template function` 的参数。

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

```

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

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

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

```

请注意，现在，`assign()`的参数 `x` 和 `*this` 的型别并不相同，所以你不再能够直接存取 `MyClass<>` 的 `private` 成员和 `protected` 成员，取而代之的是，此例中你必须使用类似 `getValue()` 之类的东西。

Template constructor 是 member template 的一种特殊形式。Template constructor 通常用于“在复制对象时实现隐式型别转换”。注意，template constructor 并不遮蔽 (*hide*) implicit copy constructor。如果型别完全吻合，implicit copy constructor 就会被产生出来并被调用。举个例子：

```

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

void f()
{
    MyClass<double> xd;

```

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

在这里，xd2 和 xd 的型别完全一致，所以它被内建的 copy ctor 初始化。xi 的型别和 xd 不同，所以它使用 template ctor 进行初始化。因此，撰写 template ctor 时，如果 default copy ctor 不符合你的需要，别忘了自己提供一个 copy ctor。member template 的另一个例子详见 p33, 4.1 节。

Nested Template Classes

嵌套的 (*nested*) classes 本身也可以是个 template:

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

2.2.2 基本型别的显式初始化 (Explicit Initialization)

如果采用不含参数的、明确的 constructor (构造函数) 调用语法，基本型别会被初始化为零:

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

这个特性可以确保我们在撰写 template 程序代码时，任何型别都有一个确切的初值。例如下面这个函数中，x 保证被初始化为零。

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

2.2.3 异常处理 (Exception Handling)

通过异常处理, C++ 标准程序库可以在不“污染”函数接口(亦即参数和返回值)的情况下处理异常。如果你遇到一个意外情况, 可以通过“抛出一个异常”来停止一般的(正常的)处理过程:

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

这句 `throw` 开始了 *stack unwinding* (堆栈辗转开解) 过程, 也就是说, 它将使得脱离任何函数区段时的行为就像以 `return` 语句返回一样, 然而程序却不会跳转到任何地点。对于所有被声明于某区段——而该区段却因程序异常而脱离——的局部对象而言, 其 *destructor* (析构函数) 会被调用。*Stack unwinding* 的动作会持续直到退出 `main()` 或直到有某个 `catch` 子句捕捉并处理了该异常为止。如果是第一种情况, 程序会结束:

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

在这里, `try` 区段中任何“型别为 `Error` 的异常”都将在 `catch` 子句获得处理^⑥。

^⑥ 异常(exception)会使对某个函数的调用动作中止, 而该函数正是异常发生之处。异常处理机制有能力将某个对象当做参数, 回传给函数调用者。但这并非回调(call back)函数。所谓回调是指从“问题被发现的地方”到“问题被处理的地方”, 也就是从下往上的方向。以此观之, 异常处理和信号处理(signal handling)完全是两码事。

异常对象 (exception objects) 其实就是一般类别或基本型别的对象, 可以是 ints、strings, 也可以是类体系中的某个 template classes。通常会设计一个特殊的 error 类体系。你可以运用异常对象的状态 (state), 将任何信息从错误被侦测到的地点带往错误被处理的地点。

注意, 这个概念叫作“异常处理 (exception handling)”, 而不是“错误处理 (error handling)”, 两者未必相同。举个例子, 许多时候用户的无效输入 (这经常发生) 并非一种异常; 这时候最好是在区域范围内采用一般的错误处理技术来处理。

你可以运用所谓的异常规格 (exception specification) 来指明某个函数可能抛出哪些异常, 例如:

```
void f() throw(bad_alloc); // f()只可能丢出 bad_alloc 异常
```

如果声明一个空白异常规格, 那就表明该函数不会抛出任何异常:

```
void f() throw(); // f()不抛出任何异常
```

违反异常规格将会导致特殊行为, 详见 p26 关于异常类别 bad_exception 的描述。

C++ 标准程序库提供了一些通用的异常处理特性, 例如标准异常类别 (standard exception classes) 和 auto_ptr 类别 (详见 p25, 3.3 节和 p38, 4.2 节)。

2.2.4 命名空间 (Namespaces)

越来越多的软件由程序库、模块 (modules) 和组件拼凑而成。各种不同事物的结合, 可能导致一场名称大冲突。Namespaces 正是用来解决此一问题的。

Namespaces 将不同的标识符号集合在一个具名作用域 (named scope) 内。如果你在 namespace 之内定义所有标识符号, 则 namespace 本身名称就成了唯一可能与其它全局符号冲突的标识符号。你必须在标识符号前加上 namespace 名字, 才能援引该 namespace 内的符号, 这和 class 处理方式雷同。namespace 的名字和标识符号之间以 :: 分隔开来 (译注: 这个符号及其意义和 class 与其 members 之间的联系有点类似)。

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

不同于 `class` 的是, `namespaces` 是开放的, 你可以在不同模块 (`modules`) 中定义和扩展 `namespace`。因此你可以使用 `namespace` 来定义模块、程序库或组件, 甚至在多个文件之间完成。`Namespace` 定义的是逻辑模块, 而非实质模块。请注意, 在 UML 及其它建模表示法 (`modeling notations`) 中, 模块又被称为 *package*。

如果某个函数的一个或多个参数型别, 乃定义于函数所处的 `namespace` 中, 那么你可以不必为该函数指定 `namespace`。这个规则称为 *Koenig lookup* (译注: Andrew Koenig 是 C++ 社群代表人物之一, 对 C++ 的形成和发展有重要贡献)。例如:

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

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

通过 *using declaration*, 我们可以避免一再写出冗长的 `namespace` 名称。例如以下声明:

```
using josuttis::File; // 译注: 这就是一个 using declaration
// 会使 File 成为当前作用域 (current scope) 内代表 josuttis::File 的一个同义字。
```

using directive 会使 `namespace` 内的所有名字曝光。*using directive* 等于将这些名字声明于 `namespace` 之外。但这么一来, 名称冲突问题就可能死灰复燃。例如:

```
using namespace josuttis; // 译注: 这就是一个 using directive
// 会使 File 和 MyGlobalFunc() 在当前作用域内完全曝光。如果全局范围内已存在同名的 File 或 MyGlobalFunc(), 而且使用者不加任何资格饰词 (qualification) 地使用这两个名字, 编译器将东西难辨。
```

注意, 如果场合 (`context`) 不甚清楚 (例如不清楚究竟是在头文件、模块里还是在程序库里), 你不应该使用 *using directive*。这个指令可能会改变 `namespace` 的作用域, 从而使程序代码被包含或使用于另一模块中, 导致意外行为的发生。事实上在头文件中使用 *using directive* 相当不明智。

C++ 标准程序库在 `namespace std` 中定义了它的所有标识符号, 详见 p23, 3.1 节。

2.2.5 bool 型别

为了支持布尔值（真假值），C++ 增加了 bool 型别。bool 可增加程序的可读性，并允许你对布尔值实现重载（*overloaded*）动作。两个常数 true 和 false 同时亦被引入 C++。此外 C++ 还提供布尔值与整数值之间的自动转换。0 值相当于 false，非 0 值相当于 true。

2.2.6 关键字 explicit

通过关键字 explicit 的作用，我们可以禁止“单参数构造函数（single argument constructor）”被用于自动型别转换。典型的例子便是群集类别（collection classes），你可以将初始长度作为参数传给构造函数，例如你可以声明一个构造函数，以 stack 的初始大小为参数：

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

在这里，explicit 的应用非常重要。如果没有 explicit，这个构造函数有能力将一个 int 自动转型为 Stack。一旦这种情况发生，你甚至可以给 Stack 指派一个整数值而不会引起任何问题：

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

“自动型别转换”动作会把 40 转换为有 40 个元素的 stack，并指派给 s，这几乎肯定不是我们所要的结果。如果我们将构造函数声明为 explicit，上述赋值操作就会导致编译错误（那很好）。

注意，explicit 同样也能阻绝“以赋值语法进行带有转型操作的初始化”：

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

这是因为以下两组操作：

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

和

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

存在一个小差异。前者透过显式转换，根据型别 x 产生了一个型别 Y 的新对象；后者通过隐式转换，产生了一个型别 Y 的新对象。

2.2.7 新的型别转换操作符 (Type Conversion Operators)

为了让你对“参数的显式型别转换”了解更透彻，C++ 引入以下四个新的操作符：

1. static_cast

将一个值以符合逻辑的方式转型。这可看做是“利用原值重建一个临时对象，并在设立初值时使用型别转换”。唯有当上述的型别转换有所定义，整个转换才会成功。所谓的“有所定义”，可以是语言内建规则，也可以是程序员自定的转换动作。例如：

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

2. dynamic_cast

将多态型别 (polymorphic type) 向下转型 (downcast) 为其实际静态型别 (real static type)。这是唯一在执行期进行检验的转型动作。你可以用它来检验某个多态对象的型别，例如：

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

class Cabriolet : public Car {
...
};

class Limousine : public Car {
...
};

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

在这个例子中，面对实际静态型别为 Cabriolet 的对象，f() 有特殊应对行为。当参数是个 reference，而且型别转换失败时，dynamic_cast 会丢出一个 bad_cast 异常 (bad_cast 的描述见 p26)。注意，从设计角度而言，你应该在运用多态技术的程序中，避免这种“程序行为取决于具体型别”的写法。

3. `const_cast`

设定或去除型别的常数性 (constness)，亦可去除 `volatile` 饰词。除此之外不允许任何转换。

4. `reinterpret_cast`

此操作符的行为由实际编译器定义。可能重新解释 bits 意义，但也不一定如此。使用此一转型动作通常带来不可移植性。

这些操作符取代了以往小圆括号所代表的旧式转型，能够清楚阐明转型的目的。小圆括号转型可替换 `dynamic_cast` 之外的其它三种转型，也因此当你运用它时，你无法明确显示使用它的确切理由。这些新式转型操作符给了编译器更多信息，让编译器清楚知道转型的理由，并在转型失败时释出一份错误报告。

注意，这些操作符都只接受一个参数。试看以下例子：

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

在这个例子中你得不到你想要的结果。它只用一个数值 100，将 `Fraction` 暂时对象设定初值，而非设定分子 15、分母 100。逗号在这里并不起分隔作用，而是形成一个 **comma (逗号) 操作符**，将两个表达式组合为一个表达式，并传回第二表达式作为最终结果。将 15 和 100 “转换”为分数的正确做法是：

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

2.2.8 常数静态成员 (Constant Static Members) 的初始化

如今，我们终于能够在 `class` 声明中对“整型 (integral) 常数静态成员”直接赋予初值。初始化后，这个常数便可用于 `class` 之中，例如：

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

注意，你还必须为 `class` 之中声明的常数静态成员，定义一个空间：

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

2.2.9 main() 的定义式

我乐于澄清这个语言中一个重要而又常被误解的问题，那就是正确而可移植的 main() 的唯一写法。根据 C++ 标准规格，只有两种 main() 是可移植的：

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

和

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

这里 argv (命令行参数数组) 也可定义为 char**。请注意，由于不允许“不言而喻”的返回型别 int，所以返回型别必须明白写为 int。你可以使用 return 语句来结束 main()，但不必一定如此。这一点和 C 不同，换句话说，C++ 在 main() 的末尾定义了一个隐式的：

```
return 0;
```

这意味如果你不采用 return 语句离开 main()，实际上就表示成功退出（传回任何一个非零值都代表某种失败）。出于这个原因，本书范例在 main() 尾端都没有 return 语句。有些编译器可能会对此发出警告（译注：例如 Visual C++），有的甚至认为这是错误的。唔，那正是标准制定前的黑暗日子。

2.3 复杂度和 Big-O 表示法

对于 C++ 标准程序库的某些部分（特别是 STL），算法和成员函数的效率需要严肃考虑，因此需要动用“复杂度”的概念。计算器科学家运用特定符号，比较算法的相对复杂度，如此便可以很快依据算法的运行时间加以分类，进行算法之间的定性比较。这种衡量方法叫做 Big-O 表示法。

Big-O 表示法系将一个算法的运行时间以输入量 n 的函数表示。例如，当运行时间随元素个数呈线性增长时（亦即如果元素个数呈倍数增长，运行时间亦呈倍数增长），复杂度为 $O(n)$ ；如果运行时间独立于输入量，复杂度为 $O(1)$ 。表 2.1 列出了典型的复杂度和其 Big-O 表示法。

请注意，Big-O 表示法隐藏了（忽略了）指数较小的因子（例如常数因子），这一点十分重要，更明确地说，它不关心算法到底耗用多长时间。根据这种量测法则，任何两个线性算法都被视为具有相同的接受度。甚至可能发生一种情况：带有巨大常数的线

性算法竟然比带有小常数的指数算法受欢迎（译注：因为 Big-O 复杂度表示法无法显现真实的运算时间）。这是对 Big-O 表示法的一种合理批评。记住，这只是一种量度规则。具有最佳（最低）复杂度的算法，不一定就是最好（最快）的算法。

表 2.1 典型的五种复杂度

型别	表示法	含义
常数	$O(1)$	运行时间与元素个数无关
对数	$O(\log(n))$	运行时间随元素个数的增加呈对数增长
线性	$O(n)$	运行时间随元素个数的增加呈线性增长
$n\text{-}\log\text{-}n$	$O(n*\log(n))$	运行时间随元素个数的增加呈“线性和对数的乘积”增长
二次	$O(n^2)$	运行时间随元素个数的增加呈平方增长

表 2.2 列出了所有复杂度分类，并以某些元素个数来说明运行时间随元素个数增长的程度。--如你所看到的，当元素较少时，运行时间的差别很小，此时 Big-O 表示法所隐藏的常数因子可能会带来很大影响。但是当元素个数愈多，运行时间差别愈大，常数因子也就变得无关紧要了。当你考虑复杂度时，请记住，输入量必须够大才有意义。

表 2.2 运行时间、复杂度、元素个数对照表

复杂度		元素数目						
型别	表示法	1	2	5	10	50	100	1000
常数	$O(1)$	1	1	1	1	1	1	1
对数	$O(\log(n))$	1	2	3	4	6	7	10
线性	$O(n)$	1	2	5	10	50	100	1,000
$n\text{-}\log\text{-}n$	$O(n*\log(n))$	1	4	15	40	300	700	10,000
二次方	$O(n^2)$	1	4	25	100	2,500	10,000	1,000,000

C++ 标准手册中的某些复杂度被称为 *amortized*（分期摊还），意思是，长期而言，大量操作将如上述描述般进行，但单一操作却可能花费比平均值更长的时间。举个例子，如果你为一个 *dynamic array* 追加元素，运行时间将取决于 *array* 是否尚有备用内存。如果内存足够，就属于常数复杂度，因为在尾端加入一个新元素，总是花费相同时间。如果备用内存不足，那么就是线性复杂度，因为你必须配置足够的内存并搬动（复制）它们，实际耗用时间取决于当时的元素个数。内存重新分配动作并不常发生（译注：STL 的 *dynamic array* 容器会以某种哲学来保持备用内存），所以任何“长度充分”的序列（*sequence*），元素附加动作几乎可说是常数复杂度。这种复杂度我们便称为 *amortized*（分期摊还）常数时间。

3

一般概念

General Concepts

本章讲述 C++ 标准程序库中的基本概念。几乎所有 C++ 标准程序库组件都用到这些概念。

- 命名空间 (namespace) std
- 头文件 (headers) 的名称与格式
- 错误 (error) 和异常 (exception) 处理的一般概念
- 配置器 (allocator) 的简单介绍

3.1 命名空间 (namespace) std

当你采用不同的模块和程序库时，经常会出现名称冲突现象，这是因为不同的模块和程序库可能针对不同的对象使用相同的标识符。namespaces (参见 p16, 2.2.4 节的介绍) 可以解决这个问题。所谓 namespace，是指标识符的某种可见范围。和 class 不同，namespace 具有扩展开放性，可以出现在任何源码文件中。因此你可以利用一个 namespace 来定义一些组件，而它们可散布于多个实质模块上。这类组件的典型例子就是 C++ 标准程序库，因为 C++ 标准程序库使用了一个 namespace。事实上，C++ 标准程序库中的所有标识符都被定义于一个名为 std 的 namespace 中。

由于 namespace 的概念，使用 C++ 标准程序库的任何标识符时，你有三种选择：

1. 直接指定标识符。例如 `std::ostream` 而不是 `ostream`。完整语句类似这样：

```
std::cout << std::hex << 3.4 << std::endl
```
2. 使用 *using declaration* (详见 p17)。例如以下程序片段使我们不必再写出域修饰符号 `std::`，而可直接使用 `cout` 和 `endl`：

```
using std::cout;  
using std::endl;
```

于是先前的例子可以写成这样：


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

3. 使用 *using directive* (详见 p17)，这是最简便的选择。如果对 `namespace std` 采用 *using directive*，便可以让 `std` 内定义的所有标识符都有效（曝光），就好像它们被声明为全局标识符一样。因此，写下：

```
using namespace std;
```

之后，就可以直接写：

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

但请注意，由于某些晦涩的重载（overloading）规则，在复杂的程序中，这种方式可能导致意外的命名冲突，更糟的是甚至导致不一样的行为。如果场合不够明确（例如在头文件、模块或程序库中），就应避免使用 *using directive*。

本书的例子程序都很小，所以，为了方便，书中范例程序通常用最后一种手法。

3.2 头文件 (Header Files)

将 C++ 标准程序库中所有标识符都定义于 `namespace std` 里头，这种做法是标准化过程中引入的。这个做法不具向下兼容性，因为原先的 C/C++ 头文件都将 C++ 标准程序库的标识符定义于全局范围（global scope）。此外标准化过程中有些 classes 的接口也有了更动（当然啦，应尽可能以向下兼容为目标）。为此，特别引入了一套新的头文件命名风格，这么一来组件开发商得以通过“提供旧的头文件”来达到向下兼容的目的。

既然有必要重新定义标准头文件的名称，正好借此机会把头文件扩展名加以规范。以往，头文件扩展名五花八门，包括 `.h`、`.hpp`、`.hxx`。相较之下如今的标准头文件扩展名简洁得令人吃惊：根本就没有扩展名。于是标准头文件的 `#include` 如下：

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

这种写法也适用于 C 标准头文件。但必须采用前缀字符 `c`，而不再是扩展名 `.h`：

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

在这些头文件中，每一个标识符都被声明于 `namespace std`。

这种命名方式的优点之一是可以区分旧头文件中的 `char*` C 函数，和新头文件中的标准 C++ `string` class：

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

注意, 从操作系统角度观之, 新头文件命名方式并不意味标准头文件没有扩展名。标准头文件的 `#include` 该如何处理, 由编译器决定。C++ 系统可以自动添加一个扩展名, 甚至可以使用内建声明, 不读入任何文件。不过实际上大多数系统只是简单含入一个“名称与 `#include` 语句中的文件名完全相同”的文件。所以, 在大部分系统中, C++ 标准头文件都没有扩展名。注意, “无扩展名”这一条件只适用于标准头文件。一般而言, 为你自己所写的头文件加上一个良好的扩展名, 仍然是个好主意, 有助于轻易识别出这些文件的性质。

为了向下兼容于 C, 旧式的 C 标准头文件仍然有效, 如果需要, 你还是可以使用它们, 例如:

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

此时, 标识符同时声明于全局范围和 `namespace std` 中。事实上这些头文件的行为类似于先在 `std` 中声明所有标识符, 再悄悄使用 *using declaration* 把这些标识符引入全局范围 (参见 p17)。

至于 `<iostream.h>` 这一类 C++ 旧式头文件, 标准中并未加以规范 (这一点在标准化过程中曾经多次改变), 意味着不再支持这些头文件。不过目前大多数厂商都会提供它们, 以求向下兼容。

注意, 除了引入 `namespace std`, 头文件还有很多改变。所以, 你要么就采用旧头文件名, 要么就应该完全改用新的标准名称。

3.3 错误 (Error) 处理和异常 (Exception) 处理

C++ 标准程序库由不同的成分构成。来源不同, 设计与实现风格迥异。而错误处理和异常处理正是这种差异的一个典型体现。标准程序库中有一部分, 例如 `string classes`, 支持具体的错误处理, 它们检查所有可能发生的错误, 并于错误发生时抛出异常。至于其它部分如 `STL` 和 `valarrays`, 效率重于安全, 因此几乎不检验逻辑错误, 并且只在执行期 (runtime) 发生错误时才抛出异常。

3.3.1 标准异常类别 (Standard Exception Classes)

语言本身或标准程序库所抛出的所有异常, 都派生自基类 `exception`。这是其它数个标准异常类别的基类, 它们共同构成一个类体系, 如图 3.1 所示。这些标准异常类别可分为三组:

1. 语言本身支持的异常
2. C++ 标准程序库发出的异常
3. 程序作用域 (scope of a program) 之外发出的异常

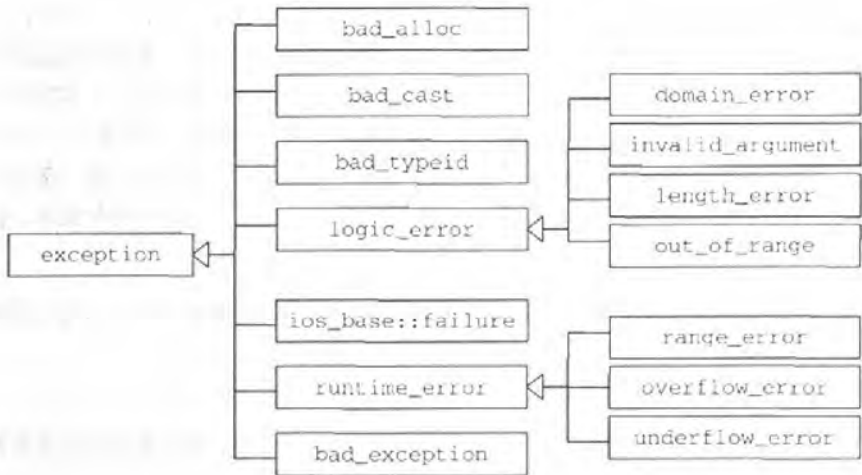


图 3.1 标准异常 (Standard Exceptions) 阶层体系

语言本身所支持的异常

此类异常用以支撑某些语言特性，所以，从某种角度来说它们不是标准程序库的一部分，而是核心语言的一部分。如果以下操作失败，就会抛出这一类异常。

- 全局操作符 `new` 操作失败，会抛出 `bad_alloc` 异常（若采用 `new` 的 `nothrow` 版本，另当别论）。由于这个异常可能于任何时间在任何较复杂的程序中发生，所以可说是最重要的一个异常。
- 执行期间，当一个加诸于 `reference` 身上的“动态型别转换操作”失败时，`dynamic_cast` 会抛出 `bad_cast` 异常。p19 对于 `dynamic_cast` 操作符有些描述。
- 执行期型别辨识 (RTTI) 过程中，如果交给 `typeid` 的参数为零或空指针，`typeid` 操作符会抛出 `bad_typeid` 异常。
- 如果发生非预期的异常，`bad_exception` 异常会接手处理，方式如下：当函数抛出异常规格（`exception specification`，p16 介绍）以外的异常，`bad_exception` 就会调用 `unexpected()`。例如：

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

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

`f()` 之中抛出“型别为 `E2`”的异常, 这种动作违反了异常规格 (exception specification) 的设定, 于是唤起 `unexpected()`, 后者通常会唤起 `terminate()` 终止程序。

然而如果在你的异常规格中列出 `bad_exception`, 那么 `unexpected()` 总是会重新抛出 (rethrows) `bad_exception` 异常。

```
class E1;
class E2; // not derived from E1
void f() throw(E1, std::bad_exception)
           // throws exception of type E1 or
           // bad_exception for any other exception type
{
    ...
    throw E1(); // throws exception of type E1
    ...
    throw E2(); // calls unexpected(), which throws
               bad_exception
}
```

因此, 如果异常规格罗列了 `bad_exception`, 那么任何未列于规格的异常, 都将在函数 `unexpected()` 中被代之以 `bad_exception`¹。

C++ 标准程序库所发生的异常

C++ 标准程序库异常总是派生自 `logic_error`。从理论上讲, 我们能够通过一些手段, 在程序中避免逻辑错误 — 例如对函数参数进行额外测试等等。所谓逻辑错误包括违背逻辑前提或违反 `class` 的不变性。C++ 标准程序库提供以下逻辑错误类别:

- `invalid_argument` 表示无效参数, 例如将 `bitset` (array of bits) 以 `char` 而非 `'0'` 或 `'1'` 进行初始化。
- `length_error` 指出某个行为 “可能超越了最大极限”, 例如对着某个字符串附加太多字符。

¹ 你可以修改 `unexpected()` 的具体操作。然而只要声明有异常规格, 函数就绝不会抛出规格中未列的异常。

- `out_of_range` 指出参数值“不在预期范围内”，例如在诸如 `array` 的容器或字符串 (`string`) 中采用一个错误索引。
- `domain_error` 指出专业领域范畴内的错误。

此外，标准程序库的 I/O 部分提供了一个名为 `ios_base::failure` 的特殊异常。当数据流 (`data stream`) 由于错误或由于到达文件尾端而发生状态改变时，就可能抛出这个异常。此一异常的具体行为见 p602, 13.4.4 节。

程序作用域 (`scope of a program`) 之外发生的异常

派生自 `runtime_error` 的异常，用来指出“不在程序范围内，且不容易回避”的事件。C++ 标准程序库针对执行期错误提供了以下三个 classes:

- `range_error` 指出内部计算时发生区间错误 (`range error`)。
- `overflow_error` 指出算术运算发生上溢位 (`overflow`)。
- `underflow_error` 指出算术运算发生下溢位 (`underflow`)。

标准程序库所抛出的异常

C++ 标准程序库自身可能抛出 `range_error`、`out_of_range` 和 `invalid_argument` 异常。然而由于标准程序库会用到语言特性及客户所写的程序代码，所以也可能间接抛出任何异常。尤其是，无论何时分配存储空间，都有可能抛出 `bad_alloc` 异常。

标准程序库的任何具体实作品，都可能提供额外的异常类别（或作为兄弟类别，或派生为子类别）。使用这些非标准类别将导致程序难以移植，因为一旦你想采用其它标准程序库实作版本，就不得不痛苦地修改你的程序。所以最好只使用标准异常。

异常类别的头文件

基础类别 `exception` 和 `bad_exception` 定义于 `<exception>`。 `bad_alloc` 定义于 `<new>`。 `bad_cast` 和 `bad_typeid` 定义于 `<typeinfo>`。 `ios_base::failure` 定义于 `<ios>`。其它异常类别都定义于 `<stdexcept>`。

3.3.2 异常类别 (Exception Classes) 的成员

为了在 `catch` 子句中处理异常，你必须采用异常所提供的接口。所有标准异常的接口只含一个成员函数：`what()`，用以获取“型别本身以外的附加信息”。它返回一个以 `null` 结束的字符串：

```
namespace std {
    class exception {
    public:
        virtual const char* what() const throw();
```

```

    ...
};
}

```

被返回的字符串，其内容由实作方定义。它很大程度（但非必然）决定了帮助的级别和信息的详细度。注意，该字符串有可能是个以 null 结尾的 "multibyte" 字符串，可被轻松转换为 `wstring`（详见 p480, 11.2.1 节）并显示出来。`what()` 返回的 C-string 在其所属的异常对象被摧毁后，就不再有效了²。

标准异常中的其它成员，用来处理生成、复制、赋值、销毁等动作。要注意的是，除了 `what()`，再没有任何异常提供任何其它成员函数，能够描述异常的种类。例如，没有可找出异常上下文 (context) 的一致性方法，或找出区间错误 (range error) 发生时的错误索引值。因此，唯一通用的异常评估手段，大概只有打印一途了：

```

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

```

唯一可能实现的另一个异常评估手段是，根据异常的精确型别，自己得出推论。例如，如果 `bad_alloc` 异常被抛出，可能是因为程序企图获得更多内存。

3.3.3 抛出标准异常

你可以在自己的程序库或程序内部抛出某些标准异常。允许你这般运用的各个标准异常，生成时都只需要一个 `string` 参数（第 11 章对 `string class` 有所描述），它将成为被 `what()` 返回的描述字符串。例如 `logic_error` 定义如下：

```

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

```

² C++ 标准规格并未对 `what()` 返回值的寿命加以规范，这里所讲的是被提出的一种建议解决方案。

提供这种功能的标准异常有: `logic_error` 及其派生类别、`runtime_error` 及其派生类别、`ios_base::failure`。你不能抛出 `exception`, 也不能抛出任何用以支持语言核心性质的异常。

想要抛出一个标准异常, 只需生成一个描述该异常的字符串, 并将它初始化, 交给异常对象:

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

由于 `char*` 可被隐式转换为 `string`, 所以你可以直接使用字符串字面量:

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

3.3.4 从标准异常类别 (Exception Classes) 中派生新类别

另一个在程序中采用标准异常类别的可能情况是, 定义一个直接或间接派生自 `exception` 的特定异常类别。要这么做, 首先必须确保 `what()` 机制正常运作。`what()` 是个虚拟函数, 所以提供 `what()` 的方法之一就是自己实现 `what()`:

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

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

提供 `what()` 函数的另一种方法是，令你的异常类别派生自 3.3.3 节所描述的标准异常：

```
namespace MyLib {  
    /* user-defined exception class  
     * - derived from a standard class for exceptions  
     * that has a constructor for the what() argument  
     */  
  
    class MyRangeProblem : public std::out_of_range {  
    public:  
        MyRangeProblem (const string& whatString)  
            : out_of_range(whatString) {  
        }  
    };  
    ...  
  
    void f() {  
        ...  
        // create an exception object by using a string constructor and throw it  
        throw MyRangeProblem("here is my special range problem");  
        ...  
    }  
}
```

完整程序见 p441 的 `Stack` 和 p450 的 `Queue`。

3.4 配置器 (Allocators)

C++标准程序库在许多地方采用特殊对象来处理内存配置和寻址，这样的对象称为配置器 (allocator)。配置器体现出一种特定的内存模型 (memory model)，成为一个抽象表征，表现出“内存需求”至“内存低阶调用”的转换。如果运用多个不同的配置器对象，你便可以在同一个程序中采用不同的内存模型。

配置器最初是作为 STL 的一部分而引进的，用于处理诸如 PC 上不同指针型别（例如 `near`, `far`, `huge` 指针）这一类乱七八糟的问题；现在则作为一种技术方案的基础。

使得诸如共享内存 (shared memory)、垃圾回收 (garbage collection)、面向对象数据库 (object oriented databases) 等特定内存模型, 能够保持一致的接口。但是这种用法还相当新颖, 尚未获得广泛的接受 (情况正在改变中)。

C++标准程序库定义了一个缺省配置器 (default allocator) 如下:

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

缺省配置器可在任何“配置器得以被当做参数使用”的地方担任默认值。缺省配置器会执行内存分配和回收的一般性手法, 也就是呼叫 new 和 delete 操作符。但是 C++ 并没有对于“在什么时候以什么方式调用这些操作符”给予明确规定。所以, 缺省配置器甚至可能对已分配之内存施行“内部 cache (internal cache)”手法。

绝大多数程序都采用缺省配置器, 但有时候其它程序库也可能提供某些配置器以满足特定需求。这种情况下只需简单地将它们当做参数即可。自行设计并实作配置器的实际意义不大。实际生活中最典型的方式还是直接采用缺省配置器, 所以我将迟至第 15 章才详细探讨配置器。(译注: 举个例子, SGI STL 对外呈现一个透通的配置器接口, 内部却维护有一、二两级配置器, 制作出十分精巧繁复的 memory pool 机制, 对于体积小而数量极大的对象需求而言, 可带来极好的时间和空间效率。详见《STL 源码剖析》第 2 章)

4

通用工具

Utilities

本章讲解 C++ 标准程序库中的通用工具。它们由短小精干的类和函数构成，执行最一般性的工作。这些工具包括：

- 数种通用型别 (general types)
- 一些重要的 C 函数
- 数值极值¹ (numeric limits)

大部分通用工具在 C++ 标准规格书第 20 款 (clause) 描述，定义于标准头文件 `<utility>` 内。其余工具则与标准程序库中一些比较主要的组件一起描述，其原因可能是该类工具主要便是和那些组件共同使用，抑或因为历史因素。例如某些通用辅助函数被定义于 `<algorithm>` 头文件中，但按照 STL 的定义，它们称不上是算法 (参见第 5 章)。

这些工具中的一部分也被运用于 C++ 标准程序库中。特别是型别 `pair`，凡需要将两个值视为一个单元的场所 (例如必须“返回两个值”的某函数)，就必须用到它。

4.1 Pairs (对组)

`class pair` 可以将两个值视为一个单元。C++ 标准程序库内多处使用了这个 `class`。尤其容器类别 `map` 和 `multimap`，就是使用 `pairs` 来管理其键值/实值 (key/value) 的成对元素 (详见 6.6 节, p194)。任何函数需返回两个值，也需要 `pair`。

¹ 可能有些人认为数值极值应该属于第 12 章，也就是专门讲解数值的那一章，但这些数值极值在程序库的其它部分也会被用到，所以我决定把它放在这里。

Structure `pair` 定义于 `<utility>`:

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

        // member
        T1 first;
        T2 second;

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

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

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

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

```

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

```

注意, `pair` 被定义为 `struct`, 而不是 `class`, 这么一来, 所有成员都是 `public`, 我们因此可以直接存取 `pair` 中的个别值。

上述 `default` 构造函数生成一个 `pair` 时, 以两个“被该 `default` 构造函数个别初始化”的值作为初值。根据语言规则, 基本型别 (如 `int`) 的 `default` 构造函数也可以引起适当的初始化动作, 所以:

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

就是以 `int()` 和 `float()` 来初始化 `p`。这两个构造函数都返回零值。p14 曾经讨论过基本型别的显式初始化动作。

这里之所以使用 `template` 形式的 `copy` 构造函数, 乃是因为构造过程中可能需要隐式型别转换。如果 `pair` 对象被复制, 调用的是由系统隐式生成的那个 `copy` 构造函数²。例如:

```

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

```

Pair 之间的比较

为了比较两个 `pair` 对象, C++ 标准程序库提供了大家惯用的操作符。如果两个 `pair` 对象内的所有元素都相等, 这两个 `pair` 对象就被视为相等 (`equal`):

```

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

```

² `template` 形式的构造函数并不会遮掩 (由编译器) 隐式生成的 `default` 构造函数。详见 p13。

两个 `pairs` 互相比对时, 第一元素具有较高的优先级。所以如果两个 `pairs` 的第一元素不相等, 其比较结果就成为整个比较行为的结果。如果第一元素相等, 才继续比较第二元素, 并把比较结果当做整体比较结果。

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

其它的比较操作符 (comparison operators) 也如法炮制。

4.1.1 便捷函数 `make_pair()`

`template` 函数 `make_pair()` 使你无需写出型别, 就可以生成一个 `pair` 对象³:

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

例如, 你可以这样使用 `make_pair()`:

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

而不必费力地这么写:

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

当我们有必要对一个接受 `pair` 参数的函数传递两个值时, `make_pair()` 尤其显得方便, 请看下例:

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

³ 使用 `make_pair` 并不会多花你任何执行时间, 编译器应该会将此一动作优化。

从例子中可以看出, `make_pair()` 使得“将两个值当做一个 `pair` 参数来传递”的动作更容易。即使两个值的型别并不准确符合要求, 也能在 `template` 构造函数提供的支持下顺利工作。当你使用 `map` 和 `multimap`, 你会经常用到这个特点 (详见 p203)。

注意, 一个算式如果明白指出型别, 便带有一个优势: 产生出来的 `pair` 将有绝对明确的型别。例如:

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

其结果与:

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

不同。后者所生成的 `pair`, 第二元素的型别是 `double` (因为“无任何饰词的浮点字面常数”, 其型别被视为 `double`)。当我们使用重载函数 (`overloaded function`) 或 `template`, 确切的型别非常重要。例如, 为了提高效率, 程序员可能同时提供分别针对 `float` 和 `double` 的 `function` 或 `template`, 这时候确切的型别就非常重要了。

4.1.2 Pair 运用实例

C++ 标准程序库大量运用了 `pair`。例如 `map` 和 `multimap` 容器的元素型别便是 `pair`, 也就是一组键值/实值 (`key/value`)。关于 `maps` 和 `multimaps` 的一般性描述, 详见 p194, 6.6 节。p91 有一个 `pair` 型别的运用实例。C++ 标准程序库中凡是“必须返回两个值”的函数, 也都会利用 `pair` 对象 (实例请见 p183)。

4.2 Class auto_ptr

本节描述 auto_ptr 型别。C++ 标准程序库提供的 auto_ptr 是一种智能型指针 (smart pointer)，帮助程序员防止“被异常抛出时发生资源泄漏”。注意我说的是“一种”智能型指针，现实生活中还有其它许多有用的智能型指针，auto_ptr 只是针对某个特定问题而设计，对于其它问题，auto_ptr 无能为力。所以，请认真阅读以下内容。

4.2.1 auto_ptr 的设计动机

函数的操作经常依以下模式进行⁴：

1. 获取一些资源。
2. 执行一些动作。
3. 释放所获取的资源。

如果一开始获取的资源被绑定于局部对象 (local objects) 身上，当函数退出时，它们的析构函数 (destructor) 被调用，从而自动释放这些资源。然而事情并不总是如此顺利，如果资源是以显式手法 (explicitly) 获得，而且没有被绑定在任何对象身上，那就必须以显式手法释放。这种情形常常发生在指针身上。

一个典型的例子就是运用 new 和 delete 来产生和销毁对象：

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

也许你尚未意识到，这个函数其实是一系列麻烦的根源。一个显而易见的问题是，我们经常忘掉 delete 动作，特别是当函数中间存在 return 语句时更是如此。然而真正的麻烦发生于更隐晦之处，那就是当异常发生时我们所要面对的灾难。异常一旦出现，函数将立刻退离，根本不会调用函数尾端的 delete 语句。结果可能是内存遗失，或更一般地说是资源遗失。防止这种资源遗失的常见办法就是捕捉所有异常，例如：

⁴ class auto_ptr 的设计推进，是以 Scott Meyers 所著《More Effective C++》书中的相关资料为基础（并获得他的允许）。这个问题的一般性技术最早描述于 Bjarne Stroustrup 的《The C++ Programming Language》2nd Edition 和《The Design and Evolution of C++》中，当时的主题是“resource allocation is initialization”。auto_ptr 被加入 C++ 标准之中，正是为了支持此一技术。

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

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

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

你看，为了在异常发生时处理对象的删除工作，程序代码变得多么复杂和累赘！如果还有第二个对象，如果还要比照办理，如果还需要更多的 catch 子句，那简直是一场恶梦。这不是优良的编程风格，复杂而且容易出错，必须尽力避免。

如果使用智能型指针，情形就会大不相同。这个智能型指针应该保证，无论在何种情形下，只要自己被摧毁，就一定连带释放其所指资源。而由于智能型指针本身就是区域变量，所以无论是正常退出，还是异常退出，只要函数退出，它就一定会被销毁。auto_ptr 正是这种智能型指针。

auto_ptr 是这样一种指针：它是“它所指向的对象”的拥有者 (owner)。所以，当身为对象拥有者的 auto_ptr 被摧毁时，该对象也将遭到摧毁。auto_ptr 要求一个对象只能有一个拥有者，严禁一物二主。

下面是上例改写后的版本：

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

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

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

不再需要 delete，也不再需要 catch 了。auto_ptr 的接口与一般指针非常相似：operator* 用来提领其所指对象，operator-> 用来指向对象中的成员。然而，所有指针算术（包括 ++）都没有定义（这可能是件好事，因为指针算术是一大麻烦来源）。

注意, `auto_ptr<>` 不允许你使用一般指针惯用的赋值 (`assign`) 初始化方式。你必须直接使用数值来完成初始化⁵:

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

4.2.2 `auto_ptr` 拥有权 (Ownership) 的转移

`auto_ptr` 所界定的乃是一种严格的拥有权观念。也就是说, 由于一个 `auto_ptr` 会删除其所指对象, 所以这个对象绝对不能同时被其它对象“拥有”。绝对不应该出现多个 `auto_ptr`s 同时拥有一个对象的情况。不幸的是, 这种事情可能会发生 (如果你以同一个对象为初值, 将两个 `auto_ptr`s 初始化, 就会出现这种事情)。程序员必须负责防范这种错误。

这个条件导致了一个问题: `auto_ptr` 的 `copy` 构造函数和 `assignment` 操作符应当如何运作? 此类操作往往是将此处数据拷贝到彼处。然而这种操作恰恰会导致上面所提的情形。解决办法很简单, 但意义深远: 令 `auto_ptr` 的 `copy` 构造函数和 `assignment` 操作符将对象拥有权交出去。试看下列 `copy` 构造函数的运用:

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

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

在第一个语句中, `ptr1` 拥有了那个 `new` 出来的对象。在第二个语句中, 拥有权由 `ptr1` 转交给 `ptr2`。此后 `ptr2` 就拥有了那个 `new` 出来的对象, 而 `ptr1` 不再拥有它。这样, 对象就只会被 `delete` 一次——在 `ptr2` 被销毁的时候。

⁵ 下面两种情况实际上是有分别的:

```
X x;
Y y(x);    // 显式转换 (explicit conversion)
和;
```

```
X x;
Y y = x;    // 隐式转换 (implicit conversion)
```

前者使用显式转换, 以型别 `x` 构造型别 `Y` 的一个新对象, 后者使用隐式转换。

赋值动作也差不多：

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

在这里，赋值动作将拥有权从 ptr1 转移至 ptr2。于是，ptr2 拥有了先前被 ptr1 所拥有的那个对象。

如果 ptr2 被赋值之前正拥有另一个对象，赋值动作发生时调用 delete，将该对象删除：

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

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

注意，拥有权的转移，实质上并非只是被简单拷贝而已。只要发生了拥有权转移，先前的拥有者（本例为 ptr1）就失去了拥有权，结果，拥有者一旦交出拥有权，就两手空空，只剩一个 null 指针在手了。在这里，copy 构造函数更动了“用以初始化新对象”的原对象，而赋值操作也修改了右侧对象，这和程序语言中惯常的初始化动作和赋值动作可说大相径庭。那么谁来保证那个“失去了所有权、只剩一个 null 指针”的原 auto_ptr 不会再次进行提领动作呢？是的，这还是程序员的责任。

只有 auto_ptr 可以拿来当做另一个 auto_ptr 的初值，普通指针是不行的：

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

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

起点和终站 (source and sink)

拥有权的移转，使得 auto_ptr 产生一种特殊用法：某个函数可以利用 auto_ptr 将拥有权转交给另一个函数。这种事情可能在两种情形下出现：

1. 某函数是数据的终点。如果 `auto_ptr` 以 `by value` (传值) 方式被当做一个参数传递给某函数, 就是这种情况。此时被调用端的参数获得了这个 `auto_ptr` 的拥有权, 如果函数不再将它传递出去, 它所指的对象就会在函数退出时被删除:

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

2. 某函数是数据的起点。当一个 `auto_ptr` 被返回, 其拥有权便被转交给调用端了。见下例:

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

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

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

每当 `f()` 被调用, 它都 `new` 一个新对象, 然后把该对象连同其拥有权一起返回给调用端。将返回值赋值给 `p`, 同时也完成了拥有权的移转。一旦循环再次执行这个赋值动作, `p` 原先拥有的对象将被删除。离开 `g()` 时, `p` 也会被销毁, 这样就删除了 `p` 所拥有的最后一个对象。无论如何都不会有资源遗失之虞。即使有异常被抛出, 拥有资料的 `auto_ptr` 也会尽职地将自己的数据删除。

缺陷

`auto_ptr` 的语义本身就包含了拥有权, 所以如果你无意转交你的拥有权, 就不要在参数列中使用 `auto_ptr`, 也不要将它作为返回值。下面例子是一个幼稚的作法, 原本是想将 `auto_ptr` 所指对象打印出来, 实际上却引发了一场灾难:

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

```

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

```

只要有一个 `auto_ptr` 被当做参数，放进这个 `bad_print()` 函数，它所拥有的对象（如果有的话）就一定会被删除。因为作为参数的 `auto_ptr` 会将拥有权转交给参数 `p`，而当函数退出时，会删除 `p` 所拥有的对象。这恐怕不是程序员所希望的，最终必然会引起致命的执行期错误：

```

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

```

你可能会认为，将 `auto_ptr`s 以 `pass by reference` 方式传递就万事大吉。然而这种行为却会使“拥有权”的概念变得难以捉摸，因为面对一个“透过 `reference` 而获得 `auto_ptr`”的函数，你根本无法预知拥有权是否被转交。所以以 `by reference` 方式传递 `auto_ptr` 是非常糟糕的设计，应该全力避免。

考虑到 `auto_ptr` 的概念，我们倒是可以运用 `constant reference`，向函数传递拥有权。然而这也十分危险，因为当你传递一个 `constant reference` 时，通常预期该对象不会被更动。幸好 `auto_ptr`s 的一个晚期设计降低了此一危险性。通过某些实作技巧，我们可以令 `constant reference` 无法交出拥有权。事实上，你无法变更任何 `constant reference` 的拥有权：

```

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

```

这一方案使得 `auto_ptr`s 比以前显得更安全一些。很多接口在需要内部拷贝时，都通过 `constant reference` 获得原值。事实上，C++ 标准程序库的所有容器（例见第 6 章或第 10 章）都如此，大致像这样：

```

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

```

如果这一类赋值操作对 `auto_ptr` 有效, 那么拥有权就会被转交给容器。然而正由于 `auto_ptr` 的实际设计, 这种行为必然会导致编译错误:

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

总而言之, 常数型 `auto_ptr` 减小了“不经意转移拥有权”所带来的危险。只要一个对象通过 `auto_ptr` 传递, 就可以使用常数型 `auto_ptr` 来终结拥有权转移链, 此后拥有权将不能再进行转移。

在这里, 关键词 `const` 并非意味你不能更改 `auto_ptr` 所拥有的对象, 而是意味你不能更改 `auto_ptr` 的拥有权。例如:

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

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

如果使用 `const auto_ptr` 作为参数, 对新对象的任何赋值操作都将导致编译期错误。就常数特性而言, `const auto_ptr` 比较类似常数指针 (`T* const p`), 而非指向常数的指针 (`const T* p`) ——尽管其语法看上去比较像后者。

4.2.3 `auto_ptr` 作为成员之一

在 `class` 中使用 `auto_ptr`, 你可以因而避免遗失资源。如果你以 `auto_ptr` 而非一般指针作为成员, 当对象被删除时, `auto_ptr` 会自动删除其所指的成员对象, 于是你也就不再需要析构函数了。此外, 即使在初始化期间抛出异常, `auto_ptr` 也可以帮助避免资源遗失。注意, 只有当对象被完整构造成功, 才有可能于将来调用其析构函数。这造成了资源遗失的隐忧: 如果第一个 `new` 成功了, 第二个 `new` 却失败了, 就会造成资源遗失。例如:

```

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

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

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

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

```

使用 auto_ptr, 你就可以轻松避免这场悲剧:

```

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

```

```

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

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

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

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

```

然而请注意，尽管你可以略过析构函数，却还是不得不亲自撰写 `copy` 构造函数和 `assignment` 操作符。缺省状况下，这两个操作都会转交拥有权，这恐怕并非你所愿。正如 p42 所说，为了避免拥有权的意外转交，如果你的 `auto_ptr` 在整个生命期内都不必改变其所指对象的拥有权，你可以使用 `const auto_ptr`。

4.2.4 `auto_ptr` 的错误运用

`auto_ptr` 确实解决了一个特定问题，那就是在异常处理过程中的资源遗失问题。不幸的是由于 `auto_ptr` 的具体行为方式曾经三番五次地改动，而且 C++ 标准程序库中只此一个智能型指针 (`smart pointer`)，别无分号，所以人们总是会误用 `auto_ptr`。为了帮助你正确使用它，这里给出一些要点：

1. `auto_ptr` 之间不能共享拥有权

一个 `auto_ptr` 千万不能指向另一个 `auto_ptr` (或其它对象) 所拥有的对象。否则，当第一个指针删除该对象后，另一个指针突然间指向了一个已被销毁的对象，那么，如果再使用那个指针进行读写操作，就会引发一场灾难。

2. 并不存在针对 array 而设计的 auto_ptrs

auto_ptr 不可以指向 array，因为 auto_ptr 是透过 delete 而非 delete[] 来释放其所拥有的对象。注意，C++ 标准程序库并未提供针对 array 而设计的 auto_ptr。标准程序库另提供了数个容器类别，用来管理数据群（参见第 5 章）。

3. auto_ptrs 决非一个“四海通用”的智能型指针

并非任何适用智能型指针的地方，都适用 auto_ptr。特别请注意的是，它不是引用计数（*reference counting*）型指针——这种指针保证，如果有一组智能型指针指向同一个对象，那么当且仅当最后一个智能型指针被销毁时，该对象才会被销毁。

4. auto_ptrs 不满足 STL 容器对其元素的要求

auto_ptr 并不满足 STL 标准容器对于元素的最基本要求，因为在拷贝（copy）和赋值（assign）动作之后，原本的 auto_ptr 和新产生的 auto_ptr 并不相等。是的，拷贝和赋值之后，原本的 auto_ptr 会交出拥有权，而不是拷贝给新的 auto_ptr。因此请绝对不要将 auto_ptr 作为标准容器的元素。幸好语言和程序库的设计本身就可以防止这种误用，如果你的工作环境符合标准，这类误用应该无法通过编译。

不幸的是，某些时候，即使误用 auto_ptr，程序仍然能够顺利运作。就此点而言，使用一个非常数（nonconstant）auto_ptr，并不比使用一个一般指针更安全。如果你的误用行为没有导致全盘崩溃，你或许会暗自庆幸，而这其实是真正的不幸，因为你或许根本就没有意识到你已经犯了错误。关于“引用计数型（reference counting）智能型指针”的讨论请见 p135, 5.10.2 节，p222, 6.8 节有一份实现例程。当我们有必要在不同容器之间共享元素时，这种指针非常有用。

4.2.5 auto_ptr 运用实例

下面第一个例子展示 auto_ptrs 移转拥有权的行为：

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

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



```
        if (p.get() == NULL) {
            strm << "NULL";      // NO: print NULL
        }
        else {
            strm << *p;           // YES: print the object
        }
        return strm;
    }

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

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

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

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

程序输出如下:

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

注意，output 操作符的第二个参数是一个 const reference，所以并没有发生拥有权的移转。

正如我在 p40 所说，请时刻铭记于心，你不能以一般指针的赋值手法来初始化一个 auto_ptr：

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

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

这是因为，“根据一般指针生成一个 auto_ptr”的那个构造函数，被声明为 explicit（关于 explicit，详见 p18, 2.2.6 节）。

下面这个例子展示 const auto_ptr 的特性：

```
// util/autoptr2.cpp

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

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

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

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

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

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

程序输出如下:

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

这个例子为 `auto_ptr` 定义了一个 `output` 操作符, 其中将 `auto_ptr` 以 `const reference` 的方式传递。根据 p43 的讨论, 你不应该以任何形式传递 `auto_ptr`, 但此处是个例外。

注意下列赋值操作是错误的:

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

这个句子对于一个“未指向任何对象”的 `auto_ptr` 进行提领 (*dereference*) 操作。C++ 标准规格, 这会导致未定义行为, 比如说导致程序的崩溃。从这个例子可以看出, 你可以操作 `const auto_ptr` 所指对象本身, 但“它所拥有的究竟是哪个对象”这一事实无法改变。就算 `r` 不具常数性, 最后一个语句也会失败, 因为 `p` 具有常数性, 其拥有权不得被更改。

4.2.6 auto_ptr 实作细目

class auto_ptr 声明于 <memory>:

```
#include <memory>
```

auto_ptr 定义于 namespace std 中, 是“可用于任何型别身上”的一个 template class。

下面是 auto_ptr 的确切声明⁶:

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

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

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

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

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

        // destructor
        ~auto_ptr() throw();

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

⁶ 这里所给的版本较之 C++ 标准程序库的版本作了一些小小改进, 修正了几个小问题 (这里的 auto_ptr_ref 是全局的, 这里并且设定了从 auto_ptr_ref 到 auto_ptr 的 assignment 操作符, 参见 p55)。

```

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

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

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

```

个别成员的描述将在后续各节中进行。讨论过程中我把 `auto_ptr<T>` 简称为 `auto_ptr`。p56 有一份完整的 `auto_ptr` 实作范例。

型别定义

`auto_ptr::element_type`

- `auto_ptr` 所拥有之对象的型别

构造函数 (constructor)、赋值操作符 (assign operator)、析构函数 (destructor)

`auto_ptr::auto_ptr()` throw()

- default 构造函数。
- 生成一个不拥有任何对象的 `auto_ptr`。
- 将 `auto_ptr` 的值初始化为零。

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

- 生成一个 `auto_ptr`，并拥有 `ptr` 所指对象。
- 这一操作完成后，`*this` 成为 `ptr` 所指对象的唯一拥有者。不允许再有其它拥有者。

- 如果 ptr 本身不是 null 指针，那就必须是个 new 返回值，因为 auto_ptr 析构函数会对其所拥有的对象自动调用 delete。
- 不能用 new[] 所生成的 array 作为初值。当你需要 array，请考虑使用 STL 容器，p75, 5.2 节对此有些介绍。

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

- 针对 non-const values 而设计的一个 copy 构造函数。
- 生成一个 auto_ptr，在入口处将 ap 所拥有的对象（如果有的话）的拥有权夺取过来。
- 此操作完毕之后，ap 不再拥有任何对象，其值变为 null 指针。所以，和一般 copy 构造函数不同，这个操作改变了原对象。
- 注意，此函数有一个重载的 member template（请参考 p11），使得 ap 可通过型别自动转换，构造出合适的 auto_ptr。例如，根据一个“衍生类的对象”，构造出一个基类对象的 auto_ptr。
- 拥有权移转问题，请参考 p40, 4.2.2 节。

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

- 针对 non-const value 而设计的一个 assignment（赋值）操作符。
- 如果自身原本拥有对象，进入本动作时将被删除，然后获得 ap 所拥有的对象。于是，原本 ap 所拥有的对象的拥有权就移转给了 *this。
- 此一操作完成后，ap 不再拥有任何对象。其值变为 null 指针。与一般赋值操作不同，此处这个操作改变了原对象。
- 左手边的 auto_ptr 原本所指对象将被删除（deleted）。
- 注意，此函数有一个重载的 member template（请参考 p11）。这使得 ap 可通过“型别自动转换”赋值给合适的 auto_ptr。例如，将一个“衍生类的对象”，赋值给一个基类对象的 auto_ptr。
- 拥有权移转问题，请参考 p40, 4.2.2 节。

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

- 析构函数
- 如果 auto_ptr 拥有某个对象，此处将调用 delete 删除之。

数值存取 (value access)

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

- 返回 `auto_ptr` 所指对象的地址。
- 如果 `auto_ptr` 未指向任何对象，返回 `null` 指针。
- 这个动作并不改变拥有权。退出此函数时，`auto_ptr` 仍然保有对对象（如果有的话）的拥有权。

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

- `dereference`（提领）操作符。
- 返回 `auto_ptr` 所拥有的对象。
- 如果 `auto_ptr` 并未拥有任何对象，此调用导致未定义行为（可能导致崩溃）。

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

- `member access`（成员存取）操作符
- 返回 `auto_ptr` 所拥有的对象中的一个成员。
- 如果 `auto_ptr` 并未拥有任何对象，此调用将导致未定义行为（可能导致崩溃）。

数值操作

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

- 放弃 `auto_ptr` 原先所拥有之对象的拥有权。
- 返回 `auto_ptr` 原先拥有对象（如果有的话）的地址。
- 如果 `auto_ptr` 原先并未拥有任何对象，返回 `null` 指针。

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

- 以 `ptr` 重新初始化 `auto_ptr`。
- 如果 `auto_ptr` 原本拥有对象，则此动作开始前先删除之。
- 调用结束后，`*this` 成为 `ptr` 所指对象的拥有者。注意，不应该有任何其它拥有者。
- 如果 `ptr` 不是 `null` 指针，应当是一个由 `new` 返回的值，因为 `auto_ptr` 的析构函数会调用 `delete` 来删除其所拥有的对象。
- 注意，不得将通过 `new[]` 生成的数组当做参数传进来。如果需要使用数组，请考虑使用 STL 容器类，详见 p75, 5.2 节。

Conversions (转型操作)

auto_ptr 中剩余的内容 (辅助型别 auto_ptr_ref 及其相关函数) 涉及非常精致的技巧, 使我们得以拷贝和赋值 non-const auto_ptrs, 却不能拷贝和赋值 const auto_ptrs (详见 p44)。下面是一份扼要解释⁷。我们有两个需求:

1. 我们需要将 auto_ptr 作为右值 (rvalue) 传递到函数去, 或由函数中返回⁸。由于 auto_ptr 是个类别, 所以这些工作应当由构造函数完成。
2. 拷贝 auto_ptr 时, 原指针务必放弃拥有权。这就要求拷贝动作必须修改原本的那个 auto_ptr。

一般的 copy 构造函数当然可以拷贝右值, 但为了做到这点, 它必须将其参数型别声明为一个 *reference to const object*。如果在 auto_ptr 中使用一般的 copy 构造函数, 我们恐怕不得不将 auto_ptr 内含的实际指针声明为 mutable, 只有这样, 才能在 copy 构造函数中更改它。你以为万事大吉了吗? 错, 这种做法将允许用户拷贝那些声明为 const 的对象, 将其拥有权转交他人, 这与其原本的常数性背道而驰。

变通作法是找出一种机制, 能够将右值转化为左值。“直接转型为 reference”的那种简单的转型操作符派不上用场, 因为当你实际上是把一个对象转化为自己原本的型别时, 不会有任何转型操作被调用 (切记, reference 属性并非型别的一部分)。为此才有了 auto_ptr_ref 类别的引进, 协助我们将右值转化为左值。这一机制的理论基础是“重载”和“template 参数推导规则”之间一个细微的不同处。这个差别实在太细微了, 不大可能作为一般程序编写技巧而用于别处, 但却足以在这里让 auto_ptr 正确运作。这就够了。

如果你的编译器对于 non-const 和 const auto_ptrs 之间的区别尚不能作出很好的阐释, 请不必惊讶。但是请你保持清醒的头脑, 如果你的编译器尚未达到这一水平, 那么 auto_ptr 的使用就会变得更加危险。因为这种情况下很容易意外地将拥有权旁落他人之手。

⁷ 感谢 Bill Gibbons 指出这一点。

⁸ rvalue (右值) 和 lvalue (左值) 的名称由来, 是从赋值运算 $\text{expr1} = \text{expr2}$ 得来。在这种表达式中, 左操作数 expr1 必须是一个 (可更改的) lvalue。不过或许更贴切的描述是: lvalue 代表 *locator value*。也就是说, 这个表达式通过名字和引用值 (pointer 或 reference) 来指定一个对象。lvalue 并非一定“可被更改”。例如常数对象的名字就是一个不可被改动的 lvalue。所有 non-lvalues 对象, 都是 rvalues。尤其显式生成 $T()$ 的临时对象和函数返回值, 都是 rvalue。

类别 auto_ptr 的实作范例

以下源码展示了一个符合标准的 auto_ptr 类别的实作范例⁹:

```
// util/autoptr.hpp

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

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

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

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

⁹ 感谢 Greg Colvin 提供的这份 auto_ptr 实作范例。注意，这个实作版本并不完全契合 C++ 标准规范。事实证明，C++ 标准所规定的形式中，当利用 auto_ptr_ref 进行转型时，在某种特殊情况下仍会出现小小瑕疵。这里所给的方案很有可能彻底解决所有问题。不过，撰写本书的时候，仍有一些相关讨论正在进行。

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

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

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

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

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

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

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

4.3 数值极限 (Numeric Limits)

一般说来, 数值型别 (Numeric types) 的极值是一个与平台相关的特性。C++ 标准程序库通过 `template numeric_limits` 提供这些极值, 取代传统 C 语言所采用的预处理器常数 (preprocessor constants)。你仍然可以使用后者, 其中整数常数定义于 `<climits>` 和 `<limits.h>`, 浮点常数定义于 `<cfloat>` 和 `<float.h>`。新的极值概念有两个优点, 第一是提供了更好的型别安全性, 第二是程序员可借此写出一些 `templates` 以核定 (*evaluate*) 这些极值。

本节的剩余部分专门讨论极值问题。注意, C++ *Standard* 规定了各种型别必须保证的最小精度, 如果你能够注意并运用这些极值, 就比较容易写出与平台无关的程序。这些最小值列于表 4.1。

表 4.1 内建型别的最小长度

型别	最小长度
char	1 byte (8 bits)
short int	2 bytes
int	2 bytes
long int	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

`Class numeric_limits<>`

使用 `template`, 通常是为了对所有型别一次性地撰写出一个通用解决方案。除此之外, 你还可以在必要时以 `template` 为每个型别提供共同接口。方法是: 不但提供通用性的 `template`, 还提供其特化 (*specialization*) 版本。`numeric_limits` 就是这项技术的一个典型例子, 作法如下:

- 通用性的 `template`, 为所有型别提供缺省极值:

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

```

        numeric_limits
    };
}

```

这个通用性 `template` 将成员 `is_specialized` 设为 `false`, 意思是, 对型别 `T` 而言, 无所谓极值的存在。

- 各具体型别的极值, 由特化版本 (specialization) 提供:

```

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

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

```

这里把 `is_specialized` 设为 `true`, 所有其它成员都根据特定型别的具体极值加以设定。

通用性的 `numeric_limits` `template`, 及其特化版本都被放在 `<limits>` 头文件中。C++ *Standard* 所囊括的特化版本, 涵盖了所有数值基本型别, 包括: `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`。你可以轻易为你自定的数值型别加上补充。

表 4.2 和表 4.3 列出 `class numeric_limits<>` 的所有成员及其意义。最右一列显示对应的 C 常数, 它们分别定义于 `<climits>`, `<limits.h>`, `<cfloat>`, `<float.h>` 内。

表 4.2 class numeric_limits<> 的所有成员

成员	意义	对应的 C 常数
is_specialized	型别是否有极值	
is_signed	型别带有正负号	
is_integer	整数型别	
is_exact	计算结果不产生舍/入误差 (此成员对所有整数型别而言均为 true)	
is_bounded	数值集的个数有限 (对所有内建型别而言, 此成员均为 true)	
is_modulo	两正值相加, 其结果可能因溢位而回绕为较小的值	
is_iec559	遵从 IEC 559 及 IEEE 754 标准	
min()	最小值 (对浮点数而言, 是标准化后的值; 只有当 is_bounded !is_signed 成立时才有意义)	INT_MIN, FLT_MIN, CHAR_MIN, ...
max()	最大值 (只有当 is_bounded 成立时才有意义)	INT_MAX, FLT_MAX, ...
digits	字符和整数: 不带正负号之位个数 浮点数: 尾数中之 radix (见下) 位个数	CHAR_BIT FLT_MANT_DIG, ...
digits10	十进制数的个数 (只有当 is_bounded 成立时才有意义)	FLT_DIG, ...
radix	整数: 表示式的底数 (base), 几乎总是 2 浮点数: 指数表示式的底数 (base)	FLT_RADIX
min_exponent	底数 radix 的最小负整数指数	FLT_MIN_EXP, ...
max_exponent	底数 radix 的最大正整数指数	FLT_MAX_EXP, ...
min_exponent10	底数 10 的最小负整数指数	FLT_MIN_10_EXP, ...
max_exponent10	底数 10 的最大正整数指数	FLT_MAX_10_EXP, ...
epsilon()	1 和最接近 1 的值之间的差距	FLT_EPSILON, ...
round_style	舍/入 (rounding) 风格 (见 p63)	
round_error()	最大舍/入误差量测 (根据 ISO/IEC 10967-1 标准)	
has_infinity	有“正无穷大”表示式	
infinity()	“正无穷大” (如有的话)	
has_quiet_NaN	本型别有不发信号 (nonsignaling) 的“非数值”表述式	
quiet_NaN()	如果可以, 安静地 (nonsignaling) 表述出“这不是个数值”	
has_signaling_NaN	本型别会有发出信号 (signaling) 的“非数值”表述式	
signaling_NaN()	如果可以, 发出信号 (signaling) 表述出“这不是个数值”	

续表 4.2

成员	意义	对应的 C 常数
has_denorm	本型别是否允许非标准化数值 (denormalized values, 也就是 variable members of exponent bits, 见 p63)	
has_denorm_loss	准确度的遗失是以一个非标准化值 (denormalization) 而不是以一个不 精密的结果被侦测出来	
denorm_min()	最小的非标准化 (denormalized) 正 值	
traps	已实作出 Trapping	
tinyness_before	在舍/入 (rounding) 之前可侦测出 tinyness	

译注 1: 本表格“意义”栏中的诸多解释, 涉及数字表示法专业术语。译者这方面能力有限, 没把握正确译出。对于没有把握的名词, 皆保留英文, 望谅。

译注 2: 本表格在英文版中因分页关系而被切割为表 4.2 和表 4.3。此处合而为一。后续表格从表 4.4 开始继续编号。

下面是对于 float 型别的数值限定模板特殊化的一个完全实作版, 当然是和平台相依的。这里同时还给出了各成员的确切标记 (signatures):

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

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

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

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

```

static const int radix = 2;

inline static float epsilon() throw() {
    return 1.19209290E-07F;
}

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

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

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

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

```

注意，所有数据成员如果不是 `const`，便是 `static`，这么一来其值便可在编译期间确定。至于由函数所定义的成员，在某些编译器中恐怕无法在编译期间确定其值。因此，同一份目标代码 (object code) 虽然可以在不同的处理器上执行，可能会得出不同的浮点值。

`round_style` 的值列于表 4.4, `has_denorm` 的值列于表 4.5。 `has_denorm` 其实也许应该称为 `denorm_style` 更贴切，可惜并非如此。这是因为 C++ 标准化后期才决定将其原本的 `bool` 型别改变为枚举值 (enumerative value) 之故。不过你还是可以把 `has_denorm` 当成 `bool` 值来用：C++ *Standard* 保证，如果 `denorm_absent` 为 0，就等于 `false`，如果 `denorm_present` 为 1 而且 `denorm_indeterminate` 为 -1，那么两者都等于 `true`。因此你可以把 `has_denorm` 视为一个 `bool` 值，用以判

定某个型别是否允许所谓的 "denormalized values".

表 4.4 numeric_limits<> 的舍入 (round) 风格

舍/入 (round) 风格	意义
round_toward_zero	向零舍/入
round_to_nearest	向最接近的可表示值舍/入
round_toward_infinity	向正无限值舍/入
round_toward_neg_infinity	向负无限值舍/入
round_indeterminate	无法确定

表 4.5 numeric_limits<> 的 "denormalization style"

舍入风格	意义
denorm_absent	此型别不允许 "denormalized values"
denorm_present	此型别允许向最接近的可表示值做 denormalized values
denorm_indeterminate	无法确定

numeric_limits<> 使用范例

下面的例子展示某些型别极值的可能运用, 例如用来了解某个型别的最大值, 或确定 char 是否带正负号:

```
// util/limits1.cpp

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

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

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

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

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

程序的输出结果和执行平台有关，下面是其中一种可能：

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

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

is_signed(char): false

is_specialized(string): false
```

最后一行表示，型别 `string` 并没有定义数值极限。这是理所当然的，因为 `strings` 并非数值型别。正如本例所示，你可以对任何型别进行询问，无论它是否定义了极值。

4.4 辅助函数

算法程序库（定义于头文件 `<algorithm>`）内含三个辅助函数，一个用来在两值之中挑选较大者，另一个用来在两值之中挑选较小者，第三个用来交换两值。

4.4.1 挑选较小值和较大值

“在两物之间选择较大值和较小值”的函数，定义于 `<algorithm>`，如下所示：

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

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

如果两值相等，通常会返回第一值。不过你的程序最好不要依赖这一点。

上述两个函数还有另一个版本，接受一个额外的 `template` 参数作为“比较准则”：

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

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

作为“比较准则”的那个参数应该是个函数或仿函数 (functor, 将于 5.9 节, p124 介绍)，接受两个参数并进行比较：在某个指定规则下，判断第一参数是否小于第二参数，并返回判断结果。

下面这个例子示范如何传入特定的比较函数作为参数，以此方式来运用 `max()`：

```
// util/minmax1.cpp

#include <algorithm>
using namespace std;

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

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

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

注意，`min()`和`max()`都要求它们所接受的两个参数的型别必须一致。如果不一致，你将无法正确调用它：

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

不过你倒是可以明白地声明参数型别（这样也就确定了返回值的型别）：

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

4.4.2 两值互换

函数 `swap()` 用来交换两对象的值。其泛型化版本定义于 `<algorithm>`：

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

运用这个函数，你可以如此交换任意变量 *x* 和 *y* 的值：

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

当然啦，只有当 `swap()` 所依赖的 `copy` 构造操作和 `assignment` 操作行为存在时，这个调用才可能有效。

`swap()` 的最大优势在于，透过 `template specialization` (模板特化) 或 `function overloading` (函数重载)，我们可以为更复杂的型别提供特殊的实作版本；我们可以交换对象内部成员，不必劳师动众地反复赋值，这无疑将大大地节约时间。标准程序库中的所有容器 (6.1.2 节, p147) 以及 `strings` (11.2.8 节, p490) 都运用了这项技术。举个例子，有个简单容器，仅仅内含一个 `array` 和一个成员 (用来指示数组元素数量)，那么为它特别实作的 `swap()` 可以是这样：

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

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

你瞧，调用 `swap()` 而非透过反复赋值操作来交换两容器的值，会带来效率上的提升。对于你自己定义的类型，如果确实能够带来效率上的改善，你就应该义不容辞地为它提供 `swap()` 特化版本。

4.5 辅助性的“比较操作符” (Comparison Operators)

有四个 `template functions`，分别定义了 `!=`, `>`, `<=`, `>=` 四个比较操作符。它们都是利用操作符 `==` 和 `<` 完成的。这四个函数定义于 `<utility>`：

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

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

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

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

只需定义 `<` 和 `==` 操作符，你就可以使用它们。只要加上 `using namespace std::rel_ops`，上述四个比较操作符就自动获得了定义。例如：

```
#include <utility>

class X {
    ...
public:
```

```

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

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

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

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

```

注意，这些操作符都定义于 `std` 的次命名空间 (sub-namespace) `rel_ops` 中。之所以如此安排，是为了防止和用户 (可能) 定义的全局命名空间中的同类型操作符发生冲突。于是，就算你这样使用 `using directive`：

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

因而把 `std` 的全部标识符引入全局命名空间，也没问题。

另一方面，那些想向 `rel_ops` 借一臂之力的用户可以这么做：

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

于是四个新的操作符就轻松到手了，无需使用复杂的搜寻规则来引用它们。

某些实作版本采用两个不同的参数型别来定义上述 `template`：

```

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

```

这么做的好处是，两个操作数的型别可以不同（只要它们之间“可以比较”就行）。但这并非 C++ 标准程序库所支持的作法。所以，如果想占这个便宜，就得付出可移植性方面的代价。

4.6 头文件 <cstdlib> 和 <stdlib>

头文件 <cstdlib> 和 <stdlib> 和其 C 对应版本兼容，在 C++ 程序中经常用到。它们是 C 头文件 <stddef.h> 和 <stdlib.h> 的较新版本，定义了一些常用的常数、宏、型别和函数。

4.6.1 <cstdlib> 内的各种定义

表 4.6 列出头文件 <cstdlib> 的各个定义项。NULL 通常用来表明一个不指向任何对象的指针，其实就是 0（其型别可以是 int，也可以是 long）。注意，C 语言中的 NULL 通常定义为 (void*)0。在 C++ 中这并不正确，NULL 的型别必须是个整型别，否则你无法将 NULL 赋值给一个指针。这是因为 C++ 并没有定义从 void* 到任何其它型别的自动转型操作¹⁰。NULL 同时也定义于头文件 <stdio>, <stdlib>, <cstring>, <ctime>, <wchar>, <locale> 内。

表 4.6 <cstdlib> 中的定义项

标识符	意义
NULL	指针值，用来表示“未定义”或“无值”
size_t	一种无正负号的型别，用来表示大小（例如元素个数）
ptrdiff_t	一种带有正负号的型别，用来表示指针之间的距离
offsetof	表示一个成员在 struct 或 union 中的偏移量

4.6.2 <stdlib> 内的各种定义

表 4.7 列出头文件 <stdlib> 内最重要的一些定义。常数 EXIT_SUCCESS 和 EXIT_FAILURE 用来当做 exit() 的参数，也可以当做 main() 的返回值。

经由 atexit() 登录的函数，在程序正常退出时会依登录的相反次序被一一调用起来。无论是透过 exit() 退出或从 main() 尾部退出，都会如此，不传递任何参数。

¹⁰ 鉴于 NULL 型别有这些晦涩的问题，有人建议 C++ 程序中最好不要使用 NULL，最好直接使用 0 或用户自定义的（例如）NIL 常数。不过我还是使用 NULL，所以本书范例程序中你还是可以看到它的踪迹。

表 4.7 <cstdlib> 中的定义项

定义	意义
<code>exit(int status)</code>	退出（离开， <code>exit</code> ）程序（并清理 <code>static</code> 对象）
<code>EXIT_SUCCESS</code>	程序正常结束。
<code>EXIT_FAILURE</code>	程序不正常结束。
<code>abort()</code>	退出程序（在某些系统上可能导致崩溃）。
<code>atexit (void (*function)())</code>	退出（ <code>exit</code> ）程序时调用某些函数。

函数 `exit()` 和 `abort()` 可用来在任意地点终止程序运行，无需返回 `main()`：

- `exit()` 会销毁所有 `static` 对象，将所有缓冲区（buffer）清空（flushes），关闭所有 I/O 通道（channels），然后终止程序（之前会先调用经由 `atexit()` 登录的函数）。如果 `atexit()` 登录的函数抛出异常，就会调用 `terminate()`。
- `abort()` 会立刻终止函数，不做任何清理（clean up）工作。

这两个函数都不会销毁局部对象（local objects），因为堆栈辗转开展动作（stack unwinding）不会被执行起来。为确保所有局部对象的析构函数获得调用，你应该运用异常（exceptions）或正常返回机制，然后再由 `main()` 离开。

5

Standard Template Library

STL, 标准模板库

STL (标准模板库) 是 C++ 标准程序库的核心, 它深刻影响了标准程序库的整体结构。STL 是一个泛型 (generic) 程序库, 提供一系列软件方案, 利用先进、高效的算法来管理数据。程序员无须了解 STL 的原理, 便可享受数据结构和算法领域中的这一革新成果。从程序员的角度看来, STL 是由一些可适应不同需求的群集类别 (collection classes), 和一些能够在这些数据群集上运作的算法构成。STL 内的所有组件都由 templates (模板) 构成, 所以其元素可以是任意型别。更妙的是, STL 建立了一个架构, 在此架构下, 你可以提供其它群集类别或算法, 与现有的组件搭配, 共同运作。总之, STL 赋予 C++ 新的抽象层次。把 dynamic arrays (动态数组)、linked list (链表)、binary trees (二叉树) 之类的东西抛开吧, 也不用再操心不同的搜寻算法了。你只需使用恰当的群集类别, 然后调用其成员函数和 (或) 算法来处理数据, 就万事大吉。当然, 如此的灵活性并非免费午餐, 代价总是有的。首要的一点是, STL 并不好懂。也正因为如此, 本书倾注了好几章篇幅, 为你讲解 STL 的内容。这一章介绍 STL 的总体概念, 探讨其使用技术。第一个范例展示如何使用 STL, 以及运用过程中有何考量。第 6 章至第 9 章详细讨论 STL 的各个组件 (包括容器 containers、迭代器 iterators、仿函数 functors、算法 algorithms), 并提供更多范例。

5.1 STL 组件 (STL Components)

若干精心勾画的组件共同合作, 构筑起 STL 的基础。这些组件中最关键的是容器、迭代器和算法。

- **容器 Containers**, 用来管理某类对象的集合。每一种容器都有其优点和缺点, 所以, 为了应付程序中的不同需求, STL 准备了不同的容器类型。容器可以是 arrays 或是 linked lists, 或者每个元素有一个特别的键值 (key)。

- **迭代器 Iterators**, 用来在一个对象群集 (collection of objects) 的元素上进行遍历动作。这个对象群集或许是个容器, 或许是容器的一部分。迭代器的主要好处是, 为所有容器提供了一组很小的公共接口。利用这个接口, 某项操作 (operations) 就可以行进至群集内的下一个元素。至于如何做到, 当然取决于群集的内部结构。不论这个群集是 array 或 tree, 此一行进动作都能成功。为什么? 因为每一种容器都提供了自己的迭代器, 而这些迭代器了解该种容器的内部结构, 所以能够知道如何正确行进。

迭代器的接口和一般指针差不多, 以 `operator++` 累进, 以 `operator*` 提领所指之值。所以, 你可以把迭代器视为一种 smart pointer, 能够把“前进至下一个元素”的意图转换成合适的操作。

- **算法 Algorithms**, 用来处理群集内的元素。它们可以出于不同的目的而搜寻、排序、修改、使用那些元素。透过迭代器的协助, 我们只需撰写一次算法, 就可以将它应用于任意容器之上, 这是因为所有容器的迭代器都提供一致的接口。你还可以提供一些特殊的辅助性函数供算法调用, 从而获取更佳的灵活性。这样你就可以一方面运用标准算法, 一方面适应自己特殊或复杂的需求。例如, 你可以提供自己的搜寻准则或元素合并时的特殊操作。

STL 的基本观念就是将数据和操作分离。数据由容器类别加以管理, 操作则由可定制 (configurable) 的算法定义之。迭代器在两者之间充当粘合剂, 使任何算法都可以和任何容器交互运作 (图 5.1)。

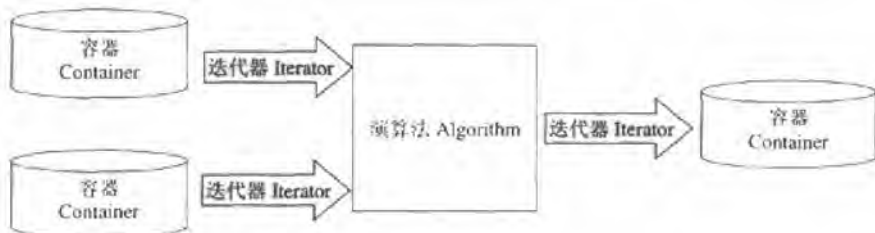


图 5.1 STL 组件之间的合作

STL 将数据和算法分开对待, 而不是合并考虑。因此从某种意义上说, STL 的概念和面向对象程序设计 (OOP) 的最初思想是矛盾的。然而这么做有着很重要的原因。首先, 你可以将各种容器与各种算法结合起来, 在很小的框架 (framework) 内拥有非常大的弹性。

STL 的一个根本特性是, 所有组件都可以针对任意型别 (types) 运作。顾名思义, 所谓 standard template library 表示其内的所有组件都是“可接受任意型别”的

templates, 前提是这些型别必须能够执行必要操作。因此 STL 成了泛型编程 (*generic programming*) 概念下的一个出色范例。容器和算法对任意型别 (types) 和类别 (classes) 而言, 都已经被一般化了。

STL 甚至提供更泛型化的组件。通过特定的配接器 (*adapters*) 和仿函数 (*functors*, 或称 *function objects*), 你可以补充、约束或订制算法, 以满足特别需求。目前说这些似乎为时太早, 眼下我还是先透过实例, 循序渐进地讲解概念, 这才是理解并熟悉 STL 的最佳方法。

5.2 容器 (Containers)

容器类别 (简称容器) 用来管理一组元素。为了适应不同需要, STL 提供了不同类型的容器, 如图 5.2。

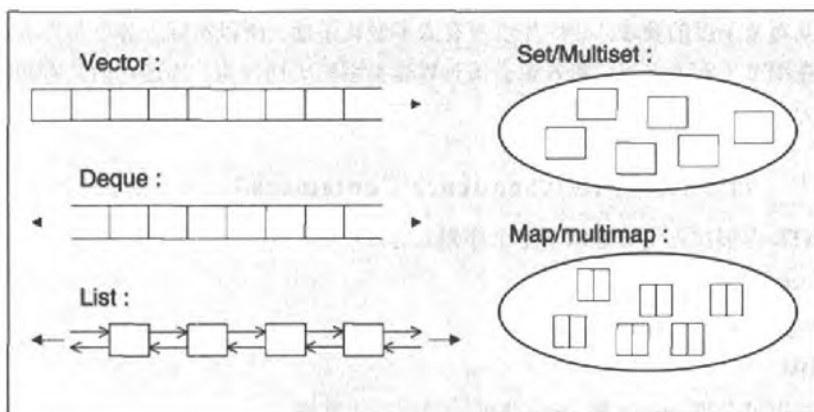


图 5.2 STL 的容器种类

总的来说, 容器可分为两类:

1. 序列式容器 **Sequence containers**, 此乃可序 (*ordered*) 群集, 其中每个元素均有固定位置——取决于插入时机和地点, 和元素值无关。如果你以追加方式对一个群集置入六个元素, 它们的排列次序将和置入次序一致。STL 提供三个定义好的序列式容器: `vector`, `deque`, `list`。
2. 关联式容器 **Associative containers**, 此乃已序 (*sorted*) 群集, 元素位置取决于特定的排序准则。如果你将六个元素置入这样的群集中, 它们的位置取决于元素值, 和插入次序无关。STL 提供了四个关联式容器: `set`, `multiset`, `map`, `multimap`。

关联式容器也可被视为特殊的序列式容器, 因为已序 (*sorted*) 群集正是根据某个排序准则排列 (*ordered*) 而成。如果你曾经用过其它群集库, 例如 Smalltalk

和 NIHCL¹所提供者, 你可能已经估计到这一点。在那些程序库中, *sorted collections* 由 *ordered collections* 派生而来。不过请注意, STL 所提供的群集型别 (collection types) 彼此独立, 各自实现, 毫无关联 (译注: 意指其间并无 classes 继承关系)。

关联式容器自动对其元素排序, 这并不意味它们就是用来排序的。你也可以对序列式容器的元素加以手动排序。自动排序带来的主要优点是, 当你搜寻元素时, 可获得更佳效率。更明确地说你可以放心使用二分搜寻法 (binary search)。该算法具有对数 (logarithmic) 复杂度, 而非线性复杂度。这什么意思呢? 如果你想在 1000 个元素中搜寻某个元素, 平均而言只需 10 次比较, 而非 500 次比较 (参见 2.3 节, p21)。因此自动排序只不过是关联式容器的一个 (有用的) 副作用而已。

下面各小节详细讨论各种容器类别。其中特别讲解了容器的典型实现。严格说来, C++ *Standard* 并未定义某一种容器的具体实现。然而 C++ *Standard* 所规定的行为和其对复杂度的要求, 让库作者没有太多变化余地。所以实际上各个实作版本之间只在细节上有所差异。第 6 章会谈谈到容器类别的确切行为、描述它们共有和特有的能力、并详细分析其成员函数。

5.2.1 序列式容器 (Sequence Containers)

STL 内部预先定义好以下三个序列式容器:

- Vectors
- Deques
- Lists

此外你也可以将 strings 和 array 当做一种序列式容器。

Vectors

Vector 将其元素置于一个 dynamic array 中加以管理。它允许随机存取, 也就是说你可以利用索引直接存取任何一个元素。在 array 尾部附加元素或移除元素均非常快速², 但是在 array 中部或头部安插元素就比较费时, 因为, 为了保持原本的相对次序, 安插点之后的所有元素都必须移动, 挪出位子来。

¹ The National Institute of Health's Class Library, 是最早的 C++ 类库之一。

² 严格说来, 元素追加动作是一种“分摊后的 (amortized)”高速。单一附加动作可能是缓慢的, 因为 vector 可能需要重新分配内存, 并将现有元素拷贝到新位置。不过这种事情不常发生, 所以总体看来这个操作十分迅速。见 p22 的复杂度讨论。

以下例子针对整数型别定义了一个 `vector`，插入 6 个元素，然后打印所有元素：

```
// stl/vector1.cpp

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

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

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

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

其中的：

```
#include <vector>
```

含入 `vectors` 的头文件。

以下声明：

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

生成一个“元素型别为 `int`”的 `vector`。由于没有任何初始化参数，缺省构造函数就将它建构为空群集。

`push_back()` 函数可为容器附加元素：

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

所有序列式容器都提供有这个成员函数。

`size()` 成员函数返回容器中的元素个数：

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

所有容器类别都提供有这个函数。

你可以通过 subscript (下标) 操作符 [], 存取 vector 内的某个元素:

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

在这里, 元素被写至标准输出, 所以整个程序的输出是:

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

Deque

所谓 deque (发音类似 "check"³), 是 "double-ended queue" 的缩写。它是一个 dynamic array, 可以向两端发展, 因此不论在尾部或头部安插元素都十分迅速。在中间部分安插元素则比较费时, 因为必须移动其它元素。

以下例子声明了一个浮点数型别的 deque, 并在容器头部安插 1.1 至 6.6 共 6 个元素, 最后打印出所有元素。

```
// stl/deque1.cpp

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

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

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

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

³ 有时候 "deque" 听起来颇为类似 "hack", 不过这纯属巧合 ©

本例之中,

```
#include <deque>
```

含入 `deque` 的头文件。

下面这一句:

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

会产生一个空的浮点数群集。

`push_front()` 函数可以用来安插元素:

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

它会将元素安插于群集前端。注意, 这种安插方式造成的结果是, 元素排放次序与安插次序恰好相反, 因为每个元素都安插于上一个元素的前面。因此, 程序输出如下:

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

你也可以使用成员函数 `push_back()` 在 `deque` 尾端附加元素。`vector` 并未提供 `push_front()`, 因为其时间性能很差 (在 `vector` 头端安插一个元素, 需要移动全部元素)。一般而言, STL 容器只提供通常具备良好时间效能的成员函数 (所谓“良好”的时间效能, 通常意味具有常数复杂度或对数复杂度), 这可以防止程序员调用性能很差的函数。

Lists

`List` 由双向链表 (doubly linked list) 实作而成。这意味 `list` 内的每个元素都以一部分内存指示其前趋元素和后继元素。`List` 不提供随机存取, 因此如果你要存取第 10 个元素, 你必须沿着串链依次走过前 9 个元素。不过, 移动至下一个元素或前一个元素的行为, 可以在常数时间内完成。因此一般的元素存取动作会花费“线性时间” (平均距离和元素数量成比例)。这比 `vector` 和 `deque` 提供的“分摊性 (amortized)”常数时间, 性能差很多。

`List` 的优势是: 在任何位置上执行安插或删除动作都非常迅速, 因为只须改变链接 (links) 就行。这表示在 `list` 中间位置移动元素比在 `vector` 和 `deque` 快得多。

以下例子产生一个空 `list`, 准备放置字符, 然后将 'a' 至 'z' 的所有字符插入其中, 利用循环每次打印并移除群集的第一个元素, 从而打印出所有元素:

```
// stl/list1.cpp
```

```
#include <iostream>
```

```
#include <list>
```



```
using namespace std;

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

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

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

就像先前的例子一样，头文件 `<list>` 内含 `lists` 的声明。以下定义一个“元素型别为字符”的 `list`：

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

成员函数 `empty()` 的返回值告诉我们容器中是否还有元素。只要这个函数返回 `false`（也就是说容器内还有元素），循环就继续进行：

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

循环之内，成员函数 `front()` 会返回第一个元素：

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

`pop_front()` 函数会删除第一个元素：

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

注意，`pop_front()` 并不会返回被删除的元素，所以无法将上述两个语句合而为一。

程序的输出结果取决于所用字集。如果是 ASCII 字集，输出如下⁴：

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

当然，为了打印 list 的所有元素而“采用循环输出并删除第一个元素”的做法实在是很奇怪。通常你只需遍历所有元素即可。lists 并没有提供以 operator[] 直接存取元素的能力，因为 lists 不支持随机存取，如果采用 operator[] 会导致不良效能。运用迭代器也可以遍历并打印所有元素。介绍过迭代器后我会给一个例子。如果你等不及，请跳到 p84。

Strings

你也可以将 string 当做 STL 容器来使用。这里的 strings 是指 C++ string 类族系 (basic_string<>, string, wstring) 的对象，第 11 章对此有所介绍。Strings 跟 vectors 很相似，只不过其元素为字符。11.2.13 节, p497 对此有详细解说。

Arrays

另一种容器不是类别(class)，而是 C/C++ 语言核心所支持的一个型别(type)：具有静态大小或动态大小的 array。但 array 并非 STL 容器，它们并没有类似 size() 和 empty() 等成员函数。尽管如此，STL 的设计允许你针对 array 调用 STL 算法。当我们以 static arrays 作为初始化行 (initializer list) 时，这一点特别有用。

Array 的运用并无新意，面对 arrays 使用算法，才是新的议题。这个议题将在 6.7.2 节, p218 讨论。

值得注意的是，我们没有必要再直接编写 dynamic array 了。Vectors 已经具备了 dynamic array 的全部性质，并提供更安全更便捷的接口。详见 6.2.3 节, p155。

5.2.2 关联式容器 (Associative Containers)

关联式容器依据特定的排序准则，自动为其元素排序。排序准则以函数形式呈现，用来比较元素值 (value) 或元素键 (key)。缺省情况下以 operator< 进行比较，不过你也可以提供自己的比较函数，定义出不同的排序准则。

通常关联式容器由二叉树 (binary tree) 实作出来。在二叉树中，每个元素 (节点) 都有一个父节点和两个子节点：左子树的所有元素都比自己小，右子树的所有元素都比自己大。关联式容器的差别主要在于元素的类型以及处理重复元素时的方

⁴ 如果是 ASCII 以外的字集，输出结果可能包含非字母字符，甚至可能什么都没有 (如果 'z' 不大于 'a' 的话)。

式（态度）。

下面是 STL 中预先定义好的关联容器。由于访问其中元素需要用到迭代器（iterator），所以我推迟至 p87 讨论过迭代器后再举例子。

- **Sets**

Set 的内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复。

- **Multisets**

Multiset 和 set 相同，只不过它允许重复元素，也就是说 multiset 可包括多个数值相同的元素。

- **Maps**

Map 的元素都是“实值/键值”所形成的一个对组（*key/value pairs*）。每个元素有一个键，是排序准则的基础。每一个键只能出现一次，不允许重复。Map 可被视为关联式数组（*associative array*），也就是具有任意索引型别（*index type*）的数组（详见 p91）。

- **Multimaps**

Multimap 和 map 相同，但允许重复元素，也就是说 multimap 可包含多个键值（*key*）相同的元素。Multimap 可被当做“字典”（译注：*dictionary*，某种数据结构）使用。p209 有个范例。

所有关联式容器都有一个可供选择的 *template* 参数，指明排序准则。缺省采用 *operator<*。排序准则同时也用来测试互等性（*equality*）：如果两个元素都不小于对方，则两者被视为相等。

你可以将 set 视为一种特殊的 map：其元素实值就是键值。实际产品中，所有这些关联式容器通常都由二叉树（*binary tree*）实作而成。

5.2.3 容器配接器（Container Adapters）

除了以上数个根本的容器类别，为满足特殊需求，C++ 标准程序库还提供了一些特别的（并且预先定义好的）容器配接器，根据基本容器类别实作而成。包括：

- **Stacks**

名字说明了一切。Stack 容器对元素采取 LIFO（后进先出）管理策略。

- **Queues**

Queue 容器对元素采取 FIFO（先进先出）管理策略。也就是说，它是个普通的缓冲区（*buffer*）。

- **Priority Queues**

Priority Queue 容器中的元素可以拥有不同的优先权。所谓优先权，乃是基于程

序员提供的排序准则（缺省使用 `operator<`）而定义。Priority queue 的效果相当于这样一个 buffer：“下一元素永远是 queue 中优先级最高的元素”。如果同时有多个元素具备最高优先权，则其次序无明确定义。

5.3 迭代器 (Iterators)

迭代器是一个“可遍历 STL 容器内全部或部分元素”的对象。一个迭代器用来指出容器中的一个特定位置。基本操作如下：

- **Operator ***

返回当前位置上的元素值。如果该元素拥有成员，你可以透过迭代器，直接以 `operator->` 取用它们⁵。

- **Operator ++**

将迭代器前进至下一元素。大多数迭代器还可使用 `operator--` 退回到前一个元素。

- **Operators == 和 Operator !=**

判断两个迭代器是否指向同一位置。

- **Operator =**

为迭代器赋值（将其所指元素的位置赋值过去）。

这些操作和 C/C++ “操作 array 元素”时的指针接口一致。不同之处在于，迭代器是个所谓的 smart pointers，具有遍历复杂数据结构的能力。其下层运行机制取决于其所遍历的数据结构。因此，每一种容器型别都必须提供自己的迭代器。事实上每一种容器都将其迭代器以嵌套（nested）方式定义于内部。因此各种迭代器的接口相同，型别却不同。这直接导出了泛型程序设计的概念：所有操作行为都使用相同接口，虽然它们的型别不同。因此，你可以使用 templates 将泛型操作公式化，使之得以顺利运行那些“能够满足接口需求”的任何型别。

所有容器类别都提供一些成员函数，使我们得以获得迭代器并以之遍访所有元素。这些函数中最重要的是：

- **begin()**

返回一个迭代器，指向容器起始点，也就是第一元素（如果有的话）的位置。

- **end()**

返回一个迭代器，指向容器结束点。结束点在最后一个元素之后，这样的迭代器又称作“逾尾（past-the-end）”迭代器。

⁵ 某些老旧的 STL 环境并不对迭代器支持 `operator->`。

于是, `begin()` 和 `end()` 形成了一个半开区间 (half-open range), 从第一个元素开始, 到最后一个元素的下一位置结束 (图 5.3)。半开区间有两个优点:

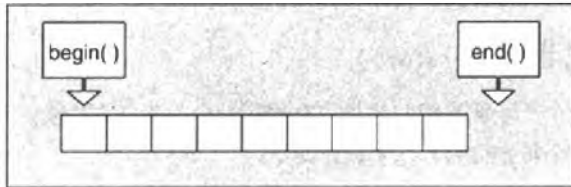


图 5.3 容器的 `begin()` 和 `end()` 成员函数

1. 为“遍历元素时, 循环的结束时机”提供了一个简单的判断依据。只要尚未到达 `end()`, 循环就可以继续进行。
2. 不必对空区间采取特殊处理手法。空区间的 `begin()` 就等于 `end()`。

下面这个例子展现了迭代器的用法, 将 `list` 容器内的所有元素打印出来 (这就是 p79 那个 `list` 实例的改进版)。

```
// stl/list2.cpp

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

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

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

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

首先产生一个 `list`，然后填入 'a' ~ 'z' 字符，然后在 `for` 循环中打印出所有元素：

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

迭代器 `pos` 声明于循环之前，其型别是“指向容器中的不可变元素”的迭代器：

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

任何一种容器都定义有两种迭代器型别：

1. `container::iterator`
这种迭代器以“读/写”模式遍历元素。
2. `container::const_iterator`
这种迭代器以“只读”模式遍历元素。

例如，在 `class list` 之中，它们的定义可能是这样：

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

至于其中 `iterator` 和 `const_iterator` 的确切型别，则于实作中定义。

在循环中，迭代器 `pos` 以容器的第一个元素位置为初值：

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

循环不断进行，直到 `pos` 到达容器的结束点：

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

在这里，`pos` 是在和“逾尾 (*past-the-end*)”迭代器作比较。当循环内部执行 `++pos` 语句，迭代器 `pos` 就会前进到下一个元素。

总而言之, `pos` 从第一个元素开始, 逐一访问了每一个元素, 直到抵达结束点为止 (图 5.4)。如果容器内没有任何元素, `coll.begin()` 等于 `coll.end()`, 循环根本不会执行。

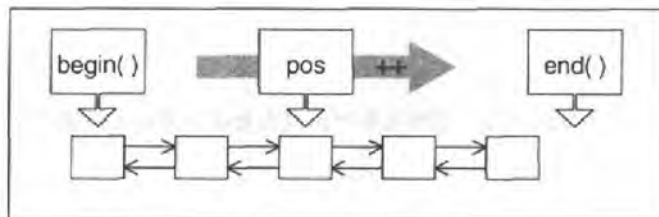


图 5.4 迭代器 `pos` 遍历 `list` 的每一个元素

在循环内部, 语句 `*pos` 代表当前 (current) 元素。本例将它输出之后, 又接着输出了一个空格。你不能改变元素内容, 因为 `pos` 是个 `const_iterator`, 从迭代器的观点看去, 元素是常量, 不能更改。不过如果你采用非常量 (nonconstant) 迭代器, 而且元素本身的型别也是非常量 (nonconstant), 那么就可以透过迭代器来改变元素值。例如:

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

注意, 这里使用“前置式递增 (preincrement)” `++pos`, 因为它比“后置式递增 (postincrement)” `pos++` 效率高。后者需要一个额外的临时对象, 因为它必须存放迭代器的原本位置并将它返回, 所以一般情况下最好使用 `++pos`, 不要用 `pos++`。也就是说, 不要这么写:

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

为了这个理由, 我建议优先采用前置式递增 (pre-increment) 或前置式递减 (pre-decrement) 操作符。

5.3.1 关联式容器的运用实例

上个例子中的迭代器循环可应用于任何容器, 只需调整迭代器型别即可。现在你知道如何打印关联式容器内的元素了吧。下面是使用关联式容器的一些例子。

Sets 和 Multisets 运用实例

第一个例子展示如何在 `set` 之中安插元素，并使用迭代器来打印它们。

```
// stl/set1.cpp

#include <iostream>
#include <set>

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

    IntSet coll; // set container for int values

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

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

一如以往，`include` 指令：

```
#include <set>
```

定义了 `sets` 的所有必要型别和操作。

既然容器的型别要用到好几次, 不妨先定义一个短一点的名字:

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

这个语句定义的 `IntSet` 型别, 其实就是“元素型别为 `int` 的一个 `set`”。这种型别有缺省的排序准则, 以 `operator<` 为依据, 对元素进行排序。这意味元素将以递增方式排列。如果希望以递减方式排列, 或是希望使用一个完全不同的排序准则, 你可以将该准则传入作为第二个 `template` 参数。下面例子即是元素以递减方式排列⁶:

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

以上所用的 `greater<>` 是一个预先定义的仿函数 (functor, or function object), 我将在 5.9.2 节, p131 讨论它。8.1.1 节, p294 另有一个例子, 仅使用元素的部分数据 (例如 ID) 进行排序。

所有关联式容器都提供一个 `insert()` 成员函数, 用以安插新元素:

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

新元素会根据排序准则自动安插到正确位置。注意, 你不能使用序列式容器的 `push_back()` 和 `push_front()` 函数, 它们在这里毫无意义, 因为你没有权力指定新元素的位置。

所有元素 (不论以任何次序) 安插完毕后, 容器的状态如图 5.5。元素以已序状态 (*sorted*) 存放于内部 *tree* 结构中。任何一个元素 (节点) 的左子树的所有元素, 永远小于右子树的所有元素 (这里的 “小于” 是指就当前排序准则而言)。Sets 不允许存在重复元素, 所以容器里头只有一个 “1”。

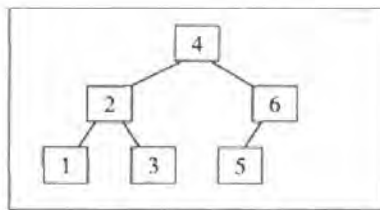


图 5.5 一个 Set, 拥有 6 个元素

⁶ 注意, 两个 “>” 符号之间一定要有一个空格。“>>” 会被编译器视为一个右移 (right-shift) 操作符, 从而导致语法错误。

现在，我们可以运用先前 `list` 例中所用的相同循环来打印 `set` 内的元素。以一个迭代器遍历全部元素，并逐一打印出来：

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

我要再一次提醒你，由于迭代器是容器定义的，所以无论容器内部结构如何复杂，它都知道如何行事。举个例子，如果迭代器指向第三个元素，操作符`++`便会将它移动到上端的第四个元素，再一次`++`，便会将它移动到下方第五个元素（图 5.6）。

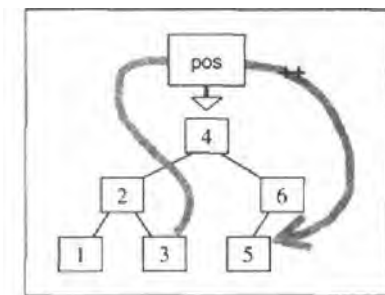


图 5.6 迭代器 `pos` 遍历 `Set` 内的元素

以下是输出结果：

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

如果你想使用 `multiset` 而不是 `set`，唯一需要改变的就是容器的型别（`set` 和 `multiset` 的定义被置于同一个头文件中）：

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

由于 `multiset` 允许元素重复存在，因此其中可包含两个数值皆为 1 的元素。输出结果如下：

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

Maps 和 Multimaps 的运用实例

`Map` 的元素是成对的键值/实值（`key/value`）。因此其声明、元素安插、元素存取皆和 `set` 有所不同。下面是一个 `multimap` 运用实例：

```
// stl/mmap1.cpp

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

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

    IntStringMMap coll; // container for int/string values

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

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

程序的输出结果可能是这样的:

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

不过由于 "this" 和 "is" 的键值相同, 两者的出现顺序也可能反过来。

拿这个例子和 p87 的 set 实例作比较，你会发现以下两点不同：

1. 这里的元素是成对的键值/实值 (*key/value pair*)，所以你必须首先生成这个 *pair*，再将它插入群集内部。辅助函数 `make_pair()` 正是为了这个目的而打造。这个问题的细节，以及其它安插方法，请见 p203，
2. 迭代器所指的是“键值/实值”对组 (*key/value pair*)，因此你无法一口气打印它们，你必须取出 *pair* 的成员，亦即所谓的 *first* 和 *second* (*pair* 型别在 4.1 节, p33 介绍过)。因此，以下语句：

```
pos->second
```

便取得了“键值/实值”对组中的第二部分，也就是 *multimap* 元素的实值 (*value*)。和一般指针的情形一样，上述语句就是以下语句的简写方案⁷：

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

同样道理，以下语句：

```
pos->first
```

取得“键值/实值”对组中的第一部分，也就是 *multimap* 元素的键值 (*key*)。

Multimaps 也可以用来作为 *dictionaries*，详见 p209 实例。

将 *Maps* 当做关联式数组 (*associative arrays*)

如果上述例子中以 *map* 取代 *multimap*，输出结果就不会有重复键值 (*keys*)，实值 (*values*) 则和上述结果一样。一个“键值/实值”对组所形成的群集中，如果所有键值都是独一无二的，我们可将它视为一个关联式数组 (*associative array*)。考虑以下例子：

```
// stl/map1.cpp

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

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

⁷ 某些老旧环境没有实现 `iterator->`，这时候你就只能使用第二个表达式了。

```

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

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

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

```

当我们声明容器型别的时候，必须同时指定键值 (*key*) 和实值 (*value*) 的型别：

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

Maps 允许你使用 `operator[]` 安插元素：

```

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

```

在这里，以键值为索引，键值可为任意型别。这正是关联式数组的接口。所谓关联式数组就是：索引可以采用任何型别。

注意这里的 subscript (下标) 操作符和一般 `array` 所用的行为有些不同：在这里，索引可以不对应于任何元素。如果你指定了一个新索引 (新键值)，会导致产生一个对应的新元素，并被安插于 `map`。也就是说，没有任何索引是“错误”的。因此，以下语句：

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

其中的式子：

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

会产生一个新元素，键值为 "Null"。然后 `assignment` (赋值) 操作符再将该元素的实值设为 0 (并转化为 `float`)。6.6.3 节, p205 更详细地讨论了如何将 `maps` 当做关

联式数组。

Multimaps 不允许我们使用 subscript (下标) 操作符, 因为 multimaps 允许单一索引对应到多个不同元素, 而下标操作符却只能处理单一实值。你必须先产生一个“键值/实值”对组, 然后再插入 multimap, 见 p90。当然, 对于 maps 也可以这么做, 细节请参考 p202。

存取 multimaps 或 maps 的元素时, 你必须透过 pair 结构的 first 成员和 second 成员, 才能取得键值 (key) 和实值 (value)。上述程序的输出如下:

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

5.3.2 迭代器分类 (Iterator Categories)

除了基本操作之外, 迭代器还有其它能力。这些能力取决于容器的内部结构。STL 总是只提供效率上比较出色的操作, 因此, 如果容器允许随机存取 (例如 vectors 或 deque), 那么它们的迭代器也能进行随机操作 (例如直接让迭代器指向第五元素)。

根据能力的不同, 迭代器被划分为五种不同类属。STL 预先定义好的所有容器, 其迭代器均属于以下两种类型:

1. 双向迭代器 (Bidirectional iterator)

顾名思义, 双向迭代器可以双向行进: 以递增 (increment) 运算前进或以递减 (decrement) 运算后退。list、set、multiset、map 和 multimap 这些容器所提供的迭代器都属此类。

2. 随机存取迭代器 (Random access iterator)

随机存取迭代器不但具备双向迭代器的所有属性, 还具备随机访问能力。更明确地说, 它们提供了“迭代器算术运算”必要的操作符 (和“一般指针的算术运算”完全对应)。你可以对迭代器增加或减少一个偏移量、处理迭代器之间的距离、或是使用 < 和 > 之类的 relational (相对关系) 操作符来比较两个迭代器。vector、deque 和 strings 所提供的迭代器都属此类。

其它迭代器类型在 7.2 节, p251 介绍。

为了撰写尽可能与容器型别无关的泛型程序代码, 你最好不要使用随机存取迭代器 (random access iterators) 的特有操作。例如以下例子, 可以在任何容器上运作:

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

而下面这样的程序代码就不是所有容器都适用了:

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

两者的唯一区别在于测试循环条件时, 第二例使用 `operator<`, 第一例使用 `operator!=`。要知道, 只有 random access iterators 才支持 `operator<`, 所以第二例中的循环对于 `lists`、`sets` 和 `maps` 无法运作。为了写出适用于任何容器的泛型程序代码, 你应该使用 `operator!=` 而非 `operator<`。不过如此一来, 程序代码的安全性可能有损, 因为如果 `pos` 的位置在 `end()` 的后面, 你未必便能发现 (关于 STL 使用上的可能错误, 请见 5.11 节, p136)。究竟使用哪种方式, 取决于当时情况, 取决于个人经验, 取决于你。

为了避免误解, 我再强调一句。注意, 我说的是类属、分类 (categories), 不是类别 (classes)。所谓类属, 只是定义迭代器的能力, 无关乎迭代器的型别 (type)。STL 的泛型概念可以以纯抽象形式工作, 也就是说, 任何东西只要行为“像”一个双向迭代器, 那么它就是一个双向迭代器。

5.4 算法 (Algorithms)

为了处理容器内的元素, STL 提供了一些标准算法, 包括搜寻、排序、拷贝、重新排序、修改、数值运算等十分基本而普遍的算法。

算法并非容器类别的成员函数, 而是一种搭配迭代器使用的全局函数。这么做 (译注: 意指搭配迭代器来使用) 有一个重要优势: 所有算法只需实作出一份, 就可以对所有容器运作, 不必为每一种容器量身订制。算法甚至可以操作不同型别 (types) 之容器内的元素, 也可以与用户定义的容器搭配。这个概念大幅降低了程序代码的体积, 提高了程序库的能力和弹性。

注意, 这里所阐述的并非面向对象思维模式 (OOP paradigm), 而是泛型函数式编程思维模式 (generic functional programming paradigm)。在面向对象编程 (OOP) 概念里, 数据与操作合为一体, 在这里则被明确划分开来, 再透过特定的接口彼此互动。当然这需要付出代价: 首先, 用法有失直观, 其次, 某些数据结构和算法之间并不兼容。更有甚者, 某些容器和算法虽然勉强兼容, 却毫无用处 (也许导致很糟的效能)。因此, 深入学习 STL 的概念并了解其缺陷, 显得十分重要, 惟其如此, 方能取其利而避其害。我将在本章剩余篇幅中, 通过实例详细介绍它们。让我们从简单的 STL 算法的运用入手。以下实例展现了某些算法的使用方式:

```
// stl/alg01.cpp

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

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

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

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

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

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

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


为了调用算法, 首先你必须舍入头文件 `<algorithm>`:

```
#include <algorithm>
```

最先出现的是算法 `min_element()` 和 `max_element()`。调用它们时, 你必须传入两个参数, 定义出欲处理的元素范围。如果想要处理容器内的所有元素, 可以使用 `begin()` 和 `end()`。两个算法都返回一个迭代器, 分别指向最小或最大元素。因此, 以下语句:

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

算法 `min_element()` 返回最小元素的位置 (如果最小元素不只一个, 则返回第一个最小元素的位置)。以下语句印出该元素:

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

当然, 你也可以合并上述两个动作于单一语句:

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

接下来的算法是 `sort()`。顾名思义, 它将“由两个参数设定出来”的区间内的所有元素加以排序。你可以 (选择性地) 传入一个排序准则; 缺省的是 `operator <`。因此, 本例容器内的所有元素以递增方式排列。

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

排序后的容器元素排列如下:

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

再来便是算法 `find()`。它在给定范围中搜寻某个值。本例在整个容器内寻找第一个数值为 3 的元素。

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

如果 `find()` 成功了, 便返回一个迭代器, 指向目标元素。如果失败, 返回一个“逾尾 (past-the-end)”迭代器, 亦即 `find()` 所接受的第二参数。本例在第三个元素位置上发现数值 3, 因此完成后 `pos` 指向 `coll` 的第三个位置。

本例所展示的最后一个算法是 `reverse()`, 将区间内的元素反转放置:

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

于是第三个至最后一个元素之间的所有元素都被反转置放。整个程序输出如下:

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

5.4.1 区间 (Ranges)

所有算法都用来处理一个或多个区间内的元素。这样的区间可以(但非强行要求)涵盖容器内的全部元素。因此,为了得以操作容器元素的某个子集,我们必须将区间首尾当做两个参数 (arguments) 传给算法,而不是一口气把整个容器传递进去。

这样的接口灵活又危险。调用者必须确保经由两参数定义出来的区间是有效的 (valid)。所谓有效就是,从起点出发,逐一前进,能够到达终点。也就是说,程序员自己必须确保两个迭代器隶属同一容器,而且前后放置正确。否则结果难料,可能会引起无限循环,也可能会存取到内存禁区。就此点而言,迭代器就像一般指针一样危险。不过请注意,所谓“结果难料”(或说行为未有定义, *undefined behavior*) 意味任何 STL 实作品均可自由选择合适的方式来处理此类错误。稍后你会发现,确保区间的有效性并不像听起来那么简单。与此相关的一些细节请参见 5.11 节, p136。所有算法处理的都是半开区间 (half-open ranges) —— 包括起始元素位置但不包括结尾元素位置。传统的数学表示方式为:

[begin, end)

或

[begin, end[

本书采用第一种表示法。

半开区间的优点已于 p84 介绍过 (主要是单纯,可避免对空群集作另外特殊处理)。当然,金无足赤,世上亦没有完美的设计。请看下面的例子;

```
// stl/find1.cpp

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

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

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

```

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

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

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

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

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

```

本例首先以 20 至 40 的整数作为容器初值。当搜寻元素值 3 的任务失败后，`find()` 返回区间的结束位置（本例为 `coll.end()`）并赋值给 `pos`。以此 `pos` 作为稍后调用 `reverse()` 时的区间起点，纯粹是空摆架子，因为其结果相当于：

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

这其实就是在逆转一个空区间，当然毫无效果了（亦即所谓的 “no-op”）。

如果使用 `find()` 来获取某个子集的第一个和最后一个元素，你必须考虑一点：半开区间并不包含最后一个元素。所以上述例子第一次调用 `max_element()`：

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

返回的是 34，而不是 35：

```
max: 34
```

为了处理最后一个元素，你必须把该元素的下一个位置传递给算法：

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

这样才能得到正确的结果：

```
max: 35
```

注意，本例使用的是 `list` 容器，所以你能以 `++` 取得 `pos35` 的下一个位置。如果面对的是 `vectors` 或 `deque`s 的随机存取迭代器 (random access iterators)，你可以写 `pos35 + 1`。这是因为随机存取迭代器允许“迭代器算术运算” (参见 p93, 5.3.2 节, p255, 7.2.5 节)。

当然，你可以使用 `pos25` 和 `pos35` 来搜寻其间的任何东西。记住，为了让搜寻动作及于 `pos35`，必须将元素 35 的下一位置传入，例如：

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

本节中的所有例子都可以正常运作，但那完全是因为你很清楚 `pos25` 一定在 `pos35` 之前。否则，`[pos25; pos35)` 就不是个有效区间。如果你对于“哪个元素在前，哪个元素在后”心中没谱儿，事情可就麻烦了，说不定会导致未定义行为。

现在假设你并不知道元素 25 和元素 35 的前后关系，甚至连它们是否存在也心存疑虑。如果你手上用的是随机存取迭代器 (random access iterators)，则可以使用 `operator<` 进行检查：

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



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

这里使用了一个特别的表达式作为搜寻规则，其目的是找到数值 25 或数值 35 第一次出现的位置。这个表达式由好几个预先定义的仿函数 (functors, 或名 function objects) 组成，我将在 5.9.2 节, p131 和 8.2 节, p305 介绍所有预先定义的仿函数。`compose_f_gx_hx` 是个灵巧的辅助型仿函数，我将在 8.3.1 节, p316 介绍它。

5.4.2 处理多个区间

有数个算法可以 (或说需要) 同时处理多个区间。通常你必须设定第一个区间的起点和终点，至于其它区间，你只需设定起点即可，终点通常可由第一区间的元素数量推导出来。下面例子中，`equal()` 从头开始逐一比较 `coll1` 和 `coll2` 的所有元素：

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

因此，`coll2` 之中参与比较的元素数量，间接取决于 `coll1` 内的元素数量。

这使我们导出一个重要心得：如果某个算法用来处理多个区间，那么当你调用它时，务必确保第二（以及其它）区间所拥有的元素个数，至少和第一区间内的元素个数相同。特别是，执行涂写动作时，务必确保目标区间（destination ranges）够大。

考虑下面这个程序：

```
// stl/copy1.cpp

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

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

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

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

这里调用了 `copy()` 算法，将第一区间内的全部元素拷贝至目标区间。如上所述，第一区间的起点和终点都已指定，第二区间只指出起点。然而，由于该算法执行的是覆写动作（overwrites）而非安插操作（inserts），所以目标区间必须拥有足够的元素来被覆写，否则就会像这个例子一样，导致未定义的行为。如果目标区间内没有足够的元素供覆写，通常意味你会覆写 `coll2.end()` 之后的任何东西，幸运的话你的程序立即崩溃——这起码还能让你知道出错了。你可以强制自己获得这种幸运：即使用 STL 安全版本。在这个安全版本中，所有未定义的行为都会被导向一个错误处理程序（error handling procedure）。请参考 5.11.1 节，p138。

要想避免上述错误，你可以 (1) 确认目标区间内有足够的元素空间，或是 (2) 采用 insert iterators。Insert iterators 将在 5.5.1 节，p104 介绍。我首先解释如何修改目标区间，从而使它有足够的空间。

要想让目标区间够大，你要一开始就给它一个正确大小，要不就显式地改变其大小。这两个办法都只适用于序列式容器 (vectors, deque, lists)。关联式容器根本不会有此问题，因为关联式容器不可能被当做覆写型算法的操作目标 (原因见 5.6.2 节，p115)。以下例子展示如何增加容器的大小：

```
// stl/copy2.cpp

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

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

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

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

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

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



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

在这里, `resize()` 的作用是改变 `coll2` 的元素个数:

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

`coll3` 则是在初始化时就指明要有足够空间, 以容纳 `coll1` 中的全部元素:

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

注意, 这两种方法都会产出新元素并赋予初值。这些元素由 `default` 构造函数初始化, 没有任何参数。你可以传递额外的参数给构造函数和 `resize()`, 这样就可以按你的意愿将新元素初始化。

5.5 迭代器 之 配接器 (Iterator Adapters)

迭代器 (Iterators) 是一个纯粹抽象概念: 任何东西, 只要其行为类似迭代器, 它就是一个迭代器。因此, 你可以撰写一些类别 (classes), 具备迭代器接口, 但有着各不相同的行为。C++ 标准程序库提供了数个预先定义的特殊迭代器, 亦即所谓迭代器配接器 (iterator adapters)。它们不仅起辅助作用, 还能赋予整个迭代器抽象概念更强大的能力。

以下数小节简介三种迭代器配接器 (iterator adapters):

1. Insert Iterators (安插型迭代器)
2. Stream Iterators (流迭代器)
3. Reverse Iterators (逆向迭代器)

第 7.4 节, p264 会对它们作更详实的讲解。

5.5.1 Insert Iterators (安插型迭代器)

迭代器配接器的第一个例子是 `insert iterators`, 或称为 `inserters`。`Inserters` 可以使算法以安插 (`insert`) 方式而非覆写 (`overwrite`) 方式运作。使用它, 可以解决算法的“目标空间不足”问题。是的, 它会促使目标区间的大小按需要成长。

`Insert iterators` 内部将接口做了新的定义:

- 如果你对某个元素设值 (`assign`), 会引发“对其所属群集的安插 (`insert`) 操作”。至于插入位置是在容器的最前或最后, 或是于某特定位置上, 须视三种不同的 `insert iterators` 而定。
- “单步前进 (`step forward`)”不会造成任何动静 (是一个 `no-op`)。

现在请看下面这个例子:

```
// stl/copy3.cpp

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

int main()
{
    list<int> coll1;

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

    // copy the elements of coll1 into coll2 by appending them
    vector<int> coll2;
    copy (coll1.begin(), coll1.end(),          // source
          back_inserter(coll2));               // destination

    // copy the elements of coll1 into coll3 by inserting them at the
    // front - reverses the order of the elements
    deque<int> coll3;
    copy (coll1.begin(), coll1.end(),          // source
          front_inserter(coll3));              // destination

    // copy elements of coll1 into coll4
    // - only inserter that works for associative collections
    set<int> coll4;
    copy (coll1.begin(), coll1.end(),          // source
          inserter(coll4,coll4.begin()));      // destination
}
```

此例运用了三种预先定义的 insert iterators:

1. Back inserters (安插于容器最尾端)

Back inserters 的内部调用 `push_back()`，在容器尾端插入元素（此即“追加”动作）。以下语句完成之后，`coll1` 的所有元素都会被附加到 `coll2` 中：

```
copy (coll1.begin(), coll1.end(), // source
      back_inserter(coll2));      // destination
```

当然，只有在提供有 `push_back()` 成员函数的容器中，back inserters 才能派上用场。在 C++ 标准程序库中，这样的容器有三：vector, deque, list。

2. Front inserters (安插于容器最前端)

Front inserters 的内部调用 `push_front()`，将元素安插于容器最前端。以下语句将 `coll1` 的所有元素插入 `coll3`：

```
copy (coll1.begin(), coll1.end(), // source
      front_inserter(coll3));      // destination
```

注意，这种动作逆转了被安插元素的次序。如果你先安插 1，再向前安插 2，那么 1 会排列在 2 的后面。

Front inserters 只能用于提供有 `push_front()` 成员函数的容器，在标准程序库中，这样的容器是 deque 和 list。

3. General inserters (一般性安插器)

这种一般性的 inserters，简称就叫 inserters，它的作用是将元素插入“初始化时接受之第二参数”所指位置的前方。Inserters 内部调用成员函数 `insert()`，并以新值和新位置作为参数。所有 STL 容器都提供有 `insert()` 成员函数，因此，这是唯一可用于关联式容器身上的一种预先定义好的 inserter。

等等，我不是说过，在关联式容器身上安插新元素时，不能指定其位置吗？它们的位置是由它们的值决定的啊！好，我解释一下，很简单：在关联式容器中，你所给的位置只是一个提示，帮助它确定从什么地方开始搜寻正确的安插位置。如果提示不正确，效率上的表现会比“没有提示”更糟糕。7.5.2 节，p288 介绍了一个用户自定的 inserter，对关联式容器特别有用。

表 5.1 列出 insert iterators 的功能。7.4.2 节，p271 还会介绍更多细节。

表 5.1 预先定义的三种 Insert iterators

算式 (expression)	Inserter 种类
<code>back_inserter(container)</code>	使用 <code>push_back()</code> 在容器尾端安插元素，元素排列次序和安插次序相同。
<code>front_inserter(container)</code>	使用 <code>push_front()</code> 在容器前端安插元素，元素排列次序和安插次序相反。
<code>inserter(container, pos)</code>	使用 <code>insert()</code> 在 <code>pos</code> 位置上安插元素，元素排列次序和安插次序相同。

5.5.2 Stream Iterators (流迭代器)

另一种非常有用的迭代器配接器 (iterator adapter) 是 stream iterator, 这是一种用来读写 stream⁸的迭代器。它们提供了必要的抽象性, 使得来自键盘的输入像是个群集 (collection), 你能够从中读取内容。同样道理, 你也可以把一个算法的输出结果重新导向到某个文件或屏幕上。

下面是展示 STL 威力的一个典型例子。和一般 C 或 C++ 程序相比, 本例仅用数条语句, 就完成了大量复杂的工作:

```
// stl/iostreaml.cpp

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

int main()
{
    vector<string> coll;

    /* read all words from the standard input
     * - source: all strings until end-of-file (or error)
     * - destination: coll (inserting)
     */
    copy (istream_iterator<string>(cin),      // start of source
          istream_iterator<string>(),          // end of source
          back_inserter(coll));                // destination

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

    /* print all elements without duplicates
     * - source: coll
     * - destination: standard output (with newline between elements)
     */
    unique_copy (coll.begin(), coll.end(),      // source
                 ostream_iterator<string>(cout, "\n")); // destination
}
```

⁸ Stream (流) 是一个用来表现 I/O 通道的对象 (详见第 13 章)。

这个程序只用三个语句就完成了一系列工作：从标准输入读取所有输入文字、排序、将它们打印于屏幕。让我们逐一思考这三个语句。下面这个语句：

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

用到两个 input stream iterators:

1. istream_iterator<string>(cin)

这会产生一个可从“标准输入流 (standard input stream) cin”读取数据的 stream iterator⁹。其中的 template 参数 string 表示这个 stream iterator 专司读取该种型别的元素 (string 型别将在第 11 章介绍) 的职责。这些元素透过一般的 operator>> 被读取进来。因此每当算法企图处理下一个元素时, istream iterator 就会将这种企图转化为以下行动:

```
cin >> string
```

针对 string 而执行的 input 操作符通常读取以空白分隔的文字 (参见 p492), 因此上述算法的行为将是“逐词读取 (word-by-word)”。

2. istream_iterator<string>()

调用 istream iterators 的 default 构造函数, 产生一个代表“流结束符号” (end-of-stream) 的迭代器, 它代表的意义是: 你不能再从中读取任何东西。

只要不断逐一前进的那个第一参数不同于第二参数, 算法 copy() 就持续动作。这里的 end-of-stream 迭代器正是作为区间终点之用, 因此这个算法便从 cin 读取所有 strings, 直到读无可读为止 (可能是因为到达了 end-of-stream, 也可能是因为读入过程发生错误)。总而言之, 算法的数据来源是“来自 cin 的所有文字”。在 back_inserter 的协助下, 这些文字被拷贝并插入 coll 中。

接下来的 sort() 算法对所有元素进行排序:

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

最后, 下面这个语句:

```
unique_copy (coll.begin(), coll.end(),
             ostream_iterator<string>(cout, "\n"));
```

将其中所有元素拷贝到目的端 cout。处理过程中算法 unique_copy() 会消除毗邻的重复值。其中的表达式:

```
ostream_iterator<string>(cout, "\n")
```

⁹ 在某些老旧系统中, 你必须使用 ptrdiff_t 作为第二个模板参数, 才能产生出一个 istream iterator (参见 7.4.3 节, p280)。

会产生一个 `output stream iterator`，透过 `operator<<` 向 `cout` 写入 `strings`。`cout` 之后的第二参数（可有可无）被用来作为元素之间的分隔符。本例指定为一个换行符号，因此每个元素都被打印于独立的一行。

这个程序内的所有组件都是 `templates`，所以你可以轻易改变程序，对其它型别如整数或更复杂的对象进行排序。7.4.3 节, p277 对于 `iostream iterators` 进行了更详细的介绍，并附带更多实例。

本例使用一个声明和三个语句，对来自标准输入装置的所有文字（单词）进行排序。你还可以更进一步，只用一个声明和一个语句就搞定一切。详见 p228。

5.5.3 Reverse Iterators (逆向迭代器)

第三种预先定义的迭代器配接器 (`iterator adapters`) 就是 `reverse iterators`，此物像是倒转筋脉似地以逆向方式进行所有操作。它将 `Increment` (递增) 运算转换为 `decrement` (递减) 运算，反之亦然。所有容器都可以透过成员函数 `rbegin()` 和 `rend()` 产生出 `reverse iterators`。例如：

```
// stl/riter1.cpp

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

int main()
{
    vector<int> coll;

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

    // print all element in reverse order
    copy (coll.rbegin(), coll.rend(),           // source
          ostream_iterator<int>(cout, " ");    // destination
    cout << endl;
}
```

其中的表达式：

```
coll.rbegin()
```

返回一个由 `coll` 定义的 `reverse iterator`。这个迭代器可作为“对群集 `coll` 的元素逆向遍历”的起点。它指向群集的结尾位置（也就是最后元素的下一位置）。因此，表达式：

```
*coll.rbegin()
```

返回的是最后一个元素的值。

对应地，表达式：

```
coll.rend()
```

返回的 `reverse iterator`，可作为“对群集 `coll` 的元素逆向遍历”的终点。它也是指向“逾尾”（`past-the-end`）位置，只不过方向相反，指的是容器内第一个元素的前一个位置。

以下表达式没有定义：

```
*coll.rend()
```

同样情况，以下表达式也没有定义：

```
*coll.end()
```

注意，当某个位置上并无合法元素时，永远不要使用 `operator*` 或 `operator->`。

如果采用 `reverse iterators`，所有算法便可以不需特殊处理就以相反方向操作容器，这自然是美事一桩。使用 `operator++` 前进至下一元素，被转化为使用 `operator--` 后退至前一元素。本例中的 `copy()`，“从尾到头”地遍历所有 `coll` 元素。程序输出如下：

```
9 8 7 6 5 4 3 2 1
```

你可以将一般迭代器转换为 `reverse iterators`，反之亦可。然而，对于具体某个迭代器而言，这样的转换会改变其所指对象。这些细节在第 7.4.1 节, p264 介绍。

5.6 更易型算法 (Manipulating Algorithms)

译注：根据实质意义，我不把 `manipulating algorithms` 译为“操控型”算法。`manipulating algorithms` 是指会“删除或重排或修改元素”的算法，见 p115。该页亦出现另一个相同意义的术语：`modifying algorithms`。有些书籍（例如 *Generic Programming and the STL*）采用 `mutating algorithms` 一词。为此，我将这些相同意义的术语都译为“更易型”或“变动型”算法。我亦曾在某些书中采用“质变算法”一词。

某些算法会变更目标区间的内容，甚至会删除元素。一旦这种情况出现，请务必注意几个特殊问题。本节将对此做出解释。它们确实令人讶异，并体现了 STL“为了将容器和算法分离，以获取灵活性”而付出的代价。

5.6.1 移除 (Removing) 元素

算法 `remove()` 自某个区间删除元素。然而如果你用它来删除容器中的所有元素，其行为肯定会让你吃惊。例如：

```
// stl/remove1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // print all elements of the collection
    cout << "pre: ";
    copy (coll.begin(), coll.end(),           // source
          ostream_iterator<int>(cout, " ")); // destination
    cout << endl;

    // remove all elements with value 3
    remove (coll.begin(), coll.end(),         // range
            3);                               // value
```



```

// print all elements of the collection
cout << "post: ";
copy (coll.begin(), coll.end(),           // source
      ostream_iterator<int>(cout, " ")); // destination
cout << endl;
}

```

缺乏 STL 深层认识的人,看了这程序,必然认为所有数值为 3 的元素都会从群集中被移除。然而,程序的输出却是这样:

```

pre: 6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6

```

啊呀, `remove()` 并没有改变群集中的元素数量。`end()` 返回的还是当初那个终点, `size()` 返回的还是当初那个大小。不过某些事情还是有了变化: 元素的次序改变了, 有些元素被删除掉了。数值为 3 的元素被其后的元素覆盖了 (图 5.7)。至于群集尾端那些未被覆盖的元素, 原封不动——但是从逻辑角度来说, 那些元素已经不属于这个群集了。

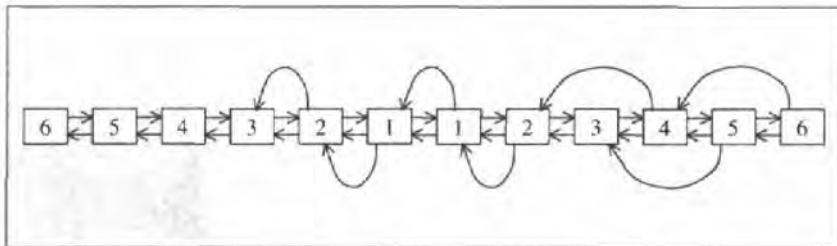


图 5.7 `remove()` 如何运作

事实上, 这个算法返回了一个新的终点。你可以利用该终点获得新区间、缩减后的容器大小, 或是获得被删除元素的个数。看看下面这个改进版本:

```

// stl/remove2.cpp

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

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6

```

```
for (int i=1; i<=6; ++i) {
    coll.push_front(i);
    coll.push_back(i);
}

// print all elements of the collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;

// remove all elements with value 3
// - retain new end
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                 3);

// print resulting elements of the collection
copy (coll.begin(), end,
      ostream_iterator<int>(cout, " "));
cout << endl;

// print number of resulting elements
cout << "number of removed elements: "
      << distance(end, coll.end()) << endl;

// remove "removed" elements
coll.erase (end, coll.end());

// print all elements of the modified collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

在这个版本中, `remove()` 的返回值被设定给 `end` 迭代器:

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                 3);
```

这个 `end` 正是“被修改之群集”经过元素移除操作后逻辑上的新终点。接下来你便可以拿它当做新的终点使用:

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout, " "));
```

另一种可能用法是，通过测定群集之“逻辑”终点和实际终点间的距离，获得“被删除元素”的数量：

```
cout << "number of removed elements: "  
      << distance(end, coll.end()) << endl;
```

在这里，针对迭代器而设计的辅助函数 `distance()` 发挥了作用。它的功用是返回两个迭代器之间的距离。如果这两个迭代器都是随机存取迭代器 (random access iterators)，你可以使用 `operator-` 直接计算其距离。不过本例所用的容器是 `list`，只提供双向迭代器 (bidirectional iterators)。关于 `distance()` 的细节，详见 7.3.2 节, p261¹⁰。

如果真想把那些被删除的元素斩草除根，你必须调用该容器的相应成员函数。容器所提供的成员函数 `erase()`，正适用于此目的。`erase()` 可以删除“参数所指示之区间”内的全部元素：

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

下面是整个程序的完整输出：

```
6 5 4 3 2 1 1 2 3 4 5 6  
6 5 4 2 1 1 2 4 5 6  
number of removed elements: 2  
6 5 4 2 1 1 2 4 5 6
```

如果你需要以单一语句来删除元素，可以如此这般：

```
coll.erase (remove(coll.begin(), coll.end(),  
                    3),  
            coll.end());
```

为何算法不自己调用 `erase()` 呢？哎，这个问题正好点出 STL 为了获取灵活性而付出的代价。透过“以迭代器为接口”，STL 将数据结构和算法分离开来。然而，迭代器只不过是“容器中某一位置”的抽象概念而已。一般来说，迭代器对自己所属的容器一无所知。任何“以迭代器访问容器元素”的算法，都不得（无法）透过迭代器调用容器类别所提供的任何成员函数。

这个设计导致一个重要结果：算法的操作对象不一定得是“容器内的全部元素”所形成的区间，而可以是那些元素的子集。甚至算法可运作于一个“并未提供成员函数 `erase()`”的容器上 (`array` 就是个例子)。所以，为了达成算法的最大弹性，不要求“迭代器必须了解其容器细节”还是很有道理的。

¹⁰ `distance()` 的定义有些变化。在 STL 旧版本中，为了使用它，你必须含入 `distance.hpp`，见 p263。

注意，通常并无必要删除那些“已被移除”的元素。通常，以逻辑终点来取代容器的实际终点，就足以应对。你可以以这个逻辑终点搭配任何算法演出。

5.6.2 更易型算法和关联式容器

更易型算法（指那些会移除 *remove*、重排 *resort*、修改 *modify* 元素的算法）用于关联式容器身上会出问题。关联式容器不能被当做操作目标，原因很简单：如果更易型算法用于关联式容器身上，会改变某位置上的值，进而破坏其已序 (*sorted*) 特性，那就推翻了关联式容器的基本原则：容器内的元素总是根据某个排序准则自动排序。因此，为了保证这个原则，关联式容器的所有迭代器均被声明为指向常量（不变量）。如果你更易关联式容器中的元素，会导致编译错误¹¹。

注意，这使你无法在关联式容器身上运用移除性 (*removing*) 算法，因为这类算法实际上悄悄更易了元素：“被移除元素”被其后的“未被移除元素”覆盖。

现在问题来了，如何从关联容器中删除元素？唔，很简单：调用它们的成员函数！每一种关联式容器都提供用以移除元素的成员函数。例如你可以调用 `erase()` 来移除元素：

```
// stl/remove3.cpp

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

int main()
{
    set<int> coll;

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

    // print all elements of the collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "));
}
```

¹¹ 糟糕的是，有些系统提供的错误处理能力令人不敢恭维：面对错误，你无法找出原因。有些编译器甚至连出错的源码都不列出来。希望这种状况在不久的将来获得改善。

```
cout << endl;

/* Remove all elements with value 3
 * - algorithm remove() does not work
 * - instead member function erase() works
 */
int num = coll.erase(3);

// print number of removed elements
cout << "number of removed elements: " << num << endl;

// print all elements of the modified collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

注意，容器类别提供了多个不同的 `erase()` 成员函数。其中一种形式是以“待删除之元素值”为唯一参数，它会返回被删除的元素个数（第 242 页）——当然，在禁止元素重复的容器中（例如 `sets` 和 `maps`），其返回值永远只能是 0 或 1。

本节范例程序输出如下：

```
1 2 3 4 5 6 7 8 9
number of removed elements: 1
1 2 4 5 6 7 8 9
```

5.6.3 算法 vs. 成员函数

就算我们符合种种条件，得以使用某个算法，那也未必就一定好。容器本身可能提供功能相似而性能更佳的成员函数。

一个极佳例子便是对 `list` 的元素调用 `remove()`。算法本身并不知道它工作于 `list` 身上，因此它在任何容器中都一样，做些四平八稳的工作：改变元素值，从而重新排列元素。如果它移除第一个元素，后面所有元素就会分别被设给各自的前一个元素。这就违反了 `lists` 的主要优点——通过修改链接（`links`）而非实值（`values`）来安插、移动、移除元素。

为了避免这么糟糕的表现，`list` 针对所有“更易型”算法提供了一些对应的成员函数。是的，如果你使用 `list`，你就应该使用这些成员函数。此外请注意，这些成员函数真的移除了“被移除”的元素（译注：而不像先前所说只是某种移动而已），如下例所示：

```
// stl/remove4.cpp

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

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // remove all elements with value 3
    // - poor performance
    coll.erase (remove(coll.begin(),coll.end(),
                       3),
               coll.end());

    // remove all elements with value 4
    // - good performance
    coll.remove (4);
}
```

如果高效率是你的最高目标，你应该永远优先选用成员函数。问题是你必须知道，某个容器确实存在有效率上明显突出的成员函数。面对 `list` 却使用 `remove()` 算法，你决不会收到任何警告讯息或错误通告。然而如果你决定使用成员函数，一旦换用另一种容器，就不得不更动程序代码。第 9 章的算法参考章节中，如果某个成员函数的性能优于某个算法，我会明白指出。

5.7 使用者自定义之泛型函数 (User-Defined Generic Functions)

STL 乃是一个可扩展的框架 (framework)。这意味你可以撰写自己的函数和算法，处理群集内的元素。当然，这些操作函数本身也可以是泛型的 (generic)。

为了在这些操作之中声明有效的迭代器，你必须使用容器提供的型别，因为每一种容器都有自己的迭代器。为了让我们方便写出真正的泛型函数，每一种容器都提供了一些内部的型别定义。请看下面的例子：

```
// stl/print.hpp

#include <iostream>

/* PRINT_ELEMENTS()
 * - prints optional C-string optcstr followed by
 * - all elements of the collection coll
 * - separated by spaces
 */
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

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

本例定义出一个泛型函数，可打印一个字符串（也可以不指定），然后打印容器的全部元素。以下声明：

```
typename T::const_iterator pos;
```

其中的 `pos` 被声明为“传入之容器型别”内的迭代器型别，关键词 `typename` 在此不可或缺，用以表明 `const_iterator` 是型别 `T` 所定义的一个型别，而不是一个型别为 `T` 的值（请见 p11 对 `typename` 的介绍）。

除了 `iterator` 和 `const_iterator`，容器还提供了其它（内部定义的）型别，帮助你写出泛型函数。例如它提供了元素型别（译注：即所谓 *value type*），以便在元素暂时拷贝场合中派上用场。详见 7.5.1 节, p285。

`PRINT_ELEMENTS` 的第二参数是个可有可无的前缀字，用来在打印时放在所有元素之前。你可以这样使用 `PRINT_ELEMENTS()`：

```
PRINT_ELEMENTS (coll, "all elements: ");
```

我之所以介绍这个函数，因为本书剩余部分会大量运用它来打印容器的所有元素。

5.8 以函数作为算法的参数

一些算法可以接受用户定义的辅助性函数，由此提高其灵活性和能力。这些函数将在算法内部被调用。

5.8.1 “以函数作为算法的参数”实例示范

最简单的例子莫过于 `for_each()` 算法了。它针对区间内的每一个元素，调用一个由用户指定的函数。下面是个例子：

```
// stl/foreach1.cpp

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

// function that prints the passed argument
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

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

    // print all elements
    for_each (coll.begin(), coll.end(),      // range
              print);                        // operation
    cout << endl;
}
```

这里的 `for_each()` 函数针对 `[coll.begin(), coll.end())` 区间内的每个元素调用 `print()` 函数。输出如下：

```
1 2 3 4 5 6 7 8 9
```


算法以数种态度来面对这些辅助函数：有的视之为可有可无，有的视之为必要。你可以利用它们来指定搜寻准则、排序准则、或定义某种操作，以便将某个容器内的元素转换至另一个容器。

下面是个运用实例：

```
// stl/transform1.cpp

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include "print.hpp"

int square (int value)
{
    return value*value;
}

int main()
{
    std::set<int> coll1;
    std::vector<int> coll2;

    // insert elements from 1 to 9 into coll1
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }
    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform each element from coll1 to coll2
    // - square transformed values
    std::transform (coll1.begin(),coll1.end(),    // source
                    std::back_inserter(coll2),    // destination
                    square);                      // operation

    PRINT_ELEMENTS(coll2,"squared: ");
}
```

此例之中，`square()`的作用是将 `coll1` 内的每一个元素予以平方运算，然后转移到 `coll2`（图 5.8）。输出如下：

```
initialized: 1 2 3 4 5 6 7 8 9  
squared: 1 4 9 16 25 36 49 64 81
```

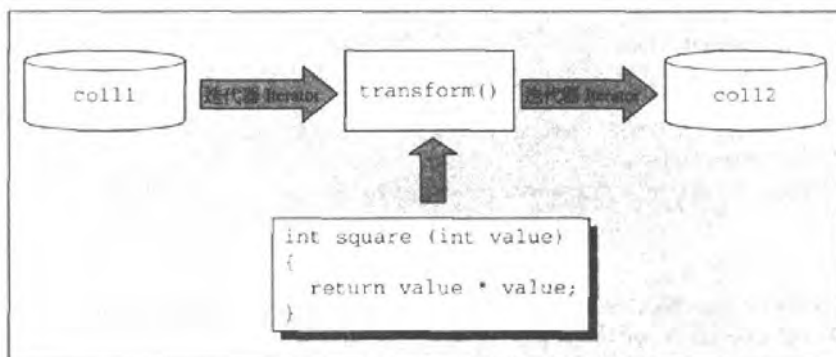


图 5.8 `transform()` 的运作方式

5.8.2 判断式 (Predicates)

算法有一种特殊的辅助函数叫做 `predicates`（判断式）。所谓 `predicates`，就是返回布尔值 (boolean) 的函数。它们通常被用来指定排序准则和搜寻准则。`Predicates` 可能有一个或两个操作数，视具体情形而定。注意，并非任何返回布尔值的一元函数或二元函数就是合法的 `predicate`。`STL` 要求，面对相同的值，`predicates` 必须得出相同的结果。这条戒律将那些“被调用时，会改变自己内部状态”的函数清除出场。细节请见 8.1.4 节，p302。

Unary Predicates（一元判断式）

`Unary predicates` 会检查唯一参数的某项特性。典型例子是像下面这样的函数，用来搜寻第一个质数：

```
// stl/prime1.cpp  
  
#include <iostream>  
#include <list>  
#include <algorithm>  
#include <cstdlib>    // for abs()  
using namespace std;
```

```
// predicate, which returns whether an integer is a prime number
bool isPrime (int number)
{
    // ignore negative sign
    number = abs(number);

    // 0 and 1 are prime numbers
    if (number == 0 || number == 1) {
        return true;
    }

    // find divisor that divides without a remainder
    int divisor;
    for (divisor = number/2; number%divisor != 0; --divisor) {
        ;
    }

    // if no divisor greater than 1 is found, it is a prime number
    return divisor == 1;
}

int main()
{
    list<int> coll;

    // insert elements from 24 to 30
    for (int i=24; i<=30; ++i) {
        coll.push_back(i);
    }

    // search for prime number
    list<int>::iterator pos;
    pos = find_if (coll.begin(), coll.end(), // range
                  isPrime); // predicate
    if (pos != coll.end()) {
        // found
        cout << *pos << " is first prime number found" << endl;
    }
    else {
```

```
        // not found
        cout << "no prime number found" << endl;
    }
}
```

在这个例子中，`find_if()` 算法在给定区间内寻找使“被传入之一元判断式（unary predicate）”运算结果为 `true` 的第一个元素。本例中的 `predicate` 是 `isPrime()` 函数，它会检查某数是否为质数。透过它，这个算法可以返回给定区间内的第一个质数。如果没有任何元素能够匹配这个（质数）条件，`find_if()` 算法就返回区间终点（也就是函数的第二参数）。本例中，24 到 30 之间确实存在一个质数，所以程序输出：

```
29 is first prime number found
```

Binary Predicates（二元判断式）

Binary predicates 的典型用途是，比较两个参数的特定属性。例如，为了依照你自己的原则对元素排序，你必须以一个简单的 `predicate` 形式提供这项原则。如果元素本身不支持 `operator<`，或如果你想使用不同的排序原则，这就派上用场了。

下面这个例子，根据每个人的姓名，对一组元素进行排序：

```
// stl/sort1.cpp

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

class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

/* binary function predicate:
 * - returns whether a person is less than another person
 */
```

```
bool personSortCriterion (const Person& p1, const Person& p2)
{
    /* a person is less than another person
    * - if the last name is less
    * - if the last name is equal and the first name is less
    */
    return p1.lastname()<p2.lastname() ||
        (!p2.lastname()<p1.lastname()) &&
        p1.firstname()<p2.firstname());
}

int main()
{
    deque<Person> coll;
    ...
    sort(coll.begin(),coll.end(), // range
        personSortCriterion);    // sort criterion
    ...
}
```

注意,你也可以使用仿函数 (functor, 或名 function object) 来实作一个排序准则。这种做法的优点是,制作出来的准则将是一个型别 (type), 可用来作为诸如“声明一个 set, 以某种型别为排序准则”之类的事情。详见 8.1.1 节, p294。

5.9 仿函数 (Functors, Function Objects)

译注: 本书英文版通篇采用的术语是 **function object**, 对应之译名为“函数对象”。这个词在 STL 发展初期曾经名为 **functor**, 取其音义, 我译为“仿函数”。考虑 STL 六大组件译名之整体性, 以及“术语最好具备独特性, 且不与其它名词混淆”的原则, 再考虑上下文阅读的顺畅性, 我认为“仿函数”较“函数对象”为佳。为此, 本中文版将 **function object** 全以 **functor** 取代, 并译为“仿函数”。

传递给算法的“函数型参数” (functional arguments), 并不一定得是函数, 可以是行为类似函数的对象。这种对象称为 **function object** (函数物件), 或称 **functor** (仿函数)。当一般函数使不上劲时, 你可以使用仿函数。STL 大量运用仿函数, 也提供 (预先定义) 了一些很有用的仿函数。

5.9.1 什么是仿函数

仿函数是泛型编程强大威力和纯粹抽象概念的又一个例证。你可以说, 任何东西, 只要其行为像函数, 它就是个函数。因此, 如果你定义了一个对象, 行为像函数, 它就可以被当做函数来用。

好，那么，什么才算是具备函数行为（也就是行为像个函数）？所谓函数行为，是指可以“使用小括号传递参数，藉以调用某个东西”。例如：

```
function(arg1,arg2); // a function call
```

如果你指望对象也可以如此这般，就必须让它们也有可能被“调用”——通过小括号的运用和参数的传递。没错，这是可能的（在 C++ 中，很少有什么是不可能的）。你只需定义 `operator()`，并给予合适的参数型别：

```
class X {
public:
    // define 'function call' operator
    return-value operator() (arguments) const;
    ...
};
```

现在，你可以把这个类别的对象当做函数来调用了：

```
X fo;
...
fo(arg1,arg2); // call operator {} for function object fo
```

上述调用等同于：

```
fo.operator() (arg1,arg2); // call operator {} for function object fo
```

下面是个完整例子，是先前 p119 范例的一个仿函数版本，其行为和使用一般函数（非仿函数）完全相同：

```
// stl/foreach2.cpp

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

// simple function object that prints the passed argument
class PrintInt {
public:
    void operator() (int elem) const {
        cout << elem << ' ';
    }
};

int main()
{
    vector<int> coll;
```

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

// print all elements
for_each (coll.begin(), coll.end(), // range
          PrintInt());              // operation
cout << endl;
}
```

PrintInt 所做的定义显示, 你可以对它的对象调用 operator(), 并传入一个 int 参数。至于语句:

```
for_each (coll.begin(), coll.end(),
          PrintInt());
```

其中的表达式:

```
PrintInt()
```

产生出此类别的一个临时对象, 当做 for_each() 算法的一个参数。for_each() 算法大致如下:

```
namespace std {
    template <class Iterator, class Operation>
    Operation for_each (Iterator act, Iterator end, Operation op)
    {
        while (act != end) {          // as long as not reached the end
            op(*act);                  // - call op() for actual element
            ++act;                     // - move iterator to the next element
        }
        return op;
    }
}
```

for_each() 使用暂时对象 op (一个仿函数), 针对每个元素调用 op(*act)。如果第三参数是个一般函数, 就以 *act 为参数调用之。如果第三参数是个仿函数, 则以 *act 为参数, 调用仿函数 op 的 operator()。因此, 本例之中, for_each() 调用:

```
PrintInt::operator()(*act)
```

你也许不以为然, 你也许认为仿函数看起来很怪异、令人讨厌、甚或毫无意义。的确, 它们带来更复杂的程序代码, 然而仿函数也有其过人之处, 比起一般函数, 它们有以下优点:

1. 仿函数是 "smart functions" (智能型函数)

“行为类似指针”的对象，我们称为 "smart pointers"。“行为类似函数”的对象呢？同样道理，我们可以称之为 "smart functions"，因为它们的能力可以超越 `operator()`。仿函数可拥有成员函数和成员变量，这意味仿函数拥有状态 (state)。事实上，在同一时间里，由某个仿函数所代表的单一函数，可能有不同的状态。这在一般函数中是不可能的。另一个好处是，你可以在执行期 (runtime) 初始化它们——当然必须在它们被使用（被调用）之前。

2. 每个仿函数都有自己的型别

一般函数，唯有在它们的标记式 (signatures) 不同时，才算型别不同。而仿函数即使标记式相同，也可以有不同的型别。事实上，由仿函数定义的每一个函数行为都有其自己的型别。这对于“利用 `template` 实现泛型程序编写”乃是一个卓越贡献，因为如此一来，我们便可以将函数行为当做 `template` 参数来运用。这使得不同型别的容器可以使用同类型的仿函数作为排序准则。这可以确保你不会在排序准则不同的群集 (collections) 之间赋值、合并或比较。你甚至可以设计仿函数继承体系 (functors hierarchies)，以此完成某些特别事情，例如在一个总体原则下确立某些特殊情况。

3. 仿函数通常比一般函数速度快

就 `template` 概念而言，由于更多细节在编译期就已确定，所以通常可能进行更好的最佳化。所以，传入一个仿函数（而非一般函数），可能获得更好的性能。

这一小节的剩余部分，我会给出数个例子，展示仿函数较之于一般函数的优势所在。第 8 章专攻仿函数，有更多例子和细节。尤其该章为你展示“以函数行为作为 `template` 参数”这一技术带给我们的利益。

假设你需要对群集 (collection) 中的每个元素加上一个固定值。如果你在编译期便确切知道这个固定数，你可以使用一般函数：

```
void add10 (int& elem)
{
    elem += 10;
}

void f1()
{
    vector<int> coll;
    ...
    for_each (coll.begin(), coll.end(),      // range
              add10);                        // operation
}
```


如果你需要数个不同的固定值，而它们在编译期都已确知，你可以使用 `template`:

```
template <int theValue>
void add (int& elem)
{
    elem += theValue;
}

void f1()
{
    vector<int> coll;
    ...
    for_each (coll.begin(), coll.end(),      // range
              add<10>);                      // operation
}
```

如果你必须在执行时期才处理这个数值，那就麻烦了。你必须在函数被调用之前先将这个数值传给该函数。这通常会导致产生一些全局变量，“算法的调用者”和“算法所调用的函数”都会用到它们。真是一团糟。

如果你两次用到该函数，每次加数不同，而都是在执行时期才处理，那么一般函数根本就无能为力。你要么传入一个标记 (tag)，要么干脆写两个函数。你是否有过这样的经历：握有一个函数，它有个 `static` 变量用以记录状态 (state)，而你需要这个函数在同一时间内有另一个不同状态 (state)？于是你只好拷贝整份函数定义，化为两个不同的函数。这正是先前所说的问题。

如果使用仿函数，你就可以写出“更机灵”的函数，遂你所愿。对象可以有自己的状态，可以被正确初始化。下面是一个完整例子¹²：

```
// stl/add1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

// function object that adds the value with which it is initialized
class AddValue {
```

¹² 辅助函数 `PRINT_ELEMENTS()` 已于 p118 介绍过。

```
private:
    int theValue; // the value to add
public:
    // constructor initializes the value to add
    AddValue(int v) : theValue(v) {
    }

    // the 'function call' for the element adds the value
    void operator() (int& elem) const {
        elem += theValue;
    }
};

int main()
{
    list<int> coll;

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

    PRINT_ELEMENTS(coll, "initialized: ");

    // add value 10 to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue(10));           // operation

    PRINT_ELEMENTS(coll, "after adding 10: ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue(*coll.begin())); // operation

    PRINT_ELEMENTS(coll, "after adding first element: ");
}
```

初始化之后，群集内含数值 1 至 9：

initialized: 1 2 3 4 5 6 7 8 9

第一次调用 `for_each()`，将每个数值加 10：

```
for_each (coll.begin(), coll.end(),    // range
          AddValue(10));               // operation
```

这里，表达式 `AddValue(10)` 生出一个 `AddValue` 物件，并以 10 为初值。`AddValue` 构造函数将这个值保存在成员 `theValue` 中。而在 `for_each()` 之内，针对 `coll` 的每一个元素调用 `{}`，实际上就是对传入的那个 `AddValue` 暂时对象调用 `operator()`，并以容器元素作为参数。仿函数（`AddValue` 对象）将每个元素加 10。结果如下：

```
after adding 10: 11 12 13 14 15 16 17 18 19
```

第二次调用 `for_each()` 亦采用相同机能，将第一元素值加到每个元素身上。首先使用第一元素值作为仿函数暂时对象的初值：

```
AddValue(*coll.begin())
```

最后结果如下：

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

p335 有这个例子的改进版，其中 `AddValue` 仿函数的型别被改为一个 `template`，可接纳不同的加数。

运用此项技术，先前所说的“一个函数、两个状态”的问题就可以用“两个不同的仿函数”加以解决。例如，你可以声明两个仿函数，然后各自运用：

```
AddValue addx(x); // function object that adds value x
AddValue addy(y); // function object that adds value y

for_each (coll.begin(), coll.end(), // add value x to each element
          addx);
...
for_each (coll.begin(), coll.end(), // add value y to each element
          addy);
...
for_each (coll.begin(), coll.end(), // add value x to each element
          addx);
```

同样道理，你也可以提供一些成员函数，在仿函数生命周期内查询或改变对象状态。

注意，C++ 标准程序库并未限制算法“对着一个容器元素”调用仿函数的次数，因此可能导致同一个仿函数有若干副本被传给元素。如果把仿函数当做判断式（`predicates`）使用，这个问题会惹来一身麻烦。8.1.4 节，p302 讨论了这个问题。

5.9.2 预先定义的仿函数

C++ 标准程序库包含了一些预先定义的仿函数, 涵盖许多基础运算。有了它们, 很多时候你就不必费心自己去写仿函数了。一个典型的例子是作为排序准则的仿函数。operator< 之缺省排序准则乃是 less<>, 所以, 如果你声明:

```
set<int> coll;
```

会被扩展为¹³:

```
set<int, less<int> > coll; // sort elements with <
```

既然如此, 想必你能猜到, 反向排列这些元素将不是什么难事¹⁴:

```
set<int, greater<int> > coll; // sort elements with >
```

类似情况, 还有许多仿函数用于数值处理。下例是将群集中的全部元素都设为相反值 (负值):

```
transform (coll.begin(), coll.end(), // source
           coll.begin(),             // destination
           negate<int>());           // operation
```

其中表达式:

```
negate<int>()
```

根据预先定义好的 `template class negate` 生成一个仿函数, 将传进来的 `int` 值设定为负。transform() 算法使用此一运算, 将第一群集的所有元素处理之后转移到第二群集。如果转移目的地就是自己, 那么这段程序代码就是“对群集内的每一个元素取负值”。

同样道理, 你也可以对群集内的所有元素求平方 (二次方)。

```
// process the square of all elements
transform (coll.begin(), coll.end(), // first source
           coll.begin(),             // second source
           coll.begin(),             // destination
           multiplies<int>());       // operation
```

这里运用了 transform() 算法的另一种形式, 以某种特定运算, 将两群集内的元素处理后的结果写入第三群集。由于本例的三个群集实际上是同一个, 所以其内

¹³ 有些系统并不支持 `default template arguments`, 那么你能只能使用后一种形式。

¹⁴ 注意, 两个 ">" 之间必须保留一个空格, 否则 ">>" 会被解析为右移 (right shift) 操作符, 因而发生语法错误。

的每个元素都被计算了平方值，并写进群集内，改写原有值¹⁵。

透过一些特殊的函数配接器 (function adaptors)，你还可以将预先定义的仿函数和其它数值组合在一起，或使用特殊状况。下面是一个完整范例：

```
// stl/fo1.cpp

#include <iostream>
#include <set>
#include <deque>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    set<int, greater<int> > coll1;
    deque<int> coll2;

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

    PRINT_ELEMENTS(coll1, "initialized: ");

    // transform all elements into coll2 by multiplying 10
    transform (coll1.begin(), coll1.end(),           // source
               back_inserter(coll2),                // destination
               bind2nd(multiplies<int>(), 10));      // operation

    PRINT_ELEMENTS(coll2, "transformed: ");

    // replace value equal to 70 with 42
    replace_if (coll2.begin(), coll2.end(),          // range
               bind2nd(equal_to<int>(), 70),         // replace criterion
               42);                                  // new value

    PRINT_ELEMENTS(coll2, "replaced: ");
```

¹⁵ STL 早期版本中，乘法运算的仿函数名为 `times`。但这和某些操作系统 (POSIX, X/Open) 中用以计算时间的函数名称冲突了，所以后来改为更清楚的名称：`multiplies`。

```

// remove all elements with values less than 50
coll2.erase(remove_if(coll2.begin(),coll2.end(), // range
    bind2nd(less<int>(),50)), // remove criterion
    coll2.end());

PRINT_ELEMENTS(coll2,"removed: ");
}

```

其中的语句:

```

transform (coll1.begin(),coll1.end(),          // source
    back_inserter(coll2),                      // destination
    bind2nd(multiplies<int>(),10)); // operation

```

将 coll1 内的所有元素乘以 10 后转移 (安插) 到 coll2 中。这里使用配接器 bind2nd, 使得进行 multiplies<int> 运算时, 以源群集 (source collection) 的元素作为第一参数, 10 作为第二参数。

配接器 bind2nd 的工作方式如下: transform() 期望它自己的第四参数是个能接纳单一参数 (也就是容器实际元素) 的表达式, 然而我们却希望先把该元素乘以 10, 再传给 transform()。所以我们必须构造出一个表达式, 接受两个参数, 并以数值 10 作为第二参数, 以此产生一个 “只需单参数” 的表达式。bind2nd() 正好胜任这项工作。它会把表达式保存起来, 把第二参数当做内部数值也保存起来。当算法以实际群集元素为参数, 调用 bind2nd 时, bind2nd 把该元素当做第一参数, 把原先保存下来的那个内部数值作为第二参数, 调用保留下来的那个表达式, 并返回结果。(译注: 这段繁复的文字说明可能解释效果不甚理想, 实际情况 (源码运作) 请看《STL 源码剖析》第 8 章, 侯捷著, 碁峰出版社 2002)

类似情况, 以下的:

```

replace_if (coll2.begin(),coll2.end(),          // range
    bind2nd(equal_to<int>(),70),                // replace criterion
    42);

```

其中的表达式:

```
bind2nd(equal_to<int>(),70)
```

被用来当做一项准则, 判断哪些元素将被 42 替代。bind2nd 以 70 作为第二参数, 调用二元判断式 (binary predicate) equal_to, 从而定义出一个一元判断式 (unary predicate), 处理群集内的每一个元素。

最后一个语句也一样:

```
bind2nd(less<int>(),50)
```

它被用来判断群集内的哪些元素应当被扫地出门。所有小于 50 的元素都被移除。程序输出如下:

```
initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced: 90 80 42 60 50 40 30 20 10
removed: 90 80 60 50
```

此种方式的程序编写，导致函数的组合。有趣的是，所有这些仿函数通常都声明为 `inline`。如此一来，你一方面使用类似函数的表示法或抽象性，一方面又能获得出色的效能。

另外还有一些仿函数。某些仿函数可用来调用群集内每个元素的成员函数：

```
for_each (coll.begin(), coll.end(),           // range
          mem_fun_ref(&Person::save));        // operation
```

仿函数 `mem_fun_ref` 用来调用它所作用的元素的某个成员函数。因此上例就是针对 `coll` 内的每个元素调用 `Person::save()`。当然啦，唯有当这些元素的型别是 `Person`，或 `Person` 的派生类，以上程序代码才能有效运作。

8.2 节, p305 对于 STL 预先定义的仿函数、函数配接器、以及各类函数组合，有更详尽的讨论，并告诉你如何撰写你自己的仿函数。

5.10 容器内的元素

容器内的元素必须符合特定条件，因为容器乃是以一种特别方式来操作它们。本节讨论这些条件。此外，容器会在内部对其元素进行复制，我也会讨论这种行为的后果。

5.10.1 容器元素的条件

STL 的容器、迭代器、算法都是 `templates`，因此可以操作任何型别——不论是 STL 预先定义好的或用户自行定义的都可以。然而，由于某些加诸于元素身上的操作行为，某些需求条件也就相应出现了。STL 容器元素必须满足以下三个基本要求：

1. 必须可透过 `copy` 构造函数进行复制。副本与原本必须相等 (`equivalent`)，亦即所有相等测试 (`equality test`) 的结果都必须显示，原本与副本行为一致。

所有容器都会在内部生成一个元素副本，并返回该暂时性副本，因此 `copy` 构造函数会被频繁地调用。所以 `copy` 构造函数的性能应该尽可能优化（这虽然不是条件之一，但可视为获得良好效能的诀窍）。如果对象的拷贝必须耗费大量时间，你可以选用“`reference` 语义”来使用容器，因而避免拷贝任何对象。详见 6.8 节, p222。

2. 必须可以透过 `assignment` 操作符完成赋值动作。容器和算法都使用 `assignment`

操作符，才能以新元素改写（取代）旧元素。

3. 必须可以透过析构函数完成销毁动作。当容器元素被移除（*removed*），它在容器内的副本将被销毁。因此析构函数绝不能被设计为 `private`。此外，依 C++ 惯例，析构函数绝不能抛出异常（`throw exceptions`），否则没戏唱了。

这三个条件对任何 `class` 而言其实都是隐式成立的。如果某个 `class` 既没有为上述动作定义特殊版本，也没有定义任何“可能破坏这些动作之健全性”的特殊成员，那么它自然而然也就满足了上述条件。

下面几个条件，也应当获得满足¹⁶：

- 对序列式容器而言，元素的 `default` 构造函数必须可用。
我们可以在没有给予任何初值的情况下，创建一个非空容器，或增加容器的元素个数。这些元素都将以 `default` 构造函数完成。
- 对于某些动作，必须定义 `operator==` 以执行相等测试。如果你有搜寻需求，这一点特别重要。
- 在关联式容器中，元素必须定义出排序准则。缺省情况下是 `operator<`，透过仿函数 `less<>` 被调用。

5.10.2 Value 语意 vs. Reference 语意

所有容器都会建立元素副本，并返回该副本。这意味容器内的元素与你放进去的对象“相等（`equal`）”但非“同一（`identical`）”。如果你修改容器中的元素，实际上改变的是副本而不是原先对象。这意味 STL 容器所提供的是“`value` 语意”。它们所容纳的是你所安插的对象值，而不是对象本身。然而实用上你也许需要用到“`reference` 语意”，让容器容纳元素的 `reference`。

STL 只支持 `value` 语意，不支持 `reference` 语意。这当然是利弊参半。好处是：

- 元素的拷贝很简单。
- 使用 `references` 时容易导致错误。你必须确保 `reference` 所指向的对象仍然健在，并需小心对付偶尔出现的循环引用（`circular references`）状态。

缺点是：

- “拷贝元素”可能导致不好的效能；有时甚至无法拷贝。
- 无法在数个不同的容器中管理同一份对象。

¹⁶ 在某些老式系统中，即使你未用到这些额外条件，也必须满足它们。例如某些 `vector` 实作版本无论如何用到元素的 `default` 构造函数。另一些实作版本则要求 `comparison`（比较）操作符必须存在。然而根据标准，这些要求是错误的，所以它们终将逐渐被取消。

实用上你同时需要两种作法。你不但需要一份独立（于原先对象）的拷贝（此乃 value 语意），也需要一份代表原数据、以能相应改变原值的拷贝（此乃 reference 语意）。不幸的是，C++ 标准程序库不支持 reference 语意。不过我们可以利用 value 语意来实现 reference 语意。

一个显而易见的方法是以指针作为元素¹⁷。然而一般指针有些常见问题。例如它们指向的对象也许不复存在，“比较”行为也未必如你所预期，因为实际比较的是指针而非指针所指对象。所以使用一般指针作为容器元素，必须非常谨慎。

好一点的办法是使用某种智能型指针（smart pointers），所谓智能型指针，是一种对象，有着类似指针的接口，但内部作了一些额外检查和处理工作。这里有一个重要的问题：它们需要多么智能？C++ 标准程序库确实提供了一个智能型指针，名为 `auto_ptr`（详见 4.2 节, p38），乍见之下用于此处似乎颇为合适。然而，你可千万别使用 `auto_ptr`，因为它们不符合作为容器元素所需的基本要求。当 `auto_ptr` 执行了拷贝（copy）或赋值（assign）动作后，目标对象与原对象并不相等：原来的那个 `auto_ptr` 发生了变化，其值并不是被拷贝了，而是被移转了（见 p43 和 p47）。这意味即使对容器中的元素进行排序和打印，也会摧毁它们！所以，千万别在容器内放置 `auto_ptr`（如果你的 C++ 系统符合标准规范，当你企图将 `auto_ptr` 当做容器元素，你应该会收到错误讯息）。详见 p43。

想要获得适用于 STL 容器的 reference 语意，你必须自己写个合适的智能型指针。但请注意：就算你使用带有引用计数（reference counting）功能的智能型指针（译注：可参考《More Effective C++》条款 28），也就是那种“当最后一个指向对象的 reference 不复存在后，能够自动摧毁对象”的智能型指针，仍然很麻烦。举个例子，如果你拥有直接存取元素的能力，你就可以更改元素值，而这在关联式容器中却会打破元素顺序关系。你肯定不想那样是吧！6.8 节 p222 更细致地探讨了容器的 reference 语意。尤其棒的是该处展示了一种做法，通过“引用计数”智能型指针，实现 STL 容器的 reference 语意。

5.11 STL 内部的错误处理和异常处理

错误是无可避免的，可能是程序（程序员）引起的逻辑性错误（logical error），也可能是程序运行时的环境或背景（例如内存不足）所引起的执行期错误（runtime error）。这两种错误都能够被异常机制（exceptions）处理（p15 有一个关于异常的简短介绍）。本节讨论 STL 内部如何处理错误（error）和异常（exceptions）。

¹⁷ C 程序员或许很能认可“以指针实现 reference 语意”的手法。因为在 C 语言中函数的参数只能 passed by value（传值），因此需要通过指针才能实现所谓的 call by reference。

5.11.1 错误处理 (Error Handling)

STL 的设计原则是效率优先，安全次之。错误检查相当花时间，所以几乎没有。如果你能正确无误地编写程序，自然很好。如果你不行，那就大难临头了。C++ 标准程序库接纳 STL 之前，对于是否应该加入更多的错误检验，曾有过一些讨论。大部分人决定不加入，原因有二：

1. 错误检验会降低效率，而速度始终是程序的总体目标。刚刚提过，良好的效率是 STL 的设计目标之一。
2. 如果你认为安全重于效率，你还是可以如愿：或增加一层包装 (wrapper)，或使用 STL 特殊版本。但是，一旦错误检验被放进所有基本操作内，再想消除它们以获得高效率，可就没门了。举个例子，如果每一个 subscript (下标) 操作符都对索引范围进行合法性检验，你就无法撰写不作检验的版本。反过来则可以。

所以，错误检验是可以获得的，但并不是 STL 的内在条件。

C++ 标准程序库指出，对于 STL 的任何运用，如果违反规则，将会导致未定义的行为。因此，如果索引、迭代器、或区间范围不合法，结果将未有定义。如果你使用的 STL 并非安全版本，就会导致未定义的内存存取，这可能导致难缠的副作用，甚至导致全盘崩溃。从这个意义上说，STL 和 C 指针一样容易引发错误。寻找这样的错误是非常困难的，尤其当你缺乏一个 STL 安全版本时，更是如此。

具体地说，使用 STL 时，必须满足以下要求：

- 迭代器务必合法而有效。例如你必须在使用它们之前先将它们初始化。注意，迭代器可能会因为其它动作的副效应而变得无效。例如当 vectors 和 deque 发生元素的安插、删除或重新配置时，迭代器可能因此失效。
- 一个迭代器如果指向“逾尾 (past-the-end)”位置，它并不指向任何对象，因此不能对它调用 `operator*` 或 `operator->`。这一点适用于任何容器的 `end()` 和 `rend()` 所返回的迭代器。
- 区间 (range) 必须是合法的：
 - 用以指出某个区间的前后两迭代器，必须指向同一个容器。
 - 从第一个迭代器出发，必须可以到达第二个迭代器所指位置。
- 如果涉及的区间不只一个，第二区间及后继各区间必须拥有“至少和第一区间一样多”的元素。
- 覆盖 (overwritten) 动作中的“目标区间” (destination ranges) 必须拥有足够元素，否则就必须采用 Insert iterators (插入型迭代器)。

以下实例展示了一些可能的错误：

```
// stl/iterbug1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll1;           // empty collection
    vector<int> coll2;           // empty collection

    /* RUNTIME ERROR:
    * - beginning is behind the end of the range
    */
    vector<int>::iterator pos = coll1.begin();
    reverse (++pos, coll1.end());

    // insert elements from 1 to 9 into coll2
    for (int i=1; i<=9; ++i) {
        coll2.push_back (i);
    }

    /* RUNTIME ERROR:
    * - overwriting nonexisting elements
    */
    copy (coll2.begin(), coll2.end(), // source
          coll1.begin());             // destination

    /* RUNTIME ERROR:
    * - collections mistaken
    * -begin() and end() mistaken
    */
    copy (coll1.begin(), coll2.end(), // source
          coll1.end());               // destination
}
```

注意，这些错误发生在执行期间而非编译期间，因而导致未定义的行为。

误用 STL 的方法有千百种，STL 没有义务预防你的各种可能不慎。因此，在软件开发阶段使用“安全版本”的 STL 是个好主意。第一个 STL 安全版本由 Cay Horstmann 开发¹⁸。不幸的是大部分 STL 开发厂商所供应的 STL，都是植基于 STL 最原始版本，其中并未包含错误处理。但是情况正在好转，有一个带有警戒能力的 STL 版本，名为“STLport”，几乎适用于任何平台，可自 <http://www.stlport.org/> 免费下载。

¹⁸ 你可以从 www.horstmann.com/safestl.html 获得一份由 Cay Horstmann 开发的“safe STL”。

5.11.2 异常处理 (Exception Handling)

STL 几乎不检验逻辑错误。所以逻辑问题几乎不会引发 STL 产生异常。事实上 C++ *Standard* 只要求唯一一个函数调用动作必要时直接引发异常: `vector` 和 `deque` 的成员函数 `at()` (它是下标操作符的受验版本)。此外, C++ *Standard* 要求, 只有一般的 (标准的) 异常才可以发生, 像是因内存不足而引发的 `bad_alloc` 或是因客户自定之操作行为而引发的异常。

异常何时发生? 异常一旦发生对 STL 组件有何影响? 在标准化过程中, 很长一段时间里, 并未对此问题定义出相关的行为规范。事实上每一个异常都会引发未定义的行为。如果执行某项动作的过程中抛出异常, 那么即使容器马上解构, 也会导致未定义行为, 例如程序整个崩解。因此如果你需要的是有担保的、确定的行为, STL 无能为力, 它甚至不可能正确地将堆栈辗转开解 (所谓 *stack unwinding*)。

如何处理异常, 这是标准化过程中最晚的几个讨论议题之一。找到好的解决方法可不容易, 而且花了很长时间, 因为:

1. 很难确定 C++ 标准程序库究竟应该提供怎样的安全程度。你大概认为应该尽可能提供最佳安全性。例如你可能觉得, 对着 `vector` 中的任何位置插入一个新元素, 要么成功, 要么应该不生任何效果。然而把后继元素向后移动以空出位置容纳新元素, 这种行为通常会导致异常, 而且无法复原。如果想要达成上述提出的目标, 安插操作就必须把 `vector` 的每一个元素拷贝到新位置去, 这对效率是莫大的折损! 如果优异效能是设计目标之一 (就像 STL), 你绝对无法完美处理所有异常状况, 你必须在效率和安全之间寻求某种妥协。
2. 还有一种考虑: 处理异常的程序代码本身, 也会对效能带来负面影响。这与“尽可能获得最佳效能”的设计目标抵触。然而编译器实作者指出, 原则上, 异常处理的实作方案应该可以免除任何明显的效能负荷 (许多编译器也确实做到了这一点)。毫无疑问, 如果效能没有明显损耗, 又能在异常发生时拥有确定、有保障的行为 (而非当机了事), 那当然比较好。

经过种种讨论, C++ 标准程序库就“异常处理问题”提供了以下基本保证¹⁹: C++ 标准程序库在面对异常时, 保证不会发生资源泄漏 (*resources leak*), 也不会与容器的恒常特性 (*container invariants*) 发生抵触。

遗憾的是很多时候这还不够, 你需要更强的保证, 保证当异常被抛出时, 进行中的操作不产生任何影响。以异常的观点来看, 这种操作可被视为“原子的” (*atomic*)。借用数据库领域的一个术语, 这些操作支持所谓“交付或恢复, 二择

¹⁹ 特别感谢 Dave Abrahams 和 Greg Colvin 对于 C++ 标准程序库的异常安全问题所做的贡献, 以及在这个主题上对我的帮助。

—” (**commit-or-rollback**) 行为, 又称为“事务安全” (**transaction safe**)。

考虑到这种强烈需求, C++ 标准程序库如今作出以下保证:

- 对于所有“以节点为实现基础” (**node-based**) 的容器如 `lists`, `sets`, `multisets`, `maps` 和 `multimaps`, 如果节点构造失败, 容器保持不变。移除节点的动作保证不会失败 (当然你得保证析构函数不得抛出异常)。然而, 如果是对关联式容器插入多个元素, 为保证已序性 (**sorted**), 失败时无法完全恢复原状。所有对关联式容器“插入单一元素”的操作, 支持 **commit-or-rollback** 行为。也就是说, 要不成功, 要不没有任何影响。此外, 所有擦除 (`erase`) 操作, 无论是针对单一元素或针对多重元素, 肯定会成功。

面对 `lists`, 就算同时插入多个元素, 这个操作也是属于“事务安全” (**transaction-safe**)。事实上 `list` 的所有操作, 除了 `remove()`, `remove_if()`, `merge()`, `sort()` 和 `unique()` 之外, 要不就成功, 要不就没有任何影响 (也就是 **commit-or-rollback**)。至于上述各函数, C++ 标准程序库也提供了有条件的保证 (见 p172)。所以如果你需要一个 **transaction-safe** 容器, 就用 `list` 吧。

- 所有“以 `array` 为构造基础” (**array-based**) 的容器如 `vectors` 和 `deque`s, 安插元素时如果失败, 都不可能做到完全回复。要达到完全回复, 就必须在安插动作之前拷贝所有 (安插点之后的) 后继元素。而且为了实现拷贝动作的完全回复性, 需要耗费大量时间。不过由于 `push` 和 `pop` 这两个动作在容器尾端执行, 不需拷贝任何既有元素, 所以万一发生异常, 这两个动作可以保证容器会回复原状。此外, 如果元素的型别能够保证拷贝动作 (也就是 `copy` 构造函数和 `assignment` 操作符) 不抛出异常, 则所有加诸于该种元素身上的操作, 都能够保证“要不就成功, 要不就毫无影响”的行为。

6.10.10 节, p248 有一份详细整理, 让你对“异常发生时, 拥有较强保证”的各种容器操作, 有一份了解。

注意, 所有这些保证都有一个前提: 析构函数不得抛出异常 (C++ 中通常如此)。C++ 标准程序库做了这个承诺, 身为应用程序员的你, 也得做出相同承诺。

如果你需要具备“完全 **commit-or-rollback** 能力”的容器, 你应当使用 `list` (但不要调用它的 `sort` 和 `unique`), 或使用任何关联式容器 (但不要对它安插多个元素)。当你使用它们, 可以确保数据不会损失, 也确保不会在任何“修改动作”之前先拷贝元素 — 要知道, 对一个容器而言, 拷贝动作极可能代价高昂。

如果你不使用“以节点为构造基础” (**node-based**) 的容器, 但又希望获得“完全 **commit-or-rollback** 能力”, 只好自己动手为每一个关键操作提供一份包装 (`wrapper`) 了。举个例子, 以下函数对任何容器而言, 几乎都可以安全地将元素安

插于某个特定位置上:

```
template <class T, class Cont, class Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
    Cont tmp(coll);           // copy container and all elements
    tmp.insert(pos,value);    // modify the copy
    coll.swap(tmp);          // use copy (in case no exception was thrown)
}
```

注意我的用词,我说的是“几乎”,因为这个函数仍然未臻完美。这是因为,当 `swap()` 针对关联性容器复制“比较准则 (comparison criterion)”时如果发生异常,那么 `swap()` 便会抛出异常。这下你明白了吧,想完美处理异常是多么不容易!

5.12 扩展 STL

STL 被设计成一个框架 (framework), 可以向任何方向扩展。你可以提供自己的容器、迭代器、算法、仿函数……, 只要你满足条件即可。事实上很多有用的扩展都没有出现在 C++ 标准程序库中。不能非难他们, C++ 标准委员会必须在某个时刻停止加入新特性, 将精力集中于现有特性的完善上, 否则标准化工作永无完结之日。STL 遗漏的最重要组件是 hash table (容器类)。这完全是因为它提出太晚之故。新的标准程序库很可能包含数种不同形式的 hash table。大部分 C++ 标准程序库实作版本已经提供了 hash table, 只可惜彼此之间有些差异。详见 6.7.3 节, p221。

另一些有用的扩展是额外的仿函数 (8.3 节, p313)、迭代器 (7.5.2 节, p288)、容器 (6.7 节, p217) 和算法 (7.5.1 节, p285)。



6

STL 容器

STL Container

本章延续第 5 章的讨论，详细讲解 STL 容器。首先对所有容器共通的能力和操作进行巡礼，然后详细讲解每一个容器，包括内部数据结构、操作（operations）、性能，以及各种操作的应用。如果某些操作值得深述，我还会给出相应的实例。每个容器的讲解都以一个典型应用实例作为结束。本章还讨论一个有趣的问题：各种容器的使用时机。比较各种容器的能力、优点、缺点之后，你便会了解如何选择最符合需求的容器。最后，本章详细介绍了每一个容器的所有成员。这一部分可视为参考手册，你可以在其中找到容器接口的细节和容器操作的确切标记式（signature）。必要的时候我会列出交叉索引，帮助你了解相似或互补的算法。

C++ 标准程序库还提供了一些特殊的容器类别——所谓的“容器配接器”（container adapters，包括 stack, queue, priority queue），以及 bitsets 和 valarrays。这些容器都有一些特殊接口，并不满足 STL 容器的一般要求，所以本书把它们放在其它章节讲解¹。容器配接器和 bitsets 安排在第 10 章，valarrays 安排在 12.2 节，p547。

¹ 从历史沿革来说，容器配接器是 STL 的一部分。然而，从概念角度观之，它们并不属于 STL framework，它们只不过是“使用” STL。

6.1 容器的共通能力和共通操作

6.1.1 容器的共通能力

本节讲述 STL 容器的共通能力。其中大部分都是必要条件，所有 STL 容器都必须满足那些条件。三个最核心的能力是：

1. 所有容器提供的都是“value 语意”而非“reference 语意”。容器进行元素的安插操作时，内部实施的是拷贝操作，置于容器内。因此 STL 容器的每一个元素都必须能够被拷贝。如果你打算存放的对象不具有 public copy 构造函数，或者你要的不是副本（例如你要的是被多个容器共同容纳的元素），那么容器元素就只能是指针（指向对象）。5.10.2 节, p135 对此有所描述。
2. 总体而言，所有元素形成一个次序（order）。也就是说，你可以依相同次序一次或多次遍历每个元素。每个容器都提供“可返回迭代器”的函数，运用那些迭代器你就可以遍历元素。这是 STL 算法赖以生存的关键接口。
3. 一般而言，各项操作并非绝对安全。调用者必须确保传给操作函数的参数符合需求。违反这些需求（例如使用非法索引）会导致未定义的行为。通常 STL 自己不会抛出异常。如果 STL 容器所调用的使用者自定义操作（user-defined operations）抛出异常，会导致各不相同的行为。参见 5.11.2 节, p139。

6.1.2 容器的共通操作

以下操作作为所有容器共有，它们均满足上述核心能力。表 6.1 列出这些操作。后续各小节分别探讨这些共通操作。

初始化 (initialization)

每个容器类别都提供了一个 default 构造函数，一个 copy 构造函数和一个析构函数。你可以以某个已知区间的内容作为容器初值——是的，负责此项工作的构造函数专门用来从另一个容器或数组或标准输入装置（standard input）得到元素并构造出容器。这些构造函数都是 member templates (p11)，所以如果提供了从“来端”到“目标端”的元素型别自动转换，那么不光是容器型别可以不同，元素型别也可以不同²。下面是个实例：

² 如果系统本身不支持 member templates，那就只能接受相同型别。你可以换用 copy() 算法，参见 p188 范例。

表 6.1 容器类别 (Container Classes) 的共通操作函数

操作	效果
<code>ContType c</code>	产生一个未含任何元素的空容器
<code>ContType c1(c2)</code>	产生一个同型容器
<code>ContType c(beg,end)</code>	复制 <code>[beg;end]</code> 区间内的元素, 作为容器初值
<code>c.~ContType()</code>	删除所有元素, 释放内存
<code>c.size()</code>	返回容器中的元素数量
<code>c.empty()</code>	判断容器是否为空 (相当于 <code>size()==0</code> , 但可能更快)
<code>c.max_size()</code>	返回元素的最大可能数量
<code>c1 == c2</code>	判断是否 <code>c1</code> 等于 <code>c2</code>
<code>c1 != c2</code>	判断是否 <code>c1</code> 不等于 <code>c2</code> , 相当于 <code>!(c1 == c2)</code>
<code>c1 < c2</code>	判断是否 <code>c1</code> 小于 <code>c2</code>
<code>c1 > c2</code>	判断是否 <code>c1</code> 大于 <code>c2</code> , 相当于 <code>c2 < c1</code>
<code>c1 <= c2</code>	判断是否 <code>c1</code> 小于等于 <code>c2</code> , 相当于 <code>!(c2 < c1)</code>
<code>c1 >= c2</code>	判断是否 <code>c1</code> 大于等于 <code>c2</code> , 相当于 <code>!(c1 < c2)</code>
<code>c1 = c2</code>	将 <code>c2</code> 的所有元素赋值给 <code>c1</code>
<code>c1.swap(c2)</code>	交换 <code>c1</code> 和 <code>c2</code> 的数据
<code>swap(c1,c2)</code>	同上, 是个全局函数
<code>c.begin()</code>	返回一个迭代器, 指向第一元素
<code>c.end()</code>	返回一个迭代器, 指向最后元素的下一位置
<code>c.rbegin()</code>	返回一个逆向迭代器, 指向逆向遍历时的第一元素
<code>c.rend()</code>	返回一个逆向迭代器, 指向逆向遍历时的最后元素的下一位置
<code>c.insert(pos,elem)</code>	将 <code>elem</code> 的一份副本安插于 <code>pos</code> 处。返回值和 <code>pos</code> 的意义并不相同
<code>c.erase(beg,end)</code>	移除 <code>[beg;end]</code> 区间内的所有元素。某些容器会返回未被移除的第一个接续元素
<code>c.clear()</code>	移除所有元素, 令容器为空
<code>c.get_allocator()</code>	返回容器的内存模型 (memory model)

- 以另一个容器的元素为初值, 完成初始化操作:

```
std::list<int> l; // l is a linked list of ints
...
// copy all elements of the list as floats into a vector
std::vector<float> c(l.begin(),l.end());
```

- 以某个数组的元素为初值，完成初始化操作：

```
int array[] = { 2, 3, 17, 33, 45, 77 };
...
// copy all elements of the array into a set
std::set<int> c(array,array+sizeof(array)/sizeof(array[0]));
```

- 以标准输入装置完成初始化操作：

```
// read all integer elements of the deque from standard input
std::deque<int> c({std::istream_iterator<int>(std::cin)},
                 {std::istream_iterator<int>()});
```

注意，不要遗漏了涵括“初始化参数”的那对“多余的”括号，否则这个表达式的意义会迥然不同，肯定让你匪夷所思，你会得到一堆奇怪的警告或错误。

以下是不写括号的情形：

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                 std::istream_iterator<int>());
```

这种情况下 `c` 被视为一个函数，返回值是 `deque<int>`，第一参数的型别是 `istream_iterator<int>`，参数名为 `cin`，第二参数无名称，型别是“一个函数，不接受任何参数，返回值型别为 `istream_iterator<int>`”。以上结构不论作为声明或表达式，语法上都正确。根据 C++ 规则它被视为声明。只要加上一对括号，便可使参数 `(std::istream_iterator<int>(std::cin))` 不再符合声明语法³，也就消除了歧义。

原则上还有一些操作，可支持从另一区间获取数据、赋值、插入元素。不过这些操作的确切接口在各容器中彼此不同，有不同的附加参数。

与大小相关的操作函数 (Size Operations)

所有容器都提供了三个和大小相关的操作函数：

1. `size()`

返回当前容器的元素数量。

2. `empty()`

这是 `size()==0` 表达式的一个快捷形式。`empty()` 的实作可能比 `size()==0` 的更有效率，所以你应该尽可能使用它。

3. `max_size()`

返回容器所能容纳的最大元素数量。其值因实作版本的不同而异。例如 `vector` 通常保有一个内存区块的全部元素，所以在 PCs 上可能会有相关限定。

³ 感谢 EDG 的 John H. Spicer 给予的说明。

`max_size()` 通常返回索引型别的最大值。

比较 (Comparisons)

包括常用的比较操作符 `==`, `!=`, `<`, `<=`, `>`, `>=`。它们的定义依据以下三条规则：

1. 比较操作的两端（两个容器）必须属于同一型别。
2. 如果两个容器的所有元素依序相等，那么这两个容器相等。采用 `operator==` 检查元素是否相等。
3. 采用字典式 (lexicographical) 顺序比较原则来判断某个容器是否小于另一个容器。参见 p360。

比较两个不同型别的容器，必须使用“比较”算法，参见 9.5.4 节, p356。

赋值 (Assignments) 和 `swap()`

当你对着容器赋值元素时，源容器的所有元素被拷贝到目标容器内，后者原本的所有元素全被移除。所以，容器的赋值操作代价比较高昂。

如果两个容器型别相同，而且拷贝后源容器不再被使用，那么我们可以使用一个简单的优化方法：`swap()`。`swap()` 的性能比上述优异得多，因为它只交换容器的内部数据。事实上它只交换某些内部指针（指向实际数据如元素、配置器、排序准则——如果有的话），所以时间复杂度是“常数”，不像实际赋值操作的复杂度为“线性”。

6.2 Vectors

`vector` 模拟出一个动态数组。因此，它本身是“将元素置于动态数组中加以管理”的一个抽象概念（图 6.1）。不过请注意，C++ *Standard* 并未要求必须以动态数组实作 `vector`，只是规定了相应条件和操作复杂度。

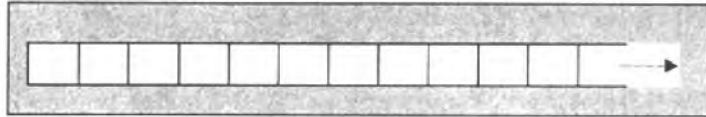


图 6.1 `vector` 的结构

使用 `vector` 之前，必须含入头文件 `<vector>`⁴

```
#include <vector>
```

其中，型别 `vector` 是一个定义于 `namespace std` 内的 `template`：

```
namespace std {  
    template <class T,  
              class Allocator = allocator<T> >  
    class vector;  
}
```

`vector` 的元素可以是任意型别 `T`，但必须具备 `assignable` 和 `copyable` 两个性质。第二个 `template` 参数可有可无，用来定义内存模型（`memory model`，参见 15 章）。缺省的内存模型是 C++ 标准程序库提供的 `allocator`⁵。

6.2.1 Vectors 的能力

`vectors` 将其元素复制到内部的 `dynamic array` 中。元素之间总是存在某种顺序，所以 `vectors` 是一种有序群集（`ordered collection`）。`vector` 支持随机存取，因此只要知道位置，你可以在常数时间内存取任何一个元素。`vector` 的迭代器是随机存取迭代器，所以对任何一个 STL 算法都可以奏效。

在末端附加或删除元素时，`vector` 的性能相当好。可是如果你在前端或中部安插或删除元素，性能就不怎么样了，因为操作点之后的每一个元素都必须移到另一个位置，而每一次移动都得调用 `assignment`（赋值）操作符。

⁴ 早期的 STL 中，`vectors` 的定义头文件是 `<vector.h>`。

⁵ 在不支持 `default template parameters` 的系统中，第二参数通常就没有了。

大小 (Size) 和容量 (Capacity)

vector 优异性能的秘诀之一，就是配置比其所容纳的元素所需更多的内存。为了能够高效运用 vectors，你应该了解大小和容量之间的关系。

vectors 之中用于操作大小的函数有 `size()`, `empty()`, `max_size()` (6.1.2 节, p144)。另一个与大小有关的函数是 `capacity()`，它返回 vector 实际能够容纳的元素数量。如果超越这个数量，vector 就有必要重新配置内部存储器。

vector 的容量之所以很重要，有以下两个原因：

1. 一旦内存重新配置，和 vector 元素相关的所有 references、pointers、iterators 都会失效。
2. 内存重新配置很耗时间。

所以如果你的程序管理了和 vector 元素相关的 references、pointers、iterators，或如果执行速度对你而言至关重要，那么就必须考虑容量问题。

你可以使用 `reserve()` 保留适当容量，避免一再重新配置内存。如此一来，只要保留的容量尚有余裕，就不必担心 references 失效。

```
std::vector<int> v;    // create an empty vector
v.reserve(80);        // reserve memory for 80 elements
```

另一种避免重新配置内存的方法是，初始化期间就向构造函数传递附加参数，构造出足够的空间。如果你的参数是个数值，它将成为 vector 的起始大小。

```
std::vector<T> v(5); // creates a vector and initializes it with five values
                    // (calls five times the default constructor of type T)
```

当然，要获得这种能力，此种元素型别必须提供一个 default 构造函数。请注意，如果型别很复杂，就算提供了 default 构造函数，初始化操作也很耗时。如果你这么做只不过是保留了足够的内存，那倒不如使用 `reserve()`。

vectors 的容量，概念上和 strings 类似（参见 11.2.5 节, p485）。不过有一个大不同点：vector 不能使用 `reserve()` 来缩减容量，这一点和 strings 不同。如果调用 `reserve()` 所给的参数比当前 vector 的容量还小，不会引发任何反应。此外，如何达到时间和空间的最佳效率，系由实作版本决定。因此具体实作版本中，容量的增长幅度可能比你我料想的还大。事实上为了防止内存破碎，在许多实作方案中即使你不调用 `reserve()`，当你第一次安插元素时也会一口气配置整块内存（例如 2K）。如果你有一大堆 vectors，每个 vector 的实际元素却寥寥无几，那么浪费的内存会相当可观。

既然 vectors 的容量不会缩减，我们便可确定，即使删除元素，其 references、pointers、iterators 也会继续有效，继续指向动作发生前的位置。然而安插操作却可能使 references、pointers、iterators 失效（译注：因为安插可能导致 vector 重新配置空间）。

这里有一个间接缩减 vector 容量的小窍门。注意，两个 vectors 交换内容后，两者的容量也会互换，因此下面的例子虽然保留了元素，却缩减了容量：

```
template <class T>
void shrinkCapacity(std::vector<T>& v)
{
    std::vector<T> tmp(v); // copy elements into a new vector
    v.swap(tmp);           // swap internal vector data
}
```

你甚至可以利用下面的语句直接缩减容量⁶：

```
// shrink capacity of vector v for type T
std::vector<T>(v).swap(v);
```

不过请注意，swap() 之后原先所有的 references、pointers、iterators 都换了指涉对象；它们仍然指向原本位置。换句话说，上述的 shrinkCapacity() 会使所有 references、pointers、iterators 失效。

6.2.2 Vector 的操作函数

构造、拷贝和解构

表 6.2 列出 vectors 的所有构造函数和析构函数。你可以在构造时提供元素，也可以不。如果只指定大小，系统便会调用元素的 default 构造函数——制造新元素。记住，即使对基本型别如 int，显式调用 default 构造函数进行初始化，也是一样可行（这个特性在 p14 介绍过）。请参考 6.1.2 节，p144 对于初始化的介绍。

表 6.2 Vectors 的构造函数和析构函数

操作	效果
<code>vector<Elem> c</code>	产生一个空 vector，其中没有任何元素
<code>vector<Elem> c1(c2)</code>	产生另一个同型 vector 的副本（所有元素都被拷贝）
<code>vector<Elem> c(n)</code>	利用元素的 default 构造函数生成一个大小为 n 的 vector
<code>vector<Elem> c(n,elem)</code>	产生一个大小为 n 的 vector，每个元素值都是 elem
<code>vector<Elem> c(beg,end)</code>	产生一个 vector，以区间 [beg,end] 做为元素初值
<code>c.~vector<Elem>()</code>	销毁所有元素，并释放内存

⁶ 你（或你的编译器）大概会认为这个语句实在荒谬，居然针对一个临时对象调用一个 non-const 成员函数。然而标准 C++ 确实允许我们这么做。

非变动性操作 (Nonmodifying Operations)

表6.3列出 vectors 的所有非变动性操作。参见 6.1.2 节, p14 附注和 6.2.1 节, p149。

表 6.3 Vectors 的非变动性操作

操作	效果
<code>c.size()</code>	返回当前的元素数量
<code>c.empty()</code>	判断大小是否为零。等同于 <code>size()==0</code> ，但可能更快
<code>c.max_size()</code>	返回可容纳的元素最大数量
<code>capacity()</code>	返回重新分配空间前所能容纳的元素最大数量
<code>reserve()</code>	如果容量不足，扩大之 ⁷
<code>c1 == c2</code>	判断 <code>c1</code> 是否等于 <code>c2</code>
<code>c1 != c2</code>	判断 <code>c1</code> 是否不等于 <code>c2</code> ，等同于 <code>!(c1==c2)</code>
<code>c1 < c2</code>	判断 <code>c1</code> 是否小于 <code>c2</code>
<code>c1 > c2</code>	判断 <code>c1</code> 是否大于 <code>c2</code> ，等同于 <code>c2<c1</code>
<code>c1 <= c2</code>	判断 <code>c1</code> 是否小于等于 <code>c2</code> ，等同于 <code>!(c2<c1)</code>
<code>c1 >= c2</code>	判断 <code>c1</code> 是否大于等于 <code>c2</code> ，等同于 <code>!(c1<c2)</code>

赋值 (Assignments)

表 6.4 Vectors 的赋值操作

操作	效果
<code>c1 = c2</code>	将 <code>c2</code> 的全部元素赋值给 <code>c1</code>
<code>c.assign(n,elem)</code>	复制 <code>n</code> 个 <code>elem</code> ，赋值给 <code>c</code>
<code>c.assign(beg,end)</code>	将区间 <code>[beg,end]</code> 内的元素赋值给 <code>c</code>
<code>c1.swap(c2)</code>	将 <code>c1</code> 和 <code>c2</code> 元素互换
<code>swap(c1,c2)</code>	同上。此为全局函数

表 6.4 列出“将新元素赋值给 vectors，并将旧元素全部移除”的方法。一系列 `assign()` 函数和构造函数一一对应。你可以采用不同的内容赋值方式（来自容器、数组和标准输入装置），这和 p144 的构造函数情况类似。所有赋值操作都可能会调用元素型别的 `default` 构造函数、`copy` 构造函数、`assignment` 操作符和/或析构函数，视元素数量的变化而定。例如：

```
std::list<Elem> l;  
std::vector<Elem> coll;
```

⁷ `reserve()` 的确会更易（变动，*modify*）vector。因为它造成所有 `references`、`pointers` 和 `iterators` 失效。但是从逻辑内容来说，容器并没有变化，所以还是把它列在这里。


```
...
// make coll be a copy of the contents of l
coll.assign(l.begin(),l.end());
```

元素存取 (Element Access)

表 6.5 列出用来直接存取 `vector` 元素的全部操作函数。按照 C 和 C++ 的惯例, 第一元素的索引为 0, 最后元素的索引为 `size()-1`。所以第 n 个元素的索引是 $n-1$ 。对于 `non-const vectors`, 这些函数都返回元素的 `reference`。也就是说你可以使用这些操作函数来更改元素内容 (如果没有其它妨碍因素的话)。

表 6.5 直接用来存取 `vectors` 元素的各项操作

操作	效果
<code>c.at(idx)</code>	返回索引 <code>idx</code> 所标示的元素。如果 <code>idx</code> 越界, 抛出 <code>out_of_range</code>
<code>c[idx]</code>	返回索引 <code>idx</code> 所标示的元素。不进行范围检查
<code>c.front()</code>	返回第一个元素。不检查第一个元素是否存在
<code>c.back()</code>	返回最后一个元素。不检查最后一个元素是否存在

对调用者来说, 最重要的事情莫过于搞清楚这些操作是否进行范围检查。只有 `at()` 会那么做。如果索引越界, `at()` 会抛出一个 `out_of_range` 异常 (详见 3.3 节, p25)。其它函数都不作检查。如果发生越界错误, 会引发未定义行为。对着一个空 `vector` 调用 `operator[]`, `front()`, `back()`, 都会引发未定义行为。

```
std::vector<Elem> coll;    // empty!

coll[5] = elem;            // RUNTIME ERROR → undefined behavior
std::cout << coll.front(); // RUNTIME ERROR → undefined behavior
```

所以, 调用 `operator[]` 时, 你必须心里有数, 确定索引有效; 调用 `front()` 或 `back()` 时必须确定容器不空:

```
std::vector<Elem> coll;    // empty!
if (coll.size() > 5) {
    coll[5] = elem;        // OK
}
if (!coll.empty()) {
    cout << coll.front();  // OK
}
coll.at(5) = elem;         // throws out_of_range exception
```

迭代器相关函数 (Iterator Functions)

vectors 提供了一些常规函数来获取迭代器，如表 6.6。vector 迭代器是 random access iterators (随机存取迭代器：关于迭代器分类详见 7.2 节, p251)，因此从理论上讲，你可以通过这个迭代器操作所有 STL 算法。

表 6.6 Vectors 的迭代器相关函数

操作	效果
c.begin()	返回一个随机存取迭代器，指向第一元素
c.end()	返回一个随机存取迭代器，指向最后元素的下一位置
c.rbegin()	返回一个逆向迭代器，指向逆向迭代的第一元素
c.rend()	返回一个逆向迭代器，指向逆向迭代的最后元素的下一位置

这些迭代器的确切型别由实作版本决定。对 vectors 来说，通常就是一般指针。一般指针就是随机存取迭代器，而 vector 内部结构通常也就是个数组，所以指针行为可以适用。不过你可不能仰仗这一点。例如也许有个 STL 安全版本，对所有区间范围和其它潜在错误实施检查，那么其 vector 迭代器可能就是个辅助类别。7.2.6 节, p258 展示了“以指针实作迭代器”和“以类别实作迭代器”之间的差异所引起的麻烦问题。

vector 迭代器持续有效，除非发生两种情况：(1) 使用者在一个较小索引位置上安插或移除元素，(2) 由于容量变化而引起内存重新分配 (详见 6.2.1 节, p149)。

安插 (insert) 和移除 (remove) 元素

表 6.7 列出 vector 元素的安插、移除操作函数。依 STL 惯例，你必须保证传入的参数合法：(1) 迭代器必须指向一个合法位置、(2) 区间的起始位置不能在结束位置之后、(3) 决不能从空容器中移除元素。

关于性能，以下情况你可以预期安插操作和移除操作会比较快些：

- 在容器尾部安插或移除元素
- 容量一开始就够大
- 安插多个元素时，“调用一次”当然比“调用多次”来得快

安插元素和移除元素，都会使“作用点”之后的各元素的 references、pointers、iterators 失效。如果安插操作甚至引发内存重新分配，那么该容器身上的所有 references、pointers、iterators 都会失效。

表 6.7 vector 的安插、移除相关操作

操作	效果
c.insert(pos,elem)	在 pos 位置上插入一个 elem 副本, 并返回新元素位置
c.insert(pos,n,elem)	在 pos 位置上插入 n 个 elem 副本。无回传值
c.insert(pos,beg, end)	在 pos 位置上插入区间 [beg,end] 内的所有元素的副本 无回传值
c.push_back(elem)	在尾部添加一个 elem 副本
c.pop_back()	移除最后一个元素 (但不回传)
c.erase(pos)	移除 pos 位置上的元素, 返回下一元素的位置
c.erase(beg,end)	移除 [beg, end] 区间内的所有元素, 返回下一元素的位置
c.resize(num)	将元素数量改为 num (如果 size() 变大了, 多出来的新元素都需以 default 构造函数构造完成)
c.resize(num,elem)	将元素数量改为 num (如果 size() 变大了, 多出来的新元素都是 elem 的副本)
c.clear()	移除所有元素, 将容器清空

vectors 并未提供任何函数可以直接移除“与某值相等”的所有元素。这是算法发挥威力的时候。以下语句可将所有其值为 val 的元素移除:

```
std::vector<Elem> coll;
...
// remove all elements with value val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
            coll.end());
```

具体解释详见 5.6.1 节, p111。

如果只是要移除“与某值相等”的第一个元素, 可以这么做:

```
std::vector<Elem> coll;
...
// remove first element with value val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(),
           val);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

6.2.3 将 Vectors 当做一般 Arrays 使用

C++ 标准程序库并未明确要求 `vector` 的元素必须分布于连续空间中。但是一份标准规格缺陷报告显示，这个缺点将获得弥补，标准规格书中将明确保证上述论点。如此一来你可以确定，对于 `vector v` 中任意一个合法索引 `i`，以下表达式肯定为 `true`：

```
&v[i] == &v[0] + i
```

保证了这一点，就可推导出一系列重要结果。简单地说，任何地点只要你需要一个动态数组，你就可以使用 `vector`。例如你可以利用 `vector` 来存放常规的 C 字符串（型别为 `char*` 或 `const char*`）：

```
std::vector<char> v;           // create vector as dynamic array of chars
v.resize(41);                 // make room for 41 characters (including '\0')
strcpy(&v[0], "hello, world"); // copy a C-string into the vector
printf("%s\n", &v[0]);        // print contents of the vector as C-string
```

不过，这么运用 `vector` 你可得小心（和使用动态数组一样小心），例如你必须确保上述 `vector` 的大小足以容纳所有数据，如果你用的是 C-String，记住最后有个 `'\0'` 元素。这个例子说明，不管出于什么原因（例如为了和既有的 C 程序库打交道），只要你需要一个元素型别为 `T` 的数组，就可以采用 `vector<T>`，然后传递第一元素的地址给它。

注意，千万不要把迭代器当做第一元素的地址来传递。`vector` 迭代器是由实作版本定义的，也许并不是个一般指针。

```
printf("%s\n", v.begin()); // ERROR (might work, but not portable)
printf("%s\n", &v[0]);     // OK
```

6.2.4 异常处理 (Exception Handling)

`vector` 只支持最低限度的逻辑错误检查。`subscript`（下标）操作符的安全版本 `at()`，是唯一被标准规格书要求可能抛出异常的一个函数（p152）。此外标准规格书也规定，只有一般标准异常（例如内存不足时抛出 `bad_alloc`），或用户自定操作函数的异常，才可能发生。

如果 `vector` 调用的函数（元素型别所提供的函数，或使用者提供的函数）抛出异常，C++ 标准程序库作出如下保证：

1. 如果 `push_back()` 安插元素时发生异常，该函数不起作用。
2. 如果元素的拷贝操作（包括 `copy` 构造函数和 `assignment` 操作符）不抛出异常，那么 `insert()` 要么成功，要么不生效用。
3. `pop_back()` 决不会抛出任何异常。

4. 如果元素拷贝操作 (包括 copy 构造函数和 assignment 操作符) 不抛出异常, erase() 和 clear() 就不抛出异常。
5. swap() 不抛出异常。
6. 如果元素拷贝操作 (包括 copy 构造函数和 assignment 操作符) 绝对不会抛出异常, 那么所有操作不是成功, 就是不起作用。这类元素可被称为 **POD** (plain old data, 简朴的老式数据)。POD 泛指那些无 C++ 特性的型别, 例如 C structure 便是。

所有这些保证都基于一个条件: 析构函数不得抛出异常。参见 5.11.2 节, p139 对于 STL 异常处理的一般性讨论。6.10.10 节, p249 列出对于异常给予特别保证的所有容器操作函数。

6.2.5 Vectors 运用实例

下面的例子展示了 vectors 的简单用法:

```
// cont/vector1.cpp

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

int main()
{
    // create empty vector for strings
    vector<string> sentence;

    // reserve memory for five elements to avoid reallocation
    sentence.reserve(5);

    // append some elements
    sentence.push_back("Hello, ");
    sentence.push_back("how");
    sentence.push_back("are");
    sentence.push_back("you");
    sentence.push_back("?");

    // print elements separated with spaces
    copy (sentence.begin(), sentence.end(),
          ostream_iterator<string>(cout, " "));
    cout << endl;
```

```
// print "technical data"
cout << " max_size(): " << sentence.max_size() << endl;
cout << " size(): " << sentence.size() << endl;
cout << " capacity(): " << sentence.capacity() << endl;

// swap second and fourth element
swap (sentence[1], sentence[3]);

// insert element "always" before element "?"
sentence.insert (find(sentence.begin(), sentence.end(), "?"),
                "always");

// assign "!" to the last element
sentence.back() = "!";

// print elements separated with spaces
copy (sentence.begin(), sentence.end(),
      ostream_iterator<string>(cout, " "));
cout << endl;

// print "technical data" again
cout << " max_size(): " << sentence.max_size() << endl;
cout << " size(): " << sentence.size() << endl;
cout << " capacity(): " << sentence.capacity() << endl;
}
```

程序的输出可能像这样:

```
Hello, how are you ?
max_size(): 268435455
size(): 5
capacity(): 5
Hello, you are how always !
max_size(): 268435455
size(): 6
capacity(): 10
```

注意我说的是“可能”。是的, `max_size()` 和 `capacity()` 的结果由实作版本决定。从这个例子中你可以看到, 当容量不足时, 此一实作版本将容量扩充一倍。

6.2.6 Class vector<bool>

C++ 标准程序库专门针对元素型别为 `bool` 的 `vector` 设计了一个特殊版本，目的是获取一个优化的 `vector`。其耗用空间远远小于以一般 `vector` 实作出来的 `bool vector`。一般 `vector` 的实作版本会为每个元素至少分配一个 `byte` 空间，而 `vector<bool>` 特殊版本内部只用一个 `bit` 来存储一个元素。所以通常小 8 倍之多。不过这里有个小麻烦：C++ 的最小可寻址值通常以 `byte` 为单位。所以上述的 `vector` 特殊版本需针对 `references` 和 `iterators` 作特殊考虑。

考虑结果是，`vector<bool>` 无法满足其它 `vectors` 必须的所有条件（例如 `vector<bool>::reference` 并不返回真正的 `lvalue`，`vector<bool>::iterator` 不是个真正的随机存取迭代器）。所以某些 `template` 程序代码可能适用于任何型别的 `vector`，唯独无法应付 `vector<bool>`。此外 `vector<bool>` 可能比一般 `vectors` 慢一些，因为所有元素操作都必须转化为 `bit` 操作。不过 `vector<bool>` 的具体方案也是由实作版本决定，所以性能（包括速度和空间消耗）也可能都有不同。

注意，`vector<bool>` 不仅仅是个特殊的 `bool` 版本，它还提供某些特殊的 `bit` 操作。你可以利用它们更方便地操作 `bit` 或标志（`flags`），而且由于 `vector<bool>` 的大小可动态改变，你还可以把它当成动态大小的 `bitfield`（位域）。如此一来你便可以添加或移除 `bits`。如果你需要静态大小的 `bitfield`，应当使用 `bitset`，而不是 `vector<bool>`。`bitset` 详见 10.4 节，p460。

表 6.8 `vector<bool>` 的特殊操作

操作	效果
<code>c.flip()</code>	将所有 <code>bool</code> 元素值取反值，亦即求补码
<code>m[idx].flip()</code>	将索引 <code>idx</code> 的 <code>bit</code> 元素取反值
<code>m[idx] = val</code>	令索引 <code>idx</code> 的 <code>bit</code> 元素值为 <code>val</code> （指定单一 <code>bit</code> ）
<code>m[idx1] = m[idx2]</code>	令索引 <code>idx1</code> 的 <code>bit</code> 元素值为索引 <code>idx2</code> 的 <code>bit</code> 元素值

表 6.8 列出 `vector<bool>` 的特殊操作。`flip()` 对 `vector` 中的每一个 `bit` 取补码。注意，你竟然可以对单一 `bool` 元素调用 `flip()`。是不是很惊讶？也许你觉得让 subscript 操作符返回 `bool`，再对如此基本型别调用 `flip()` 是不可能的。然而这里 `vector<bool>` 用了一个常见技巧，称作 **proxy**⁸，对于 `vector<bool>`，subscript 操作符（及其它返回单一元素的操作符）的返回型别实际上是个辅助类别，一旦你

⁸ proxy 可让你控制一般无法控制的东西，通常用来获取更好的安全性。上述情形中，此技术施行某种控制，使某种操作成为可能。原则上其返回的对象行为类似 `bool`。

要求返回值为 `bool`，便会触发一个自动型别转换函数。表 6.8 的其它操作由成员函数支持。`vector<bool>` 的相关声明如下：

```
namespace std {
class vector<bool> {
public:
    // auxiliary type for subscript operator
    class reference {
    ...
public:
    // automatic type conversion to bool
    operator bool() const;

    // assignments
    reference& operator= (const bool);
    reference& operator= (const reference&);

    // bit complement
    void flip();
    }
    ...

    // operations for element access
    // - return type is reference instead of bool
    reference operator[](size_type n);
    reference at(size_type n);
    reference front();
    reference back();
    ...
};
}
```

你会发现，所有用于元素存取的函数，返回的都是 `reference` 型别。所以，你可以使用以下语句：

```
c.front().flip(); // negate first Boolean element
c[5] = c.back(); // assign last element to element with index 5
```

一如往常，为了避免未定义的行为，调用者必须确保第一、第六和最后一个元素存在。

只有在 `non-const vector<bool>` 容器中才会用到内部型别 `reference`。存取元素用的 `const member function` 会返回型别为 `bool` 的普通数值。

6.3 Deques

容器 `deque` (发音为 "deck") 和 `vector` 非常相似。它也采用动态数组来管理元素, 提供随机存取, 并有着和 `vector` 几乎一模一样的接口。不同的是 `deque` 的动态数组头尾都开放, 因此能在头尾两端进行快速安插和删除 (图 6.2)。

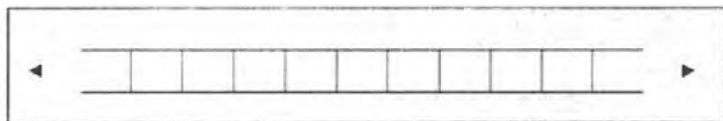


图 6.2 `deque` 的逻辑结构

为了获取这种能力, `deque` 通常实作为一组独立区块, 第一区块朝某方向扩展, 最后一个区块朝另一方向扩展, 如图 6.3。

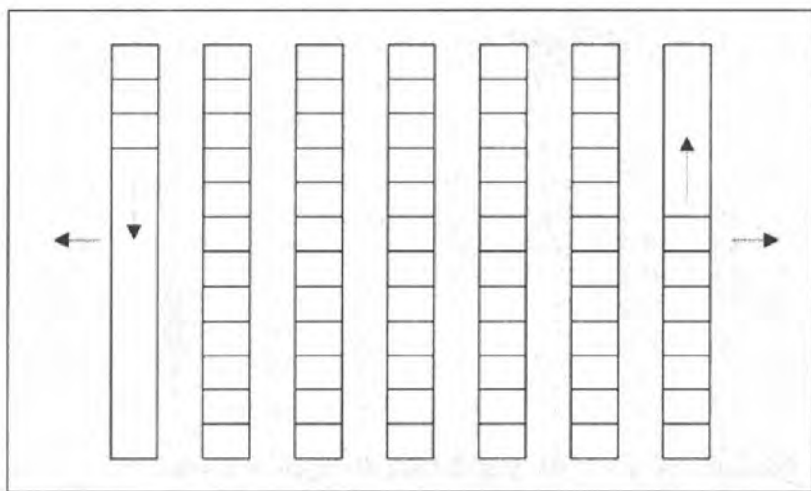


图 6.3 `deque` 的内部结构

使用 `deque` 之前, 必须先含入头文件 `<deque>`⁹:

```
#include <deque>
```

在其中, `deque` 型别是定义于命名空间 `std` 内的一个 `class template`:

```
namespace std {  
    template <class T,
```

⁹ 早期的 STL 中, `deque` 的头文件是 `<deque.h>`。

```
class Allocator = allocator<T> >
class deque;
}
```

和 `vector` 相同，第一个 `template` 参数用来表明元素型别——只要是 `assignable` 和 `copyable` 都可以胜任。第二个 `template` 参数可有可无，用来指定内存模型 (`memory model`)，缺省为 `allocator` (详见第 15 章)¹⁰。

6.3.1 Deques 的能力

与 `vectors` 相比，`deques` 功能上的不同处在于：

- 两端都能快速安插元素和移除元素 (`vector` 只在尾端逞威风)。这些操作可以在分期摊还的常数时间 (`amortized constant time`) 内完成。
- 存取元素时，`deque` 的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一些。
- 迭代器需要在不同区块间跳转，所以必须是特殊的智能型指针，非一般指针。
- 在对内存区块有所限制的系统中 (例如 PC 系统)，`deque` 可以内含更多元素，因为它使用不止一块内存。因此 `deque` 的 `max_size()` 可能更大。
- `deque` 不支持对容量和内存重分配时机的控制。特别要注意的是，除了头尾两端，在任何地方安插或删除元素，都将导致指向 `deque` 元素的任何 `pointers`、`references`、`iterators` 失效。不过，`deque` 的内存重分配优于 `vectors`，因为其内部结构显示，`deques` 不必在内存重分配时复制所有元素。
- `deque` 的内存区块不再被使用时，会被释放。`deque` 的内存大小是可缩减的。不过，是不是这么做，以及究竟怎么做，由实作版本定义之。

`deques` 的下述特性跟 `vectors` 差不多：

- 在中段部分安插、移除元素的速度相对较慢，因为所有元素都需移动以腾出或填补空间。
- 迭代器属于 `random access iterator` (随机存取迭代器)。

总之，如果是以下情形，最好采用 `deque`：

- 你需要在两端安插和移除元素 (这是 `deque` 的拿手好戏)。
- 无需引用 (*refer to*) 容器内的元素。
- 要求容器释放不再使用的元素 (不过，标准规格上并没有保证这一点)。

`vectors` 和 `deques` 的接口几乎一样，所以如果无需什么特殊性质，两者都可试试。

¹⁰ 在尚未支持 `default template parameters` 的系统中，第二参数通常会被省略。

6.3.2 Deques 的操作函数

表 6.9 至表 6.11 列出了 deque 的所有操作函数:

表 6.9 deque 的构造函数和析构函数

操作	效果
<code>deque<Elem> c</code>	产生一个空的 deque
<code>deque<Elem> c1(c2)</code>	针对某个 deque 产生同型副本 (所有元素都被拷贝)
<code>deque<Elem> c(n)</code>	产生一个 deque, 含有 n 个元素, 这些元素均以 default 构造函数产生出来
<code>deque<Elem> c(n, elem)</code>	产生一个 deque, 含有 n 个元素, 这些元素均是 elem 的副本
<code>deque<Elem> c(beg, end)</code>	产生一个 deque, 以区间 [beg; end] 内的元素为初值
<code>c.~deque<Elem>()</code>	销毁所有元素, 释放内存

表 6.10 deque 的非变动性操作 (nonmodifying operations)

操作	效果
<code>c.size()</code>	返回容器的实际元素个数
<code>c.empty()</code>	判断容器大小是否为零。等同于 <code>size()==0</code> , 但可能更快
<code>c.max_size()</code>	返回可容纳的最大元素数量
<code>c1 == c2</code>	判断是否 c1 等于 c2
<code>c1 != c2</code>	判断是否 c1 不等于 c2。等同于 <code>!(c1 == c2)</code>
<code>c1 < c2</code>	判断是否 c1 小于 c2
<code>c1 > c2</code>	判断是否 c1 大于 c2。等同于 <code>c2 < c1</code>
<code>c1 <= c2</code>	判断是否 c1 小于等于 c2。等同于 <code>!(c2 < c1)</code>
<code>c1 >= c2</code>	判断 c1 是否大于等于 c2。等同于 <code>!(c1 < c2)</code>
<code>c.at(idx)</code>	返回索引 idx 所标示的元素。如果 idx 越界, 抛出 <code>out_of_range</code>
<code>c[idx]</code>	返回索引 idx 所标示的元素, 不进行范围检查
<code>c.front()</code>	返回第一个元素。不检查元素是否存在
<code>c.back()</code>	返回最后一个元素。不检查元素是否存在
<code>c.begin()</code>	返回一个随机迭代器, 指向第一元素
<code>c.end()</code>	返回一个随机迭代器, 指向最后元素的下一位置
<code>c.rbegin()</code>	返回一个逆向迭代器, 指向逆向迭代时的第一个元素
<code>c.rend()</code>	返回一个逆向迭代器, 指向逆向迭代时的最后元素的下一位置

表 6.11 deques 的变动性操作 (modifying operations)

操作	效果
<code>c1 = c2</code>	将 <code>c2</code> 的所有元素赋值给 <code>c1</code>
<code>c.assign(n, elem)</code>	将 <code>n</code> 个 <code>elem</code> 副本赋值给 <code>c</code>
<code>c.assign(beg, end)</code>	将区间 <code>[beg; end)</code> 中的元素赋值给 <code>c</code>
<code>c1.swap(c2)</code>	将 <code>c1</code> 和 <code>c2</code> 的元素互换
<code>swap(c1, c2)</code>	同上。此为全局函数
<code>c.insert(pos, elem)</code>	在 <code>pos</code> 位置插入一个 <code>elem</code> 副本，并返回新元素的位置
<code>c.insert(pos, n, elem)</code>	在 <code>pos</code> 位置插入 <code>elem</code> 的 <code>n</code> 个副本，无返回值。
<code>c.insert(pos, beg, end)</code>	在 <code>pos</code> 位置插入在区间 <code>[beg; end)</code> 所有元素的副本，无返回值
<code>c.push_back(elem)</code>	在尾部添加 <code>elem</code> 的一个副本
<code>c.pop_back()</code>	移除最后一个元素 (但不回传)
<code>c.push_front(elem)</code>	在头部插入 <code>elem</code> 的一个副本
<code>c.pop_front()</code>	移除头部元素 (但不回传)
<code>c.erase(pos)</code>	移除 <code>pos</code> 位置上的元素，返回下一元素位置
<code>c.erase(beg, end)</code>	移除 <code>[beg, end)</code> 区间内的所有元素，返回下一元素位置
<code>c.resize(num)</code>	将大小 (元素个数) 改为 <code>num</code> 。如果 <code>size()</code> 增长了，新增元素都以 <code>default</code> 构造函数产生出来
<code>c.resize(num, elem)</code>	将大小 (元素个数) 改为 <code>num</code> 。如果 <code>size()</code> 增长了，新增元素都是 <code>elem</code> 的副本
<code>c.clear()</code>	移除所有元素，将容器清空

deques 的各项操作只有以下数点和 vectors 不同:

- 1. deques 不提供容量操作 (`capacity()`和 `reserve()`)。
- 2. deques 直接提供函数，用以完成头部元素的安插和删除 (`push_front()`和 `pop_front()`)。

其它操作都相同，所以这里不重复。它们的具体描述请见 p150, 6.2.2 节。

还有一些值得考虑的事情:

- 1. 除了 `at()`，没有任何成员函数会检查索引或迭代器是否有效。
- 2. 元素的插入或删除可能导致内存重新分配，所以任何插入或删除动作都会使所有指向 deques 元素的 `pointers`、`references` 和 `iterators` 失效。唯一例外是在头部或尾部插入元素，动作之后，`pointers` 和 `references` 仍然有效 (但 `iterators` 就没这么幸运)。

6.3.3 异常处理 (Exception Handling)

原则上 `deque` 提供的异常处理和 `vectors` 提供的一样 (p155)。新增的操作函数 `push_front()` 和 `pop_front()` 分别对应于 `push_back()` 和 `pop_back()`。因此, C++ 标准程序库保证下列行为:

- 如果以 `push_back()` 或 `push_front()` 安插元素时发生异常, 则该操作不带来任何效应。
- `pop_back()` 和 `pop_front()` 不会抛出任何异常。

STL 的异常处理一般原则请见 p139, 5.11.2 节。异常发生时, 提供特殊保障的所有容器操作函数均列于 p248, 6.10.10 节。

6.3.4 Deques 运用实例

以下程序以简单的例子说明 `deque` 的功用:

```
// cont/deque1.cpp

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

int main()
{
    // create empty deque of strings
    deque<string> coll;

    // insert several elements
    coll.assign (3, string("string"));
    coll.push_back ("last string");
    coll.push_front ("first string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout, "\n"));
    cout << endl;

    // remove first and last element
    coll.pop_front();
    coll.pop_back();
}
```

```
// insert "another" into every element but the first
for (int i=1; i<coll.size(); ++i) {
    coll[i] = "another " + coll[i];
}

// change size to four elements
coll.resize (4, "resized string");

// print elements separated by newlines
copy (coll.begin(), coll.end(),
      ostream_iterator<string>(cout, "\n"));
}
```

程序输出如下:

```
first string
string
string
string
last string

string
another string
another string
resized string
```

6.4 Lists

Lists 使用一个 doubly linked list (双向链表) 来管理元素, 如图 6.4。按惯例, C++ 标准程序库并未明定实作方式, 只是遵守 list 的名称、限制和规格。

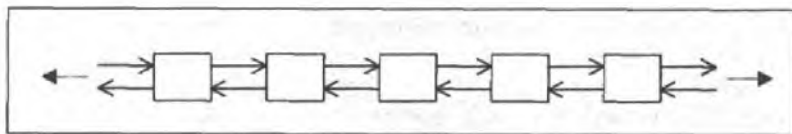


图 6.4 List 的结构

使用 list 时必须含入头文件 `<list>`¹¹:

```
#include <list>
```

其中 list 型别系定义于 namespace std 中, 是个 class template:

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class list;
}
```

任何型别 T 只要具备 assignable 和 copyable 两性质, 就可以作为 list 的元素。第二个 template 参数可有可无, 用来指定内存模型 (详见第 15 章)。缺省的内存模型是 C++ 标准程序库所提供的 allocator¹²。

6.4.1 Lists 的能力

Lists 的内部结构和 vector 或 deque 截然不同, 所以在几个主要方面与前述二者存在明显区别:

- Lists 不支持随机存取。如果你要存取第 5 个元素, 就得顺着串链一一爬过前 4 个元素。所以, 在 list 中随机遍历任意元素, 是很缓慢的行为。
- 任何位置上 (不只是两端) 执行元素的安插和移除都非常快, 始终都是常数时间内完成, 因为无需移动任何其它元素。实际上内部只是进行了一些指针操作而已。

¹¹ 早期 STL 中, list 的定义头文件是 `<list.h>`

¹² 如果系统不支持 default template parameters, 通常省略第二参数。

- 安插和删除动作并不会造成指向其它元素的各个 pointers、references、iterators 失效。
- Lists 对于异常有着这样的处理方式：要么操作成功，要么什么都不发生。你决不会陷入“只成功一半”这种前不着村后不巴店的尴尬境地。

Lists 所提供的成员函数反映出它和 vectors 以及 deque 的不同：

- 由于不支持随机存取，lists 既不提供 subscript（下标）操作符，也不提供 at（）。
- Lists 并未提供容量、空间重新分配等操作函数，因为全无必要。每个元素都有自己的内存，在被删除之前一直有效。
- Lists 提供了不少特殊的成员函数，专门用于移动元素。较之同名的 STL 通用算法，这些函数执行起来更快，因为它们无需拷贝或移动，只需调整若干指针即可。

6.4.2 Lists 的操作函数

生成（Creation），复制（Copy）和销毁（Destroy）

lists 的生成、复制和销毁动作，和所有序列式容器相同，详见表 6.12。关于初值来源（initialization sources）的若干注意事项，可参考 6.1.2 节，p144。

表 6.12 Lists 的构造函数和析构函数

操作	效果
<code>list<Elem> c</code>	产生一个空的 list
<code>list<Elem> c1(c2)</code>	产生一个与 c2 同型的 list（每个元素都被复制）
<code>list<Elem> c(n)</code>	产生拥有 n 个元素的 list，这些元素都以 default 构造函数初始化
<code>list<Elem> c(n,elem)</code>	产生拥有 n 个元素的 list，每个元素都是 elem 的副本
<code>list<Elem> c(beg,end)</code>	产生一个 list 并以 [beg;end] 区间内的元素为初值
<code>c.~list<Elem>()</code>	销毁所有元素，释放内存

非变动性操作（Nonmodifying Operations）

Lists 也提供诸如“询问大小”和“两相比较”等等一般性操作。详见表 6.13 和 6.1.2 节，p144。

表 6.13 Lists 的非变动性操作 (Nonmodifying Operations)

操作	效果
c.size()	返回元素个数
c.empty()	判断容器大小是否为零。等同于 size()==0，但可能更快
c.max_size()	返回元素的最大可能数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2。等同于 !(c1 == c2)
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2。等同于与 c2 < c1 相同
c1 <= c2	判断是否 c1 小于等于 c2。等同于 !(c2 < c1)
c1 >= c2	判断是否 c1 大于等于 c2。等同于 !(c1 < c2)

赋值 (Assignment)

和其它序列式容器一样，lists 也提供常用的赋值操作，如表 6.14。

表 6.14 Lists 的 assignment (赋值) 操作函数

操作	效果
c1 = c2	将 c2 的全部元素赋值给 c1
c.assign(n,elem)	将 elem 的 n 个拷贝赋值给 c
c.assign(beg,end)	将区间 [beg,end) 的元素赋值给 c
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1,c2)	同上。此为全局函数

一如往常，安插操作和构造函数一一匹配，如此一来就有能力提供不同的初值来源 (initialization sources)，详见 6.1.2 节, p144。

元素存取 (Element Access)

lists 不支持随机存取，只有 front() 和 back() 能够直接存取元素，如表 6.15。

表 6.15 Lists 元素的直接存取

操作	效果
c.front()	返回第一个元素。不检查元素存在与否
c.back()	返回最后一个元素。不检查元素存在与否

一如以往，这些操作并不检查容器是否为空。对着空容器执行任何操作，都会导致未定义的行为。所以调用者必须确保容器至少含有一个元素。例如：

```
std::list<Elem> coll; // empty!

std::cout << coll.front(); // RUNTIME ERROR → undefined behavior

if (!coll.empty()) {
    std::cout << coll.back(); // OK
}
```

迭代器相关函数 (Iterator Functions)

只有运用迭代器，才能够存取 list 中的各个元素。Lists 提供的迭代器函数如表 6.16。然而由于 list 不能随机存取，这些迭代器只是双向（而非随机）迭代器。所以凡是用到随机存取迭代器的算法（所有用来操作元素顺序的算法——特别是排序算法——都归此类）你都不能调用。不过你可以拿 list 的特殊成员函数 `sort()` 取而代之，详见 p245。

表 6.16 Lists 的迭代器相关函数

操作	效果
<code>c.begin()</code>	返回一个双向迭代器，指向第一个元素
<code>c.end()</code>	返回一个双向迭代器，指向最后元素的下一位置
<code>c.rbegin()</code>	返回一个逆向迭代器，指向逆向迭代的第一个元素
<code>c.rend()</code>	返回一个逆向迭代器，指向逆向迭代的最后元素的下一位置

元素的安插 (Inserting) 和移除 (Removing)

表 6.17 列出 lists 元素的安插和移除操作。Lists 提供 `deque` 的所有功能，还增加了 `remove()` 和 `remove_if()` 算法应用于 list 身上的特殊版本。

和一般运用 STL 时相似，你必须确保参数的正确。迭代器必须指向合法位置，区间终点不能位于区间起点的前头；还有，你不能从空容器中移除元素。

如果想要安插或移除多个元素，你可以对它们进行单一调用，比多次调用来得快。

为了移除元素，lists 特别配备了 `remove()` 算法（9.7.1 节, p378）的特别版本。这些成员函数比 `remove()` 算法的速度更快，因为它们只进行内部指目标操作，无需顾及元素。所以，面对 list，你应该调用成员函数 `remove()`，而不是像面对 `vectors` 和 `deque` 那样调用 STL 算法（如 p154 所示）。

表 6.17 Lists 的安插、移除操作函数

操作	效果
<code>c.insert(pos,elem)</code>	在迭代器 <code>pos</code> 所指位置上安插一个 <code>elem</code> 副本, 并返回新元素的位置
<code>c.insert(pos,n,elem)</code>	在迭代器 <code>pos</code> 所指位置上安插 <code>n</code> 个 <code>elem</code> 副本, 无返回值
<code>c.insert(pos,beg,end)</code>	在迭代器 <code>pos</code> 所指位置上安插 <code>[beg;end]</code> 区间内的所有元素的副本, 无返回值
<code>c.push_back(elem)</code>	在尾部追加一个 <code>elem</code> 副本
<code>c.pop_back()</code>	移除最后一个元素 (但不返回)
<code>c.push_front(elem)</code>	在头部安插一个 <code>elem</code> 副本
<code>c.pop_front()</code>	移除第一元素 (但不返回)
<code>c.remove(val)</code>	移除所有其值为 <code>val</code> 的元素
<code>c.remove_if(op)</code>	移除所有“造成 <code>op(elem)</code> 结果为 <code>true</code> ”的元素
<code>c.erase(pos)</code>	移除迭代器 <code>pos</code> 所指元素, 返回下一元素位置
<code>c.erase(beg,end)</code>	移除区间 <code>[beg;end]</code> 内的所有元素, 返回下一元素位置
<code>c.resize(num)</code>	将元素容量变为 <code>num</code> 。如果 <code>size()</code> 变大, 则以 <code>default</code> 构造函数构造所有新增元素
<code>c.resize(num,elem)</code>	将元素容量变为 <code>num</code> 。如果 <code>size()</code> 变大, 则以 <code>elem</code> 副本作为新增元素的初值
<code>c.clear()</code>	移除全部元素, 将整个容器清空

想要将所有“与某值相等”的元素移除, 可以用如下语句 (进一步细节详见 5.6.3 节, p116) :

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

如果只是移除“与某值相等”的第一个元素, 你得使用诸如 p154 中针对 `vectors` 所用的算法。

如果使用 `remove_if()`, 你可以通过一个函数或仿函数¹³来定义元素移除原则; 它可以将每一个“令传入之操作结果为 `true`”的元素移除:

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

如果你对以上语句感到头晕, 别急, 到 p306 看看详细解释。关于 `remove()` 和 `remove_if()` 的其它例子, 详见 p378。

¹³ 一个不支持 `member templates` 的系统, 通常不会提供 `remove_if()` 成员函数。

Splice 函数

Linked lists 的一大好处就是不论在任何位置，元素的安插和移除都只需要常数时间。如果你有必要将若干元素从 A 容器转放到 B 容器，那么上述好处就更见其效了，因为你只需要重新定向某些指针即可，如图 6.5。

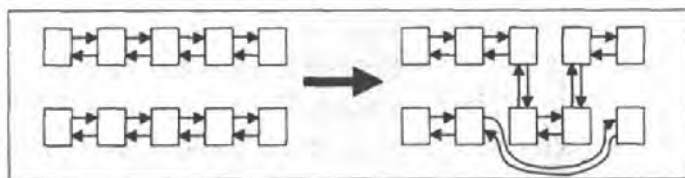


图 6.5 Splice 操作函数用以改变 list 元素的次序

为了利用这个优势，lists 不仅提供 `remove()`，还提供其它一些成员函数，用来改变元素和区间次序，或是用来重新串链。我们不仅可以调用这些函数，移动单一 list 内的元素，也可以移动两个 lists 之间的元素——只要 lists 的型别一致即可。表 6.18 列举出这些函数。6.10.8 节, p244 另有详细说明，实例可见 6.4.4 节, p172。

表 6.18 Lists 的特殊变动性操作 (Special Modifying Operations)

操作	效果
<code>c.unique()</code>	如果存在若干相邻而数值相等的元素，就移除重复元素，只留下一个
<code>c.unique(op)</code>	如果存在若干相邻元素，都使 <code>op()</code> 的结果为 <code>true</code> ，则移除重复元素，只留下一个
<code>c1.splice(pos,c2)</code>	将 <code>c2</code> 内的所有元素转移到 <code>c1</code> 之内、迭代器 <code>pos</code> 之前
<code>c1.splice(pos,c2, c2pos)</code>	将 <code>c2</code> 内的 <code>c2pos</code> 所指元素转移到 <code>c1</code> 内的 <code>pos</code> 所指位置上 (<code>c1</code> 和 <code>c2</code> 可相同)
<code>c1.splice(pos,c2, c2beg,c2end)</code>	将 <code>c2</code> 内的 <code>[c2beg;c2end)</code> 区间内所有元素转移到 <code>c1</code> 内的 <code>pos</code> 之前 (<code>c1</code> 和 <code>c2</code> 可相同)
<code>c.sort()</code>	以 <code>operator<</code> 为准则，对所有元素排序
<code>c.sort(op)</code>	以 <code>op()</code> 为准则，对所有元素排序
<code>c1.merge(c2)</code>	假设 <code>c1</code> 和 <code>c2</code> 容器都包含已序 (<i>sorted</i>) 元素，将 <code>c2</code> 的全部元素转移到 <code>c1</code> ，并保证合并后的 list 仍为已序
<code>c1.merge(c2,op)</code>	假设 <code>c1</code> 和 <code>c2</code> 容器都包含 <code>op()</code> 原则下的已序 (<i>sorted</i>) 元素，将 <code>c2</code> 的全部元素转移到 <code>c1</code> ，并保证合并后的 list 在 <code>op()</code> 原则下仍为已序
<code>c.reverse()</code>	将所有元素反序 (<i>reverse the order</i>)

6.4.3 异常处理 (Exception Handling)

所有 STL 标准容器中, lists 对于异常安全性 (**exception safety**) 提供了最佳支持。几乎所有操作都是不成功便成仁: 要么成功, 要么无效。仅有少数几个操作没有如此保证, 包括赋值 (赋值) 运算和成员函数 `sort()`, 不过它们也有基本保证: 异常发生时不会泄漏资源, 也不会与容器恒常特性 (**invariants**) 发生冲突。`merge()`, `remove()`, `remove_if()`, `unique()` 提供的保证是有前提的, 那就是元素间的比较动作 (采用 `operator==` 或判断式 *predicate*) 并不会抛出异常。用数据库编程术语来说, 只要你不调用赋值操作或 `sort()`, 并保证元素相互比较时不抛出异常, 那么 lists 便可说是“事务安全 (**transaction safe**)”。表 6.19 列出异常状况下提供特殊保证的所有操作函数。STL 异常处理的一般性讨论, 请见 5.11.2 节, p139。

表 6.19 Lists 的各种操作在异常发生时提供的特殊保证

操作	保证
<code>push_back()</code>	如果不成功, 就是无任何作用
<code>push_front()</code>	如果不成功, 就是无任何作用
<code>insert()</code>	如果不成功, 就是无任何作用
<code>pop_back()</code>	不抛出异常
<code>pop_front()</code>	不抛出异常
<code>erase()</code>	不抛出异常
<code>clear()</code>	不抛出异常
<code>resize()</code>	如果不成功, 就是无任何作用
<code>remove()</code>	只要元素比较操作不抛出异常, 它就不抛出异常
<code>remove_if()</code>	只要判断式 <i>predicate</i> 不抛出异常, 它就不抛出异常
<code>unique()</code>	只要元素比较操作不抛出异常, 它就不抛出异常
<code>splice()</code>	不抛出异常
<code>merge()</code>	只要元素比较时不抛出异常, 它便保证“要么不成功, 要么无任何作用”
<code>reverse()</code>	不抛出异常
<code>swap()</code>	不抛出异常

6.4.4 Lists 运用实例

下面这个例子突显出 list 特殊成员函数的用法:

```
// cont/list1.cpp

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

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }
    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(), list2.end(), // destination position
                     3),
                list1);                          // source list
    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(), // destination position
                list2,        // source list
                list2.begin()); // source position
    printLists(list1, list2);

    // sort second list, assign to list1 and remove duplicates
    list2.sort();
    list1 = list2;
```

```
list2.unique();  
printLists(list1, list2);  
  
// merge both sorted lists into the first list  
list1.merge(list2);  
printLists(list1, list2);  
}
```

程序输出如下:

```
list1: 0 1 2 3 4 5  
list2: 5 4 3 2 1 0  
  
list1:  
list2: 5 4 0 1 2 3 4 5 3 2 1 0  
  
list1:  
list2: 4 0 1 2 3 4 5 3 2 1 0 5  
  
list1: 0 0 1 1 2 2 3 3 4 4 5 5  
list2: 0 1 2 3 4 5  
  
list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5  
list2:
```

6.5 Sets 和 Multisets

`set` 和 `multiset` 会根据特定的排序准则，自动将元素排序。两者不同之处在于 `multisets` 允许元素重复而 `sets` 不允许（请参考 6.6 节和第 5 章关于本主题的讨论）。

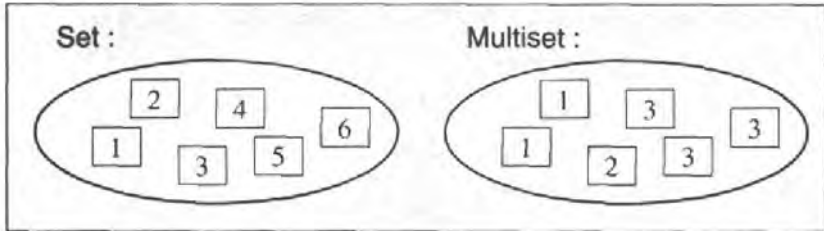


图 6.6 Sets 和 Multisets

使用 `set` 或 `multiset` 之前，必须先含入头文件 `<set>`¹⁴：

```
#include <set>
```

在这个头文件中，上述两个型别都被定义为命名空间 `std` 内的 `class templates`：

```
namespace std {
    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class set;

    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class multiset;
}
```

只要是 `assignable`、`copyable`、`comparable`（根据某个排序准则）的型别 `T`，都可以成为 `set` 或 `multiset` 的元素型别。可有可无的第二个 `template` 参数用来定义排序准则。如果没有传入特别的排序准则，就采用缺省准则 `less`——这是一个仿函数，以 `operator<` 对元素进行比较，以便完成排序（`less`¹⁵的内容详见 p305）。可有可无的第三参数用来定义内存模型（参见第 15 章）。缺省的内存模型是

¹⁴ 早期 STL 中，`sets` 的头文件是 `<set.h>`，`multisets` 的头文件是 `<multiset.h>`

¹⁵ 在不支持 `default template arguments` 的系统中，第二参数通常会被省略。

allocator, 由 C++ 标准程序库提供¹⁶。

所谓“排序准则”，必须定义 **strict weak ordering**，其意义如下：

1. 必须是“反对称的 (antisymmetric)”。

对 `operator<` 而言，如果 $x < y$ 为真，则 $y < x$ 为假。

对判断式 `predicate op()` 而言，如果 `op(x, y)` 为真，则 `op(y, x)` 为假。

2. 必须是“可传递的 (transitive)”。

对 `operator<` 而言，如果 $x < y$ 为真且 $y < z$ 为真，则 $x < z$ 为真。

对判断式 `op()` 而言，如果 `op(x, y)` 为真且 `op(y, z)` 为真，则 `op(x, z)` 为真。

3. 必须是“非自反的 (irreflexive)”。

对 `operator<` 而言， $x < x$ 永远为假。

对判断式 `predicate op()` 而言，`op(x, x)` 永远为假。

基于这些特性，排序准则也可用于相等性检验，也就是说，如果两个元素都不小于对方（或说 `op(x, y)` 和 `op(y, x)` 都为假），则两个元素相等。

译注：以上种种性质（及其它各种相关性质）是 STL 学术理论的一部分。STL 先建立起一个抽象概念阶层体系，形成一个软件组件分类学，最后再以实际工具 (C++ template) 将各个概念实作出来。这些理论架构的最佳描述书籍是《*Generic Programming and the STL*》, by Matthew H. Austern, Addison Wesley 1998；中译本《泛型程序设计与 STL》，侯捷/黄俊尧合译，暮峰 2000。

6.5.1 Sets 和 Multisets 的能力

和所有标准关联式容器类似，sets 和 multisets 通常以平衡二叉树 (balanced binary tree, 图 6.7) 完成。C++ 标准规格书并未明定，但由 sets 和 multisets 各项操作的复杂度可以得出这样的结论¹⁷。

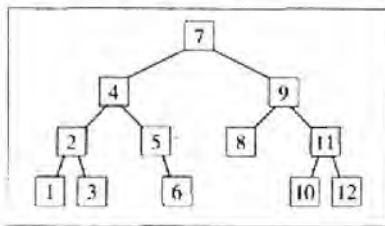


图 6.7 Sets 和 Multisets 的内部结构

¹⁶ 在不支持 default template arguments 的系统中，第三参数通常会被省略。

¹⁷ 事实上 sets 和 multisets 通常以红黑树 (red-black tree) 实作而成。红黑树在改变元素数量和元素搜寻方面都很出色，它保证节点安插时最多只会作两个重新连接 (relink) 动作，而且到达某一元素的最长路径深度，最多只是最短路径深度的两倍。

自动排序的主要优点在于使二叉树于搜寻元素时具有良好性能。其搜寻函数算法具有对数 (logarithmic) 复杂度。在拥有 1000 个元素的 sets 或 multisets 中搜寻元素, 二叉树搜寻动作 (由成员函数执行) 的平均时间为线性搜寻 (由 STL 算法执行) 时间的 1/50。关于复杂度的讨论, 详见 p21, 2.3 节。

但是, 自动排序造成 sets 和 multisets 的一个重要限制: 你不能直接改变元素值, 因为这样会打乱原本正确的顺序。因此, 要改变元素值, 必须先删除旧元素, 再插入新元素。这里提供的接口正反映了这种行为:

- sets 和 multisets 不提供用来直接存取元素的任何操作函数。
- 通过迭代器进行元素间接存取, 有一个限制: 从迭代器的角度来看, 元素值是常数。

6.5.2 Sets 和 Multisets 的操作函数

生成 (Create)、复制 (Copy) 和销毁 (Destroy)

表 6.20 列出 sets 和 multisets 的构造函数和析构函数。

表 6.20 Sets 和 Multisets 的构造函数和析构函数

操作	效果
<code>set c</code>	产生一个空的 <code>set/multiset</code> , 其中不含任何元素
<code>set c(op)</code>	以 <code>op</code> 为排序准则, 产生一个空的 <code>set/multiset</code>
<code>set c1(c2)</code>	产生某个 <code>set/multiset</code> 的副本, 所有元素均被复制
<code>set c(beg, end)</code>	以区间 <code>[beg; end]</code> 内的元素产生一个 <code>set/multiset</code>
<code>set c(beg, end, op)</code>	以 <code>op</code> 为排序准则, 利用 <code>[beg; end]</code> 内的元素生成一个 <code>set/multiset</code>
<code>c.~set()</code>	销毁所有元素, 释放内存

其中 `set` 可为下列形式:

<code>set</code>	效果
<code>set<Elem></code>	一个 <code>set</code> , 以 <code>less<> (operator<)</code> 为排序准则
<code>set<Elem, Op></code>	一个 <code>set</code> , 以 <code>op</code> 为排序准则
<code>multiset<Elem></code>	一个 <code>multiset</code> , 以 <code>less<> (operator<)</code> 为排序准则
<code>multiset<Elem, Op></code>	一个 <code>multiset</code> , 以 <code>op</code> 为排序准则

有两种方式可以定义排序准则：

1. 以 `template` 参数定义之。

例如¹⁸：

```
std::set<int, std::greater<int> > coll;
```

这种情况下，排序准则就是型别的一部分。因此型别系统确保“只有排序准则相同的容器才能被合并”。这是排序准则的通常指定法。更精确地说，第二个参数是排序准则的型别，实际的排序准则是容器所产生的函数对象（*function object*，或称 *functor*）。为了产生它，容器构造函数会调用“排序准则型别”的 `default` 构造函数。p294 有一个“使用者自定之排序准则”的运用实例。

2. 以构造函数参数定义之。

这种情况下，同一个型别可以运用不同的排序准则，而排序准则的初始值或状态也可以不同。如果执行期才获得排序准则，而且需要用到不同的排序准则（但数据型别必须相同），此一方式可派上用场。p191 有个完整例子。

如果使用者没有提供特定的排序准则，那么就采用缺省准则——仿函数 `less<>`。`less<>` 系透过 `operator<` 对元素进行排序¹⁹。

请注意，排序准则也被用于元素相等性检验工作。当采用缺省排序准则时，两元素的相等性检验语句如下：

```
if (! (elem1 < elem2 || elem2 < elem1))
```

这样做有三点好处：

1. 只需传递一个参数作为排序准则。
2. 不必针对元素型别提供 `operator==`。
3. 你可以对“相等性”有截然相反的定义（即使表达式中 `operator==` 的行为有所不同，也无关紧要）。不过当心造成混淆。

这种相等性检验方式会花费比较长的时间，因为评估上述表达式可能需要两次比较。注意，如果第一次比较结果为 `true`，就不用进行第二次比较了。

看到这里，如果容器型别名称让你很烦，采用“型别定义式”不失为一个好办法。在使用容器型别（以及迭代器型别）的任何地方都可以采用这种便捷之道，例

¹⁸ 注意，两个 `>` 之间需加上一个空格，因为 `>>` 会被编译器视为移位操作符，导致此处语法错误。

¹⁹ 在不支持 `default template parameters` 的系统中，通常必须这么设定排序准则：
`set<int, less<int> > coll;`

如：

```
typedef std::set<int, std::greater<int> > IntSet;
...
IntSet coll
IntSet::iterator pos;
```

“利用区间的起点和终点来构造容器”的构造函数，可以从其它型别的容器中，或是从 array、或是从标准输入装置（standard input）中接受元素来源。详见 6.1.2 节，p144。

非变动性操作（Nonmodifying Operations）

sets 和 multisets 提供常见的非变动性操作，用来查询大小、相互比较。

表 6.21 Sets 和 Multisets 的非变动性操作（Nonmodifying Operations）

操作	效果
c.size()	返回容器的大小
c.empty()	判断容器大小是否为零。等同于 size()==0，但可能更快
c.max_size()	返回可容纳的最大元素数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2。等同于 !(c1 == c2)
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2。等同于 c2 < c1
c1 <= c2	判断是否 c1 小于等于 c2。等同于 !(c2 < c1)
c1 >= c2	判断是否 c1 大于等于 c2。等同于 !(c1 < c2)

元素比较动作只能用于型别相同的容器。换言之，元素和排序准则必须有相同的型别，否则编译时期会产生型别方面的错误。

```
std::set<float> c1;    // sorting criterion: std::less<>
std::set<float, std::greater<float> > c2;
...
if (c1 == c2) {        // ERROR: different types
    ...
}
```

比较动作系以“字典（lexicographical）顺序”来检查某个容器是否小于另一个容器。如果要比较不同型别（拥有不同排序准则）的容器，你必须采用 p356, 9.5.4 节的“比较算法（comparing algorithms）”。

特殊的搜寻函数 (Special Search Operations)

sets 和 multisets 在元素快速搜寻方面有优化设计,所以提供了特殊的搜寻函数,如表 6.22。这些函数是同名的 STL 算法的特殊版本。面对 sets 和 multisets,你应该优先采用这些优化算法,如此可获得对数复杂度,而非 STL 算法的线性复杂度。举个例子,在 1,000 个元素中搜寻,平均 10 次比较之后便可得出结果,如果是线性复杂度,平均 500 次比较才能有结果(参见 2.3 节, p21)。

表 6.22 Sets 和 Multisets 的搜寻操作函数

操作	效果
count(elem)	返回“元素值为 elem”的元素个数
find(elem)	返回“元素值为 elem”的第一个元素,如果找不到就返回 end()
lower_bound(elem)	返回 elem 的第一个可安插位置,也就是“元素值 >= elem”的第一个元素位置
upper_bound(elem)	返回 elem 的最后一个可安插位置,也就是“元素值 > elem”的第一个元素位置
equal_range(elem)	返回 elem 可安插的第一个位置和最后一个位置,也就是“元素值 == elem”的元素区间

成员函数 find() 搜寻出与参数值相同的第一个元素,并返回一个迭代器,指向该位置。如果没找到这样的元素,就返回容器的 end()。

lower_bound() 和 upper_bound() 分别返回元素可安插点的第一个和最后一个位置。换言之,lower_bound() 返回大于等于参数值的第一个元素所处位置,upper_bound() 返回大于参数值的第一个元素位置。equal_range() 则是将 lower_bound() 和 upper_bound() 的返回值做成一个 pair 返回(型别 pair 在 p33, 4.1 节介绍),所以它返回的是“与参数值相等”的元素所形成的区间。如果 lower_bound() 或“equal_range() 的第一值”等于“equal_range() 的第二值”或 upper_bound(), 则此 sets 或 multisets 内不存在相同数值的元素。这是当然啦,同值区间中至少也得包含一个元素嘛!

下面的例子说明如何使用 lower_bound(), upper_bound() 和 equal_range():

```
// cont/set2.cpp

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

```
int main ()
{
    set<int> c;

    c.insert(1);
    c.insert(2);
    c.insert(4);
    c.insert(5);
    c.insert(6);

    cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
    cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
    cout << "equal_range(3): " << *c.equal_range(3).first << " "
        << *c.equal_range(3).second << endl;

    cout << endl;
    cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
    cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
    cout << "equal_range(5): " << *c.equal_range(5).first << " "
        << *c.equal_range(5).second << endl;
}
```

程序输入如下:

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4

lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

上例如果使用 multisets 而不是 sets, 程序输出相同。

赋值 (Assignments)

sets 和 multisets 只提供所有容器都提供的基本赋值操作 (表 6.23), 详见 p147。

这些操作函数中, 赋值操作的两端容器必须具有相同型别。尽管“比较准则”本身可能不同, 但其型别必须相同。p191 列出一个“排序准则不同, 但型别相同”的例子。如果准则不同, 准则本身也会被赋值 (assigned) 或交换 (swapped)。

表 6.23 Sets 和 Multisets 的赋值操作

操作	效果
<code>c1 = c2</code>	将 <code>c2</code> 中所有元素赋值给 <code>c1</code>
<code>c1.swap(c2)</code>	将 <code>c1</code> 和 <code>c2</code> 的元素互换
<code>swap(c1, c2)</code>	同上。此为全局函数

迭代器相关函数 (Iterator Functions)

`sets` 和 `multisets` 不提供元素直接存取，所以只能采用迭代器。`Sets` 和 `multisets` 也提供了一些常见的迭代器函数（表 6.24）。

表 6.24 Sets 和 multisets 的迭代器相关操作函数

操作	效果
<code>c.begin()</code>	返回一个双向迭代器（将元素视为常数），指向第一元素
<code>c.end()</code>	返回一个双向迭代器（将元素视为常数），指向最后元素的下一位置
<code>c.rbegin()</code>	返回一个逆向迭代器，指向逆向遍历时的第一个元素
<code>c.rend()</code>	返回一个逆向迭代器，指向逆向遍历时的最后元素的下一位置

和其它所有关联式容器类似，这里的迭代器是双向迭代器（参见 p255, 7.2.4 节）。所以，对于只能用于随机存取迭代器的 STL 算法（例如排序或随机乱序 *random shuffling* 算法），`sets` 和 `multisets` 就无福消受了。

更重要的是，对迭代器操作而言，所有元素都被视为常数，这可确保你不会人为改变元素值，从而打乱既定顺序。然而这也使得你无法对 `sets` 或 `multisets` 元素调用任何变动性算法（*modifying algorithms*）。例如你不能对它们调用 `remove()`，因为 `remove()` 算法实际上是以一个参数值覆盖被移除的元素（详细讨论见 p115, 5.6.2 节）。如果要移除 `sets` 和 `multisets` 的元素，你只能使用它们所提供的成员函数。

元素的安插 (Inserting) 和移除 (Removing)

表 6.25 列出 `sets` 和 `multisets` 的元素安插和删除函数。

按 STL 惯例，你必须保证参数有效：迭代器必须指向有效位置、序列起点不能位于终点之后、不能从空容器中删除元素。

安插和移除多个元素时，单一调用（一次处理）比多次调用（逐一处理）快得多。

表 6.25 Sets 和 Multisets 的元素安插和移除

操作	效果
<code>c.insert(elem)</code>	安插一份 <code>elem</code> 副本, 返回新元素位置 (不论是否成功——对 <code>sets</code> 而言)
<code>c.insert(pos, elem)</code>	安插一份 <code>elem</code> 副本, 返回新元素位置 (<code>pos</code> 是个提示, 指出安插操作的搜寻起点。如果提示恰当, 可大大加快速度)
<code>c.insert(beg, end)</code>	将区间 <code>[beg; end]</code> 内所有元素的副本安插到 <code>c</code> (无返回值)
<code>c.erase(elem)</code>	移除 “与 <code>elem</code> 相等” 的所有元素, 返回被移除的元素个数
<code>c.erase(pos)</code>	移除迭代器 <code>pos</code> 所指位置上的元素, 无返回值
<code>c.erase(beg, end)</code>	移除区间 <code>[beg; end]</code> 内的所有元素, 无返回值
<code>c.clear()</code>	移除全部元素, 将整个容器清空

注意, 安插函数的返回值型别不尽相同:

- `sets` 提供如下接口:

```
pair<iterator, bool> insert(const value_type& elem);
iterator            insert(iterator pos_hint,
                           const value_type& elem);
```

- `multisets` 提供如下接口:

```
iterator insert(const value_type& elem);
iterator insert(iterator pos_hint,
               const value_type& elem);
```

返回值型别不同的原因是: `multisets` 允许元素重复, 而 `sets` 不允许。因此如果将某元素安插至一个 `set` 内, 而该 `set` 已经内含同值元素, 则安插操作将告失败。所以 `set` 的返回值型别是以 `pair` 组织起来的两个值 (关于 `pair` 详见 p33, 4.1 节):

1. `pair` 结构中的 `second` 成员表示安插是否成功。
2. `pair` 结构中的 `first` 成员返回新元素的位置, 或返回现存的同值元素的位置。

其它任何情况下, 函数都返回新元素位置 (如果 `sets` 已经内含同值元素, 则返回同值元素的位置)。

以下例子把数值 3.3 的元素安插到 `set c` 中, 藉此说明如何使用上述接口:

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```


如果你还想处理新位置或旧位置，程序代码得更复杂些：

```
// define variable for return value of insert()
std::pair<std::set<float>::iterator,bool> status;

// insert value and assign return value
status = c.insert(value);

// process return value
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1
           << std::endl;
```

对于此一序列的两次调用结果可能如下：

```
8.9 inserted as element 4
7.7 already exists as element 3
```

注意，所有拥有“位置提示参数”的安插函数，其返回值型别都一样，不论是 `sets` 或 `multisets`，这些函数都只返回一个迭代器。这些函数的效果与“无位置提示参数”的函数一样，只不过性能略有差异。你可以传进一个迭代器，该位置将作为一个提示，用来提升性能。事实上如果被安插元素的位置恰好紧贴于提示位置之后，那么时间复杂度就会从“对数”一变而为“分期摊还常数 (amortized constant)”（复杂度的介绍请见 p21, 2.3 节）。和“单参数安插函数”不同的是，带有“额外提示位置”的若干安插函数，都具有相同的返回值型别，这就确保你至少有了一个通用型安插函数，在各种容器中有共同接口。事实上通用型安插迭代器 (general inserters) 就是靠这个接口的支持才得以实现的。

要删除“与某值相等”的元素，只需调用 `erase()`：

```
std::set<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

和 `lists` 不同的是, `erase()` 并非取名为 `remove()` (后者的讨论请见 p170)。是的, 它的行为不同, 它返回被删除元素的个数, 用在 `sets` 身上, 返回值非 0 即 1。

如果 `multisets` 内含重复元素, 你不能使用 `erase()` 来删除这些重复元素中的第一个。你可以这么做:

```
std::multiset<Elem> coll;
...
// remove first element with passed value
std::multiset<Elem>::iterator pos;
pos = coll.find(elem);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

这里应该采用成员函数 `find()`, 而非 STL 算法 `find()`, 因为前者速度更快(参见 p154 的例子)。

注意, 还有一个返回值不一致的情况。作用于序列式容器和关联式容器的 `erase()` 函数, 其返回值有以下不同:

1. 序列式容器提供下面的 `erase()` 成员函数:

```
iterator erase(iterator pos);
iterator erase(iterator beg, iterator end);
```

2. 关联式容器提供下面的 `erase()` 成员函数:

```
void erase(iterator pos);
void erase(iterator beg, iterator end);
```

存在这种差别, 完全是为了性能。在关联式容器中“搜寻某元素并返回后继元素”可能颇为耗时, 因为这种容器的底部是以二叉树完成, 所以如果你想编写对所有容器都适用的程序代码, 你必须忽略返回值。

6.5.3 异常处理 (Exception Handling)

`sets` 和 `multisets` 是“以节点 (nodes) 为基础”的容器。如果节点构造失败, 容器仍保持原样。此外, 由于析构函数通常并不抛出异常, 所以节点的移除不可能失败。

然而, 面对多重元素安插操作, “保持元素次序”这一条件会造成“异常抛出时能够完全复原”这一需求变得不切实际。因此只有“单一元素安插操作”才支持“成功, 否则无效”的操作原则。至于“多元素删除操作”总是能够成功。如果排序准则之复制/赋值操作会抛出异常, 则 `swap()` 也会抛出异常。

STL 异常处理的一般性讨论见于 p139, 5.11.2 节。p248 的 6.10.10 节列出“异常出现时会给予特殊保证”的所有容器操作函数。

6.5.4 Sets 和 Multisets 运用实例

以下程序展示 sets 的一些能力²⁰：

```
// cont/set1.cpp

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

int main()
{
    /* type of the collection:sets;
     * - no duplicates
     * - elements are integral values
     * - descending order
     */
    typedef set<int,greater<int> > IntSet;

    IntSet coll1; // empty set container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
```

²⁰ distance() 的定义已有改变。早期 STL 版本中，你必须含入 distance.hpp (见 p263)

```
// insert 4 again and process return value
pair<IntSet::iterator,bool> status = coll1.insert(4);
if (status.second) {
    cout << "4 inserted as element "
         << distance(coll1.begin(),status.first) + 1
         << endl;
}
else {
    cout << "4 already exists" << endl;
}

// assign elements to another set with ascending order
set<int> coll2(coll1.begin(),
              coll1.end());

// print all elements of the copy
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout," "));
cout << endl;

// remove all elements up to element with value 3
coll2.erase (coll2.begin(), coll2.find(3));

// remove all elements with value 5
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;

// print all elements
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout," "));
cout << endl;
}
```

首先，以下的型别定义：

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

定义了一个 `set`，其中容纳降序（递减）排列的 `ints`。产生一个空的 `sets` 之后，首先利用 `insert()` 安插数个元素：

```
IntSet coll1;  
  
coll1.insert(4);  
...
```

注意数值为 5 的元素被安插两次，但第二次安插操作会被程序忽略，因为 sets 不允许数值重复的元素。

打印所有元素后，程序再次安插元素 4。这次按照 p183 的方法来处理 insert() 返回值。

以下语句：

```
set<int> coll2(coll1.begin(), coll1.end());
```

产生一个新的 set，其中容纳升序（递增）排列的 ints，并以原本那个 sets 的元素作为初值²¹。

两个容器有不同的排序准则，所以它们的型别不同，不能直接相互赋值或比较。但只要元素型别相同或彼此可以转型，你就可以使用某些“有能力处理不同容器型别”的算法来达到目的。

以下语句：

```
coll2.erase(coll2.begin(), coll2.find(3));
```

移除了数值为 3 的元素之前的所有元素。注意数值为 3 的元素位于序列尾端，所以没被移除。

最后，所有数值为 5 的元素都被移除：

```
int num;  
num = coll2.erase(5);  
cout << num << " element(s) removed" << endl;
```

程序输出如下：

```
6 5 4 3 2 1  
4 already exists  
1 2 3 4 5 6  
1 element(s) removed  
3 4 6
```

21 这行语句需要两个语言新性质：member templates 和 default template arguments。如果系统不支持，你必须改成这样：

```
set<int, less<int> > coll2;  
copy(coll1.begin(), coll1.end(),  
      inserter(coll2, coll2.begin()));
```

对于 multisets, 上一个程序需要些微改变, 并产生不同结果:

```
// cont/mset1.cpp

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

int main()
{
    /* type of the collection: sets
     * - duplicates allowed
     * - elements are integral values
     * - descending order
     */
    typedef multiset<int,greater<int> > IntSet;

    IntSet coll1; // empty multiset container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;

    // insert 4 again and process return value
    IntSet::iterator ipos = coll1.insert(4);
    cout << "4 inserted as element "
        << distance(coll1.begin(),ipos) + 1
        << endl;
```

```
// assign elements to another multiset with ascending order
multiset<int> coll2(coll1.begin(),
                  coll1.end());

// print all elements of the copy
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;

// remove all elements up to element with value 3
coll2.erase (coll2.begin(), coll2.find(3));

// remove all elements with value 5
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;

// print all elements
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

这个程序把所有的 `set` 都改为 `multiset`，此外 `insert()` 返回值的处理也有所不同：

```
IntSet::iterator ipos = coll1.insert(4);
cout << "4 inserted as element "
     << distance(coll1.begin(), ipos) + 1
     << endl;
```

由于 `multisets` 可能包含重复元素，所以安插操作只在异常被抛出时才失败。因此，返回值型别只是一个迭代器，指向新元素位置。

程序输出如下：

```
6 5 5 4 3 2 1
4 inserted as element 5
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6
```

6.5.5 执行期指定排序准则

无论是将排序准则作为第二个 `template` 参数传入，或是采用缺省的排序准则 `less<>`，通常你都会将排序准则定义为型别的一部分。但有时必须在执行期处理排序准则，或者你可能需要对同一种数据型别采用不同的排序准则。此时你就需要一个“用来表现排序准则”的特殊型别，使你能够在执行期间传递某个准则。以下范例程序说明了这种做法：

```
// cont/setcmp.cpp

#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// type for sorting criterion
template <class T>
class RuntimeCmp {
public:
    enum cmp_mode {normal, reverse};

private:
    cmp_mode mode;

public:
    // constructor for sorting criterion
    // - default criterion uses value normal
    RuntimeCmp (cmp_mode m=normal) : mode(m) {
    }

    // comparison of elements
    bool operator() (const T& t1, const T& t2) const {
        return mode == normal ? t1 < t2 : t2 < t1;
    }

    // comparison of sorting criteria
    bool operator==(const RuntimeCmp& rc) {
        return mode == rc.mode;
    }
};

// type of a set that uses this sorting criterion
typedef set<int, RuntimeCmp<int> > IntSet;
```



```
// forward declaration
void fill (IntSet& set);

int main()
{
    // create, fill, and print set with normal element order
    // - uses default sorting criterion
    IntSet coll1;
    fill(coll1);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // create sorting criterion with reverse element order
    RuntimeCmp<int> reverse_order(RuntimeCmp<int>::reverse);

    // create, fill, and print set with reverse element order
    IntSet coll2(reverse_order);
    fill(coll2);
    PRINT_ELEMENTS (coll2, "coll2: ");

    // assign elements AND sorting criterion
    coll1 = coll2;
    coll1.insert(3);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // just to make sure...
    if (coll1.value_comp() == coll2.value_comp()) {
        cout << "coll1 and coll2 have same sorting criterion"
              << endl;
    }
    else {
        cout << "coll1 and coll2 have different sorting criterion"
              << endl;
    }
}

void fill (IntSet& set)
{
    // fill insert elements in random order
    set.insert(4);
    set.insert(7);
    set.insert(5);
    set.insert(1);
    set.insert(6);
    set.insert(2);
    set.insert(5);
}
```

在这个程序中, `RuntimeCmp<>` 是一个简单的 `template`, 提供“执行期间面对任意型别定义一个排序准则”的泛化能力。其 `default` 构造函数采用默认值 `normal`, 按升序排序; 你也可以将 `RuntimeCmp<>::reverse` 传递给构造函数, 便能按降序排序。

程序输出如下:

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have same sorting criterion
```

注意, `coll1` 和 `coll2` 拥有相同型别, 该型别即 `fill()` 函数的参数型别。再请注意, `assignment` 操作符不仅赋值了元素, 也赋值了排序准则 (否则任何一个赋值操作岂不会轻易危及排序准则!)。

6.6 Maps 和 Multimaps

Map 和 multimap 将 *key/value* pair (键值/实值 对组) 当做元素, 进行管理。它们可根据 *key* 的排序准则自动将元素排序。multimaps 允许重复元素, maps 不允许, 见图 6.8。

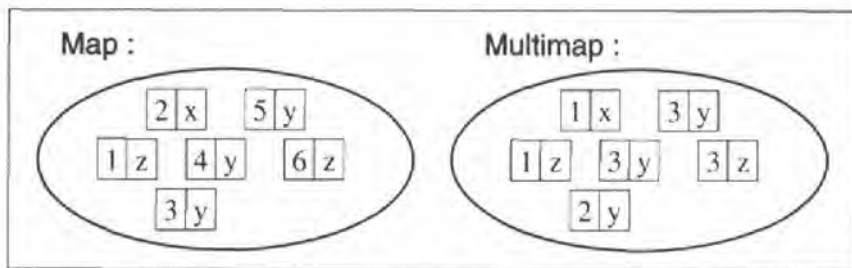


图 6.8 Maps 和 Multimaps

使用 map 和 multimap 之前, 你必须先含入头文件 `<map>`²²:

```
#include <map>
```

在其中, map 和 multimap 被定义为命名空间 std 内的 class templates:

```
namespace std {  
    template <class Key, class T,  
              class Compare = less<Key>,  
              class Allocator = allocator<pair<const Key,T> > >  
        class map;  
  
    template <class Key, class T,  
              class Compare = less<Key>,  
              class Allocator = allocator<pair<const Key,T> > >  
        class multimap;  
}
```

第一个 template 参数被当做元素的 *key*, 第二个 template 参数被当做元素的 *value*。Map 和 multimap 的元素型别 Key 和 T, 必须满足以下两个条件:

²² 在早期 STL 中, maps 被定义于 `<map.h>` 而 multimaps 被定义于 `<multimap.h>`。

- 1. *key/value* 必须具备 assignable (可赋值的) 和 copyable (可复制的) 性质。
- 2. 对排序准则而言, *key* 必须是 comparable (可比较的)。

第三个 `template` 参数可有可无, 用它来定义排序准则。和 `sets` 一样, 这个排序准则必须定义为 **strict weak ordering** (参见 p176)。元素的次序由它们的 *key* 决定, 和 *value* 无关。排序准则也可以用来检查相等性: 如果两个元素的 *key* 彼此都不小于对方, 则两个元素被视为相等。如果使用者未传入特定排序准则, 就使用缺省的 `less` 排序准则——以 `operator<` 来进行比较²³(`less` 的详细资料请见 p305)。

第四个 `template` 参数也是可有可无, 用它来定义内存模型 (详见第 15 章)。缺省的内存模型是 `allocator`, 由 C++ 标准程序库提供²⁴。

6.6.1 Maps 和 Multimaps 的能力

和所有标准的关联式容器一样, `maps/multimaps` 通常以平衡二叉树完成, 如图 6.9。

标准规范并未明定这一点, 但是从 `map` 和 `multimap` 各项操作的复杂度可以得出这一结论。典型情况下, `set`, `multisets`, `map`, `multimaps` 使用相同的内部数据结构。因此你可以把 `set` 和 `multisets` 分别视为特殊的 `map` 和 `multimaps`, 只不过 `sets` 元素的 *value* 和 *key* 是指同一对象。因此 `map` 和 `multimaps` 拥有 `set` 和 `multisets` 的所有能力和所有操作函数。当然某些细微差异还是有的: 首先, 它们的元素是 *key/value pair*, 其次, `map` 可作为关联式数组来运用。

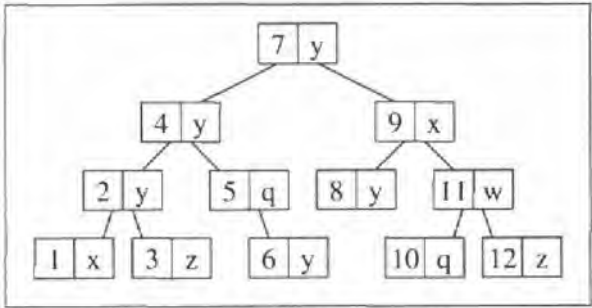


图 6.9 Maps 和 Multimaps 的内部结构

²³ 在不支持 `default template parameters` 的系统中, 第三个参数通常会被省略。

²⁴ 在不支持 `default template parameters` 的系统中, 第四个参数通常会被省略。

Map 和 multimaps 根据元素的 *key* 自动对元素进行排序。这么一来, 根据已知的 *key* 搜寻某个元素时, 就能够有很好的性能, 而根据已知 *value* 搜寻元素时, 性能就很糟糕。“自动排序”这一性质使得 map 和 multimaps 身上有了一条重要的限制: 你不可以直接改变元素的 *key*, 因为这会破坏正确次序。要修改元素的 *key*, 你必须先移除拥有该 *key* 的元素, 然后插入拥有新的 *key/value* 的元素 (详见 p201)。从迭代器的观点来看, 元素的 *key* 是常数。至于元素的 *value* 倒是可以直接修改的, 当然, 前提是 *value* 并非常数型态。

6.6.2 Maps 和 Multimaps 的操作函数

生成 (Create)、复制 (Copy) 和销毁 (Destroy)

表 6.26 列出 maps 和 multimaps 的生成、复制、销毁等各项操作。

表 6.26 Maps 和 Multimaps 的构造函数和析构函数

操作	效果
<i>map</i> c	产生一个空的 map/multimap, 其中不含任何元素
<i>map</i> c (op)	以 op 为排序准则, 产生一个空的 map/multimap
<i>map</i> c1 (c2)	产生某个 map/multimap 的副本, 所有元素均被复制
<i>map</i> c (beg, end)	以区间 [beg; end] 内的元素产生一个 map/multimap
<i>map</i> c (beg, end, op)	以 op 为排序准则, 利用 [beg; end] 内的元素生成一个 map/multimap
c.~ <i>map</i> {}	销毁所有元素, 释放内存

其中, *map* 可为下列型式:

<i>map</i>	效果
map<Key, Elem>	一个 map, 以 less<> (operator<) 为排序准则
map<Key, Elem, Op>	一个 map, 以 op 为排序准则
multimap<Key, Elem>	一个 multimap, 以 less<> (operator<) 为排序准则
multimap<Key, Elem, Op>	一个 multimap, 以 op 为排序准则

有两种方式可以定义排序准则：

1. 以 `template` 参数定义。

例如²⁵：

```
std::map<float, std::string, std::greater<float> > > coll;
```

这种情况下，排序准则就是型别的一部分。因此型别系统确保“只有排序准则相同的容器才能被合并”。这是比较常见的排序准则指定法。更精确地说，第三参数是排序准则的型别。实际的排序准则是容器所产生的函数对象（*function object*，或称 *functor*）。为了产生它，容器构造函数会调用“排序准则型别”的 `default` 构造函数。p294 有一个“使用者自定之排序准则”的运用实例。

2. 以构造函数参数定义。

在这种情况下，你可以有一个“排序准则型别”并为它指定不同的排序准则（也就是说让该型别所产生出来的对象（代表一个排序准则）的初值或状态不同）。如果执行期才获得排序准则，而且程序需要用到不同的排序准则（但其数据型别必须相同），此一方式可派上用场。一个典型的例子是在执行期指定“*key* 的型别为 `string`”的排序准则。完整例子见 p213。

如果使用者没有提供特定排序准则，就采用缺省准则——仿函数 `less<>`。`less<>` 系透过 `operator<` 对元素进行排序²⁶。

你应当做一些型别定义（`typedef`），从而简化繁琐的型别表达式：

```
typedef std::map<std::string,float,std::greater<std::string> >
        StringFloatMap;
...
StringFloatMap coll;
```

某些构造函数使用区间起点和终点作为参数，它们可以使用不同型别的容器、数组、标准输入装置（`standard input`）来进行初始化，详见 6.1.2 节，p144。然而由于元素是 *key/value pair*，因此你必须确定来自源区间的元素型别也是 `pair<key, value>`，或至少可转化成 `pair<key, value>`。

非变动性操作（Nonmodifying Operations）

`maps` 和 `multimaps` 提供常见的非变动性操作，用来查询大小、相互比较。如表 6.27。

²⁵ 注意，两个 `>` 之间需加上一个空格，因为 `>>` 会被编译器视为移位操作符，导致本处语法错误。

²⁶ 在不支持 `default template parameters` 的系统中，通常必须这样设定排序准则：

```
map<float, string, less<float> > > coll;
```

表 6.27 Maps 和 Multimaps 的非变动性操作 (Nonmodifying Operations)

操作	效果
<code>c.size()</code>	返回容器的大小。
<code>c.empty()</code>	判断容器大小是否为零。等同于 <code>size()==0</code> ，但可能更快。
<code>c.max_size()</code>	返回可容纳的最大元素数量。
<code>c1 == c2</code>	判断是否 <code>c1</code> 等于 <code>c2</code> 。
<code>c1 != c2</code>	判断是否 <code>c1</code> 不等于 <code>c2</code> 。等同于 <code>!(c1 == c2)</code> 。
<code>c1 < c2</code>	判断是否 <code>c1</code> 小于 <code>c2</code> 。
<code>c1 > c2</code>	判断是否 <code>c1</code> 大于 <code>c2</code> 。等同于 <code>c2 < c1</code> 。
<code>c1 <= c2</code>	判断是否 <code>c1</code> 小于等于 <code>c2</code> 。等同于 <code>!(c2 < c1)</code> 。
<code>c1 >= c2</code>	判断是否 <code>c1</code> 大于等于 <code>c2</code> 。等同于 <code>!(c1 < c2)</code> 。

元素比较动作只能用于型别相同的容器。换言之，容器的 *key*、*value*、排序准则都必须有相同的型别，否则编译期会产生型别方面的错误。例如：

```
std::map<float,std::string> c1;        // sorting criterion: less<>
std::map<float,std::string,std::greater<float> > c2;
...
if (c1 == c2) { // ERROR: different types
    ...
}
```

比较动作系以“字典 (lexicographical) 顺序”来检查某个容器是否小于另一个容器 (详见 p360)。如果要比较不同型别 (拥有不同排序准则) 的容器，你必须采用 p356, 9.5.4 节的“比较算法 (comparing algorithms)”。

特殊的搜寻动作 (Special Search Operations)

就像 `set` 和 `multisets` 一样，`map` 和 `multimaps` 也提供特殊的搜寻函数，以便利用内部树状结构获取较好的性能，见表 6.28。

成员函数 `find()` 用来搜寻拥有某个 *key* 的第一个元素，并返回一个迭代器，指向该位置。如果没找到这样的元素，就返回容器的 `end()`。你不能以 `find()` 搜寻拥有某特定 *value* 的元素，你必须改用通用算法如 `find_if()`，或干脆写一个显式循环。下面这个例子便是利用一个简单循环，对拥有特定 *value* 的所有元素进行某项操作：

```
std::multimap<std::string,float> coll;
...
// do something with all elements having a certain value
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
```

表 6.28 Maps 和 Multimaps 的特殊搜寻操作函数

操作	效果
count(key)	返回“键值等于 key”的元素个数
find(key)	返回“键值等于 key”的第一个元素，找不到就返回 end()
lower_bound(key)	返回“键值为 key”之元素的第一个可安插位置，也就是“键值 >= key”的第一个元素位置
upper_bound(key)	返回“键值为 key”之元素的最后一个可安插位置，也就是“键值 > key”的第一个元素位置
equal_range(key)	返回“键值为 key”之元素的第一个可安插位置和最后一个可安插位置，也就是“键值 == key”的元素区间

```
if (pos->second == value) {
    do_something();
}
```

当你要以此类循环来移除元素时，请特别当心。因为可能会发生一些意料之外的事。细节详见 p204。如果使用 find_if() 算法做类似的搜寻操作，会比写一个循环更复杂，因为你必须提供仿函数 (functor，亦即函数对象 function object)，将元素的 value 拿来和某个 value 比较。详见 p211 实例。

至于 lower_bound(), upper_bound(), equal_range(), 其行为和 sets (见 p180) 的相应函数十分相似，唯一的不同就是：元素是个 key/value pair。

赋值 (Assignments)

maps 和 multimaps 只支持所有容器都提供的基本赋值操作 (表 6.29)，详见 p147。

表 6.29 Maps 和 Multimaps 的赋值 (赋值) 操作

操作	效果
c1 = c2	将 c2 中所有元素赋值给 c1
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1, c2)	同上。此为全局函数

这些操作函数中，赋值操作的两端容器必须具有相同型别。尽管“比较准则”本身可能不同，但其型别必须相同。p213 列出一个“排序准则不同，但型别相同”的例子。如果准则不同，准则本身也会被赋值 (assigned) 或交换 (swapped)。

迭代器函数 (Iterator Functions) 和元素存取 (Element Access)

Map 和 multimap 不支持元素直接存取，因此元素的存取通常是经由迭代器进行。不过有个例外：map 提供 subscript (下标) 操作符，可直接存取元素，详见 6.6.3 节, p205。表 6.30 列出 maps 和 multimap 所支持的迭代器相关函数。

表 6.30 Maps 和 Multimap 的迭代器相关操作函数

操作	效果
c.begin()	返回一个双向迭代器 (key 被视为常数)，指向第一个元素
c.end()	返回一个双向迭代器 (key 被视为常数)，指向最后元素的下一位置
c.rbegin()	返回一个逆向迭代器，指向逆向遍历时的第一个元素
c.rend()	返回一个逆向迭代器，指向逆向遍历时的最后元素的下一位置

和其它所有关联式容器类似，这里的迭代器是双向迭代器(参见 p255, 7.2.4 节)。所以，对于只能用于随机存取迭代器的 STL 算法(例如排序或随机乱序算法 *random shuffling*)，maps 和 multimap 就无福消受了。

更重要的是，在 map 和 multimap 中，所有元素的 key 都被视为常数。因此元素的实质型别是 pair<const key,T>。这个限制是为了确保你不会因为变更元素的 key 而破坏业已排好的元素次序。所以你不能针对 map 或 multimap 调用任何变动性算法(modifying algorithms)。例如你不能对它们调用 remove()，因为 remove() 算法实际上是以一个参数值覆盖被移除的元素(详细讨论见 p115, 5.6.2 节)。如果要移除 map 和 multimap 的元素，你只能使用它们所提供的成员函数。

下面是 map 迭代器运用实例：

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

其中迭代器 pos 遍历了 string/float pair 所组成的序列。以下表达式：

```
pos->first
```

获得元素的 key，而以下表达式：

```
pos->second
```

获得元素的 *value*²⁷。

如果你尝试改变元素的 *key*，会引发错误：

```
pos->first = "hello"; // ERROR at compile time
```

不过如果 *value* 本身的型别并非 `const`，则改变 *value* 没有问题：

```
pos->second = 13.5; // OK
```

如果你一定得改变元素的 *key*，只有一条路：以一个“*value* 相同”的新元素替换掉旧元素。下面是个泛化函数：

```
// cont/newkey.hpp

namespace MyLib {
    template <class Cont>
    inline
    bool replace_key (Cont& c,
                     const typename Cont::key_type& old_key,
                     const typename Cont::key_type& new_key)
    {
        typename Cont::iterator pos;
        pos = c.find(old_key);

        if (pos != c.end()) {
            // insert new element with value of old element
            c.insert(typename Cont::value_type(new_key,
                                                pos->second));

            // remove old element
            c.erase(pos);
            return true;
        }
        else {
            // key not found
            return false;
        }
    }
}
```

关于 `insert()` 和 `erase()` 成员函数，请见下一节讨论。

²⁷ `pos->first` 是 `(*pos).first` 的简写形式。有些程序库只支持后一种形式。

这个泛型函数的用法很简单，把旧的 *key* 和新的 *key* 传递进去就行。例如：

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

如果你面对的是 *multimaps*，情况也一样。

注意，*maps* 提供了一种非常方便的手法让你改变元素的 *key*。只需如此这般：

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

关于 *maps* 的 subscript (下标) 操作符使用细节，详见 6.6.3 节, p205。

元素的安插 (Inserting) 和移除 (Removing)

表 6.31 列出 *maps* 和 *multimaps* 所支持的元素安插和删除函数。

表 6.31 Maps 和 Multimaps 的元素安插和移除

操作	效果
c.insert(elem)	安插一份 elem 副本，返回新元素位置（不论是否成功——对 maps 而言）
c.insert(pos,elem)	安插一份 elem 副本，返回新元素位置（pos 是个提示，指出安插操作的搜寻起点。如果提示恰当，可大大加快速度）
c.insert(beg,end)	将区间 [beg;end] 内所有元素的副本安插到 c（无返回值）
c.erase(elem)	移除“实值 (value) 与 elem 相等”的所有元素，返回被移除的元素个数
c.erase(pos)	移除迭代器 pos 所指位置上的元素，无返回值
c.erase(beg,end)	移除区间 [beg;end] 内的所有元素，无返回值
c.clear()	移除全部元素，将整个容器清空

p182 之中关于 *set* 和 *multisets* 的说明，此处依然适用。上述操作函数的返回值型别有些差异，其情况与 *set* 和 *multisets* 的情况完全相同。当然，这里的元素是 *key/value pair*。所以这里的用法更复杂些。

安插一个 *key/value pair* 的时候，你一定要记住，在 *map* 和 *multimaps* 内部，*key* 被视为常数。你要不就提供正确型别，要不就得提供隐式或显式型别转换。有三个不同的方法可以将 *value* 传入 *map*：

1. 运用 value_type

为了避免隐式类型转换，你可以利用 `value_type` 明白传递正确型别。`value_type` 是容器本身提供的型别定义。例如：

```
std::map<std::string, float> coll;
...
coll.insert(std::map<std::string, float>::value_type("otto",
                                                    22.3));
```

2. 运用 pair<>

另一个作法是直接运用 `pair<>`。例如：

```
std::map<std::string, float> coll;
...
// use implicit conversion:
coll.insert(std::pair<std::string, float>("otto", 22.3));
// use no implicit conversion:
coll.insert(std::pair<const std::string, float>("otto", 22.3));
```

上述第一个 `insert()` 语句内的型别并不正确，所以会被转换成真正的元素型别。为了做到这一点，`insert()` 成员函数被定义为 **member template**²⁸。

3. 运用 make_pair()

最方便的办法是运用 `make_pair()` 函数（详见 p36）。这个函数根据传入的两个参数构造出一个 `pair` 对象：

```
std::map<std::string, float> coll;
...
coll.insert(std::make_pair("otto", 22.3));
```

和作法 2 一样，也是利用 **member template** `insert()` 来执行必要的型别转换。

下面是个简单例子，在 `map` 中安插一个元素，然后检查是否成功：

```
std::map<std::string, float> coll;
...
if (coll.insert(std::make_pair("otto", 22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

²⁸ 如果你的系统不支持 **member template**，你必须传递型别正确的元素，通常必须因此进行显式型别转换（**explicit conversions**）。

关于 `insert()` 返回值的讨论, 请见 p182, 那儿有更多例子, 也适用于 `maps`。注意此处仍然透过 `map` 的 `subscript`(下标)操作符提供较为方便的元素安插和设定操作。这一点将在 6.6.3 节, p205 讨论。

如果要移除“拥有某个 *value*”的元素, 调用 `erase()` 即可办到:

```
std::map<std::string, float> coll;
...
// remove all elements with the passed key
coll.erase(key);
```

`erase()` 返回移除元素的个数。对 `maps` 而言其返回值非 0 即 1。

如果 `multimap` 内含重复元素, 你不能使用 `erase()` 来删除这些重复元素中的第一个。你可以这么做:

```
typedef std::multimap<std::string, float> StringFloatMMap;
StringFloatMMap coll;
...
// remove first element with passed key
StringFloatMMap::iterator pos;
pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

这里应该采用成员函数 `find()`, 而非 STL 算法 `find()`, 因为前者速度更快(参见 p154 的例子)。然而你不能使用成员函数 `find()` 来移除“拥有某个 *value* (而非某个 *key*)”的元素。详细讨论请见 p198。

移除元素时, 当心发生意外状况。当你移除迭代器所指对象时, 有一个很大的危险, 看看这个例子:

```
typedef std::map<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos); // RUNTIME ERROR !!!
    }
}
```

对 `pos` 所指元素实施 `erase()`, 会使 `pos` 不再成为一个有效的 `coll` 迭代器。如果此后你未对 `pos` 重新设值就径行使用 `pos`, 前途未卜! 事实上只要一个 `++pos` 操作就会导致未定义的行为。

如果 `erase()` 总是返回下一元素的位置，那就好办了：

```
typedef std::map<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        pos = coll.erase(pos); // would be fine, but COMPILE TIME ERROR
    }
    else {
        ++pos;
    }
}
```

可惜 STL 设计过程中否决了这种想法，因为万一用户并不需要这一特性，就会耗费不必要的执行时间。我个人不太赞成这项决定，因为这样一来代码会变得更复杂，更容易出错，长期而言，恐怕得不偿失！

下面是移除“迭代器所指元素”的正确作法：

```
typedef std::map<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// remove all elements having a certain value
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        coll.erase(pos++);
    }
    else {
        ++pos;
    }
}
```

注意，`pos++` 会将 `pos` 移向下一元素，但返回其原始值（指向原位置）的一个副本。因此，当 `erase()` 被调用，`pos` 已经不再指向那个即将被移除的元素了。

6.6.3 将 Maps 视为关联式数组 (Associated Arrays)

通常，关联式容器并不提供元素的直接存取，你必须依靠迭代器。不过 `maps` 是个例外。Non-const `maps` 提供下标操作符，支持元素的直接存取，如表 6.32。

不过，下标操作符的索引值并非元素整数位置，而是元素的 *key*。也就是说，索引可以是任意型别，而非局限为整数型别。这种接口正是我们所说的关联式数组 (associative array)。

表 6.32 Maps 的直接元素存取 (透过 operator[])

操作	效果
m[key]	返回一个 reference，指向键值为 key 的元素。如果该元素尚未存在，就安插该元素

和一般数组之间的区别还不仅仅在于索引型别。其它的区别包括：你不可能用上一个错误索引。如果你使用某个 *key* 作为索引，而容器之中尚未存在对应元素，那么就会自动安插该元素。新元素的 *value* 由 default 构造函数构造。如果元素的 *value* 型别没有提供 default 构造函数，你就没这个福分了。再次提醒你，所有基本数据类型别都提供有 default 构造函数，以零为初值 (见 p14)。

关联式数组的行为方式可说是毁誉参半：

- 优点是你可以透过更方便的接口对着 map 安插新元素。例如：

```
std::map<std::string, float> coll; // empty collection
/* insert "otto"/7.7 as key/value pair
 * - first it inserts "otto"/float()
 * - then it assigns 7.7
 */
coll["otto"] = 7.7;
```

其中的语句：

```
coll["otto"] = 7.7;
```

处理如下：

1. 处理 coll["otto"]：

- 如果存在键值为 "otto" 的元素，以上式子返回该元素的 reference。
- 如果没有任何元素的键值是 "otto"，以上式子便为 map 自动安插一个新元素，键值 *key* 为 "otto"，实值 *value* 则以 default 构造函数完成，并返回一个 reference 指向新元素。

2. 将 7.7 赋值给 *value*：

- 紧接着，将 7.7 赋值给上述刚刚诞生的新元素。

这样，map 之内就包含了一个键值 (*key*) 为 "otto" 的元素，其实值 (*value*) 为 7.7。

- 缺点是你可能会不小心误置新元素。例如下面的语句可能会做出一些意想不到的事情：

```
std::cout << coll["ottto"];
```

它会安插一个键值为 "otto" 的新元素，然后打印其实值，缺省情况下是 0。然而，按道理它应该产生一条错误信息，告诉你你把 "otto" 拼写错了。

同时亦请注意，这种元素安插方式比一般的 maps 安插方式来得慢，p202 曾经谈过这个主题。原因是新元素必须先使用 default 构造函数将实值 (value) 初始化，而这个初值马上又被真正的 value 给覆盖了。

6.6.4 异常处理 (Exception Handling)

就异常处理而言，Maps 和 multimaps 的行为与 sets 和 multisets 一样。参见 p185。

6.6.5 Maps 和 Multimaps 运用实例

将 Map 当做关联式数组

下面这个例子将 map 当成一个关联式数组来使用，用来反映股票行情。元素的键值 (key) 是股票名称，实值 (value) 是股票价格：

```
// cont/map1.cpp

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

int main()
{
    /* create map / associative array
     * - keys are strings
     * - values are floats
     */
    typedef map<string,float> StringFloatMap;

    StringFloatMap stocks; // create empty container

    // insert some elements
    stocks["BASF"] = 369.50;
```



```
stocks["VW"] = 413.50;
stocks["Daimler"] = 819.00;
stocks["BMW"] = 834.00;
stocks["Siemens"] = 842.20;

// print all elements
StringFloatMap::iterator pos;
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
}
cout << endl;

// boom (all prices doubled)
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    pos->second *= 2;
}

// print all elements
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
}
cout << endl;

/* rename key from "VW" to "Volkswagen"
 * - only provided by exchanging element
 */
stocks["Volkswagen"] = stocks["VW"];
stocks.erase("VW");

// print all elements
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
}
}
```

程序输出如下:

```
stock: BASF price: 369.5
stock: BMW price: 834
stock: Daimler price: 819
stock: Siemens price: 842.2
stock: VW price: 413.5

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: VW price: 827

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: Volkswagen price: 827
```

将 Multimap 当做字典

下面例子展示如何将 multimap 当成一个字典来使用:

```
// cont/mmap1.cpp

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

int main()
{
    // define multimap type as string/string dictionary
    typedef multimap<string, string> StrStrMMap;

    // create empty dictionary
    StrStrMMap dict;

    // insert some elements in random order
    dict.insert(make_pair("day", "Tag"));
}
```

```

dict.insert(make_pair("strange", "fremd"));
dict.insert(make_pair("car", "Auto"));
dict.insert(make_pair("smart", "elegant"));
dict.insert(make_pair("trait", "Merkmal"));
dict.insert(make_pair("strange", "seltsam"));
dict.insert(make_pair("smart", "raffiniert"));
dict.insert(make_pair("smart", "klug"));
dict.insert(make_pair("clever", "raffiniert"));

// print all elements
StrStrMMap::iterator pos;
cout.setf (ios::left, ios::adjustfield);
cout << ' ' << setw(10) << "english "
    << "german " << endl;
cout << setfill('-') << setw(20) << " "
    << setfill(' ') << endl;
for (pos = dict.begin(); pos != dict.end(); ++pos) {
    cout << ' ' << setw(10) << pos->first.c_str()
        << pos->second << endl;
}
cout << endl;

// print all values for key "smart"
string word("smart");
cout << word << ": " << endl;
for (pos = dict.lower_bound(word);
    pos != dict.upper_bound(word); ++pos) {
    cout << "      " << pos->second << endl;
}

// print all keys for value "raffiniert"
word = ("raffiniert");
cout << word << ": " << endl;
for (pos = dict.begin(); pos != dict.end(); ++pos) {
    if (pos->second == word) {
        cout << " " << pos->first << endl;
    }
}
}

```

程序输出如下:

```
    english    german
-----
    car        Auto
    clever      raffiniert
    day        Tag
    smart      elegant
    smart      raffiniert
    smart      klug
    strange    fremd
    strange    seltsam
    trait      Merkmal

smart:
    elegant
    raffiniert
    klug
raffiniert:
    clever
    smart
```

搜寻具有某特定实值 (*values*) 的元素

下面的例子展示如何使用全局的 `find_if()` 算法来搜寻具有某特定 *value* 的元素:

```
// cont/mapfind.cpp

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

/* function object to check the value of a map element
*/
template <class K, class V>
class value_equals {
private:
    V value;

public:
    // constructor (initialize value to compare with)
```

```
    value_equals (const V& v)
        : value(v) {
    }

    // comparison
    bool operator() (pair<const K, V> elem) {
        return elem.second == value;
    }
};

int main()
{
    typedef map<float, float> FloatFloatMap;
    FloatFloatMap coll;
    FloatFloatMap::iterator pos;

    // fill container
    coll[1]=7;
    coll[2]=4;
    coll[3]=2;
    coll[4]=3;
    coll[5]=6;
    coll[6]=1;
    coll[7]=3;

    // search an element with key 3.0
    pos = coll.find(3.0); // logarithmic complexity
    if (pos != coll.end()) {
        cout << pos->first << ": "
              << pos->second << endl;
    }

    // search an element with value 3.0
    pos = find_if(coll.begin(), coll.end(), // linear complexity
                  value_equals<float, float>(3.0));
    if (pos != coll.end()) {
        cout << pos->first << ": "
              << pos->second << endl;
    }
}
```

程序输出如下：

```
3: 2
4: 3
```

6.6.6 综合实例：

运用 Maps, Strings 并于执行期指定排序准则

这里再示范一个例子。此例针对高级程序员而非 STL 初学者。你可以把它视为展现 STL 威力与障碍的一个范例。更明确地说，这个例子展现了以下技巧：

- 如何使用 maps
- 如何撰写和使用仿函数（functor, 或名 function object）
- 如何在执行期定义排序准则
- 如何在“不在乎大小写”的情况下比较字符串（strings）

```
// cont/mapcmp.cpp

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

/* function object to compare strings
 * - allows you to set the comparison criterion at runtime
 * - allows you to compare case insensitive
 */
class RuntimeStringCmp {
public:
    // constants for the comparison criterion
    enum cmp_mode {normal, nocase};

private:
    // actual comparison mode
    const cmp_mode mode;

    // auxiliary function to compare case insensitive
    static bool nocase_compare (char c1, char c2)
    {
        return toupper(c1) < toupper(c2);
    }
}
```

```

public:
    // constructor: initializes the comparison criterion
    RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
    }

    // the comparison
    bool operator() (const string& s1, const string& s2) const {
        if (mode == normal) {
            return s1 < s2;
        }
        else {
            return lexicographical_compare(s1.begin(), s1.end(),
                                           s2.begin(), s2.end(),
                                           nocase_compare);
        }
    }
};

/* container type:
 * - map with
 * -string keys
 * -string values
 * - the special comparison object type
 */
typedef map<string, string, RuntimeStringCmp> StringStringMap;

// function that fills and prints such containers
void fillAndPrint(StringStringMap& coll);

int main()
{
    // create a container with the default comparison criterion
    StringStringMap coll1;
    fillAndPrint(coll1);
}

```

```
// create an object for case-insensitive comparisons
RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

// create a container with the case-insensitive
// comparisons criterion
StringStringMap coll2(ignorecase);
fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
    // fill insert elements in random order
    coll["Deutschland"] = "Germany";
    coll["deutsch"] = "German";
    coll["Haken"] = "snag";
    coll["arbeiten"] = "work";
    coll["Hund"] = "dog";
    coll["gehen"] = "go";
    coll["Unternehmen"] = "enterprise";
    coll["unternehmen"] = "undertake";
    coll["gehen"] = "walk";
    coll["Bestatter"] = "undertaker";

    // print elements
    StringStringMap::iterator pos;
    cout.setf(ios::left, ios::adjustfield);
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << setw(15) << pos->first.c_str() << " "
             << pos->second << endl;
    }
    cout << endl;
}
```

main() 构造出两个容器，并对它们调用 fillAndPrint()。这个函数以相同的元素值填充上述两个容器，然后打印其内容。两个容器的排序准则不同：

1. coll1 使用一个型别为 RuntimeStringCmp 的缺省仿函数。这个仿函数以元素的 operator< 来执行比较操作。
2. coll2 使用一个型别为 RuntimeStringCmp 的仿函数，并以 nocase 为初值。nocase 会令这个仿函数以“大小写无关”模式来完成字符串的比较和排序。

程序输出如下:

```
Bestatter    undertaker
Deutschland  Germany
Haken        snag
Hund         dog
Unternehmen  enterprise
arbeiten     work
deutsch      German
gehen        walk
unternehmen  undertake
```

```
arbeiten     work
Bestatter    undertaker
deutsch      German
Deutschland  Germany
gehen        walk
Haken        snag
Hund         dog
Unternehmen  undertake
```

第一部分打印第一个容器的内容, 该容器以 `operator<` 进行排序。首先输出所有键值为大写的字符串, 然后是键值为小写的字符串。

第二部分以“大小写无关”模式打印所有字符串, 次序和第一部分不同。请注意, 第二部分少列了一个元素, 因为在大小写无关的情况下“Unternehmen”和“unternehmen”被视为两个相同字符串²⁹, 而我们使用的 `map` 并不接纳重复元素。很不幸, 打印结果乱七八糟。原本“*value* 应为 “enterprise” “的那个 *key* (是个德文字), 其 *value* 却变成 “undertake”。看来这里应该使用 `multimap`。没错, `multimap` 的确是用来表现字典的一个典型容器。

²⁹ 德语中的所有名词, 第一个字母皆大写。动词全部小写。

6.7 其它 STL 容器

STL 是个框架，除了提供标准容器，它也允许你使用其它数据结构作为容器。你可以使用字符串或数组作为 STL 容器，也可以自行撰写特殊容器以满足特殊需求。如果你自行撰写容器，仍可从诸如排序、合并等算法中受益。这样的框架正是“开放性封闭 (*Open-Closed*)”原则的极佳范例³⁰：允许扩展，谢绝修改。

下面是使你的容器“STL 化”的三种不同方法：

1. The invasive approach³¹ (侵入性作法)

直接提供 STL 容器所需接口。特别是诸如 `begin()` 和 `end()` 之类的常用函数。这种作法需以某种特定方式编写容器，所以是侵入性的。

2. The noninvasive approach (非侵入性作法)

由你撰写或提供特殊迭代器，作为算法和特殊容器间的界面。此一作法是非侵入性的，它所需要的只是“遍历容器所有元素”的能力——这是任何容器都能以某种形式展现的能力。

3. The wrapper approach (包装法)

将上述两种方法加以组合，我们可以写一个外套类别 (*wrapper class*) 来包装任何数据结构，并显示出与 STL 容器相似的接口。

本节首先将 `string` 视为标准容器来讨论，当做侵入性作法的一个例子，然后再以非侵入性作法讨论重要的标准容器：`array`。当然你也可以使用包装法来存取 `array` 的数据。本节最后概略讨论了一个目前尚未被涵盖于标准规格中的容器：`hash table`。

任何 STL 容器都应该能够以不同的配置器 (*allocator*) 加以参数化。C++ 标准程序库提供了一些特殊函数和类别，帮助你撰写配置器并对付尚未初始化的内存。详见 15.2 节, p728。

6.7.1 Strings 可被视为一种 STL 容器

C++ 标准程序库的 `string` 类别，乃是“以侵入性作法编写 STL 容器”的一个好例子（关于 `string` 类别的详尽讨论，请见第 11 章）。Strings 可被视为以字符 (*characters*) 为元素的一种容器；字符构成序列，你可以在序列上来回移动遍历。因此，标准的 `string` 类别提供了 STL 容器接口。Strings 也提供成员函数 `begin()` 和 `end()`，返回随机存取迭代器，可用来遍历整个 `string`。同时，为了支持迭代器

³⁰ 我从 Robert C. Martin 那儿头一次听说这个名称，他是从 Bertrand Meyer 那儿听来的。

³¹ 有时也说成 *intrusive* 和 *nonintrusive*

和迭代器适配器(iterator adapters), strings 也提供了一些操作函数, 例如 `push_back()` 用以支持 back inserters。

从 STL 角度来思考, string 的处理有点不寻常, 因为我们通常将 string 当做一个对象来处理(我们可以传递、复制或设定 string)。但如果要对单个字符进行处理, 采用 STL 算法将大有益处。例如可以采用 istream 迭代器读取字符, 或转换 string 内的字符(譬如转成大写或小写)。此外, 透过 STL 算法, 可以对 string 采取特殊的比较规则——标准 string 接口并不提供这种能力。

p497, 11.2.13 节是 string 完整章节的一部分, 在那里我详细讨论了 string 的 STL 相关特性, 并给出一些实例。

6.7.2 Arrays 可被视为一种 STL 容器

我们也可以把数组当成 STL 容器来使用, 但 array 并不是类别, 所以不提供 `begin()` 和 `end()` 等成员函数, 也不允许存在任何成员函数。在这里, 我们只能采用非侵入性作法或包装法。

直接运用数组

采取非侵入性作法很简单, 你只需要一个对象, 它能够透过 STL 迭代器接口, 遍历数组的所有元素。事实上这样的对象早就恭候多时了, 它就是普通指针。STL 设计之初就决定让迭代器拥有和普通指针相同的接口, 于是你可以将普通指针当成迭代器来使用。这又一次展示了纯粹抽象的泛化概念: “行为类似迭代器”的任何东西就是一种迭代器。事实上指针正是一个随机存取迭代器(参见 p255, 7.2.5 节)。以下例子示范如何以 array 作为 STL 容器:

```
// cont/array1.cpp

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

int main()
{
    int coll[] = { 5, 6, 2, 4, 1, 3 };

    // square all elements
    transform (coll, coll+6,          // first source
               coll,                   // second source
               coll,                   // destination
               multiplies<int>{});    // operation
```

```
// sort beginning with the second element
sort (coll+1, coll+6);

// print all elements
copy (coll, coll+6,
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

千万注意，一定要正确传递数组尾部位置，这里是 `coll+6`。记住，一定要确保区间尾端是最后元素的下一个位置。

程序输出如下：

```
25 1 4 9 16 36
```

`p382` 和 `p421` 还有一些例子。

一个数组外包装

Bjarne Stroustrup 的《*The C++ Programming Language*》第三版中，介绍了一个很有用的数组包装类别，性能不输一般的数组，而且更安全。这是“使用者自行定义 STL 容器”的一个好例子。该容器所使用的，就是包装法：在数组之外包装一层常用的容器界面。

`Class carray`（这是“C array”或“constant size array”的缩写）定义如下³²：

```
// cont/carrray.hpp

#include <cstddef>

template<class T, std::size_t thesize>
class carray {
private:
    T v[thesize]; // fixed-size array of elements of type T

public:
    // type definitions
    typedef T    value_type;
    typedef T*   iterator;
```

³² 原始例子名为 `c_array`，定义于 Bjarne Stroustrup 的《*The C++ Programming Language*》第三版 17.5.4 节。这里我做了一些改动。

```

typedef const T*      const_iterator;
typedef T&            reference;
typedef const T&      const_reference;
typedef std::size_t   size_type;
typedef std::ptrdiff_t difference_type;

// iterator support
iterator begin() { return v; }
const_iterator begin() const { return v; }
iterator end() { return v + thesize; }
const_iterator end() const { return v+thesize; }

// direct element access
reference operator[](std::size_t i) { return v[i]; }
const_reference operator[](std::size_t i) const { return v[i]; }

// size is constant
size_type size() const { return thesize; }
size_type max_size() const { return thesize; }

// conversion to ordinary array
T* as_array() { return v; }
};

```

下面是 `carray` 的一个运用实例:

```

// cont/carray1.cpp

#include <algorithm>
#include <functional>
#include "carray.hpp"
#include "print.hpp"
using namespace std;

int main()
{
    carray<int,10> a;

    for (unsigned i=0; i<a.size(); ++i) {
        a[i] = i+1;
    }
}

```

```
    PRINT_ELEMENTS(a);

    reverse(a.begin(), a.end());
    PRINT_ELEMENTS(a);

    transform(a.begin(), a.end(),          // source
               a.begin(),                  // destination
               negate<int>{});             // operation
    PRINT_ELEMENTS(a);
}
```

如你所见，你可以使用一般容器接口（`begin()`，`end()`，`operator[]`）来直接操作这个容器。这么一来你也就可以使用那些需要调用 `begin()` 和 `end()` 的各项操作了，例如某些 STL 算法，以及 p118 所介绍的辅助函数 `PRINT_ELEMENTS()`。

程序输出如下：

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

（译注：关于这个 `class`，更细致的实作手法请参考 <http://www.boost.org/> 的 Boost 程序库）

6.7.3 Hash Tables

有一个数据结构可用于群集（collection）身上，非常重要，却未包含于 C++ 标准程序库内，那就是 `hash table`。最初的 STL 并未涵盖 `hash table`，然而确实曾有提案要求，将 `hash table` 并入标准规格。但是标准委员会会觉得这份提议来得太晚，没有采纳。（我们必须在某个时间点中止引入新功能，开始关注细节，否则工作永无止境）

不过，C++ 社群早已经有了数种可用的 `hash table` 实作版本。一般而言程序库会提供四种 `hash table`：`hash_set`，`hash_multiset`，`hash_map`，`hash_multimap`。和其它关联式容器一样，“multi”版允许元素重复，“map”版的元素是个 *key/value pair*。Bjarne Stroustrup 在其《*The C++ Programming Language*》第三版 17.6 节中曾经实作一个 `hash_map` 容器作为示范，并进行详细讨论。关于 `hash table` 的具体实现，可参见 STLport（<http://www.stlport.org>）。当然，由于 `hash table` 尚未正规化，所以不同的实作版本可能在细节上有所不同。（译注：如果你对 `hash table` 的运用和设计原理感兴趣，请参考《STL 源码剖析》by 侯捷，慕峰 2002，5.7 节~5.11 节。该处讨论的是 SGI STL 实作版本，涵括底层 `hash table` 和外显接口 `hash_set`，`hash_multiset`，`hash_map`，`hash_multimap`）

6.8 动手实现 Reference 语义

通常, STL 容器提供的是“value 语义”而非“reference 语义”, 后者在内部构造了元素副本, 任何操作返回的也是这些副本。p135, 5.10.2 节讨论了这种作法的优劣, 并分析了产生后果。总之, 要在 STL 容器中用到“reference 语义”(不论是因为元素的复制代价太大, 或因为需要在不同群集中共享同一个元素), 就要采用智能型指针, 避免可能的错误。这里有一个解决办法: 对指针所指的对象采用 **reference counting** (参考计数) 智能型指针³³。

```
// cont/countptr.hpp

#ifndef COUNTED_PTR_HPP
#define COUNTED_PTR_HPP

/* class for counted reference semantics
 * - deletes the object to which it refers when the last CountedPtr
 *   that refers to it is destroyed
 */
template <class T>
class CountedPtr {
private:
    T* ptr;          // pointer to the value
    long* count;     // shared number of owners

public:
    // initialize pointer with existing pointer
    // - requires that the pointer p is a return value of new
    explicit CountedPtr (T* p=0)
        : ptr(p), count(new long(1)) {
    }

    // copy pointer (one more owner)
    CountedPtr (const CountedPtr<T>& p) throw()
        : ptr(p.ptr), count(p.count) {
        ++*count;
    }
}
```

³³ 感谢 Greg Colvin 和 Beman Dawes 对这个 class 的实作内容所给予的回应。

```

    // destructor (delete value if this was the last owner)
    ~CountedPtr () throw() {
        dispose();
    }

    // assignment (unshare old and share new value)
    CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
        if (this != &p) {
            dispose();
            ptr = p.ptr;
            count = p.count;
            ++*count;
        }
        return *this;
    }

    // access the value to which the pointer refers
    T& operator*() const throw() {
        return *ptr;
    }
    T* operator->() const throw() {
        return ptr;
    }

private:
    void dispose() {
        if (--*count == 0) {
            delete count;
            delete ptr;
        }
    }
};

#endif /*COUNTED_PTR_HPP*/

```

这个 class 有点类似标准规格书所提供的 `auto_ptr` class (参见 p38, 4.2 节)。用来初始化智能型指针值, 应当是 `operator new` 的返回值。但是和 `auto_ptr` 不同的是, 这种智能型指针一旦被复制, 原指针和新的副本指针都是有效的。只有当指向同一对象的最后一个智能型指针被摧毁, 其所指对象才会被删除。

你可以改善这个 class，例如你可以为它实现自动类型转换，或是提供“将拥有权由智能型指针交给调用端”的能力。

以下程序说明如何使用这个 class：

```
// cont/refseml.cpp

#include <iostream>
#include <list>
#include <deque>
#include <algorithm>
#include "countptr.hpp"
using namespace std;

void printCountedPtr (CountedPtr<int> elem)
{
    cout << *elem << ' ';
}

int main()
{
    // array of integers (to share in different containers)
    static int values[] = { 3, 5, 9, 1, 6, 4 };

    // two different collections
    typedef CountedPtr<int> IntPtr;
    deque<IntPtr> coll1;
    list<IntPtr> coll2;

    /* insert shared objects into the collections
    * - same order in coll1 coll1
    * - reverse order in coll2 coll2
    */
    for (int i=0; i<sizeof(values)/sizeof(values[0]); ++i) {
        IntPtr ptr(new int(values[i]));
        coll1.push_back(ptr);
        coll2.push_front(ptr);
    }
}
```

```
// print contents of both collections
for_each (coll1.begin(), coll1.end(),
          printCountedPtr);
cout << endl;
for_each (coll2.begin(), coll2.end(),
          printCountedPtr);
cout << endl << endl;

/* modify values at different places
 * - square third value in coll1
 * - negate first value in coll1
 * - set first value in coll2 to 0
 */
*coll1[2] *= *coll1[2];
(**coll1.begin()) *= -1;
(**coll2.begin()) = 0;

// print contents of both collections again
for_each (coll1.begin(), coll1.end(),
          printCountedPtr);
cout << endl;
for_each (coll2.begin(), coll2.end(),
          printCountedPtr);
cout << endl;
}
```

程序输出如下：

```
3 5 9 1 6 4
4 6 1 9 5 3

-3 5 81 1 6 0
0 6 1 81 5 -3
```

注意，如果你调用一个辅助函数，而它在某处保存了群集（collection）内的某个元素（一个 `IntPtr`），那么即使群集被销毁，或其元素全被删除，那个智能型指针所指的元素依然有效。

关于其它智能型指针类别，请参考 <http://www.boost.org/> 的 Boost 程序库，该程序库是 C++ 标准程序库的扩充（在那儿 `CountedPtr<>` 名为 `shared_ptr<>`）。

6.9 各种容器的运用时机

C++ 标准程序库提供了各具特长的不同容器。现在的问题是：该如何选择最佳的容器类别？表 6.33 作了一番概述，但其中有些描述可能不一定实际。例如，如果你需要处理的元素数量很少，可以忽略复杂度，因为线性算法通常对元素本身的处理过程比较快，这种情况下，“线性复杂度搭配快速的元素处理”要比“对数复杂度搭配缓慢的元素处理”来得划算。

以下规则作为表 6.33 的补充，可能对你有所帮助：

- 缺省情况下应该使用 `vector`。`vector` 的内部结构最简单，并允许随机存取，所以数据的存取十分方便灵活，数据的处理也够快。
- 如果经常要在序列头部和尾部安插和移除元素，应该采用 `deque`。如果你希望元素被移除时，容器能够自动缩减内存，那么你也应该采用 `deque`。此外，由于 `vectors` 通常采用一个内存区块来存放元素，而 `deque` 采用多个区块，所以后者可内含更多元素。
- 如果需要经常在容器的中段执行元素的安插、移除和移动，可考虑使用 `list`。`List` 提供特殊的成员函数，可以在常数时间内将元素从 A 容器转移到 B 容器。但由于 `list` 不支持随机存取，所以如果只知道 `list` 的头部却要造访 `list` 的中段元素，性能会大打折扣。

和所有“以节点为基础”的容器相似，只要元素还是容器的一部分，`list` 就不会令指向那些元素的迭代器失效。`vectors` 则不然，一旦超过其容量，它的所有 `iterators`、`pointers`、`references` 都会失效；执行安插或移除操作时，也会令一部分 `iterators`、`pointers`、`references` 失效。至于 `deque`，当它的大小改变，所有 `iterators`、`pointers`、`references` 都会失效。

- 如果你要的容器是这种性质：“每次操作若不成功，便无效用”（并以此态度来处理异常），那么你应该选用 `list`（但是不保证其 `assignment` 操作符和 `sort()`；而且如果元素比较过程中会抛出异常，那就不要调用 `merge()`、`remove()`、`remove_if()`、`unique()`，参见 p172），或是采用关联式容器（但是不保证多元素安插动作，而且如果比较准则（`comparision criterion`）的复制/赋值动作都可能抛出异常，那么也不保证 `swap()`）。STL 的异常处理通论请见 5.11.2 节，p139。6.10.10 节，p249 提供了一个表，列举出“异常发生时提供特别保障”的所有容器操作函数。
- 如果你经常需要根据某个准则来搜寻元素，那么应当使用“以该排序准则对元素进行排序”的 `set` 或 `multiset`。记住，理论上，面对 1,000 个元素的排序，对数复杂度比线性复杂度好 10 倍。这也正是二叉树拿手好戏的发挥时机。

表 6.33 STL 容器能力一览表

	Vector	Deque	List	Set	Multiset	Map	Multimap
典型内部结构	dynamic array	array of arrays	doubly linked list	binary tree	binary tree	binary tree	binary tree
元素	value	value	value	value	value	key/value pair	key/value pair
元素可重复	是	是	是	否	是	对 key 而言否	是
可随机存取	是	是	否	否	否	对 key 而言是	否
迭代器类型	随机存取	随机存取	双向	双向 元素被视为常数	双向 元素被视为常数	双向 key 被视为常数	双向 key 被视为常数
元素搜寻速度	慢	慢	非常慢	快	快	对 key 而言快	对 key 而言快
快速安插移除	尾端	头尾两端	任何位置	—	—	—	—
安插移除导致除效 iterators, pointers, references	重新分配 时	总是如此	绝不会	绝不会	绝不会	绝不会	绝不会
释放被移除元素之 内存	绝不会	有时会	总是如此	总是如此	总是如此	总是如此	总是如此
允许保留内存	是	否	—	—	—	—	—
交易安全 若失败不带来任何 影响	尾端 push/pop 时	头尾两端 push/pop 时	任何时候 除了排序 和赋值	任何时候 除了多元 素安插	任何时候 除了多元 素安插	任何时候 除了多元 素安插	任何时候 除了多元 素安插

就搜寻速度而言, hash table 通常比二叉树还要快 5~10 倍。所以如果有 hash table 可用, 就算它尚未标准化, 也应该考虑使用。但是 hash table 的元素并未排序, 所以如果元素必须排序, 它就用不上了。由于 hash table 不是 C++ 标准程序库的一员, 如果你要保证可移植性, 就必须拥有其源码。

- 如想处理 *key/value pair*, 请采用 map 或 multimap (可以的话请采用 hash table)。
- 如果需要关联式数组, 应采用 map。
- 如果需要字典结构, 应采用 multimap。

有一个问题比较棘手: 如何根据两种不同的排序准则对元素进行排序? 例如存放元素时, 你希望采用客户提供的排序准则, 搜寻元素时, 希望使用另一个排序准则。这和数据库的情况相同, 你需要在数种不同的排序准则下进行快速存取。这时候你可能需要两个 sets 或 maps, 各自拥有不同的排序准则, 但共享相同的元素。注意, 数个群集共享相同的元素, 乃是一项特殊技术, 6.8 节, p222 对此有所阐述。

关联式容器拥有自动排序能力, 并不意味着它们在排序方面的执行效率更高。事实上由于关联式容器每安插一个新元素, 都要进行一次排序, 所以速度反而不及序列式容器经常采用的手法: 先安插所有元素, 然后调用 9.2.2 节, p328 介绍的排序算法进行一次完全排序。

下面两个简单的程序分别使用不同的容器, 从标准输入读取字符串, 进行排序, 然后打印所有元素 (去掉重复字符串):

1. 运用 set:

```
// cont/sortset.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
    /* create a string set
     * - initialized by all words from standard input
     */
    set<string> coll((istream_iterator<string>(cin)),
                    (istream_iterator<string>{}));

    // print all elements
    copy(coll.begin(), coll.end(),
          ostream_iterator<string>(cout, "\n"));
}
```

2. 运用 vector:

```
// cont/sortvec.cpp

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

int main()
{
    /* create a string vector
     * - initialized by all words from standard input
     */
    vector<string> coll((istream_iterator<string>(cin)),
                       (istream_iterator<string>{}));

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

    // print all elements ignoring subsequent duplicates
    unique_copy (coll.begin(), coll.end(),
                 ostream_iterator<string>(cout, "\n"));
}
```

我在我的系统上使用大约 150,000 个字符串来测试这两个程序，我发现 vectors 版本快 10% 左右。如果使用 `reserve()`，vectors 版本还可以再快将近 5%。如果允许重复元素（改用 `multiset` 取代 `set`，调用 `copy()` 取代 `unique_copy()`），则情况发生剧烈变化：vectors 版本领先超过 40%！这些比较虽然不具代表性，但至少证实了一点：对各种不同的元素处理方法多加尝试是值得的。

现实中预测哪种容器最好，往往相当困难。STL 的一大优点就是你可以轻而易举地尝试各种版本。主要工作——各种数据结构和算法——已经就位，你只需依照对自己最有利的方式将它们组合运用就行了。

6.10 细说容器内的型别和成员

本节讨论各种 STL 容器，阐述 STL 容器所支持的一切操作函数。型别和成员一律按功能分组。针对每一种型别定义和操作，本节描述其标记式 (signature)、行为、支持者(容器)。本节涉及的容器包括 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings`。后续数节中 `container` 指的是“支持该成员”的某容器型别。

6.10.1 容器内的型别

`container::value_type`

- 元素型别。
- 用于 `sets` 和 `multisets` 时是常数。
- 用于 `maps` 和 `multimaps` 时是 `pair <const key-type, value-type>`。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 之中都有定义。

`container::reference`

- 元素的引用型别 (reference type)。
- 典型定义: `container::value_type&`。
- 在 `vector<bool>` 中其实是个辅助类别 (参见 p158)。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 中都有定义。

`container::const_reference`

- 常数元素的引用型别 (reference type)。
- 典型定义: `const container::value_type&`。
- 在 `vector<bool>` 中是 `bool`。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 中都有定义。

`container::iterator`

- 迭代器型别。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 中都有定义。

`container::const_iterator`

- 常数迭代器的型别。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 中都有定义。

`container::reverse_iterator`

- 反向迭代器型别。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 中都有定义。

`container::const_reverse_iterator`

- 常数反向迭代器的型别。
- 在 `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 中都有定义。

`container::size_type`

- 无正负号整数型别，用以定义容器大小。
- 在 `vectors`, `deque`s, `lists`, `sets`, `multisets`, `maps`, `multimaps`, `strings` 中都有定义。

`container::difference_type`

- 有正负号整数型别，用以定义距离。
- 在 `vectors`, `deque`s, `lists`, `sets`, `multisets`, `maps`, `multimaps`, `strings` 中都有定义。

`container::key_type`

- 用以定义关联式容器的元素内的 `key` 型别。
- 用于 `sets` 和 `multisets` 时，相当于 `value_type`。
- 在 `sets`, `multisets`, `maps`, `multimaps` 中都有定义。

`container::mapped_type`

- 用以定义关联式容器的元素内的 `value` 型别。
- 在 `maps` 和 `multimaps` 中都有定义。

`container::key_compare`

- 关联式容器内的“比较准则”的型别。
- 在 `sets`, `multisets`, `maps`, `multimaps` 中都有定义。

`container::value_compare`

- 用于整个元素之“比较准则”的型别。
- 用于 `sets` 和 `multisets` 时，相当于 `key_compare`。
- 在 `maps` 和 `multimaps` 中，它是“比较准则”的辅助类别，仅比较两元素的 `key`。
- 在 `sets`, `multisets`, `map`, `multimap` 中都有定义。

`container::allocator_type`

- 配置器型别。
- 在 `vectors`, `deque`s, `lists`, `sets`, `multisets`, `maps`, `multimaps`, `strings` 中都有定义。

6.10.2 生成 (Create)、复制 (Copy)、销毁 (Destroy)

STL 容器支持下列构造函数和析构函数，并且大多数构造函数允许将配置器作为一个附加参数传递（参见第 6.10.9 节, p246）。

`container::container ()`

- default 构造函数
- 产生一个新的空容器
- `vectors`, `deque`s, `lists`, `sets`, `multisets`, `maps`, `multimaps`, `strings` 都支持

`explicit container::container (const CompFunc& op)`

- 以 `op` 为排序准则，产生一个空容器（参见 p191 和 p213 实例）。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- `sets`, `multisets`, `maps`, `multimaps` 支持。

`explicit container::container (const container& c)`

- `copy` 构造函数。
- 产生既有容器的一个副本。
- 针对 `c` 中的每一个元素调用 `copy` 构造函数。
- `vectors`, `deque`s, `lists`, `sets`, `multisets`, `maps`, `multimaps`, `strings` 都支持。

`explicit container::container (size_type num)`

- 产生一个容器，可含 `num` 个元素。
- 元素由其 `default` 构造函数创建。
- `vectors`, `deque`s, `lists` 都支持。

`container::container (size_type num, const T& value)`

- 产生一个容器，可含 `num` 个元素。
- 所有元素都是 `value` 的副本。
- `T` 是元素型别。
- 对于 `strings`，`value` 并非 `pass by reference`。
- `vectors`、`deque`s、`lists` 和 `strings` 都支持。

`container::container (InputIterator beg, InputIterator end)`

- 产生容器，并以区间 `[beg;end)` 内的所有元素为初值。
- 此函数为一个 `member template`（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 都支持。

`container::container (InputIterator beg, InputIterator end,
const CompFunc& op)`

- 产生一个排序准则为 `op` 的容器，并以区间 `[beg;end)` 内的所有元素进行初始化。
- 此函数为一个 `member template`（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
container::~container ()
```

- 析构函数。
- 移除所有元素，并释放内存。
- 对每个元素调用其析构函数。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

6.10.3 非变动性操作 (Nonmodifying Operations)

大小相关操作 (Size Operations)

```
size_type container::size () const
```

- 返回现有元素的数目。
- 欲检查容器是否为空，应使用 `empty()`，因为 `empty()` 可能更快。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
bool container::empty () const
```

- 检验容器是否为空，并返回检查结果。
- 相当于 `container::size()==0`，但是可能更快（尤其对 `lists` 而言）。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
size_type container::max_size () const
```

- 返回容器可包含的最大元素个数。
- 这是一个技术层次的数值，可能取决于容器的内存模型。尤其 `vectors` 通常使用一个内存区段 (`segment`)，所以 `vector` 的这个值往往小于其它容器。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

容量操作 (Capacity Operations)

```
size_type container::capacity () const
```

- 返回重分配内存之前所能容纳的最多元素个数。
- `vectors` 和 `strings` 都支持。

```
void container::reserve (size_type num)
```

- 在内部保留若干内存，至少能够容纳 `num` 个元素。
- 如果 `num` 小于实际容量，对 `vectors` 无效，对 `strings` 则是一个非绑定的缩减请求 (`nonbinding shrink request`)。

- `vectors` 的容量如何缩小, 请见 p149 例子。
- 每次重新分配都会耗用相当时间, 并造成所有 `references`、`pointers`、`iterators` 失效。因此 `reserve()` 可以提高速度, 保持 `references`、`pointers`、`iterators` 的有效性。详见 p149。
- `vectors` 和 `strings` 都支持。

元素间的比较 (Comparison Operations)

`bool comparison (const container& c1, const container& c2)`

- 返回两个同型容器的比较结果。
- `comparison` 可以是下面之一:
 - `operator ==`
 - `operator !=`
 - `operator <`
 - `operator >`
 - `operator <=`
 - `operator >=`
- 如果两个容器拥有相同数量的元素, 且元素顺序相同, 而且所有相应元素两两相比之结果为 `true`, 我们便说这两个容器相等。
- 要检验 A 容器是否小于 B 容器, 需使用“字典顺序”来比较。关于“字典顺序”, 请见 p360 `lexicographical_compare()` 的描述。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps`、`strings` 都支持。

关联式容器特有的非变动性操作

这里所介绍的成员函数都是对应于 p338, 9.5 节和 p397, 9.9 节所讨论的 STL 算法的特殊实作版本。这些函数利用了关联式容器的元素已序性, 提供更好的性能。例如在 1,000 个元素中进行搜寻, 所需的比较平均不超过 10 次 (参见 p21, 2.3 节)。

`size_type container::count (const T& value) const`

- 返回与 `value` 相等的元素个数。
- 这是 p338 所讨论的 `count()` 算法的特殊版本。
- `T` 是被排序值的型别
 - 在 `sets` 和 `multisets` 中, `T` 是元素型别。
 - 在 `maps` 和 `multimaps` 中, `T` 是 `key` 的型别。
- 复杂度: 线性。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
iterator container::find (const T& value)
const_iterator container::find (const T& value) const
```

- 返回“实值等于 *value*”的第一个元素位置。
- 如果找不到元素就返回 *end()*。
- 这是 p341 所讨论的 *find()* 算法的特殊版本。
- *T* 是被排序值的型别：
 - 在 *sets* 和 *multisets* 中，*T* 是元素型别。
 - 在 *maps* 和 *multimaps* 中，*T* 是 *key* 的型别。
- 复杂度：对数。
- *sets*、*multisets*、*maps* 和 *multimaps* 都支持。

```
iterator container::lower_bound (const T& value)
const_iterator container::lower_bound (const T& value) const
```

- 返回一个迭代器，指向“根据排序准则，可安插 *value* 副本的第一个位置”。
- 返回之迭代器指向“实值大于等于 *value* 的第一个元素”（有可能是 *end()*）。
- 如果找不到就返回 *end()*。
- 这是 p413 所讨论的 *lower_bound()* 算法的特殊版本。
- *T* 是被排序值的型别：
 - 在 *sets* 和 *multisets* 中，*T* 是元素型别。
 - 在 *map* 和 *multimap* 中，*T* 是 *key* 的型别。
- 复杂度：对数。
- *sets*、*multisets*、*maps* 和 *multimaps* 都支持。

```
iterator container::upper_bound (const T& value)
const_iterator container::upper_bound (const T& value) const
```

- 返回一个迭代器，指向“根据排序准则，可安插 *value* 副本的最后一个位置”。
- 返回之迭代器指向“实值大于 *value* 的第一个元素”（有可能是 *end()*）。
- 如果找不到就返回 *end()*。
- 这是 p413 所讨论的 *upper_bound()* 算法的特殊版本。
- *T* 是被排序值的型别：
 - 在 *sets* 和 *multisets* 中，*T* 是元素型别。
 - 在 *map* 和 *multimap* 中，*T* 是 *key* 的型别。
- 复杂度：对数。
- *sets*、*multisets*、*maps* 和 *multimaps* 都支持。

```
pair<iterator,iterator> container::equal_range (const T& value)
pair<const_iterator,const_iterator>
    container::equal_range (const T& value) const
```

- 返回一个区间（一对迭代器），指向“根据排序准则，可安插 *value* 副本的第一个位置和最后一个位置”。
- 返回一个区间，其内的元素实值皆等于 *value*。
- 相当于：


```
make_pair(lower_bound(value),upper_bound(value))
```
- 这是 p415 所讨论的 `equal_range()` 算法的特殊版本。
- *T* 是被排序值的型别：
 - 在 `sets` 和 `multisets` 中，*T* 是元素型别。
 - 在 `map` 和 `multimap` 中，*T* 是 *key* 的型别。
- 复杂度：对数。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
key_compare container::key_comp ()
```

- 返回一个“比较准则”。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
value_compare container::value_comp ()
```

- 返回一个作为比较准则的对象。
- 在 `sets` 和 `multisets` 中，它相当于 `key_comp()`。
- 在 `maps` 和 `multimaps` 中，它是一个辅助类别，用来比较两元素的 *key*。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

6.10.4 赋值 (Assignments)

```
container& container::operator = (const container& c)
```

- 将 *c* 的所有元素赋值给现有容器，亦即以 *c* 的元素替换所有现有元素。
- 这个操作符合会针对被覆盖的元素调用其 `assignment` 操作符，针对被附加的元素调用其 `copy` 构造函数，针对被移除的元素调用其析构函数。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
void container::assign (size_type num, const T& value)
```

- 将 *num* 个 *value* 赋值给现有容器，亦即以 *num* 个 *value* 副本替换掉所有现有元素。
- *T* 必须是元素型别。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
void container::assign (InputIterator beg, InputIterator end)
```

- 将区间 `[beg;end)` 内的所有元素赋值给现有容器，亦即以 `[beg;end)` 内的元素副本替换掉所有现有元素。
- 此函数为一个 **member template**（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- `vectors`、`deque`s、`lists` 和 `strings` 都支持。

```
void container::swap (container& c)
```

- 和 `c` 交换内容。
- 两个容器互换：
 - 元素
 - 排序准则（如果有的话）。
- 此函数拥有常数复杂度。如果不再需要容器中的老旧元素，则应使用本函数来取代赋值动作（参见 p147, 6.1.2 节）。
- 对于关联式容器，只要“比较准则”进行复制或赋值时不抛出异常，本函数就不抛出异常。对于其它所有容器，此函数一律不抛出异常。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
void swap (container& c1, container& c2)
```

- 相当于 `c1.swap(c2)`（参见稍早前的描述）。
- 对于关联式容器，只要“比较准则”进行复制或赋值时，不抛出异常，本函数就不抛出异常。对于其它所有容器，此函数一律不抛出异常。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

6.10.5 直接元素存取

```
reference container::at (size_type idx)
```

```
const_reference container::at (size_type idx) const
```

- 二者都返回索引 `idx` 所代表的元素（第一个元素的索引为 0）。
- 如果传入一个无效索引（`< 0` 或 `>= size()`），会导致 `out_of_range` 异常。
- 后续的修改或内存重新分配，可能会导致返回的 `reference` 无效。
- 如果调用者保证索引有效，那么最好使用速度更快的 `operator[]`。
- `vectors`、`deque`s 和 `strings` 都支持。

```
reference container::operator[] (size_type idx)
const_reference container::operator[] (size_type idx) const
```

- 二者都返回索引 `idx` 所代表的元素 (第一个元素的索引为 0)。
- 如果传入一个无效索引 (`< 0` 或 `>= size()`)，会导致未定义的行为。所以调用者必须确保索引有效，否则应该使用 `at()`。
- (1) 修改 `strings` 或 (2) 内存重新分配，可能会导致 `non-const strings` 返回的 `reference` 失效 (详见 p487)。
- `vectors`、`deque`s 和 `strings` 都支持。

```
T& map::operator[] (const key_type& key)
```

- 关联式数组的 `operator[]`。
- 在 `map` 中，会返回 `key` 所对应的 `value`。
- 注意：如果不存在“键值为 `key`”的元素，则本操作会自动生成一个新元素，其初值由 `value` 型别的 `default` 构造函数给定。所以不存在所谓的无效索引。例如：

```
map<int,string> coll;
coll[77] = "hello"; // insert key 77 with value "hello"
cout << coll[42];  // Oops, inserts key 42 with value "" and
                  // prints the value
```

详见 p205, 6.6.3 节。

- `T` 是元素的 `value` 型别。
- 相当于：
`(*((insert(make_pair(x,T()))).first)).second`
- 只有 `map` 支持此一操作。

```
reference container::front ()
const_reference container::front () const
```

- 都返回第一个元素 (第一个元素的索引为 0)。
- 调用者必须确保容器内有元素 (`size()>0`)，否则会导致未定义的行为。
- `vectors`、`deque`s 和 `lists` 都支持。

```
reference container::back ()
const_reference container::back () const
```

- 都返回最后一个元素 (索引为 `size()-1`)。
- 调用者必须确保容器内拥有元素 (`size()>0`)；否则会导致未定义的行为。
- `vectors`、`deque`s 和 `lists` 都支持。

6.10.6 “可返回迭代器”的各项操作

本节各个成员函数都会返回迭代器，凭借这些迭代器你可以遍历容器中的所有元素。表 6.34 列出各种容器所提供的迭代器类型（参见 p251, 7.2 节）。

表 6.34 各种容器提供的迭代器类型

容器	迭代器类型 (iterator category)
Vector	随机存取
Deque	随机存取
List	双向
Set	双向，元素为常量
Multiset	双向，元素为常量
Map	双向，key 为常量
Multimap	双向，key 为常量
String	随机存取

```
iterator container::begin ()
const_iterator container::begin () const
```

- 返回一个迭代器，指向容器起始处（第一元素的位置）。
- 如果容器为空，则此动作相当于 `container::end()`。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
iterator container::end ()
const_iterator container::end () const
```

- 返回一个迭代器，指向容器尾端（最后元素的下一位置）。
- 如果容器为空，则此动作相当于 `container::begin()`。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
reverse_iterator container::rbegin ()
const_reverse_iterator container::rbegin () const
```

- 返回一个逆向迭代器，指向逆向迭代时遍历的第一个元素。
- 如果容器为空，则此动作相当于 `container::rend()`。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。


```
reverse_iterator container::rend ()
const_reverse_iterator container::rend () const
```

- 返回一个逆向迭代器，指向逆向迭代时遍历的最后一个元素的下一位置。
- 如果容器为空，则此操作相当于 `container::rbegin()`。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

6.10.7 元素的安插 (Inserting) 和移除 (Removing)

```
iterator container::insert (const T& value)
pair<iterator,bool> container::insert (const T& value)
```

- 安插一个 `value` 副本于关联式容器。
- 元素可重复者 (`multisets` 和 `multimap`) 采用第一形式。返回新元素的位置。
- 元素不可重复者 (`sets` 和 `map`) 采用第二形式。如果有“具备相同 `key`”的元素已经存在，导致无法安插，会返回现有元素的位置和一个 `false`。如果安插成功，返回新元素的位置和一个 `true`。
- `T` 是容器元素的型别，对 `map` 和 `multimap` 而言那是一个 `key/value pair`。
- 函数如果不成功，不带来任何影响。
- `sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
iterator container::insert (iterator pos, const T& value)
```

- 在迭代器 `pos` 的位置上安插一个 `value` 副本。
- 返回新元素的位置。
- 对于关联式容器 (`sets`、`multisets`、`maps` 和 `multimaps`)，`pos` 只作为一个提示，指向安插时必要的搜寻操作的起始建议位置。如果 `value` 刚好可安插于 `pos` 之后，则此函数具有“分期摊还之常数时间”复杂度，否则具有对数复杂度。
- 如果容器是 `sets` 或 `maps`，并且已内含一个“实值等于 `value` (意即两者的 `key` 相等)”的元素，则此调用无效，并返回现有元素的位置。
- 对于 `vectors` 和 `deque`s，这个操作可能导致指向其它元素的某些 `iterators` 和 `references` 无效。
- `T` 是容器元素的型别，在 `maps` 和 `multimaps` 中是一个 `key/value pair`。
- 对于 `strings`，`value` 并不采用 `pass by reference`。
- 对于 `vectors` 和 `deque`s，如果元素的复制操作 (`copy` 构造函数和 `operator=`) 不抛出异常，则此函数一旦失败并不会带来任何影响。对于所有其它容器，函数一旦失败并不会带来任何影响。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
void container::insert (iterator pos, size_type num, const T& value)
```

- 在迭代器 *pos* 的位置上安插 *num* 个 *value* 副本。
- 对于 *vectors* 和 *deques*，此操作可能导致指向其它元素的 *iterators* 和 *references* 失效。
- *T* 是容器元素的型别，在 *maps* 和 *multimaps* 中是一个 *key/value* pair。
- 对于 *strings*，*value* 并不采用 *pass by reference*。
- 对于 *vectors* 和 *deques*，如果元素复制动作（*copy* 构造函数和 *operator=*）不抛出异常，则函数失败亦不会带来任何影响。对于 *lists*，函数若失败不会带来任何影响。
- *vectors*、*deques*、*lists* 和 *strings* 都支持。

```
void container::insert (InputIterator beg, InputIterator end)
```

- 将区间 [*beg*, *end*) 内所有元素的副本安插于关联式容器内。
- 此函数是个 *member template*（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- *sets*、*multisets*、*maps*、*multimaps* 和 *strings* 都支持。

```
void container::insert (iterator pos, InputIterator beg,
                        InputIterator end)
```

- 将区间 [*beg*, *end*) 内所有元素的副本安插于迭代器 *pos* 所指的位置上。
- 此函数是个 *member template*（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- 对于 *vectors* 和 *deques*，此操作可能导致指向其它元素的 *iterators* 和 *references* 失效。
- 对于 *lists*，此函数若失败不会带来任何影响。
- *vectors*、*deques*、*lists* 和 *strings* 都支持。

```
void container::push_front (const T& value)
```

- 安插 *value* 的副本，使成为第一个元素。
- *T* 是容器元素的型别。
- 相当于 *insert(begin(), value)*。
- 对于 *deques*，此一操作会造成“指向其它元素”的 *iterators* 失效，而“指向其它元素”的 *references* 仍保持有效。
- 此函数若失败不会带来任何影响。
- *deques* 和 *lists* 都支持。

```
void container::push_back (const T& value)
```

- 安插 *value* 的副本，使成为最后一个元素。
- *T* 是容器元素的型别。
- 相当于 *insert(end(), value)*。

- 对于 `vectors`, 如果造成内存重新分配, 此操作会造成“指向其它元素”的 `iterators` 和 `references` 失效。
- 对于 `deques`, 此一操作造成“指向其它元素”的 `iterators` 失效, 而“指向 (或说代表) 其它元素”的 `reference` 始终有效。
- 此函数若失败不会带来任何影响。
- `vectors`、`deques`、`lists` 和 `strings` 都支持。

```
void list::remove (const T& value)
void list::remove_if (UnaryPredicate op)
```

- `remove()` 会移除所有“实值等于 `value`”的元素。
- `remove_if()` 会移除所有“使判断式 `op(elem)` 结果为 `true`”的元素。
- 注意在函数调用过程中, `op` 不应改变状态。详见 p302, 8.14 节。
- 两者都会调用被移除元素的析构函数。
- 剩余元素的相对次序保持不变 (`stable`)。
- 这是 p378 所讨论的 `remove()` 算法的特殊版本。
- `T` 是容器元素的型别。
- 细节和范例见 p170。
- 只要元素的比较动作不抛出异常, 此函数也不抛出异常。
- 只有 `lists` 支持这个成员函数。

```
size_type container::erase (const T& value)
```

- 从关联式容器中移除所有和 `value` 相等的元素。
- 返回被移除的元素个数。
- 调用被移除元素的析构函数。
- `T` 是已序 (`sorted`) 元素的型别。
 - 在 `sets` 和 `multisets` 中, `T` 是元素型别。
 - 在 `map` 和 `multimap` 中, `T` 是 `key` 的型别。
- 此函数不抛出异常。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

```
void container::erase (iterator pos)
iterator container::erase (iterator pos)
```

- 将迭代器 `pos` 所指位置上的元素移除。
- 序列式容器 (`vectors`、`deques`、`lists` 和 `strings`) 采用第二形式, 返回后继元素的位置 (或返回 `end()`)。
- 关联式容器 (`sets`、`multisets`、`maps` 和 `multimaps`) 采用第一形式, 无返回值。
- 两者都调用被移除元素的析构函数。

- 注意，调用者必须确保迭代器 `pos` 有效。例如：
`coll.erase(coll.end());` // ERROR → undefined behavior
- 对于 `vectors` 和 `deque`s，此操作可能造成“指向其它元素”的 `iterators` 和 `references` 无效。
- 对于 `vectors` 和 `deque`s，只要元素复制操作（`copy` 构造函数和 `operator=`）不抛出异常，此函数就不抛出异常。对于其它容器，此函数不抛出异常。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
void container::erase (iterator beg, iterator end)
iterator container::erase (iterator beg, iterator end)
```

- 移除区间 `[beg, end)` 内的所有元素。
- 序列式容器（`vectors`、`deque`s、`lists` 和 `strings`）采用第二形式，返回被移除的最后一个元素的下一位置（或返回 `end()`）。
- 关联式容器（`sets`、`multisets`、`maps` 和 `multimaps`）采用第一形式，无返回值。
- 一如区间惯例，始于 `beg`（含）终于 `end`（不含）的所有元素都被移除。
- 调用被移除元素的析构函数。
- 调用者必须确保 `beg` 和 `end` 形成一个有效序列，并且该序列是容器的一部分。
- 对于 `vectors` 和 `deque`s，此操作可能导致“指向其它元素”的 `iterators` 和 `references` 失效。
- 对于 `vectors` 和 `deque`s，只要元素复制动作（`copy` 构造函数和 `operator=`）不抛出异常，此函数就不抛出异常。对于其它容器，此函数不抛出异常。
- `vectors`、`deque`s、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

```
void container::pop_front ()
```

- 将容器的第一个元素移除。
- 相当于 `container.erase(container.begin())`。
- 注意：如果容器是空的，会导致未定义行为。因此，调用者必须确保容器至少有一个元素，也就是 `size()>0`。
- 此函数不抛出异常。
- `deque`s 和 `lists` 都支持。

```
void container::pop_back ()
```

- 将容器的最后一个元素移除。
- 相当于 `container.erase(--container.end())`，前提是其中的表达式有效。在 `vector` 中此表达式不一定有效（参见 p258）。
- 注意，如果容器为空，会导致未定义行为。因此，调用者必须确保容器至少包

含一个元素, 也就是 `size()>0`。

- 此函数不抛出异常。
- `vectors`、`deques` 和 `lists` 都支持。

```
void container::resize (size_type num)
void container::resize (size_type num, T value)
```

- 两者都将容器大小改为 `num`。
- 如果 `size()` 原本就是 `num`, 则两者皆不生效用。
- 如果 `num` 大于 `size()`, 则在容器尾端产生并附加额外元素。第一形式透过 `default` 构造函数来构造新元素, 第二形式则以 `value` 的副本作为新元素。
- 如果 `num` 小于 `size()`, 则移除尾端元素, 直到大小为 `size()`。每个被移除元素的析构函数都会被调用。
- 对于 `vectors` 和 `deques`, 这些函数可能导致“指向其它元素”的 `iterators` 和 `references` 失效。
- 对于 `vectors` 和 `deques`, 只要元素复制操作 (`copy` 构造函数和 `operator=`) 不抛出异常, 这些函数就不抛出异常。对于 `lists`, 函如果失败不会带来任何影响。
- `vectors`、`deques`、`lists` 和 `strings` 都支持。

```
void container::clear ()
```

- 移除所有元素 (将容器清空)。
- 调用被移除元素的析构函数。
- 这一容器的所有 `iterators` 和 `references` 都将失效。
- 对于 `vectors` 和 `deques`, 只要元素复制操作 (`copy` 构造函数和 `operator=`) 不抛出异常, 此函数就不抛出异常。对于其它容器, 此函数不抛出异常。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

6.10.8 Lists 的特殊成员函数

```
void list::unique ()
void list::unique (BinaryPredicate op)
```

- 移除 `lists` 之内相邻而重复的元素, 使每一个元素都不同于下一个元素。
- 第一形式会将所有“和前一元素相等”的元素移除。
- 第二形式的意义是: 任何一个元素 `elem`, 如果其前一元素是 `e`, 而 `elem` 和 `e` 造成二元判断式 `op(elem, e)` 获得 `true` 值, 那么就移除 `elem`³⁴。换言之, 这个判断式并非拿元素和其目前的前一紧临元素比较, 而是拿元素和其未被移除的前一元素比较。

³⁴ 第二版的 `unique()` 仅在支持 `member templates` 的系统中可用 (参见 p11)。

- 注意, `op` 不应在函数调用过程中改变状态, 详见 p302, 8.1.4 节。
- 被移除元素的析构函数会被调用。
- 这是 p381 `unique()` 算法的 `lists` 特别版本。
- 如果“元素比较动作”中不抛出异常, 则此函数亦不抛出异常。

```
void list::splice (iterator pos, list& source)
```

- 将 `source` 的所有元素移动到 `*this`, 并安插到迭代器 `pos` 所指位置。
- 调用之后, `source` 清空。
- 如果 `source` 和 `*this` 相同, 会导致未定义的行为。所以调用端必须确定 `source` 和 `*this` 是不同的 `lists`。如果要移动同一个 `lists` 内的元素, 应该使用稍后提及的其它 `splice()` 形式。
- 调用者必须确定 `pos` 是 `*this` 的一个有效位置; 否则会导致未定义的行为。
- 本函数不抛出异常。

```
void list::splice (iterator pos, list& source, iterator sourcePos)
```

- 从 `source list` 中, 将位于 `sourcePos` 位置上的元素移动至 `*this`, 并安插于迭代器 `pos` 所指位置。
- `source` 和 `*this` 可以相同。这种情况下, 元素将在 `lists` 内部被移动。
- 如果 `source` 和 `*this` 不是同一个 `list`, 在此操作之后, 其元素个数少 1。
- 调用者必须确保 `pos` 是 `*this` 的一个有效位置、`sourcePos` 是 `source` 的一个有效迭代器, 而且 `sourcePos` 不是 `source.end()`; 否则会导致未定义行为。
- 此函数不抛出异常。

```
void list::splice (iterator pos, list& source,  
                  iterator sourceBeg, iterator sourceEnd)
```

- 从 `source list` 中, 将位于 `[sourceBeg, sourceEnd)` 区间内的所有元素移动到 `*this`, 并安插于迭代器 `pos` 所指位置。
- `source` 和 `*this` 可以相同。这种情况下, `pos` 不得为被移动序列的一部分, 而元素将在 `lists` 内部移动。
- 如果 `source` 和 `*this` 不是同一个 `list`, 在此操作之后, 其元素个数将减少。
- 调用者必须确保 `pos` 是 `*this` 的一个有效位置、`sourceBeg` 和 `sourceEnd` 形成一个有效区间, 该区间是 `source` 的一部分; 否则会导致未定义的行为。
- 本函数不抛出异常。

```
void list::sort ()
```

```
void list::sort (CompFunc op)
```

- 对 `lists` 内的所有元素进行排序。
- 第一型式以 `operator<` 对 `lists` 中的所有元素进行排序。
- 第二型式透过如下的 `op` 操作来比较两元素，进而对 `lists` 中的所有元素排序³⁵：
`op(elem1, elem2)。`
- 实值相同的元素，其顺序保持不变（除非有异常被丢出）。
- 这是 p397 所讨论的 `sort()` 和 `stable_sort()` 算法的“list 特殊版本”。

```
void list::merge (list& source)
void list::merge (list& source, CompFunc op)
```

- 将 `lists source` 内的所有元素并入 `*this`。
- 调用后 `source` 变成空容器。
- 如果 `*this` 和 `source` 在排序准则 `operator<` 或 `op` 之下已序 (*sorted*)，则新产生的 `lists` 也是已序。严格地说，标准规格书要求两个 `lists` 必须已序，但实际上对无序的 `lists` 进行合并也是可能的，不过使用前最好先确认一下。
- 第一形式采用 `operator<` 作为排序准则。
- 第二形式采用以下的 `op` 操作作为可有可无的排序准则，以此比较两个元素的大小³⁶：`op(elem, sourceElem)`
- 这是 p416 所讨论的 `merge()` 算法的 list 特殊版本。
- 只要元素的比较操作不抛出异常，此函数万一失败也不会造成任何影响。

```
void list::reverse ()
```

- 将 `lists` 中的元素颠倒次序。
- 这是 p386 所讨论的 `reverse()` 算法的“list 特殊版本”。
- 本函数不抛出异常。

6.10.9 对配置器 (Allocator) 的支持

所有 STL 容器都能够与某个配置器对象 (allocator object) 所定义的某种特定内存模型 (memory model) 搭配合作 (详见第 15 章)。本节讨论的是支持配置器的各个成员。标准容器要求：配置器 (型别) 的每一个实体都必须是可互换的 (interchangeable)，所以某一容器的空间，可透过另一同型容器释放之。因此，元素 (及其储存空间) 在同型的两个容器之间移动，并不会出现问题。

³⁵ `sort()` 的第二形式仅在支持 member templates 的系统中可用 (参见 p11)

³⁶ `merge()` 的第二形式仅在支持 member templates 的系统中可用 (参见 p11)

基本的配置器相关成员 (Fundamental Allocator Members)

`container::allocator_type`

- 配置器型别。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

`allocator_type container::get_allocator () const`

- 返回容器的内存模型 (memory model)。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

带有“可选择之配置器参数”的构造函数

`explicit container::container (const Allocator& alloc)`

- 产生一个新的空白容器，使用 `alloc` 作为内存模型 (memory model)。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

`container::container (const CompFunc& op, const Allocator& alloc)`

- 产生一个新的空白容器，使用 `alloc` 作为内存模型，并以 `op` 为排序准则。
- `op` 排序准则必须定义 **strict weak ordering** (参见 p176)。
- `sets`、`multisets`、`maps` 和 `multimaps` 都支持。

`container::container (size_type num, const T& value,
const Allocator& alloc)`

- 产生一个拥有 `num` 个元素的容器，使用 `alloc` 作为内存模型。
- 所生成的元素都是 `value` 的副本。
- `T` 是容器元素的型别。注意，对于 `strings`，`value` 采用 `by value` 的型式传递。
- `vectors`、`deques`、`lists` 和 `strings` 都支持。

`container::container (InputIterator beg, InputIterator end,
const Allocator& alloc)`

- 产生一个容器，以区间 `[beg, end)` 内的所有元素为初值，并使用 `alloc` 作为内存模型。
- 此函数是一个 `member template` (参见 p11)。所以只要源序列的元素能够转换为容器元素的型别，此函数就可执行。
- `vectors`、`deques`、`lists`、`sets`、`multisets`、`maps`、`multimaps` 和 `strings` 都支持。

`container::container (InputIterator beg, InputIterator end,
const CompFunc& op, const Allocator& alloc)`

- 产生一个以 *op* 为排序准则的容器，以区间 *[beg, end)* 中的所有元素为初值，并使用 *alloc* 作为内存模型。
- 本函数是一个 *member template* (参见 p11)。所以只要源序列的元素能够转换为容器元素的型别，本函数就可执行。
- 排序准则 *op* 必须定义 *strict weak ordering* (参见 p176)。
- *sets*、*multisets*、*map* 和 *multimap* 都支持。

6.10.10 综观 STL 容器的异常处理

p139, 5.11.2 节曾指出，不同的容器在异常发生时，给予不同程度的保证。通常 C++ 标准程序库在异常发生时并不会泄漏资源或破坏容器的恒常特性 (*invariants*)。有些操作提供更强的保证 (前提是其参数必须满足某些条件)：它们可以保证 *commit-or-rollback* (意思是“要么成功，要么不带来任何影响”)，甚至可以保证绝不抛出异常。表 6.35 列出所有支持更严格保证的操作函数³⁷。

对于 *vectors*、*deque*s 和 *lists* 而言，*resize()* 也提供特别保证。其行为或许相当于 *erase()*，或许相当于 *insert()*，或许相当于什么也没做。

```
void container::resize (size_type num, T value = T())
{
    if (num > size()) {
        insert (end(), num-size(), value);
    }
    else if (num < size()) {
        erase (begin()+num, end());
    }
}
```

因此，它所提供的保证就是 “*erase()* 和 *insert()* 两者所提供的保证” 的组合 (参见 p244)。

³⁷ 感谢 Greg Colvin 和 Dave Abrahams 提供这个表格。

表 6.35 “异常发生时给予特殊保证”的各个容器操作函数

操作	页次	保证
<code>vector::push_back()</code>	241	要么成功, 要么无任何影响
<code>vector::insert()</code>	240	要么成功, 要么无任何影响——前提是元素的复制/赋值操作不抛出异常
<code>vector::pop_back()</code>	243	不抛出异常
<code>vector::erase()</code>	242	不抛出异常——前提是元素的复制/赋值操作不抛出异常
<code>vector::clear()</code>	244	不抛出异常——前提是元素的复制/赋值操作不抛出异常
<code>vector::swap()</code>	237	不抛出异常
<code>deque::push_back()</code>	241	要么成功, 要么无任何影响
<code>deque::push_front()</code>	241	要么成功, 要么无任何影响
<code>deque::insert()</code>	240	要么成功, 要不无任何影响——前提是元素的复制/赋值操作不抛出异常
<code>deque::pop_back()</code>	243	不抛出异常
<code>deque::pop_front()</code>	243	不抛出异常
<code>deque::erase()</code>	242	不抛出异常——前提是元素的复制/赋值操作不抛出异常
<code>deque::clear()</code>	244	不抛出异常——前提是元素的复制/赋值操作不抛出异常
<code>deque::swap()</code>	237	不抛出异常
<code>list::push_back()</code>	241	要么成功, 要么无任何影响
<code>list::push_front()</code>	241	要么成功, 要么无任何影响
<code>list::insert()</code>	240	要么成功, 要么无任何影响
<code>list::pop_back()</code>	243	不抛出异常
<code>list::pop_front()</code>	243	不抛出异常
<code>list::erase()</code>	242	不抛出异常
<code>list::clear()</code>	244	不抛出异常
<code>list::remove()</code>	242	不抛出异常——前提是元素的比较操作不抛出异常
<code>list::remove_if()</code>	242	不抛出异常——前提是判断式 <i>predicate</i> 不抛出异常
<code>list::unique()</code>	244	不抛出异常——前提是元素的比较操作不抛出异常
<code>list::splice()</code>	245	不抛出异常
<code>list::merge()</code>	246	要么成功, 要么无任何影响——前提是元素的比较操作不抛出异常
<code>list::reverse()</code>	246	不抛出异常
<code>list::swap()</code>	237	不抛出异常
<code>[multi]set::insert()</code>	240	要么成功, 要么无任何影响——对单个元素而言
<code>[multi]set::erase()</code>	242	不抛出异常
<code>[multi]set::clear()</code>	244	不抛出异常
<code>[multi]set::swap()</code>	237	不抛出异常——前提是对“比较准则”执行复制/赋值操作时不抛出异常
<code>[multi]map::insert()</code>	240	要么成功, 要么无任何影响——对单个元素而言
<code>[multi]map::erase()</code>	242	不抛出异常
<code>[multi]map::clear()</code>	244	不抛出异常
<code>[multi]map::swap()</code>	237	不抛出异常——前提是对“比较准则”执行复制/赋值操作时不抛出异常

7

STL 迭代器

STL Iterators

7.1 迭代器头文件

所有容器都定义其各自的迭代器型别 (iterator types)，所以当你打算使用某种容器的迭代器时，并不需要含入专门的头文件。不过有几种特别的迭代器，例如逆向 (reverse) 迭代器，被定义于头文件 `<iterator>`¹ 中。通常你不需要在自己的程序中含入上述文件，因为容器必须定义它自己的逆向迭代器型别，所以容器本身已经含入了该头文件。

7.2 迭代器类型 (Iterator Categories)

迭代器是一种“能够遍历某个序列 (sequence) 内的所有元素”的对象。它可以透过与一般指针 (见 5.3 节, p83) 一致的接口来完成自己的工作。迭代器奉行一个纯抽象概念：任何东西，只要行为类似迭代器，就是一种迭代器。不同的迭代器具有不同的“能力” (译注：指行进和存取能力)。由于某些算法需要特定的迭代器能力，所以能力是很重要的概念。例如，排序算法需要“具备随机存取能力”的迭代器，否则执行效率会很惨。由于这个原因，迭代器被分为不同的类型 (图 7.1)，其能力列于表 7.1 中，并将于稍后进行详细讨论。

¹ 早期 STL 将迭代器头文件命名为 `<iterator.h>`。

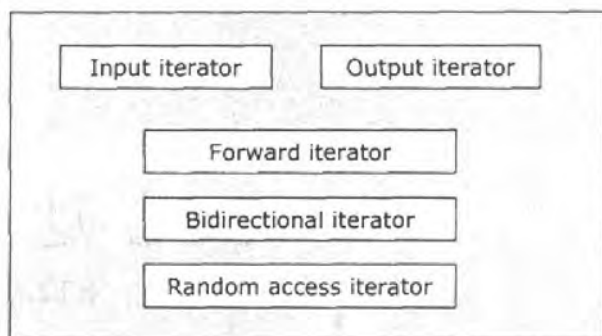


图 7.1 迭代器类型 (categories)

表 7.1 迭代器分类及其能力

迭代器类型 (category)	能力	供应者
Input 迭代器	向前读取 (<i>read</i>)	<code>istream</code>
Output 迭代器	向前写入 (<i>write</i>)	<code>ostream</code> , <code>inserter</code>
Forward 迭代器	向前读取和写入	
Bidirectional 迭代器	向前 (<i>forward</i>) 和向后 (<i>backward</i>) 读取和写入	<code>list</code> , <code>set</code> , <code>multiset</code> , <code>map</code> , <code>multimap</code>
Random access 迭代器	随机存取, 可读取也可写入	<code>vector</code> , <code>deque</code> , <code>string</code> , <code>array</code>

7.2.1 Input (输入) 迭代器

Input 迭代器只能一次一个向前读取元素, 按此顺序一个个传回元素值。表 7.2 列出了 Input 迭代器的各种操作行为。

注意, Input 迭代器只能读取元素一次。如果你复制 Input 迭代器, 并使原 Input 迭代器和新产生的副本都向前读取, 可能会遍历到不同的值。

几乎所有迭代器都具备 Input 迭代器的能力, 而且通常更强。纯粹 Input 迭代器的一个典型例子就是“从标准输入装置 (通常为键盘) 读取数据”的迭代器。同一个值不会被读取两次。一旦从输入流 (`input stream`) 读入一个字 (离开 `input` 缓冲区) 后, 下次读取时就会传回另一个字。

如果两个 Input 迭代器占用同一个位置, 则两者相等。但是, 正如上面所说, 这并不意味着它们存取元素时能够传回相同的值。

表 7.2 Input 迭代器的各项操作

表达式	效果
*iter	读取实际元素
iter->member	读取实际元素的成员（如果有的话）
++iter	向前步进（传回新位置）
iter++	向前步进（传回旧位置）
iter1 == iter2	判断两个迭代器是否相等
iter1 != iter2	判断两个迭代器是否不相等
TYPE(iter)	复制迭代器（copy 构造函数）

你应该尽可能优先选用前置式递增运算符（++iter）而不是后置式递增运算符（iter++），因为前者性能更好。前置式递增运算符不需传回旧值，所以也就不必花费一个临时对象来保存旧值。因此，面对任何迭代器（以及任何抽象数据类型），应该优先使用前置式：

```
++pos    // OK and fast
```

而不是后置式：

```
pos++    // OK, but not so fast
```

这条原则对递减运算符同样适用（但请注意，Input 迭代器并不提供递减运算符）。

7.2.2 Output（输出）迭代器

Output 迭代器和 Input 迭代器相反，其作用是将元素值一个个写入。也就是说，你只能一个元素一个元素地赋新值，而且不能使用 Output 迭代器对同一序列进行两次遍历。这就好像将元素值写到“黑洞”里去（请别介意这个字眼的具体意义），如果在相同位置上对着同一个黑洞进行第二次写入，不能确保这次写入的值会覆盖前一个值。表 7.3 列出 Output 迭代器的有效操作。operator* 只有在赋值语句的左边才有效。

表 7.3 Output 迭代器的操作

表达式	效果
*iter = value	将数值写到迭代器所指位置
++iter	向前步进（传回新位置）
iter++	向前步进（传回旧位置）
TYPE(iter)	复制迭代器（copy 构造函数）

注意，Output 迭代器无需比较（comparison）操作。你无法检验 Output 迭代

器是否有效，或“写入动作”是否成功。你唯一可做的就是写入、写入、再写入。

一般迭代器可以读取和写入元素值，所以几乎所有迭代器都具有 Output 迭代器的功能，就像它们都具备 Input 迭代器的功能一样。纯粹 Output 迭代器的一个典型例子就是“将元素写至标准输出装置（如屏幕或打印机）”的迭代器。如果采用两个 Output 迭代器写至屏幕，第二个字将跟在第一个字后面，而不是覆盖（*overwrite*）第一个字。Output 迭代器的另一个典型例子是 *Inserters*。所谓 *Inserters* 是用来将元素值插入容器内的一种迭代器；如果你向它赋予一个元素值，你其实就是将它安插到容器内。如果写入第二个值，并不会覆盖第一个值，而是将该值安插进去。*Inserters* 在 7.4.2 节, p271 讨论。

7.2.3 Forward（前向）迭代器

Forward 迭代器是 Input 迭代器和 Output 迭代器的结合，具有 Input 迭代器的全部功能和 Output 迭代器的大部分功能。表 7.4 总结了 Forward 迭代器的所有操作。

表 7.4 Forward 迭代器的各项操作

表达式	效果
*iter	存取实际元素
iter->member	存取实际元素的成员
++iter	向前步进（传回新位置）
iter++	向前步进（传回旧位置）
iter1 == iter2	判断两个迭代器是否相等
iter1 != iter2	判断两个迭代器是否不等
TYPE()	产生迭代器（default 构造函数）
TYPE(iter)	复制迭代器（copy 构造函数）
iter1 = iter2	赋值

和 Input 迭代器及 Output 迭代器不同，Forward 迭代器能多次指向同一群集（*collection*）中的同一元素，并能多次处理同一元素。

你可能会奇怪，为什么 Forward 迭代器不具备 Output 迭代器的全部功能。这是因为有一个约束条件，使得某些对 Output 迭代器有效的程序对 Forward 迭代器无效：

- 面对 Output 迭代器，我们无需检查是否抵达序列尾端，便可直接写入数据。事实上由于 Output 迭代器不提供比较操作，所以你不能将 Output 迭代器和尾端迭代器相比较。因此，以下循环对 Output 迭代器 *pos* 而言是正确的：
// OK for output iterators
// ERROR for forward iterators

```
while (true) {  
    *pos = foo();  
    ++pos;  
}
```

- 对于 Forward 迭代器，你必须在提领 (dereference, 或说存取) 数据之前确保它有效。因此上述循环对 Forward 迭代器而言并不正确。这是因为它最终会提领 end(), 从而引发未定义行为。对于 Forward 迭代器, 上述循环应改为以下方式:

```
// OK for forward iterators  
// IMPOSSIBLE for output iterators  
while (pos != coll.end()) {  
    *pos = foo();  
    ++pos;  
}
```

这个循环不适用于 Output 迭代器, 因为 Output 迭代器没有定义 operator!=。

7.2.4 Bidirectional (双向) 迭代器

Bidirectional 迭代器在 Forward 迭代器的基础上增加了回头遍历的能力。换言之, 它支持递减运算符, 用以进行一步一步的后退操作 (表 7.5)。

表 7.5 Bidirectional 迭代器的新增操作

算式	效果
--iter	步退 (传回新位置)
iter--	步退 (传回老位置)

7.2.5 Random Access (随机存取) 迭代器

Random Access 迭代器在 Bidirectional 迭代器的基础之上再增加随机存取能力。因此它必须提供“迭代器算术运算” (和一般指针的“指针算术运算”相当)。也就是说, 它能加減某个偏移量、能处理距离 (differences) 问题, 并运用诸如 < 和 > 的相互关系运算符进行比较。表 7.6 列出 Random Access 迭代器的新增操作。

以下对象和型别支持 Random Access 迭代器:

- 可随机存取的容器 (vector, deque)
- strings (字符串, string, wstring)
- 一般 array (指针)

表 7.6 Random Access 迭代器的新增操作

算式	效果
<code>iter[n]</code>	存取索引位置为 <code>n</code> 的元素
<code>iter+=n</code>	向前跳 <code>n</code> 个元素 (如果 <code>n</code> 是负数, 则向后跳)
<code>iter-=n</code>	向后跳 <code>n</code> 个元素 (如果 <code>n</code> 是负数, 则向前跳)
<code>iter+n</code>	传回 <code>iter</code> 之后的第 <code>n</code> 个元素
<code>n+iter</code>	传回 <code>iter</code> 之后的第 <code>n</code> 个元素
<code>iter-n</code>	传回 <code>iter</code> 之前的第 <code>n</code> 个元素
<code>iter1-iter2</code>	传回 <code>iter1</code> 和 <code>iter2</code> 之间的距离
<code>iter1 < iter2</code>	判断 <code>iter1</code> 是否在 <code>iter2</code> 之前
<code>iter1 > iter2</code>	判断 <code>iter1</code> 是否在 <code>iter2</code> 之后
<code>iter1 <= iter2</code>	判断 <code>iter1</code> 是否不在 <code>iter2</code> 之后
<code>iter1 >= iter2</code>	判断 <code>iter1</code> 是否不在 <code>iter2</code> 之前

以下程序说明 Random Access 迭代器的特殊能力:

```
// iter/itercat.cpp

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

int main()
{
    vector<int> coll;

    // insert elements from -3 to 9
    for (int i=-3; i<=9; ++i) {
        coll.push_back (i);
    }

    /* print number of elements by processing the distance between beginning and end
    * - NOTE: uses operator - for iterators
    */
    cout << "number/distance: " << coll.end()-coll.begin() << endl;

    /* print all elements
    * - NOTE: uses operator < instead of operator !=
    */
    vector<int>::iterator pos;
```

```
    for (pos=coll.begin(); pos<coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;

    /* print all elements
    * - NOTE: uses operator [ ] instead of operator *
    */
    for (int i=0; i<coll.size(); ++i) {
        cout << coll.begin()[i] << ' ';
    }
    cout << endl;

    /* print every second element
    * - NOTE: uses operator +=
    */
    for (pos = coll.begin(); pos < coll.end()-1; pos += 2) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

程序输出如下:

```
number/distance: 13
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -1 1 3 5 7
```

本例对 `lists`, `sets` 和 `maps` 无效, 因为程序中标识 `NOTE` 的操作都只适用于 `Random Access` 迭代器。请特别注意, 只有在面对 `Random Access` 迭代器时, 你才能以 `operator<` 作为循环结束与否的判断准则。

请注意最后一个循环内的算式:

```
pos < coll.end()-1
```

这里要求 `coll` 至少包含一个元素。如果群集为空, `coll.end()-1` 便会位于 `coll.begin()` 之前。虽然如此一来上述比较操作可能仍然有效, 但严格说来, 将迭代器移至“起点更前面”会导致未定义行为。同样地, 如果表达式 `pos += 2` 将迭代器移至 `end()` 之后, 也会导致未定义行为。

因此, 按下面方法改变上述最后那个循环, 会十分危险, 因为如果群集内含偶数个元素, 会导致未定义行为 (图 7.2):

```
for (pos = coll.begin(); pos < coll.end(); pos += 2) {
    cout << *pos << ' ';
}
```

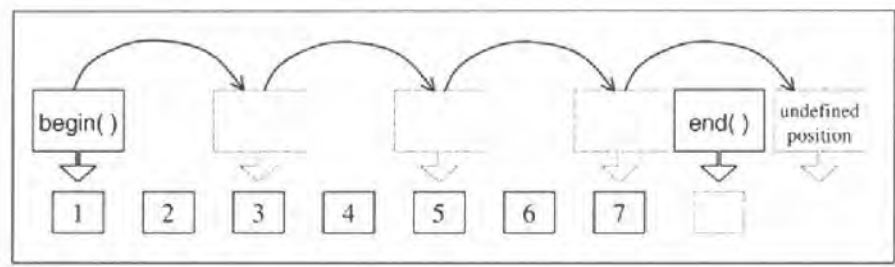


图 7.2 迭代器一次跳两个位置

7.2.6 Vector 迭代器的递增 (Increment) 和递减 (Decrement)

迭代器的递增和递减操作有个奇怪的问题。一般而言你可以递增或递减暂时性迭代器，但对于 `vectors` 和 `strings` 就不行。考虑下面的例子：

```
std::vector<int> coll;
...
// sort, starting with the second element
// - NONPORTABLE version
if (coll.size() > 1) {
    sort(++coll.begin(), coll.end());
}
```

通常编译 `sort()` 时会失败。但如果你换用 `deque` 取代 `vector`，就可以通过编译。有时 `vector` 也可以通过编译——取决于 `vector` 的具体实作。

这种奇怪问题的产生原因是，`vector` 的迭代器通常被实作为一般指针。要知道，C++ 不允许你修改任何基本型别（包括指针）的暂时值，但对于 `struct` 和 `class` 则允许。因此如果迭代器被实作为一般指针，编译会失败；如果被实作为 `class`，则编译可以成功。`deques`, `lists`, `sets` 和 `maps` 总是能通过编译，因为它们的迭代器不可能被实作为一般指针。至于 `vectors`，就要取决于实作手法了——通常其迭代器会被实作为一般指针，但如果你用的是一个“安全版本”的 STL，迭代器亦可能被实作为 `class`。为了保证可移植性，你不应该像上述例子那样地使用 `vectors`，应该使用辅助对象：

```
std::vector<int> coll;
...
// sort, starting with the second element
// - PORTABLE version
if (coll.size() > 1) {
    std::vector<int>::iterator beg = coll.begin();
    sort(++beg, coll.end());
}
```

这个问题其实并未如乍听之下那么糟糕，因为编译期就会检测出错误，所以不会发生未定义行为。不过这个问题够隐蔽了，得花点时间来解决。Strings 也有类似问题：尽管并非绝对，但 strings 迭代器也常被实作为一般的字符指针。

7.3 迭代器相关辅助函数

C++ 标准程序库为迭代器提供了三个辅助函数：advance(), distance() 和 iter_swap()。前二者提供给所有迭代器一些原本只有 Random Access 迭代器才有的能力：前进（或后退）多个元素，及处理迭代器之间的距离。第三个辅助函数允许你交换两个迭代器的值。

7.3.1 advance() 可令迭代器前进

advance() 可将迭代器的位置增加，增加的幅度由参数决定，也就是说使迭代器一次前进（或后退）多个元素：

```
#include <iterator>
void advance (InputIterator& pos, Dist n)
```

- 使名为 pos 的 Input 迭代器步进（或步退）n 个元素。
- 对 Bidirectional 迭代器和 Random Access 迭代器而言，n 可为负值，表示向后退。
- Dist 是个 template 型别。通常应该是个整数型别，因为会调用诸如 <, ++, -- 等操作，还要和 0 做比较。
- advance() 并不检查迭代器是否超过序列的 end()（因为迭代器通常不知道其所操作的容器，因此无从检查）。所以，调用 advance() 有可能导致未定义行为——因为“对着序列尾端调用 operator++”是一种未定义的操作行为。

此函数总能根据迭代器类型（category）采用最佳方案，这归功于迭代器特征（iterator traits）的运用（详见 7.5 节, p283）。面对 Random Access 迭代器，此函数只是简单地调用 pos+=n，因此，具有常量复杂度（constant complexity）。对于其它任何类型的迭代器则调用 ++pos（或 --pos，如果 n 为负值）n 次。因此，对于其它任何类型的迭代器，本函数具有线性复杂度（linear complexity）。

如果希望你的程序得以轻松更换容器和迭代器种类，你应该使用 `advance()` 而不是 `operator+=`。不过你必须意识到，这么做可能是拿性能来冒险。因为将 `advance()` 应用于“不提供 Random Access 迭代器”的容器中，你不太容易感觉到性能变差（但正是由于执行期性能不佳，Random Access 迭代器才会想要提供 `operator+=`）。另外请注意，`advance()` 没有回返值，而 `operator+=` 会传回新位置，所以后者可作为一个更大表达式的一部分。下面是 `advance()` 的运用实例：

```
// iter/advance1.cpp

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

int main()
{
    list<int> coll;

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

    list<int>::iterator pos = coll.begin();

    // print actual element
    cout << *pos << endl;

    // step three elements forward
    advance (pos, 3);

    // print actual element
    cout << *pos << endl;

    // step one element backward
    advance (pos, -1);

    // print actual element
    cout << *pos << endl;
}
```

在这个程序中, `advance()` 使迭代器 `pos` 前进三个元素, 再后退一个元素。程序输出如下:

```
1
4
3
```

使用 `advance()` 的另一个例子就是忽略“迭代器从输入流 (input stream) 中读出的某些数据”, 见 p282。

7.3.2 `distance()` 可处理迭代器之间的距离

函数 `distance()` 用来处理两个迭代器之间的距离:

```
#include <iterator>
Dist distance (InputIterator pos1, InputIterator pos2)
```

- 传回两个 Input 迭代器 `pos1` 和 `pos2` 之间的距离。
- 两个迭代器都必须指向同一个容器。
- 如果不是 Random Access 迭代器, 则从 `pos1` 开始往前走必须能够到达 `pos2`, 亦即 `pos2` 的位置必须与 `pos1` 相同或在其后。
- 回返值 `Dist` 的型别由迭代器决定 (详见第 7.5 节, p283):
`iterator_traits<InputIterator>::difference_type`

此函数能够根据迭代器类型 (category) 采取最佳实作手法, 这必须利用迭代器标志 (iterator tags) 才得以达成。面对 Random Access 迭代器, 此函数仅仅只是传回 `pos2-pos1`, 因此具备常数复杂度。对于其它迭代器类型, `distance()` 会不断递增 `pos1`, 直到抵达 `pos2` 为止, 然后传回递增次数。也就是说, 对于其它迭代器类型, `distance()` 具备线性复杂度。因此对于 non-Random Access 迭代器而言 `distance()` 的性能并不好, 应该尽力避免使用它。

7.5.1 节, p287 描述了 `distance()` 的实作内容, 下面是个使用实例:

```
// iter/distance.cpp

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

int main()
{
    list<int> coll;
```

```
// insert elements from -3 to 9
for (int i=-3; i<=9; ++i) {
    coll.push_back(i);
}

// search element with value 5
list<int>::iterator pos;
pos = find (coll.begin(), coll.end(),      // range
           5);                             // value

if (pos != coll.end()) {
    // process and print difference from the beginning
    cout << "difference between beginning and 5: "
          << distance(coll.begin(),pos) << endl;
}
else {
    cout << "5 not found" << endl;
}
}
```

find()将数值为 5 的元素的位置赋值给 pos。distance()计算出 pos 和起点位置间的距离。程序输出如下：

```
difference between beginning and 5: 8
```

如果你希望将来可以轻松更换容器型别和迭代器型别，你应该使用 distance() 而不是 operator-。不过这么一来你可能就不会意识到“将 Random Access 迭代器更换为其它类型的迭代器，会造成性能下降”这个问题了。

处理两个 non-Random Access 迭代器之间的距离时，必须十分小心。第一个迭代器绝对不能位于第二个迭代器之后，否则会导致未定义的行为。如果不知道哪个迭代器在前，你必须先算出两个迭代器分别至容器起点的距离，再根据这两个距离来判断。不过，你一定得清楚知道迭代器所指的目标容器为何。如果两个迭代器指向不同容器，那么任你如何挣扎，都无法摆脱“导致未定义行为”的悲惨命运。关于这个问题的另一些细节，请见 p99 对于子区间 (subranges) 的说明。

在 STL 早期版本中，distance() 的标记式 (signature) 与现在有所不同。它不传回距离值，而是透过第三参数传回。这么一来就无法直接在表达式中使用这个函数，很不方便。如果你不幸还在使用这个过时版本，请像这样包装一下：

```
// iter/distance.hpp
template <class Iterator>
inline long distance (Iterator pos1, Iterator pos2)
{
    long d = 0;
    distance (pos1, pos2, d);
    return d;
}
```

在这里，回返值的型别不取决于迭代器，而是被硬编码为 long。型别 long 通常够大，应该足以满足需求，不过没有人能保证这一点。

7.3.3 iter_swap() 可交换两个迭代器所指内容

下面这个简单的辅助函数用来交换两个迭代器所指的元素值。

译注：这里常出现的疑问是：究竟交换的是“迭代器的内容”或“迭代器所指元素的内容”？答案是后者。《STL 源码剖析》（侯捷著，华中科技大学出版社 2002）6.4.2 节对此有图解及源码展示。

```
#include <algorithm>
void iter_swap (ForwardIteraor1 pos1, ForwardIterator2 pos2)
```

- 交换迭代器 pos1 和 pos2 所指的值。
- 迭代器的型别不必相同，但其所指的两个值必须可以相互赋值。

这里有个简单例子（其中使用的函数 PRINT_ELEMENTS()，请看 5.7 节, p118）：

```
// iter/swap1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

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



```
    PRINT_ELEMENTS(coll);

    // swap first and second value
    iter_swap (coll.begin(), ++coll.begin());

    PRINT_ELEMENTS(coll);

    // swap first and last value
    iter_swap (coll.begin(), --coll.end());

    PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
1 2 3 4 5 6 7 8 9
2 1 3 4 5 6 7 8 9
9 1 3 4 5 6 7 8 2
```

注意, 如果以 `vector` 作为容器, 一般情况下此程序无法运作, 因为 `++coll.begin()` 和 `--coll.end()` 得到的是暂时指针 (详见 7.2.6 节, p258)。

7.4 迭代器配接器 (Iterator Adapters)

本节讨论迭代器配接器。此类特殊迭代器使得算法能够以逆向模式 (`reverse mode`)、安插模式 (`insert mode`) 进行工作, 也可以和流 (`streams`) 搭配工作。

7.4.1 Reverse (逆向) 迭代器

`Reverse` 迭代器是一种配接器, 重新定义递增运算和递减运算, 使其行为正好倒置。这么一来, 如果你使用这类迭代器, 算法将以逆向次序来处理元素。所有标准容器都允许使用 `Reverse` 迭代器来遍历元素。下面是个例子:

```
// iter/reviter1.cpp

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

```
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    list<int> coll;

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

    // print all elements in normal order
    for_each (coll.begin(), coll.end(), // range
              print);                  // operation
    cout << endl;

    // print all elements in reverse order
    for_each (coll.rbegin(), coll.rend(), // range
              print);                    // operations
    cout << endl;
}
```

容器的成员函数 `rbegin()` 和 `rend()` 各传回一个 **Reverse** 迭代器，它们就像 `begin()` 和 `end()` 的回返值一样，共同定义出一个半开区间。不同的是它们以相反的方向操作：

- `rbegin()` 传回逆向遍历的第一元素位置，也就是实际上最后一个元素的位置。
- `rend()` 传回逆向遍历时最后一个元素的下一个位置，也就是实际上第一元素的前一个位置。

迭代器和 Reverse（逆向）迭代器

你可以将一般迭代器转化成一个 **Reverse** 迭代器。当然，原本那个迭代器必须具有双向移动能力。注意，转换前后迭代器的逻辑位置发生了变化。考虑以下这个程序：

```
// iter/reviter2.cpp

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

int main()
{
    vector<int> coll;

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

    // find position of element with value 5
    vector<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),
                5);

    // print value to which iterator pos refers
    cout << "pos: " << *pos << endl;

    // convert iterator to reverse iterator rpos
    vector<int>::reverse_iterator rpos(pos);

    // print value to which reverse iterator rpos refers
    cout << "rpos: " << *rpos << endl;
}
```

程序输出如下:

```
pos: 5
rpos: 4
```

我们首先打印迭代器所指的元素值,然后将该迭代器转化成一个 Reverse 迭代器,再次打印它所指的元素,结果其值竟然变了。注意,这不是 bug,这是特性!导致这个行为的原因是区间的半开性。为了能够指定容器内的所有元素,我们必须运用“最后一个元素的下一位置”。然而对 Reverse 迭代器而言,这个位置位于第一元素之前。糟糕的是,这个位置可能并不存在,因为容器并不要求其第一元素之前的位置合法。例如 strings 和 array 也可视为容器,而语言本身并不保证 array 不可以从位置 0 开始。

为此, 逆向迭代器的设计者运用了一个小技巧: 他们实际上倒置了“半开原则”。逆向迭代器所定义区间, 实际上并不包括起点, 反倒是包括了终点。然而逻辑上其行为一如常态。这么一来, 逆向迭代器实际所指的元素位置, 和逻辑上所指的元素位置就并不一致 (图 7.3)。现在的问题是, 将一个迭代器转化为逆向迭代器的过程中发生了什么事? 该迭代器所保持 (履行) 的究竟是逻辑位置 (数值) 还是实际位置 (元素)? 从上面的例子可以了解, 后一种情况才是正确答案。因此, 其所指数值也就移到了前一个元素身上 (图 7.4)。

译注: 上段文字中所谓“将一个迭代器转化为逆向迭代器的过程中保持 (履行) 的是实际位置 (元素) 而非逻辑位置 (数值)”, 这话的意思就是, 以图 7.4 为例, 当 `pos` 转换为 `rpos`, 它们指向同一个实际地点, 但它们所代表的意义 (或说所代表的逻辑位置或数值) 却变得不同了, `*pos` 为 5, `*rpos` 为 4。

译注: 换一个角度看, 如果把 `[rbegin(), rend())` 形成的区间颠倒过来 (把书本上下颠倒), 你会发现其特性就和以 `[begin(), end())` 形成的区间完全一样。

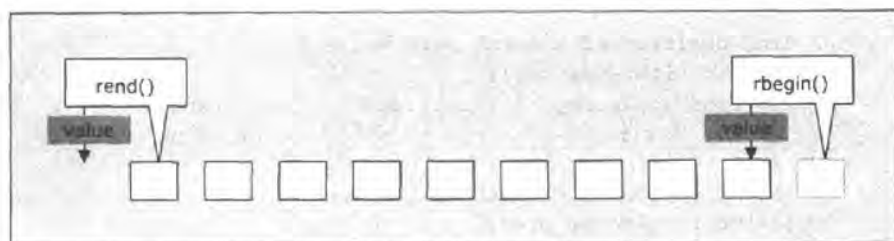


图 7.3 逆向迭代器的位置和所指数值

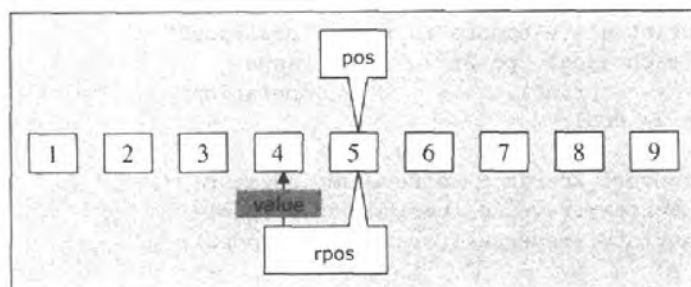


图 7.4 正向迭代器 `pos` 和 Reverse (逆向) 迭代器 `rpos` 之间的转换

不以为然吗? 喔, 它也有它的优点。如果你有个区间, 由一双 (而非一个) 迭代器定义出来, 那么变换操作就非常简单了, 所有元素仍然有效。考虑下面的例子:

```
// iter/reviter3.cpp

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

void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    deque<int> coll;

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

    // find position of element with value 2
    deque<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),    // range
                2);                          // value

    // find position of element with value 7
    deque<int>::iterator pos2;
    pos2 = find (coll.begin(), coll.end(),    // range
                7);                          // value

    // print all elements in range [pos1;pos2)
    for_each (pos1, pos2,                    // range
              print);                        // operation
    cout << endl;

    // convert iterator to reverse iterator
    deque<int>::reverse_iterator rpos1(pos1);
    deque<int>::reverse_iterator rpos2(pos2);
```

```

    // print all elements in range [pos1;pos2) in reverse order
    for_each (rpos2, rpos1,      // range
              print);           // operation
    cout << endl;
}

```

迭代器 `pos1` 和 `pos2` 定义了一个半开区间, 包括 2 而不包括 7。当你把描述这一区间的两个迭代器转化为 `Reverse` 迭代器时, 该区间仍然有效, 而且可以被逆序处理。程序输出如下:

```

2 3 4 5 6
6 5 4 3 2

```

`rbegin()` 其实很简单:

```

container::reverse_iterator(end())

```

`rend()` 也很简单:

```

container::reverse_iterator(begin())

```

当然啦, 常数迭代器 `const_iterator` 会被转化为 `const_reverse_iterator`。

以 `base()` 将逆向迭代器转回正常迭代器

你也可以将逆向迭代器转换为正常迭代器。逆向迭代器特别为此提供了一个 `base()` 成员函数:

```

namespace std {
    template <class iterator>
    class reverse_iterator ... {
        ...
        iterator base() const;
        ...
    };
}

```

下面是一个例子:

```

// iter/reviter4.cpp

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

```

```
int main()
{
    list<int> coll;

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

    // find position of element with value 5
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),      // range
               5);                             // value

    // print value of the element
    cout << "pos: " << *pos << endl;

    // convert iterator to reverse iterator
    list<int>::reverse_iterator rpos(pos);

    // print value of the element to which the reverse iterator refers
    cout << "rpos: " << *rpos << endl;

    // convert reverse iterator back to normal iterator
    list<int>::iterator rrpos;
    rrpos = rpos.base();

    // print value of the element to which the normal iterator refers
    cout << "rrpos: " << *rrpos << endl;
}
```

程序输出如下:

```
pos: 5
rpos: 4
rrpos: 5
```

可见, 透过 `base()` 进行转换:

```
*rpos.base()
```

就像转换为逆向迭代器的情形一样, 实际位置 (迭代器所指元素) 维持不动, 变化的是逻辑位置 (迭代器所指的数值)。你可以在 p353 找到另一个 `base()` 运用范例。

7.4.2 Insert (安插型) 迭代器

Insert 迭代器, 也称为 Inserters, 用来将“赋值新值”操作转换为“安插新值”操作。通过这种迭代器, 算法可以执行安插 (insert) 行为而非覆盖 (overwrite) 行为。所有 Insert 迭代器都隶属于 Output 迭代器类型。所以它只提供赋值 (assign) 新值的能力 (见 p253, 7.2.2 节) (译注: assign 就是 overwrite)。

Insert 迭代器的功能

通常算法会将数值赋值给标的迭代器。例如 copy() 算法 (见 p363):

```
namespace std {
    template <class Inputiterator, class Outputiterator>
    Outputiterator copy (Inputiterator from_pos, // beginning of source
                        Inputiterator from_end, // end of source
                        Outputiterator to_pos)   // beginning of dest.
    {
        while (from_pos != from_end) {
            *to_pos = *from_pos; // copy values
            ++from_pos;          // increment iterator
            ++to_pos;
        }
        return to_pos;
    }
}
```

循环不断执行, 直到 from_pos 抵达终点为止。在循环内, 源端迭代器 from_pos 被赋值给标的端迭代器 to_pos, 然后两个迭代器都递增。这里需要注意的是:

*to_pos = value

Insert 迭代器可以把上述这样的赋值动作转化为安插操作。不过实际上这里面有两个操作: 首先 operator* 传回迭代器当前位置, 然后再由 operator= 赋值新值。Insert 迭代器通常使用下面两个实作技巧:

1. operator* 被实作为一个无实际动作的动作 (no-op), 只是简单传回 *this。所以对 Insert 迭代器来说, *pos 与 pos 等价。
2. 赋值动作被转化为安插操作。事实上 Insert 迭代器会调用容器的 push_back(), push_front() 或 insert() 成员函数。

所以, 对于一个 Insert 迭代器, 你可以写 `pos=value`, 也可以写 `*pos=value`, 两者都可以插入新值。当然, 我这里谈的是 Insert 迭代器的实作细节, 从概念上讲, 正确的赋值表达式应该是 `*pos=value`。

类似情况, 递增运算符也被实作作为一个 no-op (无动作), 也是简单传回一个 `*this`。所以你可以改动 Insert 迭代器的位置。表 7.7 列出 Insert 迭代器的所有操作函数。

表 7.7 Insert 迭代器的操作

算式	效果
<code>*iter</code>	无实际操作 (传回 <code>iter</code>)
<code>iter = value</code>	安插 <code>value</code>
<code>++iter</code>	无实际操作 (传回 <code>iter</code>)
<code>iter++</code>	无实际操作 (传回 <code>iter</code>)

Insert 迭代器的种类

C++ 标准程序库提供三种 Insert 迭代器: `back inserters`, `front inserters`, `general inserters`。它们之间的区别在于插入位置。事实上它们各自调用所属容器中不同的成员函数。所以 Insert 迭代器初始化时一定要清楚知道自己所属的容器是哪一种。

每一种 Insert 迭代器都可以由一个对应的便捷函数加以生成和初始化。表 7.8 列出不同的 Insert 迭代器及其能力。

表 7.8 Insert 迭代器的种类

名称	Class	其所调用的函数	生成函数
Back Inserter	<code>back_insert_iterator</code>	<code>push_back(value)</code>	<code>back_inserter(cont)</code>
Front inserter	<code>front_insert_iterator</code>	<code>push_front(value)</code>	<code>front_inserter(cont)</code>
General Inserter	<code>insert_iterator</code>	<code>insert(pos,value)</code>	<code>inserter(cont,pos)</code>

当然, 容器本身必须支持 Insert 迭代器所调用的函数, 否则该种 Insert 迭代器就不可用。因此 `back inserters` 只能用在 `vectors`, `deques`, `lists`, `strings` 身上, `front inserters` 只能用在 `deques` 和 `lists` 身上。下面几个小节详细描述各种 Insert 迭代器。

Back Inserters

`Back inserters` (或称 `back insert iterator`) 透过成员函数 `push_back()` (细节详

见 p241)，将一个元素值追加于容器尾部。由于 `push_back()` 只存在于 `vectors`, `deques`, `lists`, `strings` 之中，所以 C++ 标准函数库中也只有这些容器支持 `back inserters`。

`back inserter` 生成时必须根据其所属容器进行初始化。函数 `back_inserter()` 为此提供了一个方便途径。以下例子展示 `back inserters` 的用法：

```
// iter/backins.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    // create back inserter for coll
    // - inconvenient way
    back_insert_iterator<vector<int> > iter(coll);

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll);

    // create back inserter and insert elements
    // - convenient way
    back_inserter(coll) = 44;
    back_inserter(coll) = 55;

    PRINT_ELEMENTS(coll);

    // use back inserter to append all elements again
    // - reserve enough memory to avoid reallocation
    coll.reserve(2*coll.size());
    copy (coll.begin(), coll.end(), // source
          back_inserter(coll)); // destination

    PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
1 2 3
1 2 3 44 55
1 2 3 44 55 1 2 3 44 55
```

注意,你一定要在调用 `copy()` 之前确保有足够大的空间。因为 `back inserter` 在安插元素时,可能会造成指向该 `vector` 的其它迭代器失效。因此,如果不保留足够空间,这个算法可能会形成“源端迭代器失效”状态。

`Strings` 也提供 STL 容器接口,包括 `push_back()`。所以你也可以利用 `back_inserter` 为 `string` 追加字符,参见 p502 实例。

Front Inseters

`Front inseters` (或称 `front insert iterator`) 透过成员函数 `push_front()` (细节详见 p241) 将一个元素值加在容器头部。由于 `push_back()` 只在 `deque`s 和 `lists` 中有所实现,所以 C++ 标准函数库中也就只有这两个容器支持 `front inseters`。

`front inserter` 生成时必须根据其所属容器进行初始化。函数 `front_inserter()` 为此提供了一个方便途径。以下展示 `front inseters` 的用法:

```
// iter/frontins.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // create front inserter for coll
    // - inconvenient way
    front_insert_iterator<list<int> > iter(coll);

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll);
```

```
// create front inserter and insert elements
// - convenient way
front_inserter(coll) = 44;
front_inserter(coll) = 55;

PRINT_ELEMENTS(coll);

// use front inserter to insert all elements again
copy (coll.begin(), coll.end(), // source
      front_inserter(coll));    // destination

PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
3 2 1
55 44 3 2 1
1 2 3 44 55 55 44 3 2 1
```

注意, 安插多个元素时, front inserter 以逆序方式插入, 这是因为它总是将后一个元素安插于前一个元素的前面。

General Inserters

General Inserter (或称 general insert iterator)²根据两个参数而初始化: (1) 容器 (2) 待安插位置。迭代器内部以“待安插位置”为参数, 调用成员函数 insert()。便捷函数 inserter() 则是提供了更方便的手段来产生 general inserter 并初始化。

General Inserter 对所有标准容器均适用, 因为所有容器都有 insert() 成员函数。然而对关联式容器 (sets 和 maps) 而言, 安插位置只是个提示, 因为在这两种容器中, 元素的真正位置视其实值 (或键值) 而定。请参考 p240 关于 insert() 的描述。

² 通常我们把 general inserter 简称为 insert iterator 或 inserter。这就是说, insert iterator 这个词与 inserter 的含义相同, 都是泛指所有 insert 迭代器。同时它也用来特别指出某一类特别的 insert 迭代器 (可安插于某个特定位置, 而不只是安插于头部或尾部), 这就是本书所说的 general inserter。

安插操作完成后, `general_inserter` 获得刚刚被安插的那个元素的位置。实际上相当于调用以下语句:

```
pos = container.insert(pos,value);  
++pos;
```

为什么要将 `insert()` 的传回值赋值给 `pos` 呢? 这是为了确保该迭代器的位置始终有效。如果没有这个赋值动作, 在 `deque`, `vectors` 和 `strings` 中, 该 `general_inserter` 本身可能会失效。因为每一次安插操作都会 (或至少可能会) 使指向容器的所有迭代器失效。

下面例子展示了 `general_inserter` 的用法:

```
// iter/inserter.cpp  
  
#include <iostream>  
#include <set>  
#include <list>  
#include <algorithm>  
#include "print.hpp"  
using namespace std;  
  
int main()  
{  
    set<int> coll;  
  
    // create insert iterator for coll  
    // - inconvenient way  
    insert_iterator<set<int> > iter(coll,coll.begin());  
  
    // insert elements with the usual iterator interface  
    *iter = 1;  
    iter++;  
    *iter = 2;  
    iter++;  
    *iter = 3;  
  
    PRINT_ELEMENTS(coll,"set: ");  
  
    // create inserter and insert elements  
    // - convenient way  
    inserter(coll,coll.end()) = 44;  
    inserter(coll,coll.end()) = 55;
```

```
PRINT_ELEMENTS(coll, "set: ");

// use inserter to insert all elements into a list
list<int> coll2;
copy (coll.begin(), coll.end(),           // source
      inserter(coll2, coll2.begin()));    // destination

PRINT_ELEMENTS(coll2, "list: ");

// use inserter to reinsert all elements into the list before the second element
copy (coll.begin(), coll.end(),           // source
      inserter(coll2, ++coll2.begin()));  // destination

PRINT_ELEMENTS(coll2, "list: ");
}
```

程序输出如下:

```
set: 1 2 3
set: 1 2 3 44 55
list: 1 2 3 44 55
list: 1 1 2 3 44 55 2 3 44 55
```

上述的 `copy()` 调用动作用来展示 `general inserter` 维护着元素的次序。第二个 `copy()` 调用动作将使用区间中的一个适当位置作为参数。

关联式容器 (Associative Containers) 的定制型 `Inserter`

如前所述, 对于关联式容器, `general inserters` 的“位置参数”只不过是提示, 用来强化速度。如果使用不当反而可能导致比较糟的性能。举个例子, 如果逆序安插, 这个提示就可能使程序变慢, 因为它使得程序总是从错误的位置开始搜寻安插点。这是为 C++ 标准程序库提供辅助措施的大好时机, 7.5.2 节, p288 有一个为此而作的扩展机制。

7.4.3 Stream (流) 迭代器

`Stream` 迭代器是一种迭代器配接器, 通过它, 你可以把 `stream` 当成算法的原点和终点。更明确地说, 一个 `istream` 迭代器可用来从 `input stream` 中读取元素, 而一个 `ostream` 迭代器可以用来对 `output stream` 写入元素。

`Stream` 迭代器的一种特殊形式是所谓的 `stream 缓冲区迭代器`, 用来对 `stream 缓冲区` 进行直接读取和写入操作。`stream 缓冲区迭代器` 在 13.13.2 节, p665 讨论。

Ostream 迭代器

ostream 迭代器可以将被赋予的值写入 output stream 中。它的实作机制与 p271 所说的 insert 迭代器概念一致，唯一的区别在于 ostream 迭代器将赋值操作转化为 operator<<。如此一来算法就可以使用一般的迭代器接口直接对 stream 执行涂写动作。表 7.9 列出 ostream 迭代器的各个操作函数。

表 7.9 ostream 迭代器的各项操作

算式	效果
ostream_iterator<T>(ostream)	为 ostream 产生一个 ostream 迭代器
ostream_iterator<T>(ostream, delim)	为 ostream 产生一个 ostream 迭代器，各元素间以 delim 为分隔符（请注意，delim 的型别是 const char*）
*iter	无实际操作（传回 iter）
iter = value	将 value 写到 ostream，像这样： ostream<<value。其后再输出一个 delim（分隔符；如有定义的话）
++iter	无实际操作（传回 iter）
iter++	无实际操作（传回 iter）

产生 ostream 迭代器时，你必须提供一个 output stream 作为参数，迭代器将会把元素值写到该 output stream 身上。另一个参数可有可无，是个字符串，被用来作为每个元素值之间的分隔符。

ostream 迭代器是针对尚未确定的元素型别 T 而定义：

```
namespace std {
    template <class T,
              class charT = char,
              class traits = char_traits<charT> >
        class ostream_iterator;
}
```

带有默认值的第二和第三个 template 参数用以确定将被使用的 stream 型别（参见 13.2.1 节, p590）³。

³ 在比较老旧的系统中，这些 template 参数并不存在。

以下实例展示 ostream 迭代器的用法:

```
// iter/ostriter.cpp

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

int main()
{
    // create ostream iterator for stream cout
    // - values are separated by a newline character
    ostream_iterator<int> intWriter(cout, "\n");

    // write elements with the usual iterator interface
    *intWriter = 42;
    intWriter++;
    *intWriter = 77;
    intWriter++;
    *intWriter = -5;

    // create collection with elements from 1 to 9
    vector<int> coll;
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // write all elements without any delimiter
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout));
    cout << endl;

    // write all elements with " < " as delimiter
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " < "));
    cout << endl;
}
```


程序输出如下:

```
42
77
-5
123456789
1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 <
```

注意, 分隔符的型别是 `const char*`。如果你传递一个 `string` 对象进去, 别忘了调用 `c_str()` 成员函数以获得正确型别 (参见 11.3.6 节, p513)。例如:

```
string delim;
...
ostream_iterator<int>(cout, delim.c_str());
```

istream 迭代器

`istream` 迭代器是 `ostream` 迭代器的拍档, 用来从 `input stream` 读取元素。透过 `istream` 迭代器, 算法可以从 `stream` 中直接读取数据。然而 `istream` 迭代器较之 `ostream` 迭代器稍微复杂一些 (这很正常, 读本来就比写复杂一些)。

产生 `istream` 迭代器时, 你必须提供一个 `input stream` 作为参数, 迭代器将从其中读取数据。然后它便经由 `input` 迭代器的一般接口 (7.2.1 节, p252), 利用 `operator>>` 读取元素。然而, 读取动作有可能失败 (可能因为读到文件尾部, 或读取错误), 此外算法也需要知道区间是否已到终点。为了解决这些问题, 你可以使用一个 *end-of-stream* 迭代器, 它可以利用 `istream` 迭代器的 `default` 构造函数生成。只要有任何一次读取失败, 所有 `istream` 迭代器都会变成 *end-of-stream* 迭代器。所以进行一次读取后, 你就应该将 `istream` 迭代器拿来和 *end-of-stream* 迭代器比较一番, 看看这个迭代器是否有合法值。表 7.10 列出 `istream` 迭代器的所有操作。

注意, `istream` 迭代器的构造函数会将 `stream` 打开, 读取第一个元素 (译注: 《STL 源码剖析》8.3.3 节, p443 的源码及其注释, 以及其后文字都可清楚看出为什么如此)。所以请注意, 在确实需要用到一个 `istream` 迭代器之前, 别过早定义它。上述做法是必要的, 否则一旦 `operator*` 被调用, 就无法传回第一个元素了。不过, 某些实作版本可能会延缓第一次读取操作, 直到第一次 `operator*` 被调用。

`istream` 迭代器是针对未定型别 `T` 定义的:

```
namespace std {
    template <class T,
              class charT = char,
              class traits = char_traits<charT>,
              class Distance = ptrdiff_t>
        class istream_iterator;
}
```

表 7.10 istream 迭代器的各项操作

算式	效果
<code>istream_iterator<T>()</code>	产生一个 <i>end-of-stream</i> 迭代器
<code>istream_iterator<T>(istream)</code>	为 <code>istream</code> 产生一个迭代器 (可能立刻读取第一个元素)
<code>*iter</code>	传回先前读取的值 (如果构造函数并未立刻读取 第一个元素值, 则本式执行读取任务)
<code>iter->member</code>	传回先前读取的元素的成员 (如果有的话)
<code>++iter</code>	读取下一个元素, 并传回其位置
<code>iter++</code>	读取下一个元素, 并传回迭代器指向前一个元素
<code>iter1 == iter2</code>	检查 <code>iter1</code> 和 <code>iter2</code> 是否相等
<code>iter1 != iter2</code>	检查 <code>iter1</code> 和 <code>iter2</code> 是否不相等

拥有默认值的第二和第三 `template` 参数, 用来确定 `stream` 型别 (此中含义请见 13.2.1 节, p590)。第四个 `template` 参数 (也有默认值) 用来指定 “迭代器距离 (difference)” 的表示型别⁴。

如果满足以下条件, 我们便说两个 `istream` 迭代器相等:

- 两者都是 *end-of-stream* 迭代器 (因而不能再进行读取操作), 或
- 两者都可以再进行读取操作, 并指向相同的 `stream`

以下实例展示 `istream` 迭代器的各项操作:

```
// iter/istriter.cpp

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

int main()
{
    // create istream iterator that reads integers from cin
    istream_iterator<int> intReader(cin);

    // create end-of-stream iterator
    istream_iterator<int> intReaderEOF;
    /* while able to read tokens with istream iterator
```

⁴ 老旧系统中并不支持 `template` 参数默认值, 因此要求两个明确的 `template` 参数, 其中第二参数就是这里所说的第四参数, 而用来设定 `stream` 型别的参数则不存在。

```
* - write them twice
*/
while (intReader != intReaderEOF) {
    cout << "once: " << *intReader << endl;
    cout << "once again: " << *intReader << endl;
    ++intReader;
}
}
```

如果执行这个程序并输入以下给予值:

```
1 2 3 f 4
```

程序输出如下:

```
once:      1
once again: 1
once:      2
once again: 2
once:      3
once again: 3
```

如你所见, 字符 `f` 的输入导致程序结束。是的, 由于格式错误, `stream` 不再处于 `good` 状态, 于是 `istream` 迭代器 `intReader` 就和 `end-of-stream` 迭代器 `intReaderEOF` 相同, 使得循环条件为 `false`, 进而结束循环。

Stream 迭代器的另一个例子

以下例子使用两种 `stream` 迭代器, 而且还用到了迭代器函数 `advance()`:

```
// iter/advance2.cpp

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

int main()
{
    istream_iterator<string> cinPos(cin);
    ostream_iterator<string> coutPos(cout, " ");
```

```
/* while input is not at the end of the file
 * - write every third string
 */
while (cinPos != istream_iterator<string>()) {
    // ignore the following two strings
    advance (cinPos, 2);

    // read and write the third string
    if (cinPos != istream_iterator<string>()) {
        *coutPos++ = *cinPos++;
    }
}
cout << endl;
}
```

迭代器便捷操作函数 `advance()` 用来帮助迭代器前进到另一位置 (参见 7.3.1 节, p259)。当它被用来搭配 `istream` 迭代器, 结果便是“跳过输入记号(input tokens)”。如果你的输入如下⁵:

```
No one objects if you are doing
a good programming job for
someone who you respect.
```

则输出如下:

```
objects are good for you
```

千万别忘了, 调用 `advance()` 之后、存取 `*cinPos` 之前, 一定要记得检查 `istream` 迭代器是否依然合法, 如果对 *end-of-stream* 迭代器调用 `operator*`, 会导致未定义的行为。

p107, p366, p285 还有一些例子, 其中并展示算法如何运用 `stream` 迭代器对 `stream` 进行读写。

7.5 迭代器特性 (Iterator Traits)

译注: `Iterator traits` 是掌握 STL 编程技术 (注意, 不是 STL 应用) 的关键, `traits` 技法更是 C++ 高级运用。来龙去脉可参考《STL 源码剖析》3.4 节的 `iterator traits` 和 3.7 节的 `type traits`。

迭代器可以区分为不同类型, 每个类型都具有特定的迭代器功能。如果能根据不同的迭代器类型, 将操作行为重载, 将会很有用, 甚至很必要。透过迭代器标志 (*tags*) 和特性 (*traits*, 由 `<iterator>` 提供) 可以实现这样的重载。

⁵ 感谢 Andrew Koenig 提供了这个例子。

C++ 标准程序库为每一种迭代器提供了一个迭代器标志 (*iterator tag*)，用来作为迭代器的标签 (*label*)：

```
namespace std {
    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 {
    };
}
```

请注意，这里用到了继承，所以我们可以说，任何 Forward 迭代器都是一种 (*is-a*) Input 迭代器。然而请注意，Forward 迭代器标志只派生自 Input 迭代器标志，和 Output 迭代器标志无关。事实上 Forward 迭代器的某些特性的确不符合 Output 迭代器的要求，详见 p254。

如果你撰写泛型程序代码，可能不只对迭代器类型感兴趣。你可能还需要了解迭代器所指元素的型别。C++ 标准程序库提供了一种特殊的 `template` 结构来定义所谓的迭代器特性 (*iterator traits*)，该结构包含迭代器相关的所有信息，为“迭代器应具备的所有型别定义（包括迭代器类型、元素型别等等）”提供一致的接口：

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

在这个 `template` 中，`T` 表示迭代器型别。有了它，我们就可以撰写任何运用“迭代器类型或其元素型别”等特征的泛型程序代码。例如以下表达式就可以取得迭代器 `T` 的元素数值型别：

```
typename std::iterator_traits<T>::value_type
```

这个“迭代器特性”结构有两个优点：

1. 能够确保每一个迭代器提供所有必要的型别定义。
2. 能够针对特定的迭代器进行特化 (specialization)。

上述第二条用于“以一般指针作为迭代器”时：

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

于是，针对“指向 T 型别”的一般指针，上述特化版本告诉我们，该型别具有 Random Access 迭代器类型。除了上述这个特化版本，面对常数型指针 (const T*) 也应该再准备一个相应的特化定义。

7.5.1 为迭代器编写泛型函数

通过迭代器特征 (iterator traits)，你可以撰写这样的泛型函数：根据迭代器类型而派生型别定义或采用不同的实作码。

运用迭代器型别 (iterator typs)

某些算法函数内部需要一个以元素型别为型别的暂时变量，这正是使用

“迭代器特性”的一个简单例子。这样的一个暂时变量可以声明如下：

```
typename std::iterator_traits<T>::value_type tmp;
```

其中 T 是迭代器型别。

另一个例子是将元素环形移动：

```
template <class Forwarditerator>
void shift_left (Forwarditerator beg, Forwarditerator end)
{
    // temporary variable for first element
    typedef typename
    std::iterator_traits<Forwarditerator>::value_type value_type;

    if (beg != end) {
        // save value of first element
        value_type tmp(*beg);
```

```
        // shift following values
        ...
    }
}
```

运用迭代器类型 (Iterator Categories)

如果希望针对不同的迭代器类型采用不同的实作方案，你需要按下面两步来进行：

1. 让你的 `template` 函数将迭代器类型作为附加参数，调用另一个函数。例如：

```
template <class iterator>
inline void foo (iterator beg, iterator end)
{
    foo (beg, end,
        std::iterator_traits<iterator>::iterator_category{});
}
```

2. 针对不同的迭代器类型实作出上述所调用的函数。只有“并非派生自其它迭代器类型”的迭代器类型，才需要提供特化版本。例如：

```
// foo() for bidirectional iterator
template <class BiIterator>
void foo (BiIterator beg, BiIterator end,
         std::bidirectional_iterator_tag)
{
    ...
}

// foo() for random access iterator
template <class RaIterator>
void foo (RaIterator beg, RaIterator end,
         std::random_access_iterator_tag)
{
    ...
}
```

其中，针对 `Random Access` 迭代器而做的版本，可以使用随机存取操作，针对 `Bidirectional` 迭代器而做的版本则不可以。现在，根据迭代器标志 (tags) 阶层体系 (p283)，你可以针对多个迭代器类型，综合出一个实作方案。

distance() 的实作

以下实例系根据稍早所说步骤，实作出迭代器辅助函数 distance()。此函数传回两个迭代器所指的元素位置的间距(参见 7.3.2 节, p261)。对于 Random Access 迭代器，此函数只是简单运用 operator- 便可以获得结果，对于其它任何迭代器类型，此函数必须一步一步前进至区间终点，才有能力计算出结果并传回。

```
// general distance()

template <class iterator>
typename std::iterator_traits<iterator>::difference_type
distance (iterator pos1, iterator pos2)
{
    return distance (pos1, pos2,
                     std::iterator_traits<iterator>
                     ::iterator_category());
}

// distance() for random access iterator
template <class Riterator>
typename std::iterator_traits<Riterator>::difference_type
distance (Riterator pos1, Riterator pos2,
         std::random_access_iterator_tag)
{
    return pos2 - pos1;
}

// distance() for input, forward, and bidirectional iterator
template <class Initerator>
typename std::iterator_traits<Initerator>::difference_type
distance (Initerator pos1, Initerator pos2,
         std::input_iterator_tag)
{
    typename std::iterator_traits<Initerator>::difference_type d;
    for (d=0; pos1 != pos2; ++pos1, ++d) {
        ;
    }
    return d;
}
```


回返值型别必须是迭代器距离型别 (difference type)。请注意第二版本使用 input 迭代器, 所以该版本对 Forward 迭代器、Bidirectional 迭代器都有效, 因为这两种迭代器的标志 (tag) 是从 input_iterator_tag 中派生出来的。

7.5.2 使用者自定义 (User-Defined) 的迭代器

让我们自己来写个迭代器。如前所述, 需要为这样一个迭代器提供特性 (traits)。有两种办法可行:

1. 提供必要的五种型别定义, 就像 iterator_traits 结构所描述 (p284)。
2. 提供一个特化版本的 iterator_traits 结构。

关于第一种方法, C++ 标准程序库提供了一个特殊基础类别 (base class) iterator<>, 专门用来进行这类型别定义。你只需这样指定型别⁶:

```
class Myiterator
: public std::iterator <std::bidirectional_iterator_tag,
                      type, std::ptrdiff_t, type*, type&> {
    ...
};
```

其中第一个 template 参数用来定义迭代器类型, 第二个参数用来定义元素型别, 第三个参数用来定义距离型别, 第四个参数用来定义 pointer 型别, 第五个参数用来定义 reference 型别。最后三个参数有默认值 ptrdiff_t, type*, type&, 通常这样使用就够了:

```
class Myiterator
: public std::iterator <std::bidirectional_iterator_tag, type> {
    ...
};
```

以下实例说明如何编写自定义的迭代器。这是一个关联式容器的 insert 迭代器, 下面是其实作代码:

```
// iter/assoiter.hpp

#include <iterator>

/* template class for insert iterator for associative containers
*/
```

⁶ STL 早期版本并没有这样的基础类别 iterator, 而是提供了辅助型别 input_iterator, output_iterator, forward_iterator, bidirectional_iterator, random_access_iterator。

```

template <class Container>
class asso_insert_iterator
    : public std::iterator <std::output_iterator_tag,
                          void, void, void, void>
{
protected:
    Container& container; // container in which elements are inserted

public:
    // constructor
    explicit asso_insert_iterator (Container& c) : container(c) {
    }

    // assignment operator
    // - inserts a value into the container
    asso_insert_iterator<Container>&
    operator= (const typename Container::value_type& value) {
        container.insert(value);
        return *this;
    }

    // dereferencing is a no-op that returns the iterator itself
    asso_insert_iterator<Container>& operator* () {
        return *this;
    }

    // increment operation is a no-op that returns the iterator itself
    asso_insert_iterator<Container>& operator++ () {
        return *this;
    }

    asso_insert_iterator<Container>& operator++ (int) {
        return *this;
    }
};

/* convenience function to create the inserter
*/
template <class Container>
inline asso_insert_iterator<Container> asso_inserter (Container& c)
{
    return asso_insert_iterator<Container>(c);
}

```

类别 `asso_insert_iterator` 是从类别 `iterator` 派生出来。传给 `iterator` 的第一个参数 `output_iterator_tag` 指定了迭代器类型。`Output` 迭代器只能用来写入，因此，如同所有的 `Output` 迭代器一样，元素型和距离型别都是 `void`⁷。

生成之时，此迭代器将容器储存于其成员 `container` 中。所有经由赋值 (*assign*) 而来的值都通过 `insert()` 安插到容器内。`operator*` 和 `operator++` 只是传回迭代器本身，并无任何实际动作，但这么虚晃一招使得迭代器得以维护其控制权。如果下面这样的迭代器一般性接口被使用了：

```
*pos = value
```

`*pos` 于是传回 `*this`，新值被赋值至其中。至于赋值动作 (`operator=`) 则被转化为对容器成员函数 `insert(value)` 的调用。

这一 `Insert` 迭代器定义完成后，还可以提供一个函数 `asso_inserter()`，用来简化迭代器的生成动作和初始化动作。以下程序便是使用这样一个 `inserter` 对着一个 `set` 安插一些元素：

```
// iter/assoiter.cpp

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

#include "print.hpp"

#include "assoiter.hpp"

int main()
{
    set<int> coll;

    // create inserter for coll
    // - inconvenient way
    asso_insert_iterator<set<int>> iter(coll);
```

⁷ 对早期的 STL 版本而言，`asso_insert_iterator` 类别必须派生自无任何 `template` 参数的 `output_iterator` 类别。

```
// insert elements with the usual iterator interface
*iter = 1;
iter++;
*iter = 2;
iter++;
*iter = 3;

PRINT_ELEMENTS(coll);

// create inserter for coll and insert elements
// - convenient way
asso_inserter(coll) = 44;
asso_inserter(coll) = 55;

PRINT_ELEMENTS(coll);

// use inserter with an algorithm
int vals[] = { 33, 67, -4, 13, 5, 2 };
copy (vals, vals+(sizeof(vals)/sizeof(vals[0])), // source
      asso_inserter(coll));                      // destination

PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
1 2 3
1 2 3 44 55
-4 1 2 3 5 13 33 44 55 67
```


8

STL 仿函数 (functors)

(又名函数对象, function objects)

本章详细介绍函数对象 (function objects, 另名仿函数 functors)。5.9 节, p124 也有一些介绍。本章涵盖 STL 预定义的所有仿函数和函数配接器 (function adapters), 以及功能复合 (functional composition) 概念, 并提供我自己写的一些仿函数实例。

译注: 本书英文版使用 function objects 一名, 但译为“函数对象”后, 易与前后文夹杂, 失去术语之独特性。因此我全部改用另一名称 functor, 译为“仿函数”。必要时同时加注两个名称。

8.1 仿函数 (functor) 的概念

所谓仿函数, 是一个定义了 `operator()` 的对象。下面这个例子:

```
FunctionObjectType fo;  
...  
fo(...);
```

其中表达式 `fo()` 系调用仿函数 `fo` 的 `operator()`, 而非调用函数 `fo()`。

你可以将仿函数视为一般函数, 只不过用的是一种更复杂的撰写手段: 并非将所有语句放在函数体中:

```
void fo() {  
    statements  
}
```

而是在仿函数类别的 `operator()` 体内撰写程序代码:

```
class FunctionObjectType {  
public:  
    void operator() () {  
        statements  
    }  
};
```

这种定义形式显然更为复杂, 却有三大妙处:

1. 仿函数比一般函数更灵巧, 因为它可以拥有状态 (state)。事实上对于仿函数, 你可以同时拥有两个状态不同的实体 (instance)。一般函数则力未能逮。
2. 每个仿函数都有其型别。因此你可以将仿函数的型别当做 template 参数来传递, 从而指定某种行为模式。此外还有一个好处: 容器型别也会因为仿函数的不同而不同。
3. 执行速度上, 仿函数通常比函数指针更快。

p126 详细介绍了这些优点, p127 则透过一个例子展示仿函数何以能够比一般函数更灵巧。

接下来的两个小节, 我会再展示两个例子, 详细讨论仿函数。第一个例子示范如何利用“仿函数拥有属于自己的独特型别”这一优势。第二个例子示范如何从“仿函数拥有属于自己的独特状态”这一性质获利, 并引出 `for_each()` 算法的一个有趣特性, 涵盖于另一小节。

8.1.1 仿函数可当做排序准则 (Sort Criteria)

程序员经常需要将某些 class objects 以已序 (sorted) 形式置于容器中 (例如一个 persons 群集)。然而, 或许是不想, 或许是不能, 反正你无法使用一般的 `operator<` 来对这些对象排序。你必须以某种特别规则 (通常基于某些成员函数) 来排序。此时仿函数可以派上用场, 试看下列:

```
// fo/sort1.cpp

#include <iostream>
#include <string>
#include <set>
#include <algorithm>
using namespace std;

class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

/* class for function predicate
 * - operator () returns whether a person is less than another person
 */
```

```
class PersonSortCriterion {
public:
    bool operator() (const Person& p1, const Person& p2) const {
        /* a person is less than another person
        * - if the last name is less
        * - if the last name is equal and the first name is less
        */
        return p1.lastname() < p2.lastname() ||
            (!(p2.lastname() < p1.lastname()) &&
            p1.firstname() < p2.firstname());
    }
};

int main()
{
    // declare set type with special sorting criterion
    typedef set<Person, PersonSortCriterion> PersonSet;

    // create such a collection
    PersonSet coll;
    ...

    // do something with the elements
    PersonSet::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        ...
    }
    ...
}
```

这里的 coll (一个 set) 使用了特殊排序准则 PersonSortCriterion, 而它正是一个仿函数类别。PersonSortCriterion 这么定义 operator(): 首先比较两人的姓, 如果相等再比较其名。coll 的构造函数会自动产生 class PersonSortCriterion 的一个实体 (instance), 所有元素都将以此为排序准则, 进行排序。

注意, 这里的排序准则 PersonSortCriterion 是个类别, 所以你可以把它当做 set 的 template 参数。如果以一般函数来担任排序准则, 就无法做到这一点 (参见 p123)。

“以上述型别作为排序准则”的所有 sets，都拥有自己的独特型别（本例名为 PersonSet）。你无法将这个 set 拿来和“拥有不同排序准则”的其它 set 合并或互相赋值。所以，虽然无论你怎么做都无法回避 set 的自动排序特性，但是你可以设计出“型别一致、排序准则不同”的仿函数（见下节）。关于 sets 及其排序准则，参见 p178。

8.1.2 拥有内部状态 (Internal State) 的仿函数

下面这个例子展示仿函数如何模拟函数在同一时刻下拥有多个状态：

```
// fo/general.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class IntSequence {
private:
    int value;
public:
    // constructor
    IntSequence (int initialValue)
        : value(initialValue) {
    }

    // "function call"
    int operator() () {
        return value++;
    }
};

int main()
{
    list<int> coll;

    // insert values from 1 to 9
    generate_n (back_inserter(coll),          // start
               9,                             // number of elements
               IntSequence(1));               // generates values

    PRINT_ELEMENTS(coll);
```

```
// replace second to last element but one with values
// starting at 42
generate (++coll.begin(),          // start
         --coll.end(),           // end
         IntSequence(42)); // generates values
PRINT_ELEMENTS(coll);
}
```

本例以仿函数产生一个整数序列。每当仿函数的 `operator()` 被调用，就返回一个整数值并累加 1。至于初始值，可透过构造函数的参数加以指定。

本例有两个这样的仿函数，分别被 `generate()` 和 `generate_n()` 算法运用。这两个算法的作用是产生数值以供写入群集之内。以下语句：

```
generate_n (back_inserter(coll),
           9,
           IntSequence(1));
```

其中的表达式：

```
IntSequence(1)
```

便是以 1 为初值，产生这样一个仿函数。`generate_n()` 算法前后共 9 次运用它改写元素值，而它也老老实实在地产生数值 1~9。同样道理，以下表达式：

```
IntSequence(42)
```

会产生出以 42 为起始值的整数序列。`generate()` 算法运用这些值改写了“起自 `++coll.begin()`，终于 `--coll.end()`”的元素¹。程序输出如下：

```
1 2 3 4 5 6 7 8 9
1 42 43 44 45 46 47 48 9
```

运用其它 `operator()` 版本，你可以轻松产出更复杂的序列。

¹ 表达式：

```
++coll.begin()
和
--coll.end()
```

在 `vector` 中不能有效运作。这个难缠的问题在 7.2.6, p258 讨论过。

仿函数是 *passed by value* (传值), 不是 *passed by reference* (传址), 因此算法并不会改变随参数而来的仿函数的状态。下面例子产生的两个序列都以 1 开始:

```
IntSequence seq(1); // integral sequence starting with value 1

// insert sequence beginning with 1
generate_n (back_inserter(coll), 9, seq);

// insert sequence beginning with 1 again
generate_n (back_inserter(coll), 9, seq);
```

将仿函数以 *by value* 方式传递 (而非 *by reference* 方式传递) 的好处是, 你可以传递常量或暂时表达式。如果不这么设计, 你就不可能传递 `IntSequence(1)` 这样的表达式。至于缺点就是, 你无法改变仿函数的状态。算法当然可以改变仿函数的状态, 但你无法存取并改变其最终状态, 因为你所改变的只不过是仿函数的副本而已。然而有时候我们的确需要存取最终状态, 因此, 问题在于如何从一个算法中获得结果。

有两个办法可以从“运用了仿函数”的算法中获取“结果”或“反馈”:

1. 以 *by reference* 的方式传递仿函数。
2. 运用 `for_each()` 算法的回返值。

第二种作法将于下一节讨论。

为了能够以 *by reference* 方式传递仿函数, 你只需要在调用算法时, 明白标示仿函数型别是个 *reference* 型别²。例如:

```
// fo/genera2.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class IntSequence {
private:
    int value;
public:
    // constructor
```

² 感谢 Philip Koster 指出这一点。

```
    IntSequence (int initialValue)
        : value(initialValue) {
    }

    // "function call"
    int operator() () {
        return value++;
    }
};

int main()
{
    list<int> coll;
    IntSequence seq(1); // integral sequence starting with 1

    // insert values from 1 to 4
    // - pass function object by reference
    // so that it will continue with 5
    generate_n<back_inserter<list<int> >,
               int, IntSequence&>(back_inserter(coll), // start
                                   4,                    // number of elements
                                   seq);                 // generates values

    PRINT_ELEMENTS(coll);

    // insert values from 42 to 45
    generate_n (back_inserter(coll), // start
               4,                    // number of elements
               IntSequence(42));     // generates values
    PRINT_ELEMENTS(coll);

    // continue with first sequence
    // - pass function object by value
    // so that it will continue with 5 again
    generate_n (back_inserter(coll), // start
               4,                    // number of elements
               seq);                 // generates values
    PRINT_ELEMENTS(coll);
```

```

// continue with first sequence again
generate_n (back_inserter(coll),      // start
            4,                        // number of elements
            seq);                     // generates values

PRINT_ELEMENTS(coll);
}

```

程序输出如下:

```

1 2 3 4
1 2 3 4 42 43 44 45
1 2 3 4 42 43 44 45 5 6 7 8
1 2 3 4 42 43 44 45 5 6 7 8 5 6 7 8

```

第一次调用 `generate_n()` 时, 仿函数 `seq` 是以 `by reference` 方式传递。只需将 `template` 参数明白加以标示, 就可以达到这个效果:

```

generate_n<back_insert_iterator<list<int> >,
           int, IntSequence&>(back_inserter(coll),      // start
                              4,                        // number of elements
                              seq);                     // generates values

```

调用之后, `seq` 内部值被改变了。第三次调用 `generate_n()` 时, 又再运用 `seq`, 这次产生的序列会接续第一次调用所产生的序列。然而这个调用系以 `by value` 方式传递 `seq`:

```

generate_n (back_inserter(coll),      // start
            4,                        // number of elements
            seq);                     // generates values

```

所以这次调用并不会改变 `seq` 的状态。因此, 最后一次调用 `generate_n()` 时, 序列又再从 5 开始。

8.1.3 `for_each()` 的回返回值

如果你所使用的算法是 `for_each()`, 那就不必大费周张地实作仿函数的“引用计数 (reference-counted) 版本”来存取其最终状态了。`for_each()` 有一个独门绝技, 其它算法概莫有之, 那就是可以返回其仿函数。这样, 你可以通过 `for_each()` 的回返回值来获取仿函数的状态了。

下面是一个使用 `for_each()` 回返回值的出色范例, 用来处理一个序列的平均值:

```

// fo/foreach3.cpp

#include <iostream>
#include <vector>
#include <algorithm>

```

```
using namespace std;

// function object to process the mean value
class MeanValue {
private:
    long num; // number of elements
    long sum; // sum of all element values
public:
    // constructor
    MeanValue () : num(0), sum(0) {
    }

    // "function call"
    // ~ process one more element of the sequence
    void operator()(int elem) {
        num++; // increment count
        sum += elem; // add value
    }

    // return mean value
    double value () {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;

    // insert elements from 1 to 8
    for (int i=1; i<=8; ++i) {
        coll.push_back(i);
    }

    // process and print mean value
    MeanValue mv = for_each (coll.begin(), coll.end(), // range
                             MeanValue());           // operation
    cout << "mean value: " << mv.value() << endl;
}
```

其中的表达式:

```
MeanValue()
```

会产生一个仿函数用来记录元素数量, 并计算所有元素的总和。将此仿函数传给 `for_each()`, 于是后者便针对容器 `coll` 内的每一个元素, 调用仿函数:

```
MeanValue mv = for_each (coll.begin(), coll.end(),
                          MeanValue());
```

返回的仿函数被赋值给 `mv`, 此后你可以调用 `mv.value()` 查询其状态。因此程序输出如下:

```
mean value: 4.5
```

你甚至可以将 `MeanValue` 变得更聪明些, 给它定义一个自动型别转换函数 `operator double()`。然后你就可以直接使用“`for_each()` 处理过”的平均值了。详见 p336 实例。

8.1.4 判断式 (Predicates) 和仿函数 (Functors)

所谓判断式, 就是返回布尔值 (可转换为 `bool`) 的一个函数或仿函数。对 STL 而言, 并非所有返回布尔值的函数都是合法的判断式。这可能会导致很多出人意表的行为。试看下列:

```
// fo/removeif.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class Nth { // function object that returns true for the nth call
private:
    int nth;      // call for which to return true
    int count;    // call counter
public:
    Nth (int n) : nth(n), count(0) {
    }

    bool operator() (int) {
        return ++count == nth;
    }
};
```

```

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    PRINT_ELEMENTS(coll, "coll: ");

    // remove third element
    list<int>::iterator pos;
    pos = remove_if(coll.begin(), coll.end(), // range
                    Nth(3)); // remove criterion
    coll.erase(pos, coll.end());
    PRINT_ELEMENTS(coll, "nth removed: ");
}

```

这个程序定义了一个仿函数 `Nth`，当它第 n 次被调用时返回 `true`。然而如果你把它传给 `remove_if()`——此算法会把所有“令单参数判断式 (unary predicate) 结果为 `true`”的元素移除，参见 p378——得到的结果会让你大吃一惊：

```

coll: 1 2 3 4 5 6 7 8 9
nth removed: 1 2 4 5 7 8 9

```

有两个（而不是一个）元素——也就是第 3 个和第 6 个元素——被移除了！为什么会这样？因为该算法的一般实作版本，会于算法内部保留判断式的一份副本：

```

template <class ForwIter, class Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                       Predicate op)
{
    beg = find_if(beg, end, op);
    if (beg == end) {
        return beg;
    }
    else {
        ForwIter next = beg;
        return remove_copy_if(++next, end, beg, op);
    }
}

```


这个算法使用 `find_if()` 来搜寻应被移除的第一个元素。然而, 接下来它使用传进来的判断式 `op` 的副本去处理剩余元素。这时原始状态下的 `Nth` 再一次被使用, 因而会移除剩余元素中的第 3 个元素, 也就是整体的第 6 个元素。

这种行为不能说是一种错误, 因为 C++ *Standard* 并未明定判断式是否可被算法复制一份副本。因此, 为了获得 C++ 标准程序库的保证行为, 你不应该传递一个“行为取决于被拷贝次数或被调用次数”的仿函数。也就是说, 如果你以两个相同参数来调用一个单参数判断式 (unary predicate), 该判断式应该总是返回相同结果。换句话说判断式不应该因为被调用而改变自身状态; 判断式的副本应该和其正本有着相同状态。要做到这一点, 你一定得保证不能因为函数调用而改变判断式的状态, 你应当将 `operator()` 声明为 `const` 成员函数。

其实我们完全可以避免这种尴尬结果, 让算法和 `Nth` 这样的仿函数一起正常运作, 而且效率不打折扣。你可以这么实作 `remove_if()`, 用另一种办法来代替 `find_if()`:

```
template <class ForwIter, class Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                       Predicate op)
{
    while (beg != end && !op(*beg)) {
        ++beg;
    }

    if (beg == end) {
        return beg;
    }
    else {
        ForwIter next = beg;
        return remove_copy_if(++next, end, beg, op);
    }
}
```

像上面这样改变 `remove_if()` 的实作法也许是一个好主意 (或是向标准程序库实作小组提交一份更改申请)。就我所知, 目前的 STL 实作品中, 只有 `remove_if()` 算法有问题。如果你换用 `remove_copy_if()`, 就一切正常³。然而如果考虑到可移植性, 你就永远不该依赖任何实作细节, 而应该总是将判断式的 `operator()` 声明为 `const` 成员函数。

³ C++标准程序库是否应该保证获得预期结果, 避免此处所举例子的情况, 目前仍有争议。

8.2 预定义的仿函数

正如 p131, 5.9.2 节所说, C++ 标准程序库提供了许多预定义的仿函数。表 8.1 列出了所有这些仿函数⁴。

表 8.1 预先定义好的仿函数

仿函数	效果
<code>negate<type>()</code>	<code>-param</code>
<code>plus<type>()</code>	<code>param1 + param2</code>
<code>minus<type>()</code>	<code>param1 - param2</code>
<code>multiplies<type>()</code> ⁴	<code>param1 * param2</code>
<code>divides<type>()</code>	<code>param1 / param2</code>
<code>modulus<type>()</code>	<code>param1 % param2</code>
<code>equal_to<type>()</code>	<code>param1 == param2</code>
<code>not_equal_to<type>()</code>	<code>param1 != param2</code>
<code>less<type>()</code>	<code>param1 < param2</code>
<code>greater<type>()</code>	<code>param1 > param2</code>
<code>less_equal<type>()</code>	<code>param1 <= param2</code>
<code>greater_equal<type>()</code>	<code>param1 >= param2</code>
<code>logical_not<type>()</code>	<code>! param</code>
<code>logical_and<type>()</code>	<code>param1 && param2</code>
<code>logical_or<type>()</code>	<code>param1 param2</code>

(译注：灰底区段是我加的，以此区分出三大类仿函数：

算术类、相对关系类、逻辑运算类)

对对象排序或进行比较时，一般都以 `less<>` 为预设准则，所以 `less<>` 经常被使用。预设的排序操作常按升幂排序 (`element < nextElement`)。

要使用这些仿函数，必须包含头文件 `<functional>` ⁵：

```
#include <functional>
```

为了对“国际化字符串”进行比较，C++ 标准程序库另提供了一个可作为字符串排序准则的仿函数，详见 p703。

⁴ 早期的 STL 版本中，乘法仿函数名为 `time`。后来之所以改名，--是因为和操作系统标准 (X/Open, POSIX) 的一个函数名称冲突，二是因为 `multiplies` 含义更清晰。

⁵ 早期的 STL 版本中，仿函数头文件是 `<function.h>`

8.2.1 函数配接器 (Function Adapters)

所谓“函数配接器”是指能够将仿函数和另一个仿函数 (或某个值, 或某个一般函数) 结合起来的仿函数。函数配接器也声明于 `<functional>` 中。

例如以下语句:

```
find_if (coll.begin(), coll.end(),           // range
        bind2nd(greater<int>(), 42))        // criterion
```

其中的表达式:

```
bind2nd(greater<int>(), 42)
```

导致一个组合型仿函数, 检查某个 `int` 值是否大于 42。实际上 `bind2nd` 是将一个二元仿函数 (例如 `greater<>`) 转换为一元仿函数。它通常将第二参数传给“由第一参数指出”的二元仿函数, 作为后者的第二参数。因此本例系以 42 作为第二参数来调用 `greater<>`。5.9.2 节, p132 提供了另一些函数配接器运用实例。

表 8.2 预定义的函数配接器 (function adapters)

表达式	效果
<code>bind1st(op, value)</code>	<code>op(value, param)</code>
<code>bind2nd(op, value)</code>	<code>op(value, param)</code>
<code>not1(op)</code>	<code>!op(param)</code>
<code>not2(op)</code>	<code>!op(param1, param2)</code>

函数配接器本身也是仿函数, 故可以结合仿函数以形成更强大 (更复杂) 的表达式, 见表 8.2。例如, 下面语句返回某群集中的第一个偶数值元素:

```
pos = find_if (coll.begin(), coll.end(),           // range
              not1(bind2nd(modulus<int>(), 2)));    // criterion
```

本语句中, 对于所有奇数值, 以下表达式:

```
bind2nd(modulus<int>(), 2)
```

返回 1。所以这个表达式即是一个用来“找出第一个奇数值元素”的准则, 因为 1 相当于 `true`。`not1()` 会将结果反相, 于是整个语句找出第一个偶数值元素。

通过函数配接器, 我们可以把多个仿函数结合起来, 形成强大的表达式, 这种编程方式称为 `functional composition` (功能复合、函数复合)。但是 C++ 标准程序库缺乏某些必要的函数配接器, 因此 (例如) 难以让你将两个判断式以 `and` 或 `or` 组合起来 (像是“大于 4” and “小于 7”)。如果你能够以某些复合用的函数配接器将标准函数配接器更加扩充, 你便可以获得极大的威力。8.3 节, p313 对此有些讨论。

8.2.2 针对成员函数而设计的函数配接器

译注：本节所谈的函数配接器，其实作手法十分令人难以想象（如果你从来没有见识过那种技法的话）。欲窥知其实作方法，可参考《STL 源码剖析》8.4 节）

C++ 标准程序库提供了一些额外的函数配接器，透过它们，你可以针对群集内的每个元素调用其成员函数（表 8.3）。

表 8.3 成员函数（member functions）的函数配接器

表达式	效果
mem_fun_ref(op)	调用 op，那是某对象的一个 const 成员函数
mem_fun(op)	调用 op，那是某对象指针的一个 const 成员函数

例如，在下面的程序中，我利用 mem_fun_ref，对 vector 内的每一个对象，调用其成员函数：

```
// fo/memfun1a.cpp

class Person {
private:
    std::string name;
public:
    ...
    void print () const {
        std::cout << name << std::endl;
    }
    void printWithPrefix (std::string prefix) const {
        std::cout << prefix << name << std::endl;
    }
};

void foo (const std::vector<Person>& coll)
{
    using std::for_each;
    using std::bind2nd;
    using std::mem_fun_ref;

    // call member function print() for each element
    for_each (coll.begin(), coll.end(),
              mem_fun_ref(&Person::print));
}
```

```
// call member function printWithPrefix() for each element
// - "person: " is passed as an argument to the member function
for_each(coll.begin(), coll.end(),
         bind2nd(mem_fun_ref(&Person::printWithPrefix),
                 "person: "));
}
```

foo()里头针对 vector coll 的每一个元素, 调用 Person 的两个不同的成员函数:

(1) Person::print(), 不带参数 (2) Person::printWithPrefix(), 有一个参数。

为了得以调用成员函数 Person::print(), 我们必须将仿函数:

```
mem_fun_ref(&Person::print)
```

传给算法 for_each():

```
for_each(coll.begin(), coll.end(),
         mem_fun_ref(&Person::print));
```

于是配接器 mem_fun_ref 会将原本“针对某个元素的函数调用动作”转为调用“被传递之成员函数”。

由于我们不可能直接把一个成员函数传给一个算法, 所以这里必须运用配接器。下面这种作法可能会导致编译错误:

```
for_each(coll.begin(), coll.end(),
         &Person::print); // ERROR: can't call operator ()
                          // for a member function pointer
```

问题在于 for_each() 会针对第三参数所传进来的指针, 调用 operator(), 而不是调用该指针所指的成员函数。配接器 mem_fun_ref 将 operator() 调用动作做了适当转换, 因而解决了这个问题。

透过 bind2nd, 我们还可以向被调用的成员函数传递一个参数。上例第二次调用 for_each() 时便展示了这一点⁶:

```
for_each(coll.begin(), coll.end(),
         bind2nd(mem_fun_ref(&Person::printWithPrefix),
                 "person: "));
```

你可能会疑惑, 为什么这个配接器名为 mem_fun_ref 而不是 mem_fun。这有一段历史因素: 一开始的确曾经有过一个名为 mem_fun 的成员函数配接器, 其作用对象是指针序列 (每个指针指向实际元素)。其实如果命名为 mem_fun_ptr 或许更不易混淆。所以如果你有一个指针序列 (每个指针指向一个对象), 你也可以调

⁶ 在早期的 STL 和 C++ 标准程序库中, 单一参数的成员函数配接器被命名为 mem_fun1 和 mem_fun1_ref, 不是 mem_fun 和 mem_fun_ref。

用它们的成员函数。例如：

```
// fo/memfun1b.cpp
void ptrfoo (const std::vector<Person*>& coll)
{
    using std::for_each;
    using std::bind2nd;
    using std::mem_fun;

    // call member function print() for each referred object
    for_each (coll.begin(), coll.end(),
              mem_fun(&Person::print));

    // call member function printWithPrefix() for each referred object
    // "person: " is passed as an argument to the member function
    for_each (coll.begin(), coll.end(),
              bind2nd(mem_fun(&Person::printWithPrefix),
                      "person: "));
}
```

`mem_fun_ref` 和 `mem_fun` 两者都能以无参数或单参数方式来调用成员函数。然而你无法以两个或更多参数来调用成员函数。这是因为为了实现这些配接器，你必须不厌其烦地为每一类成员函数提供大量辅助用的仿函数。例如单单为了支持 `mem_fun_ref` 和 `mem_fun`，就需要一大堆辅助类别，包括 `mem_fun_t`，`mem_fun_ref_t`，`const_mem_fun_t`，`const_mem_fun_ref_t`，`mem_fun1_t`，`mem_fun1_ref_t`，`const_mem_fun1_t`，`const_mem_fun1_ref_t`。

注意，被 `mem_fun_ref` 和 `mem_fun` 调用的成员函数必须是 `const`。遗憾的是 C++ 标准程序库并不针对 `non-const` 成员函数提供函数配接器（撰写本书时我才发现这一点）。这似乎是个小小疏漏，因为几乎所有人都知道可能出现这种情况。希望往后新的实作版本（以及新的标准规格）能解决这个问题。

8.2.3 针对一般函数（非成员函数）而设计的函数配接器

另一个函数配接器 `ptr_fun`（表 8.4），允许在其它函数配接器中使用一般函数：

例如，假设有一个能对每个参数实施某种检验的全局函数如下：

```
bool check(int elem);
```

表 8.4 用于一般函数身上的函数配接器

表达式	效果
<code>ptr_fun(op)</code>	<code>op(param)</code>
	<code>op(param1, param2)</code>

(译注: 以上两项, 在英文版皆写为 `*op(...)`。但其实 `op` 为一般函数指针, 因此 `op(...)` 即表示调用该函数, 不应写为 `*op(...)`, 故改之)。

如果要搜寻第一个令检验动作失败的元素, 可以使用下面的语句:

```
pos = find_if (coll.begin(), coll.end(), // range
              not1(ptr_fun(check)));    // search criterion
```

这里不能使用 `not1(check)`, 因为 `not1()` 需要用到由仿函数提供的某些特殊型别, 详见 8.2.4 节。

第二种用法是, 当你有一个双参数全局函数, 又想把它当成一个单参数函数来使用, 可以用如下语句:

```
// find first string that is not empty
pos = find_if (coll.begin(), coll.end(), // range
              bind2nd(ptr_fun(strcmp), "")); // search criterion
```

这里采用函数 `strcmp()` 将每个元素与空的 C-string 比较。当字符串相等, `strcmp()` 返回 0, 亦即 `false`。因此上述 `find_if()` 会返回非空字符串的第一个元素位置。p319 另有一个 `ptr_fun` 运用实例。

8.2.4 让自定仿函数也可以使用函数配接器

你可以编写自己的仿函数, 但如果希望它们能够和函数配接器搭配运用, 就必须满足某些条件: 必须提供一些型别成员 (type members) 来反映其参数和返回值的型别。为了方便程序员, C++ 标准程序库提供了一些结构如下:

```
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;
};
```

如此一来，仿函数只要继承上述三种形式之一，就能轻松满足“可配接”(adaptable)的条件。

下面的例子定义了一个完整的仿函数，该例以第二参数做为指数(power)，计算第一个参数的幂：

```
// fo/fopow.hpp

#include <functional>
#include <cmath>

template <class T1, class T2>
struct fopow : public std::binary_function<T1, T2, T1>
{
    T1 operator() (T1 base, T2 exp) const {
        return std::pow(base, exp);
    }
};
```

在这里，第一参数和返回值的型别都是 T1，指数型别有可能不同，所以是 T2。这些型别都被传递到 binary_function 以满足后者所需的型别定义。不过你也可以直接进行型别定义，不必把它们传给 binary_function。是的，和 STL 的习惯一致，函数配接器的概念也是完全抽象的；任何东西的行为只要像函数配接器，它就是一个函数配接器。

以下程序示范如何运用上述那个使用者自定义的仿函数 fopow。尤其请注意的是，在这个例子中，fopow 结合了 bind1st 和 bind2nd 一起使用。

```
// fo/fopow1.cpp

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

//include self-defined fopow<>
#include "fopow.hpp"

int main()
{
    vector<int> coll;

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



```
// print 3 raised to the power of all elements
transform (coll.begin(), coll.end(),           // source
           ostream_iterator<int>(cout, " "), // destination
           bind1st(fopow<float,int>(),3));     // operation
cout << endl;

//print all elements raised to the power of 3
transform (coll.begin(), coll.end(),           // source
           ostream_iterator<int>(cout, " "), // destination
           bind2nd(fopow<float,int>(),3));     // operation
cout << endl;
}
```

程序输出如下:

```
3 9 27 81 243 729 2187 6561 19683
1 8 27 64 125 216 343 512 729
```

注意此处 `fopow` 乃针对 `float` 和 `int` 两种型别实现完成。如果基数和指数均为 `int` 型别, 按说应采用两个 `int` 参数来调用 `pow()`, 但是根据标准规格, `pow()` 虽然有数种重载 (**overloaded**) 形式, 却并未顾及所有基本型别, 所以不具有很好的移植性。

```
transform (coll.begin(), coll.end(),
           ostream_iterator<int>(cout, " "),
           bind1st(fopow<int,int>(),3));      // ERROR: ambiguous
```

关于这个问题, 详见 p581。

8.3 辅助用（组合型）仿函数

仿函数的组合能力非常重要，藉此我们可以从一些软件组件（components）构造出另一些组件。你可以用简单的仿函数构造出非常复杂的仿函数。一般而言，所有函数行为都可以藉由仿函数的组合而实现。然而 C++ 标准程序库并未提供足够的配接器来达到这个境界。例如我们无法将两个一元运算 A、B 制定成“A and B”这样的公式。

理论上，下面这些组合型配接器（compose adapters）很有用：

- `f(g(elem))`

这是一元组合函数（unary compose function）的一般形式。一元判断式被嵌套调用，`g()` 执行结果被当做 `f()` 参数。整个表达式的操作类似一个一元判断式。

- `f(g(elem1,elem2))`

这个形式中，两元素 `elem1` 和 `elem2` 作为参数传给二元判断式 `g()`，其结果再作为参数传给一元判断式 `f()`。整个表达式的操作类似一个二元判断式。

- `f(g(elem),h(elem))`

此处 `elem` 作为参数被传递给两个不同的一元判断式 `g()` 和 `h()`，两者的结果由二元判断式 `f()` 处理。这种形式系以某种方法将单一参数“注射（injects）”到一个组合函数中，整个表达式的操作类似一个一元判断式。

- `f(g(elem1),h(elem2))`

此处 `elem1` 和 `elem2` 分别作为唯一参数传给两个不同的一元判断式 `g()` 和 `h()`，两个结果共同被二元判断式 `f()` 处理。某种程度上这种形式系在两个参数身上分布（distributes）一个组合函数。整个表达式的操作类似一个二元判断式。

遗憾的是，这些组合型配接器并未被纳入标准规范中，所以我们没有它们的标准名称。SGI STL 实作版本中定义了其中两个名称（译注：实作细节可参考《STL 源码剖析》8.4.3 节），目前 C++ 社群正在寻找上述所有配接器的通用名称。本书选用了一些名称，如表 8.5。

表 8.5 组合型函数配接器（compose function adapters）的可能名称

功能（functionality）	本书采用名称	SGI STL 采用名称
<code>f(g(elem))</code>	<code>compose_f_gx</code>	<code>compose1</code>
<code>f(g(elem1,elem2))</code>	<code>compose_f_gxy</code>	
<code>f(g(elem),h(elem))</code>	<code>compose_f_gx_hx</code>	<code>Compose2</code>
<code>f(g(elem1),h(elem2))</code>	<code>compose_f_gx_hy</code>	

请访问 <http://www.boost.org/>，那儿有个 C++ 程序库，其中的命名也许会被用于未来的标准规格。该处也提供了上述所有配接器的实作版本。以下数小节，我会讨论最常用的三个组合型函数配接器。

8.3.1 一元组合函数配接器

Unary Compose Function Object Adapters

本节讨论最基本的组合型函数配接器，它们是 SGI STL 实作版本的一部分。

以 `compose_f_gx` 进行嵌套 (nested) 计算

最简单也最基本的组合型函数配接器，是将某个一元运算结果作为另一个一元运算的输入。其实这只不过是嵌套调用两个一元仿函数。当你企图构造诸如 “先加 10 再乘以 4” 这类运算时，就会用到这个函数配接器。

我把这种函数配接器命名为 `compose_f_gx`，SGI STL 实作版本称之为 `compose1`。`compose_f_gx` 可实作如下：

```
// fo/compose11.hpp

#include <functional>

/* class for the compose_f_gx adapter
*/

template <class OP1, class OP2>
class compose_f_gx_t
    : public std::unary_function<typename OP2::argument_type,
                                typename OP1::result_type>
{
private:
    OP1 op1; // process: op1(op2(x))
    OP2 op2;
public:
    // constructor
    compose_f_gx_t(const OP1& o1, const OP2& o2)
        : op1(o1), op2(o2) {
    }

    // function call
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x) const {
        return op1(op2(x));
    }
};
```

```
/* convenience functions for the compose_f_gx adapter
*/
template <class OP1, class OP2>
inline compose_f_gx_t<OP1,OP2>
compose_f_gx (const OP1& o1, const OP2& o2) {
    return compose_f_gx_t<OP1,OP2>(o1,o2);
}
```

下面这个例子展示了 `compose_f_gx` 的用法：

```
// fo/compose1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iterator>
#include "print.hpp"
#include "compose11.hpp"
using namespace std;

int main()
{
    vector<int> coll;

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

    // for each element add 10 and multiply by 5
    transform (coll.begin(),coll.end(),
               ostream_iterator<int>(cout," "),
               compose_f_gx(bind2nd(multiplies<int>(),5),
                           bind2nd(plus<int>(),10)));

    cout << endl;
}
```

注意，传给 `compose_f_gx` 的第二个运算会先执行，所以

```
compose_f_gx(bind2nd(multiplies<int>(),5),
             bind2nd(plus<int>(),10))
```

会产生一个一元仿函数, 先将元素加上 10, 再乘以 5。程序输出如下:

```
1 2 3 4 5 6 7 8 9
55 60 65 70 75 80 85 90 95
```

以 `compose_f_gx_hx` 组合两个“运作准则”

这也许是最重要的一个组合型函数配接器, 它允许你将两个准则加以逻辑组合, 形成单一准则。当你需要诸如“大于 4 且小于 7”这样的准则时, 你就需要这个配接器。

我把这个组合型函数配接器命名为 `compose_f_gx_hx`, SGI STL 实作版本称之为 `compose2`。 `compose_f_gx_hx` 可以实作如下:

```
// fo/compose21.hpp

#include <functional>

/* class for the compose_f_gx_hx adapter
*/

template <class OP1, class OP2, class OP3>
class compose_f_gx_hx_t
    : public std::unary_function<typename OP2::argument_type,
                                typename OP1::result_type>
{
private:
    OP1 op1; // process: op1(op2(x), op3(x))
    OP2 op2;
    OP3 op3;

public:
    // constructor
    compose_f_gx_hx_t (const OP1& o1, const OP2& o2, const OP3& o3)
        : op1(o1), op2(o2), op3(o3) {
    }

    // function call
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x) const {
        return op1(op2(x), op3(x));
    }
};
```

```

/* convenience functions for the compose_f_gx_hx adapter
*/

template <class OP1, class OP2, class OP3>
inline compose_f_gx_hx_t<OP1,OP2,OP3>
compose_f_gx_hx (const OP1& o1, const OP2& o2, const OP3& o3) {
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}

```

compose_f_gx_hx 将“两个一元表达式对同一对象的运算结果”再以一个表达式加以组合，所以下面这个表达式：

```
compose_f_gx_hx(op1,op2,op3)
```

实际相当于一个一元判断式（unary predicate），并对 x 调用以下表达式：

```
op1(op2(x),op3(x))
```

下面是个完整实例：

```

// fo/compose2.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include "print.hpp"
#include "compose21.hpp"
using namespace std;

int main()
{
    vector<int> coll;

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

    // remove all elements that are greater than four and less than
    seven
    // - retain new end
    vector<int>::iterator pos;
    pos = remove_if (coll.begin(),coll.end(),
        compose_f_gx_hx(logical_and<bool>(),
            bind2nd(greater<int>(),4),
            bind2nd(less<int>(),7)));
}

```

```

    // remove "removed" elements in coll
    coll.erase(pos, coll.end());
    PRINT_ELEMENTS(coll);
}

```

其中的表达式:

```

compose_f_gx_hx(logical_and<bool>(),
                bind2nd(greater<int>(), 4),
                bind2nd(less<int>(), 7))

```

构成了一个一元判断式, 判断某值是否大于4且小于7。程序输出如下:

```

1 2 3 4 5 6 7 8 9
1 2 3 4 7 8 9

```

8.3.2 二元组合函数配接器

Binary Compose Function Object Adapters

二元组合函数配接器之一, 可以将两个一元运算 (分别接受不同参数) 的结果加以处理。我称此为 `compose_f_gx_hy`。下面是一种可能的实作手法:

```

// fo/compose22.hpp

#include <functional>

/* class for the compose_f_gx_hy adapter
*/
template <class OP1, class OP2, class OP3>
class compose_f_gx_hy_t
: public std::binary_function<typename OP2::argument_type,
                              typename OP3::argument_type,
                              typename OP1::result_type>
{
private:
    OP1 op1; // process: op1(op2(x), op3(y))
    OP2 op2;
    OP3 op3;
}

```

```

public:
    // constructor
    compose_f_gx_hy_t (const OP1& o1, const OP2& o2, const OP3& o3)
        : op1(o1), op2(o2), op3(o3) {
    }

    // function call
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x,
               const typename OP3::argument_type& y) const {
        return op1(op2(x), op3(y));
    }
};

/* convenience function for the compose_f_gx_hy adapter
*/
template <class OP1, class OP2, class OP3>
inline compose_f_gx_hy_t<OP1,OP2,OP3>
compose_f_gx_hy (const OP1& o1, const OP2& o2, const OP3& o3) {
    return compose_f_gx_hy_t<OP1,OP2,OP3>(o1,o2,o3);
}

```

下面这个例子展示 `compose_f_gx_hy` 的用法，在一个字符串中以“区分大小写”的方式搜寻一个子字符串：

```

// fo/compose3.cpp

#include <cctype> // for toupper()
#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <cctype>
#include "compose22.hpp"
using namespace std;

int main()
{
    string s("Internationalization");
    string sub("Nation");

```



```
// search substring case insensitive
string::iterator pos;
pos = search (s.begin(),s.end(),          // string to search in
              sub.begin(),sub.end(),      // substring to search
              compose_f_gx_hy(equal_to<int>(), // compar. criterion
                              ptr_fun(::toupper),
                              ptr_fun(::toupper)));

if (pos != s.end()) {
    cout << "\"" << sub << "\" is part of \"" << s << "\""
    << endl;
}
}
```

程序输出如下:

```
"Nation" is part of "Internationalization"
```

你可以在 p499 找到一个“大小写无关”的子字符串搜寻实例, 那儿并未使用 `compose_f_gx_hy`。

9

STL 算法

STL Algorithms

本章描述 C++ 标准程序库提供的所有算法。首先对所有算法及其主要特征进行概述，然后展示每一种算法的确切形式，并运用一个或多个例子说明其用法。

9.1 算法头文件 (header files)

要运用C++ 标准程序库的算法，首先必须含入头文件 `<algorithm>`¹：

```
#include <algorithm>
```

此文件也包含了一些辅助函数，其中 `min()`, `max()`, `swap()` 在 4.4.1 节(p66)和 4.4.2 节(p67)已有详述，迭代器相关函数 `iter_swap()` 在 7.3.3 节(p263)已有详述。

某些 STL 算法用于数值处理，因此被定义于头文件 `<numeric>`¹：

```
#include <numeric>
```

C++ 标准程序库的数值相关组件在本书第 12 章讨论。但我决定在这里先讨论数值算法。因为就我的观点而言，它们被视为 STL 算法，远比被用来处理数值更重要。

使用 STL 算法时，你经常需要用到仿函数 (functor，或称为 function objects) 及函数配接器 (function adapters)。它们都被定义于 `<functional>`²之中，本书第 8 章对仿函数和函数配接器有详细的讲解。

```
#include <functional>
```

¹ 早期 STL 中，所有算法都定义于 `<algo.h>`。

² 早期 STL 中，仿函数和函数配接器定义于 `<function.h>`。

9.2 算法概览

本节概述 C++ 标准程序库提供的所有算法。你可以从中体会它们的能力，进而在遇到某些问题时选择最适用的算法。

9.2.1 简介

在第 5 章中，借着整体讲解 STL 的大背景，我曾经介绍过算法。特别是 5.4 节 (p94) 和 5.6 节 (p111) 讨论了算法的角色和使用上的一些重要限制。所有 STL 算法都被设计用来处理一个或多个迭代器区间。第一个区间通常以起点和终点表示，至于其它区间，多数情况下你只需提供起点便足矣，其终点可自动以第一区间的元素数量推导出来。调用者必须确保这些区间的有效性，也就是说起点和终点必须指向同一个容器，而且起点位置不得在终点之后，第二（及其它）区间必须有足够的空间。

STL 算法采用覆盖 (overwrite) 模式而非安插 (insert) 模式。所以调用者必须保证目标区间拥有足够的元素空间。当然，你也可以运用特殊的安插型迭代器（参见 7.4.2 节，p271）将覆盖模式改变为安插模式。

为了提高灵活性和功效，某些 STL 算法允许使用者传递自定的操作，以便由 STL 算法调用之。这些操作既可以是一般函数，也可以是仿函数；如果其返回值是 bool，便称为条件判断式 (predicates)。你可以运用条件判断式完成以下工作：

- 对于搜寻算法，你可以传递一个函数或仿函数，指定一个一元判断式作为搜寻准则。该一元判断式用来判断某元素是否符合条件。例如你可以搜寻第一个“小于 50”的元素。
- 对于排序算法，你可以传递一个函数或仿函数，指定一个二元判断式作为排序准则。该二元判断式用来比较两元素。例如你可以传递一个准则，让 Person 对象按姓氏排序（参见 p294 实例）。
- 你可以传递一个一元判断式作为准则，判断是否应该对某元素施以某项运算。例如你可以令奇数值元素被移除。
- 你可以为某个数值算法指定一个数值运算。例如你可以让通常用来求“总和 (sum) 的 accumulate() 算法改为求取乘积 (product)。

注意，判断式不应该在函数调用过程中改变其自身状态（参见 8.1.4, p302）。

关于算法所用的函数和仿函数，请参考 5.8 节 (p119)，5.9 节 (p124)，以及第 8 章。

9.2.2 算法分门别类

不同的算法满足不同的需求。所以，我们可以根据它们的主要目的加以分类。例如某些算法的操作是只读性的，某些算法会改动元素本身，某些则改动元素顺序。本小节简单介绍每种算法的功能，并指出相似算法之间的区别。

为了让人顾名思义，STL 设计者为算法命名时，引入两个特别的尾词：

1. 尾词 `_if`

如果算法有两种形式，参数个数都相同，但第一形式的参数要求传递一个值，第二形式的参数要求传递一个函数或仿函数，那么尾词 `_if` 就派上了用场。无尾词的那个要求传递数值，有尾词的那个要求传递函数或仿函数。例如，`find()` 用来搜寻具有某值的元素，而 `find_if()` 接受一个被当做搜寻准则的函数或仿函数，并搜寻第一个满足该准则的元素。

不过并非所有“要求传递仿函数”的算法都有尾词 `_if`。如果算法以额外参数来接受这样的函数或仿函数，那么不同版本的算法就可以采用相同命名（译注：重载，overloaded）。例如当你以两个参数调用 `min_element()`，该算法以 `operator<` 为比较准则，返回区间中的最小元素，但如果你传递第三参数，这个参数会被当做比较准则。

2. 尾词 `_copy`

这个尾词用来表示在此算法中，元素不光被操作，还会被复制到目标区间。例如 `reverse()` 将区间中的元素颠倒次序，而 `reverse_copy()` 则是逆序将元素复制到另一个区间。

接下来数小节按以下分类方式描述各个 STL 算法：（译注：请注意，不同的书籍对于以下的 `modifying` 和 `mutating` 两字意义和用法可能不尽相同）

- 非变动性算法（nonmodifying algorithms）
- 变动性算法（modifying algorithms）
- 移除性算法（removing algorithms）
- 变序性算法（mutating algorithms）
- 排序算法（sorting algorithms）
- 已序区间算法（sorted range algorithms）
- 数值算法（numeric algorithms）

如果某个算法同时隶属多个分类，我会把它放在我认为最贴切的分类中讲述。

非变动性算法（nonmodifying algorithms）

非变动性算法既不动元素次序，也不改动元素值。它们透过 `input` 迭代器和 `forward` 迭代器完成工作，因此可作用于所有标准容器身上。表 9.1 展示 C++ 标准程序库涵盖的所有非变动性算法。还有一些非变动性算法专门用来操作已序（sorted）输入区间，参见 p330。

表 9.1 非变动性算法

名称	作用	页次
<code>for_each()</code>	对每个元素执行某操作	334
<code>count()</code>	返回元素个数	338
<code>count_if()</code>	返回满足某一准则(条件)的元素个数	338
<code>min_element()</code>	返回最小值元素(译注:以一个迭代器表示)	340
<code>max_element()</code>	返回最大值元素(译注:以一个迭代器表示)	340
<code>find()</code>	搜寻等于某值的第一个元素	341
<code>find_if()</code>	搜寻满足某个准则的第一个元素	341
<code>search_n()</code>	搜寻具有某特性的第一段“n个连续元素”	344
<code>search()</code>	搜寻某个子区间第一次出现位置	347
<code>find_end()</code>	搜寻某个子区间最后一次出现位置	347
<code>find_first_of()</code>	搜寻等于“某数个值之一”的第一个元素	352
<code>adjacent_find()</code>	搜寻连续两个相等(或说符合特定准则)的元素	354
<code>equal()</code>	判断两区间是否相等	356
<code>mismatch()</code>	返回两个序列的各组对应元素中,第一对不相等元素	358
<code>lexicographical_compare()</code>	判断某一序列在“字典顺序”(lexicographically)下是否小于另一序列	360

最重要的算法之一便是 `for_each()`。它将调用者提供的操作施加于每一个元素身上。例如你可以用 `for_each()` 来打印区间内的每个元素。`for_each()` 也可以用来变动元素(如果你传给它的操作行为会变动元素值的话)。所以它既可说是非变动性算法,也可以说是变动性算法。不过在此情况下,如果有其它算法可以满足你的需求,你最好避免使用 `for_each()`, 毕竟其它那些算法是为特别任务而量身定做的。

有好几个非变动性算法具有搜寻能力。不幸的是,搜寻算法的命名方式却是一团混乱。此外,搜寻算法的命名方式又和 `string` 搜寻函数的命名方式大相径庭(表 9.2)。为什么会这样?还不是老掉牙的“历史因素”所致。首先, `STL classes` 和 `string classes` 乃是各自独立发展设计的。其次 `find_end()`, `find_first_of()`, `search_n()` 算法并不涵盖于早期的 STL。所以最后选择的名称是 `find_end()` 而不是 `search_end()`, 这完全不是刻意的(说实在话,一旦你钻入细节,像“一致性”这种大局观问题就很容易被忽略)。同样是偶发事件, `search_n()` 的某一形式竟然违背了原始 STL 的一般性概念。这一问题的描述请见 p346。

表 9.2 String 搜寻函数和 STL 搜寻算法的比较

搜寻	String 函数	STL 算法
某元素第一次出现位置	find()	find()
某元素最后一次出现位置	rfind()	find(), 采用逆向迭代器
某子区间第一次出现位置	find()	search()
某子区间最后一次出现位置	rfind()	find_end()
某数个元素第一次出现位置	find_first_of()	find_first_of()
某数个元素最后一次出现位置	find_last_of()	find_first_of(), 采用逆向迭代器
n 个连续元素第一次出现位置		search_n()

变动性算法 (modifying algorithms)

变动性算法，要不直接改变元素值，要不就是在复制到另一区间的过程中改变元素值。如果是第二种情况，原区间不会发生变化。表 9.3 列出了 C++ 标准程序库涵括的变动性算法。

最基本的变动性算法是 `for_each()`（唔，又是它！）和 `transform()`。两者都可以变动序列中的所有元素值。它们的行为有以下不同点：

- `for_each()` 接受一项操作，该操作可变动其参数。因此该参数必须以 `by reference` 方式传递。例如：

```
void square (int& elem) // call-by-reference
{
    elem = elem * elem; // assign processed value directly
}
...
for_each(coll.begin(), coll.end(), // range
        square);                  // operation
```

- `transform()` 运用某项操作，该操作返回被改动后的参数。此间奥妙在于它可以被用来将结果赋值给原元素。例如：

```
int square (int elem) // call-by-value
{
    return elem * elem; // return processed value
}
...
transform (coll.begin(), coll.end(), // source range
          coll.begin(),             // destination range
          square);                  // operation
```

表 9.3 变动性算法 (modifying algorithms)

名称	效果	页次
for_each()	针对每个元素执行某项操作	334
copy()	从第一个元素开始, 复制某段区间	363
copy_backward()	从最后一个元素开始, 复制某段区间	363
transform()	变动 (并复制) 元素, 将两个区间的元素合并	367
merge()	合并两个区间	416
swap_ranges()	交换两区间内的元素	370
fill()	以给定值替换每一个元素	372
fill_n()	以给定值替换 n 个元素	372
generate()	以某项操作的结果替换每一个元素	373
generate_n()	以某项操作的结果替换 n 个元素	373
replace()	将具有某特定值的元素替换为另一个值	375
replace_if()	将符合某准则的元素替换为另一个值	375
replace_copy()	复制整个区间, 同时并将具有某特定值的元素替换为另一个值	376
replace_copy_if()	复制整个区间, 同时并将符合某准则的元素替换为另一个值	376

`transform()` 的速度稍许慢些, 因为它是将操作返回值赋值给元素, 而不是直接变动元素。不过其灵活性比较高, 因为它可以把某个序列复制到目标序列中, 同时变动元素内容。`transform()` 的第二形式可以将两个源区间的元素的组合结果放到目标区间。

严格地说, `merge()` 不算是变动性算法的当然一员。因为它要求输入区间必须已序 (*sorted*), 所以应该归为“作用于已序区间之算法” (详见 p330)。然而实用上 `merge()` 也可用来合并无序区间——当然其结果也是无序的。不过基于安全考量, 你最好只对已序区间调用 `merge()`。

注意, 关联式容器的元素被视为常数, 惟其如此, 你才不会在变动元素的时候有任何可能违反整个容器的排序准则。因此, 你不可以将关联式容器当做变动性算法的目标区间。

除了这些变动性算法, C++ 标准程序库还提供了不少专门用来处理已序 (*sorted*) 区间的算法。细节详见 p330。

移除性算法 (Removing Algorithms)

移除性算法是一种特殊的变动性算法。它们可以移除某区间内的元素, 也可以在复制过程中执行移除动作。和变动性算法类似, 移除性算法的目标区间也不能是个关联式容器。表 9.4 列出 C++ 标准程序库涵括的所有移除性算法:

表 9.4 移除性算法 (Removing Algorithms)

名称	效果	页次
<code>remove()</code>	将等于某特定值的元素全部移除	378
<code>remove_if()</code>	将满足某准则的元素全部移除	378
<code>remove_copy()</code>	将不等于某特定值的元素全部复制到它处	380
<code>remove_copy_if()</code>	将不满足某准则的元素全部复制到它处	380
<code>unique()</code>	移除毗邻的重复元素	381
<code>unique_copy()</code>	移除毗邻的重复元素，并复制到它处	384

注意，移除算法只是在逻辑上移除元素，手段是：将不需被移除的元素往前覆盖 (overwrite) 应被移除的元素。因此它并不改变操作区间内的元素个数，而是返回逻辑上的新终点位置。至于是否使用这个位置进行诸如“实际移除元素”之类的操作，那是调用者的事情。这个问题的详细讨论请见 5.6.1 节, p111。

变序性算法 (Mutating Algorithms)

所谓变序性算法是，透过元素值的赋值和交换，改变元素顺序 (但不改变元素值)。表 9.5 列出 C++ 标准程序库涵盖的所有变序性算法。和变动性算法 (modifying algorithms) 一样，变序性算法也不能以关联式容器作为目标，因为关联式容器的元素都被视为常数，不可更改。

表 9.5 变动性算法 (Mutating Algorithms)

名称	效果	页次
<code>reverse()</code>	将元素的次序逆转	386
<code>reverse_copy()</code>	复制的同时，逆转元素顺序	386
<code>rotate()</code>	旋转元素次序	388
<code>rotate_copy()</code>	复制的同时，旋转元素次序	389
<code>next_permutation()</code>	得到元素的下一个排列次序	391
<code>prev_permutation()</code>	得到元素的上一个排列次序	391
<code>random_shuffle()</code>	将元素的次序随机打乱	393
<code>partition()</code>	改变元素次序，使“符合某准则”者移到前面	395
<code>stable_partition()</code>	与 <code>partition()</code> 相似，但保持符合准则与不符合准则之各个元素之间的相对位置	395

排序算法 (Sorting Algorithms)

排序算法是一种特殊的变序性算法，但比一般的变序性算法复杂，花费更多时间。事实上排序算法的复杂度通常低于线性算法的³，而且需要动用随机存取迭代器。表 9.6 列出所有的排序算法。

表 9.6 排序算法 (Sorting Algorithms)

名称	效果	页次
sort()	对所有元素排序	397
stable_sort()	对所有元素排序，并保持相等元素间的相对次序	397
partial_sort()	排序，直到前 n 个元素就位	400
partial_sort_copy()	排序，直到前 n 个元素就位，结果复制于它处	402
nth_element()	根据第 n 个位置进行排序	404
partition()	改变元素次序，使符合某准则的元素放在前面	395
stable_partition()	与 partition() 相同，但保持符合准则和不符合准则的各个元素之间的相对位置	395
make_heap()	将一个区间转换成一个 heap	406
push_heap()	将元素加入一个 heap	406
pop_heap()	从 heap 移除一个元素	407
sort_heap()	对 heap 进行排序 (执行后就不再是个 heap 了)	407

对排序算法而言，时间往往很重要。所以 C++ 标准程序库提供了多个排序算法。这些算法使用不同的排序方式，有些算法并不对所有元素进行排序。例如 nth_element() 在第 n 个元素就位之后即停止排序，对其它元素而言，它只保证凡是小于“已就位之第 n 个元素”的所有元素，都排在前面，大的元素则排在 n 位置之后。如果要对所有元素进行排序，可以考虑以下算法：

- **sort()** 内部采用 quicksort 算法。因此保证了很好的平均性能，复杂度为 $n \cdot \log(n)$ ，但最差情况下也可能具有非常差的性能（二次复杂度）。

```
/* sort all elements
 * - best  $n \cdot \log(n)$  complexity on average
 * -  $n^2$  complexity in worst case
 */
sort (coll.begin(), coll.end());
```

³ 关于复杂度的讨论，请见 2.3 节, p21。

所以如果“避免最差情况”对你是一件很重要的事，你应该采用其它算法，例如接下来要讨论的 `partial_sort()` 或 `stable_sort()`。

- **`partial_sort()`** 内部采用 **heapsort** 算法。因此，它在任何情况下保证 $n \cdot \log(n)$ 复杂度。大多数情况下 **heapsort** 比 **quicksort** 慢 2~5 倍，所以大多数时候虽然 `partial_sort()` 具有较佳复杂度，但 `sort()` 具有较好的执行效率。`partial_sort()` 的优点是它在任何时候都保证 $n \cdot \log(n)$ 复杂度，绝不会变成二次复杂度。

`partial_sort()` 还有一种特殊能力：如果你只需要前 n 个元素排序，它可以在完成任务后立刻停止。所以如果想对所有元素进行排序，你可以将序列的终点作为第二参数和最后一个参数传进去：

```
/* sort all elements
 * - always  $n \cdot \log(n)$  complexity
 * - but usually twice as long as sort()
 */
partial_sort (coll.begin(), coll.end(), coll.end());
```

- **`stable_sort()`** 内部采用 **mergesort**。它对所有元素进行排序：

```
/* sort all elements
 * -  $n \cdot \log(n)$  or  $n \cdot \log(n) \cdot \log(n)$  complexity
 */
stable_sort (coll.begin(), coll.end());
```

然而只有在具备足够内存时，它才具有 $n \cdot \log(n)$ 复杂度。否则其复杂度为 $n \cdot \log(n) \cdot \log(n)$ 。`stable_sort()` 的优点是会保持相等元素之间的相对次序。

现在你对于哪种算法最合乎你的需求，应该有了一个大致印象。但是事情还没完。标准规格书中虽然规范了复杂度，却没有规范实作法。这是有好处的，任何实作版本可以在不违反标准的情况下采用新发明的、更好的排序法。例如 SGI 实作版本中的 `sort()` 内部就是采用 **introsort**，这是一种新式算法，缺省状态下类似 **quicksort**，一旦情况转坏即将走入二次复杂度时，就转而采用 **heapsort**。标准规格中未能明定复杂度，也有一个缺点，那就是某些实作版本可能会采用合乎标准但表现很糟的算法。例如以 **heapsort** 来实现 `sort()` 就符合标准。当然你也可以测试看看哪个算法最合乎需求，但这种量测可能不具移植性。

还有其它排序算法。例如 **heap** 算法中有个函数，可直接实作出一个 **heap**（这是个二叉树，应用于 **heapsort** 内部）。**heap** 算法是 **priority queues**（参见 10.3 节，p453）的实作基础。你可以像下面这样将群集内的所有元素排序：

```
/* sort all elements
 * -  $n \cdot n \cdot \log(n)$  complexity
 */
```

```
make_heap (coll.begin(), coll.end());  
sort_heap (coll.begin(), coll.end());
```

关于 heap 和 heap 算法, 细节详见 p406。

如果你只需要排序后的第 n 个元素, 或只需要令最先或最后的 n 个元素(无序)就位, 可以使用 `nth_element()`。可以利用 `nth_element()` 将元素依照某排序准则分割成两个子集。也可利用 `partition()` 或 `stable_partition()` 达到相同效果。三者区别如下:

- 对于 `nth_element()`, 传入第一子集的元素个数(当然也就确定了第二子集的元素个数)。例如:

```
// move the four lowest elements to the front  
nth_element (coll.begin(),      // beginning of range  
             coll.begin()+3,    // position between first and second  
             coll.end());       // end of range
```

然而调用之后, 你并不精确知道第一子集和第二子集之间有什么不同。事实上, 两部分都可能包含和第 n 个元素相等的元素。

- 对于 `partition()`, 你必须传入“将第一子集和第二子集区别开来”的精确排序准则:

```
// move all elements less than seven to the front  
vector<int>::iterator pos;  
pos = partition (coll1.begin(), coll1.end(),      // range  
                bind2nd(less<int>(),7));         // criterion
```

调用之后, 你并不知道第一和第二子集内各有多少元素。返回的 `pos` 指出第二子集起点。第二子集的所有元素都不满足上述(被传入的)准则。

- `stable_partition()` 的行为类似 `partition()`, 不过更具额外能力, 保证两个子集内的元素的相对次序保持不变。

你永远可以把排序准则当做可有可无的参数, 传给所有排序算法。缺省的排序准则是仿函数 `less<>`, 所以元素按升序排列。

和变动性算法的情况一样, 关联式容器不可作为排序算法的目标, 因为关联式容器的元素被视为常数, 不可改动。

`Lists` 没有提供随机存取迭代器, 所以不可以对它使用排序算法。然而 `lists` 本身提供了一个成员函数 `sort()`, 可用来对其元素排序。参见 p245。

已序区间算法 (Sorted Range Algorithms)

所谓已序区间算法, 意指其所作用的区间在某种排序准则下已序。表 9.7 列出 C++ 标准程序库中涵括的所有已序区间算法。和关联式容器一样, 这些算法的优势

就是具有较佳复杂度。

表 9.7 已序区间算法 (Sorted Range Algorithms)

名字	效果	页次
binary_search()	判断某区间内是否包含某个元素	410
includes()	判断某区间内的每一个元素是否都涵盖于另一区间中	411
lower_bound()	搜寻第一个“大于等于给定值”的元素	413
upper_bound()	搜寻第一个“大于给定值”的元素	413
equal_range()	返回“等于给定值”的所有元素构成的区间	415
merge()	将两个区间的元素合并	416
set_union()	求两个区间的并集	418
set_intersection()	求两个区间的交集	419
set_difference()	求位于第一区间但不位于第二区间的所有元素，形成一个已序 (sorted) 区间。	420
set_symmetric_difference()	找出只出现于两区间之一的所有元素，形成一个已序 (sorted) 区间。	421
inplace_merge()	将两个连续的已序 (sorted) 区间合并	423

表 9.7 中前 5 个算法属于非变动性算法，它们只是依命令搜寻元素。其它算法用来将两个已序 (sorted) 区间组合，然后把结果写到目标区间。一般而言，这些算法的结果仍然是已序的。

数值算法 (Numeric Algorithms)

数值算法以不同方式组合数值元素。表 9.8 列出 C++ 标准程序库涵括的所有数值算法。很容易顾名思义。然而它们实际上比你所得的第一印象更加灵活强劲。例如，缺省状态下，accumulate() 求取所有元素的总和。如果你把它作用在 strings 身上，就可以产生字符串连接功效。当你不采用 operator+ 而改用 operator*，得到的会是所有元素的乘积。另一个例子是，你应该了解，adjacent_difference() 和 partial_sum() 可以将某区间在相对值和绝对值之间互相转换。

accumulate() 和 inner_product() 返回一个值，而且不变动区间。其它算法会把结果写到目标区间内，该区间与源区间的元素个数相同。

表 9.8 数值算法 (Numeric Algorithms)

名字	功能	页次
<code>accumulate()</code>	组合所有元素 (求总和、求乘积...)	425
<code>inner_product()</code>	组合两区间内的所有元素	427
<code>adjacent_difference()</code>	将每个元素和其前一元素组合	431
<code>partial_sum()</code>	将每个元素和其先前的所有元素组合	429

9.3 辅助函数

本章剩余部分将对所有 STL 算法逐一详细讨论。每个算法至少配备一个应用实例。为了简化这些例子,使你集中精力于真正重要的问题上,我用了一些辅助函数:

```
// algo/algostuff.hpp

#ifndef ALGOSTUFF_HPP
#define ALGOSTUFF_HPP
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <functional>
#include <numeric>

/* PRINT_ELEMENTS()
 * - prints optional C-string optcstr followed by
 * - all elements of the collection coll
 * - separated by spaces
 */
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;
```

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

/* INSERT_ELEMENTS (collection, first, last)
 * - fill values from first to last into the collection
 * - NOTE: NO half-open range
 */
template <class T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(),i);
    }
}

#endif /*ALGOSTUFF_HPP*/
```

algostuff.hpp 首先含入本程序中可能用到的所有头文件，这样程序本身就不必多劳了。此外它定义了两个辅助函数：

1. PRINT_ELEMENTS() 此函数将第一参数所带入的“容器内的所有元素”打印出来，间以空格。第二参数可有可无，会成为前导词，打印于元素值之前（参见 p118）。
2. INSERT_ELEMENTS() 将元素安插于第一参数所带入的容器内。元素值来自第二参数和第三参数：两参数之间的所有元素值都将供安插之用。请注意这里并不是一个半开区间（half-open range）。

程序输出如下:

```
1 2 3 4 5 6 7 8 9
```

如果想要调用元素的成员函数, 可以使用 `mem_fun` 配接器。细节和范例参见 8.2.2 节, p307。

下面这个例子展示如何利用仿函数来改变每一个元素内容:

```
// algo/foreach2.cpp

#include "alghostuff.hpp"
using namespace std;

// function object that adds the value with which it is initialized
template <class T>
class AddValue {
private:
    T theValue; // value to add

public:
    // constructor initializes the value to add
    AddValue (const T& v) : theValue(v) {
    }

    // the function call for the element adds the value
    void operator() (T& elem) const {
        elem += theValue;
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // add ten to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue<int>(10));      // operation
    PRINT_ELEMENTS(coll);

    // add value of first element to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue<int>{*coll.begin()}); // operation
    PRINT_ELEMENTS(coll);
}
```


`class AddValue<>` 定义了一个仿函数，把构造函数所获得的值加到每一个元素身上。使用仿函数的好处就是，你可以在执行期间传入欲加的数值。程序输出如下：

```
11 12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29 30
```

关于这个例子的细节讨论，请参考 p128。同时请注意，你可以如此这般地运用 `transform()` 算法完成同样任务（参见 p367）：

```
transform (coll.begin(), coll.end(),      // source range
           coll.begin(),                  // destination range
           bind2nd(plus<int>(),10));      // operation
...
transform (coll.begin(), coll.end(),      // source range
           coll.begin(),                  // destination range
           bind2nd(plus<int>(),*coll.begin())); // operation
```

关于 `for_each()` 和 `transform()` 的一般性比较，请见 p325。

第三个例子展示如何利用 `for_each()` 的返回值。由于 `for_each()` 能够返回一项操作，所以我们可以利用这一特性，在该项操作中处理返回结果：

```
// algo/foreach3.cpp

#include "algotuff.hpp"
using namespace std;

// function object to process the mean value
class MeanValue {
private:
    long num; // number of elements
    long sum; // sum of all element values

public:
    // constructor
    MeanValue () : num(0), sum(0) {
    }

    // function call
    // ~ process one more element of the sequence
    void operator()(int elem) {
        num++; // increment count
        sum += elem; // add value
    }
}
```

```
// return mean value (implicit type conversion)
operator double() {
    return static_cast<double>(sum) / static_cast<double>(num);
}

};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,8);

    // process and print mean value
    double mv = for_each (coll.begin(), coll.end(), // range
                          MeanValue());           // operation
    cout << "mean value: " << mv << endl;
}
```

程序输出如下:

```
mean value: 4.5
```

p300 有一个例子，和此例非常相似，但有些微不同，请参考。

9.5 非变动性算法 (Nonmodifying Algorithms)

本节讲述的算法不会变动元素值，也不会改变元素次序。

9.5.1 元素计数

difference_type

count (InputIterator *beg*, InputIterator *end*, const T& *value*)

difference_type

count_if (InputIterator *beg*, InputIterator *end*, UnaryPredicate *op*)

- 第一形式会计算区间 [*beg*, *end*) 中元素值等于 *value* 的元素个数。
- 第二形式会计算区间 [*beg*, *end*) 中令以下一元判断式结果为 *true* 的元素个数：
`op(elem)`
- 返回值型别 *difference_type*，是表现迭代器间距的型别：
`typename iterator_traits<InputIterator>::difference_type`
 7.5 节, p283 曾经介绍过 *iterator traits* (迭代器特性)⁴
- 注意 *op* 在函数调用过程中不应改变自身状态。细节见 8.1.4 节, p302。
- *op* 不应该改动传进来的参数。
- 关联式容器 (*sets*, *multisets*, *maps* 和 *multimaps*) 提供了一个等效的成员函数 `count()`，用来计算等于某个 *value* 或某个 *key* 的元素个数 (见 p234)。
- 复杂度：线性。执行比较动作 (或调用 `op()`) 共 *numberOfElements* 次。

以下范例根据不同的准则对元素进行计数：

```
// algo/count1.cpp

#include "algostuff.hpp"
using namespace std;

bool isEven (int elem)
{
    return elem % 2 == 0;
}
```

⁴ 早期的 STL 中，`count()` 和 `count_if()` 有第四参数，可作为输入，亦能输出，当做计数器使用，返回值型别是 `void`。

```
int main()
{
    vector<int> coll;
    int num;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // count and print elements with value 4
    num = count (coll.begin(), coll.end(), // range
                4); // value
    cout << "number of elements equal to 4: " << num << endl;

    // count elements with even value
    num = count_if (coll.begin(), coll.end(), // range
                   isEven); // criterion
    cout << "number of elements with even value: " << num << endl;

    // count elements that are greater than value 4
    num = count_if (coll.begin(), coll.end(), // range
                   bind2nd(greater<int>(),4)); // criterion
    cout << "number of elements greater than 4: " << num << endl;
}
```

程序输出如下:

```
coll: 1 2 3 4 5 6 7 8 9
number of elements equal to 4: 1
number of elements with even value: 4
number of elements greater than 4: 5
```

你也可以不必使用上述的 `isEven()` 函数, 改用这个表达式:

```
not1(bind2nd(modulus<int>(),2))
```

关于这个表达式的细节, 请见 p306。

9.5.2 最小值和最大值

`InputIterator`

min_element (`InputIterator beg`, `InputIterator end`)

`InputIterator`

min_element (`InputIterator beg`, `InputIterator end`, `CompFunc op`)

```

InputIterator
max_element (InputIterator beg, InputIterator end)

InputIterator
max_element (InputIterator beg, InputIterator end, CompFunc op)

```

- 所有这些算法都返回区间 $[beg, end)$ 中最小或最大元素的位置。
- 上述无 *op* 参数的版本（第一版本），以 `operator<` 进行元素的比较。
- *op* 用来比较两个元素：
`op(elem1, elem2)`
 如果第一个元素小于第二个元素，应当返回 `true`。
- 如果存在多个最小值或最大值，上述算法返回找到的第一个最小或最大值。
- *op* 不应该改动传入的参数。
- 复杂度：线性。执行比较操作（或调用 `op()`）共 *numberOfElements-1* 次。

以下程序打印 `coll` 之中的最小元素和最大元素，并通过 `absLess()` 打印最小元素和最大元素的绝对值：

```

// algo/minmax1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

bool absLess (int elem1, int elem2)
{
    return abs(elem1) < abs(elem2);
}

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll, 2, 8);
    INSERT_ELEMENTS(coll, -3, 5);

    PRINT_ELEMENTS(coll);

    // process and print minimum and maximum
    cout << "minimum: "
         << *min_element(coll.begin(), coll.end())
         << endl;

    cout << "maximum: "
         << *max_element(coll.begin(), coll.end())
         << endl;
}

```

```
// process and print minimum and maximum of absolute values
cout << "minimum of absolute values: "
    << *min_element(coll.begin(), coll.end(),
                    absLess)
    << endl;

cout << "maximum of absolute values: "
    << *max_element(coll.begin(), coll.end(),
                    absLess)
    << endl;
}
```

程序输出如下:

```
2 3 4 5 6 7 8 -3 -2 -1 0 1 2 3 4 5
minimum: -3
maximum: 8
minimum of absolute values: 0
maximum of absolute values: 8
```

注意, 算法分别返回最大和最小元素的位置, 所以你必须使用一元运算符* 来打印其值。

9.5.3 搜寻元素

搜寻第一个匹配元素

```
InputIterator
find (InputIterator beg, InputIterator end, const T& value)

InputIterator
find_if (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- 第一形式返回区间 $[beg, end)$ 中第一个“元素值等于 $value$ ”的元素位置。
- 第二形式返回区间 $[beg, end)$ 中令以下一元判断式结果为 `true` 的第一个元素:
 $op(elem)$
- 如果没有找到匹配元素, 两种形式都返回 `end`。

- 注意 *op* 在函数调用过程中不应改变自身状态。细节见 8.1.4 节, p302。
- *op* 不应该改动传递来的参数。
- 如果是已序区间, 应使用 `lower_bound()`, `upper_bound()`, `equal_range()` 或 `binary_search()` 算法以获取更高性能 (参见 9.10 节, p409)。
- 关联式容器 (sets, multisets, maps, multimaps) 提供了一个等效的成员函数 `find()`, 拥有对数复杂度, 而非线性复杂度。
- 复杂度: 线性。最多比较 (或调用 *op*()) 共 *numberOfElements* 次。

下面这个例子展示如何运用 `find()` 来搜寻一个子区间: 以元素值为 4 的第一个元素开始, 以元素值为 4 的第二个元素结束:

```
// algo/find1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll, "coll: ");

    // find first element with value 4
    list<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),      // range
                4);                             // value

    /* find second element with value 4
     * - note: continue the search behind the first 4 (if any)
     */
    list<int>::iterator pos2;
    if (pos1 != coll.end()) {
        pos2 = find (++pos1, coll.end(),      // range
                    4);                       // value
    }
}
```

```

    /* print all elements from first to second 4 (both included)
    * - note: now we need the position of the first 4 again (if any)
    * - note: we have to pass the position behind the second 4
    * (if any)
    */
    if (pos1!=coll.end() && pos2!=coll.end()) {
        copy (--pos1, ++pos2,
              ostream_iterator<int>(cout," "));
        cout << endl;
    }
}

```

为了搜寻第二个 4, 你必须从第一个 4 的位置上前进。然而如果在群集的 `end()` 位置上再前进, 会导致未定义的行为。所以如果你没有十足把握, 最好在前进之前先检查 `find()` 的返回值。程序输出如下:

```

coll: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3 4

```

你可以在同一个区间中以不同的值先后两次调用 `find()`。然而, 当你使用搜寻结果作为子区间的起点和终点时, 务必十分小心, 否则该子区间可能形成一个无效区间。此问题的讨论和范例请见 p97。

下面这个程序展示 `find_if()` 的用法, 以极为特殊的搜寻准则来搜寻某个元素:

```

// algo/find2.cpp

#include "alghostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"coll: ");

    // find first element greater than 3
    pos = find_if (coll.begin(), coll.end(),           // range
                   bind2nd(greater<int>(),3));         // criterion
}

```



```

// print its position
cout << "the "
    << distance(coll.begin(),pos) + 1
    << ". element is the first greater than 3" << endl;

// find first element divisible by 3
pos = find_if (coll.begin(), coll.end(),
              not1(bind2nd(modulus<int>(),3)));

// print its position
cout << "the "
    << distance(coll.begin(),pos) + 1
    << ". element is the first divisible by 3" << endl;
}

```

第一次调用 `find()` 时,使用了一个以 `bind2nd` 配接器组合而成的简单仿函数,搜寻第一个大于 3 的元素。第二次调用使用一个比较复杂的组合,搜寻第一个可被 3 整除的元素。

程序输出如下:

```

coll: 1 2 3 4 5 6 7 8 9
the 4. element is the first greater than 3
the 3. element is the first divisible by 3

```

p121 有一个例子,以 `find()` 搜寻第一个质数。

搜寻前 n 个连续匹配值

```

InputIterator
search_n (InputIterator beg, InputIterator end,
          Size count, const T& value)

InputIterator
search_n (InputIterator beg, InputIterator end,
          Size count, const T& value, BinaryPredicate op)

```

- 第一形式返回区间 $[beg, end)$ 中第一组“连续 $count$ 个元素值全等于 $value$ ”的元素位置。
- 第二形式返回区间 $[beg, end)$ 中第一组“连续 $count$ 个元素造成以下一元判断式结果为 `true`”的元素位置:
 $op(elem, value)$
- 如果没有找到匹配元素,两种形式都返回 `end`。

- 注意, `op` 在函数调用过程中不应改变自身状态。细节见 8.1.4 节, p302。
- `op` 不应该变动被传进去的参数。
- 这两个算法并不在早期的 STL 规范中, 也没有获得谨慎的对待, 因此, 第二形式使用了一个二元判断式, 而非一元判断式, 这破坏了早期 STL 的一致性。请参见 p346。
- 复杂度: 线性。最多比较 (或调用 `op()`) 共 $numberOfElements * count$ 次。

下面这个例子搜寻连续 4 个 “数值大于等于 3” 的元素:

```
// algo/searchn1.cpp

#include "alghostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // find four consecutive elements with value 3
    deque<int>::iterator pos;
    pos = search_n (coll.begin(), coll.end(), // range
                   4,                          // count
                   3);                          // value

    // print result
    if (pos != coll.end()) {
        cout << "four consecutive elements with value 3 "
              << "start with " << distance(coll.begin(),pos) +1
              << ". element" << endl;
    }
    else {
        cout << "no four consecutive elements with value 3 found"
              << endl;
    }
}
```

```

// find four consecutive elements with value greater than 3
pos = search_n (coll.begin(), coll.end(), // range
               4,                          // count
               3,                          // value
               greater<int>());             // criterion

// print result
if (pos != coll.end()) {
    cout << "four consecutive elements with value > 3 "
         << "start with " << distance(coll.begin(),pos) +1
         << ". element" << endl;
}
else {
    cout << "no four consecutive elements with value > 3 found"
         << endl;
}
}

```

程序输出如下:

```

1 2 3 4 5 6 7 8 9
no four consecutive elements with value 3 found
four consecutive elements with value > 3 start with 4. element

```

关于 `search_n()` 的第二形式, 有一个险恶的问题。请看以下调用操作:

```

pos = search_n (coll.begin(), coll.end(),    // range
               4,                          // count
               3,                          // value
               greater<int>());             // criterion

```

以这种方法来搜寻符合某特殊准则的元素, 其方法和 STL 其它组件可谓大相径庭。按照 STL 的一般观念, 应该这么做:

```

pos = search_n_if (coll.begin(), coll.end(), // range
                  4,                          // count
                  bind2nd(greater<int>(),3)); // criterion

```

可惜, 当这个新算法被引入 C++ 标准时 (它不属于早期的 STL), 没有人注意到其中的不一致。当然, 也许你觉得 4 个参数的形式更加便捷。不过它实际上只需要一个一元判断式的时候, 却要求获得一个二元判断式, 这恐怕非你所愿。例如, 为了运用自己写的一个一元判断式, 你通常会这么做:

```

bool isPrime (int elem);
...
pos = search_n_if (coll.begin(), coll.end(), // range
                  4,                          // count
                  isPrime);                  // criterion

```

然而, 根据实际定义, 你却必须写一个二元判断式。所以你要不改变函数标记式 (signature), 要不就写一个简单的包装:

```

bool binaryIsPrime (int elem1, int) {
    return isPrime(elem1);
}
...
pos = search_n (coll.begin(), coll.end(), // range
               4,                          // count
               0,                          // required dummy val
               binaryIsPrime);             // binary criterion

```

搜寻第一个子区间

```

ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
        BinaryPredicate op)

```

- 两种形式都返回区间 $[beg, end)$ 内“和区间 $[searchBeg, searchEnd)$ 完全吻合”的第一个子区间内的第一个元素位置。
- 第一形式中, 子区间的元素必须完全等于 $[searchBeg, searchEnd)$ 的元素。
- 第二形式中, 子区间的元素和 $[searchBeg, searchEnd)$ 的对应元素必须造成以下二元判断式的结果为 true:
`op(elem, searchElem)`
- 如果没有找到符合条件的子区间, 两种形式都返回 `end`。
- 注意, `op` 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- `op` 不应变动传入的参数。
- 如果你在“只知道第一个元素和最后一个元素”的情况下要搜寻一个子区间, 请参见 p97。

- 复杂度：线性。最多比较（或调用 `op()`）共 $numberOfElements * numberOfSearchElements$ 次。

下面这个例子展示如何在另一个序列中搜寻一个子序列。请将这个例子和 p351 的 `find_end()` 运用实例作一番比较。

```
// algo/search1.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);

    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(subcoll, "subcoll: ");

    // search first occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),           // range
                  subcoll.begin(), subcoll.end()); // subrange

    // loop while subcoll found as subrange of coll
    while (pos != coll.end()) {
        // print position of first element
        cout << "subcoll found starting with element "
              << distance(coll.begin(),pos) + 1
              << endl;
        // search next occurrence of subcoll
        ++pos;
        pos = search (pos, coll.end(),                // range
                      subcoll.begin(), subcoll.end()); // subrange
    }
}
```

程序输出如下:

```
coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 3
subcoll found starting with element 10
```

下面这个例子展示如何运用 `search()` 算法的第二形式, 以更复杂的准则来搜寻某个子序列。这里搜寻的是“偶数、奇数、偶数”排列而成的子序列:

```
// algo/search2.cpp

#include "alghostuff.hpp"
using namespace std;

// checks whether an element is even or odd
bool checkEven (int elem, bool even)
{
    if (even) {
        return elem % 2 == 0;
    }
    else {
        return elem % 2 == 1;
    }
}

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    /* arguments for checkEven()
     * - check for: "even odd even"
     */
    bool checkEvenArgs[3] = { true, false, true };

    // search first subrange in coll
    vector<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),           // range
                  checkEvenArgs, checkEvenArgs+3,    // subrange values
                  checkEven);                          // subrange criterion
```

```

    // loop while subrange found
    while (pos != coll.end()) {
        // print position of first element
        cout << "subrange found starting with element "
              << distance(coll.begin(), pos) + 1
              << endl;

        // search next subrange in coll
        pos = search (++pos, coll.end(),           // range
                     checkEvenArgs, checkEvenArgs+3, // subr. values
                     checkEven);                  // subr. criterion
    }
}

```

程序输出如下:

```

coll: 1 2 3 4 5 6 7 8 9
subrange found starting with element 2
subrange found starting with element 4
subrange found starting with element 6

```

(译注: 以上表示共找出三个满足“偶、奇、偶”条件的子序列, 分别始自元素 2,4,6)

搜寻最后一个子区间

```

ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
           ForwardIterator searchBeg, ForwardIterator searchEnd)

ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
           ForwardIterator searchBeg, ForwardIterator searchEnd,
           BinaryPredicate op)

```

- 两种形式都返回区间 [beg, end) 之中 “和区间 [searchBeg, searchEnd) 完全吻合” 的最后一个子区间内的第一个元素位置。
- 第一形式中, 子区间的元素必须完全等于 [searchBeg, searchEnd) 的元素。
- 第二形式中, 子区间的元素和 [searchBeg, searchEnd) 的对应元素必须造成以下二元判断式的结果为 true:
`op(elem, searchElem)`

- 如果没有找到符合条件的子区间，两种形式都返回 *end*。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 如果你在“只知道第一个元素和最后一个元素”的情况下要搜寻一个子区间，请参考 p97。
- 这些算法并不是早期 STL 的一部分。很不幸它们被命名为 *find_end()* 而不是 *search_end()*，如果是后者，比较具有一致性，因为用来搜寻第一个子区间的算法名为 *search()*。
- 复杂度：线性。最多比较（或调用 *op()*）共 *numberOfElements * numberOfSearchElements* 次。

下面这个例子展示如何在一个序列中搜寻“与某序列相等”的最后一个子序列（请和 p348 的 *search()* 例子作比较）：

```
// algo/findendl.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);

    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(subcoll, "subcoll: ");

    // search last occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = find_end (coll.begin(), coll.end(), // range
                   subcoll.begin(), subcoll.end()); // subrange

    // loop while subcoll found as subrange of coll
    deque<int>::iterator end(coll.end());
```



```

while (pos != end) {
    // print position of first element
    cout << "subcoll found starting with element "
        << distance(coll.begin(), pos) + 1
        << endl;

    // search next occurrence of subcoll
    end = pos;
    pos = find_end (coll.begin(), end,                // range
                   subcoll.begin(), subcoll.end()); // subrange
}
}

```

程序输出如下:

```

coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 10
subcoll found starting with element 3

```

这个算法的第二形式, 可参考 p349 `search()` 的例子。你可以采用类似的手法来使用 `find_end()`。

搜寻某些元素的第一次出现地点

```

ForwardIterator
find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
               ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator
find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
               ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
               BinaryPredicate op)

```

- 第一形式返回第一个“既在区间 $[beg, end)$ 中出现, 也在区间 $[searchBeg, searchEnd)$ 中出现”的元素的位置。
- 第二形式返回区间 $[beg, end)$ 中第一个这样的元素: 它和区间 $[searchBeg, searchEnd)$ 内每一个元素进行以下动作的结果都是 `true`:
`op(elem, searchElem)`
- 如果没有找到吻合元素, 两种形式都返回 `end`。
- 注意, `op` 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。

- `op` 不应改动传入的参数。
- 你可以使用逆向迭代器来搜寻最后一个这样的元素。
- 这几个算法并不在早期的 STL 规范中。
- 复杂度：线性。最多比较（或调用 `op()`）共 $numberOfElements * numberOfSearchElements$ 次。

下面这个例子展示 `find_first_of()` 的用法：

```
// algo/findof1.cpp

#include "algotstuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    list<int> searchcoll;

    INSERT_ELEMENTS(coll,1,11);
    INSERT_ELEMENTS(searchcoll,3,5);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(searchcoll, "searchcoll: ");

    // search first occurrence of an element of searchcoll in coll
    vector<int>::iterator pos;
    pos = find_first_of (coll.begin(), coll.end(), // range
                        searchcoll.begin(), // beginning of search set
                        searchcoll.end()); // end of search set

    cout << "first element of searchcoll in coll is element "
         << distance(coll.begin(), pos) + 1
         << endl;

    // search last occurrence of an element of searchcoll in coll
    vector<int>::reverse_iterator rpos;
    rpos = find_first_of (coll.rbegin(), coll.rend(), // range
                        searchcoll.begin(), // beginning of search set
                        searchcoll.end()); // end of search set
    cout << "last element of searchcoll in coll is element "
         << distance(coll.begin(), rpos.base())
         << endl;
}
```

第二次调用系采用逆向迭代器，搜寻最后一个“与 `searchcoll` 内某一元素相等”的元素。为了打印这个元素的位置，此处调用 `base()` 将逆向迭代器转化成一般(正向)迭代器。这样你就可以得到从起点算起的距离。通常你应该将 `distance()` 的结果加 1，因为第一个元素的距离是 0。然而因为 `base()` 背地里改变了迭代器的所指数值位置，所以你得到相同的效果。关于 `base()`，请见 7.4.1 节, p269。

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9 10 11
searchcoll: 3 4 5
first element of searchcoll in coll is element 3
last element of searchcoll in coll is element 5
```

搜寻两个连续且相等的元素

```
InputIterator
adjacent_find (InputIterator beg, InputIterator end)

InputIterator
adjacent_find (InputIterator beg, InputIterator end,
               BinaryPredicate op)
```

- 第一形式返回区间 $[beg, end)$ 中第一对“连续两个相等元素”之中的第一元素位置。
- 第二形式返回区间 $[beg, end)$ 中第一对“连续两个元素均使以下二元判断式的结果为 `true`”的其中第一元素位置：
`op(elem, nextElem)`
- 如果没有找到吻合元素，两者都返回 `end`。
- 注意，`op` 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- `op` 不应改动传入的参数。
- 复杂度：线性。最多比较（或调用 `op()`）共 *numberOfElements* 次。

下面这个程序展示 `adjacent_find()` 两种形式的用法：

```
// algo/adjfind1.cpp

#include "algostuff.hpp"
using namespace std;

// return whether the second object has double the value of the first
bool doubled (int elem1, int elem2)
{
```

```
    return elem1 * 2 == elem2;
}

int main()
{
    vector<int> coll;

    coll.push_back(1);
    coll.push_back(3);
    coll.push_back(2);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(5);
    coll.push_back(0);

    PRINT_ELEMENTS(coll, "coll: ");

    // search first two elements with equal value
    vector<int>::iterator pos;
    pos = adjacent_find (coll.begin(), coll.end());

    if (pos != coll.end()) {
        cout << "first two elements with equal value have position "
              << distance(coll.begin(), pos) + 1
              << endl;
    }

    // search first two elements for which the second has
    // double the value of the first
    pos = adjacent_find (coll.begin(), coll.end(), // range
                        doubled);                  // criterion

    if (pos != coll.end()) {
        cout << "first two elements with second value twice the "
              << "first have pos. "
              << distance(coll.begin(), pos) + 1
              << endl;
    }
}
```

第一次调用 `adjacent_find()` 是为了搜寻相等值。第二次调用以 `doubled()` 来搜寻“连续两元素，后一个元素是前一个元素的两倍”，找到后返回其中第一个元素的位置。程序输出如下：

```
coll: 1 3 2 4 5 5 0
first two elements with equal value have position 5
first two elements with second value twice the first have pos. 3
```

9.5.4 区间的比较

检验相等性

```
bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg)

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg, BinaryPredicate op)
```

- 第一形式判断区间 $[beg, end)$ 内的元素是否都和“以 `cmpBeg` 开头的区间”内的元素相等。
- 第二形式判断区间 $[beg, end)$ 内的元素和“以 `cmpBeg` 开头的区间”内的对应元素”是否都能够使以下二元判断式得到 `true`:
`op(elem, cmpElem)`
- 注意，`op` 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- `op` 不应改动传入的参数。
- 调用者必须确保“以 `cmpBeg` 开头的区间”内含足够元素。
- 当序列不相等时，如果想要了解其间的不同，应使用 `mismatch()` 算法（参见 p358）。
- 复杂度：线性。最多比较（或调用 `op()`）共 `numberOfElements` 次。

下面这个例子展示 `equal()` 两种形式的用法。第一次调用用来检查元素是否相等。第二次调用使用一个辅助判断式，检查两个群集中的元素是否具备一一对应的奇偶关系：

```
// algo/equals1.cpp

#include "algotuff.hpp"
using namespace std;
```

```
bool bothEvenOrOdd (int elem1, int elem2)
{
    return elem1 % 2 == elem2 % 2;
}

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

    INSERT_ELEMENTS(coll1,1,7);
    INSERT_ELEMENTS(coll2,3,9);

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // check whether both collections are equal
    if (equal (coll1.begin(), coll1.end(),      // first range
              coll2.begin())) {                 // second range
        cout << "coll1 == coll2" << endl;
    }
    else {
        cout << "coll1 != coll2" << endl;
    }

    // check for corresponding even and odd elements
    if (equal (coll1.begin(), coll1.end(), // first range
              coll2.begin(),              // second range
              bothEvenOrOdd)) {           // comparison criterion
        cout << "even and odd elements correspond" << endl;
    }
    else {
        cout << "even and odd elements do not correspond" << endl;
    }
}
```

程序输出如下:

```
coll1: 1 2 3 4 5 6 7
coll2: 3 4 5 6 7 8 9
coll1 != coll2
even and odd elements correspond
```

搜寻第一处不同点

```
pair<InputIterator1, InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg)

pair<InputIterator1, InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg,
          BinaryPredicate op)
```

- 第一形式返回区间 $[beg, end)$ 和 “以 *cmpBeg* 开头的区间” 之中第一组两两相异的对应元素。
- 第二形式返回区间 $[beg, end)$ 和 “以 *cmpBeg* 开头的区间” 之中第一组 “使以下二元判断式获得 *false*” 的对应元素：
 $op(elem, cmpElem)$
- 如果没有找到相异点，就返回一个 *pair*，以 *end* 和第二序列的对应元素组成。这并不意味两个序列相等，因为第二序列有可能包含比较多的元素。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节，p302。
- *op* 不应改动传入的参数。
- 调用者必须确保 “以 *cmpBeg* 开头的区间” 内含足够元素。
- 如果想知道两个序列是否相等，应当使用 *equal()* 算法 (p356)。
- 复杂度：线性。最多比较（或调用 *op()*）共 *numberOfElements* 次。

下面这个例子展示 *mismatch()* 两种形式的用法：

```
// algo/misma1.cpp

#include "algostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1, 1, 6);

    for (int i=1; i<=16; i*=2) {
        coll2.push_back(i);
    }
    coll2.push_back(3);
```

```
PRINT_ELEMENTS(coll1, "coll1: ");
PRINT_ELEMENTS(coll2, "coll2: ");

// find first mismatch
pair<vector<int>::iterator, list<int>::iterator> values;
values = mismatch (coll1.begin(), coll1.end(), // first range
                  coll2.begin()); // second range
if (values.first == coll1.end()) {
    cout << "no mismatch" << endl;
}
else {
    cout << "first mismatch: "
         << *values.first << " and "
         << *values.second << endl;
}

/* find first position where the element of coll1 is not
 * less than the corresponding element of coll2
 */
values = mismatch (coll1.begin(), coll1.end(), // first range
                  coll2.begin(), // second range
                  less_equal<int>()); // criterion
if (values.first == coll1.end()) {
    cout << "always less-or-equal" << endl;
}
else {
    cout << "not less-or-equal: "
         << *values.first << " and "
         << *values.second << endl;
}
}
```

第一次调用 `mismatch()` 用以搜寻第一对互异的对应元素。如果找到了, 就把它们的值写到标准输出装置。第二次调用用来搜寻符合以下条件的第一对元素: “第一序列的元素比第二序列的对应元素大”, 找到后返回该两个元素。程序输出如下:


```
coll1: 1 2 3 4 5 6
coll2: 1 2 4 8 16 3
first mismatch: 3 and 4
not less-or-equal: 6 and 3
```

检验“小于”

```
bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                        InputIterator2 beg2, InputIterator2 end2)

bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                        InputIterator2 beg2, InputIterator2 end2,
                        CompFunc op)
```

- 两种形式都用来判断区间 $[beg1, end1)$ 的元素是否小于区间 $[beg2, end2)$ 的元素。所谓“小于”是指本着“字典 (lexicographical) 次序”的意义。
- 第一形式以 `operator<` 来比较元素。
- 第二形式以二元判断式 `op(elem1, elem2)` 比较元素。如果 `elem1` 小于 `elem2`，则判断式应当返回 `true`。
- “字典次序”意味两个序列中的元素一一比较，直到以下情况发生：
 - 如果两元素不相等，则这两个元素的比较结果就是整个两序列的比较结果。
 - 如果两序列中的元素数量不同，则元素较少的那个序列小于另一序列。所以，如果第一序列的元素数量较少，比较结果是 `true`。
 - 如果两序列都没有更多的元素可作比较，则这两个序列相等，整个比较结果是 `false`。
- 注意，`op` 在函数调用过程中不应改变自身状态。详见 8.1.4 节，p302。
- `op` 不应改动传入的参数。
- 复杂度：线性。最多比较（或调用 `op()`）共 $2 * \min(\text{numberOfElements1}, \text{numberOfElements2})$ 次。

下面这个例子展示如何利用这个算法对群集完成“字典次序”的排序：

```
// algo/lexico1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
void printCollection (const list<int>& l)
{
    PRINT_ELEMENTS(l);
}

bool lessForCollection (const list<int>& l1, const list<int>& l2)
{
    return lexicographical_compare
        (l1.begin(), l1.end(), // first range
         l2.begin(), l2.end()); // second range
}

int main()
{
    list<int> c1, c2, c3, c4;

    // fill all collections with the same starting values
    INSERT_ELEMENTS(c1,1,5);
    c4 = c3 = c2 = c1;

    // and now some differences
    c1.push_back(7);
    c3.push_back(2);
    c3.push_back(0);
    c4.push_back(2);

    // create collection of collections
    vector<list<int> > cc;

    cc.push_back(c1);
    cc.push_back(c2);
    cc.push_back(c3);
    cc.push_back(c4);
    cc.push_back(c3);
    cc.push_back(c1);
    cc.push_back(c4);
    cc.push_back(c2);
```

```

    // print all collections
    for_each (cc.begin(), cc.end(),
              printCollection);
    cout << endl;

    // sort collection lexicographically
    sort (cc.begin(), cc.end(), // range
          lessForCollection); // sorting criterion

    // print all collections again
    for_each (cc.begin(), cc.end(),
              printCollection);
}

```

其中 `vector cc` 是由好几个群集（都是 `lists`）初始化而来。调用 `sort()` 时使用了二元判断式 `lessForCollection()` 来比较两群集（关于 `sort()`，详见 p397）。在 `lessForCollection()` 中，算法 `lexicographical_compare()` 用来对群集进行字典序比较。程序输出如下：

```

1 2 3 4 5 7
1 2 3 4 5
1 2 3 4 5 2 0
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 2
1 2 3 4 5

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 2
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 7

```

9.6 变动性算法 (Modifying Algorithms)

本节描述的算法会变动区间内的元素内容。有两种方法可以变动元素内容：

1. 运用迭代器遍历序列的过程中，直接加以变动。
2. 将元素从源 (source) 区间复制到目标 (destination) 区间的过程中加以变动。

某些算法同时提供以上两种方法，那么使用第二法的版本会有尾词 `_copy`。

注意，目标区间不可以是关联式容器，因为关联式容器的元素被视为常数。如果没有这个限制，其自动排序特性将无法得到保证。

所有具有单一目标区间的算法，都返回区间内“最后一个被复制元素”的下一位置。

9.6.1 复制 (Copying) 元素

OutputIterator

```
copy (InputIterator sourceBeg,
      InputIterator sourceEnd,
      OutputIterator destBeg)
```

BidirectionalIterator1

```
copy_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- 这两个算法都将源区间 $[sourceBeg, sourceEnd)$ 中的所有元素复制到以 `destBeg` 为起点或以 `destEnd` 为终点的目标区间去。
- 返回目标区间内最后一个被复制元素的下一位置，也就是第一个未被覆盖 (overwritten) 的元素的位置。
- `destBeg` 或 `destEnd` 不可处于 $[sourceBeg, sourceEnd)$ 区间内。
- `copy()` 正向遍历序列，而 `copy_backward()` 逆向遍历序列。只有在源区间和目标区间存在重复区域时，这个不同点才会导致一些问题：

——如果要把一个区间复制到前端，应使用 `copy()`。所以 `destBeg` 的位置应该在 `sourceBeg` 之前。

——如果要把一个区间复制到后端，应使用 `copy_backward()`。所以 `destEnd` 的位置应该在 `sourceEnd` 之后。

所以，只要第三参数位于由前两个参数所确定下来的源区间中，你就应该使用另一形式。注意，如果转而使用另一形式，就意味你原本应传递目标区间的起点，现在要转而传递终点了。关于两者的区别，请参考 p365 的例子。

- STL 并没有所谓 `copy_if()` 算法，所以如果要复制符合某特定准则的元素，请使用 `remove_copy_if()`，参见 p380。

- 如果希望在复制过程中逆转元素次序, 应使用 `reverse_copy()` (参见 p386)。该算法比 “`copy()` 算法搭配逆向迭代器” 略快。
- 调用者必须确保目标区间有足够空间, 要不就得使用 `insert` 迭代器。
- 关于 `copy()` 算法的实作, 请看 p271。
- 如果想把容器内的所有元素赋值 (*assign*) 给另一个容器, 应当使用 `assignment` 运算符 (当两个容器的型别相同时才能这么做, 参见 p236) 或使用容器的 `assign()` 成员函数 (当两个容器的型别不同时就采用此法, 参见 p237)。
- 如果希望在复制过程中删除某些元素, 应使用算法 `remove_copy()` 和 `remove_copy_if()` (参见 p380)。
- 如果希望在复制过程中改变元素, 请使用算法 `transform()` (参见 p367) 或 `replace_copy` (参见 p367)。
- 复杂度: 线性, 执行 *numberOfElements* 次赋值 (*assign*) 动作。

下面这个例子展示 `copy()` 的一些简单用法:

```
// algo/copy1.cpp

#include "algotuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1,1,9);

    /* copy elements of coll1 into coll2
     * - use back inserter to insert instead of overwrite
     */
    copy (coll1.begin(), coll1.end(),          // source range
          back_inserter(coll2));              // destination range

    /* print elements of coll2
     * - copy elements to cout using an ostream iterator
     */
    copy (coll2.begin(), coll2.end(),          // source range
          ostream_iterator<int>(cout," "));   // destination range
    cout << endl;
```

```

/* copy elements of coll1 into coll2 in reverse order
 * - now overwriting
 */
copy (coll1.rbegin(), coll1.rend(), // source range
      coll2.begin());              // destination range

// print elements of coll2 again
copy (coll2.begin(), coll2.end(), // source range
      ostream_iterator<int>(cout, " ")); // destination range
cout << endl;
}

```

在这个例子里, `backinserters` (参见 7.4.2 节, p272) 用来在目标区间中安插元素。如果不使用这类安插型迭代器, `copy()` 算法就会在空的 `coll2` 容器上实施覆盖 (overwritten) 操作, 导致未定义的行为。同样道理, 我们可使用 `ostream_iterator` 把标准输出装置当成目标, 参见 7.4.3 节, p278 实例。

程序输出如下:

```

1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1

```

下面这个例子展示 `copy()` 和 `copy_backward()` 之间的区别:

```

// algo/copy2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    /* initialize source collection with ".....abcdef....."
     */
    vector<char> source(10, '.');
    for (int c='a'; c<='f'; c++) {
        source.push_back(c);
    }
    source.insert(source.end(), 10, '.');
    PRINT_ELEMENTS(source, "source: ");

    // copy all letters three elements in front of the 'a'
    vector<char> c1(source.begin(), source.end());
    copy (c1.begin()+10, c1.begin()+16, // source range
          c1.begin()+7);                // destination range
    PRINT_ELEMENTS(c1, "c1: ");
}

```

```

// copy all letters three elements behind the 'f'
vector<char> c2(source.begin(),source.end());
copy_backward (c2.begin()+10, c2.begin()+16, // source range
               c2.begin()+19); // destination range
PRINT_ELEMENTS(c2,"c2: ");
}

```

注意，无论是调用 `copy()` 或是 `copy_backward()`，第三参数都不处于源区间中。程序输出如下：

```

source: . . . . . a b c d e f . . . . .
c1:      . . . . . a b c d e f d e f . . . . .
c2:      . . . . . a b c a b c d e f . . . . .

```

第三个例子示范如何使用 `copy()` 作为标准输入装置和标准输出装置之间的数据过滤器：程序读取 `strings`，并以一行一个的方式打印它们：

```

// algo/copy3.cpp

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

int main()
{
    copy (istream_iterator<string>(cin), // beginning of source
          istream_iterator<string>(),    // end of source
          ostream_iterator<string>(cout, "\n")); // destination
}

```

9.6.2 转换 (Transforming) 和结合 (Combining) 元素

算法 `transform()` 提供以下两种能力：

1. 第一形式有 4 个参数。把源区间的元素转换到目标区间。也就是说，复制和修改元素一气呵成。
2. 第二形式有 5 个参数，将前两个源序列中的元素合并，并将结果写入目标区间。

转换元素

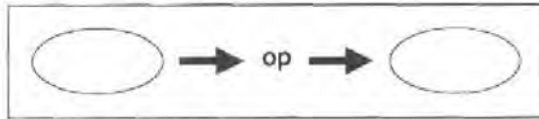
OutputIterator

```
transform (InputIterator sourceBeg, InputIterator sourceEnd,  
           OutputIterator destBeg, UnaryFunc op)
```

- 针对源区间 $[sourceBeg, sourceEnd)$ 中的每一个元素调用:

$op(elem)$

并将结果写到以 $destBeg$ 起始的目标区间内。



- 返回目标区间内“最后一个被转换元素”的下一位置，也就是第一个未被覆盖 (overwritten) 的元素的位置。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- $sourceBeg$ 与 $destBeg$ 可以相同，所以，和 `for_each()` 算法一样，你可以使用这个算法来变动某一序列内的元素。请看 p325 对两者的比较。
- 如果想以某值替换符合某一准则的元素，应使用 `replace()` 算法 (见 p375)。
- 复杂度：线性，对 $op()$ 执行 $numberOfElements$ 次调用。

下面这个例子展示以上所说的 `transform()` 用法：

```
// algo/transf1.cpp

#include "alghostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");

    // negate all elements in coll1
    transform (coll1.begin(), coll1.end(), // source range
               coll1.begin(),              // destination range
               negate<int>());              // operation
    PRINT_ELEMENTS(coll1,"negated: ");
```



```

// transform elements of coll1 into coll2 with
// ten times their value
transform (coll1.begin(), coll1.end(),      // source range
          back_inserter(coll2),           // destination range
          bind2nd(multiplies<int>(),10)); // operation
PRINT_ELEMENTS(coll2,"coll2: ");

// print coll2 negatively and in reverse order
transform (coll2.rbegin(), coll2.rend(),    // source range
          ostream_iterator<int>(cout," "), // destination range
          negate<int>());                   // operation
cout << endl;
}

```

程序输出如下:

```

coll1: 1 2 3 4 5 6 7 8 9
negated: -1 -2 -3 -4 -5 -6 -7 -8 -9
coll2: -10 -20 -30 -40 -50 -60 -70 -80 -90
90 80 70 60 50 40 30 20 10

```

关于如何“将两种不同操作组合起来, 对元素进行处理”, 请见 p315 实例。

将两序列的元素加以结合 (Combining)

OutputIterator

```

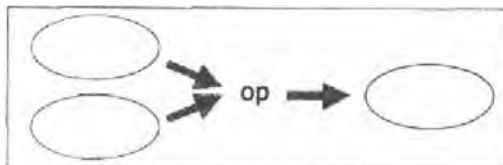
transform (InputIterator1 source1Beg, InputIterator1 source1End,
           InputIterator2 source2Beg,
           OutputIterator destBeg,
           BinaryFunc op)

```

- 针对第一个源区间 $[source1Beg, source1End)$ 以及“从 $source2Beg$ 开始的第二个源区间”的对应元素, 调用:

$op(source1Elem, source2Elem)$

并将结果写入以 $destBeg$ 起始的目标区间内。



- 返回目标区间内的“最后一个被转换元素”的下一位置，就是第一个未被覆盖 (overwritten) 的元素的位置。
- 调用者必须保证第二源区间有足够空间 (至少拥有和第一源区间相同的空间大小)。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- *source1Beg*, *source2Beg*, *destBeg* 可以相同。所以，你可以让元素自己和自己结合，然后将结果覆盖 (overwritten) 至某个序列。
- 复杂度：线性，对 *op()* 执行 *numberOfElements* 次调用。

下面这个例子展示以上所说的 *transform()* 用法：

```
// algo/transf2.cpp

#include "alghostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");

    // square each element
    transform (coll1.begin(), coll1.end(), // first source range
               coll1.begin(),             // second source range
               coll1.begin(),             // destination range
               multiplies<int>());         // operation
    PRINT_ELEMENTS(coll1,"squared: ");

    /* add each element traversed forward with each element traversed
    backward
    * and insert result into coll2
    */
    transform (coll1.begin(), coll1.end(), // first source range
               coll1.rbegin(),             // second source range
               back_inserter(coll2),       // destination range
               plus<int>());               // operation
    PRINT_ELEMENTS(coll2,"coll2: ");
```

```
// print differences of two corresponding elements
cout << "diff: ";
transform (coll1.begin(), coll1.end(), // first source range
           coll2.begin(),              // second source range
           ostream_iterator<int>(cout, " "), // destination range
           minus<int>());               // operation
cout << endl;
}
```

程序输出如下:

```
coll1: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
coll2: 82 68 58 52 50 52 58 68 82
diff: -81 -64 -49 -36 -25 -16 -9 -4 -1
```

9.6.3 互换 (Swapping) 元素内容

```
ForwardIterator2
swap_ranges (ForwardIterator1 beg1, ForwardIterator1 end1,
              ForwardIterator2 beg2)
```

- 将区间 $[beg1, end1)$ 内的元素和“从 $beg2$ 开始的区间”内的对应元素互换。
- 返回第二区间中“最后一个被交换元素”的下一位置。
- 调用者必须确保目标区间有足够空间。
- 两区间不得重迭。
- 如果要将相同型别的两个容器内的全部元素互换, 应使用 `swap()` 成员函数, 因为该成员函数通常具有常数复杂度 (参见 p237)。
- 复杂度: 线性, 执行 *numberOfElements* 次交换操作。

下面这个例子展示 `swap_ranges()` 的用法:

```
// algo/swap1.cpp

#include "algotuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1, 1, 9);
```

```

INSERT_ELEMENTS(coll2,11,23);

PRINT_ELEMENTS(coll1,"coll1: ");
PRINT_ELEMENTS(coll2,"coll2: ");

// swap elements of coll1 with corresponding elements of coll2
deque<int>::iterator pos;
pos = swap_ranges (coll1.begin(), coll1.end(), // first range
                  coll2.begin());           // second range

PRINT_ELEMENTS(coll1,"\ncoll1: ");
PRINT_ELEMENTS(coll2,"coll2: ");
if (pos != coll2.end()) {
    cout << "first element not modified: "
          << *pos << endl;
}

// mirror first three with last three elements in coll2
swap_ranges (coll2.begin(), coll2.begin()+3, // first range
            coll2.rbegin());                // second range

PRINT_ELEMENTS(coll2,"\ncoll2: ");
}

```

第一次调用 `swap_ranges()` 是为了将 `coll1` 的元素和 `coll2` 的对应元素交换。`coll2` 之内的剩余元素不予变动。`swap_ranges()` 算法返回第一个未被改动的元素。第二次调用 `swap_ranges()` 是为了将 `coll2` 内的前 3 个元素和后 3 个元素交换。由于运用了逆向迭代器，所以元素以镜像方式交换（从外向内交换）。程序输出如下：

```

coll1: 1 2 3 4 5 6 7 8 9
coll2: 11 12 13 14 15 16 17 18 19 20 21 22 23

coll1: 11 12 13 14 15 16 17 18 19
coll2: 1 2 3 4 5 6 7 8 9 20 21 22 23
first element not modified: 20

coll2: 23 22 21 4 5 6 7 8 9 20 3 2 1

```

9.6.4 赋予 (Assigning) 新值

赋予完全相同的数值

```
Void  
fill (ForwardIterator beg, ForwardIterator end,  
      const T& newValue)
```

```
Void  
fill_n (OutputIterator beg, Size num,  
        const T& newValue)
```

- fill() 将区间 [beg, end) 内的每一个元素都赋予新值 newValue。
- fill_n() 将 “从 beg 开始的前 num 个元素” 赋予新值 newValue。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- 复杂度：线性，分别进行 numberOfElements 次或 num 次赋值 (assign) 操作。

以下程序展示 fill() 和 fill_n() 的用法：

```
// algo/fill1.cpp  
  
#include "algostuff.hpp"  
using namespace std;  
  
int main()  
{  
    // print ten times 7.7  
    fill_n(ostream_iterator<float>(cout, " "), // beginning of destination  
           10, // count  
           7.7); // new value  
    cout << endl;  
  
    list<string> coll;  
  
    // insert "hello" nine times  
    fill_n(back_inserter(coll), // beginning of destination  
           9, // count  
           "hello"); // new value  
    PRINT_ELEMENTS(coll, "coll: ");  
  
    // overwrite all elements with "again"  
    fill(coll.begin(), coll.end(), // destination  
          "again"); // new value  
    PRINT_ELEMENTS(coll, "coll: ");  
}
```

```

// replace all but two elements with "hi"
fill_n(coll.begin(),          // beginning of destination
        coll.size()-2,        // count
        "hi");                // new value
PRINT_ELEMENTS(coll, "coll: ");

// replace the second and up to the last element but one with "hmmmm"
list<string>::iterator pos1, pos2;
pos1 = coll.begin();
pos2 = coll.end();
fill (++pos1, --pos2,          // destination
      "hmmmm");               // new value
PRINT_ELEMENTS(coll, "coll: ");
}

```

第一次调用动作示范如何使用 `fill_n()` 打印特定数量的值。其它针对 `fill()` 和 `fill_n()` 的调用动作则示范如何在一个 `strings list` 中安插和替换元素。程序输出如下:

```

7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7
coll: hello hello hello hello hello hello hello hello
coll: again again again again again again again again
coll: hi hi hi hi hi hi hi hi again again
coll: hi hmmm hmmm hmmm hmmm hmmm hmmm hmmm again

```

赋予新产生的数值

```

void
generate (ForwardIterator beg, ForwardIterator end,
          Func op)
void
generate_n (OutputIterator beg, Size num,
            Func op)

```

- `generate()` 会调用以下动作:

```
op()
```

产生新值, 并赋值给区间 `[beg, end)` 内的每个元素。

- `generate_n()` 会调用以下动作：
 `op()`
 产生新值，并赋值给“以 `beg` 起始的区间”内的前 `num` 个元素。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- 复杂度：线性，分别进行 `numberOfElements` 次或 `num` 次赋值 (`assign`) 操作。

以下程序展示如何利用 `generate()` 和 `generate_n()` 安插和赋值一些随机数：

```
// algo/generate.cpp

#include <cstdlib>
#include "alghostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert five random numbers
    generate_n (back_inserter(coll), // beginning of destination range
               5,                    // count
               rand); // new value generator
    PRINT_ELEMENTS(coll);

    // overwrite with five new random numbers
    generate (coll.begin(), coll.end(), // destination range
             rand);                    // new value generator
    PRINT_ELEMENTS(coll);
}
```

其中所用的 `rand()` 函数将于 12.3 节, p581 介绍。程序可能输出如下：

```
41 18467 6334 26500 19169
15724 11478 29358 26962 24464
```

实际输出结果视平台而定——因为 `rand()` 产生的随机数序列并无一定标准。

8.1.2 节, p296 有一个例子展示如何将 `generate()` 和一个仿函数 (functor, 另名 function object) 搭配使用，产生一个数列。

9.6.5 替换 (Replacing) 元素

替换序列内的元素

```
void
replace (ForwardIterator beg, ForwardIterator end,
          const T& oldValue, const T& newValue)
void
replace_if (ForwardIterator beg, ForwardIterator end,
             UnaryPredicate op, const T& newValue)
```

- `replace()` 将区间 `[beg, end)` 之内每一个 “与 `oldValue` 相等” 的元素替换为 `newValue`。
- `replace_if()` 将区间 `[beg, end)` 之内每一个令以下一元判断式：
`op(elem)`
获得 `true` 的元素替换为 `newValue`。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 复杂度：线性，执行比较操作（或调用 `op()`）*numberOfElements* 次。

以下程序示范 `replace()` 和 `replace_if()` 的用法：

```
// algo/replace1.cpp

#include "alghostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll, 2, 7);
    INSERT_ELEMENTS(coll, 4, 9);
    PRINT_ELEMENTS(coll, "coll: ");

    // replace all elements with value 6 with 42
    replace (coll.begin(), coll.end(),    // range
            6,                             // old value
            42);                           // new value
    PRINT_ELEMENTS(coll, "coll: ");
```



```

// replace all elements with value less than 5 with 0
replace_if (coll.begin(), coll.end(),      // range
            bind2nd(less<int>(),5),        // criterion for replacement
            0);                            // new value
PRINT_ELEMENTS(coll,"coll: ");
}

```

程序输出如下:

```

coll: 2 3 4 5 6 7 4 5 6 7 8 9
coll: 2 3 4 5 42 7 4 5 42 7 8 9
coll: 0 0 0 5 42 7 0 5 42 7 8 9

```

复制并替换元素

```

OutputIterator
replace_copy (InputIterator sourceBeg, InputIterator sourceEnd,
               OutputIterator destBeg,
               const T& oldValue, const T& newValue)

OutputIterator
replace_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                  OutputIterator destBeg,
                  UnaryPredicate op, const T& newValue)

```

- `replace_copy()` 是 `copy()` 和 `replace()` 的组合。它将源区间 $[beg, end)$ 中的元素复制到“以 `destBeg` 为起点”的目标区间去, 同时将其其中“与 `oldValue` 相等”的所有元素替换为 `newValue`。
- `replace_copy_if()` 是 `copy()` 和 `replace_if()` 的组合。 $[beg, end)$ 中的元素被复制到“以 `destBeg` 为起点”的目标区间, 同时将其其中“令以下一元判断式:
`op(elem)`
结果为 `true`”的所有元素替换为 `newValue`。
- 两个算法都返回目标区间中“最后一个被复制元素”的下一位置, 也就是第一个未被覆盖 (overwritten) 的元素位置。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 复杂度: 线性, 执行比较动作 (或调用 `op()`) *numberOfElements* 次。

以下程序示范如何使用 `replace_copy()` 和 `replace_copy_if()`:

```
// algo/replace2.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    PRINT_ELEMENTS(coll);

    // print all elements with value 5 replaced with 55
    replace_copy(coll.begin(), coll.end(),          // source
                 ostream_iterator<int>(cout, " "), // destination
                 5,                                  // old value
                 55);                                // new value
    cout << endl;

    // print all elements with a value less than 5 replaced with 42
    replace_copy_if(coll.begin(), coll.end(),        // source
                    ostream_iterator<int>(cout, " "), // destination
                    bind2nd(less<int>(),5), // replacement criterion
                    42);                             // new value
    cout << endl;

    // print each element while each odd element is replaced with 0
    replace_copy_if(coll.begin(), coll.end(),        // source
                    ostream_iterator<int>(cout, " "), // destination
                    bind2nd(modulus<int>(),2), // replacement criterion
                    0);                             // new value
    cout << endl;
}
```

程序输出如下:

```
2 3 4 5 6 4 5 6 7 8 9
2 3 4 55 6 4 55 6 7 8 9
42 42 42 5 6 42 5 6 7 8 9
2 0 4 0 6 4 0 6 0 8 0
```

9.7 移除性算法 (Removing Algorithms)

本节所列算法系根据元素值或某一准则，在一个区间内移除某些元素。这些算法并不能改变元素的数量，它们只是以逻辑上的思考，将原本置于后面的“不移除元素”向前移动，覆盖那些被移除元素而已。它们都返回新区间的逻辑终点（也就是最后一个“不移除元素”的下一位置）。细节详见 5.6.1 节, p111。

9.7.1 移除某些特定元素

移除某序列内的元素

```
ForwardIterator  
remove (ForwardIterator beg, ForwardIterator end,  
         const T& value)  
ForwardIterator  
remove_if (ForwardIterator beg, ForwardIterator end,  
            UnaryPredicate op)
```

- `remove()` 会移除区间 `[beg, end)` 中每一个“与 `value` 相等”的元素。
- `remove_if()` 会移除区间 `[beg, end)` 中每一个“令以下一元判断式：
`op(elem)`
获得 `true`”的元素。
- 两个算法都返回变动后的序列的新逻辑终点（也就是最后一个未被移除元素的下一位置）。
- 这些算法会把原本置于后面的未移除元素向前移动，覆盖被移除元素。
- 未被移除的元素在相对次序上保持不变。
- 调用者在调用此算法之后，应保证从此采用返回的新逻辑终点，而不再使用原始终点 `end`（细节参见 5.6.1 节, p111）。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 注意，`remove_if()` 通常会在内部复制它所获得的那个一元判断式，然后两次运用它。如果该一元判断式在函数调用过程中改变状态，就可能导致问题。细节见 8.1.4 节, p302。
- 由于会发生元素变动，所以这些算法不可用于关联式容器（参见 5.6.2 节, p115）。关联式容器提供了功能相似的成员函数 `erase()`（参见 p242）。
- `Lists` 提供了一个等效成员函数 `remove()`：不是重新赋值元素，而是重新安排指针，因此具有更佳性能（参见 p242）。
- 复杂度：线性，执行比较动作（或调用 `op()`）`numberOfElements` 次。

以下程序示范 `remove()` 和 `remove_if()` 的用法:

```
// algo/remove1.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    INSERT_ELEMENTS(coll,1,7);
    PRINT_ELEMENTS(coll,"coll:          ");

    // remove all elements with value 5
    vector<int>::iterator pos;
    pos = remove(coll.begin(), coll.end(),    // range
                 5);                          // value to remove
    PRINT_ELEMENTS(coll,"size not changed: ");

    // erase the "removed" elements in the container
    coll.erase(pos, coll.end());
    PRINT_ELEMENTS(coll,"size changed: ");

    // remove all elements less than 4
    coll.erase(remove_if(coll.begin(), coll.end(), // range
                        bind2nd<less<int>>(),4), // remove criterion
               coll.end());
    PRINT_ELEMENTS(coll,"<4 removed: ");
}
```

程序输出如下:

```
coll:          2 3 4 5 6 4 5 6 7 8 9 1 2 3 4 5 6 7
size not changed: 2 3 4 6 4 6 7 8 9 1 2 3 4 6 7 5 6 7
size changed:    2 3 4 6 4 6 7 8 9 1 2 3 4 6 7
<4 removed:     4 6 4 6 7 8 9 4 6 7
```

复制时一并移除元素

```
OutputIterator  
remove_copy (InputIterator sourceBeg, InputIterator sourceEnd,  
              OutputIterator destBeg,  
              const T& value)  
OutputIterator  
remove_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,  
                 OutputIterator destBeg,  
                 UnaryPredicate op)
```

- `remove_copy()` 是 `copy()` 和 `remove()` 的组合。它将源区间 $[beg, end)$ 内的所有元素复制到“以 `destBeg` 为起点”的目标区间去,并在复制过程中移除“与 `value` 相等”的所有元素。
- `remove_copy_if()` 是 `copy()` 和 `remove_if()` 的组合。它将源区间 $[beg, end)$ 内的元素复制到“以 `destBeg` 为起点”的目标区间去,并在复制过程中移除“造成以下一元判断式:
 `op(elem)`
结果为 `true`”的所有元素。
- 两个算法都返回目标区间中最后一个被复制元素的下一位置 (也就是第一个未被覆盖的元素)。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 调用者必须确保目标区间够大,要不就得使用插入型迭代器。
- 复杂度: 线性, 执行比较动作 (或调用 `op()`) *numberOfElements* 次。

以下程序示范 `remove_copy()` 和 `remove_copy_if()` 的用法:

```
// algo/remove2.cpp  
  
#include "algostuff.hpp"  
using namespace std;  
  
int main()  
{  
    list<int> coll1;  
  
    INSERT_ELEMENTS(coll1,1,6);  
    INSERT_ELEMENTS(coll1,1,9);  
    PRINT_ELEMENTS(coll1);
```

```

// print elements without those having the value 3
remove_copy(coll1.begin(), coll1.end(),          // source
            ostream_iterator<int>(cout, " "),    // destination
            3);                                  // removed value
cout << endl;

// print elements without those having a value greater than 4
remove_copy_if(coll1.begin(), coll1.end(),        // source
               ostream_iterator<int>(cout, " "), // destination
               bind2nd(greater<int>(), 4));       // removed elements
cout << endl;

// copy all elements greater than 3 into a multiset
multiset<int> coll2;
remove_copy_if(coll1.begin(), coll1.end(),        // source
               inserter(coll2, coll2.end()),      // destination
               bind2nd(less<int>(), 4));          // elements not copied
PRINT_ELEMENTS(coll2);
}

```

程序输出如下:

```

1 2 3 4 5 6 1 2 3 4 5 6 7 8 9
1 2 4 5 6 1 2 4 5 6 7 8 9
1 2 3 4 1 2 3 4
4 4 5 5 6 6 7 8 9

```

9.7.2 移除重复元素

移除连续重复元素

ForwardIterator

unique (ForwardIterator beg, ForwardIterator end)

ForwardIterator

unique (ForwardIterator beg, ForwardIterator end,
BinaryPredicate op)

- 以上两种形式都会移除连续重复元素中的多余元素。
- 第一形式将区间 $[beg, end)$ 内所有“与前一元素相等”的元素移除。所以，源序列必须先经过排序，才能使用这个算法移除所有重复元素。

- 第二形式将每一个“位于元素 *e* 之后并且造成以下二元判断式:

`op(elem, e)`

结果为 **true**”的所有 *elem* 元素移除。换言之此一判断式并非用来将元素和其原本的前一元素比较, 而是将它和未被移除的前一元素比较, 参见以下实例 (译注: 换言之, 如果序列 {A,B,C,D,E}, A 不符合移除条件, B 符合, 轮到 C 时, C 将被拿来和 A 比较, 而不是和原本的前一个元素 (但已被移除的) B 比较)

- 两种形式都返回被变动后的序列新终点 (逻辑终点, 也就是最后一个“未被移除的元素”的下一个位置)。
- 这两个算法将“原本位置在后”的未移除元素向前移动, 覆盖 (overwrite) 被移除元素。
- 未被移除的元素在相对次序上保持不变。
- 调用者在调用这些算法之后, 应保证从此使用返回的新逻辑终点, 不再使用原始终点 *end* (详见 5.6.1 节, p111)。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 由于会造成元素变动, 所以这些算法不可用于关联式容器 (见 5.6.2 节, p115)。
- `Lists` 提供了一个等效成员函数 `unique()`, 不是重新赋值元素, 而是重新安排指针, 因此具有更佳性能 (参见 p244)。
- 复杂度: 线性, 执行比较操作 (或调用 `op()`) *numberOfElements* 次。

以下程序示范 `unique()` 的用法:

```
// algo/unique1.cpp

#include "alghostuff.hpp"
using namespace std;

int main()
{
    // source data
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                    5, 4, 4 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    list<int> coll;

    // initialize coll with elements from source
    copy (source, source+sourceNum, // source
          back_inserter(coll));    // destination
    PRINT_ELEMENTS(coll);

    // remove consecutive duplicates
    list<int>::iterator pos;
```

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

/* print elements not removed
 * - use new logical end
 */
copy (coll.begin(), pos, // source
      ostream_iterator<int>(cout, " ")); // destination
cout << "\n\n";

// reinitialize coll with elements from source
copy (source, source+sourceNum, // source
      coll.begin());           // destination
PRINT_ELEMENTS(coll);

// remove elements if there was a previous greater element
coll.erase (unique (coll.begin(), coll.end(),
                    greater<int>()),
            coll.end());
PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4

1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 4 6 6 6 6 6 7
```

第一次调用 `unique()` 是为了移除连续重复元素。第二次调用则示范 `unique()` 第二形式的行为: 将“`greater` 比较结果为 `true`”的所有元素移除。例如第一个元素值是 6, 大于其后的 1, 2, 2, 3 和 1, 所以后面这些元素都被移除。换言之, 该判断式不是用来将元素和其直接前趋元素比较, 而是将它和未被移除的前趋元素比较 (稍后的 `unique_copy()` 另有一个类似例子)。

复制过程中移除重复元素

OutputIterator

```
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,  
             OutputIterator destBeg)
```

OutputIterator

```
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,  
             OutputIterator destBeg,  
             BinaryPredicate op)
```

- 两种形式都是 `copy()` 和 `unique()` 的组合。
- 两者都将源区间 `[sourceBeg, sourceEnd)` 内的元素复制到“以 `destBeg` 起始的目标区间”，并移除重复元素。
- 两个算法都返回目标区间内“最后一个被复制的元素”的下一位置（也就是第一个未被覆盖的元素）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 复杂度：线性，执行比较动作（或调用 `op()`）*numberOfElements* 次。

以下程序示范 `unique_copy()` 的用法：

```
// algo/unique2.cpp

#include "algostuff.hpp"
using namespace std;

bool differenceOne (int elem1, int elem2)
{
    return elem1 + 1 == elem2 || elem1 - 1 == elem2;
}

int main()
{
    // source data
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                    5, 4, 4 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    // initialize coll with elements from source
    list<int> coll;
    copy(source, source+sourceNum, // source
         back_inserter(coll));     // destination
    PRINT_ELEMENTS(coll);
```

```

    // print element with consecutive duplicates removed
    unique_copy(coll.begin(), coll.end(),           // source
                ostream_iterator<int>(cout, " ")); // destination
    cout << endl;

    // print element without consecutive duplicates that
    // differ by one
    unique_copy(coll.begin(), coll.end(),           // source
                ostream_iterator<int>(cout, " "),   // destination
                differenceOne);                     // duplicates criterion
    cout << endl;
}

```

程序输出如下:

```

1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4
1 4 4 6 1 3 1 6 6 6 4 4

```

注意, 第二次调用 `unique_copy()` 后并未移除“与其直接前趋元素相差 1”的所有元素, 而是移除“与其未被移除之前趋元素相差 1”的所有元素。例如, 出现于三个 6 之后, 紧跟着的元素是 5, 7, 5, 都与 6 相差 1, 所以被移除。然而更后面的两个 4, 并非和 6 相差 1, 所以被保留下来。

另有一个用来压缩空白序列的例子:

```

// algo/unique3.cpp

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

bool bothSpaces (char elem1, char elem2)
{
    return elem1 == ' ' && elem2 == ' ';
}

int main()
{
    // don't skip leading whitespaces by default
    cin.unsetf(ios::skipws);

    /* copy standard input to standard output
     * - while compressing spaces
     */
}

```

```

        unique_copy(istream_iterator<char>(cin),    // beginning of source: cin
                    istream_iterator<char>(),        // end of source: end-of-file
                    ostream_iterator<char>(cout),    // destination: cout
                    bothSpaces);                     // duplicate criterion
    }

```

当输入如下:

```
Hello, here are  sometimes more  and sometimes fewer  spaces.
```

输出结果是:

```
Hello, here are sometimes more and sometimes fewer spaces.
```

9.8 变序性算法 (Mutating Algorithms)

译注: 某些书籍如《*Generic Programming and the STL*》和《*STL 源码剖析*》中, mutating algorithm 是指变动性 (更易性) 算法, 也就是本书的 modifying algorithm。本书划分更为细腻。至于我将 mutating algorithm 译为变序性算法, 是着眼于其实际效应。C++ 关键词 mutable 的意义意思是“即使在 const object 内仍可变动的”。

变序性算法改变元素的次序, 但不改变元素值。这些算法不能用于关联式容器, 因为在关联式容器中, 元素有一定的次序, 不能随意变动。

9.8.1 逆转 (Reversing) 元素次序

```

void
reverse (BidirectionalIterator beg, BidirectionalIterator end)

OutputIterator
reverse_copy (BidirectionalIterator sourceBeg,
              BidirectionalIterator sourceEnd,
              OutputIterator destBeg)

```

- reverse() 会将区间 [beg, end) 内的元素全部逆序。
- reverse_copy() 会将源区间 [sourceBeg, sourceEnd) 内的元素复制到 “以 destBeg 起始的目标区间”, 并在复制过程中颠倒安置次序。
- reverse_copy() 返回目标区间内最后一个被复制元素的下一位置, 也就是第一个未被覆盖 (overwritten) 的元素。
- 调用者必须确保目标区间够大, 要不就得使用插入型迭代器。
- Lists 提供了一个等效成员函数 reverse(), 不是重新赋值元素, 而是重新安排指针, 因此具有更佳性能 (参见 p246)。
- 复杂度: 线性, 分别进行 $numberOfElements / 2$ 次交换操作或 $numberOfElements$ 次赋值 (assign) 操作。

下面这个程序展示 `reverse()` 和 `reverse_copy()` 的用法:

```
// algo/reverse1.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order of elements
    reverse (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order from second to last element but one
    reverse (coll.begin()+1, coll.end()-1);
    PRINT_ELEMENTS(coll,"coll: ");

    // print all of them in reverse order
    reverse_copy (coll.begin(), coll.end(),           // source
                  ostream_iterator<int>(cout," ")    // destination
    );
    cout << endl;
}
```

程序输出如下:

```
coll: 1 2 3 4 5 6 7 8 9
coll: 9 8 7 6 5 4 3 2 1
coll: 9 2 3 4 5 6 7 8 1
1 8 7 6 5 4 3 2 9
```

9.8.2 旋转 (Rotating) 元素次序

旋转序列内的元素

```
void  
rotate (ForwardIterator beg, ForwardIterator newBeg,  
         ForwardIterator end)
```

- 将区间 $[beg, end)$ 内的元素进行旋转，执行后 $*newBeg$ 成为新的第一元素。
- 调用者必须确保 $newBeg$ 是区间 $[beg, end)$ 内的一个有效位置，否则会引发未定义的行为。
- 复杂度：线性，最多进行 *numberOfElements* 次交换动作。

以下程序示范如何使用 `rotate()`：

```
// algo/rotate1.cpp  
  
#include "algostuff.hpp"  
using namespace std;  
  
int main()  
{  
    vector<int> coll;  
  
    INSERT_ELEMENTS(coll,1,9);  
    PRINT_ELEMENTS(coll,"coll: ");  
  
    // rotate one element to the left  
    rotate (coll.begin(),           // beginning of range  
            coll.begin() + 1,      // new first element  
            coll.end());           // end of range  
    PRINT_ELEMENTS(coll,"one left: ");  
  
    // rotate two elements to the right  
    rotate (coll.begin(),           // beginning of range  
            coll.end() - 2,         // new first element  
            coll.end());           // end of range  
    PRINT_ELEMENTS(coll,"two right: ");  
  
    // rotate so that element with value 4 is the beginning  
    rotate (coll.begin(),           // beginning of range  
            find(coll.begin(),coll.end(),4), // new first element  
            coll.end());           // end of range  
    PRINT_ELEMENTS(coll,"4 first: ");  
}
```

正如上例所示, 你可以使用正偏移量 (positive offset) 将元素向左起点方向旋转, 也可以使用负偏移量 (negative offset) 将元素向右终点方向旋转。不过请注意, 只有在随机存取迭代器身上才能为它加上偏移量。如果不是这类迭代器, 你就得使用 `advance()` (参见 p389, `rotate_copy()` 的例子)。

程序输出如下:

```
coll: 1 2 3 4 5 6 7 8 9
one left: 2 3 4 5 6 7 8 9 1
two right: 9 1 2 3 4 5 6 7 8
4 first: 4 5 6 7 8 9 1 2 3
```

复制并同时旋转元素

`OutputIterator`

```
rotate_copy (ForwardIterator sourceBeg, ForwardIterator newBeg,
             ForwardIterator sourceEnd,
             OutputIterator destBeg)
```

- 这是 `copy()` 和 `rotate()` 的组合。
- 将源区间 `[sourceBeg, sourceEnd)` 内的元素复制到“以 `destBeg` 起始的目标区间”中, 同时旋转元素, 使 `newBeg` 成为新的第一元素。
- 返回目标区间内最后一个被复制元素的下一位置。
- 调用者必须确保 `newBeg` 是区间 `[beg, end)` 内的一个有效位置, 否则会引发未定义的行为。
- 调用者必须确保目标区间够大, 要不就得使用插入型迭代器。
- 源区间和目标区间两者不可重迭。
- 复杂度: 线性, 执行 `numberOfElements` 次赋值 (*assign*) 操作。

以下程序示范 `rotate_copy()` 的用法:

```
// algo/rotate2.cpp

#include "alghostuff.hpp"
using namespace std;

int main()
{
    set<int> coll;

    INSERT_ELEMENTS(coll, 1, 9);
    PRINT_ELEMENTS(coll);
```

```
// print elements rotated one element to the left
set<int>::iterator pos = coll.begin();
advance(pos,1);
rotate_copy(coll.begin(),    // beginning of source
            pos,              // new first element
            coll.end(),       // end of source
            ostream_iterator<int>(cout," ")); // destination
cout << endl;

// print elements rotated two elements to the right
pos = coll.end();
advance(pos,-2);
rotate_copy(coll.begin(),    // beginning of source
            pos,              // new first element
            coll.end(),       // end of source
            ostream_iterator<int>(cout," ")); // destination
cout << endl;

// print elements rotated so that element with value 4
// is the beginning
rotate_copy(coll.begin(),    // beginning of source
            coll.find(4),     // new first element
            coll.end(),       // end of source
            ostream_iterator<int>(cout," ")); // destination
cout << endl;
}
```

与先前的 `rotate()` 实例不同(参见 p388), 这里用了一个 `set` 而不是一个 `vector`。这导致两个结果:

1. 你必须使用 `advance()` (参见 7.3.1 节, p259) 来改变迭代器本身的值, 因为双向迭代器不支持 `operator+`。
2. 你应当使用 `find()` 成员函数, 而非 `find()` 算法, 因为前者有更好的效能。

程序输出如下:

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
8 9 1 2 3 4 5 6 7
4 5 6 7 8 9 1 2 3
```

9.8.3 排列 (Permuting) 元素

```
bool
next_permutation (BidirectionalIterator beg,
                  BidirectionalIterator end)

bool
prev_permutation (BidirectionalIterator beg,
                  BidirectionalIterator end)
```

- `next_permutation()` 会改变区间 $[beg, end)$ 内的元素次序，使它们符合“下一个排列次序”。
- `prev_permutation()` 会改变区间 $[beg, end)$ 内的元素次序，使它们符合“上一个排列次序”。
- 如果元素得以排列成〈就字典顺序而言的〉“正规 (normal)”次序，则两个算法都返回 `true`。所谓正规次序，对 `next_permutation()` 而言为升序，对 `prev_permutation()` 而言为降序。因此，如果要走遍所有排列，你必须先将所有元素（按升序或降序）排序，然后开始以循环方式调用 `next_permutation` 或 `prev_permutation`，直到算法返回 `false`⁵。
- 所谓“字典次序”的排序，p360 有解释。
- 复杂度：线性，最多执行 $numberOfElements / 2$ 次交换操作。

下面这个例子展示利用 `next_permutation()` 和 `prev_permutation()` 获得所有元素的所有可能排列的过程：

```
// algo/perm1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,3);
    PRINT_ELEMENTS(coll,"on entry: ");

    /* permute elements until they are sorted
     * - runs through all permutations because the elements are sorted now
     */
```

⁵ `next_permutation()` 和 `prev_permutation()` 也可用来在区间内对元素排序。你只需一再调用它们直到它们传回 `false`。然而这么做的效率很低。


```
while (next_permutation(coll.begin(),coll.end())) {
    PRINT_ELEMENTS(coll," ");
}

PRINT_ELEMENTS(coll,"afterward: ");

/* permute until descending sorted
 * - this is the next permutation after ascending sorting
 * - so the loop ends immediately
 */
while (prev_permutation(coll.begin(),coll.end())) {
    PRINT_ELEMENTS(coll," ");
}

PRINT_ELEMENTS(coll,"now: ");

/* permute elements until they are sorted in descending order
 * - runs through all permutations because the elements are sorted
 * in descending order now
 */
while (prev_permutation(coll.begin(),coll.end())) {
    PRINT_ELEMENTS(coll," ");
}

PRINT_ELEMENTS(coll,"afterward: ");
}
```

程序输出如下:

```
on entry: 1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
afterward: 1 2 3
now:      3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
afterward: 3 2 1
```

9.8.4 重排元素 (Shuffling, 搅乱次序)

```
void  
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)  
void  
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,  
                RandomFunc& op)
```

- 第一形式使用一个均匀分布随机数产生器 (uniform distribution random number generator) 来打乱区间 $[beg, end)$ 内的元素次序。
- 第二形式使用 op 打乱区间 $[beg, end)$ 内的元素次序。算法内部会使用一个整数值 (其型别为“迭代器所提供之 `difference_type`”) 来调用 op :

$op(max)$

它应该返回一个大于零而小于 max 的随机数, 不包括 max 本身。

- 注意, op 是一个 non-const reference。所以你不可以将暂时数值或一般函数传进去。
- 复杂度: 线性, 执行 $numberOfElements - 1$ 次交换动作。

你大概会问, 为什么 `random_shuffle()` 那个可有可无的参数 (代表一个操作) 是个 non-const reference 呢? 必须如此, 因为典型的随机数产生器拥有一个局部状态 (local state)。旧式 C 函数如 `rand()` 是将其局部状态存储在某个静态变量中。但是这有一些缺点, 例如这种随机数发生器本质上对于多线程 (multi-threads) 而言就不安全, 而且你也不可能拥有两个各自独立的随机数流 (streams)。如果使用仿函数, 其区域状态被封装为一个或多个成员变量, 那么就有了比较好的解决方案。这样一来, 随机数产生器就不可能具备常数性, 否则何以改变内部状态, 何以产生新的随机数呢? 不过你还是可以用 by value 方式传递随机数产生器, 为什么非要 with by non-const reference 方式传递呢? 是这样的, 如果这么做, 每次调用都会在内部复制一个随机数产生器及其状态, 结果, 每次你传入随机数产生器, 所得的随机数序列都一样, 那又有何随机可言? 现在你明白为什么要以 by non-const reference 方式传递了吧⁶。

如果你需要获得同一个随机数序列两次, 复制它就是了。但如果那个随机数产生器的实作手法涉及全局状态 (global state), 你还是会获得不同的序列。

以下程序示范如何调用 `random_shuffle()` 来打乱元素次序:

```
// algo/random1.cpp  
  
#include <cstdlib>
```

⁶ 感谢 Matt Austern 就这个问题所提供的解释。

```
#include "algostuff.hpp"
using namespace std;

class MyRandom {
public:
    ptrdiff_t operator() (ptrdiff_t max) {
        double tmp;
        tmp = static_cast<double>(rand())
            / static_cast<double>(RAND_MAX);
        return static_cast<ptrdiff_t>(tmp * max);
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // shuffle all elements randomly
    random_shuffle (coll.begin(), coll.end());

    PRINT_ELEMENTS(coll,"shuffled: ");

    // sort them again
    sort (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"sorted: ");

    /* shuffle elements with self-written random number generator
     * - to pass an lvalue we have to use a temporary object
     */
    MyRandom rd;
    random_shuffle (coll.begin(), coll.end(), // range
                   rd);                      // random number generator
    PRINT_ELEMENTS(coll,"shuffled: ");
}
```

第二次调用 `random()` 时用了一个自定义的随机数产生器 `rd()`。它是根据辅助仿函数 `MyRandom` 而产生的一个对象，采用一个随机数计算法。它的效果通常比直接调用 `rand()` 好一些⁷。

⁷ `MyRandom` 的随机数产生法在 Bjarne Stroustrup 的《*The C++ Programming Language*》，3/e 中有介绍和解释。

一个可能（但非必然）的输出如下：

```
coll:      1 2 3 4 5 6 7 8 9
shuffled:  2 6 9 5 4 3 1 7 8
sorted:    1 2 3 4 5 6 7 8 9
shuffled:  2 6 9 3 1 8 7 4 5
```

9.8.5 将元素向前搬移

```
BidirectionalIterator
partition (BidirectionalIterator beg,
          BidirectionalIterator end,
          UnaryPredicate op)
```

```
BidirectionalIterator
stable_partition (BidirectionalIterator beg,
                 BidirectionalIterator end,
                 UnaryPredicate op)
```

- 这两种算法将区间 $[beg, end)$ 中“造成以下一元判断式：
 $op(elem)$
 结果为 true”的元素向前端移动。
- 这两种算法都返回“令 $op()$ 结果为 false”的第一个元素位置。
- 两者差别是，无论元素是否符合给定的准则，`stable_partition()` 会保持它们之间的相对次序。
- 你可以运用此算法，根据排序准则，将所有元素分割为两部分。`nth_element()` 具有类似能力。至于本算法和 `nth_element()` 之间的区别，参见 p330。
- op 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- 复杂度：

— `partition()`：线性，总共执行 $op()$ 操作 *numberOfElements* 次，以及最多 *numberOfElements* / 2 次的交换操作。

— `stable_partition()`：如果系统拥有足够的内存，那么就是线性复杂度，执行 $op()$ 操作及交换操作共 *numberOfElements* 次；如果没有足够内存，则是 $n \log n$ ，执行 $op()$ 操作 *numberOfElements* * $\log(\text{numberOfElements})$ 次。

以下程序示范 `partition()` 和 `stable_partition()` 的用法以及两者的区别：

```
// algo/part1.cpp

#include "alghostuff.hpp"
using namespace std;
```

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

    INSERT_ELEMENTS(coll1,1,9);
    INSERT_ELEMENTS(coll2,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");
    cout << endl;

    // move all even elements to the front
    vector<int>::iterator pos1, pos2;
    pos1 = partition(coll1.begin(), coll1.end(),          // range
                    not1(bind2nd(modulus<int>(),2)));    // criterion
    pos2 = stable_partition(coll2.begin(), coll2.end(),   // range
                           not1(bind2nd(modulus<int>(),2))); // criterion

    // print collections and first odd element
    PRINT_ELEMENTS(coll1,"coll1: ");
    cout << "first odd element: " << *pos1 << endl;
    PRINT_ELEMENTS(coll2,"coll2: ");
    cout << "first odd element: " << *pos2 << endl;
}
```

程序输出如下:

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 1 2 3 4 5 6 7 8 9

coll1: 8 2 6 4 5 3 7 1 9
first odd element: 5
coll2: 2 4 6 8 1 3 5 7 9
first odd element: 1
```

正如此例所展示的, `stable_partition()` 保持了奇数元素和偶数元素的相对次序, 这一点和 `partition()` 不同。

9.9 排序算法 (Sorting Algorithms)

STL 提供了好几种算法来对区间内的元素排序。除了完全排序 (full sorting) 外, 还支持局部排序 (partial sorting) 的数个变体。如果这些变体的功能对你已经足够, 你应该优先使用它们, 因为通常它们的性能更佳。

你也可以使用关联式容器, 让元素自动排序。然而请注意, 对全体元素进行一次性排序, 通常比始终维护它们保持已序 (*sorted*) 状态来得高效一些 (细节参见 p228)。

9.9.1 对所有元素排序

```
void
sort (RandomAccessIterator beg, RandomAccessIterator end)

void
sort (RandomAccessIterator beg, RandomAccessIterator end,
      BinaryPredicate op)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end,
             BinaryPredicate op)
```

- `sort()` 和 `stable_sort()` 的上述第一形式, 使用 `operator<` 对区间 `[beg, end)` 内的所有元素进行排序。
- `sort()` 和 `stable_sort()` 的上述第二形式, 使用二元判断式 `op(elem1, elem2)` 作为排序准则, 对区间 `[beg, end)` 内的所有元素进行排序。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- `sort()` 和 `stable_sort()` 的区别是, 后者保证相等元素的原本相对次序在排序后保持不变。
- 不可以对 `lists` 调用这些算法, 因为 `lists` 不支持随机存取迭代器。不过 `lists` 提供了一个成员函数 `sort()`, 可用来对其自身元素排序, 参见 p245。
- `sort()` 保证很不错的平均效能 $n \log n$ 。然而如果你必须极力避免可能出现的最差状况, 你应该使用 `partial_sort()` 或 `stable_sort()`。参见 p328 对于排序算法的讨论。
- 复杂度:
 - `sort()`: 平均 $n \log n$ (平均大约执行比较操作共 $\text{numberOfElements} * \log(\text{numberOfElements})$ 次)。

- `stable_sort()`: 如果系统拥有足够内存, 那么就是 $n \log n$, 也就是执行比较操作 $\text{numberOfElements} * \log(\text{numberOfElements})$ 次; 如果没有足够内存, 则复杂度是 $n \log n * \log n$, 亦即执行比较操作 $\text{numberOfElements} * \log(\text{numberOfElements})^2$ 次。

下面这个例子示范 `sort()` 的用法:

```
// algo/sort1.cpp

#include "algotstuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"on entry: ");

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

    PRINT_ELEMENTS(coll,"sorted: ");

    // sorted reverse
    sort (coll.begin(), coll.end(), // range
          greater<int>());          // sorting criterion

    PRINT_ELEMENTS(coll,"sorted >: ");
}
```

程序输出如下:

```
on entry: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
sorted: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
sorted >: 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
```

至于如何根据某个 `class member` 进行排序, 请见 p123 实例。

以下程序示范 `sort()` 和 `stable_sort()` 两者间的区别。该程序通过排序准则 `lessLength()`, 将字符串按照字符数量排序:

```
// algo/sort2.cpp

#include "alghostuff.hpp"
using namespace std;

bool lessLength (const string& s1, const string& s2)
{
    return s1.length() < s2.length();
}

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

    // fill both collections with the same elements
    coll1.push_back ("1xxx");
    coll1.push_back ("2x");
    coll1.push_back ("3x");
    coll1.push_back ("4x");
    coll1.push_back ("5xx");
    coll1.push_back ("6xxxx");
    coll1.push_back ("7xx");
    coll1.push_back ("8xxx");
    coll1.push_back ("9xx");
    coll1.push_back ("10xxx");
    coll1.push_back ("11");
    coll1.push_back ("12");
    coll1.push_back ("13");
    coll1.push_back ("14xx");
    coll1.push_back ("15");
    coll1.push_back ("16");
    coll1.push_back ("17");
    coll2 = coll1;

    PRINT_ELEMENTS(coll1, "on entry:\n ");

    // sort (according to the length of the strings)
    sort (coll1.begin(), coll1.end(),      // range
          lessLength);                     // criterion

    stable_sort (coll2.begin(), coll2.end(), // range
                 lessLength);               // criterion
    PRINT_ELEMENTS(coll1, "\nwith sort():\n ");
    PRINT_ELEMENTS(coll2, "\nwith stable_sort():\n ");
}
```


程序输出如下:

```
on entry:
1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17

with sort():
17 2x 3x 4x 16 15 13 12 11 9xx 7xx 5xx 8xxx 14xx 1xxx 10xxx 6xxxx

with stable_sort():
2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx
```

只有 `stable_sort()` 保持了元素的相对位置(每个字符串最前头的阿拉伯数字标识出原本的元素顺序)。

9.9.2 局部排序 (Partial Sorting)

```
void
partial_sort (RandomAccessIterator beg,
              RandomAccessIterator sortEnd,
              RandomAccessIterator end)

void
partial_sort (RandomAccessIterator beg,
              RandomAccessIterator sortEnd,
              RandomAccessIterator end,
              BinaryPredicate op)
```

- 以上第一形式, 以 `operator<` 对区间 $[beg, end)$ 内的元素进行排序, 使区间 $[beg, sortEnd)$ 内的元素处于有序状态 (*sorted order*)。
- 以上第二形式, 运用二元判断式:
`op(elem1, elem2)`
 对区间 $[beg, end)$ 内的元素进行排序, 使区间 $[beg, sortEnd)$ 内的元素处于有序状态 (*sorted order*)。
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 和 `sort()` 不同的是, `partial_sort()` 并不对全部元素排序: 一旦第一个元素至 `sortEnd` 之间的所有元素都排妥次序, 就立刻停止。所以如果你只需要前 3 个已序元素, 可以使用 `partial_sort()` 来节省时间, 因为它不会对剩余的元素进行非必要的排序。

- 如果 `sortEnd` 和 `end` 相等, 那么 `partial_sort()` 会对整个序列进行排序。平均而言其效率不及 `sort()`, 不过以最差情况而论则优于 `sort()`。请参考 p328 关于排序算法的讨论。
- 复杂度: 在线性和 $n \log n$ 之间, 大约执行 $\text{numberOfElements} * \log(\text{numberOfSortedElements})$ 次比较操作。

以下程序示范 `partial_sort()` 的用法:

```
// algo/psort1.cpp

#include "alghostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,1,5);
    PRINT_ELEMENTS(coll);

    // sort until the first five elements are sorted
    partial_sort (coll.begin(),           // beginning of the range
                  coll.begin()+5,         // end of sorted range
                  coll.end());            // end of full range
    PRINT_ELEMENTS(coll);

    // sort inversely until the first five elements are sorted
    partial_sort (coll.begin(),           // beginning of the range
                  coll.begin()+5,         // end of sorted range
                  coll.end(),             // end of full range
                  greater<int>());        // sorting criterion
    PRINT_ELEMENTS(coll);

    // sort all elements
    partial_sort (coll.begin(),           // beginning of the range
                  coll.end(),             // end of sorted range
                  coll.end());            // end of full range
    PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 7 6 5 5 6 4 4 3 4 5
7 6 6 5 5 1 2 2 3 3 4 4 3 4 5
1 2 2 3 3 3 4 4 4 5 5 5 6 6 7
```

```
RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg,
                  InputIterator sourceEnd,
                  RandomAccessIterator destBeg,
                  RandomAccessIterator destEnd)
```

```
RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg,
                  InputIterator sourceEnd,
                  RandomAccessIterator destBeg,
                  RandomAccessIterator destEnd,
                  BinaryPredicate op)
```

- 两者都是 `copy()` 和 `partial_sort()` 的组合。
- 它们将元素从源区间 $[sourceBeg, sourceEnd)$ 复制到目标区间 $[destBeg, destEnd)$, 同时进行排序。
- “被排序 (被复制) 的元素数量” 是源区间和目标区间两者所含元素数量的较小值。
- 两者都返回目标区间内 “最后一个被复制元素” 的下一位置 (也就是第一个未被覆盖的元素)。
- 如果目标区间 $[destBeg, destEnd)$ 内的元素数量大于或等于源区间 $[sourceBeg, sourceEnd)$ 内的元素数量, 则所有元素都会被排序并复制, 整个行为就相当于 `copy()` 和 `sort()` 的组合。
- 复杂度: 在线性和 $n \log n$ 之间, 大约执行 $numberOfElements * \log(numberOfSortedElements)$ 次比较操作。

以下程序示范 `partial_sort_copy()` 的用法:

```
// algo/psort2.cpp

#include "algorithstuff.hpp"
using namespace std;

int main()
{
    deque<int> coll1;
    vector<int> coll6(6);    // initialize with 6 elements
    vector<int> coll30(30);  // initialize with 30 elements
```

```
INSERT_ELEMENTS(coll1,3,7);
INSERT_ELEMENTS(coll1,2,6);
INSERT_ELEMENTS(coll1,1,5);
PRINT_ELEMENTS(coll1);

// copy elements of coll1 sorted into coll6
vector<int>::iterator pos6;
pos6 = partial_sort_copy (coll1.begin(), coll1.end(),
                          coll6.begin(), coll6.end());

// print all copied elements
copy (coll6.begin(), pos6,
      ostream_iterator<int>(cout," "));
cout << endl;

// copy elements of coll1 sorted into coll30
vector<int>::iterator pos30;
pos30 = partial_sort_copy (coll1.begin(), coll1.end(),
                          coll30.begin(), coll30.end(),
                          greater<int>());

// print all copied elements
copy (coll30.begin(), pos30,
      ostream_iterator<int>(cout," "));
cout << endl;
}
```

程序输出如下:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 3
7 6 6 5 5 5 4 4 4 3 3 3 2 2 1
```

第一次调用 `partial_sort_copy()` 时, 目标区间内只有 6 个元素, 所以该算法只复制了 6 个元素, 返回 `coll6` 的终点。第二次调用 `partial_sort_copy()` 时, 由于 `coll30` 有充足的空间, 所以 `coll1` 的所有元素都被复制并排序。

9.9.3 根据第 n 个元素排序

```
void
nth_element (RandomAccessIterator beg,
             RandomAccessIterator nth,
             RandomAccessIterator end)

void
nth_element (RandomAccessIterator beg,
             RandomAccessIterator nth,
             RandomAccessIterator end,
             BinaryPredicate op)
```

- 两种形式都对区间 (beg, end) 内的元素进行排序，使第 n 个位置上的元素就位，也就是说，所有在位置 n 之前的元素都小于等于它，所有在位置 n 之后的元素都大于等于它。这样，你就得到了“根据 n 位置上的元素”分割开来的两个子序列，第一子序列的元素统统小于第二子序列的元素。如果你只需要 n 个最大或最小元素，但不要求它们必须已序 (*sorted*)，那么这个算法就很有用。
- 上述第一形式使用 `operator<` 作为排序准则。
- 上述第二形式使用以下二元判断式作为排序准则：
`op(elem1, elem2)`
- `op` 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- `partition()` 算法 (参见 p395) 也可以根据某个排序准则，将序列中的元素分割成两部分。至于 `partition()` 和 `nth_element()` 间的区别，请参考 p330。
- 复杂度：平均而言为线性。

以下程序示范 `nth_element()` 的用法：

```
// algo/nth1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll, 3, 7);
    INSERT_ELEMENTS(coll, 2, 6);
    INSERT_ELEMENTS(coll, 1, 5);
    PRINT_ELEMENTS(coll);
```

```

// extract the four lowest elements
nth_element (coll.begin(),    // beginning of range
             coll.begin()+3,  // element that should be sorted correctly
             coll.end());     // end of range

// print them
cout << "the four lowest elements are: ";
copy (coll.begin(), coll.begin()+4,
      ostream_iterator<int>(cout, " "));
cout << endl;

// extract the four highest elements
nth_element (coll.begin(),    // beginning of range
             coll.end()-4,    // element that should be sorted correctly
             coll.end());     // end of range

// print them
cout << "the four highest elements are: ";
copy (coll.end()-4, coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;

// extract the four highest elements (second version)
nth_element (coll.begin(),    // beginning of range
             coll.begin()+3,  // element that should be sorted correctly
             coll.end(),      // end of range
             greater<int>()); // sorting criterion

// print them
cout << "the four highest elements are: ";
copy (coll.begin(), coll.begin()+4,
      ostream_iterator<int>(cout, " "));
cout << endl;
}

```

程序输出如下:

```

3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
the four lowest elements are: 2 1 2 3
the four highest elements are: 5 6 7 6
the four highest elements are: 6 7 6 5

```

9.9.4 Heap 算法

就排序而言, **heap** 是一种特别的元素组织方式, 应用于 **heap** 排序法 (**heapsort**)。 **heap** 可被视为一个以序列式群集 (**sequential collection**) 实作而成的二叉树, 具有两大性质 (译注: 细节可参考《STL 源码剖析》4.7 节):

1. 第一个元素总是最大。
2. 总是能够在对数时间内增加或移除一个元素。

heap 是实作 **priority queue** (其内元素会自动排序) 的一个理想结构。因此, **heap** 算法也在 **priority_queue** 容器中有所应用 (参见 10.3 节, p453)。为了处理 **heap**, STL 提供四种算法:

1. **make_heap()** 将某区间内的元素转化成 **heap**。
2. **push_heap()** 对着 **heap** 增加一个元素。
3. **pop_heap()** 对着 **heap** 取出下一个元素。
4. **sort_heap()** 将 **heap** 转化为一个已序群集 (此后它就不再是 **heap** 了)。

就像以前常见的情况一样, 你可以传递一个二元判断式作为排序准则。缺省的排序准则是 **operator<**。

heap 算法细节

```
void  
make_heap (RandomAccessIterator beg,  
            RandomAccessIterator end)
```

```
void  
make_heap (RandomAccessIterator beg,  
            RandomAccessIterator end,  
            BinaryPredicate op)
```

- 两种形式都将区间 $[beg, end)$ 内的元素转化为 **heap**。
- **op** 是一个可有可无的 (可选的) 二元判断式, 被视为排序准则:

op(*elem1*, *elem2*)

- 只有在多于一个元素的情况下, 才有必要使用这些函数来处理 **heap**, 如果只有单一元素, 那么它自动就形成一个 **heap**。
- 复杂度: 线性, 最多执行 $3 * \text{numberOfElements}$ 次比较动作。

```
void  
push_heap (RandomAccessIterator beg, RandomAccessIterator end)
```

```
void  
push_heap (RandomAccessIterator beg, RandomAccessIterator end,  
            BinaryPredicate op)
```

- 两种形式都将 **end** 之前的最后一个元素加入原本就是个 **heap** 的 $[beg, end-1)$ 区间内, 使整个区间 $[beg, end)$ 成为一个 **heap**。

- `op` 是一个可有可无的 (可选的) 二元判断式, 被视为排序准则:
`op(elem1, elem2)`
- 调用者必须保证, 进入函数时, 区间 `[beg, end-1)` 内的元素原本便已形成一个 `heap` (在相同的排序准则下), 而新元素紧跟其后。
- 复杂度: 对数, 最多执行 $\log(\text{numberOfElements})$ 次比较操作。

```

Void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end)

Void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end,
          BinaryPredicate op)

```

- 以上两种形式都将 `heap[beg, end)` 内的最高元素, 也就是第一个元素, 移到最后位置, 并将剩余区间 `[beg, end-1)` 内的元素组织起来, 成为一个新的 `heap`。
- `op` 是个可有可无的 (可选的) 二元判断式, 被当做排序准则:
`op(elem1, elem2)`
- 调用者必须保证, 进入函数时, 区间 `[beg, end)` 内的元素原本便已形成一个 `heap` (在相同的排序准则下)。
- 复杂度: 对数, 最多执行 $2 * \log(\text{numberOfElements})$ 次比较操作。

```

void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end)

void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end,
          BinaryPredicate op)

```

- 以上两种形式都可以将 `heap[beg, end)` 转换为一个已序 (*sorted*) 序列。
- `op` 是个可有可无的二元判断式, 被视为排序准则:
`op(elem1, elem2)`
- 注意, 此算法一旦结束, 该区间就不再是个 `heap` 了。
- 调用者必须保证, 进入函数时, 区间 `[beg, end)` 内的元素原本便已形成一个 `heap` (在相同的排序准则下)。
- 复杂度: $n \log n$, 最多执行 $\text{numberOfElements} * \log(\text{numberOfElements})$ 次比较动作。

heap 算法使用范例

以下程序示范如何使用各种 `heap` 算法:


```
// algo/heap1.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,5,9);
    INSERT_ELEMENTS(coll,1,4);

    PRINT_ELEMENTS (coll, "on entry: ");

    // convert collection into a heap
    make_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after make_heap(): ");

    // pop next element out of the heap
    pop_heap (coll.begin(), coll.end());
    coll.pop_back();

    PRINT_ELEMENTS (coll, "after pop_heap(): ");

    // push new element into the heap
    coll.push_back (17);
    push_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after push_heap(): ");

    /* convert heap into a sorted collection
     * - NOTE: after the call it is no longer a heap
     */
    sort_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after sort_heap(): ");
}
```

程序输出如下:

```
on entry: 3 4 5 6 7 5 6 7 8 9 1 2 3 4
after make_heap(): 9 8 6 7 7 5 5 3 6 4 1 2 3 4
after pop_heap(): 8 7 6 7 4 5 5 3 6 4 1 2 3
after push_heap(): 17 7 8 7 4 5 6 3 6 4 1 2 3 5
after sort_heap(): 1 2 3 3 4 4 5 5 6 6 7 7 8 17
```

调用 `make_heap()` 之后, 元素被排序为 `heap`:

```
9 8 6 7 7 5 5 3 6 4 1 2 3 4
```

如果你把这些元素转换为二叉树结构, 你会发现每个节点的值都小于或等于其父节点值 (图 9.1)。`push_heap()` 和 `pop_heap()` 虽然会替换元素, 但二叉树结构的恒常性质 (亦即: 每个节点不大于其父节点) 不变。

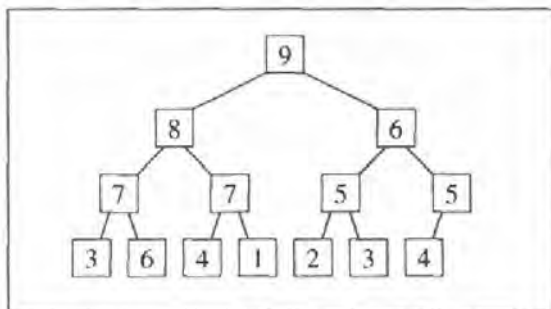


图 9.1 所谓 `Heap`, 其元素形成一个二叉树 (Binary Tree)

9.10 已序区间算法 (Sorted Range Algorithms)

针对已序区间执行的算法, 执行前提是源区间必须在某个排序准则下已序 (*sorted*)。较之其对应兄弟 (无序区间), 它们有着明显的性能优势 (通常是对数复杂度, 而不再是线性复杂度)。即使迭代器并非随机存取型, 也可以使用这些算法——不过如此一来这些算法的复杂度会降为线性, 因为迭代器只能一步一步移动。但其比较次数的复杂度仍是对数型。

根据 C++ *Standard*, 对着“无序序列”调用这些算法, 会导致未定义的行为。然而大部分实作版本中, 这些算法对于无序序列仍然有效。只不过, 如果你倚仗这个事实, 你的程序将不具移植性。

对应于此处所给的算法, 关联式容器提供了对应的成员函数。如果要搜寻某个特定的 *key* 或 *value*, 你应该使用那些成员函数。

9.10.1 搜寻元素 (Searching)

下列算法在已序 (*sorted*) 区间中搜寻某元素。

检查某个元素是否存在

```
bool
binary_search (ForwardIterator beg, ForwardIterator end,
                const T& value)

bool
binary_search (ForwardIterator beg, ForwardIterator end,
                const T& value,
                BinaryPredicate op)
```

- 两种形式都用来判断已序区间 $[beg, end)$ 中是否包含“和 *value* 等值”的元素。
- *op* 是一个可有可无的 (可选的) 二元判断式, 用来作为排序准则:
`op(elem1, elem2)`
- 如果想要获得被搜寻元素的位置, 应使用 `lower_bound()`, `upper_bound()` 或 `equal_range()` (参见 p413 和 p415)。
- 调用者必须确保进入算法之际, 该区间已序 (在指定的排序准则作用下)。
- 复杂度: 如果搭配随机存取迭代器, 则为对数复杂度, 否则为线性复杂度 (这些算法最多执行 $\log(\text{numberOfElements})+2$ 次比较操作, 但若不是随机存取迭代器, 迭代器在元素身上移动的复杂度是线性, 于是整体复杂度就是线性了)。

以下示范 `binary_search()` 的用法:

```
// algo/bsearch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll, 1, 9);
    PRINT_ELEMENTS(coll);

    // check existence of element with value 5
    if (binary_search(coll.begin(), coll.end(), 5)) {
        cout << "5 is present" << endl;
    }
    else {
        cout << "5 is not present" << endl;
    }
}
```

```
// check existence of element with value 42
if (binary_search(coll.begin(), coll.end(), 42)) {
    cout << "42 is present" << endl;
}
else {
    cout << "42 is not present" << endl;
}
}
```

程序输出如下:

```
1 2 3 4 5 6 7 8 9
5 is present
42 is not present
```

检查若干个值是否存在

```
bool
includes (InputIterator1 beg,
           InputIterator1 end,
           InputIterator2 searchBeg,
           InputIterator2 searchEnd)

bool
includes (InputIterator1 beg,
           InputIterator1 end,
           InputIterator2 searchBeg,
           InputIterator2 searchEnd,
           BinaryPredicate op)
```

- 两种形式都用来判断已序区间 $[beg, end)$ 是否包含另一个已序区间 $[searchBeg, searchEnd)$ 的全部元素。也就是说对于 $[searchBeg, searchEnd)$ 中的每一个元素, 如果 $[beg, end)$ 必有一个对应的相等元素, 那么 $[searchBeg, searchEnd)$ 肯定是 $[beg, end)$ 的子集。
- op 是一个可有可无的 (可选的) 二元判断式, 被用来作为排序准则:
 $op(elem1, elem2)$
- 调用者必须确保在进入算法之际, 两区间都应该已经按照相同的排序准则排好序了。
- 复杂度: 线性, 最多执行 $2 * (numberOfElements + searchElements) - 1$ 次比较操作。

以下程序示范 `includes()` 的用法:

```
// algo/includes.cpp

#include "algorithstuff.hpp"
using namespace std;

int main()
{
    list<int> coll;
    vector<int> search;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    search.push_back(3);
    search.push_back(4);
    search.push_back(7);
    PRINT_ELEMENTS(search,"search: ");

    // check whether all elements in search are also in coll
    if (includes (coll.begin(), coll.end(),
                  search.begin(), search.end())) {
        cout << "all elements of search are also in coll"
              << endl;
    }
    else {
        cout << "not all elements of search are also in coll"
              << endl;
    }
}
```

程序输出如下:

```
coll: 1 2 3 4 5 6 7 8 9
search: 3 4 7
all elements of search are also in coll
```

搜寻第一个或最后一个可能位置

ForwardIterator

lower_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator

lower_bound (ForwardIterator beg, ForwardIterator end, const T& value,
BinaryPredicate op)

ForwardIterator

upper_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator

upper_bound (ForwardIterator beg, ForwardIterator end, const T& value,
BinaryPredicate op)

- **lower_bound()** 返回第一个“大于等于 *value*”的元素位置。这是可插入“元素值为 *value*”且“不破坏区间 [*beg*, *end*) 已序性”的第一个位置。
- **lower_bound()** 返回第一个“大于 *value*”的元素位置。这是可插入“元素值为 *value*”且“不破坏区间 [*beg*, *end*) 已序性”的最后一个位置。
- 如果不存在“其值为 *value*”的元素，上述所有算法都返回 *end*。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
 `op(elem1, elem2)`
- 调用者必须确保进入算法之际，所有区间都已按照排序准则排好序了。
- 如要同时获得 **lower_bound()** 和 **upper_bound()** 的结果，请使用 **equal_range()**（稍后介绍）。
- 关联式容器（**set**, **multiset**, **map**, **multimap**）分别提供等效成员函数，性能更佳。
- 复杂度：如果搭配随机存取迭代器，则为对数复杂度，否则为线性复杂度（这些算法最多执行 $\log(\text{numberOfElements})+1$ 次比较动作，但若不是随机存取迭代器，迭代器在元素身上移动的复杂度是线性的，于是整体复杂度就是线性的了）。

以下程序示范 **lower_bound()** 和 **upper_bound()**⁸ 的用法：

```
// algo/bounds1.cpp

#include "alghostuff.hpp"
using namespace std;
```

⁸ 早期的 STL 版本可能需要含入 `distance.hpp` 文件，参见 p263。

```

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);
    coll.sort ();
    PRINT_ELEMENTS(coll);

    // print first and last position 5 could get inserted
    list<int>::iterator pos1, pos2;

    pos1 = lower_bound (coll.begin(), coll.end(),
                        5);

    pos2 = upper_bound (coll.begin(), coll.end(),
                        5);

    cout << "5 could get position "
         << distance(coll.begin(),pos1) + 1
         << " up to "
         << distance(coll.begin(),pos2) + 1
         << " without breaking the sorting" << endl;

    // insert 3 at the first possible position without
    // breaking the sorting
    coll.insert (lower_bound(coll.begin(),coll.end(),
                             3),
                3);

    // insert 7 at the last possible position without
    // breaking the sorting
    coll.insert (upper_bound(coll.begin(),coll.end(),
                             7),
                7);

    PRINT_ELEMENTS(coll);
}

```

程序输出如下:

```

1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9

```

搜寻第一个和最后一个可能位置

```
pair<ForwardIterator, ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value)

pair<ForwardIterator, ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value,
               BinaryPredicate op)
```

- 两种形式都返回“与 value 相等”的元素所形成的区间。在此区间内插入“其值为 value”的元素，并不会破坏区间 $[beg, end)$ 的已序性。
- 和下式等效：
`make_pair(lower_bound(...), upper_bound(...))`
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
`op(elem1, elem2)`
- 调用者必须确保在进入算法之际，区间已按照排序准则排好序了。
- 关联式容器（set, multiset, map, multimap）都提供有等效成员函数，性能更佳。
- 复杂度：如果搭配随机存取迭代器，则为对数复杂度，否则为线性复杂度（这些算法最多执行 $2 * \log(\text{numberOfElements}) + 1$ 次比较动作，但若不是随机存取迭代器，迭代器在元素身上移动的复杂度是线性的，于是整体复杂度就是线性的了）。

以下程序展示 `equal_range()`⁹ 的用法：

```
// algo/eqrangel.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll, 1, 9);
    INSERT_ELEMENTS(coll, 1, 9);
    coll.sort ();
    PRINT_ELEMENTS(coll);

    // print first and last position 5 could get inserted
    pair<list<int>::iterator, list<int>::iterator> range;
```

⁹ 早期的 STL 版本可能需要 `distance.hpp` 文件，参见 p263。


```

range = equal_range (coll.begin(), coll.end(),
                    5);
cout << "5 could get position "
    << distance(coll.begin(), range.first) + 1
    << " up to "
    << distance(coll.begin(), range.second) + 1
    << " without breaking the sorting" << endl;
}

```

程序输出如下:

```

? 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
? 5 could get position 9 up to 11 without breaking the sorting

```

9.10.2 合并元素 (Merging)

本节的算法用来将两个区间的元素合并, 包括总和 (sum)、并集 (union)、交集 (intersection) 等处理。

两个已序集合的总和 (Sum)

```

OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
      InputIterator source2Beg, InputIterator source2End,
      OutputIterator destBeg)

```

```

OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
      InputIterator source2Beg, InputIterator source2End,
      OutputIterator destBeg, BinaryPredicate op)

```

- 两者都是将源区间 $[source1Beg, source1End)$ 和 $[source2Beg, source2End)$ 内的元素合并, 使得“以 *destBeg* 起始的目标区间”内含两个源区间的所有元素。假设你对下面两个序列调用 `merge()`:

```
1 2 2 4 6 7 7 9
```

和

```
2 2 2 3 6 6 8 9
```

结果将会是:

```
1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
```

- 目标区间内的所有元素都将按顺序排列。

- 两者都返回目标区间内“最后一个被复制元素”的下一位置（也就是第一个未被覆盖的元素位置）。
- `op` 是个可有可无的（可选的）二元判断式，被当做排序准则：
`op(elem1, elem2)`
- 源区间没有任何变化。
- 根据标准，调用者应当确保两个源区间一开始都已序。然而在大部分实作版本中，上述算法可以将两个无序的源区间内的元素合并到一个无序的目标区间中。不过如果考虑移植性，这种情况下你应该调用 `copy()` 两次，而不是使用 `merge()`。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重复。
- `lists` 提供了一个特殊成员函数 `merge()`，用来合并两个 `lists`（参见 p246）。
- 如果你要确保“两个源区间中都存在的元素”在目标区间中只出现一次，请使用 `set_union()`（参见 p418）。
- 如果你只想获得“同时存在于两个源区间内”的所有元素，请使用 `set_intersection()`（参见 p419）。
- 复杂度：线性，最多执行 $\text{numberOfElements1} + \text{numberOfElements2} - 1$ 次比较。

下面这个例子展示 `merge()` 的用法：

```
// algo/merge1.cpp

#include "alghostuff.hpp"
using namespace std;

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

    // fill both collections with some sorted elements
    INSERT_ELEMENTS(coll1, 1, 6);
    INSERT_ELEMENTS(coll2, 3, 8);

    PRINT_ELEMENTS(coll1, "coll1: ");
    PRINT_ELEMENTS(coll2, "coll2: ");

    // print merged sequence
    cout << "merged: ";
    merge (coll1.begin(), coll1.end(),
           coll2.begin(), coll2.end(),
           ostream_iterator<int>(cout, " "));
```

```
    cout << endl;
}
```

程序输出如下:

```
coll1: 1 2 3 4 5 6
coll2: 3 4 5 6 7 8
merged: 1 2 3 3 4 4 5 5 6 6 7 8
```

p421 另有一个例子, 展示“用以合并已序序列”的各种算法之间的不同。

两个已序集合的并集 (Union)

OutputIterator

```
set_union (InputIterator source1Beg, InputIterator source1End,
            InputIterator source2Beg, InputIterator source2End,
            OutputIterator destBeg)
```

OutputIterator

```
set_union (InputIterator source1Beg, InputIterator source1End,
            InputIterator source2Beg, InputIterator source2End,
            OutputIterator destBeg, BinaryPredicate op)
```

- 以上两者都是将已序的源区间 $[source1Beg, source1End)$ 和 $[source2Beg, source2End)$ 内的元素合并, 得到“以 $destBeg$ 起始”的目标区间——这个区间内含的元素要不来自第一源区间, 要不就来自第二源区间, 或是同时来自两个源区间。例如, 对下面两个已序序列调用 `set_union()`:

```
1 2 2 4 6 7 7 9
```

和

```
2 2 2 3 6 6 8 9
```

结果是

```
1 2 2 2 3 4 6 6 7 7 8 9
```

- 目标区间内的所有元素都按顺序排列。
- 同时出现于两个源区间内的元素, 在并集区间中将只出现一次。不过如果原来的某个源区间内原本就存在重复元素, 则目标区间内也会有重复元素——重复的个数是两个源区间内的重复个数的较大值 (译注: 可参考《STL 源码剖析》6.5.1 节实例图解)。
- 两者都返回目标区间内“最后一个被复制元素”的下一位置 (也就是第一个未被覆盖的元素位置)。
- op 是个可有可无的 (可选的) 二元判断式, 被当做排序准则:
 $op(elem1, elem2)$

- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序 (*sorted*)。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重迭。
- 若想得到两个源区间的全部元素，请用 `merge()` (参见 p416)。
- 复杂度：线性，最多执行 $2 * (\text{numberOfElements1} + \text{numberOfElements2}) - 1$ 次比较操作。

p421 有一个 `set_union()` 使用范例。该例也展示各种已序序列合并算法的区别。

两个已序集合的交集 (Intersection)

```
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg)

OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg,
                  BinaryPredicate op)
```

- 以上两者都是将已序源区间 `[source1Beg, source1End)` 和 `[source2Beg, source2End)` 的元素合并，得到“以 `destBeg` 起始”的目标区间——这个区间内含的元素不但存在于第一源区间，也存在于第二源区间。例如，对下面两个已序序列调用 `set_intersection()`：

```
1 2 2 4 6 7 7 9
```

和

```
2 2 2 3 6 6 8 9
```

结果是：

```
2 2 6 9
```

- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是两个源区间内的重复个数的较小值 (译注：可参考《STL 源码剖析》6.5.2 节实例图解)。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- `op` 是个可有可无的二元判断式，被当做排序准则：

$$op(\text{elem1}, \text{elem2})$$
- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序 (*sorted*)。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。

- 目标区间和源区间不得重叠。
- 复杂度：线性，最多执行 $2 * (\text{numberOfElements1} + \text{numberOfElements2}) - 1$ 次比较动作。

p421 有一个 `set_intersection()` 使用范例。该例也展示各种已序序列合并算法之间的区别。

两个已序集合的差集 (Difference)

```
OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg)

OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg,
                BinaryPredicate op)
```

- 以上两者都是将已序源区间 `[source1Beg, source1End)` 和 `[source2Beg, source2End)` 的元素合并，得到“以 `destBeg` 起始”的目标区间——这个区间内含的元素只存在于第一源区间，不存在于第二源区间。例如，对下面两个已序序列调用 `set_difference()`：

1 2 2 4 6 7 7 9

和

2 2 2 3 6 6 8 9

结果是

1 4 7 7

- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是第一源区间内的重复个数减去第二源区间内的相应重复个数，如果第二源区间内的重复个数大于第一源区间内的相应重复个数，目标区间内的对应重复个数将会是零（译注：可参考《STL 源码剖析》6.5.3 节实例图解）。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- `op` 是个可有可无的（可选的）二元判断式，被当做排序准则：
`op(elem1, elem2)`
- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（sorted）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重叠。
- 复杂度：线性，最多执行 $2 * (\text{numberOfElements1} + \text{numberOfElements2}) - 1$ 次比较。

p421 上有一个 `set_difference()` 使用范例。该例也展示各种已序序列合并算法之间的区别。

OutputIterator

```
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg)
```

OutputIterator

```
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg,
                          BinaryPredicate op)
```

- 两者都是将已序源区间 $[source1Beg, source1End)$ 和 $[source2Beg, source2End)$ 的元素合并，得到“以 $destBeg$ 起始”的目标区间——这个区间内含的元素或存在于第一源区间，或存在于第二源区间，但不同时存在于两源区间内。例如，对下面两个已序序列调用 `set_symmetric_difference()`：

1 2 2 4 6 7 7 9

和

2 2 2 3 6 6 8 9

结果是

1 2 3 4 6 7 7 8

- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是两个源区间内的对应重复元素的个数差额（译注：可参考《STL 源码剖析》6.5.4 节实例图解）。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- op 是个可有可无的（可选的）二元判断式，被当做排序准则：
`op(elem1, elem2)`
- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（sorted）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重叠。
- 复杂度：线性，最多执行 $2 * (numberOfElements1 + numberOfElements2) - 1$ 次比较动作。

下一小节有个 `set_symmetric_difference()` 使用范例。该例也展示各种已序序列合并算法之间的区别。

“合并算法”的综合范例

下面这个例子对各个已序序列合并算法做了比较，展示其用法和区别：

```
// algo/setalgorithms.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    int c1[] = { 1, 2, 2, 4, 6, 7, 7, 9 };
    int num1 = sizeof(c1) / sizeof(int);

    int c2[] = { 2, 2, 2, 3, 6, 6, 8, 9 };
    int num2 = sizeof(c2) / sizeof(int);

    // print source ranges
    cout << "c1: " ;
    copy (c1, c1+num1,
          ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "c2: " ;
    copy (c2, c2+num2,
          ostream_iterator<int>(cout, " "));
    cout << '\n' << endl;

    // sum the ranges by using merge()
    cout << "merge(): ";
    merge (c1, c1+num1,
           c2, c2+num2,
           ostream_iterator<int>(cout, " "));
    cout << endl;

    // unite the ranges by using set_union()
    cout << "set_union(): ";
    set_union (c1, c1+num1,
               c2, c2+num2,
               ostream_iterator<int>(cout, " "));
    cout << endl;

    // intersect the ranges by using set_intersection()
    cout << "set_intersection(): ";
    set_intersection (c1, c1+num1,
                       c2, c2+num2,
                       ostream_iterator<int>(cout, " "));
    cout << endl;
```

```

// determine elements of first range without elements of second range
// by using set_difference()
cout << "set_difference(): ";
set_difference (c1, c1+num1,
               c2, c2+num2,
               ostream_iterator<int>(cout, " "));
cout << endl;

// determine difference the ranges with
set_symmetric_difference()
cout << "set_symmetric_difference(): ";
set_symmetric_difference (c1, c1+num1,
                          c2, c2+num2,
                          ostream_iterator<int>(cout, " "));
cout << endl;
}

```

程序输出如下:

```

c1:                1 2 2 4 6 7 7 9
c2:                2 2 2 3 6 6 8 9

merge():           1 2 2 2 2 3 4 6 6 6 7 7 8 9 9
set_union():       1 2 2 2 3 4 6 6 7 7 8 9
set_intersection(): 2 2 6 9
set_difference():  1 4 7 7
set_symmetric_difference(): 1 2 3 4 6 7 7 8

```

将连贯的 (consecutive) 已序区间合并

```

void
inplace_merge (BidirectionalIterator beg1,
                BidirectionalIterator end1beg2,
                BidirectionalIterator end2)

void
inplace_merge (BidirectionalIterator beg1,
                BidirectionalIterator end1beg2,
                BidirectionalIterator end2, BinaryPredicate op)

```

- 两者都是将已序源区间 $[beg1, end1beg2)$ 和 $[end1beg2, end2)$ 的元素合并, 使区间 $[beg1, end2)$ 成为两者之总和 (且形成已序)。
- 复杂度: 如有足够的内存则为线性, 执行 $numberOfElements-1$ 次比较操作, 否则为 $n \log n$, 执行 $numberOfElements * \log(numberOfElements)$ 次比较操作。

以下程序示范 `inplace_merge()` 的用法:

```
// algo/imerge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert two sorted sequences
    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,8);
    PRINT_ELEMENTS(coll);

    // find beginning of second part (element after 7)
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(), // range
               7);                       // value
    ++pos;

    // merge into one sorted range
    inplace_merge (coll.begin(), pos, coll.end());

    PRINT_ELEMENTS(coll);
}
```

程序输出如下:

```
1 2 3 4 5 6 7 1 2 3 4 5 6 7 8
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8
```

9.11 数值算法 (Numeric Algorithms)

本节叙述用于数值处理的 STL 算法。当然你也可以用它们来处理非数值元素。例如你可以用 `accumulate()` 求数个字符串和。运用数值算法之前必须先含入头文件 `<numeric>`¹⁰：

```
#include <numeric>
```

9.11.1 加工运算后产生结果

对序列进行某种运算

```
T  
accumulate (InputIterator beg, InputIterator end,  
             T initValue)
```

```
T  
accumulate (InputIterator beg, InputIterator end,  
             T initValue, BinaryFunc op)
```

- 以上第一种形式计算 `initValue` 和区间 `(beg, end)` 内的所有元素的总和，更具体地说，它针对每一个元素调用以下表达式：
`initValue = initValue + elem`
- 以上第二种形式计算 `initValue` 和区间 `[beg, end)` 内每一个元素进行 `op` 运算的结果，更具体地说，它针对每一个元素调用以下表达式：
`initValue = op(initValue, elem)`
- 因此，对于以下数值序列：

```
a1 a2 a3 a4 ...
```

上述两个算法分别计算

```
initValue + a1 + a2 + a3 + ...
```

和

```
initValue op a1 op a2 op a3 op ...
```

- 如果序列为空 (`beg==end`)，则两者都返回 `initValue`。
- 复杂度：线性，上述两式分别调用 `operator+` 或 `op()` 各 *numberOfElements* 次。

下面这个例子展示如何使用 `accumulate()` 得到区间内所有元素的加总和乘积：

¹⁰ 早期的 STL 中，数值算法被定义于 `<algo.h>`。

```
// algo/accu1.cpp

#include "algorith.h"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // process sum of elements
    cout << "sum: "
         << accumulate (coll.begin(), coll.end(), // range
                        0) // initial value
         << endl;

    // process sum of elements less 100
    cout << "sum: "
         << accumulate (coll.begin(), coll.end(), // range
                        -100) // initial value
         << endl;

    // process product of elements
    cout << "product: "
         << accumulate (coll.begin(), coll.end(), // range
                        1, // initial value
                        multiplies<int>()) // operation
         << endl;

    // process product of elements (use 0 as initial value)
    cout << "product: "
         << accumulate (coll.begin(), coll.end(), // range
                        0, // initial value
                        multiplies<int>()) // operation
         << endl;
}
```

程序输出如下:

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
product: 0
```

最后一个输出结果是 0, 因为任何数乘以 0 都得 0。

计算两序列的内积 (Inner Product)

```
T
inner_product (InputIterator1 beg1, InputIterator1 end1,
                InputIterator2 beg2, T initValue)
```

```
T
inner_product (InputIterator1 beg1, InputIterator1 end1,
                InputIterator2 beg2, T initValue,
                BinaryFunc op1, BinaryFunc op2)
```

- 以上第一种形式计算并返回 $[beg, end)$ 区间和 “以 $beg2$ 为起始的区间” 的对应元素组 (再加上 $initValue$) 的内积。具体地说也就是针对 “两区间内的每一组对应元素” 调用以下表达式:

$$initValue = initValue + elem1 * elem2$$

- 以上第二种形式将 $[beg, end)$ 区间和 “以 $beg2$ 为起始的区间” 内的对应元素组进行 $op2$ 运算, 然后再和 $initValue$ 进行 $op1$ 运算, 并将结果返回。具体地说也就是针对 “两区间内的每一组对应元素” 调用以下表达式:

$$initValue = op1(initValue, op2(elem1, elem2))$$

- 所以, 对于数值序列:

```
a1 a2 a3 ...
b1 b2 b3 ...
```

上述两个算法分别计算并返回:

$$initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + \dots$$

和

$$initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 \dots$$

- 如果第一区间为空 ($beg1 == end1$), 则两者都返回 $initValue$ 。
- 调用者必须确保 “以 $beg2$ 为起始的区间” 内含足够元素空间。
- $op1$ 和 $op2$ 都不得变动 (修改) 其参数内容。
- 复杂度: 线性, 调用 $operator+$ 和 $operator*$ 各 $numberOfElements$ 次, 或是调用 $op1()$ 和 $op2()$ 各 $numberOfElement$ 次。

以下程序示范 `inner_product()` 的用法。它计算两个序列的乘积总和 (sum of product)，以及总和乘积 (product of sum)：

```
// algo/inner1.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    /* process sum of all products
     * (0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
     */
    cout << "inner product: "
         << inner_product (coll.begin(), coll.end(), // first range
                           coll.begin(),           // second range
                           0)                       // initial value
         << endl;

    /* process sum of 1*6 ... 6*1
     * (0 + 1*6 + 2*5 + 3*4 + 4*3 + 5*2 + 6*1)
     */
    cout << "inner reverse product: "
         << inner_product (coll.begin(), coll.end(), // first range
                           coll.rbegin(),           // second range
                           0)                       // initial value
         << endl;

    /* process product of all sums
     * (1 * 1+1 * 2+2 * 3+3 * 4+4 * 5+5 * 6+6)
     */
    cout << "product of sums: "
         << inner_product (coll.begin(), coll.end(), // first range
                           coll.begin(),           // second range
                           1,                      // initial value
                           multiplies<int>(),       // inner operation
                           plus<int>())             // outer operation
         << endl;
}
```

程序输出如下:

```
1 2 3 4 5 6
inner product: 91
inner reverse product: 56
product of sums: 46080
```

9.11.2 相对值和绝对值之间的转换

下面两个算法可以在相对值序列和绝对值序列之间相互转换。

将相对值转换成绝对值

```
OutputIterator
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEnd,
             OutputIterator destBeg, BinaryFunc op)
```

- 第一形式计算源区间 $[sourceBeg, sourceEnd)$ 中每个元素的部分和, 然后将结果写入以 $destBeg$ 为起点的目标区间。
- 第二形式将源区间 $[sourceBeg, sourceEnd)$ 中的每个元素和其先前所有元素进行 op 运算, 并将结果写入以 $destBeg$ 为起点的目标区间。
- 因此, 对于以下数值序列:

$a_1 \ a_2 \ a_3 \ \dots$

它们分别计算:

$a_1, \ a_1 + a_2, \ a_1 + a_2 + a_3, \ \dots$

和

$a_1, \ a_1 \ op \ a_2, \ a_1 \ op \ a_2 \ op \ a_3, \ \dots$

- 两种形式都返回目标区间内“最后一个被写入的值”的下一位置 (也就是第一个未被覆盖的元素的位置)。

- 第一形式相当于把一个相对值序列转换为一个绝对值序列。就此而言，`partial_sum()`正好和`adjacent_difference()`互补。
- 源区间和目标区间可以相同。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- `op`不得变动（更改）传入的参数。
- 复杂度：线性，分别调用`operator+` 或 `op()` *numberOfElements* 次。

以下程序示范`partial_sum()`的用法：

```
// algo/partsum1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    // print all partial sums
    partial_sum (coll.begin(), coll.end(),           // source range
                 ostream_iterator<int>(cout," "),    // destination
                 cout << endl;

    // print all partial products
    partial_sum (coll.begin(), coll.end(),           // source range
                 ostream_iterator<int>(cout," "),    // destination
                 multiplies<int>{}),                 // operation
                 cout << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6
1 3 6 10 15 21
1 2 6 24 120 720
```

关于相对值和绝对值之间的互换，请参见 p432 的例子。

将绝对值转换成相对值

```
OutputIterator
adjacent_difference (InputIterator sourceBeg,
                    InputIterator sourceEnd,
                    OutputIterator destBeg)

OutputIterator
adjacent_difference (InputIterator sourceBeg,
                    InputIterator sourceEnd,
                    OutputIterator destBeg, BinaryFunc op)
```

- 第一种形式计算区间 $[sourceBeg, sourceEnd)$ 中每一个元素和其紧邻前趋元素的差额，并将结果写入以 $destBeg$ 为起点的目标区间。
- 第二种形式针对区间 $[sourceBeg, sourceEnd)$ 中的每一个元素和其紧邻前趋元素调用 op 操作，并将结果写入以 $destBeg$ 为起点的目标区间。
- 第一个元素只是被很单纯地加以复制。
- 因此，对于以下数值序列：
 $a_1 \ a_2 \ a_3 \ a_4 \ \dots$
 它们分别计算：
 $a_1, \ a_2 - a_1, \ a_3 - a_2, \ a_4 - a_3, \ \dots$
 和：
 $a_1, \ a_2 \ op \ a_1, \ a_3 \ op \ a_2, \ a_4 \ op \ a_3, \ \dots$
- 两种形式都返回目标区间内“最后一个被写入的值”的下一位置（也就是第一个未被覆盖的元素的位置）。
- 第一形式相当于把一个绝对值序列转换为一个相对值序列。就此而言， $adjacent_difference()$ 正好与 $patial_sum()$ 互补。
- 源区间和目标区间可以相同。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- op 不得变动（更改）传入的参数。
- 复杂度：线性，分别调用 $operator-$ 或 $op()$ $numberOfElements-1$ 次。

以下程序示范 $adjacent_difference()$ 的用法：

```
// algo/adjdiff1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
```



```
INSERT_ELEMENTS(coll,1,6);
PRINT_ELEMENTS(coll);

// print all differences between elements
adjacent_difference (coll.begin(), coll.end(), // source
                    ostream_iterator<int>(cout, " ")); // dest.
cout << endl;

// print all sums with the predecessors
adjacent_difference (coll.begin(), coll.end(), // source
                    ostream_iterator<int>(cout, " "), // dest.
                    plus<int>());              // operation
cout << endl;

// print all products between elements
adjacent_difference (coll.begin(), coll.end(), // source
                    ostream_iterator<int>(cout, " "), // dest.
                    multiplies<int>());         // operation
cout << endl;
}
```

程序输出如下:

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

参见下一个例子, 其中将绝对值和相对值互相转换。

相对值转换为绝对值, 实例解说

下面这个例子展示如何运用 `partial_sum()` 和 `adjacent_difference()` 将一个相对值序列转化为一个绝对值序列, 以及如何逆向实施:

```
// algo/relabs.cpp

#include "algotuff.hpp"
using namespace std;

int main()
{
```

```
vector<int> coll;

coll.push_back(17);
coll.push_back(-3);
coll.push_back(22);
coll.push_back(13);
coll.push_back(13);
coll.push_back(-9);
PRINT_ELEMENTS(coll,"coll: ");

// convert into relative values
adjacent_difference (coll.begin(), coll.end(), // source
                    coll.begin());           // destination
PRINT_ELEMENTS(coll,"relative: ");

// convert into absolute values
partial_sum (coll.begin(), coll.end(),      // source
            coll.begin());                  // destination
PRINT_ELEMENTS(coll,"absolute: ");
}
```

程序输出如下:

```
coll: 17 -3 22 13 13 -9
relative: 17 -20 25 -9 0 -22
absolute: 17 -3 22 13 13 -9
```

10

特殊容器

Special Containers

C++ 标准程序库不光只是包含了 STL framework 所提供的容器, 还包含一些为满足特殊需求而设计的容器, 它们提供简单而清晰的接口。这些容器可归类如下:

- 容器配接器 (container adapters)

这些容器配接器对标准 STL 容器予以配接 (*adapt*), 使之满足特殊需求。有三种标准容器配接器:

1. Stacks (堆栈)
2. Queues (队列)
3. Priority queues (带优先序的队列)

Priority queue 就是“根据排序准则, 自动将元素排序”的 queue, 所以在 priority queue 中所谓“下一个”元素总拥有最大值 (最高优先级)。

- 一个名为 bitset 的特殊容器。

所谓 bitset 就是一个位域 (bitfield), 其中可含任意数量的 bits, 但一旦确定就固定不再变动。你可以想象它是一个 bits 容器或 Boolean 容器。注意, C++ 标准程序库同时也为 Boolean 值提供了一个长度可变的特殊容器: `vector<bool>`, 本书 6.2.6 节, p158 曾经提过。

10.1 Stacks (堆栈)

`class stack<>` 实作出一个 stack (也称为 LIFO, 后进先出)。你可以使用 `push()` 将任意数量的元素置入 stack 中, 也可以使用 `pop()` 将元素依其插入次序的反序从容器中移除 (此即所谓“后进先出, LIFO”), 如图 10.1 所示。

为了运用 stack, 你必须先含入头文件 `<stack>`¹:

```
#include <stack>
```

¹ 早期的 STL 中, stacks 被定义于 `<stack.h>`

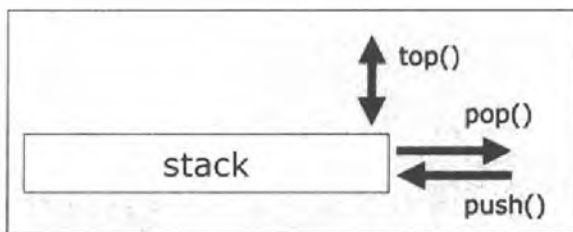


图 10.1 Stack 的接口

在头文件 `<stack>` 中, `class stack` 定义如下:

```
namespace std {
    template <class T,
              class Container = deque<T> >
        class stack;
}
```

第一个 `template` 参数代表元素型别。带有默认值的第二个 `template` 参数用来定义 `stack` 内部存放元素所用的实际容器, 缺省采用 `deque`。之所以选择 `deque` 而非 `vector`, 是因为 `deque` 移除元素时会释放内存, 并且不必在重新分配 (reallocation) 时复制全部元素 (关于如何恰当运用各种容器, 请参考 6.9 节, p226)。

例如, 以下声明就定义了一个元素型别为整数的 `stack`²:

```
std::stack<int> st; // integer stack
```

实际上 `stack` 只是很单纯地把各项操作转化为内部容器的对应调用 (图 10.2)。你可以使用任何序列式容器来支持 `stack`, 只要它们支持 `back()`, `push_back()`, `pop_back()` 等动作就行。例如你可以使用 `vector` 或 `list` 来容纳元素:

```
std::stack<int, std::vector<int> > st; // integer stack that uses a vector
```

10.1.1 核心接口

`Stacks` 的核心接口就是三个成员函数 `push()`, `top()`, `pop()`:

- `push()` 会将一个元素置入 `stack` 内。
- `top()` 会返回 `stack` 内的“下一个”元素。
- `pop()` 会从 `stack` 中移除元素。

² 在 STL 早期版本中, 你可以把容器当做唯一的 `template` 参数, 所以应当如下声明一个整数 `stack`:

```
stack< deque<int> > st;
```

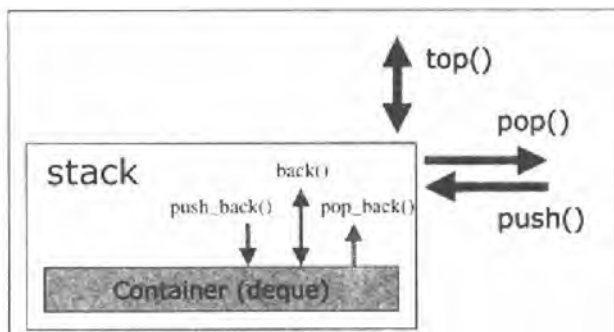


图 10.2 Stack 的内部接口

注意，`pop()` 移除下一个元素，但是并不将它返回；`top()` 返回下一个元素，但是并不移除它。所以如果你想移除 `stack` 的下一个元素同时并返回它，那么这两个函数都得调用。这样的接口可能有点麻烦，但如果你只是想移除下一个元素而并不想处理它，这样的安排就比较好。注意，如果 `stack` 内没有元素，则执行 `top()` 和 `pop()` 会导致未定义的行为。你可以采用成员函数 `size()` 和 `empty()` 来检验容器是否为空。

如果你不喜欢 `stack<>` 的标准接口，轻易便可撰写一些更方便的接口。相关实例请见 10.1.4 节, p441。

10.1.2 Stacks 运用实例

下面这个程序展示 `stack<>` 的用法：

```
// cont/stack1.cpp

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

int main()
{
    stack<int> st;

    // push three elements into the stack
    st.push(1);
    st.push(2);
    st.push(3);
```

```
// pop and print two elements from the stack
cout << st.top() << ' ';
st.pop();
cout << st.top() << ' ';
st.pop();

// modify top element
st.top() = 77;

// push two new elements
st.push(4);
st.push(5);

// pop one element without processing it
st.pop();

// pop and print remaining elements
while (!st.empty()) {
    cout << st.top() << ' ';
    st.pop();
}
cout << endl;
}
```

程序输出如下:

3 2 4 77

10.1.3 Class stack<> 细部讨论

class stack<> 的接口非常小, 你可以直接阅读其典型实现代码, 从而轻松理解它:

```
namespace std {
    template <class T, class Container = deque<T> >
    class stack {
    public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef Container container_type;
```

```

protected:
    Container c; // container
public:
    explicit stack(const Container& = Container());

    bool empty() const           { return c.empty(); }
    size_type size() const       { return c.size(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop()                   { c.pop_back(); }
    value_type& top()             { return c.back(); }
    const value_type& top() const { return c.back(); }
};

template <class T, class Container>
    bool operator==(const stack<T, Container>&,
                    const stack<T, Container>&);
template <class T, class Container>
    bool operator< (const stack<T, Container>&,
                    const stack<T, Container>&);
... // (other comparison operators)
}

```

下面是各个成员的详细说明。

型别定义

stack::value_type

- 元素型别
- 它和 `container::value_type` 相当。

stack::size_type

- 不带正负号的整数型别，用来表现大小。
- 它和 `container::size_type` 相当。

stack::container_type

- 内部容器的型别

各项操作 (Operations)

`stack::stack ()`

- default 构造函数
- 产生一个空的 `stack`。

`explicit stack::stack (const Container& cont)`

- 产生一个 `stack`，并以容器 `cont` 内的元素为初值。
- `cont` 内的所有元素均被复制。

`size_type stack::size () const`

- 返回元素个数。
- 如果要检验容器是否为空 (亦即不含任何元素)，应使用 `empty()`，因为它可能更快。

`bool stack::empty () const`

- 判断 `stack` 是否为空 (亦即不含任何元素)。
- 与 `stack::size()==0` 等效，但可能更快。

`void stack::push (const value_type& elem)`

- 将 `elem` 的副本安插到 `stack` 内，并成为其新的第一元素。

`value_type& stack::top ()`

`const value_type& stack::top () const`

- 以上两种形式都返回 `stack` 的下一个元素。所谓“下一个元素”是指最后一个 (亦即在其它所有元素之后) 被插入的元素。
- 调用者必须确保 `stack` 不为空 (`size()>0`)，否则可能导致未定义的行为。
- 第一种形式是针对 `non-const stacks` 设计的，返回一个 `reference`。所以你可以就地 (*in place*) 修改 `stack` 内的元素。这样做是否合适? 由你自己决定。

`void stack::pop ()`

- 移除 `stack` 内的下一个元素。所谓“下一个元素”是指最后一个 (亦即在其它所有元素之后) 被插入的元素。
- 此函数没有返回值。如果想处理被移除的那个元素，你必须先调用 `top()`。
- 调用者必须确保 `stack` 不为空 (`size()>0`)，否则可能导致未定义的行为。

`bool comparison (const stack& stack1, const stack& stack2)`

- 返回两个同型 `stack` 的比较结果。

- **comparison** 可以是下面各运算之一：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 如果两个 **stacks** 的元素个数相等，且相同次序上的元素值也相等（也就是说所有对应元素之间的比较都得到 **true**），则这两个容器相等。
- **stack** 之间的大小比较是以“字典顺序”而定。参见 p360 对于算法 **lexicographical_compare()** 的描述。

10.1.4 一个使用者自定义的 Stack Class

标准的 **stack<>** class 将运行速度置于方便性和安全性之上。但我通常并不很重视那个。便自己写了一个 **stack class**，它有以下优势：

1. **pop()** 会返回下一元素。
2. 如果 **stack** 为空，**pop()** 和 **top()** 会抛出异常。

此外，我把一般人平常使用的那些成员函数如比较动作（**comparison**）略去。我的 **stack class** 定义如下：

```
// cont/Stack.hpp
/* *****
 * Stack.hpp
 * - safer and more convenient stack class
 * *****/
#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <exception>

template <class T>
class Stack {
protected:
    std::deque<T> c; // container for the elements

public:
    /* exception class for pop() and top() with empty stack
    */
```

```
class ReadEmptyStack : public std::exception {
public:
    virtual const char* what() const throw() {
        return "read empty stack";
    }
};

// number of elements
typename std::deque<T>::size_type size() const {
    return c.size();
}

// is stack empty?
bool empty() const {
    return c.empty();
}

// push element into the stack
void push (const T& elem) {
    c.push_back(elem);
}

// pop element out of the stack and return its value
T pop () {
    if (c.empty()) {
        throw ReadEmptyStack();
    }
    T elem(c.back());
    c.pop_back();
    return elem;
}

// return value of next element
T& top () {
    if (c.empty()) {
        throw ReadEmptyStack();
    }
    return c.back();
}
};

#endif /* STACK_HPP */
```

改用这个 stack, 则先前的例子可以改写如下:

```
// cont/stack2.cpp

#include <iostream>
#include "Stack.hpp"      // use special stack class
using namespace std;

int main()
{
    try {
        Stack<int> st;

        // push three elements into the stack
        st.push(1);
        st.push(2);
        st.push(3);

        // pop and print two elements from the stack
        cout << st.pop() << ' ';
        cout << st.pop() << ' ';

        // modify top element
        st.top() = 77;

        // push two new elements
        st.push(4);
        st.push(5);

        // pop one element without processing it
        st.pop();

        /* pop and print three elements
        * ~ ERROR: one element too many
        */
    }
```

```

        cout << st.pop() << ' ';
        cout << st.pop() << endl;
        cout << st.pop() << endl;
    }
    catch (const exception& e) {
        cerr << "EXCEPTION: " << e.what() << endl;
    }
}

```

最后新增加的一个 `pop()` 调用动作是为了刻意引发错误。和标准 `stack` class 不同的是，我这个版本会抛出异常，而不引发未定义行为。程序输出如下：

```

3 2 4 77
EXCEPTION: read empty stack

```

10.2 Queues (队列)

`Class queue<>` 实作出一个 `queue` (也称为 FIFO, 先进先出)。你可以使用 `push()` 将任意数量的元素置入 `queue` 中 (图 10.3)，也可以使用 `pop()` 将元素依其插入次序从容器中移除 (此即所谓“先进先出, FIFO”)。换句话说 `queue` 是一个典型的数据缓冲区结构。

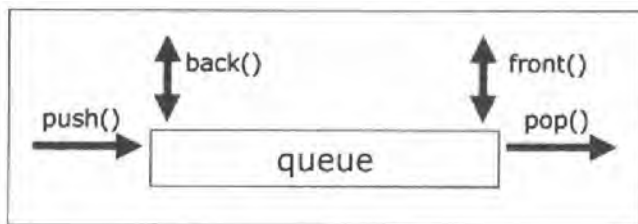


图 10.3 Queue 的接口

为了运用 `queue`，你必须先含入头文件 `<queue>`³：

```
#include <queue>
```

在头文件 `<queue>` 中，`class queue` 定义如下：

```

namespace std {
    template <class T,
              class Container = deque<T> >
        class queue;
}

```

³ 早期的 STL 中，`queues` 被定义于 `<stack.h>`

第一个 `template` 参数代表元素型别。带有默认值的第二个 `template` 参数用来定义 `queue` 内部存放元素用的实际容器，缺省采用 `deque`。

下面这个例子定义了一个内含字符串的 `queue`⁴：

```
std::queue<std::string> buffer;    // string queue
```

实际上 `queue` 只是很单纯地把各项操作转化为内部容器的对应调用(图 10.4)。你可以使用任何序列式容器来支持 `queue`，只要它们支持 `front()`, `back()`, `push_back()`, `pop_front()` 等动作就行。例如你可以使用 `list` 来容纳元素：

```
std::queue<std::string, std::list<std::string> > buffer;
```

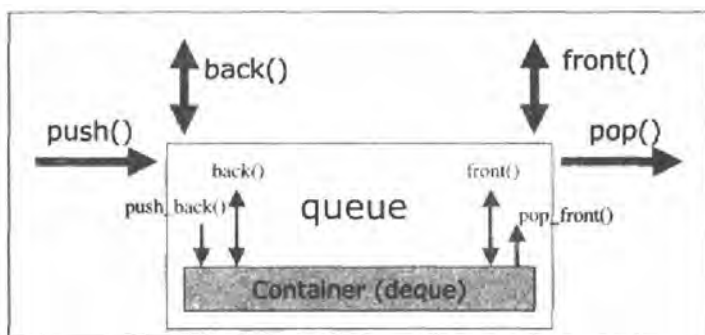


图 10.4 Queue 的内部接口

10.2.1 核心接口

`Queues` 的核心接口主要由成员函数 `push()`, `front()`, `back()`, `pop()` 构成：

- `push()` 会将一个元素置入 `queue` 中。
- `front()` 会返回 `queue` 内的下一个元素（也就是第一个被置入的元素）。

⁴ 在 STL 早期版本中，你可以把容器当做唯一的 `template` 参数，所以应当如下声明一个 `string queue`：

```
queue< deque<string> > buffer;
```

- `back()` 会返回 `queue` 中最后一个元素 (也就是第一个被插入的元素)。
- `pop()` 会从 `queue` 中移除一个元素。

注意, `pop()` 虽然移除下一个元素, 但是并不返回它, `front()` 和 `back()` 返回下一个元素, 但并不移除它。所以如果你想移除 `queue` 的下一个元素, 又想处理它, 那么就得同时调用 `front()` 和 `pop()`。这样的接口可能有点麻烦, 但如果你只是想移除下一个元素而并不想处理它, 这样的安排就比较好。注意, 如果 `queue` 之内没有元素, 则 `front()`, `back()`, `pop()` 的执行会导致未定义行为。你可以采用成员函数 `size()` 和 `empty()` 来检验容器是否为空。

如果你不喜欢 `queue<>` 的标准接口, 轻易便可撰写一些更方便的接口。相关实例请见 10.2.4 节, p450。

10.2.2 Queues 运用实例

下面这个程序展示 `queue<>` 的用法:

```
// cont/queue1.cpp

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

int main()
{
    queue<string> q;

    // insert three elements into the queue
    q.push("These ");
    q.push("are ");
    q.push("more than ");

    // read and print two elements from the queue
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();

    // insert two new elements
    q.push("four ");
    q.push("words!");
}
```

```
// skip one element
q.pop();

// read and print two elements
cout << q.front();
q.pop();
cout << q.front() << endl;
q.pop();

// print number of elements in the queue
cout << "number of elements in the queue: " << q.size()
    << endl;
}
```

程序输出如下:

```
These are four words!
number of elements in the queue: 0
```

10.2.3 Class queue<> 细部讨论

和 stack<> 相似, 典型的 queue<> 实作手法也是十分清晰易懂:

```
namespace std {
    template <class T, class Container = deque<T> >
    class queue {
    public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef Container container_type;
    protected:
        Container c;    // container
    public:
        explicit queue(const Container& = Container());

        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        void push(const value_type& x) { c.push_back(x); }
        void pop() { c.pop_front(); }
        value_type& front() { return c.front(); }
        const value_type& front() const { return c.front(); }
        value_type& back() { return c.back(); }
        const value_type& back() const { return c.back(); }
    };
}
```



```
template <class T, class Container>
    bool operator==(const queue<T, Container>&,
                    const queue<T, Container>&);
template <class T, class Container>
    bool operator< (const queue<T, Container>&,
                    const queue<T, Container>&);
... // (other comparison operators)
};
```

下面是各个成员的详细解释。

型别定义

queue::value_type

- 元素型别
- 此和 `container::value_type` 相当。

queue::size_type

- 不带正负号的整数型别，用来表现大小。
- 此和 `container::size_type` 相当。

queue::container_type

- 内部容器的型别

各项操作 (Operations)

queue::queue ()

- 缺省构造函数
- 产生一个空的 `queue`。

explicit queue::queue (const Container& cont)

- 产生一个 `queue`，并以容器 `cont` 内的元素为初值。
- `cont` 内的所有元素均被复制。

size_type queue::size () const

- 返回元素个数。
- 如果要检验容器是否为空（亦即不含任何元素），应该使用 `empty()`，因为它可能更快。

```
bool queue::empty () const
```

- 判断 `queue` 是否为空 (亦即不含任何元素)。
- 和 `queue::size()==0` 等效, 但可能更快。

```
void queue::push (const value_type& elem)
```

- 将 `elem` 的副本安插到 `queue` 内, 并成为其新的最后元素。

```
value_type& queue::front ()
```

```
const value_type& queue::front () const
```

- 两种形式都返回 `queue` 的下一个元素。所谓“下一个元素”是指第一个 (亦即在其它所有元素之前) 被置入的元素。
- 调用者必须确保 `queue` 不为空 (`size()>0`), 否则可能导致未定义的行为。
- 第一形式是针对 `non-const queues` 而设计, 返回一个 `reference`。所以你可以就地 (*in place*) 修改 `queue` 内的元素。这样做是否合适? 由你自己决定。

```
value_type& queue::back ()
```

```
const value_type& queue::back () const
```

- 两种形式都返回 `queue` 的最后一个元素。所谓“最后一个元素”是指最后一个 (亦即在其它所有元素之后) 被插入的元素。
- 调用者必须确保 `queue` 不为空 (`size()>0`), 否则可能导致未定义的行为。
- 第一形式是针对 `non-const queues` 而设计, 返回一个 `reference`。所以你可以就地 (*in place*) 修改 `queue` 内的元素。这样做是否合适? 由你自己决定。

```
void queue::pop ()
```

- 移除 `queue` 内的下一个元素。所谓“下一个元素”是指第一个 (亦即在其它所有元素之前) 被插入的元素。
- 此函数没有返回值。如果想处理被移除的那个元素, 你必须先调用 `front()`。
- 调用者必须确保 `queue` 不为空 (`size()>0`), 否则可能导致未定义的行为。

```
bool comparison (const queue& queue1, const queue& queue2)
```

- 返回两个同型的 `queue` 的比较结果。

- **comparison** 可以是下面各运算之一：
 - operator ==
 - operator !=
 - operator <
 - operator >
 - operator <=
 - operator >=
- 如果两个 **queues** 的元素个数相等，且相同次序上的元素值也相等（也就是说所有对应元素之间的比较都得到 **true**），则这两个容器相等。
- **queues** 之间的大小比较是以“字典顺序”而定。参见 p360 对于算法 `lexicographical_compare()` 的描述。

10.2.4 一个使用者自定义的 Queue Class

标准的 `queue<>` class 将运行速度置于方便性和安全性之上。但我通常并不很重视那个。便自己写了一个 `queue` class，它有以下优势：

1. `pop()` 会返回下一元素。
2. 如果 `queue` 为空，`pop()` 和 `front()` 会抛出异常。

此外，我把一般人平常使用的那些成员函数如比较操作 (`comparison`) 和 `back()` 略去。我的 `queue` class 定义如下：

```
// cont/Queue.hpp
/* *****
 * Queue.hpp
 * - safer and more convenient queue class
 * *****/
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include <deque>
#include <exception>

template <class T>
class Queue {
protected:
    std::deque<T> c; // container for the elements

public:
    /* exception class for pop() and top() with empty queue
    */
```

```
class ReadEmptyQueue : public std::exception {
public:
    virtual const char* what() const throw() {
        return "read empty queue";
    }
};

// number of elements
typename std::deque<T>::size_type size() const {
    return c.size();
}

// is queue empty?
bool empty() const {
    return c.empty();
}

// insert element into the queue
void push (const T& elem) {
    c.push_back(elem);
}

// read element from the queue and return its value
T pop () {
    if (c.empty()) {
        throw ReadEmptyQueue();
    }
    T elem(c.front());
    c.pop_front();
    return elem;
}

// return value of next element
T& front () {
    if (c.empty()) {
        throw ReadEmptyQueue();
    }
    return c.front();
}
};

#endif /* QUEUE_HPP */
```

改用这个 `queue`，则先前的例子可以改写如下：

```
// cont/queue2.cpp

#include <iostream>
#include <string>
#include "Queue.hpp"      // use special queue class
using namespace std;

int main()
{
    try {
        Queue<string> q;

        // insert three elements into the queue
        q.push("These ");
        q.push("are ");
        q.push("more than ");

        // read and print two elements from the queue
        cout << q.pop();
        cout << q.pop();

        // push two new elements
        q.push("four ");
        q.push("words!");

        // skip one element
        q.pop();

        // read and print two elements from the queue
        cout << q.pop();
        cout << q.pop() << endl;
    }
}
```

```
// print number of remaining elements
cout << "number of elements in the queue: " << q.size()
    << endl;

// read and print one element
cout << q.pop() << endl;
}
catch (const exception& e) {
    cerr << "EXCEPTION: " << e.what() << endl;
}
}
```

最后新增的一个 `pop()` 调用操作是为了刻意引发错误。和标准 `queue` class 不同的是，我这个版本会抛出异常，而不引发未定义行为。程序输出如下：

```
These are four words!
number of elements in the queue: 0
EXCEPTION: read empty queue
```

10.3 Priority Queues (优先队列)

`class priority_queue<>` 实作出一个 `queue`，其中的元素根据优先级被读取。它的接口和 `queues` 非常相近。也就是说，`push()` 对着 `queue` 置入一个元素，`top()/pop()` 存取/移除下一个元素（图 10.5）。然而这所谓“下一个元素”并非第一个置入的元素，而是“优先级最高”的元素。换句话说 `priority queue` 内的元素已经根据其值进行了排序。和往常一样，你可以透过 `template` 参数指定一个排序准则。缺省的排序准则是利用 `operator<` 形成降序排列，那么所谓“下一个元素”就是“数值最大的元素”。如果同时存在若干个数值最大的元素，无法确知究竟哪一个会入选。

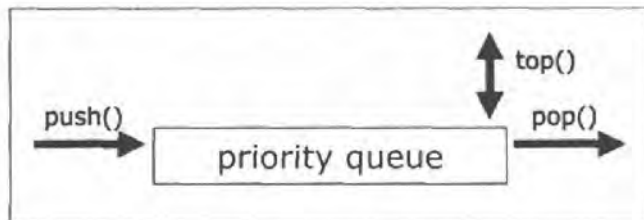


图 10.5 Priority Queue 的接口

`priority queue` 和一般的 `queue` 都定义于头文件 `<queue>`⁵;

```
#include <queue>
```

在头文件 `<queue>` 中, `class priority_queue` 定义如下:

```
namespace std {  
    template <class T,  
              class Container = vector<T>,  
              class Compare = less<typename Container::value_type>  
    class priority_queue;  
}
```

第一个 `template` 参数是元素型别, 带有默认值的第二个 `template` 参数定义了 `priority queue` 内部用来存放元素的容器。缺省容器是 `vector`。带有默认值的第三个 `template` 参数定义出“用以搜寻下一个最高优先元素”的排序准则。缺省情况下是以 `operator<` 作为比较标准。

下面这个例子定义了一个元素型别为 `float` 的 `priority queue`⁶:

```
std::priority_queue<float> pbuffer; // priority queue for floats
```

实际上 `priority queue` 只是很单纯地把各项操作转化为内部容器的对应调用。你可以使用任何序列式容器来支持 `priority queue`, 只要它们支持随机存取迭代器和 `front()`, `push_back()`, `pop_back()` 等动作就行。由于 `priority queue` 需要用到 STL `heap` 算法 (参见 9.9.4 节, p406), 所以其内部容器必须支持随机存取迭代器。例如你可以使用 `deque` 来容纳元素:

```
std::priority_queue< float, std::deque<float> > pbuffer;
```

如果需要定义自己的排序准则, 就必须传递一个函数 (或仿函数) 作为二元判断式 (`binary predicate`), 用以比较两个元素并以此作为排序准则 (关于排序准则的更多介绍, 请见 6.5.2 节, p178 和 8.1.1 节, p294)。例如以下式子定义了一个逆向的 (降序的) `priority queue`:

```
std::priority_queue<float, std::vector<float>,  
                  std::greater<float> > pbuffer;
```

在此 `priority queue` 中, “下一个元素”始终拥有最小元素值。

⁵ 早期的 STL 中, `priority queue` 被定义于 `<stack.h>`

⁶ 早期的 STL 中, 你必须以 `template` 参数传入内部容器。所以一个元素型别为 `float` 的 `priority queue` 声明如下:

```
priority_queue< vector<float>, less<float> > buffer;
```

10.3.1 核心接口

priority queue 的核心接口主要由成员函数 `push()`、`top()`、`pop()` 组成:

- `push()` 会将一个元素置入 priority queue 中。
- `top()` 会返回 priority queue 中的“下一个元素”。
- `pop()` 会从 priority queue 中移除一个元素。

和其它容器配接器一样, `pop()` 会移除下一个元素, 但是不返回它; `top()` 会返回下一个元素, 但并不移除它。所以如果你想移除 priority queue 内的下一个元素, 又想处理它, 就得同时调用这两个函数。注意, 如果 priority queue 内没有元素, 执行 `top()` 和 `pop()` 会导致未定义行为。你可以采用成员函数 `size()` 和 `empty()` 来检验容器是否为空。

10.3.2 Priority Queues 运用实例

以下程序展示了 `class priority_queue<>` 的用法:

```
// cont/pqueue1.cpp

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

int main()
{
    priority_queue<float> q;

    // insert three elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // read and print two elements
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // insert three more elements
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
```



```

    // skip one element
    q.pop();

    // pop and print remaining elements
    while (!q.empty()) {
        cout << q.top() << ' ';
        q.pop();
    }
    cout << endl;
}

```

程序输出如下:

```

66.6 44.4
33.3 22.2 11.1

```

由上可见,在元素 66.6, 22.2, 44.4 被置入后,程序打印出优先级最高的元素 66.6 和 44.4。另三个元素被置入后, priority queue 内含 22.2, 11.1, 55.5, 33.3 (按照插入次序)。下一个元素被 pop() 忽略掉了,所以最后一个循环依次打印 33.3, 22.2, 11.1。

10.3.3 Class priority_queue<> 细部讨论

和 stack<> 以及 queue<> 一样, priority_queue<> 的大部分操作都非常清晰明确,足以顾名思义:

```

namespace std {
    template <class T, class Container = vector<T>,
              class Compare = less<typename Container::value_type> >
    class priority_queue {
    public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef Container container_type;

    protected:
        Compare comp;    // sorting criterion
        Container c;     // container

    public:
        // constructors
        explicit priority_queue(const Compare& cmp = Compare(),
                               const Container& cont = Container())
            : comp(cmp), c(cont) {
            make_heap(c.begin(), c.end(), comp);
        }
    }
}

```

```
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last,
               const Compare& cmp = Compare(),
               const Container& cont = Container())
: comp(cmp), c(cont) {
    c.insert(c.end(), first, last);
    make_heap(c.begin(), c.end(), comp);
}

void push(const value_type& x); {
    c.push_back(x);
    push_heap(c.begin(), c.end(), comp);
}

void pop() {
    pop_heap(c.begin(), c.end(), comp);
    c.pop_back();
}

bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
const value_type& top() const { return c.front(); }
};
}
```

可见 `priority_queue` 使用了 STL `heap` 算法, 这些算法在 9.9.4 节, p406 中介绍过。注意, 和前述另两个容器配接器不同的是, 这里没有比较 (`comparison`) 操作符。

下面是各个成员的详细说明。

型别定义

`priority_queue::value_type`

- 元素型别
- 此和 `container::value_type` 相当。

`priority_queue::size_type`

- 不带正负号的整数型别，用来表现大小。
- 此和 `container::size_type` 相当。

`priority_queue::container_type`

- 内部容器的型别

构造函数

`priority_queue::priority_queue ()`

- default 构造函数
- 产生一个空的 `priority queue`。

`explicit priority_queue::priority_queue (const CompFunc& op)`

- 产生一个 `priority queue`，以 `op` 为排序准则。
- 如何将排序准则当做构造函数参数传入，请见 p191 和 p213。

`priority_queue::priority_queue (const CompFunc& op,
const Container& cont)`

- 产生一个 `priority queue`，以 `op` 为排序准则，并以容器 `cont` 内的元素为初值。
- `cont` 中的所有元素都会被复制。

`priority_queue::priority_queue (InputIterator beg,
InputIterator end)`

- 产生一个 `priority queue`，以区间 `[beg, end]` 内的元素为初值。
- 此构造函数是一个 `template member` (见 p11)，也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此构造函数即可运行。

`priority_queue::priority_queue (InputIterator beg,
InputIterator end,
const CompFunc& op)`

- 产生一个 `priority queue`，以 `op` 为排序准则，并以区间 `[beg, end]` 内的元素为初值。
- 此构造函数是一个 `template member` (见 p11)，也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此构造函数即可运作。
- 如何将排序准则当做构造函数参数传入，请见 p191 和 p213。

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op,
                                const Container& cont)
```

- 产生一个 `priority queue`，以 `op` 为排序准则，并以区间 `[beg, end)` 内的元素以及容器 `cont` 内的元素为初值。
- 此构造函数是一个 `template member` (见 p11)，也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此一构造函数即可运作。

其它操作

```
size_type priority_queue::size () const
```

- 返回元素数量。
- 如果要检验容器是否为空 (亦即不含任何元素)，应使用 `empty()`，因为它可能更快。

```
bool priority_queue::empty () const
```

- 判断 `priority queue` 是否为空 (亦即不含任何元素)。
- 与 `priority_queue::size()==0` 等效，但可能更快。

```
void priority_queue::push (const value_type& elem)
```

- 将 `elem` 的副本安插到 `priority queue` 内。

```
const value_type& priority_queue::top () const
```

- 返回 `priority queue` 内的“下一个元素”。所谓下一个元素是指所有元素中数值最大的那个。如果同时存在若干相等的最大元素，则无法确知会返回哪一个。
- 调用者必须确保 `priority queue` 不为空 (`size()>0`)，否则导致未定义行为。

```
void priority_queue::pop ()
```

- 移除 `priority queue` 内的“下一个元素”。所谓下一个元素是指所有元素中数值最大的那个。如果同时存在若干相等的最大元素，则无法确知会返回哪一个。
- 这个函数无返回值。如果想处理“下一个元素”，必须先调用 `top()`。
- 调用者必须确保 `priority queue` 不为空 (`size()>0`)，否则导致未定义行为。

10.4 Bitsets

Bitsets 造出一个内含位 (bits) 或布尔 (boolean) 值且大小固定的 array。当你需要管理各式标志 (flags)，并以标志的任意组合来表现变量时，就可运用 bitsets。C 程序和传统 C++ 程序通常使用型别 long 来作为 bits array，再通过 &, |, ~ 等位操作符 (bit operators) 操作各个位。Class bitset 的优点在于可容纳任意个数的位 (译注：但不能动态改变)，并提供各项操作。例如你可以对某个特定位置赋值一个位，也可以将 bitsets 作为由 0 和 1 组成的序列，进行读写。

注意，你不可以改变 bitset 内位的数量，这个数量的具体值是由 template 参数决定的。如果你需要一个可变长度的位容器，可考虑使用 vector<bool> (参见 6.2.6 节, p158)。

Class bitset 定义于头文件 <bitset> 之中：

```
#include <bitset>
```

其中的 class bitset 是个 template class，有一个 template 参数，用来指定位的数量：

```
namespace std {  
    template <size_t Bits>  
    class bitset;  
}
```

在这里，template 参数并不是一个型别，而是一个不带正负号的整数（此一语言特性请参考 p10）。

注意，如果 template 参数不同，具现化所得的 template 型别就不同。换句话说你只能针对位个数相同的 bitsets 进行比较和组合。

10.4.1 Bitsets 运用实例

将 Bitsets 当做一组标志

第一个例子展示如何运用 bitsets 来管理一组标志。每个标志都有一个由枚举型别 (enum) 定义出来的值。该枚举值就表示位在 bitset 中的位置。举个例子，这些 bits 可以代表颜色，那么每一个枚举值都代表一种颜色。通过运用 bitset，你可以管理一个变量，其中包含颜色的任意组合：

```
// cont/bitset1.cpp  
  
#include <bitset>  
#include <iostream>  
using namespace std;
```

```
int main()
{
    /* enumeration type for the bits
    * - each bit represents a color
    */
    enum Color { red, yellow, green, blue, white, black, ...,
                numColors };

    // create bitset for all bits/colors
    bitset<numColors> usedColors;

    // set bits for two colors
    usedColors.set(red);
    usedColors.set(blue);

    // print some bitset data
    cout << "bitfield of used colors: " << usedColors
          << endl;
    cout << "number of used colors: " << usedColors.count()
          << endl;
    cout << "bitfield of unused colors: " << ~usedColors
          << endl;

    // if any color is used
    if (usedColors.any()) {
        // loop over all colors
        for (int c = 0; c < numColors; ++c) {
            // if the actual color is used
            if (usedColors[(Color)c]) {
                ...
            }
        }
    }
}
```


此例中的：

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

将数值 267 转换成一个 `bitset`，其位数与型别 `unsigned short` 相符（关于数值极限的讨论，请见 4.3 节，p60）。针对 `bitset` 而设计的 `output` 操作符（`operator<<`）能够将这组位翻译成一个“0/1 序列”并打印出来。

同样道理，以下式子：

```
bitset<100>(string("1000101011"))
```

将一个二进制字符透过 `to_ulong()` 转化为一个整数值，然后转化至一个 `bitset`。注意，`bitset` 内的位数应小于 `sizeof(unsigned long)`，因为如果该 `bitset` 无法按照 `unsigned long` 来表现这个值，它会抛出一个异常⁷。

10.4.2 Class `bitset` 细部讨论

`bitset` class 提供了以下操作。

生成，拷贝和销毁

`Bitsets` 定义了一些特殊的构造函数，但是没有定义特殊的 `copy` 构造函数、`assignment` 操作符和析构函数。所以 `bitsets` 的赋值和复制行为都是采用缺省行为，也就是按位次序逐一拷贝（所谓 `bitwise copy`）。

```
bitset<bits>::bitset ()
```

- default 构造函数。
- 生成一个 `bitset`，所有位均初始化为零。
- 例如：

```
bitset<50> flags;    // flags: 0000...000000
                    // thus, 50 unset bits
```

```
bitset<bits>::bitset (unsigned long value)
```

- 产生一个 `bitset`，以整数值 `value` 的位作为初值。
- 如果 `value` 的值太小，前面不足的位被设为 0。
- 例如：

```
bitset<50> flags(7);    // flags: 0000...000111
```

⁷ 注意，你必须先将这个初始值显式转化为 `string`。这恐怕是标准规格中的一个失误，因为标准规格早期版本允许我们这么做：

```
bitset<100>("1000101011")
```

但是此构造函数被用于其它不同的 `string` 型别时，会意外导致隐式型别转换被排除。这个问题目前已经在标准委员会中有了解决提案。


```
explicit bitset<bits>::bitset (const string& str)
bitset<bits>::bitset (const string& str, string::size_type str_idx)
bitset<bits>::bitset (const string& str, string::size_type str_idx,
                      string::size_type str_num)
```

- 所有形式都用来产生 `bitset`，以字符串 `str` 或其子字符串加以初始化。
- 该字符串或子字符串中只能包含字符 "0" 和 "1"。
- `str_idx` 是 `str` 中用于初始化的第一个字符。
- 如果省略 `str_num`，从 `str_idx` 开始到 `str` 结束的所有字符都将用于初始化。
- 如果该字符串或子字符串中的字符数量少于所需，则前面多余的位将设初值 0。
- 如果该字符串的子字符串中的字符数量多于所需，则多出来的字符会被忽略。
- 如果 `str_idx > str.size()`，抛出 `out_of_range` 异常。
- 如果有个字符既不是 "0" 也不是 "1"，抛出 `invalid_argument` 异常。
- 这个构造函数是个 `template member` (参见 p11)。因此就第一参数而言，不会发生从 `const char*` 向 `string` 的隐式型别转换⁸。
- 举个例子：

```
bitset<50> flags(string("1010101"));
// flags: 0000...0001010101
bitset<50> flags(string("1111000"),2,3);
// flags: 0000...0000000110
```

非更易性操作 (Nonmanipulating Operations)

```
size_t bitset<bits>::size () const
```

- 返回位的个数。

```
size_t bitset<bits>::count () const
```

- 返回 “位值为 1” 的位个数。

```
bool bitset<bits>::any () const
```

- 判断是否有任何位被设立 (数值为 1)。

```
bool bitset<bits>::none () const
```

- 判断是否没有任何一个位被设立 (亦即所有的位值皆为 0)。

⁸ 这可能是标准规格中的一个失误，因为早期标准规格允许我们这么做：

```
bitset<50> flags("1010101")
```

但是此一构造函数被用于其它不同的 `string` 型别时，会意外导致隐式型别转换被排除。这个问题目前已经在标准委员会中有了解决提案。

```
bool bitset<bits>::test (size_t idx) const
```

- 判断 *idx* 位置上的位是否被设立（数值为 1）。
- 如果 *idx* >= *size()* 则抛出 *out_of_range* 异常。

```
bool bitset<bits>::operator == (const bitset<bits>& bits) const
```

- 判断 **this* 和 *bits* 的所有位是否都相等。

```
bool bitset<bits>::operator != (const bitset<bits>& bits) const
```

- 判断 **this* 和 *bits* 之中是否有些位不相等。

更易性操作（manipulating Operations）

```
bitset<bits>& bitset<bits>::set ()
```

- 将所有位设为 *true*（1）。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::set (size_t idx)
```

- 将位置 *idx* 上的位设为 *true*（1）。
- 返回更动后的 *bitset*。
- 如果 *idx* >= *size()* 则抛出 *out_of_range* 异常。

```
bitset<bits>& bitset<bits>::set (size_t idx, int value)
```

- 根据 *value* 上的值设定 *idx* 位置的位值。
- 返回更动后的 *bitset*。
- *value* 将被当做 *boolean* 值处理。如果 *value* 等于 0，则 *idx* 位置上的位值被设为 *false*；其它的 *value* 值都会使该位被设为 *true*。
- 如果 *idx* >= *size()* 则抛出 *out_of_range* 异常。

```
bitset<bits>& bitset<bits>::reset ()
```

- 将所有位设为 *false*（0）。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::reset (size_t idx)
```

- 将位置 *idx* 上的位设为 *false*（0）。
- 返回更动后的 *bitset*。
- 如果 *idx* >= *size()* 则抛出 *out_of_range* 异常。

```
bitset<bits>& bitset<bits>::flip ()
```

- 反转所有位（原本 1 者转为 0，原本 0 者转为 1）。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::flip (size_t idx)
```

- 反转 *idx* 位置上的位。
- 返回更动后的 *bitset*。
- 如果 *idx* >= *size()* 则抛出 *out_of_range* 异常。

```
bitset<bits>& bitset<bits>::operator ^= (const bitset<bits>& bits)
```

- 对每个位逐一进行 exclusive-or 运算。
- 将 *this 之中所有和 “*bits* 内数值为 1 的位” 的对应位都翻转，其它位保持不动。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::operator |= (const bitset<bits>& bits)
```

- 位逐一进行 or 运算。
- 将 *this 之中所有和 “*bits* 内数值为 1 的位” 的对应位都设为 1，其它位保持不动。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::operator &= (const bitset<bits>& bits)
```

- 位逐一进行 and 运算。
- 将 *this 之中所有和 “*bits* 内数值为 0 的位” 的对应位都设为 0，其它位保持不动。
- 返回更动后的 *bitset*。

```
bitset<bits>& bitset<bits>::operator <<= (size_t num)
```

- 将所有位向左移动 *num* 个位置。
- 返回更动后的 *bitset*。
- 空出来的位设为 false (0)。

```
bitset<bits>& bitset<bits>::operator >>= (size_t num)
```

- 将所有位向右移动 *num* 个位置。
- 返回更动后的 *bitset*。
- 空出来的位设为 false (0)。

使用 Operator[] 存取位

```
bitset<bits>::reference bitset<bits>::operator[] (size_t idx)
```

```
bool bitset<bits>::operator[] (size_t idx) const
```

- 这两种形式都返回 *idx* 位置上的位值。
- 第一种形式针对 non-const bitsets，使用了一个 proxy (代理人、替身) 型别，使得返回值成为一个可被更动的值 (左值, lvalue)。详见下一段文字叙述。
- 调用者必须确保 *idx* 有效，否则会导致未定义的行为。

当我们针对 `non-const bitsets` 调用 `operator[]` 时，返回的是一个特殊的临时对象 `bitset<>::reference`。这个对象称为 **proxy**（代理人，替身）⁹，它的存在使我们得以对 `operator[]` 所存取的位做适当的内容变动（译注：**proxy** 是一种设计模式，可参考《*Design Patterns*》或《*More Effective C++*》条款 30）。

具体地说，对于上述 `references`，允许以下五种操作：

1. `reference& operator= (bool)`
根据传来的值设置该位。
2. `reference& operator= (const reference&)`
根据另一个 `reference` 的内容设定该位。
3. `reference& flip ()`
翻转该位。
4. `operator bool () const`
（自动地）将该位转换为布尔值。
5. `bool operator~ () const`
返回该位的补码（翻转值）。

举个例子，你可以这么做：

```
bitset<50> flags;
...
flags[42] = true;           // set bit 42
flags[13] = flags[42];      // assign value of bit 42 to bit 13
flags[42].flip();           // toggle value of bit 42
if (flags[13]) {            // if bit 13 is set,
    flags[10] = ~flags[42]; // then assign complement of bit 42 to bit
}
```

产生新的（经修改的）`bitset`

```
bitset<bits> bitset<bits>::operator ~ () const
```

- 产生一个新的 `bitset` 并返回；以 `*this` 的位翻转值作为初值。

```
bitset<bits> bitset<bits>::operator << (size_t num) const
```

- 产生一个新的 `bitset` 并返回；以 `*this` 的位向左移动 `num` 个位置作为初值。

```
bitset<bits> bitset<bits>::operator >> (size_t num) const
```

- 产生一个新的 `bitset` 并返回；以 `*this` 的位向右移动 `num` 个位置作为初值。

⁹ **proxy**（替身，代理人）使你能对通常力不能及的问题加以控制。用来获取更高的安全性。本例中虽然返回值理论上如同 `bool`，但 **proxy** 使我们得以进行某些特殊操作。

```
bitset<bits> operator & (const bitset<bits>& bits1,
                        const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 and 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 和 *bits2* 的位值都为 1”的那些位才会被设为 1。

```
bitset<bits> operator | (const bitset<bits>& bits1,
                        const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 or 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 或 *bits2* 的位值为 1”的那些位才会被设为 1。

```
bitset<bits> operator ^ (const bitset<bits>& bits1,
                        const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 exclusive or 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 或 *bits2* 的位值相异”的那些位才会被设为 1。

型别转换

```
unsigned long bitset<bits>::to_ulong () const
```

- 返回 bitset 所有位所代表的整数
- 如果 unsigned long 不足以表现这个整数，抛出 `overflow_error` 异常。

```
string bitset<bits>::to_string () const
```

- 返回一个 string，以字符串形式表现该 bitset 的二进制值（不是 0 就是 1）。
- 字符顺序按照 bitset 的索引高低排列。
- 这是一个 template 函数，其返回值型别被参数化了。根据 C++ 语言规则，你必须这么使用：

```
bitset<50> b;
...
b.template to_string<char, char_traits<char>, allocator<char> >()
```

I/O 操作

```
istream& operator >> (istream& strm, bitset<bits>& bits)
```

- 将一个包含 '0' 和 '1' 的字符序列转换为对应位，读入 *bits*。
- 读取行为一直进行下去，直到发生以下数种情况之一：
 - 读取结束（多半是这种情况）。
 - *strm* 中出现 *end-of-file* 符号。
 - 下一个字符既不是 '0' 也不是 '1'。

- 返回 *strm*。
- 如果读入的位少于 *bitset* 的位数量，前面不足的位填 0。
- 如果此一操作无法读取任何字符，则 *strm* 的 `ios::failbit` 会被设立，导致相关异常被抛出来（参见 13.4.4 节, p602）。

`ostream& operator<< (ostream& strm, const bitset<bits>& bits)`

- 将 *bits* 的二进制形式转换为字符串（成为一个包含 '0' 和 '1' 的序列）。
- 使用 `to_string()` 来产生输出字符（参见 p468）。
- 返回 *strm*。
- 参见 p462 的范例。

11

Strings

字符串

本章讲述 C++ 标准程序库中的 *string*（字符串）型别，包括针对基本 `template class basic_string<>` 及其标准特化型别 `string` 和 `wstring` 的详细内容。

String 可能是困惑的根源，因为这个字眼的确切含义比较模糊，可能是指一个型别为 `char*`（亦可加上 `const` 饰词）的字符数组（character array），也可能是 `class string` 的一个实体，或泛指代表字符串的某个对象。本章之中我对术语 “*string*” 的定义是：C++ 标准程序库中某个字符串型别（`string` 或 `wstring`）的对象。至于一般字符串，也就是 `char*` 或 `const char*`，我用的术语是 “*C-string*”。

注意，字符串字面常数（例如 “hello”）会被转为 `const char*`。为了向下兼容，它们可以被隐式转换为 `char*`，不过这种转换并不值得赞赏。

11.1 动机

C++ 标准程序库中的 *string* class 使你可以将 *string* 当做一个一般型别而不会令用户感觉有任何问题。你可以像对待基本型别那样地复制、赋值和比较 *string*，再也不必担心内存是否足够、占用的内存实际长度等问题，只需运用操作符操作函数即可，例如以 `=` 进行赋值动作，以 `==` 进行比较动作，以 `+` 进行串联动作。简而言之，C++ 标准程序库对于 *string* 的设计思维就是，让它的行为尽可能像基本型别，不会在操作上引起什么麻烦（至少原则如此）。现今世界的数据处理大部分就是字符串处理，所以对于从 C, Fortran 或类似语言一路走来的程序员而言，这是非常重要的进步，因为 *string* 在那些语言中往往是烦恼之源。

下面数节给了两个例子，展示 *string* class 的能力和用法。不以实用为目的，而是以示范为目的。

11.1.1 例一：引出一个临时文件名

第一个例子通过命令行 (command line) 参数产生一个临时文件名。如果你这么启动该程序：

```
string1 prog.dat mydir hello. oops.tmp end.dat
```

输出如下：

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

通常产生的扩展名是.tmp，但如果原本的扩展名就是.tmp，则换成.xxx。

程序如下：

```
// string/string1.cpp

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

int main (int argc, char* argv[])
{
    string filename, basename, extname, tmpname;
    const string suffix("tmp");

    /* for each command-line argument
     * (which is an ordinary C-string)
     */
    for (int i=1; i<argc; ++i) {
        // process argument as file name
        filename = argv[i];

        // search period in file name
        string::size_type idx = filename.find('.');
        if (idx == string::npos) {
            // file name does not contain any period
            tmpname = filename + '.' + suffix;
        }
        else {
```

```

    /* split file name into base name and extension
    * - base name contains all characters before the period
    * - extension contains all characters after the period
    */
    basename = filename.substr(0, idx);
    extname = filename.substr(idx+1);
    if (extname.empty()) {
        // contains period but no extension: append tmp
        tmpname = filename;
        tmpname += suffix;
    }
    else if (extname == suffix) {
        // replace extension tmp with xxx
        tmpname = filename;
        tmpname.replace (idx+1, extname.size(), "xxx");
    }
    else {
        // replace any extension with tmp
        tmpname = filename;
        tmpname.replace (idx+1, string::npos, suffix);
    }
}

// print file name and temporary name
cout << filename << " ==> " << tmpname << endl;
}
}

```

首先:

```
#include <string>
```

用来将定义有标准 C++ *string* 类别(s) 的头文件含入。一如惯例, 这些类别被声明在 `std` 命名空间中。

以下声明式会产生四个 *string* 变量:

```
string filename, basename, extname, tmpname;
```

既然没有传入参数, 它们均使用 *string* 的 `default` 构造函数, 这会使它们被初始化为空字符串。

以下声明式产生一个 *string* 常数 `suffix`:

```
const string suffix("tmp");
```

在本程序中，它被用来作为临时文件名的正常后缀。此字符串以一个 *C-string* 初始化，其值为 `tmp`。注意，在几乎所有“两个 `strings` 对象的组合操作”场合中，你可以改而使用“一个 *C-string* 和一个 `string` 对象”来进行组合操作。例如在这个程序里，你可以直接以 `"tmp"` 取代所有的 `suffix`。

每当 `for` 循环被执行一次，其中的语句：

```
filename = argv[i];
```

对字符串变量 `filename` 赋予一个新值。本例中这个新值是个 *C-string*，不过你也可以把另一个 `string` 对象或型别为 `char` 的单个字符赋值给它。

以下语句：

```
string::size_type idx = filename.find('.');
```

在字符串 `filename` 中搜寻第一个 `'.'` 字符。有好几个函数可以在字符串内实施搜寻功能，函数 `find()` 是其中之一。你也可以从后向前搜寻，或是搜寻子字符串，或是在字符串的某个范围内搜寻，或是同时搜寻数个字符。所有这些搜寻函数都返回第一个匹配位置（一个索引）。没错，返回值是个整数，而不是迭代器。字符串的一般接口并不依赖 `STL` 概念。然而字符串的确也提供了数种迭代器（参见 11.2.12 节，p497）。所有搜寻函数的返回型别都是 `string::size_type`，这是 `string` class 定义的一个无正负号整数型别¹。当然啦，第一个字符的索引值为 0，最后一个字符的索引值是 `numberOfCharacters-1`。注意，`numberOfCharacters` 并不是一个有效索引。和 *C-string* 不同，`string` 对象的字符串尾部并没有一个特殊字符 `'\0'`。

如果搜寻失败，必须返回一个特殊值来表示，该值就是 `npos`，定义于 `string` class 中。所以下面这一行可用来检验搜寻动作是否失败：

```
if (idx == string::npos)
```

请特别注意，当你打算检验搜寻函数的返回值时，应该使用 `string::size_type` 型别而不是 `int` 或 `unsigned`。否则上述与 `string::npos` 的比较动作将无法有效运行。细节见 11.2.12 节，p495。

本例如果搜寻失败，表示该文件没有扩展名。此时，文件名应该是原文件名之后接上一个句点，再接上先前定义的后缀：

```
tmpname = filename + '.' + suffix;
```

你可以用 `operator+` 直接串连两个字符串。也可以将字符串与 *C-string* 以及单个字符串串联起来。

¹ 更明确地说，字符串的 `size_type` 型别系根据字符串类别的内存模型而定。见 11.3.12 节，p526。

如果找到 '.', else 部分就要发挥作用了。这里, 句点所在的位置 (索引) 用来将文件名分割成主文件名和扩展名, 由成员函数 `substr()` 完成:

```
basename = filename.substr(0, idx);
extname = filename.substr(idx+1);
```

`substr()` 的第一参数是起点索引, 可有可无的第二参数是字符个数 (而非终点索引)。如果没有指定第二参数, 那么所有剩余字符都将被视为子字符串返回。

凡是“以一个索引和一个长度作为参数”的地方, 字符串行为遵循下面两项规则:

1. 索引值必须合法。该值必须小于字符串的字符个数 (第一个字符的索引是 0)。最后一个字符的下一个位置 (的索引值) 可用来标明结束位置。

大部分情况下, 如果你指定的索引超过实际字符数, 会引发 `out_of_range` 异常。不过, 用以搜寻单一字符或某个位置的所有搜寻函数, 均可接受任意索引。如果索引超过实际字符数, 这些函数会返回 `string::npos` (表示没找到)。

2. 字符数量 (长度) 可为任意值。如果其值大于实际剩余的字符数, 则这些剩余字符都会被用到。如果使用 `string::npos`, 相当于指明“剩余所有字符”。

所以, 如果找不到 '.', 以下表达式会抛出一个异常:

```
filename.substr(filename.find('.'))
```

但如果找不到 '.', 以下表达式不会抛出异常:

```
filename.substr(0, filename.find('.'))
```

而是会返回整个文件名。

即使找到了 '.', 如果其后没有任何字符, `substr()` 返回的扩展名为空。下面这个式子对这种情况实施检查:

```
if (extname.empty())
```

如果这个条件式得到 `true`, 产生的临时文件名将由其主文件名加上一般扩展名组成:

```
tmpname = filename;
tmpname += suffix;
```

这里, `operator+=` 的作用是在尾端附加扩展名。

原扩展名可能就是临时文件的标准扩展名。我们可以用 `operator==` 来比较这两个字符串:

```
if (extname == suffix)
```

如果比较结果是 `true`, 则以 "xxx" 作为临时扩展名:

```
tmpname = filename;
tmpname.replace (idx+1, extname.size(), "xxx");
```

其中的:

```
extname.size()
```

返回字符串 `extname` 的字符个数。你也可以使用 `length()` 获得相同结果, 其行为和 `size()` 完全一样。是的, `size()` 和 `length()` 都返回字符串的字符个数。注意, `size()` 的结果和字符串实际占用的内存无关²。

当所有特殊情况都照顾到了之后, 接下来就可以进行常规处理了。程序以标准扩展名替换原文件的扩展名:

```
tmpname = filename;  
tmpname.replace (idx+1, string::npos, suffix);
```

此处的 `string::npos` 表示“剩余的所有字符”。所以句点之后的所有字符被替换为 `suffix`。这一替换对于“句号之后为空字符串”的文件名同样有效, 相当于以 `suffix` 来替换“无物”。

以下语句输出原文件名和新产生的临时文件名。看, 你可以使用一般的 `stream output` 操作符来打印字符串 (惊讶吧):

```
cout << filename << " => " << tmpname << endl;
```

11.1.2 例二: 引出一段文字并逆向打印

第二个例子从标准输入装置取得一个个英文单词, 然后将其中各个字符 (字母) 逆序印出。单词和单词之间以一般空格符 (换行符号 `newline`、空格符 `space` 或定位符号 `tab`) 或逗号、句号、分号分隔开来。

```
// string/string2.cpp  
  
#include <iostream>  
#include <string>  
using namespace std;  
  
int main (int argc, char** argv)  
{  
    const string delims(" \\t,.;");  
    string line;  
  
    // for every line read successfully  
    while (getline(cin, line)) {
```

² 这里出现的两个成员函数依不同的设计原则执行了相同的动作。`length()` 传回字符串长度, 就好像 *C-strings* 以 `strlen()` 所得结果一样。`size()` 则是根据 STL 习惯而设的成员函数, 用来表明元素数量。

```
    string::size_type begIdx, endIdx;

    // search beginning of the first word
    begIdx = line.find_first_not_of(delims);

    // while beginning of a word found
    while (begIdx != string::npos) {
        // search end of the actual word
        endIdx = line.find_first_of (delims, begIdx);
        if (endIdx == string::npos) {
            // end of word is end of line
            endIdx = line.length();
        }

        // print characters in reverse order
        for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
            cout << line[i];
        }
        cout << ' ';

        // search beginning of the next word
        begIdx = line.find_first_not_of (delims, endIdx);
    }
    cout << endl;
}
```

本程序中，所有间隔字符被定义于一个特殊的字符串常数中：

```
const string delims(" \t,.;");
```

换行符号也是一个间隔字符，但这里不必特别在意，因为程序本身就是一行一行地读取标准输入装置。

外层循环不断地将新行读入字符串 `line` 之中：

```
string line;
while (getline(cin, line)) {
    ...
}
```

`getline()` 是一个可以从 `stream` 读取字符串的特殊函数，它逐字读取，直到一行结束（通常以换行符号作为标示）。换行符号可以由你自定，你可以将自己喜欢的换行符号作为第三参数传入，令 `getline()` 在该符号所区隔出来的各个语汇单元（`token`）中一一读取。

在外层循环内，各个单词分别被搜寻和打印。第一个语句：

```
begIdx = line.find_first_not_of(delims);
```

搜寻第一个单词的起始位置。函数 `find_first_not_of()` 返回“不隶属参数所指字符串”的第一个字符的索引，所以本例返回第一个“分隔符（`delims`）以外”的字符。和一般搜寻函数一样，如果没有找到匹配字符，就返回 `string::npos`。

如果找到了一个单字，就进入内层循环：

```
while (begIdx != string::npos) {  
    ...  
}
```

内层循环的第一个语句用来搜寻当前单词的结尾：

```
endIdx = line.find_first_of (delims, begIdx);
```

函数 `find_first_of()` 用来搜寻“第一参数所指字符串内的任何字符”的第一次出现位置。可有可无的（可选的）第二参数用来标示搜寻起点。上述动作会找到单词后的第一个分隔符。

如果没找到，就将 `endIdx` 设定为“行结束标记（*end-of-line*）”：

```
if (endIdx == string::npos) {  
    endIdx = line.length();  
}
```

这里使用 `length()`，效果和 `size()` 相同：返回字符个数。

以下语句将所有字符逆序打印出来：

```
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {  
    cout << line[i];  
}
```

是的，你可以使用 `operator[]` 访问字符串的个别字符。请注意，这个操作符并不检查索引是否合法。所以你必须确保索引的合法性（一如上例）。比较安全的做法是利用成员函数 `at()` 来存取字符。不过这种方式会带来效率上的负担，所以一般还是采取不检查态度。

另一个讨厌的问题由字符串索引造成。如果你忘记将 `begIdx` 转型为 `int`，程序可能会陷入无穷循环中，甚至崩溃掉。和第一个例子一样，这里的 `string::size_type` 是一个不带正负号的整数型别。如果不转型，当带正负号的 `i` 与不带正负号的型别进行比较时，`i` 会被自动转型为无正负号值，于是如果目前处

理的这个单字位于一行的开头，则表达式：

```
i >= begIdx
```

的结果永远为 `true`——因为 `begIdx` 为 0，而任何无正负号值都大于等于 0。这将引发无穷循环，直到存取某个非法地址而崩溃。

因为这个缘故，我并不欣赏 `string::size_type` 和 `string::npos` 的设计思维。11.2.12 节, p496 有一个比较安全的方案，但也并非完美。

内层循环中的最后一个语句重新将 `begIdx` 初始化，使它标示下一个单词的起点（如果有下一个单词的话）：

```
begIdx = line.find_first_not_of (delims, endIdx);
```

和先前对 `find_first_not_of()` 的调用不同，这里把上一个单词的终点当做搜寻起点。如果上一个单词是该行的最后一个单词，则 `endIdx` 就是该行的终点。这意味着搜寻将从字符串终点开始，结果当然是返回 `string::npos`。

让我们试试这个“有用而且重要”☺ 的程序。下面是输入：

```
pots & pans
I saw a reed
```

输出如下：

```
stop & snap
I was a deer
```

如果能找到更多有趣的例子，我很愿意在本书下一版收录进来。

11.2 String Classes 细部描述

11.2.1 String 的各种相关型别

表头文件

表头文件 `<string>` 定义了所有的字符串型别和函数。

```
#include <string>
```

一如既往，所有标识符都定义于命名空间 `std` 之中。

模板类别 (template class) `basic_string<>`

在 `<string>` 之中, `basic_string<>` 被定义为所有字符串型别的基本模板类别 (basic template class)：


```
namespace std {
    template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
        class basic_string;
}
```

此一类别将字符型别、字符型别特性 (traits)、内存模型 (memory model) 加以参数化:

- 第一参数是单个字符所属型别 (译注: 也许是 ASCII 字符或 Unicode 字符...)。
- 带默认值的第二参数是个特性类别 (traits class), 提供字符串类别中所有的字符核心操作。此种特性类别规定了“复制字符”或“比较字符”的做法 (详见 p687, 14.1.2 节), 如果没有指定它, 就会根据现有的字符型别采用缺省的特性类别。p503, 11.2.14 节实作出一个使用者自定的特性类别, 让字符串以“不分大小写”的方式进行各种操作。
- 带默认值的第三参数定义了字符串类别所采用的内存模式, 通常设定为“缺省的内存模型 allocator” (详见 p31, 3.4 节和第 15 章)³。

string 型别和 wstring 型别

C++ 标准程序库提供了两个 `basic_string<>` 特化版本:

1. `string` 是针对 `char` 而预先定义的特化版本:

```
namespace std {
    typedef basic_string<char> string;
}
```

2. `wstring` 是针对 `wchar_t` 而预先定义的特化版本:

```
namespace std {
    typedef basic_string<wchar_t> wstring;
}
```

如此一来你就可以使用宽字符集, 例如 Unicode 或某些亚洲字符集 (国际化议题详见第 14 章)。

以下数节的讨论并不区分如上所述不同的字符串类型。由于所有字符串类型都采用相同接口, 所以用法和问题都一样。我将以 “*string*” 表示任何字符串型别, 包括 `string` 和 `wstring`。由于一般软件开发大多顺应欧美环境, 所以本书的例子大多采用 `string` 型别。

³ 如果你的系统不支持 default template parameters (缺省模板参数), 则第三参数通常会被省略。

11.2.2 操作函数 (Operations) 综览

表 11.1 列出针对字符串而设计的所有操作函数。

表 11.1 字符串的各种操作函数

操作函数 (Operation)	效果 (Effect)
构造函数 (<i>constructors</i>)	产生或复制字符串
析构函数 (<i>destructors</i>)	销毁字符串
<code>=, assign()</code>	赋以新值
<code>swap()</code>	交换两个字符串的内容
<code>+=, append(), push_back()</code>	添加字符
<code>insert()</code>	插入字符
<code>erase()</code>	删除字符
<code>clear()</code>	移除全部字符 (使之为空)
<code>resize()</code>	改变字符数量 (在尾端删除或添加字符)
<code>replace()</code>	替换字符
<code>+</code>	串联字符串
<code>==, !=, <, <=, >, >=, compare()</code>	比较字符串内容
<code>size(), length()</code>	返回字符数量
<code>max_size()</code>	返回字符的最大可能个数
<code>empty()</code>	判断字符串是否为空
<code>capacity()</code>	返回重新分配之前的字符容量
<code>reserve()</code>	保留一定量内存以容纳一定数量的字符
<code>[] , at()</code>	存取单一字符
<code>>>, getline()</code>	从 <i>stream</i> 中读取某值
<code><<</code>	将某值写入 <i>stream</i>
<code>copy()</code>	将内容复制为一个 C-string
<code>c_str()</code>	将内容以 C-string 形式返回
<code>data()</code>	将内容以字符数组 (<i>character array</i>) 形式返回
<code>substr()</code>	返回某个子字符串 (<i>substring</i>)
搜寻函数 (<i>find functions</i>)	搜寻某个子字符串或字符
<code>begin(), end()</code>	提供正常的 (正向) 迭代器支持
<code>rbegin(), rend()</code>	提供逆向迭代器支持
<code>get_allocator()</code>	返回配置器 (<i>allocator</i>)

字符串操作函数的参数

STL 提供了很多字符串操作函数。其中许多往往具有数个重载版本，分别以一个、两个或三个参数来指定新值。表 11.2 整理出所有字符串操作的参数规格。

表 11.2 字符串操作函数的参数规格

参数 (Arguments)	含义
<code>const string & str</code>	整个 <code>str</code> 字符串
<code>const string & str,</code> <code>size_type idx,</code> <code>size_type num</code>	大部分情况下是指字符串 <code>str</code> 中以 <code>idx</code> 开始的 <code>num</code> 个字符
<code>const char* cstr</code>	整个 C-string <code>cstr</code>
<code>const char* chars,</code> <code>size_type len</code>	字符数组 <code>chars</code> 中的 <code>len</code> 个字符
<code>char c</code>	字符 <code>c</code>
<code>size_type num, char c</code>	<code>num</code> 个字符 <code>c</code>
<code>iterator beg, iterator end</code>	区间 <code>[beg;end)</code> 内所有字符

注意，只有在单参数版本中，才将 `char*` 字符 `'\0'` 当做字符串结尾特殊符号来处理，其它所有情况下 `'\0'` 都不被视为特殊字符：

```
std::string s1("nico"); // initializes s1 with: 'n' 'i' 'c' 'o'
std::string s2("nico",5); // initializes s2 with: 'n' 'i' 'c' 'o' '\0'
std::string s3(5,'\0'); // initializes s3 with: '\0' '\0' '\0' '\0' '\0'
s1.length() // yields 4
s2.length() // yields 5
s3.length() // yields 5
```

因此，一般而言一个字符串内可以包含任何字符，甚至可以包含二进制文件的内容。

至于各个操作函数对应何种参数，请见表 11.3。所有操作符都只能把对象当做单一值来处理，因此如果要赋值、比较或添加 *string*（或 *C-string*）的一部分，就必须采用相应的函数。

未提供的操作函数

C++ 标准程序库的 *string* class 并没有解决所有可能遇到的字符串问题。事实上它并没有提供下列问题的解决之道：

- 正则表达式 (Regular expressions)
- 文本处理（例如大写化、大小写不计的字符串比较动作等等）

不过，文本处理方面的问题不大，参见 11.2.13 节, p497 的例子。

表 11.3 拥有字符串参数的各种操作函数

	Full String	Part of String	C-string (char*)	char Array	Single char	num chars	Iterator Range
<i>constructors</i>	Yes	Yes	Yes	Yes	—	Yes	Yes
=	Yes	—	Yes	—	Yes	—	—
assign()	Yes	Yes	Yes	Yes	—	Yes	Yes
+=	Yes	—	Yes	—	Yes	—	—
append()	Yes	Yes	Yes	Yes	—	Yes	Yes
push_back()	—	—	—	—	Yes	—	—
insert(), index version	Yes	Yes	Yes	Yes	—	Yes	—
insert(), iterator version	—	—	—	—	Yes	Yes	Yes
replace(), index version	Yes	Yes	Yes	Yes	Yes	Yes	—
replace(), iterator version	Yes	—	Yes	Yes	—	Yes	Yes
<i>find functions</i>	Yes	—	Yes	Yes	Yes	—	—
+	Yes	—	Yes	—	Yes	—	—
==, !=, <, <=, >, >=	Yes	—	Yes	—	—	—	—
compare()	Yes	Yes	Yes	Yes	—	—	—

11.2.3 构造函数和析构函数 (Constructors and Destructors)

表 11.4 列出 *strings* 的所有构造函数和析构函数。本节将一一介绍它们。“利用迭代器指出的区间进行初始化”的构造函数将在第 11.2.13 节, p497 介绍。

表 11.4 *strings* 的构造函数和析构函数

表达式	效果
<i>strings</i>	生成一个空字符串 <i>s</i>
<i>string s(str)</i>	copy 构造函数, 生成字符串 <i>str</i> 的一个复制品
<i>string s(str, stridx)</i>	将字符串 <i>str</i> 内“始于位置 <i>stridx</i> ”的部分, 当做字符串 <i>s</i> 的初值。
<i>string s(str, stridx, strlen)</i>	将字符串 <i>str</i> 内“始于位置 <i>stridx</i> 且长度顶多 <i>strlen</i> ”的部分, 当做字符串 <i>s</i> 的初值。
<i>string s(cstr)</i>	以 C-string <i>cstr</i> 作为字符串 <i>s</i> 的初值。
<i>string s(chars, chars_len)</i>	以 C-string <i>cstr</i> 的前 <i>chars_len</i> 个字符作为字符串 <i>s</i> 的初值。
<i>string s(num, c)</i>	生成一个字符串, 包含 <i>num</i> 个 <i>c</i> 字符
<i>string s(beg, end)</i>	以区间 [<i>beg</i> , <i>end</i>] 内的字符作为字符串 <i>s</i> 的初值。
<i>s.~string()</i>	销毁所有字符, 释放内存

注意，你不能以单一字符来初始化某个字符串，但是你可以这么做：

```
std::string s('x');           // ERROR
std::string s(1, 'x');       // OK, creates a string that has one character 'x'
```

这表示编译器提供了一个从 `const char*` 到 `string` 的自动型别转换功能，但不存在一个从 `char` 到 `string` 的自动型别转换功能。

11.2.4 Strings 和 C-Strings

C++ *Standard* 将字符串字面常数（string literals）的型别由 `char*` 改为 `const char*`。为了提供向下兼容性，C++ *Standard* 规定了一个颇有争议的隐式转换，可从 `const char*` 隐式转为 `char*`。由于字符串字面常数的型别并非 `string`，因此新的 `string` object 和传统的 *C-strings* 之间必须存在一种强烈关系：在“*strings* 和 *string-like object* 共通的操作场合”（例如比较、追加、插入等等动作）都应该可以使用 *C-strings*。或者具体地说，存在一个从 `const char*` 到 *strings* 的隐式型别转换。然而却不存在一个从 *string* object 到 *C-string* 的自动型别转换。这是出于安全考虑，防止意外转型导致奇异行为（`char*` 经常有奇异的行为）和模棱两可（例如在一个结合了 `string` 和 *C-string* 的表达式中，既可以把 `string` 转化为 `char*`，也可以反其道而行，这就导致模棱两可）。有好几种办法可以产生或改写/复制 *C-string*。更明确地说，`c_str()` 可以得到“*string* 对应的 *C-string*”，所得结果和“以 `'\0'` 为结尾的字符数组一样。运用 `copy()`，你也可以将字符串内容复制或写入既有的 *C-string* 或字符数组内。

请注意，`'\0'` 在 *string* 之中并不具有特殊意义，但在一般 *C-string* 中却用来标识字符串结束。在 *string* 中，字符 `'\0'` 和其它字符的地位完全相同。

请注意，千万不要以 `null` 指标（`NULL`）取代 `char*` 作为参数，这样会导致奇异行为，因为 `NULL` 具有整数型别，在单整数型别的重载函数版本上会被解释为数字 0 或“其值为 0”的字符。

有三个函数可以将字符串内容转换为字符数组或 *C-String*：

1. `data()` 以字符数组的形式返回字符串内容。由于并未追加 `'\0'` 字符，所以返回型别并非有效的 *C-string*。
2. `c_str()` 以 *C-string* 形式返回字符串内容，也就是在尾端添加 `'\0'` 字符。
3. `copy()` 将字符串内容复制到“调用者提供的字符数组”中。不添加 `'\0'` 字符。

注意，`data()` 和 `c_str()` 返回的字符数组由该字符串拥有。也就是说调用者千万不可修改它或释放其内存。例如：

```
std::string s("12345");

atoi(s.c_str())           // convert string into integer
f(s.data(),s.length())    // call function for a character array
                           // and the number of characters

char buffer[100];
s.copy(buffer,100);        // copy at most 100 characters of s into buffer
s.copy(buffer,100,2);      // copy at most 100 characters of s into buffer
                           // starting with the third character of s
```

一般而言，整个程序中你应该坚持使用 *strings*，直到你必须将其内容转化为 *char** 时才把它们转换为 *C-string*。请注意 *c_str()* 和 *data()* 的返回值有效期限在下一次调用 *non-const* 成员函数时即告终止。

```
std::string s;
...
foo(s.c_str()); // s.c_str() is valid during the whole statement

const char* p;
p = s.c_str(); // p refers to the contents of s as a C-string p
foo(p);        // OK (p is still valid)
s += "ext";     // invalidates p
foo(p);        // ERROR: argument p is not valid
```

11.2.5 大小 (Size) 和容量 (Capacity)

为了高效无误地运用 *strings*，你应该理解 *strings* 的大小和容量是如何配合的。一个 *strings* 存在三种“大小”：

1. *size()* 和 *length()*

返回 *string* 中现有的字符个数。上述两个函数等效⁴。

成员函数 *empty()* 用来检验字符数是否为 0，亦即字符串是否为空。你应该优先使用该函数，因为它比 *length()* 或 *size()* 来得快。

⁴ 这里两个成员函数所做的事情相同。根据 STL 概念，*size()* 是获取容器元素个数的通用成员函数，*length()* 则对应于一般 C-string *strlen()* 函数，传回字符串长度。

2. max_size()

此函数返回一个 *string* 最多能够包含的字符数。一个 *string* 通常包含一块单独内存区块内的所有字符，所以可能跟 PC 机器本身的限制有关系。返回值一般而言是索引型别的最大值减 1。之所以“减 1”有两个原因：(a) 最大值本身是 `npos`；(b) 具体实作中，可因此轻易在内部缓冲区之后添加一个 `'\0'`，以便将这个 *string* 当做 *C-string* 使用（例如透过 `c_str()`）。一旦某个操作函数使用一个长度大于 `max_size()` 的 *string*，`length_error` 异常就会被抛出来。

3. capacity()

重新分配内存之前，*string* 所能包含的最大字符数。

让 *string* 拥有足够的容量是很重要的，原因有二：

1. 重新分配会造成所有指向 *string* 的 *references*、*pointers* 和 *iterators* 失效。
2. 重新分配 (*reallocation*) 很耗时间。

因此，如果程序要用到指向 *string*（或其内部字符）的 *references*、*pointers* 和 *iterators*，抑或需要很快的执行速度，就必须考虑容量 (*capacity*) 问题。成员函数 `reserve()` 就是用来避免重分配行为。`reserve()` 使你得以预留一定容量，并确保该容量尚有余裕之时，*reference* 能够一直保持有效：

```
std::string s; // create empty string
s.reserve(80); // reserve memory for 80 characters
```

容量概念应用于 *string* 和应用于 *vector* 是相同的（参见第 6.2.1 节，p149）：但有一个显著差异：面对 *string* 你可以调用 `reserve()` 来缩减实际容量，而 *vector* 的 `reserve()` 却没有这项功能。拿一个“小于现有容量”的参数来调用 `reserve()`，实际上就是一种非强制性缩减请求 (*nonbinding shrink request*)——如果参数小于现有字符数，则这项请求被视为非强制性适度缩减请求 (*nonbinding shrink-to-fit request*)。也就是说你可能想要缩减容量至某个目标，但不保证你一定可以如愿。*String* 的 `reserve()` 参数默认值为 0，所以调用 `reserve()` 并且不给参数，就是一种“非强制性适度缩减请求”：

```
s.reserve(); // “would like to shrink capacity to fit the current size”
```

为什么缩减动作是非强制性的呢？因为“如何获取最佳性能”系由实作者定义。具体实作 *string* 时，如何处理速度和内存耗用量之间关系可能有不同的设计思路。因此任何实作作品都可以以较大的魄力增加容量，并且永不缩减。

C++ *Standard* 规定，唯有在响应 `reserve()` 调用时，容量才有可能缩减。因此即使发生“字符被删除或被改变”的事情，任何其它字符只要位于“被操作字符”之前，指向它们身上的那些 *references*、*pointers* 和 *iterators* 就仍然保持有效。

11.2.6 元素存取 (Element Access)

String 允许我们对其所包含的字符进行读写。有两种方法可以访问单一字符：subscript (下标) 操作符 `[]` 和成员函数 `at()`。两者都返回某指定索引的对应位置上的字符。一如既往，第一个字符索引为 0，最后的字符索引为 `length()-1`。但是请注意以下区别：

- `operator[]` 并不检查索引是否有效，`at()` 则会检查。如果调用 `at()` 时指定的索引无效，系统会抛出 `out_of_range` 异常。如果调用 `operator[]` 时指定的索引无效，其行为未有定义——可能存取非法内存，因而引起某些讨厌的边缘效应或甚至崩溃（崩溃了还算运气好，因为你好歹知道出错了）。
- 对于 `operator[]` 的 `const` 版本，最后一个字符的后面位置也是有效的。此时的实际字符数是有效索引。在此情况下 `operator[]` 的返回值是“由 `char` 型别之 `default` 构造函数所产生”的字符。因此，对于型别为 `string` 的对象，返回值为 `'\0'` 字符。

其它任何情况（包括成员函数 `at()` 和 `operator[]` 的 `non-const` 版本），实际字符数都是个无效索引。如果使用该索引，会引发异常，或导致未定义行为。

举个例子：

```
const std::string cs("nico"); // cs contains: 'n' 'i' 'c' 'o'
std::string s("abcde");      // s contains: 'a' 'b' 'c' 'd' 'e'

s[2]           // yields 'c'
s.at(2)        // yields 'c'

s[100]         // ERROR: undefined behavior
s.at(100)      // throws out_of_range

s[s.length()] // ERROR: undefined behavior
cs[cs.length()] // yields '\0'
s.at(s.length()) // throws out_of_range
cs.at(cs.length()) // throws out_of_range
```

为了允许更改 *string* 内容，`operator[]` 的 `non-const` 版本和 `at()` 都返回字符的 *reference*。一旦发生重分配行为，那些 *reference* 立即失效：

```
std::string s("abcde"); // s contains: 'a' 'b' 'c' 'd' 'e'

char& r = s[2]; // reference to third character
char* p = &s[3]; // pointer to fourth character
```



```

r = 'X';           // OK, s contains: 'a' 'b' 'X' 'd' 'e'
*p = 'Y';          // OK, s contains: 'a' 'b' 'X' 'Y' 'e'

s = "new long value"; // reallocation invalidates r and p

r = 'X';           // ERROR: undefined behavior
*p = 'Y';          // ERROR: undefined behavior

```

在这里, 为了避免运行时出错, 我们应该在 `r` 和 `p` 被初始化之前, 先运用 `reserve()` 保留足够的容量。

以下操作可能导致指向字符的 `references` 和 `pointers` 失效:

- 以 `swap()` 交换两值
- 以 `operator>>()` 或 `getline()` 读入新值
- 以 `data()` 或 `c_str()` 输出内容
- 调用 `operator[], at(), begin(), rbegin(), end()` 或 `rend()` 之外的任何 `non-const` 成员函数
- 调用任何函数并于其后跟着 `operator[], at(), begin(), rbegin(), end()` 或 `rend()`。

以上讨论同样适用于迭代器 (参见 11.2.13 节, p497)。

11.2.7 比较 (Comparisons)

Strings 支持常见的比较 (`comparison`) 操作符, 操作数可以是 *strings* 或 *C-strings*:

```

std::string s1, s2;
...

```

```

s1 == s2    // returns true if s1 and s2 contain the same characters
s1 < "hello" // return whether s1 is less than the C-string "hello"

```

如果以 `<`, `<=`, `>`, `>=` 来比较 *strings*, 得到的结果是根据“当前字符特性 (`current character traits`)”将字符依字典顺序逐一比较。例如以下所有比较结果均为 `true`:

```

std::string("aaaa") < std::string("bbbb")
std::string("aaaa") < std::string("abba")
std::string("aaaa") < std::string("aaaaaa")

```

你可以使用成员函数 `compare()` 来比较子字符串, 此函数针对一个 *string* 可使用多个参数进行处理, 如此一来就可以采用索引和长度, 双管齐下定位出子字符串。请注意 `compare()` 返回的是整数值而非布尔值。返回值意义如下: 0 表示相等, 小于 0 表示小于, 大于 0 表示大于。例如:

```

std::string s("abcd");

s.compare("abcd")      // returns 0
s.compare("dcba")      // returns a value < 0 (s is less)
s.compare("ab")        // returns a value > 0 (s is greater)

s.compare(s)           // returns 0 (s is equal to s)
s.compare(0,2,s,2,2)   // returns a value < 0 ("ab" is less than "cd")
s.compare(1,2,"bcx",2) // returns 0 ("bc" is equal to "bc")

```

如果想采用不同的比较准则，你也可以自己定义，并采用 STL 的比较算法（参见 11.2.13 节，p499 示例），或使用特殊的字符特性（character traits）完成“不计大小写”的比较动作。不过由于“具备特殊 traits class”的 *string* 是另一个不同的数据类型别，所以不能将这种 *string* object 拿来和 *string* object 共同处理。参见第 11.2.14 节，p503 示例。

针对国际市场而制作的程序，可能需要按特殊的当地（国别，locale）规则来比较字符串。为简化这个问题，locale class 提供了圆括号操作符（参见 p703），采用 *string* collation facet，根据对应的当地惯例来比较字符串，从而进行排序。详见 14.4.5 节，p724。

11.2.8 更改内容 (Modifiers)

你可以运用不同的成员函数或操作符来更改字符串内容。

赋值 (Assignments)

可运用 `operator=` 来对字符串赋新值。新值可以是 *string*、*C-string* 或单一字符。如果需要多个参数来描述新值，可采用成员函数 `assign()`。举个例子：

```

const std::string aString("othello");
std::string s;

s = aString;           // assign "othello"
s = "two\nlines";      // assign a C-string
s = ' ';               // assign a single character

s.assign(aString);      // assign "othello" (equivalent to operator =)
s.assign(aString,1,3);  // assign "the"
s.assign(aString,2,std::string::npos); // assign "hello"

s.assign("two\nlines");
    // assign a C-string (equivalent to operator =)
s.assign("nico",5);
    // assign the character array: 'n' 'i' 'c' 'o' '\0'
s.assign(5,'x');

```

```
// assign five characters: 'x' 'x' 'x' 'x' 'x'
```

也可以运用两个迭代器定义出来的字符区间进行赋值动作, 详见 11.2.13 节, p497。

交换 (Swapping Values)

和许多“非寻常的 (nontrivial)”型别一样, `string` 型别提供了一个特殊的 `swap()` 函数, 用来交换两字符串内容(4.4.2 节, p67 介绍全局函数 `swap()`)。这个用于 *strings* 的特殊 `swap()` 保证常数复杂度, 所以如果赋值之后不再需要旧值, 你应该利用它进行交换, 从而达成赋新值的目的。

令 Strings 成空

许多动作都可以令字符串成为空字符串, 例如:

```
std::string s;

s = "";      // assign the empty string
s.clear();   // clear contents
s.erase();   // erase all characters
```

安插 (Inserting) 和移除 (Removing) 字符

Strings 提供许多成员函数用于安插 (insert)、移除 (remove)、替换 (replace)、擦除 (erase) 字符。另有 `operator+=`, `append()` 和 `push_back()` 可添加字符。下面是个实例:

```
const std::string aString("othello");
std::string s;

s += aString;           // append "othello"
s += "two\nlines";      // append C-string
s += '\n';              // append single character

s.append(aString);      // append "othello" (equivalent to operator +=)
s.append(aString, 1, 3); // append "the"
s.append(aString, 2, std::string::npos); // append "hello"

s.append("two\nlines"); // append C-string (equivalent to operator +=)
s.append("nico", 5);    // append character array: 'n' 'i' 'c' 'o' '\0'
s.append(5, 'x');       // append five characters: 'x' 'x' 'x' 'x' 'x'

s.push_back('\n');      // append single character (equivalent to operator +=)
```

`operator+=` 将单一参数添加在 *string* 尾部, `append()` 可使用多个参数指定添加值。还有一个 `append()` 版本可以将两个迭代器指定的字符区间添加在 *string* 尾部 (参见 11.2.13 节, p497)。`push_back()` 是为了支持 *back inserters* 而设, STL 算法可由此往 *string* 尾部添加字符 (关于 *back inserters*, 详见 7.4.2 节, p272。11.2.13 节, p502 有其运用实例)。

和 `append()` 类似, 成员函数 `insert()` 也允许你安插字符。使用 `insert()` 时需知道安插位置的索引, 新字符将安插于此位置之后。

```
const std::string aString("age");
std::string s("p");

s.insert(1, aString);      // s: page
s.insert(1, "ersifl");    // s: persiflage
```

注意, 成员函数 `insert()` 不接受“索引 + 单独字符”的参数组合, 所以你必须传入一个 *string* 或一个额外数字:

```
s.insert(0, ' ');          // ERROR
s.insert(0, " ");          // OK
```

也许你还想试试这样:

```
s.insert(0, 1, ' ');       // ERROR: ambiguous
```

然而由于 `insert()` 具有以下重载形式, 导致令人厌烦的模棱两可现象:

```
insert (size_type idx, size_type num, charT c); // position is index
insert (iterator pos, size_type num, charT c); // position is iterator
```

string 的 `size_type` 通常被定义为 `unsigned`, *string* 的 `iterator` 通常被定义为 `char*`。这种情况下, 第一个参数 0 有两种转换可能, 不分优劣。为了获得正确操作, 你必须如此:

```
s.insert({std::string::size_type}0, 1, ' ');    // OK
```

刚才提到的模棱两可的第二形式, 恰恰给出了一个运用迭代器来安插字符的范例。如果你想运用迭代器来指定安插位置, 有三种情况: 安插一个字符、安插多个相同字符、安插“两个迭代器所指区间”内的字符 (参见 11.2.13 节, p497)。

和 `append()` 及 `insert()` 类似, 移除字符用的 `erase()` 函数也有好几个, 替换字符用的 `replace()` 函数也有好几个。例如:

```
std::string s = "i18n";          // s: i18n
s.replace(1, 2, "nternationalizatio"); // s: internationalization
s.erase(13);                     // s: international
s.erase(7, 5);                   // s: internal
s.replace(0, 2, "ex");            // s: external
```

你可以使用 `resize()` 改变 *string* 的字符数量。如果参数所指定的大小比现有字符数少, 则尾部字符会被移除。如果参数所指定的大小比现有字符数多, 则以某个字符填充尾部。你可以传入一个字符参数, 作为填充用的字符, 否则就采用字符型别的 `default` 构造函数产生填充字符 (对于型别 `char*`, 其值为 `'\0'`)。

11.2.9 子串和字符串接合 (concatenation)

你可以使用成员函数 `substr()` 从 *string* 身上提取出子字符串。例如:

```
std::string s("interchangeability");

s.substr()           // returns a copy of s
s.substr(11)         // returns string("ability")
s.substr(5,6)        // returns string("change")
s.substr(s.find('c')) // returns string("changeability")
```

你可以使用 `operator+` 把两个 *strings* (或 *C-strings*) 接合起来。例如:

```
std::string s1("enter");
std::string s2("nation");
std::string i18n;
i18n = 'i' + s1.substr(1) + s2 + "aliz" + s2.substr(1);
std::cout << "i18n means: " + i18n << std::endl;
```

输出如下:

```
i18n means: internationalization
```

11.2.10 I/O 操作符

Strings 定义了常用的 I/O 操作符:

- `operator >>` 从 input stream 读取一个 *string*。
- `operator <<` 把一个 *string* 写到 output stream 中。

这些操作符的使用方法和面对一般 *C-strings* 时相同。更明确地说, `operator>>` 的执行方式如下:

1. 如果设置了 `skipws` 标志 (参见 13.7.7 节, p625), 则跳过开头空格。
2. 持续读取所有字符, 直到发生以下情形之一:
 - 下个字符为空格符 (`whitespace`)
 - `stream` 不再处于 `good` 状态 (例如遇到 *end-of-file*)

- `stream` 的 `width()` 结果大于 0 (13.7.3 节, p618), 而目前已读出 `width()` 个字符。
- 已读取 `max_size()` 个字符。

3. `stream width()` 被设为 0。

一般而言, `input` 操作符读入下一个字时, 会跳过硬导的空格符。所谓空格符是指任何令 `isspace(c, strm.getloc())` 结果值为 `true` 的字符 (关于 `isspace()`, 请见 14.4.4 节, p718)。

`output` 操作符通常也会考虑 `stream width()`。如果 `width()` 大于 0, 则 `operator<<` 至少要写入 `width()` 个字符。

String classes 在命名空间 `std` 内还提供了一种用于逐行读取的特殊函数: `std::getline()`。该函数读取所有字符, 包括开头的空格符, 直到遭遇分行符号或 *end-of-file*。分行符号可指定, 不可添加。缺省情况下分行符号为 `newline` 字符, 但你也可以把自己喜欢的分行符号作为参数传给 `getline()`⁵。

```
std::string s;
while (getline(std::cin, s)) { // for each line read from cin
    ...
}
while (getline(std::cin, s, ',')) { // for each token separated by ','
    ...
}
```

注意, 如果你是逐一读取语汇单元, 那么 `newline` 字符不被视为特殊字符。因此语汇单元中也可能包含 `newline` 字符。

11.2.11 搜索和查找 (Searching and Finding)

Strings 提供了许多用于搜索和查找字符及子字符串的函数⁶。你可以:

- 搜寻单一字符、字符区间 (子字符串)、或若干字符中的一个
- 前向搜寻和后向搜寻
- 从字符串头部或内部任何地方开始搜寻

⁵ 由于 "Koenig lookup" 搜寻法则会在函数被调用时考虑参数类别定义所处的命名空间 (参见 p17), 所以我们不必在 `getline()` 之前加上 `std::`。

⁶ 在这里我写搜索和查找 (*searching and finding*), 但是请不要被弄糊涂了, 它们 (几乎) 同义。搜索 (*search*) 函数的名称同样有 *find* 这个字眼, 但它们并不保证能找到任何东西。所以事实上它们只是试图找到某些东西。我以术语 *search* 来说明这些函数的行为, 而在涉及其函数名字时, 采用 *find*。译注: 中译本两者不分, 一律译为搜寻。

另外，如果配上迭代器，STL 的所有搜寻 (search) 算法都可派上用场。

所有搜寻函数的名字中都有 *find* 这个字。它们试图找到“与参数传入的 *value* 值相等”的字符所处位置，如表 11.5 所示。

表 11.5 Strings 搜寻函数

string 函数	效果
<code>find()</code>	搜寻第一个与 <i>value</i> 相等的字符
<code>rfind()</code>	搜寻最后一个与 <i>value</i> 相等的字符 (逆向搜寻)
<code>find_first_of()</code>	搜寻第一个“与 <i>value</i> 中的某值相等”的字符
<code>find_last_of()</code>	搜寻最后一个“与 <i>value</i> 中的某值相等”的字符
<code>find_first_not_of()</code>	搜寻第一个“与 <i>value</i> 中任何值都不相等”的字符
<code>find_last_not_of()</code>	搜寻最后一个“与 <i>value</i> 中任何值都不相等”的字符

所有搜寻函数都返回符合搜寻条件之字符区间内的第一个字符的索引。如果搜寻不成功 (没找到目标)，则返回 `npos`。这些搜寻函数都采用下面的参数方案：

- 第一参数总是被搜寻的对象。
- 第二参数 (可有可无) 指出 *string* 内的搜寻起点 (索引)。
- 第三参数 (可有可无) 指出搜寻的字符个数。

不幸的是上面这个参数方案与其它 *string* 相关函数不同。其它 *string* 函数的第一个参数是起点索引，随后是数值和长度。特别要指出的是，每个搜寻函数都下面的参数集进行重载：

- **`const string& value`**
搜寻对象为 *value* (一个 *string*)。
- **`const string& value, size_type idx`**
从 **this* 的 *idx* 索引位置开始，搜寻 *value* (一个 *string*)。
- **`const char* value`**
搜寻 *value* (一个 C-string)。
- **`const char* value, size_type idx`**
从 **this* 的 *idx* 索引位置开始，搜寻 *value* (一个 C-string)。
- **`const char* value, size_type idx, size_type value_len`**
从 **this* 的 *idx* 索引位置开始，搜索 *value* (一个 C-string) 内的前 *value_len* 个字符所组成的字符区间。*value* 内的 null 字符 ('\\0') 将不复特殊意义。
- **`const char value`**
搜寻 *value* (一个字符)。
- **`const char value, size_type idx`**
从 **this* 的 *idx* 索引位置开始，搜寻 *value* (一个字符)。

例如:

```
std::string s("Hi Bill, I'm ill, so please pay the bill");

s.find("il")           // returns 4 (first substring "il")
s.find("il",10)        // returns 13 (first substring "il" starting from s[10])
s.rfind("il")          // returns 37 (last substring "il")
s.find_first_of("il")  // returns 1 (first char 'i' or 'l')
s.find_last_of("il")   // returns 39 (last char 'i' or 'l')
s.find_first_not_of("il") // returns 0 (first char neither 'i' nor 'l')
s.find_last_not_of("il") // returns 36 (last char neither 'i' nor 'l')
s.find("hi")           // returns npos
```

你也可以通过 STL 算法来搜寻 *string* 内的字符或子字符串。这些算法通常都允许你使用自己的比较准则（实例参见 11.2.13 节, p499）。但是请注意, STL 搜索算法的命名方式和 *string* 搜寻函数的命名方式非常不同（详见 9.2.2 节, p324）。

11.2.12 数值 npos 的意义

如果搜寻函数失败, 就会返回 *string::npos*。试看下面例子:

```
std::string s;
std::string::size_type idx; // be careful: don't use any other type!
...

idx = s.find("substring");
if (idx == std::string::npos) {
    ...
}
```

只有当 "substring" 不是字符串 *s* 的子字符串时, *if* 语句才会得到 *true* 值。使用 *string* 的 *npos* 值及其型别时要格外小心; 若要检查返回值, 一定要使用型别 *string::size_type*, 不能以 *int* 或 *unsigned* 作为返回值型别; 否则返回值与 *string::npos* 之间的比较可能无法正确执行。这是因为 *npos* 被设计为 *-1*:

```
namespace std {
    template<class charT,
             class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
    class basic_string {
    public:
```



```

typedef typename Allocator::size_type size_type;
...
static const size_type npos = -1;
...
};
}

```

不幸的是 `size_type` (由字符串配置器 `allocator` 定义) 需为无正负号整数型别。因为缺省配置器以型别 `size_t` 作为 `size_type` (参见 15.3 节, p732)。于是 `-1` 被转换为无正负号整数型别, `npos` 也就成了该型别的最大无符号值。不过实际数值还是取决于型别 `size_type` 的实际定义。不幸的是这些最大值都不相同。事实上 `(unsigned long)-1` 和 `(unsigned short)-1` 不同 (前提是两者型别大小不同)。因此, 以下比较式:

```
idx == std::string::npos
```

如果 `idx` 的值为 `-1`, 由于 `idx` 和字符串 `string::npos` 型别不同, 比较结果可能得到 `false`:

```

std::string s;
...
int idx = s.find("not found"); // assume it returns npos
if (idx == std::string::npos) { // ERROR: comparison might not work
    ...
}

```

避免这种错误的办法之一就是直接检验搜寻是否失败:

```

if (s.find("hi") == std::string::npos) {
    ...
}

```

但由于我们常常需要用到匹配字符的位置索引, 所以另一个简单的解决方法是自行定义对应于 `npos` 的带正负号数值:

```
const int NPOS = -1;
```

前述的比较式必须略加修改 (但方便多了) 如下:

```

if (idx == NPOS) { // works almost always
    ...
}

```

遗憾的是如果 `idx` 的型别为 `unsigned short`, 抑或索引大于 `int` 最大值, 上述比较式就会失败, 所以这种解法并不完善 (也因此, *C++ Standard* 并没有按这种方式定义)。但是这两种情况很少发生, 所以此解法在大多数情况下都有效。如果你希望你的程序代码有高度移植性, 你应该对 `string` 型别的任何索引都采用 `string::size_type`。若要获得完善解法, 还需考虑精确型别 `string::size_type` 的重载函数。我希望将来 *C++ Standard* 能提出更好的解决方案。

11.2.13 Strings 对迭代器的支持

string 是字符的有序群集 (ordered collection)。所以 C++ 标准程序库为 *strings* 提供了相应接口，以便将字符串当做 STL 容器使用⁷。

更明确地说，你可以调用常用的成员函数，取得“能够遍历 *string* 内所有字符”的迭代器。如果对迭代器不熟悉，可以把它们看做是指向字符串内部单个字符的东西，就像普通指针之于 *C-string*。采用迭代器，你便可以通过调用 C++ 标准程序库提供（或用户自行定义）的算法，遍历 *string* 内的全部字符。你可以因此对字符串内的字符进行排序、逆向重排、找出最大值（字符）等等动作。

String 迭代器是 random access（随机存取）迭代器。也就是说它支持随机存取，所以任何一个 STL 算法都可与它搭配（关于迭代器的分类，请见 5.3.2 节，p93 和 7.2 节，p251）。通常 *string* 的“迭代器型别”（iterator, const_iterator 等）由 *string* class 本身定义，确切型别则由实作作品定义，但通常被简单定义为一般指针。“以指针实作而成”和“以 class 实作而成”的迭代器之间有着难对付的差别，参见 7.2.6 节，p258。

对迭代器而言，如果发生重分配（reallocation），或其所指值发生某些变化，迭代器就会失效，详见 p487, 11.2.6 节。

String 的迭代器相关函数

表 11.6 列出 *strings* 在迭代器方面提供的所有成员函数。通常，beg 和 end 所规范的区间包括 beg 但不包括 end，是个半开区间，常写作 [beg; end)，参见 5.3 节，p83]。

为了支持运用 back inserter，*string* 定义了 push_back()。关于 back inserter，详见 7.4.2 节，p272，它们在 *string* 中的使用实例请见 p502。

String 迭代器的运用实例

String 迭代器可以做一件非常有用的事情：透过一个简单的语句，把 *string* 内的所有字符都转为大写或小写。例如：

```
// string/iter1.cpp
#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
using namespace std;
```

⁷ 第 5 章已介绍过 STL。

表 11.6 *string* 的迭代器操作函数

表达式	效果
<code>s.begin()</code>	返回一个随机存取迭代器, 指向第一字符
<code>s.end()</code>	返回一个随机存取迭代器, 指向最后一个字符的下一个位置
<code>s.rbegin()</code>	返回一个逆向迭代器, 指向倒数第一个字符 (亦即最后一个字符)
<code>s.rend()</code>	返回一个逆向迭代器, 指向倒数最后一个字符的下一位置 (亦即第一字符的前一位置)
<code>string s(beg, end)</code>	以区间 <code>[beg; end)</code> 内的所有字符作为 <i>string</i> <code>s</code> 的初值
<code>s.append(beg, end)</code>	将区间 <code>[beg; end)</code> 内的所有字符添加于 <code>s</code> 尾部
<code>s.assign(beg, end)</code>	将区间 <code>[beg; end)</code> 内的所有字符赋值给 <code>s</code>
<code>s.insert(pos, c)</code>	在迭代器 <code>pos</code> 所指之处插入字符 <code>c</code> , 并返回新字符的迭代器位置
<code>s.insert(pos, num, c)</code>	在迭代器 <code>pos</code> 所指之处插入 <code>num</code> 个字符 <code>c</code> , 并返回第一个新字符的迭代器位置
<code>s.insert(pos, beg, end)</code>	在迭代器 <code>pos</code> 所指之处插入区间 <code>[beg; end)</code> 内的所有字符
<code>s.erase(pos)</code>	删除迭代器 <code>pos</code> 所指字符, 并返回下一个字符位置
<code>s.erase(beg, end)</code>	删除区间 <code>[beg; end)</code> 内的所有字符, 并返回下一个字符的下一位置
<code>s.replace(beg, end, str)</code>	以 <i>string</i> <code>str</code> 内的字符替代 <code>[beg; end)</code> 区间内的所有字符
<code>s.replace(beg, end, cstr)</code>	以 <i>C-string</i> <code>cstr</code> 内的字符替代 <code>[beg; end)</code> 区间内的所有字符
<code>s.replace(beg, end, cstr, len)</code>	以字符数组 <code>str</code> 的前 <code>len</code> 个字符替代 <code>[beg; end)</code> 区间内的所有字符
<code>s.replace(beg, end, num, c)</code>	以 <code>num</code> 个字符 <code>c</code> 替代 <code>[beg; end)</code> 区间内的所有字符
<code>s.replace(beg, end, newBeg, newEnd)</code>	以 <code>[newBeg; newEnd)</code> 区间内的所有字符替代 <code>[beg; end)</code> 区间内的所有字符

```
int main()
{
    // create a string
    string s("The zip code of Hondelage in Germany is 38108");
    cout << "original: " << s << endl;

    // lowercase all characters
    transform (s.begin(), s.end(),    // source
               s.begin(),             // destination
               tolower);              // operation
    cout << "lowered: " << s << endl;

    // uppercase all characters
    transform (s.begin(), s.end(),    // source
               s.begin(),             // destination
               toupper);              // operation
    cout << "uppered: " << s << endl;
}
```

程序输出如下:

```
original: The zip code of Hondelage in Germany is 38108
lowered:  the zip code of hondelage in germany is 38108
uppered:  THE ZIP CODE OF HONDELAGE IN GERMANY IS 38108
```

请注意, `tolower()` 和 `toupper()` 用的是旧式的 C 全局函数。如果国别 (locale) 不同或程序涵盖多种国别, 应采用 `tolower()` 和 `toupper()` 的新形式。详见 14.4.4 节, p718。

以下例子说明 STL 如何使用你自己定义的搜寻和排序准则, 以“不计大小写”的方式对 *string* 进行比较和搜寻:

```
// string/iter2.cpp

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

bool nocase_compare (char c1, char c2)
{
    return toupper(c1) == toupper(c2);
}
```

```

int main()
{
    string s1("This is a string");
    string s2("STRING");

    // compare case insensitive
    if (s1.size() == s2.size() &&    // ensure same sizes
        equal (s1.begin(),s1.end(),  // first source string
               s2.begin(),           // second source string
               nocase_compare)) {    // comparison criterion
        cout << "the strings are equal" << endl;
    }
    else {
        cout << "the strings are not equal" << endl;
    }

    // search case insensitive
    string::iterator pos;
    pos = search (s1.begin(),s1.end(),    // source string in which to search
                  s2.begin(),s2.end(),    // substring to search
                  nocase_compare);        // comparison criterion
    if (pos == s1.end()) {
        cout << "s2 is not a substring of s1" << endl;
    }
    else {
        cout << "'" << s2 << "\" is a substring of \""
              << s1 << "\" (at index " << pos - s1.begin() << ")"
              << endl;
    }
}

```

注意, `equal()` 的调用者必须保证第二区间至少要 and 第一区间具有一样多的元素 (字符)。因此先比较字符串的大小是必要的, 否则可能导致未定义的行为。

你可以在最后一条语句中处理两个 *string* 迭代器间的差距 (difference), 用以获取字符位置的索引:

```
pos - s1.begin()
```

之所以能够这么做, 因为 *string* 迭代器是一种 Random Access (随机存取) 迭代器。将索引转换为迭代器位置也是一样: 只要简单地把索引值加到迭代器身上即可。

本例中，使用者自行定义的辅助函数 `nocase_compare()` 用于“大小写不分”的字符串比较方式。其实你也可以组合运用某些函数适配器（function adapter）并采用以下表达式替换先前的 `nocase_compare`:

```
compose_f_gx_hy(equal_to<int>(),
                ptr_fun(toupper),
                ptr_fun(toupper))
```

细节请见 p309 和 p318。

如果在 `sets` 或 `maps` 中使用 *strings*，也许你会想要一个特定的排序准则，让这些容器能够以“大小写不分”的方式对 *string* 排序。相应例子请见 p213。

以下程序示范 *string* 迭代器的另一种运用：

```
// string/iter3.cpp

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

int main()
{
    // create constant string
    const string hello("Hello, how are you?");

    // initialize string s with all characters of string hello
    string s(hello.begin(),hello.end());

    // iterate through all of the characters
    string::iterator pos;
    for (pos = s.begin(); pos != s.end(); ++pos) {
        cout << *pos;
    }
    cout << endl;

    // reverse the order of all characters inside the string
    reverse (s.begin(), s.end());
    cout << "reverse: " << s << endl;

    // sort all characters inside the string
    sort (s.begin(), s.end());
    cout << "ordered: " << s << endl;
```

```

/* remove adjacent duplicates .
* - unique() reorders and returns new end
* - erase() shrinks accordingly
*/
s.erase (unique(s.begin(),
                s.end()),
         s.end());
cout << "no duplicates: " << s << endl;
}

```

程序输出如下:

```

Hello, how are you?
reverse:      ?uoy era woh ,olleH
ordered:      ,?HaeehlloooruwY
no duplicates: ,?HaeHloruwY

```

以下例子采用 backInserters, 从标准输入装置读取数据到 *strings* 内:

```

// string/unique.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <locale>
using namespace std;

class bothWhiteSpaces {
private:
    const locale& loc; // locale
public:
    /* constructor
    * - save the locale object
    */
    bothWhiteSpaces (const locale& l) : loc(l) {
    }
    /* function call
    * - returns whether both characters are whitespaces
    */
    bool operator() (char elem1, char elem2) {
        return isspace(elem1,loc) && isspace(elem2,loc);
    }
};

```

```
int main()
{
    string contents;

    // don't skip leading whitespaces
    cin.unsetf (ios::skipws);

    // read all characters while compressing whitespaces
    unique_copy(istream_iterator<char>(cin),    // beginning of source
                istream_iterator<char>(),        // end of source
                back_inserter(contents),         // destination
                bothWhiteSpaces(cin.getloc()));  // criterion for removing

    // process contents
    // - here: write it to the standard output
    cout << contents;
}
```

透过 `unique_copy()` 算法 (参见 p384, 9.7.2 节), `input stream cin` 的所有字符都被安插到字符串 `contents` 中。仿函数 `bothWhiteSpaces` 用于检查两个相邻字符是否都是空格——为了实现这个功能, 我们以 “`cin` 的 `locale` (译注: 上述粗体部分)” 作为其初值, 并调用 `isspace()` 检查字符是否为空格 (详见 14.4.4 节, p718)。`unique_copy()` 以 `bothWhiteSpaces` 为行为准则, 删除相邻重复空格。p385 可以找到类似的例子。

11.2.14 国际化 (Internationalization)

正如先前 (11.2.1 节, p479) 介绍 *string classes* 时所说, *template string class* `basic_string<>` 乃是以 “字符型别” 完成参数化: `string` 是一个针对 `char` 的特化体 (*specialization*), `wstring` 则是针对 `wchar_t` 的一个特化体。

为了应付那些 “答案取决于字符型别” 的技术问题, *strings* 使用所谓的字符特性 (*character traits*) 提供相关细节。我们需要一个新的 *class*, 因为我们无法改变内建型别 (例如 `char` 和 `wchar_t`) 的接口。关于特性类别 (*traits classes*) 的详细讨论请见 14.1.2 节, p687。

下面展示一个为 *strings* 打造的特性类别 (traits classes)，使字符可以在“不计大小写”的方式下被操作：

```
// string/icstring.hpp

#ifndef ICSTRING_HPP
#define ICSTRING_HPP
#include <string>
#include <iostream>
#include <cctype>      // 译注: for toupper()

/* replace functions of the standard char_traits<char>
 * so that strings behave in a case-insensitive way
 */
struct ignorecase_traits : public std::char_traits<char> {
    // return whether c1 and c2 are equal
    static bool eq(const char& c1, const char& c2) {
        return std::toupper(c1)==std::toupper(c2);
    }
    // return whether c1 is less than c2
    static bool lt(const char& c1, const char& c2) {
        return std::toupper(c1)<std::toupper(c2);
    }
    // compare up to n characters of s1 and s2
    static int compare(const char* s1, const char* s2,
                      std::size_t n) {
        for (std::size_t i=0; i<n; ++i) {
            if (!eq(s1[i],s2[i])) {
                return lt(s1[i],s2[i])?-1:1;
            }
        }
        return 0;
    }
    // search c in s
    static const char* find(const char* s, std::size_t n,
                          const char& c) {
        for (std::size_t i=0; i<n; ++i) {
            if (eq(s[i],c)) {
                return &(s[i]);
            }
        }
        return 0;
    }
};
```

```
// define a special type for such strings
typedef std::basic_string<char,ignorecase_traits> icstring;

/* define an output operator
 * because the traits type is different than that for std::ostream
 */
inline
std::ostream& operator << (std::ostream& strm, const icstring& s)
{
    // simply convert the icstring into a normal string
    return strm << std::string(s.data(),s.length());
}
#endif // ICSTRING_HPP
```

由于 C++ *Standard* 只为“采用相同字符型别和特性类别”的 stream 定义 I/O 操作，而此处的特性类别并不相同，所以我们不得不定义自己的 output 操作符。同样道理也适用于 input 操作符。

下面这个程序说明如何使用这些特殊字符串：

```
// string/icstring1.cpp

#include "icstring.hpp"

int main()
{
    using std::cout;
    using std::endl;

    icstring s1("hallo");
    icstring s2("otto");
    icstring s3("hALLo");

    cout << std::boolalpha;
    cout << s1 << " == " << s2 << " : " << (s1==s2) << endl;
    cout << s1 << " == " << s3 << " : " << (s1==s3) << endl;

    icstring::size_type idx = s1.find("All");
    if (idx != icstring::npos) {
        cout << "index of \"All\" in \"" << s1 << "\" : "
            << idx << endl;
    }
    else {
        cout << "\"All\" not found in \"" << s1 << endl;
    }
}
```

程序输出如下:

```
hallo == otto : false
hallo == HALLO : true
index of "All" in "hallo": 1
```

关于国际化 (internationalization) 议题, 详见第 14 章。

11.2.15 效率 (Performance)

C++ *Standard* 并未规定如何实作 *string classes*, 只是定义了其界面。

由于理念和侧重点各有不同, 不同的实作作品可能在速度和内存占用方面存在显著的差异。

如果你希望速度更快, 请确认你所使用的 *string classes* 采用了类似 *reference counting* (引用计数) 概念 (译注: 根据我对 SGI STL 源码的理解, 可确认 SGI *string* 支持 *reference counting*)。这种手法可以加速 *string* 的复制和赋值 (赋值), 因为在实作之中不再是对字符串内容进行操作, 而仅仅是复制和赋值字符串的 *reference* (本书有一个可适用于任何型别的智能型指标便是运用这种手法, 详见 6.8 节, p222)。通过 *reference counting*, 你甚至不必透过 *const reference* 来传递字符串; 不过基于灵活性和可移植性的考虑, 一般还是应该采用 *const reference* 来传递参数。

11.2.16 Strings 和 Vectors

Strings 和 *vectors* 很相似, 这也不奇怪, 因为它们都是一种动态数组。因此, 可以把 *strings* 视为一种“以字符作为元素”的特定 *vectors*。实用上可把 *string* 当做 STL 容器使用, 这在 p497, 11.2.13 节已有介绍。但由于 *string* 和 *vectors* 之间有许多本质上的不同, 所以把 *string* 当做特殊的 *vectors* 还是存在一定的危险。最重要的差异在于两者的主要目标:

- *vectors* 首要目标是处理和操作容器内的元素, 而非容器整体。因此实作时通常会为“容器元素的操作行为”进行优化。
- *strings* 主要是把整个容器视为整体, 进行处理和操作, 因此实作时通常会为“整个容器的赋值和传递”进行优化。

不同的目标导致完全不同的实作手法。例如 *strings* 通常采用 *reference counting* (引用计数) 手法, *vectors* 则决不会如此。然而我们还是可以把 *vectors* 当做一般的 *C-strings* 来使用, 详见 6.2.3 节, p155。

11.3 细说 String Class

本节如果出现 *string*，指的是实际的 *string class*，可以是 *string*，*wstring* 或 *basic_string<>* 的任何一种特定形式；*char* 则是指实际字符型别，亦即 *string* 所使用的 *char* 和 *wstring* 所使用的 *wchar_t*。其它以斜体字标出的型别和实值的定义都取决于字符型别或特性类别（*traits class*）的个别定义。14.1.2 节, p687 详细介绍了所谓特性类别。

11.3.1 内部的型别定义和静态值

string::traits_type

- 字符特征（*character traits*）的型别。
- *basic_string<>* 的第二个 *template* 参数。
- 对型别 *string* 而言，此值等价于 *char_traits<char>*。

string::value_type

- 字符型别。
- 等价于 *traits_type::char_type*。
- 对型别 *string* 而言，此值等价于 *char*。

string::size_type

- 未带正负号的整数型别，用来指定大小值和索引。
- 等价于 *allocator_type::size_type*。
- 对型别 *string* 而言，此值等价于 *size_t*。

string::difference_type

- 带正负号的整数型别，用来指定差值（距离）。
- 等价于 *allocator_type::difference_type*。
- 对型别 *string* 而言，此值等价于 *ptrdiff_t*。

string::reference

- 字符的 *references* 型别。
- 等价于 *allocator_type::reference*。
- 对型别 *string* 而言，此值等价于 *char&*。

string::const_reference

- 常数型的字符 *references* 型别。
- 等价于 *allocator_type::const_reference*。
- 对型别 *string* 而言，此值等价于 *const char&*。

`string::pointer`

- 字符的 `pointers` 型别。
- 等价于 `allocator_type::pointer`。
- 对型别 `string` 而言, 此值等价于 `char*`。

`string::const_pointer`

- 常数型的字符 `pointers` 型别。
- 等价于 `allocator_type::const_pointer`。
- 对型别 `string` 而言, 此值等价于 `const char&`。

`string::iterator`

- 迭代器型别。
- 确切型别由实作作品负责定义。
- 对型别 `string` 而言通常为 `char*`。

`string::const_iterator`

- 常数型迭代器型别。
- 确切型别由实作作品负责定义。
- 对型别 `string` 而言通常为 `const char*`。

`string::reverse_iterator`

- 逆向迭代器 (`reverse iterators`) 型别。
- 等价于 `reverse_iterator<iterator>`。

`string::const_reverse_iterator`

- 常数型逆向迭代器 (`constant reverse iterators`) 型别。
- 等价于 `reverse_iterator<const_iterator>`。

`static const size_type string::npos`

- 这是一个特殊值, 表示下列情形:
 - “未找到”
 - “所有剩余字符”
- 初始值为 `-1` (一个无正负号整数值)。
- 使用 `npos` 时要十分小心, 详见 11.2.3 节, p495。

11.3.2 生成 (Create)、拷贝 (Copy)、销毁 (Destroy)

`string::string ()`

- `default` (缺省) 构造函数。
- 产生一个空字符串。

```
string::string (const string& str)
```

- copy (拷贝) 构造函数。
- 产生一个新字符串, 是 *str* 的副本。

```
string::string (const string& str, size_type str_idx)
```

```
string::string (const string& str, size_type str_idx,  
               size_type str_num)
```

- 产生一个新字符串, 其初值为 “*str* 内, 从索引 *str_idx* 开始的最多 *str_num* 个字符”。
- 如果没有指定 *str_num*, 则取用 “从 *str_idx* 开始到 *str* 尾部” 的所有字符。
- 如果 *str_idx* > *str.size()*, 抛出 *out_of_range* 异常。

```
string::string (const char* cstr)
```

- 产生一个字符串, 并以 *C-string* *cstr* 作为初值。
- 初值为 *cstr* 内以 ‘\0’ 为结束符号 (但不包括 ‘\0’) 的所有字符。
- *cstr* 不可为 NULL 指标。
- 如果所得结果超出最大字符数, 抛出 *length_error* 异常。

```
string::string (const char* chars, size_type chars_len)
```

- 产生一个字符串, 以字符数组 *chars* 内的 *chars_len* 个字符为初值。
- *chars* 必须至少包含 *chars_len* 个字符。这些字符可以为任意值, ‘\0’ 无特殊意义。
- 如果 *chars_len* 等于 *string::npos*, 抛出 *length_error* 异常。
- 如果所得结果超出最大字符数, 抛出 *length_error* 异常。

```
string::string (size_type num, char c)
```

- 产生一个字符串, 其内容初值为 *num* 个字符 *c*。
- 如果 *num* 等于 *string::npos*, 抛出 *length_error* 异常。
- 如果所得结果超出最大字符数, 抛出 *length_error* 异常。

```
string::string (InputIterator beg, InputIterator end)
```

- 产生一个字符串, 以区间 [*beg*; *end*) 内的字符为初值。
- 如果所得结果超出最大字符数, 抛出 *length_error* 异常。

```
string::~string ()
```

- 析构函数。
- 销毁所有字符并释放内存。

大部分构造函数都可以接受配置器作为附加参数传入 (见 11.3.12 节, p526)。

11.3.3 大小 (Size) 和容量 (Capacity)

关于大小

```
size_type string::size () const  
size_type string::length () const
```

- 两个函数都返回现有字符的个数。
- 二者等价 (*equivalent*)。
- 如果想要检查字符串是否为空, 应采用速度更快的 `empty()`。

```
bool string::empty () const
```

- 判断字符串是否为空, 亦即是否未包含任何字符。
- 等价于 `string::size()==0`, 但可能更快。

```
size_type string::max_size () const
```

- 返回字符串可含的最大字符数目。
- 任何操作一旦产生长度大于 `max_size()` 的字符串, 就抛出 `length_error` 异常。

关于容量

```
size_type string::capacity () const
```

- 返回重分配之前字符串所能包含的最多字符个数。

```
void string::reserve ()
```

```
void string::reserve (size_type num)
```

- 第二种形式用以保留“至少能容纳 `num` 个字符”的内存。
- 如果 `num` 小于目前实际容量, 调用这个函数相当于“非强制性容量缩减请求”。
- 如果 `num` 小于目前字符数, 调用这个函数相当于“非强制性适度缩减 (*shrink-to-fit*) 请求”, 其意义是请求将容量缩小至实际字符数的大小。
- 如果没有传递参数 (上述第一形式), 调用该函数相当于一个“非强制性适度缩减 (*shrink-to-fit*) 请求”。
- 容量永远不能小于实际字符数。
- 每次重分配都会造成所有 `references`、`pointers` 和 `iterators` 失效, 并耗费一定时间。因此可事先调用 `reserve()` 来加快速度, 并因此保持 `references`、`pointers` 和 `iterators` 的有效性 (详见第 11.2.5 节, p486)。

11.3.4 比较 (Comparisons)

```
bool comparison (const string& str1, const string& str2)
bool comparison (const string& str, const char* cstr)
bool comparison (const char* cstr, const string& str)
```

- 第一种形式返回两个 *strings* 的比较结果。
- 后两种形式返回 *string* 和 *C-string* 的比较结果。
- **comparison** 指的是以下任何一种动作：
 - operator ==
 - operator !=
 - operator <
 - operator >
 - operator <=
 - operator >=
- 按字典次序 (lexicographically) 进行比较 (参见 p488)。

```
int string::compare (const string& str) const
```

- 把 *this 拿来和 *str* 进行比较。
- 返回值：
 - > 0, 表示两端字符串相等
 - > < 0, 表示 *this 小于 *str* (按字典次序)
 - > > 0, 表示 *this 大于 *str* (按字典次序)
- 以 traits::compare() 为比较准则 (参见 14.1.2 节, p689)。
- 详见 11.2.7 节, p488。

```
int string::compare (size_type idx, size_type len, const string& str) const
```

- 将 *this 之内 “从 *idx* 开始的最多 *len* 个字符” 拿来和 *str* 比较
- 如果 *idx* > size(), 抛出 out_of_range 异常。
- 比较动作和前述的 compare(*str*) 相同。

```
int string::compare (size_type idx, size_type len,
                    const string& str, size_type str_idx,
                    size_type str_len) const
```

- 将 *this 之内 “从 *idx* 开始的最多 *len* 个字符” 拿来和 *str* 之内 “从 *str_idx* 开始的最多 *str_len* 个字符” 相较。
- 如果 *idx* > size(), 抛出 out_of_range 异常。
- 如果 *str_idx* > str.size(), 抛出 out_of_range 异常。
- 比较动作和前述的 compare(*str*) 相同。


```
int string::compare (const char* cstr) const
```

- 将*this 的字符和 C-string cstr 的字符进行比较。
- 比较动作和前述的 compare(str) 相同。

```
int string::compare (size_type idx, size_type len, const char* cstr) const
```

- 将*this 之内“从 idx 开始的最多 len 个字符”拿来和 C-string cstr 的所有字符比较⁸。
- 比较动作和前述的 compare(str) 相同。
- cstr 绝不可为 null 指针。

```
int string::compare (size_type idx, size_type len,
                    const char* chars, size_type chars_len) const
```

- 将*this 之内“从 idx 开始的最多 len 个字符”拿来和字符数组 chars 内的 chars_len 个字符相较。
- 比较动作和前述的 compare(str) 相同。
- chars 必须至少包含 chars_len 个字符 (可为任意值)。`'\0'` 没有特殊意义。
- 如果 chars_len 等于 string::npos, 抛出 length_error 异常。

11.3.5 字符存取 (Character Access)

```
char& string::operator[] (size_type idx)
```

```
char string::operator[] (size_type idx) const
```

- 两种形式都返回索引 idx 所指示的字符 (首字符的索引为 0)。
- 常量字符串的 length() 是一个有效索引, 上述函数会因此返回一个“由字符型别的缺省构造函数所产生”的值 (对 string 而言为 `'\0'`)。
- 非常量 (non-const) 字符串的 length() 是一个无效索引。
- 无效索引会导致未定义的行为。
- 非常量 (non-const) 字符串返回的 reference 会因为字符串的修改或重分配而失效 (详见 11.2.6 节, p487)。
- 如果调用者无法确定索引有效, 就应该采用 at()。

⁸ C++ Standard 对于此形式的 compare() 的描述与本书不同: 它将 cstr 视为字符数组而非 C-string, 并以 npos 作为其长度 (实际上是呼叫 compare() 的后继形式并以 npos 为附加参数)。这是 C++ Standard 的一个错误 (它理应抛出 length_error 异常)。

```
char& string::at (size_type idx)
const char& string::at (size_type idx) const
```

- 两种形式都返回索引 *idx* 所指示的字符（首字符的索引为 0）。
- 对于所有 *strings*，*length()* 均为非法（无效）索引。
- 传递无效索引（小于 0 或大于等于 *size()*）会导致 *out_of_range* 异常。
- 非常量（non-const）字符串返回的 *reference* 会因为字符串的修改或重分配而失效（详见 11.2.6 节，p487）。
- 如果调用者确定索引是有效的，可采用操作符 *[]*，速度更快。

11.3.6 产生 C-string 和字符数组 (Character Arrays)

```
const char* string::c_str () const
```

- 将 *string* 的内容以 C-string（一个字符数组，尾部添加 '\0'）形式返回。
- 返回值隶属于该 *string*，所以调用者不能修改、释放或删除该返回值。
- 唯有当 *string* 存在，并且用来处理该返回值的函数是个“常数型函数”时，这个返回值才保持有效。

```
const char* string::data () const
```

- 将 *string* 的内容以字符数组的形式返回。
- 返回值内含 *string* 的所有字符，完全未加修改或扩充。更明确地说，并没有附加 *null* 字符。因此这个返回值往往不是有效的 C-string。
- 返回值隶属于该 *string*，所以调用者不能修改、释放或删除该返回值。
- 唯有当 *string* 存在，并且用来处理该返回值的函数是个“常数型函数”时，这个返回值才保持有效。

```
size_type string::copy (char* buf, size_type buf_size) const
```

```
size_type string::copy (char* buf, size_type buf_size, size_type idx) const
```

- 以上两种形式都将字符串 **this*（从索引 *idx* 开始）内最多 *buf_size* 个字符复制到字符数组 *buf* 中。
- 返回被复制的字符数。
- 不添加 *null* 字符。因此函数执行后的 *buf* 内容可能不是有效的 C-string。
- 调用者必须确保 *buf* 有足够的内存；否则会导致未定义行为。
- 如果 *idx > size()*，抛出 *out_of_range* 异常。

11.3.7 更改内容

赋值 (Assignments)

```
string& string::operator= (const string& str)
string& string::assign (const string& str)
```

- 以上两种形式都将 *str* 的值赋给 *this。
- 都返回 *this。

```
string& string::assign (const string& str, size_type str_idx,
                      size_type str_num)
```

- 将字符串 *str* 之内“从索引 *str_idx* 开始最多 *str_num* 个字符”赋值给 *this。
- 返回 *this。
- 如果 *str_idx* > *str.size()*，抛出 *out_of_range* 异常。

```
string& string::operator= (const char* cstr)
string& string::assign (const char* cstr)
```

- 以上两种形式都将 *C-string* 的值赋给 *this。
- 赋值“以 '\0' 结尾 (但不包括 '\0’) ”的 *cstr* 所有字符。
- 都返回 *this。
- *cstr* 不得为 null 指标。
- 如果所得结果超出最大字符数，两个函数都抛出 *length_error* 异常。

```
string& string::assign (const char* chars, size_type chars_len)
```

- 将数组 *chars* 内的 *chars_len* 个字符赋值给 *this。
- 返回 *this。
- *chars* 至少必须包含 *chars_len* 个字符，字符可为任意值，'\0' 并无特殊意义。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
string& string::operator= (char c)
```

- 将字符 *c* 赋值给 *this。
- 返回 *this。
- 调用后，*this 只含这一个字符。

```
string& string::assign (size_type num, char c)
```

- 将 *num* 个字符 *c* 赋值给 *this。
- 返回 *this。
- 如果 *num* 等于 *string::npos*，抛出 *length_error* 异常。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
void string::swap (string& str)
void swap (string& str1, string& str2)
```

- 两种形式都用来交换两个 *strings*:
 - 成员函数版本用来交换 **this* 和 *str* 的内容。
 - 全局函数版本用来交换 *str1* 和 *str2* 的内容。
- 你应该尽可能采用这些函数取代赋值操作 (assignment)，因为它们更快。它们具有常数复杂度，详见 11.2.8 节, p490。

添加 (Appending) 字符

```
string& string::operator+= (const string& str)
string& string::append (const string& str)
```

- 两种形式都将 *str* 的字符添加到 **this* 尾部。
- 都返回 **this*。
- 如果所得结果超出最大字符数，两者都抛出 *length_error* 异常。

```
string& string::append (const string& str, size_type str_idx,
                        size_type str_num)
```

- 将 *str* 之内 “从 *str_idx* 开始最长 *str_num* 个字符” 添加到 **this* 尾部。
- 返回 **this*。
- 如果 *str_idx* > *str.size()*，抛出 *out_of_range* 异常。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
string& string::operator+= (const char* cstr)
string& string::append (const char* cstr)
```

- 都将 *C-string* 内的字符添加到 **this* 尾部。
- 都返回 **this*。
- *cstr* 不能为 *null* 指标。
- 如果所得结果超出最大字符数，两者都抛出 *length_error* 异常。

```
string& string::append (const char* chars, size_type chars_len)
```

- 将字符数组 *chars* 之内的 *chars_len* 个字符添加到 **this* 尾部。
- 返回 **this*。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
string& string::append (size_type num, char c)
```

- 将 *num* 个字符 *c* 添加到 *this 尾部。
- 返回 *this。
- 如果所得结果超出最大字符数，抛出 `length_error` 异常。

```
string& string::operator+= (char c)
```

```
void string::push_back (char c)
```

- 将字符 *c* 添加到 *this 尾部。
- `operator+=` 返回 *this。
- 如果所得结果超出最大字符数，两者都抛出 `length_error` 异常。

```
string& string::append (InputIterator beg, InputIterator end)
```

- 将区间 [*beg*, *end*) 内所有字符添加到 *this 尾部。
- 返回 *this。
- 如果所得结果超出最大字符数，抛出 `length_error` 异常。

安插 (inserting) 字符

```
string& string::insert (size_type idx, const string& str)
```

- 将 *str* 插入 *this 之内，新增字符从索引 *idx* 处开始安插。
- 返回 *this。
- 如果 *idx* > `size()`，抛出 `out_of_range` 异常。
- 如果所得结果超出最大字符数，抛出 `length_error` 异常。

```
string& string::insert (size_type idx, const string& str,  
                      size_type str_idx, size_type str_num)
```

- 将 *str* 之内“从 *str_idx* 开始最多 *str_num* 个字符”插入 *this，新增字符从索引 *idx* 处开始安插。
- 返回 *this。
- 如果 *idx* > `size()`，抛出 `out_of_range` 异常。
- 如果 *str_idx* > *str.size()*，抛出 `out_of_range` 异常。
- 如果所得结果超出最大字符数，抛出 `length_error` 异常。

```
string& string::insert (size_type idx, const char* cstr)
```

- 将 C-string *cstr* 插入 *this，新增字符从索引 *idx* 处开始安插。
- 返回 *this。
- *cstr* 不得为 `null` 指标。
- 如果 *idx* > `size()`，抛出 `out_of_range` 异常。
- 如果所得结果超出最大字符数，抛出 `length_error` 异常。

```
string& string::insert (size_type idx, const char* chars,
                      size_type chars_len)
```

- 将字符数组 *chars* 之内的 *chars_len* 个字符插入 *this，新增字符从索引 *idx* 处开始安插。
- 返回 *this。
- *chars* 必须包含至少 *chars_len* 个字符，字符可为任意值，'\0' 无特殊含义。
- 如果 *idx* > *size()*，抛出 *out_of_range* 异常。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
string& string::insert (size_type idx, size_type num, char c)
void string::insert (iterator pos, size_type num, char c)
```

- 两种形式分别在 *idx* 或 *pos* 指定的位置上安插 *num* 个字符 *c*。
- 第一形式将新字符插入 *str*，新增字符从索引 *idx* 处开始。
- 第二形式在迭代器 *pos* 所指字符之前方插入新字符。
- 这两个函数构成重载 (overloaded) 形式，可能导致模棱两可。如果你以 0 为第一参数，由于 0 可被视为索引 (通常被转换为 *unsigned*)，也可被视为迭代器 (通常被转换为 *char**)，因而导致模棱两可。这种情况下你应该明确告知参数是个“索引”。例如：

```
std::string s;
...
s.insert(0,1,' '); // ERROR: ambiguous
s.insert((std::string::size_type)0,1,' '); // OK
```

- 两种形式都返回 *this。
- 如果 *idx* > *size()*，两种形式都抛出 *out_of_range* 异常。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
iterator string::insert (iterator pos, char c)
```

- 在迭代器 *pos* 所指字符之前插入字符 *c* 的副本。
- 返回新被插入的字符的位置。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

```
void string::insert (iterator pos, InputIterator beg,
                  InputIterator end)
```

- 在迭代器 *pos* 所指字符之前插入区间 [*beg*; *end*) 的所有字符。
- 如果所得结果超出最大字符数，抛出 *length_error* 异常。

擦除 (Eraseing) 字符

```
void string::clear ()  
string& string::erase ()
```

- 两个函数都会删除 (delete) 字符串的所有字符, 因此调用后字符串成空。
- `erase()` 返回 `*this`。

```
string& string::erase (size_type idx)  
string& string::erase (size_type idx, size_type len)
```

- 两种形式都删除 `*this` 之内从索引 `idx` 开始的最多 `len` 个字符。
- 都返回 `*this`。
- 如果未指定 `len`, 则删除 `idx` 之后的所有字符。
- 如果 `idx > size()`, 两种形式都抛出 `out_of_range` 异常。

```
string& string::erase (iterator pos)  
string& string::erase (iterator beg, iterator end)
```

- 两种形式分别删除 `pos` 所指的单一字符或 `[beg;end)` 区间内的所有字符。
- 两者都返回最后一个被删除字符的下一个字符 (因此第二形式返回 `end`)⁹。

改变大小

```
void string::resize (size_type num)  
void string::resize (size_type num, char c)
```

- 两种形式都将 `*this` 的字符数改为 `num`。也就是说如果 `num` 不等于目前的 `size()`, 则函数将在尾部添加或删除足够字符, 使字符数量等于新的大小 `num`。
- 如果字符数增加, 则以 `c` 作为初值。如果未指定 `c`, 则使用 “字符型别”的 default 构造函数来为新字符设初值 (对 `string` 而言将是 `'\0'`)。
- 如果 `num` 等于 `string::npos`, 两者都抛出 `length_error` 异常。
- 如果所得结果超出最大字符数, 两者都抛出 `length_error` 异常。

⁹ C++ Standard 规定此一函数的第二形式传回 `end` 之后的位置, 那是一种错误描述。

替换 (Replacing) 字符

```
string& string::replace (size_type idx, size_type len, const string& str)
string& string::replace (iterator beg, iterator end, const string& str)
```

- 第一种形式将*this之内“从idx开始, 最长为len”的字符替换为str内的所有字符。
- 第二种形式将[beg;end)区间内的字符替换为str的所有字符。
- 返回*this。
- 如果idx > size(), 抛出out_of_range异常。
- 如果所得结果超出最大字符数, 两者都抛出length_error异常。

```
string& string::replace (size_type idx, size_type len,
                        const string& str, size_type str_idx, size_type str_num)
```

- 将*this之内“从idx开始, 最长为len”的字符替换为str之内“从str_idx开始, 最长为str_num”的所有字符。
- 返回*this。
- 如果idx > size(), 抛出out_of_range异常。
- 如果str_idx > str.size(), 抛出out_of_range异常。
- 如果所得结果超出最大字符数, 抛出length_error异常。

```
string& string::replace (size_type idx, size_type len, const char* cstr)
string& string::replace (iterator beg, iterator end, const char* cstr)
```

- 两种形式分别将*this之中“以idx开始, 最长为len”的字符, 或[begin; end)区间内的字符替换为C-string cstr中的所有字符。
- 都返回*this。
- cstr不得为null指标。
- 如果idx > size(), 两者都抛出out_of_range异常。
- 如果所得结果超出最大字符数, 两者都抛出length_error异常。

```
string& string::replace (size_type idx, size_type len,
                        const char* chars, size_type chars_len)
string& string::replace (iterator beg, iterator end,
                        const char* chars, size_type chars_len)
```

- 两种形式分别将*this之中“以idx开始, 最长为len”的字符或[begin; end)区间内的字符, 替换为字符数组chars的chars_len个字符。
- 都返回*this。
- chars必须包含至少chars_len个字符, 字符可为任意值, '\0'无特殊含义。
- 如果idx > size(), 两种形式都抛出out_of_range异常。
- 如果所得结果超出最大字符数, 两者都抛出length_error异常。


```
string& string::replace (size_type idx, size_type len,
                        size_type num, char c)
string& string::replace (iterator beg, iterator end,
                        size_type num, char c)
```

- 两种形式分别将*this之内“从idx开始, 最长为len”的字符, 或区间[begin; end)内的字符, 替换为num个字符c。
- 都返回*this。
- 如果idx > size(), 两种形式都抛出out_of_range异常。
- 如果结果大小超出最大字符数, 则两种形式都抛出length_error。

```
string& string::replace (iterator beg, iterator end,
                        InputIterator newBeg, InputIterator newEnd)
```

- 以区间[newBeg;newEnd)内的所有字符替换区间[beg;end)内的所有字符。
- 返回*this。
- 如果所得结果超出最大字符数, 抛出length_error异常。

11.3.8 搜寻 (Searching and Finding)

搜寻单一字符

```
size_type string::find (char c) const
size_type string::find (char c, size_type idx) const
size_type string::rfind (char c) const
size_type string::rfind (char c, size_type idx) const
```

- 这些函数都从索引idx开始搜索第一个或最后一个字符c。
- 函数find()正向 (forward) 搜寻, 并返回第一个搜寻结果。
- 函数rfind()逆向 (backward) 搜索, 并返回最后一个搜寻结果。
- 如果这些函数成功, 就返回字符索引; 否则返回string::npos。

搜寻子字符串

```
size_type string::find (const string& str) const  
size_type string::find (const string& str, size_type idx) const  
size_type string::rfind (const string& str) const  
size_type string::rfind (const string& str, size_type idx) const
```

- 这些函数都从索引 *idx* 开始搜寻第一个或最后一个子字符串 *str*。
- 函数 *find()* 正向 (*forward*) 搜寻, 并返回第一个子字符串。
- 函数 *rfind()* 逆向 (*backward*) 搜索, 并返回最后一个子字符串。
- 如果这些函数成功, 就返回子字符串的第一字符索引; 否则返回 *string::npos*。

```
size_type string::find (const char* cstr) const  
size_type string::find (const char* cstr, size_type idx) const  
size_type string::rfind (const char* cstr) const  
size_type string::rfind (const char* cstr, size_type idx) const
```

- 这些函数都从索引 *idx* 开始搜寻“与 *C-string* *cstr* 内容相同”的第一个或最后一个子字符串。
- 函数 *find()* 正向 (*forward*) 搜寻, 并返回第一个子字符串。
- 函数 *rfind()* 逆向 (*backward*) 搜索, 并返回最后一个子字符串。
- 如果这些函数成功, 就返回子字符串的第一字符索引; 否则返回 *string::npos*。
- *cstr* 不得为 *null* 指标。

```
size_type string::find (const char* chars, size_type idx,  
                        size_type chars_len) const  
size_type string::rfind (const char* chars, size_type idx,  
                        size_type chars_len) const
```

- 这些函数都从索引 *idx* 开始搜寻“与字符数组 *chars* 内的 *chars_len* 个字符内容相同”的第一个或最后一个子字符串。
- 函数 *find()* 正向 (*forward*) 搜寻, 并返回第一个子字符串。
- 函数 *rfind()* 逆向 (*backward*) 搜索, 并返回最后一个子字符串。
- 如果这些函数成功, 就返回子字符串的第一字符索引; 否则返回 *string::npos*。
- *chars* 必须包含至少 *chars_len* 个字符, 字符可为任意值, '\0' 无特殊含义。

搜寻第一个匹配字符

```
size_type string::find_first_of (const string& str) const
size_type string::find_first_of (const string& str, size_type idx) const
size_type string::find_first_not_of (const string& str) const
size_type string::find_first_not_of (const string& str, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于 (或不属于) *str* 的第一个字符。
- 如果函数成功, 就返回子字符串或字符索引; 否则返回 *string::npos*。

```
size_type string::find_first_of (const char* cstr) const
size_type string::find_first_of (const char* cstr, size_type idx) const
size_type string::find_first_not_of (const char* cstr) const
size_type string::find_first_not_of (const char* cstr, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于 (或不属于) C-string *cstr* 的第一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。
- *cstr* 不得为 null 指针。

```
size_type string::find_first_of (const char* chars, size_type idx,
                                size_type chars_len) const
size_type string::find_first_not_of (const char* chars, size_type idx,
                                    size_type chars_len) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中“属于 (或不属于) 字符数组 *chars* 内的前 *chars_len* 个字符”的第一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。
- *chars* 必须包含至少 *chars_len* 个字符, 字符可为任意值, '\0' 无特殊含义。

```
size_type string::find_first_of (char c) const
size_type string::find_first_of (char c, size_type idx) const
size_type string::find_first_not_of (char c) const
size_type string::find_first_not_of (char c, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中等于 (或不等于) 字符 *c* 的第一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。

搜寻最后一个匹配字符

```
size_type string::find_last_of (const string& str) const
size_type string::find_last_of (const string& str, size_type idx) const
size_type string::find_last_not_of (const string& str) const
size_type string::find_last_not_of (const string& str, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于 (或不属于) *str* 的最后一个字符。
- 如果函数成功, 就返回子字符串或字符索引; 否则返回 *string::npos*。

```
size_type string::find_last_of (const char* cstr) const
size_type string::find_last_of (const char* cstr, size_type idx) const
size_type string::find_last_not_of (const char* cstr) const
size_type string::find_last_not_of (const char* cstr, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中属于 (或不属于) *C-string* *cstr* 的最后一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。
- *cstr* 不得为 null 指标。

```
size_type string::find_last_of (const char* chars, size_type idx,
                               size_type chars_len) const
```

```
size_type string::find_last_not_of (const char* chars, size_type idx,
                                   size_type chars_len) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中“属于 (或不属于) 字符数组 *chars* 内的前 *chars_len* 个字符”的最后一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。
- *chars* 必须包含至少 *chars_len* 个字符, 字符可为任意值, '\0' 无特殊含义。

```
size_type string::find_last_of (char c) const
size_type string::find_last_of (char c, size_type idx) const
size_type string::find_last_not_of (char c) const
size_type string::find_last_not_of (char c, size_type idx) const
```

- 这些函数从索引 *idx* 处开始搜寻 *this 之中等于 (或不等于) 字符 *c* 的最后一个字符。
- 如果函数成功, 就返回字符索引; 否则返回 *string::npos*。

11.3.9 子字符串及字符串接合 (String Concatenation)

```
string string::substr () const
string string::substr (size_type idx) const
string string::substr (size_type idx, size_type len) const
```

- 这几种形式都返回**this* 之内“从索引 *idx* 开始的最多 *len* 个字符”所组成的子字符串。
- 如果没有 *len*，则将“余下的所有字符”当做子字符串返回。
- 如果没有 *idx* 和 *len*，则返回字符串副本。
- 如果 *idx* > *size()*，则抛出 *out_of_range* 异常。

```
string operator+ (const string& str1, const string& str2)
string operator+ (const string& str, const char* cstr)
string operator+ (const char* cstr, const string& str)
string operator+ (const string& str, char c)
string operator+ (char c, const string& str)
```

- 所有形式都可以接合两个操作数内的所有字符，并返回接合后的字符串。
- 操作数可以是下列任意一种：
 - 一个 *string*
 - 一个 *C-string*
 - 单一字符
- 如果接合结果超出最大字符数，则所有形式都抛出 *length_error* 异常。

11.3.10 I/O 函数

```
ostream& operator << (ostream& strm, const string& str)
```

- 将 *str* 内的字符写入 *stream strm*。
- 如果 *strm.width()* 大于 0，则至少写入 *width()* 个字符，然后 *width()* 被设置为 0。
- *ostream* 的型别是 *basic_ostream<char>*，具体型别取决于字符型别（参见 13.2.1 节, p588）。

```
istream& operator >> (istream& strm, string& str)
```

- 从 *strm* 读取下一个单字（字符串）的所有字符，放到 *str* 中。
- 如果 *strm* 的 *skipws* 标志被设立，则前导空格将忽略不计。
- 字符读取动作遇到下面情形之一即结束：
 - *strm.width()* 大于 0，且已存入 *width()* 个字符
 - *strm.good()* 为 *false*（可能导致相应的异常）

- 对下一个字符 *c*, `isspace(c, strm.getloc())` 为 `true`。
- 已存入 `str.max_size()` 个字符。
- 视情况重新分配内存。
- *istream* 的型别是 `basic_istream<char>`, 具体型别取决于字符型别 (参见 13.2.1 节, p588)。

```
istream& getline (istream& strm, string& str)
istream& getline (istream& strm, string& str, char delim)
```

- 从 *strm* 读取下一整行的所有字符到字符串 *str* 内。
- 读取所有字符 (包括前导空格) 直到下列情形之一发生:
 - `strm.good()` 为 `false` (可能导致相应的异常)
 - 读到 *delim* 或 `strm.widen('\n')`。
 - 已读入 `str.max_size()` 个字符。
- 行分隔符 (line delimiter) 乃是从参数中获取。
- 视情况重新分配内存。
- *istream* 的型别是 `basic_istream<char>`, 具体型别取决于字符型别 (参见 13.2.1 节, p588)。

11.3.11 产生迭代器

```
iterator string::begin ()
const_iterator string::begin () const
```

- 两种形式都返回一个 `random access` (随机存取) 迭代器, 指向字符串头部 (首字符位置)。
- 如果字符串为空, 以上调用等价于 `end()`。

```
iterator string::end ()
const_iterator string::end () const
```

- 两种形式都返回一个 `random access` (随机存取) 迭代器, 指向字符串尾部 (最后字符的下一个位置)。
- `end` 处并未定义字符, 所以 `*s.end()` 会导致未定义行为。
- 如果字符串为空, 以上调用等价于 `begin()`。

```
reverse_iterator string::rbegin ()
const_reverse_iterator string::rbegin () const
```

- 两种形式都返回一个 reverse random access (逆向随机存取) 迭代器, 指向倒数第一个字符 (亦即最后一个字符位置)。
- 如果字符串为空, 以上调用等价于 rend()。
- 关于逆向迭代器, 详见 7.4.1 节, p264。

```
reverse_iterator string::rend ()
const_reverse_iterator string::rend () const
```

- 两种形式都返回一个 reverse random access (逆向随机存取) 迭代器, 指向倒数最后一个元素的下一个位置 (亦即第一个字符的前一个位置)。
- rend 处并未定义字符, 所以*s.rend()会导致未定义行为。
- 如果字符串为空, 以上调用等价于 rbegin()。
- 关于逆向迭代器, 详见 7.4.1 节, p264。

11.3.12 对配置器 (allocator) 的支持

就像其它运用配置器的 classes 一样, *strings* 也提供常见的配置器相关支持。

```
string::allocator_type
```

- 这是配置器型别。
- 同时也是 basic_string<> 的第三个 template 参数。
- 对 string 型别而言, 等价于 allocator<char>。

```
allocator_type string::get_allocator () const
```

- 返回字符串的内存模型 (memory model)。

Strings 的所有构造函数都具有可有可无的配置器参数。根据 C++ Standard, 下面列出所有 *string* 构造函数:

```
namespace std {
    template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT>>
    class basic_string {
    public:
        // default constructor
        explicit basic_string(const Allocator& a = Allocator());

        // copy constructor and substrings
```

```
        basic_string(const basic_string& str,
                     size_type str_idx = 0,
                     size_type str_num = npos);
        basic_string(const basic_string& str,
                     size_type str_idx, size_type str_num,
                     const Allocator&);

        // constructor for C-strings
        basic_string(const charT* cstr,
                     const Allocator& a = Allocator());

        // constructor for character arrays
        basic_string(const charT* chars, size_type chars_len,
                     const Allocator& a = Allocator());

        // constructor for num occurrences of a character
        basic_string(size_type num, charT c,
                     const Allocator& a = Allocator());

        // constructor for a range of characters
        template<class InputIterator>
        basic_string(InputIterator beg, InputIterator end,
                     const Allocator& a = Allocator());
        ...
    };
}
```

这些构造函数的行为一如 11.3.2 节, p508 所介绍, 并带有额外机能: 允许你传递你自己的内存模型对象 (memory model object)。如果 *string* 系以另一个 *string* 为初值, 配置器也会被复制¹⁰。关于配置器, 详见第 15 章。

¹⁰ C++ *Standard* 最初版本规定: 当字符串被复制时, 应采用预设配置器。但此说法并无多大意义, 所以又有人提议对此行为进行修改, 最后形成目前的结果。

12

数值

Numerics

本章讲述 C++ 标准程序库的数值相关组件，其中包括复数 (complex)、数值数组 (value arrays)，以及从 C 标准程序库继承而来的全局数值函数。

C++ 标准程序库中，有两个数值组件在本书其它部分已经做过介绍：

1. STL 内含的数值算法，已在 9.11 节, p425 做过介绍。
2. 所有基本数值型别，各种与编译器相关 (implementation specific) 的表述方式已经在 4.3 节, p59 的 `numeric_limits` 一节讲过。

12.1 复数 (Complex Numbers)

C++ 标准程序库提供了一个 `template class complex<>`，用于操作复数。让我们回顾一下，所谓复数就是由实部 (real) 和虚部 (imaginary) 组成的数值。虚部的特点是“其平方值为负数”。换言之复数虚部带着 i ，其中 i 是 -1 的平方根。

`Class complex` 定义于头文件 `<complex>`：

```
#include <complex>
```

在其中，`class complex` 定义如下：

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

其中 `template` 参数 `T` 被用来作为复数的实部和虚部的标量型别 (scalar type)。

此外, C++ 标准程序库还为复数提供了针对标量型别 `float`, `double`, `long double` 的特化版本:

```
namespace std {
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;
}
```

通过这些特化版本, 我们可以提供一些特别的优化措施, 以及某些更安全的转换(从某个复数型别转换为另一个复数型别)。

12.1.1 Class Complex 运用实例

以下程序展示 `class complex` 的部分功能, 诸如产生复数、以不同表示法打印复数、在复数中执行某些共同操作等等。

```
// num/complex1.cpp

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

int main()
{
    /* complex number with real and imaginary parts
     * - real part: 4.0
     * - imaginary part: 3.0
     */
    complex<double> c1(4.0,3.0);

    /* create complex number from polar coordinates
     * - magnitude: 5.0
     * - phase angle: 0.75
     */
    complex<float> c2(polar(5.0,0.75));

    // print complex numbers with real and imaginary parts
    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
```

```

// print complex numbers as polar coordinates
cout << "c1: magnitude: " << abs(c1)
    << " (squared magnitude: " << norm(c1) << " ) "
    << " phase angle: " << arg(c1) << endl;
cout << "c2: magnitude: " << abs(c2)
    << " (squared magnitude: " << norm(c2) << " ) "
    << " phase angle: " << arg(c2) << endl;

// print complex conjugates
cout << "c1 conjugated: " << conj(c1) << endl;
cout << "c2 conjugated: " << conj(c2) << endl;

// print result of a computation
cout << "4.4 + c1 * 1.8: " << 4.4 + c1 * 1.8 << endl;

/* print sum of c1 and c2:
 * - note: different types
 */
cout << "c1 + c2: "
    << c1 + complex<double>(c2.real(),c2.imag()) << endl;

// add square root of c1 to c1 and print the result
cout << "c1 += sqrt(c1): " << (c1 += sqrt(c1)) << endl;
}

```

程序可能输出如下（确切的输出还得视“double 型别相关于编译器的某些性质”而定）：

```

c1: (4,3)
c2: (3.65844,3.40819)
c1: magnitude: 5 (squared magnitude: 25) phase angle: 0.643501
c2: magnitude: 5 (squared magnitude: 25) phase angle: 0.75
c1 conjugated: (4,-3)
c2 conjugated: (3.65844,-3.40819)
4.4 + c1 * 1.8: (11.6,5.4)
c1 + c2: (7.65844,6.40819)
c1 += sqrt(c1): (6.12132,3.70711)

```

下面是第二个例子。其中有个循环，其内读取两个复数，并计算出以第一个复数为底、以第二个复数为指数的幂次方值（power）：

```
// num/complex2.cpp

#include <iostream>
#include <complex>
#include <cstdlib>
#include <limits>
using namespace std;

int main()
{
    complex<long double> c1, c2;

    while (cin.peek() != EOF) { // 译注：请注意“复数输入方式”的设计

        // read first complex number
        cout << "complex number c1: ";
        cin >> c1;
        if (!cin) {
            cerr << "input error" << endl;
            return EXIT_FAILURE;
        }

        // read second complex number
        cout << "complex number c2: ";
        cin >> c2;
        if (!cin) {
            cerr << "input error" << endl;
            return EXIT_FAILURE;
        }

        if (c1 == c2) {
            cout << "c1 and c2 are equal !" << endl;
        }

        cout << "c1 raised to the c2: " << pow(c1, c2)
              << endl << endl;

        // skip rest of line
        cin.ignore(numeric_limits<int>::max(), '\n');
    }
}
```

表 12.1 列出此程序可能的输入和输出。

表 12.1 complex2.cpp 例中的输入和输出 (可能情况)

c1	c2	输出
2	2	c1 raised to c2: (4, 0)
(16)	0.5	c1 raised to c2: (4, 0)
(8, 0)	0.333333333	c1 raised to c2: (2, 0)
0.99	(5)	c1 raised to c2: (0.95099, 0)
(0, 2)	2	c1 raised to c2: (-4, 4.89843e-16)
(1.7, 0.3)	0	c1 raised to c2: (1, 0)
(3, 4)	(-4, 3)	c1 raised to c2: (4.32424e-05, 8.91396e-05)
(1.7, 0.3)	(4.3, 2.8)	c1 raised to c2: (-4.17622, 4.86871)

注意, 输入复数时, 你可以在括号内 (或是不需要括号) 只写实部, 也可以在括号内以逗号隔开实部和虚部。

12.1.2 复数的各种操作

template class complex 提供一系列操作, 细节如下。

创建 (Create), 复制 (Copy) 和赋值 (Assign)

表 12.2 列举了 complex 的构造函数和赋值操作。我们可透过构造函数传递初值的实部和虚部。如果不提供初值, 则分别以实部和虚部标量型别的缺省构造函数进行初始化。

Assignment (赋值) 操作符是改变既有复数的唯一途径。复合赋值操作符如 +=, -=, *=, /= 会对第一操作数和第二操作数进行加、减、乘、除等运算, 并将结果储存于第一操作数内。

运用辅助函数 polar(), 你可以采用极坐标 (距原点距离和弧度 (radians) 相位角) 来对一个复数进行初始化:

```
// create a complex number initialized from polar coordinates
std::complex<double> c2(std::polar(4.2, 0.75));
```

如果有隐式类型转换, 那么创建复数时会有一些问题。例如下面的写法没有问题:

```
std::complex<float> c2(std::polar(4.2, 0.75)); // OK
```

可是, 换用一个等号就不行了:

```
std::complex<float> c2 = std::polar(4.2, 0.75); // ERROR
```

此问题将在下一小节讨论。

表 12.2 Class `complex<>` 的构造函数和赋值操作

表达式	效果
<code>complex c</code>	产生一个复数，实部和虚部都为零；(0 + 0i)。
<code>complex c(1.3)</code>	产生一个复数，实部为 1.3，虚部为 0；(1.3 + 0i)。
<code>complex c(1.3,4.2)</code>	产生一个复数，实部为 1.3，虚部为 4.2；(1.3 + 4.2i)。
<code>complex c1(c2)</code>	产生一个复数，是 c2 的一个副本。
<code>polar(4.2)</code>	产生一个极坐标表示法的临时复数，模 (magnitude) 为 4.2，相位角 (phase angle) 为 0。
<code>polar(4.2,0.75)</code>	产生一个极坐标表示法的临时复数，模 (magnitude) 为 4.2，相位角 (phase angle) 为 0.75。
<code>conj(c)</code>	产生一个临时复数，是 c 的共轭复数 (实部相同而虚部相反)。
<code>c1 = c2</code>	将 c2 的值赋值给 c1
<code>c1 += c2</code>	将 c2 的值加入 c1
<code>c1 -= c2</code>	将 c1 的值减去 c2
<code>c1 *= c2</code>	将 c1 的值乘以 c2
<code>c1 /= c2</code>	将 c1 的值除以 c2

辅助函数 `conj()` 用来协助我们以某个复数的共轭复数 (conjugated complex) 产生一个新复数。所谓共轭复数就是将原复数的虚部反相 (negated) 而后得到的复数：

```
std::complex<double> c1(1.1,5.5);
std::complex<double> c2(conj(c1)); // initialize c2 with
                                   // complex<double>(1.1,-5.5)
```

隐式型别转换 (Implicit Type Conversions)

复数的 `float`, `double`, `long double` 等特化版本，遵循以下设计思想：允许安全转换 (例如 `complex<float>` 转为 `complex<double>`) 可以隐式进行，而不安全的转换 (例如 `complex<long double>` 转为 `complex<double>`) 必须显式进行 (详见 p542 的声明细节)：

```
std::complex<float> cf;
std::complex<double> cd;
std::complex<long double> cld;
...
std::complex<double> cd1 = cf;           // OK: safe conversion
std::complex<double> cd2 = cld;          // ERROR: no implicit conversion
std::complex<double> cd3(cld);           // OK: explicit conversion
```

此外就再也没有其它构造函数可以执行“由他种类型之复数转换而来”的构造行为了。特别提示一点，你不能把一个“实部和虚部为整数”的复数，转换为“实部和虚部为 float 或 double 或 long double”的复数。不过你可以将实部和虚部分开来当做参数，进行相同意义的（转换）操作：

```
std::complex<double> cd;
std::complex<int> ci;
...
std::complex<double> cd4 = ci; // ERROR: no implicit conversion
std::complex<double> cd5(ci); // ERROR: no explicit conversion
std::complex<double> cd6(ci.real(),ci.imag()); // OK
```

不幸的是，assignment（赋值）操作符允许接受不十分安全的转换——因为它们系以 template 函数的形式被提供出来，可接受任何型别。所以只要数值型别之间可以转换，你就可以赋值一个复数型别¹：

```
std::complex<double> cd;
std::complex<long double> cld;
std::complex<int> ci;
...
cd = ci;           // OK
cd = cld;          // OK
```

这个问题也发生在 polar() 及 conj()。例如下面的写法没有问题：

```
std::complex<float> c2(std::polar(4.2, 0.75)); // OK
```

可是下面的写法就不行：

```
std::complex<float> c2 = std::polar(4.2, 0.75); // ERROR
```

因为以下表达式：

```
std::polar(4.2, 0.75)
```

¹ 复数特化版本的构造只允许安全隐式转换，而赋值操作却允许任意隐式转换，这可能是 C++ Standard 的一个失误。

产生一个临时的 `complex<double>`，但 C++ 标准程序库之内并未定义从 `complex<double>` 到 `complex<float>` 的隐式转换²。

数值的存取 (Value Access)

表 12.3 列出各种复数属性的存取函数。

表 12.3 `class complex<>` 的各种属性 (数值) 的存取操作

表达式	效果
<code>c.real()</code>	返回实部值 (这是一个成员函数)
<code>real(c)</code>	返回实部值 (这是一个全局函数)
<code>c.imag()</code>	返回虚部值 (这是一个成员函数)
<code>imag(c)</code>	返回虚部值 (这是一个全局函数)
<code>abs(c)</code>	返回 <code>c</code> 的绝对值 ($\sqrt{c.real()^2 + c.imag()^2}$)
<code>norm(c)</code>	返回 <code>c</code> 绝对值的平方 ($c.real()^2 + c.imag()^2$)
<code>arg(c)</code>	返回 <code>c</code> 的极坐标相位角 (ϕ ，相当于 <code>atan2(c.imag(), c.real())</code>)

注意，`real()` 和 `imag()` 只提供读取实部和虚部的能力。如果你只是想改变实部或虚部，仍然必须赋值一个完整复数。例如下面语句将 `c` 的虚部设定为 3.7：

```
std::complex<double> c;
...
c = std::complex<double>(c.real(), 3.7);
```

2 在

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

和

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

之间有轻微的不同。前者使用显式转换，从型别 `x` 产生一个型别 `Y` 的新对象。后者使用隐式转换，产生一个型别 `Y` 的新对象。

比较 (Comparison)

复数之间的比较很方便，直接检查相等性就行了，如表 12.4。operator== 和 operator!= 被定义为全局函数，如此一来两个操作数之中就可以有一个为标量 (scalar value)。如果你使用一个标量作为操作数，它会被解释为复数的实部，相应的虚部则以虚部标量型别的默认构造函数产生出来 (通常是 0)。

表 12.4 class complex<> 定义的比较操作

表达式	效果
c1 == c2	判断是否 c1 等于 c2 (c1.real()==c2.real() && c1.imag()==c2.imag())
c == 1.7	判断是否 c1 等于 1.7 (c.real()==1.7 && c.imag()==0.0)
1.7 == c	判断是否 c1 等于 1.7 (c.real()==1.7 && c.imag()==0.0)
c1 != c2	判断是否 c1 和 c2 不同 (c1.real()!=c2.real() c1.imag()!=c2.imag())
c != 1.7	判断是否 c1 不等于 1.7 (c.real()!=1.7 c.imag()!=0.0)
1.7 != c	判断是否 c1 不等于 1.7 (c.real()!=1.7 c.imag()!=0.0)

其它的比较操作，例如 operator<，并未被定义出来。为复数定义顺序关系，虽然不是不可能，但不直观，也没什么用。例如复数的大小 (magnitude) 就不是很好的排序依据，因为两个复数可能大小相同，但非常不一样 (1 和 -1 就是例子)。你可以加上一个特别规则以产生合理顺序，例如面对两个复数 c1 和 c2，你可以认为当 $|c1| < |c2|$ 时 $c1 < c2$ ，如果两者大小 (magnitude) 相同则 $\arg(c1) < \arg(c2)$ 时视为 $c1 < c2$ 。不过，这些规则基本上没什么数学意义³。

因此，你不能在关联式容器中以 complex 作为元素型别 (如果你没有自行定义排序准则的话)，因为关联式容器需要对元素进行排序，需要用到函数对象 (仿函数) less<>，而后者会调用 operator< (详见 5.10.1 节, p134)。

不过如果你自行定义了一个 operator<，就可以对复数进行排序，并可以在关联式容器中使用复数。注意，请小心，不要污染了标准命名空间 (standard namespace)。例如：

³ 感谢 David Vandevoorde 指出这一点。

```
template <class T>
bool operator< (const std::complex<T>& c1,
               const std::complex<T>& c2)
{
    return std::abs(c1)<std::abs(c2) ||
        (std::abs(c1)==std::abs(c2) &&
         std::arg(c1)<std::arg(c2));
}
```

算术运算

复数支持四种基本运算，以及正负号，见表 12.5。

表 12.5 class complex<> 的算术操作

表达式	效果
c1 + c2	返回 c1 与 c2 的和
c + 1.7	返回 c1 与 1.7 的和
1.7 + c	返回 1.7 与 c1 的和
c1 - c2	返回 c1 与 c2 的差
c - 1.7	返回 c1 与 1.7 的差
c1 * c2	返回 c1 与 c2 的乘积
c * 1.7	返回 c1 与 1.7 的乘积
1.7 * c	返回 c1 与 1.7 的乘积
c1 / c2	返回 c1 与 c2 的商
c / 1.7	返回 c1 与 1.7 的商
1.7 / c	返回 1.7 与 c1 的商
-c	返回 c 的反相 (negated value)
+c	返回 c 本身
c1 += c2	等同于 c1 = c1 + c2
c1 -= c2	等同于 c1 = c1 - c2
c1 *= c2	等同于 c1 = c1 * c2
c1 /= c2	等同于 c1 = c1 / c2

输入/输出

Class `complex` 提供了一般的 I/O 操作符 `operator<<` 和 `operator>>`，如表 12.6。

表 12.6 class `complex<>` 的 I/O 操作

表达式	效果
<code>strm << c</code>	将复数 <code>c</code> 写入 ostream <code>strm</code> 中
<code>strm >> c</code>	从 istream <code>strm</code> 中读取复数 <code>c</code>

`output` 操作符根据 `stream` 的当前状态和格式，将复数写出：

`(realpart, imagpart)`

它的定义相当于：

```
template <class T, class charT, class traits>
std::basic_ostream<charT, traits>&
operator<< (std::basic_ostream<charT, traits>& strm,
           const std::complex<T>& c)
{
    // temporary value string to do the output with one argument
    std::basic_ostringstream<charT, traits> s;

    s.flags(strm.flags());           // copy stream flags
    s.imbue(strm.getloc());          // copy stream locale
    s.precision(strm.precision());  // copy stream precision

    // prepare the value string
    s << '(' << c.real() << ', ' << c.imag() << ')';

    // write the value string
    strm << s.str();

    return strm;
}
```

`input` 操作符可以接纳下面任何一种格式，从中读取一个复数：

`(realpart, imagpart)`
`(realpart)`
`realpart`

如果 `input stream` 内的下一个字符不符合上述所有格式, 则设立 `ios::failbit`, 并且可能抛出相应的异常 (参见 13.4.4 节, p602)。

可惜的是, 你不能设定复数表示式中的实部和虚部间的分隔符。所以如果有的国家以逗号作为小数点 (例如德国), `I/O` 看上去就十分奇异了。一个实部为 4.6, 虚部为 2.7 的复数, 输出结果是这样:

```
{4,6,2,7}
```

`I/O` 操作的运用, 详见 p532。

超越函数 (Transcendental Functions)

表 12.7 列出 `complex` 的所有超越函数 (三角函数、指数等等)。

表 12.7 Class `complex<>` 的超越函数 (Transcendental Functions)

表达式	效果
<code>pow(c,3)</code>	计算幂次方数 c^3
<code>pow(c,1.7)</code>	计算幂次方数 $c^{1.7}$
<code>pow(c1,c2)</code>	计算幂次方数 $c1^{c2}$
<code>pow(1.7,c)</code>	计算幂次方数 1.7^c
<code>exp(c)</code>	计算以 e 为底, c 为指数的幂次方数 (e^c)
<code>sqrt(c)</code>	计算 c 的平方根 (\sqrt{c})
<code>log(c)</code>	计算 c 的自然对数 ($\ln c$)
<code>log10(c)</code>	计算以 10 为底的 c 的对数 ($\lg c$)
<code>sin(c)</code>	计算 c 的正弦值 ($\sin c$)
<code>cos(c)</code>	计算 c 的余弦值 ($\cos c$)
<code>tan(c)</code>	计算 c 的正切值 ($\tan c$)
<code>sinh(c)</code>	计算 c 的双曲正弦值 ($\sinh c$)
<code>cosh(c)</code>	计算 c 的双曲余弦值 ($\cosh c$)
<code>tanh(c)</code>	计算 c 的双曲正切值 ($\tanh c$)

12.1.3 Class `complex<>` 细部讨论

本节详细探讨 `class complex<>` 的所有操作函数。以下所有定义中，`T` 是 `class complex<>` 的 `template` 参数，也就是复数的实部和虚部的标量型别。

型别定义

`complex::value_type`

- 实部和虚部的标量型别

构造、复制、赋值

`complex::complex ()`

- 缺省构造函数
- 构造一个复数，其中实部和虚部的初值系透过调用实部和虚部的缺省构造函数设定。所以如果是基本型别，实部和虚部的初值为 0（参见 p14 对于基本型别默认值的说明）。

`complex::complex (const T& re)`

- 构造一个复数，实部为 `re`，虚部则透过调用其缺省构造函数设定（基本型别的初值为 0）。
- 此构造函数同时定义了一个从 `T` 到 `complex` 的隐式型别转换。

`complex::complex (const T& re, const T& im)`

- 构造一个复数，实部初值为 `re`，虚部初值为 `im`。

`complex polar (const T& rho)`

`complex polar (const T& rho, const T& theta)`

- 以上两种形式都产生并返回一个复数，其初值以极坐标形式来设定。
- `rho` 是大小 (magnitude)。
- `theta` 是以弧度 (radians) 为单位的相位角（缺省为 0）。

`complex conj (const complex& cmplx)`

- 产生并返回一个复数：以复数 `cmplx` 的共轭复数为初值。所谓共轭复数是指虚部与原复数的虚部互为反相。

`complex::complex (const complex& cmplx)`

- `copy` 构造函数
- 产生一个新的复数，成为 `cmplx` 的复本。

- 复制实部和虚部。
- 此函数通常同时供应 `non-template` 和 `template` 两种形式 (参见 p11 对 `member templates` 的介绍)。因此具备对元素型别的自动转型能力。
- 然而, `float`, `double`, `long double` 等复数特化版本, 对于 `copy` 构造函数有所限制, 所以不安全的转换 (例如从 `double` 和 `long double` 转为 `float`, 或是从 `long double` 转为 `double`) 就必须显式进行, 并且不允许有其它的“元素转型”行为。

```
namespace std {
    template<> class complex<float> {
    public:
        explicit complex(const complex<double>&);
        explicit complex(const complex<long double>&);
        // no other kinds of copy constructors
        ...
    };
    template<> class complex<double> {
    public:
        complex(const complex<float>&);
        explicit complex(const complex<long double>&);
        // no other kinds of copy constructors
        ...
    };
    template<> class complex<long double> {
    public:
        complex(const complex<float>&);
        complex(const complex<double>&);
        // no other kinds of copy constructors
        ...
    };
}
```

关于其确切意义, 请参考 p534。

```
complex& complex::operator = (const complex& cplx)
```

- 将复数 `cplx` 赋值给 `*this`
- 返回 `*this`
- 此函数通常同时供应 `non-template` 和 `template` 两种形式 (参见 p11 对 `member templates` 的介绍)。因此具备对元素型别的自动型别转换能力。(对于 C++ 标准程序库提供的特化版本, 这一点也成立)。

```

complex& complex::operator += (const complex& cmplx)
complex& complex::operator -= (const complex& cmplx)
complex& complex::operator *= (const complex& cmplx)
complex& complex::operator /= (const complex& cmplx)

```

- 上述操作分别对 **this* 和 *cmplx* 进行加、减、乘、除运算，并将结果存入 **this*
- 返回 **this*
- 此函数通常同时供应 non-template 和 template 两种形式（参见 p11 对 member templates 的介绍）。因此具备对元素型别的自动型别转换能力。（对于 C++ 标准程序库提供的特化版本，这一点也成立）。

注意，赋值操作符是改变既有 *complex* 的唯一途径。

元素存取

```

T complex::real () const
T real (const complex& cmplx)
T complex::imag () const
T imag (const complex& cmplx)

```

- 上述函数分别返回实部和虚部。
- 注意，返回值并不是一个 reference，所以你不能运用这些函数来改变复数的实部和虚部。若要单独改变实部或虚部，必须赋予一个新的复数值（参见 p536）。

```

T abs (const complex& cmplx)

```

- 返回 *cmplx* 的绝对值（模，magnitude）。
- 绝对值计算公式： $\sqrt{\text{cmplx}.\text{real}()^2 + \text{cmplx}.\text{imag}()^2}$

```

T norm (const complex& cmplx)

```

- 返回 *cmplx* 绝对值的平方。
- 计算公式： $\text{cmplx}.\text{real}()^2 + \text{cmplx}.\text{imag}()^2$

```

T arg (const complex& cmplx)

```

- 返回以弧度（radians）为单位的极坐标相位角（ φ ）
- 相位角计算方法：`atan2(cmplx.imag(), cmplx.real())`

I/O 操作

`ostream& operator << (ostream& strm, const complex& cmplx)`

- 将 `cmplx` 的值以 (realpart, imagpart) 的格式写入 stream。
- 返回 `strm`。
- 此操作的具体行为见 p539。

`istream& operator >> (istream& strm, complex& cmplx)`

- 从 `strm` 中将一个新值读至 `cmplx`。
- 合法的输入格式是：
 - (realpart,imagpart)
 - (realpart)
 - realpart
- 返回 `strm`。
- 此操作的具体行为请见 p539。

操作符 (Operators)

`complex operator + (const complex& cmplx)`

- 正号。
- 返回 `cmplx`。

`complex operator - (const complex& cmplx)`

- 负号。
- 将复数 `cmplx` 的实部和虚部都取反相 (negated)。

`complex binary-op (const complex& cmplx1, const complex& cmplx2)`

`complex binary-op (const complex& cmplx, const T& value)`

`complex binary-op (const T& value, const complex& cmplx)`

- 上述各项操作返回 `binary-op` 计算所得的复数。
- 这里的 `binary-op` 可以是以下四种运算之一：
 - operator +
 - operator -
 - operator *
 - operator /
- 如果传入一个元素型别的标量值 (scalar value)，它会被视为一个复数的实部，虚部则由其标量型别的缺省初值决定 (如果是基本型别，初值为 0)。

```
bool comparison (const complex& cmplx1, const complex& cmplx2)
bool comparison (const complex& cmplx, const T& value)
bool comparison (const T& value, const complex& cmplx)
```

- 返回两个复数的比较结果, 或是一个复数与一个标量 (scalar value) 的比较结果。
- 这里的 **comparison** 可以是下面两种运算之一:
 - operator ==
 - operator !=
- 如果传入一个元素型别的标量值 (scalar value), 它会被视为一个复数的实部, 虚部则由其型别的缺省初值决定 (如果是基本型别, 初值为 0)。
- 注意, 并没有定义 <, <=, >, >= 等等操作符。

超越函数 (Transcendental Functions)

```
complex pow (const complex& base, int exp)
complex pow (const complex& base, const T& exp)
complex pow (const complex& base, const complex& exp)
complex pow (const T& base, const complex& exp)
```

- 上述所有形式都是计算“以 base 为基底, exp 为指数”的幂次方数, 定义为 $\exp(\exp \cdot \log(\text{base}))$ 。
- branch cuts 沿着负实数轴进行。
- **pow**(0, 0) 的结果由实作版本 (implementations) 自行定义。

```
complex exp (const complex& cmplx)
```

- 返回“以 e 为基底, cmplx 为指数”的幂次方结果。

```
complex sqrt (const complex& cmplx)
```

- 返回位于右半象限的 cmplx 平方根
- 如果参数是负实数, 则运算结果位于正虚数轴上。
- branch cuts 沿着负实数轴进行。

```
complex log (const complex& cmplx)
```

- 返回 cmplx 的自然对数 (亦即以 e 为底的对数)。
- 当 cmplx 是负实数时, $\text{imag}(\log(\text{cmplx}))$ 的值为 π (pi)。
- branch cuts 系沿着负实数轴进行。

```
complex log10 (const complex& cmplx)
```

- 返回 cmplx 的 (以 10 为基底的) 对数。
- 相当于 $\log(\text{cmplx}) / \log(10)$ 。
- branch cuts 系沿着负实数轴进行。

```
complex sin (const complex& cmplx)  
complex cos (const complex& cmplx)  
complex tan (const complex& cmplx)  
complex sinh (const complex& cmplx)  
complex cosh (const complex& cmplx)  
complex tanh (const complex& cmplx)
```

- 以上各操作函数分别对 *cmplx* 进行复数三角运算 (trigonometric operations)。

12.2 Valarrays

C++ 标准程序库提供了一个 `class valarray`，用以进行数值数组的运算。`valarray` 代表一个数学概念：数值线性序列。它是一维的，但你可以运用特殊技巧得到多维效果，这所谓特殊技巧，就是“经过运算的索引（computed indices）”和威力强大的子集（*subsetting*）能力。所以，`valarray` 可作为向量和矩阵运算的基础，也可作为多项式数学系统，并且有很好的性能。

`Valarray classes` 做了一些很精巧的优化工作，以求在处理数值数组时获得最佳性能。然而目前我们还不清楚这个组件未来的地位，因为另有一些开发工作获得了更出色的成果，其中最有趣的例子（之一）就是 `Blitz` 系统。如果你对数值处理感兴趣，你应该试着了解它，请访问 <http://www.oonumerics.org/blitz>。

`Valarray classes` 的设计并不是很好，事实上根本没有人确认过其最终规格是否能够运作。为什么会发生这样的事呢？因为没有人觉得自己应该对这些 `classes` 负责。把 `valarray` 引入 C++ *Standard* 的人在标准化工作结束之前很久就离开了标准委员会。因此使用 `valarray` 时你经常需要做一些不方便并且耗时的转型操作（p554）。

12.2.1 认识 Valarrays

`Valarrays` 是个一维数组，元素从零开始计数。它可以针对一个或多个数值数组的全体或部分进行数值处理。例如，假设 `a`, `b`, `c`, `x`, `z` 都是包含了数百个数值的数组，我们可以直接计算下面这个式子：

$$z = a * x * x + b * x + c$$

这种写法多么简洁，而且性能很好，因为这个 `class` 做了特殊的优化处理，避免在处理过程中产生临时对象。此外它还提供特别的接口和辅助类别，从而使我们能够处理数值数组的子集，以及作多维处理。因此，`valarray` 概念也有助于我们实作出向量和矩阵运算的 `classes`。

标准程序库保证 `valarray` 绝不会有别名（也就是 *alias free*）。换句话说任何 `non-constant valarray` 只能经由唯一途径存取。这么一来对其中数值的操作就可以实施更好的优化措施，因为编译器可以确保数据只有一种存取方式，无需顾虑太多。

头文件（Headers）

`Valarrays` 声明于头文件 `<valarray>`：

```
#include <valarray>
```

下面是其声明：

```
namespace std {  
    template<class T> class valarray; // numeric array of type T  
  
    class slice; // slice out of a valarray  
    template<class T> class slice_array;  
  
    class gslice; // a generalized slice  
    template<class T> class gslice_array;  
  
    template<class T> class mask_array; // a masked valarray  
    template<class T> class indirect_array; // an indirected valarray  
}
```

上述各个 classes 的意义如下：

- `valarray` 是核心类别，管理一个数值数组。
- `slice` 和 `gslice` 用来为 `valarray` 提供类似 BLAS⁴ 的切割 (slice) 和子集 (subset) 操作。
- `slice_array`, `gslice_array`, `mask_array`, `indirect_array` 是内部辅助类别，用来存放临时的数值或数据。你不能在应用程序中直接使用它们，它们由 `valarray` 的某些操作过程间接产生。

所有 classes 都针对元素型别进行了模板化 (templated)。原则上元素可以是任意数据类型，但是鉴于 `valarrays` 的天性，你还是应该使用数值型别。

Valarrays 的构造

构造 `valarray` 时，通常应该将元素的数量当做参数传入：

```
std::valarray<int> va1(10); // valarray of ten ints with value 0  
std::valarray<float> va2(5.7,10); // valarray of ten floats with value 5.7  
// (note the order)
```

如果你只传入一个参数，它将被视为 `valarray` 的大小，各元素则以其型别的缺省构造函数加以初始化。如果元素是基本型别，初值就是 0（关于“基本型别以缺省构造函数设初值”的特性描述，请见 2.2.2 节, p14）。如果你传入第二个参数，那么第一参数就是元素初值，第二参数就是元素个数。你瞧，这儿的行为和 C++ 标准程序库其它所有 classes 的习惯都不同，所有 STL 容器都是把第一参数视为元素个数，第二参数视为元素初值。

⁴ 所谓 BLAS 就是 The Basic Linear Algebra Subprograms library（基本线性代数子程序库），其中对于基本线性代数操作如矩阵乘法、三角计算、简单向量操作等，提供了核心计算引擎。

你可以像初始化一般数组那样地初始化一个 `valarray`:

```
int array[] = { 3, 6, 18, 3, 22 };

// initialize valarray by elements of an ordinary array
std::valarray<int> va3(array, sizeof(array)/sizeof(array[0]));

// initialize by the second to the fourth element
std::valarray<int> va4(array+1, 3);
```

`valarray` 会对传进来的值进行复制。所以你可以在初始化时传递临时数据，不会带来什么问题。

Valarray 的各项操作

`valarrays` 可以透过 subscript (下标) 操作符来存取某个元素。一如惯例，第一个元素的索引为 0:

```
va[0] = 3 * va[1] + va[2];
```

此外还定义了所有的普通数值运算 (加、减、乘、模数 `modulo`、反相、位操作、比较运算、逻辑操作、赋值操作)。这些操作符会针对“参与运算之 `valarrays`”的每一个元素被调用起来。因此，`valarray` 的运算结果也是一个 `valarray`，其元素个数和“参与运算之 `valarrays`”相同，每个元素承载着运算结果。例如以下语句:

```
va1 = va2 * va3;
```

等同于:

```
va1[0] = va2[0] * va3[0];
va1[1] = va2[1] * va3[1];
va1[2] = va2[2] * va3[2];
...
```

如果参与运算的若干 `valarrays` 的元素数量不同，则运算结果未有定义。

当然，只有当元素型别支持上述这些操作，它们才能进行。操作的确切意义取决于这些操作对元素的意义。所有这些操作都是直接对被处理之 `valarrays` 的每一个或每一对元素作相同运算。

如果是二元运算，操作数之一可以是元素型别的某个单值。这种情况下，该值将与另一个操作数 (某个 `valarray`) 中的每一个元素组合运算。例如:

```
va1 = 4 * va2;
```

等同于:

```
va1[0] = 4 * va2[0];
va1[1] = 4 * va2[1];
va1[2] = 4 * va2[2];
...
```

注意，这个单值的型别必须和 `valarray` 的元素型别完全一致。因此上述例子只有在元素型别是 `int` 的时候才能工作。下面这个语句就无法工作：

```
std::valarray<double> va(20);
...
va = 4 * va; // ERROR: type mismatch
```

二元运算的这种工作方式也应用在比较动作中。也就是说 `operator==` 并非返回一个布尔值以判断两个 `valarrays` 是否相等，而是返回一个新的、元素个数相同的 `valarray`，其元素型别为 `bool`，其中的每个元素都是 `operator==` 左右操作数的每一对相应元素的比较结果。例如以下程序代码：

```
std::valarray<double> val(10);
std::valarray<double> va2(10);
std::valarray<bool> vab(10);
...
vab = (val == va2);
```

最后一个表达式等同于：

```
vab[0] = (val[0] == va2[0]);
vab[1] = (val[1] == va2[1]);
vab[2] = (val[2] == va2[2]);
...
vab[9] = (val[9] == va2[9]);
```

因此你不可以使用 `operator<` 来进行 `valarrays` 的排序。如果 `valarrays` 的相等性操作系以 `operator==` 进行，那么 `valarrays` 不能作为 STL 容器的元素（关于 STL 容器元素的必要条件，详见 5.10.1 节，p134）。

以下程序展示 `valarrays` 的简单用法：

```
// num/val1.cpp

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

// print valarray
template <class T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
    cout << endl;
}
```

```
int main()
{
    // define two valarrays with ten elements
    valarray<double> val(10), va2(10);

    // assign values 0.0, 1.1, up to 9.9 to the first valarray
    for (int i=0; i<10; i++) {
        val[i] = i * 1.1;
    }

    // assign -1 to all elements of the second valarray
    va2 = -1;

    // print both valarrays
    printValarray(val);
    printValarray(va2);

    // print minimum, maximum, and sum of the first valarray
    cout << "min(): " << val.min() << endl;
    cout << "max(): " << val.max() << endl;
    cout << "sum(): " << val.sum() << endl;

    // assign values of the first to the second valarray
    va2 = val;

    // remove all elements of the first valarray
    val.resize(0);

    // print both valarrays again
    printValarray(val);
    printValarray(va2);
}
```

程序输出如下:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
min(): 0
max(): 9.9
sum(): 49.5

0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
```


超越函数 (Transcendental Functions)

超越计算 (三角运算和指数运算) 的定义如同一般数值运算: 运算目标是 `valarrays` 的所有元素。对于二元运算, 操作数之一可以是隶属元素型别之下的某个单值——这种情况下此一单值将和另一操作数 (某个 `valarray`) 的每一个元素组合运算。

所有这些运算都被定义为全局函数, 而非成员函数。这是为了使 `valarrays` 的子集可以通过自动 (隐式) 转型而成为操作数 (`valarrays` 的子集将在 12.2.2 节, p553 中描述)。

下面是 `valarrays` 的第二个运用实例, 展示超越函数的用法:

```
// num/val2.cpp

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

// print valarray
template <class T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
    cout << endl;
}

int main()
{
    // create and initialize valarray with nine elements
    valarray<double> va(9);
    for (int i=0; i<va.size(); i++) {
        va[i] = i * 1.1;
    }

    // print valarray
    printValarray(va);

    // double values in the valarray
    va *= 2.0;
```

```
// print valarray again
printValarray(va);

// create second valarray initialized by the values of the first plus 10
valarray<double> vb(va+10.0);

// print second valarray
printValarray(vb);

// create third valarray as a result of processing both existing valarrays
valarray<double> vc;
vc = sqrt(va) + vb/2.0 - 1.0;

// print third valarray
printValarray(vc);
}
```

程序输出如下:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8
0 2.2 4.4 6.6 8.8 11 13.2 15.4 17.6
10 12.2 14.4 16.6 18.8 21 23.2 25.4 27.6
4 6.58324 8.29762 9.86905 11.3665 12.8166 14.2332 15.6243 16.9952
```

12.2.2 Valarray 的子集 (Subsets)

下标操作符 `operator[]` 针对 `valarrays` 的一些特殊辅助对象进行了重载。这些辅助对象以不同方式定义 `valarrays` 的子集，从而得以优雅的方式对 `valarrays` 的某些子集进行读写操作。

`Valarray` 的子集可以使用某个“子集定义”作为索引，加以定义，例如：

```
va[std::slice(2,4,3)] // 4 elements with distance 3 starting from index 2
va[va>7]              // all elements with a value greater than 7
```

如果是在一个 `const valarray` 中使用类似 `std::slice(2,4,3)` 或 `va>7` 这样的定义式，则此表达式返回一个新的 `valarray`，其中包含相应元素。然而如果是在一个 `non-const valarray` 中，这样的表达式返回一个特殊的辅助性 `valarray` 临时对象。此临时对象并不包含子集实值，而只是该子集的定义；直到需要最终结果的时候，才真正对该表达式进行评估核定 (evaluation)。

这种技法称为缓式评估 (*lazy evaluation*) (译注: 可参考《*More Effective C++*》条款 17), 其优点是不必计算临时值, 从而节省了时间和内存。此外这项技术也提供了 *reference* 语意, 也就是说, 逻辑上的子集实际上也就引用着原值。你可以把这些子集当做语句目标 (左值, *lvalue*)。例如你可以对一个 *valarray* 的两个子集进行运算, 然后把结果赋值给该 *valarray* 的另一个子集 (详见稍后实例)。

不过, 避免“临时对象”的同时, 如果目标子集 (*destination subset*) 和源子集 (*source subset*) 有共同元素, 也可能会带来一些问题。所以只有在目标子集和所有源子集都没有共有元素的时候, *valarrays* 才能保证操作正确。

由于子集的精致定义, 你可以给 *valarrays* 赋予二维或多维语义。也就是说, *valarrays* 可以以某种方式作为多维数组使用。

定义 *valarrays* 子集的方法有四种:

1. *Slices* (切割)
2. *General slices* (一般化切割)
3. *Masked subsets* (屏蔽式子集)
4. *Indirect subsets* (间接式子集)

下面各小节通过一些实例进行解说。

Valarray 的子集 (Subset) 问题

在探讨个别的子集之前, 首先必须讲一个基本问题。*Valarray* 的子集处理在设计上不是很全面。生成子集很容易, 但合并 (*combine*) 很困难。更不幸的是你几乎时时都需要对 *valarray* 进行显式转型, 因为 C++ 标准程序库并没有要求 *valarray* 子集提供和 *valarrays* 一致的操作。

例如, 将两个 *subsets* 相乘, 并将结果赋值给另一个 *subset*, 下面这样做是不行的:

```
// ERROR: conversions missing
va[std::slice(0,4,3)]
    = va[std::slice(1,4,3)] * va[std::slice(2,4,3)];
```

你必须使用新式转型 (参见 p19) 如下⁵:

```
va[std::slice(0,4,3)]
    = static_cast<std::valarray<double>>(va[std::slice(1,4,3)]) *
      static_cast<std::valarray<double>>(va[std::slice(2,4,3)]);
```

或旧式转型如下:

⁵ 注意, 两个 > 字符之间必须有一个空格。>> 会被认为是右移位运算符, 从而引发语法错误。

```
va[std::slice(0,4,3)]
    = std::valarray<double>(va[std::slice(1,4,3)]) *
      std::valarray<double>(va[std::slice(2,4,3)]);
```

这实在太繁琐、太容易出错了。可还有更糟糕的，如果编译器没有很好的优化措施，上述转型还会耗损性能，因为每一个转型动作都会产生一个临时对象。如果不需转型，就不会生出这一大堆事情。

你可以运用下面的 `template function` 做一点小小改进：

```
/* template to convert valarray subset into valarray
*/
template <class T>
inline
std::valarray<typename T::value_type> VA (const T& valarray_subset)
{
    return std::valarray<typename T::value_type>(valarray_subset);
}
```

你必须这样使用它：

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                      VA(va[std::slice(2,4,3)]);    // OK
```

好一些了，不过性能耗损仍然不可避免。

如果元素型别是固定的，你可以定义简单的型别定义式：

```
typedef valarray<double> VAD;
```

这么一来，如果 `va` 的元素型别是 `double`，我们就可以这么写：

```
va[std::slice(0,4,3)] = VAD(va[std::slice(1,4,3)]) *
                      VAD(va[std::slice(2,4,3)]);    // OK
```

Slices（切割）

一次切割动作（一个 `slice`）定义出一个索引集，其中具备三个属性：

1. 起始索引
2. 元素数量（`size`，大小）
3. 元素间距（`stride`，步幅）

你可以将这三个属性以上述次序作为参数，传给 `class slice` 的构造函数。举个例子，以下表达式指定 4 个元素，从索引 2 开始，间距为 3：

```
std::slice(2,4,3)
```

其实也就是指定了下面 4 个索引所对应的元素：

```
2 5 8 11
```

间距可以为负，例如：

```
std::slice(9,5,-2)
```

这就指定了下面四个元素：

```
9 7 5 3 1
```

为了定义 `valarray` 的子集，你可以使用 `slice` 作为下标操作符的参数。例如以下表达式定义出 `valarray` `va` 的一个子集，其中内含索引为 2, 5, 8, 11 的四个元素：

```
va[std::slice(2,4,3)]
```

调用者必须确保这些索引都合法。

如果这个子集位于一个 `const valarray` 中，则这个子集是一个新的 `valarray`。否则此一子集内含的是原 `valarray` 相应元素的 `reference`。辅助类别 `slice_array` 正是为此问题而设：

```
namespace std {
    class slice;

    template <class T>
    class slice_array;

    template <class T>
    class valarray {
    public:
        // slice of a constant valarray returns a new valarray
        valarray<T> operator[] (slice) const;

        // slice of a variable valarray returns a slice_array
        slice_array<T> operator[] (slice);
        ...
    };
}
```

对于 `slice_array`，以下操作都有定义：

- 将同一个值赋予每一个元素。
- 赋值另一个 `valarray`（或 `valarray` 子集）。
- 调用任何一个赋值复合运算，例如 `operator+=` 或 `operator*+=`。

至于其它操作，你必须先将子集转换为 `valarray` 才能进行（见 p554）。注意，`class slice_array<>` 的用意完全是为 `slices` 提供内部辅助类别，对客户端应该是透明的。所以 `slice_array<>` 的所有构造函数和赋值操作符都是 `private`。

举个例子。以下表达式：

```
va[std::slice(2,4,3)] = 2;
```

将 `valarray` `va` 的第 3, 6, 9, 12 号元素赋予数值 2，动作相当于：

```
va[2] = 2;
va[5] = 2;
va[8] = 2;
va[11] = 2;
```

再举一例，以下语句将索引 2, 5, 8, 11 的四个元素加以平方：

```
va[std::slice(2,4,3)]
    *= std::valarray<double>(va[std::slice(2,4,3)]);
```

如同 p554 所说，你不能这么写：

```
va[std::slice(2,4,3)] *= va[std::slice(2,4,3)]; // ERROR
```

但你可以使用 p555 所用的 `VA()` `template function`：

```
va[std::slice(2,4,3)] *= VA(va[std::slice(2,4,3)]); // OK
```

你可以把同一个 `valarray` 的不同 `slices` 当做参数传递，这样就可以组合不同的子集，并将结果存入 `valarray` 的另一个子集中。例如以下语句：

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
    VA(va[std::slice(2,4,3)]);
```

相当于：

```
va[0] = va[1] * va[2];
va[3] = va[4] * va[5];
va[6] = va[7] * va[8];
va[9] = va[10] * va[11];
```

如果你把你的 `valarray` 当做一个二维矩阵，那么这个例子就是矩阵乘法（图 12.1）。不过请注意各个赋值操作的执行次序并未定义，如果“源子集”和“目标子集”有所重叠，结果未可预期。

我们可以以同样方式构造更复杂的运算，例如：

```
va[std::slice(0,100,3)]
    = std::pow(VA(va[std::slice(1,100,3)]) * 5.0,
    VA(va[std::slice(2,100,3)]));
```

请再次注意，参与运算的单值（例如本例的 5.0）必须和 `valarray` 的元素型别完全一致。

```
// fill valarray with values
for (int i=0; i<12; i++) {
    va[i] = i;
}

printValarray (va, 3);

// first column = second column raised to the third column
va[slice(0,4,3)] = pow (valarray<double>(va[slice(1,4,3)]),
                        valarray<double>(va[slice(2,4,3)]));

printValarray (va, 3);

// create valarray with three times the third element of va
valarray<double> vb(va[slice(2,4,0)]);

// multiply the third column by the elements of vb
va[slice(2,4,3)] *= vb;

printValarray (va, 3);

// print the square root of the elements in the second row
printValarray (sqrt(valarray<double>(va[slice(3,3,1)])), 3);

// double the elements in the third row
va[slice(2,4,3)] = valarray<double>(va[slice(2,4,3)]) * 2.0;

printValarray (va, 3);
}
```

程序输出如下:

```
0 1 2
3 4 5
6 7 8
9 10 11

1 1 2
1024 4 5
5.7648e+006 7 8
1e+011 10 11

1 1 4
1024 4 10
5.7648e+006 7 16
1e+011 10 22

32 2 3.16228
```

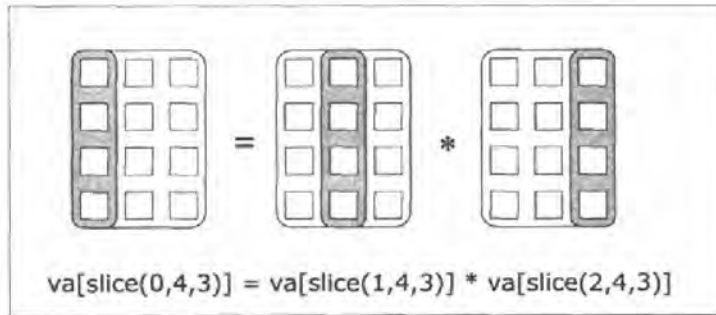


图 12.1 运用 Valarray Slices 进行向量乘法运算

下面是 valarray slices 的一个完整运用实例：

```
// num/slice1.cpp

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

// print valarray line-by-line
template<class T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; ++i) {
        for (int j=0; j<num; ++j) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    /* valarray with 12 elements
     * - four rows
     * - three columns
     */
    valarray<double> va(12);
```



```
1 1 8
1024 4 20
5.7648e+006 7 32
1e+011 10 44
```

General Slices (一般化切割)

General slices, 或称 *gslices*, 是 *slices* 的一般形式。与 *slices* 相似, 它也提供“在多维中进行多维运算”的能力。大体上 *gslices* 跟 *slices* 有以下相同属性:

- 起始索引
- 元素数量 (size, 大小)
- 元素间距 (stride, 步幅)

不过, 和 *slices* 不同之处在于, *gslices* 的元素数量和间距也是一个数组, 其中的元素个数和其维度相同。假设有下列 *gslice*:

```
start: 2
size: [ 4 ]
stride: [ 3 ]
```

则这个 *gslice* 和一般的 *slice* 相同, 因为该数组只处理一维。它定义了 4 个元素, 间距为 3, 起始索引为 2:

```
2 5 8 11
```

然而如果 *gslice* 如下:

```
start: 2
size: [ 2 4 ]
stride: [ 10 3 ]
```

那么这个 `gslice` 处理的是二维。最小的索引处理最高维度，所以这个 `gslices` 从索引 2 开始，以间距 10 取元素 2 次，以间距 3 取元素 4 次：

```
2 5 8 11
12 15 18 21
```

下面是一个三维 `gslice` 范例：

```
start: 2
size:  [ 3 2 4 ]
stride: [ 30 10 3 ]
```

它从索引 2 开始，以间距 30 取 3 个值，以间距 10 取 2 个值，以间距 3 取 4 个值：

```
2 5 8 11
12 15 18 21

32 35 38 41
42 45 48 51

62 65 68 71
72 75 78 81
```

能够运用数组来定义大小和间距，是 `gslice` 与 `slices` 之间唯一的区别。除此之外，两者相同：

1. 若要定义某个 `valarray` 的一个具体子集 (concrete subset)，你可以将 `gslice` 作为参数传给 `valarray` 的下标操作符。
2. 如果 `valarray` 是常量，则子集表达式 (expression) 将导致一个新的 `valarray`。
3. 如果 `valarray` 不是常量，则子集表达式将导致一个 `gslice_array`，后者以 `reference` 语意来表现 `valarray` 的相应元素：

```
namespace std {
    class gslice;

    template <class T>
    class gslice_array;

    template <class T>
    class valarray {
    public:
        // gslice of a constant valarray returns a new valarray
        valarray<T> operator[] (const gslice&) const;

        // gslice of a variable valarray returns a gslice_array
        gslice_array<T> operator[] (const gslice&);
        ...
    };
}
```

4. `gslice_array` 提供赋值 (`assign`) 操作符和复合赋值操作符, 用以修改子集内的元素。
5. 透过型别转换, 你可以将 `gslice array` 和其它 `valarrays` 以及 `valarrays` 子集组合起来 (参见 p554)。

下例展示 `valarray` 的 `gslice array` 的用法:

```
// num/gslice1.cpp

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

// print three-dimensional valarray line-by-line
template<class T>
void printValarray3D (const valarray<T>& va, int dim1, int dim2)
{
    for (int i=0; i<va.size()/(dim1*dim2); ++i) {
        for (int j=0; j<dim2; ++j) {
            for (int k=0; k<dim1; ++k) {
                cout << va[i*dim1*dim2+j*dim1+k] << ' ';
            }
            cout << '\n';
        }
        cout << '\n';
    }
    cout << endl;
}

int main()
{
    /* valarray with 24 elements
     * - two groups
     * - four rows
     * - three columns
     */
}
```

```
valarray<double> va(24);

// fill valarray with values
for (int i=0; i<24; i++) {
    va[i] = i;
}

// print valarray
printValarray3D (va, 3, 4);

// we need two two-dimensional subsets of three times 3 values
// in two 12-element arrays
size_t lengthvalues[] = { 2, 3 };
size_t stridevalues[] = { 12, 3 };
valarray<size_t> length(lengthvalues,2);
valarray<size_t> stride(stridevalues,2);

// assign the second column of the first three rows
// to the first column of the first three rows
va[gslice(0,length,stride)]
    = valarray<double>(va[gslice(1,length,stride)]);

// add and assign the third of the first three rows
// to the first of the first three rows
va[gslice(0,length,stride)]
    += valarray<double>(va[gslice(2,length,stride)]);

// print valarray
printValarray3D (va, 3, 4);
}
```

程序输出如下:

```
0 1 2
3 4 5
6 7 8
9 10 11

12 13 14
15 16 17
18 19 20
21 22 23
```

```
3 1 2
9 4 5
15 7 8
9 10 11

27 13 14
33 16 17
39 19 20
21 22 23
```

Masked Subsets (屏蔽式子集)

在 `valarray` 中定义子集的另一种方法就是 `Mask arrays`。你可以用一个布尔表达式来屏蔽 (mask) 相应元素, 例如以下表达式:

```
va[va > 7]
```

其中的子表达式:

```
va > 7
```

返回一个大小和 `va` 相同的 `valarray`, 其内每个元素都是一个布尔值, 表明 `va` 中的相应元素是否大于 7。而这个布尔型的 `valarray` 被 `subscript` (下标) 操作符用来筛选出 `valarray` 之中“使布尔运算获得 true”的所有元素。所以, 以下表达式:

```
va[va > 7]
```

指出 `valarray va` 的一个子集, 内含所有大于 7 的元素。

除此之外, `mask arrays` 和所有 `valarray subsets` 一样:

1. 若要定义某 `valarray` 的一个具体子集 (concrete subset), 你可以将布尔型的 `valarray` 作为参数传给 `valarray` 的 `subscript` (下标) 操作符。
2. 如果 `valarray` 是常量, 则子集表达式将导致一个新的 `valarray`。
3. 如果 `valarray` 不是常量, 则子集表达式将导致一个 `mask_array`, 后者以 `reference` 语意来表现 `valarray` 的相应元素:

```
namespace std {
    template <class T>
    class mask_array;
```

```

template <class T>
class valarray {
public:
    // masking a constant valarray returns a new valarray
    valarray<T> operator[] (const valarray<bool>&) const;

    // masking a variable valarray returns a mask_array
    mask_array<T> operator[] (const valarray<bool>&);
    ...
};
}

```

4. `mask_array` 提供赋值操作符和复合赋值操作符来修改子集中的元素。
5. 通过型别转换, 你可以将 `mask array` 与其它 `valarrays` 和 `valarrays` 子集组合起来 (参见 p554)。

下例展示 `valarray` 之中的 `masked subset` (屏蔽式子集) 用法:

```

// num/masked1.cpp

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

// print valarray line-by-line
template<class T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; ++i) {
        for (int j=0; j<num; ++j) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    /* valarray with 12 elements
     * - four rows
     * - three columns
     */
    valarray<double> va(12);
}

```

```
// fill valarray with values
for (int i=0; i<12; i++) {
    va[i] = i;
}

printValarray (va, 3);

// assign 77 to all values that are less than 5
va[va<5.0] = 77.0;

// add 100 to all values that are greater than 5 and less than 9
va[va>5.0 && va<9.0]
    = valarray<double>(va[va>5.0 && va<9.0]) + 100.0;

printValarray (va, 3);
}
```

程序输出如下:

```
0 1 2
3 4 5
6 7 8
9 10 11

77 77 77
77 77 5
106 107 108
9 10 11
```

注意, 用来和 `valarray` 相比较的那个单值, 其型别必须和 `valarray` 的元素型别完全一致。所以如果你以 `int` 单值和“`doubles` 型的 `valarray`”比较, 将无法通过编译:

```
valarray<double> va(12);
...
va[va < 5] = 77; // ERROR
```

Indirect Subsets (间接式子集)

定义 valarray 子集的第四个 (也是最后一个) 方法, 是运用 indirect arrays。只需传递一个索引数组, 就可以定义出 valarray 的一个子集。注意, 用来指定子集的索引无需排序, 并且可以重复出现。

除此之外, indirect arrays 跟所有 valarray 子集都一样:

1. 若要定义某个 valarray 的一个具体子集 (concrete subset), 可以将 “元素型别为 size_t” 的 valarray 作为参数传给 valarray 的 subscript (下标) 操作符。
2. 如果 valarray 是常量, 则子集表达式将导致一个新的 valarray。
3. 如果 valarray 不是常量, 则子集表达式将导致一个 indirect_array, 后者以 reference 语义来表现 valarray 的相应元素:

```
namespace std {
    template <class T>
        class indirect_array;

    template <class T>
        class valarray {
        public:
            // indexing a constant valarray returns a new valarray
            valarray<T> operator[] (const valarray<size_t>&) const;

            // indexing a variable valarray returns an indirect_array
            indirect_array<T> operator[] (const valarray<size_t>&);
            ...
        };
}
// 译注: 我自己在这里曾有一些迷惑, 兹将心得提醒各位读者。上述第 2 点说的 const
// valarray, 唯一能够调用的就是 operator[] const 版本。上述第 3 点
// 说的 non-const valarray, 能够调用 operator[] 的 non-const 版本
// 和 const 版本; 但是当两版本同时出现 (如上, 形成重载), 根据重载操作
// 符的决议规则, 调用的将是 non-const 版本。
// 以上讨论都和 operator[] 的参数的常数性无关。无论是 non-const
// 对象或 const 对象, 都可以 pass-by-reference-to-const。
```

4. indirect_array 提供赋值操作符和赋值复合操作符来修改子集中的元素。
5. 通过型别转换, 你可以将 indirect array 和其它 valarrays 以及 valarrays 子集组合起来 (参见 p554)。

下例展示 valarray 中的 indirect array 用法:

```
// num/ind11.cpp

#include <iostream>
#include <valarray>
```



```
using namespace std;

// print valarray as two-dimensional array
template<class T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; i++) {
        for (int j=0; j<num; j++) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    // create valarray for 12 elements
    valarray<double> va(12);

    // initialize valarray by values 1.01, 2.02, ... 12.12
    for (int i=0; i<12; i++) {
        va[i] = (i+1) * 1.01;
    }
    printValarray(va,4);

    /* create array of indexes
    * - note: element type has to be size_t
    */
    valarray<size_t> idx(4);
    idx[0] = 8;
    idx[1] = 0;
    idx[2] = 3;
    idx[3] = 7;

    // use array of indexes to print the ninth, first, fourth, and eighth elements
    printValarray(valarray<double>(va[idx]), 4);

    // change the first and fourth elements and print them again indirectly
    va[0] = 11.11;
    va[3] = 44.44;
    printValarray(valarray<double>(va[idx]), 4);
}
```

```
// now select the second, third, sixth, and ninth elements
// and assign 99 to them
idx[0] = 1;
idx[1] = 2;
idx[2] = 5;
idx[3] = 8;
va[idx] = 99;

// print the whole valarray again
printValarray (va, 4);
}
```

valarray idx 用来定义 valarray va 的一个子集。程序输出如下：

```
1.01 2.02 3.03 4.04
5.05 6.06 7.07 8.08
9.09 10.1 11.11 12.12

9.09 1.01 4.04 8.08

9.09 11.11 44.44 8.08

11.11 99 99 44.44
5.05 99 7.07 8.08
99 10.1 11.11 12.12
```

12.2.3 Class valarray 细部讨论

class valarray<> 是 valarray 组件的核心部分，被定义为一个 template class，针对元素型别实施参数化：

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

注意，大小并非型别的一部分。也就是说，原则上你可以处理不同大小的 valarrays，而且你可以改变大小。但如果想要改变大小，必须通过一个双步骤的初始化动作完成（产生对象，然后设定大小），过程中不可避免要操作 valarrays 所形成的数组。注意，将不同大小的 valarrays 进行组合，其结果未有定义。

构造 (Create)、复制 (Copy)、销毁 (Destroy)

valarray::valarray ()

- 缺省构造函数。
- 产生出一个空的 **valarray**。
- 此构造函数只是产生一个 **valarrays** 数组, 下一步需以成员函数 **resize()** 设定正确大小。

explicit valarray::valarray (size_t num)

- 产生一个包含 *num* 个元素的 **valarray**。
- 各元素以其缺省构造函数完成初始化动作 (基本型别的初值为 0)。

valarray::valarray (const T& value, size_t num)

- 产生一个包含 *num* 个元素的 **valarray**。
- 各元素以 *value* 初始化。
- 注意, 参数的次序很特别。其它所有 C++ 标准程序库的接口都是 *num* 在前而 *value* 在后。

valarray::valarray (const T* array, size_t num)

- 产生一个包含 *num* 个元素的 **valarray**。
- 各元素以 *array* 中的对应元素为初值。
- 调用者必须确保 *array* 内含 *num* 个元素, 否则会引发未定义行为。

valarray::valarray (const valarray& va)

- **copy** 构造函数。
- 产生 **valarray** *va* 的一个复制品。

valarray::~valarray ()

- 析构函数
- 销毁所有元素, 释放内存。

此外, 你可以产生 **valarrays** 并运用内部辅助类别如 **slice_array**, **gslice_array**, **mask_array**, **indirect_array** 来初始化。相应细节分别见于 p575, p577, p578 和 p579。

赋值 (assign)

valarray& valarray::operator= (const valarray& va)

- 将 **valarray** *va* 的元素赋值给 **this*。

- 如果 `va` 的大小和 `*this` 不同, 则行为未可预期。
- 任何 `valarray` 赋值运算的左侧元素都不可取决于其右侧的元素。换句话说, 如果一个赋值操作覆盖了其右侧值, 结果未可预期。这意味表达式右侧所含的任何元素都不可以出现在表达式左侧。之所以如此, 因为 `valarray` 的评估 (evaluation) 次序没有明确定义。细节见 p557 和 p554。

```
valarray& valarray::operator= (const T& value)
```

- 将 `value` 赋值给此一 `valarray` 的所有元素⁶。
- `valarray` 大小不变。指向元素的所有 `pointer` 和 `reference` 也都继续有效。

此外, 你也可以产生一个 `valarrays`, 再以内部的辅助类别如 `slice_array`, `gslice_array`, `mask_array`, `indirect_array` 进行初始化。相应细节分别见于 p575, p577, p578 和 p579。

成员函数

`class valarray` 提供下列成员函数:

```
size_t valarray::size () const
```

- 返回当前的元素数量⁷。

```
void valarray::resize (size_t num)
```

```
void valarray::resize (size_t num, T value)
```

- 两种形式都可将 `valarray` 的大小改为 `num`。
- 如果大小增长, 则新的元素分别以缺省构造函数初始化, 或以 `value` 为初值。
- 两种形式都会使指向元素的各个 `pointers` 或 `references` 失效。
- 这些函数仅用来帮助产生 `valarrays` 数组。缺省构造函数产生出数组之后, 你应该调用这些函数来设定正确大小。

```
T valarray::min () const
```

```
T valarray::max () const
```

- 第一形式返回最小元素值。
- 第二形式返回最大元素值。
- 元素以 `operator<` 或 `operator>` 进行比较, 所以元素型别必须支持这两个操作符。
- 如果 `valarray` 无元素, 返回值没有定义。

⁶ 早期版本使用成员函数 `fill()` 来赋单值。

⁷ 早期版本中, 成员函数 `size()` 的名字是 `length()`。

```
T valarray::sum () const
```

- 返回所有元素的和。
- 元素以 `operator+=` 处理，所以元素型别必须支持 `operator+=` 操作符。
- 如果 `valarray` 无元素，返回值未有定义。

```
valarray valarray::shift (int num) const
```

- 返回一个新的 `valarray`，其中的元素是 `*this` 内的元素移动 `num` 个位置后得到的结果。
- 返回的 `valarray` 拥有相同元素个数。
- 被移空的元素位置，以缺省构造函数设初值（新值）。
- 移动方向由 `num` 的正负号决定。
 - 如果 `num` 为正，就向左（前）移动，各元素对应的索引减小。
 - 如果 `num` 为负，就向右（后）移动，各元素对应的索引增大。

```
valarray valarray::cshift (int num) const
```

- 返回一个新的 `valarray`，其中的元素是 `*this` 中的元素循环移动 `num` 个位置后得到的结果。
- 返回的 `valarray` 拥有相同元素个数。
- 移动的方向由 `num` 的正负号决定。
 - 如果 `num` 为正，就向左（前）移动，各元素对应的索引减小，或被安插于后。
 - 如果 `num` 为负，就向右（后）移动，各元素对应的索引增大，或被安插于前。

```
valarray valarray::apply (T op(T)) const
```

```
valarray valarray::apply (T op(const T&)) const
```

- 两种形式都返回新的 `valarray`，其内容为原 `valarray` 的所有元素以 `op()` 处理后的结果。
- 返回的 `valarray` 拥有相同元素个数。
- 针对 `*this` 的每一个元素调用 `op(elem)`

并将结果作为新的 `valarray` 的对应元素的初值，然后将新的 `valarray` 返回。

元素存取

```
T& valarray::operator[] (size_t idx)
```

```
T valarray::operator[] (size_t idx) const
```

- 两种形式都返回 `valarray` 之中索引为 `idx` 的元素（注意，第一元素的索引为 0）。
- Non-const 版本返回一个 `reference`。因此你可以运用这个操作符的回返值更改相应元素。只要 `valarray` 存在，而且没有调用“可改变 `valarray` 大小”的函数，那么返回的 `reference` 保证有效。

Valarray 的操作符

valarray 的一元操作符 (unary operators) 遵循以下格式:

```
valarray valarray::unary-op () const
```

- 返回一个新的 valarray, 其中元素是 *this 的相应元素经过 unary-op 处理后的结果。
- unary-op 可以是下列之一:
 - operator +
 - operator -
 - operator ~
 - operator !
- operator! 的返回值型别是 valarray<bool>。

valarray 的二元操作符 (binary operators) 遵循以下格式 (“比较”和“赋值”除外):

```
valarray binary-op (const valarray& va1, const valarray& va2)
valarray binary-op (const valarray& va, const T& value)
valarray binary-op (const T& value, const valarray& va)
```

- 返回一个新的 valarray, 其元素个数与 va, va1 或 va2 相同。新的 valarray 包含每一组 (两个) 值经过 binary-op 处理后的结果。
- 如果操作数是个单值 value, 它将和 va 中的每一个值组合运算。
- binary-op 可以是下列之一:
 - operator +
 - operator -
 - operator *
 - operator /
 - operator %
 - operator ^
 - operator &
 - operator |
 - operator <<
 - operator >>
- 如果 va1 和 va2 的元素数量不同, 则结果未可预期。

逻辑操作符和比较操作符的情况和上面相同, 只不过返回的是个充满布尔值的 valarray:

```
valarray<bool> logical-op (const valarray& va1, const valarray& va2)
valarray<bool> logical-op (const valarray& va, const T& value)
valarray<bool> logical-op (const T& value, const valarray& va)
```

- 返回一个新的 `valarray`，其元素个数和 `va`, `va1` 或 `va2` 相同。新的 `valarray` 包含每一组（两个）值经过 `logical-op` 处理后的结果。
- 如果操作数是个单值 `value`，它将与 `va` 中的每一个值组合运算。
- `logical-op` 可以是下面其中之一：


```
operator ==
operator !=
operator <
operator <=
operator >
operator >=
operator &&
operator ||
```
- 如果 `va1` 和 `va2` 元素数量不同，则结果未可预期。

类似情况，`valarrays` 的赋值复合操作符定义如下：

```
valarray& valarray::assign-op (const valarray& va)
valarray& valarray::assign-op (const T& value)
```

- 这两种形式都针对 `*this` 中的每一个元素调用 `assign-op`，并以 `va` 中的对应元素或单值 `value` 作为第二个操作数。
- 返回一个 `reference`，指向改变后的 `valarray`。
- `assign-op` 可以是下列其中之一：


```
operator +=
operator -=
operator *=
operator /=
operator %=
operator ^=
operator &=
operator |=
operator <<=
operator >>=
```
- 如果 `*this` 和 `va2` 元素数量不同，则结果未可预期。
- 只要 `valarray` 存在，而且我们并未调用“可改变 `valarray` 大小”的函数，则任何指向“改变后之 `valarray`”的 `reference` 和 `pointer`，都保证持续有效。

超越函数 (Transcendental Functions)

```
valarray abs (const valarray& va)
valarray pow (const valarray& va1, const valarray& va2)
valarray pow (const valarray& va, const T& value)
```

```

valarray pow (const T& value, const valarray& va)
valarray exp (const valarray& va)
valarray sqrt (const valarray& va)
valarray log (const valarray& va)
valarray log10 (const valarray& va)
valarray sin (const valarray& va)
valarray cos (const valarray& va)
valarray tan (const valarray& va)
valarray sinh (const valarray& va)
valarray cosh (const valarray& va)
valarray tanh (const valarray& va)
valarray asin (const valarray& va)
valarray acos (const valarray& va)
valarray atan (const valarray& va)
valarray atan2 (const valarray& va1, const valarray& va2)
valarray atan2 (const valarray& va, const T& value)
valarray atan2 (const T& value, const valarray& va)

```

- 上述每个函数都返回一个新的 `valarray`，其元素个数和 `va`, `va1` 或 `va2` 相同。新的 `valarray` 内含的元素是每一个（或一对）元素被施行相应某种运算后的结果。
- 如果 `va1` 和 `va2` 元素数量不同，则结果未可预期。

12.2.4 Valarray 子集类别 (Subset Classes) 细部讨论

本节详细讲解 `valarray` 的子集类别。这些类别非常简单，并未提供很多操作函数。往往只是给出其声明式，再加上一点注解而已。

`class slice` 和 `class slice_array`

如果以一个 `slice` 作为 `non-const valarray` 的索引，就可以得到 `slice_array` 对象：

```

namespace std {
    template<class T>
    class valarray {
    public:
        ...
        slice_array<T> operator[] (slice);
        ...
    };
}

```


`class slice` 的 `public` 接口确切定义如下:

```
namespace std {
    class slice {
    public:
        slice (); // empty subset
        slice (size_t start, size_t size, size_t stride);
        size_t start() const;
        size_t size() const;
        size_t stride() const;
    };
}
```

缺省构造函数产生一个空子集。你可以运用 `start()`, `size()`, `stride()` 等成员函数询问某个 `slice` 的属性。

`Class slice_array` 支持以下操作:

```
namespace std {
    template <class T>
    class slice_array {
    public:
        typedef T value_type;

        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<<= (const valarray<T>&) const;
        void operator>>= (const valarray<T>&) const;
        ~slice_array();
    private:
        slice_array();
        slice_array(const slice_array&);
        slice_array& operator=(const slice_array&);
        ...
    };
}
```

注意, `slice_array<>` 的角色完全是为 `slices` 提供内部辅助, 因此它对用户而言应该是透明的。也因此 `slice_array<>` 的所有构造函数和赋值操作符都是 `private`。

`class gslice` 和 `class gslice_array`

将 `gslice` 作为某个 `non-const valarray` 的索引, 便可获得 `slice_array` 对象:

```
namespace std {
    template<class T>
    class valarray {
    public:
        ...
        gslice_array<T> operator[] (const gslice&);
        ...
    };
}
```

`class gslice` 的 `public` 接口确切定义如下:

```
namespace std {
    class gslice {
    public:
        gslice ();           // empty subset
        gslice (size_t start,
            const valarray<size_t>& size,
            const valarray<size_t>& stride);
        size_t start() const;
        valarray<size_t> size() const;
        valarray<size_t> stride() const;
    };
}
```

缺省构造函数会产出一个空子集。你可以运用 `start()`, `size()`, `stride()` 等成员函数询问某个 `gslice` 的属性。

`Class gslice_array` 支持如下操作:

```

namespace std {
    template <class T>
    class gslice_array {
    public:
        typedef T value_type;
        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<<= (const valarray<T>&) const;
        void operator>>= (const valarray<T>&) const;
        ~gslice_array();
    private:
        gslice_array();
        gslice_array(const gslice_array<T>&);
        gslice_array& operator=(const gslice_array<T>&);
        ...
    };
}

```

和 `slice_array<>` 一样, `gslice_array<>` 扮演的完全是内部辅助类别的角色, 对用户而言应该是透明的。所以 `gslice_array<>` 的所有构造函数和赋值操作符都是 `private`。

`class mask_array`

将 `valarray<bool>` 当做某个 `non-const valarray` 的索引, 便可获得一个 `slice_array` 对象。

```

namespace std {
    template<class T>
    class valarray {
    public:
        ...
        mask_array<T> operator[] (const valarray<bool>&);
        ...
    };
}

```

`class mask_array` 支持下列操作:

```
namespace std {
    template <class T>
    class mask_array {
    public:
        typedef T value_type;

        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<=<= (const valarray<T>&) const;
        void operator>=>= (const valarray<T>&) const;
        ~mask_array();

    private:
        mask_array();
        mask_array(const mask_array<T>&);
        mask_array& operator=(const mask_array<T>&);
        ...
    };
}
```

再一次强调, `mask_array<>` 扮演的角色完全是为了提供内部辅助, 对用户而言应该是透明的。因此 `mask_array<>` 的所有构造函数和赋值操作符都是 `private`。

`class indirect_array`

将一个 `valarray<size_t>` 作为 `non-const valarray` 的索引, 便可获得一个 `slice_array` 对象:

```

namespace std {
    template<class T>
    class valarray {
    public:
        ...
        indirect_array<T> operator[](const valarray<size_t>&);
        ...
    };
}

```

class mask_array 支持下列操作:

```

namespace std {
    template <class T>
    class indirect_array {
    public:
        typedef T value_type;

        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<<= (const valarray<T>&) const;
        void operator>>= (const valarray<T>&) const;
        ~indirect_array();

    private:
        indirect_array();
        indirect_array(const indirect_array<T>&);
        indirect_array& operator=(const indirect_array<T>&);
        ...
    };
}

```

`indirect_array<>` 扮演的角色完全是提供内部辅助, 对用户应该是透明的。所以, `indirect_array<>` 的所有构造函数和赋值操作符都是 `private`。

12.3 全局性的数值函数

头文件 `<cmath>` 和 `<cstdlib>` 提供了从 C 继承下来的全局性数值函数。表 12.8 和表 12.9 列出了这些函数[■]。

表 12.8 头文件 `<cmath>` 定义的函数

函数	效果
<code>pow()</code>	求幂函数
<code>exp()</code>	指数函数
<code>sqrt()</code>	平方根
<code>log()</code>	自然对数
<code>log10()</code>	以 10 为底的对数
<code>sin()</code>	正弦函数
<code>cos()</code>	余弦函数
<code>tan()</code>	正切函数
<code>sinh()</code>	双曲正弦函数
<code>cosh()</code>	双曲余弦函数
<code>tanh()</code>	双曲正切函数
<code>asin()</code>	反正弦函数
<code>acos()</code>	反余弦函数
<code>atan()</code>	反正切函数
<code>atan2()</code>	商的反正切函数
<code>ceil()</code>	大于某个浮点数的最小整数
<code>floor()</code>	小于某个浮点数的最大整数
<code>fabs()</code>	浮点数的绝对值
<code>fmod()</code>	浮点数相除的余数 (modulo)
<code>frexp()</code>	将一个浮点数转换成小数部分和整数部分
<code>ldexp()</code>	将某个浮点数乘以 2 的某个整数幂次方
<code>modf()</code>	将浮点数分离为一个带正负号的整数和一个分数

和 C 不同的是, C++ 将某些操作函数针对不同型别进行了重载, 使得某些 C 函数失去了价值。例如 C 提供 `abs()`, `labs()`, `fabs()` 分别计算 `int`, `long`, `double` 的绝对值, 而 C++ 的 `abs()` 完成了重载, 适用于上述各种不同型别。

[■] 由于历史因素, 某些数值函数定义于 `<cstdlib>` 而非 `<cmath>`。

表 12.9 头文件 <cstdlib> 内定义的数值函数

函数	效果
abs()	求某个 int 的绝对值
labs()	求某个 long 的绝对值
div()	求 int 相除的商和余数
ldiv()	求 long 相除的商和余数
srand()	随机数产生器 (种下新的随机数种子)
rand()	随机数产生器 (取得一个随机数)

明确地说, 所有针对浮点数的数值函数, 都对 float, double, long double 三个型别实现了重载。不过这也带来一个严重的副作用: 当你传入一个整数, 会引起模棱两可的情况⁹:

```
std::sqrt(7) // AMBIGUOUS: sqrt(float), sqrt(double), or
              // sqrt(long double)?
```

你必须这么写:

```
std::sqrt(7.0) // OK
```

如果你用的是一个变量, 则必须这么写:

```
int x;
...
std::sqrt(float(x)) // OK
```

程序库设计者处理这一问题的办法南辕北辙: 有的不提供重载, 有的遵循标准规格 (于是针对所有浮点型别都提供重载), 有的针对所有数值型别加以重载, 有的允许你借助预处理器 (pre-processor) 在不同策略之间切换。因此, 应用程序开发过程中, 上述的模棱两可情况可能发生, 也可能不发生。如果要写出可移植性高的代码, 你应该总是要求参数型别有精确的匹配。

⁹ 感谢 David Vandevorde 指出这一点。

13

以 Stream Classes 完成输入和输出

用于 I/O（输入/输出）的各个 classes，是 C++ 标准程序库的重要组成。一个程序如果没有 I/O，必然没什么用处。标准程序库中的 I/O classes 不仅局限于文件、屏幕或键盘，事实上它们形成了一套富有弹性的框架（framework），可用来将任意数据格式化，可处理（存取）任意外部表述（external representations）。

一如其名称所示，IOStream 程序库提供了一系列 I/O classes。它们在标准化之前已得到广泛应用，出道之早，在标准程序库中仅此一家。早期 C++ 系统配送一套由 AT&T 开发的 I/O classes，可视为当时的一个“准标准”。虽然为了标准程序库的一致性和某些新需要，I/O classes 作了数度修改，但其基本原则始终如一。

本章先对其中最重要的组件和技术作一个总体介绍，然后用实例说明 IOStream 程序库的实际应用。其应用范围可从简单格式化到多个新的外部表述（new external representations）的整合（这可是个常常被搞错的主题）。

本章并不打算讨论 IOStream 程序库的所有细节，因为光是这样就需要一整本书。如果想查询本章没有提到的一些细枝末节，请参考 IOStream 程序库专门书籍或 C++ 标准程序库参考手册。

非常感谢 Dietmar Kühl，他是 I/O 及国际化（internationalization, i18n）方面的专家。他对本书提出了很多有价值的意见，并亲自撰写了本章的一部分。

IOStream 程序库最近的变化

这一小节是特别为那些习惯于 IOStream 老式风格的朋友们，略述标准化过程中的一些变化。基本的 I/O stream classes 并没有变，但为了更方便定制（customization），引入了一些重要特性。以下即是一些主要的变化：

- I/O 被国际化了 (internationalized)。
- 用于字符数组 (型别为 `char*`) 的 `string stream classes`, 被标准程序库提供的 `string` 取代。前者仍然保留, 从而能向下兼容, 但不建议使用¹。
- “异常 (exception) 处理” 已和 “状态 (state) 处理” 及 “错误 (error) 处理” 整合。
- 支持赋值操作 (assignment) 的 `IOStream classes` (名称以 `_withassign` 结尾者), 改用新方法支持赋值操作。新方法对所有 `stream classes` 均适用。
- `IOStream classes` 均已泛型化, 可支持不同的字符表述 (character representation)。由此而来的副作用是, `stream classes` 的前置声明不再合法。你可含入某特定头文件以提供适当的声明。因此, 你不应当再写这样:

```
class ostream;           // wrong
```

你应该使用新的头文件:

```
#include <iosfwd>        // OK
```

- 和标准程序库的其它部分一样, `IOStream` 的所有符号都定义于命名空间 `std` 内。

译注: 根据我对各家源码的实际观察, `IOStream` 的实作目前还十分混乱, 例如 GNU C++ 的许多版本尚未支持 `template`, VC 和 BCB 的相关文件组态也不十分遵循 C++ *Standard* 的规定 (都是历史因素)。因此你在本章看到的描述, 不见得都能够在你手上的编译器实作版本中获得证实 (尤其是文件组态)。应有的功能倒是都有。

13.1 IOStreams 基本概念

具体讨论 `stream classes` 之前, 我将简要叙述 `streams` 的基本概念, 熟悉 `IOStream` 基本知识的读者可以跳过本节。

13.1.1 Stream 对象

C++ I/O 由 `streams` 完成。所谓 `stream` 就是一条数据“流”, 字符序列在其中“川流不息”。按面向对象原则, `stream` 是由某个类别定义出来的具有特定性质的对象。输出操作被解读为“数据流入 `stream`”, 输入操作则是“数据流出 `stream`”。另有一些为标准 I/O 通道 (channels) 而定义的全局对象。

13.1.2 Stream 类别

正如有不同种类的 I/O (输入、输出、文件存取), 对应地也有不同的 `stream classes`, 其中最重要的是:

¹ 所谓“不建议”, 意味有更好的替代品, 故不推荐, 而且以后的标准规格很可能去掉旧式用法。

- **class istream**

定义 input stream, 可用来读取数据。

- **class ostream**

定义 output stream, 可用来写出数据。

两者分别具体实现自 `template class basic_istream<>` 和 `basic_ostream<>`, 以 `char` 作为字符型别。事实上整个 IOStream 程序库均不依赖任何特定的字符型别, 而以一个 `template` 参数替代之。这样的参数化在 `string classes` 也存在, 并且用于国际化议题 (第 14 章)。

以下各节集中讨论所谓 `narrow streams` (亦即以 `char` 为字符型别) 的输入输出。后继的讨论会扩展至其它字符型别。

13.1.3 全局性的 Stream 对象

IOStream 程序库定义了数个型别为 `istream` 和 `ostream` 的全局对象, 它们对应于标准的 I/O 通道 (channels):

- **cin**

`cin` (隶属于 `istream`) 是供使用者输入用的标准输入通道, 对应于 `C stdin`。操作系统通常将它和键盘连接。

- **cout**

`cout` (隶属于 `ostream`) 是供使用者输出用的标准输出通道, 对应于 `C stdout`。操作系统通常将它和监视器 (屏幕) 连接。

- **cerr**

`cerr` (隶属于 `ostream`) 是所有错误信息所使用的标准错误输出通道, 对应于 `C stderr`。操作系统通常也将它和监视器 (屏幕) 连接。缺省情况下 `cerr` 无缓冲装置。

- **clog**

`clog` (隶属于 `ostream`) 是标准日志通道, `C` 没有对应物。缺省情况下操作系统将它连接于 `cerr` 所连接的装置, 但 `clog` 设有缓冲装置。

将“正常输出”和“错误信息的输出”加以分离, 可以让程序以不同的方式对待两种不同的输出。例如可以将正常输出重新定向至某个文件, 而同时仍然令错误信息显示于控制台 (`console`)。当然, 前提是操作系统必须支持标准 I/O 通道的重定向功能 (`redirection`, 大部分操作系统的确可以)。这种分离方式起源于 UNIX 的 I/O 重定向概念。

13.1.4 Stream 操作符

`operator>>` 和 `operator<<` 被相应的 stream classes 重载, 分别用于输入和输出。因此, C++ 移位操作符摇身一变成了 I/O 操作符²。运用此二者, 便可以把多个 I/O 操作串成一系列。

下面这段程序从标准输入端读入两个整数 (只能是整数), 并写到标准输出端:

```
int a, b;

// as long as input of a and b is successful
while (std::cin >> a >> b) {
    // output a and b
    std::cout << "a: " << a << " b: " << b << std::endl;
}
```

13.1.5 操控器 (Manipulators)

在大部分输出语句的最后, 我们都会写一个如下的东西, 称为操控器 (manipulator):

```
std::cout << std::endl
```

操控器是专门用来操控 stream 的对象, 常常只会改变输入或格式化输出的解释方式, 例如数值进制 `dec` (10 进位), `hex` (16 进位), `oct` (8 进位)。用于 `ostreams` 的操控器并不会凭空造出输出数据, 用于 `istreams` 的操控器也不会吃掉任何输入数据。有些操控器会引发立即操作, 例如用于“刷新 (*flush*) output 缓冲区”或“跳过 input 缓冲区空格”的那些操控器。

操控器 `endl` 的意思是终止一行 (end line)。它做两件事情:

1. 输出换行符号 '\n'
2. 刷新 output 缓冲区 (也就是说对于给定的 stream, 应用其 `flush()` 将缓冲区内的所有数据强制输出)

`IOStream` 程序库定义的一些最重要的操控器显示于表 13.1。

13.6 节, p612 有操控器的更多讨论, 包括 `IOStream` 程序库已定义的一些操控器, 以及如何定义你自己的操控器。

² 由于这两个操作符实际上在 stream 中负责插入和提取字符, 所以也有人称它们为 *inserter* 和 *extractor*。

表 13.1 IOStream 程序库中最重要的一些操控器

操控器 (Manipulator)	类别 (Class)	意义
endl	ostream	输出 '\n' 并刷新 output 缓冲区
ends	ostream	输出 '\0'
flush	ostream	刷新 output 缓冲区
ws	istream	读入并忽略空格

13.1.6 一个简单的例子

以下展示 stream classes 的应用。此程序读入两个浮点数，并输出其乘积：

```
// io/io1.cpp

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

int main()
{
    double x, y; // operands

    // print header string
    cout << "Multiplication of two floating point values" << endl;

    // read first operand
    cout << "first operand: ";
    if (! (cin >> x)) {
        /* input error
         * => error message and exit program with error status
         */
        cerr << "error while reading the first floating value"
              << endl;

        return EXIT_FAILURE;
    }

    // read second operand
    cout << "second operand: ";
    if (! (cin >> y)) {
        /* input error
         * => error message and exit program with error status
         */
        cerr << "error while reading the second floating value"
              << endl;
        return EXIT_FAILURE;
    }

    // calculate and print result
    double result = x * y;
    cout << "result: " << result << endl;
}
```

```
    }  
  
    // print operands and result  
    cout << x << " times " << y << " equals " << x * y << endl;  
}
```

13.2 基本的 Stream 类别和 Stream 对象

13.2.1 相关类别及其阶层体系

IOStream 程序库中的 stream classes 形成了图 13.1 所示的阶层体系。图中所示的 template classes，上一行代表其名称，下一行是以字符型别 char 和 wchar_t 具现化 (instantiated) 之后的名字。

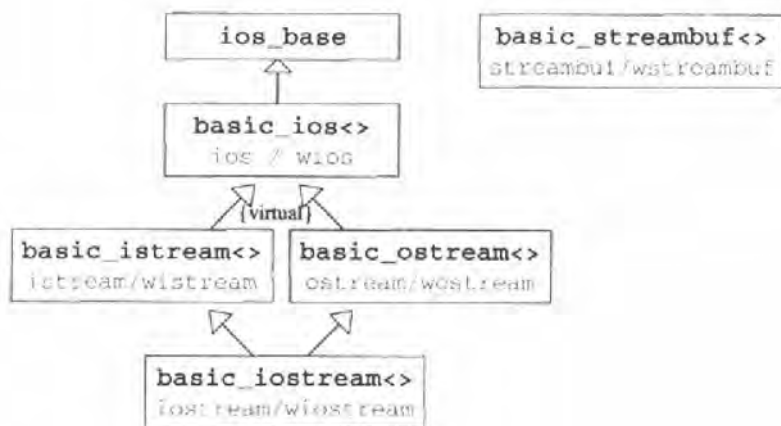


图 13.1 基本的 Stream Classes 阶层体系

这个阶层体系中的 classes 分别扮演以下角色：

- 基类 `ios_base` 定义了 stream classes 的所有“与字符型别及其相应之字符特性 (traits) 无关”的属性，主要包含状态和格式标志等组件和函数。
- 由 `ios_base` 派生的 `template class basic_ios<>`，定义出“与字符型别及其相应之字符特性 (traits) 相关”的 stream classes 共同属性，其中包括 stream 所用的缓冲器。缓冲器所属类别派生自 `template class basic_streambuf<>`，其具现参数和 `basic_ios<>` 一致。`basic_streambuf<>` 负责实际的读写操作。
- `template class basic_istream<>` 和 `basic_ostream<>` 两者都虚拟继承自 `basic_ios<>`，分别定义出用于读/写的对象。和 `basic_ios<>` 一样，它们以字符型别及其特性 (traits) 作为参数。如果无关乎国际化议题，一般使用由字符型别 `char` 体现出来的 `istream` 和 `ostream` 就够了。
- `template class basic_iostream<>` 派生（多重继承）自 `basic_istream<>` 和 `basic_ostream<>`，用来定义既可读取亦可改写的对象。
- `template class basic_streambuf<>` 是 IOStream 程序库的核心，定义出所有“可改写的 stream”或“可读取的 stream”的接口。其它 stream classes 均利用它进行实际的字符读写操作。程序中为处理某些外部表达 (external representation)，必须从 `basic_streambuf<>` 派生一些可用类别，详见下一小节。

Stream Buffer Classes 的用途

IOStream 程序库严格依照“职责分离”的原则来设计。`basic_ios` 派生类别只处理数据的格式化³，实际读写操作由 `basic_ios` 派生类别所维护的 stream buffers 完成。stream buffers 提供读写时所使用的字符缓冲区，并形成对外部表述（如文件和字符串）的一种抽象概念。

因此，对着新的外部表述（例如 sockets 或 GUI 组件）进行 I/O 操作，或是对 streams 重定向，或是组合 stream 以形成 pipelines（例如在改写另一个 stream 之前先压缩数据）时，stream buffers 扮演重要的角色。此外，对同一个外部表述进行 I/O 操作时，stream buffer 与 I/O 同步。细节详见 13.10.2 节，p638。

运用 stream buffers，我们可以轻松定义出对于新的外部表述（例如某种新的储存设备）的存取操作。我们需要做的仅仅是从 `basic_streambuf<>` 派生出一个

³ 事实上它们连格式化也不亲自处理！它们把格式化操作委托给 locale 程序库中相应的 facets 进行。关于 facets 细节请看 p698, 14.2.2 节。

新的 stream buffer 类别（或其适当特化版本），并定义该外部表述的字符读写函数即可。如果某个 stream 对象初始化时候使用了该 stream buffer，则所有 I/O 格式化操作的选项均可自动生效。13.13 节, p663 说明如何定义一个 stream buffer 用以存取特定的储存设备。

具体的类别定义

和 IOStream 程序库的所有 template classes 一样, basic_ios<> 有两个参数:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
        class basic_ios;
}
```

这两个参数分别是 (1) stream classes 所使用的字符型别, (2) 前者的特性类别 (traits class)。

在特性类别中, *end-of-file*⁴值和复制/移动字符序列的各个指令, 都属于特性 (traits) 的一部分。字符型别的特性 (traits) 和字符型别本身通常密不可分, 因此定义一个 template class 并针对特定型别实施特化也就合情合理。针对字符型别 charT, 特性类别缺省为 char_traits<charT>。关于 char_traits<>, 标准程序库提供了 char 和 wchar_t 两个特化版本。字符特性的更多细节请看 14.1.2 节, p687。

下面是两个最常使用的 basic_ios<> 具现实体:

```
namespace std {
    typedef basic_ios<char> ios;      // 译注: 针对 char 的特化版本
    typedef basic_ios<wchar_t> wios; // 译注: 针对 wchar_t 的特化版本
}
```

其中 ios 对应于旧式 (起源于 AT&T 的) IOStream 程序库基类, 用来保证对旧式 C++ 程序的兼容性。

basic_ios 所用的 stream buffer class, 定义十分类似:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
        class basic_streambuf;
    typedef basic_streambuf<char> streambuf; // 译注: char 特化版本
    typedef basic_streambuf<wchar_t> wstreambuf; // 译注: wchar_t 特化版本
}
```

⁴ 我用术语 *end-of-file* 表示数据输入结束符号, 相当于 C 常数 EOF。

`template class basic_istream<>, basic_ostream<>, basic_iostream<>` 当然也以字符型别和特性类别 (traits class) 作为参数:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_istream;
    template <class charT,
              class traits = char_traits<charT> >
    class basic_ostream;
    template <class charT,
              class traits = char_traits<charT> >
    class basic_iostream;
}
```

下面是针对两个最重要的字符型别 (char 和 wchar_t) 所做的型别定义:

```
namespace std {
    typedef basic_istream<char> istream;
    typedef basic_istream<wchar_t> wistream;
    typedef basic_ostream<char> ostream;
    typedef basic_ostream<wchar_t> wostream;
    typedef basic_iostream<char> iostream;
    typedef basic_iostream<wchar_t> wiostream;
}
```

型别 `istream` 和 `ostream` 常用于西半球 (译注: 东方文字大多是双字节编码), 和老式的 stream classes (源于 AT&T) 基本上兼容。

C++ *Standard* 不再支持旧式 `IOStream` 原本有的三个类别: `istream_withassign`, `ostream_withassign` 和 `iostream_withassign` (分别派生自 `istream`, `ostream` 和 `iostream`), 它们的相关功能另有实现方式 (见 13.10.3 节, p641)。

另有一些 classes 用来对文件和字符串进行 I/O 格式化, 相关讨论请看 13.9 节, p627, 以及 13.11 节, p645。

13.2.2 全局性的 Stream 对象

`IOStream` 程序库定义了数个全局性的 stream objects, 前面也已提过, 它们分别以 `char` 或对应的 `wchar_t` 作为字符型别, 用来存取标准 I/O 通道。请见表 13.2。

表 13.2 全局性的 stream 对象

型别	名称	用途
istream	cin	从标准 input 通道读入数据
ostream	cout	将一般数据写至标准 output 通道
ostream	cerr	将错误信息写至标准 error 通道
ostream	clog	将日志信息写至标准 logging 通道
wistream	wcin	从标准 input 通道读入宽字符数据
wostream	wcout	将宽字符一般数据写至标准 output 通道
wostream	wcerr	将宽字符错误信息写至标准 error 通道
wostream	wclog	将宽字符日志信息写至标准 logging 通道

缺省情况下，这些 stream 都和标准 C stream 同步。也就是说 C++ 标准程序库确保在“混合使用 C++ streams 和 C streams”的情况下，顺序有保障。任何标准 C++ stream 缓冲区在改写数据前，都会先刷新其所对应的 C Stream 缓冲区，反之亦然。当然，保持同步会占用一定时间。如果你不需要这样的功能，可以在输入或输出前调用 `sync_with_stdio(false)`，便可取消同步（p682）。

13.2.3 头文件 (Headers)

各个 stream classes 的定义分散于以下数个头文件：

- **<iosfwd>**
内含 stream classes 的前置声明。这个文件是必要的，因为前置声明不能只是简单地写一句诸如 `class ostream` 这样的声明就行。
- **<streambuf>**
内含 stream buffer 基类 (`basic_streambuf<>`) 的定义。
- **<istream>**
内含仅支持输入的类别 (`basic_istream<>`) 和同时支持输入输出的类别 (`basic_iostream<>`) 的定义⁵。
- **<ostream>**
内含 output stream (`basic_ostream<>`) 的类别定义。
- **<iostream>**
内含全局性的 stream 对象（例如 `cin` 和 `cout`）的定义。

⁵ 乍见之下，`<istream>` 内含同时支持输入和输出的类别定义，似乎不太合逻辑。不过由于对每一个含入 `<iostream>` 的编译单元而言，`start-up`（运行启动）时会有初始化带来的额外负担（详见下一段），所以才把输入类别和输出类别的定义都放进了 `<istream>`。

大部分头文件主要用于 C++ 标准程序库的内部组织。IOStream 程序库使用者只需含入拥有各个 stream classes 声明式的 <iosfwd>, 并在运用输入或输出功能时分别含入 <istream> 或 <ostream> 即可。除非用到标准 stream 对象, 否则不需要含入 <iostream>。在某些实现版本中, 每一个含入 <iostream> 的编译单元在启动 (start-up) 时都需要执行一段程序代码; 虽说其执行负荷并不高, 但却必须加载相应的执行分页, 这项耗费可能不小。一般来说, 有必要被含入的头文件, 我们才含入它。这里只需含入 <iosfwd>, 只有相应的实现文件 (.cpp) 才需含入包含完整定义的头文件。

对于某些特殊的 stream 特性, 例如参数化的操控器 (manipulators)、file streams、string streams, 需要含入其它头文件 (<iomanip>, <fstream>, <sstream> 和 <strstream>)。介绍这些特性时我再来详细说明它们对应的头文件。

13.3 标准的 Stream 操作符 << 和 >>

C/C++ 的操作符 << 和 >> 分别用于位左移和右移, 然而 basic_istream<> 和 basic_ostream<> 重载了它们, 使之成为标准的 I/O 操作符⁶。

13.3.1 output 操作符 <<

basic_ostream (ostream 和 wostream 当然也算) 将 << 定义为 output 操作符, 对所有基本 (语言内建) 型别均进行重载, 包括 char*, void*, bool。stream 将其 output 操作符定义为: 把第二参数发送到相应的 stream 去。数据按箭头方向发送, 例如:

```
int i = 7;
std::cout << i;    // outputs: 7
float f = 4.5;
std::cout << f;    // outputs: 4.5
```

由于 operator<< 可被重载, 所以第二参数可以是任意型别。如此一来我们自己的数据型别就可以整合进入 I/O 系统。编译器负责调用正确的函数以输出第二参数。当然, 这个“正确的函数”的本分应该是把第二参数转换成字符序列, 然后发送给 stream。

C++ 标准程序库利用相同的机制提供 string (p524)、bitset (p468) 及 complex (p539) 的 output 操作符。下面是运用实例:

⁶ 也有人把 I/O 操作符称为 *inserters* 和 *extractors*。

```
std::string s("hello");

s += ", world";
std::cout << s;           // outputs: hello, world

std::bitset<10> flags(7);
std::cout << flags;       // outputs: 0000000111

std::complex<float> c(3.1,7.4);
std::cout << c;           // outputs: (3.1,7.4)
```

13.12 节, p652 页曾经提到如何为使用者自定义型别撰写 output 操作符。

输出机制的可扩展性, 让使用者自定义型别得以天衣无缝地融入 I/O 系统。比起 C 的 `printf()` I/O 机制, 这是一个巨大的进步: 程序员不再需要指定待打印的型别, 只要针对不同型别进行重载 (overload), 就可以保证编译器会自动推断出正确的打印函数。此一机制并不局限于标准型别, 因此程序员可以利用这一机制完成所有型别 (包括自定义型别) 的操作。

`operator<<` 还可在单一语句中打印多个对象。按规定它会返回第一参数, 也就是 output stream, 这使我们得以下列方式将输出操作串成一串:

```
std::cout << x << " times " << y << " is " << x * y << std::endl;
```

`operator<<` 的计算 (evaluate) 次序是由左到右, 因此以下操作最先执行:

```
std::cout << x
```

请注意, 这个操作符的计算次序和其参数的计算次序并无关联, 纯粹就只是操作符本身的计算顺序 (译注: 请你思考 `cout << i++ << i--;` 的结果, 体会上句话的精义。同时容我提醒, 这个式子在不同的平台上可能有不同的结果)。上述表达式返回第一个操作数 `std::cout`。接下来轮到执行:

```
std::cout << " times "
```

然后依序打印对象 `y`、字符串 `" is "` 和乘积 `x*y`。请注意乘法优先序高于操作符 `<<`, 所以不需要为 `x*y` 加上小圆括号。不过有些操作符的优先序比较低, 例如逻辑操作符, 使用时请注意。上例中如果 `x` 和 `y` 分别是浮点数 2.4 和 5.1, 打印结果为:

```
2.4 times 5.1 is 12.24
```

13.3.2 input 操作符 >>

`basic_istream` (istream 和 wistream 当然也算) 将 `>>` 定义为 input 操作符。和 `basic_ostream` 类似, `basic_istream` 也对基本型别如 `char*`, `void*`, `bool`, 重载了 `operator>>`。Stream input 操作符被定义为: 将读入值储存于第二参数, 数据按箭头方向发送, 如下:

```
int i;
std::cin >> i; // reads an int from standard input and stores it
               in i

float f;
std::cin >> f; // reads a float from standard input and stores it
               in f
```

请注意，第二参数值会改变，因此它必须是个 non-const reference。

和 << 一样，我们也可对任意型别重载 input 操作符，并串连运用它们：

```
float f;
std::complex<double> c;

std::cin >> f >> c;
```

这么做缺省情况下会跳过一开始的空格（如果有的话），但这项功能也可以关闭（p625）。

13.3.3 特殊型别的 I/O

标准 I/O 操作符还定义了 `bool`, `char*` 和 `void*` 型别的输入/输出。此外我们也可以更加扩展，将 << 和 >> 运用在我们自己定义的类型身上。

bool

缺省情况下，布尔值（Boolean）的读入和打印均以数字表示。`false` 对应 0，`true` 对应 1。如果读入值既非 0 亦非 1，就被认为错误，此时会设立 `ios::failbit`，并可能抛出相应的异常（p602）。

我们可以设立 stream 的格式化选项，以字符串形式对布尔值进行 I/O 操作（p617）。这是属于国际化的一个议题。一般采用字符串 `"true"` 和 `"false"`，但特殊的 locale objects 可能使用不同字符串，例如德国用的 German locale object 会使用字符串 `"wahr"` 和 `"falsch"`。更多细节请看第 14 章，特别是 p698。

char 和 wchar_t

经由 >> 读入一个 `char` 或 `wchar_t` 字符时，缺省情况下会跳过起头的空格。如果你想读入所有字符（包括空格），可清除 `skipws` 标志（p625）或利用成员函数 `get()`（p608）。

char*

C 字符串（亦即 `char*`）只读入真正的文字，因此缺省情况下，读入的起头空格均会跳过，一直读到非空格或 *end-of-file* 为止。是否跳过起头空格，其实可由标志 `skipws` 控制（见 13.7.7 节, p625）。

请注意，这意味你可读入任意长的字符串。C 程序的一个常见错误就是许多人误以为字符串最长不得超过 80 个字符，因此如果实际字符串太长，你只得贸然终止输入。于是通常采用以下做法，设置字符串的最大读取长度：

```
char buffer[81];          // 80 characters and '\0'
std::cin >> std::setw(81) >> buffer;
```

关于操控器 `setw()` 和相应的 `stream` 参数，详见 13.7.3 节, p618。

C++ 标准程序库的 `string`（第 11 章）可根据需要扩张，能够容纳下相当长的字符串。以 `string` 代替 `char*` 可以让程序更轻松也更安全。此外 `string` 还提供了一个很方便的函数，可以一行一行读入（见 p493）。所以请尽量避免使用 C 字符串，多使用 `string`。

void*

操作符 `<<` 和 `>>` 为指针的打印和读入提供了一种可能。如果一个型别为 `void*` 的参数被传递至 `output` 操作符，其地址将被打印出来——形式则视实作版本而定。例如下面这段程序代码打印出一个 C 字符串的内容和地址：

```
char* cstring = "hello";
std::cout << "string \"" << cstring << "\" is located at address:"
          << static_cast<void*>(cstring) << std::endl;
```

执行结果可能如下：

```
string "hello" is located at address: 0x10000018
```

我们甚至可以借助 `input` 操作符读入一个地址。不过你得注意，地址往往是临时性的；同一对象的地址在程序每次启动之后可能不同。地址的打印和读取的一个可能应用是：为了“对象识别（object identification）”目的而交换地址，或是为了共享内存。

Stream Buffers (串流缓冲区)

操作符 `>>` 和 `<<` 可以直接用于读取或改写 stream buffer, 这恐怕是运用 C++ I/O streams 来拷贝文件的最快办法, 见 p683。

使用者自定义型别 (user-defined types)

原则上, 为自定义型别扩展 I/O 机制是相当容易的, 不过如果要考虑所有可能的格式和错误条件, 恐怕需要付出更多努力。13.12 节, p652 对于如何针对自定义型别扩展标准 I/O 机制, 有更多讨论。

13.4 Streams 的状态 (state)

Streams 维护着一种状态, 标志 I/O 是否成功, 并且能够指出不成功的原因。

13.4.1 用来表示 Streams 状态的一些常数

Streams 定义了一些型别为 `iostate` 的常数, 用以反映 stream 的状态(表 13.3)。`iostate` 是 `class ios_base` 的一个成员, 其具体型别由实作版本决定, 也就是说 `iostate` 并未被限定是枚举型别、整数型别 (译注: 不仅 `int`) 或 `class bitset` 的某种具体实现)。

表 13.3 `iostate` 型别的常量

常量	意义
<code>goodbit</code>	一切都好; 没有任何其它状态位被设立
<code>eofbit</code>	遇到 <i>end-of-file</i>
<code>failbit</code>	错误; 某个 I/O 操作未成功
<code>badbit</code>	毁灭性错误; 未定义的 (不确定的) 状态

`goodbit` 值被定义为 0, 因此 `goodbit` 的设立意味其它位均被清为 0。或许 `goodbit` 这名字有点儿模糊, 因为它这并不意味有哪个位被设立, 而是指所有位均被清为 0。

`failbit` 和 `badbit` 的区别主要在于后者代表更严重的错误:

- `failbit`: 如果某项操作未能完成, 但 stream 仍大体 OK, 那么就设立这个位。这通常是由于读入格式错误, 例如程序想读入一个整数, 却遇到一个字符。
- `badbit`: 如果 stream 因不明原因而损坏或丢失数据, 就设立这个位。例如将一个 stream 定位 (positioning) 指向“某个文件的起点”的更前方。

注意, `eofbit` 常常和 `failbit` 同时出现, 因为在 *end-of-file* 之后再试图读取数据, 就会检测出 *end-of-file* 状态。读取最后一个字符时, `eofbit` 并未设立, 但再一次试图读取字符时, 就会导致 `eofbit` 和 `failbit` 同时被设立, 因为读取操作也失败了。

有些早期实作版本还支持标志 `hardfail`, 但 C++ *Standard* 已不再支持它。

这些常数并非全局性的, 而只定义于 `class ios_base` 内, 因此我们得加上作用域 (scope) 操作符 (或和某个对象联袂使用)。例如:

```
std::ios_base::eofbit
```

当然, 使用 `ios_base` 派生类别也行。早期实作版本中, 这些常数均定义于 `class ios` 内。由于 `ios` 派生自 `ios_base`, 还可以少打几个字呢, 所以经常被这么使用:

```
std::ios::eofbit
```

这些标志由 `class basic_ios` 维护, 所以可在 `basic_istream` 和 `basic_ostream` 的所有对象中使用。不过 `stream buffers` 并无状态标志。一个 `stream buffer` 可被多个 `stream objects` 共享, 所以上述所有标志都只能反映最后一次操作的 `stream` 状态 (甚至即使在任何操作之前总是先设立 `goodbit`), 因为这些标志可能被某些更早发生的操作设立。

13.4.2 用来处理 Streams 状态的一些成员函数

用来表示 `stream` 当前状态的一些标志, 可由表 13.4 的成员函数加以确认。

表 13.4 用以处理 Streams 状态的各个成员函数

成员函数	意义
<code>good()</code>	若 <code>stream</code> 正常无误, 返回 <code>true</code> (表示 <code>goodbit</code> 设立)
<code>eof()</code>	若遭遇 <i>end-of-file</i> , 返回 <code>true</code> (表示 <code>eofbit</code> 设立)
<code>fail()</code>	若发生错误, 返回 <code>true</code> (表示 <code>failbit</code> 或 <code>badbit</code> 设立)
<code>bad()</code>	若发生毁灭性错误, 返回 <code>true</code> (表示 <code>badbit</code> 设立)
<code>rdstate()</code>	返回当前已设立的所有标志
<code>clear()</code>	清除所有标志
<code>clear(state)</code>	清除所有标志后, 设立 <code>state</code> 标志
<code>setstate(state)</code>	加设 <code>state</code> 标志

表 13.4 的前四个成员函数可确认 `stream` 状态, 返回一个布尔值。注意 `fail()` 返回的是 `failbit` 或 `badbit` 两者之中是否有任一设立, 这么做主要是因为历史因素, 同时也有一个好处: 测试一次即可确认是否有错误发生。

除此之外, 标志状态还可由更一般的成员函数加以确认或修改。如果调用不带参数的 `clear()`, 所有错误标志 (包括 `eofbit`) 均会被清除 (这正是其名称 '*clear*' 的用意):

```
// clear all error flags (including eofbit):
strm.clear();
```

如果调用带有参数的 `clear()`, `stream` 会把状态调整为该参数所代表的状态, 也就是说 `stream` 会设立该标志并清除其它标志。唯一的例外是, 如果没有 `stream buffer` (也就是说当 `rdbuf()==0` 时, 详见 13.10.2 节, p638), `badbit` 始终会被设立。

下面这个例子检查 `failbit` 是否设立。若是, 则清除之:

```
// check whether failbit is set
if (strm.rdstate() & std::ios::failbit) {
    std::cout << "failbit was set" << std::endl;

    // clear only failbit
    strm.clear (strm.rdstate() & ~std::ios::failbit);
}
```

本例使用位操作符 `&` 和 `~`。 `~` 操作符对其参数的所有位逐一求反相, 因此:

```
~ios::failbit
```

会返回一个临时数值, 其中每个位均和 `failbit` 反相。 `&` 操作符会对其操作数的所有位逐一进行 "and" 运算——两对应位均被设立时, 该位才得以保留。因此如果对于 (1) 当前设立之所有标志 (利用 `rdstate()` 取得) 和 (2) 除 `failbit` 外的所有位进行 "and" 运算, 便能清除 `failbit` 并保留其它位。

如果某些标志被 `clear()` 或 `setstate()` 设立 (见 13.4.4 节, p602), `streams` 便有可能抛出异常 (exceptions)。如果对应的标志被设立起来, 那么当那些标志的处置函数结束之际, `stream` 会抛出异常。

注意, 我们必须明白地清除错误位。C 语言可以在 "格式错误" 发生之后仍然读入字符。例如虽然 `scanf()` 未能读入一个整数, 我们仍能读入剩余字符, 因此虽然读入操作失败, `input stream` 的状态依然 OK。但 C++ 不同: 如果设置了 `failbit`, 除非显式予以清除, 否则无法进行下一个操作。

请注意, 被设立的位只是反映过去曾发生的事。如果某次操作后发现某个位被设立了, 我们无法确定究竟是这一次或先前操作导致这个结果。因此如果想通过标志了解错误, 操作前应先设立 `goodbit` (如果尚未设立的话)。此外, 清除标志之后, 各项操作可能会得到不同的结果, 例如, 纵使 `eofbit` 是被某个操作设立的, 并不意味清除 `eofbit` 之后该操作会再次设置 `eofbit`。例如当 "两次调用之间, 被处理的文件体积增长" 时, 就会发生这种情况。

13.4.3 Stream 状态与布尔条件测试

streams 定义了两个可用于布尔表达式 (boolean expressions) 的函数, 如表 13.5。

表 13.5 可用于布尔表达式的 Stream 操作符

成员函数	意义
operator void* ()	stream 是否未出错 (相当于 !fail())
operator ! ()	stream 是否已出错 (相当于 fail())

我们可以运用 operator void*() 在控制结构中简洁测出 stream 的当前状态:

```
// while the standard input stream is OK
while (std::cin) {
    ...
}
```

控制结构中的布尔条件并不是非得直接转换为 bool 不可, 只要能够转换为某个整数型别 (例如 int 或 char) 或指针型别就够了。转换为 void* 常常是为了在同一表达式中读入对象并测试是否成功:

```
if (std::cin >> x) {
    // reading x was successful
    ...
}
```

正如先前的讨论, 表达式:

```
std::cin >> x
```

会返回 cin, 所以读入 x 后, 上述语句变为:

```
if (std::cin) {
    ...
}
```

此时的 cin 被用于条件判断, 所以 cin 会调用 operator void*, 返回 “stream 是否发生错误”。

以上技术的一个典型应用就是以循环读入对象并处理:

```
// as long as obj can be read
while (std::cin >> obj) {
    // process obj (in this case, simply output it)
    std::cout << obj << std::endl;
}
```

这是把古典的 C filter framework 用在 C++ 对象身上。如果 failbit 或 badbit 被设立, 该循环立即终止——出现错误或读到 *end-of-file* (会同时设置 eofbit 和 failbit, 见 p598) 时就有可能发生这种情况。操作符 >> 的缺省行为会跳过起头的空格, 这往往正是我们所希望的。但如果 obj 的型别是 char, 空格就带有意义了, 这时我们可以运用 stream 的成员函数 put() 和 get() (p611), 或是运用更方便的 istreambuf_iterator (见 p667) 实现 I/O filter。

我们可以利用 operator! 进行相反测试, 它会返回“stream 是否已发生错误”, 也就是说如果 failbit 或 badbit 被设立就返回 true。用法如下:

```
if (! std::cin) {  
    // the stream cin is not OK  
    ...  
}
```

就像隐式转换为布尔值一样, 这个操作符常用于“在同一个表达式中读取对象并测试是否成功”:

```
if (! (std::cin >> x)) {  
    // the read failed  
    ...  
}
```

在这里, 表达式:

```
std::cin >> x
```

返回 cin, 然后我们再对它施行 operator!。'!' 之后的表达式必须以小括号括起来, 这是为了运算优先序: 如果没有小括号, operator! 会先被评估。换句话说, 以下表达式:

```
! std::cin >> x
```

相当于:

```
(!std::cin) >> x
```

这可不是我们想要的。

尽管上述操作符用于布尔表达式时很方便, 但是请注意: 两次 ! 运算并不能返回原始对象, 因为:

- cin 是一个 istream 对象;
- !!cin 是一个描述 cin 状态的布尔值。

和 C++ 的其它特性一样, 使用“转换为布尔值”的方式, 会引起编程风格的争论。通常, 使用诸如 fail() 这样的成员函数可以使程序拥有较佳可读性:

```
std::cin >> x;  
if (std::cin.fail()) {  
    ...  
}
```

13.4.4 Stream 的状态和异常

C++ 的异常处理机制用于处理错误 (errors) 和异常 (exceptions)，见 p15。不过那已是 streams 大行其道之后的事了。为保持向下兼容，streams 于缺省情况下并不抛出异常。标准化之后的 streams 允许我们对任何一种状态标志进行定义：此一状态标志被设立时是否引发异常。这可由成员函数 exceptions() 完成，如表 13.6。

表 13.6 Stream 的异常相关成员函数

成员函数	意义
exceptions(flags)	设定“会引发异常”的标志
exceptions()	返回引发异常的标志

调用无参数的 exceptions()，可获得引发目前异常的那个标志。如果它返回 goodbit，表示没有任何异常被抛出——这是为了向下兼容而设计的一种缺省情况。如果调用带有唯一参数的 exceptions()，那么一旦指定的那个标志被设立起来，立刻就会引发相应异常。

下面这个例子要求 stream 对所有标志均抛出异常：

```
// throw exceptions for all "errors"
strm.exceptions (std::ios::eofbit | std::ios::failbit |
                std::ios::badbit);
```

但如果传入 0 或 goodbit，就不会引发异常：

```
// do not generate exceptions
strm.exceptions (std::ios::goodbit);
```

异常的抛出时机是在“程序调用了 clear() 或 setstate() 之后”又设立某些标志之际。如果某个标志已被设立但未被清除，也会抛出异常：

```
// this call throws an exception if failbit is set on entry
strm.exceptions (std::ios::failbit);
...
// throw an exception (even if failbit was already set)
strm.setstate (std::ios::failbit);
```

抛出的异常是个 std::ios_base::failure 对象，该类别派生自 exception (3.3.1 节, p25)：

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

```

public:
    // constructor
    explicit failure (const string& msg);

    // destructor
    virtual ~failure();

    // return information about the exception
    virtual const char* what() const;
};

```

遗憾的是, C++ *Standard* 并未要求该异常对象包含“出错的 stream”或“错误种类”的任何信息。获取错误信息的唯一可移植方式是借助 `what()`。但是请注意, 只有“调用 `what()`”的操作具移植性, 其返回字符串不具移植性。如果还需要额外信息, 程序员就得自己动手了。

这种行为显示, 异常处理多用于处理意外情形。既然叫做“异常处理”而非“错误处理”, 那么诸如“用户输入数据之格式化”等可预见的错误, 都被认为是“正常”而非“异常”, 可以用状态标志做更好的处理。

Stream 的异常主要是在读取“格式化数据 (例如来自一个自动被改写的文件)”方面显身手。不过即使使用异常仍然会有些问题。如果你希望读取数据至 *end-of-file* 为止, 那么你可能只获得错误发生时所产生的异常, *end-of-file* 带来的异常也会接踵而至, 这是因为检测到 *end-of-file* 时会设置 `failbit` (意味没能成功读入一个对象)。要区分究竟发生了 *end-of-file* 或是发生了输入错误, 就得检查 stream 状态。下例对此做了一个示范, 有个函数从 stream 读取浮点数, 一直读到 *end-of-file* 为止, 然后返回浮点数的总和:

```

// io/sum1a.cpp

#include <istream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        using std::ios;
        double value, sum;

        // save current state of exception flags
        ios::iostate oldExceptions = strm.exceptions();

```

```

/* let failbit and badbit throw exceptions
* - NOTE: failbit is also set at end-of-file
*/
strm.exceptions (ios::failbit | ios::badbit);

try {
    /* while stream is OK
    * - read value and add it to sum
    */
    sum = 0;
    while (strm >> value) {
        sum += value;
    }
}
catch (...) {
    /* if exception not caused by end-of-file
    * - restore old state of exception flags
    * - rethrow exception
    */
    if (!strm.eof()) {
        strm.exceptions(oldExceptions);
        // restore exception flags
        throw;      // rethrow
    }
}

// restore old state of exception flags
strm.exceptions (oldExceptions);

// return sum
return sum;
}
}

```

以上函数首先将原本设立的 stream 异常记录于 oldExceptions，以便稍后恢复，然后要求 stream 在某些确定条件下抛出异常。只要 stream 处于 OK 状态，就进入循环读取数值并累加。如果读到 *end-of-file*，stream 不再处于 OK 状态，它会抛出相应异常——即使我们并未定义“当 eofbit 被设立时引发异常”。这是因为，此处乃通过“继续读取数据”的失败才检测出 *end-of-file*，而这么一来，由于读取失败会设立 failbit，于是也引发异常。为了避免在这种情况下抛出异常，程序于区域内 (locally) 捕捉异常之后，先以 eof() 检查 stream 状态，如果返回 false (表示这个异常不是因为 *end-of-file* 而引发) 才将该异常传播出去。

注意，恢复原先的异常标志时（亦即调用 `exceptions()` 时），如果 `stream` 已设立相应标志，那么此时也会抛出异常。因此如果程序在进入上述函数之前，原本就设定会因 `eofbit`, `failbit` 或 `badbit` 而抛出异常，那么这些异常也会被传播给调用者（译注：由于 `rethrow` 之故）。

上述函数可以在 `main` 之中以下列最简单方式被调用：

```
// io/summain.cpp
#include <iostream>
#include <cstdlib>

namespace MyLib {
    double readAndProcessSum (std::istream&);
}

int main()
{
    using namespace std;
    double sum;

    try {
        sum = MyLib::readAndProcessSum(cin);
    }
    catch (const ios::failure& error) {
        cerr << "I/O exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    catch (const exception& error) {
        cerr << "standard exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    catch (...) {
        cerr << "unknown exception" << endl;
        return EXIT_FAILURE;
    }

    // print sum
    cout << "sum: " << sum << endl;
}
```

问题来了，这样做是否值得？我们也可以让 `stream` 不抛出异常，只有在检测到错误时才抛出异常。这么做带来的额外好处是可以使用自定义的错误信息和自定义的错误相关类别：

```
// io/sum2a.cpp

#include <istream>
namespace MyLib {
double readAndProcessSum (std::istream& strm)
{
    double value, sum;

    /* while stream is OK
     * - read value and add it to sum
     */
    sum = 0;
    while (strm >> value) {
        sum += value;
    }

    if (!strm.eof()) {
        throw std::ios::failure
            ("input error in readAndProcessSum()");
        // 译注：上述文字说可使用自定义的错误信息和自定义的错误相关类别，
        //      就是指在这里丢出。
    }

    // return sum
    return sum;
}
}
```

这么做看起来是不是比较简单一些？这么做需要头文件 `<string>`，因为 `class failure` 的构造函数需要一个 `reference to const string` 作为参数。构造 `string` 对象需要其型别定义式，不过如果只是声明，则只需要 `<istream>` 即可。

13.5 标准 I/O 函数

这一节让我们看看可以取代标准 stream 操作符 (<< 和 >>) 的一些成员函数。它们均用于读写无格式 (unformatted) 数据 (和读写格式化数据的 >>, << 操作符不同)。读取数据时它们并不跳过起始空格, 此外它们对异常的处理方式也不同于格式化 I/O 函数: 如果发生异常, 不论源自某个被调用函数, 或因为某个状态标志被设立 (13.4.4 节, p602), badbit 标志均会被设立。如果异常掩码 (exception mask) 中包含 badbit, 就重新抛出该异常。和格式化函数一样, 无格式相关函数会产生一个 sentry (岗哨) 对象 (13.12.4 节, p658)。

这些函数运用定义于 <ios> 中的 streamsize 型别来指定数量:

```
namespace std {
    typedef ... streamsize;
    ...
}
```

型别 streamsize 通常是带正负号的 size_t, 因为它需要表达负数。

13.5.1 输入用的成员函数

接下来的文字解说中, istream 只是一个标记, 表示用于读取操作的一个 stream class, 可能是 istream, wistream 或 template class basic_istream 的其它具体体。char 也是一个标记, 表示相应的字符型别, 例如 char 之于 istream, wchar_t 之于 wistream。其余以斜体表示的型别或数值, 取决于字符型别或“与 stream 关联的”特性类别 (traits class) 的确切定义。

C++ 标准程序库提供了数个读写字符序列的成员函数, 表 13.7 列出其性能比较。

表 13.7 Stream 函数读取字符序列的性能

成员函数	读取, 直到...	字符数	添加 结束符号	返回值
get(s,num)	不包括 new line 或 EOF	最多 num-1	是	istream
get(s,num,t)	不包括 t 或 EOF	最多 num-1	是	istream
getline(s,num)	包括 new line 或 EOF	最多 num-1	是	istream
getline(s,num,t)	包括 t 或 EOF	最多 num-1	是	istream
read(s,num)	EOF	num	否	istream
readsome(s,num)	EOF (<i>end-of-file</i>)	最多 num	否	count


```
int istream::get ()
```

- 读入下一个字符。
- 返回读入的字符，或 *EOF*。
- 通常返回型别是 `traits::int_type`。`traits::eof()` 会返回 *EOF*。对 *istream* 而言，返回型别是 `int`，*EOF* 则为常数 *EOF*。因此对 *istream* 来说，此函数对应于 C 语言的 `getchar()` 或 `getc()`。
- 注意，返回值不一定是字符型别，可以是一种范围更大的数值型别，否则就不能根据相应实值区分 *EOF* 和一般字符。

```
istream& istream::get (char& c)
```

- 把下一个字符设给参数 *c*。
- 返回 *stream*；*stream* 的状态可说明是否读取成功。

```
istream& istream::get (char* str, streamsize count)
```

```
istream& istream::get (char* str, streamsize count, char delim)
```

- 两种形式均可读取多达 `count-1` 个字符，并存入 *str* 所指的字符序列中。
- 第一种形式的读取终止条件是，下一字符是相应字符集中的换行符号，例如 `'\n'` 之于 *istream*，`wchar_t('\n')` 之于 *wistream*（见 p691）。一般情况下可使用 `widen('\n')`（见 p626）。
- 第二种形式的读取终止条件是：下一字符是 *delim*。
- 两种形式均返回 *stream*；*stream* 的状态可说明读入是否成功。
- 不会读入终止符号 *delim*。
- 读入的字符序列以字符串终止符号为结尾。
- 调用者必须保证 *str* 足够存入 `count` 个字符。

```
istream& istream::getline (char* str, streamsize count)
```

```
istream& istream::getline (char* str, streamsize count, char delim)
```

- 两种形式和先前对应的 `get()` 完全一样，除了一点：终止读取时，读入的内容包括换行符号或 *delim*，而非在两者之前就终止。
- 因此当换行符号或 *delim* 位于 `count-1` 个字符内时，它们会被读入，但不储存于 *str* 内。

```
istream& istream::read (char* str, streamsize count)
```

- 读取 `count` 个字符，并存入字符串 *str* 中。
- 返回 *stream*。*stream* 的状态可以说明读取是否成功。
- *str* 内的字符串不会自动以字符串终止符号结束。
- 调用者必须确保 *str* 有足够空间储存 `count` 个字符。
- 读入过程中如果遇到 *end-of-file* 会出错，此时 *failbit* 会被设立（当然 *eofbit* 也有份）。

```
streamsize istream::readsome (char* str, streamsize count)
```

- 可读入多达 *count* 个字符，存入字符串 *str* 中。
- 返回读取的字符个数。
- *str* 内的字符串不会自动以字符串终止符号结束。
- 调用者必须确保 *str* 有足够空间储存 *count* 个字符。
- 与 `read()` 相反，`readsome()` 会读入 *stream buffer* 内的所有有效字符（利用 *buffer* 的成员函数 `in_avail()`）。当我们不希望等待输入时，这很有用，因为输入来自键盘或其它行程（*processes*）。遭遇 *end-of-file* 并不算错，也不会因此设立 `eofbit` 或 `failbit`。

```
streamsize istream::gcount () const
```

- 返回上次“非格式化读取操作”所读入的字符个数。

```
istream& istream::ignore ()
```

```
istream& istream::ignore (streamsize count)
```

```
istream& istream::ignore (streamsize count, int delim)
```

- 所有形式均提取（*extract*）字符并舍弃不用。
- 第一形式忽略（*ignores*）一个字符。
- 第二形式可忽略多达 *count* 个字符。
- 第三形式可忽略多达 *count* 个字符，直到提取并舍弃掉 *delim*。
- 如果 *count* 的值等于 `std::numeric_limits<std::streamsize>::max()`（这是型别 `std::streamsize` 的最大值，见 4.3 节，p59），那么 *delim* 或 *end-of-file* 之前的所有字符均被舍弃。
- 上述形式均返回 *stream*。
- 举例如下：

- 以下操作会弃置当前这一行的剩余部分：

```
cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
```
- 以下操作会弃置 `cin` 的所有剩余内容：

```
cin.ignore(numeric_limits<std::streamsize>::max());
```

```
int istream::peek ()
```

- 返回 *stream* 之内下一个将被读入的字符，但不真的把它读出来。下一次读取结果便是该字符（除非改变了读取位置）。
- 如果不能再读入任何字符，返回 *EOF*。
- *EOF* 即 `traits::eof()` 的返回值；对 *istream* 来说就是常量 `EOF`。

```
istream& istream::unget ()
```

```
istream& istream::putback (char c)
```

- 两者均把上一次读取字符放回 *stream*，使之可被下一次读取（除非改变了读取位置）。

- 两者的区别在于, `putback()` 会检查传入的 `c` 是否确是上一次读取的字符。
- 如果无法放回字符, 或者 `putback()` 发现待放字符不正确, 就设立 `badbit`, 相应的异常可能被抛出 (13.4.4 节, p602)。
- 两函数可放回的最大字符数, 由实作版本决定。C++ *Standard* 保证在两次读取之间允许你调用上述任一函数一次, 并且具有可移植性。

读入 C 字符串时, 本节函数比操作符 `>>` 更安全, 因为必须向本节函数明白传入最大字符数作为参数。尽管操作符 `>>` 也可以限制字符个数 (p618), 但使用者很容易就会忘记那么做。

直接运用 `stream buffer` 常常比使用 `istream` 成员函数更好。由于不需构造 `sentry` 对象 (详见 13.12.4 节, p658), 因而避免了相应的额外开销, 因而使得 `stream buffers` 提供的成员函数在读取单个字符或字符序列时更为高效。13.13 节, p663 详细说明了 `stream buffer` 的接口。另一种方法是使用 `template class istreambuf_iterator`, 它提供一个 `stream buffer` 迭代器接口 (见 13.13.2 节, p665)。

另有两个用以操控读取位置的函数: `tellg()` 和 `seekg()`, 主要和文件连用, 所以我把相关描述延后到 13.9.2 节, p634。

13.5.2 输出用的成员函数

接下来的文字解说中, `ostream` 只是一个标记, 表示用于改写操作的一个 `stream class`, 可能是 `ostream`, `wostream` 或 `template class basic_ostream` 的其它具现体。`char` 也是一个标记, 表示相应的字符型别, 例如 `char` 之于 `ostream`, `wchar_t` 之于 `wostream`。其余以斜体表示的型别或数值, 取决于字符型别或“与 `stream` 关联的”特性类别 (`traits class`) 的确切定义。

```
ostream& ostream::put (char c)
```

- 将参数 `c` 写至 `stream`。
- 返回 `stream`。 `stream` 的状态可说明改写是否成功。

```
ostream& ostream::write (const char* str, streamsize count)
```

- 把字符串 `str` 中的 `count` 个字符写入 `stream`。
- 返回 `stream`。 `stream` 的状态可说明改写操作是否成功。
- 字符串终止符号并不会终止改写操作, 它本身也会被写入 `stream`。
- 调用者必须确保 `str` 的确包含至少 `count` 个字符, 否则可能导致无法预期的行为。

```
ostream& ostream::flush ()
```

- 刷新 (*flushes*) *stream* 的缓冲区，也就是把所有缓冲数据强制写入其所属的设备或 I/O 通道。

另有两个用以改变写入位置的函数：`tellg()` 和 `seekg()`，它们主要与文件连用，所以我把相关描述延后到 13.9.2 节，p634。

和输入函数一样，直接运用 `stream buffer` 或 `template class ostreambuf_iterator` 进行“非格式化改写操作”，将更为合理。因此这些非格式化输出函数少有着力点，不过在多线程 (`multi-threaded`) 环境下它们可能利用 `sentry` (岗哨) 对象来处理锁 (`locks`) 问题，详见 13.14.3 节，p683。

13.5.3 运用实例

下面是以 C++ 完成的古典的 `filter framework`，作法是很单纯地写出所有读取字符：

```
// io/charcat1.cpp

#include <iostream>
using namespace std;

int main()
{
    char c;

    // while it is possible to read a character
    while (cin.get(c)) {
        // print it
        cout.put(c);
    }
}
```

每次调用 `cin.get(c)`，下一字符就被设给了 `c` (它是 `pass by reference`)。 `get()` 返回 `stream`，我们可直接利用它来测试 `cin` 的状态是否正常⁷。

为了精益求精，我们还可以直接操作 `stream buffers`。请看两个实例：p667 运用 `stream buffer` 迭代器进行 I/O，p683 在单一语句中复制所有输入数据。

⁷ 注意，对 `filters` 来说，这比一般的 C 接口更好。C 里头必须使用 `getchar()` 或 `getc()`，其返回值既可能是下一字符，也可能表示到达 *end-of-file* 与否，这导致返回值必须是个 `int`，才能区分 `char` 值和 *end-of-file*。

13.6 操控器 (Manipulators)

我已经在 13.1.5 节, p586 介绍过 stream 操控器 (manipulators), 那些对象如果和标准 I/O 操作符连用, 便可用来修改 stream, 但这并不意味着会额外读取或写入什么。基本操控器定义于 `<istream>` 或 `<ostream>`, 见表 13.8。

表 13.8 定义于 `<istream>` 或 `<ostream>` 中的操控器 (manipulators)

操控器	类别	意义
flush	basic_ostream	刷新 output 缓冲区, 将内容写入输出设备
endl	basic_ostream	向缓冲区插入换行符号并刷新, 将内容写入设备
ends	basic_ostream	向缓冲区插入字符串终止符号
ws	basic_istream	读取时忽略空格

另有一些操控器, 例如改变 I/O 格式等等, 在 p615 的 13.7 节“格式化”另行介绍。

13.6.1 操控器如何运作

操控器的实现借助一种很简单的技巧, 这种技巧不仅让 stream 的操控更为方便, 也是函数重载 (function overloading) 威力的一次绝好示范。作为 I/O 操作符的参数, 操控器其实就是一个被 I/O 操作符调用的函数。例如“针对 ostream 运作”的 output 操作符, 基本上被重载如下⁸:

```
ostream& ostream::operator << ( ostream& (*op)(ostream&))
{
    // call the function passed as parameter with this stream as the argument
    return (*op)(*this);
}
```

参数 `op` 是个函数指针, 或者更准确地说, 这个指针指向一个函数, 以 `ostream` 为参数并返回 `ostream` (假定就返回作为参数的那个 `ostream`)。如果操作符 `<<` 的第二操作数 (右操作数) 是这样一个函数, 就以 `<<` 的第一操作数 (左操作数) 为参数, 调用之。

听起来很复杂, 实际上相对简单, 举个例子就清楚了。用于 `ostream` 身上的 `endl()` 操控器 (其实是个函数) 的主要实现如下:

⁸ 实际实现会稍微复杂些, 因为需要构造一个 `sentry` 对象, 而且实际上这是一个 `function template`。

```
std::ostream& std::endl (std::ostream& strm)
{
    // write newline
    strm.put('\n');

    // flush the output buffer
    strm.flush();

    // return strm to allow chaining
    return strm;
}
```

我们可以采用下列方式，在一个表达式中使用该操控器：

```
std::cout << std::endl
```

这里的 `<<` 分别以 `cout` 和 `endl()` 为操作数，从前述实作法可知，操作符 `<<` 把它本身的调用操作转换为一个以 `stream` 为参数的函数调用：

```
std::endl(std::cout)
```

直接调用上述表达式，其效果和使用操控器相同，而且还有一个优点：不用写出命名空间（`namespace`），这样就可以了：

```
endl(std::cout)
```

因为如果全局空间中没有这个函数，编译器会自动在其参数定义所在的命名空间中搜寻（见 p17。译注：Koenig lookup 法则）。

由于 `stream classes` 全都是“以字符型别为 `template` 参数”的 `template classes`，所以 `endl()` 实际实现如下：

```
template<class charT, class traits>
std::basic_ostream<charT, traits>&
std::endl (std::basic_ostream<charT, traits>& strm)
{
    strm.put(strm.widen('\n'));
    strm.flush();
    return strm;
}
```

成员函数 `widen()` 用来将换行符号转换成当前 `stream` 所用的字符，更多细节请看 13.8 节, p625。

C++ 标准程序库也提供了一些带有参数的操控器，其运作方式视实作版本而定。目前并无任何标准规格用来规范如何实作“带参数的使用者自定义操控器”。

标准操控器均定义于 `<iomanip>`，使用前必须先含入它：

```
#include <iomanip>
```

带有参数的标准操控器均和格式化细节有关，我将在格式化选项（`formatting options`）部分对此再加说明。

13.6.2 使用者自定义操控器

我们可以定义自己的操控器，所需做的不过是写一个像 `endl()` 一样的函数。例如以下函数定义了一个操控器，将 *end-of-line* 之前的所有字符（即当前此行的剩余部分）忽略掉：

```
// io/ignore.hpp

#include <istream>
#include <limits>

template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
ignoreLine (std::basic_istream<charT,traits>& strm)
{
    // skip until end-of-line
    strm.ignore( std::numeric_limits<int>::max(),
                 strm.widen('\n') );

    // return stream for concatenation
    return strm;
}
```

这个操控器简单地把任务交给函数 `ignore()` 来完成，这便会忽略 *end-of-line* 之前的所有字符（`ignore()` 详见 p609）。这个操控器用起来很简单：

```
// ignore the rest of the line
std::cin >> ignoreLine;
```

如果想忽略若干行，就多用几次：

```
// ignore two lines
std::cin >> ignoreLine >> ignoreLine;
```

这样之所以可以有效运作，乃是因为函数 `ignore(max,c)` 会略去 `input stream` 内“字符 `c` 之前的所有字符”（如果 `c` 之前的字符多于 `max` 个，那么就略去 `max` 个字符；如果先遇到 `stream` 结尾，那就全部忽略之）。字符 `c` 也会一并被忽略。

13.7 格式化 (Formatting)

两个概念支配着 I/O 格式：最重要的是格式标志 (format flags)，它们可定义诸如数字精度、填充字符、数字进制等格式。另一个概念是针对特定地域之人民习惯而调整格式 (国际化议题)。本节介绍的是格式标志，13.8 节, p625 及第 14 章整章描述国际化格式问题。

13.7.1 格式标志 (Format Flags)

class ios_base 提供数个成员，用来定义各种 I/O 格式，例如最小字段宽度、浮点数精度、填充字符等等。成员型别 ios::fmtflags 用来储存组态标志 (configuration flags)，定义诸如“正数前是否加正号、布尔值以数字或文字方式打印”等选项。

某些标志甚至形成群组，例如用于整数的八进制、十进制和十六进制相应标志。使用特定的掩码 (masks) 便可轻松处理此类“标志群组”。

表 13.9 访问格式标志的成员函数

成员函数	意义
setf(flags)	添设格式标志 flags，返回所有标志的原本状态
setf(flags, mask)	添设格式标志 flags (配合掩码 mask)，返回所有标志的原本状态
unsetf(flags)	清除 flags
flags()	返回所有已设立的格式标志
flags(flags)	将 flags 设为新的格式标志，返回所有标志的原本状态
copyfmt(stream)	从 stream 中复制所有格式定义

表 13.9 的成员函数可用于处理 stream 的所有格式定义。函数 setf() 和 unsetf() 分别设置和清除一个或多个标志。二元操作符 "or" (也就是 operator) 可将多个标志合并，从而一次操控多个标志。函数 setf() 以第二参数为掩码，清除该掩码所标识的所有标志，然后设置第一参数所代表的标志。单参数版本的 setf() 则稍有不同。下面是个运用实例：

```
// set flags showpos and uppercase
std::cout.setf (std::ios::showpos | std::ios::uppercase);

// set only the flag hex in the group basefield
std::cout.setf (std::ios::hex, std::ios::basefield);
```



```
// clear the flag uppercase
std::cout.unsetf (std::ios::uppercase);
```

使用 `flags()`，我们得以一次操控所有格式标志。调用无参数的 `flags()`，会返回当前格式标志；如果传给 `flags()` 一个参数，便以该参数作为新的格式标志状态，并返回先前状态。因此带有一个参数的 `flags()` 会清除所有标志，并将状态设置如同传入的标志。`flags()` 相当有用，例如可用以储存当前标志状态，便于适当时机再恢复。例如：

```
using std::ios, std::cout;

// save current format flags
ios::fmtflags oldFlags = cout.flags();

// do some changes
cout.setf(ios::showpos | ios::showbase | ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << std::hex << x << std::endl;

// restore saved format flags
cout.flags(oldFlags);
```

`copyfmt()` 可将 `stream` 的所有格式信息复制给另一个，例见 p653。

表 13.10 的操控器也可用于设立和清除格式标志。

表 13.10 用于存取格式标志的两个操控器

操控器 (manipulator)	作用
<code>setiosflags(flags)</code>	将 <code>flags</code> 设为格式标志(调用相应 <code>stream</code> 的 <code>setf(flags)</code>)
<code>resetiosflags(mask)</code>	清除 <code>mask</code> 所标识的一组标志 (调用相应 <code>stream</code> 的 <code>setf(0,mask)</code>)

操控器 `setiosflags()` 或 `resetiosflags()` 让我们得以在一个改写语句 (搭配 `operator<<`) 或读取语句 (搭配 `operator>>`) 中设定或清理标志。欲使用这些操控器，首先必须含入 `<iomanip>`，例如：

```
#include <iostream>
#include <iomanip>

...
std::cout << resetiosflags(std::ios::adjustfield) // clear adjustm. flags
          << setiosflags(std::ios::left);         // left-adjust values
```

某些格式操控工作可由专门的操控器完成，更方便更易读。详细讨论见下一小节。

13.7.2 布尔值 (Boolean Values) 的 I/O 格式

标志 `boolalpha` 定义了布尔值的读写格式：数字表示或文字表示 (表 13.11)。

表 13.11 用于布尔值表示法的标志

标志	意义
<code>boolalpha</code>	若被设置，便以文字表示，否则以数字表示

如果此标志未被设立 (缺省如此)，布尔值便以数字表示：`false` 始终是 0，`true` 始终是 1。如果以数字方式读入布尔值时遭遇了 0 和 1 以外的数，就会出错 (会设立 `stream` 身上的 `failbit`)。

如果此标志被设立，布尔值便以文字表示。读入的字符串需为 `true` 或 `false`，其实际表述法还和 `stream locale object` 有关 (见 p626 和 p698)。标准的 "C" `locale object` 使用字符串 `"true"` 和 `"false"` 来表示布尔值。

为了方便操控这个标志，标准程序库还专门定义了两个操控器 (表 13.12)。

表 13.12 用于布尔值表示法的操控器

操控器	意义
<code>boolalpha</code>	强制使用文字表示法 (设立标志 <code>ios::boolalpha</code>)
<code>noboolalpha</code>	强制使用数字表示法 (清除标志 <code>ios::boolalpha</code>)

例如，下面语句先后以数字表示法和文字表示法打印 `b`：

```
bool b;
...
std::cout << std::noboolalpha << b << " == " << std::boolalpha
          << b << std::endl;
```

13.7.3 字段宽度、充填字符、位置调整

成员函数 `width()` 和 `fill()` 分别用来定义字段宽度和充填字符（表 13.13）。

表 13.13 成员函数，用于字段宽度和充填字符的设定和取得

成员函数	意义
<code>width()</code>	返回当前的字段宽度
<code>width(val)</code>	设立 <code>val</code> 为当前字段宽度，并返回先前的字段宽度
<code>fill()</code>	返回当前的充填字符
<code>fill(c)</code>	定义 <code>c</code> 为当前充填字符，并返回先前的充填字符

在输出操作中使用字段宽度、充填字符和位置调整

对输出而言，`width()` 定义了最小字段，该定义只应用于下一次格式化输出。无参数的 `width()` 会返回当前字段宽度；如果调用时传入一个整数，则可改变字段宽度并返回先前宽度。最小字段宽度的默认值为 0，意味字段可为任意长——任何一个数值被输出后，字段宽度值将恢复为 0（译注：因为稍早才说过，字段宽度的设定只及于下一次格式化输出）。

请注意，字段宽度从来不是被用来“将输出截尾”。因此你无法指定一个“最大字段”（译注：用以“将输出截尾”）。如果你需要类似功能，应该自己写点程序，例如将运算结果写入一个字符串，再按需求决定输出的字符数。

`fill()` 定义了用来充填“格式化表述”和最小字段之间的充填字符。缺省的充填字符是空格符。

有三个标志被用来在字段中对齐数值，如表 13.14 所示。它们和相应的掩码均定义于 `class ios_base` 中。

表 13.14 一些掩码，用来在字段中对齐数值

掩码	标志	意义
<code>adjustfield</code>	<code>left</code>	靠左对齐
	<code>right</code>	靠右对齐
	<code>internal</code>	符号靠左对齐，数值靠右对齐
	<code>None</code>	靠右对齐（缺省）

执行了任何格式化 I/O 操作后，字段宽度就会恢复为默认值；充填字符和位置调整方式不变，除非你明白地对它们做了修改。

表 13.15 显示将这些函数和标志用于各种数值后的效果。充填字符为 '^'。

表 13.16 位置调整实例

位置调整	width()	-42	0.12	"Q"	'Q'
left	6	-42^^^	0.12^^	Q^^^^^	Q^^^^^
Right	6	^^^-42	^^0.12	^^^^Q	^^^^Q
internal	6	-^^^42	^^0.12	^^^^Q	^^^^Q

注意，单一字符的对齐方式在标准化过程中已经有了变化。标准化前，面对单个字符，会忽略其字段宽度，直到下一次多字符格式化输出时才使用。这个 bug 已经修补好，不过对于某些依赖该 bug 行为的程序来说，这样的修补却破坏了它们的生存机会☹。

C++ 标准程序库定义数个操控器，用以处理字段宽度、充填字符和位置调整(表 13.16)。

表 13.16 操控器，用于位置调整

操控器	意义
setw(val)	令 val 为 I/O 字段宽度，相当于 width()。
setfill(c)	将 c 定义为充填字符，相当于 fill()。
left	靠左对齐
right	靠右对齐
internal	符号靠左对齐，数值靠右对齐

操控器 setw() 和 setfill() 需要一个参数，所以使用前必须先含入其头文件 <iomanip>。例如：

```
#include <iostream>
#include <iomanip>
...
std::cout << std::setw(8) << std::setfill('_') << -3.14
          << ' ' << 42 << std::endl;
std::cout << std::setw(8) << "sum: "
          << std::setw(8) << 42 << std::endl;
```

输出结果为：

```
____-3.14 42
____sum: _____42
```

输入时运用字段宽度

读入型别为 `char*` 的字符序列时，我们也可以运用字段宽度来定义读取的最大字符数。如果 `width()` 不是 0，最多读取 `width()-1` 个字符。

由于一般 C-string 在读取时无法成长其空间，所以当以操作符 `>>` 进行读取时，应该总是使用 `width()` 或 `setw()`，例如：

```
char buffer[81];

// read, at most, 80 characters:
cin >> setw(sizeof(buffer)) >> buffer;
```

这最多能读入 80 个字符，虽然 `sizeof(buffer)` 是 81，但字符串终止符号占用一个字符（这是被自动添加上去的）。注意，下面是常见错误：

```
char* s;
cin >> setw(sizeof(s)) >> s;    // RUNTIME ERROR
```

这是因为该指针只有声明，并未实际拥有储存空间。`sizeof(s)` 仅仅是指针大小，而非它所指的储存空间的大小。这是 C-string 的一个典型错误运用。如果使用 `string` 就不会遇到这类麻烦：

```
string buffer;
cin >> buffer;    // OK
```

13.7.4 正记号与大写字符

两个格式标志 `showpos` 和 `uppercase` 用来改变数值的一般表述（表 13.17）：

表 13.17 两个标志，可影响数值的正负号和字母大小写

标志	意义
<code>showpos</code>	在正数前加上正记号
<code>uppercase</code>	使用大写字符

`ios::showpos` 规定必须打印出正数前的正记号。如果没有设置该标志，则只有负数之前会加上负记号。`ios::uppercase` 规定数值中的字母采用大写；此一标志可用于“十六进制格式表述”的整数，以及“科学记号表述”的浮点数。缺省情况下字母为小写并省略正记号。例如以下句子：

```
std::cout << 12345678.9 << std::endl;

std::cout.setf (std::ios::showpos | std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
```

输出结果为：

```
1.23457e+07
+1.23457E+07
```

这两个标志均可使用表 13.18 的操控器加以设立或清除。

表 13.18 一些操控器，用来操控数值符号和字母大小写

操控器	意义
showpos	强制输出正数前的正记号（设置标志 <code>ios::showpos</code> ）
noshowpos	强制省略正数前的正记号（清除标志 <code>ios::showpos</code> ）
uppercase	强制字母大写（设置标志 <code>ios::uppercase</code> ）
nouppercase	强制字母小写（清除标志 <code>ios::uppercase</code> ）

13.7.5 数值进制 (Numeric Base)

有一组（共三个）标志，用来设定整数 I/O 进制。它们和相应的掩码一同定义于 `ios_base`（表 13.19）。

表 13.19 一些标志，用来定义整数值进制

掩码	标志	意义
basefield	oct	以八进制进行读写
	dec	以十进制进行读写（这是缺省情况）
	hex	以十六进制进行读写
	none	以十进制输出：读取时则视起始字符而定

进制被改变后，除非重新设置相关标志，否则会持续应用于后继的整数处理过程。缺省情况下使用十进制。`IOStream` 并不支持二进制，不过可借助 `class bitset` 以二进制方式读写整数值，详见 10.4.1 节，p462。

如果没有设置任何进制标志，或同时设置了多个标志，输出时便采用十进制。

进制标志也会影响输入。如果设置了上述任何一个进制标志，读取的所有数字便按该进制处理。如果没有设置任何进制标志，则由起始字符决定进制：以 `0x` 或 `0X` 起始的是十六进制，以 `0` 起始的是八进制，其余被视为十进制数字。

主要有两种方法可以切换上述标志：

- 1. 清除某个标志，然后设置另一个：
std::cout.unsetf (std::ios::dec);
std::cout.setf (std::ios::hex);
- 2. 设置一个标志，同时自动清理同组的其它标志：
std::cout.setf (std::ios::hex, std::ios::basefield);

此外另有一些操控器，可以简化标志的处理（表 13.20）。

表 13.20 一些操控器，用来定义整数进制

操控器	意义
oct	以八进制进行读写
dec	以十进制进行读写
hex	以十六进制进行读写

例如以下语句以十六进制输出 x 和 y，以十进制输出 z：

```
int x, y, z;  
...  
std::cout << std::hex << x << std::endl;  
std::cout << y << ' ' << std::dec << z << std::endl;
```

另一个标志 showbase 可以 C/C++ 惯例（如下所述）显示数值进制（表 13.21）。

表 13.21 一个标志，用来显示数字进制

标志	意义
showbase	如果设置，就显示出数字进制

如果设置了 ios::showbase，八进制数字将以 0 开头，十六进制以 0x 开头（如果同时还设置了 ios::uppercase，则以 0x 开头）。例如以下语句：

```
std::cout << 127 << ' ' << 255 << std::endl;  
  
std::cout << std::hex << 127 << ' ' << 255 << std::endl;  
  
std::cout.setf(std::ios::showbase);  
std::cout << 127 << ' ' << 255 << std::endl;  
std::cout.setf(std::ios::uppercase);  
std::cout << 127 << ' ' << 255 << std::endl;
```

输出如下:

```
127 255
7f ff
0x7f 0xff
0X7F 0XFF
```

表 13.22 的操控器也可用来操控 `ios::showbase`。

表 13.22 用两个操控器来显示数值进制

操控器	意义
showbase	显示数值进制 (设置标志 <code>ios::showbase</code>)
noshowbase	不显示数值进制 (清除标志 <code>ios::showbase</code>)

13.7.6 浮点数 (Floating-Point) 表示法

有数个 `stream` 标志和 `stream` 成员可用来控制浮点数的输出。表 13.23 的标志可决定输出时采用小数或科学记号。它们和相应的掩码一起定义于 `class ios_base` 内。如果设置了 `ios::fixed`, 浮点数将以小数表示; 如果设置了 `ios::scientific`, 则以科学记号 (指数方式) 表示。

表 13.23 用于浮点数表述法的一些标志

掩码	标志	意义
floatfield	fixed	使用小数记数法
	scientific	使用科学记数法
	none	使用上述两者中最适合者 (这是缺省情况)

成员函数 `precision()` 用来定义精度 (表 13.24)。

表 13.24 用来决定浮点数显示精度的成员函数

成员函数	意义
<code>precision()</code>	返回当前的浮点数精度
<code>precision(val)</code>	令 <code>val</code> 为新的浮点数精度, 并返回原设置值

如果使用科学记数法，precision() 定义出小数位数，余数均四舍五入而非硬生生截断。无参数的 precision() 会返回当前精度；但如果调用时向它传入一个参数，则以传入值为新的精度，并返回原精度。缺省的精度是 6 个十进制位数。

缺省情况下，ios::fixed 和 ios::scientific 均未设置，这时使用的记数法取决于待输出值。所有数值都有意义，但至多只能输出“precision() 个”十进制位数，形式如下：小数点之前有一个前导的 0，然后是所有必要的 0。如果“precision() 个”十进制位数已足够表现，那么就使用十进制表示法，否则使用科学记号表示法。

标志 showpoint 可强制书写小数点并补 0 至足够精度为止（表 13.25）。

表 13.25 以上标志用来强制书写小数点

标志	意义
showpoint	总是书写小数点

表 13.26 以两个具体数值为例，说明标志和精度之间稍显复杂的相依关系。

表 13.26 浮点数格式化实例

		precision() 421.0	0.0123456789
一般情况	2	4.2e+02	0.012
	6	421	0.0123457
带有 showpoint	2	4.2e+02	0.012
	6	421.000	0.0123457
fixed	2	421.00	0.01
	6	421.000000	0.012346
scientific	2	4.21e+02	1.23e-02
	6	4.210000e+02	1.234568e-02

对整数而言，ios::showpos 可用来写出正号，ios::uppercase 可用来指定科学记数法中的大写 E 或小写 e。

表 13.27 的操控器可用来改变标志 ios::showpoint、记数法和精度。

举个实例，以下语句：

```
std::cout << std::scientific << std::showpoint
           << std::setprecision(8)
           << 0.123456789 << std::endl;
```

会产生以下输出：

```
1.23456789e-01
```

表 13.27 用于浮点数输出形式的一些操控器

操控器	意义
showpoint	总是输出小数点 (设置标志 <code>ios::showpoint</code>)
noshowpoint	不需要小数点 (清除标志 <code>ios::showpoint</code>)
setprecision(val)	以 <code>val</code> 为新的精度
fixed	使用小数记数法
scientific	使用科学记数法

`setprecision()` 是带有参数的操控器, 所以使用前必须先含入 `<iomanip>`。

13.7.7 一般性的格式定义

剩下两个格式标志还没有介绍: `skipws` 和 `unitbuf` (表 13.28)。

表 13.28 剩余的格式标志

标志	意义
skipws	调用 <code>>></code> 读取数值时, 自动跳过起始空格
nounitbuf	每次输出后, 清空 <code>output</code> 缓冲区

缺省情况是设置 `ios::skipws`, 意味缺省情况下读取数值时会跳过起始空格。这通常很有用, 因为数字间的空格是不需要读取的。不过这也意味我们不能以 `>>` 读取空格字符, 因为它们都被跳过了。

`ios::unitbuf` 用来控制 `output` 缓冲区。如果 `ios::unitbuf` 被设立, 就不使用缓冲装置, 每次写出后均清空 (*flush*) 缓冲区。缺省情况下并未设立此一标志, 不过 `cerr` 和 `wcerr` 已预先设置它。

上述两个标志也可由表 13.29 的操控器控制。

13.8 国际化 (Internationalization)

我们可以让 I/O 格式更符合各国家、地域、人民的习性。为此, `class ios_base` 定义了数个成员函数, 见表 13.30。

每个 `stream` 都采用某个相关的 `locale` 对象。缺省情况下 `stream` 构造会拷贝全局性的 `locale` 对象作为自己的 `locale` 对象。`locale` 对象之中定义有数字格式化细节, 例如以哪一个字符作为小数点、以什么文字来表示布尔值 (`Boolean`) 等等。

表 13.29 对应表 13.28 格式标志的操控器

操控器	意义
skipws	调用 >> 时, 跳过起始空格 (设置标志 ios::skipws)
noskipws	调用 >> 时, 不跳过起始空格 (清除标志 ios::skipws)
unitbuf	每次写出后, 清空 output 缓冲区 (设置标志 ios::unitbuf)
nounitbuf	每次写出后, 不清空 output 缓冲区 (清除标志 ios::unitbuf)

表 13.30 用于国际化的成员函数

成员函数	意义
imbue(loc)	设置 locale 对象
getloc()	返回当前的 locale 对象

和 C 的本土化 (localization) 设施相比, 现在我们可为每个 stream 单独分配一个 locale 对象。这样的功能很有用, 例如我们可以采用美国格式读入一个浮点数, 再以德国格式写出 (德文的小数点是个逗号)。14.2.1 节, p694 提供了一个例子, 并讨论相关细节。

某些字符, 主要是一些特殊字符, 常常为相应之 stream 字符集所需要。为此, stream 提供了数个转换函数, 如表 13.31。

表 13.31 用于字符国际化的两个 stream 函数

成员函数	意义
widen(c)	把 char 型别的字符 c 转换为 stream 字符集中的字符
narrow(c,def)	把 stream 字符集中的字符 c 转换为一个 char 字符 (如果无对应的 char 字符, 则返回 def)

例如, 要得到 stream strm 字符集中的换行符号, 可以使用如下函数:

```
strm.widen('\n')
```

关于 locale 和国际化的更多细节, 请看第 14 章。

13.9 文件存取 (File Access)

`stream` 可用来存取文件。C++ 标准程序库提供了四个 `template classes`，并预先定义了四个标准特化版本：

1. `template class basic_ifstream<>` 及其特化版本 `ifstream` 和 `wifstream`，用来读取文件（是一种 "input file stream"）。
2. `template class basic_ofstream<>` 及其特化版本 `ofstream` 和 `wofstream`，用来将数据写入文件（是一种 "output file stream"）。
3. `template class basic_fstream<>` 及其特化版本 `fstream` 和 `wfstream`，用于读写文件。
4. `template class basic_filebuf<>` 及其特化版本 `filebuf` 和 `wfilebuf`，被其它 `file stream classes` 用来进行实际的字符读写工作。

这些 `classes` 和 `stream base classes` 的关系如图 13.2。

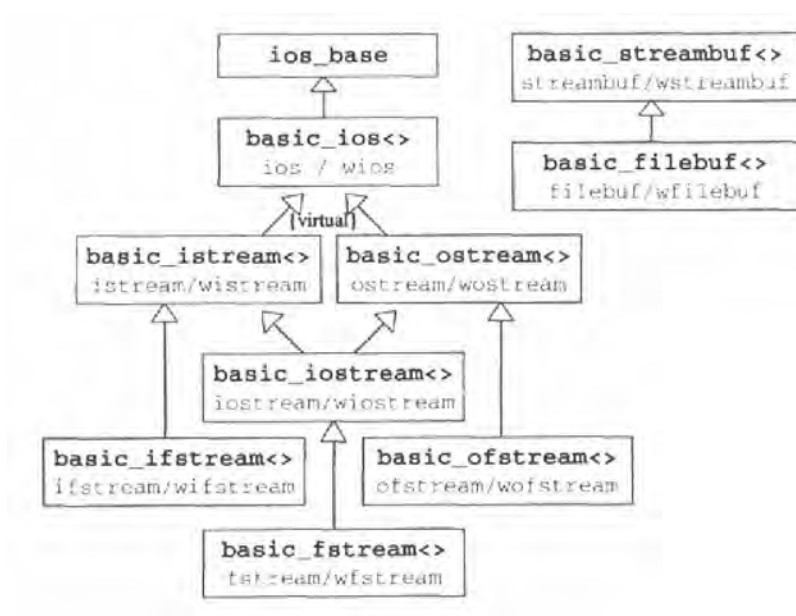


图 13.2 file stream classes 阶层体系

这些 classes 都被声明于头文件 `<fstream>`:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
        class basic_ifstream;
    typedef basic_ifstream<char> ifstream;
    typedef basic_ifstream<wchar_t> wifstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_ofstream;
    typedef basic_ofstream<char> ofstream;
    typedef basic_ofstream<wchar_t> wofstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_fstream;
    typedef basic_fstream<char> fstream;
    typedef basic_fstream<wchar_t> wfstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_filebuf;
    typedef basic_filebuf<char> filebuf;
    typedef basic_filebuf<wchar_t> wfilebuf;
}
```

和 C 的文件存取机制相比, C++ file stream classes 的最大好处就是文件的自动管理。文件可在构造时自动打开,析构时自动关闭。这当然是因为定义了适当的构造函数和析构函数。

注意,对于既可读取亦可改写的 stream,我们不能在读写操作之间任意转换其读写属性⁹!一旦开始读或写,如果一定需要转换,怎么办呢?必须进行一个 seek 操作,到达当前位置,再转换读写属性。唯一的例外是,如果已经读到 *end-of-file*,可立即接着写入字符。如果违背上述规则,可能导致各式各样稀奇古怪的结果。

⁹ 这一限制始自 C 语言。不过很可能 C++ 标准程序库的实作手法决定运用这一限制。
(译注:作者想表达的意思可能是:任意转换读写操作在技术上其实是可行的)

如果 `file stream` 对象以某个 `C-string` (`char*` 型别) 为构造函数参数, 便会自动打开该字符串所代表的文件, 用于读和 (或) 写。成功与否会反映在 `stream` 状态中。因此我们应该在构造 `stream` 之后检查其状态。

以下程序会打开文件 `charset.out`, 写入当前字符集 (数值 32~255 之间的所有字符) :

```
// io/charset.cpp

#include <string>           // for strings
#include <iostream>        // for I/O
#include <fstream>         // for file I/O
#include <iomanip>          // for setw()
#include <cstdlib>          // for exit()
using namespace std;

// forward declarations
void writeCharsetToFile (const string& filename);
void outputFile (const string& filename);

int main ()
{
    writeCharsetToFile("charset.out");
    outputFile("charset.out");
}

void writeCharsetToFile (const string& filename)
{
    // open output file
    ofstream file(filename.c_str());

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open output file \"" << filename << "\""
              << endl;
        exit(EXIT_FAILURE);
    }

    // write character set
    for (int i=32; i<256; i++) {
        file << "value: " << setw(3) << i << " "
              << "char: " << static_cast<char>(i) << endl;
    }

    // closes file automatically
}
```

```
void outputFile (const string& filename)
{
    // open input file
    ifstream file(filename.c_str());

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open input file \"" << filename << "\"\n"
              << endl;
        exit(EXIT_FAILURE);
    }

    // copy file contents to cout
    char c;
    while (file.get(c)) {
        cout.put(c);
    }

    // closes file automatically
}
```

在 writeCharsetToFile() 函数中, class ofstream 的构造函数把接获的字符串当做文件名称, 打开它:

```
std::ofstream file(filename.c_str());
```

由于接获的文件名称是个 string 对象, 所以先以 c_str() 转换为 const char* (细节见 p484)。file stream classes 都不以 string 作为构造函数的参数型别, 这真令人遗憾。接下来确定 stream 是否处于良好状态:

```
if (! file) {
    ...
}
```

如果先前未能成功打开 stream, 上述测试就会失败。测试成功后便以一个循环打印出 32~255 的数值及相应字符。

在 `outputFile()` 函数中, `class ifstream` 的构造函数将 `input` 文件打开, 然后将该文件内容一个字符一个字符地输出到 `cout`。

上述两个函数结束时, 其内的 `stream class object` 会离开生存范围 (`scope`), 被自动调用的 `ifstream` 析构函数和 `ofstream` 析构函数会负责关闭它们所开启的文件。

如果文件有可能在它被产生的范围 (`scope`) 之外被使用, 我们应该从 `heap` 分配该文件对象, 并且在不需要时删除之:

```
std::ofstream* filePtr = new std::ofstream("xyz");
...
delete filePtr;
```

这时候就应当使用某种 `smart pointer class`, 例如 `CountedPtr` (6.8 节, p222) 或 `auto_ptr` (4.2 节, p38)。

除了将文件中的字符逐一加以复制, 我们还可以使用单一语句输出所有内容: 将 `file stream` 的缓冲区指针当做参数, 传给 `operator<<` (详见 p683):

```
// copy file contents to cout
std::cout << file.rdbuf();
```

13.9.1 文件标志 (File Flags)

为了准确控制文件处理模式, `class ios_base` 定义了一组标志, 其类型都是 `openmode`, 这是类似 `fmode` 的一种位掩码型别 (`bit mask type`), 如表 13.32:

表 13.32 用于打开文件的标志

标志	意义
<code>in</code>	打开, 用于读取 (这是 <code>ifstream</code> 的缺省模式)
<code>out</code>	打开, 用于改写 (这是 <code>ofstream</code> 的缺省模式)
<code>app</code>	写入时始终添加于尾端
<code>ate</code>	打开文件之后令读写位置移至文件尾端 (标志名称是 "at end" 之意)
<code>trunc</code>	将先前的文件内容移除 (<i>remove</i>)
<code>binary</code>	不要替换特殊字符

`binary` 使得 `stream` 能够封锁特殊字符或字符序列 (例如 *end-of-line* 或 *end-of-file*) 的转换。某些操作系统 (例如 `MS-DOS` 或 `OS/2`) 的文字文件, 每一行结束时以两个字符 (`CR` 和 `LF`) 表示。正常模式下 (未设置 `binary`) 将以上两个字符替换 `newline`(换行) 字符, 反之亦然。如果处于二进制模式 (设置了 `binary`), 就不会进行这样的转换。

如果文件内容是二进制数据而非字符序列，你应该设置 `binary`。例如复制文件，将源文件字符逐一读入，不作任何修改地写入目标文件。但如果将文件当做文本来处理，则不应设置 `binary`，因为此时需要特别处理换行符号，改为两个字符。

某些实作版本还提供额外的标志，例如 `nocreate`（意指“打开文件时文件必须存在”）和 `noreplace`（意指“打开文件时文件必须不存在”）。不过这些标志都不是标准规格的一部分，也不具可移植性。

这些标志可用操作符 `|` 组合起来，最终结果可作为构造函数的第二参数（此参数可有可无）。例如以下语句以添加（`appending`）方式打开文件：

```
std::ofstream file("xyz.out", std::ios::out | std::ios::app);
```

表 13.33 展示“C++ 标志组合”和“C 的文件开启函数 `fopen()` 所使用之接口字符串”间的关联。标志 `binary` 和 `ate` 的组合没有列出。设置 `binary` 相当于在接口字符串后加一个 `b`，设置 `ate` 则相当于打开文件后立即跳至文件尾端。其它未列入表 13.33 的组合，例如 `trunc|app`，是不允许存在的。

表 13.33 C++ 文件开启模式的意义

ios_base 标志	意义	C 模式
<code>in</code>	读取（文件必须存在）	<code>"r"</code>
<code>out</code>	清空而后改写（有必要才产生）	<code>"w"</code>
<code>out trunc</code>	清空而后改写（有必要才产生）	<code>"w"</code>
<code>out app</code>	添加（有必要才产生）	<code>"a"</code>
<code>in out</code>	读和写；最初位置在起始点（文件必须存在）	<code>"r+"</code>
<code>in out trunc</code>	先清空，再读写（有必要才产生）	<code>"w+"</code>

究竟文件是为 `read` 而开，或是为了 `write` 而开，这和相应的 `stream object class` 无关——那个 `class` 只是在第二参数缺席的情况下定义缺省的开启模式。这意味只有被 `ifstream` 或 `ofstream` 使用的文件，才能用于 `read` 或 `write`。开启模式被传递给相应的 `stream buffer class`，后者才是真正打开文件的人。然而，可能运行于对象身上的操作，则是由 `stream class` 决定。

`file stream` 所拥有的文件，也可以被显式地（`explicitly`）开启或关闭。为此 C++ *Standard* 定义了三个成员函数（表 13.34）。

这些函数之所以有用，大部份是因为 `file stream` 产生时未被初始化。下面例子展示它们的用法：程序首先打开所有文件（文件名由参数指定），然后输出内容（相当于 UNIX 系统中的 `cat` 程序）。

表 13.34 用来打开或关闭文件的一些成员函数

成员函数	意义
<code>open(name)</code>	以缺省模式打开 <code>file stream</code>
<code>open(name, flags)</code>	以 <code>flags</code> 模式打开 <code>file stream</code>
<code>close()</code>	关闭 <code>streams file</code>
<code>is_open()</code>	判断文件是否被打开

```
// io/cat1.cpp
// header files for file I/O
#include <fstream>
#include <iostream>
using namespace std;

/* for all file names passed as command-line arguments
 * - open, print contents, and close file
 */
int main (int argc, char* argv[])
{
    ifstream file; // 译注: 注意, 这个 file stream 稍后将被许多文件共享。

    // for all command-line arguments
    for (int i=1; i<argc; ++i) {

        // open file
        file.open(argv[i]);

        // write file contents to cout
        char c;
        while (file.get(c)) {
            cout.put(c);
        }

        // clear eofbit and failbit set due to end-of-file
        file.clear();

        // close file
        file.close();
    }
}
```

注意, 处理过文件之后, 必须调用 `clear()` 以清除当时被设于文件尾端的状态标志。这是必要的, 因为这个 `stream` 对象被多个文件共享。`open()` 并不会清除任何状态标志, 因此如果某个 `stream` 未处于良好状态, 在关闭并重新打开之后, 你还是必须调用 `clear()` 以取得一个良好状态。即使你透过它开启另一个文件, 情况也一样。

你可以不采用字符逐一处理方式，改以单独一行打印整个内容，作法是将一个指向 file stream 缓冲区的指针当做参数传给 operator<<:

```
// write file contents to cout
std::cout << file.rdbuf();
```

细节请看 p683。

13.9.2 随机存取

表 13.35 列出的成员函数，用来为 C++ streams 确定读写位置。

表 13.35 用于确定 stream 读写位置的成员函数

类别	成员函数	意义
basic_istream<>	tellg()	返回读取位置
	seekg(pos)	设置“绝对读取位置”
	seekg(offset, rpos)	设置“相对读取位置”
basic_ostream<>	tellp()	返回写入位置
	seekp(pos)	设置“绝对写入位置”
	seekp(offset, rpos)	设置“相对写入位置”

这些函数以特殊字尾区隔读或写（g 表示 get, p 表示 put）。用于读取的位置函数定义于 basic_istream，用于改写的位置函数定义于 basic_ostream。并不是所有 stream classes 都支持读写定位，例如 streams cin, cout 和 cerr 就不支持读写定位。File stream 的定位函数之所以定义于基类，因为它们总是会收到一个 references，指向型别为 istream 或 ostream 的对象。（译注：我对这句话的含义没有十足把握，其中所说的 reference 也可能是指 this 指针。原文如下：The positioning of files is defined in the base classes because, usually, references to objects of type istream and ostream are pass around.）

函数 seekg() 和 seekp() 可以接受一个绝对位置或一个相对位置。为了运用绝对位置，你必须采取 tellg() 和 tellp()。它们都会返回一个绝对位置，型别为 pos_type。这个值并不是整数或一个形同索引的字符位置。这是因为逻辑位置和实际位置可能不同。例如在 MS-DOS 文本文件中，newline 字符是以两个字符表现，而逻辑观点上它只是一个字符。如果文件以某种多字节（multibytes）表述法来表现字符，情况会变得更糟。

pos_type 的精确定义有点复杂：C++ 标准程序库为文件位置定义了一个全局性的 template class fpos<>。它被用来定义出 streampos 型别（针对 char streams）和 wstreampos 型别（针对 wchar_t streams），这些型别被用来定义相应之字符特性（character traits，见 14.1.2 节，p689）中的 pos_type。特性类别（traits class）

中的 `pos_type` 成员被用来定义相应的 `stream classes` 的 `pos_type`。因此，你也可以使用 `streampos` 作为 `stream` 的位置型别。使用 `long` 或 `unsigned long` 则是错误的，因为 `streampos` 无论如何不是一个整数型别¹⁰。下面是个运用实例：

```
// save current file position
std::ios::pos_type pos = file.tellg();
...
// seek to file position saved in pos
file.seekg(pos);
```

你可以将下面的写法：

```
std::ios::pos_type pos;
```

改为这样：

```
std::streampos pos;
```

至于相对位置，偏移值 (`offset`) 可以和三个位置相关，如表 13.36 所示。这些常数被定义于 `class ios_base` 中，型别为 `seekdir`。

表 13.36 用于相对位置的常数

常数	意义
<code>beg</code>	位置是相对于开头 (<code>beginning</code>) 而言
<code>cur</code>	位置是相对于当前位置 (<code>current</code>) 而言
<code>end</code>	位置是相对于结尾 (<code>end</code>) 而言

偏移值属于 `off_type` 型别，那是 `streamoff` 的一个间接定义。和 `pos_type` 类似，`streamoff` 被用来定义 `stream classes` 和字符特性 (`traits`) 的 `off_type` (后者见 p689)。由于 `streamoff` 是一个带正负号的整数型别，所以你可以使用一个整数当做 `stream` 的偏移值。例如：

```
// seek to the beginning of the file
file.seekg (0, std::ios::beg);
...
// seek 20 character forward
file.seekg (20, std::ios::cur);
...
// seek 10 characters before the end
file.seekg (-10, std::ios::end);
```

不论哪一种情况，务请注意，读写位置只在文件长度中有效。如果某个位置在文件起头之前，或文件结尾之后，将导致未定义的行为。

¹⁰ 早先时候，用于 `stream` 位置的 `streampos` 的确只是一个 `unsigned long`。

下面的例子展示 `seekg()` 的用法，以一个函数将文件内容写出两次：

```
// io/cat2.cpp

// header files for file I/O
#include <iostream>
#include <fstream>

void printFileTwice (const char* filename)
{
    // open file
    std::ifstream file(filename);

    // print contents the first time
    std::cout << file.rdbuf();

    // seek to the beginning
    file.seekg(0);

    // print contents the second time
    std::cout << file.rdbuf();
}

int main (int argc, char* argv[])
{
    // print all files passed as a command-line argument twice
    for (int i=1; i<argc; ++i) {
        printFileTwice(argv[i]);
    }
}
```

注意，`file.rdbuf()` 被用来打印文件内容（见 p683）。此时是直接操作 `stream` 缓冲区，那并不会改变 `stream` 状态。如果透过 `stream` 接口函数（例如透过 `getline()`，见 13.5.1 节，p607）打印 `file` 内容，必须先调用 `clear()` 清除 `file` 的状态——在它被任何方式处理之前（包括改变读取位置），因为这些函数到达文件尾端时会设立 `ios::eofbit` 和 `ios::failbit`。

处理读写位置的函数有许多个，但是对标准 `streams` 而言，同一个 `stream` 缓冲区内系以同一个位置指出（维护）读取位置和改写位置。如果有多个 `streams` 共同使用同一个 `stream` 缓冲区，这一点便很值得注意。13.10.2 节，p638 对此有较多的解释。

13.9.3 使用文件描述符 (File Descriptors)

某些实现版本提供这种可能性：将一个 stream 附着到一个已开启的 I/O 通道。为了这么做，你必须以一个文件描述符 (*file descriptor*) 将 file stream 初始化。

文件描述符是个整数，用来辨识某个开启的 I/O 通道。在 UNIX 族系的操作系统中，文件描述符被用于低层界面：操作系统提供的 I/O 函数。有三个文件描述符是预先定义好的：

1. 0 代表标准输入通道 (standard input channel)
2. 1 代表标准输出通道 (standard output channel)
3. 2 代表标准错误信息通道 (standard error channel)

这些通道可能被连接至文件、控制台 (console)、其它行程 (processes)、或其它 I/O 设施。

很遗憾，C++ 标准程序库并未提供“运用文件描述符将一个 stream 附着到某个 I/O 通道”的可能性。这是因为这种语言被认为应该独立于任何操作系统。但事实上可能性依然存在，唯一的缺点是它们不具移植性。这里所缺乏的是一个存在于操作系统标准接口（例如 POSIX 或 X/OPEN）上的相应规格。这样的标准至今还没有一点儿谱。

不过，倒是有可能以一个文件描述符将某个 stream 初始化。见 13.13.3 节, p672, 该处描述并实作了一个可能的解答。

13.10 连接 Input Streams 和 Output Streams

常常会需要连接两个 streams。例如你可能想在读取数据前确保屏幕上已经打印出提示文字。或者你可能希望对同一个 stream 读取和改写——这种情况主要发生在 file stream 身上。有时候也可能需要以不同的格式处理同一个 stream。本节讨论所有这些技术。

13.10.1 以 tie() 完成“松耦合” (Loose Coupling)

你可以把一个 stream 连接到一个 output stream 身上。这意味两者的缓冲区是同步的，其具体作法是：output stream 将在另一个 stream 执行输入或输出操作前先清空自己的缓冲区。也就是说对 output stream 而言，其 flush() 函数会被调用。表 13.37 列出 basic_ios 定义的数个成员函数，它们用来将某个 stream 连接到另一个 stream 身上。

无参数的那个 tie(), 会返回一个指针，指向当前所连接的 output stream。如果要把某个 output stream A 连接到某个 stream B 身上，那就必须把指向 A 的一个指

针当做参数传给单一参数的 `tie()`。之所以拿指针作为参数，是因为这样可以把 `0` 或 `NULL` 也当做参数，表示“无连接”，用以解除任何被连接的 `output stream`。如果未曾连接 `output stream`, `tie()` 会返回 `0`。每个 `stream` 只能连接一个 `output stream`，但你可以把一个 `output stream` 连接到多个 `streams` 身上。

表 13.37 将一个 `stream` 连接到另一个 `stream` 身上

成员函数	意义
<code>tie()</code>	返回一个指针，指向一个 <code>output stream</code> ，该 <code>output stream</code> 将被连接到当前 <code>stream</code> 身上
<code>tie(ostream* strm)</code>	将 <code>strm</code> 所指的 <code>output stream</code> 连接到当前 <code>stream</code> 身上，并返回一个指针指向先前所连接的 <code>output stream</code> （如有的话）

缺省情况下，标准 `input` 装置以下列方式连接到标准 `output` 装置上：

```
// predefined connections:
std::cin.tie (&std::cout);
std::wcin.tie (&std::wcout);
```

这样就保证了在真正请求输入之前，一定会先清空 `output` 缓冲区。例如以下语句：

```
std::cout << "Please enter x: ";
std::cin >> x;
```

程序读取 `x` 之前会先隐式（`implicitly`）对 `cout` 调用函数 `flush()`。

如果想要删除两个 `stream` 间的连接，可传递 `0` 或 `NULL` 给 `tie()`。例如：

```
// decouple cin from any output stream
std::cin.tie (static_cast<std::ostream*>(0));
```

这么做或许可以提高程序性能，因为或许避免了非必要的 `stream` 缓冲区清空操作。关于 `stream` 性能的讨论，请看 p683 第 3 点。

你也可以将一个 `output stream` 连接到另一个 `output stream` 上。例如以下语句就保证在对着 `error stream` 写东西之前，先清空正常的 `output` 缓冲区。

```
// tying cout to cerr
cerr.tie (&cout);
```

13.10.2 以 `stream` 缓冲区完成“紧耦合”（`Tight Coupling`）

透过函数 `rdbuf()`，可以使不同的 `streams` 共享同一个缓冲区，从而实作 `streams` 的紧耦合。表 13.38 所列函数适用于不同目的，分别在本节和接下来的小节中讨论。

`rdbuf()` 允许数个 `stream` 对象从同一个 `input` 通道读取信息，或者对同一个 `output` 通道写入信息，而不必困扰于 `I/O` 次序。由于 `I/O` 操作被施以缓冲措施，所以同时使用多个 `stream` 缓冲区是很麻烦的。因为，对着同一个 `I/O` 通道使用不同的

streams, 而这些 streams 的缓冲区又各不相同, 意味 I/O 得传递给其它 I/O。basic_istream 和 basic_ostream 各有构造函数接受一个 stream 缓冲区作为参数, 以此将 stream 初始化。下面是运用实例:

表 13.38 Stream 缓冲区的存取

成员函数	意义
rddbuf()	返回一个指针, 指向 stream 缓冲区
rddbuf(streambuf*)	将参数所指的 stream 缓冲区安装 (installed) 到当前 stream 身上, 并返回一个指针, 指向先前安装的 stream 缓冲区

```
// io/rdbuf1.cpp

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

int main()
{
    // stream for hexadecimal standard output
    ostream hexout(cout.rdbuf());
    hexout.setf (ios::hex, ios::basefield);
    hexout.setf (ios::showbase);

    // switch between decimal and hexadecimal output
    hexout << "hexout: " << 177 << " "; // 输出 hexout: 0xb1
    cout << "cout: " << 177 << " ";    // 输出 cout: 177
    hexout << "hexout: " << -49 << " "; // 输出 hexout: 0xfffffcf
    cout << "cout: " << -49 << " ";    // 输出 cout: -49
    hexout << endl;
}
```

注意, basic_istream 和 basic_ostream 的析构函数并不删除相应的 stream 缓冲区 (毕竟该缓冲区并非由这些 classes 打开)。因此你可以传递一个指向 stream 缓冲区的指针, 而不必使用 stream reference。

```
// io/rdbuf2.cpp

#include <iostream>
#include <fstream>

void hexMultiplicationTable (std::streambuf* buffer, int num)
{
```



```

        std::ostream hexout(buffer);
        hexout << std::hex << std::showbase;
        for (int i=1; i<=num; ++i) {
            for (int j=1; j<=10; ++j) {
                hexout << i*j << ' ';
            }
            hexout << std::endl;
        }
    } // does NOT close buffer

int main()
{
    using namespace std;
    int num = 5;

    cout << "We print " << num
          << " lines hexadecimal" << endl;

    hexMultiplicationTable(cout.rdbuf(), num);

    cout << "That was the output of " << num
          << " hexadecimal lines " << endl;
}

```

这种方法的优点在于格式被修改后不必恢复其原先状态，因为格式乃是针对 `stream` 对象而不是针对 `stream` 缓冲区。上述程序输出如下：

```

We print 5 lines hexadecimal
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0x2 0x4 0x6 0x8 0xa 0xc 0xe 0x10 0x12 0x14
0x3 0x6 0x9 0xc 0xf 0x12 0x15 0x18 0x1b 0x1e
0x4 0x8 0xc 0x10 0x14 0x18 0x1c 0x20 0x24 0x28
0x5 0xa 0xf 0x14 0x19 0x1e 0x23 0x28 0x2d 0x32
That was the output of 5 hexadecimal lines

```

但是此法也有缺点：`stream` 对象的构造和析构会有更多额外开销，不仅仅是设置和恢复一些格式标志而已。同时也请注意，`stream` 对象析构时并不清空缓冲区。要确保 `output` 缓冲区被清空，就必须手工进行。

只有 `basic_istream` 和 `basic_ostream` 不销毁 `stream` 缓冲区。其它 `streams` 都会销毁它们最初分配的 `stream` 缓冲区，但它们不会销毁以 `rdbuf()` 设置的缓冲区（详见下一小节）。

13.10.3 将标准 Streams 重新定向 (Redirecting)

`Iostream` 程序库的早期实现版本中, 全局性的 streams 如 `cin`, `cout`, `cerr` 和 `clog` 都是隶属于 `istream_withassign` 和 `ostream_withassign` 的全局对象。因此我们才有可能把这些 streams 赋值给其它 streams, 用以将 streams 重新定向。C++ 标准程序库已经将这种可能性剔除, 但仍然保留重定向的可能性, 并扩及所有 streams。现在, 只要透过“设置 stream 缓冲区”就可以重定向某个 stream。

“设置 stream 缓冲区”意味 I/O stream 的重定向可由程控, 不必借助操作系统。例如以下语句作了些设置, 使得写入 `cout` 的信息不被送到标准 output 通道, 而是被送到 `cout.txt` 去:

```
std::ofstream file ("cout.txt");
std::cout.rdbuf (file.rdbuf());
```

函数 `copyfmt()` 可用来将某个 stream 的所有格式信息赋值给另一个 stream 对象:

```
std::ofstream file ("cout.txt");
file.copyfmt (std::cout);
std::cout.rdbuf (file.rdbuf());
```

小心! 上述的 `file` 是局部对象, 将在上述程序区段结束时被销毁, 相应的 stream 缓冲区也一并被销毁。这和标准的 streams 不同, 因为通常 `file streams` 在构造过程分配 stream 缓冲区, 并于析构时销毁它们。所以本例中的 `cout` 不能再被用于写入。事实上它甚至无法在程序结束时被安全销毁。因此我们应该保留旧缓冲区并于事后恢复! 下例中的函数 `redirect()` 就负责这件事:

```
// io/redirect.cpp

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

void redirect(ostream&);

int main()
{
```

```
    cout << "the first row" << endl;

    redirect(cout);

    cout << "the last row" << endl;
}

void redirect (ostream& strm)
{
    ofstream file("redirect.txt");

    // save output buffer of the stream
    streambuf* strm_buffer = strm.rdbuf();

    // redirect output into the file
    strm.rdbuf (file.rdbuf());

    file << "one row for the file" << endl;
    strm << "one row for the stream" << endl;

    // restore old output buffer
    strm.rdbuf (strm_buffer);

} // closes file AND its buffer automatically
```

程序输出如下：

```
the first row
the last row
```

文件 redirect.txt 的内容如下：

```
one row for the file
one row for the stream
```

正如你所见，redirect() 之中写入 cout (透过参数名 strm) 的输出信息被送到文件去了。main() 执行完 redirect() 后，接下来的输出又再次被送到 (已恢复了) 输出通道中。

13.10.4 用于读写的 Streams

本节列出“两个 streams 连接”的最后一例：运用同一个 stream 进行读写操作。通常我们可以利用 class `fstream` 打开一个文件进行读写：

```
std::fstream file ("example.txt", std::ios::in | std::ios::out);
```

也可以采用两个不同的 stream 对象，一个用于读取，一个用于改写。例如：

```
std::ofstream out ("example.txt", ios::in | ios::out);  
std::istream in (out.rdbuf());
```

`out` 的声明式会开启文件。`in` 的声明式使用 `out` 的 stream 缓冲区，从中读出数据。注意 `out` 必须同时允许读取和改写，如果仅能改写，则从它身上读取数据会导致未定义的行为。另外还请注意，`in` 并非隶属 `ifstream` 型别，只是隶属 `istream` 型别——文件被打开后，就有相应的 stream 缓冲区，此时唯一需要的只是另一个 stream 对象。一如前例，file stream object `out` 被销毁时，文件也被关闭。

你也可以产生一个 file stream 缓冲区，并将它安装在两个 stream 对象上，例如：

```
std::filebuf buffer;  
std::ostream out (&buffer);  
std::istream in (&buffer);  
buffer.open("example.txt", std::ios::in | std::ios::out);
```

其中的 `filebuf` 是 class `basic_filebuf<>` 对于字符型别 `char` 的特化实体，它定义出 file stream 所用的缓冲区类别。

以下程序是个完整实例。循环中对着一个文件写入四行。每次写完一行，文件的所有内容就被写到标准输出装置上：

```
// io/rwl.cpp  
  
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main()  
{  
    // open file "example.dat" for reading and writing  
    filebuf buffer;  
    ostream output(&buffer);  
    istream input(&buffer);  
    buffer.open ("example.dat", ios::in | ios::out | ios::trunc);
```

```
for (int i=1; i<=4; i++) {  
    // write one line  
    output << i << ". line" << endl;  
  
    // print all file contents  
    input.seekg(0); // seek to the beginning  
    char c;  
    while (input.get(c)) {  
        cout.put(c);  
    }  
    cout << endl;  
    input.clear(); // clear eofbit and failbit  
}
```

程序输出如下：

1. line

1. line
2. line

1. line
2. line
3. line

1. line
2. line
3. line
4. line

尽管两个不同的 `stream` 对象分别用于读取和改写，读写位置还是“紧耦合”（`tightly coupling`）的。`seekg()` 和 `seekp()` 都调用 `stream` 缓冲区的同一个成员函数¹¹。这样一来，为了保证能完整写出文件的全部内容，必须把读取位置置于文件起始处。文件的全部内容被写出后，读/写位置再次到达文件尾部，于是新行就可以追加到文件上。

¹¹ 实际上这个函数可以判断读取位置、写入位置、或两者是否被改变了。只有标准 `stream` 缓冲区才将读取和写入操作维护同一个位置。

有一点非常重要，除非读至文件尾部，否则必须在读取和改写操作之间进行一次 `seek`（搜寻）操作。否则极有可能导致一个被篡改的文件，甚至可能带来更致命的错误。

一如先前所说，你可以不必逐字处理，你可以运用单一语句打印出全部内容，只要将一个指向 `file stream` 缓冲区的指针当做参数，传给 `operator<<` 即可（详见 p683）：

```
std::cout << input.rdbuf();
```

13.11 String Stream Classes

`Stream classes` 机制也可以用来读取 `strings` 或将数据写入 `strings`。`String streams` 提供有缓冲区，但没有 I/O 通道；我们可以借着特殊函数来处理 `buffer/string`。这项技术的一个主要用途就是以“独立于真实 I/O 装置以外”的方式来处理 I/O。例如待输出文字的格式可以在 `string` 中设定，然后再将 `string` 发送到某个输出通道。也可以逐行读取输入，并以 `string streams` 处理每一行。

C++ 标准程序库将原有的 `strings stream classes` 以一系列新类别取代了。原先的 `string stream classes` 使用旧式型别 `char*` 来表现字符串，现在使用 `string`（或更泛化的 `basic_string<>`）。旧式的 `string stream classes` 仍然内含于 C++ 标准程序库中，但大家最好不要再使用。它们的保留纯粹只为了向下兼容，更新的标准版本极可能将它们全部删掉。所以编写新程序时应该尽量少用它们，甚至老旧程序最好也能够将它们替换掉。不过本节末尾还是对这些老旧类别做了一些简单描述。

13.11.1 String Stream Classes

下面的 `stream classes` 是针对 `string` 而定义的（与“针对 `file` 而定义”的 `stream` 互相对应）：

- `basic_istream`，以及特化版本 `istream` 和 `wistream`，用于从 `string` 读取数据，是所谓的 `input string stream`。
- `basic_ostream`，以及特化版本 `ostream` 和 `wostream`，用于将数据写入 `string`，是所谓的 `output string stream`。
- `basic_stringstream`，以及特化版本 `stringstream` 和 `wstringstream`，用于对 `string` 读写数据。
- `template class basic_stringbuf<>`，以及特化版本 `stringbuf` 和 `wstringbuf`，用来为其它 `string stream classes` 执行字符的实际读写操作。

这些 `classes` 和 `stream base classes` 之间的关系，就像 `file stream` 和 `stream base classes` 之间的关系一样。图 13.3 描述了其间的继承体系。

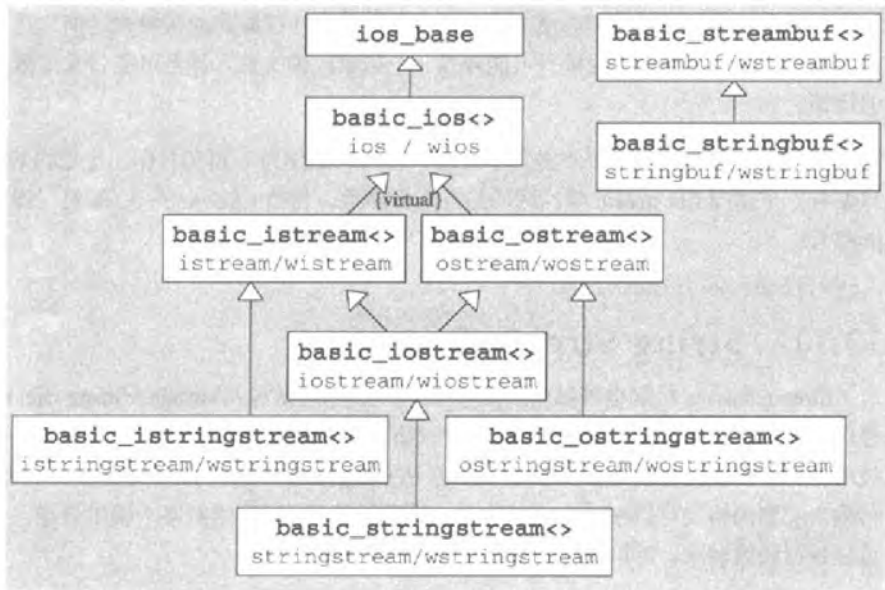


图 13.3 string stream classes 的继承关系

这些 classes 统统被声明于头文件 `<sstream>` 内：

```

namespace std {
    template <class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_istringstream;
    typedef basic_istringstream<char> istringstream;
    typedef basic_istringstream<wchar_t> wistringstream;

    template <class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
        class basic_ostringstream;
    typedef basic_ostringstream<char> ostringstream;
    typedef basic_ostringstream<wchar_t> wostringstream;

```

```

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_stringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_stringstream<wchar_t> wstringstream;

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_stringbuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
}

```

成员函数 `str()` 是 string stream classes 最主要的一个函数, 用来管理缓冲区(见表 13.39)。

表 13.39 String Streams 的基本操作

成员函数	意义
<code>str()</code>	将缓冲区内容当做一个 string 返回
<code>str(string)</code>	将 string 设为缓冲区内容

以下程序说明如何使用 string streams:

```

// io/sstr1.cpp

#include <iostream>
#include <sstream>
#include <bitset>
using namespace std;

int main()
{
    ostringstream os;

    // decimal and hexadecimal value
    os << "dec: " << 15 << " hex: " << 15 << endl;
    cout << os.str() << endl;
}

```



```
// append floating value and bitset
bitset<15> b(5789);
os << "float: " << 4.67 << " bitset: " << b << endl;

// overwrite with octal value
os.seekp(0);
os << "oct: " << oct << 15;
cout << os.str() << endl;
}
```

程序输出如下：

```
dec: 15 hex: f

oct: 17 hex: f
float: 4.67 bitset: 001011010011101
```

首先是一个十进制数和一个十六进制数写入 `os`，接下来再增加一个浮点数和一个 `bitset`。然后运用 `seekp()` 将改写位置设于 `stream` 起始处，这么一来后继的 `operator<<` 也就把输出写到 `string` 头部，于是覆盖了原本的 `string stream` 头部内容。未被覆盖的字符依然有效。如果你要删除 `stream` 的现有内容，可以利用函数 `str()` 将崭新内容赋予缓冲区：

```
strm.str("");
```

最先写入 `os` 的文字行系以 `endl` 结尾，表示以换行符号结束。由于字符串打印后紧跟着又输出 `endl`，所以有两个相邻的新行符号被写入，这也是为什么输出文字中带有空行的原因。

处理 `string stream` 时，一个典型的错误是忘了使用 `str()` 提取字符串，而直接往 `stream` 输出。从编译器的角度看，这是合情合理的，因为此处确实存在一个转换操作可将参数转换为 `void*`。于是 `stream` 的状态以地址形式被写入（参见 p596）。

往 `output string stream` 写入数据的一个典型应用就是，定义使用者自定义型别的 `output` 操作符（详见 13.12.1 节, p652）。

`Input string stream` 主要用途是“按格式，从现有字符串中读取数据”。例如我们可以轻易地逐行读取数据，然后分析每行数据。以下程序代码从字符串 `s` 中读得数值为 3 的整数 `x`，和数值为 0.7 的浮点数 `f`：

```
int x;
float f;
std::string s = "3.7";

std::istringstream is(s);
is >> x >> f;
```

产生 `string stream` 时, 可以运用文件开启模式作为标志 (参见 13.9.1 节, p631) 以及某个既有字符串作为初值。如果使用 `ios::app` 或 `ios::ate`, 写入 `string stream` 的字符便可以附加于该既有字符串之后:

```
std::string s;
...
std::ostringstream os (s, ios::out|ios::app);
os << 77 << std::hex << 77;
```

然而这就意味 `str()` 返回的字符串是 `s` 的副本, 并尾附 77 的一个十进制表述和一个十六进制表述。字符串 `s` 本身并未改变。

13.11.2 char* Stream Classes

`char* stream classes` 只是为了向下兼容才被保留下来。它们的接口容易出错, 不容易正确使用。但是既然曾经被广泛运用, 所以这里还是为你做一个简单介绍。请注意这里所说的标准版本, 已经对旧式接口作了一些修改。

本节采用术语“字符序列 *character sequence*”, 而不说字符串 *string*。是因为 `char* stream classes` 所维护的字符序列并不总是以字符串终止符号作为结束, 因此不能说它是一个真正的 *string*。

`char* stream classes` 是特别为字符型别 `char` 而定义的, 包括:

- `class istrstream`, 用来从字符序列中读取数据。是一种 `input string stream`。
- `class ostrstream`, 用来将数据写入字符序列。是一种 `output string stream`。
- `class strstream`, 用来对字符序列进行读写操作。
- `class strstreambuf`, 用来当做 `char* streams` 的缓冲区。

所有 `char* stream classes` 都在头文件 `<strstream>` 中定义。

`istrstream` 的初始化有两种做法, 一是“以字符串终止符号 `'\0'` 为结尾”的字符序列, 一是将字符数量也当做参数传给构造函数。`istrstream` 的一个典型用途就是读取并处理一整行文字:

```
char buffer[1000]; // buffer for at most 999 characters

// read line
std::cin.get(buffer, sizeof(buffer));

// read/process line as stream
std::istrstream input(buffer);
...
input >> x;
```

改写用的 `char* stream` 可以包含一个应需求而扩展的字符序列，也可以在初始化时带有一个定长缓冲区。如果运用 `ios::app` 或 `ios:ate`，便可以附加于缓冲区内既有的字符序列后面。

使用 `char* stream` 作为字符串时必须十分小心，因为它和 `string stream` 不同，它并不负责储存字符序列所需的内存。

利用成员函数 `str()`，字符序列可以和其调用者一起共同管理内存。除非 `stream` 被初始化为定长缓冲区（这么一来 `stream` 就不必负责），否则必须遵守下面三条原则：

1. 由于内存的拥有权移转给了调用者，所以如果 `stream` 没有被初始化为定长缓冲区，那么字符序列将被释放。但我们无法确定内存是如何被分配的¹²，在此情况下使用 `delete[]` 来释放字符序列并不安全。最安全的作法是调用成员函数 `freeze()` 并给予参数 `false`，将内存回传给 `stream`（稍后有个实例）。
2. 如果调用 `str()`，`stream` 便不能再修改字符序列。`Stream` 会暗中调用成员函数 `freeze()`，它会冻结字符序列。当缓冲区不够大以至于需要分配新缓冲区时，这么做可以避免事情变得复杂。
3. 成员函数 `str()` 不会附加字符串终止符号（`'\0'`）。我们只有直接在 `stream` 内加上该特殊字符，才能结束字符序列。这可通过操控器（manipulator）`ends` 完成。某些实作版本会自动添加字符串终止符号，但这种行为不具可移植性。

以下说明 `char* streams` 的用法：

```
float x;
...
/* create and fill char* stream
/* - don't forget ends or '\0' !!!
*/
std::ostream buffer; // dynamic stream buffer
buffer << "float x: " << x << std::ends;

// pass resulting C-string to foo() and return memory to buffer
char* s = buffer.str();
foo(s);
buffer.freeze(false);
```

¹² 实际上存在一个“接受两个函数指针作为参数”的构造函数，其中一个函数用于分配内存，另一个用于释放内存。

为了方便其它处理，被冻结的 `char* stream` 可以恢复其正常状态。要做到这一点，必须调用成员函数 `freeze()` 并传入参数 `false`。这么一来字符序列的所有权就转给了 `stream` 对象。这也是释放字符序列内存的唯一安全做法。下面例子示范这个做法：

```
float x;
...
std::ostream buffer; // dynamic char* stream

// fill char* stream
buffer << "float x: " << x << std::ends;

/* pass resulting C-string to foo()
 * - freezes the char* stream
 */
foo(buffer.str());

// unfreeze the char* stream
buffer.freeze(false);

// seek writing position to the beginning
buffer.seekp (0, ios::beg);

// refill char* stream
buffer << "once more float x: " << x << std::ends;

/* pass resulting C-string to foo() again
 * - freezes the char* stream
 */
foo(buffer.str());

// return memory to buffer
buffer.freeze(false);
```

`string stream classes` 之中已经不再有与“冻结 `stream`”相关的任何问题，主要是因为字符串被复制了，而且 `string classes` 负起了内存相关责任。

13.12 “使用者自定义型别”之 I/O 操作符

正如本章先前所说, 和旧式的 C I/O 机制比较, stream 的主要优势在于: stream 机制可以扩及使用使用者自定义型别 (user-defined types)。要做到这一点, 你必须将操作符 << 和 >> 重载。以下小节透过一个 class Fraction (分数) 进行说明。

13.12.1 实作一个 output 操作符

在一个带有 output 操作符的表达式中, 左操作数是 stream, 右操作数是待写对象:

```
stream << object
```

根据语法规则, 上式有两种解释:

1. stream.operator<<(object)
2. operator<<(stream, object)

第一种方式用于内建 (基本) 型别。至于使用者自定义型别必须采用第二种方式, 因为 stream class 的扩展之门已经关闭。你需要做的就是针对你的自定义型别实作出全局性的 operator<<。这很简单, 除非你需要存取对象的私有成员 (这一点稍后再来解决)。

例如, 如果要按“分子/分母”的格式打印 Fraction 对象, 你可以编写以下函数:

```
// io/fraclout.hpp

#include <iostream>

inline
std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    strm << f.numerator() << '/' << f.denominator();
    return strm;
}
```

这个函数将分子和分母作为参数写入 stream, 其间以字符 “/” 隔开。这里的 stream 可以是 file stream、string stream 或其它任何 streams。为了支持输出操作的链式 (chaining) 形式, 或是为了在同一个语句中处理 streams 状态, 上述函数必须返回当前的 stream。

这种简单形式有两个缺点:

1. 由于标记式 (signature) 中使用了 `ostream`，所以这个函数只适用于“字符型别为 `char`”的 streams。如果这个函数要在西欧或北美使用，当然毫无问题。回头来说，要开发一个更通用的版本并不很费事，值得考虑。
2. 如果字段宽度 (field width) 被设定，会产生新问题：得到的结果可能和预期不同。字段宽度会施行于紧随其后的改写操作中，此处将施行于分子身上。因此，以下语句：

```
Fraction vat(16,100); // I'm German and we have a uniform VAT of 16%...
std::cout << "VAT: \" << std::left << std::setw(8)
          << vat << '\"' << std::endl;
```

会导致以下输出：

```
VAT: "16      /100"
```

下面这个版本可以同时解决上述两个问题：

```
// io/frac2out.hpp

#include <iostream>
#include <sstream>

template <class charT, class traits>
inline
std::basic_ostream<charT,traits>&
operator << (std::basic_ostream<charT,traits>& strm,
            const Fraction& f)
{
    /* string stream
     * - with same format
     * - without special field width
     */
    std::basic_ostringstream<charT,traits> s;
    s.copyfmt(strm);
    s.width(0);

    // fill string stream
    s << f.numerator() << '/' << f.denominator();

    // print string stream
    strm << s.str();

    return strm;
}
```

新版本的操作符已经成了一个“可适用于各种 streams”的 template function。字段宽度的问题已经获得解决：先将分数写入一个 string stream，不设任何字段宽度；再将这个构造好的字符串作为参数传给 stream。这导致用以表现分数的那些字符仅以一次改写操作就完成，而字段宽度就实施于该唯一的改写操作上。于是，以下语句：

```
Fraction vat(16,100); // I'm German ...
std::cout << "VAT: \" << std::left << std::setw(8)
          << vat << '\"' << std::endl;
```

产生如下结果：

```
VAT: "16/100 "
```

13.12.2 实作一个 input 操作符

input 操作符的实作原则和 output 操作符相同（参见先前小节），但输入时可能产生诸如“读取失败”等问题。一旦出现这种情况就需要作一些特殊处理。

实作一个读取函数时，可以采用简单法或灵活法。以下函数就采用了简单法，它不检测（不理睬）出错状态，直接读出分数：

```
// io/fraclin.hpp
#include <iostream>

inline
std::istream& operator >> (std::istream& strm, Fraction& f)
{
    int n, d;

    strm >> n;          // read value of the numerator
    strm.ignore();       // skip '/'
    strm >> d;          // read value of the denominator

    f = Fraction(n,d); // assign the whole fraction

    return strm;
}
```

这种做法只适用于“字符型别 `char` 所对应的 `streams`”，而且也不检测两数字间的符号是否为 `'/'`。

如果读取的是个未定义值，会引发其它问题。当读入的分母值为 0，读入的分数就没有定义。函数 `Fraction(n,d)` 所触发的 `Fraction` 构造函数可以检测出这个问题。但是在 `class Fraction` 内部处理此事，意味一个（输入）格式错误（例如分母为 0）会自动导致一个错误处理（`error handling`）。由于通常大家习惯在 `stream` 中记录格式错误，所以此时你最好设立 `ios_base::failbit`。

最后一点，即使读取不成功，以 `by reference` 方式传入的那个分数也可能被修改。当分子读取成功而分母读取失败时就可能发生这种情况。此种行为和“语言预先定义好的各种 `Input` 操作符”惯例不符，所以最好避免，使读取操作“要么成功、要么就完全无效（不带来任何影响）”。

下面是改进后的实作版本，避免了上述种种问题。由于函数已被参数化，得以适用所有 `stream`，所以更加灵活：

```
// io/frac2in.hpp

#include <iostream>

template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
operator >> (std::basic_istream<charT,traits>& strm, Fraction& f)
{
    int n, d;

    // read value of numerator
    strm >> n;

    /* if available
     * - read '/' and value of demonimator
     */
    if (strm.peek() == '/') {
        strm.ignore();
        strm >> d;
    }
    else {
        d = 1;
    }
}
```



```
/* if denominator is zero
 * - set failbit as I/O format error
 */
if (d == 0) {
    strm.setstate(std::ios::failbit);
    return strm;
}

/* if everything is fine so far
 * change the value of the fraction
 */
if (strm) {
    f = Fraction(n,d);
}
return strm;
}
```

此处，只有当第一个数之后紧跟着字符 ‘/’，才会继续读入分母，否则便假设分母为 1，而先前读入的整数被当做完整分数内容。所以实际输入时分母可有可无。

这个版本也会检查分母值是否为 0。如果真是 0，`ios_base::failbit` 会被设立起来，那会触发相应的异常（见 13.4.4 节，p602）。当然，如果分母为 0，也可以执行别的操作，例如直接抛出异常，或直接跳开检查并让分数为 0，后者将由 `class Fraction` 抛出相应的异常。

函数最后会检查 `stream` 状态，只有在无任何错误发生时才为分数赋予新值。这类最后检验工作总是应该实施，保证“唯有读取成功，对象值才会被改变”。

当然，“读取整数并视之为分数”的作法是否合理，有待商榷。此外还有一些细节需要改进。例如分子后面应该紧邻符号 ‘/’，中间不能有空格，但分母后面允许任意多个空格，因为它们通常会被忽略。要做到这些，势必增加读取时的复杂度。

13.12.3 以辅助函数完成 I/O

如果执行 I/O 操作符时需要存取对象的私有成员，标准操作符应该将实际任务委派给辅助的成员函数。这种技术允许具有多态性（polymorphic）的读写函数，其具体实现可如下实例：

```
class Fraction {
    ...
public:
    virtual void printOn (std::ostream& strm) const; // output
    virtual void scanFrom (std::istream& strm);      // input
    ...
};

std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    f.printOn (strm);
    return strm;
}

std::istream& operator >> (std::istream& strm, Fraction& f)
{
    f.scanFrom (strm);
    return strm;
}
```

一个典型的例子就是在输入时直接处理分子和分母：

```
void Fraction::scanFrom (std::istream& strm)
{
    ...
    // assign values directly to the components
    num = n;
    denom = d;
}
```

如果你的 `classes` 并不打算被当做基类，那么你的 I/O 操作符可以设计为其 `friends`。但是要注意，一旦用上了继承，这种方法就会有极大的局限性。`friend` 函数不能成为虚函数，所以你的程序可能会调用错误的函数。例如，如果某个 `base class` `reference` 实际指向一个 `derived class object`，并被当做 `input` 操作符的参数，则被调用的将是 `base class` 的操作符。为了避免出现这种情况，`derived class` 不得实作自己的 I/O 操作符。因此，先前的实作手法比 `friend` 函数手法通用得多。因此尽管你在其它文件或书籍上看到的绝大多数例子都使用 `friend` 函数，你还是应该把前一种手法视为标准作法。

13.12.4 以非格式化函数完成使用者自定义的操作符

前一小节实作的 I/O 操作符把大部分工作都委托给某些预先定义好的、用于格式化 I/O 的操作符。也就是说，`operator<<` 和 `operator>>` 是以更基本型别的对应操作符实作而得。

C++ 标准程序库中的 I/O 操作符各有定义。面对这些操作符，普遍适用的方案是：先备妥经过处理的 stream（准备用于实际 I/O），再完成实际 I/O 及后续处理。这个方案也应该运用于你自己的 I/O 操作符，以保持 I/O 操作符的一致性。

Class `basic_istream` 和 `basic_ostream` 都定义有一个辅助类别 `sentry`（岗哨），其构造函数负责进行预处理，析构函数负责对应的后处理。这些 classes 可以取代 `IOStream` 程序库旧式版本的一些成员函数（`ipfx()`, `isfx()`, `opfx()`, `osfx()`）。采用这些 classes 可以确保即使 I/O 由于异常发生而失败，也能调用后处理程序。

如果某个 I/O 操作符使用了一个非格式化的 I/O 函数，或是直接对 stream 缓冲区进行操作，那么第一件要做的事情就应该是构建一个对应的 `sentry`（岗哨）对象，其余处理应该取决于该对象的状态——那可以说明 stream 是否正常良好。将 `sentry` 对象转换为 `bool`，便可检测其状态。所以 I/O 操作符通常会这么做：

```
sentry se(strm); // indirect pre- and postprocessing
if (se) {
    ...           // the actual processing
}
```

`sentry` 构造函数接受 stream `strm` 为参数。预处理和后处理都针对 `strm` 进行。

还有一些额外的处理用于筹备 I/O 操作符的一般工作，包括多个 streams 之间的同步化（synchronizing）、检查 stream 是否正常、跳过空格，以及其它可能的特定任务。例如在多线程环境中可进行相应的锁操作。

对于 input streams，`sentry` 构造函数有个可有可无的布尔值参数，用来决定是否跳过空格（无视 `skipws` 标志是否设立）：

```
sentry se(strm, true); // don't skip whitespaces during the additional processing
```

以下关于 class `Row`（用来在文字处理器或编辑器中表现单行）的实例可以说明这一点：

- output 操作符运用 stream 成员函数 write() 来写入一行:

```
std::ostream& operator<< (std::ostream& strm, const Row& row)
{
    // ensure pre- and postprocessing
    std::ostream::sentry se(strm);

    if (se) {
        // perform the output
        strm.write(row.c_str(), row.len());
    }
    return strm;
}
```

- input 操作符在循环中逐字读取一行。参数 true 被传给 sentry 对象的构造函数，以免错过空格:

```
std::istream& operator>> (std::istream& strm, Row& row)
{
    /* ensure pre- and postprocessing
     * - true: Yes, don't ignore leading whitespaces
     */
    std::istream::sentry se(strm, true);
    if (se) {
        // perform the input
        char c;
        row.clear();
        while (strm.get(c) && c != '\n') {
            row.append(c);
        }
    }
    return strm;
}
```

即使函数不使用非格式化函数，而以 I/O 操作符取代，也能使用上述框架。但是没有必要在以 sentry 为岗哨的检测码中使用 basic_istream 或 basic_ostream 的成员函数来读/写字符，那样消耗太大。应该尽可能使用对应的 basic_streambuf。

13.12.5 使用者自定义的格式标志 (Format Flags)

一旦“使用者自定义操作符”被实作出来，程序往往需要一些格式标志，用来对那些操作符提供指示。这可通过相应的操控器 (manipulator) 完成。举个例子，先前所说的分数 (Fraction)，其 output 操作符如果允许在分子和分母间的斜线两边安插空格，那可漂亮了。

Stream 对象提供一种机制，可将数据关联到 stream 身上。这样的机制可以用来将相应数据关联起来（例如使用一个操控器），之后再取回。此一机制即可支持上述技术。Class ios_base 定义有两个函数 iword() 和 pword()，两者都以 int 参数为索引，分别存取特定的 long& 或 void*&。其想法是以 iword()/pword() 分别存取储存于数组（任意长度，伴随 stream 对象）中的 long/void* 对象；我们把 stream 格式标志放到一个特定位置上，该位置对所有 stream 而言都相同。Class ios_base 的静态成员函数 xalloc() 用来取得一个尚未被用于此目的的索引值。

一开始，被 iword() 或 pword() 存取的对象都被设置为 0。这个值可以用来表现缺省格式，或表示“相应的数据尚未被存取”。下面是个例子：

```
// get index for new ostream data
static const int iword_index = std::ios_base::xalloc();

// define manipulator that sets this data
std::ostream& fraction_spaces (std::ostream& strm)
{
    strm.iword(iword_index) = true;
    return strm;
}

std::ostream& operator<< (std::ostream& strm, const Fraction& f)
{
    /* query the ostream data
     * - if true, use spaces between numerator and denominator
     * - if false, use no spaces between numerator and denominator
     */
    if (strm.iword(iword_index)) {
        strm << f.numerator() << " / " << f.denominator();
    }
    else {
        strm << f.numerator() << "/" << f.denominator();
    }
    return strm;
}
```

本例只对 output 操作符采取简易作法，因为本例的主要目的是揭示函数 iword() 的用法。这里的格式标志被视为一个布尔值，用来定义“分子和分母之间能否存在空格”。第一行内的函数 ios_base::xalloc() 用于获取“可储存格式标志”的索引值，所得结果绝不应该被修改，所以我把它存在一个常量中。fraction_spaces() 是个操控器，用来设置 int 值为 true（该值储存于 stream 相关整数数组的索引 iword_index 处）。output 操作符会重新获得该值，并根据其值输出分数。如果其值为 false，采用缺省的“不支持空格”格式。否则就在斜线两端加入空格。

`isword()` 和 `isword()` 会返回 `int` 对象或 `void*` 对象的 reference。这些 references 将持续有效，除非你针对相应的 `stream` 对象再次调用 `isword()` 或 `isword()`，或是你销毁了 `stream` 对象。通常 `isword()` 和 `isword()` 返回的结果不需要保存。一般的共识是：这是一个速度很快的存取操作，尽管并无明确要求非采用数组不可。

函数 `copyfmt()` 会复制所有格式信息（见 p615），包括 `isword()` 和 `isword()` 存取的数组。这可能会给某些对象带来问题——那些对象系储存于某个曾经运用 `isword()` 的 `stream` 内）。举个例子，如果某值为对象地址，那么被复制的将是地址而不是对象。如果只复制地址，一旦 `stream` 格式发生改变，其它 `stream` 的格式也会受影响。此外，如果一个 `stream` 用上 `isword()`，那么当它被销毁时，它所关联的对象最好一并销毁。所以对这样的对象而言，应该使用深拷贝（`deep copy`）而不是浅拷贝（`shallow copy`）。

`ios_base` 定义了一个回调（`callback`）机制，用以支持两种行为：(1) 必要时执行深拷贝，(2) 销毁 `stream` 时连带删除某个对象。函数 `register_callback()` 可用来注册某个函数，该函数会在某些特定操作发生于 `ios_base` 对象身上时被调用。

```
namespace std {
    class ios_base {
    public:
        // kinds of callback events
        enum event { erase_event, imbue_event, copyfmt_event };
        // type of callbacks
        typedef void (*event_callback) (event e, ios_base& strm,
                                         int arg);
        // function to register callbacks
        void register_callback (event_callback cb, int arg);
        ...
    };
}
```

`register_callback()` 的第一参数和第二参数分别是一个函数指针和一个 `int` 值。这个 `int` 值将成为已注册函数（被调用时）的第三参数，可以（例如）用来表现一个索引，标示出 `isword()` 要处理的数组元素。回调函数的第二参数 `strm` 是个 `ios_base` 对象，正是该对象引发了对回调函数的调用。第一参数 `e` 用来区分回调的原因。表 13.40 列出所有（三种）回调原因。

表 13.40 回调时间的发生原因

事件 (events)	发生原因
<code>ios_base::imbue_event</code>	程序运用 <code>imbue()</code> 设定了一个 locale
<code>ios_base::erase_event</code>	stream 被摧毁, 或 <code>copyfmt()</code> 被执行起来
<code>ios_base::copy_event</code>	<code>copyfmt()</code> 被执行

如果 `copyfmt()` 被执行, 则执行者 (某对象) 将进行两次回调操作。首先是在复制之前进行第一次回调并携带参数 `erase_event`, 用以完成必要的清除工作 (例如删除数组中的对象), 此时调用的是针对该对象注册好的回调函数。其次是格式标志复制完毕后 (其中包括随参数而来的 stream 的回调列表 (callbacks list)), 再次调用回调函数并携带参数 `copy_event`。这一次操作可以 (例如) 用来给商对储存于 `pword()` 数组中的对象的深拷贝 (deep copy)。注意此时整个回调串行也被复制了, 原先的串行则被移除。所以第二次回调操作所激发的就是才刚刚被复制过来的回调函数。

这个回调机制非常基本 (primitive)。所有回调函数都必须先注册, 除非你使用 `copyfmt()` 并携带一个参数而其中无任何注册函数。此外如果你将某个回调函数注册两次, 那么即使是带有相同的参数, 也会导致它被调用两次。回调次序肯定和注册次序相反。这导致一个问题: 如果回调函数 B 是在回调函数 A 内注册的, 那么 B 在 A 下次被回调之前, 绝不可能被回调起来。

13.12.6 使用者自定义之 I/O 操作符的数个依循惯例

运用自行定义之 I/O 操作符时, 有些惯例应该遵循。这些惯例和预先定义之 I/O 操作符的典型操作是对应的, 总结如下:

- 你的输出格式应该让其它使用者能够以一个 Input 操作符无损地读取数据。尤其是 strings, 由于空格符的问题, 几乎不可能实现这一点。是的, “strings 内的空格符” 很难和 “两个 string 之间的空格符” 区分开来。
- 进行 I/O 时, 应考虑现有的 stream 格式规范, 尤其在处理改写宽度的时候。
- 如果产生错误, 应设置相应的状态位 (state flag)。
- 如果发生错误, 原对象不该有任何改变。如果读取多笔数据, 读入的数据应该在被设为 “被当做参数传递” 的对象前, 先储存于辅助对象中。
- 输出不应该以 new line 符号结束, 否则就不能在同一行输出其它对象了。
- 即使数值太大, 也应该完全读入。读完后设置相应的错误标志。此时的返回值应具备特定意义, 例如返回可容许的最大值。
- 如果检测到某个格式错误 (format error), 应该尽可能不读取任何字符。

13.13 Stream Buffer Classes

一如 13.2.1 节, p589 所介绍, `stream` 并不负责实际读写操作, 而是委托给 `stream buffers` (缓冲区) 完成。本节说明缓冲区如何运作。如此一来不仅能加深理解 `stream` 的运用结果, 也为“定义新的 I/O 通道”提供技术基础。在详细介绍 `stream` 缓冲区的操作之前, 本节先为“只对 `stream` 缓冲区之运用”感兴趣的人介绍其 `public` 接口。

13.13.1 从使用者的角度看 Stream 缓冲区

对于 `stream` 缓冲区的使用者来说, `class basic_streambuf` 只是发送 (`sent`) 或提取 (`extracted`) 字符的地方。表 13.41 列出两个 `public` 函数, 用来写入字符。

表 13.41 `public` 成员函数, 用来写入字符

成员函数	意义
<code>sputc(c)</code>	将字符 <code>c</code> 送入 <code>stream</code> 缓冲区
<code>sputn(s, n)</code>	将字符序列 <code>s</code> 中的 <code>n</code> 个字符送入 <code>stream</code> 缓冲区

如果发生错误, 函数 `sputc()` 返回 `traits_type::eof()`, 这里的 `traits_type` 是 `class` 内的一个型别定义。函数 `sputn()` 将写入由第二参数指定的字符数, 除非 `stream` 缓冲区无法耗用 (`consume`) 它们。这个函数并不考虑字符串终止符号; 它返回的是实际写出的字符数。

“从 `stream` 缓冲区中读取字符”的接口比较复杂 (表 13.42)。这是因为对输入而言, 必须时时观察监视未耗用 (`consuming`) 的字符。解析 (`parsing`) 时, 字符最好能被送回 `stream` 缓冲区。为此, `stream buffer classes` 特别提供了相应函数。

表 13.42 `public` 成员函数, 用来读取字符

成员函数	意义
<code>in_avail()</code>	返回有效字符的下界 (lower bound)
<code>sgetc()</code>	返回当前字符, 不耗用它 (<i>without consuming it</i>)
<code>sputc()</code>	返回当前字符并耗用它 (<i>consuming it</i>)
<code>snextc()</code>	耗用当前字符并返回下一个字符
<code>sgetn(b, n)</code>	读取 <code>n</code> 个字符, 并将它们存储到缓冲区 <code>b</code>
<code>sputbackc(c)</code>	将字符 <code>c</code> 返回 <code>stream</code> 缓冲区
<code>sungetc()</code>	退回至前一个字符

`in_avail()` 可用来确定至少有多少个有效字符。这可用来 (举个例子) 确定从

键盘读取数据时不会阻塞。不过实际上可能有更多有效字符。

除非 stream 缓冲区已经到达 stream 尾部, 否则始终存在一个当前 (current) 字符。函数 `sgetc()` 可以不必移至下一字符就获得当前字符。函数 `sbumpc()` 读取当前字符并移至下一字符, 使之成为当前字符。函数 `snextc()` 将下一个字符视为当前字符然后读取之。这三个函数如果失败都会返回 `traits_type::eof()`。函数 `sgetn()` 读取字符序列并送至缓冲区, 其参数可以代表欲读取的字符数。返回的则是实际读取的字符数。

函数 `sputbackc()` 和 `sungetc()` 被用来退一步, 使前一个字符成为当前字符。函数 `sputbackc()` 可将前一字符替换为其它字符。使用这两个函数必须谨慎。通常它们只能退一个字符。

还有一些函数用来存取局部对象、改变位置、或影响缓冲区, 见表 13.43。

表 13.43 难以分类的 Stream Buffer public 函数

成员函数	意义
<code>pubimbue(loc)</code>	为 stream 缓冲区安装 locale <code>loc</code>
<code>getloc()</code>	返回当前 (current) 的 locale
<code>pubseekpos(pos)</code>	将当前 (current) 位置重新设定为某绝对位置
<code>pubseekpos(pos, which)</code>	与上同, 并可指定 I/O 方向
<code>pubseekoff(offset, rpos)</code>	将当前位置重新设定为另一位置的相对位置
<code>pubseekoff(offset, rpos, which)</code>	与上同, 并可指定 I/O 方向
<code>pubsetbuf(b, n)</code>	影响缓冲行为

`pubimbue()` 和 `getloc()` 用于国际化议题 (参见 p625)。`pubimbue()` 在 stream 缓冲区中安装一个新的 locale 对象, 并返回前一个被安装的 locale 对象; `getloc()` 返回当前的 locale 对象。

函数 `pubsetbuf()` 试图对 stream 缓冲区的缓冲策略进行某种控制, 是否有效则取决于具体的 stream buffer classes。例如对 string stream 缓冲区运用 `pubsetbuf()` 就毫无意义。即使将它用于 file stream 缓冲区, 也只能在第一个 I/O 操作完成后以 `pubsetbuf(0, 0)` (意即不采用缓冲区) 方式调用才能起作用。如果出错, 函数返回 0, 否则返回该 stream 缓冲区。

函数 `pubseekoff()` 和 `pubseekpos()` 控制读写操作的当前位置。究竟控制的是读或写, 取决于最后一个参数 (其型别为 `ios_base::openmode`), 如果没有特别指定, 参数默认值为 `ios_base::in | ios_base::out`。一旦设置了 `ios_base::in`, 读取位置就会跟着改变, 一旦设置了 `ios_base::out`, 改写位置也会跟着变

第一行根据 `cout` 构造了一个 `output` 迭代器，型别为 `ostreambuf_iterator`。除了传递 `output stream` 外，你也可以直接传递一个指针，指向 `stream` 缓冲区。其余程序代码构造出一个 `string` 对象，并将其内字符复制到上述 `output` 迭代器。

表 13.44 列出 `output stream` 缓冲区迭代器的所有操作函数。其实作和 `ostream` 迭代器近似（参见 p278）。此外你也可以拿一个缓冲区将迭代器初始化，并调用 `failed()` 检查迭代器能否用于输出。如果有任何一个字符的预写入操作失败，`failed()` 就会返回 `true`。此时以 `operator=` 进行的任何改写操作都无效。

表 13.44 Output Stream 缓冲区迭代器的各项操作

算式	效果
<code>ostreambuf_iterator<char>(ostream)</code>	为 <code>ostream</code> 产生一个 <code>output stream</code> 缓冲区迭代器
<code>ostreambuf_iterator<char>(buffer_ptr)</code>	为 <code>buffer_ptr</code> 所指的缓冲区产生一个 <code>output stream</code> 缓冲区迭代器
<code>*iter</code>	无操作 (no-op)，返回 <code>iter</code>
<code>iter = c</code>	调用 <code>sputc(c)</code> ，对缓冲区写入字符 <code>c</code>
<code>++iter</code>	无操作 (no-op)，返回 <code>iter</code>
<code>iter++</code>	无操作 (no-op)，返回 <code>iter</code>
<code>failed()</code>	判断 <code>output stream</code> 迭代器是否能执行改写操作

Input Stream 缓冲区迭代器

表 13.45 列出 `input stream` 缓冲区迭代器的所有操作函数。其实作和 `istream` 迭代器近似（参见 p280）。此外你可以拿一个缓冲区来初始化迭代器。成员函数 `equal()` 用来判断两个 `input stream` 缓冲区迭代器是否相等。当两个 `stream` 缓冲区迭代器都是（或都不是）`end-of-stream` 迭代器时，两者被视为相等。

让人不明白的是到底“型别为 `istreambuf_iterator` 的两个对象”相等，是什么意思。如果两个 `istreambuf_iterator` 迭代器都是（或都不是）`end-of-stream` 迭代器，则两者被视为相等（至于其 `output` 缓冲区是否相同，并不影响）。获得一个 `end-of-stream` 迭代器的可行方法是以 `default` 构造函数产生一个迭代器。此外如果试图将迭代器移至 `stream` 尾部之外（也就是说如果 `sputc()` 返回 `traits_type::eof()`），`istreambuf_iterator` 就会成为一个 `end-of-stream` 迭代器。这种行为有两个主要含义：

表 13.45 Input Stream 缓冲区迭代器的各项操作

算式	效果
<code>istreambuf_iterator<char>()</code>	产生一个 <i>end-of-stream</i> 迭代器
<code>istreambuf_iterator<char>(istream)</code>	为 <i>istream</i> 建立一个 input stream 缓冲区迭代器，并可能调用 <code>sgetc()</code> 读取第一个字符
<code>istreambuf_iterator<char>(buffer_ptr)</code>	为 <i>buffer_ptr</i> 所指向的 input stream 产生一个 input stream 缓冲区迭代器，并可能调用 <code>sgetc()</code> 读取第一个字符
<code>* iter</code>	返回当前字符，也就是先前以 <code>sgetc()</code> 读取的字符（如果构造函数未执行读取操作，这里执行）
<code>++iter</code>	以 <code>sbumpc()</code> 读取下一字符，并返回其位置
<code>iter++</code>	以 <code>sbumpc()</code> 读取下一字符，返回一个迭代器，指向前一个字符位置
<code>iter1.equal(iter2)</code>	判断是否两个迭代器相等
<code>iter1 == iter2</code>	判断是否两个迭代器相等
<code>iter1 != iter2</code>	判断是否两个迭代器不相等

- 1. 从“stream 当前位置”到“stream 尾部”之间的范围，由以下两个迭代器定义出来：`istreambuf_iterator<charT,traits>(stream)`（此乃针对当前位置）和 `istreambuf_iterator<charT,traits>()`（此乃针对 stream 尾部），其中 stream 的型别是 `basic_istream<charT,traits>` 或 `basic_streambuf<charT,traits>`。
- 2. 不可能以 `istreambuf_iterators` 建立出一个子序列。

Stream 缓冲区迭代器运用实例

下面的例子是一个典型的筛检框架（filter framework），它用 stream 缓冲区迭代器简单地写出所有读取字符，是 p611 页例子的一个改进版本：

```
// io/charcat2.cpp

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

int main()
{
    // input stream buffer iterator for cin
    istreambuf_iterator<char> inpos(cin);
```

```

// end-of-stream iterator
istreambuf_iterator<char> endpos;

// output stream buffer iterator for cout
ostreambuf_iterator<char> outpos(cout);

// while input iterator is valid
while (inpos != endpos) {
    *outpos = *inpos; // assign its value to the output iterator
    ++inpos;
    ++outpos;
}
}

```

13.13.3 使用者自定义的 Stream 缓冲区

Stream 缓冲区是一种 I/O 缓冲区，其接口由 `class basic_streambuf<>` 定义。针对字符型别 `char` 和 `wchar_t`，标准程序库分别提供了预先定义好的 `streambuf` 和 `wstreambuf`。实作特殊 I/O 通道上的通信时，以上 classes 可被拿来当做基类。要做到这一点，首先得对 stream 缓冲区的操作有所了解。

缓冲区的主要接口由三个指针(指向两个缓冲区)构成。函数 `eback()`, `gptr()`, `egptr()` 返回的指针构成了 `read (input)` 缓冲区的界面。函数 `pbase()`, `pptr()`, `epptr()` 返回的指针构成了 `write (output)` 缓冲区的接口。这些指针分别由 I/O 操作操控，后者可能导致相关 I/O 通道上的相关响应。精确操作将分为读取或写入来讨论。

使用者自定义的 output 缓冲区

output 缓冲区由三个指针维护，三个指针分别由函数 `pbase()`, `pptr()`, `epptr()` 取得(图 13.4)。它们表示的意义为：

1. `pbase()` (意思是 "put base") 是 output stream 缓冲区的开始。
2. `pptr()` (意思是 "put pointer") 是当前写入位置。
3. `epptr()` (意思是 "end put pointer") 是 output 缓冲区的结尾，指向“得被缓冲 (*can be buffered*) 之最后一个字符”的下一位置。

`pbase()` 至 `pptr()` 之间的序列字符 (不包括 `pptr()` 所指字符) 已被写至相应的输出通道，但尚未清空 (flush)。

成员函数 `sputc()` 可以写入一个字符；如果当时有个空的改写位置，字符就被复制到该位置上。然后，指向当前改写位置的那个指针值会加 1。如果缓冲区是空

的 (`pptr() == epptr()`)，就调用虚函数 `overflow()` 将 output 缓冲区的内容发送到对应的输出通道去。这个函数能有效地把字符送至某种“外部表述”（但实际上也可能是内部的，例如 `string stream`）。基类 `basic_streambuf` 所实作的 `overflow()` 只返回 *end-of-file*，表示没有更多字符可被写入。

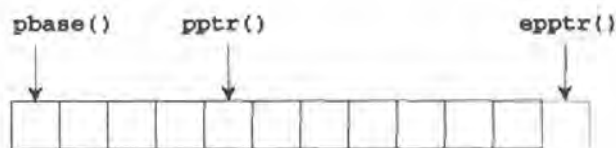


图 13.4 Output 缓冲区的界面

成员函数 `sputn()` 可用来一次写入多个字符。这个函数把实际任务委派给虚函数 `xsgputn()`，后者可针对“多个字符”做出更有效的操作。Class `basic_streambuf` 中的 `xsgputn()` 对每个字符调用 `sputc()`，因此改写 `xsgputn()` 并非必要。但通常“同时写入多个字符”会比“一次写入一个字符”效率高得多，所以 `sputn()` 可用来优化对字符序列的处理。

对着一个 `stream` 缓冲区写入数据时不必一定得采取缓冲行为，而是可以令字符一到达就写入。此时 `default` 构造函数会自动将“维护 `write` 缓冲区”的指针设为 0 或 `NULL`。

利用上述信息，我们可以实作出以下所示的 `stream` 缓冲区简单例子。这个 `stream` 缓冲区并未采取缓冲行为，所以对每个字符都调用 `overflow()`——我们唯一要做的就是实作出这个函数：

```
// io/outbuf1.hpp

#include <streambuf>
#include <locale>
#include <cstdio>
```

```

class outbuf : public std::streambuf
{
protected:
    /* central output function
    * - print characters in uppercase mode
    */
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            // convert lowercase to uppercase
            c = std::toupper(c, getloc());

            // and write the character to the standard output
            if (putchar(c) == EOF) {
                return EOF;
            }
        }
        return c;
    }
};

```

本例之中，每一个被发送到 stream 缓冲区的字符，都以 C 函数 `putchar()` 写入。但是写入之前这些字符都被 `toupper()` 转换成大写（参见 p718）。函数 `getloc()` 用来取得和 stream 缓冲区有关的 locale 对象（参见 p626）。

本例的 output 缓冲区是特别针对字符型别 `char` 实作的（`streambuf` 就是特别针对字符型别 `char` 的一个 `basic_streambuf<>` 特化版本）。如果你还用其它字符型别，就必须运用字符特性（character traits，详见 p687, 14.1.2 节）来实作这个函数。这种情况下 *end-of-file* 和 `c` 的比较操作应该有所不同，返回的应该是 `traits::eof()` 而不是 `EOF`。如果参数 `c` 是 `EOF`，则应该返回 `traits::not_eof(c)`（其中 `traits` 是 `basic_streambuf` 的第二个 `template` 参数）。情况大致如下：

```

// io/outbuf1x.hpp

#include <streambuf>
#include <locale>
#include <cstdio>

template <class charT, class traits = std::char_traits<charT> >
class basic_outbuf : public std::basic_streambuf<charT, traits>
{
protected:
    /* central output function
    * - print characters in uppercase mode
    */
    virtual typename traits::int_type overflow (
        typename traits::int_type c) {
        if (!traits::eq_int_type(c, traits::eof())) {

```

```

        // convert lowercase to uppercase
        c = std::toupper(c, getloc());

        // and write the character to the standard output
        if (putchar(c) == EOF) {
            return traits::eof();
        }
    }
    return traits::not_eof(c);
}

};

typedef basic_outbuf<char> outbuf;
typedef basic_outbuf<wchar_t> woutbuf;

```

以下是上面这个 stream 缓冲区的运用情况：

```

// io/outbuf1.cpp

#include <iostream>
#include "outbuf1.hpp"

int main()
{
    outbuf ob;                // create special output buffer
    std::ostream out(&ob);    // initialize output stream with that output buffer

    out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}

```

输出如下（译注：所有字符都自动变成大写）：

```
31 HEXADECIMAL: 1F
```

同样方法可用于对其它任意目标的写入操作。例如 stream 缓冲区的构造函数可能接受一个文件描述符（file descriptor）、或一个 socket 连接名、或另两个 stream 缓冲区（用来在对象初始化时同步写入）。若要向目标端改写数据，我们只需实作相应的 overflow() 即可。此外最好也能实作 xspn()，这样有助于提升效率。

为求方便构造 stream 缓冲区，我们也可以实作出一个特殊 class，用来将构造函数参数传递给相应的 stream 缓冲区。下例说明这一点，它定义了一个以文件描述符（file descriptor）初始化的 stream 缓冲区类别，文件描述符的字符由函数 write() 写入（write() 是 UNIX 族系操作系统使用的一个低层 I/O 函数）。另外它还定义了一个继承自 ostream 的类别，用以维护这样一个 stream 缓冲区；文件描述符就是在那儿被传递的：


```

// io/outbuf2.hpp

#include <iostream>
#include <streambuf>
#include <cstdio>

extern "C" {
    int write (int fd, const char* buf, int num);
}

class fdoutbuf : public std::streambuf {
protected:
    int fd; // file descriptor
public:
    // constructor
    fdoutbuf (int _fd) : fd(_fd) {
    }
protected:
    // write one character
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            char z = c;
            if (write (fd, &z, 1) != 1) {
                return EOF;
            }
        }
        return c;
    }
    // write multiple characters
    virtual
    std::streamsize xsputn (const char* s,
                           std::streamsize num) {
        return write(fd,s,num);
    }
};

class fdostream : public std::ostream {
protected:
    fdoutbuf buf;
public:
    fdostream (int fd) : std::ostream(0), buf(fd) {
        rdbuf(&buf);
    }
};

```

这个 stream 缓冲区也实作出函数 `xspn()`，避免发送字符序列到该 stream 缓冲区时，对每个字符都调用 `overflow()`。此函数在一次调用过程中将整个字符序列写入由文件描述符 `fd` 指定的文件内，并返回成功写入的字符数。下面是一个运用实例：

```
// io/outbuf2.cpp

#include <iostream>
#include "outbuf2.hpp"

int main()
{
    fdostream out(1); // stream with buffer writing to file descriptor 1

    out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

这个程序产生一个以文件描述符 1 初始化的 output stream。通常这样的文件描述符代表标准输出通道，所以本例只是很单纯地打印出字符。如果你有其它文件描述符（例如表示文件或 socket）可用，也可以将它作为构造函数的参数。

如果要实作出具备缓冲能力的 stream 缓冲区，write 缓冲区必须以函数 `setp()` 初始化。以下例子说明这一点：

```
// io/outbuf3.hpp

#include <cstdio>
#include <streambuf>

extern "C" {
    int write (int fd, const char* buf, int num);
}
```

```

class outbuf : public std::streambuf {
protected:
    static const int bufferSize = 10;    // size of data buffer
    char buffer[bufferSize];            // data buffer

public:
    /* constructor
     * - initialize data buffer
     * - one character less to let the bufferSizeth character
     * cause a call of overflow()
     */
    outbuf() {
        setp (buffer, buffer+(bufferSize-1));
    }

    /* destructor
     * - flush data buffer
     */
    virtual ~outbuf() {
        sync();
    }

protected:
    // flush the characters in the buffer
    int flushBuffer () {
        int num = pptr()-pbase();
        if (write (1, buffer, num) != num) {
            return EOF;
        }
        pbump (-num);    // reset put pointer accordingly
        return num;
    }

    /* buffer full
     * - write c and all previous characters
     */
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            // insert character into the buffer
            *pptr() = c;
            pbump(1);
        }

        // flush the buffer
        if (flushBuffer() == EOF) {
            // ERROR
            return EOF;
        }
        return c;
    }

```

```
    }

    /* synchronize data with file/destination
    * - flush the data in the buffer
    */
    virtual int sync () {
        if (flushBuffer() == EOF) {
            // ERROR
            return -1;
        }
        return 0;
    }
};
```

其中，构造函数运用 `setp()` 将 `write` 缓冲区初始化：

```
setp (buffer, buffer+(size-1));
```

我们所建立的这个 `write` 缓冲区，只要尚余一个字符空间，就调用 `overflow()`。如果不是以 `EOF` 为参数调用 `overflow()`，那么，由于指向改写位置的指针增加后并未超过尾端，所以字符会被写入当前位置。一旦将 `overflow()` 的参数放在正确位置上后，整个缓冲区就可以被倒空（emptied）了。

成员函数 `flushBuffer()` 正是负责此事的。它运用函数 `write()` 将字符写向标准输出通道（文件描述符 1）。`stream` 缓冲区的成员函数 `pbump()` 用来将改写位置移回缓冲区起始处。

函数 `overflow()` 将引发此一调用而且不是 `EOF` 的字符安插到缓冲区内。然后运用 `pbump()` 移动改写位置，形成“被缓冲字符”的新尾端。这样做会把改写位置临时移到结束位置（`epptr()`）之后。

这个 `class` 也实作了虚函数 `sync()`——它可以造成 `stream` 缓冲区的当前状态和相应的储存介质同步。通常，唯一需要做的就是清空（flush）缓冲区。对于无缓冲行为的 `stream` 缓冲区来说，由于没有必要清空缓冲区，所以不必改写这个虚函数。

虚拟析构函数 (virtual destructor) 可确保当 stream 缓冲区被销毁时, 缓冲区内的数据仍会被写入目标区。

对大多数 stream 缓冲区而言, 这些函数都会被改写 (overridden)。如果外部表述结构比较复杂, 改写其它函数也可能带来帮助。例如你可能会改写函数 seekoff() 和 seekpos(), 以便实现对改写位置的控制。

使用者自定义的 input 缓冲区

输入机制和输出机制的工作原理基本相同。但是对输入而言, 有可能不进行最后的读取操作。函数 sungetc() (由 Input stream 的 ungetc() 调用) 或 sputback() (由 Input stream 的 putback() 调用) 可用来储存 stream 缓冲区最后一次读取前的状态。此外也可能需要读取下一字符而不移动读取位置。所以如果你要实作出“从 stream 缓冲区中读取”操作, 那么和实作出“向 stream 缓冲区写入数据”的操作相比, 你必须改写 (override) 更多函数。

stream 缓冲区以三个指针维护一个 read 缓冲区, 这些指针可透过成员函数 eback(), gptr(), egptr() 取得 (图 13.5) :

1. eback() (意思是 "end back") 是 input 缓冲区的起始位置, 或者回退区 (putback area) 的尾端 (这是其名称的由来)。如果不采取特殊措施, 字符最多只能被回退 (putback) 到这个位置。
2. gptr() (意思是 "get pointer") 是当前的“读取位置”。
3. egptr() (意思是 "end get pointer") 是 Input 缓冲区的尾端。

读取位置和结束位置之间的字符已经从外部表述装置被传至程序内存中, 但仍然等待着程序的处理。

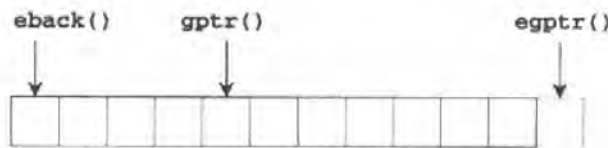


图 13.5 “从 Stream 缓冲区读取”的接口

函数 sgetc() 或 sbumpc() 可以读取单一字符。两者的差别在于后者会令“读取指针”前进, 而前者不会。如果缓冲区读取完毕 (gptr() == egptr()), 就不再有用字符了, 此时缓冲区必须重新获得补给。这项工作可由虚函数 underflow() 完成, 它会负责读取数据。如果没有可用字符, 函数 sbumpc() 会调用虚函数

`uflow()`，而 `uflow()` 的缺省行为就是调用 `underflow()`，并移动（前进）“读取指针”。基类 `basic_streambuf` 中对 `underflow()` 的缺省做法就是令它返回 EOF，意味不可能以此缺省版本读取字符。

函数 `sgetn()` 用于一次读取多个字符。这个函数把任务委派给虚函数 `xsgetn()`，后者的缺省做法是简单地对每个字符调用 `sbumpc()`。但就像改写函数 `xspn()` 一样，我们也可以改进 `xsgetn()` 以优化对多个字符的读取。

和 `output` 截然不同的是，对 `input` 而言，仅仅改写一个函数是不够的。你要么就必须建立一个缓冲区，要么就至少实作出 `underflow()` 和 `uflow()`。这是因为 `underflow()` 不会将“读取指针”移到当前字符之后，但它可以被 `sgetc()` 调用。移至下一字符，需以缓冲区操作函数或 `uflow()` 完成。无论如何，任何一个具备字符读取功能的 `stream` 缓冲区，都必须实作出 `underflow()`。如果 `underflow()` 和 `uflow()` 都实作了，就没有必要建立缓冲区。

成员函数 `setg()` 可以建立一个 `read` 缓冲区。该函数有三个参数，依次为：

1. 一个指针，指向缓冲区头部（`eback()`）。
2. 一个指针，指向当前读取位置（`gptr()`）。
3. 一个指针，指向缓冲区尾部（`egptr()`）。

和 `setp()` 不同，`setg()` 有三个参数。这是必要的，从而能定义出用来储存“将被回退（`putback`）给 `stream`”的字符空间。因此，一旦指向 `read` 缓冲区的指针已经被设定好，就会有一些（至少一个）字符已被读取但仍存放于缓冲区内。

前面已经说过，运用 `sputbackc()` 和 `sungetc()` 便可将字符回退（`putback`）到 `read` 缓冲区。`sputbackc()` 以待退字符作为参数，并确保该字符确实是被读取的字符。如果可能，这两个函数都会将读取指针退回一步。当然这只有在读取指针不指向 `read` 缓冲区头部的时候才能办到。如果到达缓冲区头部后，你还试图退一个字符，虚函数 `pbackfail()` 就会被调用。只要改写该函数，你就可以实作出“即使在这种情况下也能恢复原读取位置”的机制。基类 `basic_streambuf` 并未定义相应操作，因此实际上不可能回退任意个字符。对于不使用缓冲区的 `streams` 而言，函数 `pbackfail()` 应该被实作出来，因为这些 `streams` 通常假设至少有一个字符可被回退（`putback`）。

如果新缓冲区只用于读取，会产生另一个问题：要是缓冲区没有将旧数据保存下来，那就连一个字符也无法回退。因此，实作 `underflow()` 时经常把当前缓冲区的最后数个字符（譬如 4 个字符）移到头部，然后才添加（`append`）新读取的字符。这么一来就允许在调用 `pbackfail()` 之前回退一些字符了。

下例说明上述做法的一种可能模样。Class `inbuf` 实作出一个拥有 10 个字符空间的 Input 缓冲区, 前 4 个字符空间作为回退区域, 后 6 个字符空间作为常规的 Input 缓冲区。

```
// io/inbuf1.hpp

#include <cstdio>
#include <cstring>
#include <streambuf>

extern "C" {
    int read (int fd, char* buf, int num);
}

class inbuf : public std::streambuf {
protected:
    /* data buffer:
     * - at most, four characters in putback area plus
     * - at most, six characters in ordinary read buffer
     */
    static const int bufferSize = 10;    // size of the data buffer
    char buffer[bufferSize];            // data buffer

public:
    /* constructor
     * - initialize empty data buffer
     * - no putback area
     * => force underflow()
     */
    inbuf() {
        setg (buffer+4,    // beginning of putback area
              buffer+4,    // read position
              buffer+4);   // end position
    }

protected:
    // insert new characters into the buffer
    virtual int_type underflow () {

        // is read position before end of buffer?
        if (gpptr() < egptr()) {
            return *gpptr();
        }
    }
}
```

```

    /* process size of putback area
    * - use number of characters read
    * - but at most four
    */
    int numPutback;
    numPutback = gptr() - eback();
    if (numPutback > 4) {
        numPutback = 4;
    }

    /* copy up to four characters previously read into
    * the putback buffer (area of first four characters)
    */
    std::memcpy (buffer+(4-numPutback), gptr()-numPutback,
                  numPutback);

    // read new characters
    int num;
    num = read (0, buffer+4, bufferSize-4);
    if (num <= 0) {
        // ERROR or EOF
        return EOF;
    }

    // reset buffer pointers
    setg (buffer+(4-numPutback),    // beginning of putback area
          buffer+4,                // read position
          buffer+4+num);           // end of buffer

    // return next character
    return *gptr();
}
};

```

构造函数会初始化所有指针，这样就可以令缓冲区完全清空（图 13.6）。如果这个 `stream` 缓冲区被读取一个字符，函数 `underflow()` 会被调用，它总是被 `stream` 缓冲区用来读取下一个字符。一开始它会先检查 `input` 缓冲区内的字符，如果确认实际存在字符，就透过函数 `memcpy()` 把这些字符移到回退区（*putback area*）。因此 `input` 缓冲区至多留有最后 4 个字符。接下来运用 POSIX 的低层 I/O 函数 `read()` 从标准输入通道读取下一个字符。一旦缓冲区调整至新状态时，返回读取的第一个字符。

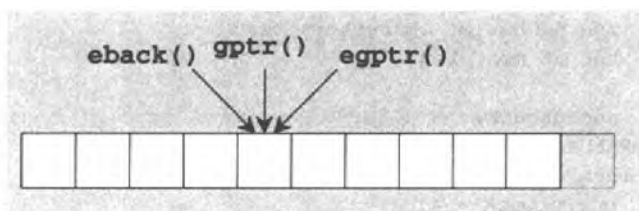


图 13.6 初始化后的缓冲区

举个例子。假设第一次调用 `read()` 读得 'H', 'a', 'l', 'l', 'o', 'w', `Input` 缓冲区的状态发生变化, 如图 13.7。因为第一次总是填入缓冲区, 所以回退区的内容是空的, 没有任何字符可被回退 (`putback`)。

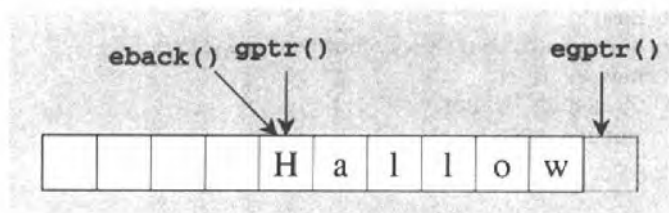


图 13.7 读入 H a l l o w 后的缓冲区

这些字符被提取后, 最后 4 个字符被移到回退区, 然后新字符又被读取。假设再次调用 `read()` 读入 'e', 'e', 'n', '\n', 结果如图 13.8。

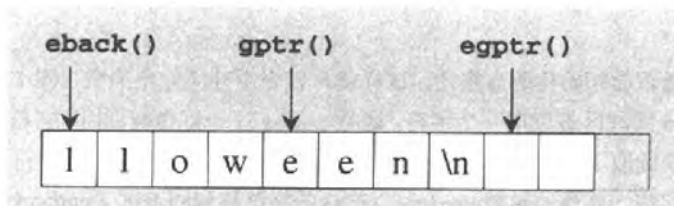


图 13.8 续读 4 个字符后的缓冲区

下面是这个 stream 缓冲区的运用实例:

```
// io/inbuf1.cpp

#include <iostream>
#include "inbuf1.hpp"

int main()
{
    inbuf ib; // create special stream buffer
    std::istream in(&ib); // initialize input stream with that buffer

    char c;
    for (int i=1; i<=20; i++) {
        // read next character (out of the buffer)
        in.get(c);

        // print that character (and flush)
        std::cout << c << std::flush;

        // after eight characters, put two characters back into the
        // stream
        if (i == 8) {
            in.unget();
            in.unget();
        }
    }
    std::cout << std::endl;
}
```

程序在循环中读取并写出字符。读得第 8 个字符后, 放回 2 个字符。所以, 第 7 个字符和第 8 个字符被打印两次。

13.14 关于性能 (Performance)

本节着重于性能 (performance) 上的讨论。一般而言 stream classes 已经是十分高效了, 但当 I/O 是程序性能的关键时, 我们还可以进一步强化这些 classes。

13.2.3 节, p593 已经就性能议题进行过讨论: 你应该只含入编译时需要的头文件。尤其当你并没有使用标准 stream 对象时, 你应该避免含入 <iostream>。

13.14.1 与 C 标准串流 (Standard Streams) 同步

缺省情况下, 八个 C++ 标准 streams (四个窄字符 stream: `cin`, `cout`, `cerr`, `clog`, 以及相应的四个宽字符版本) 和 C 标准程序库的相应文件 (`stdin`, `stdout`, `stderr`) 是同步的。`clog` 和 `wclog` 缺省采用和 `cerr`, `wcerr` 相同的 stream 缓冲区。

视实际作法不同, 同步 (synchronization) 可能带来某些不必要的额外负担。举个例子, 如果标准 C++ stream 是以标准 C 文件实作出来, 基本上就会限制相应的 stream 缓冲区的缓冲行为。然而, 对于某些优化, 尤其是对格式化读取进行的优化 (参见 13.14.2 节, p682), stream 缓冲区中的缓冲行为非常必要。C++ 标准程序库为 `ios_base` 定义了一个静态成员函数 `sync_with_stdio()` (表 13.46), 允许我们改用更好的实作法。

表 13.46 同步标准 C++ Streams 和标准 C Streams

静态函数	含义
<code>sync_with_stdio()</code>	判断标准的 stream 对象是否与标准的 C streams 同步
<code>sync_with_stdio(false)</code>	取消 C++ streams 和 C streams 的同步; 前提是必须在任何 I/O 操作前被调用

`sync_with_stdio()` 以一个可有可无的布尔值为参数, 用以决定是否和标准 C stream 同步。如果要取消同步, 可以 `false` 为参数:

```
std::ios::sync_with_stdio(false);    // disable synchronization
```

注意: 你必须在任何 I/O 操作之前取消同步。如果你在任何 I/O 发生之后才调用这个函数, 导致的行为视实作版本而定。

这个函数返回前一次被调用时的参数。如果先前不曾被调用过, 就返回 `true`, 反映出标准 stream 的默认值。

13.14.2 Stream 缓冲区内的缓冲机制

I/O 是否具备缓冲能力, 很大程度地影响了效率。原因之一是, 通常系统调用比较耗资源, 应该尽可能避免。但在 C++ 中还有许多更微妙的原因使我们至少应该在 stream 缓冲区中对 input 进行缓冲, 这些原因包括: 格式化 I/O 操作所使用的函数, 以 stream 缓冲迭代器 (buffer iterators) 来存取 stream; 而对 stream 缓冲迭代器做操作, 比对指针做操作要慢。虽然差别不大, 但也足够证明有必要对频繁的操作 (例如格式化读取) 进行改善。

因此, 所有 I/O 操作都以 stream 缓冲区完成, 后者实现了某种缓冲机制。但是仅仅依赖缓冲机制还不够, 因为高效缓冲有三个冲突因素:

1. 不采用缓冲机制来实作 stream 缓冲区, 往往更简单。如果 stream 并不常被使用, 或只用于输出, 那么缓冲机制倒也不那么重要 (对输出而言, stream 缓冲迭代器和指针之间的差异, 不如输入那么严重)。但是一旦面对大量被使用的 stream 缓冲区, 就应该毫不犹豫地实现缓冲机制。
2. 每次输出操作完成后, 标志 `unitbuf` 会提示 output streams 清空自己的 stream。操控器 `flush` 和 `endl` 也会清空各自的 stream。若为求得最佳性能, 三者都应该避免。但是 (举个例子), 当写向控制台 (console) 时, 写完一整行后还是可能需要清空 stream。如果你的程序被迫频繁使用 `unitbuf`, `flush` 或 `endl`, 就要考虑采用某种特殊的 stream 缓冲装置, 后者并不调用 `sync()` 清空 stream 缓冲区, 而是在适当时候采用其它函数。
3. 运用函数 `tie()` 绑定某个 stream (详见 13.10.1 节, p637), 也会非必要地清空 stream。因此, 非必要请勿绑定 stream。

实作一个新的 stream 缓冲区时, 我们可以先不考虑缓冲机制。然后如果该 stream 缓冲区成了效率瓶颈, 还是可以在不影响程序其它部分的情况下实现缓冲。

13.14.3 直接使用 Stream 缓冲区

`classes basic_istream` 和 `basic_ostream` 中, 所有用于读写字符的成员函数都以同样方式在运作: 首先构造相应的 `sentry` 对象, 然后执行实际操作。`sentry` 对象的构造可能导致: 暴露潜在的绑定对象、忽略空格 (对输入而言)、一些“与实作品相关”的行为 (例如多线程环境中的锁, 参见 p658, 13.12.4 页)。

对于无格式 I/O, 大多数操作通常都是无益的。如果在多线程环境中运用 stream, 只有“上锁”操作可用 (请注意, C++ Standard 并不考虑多线程议题)。因此, 进行无格式 I/O 时, 最好直接使用 stream 缓冲区。

欲支持这种行为, 你可以对 stream 缓冲区使用 `operator<<` 和 `operator>>`:

- 将指向 stream 缓冲区的指针传给 `operator<<`, 你便可以输出该 stream 内的所有输入。这可能是“透过 C++ I/O stream 来复制文件”的最快方法。例如:

```
// io/copy1.cpp
#include <iostream>

int main ()
{
    // copy all standard input to standard output
    std::cout << std::cin.rdbuf();
}
```

```
}  
// 译注：以上程序执行时，cin 获得的任何一笔输入（以 <Enter> 隔开），  
// 都会立刻在 cout 出现。
```

这里的 `rdbuf()` 取得 `cin` 的缓冲区（见 p638）。这么一来程序就将所有标准输入复制到了标准输出。

- 将指向 `stream` 缓冲区的指针传给 `operator>>`，你便可以将数据直接读进该 `stream` 缓冲区内。例如你可以通过以下方法把所有标准输入复制到标准输出：

```
// io/copy2.cpp  
  
#include <iostream>  
int main ()  
{  
    // copy all standard input to standard output  
    std::cin >> std::noskipws >> std::cout.rdbuf();  
}  
// 译注：此程序之执行情况亦如前例。
```

注意，你必须清除标志 `skipws`，否则会漏掉输入时的起始空格（见 p625）。

即使是格式化 I/O，也可以直接使用 `stream` 缓冲区。举个例子，某循环读取了许多数值，整个循环生命周期只需构造一个 `sentry` 对象就够了，于是在循环之内，便可手工略去空格（使用 `ws` 操控器也可以构造出一个 `sentry` 对象），然后可以使用 `facet::num_get`（参见 14.4.1 节, p707）直接读取数值。

注意，`stream` 缓冲区本身并无所谓错误状态，也不知道自己和哪些 `input stream` 或 `output stream` 相连接。所以下面这个式子中：

```
// copy contents of in to out  
out << in.rdbuf();
```

如果出错或到达文件尾端，没有什么办法可以改变错误状态。

14

国际化

Internationalization, i18n

随着全球市场的不断增长,国际化 (*internationalization*, 简写为 *i18n*¹) 越来越成为软件开发的重要议题。为了顺应这个潮流, C++ 标准程序库提供了编写国际化程序代码支持机制。这一机制的主要影响 I/O 的使用和字符串的处理。本章将介绍此一机制。非常感谢 Dietmar Kühl, 他是 C++ 标准程序库中有关 I/O 和国际化方面的专家, 本章大部分内容出自他的手笔。

C++ 标准程序库面对“国家习惯”的转换问题, 并非只着眼于某些特定转换, 而是提供了一个一般化方案。例如为了支持 16-bit 亚洲字符, 这个方案并不限定字符型别。对于程序国际化而言, 以下两个相关面貌很重要:

1. 不同的字符集有不同的性质。想要处理它们, 我们就得对诸如“什么才算是字母”或甚至“用什么型别来存储字符”这样的问题给出灵活的解决方案。面对字符数多于 256 个的字符集, `char` 型别很显然不够用。
2. 使用者希望程序能够针对国家和文化传统间的差异, 进行必要转化 (例如日期格式、货币书写格式、布尔值表示法等等)。

对于这两个面貌, C++ 标准程序库都提供了相关解决方案。

国际化问题的主要思路是以 *locale* (地域、本土、国别) 对象来代表一个可扩展的面貌集合 (*collection of aspects*), 以此进行地区转换工作。Locale 在 C 之中已经有所应用, 而在 C++ 标准规格书中, 它更被泛型化, 设计得更具弹性。事实上 C++ *locale* 机制可以根据用户环境或偏好进行各种专用订制, 例如可以对它进行扩展, 使它能处理度量衡系统、时区、纸张规格等问题。

¹ i18n 是 *internationalization* 的一个常见缩写, 其意义是: 字母 i 后面跟着 18 个字母, 最后再跟着一个字母 n。

大部分国际化相关机制只需程序员付出最低程度的努力,甚至根本不需任何额外付出。例如运用 C++ stream 机制进行 I/O 时,所有数值会根据 locale 规定的规则自动格式化。程序员需要做的只是要求那些 I/O stream classes 采用使用者的偏好。

除了这一类自动运用,程序员也可以使用 locale 对象直接进行格式化、校勘 (collation)、字符分类等工作。C++ 标准程序库所支持的某些面貌 (aspects) 并未为其自身所用,因此程序员必须手工调用相应函数才能运用之。例如在 C++ 标准程序库中并没有任何 stream 函数可以用来格式化时间、日期或货币表示式。想要享受这些服务,必须直接调用相应函数 (例如在用户自定的 stream 操作符中输出货币类别的对象)。

strings 和 streams 使用了国际化机制的另一个概念:字符特性 (character traits)。二者都定义出“能够把不同字符集区分开来”的某些基本特性和操作,例如 *end-of-file* 的实际值、比较函数、赋值函数、复制函数等等。

国际化机制的相关 classes 比较晚近才被引入 C++ 标准里头。尽管整体方案极为灵活,但要达到真正的完备仍然需要做一些工作。例如用于字符串校勘 (collation, 根据某些地区转换原则比较字符串的“大小”,以便进行排序) 的函数直接使用了型别为 `const charT*` 的迭代器,这里的 `charT` 代表某种字符型别。尽管 `basic_string<charT>` 通常都会使用此一型别作为迭代器型别,但并没有明文保证。因此严格来说我们不能确保字符串的迭代器能够作为此函数的合法参数。然而我们可以使用 `basic_string` 的成员函数 `data()` 来配合字符串校勘函数。

14.1 不同的字符编码 (Character Encodings)

国际化机制所关注的一个主题是如何处理不同的字符编码。这个问题主要出现在亚洲,在那儿,同一个字符集 (character set) 往往有不同的编码 (译注:例如汉字就至少有 BIG5 和 GB 两种编码)。这类问题往往伴随那些单一字符编码超过 8 bits 的情形出现。想要处理这类问题,必须采用新的思维和新的函数来进行文本处理。

14.1.1 宽字符 (Wide-Character) 和多字节 (Multibyte) 文本

面对多于 256 个字符的字符集,有两种不同的解决方案:多字节 (multibyte) 表示法和宽字符 (wide-character) 表示法。

1. 在多字节 (multibyte) 表示法中,字符所用的 byte 个数是可变的。一个 1-byte 字符 (例如 ISO Latin-1 字符) 后面可以跟着一个 3-byte 字符,例如日本表意文字 (ideogram)。(译注:或中国象形文字)
2. 宽字符 (wide-character) 表示法,字符所用的 byte 数目恒定,与所表示的字符无关。典型的个数是 2 或 4 个 bytes。观念上来说,这和那些只使用 1 byte 的表示法 (例如 ISO Latin-1 或甚至 ASCII) 没有什么区别。

多字节表示法比宽字符表示法更紧凑, 因此前者通常用来在程序外部储存数据, 而后者由于比较容易处理固定大小的字符, 所以通常在程序内部被使用。

与 ISO C 相似, ISO C++ 使用 `wchar_t` 来表示宽字符。不过在 C++ 中 `wchar_t` 已被扶正为关键词, 不再是区区一个型别定义 (typedef) 了。这样一来我们就可以使用这个型别来对所有函数进行重载。

在 `multibytes string` 中, 某个 `byte` 可能代表一个字符, 也可能只是字符的一部分。遍历 `multibytes string` 时, 每一个 `byte` 的意义都由当时的“转换状态 (shift state)”决定。根据转换状态, 一个 `byte` 可能代表一个字符, 也可能表示要更改转换状态。一个 `multibytes string` 开始时通常具有某些预先定义好的初始转换状态。例如在初始转换状态下, 可能每个 `byte` 都代表 ISO Latin-1 字符, 直到遭遇脱字符 (escape character) 为止。紧跟在脱字符之后的那个字符就用来指示新的转换状态: 它可能意味随后的 `bytes` 都应当被理解为阿拉伯字符, 直到遇到下一个脱字符为止。

`Class template codecvt<>` (参见 14.4.4 节, p720) 用来在不同的字符编码方案之间进行转换。该类别主要供 `basic_filebuf<>` (参见 p627) 在数据内部表述和外部表述之间进行转换。C++ *Standard* 并未对 `multibytes` 字符编码做任何假定, 它只是支持“转换状态”这个概念。`codecvt<>` 的成员可以运用一个参数来储存字符串的任意状态。此外它还支持一个函数, 允许你以某个字符序列将转换状态还原为初始状态。

14.1.2 字符特性 (Character Traits)

不同的字符集, 不同的表述 (representations), 这给 `string` 和 I/O 的处理带来了变数。例如, 面对不同表述, `end-of-file` 值和字符比较细节, 都可能不同。

`String` 和 `stream classes` 可以根据内建型别 (特别是 `char` 和 `wchar_t`) 完成具现化 (instantiated)。但内建型别的接口不可改变, 因而才会引进一个独立类别, 将“如何处理和表述 (representations) 相关的各个面貌 (aspects)”的相关细节统统放入该类别中, 这就是大名鼎鼎的 **traits class** (特性类别)。`String` 和 `stream classes` 都将 **traits class** 当做 `template` 参数: 此一参数缺省为 `char_traits`, 并以 `string` 或 `stream` 获得之 `template` 参数 (用来定义字符型别) 为参数:

```
namespace std {
    template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
        class basic_string;
}
```



```

namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_istream;

    template <class charT,
              class traits = char_traits<charT> >
    class basic_ostream;
    ...
}

```

字符特性 (character traits) 的型别为 `char_traits<>`。这个型别在 `<string>` 中定义, 以字符型别 (character type) 为参数:

```

namespace std {
    template <class charT>
    struct char_traits {
        ...
    };
}

```

`traits class` 定义了字符型别的所有基本属性, 并以静态组件的形式定义出实作 `strings` 和 `streams` 时不可或缺的相关操作。表 14.1 列出 `char_traits` 的所有成员。

字符串和字符序列的处理函数, 其存在价值完全是为了效率优化。它们其实完全可以运用“只处理单个字符”的函数实作出来。例如 `copy()` 可以运用 `assign()` 实作出来, 然而在处理字符串时, `copy()` 可能会采用更有效率的实作技术。

注意, 函数中使用的计数 (counts) 是确切的计数值, 不是最大计数值。也就是说序列的终止字符 (termination characters) 将被忽略。

最后一组函数专门对 *end-of-file* (EOF) 字符进行特殊处理。这个字符通过人工字符扩充了字符集, 用以指示某种特殊的处理。对某些表述来说, 字符型别不足以容纳这个特殊字符, 因为 EOF 字符必须使用字符集中一般字符以外的值。C 建立了一种转换模式: 令字符读取函数返回一个 `int`, 而不是一个 `char`。这一技巧被 C++ 发扬光大。字符特性类别 (Character traits class) 定义了 `char_type` 型别用以代表所有字符型别, 定义了 `int_type` 用以代表“所有字符, 外加 EOF”的型别。函数 `to_char_type()`, `to_int_type()`, `not_eof()`, `eq_int_type()` 定义了相应的转换和比较操作。当然, 对于某些字符特性, `char_type` 和 `int_type` 可能相同。例如, 如果 `char_type` 中并非所有的值都必须拿来表示字符, 那么便一定会有多余的值可以拿来代表 EOF。

`pos_type` 和 `off_type` 用来定义文件位置和偏移距离 (详见 p634)。

表 14.1 字符特性 (Character Traits) 类别成员

表达式	意义
char_type	字符型别 (即 char_traits 的 template 参数)
int_type	足以容纳 “附加之 end-of-file 值” 的型别
pos_type	此型别用以表现 “streams 内的位置”
off_type	此型别用以表现 “streams 内的两个位置之间的间距”
state_type	此型别用以表现 “multibytes stream 的当前状态”
assign(c1,c2)	将字符 c2 赋值给 c1
eq(c1,c2)	判断字符 c1 和 c2 是否相等
lt(c1,c2)	判断字符 c1 是否小于字符 c2
length(s)	返回字符串 s 的长度
compare(s1,s2,n)	比较字符串 s1 和 s2 的前 n 个字符
copy(s1,s2,n)	将字符串 s2 的前 n 个字符复制到 s1
move(s1,s2,n)	将字符串 s2 的前 n 个字符复制到 s1, s1 和 s2 可重迭
assign(s,n,c)	将字符 c 赋值给字符串 s 的前 n 个字符
find(s,n,c)	在字符串 s 中搜寻第一个与 c 相等的字符, 返回一个指针指向它。如果 s 的前 n 个字符中没有找到它, 就返回零
eof()	返回 end-of-file 值
to_int_type(c)	将字符 c 转换成 “型别为 int_type” 的相应值
to_char_type(i)	将型别为 int_type 的 i 转换为字符 (如果转换 EOF, 会导致未定义行为)
not_eof(i)	如果 i 不是 EOF, 返回 i; 如果 i 是 EOF, 返回一个由实作版本所定义的、不同于 EOF 的值
eq_int_type(i1,i2)	检查字符 i1 和 i2 的 int_type 对应值是否相等 (字符可以是 EOF)

C++ 标准程序库将 char_traits<> 型别针对 char 和 wchar_t 进行了特化设计:

```
namespace std {
    template<> struct char_traits<char>;
    template<> struct char_traits<wchar_t>;
}
```

针对 char 而设计的特化版本, 通常在实作时会运用定义于 <cstring> 或 <string.h> 内的 C 全局函数。下面是一个可行的例子:

```
namespace std {
    template<> struct char_traits<char> {
        // type definitions:
        typedef char          char_type;
```

```
typedef int          int_type;
typedef streampos    pos_type;
typedef streamoff    off_type;
typedef mbstate_t    state_type;

// functions:
static void assign(char& c1, const char& c2) {
    c1 = c2;
}
static bool eq(const char& c1, const char& c2) {
    return c1 == c2;
}
static bool lt(const char& c1, const char& c2) {
    return c1 < c2;
}
static size_t length(const char* s) {
    return strlen(s);
}
static int compare(const char* s1, const char* s2, size_t n)
{
    return memcmp(s1,s2,n);
}
static char* copy(char* s1, const char* s2, size_t n) {
    return (char*)memcpy(s1,s2,n);
}
static char* move(char* s1, const char* s2, size_t n) {
    return (char*)memmove(s1,s2,n);
}
static char* assign(char* s, size_t n, char c) {
    return (char*)memset(s,c,n);
}
static const char* find(const char* s, size_t n, const char& c)
{
    return (const char*)memchr(s,c,n);
}
static int eof() {
    return EOF;
}
static int to_int_type(const char& c) {
    return (int)(unsigned char)c;
}
```

```

        static char to_char_type(const int& i) {
            return (char)i;
        }
        static int not_eof(const int& i) {
            return i!=EOF ? i : !EOF;
        }
        static bool eq_int_type(const int& i1, const int& i2) {
            return i1 == i2;
        }
    };
}

```

至于如何让字符串表现出“大小写无关”的行为，请看 11.2.14 节，p503 所展示的一个使用者自定的特性类别 (user-defined traits class)。

14.1.3 特殊字符国际化

还有一个问题没解决：如何处理特殊字符，例如换行符号 (newline) 或字符串终止符号的国际化？class basic_ios 的成员函数 widen(), narrow() 可用来对付这个问题。例如面对 stream strm，可以将换行符号这么编码：

```
strm.widen('\n')    // internationalized newline character
```

同样的，字符串终止符号可以编码如下：

```
strm.widen('\0')    // internationalized string termination character
```

运用实例可参考 p613 的 endl 操控器 (manipulator) 之实作。

函数 widen() 和 narrow() 确实使用了一个 locale 对象，确切地说是该对象的 ctype facet。这个 facet 用来对所有字符“在 char 和其它表现形式之间”进行转换，具体描述于 14.4.4 节，p716。例如以下表达式使用 locale 对象 loc 将一个 char 型别的字符 c 转化为一个 char_type 型别的对象²：

```
std::use_facet< std::ctype<char_type> >(loc).widen(c)
```

Locales 及其 facets 的使用细节随后便会提到。

² 注意，你必须在两个 ">" 之间插入一个空格。">>" 会被视为移位操作符 (shift operator)，导致语法错误。

14.2 Locales 的概念

解决国际化问题,通常是通过一个被称为 *locales* 的环境,它被用来封装国家(地域)和文化之间的转换行为。C 正是使用这种做法。在国际化背景中,一个 *locale* 就是一个“参数和函数的集合”,用以进行国家和文化间的转换。根据 X/Open 公约³,环境变量 *LANG* 用来确定当前使用的 *locale*。不同的浮点数、日期、货币格式则根据这个 *locale* 确定。

定义一个 *locale*,需采用以下字符串格式:

```
language[_area[.code]]
```

language 表示语言,例如英语或德语。*area* 表示该语言所处的地域、国家或文化;借助它,便可以在相同语言的不同国家之间实施转换。*code* 定义出字符编码方案,其重要性在亚洲地区特别突显,因为在那里,相同的字符集常常有不同的字符编码方案。

表 14.2 选列出典型的语言名称。请注意它们尚未标准化,例如有时候 *language* 名称的首字母大写。此外某些实作版本和表 14.2 的格式不符(例如以 *english* 来选择一个英语 *locale*)。总而言之,系统所支持的 *locales* 视实作版本而定。

对程序而言,这些名称是否标准化无关紧要,因为 *locale* 的信息是由使用者以某种形式提供的。普遍的做法是,程序只需读取环境变量或类似的数据库,判断可用的 *locales*,然后便可把“选择正确 *locales* 名称”的任务交给使用者。只有那些永远只使用某个特定 *locale* 的程序,才需要把其名称编死在程序中。通常,对于这种情况,C *locale* 已经足够,而且保证能获得所有冠以 C 名号的编译器的支持。

下一节展示如何在 C++ 程序中使用不同的 *locales*。更具体地说,下一节将介绍 *locales facets*,它是用来应对特殊格式化细节的机制。

C 同时还提供了一种方案,用来解决字符数多于 256 个的大字符集问题。该方案使用一个字符型别 *wchar_t*,并被定义为某种整数型别,可支持使用“宽字符常量”和“宽字符串字面常量(string literals)”的语言。不过除此之外,C 就只提供在宽字符和窄字符之间的转换函数了。这一方案可在 C++ 中运用,型别 *wchar_t* 是 C++ 的正牌内建型别,不像 C 那样只是一个 *typedef*。C++ 提供的程序库也比 C 强大得多,因为基本上任何一个对 *char* 有效的东西对于 *wchar_t* 以及任何其它可作为字符型别的型别也都有效。

³ POSIX 和 X/Open 都是操作系统的接口标准。

表 14.2 Locale 名称选列

Locale	意义
C	缺省: ANSI-C 公约 (英语, 7 bit)
de_DE	德语, 德国
de_DE.88591	德语, 德国, 配合 ISO Latin-1 编码
de_AT	德语, 奥地利
de_CH	德语, 瑞士
en_US	英语, 美国
en_GB	英语, 大不列颠
en_AU	英语, 澳洲
en_CA	英语, 加拿大
fr_FR	法语, 法国
fr_CH	法语, 瑞士
fr_CA	法语, 加拿大
ja_JP.jis	日语, 日本, 配合 JIS (日本工业标准) 编码
ja_JP.sjis	日语, 日本, 配合 Shift JIS 编码
ja_JP.ujis	日语, 日本, 配合 UNIXized JIS 编码
ja_JP.EUC	日语, 日本, 配合 Extended UNIX Code 编码
ko_KR	韩语, 韩国
zh_CN	汉语, 中国
zh_TW	汉语, 中国台湾
lt_LN.bit7	ISO 拉丁语, 7 bit
lt_LN.bit8	ISO 拉丁语, 8 bit
POSIX	POSIX 公约 (英语, 7 bit)

14.2.1 运用 Locales

对真正的国际化来说, 仅仅翻译“文字所带的信息”通常是不够的。各种不同的数值、货币、日期……的规格也都必须遵守。另外, 用来操作字母 (letters) 的函数, 应该根据字符 (characters) 进行编码 (encoding), 以确保正确处理特定语言中所有身为字母的字符。

根据 POSIX 和 X/Open 标准, C 程序可使用函数 `setlocale()` 来设定一个 locale。改变 locale 会对 `isupper()` 和 `toupper()` 之类的字符分类/操作函数以及 `printf()` 之类的 I/O 函数产生影响。

然而 C 的解决方案毕竟有诸多限制。由于 locale 是全局属性, 所以同时使用一个以上的 locale (例如按英文规则读取浮点数, 按德文规则写出), 即使不是不可能, 也得费尽九牛二虎之力才有希望。此外 locales 不能扩展, 只能提供“由编译器

选择供应”的设施。如果某个必须遵守的国家协议未被 C locales 支持, 就只好改弦更张了。最后一点, 我们根本不可能为了支持特殊文化而定义新的 locales。

C++ 标准程序库利用面向对象方式解决了上述所有问题。首先, “与 locale 相关的细节”被封装在型别为 locale 的对象中。仅仅如此这般, 在同一时刻运用多个 locales 的美梦便立刻成真。与 locales 相依的各种操作, 将运用相应的 locale 对象。例如我们可以把一个 locale 对象安装到每一个 I/O stream 中, 后者的各成员函数便利用该对象迎合相应规格。请见下例:

```
// i18n/loc1.cpp

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

int main()
{
    // use classic C locale to read data from standard input
    cin.imbue(locale::classic());

    // use a German locale to write data to standard output
    cout.imbue(locale("de_DE"));

    // read and output floating-point values in a loop
    double value;
    while (cin >> value) {
        cout << value << endl;
    }
}
```

其中的语句:

```
cin.imbue(locale::classic());
```

将经典的 C locale 赋值到标准输入通道中。所谓经典的 C locale, 其实就是没有任何 locale, 而是使用原原本本的 C 形式来格式化数字、日期、字符分类等工作。以下表达式:

```
std::locale::classic()
```

可获取经典 locale 的对应对象。以下表达式:

```
std::locale("C")
```

效果相同。这个表达式根据一个给定的名字来产生一个 locale 对象。“C”是一个特殊名称, 它实际上是各个 C++ 实作版本唯一必须支持的名称。C++ Standard 并未强制要求支持其它 locale, 不过通常各个 C++ 实作版本都会支持其它 locales。

以下语句:

```
cout.imbue (locale("de_DE"));
```

会将 locale de_DE 赋值到标准输出通道中。唯有当系统支持该 locale, 这个动作才可能成功。如果你所希望构造的 locale 的名字不能被 C++ 实现版本识别出来, 它会抛出一个 runtime_error 异常。

如果一切顺利, 上例输入时将按照经典 C 规格, 输出时将按照德国规格。如此一来就可以按英文格式读取浮点数如:

```
47.11
```

然后按德文规格输出:

```
47,11
```

没错, 德国人确实以逗号当做“小数点”。

一般来说, 除非需要按照某个固定格式来读写数据, 否则程序不会预先定义一个特定的 locale, 而是会利用环境变量 LANG 来确定相应的 locale。另一种可能是读取一个 locale 名称然后运用之。请看下面实例:

```
// i18n/loc2.cpp

#include <iostream>
#include <locale>
#include <string>
#include <cstdlib>
using namespace std;

int main()
{
    // create the default locale from the user's environment
    locale langLocale("");

    // and assign it to the standard output channel
    cout.imbue(langLocale);

    // process the name of the locale
    bool isGerman;
    if (langLocale.name() == "de_DE" ||
        langLocale.name() == "de" ||
        langLocale.name() == "german") {
        isGerman = true;
    }
}
```



```
    else {
        isGerman = false;
    }

    // read locale for the input
    if (isGerman) {
        cout << "Sprachumgebung fuer Eingaben: ";
    }
    else {
        cout << "Locale for input: ";
    }
    string s;
    cin >> s;
    if (!cin) {
        if (isGerman) {
            cerr << "FEHLER beim Einlesen der Sprachumgebung"
                 << endl;
        }
        else {
            cerr << "ERROR while reading the locale" << endl;
        }
        return EXIT_FAILURE;
    }
    locale cinLocale(s.c_str());

    // and assign it to the standard input channel
    cin.imbue(cinLocale);

    // read and output floating-point values in a loop
    double value;
    while (cin >> value) {
        cout << value << endl;
    }
}
```

本例之中，以下语句生成一个 locale 对象：

```
locale langLocale("");
```

其中的空字符串具有特殊含义：使用客户环境中的缺省 locale（通常由环境变量 LANG 定义）。该 locale 被以下语句赋值给标准输出装置：

```
cout.imbue(langLocale);
```

至于以下表达式：

```
langLocale.name()
```

可从缺省 locale 中获得名称——一个型别为 string（参见第 11 章）的对象。

下面各语句从标准输入装置中读取一个名字，然后产生相应的 locale：

```
string s;
cin >> s;
...
locale cinLocale(s.c_str());
```

这么一来，从标准输入装置读取的字词，就成为 locale 构造函数的参数。如果读取失败，input stream 的 ios_base::failbit 会设立起来，本程序进行以下检验和处理：

```
if (!cin) {
    if (isGerman) {
        cerr << "FEHLER beim Einlesen der Sprachumgebung"
              << endl;
    }
    else {
        cerr << "ERROR while reading the locale" << endl;
    }
    return EXIT_FAILURE;
}
```

因此如果读入的字符串对于“产生一个 locale”来说不合法，便会如上所述地抛出 runtime_error 异常。

如果想让程序遵循各地习惯，就应该使用相应的 locale 对象。Class locale 的静态成员函数 global() 可以安装一个全局的 locale 对象。这个 locale 对象可用来作为某函数的 locale 对象参数的缺省参数。如果 global() 函数所设定的 locale 对象是有名称的，C locale 相关函数也会作出相应的响应；如果该对象没有名称，则 C 函数的执行结果由实作版本决定。

以下实例展示如何根据程序运行的环境，设定全局 locale 对象：

```
/* create a locale object depending on the program's environment and
 * set it as the global object
 */
std::locale::global(std::locale(""));
```

别的不说, 以上对于后继将被执行起来的 C 函数会作出相应的注册操作。也就是说那些 C 函数所受到的影响, 和以下调用操作所带来的影响相同:

```
std::setlocale(LC_ALL, "");
```

不过, 设定全局 locale, 并不能替换掉已经储存于对象内的 locales。它只能改变由缺省构造函数所产生的 locale 对象。例如 stream 对象所存储的 locale 对象就不会被 locale::global() 替换掉。如果你希望一个已经存在的 stream 使用某个特定的 locale, 你必须使用 imbue() 函数告诉那个 stream。

当缺省构造函数构造一个 locale 对象时, 全局 locale 就发挥作用了。此时产生的新 locale 就是全局 locale 的副本。下面三行代码为三个标准的 streams 安装缺省的 locale:

```
// register global locale object for streams
std::cin.imbue(std::locale());
std::cout.imbue(std::locale());
std::cerr.imbue(std::locale());
```

在 C++ 中使用 locales, 切记 C++ locale 机制和 C locale 机制之间只有很松散的耦合 (coupling) 关系。其实只有一处相连: 如果一个具名的 C++ locale 对象被设为全局 locale, 则全局 C locale 将被改动。一般来说, 你不应该假设 C 和 C++ 函数所操作的 locales 一致。

14.2.2 Locale Facets

国家内部约定俗成的具体项目被划分为数个不同的面貌 (aspect), 分别由相应的对象处理。处理“国际化议题中的某一特定面貌”的对象, 我们称为一个 facet。locale 对象就是扮演 facets 的容器。要存取 locale 的某个面貌, 可以相应的 facet 型别作为索引。将 facet 当做 template 参数, 明白传给 template 函数 use_facet(), 便可取用特定的 facet。例如以下式子:

```
std::use_facet< std::num_punct<char> > (loc)
```

便可取得 locale loc 所辖的“字符型别为 char”的 facet num_punct。每一个 facet 都由“定义某些特定服务”的类别定义出来。例如 facet num_punct 为格式化数值和布尔值提供服务。以下表达式返回 locale loc 中用以表示 true 的字符串:

```
std::use_facet< std::num_punct<char> > (loc).truename()
```

表 14.3 大致描述了 C++ 标准程序库中预先定义好的一些 facets。每个 facet 都和一个类别关联。这些类别在 locales 的某些构造函数中用来组合其它 locales, 产生新的 locales。

表 14.3 C++ 标准程序库预先定义好的一些 facets 型别

分类	Facet 型别	用于
numeric	num_get<>()	数值输入
	num_put<>()	数值输出
	numpunct<>()	数值 I/O 中用到的符号
time	time_get<>()	时间和日期输入
	time_put<>()	时间和日期输出
monetary	money_get<>()	货币输入
	money_put<>()	货币输出
	moneypunct<>()	货币 I/O 中用到的符号
ctype	ctype<>()	字符信息 (toupper(), isupper())
	codecvt<>()	在不同字符编码之间进行转换
collate	collate<>()	字符串校勘 (collation)
messages	messages<>()	获取字符信息

你也可以定义自己的 facets 来创建特定的 locales。下例展示实际做法，定义一个 facet，采用德国人对于布尔值的表示法：

```
class germanBoolNames : public std::num_punct_byname<char> {
public:
    germanBoolNames (const char *name)
        : std::num_punct_byname<char>(name) {}
protected:
    virtual std::string do_truename () const {
        return "wahr";
    }
    virtual std::string do_falsename () const {
        return "falsch";
    }
};
```

class germanBoolNames 系从 class num_punct_byname 中派生出来，后者由 C++ 标准程序库定义。此类别用来定义 locale 中的数值格式化标点属性。从 num_punct_byname（而不是 num_punct）派生出来，使你可以对那些未被显式改写（overridden）的成员进行定制。这些成员返回的值仍然取决于构造函数参数所使用的名字。如果以 num_punct 为基类，则这些函数的行为都是固定的。不过 class germanBoolNames 必须改写两个函数，以便确定 true 和 false 的文本表示形式。

想要在某个 locale 中使用上述 facet, 你需要以 class locale 的一个特殊构造函数产生出一个新的 locale。该构造函数的第一参数是个 locale 对象, 第二参数是个指向 facet 的指针。产生出来的 locale 和第一参数所给的 locale 对象基本相同, 唯一不同的是第二参数所指定的那个 facet。第一参数被复制后, 该 facet 会被安装到新产生的 locale 对象:

```
std::locale loc (std::locale(""), new germanBoolNames(""));
```

其中 new 表达式生成一个 facet, 后者被安装到新的 locale 中。也就是说这里的 loc 是 locale("") 的一个变体。由于 locales 是不可变的 (*immutable*), 所以如果想要在 locale 之内安装一个新的 facet, 你必须首先产生一个新的 locale, 你可以像使用其它 locale 对象一样地使用它。例如:

```
std::cout.imbue(loc);  
std::cout << std::boolalpha << true << std::endl;
```

输出如下:

```
wahr
```

你也可以产生一个全新的 facet, 并以 has_facet() 检查它是否已被注册于特别的 locale 对象中。

14.3 Locales 细部讨论

C++ locale 是一个不可变的 (*immutable*) facets 容器。定义于 <locale> 头文件内:

```
namespace std {  
    class locale {  
    public:  
        // global locale objects  
        static const locale& classic();           // classic C locale  
        static locale global(const locale&);     // set global locale  
  
        // internal types and values  
        class facet;  
        class id;  
        typedef int category;  
        static const category none, numeric, time, monetary,  
                               ctype, collate, messages, all;  
  
        // constructors  
        locale() throw();  
        explicit locale (const char* name);  
    }  
};
```

```

// create locale based on other locales
locale (const locale& loc) throw();
locale (const locale& loc, const char* name, category);
template <class Facet>
    locale (const locale& loc, Facet* fp);
locale (const locale& loc, const locale& loc2, category);

// assignment operator
const locale& operator= (const locale& loc) throw();
template <class Facet>
    locale combine (const locale& loc);

// destructor
~locale() throw();

// name (if any)
basic_string<char> name() const;

// comparisons
bool operator== (const locale& loc) const;
bool operator!= (const locale& loc) const;

// sorting of strings
template <class charT, class Traits, class Allocator>
    bool operator() (
        const basic_string<charT,Traits,Allocator>& s1,
        const basic_string<charT,Traits,Allocator>& s2) const;
};

// facet access
template <class Facet>
    const Facet& use_facet (const locale&);
template <class Facet>
    bool has_facet (const locale&) throw();
}

```

关于 locales 有一桩怪事，就是容器对象的存取方式。locale 内含的 facet，是以 facet 型别为索引来进行存取。由于每一个 facet 都有不同的界面，用于不同的目的，所以我们期望 locale 的存取函数返回对应于索引的一个型别。以 facet 型别作为索引还有一个额外好处，那就是得以拥有一个型别安全的接口。

Locales 是不可变的。也就是说存储于 locale 之内的 facets 不可以被改变（除非 locales 被赋新值）。现有的 locales 和 facets 组合起来，可产生新的 locale 变体。表 14.4 列出了 locales 的所有构造函数。

表 14.4 Locales 的构造

表达式	功能
locale()	产生一个当前全局 locale 的副本
locale(name)	根据名称产生出一个 locale
locale(loc)	产生 locale loc 的副本
locale(loc1, loc2, cat)	产生 locale loc1 的一个副本，类别 cat 中所有的 facets 将被 locale loc2 的 facets 替换
locale(loc, name, cat)	此动作等同于 locale(loc, locale(name), cat)
locale(loc, fp)	产生 locale loc 的一个副本，并安装 fp 所指的 facet
loc1 = loc2	将 loc2 赋值给 loc1
loc1.template combine<F>(loc2)	产生 locale loc1 的一个副本，并将 loc2 中型别为 F 的 facet 装入

几乎所有构造函数都产生某个 locale 副本。复制 locale 是一个很廉价的操作，基本上只是设定一个指针并将引用计数（reference count）累加 1 而已。产生一个变化过的 locale 可就昂贵得多，因为必须调整 locale 内的每个 facet 的引用计数值。虽然 C++ Standard 并没有对此类操作的效率做出任何保证，但所有实作版本大概都优先照顾 locales 的拷贝效率。

表 14.4 有两个构造函数需要 locales 名称。除了 "c" 这个名称，其它名称都没有标准规定。不过 C++ Standard 要求每一个 C++ 标准程序库附带的说明文件中必须列出可接受的名称。基本上我们可以假设大部分实作版本都会接收 14.2 节, p692 所列出的那些名字。

这里还需对成员函数 combine() 作一番解释，因为它运用了一个最近才被实作出来的编译器特性：一个“带有显式指定之 template 参数”的 member function template。这意味该 template 参数并非由另一个参数隐式推导而来。这么做的原因是根本就没有参数可供推导之用。于是该 template 参数就直接被显式指定了（也就是这里的型别 F）。

有两个函数用来存取 locale 对象内的 facets（表 14.5），它们也采用相同的技术。不同之处在于，这两个函数都是全局的 template 函数，所以没有必要加上关键词 template，也就不显得那么丑陋了。

函数 use_facet() 会返回一个 reference（指向 facet），其型别是显式传递的 template 参数型别。如果传入的 locale 并不包含相应的 facet，则函数抛出 bad_cast 异常。函数 has_facet() 可用来检查某个 locale 中是否包含一个特定的 facet。

表 14.5 facets 的存取

表达式	功能
<code>has_facet<F>(loc)</code>	如果 locale <code>loc</code> 中有一个型别为 <code>F</code> 的 facet, 则返回 <code>true</code>
<code>use_facet<F>(loc)</code>	返回一个 reference, 指向 locale <code>loc</code> 中型别为 <code>F</code> 的 facet

locales 的其它操作列于表 14.6。如果某个 locale 是从某个名称或若干个具名的 locales 构造而来, 名称将被保存起来。如果参与构造的 locales 有两个以上, 则 C++ *Standard* 并未对构造出来的名称给予任何保证。如果一个 locale 是另一个 locale 的副本, 或两个 locale 名字相同, 则两个 locales 被认为相同。前者很好理解, 但为什么名字相同就是实体相同呢? 因为 locale 的名称反映出被用来创建“具名 facets”的名称。例如 locales 的名称可能就是由各个 facets 名称以某种特定顺序连接而成, 各个名称之间以分隔符隔开。运用这种体制, 两个 locales 如果由具有相同名称的 facets 参与构造, 生成的新名称也会相同。换言之, 由于 C++ *Standard* 本质上要求“由相同名称之 facets 组合而成”的 locales 都相同, 所以才有可能以精心遴选出的名称支持上述这种“相等性”思维。

表 14.6 Locales 的操作函数

表达式	功能
<code>loc.name()</code>	返回 locale <code>loc</code> 的名称字符串
<code>loc1 == loc2</code>	如果 <code>loc1</code> 和 <code>loc2</code> 是相同的 locale, 返回 <code>true</code>
<code>loc1 != loc2</code>	如果 <code>loc1</code> 和 <code>loc2</code> 是不同的 locale, 返回 <code>true</code>
<code>loc(str1, str2)</code>	返回布尔值, 显示字符串 <code>str1</code> 和 <code>str2</code> 的比较结果 (<code>str1</code> 是否小于 <code>str2</code>)
<code>locale::classic()</code>	返回 <code>locale("C")</code>
<code>locale::global(loc)</code>	将 <code>loc</code> 安装为全局 locale, 并返回上一个全局 locale

小括号操作符 `()` 使我们得以运用 locale 对象作为字符串比较工具。此操作符运用 `collate facet`, 按序比较参数所传递的字符串, 也就是说它判断在该 locale 对象的掌控下第一个字符串是否小于第二个字符串。这正是 STL 函数对象 (仿函数) 的行为 (8.1 节, p293), 所以你可以采用 locale 对象作为 STL 算法所需的字符串排序规则。例如你可以运用 German locale 所定义的字符串校勘规则, 对一个容纳字符串的 vector 进行排序:

```
std::vector<std::string> v;
...
// sort strings according to the German locale
std::sort (v.begin(), v.end(), // range
           locale("de_DE")); // sorting criterion
```


14.4 Facets 细部讨论

locales 的重要面貌就是它所包容的那些 facets。所有 locales 都必须包容某些标准 facets。下面数节将讨论数种 facets，分别对应于 C++ *Standard* 要求一定要被具现化 (instantiated) 的各个 facets。除了这些 facets，任何 C++ 标准程序库实作版本都可以在 locales 之中附加其它 facets。最重要的一点是，用户自己也可以安装属于他自己的 facets，或替换标准 facets。

14.2.2 节, p698 曾经讨论过如何为 locale 安装一个 facet。举个例子，class `germanBoolNames` 派生自 `num_punct_byname<char>` (一个标准 facet)，它利用构造函数安装到一个 locale 中，该构造函数接受一个 locale 和一个 facet 作为参数。但是我们该如何产生自己的 facet 呢？任何一个类别 `F`，只要符合以下两个要求，就可以作为 facet：

1. `F` 以 `public` 形式派生自 `locale::facet` 类别。后者 (基类) 主要定义了一些机制，供 locale 对象内部的引用计数器运用。此外还将 `copy` 构造函数和 `assignment` 操作符声明为 `private`，禁止外界拷贝 facets，也禁止外界对 facets 进行赋值。
2. `F` 必须拥有一个型别为 `locale::id` 的 `public static member`，名为 `id`。此成员用来“以某个 facet 型别为索引，搜寻 locale 之内的一个 facet”。之所以要以型别为索引，全是因为要保证型别安全之故，其实内部还是一个普通容器，以整数为索引管理 facets。

标准 facets 不单遵循上述要求，而且还遵循一些特殊实作方针。尽管并未强制要求必须遵循这些方针，但如果能够做到，还是好处多多。这些原则包括：

1. 所有成员函数均声明为 `const`。这一点很有必要，因为 `use_facet()` 返回一个 `reference` 指向 `const facet`。如果某个成员函数不声明为 `const`，就会失去被调用的可能。
2. 所有 `public` 函数都不是虚函数，而是将调用操作委托给一个 `protected` 虚函数，后者的名称类似对应的 `public` 函数，只不过前面加上 `do_`。例如 `num_punct::truename()` 会调用 `num_punct::do_truename()`。此种风格得以在试图改写 (overriding) 某个虚函数时避免遮掩 (hiding) 了同名成员函数 (例如 class `num_put` 就有好几个成员函数都叫做 `put()`)。这种作法也使得基类的设计者有机会在非虚函数中撰写一些“即使对应之虚函数被改写，也一定会被执行到”的程序代码。

下面对于标准 facets 的描述仅限于 `public` 函数。如欲改变某个 facet，你应该改写其对应的 `protected` 函数。如果你定义了与 `public facet` 函数接口相同的函数，它们就只能被重载了 (overloaded)，毕竟这些函数不是虚函数。

大部分标准 facets 都定义一个 “_byname” 版本。这个版本派生自标准 facet，被用来“根据相应之 locale 名字”产生出相应的 facet 具体体 (instantiation)。例如 class `num_punct_byname` 被用来针对一个具名的 locale 产生一个 `num_punct facet`。因此一个 `German num_punct facet` 可以如下产生：

```
std::num_punct_byname("de_DE")
```

这些 `_byname` 类别在 `locale` 的一个“以名称为参数”的构造函数中被当做内部运用。每一个标准 facets 都有一个对应名称，相应的 `_byname` 类别可用来构造该 facet 的一个实体。

14.4.1 数值格式化

数值格式在“数值的内部表述”和“相应的文本表述”之间进行转化。`IOStream` 操作符将实际转换工作委托给 `locale::numeric` 类型 (category) 中的 facets 完成。这一类型由三个 facets 组成：

1. `num_punct`，处理和数值格式化及数值解析相关的标点符号。
2. `num_put`，处理数值格式化 (numeric formatting)。
3. `num_get`，处理数值解析 (numeric parsing)。

13.7 节, p615 已经简短介绍过“以 `num_put` 进行数值格式化”和“以 `num_get` 进行相应字符串解析”的相关知识了。`num_punct` facet 提供的是 `streams` 接口未能直接支持的一些灵活性。

数值标点符号 (Numeric Punctuation)

`num_punct` facet 用来控制小数点、可有可无的千位分割符号、布尔值的文本表示符号。表 14.7 列出 `num_punct` 的成员。

表 14.7 `num_punct` facet 的成员

表达式	功能
<code>np.decimal_point()</code>	返回一个字符，用来表示小数点
<code>np.thousands_sep()</code>	返回一个字符，用来表示千位符号
<code>np.grouping()</code>	返回一个 string，描述千位符号的设置位置
<code>np.truename()</code>	返回 true 的文本 (text) 表示
<code>np.falsename()</code>	返回 false 的文本 (text) 表示

`num_punct` 以一个字符型别 `charT` 作为 `template` 参数。`decimal_point()` 和 `thousand_sep()` 返回的字符型别就是 `charT`。`truename()` 和 `falsename()` 返回一个 `basic_string<charT>`。C++ Standard 要求，`num_punct<char>` 和 `num_punct<wchar_t>` 的具现体必须存在。

如果没有间隔符号，长数字很难阅读。用于数值格式化和数值解析的那个标准 facets，也支持千位分割符号。通常整数的阿拉伯数字三个三个一组，例如一百万应该这么写：

1,000,000

遗憾的是，并非所有地区都这么写。例如德国就不使用逗号，而是使用点号。德国人的一百万是这么写的：

1.000.000

这个差异可由 `thousands_sep()` 应付。但这还不够，因为在某些国家，数字并不以三三分组。例如尼泊尔人会这么写：

10.00.000

其中每一组阿拉伯数字的个数甚至都不尽相同。这就是函数 `grouping()` 返回的字符串发挥效力的地方了。索引 `i` 处的数值代表第 `i` 组的数字个数；从最右边数起；索引计数从 0 开始。如果字符串中的字符数量少于每一组组中的字符数量，最后被确定的那一组内的字符数量将重复。如果要创造无限大组，你应当使用 `numeric_limits<char>::max()` 的返回值。如果根本就不分组，那么干脆就给一个空字符串。表 14.8 列出了一百万的格式化实例。

表 14.8 一百万的各种表示法

字符串	结果
{ 0 } 或 "" (这是 <code>grouping()</code> 的缺省结果)	1000000
{ 3, 0 } 或 "\3"	1,000,000
{ 3, 2, 3, 0 } 或 "\3\2\3"	10,00,000
{ 2, CHAR_MAX, 0 }	10000,00

注意，直接使用数字可能是不对的。例如字符串 "2" 指定的是一个 50 个阿拉伯数字的数字分组，因为在 ASCII 编码中，字符 '2' 就是整数 50。

数值的格式化

`num_put` facet 用来对数值的“表述文本”进行格式化。它是一个 `template class`，有两个 `template` 参数：字符型别 `charT` 和 `output` 迭代器型别 `OutIt`。产生的字符将写入迭代器所指位置。`output` 迭代器的缺省型别是 `ostreambuf_iterator<charT>`。`num_put` facet 提供了一系列 `put()` 函数，其间只有最后一个参数不同。你可以这么使用这些 facets：

```
std::locale      loc;
OutIt            to = ...;
std::ios_base&   fnt = ...;
charT            fill = ...;
T                value = ...;

// get numeric output facet of the loc locale
const std::num_put<charT, OutIt>& np
    = std::use_facet<std::num_put<charT, OutIt>>(loc);

// write value with numeric output facet
np.put(to, fnt, fill, value);
```

这些语句运用型别为 `charT` 的字符, 在 `output` 迭代器 `to` 所指位置处写出 `value` 的文本表述式。确切格式由 `fmt` 的格式化标志确认。`fill` 被用来当做填充字符。`put()` 返回一个迭代器, 指向最后一个被写出的字符的下一位置。

`num_put` facet 提供一系列成员函数, 分别接受型别为 `bool`, `long`, `unsigned long`, `double`, `long double` 及 `void*` 的对象作为最后一个参数。由于诸如 `short` 和 `int` 等内建型别的数值可以自动晋升 (转型) 为相应型别, 所以也就没有提供针对 `short` 和 `int` 的成员函数。

C++ *Standard* 要求, 每一个 `locale` 内都必须有 `num_put<char>` 和 `num_put<wchar_t>` 的具现实体 (两者的第二个 `template` 参数都采用默认值)。此外, C++ 标准程序库支持具备如下特性的所有具现实体: 以字符型别作为第一个 `template` 参数, 以 `output` 迭代器型别作为第二个 `template` 参数。当然, 标准规格书并未要求每一个 `locale` 都储存这些具现实体, 因为那会导致无穷多的 facet。

数值解析 (Numeric Parsing)

facet `num_get` 用来解析数值的文本表述。和 facet `num_put` 相较, 前者有两个 `template` 参数: 字符型别 `charT` 和 `input` 迭代器型别 `InIt` (默认值为 `istreambuf_iterator<charT>`)。它提供一系列的 `get()` 函数, 其间只有最后一个参数不同。这个 facet 的用法如下:

```
std::locale      loc;           // locale
InIt             beg = ...;     // begin of input sequence
InIt             end = ...;     // end of input sequence
std::ios_base&   fmt = ...;     // stream which defines input format
std::ios_base::iostate err;     // state after call
T               value;          // value after successful call

// get numeric input facet of the loc locale
const std::num_get<charT, InIt>& ng
    = std::use_facet<std::num_get<charT, InIt>>(loc);

// read value with numeric input facet
ng.get(beg, end, fmt, err, value);
```

这些语句的目的是解析 `beg` 和 `end` 之间的字符序列, 得出对应于型别 `T` 的一个数值。数值格式由参数 `fmt` 确定。如果解析失败, 就将 `err` 设为 `ios_base::failbit`, 反之则将 `err` 设为 `ios_base::goodbit`, 并将所生数值放入 `value` 之中。`value` 内的原值只有在解析成功后才被修改。如果全部字符序列都被解析完毕, `get()` 返回第二个参数 (`end`), 否则返回一个迭代器, 指向“无法被解析为数值得那一部分 (字符序列)”的第一个字符处。

`facet num_get` 支持“用来读取型别为 `bool`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double`, `long double` 和 `void*` 等对象”的函数。某些型别在 `num_put` 中并无对应函数, 例如 `unsigned short`, 这是因为写一个 `unsigned short` 值和写一个 `unsigned short` 值再提升转型为 `unsigned long` 是一样的。然而读取一个 `unsigned long` 值再转型为 `unsigned short`, 就和直接读取一个 `unsigned short` 的结果可能不同。

C++ *Standard* 要求每一个 `locale` 都必须保有 `num_get<char>` 和 `num_get<wchar_t>` 两个具现实体 (两者的第二个 `template` 参数均使用默认值)。此外, C++ 标准程序库还要求支持以下实体: 以字符型别作为第一个 `template` 参数, 以 `input` 迭代器型别作为第二个 `template` 参数。和 `num_put` 一样, 并非所有被支持的具现实体都必须储存在每一个 `locale` 对象中。

14.4.2 时间和日期的格式化

时间/日期的解析和格式化工作, 由 `time` 类型 (`category`) 中的两个 `facets` 提供: `time_get` 和 `time_put`。它们的几个成员函数借着操作“型别为 `tm`”的对象完成这项工作。型别 `tm` 定义于头文件 `<ctime>`。对象并非被直接传递, 而是以其地址为参数。

`time` 类型中两个 `facets` 的行为都和 `strftime()` 很有关系 (此函数亦定义于头文件 `<ctime>`)。此函数根据一个含有转换规格的字符串, 从一个 `tm` 对象生成一个字符串。表 14.9 对于这些转换规格做了简单的总结。这些转换规格在 `time_put facet` 中同样适用。

当然, `strftime()` 所产生的具体字符串还要根据 C `locale` 而定。表 14.9 中的例子系根据 “C” `locale`。

时间和日期的解析

`facet time_get` 是一个 `template`, 以一个字符型别 `charT` 和一个 `input` 迭代器型别 `InIt` 作为 `template` 参数。该 `input` 迭代器的缺省型别为 `istreambuf_iterator<charT>`。表 14.10 列出了 `time_get facet` 的成员。除了 `date_order()` 之外, 所有成员都对字符串进行解析, 并将结果储存于参数 `t` 所指的一个 `tm` 对象中。如果字符串不能被正确解析, 要不就回报错误 (也许通过参数 `err` 的修改), 要不就在 `tm` 对象中存储一个未定值。这意味程序产生出来的时间可以被确实地解析, 而用户输入的时间就未必了。通过参数 `fmt` 可以决定解析过程中使用其它哪些 `facets`。至于 `fmt` 的其它标志对于解析过程的影响则未予规定。

所有函数都返回一个迭代器, 指向最后一个读取字符的下一位置。一旦解析结束或有错误发生 (例如字符串无法被解析为日期), 解析动作立刻停止。

表 14.9 strftime() 的转换规格所用符号

规格符号	意义	实例
%a	星期缩写	Mon
%A	星期全名	Monday
%b	月份缩写	Jul
%B	月份全名	July
%c	locale 首选的日期和时间格式	Jul 12 21:53:22 1998
%d	日	12
%H	24 时制下的小时	21
%I	12 时制下的小时	9
%j	某年的第几天	193
%m	月份以数字表示	7
%M	分钟	53
%p	上午或下午 (am 或 pm)	pm
%S	秒	22
%U	从第一个星期日算起的周数	28
%W	从第一个星期一算起的周数	28
%w	星期, 以数字表示 (星期天为 0)	0
%x	locale 首选的日期格式	Jul 12 1998
%X	locale 首选的时间格式	21:53:22
%y	两位数年份	98
%Y	完整年份	1998
%Z	时区	MEST
%%	字符 '%'	%

函数用来读取星期和月份者, 既可接受缩写, 也可接受全名。如果缩写之后跟着一个字母, 而它恰好是全名中的某个有效字母, 则函数将试图读取全名。如果读取失败, 将无视“缩写名称曾经解析成功过”这个事实, 声明解析失败。

函数是否可以解析两位数年份? 这一点并无强制规定。

Date_order() 返回日期字符串中的年月日顺序。对日期而言这很有必要, 因为单单从字符串表示式来看, 你很难弄清楚顺序。例如 2003 年 2 月的第一天可以写成 3/2/1, 也可以写成 1/2/3。facet time_get 的基类 time_base 针对所有可能的年月日次序定义了一个枚举 (enum) 型别: dateorder。表 14.11 列出了其值。

C++ Standard 要求每一个 locale 都必须保存 time_get<char> 和 time_get<wchar_t> 的具现实体。此外 C++ 标准程序库要求支持以下形式的具现实体: 以 char 或 wchar_t 作为第一个 template 参数, 以相应的 input 迭代器作为第二个 template 参数。所有这些具现体都不要求储存于 locale 对象中。

表 14.10 time_get facet 的成员

表达式	意义
tg.get_time(beg,end,fmt,err,t)	解析 beg 和 end 之间 (用以表示时间) 的字符串, 其格式与 strftime() 以 %X 产生者一致
tg.get_date(beg,end,fmt,err,t)	解析 beg 和 end 之间 (用以表示日期) 的字符串, 其格式与 strftime() 以 %X 产生者一致
tg.get_weekday(beg,end,fmt, err,t)	解析 beg 和 end 之间用以表示星期的字符串
tg.get_monthname(beg,end,fmt, err,t)	解析 beg 和 end 之间用以表示月份的字符串
tg.get_year(beg,end,fmt,err,t)	解析 beg 和 end 之间用以表示年份的字符串
tg.date_order()	返回 facet 所用的年月日次序

表 14.11 dateorder 的枚举 (enum) 值

值	意义
no_order	无特别次序 (例如可以将一个日期写成西泽历格式)
dmy	次序为日, 月, 年
mday	次序为月, 日, 年
ymd	次序为年, 月, 日
ydm	次序为年, 日, 月

时间和日期的格式化

facet time_put 用来格式化时间和日期。它是一个 template, 接受一个字符型别 charT 作为 template 参数, 另一个可有可无的 (可选的) template 参数是 output 迭代器型别 OutIt。后者缺省为 ostreambuf_iterator (参见 p665)。

facet time_put 定义两个 put() 函数, 均被用来将“储存于 tm 对象中的日期信息”转化成一个字符序列, 写至 output 迭代器。表 14.12 列出了 time_put 的成员。

表 14.12 time_put facet 的成员

表达式	意义
tp.put(to,fmt,fill,t,cbeg,cend)	根据字符串 [cbeg;cend] 进行转换
tp.put(to,fmt,fill,t,cvt,mod)	运用转换规则 cvt 进行转换

上述两个函数都将结果写到 output 迭代器 to, 然后返回最后一个被写出字符的下一位置。参数 fmt 的型别是 ios_base, 用来存取其它 facets 及可能会有的附加信息。如果需要充填字符, 就使用 fill。参数 t 指向一个 tm 对象, 其中存放即

将被格式化的日期。

表 14.12 的第二个 `put()` 的最后两个参数是两个字符，此一版本将 `t` 所指的 `tm` 对象格式化，并将参数 `cvt` 视为转换指示符，传给 `strftime()`。这个 `put()` 函数只能进行一项转换，也就是由字符 `cvt` 指示的转换。每当有一个转换指示符被发现，此函数便被另一个 `put()` 函数调用。举个例子，以 ‘x’ 作为转换指示符，会导致储存于 `*t` 内的时间信息被写至 `output` 迭代器。C++ *Standard* 并未对参数 `mod` 的意义有任何强制定义，其主要意图是作为一个修饰符（modifier），针对 `strftime()` 函数的数个实作版本之间的转换进行修饰。

表 14.12 的第一个 `put()` 函数接受一个由区间 `[cbeg; cend)` 定义出来的字符串，其转换行为和 `strftime()` 非常相似。它扫描字符串，将任何一个“不隶属转换指示符”的字符直接写至 `output` 迭代器 `to`。如果遇到由 `%` 引导的转换指示符，它就从中提炼出一个（或有或无）的修饰符（modifier）和一个转换指示符。此函数将转换指示符和修饰符当做最后两个参数并调用另一版本的 `put()` 函数，直到处理完毕一个转换指示符后，`put()` 继续扫描字符串。

注意，这个 `facet` 有些不寻常，它提供了一个非虚拟成员函数：那个“以一个字符串作为转换指示符”的 `put()` 函数。`time_put` 派生类别无法对此函数进行改写，只能改写其它 `put()` 函数。

C++ *Standard* 要求每一个 `locale` 之中都必须储存两个具现体（instantiations）：`time_put<char>` 和 `time_put<wchar_t>`。此外 C++ 标准程序库还要求支持所有“第一个 `template` 参数为 `char` 或 `wchar_t`，第二个 `template` 参数为相应之 `input` 迭代器”的 `time_put<>` 具现体。C++ *Standard* 不保证支持以“`char` 和 `wchar_t` 以外的字符型别”作为第一个 `template` 参数的具现体，也不保证 `locale` 对象在缺省情况下带有 `time_put<char>` 和 `time_put<wchar_t>` 以外的任何具现体。

14.4.3 货币符号格式化

在 `monetary` 类型（category）中，有三个 facets：`moneypunct`，`money_get`，`money_put`。其中 `moneypunct` 用来定义货币格式，另两个 facets 使用该格式信息解析货币值。

货币符号

货币值在不同的场合中以不同的格式打印出来。不同的文化社群所习惯的格式有非常大的差异。例如货币符号（如果有的话）的布置、负值或正值的记法、国家或国际货币符号的使用、千位分隔符等等。为了具有必要的灵活性，格式细节被放在 `facet moneypunct` 中。

facet moneypunct 是一个 template，接受一个字符型别 charT 和一个默认值为 false 的布尔值。该布尔值表示将运用当地 (false) 或国际 (true) 货币符号。表 14.13 列出了 moneypunct 的成员。

表 14.13 moneypunct facet 的成员

表达式	意义
mp.decimal_point()	返回小数点表示字符
mp.thousands_sep()	返回千位符表示字符
mp.grouping()	返回一个字符串，用来指定千位符的布置格式
mp.curr_symbol()	返回货币符号的字符串
mp.positive_sign()	返回正号表示字符串
mp.negative_sign()	返回负号表示字符串
mp.frac_digits()	返回小数的位数
mp.pos_format()	返回非负数的格式
mp.neg_format()	返回负数的格式

moneypunct 派生自 class money_base，此一基类定义了一个枚举量，名为 part，用以形成一个货币值样式 (pattern)。这个类别还定义了一个 pattern 型别 (其实就是 char[4])，用来储存型别为 part 的四个值，形成一个样式，用以描述货币表述布局方式。表 14.14 列出了可以放进一个样式中的五种可能的 parts。

表 14.14 货币布局样式的局部内容

值	意义
none	在此位置上可以出现空白，但不强制要求
space	至少需要一个空白
sign	在此位置上需要一个正负符号
symbol	在此位置上需要一个货币符号
value	在此位置上需要一个值

moneypunct 定义有两个函数，用来返回样式：neg_format() 专门针对负值，pos_format() 专门针对非负值。在一个样式中，sign, symbol, value 三部分是必要的，none 和 space 两者必须出现其一。这并不意味着真的会打印出一个正负号或货币符号。parts 所指之处到底要印出什么，还得根据 facet 其它成员的返回值，以及格式化函数传来的格式化标志而定。

货币值肯定会被印出，它会根据 `value` 在样式中出现的位置而获得布局。货币值的小数字数由 `frac_digits()` 指定，`decimal_point()` 则可确定小数点字符（如果没有小数，就不会出现小数点）。

读取货币值时，允许千位符存在，但不强制要求。如果确实存在，就根据 `grouping()` 检查“分组布局”的正确性。如果 `grouping()` 是空的，就不允许出现千位符。千位符字符可由 `thousands_sep()` 返回。这里千位符的布局规则和数值格式化的情形一样（参见 p705）。打印货币值时会根据 `grouping()` 返回的字符串插入和打印千位符。读取货币值时，千位符可有可无，除非分组（`grouping`）字符串为空。所有其它解析工作成功后，便会检查千位符布局是否正确。

`space` 和 `none` 主要用来控制空白的布置。`space` 用于至少需要一个空白的位置处。格式化时，如果格式标志中指定了 `ios_base::internal`，则充填字符会被插入到 `space` 或 `none` 部分。当然，只有在指定的最小宽度没有使用其它字符时，充填才会发生。“被用来作为空格符”的充填字符应该以参数形式传给格式化函数。如果格式化的货币值不包含空白，`none` 可以被放在最后一个位置。`space` 和 `none` 不能出现在样式的第一部分。`space` 不能是样式的最后一部分。

货币值的正负符号可由一个以上的字符表示。例如某些场合，用括号括起一个值，便表示其为负值。样式中 `sign` 出现的位置，也就是正负号第一个字符出现的位置，其余所有正负号相关字符均出现在其它所有组件之后。如果正负号字符串为空，则不以任何符号标示正负。两个函数用来决定正负号字符：`positive_sign()` 和 `negative_sign()` 分别对应于正号和负号。

`symbol` 位置上将出现货币符号。只有当格式化过程 and 解析过程中 `ios_base::showbase` 标志被设立的情况下，此符号才有效。货币符号由函数 `curr_symbo()` 返回的字符串确定，此为当地符号——如果第二个 `template` 参数是 `false`（默认值），便以它来表示货币，否则将使用国际货币符号。

表 14.15 所示的以货币值 `$-1234.56` 为例，说明上述问题。当然，这意味着 `frac_digit()` 将返回 2。此外，宽度始终为 0。

C++ *Standard* 要求每一个 `locale` 之内都必须拥有四个具现体：`moneypunct<char>`，`moneypunct<wchar_t>`，`moneypunct<char, true>`，`moneypunct<wchar_t, true>`。C++ 标准程序库不支持其它任何具现体。

货币的格式化

`facet money_put` 被用来格式化货币值。它是一个 `template class`，接受字符型别 `charT` 作为第一个 `template` 参数，接受 `output` 迭代器型别 `OutIt` 作为第二个 `template` 参数。`output` 迭代器缺省为 `ostreambuf_iterator<charT>`。两个成员函数 `put()`

表 14.15 使用货币样式的例子

样式	正负号	结果
symbol none sign value		\$1234.56
symbol none sign value	-	\$-1234.56
symbol space sign value	-	\$ -1234.56
symbol space sign value	()	\$ (1234.56)
sign symbol space value	()	(\$ 1234.56)
sign value space symbol	()	(1234.56 \$)
symbol space value sign	-	\$ 1234.56-
sign value space symbol	-	-1234.56 \$
sign value none symbol	-	-1234.56\$

根据 `money_punct facet` 指定的格式产生一个字符序列。用来格式化的值或以 `long double` 传递，或以 `basic_string<charT>` 传递。下面是运用实例：

```
// get monetary output facet of the loc locale
const std::money_put<charT, OutIt>& mp
    = std::use_facet<std::money_put<charT, OutIt>> >(loc);

// write value with monetary output facet
mp.put(to, intl, fmt, fill, value);
```

参数 `to` 是一个型别为 `OutIt` 的 `output` 迭代器，格式化后的字符串将写到那儿去。`put()` 返回该型对象，指向“生成的最后一个字符”的下一位置。参数 `intl` 表示使用当地货币符号或国际货币符号。`fmt` 用来确定格式化标志，例如宽度和 `money_punct facet`（用来定义货币值打印格式）。`fill` 为充填字符。

参数 `value` 的型别是 `long double` 或 `basic_string<charT>`，其内容即为待被格式化的值。如果传入的是个字符串，此字符串可以全由小数组成，前面可以带上负号。如果字符串的第一字符是个负号，则该值将依负值形式加以格式化。一旦确定其值为负，负号即被丢弃。字符串中的小数个数可由 `money_punct facet` 的成员函数 `frac_digits()` 确定。

C++ *Standard* 要求每一个 `locale` 之内都必须拥有两个具现体：`money_put<char>` 和 `money_put<wchar_t>`。此外 C++ 标准程序库还要求支持“第一个 `template` 参数为 `char` 或 `wchar_t`，第二个 `template` 参数为相应之 `output` 迭代器”的所有 `money_put<>` 具现体，不过并不要求每一个 `locale` 都拥有它们。

货币的解析 (Monetary Parsing)

`facet money_get` 用来解析货币值。它是一个 `template class`, 接受字符型别 `charT` 作为第一个 `template` 参数, 接受 `Input` 迭代器型别 `InIt` 作为第二个 `template` 参数(缺省为 `istreambuf_iterator<charT>`)。此类别定义了两个 `get()` 成员函数, 用来解析一个字符序列。如果解析成功, 结果会被储存为 `long double` 型别, 或 `basic_string<charT>` 型别。下面是运用实例:

```
// get monetary input facet of the loc locale
const std::money_get<charT, InIt>& mg
    = std::use_facet<std::money_get<charT, InIt> >(loc);

// read value with monetary input facet
mg.get(beg, end, intl, fmt, err, val);
```

即将被解析的就是 `beg` 和 `end` 之间的字符序列。当所有元素都被解析完毕, 或是解析过程中遇到错误, 解析动作就立刻停止。如果发生错误, `err` 将设立 `ios_base::failbit`, 而 `val` 之中将不会有任何东西。如果成功, 解析结果放在参数 `val` 中 (它是 `passed by reference`), 型别为 `long double` 或 `basic_string`。

参数 `intl` 是个布尔值, 用来选定当地货币字符串或国际货币字符串。我们可以通过“被参数 `fmt` 渗入”的 `locale` 对象, 获取一个 `moneypunct facet`, 后者定义出待解析值的格式。对于货币的解析来说, `moneypunct facet` 的成员函数 `neg_format()` 所返回的样式始终都会被使用。

在 `none` 或 `space` 位置上, 货币解析函数会吞噬所有有效空白, 除非 `none` 是样式的最后一部分。尾部空白不会被略过。`get()` 函数会返回一个迭代器, 指向最后一个被吞噬字符的下一位置。

C++ *Standard* 要求每一个 `locale` 之内都必须拥有两个具现体: `money_get<char>` 和 `money_get<wchar_t>`。此外 C++ 标准程序库还要求支持所有“第一个 `template` 参数为 `char` 或 `wchar_t`, 第二个 `template` 参数为相应之 `input` 迭代器”的 `money_get<>` 具现体, 但并不要求每个 `locale` 都拥有它们。

14.4.4 字符的分类和转换

C++ 标准函数库有两个 `facets` 专门处理字符: `ctype` 和 `codecvt`, 它们都隶属于 `locale::ctype` 类型 (category)。`ctype` 主要用于字符分类, 例如判断某个字符是否为字母。此外还提供字母的大小写转换, 以及在“型别 `char`”和“该 `facet` 用以具现化的字符型别”之间的转换方法。`codecvt` 用来在不同编码之间进行字符转换, 主要用于 `basic_filebuf` 中的外部表述和内部表述之间的转换。

字符分类

`facet ctype` 是一个 `template class`, 针对字符型别实现参数化。在 `ctype<charT>` 中有三种函数:

1. `char` 和 `charT` 之间的转换函数。
2. 字符分类函数。
3. 字母大小写转换函数。

表 14.16 列出了 `facet ctype` 中定义的成员和功能。

表 14.16 `ctype<charT> facet` 所定义的功能

表达式	功能
<code>ct.is(m, c)</code>	测试字符 <code>c</code> 是否匹配 <code>mask</code> (掩码) <code>m</code>
<code>ct.is(beg, end, vec)</code>	为 <code>beg, end</code> 区间内的每个字符放置一个 <code>mask</code> , 后者必须匹配 <code>vec</code> 中对应位置上的字符
<code>ct.scan_is(m, beg, end)</code>	返回一个指针, 指向 <code>beg, end</code> 区间内第一个匹配 <code>mask m</code> 的字符; 如果没有找到, 返回 <code>end</code>
<code>ct.scan_not(m, beg, end)</code>	返回一个指针, 指向 <code>beg, end</code> 区间内第一个不匹配 <code>mask m</code> 的字符; 如果全部匹配, 返回 <code>end</code>
<code>ct.toupper(c)</code>	返回 <code>c</code> 的大写字符。如果没有的话就返回 <code>c</code>
<code>ct.toupper(beg, end)</code>	将 <code>beg, end</code> 区间内的每一个字符都以 <code>toupper()</code> 的转换结果取代之
<code>ct.tolower(c)</code>	返回 <code>c</code> 的小写字符。如果没有的话就返回 <code>c</code>
<code>ct.tolower(beg, end)</code>	将 <code>beg, end</code> 区间内的每一个字符都以 <code>tolower()</code> 的转换结果取代之
<code>ct.widen(c)</code>	将 <code>char</code> 转换为 <code>charT</code> , 并将结果返回
<code>ct.widen(beg, end, dest)</code>	将 <code>beg, end</code> 区间内的每一个字符都以 <code>widen()</code> 转换之, 并将结果存放于 <code>dest</code> 的相应位置
<code>ct.narrow(c, default)</code>	将 <code>char</code> 转换为 <code>charT</code> , 并将结果返回。如果没有合适的对应字符, 就返回 <code>default</code>
<code>ct.narrow(beg, end, default, dest)</code>	将 <code>beg, end</code> 区间内的每一个字符都以 <code>narrow()</code> 转换之, 并将结果存放于 <code>dest</code> 的相应位置

函数 `is(beg, end, vec)` 用来在一个数组中存储一系列 `masks` (掩码)。对于 `beg` 和 `end` 之间的每一个字符, 在 `vec` 所指数组中都有一个对应的 `mask` 以及相应的属性。一旦有很多字符需要被分类时, 这有助于避免“字符分类”的虚函数调用操作。

函数 `widen()` 用来将一个原生 (native) 字符集内 “型别为 `char`” 的字符转换为 “locale 所用字符集” 内的对应字符。也就是说此函数用来 “拓宽” 一个字符，即使结果亦为 `char` 型别也无妨。相反地，函数 `narrow()` 用来将 “locale 所用的字符集” 内的字符，转化为原生字符集内对应的 (型别为 `char`) 字符。以下例子将数字字符由 `char` 型别转换为 `wchar_t` 型别：

```
std::locale loc;
char narrow[] = "0123456789";
wchar_t wide[10];

std::use_facet<std::ctype<wchar_t>>>(loc).widen(narrow, narrow+10,
                                                    wide);
```

`class ctype` 派生自 `class ctype_base`。此类别仅用来定义一个枚举 (enum)，名为 `mask`，其中定义了一系列值，可用来合成一个 `bitmask`，用以检测字符属性。`ctype_base` 定义的值均列于表 14.17。字符分类的所有函数都需要一个 `bitmask` 参数，后者就是由 `ctype_base` 内的值组合而成。为了生成所需的 `bitmasks`，你得使用各种位操作符 (`|`, `&`, `^`, `~`)。如果某字符涵盖于某个 `mask` 所规范的字符内，那么该字符一定匹配那个 `mask`。

表 14.17 ctype 使用的各种字符掩码 (mask)

值	意义
<code>ctype_base::alnum</code>	测试是否为字母或数字 (等同于 <code>alpha digit</code>)
<code>ctype_base::alpha</code>	测试是否为字母
<code>ctype_base::cntrl</code>	测试是否为控制字符
<code>ctype_base::digit</code>	测试是否为数字
<code>ctype_base::graph</code>	测试是否为标点符号、字母或数字 (等同于 <code>alnum punct</code>)
<code>ctype_base::lower</code>	测试是否为小写字母
<code>ctype_base::print</code>	测试是否为可打印字符
<code>ctype_base::punct</code>	测试是否为标点符号
<code>ctype_base::space</code>	测试是否是空白
<code>ctype_base::upper</code>	测试是否是大写字母
<code>ctype_base::xdigit</code>	测试是否是十六进制数字

针对 `char` 而作的 `ctype` 特化版本

为了使字符分类函数获得更佳性能, `ctype` 针对字符型别 `char` 做了一个特化版本。此特化版本并未将字符分类函数 (`is()`, `scan_is()`, `scan_not()`) 委托给相应的虚函数去处理, 而是通过一个查表动作, 以 `inline` 方式直接实作。为此, 它提供了一些额外函数 (表 14.18)。

表 14.18 `ctype<char>` 的额外成员和功能

表达式	功能
<code>ctype<char>::table_size</code>	返回表格大小 (≥ 256)
<code>ctype<char>::classic_table()</code>	返回经典的 C locale 的表格
<code>ctype<char>(table, del=false)</code>	以 <code>table</code> 为表格, 产生一个 facet
<code>ct.table()</code>	返回 facet <code>ct</code> 的当前格式

你可以针对某个特定 locales, 操控上述函数的行为, 作法是将某个 `mask` 表格传给 `ctype` 构造函数作为参数:

```
// create and initialize the table
std::ctype_base::mask mytable[std::ctype<char>::table_size] = {
    ...
};

// use the table for the ctype<char> facet ct
std::ctype<char> ct(mytable, false);
```

上述程序代码生成一个 `ctype<char>` `facet`, 并以 `mytable` 确定某字符的字符类别 (character class)。说得更明确些, 字符 `c` 的字符类别由以下这个式子决定:

```
mytable[static_cast<unsigned char>(c)]
```

静态成员 `table_size` 是一个常量, 用来指示表格大小, 由标准程序库实作版本自行定义。其值至少为 256。 `ctype<char>` 构造函数的第二个 (可有可无的) 参数用来指示“当 `facet` 被销毁时, 表格是否应被删除”。如果是 `true`, 则当 `facet` 被销毁时, 传入构造函数的那个表格会被 `delete[]` 释放掉。

Protected 成员函数 `table()` 返回的是构造函数的第一参数 (表格)。Static protected 成员函数 `classic_table()` 返回的是经典 C locale 中用来进行字符分类的表格。

用于字符分类的全局辅助函数

标准程序库定义了一些全局函数, 它们可以协助我们很方便地运用 `ctype` facets。表 14.19 列出了所有这些全局辅助函数。

表 14.19 用于字符分类的全局辅助函数

函数	功能
isalnum(c, loc)	判断 c 是否为字母或数字 (等同于 isalpha() && isdigit())
isalpha(c, loc)	测试 c 是否为字母
iscntrl(c, loc)	测试 c 是否为控制字符
isdigit(c, loc)	测试 c 是否为数字
isgraph(c, loc)	测试 c 是否为标点符号、字母或数字 (等同于 alnum punct)
islower(c, loc)	测试 c 是否为小写字母
isprint(c, loc)	测试 c 是否为可打印字符
ispunct(c, loc)	测试 c 是否为标点符号 (可打印, 但不是空白、数字和字母)
isspace(c, loc)	测试 c 是否为空格符
isupper(c, loc)	测试 c 是否为大写字母
isxdigit(c, loc)	测试 c 是否为十六进制数字
tolower(c, loc)	将 c 从大写字母转换为小写字母
toupper(c, loc)	将 c 从小写字母转换为大写字母

例如，以下表达式用来确定 locale loc 中的一个字符 c 是不是小写字母：

```
std::islower(c, loc)
```

它会返回一个 bool 值。

如果 c 是 locale loc 中的小写字母，以下表达式返回 c 对应的大写字母：

```
std::toupper(c, loc)
```

如果 c 不是小写字母，则第一个参数会被原封不动地返回来。

以下表达式：

```
std::islower(c, loc)
```

等同于以下表达式：

```
std::use_facet<std::ctype<char> >(loc).is(std::ctype_base::lower, c)
```

此表达式调用 facet `ctype<char>` 的成员函数 `is()`，用以判定字符 c 是否符合第一个参数传来的 `bitmask` 所指定的字符属性。`Bitmask` 实值定义于 `ctype_base` 内。这些全局辅助函数的运用实例见 p502 和 p669。

这些字符分类全局函数和“同名但只有一个参数”的 C 函数相对应。这些定义于 `<cctype>` 和 `<ctype.h>` 的 C 函数总是采用当前的全局 C locales⁴, 用法更直接:

```
if (std::isdigit(c)) {  
    ...  
}
```

不过, 如果使用它们, 你没办法在同一个程序中使用不同的 locales。此外它们也无法运用使用者自定的 ctype facet。p497 有个例子, 展示如何运用这些 C 函数将一个字符串中的全部字符转换为大写字母。

有一点很重要: 如果效率问题至关重要, 你应该尽力避免使用这些 C++ 函数。你应该从 locale 中获取对应的 facet 对象, 然后直接使用该对象的成员函数, 这种做法要快得多。如果很多字符需要根据同一个 locale 进行分类, 那么这种做法获得的性能提升更加显著, 至少对于 non-char 字符来说如此。函数 `is(beg, end, vec)` 可用来判定典型字符的 masks (掩码): 此函数对 `[beg; end)` 区间内的每一个字符确定一个 mask 用以描述该字符属性。这些 mask 被存储在 `vec` 中, 其位置对应于字符位置。快速搜寻时可采用 vector。

字符编码 (Character Encoding) 转换

facet `codecvt` 用来在字符的内部编码和外部编码之间进行转换。举个例子, 只要某个“C++ 标准程序库实现版本”支持相应的 facet, 我们就可以运用 `codecvt` 在 Unicode 和 EUC (Extended UNIX Code) 之间进行转换。

这个 facet 在 `class basic_filebuf` 中被用来在“内部表述”和“文件表述”之间进行转换。Class `basic_filebuf<charT, traits>` (见 p627) 使用 `codecvt<charT, char, typename traits::state_type>` 的一个具现实体来完成此项任务; 所使用的 facet 是从储存于 `basic_filebuf` 中的 locale 取出来的。这是 `codecvt` 的主要用途——这个 facet 很少被直接使用。

14.1 节, p686 介绍了一些基本的字符编码方案。为了理解 `codecvt`, 你应该知道, 字符编码有两种方案: 一种是对每一个字符以固定个数的 bytes 表示 (此即宽字符表示法), 另一种是对每一个字符以不同个量的 bytes 表示 (multibytes 表示法)。

你还应该了解, multibytes 表示法为了使字符的空间储存率更高, 使用所谓的 **shift states** (转换状态)。只有当前位置之 shift state 正确, 才可能正确翻译 byte 的含义。进一步说, 也只有遍历了整个 multibytes 字符序列后, 才能作出正确理解 (细节参见 14.1 节, p686)。

facet `codecvt<>` 有三个 template 参数:

⁴ 只有当“前一次调用 `locale::global()` 时系根据一个具名 locale 实施”, 而且从那时候起没有再调用过 `setlocale()`, 这个 locale 才等同于 C++ 的全局 locale。否则 C 函数使用的 locale 和 C++ 全局 locale 并不一致。

1. 字符型别 `internT`，用来指定内部表述方式。
2. 型别 `externT`，用来指定外部表述方式。
3. 型别 `stateT`，表示转换过程中的中间状态。

中间状态可能由不完全的宽字符或当前的转换状态（`shift state`）组成。C++ 标准程序库对于该 `state` 对象中到底存放什么东西，没有强制要求。

内部表述始终采用“每个字符的 `bytes` 个数固定”的表述方案。大部分情况下程序使用的都是 `char` 和 `wchar_t` 两个型别。至于外部表述法，可能使用固定 `bytes` 表示法，也可能使用 `multibytes` 表示法。如果采用后者，那么第二个 `template` 参数就用来表示 `multibytes` 编码中的基本单位的型别——每一个 `multibytes` 字符都存放在一个或多个该型对象中。通常这个型别是 `char`。

第三个参数用来表示当前转换状态（`shift state`）。某些时候——例如采用 `multibytes` 编码时——它就显现出必要性来了。这种情况下，在处理一个字符的过程中，可能由于“源端缓冲区”已空，或“目的端缓冲区”已满，导致 `multibytes` 字符的处理中断。如果这种情况出现，就将当前转换状态储存于该型对象中。

就像对其它 `facets` 一样，C++ *Standard* 只强制要求支持少数几个转换。C++ 标准程序库只支持以下两个具现实体（`instantiations`）：

1. `codecvt<char, char, mbstate_t>`，将原生（`native`）字符集转换为其自身（其实就是 `codecvt facet` 的退化版本）。
2. `codecvt<wchar_t, char, mbstate_t>`，在原生窄字符集（亦即 `char`）与原生宽字符集（亦即 `wchar_t`）之间进行转换。

C++ 标准程序库并没有指定第二个转换的确切语意。比较自然的做法是在 `wchar_t` 转换为 `char` 时，将每一个 `wchar_t` 切割为 `sizeof(wchar_t)` 个 `char` 对象。反向转换时则将 `sizeof(wchar_t)` 个 `chars` 组装成一个 `wchar_t`。注意，这个转换动作与 `ctype facet` 的成员函数 `widen()` 和 `narrow()` 很不相同：`codecvt` 函数使用多个 `chars` 构成一个 `wchar_t`（或反之），`ctype` 函数则是将某种编码下的某个字符转换为另一种编码下的对应字符（如果可以的话）。

和 `ctype facet` 一样，`codecvt` 也派生自一个基类 `codecvt_base`，该基类之中也定义了一个枚举型别（`enum`）`result`，其枚举值用以指定 `codecvt` 成员函数的结果（型别），确切意义视具体的成员函数而定。表 14.20 列出 `codecvt` 的成员函数。

函数 `in()` 将一个外部表述转换为内部表述。参数 `s` 是一个 `reference`，指向 `stateT`。这个“代表转移状态（`shift state`）”的参数在转换开始时派上用场；结束时最后一个转移状态亦被记录于其中。如果待转换的输入缓冲区不是第一个被转换的缓冲区，则传入的转移状态可能和初始状态有所不同。参数 `fb`（意思是 `from begin`）和 `fe`（意思是 `from end`）的型别都是 `const internT*`，分别代表输入缓冲区的起

点和终点。参数 `tb` (意思是 `to begin`) 和 `te` (意思是 `to end`) 的型别为 `externT*`, 分别代表输出缓冲区的起点和终点。参数 `fn` (意思是 `from next`, 型别为 `externT*&`) 和 `tn` (意思是 `to next`, 型别为 `internT*&`) 分别用来返回输入和输出缓冲区中经过转换的序列的终点。函数返回型别为 `codecvt_base::result`, 其值见表 14.21。

表 14.20 `the codecvt facet` 的成员

表达式	意义
<code>cvt.in(s, fb, fe, fn, tb, te, tn)</code>	将外部表述转换为内部表述
<code>cvt.out(s, fb, fe, fn, tb, te, tn)</code>	将内部表述转换为外部表述
<code>cvt.unshift(s, tb, te, tn)</code>	撰写脱字序列 (escape sequence) 以切换最初的转移状态 (shift state)
<code>cvt.encoding()</code>	返回外部编码相关信息
<code>cvt.always_noconv()</code>	如果不曾有过成功的转换, 返回 <code>true</code>
<code>cvt.length(s, fb, fe, max)</code>	从序列 (<code>fb</code> 和 <code>fe</code> 之间) 返回 <code>externTs</code> 的数量, 以便生成 <code>max</code> 个内部字符
<code>cvt.max_length()</code>	返回“生成一个 <code>internT</code> ”所需的最大数量的 <code>externTs</code>

表 14.21 转换函数的返回值

返回值	意义
<code>ok</code>	所有字符都被成功转换了
<code>partial</code>	(1) 并非所有字符都被成功转换, 或 (2) 需要更多字符以求生成目标字符
<code>error</code>	遇到一个不能转换的源字符 (source character)
<code>noconv</code>	无需转换

返回值 `ok` 表示函数已经取得了一些进展。如果保持 `fn == fe`, 则代表整个输入缓冲区都被处理了, 而且 `tb` 和 `tn` 间的序列内含转换结果, 其中的字符表示输入序列中的字符, 并有一个上次转换留下的结束字符。如果 `in()` 的参数 `s` 不是初始状态, 则其中储存上次转换未完成的那个“局部字符 (partial character)”。

返回值 `partial` 有两层意义, 其一表示输入缓冲区的内容尚未耗尽之前输出缓冲区已经满溢, 其二表示字符处理完毕之前输入缓冲区的内容已经耗尽。此时 `tb` 和 `tn` 之间的序列包含了所有转换完毕的字符, 但输入缓冲区的尾端有一个尚未被完全转换的“部分字符”。那么下次转换时, 为了正确转换这个“部分字符”, 需要一些信息, 它们就储存在转移状态 `s` 中。如果 `fe != fn`, 则输入缓冲区尚未清空, 此时必定 `te == tn`, 也就是输出缓冲区满溢。下次转换便会从 `fn` 开始。

返回值 `noconv` 用来指示一个特殊情况：外部表述转换为内部表述时无需进行任何转换。此时 `fn` 和 `fb` 相同，`tn` 和 `tb` 相同。目标序列中没有任何东西，因为输入缓冲区的内容便已够用。

返回值 `error` 意味遇到不可转换的字符。这有好几种可能，也许是“标的字符集”中没有合适的表述可以对应该字符，也许是输入序列结束时的转移状态是非法的。*C++ Standard* 并未进一步规划任何函数用来调查错误原因。

函数 `out()` 和 `in()` 相似，唯其转换方向相反。也就是说它将内部表述转化为外部表述。参数和返回值的意义和 `in()` 一样，只不过参数型别刚好调换过来：`tb` 和 `te` 是 `internT*`，`fb` 和 `fe` 是 `const externT*`。同样的情况也在 `fn` 和 `tn` 身上发生。

当目前转换状态被当做函数 `unshift()` 的参数 `s`，该函数会安插必要字符以形成一个序列。这通常意味 `shift state` 被切换为 `switch state`，只有外部表述到达尾端。因此，参数 `tb` 和 `tf` 的型别都是 `externT*`，`tn` 的型别是 `externT&*`。`tb` 和 `te` 之间的序列就是输出缓冲区，其中储存着转换后的字符。成果序列的终点储存于 `tn` 之中。`unshift()` 的返回值列于表 14.22。

表 14.22 `unshift()` 的返回值

返回值	意义
<code>ok</code>	序列成功完成
<code>partial</code>	需要存进更多字符，才能完成这个序列
<code>error</code>	错误（无效）状态
<code>noconv</code>	完成这个序列无需任何字符

函数 `encoding()` 返回外部表述的编码信息。如果返回 `-1`，则转换动作将由状态决定。如果返回 `0`，则产生内部字符所需的 `externTs` 数量不是常数。如果返回 `-1` 和 `0` 之外的其它数值，该值表示生成内部字符所需的 `externTs` 数量。这个信息可用来判断缓冲区的概略大小。

如果函数 `in()` 和 `out()` 永远不进行转换，则函数 `always_noconv()` 返回 `true`。例如 `codecvt<char,char,mbstate>` 不作任何转换，所以其 `always_noconv()` 返回 `true`。然而这一点只对 "C" locale 中的 `codecvt facet` 是肯定的，其它实体可能确实会执行转换动作。

函数 `length()` 返回“生成 `max` 个内部字符”所需的必要的 `externTs` 个数 (在 `fb` 和 `fe` 之间)。如果 `fb-fe` 序列之中完整的 `internT` 字符数少于 `max`, 此函数就返回“能够产生序列内最多 `internTs`”的 `externTs` 的数量。

14.4.5 字符串校勘 (String Collation)

`facet collate` 用来处理字符串排序时各种不同约定间的差异。例如, 在德国, 对字符串进行排序时, 字母 `ü` 与字母 `u` 及 `ue` 等同。但是对其它语言而言, 这个字母甚至不是字母, 就算遇上了也只是被当成一个特殊字符。针对同一个字符序列, 不同的语言进行排序时, 规则可能稍有不同。`facet collate` 可以提供用户熟悉的字符串排序。表 14.23 列出这个 `facet` 的所有成员函数。在此表中, `col` 是 `collate` 的一个具体体, 函数参数是一些迭代器, 用来定义目标字符串。

表 14.23 `collate<> facet` 的成员函数

表达式	意义
<code>col.compare(beg1, end1, beg2, end2)</code>	返回 1——如果第一字符串大于第二字符串 返回 0——如果两字符串相等 返回 -1——如果第一字符串小于第二字符串
<code>col.transform(beg, end)</code>	返回一个字符串, 被用来和其它被改造的字符串做比较
<code>col.hash(beg, end)</code>	返回字符串的一个 hash value (型别为 <code>long</code>)

`facet collate` 是个 `template class`, 以字符型别 `charT` 作为 `template` 参数。传给 `collate` 成员函数的字符串, 乃是以型别为 `const charT*` 的迭代器指出。这多少有点不幸, 因为没有谁能够保证 `basic_string<charT>` 中的迭代器也是指针, 所以比较字符串时不得不这么做:

```

locale loc;
string s1, s2;
...
// get collate facet of the loc locale
const std::collate<charT>& col
    = std::use_facet<std::collate<charT>> >(loc);

// compare strings by using the collate facet
int result = col.compare(s1.data(), s1.data()+s1.size(),
                        s2.data(), s2.data()+s2.size());

if (result == 0) {
    // s1 and s2 are equal
    ...
}

```

之所以有这样的限制，是因为你无法事先猜出需要哪一种迭代器型别。因此其实有必要为指针型别和数量不限的迭代器型别配备 collation facets。

当然，locale 之中特殊的便捷函数（convenience functions）可以在这里进行字符串比较（参见 p703）：

```
int result = loc(s1,s2);
```

但这只对 compare() 有效。C++ 标准程序库并未针对 collate 的另外两个成员函数定义对应的便捷函数。

transform() 返回一个型别为 basic_string<charT> 的对象。这些字符串的字典顺序（lexicographical order）和使用 collate() 的原始字符串相同。如果一个字符串有必要和其它众多字符串比较，那么这个顺序可以协助提高性能。决定字符串字典顺序的动作，它远比 collate() 快得多，因为地域化的排序规则可能相对复杂。

C++ 标准程序库只强制要求支持 collate<char> 和 collate<wchar_t> 两个具现体。至于其它字符型别，用户必须自行撰写特化版本——其中可能会运用标准具现体。

14.4.6 信息国际化

messages facet 用来从一个信息索引（message catalog）中获得已被国际化的信息。这个 facet 主要用来提供类似 perror() 的功能。perror() 在 POSIX 系统中系根据储存于全局变量 errno 中的错误编号，印出一个系统错误信息。当然，messages 提供的功能更加灵活，可惜其定义不尽精确。

messages facet 是一个 template class，以字符型别 charT 为其 template 参数。这个 facet 返回的字符串，型别为 basic_string<charT>。此一 facet 的基本用途是：打开一个信息名册、从中获取信息、然后关闭该名册。Class messages 派生自 class messages_base，后者定义了一个 catalog 型别（其实就是一个 int 型别）。此型别的对象标示一个名册，messages 的成员函数即针对此一名册进行操作。表 14.24 列出 messages 的成员函数。

open() 函数接受的 name 参数标示出一个名册，相应信息字符串就储存于其中。它也可以是文件名称。参数 loc 标示出一个 locale 对象，通过它可以存取 ctype facet——我们可以运用这个 facet 将信息转换成我们所期望的字符型别。

成员函数 get() 的确切语义未有定义。举个例子，POSIX 系统中的一个实作版本会返回一个对应于 msgid 的错误信息字符串。但 C++ Standard 并未作强制规定。参数 set 用来在信息中建立一个子结构，例如可用来区别系统错误和 C++ 标准程序库的错误。

表 14.24 messages<> facet 的成员函数

表达式	意义
msg.open(name, loc)	打开一个名册 (catalog)，返回对应的 ID
msg.get(cat, set, msgid, def)	从名册 cat 中返回 msgid 的对应信息
msg.close(cat)	关闭名册 cat

如果不再需要某个信息名册，可以使用 close() 关闭之。虽然像 open() 和 close() 这样的接口名称似乎暗示信息来自某个文件，但 C++ *Standard* 其实并无此硬性要求。现实中更有可能的情况是 open() 从文件中读取，然后将信息储存于内存。后继调用的 close() 则释放该内存。

C++ *Standard* 要求每一个 locale 之内都必须储存两个具现体: messagest<char> 和 messages<wchar_t>。除此之外 C++ 标准程序库不支持其它任何具现体。

15

空间配置器

Allocator

我在 3.4 节, p31 曾经介绍过空间配置器 (allocator, 简称配置器), 它代表一种特定内存模型 (memory model), 并提供一种抽象概念, 以便将对内存的申请最终转变为对内存的直接调用。本章将更详细地描述 allocator。

15.1 应用程序开发者如何使用配置器

就应用程序员而言, 使用不同的配置器不应该遭遇什么困难。你只需简单地将配置器当做一个 template 参数传入即可。下例为使用一个特殊的配置器 MyAlloc, 产生特殊的容器和 strings:

```
// a vector with special allocator
std::vector< int, MyAlloc<int> > v;

// an int/float map with special allocator
std::map< int, float, less<int>,
        MyAlloc< std::pair<const int, float> > > m;

// a string with special allocator
std::basic_string< char, std::char_traits<char>, MyAlloc<char> > s;
```

如果打算使用自己开发的配置器, 那么先制定一些型别定义以避免冗长的代码, 通常会比较好。例如:

```
// special string type that uses special allocator
typedef std::basic_string< char, std::char_traits<char>,
                        MyAlloc<char> > xstring;

// special string/string map type that uses special allocator
typedef std::map< xstring, xstring, less<xstring>,
                MyAlloc<std::pair<const xstring, xstring> > > xmap;

// create object of this type
xmap mymap;
```


使用非缺省的配置器分配出来的对象，并不会引起你的什么异样感觉。然而请注意，别把不同配置器分配出来的元素弄混了，那会造成未定义的行为。你可以运用 `operator==` 来比较两个配置器是否使用相同的内存模型。如果它返回 `true`，表示其中一个配置器分配的储存空间可以由另一个配置器收回。所有“以配置器为 `template` 参数”的型别，都会提供一个 `get_allocator()` 函数，我们可由此得到对应的配置器。例如：

```
if (mymap.get_allocator() == s.get_allocator()) {
    // OK, mymap and s use the same or interchangeable allocators
    ...
}
```

15.2 程序库开发者如何使用配置器

这一节为另一种人讲解配置器的用法，这些人运用配置器来实作出容器或其它组件，而那些容器或组件可掌握不同的配置器。本节由 Bjarne Stroustrup 的 *The C++ Programming Language, 3/c* 书中 19.4 节发展而来——当然是经过授权的©。

配置器提供了一个接口，包括分配、生成、销毁和回收对象（表 15.1）。通过配置器 `s`，容器和算法的元素储存方式也因而得以被参数化。例如你可以实作出运用共享内存（shared memory）的配置器，或是让 `map` 的元素储存在持久性数据库（persistent database）中。

表 15.1 基本的空间分配操作

表达式	效果
<code>a.allocate(num)</code>	为 <code>num</code> 个元素分配内存
<code>b.construct(p)</code>	将 <code>p</code> 所指的元素初始化
<code>destroy(p)</code>	销毁 <code>p</code> 所指的元素
<code>deallocate(p,num)</code>	回收 <code>p</code> 所指的©©© “可容纳 <code>num</code> 个元素”的内存空间

让我以一个简单的 `vector` 实作方案为例。配置器被当做 `template` 参数或构造函数参数，传递给 `vector`，然后被保存在其内部某处：

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class vector {
        ...
    private:
```

```

        Allocator alloc;           // allocator
        T* elems;                  // array of elements
        size_type numElems;        // number of elements
        size_type sizeElems;       // size of memory for the elements
        ...

public:
    // constructors
    explicit vector(const Allocator& = Allocator());
    explicit vector(size_type num, const T& val = T(),
                   const Allocator& = Allocator());
    template <class InputIterator>
    vector(InputIterator beg, InputIterator end,
           const Allocator& = Allocator());
    vector(const vector<T,Allocator>& v);
    ...
};
}

```

上述第二个构造函数根据元素个数 `num` 及数值 `val` 来初始化 `vector`，可实作如下：

```

namespace std {
    template <class T, class Allocator>
    vector<T,Allocator>::vector(size_type num, const T& val,
                               const Allocator& a)
    : alloc(a) // initialize allocator
    {
        // allocate memory
        sizeElems = numElems = num;
        elems = alloc.allocate(num);

        // initialize elements
        for (size_type i=0; i<num; ++i) {
            // initialize ith element
            alloc.construct(&elems[i],val);
        }
    }
}

```

表 15.2 针对“未初始化之内存”的一些方便好用的函数

表达式	效果
<code>uninitialized_fill(beg, end, val)</code>	以 <code>val</code> 初始化 <code>[beg; end]</code>
<code>uninitialized_fill_n(beg, num, val)</code>	以 <code>val</code> 初始化 <code>beg</code> 开始的 <code>num</code> 个元素
<code>uninitialized_copy(beg, end, mem)</code>	以 <code>[beg; end]</code> 的各个元素初始化 <code>mem</code> 开始的各个元素

为了初始化那些尚未初始化的内存, C++ 标准程序库提供了一些简便函数, 如表 15.2。有了这些函数, 构造函数的实作就更简单了:

```
namespace std {
    template <class T, class Allocator>
    vector<T, Allocator>::vector(size_type num, const T& val,
                                const Allocator& a)
        : alloc(a) // initialize allocator
    {
        // allocate memory
        sizeElems = numElems = num;
        elems = alloc.allocate(num);

        // initialize elements
        uninitialized_fill_n(elems, num, val);
    }
}
```

成员函数 `reserve()` 用来在“不改变元素个数”的前提下保留更多内存, 可实作如下:

```
namespace std {
    template <class T, class Allocator>
    void vector<T, Allocator>::reserve(size_type size)
    {
        // reserve() never shrinks the memory
        if (size <= sizeElems) {
            return;
        }

        // allocate new memory for size elements
        T* newmem = alloc.allocate(size);
```

```

        // copy old elements into new memory
        uninitialized_copy(elems,elems+numElems,newmem);

        // destroy old elements
        for (size_type i=0; i<numElems; ++i) {
            alloc.destroy(&elems[i]);
        }

        // deallocate old memory
        alloc.deallocate(elems,sizeElems);

        // so, now we have our elements in the new memory
        sizeElems = size;
        elems = newmem;
    }
}

```

原始储存区的迭代器 (Raw Storage Iterators)

C++ 标准程序库还提供了一个 `class raw_storage_iterator`，用来在未初始化的内存中走访并进行初始化工作。这么一来你就可以借助它，使用任何算法并以该算法的结果作为内存的初值。

例如以下语句以 `[x.begin():x.end())` 区间内的元素值初始化 `elems` 所指的储存区：

```

copy (x.begin(), x.end(), // source
      raw_storage_iterator<T*,T>(elems)); // destination

```

第一个 `template` 参数（上述的 `T*`）必须是一个对应于元素型别的 `output` 迭代器。

第二个 `template` 参数（上述的 `T`）必须是元素型别。

临时缓冲区 (Temporary Buffers)

也许你曾经在某些程序里发现它们用了两个函数 `get_temporary_buffer()` 和 `return_temporary_buffer()`。这两个函数用来处理一些未初始化的内存，以供函数短暂需求。注意，`get_temporary_buffer()` 返回的内存可能少于预期，所以其返回值是个 `pair`，内含所获得的内存地址和实际大小（以元素为单位）。下面是一个运用实例：

```

void f()
{
    // allocate memory for num elements of type MyType
    pair<MyType*,std::ptrdiff_t> p = get_temporary_buffer<MyType>(num);
}

```

```
    if (p.second == 0) {
        // could not allocate any memory for elements
        ...
    }
    else if (p.second < num) {
        // could not allocate enough memory for num elements
        // however, don't forget to deallocate it
        ...
    }

    // do your processing
    ...

    // free temporarily allocated memory, if any
    if (p.first != 0) {
        return_temporary_buffer(p.first);
    }
}
```

不过, 使用 `get_temporary_buffer()` 和 `return_temporary_buffer()` 很难写出“异常安全”(exception-safe)的程序代码, 所以基本上它们已经不再被运用于程序库中。

15.3 C++ 标准程序库的缺省配置器

C++ 标准程序库的缺省配置器声明如下:

```
namespace std {
    template <class T>
    class allocator {
    public:
        // type definitions
        typedef size_t      size_type;
        typedef ptrdiff_t   difference_type;
        typedef T*          pointer;
        typedef const T*    const_pointer;
        typedef T&          reference;
        typedef const T&    const_reference;
        typedef T           value_type;
```

```

// rebind allocator to type U
template <class U>
struct rebind {
    typedef allocator<U> other;
};

// return address of values
pointer address(reference value) const;
const_pointer address(const_reference value) const;

// constructors and destructor
allocator() throw();
allocator(const allocator&) throw();
template <class U>
    allocator(const allocator<U>&) throw();
~allocator() throw();

// return maximum number of elements that can be allocated
size_type max_size() const throw();

// allocate but don't initialize num elements of type T
pointer allocate(size_type num,
                allocator<void>::const_pointer hint = 0);

// initialize elements of allocated storage p with value value
void construct(pointer p, const T& value);

// delete elements of initialized storage p
void destroy(pointer p);

// deallocate storage p of deleted elements
void deallocate(pointer p, size_type num);
};
}

```

缺省配置器使用全局的 `operator new` 和 `operator delete` 分配和回收内存。所以 `allocate()` 可能会抛出一个 `bad_alloc` 异常。然而缺省配置器可以优化，例如“将刚被回收的内存分配出去”或是“一次分配较大的空间以备后续之需”。所以 `operator new` 或 `operator delete` 被调用的确切时机难以预测。参见 p735，那儿有一个缺省配置器的实作案例。

配置器中有一个奇怪的 template 结构定义: rebind。这个 template 结构使得任何配置器可以间接为其它型别分配空间。举个例子, 假设 Allocator 是一个配置器型别, 那么:

```
Allocator::rebind<T2>::other
```

是同一个配置器针对另一个元素型别 T2 所做的特化版本。

如果你需要实作一个容器, 其中必须为“非元素型别”之对象分配空间, rebind<>就可以派上用场。例如实作 deque 时你通常需要将内存分配给一个数组, 用以管理元素区块 (deque 的典型实作请参考 p160)。因此你需要一个配置器来分配一个数组, 其元素型别为指针, 指向 deque 的元素:

```
namespace std {
    template <class T,
               class Allocator = allocator<T> >
    class deque {
    ...
    private:
        // rebind allocator for type T*
        typedef typename Allocator::rebind<T*>::other PtrAllocator;

        Allocator alloc;           // allocator for values of type T
        PtrAllocator block_alloc; // allocator for values of type T*
        T** elems;                 // array of blocks of elements
        ...
    };
}
```

为了管理 deque 的元素, 你必须有一个配置器用来处理元素的数组/区块, 另一个配置器用来处理区块所形成的数组 (译注: 《STL 源码剖析》4.4 节对于 deque 的实作有很好的剖析和图解)。deque 元素的配置器通过 rebind<> 被结合于元素数组型别 (T*) 之上。

缺省配置器也针对 void 做了如下特化版本:

```
namespace std {
    template <>
    class allocator<void> {
    public:
        typedef void*           pointer;
        typedef const void*     const_pointer;
        typedef void            value_type;
        template <class U>
        struct rebind {
            typedef allocator<U> other;
        };
    };
}
```

```
};  
}
```

15.4 使用者自行定义的配置器

自己写一个配置器并不很难。最重要的问题是你如何分配和回收储存空间，剩下的工作就很简单了。下面是个范例，看看缺省配置器的一个简单实作方案：

```
// util/defalloc.hpp  
  
namespace std {  
    template <class T>  
    class allocator {  
    public:  
        // type definitions  
        typedef size_t      size_type;  
        typedef ptrdiff_t   difference_type;  
        typedef T*          pointer;  
        typedef const T*    const_pointer;  
        typedef T&          reference;  
        typedef const T&    const_reference;  
        typedef T           value_type;  
  
        // rebind allocator to type U  
        template <class U>  
        struct rebind {  
            typedef allocator<U> other;  
        };  
  
        // return address of values  
        pointer address (reference value) const {  
            return &value;  
        }  
        const_pointer address (const_reference value) const {  
            return &value;  
        }  
    }  
}
```



```
/* constructors and destructor
 * - nothing to do because the allocator has no state
 */
allocator() throw() {
}
allocator(const allocator&) throw() {
}
template <class U>
    allocator (const allocator<U>&) throw() {
}
~allocator() throw() {
}

// return maximum number of elements that can be allocated
size_type max_size () const throw() {
    // for numeric_limits see Section 4.3, page 59
    return numeric_limits<size_t>::max() / sizeof(T);
}

// allocate but don't initialize num elements of type T
pointer allocate (size_type num,
                  allocator<void>::const_pointer hint = 0) {
    // allocate memory with global new
    return (pointer) (::operator new(num*sizeof(T)));
}

// initialize elements of allocated storage p with value value
void construct (pointer p, const T& value) {
    // initialize memory with placement new
    new((void*)p)T(value);
}

// destroy elements of initialized storage p
void destroy (pointer p) {
    // destroy objects by calling their destructor
    p->~T();
}
```

```
// deallocate storage p of deleted elements
void deallocate (pointer p, size_type num) {
    // deallocate memory with global delete
    ::operator delete((void*)p);
}

// return that all specializations of this allocator are interchangeable
template <class T1, class T2>
bool operator== (const allocator<T1>&,
                 const allocator<T2>&) throw() {
    return true;
}

template <class T1, class T2>
bool operator!= (const allocator<T1>&,
                 const allocator<T2>&) throw() {
    return false;
}
}
```

在这个基础上，你会发现，完成自己的配置器轻而易举。通常你需要变化的只是 `max_size()`, `allocate()`, `deallocate()`。你可以将自己在内存分配方面的独到策略体现于这三个函数内，例如重新运用内存（而不是立刻释放），或使用共享内存（shared memory），或是把内存映像到一个面向对象数据库内。你可以在 <http://www.josuttis.com/libbook/examples.html> 中找到一些其它例子。

15.5 配置器细部讨论

根据 C++ *Standard* 规定，配置器必须提供以下各个型别定义和操作函数。如果配置器希望被标准容器运用，那么还另有一些特殊要求；如果不打算配合标准容器，要求就少一些。

15.5.1 型别定义

allocator::value_type

- 元素型别。
- 等同于 `allocator<T>` 中的 `T`。

allocator::size_type

- 一个无正负号的整数型别，表示应用程序模型 (application model) 中最大对象之大小。
- 如欲配合标准容器使用，此型别必须等于 `size_t`。

allocator::difference_type

- 一个有正负号的整数型别，表示分配模型中两个指针之间的差距。
- 如欲配合标准容器使用，此型别必须等于 `ptrdiff_t`。

allocator::pointer

- 一个“指向元素”的指针型别。
- 如欲配合标准容器使用，此型别必须等同于 `allocator<T>` 中的 `T*`。

allocator::const_pointer

- 一个“指向元素”的常数指针型别。
- 如欲配合标准容器使用，此型别必须等同于 `allocator<T>` 中的 `const T*`。

allocator::reference

- 一个“指向元素”的 `reference` 型别。
- 此型别必须等同于 `allocator<T>` 中的 `T&`。

allocator::const_reference

- 一个“指向元素”的 `const reference` 型别。
- 此型别必须等同于 `allocator<T>` 中的 `const T&`。

allocator::rebind

- 是一个 `template` 结构体，使任何配置器可以间接为其它型别分配空间。
- 必须声明如下：

```
template <class T>
class allocator {
public:
    template <class U>
    struct rebind {
        typedef allocator<U> other;
    };
    ...
}
```

- 关于 `rebind` 的意义和说明，详见 p734。

15.5.2 各项操作

allocator::allocator ()

- 缺省构造函数。
- 产生一个配置器对象。

allocator::allocator (const allocator& a)

- 复制构造函数。
- 产生一个配置器副本，使得原配置器所分配的空间可通过另一个配置器回收。

allocator::~allocator ()

- 析构函数。
- 销毁配置器对象。

pointer **allocator::address** (reference value)

const_pointer **allocator::address** (const_reference value)

- 第一个函数返回一个非常数指针，指向一个非常数值。
- 第二个函数返回一个常数指针，指向一个常数值。

size_type **allocator::max_size** ()

- 返回一个③③③“对 `allocate()` 有意义、用于分配空间”的最大许可值。

pointer **allocator::allocate** (size_type num)

pointer **allocator::allocate** (size_type num,
allocator<void>::const_pointer hint)

- 两种形式都返回一块空间，可容纳 `num` 个型别为 `T` 的元素。
- 元素不会被构造/初始化（亦即构造函数不会被调用）。
- 可有可无（可选）的第二参数，其真实意义由实作版本决定。它或许可用来辅助提升效能。

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

- 释放 `p` 所指的空间。
- `p` 所指的空间必须是由同一个（或等同的）配置器以 `allocate()` 分配出来。
- `p` 不可以是 `NULL` 或 `0`。
- 区块内的元素必须已经被析构。

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

- 以 `value` 作为 `p` 所指的那个元素的初值。
- 相当于 `new((void*)p)T(value)`。

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

- 销毁 p 所指的对象，但不回收空间。
- 只是单纯调用对象的析构函数。
- 相当于 $((T^*)p) \rightarrow \sim T()$ 。

```
bool operator == (const allocator& a1, const allocator& a2)
```

- 如果配置器 $a1$ 和 $a2$ 是可互换的 (interchangeable)，就返回 true。
- 如果某个配置器所分配的空间可由另一个配置器收回，我们称两者可互换。
- 如欲配合标准容器使用，针对相同型别而产生的配置器彼此必须可互换。所以此函数应当始终返回 true。

```
bool operator != (const allocator& a1, const allocator& a2)
```

- 如果配置器 $a1$ 和 $a2$ 并非可互换 (interchangeable)，就返回 true。
- 相当于 $!(a1 == a2)$ 。
- 如欲配合标准容器使用，针对相同型别而产生的配置器彼此必须可互换。所以此函数应当始终返回 false。

15.6 “未初始化内存”之处理工具细部讨论

本节详细阐述三个辅助函数，它们都应用于未初始化内存身上。这里的实作系以 Greg Colvin 提供的源码为基础，异常安全 (所谓 exception safe)，十分具有示范性。

```
void uninitialized_fill (ForwardIterator beg,
                        ForwardIterator end,
                        const T& value)
```

- 以 $value$ 初始化 $[beg; end)$ 区间内的元素
- 此函数要么成功，要么没有任何影响。
- 通常实作如下：

```
namespace std {
    template <class ForwIter, class T>
    void uninitialized_fill(ForwIter beg, ForwIter end,
                          const T& value)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(beg);
        try {
            for (; beg!=end; ++beg) {
                new (static_cast<void*>(&*beg)) VT(value);
            }
        }
    }
}
```

```

        catch (...) {
            for (; save!=beg; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}

```

```

void uninitialized_fill_n (ForwardIterator beg,
                          Size num,
                          const T& value)

```

- 以 *value* 初始化从 *beg* 开始的 *num* 个元素
- 此函数要么成功，要么没有任何影响。
- 通常实作如下：

```

namespace std {
    template <class ForwIter, class Size, class T>
    void uninitialized_fill_n (ForwIter beg, Size num,
                              const T& value)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(beg);
        try {
            for (; num-->0; ++beg) {
                new (static_cast<void*>(&*beg)) VT(value);
            }
        }
        catch (...) {
            for (; save!=beg; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}

```

- 参见 p730 的 `uninitialized_fill_n()` 运用实例。

```
ForwardIterator uninitialized_copy (InputIterator sourceBeg,
                                   InputIterator sourceEnd,
                                   ForwardIterator destBeg)
```

- 以[sourceBeg;sourceEnd]的元素为根据,将destBeg起始的元素加以初始化。
- 此函数要么成功,要么没有任何影响。
- 通常实作如下:

```
namespace std {
    template <class InputIter, class ForwIter>
    ForwIter uninitialized_copy(InputIter beg, InputIter end,
                               ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try {
            for (; beg!=end; ++beg,++dest) {
                new (static_cast<void*>(&*dest))VT(*beg);
            }
            return dest;
        }
        catch (...) {
            for (; save!=dest; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}
```

- 参见 p730 的 uninitialized_copy() 运用实例。

网络上的资源

Internet Resources

因特网 (Internet) 涵盖了大量关于本书主题的信息。下面是一份我的推荐名单, 你可以从中找到其它相关信息。

何处可得 C++ 标准规格书 (C++ standard)

美国国家标准委员会 (The American National Standards Institute, ANSI) 在美国当地出售 C++ 标准规格书。在我下笔此刻, 你可以从以下网址购买一份电子版, 要价 18 美元:

<http://www.ansi.org/>

新闻群组 (Newsgroups)

以下新闻群组专门讨论 C++、C++ 标准、C++ 标准程序库:

- C++ 一般议题 (无主持人)
`comp.lang.c++`
- C++ 一般议题 (有主持人)
`comp.lang.c++.moderated`
- 特别针对 C++ 标准规格 (有主持人)
`comp.std.c++`

欲更加了解这些新闻群组, 请看:

<http://reality.sgi.com/austern/std-c++/faq.html>

网址 (Internet Addresses/URLs)

以下列出的链接 (links) 提供了 C++ 标准程序库和 STL 的许多相关信息。然而, 毕竟, 书籍的寿命比网站的寿命长得多, 因此这些链接不一定永远有效。不过我总是会在以下网站提供它们的最新链接 (希望我的网站长命百岁):

<http://www.josuttis.com/libbook/>

以下链接 (links) 讨论整个 C++ 标准程序库:

- C++ 标准化常见问答集 (FAQs, frequently asked questions):
<http://reality.sgi.com/austern/std-c++/faq.html>
- C++ 标准化工作, ISO 工作小组官方主页:
<http://www.dkuug.dk/jtc1/sc22/wg21/>
- The Dinkum C++ Library Reference
<http://www.dinkumware.com/refxcpp.html>
- C++ 标准程序库, EGCS C++ 编译器实作版本
<http://sourceware.cygnum.com/libstdc++/>
- EGCS C++ 编译器
<http://egcs.cygnum.com/>
- Boost, 一个免费的、同僚复审的 (peer-reviewed) C++ 程序库
<http://www.boost.org/>
- Blitz++, 一个针对科学计算的 C++ class library
<http://www.oonumerics.org/blitz/>

以下链接 (links) 讨论 STL:

- SGI STL 版本 (免费)
<http://www.sgi.com/Technology/STL/>
- STLport (针对多个平台)
<http://www.stlport.org/>
- Mumit 的 STL Newbie Guide
<http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html>
- David Musser 的 STL site
<http://www.cs.rpi.edu/~musser/stl.html>
- STL FAQs (常见问答集)
<ftp://butler.hpl.hp.com/stl/stl.faq>
- Safe STL by Cay Horstmann
<http://www.horstmann.com/safestl.html>
- Warren Young 的 STL Resource List
<http://www.cyberport.com/~tangent/programming/stl/resources.html>

参考书目

Bibliography

下面这份名单，列出本书曾经提及、采纳、引用过的书籍，并列出可提供更多详细信息的书籍。请注意这并不是是一份面面俱到、无所不包的名单，这只是一份针对本书主题的个人阅读清单。

Matthew H. Austern

Generic Programming and the STL

— *Using and Extending the C++ Standard Template Library*

Addison-Wesley, Reading, MA, 1998

Ulrich Breymann

Komponenten entwerfen mit der STL

Addison-Wesley, Bonn, Germany, 1999

Bernd Eggink

Die C++ iostreams-Library

Hanser Verlag, München, Germany, 1995

Margaret A. Ellis, Bjarne Stroustrup

The Annotated C++ Reference Manual (ARM)

Addison-Wesley, Reading, MA, 1990

Graham Glass, Brett Schuchert

The STL <Primer>

Prentice-Hall, Englewood Cliffs, NJ, 1996

ISO

Information Technology — Programming Languages — C++

Document Number ISO/IEC 14882-1998

ISO/IEC, 1998

Scott Meyers

More Effective C++

— *35 New Ways to Improve Your Programs and Designs*

Addison-Wesley, Reading, MA, 1996

David R. Musser, Atul Saini

STL Tutorial and Reference Guide

— *C++ Programming with the Standard Template Library*

Addison-Wesley, Reading, MA, 1996

Mark Nelson

C++ Programmer's Guide to the Standard Template Library

IDG Books Worldwide, Foster City, CA, 1995

ObjectSpace

Systems<Toolkit> UNIX Reference Manual

ObjectSpace, 1995

P. J. Plauger

The Draft Standard C++ Library

Prentice Hall, Englewood Cliffs, NJ, 1995

Bjarne Stroustrup

The C++ Programming Language, 3rd edition

Addison-Wesley, Reading, MA, 1997

Bjarne Stroustrup

The Design and Evolution of C++

Addison-Wesley, Reading, MA, 1994

Steve Teale

C++ IOStreams Handbook

Addison-Wesley, Reading, MA, 1993

索引

Index

- ~ 599
 - for valarrays 573
- ! 601
 - for valarrays 573
- !=
 - derived from == 69
 - for allocators 740
 - for complex 537, 545
 - for containers 145, 234
 - for deque 162
 - for iterators 83, 252
 - for lists 168
 - for locales 700, 703
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for pairs 34
 - for queues 449
 - for sets 179
 - for stacks 440
 - for strings 511
 - for valarrays 573
 - for vectors 151
- %
 - for valarrays 573
- %=
 - for valarrays 574
- & 599
 - for bitsets 467
 - for valarrays 573
- &&
 - for valarrays 573
- &=
 - for valarrays 574
- ()
 - as operator 125
 - for locales 703
- *
 - for auto_ptrs 54
 - for complex 538, 544
 - for iterators 83, 252
 - for valarrays 573
- *=
 - for complex 538
 - for valarrays 574
- +
 - for complex 538, 544
 - for iterators 255
 - for strings 492, 524
 - for valarrays 573
- ++ 86
 - for iterators 83, 252, 258
 - for vector iterators 258
 - problem 258
- +=
 - for complex 538

- for iterators 255
- for strings 490, 515
- for valarrays 574
- - for complex 538, 544
 - for iterators 255
 - for valarrays 573
- 86
 - for iterators 255, 258
 - for vector iterators 258
- problem 258
- =
 - for complex 538
 - for iterators 255
 - for valarrays 574
- >
 - for auto_ptr 54
 - for iterators 91, 252
- /
 - for complex 538, 544
 - for valarrays 573
- /=
 - for complex 538
 - for valarrays 574
- <
 - for complex 538
 - for containers 145, 234
 - for deque 162
 - for iterators 255
 - for lists 168
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for pairs 34
 - for queues 449
 - for sets 179
 - for stacks 440
 - for strings 511
 - for valarrays 573
 - for vectors 151
- =
 - for deque 163
 - for iterators 83
 - for lists 168
 - for strings 489
 - for vectors 151
- ==
 - for allocators 728, 740
 - for complex 537, 545
 - for containers 145, 234
 - for deque 162
 - for valarrays 573
 - for vectors 151
- << 593, 652
 - conventions 662
 - for bitsets 462, 468
 - for complex 532, 539, 544
 - for stream buffers 597, 683
 - for strings 492, 524
 - for valarrays 573
- <<=
 - for valarrays 574
- <=
 - derived from < 69
 - for containers 145, 234
 - for deque 162
 - for iterators 255
 - for lists 168
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for pairs 34
 - for queues 449
 - for sets 179
 - for stacks 440
 - for strings 511
 - for valarrays 573
 - for vectors 151

- for iterators 83, 252
- for lists 168
- for locales 700, 703
- for maps 198
- for multimaps 198
- for multisets 179
- for pairs 34
- for queues 449
- for sets 179
- for stacks 440
- for strings 511
- for valarrays 573
- for vectors 151
- >
 - derived from < 69
 - for containers 145, 234
 - for deque 162
 - for iterators 255
 - for lists 168
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for pairs 34
 - for queues 449
 - for sets 179
 - for stacks 440
 - for strings 511
 - for valarrays 573
 - for vectors 151
- >=
 - derived from < 69
 - for containers 145, 234
 - for deque 162
 - for iterators 255
 - for lists 168
 - for maps 198
 - for multimaps 198
- for multisets 179
- for pairs 34
- for queues 449
- for sets 179
- for stacks 440
- for strings 511
- for valarrays 573
- for vectors 151
- >> 594, 652
 - conventions 662
 - for bitsets 468
 - for complex 532, 539, 544
 - for stream buffers 683
 - for strings 492, 524
 - for valarrays 573
- >>=
 - for valarrays 574
- []
 - for deque 162
 - for iterators 255
 - for maps 92, 205
 - for vectors 152
- ^
 - for bitsets 468
 - for valarrays 573
- ^=
 - for valarrays 574
- | 632
 - for bitsets 468
 - for valarrays 573
- |=
 - for valarrays 574
- ||
 - for valarrays 573

A

- `abort()` 72
- `abs()` 121
 - for complex 536, 543
 - for valarrays 574
 - global function 582
- absolute to relative values 331, 431
- `accumulate()` 425
- `acos()`
 - for valarrays 575
 - global function 581
- adapter
 - for containers 435
 - for functions
 - see function adapter
 - for member functions
 - see member function adapter
- address
 - input 596
 - output 596
- `address()`
 - for allocators 739
- address of the author 6
- `adjacent_difference()` 431, 432
- `adjacent_find()` 354
- `adjustfield` 618
- `adjustment` 618
- `advance()` 259, 282, 389
- `<algo.h>` 321, 425
- algorithm 74, 94, 321
 - absolute to relative values 331, 431
 - `accumulate()` 425
 - `adjacent_difference()` 431
 - `adjacent_find()` 354
 - auxiliary functions 332
 - `binary_search()` 410
 - call element member function 307
 - change order of elements 386
 - comparing 356
 - complexity 21
 - `copy()` 271, 363
 - copy and modify elements 367
 - `copy_backward()` 363
 - copy elements 363
 - `count()` 338
 - `count_if()` 338
 - counting elements 338
 - destination 111
 - `equal()` 356, 499
 - `equal_range()` 415
 - `fill()` 372
 - `fill_n()` 372
 - `find()` 341
 - `find_end()` 350
 - `find_first_of()` 352
 - `find_if()` 121, 341
 - `for_each()` 300, 334
 - for sorted ranges 330, 409
 - function as argument 119
 - `generate()` 373
 - `generate_n()` 373
 - header file 321
 - heap 406
 - `includes()` 411
 - `inner_product()` 427
 - `inplace_merge()` 423
 - intersection 419
 - `lexicographical_compare()` 360
 - `lower_bound()` 413
 - `make_heap()` 406
 - manipulating 111
 - `max_element()` 340
 - maximum 339
 - `merge()` 416

- `min_element()` 339
- `minimum` 339
- `mismatch()` 358
- `modify elements` 372
- `modifying` 111, 325, 363
- `multiple ranges` 101
- `mutating` 327, 386
- `next_permutation()` 391
- `nonmodifying` 323, 338
- `nth_element()` 404
- `numeric` 331, 425
- `overview` 322
- `partial_sort()` 400
- `partial_sort_copy()` 402
- `partial_sum()` 429
- `partition()` 395
- `pop_heap()` 407
- `prev_permutation()` 391
- `push_heap()` 406
- `random_shuffle()` 393
- `ranges` 97
- `relative to absolute values` 331, 429
- `remove()` 378
- `remove_copy()` 380
- `remove_copy_if()` 380
- `remove_if()` 378
- `removing` 326, 378
- `removing duplicates` 381
- `removing elements` 111
- `replace()` 375
- `replace_copy()` 376
- `replace_copy_if()` 376
- `replace_if()` 375
- `replacing elements` 375
- `result` 298
- `reverse()` 386
- `reverse_copy()` 386
- `rotate()` 388
- `rotate_copy()` 389
- `search()` 347, 499
- `searching elements` 324, 341
- `search_n()` 344
- `set complement` 421
- `set difference` 420
- `set_difference()` 420
- `set_intersection()` 419
- `set_symmetric_difference()` 420, 421
- `set_union()` 418
- `sort()` 397
- `sort_heap()` 407
- `sorting` 328, 397
- `stable_partition()` 395
- `stable_sort()` 397
- `suffix_copy` 323
- `suffix_if` 323
- `swapping` 370
- `swap_ranges()` 370
- `transform()` 120, 131, 367, 368, 497
- `transform elements` 367
- `union elements` 418
- `unique()` 381
- `unique_copy()` 384
- `upper_bound()` 413
- `user-defined` 285
- `<algorithm>` 33, 66, 96, 321
- `algostuff.hpp` 332
- `alias free` 547
- `allocate()`
 - for allocators 728, 739
- `allocator` 31, 727
 - != 740
 - == 728, 740
- `address()` 739
- `allocate()` 728, 739

- `const_pointer` 738
- `const_reference` 738
- `construct()` 728, 739
- `constructor` 739
- `deallocate()` 728, 739
- `default` 32, 732
- `destroy()` 728, 740
- `destructor` 739
- `difference_type` 738
- `get_allocator()` 728
- `max_size()` 739
- `pointer` 738
- `rebind` 734, 738
- `reference` 738
- `size_type` 738
- `usage` 727
- `user-defined` 735
- `value_type` 737
- `allocator_type`
 - for containers 231, 247
 - for strings 526
- `alnum`
 - for `ctype_base` 717
- `alpha`
 - for `ctype_base` 717
- `always_noconv()`
 - for `codecvt facet` 721
- `ambiguous`
 - numeric functions 581
- `amortized complexity` 22
- `antisymmetric` 176
- `any()`
 - for bitsets 464
- `app flag` 631
- `append()`
 - for strings 490, 515, 516
- `apply()`
 - for `valarrays` 572
- `arg()`
 - for complex 536, 543
- `argc` 21, 633
- `argument_type` 310
- `argv` 21, 633
- `arithmetic`
 - of iterators 255
- `array`
 - associative 92, 205
 - as STL container 218
 - container 155
 - wrapper 219
- `asin()`
 - for `valarrays` 575
 - global function 581
- `assign()`
 - for `char_traits` 689
 - for containers 236, 237
 - for deques 163
 - for lists 168
 - for strings 489, 514
 - for vectors 151
- `assignable` 135
- `assignment`
 - for containers 147
 - for deques 163
 - for iterators 83
 - for lists 168
 - for vectors 151
- `associative array` 92, 205, 207
- `associative container` 75
 - manipulating access 115
 - sorting criterion 538
 - user-defined inserter 288
- `at()`
 - for containers 237
 - for deques 162, 237

- for strings 513
- for vectors 152, 237

atan()

- for valarrays 575
- global function 581

atan2()

- for valarrays 575
- global function 581

ate flag 631

atexit() 72

author 6

auto pointer

- see auto_ptr

auto_ptr 38

- * 54

- > 54

- = 41

- assignment operator 41

- constructor 52, 53

- conversions 55

- destructor 53

- element_type 52

- get() 54

- header file 51

- implementation 56

- initialization 40

- release() 54

- reset() 54

auto_ptr_ref 51, 55

B

back()

- for containers 238

- for deques 162

- for lists 168

- for queues 449

- for vectors 152

back inserter 106, 272

back_inserter 106, 272

bad()

- for streams 598

bad_alloc 25, 26

badbit 597

bad_cast 25, 26

bad_exception 25, 26

bad_typeid 25, 26

base() 269

basefield 621

basic_filebuf 627, 643

- for streams 643

basic_fstream 627

basic guarantee 139

basic_ifstream 627

basic_ios 588

basic_istream 588

- see input stream

basic_istreamstream 645

basic_ofstream 627

basic_ostream 588

- see output stream

basic_ostreamstream 645

basic_streambuf 588, 668

- see output buffer and input buffer

basic_string 471

- see string

basic_stringbuf 645

basic_stringstream 645

beg 635

begin()

- for containers 83, 145, 239

- for deques 162

- for lists 169

- for maps 200

- for multimaps 200

- for multisets 182

- for sets 182

- for strings 497, 525
 - for vectors 153
- bibliography 745
- bidirectional iterator 93, 255
 - distance 261
 - step backward 259
 - step forward 259
- bidirectional_iterator 288
- Big-O notation 21
- binary flag 631
- binary_function 310
- binary predicate 123
- binary representation 462
- binary_search() 410
- bind1st 306, 311
- bind2nd 132, 133, 306, 311, 338
- bitfield
 - with dynamic size 158
 - see vector<bool>
 - with static size 460
 - see bitset
- bitset 460
 - & 467
 - << 462, 468
 - >> 468
 - ^ 468
 - | 468
 - any() 464
 - binary representation 462
 - constructor 463, 464
 - count() 464
 - examples 460
 - flip() 465, 466
 - header file 460
 - input 468
 - none() 464
 - output 468
 - reference 467

- reset() 465
 - set() 465
 - size() 464
 - test() 465
 - to_string() 468
 - to_ulong() 462, 468
- <bitset> 460
- BLAS 548
- Blitz 547, 744
- books 745
- bool
 - input 595, 617
 - input format 617
 - numeric limits 60
 - output 595, 617
 - output format 617
- bool type 18
- boolalpha flag 617
- boolalpha manipulator 617
- Boolean conditions
 - in loops 600
 - of streams 600
- Boolean vector 158
- Boost 225, 313, 744
- buffer
 - see output buffer and input buffer

C

- "C" locale 694
- callback
 - for streams 661
- capacity
 - of strings 485
 - of vectors 149
- capacity()
 - for containers 233
 - for strings 486, 510
 - for vectors 149

- cararray 219
- catalog
 - for message_base 725
- category
 - of container iterators 239
 - of iterators 93, 251
- category
 - for locales 700
- <cctype> 720
- ceil()
 - global function 581
- cerr 585, 592
 - redirecting 641
- <cfloat> 59, 60
- char
 - classification 716
 - input 595
 - numeric limits 60
- char*
 - input 596
- char* stream 649
 - freeze() 650
 - str() 650
- character
 - classification 716
 - encoding conversion 720
 - traits 504, 687
- char_traits 479, 590, 687
 - assign() 689
 - char_type 689
 - compare() 504, 689
 - copy() 689
 - eof() 689
 - eq() 504, 689
 - eq_int_type() 689
 - find() 504, 689
 - fint_type 689
 - length() 689
 - lt() 504, 689
 - move() 689
 - not_eof() 689
 - off_type 689
 - pos_type 689
 - state_type 689
 - to_char_type() 689
 - to_int_type() 689
- char_type
 - for char_traits 689
- cin 585, 592
 - redirecting 641
- class
 - auto_ptr 38
 - bad_alloc 25
 - bad_cast 25
 - bad_exception 25
 - bad_typeid 25
 - basic_filebuf 627, 643
 - basic_fstream 627
 - basic_ifstream 627
 - basic_ios 588
 - basic_istream 588
 - basic_istreamstream 645
 - basic_ofstream 627
 - basic_ostream 588
 - basic_ostreamstream 645
 - basic_streambuf 588, 668
 - basic_string 471
 - basic_stringbuf 645
 - basic_stringstream 645
 - bitset 460
 - see bitset
 - cararray 219
 - codecvt 720
 - codecvt_base 721
 - collate 724
 - complex 529

- ctype 716
- ctype_base 717
- deque
 - see deque
- domain_error 25
- exception 25
- facet 700
- failure 25, 602
- filebuf 627, 643
- fpos 634
- fstream 627
- gslice 560, 577
- gslice_array 561, 577
- hash_map 221
- hash_multimap 221
- hash_multiset 221
- hash_set 221
- ifstream 627, 629
- indirect_array 567
- invalid_argument 25
- I/O functions** 652
- ios 590
- ios_base 588
- ios_base::failure 25, 602
- iostream 591
- iostream_withassign 591
- istream 584, 591
- istreambuf_iterator 665
- istream_withassign 591
- istreamstream 645
- istrstream 649
- iterator 288
- length_error 25
- list
 - see list
- locale 694
- logic_error 25
- map
 - see map
- mask_array 564
- message_base 725
- messages 725
- money_base 712
- money_get 715
- moneypunct 711
- money_put 713
- multimap
 - see multimap
- multiset
 - see multiset
- numeric_limits 59
- num_get 707
- numpunct 705
- num_put 706
- ofstream 627, 629
- ostream 585, 591
- ostreambuf_iterator 665
- ostream_withassign 591
- ostreamstream 645
- ostrstream 649
- out_of_range 25
- overflow_error 25
- priority_queue 453
- queue 444
- range_error 25
- runtime_error 25
- sentry 658
- set
 - see set
- slice 555, 575
- slice_array 556, 575
- stack 435
- streambuf 590, 668
- string 471
- stringbuf 645

- stringstream 645
- strstream 649
- strstreambuf 649
- time_base 709
- time_get 708
- time_put 710
- underflow_error 25
- valarray 547
- vector
 - see vector
- wfilebuf 627
- wfstream 627
- wifstream 627
- wios 590
- wiostream 591
- wistream 591
- wistringstream 645
- wofstream 627
- wostream 591
- wostreamstream 645
- wstreambuf 590, 668
- wstring 471
- wstringbuf 645
- wstringstream 645
- class hierarchy
 - of exceptions 25
 - of file stream classes 627
 - of stream classes 588
 - of string stream classes 645
- classic()
 - for locales 694, 700, 703
- classic_table()
 - for ctype facet 718
- clear()
 - for containers 145, 244
 - for deques 163
 - for lists 170
 - for maps 202
 - for multimaps 202
 - for multisets 183
 - for sets 183
 - for streams 598, 599, 633, 636
 - for strings 490, 518
 - for vectors 154
- <climits> 59, 60
- clog 585
- close()
 - for messages facet 725
 - for streams 633
- <cmath> 581
- cntrl
 - for ctype_base 717
- codecvt facet 720
 - always_noconv() 721
 - encoding() 721
 - in() 721
 - length() 721
 - max_length() 721
 - out() 721
 - unshift() 721
- codecvt_base 721
 - error 722
 - noconv 722
 - ok 722
 - partial 722
 - result 722
- collate facet 724
 - compare() 724
 - hash() 724
 - transform() 724
- collate locale category 724
- collection 73
 - of collections 360
- combine()
 - for locales 700, 702
- command-line arguments 21, 633

- commit-or-rollback 139
- compare
 - lexicographical 360
 - ranges 356
- compare()
 - for char_traits 504, 689
 - for collate facet 724
 - for strings 511, 512
- comparison operators 69
 - for containers 147
- compiler requirements 9
- complementary set 421
- complex 529
 - != 537, 545
 - * 538, 544
 - *= 538
 - + 538, 544
 - += 538
 - 538, 544
 - = 538
 - / 538, 544
 - /= 538
 - < 538
 - << 532, 539, 544
 - == 537, 545
 - >> 532, 539, 544
 - abs() 536, 543
 - and associative containers 537, 538
 - arg() 536, 543
 - conj() 533, 541
 - constructor 533, 541
 - cos() 540, 546
 - cosh() 540, 546
 - examples 530
 - exp() 540, 545
 - header file 529
 - imag() 536, 543
 - I/O 532, 539, 544
 - log() 540, 545
 - log10() 540, 545
 - norm() 536, 543
 - polar() 533, 541
 - pow() 540, 545
 - read 532, 544
 - reading 539
 - real() 536, 543
 - sin() 540, 546
 - sinh() 540, 546
 - sqrt() 540, 545
 - tan() 540, 546
 - tanh() 540, 546
 - type conversions 534
 - value_type 541
 - write 532, 544
 - writing 539
- <complex> 529
- complexity 21
 - amortized 22
- compose1 314
- compose1 313
- compose2 316
- compose2 313
- compose_f_gx 313, 314
- compose_f_gx_hx 313, 316
- compose_f_gx_hy 313, 318
- compose_f_gxy 313
- compose function object 313
- compressing whitespaces 385
- conditions
 - in loops 600
- conj()
 - for complex 533, 541
- constant
 - EXIT_FAILURE 72
 - EXIT_SUCCESS 72
 - NULL 71

- constant complexity 21
- const_cast 20
- const_iterator
 - for containers 85, 230
 - for strings 508
- const_mem_fun1_ref_t 309
- const_mem_fun1_t 309
- const_mem_fun_ref_t 309
- const_mem_fun_t 309
- const_pointer
 - for allocators 738
 - for strings 508
- const_reference
 - for allocators 738
 - for containers 230
 - for strings 507
- const_reverse_iterator
 - for containers 230
 - for strings 508
- construct()
 - for allocators 728, 739
- constructor
 - as template 13
 - for allocators 739
 - for auto_ptrs 52, 53
 - for bitsets 463, 464
 - for complex 533, 541
 - for containers 231, 232, 247
 - for deques 162
 - for lists 167
 - for locales 700, 702
 - for maps 196
 - for multimaps 196
 - for multisets 177, 189
 - for pairs 34
 - for priority queues 458, 459
 - for queues 448
 - for sets 177
 - for stacks 440
 - for strings 501, 508, 509
 - for valarrays 548, 570
 - for vectors 150
- contact with the author 5
- container 73, 75, 143, 435
 - != 145, 234
 - < 145, 234
 - <= 145, 234
 - == 145, 234
 - > 145, 234
 - >= 145, 234
- adapters 435
- allocator_type 231, 247
- assign() 236, 237
- assignment 147
- at() 237
- back() 238
- begin() 83, 145, 239
- call member function for elements
 - 134, 307
- capacity() 233
- clear() 145, 244
- comparison 226
- comparison operators 147
- const_iterator 85, 230
- const_reference 230
- const_reverse_iterator 230
- constructor 231, 232, 247
- count() 234
- deque
 - see deque
- destructor 231, 232
- difference_type 231
- element requirements 134
- empty() 145, 146, 233
- end() 83, 145, 239
- equal_range() 236

- `erase()` 242, 243
- exceptions handling overview 248
- `find()` 235
- `front()` 238
- `get_allocator()` 247
- initialization 144
- `insert()` 240, 241
- iterator 85
- iterator 230
- iterator category 239
- `key_comp()` 236
- `key_compare` 231
- `key_type` 231
- list
 - see list
- `lower_bound()` 235
- map
 - see map
- `mapped_type` 231
- `max_size()` 145, 146, 233
- multimap
 - see multimap
- multiset
 - see multiset
- of containers 360
- ordinary arrays 218
- `pop_back()` 243
- `pop_front()` 243
- print elements 118
- `push_back()` 241
- `push_front()` 241
- `rbegin()` 109, 145, 239
- reference 230
- reference counting 222
- reference semantics 135
- `rend()` 109, 145, 240
- `reserve()` 233
- `resize()` 244, 248
- `reverse_iterator` 230
- set
 - see set
- `size()` 145, 146, 233
- size operations 146
- `size_type` 231
- `swap()` 140, 145, 147, 237
- swapping 147
- `upper_bound()` 235
- user-defined 217
- `value_comp()` 236
- `value_compare` 231
- value semantics 135
- `value_type` 230
- vector
 - see vector
- `container_type`
 - for priority queues 458
 - for queues 448
 - for stacks 439
- conversion
 - absolute to relative values 331, 431
 - between character encodings 720
 - relative to absolute values 331, 429
- `copy()`
 - algorithm 363
 - algorithm implementation 271
 - for `char_traits` 689
 - for strings 484, 513
- copyable 134
- copy and modify elements 367
- copy and replace elements 376
- `copy_backward()` 363
- copy constructor
 - as template 13, 35
- `copyfmt()`
 - for streams 615, 616, 641, 653, 661
- `copyfmt_event` 661

- copying elements 363
 - cos()
 - for complex 540, 546
 - for valarrays 575
 - global function 581
 - cosh()
 - for complex 540, 546
 - for valarrays 575
 - global function 581
 - count() 338
 - for bitsets 464
 - for containers 234
 - for maps 198
 - for multimaps 198
 - for multisets 180
 - for sets 180
 - CountedPtr 222
 - count_if() 338
 - counting elements 338
 - cout 585, 592
 - redirecting 641
 - cshift()
 - for valarrays 572
 - <cstddef> 71
 - <cstdlib> 71, 581
 - c_str()
 - for strings 484, 513, 629
 - C-string 471
 - <cstring> 689
 - <ctime> 708
 - ctype facet 716
 - classic_table() 718
 - is() 716
 - narrow() 716
 - scan_is() 716
 - scan_not() 716
 - table() 718
 - table_size 718
 - tolower() 716
 - toupper() 716
 - widen() 716
 - ctype locale category 715
 - ctype_base 717
 - alnum 717
 - alpha 717
 - cntrl 717
 - digit 717
 - graph 717
 - lower 717
 - mask 717
 - print 717
 - punct 717
 - space 717
 - upper 717
 - xdigit 717
 - <ctype.h> 720
 - cur 635
 - curr_symbol()
 - for moneypunct facet 712
- ## D
- data()
 - for strings 484, 513
 - data type
 - see type
 - dateorder
 - for time_base 709
 - date_order()
 - for time_get facet 708
 - deallocate()
 - for allocators 728, 739
 - dec flag 621
 - dec manipulator 622
 - decimal_point()
 - for moneypunct facet 712
 - for numpunct facet 705

decimal representation 621**default**

allocator 32, 732

template parameter 10

denorm_absent 64

denorm_indeterminate 64

denorm_min()

for numeric limits 62

denorm_present 64

deque 160, 164

see container

!= 162

< 162

<= 162

= 163

== 162

> 162

>= 162

[] 162

assign() 163

assignment 163

at() 162

back() 162

begin() 162

clear() 163

constructor 162

empty() 162

end() 162

erase() 163

exception handling 164

front() 162

header file 160

insert() 163

max_size() 162

member functions 162

operations 162

pop_back() 163

pop_front() 163

push_back() 163

push_front() 163

rbegin() 162

rend() 162

resize() 163

size() 162

subscript operator 162

swap() 163

<deque> 160

<deque.h> 160

destination of algorithms 111

destroy()

for allocators 728, 740

destroyable 135

destructor

for allocators 739

for auto_ptrs 53

for containers 231, 232

for lists 167

for locales 700

for maps 196

for multimaps 196

for multisets 177

for sets 177

for strings 509

for valarrays 570

for vectors 150

dictionary 209

difference of two sets 420

difference_type

for allocators 738

for containers 231

for strings 507

digit

for ctype_base 717

digits

for numeric limits 61, 462

- `digits10`
 - for numeric limits 61
- `distance()` 261, 287
- `div()`
 - global function 582
- `divides` 305
- `dmy` date order 710
- `domain_error` 25, 28
- `double`
 - numeric limits 60
- `duplicates removing` 381
- `dynamic array container` 155
- `dynamic_cast` 19, 26
- E**
- `eback()`
- for input buffers 676
- EGCS 5, 744
- `egptr()`
 - for input buffers 676
- `element access`
 - for lists 168
 - for vectors 152
- `element_type`
 - for `auto_ptrs` 52
- `email` 5
- `empty()`
 - for containers 145, 146, 233
 - for deques 162
 - for lists 168
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for priority queues 459
 - for queues 449
 - for sets 179
 - for stacks 440
 - for strings 485, 510
 - for vectors 151
- empty range 84
- encoding
 - conversion 720
- `encoding()`
- for `codecvt` facet 721
- `end` 635
- `end()`
 - for containers 83, 145, 239
 - for deques 162
 - for lists 169
 - for maps 200
 - for multimaps 200
 - for multisets 182
 - for sets 182
 - for strings 497, 525
 - for vectors 153
- `endl` manipulator 586, 587, 612, 683
- end-of-file 590
- end-of-stream iterator 108, 280
- `ends` manipulator 587, 612, 650
- environment variable
 - LANG 692
- EOF 590, 597
 - internationalized 689
- `eof()`
 - for `char_traits` 689
 - for streams 598
- `eofbit` 597
- `epptr()`
 - for output buffers 668
- `epsilon()`
 - for numeric limits 61
- `eq()`
 - for `char_traits` 504, 689
- `eq_int_type()`
 - for `char_traits` 689

- `equal()` 356, 499
 - for `istreambuf_iterator` 666
- `equal_range()` 415
 - for containers 236
 - for maps 198
 - for `multimaps` 198
 - for `multisets` 180
 - for sets 180
- `equal_to` 132, 305, 319
- `erase()`
 - for containers 185, 242, 243
 - for `deque`s 163
 - for lists 170
 - for maps 202
 - for `multimaps` 202
 - for `multisets` 182, 189
 - for sets 182, 186
 - for strings 491, 501, 518
 - for vectors 154
- `erase_event` 661
- `error`
 - for `codecvt_base` 722
- error handling 25
 - in the STL 137
- even element 306
- event 661
- `event_callback` 661
- example code 5
 - auxiliary functions 332
- exception 29
 - `bad_alloc` 25
 - `bad_cast` 25
 - `bad_exception` 25
 - `bad_typeid` 25
 - classes 25
 - declaration 16
 - deriving 30
 - `domain_error` 25
 - exception 25
 - failure 25
 - header files 28
 - `invalid_argument` 25
 - `ios_base::failure` 25, 602
 - `length_error` 25
 - `logic_error` 25
 - members 28
 - `out_of_range` 25
 - `overflow_error` 25
 - `range_error` 25
 - `runtime_error` 25
 - specification 16, 26
 - throw 16
 - `underflow_error` 25
 - user-defined 29, 441, 450
 - `what()` 28, 29
- exception 25
- `<exception>` 28
- exception handling 15, 25
 - `auto_ptr` 38
 - for `deque`s 164
 - for lists 172
 - for maps 207
 - for `multimaps` 207
 - for `multisets` 185
 - for sets 185
 - for vectors 155
 - in the STL 139, 248
- exceptions()
 - for streams 602
- exception safety 139
- exception specification 16, 26
- `exit()` 72
- `EXIT_FAILURE` constant 72
- `EXIT_SUCCESS` constant 72
- `exp()`
 - for complex 540, 545

- for valarrays 575
 - global function 581
- explicit 18
- export 10
- extending STL 141
- extractor
 - for streams 586, 593

F

- fabs()
 - global function 581
- facet 698, 704
 - codecvt 720
 - collate 724
 - ctype 716
 - for character classification 716
 - for character encoding conversion 720
 - for date formatting 708
 - for internationalized messages 725
 - for monetary formatting 711
 - for numeric formatting 705
 - for string collation 724
 - for time formatting 708
- id 704
- messages 725
- money_get 715
- money_punct 711
- money_put 713
- num_get 707
- num_punct 705
- num_put 706
- time_get 708
- time_put 710
- type 700

- fail()
 - for streams 598
- failbit 597

- failed()
 - for ostreambuf_iterator 666
- failure 25, 602
- false 18
- falsename()
 - for numpunct facet 705
- feedback to the author 5
- field width 618
- file 627
 - access 627, 629
 - opening 627
 - positioning 634
 - read and write 643
- filebuf 627, 643
- file descriptor 637, 672
- fill() 372
 - for streams 618
 - for valarrays 571
- fill character 618
- fill_n() 372
- filter 601, 611
- find()
 - algorithm 341
 - finding subrange 99
 - for char_traits 504, 689
 - for containers 235
 - for maps 198
 - for multimaps 198
 - for multisets 180, 189
 - for sets 180, 186
 - for strings 520, 521
 - return value 99
- find_end() 350
- find_first_not_of()
 - for strings 522
- find_first_of()
 - algorithm 352
 - for strings 522

- `find_if()` 121, 211, 341
- finding algorithms** 324, 341
- `find_last_not_of()`
 - for strings 523
- `find_last_of()`
 - for strings 523
- find limit** 343
- first**
 - for pairs 34
- `first_argument_type` 310
- `first_type`
 - for pairs 34
- fixed flag** 623
- fixed manipulator** 625
- `flags()` 539
 - for streams 615, 616
- `flip()`
 - for bitsets 465, 466
 - for `vector<bool>::reference` 158, 159
- `float`
 - numeric limits 60
- `float_denorm_style` 63
- `floatfield` 623
- `<float.h>` 59, 60
- floating-point I/O formats** 623
- `float_round_style` 63
- `floor()`
 - global function 581
- `flush()`
 - for output streams 611
- flush manipulator** 587, 612, 683
- `fmod()`
 - global function 581
- `for_each()` 300, 334
 - return value 300
 - versus `transform()` 325
- format flags** 615
- formatted I/O** 615
- formatting**
 - of bool 595, 617
 - of floating-point values 623
- forward iterator** 254
 - distance 261
 - step forward 259
 - versus output iterator 254
- `forward_iterator` 288
- `fpos` 634
- `frac_digits()`
 - for `money_punct` facet 712
- `freeze()` 650, 651
- `frexp()`
 - global function 581
- `front()`
 - for containers 238
 - for deques 162
 - for lists 168
 - for queues 449
 - for vectors 152
- front inserter** 106, 274
- `front_inserter` 106, 272
- `fstream` 627
- `<fstream>` 628
- function**
 - as argument 119
 - as sorting criterion 123
- function adapter** 132, 306
 - `bind1st` 306, 311
 - `bind2nd` 306, 311
 - `compose1` 314
 - `compose1` 313
 - `compose2` 316
 - `compose2` 313
 - `compose_f_gx` 313, 314
 - `compose_f_gx_hx` 313, 316
 - `compose_f_gx_hy` 313, 318

compose_f_gxy 313
mem_fun 307
mem_fun_ref 307
not1 306
not2 306
ptr_fun 310
<functional> 305, 306, 321
functional composition 134, 313
<function.h> 305, 321
function object 124, 213, 293
 as sorting criterion 294
 bind1st 306, 311
 bind2nd 306, 311
 compose1 314
 compose1 313
 compose2 316
 compose2 313
 compose_f_gx 313, 314
 compose_f_gx_hx 313, 316
 compose_f_gx_hy 313, 318
 compose_f_gxy 313
 divides 305
 equal_to 305
 greater 305
 greater_equal 305
 header file 305
 less 305
 less_equal 305
 logical_and 305
 logical_not 305
 logical_or 305
 mem_fun 307, 308
 mem_fun_ref 307
 minus 305
 modulus 305
 multiplies 305
 negate 305
 not1 306

not2 306
not_equal_to 305
plus 305
predefined 131
ptr_fun 310
state 296
times 305
user-defined 310
functor 124, 293

G

gcount()
 for input streams 609
general inserter 275
general slice 560
 see gslice
generate() 373
generate_n() 373
get()
 for auto_ptrs 54
 for input streams 608, 629
 for messages facet 725
 for money_get facet 715
 for num_get facet 707
get_allocator() 728
 for containers 247
 for strings 526
get buffer 676
 iterator 666
get_date()
 for time_get facet 708
getline()
 for input streams 608
 for strings 493, 525
getloc()
 for stream buffers 664, 669
 for streams 626

- get_monthname()
 - for time_get facet 708
 - get_temporary_buffer() 731
 - get_time()
 - for time_get facet 708
 - get_weekday()
 - for time_get facet 708
 - get_year()
 - for time_get facet 708
 - global()
 - for locales 697, 700, 703
 - good() 601
 - for streams 598
 - goodbit 597
 - gptr()
 - for input buffers 676
 - graph
 - for ctype_base 717
 - greater 186, 305, 338
 - greater_equal 305
 - grouping()
 - for moneypunct facet 712
 - for numpunct facet 705
 - gslice 560, 577
 - constructor 577
 - size() 577
 - start() 577
 - stride() 577
 - gslice_array 561, 577
- ## H
- half-open range 84, 97
 - hardfail 598
 - has_denorm
 - for numeric limits 62, 63
 - has_denorm_loss
 - for numeric limits 62
 - has_facet()
 - for locales 700, 702
 - hash()
 - for collate facet 724
 - hash_map 221
 - hash_multimap 221
 - hash_multiset 221
 - hash_set 221
 - hash table 221
 - has_infinity
 - for numeric limits 61
 - has_quiet_NaN
 - for numeric limits 61
 - has_signaling_NaN
 - for numeric limits 61
 - header file 24
 - "alghostuff.hpp" 332
 - <algo.h> 321, 425
 - <algorithm> 33, 66, 96, 321
 - <bitset> 460
 - <cctype> 720
 - <cfloat> 59, 60
 - <climits> 59, 60
 - <cmath> 581
 - <complex> 529
 - <cstdint> 71
 - <cstdlib> 71, 581
 - <cstring> 689
 - <ctime> 708
 - <ctype.h> 720
 - <deque> 160
 - <deque.h> 160
 - <exception> 28
 - extension 24
 - <float.h> 59, 60
 - for algorithms 321
 - for auto_ptrs 51
 - for bitsets 460

- for complex 529
 - for deques 160
 - for exceptions 28
 - for function objects 305
 - for I/O 592
 - for lists 166
 - for maps 194
 - for multimaps 194
 - for multisets 175
 - for priority queues 454
 - for queues 444
 - for sets 175
 - for stacks 435
 - for streams 592
 - for strings 479
 - for valarrays 547
 - for vectors 148
 - <fstream> 628
 - <functional> 305, 306, 321
 - <function.h> 305, 321
 - <iomanip> 614
 - <ios> 28, 607
 - <iosfwd> 592
 - <iostream> 592, 681
 - <iostream.h> 25
 - <istream> 592
 - <iterator> 251, 665
 - <iterator.h> 251
 - <limits> 60
 - <limits.h> 59, 60
 - <list> 166
 - <list.h> 166
 - <locale> 700
 - <map> 194
 - <map.h> 194
 - <memory> 51
 - <multimap.h> 194
 - <multiset.h> 175
 - <new> 28
 - <numeric> 321, 425
 - <ostream> 592
 - <queue> 444, 454
 - <Queue.hpp> 450
 - <set> 175
 - <set.h> 175
 - <sstream> 646
 - <stack> 435
 - <stack.h> 435, 444, 454
 - <Stack.hpp> 441
 - <stddef.h> 71
 - <stdexcept> 28
 - <stdlib.h> 71
 - <streambuf> 592
 - <string> 479, 688
 - <string.h> 689
 - <stringstream> 649
 - <typeinfo> 28
 - <utility> 33, 34, 69
 - <valarray> 547
 - <vector> 148
 - <vector.h> 148
 - heap algorithms 406
 - heapsort 329, 406
 - hex flag 621
 - hex manipulator 622
 - hexadecimal representation 621
- ## I
- i18n
 - see internationalization
 - id
 - for locales 700
 - ifstream 627, 629
 - ignore()
 - for input streams 532, 609, 614
 - for streams 614

- `imag()`
 - for complex 536, 543
- `imbue()` 539, 694
 - for streams 626, 694
- `imbue_event` 661
- implementation
 - of manipulators 614
- `in()`
 - for `codecvt` facet 721
- in flag 631
- `in_avail()`
 - for input buffers 663
- include file
 - see header file
- `includes()` 411
- increment
 - for iterators 83
- index of the book 747
- index operator
 - for iterators 255
 - for maps 92, 205
- indirect array 567
- `indirect_array` 567, 579
- `infinity()`
 - for numeric limits 61
- initialize
 - a container 144
- `inner_product()` 427
- `inplace_merge()` 423
- input 583
 - see input stream and stream
 - defining numeric bases 621
 - field width 620
 - hexadecimal 621
 - line-by-line 493
 - octal 621
 - of addresses 596
 - of bitsets 468
 - of bool 595, 617
 - of char 595
 - of `char*` 596
 - of complex 532, 539, 544
 - of objects in a loop 600
 - of strings 492, 524, 620
 - of `void*` 596
 - of `wchar_t` 595
 - operator `>>` 594
 - redirecting 641
 - skip input 282
 - standard functions 607
- input buffer 676
 - `eback()` 676
 - `egptr()` 676
 - `gptr()` 676
 - `in_avail()` 663
- iterator 666
 - `pbackfail()` 677
 - `sbumpc()` 663, 676
 - `setg()` 677
 - `sgetc()` 663, 676
 - `sgetn()` 663, 677
 - `snextc()` 663
 - `sputbackc()` 663, 676
 - `sungetc()` 663, 676
 - `uflow()` 676
 - `underflow()` 676
 - `xsgetn()` 677
- input iterator 252
 - distance 261
 - step forward 259
- `input_iterator` 288
- input stream
 - buffer iterators 665
 - buffers 663
 - `gcount()` 609
 - `get()` 608

- getline() 608
- ignore() 532, 609, 614
- iterator 107, 280
- member functions 607
- peek() 609
- putback() 609
- read() 608
- readsome() 609
- sentry 658
- unget() 609
- insert()
 - called by inserters 272
 - for containers 240, 241
 - for deques 163
 - for lists 170
 - for maps 202, 203
 - for multimaps 202, 203
 - for multisets 182, 183, 184, 189
 - for sets 182, 183, 184, 186
 - for strings 491, 516, 517
 - for vectors 154
- INSERT_ELEMENTS() 332
- inserter 104, 106, 271, 272, 275
 - for streams 586, 593
 - user-defined 288
- insert iterator 104, 271
- int
 - numeric limits 60
- internal flag 618
- internal manipulator 619
- internationalization 685
 - of EOF 689
 - of I/O 625
 - of special characters 691
- Internet 5, 743
- intersection 419
- introsort 329
- intrusive approach 217
- int_type
 - for char_traits 689
- invalid_argument 25, 27
- invasive approach 217
- I/O 583
 - see input, output, and stream
 - binary representation 462
 - classes 588
 - file access 627
 - filter framework 601, 611
 - for classes 652
 - for complex 532, 539, 544
 - formatted 615
 - header files 592
 - in C++ 587
 - internationalization 625
 - manipulators 586, 612
 - operators 593
 - overloading operators 652
 - redirecting standard streams 641
 - user-defined stream buffers 668
 - with C++ 611
 - with streams 587
- <iomanip> 614
- ios
 - see stream
- ios 590
- <ios> 28, 607
- ios_base 588
 - see stream
- ios_base::failure 25, 28, 602
- <iosfwd> 592
- iostream 591
- <iostream> 592, 681
- <iostream.h> 25
- ostream_withassign 591
- ipfx() 658
- irreflexive 176

- `is()`
 - for ctype facet 716
- `isalnum()` 719
- `isalpha()` 719
- `is_bounded`
 - for numeric limits 61
- `iscntrl()` 719
- `isdigit()` 719
- `is_exact`
 - for numeric limits 61
- `isfx()` 658
- `isgraph()` 719
- `is_iec559`
 - for numeric limits 61
- `is_integer`
 - for numeric limits 61
- `islower()` 719
- `is_modulo`
 - for numeric limits 61
- `is_open()`
 - for streams 633
- `isprint()` 719
- `ispunct()` 719
- `is_signed`
 - for numeric limits 61
- `isspace()` 502, 719
- `is_specialized`
 - for numeric limits 61
- `istream`
 - see input stream and stream
- `istream` 584, 591
- `<istream>` 592
- `istreambuf_iterator` 665, 666
 - `equal()` 666
- `istream iterator` 107, 280
 - end-of-stream 108, 280
 - skip input 282
- `istream_withassign` 591
- `istringstream` 645
- `istrstream` 649
- `isupper()` 719
- `isxdigit()` 719
- `iterator` 74, 83, 251
 - `!=` 83, 252
 - `*` 83, 252
 - `+` 255
 - `++` 83, 86, 252, 258
 - `+=` 255
 - `+=` versus `advance()` 260
 - `-` 255
 - `--` 86, 255, 258
 - `--` 255
 - `->` 91, 252
 - `<` 255
 - `<=` 255
 - `=` 83
 - `==` 83, 252
 - `>` 255
 - `>=` 255
 - `[]` 255
- adapters 104, 264
- `advance()` 259, 389
- arithmetic 255
- assignment 83
- auxiliary functions 259
- `back_inserter` 106, 272
- back inserters 272
- bidirectional 93, 255
- categories 93, 251
- check order 99
- convert into reverse iterator 265
- `distance()` 261
- end-of-stream 108, 280
- for containers 85
- for lists 169
- for maps 200

- for multimaps 200
- for multisets 182, 189
- for sets 182
- for stream buffers 665
- for streams 107, 277, 282
- for strings 497
- for vectors 153
- forward 254
- front_inserter 106, 272
- front inserters 274
- general inserters 275
- increment 83
- input 252
- inserter 106, 272
- inserters 104, 275
- iterator tags 283
- iterator traits 283
- iter_swap() 263
- output 253
- past-the-end 83
- random access 93, 255
- ranges 84
- raw_storage_iterator 731
- reverse 109, 264
- step forward 259
- swapping values 263
- user-defined 288
- iterator 288
 - for containers 230
 - for strings 508
- <iterator> 251, 665
- iterator adapter 104, 264
 - for streams 107, 277
 - inserter 271
 - inserters 104
 - reverse 109
 - user-defined 288
- <iterator.h> 251

- iterator tag 283
- iterator traits 283
 - for pointers 285
- iter_swap() 263
- iword()
 - for streams 659

K

- key_comp()
 - for containers 236
- key_compare
 - for containers 231
- key_type
 - for containers 231
- Koenig lookup 17

L

- labs()
 - global function 582
- L_{ANG} environment variable 692
- language features 9
- lazy evaluation with valarrays 554
- ldexp()
 - global function 581
- ldiv()
 - global function 582
- left flag 618
- left manipulator 619
- length()
 - for char_traits 689
 - for codecvt facet 721
 - for strings 485, 510
 - for valarrays 571
- length_error 25, 27
- less 132, 305, 375, 379
- less_equal 305
- lexicographical_compare() 360
- lexicographical comparison 360

- `<limits>` 60
- `<limits.h>` 59, 60
- limits of types 59
- linear complexity 21
- line-by-line input 493
- list 166, 172
 - see container
 - `!=` 168
 - `<` 168
 - `<=` 168
 - `=` 168
 - `==` 168
 - `>` 168
 - `>=` 168
 - `assign()` 168
 - assignment 168
 - `back()` 168
 - `begin()` 169
 - `clear()` 170
 - constructor 167
 - destructor 167
 - element access 168
 - `empty()` 168
 - `end()` 169
 - `erase()` 170
 - exception handling 172
 - `front()` 168
 - header file 166
 - `insert()` 170
 - iterators 169
 - `max_size()` 168
 - member functions 167
 - `merge()` 172, 246
 - operations 167
 - `pop_back()` 170
 - `pop_front()` 170
 - `push_back()` 170, 172
 - `push_front()` 170, 172
 - `rbegin()` 169
 - `remove()` 169, 170, 242
 - `remove_if()` 169, 242
 - removing elements 169
 - `rend()` 169
 - `resize()` 170
 - `reverse()` 246
 - `size()` 168
 - `sort()` 172, 245
 - special member functions 244
 - `splice()` 172, 245
 - splice functions 171
 - `swap()` 168
 - `unique()` 172, 244
- `<list>` 166
- `<list.h>` 166
- literal of type string 471
- literature 745
- locale 692
 - `!=` 700, 703
 - `()` 703
 - `==` 700, 703
 - as sorting criterion 703
 - "C" 694
 - category 700
 - class 694
 - `classic()` 694, 700, 703
 - collate category 724
 - `combine()` 700, 702
 - constructor 700, 702
 - cctype category 715
 - default constructor 698
 - destructor 700
 - facet 700
 - facets 698, 704
 - `global()` 697, 700, 703
 - `has_facet()` 700, 702
 - `id` 700, 704

- `imbue()` a stream 694
- messages category 725
- monetary category 711
- `name()` 697, 700, 703
- numeric category 705
- string collation 724
- string comparisons 703
- time category 708
- type 694
- `use_facet()` 700, 702
- `<locale>` 700
- `log()`
 - for complex 540, 545
 - for `valarrays` 575
 - global function 581
- `log10()`
 - for complex 540, 545
 - for `valarrays` 575
 - global function 581
- logarithmic complexity 21
- `logical_and` 305, 317
- `logical_not` 305
- `logical_or` 305
- `logic_error` 25, 27
- `long`
 - numeric limits 60
- `loop`
 - condition 600
 - for reading objects 600
- `lower`
 - for `ctype_base` 717
- `lower_bound()` 413
 - for containers 235
 - for maps 198
 - for multimaps 198
 - for multisets 180
 - for sets 180
- lower string characters 497

- `lt()`
 - for `char_traits` 504, 689
- `lvalue` 55

M

- `main()` 21
- `make_heap()` 329, 406, 407, 456
- `make_pair()`
 - for pairs 34, 36, 203
- manipulator 586, 612
 - `boolalpha` 617
 - `dec` 622
 - `endl` 586, 587, 612, 683
 - `ends` 587, 612, 650
 - `fixed` 625
 - `flush` 587, 612, 683
 - `hex` 622
 - implementing 614
 - internal 619
 - left 619
 - mechanism* 612
 - `noboolalpha` 617
 - `noshowbase` 623
 - `noshowpoint` 625
 - `noshowpos` 621
 - `noskipws` 626, 684
 - `nounitbuf` 626
 - `nouppercase` 621
 - `oct` 622
 - `resetiosflags()` 616
 - right 619
 - scientific 625
 - `setfill()` 619
 - `setiosflags()` 616
 - `setprecision()` 625
 - `setw()` 596, 619, 629
 - `showbase` 623
 - `showpoint` 625

- showpos 621
- skipws 626
- unitbuf 626
- uppercase 621
- user-defined 614
- ws 587, 612
- map 91, 194, 207, 211, 213
 - see container
 - != 198
 - < 198
 - <= 198
 - == 198
 - > 198
 - >= 198
 - [] 92, 205
 - as associative array 92, 205
 - begin() 200
 - clear() 202
 - constructors 196
 - count() 198
 - destructor 196
 - empty() 198
 - end() 200
 - equal_range() 198
 - erase() 202
 - exception handling 207
 - find() 198
 - header file 194
 - index operator 92, 205
 - insert() 202, 203
 - iterators 200
 - lower_bound() 198
 - manipulating access 115
 - max_size() 198
 - member functions 196
 - operations 196
 - rbegin() 200
 - removing elements 204
 - rend() 200
 - replace key 201
 - size() 198
 - sorting criterion 195, 197, 213
 - subscript 92
 - subscript operator 205
 - swap() 199
 - upper_bound() 198
 - user-defined inserter 288
- <map> 194
- <map.h> 194
- mapped_type
 - for containers 231
- mask
 - for ctype_base 717
- mask_array 564, 578
- max() 66
 - for numeric limits 61, 614
 - for valarrays 571
- max_element() 340
- max_exponent
 - for numeric limits 61
- max_exponent10
 - for numeric limits 61
- maximum
 - of elements 339
 - of two values 66
 - of types 64
 - value of numeric types 59
- max_length()
 - for codecvt facet 721
- max_size()
 - for allocators 739
 - for containers 145, 146, 233
 - for deques 162
 - for lists 168
 - for maps 198
 - for multimaps 198

- for multisets 179
- for sets 179
- for strings 486, 510
- for vectors 151
- mdy_date_order 710
- member
 - as sorting criterion 123
- member function
 - as template 11
- member function adapter 307
 - mem_fun 308
 - mem_fun_ref 307
- member template 11
- memchr() 689
- memcmp() 689
- memcpy() 678, 689
- mem_fun 307, 308
- mem_fun1_ref_t 309
- mem_fun1_t 309
- mem_fun_ref 307
- mem_fun_ref_t 309
- mem_fun_t 309
- memmove() 689
- <memory> 51
- memory leak 38
- memset() 689
- merge() 416
 - for lists 172, 246
- message_base 725
 - catalog 725
- messages facet 725
 - close() 725
 - get() 725
 - open() 725
- messages locale category 725
- min() 66
 - for numeric limits 61
 - for valarrays 571
- min_element() 339
- min_exponent
 - for numeric limits 61
- min_exponent 10
 - for numeric limits 61
- minimum
 - of elements 339
 - of two values 66
 - of types 64
 - value of numeric types 59
- minus 305
- mirror elements 370
- mismatch() 358
- modf()
 - global function 581
- modifying algorithms 325, 363
- modifying elements 325, 363
- modulus 305, 306, 377
- monetary locale category 711
- money_base 712
 - none 712
 - part 712
 - pattern 712
 - sign 712
 - space 712
 - symbol 712
 - value 712
- money_get facet 715
 - get() 715
- money_punct facet 711
 - curr_symbol() 712
 - decimal_point() 712
 - frac_digits() 712
 - grouping() 712
 - negative_sign() 712
 - neg_format() 712
 - pos_format() 712

- positive_sign() 712
- thousands_sep() 712
- money_put facet 713
 - put() 714
- move()
 - for char_traits 689
- multibyte format 686
- multimap 194, 209
 - see container
 - != 198
 - < 198
 - <= 198
 - == 198
 - > 198
 - >= 198
 - begin() 200
 - clear() 202
 - constructors 196
 - count() 198
 - destructor 196
 - empty() 198
 - end() 200
 - equal_range() 198
 - erase() 202
 - exception handling 207
 - find() 198
 - header file 194
 - insert() 202, 203
 - iterators 200
 - lower_bound() 198
 - manipulating access 115
 - max_size() 198
 - member functions 196
 - operations 196
 - rbegin() 200
 - removing elements 204
 - rend() 200
 - replace key 201
 - size() 198
 - sorting criterion 195, 197, 213
 - swap() 199
 - upper_bound() 198
 - user-defined inserter 288
- <multimap.h> 194
- multiplies 132, 305, 367
- multiset 175, 189
 - see container
 - != 179
 - < 179
 - <= 179
 - == 179
 - > 179
 - >= 179
 - begin() 182
 - clear() 183
 - constructor 189
 - constructors 177
 - count() 180
 - destructor 177
 - empty() 179
 - end() 182
 - equal_range() 180
 - erase() 182, 189
 - exception handling 185
 - find() 180, 189
 - header file 175
 - insert() 182, 183, 184, 189
 - iterator 189
 - iterators 182
 - lower_bound() 180
 - manipulating access 115
 - max_size() 179
 - member functions 177
 - operations 177
 - rbegin() 182
 - rend() 182

size() 179
sorting criterion 176, 178, 191
swap() 182
upper_bound() 180
user-defined inserter 288
<multiset.h> 175
mutating algorithms 327, 386

N

name()
 for locales 697, 700, 703
namespace 16
 Koenig lookup 17
 rel_ops 69
 std 23
 using declaration 17, 23
 using directive 17, 24
narrow()
 for ctype facet 716
 for streams 626
narrow stream 585
negate 305
negative_sign()
 for moneypunct facet 712
neg_format()
 for moneypunct facet 712
nested class
 as template 14
new 26
 and auto_ptr 39
<new> 28
newline
 internationalized 691
newsgroups 743
next_permutation() 391
n-log-n complexity 21
noboolalpha manipulator 617

noconv
 for codecvt_base 722
nocreate flag 632
none()
 for bitsets 464
none monetary pattern 712
nonmodifying algorithms 323, 338
no-op 98
no_order date order 710
noreplace flag 632
norm()
 for complex 536, 543
noshowbase manipulator 623
noshowpoint manipulator 625
noshowpos manipulator 621
noskipws manipulator 626, 684
not1 306, 343
not2 306
not_eof()
 for char_traits 689
not_equal_to 305
nounitbuf manipulator 626
nouppercase manipulator 621
npos
 for strings 474, 495, 508
nth_element() 404
 versus partition() 330
NULL 71
 and strings 484
number of elements 338
numeric
 algorithms 331, 425
 base 621
 formatting 620, 705
 global functions 581
 libraries 529
 limits 59
<numeric> 321, 425

numeric locale category 705
 numeric_limits 59, 462, 614
 num_get facet 707
 get() 707
 num_punct facet 705
 decimal_point() 705
 falsename() 705
 grouping() 705
 thousands_sep() 705
 truename() 705
 num_put facet 706
 put() 706

O

oct flag 621
 oct manipulator 622
 octal representation 621
 odd element 306
 offsetof() 71
 off_type
 for char_traits 689
 for streams 635
 ofstream 627, 629
 ok
 for codecvt_base 722
 O(n) 21
 open()
 for messages facet 725
 for streams 633, 634
 Open Closed Principle 217
 openmode
 for streams 631
 operator
 -> for iterators 91
 const_cast 20
 dynamic_cast 19, 26
 for I/O 593
 for type conversion 19

reinterpret_cast 20
 static_cast 19
 typeid 26
 ! 601
 & 599
 << 593, 652
 >> 594, 652
 | 632
 ~ 599
 opfx() 658
 ordered collection 75
 order of elements change 386
 osfx() 658
 ostream
 see output stream and stream
 ostream 585, 591
 <ostream> 592
 ostreambuf_iterator 665, 666
 failed() 666
 ostream iterator 107, 278
 ostream_withassign 591
 ostringstream 645
 ostrstream 649
 out()
 for codecvt facet 721
 out flag 631
 out_of_range 25, 28
 output 583
 see output stream and stream
 adjustment 618
 defining floating-point notation 623
 field width 618
 fill character 618
 hexadecimal 621
 numeric bases 621
 octal 621
 of addresses 596
 of bitsets 468

- of bool 595, 617
- of complex 532, 539, 544
- of numeric values 620
- of strings 492, 524
- of void* 596
- operator << 593
- positive sign 620
- redirecting 641
- signs 620
- standard functions 610
- output buffer 668
 - epptr() 668
 - iterator 665
 - overflow() 669
 - pbase() 668
 - pbump() 675
 - pptr() 668
 - seekoff() 676
 - seekpos() 676
 - setp() 673
 - sputc() 663, 668
 - sputn() 663, 669
 - sync() 675
 - xspn() 669
- output iterator 253
 - versus forward iterator 254
- output_iterator 288
- output stream
 - buffer iterators 665
 - buffers 663
 - flush() 611
 - iterator 107, 278
 - member functions 610
 - put() 610
 - sentry 658
 - write() 610, 658
- overflow() 669
 - for output buffers 669

- overflow_error 25, 28
- overloading
 - of I/O operators 652
 - with functions as parameter 612

P

- pair 33, 184
 - != 34
 - < 34
 - <= 34
 - == 34
 - > 34
 - >= 34
 - constructor 34
 - first 34
 - first_type 34
 - make_pair() 34, 36, 203
 - second 34
 - second_type 34
- part
 - for money_base 712
- partial
 - for codecvt_base 722
- partial_sort() 329, 400
- partial_sort_copy() 402
- partial_sum() 429, 432
- partition() 395
 - versus nth_element() 330
- past-the-end iterator 83
- pattern
 - for money_base 712
- pbackfail()
 - for input buffers 677
- pbase()
 - for output buffers 668
- pbump()
 - for output buffers 675

- `peek()`
 - for input streams 609
- performance 21
 - of streams 681
- `perror()` 725
- plus 305, 336, 431
- POD 156
- pointer
 - `auto_ptr` 38
 - input 596
 - iterator traits 285
 - NULL 71
 - output 596
- pointer
 - for allocators 738
 - for strings 508
- `polar()`
 - for complex 533, 541
- `pop()`
 - for priority queues 459
 - for queues 449
 - for stacks 440
- `pop_back()`
 - for containers 243
 - for deques 163
 - for lists 170
 - for vectors 154
- `pop_front()`
 - for containers 243
 - for deques 163
 - for lists 170
- `pop_heap()` 407, 456
- `pos_format()`
 - for `moneypunct` facet 712
- positioning
 - in files 634
- `positive_sign()`
 - for `moneypunct` facet 712
- `pos_type`
 - for `char_traits` 689
 - for streams 634
- `pow()`
 - for complex 540, 545
 - for `valarrays` 574, 575
 - global function 311, 581
- `pptr()`
 - for output buffers 668
- `precision()` 539
 - for streams 623
- predicate 121, 302, 322
 - binary 123
 - unary 121
- `prev_permutation()` 391
- `print`
 - for `ctype_base` 717
- `PRINT_ELEMENTS()` 118, 332
- printing
 - see output
- priority queue 453
 - constructor 458, 459
 - `container_type` 458
 - `empty()` 459
 - header file 454
 - `pop()` 459
 - `push()` 459
 - `size()` 459
 - `size_type` 458
 - `top()` 459
 - `value_type` 457
- `priority_queue` 453
- proxy
 - for bitsets 466
 - for `vector<bool>` 158
- `ptrdiff_t` type 71
- `ptr_fun` 310, 319

`pubimbue()`
 for stream buffers 664
`pubseekoff()`
 for stream buffers 664
`pubseekpos()`
 for stream buffers 664
`pubsetbuf()`
 for stream buffers 664
`punct`
 for `ctype_base` 717
pure abstraction 94
`push()`
 for priority queues 459
 for queues 449
 for stacks 440
`push_back()`
 called by inserters 272
 for containers 241
 for deques 163
 for lists 170, 172
 for strings 490, 516
 for vectors 154
`push_front()`
 called by inserters 272
 for containers 241
 for deques 163
 for lists 170, 172
`push_heap()` 406, 407, 456
`put()`
 for `money_put` facet 714
 for `num_put` facet 706
 for output streams 610, 629
 for `time_put` facet 710
`putback()`
 for input streams 609
put buffer 668
 iterator 665
`putchar()` 669

`pwd()`
 for streams 659

Q

quadratic complexity 21

queue

 != 449
 < 449
 <= 449
 == 449
 > 449
 >= 449
 back() 449
 constructor 448
 container_type 448
 empty() 449
 front() 449
 header file 444
 pop() 449
 push() 449
 size() 448
 size_type 448
 user-defined version 450
 value_type 448
queue 444
<queue> 444, 454
<Queue.hpp> 450
quicksort 328
quiet_NaN()
 for numeric limits 61

R

radix
 for numeric limits 61
rand() 374
 global function 582
random access 76
 to files 634

- random access iterator 93, 255
 - distance 261
 - step backward 259
 - step forward 259
- random_access_iterator 288
- random_shuffle() 393
- range 97
 - change order of elements 386
 - comparing 356
 - copy 363
 - copy and modify elements 367
 - counting elements 338
 - empty 84
 - for iterators 84
 - half-open 97
 - in algorithms 97
 - maximum 339
 - minimum 339
 - modify elements 372
 - modifying 363
 - multiple 101
 - mutating 386
 - notation 97
 - numeric processing 425
 - of values 59
 - removing duplicates 381
 - removing elements 111, 378
 - replacing elements 375
 - searching elements 324, 341
 - sorting 397
 - swapping elements 370
 - transform elements 367
 - valid 97,99
- range_error 25, 28
- raw_storage_iterator 731
- rbegin() 109, 265, 269
 - for containers 145, 239
 - for deques 162
 - for lists 169
 - for maps 200
 - for multimaps 200
 - for multisets 182
 - for sets 182
 - for strings 526
 - for vectors 153
- rdbuf()
 - for streams 636, 639, 641, 683
- rdstate() 599
 - for streams 598
- reachable 97
- read()
 - for input streams 608
 - global function 678
- reading
 - see input
- readsome()
 - for input streams 609
- real()
 - for complex 536, 543
- reallocation
 - for strings 486
 - for vectors 149
- rebind
 - for allocators 734, 738
- red-black tree 176
- redirecting
 - streams 641
- reference
 - for allocators 738
 - for bitsets 467
 - for containers 230
 - for strings 507
 - for vector<bool> 159
- reference counting
 - for containers 222
 - for strings 506

- reference semantics
 - of container elements 135
- register_callback()
 - for streams 661
- reinterpret_cast 20
- relative to absolute values 331, 429
- release()
 - for auto_ptrs 54
- rel_ops 69
- remove() 378
 - for lists 169, 170, 242
- remove_copy() 380
- remove_copy_if() 380
- remove_if() 302, 378
 - for lists 169, 242
- removing algorithms 326, 378
- removing duplicates 381
- removing elements 111, 326
- rend() 109, 265, 269
 - for containers 145, 240
 - for deques 162
 - for lists 169
 - for maps 200
 - for multimaps 200
 - for multisets 182
 - for sets 182
 - for strings 526
 - for vectors 153
- replace() 375
 - for strings 491, 501, 519, 520
- replace and copy elements 376
- replace_copy() 376
- replace_copy_if() 376
- replace_if() 375
- representation
 - binary 462
 - decimal 621
 - hexadecimal 621
 - octal 621
- requirements
 - for container elements 134
 - for sorting criteria 176
 - for the compiler 9
- reserve()
 - for containers 233
 - for strings 486, 488, 510
 - for vectors 149
- reset()
 - for auto_ptrs 54
 - for bitsets 465
- resetiosflags() manipulator 616
- resize()
 - for containers 244, 248
 - for deques 163
 - for lists 170
 - for strings 491, 518
 - for valarrays 571
 - for vectors 154
- resource leak 38
- result
 - for codecvt_base 722
- result_type 310
- return_temporary_buffer() 731
- reverse() 386
 - for lists 246
 - for strings 501
- reverse_copy() 386
- reverse iterator 109, 264
 - base() 269
 - convert into iterator 269
- reverse_iterator
 - for containers 230
 - for strings 508
- rfind()
 - for strings 520, 521

right flag 618
right manipulator 619
rotate() 388
rotate_copy() 389
round_error()
 for numeric limits 61
round_indeterminate 64
round_style
 for numeric limits 61, 63
round_to_nearest 64
round_toward_infinity 64
round_toward_neg_infinity 64
round_toward_zero 64
runtime_error 25, 28
rvalue 55

S

safe STL 138
sbumpc()
 for input buffers 663, 676
scan_is()
 for ctype facet 716
scan_not()
 for ctype facet 716
scientific flag 623
scientific manipulator 625
search() 347, 499
searching algorithms 324, 341
search_n() 344
search_n_if() 346
second
 for pairs 34
second_argument_type 310
second_type
 for pairs 34
seekdir 635
seekg()
 for streams 634, 636, 644

seekoff()
 for output buffers 676
seekp() 651
 for streams 634, 644
seekpos()
 for output buffers 676
self-defined
 see user-defined
sentry 658
sequence
 see range
sequence container 75
set 87, 175, 186
 see container
 != 179
 < 179
 <= 179
 == 179
 > 179
 >= 179
begin() 182
clear() 183
constructors 177, 186
count() 180
destructor 177
empty() 179
end() 182
equal_range() 180
erase() 182, 186
exception handling 185
find() 180, 186
header file 175
insert() 182, 183, 184, 186
insert elements 183
iterators 182, 186
lower_bound() 180
manipulating access 115
max_size() 179

- member functions 177
- operations 177
- rbegin() 182
- rend() 182
- size() 179
- sorting criterion 176, 178, 191
- swap() 182
- upper_bound() 180
- user-defined inserter 288
- user-defined sorting criterion 294
- set()
 - for bitsets 465
- <set> 175
- set_difference() 420
- setf()
 - for streams 615
- setfill() manipulator 619
- setg() 678
 - for input buffers 677
- <set.h> 175
- set_intersection() 419
- setiosflags() manipulator 616
- setlocale() 693
- setp()
 - for output buffers 673
- setprecision() manipulator 625
- setstate()
 - for streams 598
- set_symmetric_difference() 420, 421
- set_union() 418
- setw() manipulator 596, 619, 629
- sgetc()
 - for input buffers 663, 676
- sgetn()
 - for input buffers 663, 677
- shift()
 - for valarrays 572
- short
 - numeric limits 60
- showbase flag 622
- showbase manipulator 623
- showpoint flag 624
- showpoint manipulator 625
- showpos flag 620
- showpos manipulator 621
- sign monetary pattern 712
- signaling_NaN()
 - for numeric limits 61
- sin()
 - for complex 540, 546
 - for valarrays 575
 - global function 581
- sinh()
 - for complex 540, 546
 - for valarrays 575
 - global function 581
- size
 - of containers 146
 - of strings 485
 - of vectors 149
- size()
 - for bitsets 464
 - for containers 145, 146, 233
 - for deques 162
 - for lists 168
 - for maps 198
 - for multimaps 198
 - for multisets 179
 - for priority queues 459
 - for queues 448
 - for sets 179
 - for stacks 440
 - for strings 485, 510
 - for valarray gslice 577
 - for valarrays 571

- for `valarray` slice 576
 - for vectors 151
- `size_t` type 71
- `size_type` 474
 - for allocators 738
 - for containers 231
 - for priority queues 458
 - for queues 448
 - for stacks 439
 - for strings 495, 507
- `skipws` flag 625
- `skipws` manipulator 626
- `slice` 555, 575
 - constructor 576
 - `size()` 576
 - `start()` 576
 - `stride()` 576
- `slice_array` 556, 575
- smart pointer
 - `auto_ptr` 38
 - for reference counting 222
- `snextc()`
 - for input buffers 663
- `sort`
 - elements 328
- `sort()` 123, 328, 397
 - for lists 172, 245
 - versus `stable_sort()` 398
- sorted collection 75
- sorted range 409
- `sort_heap()` 329, 407
- sorting algorithms 328, 397
- sorting criterion
 - as constructor parameter 178, 197
 - as template parameter 178, 197
 - at runtime 191, 213
 - for maps 195, 197, 213
 - for multimaps 195, 197, 213
 - for multisets 176, 178, 191
 - for sets 176, 178, 191
 - for strings 213, 499
 - function 123
 - function object 294
 - locale as 703
 - requirements 176
 - user-defined 123, 294
- sorting elements 397
- space
 - compressing 385
- space
 - for `ctype_base` 717
- space monetary pattern 712
- special characters
 - internationalized 691
- `splice()`
 - for lists 172, 245
- `sputback()`
 - for input buffers 663, 676
- `sputc()`
 - for output buffers 663, 668
- `sputn()`
 - for output buffers 663, 669
- `sqrt()`
 - for complex 540, 545
 - for `valarrays` 575
 - global function 581
- `srand()`
 - global function 582
- `<sstream>` 646
- `stable_partition()` 395
- `stable_sort()` 329, 397
 - versus `sort()` 398
- stack 435
 - `!=` 440
 - `<` 440
 - `<=` 440

- `==` 440
- `>` 440
- `>=` 440
- constructor** 440
- `container_type` 439
- `empty()` 440
- header file 435
- `pop()` 440
- `push()` 440
- `size()` 440
- `size_type` 439
- `top()` 440
- user-defined version 441
- `value_type` 439
- `<stack>` 435
- `<stack.h>` 435, 444, 454
- `<Stack.hpp>` 441
- stack unwinding 15
- standard error channel 585
 - redirecting 641
- standard input channel 585
 - redirecting 641
- standard operators
 - for I/O 593
- standard output channel 585
 - redirecting 641
- standard template library 73
 - see STL
- `start()`
 - for `valarray::gslice` 577
 - for `valarray::slice` 576
- state**
 - of function objects 296
 - of streams 597
- `state_type`
 - for `char_traits` 689
- static array container 155
- `static_cast` 19
- `std namespace` 23
- `<stddef.h>` 71
- `stderr` 585
- `<stdexcept>` 28
- `stdin` 585
- `<stdlib.h>` 71
- `stdout` 585
- STL** 73
 - algorithms 74, 94, 111, 321
 - commit-or-rollback 139
 - container adapters 435
 - containers 73, 75, 143
 - element requirements 134
 - error handling 137
 - exceptions handling 139, 248
 - extending 141
 - function objects 124, 293
 - functor 293
 - functors 124
 - introduction 73
 - iterator adapters 104, 264
 - iterators 74, 83, 251
 - manipulating algorithms 111
 - predicates 121, 322
 - priority queues 453
 - problems 136
 - queues 444
 - ranges 97
 - safe STL 138
 - stacks 435
 - transaction safe 139
- `str()` 539, 650
 - for string streams 647
- stream 583, 587
 - `<<` conventions 662
 - `>>` conventions 662
 - `adjustfield` 618
 - adjustment 618

- app flag 631
- ate flag 631
- bad() 598
- badbit 597
- basefield 621
- basic_filebuf 643
- beg 635
- binary flag 631
- boolalpha flag 617
- boolalpha manipulator 617
- buffer iterators 665
- buffers 663, 668
- callback 661
- character traits 687
- classes 588
- clear() 598, 633, 636
- close() 633
- connecting 637
- copyfmt() 615, 616, 641, 653, 661
- copyfmt_event 661
- cur 635
- dec flag 621
- dec manipulator 622
- defining floating-point notation 623
- end 635
- manipulator 586, 587, 612, 683
- end-of-file 590
- ends 650
- ends manipulator 587, 612
- EOF 590
- eof() 598
- eofbit 597
- erase_event 661
- event 661
- event_callback 661
- examples 611
- exceptions() 602
- fail() 598
- failbit 597
- failure 602
- field width 618
- file access 627
- filebuf 643
- fill() 618
- fill character 618
- fixed flag 623
- fixed manipulator 625
- flags() 615, 616
- floatfield 623
- flush manipulator 587, 612, 683
- for char* 649
- for file descriptors 637, 672
- format flags 615
- formatting 615
- formatting of bool 595, 617
- fpos 634
- freeze() 650
- getloc() 626
- good() 598
- goodbit 597
- hardfail 598
- header files 592
- hex flag 621
- hex manipulator 622
- hexadecimal 621
- ignore() 614
- imbue() 626, 694
- imbue_event 661
- in flag 631
- input buffers 676
- input functions 607
- internal flag 618
- internal manipulator 619
- internationalization 625
- is_open() 633
- iterators 277

- isword() 659
- left flag 618
- left manipulator 619
- manipulators 586, 612
- member functions 607
- narrow() 626
- noboolalpha manipulator 617
- nocreate flag 632
- noreplace flag 632
- noshowbase manipulator 623
- noshowpoint manipulator 625
- noshowpos manipulator 621
- noskipws manipulator 626, 684
- nounitbuf manipulator 626
- nouppercase manipulator 621
- numeric bases 621
- oct flag 621
- oct manipulator 622
- octal 621
- off_type 635
- open() 633, 634
- openmode 631
- operator ! 600
- operator .600
- operator .600
- out flag 631
- output buffers 668
- output functions 610
- performance 681
- positioning 634
- pos_type 634
- precision() 623
- pword() 659
- rdbuf() 636, 639, 683
- rdstate() 598
- read and write 643
- read and write position 644
- redirecting standard streams 641
- register_callback() 661
- resetiosflags() manipulator 616
- right flag 618
- right manipulator 619
- scientific flag 623
- scientific manipulator 625
- seekg() 634, 636, 644
- seekp() 634, 644
- sentry 658
- setf() 615
- setfill() manipulator 619
- setiosflags() manipulator 616
- setprecision() manipulator 625
- setstate() 598
- setw() 629
- setw() manipulator 596, 619
- showbase flag 622
- showbase manipulator 623
- showpoint flag 624
- showpoint manipulator 625
- showpos flag 620
- showpos manipulator 621
- skipws flag 625
- skipws manipulator 626
- state 597
- state and open() 634
- str() 650
- strings 645
- synchronize streams 637
- synchronize with C 682
- sync_with_stdio() 682
- tellg() 634
- tellp() 634
- testing the state 600
- tie() 637, 638
- trunc flag 631
- unitbuf 683
- unitbuf flag 625

- unitbuf manipulator 626
- unsetf() 615
- uppercase flag 620
- uppercase manipulator 621
- user-defined buffers 668
- widen() 626
- width() 618, 653
- ws manipulator 587, 612
- xalloc() 659
- streambuf 590, 668
 - see stream buffer
- <streambuf> 592
- stream buffer 663
 - see output buffer and input buffer
- << 597, 683
- >> 683
- for file descriptors 637, 672
- getloc() 664
- pubimbue() 664
- pubseekoff() 664
- pubseekpos() 664
- pubsetbuf() 664
- user-defined 668
- stream iterator 107, 277, 282
 - end-of-stream 108, 280
 - skip input 282
- streamoff 635
- streampos 634
- streamsize 607
- strftime() 708
- strict weak ordering 176
- stride()
 - for valarray::slice 577
 - for valarray::slice 576
- string 471
 - != 511
 - + 492, 524
 - += 490, 515
 - < 511
 - << 492, 524
 - <= 511
 - = 489
 - == 511
 - > 511
 - >= 511
 - >> 492, 524
 - allocator_type 526
 - and NULL 484
 - and vectors 155
 - append() 490, 515, 516
 - assign() 489, 514
 - at() 513
 - automatic type conversions 484
 - begin() 497, 525
 - capacity 485
 - capacity() 486, 510
 - char* stream 649
 - character traits 503, 687
 - clear() 490, 518
 - compare() 511, 512
 - compare case-insensitive 213
 - comparisons 488
 - concatenation 492
 - const_iterator 508
 - const_pointer 508
 - const_reference 507
 - const_reverse_iterator 508
 - constructor 501, 508, 509
 - converting index into iterator 500
 - converting into char* 484, 513, 629
 - converting iterator into index 500
 - copy() 484, 513
 - c_str() 484, 513, 629
 - data() 484, 513
 - destructor 509
 - difference_type 507

- `empty()` 485, 510
- `end()` 497, 525
- `erase()` 491, 501, 518
- `find()` 520, 521
- `find_first_not_of()` 522
- `find_first_of()` 522
- `find_last_not_of()` 523
- `find_last_of()` 523
- `get_allocator()` 526
- `getline()` 493, 525
- header file 479
- index to iterator conversion 500
- input 492, 524, 596, 620
- `insert()` 491, 516, 517
- internationalization 503
- iterator 508
- iterator operator ++ 258
- iterator operator -- 258
- iterators 497
- iterator to index conversion 500
- `length()` 485, 510
- literal 471
- locale dependent collations 724
- locale dependent comparisons 489
- lower characters 497
- `max_size()` 486, 510
- not case-sensitive 499
- `npos` 474, 495, 508
- output 492, 524
- pointer 508
- `push_back()` 490, 516
- `rbegin()` 526
- reallocation 486
- reference 507
- `rend()` 526
- `replace()` 491, 501, 519, 520
- `reserve()` 486, 488, 510
- `resize()` 491, 518
- `reverse()` 501
- `reverse_iterator` 508
- `rfind()` 520, 521
- search functions 493
- size 485
- `size()` 485, 510
- `size_type` 495, 507
- sorting criterion 213, 499
- `str()` 653
- stream functions 645
- `substr()` 492, 524
- substrings 492
- `swap()` 490, 515
- `traits_type` 507
- upper characters 497
- `value_type` 507
- string 471
- `<string>` 479, 688
- `stringbuf` 645
- `<string.h>` 689
- `string::npos` 474, 495
- `stringstream` 645
- string streams 645
 - `str()` 647
- string termination character
 - internationalized 691
- `strlen()` 689
- `strstream` 649
- `<strstream>` 649
- `strstreambuf` 649
- subscript operator
 - for deques 162
 - for maps 92, 205
 - for vectors 152
- `substr()`
 - for strings 492, 524

suffix
 _copy 323
 _if 323
 sum()
 for valarrays 572
 sungetc()
 for input buffers 663, 676
 swap() 67
 for containers 140, 145, 147, 237
 for deque 163
 for lists 168
 for maps 199
 for multimaps 199
 for multisets 182
 for sets 182
 for strings 490, 515
 for vectors 149, 151
 swapping
 for containers 147
 iterator values 263
 two values 67
 swapping elements 370
 swap_ranges() 370
 symbol monetary pattern 712
 sync() 658
 for output buffers 675
 sync_with_stdio()
 for streams 682

T

table()
 for ctype facet 718
 table_size
 for ctype facet 718
 tags
 for iterators 283
 tan()
 for complex 540, 546
 for valarrays 575
 global function 581
 tanh()
 for complex 540, 546
 for valarrays 575
 global function 581
 tellg()
 for streams 634
 tellp()
 for streams 634
 template 9
 constructor 13
 copy constructor 35
 default parameter 10
 member templates 11
 nested class 14
 nontype parameters 10
 typename 11
 test()
 for bitsets 465
 thousands_sep()
 for moneypunct facet 712
 for numpunct facet 705
 throw 16, 441, 450
 throw specification 16
 tie()
 for streams 637, 638
 time locale category 708
 time_base 709
 dateorder 709
 dmy 710
 mdy 710
 no_order 710
 ydm 710
 ymd 710
 time_get facet 708
 date_order() 708
 get_date() 708

- get_monthname() 708
- get_time() 708
- get_weekday() 708
- get_year() 708
- time_put facet 710
 - put() 710
- times 305
- tinyness_before
 - for numeric limits 62
- to_char_type()
 - for char_traits 689
- to_int_type()
 - for char_traits 689
- tolower() 497, 719
 - for ctype facet 716
- top()
 - for priority queues 459
 - for stacks 440
- to_string()
 - for bitsets 468
- to_ulong() 462
 - for bitsets 468
- toupper() 497, 669, 719
 - for ctype facet 716
- traits
 - for characters 504, 687
 - for iterators 283
- traits_type
 - for strings 507
- transaction safe 139
- transform() 120, 131, 367, 368, 497
 - for collate facet 724
 - versus for_each() 325
- transform elements 367
- transitive 176
- traps
 - for numeric limits 62
- true 18

- truename()
 - for numpunct facet 705
- trunc flag 631
- type
 - bool 18
 - ptrdiff_t 71
 - size_t 71
 - wchar_t 687, 692
- type conversion 19
- typeid 26
- <typeinfo> 28
- typename 11

U

- uflow()
 - for input buffers 676
- unary_function 310
- unary predicate 121
- underflow()
 - for input buffers 676
- underflow_error 25, 28
- unexpected() 26
- unget()
 - for input streams 609
- uninitialized_copy() 730, 742
- uninitialized_fill() 730, 740
- uninitialized_fill_n() 730, 741
- union set 418
- unique() 381
 - for lists 172, 244
- unique_copy() 384
- unitbuf 683
- unitbuf flag 625
- unitbuf manipulator 626
- unsetf()
 - for streams 615
- unshift()
 - for codecvt facet 721

- upper
 - for ctype_base 717
- upper_bound() 413
 - for containers 235
 - for maps 198
 - for multimaps 198
 - for multisets 180
 - for sets 180
- uppercase flag 620
- uppercase manipulator 621
- upper string characters 497
- URLs 743
- use_facet() 698
 - for locales 700, 702
- user-defined
 - algorithm 285
 - allocator 735
 - container 217
 - exception 441, 450
 - function object 310
 - inserter 288
 - iterator 288
 - iterator adapter 288
 - manipulators 614
 - sorting criterion 123, 294
 - stream buffers 668
- using declaration 17, 23
- using directive 17, 24
- utilities 33
- <utility> 33, 34, 69
- V**
- valarray 547
 - ~ 573
 - ! 573
 - != 573
 - % 573
 - %= 574
 - & 573
 - && 573
 - &= 574
 - * 573
 - *= 574
 - + 573
 - += 574
 - 573
 - = 574
 - / 573
 - /= 574
 - < 573
 - << 573
 - <<= 574
 - <= 573
 - == 573
 - > 573
 - >= 573
 - >> 573
 - >>= 574
 - ^ 573
 - ^= 574
 - | 573
 - |= 574
 - || 573
- abs() 574
- acos() 575
- apply() 572
- asin() 575
- atan() 575
- atan2() 575
- constructor 548, 570
- cos() 575
- cosh() 575
- cshift() 572
- destructor 570
- exp() 575
- fill() 571

- gslice 560
- gslice_array 561
- header file 547
- indirect array 567
- indirect_array 567
- lazy evaluation 554
- length() 571
- log() 575
- log10() 575
- mask_array 564
- max() 571
- min() 571
- pow() 574, 575
- resize() 571
- shift() 572
- sin() 575
- sinh() 575
- size() 571
- slice 555
- slice_array 556
- sqrt() 575
- sum() 572
- tan() 575
- tanh() 575
- type conversions 554
- valarray 547
- <valarray> 547
- valid range 97,99
- value monetary pattern 712
- value_comp()
 - for containers 236
- value_compare
 - for containers 231
- value pair 33
- value semantics
 - of container elements 135
- value_type
 - for allocators 737
 - for complex 541
 - for containers 230
 - for insert() 203
 - for priority queues 457
 - for queues 448
 - for stacks 439
 - for strings 507
- vector 148, 156
 - see container
 - != 151
 - < 151
 - <= 151
 - = 151
 - == 151
 - > 151
 - >= 151
 - [] 152
 - and strings 155
 - as dynamic array 155
 - assign() 151
 - assignment 151
 - at() 152
 - back() 152
 - begin() 153
 - capacity 149
 - capacity() 149
 - clear() 154
 - constructor 150
 - constructor 729
 - contiguity of elements 155
 - destructor 150
 - element access 152
 - empty() 151
 - end() 153
 - erase() 154
 - exception handling 155
 - for bool 158
 - front() 152

- header file 148
- insert() 154
- iterator operator ++ 258
- iterator operator -- 258
- iterators 153
- max_size() 151
- member functions 150
- operations 150
- pop_back() 154
- push_back() 154
- rbegin() 153
- reallocation 149
- removing elements 153
- rend() 153
- reserve() 149, 730
- resize() 154
- shrink capacity 149
- size 149
- size() 151
- subscript operator 152
- swap() 149, 151
- <vector> 148
- vector<bool> 158
 - flip() 158, 159
 - reference 159
- <vector.h> 148
- void*
 - input 596
 - output 596

W

- wcerr 592
- wchar_t
 - input 595
 - numeric limits 60
- wchar_t type 687, 692
- wcin 592
- wclog 592
- wcout 592
- Web site 5
- wfilebuf 627
- wfstream 627
- what() 28, 29
- whitespace
 - compressing 385
- wide-character format 686
- widen()
 - for ctype facet 716
 - for streams 626
- width()
 - for streams 618, 653
- wfstream 627
- wios 590
- wiostream 591
- wistream 591
- wistringstream 645
- wofstream 627
- wostream 591
- wostreamstream 645
- wrapper
 - for arrays 219
- write()
 - for output streams 610, 658
 - global function 672, 673
- writing
 - see output
- ws manipulator 587, 612
- wstreambuf 590, 668
 - see output buffer and input buffer
- wstreampos 634
- wstring 471
 - see string
- wstringbuf 645
- wstringstream 645

X

`xalloc()`

 for streams 659

`xdigit`

 for `ctype_base` 717

`xgetn()`

 for input buffers 677

`xspn()`

 for output buffers 669

Y

`ydm date order` 710

`ynd date order` 710
