



深入 C++ 系列

C++ STL 中文版

P. J. PLAUGER

ALEXANDER A. STEPANOV

著

MENG LEE

DAVID R. MUSSER

王昕 译



PH
PTR



中国电力出版社

www.infopower.com.cn



深入 C++ 系列

TP312

860

C++ STL 中文版

P.J. PLAUGER

ALEXANDER A. STEPANOV

MENG LEE

DAVID R. MUSSER

著

王昕 译



北方工业大学图书馆



00516481

中国电力出版社

内 容 提 要

本书对 C++ STL 进行了全面而深入的阐述。STL（标准模板库）是在惠普实验室中开发的，已纳入 ANSI/ISO C++ 标准。其中的代码采用模板类及模板函数的方式，可以极大地提高编程效率。本书由 P.J. Plauger 等四位对 C++ STL 的实现有着卓越贡献的大师撰写，详细讨论了 C++ STL 的各个部分。全书分为 16 章，其中的 13 章通过背景知识、功能描述、头文件代码、测试程序和习题，分别讲述了 C++ STL 中的 13 个头文件，其他章节介绍了 STL 中广泛涉及的三个主题——迭代器、算法和容器。本书附录列出了接口和术语表，最后列出了参考文献。

本书适合对 C++ 有一定了解的程序员及高等院校师生阅读。

图书在版编目 (CIP) 数据

C++ STL 中文版 / (美) 普劳格 (P.J. Plauger) 等著；王昕译。
-北京：中国电力出版社，2002.5
ISBN 7-5083-1058-6

I.C... II.①普...②王... III.C 语言-程序设计 IV.TP312

中国版本图书馆 CIP 数据核字 (2002) 第 028461 号

著作权合同登记号 图字：01-2002-0709 号

本书英文版原名：The C++ Standard Template Library

Published by arrangement with Prentice-Hall, Inc.

All rights reserved.

本书由美国培生集团授权出版

中国电力出版社出版、发行

(北京三里河路 6 号 100044 <http://www.infopower.com.cn>)

北京地矿印刷厂印刷

各地新华书店经售

*

2002 年 5 月第一版 2002 年 5 月北京第二次印刷

787 毫米×1092 毫米 16 开本 34.5 印张 784 千字

定价 69.00 元

版 权 所 有 翻 印 必 究

(本书如有印装质量问题，我社发行部负责退换)

译者序

众所周知，C++是一门功能强大的编程语言，支持多种编程典范（paradigm），其间包括 PB（Procedure-Based）、OB（Object-Based）、OO（Object-Oriented）以及新近出现的 GP（Generic Programming）。

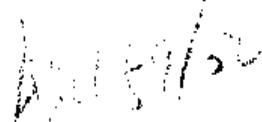
作为 GP 的第一个广为流传的实现，STL（Standard Template Library）自被 C++ 标准化委员会接纳以来，已经对整个 C++ 社区产生了极为深远的影响。程序员可以放心地使用 STL 所提供的常用数据结构和算法，避免“每次都重新发明一个轮子”的情况出现，有效地提高编程效率；再者，为了充分利用 STL 已有的资源，C++ 中 SL（Standard Library）的实现方式也有了很大的变化，一些常用的组件（如：iostream、string 等）都提供了和 STL 交互的接口。

随着 STL 在 C++ 社区的逐渐普及，对于 STL 的教学也开始变得红火起来。在国外，早在 1995 年，就有 Alexander A. Stepanov（STL 之父）和 Meng Lee（STL 的第一个实现者之一）的文章面世，1996 年更有 David R. Musser（GP 理论的又一元老）所著的《STL Tutorial and Reference Guide》出版，之后更是出现了《The C++ Standard Library》（Josuttis, 1999）、《Generic Programming and the STL》（Austern, 1999，本书中文版将由中国电力出版社引进出版，详情请访问：<http://www.infopower.com.cn>）、《Effective STL》（Meyers, 2001）、《The C++ Standard Template Library》（Plauger 等, 2001）等大师级的作品。

而反观国内，在 2000 年前，市面上讲述标准 C++ 的书籍寥寥无几，而讲述 STL 的书籍更是有如凤毛麟角。这使得绝大多数 C++ 程序员对最新的 ISO/ANSI C++ 为开发提供的强大支持缺乏必要的了解，导致了诸如 list、binary_search 等常见数据结构和算法的重复实现，造成了人力物力的浪费。这不可不说是国内 C++ 教学的失败。

但是，从 2001 年开始情况有所好转。国内一些 ANSI C++ 学习的先行者开始在网络上向大家介绍 ISO/ANSI C++ 的一些新特性，以使国内的 C++ 学习能够跟得上 C++ 社区的最新发展，这其中自然少不了对 STL 的介绍。在这段时期内，出现了一些以 ANSI C++ 为主题的网站，如：STL 小组 (http://www.smiling.com.cn/group/homepage.ecgi?group_id=11916)、dimension5 网站 (<http://www.dimension5.org>)，还有 csdn (<http://www.csdn.net>) 讨论区的 C/C++ 板块等。另外，国内的各家出版社也开始有目的地从国外引入一些大师的著作（如我上面提及的四本大师级著作都将在近期出版中文译本），以更快地让国内的读者接触到来自国外主流的声音。

机缘巧合之下，我有幸从中国电力出版社得到了翻译由 P.J. Plauger、Alexander A.



Stepanov、Meng Lee 和 David R. Musser 四位大师所著的《The C++ Standard Template Library》一书的机会，从而得以更早地接触到大师们的言论。作为本书的译者，我有必要介绍一下四位作者的身份，以加重本书的砝码（古人有云：不看僧面看佛面，虽然我人轻言微，但这四人在国外可都是响当当的人物哦②）。作为本书第一作者的 P.J. Plauger，对于国外 C/C++ 社区来说可是一个如雷贯耳的名字，他所著的《The Standard C Library》等书都早已成为经典著作，并且，目前使用人数最多的 C++ 编译器（MS VC++）中自带的 STL 就是出自他手，其实力可见一斑。第二作者 Alex Stepanov，则更是不得了，听听别人都叫他什么：GP 之父！哇……，STL 就是出自他那天才的思想，并且他还身体力行，连同本书的第三作者——Meng Lee，实现了 STL 的第一个实现（HP STL）。而本书的第四作者 David Musser，作为 Stepanov 的最初且时间最长的合作者，对 GP 理论的形成也称得上是功不可没。

由于 P.J. Plauger 亲自实现了一个商业化的 STL 版本，那么作为讲述 STL 实现的本书来说，则更显得意义非凡了。虽说，只要轮子好用就可以不用去管轮子是怎么做出来的，但适当地了解一些轮子制作过程中的知识，肯定可以帮助我们更好地使用这个轮子。在本书中，Plauger 把 STL 分成不同的部分，一个一个头文件地给我们讲述每一部分是如何做出来的，我们该如何去使用它们，然后给出了测试这些组件的完整代码，最后为了让读者加深对 STL 的理解，更是给出了一些颇具挑战性的习题[由于这些习题具有一定的参考价值，所以我们可能会在 C++view (<http://www.c-view.org/>) 杂志上逐步对这些习题的解法给出一些讨论以及自己的方案，以方便大家对于 STL 的理解。]从某种意义上来说，本书甚至可以改名叫做《STL 从入门到精通》。不过，金无足赤，本书也存在着一些小小的问题，因为是为了向读者展示一个简单的 STL 实现，本书给出的代码在不同程度上存在着异常安全性的问题[异常安全性，始终是 C++ 中的一个难点所在，目前对于这方面的最好探讨可以参见 Herb Sutter 所著的《Exceptional C++》(Sutter, 1999, 本书中文版将在近期内由中国电力出版社引进出版) 或 GotW 网站 (<http://www.gotw.ca>)；而且在不少地方的做法与现有的 C++ 标准稍有出入（这也可能是由于 VC++ 对于 C++ 标准的支持不足的缘故吧）；另外可能也是最重要的一点是，本书对于读者的要求比较高，它假定读者对 C++ 中的一些比较高级的用法都有所了解，虽说我在翻译过程中尽量对这些地方给出一些注释，但这依然会使读者产生一些困惑（幸好据小道消息透露，最近国内应该会出版很多有关 C++ 的经典著作，因此这个问题在一段时期后也就自然会不成为问题了③）。

最后，对于本书中文版的出现，我想感谢以下几个人，首先当然是本书的作者 P.J. Plauger 等人，感谢他们给我们带来了如此好的一本书，在翻译过程中，我曾经就书中的一些问题和 Plauger 互通过几次 email，更正了原书中的一些错误；另外就是来自于 brainbench 的 Andrei Itchenko，对于本书代码的异常安全性问题，就是他在 comp.lang.c++.moderated 上提出来的；然后就是中国电力出版社负责该书编辑的刘江先

生、关敏女士、宋宏女士以及参与此书中文版出版的全体员工，感谢他们给予我机会翻译此书并最终出版；再接下来就是 jzhou，没有他的帮助，我也不会那么早就开始学习 STL；最后我要感谢的就是经常在网上和我讨论 ISO/ANSI C++ 以及 STL 的朋友们，包括 babysloth、bugn、myan 等人，和他们的讨论也使得我对 STL 有了更深层次的理解。另外，如果对本书有任何疑问的话，可以与我联系，我的 email 是：cber@email.com.cn。

王昕

2002 年 5 月

序 言

标准模板库（Standard Template Library，简称 STL）是 ANSI/ISO C++ 语言的库的一个主要组成部分。它最初是惠普实验室（Hewlett-Packard Labs）的产物，开发者为 Alexander Stepanov 及 Meng Lee（参见 S&L95），主要基于早期的、由 Stepanov 和 David R Musser 两人完成的工作。（参见 M&S87、M&S89 及 M&S94。这里所有参考文献都列于本书后的“参考文献”中。）你将会发现在这个库中包含着对于 C++ 中的模板（template）的各种使用，其中大部分的使用都值得炫耀，并且具有良好的连贯性。确实，STL 已经开始有意义地改变着许多程序员编写 C++ 代码的方式。

本书展示了怎样使用 STL 中提供的模板类（template class）及模板函数（template function），一如 C++ 标准中所要求的一样（参见条款 20 及 23~26）。我们在此将 STL 视为标准 C++ 中所定义的庞大的库中一个相对独立的子集。C++ 中库的设计与 ANSI/ISO 标准 C（ANS89 与 ISO90）中的库设计一脉相承。因此，也可以将本书视为 P.J. Plauger 先前的两本书，《The Standard C Library》（Pla92）与《The Draft Standard C++ Library》（Pla95）的一个后续版本。几乎所有 C++ 程序员感兴趣的有关库的内容都可以在这三本书中找到。

C++ 标准

1998 年，C++ 标准正式通过，并且将在接下来的一段时间中保持一种稳定的状态^①。它既是 ANSI 标准，也是 ISO 标准，也就是说，它既是美国的国家标准，也是一种全球通用的标准。作为标准化进程的一部分，整个标准 C++ 语言及库第一次完整地描述在一起。一个相对较晚加入这个标准化进程的事件是在 C++ 标准的草案中接纳 STL 为 C++ 标准的一部分。现在，标准 C++ 的编译器及库的各种实现也不过刚刚出现。也就是说，即使是很经验的 C++ 程序员也将发现本书中有许多东西是全新的。

同样，在 1994 年 6 月被标准化委员会接纳为 C++ 标准草案的一部分之前，对 STL 的早期描述（至少早于 S&L95）仅限于一个相对狭窄的范围内。在被接纳的过程中，STL 本身又被重新组织并且在几个重要的环节做了修改。现在，在标准 C++ 中的 STL 已经不再是由惠普实验室开发出来的那套软件开发包，同时它与那些由不同的代理提供的强化版

^① 一般的标准化文档在正式通过后的 5 年内将保持一种稳定的状态，期间即使需要对它进行修改，也要等到 5 年后再提交给标准化委员会进行表决。——译者注

本也不同。这也就是说，即使是那些有着早期的 STL 使用经验的程序员也可以从本书中获得新的知识。你将会在本书中发现对 STL 的完整描述，一如在 C++标准中指定的那样。

本书中还将至少告诉你一种实现 STL 的方法。书中提供了大概 6000 行经过测试、证明可行的代码，同时这些代码也被证实可以移植到许多 C++编译器上。实际上，这些代码在本质上和 Microsoft Visual C++、IBM Visual Age 或其他厂商所提供的 C++编译器中的 STL 代码几乎是相同的。我们仅仅是在排版及符号表达方面做了小小的改动，以使得这些代码在书中更具有可读性，同时将它们用来作为教学示例也比较方便。

作为对 C++库的扩展，书中提供的这些代码可以工作于任何其他 C++库上面（参见附录 A）。然而，当与一个完全符合 C++标准的库一起工作时，它表现得尤为出色。我们尽可能地去除了那些不具有可移植性或不可能被广泛利用的代码。那些过分依赖于 C++语言中最近才加入的特性的代码，如模板的部分特化（template partial specialization）等，有可能在一些编译器上导致问题的产生。你也可以在那些商业化的 STL 版本中发现对于这些特殊情况的各种不同的折衷处理。

无论如何，你都可以通过使用本书中提供的代码，获得一些对于如何使用由模板构成的库的宝贵经验，而使用模板库也已经开始成为 C++世界中的一个重要标准。同样重要的是，我们相信看完一个实际的 STL 实现将有助于你更好地理解和应用它。

这就引入了本书的另一个目的。除了给出 STL 的标准，以及实现它的可行代码之外，本书还可以作为一本如何使用该库的教程。你将发现在本书中存在着一些有用的背景信息，告诉你 STL 是如何演变成现在的这个版本的，使用它意味着什么，以及如何使用它。为此并不需要阅读和理解我们所提供的所有代码。哪怕是对于本书的粗略学习，也能给你带来不少有益的收获。不需要是老手才能从本书中获益，那些偶尔才显得比较老练的程序员将会发现这里所提供的信息简直就是无价之宝。

本书的目的不是教你如何写 C++代码。我们假定你已经了解了足够多的 C++知识，已经可以看懂简单的 C++代码。虽然在本书中出现的代码并不是那么简单，但不要怕，我们会解释其间所使用的各种技巧。

扩展 STL

本书的最终目的就是告诉程序员如何设计及实现对 STL 的扩展。STL 集合了大量的算法、数据结构以及编程技巧。然而，它并没有提供程序员可能需要的全部特性。相比而言，它只是提供了那些核心的、被广为使用的特性，并且描述了这些核心代码的编写规则。

当你掌握了这些规则后，就可以往 STL 中添加自己的算法，并且让它与已有的数据结构一同正常地工作。你也可以添加自己的数据结构，使之与已有的算法协同工作。通过将本书中提供的 STL 代码作为例子来

使用，你将很快学会如何以最少的新增代码来处理新的问题，并且发现这些新增的代码可以在以后的项目中重用。这也就是设计库的最终原因。

本书的结构

本书的结构看起来更像是 STL 代码本身的结构。*C++*标准中列出了大量的头文件，但只有其中的 13 个头文件中定义了 STL 中所有的模板。这 13 个头文件的每一个在本书中都有对应的一章。其他的章节则全面地介绍了 STL，并讨论了 STL 中三个涉及较为广泛的主题——迭代器 (iterator)、算法 (algorithm) 和容器 (container)。这些头文件中的大部分都有着适量的相关内容，因此也就引起了一些相关的讨论。有关它们的相关章节必然会导致讨论范围的扩大。

每章都以相同的模式讲述了一个头文件。一开始是一个简短的背景介绍小节，接着就是有关这个头文件内容的功能性描述，然后是对于如何更好地使用该头文件中提供的特性所给出的一些建议。再接着给出了组成头文件的 C++ 代码，并伴随有代码是如何工作的有关注释。我们还为每一个头文件给出了一个小小的测试程序，给出至少一个粗略的例子演示每一个定义的模板是如何使用的。

在每一章的最后给出了一些习题。如果本书是作为某个大学课程的课本的话，这些习题可以当作家庭作业来完成。大部分习题都是简单的练习，例如使用库，或是代码重写等。它们能够使人理解实现中的某些重点，或是展示了实现中的一些合理变更。那些较难的习题都标注出来了，它们可作为一个更大范围的项目的基础。那些自学本书的人可以将这些习题作为一种锻炼，以激发更多的思考。

代码

本书中提供的代码以及对这些代码的描述都是基于惠普公司被广泛使用的 STL 版本。这个版本包括如下的声明：

Copyright © 1994 by Hewlett-Packard Company

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

本书中所提供的代码都经过了各种程度的测试，能够正常工作于如下公司提供的 C++ 编译器：Microsoft、IBM、Edison Design Group 以及 Project GNU。这些代码通过了 Dinkum C++ Proofer 和 Dinkum Abridged Proofer 所提供的 STL 测试，这两种工具是由 Dinkumware 公司开发的用于评估相应的库质量的工具。它还通过了由 Perennial Software 公司和 Plum Hall 股份有限公司所提供的各种版本的商业库验证套件的测试。我们尽力将出错情况最小化，但不能保证没有任何出错情况。

同时请大家注意，在本书中所出现的代码同样处于版权保护之下。它并没有放置在公共域中，也不具有共享性质。它不处于“copyleft”^② 协议的保护下，即与自由软件基金会（FSF）所提供的代码不一样。P.J. Plauger 拥有这些代码的所有版权，Dinkumware 公司视它们为商业化的基础代码而提供许可。

给定的 C++ 编译器中所带有的 STL 代码可能与我们在本书中提供的代码在很多方面存在着不同。这是可以理解的，即便是那些基于本书中所提供代码的 STL 代码也会如此。C++ 中的各种方言（dialect）依然存在，尤其是在那些复杂的领域中（如模板处理）。随着时间的推移，这些不同肯定会被消除，为此我们应该感谢 C++ 标准化进程的完成。但是在未来的某些时间，你应该将本书中提供的代码视为许多实现的典型。

致谢

惠普公司在引导 STL 发展到现阶段的形式的过程中起了极大的作用。我们极为感激他们的重要贡献。我们同样感谢惠普公司开发出最初的 STL 版本并将其作为免费库发放。他们的慷慨使得 STL 在 C++ 社区中得以迅速传播。

在标准化进程的后期，Bjarne Stroustrup 和 Andy Koenig 极力向 ANSI 委员会 X3J16（现在是 J16）和 ISO 委员会 WG21 建议将 STL 加入到 C++ 标准草案中。如果没有他们的热情支持，本书也不可能出现，更不会以现在这种形式出现。

Matt Austern 对本书的最终草案提出了很多有用的注解。我们也非常感谢他的建设性批评。

本书中的一些原始素材最初出现在月刊《The C/C++ Users Journal》

^② copyleft 也是由 Project GNU 提出来的一种版权方式，对应于传统的 copyright（版权）。它允许用户自由地分发软件及其源代码，不必受到传统的版权的限制，但又对用户修改软件源代码的行为做了一定的限制，在这一点上与在公共域中不一样。——译者注

的“Standard C/C++”专栏中，此专栏由 P. J. Plauger 负责。我们同样感谢出版社能够让我们重复利用这些文章。

Geoffrey Plauger 协助了本书排版工作的完成。

P.J. Plauger 于马萨诸塞州，康科德城

Alexander Stepanov 于加利福尼亚州，帕洛阿尔托

Meng Lee 于加利福尼亚州，库珀蒂诺

David R. Musser 于纽约，Loudon ville

目 录

译者序

序 言

第0章 简介	1
背景知识	1
功能描述	8
使用 STL	9
实现 STL	10
测试 STL	16
习题	17
第1章 迭代器	19
背景知识	19
功能描述	19
使用迭代器	26
习题	28
第2章 <utility>	30
背景知识	30
功能描述	32
使用<utility>	35
实现<utility>	36
测试<utility>	37
习题	39
第3章 <iterator>	40
背景知识	40
功能描述	52
使用<iterator>	72
实现<iterator>	78
测试<iterator>	102

习题	109
第4章 <memory>	110
背景知识	110
功能描述	120
使用<memory>	130
实现<memory>	132
测试<memory>	144
习题	147
第5章 算法	148
背景知识	148
功能描述	150
使用算法	152
习题	152
第6章 <algorithm>	154
背景知识	154
功能描述	154
使用<algorithm>	183
实现<algorithm>	187
测试<algorithm>	228
习题	235
第7章 <numeric>	236
背景知识	236
功能描述	236
使用<numeric>	238
实现<numeric>	239
测试<numeric>	241
习题	242
第8章 <functional>	243
背景知识	243
功能描述	245
使用<functional>	256
实现<functional>	259

测试<functional>.....	267
习题.....	271
第 9 章 容器.....	272
背景知识.....	272
功能描述.....	274
使用容器.....	281
习题.....	283
第 10 章 <vector>.....	284
背景知识.....	284
功能描述.....	285
使用<vector>.....	296
实现<vector>.....	299
测试<vector>.....	320
习题.....	323
第 11 章 <list>.....	325
背景知识.....	325
功能描述.....	326
使用<list>.....	337
实现<list>.....	339
测试<list>.....	357
习题.....	360
第 12 章 <deque>.....	361
背景知识.....	361
功能描述.....	362
使用<deque>.....	372
实现<deque>.....	374
测试<deque>.....	393
习题.....	394
第 13 章 <set>.....	395
背景知识.....	395
功能描述.....	401
使用<set>.....	418

实现<set>.....	420
测试<set>.....	445
习题.....	449
第 14 章 <map>.....	450
背景知识.....	450
功能描述.....	451
使用<map>	469
实现<map>	472
测试<map>	476
习题.....	481
第 15 章 <stack>.....	482
背景知识.....	482
功能描述.....	483
使用<stack>	486
实现<stack>	488
测试<stack>	489
习题.....	491
第 16 章 <queue>.....	492
背景知识.....	492
功能描述.....	492
使用<queue>.....	499
实现<queue>	502
测试<queue>	504
习题.....	507
附录 A 接口.....	508
附录 B 术语	512
参考文献.....	534

第0章 简介

背景知识

库 (library) 是一系列程序组件的集合，它们可以在不同的程序中重复使用。C++语言依照传统的习惯，提供了由各种各样的函数 (function) 所组成的库，用于完成诸如输入/输出、数学计算等功能。这其中的某些函数直接来源于标准C的库（如printf），其他的则专属于标准的C++库（如set_new_handler）。然而，不论其来源如何，库函数都遵照以下规则：接受一些符合预先指定类型的参数，返回一个特定类型的值或改变一些已有的值。设计一个能被广泛使用的C或C++库的一个重要组成部分就是，猜测出这些函数中最有可能出现的参数组合情况。

C++与C相比而言有着重大的改进。它将C中的数据结构的概念进行了扩展，使得它不但能包含成员对象，还能包含成员函数。C++中的类 (class) 可以更好地将一些相关的数据及其上进行的操作封装在一起。设计一个能被广泛使用的C++库的另一个重要组成部分就是猜测出这些类中最有可能出现的成员对象类型的组合情况。

在其最新的发展过程中，C++新增了模板 (template) 这个特性。通过使用模板，C++允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化 (specialize) 的时候。可以定义模板类及模板函数。在一定的限制下，模板类或模板函数可以允许程序员针对不同的类型提取出特定的、一般化的代码为这些类型服务。正如你所想到的，模板库相对于传统的、由函数及类组成的库来说，可以提供更好的代码重用机会。当然，标准化委员会也不会忽视这些机会，所以它也被标准化委员会添加在C++标准中。

ANSI X3J16
ISO WG21

ANSI (American National Standards Institute, 美国国家标准学会) 负责制定计算机编程语言的美国标准。X3J16是经ANSI组织授权制定C++语言标准的标准化委员会的名称，它于1989年正式启动，现在我们称它为J16。ISO (International Organisation for Standardization, 国际标准化组织) 是一个与ANSI相似的组织，不过它负责的是制定全球化的标准。ISO于1991年成立了技术委员会JTC1/SC22/WG21，它协同X3J16一起工作，目的是制定出一个既符合ANSI要求，又符合ISO要求的C++标准。在本书中，“标准化委员会”同时指代X3J16/J16与WG21。在过去

的8年间，标准化委员会每年都举行三次会议，一次持续一周，每次会议都会对C++标准的制定过程产生重大的影响。

我们现在所见的C++标准的最后一次技术上的修改完成于1997年11月，其官方名称为ISO 14882。这意味着从现在起，这份文档将进入一个相对稳定的阶段——一般来说每一份经由ISO制定出来的标准都将在以后的5年间（至少是5年）被冻结^①。然而，在这份草案被制定为标准前，标准化委员会还是对它做了一些较大的改动，扩充了其中有关库的部分条款。其中一个比较大的改动就是将模板引入标准库，使用模板类来代替传统的C++中定义的类。但是对标准库的最大的一次改动出现在1994年7月。在1994年7月，标准化委员会通过投票决定，将STL加入到C++的标准中。

STL

STL（Standard Template Library，标准模板库）是惠普实验室开发的一系列软件的统称。它是由Alexander Stepanov（现在AT&T实验室工作）、Meng Lee（仍在惠普实验室工作）和David R Musser（现在Rensselaer Polytechnic Institute工作）在惠普实验室工作时所开发出来的（S&L95和M&S96）。现在虽说它主要出现在C++中，但在被引入C++之前该技术就已经存在了很长的一段时间。实际上，Musser和Stepanov在十多年前就已经向世人展示了它的一个早期实现，那时他们用以完成STL的工具是Ada中的generics（这也是模板的一种形式，M&S87和M&S89）。这些工作也是基于他们两人在更早的时期所完成的一些工作。

如同我们在这里所展示的一样，惠普实验室最初的工作产生了超过6000行的、非常精致的代码。几乎所有的代码都采用了模板类及模板函数的方式。可以为这些模板指定各种类型来进行特化，这些类型可以是语言本身提供的内置（built-in）类型，也可以是自己所定义的类。通过指定类型对模板进行特化所产生的代码与自己所写的代码非常接近。而且对于STL中所提供的数据结构和算法，你也很难自己编写代码来正确地完成它们。因此我们可以尽情地享受STL带来的好处，使用它所提供的精巧的方案来解决常见问题，从而获得在代码空间及执行时间上的效率。

STL的代码从广义上大致分为三类：算法（algorithm）、容器（container）和迭代器（iterator）。在这些分类中，迭代器可能是其中最有趣、最有创造性的一类，不过也是最难解释清楚的一类。因此，我们在此先介绍算法和容器。

算法

严格地说，算法是用以决定结果的一个数学过程。它与数学中的函数在某些方面是有区别的。可能对于我们来说最重要的还是，算法

① 即不会被修改。——译者注

是由有限次数的操作所完成的。相比之下，函数在概念上来说可以执行无限次数的操作，然后得到一个定义良好的结果。

在实际的编程过程中，我们经常随意使用“算法”这个术语。在C/C++中，我们经常用它来表示那些能够得到有用结果的函数。好的算法在完成一件工作时能够有效地减少所需要的操作步骤和所需要使用的存储空间，并且还能够提供一个可以预期的时间复杂度。它应该谨慎地处理一些特殊的情况，如进行0次的重复计数等。它还应该避免出现在执行期间的溢出或其他不正常的计算情况。它应该是高内聚的，对外只提供一个合理的尽量小的接口。

对于程序员来说，写得好的算法是一笔很重要的财富。例如，我们一直都在大量地使用标准C库。所有数学函数、字符串操作符及I/O实现程序都使用那些被时间证明具有非常好的可重用性的算法。

正如我们在最开始所说的，函数库对数据类型的选择对其可重用性起着关键性的作用。一个用来求平方根的函数，在使用double作为其参数类型的情况下，其可重用性肯定比使用int作为它的参数类型要高。同样，由一系列char所组成的字符串或者文件，其可重用性也肯定比一系列float所组成的序列要高。现在，让我们来想像一下，当我们对这些函数所使用数据的类型有自己的看法时，它们的适用性又是如何呢？

这时我们就需要借助模板的强大功能了。仅仅是多了一点点限制，就可以使用一个模板函数或类来为不同的类型服务，至于填空的工作交给翻译器就可以了。使用模板时需要遵循的限制条件是由能够怎样操作参数类型所决定的。举例来说，如果模板需要对参数类型的对象“加一”的话，那么你所选择的类型最好对于对那种类型的对象“加一”意味着什么有着良好的定义。

STL的一个重大成就在于，它提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——你可以将所有的类型划分为少数的几类，然后就可以在模板的参数中使用一种类型轻易地替换掉同一种类中的其他类型。每一个种类都拼写得很清楚，编写的模板定义也十分健壮。也就是说，对于给定算法可能会使用到的类型通常都能够猜测到正确的答案，而且通常代码也能完成你想要它完成的功能。

STL提供了大约100个实现算法的模板函数。其中简单者如for_each，它将为指定序列中的每一个元素调用指定的函数；复杂者如stable_sort，它以你所指定的规则对序列进行排序，如果在序列中有两个元素在给定规则下相等，那么使用stable_sort排序不会导致它们在序列中的前后关系发生改变。

熟悉了STL后，你就会发现：以前所写的许多“有趣功能”的代码现在只需要用短短几行就能完全替换了。通过调用一两个算法模板，就

可以得到一些优雅且高效的代码，它同样也能够完成你所需要的功能，并且比你自己的代码所完成的还要好。

容器

在实际的开发过程中，组织数据和选择操作于这些数据上的算法几乎一样重要。组织一系列元素的方式将直接影响到诸如添加、删除、以多种方式存取以及重新组织它们等操作的复杂程度和时间消耗。事实上，当程序中存在着对时间要求很高的部分时，数据结构的选择就显得尤为重要，有时它甚至会直接导致程序的成败。

在物理学中也存在着这样类似的问题。在物理学中有一些计算非常复杂（简直就是声名狼藉），尤其是那些需要在边界匹配两种解决方法的计算更是如此。对于这种问题的一种解决方法是改变当前所使用的坐标系统，使得原来系统中复杂且难处理的边界值在新的系统中变得简单而且易于处理。当然，坐标系统之间的转换可能会比较复杂。这样做的优势在于，你要解决的问题可能在以前就有人解决过了，你所需要的只不过是将已有的解法拿来重新利用一番而已。

在计算过程中，我们不断地重复发现一些有用的方法用于组织一系列数据。也可以把它们想像成和上面讨论的坐标系统类似的系统。数组能够很方便地完成随机存取的操作，不过它不能简单地实现其本身的增长，因而往数组中插入元素简直就是一件麻烦事。链表对于添加和插入操作只需要做少量的动作，但存取任意元素的复杂度却成线性增长，而不像随机存取那样是一个常数，等等。

令人悲哀的是，在过去的时间中，我们不断地重复实现一些诸如向量（vector）、列表（list）等常见的数据结构。这些代码都十分相似，只是为了适应不同数据的变化而在一些细节方面有一些不同之处。那么我们自然可以这样想：如果有人能够实现一个链表解决上述问题的话，我们是不是可以重复利用这个已有的实现来实现我们自己的链表呢？如果能的话，那样做是不是很棒呢？

于是STL容器就闪亮登场了。STL容器是一些模板类，它们支持半打左右组织数据的最通常的方法。这些模板的参数允许我们指定容器中元素的数据类型。所有用于增长、收缩、插入以及存取容器中所有元素的代码一旦提供，就可以被所有容器重用。

再重复一次，慢慢地你将会发现，许多要为一个新程序所写的代码其实只是对于一些通用的数据组织方法的笔记工作，STL中提供的一两个容器就能够完全搞定这些单调乏味的工作。

迭代器

迭代器在STL中用来将算法和容器联系起来，起着“胶合剂”的作用。通过为C++中新近定义的类重载一些操作符，迭代器一般化了C中指针的概念。例如，考虑一下first和last这两个通常用于指定序列边界的指针：

```
for (sum = 0; first != last; ++first)
    sum += *first;
```

毫无疑问, first在此处看起来就是一个指针。不过在C++中, 它不需要一定为一个指针。实际上, 它可以是任意符合下列条件的类型:

- 比较两个值是否相等 (`first != last`)
- 间接取值 (`dereference`), 以存取某个值 (`*first`)
- (前) 递增 (`++first`)

当然, 你也可以在这段代码中使用传统的对象指针。

通过在上例中添加几行代码, 可以得到一个模板函数, 然后就可以利用这个模板函数来对任意类型T所形成的元素序列求和:

```
template<class T, class It> inline
    T sum_all( It first, It last)
    { T sum;
        for (sum = 0; first != last; ++first)
            sum += *first;
        return (sum); }
```

如果在上述代码中再添加一些需求, 我们就可以得到一个典型迭代器的规范。实际上, 上面给出的这个模板函数与第7章中描述的模板函数`accumulate`十分相似。

几乎STL提供的所有算法都是通过迭代器存取元素序列进行工作的。对于不同的迭代器, STL中提供了一些分类方法, 那些算法函数也都是围绕这些分类来定义的。正是由于这样的组织原则, 使得STL具有更强的功能以及更大的弹性。

STL中的每一个容器都定义了其本身所专有的迭代器, 用以存取容器中的元素。当然, 也可以为自己定义的数据结构提供自己的迭代器。值得注意的是, 普通的指针也可以像迭代器一样正常地工作。如果你没有兴趣, 也可以不用去定义一堆奇特的类。另外, 你也可以很容易地扩展已有的算法和容器, 并使之在STL框架中正常工作。

惠普实验室 的代码

惠普实验室已经将其STL实现免费向公众开放, 这是一件值得庆幸的好事情。从此你就可以自由地使用这些代码以及附带的文档, 甚至还可以将其作为商业化产品的一部分重新发行。为此不需要付任何版税。所有的要求就是: 在使用这些代码时必须同时附上他们提供的版权及责任说明, 这完全是无可厚非的(详细的说明参见“序言”中的“代码”一节)。

惠普实验室同样也保留了STL中的几个自适应算法的专利。他们也声明了如何使用这些专利——我们可以在没有预先获得许可及不支付版税的前提下使用它们。对于该公司在此领域所做出的巨大贡献, 我们不

得不给予高度评价。

然而,请注意:那些公众可获得的HP STL版本及其最新的经过重新组织的版本并没有被ISO14882(也就是C++标准)所采纳。如同被标准化委员会所采纳的其他建议一样,相对于其最初提交给委员会的版本来说,STL的规范几经修订,早已不复当时的面貌了。那些众所周知的大小小的变化对于STL代码的影响已经遍布了STL中几乎所有的角落。这就导致了这样的一个问题:使用现有的包编写的程序要想移植到兼容C++标准的实现中必须做一些改动。

同时,STL的组织方式也有了改变。举例来说,在惠普最初发行的版本中,STL组织为下列的48个头文件:

algo.h	algobase.h	bool.h	bvector.h
defalloc.h	deque.h	faralloc.h	fdeque.h
flist.h	fmap.h	fmultimap.h	fmultset.h
fset.h	function.h	hdeque.h	heap.h
hlist.h	hmap.h	hmultimap.h	hmultset.h
hset.h	hugalloc.h	hvector.h	iterator.h
lbvector.h	ldeque.h	list.h	llist.h
lmap.h	lmultimap.h	lmultset.h	lngalloc.h
lset.h	map.h	multimap.h	multiset.h
neralloc.h	nmap.h	nmultimap.h	nmultset.h
nset.h	pair.h	projectn.h	set.h
stack.h	tempbuf.h	tree.h	vector.h

而在C++标准中,它们被重新组织为下面的13个头文件:

algorithm	deque	functional	iterator	list
map	memory	numeric	queue	set
stack	utility	vector		

这样的改动所带来的影响非常大,但却很容易解决。如果你碰巧有一些代码使用了HP所提供的STL版本,那么将它们改成符合标准的形式并不太难。所需要做的只是将代码所包含的文件列表稍加修改,不断地向其中加入新的头文件直到编译器不再报错为止。(不过请注意:对于移植性产生的问题来说,这并不是一个具有可移植性的解决方案。不同的编译器可能会有不同的包含头文件的指令。但是这种方法却可以让你很快高兴起来。)

更细微处的变化体现在“容器是如何分配内存的”这样的地方。原则上来说,它们仅仅影响那些喜欢追根究底的STL用户(也就是那些试图让容器使用他们自己的内存分配方法的用户,具体的做法参见第4章)。而不幸的是,它们同样也会影响那些简单使用STL容器的用户。随着STL的日益标准化,我们应该做好学习大量新条约的准备。

商业编译器

现在的市场中存在着大量商业C++编译器，它们一般都有STL完全更新的版本。例如，本书中所提供的代码就是从Dinkum C++库的STL部分提取出来的，经过重新组织后在此处作为教学代码。这使得这些代码与Microsoft Visual C++、IBM Visual Age以及其他厂商提供的C++编译器中的STL代码极为相似。（造成它们之间还存在不同之处的主要原因是，代码必须能够无条件地符合编译器所提供的特性，即使是编译器本身并没有完全支持C++标准中所给定的全部语言特性。）虽然其他的厂商可能会提供一些有着不同语源的STL代码，但它们也必须符合C++标准。用于标准C++库的商业验证套件现在已经可以在市场中获得了，提供这样的套件的公司包括：Dinkumware有限公司和Perennial Software等。（这些验证套件的作用是给那些库厂商以某种荣誉。当升级以及除错工作都已经达到某种程度而不需要继续大力进行下去后，你就可以期待不同的商业化的标准C++库的新版本更加统一。）本书中所提供的代码已经通过了Dinkum C++ Proofer和Dinkum Abridged Proofer中所有的相关测试。上述的两种工具都是由Dinkumware有限公司提供的。

本书将STL视为标准C++所定义的那个巨大的库的一个相对独立的子集。而C++中的库又是被设计成为与ANSI/ISO标准C（ANSI89和ISO90）中的库一脉相承，可以协同其一起工作。因此，你可以将本书视为对P.J. Plauger以前所完成的两本书，《The Standard C Library》（Pla92）和《The Draft Standard C++ Library》（Pla95）的一个后续版本。几乎所有C++程序员感兴趣的有关库的内容都可以在这三本书中找到。

本书每一章的组织结构都与本简介的结构差不多：

- 首先是一个讲述“背景知识”的小节，用于引入主题。
- “功能描述”小节用于提供一些简洁的参考信息。在阅读本书时，你可以从这节中发现很多有用的信息。因为在此节中指出了该章的教学主旨，并且通常还包括一些没有在该章中说明的参考信息。
- “使用...”小节介绍如何使用在该章中所描述的以及你可能希望使用的一些特性。
- 在“实现...”小节中讨论如何使这些特性在典型的STL实现中正常运行。在该节中将给出组成每个头文件的实际代码。
- 在“测试...”小节中讨论在测试一个STL实现的基本功能时产生的问题。在该节会给出针对每一个头文件的测试代码。也就是说，该节会提供一些简单但具有说明意义的示例代码。
- 每一章的最后是“习题”小节，它的目的是使你思考刚刚所阅读的信息。如果你把本书作为一门课程的一部分来阅读的话，那么它们也可以作为家庭作业用以检验学习成果。

与本章的主旨（简介）一致的是，本章剩下的几个小节也都是把STL

作为一个整体而不是一些特定的部分来讨论的。对于各个部分的详细讨论将由本书的其他章节来完成。

功能描述

标准头文件

所有的标准C++库的实体都定义或声明在一个或多个标准头文件中。所有这些头文件包括那些在标准C中已经存在的头文件，以及一些由标准C++所引入的新的头文件。组成标准C++中的STL部分的那13个头文件已经在“惠普实验室的代码”一节中列出。可以使用一条include指令将标准头文件中的内容包含到程序中，如：

```
#include <algorithm> // include all algorithms
```

可以以任何顺序来包含那些标准头文件，或是将一个标准头文件包含多次，或是在两个或多个标准头文件中定义同一个宏或类型。但是，请不要在声明中包含标准头文件，不要在包含标准头文件前定义那些与关键字^②使用相同名字的宏。

一个标准的C++头文件通常还会包含其他一些标准的C++头文件，用以定义它所需要的数据类型。然而，一个标准的C头文件从不包含其他的标准头文件。标准头文件仅仅声明或定义那些在C++标准中所描述的与其有关的实体。

库中的每一个函数都在某个标准头文件中声明。与在标准C中的不同点在于，C标准头文件从不提供masking macro（即与函数同名的宏。它会屏蔽函数声明，并且达到同样的效果）。

名字空间 std

所有在标准C++头文件中出现的名字，除了operator new以及operator delete之外，都定义在一个叫做std的名字空间(namespace)内，或是定义在std所嵌套的名字空间内。包含一个标准的C++头文件一般不会将任何库名字引入当前的名字空间。

除非明确指出，否则你不可能在名字空间std内，或是std所嵌套的名字空间内定义自己的名字。该名字空间被保留给了标准C++库中的组件，现在如此，将来也是如此。

自由度

在声明C++库中的类型及函数时，不同的实现有着其特定的自由：

- 在标准C库中，函数的名字可能有着extern "C++"或extern "C"这样的连接属性。也就是说，可以直接包含适当的标准C头文件，而不需要通过内联(inline)的方式来声明库实体。
- 库中属于某个类的成员函数的名字可能有多于文档中所列其他

^② 此处的关键字并不是指在C++标准中所定义的那些关键字，而是指那些可能会在标准头文件中定义的关键字，如 iterator、reference 等。——译者注

函数的函数签名。你可以确信此处讨论的函数调用与你所预期的一样，但不能依赖从库中所取得的成员函数的地址来进行函数调用（实际的类型可能与你所预期的不一样）。

- 有的库类可能拥有一些没有文档资料记载的非虚（non-virtual）基类。实际上，一个有文档资料记载为从其他类所派生出来的类很有可能就是由其他的一些没有文档资料记载的基类所派生出来的。
- 一个被定义为某种integer类型同义词的类型也可能会与其他不同的一种integer类型完全一样。
- 库函数如果没有指定任何的异常（exception）规范的话，那么就意味着该函数可以抛出任意的异常，除非我们在函数的定义处明确限制了可能抛出的异常。

保证

在另一方面，下面的这些限制条件能对你有所帮助：

- 标准C库不使用masking macro。只有特定的函数签名而不是函数名本身被保留下来另作其他用途。
- 只要不是属于某个类的成员，库函数的名字就没有其他附加的、没有文档资料记载的函数签名。也就是说，通过使用地址来进行函数调用是可以信赖的。
- 被描述成为虚拟的基类及成员函数就一定为虚拟的，而被描述为非虚拟的就一定不是虚拟的。
- 标准C++库中所定义的两种类型一般都不一样，除非它们被显式指定为一样。
- 库所提供的函数，包括那些可替换函数的默认版本，可以向外抛出尽可能多的异常（只要是在异常规范中所列出的异常种类就行）。库所提供的析构函数都不会向外抛出异常。标准C库中的函数可能会传播异常，如`qsort`调用一个会抛出异常的比较函数，但`qsort`函数本身（以及其他一些标准C的函数）并不会抛出任何异常。

使用 STL

为了获得最佳的可移植性，必须在每个翻译单元中都显式包含所需要的标准C++头文件。从一种实现移植到另一种实现上时，以前可以正常工作的一些代码有可能不能正常工作。通常我们可以这样认为：在一种很明显的依赖关系下，一个头文件必须包含另外一个头文件；但对于给定实现如何处理实际中的依赖关系我们仍可能猜错。

名字空间

在C++的标准化进程中，名字空间被加入到C++语言中。你可以把名字空间想像成为类的包装器。它给那些在程序中定义的名字强加了一个层次，这一点就像我们使用目录结构来组织磁盘上的文件一样。

例如，可使用实际上的std::cin来指代名字cin。也可以换一种做法，使用下面的声明形式：

```
using namespace std;
```

这样做会使所有在库中定义的名字都暴露在当前的名字空间中，于是就不再需要在这些名字前添加前缀std::了。如果想把在库中定义的所有名字都暴露在全局名字空间（global namespace）中，那么请在每个源文件的开始处都写上这句声明，紧跟在include指令后。同样，如果包含一个符合C标准的头文件（如<stdio.h>等），在该头文件中所声明（或定义）的名字就会自动暴露在全局名字空间中。（如果只想使用在名字空间std中声明的名字，可使用标准C++头文件<cstdio>。）

注意，宏名字并不遵循名字空间的嵌套规则。

对于这种综合声明的形式，人们有着不同的意见。一些人认为所有新的C++代码必须在库名字前面严谨地添加上std::前缀。另外一些人认为这样做将会导致代码的杂乱无章，它将附带地使将现有的C++代码升级到符合标准形式的代价有所增加。我们所使用的测试代码使用了那种综合声明的形式，这样做的主要目的是为了简化将它们移植到那些不支持名字空间（或是支持得不是很好）的编译器时的转化工作。我们并没有因此就将这种容易导致众怒的编码风格作为一种标准来推行。

方言

作为最后的警告，我们必须再一次强调，C++中的一些方言（dialect）依然存在。一些新近的语言特性（如名字空间，成员模板，模板的部分特化等）都是在C++标准逐步形成的过程中添加到该语言中的。在本书的写作过程中，并不是目前市面上可以得到的所有商业化的C++编译器都实现了完全支持STL所需的所有语言特性。在我们认为恰当的地方，我们会指出如果实现没有提供语言中某种特定的特性的话，STL中的哪些特性将不会出现。

但是请不要让我们的保守影响你使用STL。目前编译器都在向着C++标准稳步靠拢。即便是在现在的情况下，你可以信赖的、跨实现的STL子集也已经非常庞大了，其功能也非常强大。

实现STL

本书所提供的代码是作为随同C++编译器所提供的标准头文件的集合而出现的。在包含这些标准头文件的目录中，你很可能会发现一些头文件的名字与我们提供的名字十分相似，如deque或iterator等。然而，请注意：不要用本书提供的代码去替换标准头文件。STL本身在很大程度上是独立的，但它还是和标准C++库中的其他部分有着接口。如何实现这些接口在不同的实现中都存在它们自己的合法做法。我们将在以后的

章节中讨论有关接口方面的内容，但我们在此处能做的就是提供一种可行的解决方案。但对于一个给定的编译器来说，该方案并不一定正确。

编码风格

本书中代码的编码风格遵循下面的一些规则。其中许多规则也适用于其他项目，有一些则比较特殊：

- 每个秘密的（secret）宏或全局名字（即为我们的STL实现所私有的名字）都以大写字母开头，如Getint。（为了教学上的可读性，我们在此省略了应该出现在库名字前面的下划线。）
- 代码的布局格式适当统一。在函数中对象的声明通常都应该出现在尽可能深的嵌套层中。嵌套的缩进形式应该忠实地反映出控制结构的嵌套形式。
 - 在头文件中，所有的函数都被定义为内联函数，这样可以保证它们的独立性。没有用于库的被分拆的源代码存在。
 - 为了避免代码的形式过度不规则，相对于传统的C++源文件中的函数定义来说，内联函数应该尽可能写得紧凑些。（即使因此丧失了代码的部分可读性，有的头文件还是显得很大。）
 - 代码中不包含有任何register声明。理由是我们很难做到聪明地放置它们，而且它们通常都会使代码变得杂乱无章。而且，现在的编译器一般都忽略程序员自己所做的register声明，自动进行寄存器的分配。通常，编译器所做的寄存器分配工作比程序员做的要好很多。
 - 本书把每个C++源文件用程序清单的形式显示出来。每个程序清单的名称即该文件的文件名。

有时，这样做产生的代码看起来比较密集。在一个典型的项目中，程序员通常都会加上一些空格及注释，这样做会使程序增大至少30%。（有的程序员可能会使程序变得更大，但我们在本书中这样做是为了从密集的文字中获得更多对代码的描述。）

这些代码同样也包含了一些头文件，不过它们被分割成几个小片段出现。在实际的实现中，这种分割的目的在于：减少在这些头文件中出现的循环引用。在今天高速计算机的运行条件下，编译时间已经只能算是一个小问题了。我们会在需要添加头文件的时候对此进行详细的解释。

实现头文件

作为一个例子，程序清单0-1中列出了文件utility，它实现了头文件<utility>。我们将在第2章中对这个小型的头文件进行更加详细的讨论。现在，我们主要关注头文件的总体结构。

```
程序清单0-1: // utility standard header
utility #ifndef UTILITY_
#define UTILITY_
#include <iostream>
```

```
namespace std {

    // TEMPLATE STRUCT pair
    template<class T1, class T2> struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        pair()
            : first(T1()), second(T2()) {}
        pair(const T1& V1, const T2& V2)
            : first(V1), second(V2) {}
        template<class U1, class U2>
            pair(const pair<U1, U2>& X)
                : first(X.first), second(X.second) {}
        T1 first;
        T2 second;
    };

    // pair TEMPLATE OPERATORS
    template<class T1, class T2> inline
        bool operator==(const pair<T1, T2>& X,
                          const pair<T1, T2>& Y)
        {return (X.first == Y.first && X.second == Y.second); }
    template<class T1, class T2> inline
        bool operator!=(const pair<T1, T2>& X,
                          const pair<T1, T2>& Y)
        {return (!(X == Y)); }
    template<class T1, class T2> inline
        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); }
    template<class T1, class T2> inline
        bool operator>(const pair<T1, T2>& X,
                         const pair<T1, T2>& Y)
        {return (Y < X); }
    template<class T1, class T2> inline
        bool operator<=(const pair<T1, T2>& X,
                         const pair<T1, T2>& Y)
        {return (!(Y < X)); }
    template<class T1, class T2> inline
        bool operator>=(const pair<T1, T2>& X,
                         const pair<T1, T2>& Y)
        {return (!(X < Y)); }
    template<class T1, class T2> inline
        pair<T1, T2> make_pair(const T1& X, const T2& Y)
        {return (pair<T1, T2>(X, Y)); }
```

```
// TEMPLATE OPERATORS
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); }
}
/* namespace std */
#endif /* UTILITY_ */
```

首先，也是最明显的是，头文件被通常用于解释的注释及用于防范措施的守卫宏（macro guard）所包围：

```
// utility standard header
#ifndef _UTILITY_
#define _UTILITY_
.....
#endif /* _UTILITY_ */
```

许多C和C++程序员都把这当作一种常用的惯用法，用以防止一个头文件被程序包含多次。typedef或者宏定义可以在一个翻译单元中出现多次，只要它们每次的定义结果都一样就可以了（这也称为良性重定义，benign redefinition）。但实质上所有其他实体至多只能定义一次。在此处我们定义了宏_UTILITY_，把它作为一个开关来使用：在第一次使用（即被包含）时它就被设置（即被定义），然后我们就可以使用它来确保该头文件只被考虑一次，即使在一个翻译单元的include指令中多次包含它也是如此。这就可以防止那些讨厌的多次定义（这种情况经常出现在使用了大量include指令的用户代码中）。

宏名字_UTILITY_的选择一定要小心。C和C++标准中都保留了部分名字集给实现者使用。尤其是那些以下划线开头，后面紧接着一个大写字母的名字更是要严格控制其使用范围（只能由实现者使用）。也就是说，使用这个集合中名字的实现可能会干扰其他程序的正常翻译。（同一个库中不同部分的实现更加需要注意，千万不要一不小心就彼此冲突。）

包含

下一行就是include（包含）指令：

```
#include <iostream>
```

头文件<iostream>是标准化委员会的一个创举。它为在其他头文件中定义的部分模板及类提供了一些前向引用（forward reference，即对不完整类型的声明）。一般人很少会包含这个头文件，但对实现来说，它简直就是上天赐予的礼物。它可以帮助我们打破在标准C++库的不同组件中出现的循环依赖关系。

严格地说，在这个实现的头文件<utility>中，<iostream>根本就无用武之地。然而，它却能简化标准头文件中包含关系的结构。在这个特殊的实现中，其他包含<utility>的头文件都需要包含<iostream>。于是我们就可以在此处使用 include 指令来包含它，这样做将对以后产生很大的便利。附录 A 将概述 STL 头文件是如何与其他头文件（不但是和 STL 有关的头文件，也包括其他的标准 C++ 库中的头文件）相互作用的。

我们本可以掩盖这种实现细节，把它作为一种不受欢迎且容易分散注意力的做法给忽略掉。然而，我们宁可强调在大量的软件中顺利地协同工作的困难程度。这个特殊的例子同样也强调了：STL 并非存在于真空中。虽说在很多方面它是一个独立的组件，但它更是一个（更）大型的库的一部分。

名字空间

在头文件<utility>的结构中，接下来的部分是名字空间声明：

```
namespace std {  
    ...  
}; /* namespace std */
```

这个特殊的声明确保在大括号中定义的所有名字（宏名字除外）都位于名字空间std内。在稍早的时候我们说过，名字空间的行为更像是一个类包装器，但这样说只能算是接近正确。与类包装器不同的是，名字空间的声明可以在同一个翻译单元中多次出现，并且代表同一个名字空间。名字相同的部分会有效地连接起来。

产品代码

实际产品代码不可能像我们所展示的那样整洁有序。通常，为了在不同的实现之间取得很好的移植性，库中的代码中充满了各种不同的条件编译指令。给定的实现可以打开它所能达到的条件开关，在代码中反复使用C++中的方言。对于不支持的特性来说，解决的方法有两种：一是禁用它们；二是提供一种简单的替代形式。你可以很容易地得到一个中文菜单，代价就是头文件的可读性受到影响。本书中的代码确实是基于一个完整的STL产品。在此处我们以一种最通常的形式把这些代码展现给大家，目的是为了更好地利用这些代码来辅助我们的讲解，使大家能够很容易地写出具有可移植性的商业化代码。但这本书的最主要的

目的是讲解STL的复杂难懂之处。就算不再往其中添加进新的特性，这已经够复杂了。于是，我们决定从该代码中抽取出部分代码，稍加修饰后当作实际的教学代码使用，这些代码在本质上能够完成同样的功能。这两者之间最主要的不同之处在于：用作教学目的的代码其可读性更高；这其实也是我们设计它们的初衷。

如我们在前面所提到的，在可读性方面我们也作了小小的让步。程序清单0-1中大量使用了传统的用于表示类型的名字（如T1和T2）以及表示对象的名字（如X和Y）。这种使用名字的做法在C++标准中经常（如果不是“总是”的话）出现。它们也确实遵循了C++中对事物（包括类型、对象等）命名的一种广泛采用的惯例（虽然实际上对象的名字不太可能以大写方式出现）。也就是说，头文件“内部”的名字的选择是为了让我们尽可能地理解它们。

惟一的问题在于，这样做并不完全遵循C++标准。并不是说使用它们存在着破坏用户代码的问题。参数的名字，以及嵌套在类及函数中的其他名字，并不会被外界的代码所认知，当然也就不会对外界环境产生所谓的名字污染（name pollution）。问题是以另一种表现形式而存在的。在各种不同的环境下，用户选择使用的名字可能会渗透到库声明和定义中，并且以一种危险的方式改变这些名字的原有意义。这种连锁反应通常很难诊断出来。有的时候，你可能会写出这样的程序：它们可以正常编译，但执行起来却与你所预期的结果有着天壤之别。

最容易导致这样的问题出现的情况是：定义一个有着普通名字（如T1或者X）的宏。如果在第一次包含库的头文件前这样做了，就会导致重写任何使用该名字的声明。不管你是有意还是无意地试图破坏库，在库中都缺乏一种严肃的编程方法来避免受到这种攻击的破坏。

是的，我们知道，人们经常使用诸如T1和X这样的宏名字。一个有着严密思维的程序员不应该幻想并沉迷于这样的行为之中。许多程序员很快就会意识到，在把类型、宏及其他名字的命名方式格式化之后，维护它们的工作将会得到极大的减轻。但是C++标准仍然允许在程序中使用这样的名字来定义宏——它们通常是作为为用户保留的name space的部分而使用的。（name space，或名字集，虽说和C++中的namespace声明十分相似，但请不要把这两者混为一谈。）

实际上，使用这样普通的名字来声明类型经常会产生问题。在这里我们并不想给出任何示例，但这个问题在C以及C++中确实是一个尽人皆知的事实。在太多的环境中，外部包容环境中类型意义的改变将会对内部嵌套环境中的声明（甚至是某些表达式）的解释产生深刻的影响。

在产品库代码中，所有名字在被当作宏名字使用时都应该同样小心，应尽量让它保持它所代表的真正目的。这意味着我们应该这样写：

```
template<class _T1, class _T2> inline  
bool operator==(const pair<_T1, _T2>& _X,  
                  const pair<_T1, _T2>& _Y)  
{return (_X.first == _Y.first  
&& _X.second == _Y.second); }  
  
template<class _T1, class _T2> inline  
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); }
```

虽说与程序清单0-1中的代码相比，上面代码的可读性可能稍差点，但它确实是节选自实际的产品代码。你可能会理解，为什么我们没有将许多这样的代码强加给那些原本可能会对学习STL更感兴趣的人们。

测试 STL

测试本身是一个很严肃的主题。（从Pla95中可以获得对此话题更广泛的讨论，而那些也不过只是一些表层的内容而已。）我们都应该知道，一个典型的库函数不可能被详尽地测试。输入值的组合通常都是一些天文数字，特别是对浮点参数更是如此。请想一下对于模板而言，可能出现的最糟糕的情况是什么？（可为任意类型参数对模板进行特化。）

实际上，我们甚至不需要做任何尝试。作为替代方案，我们提供了一系列简单的测试程序。它们中的每一个都会对STL头文件所提供功能的部分或全部进行测试。你将会发现，这些测试程序主要关注于一些外部行为。这意味着，它们本质上就是一个简单验证套件。然而，它们偶尔也会闯入“测试内部结构”的领域中。某些实现中出现的错误有时是如此普遍，如此危险，以至于我们很难做到不去测试它们。另外，我们所给出的这些测试程序很少进行性能测试。

你最终将会发现，这些测试都非常粗浅和简单。然而，即使是简单的测试也能有效地解决实际中的问题。对于每个特性，我们都提供了一个基本且可行的例子，这可以使你能够切实地从错综复杂的STL之中获得真正的好处。可以仅凭少量的几行代码来证实一个模板是否达到了它的最初设计目的。这就再次保证了实现的正确性。当发生变化时（毫无疑问一定会发生），重复这些测试就可以重新保证正确性。基于上述理由，我们相信：写出简单的测试代码以及保存它们是非常值得做的事情。

我们发现：当执行成功时，那些简单且有把握的测试将打印出一条标准可靠的信息，并且在退出时返回一个表示执行正确的状态。如果有

可能的话，测试代码就不再告诉我们其他的信息了。

我们通过在每个头文件的名字前面添加字母“t”来构造测试代码的文件名，如果有必要的话，还需要将它们截短，以使得文件名（不包括后缀“.c”）不超过8个字符。在以此规则得到的文件名稍微有点奇怪的同时，在各种操作环境下文件管理系统的麻烦会少很多。也就是说，utility.c就是用来测试头文件<utility>的。它测试该头文件所定义的是否是它所支持的，并在测试结束时显示一个可靠的测试信息：

```
SUCCESS testing <utility>
```

然后就正常地退出。如果愿意的话，可以通过一个普通的命令脚本来运行本测试或其他测试，并且简单地对它的退出状态进行检测。

注意，在每个这样的测试文件中都定义了其本身的main函数。把每个文件与标准C++库链接后，就可以得到独立的测试程序。不能把它们中的任何一个加入到你自己的标准C++对象-模块(object-module)库中。因为每个文件都是被设计成一个独立的程序来运行的。

习题

习题0-1

在先前给出的那个例子模板中：

```
template<class T> inline
    T sum_all(T *first, T *last)
    {T sum;
     for (sum = 0; first != last; ++first)
         sum += *first;
     return (sum); }
```

请说出模板是否可以被下列的类型成功地特化：

- int类型
- float类型
- 一个void类型
- 一个指针类型

对于你所认为不能的那些类型，试解释其原因。

习题0-2

对于上面给出的模板sum_all，请列出对于能够成功特化该模板的类型参数T所需要支持的全部操作。

习题0-3

定义一个类X，使得当x是类X的一个对象时，调用sum_all(x)将得到定义良好的结果。

习题0-4

sum_all(&x, &x)的结果是什么？通过这种方式来得到x的值是否是一种很好的做法？请说出你的理由。

习题0-5

假设需要检测用类型float去特化该模板时sum_all是否会按照所预

期的方式工作，请列出你的测试代码。

习题0-6

[较难] 假设我们把上题中的类型float换成任意的类型T，请列出你的测试代码。

习题0-7

[特难] 如何更改sum_all的定义，使得对于有着奇怪定义operator=(const T&)^③以及operator+=(const T&)的参数类型T，其模板的特化^④依然有着“明智的”行为。

-
- ③ 在函数原型定义中如果存在没有实际意义的参数名字，这通常意味着该参数在函数中从来没有被引用过。这样写的目的—是为了函数的重载(overload)，二是为了让编译器不会对此产生报警信息(某些编译器对于从未在函数中使用过的参数会产生一个报警信息)。——译者注
 - ④ 关于模板特化或部分特化，可参看中国电力出版社引进出版的《C++ Primer 中文版》。——译者注

第1章 迭代器

背景知识

迭代器在STL中起着粘合剂的作用，用来将STL的各个部分结合在一起。从本质上来说，STL提供的所有算法都是模板，我们可以通过使用自己指定的迭代器来对这些模板进行特化。我们也建议大家按照这种格式写出自己的算法模板。同样，在标准库中所提供的所有容器也都提供了这样的迭代器，用以存取它们所管理的数据序列。同样也建议大家在自己定义的容器中提供与之相配的迭代器。由于对象指针可以被当作任意种类的迭代器使用，所以这不会给你带来许多限制。

<utility>

<iterator>

<memory>

在此，我们主要以<utility>、<iterator>和<memory>这三个头文件为线索展开讨论。这三个头文件中的第一个能提供的信息远比迭代器技术要多。但是，对这三个头文件的使用一直贯穿着整个STL，它们也用来提供实现STL所必需的机制。在接下来的三章中，我们将分别对它们进行更详细的讲解。在此先提醒大家一下，对于这些细节方面的讨论有时十分单调乏味，而且在开始讨论算法和容器前，我们几乎很难察觉到这些细节的好处。从现在起，我们仅仅关心迭代器所共有的一些属性，像如何将它们分类以及不同种类的特点是什么等。

功能描述

C++中的迭代器相对于C中的对象指针来说更加一般化。指针本身就可以作为定义好的迭代器来使用。这种一般化行为主要体现在可以在C++中声明新类，然后对于这些类中的大部分操作符进行重载（overload），赋予它们新的意义。甚至还可以让迭代器指向一个奇异值（singular value），该迭代器的行为与一个定义有问题的指针十分相似。

输出迭代器

可使用迭代器来存取有序序列中的元素。在此，“存取（access）”是一个一般化的术语，我们既用它来表示将值存储到对象中，也用它来表示从一个对象中获得它所保存的值。如果需要创建一个新的值序列，并且以有序的方式来产生值，可以写一个如下的循环语句：

```
for (; <not done>; ++next)
    *next = <whatever>;
```

在此，`next`表示一个迭代器类型为`X`的对象，`<not done>`是一个用来检测循环是否应该终止的谓词(`predicate`)，而`<whatever>`则是一个表达式，类型为序列中元素的类型`T`，最起码也是那种很容易转化为`T`的类型。

表 1-1：
输出迭代器
的属性

表达式	结果的类型	含 义	注 释
<code>X(a)</code>	<code>X</code>	产生 <code>a</code> 的一个拷贝	析构函数是可见的 <code>*X(a) = t</code> 与 <code>*a = t</code> 的作用相同
<code>X u(a)</code> <code>X u = a</code>	<code>X&</code>	<code>u</code> 是 <code>a</code> 的拷贝	
<code>r = a</code>	<code>X&</code>	将 <code>a</code> 赋值给 <code>r</code>	结果 <code>*r = t</code> 和 <code>*a = t</code> 作用相同
<code>*a = t</code>	<code>void</code>	在序列中存储新元素	
<code>++r</code>	<code>X&</code>	指向下一个元素	<code>&r == &++r</code>
<code>r++</code>	可以转换为 <code>const X&</code>	{ <code>X tmp = r;</code> <code>++r;</code> <code>return tmp;</code> }	
<code>*r++ = t</code>	<code>void</code>		

注释：`X` 是迭代器类型；`a`的类型为`X&`；`T`是元素类型；`t` 的类型为`T`。

具有这样的类型`X`，可以这样使用的迭代器，称为输出迭代器(`output iterator`)。从上面的讨论来看，一个输出迭代器至少还需要定义下面这些操作：

- `*next = <whatever>`将`<whatever>`的值赋给序列中将要产生的下一个元素。
- `++next`改变`next`的值，使之指向序列中的下一个元素。

在C中经常使用一种稍有不同的方法来完成同样的操作：

```
while (<not done>)
    *next++ = <whatever>;
```

为了支持这种写法，表达式`*next++ = <whatever>`必须将上面所描述的两种操作结合起来，使之具有和C中的指针一样的意义。

实际上，一个输出迭代器所能保证的一点也不比上面列出来的属性多（甚至不能判断一个输出迭代器是否前进得太多）。围绕着输出迭代器的一系列操作包括一个复制构造函数、一个析构函数、一个赋值操作符，它们都具有通常所需的属性。与这些可怜的属性相对应的是，输出迭代器允许以各种各样的方式来实现它们。甚至可以将一个合适的输出迭代器装扮成每次存储一个输出记录的形式，使之输出到一个文件中。

表1-1以接近数学中常用符号的形式形象地描述了输出迭代器的各种属性。对于其他种类的迭代器，本书也将提供同样形式的表格。我们发现，使用这样的符号记录方式对于示例代码及上面的注释来说能起很大的作用，但也没有必要使用它来完全替换掉那些代码及注释。在某些对于输出迭代器来说非常重要的限制条件中，这样的表并不能完全清楚地表达它们。如：

- 必须保证输出迭代器在存储每一个元素后得以增加。
- 必须保证输出迭代器在两次存储的间隔内增加的次数不会超过一次。

如果输出迭代器实际上就是指针的话，上面的限制就不明显了。但当你看到STL使用一些具有特殊目的的输出迭代器的技巧时，就会知道对于这些限制的需求了。你可能会更欣赏表1-1中一些相对深奥的条目，如有关 $r++$ 的返回值类型等。但从现在起，只需要记住，输出迭代器只是在类似上面给出的循环语句中使用。

输入迭代器

输入迭代器（input iterator）用来产生新的序列。为了顺序存取已有的值，或只是需要对已有的序列进行遍历，可以这样做（和前面的方法略有不同）：

```
for (p = first; p != last; ++p)
    <process>(*p);
```

在这里，`p`、`first`和`last`都是迭代器类型`X`的对象，`<process>`是一个函数，它能够接受元素类型为`T`的参数。对于上面出现的由元素组成的序列，我们用一个半封闭区间`[first, last)`来表示，其中，`first`和`last`各代表一个迭代器。

注意，`last`其实并不表示序列中的任何一个元素。实际上，它表示的是“end-of-sequence”标记，如在实际的序列末端紧接着的第一个元素。空序列是满足`first == last`的序列。C标准中也经常使用这种方法来操作C中的数组——可以将指向数组末端后的第一个元素的地址存储在指针中，但不能得到这个指针所指向的元素的值。迭代器一般化了这种概念，使其更加完美。

具有这样的类型`X`，可以这样使用的迭代器称为输入迭代器（input iterator）。从上面的讨论来看，输入迭代器至少还需要定义下面这些操作：

- 当两个类型为`X`的迭代器`p`和`q`没有同时指向一个元素时，`p != q`就为真（为方便起见，`p == q`通常都被定义成与`p != q`在逻辑上相反）。
- `*p`是类型`T`的一个右值（rvalue）。（右值是一个拥有值的表达式，如`-37`，但它并不一定必须用来引用一个对象。）表达式`p == last`没有定义。
- `++p`将改变`p`的值，将它指向序列中目前所指向元素的下一个元素。表达式`p == last`没有定义。

在C中经常使用一种稍有不同的方法来完成同样的操作：

```
while (first != last)
    <process>(*first++);
```

表 1-2:
输入迭代器
的属性

表达式	结果的类型	含 义	注 释
X(a)	X	产生a的一个拷贝	析构函数是可见的, *X(a)和*a作用 相同
X u (a) X u = a	X&	u是a的一个拷贝	创建完成后 u == a
r=a	X&	将a赋值给r	结果: r == t
a == b	可以转换为bool	相等比较	a和b在同一值域内
a != b	可以转换为bool	!(a == b)	
*a	T	从序列中存取元素	不是 end-of-sequence
a->m	m的类型	(*a).m	T有成员m
++r	X&	指向下一个元素	&r == &++r, r不是 end-of-sequence, r的拷贝无效
(void)r++	void	(void)++r	
*r++	T	{T tmp = *r; ++r; return tmp; }	

注释: X是迭代器类型; a和b 的类型为X&; r 的类型为X&; T是元素类型;
t 的类型为T。

为了支持这种写法, 表达式*first++必须将上面所描述的两种操作结合起来, 使之具有和C中的指针一样的意义。

从这些属性我们可以得知, 只有当可以从first到达last时, [first, last)才是一个有限的序列。换另一种说法就是, first的值在经过有限次的增加后必定会得到一个确定的值, 它与last的值是相等的。

与其兄弟输出迭代器一样, 输入迭代器所保证的也不比上面的这些属性多。同样围绕着输入迭代器的一系列操作包括一个复制构造函数、一个析构函数和一个赋值操作符, 它们都带有通常所需的一些属性。这其中唯一新增的就是指针指向操作符p->m, 只有在类型T是一个结构化类型时, 它才被定义。同样, 输入迭代器也允许有各种各样的实现。

end-of-
sequence
值

你甚至可以将一个合适的输入迭代器装扮成每次存取一个输入记录的形式, 从一个文件中获得所需要的输入。为了能使用这种技巧, 输入迭代器定义了一个end-of-sequence值, 它在大多数情况下是一种end-of-file标记。这个end-of-sequence值存储在last中。用来指向一个真正记录的迭代器first

不会等于last。然而，当first指向的记录是序列中的最后一个元素时，再增加first的值将导致其值发生改变，变为与last相等。也就是说，我们可以以同样的控制方式来完成这两件事情：通过增加指针的值来遍历一个数组与通过增加输入迭代器的值来遍历一个文件。

表1-2以接近数学中常用的符号形式形象地描述了输入迭代器的各种属性。我们再次重申一遍，这样的表并不能将所有有关输入迭代器的重要限制都讲述清楚。例如，它没有说，只有当两个迭代器中至少有一个具有end-of-sequence值时，它们之间的比较才一定有意义。（当使用任意类型的输入迭代器时，对于序列中的两个明显不同的地方，你甚至不能很明确地说出序列是否被越界存取。）建议在处理输入迭代器时遵循我们在输出迭代器处所提出的建议——只在类似上面给出的循环语句中使用它们。

前向 迭代器

从我们先前列举的一些理由来看，输出迭代器和输入迭代器可以说“足智多谋”。它们可以用来处理任意长度的文件。然而，迭代器的一个更加普遍的用法是：存取一个完全存储在内存中的序列。在这种情况下，所需要使用的迭代器就必须具有较少让人觉得惊奇的属性。

举例来说，你仍然只是满足于从头至尾地存取一个序列中的所有元素。但这次你想要对于该序列中的元素同时具有读和写的权利，或者想在先前所存取的任意地方标注出一个位置作为“书签”以方便下次存取。在这种情况下，仍然使用我们在讲解输出迭代器和输入迭代器时所用的那个控制循环，你所需要的很简单，那就是让迭代器看起来更像是一个传统的指针。

运用前向迭代器（forward iterator）就可以达到所有的这些要求。和输入迭代器一样，可以比较两个前向迭代器是否相等，但现在它们可以都为（或都不为）end-of-sequence。当然，和先前所讨论的一样，这两个迭代器的值必须处于同一个值域中。和输入迭代器一样，前向迭代器也可以有end-of-sequence值，它的意义和数组中“off the end”^①元素的地址差不多，你也可以将它想像成其他任意的end-of-sequence标记。你还可以用一个前向迭代器的多个有效拷贝来指向当前序列中的任意位置。

可以把前向迭代器想像成为一个指向单向链表中元素的指针。你可以明确地指出它是否指向链表的末端（用null来标记end-of-sequence）。如果它指向的元素不是end-of-sequence，就可以通过它来存取该链表的元素，或是把它移到序列中下一个元素的位置。但是不能让它回退，也不能通过它来直接存取链表中的任意元素。

严格地讲，你“可以”通过使用前向迭代器来完成这些操作中的一部分。但这具有一定的欺骗性，不管链表的长度如何，你都不可能在恒定的

^① 即数组中最后的那个元素后面所紧接着的那个元素。一般来说，它并不具有实际意义，通常我们用它的地址来判断数组是否结束。——译者注

时间内完成这些操作。对于所有的迭代器来说，都存在着一个隐式要求：我们对于迭代器的所有操作都不能有太大的开销。不能随着迭代器所指向序列的长度改变而改变操作所需花费的时间。

表1-3以接近数学中常用符号的形式形象地描述了前向迭代器的各种属性。

表1-3：
前向迭代器
的属性

表达式	结果的类型	含 义	注 释
X()	X	产生一个默认值	析构函数是可见的，值可以是end-of-sequence
X u	X&	U具有默认值	
X u = a			
X(a)	X	产生a的一个拷贝	析构函数是可见的，*x(a)和*a 的作用相同
X u(a)	X&	u是a的一个拷贝	创建完成后u == a
X u = a			
r = a	X&	将a赋值给r	结果: r == a
a == b	可转换为bool	相等比较	a和b在同一值域中
a != b	可转换为bool	!(a == b)	
*a	T&	从序列中存取元素	a不是end-of-sequence, a == b 意味着 *a == *b
*a = t	T&	在元素中存储	a不是end-of-sequence, x 是可变的
a->m	m的类型	(*a).m	T有成员m
++r	X&	指向下一个元素	&r == &++r, r 不是 end-of-sequence, r == s 意味着 ++r == ++s
r++	可转换为常量x&	{ x tmp = r; ++r; return tmp; }	
*r++	T&	{ T tmp = *r; ++r; return tmp; }	
注释：X是迭代器类型；a和b的类型为X；r和s的类型为X&；T是元素类型，t的类型为T。			

比前向迭代器应用更为广泛的一种迭代器同时支持递增及递减操作。通过使用这种迭代器的这些特性，许多算法都得以以更加高效的方式实现。STL所定义的双向迭代器（bidirectional iterator）就是这样的一种迭代器，

与前向迭代器相比，它多了可以在序列中逆向移动这种特性。

我们可以将双向迭代器想像成指向一个双向链表中的元素的指针。我们可以明确地指出它是否指向该链表的末端（用`null`来标记`end-of-sequence`）。如果迭代器指向的不是链表的末端，我们就可以通过它来存取该链表中的元素，或是将它移到序列中下一个元素的位置。如果迭代器指向的不是链表中的第一个元素，我们就可以把它回退到序列中前一个元素的位置。但是我们不可能通过它直接存取链表中任意位置的元素，至少我们不可能通过恒定时间的操作来达到这个目的。

表1-4以类似于常用数学符号的形式描述了双向迭代器的各种附加属性。所有适用于前向迭代器的属性，也同样适用于双向迭代器。

**表1-4:
双向迭代器的
附加属性**

表达式	结果的类型	含 义	注 释
<code>--r</code>	<code>X&</code>	指向下一个元素	对于一些 <code>s</code> , <code>++s = r</code> , <code>&r == &--r</code> , <code>r</code> 不是 <code>end-of-sequence</code> , <code>r == s</code> 意味着 <code>-r == --s</code>
<code>r--</code>	可转换为常量 <code>x&</code>	{ <code>x tmp = r;</code> <code>--r;</code> <code>return tmp;</code> }	
<code>*r--</code>	<code>T&</code>	{ <code>T tmp = *r;</code> <code>--r;</code> <code>return tmp;</code> }	

注释：`X`是迭代器类型；`r`和`s`的类型为`X&`；`T`是元素类型；所有其他属性和前向迭代器相同。

随机存取 迭代器

我们所讨论的最后一种迭代器具有C语言中对象指针的所有强大功能。除了在双向迭代器中所提到的那些特性外，随机存取迭代器还支持与整型值的加减操作、指针之间的相减、两个迭代器在序列中的顺序比较，以及使用下标方式操作该迭代器等。某些算法只能依靠这种程度的弹性才有可能运作良好。（通过二分法来对序列进行排序和快速查找就是这样的两个例子。）

然而，请记住，随机存取迭代器仍然不是C语言风格的指针。例如，它们之间存在的一个区别是某种类型`Dist`，它可以是也可以不是一种最基本的整数类型。我们可以将作用于整型值的算法应用于`Dist`对象，但这并不能够阻止我们将`Dist`定义为一个类。

表1-5以类似于常用数学符号的形式描述了随机存取迭代器的各种附加属性（相对于双向迭代器来说）。所有适用于双向迭代器的属性同样也适用于随机存取迭代器。

表 1-5:
随机存取迭代器的附加属性

表达式	返回值类型	含 义	注 释
$a < b$	可转换为bool的任意类型	从a可以到达b	a和b处于同一个值域中
$a > b$	可转换为bool的任意类型	$b < a$	
$a \leq b$	可转换为bool的任意类型	$!(b < a)$	
$a \geq b$	可转换为bool的任意类型	$!(a < b)$	
$r += n$	$X\&$	{ Dist m = n; while (0 < m) --m, ++r; while (m < 0) ++m, --r; return r; }	
$a + n$ $n + a$	X	{ X tmp = a; tmp += n; return tmp; }	
$r -= n$	$X\&$	$r += -n$	
$a - n$	X	$a + (-n)$	
$b - a$	Dist	{ Dist m = 0; while (a < b) ++a, ++m; while (b < a) ++b, --m; return m; }	a和b处于同一个值域中
$a[n]$	可转换为T的任意类型	$*(a + n)$	

注释：X是迭代器类型；a 和 b 的类型为X；r 和 s 的类型为X&；T是序列中元素的类型；Dist是类型X 的差距类型；其他属性和双向迭代器中讨论的一样。

使用迭代器

STL中大量使用了迭代器，它们用于不同的算法和算法所作用的序列之间，起着桥梁的作用。为了本书中其他章节的简洁起见，我们使用迭代器类型的名字（或前缀）来代指迭代器的种类。为了提升其功能，我们把不同的迭代器总结为以下几类：

- **Output** ——假设X为一个输出迭代器，那么它只能通过存储来间接地

拥有一个值V。我们在向输出迭代器中存储了一个值之后，必须将其递增。如：(*X++ = V)、(*X = V, ++X)，或者是(*X = V, X++)。

- InIt——假设X为一个输入迭代器，那么它的值也可以为end-of-sequence。如果它不等于end-of-sequence的话，我们就可以通过间接的方式来存取它所拥有的值V，如：(V = *X)。如果想要取得序列中的下一个元素的值（或是end-of-sequence），必须将其递增，如：++X、X++、或者是(V = *X++)。一旦我们对一个输入迭代器进行了递增，它的所有其他拷贝就不保证一定能够完成下面的操作：比较、间接取值、或者是再对其进行递增等。

- FwdIt——假设X是一个前向迭代器，如果*X是可变的，那么它就可以用来替换输出迭代器（因为这时它本身就是一个输出迭代器）。同样，它也可以用来替换（或者就是）一个输入迭代器。然而，也可以通过一个前向迭代器来读取它所拥有的值（通过使用V = *X），而这个值就是你刚刚存储到它之中的那个值（通过使用*X = V）。你可以同时拥有一个前向迭代器的多份拷贝，它们中的每一份都可以用来间接取值，或是各自进行递增。

- BidIt——假设X是一个双向迭代器，那么我们可以用它来替换一个前向迭代器。并且，你还可以对双向迭代器进行递减，如：--X、X--、或者是(V = *X--)。

- RanIt——假设X和Y都是随机存取迭代器，那么我们就可以用它们来替换一个双向迭代器（因为它们本来就是一个双向迭代器）。我们同样也可以对随机存取迭代器进行许多整数运算（这一点和对象指针一样）。如果N为一个整型对象，那么我们就可以这样写：X[N]、X < Y、X - N、N + X等。

注意，对象指针可以用来替换一个随机存取迭代器（或者说它本来就是随机存取迭代器）。

迭代器分类的层次可以总结为下面的三种情况。如果只需对序列进行只写(write-only)操作，我们可以选择以下的任意一种迭代器：

输出迭代器

- > 前向迭代器
- > 双向迭代器
- > 随机存取迭代器

右向箭头(->)意味着“可以被…替换”。例如，对于那些需要使用输出迭代器的算法，如果我们传给它们一个前向迭代器，它们的执行不应该有任何异常。但反之则不成立。

如果只需对序列进行只读(read-only)操作，我们可以选择以下的任

意一种迭代器：

输入迭代器

- > 前向迭代器
- > 双向迭代器
- > 随机存取迭代器

在这种情况下，输入迭代器是这些种类中功能最少的一种。

最后，如果需要对序列进行读写（read/write）操作，我们可以选择以下的任意一种迭代器：

前向迭代器

- > 双向迭代器
- > 随机存取迭代器

记住，对象指针总是可以当作随机存取迭代器来使用。因此，只要它支持对指定的序列进行适当的读写操作，它也就可以当作任意种类的迭代器来使用。

迭代器的这种“代数学上”的应用几乎是STL中其他部分的基础。清楚地了解每一种迭代器的适用范围及限制条件，对于我们了解迭代器在STL容器及算法中的应用极为重要。

习题

习题1-1

写出下面操作所需的功能最少的迭代器种类：

- 提供无限个0
- 向文件中写入一个值序列
- 实现一个栈（后进先出队列）

习题1-2

下面列出的几种迭代器中，哪一种是可以替换的：

- 输出迭代器
- 只读前向迭代器
- 随机存取迭代器

习题1-3

迭代器同样也可以基于Fortran语言格式的Do循环：

```
for (p = first; p <= last; ++p)
    <process>(*p);
```

试比较这种格式与STL中所选择的那种格式（见本章“输入迭代器”一节）。

- 习题1-4 解释为什么在所写的算法中使用其他种类的迭代器，而不是随机存取迭代器？
- 习题1-5 解释为什么宁愿定义一个仅能通过迭代器来存取的数据结构，而不是让它可以被随机存取呢？
- 习题1-6 [较难] 写出这样一个模板类`bidir<FwdIt>`，当我们用一个前向迭代器类型来特化`FwdIt`时，它的表现就和双向迭代器一样。你会采用何种方法来使它和预期中的双向迭代器行为一致？
- 习题1-7 [特难] 写出这样的一个模板类`ran_read<InIt>`，当我们用一个输入迭代器类型来特化`InIt`时，它的表现就和一个只读的随机存取迭代器一样。

第 2 章 <utility>

背景知识

<utility>是一个很小的头文件。它包括了贯穿使用在 STL 中的几个模板的声明。与最初指定的一样，它看上去还是比较大（虽然与其他的头文件相比还是比较小）。然而，<utility>中的两个成员后来被消除掉了。在惠普的实现中，类 `empty` 与类 `restrictor` 的设计目的是避免一些在模板定义的翻译初始阶段可能会出现的问题。现在，通过改进了翻译技术，这样的类也就不再需要了。

现在，在<utility>中剩下来的就只有模板类 `pair`、一些与之相关联的模板函数和操作符，以及其他四个模板操作符了。该模板类用来将两个对象表示成一个对象——当你想要一个函数返回两个值，或者想用一个容器来存储具有成对值的元素时，这样做就比较方便。那四个模板操作符赋予了`==`操作符新的内涵，它要求与之相应的两个操作值具有相同的类型，通过一致性的方式来定义剩余的相关操作符。

pair

我们可以构造一个带有明确初始值的模板类 `pair<T, U>` 的对象，也可以让默认构造函数提供默认初始值来完成这个工作。在此处我们并没有卖弄什么花哨的技巧——如果需要取得 `x`（为一个 `pair` 对象）的第一个成员对象（其类型为 `T`）的话，我们可以这样写：`x.first`；如果是第二个的话（其类型为 `U`），则使用：`x.second`。

使用成员模板构造函数，也可以用另一个模板类 `pair<V, W>` 的对象来构造模板类 `pair<T, U>` 的对象。相应的成员提供初始值。

make_pair

模板函数 `make_pair` 是一个十分方便的工具。通过使用它们，我们就可以在必要时实时产生出一个 `pair` 对象。然而不幸的是，并不是每一次都能成功。模板函数在检测模板参数时将忽略掉所有的 `const` 属性。

（至少它们可以这样做——某些现在常见的编译器就是这样处理的。）也就是说，我们不能依赖 `make_pair` 来产生一个含有一个或多个常量成员对象的 `pair` 对象。而这些正是容器 `map` 和 `multimap` 有时所需要的（参见第 14 章）。不过，这个模板函数也有它自己的用途。

operator== operator<

有时，比较两个 `pair` 对象 `x` 和 `y` 还是有意义的。由两个模板操作符来完成这些必要的比较操作。在这两种比较操作中，比较容易证明的是“相等（equality）”。如果 `x` 和 `y` 对应的成员对象都相等的话，我们就

可以说 x 和 y 是相等的。但我们怎样才能说出两个 pair 对象谁“小于”谁呢？

我们可以这样假设：在 pair 的两个成员对象中，第一个成员对象的权重（weight）总是大于第二个的。在这种情况下，如果我们可以确认 $x.first < y.first$ 的话，那 x 就小于 y 。还可以继续地加上这样的判断条件： $x.first == y.second \&& x.second < y.second$ 。然而，令我们好奇的是，该操作符的定义实际上并不完全是这样。它完全是由 operator< 来决定的，就如在它的成员对象中所定义的一样：

```
template<class T, class U> inline
    bool operator<(const pair<T, U>& x,
                      const pair<T, U>& y)
    {return (x.first < y.first ||
            !(y.first < x.first) && x.second < y.second); }
```

考虑一会儿后，我们可能相信：实际中使用的这种形式与我们上面所谈论的形式在逻辑上是等价的（equivalent）——至少对于该成员对象的类型来说，如果比较操作符的定义和我们直觉上认为的情况相同，那么上述结论就是成立的。

对于这种特有的形式，有一种理由是：简单即优雅。如果对于每个参数类型来说，需要定义的操作符是一个而不是两个，那么这个模板的定义就很不错了。而另一个理由就比较微妙了。就一个 operator< 来说，比之那些“一堆互相作用的比较操作符”，其由定义顺序导致的依赖关系要更健壮、更具有威力。我们将连同其他一些能够改变序列的顺序的算法来详细地说明这其中的原因（参见第 5 章）。我们现在所说的是，STL 在类似这样的一些细节方面考虑得非常周到。这也正是它的长处之一。

operator!=

现在我们可能会猜测：为什么在头文件<utility>中还需要存在着剩下的那四个模板呢？对于那些定义了 operator==(const T&, const T&) 的类型 T 来说，第一个模板函数为其提供了一个合理的 operator!= 定义。STL 代码中大量地组合使用了这两个操作符，如：在循环控制中用它们来比较序列中的迭代器。在这个模板中，我们所需做的只是为类定义一个 operator==，剩下的事情库会替我们完成的。

operator> operator<=

其他三个模板函数以相同的方式扩展了 operator< 的使用，为其他三种关系操作（operator>、operator<= 和 operator>=）提供了合适的定义。然而，在此处它们的结果却有些争议。这些定义在类型为 T 的对象上实施完全排序（total ordering）。在多数情况下，程序员所需要的就是这种效果。但并不是在所有的情况下都如此。

有些类可能只定义了部分排序（partial ordering）。部分排序的一个

经典例子就是由依赖关系所形成的树（dependency tree）。其他的类可能会选择和常见的排序规则没多大关系的方式来定义这些操作符。在这种情况下，这些模板函数的存在就完全是一件麻烦事了。它们将会导致分析过程中产生歧义或程序产生令人惊讶的代码、或支持那些类设计者认为是惹人嫌的标记法。

这些模板函数都被定义在名字空间 `rel_ops` 中（嵌套定义在名字空间 `std` 中）。它们将不会参加在程序中进行的重载解析（除非是显式地要求它们这样做）。我们将在下面就这个问题继续讨论。

功能描述

```

namespace std {
    template<class T, class U>
        struct pair;

        // TEMPLATE FUNCTIONS
    template<class T, class U>
        pair<T, U> make_pair(const T& x, const U& y);
    template<class T, class U>
        bool operator==(const pair<T, U>& x,
                           const pair<T, U>& y);
    template<class T, class U>
        bool operator!=(const pair<T, U>& x,
                           const pair<T, U>& y);
    template<class T, class U>
        bool operator<(const pair<T, U>& x,
                           const pair<T, U>& y);
    template<class T, class U>
        bool operator>(const pair<T, U>& x,
                           const pair<T, U>& y);
    template<class T, class U>
        bool operator<=(const pair<T, U>& x,
                           const pair<T, U>& y);
    template<class T, class U>
        bool operator>=(const pair<T, U>& x,
                           const pair<T, U>& y);

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

```

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

}

```

包含 STL 的标准头文件<utility>就可以得到在 STL 中普遍使用的几个模板的定义。

在给出了 `operator==` 和 `operator<` 的定义的情况下，四个模板操作符（`operator!=`、`operator<=`、`operator>` 以及 `operator>=`）定义了相同类型的成对操作数上的完全排序。

如果实现支持名字空间，那么这些模板操作符就定义在名字空间 `rel_ops` 中（嵌套定义于名字空间 `std` 中）。要想使用这些模板操作符，就应该在程序中写出如下的声明：

```
using namespace std::rel_ops;
```

它会把这些操作符暴露在当前的名字空间中。

口 `make_pair`

```

template<class T, class U>
pair<T, U> make_pair(const T& x, const U& y);

```

该模板函数返回 `pair<T, U>(x, y)`。

口 `operator!=`

```

template<class T>
bool operator!=(const T& x, const T& y);
template<class T, class U>
bool operator!=(const pair<T, U>& x,
const pair<T, U>& y);

```

该模板函数返回 `!(x == y)`。

口 `operator==`

```

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

```

该模板函数返回 `x.first == y.first && x.second == y.second`。

口 `operator<`

```

template<class T, class U>
bool operator<(const pair<T, U>& x,
const pair<T, U>& y);

```

该模板函数返回 `x.first < y.first || !(y.first < x.first) && x.second < y.second`。

```

# operator<=
    template<class T>
        bool operator<=(const T& x, const T& y);
    template<class T, class U>
        bool operator<=(const pair<T, U>& x,
                        const pair<T, U>& y);

    该模板函数返回!(y < x)。

# operator>=
    template<class T>
        bool operator>(const T& x, const T& y);
    template<class T, class U>
        bool operator>(const pair<T, U>& x,
                        const pair<T, U>& y);

    该模板函数返回 y < x。

# operator>=
    template<class T>
        bool operator>=(const T& x, const T& y);
    template<class T, class U>
        bool operator>=(const pair<T, U>& x,
                        const pair<T, U>& y);

    该模板函数返回!(x < y)。

# pair
    template<class T, class U>
        struct pair {
            typedef T first_type;
            typedef U second_type;
            T first;
            U second;
            pair();
            pair(const T& x, const U& y);
            template<class V, class W>
                pair(const pair<V, W>& pr);
        };

```

上述模板类中存储了一对对象，first 的类型为 T，second 的类型为 U。类型定义 first_type 的含义和模板参数 T 一样，而 second_type 则相当于模板参数 U。

第一个（默认）构造函数将 first 初始化为 T()，second 初始化为 U()。第二个构造函数将 first 初始化为 x，second 初始化为 y。第三个（模板）构造函数将 first 初始化为 pr.first，second 初始化为 pr.second。T 和 U 只

需各自提供一个默认构造函数，一个接受单参数的构造函数，以及一个析构函数。

使用<utility>

如果我们在翻译单元中包含其他 STL 头文件，那么<utility>就很可能一同包含入该翻译单元中。只有在程序中需要使用该头文件中定义的 STL 组件并且不指望通过其他 STL 头文件将这些组件的定义带入翻译单元时，我们才有必要在程序中显式地包含它。但是在程序中需要使用该头文件中所定义的实体的可能性同样也很大。如果在翻译过程中我们得到了这样的诊断信息：pair 未定义，请记住：最简单的解决方案就是将这个头文件包含到程序中。

pair
make_pair

简单地来讲，对于返回一对相关值的函数调用来说，模板类 pair 是一个相当不错的工具。如果这两个类型都不是常量类型，我们就可以通过调用 make_pair(x, y)，来实时地产生一个 pair 对象用以存储 x 和 y 的值。否则，我们只能显式调用构造函数：pair<T, U>(x, y)。

如我们在前面所提及的一样，模板类 pair 的定义中包含一个成员模板的构造函数：

```
template<class U1, class U2>
pair(const pair<U1, U2>& x)
: first(x.first), second(x.second) {}
```

这个构造函数解决了使用 make_pair 时存在的一些问题。只要两个 pair 之间相应的成员对象的类型可以赋值的话，我们就可以将 make_pair(x, y) 所返回的结果赋给类 pair<T, U>的一个对象。这个函数并没有要求这两个 pair 的类型完全一致。不过虽然它很方便，但是并不是每一个现有的编译器都支持这样的构造函数。为了得到最大的可移植性，我们最好还是使用那种看起来不是那么优美的构造函数来得到同样的结果：pair<T, U>(x, y)。

operator!=
operator>
operator<=
operator>=

如果你在自己定义的类 T 的两个对象间定义了相等性比较 (operator==(const T&))，那么请在代码中添加我们在本章早些时候所描述过的 using 指令，这样就可以很自然地获得对它们之间不等性判断的定义 (operator!=(const T&))。如果想为类 T 定义一个完全排序关系的话，我们只需要定义用于判断“小于”的比较操作 (operator<(const T&))。

然而，在实际生活中，这四个模板操作符的作用并不明显。重载操作符的检索规则已经和名字空间的引入完全结合起来了。并不是所有的

编译器都能对此做出完全正确的反应，有时可能会得不到你所希望的结果。我们建议在代码中最好不要依赖名字空间 `std::rel_ops` 中的那些定义。最好在代码所使用的名字空间中将它们替换成你自己的定义。

实现 `<utility>`

`utility`

程序清单 2-1 列出了文件 `utility`，它实现了头文件 `<utility>`。我们已经在本章的前面部分讨论了其中存在着的一些特性。除此之外，其他的也就没什么了。

```
程序清单 2-1: // utility standard header
utility
#ifndef UTILITY_
#define UTILITY_
#include <iostream>
namespace std {
    // TEMPLATE STRUCT pair
    template<class T1, class T2> struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        pair()
            : first(T1()), second(T2()) {}
        pair(const T1& V1, const T2& V2)
            : first(V1), second(V2) {}
        template<class U1, class U2>
        pair(const pair<U1, U2>& X)
            : first(X.first), second(X.second) {}
        T1 first;
        T2 second;
    };

    // pair TEMPLATE OPERATORS
    template<class T1, class T2> inline
    bool operator==(const pair<T1, T2>& X,
                      const pair<T1, T2>& Y)
    {return (X.first == Y.first && X.second == Y.second); }
    template<class T1, class T2> inline
    bool operator!=(const pair<T1, T2>& X,
                      const pair<T1, T2>& Y)
    {return !(X == Y); }
    template<class T1, class T2> inline
    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); }

template<class T1, class T2> inline
    bool operator>(const pair<T1, T2>& X,
                      const pair<T1, T2>& Y)
    {return (Y < X); }

template<class T1, class T2> inline
    bool operator<=(const pair<T1, T2>& X,
                      const pair<T1, T2>& Y)
    {return (!(Y < X)); }

template<class T1, class T2> inline
    bool operator>=(const pair<T1, T2>& X,
                      const pair<T1, T2>& Y)
    {return (!(X < Y)); }

template<class T1, class T2> inline
    pair<T1, T2> make_pair(const T1& X, const T2& Y)
    {return (pair<T1, T2>(X, Y)); }

// TEMPLATE OPERATORS

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)); }
}

/* namespace std */
#endif /* UTILITY_ */

```

测试 <utility>

tutility.c 程序清单 2-2 列出了文件 tutility.c。由于它所测试的头文件本身很小且没有什么好测试的，它也就比较简单。它主要检测那些明显存在着的定义。如果一切正常的话，它将打印出：

```
SUCCESS testing <utility>
```

然后就正常退出。

```
程序清单 2-2: // test <utility>
utility
#include <assert.h>
#include <iostream>
#include <utility>
using namespace std;

typedef pair<int, char> Pair_ic;
Pair_ic p0;

class Int {
public:
    Int(int v)
        : val(v) {}
    bool operator==(Int x) const
        {return (val == x.val); }
    bool operator<(Int x) const
        {return (val < x.val); }
private:
    int val;
};

// TEST <utility>
int main()
{Pair_ic p1 = p0, p2(3, 'a');

// TEST pair
assert(p1.first == 0);
assert(p1.second == 0);
assert(p2.first == 3);
assert(p2.second == 'a');
assert(p2 == make_pair((Pair_ic::first_type)3,
    (Pair_ic::second_type)'a'));
assert(p2 < make_pair((Pair_ic::first_type)4,
    (Pair_ic::second_type)'a'));
assert(p2 < make_pair((Pair_ic::first_type)3,
    (Pair_ic::second_type)'b'));
assert(p1 != p2);
assert(p2 > p1);
assert(p2 <= p2);
assert(p2 >= p2);

// TEST rel_ops
using namespace std::rel_ops;
Int a(2), b(3);
```

```
assert(a == a);
assert(a < b);
assert(a != b);
assert(b > a);
assert(a <= b);
assert(b >= a);
cout << "SUCCESS testing <utility>" << endl;
return (0); }
```

习题

习题2-1 写出模板类trio<class T, class U, class V>。要求：以模板类pair为原型，并且能够存储任意类型的三个对象。

习题2-2 在上题中，如果要求只使用模板类pair，模板类trio又应该怎么写才能达到同样的效果？

习题2-3 当T和U中有一个是const类型时，我们可能需要经常构造一个pair<T, U>对象。下面的定义是否可以达到这样的效果？如果不可以，请说出你的理由。

```
template<class T, class U> inline
pair<T, U> make_pair(T& x, U& y)
{return (pair<T, U>(x, y)); }
```

习题2-4 [较难] 请描述出一种可能的编码情况，使得表达式 $x.\text{first} < y.\text{first} \parallel !(y.\text{first} < x.\text{first}) \&\& x.\text{second} < y.\text{second}$ 并不等同于 $x.\text{first} < y.\text{first} \parallel x.\text{first} == y.\text{first} \&\& x.\text{second} < y.\text{second}$ 。

习题2-5 [特难] 怎样使用遍布于STL中的模板函数operator<= (const T& x, const T& y)，才能保证不会影响到定义在库外面的类？

第3章 <iterator>

背景知识

正如其名字所暗示的那样，头文件<iterator>中提供了迭代器使用的许多方法。在这个头文件中没有很复杂的东西，但却有很多需要详细解释的细节。除此之外，迭代器还被划分为五种不同的风格（或种类），这其中还不包括对象指针（它们可以当作不同种类的迭代器来使用）。我们将会发现，在本章中所提供的大部分内容都将可能拥有将近半打的不同版本。

由于这个头文件实在是太大了，所以我们决定把它分成不同的片段来展示给大家。不管怎样，大部分的片段所满足的需求都不尽相同。

迭代器属性

首先需要解决的问题是，对所有种类的迭代器施加某种规则。我们在写算法时，常常需要知道给定的迭代器 X 的某些属性。被问的最多的问题有：

- X 到底属于哪种迭代器（是输出的，输入的，前向的，还是其他）？
- *X 指定的元素类型（通常写做 T）到底是什么？
- N 到底应该为哪种差距类型（通常写做 Dist），才可以在“重复地递增 X”时用来计算所有元素的个数？

记住，整个 STL 几乎都是由模板组成的。模板所能够知道的就是被用来特化它的那些类型（再加上一些由此派生出来的类型）。如果想传递更多的信息，摆在你面前的有两种选择：一是在模板中新增模板参数来夹带所需的信息；另外一种就是让迭代器本身就包含有这些额外信息。

iterator

很明显，如果能够实现的话，第二种方法更为优雅。事实证明，大部分情况下都可以做到这一点。简单点的做法就是：让你所定义的所有迭代器类都派生于同一个基类，然后在这个基类中定义所有必要的属性。这个基类的一个原型就是模板类 iterator：

```
template<class C, class T, class Dist = ptrdiff_t
         class Pt = T *, class Rt = T&>
struct iterator {
    typedef C iterator_category;
    typedef T value_type;
    typedef Dist difference_type;
```

```

    typedef Pt pointer;
    typedef Rt reference;
}|.

```

这个模板类中惟一有点特别的东西就是它定义的成员类型的名字。在另一个使用其他方式定义这些成员类型的迭代器类中，我们就不需要它们了。在本章接下来的部分中，我们将详细讨论这五个成员类型的使用。

迭代器标签

第一个模板参数定义了迭代器的种类，它必须是下面已经定义好的几种迭代器标签 (iterator tag) 之一：

```

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 {};

```

上面列出的这些简单的空结构体只是用来表达惟一的类型，没有需要构造（或析构）的值。这个继承层次和第 1 章的“迭代器的种类”一节展示的具有只读属性的迭代器的层次完全一样。当我们在不同的迭代器种类之间重载函数时，这种方法十分方便。我们将在本章的“实现<iterator>”一节中详细讨论这个问题。

例如，如果在这个基本结构上构造一个前向迭代器：

```

class My_it
    : public iterator<forward_iterator_tag, char, long>
{....}

```

那么，算法就可以从类型 `My_it::iterator_category` 中检测到迭代器的种类信息；从 `My_it::value_type` 中得到元素的类型信息；从 `My_it::distance_type` 中得到差距类型信息。像 `My_it` 这样特定形式的迭代器可以用来遍历一个元素类型为 `char` 的二进制文件。由于在这个特殊的例子中使用了类型 `long` 作为 `My_it` 的差距类型，因此在 `My_it` 迭代器上进行的运算，与以前 C 库函数 `fseek` 和 `ftell` 遵循的规则相同（这两个函数也是以 `long` 来作为文件的偏移量）。

附带说一声，按照一般的惯例，输出迭代器都是由基类 `iterator<output_iterator, void, void, void, void>` 所派生出来的。它并不需要向外提供参数类型 `T` 或者 `Dist`。对于一个输出迭代器来说，我们惟一能

做的就是用它存储一个元素，而且这个值不能再读回来。我们也不能够在这样的迭代器上面进行任何运算。造成所有这一切的原因就是迭代器种类的类型。

迭代器的种类

现在，通过一个小小的技巧，我们就可以来回答上面所列出的三个问题中的第一个了。当一个算法的实现方法不止一种时，我们就有必要考虑迭代器的种类问题。下面是一个来自 STL 头文件<algorithm>的简单例子。模板函数 reverse 用来反转一个序列（即第一个元素和最后一个交换，第二个和倒数第二个交换，以此类推）。通过使用模板函数 iter_swap 来进行实际的交换动作，可以使用两个双向迭代器来完成这项工作：

```
template<class BidIt> inline
void reverse(BidIt first, BidIt last)
{for (; first != last && first != --last; ++first)
    iter_swap(first, last); }
```

但是如果我们知道这两个迭代器实际上是随机存取迭代器的话，我们就可以把上面的代码改进得更快一些：

```
template<class RanIt> inline
void reverse(RanIt first, RanIt last)
{for (; first < last; ++first)
    iter_swap(first, --last); }
```

如果所有的迭代器都基于上面给出的那些类的话，我们就可以采取一个小小的技巧，使得自己在这个二选一的选择中更加聪明一点。我们所做的只是通过使用迭代器的种类信息，在被重载的模板函数 Reverse 的定义中进行选择：

```
template<class BidIt> inline
void reverse(BidIt first, BidIt last)
{Reverse(first, last,
        typename BidIt::iterator_category()); }

template<class BidIt> inline
void Reverse(BidIt first, BidIt last,
             bidirectional_iterator_tag)
{for (; first != last && first != --last; ++first)
    iter_swap(first, last); }

template<class RanIt> inline
void Reverse(RanIt first, RanIt last,
             random_access_iterator_tag)
{for (; first < last; ++first)
    iter_swap(first, --last); }
```

现在，对 reverse 的特化已经被解释成为对于某个 Reverse 的调用了。决定调用哪个 Reverse 完全依赖于迭代器的成员类型 iterator_category 的定义，它提供模板参数的类型。在上面给出的函数调用中，构造函数 BidIt::iterator_category() 返回了一个迭代器标签类型的对象，这个对象并没有被其他语句使用，它存在的唯一理由就是作为一个唯一的类型来参与重载函数的解析。

最终的结果就是：模板特化的机制在此被当作翻译时的 switch 语句使用。稍微智能的翻译器甚至能够在优化过程中将额外的函数调用及哑参数的设置给优化掉。这看起来很巧妙了，不是吗？

唔，它还不够巧妙。诚然，STL 以自己能够与其他代码实现无缝连接而自豪。尤其是，它还允许我们在任意可以使用迭代器的地方使用对象指针来替代迭代器。而上面给出的方案却不适合单纯由指针所构成的迭代器。也就是说，不存在一种简单的方法来扩展 Reverse 的定义，使之能够处理任意类型的对象指针。我们需要进一步地改进已有的方法。

实际上，我们将描述解决这个问题的两种不同方法。第一种比较理想化，它目前也被 C++ 标准所接纳了。不幸的是，在本书的书写过程中，这种方法所要求的语言特性还没有得到广泛推广。第二种比较实际，在早期的日子里，它也被采用以使得 STL 能够正常工作。

iterator_traits 正式的解决方案是引入称为 iterator_traits 的又一个模板类。它仅有的模板参数就是迭代器的类型，而它的属性也正是我们所希望获得的。

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

从表面上看，它好像没有增加什么。这个新类所干的不过是重复了其“可能是基于模板类 iterator 的”模板参数所定义过的类型定义而已。但是这个新的模板类却提供了一个很好的机会，引入部分特化来处理所有的指针类型。

```
template<class T>
struct iterator_traits<T *> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t distance_type;
    typedef T *pointer;
```

```
typedef T& reference;
};
```

(库中也包含了对于常量指针的稍有不同的部分特化。) 这个定义也就注定了所有的指针都是随机存取迭代器，并且其差距类型为 ptrdiff_t。你可能会想起：在 C 标准中，ptrdiff_t 被确定为一个整数类型，我们用它来表示指向同一数组中的两个对象指针之间的差距，在标准 C 的头文件<stddef.h>中，使用 typedef 定义了它。

现在，我们可以将模板函数 reverse 重写如下：

```
template<class BidIt> inline
void reverse(BidIt first, BidIt last)
{Reverse(first, last, typename iterator_traits<BidIt>::
iterator_category()); }
```

如果 BidIt 是一个基于模板类 iterator 的类，那么模板类 iterator_traits 就被特化成一般形式。成员类型 iterator_category 也就被定义为 BidIt::iterator_category。但如果 BidIt 是一个指针类型的话，这时部分特化就起作用了。成员类型 iterator_category 也就代表了 random_access_iterator_tag。

Iter_cat

然而，现阶段模板的部分特化并没有普遍获得支持。我们还需要一种可靠的方案来完成这项工作。幸运的是，函数重载正好能提供我们所需的“魔法”。我们以两种方式定义了模板函数 Iter_cat：

```
template<class C, class T, class Dist> inline
C Iter_cat(const iterator<C, T, Dist>&)
{C x;
return (x); }

template<class T> inline
random_access_iterator_tag Iter_cat(const T *)
{random_access_iterator_tag x;
return (x); }
```

附带说一句，在 STL 的最初实现中，这个模板函数被命名为 iterator_category。这个名字没有被 C++ 标准保留下来，所以我们使用了另一个为实现者保留的名字（至少我们还保留了以下划线开头的那些名字）。同样，在理论上，这两个函数都可以简单地返回一个构造函数调用的结果（如：C()）。在此处所使用的那种稍显啰嗦的形式避免了在现有的编译器中所存在的一些问题。和其他大部分可行的 C++ 代码一样，本书所提供的代码也偶尔因为实际中的需要而显得有点奇怪。

现在，我们可以将模板函数 reverse 重写如下：

```
template<class BidIt> inline
```

```
void reverse(BidIt first, BidIt last)
{Reverse(first, last, Iter_cat(first)); }
```

如果 BidIt 是一个基于模板类 iterator 的类，那么模板函数 Iter_cat 的第一种形式就会被调用。这个函数返回一个对象，其类型与 iterator 的第一个模板参数类型（即 BidIt::iterator_category）相同。但如果 BidIt 是一个指针类型的话，这时被调用的就是模板函数的第二种形式了。此时函数明确地返回一个类型为 random_access_iterator_tag 的对象。

本书中的代码使用了模板函数 Iter_cat，而不是模板类 iterator_traits。这使得它们可以很好地适应过渡时期的编译器。

val_type

现在，轮到我们来回答前面剩下的那两个问题了。如果我们想知道一个类型为 It 的迭代器所关联的元素的类型 T，原则上可以写出它的类型名称：iterator_traits<It>::value_type。但当 It 是一个对象指针时，这仍然依赖于部分特化。为了避免出现这个问题，我们在书中的代码里面使用了另一个模板函数 Val_type。如果编译器支持部分特化，我们就可以把这个模板函数简单地写作：

```
template<class It> inline
typename iterator_traits<It>::value_type *Val_type(It)
{return (0); }
```

如果编译器不支持部分特化，我们也可以通过一对重载函数来达到同样的效果：

```
template<class C, class T, class D,
class Pt, class Rt> inline
T *Val_type(iterator, T, D, Pt, Rt)
{return (0); }

template<class T> inline
T *Val_type(const T *)
{return (0); }
```

（原来的名字 value_type 现在已经不再被 C++ 标准所允许了，商业化的实现一般都用一个其他的名字来代替它，如：_Val_type。）

这个函数的任何版本都没有返回一个我们所关注的类型为 T 的对象。如果那样的话将会在该类型描述的对象很大的情况下出现各种问题，或是在构造和析构中出现许多“有趣的”副作用，又或是仅仅为了操作和复制而不得不等上很长时间。为了避免这些问题的出现，我们使这些函数返回的都是一个指向所期望类型的指针。而这对于模板实现其“魔法”来说，已经足够了。

例如，在此我们给出在前面的例子中被 reverse 调用的模板函数

`iter_swap` 的实现：

```
template<class FwdIt1, class FwdIt2> inline
void iter_swap(FwdIt1 x, FwdIt2 y)
{Iter_swap(x, y, Val_type(x)); }

template<class FwdIt1, class FwdIt2, class T> inline
void iter_swap(FwdIt1 x, FwdIt2 y, T *)
{T Tmp = *x;
*x = *y, *y = Tmp; }
```

为了交换两个值，需要使用一个临时对象来保存其中的一个值。于是 `iter_swap` 调用了一个模板函数 `Iter_swap`，并增加了一个额外的参数。`Val_type` 的返回类型 `T*` 也正好匹配 `Iter_swap` 的参数类型 `T*`。也就是说，我们已经从作为基类的迭代器中（或是从作为一个迭代器使用的对象指针中）得到了元素的类型 `T`，然后就可以把它传递到任意需要使用它的函数中去。

`Dist_type`

同样，如果我们想要确定计量两个类型为 `It` 的迭代器之间差距的类型时，可以简单而直接地写出类型的名称：`iterator_traits<It>::difference_type`。实际上，我们在书中的代码里使用了模板函数 `Dist_type`。使用了部分特化的版本看起来如下所示：

```
template<class It> inline
typename iterator_traits::difference_type
*Dist_type(It)
{return (0); }
```

而没有使用部分特化的替代版本看起来如下所示：

```
template<class C, class T, class D,
class Pt, class Rt> inline
D *Dist_type(iterator<C, T, D, Pt, Rt>)
{return (0); }
template<class T> inline
ptrdiff_t *Dist_type(const T *)
{return (0); }
```

（原来的名字是 `distance_type`。）

`advance` `distance`

有两种操作经常被各种迭代器使用，它们并不总是表达为操作符的形式：

- `X += N` 将一个 `Dist` 对象 `N` 加到迭代器 `X` 上。只有当 `X` 是双向迭代器或随机存取迭代器时，`N` 才可以为负。
- `N = X2 - X1` 把两个迭代器 `X2` 和 `X1` 之间的差距赋给 `N`。只有当

迭代器的种类是随机存取时，N 才可能为负。

当然，对于任意的输出迭代器来说，上面的操作毫无意义。对于输入迭代器来说，也几乎没有意义，因为上面的任意一种操作都将使得输入迭代器无效。STL 包含它们更多地是为了完整性而不是通用性。

诚然，使用这样的操作符表达形式很方便，但在最初的 STL 中却很少使用它们。诸如 $X += N$ 及 $X2 - X1$ 的形式仅为随机存取迭代器所需要。它们不可能被那些功能稍差的迭代器种类（即输入、输出、前向、双向迭代器）完全支持。对于迭代器所支持的简单操作，其时间复杂度应该为分摊付出的常量时间（amortized constant time），例如，取得两个迭代器之间差距的时间不应该随其差距的大小而变化。我们甚至被误导去定义具有线性时间复杂度的操作。为了避免这些，STL 中的算法使用了如下两个模板函数：

```
template<class InIt, class Dist> inline
void advance(InIt& p, Dist n);

template<class InIt> inline
typename iterator_traits<InIt>::distance_type
distance(InIt first, InIt last);
```

第一个模板函数实质上计算 $p += n$ ；第二个返回 $n = last - first$ ，但这一切都是建立在迭代器的种类能够使其结果有意义的前提下。如果这些函数中的一个对于给定的迭代器种类所花费的时间不是一个常量，那么那些额外的时间复杂度就将会被纳入考虑的范围内，用来检测那些调用该函数的算法的时间复杂度。

当我们成功地引入了 `iterator_traits` 时，模板类 `distance` 同样也能够说明上述情况。模板函数 `Dist_type` 不能根据上面给出的 `distance` 定义的需求，及时地把它的回答传递给外界，以定义模板函数的返回值。所以，在这个实现中，我们将 `distance` 定义为返回类型 `ptrdiff_t`。但为了求稳，我们按照一贯的方式，在应该使用它的地方使用了模板函数 `Distance`，其声明如下：

```
template<class InIt, class D> inline
void Distance(InIt first, InIt last, D n0);
```

它计算 $n0 += last - first$ ，并返回一个对象，其类型就是我们所希望的差距类型 `D`。（`Distance` 是早先版本中模板函数 `distance` 重命名后的名字。）

也就是说，在这个实现，我们可以避免要求翻译器必须支持模板的部分特化。只要模板类能够明确地利用这个语言特性就可以了。

专用迭代器 头文件<iterator>中剩余的其他部分为一些专门的迭代器定义了一个分类：

- 反转型迭代器 (reverse iterator)
- 插入型迭代器 (insertion iterator)
- 流迭代器 (stream iterator)
- 流缓冲迭代器 (stream buffer iterator)

反转型迭代器 如同其名字所暗示的那样，在使用反转型迭代器向前遍历时，它实际上是在往回遍历一个序列。同样，如果用它向后遍历，它实际上是在向前遍历序列。这是一个很方便的工具，尤其在需要操作一个逆序序列时更加有用——这时不需要真正地去“反转”这个序列，然后在完成操作后再把它转回去，只需完成反向的操作就可以了。

reverse_iterator 模板类 reverse_iterator 是一个随机存取迭代器，它被声明为：

```
template<class RanIt>
class reverse_iterator;
```

在此，RanIt 就是我们想要使之“回退”的那个基本的随机存取迭代器的类型。我们可以这样定义：

```
typedef reverse_iterator<char *> Revptr;
```

它定义了一个可以在由 char 组成的数组中“回退”的迭代器类型 Revptr。

这个模板类实际上只储存了一个对象，也就是由类 RanIt 所得到的那个基本的迭代器对象。任何人都可以通过调用它的成员函数 base() 来获得这个对象的值。从 reverse_iterator 所派生出来的类也可以通过使用保护型的对象 current 来存取它。

附带说一句，模板类 reverse_iterator 对于我们选择开发的任何随机存取迭代器来说都是一个非常好的原型。它提供了所有必需的成员函数。在头文件<iterator>中还同样提供了所有我们可能需要为它定义的模板操作符。我们将在容器模板类 vector（参见第 10 章）和 deque（参见第 12 章）中利用这种特殊的迭代器。

反转型双向迭代器 从前，在 STL 中还存在另外一个模板类 reverse_bidirectional_iterator。它的意图很明显，就是把一个双向迭代器变成一个反转型迭代器。然而，在后来的标准化进程中，这个模板类消失了。标准化委员会认定模板类 reverse_iterator 同样可以完成这项工作。

为了使得上面的替代工作得以顺利地完成，我们需要对语言本身有所改进。在随机存取迭代器中，我们定义了一些不可能被双向迭代器支持的成员函数，如 operator+=，它用来将一个整型偏移量加到迭代器上。

许多旧版本的编译器会检测所有的模板成员函数（甚至在该函数从未被调用的情况下也是如此），以判断表达式是否有效。标准化委员会最终决定，只有那些被调用的成员函数才应该被检测。这个改动解决了在定义 `operator->` 的迭代器模板类中存在着的一个相似的问题，因为只有在迭代器指向的类型是类、结构体或联合时，`operator->` 才有意义。

然而，编译器还需要一些时间来赶上这些变化。在对双向迭代器使用 `reverse_iterator` 时，有的编译器仍然会给出出错消息。为此，我们在此处提供的实现仍然保留了模板类 `reverse_bidirectional_iterator`，不过名字换成了 `Revbidit`。你将不会再看到这个模板类以这种表达方式出现，但有些编译器却会在容器模板类 `list`（参见第 11 章）、`set` 和 `multiset`（参见第 13 章）以及 `map` 和 `multimap`（参见第 14 章）中利用它。

在此我们保留下模板类 `Revbidit` 还有另一个原因。对于我们所选择开发的双向迭代器来说，它是一个非常好的原型。它提供了所有必需的成员函数，其中包括所有双向迭代器所要求支持的模板操作符。

我们经常需要通过“从头到尾来一遍”来产生一个元素序列。处理这些产生出来的元素的最完美的代理就是输出迭代器。在所有允许往序列中写入的迭代器类型中，输出迭代器具有最少的语义需求。这些简单需求让我们得以以相当大的自由度来写输出迭代器。

举例来说，我们希望往类型为 `Cont` 的容器对象 `C` 中插入我们所产生的序列。在 STL 中所定义的容器模板类都提供了许多统一的可供我们利用的属性。尤其是：

- 每个容器都定义它的元素类型为 `Cont::value_type`，把用于被控序列的迭代器类型定义为 `Cont::iterator`。
- 所有支持在常数时间内向被控序列末端插入新元素的容器都定义了成员函数：

```
C.push_back(Cont::value_type&)
```

- 所有支持在常数时间内被控序列前端插入新元素的容器都定义了成员函数：

```
C.push_front(Cont::value_type&)
```

- 所有支持在常数时间内向被控序列中由迭代器指定的任意位置处插入新元素的容器都定义了成员函数：

```
C.insert(Cont::value_type&, Cont::iterator)
```

最后那个函数返回一个指向被插入元素的迭代器。

当然，我们也可以遵照这些规则来定义自己的容器。STL 也鼓励我们按照这样的模型去做，以便我们添加的部分可以很容易地和 STL 的其他部分融合在一起。

插入型迭代器

头文件<iterator>中定义了几个使用了这些成员函数的模板类：

- back_insert_iterator** • `back_insert_iterator<Cont>`是一个输出迭代器，它用来将产生的元素添加到类型为 `Cont` 的容器对象的末端。
- front_insert_iterator** • `front_insert_iterator<Cont>`是一个输出迭代器，它用来将产生的元素添加到容器的前端。（是的，产生出来的元素以逆序的方式结束于被控序列前端。）
- insert_iterator** • `insert_iterator<Cont, Iter>`是一个输出迭代器，它用来将产生的元素插入到一个由迭代器指定的元素前面。

每个这样的模板类中都存储了一个指向容器对象的引用，其名字为保护型的 `container`。最后的那个模板类（`insert_iterator`）还在对象中存储了插入点，它拥有保护型的名字 `iter`。

头文件<iterator>也定义了三个相关的模板函数：

- back_inserter** • `back_inserter (Cont&)` 对于容器参数返回一个 `back_insert_iterator`。
- front_inserter** • `front_inserter (Cont&)` 对于容器参数返回一个 `front_insert_iterator`。
- inserter** • `inserter (Cont&, Iter)` 对于容器参数和迭代器参数返回一个 `insert_iterator`。

这些函数提供了一种方便的方法，用以实时产生出一个插入型迭代器。

流迭代器

如果我们可以“聪明地”使用输出迭代器，将产生的序列传给一个容器的话，我们就同样可以通过某种“聪明的”形式，将序列插入到一个输出流（output stream）中。同样，我们也可以通过“聪明地”使用输入迭代器，从一个输入流（input stream）中获得一个元素序列。实际上，这就是我们将要讨论的东西。

假设我们有类型为 `T` 的元素，且定义了从流中提取值的操作：

```
istream& operator>>(istream&, T&);
```

我们所需做的只是从输入流（如 `cin`）中读取所需的字符，将其转化为一个类型为 `T` 的值，然后存储到对象 `X` 中去，这么写：

```
cin >> X;
```

istream_iterator

模板类 `istream_iterator<T>`为我们做到了这一点。从表面上看，它是通过使用一个输入迭代器来从序列中得到一个元素。（该模板类还有其他的参数，它们用来处理其他的更为花哨的输入流，但默认值已足以应付广泛使用的 `istream` 的要求了。）如果我们这样写：

```
typedef istream_iterator<int, ptrdiff_t> Int_init;
Int_init int_in(cin);
```

那么表达式`*int_in++`就会从`cin`中提取出一个`int`，并把它作为该表达式的值传递。

当然，我们首先应该判断在输入流中这个值是否有效。适当的方法是将`int_it`与一个标明文件结束的`end-of-sequence`值进行相等性比较。

(见第1章的“`end-of-sequence`值”一节。)默认构造函数总是产生具有`end-of-sequence`值的对象。也就是说，我们可以这么写：

```
Int_init first(cin), last;
```

然后，就可以从流中不断地提取元素，直到遇到文件结束为止。我们可以这么写(以固定格式的循环方式)：

```
for (p = first; p != last; ++p)
    <process>(*p);
```

或它的变体：

```
while (first != last)
    <process>(*first++);
```

ostream_iterator

头文件<iterator>中也定义了模板类`ostream_iterator`。它定义了一个用来向输出流中插入元素的输出迭代器。假设我们有类型为`T`的元素，且定义了向流中插入值的操作：

```
ostream& operator<<(ostream&, T);
```

我们所需做的只是将一个类型为`T`的值转化为一个字符序列，并把这个序列写入到输出流(如`cout`)中，这么写：

```
cout << X;
```

模板类`ostream_iterator<T>`为我们做到了这一点。从表面上看，它是通过使用一个输出迭代器来向序列中存储一个元素。(再次重申一遍，该模板类还有其他的参数，它们用来处理其他更为花哨的输出流，但默认值已足以应付广泛使用的`ostream`的要求了。)如果我们这样写：

```
typedef ostream_iterator<int> Int_outit;
Int_outit int_out(cout);
```

那么表达式`*int_out++ = X`就会将类型为`int`的值`X`插入到`cout`中。更好的一种声明方式为：

```
Int_outit int_out(cout, " ");
```

它在每个被插入的值后面都添加了一个空格。这样，插入值所产生的字段就不会连成一片，不好辨别了。

流缓冲迭代器

STL 中一个后来添加的部分是一对迭代器，它们用来直接向（从）一个流缓冲区（stream buffer）中插入提取类型为 E（通常为 char）的元素。模板类 `basic_streambuf<E, T>` 定义了一个流缓冲区对象，它被用在实际的流（如文件）与程序存储器之间，起传输媒介的作用。它是原有的 IOStreams 类 `streambuf` 的模板化版本，类似于标准 C 库中的类型 FILE。在此，T 表示一个 traits 类，通常用来提供元素类型 E 的各种信息。

由于 `iostreams` 已经被标准化了，因此我们不去研究它的纷繁难懂之处，那将会是一个非常大的主题。在此，我们只需知道模板类 `istreambuf_iterator<E, T>` 在新增的方法里面起着重要的作用就行了。从输入流中所读取的每个字符其实都是通过这种类型的输入迭代器而间接获得的。与之相似的是，写到输出流中的每个字符，其实也是通过类型为 `ostreambuf_iterator<E, T>` 的一个输出迭代器而间接地写到输出流上的。

在头文件`<iterator>`中所定义的方法被贯穿地使用于 STL 中。我们应该让自己足够地熟悉这个头文件，这样才可能很舒服地使用它。如果没有大量使用在本章开始处所描述的迭代器的属性，我们几乎不可能写出一个很好的算法来。在使用容器时，我们通常会惊讶于使用插入型迭代器是多么方便。在真正处理有关 `iostreams` 的工作时，我们几乎不可避免地至少要和流缓冲迭代器打交道。

让我们最后一次强调，这个头文件有着非常重要的应用。在所有的五种迭代器中，只有前向迭代器在头文件中没有任何原型的定义。其他的几种都给我们提供了极好的模型。我们只需大胆使用它们就可以了。

功能描述

```
namespace std {
    struct input_iterator_tag;
    struct output_iterator_tag;
    struct forward_iterator_tag;
    struct bidirectional_iterator_tag;
    struct random_access_iterator_tag;

    // TEMPLATE CLASSES
    template<class C, class T, class Dist,
              class Pt, class Rt>
        struct iterator;
    template<class It>
        struct iterator_traits;
    template<class T>
        struct iterator_traits<T *>;
}
```

```
template<class RanIt>
    class reverse_iterator;
template<class Cont>
    class back_insert_iterator;
template<class Cont>
    class front_insert_iterator;
template<class Cont>
    class insert_iterator;
template<class U, class E, class T, class Dist>
    class istream_iterator;
template<class U, class E, class T>
    class ostream_iterator;
template<class E, class T>
    class istreambuf_iterator;
template<class E, class T>
    class ostreambuf_iterator;

        // TEMPLATE FUNCTIONS
template<class RanIt>
    bool operator==(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator==(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt>
    bool operator!=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator!=(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt>
    bool operator<(
        const reverse_iterator<RanIt>& lhs,
```

```

        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator<=((
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    Dist operator-(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    reverse_iterator<RanIt> operator+(
        Dist n,
        const reverse_iterator<RanIt>& rhs);
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
template<class InIt>
    iterator_traits<InIt>::difference_type
        distance(InIt first, InIt last);
};

```

包含 STL 标准头文件<iterator>就可以得到一系列类、模板类及模板函数的定义，这都对迭代器的声明及操作有着辅助作用。

口 advance

```

template<class InIt, class Dist>
    void advance(InIt& it, Dist n);

```

该模板函数通过将 it 递增 n 次后，最终有效地向前移动了 it。如果 InIt 是一个随机存取迭代器的话，函数就只计算表达式 $it += n$ 。否则，函数通过计算 $++it$ 来完成每次递增。当 InIt 是一个输入或前向迭代器时，n 不允许为负。

口 back_insert_iterator

```

template<class Cont>
class back_insert_iterator
    : public iterator<output_iterator_tag,
        void, void, void, void> {

public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    explicit back_insert_iterator(Cont& x);
    back_insert_iterator&
        operator=(typename Cont::const_reference val);
    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);
protected:
    Cont *container;
};

```

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为 `Cont` 的容器中，我们也可以通过它所存储的保护型指针对象 `container` 来存取该容器。这个容器必须定义：

- 成员类型 `const_reference`，即容器所控制的序列中元素的常量引用（constant reference）的类型。
- 成员类型 `reference`，即容器所控制的序列中元素的引用（reference）的类型。
- 成员类型 `value_type`，即容器所控制的序列中的元素的类型。
- 成员函数 `push_back(value_type c)`，它可以将一个值为 `c` 的新元素添加到序列的末端。

口 `back_insert_iterator::back_insert_iterator`
`explicit back_insert_iterator(Cont& x);`

该构造函数将 `container` 初始化为 `&x`。

口 `back_insert_iterator::container_type`
`typedef Cont container_type;`

该类型为模板参数 `Cont` 的同义词。

口 `back_insert_iterator::operator*`
`back_insert_iterator& operator*();`

该成员函数返回 `*this`。

- `back_insert_iterator::operator++`
`back_insert_iterator& operator++();`
`back_insert_iterator operator++(int);`
 这两个成员函数都返回*this。
- `back_insert_iterator::operator=`
`back_insert_iterator&`
`operator=(typename Cont::const_reference val);`
 该成员函数先完成`container.push_back(val)`的动作，然后返回*this。
- `back_insert_iterator::reference`
`typedef typename Cont::reference reference;`
 该类型描述了由关联容器所控制的序列中的元素的引用。
- `back_insert_iterator::value_type`
`typedef typename Cont::value_type value_type;`
 该类型描述了由关联容器所控制的序列中的元素。
- `back_inserter`
`template<class Cont>`
`back_insert_iterator<Cont> back_inserter(Cont& x);`
 该模板函数返回`back_insert_iterator<Cont>(x)`。
- `bidirectional_iterator_tag`
`struct bidirectional_iterator_tag`
`: public forward_iterator_tag {`
`};`
 当 It 描述的对象可以当作双向迭代器来使用时，该类型等价于`iterator<It>::iterator_category`。
- `distance`
`template<class InIt>`
`typename iterator_traits<InIt>::difference_type`
`distance(InIt first, InIt last);`
 该模板函数先将一个计数器 n 设为 0，然后不断地向前移动 first，并在移动过程中对 n 进行递增，直到 first == last。如果 InIt 是一个随机存取迭代器的话，那该函数就只计算表达式 n += last - first。否则，它通过计算++first 完成每次迭代器递增。
- `forward_iterator_tag`
`struct forward_iterator_tag`
`: public input_iterator_tag {};`
 当 It 描述的对象可以当作前向迭代器来使用时，该类型等价于`iterator<It>::iterator_category`。

```

# front_insert_iterator

    template<class Cont>
        class front_insert_iterator
            : public iterator<output_iterator_tag,
                void, void, void, void> {
public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    explicit front_insert_iterator(Cont& x);
    front_insert_iterator&
        operator=(typename Cont::const_reference val);
    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);
protected:
    Cont *container;
};

```

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为 **Cont** 的容器中，可以通过它所存储的保护型指针对象 **container** 来存取该容器。这个容器必须定义：

- 成员类型 **const_reference**，即容器所控制的序列中元素的常量引用的类型。
- 成员类型 **reference**，即容器所控制的序列中元素的引用的类型。
- 成员类型 **value_type**，即容器所控制的序列中的元素的类型。
- 成员函数 **push_front(value_type c)**，它可以将一个值为 **c** 的新元素添加到序列的前端。

```

# front_insert_iterator::container_type
    typedef Cont container_type;
    该类型为模板参数 Cont 的同义词。

# front_insert_iterator::front_insert_iterator
    explicit front_insert_iterator(Cont& x);
    该构造函数将 container 初始化为 &x。

# front_insert_iterator::operator*
    front_insert_iterator& operator*();
    该成员函数返回 *this。

# front_insert_iterator::operator++
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);

```

这两个成员函数都返回*this。

- `front_insert_iterator::operator=`
`front_insert_iterator&`
`operator=(typename Cont::const_reference val);`
 该成员函数先完成 `container.push_front(val)` 的动作，然后返回*this。
- `front_insert_iterator::reference`
`typedef typename Cont::reference reference;`
 该类型描述了由关联容器所控制的序列中的元素的引用。
- `front_insert_iterator::value_type`
`typedef typename Cont::value_type value_type;`
 该类型描述了由关联容器所控制的序列中的元素。
- `front_inserter`
`template<class Cont>`
`front_insert_iterator<Cont> front_inserter(Cont& x);`
 该模板函数返回 `front_insert_iterator<Cont>(x)`。
- `input_iterator_tag`
`struct input_iterator_tag {};`
 当 `It` 描述的对象可以当作输入迭代器来使用时，该类型等价于 `iterator<It>::iterator_category`。
- `insert_iterator`
`template<class Cont>`
`class insert_iterator`
`: public iterator<output_iterator_tag,`
`void, void, void, void> {`
`public:`
 `typedef Cont container_type;`
 `typedef typename Cont::reference reference;`
 `typedef typename Cont::value_type value_type;`
 `insert_iterator(Cont& x,`
 `typename Cont::iterator it);`
 `insert_iterator& operator=(typename Cont::const_reference val);`
 `insert_iterator& operator*();`
 `insert_iterator& operator++();`
 `insert_iterator& operator++(int);`
`protected:`
 `Cont *container;`
 `typename Cont::iterator iter;`
`};`

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为 Cont 的容器中，可以通过它所存储的保护型指针对象 container 来存取该容器。它还存储了类 Cont::iterator 的保护型迭代器对象 iter。这个容器必须定义：

- 成员类型 const_reference，即容器所控制的序列中元素的常量引用的类型。
- 成员类型 iterator，即容器的迭代器的类型。
- 成员类型 reference，即容器所控制的序列中元素的引用的类型。
- 成员类型 value_type，即容器所控制的序列中的元素的类型。
- 成员函数 insert(iterator it, value_type c)，它可以将一个值为 c 的新元素插入到被控序列中，位置紧接着由 it 指定的元素的前面，并返回指向被插入元素的那个迭代器。

口 insert_iterator::container_type

```
typedef Cont container_type;
```

该类型为模板参数 Cont 的同义词。

口 insert_iterator::insert_iterator

```
insert_iterator(Cont& x, typename Cont::iterator it);
```

该构造函数将 container 初始化为 &x，将 iter 初始化为 it。

口 insert_iterator::operator*

```
insert_iterator& operator*();
```

该成员函数返回*this。

口 insert_iterator::operator++

```
insert_iterator& operator++();
```

```
insert_iterator& operator++(int);
```

这两个成员函数都返回*this。

口 insert_iterator::operator=

```
insert_iterator&
```

```
operator=(typename Cont::const_reference val);
```

该成员函数先完成 iter=container.insert(iter, val) 的动作，然后返回

*this。

口 insert_iterator::reference

```
typedef typename Cont::reference reference;
```

该类型描述了由关联容器所控制的序列中的元素的引用。

口 insert_iterator::value_type

```
typedef typename Cont::value_type value_type;
```

该类型描述了由关联容器所控制的序列中的元素。

¶ inserter

```
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
```

该模板函数返回 `insert_iterator<Cont>(x,it)`。

¶ istream_iterator

```
template<class U, class E = char,
         class T = char_traits>
    class Dist = ptrdiff_t>
    class istream_iterator
        : public iterator<input_iterator_tag,
                           U, Dist, U *, U&> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef basic_istream<E, T> istream_type;
    istream_iterator();
    istream_iterator(istream_type& is);
    const U& operator*() const;
    const U *operator->() const;
    istream_iterator<U, E, T, Dist>& operator++();
    istream_iterator<U, E, T, Dist> operator++(int);
};
```

该模板类描述了一个输入迭代器对象。它可以从一个输入流中提取类 U 的对象，可通过它所存储的类型为指向 `basic_istream<E, T>` 的指针的对象存取输入流。在通过使用一个非空指针来构造或递增一个类 `istream_iterator` 的对象后，该对象就会试图从与之相关的输入流中提取并存储一个类型为 U 的对象。如果提取操作失败的话，该对象就会用一个空指针有效地替换掉它以前所存储的指针（也就是说，产生一个 `end-of-sequence` 指示符）。

¶ istream_iterator::char_type

```
typedef E char_type;
```

该类型为模板参数 E 的同义词。

¶ istream_iterator::istream_iterator

```
istream_iterator();
istream_iterator(istream_type& is);
```

第一个构造函数将指向输入流的指针初始化为一个空指针。第二个构造函数将指向输入流的指针初始化为 `&is`，然后试图用它来提取并存储

一个类型为 U 的对象。

- 口 `istream_iterator::istream_type`
`typedef basic_istream<E, T> istream_type;`
该类型为 `basic_istream<E, T>` 的同义词。
- 口 `istream_iterator::operator*`
`const U& operator*() const;`
该操作符返回所存储的类型为 U 的对象。
- 口 `istream_iterator::operator->`
`const U *operator->() const;`
该操作符返回 `&**this`。
- 口 `istream_iterator::operator++`
`istream_iterator<U, E, T, Dist>& operator++();`
`istream_iterator<U, E, T, Dist> operator++(int);`
第一个操作符试图从关联输入流中提取并存储一个类型为 U 的对象。第二个操作符将该对象复制了一份，然后将其递增，最终返回它的那份拷贝。
- 口 `istream_iterator::traits_type`
`typedef T traits_type;`
该类型为模板参数 T 的同义词。
- 口 `istreambuf_iterator`
`template<class E, class T = char_traits<E> >`
`class istreambuf_iterator`
`: public iterator<input_iterator_tag,`
`E, typename T::off_type, E *, E&> {`
`public:`
 `typedef E char_type;`
 `typedef T traits_type;`
 `typedef typename T::int_type int_type;`
 `typedef basic_streambuf<E, T> streambuf_type;`
 `typedef basic_istream<E, T> istream_type;`
 `istreambuf_iterator(streambuf_type *sb = 0) throw();`
 `istreambuf_iterator(istream_type& is) throw();`
 `const E& operator*() const;`
 `const E *operator->() const;`
 `istreambuf_iterator& operator++();`
 `istreambuf_iterator operator++(int);`
 `bool equal(const istreambuf_iterator& rhs) const;`
};

该模板类描述了一个输入迭代器对象。它可以从一个输入流缓冲区中提取类 E 的元素，可通过它所存储的类型为指向 `basic_streambuf<E, T>` 的指针的对象存取流缓冲区。在通过使用一个非空指针来构造或递增一个类 `istreambuf_iterator` 的对象后，该对象就会试图从与之相关的输入流中提取并存储一个类型为 E 的对象。（然而，提取操作也可以延迟到该对象被用于间接取值或拷贝时。）如果提取操作失败的话，该对象就会用一个空指针来替换掉它以前所存储的指针（也就是说，产生一个 `end-of-sequence` 指示符）。

口 `istreambuf_iterator::char_type`

```
typedef E char_type;
```

该类型是模板参数 E 的同义词。

口 `istreambuf_iterator::equal`

```
bool equal(const istreambuf_iterator& rhs) const;
```

只有当存储在该 `istreambuf_iterator` 对象中的输入流缓冲区指针与存储在 `rhs` 中的同时为空或同时为不空时，该成员函数才返回 `true`。

口 `istreambuf_iterator::int_type`

```
typedef typename T::int_type int_type;
```

该类型是 `T::int_type` 的同义词。

口 `istreambuf_iterator::istream_type`

```
typedef basic_istream<E, T> istream_type;
```

该类型是 `basic_istream<E, T>` 的同义词。

口 `istreambuf_iterator::istreambuf_iterator`

```
istreambuf_iterator(streambuf_type *sb = 0) throw();
```

```
istreambuf_iterator(istream_type& is) throw();
```

第一个构造函数将输入流缓冲区指针初始化为 `sb`。第二个构造函数将输入流缓冲区指针初始化为 `is.rdbuf()`，然后试图提取并存储一个类型为 E 的对象。

口 `istreambuf_iterator::operator*`

```
const E& operator*() const;
```

该操作符返回所存储的类型为 E 的对象。

口 `istreambuf_iterator::operator++`

```
istreambuf_iterator& operator++();
```

```
istreambuf_iterator operator++(int);
```

第一个操作符试图从相关输入流中提取并存储一个类型为 E 的对象。第二个操作符则将该对象复制了一份，然后将其递增，最终返回那

份拷贝。

- `istreambuf_iterator::operator->`
`const E *operator->() const;`
 该操作符返回`&**this`。
- `istreambuf_iterator::streambuf_type`
`typedef basic_streambuf<E, T> streambuf_type;`
 该类型为`basic_streambuf<E, T>`的同义词。
- `istreambuf_iterator::traits_type`
`typedef T traits_type;`
 该类型为模板参数`T`的同义词。

- `iterator`

```
template<class C, class T, class Dist = ptrdiff_t
         class Pt = T *, class Rt = T&>
struct iterator {
    typedef C iterator_category;
    typedef T value_type;
    typedef Dist difference_type;
    typedef Pt pointer;
    typedef Rt reference;
};
```

该模板类可以很方便地用来当作我们所定义的迭代器的基类。它定义了成员类型`iterator_category`(模板参数`C`的同义词)、`value_type`(模板参数`T`的同义词)、`difference_type`(模板参数`Dist`的同义词)、`pointer`(模板参数`Pt`的同义词)以及`reference`(模板参数`Rt`的同义词)。

注意，即便当`pointer`指向的对象与`reference`代指的对象都具有常量属性时，`value_type`也不必是一个常量类型。

- `iterator_traits`

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

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

```

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

```

该模板类定义了几个与迭代器类型 It 密切相关的关键性类型。它定义了成员类型 iterator_category (It::iterator_category 的同义词)、value_type (It::value_type 的同义词)、difference_type (It::difference_type 的同义词)、pointer (It::pointer 的同义词) 以及 reference (It::reference 的同义词)。

部分特化的版本中定义了几个与对象指针类型 T* 密切相关的关键性类型。在这个实现中，我们还可以使用一些没有用到部分特化的模板函数：

```

template<class C, class T, class Dist>
    C Iter_cat(const iterator<C, T, Dist>&);
template<class T>
    random_access_iterator_tag Iter_cat(const T *);

template<class C, class T, class Dist>
    T *Val_type(const iterator<C, T, Dist>&);
template<class T>
    T *Val_type(const T *);

template<class C, class T, class Dist>
    Dist *Dist_type(const iterator<C, T, Dist>&);
template<class T>
    ptrdiff_t *Dist_type(const T *);

```

这些函数也能够为我们提供所需的同样类型（只不过使用的方法有点间接）。我们可以在函数调用时将这些函数作为参数。它们存在的惟一目的就是：为我们所调用的函数提供有用的模板类参数。

口 operator!=

```

template<class RanIt>
```

```

    bool operator!=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator!=(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);

```

该模板操作符返回!(lhs == rhs)。

口 operator==

```

template<class RanIt>
    bool operator==(const reverse_iterator<RanIt>& lhs,
                      const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator==(const istream_iterator<U, E, T, Dist>& lhs,
                      const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator==(const istreambuf_iterator<E, T>& lhs,
                      const istreambuf_iterator<E, T>& rhs);

```

只有当 lhs.current == rhs.current 时，第一个模板操作符才会返回 true。只有当 lhs 和 rhs 存储相同的流指针时，第二个模板操作符才会返回 true。第三个模板操作符返回 lhs.equal(rhs)。

口 operator<

```

template<class RanIt>
    bool operator<(const reverse_iterator<RanIt>& lhs,
                     const reverse_iterator<RanIt>& rhs);

```

该模板操作符返回 rhs.current < lhs.current[sic]。

口 operator<=

```

template<class RanIt>
    bool operator<=(const reverse_iterator<RanIt>& lhs,
                     const reverse_iterator<RanIt>& rhs);

```

该模板操作符返回!(rhs < lhs)。

```

# operator>

    template<class RanIt>
        bool operator>(
            const reverse_iterator<RanIt>& lhs,
            const reverse_iterator<RanIt>& rhs);
    该模板操作符返回 rhs < lhs.

# operator>=

    template<class RanIt>
        bool operator>=(
            const reverse_iterator<RanIt>& lhs,
            const reverse_iterator<RanIt>& rhs);
    该模板操作符返回 !(lhs < rhs).

# operator+

    template<class RanIt>
        reverse_iterator<RanIt> operator+(Dist n,
            const reverse_iterator<RanIt>& rhs);
    该模板操作符返回 rhs + n.

# operator-

    template<class RanIt>
        Dist operator-(
            const reverse_iterator<RanIt>& lhs,
            const reverse_iterator<RanIt>& rhs);
    该模板操作符返回 rhs.current - lhs.current[sic]. 

# ostream_iterator

    template<class U, class E = char,
        class T = char_traits<E> >
        class ostream_iterator
            : public iterator<output_iterator_tag,
                void, void, void, void> {

    public:
        typedef U value_type;
        typedef E char_type;
        typedef T traits_type;
        typedef basic_ostream<E, T> ostream_type;
        ostream_iterator(ostream_type& os);
        ostream_iterator(ostream_type& os, const E *delim);
        ostream_iterator<U, E, T>& operator=(const U& val);
        ostream_iterator<U, E, T>& operator*();
        ostream_iterator<U, E, T>& operator++();
        ostream_iterator<U, E, T> operator++(int);
    };

```

该模板类描述了一个输出迭代器对象。它可以将一个类型为 U 的对象插入到一个输出流中去。可通过它所存储的类型为指向 `basic_ostream<E, T>` 的指针的对象存取该输出流。它同时还存储了一个指向分界符字符串的指针，该字符串是一个元素类型为 E 且以空字符为结束的字符串，在每个插入动作后，它都会被添加到流的末端。（注意：该字符串本身并非是由构造函数中复制的。）

- `ostream_iterator::char_type`
`typedef E char_type;`
该类型为模板参数 E 的同义词。
- `ostream_iterator::operator*`
`ostream_iterator<U, E, T>& operator*();`
该操作符返回`*this`。
- `ostream_iterator::operator++`
`ostream_iterator<U, E, T>& operator++();`
`ostream_iterator<U, E, T> operator++(int);`
这两个操作符都返回`*this`。
- `ostream_iterator::operator=`
`ostream_iterator<U, E, T>& operator=(const U& val);`
该操作符先将 `val` 插入到关联输出流中，然后返回`*this`。
- `ostream_iterator::ostream_iterator`
`ostream_iterator(ostream_type& os);`
`ostream_iterator(ostream_type& os, const E *delim);`
第一个构造函数将输出流指针初始化为`&os`，指向分界符字符串的指针初始化为空字符串。第二个构造函数将输出流指针初始化为`&os`，指向分界符字符串的指针初始化为 `delim`。
- `ostream_iterator::ostream_type`
`typedef basic_ostream<E, T> ostream_type;`
该类型为 `basic_ostream<E, T>` 的同义词。
- `ostream_iterator::traits_type`
`typedef T traits_type;`
该类型为模板参数 T 的同义词。
- `ostream_iterator::value_type`
`typedef U value_type;`
该类型为模板参数 U 的同义词。

```

口 ostreambuf_iterator

    template<class E, class T = char_traits<E>>
        class ostreambuf_iterator
            : public iterator<output_iterator_tag,
                void, void, void, void> {
    public:
        typedef E char_type;
        typedef T traits_type;
        typedef basic_streambuf<E, T> streambuf_type;
        typedef basic_ostream<E, T> ostream_type;
        ostreambuf_iterator(streambuf_type *sb) throw();
        ostreambuf_iterator(ostream_type& os) throw();
        ostreambuf_iterator& operator=(E x);
        ostreambuf_iterator& operator*( );
        ostreambuf_iterator& operator++();
        T1 operator++(int);
        bool failed() const throw();
    };

```

该模板类描述了一个输出迭代器对象。它可以将类 E 的元素插入到输出流缓冲区中。可通过它所存储的类型为指向 basic_streambuf<E, T> 的指针的对象存取该输出流缓冲区。

口 ostreambuf_iterator::char_type

typedef E char_type;

该类型是模板参数 E 的同义词。

口 ostreambuf_iterator::failed

bool failed() const throw();

只有当前面对输入流缓冲区进行的插入动作失败时，该成员函数才返回 true。

口 ostreambuf_iterator::operator*

ostreambuf_iterator& operator*();

该操作符返回*this。

口 ostreambuf_iterator::operator++

ostreambuf_iterator& operator++();

T1 operator++(int);

第一个操作符返回*this。第二个操作符返回一个类型为 T1 的对象，其中 T1 可以转化为 ostreambuf_iterator<E, T>。

口 ostreambuf_iterator::operator=

ostreambuf_iterator& operator=(E x);

该操作符先将 x 插入到关联流缓冲区中，然后返回*this。

- `ostreambuf_iterator::ostream_type`
`typedef basic_ostream<E, T> ostream_type;`
该类型是 `basic_ostream<E, T>` 的同义词。
- `ostreambuf_iterator::ostreambuf_iterator`
`ostreambuf_iterator(streambuf_type *sb) throw();`
`ostreambuf_iterator(ostream_type& os) throw();`
第一个构造函数将输入流缓冲区指针初始化为 sb。第二个构造函数将输出流缓冲区指针初始化为 os.rdbuf()。(存储的指针不能为空指针。)
- `ostreambuf_iterator::streambuf_type`
`typedef basic_streambuf<E, T> streambuf_type;`
该类型为 `basic_streambuf<E, T>` 的同义词。
- `ostreambuf_iterator::traits_type`
`typedef T traits_type;`
该类型为模板参数 T 的同义词。
- `output_iterator_tag`
`struct output_iterator_tag {`
`};`
当 It 描述的对象可以当作输出迭代器来使用时，该类型等价于 `iterator<It>::iterator_category`。
- `random_access_iterator_tag`
`struct random_access_iterator_tag`
`: public bidirectional_iterator_tag {`
`};`
当 It 描述的对象可以当作随机存取迭代器来使用时，该类型等价于 `iterator<It>::iterator_category`。
- `reverse_iterator`
`template<class RanIt>`
`class reverse_iterator : public iterator<`
 `typename iterator_traits<RanIt>::iterator_category,`
 `typename iterator_traits<RanIt>::value_type,`
 `typename iterator_traits<RanIt>::difference_type,`
 `typename iterator_traits<RanIt>::pointer,`
 `typename iterator_traits<RanIt>::reference> {`
 `typedef typename iterator_traits<RanIt>::difference_type Dist;`
 `typedef typename iterator_traits<RanIt>::pointer Ptr;`

```

    typedef typename iterator_traits<RanIt>::reference Ref;
public:
    typedef RanIt iterator_type;
    reverse_iterator();
    explicit reverse_iterator(RanIt x);
    template<class U>
        reverse_iterator(const reverse_iterator<U>& x);
    RanIt base() const;
    Ref operator*() const;
    Ptr operator->() const;
    reverse_iterator& operator++();
    reverse_iterator operator++(int);
    reverse_iterator& operator--();
    reverse_iterator operator--(int);
    reverse_iterator& operator+=(Dist n);
    reverse_iterator operator+(Dist n) const;
    reverse_iterator& operator-=(Dist n);
    reverse_iterator operator-(Dist n) const;
    Ref operator[](Dist n) const;
protected:
    RanIt current;
};

```

该模板类描述了一个各方面动作都像（只是顺序相反）随机存取迭代器的对象。它在保护型对象 `current` 中存储了一个类型为 `RanIt` 的随机存取迭代器。对类型为 `reverse_iterator` 的对象 `x` 进行递增其实是在对 `x.current` 进行递减，反之亦然。此外，表达式`*x` 其实代表的是`*(current - 1)`，且其结果的类型为 `Ref`。`Ref` 通常都是类型 `T&`。

也就是说，我们可以使用一个类 `reverse_iterator` 的对象来对一个随机存取迭代器在其中来回移动的序列进行逆向存取。

不少 `STL` 中的容器特别为 `RanIt` 是一个双向迭代器这种情况对 `reverse_iterator` 进行了特化。在这种情况下，我们就不能再调用 `operator+=`、`operator+`、`operator-=`、`operator-` 以及 `operator[]` 这些成员函数了。

□ `reverse_iterator::base`
`RanIt base() const;`

该成员函数返回 `current`。

□ `reverse_iterator::iterator_type`
`typedef RanIt iterator_type;`

该类型为模板参数 `RanIt` 的同义词。

- reverse_iterator::operator*
 - Ref operator*() const;
 - 该操作符返回*(current - 1)。
- reverse_iterator::operator+
 - reverse_iterator operator+(Dist n) const;
 - 该操作符返回 reverse_iterator(*this) += n。
- reverse_iterator::operator++
 - reverse_iterator& operator++();
 - reverse_iterator operator++(int);
 - 第一个(前递增)操作符先计算 --current, 然后返回*this。
 - 第二个(后递增)操作符先对*this 进行复制, 然后计算 --current, 最后返回那份对*this 的拷贝。
- reverse_iterator::operator+=
 - reverse_iterator& operator+=(Dist n);
 - 该操作符先计算 current + n, 然后返回*this。
- reverse_iterator::operator-
 - reverse_iterator operator-(Dist n) const;
 - 该操作符返回 reverse_iterator(*this) -= n。
- reverse_iterator::operator--
 - reverse_iterator& operator--();
 - reverse_iterator operator--();
 - 第一个(前递减)操作符先计算 ++current, 然后返回*this。
 - 第二个(后递减)操作符先对*this 进行复制, 然后计算 ++current, 最后返回那份对*this 的拷贝。
- reverse_iterator::operator-=
 - reverse_iterator& operator-=(Dist n);
 - 该操作符先计算 current - n, 然后返回*this。
- reverse_iterator::operator->
 - Ptr operator->() const;
 - 该操作符返回&**this。
- reverse_iterator::operator[]
 - Ref operator[](Dist n) const;
 - 该操作符返回*(this + n)。
- reverse_iterator::pointer
 - typedef Ptr pointer;

该类型为模板参数 `Ptr` 的同义词。

reverse_iterator::reference
`typedef Ref reference;`

该类型为模板参数 `Ref` 的同义词。

reverse_iterator::reverse_iterator
`reverse_iterator();`
`explicit reverse_iterator(RarIt x);`
`template<class U>`
`reverse_iterator(const reverse_iterator<U>& x);`

第一个构造函数使用默认构造函数对 `current` 进行初始化。第二个构造函数将 `current` 初始化为 `x.current`。

模板构造函数将 `current` 初始化为 `x.base()`。

使用<iterator>

如果我们在翻译单元中包含任何 STL 头文件，那么 `<iterator>` 就很可能被一同包含入该翻译单元中。只有需要在程序中使用该头文件中的定义并且不指望通过其他 STL 头文件将这些定义带入翻译单元时，才有必要在程序中显式地包含它。但是我们也很有可能需要在程序中使用该头文件中所定义的实体。如果在翻译过程中我们得到了这样的诊断信息：`reverse_iterator` 未定义，请记住：最简单的解决方案就是将这个头文件包含到程序中。

下面是对于 `<iterator>` 中的各种定义的简单介绍：

迭代器

迭代器可以是对象指针或我们所定义的类。如果我们定义了自己的迭代器类，就必须同时为其定义一些成员类型。最简单的方法就是：在可能的情况下，从一个已有的迭代器类公共性地派生出我们自己的迭代器类。否则的话，请按照下面的几种模式中的一种进行：

```
class OutIt
    : public iterator<output_iterator_tag, void, void,
        void, void>
    {....} // for an output iterator
class InIt
    : public iterator<input_iterator_tag, T, Dist>
    {....} // for an input iterator
class FwdIt
    : public iterator<forward_iterator_tag, T, Dist>
    {....} // for a forward iterator
class BidIt
```

```

        : public iterator<bidirectional_iterator_tag, T, Dist>
        {....} // for a bidirectional iterator
    class RanIt
        : public iterator<random_access_iterator_tag, T, Dist>
        {....} // for a random-access iterator

```

在此处，`T` 是迭代器的元素类型，`Dist` 是其差距类型。对于类型为 `T*` 的对象指针来说，种类为 `random_access_iterator_tag`，元素类型当然为 `T`，差距类型为 `ptrdiff_t`。

我们可以通过两种方法中的一种来操作类型为 `Iter` 的迭代器 `first` 和 `last`。在支持部分特化的实现中，我们可以这么写：

iterator_traits advance distance	<pre> iterator_traits<Iter>::iterator_category // the iterator category type iterator_traits<Iter>::value_type // the element type iterator_traits<Iter>::distance_type // the distance type advance(first, N) // adds N to X distance(first, last) // returns the distance from first to last </pre>
---	---

如果实现不支持部分特化，或是不能确信是否支持它，那么本书中提供的 STL 版本可以让我们这么写：

Iter_cat val_type Dist_type advance Distance	<pre> Iter_cat(first) // returns an object of the iterator category type val_type(first) // returns a null pointer to the element type Dist_type(first) // returns a null pointer to the distance type advance(first, N) // adds N to X Distance(first, last, N) // adds to N the distance from first to last </pre>
---	--

在上面提到的所有情况中，迭代器种类始终为一个空结构类型。

迭代器标签	<pre> output_iterator_tag input_iterator_tag forward_iterator_tag bidirectional_iterator_tag random_access_iterator_tag </pre>
--------------	--

reverse_iterator

其中，后三个标签都是从其上面的某个标签公共性地派生出来的。

对于给定的一个类型为 `RanIt` 的随机存取迭代器，我们可以定义一个与之相关的反转型迭代器，并用其来逆向存取序列。

```
typedef reverse_iterator<RanIt> RevIt;
```

为了在指定范围 `[first, last)` 中逆向存取由迭代器 `RanIt` 指定的序列，我们可以这么写：

```
RevIt rfirst(last), rlast(first);
for (; rfirst != rlast; ++rfirst)
    <process>(rfirst);
```

对于类型为 `BidIt` 的双向迭代器，我们也有可能像上面一样使用 `reverse_iterator`。只是需要确保把得到的反转型迭代器看成是一个双向迭代器，那些仅为随机存取迭代器所定义的操作可能会导致编译器产生一些难以理解的诊断信息。但是，如果给定的实现不支持这种用法（或是我们不能确信是否支持它）的话，那么本书中提供的 STL 版本可以让我们这么写：

Revbidit

```
typedef Revbidit<BidIt> RevBidIt;
RevBidIt rfirst(last), rlast(first);
for (; rfirst != rlast; ++rfirst)
    <process>(rfirst);
```

插入型迭代器

我们使用插入型迭代器来将元素插入到容器中，并使之看起来好像在用一个输出迭代器来存储产生的序列。

back_insert_iterator

给定一个类型为 `Cont` 的容器，我们可以定义一个输出迭代器 `OutIt`，用其来向被控序列末端插入元素：

```
typedef back_insert_iterator<Cont> OutIt;
```

这样，我们就可以使用模板函数 `copy`（参见第 6 章）将由区间 `[first, last)` 中的迭代器指定的序列添加到容器 `c` 的末端：

```
OutIt it(c);
copy(first, last, it);
```

back_inserter

或者，我们也可以实时产生所需的输出迭代器，用它来把序列添加到容器的末端：

```
copy(first, last, back_inserter(c));
```

该迭代器通过调用 `Cont::push_back(const T&)` 来完成插入动作。只有在能够有效地完成这个操作的前提下，STL 容器才会定义这个成员函

数。

front_insert_iterator 我们同样可以定义一个输出迭代器 OutIt, 用其来向被控序列前端以逆序的方式插入元素:

```
typedef front_insert_iterator<Cont> Outit;
```

为了以逆序的方式将一个序列插入到容器的前端, 我们可以这么写:

```
OutIt it(c);
copy(first, last, it);
```

front_inserter 或者, 我们可以实时产生所需的输出迭代器, 并用它来把反转的序列添加到容器的前端:

```
copy(first, last, front_inserter(c));
```

该迭代器通过调用 Cont::push_front(const T&) 来完成插入动作。只有在能够有效地完成这个操作的前提下, STL 容器才会定义这个成员函数。

insert_iterator 最后, 我们定义了一个输出迭代器 OutIt, 并用它来向被控序列中指定的位置前面插入元素:

```
typedef insert_iterator<Cont> Outit;
```

为了插入一个序列, 我们以一个迭代器来指定插入点。例如, 为了将一个序列插入到容器的前端且不反转该序列, 我们可以这么写:

```
OutIt it(c, c.begin());
copy(first, last, it);
```

对所有 STL 容器来说, 成员函数 begin() 要么返回指向被控序列中的第一个元素的迭代器, 要么就是指向紧接着一个空序列的末端的下一个位置的迭代器。在这两种情况下, 它都定义了我们想要将被复制的序列插入到其前面的插入点。

inserter 为了实时产生出我们所需的输出迭代器, 并用它来把序列插入到容器的开始处, 我们可以这么写:

```
copy(first, last, inserter(c, c.begin()));
```

该迭代器通过调用 Cont::insert(Cont::iterator, const T&) 来完成插入动作。该成员函数返回一个指向刚刚插入的那个元素的迭代器。不管其开销如何, 每个 STL 容器都定义了这个成员函数。

流迭代器 有时, 在输入流中会包含一系列的字段 (field), 而对于每个字段, 我们都希望将其提取出来并作为某种类型 T 的对象。操作这类流的一种常见方法就是向其写入, 例如:

```
T x;
while (cin >> x)
    <process>(x);
```

当从标准输入中进行提取的动作失败时（通常是在文件结束处），上述循环就会终止。

istream_iterator

通过引入一个类 `istream_iterator` 的对象，我们就可以以一种看上去像是用输入迭代器来遍历一个序列的方式来完成与上面相同的逻辑：

```
typedef istream_iterator<T, char, char_traits<char> > InIt;
```

这种特殊的类型定义使得 `InIt` 可以与一个类 `istream` 的对象（如 `cin`）配合使用。模板类 `char_traits` 是一种更为一般化的机制的一部分，用来指定输入和输出流（在 C++ 标准中对此有介绍）。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成 `istream_iterator<T>`。

给定上面提供的类型定义，我们就可以这么写：

```
InIt first(cin), last;
while (first != last)
    <process>(*first++);
```

在上面提供的代码里面，迭代器完成了我们所需的提取动作。然而，请不要试图通过 `first` 以外的其他方法来存取该输入流。不要直接从流中读取，也不要创建另一个迭代器以从中读取。每个迭代器都必须不时地向前读取所需的对象，这使得输入流很少会保持一种很容易描述的状态（即便 `first` 被销毁后也是如此）。

有时我们可能希望向一个输出流中插入一个由某种类型 `T` 的对象构成的序列。产生这样的流的一种常见方法就是向其写入，例如：

```
while (<not done>)
    {T x = <next value>;
     cout << x; }
```

ostream_iterator

通过引入一个类 `ostream_iterator` 的对象，我们就可以以一种看上去像是用一个输出迭代器来存储生成序列的方式来完成与上面相同的逻辑：

```
typedef ostream_iterator<T,
    char, char_traits<char> > OutIt;
```

这种特殊的类型定义使得 `OutIt` 可以与一个类 `ostream` 的对象（如 `cout`）配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成 `ostream_iterator<T>`。

于是，我们可以这么写：

```
OutIt next(ostream &cout);
while (<not done>)
    {T x = <next value>;
     *next++ = x; }
```

如果想在每次插入动作后添加一个常见的文本串，可以把上面的代码稍做改动，这么写：

```
OutIt next(ostream &cout, "\n");
while (<not done>)
    {T x = <next value>;
     *next++ = x; }
```

在这种情况下，每个被插入的条目后面都紧接着一个换行符。注意，我们在构造函数中所指定的那个以空字符结束的字符串并没有被复制。我们必须确保该字符串的生命周期要长于迭代器的生命周期。

附带说一句，通过 `next` 和其他机制来向输出流中写入是安全的。迭代器对其产生的输出字段没有做任何缓冲。

流缓冲 迭代器

我们可以以一种看上去像是用输入迭代器来存取序列中的元素的方式来从输入流中提取单个元素。为了这样操作标准输入流，我们可以这么写：

```
istreambuf_iterator<char>
    InIt;
```

这种特殊的类型定义使得 `InIt` 可以与一个类 `istream` 的对象（如 `cin`）配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成 `istreambuf_iterator<char>`。

于是，我们就可以这样声明：

```
InIt first(cin), last;
```

如果 `first != last`，表达式 `*first` 就会产生可以从标准输入流中得到的下一个字符（元素类型为 `char`）。迭代器实际上存储了一个指针 `p`，`p` 指向的是底层的流缓冲区（在这种情况下，它的类型为 `streambuf`）。我们也可以将 `first` 声明为：

```
InIt first(cin >rdbuf());
```

在上面提供的代码里面，迭代器完成了我们所需的提取动作。它实质上是通过调用 `p->sgetc()` 或者 `p->sbumpc()` 来完成提取动作的。不要试图通过 `first` 以外的其他方法来存取该输入流。迭代器必须向前读取，这使得输入流很少会保持一种容易描述的状态。不要直接从流中读取，也

不要创建另一个迭代器以从中读取。然而，一旦 `first` 被销毁，我们就可以确信：第一个没有被读取的元素就是流缓冲区中的下一个可以被读取的元素。简单地说，在任何时间内，输入流都应该拥有不超过一个的与之相关的类 `istreambuf_iterator` 的对象。当存在这样的对象时，它就应该是从输入流中提取元素的惟一来源。

我们可以以一种看上去像是用输出迭代器来存储生成序列的方式来向输出流插入单个元素。为了这样操作标准输出流，我们可以这么写：

```
ostreambuf_
iterator      typedef ostreambuf_iterator<char,
                           char_traits<char>>OutIt;
```

这种特殊的类型定义使得 `OutIt` 可以与一个类 `ostream` 的对象（如 `cout`）配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成 `ostreambuf_iterator<char>`。于是，我们就可以这样声明：

```
OutIt next(cout);
```

表达式`*next++ = c` 将字符 `c`（类型为 `char`）插入到标准输出流中。实际上，迭代器存储了一个指针 `p`，`p` 指向的是底层的流缓冲区（在此例中是类 `streambuf`）。我们也可以将 `next` 声明为：

```
OutIt next(cout->rdbuf());
```

迭代器通过调用 `p->putc(c)` 来完成我们所需的插入动作。

附带说一句，通过 `next` 和其他机制来向输出流中写入是安全的。迭代器对其产生的输出字段没有做任何缓冲。

实现 `<iterator>`

本书所提供的实现将一部分名义上是定义在头文件 `<iterator>` 中的定义移动到其他头文件中。新增的头文件 `<xutility>`（它不是 C++ 标准中所指定的头文件）包含一系列定义，它们贯穿使用于一个典型的标准 C++ 库中，甚至在没有显式包含 `<iterator>` 的程序中也可以使用。只要程序中包含 `<iterator>` 就一定会包含 `<xutility>`，我们的实现就可以自由地进行这样的定义重组。实际上，我们也正是这么干的。

实现通常会因为一些不同的原因而导入一些“内部的”头文件，如 `<xutility>` 等。标准 C++ 库中包含不少循环依赖关系——如类 `basic_string` 的对象与类 `exception` 的对象之间就存在这样的循环依赖关系（类 `basic_string` 的对象可以抛出异常，而类 `exception` 的对象可以由类

`basic_string` 的对象构造）。库同样也包含一些比较大的头文件（如 `<algorithm>` 和 `<iterator>`），但可能在其他的头文件中，我们只需要它们中的部分定义。在这两种情况下，将部分定义移到另一个新增的内部头文件中将会使实现者的任务更容易处理。

<xutility>

程序清单 3-1 中列出了文件 `xutility`。它包含了那些广泛使用的 `<iterator>` 和 `<algorithm>` 中的定义。我们将会在第 6 章中讨论它的作用。`xutility` 首先定义了迭代器标签，它们只是一些简单的空类。迭代器标签本身没有存储任何值，它们只是作为模板参数，通过它们的类型和类型层次来传递信息。

程序清单 3-1:

```
// xutility internal header
#ifndef XUTILITY_
#define XUTILITY_
#include <utility>
namespace std {

    // ITERATOR STUFF (from <iterator>

        // ITERATOR TAGS
        struct input_iterator_tag {};
        struct output_iterator_tag {};
        struct forward_iterator_tag
            : public input_iterator_tag {};
        struct bidirectional_iterator_tag
            : public forward_iterator_tag {};
        struct random_access_iterator_tag
            : public bidirectional_iterator_tag {};
        struct Int_iterator_tag {};

        // TEMPLATE CLASS iterator
        template<class C, class T, class D = ptrdiff_t,
                 class Pt = T *, class Rt = T&>
        struct iterator {
            typedef C iterator_category;
            typedef T value_type;
            typedef D difference_type;
            typedef Pt pointer;
            typedef Rt reference;
        };
}
```

```
template<class T, class D, class Pt, class Rt>
    struct Bidit : public iterator<bidirectional_iterator_tag,
        T, D, Pt, Rt> {};
template<class T, class D, class Pt, class Rt>
    struct Ranit : public iterator<random_access_iterator_tag,
        T, D, Pt, Rt> {};
    struct Outit : public iterator<output_iterator_tag,
        void, void, void, void> {};

                                // TEMPLATE CLASS iterator_traits
template<class It>
    struct iterator_traits {
        typedef typename It::iterator_category iterator_category;
        typedef typename It::value_type value_type;
        typedef typename It::difference_type difference_type;
        typedef typename It::pointer pointer;
        typedef typename It::reference reference;
    };

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

                                // TEMPLATE FUNCTION Iter_cat
template<class C, class T, class D,
        class Pt, class Rt> inline
C Iter_cat(const iterator<C, T, D, Pt, Rt>&)
(C X;
```

```
        return (X); }

template<class T> inline
random_access_iterator_tag Iter_cat(const T *)
{random_access_iterator_tag X;
return (X); }

// INTEGER FUNCTION Iter_cat
inline Int_iterator_tag Iter_cat(bool)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(char)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(signed char)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(unsigned char)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(wchar_t)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(short)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(unsigned short)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(int)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(unsigned int)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(long)
{Int_iterator_tag X;
return (X);}

inline Int_iterator_tag Iter_cat(unsigned long)
{Int_iterator_tag X;
return (X);}
```

```
// TEMPLATE FUNCTION Distance
template<class InIt> inline
    typename iterator_traits<InIt>::difference_type
        distance(InIt F, InIt L)
    (typename iterator_traits<InIt>::difference_type
        N = 0;
    Distance2(F, L, N, Iter_cat(F));
    return (N); }
template<class InIt, class D> inline
    void Distance(InIt F, InIt L, D& N)
    (Distance2(F, L, N, Iter_cat(F)); )
template<class InIt, class D> inline
    void Distance2(InIt F, InIt L, D& N,
        input_iterator_tag)
    (for (; F != L; ++F)
        ++N; }
template<class InIt, class D> inline
    void Distance2(InIt F, InIt L, D& N,
        forward_iterator_tag)
    (for (; F != L; ++F)
        ++N; }
template<class InIt, class D> inline
    void Distance2(InIt F, InIt L, D& N,
        bidirectional_iterator_tag)
    (for (; F != L; ++F)
        ++N; }
template<class RanIt, class D> inline
    void Distance2(RanIt F, RanIt L, D& N,
        random_access_iterator_tag)
    (N += L - F; }

// TEMPLATE CLASS Ptrit
template<class T, class D, class Pt, class Rt,
    class Pt2, class Rt2>
    class Ptrit : public iterator<random_access_iterator_tag,
        T, D, Pt, Rt> {
public:
    typedef Ptrit<T, D, Pt, Rt, Pt2, Rt2> Myt;
    Ptrit()
    {}
    explicit Ptrit(Pt P)
```

```
    : current(P) {}

Ptrit(const Ptrit<T, D, Pt2, Rt2, Pt2, Rt2>& X)
    : current(X.base()) {}

Pt base() const
    {return (current); }

Rt operator*() const
    {return (*current); }

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

Myt& operator++()
    {++current;
     return (*this); }

Myt operator++(int)
    {Myt Tmp = *this;
     ++current;
     return (Tmp); }

Myt& operator--()
    {--current;
     return (*this); }

Myt operator--(int)
    {Myt Tmp = *this;
     --current;
     return (Tmp); }

bool operator==(int Y) const
    {return (current == (Pt)Y); }

bool operator==(const Myt& Y) const
    {return (current == Y.current); }

bool operator!=(const Myt& Y) const
    {return (!(*this == Y)); }

Myt& operator+=(D N)
    {current += N;
     return (*this); }

Myt operator+(D N) const
    {return (Myt(current + N)); }

Myt& operator-=(D N)
    {current -= N;
     return (*this); }

Myt operator-(D N) const
    {return (Myt(current - N)); }

Rt operator[](D N) const
    {return (*(*this + N)); }
```

```
    bool operator<(const Myt& Y) const
        {return (current < Y.current); }
    bool operator>(const Myt& Y) const
        {return (Y < *this); }
    bool operator<=(const Myt& Y) const
        {return (!(Y < *this)); }
    bool operator>=(const Myt& Y) const
        {return (!(*this < Y)); }
    D operator-(const Myt& Y) const
        {return (current - Y.current); }
protected:
    Pt current;
};

template<class T, class D, class Pt, class Rt,
          class Pt2, class Rt2> inline
Ptrit<T, D, Pt, Rt, Pt2, Rt2>
operator+(D N,
           const Ptrit<T, D, Pt, Rt, Pt2, Rt2>& Y)
{return (Y + N); }
// TEMPLATE CLASS reverse_iterator
template<class RanIt>
class reverse_iterator : public iterator<
    typename iterator_traits<RanIt>::iterator_category,
    typename iterator_traits<RanIt>::value_type,
    typename iterator_traits<RanIt>::difference_type,
    typename iterator_traits<RanIt>::pointer,
    typename iterator_traits<RanIt>::reference> {
public:
    typedef reverse_iterator<RanIt> Myt;
    typedef typename iterator_traits<RanIt>::difference_type D;
    typedef typename iterator_traits<RanIt>::pointer Pt;
    typedef typename iterator_traits<RanIt>::reference Rt;
    typedef RanIt iterator_type;
    reverse_iterator()
    {}
    explicit reverse_iterator(RanIt X)
        : current(X) {}
template<class U>
reverse_iterator(const reverse_iterator<U>& X)
    : current(X.base()) {}
```

```
RanIt base() const
    {return (current); }

Rt operator*() const
    {RanIt Tmp = current;
     return (*--Tmp); }

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

Myt& operator++()
    {--current;
     return (*this); }

Myt operator++(int)
    {Myt Tmp = *this;
     --current;
     return (Tmp); }

Myt& operator--()
    {++current;
     return (*this); }

Myt operator--(int)
    {Myt Tmp = *this;
     ++current;
     return (Tmp); }

bool Eq(const Myt& Y) const
    {return (current == Y.current); }

// random-access only beyond this point

Myt& operator+=(D N)
    {current += N;
     return (*this); }

Myt operator+(D N) const
    {return (Myt(current + N)); }

Myt& operator-=(D N)
    {current -= N;
     return (*this); }

Myt operator-(D N) const
    {return (Myt(current - N)); }

Rt operator[](D N) const
    {return (*(*this + N)); }

bool Lt(const Myt& Y) const
    {return (Y.current < current); }

D Mi(const Myt& Y) const
    {return (Y.current - current); }

protected:
```

```
RanIt current;
};

// reverse_iterator TEMPLATE OPERATORS
template<class RanIt, class D> inline
reverse_iterator<RanIt> operator+(D N,
    const reverse_iterator<RanIt>& Y)
{return (Y + N); }
template<class RanIt> inline
typename reverse_iterator<RanIt>::D
operator-(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (X.Mi(Y)); }
template<class RanIt> inline
bool operator==(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (X.Eq(Y)); }
template<class RanIt> inline
bool operator!=(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (!(X == Y)); }
template<class RanIt> inline
bool operator<(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (X.Lt(Y)); }
template<class RanIt> inline
bool operator>(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (Y < X); }
template<class RanIt> inline
bool operator<=(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (!(Y < X)); }
template<class RanIt> inline
bool operator>=(const reverse_iterator<RanIt>& X,
    const reverse_iterator<RanIt>& Y)
{return (!(X < Y)); }

// TEMPLATE CLASS Revbidit
template<class BidIt>
class Revbidit : public iterator<
```

```
        typename
iterator_traits<BidIt>::iterator_category,
        typename iterator_traits<BidIt>::value_type,
        typename iterator_traits<BidIt>::difference_type,
        typename iterator_traits<BidIt>::pointer,
        typename iterator_traits<BidIt>::reference> {
public:
    typedef Revbidit<BidIt> Myt;
    typedef typename iterator_traits<BidIt>::difference_type D;
    typedef typename iterator_traits<BidIt>::pointer Pt;
    typedef typename iterator_traits<BidIt>::reference Rt;
    typedef BidIt iterator_type;
    Revbidit()
    {}
    explicit Revbidit(BidIt X)
        : current(X) {}
    BidIt base() const
        {return (current); }
    Rt operator*() const
        {BidIt Tmp = current;
         return (*--Tmp); }
    Pt operator->() const
        {Rt Tmp = **this;
         return (&Tmp); }
    Myt& operator++()
        {--current;
         return (*this); }
    Myt operator++(int)
        {Myt Tmp = *this;
         --current;
         return (Tmp); }
    Myt& operator--()
        {++current;
         return (*this); }
    Myt operator--(int)
        {Myt Tmp = *this;
         ++current;
         return (Tmp); }
    bool operator==(const Myt& Y) const
        {return (current == Y.current); }
    bool operator!=(const Myt& Y) const
```

```
    {return (!(*this == Y)); }

protected:
    BidIt current;
};

// TEMPLATE CLASS istreambuf_iterator
template<class E, class Tr>
class istreambuf_iterator
    : public iterator<input_iterator_tag,
        E, typename Tr::off_type, E *, E&> {
public:
    typedef istreambuf_iterator<E, Tr> Myt;
    typedef E char_type;
    typedef Tr traits_type;
    typedef basic_streambuf<E, Tr> streambuf_type;
    typedef basic_istream<E, Tr> istream_type;
    typedef typename traits_type::int_type int_type;

    istreambuf_iterator(streambuf_type *Sb = 0) throw ()
        : Sbuf(Sb), Got(Sb == 0) {}
    istreambuf_iterator(istream_type& I) throw ()
        : Sbuf(I.rdbuf()), Got(I.rdbuf() == 0) {}
    const E& operator*() const
        {if (!Got)
            ((Myt *)this)->Peek();
         return (Val); }
    const E *operator->() const
        {return (&**this); }
    Myt& operator++()
        {Inc();
         return (*this); }
    Myt operator++(int)
        {if (!Got)
            Peek();
         Myt Tmp = *this;
         Inc();
         return (Tmp); }
    bool equal(const Myt& X) const
        {if (!Got)
            ((Myt *)this)->Peek();
         if (!X.Got)
```

```
        ((Myt *)&X)->Peek();
    return (Sbuf == 0 && X.Sbuf == 0
        || Sbuf != 0 && X.Sbuf != 0); }

private:
    void Inc()
    {if (Sbuf == 0
        || traits_type::eq_int_type(traits_type::eof(),
            Sbuf->sbufmpc())))
        Sbuf = 0, Got = true;
    else
        Got = false; }

E Peek()
{int_type C;
if (Sbuf == 0
    || traits_type::eq_int_type(traits_type::eof(),
        C = Sbuf->sgetc())))
    Sbuf = 0;
else
    Val = traits_type::to_char_type(C);
Got = true;
return (Val); }

streambuf_type *Sbuf;
bool Got;
E Val;
};

// istreambuf_iterator TEMPLATE OPERATORS
template<class E, class Tr> inline
bool operator==(const istreambuf_iterator<E, Tr>& X,
    const istreambuf_iterator<E, Tr>& Y)
{return (X.equal(Y)); }

template<class E, class Tr> inline
bool operator!=(const istreambuf_iterator<E, Tr>& X,
    const istreambuf_iterator<E, Tr>& Y)
{return (!(X == Y)); }

// TEMPLATE CLASS ostreambuf_iterator
template<class E, class Tr>
class ostreambuf_iterator
: public Outit {
typedef ostreambuf_iterator<E, Tr> Myt;
public:
```

```

        typedef E char_type;
        typedef Tr traits_type;
        typedef basic_streambuf<E, Tr> streambuf_type;
        typedef basic_ostream<E, Tr> ostream_type;

        ostreambuf_iterator(streambuf_type *Sb) throw ()
            : Failed(false), Sbuf(Sb) {}
        ostreambuf_iterator(ostream_type& O) throw ()
            : Failed(false), Sbuf(O.rdbuf()) {}
        Myt& operator=(E X)
            {if (Sbuf == 0
                || traits_type::eq_int_type(Tr::eof(),
                    Sbuf->sputc(X)))
                Failed = true;
            return (*this); }
        Myt& operator*()
            {return (*this); }
        Myt& operator++()
            {return (*this); }
        Myt& operator++(int)
            {return (*this); }
        bool failed() const throw ()
            {return (Failed); }

private:
    bool Failed;
    streambuf_type *Sbuf;
};

// ALGORITHM STUFF (from <algorithm>)

        // TEMPLATE FUNCTION copy
template<class InIt, class OutIt> inline
    OutIt copy(InIt F, InIt L, OutIt X)
    {for (; F != L; ++X, ++F)
        *X = *F;
    return (X); }

        // TEMPLATE FUNCTION copy_backward
template<class BidIt1, class BidIt2> inline
    BidIt2 copy_backward(BidIt1 F, BidIt1 L, BidIt2 X)
    {while (F != L)

```

```
    *--X = *--L;
    return (X); }

        // TEMPLATE FUNCTION equal
template<class InIt1, class InIt2> inline
    bool equal(InIt1 F, InIt1 L, InIt2 X)
    {return (mismatch(F, L, X).first == L); }

        // TEMPLATE FUNCTION equal WITH PRED
template<class InIt1, class InIt2, class Pr> inline
    bool equal(InIt1 F, InIt1 L, InIt2 X, Pr P)
    {return (mismatch(F, L, X, P).first == L); }

        // TEMPLATE FUNCTION fill
template<class FwdIt, class T> inline
    void fill(FwdIt F, FwdIt L, const T& X)
    {for (; F != L; ++F)
        *F = X; }

        // TEMPLATE FUNCTION fill_n
template<class OutIt, class Sz, class T> inline
    void fill_n(OutIt F, Sz N, const T& X)
    {for (; 0 < N; --N, ++F)
        *F = X; }

        // TEMPLATE FUNCTION lexicographical_compare
template<class InIt1, class InIt2> inline
    bool lexicographical_compare(InIt1 F1, InIt1 L1,
        InIt2 F2, InIt2 L2)
    {for (; F1 != L1 && F2 != L2; ++F1, ++F2)
        if (*F1 < *F2)
            return (true);
        else if (*F2 < *F1)
            return (false);
    return (F1 == L1 && F2 == L2); }

        // TEMPLATE FUNCTION lexicographical_compare WITH PRED
template<class InIt1, class InIt2, class Pr> inline
    bool lexicographical_compare(InIt1 F1, InIt1 L1,
        InIt2 F2, InIt2 L2, Pr P)
    {for (; F1 != L1 && F2 != L2; ++F1, ++F2)
        if (P(*F1, *F2))
```

```

        return (true);
    else if (P(*F2, *F1))
        return (false);
    return (F1 == L1 && F2 != L2); }

        // TEMPLATE FUNCTION max
template<class T> inline
const T& max(const T& X, const T& Y)
{return (X < Y ? Y : X); }

        // TEMPLATE FUNCTION max WITH PRED
template<class T, class Pr> inline
const T& max(const T& X, const T& Y, Pr P)
{return (P(X, Y) ? Y : X); }

        // TEMPLATE FUNCTION min
template<class T> inline
const T& min(const T& X, const T& Y)
{return (Y < X ? Y : X); }

        // TEMPLATE FUNCTION min WITH PRED
template<class T, class Pr> inline
const T& min(const T& X, const T& Y, Pr P)
{return (P(Y, X) ? Y : X); }

        // TEMPLATE FUNCTION mismatch
template<class InIt1, class InIt2> inline
pair<InIt1, InIt2> mismatch(InIt1 F, InIt1 L, InIt2 X)
{for (; F != L && *F == *X; ++F, ++X)
;
return (pair<InIt1, InIt2>(F, X)); }

        // TEMPLATE FUNCTION mismatch WITH PRED
template<class InIt1, class InIt2, class Pr> inline
pair<InIt1, InIt2> mismatch(InIt1 F, InIt1 L, InIt2 X, Pr P)
{for (; F != L && P(*F, *X); ++F, ++X)
;
return (pair<InIt1, InIt2>(F, X)); }

        // TEMPLATE FUNCTION swap
template<class T> inline
void swap(T& X, T& Y)
{T Tmp = X;
X = Y, Y = Tmp; }
} /* namespace std */

```

```
#endif /* XUTILITY_ */
```

Int_iterator_tag

注意，到目前为止，我们对新增的标签 **Int_iterator_tag** 还没有进行任何讨论。它用来标注一个具有 **integer** 类型的模板参数，这一点就如同 **input_iterator_tag** 用来标注一个可以当作输入迭代器来使用的模板参数一样。**Int_iterator_tag** 和其他迭代器标签一样，被各种 **Iter_cat** 重载版本所使用，这一点将在下面讨论。它在实现某些模板容器类时很有用，解决了某些成员模板函数的重载版本之间存在的歧义。

iterator
Bidit
Ranit
Outit

我们在这个头文件中定义了模板类 **iterator**，同时，我们还定义了另外两个派生自 **iterator** 的模板类：**Bidit** 用于描述双向迭代器，**Ranit** 用于描述随机存取迭代器。这些模板有助于避免在一些容器模板类中经常会出现的命名问题；这时（如果没有定义这两个派生类的话），由于它们都是代指 **iterator**，就会出现该名字重复定义的问题。与之对应的是，类 **Outit** 仅仅是一个缩写以定义贯穿使用于 STL 头文件中的输出迭代器。

iterator_traits

模板类 **iterator_traits** 是最近才引入到 STL 中的一个机制，它用来确定迭代器的属性。对于这个机制来说，对指针和常量指针的那两个部分特化是它不可或缺的一部分。对于那些不支持部分特化的实现来说，模板函数 **Iter_cat**、**Val_type** 以及 **Dist_type** 也能够提供与上面近似一致的信息（我们已经在本章的前面部分描述过这个问题）。这三个模板函数中的任意一个都拥有两个版本：一个用来接受某种形式的迭代器作为参数，另一个则把指针作为参数使用。**Iter_cat** 甚至还对每个基本的 **integer** 类型都有一个重载的版本，所有这些重载的 **Iter_cat** 函数都返回一个类 **Int_iterator_tag** 的对象，用以标识传入的迭代器参数实际上是一个整数。通过使用这三个模板函数，我们就可以用重载函数的解析机制来弥补缺乏部分特化的遗憾了。

distance
Distance

模板函数 **distance** 是一个标准的机制，用来确定同一范围内两个迭代器之间的差距。再次重申一次，我们同样还保留了一个旧的、不依赖于 **iterator_traits**（以及部分特化）的机制来和对象指针配合工作。模板函数 **Distance** 通过计算两个迭代器之间的差距来增加其第三个参数的值，而不是将这个差距的值计算出来并且直接作为该函数的返回值。这两个模板函数最终都是通过调用模板函数 **Distance2** 完成实际差距的计算。这样做虽说简单，但对于不同种类的迭代器，它仍然可以有独到的行为。在此处我们给出了一个非常经典的例子，它使用了标签重载以在一个算法的不同实现版本之间选择最适合的。

这个例子同时也展现了当前某些 C++ 翻译器存在的实际限制。对于 **forward_iterator_tag** 以及 **bidirectional_iterator_tag** 的重载其实并非是必要的。我们之所以从 **input_iterator_tag** 中派生出这两个标签，主要是为

了避免在后续的工作中再回到前面做这件事情，我们给出的例子就属于这种情况。但是我们还是发现，有些翻译器面对这种“艰巨”的情况，不能正确地解析重载。因此，为了一些纯粹的实际需求，我们在此引入了冗余的定义以使得代码更加易于移植。

ptrit

模板类 `Ptrit` 是我们对编译器限制条件的又一个让步。它将对象指针用一个派生自 `iterator` 的类封装起来，该 `iterator` 拥有随机存取迭代器所需的全部属性。你将会在第 10 章中看到它的使用方法。在一个完整的标准 C++ 库的实现中，它也可以被模板类 `basic_string` 所使用。类 `vector<T>` 可以将它的迭代器定义为对象指针 `T*` 的同义词。实际上，许多实现都给了这种做法一定的自由度。在本书中给出的实现将这样的迭代器定义为 `Ptrit<T, ptrdiff_t, T*, T&, T*, T&>`，这样做有两个原因：

- 在使用模板类 `reverse_iterator` 来声明 `vector<T>` 的反转型迭代器时，避免了对模板部分特化的依赖。（详细描述见后续部分。）
- 当程序员不经意地混淆了对象指针和迭代器时，可以防范许多不安全的编程实践。

除此之外，`Ptrit` 还有另一个优点，那就是它可以当作随机存取迭代器类的极好的原型。

一开始，我们可能会认为最后的两个模板参数是冗余的。然而，更进一步的检查显示，它们有助于指定某些种类的转换构造函数。对于一个常量迭代器来说，这样的构造函数将会把它的非常量兄弟转换成为常量迭代器。而对于非常量迭代器来说，这样的构造函数实际上就相当于一个显式指定的默认复制构造函数。（我们不希望出现隐式地将一个常量迭代器转换成为一个非常量迭代器的情况，但我们却希望与之相反的转换过程能够是隐式的。）

reverse_iterator

模板类 `reverse_iterator` 从一个随机存取迭代器中创建出一个反转型迭代器。当你了解了设计反转型迭代器的实质原理后，就会发现，其实在 `reverse_iterator` 的代码中并没有用到什么特殊的技巧。下面是我们从它的代码中提取出来的成员函数 `operator*()`：

```
Ref operator*() const
    {return (*current - 1); }
```

简而言之，反转型迭代器返回的是在 `current` 指向处前一位置的元素。当我们仔细想过之后，会发现理由其实很简单。我们一般都是用迭代器的范围 `[first, last)` 来指定一个序列。这种半开半闭的表达方法意味着：对于一个非空的序列来说，`first` 是它的一部分，而 `last` 只能是越过其结尾。

于是我们为 `current` 指定了反转型迭代器，范围为 `[last, first)`。为了使 `last` 能够作为存储在 `current` 中的值使用，在一个非空的序列中，必须

指定 `current - 1` 处的元素。一旦做了这样的转变，其他所有迭代器的属性也就自然起来。

Revbidit

模板类 `Revbidit` 从一个双向迭代器中建出一个反转型迭代器。大部分在模板类 `reverse_iterator` 中实现了的方法都在它之中实现出来了。如前面所描述的一样，成员函数 `operator*()` 的定义体现出最基本的技巧。

istreambuf_iterator

模板类 `istreambuf_iterator` 创建了一个输入迭代器，并用它来从一个流缓冲区中提取出类型为 `E`（通常为 `char`）的元素。虽然输入迭代器可能没有输出迭代器那么奇怪（如我们将在下面紧接着讨论的模板类 `ostreambuf_iterator`），但它还是可以被风格化的。设计一个输入迭代器所用的技巧包括：

- 向前读取，必要时拥有一个输入值。
- 检测及记录序列结束（或文件结束）。
- 创建一个 `end-of-sequence` 值。
- 比较两个迭代器，以检测它们是否到达序列的末端。

模板类 `istreambuf_iterator` 为输入迭代器提供了一个很好的原型，但你还是必须将它与本书中所提供的实现中的其他输入迭代器仔细比较，以弄懂它们之间出现不同的原因。例如，该模板类与早先的模板类 `istream_iterator`（将在本章稍后处讨论）的主要不同在于向前读取的逻辑。后者必须读入一堆元素以检测它所要提取的值（类型为 `U`），这使得它在将向前读取的代价变得最小化方面没有做太多的努力。

从另一方面来说，标准 C++ 库可以在从输入流中提取值的过程中创建及销毁任意数量的模板类 `istreambuf_iterator` 的对象。销毁每个这样的对象使得输入流处于一个可以获知的状态中，并且不会漏掉任何未被读取的输入。通过使用它的成员函数 `sgetc()`，流缓冲区可以让我们“取数（`peek`）”输入的元素，也就是说：获得输入的值但不在流中移动定位用的游标。但即使是这样的操作也会导致程序无限期地中止，举例来说，如果该流缓冲区是和一个交互式的输入流（如键盘）相关联的话，就会导致这种情况出现。也就是说，在它必须将所指向的值传给成员函数之前，该迭代器将会推迟所有“需要知道下一个元素的值”的动作。注意，取数操作必须在比较两个迭代器之前进行，以检测该迭代器是否到达了序列结束处。

ostreambuf_iterator

模板类 `ostreambuf_iterator` 创建了一个输出迭代器，用于将类型为 `E`（通常为 `char`）的元素插入到流缓冲区中。它与早先的模板类 `ostream_iterator`（将在本章稍后处讨论）的主要的不同在于对输出错误的处理。一般来说，输出迭代器就像是一个能够无限地产生出元素的源头。没有任何方法可以报告它是否还能存储。但是，标准 C++ 库需要有一种方法，用来拒绝从底层的流缓冲区中向上传递的错误。这也就是我

们所新增的成员函数 failed 及其相关逻辑的由来。

从表面判断，这个模板类中的成员操作符的定义让人难以容忍。它们并没有提供通常与这些操作符相关的语义：

- operator*()返回的是迭代器的引用，而不是它指向的值。
- operator++()并没有将迭代器向前移动。
- operator++(int)也没有将迭代器向前移动。
- operator=(E)是对赋值操作符的一个非常规的重载。它完成了实际上的输出操作，并按照其语义，将迭代器有效地向前移动了。

然而，请记住：对于输出迭代器 next 的使用，只能存在于下面的两种格式中：

```
for (;<not done>; ++next)
    *next = <whatever>;
```

或

```
while (<not done>)
    *next++ = <whatever>
```

粗略看一下在这两种格式的循环中定义的这些成员操作符。我们将会看到，这些操作符提供了最少的功能，以满足输出迭代器的语义需求。

也就是说，模板类 ostreambuf_iterator 为我们可能实现的输出迭代器提供了一个很好的原型。STL 定义了几种不同的输出迭代器（将在本章及后续章节中讨论）。请仔细学习它们之间的不同处和相同处。输出迭代器是一个强有力的工具，但它们也很特殊，并且隐藏了一些缺陷。

iterator

文件 xutility 的其他部分包括一些与头文件<algorithm>相关的定义。第 6 章将对这些定义进行讨论。程序清单 3-2 列出了文件 iterator。它定义了剩下的那些与头文件<iterator>相关但又未在文件 xutility 中定义的实体。

程序清单 3-2:

iterator

```
// iterator standard header
#ifndef ITERATOR_
#define ITERATOR_
#include <xutility>
namespace std {
    // TEMPLATE CLASS back_insert_iterator
    template<class C>
    class back_insert_iterator
        : public Outit {
    public:
        typedef C container_type;
        typedef typename C::reference reference;
        typedef typename C::value_type value_type;
```

```
    explicit back_insert_iterator(C& X)
        : container(&X) {}
    back_insert_iterator<C>& operator=(  
        typename C::const_reference V)
        {container->push_back(V);
         return (*this); }
    back_insert_iterator<C>& operator*()
        {return (*this); }
    back_insert_iterator<C>& operator++()
        {return (*this); }
    back_insert_iterator<C> operator++(int)
        {return (*this); }
protected:  
    C *container;
    };
template<class C> inline
    back_insert_iterator<C> back_inserter(C& X)
    {return (back_insert_iterator<C>(X)); }

// TEMPLATE CLASS front_insert_iterator
template<class C>
    class front_insert_iterator
        : public Outit {
public:  
    typedef C container_type;
    typedef typename C::reference reference;
    typedef typename C::value_type value_type;
    explicit front_insert_iterator(C& X)
        : container(&X) {}
    front_insert_iterator<C>& operator=(  
        typename C::const_reference V)
        {container->push_front(V);
         return (*this); }
    front_insert_iterator<C>& operator*()
        {return (*this); }
    front_insert_iterator<C>& operator++()
        {return (*this); }
    front_insert_iterator<C> operator++(int)
        {return (*this); }
protected:  
    C *container;
    };
template<class C> inline
    front_insert_iterator<C> front_inserter(C& X)
    {return (front_insert_iterator<C>(X)); }
```

```

        // TEMPLATE CLASS insert_iterator
template<class C>
    class insert_iterator
        : public Outit {
public:
    typedef C container_type;
    typedef typename C::reference reference;
    typedef typename C::value_type value_type;
    insert_iterator(C& X, typename C::iterator I)
        : container(&X), iter(I) {}
    insert_iterator<C>& operator=(  

        typename C::const_reference V)  

        {iter = container->insert(iter, V);  

++iter;  

        return (*this); }
    insert_iterator<C>& operator*()  

        {return (*this); }
    insert_iterator<C>& operator++()  

        {return (*this); }
    insert_iterator<C>& operator++(int)
        {return (*this); }
protected:
    C *container;
    typename C::iterator iter;
    };
template<class C, class XI> inline
    insert_iterator<C> inserter(C& X, XI I)
    {return (insert_iterator<C>(X, C::iterator(I))); }

        // TEMPLATE CLASS istream_iterator
template<class T, class E = char,
         class Tr = char_traits<E>,
         class Dist = ptrdiff_t>
    class istream_iterator
        : public iterator<input_iterator_tag, T, Dist,
          T *, T& > {
public:
    typedef istream_iterator<T, E, Tr, Dist> Myt;
    typedef E char_type;
    typedef Tr traits_type;
    typedef basic_istream<E, Tr> istream_type;
    istream_iterator()
        : Istr(0) {}
    istream_iterator(istream_type& I)

```

```
        : Istr(&I) {Getval(); }
    const T& operator*() const
        {return (Val); }
    const T *operator->() const
        {return (&**this); }

    Myt& operator++()
        {Getval();
         return (*this); }
    Myt operator++(int)
        (Myt Tmp = *this;
         Getval();
         return (Tmp); }
    bool Equal(const Myt& X) const
        {return (Istr == X.Istr); }
protected:
    void Getval()
        {if (Istr != 0 && !(*Istr >> Val))
         Istr = 0; }
    istream_type *Istr;
    T Val;
};

// istream_iterator TEMPLATE OPERATORS
template<class T, class E, class Tr, class Dist> inline
bool operator==(const istream_iterator<T, E, Tr, Dist>& X,
                  const istream_iterator<T, E, Tr, Dist>& Y)
{return (X.Equal(Y)); }
template<class T, class E, class Tr, class Dist> inline
bool operator!=(const istream_iterator<T, E, Tr, Dist>& X,
                  const istream_iterator<T, E, Tr, Dist>& Y)
{return (!(X == Y)); }

// TEMPLATE CLASS ostream_iterator
template<class T, class E = char,
          class Tr = char_traits<E> ,
          class ostream_iterator
          : public Outit {
public:
    typedef T value_type;
    typedef E char_type;
    typedef Tr traits_type;
    typedef basic_ostream<E, Tr> ostream_type;
```

```

        ostream_iterator(ostream_type& O,
            const E *D = 0)
        : Ostr(&O), Delim(D) {}
    ostream_iterator<T, E, Tr>& operator=(const T& X)
        {*Ostr << X;
        if (Delim != 0)
            *Ostr << Delim;
        return (*this); }
    ostream_iterator<T, E, Tr>& operator*()
        {return (*this); }
    ostream_iterator<T, E, Tr>& operator++()
        {return (*this); }
    ostream_iterator<T, E, Tr> operator++(int)
        {return (*this); }

protected:
    const E *Delim;
    ostream_type *Ostr;
};

// TEMPLATE FUNCTION Val_type
template<class It> inline
typename iterator_traits<It>::value_type *Val_type(It)
{return (0); }

// TEMPLATE FUNCTION advance
template<class Init, class D> inline
void advance(Init& I, D N)
{Advance(I, N, Iter_cat(I)); }
template<class Init, class D> inline
void Advance(Init& I, D N, input_iterator_tag)
{for (; 0 < N; --N)
    ++I; }
template<class FwdIt, class D> inline
void Advance(FwdIt& I, D N, forward_iterator_tag)
{for (; 0 < N; --N)
    ++I; }
template<class BidIt, class D> inline
void Advance(BidIt& I, D N, bidirectional_iterator_tag)
{for (; 0 < N; --N)
    ++I;
for (; N < 0; ++N)
    --I; }
template<class RanIt, class D> inline
void Advance(RanIt& I, D N, random_access_iterator_tag)

```

```

    { l += N; }

    // TEMPLATE FUNCTION Dist_type
template<class It> inline
    typename iterator_traits<It>::difference_type
        *Dist_type(It)
    { return (0); }
} /* namespace std */
#endif /* ITERATOR_ */

```

back_insert_iterator

模板类 `back_insert_iterator` 创建了一个输出迭代器，用来往容器末端添加新的元素。与其他输出迭代器一起，它使用了一个不常见的赋值操作符 `operator=(Cont::const_reference V)` 来实现这样的添加动作。实际上的添加动作是由它调用指定容器的 `container->push_back(V)` 来完成的。

back_inserter

模板函数 `back_inserter` 创建了类 `back_insert_iterator<Cont>` 的一个对象，并使之可以与其容器参数协同工作。

front_insert_iterator

模板类 `front_insert_iterator` 创建了一个输出迭代器，它通过调用 `container->push_front(V)`，在容器的前端添加元素（逆序）。在其他方面，它都类似于模板类 `back_insert_iterator`。

front_inserter

模板函数 `front_inserter` 创建了类 `front_insert_iterator<Cont>` 的一个对象，并使之可以与其容器参数协同工作。

insert_iterator

模板类 `insert_iterator` 创建了一个输出迭代器，用它来向容器中的指定位置插入元素。与模板类 `back_insert_iterator` 不同的是，这样的迭代器必须还保存一个迭代器 `iter`，用来指定迭代器中的插入点。请注意在这两个模板类之中成员函数 `operator++(int)` 返回值的不同。保存有状态改变信息的迭代器必须返回一个该迭代器的最新版本的引用。而传统的返回一个新对象的方法在此时就不再适合了。

inserter

模板函数 `inserter` 创建了类 `insert_iterator<Cont, Iter>` 的一个对象，并使之可以与其容器参数协同工作。

istream_iterator

模板类 `istream_iterator` 创建了一个输入迭代器，并通过使用一个合适的 `operator>>` 的重载，从输入流中提取类型为 `U` 的值。请将它与我们在本章早些时候描述的模板类 `istreambuf_iterator` 仔细比较。

ostream_iterator

模板类 `ostream_iterator` 创建了一个输出迭代器，并通过使用一个合适的 `operator<<` 的重载，向输出流中插入类型为 `U` 的值。它遵循了输出迭代器的通常形式。请将它与我们在本章早些时候描述的模板类 `istreambuf_iterator` 仔细比较。

val_type
dist_type

模板函数 `Val_type` 返回一个空指针，它的类型揭示了该函数的迭代器参数的“值类型”（即成员类型 `value_type`）。同样地，模板函数

`Dist_type` 返回的是“差距类型”（即成员类型 `difference_type`，有段时间我们也把它叫做 `distance_type`）。在本章的早些时候，我们就已经讨论过这些看起来没什么意义的函数存在的历史原因。

advance

最后，模板函数 `advance` 实现了一件显然是很简单的事情：把迭代器向前移动指定次数。但它却更像前面讨论的模板函数 `distance`。在不同种类的迭代器中，对迭代器向前移动的最佳方法都不相同。我们在此又使用了一次标签的重载，以在算法的不同实现中选择最佳的。

测试`<iterator>`

`titerator.c`

程序清单 3-3 列出了文件 `titerator.c`。由于头文件 `<iterator>` 太大了，我们的测试程序就将它分做不同部分，分别进行了测试：

- 用以测试迭代器的不同属性及完成有关迭代器的简单操作的模板
 - 模板类 `reverse_iterator`
 - 插入型迭代器
 - 模板类 `istream_iterator`
 - 模板类 `ostream_iterator`
 - 模板类 `istreambuf_iterator`
 - 模板类 `ostreambuf_iterator`

程序清单 3-3:

```
// test <iterator>
#include <assert.h>
#include <iostream>
#include <string.h>
#include <strstream>
#include <deque>
#include <iterator>
using namespace std;

typedef char *PtrIt;

// TEST GENERAL PROPERTY TEMPLATES
void takes_ran_tag(random_access_iterator_tag)
{}

void test_prop()
{
    random_access_iterator_tag *ran_tag =
        (random_access_iterator_tag *)0;
```

```
bidirectional_iterator_tag *bid_tag =
    (random_access_iterator_tag *)0;
forward_iterator_tag *fwd_tag =
    (bidirectional_iterator_tag *)0;
input_iterator_tag *in_tag =
    (forward_iterator_tag *)0;
output_iterator_tag *p_out_tag = 0;

typedef iterator<input_iterator_tag, float, short,
    float *, float&> Iter;
float f1;
Iter::iterator_category *it_tag =
    (input_iterator_tag *)0;
Iter::value_type *it_val = (float *)0;
Iter::difference_type *it_dist = (short *)0;
Iter::pointer it_ptr = (float *)0;
Iter::reference it_ref = f1;

typedef iterator_traits<Iter> Traits;
Traits::iterator_category *tr_tag =
    (input_iterator_tag *)0;
Traits::value_type *tr_val = (float *)0;
Traits::difference_type *tr_dist = (short *)0;
Traits::pointer tr_ptr = (float *)0;
Traits::reference tr_ref = f1;
typedef iterator_traits<PtrIt> Ptraits;
char ch;
takes_ran_tag(Ptraits::iterator_category());
Ptraits::value_type *ptr_val = (char *)0;
Ptraits::difference_type *ptr_dist = (ptrdiff_t *)0;
Ptraits::pointer ptr_ptr = (char *)0;
Ptraits::reference ptr_ref = ch;

const char *pc = "abcdefg";
advance(pc, 4);
assert(*pc == 'e');
advance(pc, -1);
assert(*pc == 'd');
assert(distance(pc, pc + 3) == 3); }

// TEST reverse_iterator
```

```
typedef reverse_iterator<PtrIt> RevIt;
class MyrevIt : public RevIt {
public:
    MyrevIt(RevIt::iterator_type p)
        : RevIt(p) {}
    RevIt::iterator_type get_current() const
        {return (current); }
};

void test_revit()
{
    char *pc = (char *)"abcdefg" + 3;
    PtrIt pcit(pc);
    RevIt::iterator_type *p_iter = (PtrIt *)0;
    RevIt rit0, rit(pcit);

    assert(rit.base() == pcit);
    assert(*rit == 'c');
    assert(*++rit == 'b');
    assert(*rit++ == 'b' && *rit == 'a');
    assert(*--rit == 'b');
    assert(*rit-- == 'b' && *rit == 'c');
    assert(*(rit += 2) == 'a');
    assert(*(rit -= 2) == 'c');
    assert(*(rit + 2) == 'a' && *rit == 'c');
    assert(*(rit - 2) == 'e' && *rit == 'c');
    assert(rit[2] == 'a');
    assert(rit == rit);
    assert(!(rit < rit) && rit < rit + 1);
    assert((rit + 2) - rit == 2);

    MyrevIt myrit(pct);
    assert(myrit.get_current() == pcit); }

// TEST INSERTION ITERATORS
typedef deque<char, allocator<char> > Cont;
typedef back_insert_iterator<Cont> BackIt;
class MybackIt : public BackIt {
public:
    MybackIt(BackIt::container_type& c)
        : BackIt(c) {}
    BackIt::container_type *get_container() const
```

```
        {return (container); }
    };

typedef front_insert_iterator<Cont> FrontIt;
class MyfrontIt : public FrontIt {
public:
    MyfrontIt(FrontIt::container_type& c)
        : FrontIt(c) {}
    FrontIt::container_type *get_container() const
        {return (container); }
};

typedef insert_iterator<Cont> InsIt;
class MyinsIt : public InsIt {
public:
    MyinsIt(InsIt::container_type& c, Cont::iterator it)
        : InsIt(c, it) {}
    InsIt::container_type *get_container() const
        {return (container); }
    Cont::iterator get_iterator() const
        {return (iter); }
};

void test_inserts()
{
    Cont c0;
    char ch;
    BackIt::container_type *pbi_cont = (Cont *)0;
    BackIt::reference pbi_ref = ch;
    BackIt::value_type *pbi_val = (char *)0;
    BackIt bit(c0);
    *bit = 'a', ++bit;
    *bit++ = 'b';
    assert(c0[0] == 'a' && c0[1] == 'b');
    MybackIt mybkit(c0);
    assert(mybkit.get_container() == &c0);
    *back_inserter(c0)++ = 'x';
    assert(c0[2] == 'x');

    FrontIt::ccontainer_type *pfi_cont = (Cont *)0;
    FrontIt::reference pfi_ref = ch;
    FrontIt::value_type *pfi_val = (char *)0;
```

```

        FrontIt fit(c0);
        *fit = 'c', ++fit;
        *fit++ = 'd';
        assert(c0[0] == 'd' && c0[1] == 'c');
        MyfrontIt myfrit(c0);
        assert(myfrit.get_container() == &c0);
        *front_inserter(c0)++ = 'y';
        assert(c0[0] == 'y');

        InsIt::container_type *pii_cont = (Cont *)0;
        InsIt::reference pii_ref = ch;
        InsIt::value_type *pii_val = (char *)0;
        InsIt iit(c0, c0.begin());
        *iit = 'e', ++iit;
        *iit++ = 'f';
        assert(c0[0] == 'e' && c0[1] == 'f');
        MyinsIt myinsit(c0, c0.begin());
        assert(myinsit.get_container() == &c0);
        assert(myinsit.get_iterator() == c0.begin());
        *inserter(c0, c0.begin())++ = 'z';
        assert(c0[0] == 'z'); }

        // TEST istream_iterator

void test_istreamit()
{
    istrstream istr("0 1 2 3");
    typedef istream_iterator<int, char,
        char_traits<char>, ptrdiff_t> IstrIt;
    IstrIt::char_type *p_char = (char *)0;
    IstrIt::traits_type *p_traits = (char_traits<char> *)0;
    IstrIt::istream_type *p_istream = (istream *)0;
    IstrIt iit0, iit(istr);
    int n;
    for (n = 0; n < 5 && iit != iit0; ++n)
        assert(*iit++ == n);
    assert(n == 4); }

        // TEST ostream_iterator

void test_ostreamit()
{
    ostrstream ostr0, ostr;
    typedef ostream_iterator<int, char,

```

```
        char_traits<char> > OstrIt;
OstrIt::value_type *p_val = (int *)0;
OstrIt::char_type *p_char = (char *)0;
OstrIt::traits_type *p_traits = (char_traits<char> *)0;
OstrIt::ostream_type *p_ostream =
    (basic_ostream<char, char_traits<char> > *)0;
OstrIt oit0(ostr0), oit(ostr, "||");
*oit0 = 1, ++oit0;
*oit0++ = 2;
ostr0 << ends;
assert(strcmp(ostr0.str(), "12") == 0);
ostr.freeze(false);

*oit = 1, ++oit;
*oit++ = 2;
ostr << ends;
assert(strcmp(ostr.str(), "1||2||") == 0);
ostr.freeze(false); }

// TEST istreambuf_iterator
void test_istrbufit()
{istrstream istr("0123"), istr1("");
typedef istreambuf_iterator<char,
    char_traits<char> > IsbIt;
IsbIt::char_type *p_char = (char *)0;
IsbIt::traits_type *p_traits = (char_traits<char> *)0;
IsbIt::int_type *p_int = (int *)0;
IsbIt::streambuf_type *p_streambuf =
    (basic_streambuf<char, char_traits<char> > *)0;
IsbIt::istream_type *p_istream =
    (basic_istream<char, char_traits<char> > *)0;
IsbIt iit0, iit(istr), iit1(istr1.rdbuf());
int n;
for (n = 0; n < 5 && iit != iit0; ++n)
    assert(*iit++ == n + '0');
assert(n == 4);
assert(iit0.equal(iit1)); }
```

```
// TEST ostreambuf_iterator
void test_ostreambufit()
{
    ostrstream ostr;
    typedef ostreambuf_iterator<char,
        char_traits<char> > Osbit;
    Osbit::char_type *p_char = (char *)0;
    Osbit::traits_type *p_traits = (char_traits<char> *)0;
    Osbit::streambuf_type *p_streambuf =
        (basic_streambuf<char, char_traits<char> > *)0;
    Osbit::ostream_type *p_ostream =
        (basic_ostream<char, char_traits<char> > *)0;
    Osbit oit0((Osbit::streambuf_type *)0), oit(ostr);
    *oit0++ = 'x';
    assert(oit0.failed());

    *oit = '1', ++oit;
    *oit++ = '2';
    ostr << ends;
    assert(strcmp(ostr.str(), "12") == 0);
    assert(!oit.failed());
    ostr.freeze(false); }

// TEST <iterator>
int main()
{
    {test_prop();
    test_revit();
    test_inserts();
    test_istreamit();
    test_ostreamit();
    test_istrbufit();
    test_ostreambufit();
    cout << "SUCCESS testing <iterator>" << endl;
    return (0); }
```

如果所有的测试都正常运行的话，测试程序将打印出：

```
SUCCESS testing <iterator>
```

并正常退出。

习题

- 习题3-1 为什么`input_iterator_tag`不派生自`output_iterator_tag`?
- 习题3-2 STL为什么要定义模板类`reverse_forward_iterator`?
- 习题3-3 定义一个迭代器，它可以将元素插入到容器的开始处，但与模板类`front_insert_iterator`不同的是，它的插入动作并不是逆序的。
- 习题3-4 定义一个迭代器，它可以将元素插入到容器的末端，但是插入是以逆序进行的（就好像模板类`front_insert_iterator`的对象一样）。
- 习题3-5 对迭代器来说，成员函数`operator++(int)`通常都返回该迭代器的一份拷贝。然而模板类`insert_iterator`及模板类`ostreambuf_iterator`返回的却是对象本身的一个引用。请解释它们这样做的原因。
- 习题3-6 定义一个模板类`forward_iterator<FwdIt>`，它简单地“包装”了一个类`FwdIt`的前向迭代器。请确保它包括所有前向迭代器要求的操作。
- 习题3-7 [较难] 定义一系列“严格的”迭代器，以包装所有的五种迭代器中的每一种。这些“严格的”迭代器应该能够报告出程序中对它们的错误使用。当迭代器被创建后，它们甚至还可以对试图将迭代器向前移动到指定的范围外这种情况给出警告信息。
- 习题3-8 [特难] 给定一个迭代器以及它所属的种类，如何验证该迭代器的行为不超过其种类所要求的行为集？

第 4 章 <memory>

背景知识

在 STL 的 13 个头文件中，<memory>是其中最难描述的一个。它以不同寻常的方式为容器中的元素分配存储空间。它同样也为某些算法执行期间产生的临时对象提供机制。早期的 STL 代码中只存在一些与之大体上相似的方法，而且也不值得炫耀。例如，在流行的 PC 体系结构上，通过采用一种没有被 C 标准所概括的指针表达形式，容器可以在“近堆（near heap）”或“远堆（far heap）”上分配元素。然而，在 C++ 标准中，这种特殊的分配方法已经被分配器（allocator，一种在分配及释放元素存储空间时使用的中间对象）所代替。

allocator

头文件<memory>中的主要部分就是模板类 allocator，由它产生所有容器中的默认分配器。该模板类的最初版本很快就由一个更加值得炫耀的版本所代替。新的版本有一个很显著的缺陷——在它被引入到 C++ 标准时的 C++ 编译技术下，它不可能被实现。随后，这个版本又被另外的、同样也是值得炫耀的版本替换了两次——不幸的是，这些新的版本当时也同样是不可实现的。只有到了近期，为了支持分配器而需要的语言特性才开始广泛地应用开来。因此，我们实际使用分配器的经验是有所限制的。

好消息是：我们很少需要考虑直接与分配器打交道。我们所得到的默认分配器已经能够很好地完成所需的大部工作了。坏消息是：分配器仍然是到处存在的。至少在标准 C++ 库所定义的每个 STL 容器模板类的模板参数中，必然有一个代表了其分配器类型。在每个容器对象所存储的对象中，必然有一个是它的分配器。幸运的是，我们可以通过默认的模板参数来得到一个默认的分配器类型，而默认的函数参数则为我们产生了默认的分配器对象。对于实现者来说，分配器让他们头痛，使他们不得不关注于一些细微之处并将这些细微之处添加到他们的 C++ 翻译器中。但是，对于那些只想使用 STL 的程序员来说，它们不应该成为妨害他们工作的讨厌事。

我们在本章中最初的目的是讨论在书写 STL 代码时模板类 allocator 所能起到的作用。我们将向大家展示分配器是如何与它们所支持的模板函数一同工作的。我们将概括出那些可能会导致我们定义自己的分配器情况。与以往一样，我们也将展示如何实现模板类 allocator 以及在头文

为什么要 有分配器？

件<memory>中定义的其他有用的工具。更进一步，我们同样也会描述一些可能会在那些缺少必要机制的编译器中碰到的折衷分配器。

模板容器类所提供的一个重要的服务就是存储空间管理。容器对象会为其管理的序列中的元素分配及释放存储空间。有些容器还必须分配一部分附加的存储空间。例如，列表（list）需要存储的不但有元素本身，还有那些将元素链接在一起的指针。双队列（deque）通常都维护着一张图，这张图是一个指针数组，指向了不同的用以存储元素的内存块。容器也就因而封装了两种重要的服务：

- 隐藏实际中对存储空间的分配及释放的细节。
- 确保所有被分配的存储空间最终都能获得释放。

分配器对象代表容器对象处理第一项工作。它拥有用于分配及释放给定类型的对象的成员函数。它同样也可以在指定的存储空间上对这些对象进行构造及析构。它甚至还能够告诉我们，对于给定的类型，我们最多可同时分配几个对象。

我们可以通过两个步骤将一个分配器与一个容器联系起来。在特化一个模板容器类型时，可以与容器所维护的元素类型一道，指定该容器的分配器类型。例如，为了定义一个类 Mylist，它维护着一个由 float 元素组成的列表，并且其分配器的类为 Myalloc，我们可以这么写：

```
#include <list>
...
typedef list<float, Myalloc> Mylist;
```

于是，当我们构造一个类 Mylist 的对象时，就指定了实际的分配器对象，如：

```
Myalloc an_allocator;
Mylist a_list(an_allocator);
```

然而，通过使用默认的模板参数，可以不必这么麻烦，只需简单地写成：

```
typedef list<float> Mylist2;
```

在这种情况下，我们所提供的默认 allocator 类型是 allocator<float>。同样地，我们也可以在构造函数中忽略 allocator 这个参数，如：

```
Mylist a_list;
```

这时所提供的默认 allocator 对象就是 Myalloc() 了。

自定义的 分配器

我们为什么需要写自定义的分配器呢？例如，我们有时可能需要维护一个已释放列表元素的存储池。默认的分配器通过调用 operator new^①

① 有关 operator new 以及 operator delete 的详细讨论，可以参考《More Effective C++》中的 Item 8。
——译者注

来满足对新元素的需求。同样地，它为每个已释放元素调用 `operator delete`。但一般来说，从这样的存储池中回收列表元素的操作相当快。自定义的分配器可以将释放掉的元素添加到一个自由列表上，然后就可以在需要分配时从该列表上直接获取元素。

但分配器并不仅仅是提供对私有内存池的管理那么简单。它们同时也指定各种各样的、与存取它所分配及释放的对象相关的类型。这就使事情变得复杂起来。在一个典型的 C++ 程序中，我们可以知道类型为 T 的对象 x 的部分信息：

- 表达式`&x`的类型要么是 `T *`，要么就是 `const T *`。
- 对 `x` 的引用，其类型要么是 `T&`，要么就是 `const T&`。
- 如果 `p` 是一个指向 `x` 的指针，那么表达式`*p` 的类型为 `T`。
- 对于可能存在的元素类型为 `T` 的最大数组，其元素的个数可以由一个类型为 `size_t` 的对象表示。
- 两个指向 `T` 的指针之间的差距类型为 `ptrdiff_t`。

如果你了解 C 和 C++ 的底层细节的话，所有这一切都显而易见。但它们并不总是必需为真的。

远指针

例如，如我们在前面所提到的一样，请考虑一个支持远堆的基于 PC 的实现。堆在由那些大小大于常规指针的特殊指针来寻址的存储器中实现。这虽然不是标准 C++ 所支持的，但它确实是一种被广泛支持的方言。即使在程序的其他部分所使用的指针是普通大小的指针时，将我们的列表元素分配到远堆上也是有意义的。在这种情况下：

- 表达式`&x`的类型要么是 `T far *`，要么就是 `const T far *`。
- 对 `x` 的引用，其类型要么是 `T far &`，要么就是 `const T far &`。
- 如果 `p` 是一个指向 `x` 的远指针，那么表达式`*p` 的类型为 `T`（如前所述）。
- 对于可能存在的元素类型为 `T` 的最大数组，其元素的个数可以由一个类型为 `unsigned long` 的对象表示。
- 两个指向 `T` 的指针之间的差距类型为 `long`。

对于分配器的限制

分配器类包含这些基本属性的类型定义。我们对每个 STL 容器的书写都十分谨慎，尽量从它的分配器类型参数中获得这些类型信息。通过使用一个构造巧妙的分配器模板类，我们就可以对 STL 容器中元素的存储管理进行一些适当的、大胆的操作。那么，我们可以做到多么大胆呢？目前这还处在讨论中。在 C++ 标准允许多大的自由度，以使得我们可以对所分配对象的基本操作进行重定义这个问题上，标准化委员会中存在着不同的意见。由于近指针及远指针是非标准扩展，所以 C++ 标准对此特殊的主题保持了沉默。虽然现在存在着支持远堆的实现，但它们是不可移植的。

智能指针

在一个可移植的 C++ 程序中，我们可以维护一个上面所描述过的存储池。但我们是否可以定义一个分配器，使它在分配新的对象时返回一个“智能的（smart）”指针呢？智能指针 `p` 是某个类的一个对象，其行为类似一个指针。表达式 `*p` 将会产生一个对象的引用，这看起来好像没什么，但它却给了成员函数 `operator*()` 一个机会去完成某些“奇妙的”事情。有时，分配器维护的存储池在我们的程序之外，如在一个磁盘文件中。于是，在必要的时候，智能指针就会将适当的对象读入到 in-memory 高速缓存中。（写一个具备所有优点的高速缓存分配器是一件有意义的练习，但是我们在此将不对其作讨论，因为这与我们在此讨论的主题相去甚远。）

经年累月下来，对于如何定义及操作各种各样的智能指针，我们这些 C++ 程序员也变得熟练起来。但对于如何在引用上进行类似活动，我们却一直没有获得太多的成功。是的，我们可以为我们所写的类定义自己的一元取址操作符（`operator&()`），但在这样做之前，最好小心行事。写出能够容许特殊引用的代码是一件很困难的事情。简而言之，不论 `p` 是如何的智能化，通过表达式 `&*p` 最好产生一个普通的指针，并且它指向的对象存在于程序的存储空间之内。否则，要想写出在各种情况下都能工作正常的容器模板类几乎是不可能的事情。

对于各种不同的、可供选择的编址方案，分配器可以有着不同的规定，但 C++ 标准对此仍然选择了保守的做法。C++ 标准说：在书写使用分配器的代码时，程序员可以依赖对地址运算的传统定义。更进一步地讲，在假设程序员所提供的分配器都同样普通的前提下，实现可以随意地撰写 STL 容器的代码。例如，`x` 是类型 `T` 的一个左值（lvalue），代码可以假设 `&x` 将产生一个指向 `T` 的指针。同样也可以假设指针是 C 语言风格的指针——也就是说，不存在任何智能指针、代理（proxy）或其他技巧。

相等的分配器

在 C++ 标准中，对分配器还存在着另一种限制。由于分配器是一个对象，所以至少在原则上它可以存储数据。我们可能会考虑利用这一点来定义一个分配器类，使其在 `allocator` 对象的不同组之间维护着不同的存储池。每个分配器对象都存储着一个指针，指向它准备使用的存储池。存储的指针可能对于同一类型的分配器对象有所不同。可能你也知道，对于一组向量（vector）对象，我们可以在其上重复地进行成组元素的交换。利用相同组中的分配器对象来构造所有的向量对象是很有意义的。

如果两个分配器对象相等的话，我们就可以利用其中的一个分配器来分配元素，利用另外一个分配器来释放先前所分配的元素。它们都指向同一个存储池。构造巧妙的容器可以利用这个信息来优化它与另外一

个容器进行的内容交换。用于实现交换的代码知道：只有在两个分配器对象不相等的情况下，它才需要真正复制那些元素。否则，它只需要对一些指针进行重排来完成交换。

但是，让我们再次重申一次，C++标准最终选择的是较为保守的做法。它允许程序员可以假设所有同一类型的分配器对象都是相等的。更特别的是，基于程序员所提供的分配器也同样遵循这个条件的假设，实现就可以随意地撰写 STL 容器的代码。也可以假设一堆同一类型的分配器对象之中没有维护私有的存储池。也就是说，例如：所有的交换都可以很快完成，但我们这些程序员却并不能写出一个可移植的分配器，使之维护上面所描述的多个存储池。

此处所给出的实现比较优雅。容器模板类允许分配器将一个指针定义成某种类的类型。在操作有意义的情况下，它们甚至还会处理如何构造及析构这样的指针。容器还允许两个分配器对象不相等（在这种情况下，它们认为这两个分配器对象管理着不同的存储池）。通过将一个值为 0 的整型常量表达式指定给一个指针对象，代码就认为我们创建了一个空指针。它假定表达式 `&x` 将产生一个普通的指针。然而，即使是在存在着这样的限制条件，我们仍然可以写出一些值得炫耀的分配器，并且在本书中所提供的容器也都能够与这些分配器一同工作并得到正确的结果。但是，请记住：千万不要将这些值得炫耀的代码直接拿到标准 C++ 的任意实现中，不能这样做。

allocator

分配器存在着许多特殊之处。下面列出的是我们可能会在实际应用中使用的方法。如前面所提到的，模板类 `allocator` 是所有 STL 容器都会采用的默认分配器。它同样也为我们自己所定义的分配器提供了一个很好的原型。它定义了所有用来描述如何对已分配对象进行寻址和操作的类型。这些成员类型的定义如下：

```
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;
```

address

成员函数 `address`，不论是常量还是非常量版本，从本质上说都是一元操作符 `operator&`（如 `&x`）的智能版本。它保证了被分配对象的地址可以用适当的指针表达出来。对于 `address(x)` 的调用，模板类 `allocator` 只是简单地返回 `&x`。

大部分其余的成员函数所完成的功能都可以从它们的名字上得到

暗示：

- allocate**
 - `allocate` 用于分配元素类型为 `T` 的数组。
- deallocate**
 - `deallocate` 用于释放由同一个分配器在先前所分配的数组（或者是由与该分配器相等的分配器所分配的数组，我们将在下面详细讨论这个问题）。
- construct**
 - `construct` 用于在指定的原始存储空间 (`raw storage`) 中构造一个类型为 `T` 的对象。
- destroy**
 - `destroy` 在适当的位置析构一个对象，并且不释放其所占的内存空间。
- max_size**
 - `max_size` 告知我们在一个元素类型为 `T` 的数组中最多可以寻址多少个元素。
- rebind**
 - 实际上这些奇妙的功能是由成员模板 `rebind` 完成的：

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

它所做的就是让程序在给定一个类型 `allocator<T>` 时，产生一个类型 `allocator<U>` 的名字。下面我们将讲述它是如何这样做的。假定我们特化一个 `list` 容器（代码如先前所示）。只有在这种情况下，我们需要按照模板类 `allocator` 写类型定义：

```
typedef list<float, allocator<float> > Mylist;
```

这样我们就为模板类 `list` 提供了一个分配 `float` 对象的方法。每个 `list` 对象都包含了一个私有成员对象 `myal`，我们通过它来实现所需的对象分配方法。

现在惟一的问题是，这些并不是 `list` 所关心的。我们需要的是分配包含前向和后向指针的元素，并用它们把整个列表链接起来。在这种特殊的情况下，元素看起来就可能是下面这样：

```
struct Myelement {
    Myelement *next, *prev;
    float item;
};
```

但是 `allocator<float>` 对象只知道如何分配由一个或多个 `float` 对象组成的数组。容器实际上需要的是类 `allocator<Myelement>` 的对象。为了声明这样一个对象，容器必须声明如下：

```
allocator<float>::rebind<Myelement>::other
Myalloc(myal);
```

模板成员结构体 `rebind` 提供了一种对所需要的分配器类型命名的方法。并且模板构造函数也使得：即使这两个对象的类型之间仅仅存在着一定的联系，`Myalloc` 也可以从对象 `myal` 中构造出来。这样做的最终结果就是，给定我们所支持的分配器，容器就可以引出所需的分配器。

临时的分配器

多棒的技巧啊！现在惟一的问题就是：为了实现这个技巧，STL 需要编译器支持成员模板类，而不仅仅是常规的成员模板函数。这种特殊的特性也是最近才添加到商业化的 C++ 编译器中的。为了找到一种合适的中间层用以避免对成员模板类的需求，实现者们面对着一个极大的挑战。不幸的是，在定义了许多不同且不兼容的方法后，实现者又将这个挑战推到了新的高度。

下面展示的就是一种可行的中间层形式，它也用于我们在此处所提及的实现的商业化版本中。它并不很难。当容器需要一个不同于我们所提供的分配器时，它就抄了点近路：

- 按照 C++ 的惯例，假设对于模板类 `allocator`，不同的成员类型都相同。
- 我们假设所提供的分配器对象拥有一个新增的成员函数 `Charalloc(size_type n)`，它可以用来分配大小为 `n` 字节的任意对象。
- 所有通过这种方法分配的对象在随后都可以通过调用我们所提供的成员对象中的成员函数 `deallocate` 来释放。

总而言之，分配器赋予程序员某种弹性，使得他们可以自由地控制容器为其所管理的序列中的元素分配及释放内存空间的方式。在缺乏实践经验的情况下，实际上能获得多大的弹性仍然是一个没有结果的争论。在现阶段，坚持使用默认的分配器不失为一种保守的办法。把它交给库实现者，让他们提供可行的方案。

其他机制

对于所有在头文件 `<allocator>` 中定义的实体，其公共的主题思想就是存储空间的管理。分配器 STL 容器做了这个工作，但在该头文件中还存在其他一些机制。例如，在可能的情况下，某些 STL 算法可能会从使用临时存储空间中获益。于是明显需要存在一种机制，用来分配及释放这样的存储空间，并且确保它所管理的存储空间块不会丢失。C 程序员可能会把这样的服务与早先的、有点结构化的、并且还是值得信赖的 `malloc` 和 `free` 联系起来。

但是 C++ 还有新的需求。一个典型的对象在被分配时必须被构造，在释放前必须被析构。与之相对应的是，通过使用赋值，我们就可以初始化从 C 中继承而来的基础类型的对象，并且这样的对象在释放前不需要任何特殊的准备工作。C++ 中对于存储空间的分配及释放机制的一个重要的附带产物就是，对于讨论中的对象的构造及析构的机制。

uninitialized_copy 例如，模板容器 `vector` 将它所管理的元素序列表示为一个连续的数组（参见第 10 章）。我们可以用另一个序列来初始化它的元素，或者是用一个值复制填充整个序列。为了改变序列的长度，容器必须分配一部分原始存储空间来作为所需长度的数组，然后从已有的存储空间将值复制到新分配的空间中。
uninitialized_fill 所有的这些操作可以表达为下面的简单算法：

```
_fill_n
    template<class InIt, class FwdIt> inline
        FwdIt uninitialized_copy(InIt first, InIt last,
                                  FwdIt x);
    template<class FwdIt, class T> inline
        void uninitialized_fill(FwdIt first, FwdIt last,
                               const T& x);
    template<class FwdIt, class Size, class T> inline
        void uninitialized_fill_n(FwdIt first, Size n,
                                 const T& x);
```

第一个算法将由范围`[first, last)`中的迭代器指定的序列复制到起始处为 `x` 的序列中去。（注意：`InIt` 可以是任何种类的输入迭代器，`FwdIt` 可以是任何种类的前向迭代器。）它通过使用元素的复制构造函数构造出目标序列中的每个元素，由此来完成其功能。

第二个算法将指定的范围视为目标，而不是源。它通过对值 `x` 的复制，构造出由`[first, last)`指定的序列中的每个元素。第三个算法和它差不多，不过它的目标序列起始于 `first` 且元素个数为 `n`。

异常情况下 的安全性

这三个模板函数实现了一个新增的服务。模板的特化中通常都包含大量实现者提供的代码及程序员书写的代码。没有一方可以完全控制最终的产品。但库实现者们仍然要对此承担某些义务。如果在程序员所提供的代码中有异常抛出，那么厂商提供的那部分必须对此有明确的反应。`C++` 标准的一个新增特性就是对于容器的操作有一个一般化的要求：即便在有异常抛出的情况下，容器的行为仍然必须是可预期的。你可能不会总是喜欢这样的结果，但容器必须至少保持在一种可以被正常销毁的状态下。此外，容器还必须保证不会对分配的存储空间失去控制，对已经构造的对象的销毁不会出错。这时，就轮到这三个模板函数发挥作用了。

这三个模板函数都只做了一个简单的承诺。如果在它们执行期间有异常抛出的话，它们将会捕获这个异常，销毁它们所构造出的所有对象，然后再将该异常向外抛出。对于实现 STL 容器的代码来说，这很容易实现。事实上，本书所提供的代码并没有用到这几个特殊的模板函数。（但是可以参见第 10 章来得到一些类似的私有成员函数。）然而，我们所写的容器必须遵循与 STL 容器一样的限制，这样，这些模板函数才可以提供一些便利。

临时缓冲区

我们在前面提到，某些算法会利用到临时的存储空间。与利用一个到两个临时对象不同的是，使用临时的存储空间来（向内及向外）复制元素将会使得排序、合并以及划分这样的操作更加快速。头文件`<memory>`定义了一对模板函数来分配及释放临时缓冲区，以达到上述目的：

```
template<class T> inline
pair<T *, ptrdiff_t>
get_temporary_buffer(ptrdiff_t n);
template<class T> inline
void return_temporary_buffer(T *p);
```

get_temporary_buffer**return_temporary_buffer**

第一个模板函数用来分配存储空间，通过调用`get_temporary_buffer<T>(n)`，我们可以分配到一个类型为 T、最大长度为 n 的数组。如果返回的 pair 对象中第一个成员不是空指针的话，那么它就拥有值 p，p 指向被分配的存储空间的开始处。被分配到的存储空间的实际大小是由 pair 的第二个成员所表示的，它也有可能小于 n。在使用完用这种方法分配到的存储空间后，我们必须通过调用`return_temporary_buffer(p)`来将它归还给系统。

STL 中的算法从未在同一时间内请求超过一个这样的缓冲区。它们同样也能容忍被稍微地修改。如果没有缓冲区可供分配，或是分配到的缓冲区大小小于请求的大小，这些算法也能工作正常，惟一的不同就是它们的执行速度会因此变得慢下来。

那些利用了临时缓冲区的算法面对着一个管理上的问题。在我们第一次将值存储到缓冲区中的元素内时，我们必须构造出该元素来。上面列出的“填充”及“复制”算法有时（但不是一直）会为我们完成这项工作。在大部分情况下，算法会实时地产生元素。它想假装自己正在与另一个迭代器协同工作。

raw_storage_iterator

这就是模板类`raw_storage_iterator`的由来。它是一个输出迭代器，接受严格按照递增的顺序产生的值序列，它不容许有“回退”及“多次存储到同一对象”的情况出现。它的这种特殊要求是因为它用来将一个产生的序列存储到原始存储空间。也就是说，在向前移动的过程中，元素就被构造出来了。

那些使用了临时缓冲区的算法面对着额外的管理问题。一旦在临时缓冲区中的元素被构造好了之后，随后的存储就不能涉及到任何的构造活动，以免产生两次构造这样的严重问题。这使得那些对临时缓冲区进行多重遍历的算法必须十分小心：没有构造的元素要构造出来；而已经构造出来的则只是进行赋值。在释放临时缓冲区前，必须小心翼翼地销毁构造出来的元素。

惠普公司最初的 STL 代码用内联逻辑来做所有的这一切。这样做导

致算法的复杂程度变大，使它们变得难以处理。当簿记工作开始出错时，bug 就将不可避免地产生了。所以我们在此选择了一种更结构化的方法，它将大部分与使用临时缓冲区相关的混乱逻辑封装起来。

Temp_iterator

为此我们引入了模板类 Temp_iterator。它是一个输出迭代器，用以管理一个临时缓冲区，但它比模板类 raw_storage_iterator 更加彻底：

- 构造函数 Temp_iterator<T>(n) 请求一个临时缓冲区，它最多可以容纳 n 个类型为 T 的元素。
- 它的析构函数会自动释放它所获得的临时缓冲区。
- 迭代器跟踪缓冲区中的最高“水位线”。它会根据需要构造元素或给元素赋值。
- 当然，在释放缓冲区前，它会对元素进行必要的销毁。

这还没完。Temp_iterator<T> 的拷贝与源使用同一块存储空间，这使得即使当这样的迭代器是作为函数返回值出现时，簿记工作仍然能够保持正确。该模板类还定义了一些特殊的成员函数，用它们来控制对缓冲区的重新扫描。

- Init() 重新从缓冲区的起始处开始存储。
- First() 返回指向缓冲区起始处的指针。
- Last() 返回一个指针，指向的是该缓冲区的下一个存储元素的地方（即紧接着最后被存储的那个元素的位置）。
- Maxlen() 返回缓冲区的长度。

我们将连同与之协同工作的算法来详细地描述 Temp_iterator 的用法。

我们有足够的理由说，此处所提供的 STL 实现没有直接用到 get_temporary_buffer 以及 return_temporary_buffer。它也没有用到 raw_storage_iterator。Temp_iterator 提供了一种更加结构化的解决方案。

auto_ptr

模板类 auto_ptr 并不存在于最初的 STL 版本中。更确切地说，它是 C++ 中新近才增加的一个特性，把它放在头文件 <memory> 中是因为这是最适合放置它的场所。它被设计成一个专门的工具，用来解决在异常情况下出现的内存泄露问题。

简而言之，auto_ptr<T> 封装了一个指针，它指向一个已分配的、类型为 T 的对象（通常都是通过调用 operator new 来得到这个对象）。

- 假设被存储的指针不为空，那么 auto_ptr<T> 对象就“拥有”它所指向的那个对象。
- 在对其进行赋值或调用复制构造函数时，内部对象的所有权可以转移。在这种情况下，最初的拥有者将会用一个空指针来替换掉它原来存储的那个指针。
- 可以通过调用成员函数 release 来显式地放弃对已有对象的所有权。

权。

- 当 auto_ptr<T>对象被销毁时，它将删除掉它拥有的指针所指向的对象。

为什么我们需要这样的方法呢？假设在程序运行中有异常抛出，对于程序中的动态分配的对象（存储类型为 auto 或 register），异常处理机制会小心翼翼地将它们销毁。但对于那些依靠显式地调用 operator new（如使用 new 表达式）得到的对象，它就显得无能为力了。这些对象的销毁必须依靠显式的 operator delete 调用（如使用 delete 表达式）。模板类 auto_ptr 提供了一种途径，将显式分配的存储空间与动态分配的存储空间联系在一起。它能够确保，不论控制对象的指针以何种方式超出了当前它所处的生存空间，它所控制的对象都会被销毁。

功能描述

```

namespace std {
template<class T>
    class allocator;
template<>
    class allocator<void>;
template<class FwdIt, class T>
    class raw_storage_iterator;
template<class T>
    class auto_ptr;

        // TEMPLATE OPERATORS
template<class T>
    bool operator==(allocator<T>& lhs,
          allocator<T>& rhs);
template<class T>
    bool operator!=(allocator<T>& lhs,
          allocator<T>& rhs);

        // TEMPLATE FUNCTIONS
template<class T>
    pair<T *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t n);
template<class T>
    void return_temporary_buffer(T *p);
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last,
                           FwdIt result);
template<class FwdIt, class T>

```

```

        void uninitialized_fill(FwdIt first, FwdIt last,
                               const T& x);
template<class FwdIt, class Size, class T>
        void uninitialized_fill_n(FwdIt first, Size n,
                               const T& x);
    };

```

如果想自己定义一个类、操作符或某些模板来帮助分配和释放对象的话，请将 STL 的标准头文件<memory>包含到程序里面。

```

# allocator

template<class T>
    class allocator {
        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;
        pointer address(reference x) const;
        const_pointer address(const_reference x) const;
        template<class U>
            struct rebind;
        allocator();
        template<class U>
            allocator(const allocator<U>& x);
        template<class U>
            allocator& operator=(const allocator<U>& x);
        template<class U>
            pointer allocate(size_type n, const U *hint = 0);
            void deallocate(pointer p, size_type n);
            void construct(pointer p, const T& val);
            void destroy(pointer p);
            size_type max_size() const;
    };

```

该模板类描述的对象用于为类型为 T 的对象数组管理存储空间的分配和释放。在标准 C++ 库中，有些容器模板类中的默认 allocator 对象其实就是类 allocator 的对象。

模板类 allocator 提供了一些更为通俗的类型定义。这些定义似乎没有任何价值。但我们可能对使用一个具有相同成员的类将其替换掉非常感兴趣。如果我们以这样的分配器来创建一个容器，那么容器中对象的分配及释放操作就可以单独控制。

例如，一个分配器对象可以从一个私有堆中分配存储空间；或者从一个远堆中分配到存储空间（此时我们需要使用非标准指针来存取被分配的对象）；或者通过自身所提供的类型定义来指定一些特殊的管理共享内存的存取器（accessor）对象来存取这些元素，或是用这个分配器实现自动垃圾收集（garbage collection）。因此，那些使用分配器对象来分配存储空间的类必须严谨地使用这些类型来声明指针和引用对象。（标准 C++ 库中的容器也是如此。）

也就是说，一个 `allocator` 定义了下面这些类型：

- `pointer` —— 行为像指向 `T` 的指针。
- `const_pointer` —— 行为像指向 `T` 的常量指针。
- `reference` —— 行为像 `T` 的引用。
- `const_reference` —— 行为像 `T` 的常量引用。

这些类型指定了那些作用于元素身上的指针及引用的形式。（虽然我们在类 `allocator` 中明确地把 `allocator::pointer` 定义为 `T*`，但它不必对所有分配器对象都等同于 `T*`。）

口 `allocator::address`

```
pointer address(reference x) const;
const_pointer address(const_reference x) const;
```

该成员函数以指针的形式返回所分配元素 `x` 的地址。

口 `allocator::allocate`

```
template<class U>
pointer allocate(size_type n, const U *hint = 0);
```

该成员函数通过调用 `operator new(n)`，为一个长度为 `n`、元素类型为 `T` 的数组分配存储空间。它返回一个指向被分配对象的指针。参数 `hint` 可以帮助某些分配器提高引用的局部性（locality），一种可行的做法就是使用先前由同一个分配器所分配的、还没有被释放的对象的地址。如果不想提供任何 `hint` 的话，只需用一个空指针替换它即可。

口 `allocator::allocator`

```
allocator();
template<class U>
allocator(const allocator<U>& x);
```

该构造函数什么也没有做。然而，通常当一个分配器对象构造自另一个分配器对象时，它应该和这个对象相等（这也允许我们混用这两个分配器对象的分配及释放）。

口 `allocator::const_pointer`

```
typedef const T *pointer;
```

该指针类型描述了这样的一个对象 `p`，通过使用表达式`*p`，我们可

以用它来指定模板类 allocator 的对象可以分配的任意常量对象。

口 allocator::const_reference

```
typedef const T& const_reference;
```

该引用类型描述了这样的一个对象 x，它可以指定模板类 allocator 的对象可以分配的任意常量对象。

口 allocator::construct

```
void construct(pointer p, const T& val);
```

该成员函数通过执行 placement (定位放置) new 表达式 new((void*)p)T(val)，在指针 p 所指向的存储空间中构造出一个类型为 T 的对象。

口 allocator::deallocate

```
void deallocate(pointer p, size_type n);
```

该成员函数通过调用 operator delete(p)，释放开始于 p、类型为 T、长度为 n 的数组所处的存储空间。其中指针 p 必须是先前由等同于*this 的分配器对象通过调用 allocate 成员函数所得到的返回值，并且那次所分配的数组与此次所释放的数组有着相同的类型和长度。deallocate 不会抛出任何异常。

口 allocator::destroy

```
void destroy(pointer p);
```

该成员函数通过调用析构函数 p->T::~T()，销毁一个由 p 指定的对象。

口 allocator::difference_type

```
typedef ptrdiff_t difference_type;
```

该有符号整数类型描述了模板类 allocator 的对象可以分配的序列中两个元素地址之间的差距。

口 allocator::max_size

```
size_type max_size() const;
```

该成员函数返回由类 allocator 的对象分配的、类型为 T 的序列所能达到的最大长度。

口 allocator::operator=

```
template<class U>
allocator& operator=(const allocator<U>& x);
```

该模板赋值操作符什么也没有做。然而，通常将一个分配器对象赋值给另一个分配器时，它们必须相等（这也就允许我们混用这两个分配器对象的分配及释放）。

```
# allocator::pointer
    typedef T *pointer;
```

该指针类型描述了这样的一个对象 p，通过使用表达式*p，我们可以用它来指定模板类 allocator 可以分配的任意对象。

```
# allocator::rebind
    template<class U>
    struct rebind {
        typedef allocator<U> other;
    };
```

该成员模板类定义类型 other。它惟一的作用就是为给定的类型名字 allocator<T>提供另一个类型名字 allocator<U>。

例如，给定一个分配器对象 al，它的类型为 A。可以使用如下的表达式来分配一个类型为 U 的对象：

```
A::rebind<U>::other(al).allocate(1, (U *)0)
```

或者通过这样写来简单地命名它的指针类型：

```
A::rebind<U>::other::pointer
```

```
# allocator::reference
    typedef T& reference;
```

该引用类型描述了这样一个对象 x，它可以指定模板类 allocator 的对象可以分配的任意对象。

```
# allocator::size_type
    typedef size_t size_type;
```

该无符号整数类型描述的对象可以用来表示可由模板类 allocator 的对象分配的任意序列的长度。

```
# allocator::value_type
    typedef T value_type;
```

该类型是模板参数 T 的同义词。

```
# allocator<void>
    template<>
    class allocator<void> {
        typedef void *pointer;
        typedef const void *const_pointer;
        typedef void value_type;
        template<class U>
            struct rebind;
        allocator();
        template<class U>
```

```

    allocator(const allocator<U>);

template<class U>
    allocator<void>& operator=(const allocator<U>);
};


```

该类显式地为类型 void 特化模板类 allocator。它的各个构造函数与赋值操作符的行为都和模板类的相同，但它只定义了类型 const_pointer、pointer、value_type 以及嵌套的模板类 rebinder。

4 auto_ptr

```

template<class T>
class auto_ptr {
template<U>
    struct auto_ptr_ref;
T *q;
public:
    typedef T element_type;
explicit auto_ptr(T *p = 0) throw();
auto_ptr(auto_ptr<T>& rhs) throw();
template<class U>
    auto_ptr(auto_ptr<U>& rhs) throw();
auto_ptr(auto_ptr_ref<T> rhs) throw();
~auto_ptr();
template<class U>
    operator auto_ptr<U>() throw();
template<class U>
    operator auto_ptr_ref<U>() throw();
template<class U>
auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
auto_ptr<T>& operator=(auto_ptr<T>& rhs) throw();
T& operator*() const throw();
T *operator->() const throw();
T *get() const throw();
T *release() const throw();
void reset(T *p = 0);
};

```

该类描述的对象中存储着一个指针（我们把它叫做 q），它指向一个已分配的、类型为 T 的对象。这个指针要么为空，要么指向一个由 new 表达式分配的对象。以一个非空指针构造出来的对象拥有该指针。当它所存储的值被赋给另一个对象时，所有权也同时发生了传递。（在传递之后，它所存储的指针将会被一个空指针代替。）如果 auto_ptr<T> 的析构函数拥有一个被分配的对象，它会负责将其删除。因此，类 auto_ptr<T> 的对象就确保了被分配的对象在离开其控制范围时会自动删除，甚至在

出现异常抛出的情况下也能如此。我们不可能构造出两个 `auto_ptr<T>` 对象，使得它们拥有同一个对象。

我们可以将一个 `auto_ptr<T>` 对象以传值参数的方式传递给一个函数。同样，我们也可以以传值的方式返回一个这样的对象。（这两个操作都依赖于类 `auto_ptr<T>::auto_ptr_ref<U>` 中介对象通过不同的转换方式所进行的隐式构造过程。）然而，我们不能用 STL 容器来可靠地管理由 `auto_ptr<T>` 对象组成的序列。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            explicit auto_ptr(T *p = 0) throw();
            auto_ptr(auto_ptr<T>& rhs) throw();
            auto_ptr(auto_ptr_ref<T> rhs) throw();
            template<class U>
                auto_ptr(auto_ptr<U>& rhs) throw();
    };
}
```

第一个构造函数把 `p` 作为指向被分配对象的指针保存起来。第二个构造函数通过在被构造的对象中存储 `rhs.release()`，将 `rhs` 存储的指针的所有权传递过来。除了存储的是 `rhs.ref.release()`（`ref` 是存储在 `rhs` 中的一个引用）外，第三个构造函数与第二个构造函数的行为一样。

模板构造函数的行为也和第二个构造函数的一样，它所提供的指向类型为 `U` 的对象的指针可以隐式转化为指向类型为 `T` 的对象的指针。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            template<class U>
                struct auto_ptr_ref {
                    auto_ptr_ref(auto_ptr<U>& rhs);
                };
    };
}
```

该成员类描述的对象存储了类 `auto_ptr<T>` 的对象的引用。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            ~auto_ptr();
    };
}
```

该析构函数执行表达式 `delete q`。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            typedef T element_type;
    };
}
```

该类型是模板参数 `T` 的同义词。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            T *get() const throw();
    };
}
```

该成员函数返回被存储的指针。

```
#include <memory>
namespace std {
    class auto_ptr {
        public:
            template<class U>
                auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
    };
}
```

```
auto_ptr<T>& operator=(auto_ptr<T>& rhs) throw();
```

只有在赋值使得存储的指针 q 的值改变的情况下，该赋值操作才会执行表达式 delete q。然后通过在*this 中存储 rhs.release()，把存储于 rhs 中的指针的所有权传递过来。该函数返回*this。

■ auto_ptr::operator*

```
T& operator*() const throw();
```

该间接操作符返回*get()。因此，存储的指针不能为空。

■ auto_ptr::operator->

```
T *operator->() const throw();
```

该选择操作符返回 get()，因此，如果 al 是类 auto_ptr<T>的一个对象，表达式 al->m 的行为就和(al.get())->m 的一样。因此，存储的指针不能为空，而且 T 必须为一个拥有成员 m 的类、结构体或联合。

■ auto_ptr::operator auto_ptr<U>

```
template<class U>
operator auto_ptr<U>() throw();
```

该类型转换操作符返回 auto_ptr<U>(*this)。

■ auto_ptr::operator auto_ptr_ref<U>

```
template<class U>
operator auto_ptr_ref<U>() throw();
```

该类型转换操作符返回 auto_ptr_ref<U>(*this)。

■ auto_ptr::release

```
T *release() throw();
```

该成员函数以一个空指针替换以前所存储的指针，并返回原来所存储的指针。

■ auto_ptr::reset

```
void reset(T *p = 0);
```

只有在函数调用使得存储的指针 q 的值发生改变的情况下，该成员函数才会执行表达式 delete q。然后以 p 来替换以前所存储的指针。

■ get_temporary_buffer

```
template<class T>
pair<T *, ptrdiff_t>
get_temporary_buffer(ptrdiff_t n);
```

该模板函数会从一个未指定的源（可以是 operator new 所操作的标准堆）中为一个序列分配一块存储空间，这个序列最多可以有 n 个类型为 T 的元素。它返回类型为 pair<T*, ptrdiff_t>的值 pr。如果函数分配了存储空间，那么 pr.first 指向被分配的空间，pr.second 则代表这块空间最

多可以容纳的元素的个数。否则，`pr.first` 为一个空指针。

口 `operator!=`

```
template<class T>
bool operator!=(allocator<T>& lhs,
                  allocator<T>& rhs);
```

该模板操作符返回 `false`。

口 `operator==`

```
template<class T>
bool operator==(allocator<T>& lhs,
                  allocator<T>& rhs);
```

该模板操作符返回 `true`。（只有在由一个分配器分配的对象可以由另一个分配器释放的情况下，这两个分配器对象才应该相等。如果其中一个对象的值是通过赋值或者构造函数从另一个对象中得来的话，这两个对象也应该相等。）

口 `raw_storage_iterator`

```
template<class FwdIt, class T>
class raw_storage_iterator
    : public iterator<output_iterator_tag,
                      void, void, void, void> {
public:
    typedef FwdIt iter_type;
    typedef T element_type;
    explicit raw_storage_iterator(FwdIt it);
    raw_storage_iterator<FwdIt, T>& operator*();
    raw_storage_iterator<FwdIt, T>&
        operator=(const T& val);
    raw_storage_iterator<FwdIt, T>& operator++();
    raw_storage_iterator<FwdIt, T> operator++(int);
};
```

该类描述了一个输出迭代器，它会为它产生的序列构造类型为 `T` 的对象。类 `raw_storage_iterator<FwdIt, T>` 的对象会通过一个类 `FwdIt` 的前向迭代器对象存取存储空间，我们可以在构造对象时指定这个 `FwdIt`。对于类 `FwdIt` 的对象 `it` 来说，表达式 `&*it` 必须指向一块未构造的存储空间，以备产生序列中的下一个对象（其类型为 `T`）使用。

口 `raw_storage_iterator::element_type`

```
typedef T element_type;
```

该类型是模板参数 `T` 的同义词。

- `raw_storage_iterator::iter_type`
`typedef FwdIt iter_type;`
该类型是模板参数 `FwdIt` 的同义词。
- `raw_storage_iterator::operator*`
`raw_storage_iterator<FwdIt, T>& operator*();`
该间接操作符返回`*this`（这样就使得 `operator=(const T&)`可以在诸如`*x = val`这样的表达式中实现实际上的存储）。
- `raw_storage_iterator::operator=`
`raw_storage_iterator<FwdIt, T>& operator=(const T& val);`
该赋值操作符通过执行定位放置 `new` 表达式 `new ((void*)&*it) T(val)`，可以使用已有的迭代器值 `it` 来构造出输出序列中的下一个对象。
该函数返回`*this`。
- `raw_storage_iterator::operator++`
`raw_storage_iterator<FwdIt, T>& operator++();`
`raw_storage_iterator<FwdIt, T> operator++(int);`
第一个（前递增）操作符先把所存储的输出迭代器对象递增，然后返回`*this`。
第二个（后递增）操作符先复制一份`*this`，然后把所存储的输出迭代器对象递增，最后返回那份拷贝。
- `raw_storage_iterator::raw_storage_iterator`
`explicit raw_storage_iterator(FwdIt it);`
该构造函数把 `it` 作为输出迭代器对象存储起来。
- `return_temporary_buffer`
`template<class T>`
`void return_temporary_buffer(T *p);`
该模板函数将会释放掉 `p` 所指向的存储空间，该空间是由先前调用 `get_temporary_buffer` 所分配的。
- `uninitialized_copy`
`template<class InIt, class FwdIt>`
`FwdIt uninitialized_copy(InIt first, InIt last,`
`FwdIt result);`
该模板函数实际上执行的是：

```
while (first != last)
    new ((void *)&*result++) U(*first++);
return first;
```

其中 U 为 iterator_traits<InIt>::value_type，除非在执行时代码抛出异常。在这种情况下，所有构造好的对象都将被销毁，并且继续将异常向外抛出。

¶ uninitialized_fill

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

该模板函数实际上执行的是：

```
while (first != last)
    new ((void *)*first++) U(x);
```

其中 U 为 iterator_traits<FwdIt>::value_type，除非在执行时代码抛出异常。在这种情况下，所有构造好的对象都将被销毁，并且继续将异常向外抛出。

¶ uninitialized_fill_n

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

该模板函数实际上执行的是：

```
while (0 < n--)
    new ((void *)*first++) U(x);
```

其中 U 为 iterator_traits<FwdIt>::value_type，除非在执行时代码抛出异常。在这种情况下，所有构造好的对象都将被销毁，并且继续将异常向外抛出。

使用<memory>

如果在翻译单元中包含任意的 STL 头文件，那么<memory>就很可能同时被包含。只有在需要使用该头文件中的定义并且不指望通过其他 STL 头文件将这些定义带入翻译单元时，才有必要在程序中显式地包含它。但是“将会”需要使用该头文件中所定义的实体的可能性同样也很大。如果在翻译过程中我们得到了这样的诊断信息：uninitialized_fill 未定义，请记住：最简单的解决方案就是将这个头文件包含到程序中。

下面是对<memory>中的各种定义的简单介绍：

allocator

在特化一个容器模板类时，我们通常会把模板类 allocator 作为它的分配器类型，如：

```
typedef list<char, allocator<char>> Mylist;
```

然而，通过使用默认的模板参数，我们至少可以避免这样提及分配

器。STL 中所有的容器构造函数都提供了一个默认的分配器对象，这样我们就可以根本不用提到分配器。

如果需要直接使用 allocator 对象，或是写一个自己的分配器，那么请仔细学习容器模板类是如何使用分配器的（将在本书的后面部分涉及到）。

uninitialized_copy 模板函数 uninitialized_copy(first, last, x) 将由 [first, last) 指定范围的序列中的元素复制到以 x 开始的、未被构造的序列中去。生成序列中的每个元素都是通过一个复制构造函数进行初始化的。如果在这个模板函数的执行过程中有异常抛出的话，所有构造出来的对象都会被销毁，并且该异常也会继续向外抛出。

uninitialized_fill 另一方面，模板函数 uninitialized_fill(first, last, x) 以一个单独的元素 x 为源，向由 [first, last) 指定的未构造的区间内复制对象。这个生成序列中的每个元素也都是由一个复制构造函数构造出来的。模板函数

uninitialized_fill_n uninitialized_fill_n(first, n, x) 和它差不多，只是那个未构造的序列是一个以 first 开始且包含了 n 个元素的序列。如果这两个模板函数中的任意一个在执行过程中抛出异常，所有构造出来的对象都将会被销毁，并且该异常也会继续向外抛出。

get_temporary_buffer 模板函数 get_temporary_buffer<T>(n) 会为一个类型为 T、长度最大为 n 的数组分配原始的存储空间。如果在返回值 x 中 x.first 为一个空指针的话，那么该函数就没有分配到任何存储空间；否则，x.second 中存储的就是实际上分配的元素的个数，它永远也不可能大于 n。

return_temporary_buffer 对于由表达式 x = get_temporary_buffer 所分配的存储空间，可通过调用 return_temporary_buffer(x.first) 将其归还给系统。一个实现没有必要在同一时间内提供超过一个的临时缓冲区。

raw_storage_iterator 模板类 raw_storage_iterator<FwdIt, T> 定义了一个输出迭代器，我们用它来构造类型为 T 的元素并通过一个类型为 FwdIt 的迭代器将这些元素存储起来。可以使用如下的声明来构造一个这样的输出迭代器：

```
raw_storage_iterator<FwdIt, T> x(iter);
```

其中 iter 是一个类型为 FwdIt 的对象，指向我们将要向其中存储元素的那个未构造序列的第一个元素。

auto_ptr 模板类 auto_ptr<T> 为类型为 T 的对象定义了一个“智能指针”。我们可以通过如下的声明来构造一个这样的指针：

```
auto_ptr<T> p(new T);
```

随后，*p 指定那个新分配的对象，p->m 指定该对象的成员 m（如果存在这样一个成员的话）。当 p 被销毁时，它所关联的那个类型为 T 的对象同样也会被销毁。

我们可以通过对该智能指针进行赋值或者复制，传递它所管理的对象的“所有权”（即负责销毁它的职责）。

```
auto_ptr<T> q = p;
```

在这种情况下，`p` 随后将存储一个空指针。我们也可以通过调用 `p.release()` 简单地放弃这个所有权，此时假设我们有销毁已分配的对象的职责。

请对下面两种情况保持高度的警惕：“将类 `auto_ptr<T>` 的对象作为参数传递给一个函数”及“函数返回一个这样的对象”。要想明了这些所有权的问题是一件很困难的事情。请不要尝试在容器元素中存储这样的对象。容器不能够确保以一种可以预见的方法来复制它们以保证所有权的正确拥有。

实现<memory>

本书提供的实现将一部分名义上在头文件 `<memory>` 中定义的实用工具移到了其他头文件中。新增的头文件 `<xmemory>`（不是 C++ 标准中所指定的头文件）包含一系列定义，它们贯穿使用于一个典型的标准 C++ 库中，甚至在没有显式包含 `<memory>` 的程序中也可以使用它们。请回想一下，在前面的章节中，我们曾经讨论过为什么一部分通常应该在头文件 `<iterator>` 中定义的实用工具实际上却出现在新增的头文件 `<xutility>` 中。

`xmemory`

程序清单 4-1 列出了文件 `xmemory`。

```
程序清单 4-1: // xmemory internal header (from <memory>)
#ifndef XMEMORY_
#define XMEMORY_
#include <new>
#include <xutility>

#ifndef FARQ      /* specify standard memory model */
#define FARQ
#define PDFT ptrdiff_t
#define SIZT size_t
#endif

namespace std {
    // TEMPLATE FUNCTION Allocate
    template<class T> inline
```

```
T FARQ *Allocate(SIZT N, T FARQ *)
{return ((T FARQ *)operator new(N * sizeof (T))); }

// TEMPLATE FUNCTION Construct
template<class T1, class T2> inline
void Construct(T1 FARQ *P, const T2& V)
{new ((void FARQ *)P) T1(V); }

// TEMPLATE FUNCTION Destroy
template<class T> inline
void Destroy(T FARQ *P)
{P->~T(); }
template<> inline void Destroy(char FARQ *P)
{}
template<>inline void Destroy(wchar FARQ *P)
{}


// TEMPLATE CLASS allocator
template<class T>
class allocator {
public:
    typedef SIZT size_type;
    typedef PDFT difference_type;
    typedef T FARQ *pointer;
    typedef const T FARQ *const_pointer;
    typedef T FARQ& reference;
    typedef const T FARQ& const_reference;
    typedef T value_type;
    template<class U>
        struct rebind {
            typedef allocator<U> other;
        };
    pointer address(reference X) const
        {return (&X); }
    const_pointer address(const_reference X) const
        {return (&X); }
    allocator()
        {}
    allocator(const allocator<T>&)
        {}
    template<class U>
```

```

        allocator(const allocator<U>&)
        {}
template<class U>
allocator<T>& operator=(const allocator<U>&)
{return (*this); }
template<class U>
pointer allocate(size_type N, const U *)
{return (Allocate(N, (pointer)0)); }
pointer allocate(size_type N)
{return (Allocate(N, (pointer)0)); }
void deallocate(pointer P, size_type)
{operator delete(P); }
void construct(pointer P, const T& V)
{Construct(P, V); }
void destroy(pointer P)
{Destroy(P); }
SIZT max_size() const
{SIZT N = (SIZT)(-1) / sizeof (T);
return (0 < N ? N : 1); }
};

// allocator TEMPLATE OPERATORS
template<class T, class U> inline
bool operator==(const allocator<T>&, const allocator<U>&)
{return (true); }
template<class T, class U> inline
bool operator!=(const allocator<T>&, const allocator<U>&)
{return (false); }

// CLASS allocator<void>
template<> class allocator<void> {
public:
typedef void T;
typedef T FARQ *pointer;
typedef const T FARQ *const_pointer;
typedef T value_type;
template<class U>
struct rebind {
    typedef allocator<U> other;
};
}

```

```

allocator()
{}
allocator(const allocator<T>&)
{}
template<class U>
allocator(const allocator<U>&)
{}
template<class U>
allocator<T>& operator=(const allocator<U>&)
{return (*this); }
};

} /* namespace std */
#endif /* XMEMORY_ */

```

在开始处使用了一点小技巧，靠近文件的顶端是一小组宏定义：

FARQ

```

#ifndef FARQ
#define FARQ
#define PDFT ptrdiff_t
#define SIZT size_t
#endif

```

定义这些宏是为了帮助产生一个远堆分配器。要想使用这样的分配器，只需简单地将常见的头文件包含指令：

```
#include <memory>
```

改变为：

```

#define FARQ far
#define PDFT long
#define SIZT unsigned long
#include <memory>

```

这样得到的模板类 allocator 就会为我们管理远堆。（对于一个给定的实现，我们可能会对上述代码进行一些小小的改动。）由于近年来 PC 上的存储空间越来越大，所以这种古怪的需求也就越来越小了。但仍有大量的代码使用远指针。一个远堆分配器就是一个仍具有潜在作用的备用的简单例子。

**Allocate
Construct
Destroy**

模板函数 Allocate、Construct 和 Destroy 所做的事情都可以从它们的名字上面得到提示。前两个模板函数以这种形式出现是因为在 STL 头文件中，它们被用在不止一处。最后那个函数解决了一个让人讨厌的问题。当被要求产生一个标量（scalar）类型（如 char 或 wchar_t）的析构

函数时，许多现有的编译器都会给出警告信息。我们在此给出这两种类型的显式特化是因为：由于库的模板特化机制，几乎每个使用完整的标准 C++ 库的程序都会试图去销毁它们。在 STL 的早期版本中，我们为这些标量类型提供了析构函数。但对于你能写出的所有指针类型，这个列表是没有尽头的，因而我们在此只给出了其中的两种。

allocator

然而，模板类 allocator 才是我们在此关注的焦点。看上去它好像是一个很简单的模板类，它所展现的实现方面的问题却能引起我们广泛的讨论。然而，你在此不能看到的是，该模板类的一个重要部分要求在给定现有翻译器的情况下有可替换的实现技术。你在此可以看到的是一种渐近的形式，目前的实现也正在向这种形式靠拢。

allocator<void>

在模板定义后面是对类 allocator<void> 的显式特化。在需要以适当的风格声明泛型指针时，写出这样的类型是具有实际意义的。但对于 void 来说，大多数成员类型与函数都没有意义。该显式特化也只提供了实际上有意义的那些成员。

与模板类 allocator 一同出现的还有两个模板操作符。第一个可以让我们比较任意两个 allocator<T> 对象的相等性而不管它们的模板参数，并且这两个对象总是相等。第二个实际上只是为了一致性而简单定义了不等操作符。正如我们前面讨论的一样，分配器对象之间的相等性有着特殊的意义。如果两个这样的对象相等的话，它们就可能同时工作于一个存储池中。用其中一个分配器对象分配的对象也就可以被另一个释放。在本实现中，该模板容器会在不同的容器对象间交替所分配的元素前检测分配器对象的相等性。

我们有充分的理由以一种一般化的认识来谈论 allocator。没有任何事情可以阻止你定义一个全新的类并把它作为一个分配器使用。甚至不需要使这个类派生自模板类 allocator。然而，你却必须提供 allocator 所提供的所有功能。否则，编译器将会为此给出错误信息或产生错误的代码。

这些功能的一部分包括成员模板类 rebinding。按照它本身的方式将一个分配器写成一个模板类会有一些限制。不过，对于如何撰写这些代码，你还是有着相当的自由度的。

memory

程序清单 4-2 中列出了文件 memory。它定义了剩下的那些与头文件 <memory> 相关但又未在 xmemory 中定义的实体。

```
程序清单 4-2:      // memory standard header
memory          #ifndef MEMORY_
                  #define MEMORY_
                  #include <iterator>
                  #include <xmemory>
```

```
namespace std {
    // TEMPLATE FUNCTION get_temporary_buffer
    template<class T> inline
        pair<T FARQ *, PDFT>
            get_temporary_buffer(PDFT N)
    {T FARQ *P;
    for (P = 0; 0 < N; N /= 2)
        if ((P = (T FARQ *)operator new(
            (SIZT)N * sizeof (T), nothrow)) != 0)
            break;
    return (pair<T FARQ *, PDFT>(P, N)); }

    // TEMPLATE FUNCTION return_temporary_buffer
    template<class T> inline
        void return_temporary_buffer(T *P)
    {operator delete(P); }

    // TEMPLATE FUNCTION uninitialized_copy
    template<class InIt, class FwdIt> inline
        FwdIt uninitialized_copy(InIt F, InIt L, FwdIt X)
    {return (Uninit_copy(F, L, X, Val_type(F))); }
    template<class InIt, class FwdIt, class T> inline
        FwdIt Uninit_copy(InIt F, InIt L, FwdIt X, T *)
    {FwdIt Xs = X;
    try {
        for (; F != L; ++X, ++F)
            Construct(&*X, T(*F));
        } catch (...) {
        for (; Xs != X; ++Xs)
            Destroy(&*Xs);
        throw;
    }
    return (X); }

    // TEMPLATE FUNCTION uninitialized_fill
    template<class FwdIt, class Tval> inline
        void uninitialized_fill(FwdIt F, FwdIt L, const Tval& V)
    {Uninit_fill(F, L, V, Val_type(F)); }
    template<class FwdIt, class Tval, class T> inline
        void Uninit_fill(FwdIt F, FwdIt L, const Tval& V, T *)
    {FwdIt Fs = F;
```

```

        try {
            for (; F != L; ++F)
                Construct(&*F, T(V));
        } catch (...) {
            for (; Fs != F; ++Fs)
                Destroy(&*Fs);
            throw;
        }

        // TEMPLATE FUNCTION uninitialized_fill_n
template<class FwdIt, class S, class Tval> inline
void uninitialized_fill_n(FwdIt F, S N, const Tval& V)
{Uninit_fill_n(F, N, V, Val_type(F)); }
template<class FwdIt, class S, class Tval, class T> inline
void Uninit_fill_n(FwdIt F, S N, const Tval& V,
                    T *)
{FwdIt Fs = F;
try {
    for (; 0 < N; --N, ++F)
        Construct(&*F, T(V));
} catch (...) {
    for (; Fs != F; ++Fs)
        Destroy(&*Fs);
    throw;
}

        // TEMPLATE CLASS raw_storage_iterator
template<class OutIt, class T>
class raw_storage_iterator
    : public OutIt {
public:
    typedef OutIt iter_type;
    typedef T element_type;
    explicit raw_storage_iterator(OutIt X)
        : Next(X) {}
    raw_storage_iterator<OutIt, T>& operator*()
        {return (*this); }
    raw_storage_iterator<OutIt, T>& operator=(const T& X)
        {Construct(&*Next, X);
        return (*this); }
    raw_storage_iterator<OutIt, T>& operator++()

```

```
        {++Next;
         return (*this); }
     raw_storage_iterator<OutIt, T> operator-+(int)
     {raw_storage_iterator<OutIt, T> Ans = *this;
      ++Next;
      return (Ans); }
private:
    OutIt Next;
};

// TEMPLATE CLASS Temp_iterator
template<class T>
class Temp_iterator
    : public Outit {
public:
    typedef T FARQ *Pty;
    Temp_iterator(PDFT N = 0)
        {pair<Pty, PDFT> Pair =
         get_temporary_buffer<T>(N);
          Buf.Begin = Pair.first;
          Buf.Cur = Pair.first;
          Buf.Hiwater = Pair.first;
          Buf.Len = Pair.second;
          Pb = &Buf; }
    Temp_iterator(const Temp_iterator<T>& X)
        {Buf.Begin = 0;
         Buf.Cur = 0;
         Buf.Hiwater = 0;
         Buf.Len = 0;
         *this = X; }
    ~Temp_iterator()
        {if (Buf.Begin != 0)
         (for (Pty F = Buf.Begin;
               F != Buf.Hiwater; ++F)
             Destroy(&*F);
          return_temporary_buffer(Buf.Begin); )}
    Temp_iterator<T>& operator=(const Temp_iterator<T>& X)
        {Fb = X.Fb;
         return (*this); }
    Temp_iterator<T>& operator=(const T& V)
        {if (Pb->Cur < Pb->Hiwater)
```

```
    *Pb->Cur++ = V;
else
    (Construct(&*Pb->Cur, V);
Pb->Hiwater = ++Pb->Cur; }
return (*this); }

Temp_iterator<T>& operator*()
{return (*this); }

Temp_iterator<T>& operator++()
{return (*this); }

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

Temp_iterator<T>& Init()
{Pb->Cur = Pb->Begin;
return (*this); }

Pty First() const
{return (Pb->Begin); }

Pty Last() const
{return (Pb->Cur); }

PDFT Maxlen() const
{return (Pb->Len); }

private:
    struct Bufpar {
        Pty Begin;
        Pty Cur;
        Pty Hiwater;
        PDFT Len;
    } Buf, *Pb;
};

// TEMPLATE CLASS auto_ptr
template<class T>
class auto_ptr {

    // TEMPLATE CLASS auto_ptr_ref
    template<class U>
    struct auto_ptr_ref {
        auto_ptr_ref(auto_ptr<U>& Y)
            : Ref(Y) {}
        auto_ptr<U>& Ref;
    };
public:
```

```
typedef T element_type;
explicit auto_ptr(T *P = 0) throw ()
    : Ptr(P) {}
auto_ptr(auto_ptr<T>& Y) throw ()
    : Ptr(Y.release()) {}
auto_ptr(auto_ptr_ref<T> Y) throw ()
    : Ptr(Y.Ref.release()) {}
template<class U>
operator auto_ptr<U>() throw ()
{return (auto_ptr<U>(*this)); }
template<class U>
operator auto_ptr_ref<U>() throw ()
{return (auto_ptr_ref<U>(*this)); }
template<class U>
auto_ptr<T>& operator=(auto_ptr<U>& Y) throw ()
{reset(Y.release());
return (*this); }
template<class U>
auto_ptr(auto_ptr<U>& Y) throw ()
: Ptr(Y.release()) {}
auto_ptr<T>& operator=(auto_ptr<T>& Y) throw ()
{reset(Y.release());
return (*this); }
~auto_ptr()
{delete Ptr; }
T& operator*() const throw ()
{return (*get()); }
T *operator->() const throw ()
{return (get()); }
T *get() const throw ()
{return (Ptr); }
T *release() throw ()
{T *Tmp = Ptr;
Ptr = 0;
return (Tmp); }
void reset(T* P = 0)
{if (P != Ptr)
    delete Ptr;
Ptr = P; }
private:
T *Ptr;
```

```

    };
} /* namespace std */
#endif /* MEMORY_ */

```

`get_temporary_buffer`

`return_temporary_buffer`

`nothrow`

`uninitialized_copy`
`uninitialized_fill`
`uninitialized_fill_n`

异常情况下
的安全性

函数调用 `get_temporary_buffer<T>(n)` 尽量为有几个类型为 `T` 的元素的数组提供原始存储空间。只要分配空间的请求失败，它就会不断地将所要求的空间大小减半。（对于那些为其设计了 `get_temporary_buffer` 的算法，无论获得什么样的临时存储空间，它们都很满意。）模板函数 `return_temporary_buffer` 会向系统归还以此方式分配到的任意空间。顺便说一句，请注意：不可以简单地调用 `get_temporary_buffer(n)`，因为它没有提供足够的信息来确定模板参数。所以，在上述调用中，紧接着函数名显式地列出模板参数。由于这种表达方式是最近才添加进 C++ 标准中的，所以，到目前为止并不是所有的编译器都支持它。对于那些不支持它的编译器，我们可以把函数调用的方式改为：`get_temporary_buffer(n, (T*)0)`。虽然这并不符合 C++ 标准，但是至少是有用的。

本实现中用到了 `operator new` 的一种特殊形式，它也是在 C++ 标准化进程中引入的一个部分。当无法分配一个 `N` 字节的对象时，调用 `operator new(N, nothrow)` 将会返回一个空指针。（`nothrow` 和 `operator new` 的这个重载版本在标准头文件 `<new>` 中定义。）由于 `get_temporary_buffer` 允许被分配的空间大小小于所要求分配的空间大小，甚至可以什么都不分配，那么，在检测可用临时缓冲区空间大小时抛出一个异常是一种不好的方式。相反，`operator new(N)` 在无法分配到所需大小的空间时，会强制抛出一个异常。

模板函数 `uninitialized_copy`、`uninitialized_fill` 和 `uninitialized_fill_n` 显然十分简单，最起码在根据上面所描述的模板函数 `Construct` 进行定义时是如此。然而，有两方面的考虑使得它们实现起来有点混乱。首先，所有这三个模板函数都必须知道与迭代器参数相关的值类型。在模板函数中，这是一个很普遍的要求，在此你可以看到一种能够达到这种要求的通常做法（最起码适合于那些不依赖于模板的部分特化来使 `iterator_traits` 正确工作的实现）。这三个模板函数都会调用一个多了一个参数的辅助函数。模板函数 `Val_type` 会为这个辅助函数提供一个哑参数，用以告诉它想要的类型信息。一个足够智能的编译器会消除这个哑参数的实际传递。更智能化的编译器甚至会内联这个辅助函数，并消除额外的函数调用。（STL 的大部分都是基于以下的期望所写成的：这样的优化在将来的 C++ 编译器中会越来越普遍。）

第二个复杂的因素就是：在有异常抛出的情况下函数的行为必须是可预期的。每个函数都是在一个 `try-catch` 块中实现它的功能，以此来知

道它的完成程度。如果程序员提供的构造函数抛出了一个异常，函数就会捕获它，然后销毁所有已构造好的对象，并重新抛出异常。

raw_storage_iterator
Temp_iterator

模板类 `raw_storage_iterator` 遵循输出迭代器的通常模式。它的特殊能力在于它可以在原始存储空间上构造对象，就好像它在“输出”它们一样。模板类 `Temp_iterator` 是一个类似的输出迭代器，但它具有更多的机制。如果可能的话，它会在构造对象时分配一块临时缓冲区，然后在析构时把它所获得的缓冲区释放掉。它知道应该在什么时候构造原始存储空间，什么时候分配已构造的存储空间。它还维护着一个“快餐”式的基础实现子对象 `Bufpar`。这样的迭代器的拷贝将共享最初的子对象。

这还远不是一种完整的或健壮的设计。如果想彻底地完成一件事情，该子对象必须是一个独立的实体，并且有着引用计数（reference counted）功能。它所封装的应该多于它自身所维护的函数。不过，在此处给出的简单设计已经能够达到那些使用 `Temp_iterator` 的算法的需求了。它们足够程式化，可以避免更多的完整语义的要求。

auto_ptr

最后，模板类 `auto_ptr` 给出了一种智能指针形式。类 `auto_ptr<T>` 的对象存储一个指向类型为 `T` 的对象的指针，该对象分配在堆上（通过使用 `new` 表达式）。在它的析构函数中，它的直接任务是删除它所拥有的对象。然而，当考虑到所有的那些成员模板函数时，事情变得复杂了。在对象所存储的指针可以互换或者赋值的前提下，它们有意地允许在这些对象间进行互换或者赋值。（例如，将类 `auto_ptr<const int>` 的对象赋值给类 `auto_ptr<int>` 的对象是可能的。）它们同样也有意地允许将这样的对象作为参数传递给函数调用以及作为函数的返回值使用。这也就是我们为什么要加上一个新增的（也是私有的）嵌套类 `auto_ptr_ref` 以及那个有趣的构造函数和模板类型转换操作符来处理该嵌套类的对象的原因。

这种方法的理论基础依赖于 C++ 标准中的一个相当深奥的话题。我们不会在此讨论这种语言问题。我们只评述我们所知道的所有 C++ 编译器中模板类 `auto_ptr` 的实现。至少有两个相当完整的编译器可以让它以现有的方式工作，但对于其他的编译器来说就不得不对它进行简化。从标准化委员会对提交上来的缺陷报告的回应来看，它的精确规范可能会有一些小小的改动。从实用的角度来讲，不让你所写的代码依赖于 `auto_ptr` 中有歧义的特性是一个聪明的做法。即使在代码看起来工作正常时也不要这么做，因为它可能不具有很好的移植性，而且以后可能会需要改动。

测试<memory>

tmemory.c 程序清单 4-3 中列出了文件 tmemory.c。由于头文件<memory>实在太大了，所以让测试程序将它分做不同部分分别进行了测试：

```
程序清单 4-3: // test <memory>
tmemory.c
#include <assert.h>
#include <iostream>
#include <memory>
#include <new>
using namespace std;

// CLASS Myint
static size_t cnt;
class Myint {
public:
    Myint(int x)
        : val(x) {++cnt; }
    Myint(const Myint& x)
        : val(x.val) {++cnt; }
    ~Myint()
        {--cnt; }
    int get_val() const
        {return (val); }
private:
    int val;
};

typedef allocator<float> Myal;
Myal get_al()
    {return (Myal()); }

// TEST allocator
void test_alloc()
{
    float f1;
    Myal::size_type *p_size = (size_t *)0;
    Myal::difference_type *p_val = (ptrdiff_t *)0;
    Myal::pointer *p_ptr = (float **)0;
    Myal::const_pointer *p_cptr = (const float **)0;
    Myal::reference p_ref = f1;
    Myal::const_reference p_cref = (const float&)f1;
    Myal::value_type *p_dist = (float *)0;
    Myal::rebind<int>::pointer *p_iptr = (int **)0;
```

```
Myal al0 = get_al(), al(al0);

allocator<void>::pointer *pv_ptr = (void **)0;
allocator<void>::const_pointer *pv_cptr= (const void**)0;
allocator<void>::value_type *pv_dist = (void *)0;
Myal::rebind<int>::pointer *pv_iptr = (int **)0;
allocator<void> alv0, alv(alv0);
alv = alv0;

float *pfl = al0.address(f1);
assert(pfl == &f1);
pfl = al.allocate(3, 0);
al.construct(&pfl[2], 2.0F);
assert(pfl[2] == 2.0F);
al.destroy(pfl);
al.deallocate(pfl, 1);
assert(0 < al0.max_size());
assert(al0 == al);
assert(!(al0 != al0));
al.destroy(pfl);
al.deallocate(pfl, 1); }

// TEST UNINITIALIZED COPY AND FILL
void test_uninit()
{
    cnt = 0;
    Myint *p = (Myint *)operator new(6 * sizeof (Myint));
    uninitialized_fill(p, p + 2, 3);
    assert(p[1].get_val() == 3 && cnt == 2);
    uninitialized_fill_n(p + 2, 2, 5);
    assert(p[3].get_val() == 5 && cnt == 4);
    assert(uninitialized_copy(p + 1, p + 3, p + 4) == p + 6);
    assert(p[4].get_val() == 3 && cnt == 6);
    assert(p[5].get_val() == 5);
    operator delete(p); }

// TEST TEMPORARY BUFFERS
void test_tempbuf()
{
    pair<short *, ptrdiff_t> tbuf =
        get_temporary_buffer(5);
    assert(tbuf.first != 0 && tbuf.second == 5);
    typedef raw_storage_iterator<short *, short> Rit;
    Rit::iter_type *p_iter = (short **)*0;
    Rit::element_type *p_elem = (short *)0;
    Rit it(tbuf.first);
    for (int i = 0; i < 5; ++i)
```

```
        (*it++ = i;
         assert(tbuf.first[i] == i); }
        return_temporary_buffer(tbuf.first); }

// TEST auto_ptr
void test_autoptr()
{cnt = 0;
typedef auto_ptr<Myint> Myptr;
Myptr::element_type *p_elem = (Myint *)0;

Myptr p0;
assert(p0.get() == 0);
(Myptr p1(new Myint(3));
Myint *p = p1.get();
assert(cnt == 1);
assert(p->get_val() == 3
&& (*p1).get_val() == 3
&& p1.release()->get_val() == 3
&& p1.get() == 0);
delete(p);
assert(cnt == 0); }
assert(cnt == 0);
(Myptr p2(new Myint(5));
assert(cnt == 1); }
assert(cnt == 0);
(Myptr p3(new Myint(7)), p4(p3));
assert(cnt == 1);
assert(p3.get() == 0 && p4.get()->get_val() == 7);
p3 = p4;
assert(p4.get() == 0 && p3.get()->get_val() == 7);
p3.reset();
assert(p3.get() == 0 && cnt == 0);
p4.reset();
assert(p4.get() == 0); }
assert(cnt == 0); }

// TEST <memory>
int main()
{test_alloc();
test_uninit();
test_tempbuf();
test_autoptr();
cout << "SUCCESS testing <memory>" << endl;
return (0); }
```

- 那些用于检测迭代器的各种属性以及对迭代器进行简单操作的模板。
- 模板类 allocator。
- 模板函数 uninitialized_copy 及其伙伴。
- 临时缓冲区以及原始存储空间迭代器。
- 模板类 auto_ptr。

没有一个测试会异常严厉——特别是对模板类 allocator 的测试。大部分的测试只是检测那些最基本的功能。

如果所有的测试都正常运行的话，测试程序将打印出：

```
SUCCESS testing <memory>
```

并正常退出。

习题

- 习题4-1 为什么显式特化 allocator<void> 没有定义成员类型 reference 和 const_reference?
- 习题4-2 如何使用分配器的成员函数 max_size?
- 习题4-3 写出一个 allocator，它维护着一个由预先分配的对象组成的列表，在使用 operator new 分配新的对象时，那些已分配的对象会优先考虑重用。
- 习题4-4 某些对于序列的操作（如排序、合并以及反转），如果它们产生的结果是复制到一块新的存储空间中而不是在已有的序列上进行重排列的话，那么实现它们可以更容易。如何使用临时缓冲区以从在适当的位置完成操作的复制方法中获得好处？当临时缓冲区大小达不到你所要求的时，这样获得的好处又会有一个什么样的折衷呢？
- 习题4-5 模板类 auto_ptr 可不可以“拥有”一个数组对象？如果可以，请描述如何声明及使用这样的智能指针。如果不可以，请描述如何对 auto_ptr 的语法加以扩展以支持数组对象。
- 习题4-6 [较难] 写出一个分配器，它将对象存储在一个磁盘文件中，并且维护一个小小的高速缓存（cache）来存储程序内存（program memory）中的有效对象。
- 习题4-7 [特难] 在哪种情况下，分配器应该以与模板类 allocator 不一致的方式来定义成员函数 construct 和 destroy？

第5章 算法

背景知识

算法是STL中的“苦工”。它们表现为一系列的模板函数。也就是说，在STL中最类似于传统函数库的那部分就是算法。我们一直强调，算法的主要不同之处在于：算法是一些模板函数。它们并不是作为预先编译好的对象模块组成的可链接库来提供的。确切地说，它们一般都是完整地定义在STL头文件中。我们可以以众多的方式来特化每一个模板函数，以此极大地提升它作为泛型程序组件的适用性。

迭代器

除了少数的例外情况，这些模板函数都是使用迭代器作为它的参数，以此来在序列上进行各种操作。这也就是我们在本书的一开始花大力气来详细介绍迭代器的原因，尽管这个话题有时候会很枯燥。在了解了迭代器的广泛潜能之后，现在你应该准备好去了解这些算法中的含意了。

这些模板函数相当独立。我们只需添加极少的代码，就可以把它们放置在程序的各个地方，尤其是在那些我们原本使用传统指针作为迭代器的地方。也就是说，我们在下几章中所提出的方法可以很快运用在我们所写的任何C++代码中，并且作用相当明显。

容器

这些算法和我们将要在本书的后面部分描述的STL容器之间合作得非常好。实际上，许多容器模板类的成员函数为了突出其优势而使用了这些算法。但是请注意，对容器的理解并不是使用算法所必需的。算法并不直接使用STL容器。确切地说，算法通过使用这些容器对象的成员函数提供的迭代器来操作容器对象所管理的序列。这也就是我们把容器放在本书的最后讲述的原因。可以在不知道容器的前提下理解算法，但在一点也不了解算法的前提下想把容器描述清楚就不那么简单了。

<algorithm>
<numeric>
<functional>

我们已经在前面介绍的几个头文件中描述过一些算法，例如第4章中的uninitialized_copy、uninitialized_fill和uninitialized_fill_n。剩下的算法在两个头文件<algorithm>和<numeric>中定义。另外还有一个头文件<functional>，在它里面定义了许多的用以描述函数对象(function object)的模板类，很多算法可以使用这些函数对象以发挥其优势。

很多算法的一个关键部分是它必须能够完成确定的测试。举例来说，为了从一个序列中找到最小的元素，我们可能需要定义一些类似于“smallest”的谓词。更确切地说，算法必须重复性地检测诸如smaller(x,

y)之类的表达式的布尔结果(在这个表达式中, x和y都是序列中的元素)。在这种情况下, 显而易见的解决方法就是使用表达式 $x < y$ ——实际上有很多的算法也是这么做的。例如, 我们可以声明模板函数如下:

```
template<class FwdIt>
FwdIt smallest(FwdIt first, FwdIt last);
```

它返回一个迭代器, 指向区间[first, last)中最小的那个元素(使用operator<来判断元素的大小)。

函数对象

但是采取这样的方法来实现该算法的一个重要方面实在让人觉得遗憾, 尤其是当模板机制容许以很大的自由度存取序列的时候更是如此。于是, 我们引入了函数对象(function object)。函数对象没有必要存储任何数据。和我们在第3章中讲述的迭代器标签一样, 函数对象的类型本身就已经传达了足够的信息。下面就是一种声明函数对象f的方法:

```
struct smaller_int {
    bool operator()(int x, int y) const
        {return (x < y); }
} f;
```

可以将f看作函数名。例如, 如果一切无误的话, 表达式f(x, y)就会调用在smaller_int中所定义的那个成员函数。

也就是说, 可以声明另一个版本的模板类smallest, 使它看起来如下所示:

```
template<class FwdIt, class Pred>
FwdIt smallest(FwdIt first, FwdIt last, Pred pr);
```

然后就可以在实际调用该函数时通过smallest提供你所希望的功能, 如下所描述的一样:

```
int get_smallest(int *first, int *last)
{return (*smallest(first, last, smaller_int())); }
```

在上面的函数调用中, 第三个参数产生了一个无关紧要的对象, 它的主要目的就是在模板函数特化时, 检测其参数Pred的实际类型。

顺便说一句, 请注意: 第三个参数也可以为一个指向函数的指针。也就是说, 也可以将上面的例子重写成如下模样:

```
bool is_smaller_int(int x, int y)
{return (x < y); }
```

```
int get_smallest(int *first, int *last)
    {return (smallest(first, last, &is_smaller_int)); }
```

(其中在is_smaller_int前面的&符号是可选的。) C和C++都允许将表达式pfn(x, y)作为更有启迪作用的(*pfn)(x, y)的替代形式。这样，模板特化将产生实际有效的代码。

与指向函数的指针相比，函数对象有如下的几个好处：

- 函数对象可以定义多个重载版本。
- 在函数被调用时，可以使用函数对象所存储的值。
- 在必要时可以用一个函数指针所替换。
- 函数调用很有可能会被内联代码替换。

和上面给出的模板函数smallest差不多的是，算法由通常以两种形式出现在STL中的谓词来表现，其中一种形式固定了最可能使用的谓词形式；另一种形式则以一个附加的函数对象参数来给出该谓词。

这种方法的最终结果就是使实现算法的代码体积几乎大了一倍，至少在一个典型的实现中是这样的。但是我们由此获得的回报就是：对于那些经常使用的谓词，可以得到紧凑且高效的代码，并且不会为此牺牲我们在使用函数对象时所带来的功能。

我们将在后续章节中发现函数对象的功能确实是非常强大的。第8章将提供大量的细节。

功能描述

在对算法的描述中，我们会使用这几个惯用语：

在区间...中

惯用语“在区间[A, B)中”意味着具有0个或多个离散值的序列从A开始到B结束（但不包括B）。区间只有在B对于A来说是可到达(reachable)时才算是有效的——A可到达B意味着：把A存储在一个对象N中($N = A$)，然后对这个对象进行0次或多次递增($++N$)，在进行有限次的递增后该对象和B相等($N == B$)。

位于区间...中的每个N

惯用语“位于区间[A, B)中的每个N”意味着N是从值A开始、经过0次或多次递增后直到等于B时的所有值。其中 $N == B$ 这种情况不在区间范围内。

最小值

惯用语“对于位于区间[A, B)中的每个N，使得X成立的最小值(lowest value)”意味着从A开始，对于位于区间[A, B)中的每个N，当判断条件X成立时，那个值就是我们所说的“最小值”。

最大值

惯用语“对于位于区间[A, B)中的每个N，使得X成立的最大值(highest value)”通常意味着对于所有位于区间[A, B)中的N都检测一次

X是否成立。该函数在每次X成立时都会把N的一个拷贝存储到K中。如果最后K中存储有某个值的话，我们就会用K的值来替换N的最终值（它本来等于B）。然而，对于双向或者随机存取迭代器来说，它也可能意味着N是从区间末端开始，经过有限次的递减后，使判断条件X成立的那个值。

X - Y

当X和Y可以为不同于随机存取迭代器的迭代器时，诸如X - Y这样的表达式有着和其外表相符的数学意义。如果该函数一定要得到这样的一个值的话，它并不一定要执行operator-。这同样也适用于诸如X + N和X - N（此处N为一个整型值）这样的表达式。

严格弱序

有些算法使用的谓词对序列中的元素对施加严格弱序（strict weak ordering）。例如，对于谓词pr(X, Y)来说：

- pr(X, X)为false（即在排序好的序列中，X不可能排在它自身前面）。
- 当!pr(X, Y) && !pr(Y, X)为true时，我们就说X和Y有相等次序（equivalent ordering）（并没有定义X == Y）。
- pr(X, Y) && pr(Y, Z)意味着pr(X, Z)（次序具有传递性）。

这些算法中的部分隐式地使用了谓词X < Y，还有部分使用了作为函数对象传递过来的谓词pr(X, Y)。满足严格弱序需求的谓词有：操作于算术类型以及string对象上的X < Y和X > Y。然而，请注意：操作于同样类型上面的谓词X <= Y和X >= Y并不能够满足这个需求。

按...排序

如果对于区间[0, last - first]中的每个N以及区间(N, last - first)中的每个M，谓词!(*first + M) < *(first + N)都为true，那么我们就说：由区间[first, last]中的迭代器所指定的元素序列是一个“按operator<排序的序列”。（注意元素是以升序排序的。）谓词函数operator<（或者其他替代它的函数）不应该改变它们的操作数。而且，它必须在所比较的操作数上施加严格弱序。

堆

在以下条件满足时，我们就说由区间[first, last]中的迭代器所指定的元素序列是一个按operator<排序的堆（heap）：

- 对于每个位于区间[1, last - first]中的N来说，谓词!(*first < *(first + N))都为true。（即第一个元素就是堆中最大的那个元素。）
- 可能在对数时间内向堆中插入（push_heap）一个新元素或者从堆中删除（pop_heap）最大的元素并且结果序列也能够保持堆的准则。

对于堆的内部结构，我们仅能通过模板函数make_heap、pop_heap和push_heap来得知一二。（参见第6章。）对于一个有序序列来说，谓词函数operator<或者其他替代函数不应该改变它们的操作数，并且它们必须在所比较的操作数上施加严格弱序。

使用算法

我们可以像使用传统的函数库一样使用STL算法。将定义你想使用的模板函数的头文件包含到程序中。然而，请记住：对于不正常使用它们的程序来说，其出错模式将会很不一样。如果在程序中错误地调用了传统的函数，翻译器不是给出一个诊断信息，就是暗地里将一个或多个参数的类型转换成该函数所期望的类型。但对于后者来说，它将导致运行期间的古怪行为，不过幸好现在的调试程序（debugger）一般都可以帮我们识别这样的错误。

如果我们错误地特化一个模板函数，翻译器的反应可能和上面一样，也可能为模板参数选择一个错误的类型。这种不正确的类型可能会导致难以理解的出错信息产生，尤其是对于那些要特化其他模板的模板更是如此。在经过多次对模板的特化后，即便是运行时的除错工作也会变得异常困难。

为了缓和这个问题，我们惟一能做到的最重要的事情就是：将函数参数表达式中的隐式类型转换的使用最小化。无论如何，这都是一个很棒的主意，尤其是当函数有着多个重载版本时更是如此。参数表达式的类型可以影响到重载版本的选择，当没有一个重载版本的参数类型能够精确地和它匹配时，我们就有可能得到错误的重载调用。如果翻译器能够报告一个歧义性错误而不是自己胡乱地猜测，我们就应该感到庆幸了。

由于模板函数可以使用参数类型来决定模板参数，所以它误入歧途的机会是多样的。本书中的代码在参数表达式中所使用的类型转换机制都很明智，这极大地降低了出现类型不正确的可能性。我们建议大家采用类似的风格。

习题

习题5-1

请写出本章中所描述过的模板函数smallest的两种形式。

习题5-2

请对上一题中所写出的模板函数进行修改，使之可以使用输入迭代器（而不是前向迭代器）正常地工作。

习题5-3

在下面的序列中，dominates是否构成了一个严格弱序？如果没有，为什么？

```
rock dominates scissors  
scissors dominates paper  
paper dominates rock
```

- 习题5-4 在下面的序列中，哪些可能是堆？
{1}
{1, 1}
{1, 2}
{8, 7, 6, 5, 4, 3, 2, 1}
- 习题5-5 写出模板类less，使得`less<T>()(x, y)`在x小于y时返回true（此处x和y的类型都是T）。
- 习题5-6 写出模板类`less_by`，使得`less_by<T>(d)(x, y)`在 $x + d$ 小于y时返回true（此处d、x和y的类型都是T）。
- 习题5-7 [较难] 请写出模板函数`is_less_by`，使得指向它的一个特化版本的函数指针可以取代上题中的函数对象`less_by<T>(d)`。
- 习题5-8 [特难] 如果有一个序列，它所遵循的排序规则会在排序过程中发生改变（如：任务的优先化列表），你将如何对这个序列进行排序？

第6章 <algorithm>

背景知识

头文件<algorithm>无疑是所有的STL头文件中最大的一个。除此之外最大的那个头文件也不过只有它的三分之一大。然而，从许多方面来讲，头文件<algorithm>都是最好理解的一个。它是由一大堆模板函数组成的，可以认为每个函数在很大程度上都是独立的。

功能描述

```
namespace std {
    template<class InIt, class Fun>
        Fun for_each(InIt first, InIt last, Fun f);
    template<class InIt, class T>
        InIt find(InIt first, InIt last, const T& val);
    template<class InIt, class Pred>
        InIt find_if(InIt first, InIt last, Pred pr);
    template<class FwdIt1, class FwdIt2>
        FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                           FwdIt2 first2, FwdIt2 last2);
    template<class FwdIt1, class FwdIt2, class Pred>
        FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                           FwdIt2 first2, FwdIt2 last2, Pred pr);
    template<class FwdIt1, class FwdIt2>
        FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                               FwdIt2 first2, FwdIt2 last2);
    template<class FwdIt1, class FwdIt2, class Pred>
        FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                               FwdIt2 first2, FwdIt2 last2, Pred pr);
    template<class FwdIt>
        FwdIt adjacent_find(FwdIt first, FwdIt last);
    template<class FwdIt, class Pred>
        FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
    template<class InIt, class T, class Dist>
        typename iterator_traits<InIt>::difference_type
            count(InIt first, InIt last,
                    const T& val);
```

```
template<class InIt, class Pred, class Dist>
    typename iterator_traits<InIt>::difference_type
        count_if(InIt first, InIt last,
                  Pred pr);
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
                                 InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
                                 InIt2 x, Pred pr);
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
                  FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
                  FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
                  Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
                  Dist n, const T& val, Pred pr);
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
                         BidIt2 x);
template<class T>
    void swap(T& x, T& y);
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
                        FwdIt2 x);
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
                  Unop uop);
template<class InIt1, class InIt2, class OutIt,
         class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
                  InIt2 first2, OutIt x, Binop bop);
```

```
template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
                  const T& vold, const T& vnew);
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
                     Pred pr, const T& val);
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
                         vconst T& vold, const T& vnew);
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
                           Pred pr, const T& val);
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x,
                        const T& val);
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x,
                          Pred pr);
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x,
                       Pred pr);
template<class BidIt>
    void reverse(BidIt first, BidIt last);
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

```
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle,
                      FwdIt last, OutIt x);
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
template<class BidIt, class Pred>
    BidIt stable_partition(BidIt first, BidIt last,
                           Pred pr);
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
                       RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle,
                       RanIt last, Pred pr);
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
                           RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
                           RanIt first2, RanIt last2, Pred pr);
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
                      Pred pr);
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last,
                        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last,
                        const T& val, Pred pr);
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
```

```
    const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val, Pred pr);
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last, Pred pr);
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bbool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
```

```
template<class InIt1, class InIt2, class OutIt,
         class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
                           InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                           InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
         class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                           InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1,
                                    InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
         class Pred>
    OutIt set_symmetric_difference(InIt1 first1,
                                    InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
                                    Pred pr);
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
```

```

template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
        Pred pr);
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last,
        Pred pr);
};

```

包含 STL 的标准头文件<algorithm>可以得到大量模板函数的定义，以便为我们实现一些有用的算法。下面的描述扩展了普通模板参数名字（或前缀）的使用，指出了允许作为实际参数类型的迭代器的最小种类：

- **OutIt** ——一个输出迭代器
- **InIt** ——一个输入迭代器
- **FwdIt** ——一个前向迭代器
- **BidIt** ——一个双向迭代器
- **RanIt** ——一个随机存取迭代器

这些模板的描述使用了在算法中常见的大量惯例。

口 adjacent_find

```

template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);

```

第一个模板函数会查找区间[0, last - first)中的最小值 N，使得 $N + 1 \neq \text{last} - \text{first}$ 并且谓词 $*(\text{first} + N) == *(\text{first} + N + 1)$ 为 true。然后它会返回 $\text{first} + N$ 。如果不存在这样的值，函数返回 last。它计算该谓词 $N +$

1 次。

第二个模板函数的行为和第一个相同，不过它用的谓词是 $\text{pr}(*(\text{first} + N), *(\text{first} + N + 1))$ 。

口 binary_search

```
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

第一个模板函数会查找区间 $[0, \text{last} - \text{first}]$ ，看其中是否存在值 N 使得 $*(\text{first} + N)$ 与 val 有相等次序；此处由区间 $[\text{first}, \text{last}]$ 中的迭代器指定的元素是按 $\text{operator}<$ 排序的。如果找到了这样的值，函数就返回 true ，否则返回 false 。

该函数计算次序谓词 $X < Y$ 的次数最多为 $\text{ceil}(\log(\text{last} - \text{first})) + 2$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

口 copy

```
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
```

该模板函数会对存在于区间 $[0, \text{last} - \text{first}]$ 中的每个 N ，严格地按照从最小到最大的顺序计算 $*(\text{x} + N) = *(\text{first} + N)$ ，然后返回 $\text{x} + N$ 。如果 x 和 first 指定存储空间的区域，那么 x 一定不能出现在区间 $[\text{first}, \text{last}]$ 中。

口 copy_backward

```
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
        BidIt2 x);
```

该模板函数会对位于区间 $[0, \text{last} - \text{first}]$ 中的每个 N ，严格地按照从最小到最大的顺序计算 $*(\text{x} - N - 1) = *(\text{last} - N - 1)$ ，然后返回 $\text{x} - (\text{last} - \text{first})$ 。如果 x 和 first 指定存储空间的区域，那么 x 一定不能出现在区间 $[\text{first}, \text{last}]$ 中。

口 count

```
template<class InIt, class T>
    typename iterator_traits<InIt>::difference_type
        count(InIt first, InIt last, const T& val);
```

该模板函数先将一个计数器 n 置为 0。然后对位于区间 [0, last - first) 中的每个 N，如果谓词 $*(first + N) == val$ 返回 true，那么就执行 $++n$ 。在此，operator== 必须是对它的操作数进行相等关系的判断。函数最后返回 n。它恰好计算谓词 last - first 次。

口 count_if

```
template<class InIt, class Pred, class Dist>
typename iterator_traits<InIt>::difference_type
count_if(InIt first, InIt last,
Pred pr);
```

该模板函数先将一个计数器 n 置为 0。然后对位于区间 [0, last - first) 中的每个 N，如果谓词 $pr(*(\text{first} + N))$ 返回 true，那么就执行 $++n$ 。函数最后返回 n。它恰好计算谓词 last - first 次。

口 equal

```
template<class InIt1, class InIt2>
bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
```

第一个模板函数只有在下面条件满足的情况下才返回 true：对位于区间 [0, last1 - first1) 中的每个 N，谓词 $*(\text{first1} + N) == *(\text{first2} + N)$ 都返回 true。在此，operator== 必须对它的操作数进行相等关系的判断。对于区间内的每个 N，该函数最多计算谓词一次。

第二个模板函数的行为和第一个相同，不过它用的谓词是 $pr(*(\text{first} + N), *(\text{first2} + N))$ 。

口 equal_range

```
template<class FwdIt, class T>
pair<FwdIt, FwdIt> equal_range(FwdIt first,
FwdIt last, const T& val);

template<class FwdIt, class T, class Pred>
pair<FwdIt, FwdIt> equal_range(FwdIt first,
FwdIt last, const T& val, Pred pr);
```

当由区间 [first, last) 中的迭代器指定的元素序列是按 operator< 排序时，第一个模板函数有效地返回 pair(lower_bound(first, last, val), upper_bound(first, last, val))。也就是说，该函数会在区间内检测到最大的那个子区间，在这个子区间内可以随处插入 val 而不会影响它的次序。

该函数计算次序谓词 X < Y 的次数最多为 $\text{ceil}(2 * \log(\text{last} - \text{first})) + 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 fill

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

该模板函数会对位于区间[0, last - first)中的每个 N 计算一次*(first + N) = x。

口 fill_n

```
template<class OutIt, class Size, class T>
void fill_n(OutIt first, Size n, const T& x);
```

该模板函数会对位于区间[0, n)中的每个 N 计算一次*(first + N) = x。

口 find

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

该模板函数会检测位于区间[0, last - first)中的最小的 N，使得谓词 *(first + N) == val 返回 true。在此，operator==必须对它的操作数进行相等关系的判断。然后函数会返回 first + N。如果不存在这样的值，函数将返回 last。对于区间的每个 N，该函数最多计算谓词一次。

口 find_end

```
template<class FwdIt1, class FwdIt2>
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                  FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
                  FwdIt2 first2, FwdIt2 last2, Pred pr);
```

第一个模板函数会检测位于区间[0, last1 - first1 - (last2 - first2))中的最大的 N，使得对于区间[0, last2 - first2)内的每个 M，谓词*(first1 + N + M) == *(first2 + N + M)都为 true。在此，operator==必须对它的操作数进行相等关系的判断。函数最后返回 first1 + N。如果不存在这样的值，函数将返回 last1。它最多计算谓词(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)次。

第二个模板函数的行为和第一个相同，不过它使用的谓词是 pr(*(first1 + N + M), *(first2 + N + M))。

口 find_first_of

```
template<class FwdIt1, class FwdIt2>
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2, Pred pr);
```

第一个模板函数会检测位于区间[0, last1 - first1)中的最小的 N，使得对于区间[0, last2 - first2)中的某个 M，谓词*(first1 + N) == *(first2 + M)都为 true。在此，operator==必须是对它的操作数进行相等关系的判断。函数最后返回 first1 + N。如果不存在这样的值，函数将返回 last1。它最多计算谓词(last1 - first1) * (last2 - first2)次。

第二个模板函数的行为和第一个相同，不过它使用的谓词是 pr(*(first1 + N), *(first2 + M))。

口 find_if

```
template<class InIt, class Pred>
InIt find_if(InIt first, InIt last, Pred pr);
```

该模板函数会检测位于区间[0, last - first)中的最小的 N，使得谓词 pred(*(first + N))为 true。然后它将返回 first + N。如果不存在这样的值，函数将返回 last。对于区间内的每个 N，函数最多计算谓词一次。

口 for_each

```
template<class InIt, class Fun>
Fun for_each(InIt first, InIt last, Fun f);
```

该模板函数会对位于区间[0, last - first)中的每个 N 计算 f(*(first + N))。然后它将返回 f。

口 generate

```
template<class FwdIt, class Gen>
void generate(FwdIt first, FwdIt last, Gen g);
```

该模板函数会对位于区间[0, last - first)中的每个 N 计算*(first + N) = g()一次。

口 generate_n

```
template<class OutIt, class Pred, class Gen>
void generate_n(OutIt first, Dist n, Gen g);
```

该模板函数会对位于区间[0, n)中的每个 N 计算*(first + N) = g()一次。

口 includes

```
template<class InIt1, class InIt2>
bool includes(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
bool includes(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, Pred pr);
```

对于两个按 operator<排序的由区间[first1, last1)和区间[first2, last2)中的迭代器所指定的元素形成的序列，第一个模板函数会检测区间[0,

$\text{last2} - \text{first2}$)中是否存在一个值 N , 使得对于区间 $[0, \text{last1} - \text{first1}]$ 中的每个 M , $*(\text{first1} + M)$ 与 $*(\text{first2} + N)$ 之间都不存在相等次序关系。如果存在这样的值, 函数就返回 `false`; 否则函数返回 `true`。也就是说, 该函数会检测由区间 $[\text{first2}, \text{last2}]$ 中的迭代器指定的有序序列中的全体元素是否都能在由区间 $[\text{first1}, \text{last1}]$ 中的迭代器指定的元素中找到对应的相等次序的元素。

该函数最多计算谓词 $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ 次。

第二个模板函数行为和第一个相同, 不过它用 `pr(X, Y)` 替代了 `operator<(X, Y)`。

口 `inplace_merge`

```
template<class BidIt>
void inplace_merge(BidIt first, BidIt middle,
                    BidIt last);
template<class BidIt, class Pred>
void inplace_merge(BidIt first, BidIt middle,
                    BidIt last, Pred pr);
```

第一个模板函数会对两个按 `operator<` 排序的由区间 $[\text{first}, \text{middle}]$ 和 $[\text{middle}, \text{last}]$ 中的迭代器指定的序列进行合并, 以形成起始于 `first`、长度为 $\text{last} - \text{first}$ 的合并序列, 它也是按 `operator<` 排序的。合并操作并不会破坏原来序列中的元素之间的顺序。而且, 对于任意两个来自于不同源序列的元素, 如果它们之间存在相等次序关系, 有序区间 $[\text{first}, \text{middle}]$ 中的那个元素将排在另一个前面。

该函数计算谓词 $X < Y$ 的次数最多为 $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ 次。(如果存在足够大的临时存储空间, 它甚至可以最多只计算谓词 $(\text{last} - \text{first}) - 1$ 次。)

第二个模板函数的行为和第一个相同, 不过它用 `pr(X, Y)` 替代了 `operator<(X, Y)`。

口 `iter_swap`

```
template<class FwdIt1, class FwdIt2>
void iter_swap(FwdIt1 x, FwdIt2 y);
```

该模板函数使原来存储于`*y` 的值在函数调用后存储在`*x` 中, 而原存储于`*x` 的值却在函数调用后存储于`*y` 中。

口 `lexicographical_compare`

```
template<class InIt1, class InIt2>
bool lexicographical_compare(InIt1 first1,
                            InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
bool lexicographical_compare(InIt1 first1,
                            InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

第一个模板函数会检测 K 的大小 (K 是 last1 - first1 和 last2 - first2 中较小的那个值，我们用它表示要比较的元素的个数)。然后它会检测区间 [0, K] 中的最小的 N，使得 *(first1 + N) 与 *(first2 + N) 之间不存在相等次序关系。如果不存在这样的值，只有在 K < (last2 - first2) 时，函数返回 true。否则，只有在 *(first1 + N) < *(first2 + N) 时，函数才会返回 true。也就是说，函数只有在由区间 [first1, last1] 中的迭代器指定的序列按照字典顺序小于另一个序列时才会返回 true。

该函数计算次序谓词 X < Y 的次数最多为 $2 * K$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 lower_bound

```
template<class FwdIt, class T>
FwdIt lower_bound(FwdIt first, FwdIt last,
                     const T& val);
template<class FwdIt, class T, class Pred>
FwdIt lower_bound(FwdIt first, FwdIt last,
                     const T& val, Pred pr);
```

当由区间 [first, last] 中的迭代器所指定的序列是按 operator< 排序时，第一个模板函数会检测位于区间 (0, last - first] 中的最大的 N，使得对于区间 [0, N] 中的任意 M，谓词 *(first + M) < val 都为 true。然后函数会返回 first + N。也就是说，该函数会检测最前面的一个位置，使得将 val 插入到它的前面时，序列中已有的次序不会被破坏。

该函数计算次序谓词 X < Y 的次数最多为 $\text{ceil}(last - first) + 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 make_heap

```
template<class RanIt>
void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
void make_heap(RanIt first, RanIt last, Pred pr);
```

第一个模板函数会将由区间 [first, last] 中的迭代器指定的序列进行重排，生成一个按 operator< 排序的堆。

该函数计算次序谓词 X < Y 的次数最多为 $3 * (last - first)$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 max

```
template<class T>
const T& max(const T& x, const T& y);
```

```
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
```

当 $x < y$ 时，第一个模板函数将返回 y ，否则将返回 x 。在此处， T 只需提供一个单参数的构造函数和一个析构函数。

第二个模板函数的行为和第一个相同，不过它用 $pr(X, Y)$ 替代了 $\operatorname{operator}<(X, Y)$ 。

口 max_element

```
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
```

第一个模板函数会检测区间 $[0, \text{last} - \text{first}]$ 中的最小的 N ，使得对于区间 $[0, \text{last} - \text{first}]$ 中任意的 M ，谓词 $*(\text{first} + N) < *(\text{first} + M)$ 都为 `false`。然后函数返回 $\text{first} + N$ 。也就是说，函数会检测区间内的最大值第一次出现的地方。

该函数计算次序谓词 $X < Y$ 的次数正好为 $\max((\text{last} - \text{first}) - 1, 0)$ 次。

第二个模板函数的行为和第一个相同，不过它用 $pr(X, Y)$ 替代了 $\operatorname{operator}<(X, Y)$ 。

口 merge

```
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
                  InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
         class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
                  InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

第一个模板函数会检测 K 的大小 (K 为 $(\text{last1} - \text{first1}) + (\text{last2} - \text{first2})$)，我们用它来表示要复制的元素的个数。然后函数将交替地复制这两个由区间 $[\text{first1}, \text{last1}]$ 和 $[\text{first2}, \text{last2}]$ 中的迭代器指定按 $\operatorname{operator}<$ 排序的序列，最终生成一个起始于 x ，长度为 K 的合并序列，这个序列也是按 $\operatorname{operator}<$ 排序的。函数最后返回 $x + K$ 。

合并不会导致任何一个序列中元素之间次序的改变。而且，对于任意两个从不同序列中得来的元素，如果它们符合相等次序，那么来自于有序区间 $[\text{first1}, \text{last1}]$ 中的那个元素将排在前面。也就是说，该函数将两个有序序列合并以产生另一个有序的序列。

如果 x 和 first1 指定存储空间的区域，那么区间 $[x, x + K]$ 一定不能和区间 $[\text{first1}, \text{last1}]$ 重叠。如果 x 和 first2 指定存储空间的区域，那么区

区间 $[x, x + K)$ 一定不能和区间 $[first2, last2)$ 重叠。函数计算次序谓词 $X < Y$ 的次数最多为 $K - 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

口 min

```
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
```

当 $y < x$ 时，第一个模板函数返回 y ，否则将返回 x 。在此处， T 只需提供一个单参数的构造函数和一个析构函数。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

口 min_element

```
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
```

第一个模板函数会检测区间 $[0, last - first)$ 中的最小的 N ，使得对于区间 $[0, last - first)$ 中任意的 M ，谓词 $*(first + M) < *(first + N)$ 都为 `false`。然后函数返回 $first + N$ 。也就是说，函数将会检测区间内的最小值第一次出现的地方。

该函数计算次序谓词 $X < Y$ 的次数正好为 $\max((last - first) - 1, 0)$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

口 mismatch

```
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
```

第一个模板函数会检测位于区间 $[0, last1 - first1)$ 中的最小的 N ，使得谓词 $!(*(\text{first1} + N) == *(\text{first2} + N))$ 为 `true`。在此， $\text{operator}==$ 必须是对它的操作数进行相等关系的判断。然后函数返回 $\text{pair}(\text{first1} + N, \text{first2} + N)$ 。如果不存在这样的值， N 的值就为 $last1 - first1$ 。函数最多会对每个 N 计算一次谓词。

第二个模板函数的行为和第一个相同，不过它使用的谓词为 $\text{pr}(*(\text{first1} + N), *(\text{first2} + N))$ 。

¶ next_permutation

```
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
        Pred pr);
```

第一个模板函数会检测序列的重复排列，初始排列为由区间[first, last)中的迭代器指定的序列按 operator<排序时的那个序列。（即元素在序列中是以升序存储的。）然后，函数会对序列中的元素进行重排，通过对元素 X 和 Y 调用 swap(X, Y)0 次或多次，以产生下一个排列。只有在得到的序列不是最初的那个排列时，函数才会返回 true。否则，得到的序列是所有排列中按照字典顺序比原来的序列大的下一个序列。没有两个元素拥有相同次序。

该函数调用 swap(X, Y) 的次数最多为 $(\text{last} - \text{first})/2$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 operator<(X, Y)。

¶ nth_element

```
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
        Pred pr);
```

第一个模板函数将会对由区间[first, last)中的迭代器指定的序列进行重排，使得对于区间[0, nth - first)中的任意 N 和区间[nth-first, last - first)中的任意 M，谓词 $\text{!}(*(\text{first} + M) < (\text{first} + N))$ 都为 true。此外，对于等于 nth - first 的那个 N，以及区间(nth - first, last - first)中的任意 M，谓词 $\text{!}((\text{first} + M) < *(\text{first} + N))$ 为 true。也就是说，如果 nth != last，当整个序列是以升序排列（按 operator<排序）时，那么元素*nth 位于它所应该在的位置上。在该有序序列中，任何在它以前的元素都位于它之前的区间中，任何在它之后的元素都位于它之后的区间中。

该函数计算次序谓词 X < Y 的次数平均值与 last - first 成比例。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(x,y)$ 替代了 operator<(x,y)。

¶ partial_sort

```
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
```

```

        RanIt last);
template<class RanIt, class Pred>
void partial_sort(RanIt first, RanIt middle,
                    RanIt last, Pred pr);

```

第一个模板函数会重排由区间[first, last)中的迭代器指定的序列，使得对于区间[0, middle - first)中的任意 N 和区间(N, last - first)中的任意 M，谓词 $\text{!}(*\text{first} + M) < *(\text{first} + N)$ 都为 true。也就是说，整个序列中最小的 middle - first 个元素是以升序的方式（按 operator<排序）存储的。其他元素的次序则没有指定。

该函数计算次序谓词 X < Y 的次数最多为 $(last - first) * \log(middle - first)$ 的常数倍。

第二个模板函数的行为和第一个相同，不过它用 pr(x,y) 替代了 operator<(x,y)。

¶ partial_sort_copy

```

template<class InIt, class RanIt>
RanIt partial_sort_copy(InIt first1, InIt last1,
                        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
RanIt partial_sort_copy(InIt first1, InIt last1,
                        RanIt first2, RanIt last2, Pred pr);

```

第一个模板函数会检测 K 的大小（它是 last1-first1 和 last2-first2 中较小的那个值，我们用它来表示要复制元素的个数）。然后复制及重排由区间[first1, last1)中的迭代器指定的 K 个元素到由 first2 指定开始处的序列中去（按 operator<排序）。此外，对于与未复制的元素相对应的区间[0, K)中的任意 N 和区间(0, last1 - first1)中的任意 M，谓词 $\text{!}(*(\text{first2} + M) < *(\text{first1} + N))$ 都为 true。也就是说，整个由区间[first1, last1)中的迭代器指定的序列中最小的 K 个元素被复制到了区间[first2, first2 + K)中，并且以升序方式存储。

该函数计算次序谓词 X < Y 的次数最多为 $(last - first) * \log(K)$ 的常数倍。

第二个模板函数的行为和第一个相同，不过它用 pr(x,y) 替代了 operator<(x,y)。

¶ partition

```

template<class BidIt, class Pred>
BidIt partition(BidIt first, BidIt last, Pred pr);

```

该模板函数会重排由区间[first, last)中的迭代器指定的序列，同时还会检测 K 的大小，使得对于区间[0, K)中的任意 N，谓词 $\text{pr}(*(\text{first} + N))$ 都为 true，并且对于区间[K, last - first)中的任意 N，谓词 $\text{pr}(*(\text{first} + N))$

都为 false。然后函数返回 first + K。

该谓词不能改变其操作数。函数调用 pr(*(first + N))的次数正好是 last - first 次，并且交换元素对的次数最多为(last - first)/2 次。

口 pop_heap

```
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
```

第一个模板函数会重排由区间[first, last)中的迭代器指定的序列，产生按 operator<排序、由区间[first, last - 1)中的迭代器指定的新的堆，将最初位于*first 的元素放置在*(last - 1)处。最初的序列必须指定一个存在的堆，它也是按 operator<排序。这样，first != last 必须为 true，并且 *(last - 1)是要从堆中移除（弹出）的元素。

该函数计算次序谓词 X < Y 的次数最多为 ceil(2 * log(last - first)) 次。

第二个模板函数的行为和第一个相同，不过它使用谓词 pr(X, Y) 替代了 operator<(x,y)。

口 prev_permutation

```
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last, Pred pr);
```

第一个模板函数会检测序列的重复排列，该序列的初始排列为由区间[first, last)中的迭代器指定的元素按照 operator<的相反方式排序的那个序列。（即元素在序列中是以降序方式存储的。）然后，函数会对序列中的元素进行重排，通过对元素 X 和 Y 调用 swap(X, Y)0 次或多次，以产生前一个排列。只有当得到的序列不是最初的那个排列时，函数才返回 true。否则，得到的序列是所有排列中按照字典排序方式比原来的序列小的下一个序列。不存在两个相同次序的元素。

该函数调用 swap(X, Y)的次数最多为(last - first)/2 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 push_heap

```
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
```

第一个模板函数会重排由区间[first, last)中的迭代器指定的序列，产

生一个新的按 operator<排序的堆。区间[first, last - 1)中的迭代器必须指定一个已经存在的、同样也是按 operator<排序的堆。这样，first != last 必须为 true，并且*(last - 1)就是那个要新增入（推入）堆的元素。

该函数调用次序谓词 X < Y 的次数最多为 $\text{ceil}(2 * \log(\text{last} - \text{first}))$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(x,y)。

口 random_shuffle

```
template<class RanIt>
void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
void random_shuffle(RanIt first, RanIt last, Fun& f);
```

第一个模板函数会为区间[1, last - first)中的每个 N 都调用一次 swap(*(first + N), *(first+M))，此处 M 是区间[0, N]中随机分布的值。也就是说，该函数会随机地对序列中的元素进行重新排列。

该函数计算 M 的值及调用 swap 的次数正好为 last - first - 1 次。

第二个函数的行为和第一个相同，不过它的 M 是(Dist)f((Dist)N)，此处 Dist 为一个可以转换为 iterator_traits::difference_type 的类型，f 则是我们指定的一个函数。

口 remove

```
template<class FwdIt, class T>
FwdIt remove(FwdIt first, FwdIt last, const T& val);
```

该模板函数会先将 first 赋值给 X，然后对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (!(*(first + N) == val))
    *X++ = *(first + N);
```

在此，operator== 必须对它的操作数进行相等关系的判断。函数最后返回 X。也就是说，在结果序列中，函数将会移除掉所有令谓词*(first + N) == val 为 true 的元素，而剩下的元素的排列顺序不会改变，最后返回指向结果序列末端的迭代器值。

口 remove_copy

```
template<class InIt, class OutIt, class T>
OutIt remove_copy(InIt first, InIt last, OutIt x,
                    const T& val);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (!(*(first + N) == val))
    *x++ = *(first + N);
```

在此，operator==必须对它的操作数进行相等关系的判断。函数最后返回 x。也就是说，在结果序列中，函数将会移除掉所有令谓词*(first + N) == val 为 true 的元素，而剩下的元素的排列顺序不会改变，最后返回指向结果序列末端的迭代器值。

如果 x 和 first 指定存储空间的区域，那么区间[x, x + (last - first)) 不能和区间[first, last)重叠。

口 remove_copy_if

```
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x,
                           Pred pr);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (!pr(*(first + N)))
    *x++ = *(first + N);
```

函数最后返回 x。也就是说，在结果序列中，函数将会移除掉所有令谓词 pr(*(first + N)) 为 true 的元素，而剩下的元素的排列顺序不会改变，最后返回指向结果序列末端的迭代器值。

如果 x 和 first 指定存储空间的区域，那么区间[x, x + (last - first)) 不能和区间[first, last)重叠。

口 remove_if

```
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
```

该模板函数先将 first 赋值给 X，然后对于区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (!pr(*(first + N)))
    *X++ = *(first + N);
```

然后函数返回 X。也就是说，在结果序列中，函数将会移除掉所有令谓词 pr(*(first + N)) 为 true 的元素，而剩下的元素的排列顺序不会改变，最后返回指向结果序列末端的迭代器值。

口 replace

```
template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
                  const T& vold, const T& vnew);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (*(first + N) == vold)
    *(first + N) = vnew;
```

在此，operator==必须对它的操作数进行相等关系的判断。

口 replace_copy

```
template<class InIt, class OutIt, class T>
OutIt replace_copy(InIt first, InIt last, OutIt x,
const T& vold, const T& vnew);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (*first + N) == vold)
    *(x + N) = vnew;
else
    *(x + N) = *(first + N)
```

在此，operator==必须是对它的操作数进行相等关系的判断。

该函数返回指向结果序列末端的迭代器值。

如果 x 和 first 指定存储空间的区域，那么区间[x, x + (last - first))
不能和区间[first, last)重叠。

口 replace_copy_if

```
template<class InIt, class OutIt, class Pred, class T>
OutIt replace_copy_if(InIt first, InIt last,
OutIt x, Pred pr, const T& val);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (pr(*first + N))
    *(x + N) = val;
else
    *(x + N) = *(first + N)
```

如果 x 和 first 指定存储空间的区域，那么区间[x, x + (last - first))
不能和区间[first, last)重叠。

该函数返回指向结果序列末端的迭代器值。

口 replace_if

```
template<class FwdIt, class Pred, class T>
void replace_if(FwdIt first, FwdIt last,
Pred pr, const T& val);
```

该模板函数会对区间[0, last - first)中的每个 N 执行一次下列语句：

```
if (pr(*first + N))
    *(first + N) = val;
```

口 reverse

```
template<class BidIt>
void reverse(BidIt first, BidIt last);
```

该模板函数会对区间 $[0, (last - first) / 2]$ 中的每个 N 调用一次 $\text{swap}(*(\text{first} + N), *(\text{last} - 1 - N))$ 。也就是说，该函数会把序列中的元素顺序进行一次反转。

口 reverse_copy

```
template<class BidIt, class OutIt>
OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
```

该模板函数会对区间 $[0, last - first)$ 中的每个 N 计算一次 $*(\text{x} + N) = *(\text{last} - 1 - N)$ 。然后函数返回 $\text{x} + (last - first)$ 。也就是说，该函数会反转所复制序列中的元素的顺序。

如果 x 和 first 指定存储空间的区域，那么区间 $[\text{x}, \text{x} + (last - first))$ 不能和区间 $[\text{first}, \text{last})$ 重叠。

口 rotate

```
template<class FwdIt>
void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

该模板函数会对区间 $[0, last - first)$ 中的每个 N，将最初存储于 $*(\text{first} + (N + (\text{middle} - \text{last})) \% (\text{last} - \text{first}))$ 中的值转而存储到 $*(\text{first} + N)$ 中。也就是说，如果元素的“左移”导致最初存储于 $*(\text{first} + (N + 1) \% (\text{last} - \text{first}))$ 中的元素转而存储到 $*(\text{first} + N)$ 中，那么我们就可以说函数对序列加了旋转，或是对 $\text{middle} - \text{first}$ 个元素加以左移，或是对 $\text{last} - \text{middle}$ 个元素加以右移。在此， $[\text{first}, \text{middle})$ 和 $[\text{middle}, \text{last})$ 都必须是有效的区间。函数最多交换 $\text{last} - \text{first}$ 次元素对。

口 rotate_copy

```
template<class FwdIt, class OutIt>
OutIt rotate_copy(FwdIt first, FwdIt middle,
                    FwdIt last, OutIt x);
```

该模板函数会对区间 $[0, last - first)$ 中的每个 N 计算一次 $*(\text{x} + N) = *(\text{first} + (N + (\text{middle} - \text{first})) \% (\text{last} - \text{first}))$ 。也就是说，如果一个元素的“左移”导致最初存储于 $*(\text{first} + (N + 1) \% (\text{last} - \text{first}))$ 中的元素转而存储到 $*(\text{first} + N)$ 中，那么我们就可以说函数对序列加了旋转，或是对 $\text{middle} - \text{first}$ 个元素加以左移，或是对 $\text{last} - \text{middle}$ 个元素加以右移。在此， $[\text{first}, \text{middle})$ 以及 $[\text{middle}, \text{last})$ 都必须是有效的区间。

该函数返回指向结果序列末端的迭代器值。

如果 x 和 first 指定存储空间的区域，那么区间 $[\text{x}, \text{x} + (last - first))$ 不能和区间 $[\text{first}, \text{last})$ 重叠。

口 search

```
template<class FwdIt1, class FwdIt2>
FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
```

```

        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
                    FwdIt2 first2, FwdIt2 last2, Pred pr);

```

第一个模板函数会检测区间 $[0, (last1 - first1) - (last2 - first2)]$ 中的最小的 N，使得对于区间 $[0, last2 - first2]$ 中的每个 M，谓词 $*(first1 + N + M) == *(first2 + M)$ 都为 true。在此，operator==必须对它的操作数进行相等关系的判断。函数然后返回 first1 + N。如果不存在这样的 N，函数将返回 last1。它计算谓词的次数最多为 $(last2 - first2) * (last1 - first1)$ 次。

第二个模板函数的行为和第一个相同，不过它用的谓词是 pr($*(first + N + M), *(first2 + M)$)。

口 search_n

```

template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
                     Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
                     Dist n, const T& val, Pred pr);

```

第一个模板函数会检测区间 $[0, (last - first) - n]$ 中最小的 N，使得对于区间 $[0, n]$ 中的每个 M，谓词 $*(first + N + M) == val$ 都为 true。在此，operator==必须对它的操作数进行相等关系的判断。函数然后返回 first + N。如果不存在这样的 N，函数将返回 last。它计算谓词的次数最多为 $n * (last - first)$ 次。

第二个模板函数的行为和第一个相同，不过它用的谓词是 pr($*(first + N + M), val$)。

口 set_difference

```

template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                           InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
         class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
                           InIt2 first2, InIt2 last2, OutIt x, Pred pr);

```

第一个模板函数会交替地从由区间[first1, last1)和区间[first2, last2)中的迭代器指定的序列（它们都是按 operator<排序的有序序列）中复制值，最终产生一个长度为 K、起始于 x 的合并序列，该序列同样也是按 operator<排序的。函数然后返回 x + K。

合并并不会导致这两个序列中元素的相对顺序发生改变。而且，对

于不同序列中的两个具有相等次序的元素，函数并不是把它们复制到相邻的位置上，而是一个也不复制。对于一个序列中存在的与另一个序列中所有元素都没有相等次序关系的元素，函数将会仅仅从区间[first1, last1)中复制这样的元素而跳过另一个区间中具有同样特性的元素。也就是说，函数会将两个有序序列合并为另外一个有序序列，它的内容就是这两个有序序列的差集^①。

如果 x 和 first1 指定存储空间的区域，那么区间 $[x, x + K)$ 不能和区间 $[\text{first1}, \text{last1})$ 重叠。如果 x 和 first2 指定存储空间的区域，那么区间 $[x, x + K)$ 不能和区间 $[\text{first2}, \text{last2})$ 重叠。函数计算次序谓词 $X < Y$ 的次数最多为 $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

¶ set_intersection

```
template<class Init1, class Init2, class OutIt>
    OutIt set_intersection(Init1 first1, Init1 last1,
                           Init2 first2, Init2 last2, OutIt x);
template<class Init1, class Init2, class OutIt,
         class Pred>
    OutIt set_intersection(Init1 first1, Init1 last1,
                           Init2 first2, Init2 last2, OutIt x, Pred pr);
```

第一个模板函数会交替地从由区间 $[\text{first1}, \text{last1})$ 和区间 $[\text{first2}, \text{last2})$ 中的迭代器指定的按 $\text{operator}<$ 排序的有序序列中复制值，最终产生一个长度为 K 、起始于 x 的合并序列，该序列同样也是按 $\text{operator}<$ 排序的。函数然后返回 $x + K$ 。

合并并不会导致这两个序列中元素的相对顺序发生改变。而且，对于不同序列中的两个有相等次序的元素，函数并不是把它们复制到相邻的位置上，而是仅仅从有序区间 $[\text{first1}, \text{last1})$ 中复制元素而跳过另外那个元素。对于一个序列中存在的与另一个序列中所有元素都没有相等次序的元素，函数同样也会跳过它。也就是说，函数将会就两个有序序列产生另外一个有序序列，它的内容就是这两个有序序列的交集。

如果 x 和 first1 指定存储空间的区域，那么区间 $[x, x + K)$ 不能和区间 $[\text{first1}, \text{last1})$ 重叠。如果 x 和 first2 指定存储空间的区域，那么区间 $[x, x + K)$ 不能和区间 $[\text{first2}, \text{last2})$ 重叠。函数计算次序谓词 $X < Y$ 的次数最多为 $2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

^① 即出现于第一个序列中而不出现于第二个序列中的元素的集合。——译者注

```
# set_symmetric_difference
    template<class InIt1, class InIt2, class OutIt>
        OutIt set_symmetric_difference(InIt1 first1,
            InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
    template<class InIt1, class InIt2, class OutIt,
        class Pred>
        OutIt set_symmetric_difference(InIt1 first1,
            InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
            Pred pr);
```

第一个模板函数会交替地从由区间[first1, last1)和区间[first2, last2)中的迭代器指定的按 operator<排序的有序序列中复制值，最终产生一个长度为 K、起始于 x 的合并序列，该序列同样也是按 operator<排序的。函数然后返回 x + K。

合并并不会导致这两个序列中元素的相对顺序发生改变。而且，对于不同序列中的两个相等次序的元素，函数并不是把它们复制到相邻的位置上，而是一个也不复制。对于一个序列中存在的与另一个序列中所有元素都没有相等次序关系的元素，函数将会把它复制到目标序列中去。也就是说，函数将会就两个有序序列产生另外一个有序序列，它的内容就是这两个有序序列的对称差集^②。

如果 x 和 first1 指定存储空间的区域，那么区间[x, x + K)不能和区间[first1, last1)重叠。如果 x 和 first2 指定存储空间的区域的话，那么区间[x, x + K)不能和区间[first2, last2)重叠。函数计算次序谓词 X < Y 的次数最多为 $2 * ((last1 - first1) + (last2 - first2)) - 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

```
# set_union
    template<class InIt1, class InIt2, class OutIt>
        OutIt set_union(InIt1 first1, InIt1 last1,
            InIt2 first2, InIt2 last2, OutIt x);
    template<class InIt1, class InIt2, class OutIt,
        class Pred>
        OutIt set_union(InIt1 first1, InIt1 last1,
            InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

第一个模板函数会交替地从由区间[first1, last1)和区间[first2, last2)中的迭代器指定的按 operator<排序的有序序列中复制值，最终产生一个

^② 即“出现于第一个序列中而不出现于第二个序列中”的元素以及“出现于第二个序列中而不出现于第一个序列中”的元素的合集。——译者注

长度为 K、起始于 x 的合并序列，该序列同样也是按 operator<排序的。函数然后返回 $x + K$ 。

合并并不会导致这两个序列中元素的相对顺序发生改变。而且，对于不同序列中的两个相等次序的元素，函数并不是把它们复制到相邻的位置上，而是仅仅从有序区间 [first1, last1) 中复制元素而跳过另外那个序列中的元素。也就是说，函数将会就两个有序序列产生另外一个有序序列，它的内容就是这两个有序序列的合集。

如果 x 和 first1 指定存储空间的区域，那么区间 [x, x + K) 不能和区间 [first1, last1) 重叠。如果 x 和 first2 指定存储空间的区域，那么区间 [x, x + K) 不能和区间 [first2, last2) 重叠。函数计算次序谓词 X < Y 的次数最多为 $2 * ((last1 - first1) + (last2 - first2)) - 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 sort

```
template<class RanIt>
void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

第一个模板函数会重排由区间 [first, last) 中的迭代器指定的序列，产生一个按 operator< 排序的序列。也就是说，序列中的元素以升序的方式出现在序列中。

函数计算次序谓词 X < Y 的次数最多是 $(last - first) * \log(last - first)$ 的常数倍。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

口 sort_heap

```
template<class RanIt>
void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
void sort_heap(RanIt first, RanIt last, Pred pr);
```

第一个模板函数会重排由区间 [first, last) 中的迭代器指定的序列，产生一个按 operator< 排序的序列。最初的序列必须指定一个同样按 operator< 排序的堆。也就是说，元素以升序的方式出现在序列中。

函数计算次序谓词 X < Y 的次数最多是 $\text{ceil}((last - first) * \log(last - first))$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了 operator<(X, Y)。

```
# stable_partition
    template<class BidIt, class Pred>
        BidIt stable_partition(BidIt first, BidIt last,
                               Pred pr);
```

该模板函数会重排由区间[first, last)中的迭代器指定的序列，然后检测其中 K 的值，使得对于区间[0, K)中的每个值 N，谓词 $\text{pr}(*(\text{first} + N))$ 都为 true；而对于区间[K, last - first)中的每个值 N，谓词 $\text{pr}(*(\text{first} + N))$ 都为 false。这样做并不会改变区间[0, K)或是区间[K, last - first)中的元素之间原有的相对顺序。函数最终返回 $\text{first} + K$ 。

谓词不能改变它的操作数。函数调用 $\text{pr}(*(\text{first} + N))$ 的次数正好为 $\text{last} - \text{first}$ 次，交换元素对的次数最多为 $\text{ceil}((\text{last} - \text{first}) * \log(\text{last} - \text{first}))$ 次。（在有足够的临时存储空间的情况下，它最多只需要交换 $2 * (\text{last} - \text{first})$ 次）。

```
# stable_sort
    template<class BidIt>
        void stable_sort(BidIt first, BidIt last);
    template<class BidIt, class Pred>
        void stable_sort(BidIt first, BidIt last, Pred pr);
```

第一个模板函数会重排由区间[first, last)中的迭代器指定的序列，产生一个按 $\text{operator}<$ 排序的序列。这样做并不会改变最初在序列中拥有相等次序关系的元素之间的相对顺序。也就是说，元素以升序的方式出现在序列中。

函数计算次序谓词 $X < Y$ 的次数最多是 $(\text{last} - \text{first}) * (\log(\text{last} - \text{first}))^2$ 的常数倍。（在有足够的临时存储空间的情况下，函数最多只需要调用谓词 $(\text{last} - \text{first}) * \log(\text{last} - \text{first})$ 的常数倍。）

第二个模板函数的行为和第一个相同，不过它用 $\text{pr}(X, Y)$ 替代了 $\text{operator}<(X, Y)$ 。

```
# swap
    template<class T>
        void swap(T& x, T& y);
```

该模板函数将原来存储于 y 中的值转而存储到 x 中，存储于 x 中的值转而存储到 y 中。

```
# swap_ranges
    template<class FwdIt1, class FwdIt2>
        FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
                           FwdIt2 x);
```

该模板函数会对区间[0, last - first)中的每个N调用一次swap(*(first + N), *(x + N))。函数最终返回x + (last - first)。如果x和first指定存储空间的区域，那么区间[x, x + (last - first))不能和区间[first, last)重叠。

口 transform

```
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
                    Unop uop);
template<class InIt1, class InIt2, class OutIt,
         class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
                     InIt2 first2, OutIt x, Binop bop);
```

第一个模板函数会对区间[0, last - first)中的每个N计算一次*(x + N) = uop(*(first + N))。函数最终返回x + (last - first)。函数调用uop(*(first + N))不能改变*(first + N)。

第二个模板函数会对区间[0, last1 - first1)中的每个N计算一次*(x + N) = bop(*(first1 + N), *(first2 + N))。函数最终返回x + (last1 - first1)。函数调用bop(*(first1 + N), *(first2 + N))不能改变*(first1 + N)或*(first2 + N)。

口 unique

```
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
```

第一个模板函数先将first赋给X，然后对区间[0, last - first)中的每个N执行一遍下列语句：

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *X++ = V;
```

函数最终返回X。也就是说，函数在不停地移除符合*(first + N) == *(first + N - 1)的元素对中的第二个元素，直到只剩下该相等元素序列中的第一个元素为止。在此，operator==必须对它的操作数进行相等关系的判断。函数这样做并不会改变其他元素之间的相对顺序，它返回一个指向结果序列末端的迭代器值。函数计算谓词的次数最多为last - first次。

第二个模板函数的行为和第一个相同，不过它执行的语句是这样的：

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *X++ = V;
```

```
# unique_copy
    template<class InIt, class OutIt>
        OutIt unique_copy(InIt first, InIt last, OutIt x);
    template<class InIt, class OutIt, class Pred>
        OutIt unique_copy(InIt first, InIt last, OutIt x,
                           Pred pr);
```

第一个模板函数会对区间[0, last - first)中的每个N执行一遍下列语句：

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *x++ = V;
```

函数最终返回 x。也就是说，函数会不停地在它所复制的目标序列中移除掉那些符合 $*(\text{first} + N) == *(\text{first} + N - 1)$ 的元素对中的第二个元素，直到只剩下该相等元素序列中的第一个元素为止。在此，operator==必须对它的操作数进行相等关系的判断。函数这样做并不会改变其他元素之间的相对顺序，它返回一个指向复制序列末端的迭代值。

如果 x 和 first 指定存储空间的区域，那么区间[x, x + (last - first))不能和区间[first, last)重叠。

第二个模板函数的行为和第一个相同，不过它执行的语句是这样的：

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *x++ = V;
```

upper_bound

```
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
                        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
                        const T& val, Pred pr);
```

当区间[first, last)中的迭代器所指定的序列是按 operator<排序的有序序列时，第一个模板函数会检测位于区间(0, last - first]中的最大的N，使得对于区间[0, N)中的任意M，谓词 $!(\text{val} < *(\text{first} + M))$ 都为 true。然后函数会返回 first + N。也就是说，该函数会检测一个最后面的位置，使得将 val 插入到它的前面时，序列中已有的次序不会被破坏。

该函数计算次序谓词 X < Y 的次数最多为 $\text{ceil}(\log(\text{last} - \text{first})) + 1$ 次。

第二个模板函数的行为和第一个相同，不过它用 pr(X, Y) 替代了

operator<(X, Y)。

使用<algorithm>

将头文件<algorithm>包含到程序中，我们就可以开始使用它所定义的大量模板函数了。下面所给出的摘要仅仅给出了每个算法的最简短说明，给出了可用算法的综述。请注意，下面所涉及到的计数器 *n* 可能是一个特定的整型参数值，也可能是两个迭代器参数之间的差距。你可以从前一小节中的以字母顺序给出的列表中得到更详细的描述。

只有少数几个定义于<algorithm>中的模板函数处理成对的元素，其他的都作用于序列上。这些函数包括：

max

min

要检测两个元素中的较大值（使用 operator<或是函数对象），调用 max。

要检测两个元素中的较小值（使用 operator<或是函数对象），调用 min。

swap

要交换两个已存储的值，调用 swap。

iter_swap

要交换两个由迭代器所指定的存储值，调用 iter_swap。

许多模板函数会以各种方法扫描序列而不改变它们：

max_element

要检测序列中的最大值（使用 operator<或是函数对象），调用 max_element。

min_element

要检测序列中的最小值（使用 operator<或是函数对象），调用 min_element。

equal

要比较两个序列是否相等（按照元素的存储顺序以 operator==或是函数对象挨个进行比较），调用 equal。

lexicographical_

要比较两个序列中一个序列是否排在另一个前面（按照元素的存储顺序以 operator<或是函数对象挨个进行比较），调用 lexicographical_compare。

mismatch

要检测两个序列中出现的第一处不相等的地方（使用 operator<或是函数对象），调用 mismatch。

find

要检测序列中第一个等于指定值的元素，调用 find。

find_if

要检测序列中第一个使得 pr(x)返回 true 的元素(pr 是一个函数对象)，调用 find_if。

adjacent_

要检测第一对相等的相邻元素（使用 operator==或是函数对象），调用 adjacent_find。

count

要计算在序列中等于一个指定值的元素的个数，调用 count。

count_if

要计算在序列中使得 pr(x)返回 true 的元素的个数(pr 是一个函数对象)，调用 count_if。

search

要在另外一个序列中检测一个序列第一次出现的地方（使用 operator==或是函数对象），请调用 search。

search_n

要在一个序列中检测第一个“连续出现 n 次指定值”的地方（使用 operator==或是函数对象），请调用 search_n。

<code>find_end</code>	要在另外一个序列中检测一个序列最后出现的地方（使用 operator== 或是函数对象），请调用 <code>find_end</code> 。
<code>find_first_of</code>	要在序列中检测另一个序列中任意元素第一次出现的地方（使用 operator== 或是函数对象），请调用 <code>find_first_of</code> 。
<code>for_each</code>	有些模板函数会对序列中的每个元素进行一次特定的操作：
<code>generate</code>	要使用函数对象 <code>op</code> ，对序列中的每个元素 <code>x</code> 都调用 <code>op(x)</code> ，请调用 <code>for_each</code> 。
<code>generate_n</code>	要使用函数对象 <code>fun</code> ，对序列中每个元素都赋值为 <code>fun()</code> ，请调用 <code>generate</code> 。
<code>transform</code>	要使用函数对象 <code>fun</code> ，从一个已有的序列中，对每个元素 <code>x</code> 调用 <code>op(x)</code> 并将返回值赋给另外一个序列中的相应元素，请调用 <code>transform</code> 。我们同样也可以调用 <code>transform</code> ，通过对两个已有的序列中相应的元素 <code>x</code> 和 <code>y</code> 调用 <code>op(x,y)</code> ，来为一个序列赋值。
<code>copy</code>	有些模板函数会将一个已有的序列（或是一个值的重复）赋给另一个序列：
<code>copy_backward</code>	要将一个序列从头到尾地复制到另外一个序列，调用 <code>copy</code> 。 要将一个序列从尾到头地复制到另外一个序列，调用 <code>copy_backward</code> 。
<code>fill</code>	要将一个特定的值赋给序列中的每个元素，调用 <code>fill</code> 。
<code>fill_n</code>	要将一个特定的值赋给序列中的前 <code>n</code> 个元素，调用 <code>fill_n</code> 。
<code>swap_ranges</code>	要交换两个序列中所存储的值，调用 <code>swap_ranges</code> 。
<code>replace</code>	有些模板函数会将序列中的一个特定值替换成另外一个特定值：
<code>replace_if</code>	要将序列中等于一个特定值的所有元素的值都替换成另一个特定值，调用 <code>replace</code> 。 要使用函数对象 <code>pr</code> 将序列中所有使得 <code>pr(x)</code> 返回 <code>true</code> 的元素 <code>x</code> 的值都替换成一个特定值，调用 <code>replace_if</code> 。
<code>replace_copy</code>	要复制一个序列并将序列中等于一个特定值的所有元素的值都替换成另一个特定值，调用 <code>replace_copy</code> 。
<code>replace_copy_if</code>	要使用函数对象 <code>pr</code> 复制一个序列并将序列中所有使得 <code>pr(x)</code> 返回 <code>true</code> 的元素 <code>x</code> 的值都替换成一个特定值，调用 <code>replace_copy_if</code> 。
<code>remove</code>	有几个模板函数会从序列中移除掉 0 个或多个元素：
<code>remove_if</code>	要移除掉所有等于一个特定值的元素，调用 <code>remove</code> 。 要使用函数对象 <code>pr</code> 移除掉所有使得 <code>pr(x)</code> 返回 <code>true</code> 的元素 <code>x</code> ，调用 <code>remove_if</code> 。
<code>remove_copy</code>	要复制一个序列并从中移除掉所有等于一个特定值的元素，调用 <code>remove_copy</code> 。

remove_copy_if	要使用函数对象 pr 复制一个序列并从中移除掉所有使得 pr(x) 返回 true 的元素 x，调用 remove_copy_if。
unique	要移除掉一个相等元素子序列中除去第一个元素以外的所有元素（使用 operator== 或是函数对象），调用 unique。
unique_copy	要复制一个序列并移除掉里面所有相等元素子序列中除第一个元素外的所有元素（使用 operator== 或是函数对象），调用 unique_copy。
	有些模板函数会改变序列中元素的顺序：
reverse	要将一个序列反转，调用 reverse。
reverse_copy	要复制一个序列并将其反转，调用 reverse_copy。
rotate	要在位置 n 旋转序列中的元素，调用 rotate。
rotate_copy	要复制一个序列并在位置 n 旋转它的元素，调用 rotate_copy。
random_shuffle	要对序列中的元素进行随机重排，调用 random_shuffle。
	此外还有一种特殊形式的重排会作用于一个序列上，它们将导致序列的排列顺序与最初的排列顺序有些不同：
partition	要使用函数对象 pr 将所有使得 pr(x) 返回 true 的元素 x 移到序列的开始处，调用 partition。
stable_partition	要按照上述方式分割序列且又不破坏每个分割中原有（未分割前）元素之间的顺序，调用 stable_partition。
sort	要使得序列中的元素以升序方式存储（使用 operator< 或是函数对象），调用 sort。
stable_sort	要按照上述方式对序列进行排序且又不破坏序列元素之间原有的顺序，调用 stable_sort。
partial_sort	要仅将最小的 n 个元素以升序的方式排序（使用 operator< 或是函数对象）并将它们移到序列的开始处，调用 partial_sort。
partial_sort_copy	要复制一个序列并按照上述的方式仅对最大的 n 个元素进行排序，调用 partial_sort_copy。
nth_element	要将元素 n 放置在符合升序顺序的位置处（使用 operator< 或是函数对象），调用 nth_element。所有在 n 前面的元素都应该小于它，所有在它后面的元素都应该大于它，但对于这些元素没有任何排序要求。
	有两个模板函数会将两个有序序列合并起来产生另外一个有序序列：
merge	要将两个有序序列合并并产生一个新的序列（使用 operator< 或是函数对象），调用 merge。
inplace_merge	要在适当的位置上合并两个有序序列（使用 operator< 或是函数对象），调用 inplace_merge。
	有些模板函数会扫描一个以升序方式存储元素的有序序列（按 operator< 或是函数对象排序）：
lower_bound	要在有序序列中检测第一个不小于给出的特定值的元素位置（使用 operator< 或是函数对象），调用 lower_bound。

<code>upper_bound</code>	要在有序序列中检测最后一个不小于给出的特定值的元素位置（使用 <code>operator<</code> 或是函数对象），调用 <code>upper_bound</code> 。
<code>equal_range</code>	要检测上述有序序列中对于一个特定值的第一个和最后一个不小于它的边界，调用 <code>equal_range</code> 。
<code>binary_search</code>	要在有序序列中检测是否有与一个特定值次序相等的元素（使用 <code>operator<</code> 或是函数对象），调用 <code>binary_search</code> 。请注意“相等次序”并不一定意味着相等。
<code>includes</code>	有些模板函数会要求两个以升序方式存储元素的有序序列（按 <code>operator<</code> 或是函数对象排序）作为其输入：
<code>set_union</code>	要检测一个有序序列是否包含与另一个序列中的每个元素相等的元素，调用 <code>includes</code> 。
<code>set_intersection</code>	要合并两个有序序列并产生一个新的序列（该新序列中将不保留第二个序列中所存在的与第一个序列中某些元素次序相等的元素），调用 <code>set_union</code> 。该操作大致等同于对两个元素集合取合集的操作。
<code>set_difference</code>	要合并两个有序序列并产生一个新的序列（该新序列中将仅保留第一个序列中所存在的与第二个序列中某些元素次序相等的元素），调用 <code>set_intersection</code> 。该操作大致等同于对两个元素集合取交集的操作。
<code>set_symmetric_difference</code>	要合并两个有序序列并产生一个新的序列（该新序列中将仅保留第一个序列中所存在的与第二个序列中所有元素都没有相等次序关系的元素），调用 <code>set_difference</code> 。该操作大致等同于对两个元素集合取差集的操作。
<code>make_heap</code>	要合并两个有序序列并产生一个新的序列（该新序列中将仅保留那些与另外一个序列中所有元素都没有相等次序关系的元素），调用 <code>set_symmetric_difference</code> 。该操作大致等同于对两个元素集合取对称差集的操作。
<code>push_heap</code>	有些模板函数会将序列作为堆来处理（它们的值按照 <code>operator<</code> 或是函数对象来排序）：
<code>pop_heap</code>	要重排一个序列以得到一个堆（使用 <code>operator<</code> 或是函数对象），调用 <code>make_heap</code> 。
<code>sort_heap</code>	要向堆中新增一个元素（使用 <code>operator<</code> 或是函数对象），调用 <code>push_heap</code> 。
<code>next_permutation</code>	要从堆中移除掉值最大的那个元素（使用 <code>operator<</code> 或是函数对象），调用 <code>pop_heap</code> 。
	要对堆进行排序，以产生一个以升序方式存储元素的序列（使用 <code>operator<</code> 或是函数对象），调用 <code>sort_heap</code> 。
	最后，还有两个模板函数会对序列中的元素进行排序（按照 <code>operator<</code> 或函数对象来排序）：
	要改变序列的排列，调用 <code>next_permutation</code> 。当序列中的所有元素已经是按升序排列时，该模板函数会返回 <code>false</code> 。

prev_permutation 要改变序列的排列，也可以调用 `prev_permutation`。当序列中的所有元素已经是按降序排列时，该模板函数会返回 `false`。注意，该模板函数产生的排列和 `next_permutation` 所产生的正好相反。

实现<algorithm>

algorithm 我们已经在第 3 章中列出了一些和头文件<algorithm>有关的算法。这个实现将许多模板函数都放置在一个内部头文件<xutility>中。在这个头文件中，可以找到的模板函数有：`copy`、`copy_backward`、`equal`、`fill`、`fill_n`、`lexicographical_compare`、`mismatch`、`max`、`min` 以及 `swap`。在这些模板函数中，只有后三个的操作对象是一对元素，其他的都是作用在元素序列上面。所有这些模板函数都不很复杂，它们中只有一个值得我们稍微关注一下。

lexicographical_compare 模板函数 `lexicographical_compare` 给我们阐明了“相等”和“相等次序”的区别。如果 `x == y`，那么我们就认为 `x` 和 `y` 是相等的；如果 `!(x < y) && !(y < x)`，则 `x` 和 `y` 就是相等次序的。请注意的一点是：相等次序关系纯粹是根据 `operator<` 定义出来的。对于整数类型来说，相等关系和相等次序关系是完全一样的，但这并不意味着对于其他类型来说也是如此。

程序清单 6-1 中列出了文件 `algorithm`，它实现了<algorithm>中剩下的算法。由于该头文件实在是太大了，我们不得不限制代码中的注释数量，所有的注释只是起着突出作用。和 `xutility` 中的模板函数一样，`algorithm` 中的大部分代码都只保留了很少的注释。

```
程序清单 6-1: // algorithm standard header
algorithm ifndef ALGORITHM_
#define ALGORITHM_
#include <memory>
namespace std {
    // COMMON SORT PARAMETERS
    const int CHUNK_SIZE = 7;
    const int SORT_MAX = 16;

    // TEMPLATE FUNCTION for_each
    template<class InIt, class Fn> inline
        Fn for_each(InIt F, InIt L, Fn Op)
    {for (; F != L; ++F)
        Op(*F);
    return (Op); }

    // TEMPLATE FUNCTION find
    template<class InIt, class T> inline
```

```

        InIt find(InIt F, InIt L, const T& V)
        {for (; F != L; ++F)
         if (*F == V)
             break;
        return (F); }

                // TEMPLATE FUNCTION find_if
template<class InIt, class Pr> inline
    InIt find_if(InIt F, InIt L, Pr P)
    {for (; F != L; ++F)
     if (P(*F))
         break;
    return (F); }

                // TEMPLATE FUNCTION adjacent_find
template<class FwdIt> inline
    FwdIt adjacent_find(FwdIt F, FwdIt L)
    {for (FwdIt Fb; (Fb = F) != L && ++F != L; )
     if (*Fb == *F)
         return (Fb);
    return (L); }

                // TEMPLATE FUNCTION adjacent_find WITH PRED
template<class FwdIt, class Pr> inline
    FwdIt adjacent_find(FwdIt F, FwdIt L, Pr P)
    {for (FwdIt Fb; (Fb = F) != L && ++F != L; )
     if (P(*Fb, *F))
         return (Fb);
    return (L); }

                // TEMPLATE FUNCTION count
template<class InIt, class T> inline
    typename iterator_traits<InIt>::difference_type
        count(InIt F, InIt L, const T& V)
    {typename iterator_traits<InIt>::difference_type
        N = 0;

    for (; F != L; ++F)
        if (*F == V)
            ++N;
    return (N); }

                // TEMPLATE FUNCTION count_if
template<class InIt, class Pr> inline
    typename iterator_traits<InIt>::difference_type

```

```
        count_if(Init F, Init L, Pr P)
        {typename iterator_traits<Init>::difference_type
         N = 0;
        for (; F != L; ++F)
            if (P(*F))
                ++N;
        return (N); }

        // TEMPLATE FUNCTION search
template<class FwdIt1, class FwdIt2> inline
    FwdIt1 search(FwdIt1 F1, FwdIt1 L1,
                  FwdIt2 F2, FwdIt2 L2)
{return (Search(F1, L1, F2, L2,
               Dist_type(F1), Dist_type(F2))); }
template<class FwdIt1, class FwdIt2, class Pd1,
         class Pd2> inline
    FwdIt1 Search(FwdIt1 F1, FwdIt1 L1, FwdIt2 F2,
                  FwdIt2 L2, Pd1 *, Pd2 *)
{Pd1 D1 = 0;
Distance(F1, L1, D1);
Pd2 D2 = 0;
Distance(F2, L2, D2);
for (; D2 <= D1; ++F1, --D1)
    {FwdIt1 X1 = F1;
     for (FwdIt2 X2 = F2; ; ++X1, ++X2)
         if (X2 == L2)
             return (F1);
         else if (!(*X1 == *X2))
             break; }
return (L1); }

        // TEMPLATE FUNCTION search WITH PRED
template<class FwdIt1, class FwdIt2, class Pr> inline
    FwdIt1 search(FwdIt1 F1, FwdIt1 L1,
                  FwdIt2 F2, FwdIt2 L2, Pr P)
{return (Search(F1, L1, F2, L2, P,
               Dist_type(F1), Dist_type(F2))); }
template<class FwdIt1, class FwdIt2, class Pd1, class
         Pd2, class Pr> inline
    FwdIt1 Search(FwdIt1 F1, FwdIt1 L1, FwdIt2 F2,
                  FwdIt2 L2, Pr P, Pd1 *, Pd2 *)
{Pd1 D1 = 0;
Distance(F1, L1, D1);
Pd2 D2 = 0;
Distance(F2, L2, D2);
```

```

        for (; D2 <= D1; ++F1, --D1)
            {FwdIt1 X1 = F1;
             for (FwdIt2 X2 = F2; ; ++X1, ++X2)

                 if (X2 == L2)
                     return (F1);
                 else if (!P(*X1, *X2))
                     break; }
             return (L1); }

         // TEMPLATE FUNCTION search_n
template<class FwdIt1, class Pd2, class T> inline
FwdIt1 search_n(FwdIt1 F1, FwdIt1 L1, Pd2 N, const T& V)
{return (Search_n(F1, L1, N, V, Dist_type(F1))); }

template<class FwdIt1, class Pd2, class T, class Pd1>
inline FwdIt1 Search_n(FwdIt1 F1, FwdIt1 L1,
Pd2 N, const T& V, Pd1 *)
{Pd1 D1 = 0;
Distance(F1, L1, D1);
for (; N <= D1; ++F1, --D1)
    {FwdIt1 X1 = F1;
     for (Pd2 D2 = N; ; ++X1, --D2)
         if (D2 == 0)
             return (F1);
         else if (!(*X1 == V))
             break; }
     return (L1); }

         // TEMPLATE FUNCTION search_n WITH PRED
template<class FwdIt1, class Pd2, class T, class Pr>
inline FwdIt1 search_n(FwdIt1 F1, FwdIt1 L1,
Pd2 N, const T& V, Pr P)
{return (Search_n(F1, L1,
N, V, P, Dist_type(F1)));}
template<class FwdIt1, class Pd2,
class T, class Pd1, class Pr> inline
FwdIt1 Search_n(FwdIt1 F1, FwdIt1 L1,
Pd2 N, const T& V, Pr P, Pd1 *)
{Pd1 D1 = 0;
Distance(F1, L1, D1);
for (; N <= D1; ++F1, --D1)
    {FwdIt1 X1 = F1;
     for (Pd2 D2 = N; ; ++X1, --D2)
         if (D2 == 0)
             return (F1);
         else if (!(*X1 == V))
             break; }
     return (L1); }
}

```

```
        return (F1);
    else if (!P(*X1, V))
        break;
    return (L1); }

// TEMPLATE FUNCTION find_end
template<class FwdIt1, class FwdIt2> inline
    FwdIt1 find_end(FwdIt1 F1, FwdIt1 L1,
                    FwdIt2 F2, FwdIt2 L2)
{return (Find_end(F1, L1, F2, L2,
                  Dist_type(F1), Dist_type(F2))); }

template<class FwdIt1, class FwdIt2, class Pd1, class Pd2>
inline
    FwdIt1 Find_end(FwdIt1 F1, FwdIt1 L1,
                    FwdIt2 F2, FwdIt2 L2, Pd1 *, Pd2 *)
{Pd1 D1 = 0;

Distance(F1, L1, D1);
Pd2 D2 = 0;
Distance(F2, L2, D2);
FwdIt1 Ans = L1;
if (0 < D2)
    for (; D2 <= D1; ++F1, --D1)
        (FwdIt1 X1 = F1;
         for (FwdIt2 X2 = F2; ; ++X1)
             if (!(*X1 == *X2))
                 break;
             else if (++X2 == L2)
                 {Ans = F1;
                  break; })
return (Ans); }

// TEMPLATE FUNCTION find_end WITH PRED
template<class FwdIt1, class FwdIt2, class Pr> inline
    FwdIt1 find_end(FwdIt1 F1, FwdIt1 L1,
                    FwdIt2 F2, FwdIt2 L2, Pr P)
{return (Find_end(F1, L1, F2, L2, P,
                  Dist_type(F1), Dist_type(F2))); }

template<class FwdIt1, class FwdIt2, class Pd1,
         class Pd2, class Pr> inline
    FwdIt1 Find_end(FwdIt1 F1, FwdIt1 L1,
                    FwdIt2 F2, FwdIt2 L2, Pr P, Pd1 *, Pd2 *)
{Pd1 D1 = 0;
Distance(F1, L1, D1);
```

```

Pd2 D2 = 0;
Distance(F2, L2, D2);
FwdIt1 Ans = L1;
if (0 < D2)
    for (; D2 <= D1; ++F1, --D1)
        {FwdIt1 X1 = F1;
         for (FwdIt2 X2 = F2; ; ++X1)
             if (!P(*X1, *X2))
                 break;
             else if (++X2 == L2)
                 {Ans = F1;
                  break; })
return (Ans); }

// TEMPLATE FUNCTION find_first_of
template<class FwdIt1, class FwdIt2> inline
FwdIt1 find_first_of(FwdIt1 F1, FwdIt1 L1,
                     FwdIt2 F2, FwdIt2 L2)
{for (; F1 != L1; ++F1)
    for (FwdIt2 X2 = F2; X2 != L2; ++X2)
        if (*F1 == *X2)
            return (F1);
return (F1); }

// TEMPLATE FUNCTION find_first_of WITH PRED
template<class FwdIt1, class FwdIt2, class Pr> inline
FwdIt1 find_first_of(FwdIt1 F1, FwdIt1 L1,
                     FwdIt2 F2, FwdIt2 L2, Pr P)

{for (; F1 != L1; ++F1)
    for (FwdIt2 X2 = F2; X2 != L2; ++X2)
        if (P(*F1, *X2))
            return (F1);
return (F1); }

// TEMPLATE FUNCTION iter_swap
template<class FwdIt1, class FwdIt2> inline
void iter_swap(FwdIt1 X, FwdIt2 Y)
    (Iter_swap(X, Y, Val_type(X)); }

template<class FwdIt1, class FwdIt2, class T> inline
void Iter_swap(FwdIt1 X, FwdIt2 Y, T *)
{T Tmp = *X;
*X = *Y, *Y = Tmp; }

```

```
// TEMPLATE FUNCTION swap_ranges
template<class FwdIt1, class FwdIt2> inline
    FwdIt2 swap_ranges(FwdIt1 F, FwdIt1 L, FwdIt2 X)
{for (; F != L; ++F, ++X)
    iter_swap(F, X);
return (X); }

// TEMPLATE FUNCTION transform WITH UNARY OP
template<class InIt, class OutIt, class Uop> inline
    OutIt transform(InIt F, InIt L, OutIt X, Uop U)
{for (; F != L; ++F, ++X)
    *X = U(*F);
return (X); }

// TEMPLATE FUNCTION transform WITH BINARY OP
template<class InIt1, class InIt2, class OutIt, class Bop>
inline
    OutIt transform(InIt1 F1, InIt1 L1, InIt2 F2,
                    OutIt X, Bop B)
{for (; F1 != L1; ++F1, ++F2, ++X)
    *X = B(*F1, *F2);
return (X); }

// TEMPLATE FUNCTION replace
template<class FwdIt, class T> inline
void replace(FwdIt F, FwdIt L, const T& Vo, const T& Vn)
{for (; F != L; ++F)
    if (*F == Vo)
        *F = Vn; }

// TEMPLATE FUNCTION replace_if
template<class FwdIt, class Pr, class T> inline
void replace_if(FwdIt F, FwdIt L, Pr P, const T& V)
{for (; F != L; ++F)
    if (P(*F))
        *F = V; }

// TEMPLATE FUNCTION replace_copy
template<class InIt, class OutIt, class T> inline
    OutIt replace_copy(InIt F, InIt L, OutIt X,
                        const T& Vo, const T& Vn)

{for (; F != L; ++F, ++X)
    *X = *F == Vo ? Vn : *F;
```

```
        return (X); }

        // TEMPLATE FUNCTION replace_copy_if
template<class InIt, class OutIt, class Pr, class T>
    inline OutIt replace_copy_if(InIt F, InIt L,
        OutIt X, Pr P, const T& V)
    {for (; F != L; ++F, ++X)
        *X = P(*F) ? V : *F;
    return (X); }

        // TEMPLATE FUNCTION generate
template<class FwdIt, class Gen> inline
    void generate(FwdIt F, FwdIt L, Gen G)
    {for (; F != L; ++F)
        *F = G(); }

        // TEMPLATE FUNCTION generate_n
template<class OutIt, class Pd, class Gen> inline
    void generate_n(OutIt F, Pd N, Gen G)
    {for (; 0 < N; --N, ++F)
        *F = G(); }

        // TEMPLATE FUNCTION remove
template<class FwdIt, class T> inline
    FwdIt remove(FwdIt F, FwdIt L, const T& V)
    {F = find(F, L, V);
    if (F == L)
        return (F);
    else
        {FwdIt Fb = F;
        return (remove_copy(++F, L, Fb, V)); }}

        // TEMPLATE FUNCTION remove_if
template<class FwdIt, class Pr> inline
    FwdIt remove_if(FwdIt F, FwdIt L, Pr P)
    {F = find_if(F, L, P);
    if (F == L)
        return (F);
    else
        {FwdIt Fb = F;
        return (remove_copy_if(++F, L, Fb, P)); }}

        // TEMPLATE FUNCTION remove_copy
template<class InIt, class OutIt, class T> inline
    OutIt remove_copy(InIt F, InIt L, OutIt X, const T& V)
```

```
{for (; F != L; ++F)
    if (!(*F == V))
        *X++ = *F;
return (X); }

// TEMPLATE FUNCTION remove_copy_if
template<class InIt, class OutIt, class Pr> inline
OutIt remove_copy_if(InIt F, InIt L, OutIt X, Pr P)

{for (; F != L; ++F)
    if (!P(*F))
        *X++ = *F;
return (X); }

// TEMPLATE FUNCTION unique
template<class FwdIt> inline
FwdIt unique(FwdIt F, FwdIt L)
{F = adjacent_find(F, L);
return (unique_copy(F, L, F)); }

// TEMPLATE FUNCTION unique WITH PRED
template<class FwdIt, class Pr> inline
FwdIt unique(FwdIt F, FwdIt L, Pr P)
{F = adjacent_find(F, L, P);
return (unique_copy(F, L, F, P)); }

// TEMPLATE FUNCTION unique_copy
template<class InIt, class OutIt> inline
OutIt unique_copy(InIt F, InIt L, OutIt X)
{return (F == L ? X :
    Unique_copy(F, L, X, Iter_cat(F))); }
template<class InIt, class OutIt> inline
OutIt Unique_copy(InIt F, InIt L, OutIt X,
    input_iterator_tag)
{return (Unique_copy(F, L, X, Val_type(F))); }
template<class InIt, class OutIt, class T> inline
OutIt Unique_copy(InIt F, InIt L, OutIt X, T *)
{T V = *F;
for (*X++ = V; ++F != L; )
    if (!(V == *F))
        V = *F, *X++ = V;
return (X); }
template<class FwdIt, class OutIt> inline
OutIt Unique_copy(FwdIt F, FwdIt L, OutIt X,
```

```

        forward_iterator_tag)
    {FwdIt Fb = F;
     for (*X++ = *Fb; ++F != L; )
        if (!(*Fb == *F))
            Fb = F, *X++ = *Fb;
     return (X); }
template<class BidIt, class OutIt> inline
    OutIt Unique_copy(BidIt F, BidIt L, OutIt X,
                      bidirectional_iterator_tag)
    {return (Unique_copy(F, L, X, forward_iterator_tag()));}
template<class RanIt, class OutIt> inline
    OutIt Unique_copy(RanIt F, RanIt L, OutIt X,
                      random_access_iterator_tag)
    {return (Unique_copy(F, L, X, forward_iterator_tag()));}

        // TEMPLATE FUNCTION unique_copy WITH PRED
template<class InIt, class OutIt, class Pr> inline
    OutIt unique_copy(InIt F, InIt L, OutIt X, Pr P)
    {return (F == L ? X :
             Unique_copy(F, L, X, P, Iter_cat(F)));}

template<class InIt, class OutIt, class Pr> inline
    OutIt Unique_copy(InIt F, InIt L, OutIt X, Pr P,
                      input_iterator_tag)
    {return (Unique_copy(F, L, X, P, Val_type(F)));}
template<class InIt, class OutIt, class T, class Pr> inline
    OutIt Unique_copy(InIt F, InIt L, OutIt X, Pr P, T *)
    {T V = *F;
     for (*X++ = V; ++F != L; )
        if (!P(V, *F))
            V = *F, *X++ = V;
     return (X); }
template<class FwdIt, class OutIt, class Pr> inline
    OutIt Unique_copy(FwdIt F, FwdIt L, OutIt X, Pr P,
                      forward_iterator_tag)
    {FwdIt Fb = F;
     for (*X++ = *Fb; ++F != L; )
        if (!P(*Fb, *F))
            Fb = F, *X++ = *Fb;
     return (X); }
template<class BidIt, class OutIt, class Pr> inline
    OutIt Unique_copy(BidIt F, BidIt L, OutIt X, Pr P,
                      bidirectional_iterator_tag)
    {return (Unique_copy(F, L, X, P,

```

```
        forward_iterator_tag())); }
template<class RanIt, class OutIt, class Pr> inline
    OutIt Unique_copy(RanIt F, RanIt L, OutIt X, Pr P,
                      random_access_iterator_tag)
{return (Unique_copy(F, L, X, P,
                     forward_iterator_tag())); }

        // TEMPLATE FUNCTION reverse
template<class BidIt> inline
    void reverse(BidIt F, BidIt L)
{Reverse(F, L, Iter_cat(F)); }
template<class BidIt> inline
    voidReverse(BidIt F, BidIt L, bidirectional_iterator_tag)
{for (; F != L && F != --L; ++F)
    iter_swap(F, L); }
template<class RanIt> inline
    void Reverse(RanIt F, RanIt L, random_access_iterator_tag)
{for (; F < L; ++F)
    iter_swap(F, --L); }

        // TEMPLATE FUNCTION reverse_copy
template<class BidIt, class OutIt> inline
    OutIt reverse_copy(BidIt F, BidIt L, OutIt X)
{for (; F != L; ++X)
    *X = *--L;
return (X); }

        // TEMPLATE FUNCTION rotate
template<class FwdIt> inline
    void rotate(FwdIt F, FwdIt M, FwdIt L)
{if (F != M && M != L)
    Rotate(F, M, L, Iter_cat(F)); }

template<class FwdIt> inline
    void Rotate(FwdIt F, FwdIt M, FwdIt L,
                forward_iterator_tag)
{for (FwdIt X = M; ; )
    {iter_swap(F, X);
     if (++F == M)
         if (++X == L)
             break;
     else
         M = X;
     else if (++X == L)
```

```

        X = M; }]
template<class BidIt> inline
    void Rotate(BidIt F, BidIt M, BidIt L,
                bidirectional_iterator_tag)
{reverse(F, M);
 reverse(M, L);
 reverse(F, L); }
template<class RanIt> inline
    void Rotate(RanIt F, RanIt M, RanIt L,
                random_access_iterator_tag)
{Rotate(F, M, L, Dist_type(F), Val_type(F)); }
template<class RanIt, class Pd, class T> inline
    void Rotate(RanIt F, RanIt M, RanIt L, Pd *, T *)
{Pd D = M - F;
 Pd N = L - F;
 for (Pd I = D; I != 0; )
{Pd J = N % I;
 N = I, I = J; }
if (N < L - F)
for (; 0 < N; --N)
{RanIt X = F + N;
 RanIt Y = X;
 T V = *X;
 RanIt Z = Y + D == L ? F : Y + D;
 while (Z != X)
{*Y = *Z;
 Y = Z;
 Z = D < L - Z ? Z + D
 : F + (D - (L - Z)); }
*Y = V; }}

// TEMPLATE FUNCTION rotate_copy
template<class FwdIt, class OutIt> inline
OutIt rotate_copy(FwdIt F, FwdIt M, FwdIt L, OutIt X)
{X = copy(M, L, X);
 return (copy(F, M, X)); }

// TEMPLATE FUNCTION random_shuffle
template<class RanIt> inline
void random_shuffle(RanIt F, RanIt L)
{if (F != L)
 Random_shuffle(F, L, Dist_type(F)); }
template<class RanIt, class Pd> inline
void Random_shuffle(RanIt F, RanIt L, Pd *)

```

```

    {const int RBITS = 15;
    const int RMAX = (1U << RBITS) - 1;
    RanIt X = F;
    for (unsigned long D = 2; ++X != L; ++D)
        {unsigned long Rm = RMAX;
         unsigned long Rn = rand() & RMAX;
         for (; Rm < D && Rm != ~0UL;
              Rm = Rm << RBITS | RMAX)
             Rn = Rn << RBITS | RMAX;
         iter_swap(X, F + Pd(Rn % D)); }
    }

    // TEMPLATE FUNCTION random_shuffle WITH RANDOM FN
template<class RanIt, class Pf> inline
void random_shuffle(RanIt F, RanIt L, Pf& R)
{if (F != L)
    Random_shuffle(F, L, R, Dist_type(F)); }

template<class RanIt, class Pf, class Pd> inline
void Random_shuffle(RanIt F, RanIt L, Pf& R, Pd *)
{RanIt X = F;
for (unsigned long D = 2; ++X != L; ++D)
    iter_swap(X, F + Pd(R(D))); }

    // TEMPLATE FUNCTION partition
template<class BidIt, class Pr> inline
BidIt partition(BidIt F, BidIt L, Pr P)
{for (; ; ++F)
    {for (; F != L && P(*F); ++F)
        ;
        if (F == L)
            break;
        for (; F != --L && !P(*L); )
            ;
        if (F == L)
            break;
        iter_swap(F, L); }
    return (F); }

    // TEMPLATE FUNCTION stable_partition
template<class BidIt, class Pr> inline
BidIt stable_partition(BidIt F, BidIt L, Pr P)
{return (F == L ? F : Stable_partition(F, L, P,
    Dist_type(F), Val_type(F))); }

template<class BidIt, class Pr, class Pd, class T> inline
BidIt Stable_partition(BidIt F, BidIt L, Pr P, Pd *, T *)
{Pd N = 0;

```

```

        Distance(F, L, N);
        Temp_iterator<T> Xb(N);
        return (Stable_partition(F, L, P, N, Xb)); }

template<class BidIt, class Pr, class Pd, class T> inline
    BidIt Stable_partition(BidIt F, BidIt L, Pr P, Pd N,
                           Temp_iterator<T>& Xb)
{if (N == 1)
    return (P(*F) ? L : F);
else if (N <= Xb.Maxlen())
    {BidIt X = F;

        for (Xb.Init(); F != L; ++F)
            if (P(*F))
                *X++ = *F;
            else
                *Xb++ = *F;
        copy(Xb.First(), Xb.Last(), X);
        return (X); }
else
    {BidIt M = F;
    advance(M, N / 2);
    BidIt Lp = Stable_partition(F, M, P, N / 2, Xb);
    BidIt Rp = Stable_partition(M, L, P, N - N / 2, Xb);
    Pd D1 = 0;
    Distance(Lp, M, D1);
    Pd D2 = 0;
    Distance(M, Rp, D2);
    return (Buffered_rotate(Lp, M, Rp, D1, D2, Xb)); }}

// TEMPLATE FUNCTION sort
template<class RanIt> inline
void sort(RanIt F, RanIt L)
{if (L - F <= SORT_MAX)
    Insertion_sort(F, L);
else
    {Sort(F, L, L - F);
    Insertion_sort(F, F + SORT_MAX);
    Sort_end(F + SORT_MAX, L, Val_type(F)); }}

template<class RanIt, class T>
void Sort_end(RanIt F, RanIt L, T *)
{for (; F != L; ++F)
    Unguarded_insert(F, T(*F)); }

template<class RanIt, class Pd> inline
void Sort(RanIt F, RanIt L, Pd Ideal)

```

```

    {for ( ; SORT_MAX < L - F; )
        if (Ideal == 0)
            (make_heap(F, L);
             sort_heap(F, L);
             break; )
        else
            {RanIt M = Unguarded_partition(F, L,
                Val_type(F));
             Sort(F, M, Ideal /= 2);
             F = M; })
    template<class RanIt, class T> inline
        RanIt Unguarded_partition(RanIt F, RanIt L, T *)
    {RanIt M = F + (L - F) / 2;
     if (*M < *F)
         iter_swap(F, M);
     if (*(L - 1) < *M)
         iter_swap(M, L - 1);
     if (*M < *F)
         iter_swap(F, M);
     for (T Piv = *M; ; ++F)
         {for ( ; *F < Piv; ++F)
             ;
             for ( ; Piv < *--L; )
                 ;
                 if (L <= F)
                     return (F);
                 iter_swap(F, L); }
    template<class BidIt> inline
        void Insertion_sort(BidIt F, BidIt L)
    {Insertion_sort_1(F, L, Val_type(F)); }
    template<class BidIt, class T> inline
        void Insertion_sort_1(BidIt F, BidIt L, T *)
    {if (F != L)
        for (BidIt M = F; ++M != L; )
            {T V = *M;
             if (!(V < *F))
                 Unguarded_insert(M, V);
             vlse
                 {BidIt Mpl = M;
                  copy_backward(F, M, ++Mpl);
                  *F = V; }}}
    template<class BidIt, class T> inline
        void Unguarded_insert(BidIt L, T V)

```

```

        {for (BidIt M = L; V < *--M; L = M)
         *L = *M;
        *L = V; }

        // TEMPLATE FUNCTION sort WITH PRED
template<class RanIt, class Pr> inline
void sort(RanIt F, RanIt L, Pr P)
{if (L - F <= SORT_MAX)
    Insertion_sort(F, L, P);
else
    {Sort(F, L, L - F, P);
     Insertion_sort(F, F + SORT_MAX, P);
     Sort_end(F + SORT_MAX, L, P, Val_type(F)); }}

template<class RanIt, class Pr, class T>
void Sort_end(RanIt F, RanIt L, Pr P, T *)
{for (; F != L; ++F)
    Unguarded_insert(F, T(*F), P); }

template<class RanIt, class Pd, class Pr> inline
void Sort(RanIt F, RanIt L, Pd Ideal, Pr P)
{for (; SORT_MAX < L - F; )
    if (Ideal == 0)
        {make_heap(F, L, P);
         sort_heap(F, L, P);
         break; }
    else
        {RanIt M = Unguarded_partition(F, L, P,
                                         Val_type(F));
         Sort(F, M, Ideal /= 2, P);
         F = M; }}

template<class RanIt, class Pr, class T> inline
RanIt Unguarded_partition(RanIt F, RanIt L, Pr P, T *)
{RanIt M = F + (L - F) / 2;
if (P(*M, *F))
    iter_swap(F, M);

if (P(*(L - 1), *M))
    iter_swap(M, L - 1);
if (P(*M, *F))
    iter_swap(F, M);
for (T Piv = *M; ; ++F)
    {for (; P(*F, Piv); ++F)
        ;
        for (; P(Piv, *--L); )
        ;}
}

```

```
        if (L <= F)
            return (F);
        iter_swap(F, L); }
template<class BidIt, class Pr> inline
void Insertion_sort(BidIt F, BidIt L, Pr P)
{Insertion_sort_1(F, L, P, Val_type(F)); }
template<class BidIt, class T, class Pr> inline
void Insertion_sort_1(BidIt F, BidIt L, Pr P, T *)
{if (F != L)
    for (BidIt M = F; ++M != L; )
        {T V = *M;
        if (!P(V, *F))
            Unguarded_insert(M, V, P);
        else
            {BidIt Mpl = M;
            copy_backward(F, M, ++Mpl);
            *F = V; }})
template<class BidIt, class T, class Pr> inline
void Unguarded_insert(BidIt L, T V, Pr P)
{for (BidIt M = L; P(V, *--M); L = M)
    *L = *M;
*L = V; }

// TEMPLATE FUNCTION stable_sort
template<class BidIt> inline
void stable_sort(BidIt F, BidIt L)
{if (F != L)
    Stable_sort(F, L, Dist_type(F), Val_type(F)); }
template<class BidIt, class Pd, class T> inline
void Stable_sort(BidIt F, BidIt L, Pd *, T *)
{Pd N = 0;
Distance(F, L, N);
Temp_iterator<T> Xb(N);
Stable_sort(F, L, N, Xb); }
template<class BidIt, class Pd, class T> inline
void Stable_sort(BidIt F, BidIt L, Pd N,
Temp_iterator<T>& Xb)
{if (N <= SORT_MAX)
    Insertion_sort(F, L);
else
    {Pd N2 = (N + 1) / 2;
BidIt M = F;
advance(M, N2);
if (N2 <= Xb.Maxlen())
    {Buffered_merge_sort(F, M, N2, Xb);}}
```

```

        Buffered_merge_sort(M, L, N - N2, Xb); }
    else
        (Stable_sort(F, M, N2, Xb);
        Stable_sort(M, L, N - N2, Xb); }
    Buffered_merge(F, M, L, N2, N - N2, Xb); } }

template<class BidIt, class Pd, class T> inline
void Buffered_merge_sort(BidIt F, BidIt L, Pd N,
    Temp_iterator<T>& Xb)
{BidIt M = F;
for (Pd I = N; CHUNK_SIZE <= I; I -= CHUNK_SIZE)
{BidIt Mn = M;
advance(Mn, (int)CHUNK_SIZE);
Insertion_sort(M, Mn);
M = Mn; }
Insertion_sort(M, L);
for (Pd D = CHUNK_SIZE; D < N; D *= 2)
{Chunked_merge(F, L, Xb.Init(), D, N);
Chunked_merge(Xb.First(), Xb.Last(), F,
D *= 2, N); }}

template<class BidIt, class OutIt, class Pd> inline
void Chunked_merge(BidIt F, BidIt L, OutIt X, Pd D, Pd N)
{Pd D2 = D * 2;
for (; D2 <= N; N -= D2)
{BidIt F1 = F;
advance(F1, D);
BidIt F2 = F1;
advance(F2, D);
X = merge(F, F1, F1, F2, X);
F = F2; }
if (N <= D)
copy(F, L, X);
else
{BidIt F1 = F;
advance(F1, D);
merge(F, F1, F1, L, X); } }

// TEMPLATE FUNCTION stable_sort WITH PRED
template<class BidIt, class Pr> inline
void stable_sort(BidIt F, BidIt L, Pr P)
{if (F != L)
    Stable_sort(F, L,
    Dist_type(F), Val_type(F), P); }

template<class BidIt, class Pd, class T, class Pr> inline
void Stable_sort(BidIt F, BidIt L, Pd *, T *, Pr P)
{Pd N = 0;
}

```

```
Distance(F, L, N);
Temp_iterator<T> Xb(N);
Stable_sort(F, L, N, Xb, P); }

template<class BidIt, class Pd, class T, class Pr> inline
void Stable_sort(BidIt F, BidIt L, Pd N,
                 Temp_iterator<T>& Xb, Pr P)
{if (N <= SORT_MAX)
    Insertion_sort(F, L, P);
else
    {Pd N2 = (N + 1) / 2;
     BidIt M = F;
     advance(M, N2);
     if (N2 <= Xb.Maxlen())
         (Buffered_merge_sort(F, M, N2, Xb, P));
     Buffered_merge_sort(M, L, N - N2, Xb, P); }
    else
        {Stable_sort(F, M, N2, Xb, P);
         Stable_sort(M, L, N - N2, Xb, P); }
    Buffered_merge(F, M, L, N2, N - N2, Xb, P); }}

template<class BidIt, class Pd, class T, class Pr> inline
void Buffered_merge_sort(BidIt F, BidIt L, Pd N,
                         Temp_iterator<T>& Xb, Pr P)
{BidIt M = F;
for (Pd I = N; CHUNK_SIZE <= I; I -= CHUNK_SIZE)
    {BidIt Mn = M;
     advance(Mn, (int)CHUNK_SIZE);
     Insertion_sort(M, Mn, P);
     M = Mn; }
Insertion_sort(M, L, P);
for (Pd D = CHUNK_SIZE; D < N; D *= 2)
    {Chunked_merge(F, L, Xb.Init(), D, N, P);
     Chunked_merge(Xb.First(), Xb.Last(), F,
                    D *= 2, N, P); }}

template<class BidIt, class OutIt, class Pd, class Pr> inline
void Chunked_merge(BidIt F, BidIt L, OutIt X,
                   Pd D, Pd N, Pr P)
{Pd D2 = D * 2;
for (; D2 <= N; N -= D2)
    {BidIt F1 = F;
     advance(F1, D);
     BidIt F2 = F1;
     advance(F2, D);
     X = merge(F, F1, F1, F2, X, P);
     F = F2; }
if (N <= D)
```

```

        copy(F, L, X);
    else
        (BidIt F1 = F;
        advance(F1, D);
        merge(F, F1, F1, L, X, P); })

        // TEMPLATE FUNCTION partial_sort
template<class RanIt> inline
void partial_sort(RanIt F, RanIt M, RanIt L)
{Partial_sort(F, M, L, Val_type(F)); }
template<class RanIt, class T> inline
void Partial_sort(RanIt F, RanIt M, RanIt L, T *)
{make_heap(F, M);
for (RanIt I = M; I < L; ++I)
if (*I < *F)
    Pop_heap(F, M, I, T(*I), Dist_type(F));
sort_heap(F, M); }

        // TEMPLATE FUNCTION partial_sort WITH PRED
template<class RanIt, class Pr> inline
void partial_sort(RanIt F, RanIt M, RanIt L, Pr P)
{Partial_sort(F, M, L, P, Val_type(F)); }
template<class RanIt, class T, class Pr> inline
void Partial_sort(RanIt F, RanIt M, RanIt L, Pr P, T *)
{make_heap(F, M, P);
for (RanIt I = M; I < L; ++I)
if (P(*I, *F))
    Pop_heap(F, M, I, T(*I), P, Dist_type(F));
sort_heap(F, M, P); }

        // TEMPLATE FUNCTION partial_sort_copy
template<class InIt, class RanIt> inline
RanIt partial_sort_copy(InIt F1, InIt L1, RanIt F2, RanIt L2)
{return (Partial_sort_copy(F1, L1, F2, L2,
Dist_type(F2), Val_type(F1))); }
template<class InIt, class RanIt, class Pd, class T> inline
RanIt Partial_sort_copy(InIt F1, InIt L1, RanIt F2,
RanIt L2, Pd *, T *)
{RanIt X = F2;
if (X != L2)
{for (; F1 != L1 && X != L2; ++F1, ++X)
*X = *F1;
make_heap(F2, X);
for (; F1 != L1; ++F1)
if (*F1 < *F2)
}
}

```

```

        Adjust_heap(F2, Pd(0), Pd(X - F2),
                    T(*F1));
        sort_heap(F2, X); }
    return (X); }

        // TEMPLATE FUNCTION partial_sort_copy WITH PRED
template<class InIt, class RanIt, class Pr> inline
    RanIt partial_sort_copy(InIt F1, InIt L1, RanIt F2,
                           RanIt L2, Pr P)
{return (Partial_sort_copy(F1, L1, F2, L2, P,
                           Dist_type(F2), Val_type(F1))); }

template<class InIt, class RanIt, class Pd,
         class T, class Pr> inline
    RanIt Partial_sort_copy(InIt F1, InIt L1, RanIt F2,
                           RanIt L2, Pr P, Pd *, T *)
{RanIt X = F2;
if (X != L2)
    {for (; F1 != L1 && X != L2; ++F1, ++X)
     *X = *F1;
     make_heap(F2, X, P);
     for (; F1 != L1; ++F1)
         if (P(*F1, *F2))
             Adjust_heap(F2, Pd(0), Pd(X - F2),
                         T(*F1), P);
     sort_heap(F2, X, P); }
return (X); }

        // TEMPLATE FUNCTION nth_element
template<class RanIt> inline
void nth_element(RanIt F, RanIt Nth, RanIt L)
{for (; SORT_MAX < L - F; )

    {RanIt M = Unguarded_partition(F, L,
                                    Val_type(F));
     if (M <= Nth)
         F = M;
     else
         L = M; }
    Insertion_sort(F, L); }

        // TEMPLATE FUNCTION nth_element WITH PRED
template<class RanIt, class Pr> inline
void nth_element(RanIt F, RanIt Nth, RanIt L, Pr P)
{for (; SORT_MAX < L - F; )

```

```

    {RanIt M = Unguarded_partition(F, L, P,
        Val_type(F));
     if (M <= Nth)
         F = M;
     else
         L = M; }
    Insertion_sort(F, L, P); }

        // TEMPLATE FUNCTION lower_bound
template<class FwdIt, class T> inline
    FwdIt lower_bound(FwdIt F, FwdIt L, const T& V)
    {return (Lower_bound(F, L, V, Dist_type(F))); }
template<class FwdIt, class T, class Pd> inline
    FwdIt Lower_bound(FwdIt F, FwdIt L, const T& V, Pd *)
    {Pd N = 0;
     Distance(F, L, N);
     for (; 0 < N; )
         {Pd N2 = N / 2;
          FwdIt M = F;
          advance(M, N2);
          if (*M < V)
              F = ++M, N -= N2 + 1;
          else
              N = N2; }
     return (F); }

        // TEMPLATE FUNCTION lower_bound WITH PRED
template<class FwdIt, class T, class Pr> inline
    FwdIt lower_bound(FwdIt F, FwdIt L, const T& V, Pr P)
    {return (Lower_bound(F, L, V, P, Dist_type(F))); }
template<class FwdIt, class T, class Pd, class Pr> inline
    FwdIt Lower_bound(FwdIt F, FwdIt L, const T& V, Pr P, Pd *)
    {Pd N = 0;
     Distance(F, L, N);
     for (; 0 < N; )
         {Pd N2 = N / 2;
          FwdIt M = F;
          advance(M, N2);
          if (P(*M, V))
              F = ++M, N -= N2 + 1;
          else
              N = N2; }
     return (F); }

```

```
// TEMPLATE FUNCTION upper_bound
template<class FwdIt, class T> inline
    FwdIt upper_bound(FwdIt F, FwdIt L, const T& V)
        {return (Upper_bound(F, L, V, Dist_type(F))); }

template<class FwdIt, class T, class Pd> inline
    FwdIt Upper_bound(FwdIt F, FwdIt L, const T& V, Pd *)
        {Pd N = 0;
        Distance(F, L, N);
        for (; 0 < N; )
            {Pd N2 = N / 2;
            FwdIt M = F;
            advance(M, N2);
            if (!(V < *M))
                F = ++M, N -= N2 + 1;
            else
                N = N2; }
        return (F); }

// TEMPLATE FUNCTION upper_bound WITH PRED
template<class FwdIt, class T, class Pr> inline
    FwdIt upper_bound(FwdIt F, FwdIt L, const T& V, Pr P)
        {return (Upper_bound(F, L, V, P, Dist_type(F))); }

template<class FwdIt, class T, class Pd, class Pr> inline
    FwdIt Upper_bound(FwdIt F, FwdIt L, const T& V, Pr P, Pd *)
        {Pd N = 0;
        Distance(F, L, N);
        for (; 0 < N; )
            {Pd N2 = N / 2;
            FwdIt M = F;
            advance(M, N2);
            if (!P(V, *M))
                F = ++M, N -= N2 + 1;
            else
                N = N2; }
        return (F); }

// TEMPLATE FUNCTION equal_range
template<class FwdIt, class T> inline
    pair<FwdIt, FwdIt> equal_range(FwdIt F, FwdIt L, const T& V)
        {return (Equal_range(F, L, V, Dist_type(F))); }

template<class FwdIt, class T, class Pd> inline
    pair<FwdIt, FwdIt> Equal_range(FwdIt F, FwdIt L,
        const T& V, Pd *)
        {Pd N = 0;
        Distance(F, L, N);
```

```

        for ( ; 0 < N; )
            {Pd N2 = N / 2;
             FwdIt M = F;
             advance(M, N2);
             if (*M < V)
                 F = ++M, N -= N2 + 1;
             else if (V < *M)
                 N = N2;
             else
                 {FwdIt F2 = lower_bound(F, M, V);
                  advance(F, N);
                  FwdIt L2 = upper_bound(++M, F, V);
                  return (pair<FwdIt, FwdIt>(F2, L2)); }
             return (pair<FwdIt, FwdIt>(F, F)); }

        // TEMPLATE FUNCTION equal_range WITH PRED
template<class FwdIt, class T, class Pr> inline
    pair<FwdIt, FwdIt> equal_range(FwdIt F, FwdIt L,
        const T& V, Pr P)
    {return (Equal_range(F, L, V, P, Dist_type(F))); }
template<class FwdIt, class T, class Pd, class Pr> inline
    pair<FwdIt, FwdIt> Equal_range(FwdIt F, FwdIt L,
        const T& V, Pr P, Pd *)
    {Pd N = 0;
     Distance(F, L, N);
     for ( ; 0 < N; )
         {Pd N2 = N / 2;
          FwdIt M = F;
          advance(M, N2);
          if (P(*M, V))
              F = ++M, N -= N2 + 1;
          else if (P(V, *M))
              N = N2;
          else
              {FwdIt F2 = lower_bound(F, M, V, P);
               advance(F, N);
               FwdIt L2 = upper_bound(++M, F, V, P);
               return (pair<FwdIt, FwdIt>(F2, L2)); }
          return (pair<FwdIt, FwdIt>(F, F)); }

        // TEMPLATE FUNCTION binary_search
template<class FwdIt, class T> inline
    bool binary_search(FwdIt F, FwdIt L, const T& V)
    {FwdIt I = lower_bound(F, L, V);
     return (I != L && !(V < *I)); }

```

```
// TEMPLATE FUNCTION binary_search WITH PRED
template<class FwdIt, class T, class Pr> inline
    bool binary_search(FwdIt F, FwdIt L, const T& V, Pr P)
    {FwdIt I = lower_bound(F, L, V, P);
     return (I != L && !P(V, *I)); }

// TEMPLATE FUNCTION merge
template<class InIt1, class InIt2, class OutIt> inline
    OutIt merge(InIt1 F1, InIt1 L1, InIt2 F2, InIt2 L2, OutIt X)
    {for (; F1 != L1 && F2 != L2; ++X)
        if (*F2 < *F1)
            *X = *F2, ++F2;
        else
            *X = *F1, ++F1;
    X = copy(F1, L1, X);
    return (copy(F2, L2, X)); }

// TEMPLATE FUNCTION merge WITH PRED
template<class InIt1, class InIt2, class OutIt, class Pr>
    inline OutIt merge(InIt1 F1, InIt1 L1, InIt2 F2,
                       InIt2 L2, OutIt X, Pr P)
    {for (; F1 != L1 && F2 != L2; ++X)
        if (P(*F2, *F1))
            *X = *F2, ++F2;
        else
            *X = *F1, ++F1;
    X = copy(F1, L1, X);
    return (copy(F2, L2, X)); }

// TEMPLATE FUNCTION inplace_merge
template<class BidIt> inline
    void inplace_merge(BidIt F, BidIt M, BidIt L)
    {if (F != L)
        Inplace_merge(F, M, L,
                      Dist_type(F), Val_type(F)); }
template<class BidIt, class Pd, class T> inline
    void Inplace_merge(BidIt F, BidIt M, BidIt L, Pd *, T *)
    {Pd D1 = 0;
     Distance(F, M, D1);
     Pd D2 = 0;
     Distance(M, L, D2);
     Temp_iterator<T> Xb(D1 < D2 ? D1 : D2);
     Buffered_merge(F, M, L, D1, D2, Xb); }
```

```

template<class BidIt, class Pd, class T> inline
    void Buffered_merge(BidIt F, BidIt M, BidIt L,
        Pd D1, Pd D2, Temp_iterator<T>& Xb)
    {if (D1 == 0 || D2 == 0)
        ;
    else if (D1 + D2 == 2)
        {if (*M < *F)
            iter_swap(F, M); }
    else if (D1 <= D2 && D1 <= Xb.Maxlen())
        {copy(F, M, Xb.Init());
        merge(Xb.First(), Xb.Last(), M, L, F); }
    else if (D2 <= Xb.Maxlen())
        {copy(M, L, Xb.Init());
        Merge_backward(F, M, Xb.First(), Xb.Last(), L); }
    else
        {BidIt Fn, Ln;
        Pd D1n, D2n;
        if (D2 < D1)
            {D1n = D1 / 2, D2n = 0;
            Fn = F;
            advance(Fn, D1n);
            Ln = lower_bound(M, L, *Fn);
            Distance(M, Ln, D2n); }
        else
            {D1n = 0, D2n = D2 / 2;
            Ln = M;
            advance(Ln, D2n);
            Fn = upper_bound(F, M, *Ln);
            Distance(F, Fn, D1n); }
        BidIt Mn = Buffered_rotate(Fn, M, Ln,
            D1 - D1n, D2n, Xb);
        Buffered_merge(F, Fn, Mn, D1n, D2n, Xb);
        Buffered_merge(Mn, Ln, L,
            v1 - D1n, D2 - D2n, Xb); }}
template<class BidIt1, class BidIt2, class BidIt3> inline
    BidIt3 Merge_backward(BidIt1 F1, BidIt1 L1,
        BidIt2 F2, BidIt2 L2, BidIt3 X)
    {for (; ; )
        if (F1 == L1)
            return (copy_backward(F2, L2, X));
        else if (F2 == L2)
            return (copy_backward(F1, L1, X)); }

```

```
        else if (*--L2 < *--L1)
            *--X = *L1, ++L2;
        else
            *--X = *L2, ++L1; }

template<class BidIt, class Pd, class T> inline
    BidIt Buffered_rotate(BidIt F, BidIt M, BidIt L,
                          Pd D1, Pd D2, Temp_iterator<T>& Xb)
{if (D1 <= D2 && D1 <= Xb.Maxlen())
    {copy(F, M, Xb.Init());
     copy(M, L, F);
     return (copy_backward(Xb.First(), Xb.Last(), L)); }
else if (D2 <= Xb.Maxlen())
    {copy(M, L, Xb.Init());
     copy_backward(F, M, L);
     return (copy(Xb.First(), Xb.Last(), F)); }
else
    {rotate(F, M, L);
     advance(F, D2);
     return (F); } }

// TEMPLATE FUNCTION inplace_merge WITH PRED
template<class BidIt, class Pr> inline
void inplace_merge(BidIt F, BidIt M, BidIt L, Pr P)
{if (F != L)
    Inplace_merge(F, M, L, P,
                  Dist_type(F), Val_type(F)); }

template<class BidIt, class Pd, class T, class Pr> inline
void Inplace_merge(BidIt F, BidIt M, BidIt L, Pr P,
                  Pd *, T *)
{Pd D1 = 0;
 Distance(F, M, D1);
 Pd D2 = 0;
 Distance(M, L, D2);
 Temp_iterator<T> Xb(D1 < D2 ? D1 : D2);
 Buffered_merge(F, M, L, D1, D2, Xb, P); }

template<class BidIt, class Pd, class T, class Pr> inline
void Buffered_merge(BidIt F, BidIt M, BidIt L,
```

```

Pd D1, Pd D2, Temp_iterator<T>& Xb, Pr P)
(if (D1 == 0 || D2 == 0)
;
else if (D1 + D2 == 2)
(if (*M, *F))
iter_swap(F, M); }
else if (D1 <= D2 && D1 <= Xb.Maxlen())
{copy(F, M, Xb.Init());
merge(Xb.First(), Xb.Last(), M, L, F, P); }
else if (D2 <= Xb.Maxlen())
{copy(M, L, Xb.Init());
Merge_backward(F, M, Xb.First(), Xb.Last(),
L, P); }
else
(BidIt Fn, Ln;
Pd D1n, D2n;
if (D2 < D1)
{D1n = D1 / 2, D2n = 0;
Fn = F;
advance(Fn, D1n);
Ln = lower_bound(M, L, *Fn, P);
Distance(M, Ln, D2n); }
else
(D1n = 0, D2n = D2 / 2;
Ln = M;
advance(Ln, D2n);
Fn = upper_bound(F, M, *Ln, P);
Distance(F, Fn, D1n); }
BidIt Mn = Buffered_rotate(Fn, M, Ln,
D1 - D1n, D2n, Xb);
Buffered_merge(F, Fn, Mn, D1n, D2n, Xb, P);
Buffered_merge(Mn, Ln, L,
D1 - D1n, D2 - D2n, Xb, P); })
template<class BidIt1, class BidIt2, class BidIt3, class Pr>
inline BidIt3 Merge_backward(BidIt1 F1, BidIt1 L1,
BidIt2 F2, BidIt2 L2, BidIt3 X, Pr P)
{for (; ; )
if (F1 == L1)
return (copy_backward(F2, L2, X));
else if (F2 == L2)
return (copy_backward(F1, L1, X));
else if (P(--L2, --L1))
--X = *L1, ++L2;
else
;
}

```

```
    *--X = *L2, ++L1; }

    // TEMPLATE FUNCTION includes
template<class InIt1, class InIt2> inline
    bool includes(InIt1 F1, InIt1 L1, InIt2 F2, InIt2 L2)
{for (; F1 != L1 && F2 != L2; )
    if (*F2 < *F1)
        return (false);
else if (*F1 < *F2)
    ++F1;
else
    ++F1, ++F2;
return (F2 == L2); }

    // TEMPLATE FUNCTION includes WITH PRED
template<class InIt1, class InIt2, class Pr> inline
    bool includes(InIt1 F1, InIt1 L1, InIt2 F2, InIt2
L2, Pr P)
{for (; F1 != L1 && F2 != L2; )
    if (P(*F2, *F1))
        return (false);
else if (P(*F1, *F2))
    ++F1;
else
    ++F1, ++F2;
return (F2 == L2); }

    // TEMPLATE FUNCTION set_union
template<class InIt1, class InIt2, class OutIt> inline
    OutIt set_union(InIt1 F1, InIt1 L1,
                    InIt2 F2, InIt2 L2, OutIt X)
{for (; F1 != L1 && F2 != L2; )
    if (*F1 < *F2)
        *X++ = *F1, ++F1;
    else if (*F2 < *F1)
        *X++ = *F2, ++F2;
    else
        *X++ = *F1, ++F1, ++F2;
X = copy(F1, L1, X);
return (copy(F2, L2, X)); }

    // TEMPLATE FUNCTION set_union WITH PRED
template<class InIt1, class InIt2, class OutIt, class Pr>
    inline OutIt set_union(InIt1 F1, InIt1 L1,
```

```

        InIt2 F2, InIt2 L2, OutIt X, Pr P)
(for (; F1 != L1 && F2 != L2; )
    if (P(*F1, *F2))
        *X++ = *F1, ++F1;
    else if (P(*F2, *F1))
        *X++ = *F2, ++F2;
    else
        *X++ = *F1, ++F1, ++F2;
    X = copy(F1, L1, X);
    return (copy(F2, L2, X)); }

        // TEMPLATE FUNCTION set_intersection
template<class InIt1, class InIt2, class OutIt> inline
OutIt set_intersection(InIt1 F1, InIt1 L1,
                      InIt2 F2, InIt2 L2, OutIt X)
(for (; F1 != L1 && F2 != L2; )
    if (*F1 < *F2)
        ++F1;
    else if (*F2 < *F1)
        ++F2;
    else
        *X++ = *F1++, ++F2;
return (X); }

        // TEMPLATE FUNCTION set_intersection WITH PRED
template<class InIt1, class InIt2, class OutIt, class Pr>
inline OutIt set_intersection(InIt1 F1, InIt1 L1,
                           InIt2 F2, InIt2 L2, OutIt X, Pr P)
(for (; F1 != L1 && F2 != L2; )
    if (P(*F1, *F2))
        ++F1;
    else if (P(*F2, *F1))
        ++F2;
    else
        *X++ = *F1++, ++F2;
return (X); }

        // TEMPLATE FUNCTION set_difference
template<class InIt1, class InIt2, class OutIt> inline
OutIt set_difference(InIt1 F1, InIt1 L1,
                     InIt2 F2, InIt2 L2, OutIt X)
(for (; F1 != L1 && F2 != L2; )
    if (*F1 < *F2)
        *X++ = *F1, ++F1;
    else
        ++F2;
return (X); }

```

```
        else if (*F2 < *F1)
            ++F2;
        else
            ++F1, ++F2;
    return (copy(F1, L1, X)); }

// TEMPLATE FUNCTION set_difference WITH PRED
template<class InIt1, class InIt2, class OutIt, class Pr>
inline OutIt set_difference(InIt1 F1, InIt1 L1,
                           InIt2 F2, InIt2 L2, OutIt X, Pr P)
{for (; F1 != L1 && F2 != L2; )
    if (P(*F1, *F2))
        *X++ = *F1, ++F1;
    else if (P(*F2, *F1))
        ++F2;
    else
        ++F1, ++F2;
return (copy(F1, L1, X)); }

// TEMPLATE FUNCTION set_symmetric_difference
template<class InIt1, class InIt2, class OutIt> inline
OutIt set_symmetric_difference(InIt1 F1, InIt1 L1,
                               InIt2 F2, InIt2 L2, OutIt X)
{for (; F1 != L1 && F2 != L2; )
    if (*F1 < *F2)
        *X++ = *F1, ++F1;
    else if (*F2 < *F1)
        *X++ = *F2, ++F2;
    else
        ++F1, ++F2;
X = copy(F1, L1, X);
return (copy(F2, L2, X)); }

// TEMPLATE FUNCTION set_symmetric_difference WITH PRED
template<class InIt1, class InIt2, class OutIt, class Pr>
inline OutIt set_symmetric_difference(InIt1 F1,
                                      InIt1 L1, InIt2 F2, InIt2 L2, OutIt X, Pr P)
{for (; F1 != L1 && F2 != L2; )
    if (P(*F1, *F2))
        *X++ = *F1, ++F1;
    else if (P(*F2, *F1))
        *X++ = *F2, ++F2;
    else
        ++F1, ++F2;
```

```

        X = copy(F1, L1, X);
        return (copy(F2, L2, X)); }

        // TEMPLATE FUNCTION push_heap
template<class RanIt> inline
    void push_heap(RanIt F, RanIt L)
    {Push_heap_0(F, L, Dist_type(F), Val_type(F)); }
template<class RanIt, class Pd, class T> inline
    void Push_heap_0(RanIt F, RanIt L, Pd *, T *)
    {Push_heap(F, Pd(L - F - 1), Pd(0), T(*(L - 1))); }
template<class RanIt, class Pd, class T> inline
    void Push_heap(RanIt F, Pd H, Pd J, T V)
    {for (Pd I = (H - 1) / 2; J < H && *(F + I) < V;
        I = (H - 1) / 2)
        *(F + H) = *(F + I), H = I;
        *(F + H) = V; }

        // TEMPLATE FUNCTION push_heap WITH PRED
template<class RanIt, class Pr> inline
    void push_heap(RanIt F, RanIt L, Pr P)
    {Push_heap_0(F, L, P,
        Dist_type(F), Val_type(F)); }
template<class RanIt, class Pd, class T, class Pr> inline
    void Push_heap_0(RanIt F, RanIt L, Pr P, Pd *, T *)
    {Push_heap(F, Pd(L - F - 1), Pd(0),
        T(*(L - 1)), P); }
template<class RanIt, class Pd, class T, class Pr>
    inline void Push_heap(RanIt F, Pd H, Pd J, T V, Pr P)
    {for (Pd I = (H - 1) / 2; J < H && P(*(F + I), V);
        I = (H - 1) / 2)
        *(F + H) = *(F + I), H = I;
        *(F + H) = V; }

        // TEMPLATE FUNCTION pop_heap
template<class RanIt> inline
    void pop_heap(RanIt F, RanIt L)
    {Pop_heap_0(F, L, Val_type(F)); }
template<class RanIt, class T> inline
    void Pop_heap_0(RanIt F, RanIt L, T *)
    {Pop_heap(F, L - 1, L - 1, T(*(L - 1)),
        Dist_type(F)); }
template<class RanIt, class Pd, class T> inline
    void Pop_heap(RanIt F, RanIt L, RanIt X, T V, Pd *)

```

```

(*X = *F;
Adjust_heap(F, Pd(0), Pd(L - F), V); )
template<class RanIt, class Pd, class T> inline
void Adjust_heap(RanIt F, Pd H, Pd N, T V)
{Pd J = H;
Pd K = 2 * H + 2;
for (; K < N; K = 2 * K + 2)
{if (*F + K) < *(F + (K - 1)))
--K;
*(F + H) = *(F + K), H = K; }
if (K == N)
*(F + H) = *(F + (K - 1)), H = K - 1;
Push_heap(F, H, J, V); }

// TEMPLATE FUNCTION pop_heap WITH PRED
template<class RanIt, class Pr> inline
void pop_heap(RanIt F, RanIt L, Pr P)
{Pop_heap_0(F, L, P, Val_type(F)); }
template<class RanIt, class T, class Pr> inline
void Pop_heap_0(RanIt F, RanIt L, Pr P, T *)
{Pop_heap(F, L - 1, L - 1, T(*L - 1)), P,
Dist_type(F)); }
template<class RanIt, class Pd, class T, class Pr> inline
void Pop_heap(RanIt F, RanIt L, RanIt X, T V, Pr P,
Pd *)
{*X = *F;
Adjust_heap(F, Pd(0), Pd(L - F), V, P); }
template<class RanIt, class Pd, class T, class Pr> inline
void Adjust_heap(RanIt F, Pd H, Pd N, T V, Pr P)
{Pd J = H;
Pd K = 2 * H + 2;
for (; K < N; K = 2 * K + 2)
{if (P(*F + K), *(F + (K - 1))))
--K;
*(F + H) = *(F + K), H = K; }
if (K == N)
*(F + H) = *(F + (K - 1)), H = K - 1;
Push_heap(F, H, J, V, P); }

// TEMPLATE FUNCTION make_heap
template<class RanIt> inline
void make_heap(RanIt F, RanIt L)
{if (2 <= L - F)
Make_heap(F, L, Dist_type(F), Val_type(F)); }
template<class RanIt, class Pd, class T> inline

```

```

void Make_heap(RanIt F, RanIt L, Pd *, T *)
{Pd N = L - F;
for (Pd H = N / 2; 0 < H; )
    --H, Adjust_heap(F, H, N, T(*(F + H))); }

// TEMPLATE FUNCTION make_heap WITH PRED
template<class RanIt, class Pr> inline
void make_heap(RanIt F, RanIt L, Pr P)
{if (2 <= L - F)
    Make_heap(F, L, P,
              Dist_type(F), Val_type(F)); }

template<class RanIt, class Pd, class T, class Pr>
inline void Make_heap(RanIt F, RanIt L, Pr P, Pd *, T *)
{Pd N = L - F;
for (Pd H = N / 2; 0 < H; )
    --H, Adjust_heap(F, H, N, T(*(F + H)), P); }

// TEMPLATE FUNCTION sort_heap
template<class RanIt> inline
void sort_heap(RanIt F, RanIt L)
{for (; 1 < L - F; --L)
    pop_heap(F, L); }

// TEMPLATE FUNCTION sort_heap WITH PRED
template<class RanIt, class Pr> inline
void sort_heap(RanIt F, RanIt L, Pr P)
{for (; 1 < L - F; --L)
    pop_heap(F, L, P); }

// TEMPLATE FUNCTION max_element
template<class FwdIt> inline
FwdIt max_element(FwdIt F, FwdIt L)
{FwdIt X = F;
if (F != L)
    for (; ++F != L; )
        if (*X < *F)
            X = F;
return (X); }

// TEMPLATE FUNCTION max_element WITH PRED
template<class FwdIt, class Pr> inline
FwdIt max_element(FwdIt F, FwdIt L, Pr P)
{FwdIt X = F;

```

```
    if (F != L)
        for (; ++F != L; )
            if (*P(*X, *F))
                X = F;
        return (X); }

        // TEMPLATE FUNCTION min_element
template<class FwdIt> inline
FwdIt min_element(FwdIt F, FwdIt L)
{FwdIt X = F;
if (F != L)
    for (; ++F != L; )
        if (*F < *X)
            X = F;
return (X); }

        // TEMPLATE FUNCTION min_element WITH PRED
template<class FwdIt, class Pr> inline
FwdIt min_element(FwdIt F, FwdIt L, Pr P)
{FwdIt X = F;
if (F != L)
    for (; ++F != L; )
        if (P(*F, *X))
            X = F;
return (X); }

        // TEMPLATE FUNCTION next_permutation
template<class BidIt> inline
bool next_permutation(BidIt F, BidIt L)
{BidIt I = L;
if (F == L || F == --I)
    return (false);
for (; ; )
    {BidIt Ip = I;
    if (*--I < *Ip)
        {BidIt J = L;
        for (; !( *I < *--J); )
            ;
        iter_swap(I, J);
        reverse(Ip, L);
        return (true); }
    if (I == F)
        {reverse(F, L);
        return (false); }}}
```

```
// TEMPLATE FUNCTION next_permutation WITH PRED
template<class BidIt, class Pr> inline
    bool next_permutation(BidIt F, BidIt L, Pr P)
    {BidIt I = L;
     if (F == L || F == --I)
         return (false);
     for (; ; )
         {BidIt Ip = I;
          if (!P(*--I, *Ip))
              {BidIt J = L;
               for (; !P(*I, *--J); )
                   ;
                   iter_swap(I, J);
                   reverse(Ip, L);
                   return (true); }
          if (I == F)
              {reverse(F, L);
               return (false); }}}

// TEMPLATE FUNCTION prev_permutation
template<class BidIt> inline
    bool prev_permutation(BidIt F, BidIt L)
    {BidIt I = L;
     if (F == L || F == --I)
         return (false);
     for (; ; )
         {BidIt Ip = I;
          if (!(*--I < *Ip))
              {BidIt J = L;
               for (; *I < *--J; )
                   ;
                   iter_swap(I, J);
                   reverse(Ip, L);
                   return (true); }
          if (I == F)
              {reverse(F, L);
               return (false); }}}

// TEMPLATE FUNCTION prev_permutation WITH PRED
template<class BidIt, class Pr> inline
    bool prev_permutation(BidIt F, BidIt L, Pr P)
    {BidIt I = L;
     if (F == L || F == --I)
         return (false);
     for (; ; )
```

```

{BidIt Ip = I;
if (!P(--I, *Ip))
{BidIt J = L;
for (; P(*I, *--J); )
;
iter_swap(I, J);
reverse(Ip, L);
return (true);
if (I == F)
(reverse(F, L);
return (false); )})
} /* namespace std */
#endif /* ALGORITHM_ */

```

CHUNK_SIZE
SORT_MAX

该头文件一开始就给出了两个常数。模板函数 `Bufered_merge_sort` 会将一个序列分成几块来排序，每块能够容纳 `CHUNK_SIZE` 个元素，然后再将这些块合并起来。许多模板函数可以用来排序，它们能够在元素个数不大于 `SORT_MAX` 的任意序列中执行插入排序 (`insertion sort`)。也就是说，在选择不同的辅助算法时，这两个常数可以描述近似的交叉点。在以后的使用中我们也不需要去改动它们——它们在此处出现的意义更多是为了可读性，而不是为了轻易地被改动。

count
count_if

模板函数 `count` 和 `count_if` 的返回值只有在支持部分特化的实现中才能表达清楚。我们甚至不能通过调用另一个模板函数来检测有疑问的类型。某些实现可能会替换掉那些不能命名为 `ptrdiff_t` 的类型。除了对于那些奇异的迭代器外，`difference_type` 都是 `ptrdiff_t` 的同义词。

search

但是，如果有一个迭代器正好拥有一个奇异的差距类型（而且实现又不能使用 `iterator_traits` 的那些特性），我们又该怎么办呢？模板函数 `search` 向我们展示了如何处理这种情况。它调用了另一个新增了两个参数的模板函数 `Search`。每个新增的参数使用模板函数 `Dist_type`，将 `difference_type` 表达为一个模板函数类型。也就是说，`Search` 可以为 `search` 中出现的迭代器对中的一个声明一个适当的计数器。

我们在前面已经讲述过了如何使用辅助的模板函数及 `Iter_cat` 来检测迭代器种类，如第 3 章中出现过的模板函数 `distance`。我们也见过如何使用辅助的模板函数及 `Val_type` 来检测迭代器的值类型，如在第 4 章中出现过的模板函数 `uninitialized_copy`。在本章，我们将见到如何使用 `Dist_type` 来为 `search` 检测迭代器的差距类型。我们不会在此再次详细讨论这个机制。

unique_copy

然而，还是有一个模板函数值得我们简短地说一下，这就是 `unique_copy`。它使用 `Iter_cat` 来为不同种类的迭代器选择不同的算法。在这种情况下，不同算法之间的差别都非常小，但我们却不能因此而忽

略掉它们。输入迭代器不支持在序列中标注一个“书签”。因而在对元素进行比较时，就需要先前的元素存储在一个临时的对象中而不管该对象实际上会有多大。前向（以及其他功能更强的）迭代器则只需要存储一个临时的迭代器对象，而这个对象的大小通常都是比较合理的。

如我们先前所观察的一样，用于双向迭代器以及随机存取迭代器的重载实际上并不是必需的，但我们还是会碰到需要它们的 C++ 编译器。也就是说，它们是为了健壮性才被包含的，这主要也是为了适合当前实现的需求。

`reverse`

模板函数 `reverse` 给我们演示了一种相似的、小小的优化。随机存取迭代器支持用 `operator<` 来进行比较，而这比双向迭代器所需的两步测试要稍微高效一点。

`rotate`

模板函数 `rotate` 可能是 STL 的所有算法中最具有自适应性的算法了：

- 对前向迭代器来说，它执行 n 次交换以旋转 n 个元素。
- 对双向迭代器来说，它先是反转两个子序列，然后是整个序列，以此来实现真正的旋转。
- 对随机存取迭代器来说，它只是尽可能少地旋转序列中的一些子循环。

后两个版本是值得我们好好学习的，尤其是当我们对这种特殊的算法不是那么熟悉的时候。

`random_shuffle`

模板函数 `random_shuffle` 需要一个很好的随机数源，越自然越好。这些数字必须（或多或少地）均匀分布在区间 $[0, n]$ 中（此处 n 是序列的长度）。不幸的是， n 的类型依赖于模板常数——原则上来说它可以是一个有着任意精度的类型。

本实现并没有尝试去解决这样的“开放型”问题。它只是根据需要，通过不断地调用标准 C 中的库函数 `rand`（声明于 `<stdlib.h>` 中），将随机的位聚集在一个 `unsigned long` 对象中。由于 `rand` 不产生超过 15 位的随机数，所以我们调用它的次数可能不止一次。如果需要对一个 10GB 的磁盘上的所有字节进行重排，那么该算法就会失效。此外，`rand` 的一般实现都清楚那些能够影响对于随机数有着苛刻要求的应用程序的缺陷。

（在惠普的 STL 版本中所提供的随机数产生器在此方面做的尤其值得炫耀，不过它还是受限于所产生的随机数的范围。）然而，对于典型的应用程序来说，`random_shuffle` 已经足够随机了。

`stable_partition`

模板函数 `stable_partition` 在可能的情况下有效地利用临时缓冲区。模板函数 `Stable_partition` 的第一个重载版本要求能有一个大到可以容纳整个序列的缓冲区。（我们在第 4 章的“背景知识”一节中已经描述过了模板类 `Temp_iterator`。）不管得到的缓冲区有多大，该模板函数的第

二个操作版本都可以获得好处。如果序列比缓冲区要大，第二个函数就将退回到经典的“分而治之”的策略中，以此来减少由于可管理尺寸所产生的问题。如果实在没有得到任何临时缓冲区，我们将重复地对区间进行对分直到最后得到的间隔差距等于 1 为止。

模板函数 `Buffered_rotate`（我们将在下面连同 `inplace_merge` 一同介绍）同样也利用了临时缓冲区。它在此处的工作是重排那些通过递归调用 `Stable_partition` 所得到的相邻的、但又是杂乱无序的子区间。

`stable_partition` 是少数几个在头文件<algorithm>中定义的、在可能的情况下能够利用临时缓冲区获益的模板函数中的一个。注意，在有着足够的临时缓冲区时，这些函数可以执行得更快，但即使没有任何缓冲区，它们也可以通过其他方法完成工作。

sort

模板函数 `sort` 利用大量的附加函数尽可能地加速其执行速度。对于长度不超过 `SORT_MAX` 个元素的任意序列，它依赖于模板函数 `Insertion_sort` 来完成实际的排序。它依次调用 `Insertion_sort_I` 来完成每次的插入及排序。（在较小的序列上）这样做弥补了操作技巧方面的简单性。模板函数 `Unguarded_insert` 从它的名字上就可以看出它对于参数的信任——它假设在运行到超过（子）序列的开始处时，函数会被终止。

模板函数 `Sort` 仅仅完成了一个很简陋的排序。通过分而治之，它确保将一个很长的序列划分为不同的块（每个块最多有 `SORT_MAX` 个元素）被排序。这些块之间元素的顺序可以保证，但是每个块中元素的顺序就不能保证了。通过调用模板函数 `Sort_end` 进行最后的排序并终止运行。`Sort_end` 可以用很简单但很快速的方法实现，因为它所产生的每次对于 `Unguarded_insert` 的调用都只是对元素进行一次局部的重排而已。

模板函数 `Sort` 还利用了 Dave Musser 所发现的一种有趣的策略，我们把它叫做 `intro sort`。它的行为和传统的快速排序（quick sort）十分相似，但却多了一些新增的簿记工作。每次对 `Sort` 的递归调用都包括一个对该子区间中有多少元素需要被排序的评估，我们把它叫做 `Ideal`。在理想的情况下，每次分割都将会把一个区间划分为两个相等长度的子区间。在这种理想的情况下，对一个长度为 N 的序列进行排序时，`Sort` 只需被递归调用 $\log_2(N)$ 次。快速排序在序列本身的划分接近理想情况时十分快，然而在处理那些异常划分的序列时，它又十分慢。这就需要我们关注快速排序是如何进行排序的。

除非要排序的序列的元素个数不超过 `SORT_MAX`（此时不值得使用快速排序），或是递归调用的次数超出了“理想的”限制，`Sort` 总是使用快速排序方法。严格地说，由于 `SORT_MAX` 为 16，所以允许的额外递归调用次数最多为 4 次。但当情况变得糟糕时（例如在 `Sort` 中使用快速排序的结果不理想时），这种策略就会发生改变。在对子区间进行

排序时，它会调用 `make_heap` 和 `sort_heap`（它们也在`<algorithm>`中定义）。这两个函数组合起来就可以完成一次堆排序（`heap sort`），虽然在一般情况下这种排序方式不如快速排序高效，但在最坏的情况下，它的时间复杂度却比快速排序好很多。实际上，这种组合式的策略可以保证即使是在最坏的情况下，时间复杂度也可以达到 $N * \log(N)$ 。这样做所需的额外开销只是新增的部分代码，但却可以换回良好的性能。

为了实现传统的快速排序，`Sort` 调用了 `Unguarded_partition`。该模板函数一开始会选出一个作为枢轴（pivot）的值，并确保它存在于所有子序列中元素的值所形成的区间内。它的做法是：对现有子序列中第一个、中间那个以及最后一个元素进行排序，并取它们中处于中间大小的值来作为枢轴值。对这三个值进行重排可以去除掉我们对于始端及末端元素的偏爱。这可以使潜在的异常问题的数量最小化，如：一个以逆序存在的序列，或是向一个已排序好的序列末端添加一个最小的元素。一旦我们确定了这个枢轴，本该位于该枢轴右边但却处于其左边的元素将会和处于该枢轴右边有着类似问题的元素进行交换。该过程将一直重复下去，直到子区间被适当地划分为止。

`stable_sort`

模板函数 `stable_sort` 与 `sort` 很类似，但它又多了一些更为复杂的任务。模板函数 `Stable_sort` 的第二个重载版本按照需要执行一种“分而治之”的策略，直到所产生的子序列短得足够进行其他方式的排序为止。如果该算法能够获得到足够大的临时缓冲区，它会在调用 `Buffered_merge_sort` 时使用该缓冲区。否则，它就转而依赖于 `Insertion_sort`，后者可以针对比较短的子序列进行一种稳定的排序。

模板函数 `Buffered_merge_sort` 几乎与 `sort` 的行为完全相反。首先，它通过调用 `Insertion_sort`，对序列中长度为 `CHUNK_SIZE` 的子序列进行排序，然后将这些块合并起来形成一个更大的序列，一直到整个序列都稳定地排序好为止。每次调用 `Chunked_merge` 得到的合并后的块都是前一次调用的两倍那么大。合并首先是在临时缓冲区内完成的，最后才把它复制回最初的序列中去。

`partial_sort`

模板函数 `partial_sort` 调用 `Partial_sort`，它一开始会为序列前 n 个元素产生一个堆（ n 就是我们最终希望排序的元素的个数）。函数然后会重复地把堆的顶端元素*`first`（它也是堆中最大的一个元素）和序列中剩下的每个元素进行比较。通过调用 `Pop_heap`（将在本章的后面对它进行描述），把其中较小的元素和堆中的顶端元素互换，然后重新对堆进行整理。最后对得到的那个堆进行排序就可以得到我们所期望的序列的部分排序。

`partial_sort_copy`

模板函数 `partial_sort_copy` 和 `partial_sort` 的行为几乎相同，但它们之间还是有显著的区别。它调用模板函数 `Partial_sort_copy`，如果可能的

话，该函数会往目标序列中复制足够多的元素以填充它，然后把目标序列转化为一个堆。所有源序列中剩下的元素都会和这个新产生的堆的顶端元素 $*F2$ 进行比较。通过调用函数 `Adjust_heap`（同样也会在本章后面部分进行描述），把堆的顶端元素替换成它们中较小的那个元素，然后重新对堆进行整理。最后对得到的堆再进行一次排序，就可以得到我们所期望得到的部分排序的序列。

nth_element

模板函数 `nth_element` 执行一种通常的“分而治之”的排序方法。（我们已经在前面的 `Unguarded_partition` 中介绍过这种策略的使用。）然而，在这种情况下，算法将忽略掉所有不包括第 n 个元素的子区间。一旦它创建了一个能够容纳该元素的足够小的划分，它就会调用 `Insertion_sort` 来完成（部分排序）工作。

inplace_merge

模板函数 `inplace_merge` 综合了许多不同的策略。最基本的方法就是“分而治之”，它是由 `Buffered_merge` 所实现的。对于足够小的子区间，该函数会使用临时缓冲区来在复制时完成更常规的合并。注意，特殊的模板函数 `Merge_backward` 将在目标子序列的开始处和两个源子序列中的一个空穴重叠时调用。

当临时缓冲区不够大时，`Buffered_merge` 将会把两个序列中较大的那个分割成两个长度差不多相等的子序列。然后它会检测另一个序列中需要和这两个子序列进行互换的元素个数。迭代器 Fn 用来标记位于 M 左侧的子序列的分割点， Ln 用来标记右侧的分割点。模板函数 `Buffered_rotate` 通过旋转与 M 相关的子序列 $[Fn, Ln]$ 来实现真正的交换。于是，合并被简化为两个相似但却要小得多的问题。

除了在 `inplace_merge` 中使用外，`Buffered_rotate` 也可在 `stable_partition` 中使用（这一点我们已经在前面描述过了）。如果它能够把希望交换的两个子序列中较小的那个复制到临时缓冲区中去的话，它就会这么干。否则，它就会退回去使用更一般化的模板函数 `rotate` 来完成旋转。

push_heap

这个堆模板函数在序列中维护堆的准则：

- 第 0 个元素代表堆的顶端，它存储着堆中最大的元素。
- 如果存在着位于 $2 * k + 1$ 以及 $2 * k + 2$ 的元素，那么它们都小于位于 K 处的父元素。

模板函数 `push_heap` 假定堆序列被扩展时，新的元素会添加到序列的末端。它会复制这个值，并在堆的末端产生一个空穴。模板函数 `Push_heap` 会把这个空穴向堆的顶端传递，直到它足够高或这个新值不大于该空穴的父元素为止。参数 j 用来确定怎样才算是足够高，在这种情况下，它始终会是 0。

pop_heap

模板函数 `pop_heap` 调用 `Pop_heap` 把堆的顶端元素与末端元素相交

换，然后恢复堆的准则。Pop_heap 转而告诉 Adjust_heap 空穴现在位于第 0 个元素处，而最初存储于序列末端的值现在必须被插入到堆中适当的位置中去。

模板函数 Adjust_heap 会将空穴一直往堆的下部传递，直到它不再拥有子元素为止。然后调用 Push_heap 来将这个空穴原路返回，向堆的上面传递，直到找到一个合适的地方存储指定的值为止。Adjust_heap 在模板函数 Partial_sort_copy 中调用过，并且在其他有关堆的模板函数中也经常调用。

make_heap

模板函数 make_heap 只是简单地为堆序列的前半部分的每个元素创建一个空穴，然后调用 Adjust_heap 来在这些元素的子元素中建立堆的准则。

prev_permutation

模板函数 prev_permutation 会在序列中寻找最右边的排序好的相邻元素对。如果没有找到这样的元素对，函数就将通过反转整个序列，恢复到最初的排列，然后返回 false。否则，它就会重排序列中从元素对开始的其他元素，以产生下一个排列，然后返回 true。（请参考一下三至四个元素所能形成的所有排列，你将会发现这些排列是如何以字典中的降序产生的。）

next_permutation

模板函数 next_permutation 的行为和 prev_permutation 一样，不过它用来判断元素对的排序情况的谓词正好与 prev_permutation 使用的相反。

测试<algorithm>

talgorit.c

程序清单 6-2 列出了文件 talgorit.c。它只是粗略地测试了<algorithm>中定义的每个模板函数，并且按照往常的方式以相关的函数为一组进行测试。这些测试广泛地使用了在头文件<functional>中描述的函数对象（参见第 8 章）。例如，对 equal_to<char>的特化几乎和 operator==(char, char)是一样的。更加详细的使用如 bind2nd(equal_to('c'), ch) 返回的是一个函数对象 f，对 f(ch)来说，当 ch 是一个 char 对象时，它等价于 char == 'c'。

程序清单 6-2:

talgorit.c

```
// test <algorithm>
#include <cassert.h>
#include <ctype.h>
#include <iostream>
#include <string.h>
#include <algorithm>
#include <functional>
using namespace std;
```

```
// FUNCTION OBJECTS
equal_to<char> equf;
less<char> lessf;

// TEST SINGLE-ELEMENT TEMPLATE FUNCTIONS
void test_single(char *first, char *last)
    {assert(max('0', '2') == '2');
     assert(max('0', '2', lessf) == '2');
     assert(min('0', '2') == '0');
     assert(min('0', '2', lessf) == '0');
     strcpy(first, "abcdefg");
     swap(first[0], first[1]);
     assert(strcmp(first, "bacdefg") == 0);
     iter_swap(&first[0], &first[1]);
     assert(strcmp(first, "abcdefg") == 0); }

// TEST SEARCHING TEMPLATE FUNCTIONS
void test_find(char *first, char *last)
    {strcpy(first, "abccefg");
     assert(*max_element(first, last) == 'g');
     assert(*max_element(first, last, lessf) == 'g');
     assert(*min_element(first, last) == 'a');
     assert(*min_element(first, last, lessf) == 'a');
     assert(equal(first, last, first));
     assert(equal(first, last, first, equf));
     assert(lexicographical_compare(first, last - 1,
                                    first, last));
     assert(lexicographical_compare(first, last - 1,
                                    first, last, lessf));
     assert(mismatch(first, last, first).second == last);
     assert(mismatch(first, last, first, equf).second == last);

     assert(find(first, last, 'c') == first + 2);
     assert(find_if(first, last, bind2nd(equf, 'c'))
            == first + 2);
     assert(adjacent_find(first, last) == first + 2);
     assert(adjacent_find(first, last, equf) == first + 2);
     assert(count(first, last, 'c') == 2);
     assert(count_if(first, last, bind2nd(equf, 'c')) == 2);
     assert(search(first, last, first + 2, last) == first + 2);
     assert(search(first, last, first + 2, last, equf)
            == first + 2);
     assert(search_n(first, last, 2, 'c') == first + 2);
     assert(search_n(first, last, 2, 'c', equf) == first + 2); }
```

```

        assert(find_end(first, last, first + 2, last) == first + 2);
        assert(find_end(first, last, first + 2, last, equf)
               == first + 2);
        assert(find_first_of(first, last, first + 2, last)
               == first + 2);
        assert(find_first_of(first, last, first + 2, last, equf)
               == first + 2); }

        // TEST GENERATING TEMPLATE FUNCTIONS
size_t gen_count = 0;
void count_c(char ch)
{
    if (ch == 'c')
        ++gen_count; }
char gen_x()
{
    return ('x'); }
void test_generate(char *first, char *last, char *dest)
{
plus<char> plusf;
strcpy(first, "abccefg");
for_each(first, last, &count_c);
assert(gen_count == 2);
generate(first, first + 2, &gen_x);
assert(strcmp(first, "xxccefg") == 0);
generate_n(first + 3, last - first - 3, &gen_x);
assert(strcmp(first, "xxcxxxx") == 0);
assert(transform(first, last, dest,
                bind2nd(plusf, '\1')) == dest + 7);
assert(strcmp(dest, "yydyyyy") == 0);
assert(transform(first, last, "\1\1\1\2\1\1\1",
                dest, plusf) == dest + 7);
assert(strcmp(dest, "yydzyyy") == 0); }

        // TEST COPYING TEMPLATE FUNCTIONS
void test_copy(char *first, char *last, char *dest)
{
strcpy(first, "abcdefg");
copy(first, first + 3, first + 1);
assert(strcmp(first, "aaaaefg") == 0);
copy_backward(first + 2, first + 5, first + 3);
assert(strcmp(first, "eaaaefg") == 0);
fill(first, first + 3, 'x');
assert(strcmp(first, "xxxaeefg") == 0);
fill_n(first, 2, 'y');
assert(strcmp(first, "yyxaefg") == 0);
swap_ranges(first, first + 3, first + 4);
assert(strcmp(first, "efgayyx") == 0);
replace(first, last, 'y', 'c'); }

```

```
assert(strcmp(first, "efgaccx") == 0);
replace_if(first, last, bind2nd(equf, 'c'), 'z');
assert(strcmp(first, "efgazzx") == 0);
replace_copy(first, last, dest, 'z', 'c');
assert(strcmp(dest, "efgaccx") == 0);
replace_copy_if(first, last, dest, bind2nd(equf, 'z'), 'y');
assert(strcmp(dest, "efgayyx") == 0); }

// TEST MUTATING TEMPLATE FUNCTIONS
void test_mutate(char *first, char *last, char *dest)
{
    strcpy(first, "abcdefg"), strcpy(dest, first);
    remove_if(first, last, 'c');
    assert(strcmp(first, "abdefgg") == 0);
    remove_if(first, last, bind2nd(equf, 'd'));
    assert(strcmp(first, "abefggg") == 0);
    remove_copy(first, last, dest, 'e');
    assert(strcmp(dest, "abfgggg") == 0);
    remove_copy_if(first, last, dest, bind2nd(equf, 'e'));
    assert(strcmp(dest, "abfgggg") == 0);

    unique(dest, dest + 8);
    assert(strcmp(dest, "abfg") == 0);
    unique(dest, dest + 5, lessf);
    assert(strcmp(dest, "a") == 0);
    unique_copy(first, last + 1, dest);
    assert(strcmp(dest, "abefg") == 0);
    unique_copy(first, last + 1, dest, equf);
    assert(strcmp(dest, "abefg") == 0);
    reverse(first, last);
    assert(strcmp(first, "gggfeba") == 0);
    reverse_copy(first, last, dest);
    assert(strcmp(dest, "abefggg") == 0);
    rotate(first, first + 2, last);

    assert(strcmp(first, "gfebagg") == 0);
    rotate_copy(first, first + 2, last, dest);
    assert(strcmp(dest, "ebagggf") == 0);
    random_shuffle(first, last); }

// TEST ORDERING TEMPLATE FUNCTIONS
bool cmp_caseless(char c1, char c2)
{ return (tolower(c1) < tolower(c2)); }
```

```
void test_order(char *first, char *last, char *dest)
    {greater<char> greatf;
    strcpy(first, "gfedcba");
    stable_partition(first, last, bind2nd(lessf, 'd'));
    assert(strcmp(first, "cbagfed") == 0);
    assert(partition(first, last, bind2nd(equf, 'd'))
        == first + 1);
    sort(first, last);
    assert(strcmp(first, "abcdefg") == 0);
    sort(first, last, greatf);
    assert(strcmp(first, "gfedcba") == 0);
    partial_sort(first, first + 2, last);
    assert(first[0] == 'a' && first[1] == 'b');
    partial_sort(first, first + 2, last, greatf);
    assert(first[0] == 'g' && first[1] == 'f');
    stable_sort(first, last);
    assert(strcmp(first, "abcdefg") == 0);
    rotate(first, first + 2, last);
    inplace_merge(first, last - 2, last);
    assert(strcmp(first, "abcdefg") == 0);
    rotate(first, first + 2, last);
    inplace_merge(first, last - 2, last, lessf);
    assert(strcmp(first, "abcdefg") == 0);

    strcpy(dest, "uvwxyz");
    partial_sort_copy(first, last, dest, dest - 1);
    assert(strcmp(dest, "auvwxyz") == 0);
    partial_sort_copy(first, last, dest, dest + 1, greatf);
    assert(strcmp(dest, "guvwxyz") == 0);
    nth_element(first, first + 2, last, greatf);
    assert(first[2] == 'e');
    nth_element(first, first + 2, last);
    assert(first[2] == 'c');

    strcpy(first, "dCcBbBa");
    stable_sort(first, last, &cmp_caseless);
    assert(strcmp(first, "abBbCcd") == 0);
    merge(first + 5, last, first, first + 5, dest);
    assert(strcmp(dest, "abBbCcd") == 0);
    merge(first + 5, last, first, first + 5, dest, lessf);
    assert(strcmp(dest, "abBbCcd") == 0); }

// TEST SEARCHING TEMPLATE FUNCTIONS
void test_search(char *first, char *last)
{char val = 'c';
```

```
strcpy(first, "abcccfg");

assert(lower_bound(first, last, val) == first + 2);
assert(lower_bound(first, last, val, lessf) == first + 2);
assert(upper_bound(first, last, val) == first + 5);
assert(upper_bound(first, last, val, lessf) == first + 5);
assert(equal_range(first, last, val).first == first + 2);
assert(equal_range(first, last, val, lessf).second
      == first + 5);
assert(binary_search(first, last, val));
assert(binary_search(first, last, val, lessf));
assert(includes(first, last, first + 3, last));
assert(includes(first, last, first + 3, last, lessf)); }

// TEST SET TEMPLATE FUNCTIONS
void test_set(char *first, char *last, char *dest)
{strcpy(first, "abccefg"), strcpy(dest, first);
set_union(first, first + 3, first + 3, last, dest);
assert(strcmp(dest, "abcefgg") == 0);
set_union(first, first+3, first+3, last, dest, lessf);
assert(strcmp(dest, "abcefgg") == 0);
set_intersection(first, first+3, first+3, last, dest);
assert(strcmp(dest, "cbcefgg") == 0);
set_intersection(first, first + 3, first + 3, last,
dest, lessf);
assert(strcmp(dest, "cbcefgg") == 0);
set_difference(first, first+3, first+3, last, dest);
assert(strcmp(dest, "abcefgg") == 0);
set_difference(first, first + 3, first + 3, last,
dest, lessf);
assert(strcmp(dest, "abcefgg") == 0);
set_symmetric_difference(first, first+3, first+3, last,
dest);
assert(strcmp(dest, "abefggg") == 0);
set_symmetric_difference(first, first+3, first+3, last,
dest, lessf);
assert(strcmp(dest, "abefggg") == 0); }

// TEST HEAP TEMPLATE FUNCTIONS
void test_heap(char *first, char *last)
{strcpy(first, "abccefg");
make_heap(first, last);
assert(first[0] == 'g');
make_heap(first, last, lessf);
assert(first[0] == 'g');
```

```

        pop_heap(first, last);
        assert(last[-1] == 'g' && first[0] == 'f');
        pop_heap(first, last - 1, lessf);
        assert(last[-2] == 'f' && first[0] == 'e');
        push_heap(first, last - 1);
        assert(first[0] == 'f');
        push_heap(first, last, lessf);
        assert(first[0] == 'g');
        sort_heap(first, last);
        assert(strcmp(first, "abccefg") == 0);
        make_heap(first, last, lessf);
        sort_heap(first, last, lessf);
        assert(strcmp(first, "abccefg") == 0); }

        // TEST PERMUTING TEMPLATE FUNCTIONS
void test_permute(char *first, char *last)
{
    strcpy(first, "abcdefg");
    next_permutation(first, last);
    assert(strcmp(first, "abcdegf") == 0);
    next_permutation(first, last, lessf);
    assert(strcmp(first, "abcdfeg") == 0);
    prev_permutation(first, last);
    assert(strcmp(first, "abcdegf") == 0);
    prev_permutation(first, last, lessf);
    assert(strcmp(first, "abcdefg") == 0); }

        // TEST <algorithm>
int main()
{
    char buf[] = "abccefg";
    char dest[] = "1234567";
    char *first = buf, *last = buf + 7;
    test_single(first, last);
    test_find(first, last);
    test_generate(first, last, dest);
    test_copy(first, last, dest);
    test_mutate(first, last, dest);
    test_order(first, last, dest);
    test_search(first, last);
    test_set(first, last, dest);
    test_heap(first, last);
    test_permute(first, last);

    cout << "SUCCESS testing <algorithm>" << endl;
    return (0); }

```

如果一切顺利的话，测试程序将打印出：

```
SUCCESS testing <algorithm>
```

然后正常退出。

习题

习题6-1 模板函数adjacent_find的一个早期版本按照推测应该可以接受输入迭代器。为什么规范要把它改为前向迭代器？

习题6-2 在头文件<functional>中定义的模板函数对象less把less(x, y)定义为x < y。请描述一下与已有的、功能相同的模板函数max的定义相比，它有什么好处：

```
template<class T> inline  
const T& max(const T& x, const T& y)  
{return (max(x, y, less)); }
```

习题6-3 模板函数count有一个相应的模板函数count_if，它需要一个谓词。为什么我们不简单地重载count呢？请列出所有遵照这种模式的模板函数对。

习题6-4 某些浮点算法中至少拥有一种表示“非数（not a number, NaN）”的形式。如果x是一个NaN，y是任意一个浮点值（也包括NaN），那么对于诸如x < y、x != y、y >= x等形式的比较都将返回false。请列出这种规则在测试相等次序时的作用。

习题6-5 在什么情况下，你会考虑使用模板函数nth_element，而不是sort？什么时候你会执行翻转操作？

习题6-6 给定一个序列，其中有两个或多个元素拥有相等的次序，此时模板函数next_permutation和prev_permutation的行为会是如何？

习题6-7 [较难] 请试着改变模板函数next_permutation和prev_permutation，使得它们可以在所有可区分的排列中以所要求的次序循环，即使是序列中两个或多个元素拥有相等的次序时也是如此。

习题6-8 [特难] 试着改变模板函数sort，使得它可以正常地处理前向迭代器，而不是随机存取迭代器。

第7章 <numeric>

背景知识

头文件<numeric>很小。它只包括几个在序列中的元素上面进行简单数学运算的模板函数。按照前一章中所描述的，它可以放到头文件<algorithm>中去。实际上，在STL的最初版本中，大部分这样的算法都是放在一个头文件中的。

注意，这些算法并不一定局限于只能操作算术类型。通过给定合适的重载操作符，就像一些简单的算术运算一样，我们可以进行一些非常一般化的操作。例如，我们可以使用模板函数 accumulate 来连接 string 对象。通过定义我们自己的类以及函数对象，我们就可以扩展这些算法的定义范围。

功能描述

```
namespace std {
    template<class InIt, class T>
        T accumulate(InIt first, InIt last, T val);
    template<class InIt, class T, class Pred>
        T accumulate(InIt first, InIt last, T val, Pred pr);
    template<class InIt1, class InIt2, class T>
        T inner_product(InIt1 first1, InIt1 last1,
                          InIt2 first2, T val);
    template<class InIt1, class InIt2, class T,
             class Pred1, class Pred2>
        T inner_product(InIt1 first1, InIt1 last1,
                          InIt2 first2, T val, Pred1 pr1, Pred2 pr2);
    template<class InIt, class OutIt>
        OutIt partial_sum(InIt first, InIt last,
                            OutIt result);
    template<class InIt, class OutIt, class Pred>
        OutIt partial_sum(InIt first, InIt last,
                            OutIt result, Pred pr);
    template<class InIt, class OutIt>
        OutIt adjacent_difference(InIt first, InIt last,
                                    OutIt result);
```

```

template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
}
```

包含 STL 标准头文件<numeric>, 就得到了几个模板函数的定义, 它们在进行数值运算时很有用。对于这些模板的描述使用了大量对于其他算法也很常见的惯例。

口 accumulate

```

template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
```

第一个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值, 重复地将 val 替换成 $val + *I$ 。最后它返回 val。

第二个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值, 重复地将 val 替换成 $pr(val, *I)$ 。最后它返回 val。

口 adjacent_difference

```

template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
```

第一个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值, 从 result 开始存储一系列的值。第一个存储的值 val (如果有的话)是 $*I$ 。随后存储的值是 $*I - val$, 然后 val 被替换成 $*I$ 。该函数返回 result 递增了 $last - first$ 次后的结果。

第二个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值, 从 result 开始存储一系列的值。第一个存储的值 val (如果有的话)是 $*I$ 。随后存储的值是 $pr(*I, val)$, 然后 val 被替换成 $*I$ 。该函数返回 result 递增了 $last - first$ 次后的结果。

口 inner_product

```

template<class InIt1, class InIt2, class T>
    T inner_product(InIt1 first1, InIt1 last1,
        InIt2 first2, T val);
template<class InIt1, class InIt2, class T,
        class Pred1, class Pred2>
    T inner_product(InIt1 first1, InIt1 last1,
        InIt2 first2, T val, Pred1 pr1, Pred2 pr2);
```

第一个模板函数对处于区间[first1, last1)中的类型为 InIt1 的迭代器 I1 的每个值，重复地将 val 替换成 val + (*I1 * *I2)。在此处，类型为 InIt2 的迭代器 I2 等于 first2 + (I1 - first1)。函数返回 val。

第二个模板函数对处于区间[first1, last1)中的类型为 InIt1 的迭代器 I1 的每个值，重复地将 val 替换成 pr1(val, pr2(*I1, *I2))。在此处，类型为 InIt2 的迭代器 I2 等于 first2 + (I1 - first1)。函数返回 val。

口 partial_sum

```
template<class InIt, class OutIt>
OutIt partial_sum(InIt first, InIt last,
                    OutIt result);
template<class InIt, class OutIt, class Pred>
OutIt partial_sum(InIt first, InIt last,
                    OutIt result, Pred pr);
```

第一个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值，从 result 开始存储一系列的值。第一个存储的值 val（如果有的话）是*I。随后存储的值是 val + *I。该函数返回 result 递增了 last - first 次后的结果。

第二个模板函数对处于区间[first, last)中的类型为 InIt 的迭代器 I 的每个值，从 result 开始存储一系列的值。第一个存储的值 val（如果有的话）是*I。随后存储的值是 pr(val, *I)。该函数返回 result 递增了 last - first 次后的结果。

使用<numeric>

要想使用<numeric>中定义的任何模板函数，请把该头文件包含到程序中。下面给出的概括信息仅仅给出了每个算法的简短说明，提供了一个可用模板函数的综述。

accumulate

要使用 operator+或指定的二元操作符对序列中所有元素求和，调用 accumulate。

inner_product

要使用 operator+和 operator*或两个指定的二元操作符，对两个序列中相应元素的乘积求和，调用 inner_product。

partial_sum

要使用 operator+或指定的二元操作符产生一个求和序列，其中每个元素都是在已有的总数上再加上一个元素所得到的结果，调用 partial_sum。

adjacent_difference

要使用 operator-或指定的二元操作符产生一个存储相邻元素差的序列，调用 adjacent_difference。

实现<numeric>

numeric 程序清单 7-1 列出了文件 numeric。所有的模板函数都很简单，没有让人惊奇的地方。

```
程序清单 7-1: // numeric standard header
numeric #ifndef NUMERIC_
#define NUMERIC_
#include <iterator>
namespace std {
    // TEMPLATE FUNCTION accumulate
    template<class InIt, class T> inline
        T accumulate(InIt F, InIt L, T V)
    {for (; F != L; ++F)
        V = V + *F;
    return (V); }

    // TEMPLATE FUNCTION accumulate WITH BINOP
    template<class InIt, class T, class Bop> inline
        T accumulate(InIt F, InIt L, T V, Bop B)
    {for (; F != L; ++F)
        V = B(V, *F);
    return (V); }

    // TEMPLATE FUNCTION inner_product
    template<class InIt1, class InIt2, class T> inline
        T inner_product(InIt1 F, InIt1 L, InIt2 X, T V)
    {for (; F != L; ++F, ++X)
        V = V + *F * *X;
    return (V); }

    // TEMPLATE FUNCTION inner_product WITH BINOPS
    template<class InIt1, class InIt2, class T,
             class Bop1, class Bop2> inline
        T inner_product(InIt1 F, InIt1 L, InIt2 X, T V,
                        Bop1 B1, Bop2 B2)
    {for (; F != L; ++F, ++X)
        V = B1(V, B2(*F, *X));
    return (V); }

    // TEMPLATE FUNCTION partial_sum
    template<class InIt, class OutIt> inline
        OutIt partial_sum(InIt F, InIt L, OutIt X)
    {return (F == L ? X
                   : Partial_sum(F, L, X, Val_type(F))); }

    template<class InIt, class OutIt, class T> inline
```

```

    OutIt Partial_sum(Init F, Init L, OutIt X, T *)
    {T V = *F;
     for (*X = V; ++F != L; *++X = V)
        V = V + *F;
     return (++X); }

        // TEMPLATE FUNCTION partial_sum WITH BINOP
template<class Init, class OutIt, class Bop> inline
    OutIt partial_sum(Init F, Init L, OutIt X, Bop B)
    {return (F == L ? X
        : Partial_sum(F, L, X, B, Val_type(F))); }

template<class Init, class OutIt, class Bop, class T> inline
    OutIt Partial_sum(Init F, Init L, OutIt X, Bop B, T *)
    {T V = *F;
     for (*X = V; ++F != L; *++X = V)
        V = B(V, *F);
     return (++X); }

        // TEMPLATE FUNCTION adjacent_difference
template<class Init, class OutIt> inline
    OutIt adjacent_difference(Init F, Init L, OutIt X)
    {return (F == L ? X
        : Adjacent_difference(F, L, X, Val_type(F))); }

template<class Init, class OutIt, class T> inline
    OutIt Adjacent_difference(Init F, Init L, OutIt X, T *)
    {T V = *F;
     for (*X = V; ++F != L; )
        {T Tmp = *F;
         *++X = Tmp - V;
         V = Tmp; }
     return (++X); }

        // TEMPLATE FUNCTION adjacent_difference WITH BINOP
template<class Init, class OutIt, class Bop> inline
    OutIt adjacent_difference(Init F, Init L, OutIt X, Bop B)
    {return (F == L ? X
        : Adjacent_difference(F, L, X, B, Val_type(F))); }

template<class Init, class OutIt, class Bop, class T> inline
    OutIt Adjacent_difference(Init F, Init L, OutIt X,
        Bop B, T *)
    {T V = *F;
     for (*X = V; ++F != L; )
        {T Tmp = *F;
         *++X = B(Tmp, V);
         V = Tmp; }
     return (++X); }

} /* namespace std */
#endif /* NUMERIC_ */

```

测试<numeric>

tnumeric.c 程序清单 7-2 列出了文件 tnumeric.c。由于头文件很小，所以它很简单。惟一值得注意的就是在 adjacent_difference 的第二个版本中，使用了函数对象 addr 来替代通常使用的 operator -。我们选择这种形式来表明，通过给它一个函数对象参数，模板函数就不一定非得执行从它的名字中可以猜测出来的操作。（同样也可产生更有趣的结果。）

```
程序清单 7-2: // test <numeric>
tnumeric.c #include <assert.h>
#include <iostream>
#include <string.h>
#include <functional>
#include <numeric>
using namespace std;

// FUNCTION OBJECTS
multiplies<char> multipliesf;
plus<char> plusf;

// TEST <numeric>
int main()
{
    char buf[] = "\1\2\3\4\5\6";
    char dest[] = "123456";
    char *first = buf, *last = buf + 6;
    char val = 0;
    assert(accumulate(first, last, val) == 21);
    assert(accumulate(first, last, val, plusf) == 21);
    assert(inner_product(first, last, first, val) == 91);
    assert(inner_product(first, last, first, val,
                         plusf, multipliesf) == 91);
    partial_sum(first, last, dest);
    assert(strcmp(dest, "\1\3\6\12\17\25") == 0);
    adjacent_difference(first, last, dest);
    assert(strcmp(dest, "\1\1\1\1\1\1") == 0);
    partial_sum(first, last, dest, plusf);
    assert(strcmp(dest, "\1\3\6\12\17\25") == 0);
    adjacent_difference(first, last, dest, plusf);
    assert(strcmp(dest, "\1\3\5\7\11\13") == 0);

    cout << "SUCCESS testing <numeric>" << endl;
    return (0); }
```

如果一切顺利的话，测试程序将打印出：

```
SUCCESS testing <numeric>
```

然后正常退出。

习题

- 习题7-1 重写模板函数 `adjacent_difference` 的两个版本，消除其中对于临时对象 `tmp` 的使用。
是否也能够消除临时对象 `val`? 如果不能，为什么?
- 习题7-2 展示如何使用模板函数 `accumulate` 来连接 `string` 对象的序列。
- 习题7-3 展示如何使用模板函数 `accumulate` 来测试一个序列是否仅由从不以降序排序的正的元素组成。
- 习题7-4 写出一个模板函数，从一个给定的行中产生 Pascal 三角的下一行。你会使用`<numeric>`中定义的那个模板函数?
- 习题7-5 [较难] 指定一个模板函数来将两个矩阵相乘（每个矩阵都以一系列的行的方式存储着），并把结果存储到第三个序列中。该模板函数需要使用什么种类的迭代器?
- 习题7-6 [特难] 实现上题中的模板函数。

第 8 章 <functional>

背景知识

函数对象极大地增强了 STL 中算法的能力。每个函数对象都封装了一个关键性的谓词或者是其他的计算，这使得它可以与那些单调的实现算法的模板函数簿记区分开来。当然，我们也可以写出自己的函数对象。有时，我们可以把一个指向已有函数的指针作为一个函数对象使用。但更多的情况下，我们只需要一个计算普通简单表达式的函数对象。

这也就是引入头文件<functional>的原因。它定义了一些模板类，用以声明函数对象。在前面章节的测试部分中，我们使用了部分函数对象用以测试算法模板函数。(参见第 6 章和第 7 章。)

unary_function

最简单的模板类所产生的函数对象被设计为仅需要一个单独的参数就可以调用。也就是说，它们的行为与一元（unary）函数一样。这些特定类的基类是模板类 unary_function:

```
template<class A, class R>
struct unary_function {
    typedef A argument_type;
    typedef R result_type;
};
```

这样做使得所有在头文件<functional>中定义的一元函数对象都可以从基类 unary_function 中派生得到成员类型 argument_type，并以此作为自己的参数类型。同样，它也可以以类似的方法得到成员类型 result_type 作为返回值。

negate

例如，X 是模板类 negate<T>的一个对象，假设 a 的类型为 T，那么我们就可以确信调用 X(a)将返回 -a。该模板类的简单定义如下：

```
template<class T>
struct negate : unary_function<T, T> {
    T operator()(const T& x) const
        {return (-x); }
};
```

注意，在这个模板类中，参数的类型和返回值的类型是一致的，在大多数情况下都是如此。

binary_function

另一种比较简单的模板类所产生的函数对象被设计为调用时带有两个类型相同的参数。也就是说，它们的行为和二元函数一样。这些特殊的类都派生自模板类 `binary_function`:

```
template<class A1, class A2, class R>
struct binary_function {
    typedef A1 first_argument_type;
    typedef A2 second_argument_type;
    typedef R result_type;
};
```

和前面所讨论的一样，这样做使得任意在头文件`<functional>`中定义的二元函数对象都可以从基类 `binary_function` 中继承得到成员类型 `first_argument_type` 作为第一个参数类型。同样，用类似的方法可得到成员类型 `second_argument_type` 作为第二个成员类型以及成员类型 `result_type` 作为返回值类型。

plus

例如，当 `X` 是模板类 `plus<T>` 的一个对象时，假设 `a` 和 `b` 的类型为 `T`，那么我们可以确信调用 `X(a, b)` 将返回 `a + b`。该模板类的简单定义如下：

```
template<class T>
struct plus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const
        {return (x + y); }
};
```

我们将在本章的后面概括这些简单的一元及二元函数对象（参见本章的“使用`<functional>`”一节）。

复合函数对象

头文件`<functional>`中还定义了这样一些模板类：它们描述的函数对象又存储了其他函数对象。我们把这样的函数对象叫做“复合函数对象（compound function object）”。对复合函数对象的调用将导致对所存储的函数对象的调用。

**binder1st
bind1st**

一个很有趣的例子是模板类 `binder1st`，它里面存储着一个二元函数对象 `f` 以及调用 `f` 所需要的第一个参数 `a`。从 `binder1st` 得到的一元函数对象 `X` 可以确保调用 `X(b)` 会返回 `f(a, b)`。你可能会在模板函数 `transform`（定义于`<algorithm>`中）中使用它，以此来为序列中的每个元素加上一个常数。为了方便起见，我们可以使用模板函数 `bind1st` 来产生一个类 `binder1st<T>` 的对象，如调用 `bind1st(plus<double>(), 3.0)`。从这个表达式中我们可以得到一个函数对象，如果使用一个类型为 `double` 的参数对其进行调用，得到的返回值就是该参数加上 3 后得到的值。

指针函数对象

最后，头文件`<functional>`中还定义了一些模板类来存储指向函数的指针或引用。我们把它们叫做“指针函数对象”。调用这样的对象会导

致对它所存储的指针或引用的调用。不幸的是，对于这样的模板类的要求是无止境的。我们需要为指针和引用、全局函数和成员函数以及常量成员函数、无参函数和一元函数以及二元函数等等定义不同的模板类。

STL 所能做的只是提供了一些简单的、最有可能在实际中使用的情况。如果你需要一个 STL 没有提供的版本，最好以一个已有的相似版本为模型创建一个。

功能描述

```
namespace std {
    template<class Arg, class Result>
        struct unary_function;
    template<class Arg1, class Arg2, class Result>
        struct binary_function;
    template<class T>
        struct plus;
    template<class T>
        struct minus;
    template<class T>
        struct multiples;
    template<class T>
        struct divides;
    template<class T>
        struct modulus;
    template<class T>
        struct negate;
    template<class T>
        struct equal_to;
    template<class T>
        struct not_equal_to;
    template<class T>
        struct greater;
    template<class T>
        struct less;
    template<class T>
        struct greater_equal;
    template<class T>
        struct less_equal;
    template<class T>
        struct logical_and;
    template<class T>
        struct logical_or;
    template<class T>
```

```

        struct logical_not;
template<class Pred>
        struct unary_negate;
template<class Pred>
        struct binary_negate;
template<class Pred>
        class binder1st;
template<class Pred>
        class binder2nd;
template<class Arg, class Result>
        class pointer_to_unary_function;
template<class Arg1, class Arg2, class Result>
        class pointer_to_binary_function;
template<class R, class T>
        struct mem_fun_t;
template<class R, class T, class A>
        struct mem_fn1_t;
template<class R, class T>
        struct const_mem_fun_t;
template<class R, class T, class A>
        struct const_mem_fn1_t;
template<class R, class T>
        struct mem_fun_ref_t;
template<class R, class T, class A>
        struct mem_fn1_ref_t;
template<class R, class T>
        struct const_mem_fun_ref_t;
template<class R, class T, class A>
        struct const_mem_fn1_ref_t;

        // TEMPLATE FUNCTIONS
template<class Pred>
        unary_negate<Pred> not1(const Pred& pr);
template<class Pred>
        binary_negate<Pred> not2(const Pred& pr);
template<class Pred, class T>
        binder1st<Pred> bind1st(const Pred& pr, const T& x);
template<class Pred, class T>
        binder2nd<Pred> bind2nd(const Pred& pr, const T& x);
template<class Arg, class Result>
        pointer_to_unary_function<Arg, Result>
            ptr_fun(Result (*)(Arg));
template<class Arg1, class Arg2, class Result>
        pointer_to_binary_function<Arg1, Arg2, Result>
            ptr_fun(Result (*)(Arg1, Arg2));

```

```

template<class R, class T>
    mem_fun_t<R, T> mem_fun(R (T::*pm) ());
template<class R, class T, class A>
    mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A arg));
template<class R, class T>
    const_mem_fun_t<R, T> mem_fun(R (T::*pm) () const);
template<class R, class T, class A>
    const_mem_fun1_t<R, T, A> mem_fun(R (T::*pm)(A arg)
const);
template<class R, class T>
    mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm) ());
template<class R, class T, class A>
    mem_fun1_ref_t<R, T, A>
        mem_fun_ref(R (T::*pm)(A arg));
template<class R, class T>
    const_mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm) () const);
template<class R, class T, class A>
    const_mem_fun1_ref_t<R, T, A>
        mem_fun_ref(R (T::*pm)(A arg) const);
};

```

包含 STL 标准头文件<functional>, 我们就得到了几个帮助创建函数对象（即定义 operator() 的类型的对象）的模板的定义。函数对象可以是一个函数指针，但更多的情况下，函数对象可以存储在函数调用时用得着的附加信息。

■ binary_function

```

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

```

该模板类可以当作一个基类，方便我们定义有如下形式的成员函数的类：

```

result_type operator()(const first_argument_type&,
    const second_argument_type&) const

```

这样，所有这样的二元函数都可以把 first_argument_type 当作第一个参数类型，second_argument_type 当作第二个参数类型，result_type 当作返回值类型了。

```

□ binary_negate

    template<class Pred>
        class binary_negate
            : public binary_function<
                typename Pred::first_argument_type,
                typename Pred::second_argument_type, bool> {
    public:
        explicit binary_negate(const Pred& pr);
        bool operator()(
            const typename Pred::first_argument_type& x,
            const typename Pred::second_argument_type& y) const;
    };

```

该模板类存储 pr 的一个拷贝 (pr 必须是一个二元函数对象)。它定义它的成员函数 operator()返回!pr(x, y)。

```

□ bind1st

    template<class Pred, class T>
        binder1st<Pred> bind1st(const Pred& pr, const T& x);

    该函数返回 binder1st<Pred>(pr, typename Pred::first_argument_type(x));

```

```

□ bind2nd

    template<class Pred, class T>
        binder2nd<Pred> bind2nd(const Pred& pr, const T& y);

    该函数返回 binder2nd<Pred>(pr, typename Pred::second_argument_type(y));

```

```

□ binder1st

    template<class Pred>
        class binder1st
            : public unary_function<
                typename Pred::second_argument_type,
                typename Pred::result_type> {
    public:
        typedef typename Pred::second_argument_type argument_type;
        typedef typename Pred::result_type result_type;
        binder1st(const Pred& pr,
            const typename Pred::first_argument_type& x);
        result_type operator()(const argument_type& y) const;
    protected:
        Pred op;
        typename Pred::first_argument_type value;
    };

```

该模板类在 op 中存储 pr 的一个拷贝 (pr 必须是一个二元函数对象), 在 value 中存储 x 的一个拷贝。它定义它的成员函数 operator() 返回 op(value, y)。

¶ binder2nd

```
template<class Pred>
class binder2nd
    : public unary_function<
        typename Pred::first_argument_type,
        typename Pred::result_type> {
public:
    typedef typename Pred::first_argument_type
    argument_type;
    typedef typename Pred::result_type result_type;
    binder2nd(const Pred& pr,
              const typename Pred::second_argument_type& y);
    result_type operator()(const argument_type& x) const;
protected:
    Pred op;
    typename Pred::second_argument_type value;
};
```

该模板类在 op 中存储 pr 的一个拷贝 (pr 必须是一个二元函数对象), 在 value 中存储 y 的一个拷贝。它定义它的成员函数 operator() 返回 op(x, value)。

¶ const_mem_fun_t

```
template<class R, class T>
struct const_mem_fun_t
    : public unary_function<T *, R> {
    explicit const_mem_fun_t(R (T::*pm)() const);
    R operator()(const T *p) const;
};
```

该模板类在它的一个私有成员对象中存储了 pm 的一个拷贝 (pm 必须是一个指向类 T 的成员函数的指针)。它定义它的成员函数 operator() 返回 (p->*pm)() const。

¶ const_mem_fun_ref_t

```
template<class R, class T>
struct const_mem_fun_ref_t
    : public unary_function<T, R> {
    explicit const_mem_fun_t(R (T::*pm)() const);
    R operator()(const T& x) const;
};
```

该模板类在它的一个私有成员对象中存储了 pm 的一个拷贝（pm 必须是一个指向类 T 的成员函数的指针）。它定义它的成员函数 operator() 返回(x.*pm)(const)。

```
#include <functional>
namespace std {
    template<class R, class T, class A>
    struct const_mem_fnl_t
        : public binary_function<T *, A, R> {
        explicit const_mem_fnl_t(R (T::*pm)(A) const);
        R operator()(const T *p, A arg) const;
    };
}
```

该模板类在它的一个私有成员对象中存储了 pm 的一个拷贝（pm 必须是一个指向类 T 的成员函数的指针）。它定义它的成员函数 operator() 返回(p->*pm)(arg)const。

```
#include <functional>
namespace std {
    template<class R, class T, class A>
    struct const_mem_fnl_ref_t
        : public binary_function<T, A, R> {
        explicit const_mem_fnl_ref_t(R (T::*pm)(A) const);
        R operator()(const T& x, A arg) const;
    };
}
```

该模板类在它的一个私有成员对象中存储了 pm 的一个拷贝（pm 必须是一个指向类 T 的成员函数的指针）。它定义它的成员函数 operator() 返回(x.*pm)(arg)const。

```
#include <functional>
namespace std {
    template<class T>
    struct divides : public binary_function<T, T, T> {
        T operator()(const T& x, const T& y) const;
    };
}
```

该模板类定义它的成员函数 operator() 返回 x / y。

```
#include <functional>
namespace std {
    template<class T>
    struct equal_to
        : public binary_function<T, T, bool> {
        bool operator()(const T& x, const T& y) const;
    };
}
```

该模板类定义它的成员函数 operator() 返回 x == y。

```
#include <functional>
namespace std {
    template<class T>
```

```
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x > y$ 。如果 T 是一个对象指针类型，那么该成员函数定义一个完全排序。

¶ greater_equal

```
template<class T>
struct greater_equal
    : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x \geq y$ 。如果 T 是一个对象指针类型，那么该成员函数定义一个完全排序。

¶ less

```
template<class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x < y$ 。如果 T 是一个对象指针类型，那么该成员函数定义一个完全排序。

¶ less_equal

```
template<class T>
struct less_equal
    : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x \leq y$ 。如果 T 是一个对象指针类型，那么该成员函数定义一个完全排序。

¶ logical_and

```
template<class T>
struct logical_and
    : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x \&& y$ 。

¶ logical_not

```
template<class T>
```

```

    struct logical_not : public unary_function<T, bool> {
        bool operator()(const T& x) const;
    };

```

该模板类定义它的成员函数 operator()返回!x。

口 logical_or

```

template<class T>
struct logical_or
    : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};

```

该模板类定义它的成员函数 operator()返回 x || y。

口 mem_fun

```

template<class R, class T>
mem_fun_t<R, T> mem_fun(R (T::*pm) ());
template<class R, class T, class A>
mem_fun1_t<R, T, A> mem_fun(R (T::*pm) (A));
template<class R, class T>
const_mem_fun_t<R, T> mem_fun(R (T::*pm) () const);
template<class R, class T, class A>
const_mem_fun1_t<R, T, A> mem_fun(R (T::*pm) (A) const);

```

该模板函数把 pm 转化为返回值的类型，然后将它返回。

口 mem_fun_ref

```

template<class R, class T>
mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm) ());
template<class R, class T, class A>
mem_fun1_ref_t<R, T, A> mem_fun_ref(R (T::*pm) (A));
template<class R, class T>
const_mem_fun_ref_t<R, T>
    mem_fun_ref(R (T::*pm) () const);
template<class R, class T, class A>
const_mem_fun1_ref_t<R, T, A>
    mem_fun_ref(R (T::*pm) (A) const);

```

该模板函数把 pm 转化为返回值的类型，然后将它返回。

口 mem_fun_t

```

template<class R, class T>
struct mem_fun_t : public unary_function<T *, R> {
    explicit mem_fun_t(R (T::*pm) ());
    R operator()(T *p) const;
};

```

该模板类在它的一个私有成员对象中存储 pm 的一个拷贝 (pm 必须是一个指向类 T 的成员函数的指针)。它定义它的成员函数 operator() 返回(p->*pm)()。

口 mem_fun_ref_t

```
template<class R, class T>
struct mem_fun_ref_t
    : public unary_function<T, R> {
    explicit mem_fun_t(R (T::*pm)());
    R operator()(T& x) const;
};
```

该模板类在它的一个私有成员对象中存储 pm 的一个拷贝 (pm 必须是一个指向类 T 的成员函数的指针)。它定义它的成员函数 operator() 返回(x.*pm)()。

口 mem_fun1_t

```
template<class R, class T, class A>
struct mem_fun1_t
    : public binary_function<T *, A, R> {
    explicit mem_fun1_t(R (T::*pm)(A));
    R operator()(T *p, A arg) const;
};
```

该模板类在它的一个私有成员对象中存储 pm 的一个拷贝 (pm 必须是一个指向类 T 的成员函数的指针)。它定义它的成员函数 operator() 返回(p->*pm)(arg)。

口 mem_fun1_ref_t

```
template<class R, class T, class A>
struct mem_fun1_ref_t
    : public binary_function<T, A, R> {
    explicit mem_fun1_ref_t(R (T::*pm)(A));
    R operator()(T& x, A arg) const;
};
```

该模板类在它的一个私有成员对象中存储 pm 的一个拷贝 (pm 必须是一个指向类 T 中成员函数的指针)。它定义它的成员函数 operator() 返回(x.*pm)(arg)。

口 minus

```
template<class T>
struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x - y$ 。

口 modulus

```
template<class T>
struct modulus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x \% y$ 。

口 multiplies

```
template<class T>
struct multiplies : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x * y$ 。

口 negate

```
template<class T>
struct negate : public unary_function<T, T> {
    T operator()(const T& x) const;
};
```

该模板类定义它的成员函数 operator()返回 $-x$ 。

口 not1

```
template<class Pred>
unary_negate<Pred> not1(const Pred& pr);
```

该模板函数返回 unary_negate<Pred>(pr)。

口 not2

```
template<class Pred>
binary_negate<Pred> not2(const Pred& pr);
```

该模板函数返回 binary_negate<Pred>(pr)。

口 not_equal_to

```
template<class T>
struct not_equal_to
    : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x != y$ 。

口 plus

```
template<class T>
```

```
struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
};
```

该模板类定义它的成员函数 operator()返回 $x + y$ 。

口 pointer_to_binary_function

```
template<class Arg1, class Arg2, class Result>
class pointer_to_binary_function
    : public binary_function<Arg1, Arg2, Result> {
public:
    explicit pointer_to_binary_function(
        Result (*pf)(Arg1, Arg2));
    Result operator()(const Arg1 x, const Arg2 y) const;
};
```

该模板类存储 pf 的一个拷贝。它定义它的成员函数 operator()返回 $(*pf)(x,y)$ 。

口 pointer_to_unary_function

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

该模板类存储 pf 的一个拷贝。它定义它的成员函数 operator()返回 $(*pf)(x)$ 。

口 ptr_fun

```
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*pf)(Arg));
template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*pf)(Arg1, Arg2));
```

第一个模板函数返回 pointer_to_unary_function<Arg, Result>(pf)。

第二个模板函数返回 pointer_to_binary_function<Arg1, Arg2, Result>(pf)。

口 unary_function

```
template<class Arg, class Result>
struct unary_function {
```

```
typedef Arg argument_type;
typedef Result result_type;
};
```

该模板类可以当作一个基类，方便我们定义有着如下形式的成员函数的类：

```
result_type operator()(const argument_type&) const
```

这样，所有这样的一元函数都可以把 argument_type 当作它们的惟一参数类型，把 result_type 当作它们的返回值类型。

口 unary_negate

```
template<class Pred>
class unary_negate
: public unary_function<
    typename Pred::argument_type,
    bool> {
public:
    explicit unary_negate(const Pred& pr);
    bool operator()(const typename Pred::argument_type& x) const;
};
```

该模板类存储 pr 的一个拷贝（pr 必须是一个一元函数对象）。它定义它的成员函数 operator() 返回 !pr(x)。

使用<functional>

unary_function

如果想使用头文件<functional>中定义的模板类或者模板函数，请把它包含到程序中。我们将用 3 张表来概括它们。例如，表 8-1 中列出了在该头文件中定义的所有简单函数对象。如果你希望向其中有所增补的话，请确保所加的所有一元函数对象都是由模板类 unary_function 派生而来的^①，如：

```
template<class T>
struct logical_not : unary_function<T, bool> {
    bool operator()(const T& x) const
    {return (!x); }
};
```

^① 这也是 C++ 标准中所要求的，不过在实际中，其实可以不遵循这个规则。因为真正的重点在于：你所给出的函数对象必须具有和 unary_function 一样的成员类型，而让它派生自 unary_function 只是一种很方便的做法而已。——译者注

binary_function

同样，应该让所有的二元函数对象派生自模板类 `binary_function`^②，如：

```
template<class T>
struct plus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const
    {return (x + y); }
};
```

表 8-1:
函数对象

声 明	调用成员函数	返 回
<code>plus<T> X;</code>	<code>X(a, b)</code>	<code>a + b</code>
<code>minus<T> X;</code>	<code>X(a, b)</code>	<code>a - b</code>
<code>multiplies<T> X;</code>	<code>X(a, b)</code>	<code>a * b</code>
<code>divides<T> X;</code>	<code>X(a, b)</code>	<code>a / b</code>
<code>modulus<T> X;</code>	<code>X(a, b)</code>	<code>a % b</code>
<code>equal_to<T> X;</code>	<code>X(a, b)</code>	<code>a == b</code>
<code>not_equal_to<T> X;</code>	<code>X(a, b)</code>	<code>a != b</code>
<code>greater<T> X;</code>	<code>X(a, b)</code>	<code>a > b</code>
<code>less<T> X;</code>	<code>X(a, b)</code>	<code>a < b</code>
<code>greater_equal<T> X;</code>	<code>X(a, b)</code>	<code>a >= b</code>
<code>less_equal<T> X;</code>	<code>X(a, b)</code>	<code>a <= b</code>
<code>logical_and<T> X;</code>	<code>X(a, b)</code>	<code>a && b</code>
<code>logical_or<T> X;</code>	<code>X(a, b)</code>	<code>a b</code>
<code>logical_not<T> X;</code>	<code>X(a)</code>	<code>!a</code>
<code>negate<T> X;</code>	<code>X(a)</code>	<code>-a</code>

注释：a 和 b 的类型都为 T。

复合函数
对象

表 8-2 列出了在头文件<functional>中定义的所有复合函数对象。注意，这些模板类利用了在基类 `unary_function` 及 `binary_function` 中定义的成员类型。还要注意，每个模板类都有一个相应的模板函数（也列在该表中）用来实时地产生一个适当的函数对象。

表 8-2:
复合函数
对象

声 明 / 调 用	调用成员函数	返 回
<code>binary_negate<F> X(f);</code> <code>X = not2(f);</code>	<code>X(a, b)</code>	<code>'f(a, b)</code>
<code>unary_negate<G> X(g);</code> <code>X = not1(g);</code>	<code>X(a)</code>	<code>'g(a)</code>
<code>binder1st<F> X(f, a);</code> <code>X = bind1st(f, a);</code>	<code>X(b)</code>	<code>f(a, b)</code>

② 同译注①。——译者注

续表

声明 / 调用	调用成员函数	返 回
binder2nd<F> X(f, b); X = bind2nd(f, b);	X(a)	f(a, b)

注释: f 的类型 F 派生自 binary_function, g 的类型 G 派生自 unary_function。

指针函数 对象

最后, 表 8-3 列出了在头文件<functional>中定义的所有指针函数对象。每个模板类同样有一个相应的模板函数用以实时地产生所需的指针函数对象。

表 8-3:
指针函数
对象

声 明 / 调 用	调用成员函数	返 回
pointer_to_binary_function<T1, T2, R> X(pf); X = ptr_fun(pf);	X(a, b)	(*pf)(a, b)
pointer_to_unary_function<T1, R> X(pf); X = ptr_fun(pf);	X(a)	(*pf)(a)
mem_fun1_t<R, T, T1> X(pm); X = mem_fun(pm);	X(p, a)	(p->*pm)(a)
mem_fun1_ref_t<R, T, T1> X(pm); X = mem_fun_ref(pm);	X(t, a)	(t.*pm)(a)
mem_fun_t<R, T> X(pm); X = mem_fun(pm);	X(p)	(p->*pm)()
mem_fun_ref_t<R, T> X(pm); X = mem_fun_ref(pm);	X(t)	(t.*pm)()
const_mem_fun1_t<R, T, T1> X(pcm); X = mem_fun(pcm);	X(p, a)	(p->*pcm)(a)
const_mem_fun1_ref_t<R, T, T1> X(pcm); X = mem_fun_ref(pcm);	X(t, a)	(t.*pcm)(a)
const_mem_fun_t<R, T> X(pcm); X = mem_fun(pcm);	X(p)	(p->*pcm)()
const_mem_fun_ref_t<R, T> X(pcm); X = mem_fun_ref(pcm);	X(t)	(t.*pcm)()

注释: a 的类型为 T1, b 的类型为 T2, *pf 是一个函数, *pm 是一个成员函数, *pcm 是一个常量成员函数, p 的类型是 T *, t 的类型是 T&, 返回值的类型为 R。

实现<functional>

functional 程序清单 8-1 中列出了文件 functional。其中虽然存在着许多琐碎的细节，但没有一个模板类或函数会让人觉得吃惊。我们在此简单地列出了代码，而没有给出更多的注释。

```
程序清单 8-1: // functional standard header
functional
#ifndef FUNCTIONAL_
#define FUNCTIONAL_
namespace std {
    // TEMPLATE STRUCT unary_function
    template<class A, class R>
    struct unary_function {
        typedef A argument_type;
        typedef R result_type;
    };

    // TEMPLATE STRUCT binary_function
    template<class A1, class A2, class R>
    struct binary_function {
        typedef A1 first_argument_type;
        typedef A2 second_argument_type;
        typedef R result_type;
    };

    // TEMPLATE STRUCT plus
    template<class T>
    struct plus : binary_function<T, T, T> {
        T operator()(const T& X, const T& Y) const
        {return (X + Y); }
    };

    // TEMPLATE STRUCT minus
    template<class T>
    struct minus : binary_function<T, T, T> {
        T operator()(const T& X, const T& Y) const
        {return (X - Y); }
    };

    // TEMPLATE STRUCT multiplies
    template<class T>
    struct multiplies : binary_function<T, T, T> {
```

```
T operator()(const T& X, const T& Y) const
    {return (X * Y); }
};

// TEMPLATE STRUCT divides
template<class T>
struct divides : binary_function<T, T, T> {
    T operator()(const T& X, const T& Y) const
        {return (X / Y); }
};

// TEMPLATE STRUCT modulus
template<class T>
struct modulus : binary_function<T, T, T> {
    T operator()(const T& X, const T& Y) const
        {return (X % Y); }
};

// TEMPLATE STRUCT negate
template<class T>
struct negate : unary_function<T, T> {
    T operator()(const T& X) const
        {return (-X); }
};

// TEMPLATE STRUCT equal_to
template<class T>
struct equal_to : binary_function<T, T, bool> {
    bool operator()(const T& X, const T& Y) const
        {return (X == Y); }
};

// TEMPLATE STRUCT not_equal_to
template<class T>
struct not_equal_to : binary_function<T, T, bool> {
    bool operator()(const T& X, const T& Y) const
        {return (X != Y); }
};

// TEMPLATE STRUCT greater
template<class T>
struct greater : binary_function<T, T, bool> {
    bool operator()(const T& X, const T& Y) const
        {return (X > Y); }
};
```

```
// TEMPLATE STRUCT less
template<class T>
    struct less : binary_function<T, T, bool> {
        bool operator()(const T& X, const T& Y) const
            {return (X < Y); }
    };

// TEMPLATE STRUCT greater_equal
template<class T>
    struct greater_equal : binary_function<T, T, bool>{
        bool operator()(const T& X, const T& Y) const
            {return (X >= Y); }
    };

// TEMPLATE STRUCT less_equal
template<class T>
    struct less_equal : binary_function<T, T, bool> {
        bool operator()(const T& X, const T& Y) const
            {return (X <= Y); }
    };

// TEMPLATE STRUCT logical_and
template<class T>
    struct logical_and : binary_function<T, T, bool> {
        bool operator()(const T& X, const T& Y) const
            {return (X && Y); }
    };

// TEMPLATE STRUCT logical_or
template<class T>
    struct logical_or : binary_function<T, T, bool> {
        bool operator()(const T& X, const T& Y) const
            {return (X || Y); }
    };

// TEMPLATE STRUCT logical_not
template<class T>
    struct logical_not : unary_function<T, bool> {
        bool operator()(const T& X) const
            {return (!X); }
    };

// TEMPLATE CLASS unary_negate
template<class Ufn>
    class unary_negate
```

```
:public unary_function<typename Ufn::argument_type, bool>{
public:
    explicit unary_negate(const Ufn& X)
        : Fn(X) {}
    bool operator()(const typename Ufn::argument_type& X)
        const {return (!Fn(X)); }
protected:
    Ufn Fn;
};

// TEMPLATE FUNCTION not1
template<class Ufn> inline
unary_negate<Ufn> not1(const Ufn& X)
    {return (unary_negate<Ufn>(X)); }

// TEMPLATE CLASS binary_negate
template<class Bfn>
class binary_negate
:public binary_function<typename Bfn::first_argument_type,
    typename Bfn::second_argument_type, bool> {
public:
    explicit binary_negate(const Bfn& X)
        : Fn(X) {}
    bool operator()(const typename Bfn::first_argument_type&
        X, const typename Bfn::second_argument_type& Y) const
        {return (!Fn(X, Y)); }
protected:
    Bfn Fn;
};

// TEMPLATE FUNCTION not2
template<class Bfn> inline
binary_negate<Bfn> not2(const Bfn& X)
    {return (binary_negate<Bfn>(X)); }

// TEMPLATE CLASS binder1st
template<class Bfn>
class binder1st
:public unary_function<typename Bfn::second_argument_type,
    typename Bfn::result_type> {
public:
    typedef unary_function<typename Bfn::second_argument_type,
        typename Bfn::result_type> Base;
    typedef typename Base::argument_type argument_type;
    typedef typename Base::result_type result_type;
```

```
binder1st(const Bfn& X,
           const typename Bfn::first_argument_type& Y)
           : op(X), value(Y) {}
result_type operator()(const argument_type& X) const
    {return (op(value, X)); }

protected:
    Bfn op;
    typename Bfn::first_argument_type value;
};

// TEMPLATE FUNCTION bind1st
template<class Bfn, class T> inline
binder1st<Bfn> bind1st(const Bfn& X, const T& Y)
    {typename Bfn::first_argument_type Arg(Y);
     return (binder1st<Bfn>(X, Arg)); }

// TEMPLATE CLASS binder2nd
template<class Bfn>
class binder2nd
    : public unary_function<typename Bfn::first_argument_type,
                           typename Bfn::result_type> {
public:
    typedef unary_function<typename Bfn::first_argument_type,
                           typename Bfn::result_type> Base;
    typedef typename Base::argument_type argument_type;
    typedef typename Base::result_type result_type;
    binder2nd(const Bfn& X,
              const typename Bfn::second_argument_type& Y)
              : op(X), value(Y) {}
    result_type operator()(const argument_type& X) const
        {return (op(X, value)); }
protected:
    Bfn op;
    typename Bfn::second_argument_type value;
};

// TEMPLATE FUNCTION bind2nd
template<class Bfn, class T> inline
birder2nd<Bfn> bind2nd(const Bfn& X, const T& Y)
    {typename Bfn::second_argument_type Arg(Y);
     return (binder2nd<Bfn>(X, Arg)); }

// TEMPLATE CLASS pointer_to_unary_function
template<class A, class R>
class pointer_to_unary_function
```

```

        : public unary_function<A, R> {
public:
    explicit pointer_to_unary_function(R (*X)(A))
        : Fn(X) {}
    R operator()(A X) const
        {return (Fn(X)); }
protected:
    R (*Fn)(A);
};

// TEMPLATE CLASS pointer_to_binary_function
template<class A1, class A2, class R>
class pointer_to_binary_function
    : public binary_function<A1, A2, R> {
public:
    explicit pointer_to_binary_function(
        R (*X)(A1, A2))
        : Fn(X) {}
    R operator()(A1 X, A2 Y) const
        {return (Fn(X, Y)); }
protected:
    R (*Fn)(A1, A2);
};

// TEMPLATE FUNCTION ptr_fun
template<class A, class R> inline
pointer_to_unary_function<A, R>
ptr_fun(R (*X)(A))
    {return (pointer_to_unary_function<A, R>(X)); }
template<class A1, class A2, class R> inline
pointer_to_binary_function<A1, A2, R>
ptr_fun(R (*X)(A1, A2))
    {return (pointer_to_binary_function<A1, A2, R>(X)); }

// TEMPLATE CLASS mem_fun_t
template<class R, class T>
class mem_fun_t : public unary_function<T *, R> {
public:
    explicit mem_fun_t(R (T::*Pm)())
        : Ptr(Pm) {}
    R operator()(T *P) const
        {return ((P->*Ptr)()); }
private:
    R (T::*Ptr)();
};

```

```
// TEMPLATE CLASS mem_fun1_t
template<class R, class T, class A>
    class mem_fun1_t : public binary_function<T *, A, R> {
public:
    explicit mem_fun1_t(R (T::*Pm)(A))
        : Ptr(Pm) {}
    R operator()(T *P, A Arg) const
        {return ((P->*Ptr)(Arg)); }
private:
    R (T::*Ptr)(A);
};

// TEMPLATE CLASS const_mem_fun_t
template<class R, class T>
    class const_mem_fun_t
        : public unary_function<const T *, R> {
public:
    explicit const_mem_fun_t(R (T::*Pm)() const)
        : Ptr(Pm) {}
    R operator()(const T *P) const
        {return ((P->*Ptr)()); }
private:
    R (T::*Ptr)() const;
};

// TEMPLATE CLASS const_mem_fun1_t
template<class R, class T, class A>
    class const_mem_fun1_t
        : public binary_function<T *, A, R> {
public:
    explicit const_mem_fun1_t(R (T::*Pm)(A) const)
        : Ptr(Pm) {}
    R operator()(const T *P, A Arg) const
        {return ((P->*Ptr)(Arg)); }
private:
    R (T::*Ptr)(A) const;
};

// TEMPLATE FUNCTION mem_fun
template<class R, class T> inline
mem_fun_t<R, T> mem_fun(R (T::*Pm)())
{return (mem_fun_t<R, T>(Pm)); }
template<class R, class T, class A> inline
mem_fun1_t<R, T, A> mem_fun(R (T::*Pm)(A))
```

```

        {return (mem_fun1_t<R, T, A>(Pm)); }
template<class R, class T> inline
    const_mem_fun_t<R, T>
        mem_fun(R (T::*Pm)() const)
        {return (const_mem_fun_t<R, T>(Pm)); }
template<class R, class T, class A> inline
    const_mem_fun1_t<R, T, A>
        mem_fun(R (T::*Pm)(A) const)
        {return (const_mem_fun1_t<R, T, A>(Pm)); }

        // TEMPLATE CLASS mem_fun_ref_t
template<class R, class T>
    class mem_fun_ref_t : public unary_function<T, R> {
public:
    explicit mem_fun_ref_t(R (T::*Pm)())
        : Ptr(Pm) {}
    R operator()(T& X) const
        {return ((X.*Ptr)()); }
private:
    R (T::*Ptr)();
};

        // TEMPLATE CLASS mem_fun1_ref_t
template<class R, class T, class A>
    class mem_fun1_ref_t : public binary_function<T, A, R> {
public:
    explicit mem_fun1_ref_t(R (T::*Pm)(A))
        : Ptr(Pm) {}
    R operator()(T& X, A Arg) const
        {return ((X.*Ptr)(Arg)); }
private:
    R (T::*Ptr)(A);
};

        // TEMPLATE CLASS const_mem_fun_ref_t
template<class R, class T>
    class const_mem_fun_ref_t
        : public unary_function<T, R> {
public:
    explicit const_mem_fun_ref_t(R (T::*Pm)() const)
        : Ptr(Pm) {}
    R operator()(const T& X) const
        {return ((X.*Ptr)()); }
private:
    R (T::*Ptr)() const;
};

```

```
};

// TEMPLATE CLASS const_mem_fun1_ref_t
template<class R, class T, class A>
class const_mem_fun1_ref_t
    : public binary_function<T, A, R> {
public:
    explicit const_mem_fun1_ref_t(R (T::*Pm) (A) const)
        : Ptr(Pm) {}
    R operator()(const T& X, A Arg) const
        {return ((X.*Ptr)(Arg)); }
private:
    R (T::*Ptr) (A) const;
};

// TEMPLATE FUNCTION mem_fun_ref
template<class R, class T> inline
mem_fun_ref_t<R, T> mem_fun_ref(R (T::*Pm) ())
{return (mem_fun_ref_t<R, T>(Pm)); }
template<class R, class T, class A> inline
mem_fun1_ref_t<R, T, A>
mem_fun_ref(R (T::*Pm) (A))
{return (mem_fun1_ref_t<R, T, A>(Pm)); }
template<class R, class T> inline
const_mem_fun_ref_t<R, T>
mem_fun_ref(R (T::*Pm) () const)
{return (const_mem_fun_ref_t<R, T>(Pm)); }
template<class R, class T, class A> inline
const_mem_fun1_ref_t<R, T, A>
mem_fun_ref(R (T::*Pm) (A) const)
{return (const_mem_fun1_ref_t<R, T, A>(Pm)); }
} /* namespace std */
#endif /* FUNCTIONAL */
```

测试<functional>

tfunction.c

程序清单 8-2 中列出了文件 tfunction.c。与实现本身类似，测试代码简单而重复。通常，它们会检测那些定义中比较明显的部分。如果顺利的话，测试程序将打印出：

```
SUCCESS testing <functional>
```

然后正常退出。

```
程序清单 8-2: // test <functional>
tfuncio.c #include <assert.h>
#include <iostream>
#include <string.h>
#include <algorithm>
#include <functional>
using namespace std;

// TEST SIMPLE FUNCTION OBJECTS
void test_simple(char *first, char *last, char *dest)
    {typedef unary_function<char, int> Uf;
    Uf::argument_type *pa0 = (char *)0;
    Uf::result_type *pr0 = (int *)0;
    typedef binary_function<char, int, float> Bf;
    Bf::first_argument_type *pal = (char *)0;
    Bf::second_argument_type *pa2 = (int *)0;
    Bf::result_type *pr1 = (float *)0;

    char *mid = first + 2;
    strcpy(first, "\4\3\2\1"), strcpy(dest, "abcd");
    transform(first, mid, mid, dest, plus<char>());
    assert(strcmp(dest, "\6\4cd") == 0);
    transform(first, mid, mid, dest, minus<char>());
    assert(strcmp(dest, "\2\2cd") == 0);
    transform(first, mid, mid, dest, multiplies<char>());
    assert(strcmp(dest, "\10\3cd") == 0);
    transform(first, mid, mid, dest, divides<char>());
    assert(strcmp(dest, "\2\3cd") == 0);
    transform(first, mid, first + 1, dest, modulus<char>());
    assert(strcmp(dest, "\1\1cd") == 0);
    transform(first, mid, dest, negate<char>());
    assert(((signed char *)dest)[0] == -4
        && ((signed char *)dest)[1] == -3);
    transform(first, mid, "\4\4", dest, equal_to<char>());
    assert(memcmp(dest, "\1\0cd", 4) == 0);
    transform(first, mid, "\4\4", dest, not_equal_to<char>());
    assert(memcmp(dest, "\0\1cd", 4) == 0);

    transform(first, mid, "\3\3", dest, greater<char>());
    assert(memcmp(dest, "\1\0cd", 4) == 0);
    transform(first, mid, "\4\4", dest, less<char>());
    assert(memcmp(dest, "\0\1cd", 4) == 0);
    transform(first, mid, "\4\4", dest, greater_equal<char>());
    assert(memcmp(dest, "\1\0cd", 4) == 0);
```

```
    transform(first, mid, "\3\3", dest, less_equal<char>());
    assert(memcmp(dest, "\0\1cd", 4) == 0);

    transform(last - 1, last + 1, "\1\0",
              dest, logical_and<char>());
    assert(memcmp(dest, "\1\0cd", 4) == 0);
    transform(last - 1, last + 1, dest, logical_not<char>());
    assert(memcmp(dest, "\0\1cd", 4) == 0);
    transform(last - 1, last + 1, "\0\0",
              dest, logical_or<char>());
    assert(strcmp(dest, "\1\0cd") == 0); }

// TEST COMPOUND FUNCTION OBJECTS
void test_compound(char *first, char *last, char *dest)
{char *mid = first + 2;
strcpy(first, "\4\3\2\1"), strcpy(dest, "abcd");
unary_negate<logical_not<char>> unop(logical_not<char>());
transform(last - 1, last + 1, dest, unop);
transform(last - 1, last + 1, dest + 2,
          not1(logical_not<char>()));
assert(memcmp(dest, "\1\0\1\0", 4) == 0);
binary_negate<less<char>> binop(less<char>());
transform(first, mid, "\5\2", dest, binop);
transform(first, mid, "\5\2", dest + 2, not2(less<char>()));
assert(memcmp(dest, "\0\1\0\1", 4) == 0);

binder1st<plus<char>> add1(plus<char>(), '\1');
transform(first, mid, dest, add1);
transform(mid, last, dest + 2, bind1st(plus<char>(), '\1'));
assert(strcmp(dest, "\5\4\3\2") == 0);
binder2nd<minus<char>> sub1(minus<char>(), '\1');
transform(first, mid, dest, sub1);
transform(mid, last, dest + 2, bind2nd(minus<char>(), '\1'));
assert(memcmp(dest, "\3\2\1\0", 4) == 0); }

// TEST POINTER FUNCTION OBJECTS
char ufn(char ch)
{return (ch + 1); }
char bfn(char ch1, char ch2)
{return (ch1 + ch2); }
struct Myclass {
    char fn0()
        {return ('\7'); }
    char fn1(char ch)
        {return (ch + 1); }
```

```

        } mycl;
    struct Myclass {
        char fn0() const
            {return ('\\7'); }
        char fn1(char ch) const
            {return (ch + 1); }
    } myccl;

    void test_pointer(char *first, char *last, char *dest)
    {char *mid = first + 2;
     strcpy(first, "\\4\\3\\2\\1"), strcpy(dest, "abcd");
     pointer_to_unary_function<char, char> uf(ufn);
     transform(first, mid, dest, uf);
     transform(mid, last, dest + 2, ptr_fun(ufn));
     assert(strcmp(dest, "\\5\\4\\3\\2") == 0);
     pointer_to_binary_function<char, char, char> bf(bfn);
     transform(first, mid, "\\2\\2", dest, bf);
     transform(mid, last, "\\2\\2", dest + 2, ptr_fun(bfn));
     assert(strcmp(dest, "\\6\\5\\4\\3") == 0);

     mem_fun_t<char, Myclass> mf(Myclass::fn0);
     assert(mf(&mycl) == '\\7');
     assert(mem_fun(Myclass::fn0)(&mycl) == '\\7');
     mem_fun1_t<char, Myclass, char> mfl(Myclass::fn1);
     assert(mfl(&mycl, '\\3') == '\\4');
     assert(mem_fun1(Myclass::fn1)(&mycl, '\\3') == '\\4');
     mem_fun_ref_t<char, Myclass> mfr(Myclass::fn0);
     assert(mfr(mycl) == '\\7');
     assert(mem_fun_ref(Myclass::fn0)(mycl) == '\\7');
     mem_fun1_ref_t<char, Myclass, char> cmflr(Myclass::fn1);
     assert(cmflr(&myccl, '\\3') == '\\4');
     assert(mem_fun1_ref(Myclass::fn1)(myccl, '\\3') == '\\4');

     const_mem_fun_t<char, Myclass> cmf(Myclass::fn0);
     assert(cmf(&myccl) == '\\7');
     const_mem_fun1_t<char, Myclass, char> cmf1(Myclass::fn1);
     assert(cmf1(&myccl, '\\3') == '\\4');
     const_mem_fun_ref_t<char, Myclass> cmfr(Myclass::fn0);
     assert(cmfr(myccl) == '\\7');
     const_mem_fun1_ref_t<char, Myclass, char>
         cmflr(Myclass::fn1);
     assert(cmflr(myccl, '\\3') == '\\4'); }

// TEST <functional>
int main()

```

```
char buf[] = "\4\3\2\1";
char dest[] = "abcd";
char *first = buf, *last = buf + 4;
test_simple(first, last, dest);
test_compound(first, last, dest);
test_pointer(first, last, dest);

cout << "SUCCESS testing <functional>" << endl;
return (0); }
```

习题

- 习题8-1 写出这样的模板类，它所产生的函数对象能够实现二元操作符&、!和*，一元操作符-和+的功能。说出你需要最后的那个模板类（它实际上什么也没有做）的理由。
- 习题8-2 写出模板类binderfgh，使得声明符X(f, g, h)将导致X(a)返回f(g(a), h(a))。为什么需要这样一个模板类？
- 习题8-3 请写出模板函数bindfgh，它返回一个模板类binderfgh的对象。为什么需要这样一个模板函数？
- 习题8-4 [较难] 利用上面习题中得到的结果来写出这样一个表达式：它返回一个函数对象X，而对X(a)的调用又将返回用a去调用函数 $c_2 \cdot x^2 + c_1 \cdot x + c_0$ 所得到的值。
- 习题8-5 [特难] 写一个程序，使它可以将仅使用了C++操作符的任意函数x转化为如下代码：它可以产生一个函数对象X，对X(a)的调用将返回原来函数在用a去调用时所得到的值。

第9章 容器

背景知识

容器是管理序列的类。通过它提供的成员函数，我们可以实现一些诸如往序列中插入新元素、删除已有的元素以及查找元素等操作。这些成员函数返回迭代器来指定元素在序列中的位置。这样，我们就可以将前一章中所描述过的算法应用于容器所控制的序列的部分或全部。

时间复杂度

在管理序列时，我们不可避免地要进行一些折衷。例如，如果计划要频繁地在序列上进行插入或删除，那么我们就会期望容器能够在常数时间中完成该操作——即操作完成的时间不会随着序列中元素个数N的改变而改变。然而，这样的容器有可能并不会在常数时间内完成元素查找操作。于是我们就得决定在整个程序的性能决定因素中，哪种操作带来的影响会更大些，然后从不同的容器类型中挑选出一种适合我们需求的。

对于这种特殊的折衷，我们也可以再进行一定的折衷。对于容器来说，有可能在其上的插入、删除以及查找动作都可以在 $\log N$ （即与N的对数成比例）的时间内完成。对数会随着N一同增长，但它增长的速度要远慢于N的增长速度，于是这样的时间复杂度就有可能满足所有的情况了。但这时，我们又不得不为容器的不同尺度而付出代价。

存储开销

为了支持上述的对数行为，容器必须能够把它所控制的序列表达为某种有序树（ordered tree）结构。树为每个元素额外存储三个指针，用它们来指定每个元素的父元素以及两个子元素。这样的话，存储空间分配器（storage allocator）就可能会消耗更多的存储空间。如果元素值本身的存储就需要大量存储空间的话，那这些额外的损耗也就算不了什么。但如果元素值只需要很少的存储空间而程序又分配了大量的元素，这种对树的额外开销就可能证明是有害的。在这种情况下，简单点的容器类型可能会提供更好的折衷：它需要为每个元素配置的指针将会很少，甚至完全没有；但受控序列上进行的操作会出现糟糕的时间复杂度。

STL提供了不同的容器模板类，以供我们选择使用。在所有的情况下，总有一个模板参数代表着我们所希望存储被控序列中的元素的类型。我们可以为满足下面一些要求的任意对象类型的元素特化这些模板类：

- 对象类型T必须有一个可以存取的默认构造函数。

- 它必须有一个可以存取的析构函数。
- 它必须有一个可以存取的其语义有意义的赋值操作符以及函数签名operator=(const T&)。

STL容器模板类通常实现为以下几种：

vector

- vector（向量）——一个有着N个或更多连续存储的元素的数组。

list

- list（列表）——一个由节点组成的双向链表，每个节点中包含着一个元素。

deque

- deque（双队列）——一个有着N个（或更多）连续存储的指向不同元素的指针组成的数组。

set

- set（集合）——一个由节点组成的红 / 黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排序，没有两个不同的元素能够拥有相同的次序。

multiset

- multiset（多重集合）——允许存在两个次序相等的元素的集合。

map

- map（映射）——一个由{键，值}对组成的集合，以某种作用于键对上的谓词排序。

multimap

- multimap（多重映射）——允许键对有相等的次序的映射。

表9-1中总结了在这些模板容器类上操作所花费的时间复杂度。它还给出了每个模板容器类会为其所控制序列中的元素添加多少个附加指针，在此忽略了存储空间分配器的开销。

表9-1：
容器的时间及
空间复杂度

	vector	deque	list	set/map
插入/删除	N	N	常量	$\log N$
从前面添加	(N)	常量	常量	$(\log N)$
find(val)	(N)	(N)	(N)	$\log N$
X(N)	常量	常量	(N)	(N)
指针	0	1	2	3

注释：(N)或($\log N$)代表着不被成员函数直接支持的操作的时间复杂度。

容器适配器

STL还定义了一些容器适配器（container adapter）。它们“几乎是”由其他容器实现的容器。它们会对如何存取元素有意地设置一些限制：

stack

- stack（栈）——一个后进先出（LIFO）的值的队列。

queue

- queue（队列）——一个先进先出（FIFO）的值的队列。

priority_queue

- priority_queue（优先队列）——一个队列，其中元素的次序是由作用于所存储的值对上的某种谓词决定的，拥有高优先级的元素会被先行交付。

我们将会在后续章节讲述这些容器适配器。

功能描述

```

namespace std {
template<class T>
class Cont;

    // TEMPLATE FUNCTIONS
template<class T>
bool operator==(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
bool operator!=(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
bool operator<(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
bool operator>(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
bool operator<=(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
bool operator>=(const Cont<T>& lhs, const Cont<T>& rhs);
template<class T>
void swap(Cont<T>& lhs, Cont<T>& rhs);
}

```

容器是用来管理元素序列的STL模板类。这些元素可以是满足下列条件的任意对象类型：提供一个复制构造函数、一个析构函数以及一个赋值操作符（当然，它们都有着明智的行为）。析构函数是不会向外抛出任何异常的。通过使用一个普通的模板类Cont（它的参数类型为其元素类型T），我们描述了所有这样的容器所需要的属性。一个实际的容器

模板类拥有的模板参数比这个模板类的要多，当然，也拥有更多的成员函数。

STL中有着如下的模板容器类：

```
deque  
list  
map  
multimap  
multiset  
set  
vector
```

basic_string

标准C++库模板类**basic_string**同样也满足那些对于一个模板容器类的要求。

▪ Cont

```
template<class T>  
    class Cont {  
public:  
    typedef T0 size_type;  
    typedef T1 difference_type;  
    typedef T2 reference;  
    typedef T3 const_reference;  
    typedef T4 value_type;  
    typedef T5 iterator;  
    typedef T6 const_iterator;  
    typedef T7 reverse_iterator;  
    typedef T8 const_reverse_iterator;  
    iterator begin();  
    const_iterator begin() const;  
    iterator end();  
    const_iterator end() const;  
    reverse_iterator rbegin();  
    const_reverse_iterator rbegin() const;  
    reverse_iterator rend();  
    const_reverse_iterator rend() const;  
    size_type size() const;  
    size_type max_size() const;  
    bool empty() const;  
    iterator erase(iterator it);  
    iterator erase(iterator first, iterator last);  
    void clear();  
    void swap(Cont& x);  
};
```

异常情况下 的安全性

该模板类描述的对象控制一个元素类型通常为T、长度可变的序列。该序列可以以不同的方式来存储，具体的方式依赖于不同的实际容器。

容器的构造函数或成员函数可能会调用构造函数T(const T&)或函数T::operator=(const T&)。如果在这些调用过程中有异常抛出，我们要求容器对象能够保持其完整性，并且还能够把它所捕获的异常再次向外抛出。这样，即使在容器对象抛出异常后，我们也可以安全地对其进行交换、赋值、删除或者销毁等操作。然而，通常我们都不能以别的方式来预言该容器对象所控制的序列的状态。

另外还有一些附加的警告：

- 如果表达式~T()抛出异常，那么该容器对象的状态为未定义的。
- 如果容器中存储有一个分配器对象al，并且除了作为调用al.allocate的结果之外，al在操作过程中还会抛出其他的异常，那么该容器对象的状态为未定义的。
- 如果容器中存储有一个函数对象comp（我们用它来对序列排序），并且comp向外抛出任意类型的异常，那么该容器对象的状态为未定义的。

STL定义的容器类还满足一些附加的需求，我们将在下面对此一一进行描述。

即使在上面所描述的异常情况下，容器模板类list也可以向外提供确定的、有用的行为。例如，如果在向一个list插入一个或多个元素时有异常抛出，容器将保持插入元素前的状态并且把捕获到的异常继续向外抛出。

对所有STL定义的容器类来说，如果在下列成员函数的调用过程中有着异常抛出的话：

```
insert // single element inserted
push_back
push_front
```

容器将不会发生任何改变，并且所捕获到的异常也将继续向外抛出。

对所有STL定义的容器类来说，在下列成员函数的调用过程中不会有异常抛出：

```
erase // single element erased
pop_back
pop_front
```

此外，在复制由成员函数返回的迭代器时，不会有异常抛出。

对所有STL定义的容器类来说，对于成员函数swap还有一些附加的要求：

- 该成员函数只有在容器存储有一个分配器对象al，并且al在被复制时抛出异常的情况下才会抛出异常。

- 指定受控序列中的元素的引用、指针以及迭代器在交换后仍然保持有效。

分配器

STL定义的容器类对象都是通过一个存储于容器内、类型为A（通常是指模板类的一个模板参数）的对象来为其控制的序列进行分配以及释放存储空间的。这样的存储空间分配器必须与类allocator<T>有一样的外部接口。特别是，A的类型必须和A::rebind<value_type>::other一样。

对所有STL定义的容器类来说，成员函数：

```
A get_allocator() const;
```

将返回被存储的分配器对象的一个拷贝。注意，当对容器对象进行赋值时，容器内的分配器对象并没有被复制。如果构造函数没有包含任何的分配器参数，那么它们就会将存储于该分配器中的值初始化为A()。

依照C++标准，对于STL定义的容器类，我们可以假定：

- 所有类A的对象都相等。
- 类型A::const_pointer等同于const T*。
- 类型A::const_reference等同于const T&。
- 类型A::pointer等同于T*。
- 类型A::reference等同于T&。

然而，在本实现中，容器并没有遵循这样简单化的假设。它们是和更值得炫耀的分配器对象在一起协同工作的：

- 所有类A的对象并不一定要相等。（我们可以维护多个存储池。）
- 类型A::const_pointer并不一定要等同于const T*。（常量指针可以是一个类。）
- 类型A::pointer并不一定要等同于T*。（指针可以是一个类。）

口 Cont::begin

```
const_iterator begin() const;
iterator begin();
```

该成员函数返回一个指向被控序列中第一个元素的迭代器（如果序列是一个空序列的话，则指向紧接着序列末端的下一个位置）。

口 Cont::clear

```
void clear();
```

该成员函数会调用erase(begin(), end())。

口 Cont::const_iterator

```
typedef T6 const_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量迭代器来使用。在此处它被描述为未指定的类型T6的同义词。

■ Cont::const_reference

```
typedef T3 const_reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量引用。

在此处它被描述为未指定的类型T3(通常是A::const_reference)的同义词。

■ Cont::const_reverse_iterator

```
typedef T8 const_reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量反转型迭代器。在此处它被描述为未指定的类型T8(通常是reverse_iterator<const_iterator>)的同义词。

■ Cont::difference_type

```
typedef T1 difference_type;
```

该有符号整数类型描述的对象表示被控序列中任意两个元素地址之间的差距。在此处它被描述为未指定的类型T1(通常是A::difference_type)的同义词。

■ Cont::empty

```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。

■ Cont::end

```
const_iterator end() const;
iterator end();
```

该成员函数返回一个迭代器，指向了紧接着被控序列末端的下一个位置。

■ Cont::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

第一个成员函数将会从被控序列中删除掉由it所指定的元素。第二个成员函数删除被控序列中由(first, last)所指明的区间内的所有元素。这两个函数返回的都是一个迭代器，指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

该成员函数不会抛出任何异常。

■ Cont::iterator

```
typedef T5 iterator;
```

该类型描述的对象可以作为一个指向被控序列的迭代器来使用。在此处它被描述为未指定的类型T5的同义词。类型为iterator的对象可以转换为类型为const_iterator的对象。

■ Cont::max_size

```
size_type max_size() const;
```

该成员函数返回的是容器对象所能控制的最长序列的长度，不管被控序列的长度是多少，调用它的时间总是一个常数时间。

口 Cont::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

该成员函数返回的是一个反转型迭代器，它指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 Cont::reference

```
typedef T2 reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。在此处它被描述为未指定的类型T2（通常是A::reference）的同义词。类型为reference的对象可以转换为类型为const_reference的对象。

口 Cont::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

该成员函数返回的是一个反转型迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 Cont::reverse_iterator

```
typedef T7 reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型迭代器来使用。在此处它被描述为未指定的类型T7（通常是reverse_iterator<iterator>）的同义词。

口 Cont::size

```
size_type size() const;
```

该成员函数返回被控序列的长度，不管被控序列的长度是多少，调用它的时间总是一个常数时间。

口 Cont::size_type

```
typedef T0 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为未指定的类型T0（通常是A::size_type）的同义词。

口 Ccont::swap

```
void swap(Cont& x);
```

该成员函数在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator()，它将在常数时间内完成此次交换动作。否则，它将以与这两个被控序列的元素个数成比例的次数调用元素的构造函数以及为元素赋值。

Cont::value_type

```
typedef T4 value_type;
```

该类型等同于模板参数T。在此处它被描述为未指定的类型T4（通常是A::value_type）的同义词。

operator!=

```
template<class T>
bool operator!=(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数返回!(lhs == rhs)。

operator==

```
template<class T>
bool operator==(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数重载operator==来比较两个模板类Cont的对象。该函数返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())。

operator<

```
template<class T>
bool operator<(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数重载operator<来比较两个模板类Cont的对象的大小。该函数返回 lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

operator<=

```
template<class T>
bool operator<=(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数返回!(rhs < lhs)。

operator>

```
template<class T>
bool operator>(
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数返回rhs < lhs。

operator>=

```
template<class T>
```

```
bool operator>=
    const Cont <T>& lhs,
    const Cont <T>& rhs);
```

该模板函数返回!(lhs < rhs)。

口 swap

```
template<class T>
void swap(
    Cont <T>& lhs,
    Cont <T>& rhs);
```

该模板函数执行lhs.swap(rhs)。

使用容器

为了使用STL容器或者容器适配器，请把定义该容器（或容器适配器）模板类的头文件包含到程序中：

- | | |
|--|--|
| <deque>
<list>
<map>
<set>
<queue>
<stack>
<vector> | <ul style="list-style-type: none"> • deque 定义于头文件<deque>中。 • list 定义于头文件<list>中。 • map 和 multimap 定义于头文件<map>中。 • set 和 multiset 定义于头文件<set>中。 • priority_queue 和 queue 定义于头文件<queue>中。 • stack 定义于头文件<stack>中。 • vector 定义于头文件<vector>中。 |
|--|--|

我们将在后续章节中详细地描述上述的各个头文件。

共有的属性

正如你所预期的那样，每个STL容器都有一些独一无二的属性。如何构造一个容器对象，如何向它里面插入元素，以及如何在容器里面定位这些元素，这些都会随着容器种类的不同而改变。但是，所有的容器模板类之间还是存在一些共同的属性（我们将会在此描述它们）。例如，每个容器模板类都定义了一些成员类型，用于提供有关容器的一些有用信息：

- | | |
|---|---|
| value_type
size_type
difference_type
allocator_type
iterator | <ul style="list-style-type: none"> • value_type 是被控序列中元素的类型。 • size_type 是可以表示任意被控序列长度的类型。 • difference_type 是可以表示类型为 iterator 的对象之间差距的代数类型。 • allocator_type 是分配器对象的类型，我们使用分配器对象来为被控序列提供所有有关的存储管理操作。 • iterator 可以是任意由容器的非常量成员函数返回的迭代器类 |
|---|---|

	型，我们可以使用它来存取被控序列。
const_iterator	<ul style="list-style-type: none"> • <code>const_iterator</code> 可以是任意由容器的常量成员函数返回的迭代器类型，我们可以使用它来存取被控序列。
reverse_iterator	<ul style="list-style-type: none"> • <code>reverse_iterator</code> 可以是任意由容器的非常量成员函数返回的反转型迭代器类型，我们可以使用它来存取被控序列。
const_reverse_iterator	<ul style="list-style-type: none"> • <code>const_reverse_iterator</code> 可以是任意由容器的常量成员函数返回的反转型迭代器类型，我们可以使用它来存取被控序列。
reference	<ul style="list-style-type: none"> • <code>reference</code> 可以是任意由容器的非常量成员函数返回的引用的类型，我们可以使用它来存取被控序列。
const_reference	<ul style="list-style-type: none"> • <code>const_reference</code> 可以是任意由容器的常量成员函数返回的引用的类型，我们可以使用它来存取被控序列。
	有些成员函数会返回迭代器。例如，成员函数 <code>erase(iterator)</code> 就返回一个指向（剩下的）紧接着被删除元素的下一个元素的迭代器。如果希望直接存取整个被控序列，通常需要调用：
begin	<ul style="list-style-type: none"> • 通过调用 <code>begin</code> 来获得指向被控序列开始处的迭代器。
end	<ul style="list-style-type: none"> • 通过调用 <code>end</code> 来获得指向被控序列末端的迭代器。
rbegin	<ul style="list-style-type: none"> • 通过调用 <code>rbegin</code> 来获得指向被控序列末端的反转型迭代器。
rend	<ul style="list-style-type: none"> • 通过调用 <code>rend</code> 来获得指向被控序列开始处的反转型迭代器。
erase	<ul style="list-style-type: none"> • 给定一个迭代器，或者是一个迭代器的区间，我们就可以通过调用 <code>erase</code> 来从被控序列中删除一个或多个元素。或者，我们可以通过调用 <code>clear</code> 来删除序列中所有的元素。
clear	
size	<ul style="list-style-type: none"> • 我们可以通过调用成员函数 <code>size</code> 来检测被控序列中元素的个数。
empty	<p>如果我们仅需要知道当前序列中是否存在元素，调用 <code>empty</code> 就可以达到这个目的。成员函数 <code>max_size</code> 可以提供容器控制的序列最长可以是多少的信息。然而，请注意：可用内存的大小比该函数给出的序列大小更具有约束力。</p>
max_size	
get_allocator	<ul style="list-style-type: none"> • 一般来说，我们很少有机会直接操作存储在容器对象中的分配器对象。但是如果需要这么做，我们可以通过调用成员函数 <code>get_allocator</code> 来获得一份该分配器对象的拷贝。
swap	<ul style="list-style-type: none"> • 最后，每个容器模板类都提供了各自版本的模板函数 <code>swap</code>。（参见第 6 章。）如果两个要交换的容器对象所存储的分配器对象相等的话，该函数将仅在这两个容器对象间交换那些被控序列的信息。与那种“野蛮地”逐个复制的方法相比，这种方法可以极大地缩短交换所需的时间。

习题

习题9-1

将下面的表填满。对于一个包含有N个元素的序列来说，左栏中给出的是在序列中搜索一个元素所需时间的公式（单位是微秒）。顶端的其他栏表示的是N的值。你可以使用任何能够被人理解的方式来表达这些值（如：微秒、秒、年或是其他计量单位）：

Time Complexity	$N = 10^1$	$N = 10^5$	$N = 10^9$
$N^2 \mu\text{sec}$			
$10 * N$			
$5 * \log N$			
300			

习题9-2

对于STL中定义的每个容器模板类，请描述一种特殊的情况，使你能很自然地使用它来模拟已有的数据结构。（其中需要在该数据结构上进行的操作通常对于你所选择的容器模板类也很高效。）

习题9-3

在STL定义的容器模板类中，哪一个适合作为栈的基础实现？

习题9-4

[较难] 如何实现一个容器模板类，使得不管被控序列的长度如何，它都可以在常数时间内查找匹配某些谓词的元素？为了达到这个目的，你又将做出什么样的折衷？

习题9-5

[特难] 如何实现一个模板容器类，使得它不但可以在常数时间内查找匹配某些谓词的元素，还可以在常数时间内向查找到的元素旁边插入新元素？

第 10 章 <vector>

背景知识

<vector>
vector

头文件<vector>中仅定义了模板类vector，它是一个容器，它所控制的序列是以连续数组的方式存储的。也就是说，这种特殊的容器并没有为每个元素存储一个附加的指针。我们可以通过使用 $v[i]$ 来在常数时间内存取容器v中的第*i*个元素。另一方面，该容器并没有在被控的序列上施加任何次序。我们只能通过从头到尾地扫描一遍被控序列来查找一个给定值的元素。这样的操作所需的时间将会是一个线性时间——与序列中元素的个数N成比例。向前添加以及插入通常都会导致容器把后面所有的元素复制一遍，甚至分配一个新的数组，然后将所有已有的元素都复制到新的数组中去。这样的重新分配可以在线性的时间内完成，在本章的后面部分，我们还将描述一种小技巧来优化它。

然而，模板类vector还提供了一种特殊的优化措施。它允许我们保留大于当前序列长度的数组所需的存储空间。一旦给定了保留的存储空间，容器就得以避免在每次插入新元素时都需要重新分配存储空间，它只需简单地重新对已有的数组进行排列。同样地，容器也可以通过重新排列来删除元素，这样保留空间也就变大了。所有这些操作的时间同样和N成比例，但它们明显比上面的要快多了。

更重要的是，保留空间可以使得在容器末端添加元素，或者删除最后一个元素的时间为一个常数。这意味着模板类vector可以作为一种可以接受的实现栈的方式。猜测被保留的空间需要多大也就成了我们新的任务。如果我们的猜测错了，那么扩展被控序列所需的花费将显著增加。不过，向量有时还是可以作为合理的栈来使用。

basic_
string

模板类vector保证了进行比较的可能性。标准C++库中同样也定义了模板类basic_string，并把它作为从过去的实践中得到的众多string类的一个一般化版本。模板类basic_string并不是最初由惠普定义的STL版本的一部分。随着标准化进程的开展，它慢慢地与其他STL容器所要求的属性一同产生了。（我们并不准备在本书中描述basic_string。）

basic_string的一个模板参数是它的元素类型——它并不一定要是类型char。basic_string对象所控制的序列一般都要求以数组的方式存储起来。于是问题就产生了，什么时候我们该用basic_string？什么时候又该用vector呢？下面就是一些可供大家参考的准则：

PODS

- `basic_string` 的元素不可以拥有非平凡（*nontrivial*）构造函数和析构函数。实际上，它必须是我们可以在 C 程序中声明的类型。可以通过逐位复制（*bitwise copy*）的方式来为这样类型的对象赋值。（这种类型的正式名字叫做 **POD**，意味着“plain old data structure”。）相反，`vector` 元素可以是有着某些构造函数、一个相当常规的赋值操作符以及一个析构函数的任意类型。

char_traits

- 模板类 `basic_string` 需要一个“`traits`”类来作为它的另一个模板参数。`traits` 指定了如何移动或是比较序列中的元素，并且还指定了如何读写由元素构成的文件。标准 C++ 库中提供了一个模板类 `char_traits`，它是 `traits` 对于元素类型 `char` 和 `wchar_t` 的一个特化版本。如果它们都不符合需要，可以（也必须）定义一个自己的 `traits` 类。而 `vector` 元素则不使用这样的 `traits`。

以 null 结尾

- `basic_string` 对象可以移交以 `null` 结尾的序列。而 `vector` 对象只处理我们存储在它里面的序列。

写时复制

- `basic_string` 对象可以使用写时复制（*copy-on-write*）语义，这样就可以在一些特定的使用模式下极大地提升效率。通常，模板类 `vector` 不能实现这样的优化。

模板类 `basic_string` 还定义了许多成员函数，并用它们来在被控序列上做一些与字符串相关的事情。当然，我们也可以不使用这些附加的成员函数，而只是做那些与向量相关的事情。通常，上面所列出的考量已经限定了在给定的应用程序中应该选择使用这两个容器中的哪一个。

**string
wstring**

概括说来，当我们需要快速地随机存取被控序列中的元素时，我们就会使用模板类 `vector`。除非我们可以预见到它以后的增长情况并为之保留足够多的存储空间，通常 `vector` 对象增长的花费都会很大。对于类型为 `char` 或者 `wchar_t` 的元素来说，使用模板类 `basic_string` 可能会更好一些。为了方便起见，标准 C++ 库中还定义了 `string`（是 `basic_string<char, char_traits<char>>` 的同义词），以及 `wstring`（是 `basic_string<wchar_t, char_traits<wchar_t>>` 的同义词）。

**vector<bool>
flip
swap**

头文件 `<vector>` 中还提供了模板类 `vector` 的一个部分特化版本，并以之来处理元素类型为 `bool` 的序列。它这样做的目的部分是为了更加经济地使用内存。该特化版本将 8 个（或更多）元素同时存储在一个连续空间中的字节内，而不是为每个元素准备一个（或更多）字节。它还定义了成员函数 `flip`（用来颠倒一个元素）和 `swap`（用来交换两个元素）。也就是说，我们可以有效地利用存储空间声明并处理一个大的布尔向量。

功能描述

```
namespace std {
    template<class T, class A>
        class vector;
```

```

template<class A>
class vector<bool>

    // TEMPLATE FUNCTIONS
template<class T, class A>
bool operator==(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
bool operator!=(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
bool operator<(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
bool operator>(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
bool operator<=(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
bool operator>=(const vector<T, A>& lhs, const vector<T, A>& rhs);
template<class T, class A>
void swap(vector<T, A>& lhs, vector<T, A>& rhs);
};

```

包含STL标准头文件<vector>, 可以得到模板类vector以及几个支持模板的定义。

■ operator!=

```

template<class T, class A>
bool operator!=(const vector<T, A>& lhs, const vector<T, A>& rhs);

```

该模板函数返回!(lhs == rhs)。

■ operator==

```

template<class T, class A>
bool operator==(const vector<T, A>& lhs, const vector<T, A>& rhs);

```

```
    const vector <T, A>& lhs,
    const vector <T, A>& rhs);
```

该模板函数重载operator==来比较两个模板类vector的对象。该函数返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())。

口 operator<

```
template<class T, class A>
bool operator<(
    const vector <T, A>& lhs,
    const vector <T, A>& rhs);
```

该模板函数重载operator<来比较两个模板类vector的对象。该函数返回lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

口 operator<=

```
template<class T, class A>
bool operator<=(
    const vector <T, A>& lhs,
    const vector <T, A>& rhs);
```

该模板函数返回!(rhs < lhs)。

口 operator>

```
template<class T, class A>
bool operator>(
    const vector <T, A>& lhs,
    const vector <T, A>& rhs);
```

该模板函数返回rhs < lhs。

口 operator>=

```
template<class T, class A>
bool operator>=(
    const vector <T, A>& lhs,
    const vector <T, A>& rhs);
```

该模板函数返回!(lhs < rhs)。

口 swap

```
template<class T, class A>
void swap(
    vector <T, A>& lhs,
    vector <T, A>& rhs);
```

该模板函数执行lhs.swap(rhs)。

口 vector

```
template<class T, class A = allocator<T> >
class vector {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
```

```
typedef typename A::const_pointer
    const_pointer;
typedef typename A::reference reference;
typedef typename A::const_reference
    const_reference;
typedef typename A::value_type value_type;
typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator>
    reverse_iterator;
vector();
explicit vector(const A& a);
explicit vector(size_type n);
vector(size_type n, const T& x);
vector(size_type n, const T& x,
       const A& a);
vector(const vector& x);
template<class InIt>
vector(InIt first, InIt last);
template<class InIt>
vector(InIt first, InIt last,
       const A& a);
void reserve(size_type n);
size_type capacity() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
void resize(size_type n);
void resize(size_type n, T x);
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
```

```

        const_reference operator[](size_type pos);
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;
        void push_back(const T& x);
        void pop_back();
        template<class InIt>
            void assign(InIt first, InIt last);
        void assign(size_type n, const T& x);
        iterator insert(iterator it, const T& x);
        void insert(iterator it, size_type n, const T& x);
        template<class InIt>
            void insert(iterator it, InIt first, InIt last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        void clear();
        void swap(vector& x);
    };

```

该模板类描述的对象控制一个元素类型为T的可变长度序列。该序列是以类型为T的数组的方式存储的。

vector对象通过存储的一个类A的分配器对象来进行内存的分配及释放。该分配器对象必须拥有和模板类allocator的对象一样的外部接口。注意，当对容器赋值时，已存储的分配器对象不会被复制。

只有在调用成员函数会导致被控序列的长度超过vector所能承担的容量时，vector对象才会重新分配空间。其他的插入及删除操作可能会改变该序列中许多元素的存储地址。在所有的这些情况下，指向被控序列中被改变位置的迭代器和引用在操作完成后都将变为无效。

vector::allocator_type

```
typedef A allocator_type;
```

该类型是模板参数A的同义词。

vector::assign

```

template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);

```

如果Init是一个整数类型，第一个成员函数就相当于assign((size_type)first, (T)last)。否则，第一个成员函数会把*this所控制的序列替换成序列[first, last]，其中[first, last]不能和*this最初所控制的序列重叠。第二个成员函数将*this所控制的序列替换成一个由n个值为x的元素组成的序列。

口 `vector::at`

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

该成员函数返回一个引用，它指向被控序列中位置为pos的元素。如果给定的位置是无效的，该函数就将抛出一个`out_of_range`异常。

口 `vector::back`

```
reference back();
const_reference back() const;
```

该成员函数返回一个引用，它指向被控制序列中最后一个元素（注意：被控序列不能为空）。

口 `vector::begin`

```
const_iterator begin() const;
iterator begin();
```

该成员函数返回一个随机存取迭代器，它指向被控序列的第一个元素（如果被控序列为`空`，则指向紧接着该序列末端的下一个位置）。

口 `vector::capacity`

```
size_type capacity() const;
```

该成员函数返回当前分配给被控序列的存储空间的大小，它至少为`size()`。

口 `vector::clear`

```
void clear();
```

该成员函数调用`erase(begin(), end())`。

口 `vector::const_iterator`

```
typedef T1 const_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量随机存取迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。

口 `vector::const_pointer`

```
typedef typename A::const_pointer
const_pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。

口 `vector::const_reference`

```
typedef typename A::const_reference
const_reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。

口 `vector::const_reverse_iterator`

```
typedef reverse_iterator<const_iterator>
```

```
const_reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量反转型迭代器来使用。

口 `vector::difference_type`

```
typedef T3 difference_type;
```

该有符号整数类型描述的对象表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。

口 `vector::empty`

```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。

口 `vector::end`

```
const_iterator end() const;  
iterator end();
```

该成员函数返回一个随机存取迭代器，它指向紧接着被控序列末端的下一个位置。

口 `vector::erase`

```
iterator erase(iterator it);  
iterator erase(iterator first, iterator last);
```

第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除被控序列中由(first, last)所指明的区间内的所有元素。这两个函数返回的都是一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

删除N个元素将导致调用N次析构函数以及从删除点至序列末端的每个元素都被赋值一次。由于不需要进行存储空间的重新分配，所以只有从指向第一个被删除元素到序列末端这个区域的迭代器以及引用才会变成无效的。

这些成员函数不会抛出任何异常。

口 `vector::front`

```
reference front();  
const_reference front() const;
```

该成员函数返回指向被控序列中第一个元素的引用（被控序列不能为空）。

口 `vector::get_allocator`

```
A get_allocator() const;
```

该成员函数返回存储在vector中的分配器对象。

口 `vector::insert`

```
iterator insert(iterator it, const T& x);  
void insert(iterator it, size_type n, const T& x);
```

```
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

所有的这些成员函数都是在被控序列中由it所指定的元素前插入由其他操作数指定的序列。第一个成员函数插入一个单独的值为x的元素，然后返回一个指向刚被插入的这个元素的迭代器。第二个成员函数则插入连续的n个值为x的元素。

如果InIt是一个整数类型的话，那么最后一个成员函数就相当于insert(it, (size_type)first, (T)last)。否则，最后的那个成员函数将向被控序列中插入序列[first, last)（它不能和最初的被控序列有重叠）。

当插入一个单独的元素时，被复制元素的个数与插入点到序列末端之间元素的个数成线性比例。当向序列的末端插入一个单独的元素时，那么被分摊(amortized)复制的元素个数为一个常数。当插入N个元素时，被复制元素的个数与N加上插入点到序列末端之间元素的个数成线性比例——除非此模板成员函数是对InIt为输入迭代器的一个特化版本，此时它相当于进行N次单独的插入。

如果出现了存储空间的重新分配，那么vector对象的容量也就会因为一些确定的因素（至少）变大，此时所有迭代器和引用都将变为无效。如果没有出现重新分配的情况，则只有指向从插入点到序列末端这段区间内的迭代器和引用才会变为无效。

如果在插入单个元素时有异常抛出，那么容器将保持不变（即没有任何元素被插入）并且继续将异常向外抛出。如果在插入多个元素时有异常抛出，并且该异常并不是在复制某个元素时抛出的，那么容器将保持不变并且继续将异常向外抛出。

口 vector::iterator
 typedef T0 iterator;

该类型描述的对象可以作为一个指向被控序列的随机存取迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

口 vector::max_size
 size_type max_size() const;

该成员函数返回容器对象所能控制的最长序列的长度。

口 vector::operator[]
 const_reference operator[](size_type pos) const;
 reference operator[](size_type pos);

该成员函数返回一个指向被控序列中位置为pos的元素的引用。如果给定的位置是一个无效的位置，那么该函数的行为将会是未定义的。

口 vector::pointer
 typedef typename A::pointer pointer;

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 `vector::pop_back`

```
void pop_back();
```

该成员函数会将被控序列中的最后一个元素删除，在执行`pop_back`前序列不能为空。

该成员函数不会抛出任何异常。

口 `vector::push_back`

```
void push_back(const T& x);
```

该成员函数会把一个值为x的元素插入到被控序列的末端。

如果在调用过程中有异常抛出，那么容器将保持不变并且继续将异常向外抛出。

口 `vector::rbegin`

```
const_reverse_iterator rbegin() const;  
reverse_iterator rbegin();
```

该成员函数返回一个反转型迭代器，它指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 `vector::reference`

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 `vector::rend`

```
const_reverse_iterator rend() const;  
reverse_iterator rend();
```

该成员函数返回一个反转型迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 `vector::reserve`

```
void reserve(size_type n);
```

如果n大于`max_size()`，该成员函数就会向外抛出一个`length_error`异常，以此来报告长度错误。否则，它将确保自此以后调用`capacity()`得到的返回值不会小于n。

口 `vector::resize`

```
void resize(size_type n);  
void resize(size_type n, T x);
```

这两个成员函数都可以确保在被调用之后再调用`size()`，得到的返回值是n。如果在调用过程中必须扩展被控序列的长度，那么第一个成员函

数会向被控序列末端添加值为T()的元素，而第二个元素添加值为x的元素。如果想缩短被控序列的长度，这两个成员函数都会调用erase(begin() + n, end())。

■ `vector::reverse_iterator`
`typedef reverse_iterator<iterator>`
`reverse_iterator;`

该类型描述的对象可以作为一个指向被控序列的反转型迭代器来使用。

■ `vector::size`
`size_type size() const;`

该成员函数返回被控序列的长度。

■ `vector::size_type`
`typedef T2 size_type;`

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

■ `vector::swap`
`void swap(vector& x);`

该成员函数会在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator()，它将在常数时间内完成交换并且不会抛出任何异常，另外，它还不会导致任何指向这两个被控序列中元素的引用、指针和迭代器无效。否则，它将以与这两个被控序列的元素个数成比例的次数调用元素的构造函数以及为元素赋值。

■ `vector::value_type`
`typedef typename A::value_type value_type;`

该类型是模板参数T的同义词。

■ `vector::vector`
`vector();`
`explicit vector(const A& al);`
`explicit vector(size_type n);`
`vector(size_type n, const T& x);`
`vector(size_type n, const T& x, const A& al);`
`vector(const vector& x);`
`template<class Init>`
`vector(Init first, Init last);`
`template<class Init>`
`vector(Init first, Init last, const A& al);`

所有的构造函数都会存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，它就是A()。

开始的两个构造函数指定一个空的被控序列。第三个构造函数则指定一个由n个值为T()的元素组成的序列。第四个和第五个构造函数指定由n个值为x的元素组成的序列。第六个构造函数指定一个由x控制的序列的拷贝。如果Int是一个整数类型，最后两个构造函数将指定由(size_type)first个值为(T)last的元素组成的序列；否则，这两个构造函数指定序列[first, last]。

```
#include <vector>
template<class A>
class vector<bool, A> {
public:
    class reference;
    typedef bool const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T4 pointer;
    typedef T5 const_pointer;
    void flip();
    static void swap(reference x, reference y);

    // rest same as template class vector
};
```

这个类是模板类vector对于类型为bool的元素的一个特化版本。它改变了原来在vector中的四个成员类型的定义（以优化元素的压缩和解压缩），并新增了两个成员函数。如果不是这样它的行为会和模板类vector相同。

#include <vector>
typedef T1 const_iterator;

该类型描述的对象可以作为一个指向被控序列的常量随机存取迭代器来使用。在此处它被描述为未指定的类型T1的同义词。

#include <vector>
typedef T5 const_pointer;

该类型描述的对象可以作为一个指向被控序列中常量元素的指针来使用。在此处它被描述为未指定的类型T5的同义词。

#include <vector>
typedef bool const_reference;

该类型描述的对象可以作为一个指向被控序列中元素（在此处其类型为bool）的常量引用来使用。

#include <vector>
void flip();

该成员函数对被控序列中所有成员的值取反。

■ `vector<bool, A>::iterator`
`typedef T0 iterator;`

该类型描述的对象可以作为一个指向被控序列的随机存储迭代器来使用。在此处它被描述为未指定的类型T0的同义词。

■ `vector<bool, A>::pointer`
`typedef T4 pointer;`

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。在此处它被描述为未指定的类型T4的同义词。

■ `vector<bool, A>::reference`
`class reference {`
`public:`
 `reference& operator=(const reference& x);`
 `reference& operator=(bool x);`
 `void flip();`
 `bool operator~() const;`
 `operator bool() const;`
`};`

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。值得注意的是，对于两个类reference的对象x和y来说：

- `bool(x)`得到的是由x指定的元素的值。
- `~x`得到的是把由x指定的元素的值取反所得到的值。
- `x.flip()`将存储于x的值取反。
- `y = bool(x)`以及`y = x`都会把由x指定的元素的值赋值给由y指定的元素。

对于类`vector<bool>`中的成员函数如何构造出reference的对象（它指向被控序列中的元素），标准中没有明确指出。类reference的默认构造函数产生的对象并不会指向这样的元素。

■ `vector<bool, A>::swap`
`void swap(reference x, reference y);`

该静态成员函数交换由x和y所指定的两个被控序列中的元素。

使用<vector>

`vector` 如果想在程序中使用模板类`vector`，请把头文件`<vector>`包含到程序中。我们也可以指定`vector`来存储类型为T的元素，只需这样写一条类型定义语句：

```
typedef vector<T, allocator<T>> Mycont;
```

通过使用默认的模板参数，我们可以省略第二个模板参数。

模板类vector支持所有我们在第9章给出的、对于容器来说是常见的一些操作。（具体可以参见第9章的“使用容器”一节的讨论。）我们在此仅概括模板类vector所特有的属性。

构造函数

为了构造一个类vector<T, A>的对象，可以这么写：

- vector(), 用以声明一个空的vector；
- vector(al)，也是声明一个空的vector，但它还同时存储一个分配器对象al；
- vector(n)，用以声明一个有着n个元素的vector，每个元素都是由其默认构造函数 T() 所构造出来的；
- vector(n, val)，用以声明一个有着n个元素的vector，每个元素都是由其复制构造函数 T(val) 得来的；
- vector(n, val, al)，声明一个和上面一样的vector，但它还存储分配器对象al；
- vector(first, last)，用以声明一个vector，其元素的初始值是从区间[first, last) 所指定的序列中的元素复制而来的；
- vector(first, last, al)，用以声明一个和上面一样的vector，但它还存储分配器对象al。

如果我们已经为类型为allocator<T>的分配器特化了模板类（这也是我们所常做的并且也是STL提供默认值），那么再在构造函数中显式地给出分配器参数al并不能给我们带来任何好处。仅对程序中显式定义的某些分配器，这样的参数才会起作用。（具体可以参见第4章中对于分配器的讨论。）

下面所有的描述中都假定cont是一个类vector<T, A>的对象。

类vector<T, A>的对象可以保留一个大小大于实际所需大小的数组空间。多余的存储空间在成为有效数组的一部分前保持为未构造状态：

reserve

- 为了确保 cont 所分配的存储空间最少可以容纳 n 个元素，请调用 cont.reserve(n)。

capacity

- 为了在不重新分配内存的前提下得到当前 cont 所能容纳的最大数组的大小，请调用 cont.capacity()。

正如在本章前面所提到的，如果想把 vector<T, A> 对象作为一个高效的栈使用的话，让容器保持一定的多余容量是很重要的一件事情。

resize

为了将被控序列的长度改为只容纳 n 个元素，请调用 cont.resize(n)。超出的元素将会被删除。如果序列需要扩展，那么值为 T() 的元素将会插入到序列末端，直到填满整个序列。我们也可以调用 cont.resize(n, val) 来使得插入元素的值为 val。

clear

为了删除容器中所有的元素，请调用 cont.clear()。然而请注意：clear 和 resize 都不能保证一定会减少容器中所保留的存储空间的大小。（即

在调用完 `cont.clear()` 或 `cont.resize(n)` 后，再调用 `cont.capacity()` 所得到的值并不一定小于没有调用 `clear` 或 `reszie` 时所得到的返回值，虽然有的时候它确实会小于这个值。) 在调用 `cont.clear()` 或把它赋值为一个大小为 0 的 `vector` 后，本实现确实释放了所有保留的存储空间，但这种行为并不是 C++ 标准所要求的。有一种特殊的能确保释放掉所有保留存储空间的办法为：

```
cont.swap(vector<T, A>());
```

该语句先是构造一个临时的空向量，然后再把它和 `cont` 相交换。该语句结束时，这个临时的对象将会被销毁，先前由 `cont` 所分配的存储空间也会随之一同释放。

`front`
`back`

为了存取被控序列的第一个元素，请调用 `cont.front()`。为了存取最后一个元素，请调用 `cont.back()`。如果 `cont` 不是一个常量对象，调用这两个成员函数得到的表达式将会是一个左值 (lvalue)，于是就可以改变原来存储于该元素内的值，我们只需这么写：`cont.front() = T()`。然而，如果序列为空，上述表达式的行为将是未定义的。

`operator[]`
`at`

为了存取元素 `i` (以 0 为基值)，可以这么写：`cont[i]`。如果 `cont` 不是一个常量对象，该表达式产生的将会是一个左值。如果 `i` 不在半开区间 `[0, cont.size())` 中，那么 `cont[i]` 将会有未定义的行为。我们也可以用 `cont.at(i)` 来替换 `cont[i]`。在这种调用方式下，如果 `i` 不是一个有效的元素编号，成员函数 `at` 将会向外抛出一个类 `out_of_range` 的异常。

`push_back`
`pop_back`

为了向对象末端插入值为 `x` 的元素，调用 `cont.push_back(x)`。为了删除最后的那个元素，调用 `cont.pop_back()`，此时被控序列不能为空，否则行为将为未定义的。模板类 `vector` 没有定义成员函数 `push_front` 和 `pop_front`。在给定实现一个向量的限制条件后，它们所需的操作时间将和被控序列中元素的个数成比例。

`assign`

为了将被控序列替换成由 `[first, last)` 所指定的序列，调用 `cont.assign(first, last)`。由 `[first, last)` 指定的序列不能为最初被控序列的一部分。为了将被控序列替换成 `n` 个值为 `x` 的元素，调用 `cont.assign(n, x)`。

`insert`

为了在由迭代器 `it` 指定的元素前插入一个值为 `x` 的元素，调用 `cont.insert(it, x)`。它的返回值是一个迭代器，指向刚插入的那个元素。为了在由迭代器 `it` 所指定的元素前插入一个由 `[first, last)` 指定的序列，调用 `cont.insert(it, first, last)`。由 `[first, last)` 指定的序列不能为最初被控序列中的一部分。为了插入 `n` 个值为 `x` 的元素，调用 `cont.insert(it, n, x)`。

`erase`

为了删除由迭代器 `it` 所指定的元素，调用 `cont.erase(it)`。它的返回值为一个迭代器，指向被删除元素的下一个元素。为了删除出区间 `[first, last)` 所指定的所有元素，调用 `cont.erase(first, last)`。

`vector<bool>`

模板类 `vector<bool, A>` 是模板类 `vector` 的一个部分特化版本。它确

保只要我们为 vector 所写的特化版本的第一个模板参数为 bool，就会使用一些不同于 vector 中的定义。我们也可以直接写 vector<bool>，此时它使用默认的分配器。但是这个特化版本也可以让我们指定一个显式的分配器参数 A，只要我们有理由这么做就可以了。

Bvector

不支持部分特化的实现在面对 vector<bool>时，不管是符号表达上还是其他方面都存在着不少困难。比较明智的做法是避免在程序中显式地写出这样形式的代码——有的实现可能会为 vector<bool> 提供另外一个名字。例如，本实现就提供了一个别名 Bvector，使得它可以在老式的、不支持部分特化的编译器中使用。一个比较健壮的做法就是引入一个单独的类型定义，如：

```
typedef Bvector vector_bool;
```

然后在程序中用 vector_bool 来替代 vector<bool> 或任意特定于实现的别名。

vector<bool> 的行为和其他 vector<T, A> 的特化版本差不多。它并没有完全满足 STL 容器的所有要求，但对于大部分的需求来说，它已经足够了。它还提供了一些附加特性：

flip

- 为了把元素 i 的值取反，可以这么写：cont[i].flip() 或 cont.at(i).flip()。

swap

- 为了把被控序列中所有元素的值取反，这么写：cont.flip()。

- 为了交换元素 i 和 j 的值，可以这么写：cont.swap(cont[i], cont[j])。

vector<bool> 还暗示着它所控制的序列有足够的有效的存储方式。常见的实现为每个元素准备 1 位存储空间，而不是一个单独的字节（甚至是为整型所准备的 32 位）。

实现<vector>

vector

程序清单 10-1 中列出了文件 vector。它定义了模板类 vector，以及一些以向量作为操作数的模板函数。它还为类型为 bool 的元素定义了 vector 的部分特化版本。

程序清单 10-1:

```
// vector standard header
#ifndef VECTOR_
#define VECTOR_
#include <memory>
#include <stdexcept>
namespace std {

    // TEMPLATE CLASS Vector_val
template<class T, class A>
```

```
class Vector_val {
protected:
    Vector_val(A Al = A())
        : Alval(Al) {}
    typedef typename A::template
        rebind<T>::other Alty;
    Alty Alval;
};

// TEMPLATE CLASS vector
template<class T, class Ax = allocator<T> >
class vector : public Vector_val<T, Ax> {
public:
    typedef vector<T, Ax> Myt;
    typedef Vector_val<T, Ax> Mybase;
    typedef typename Mybase::Alty A;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;
    typedef typename A::pointer Tptr;
    typedef typename A::const_pointer Cptr;
    typedef Tptr pointer;
    typedef Cptr const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef Ptrit<value_type, difference_type, Tptr,
        reference, Tptr, reference> iterator;
    typedef Ptrit<value_type, difference_type, Cptr,
        const_reference, Tptr, reference> const_iterator;
    typedef std::reverse_iterator<iterator>
        reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;
    vector()
        : Mybase()
    vBuy(0); }
    explicit vector(const A& Al)
        : Mybase(Al)
    {Buy(0); }
    explicit vector(size_type N)
        : Mybase()
    {if (Buy(N))
        Last = Ufill(First, N, T()); }
    vector(size_type N, const T& V)
        : Mybase()
    {if (Buy(N))
```

```
        Last = Ufill(First, N, V); }
vector(size_type N, const T& V, const A& Al)
: Mybase(Al)
(if (Buy(N))
    Last = Ufill(First, N, V); )
vector(const Myt& X)
: Mybase(X.Alval)
(if (Buy(X.size())))
    Last = Uccpy(X.begin(), X.end(), First); }
template<class It>
vector(It F, It L)
: Mybase()
vConstruct(F, L, Iter_cat(F)); }
template<class It>
vector(It F, It L, const A& Al)
: Mybase(Al)
(Construct(F, L, Iter_cat(F)); }
template<class It>
void Construct(It F, It L, Int_iterator_tag)
{size_type N = (size_type)F;
if (Buy(N))
    Last = Ufill(First, N, (T)L); }
template<class It>
void Construct(It F, It L, input_iterator_tag)
(Buy(0);
insert(begin(), F, L); }
~vector()
{Clear(); }
Myt& operator=(const Myt& X)
{if (this == &X)
;
else if (X.size() == 0)
{Clear(); }
else if (X.size() <= size())
(pointer Q = copy(X.begin(), X.end(), First);
Destroy(Q, Last);
Last - First + X.size(); }
else if (X.size() <= capacity())
{const_iterator S = X.begin() + size();
copy(X.begin(), S, First);
Last = Uccpy(S, X.end(), Last); }
else
{Destroy(First, Last);
Mybase::Alval.deallocate(First,
End - First);
if (Buy(X.size()))
Last = Uccpy(X.begin(), X.end(), First); }
```

```
        return (*this); }
void reserve(size_type N)
{
    if (max_size() < N)
        Xlen();
    else if (capacity() < N)
        {pointer Q = Mybase::Alval.allocate(N,
            (void *)0);
        try {
            Ucopy(begin(), end(), Q);
        } catch (...) {
            Mybase::Alval.deallocate(Q, N);
            throw;
        }
        if (First != 0)
            {Destroy(First, Last);
            Mybase::Alval.deallocate(First,
                End - First); }
        End = Q + N;
        Last = Q + size();
        First = Q; }
size_type capacity() const
{
    return (First == 0 ? 0 : End - First); }
iterator begin()
{
    return (iterator(First)); }
const_iterator begin() const
{
    return (const_iterator(First)); }
iterator end()
{
    return (iterator(Last)); }
const_iterator end() const
{
    return (const_iterator(Last)); }
reverse_iterator rbegin()
{
    return (reverse_iterator(end())); }
const_reverse_iterator rbegin() const
{
    return (const_reverse_iterator(end())); }
reverse_iterator rend()
{
    return (reverse_iterator(begin())); }
const_reverse_iterator rend() const
{
    return (const_reverse_iterator(begin())); }
void resize(size_type N)
{
    resize(N, T()); }
void resize(size_type N, T X)
{
    if (size() < N)
        insert(end(), N - size(), X);
    else if (N < size())
        erase(begin() + N, end()); }
size_type size() const
{
    return (First == 0 ? 0 : Last - First); }
```

```
size_type max_size() const
    {return (Mybase::Alval.max_size()); }
bool empty() const
    {return (size() == 0); }
A get_allocator() const
    {return (Mybase::Alval); }
const_reference at(size_type P) const
    {if (size() <= P)
        Xran();
     return (*begin() + P)); }
reference at(size_type P)
    {if (size() <= P)
        Xran();
     return (*begin() + P)); }
const_reference operator[](size_type P) const
    {return (*begin() + P)); }
reference operator[](size_type P)
    {return (*begin() + P)); }
reference front()
    {return (*begin()); }
const_reference front() const
    {return (*begin()); }
reference back()
    {return (*end() - 1)); }
const_reference back() const
    {return (*end() - 1)); }
void push_back(const T& X)
    {insert(end(), X); }
void pop_back()
    {erase(end() - 1); }
template<class It>
void assign(It F, It L)
    {Assign(F, L, Iter_cat(F)); }
template<class It>
void Assign(It F, It L, Int_iterator_tag)
    {assign((size_type)F, (T)L); }
template<class It>
void Assign(It F, It L, input_iterator_tag)
    {erase(begin(), end());
     insert(begin(), F, L); }
void assign(size_type N, const T& X)
    {T Tx = X;
     erase(begin(), end());
     insert(begin(), N, Tx); }
iterator insert(iterator P, const T& X)
    {size_type Off = size() == 0 ? 0 : P - begin();
     insert(P, (size_type)1, X); }
```

```
        return (begin() + Off); }
void insert(iterator P, size_type M, const T& X)
{T Tx = X;
size_type N = capacity();
if (M == 0)
{
}
else if (max_size() - size() < M)
    Xlen();
else if (N < size() + M)
{N = max_size() - N / 2 < N
? 0 : N + N / 2;
if (N < size() + M)
    N = size() + M;
pointer S = Mybase::Alval.allocate(N,
(void *)0);
pointer Q;
try {
Q = Ucopy(begin(), P, S);
Q = Ufill(Q, M, Tx);
Ucopy(P, end(), Q);
} catch (...) {
Destroy(S, Q);
Mybase::Alval.deallocate(S, N);
throw;
}
if (First != 0)
(Destroy(First, Last);
Mybase::Alval.deallocate(First,
End - First); )
End = S + N;
Last = S + size() + M;
First = S; }
else if ((size_type)(end() - P) < M)
{Ucopy(P, end(), P.base() + M);
try {
Ufill(Last, M - (end() - P), Tx);
} catch (...) {
Destroy(P.base() + M, Last + M);
throw;
}
Last -= M;
fill(P, end() - M, Tx); }
else
{iterator Oend = end();
Last = Ucopy(Oend - M, Oend, Last);
copy_backward(P, Oend - M, Oend);
fill(P, P + M, Tx); }}
```

```
template<class It>
    void insert(iterator P, It F, It L)
    {insert(P, F, L, Iter_cat(F)); }
template<class It>
    void Insert(iterator P, It F, It L,
                Int_iterator_tag)
    {insert(P, (size_type)F, (T)L); }
template<class It>
    void Insert(iterator P, It F, It L,
                input_iterator_tag)
    {for (; F != L; ++F, ++P)
        P = insert(P, *F); }
template<class It>
    void Insert(iterator P, It F, It L,
                forward_iterator_tag)
    {size_type M = 0;
     Distance(F, L, M);
     size_type N = capacity();
     if (M == 0)
         ;
     else if (max_size() - size() < M)
         Xlen();
     else if (N < size() + M)
         {N = max_size() - N / 2 < N
          ? 0 : N + N / 2;
          if (N < size() + M)
              N = size() + M;
          pointer S = Mybase::Alval.allocate(N,
                                              (void *)0);
          pointer Q;
          try {
              Q = Ucopy(begin(), P, S);
              Q = Ucopy(F, L, Q);
              Ucopy(P, end(), Q);
          } catch (...) {
              Destroy(S, Q);
              Mybase::Alval.deallocate(S, N);
              throw;
          }
          if (First != 0)
              {Destroy(First, Last);
               Mybase::Alval.deallocate(First,
                                       End - First); }
          End = S + N;
          Last = S + size() + M;
          First = S; }
     else if ((size_type)(end() - P) < M)
```

```

        {Ucopy(P, end(), P.base() + M);
        It Mid = F;
        advance(Mid, end() - P);
        try {
            Ucopy(Mid, L, Last);
        } catch (...) {
            Destroy(P.base() + M, Last + M);
            throw;
        }
        Last += M;
        copy(F, Mid, P); }
    else if (0 < M)
        (iterator Oend = end());
        Last = Ucopy(Oend - M, Oend, Last);
        copy_backward(P, Oend - M, Oend);
        copy(F, L, P); }
    iterator erase(iterator P)
        {copy(P + 1, end(), P);
        Destroy(Last - 1, Last);
        --Last;
        return (P); }
    iterator erase(iterator F, iterator L)
        {if (F != L)
            {pointer S = copy(L, end(), F.base());
            Destroy(S, Last);
            Last = S; }
        return (F); }
    void clear()
        {erase(begin(), end()); }
    bool Eq(const Myt& X) const
        {return (size() == X.size())
         && equal(begin(), end(), X.begin()); }
    bool Lt(const Myt& X) const
        {return (lexicographical_compare(begin(), end(),
         X.begin(), X.end())); }
    void swap(Myt& X)
        {if (Mybase::Alval == X.Alval)
            {std::swap(First, X.First);
            std::swap(Last, X.Last);
            std::swap(End, X.End); }
        else
            {Myt Ts = *this; *this = X, X = Ts; }}
protected:
    bool Buy(size_type N)
        {First = 0, Last = 0, End = 0;
        if (N == 0)
            return (false);

```

```
    else
        {First = Mybase::Alval.allocate(N,
            (void *)0);
        Last = First;
        End = First + N;
        return (true); }
void Clear()
{if (First != 0)
    {Destroy(First, Last);
    Mybase::Alval.deallocate(First,
        End - First); }
    First = 0, Last = 0, End = 0; }
void Destroy(pointer F, pointer L)
{for (; F != L; ++F)
    Mybase::Alval.destroy(F); }
template<class It>
pointer Ucopy(It F, It L, pointer Q)
{pointer Qs = Q;
try {
    for (; F != L; ++Q, ++F)
        Mybase::Alval.construct(Q, *F);
} catch (...) {
    Destroy(Qs, Q);
    throw;
}
return (Q); }
pointer Ufill(pointer Q, size_type N, const T &X)
{pointer Qs = Q;
try {
    for (; 0 < N; --N, ++Q)
        Mybase::Alval.construct(Q, X);
} catch (...) {
    Destroy(Qs, Q);
    throw;
}
return (Q); }
void Xlen() const
{throw length_error("vector<T> too long"); }
void Xran() const
{throw out_of_range("vector<T> subscript"); }
pointer First, Last, End;
};

// vector TEMPLATE FUNCTIONS
template<class T, class A> inline
bool operator==(const vector<T, A>& X,
    const vector<T, A>& Y)
```

```

        {return (X.Eq(Y)); }
template<class T, class A> inline
    bool operator!=(const vector<T, A>& X,
                      const vector<T, A>& Y)
    {return (!(X == Y)); }
template<class T, class A> inline
    bool operator<(const vector<T, A>& X,
                      const vector<T, A>& Y)
    {return (X.Lt(Y)); }
template<class T, class A> inline
    bool operator>(const vector<T, A>& X,
                      const vector<T, A>& Y)
    {return (Y < X); }
template<class T, class A> inline
    bool operator<=(const vector<T, A>& X,
                      const vector<T, A>& Y)
    {return (!(Y < X)); }
template<class T, class A> inline
    bool operator>=(const vector<T, A>& X,
                      const vector<T, A>& Y)
    {return (!(X < Y)); }
template<class T, class A> inline
    void swap(vector<T, A>& X, vector<T, A>& Y)
    {X.swap(Y); }

        // CLASS vector<bool, allocator>
typedef unsigned int Vbase;
const int VBITS = 8 * sizeof (Vbase); // min CHAR_BITS

template<class A>
    class vector<bool, A> {
public:
    typedef typename A::size_type size_type;
    typedef typename A::difference_type Dift;
    typedef std::vector<Vbase,
        typename A::template rebind<Vbase>::other>
        Vbtype;
    typedef std::vector<bool, A> Myt;
    typedef Dift difference_type;
    typedef bool T;
    typedef A allocator_type;
    // CLASS reference
class reference {
public:
    reference()
        : Mask(0), Ptr(0) {}
    reference(size_t Off, Vbase *P)

```

```
    : Mask((Vbase)1 << Off), Ptr(P) {}
reference& operator=(const reference& X)
    {return (*this = bool(X)); }
reference& operator=(bool V)
    {if (V)
        *Ptr |= Mask;
     else
        *Ptr &= ~Mask;
     return (*this); }
void flip()
    {*Ptr ^= Mask; }
bool operator~() const
    {return (!bool(*this)); }
operator bool() const
    {return ((*Ptr & Mask) != 0); }
protected:
    Vbase Mask, *Ptr;
};
typedef reference Reft;
typedef bool const_reference;
typedef bool value_type;

// CLASS iterator
class const_iterator;
class iterator
    : public Ranit<bool, Dift, Reft *, Reft> {
public:
    typedef Ranit<bool, Dift, Reft *, Reft> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    friend class const_iterator;
    iterator()
        : Off(0), Ptr(0) {}
    iterator(size_t O, typename Vbtype::iterator P)
        : Off(O), Ptr(P.base()) {}
    reference operator*() const
        {return (Reft(Off, Ptr)); }
    iterator& operator++()
    vInc();
        return (*this); }
    iterator operator++(int)
        {iterator Tmp = *this;
```

```
    Inc();
    return (Tmp); }
iterator& operator--()
{Dec();
 return (*this); }
iterator operator--(int)
{iterator Tmp = *this;
 Dec();
 return (Tmp); }
iterator& operator+=(difference_type N)
{Off += N;
 Ptr += Off / VBITS;
 Off %= VBITS;
 return (*this); }
iterator& operator-=(difference_type N)
{return (*this += -N); }
iterator operator+(difference_type N) const
{iterator Tmp = *this;
 return (Tmp += N); }
iterator operator-(difference_type N) const
{iterator Tmp = *this;
 return (Tmp -= N); }
difference_type operator-(const iterator X) const
{return (VBITS * (Ptr - X.Ptr)
 + (difference_type)Off
 - (difference_type)X.Off); }
reference operator[](difference_type N) const
{return (*(*this + N)); }
bool operator==(const iterator& X) const
{return (Ptr == X.Ptr && Off == X.Off); }
bool operator!=(const iterator& X) const
{return (!(*this == X)); }
bool operator<(const iterator& X) const
{return (Ptr < X.Ptr
 || Ptr == X.Ptr && Off < X.Off); }
bool operator>(const iterator& X) const
{return (X < *this); }
bool operator<=(const iterator& X) const
{return (!(*this < X)); }
bool operator>=(const iterator& X) const
{return (!(*this < X)); }

protected:
void Dec()
{if (Off != 0)
 --Off;
else
 Off = VBITS - 1, --Ptr; }
```

```
void Inc()
{
    if (Off < VBITS - 1)
        ++Off;
    else
        Off = 0, ++Ptr; }

size_t Off;
Vbase *Ptr;
};

typedef iterator Myit;

// CLASS const_iterator
class const_iterator : public Ranit<bool, Dift,
    const_reference *, const_reference> {
public:
    typedef Ranit<bool, Dift,
        const_reference *, const_reference> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    const_iterator()
        : Off(0), Ptr(0) {}
    const_iterator(size_t O,
        typename Vbtype::const_iterator P)
        : Off(O), Ptr(P.base()) {}
    const_iterator(const Myit& X)
        : Off(X.Off), Ptr(X.Ptr) {}
    const_reference operator*() const
        {return (Reft(Off, (Vbase *)Ptr)); }
    const_iterator& operator++()
        {Inc();}
        return (*this); }
    const_iterator operator++(int)
        {const_iterator Tmp = *this;
        Inc(); return (Tmp); }
    const_iterator& operator--()
        {Dec();}
        return (*this); }
    const_iterator operator--(int)
        {const_iterator Tmp = *this;
        Dec(); return (Tmp); }
    const_iterator& operator+=(difference_type N)
```

```

        (Off += N;
        Ptr += Off / VBITS;
        Off %= VBITS;
        return (*this); }
const_iterator& operator-=(difference_type N)
    {return (*this += -N); }
const_iterator operator+(difference_type N) const
    {const_iterator Tmp = *this;
    return (Tmp += N); }
const_iterator operator-(difference_type N) const
    {const_iterator Tmp = *this;
    return (Tmp -= N); }
difference_type operator-(const const_iterator X) const
    {return (VBITS * (Ptr - X.Ptr)
        + (difference_type)Off
        - (difference_type)X.Off); }
const_reference operator[](difference_type N) const
    {return (*(*this + N)); }
bool operator==(const const_iterator& X) const
    {return (Ptr == X.Ptr && Off == X.Off); }
bool operator!=(const const_iterator& X) const
    {return (!(*this == X)); }
bool operator<(const const_iterator& X) const
    {return (Ptr < X.Ptr
        || Ptr == X.Ptr && Off < X.Off); }
bool operator>(const const_iterator& X) const
    {return (X < *this); }
bool operator<=(const const_iterator& X) const
    {return (!(*this < X)); }
bool operator>=(const const_iterator& X) const
    {return (!(*this < X)); }

protected:
    void Dec()
        {if (Off != 0)
            --Off;
        else
            Off = VBITS - 1, --Ptr; }
    void Inc()
        {if (Off < VBITS - 1)
            ++Off;
        else
            Off = 0, ++Ptr; }
size_t Off;
const Vbase *Ptr;
};

typedef iterator pointer;

```

```
typedef const_iterator const_pointer;
typedef std::reverse_iterator<iterator>
    reverse_iterator;
typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;

vector()
    : Size(0), Vec() {}
explicit vector(const A& A1)
    : Size(0), Vec(A1) {}
explicit vector(size_type N, const bool V = false)
    : Size(0), Vec(Nw(N), V ? -1 : 0)
    {Trim(N); }
vector(size_type N, const bool V, const A& A1)
    : Size(0), Vec(Nw(N), V ? -1 : 0, A1)
    {Trim(N); }
template<class It>
vector(It F, It L)
    : Size(0), Vec()
    {BConstruct(F, L, Iter_cat(F)); }
template<class It>
vector(It F, It L, const A& A1)
    : Size(0), Vec(A1)
    {BConstruct(F, L, Iter_cat(F)); }
template<class It>
void BConstruct(It F, It L, Int_iterator_tag)
{size_type N = (size_type)F;
Vec.assign(N, (T)L ? -1 : 0);
Trim(N); }
template<class It>
void BConstruct(It F, It L, input_iterator_tag)
{insert(begin(), F, L); }
~vector()
{Size = 0; }
void reserve(size_type N)
{Vec.reserve(Nw(N)); }
size_type capacity() const
{return (Vec.capacity() * VBITS); }
iterator begin()
{return (iterator(0, Vec.begin())); }
const_iterator begin() const
{return (const_iterator(0, Vec.begin())); }
iterator end()
{iterator Tmp = begin();
if (0 < Size)
    Tmp += Size;
return (Tmp); }
```

```
const_iterator end() const
    {const_iterator Tmp = begin();
     if (0 < Size)
         Tmp += Size;
     return (Tmp); }
reverse_iterator rbegin()
    {return (reverse_iterator(end())); }
const_reverse_iterator rbegin() const
    {return (const_reverse_iterator(end())); }
reverse_iterator rend()
    {return (reverse_iterator(begin())); }
const_reverse_iterator rend() const
    {return (const_reverse_iterator(begin())); }
void resize(size_type N, bool X = false)
    {if (size() < N)
        insert(end(), N - size(), X);
     else if (N < size())
        erase(begin() + N, end()); }
size_type size() const
    {return (Size); }
size_type max_size() const
    {return (Vec.max_size() * VBITS); }
bool empty() const
    {return (size() == 0); }
A get_allocator() const
    {return (Vec.get_allocator()); }
const_reference at(size_type P) const
    {if (size() <= P)
        Xran();
     return (*begin() + P)); }
reference at(size_type P)
    {if (size() <= P)
        Xran();
     return (*begin() + P)); }
const_reference operator[](size_type P) const
    {return (*begin() + P)); }
reference operator[](size_type P)
    {return (*begin() + P)); }
reference front()
    {return (*begin()); }
const_reference front() const
    {return (*begin()); }
reference back()
    {return (*(end() - 1)); }
const_reference back() const
    {return (*(end() - 1)); }
void push_back(const bool X)
```

```
    {insert(end(), x); }
void pop_back()
    {erase(end() - 1); }
template<class It>
    void assign(It F, It L)
        {Assign(F, L, Iter_cat(F)); }
template<class It>
    void Assign(It F, It L, Int_iterator_tag)
        {assign((size_type)F, (T)L); }
template<class It>
    void Assign(It F, It L, input_iterator_tag)
        {erase(begin(), end());
         insert(begin(), F, L); }
void assign(size_type N, const T& X)
    {T Tx = X;
     erase(begin(), end());
     insert(begin(), N, Tx); }
iterator insert(iterator P, const bool X)
    {size_type Off = P - begin();
     insert(P, (size_type)1, X);
     return (begin() + Off); }
void insert(iterator P, size_type M, const bool X)
    {if (M == 0)
     ;
     else if (max_size() - size() < M)
         Xlen();
     else
         {if (size() + M <= capacity())
          ;
          else if (size() == 0)
              (Vec.resize(Nw(size() + M), 0));
              P = begin(); }
     else
         {size_type Off = P - begin();
          Vec.resize(Nw(size() + M), 0);
          P = begin() + Off;
          copy_backward(P, end(), end() + M); }
     fill(P, P + M, X);
     Size += M; }}
template<class It>
    void insert(iterator P, It F, It L)
        {Insert(P, F, L, Iter_cat(F)); }
template<class It>
    void Insert(iterator P, It F, It L,
                Int_iterator_tag)
        {insert(P, (size_type)F, (T)L); }
template<class It>
```

```

        void Insert(iterator P, It F, It L,
                    input_iterator_tag)
    {size_type Off = P - begin();
     for (; F != L; ++F, ++Off)
         insert(begin() + Off, *F); }
template<class It>
    void Insert(iterator P, It F, It L,
                forward_iterator_tag)
{size_type M = 0;
Distance(F, L, M);
if (M == 0)
    ;
else if (max_size() - size() < M)
    Xlen();
else
    {if (size() + M <= capacity())
     ;
     else if (size() == 0)
         {Vec.resize(Nw(size() + M), 0);
          P = begin(); }
     else
         {size_type Off = P - begin();
          Vec.resize(Nw(size() + M), 0);
          P = begin() + Off;
          copy_backward(P, end(), end() + M); }
     copy(F, L, P);
     Size += M; }}
iterator erase(iterator P)
{copy(P + 1, end(), P);
Trim(Size - 1);
return (P); }
iterator erase(iterator F, iterator L)
{iterator S = copy(L, end(), F);
Trim(S - begin());
return (F); }
void clear()
{erase(begin(), end()); }
void flip()
{for (typename Vbtype::iterator S = Vec.begin();
      S != Vec.end(); ++S)
    *S = ~*S;
Trim(Size); }
bool Eq(const Myt& X) const
{return (Size == X.Size && Vec == X.Vec); }
bool Lt(const Myt& X) const
{return (lexicographical_compare(begin(), end(),
X.begin(), X.end())); }

```

```

void swap(Myt& X)
    {std::swap(Size, X.Size);
     Vec.swap(X.Vec); }
static void swap(reference X, reference Y)
    {bool V = X;
     X = Y;
     Y = V; }

protected:
    static size_type Nw(size_type N)
        {return ((N + VBITS - 1) / VBITS); }
    void Trim(size_type N)
        {if (size() < N && max_size() <= N)
         Xlen();
         size_type M = Nw(N);
         if (M < Vec.size())
             Vec.erase(Vec.begin() + M, Vec.end());
         Size = N;
         N %= VBITS;
         if (0 < N)
             Vec[M - 1] &= ((Vbase)1 << N) - 1; }
    void Xlen() const
        {throw out_of_range("vector<bool> too long"); }
    void Xran() const
        {throw out_of_range("vector<bool> subscript"); }
    size_type Size;
    Vbtype Vec;
};

typedef vector<bool, allocator<bool> > Bvector;
} /* namespace std */
#endif /* VECTOR_ */

```

Vector_val

在该文件的一开始处定义了模板类**Vector_val**。其特化版本**Vector_val<T, A>**用来作为**vector<T, A>**的公共基类。它惟一的目的就是存储分配器对象**Alval**。我们可以从先前的讨论中回想起来，所有为STL容器控制的存储空间实际上是由构造该容器对象时指定的分配器来管理的。（参见第4章中讨论的分配器，以及第9章，在那里我们讨论的容器的一般性质。）最初发明分配器的目的是为了分配和释放某些元素类型为**T**的对象的数组。从那以后，它的功能也变得越来越值得炫耀，越来越复杂了。但是模板类**vector**还是满足了大部分直接使用分配器功能的要求。

- 它通过调用**Alval.allocate**分配数组，然后调用**Alval.deallocate**来释放它们。
- 它通过调用**Alval.construct**构造元素，然后调用**Alval.destroy**来销毁它们。

大小为零的分配器

- 它通过调用 Alval.max_size 估计所能允许的最大序列的长度。

我们马上就可以在后续章节中看到，它不可能比这还要简单了。

但就是这样一个简单的用法还是存在着一个烦人的实现问题。Alval 通常都是一个没有成员对象的对象。（C++ 标准并不保证能够从有成员对象的分配器中获益。）不幸的是，尽管没有任何内容，这样的对象也很少不占存储空间。在一个典型的实现中，一个 `vector<T, A>` 对象可以轻易地占据 12 到 16 个字节，这主要是由于它被迫存储一个本身不需要占据存储空间的分配器对象。对于其他 STL 容器来说，这个比例可能会更大。

幸运的是，C++ 标准给出了一种方法来避免这个额外开销。它允许实现在基类的对象没有成员对象时在派生类的对象中保留 0 个字节的空间。我们所需做的就是把这个分配器对象放到一个基类对象中，然后让智能编译器去节省空间。本实现对于所有的分配器对象都采取了这种办法，并期待着这样的空间优化能够普遍起来。由此产生的代码可能可读性比较差，但所获得的回报却高于为此所增加的复杂度的代价。

`rebind`

模板类 `Vector_val` 还有着一个从严格意义上说不必要的复杂性。它使用模板成员类 `rebind` 来定义分配器类型 `Alty`，以此来映射分配器模板参数 `A`。严格地说，`A` 必须已经是一个服务于类型 `T` 对象的分配器，这样，映射实际上什么也不做。然而，对于后续章节中给出的其他容器来说，情况就往往会不一样。它们需要的分配器可能并不是为那些从模板参数中得到的元素类型服务的。本实现基于健壮性的考虑，选择了映射所有的分配器的模板参数。即使我们用一个服务于不正确类型的对象分配器来特化一个容器，所有的容器也都会产生适当的分配器对象。

`vector` 对象的核心就是它所存储的用于表达被控序列的数据。除了分配器对象外，`vector` 还存储了三个指针：

`First`

- `First` 指向被分配的用于存储被控序列的数组中的第一个元素。如果没有分配任何数组，它将是一个空指针。

`Last`

- 如果 `First` 不为空，那么 `Last` 就指向数组中最后一个有效元素的下一个位置。

`End`

- 如果 `First` 不为空，`End` 指向的是数组中最后一个元素的下一个位置。于是我们就可以得到一个显而易见的恒等式：

```
First == 0 || First <= Last && Last <= End
```

`Buy` `Clear` `Destroy`

有几个保护型的成员函数可完成一些常见的操作。调用 `Buy(N)` 会分配一个 `N` 个元素的数组并正确初始化三个指针成员对象。`Destroy(F, L)` 会销毁由区间 `[F, L)` 指定的所有数组元素。（注意，这里的参数是指针，而不是迭代器。）`Clear()` 销毁已分配的数组并清空那三个指针成员对象。

`Ucopy` `Ufill`

STL 容器要求能够在程序员所提供的构造函数抛出异常的情况下保持其恢复能力。为此，标准 C++ 库提供了三个模板函数用以处理在初始化序

列时出现的异常。它们是 `uninitialized_copy`、`uninitialized_fill` 和 `uninitialized_fill_n`。（参见第 6 章。）但这三个模板函数都不能很好地满足模板类 `vector` 的要求，`vector` 要求的是用一个分配器对象来实现所有的对象构造工作。为此，它定义了保护型的模板成员函数 `Ucopy` 和保护型的模板成员函数 `Ufill`，用它们来实现一种类似的、但设计得更好的功能。

例如，调用 `Ucopy(F, L, Q)` 将会以 `[F, L)`（任意迭代器类型）指定的序列为初始值来初始化数组中由 `Q`（为一个指针）开始的元素。而调用 `Ufill(Q, N, X)` 将会把数组中从 `Q`（同样也是一个指针）开始的 `N` 个元素初始化为值 `X`。在这两种情况下，函数会捕获抛出的异常并销毁已经构造好的对象，然后重新抛出异常。

`xlen`
`Xran`

最后，保护型的成员函数 `Xlen` 中封装了抛出异常 `length_error` 的代码，`Xran` 中封装了抛出异常 `out_of_range` 的代码。这两个函数都会构造出一个异常对象和一条模板类 `vector` 所独有的异常信息。

模板成员函数：

```
template<class It>
void insert(iterator P, It F, It L);
```

`Int_iterator_tag`

使用了一个有趣的设计。（在那两个模板构造函数和另一个模板成员函数中也是这样，但 `insert` 是最容易说明的。）当 `It` 是一个输入迭代器或者一个前向迭代器（甚至功能更强的迭代器）时，它会选择不同的策略。选择最佳策略的技术我们已经在前面的章节中见识过了。它的本质就是根据模板函数 `Iter_cat(F)` 的返回类型在模板成员函数 `Insert` 的不同重载版本中选择一个合适的版本。但在此次有个新的诀窍。当 `Iter_cat(F)` 返回一个类型为 `Int_iterator_tag` 的对象时，它选择的重载版本为：

```
template<class It>
void Insert(iterator P, It F, It L, Int_iterator_tag)
{insert(P, (size_type)F, (T)L); }
```

该模板成员函数对于 `F` 和 `L` 这两个“迭代器”的解释完全不同。它把第一个看做是重复计数，把第二个看做是元素中所存储的值，当这样使用这个模板成员函数时：

```
iterator insert(iterator P, const T& X);
```

为什么所有的都重写了？C++ 标准认为，在有些时候 `insert` 的模板版本的选择可能会有问题。假设有一个类型为 `vector<size_t>` 的对象 `cont`。调用 `cont.insert(begin(), 3, 0)` 明显是想往受控序列的开始处插入三个值为 0 的元素。然而，它无论如何都像是一个带有两个类型为 `int` 的迭代器的调用。C++ 标准只讲述了实现应该“把事情做对”，但却没有讲应该如何做。

本实现引入了一个伪迭代器种类，用它来包含所有的整数类型（整数类型永远不可能是一个真正的迭代器）。它还定义相应的迭代器标签

`Int_iterator_tag`, 并且重载了`Iter_cat`使得当输入为整型时, 返回一个该类型的对象。(参见第3章。)于是, 它就可以凭借通常的标签重载(参见第3章的“实现`<iterator>`”一节)来将整型参数同真正的迭代器区分开来。

`insert`

如果已有的数组缺乏足够的容量来存储最终的被控序列, 成员函数`insert`的两个版本(一个是模板)会分配一个新的数组。标准的做法是在每次增长时使数组大小加倍。总共调用构造函数的次数(包括创建初始元素以及在两个数组中元素的复制)保持和被控序列的总长度成线性关系。存储分配大概还有着一个固定的开销, 它和请求的存储空间大小无关; 当整个长度增大时, 它们也会相应地成对数级数增长。也就是说, 按照C++标准所要求的, 增加每个元素所产生的分摊的开销在本质上可以为常数。我们所付出的代价是降低存储空间的有效利用。空闲的容量保持在整个数组大小的0至50%之间。平均来说, 一个增长着的被控序列中, 其空闲的数组元素与有效的数组元素数目的比例大致为1:3。

`insert`的两个版本中都使用了这种策略的一个修改版本。只要可能, 它们就会分配至少比已有数组大50%的新数组。这种方法使得新增一个元素有相近的时间复杂度, 完成更多的存储分配, 并提高存储空间的使用率。

`vector<bool>`

模板部分特化版本的`vector<bool>`至少和模板类`vector`一样大小。它们都必须提供本质上一样的服务, 但`vector<bool>`还有另外的特性。它必须处理单个位的元素, 这种技巧在C或C++程序中不常见。

`Vbase` `VBITS`

我们用以实现`vector<bool>`的技巧就是利用`vector<Vbase>`, 此处`Vbase`是一个无符号整数类型, 实现可以很高效地对它进行操作。我们为它选择的类型是`unsigned int`。假设一个字节至少有8位(这是一个对于C标准来说比较安全的假设), 代码就会去检测`VBITS`的大小(它等于一个类型为`Vbase`的对象所占据的最小位数)。于是`vector<bool>`对象就可以在它的基础序列中的每个元素里面存储`VBITS`个布尔元素。

`reference` `iterator`

真正的技巧来源于成员类`reference`和`iterator`的定义。一个`reference`对象必须可以根据需要对基础序列中的位进行打包(pack)和解包(unpack)。一个`iterator`对象必须可以产生一连串的引用。注意, 这些迭代器和类`vector<bool>`本身都定义了附加的成员函数`flip`, 用来翻转(即对真值取反)讨论中的一位或所有的位。类`vector<bool>`还重载了`swap`, 以此来交换被控序列中的两个位。

测试`<vector>`

`tvector.c`

程序清单10-2列出了文件`tvector.c`。它是看起来很相似的三个测试程序中的一个, 另外两个是文件`tlist.c`和文件`tdeque.c`。为了简单地比较这三个测试程序, 我们只是简单地注释掉那些不适用于给定容器的测试段, 而不是把这些没有使用的代码删除。

```
程序清单 10-2: // test <vector>
tvector.c      #include <assert.h>
               #include <iostream>
               #include <vector>
using namespace std;

               // TEST <vector>
int main()
{typedef allocator<char> Myal;

               // TEST vector
typedef vector<char, Myal> Mycont;
char ch, carr[] = "abc";
Mycont::allocator_type *p_alloc = (Myal *)0;
Mycont::pointer p_ptr = (char *)0;
Mycont::const_pointer p_cptr = (const char *)0;
Mycont::reference p_ref = ch;
Mycont::const_reference p_cref = (const char&)ch;
Mycont::value_type *p_val = (char *)0;
Mycont::size_type *p_size = (size_t *)0;
Mycont::difference_type *p_diff = (ptrdiff_t *)0;
Mycont v0;
Myal al = v0.get_allocator();
Mycont v0a(al);
assert(v0.empty() && v0.size() == 0);
assert(v0a.size() == 0 && v0a.get_allocator() == al);
Mycont v1(5), v1a(6, 'x'), v1b(7, 'y', al);
assert(v1.size() == 5 && v1.back() == '\0');
assert(v1a.size() == 6 && v1a.back() == 'x');
assert(v1b.size() == 7 && v1b.back() == 'Y');
Mycont v2(v1a);
assert(v2.size() == 6 && v2.front() == 'x');
Mycont v3(v1a.begin(), v1a.end());
assert(v3.size() == 6 && v3.front() == 'x');
const Mycont v4(v1a.begin(), v1a.end(), al);
assert(v4.size() == 6 && v4.front() == 'x');
v0 = v4;
assert(v0.size() == 6 && v0.front() == 'x');
assert(v0[0] == 'x' && v0.at(5) == 'x');

v0.reserve(12);
assert(12 <= v0.capacity());
v0.resize(8);
assert(v0.size() == 8 && v0.back() == '\0');
v0.resize(10, 'z');
```

```
assert(v0.size() == 10 && v0.back() == 'z');
assert(v0.size() <= v0.max_size());

Mycont::iterator p_it(v0.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v0.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert(*p_it == 'x' && *--(p_it = v0.end()) == 'z');
assert(*p_cit == 'x' && *--(p_cit = v4.end()) == 'x');
assert(*p_rit == 'z'
       && *--(p_rit = v0.rend()) == 'x');
assert(*p_crit == 'x'
       && *--(p_crit = v4.rend()) == 'x');

assert(v0.front() == 'x' && v4.front() == 'x');
// v0.push_front('a');
// assert(v0.front() == 'a');
// v0.pop_front();
// assert(v0.front() == 'x' && v4.front() == 'x');
v0.push_back('a');
assert(v0.back() == 'a');
v0.pop_back();
assert(v0.back() == 'z' && v4.back() == 'x');

v0.assign(v4.begin(), v4.end());
assert(v0.size() == v4.size()
      && v0.front() == v4.front());
v0.assign(4, 'w');
assert(v0.size() == 4 && v0.front() == 'w');
assert(*v0.insert(v0.begin(), 'a') == 'a');
assert(v0.front() == 'a'
      && *++v0.begin() == 'w');
v0.insert(v0.begin(), 2, 'b');
assert(v0.front() == 'b'
      && *++v0.begin() == 'b'
      && *++v0.begin() == 'a');
v0.insert(v0.end(), v4.begin(), v4.end());
assert(v0.back() == v4.back());
v0.insert(v0.end(), carr, carr + 3);
assert(v0.back() == 'c');
v0.erase(v0.begin());
assert(v0.front() == 'b' && *++v0.begin() == 'a');
v0.erase(v0.begin(), ++v0.begin());
assert(v0.front() == 'a');
v0.clear();
assert(v0.empty());
v0.swap(v1);
```

```
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

// TEST vector<bool>
typedef vector<bool, allocator<bool> > Bvector;
Bvector bv(3);
bv[0] = ~bv[1];
bv.flip();
assert(!bv[0] && bv[1] && bv[2]);
Bvector::swap(bv[0], bv[1]);
assert(bv[0] && !bv[1]);
cout << "SUCCESS testing <vector>" << endl;
return (0); }
```

该测试程序只是对模板类vector的一个特化版本进行了简单的测试，按照其意图测试了它所给出的每个成员函数和成员类型。它还测试了类vector<bool>所特有的成员函数。如果测试一切顺利的话，测试程序将打印出：

```
SUCCESS : testing <vector>
```

然后正常退出。

习题

习题10-1

为什么整数类型不可以当作迭代器来使用？

习题10-2

怎样才可以“把事情做对”并且在不使用Int_iterator_tag的情况下在调用模板函数时，将它的整型参数和迭代器区分开来？

习题10-3

有一种可以用来增长向量存储空间的方法（它不符合C++标准的要求），它每次分配的新数组都比原来的数组大一个元素的空间。已有的数组复制过去后又被销毁，然后在新数组的末端构造出新增的元素。假设一次只增加一个元素，请分别计算为了产生有1、10、100、1000和10000个元素的向量，需要进行多少次分配，调用多少次构造函数，以及调用多少次析构函数。对于增加一个单独的元素到向量中所产生的分摊的时间复杂度，从这个例子中你可以学到什么？

习题10-4

还有一种可以用来增长向量存储空间的方法（它符合C++标准的要求），它每次分配的新数组都是原来数组的两倍大。对于这种策略，请重复上面的练习，同时计算当向量增长时，未使用存储容量的平均值是多少。

- 习题10-5 还有一种用来增长向量存储空间的方法（它同样也符合C++标准的要求），它每次分配的新数组都是原来数组的1.5倍大。对于这种策略，请重复上面的练习。
- 习题10-6 概括从上面几题中所学习到的知识并描述不同的增长策略在时间复杂度、性能以及空间利用率之间做出的折衷。
- 习题10-7 如果没有提供部分特化版本vector<bool>，那么一个存储有10000个元素的布尔向量所占据的空间将会是多少？在什么情况下，我们应当选择前一种表示法，而不是后一种？
- 习题10-8 在什么方式下，vector<bool>不符合对于STL容器的要求？
- 习题10-9 [较难] 请给出一个好的规则来缩小模板类vector用于存储受控序列的存储空间。
- 习题10-10 [特难] 在不影响其性能要求的前提下，如何实现模板类vector，使得它所控制的序列存储在不连续的存储空间中？

第11章 <list>

背景知识

<list>
list

头文件<list>中仅定义了模板类list，它是一个容器，它所控制的长度为N的序列是以一个有着N个节点的双向链表来存储的，每个节点存储一个单独的元素。链表的优势在于它的弹性。我们可以向链表中随意地插入和删除元素，所需做的仅是改变一下节点中的前向和后向链接。我们甚至还可以往链表中接入一个子链表。此时链表节点本身并没有在内存中移来移去。我们可以得到这样的推论：我们所维护的任何一个指向单独节点的迭代器在该节点的生命周期内都能维持其有效性。同样，任何我们所维护的指向链表中单独元素的指针在包含这个元素的节点的生命周期内也都有效。

我们为此所付出的代价是只能以连续的方式对序列中的元素进行存取。例如，为了存取第*i*个元素，我们不得不从容器所存储的链表表头开始，一个节点一个节点地移动*i*次，直到碰到第*i*个节点为止。可以从任意的方向链接，但还是应当遵循必需的原则。这样，查找任意元素的平均时间就和受控序列长度成线性比例。按照STL的术语来说，模板类list支持双向迭代器。

表9-1列出了模板类list与其他STL容器的对比。在所有需要重排序列的操作（插入、删除和替换）中，它是毫无争议的赢家。但在所有的元素定位操作（查找和随机存取）中，它却是实实在在的输家。同时，它还需要在每个元素上增加一些比较严重的开销：即每个节点的前向和后向指针。

splice
sort
merge

模板类list定义了几个成员函数，利用了它所特有的属性。例如，我们可以把一个列表中的元素接入到另一个列表中，还可以对一个列表排序，或是把一个排好序的列表合并到另一个列表中。所有的这些操作实际上只是改动了链表节点之间的链接，并没有发生复制元素的行为。当元素列表的复制成本很高时（例如元素很大或者有着非平凡的复制语义），这样做就可以获得很大的收益。

异常情况下
的安全性

模板类list还有另外一个优点。它是唯一保证在用户代码抛出异常时有可预见行为的模板容器。其他的容器则只能给出一个软弱无力的保证（参见第9章）。对任何容器来说，其成员函数在执行中抛出的异常，使

容器本身处于一种一致的状态，可以被销毁，并且容器没有对其所分配的存储空间失去控制。但对大部分操作，尤其是那些能够影响多个元素的操作来说，当异常抛出时，并没有指定容器的精确状态。相反，list保证对于大部分成员函数中所抛出的异常，容器都能够恢复到操作前的最初状态，并且将异常继续抛出。

概括说来，当我们需要在重排序列以及用重排后仍然保持有效的迭代器来跟踪单个元素时获得足够的弹性，就应该使用模板类list。此外，如果需要在异常出现时有足够的决定性，也应该使用模板类list。另一方面，即使列表已经排好序，对它进行任意元素的定位操作所需的开销还是会比较大，这是因为每次我们都不得不进行一次线性查找。当快速存取元素显得很重要时，请考虑使用其他容器。

功能描述

```

namespace std {
template<class T, class A>
class list;

// TEMPLATE FUNCTIONS
template<class T, class A>
bool operator==(const list<T, A>& lhs, const list<T, A>& rhs);
template<class T, class A>
bool operator!=(const list<T, A>& lhs, const list<T, A>& rhs);
template<class T, class A>
bool operator<(const list<T, A>& lhs, const list<T, A>& rhs);
template<class T, class A>
bool operator>(const list<T, A>& lhs, const list<T, A>& rhs);
template<class T, class A>
bool operator<=(const list<T, A>& lhs, const list<T, A>& rhs);
template<class T, class A>
bool operator>=(const list<T, A>& lhs,
```

```
        const list<T, A>& rhs);
template<class T, class A>
void swap(
    list<T, A>& lhs,
    list<T, A>& rhs);
};
```

包含STL标准头文件<list>可以得到模板类list以及几个支持模板的定义。

¶ list

```
template<class T, class A = allocator<T> >
class list {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer
        const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    list();
    explicit list(const A& al);
    explicit list(size_type n);
    list(size_type n, const T& v);
    list(size_type n, const T& v, const A& al);
    list(const list& x);
    template<class InIt>
        list(InIt first, InIt last);
    template<class InIt>
        list(InIt first, InIt last, const A& al);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n);
    void resize(size_type n, T x);
    size_type size() const;
    size_type max_size() const;
```

```

        bool empty() const;
        A get_allocator() const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;
        void push_front(const T& x);
        void pop_front();
        void push_back(const T& x);
        void pop_back();
        template<class InIt>
            void assign(InIt first, InIt last);
            void assign(size_type n, const T& x);
            iterator insert(iterator it, const T& x);
            void insert(iterator it, size_type n, const T& x);
        template<class InIt>
            void insert(iterator it, InIt first, InIt last);
            iterator erase(iterator it);
            iterator erase(iterator first, iterator last);
            void clear();
            void swap(list& x);
            void splice(iterator it, list& x);
            void splice(iterator it, list& x, iterator first);
            void splice(iterator it, list& x, iterator first,
                         iterator last);
            void remove(const T& x);
        template<class Pred>
            void remove_if(Pred pr);
            void unique();
        template<class Pred>
            void unique(Pred pr);
            void merge(list& x);
        template<class Pred>
            void merge(list& x, Pred pr);
            void sort();
        template<class Pred>
            void sort(Pred pr);
            void reverse();
    };

```

该模板类描述的对象控制的是一个元素类型为T的可变长度序列。该序列以双向链表的方式存储，链表中每个元素中都包含一个类型为T的成员。

list对象是通过存储于其内部的一个类A的分配器对象进行存储空间的分配及释放的。该分配器对象必须拥有和模板类allocator一样的外部接

口。注意，在对容器进行赋值时，存储的分配器对象并没有被复制。

只有在成员函数需要对被控序列插入或删除元素时，list对象才会重新分配空间（list reallocation）。在这种情况下，只有指向被控序列中被删除位置的迭代器和引用才会变为无效。

所有向被控序列中添加元素的行为最后都要调用insert，它是list中惟一一个调用构造函数T(const T&)的成员函数。如果上述表达式抛出异常，容器将不会插入新元素，并且捕获到的异常也会继续向外抛出。也就是说，当这样的异常发生时，模板类list的对象会处于一个可知的状态中。

口 list::allocator_type

```
typedef A allocator_type;
```

该类型是模板参数A的同义词。

口 list::assign

```
template<class InIt>
void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

如果InIt是整数类型，第一个成员函数的行为与assign((size_type)first, (T)last)相同。否则，第一个成员函数会把*this所控制的序列替换成序列[first, last)，其中[first, last)不能和最初所控制的序列重叠。第二个成员函数将*this所控制的序列替换成一个由n个值为x的元素组成的新序列。

口 list::back

```
reference back();
const_reference back() const;
```

该成员函数返回一个引用，它指向被控序列中的最后一个元素（注意：被控序列不能为空）。

口 list::begin

```
const_iterator begin() const;
iterator begin();
```

该成员函数返回一个双向迭代器，它指向被控序列的第一个元素（如果被控序列为空，则指向紧接着序列末端的下一个位置）。

口 list::clear

```
void clear();
```

该成员函数调用erase(begin(), end())。

口 list::const_iterator

```
typedef Ti const_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量双向迭代器来使用。在此处它被描述为由实现定义的类型Ti的同义词。

- `list::const_pointer`
`typedef typename A::const_pointer
const_pointer;`
该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。
- `list::const_reference`
`typedef typename A::const_reference const_reference;`
该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。
- `list::const_reverse_iterator`
`typedef reverse_iterator<const_iterator>
const_reverse_iterator;`
该类型描述的对象可以作为一个指向被控序列的常量反转型双向迭代器来使用。
- `list::difference_type`
`typedef T3 difference_type;`
该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。
- `list::empty`
`bool empty() const;`
当被控序列为空时，该成员函数返回true。
- `list::end`
`const_iterator end() const;
iterator end();`
该成员函数返回一个双向迭代器，它指向紧接着序列末端的下一个位置。
- `list::erase`
`iterator erase(iterator it);
iterator erase(iterator first, iterator last);`
第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除被控序列中由[first, last)所指定的区间内的所有元素。这两个函数都返回一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。
删除N个元素将导致调用N次析构函数。由于没有任何重新分配发生，只有指向被删除对象的迭代器和引用才会变为无效。
这些成员函数不会抛出任何异常。

- `list::front`

```
    reference front();
    const_reference front() const;
```

该成员函数返回指向被控序列中第一个元素的引用（被控序列不能为空）。

- `list::get_allocator`

```
    A get_allocator() const;
```

该成员函数返回存储的分配器对象。

- `list::insert`

```
    iterator insert(iterator it, const T& x);
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
```

所有这些成员函数都是在被控序列中由it所指定的元素前插入新的序列（由其他的操作数决定）。第一个成员函数插入一个值为x的单个元素，然后返回一个指向刚插入的这个元素的迭代器。第二个成员函数插入n个值为x的元素。

如果InIt是一个整数类型，那么最后一个成员函数就相当于insert(it, (size_type)first, (T)last)。否则，最后那个成员函数将向被控序列中插入序列[first, last]（它不能和最初的被控序列重叠）。

插入N个元素将导致调用N次构造函数。由于没有任何重新分配发生，所以没有迭代器或引用会变为无效。

如果在插入一个或多个元素时有异常抛出，那么容器将保持不变，并且继续将该异常向外抛出。

- `list::iterator`

```
    typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的双向迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

- `list::list`

```
    list();
    explicit list(const A& a1);
    explicit list(size_type n);
    list(size_type n, const T& v);
    list(size_type n, const T& v,
          const A& a1);
    list(const list& x);
    template<class InIt>
        list(InIt first, InIt last);
    template<class InIt>
```

```
list(InIt first, InIt last, const A& al);
```

所有的构造函数都会向对象中存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它是x.get_allocator()。否则，分配器对象为A()。

开始的两个构造函数指定空的被控序列。第三个构造函数指定有n个值为T()的元素的序列。第四个和第五个构造函数指定有n个值为x的元素的序列。第六个构造函数指定一个由x控制的序列的拷贝。如果InIt是一个整数类型，最后两个构造函数指定有(size_type)first个值为(T)last的元素的序列；否则，这两个构造函数指定序列[first, last]。没有一个构造函数在调用过程中会产生临时的重新分配。

口 `list::max_size`

```
size_type max_size() const;
```

该成员函数返回对象所能控制的最长序列的长度。

口 `list::merge`

```
void merge(list& x);
template<class Pred>
void merge(list& x, Pred pr);
```

这两个成员函数删除最初被x所控制的序列中的所有元素，并把它们插入到被对象本身所控制的序列中。这两个序列必须是按相同的谓词（我们将在下面描述该谓词）排序的。结果序列同样也必须是按该谓词排序。

假设迭代器Pi指向位于位置i处的元素，Pj指向位于位置j处的元素，当i < j时，第一个成员函数将会要求!(*Pj < *Pi)为true。（即元素是以升序排序的。）第二个成员函数则要求当i < j时，!pr(*Pj, *Pi)为true。

最初序列中元素对的顺序在结果序列中不会改变。如果一对元素在结果序列中相等 (!(*Pi < *Pj) && !(*Pj < *Pi))，那么来自于最初被控序列的元素将会出现在来自于x所控制的序列的那个元素前面。

在这两个成员函数中，仅有在pr抛出异常时它们才会有异常发生。在这种情况下，被控序列将会处于一个未指定的排序状态下，而异常则继续向外抛出。

口 `list::pointer`

```
typedef typename A::pointer pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 `list::pop_back`

```
void pop_back();
```

该成员函数删除被控序列的最后一个元素，在执行pop_back前序列不能为空。

该成员函数不会抛出任何异常。

口 `list::pop_front`

```
void pop_front();
```

该成员函数删除序列中的第一个元素，在执行`pop_front`前序列不能为空。

该成员函数不会抛出任何异常。

口 `list::push_back`

```
void push_back(const T& x);
```

该成员函数把一个值为x的元素插入到被控序列的末端。

如果在调用过程中有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。

口 `list::push_front`

```
void push_front(const T& x);
```

该成员函数把一个值为x的元素插入到被控序列的开始处。

如果在调用过程中有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。

口 `list::rbegin`

```
const_reverse_iterator rbegin() const;  
reverse_iterator rbegin();
```

该成员函数返回一个反转型双向迭代器，它指向紧接着序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 `list::reference`

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 `list::remove`

```
void remove(const T& x);
```

该成员函数在被控序列中删除符合`*P == x` (`P`为一个指向元素的迭代器) 的元素。

该成员函数不会抛出任何异常。

口 `list::remove_if`

```
template<class Pred>  
void remove_if(Pred pr);
```

该成员函数在被控序列中删除使`pr(*P)` (`P`为一个指向元素的迭代器) 为`true`的元素。

只有在`pr`抛出异常时，该成员函数才会有异常发生，在这种情况下，

被控序列将会处于一个未指定的状态，而异常则继续向外抛出。

口 `list::rend`

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

该成员函数返回一个反转型双向迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 `list::resize`

```
void resize(size_type n);
void resize(size_type n, T x);
```

这两个成员函数都确保调用`size()`返回n。如果必须要扩展被控序列的长度，那么第一个成员函数会向被控序列末端添加值为T()的元素，而第二个元素添加值为x的元素。如果想缩短被控序列的长度，这两个成员函数都会调用`erase(begin() + n, end())`。

口 `list::reverse`

```
void reverse();
```

该成员函数反转出现在被控序列中的所有元素的顺序。

口 `list::reverse_iterator`

```
typedef reverse_iterator<iterator>
reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型双向迭代器来使用。

口 `list::size`

```
size_type size() const;
```

该成员函数返回被控序列的长度。

口 `list::size_type`

```
typedef T2 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

口 `list::sort`

```
void sort();
template<class Pred>
void sort(Pred pr);
```

这两个成员函数都会按下面所描述的谓词来对被控序列中的元素排序。

假设迭代器`Pi`指向位于位置i处的元素，`Pj`指向位于位置j处的元素，当`i < j`时，第一个成员函数要求`(*Pj < *Pi)`为true。（即元素是以升序排序

的。) 模板成员函数要求当 $i < j$ 时, $\text{!pr}(*Pj, *Pi)$ 为true。最初被控序列中排序好的元素对在结果序列中的位置关系不会改变(即排序是稳定的)。

仅有在pr抛出异常时这两个成员函数才会有异常发生。在这种情况下, 被控序列将处于一个未指定的排序状态下, 而异常则继续向外抛出。

口 list::splice

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
            iterator last);
```

第一个成员函数把由x控制的序列插入到被控序列中由it指定的元素前面。它也会删除x中的所有元素。($\&x$ 不能等于this。)

第二个成员函数删除x所控制的序列中由first指定的元素并把它插入到被控序列中由it指定的元素前面。(如果 $it == first \parallel it == ++first$, 那么什么都没有改变。)

第三个成员函数将x所控制的序列中由[first, last)指定的子区间插入到被控序列中由it所指定的元素前面, 同样, 最初子区间中的所有元素也会被删除。(如果 $\&x == this$, 则区间[first, last)里面不能有由it指定的元素。)

如果第三个成员函数插入了N个元素, 并且 $\&x != this$, 那么类iterator的对象会递增了N次。对所有的splice成员函数来说, 如果`get_allocator() == x.①get_allocator()`, 那么就不会有任何异常抛出。此外, 在本实现中, 在每插入一个元素时, 都会有一次复制和一次析构函数调用发生。

在上述所有情况中, 只有指向接入元素的迭代器和引用才会变为无效。

口 list::swap

```
void swap(list& x);
```

该成员函数会在`*this`和`x`之间相互交换被控序列。如果`get_allocator() == x.get_allocator()`, 它将可以在常数时间内完成交换并且不会抛出任何异常, 另外, 它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则, 它调用元素的构造函数以及为元素赋值的次数与这两个被控序列的元素个数成比例。

口 list::unique

```
void unique();
template<class Pred>
void unique(Pred pr);
```

第一个成员函数删除被控序列中所有与其前面的元素相等的元素。

^① 此处原文为str。——译者注

假设迭代器Pi指向位于位置i处的元素，Pj指向位于位置j处的元素，则第二个成员函数会删除所有满足 $i + 1 == j \&& pr(*Pi, *Pj)$ 的元素。

对于长度为N(N大于0)的被控序列来说，谓词pr(*Pi, *Pj)被计算N-1次。

只有在pr抛出异常时，该成员函数才会有异常发生。在这种情况下，被控序列将处于一个未指定的状态，而异常则继续向外抛出。

口 list::value_type

```
typedef typename A::value_type value_type;
```

该类型是模板参数T的同义词。

口 operator!=

```
template<class T, class A>
bool operator!=(
    const list <T, A>& lhs,
    const list <T, A>& rhs);
```

该模板函数返回!(lhs == rhs)。

口 operator==

```
template<class T, class A>
bool operator==((
    const list <T, A>& lhs,
    const list <T, A>& rhs);
```

该模板函数重载operator==来比较两个模板类list的对象。该函数返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())。

口 operator<

```
template<class T, class A>
bool operator<(
    const list <T, A>& lhs,
    const list <T, A>& rhs);
```

该模板函数重载operator<来比较两个模板类list的对象。该函数返回lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

口 operator<=

```
template<class T, class A>
bool operator<=(
    const list <T, A>& lhs,
    const list <T, A>& rhs);
```

该模板函数返回!(rhs < lhs)。

口 operator>

```
template<class T, class A>
bool operator>(
    const list <T, A>& lhs,
```

```
const list <T, A>& rhs);
```

该模板函数返回rhs < lhs。

口 operator>=

```
template<class T, class A>
bool operator>=(  
    const list <T, A>& lhs,  
    const list <T, A>& rhs);
```

该模板函数返回!(lhs < rhs)。

口 swap

```
template<class T, class A>
void swap(  
    list <T, A>& lhs,  
    list <T, A>& rhs);
```

该模板函数执行lhs.swap(rhs)。

使用 <list>

list

如果想在程序中使用模板类list，请把头文件<list>包含到程序中。我们可以特化list来存储类型为T的元素，只需这样写一条类型定义语句：

```
typedef list<T, allocator<T> > Mycont;
```

通过使用默认模板参数，我们就可以省略第二个模板参数。

模板类list支持所有我们在第9章中所给出的、对于容器来说是常见的一些操作。（参见第9章的“使用容器”一节。）我们在此仅概括出模板类list所特有的属性。

构造函数

为了构造一个类list<T, A>的对象，可以这么写：

- list(), 声明一个空列表；
- list(al), 也是声明一个空列表，但它还存储一个分配器对象al；
- list(n), 声明一个有n个元素的列表，每个元素都是由其默认构造函数T()构造出来的；
- list(n, val), 声明一个有n个元素的列表，每个元素都是由其复制构造函数T(val)得来的；
- list(n, val, al), 声明一个和上面一样的列表，但它还存储分配器对象al；
- list(first, last), 声明一个列表，其元素的初始值来源于由区间[first, last)所指定的序列中的元素；
- list(first, last, al), 声明一个和上面一样的列表，但它还存储分配器对象al。

如果我们已经为类型为allocator<T>的分配器特化了模板类（这是我

们常做的也是STL提供的默认值），那么再显式地指定分配器参数`al`并不能给我们带来任何好处。仅对程序中显式定义的某些分配器，这样的参数才会起作用（参见第4章中对分配器的讨论）。

下面所有的描述中都假定`cont`是类`list<T, A>`的一个对象。

resize 为了将被控序列的长度改为只容纳`n`个元素，请调用`cont.resize(n)`。超出的元素将会被删除。如果序列需要扩展，那么值为`T()`的元素会被插入到序列末端，直到填满整个序列。我们也可以调用`cont.resize(n, val)`来插入值为`val`的元素。为了删除所有的元素，请调用`cont.clear()`。

front 为了存取被控序列的第一个元素，请调用`cont.front()`。
back 为了存取被控序列的最后一个元素，请调用`cont.back()`。如果`cont`不是一个常量对象，调用这两个成员函数得到的表达式将会是一个左值，于是我们就可以改变原来存储于该元素内的值，我们只需这么写：`cont.front() = T()`。然而，如果序列为空，上述表达式的行为将是未定义的。

push_back 为了向对象末端插入值为`x`的元素，请调用`cont.push_back(x)`；为了在对象开始处插入，请调用`cont.push_front(x)`。为了删除最后一个元素，请调用`cont.pop_back()`，为了删除第一个元素，请调用`cont.pop_front()`。当然，在上述所有的情况下，序列必须不为空，否则调用将有未定义的行为。

assign 为了将被控序列替换成由`[first, last)`所指定的序列，请调用`cont.assign(first, last)`。`[first, last)`不能为最初被控序列中的一部分。为了将被控序列替换成`n`个值为`x`的元素，请调用`cont.assign(n, x)`。

insert 为了在由迭代器`it`指定的元素前插入一个值为`x`的元素，请调用`cont.insert(it, x)`。它的返回值是一个迭代器，指向刚插入的那个元素。为了在由迭代器`it`所指定的元素前插入一个由`[first, last)`指定的序列，请调用`cont.insert(it, first, last)`。序列`[first, last)`不能为最初被控序列中的一部分。为了插入`n`个值为`x`的元素，请调用`cont.insert(it, n, x)`。

erase 为了删除由`it`所指定的元素，请调用`cont.erase(it)`。它的返回值为一个迭代器，指向紧接着被删除元素的下一个元素。为了删除由区间`[first, last)`所指定的所有元素，请调用`cont.erase(first, last)`。

我们还可以在List对象上执行一些能够从它的独一无二的表达方式中获益的操作。

splice 我们可以将一系列的列表节点接入到一个列表中。被接入的节点将会从它原来的位置删除。在接入操作中并没有元素复制——接入只是根据需要简单地改写了节点中的链接。

- 为了把对象`cont2`中所有的内容都接入到由迭代器`it`指定的元素前面，请调用`cont.splice(it, cont2)`。当然，这两个对象必须不同。

- 为了将对象`cont2`中由迭代器`p`所指定的节点接入到由迭代器`it`所指定的节点前面，请调用`cont.splice(it, cont2, p)`。这两个列表对象没有必要一定要不同。（将一个节点接入到自身前面不会导致任何变化。）

- 为了将对象cont2中由[first, last)所指定的序列中的节点都接入到由迭代器it所指定的元素前面, 请调用cont.splice(it, cont2, first, last)。这两个list对象没有必要一定要不同, 但是it不能指定被接入节点中的一员。

如果作为接入的结果, 一个节点从一个列表转移到另一个列表中, 那么很重要的一点就是: 这两个列表所拥有的分配器对象必须相等。对于默认的分配器来说, 这个要求很容易达到。否则, 该节点就不能在新的列表中安全地删除。

remove
remove_if

为了删除所有值等于v(使用operator==去判断)的元素, 请调用cont.remove(v)。为了删除所有使得pr(x)为true的元素x, 请调用cont.remove_if(pr)。

unique

为了删除所有元素都相等(以operator==判断)的子序列中除第一个以外的所有元素, 请调用cont.unique()。为了将operator==替换为pr并以此来作为判断函数, 请调用cont.unique(pr)。注意, 如果我们先对序列进行排序, 使得所有相等元素形成的群组都是相邻的, 执行unique的效率就会变得高一些。

sort

为了对cont所控制的序列进行排序, 请调用cont.sort()。结果序列是按operator<排序的。(元素按升序存储。)排序操作中用到了接入操作。为了用pr替换operator<来作为排序函数, 请调用cont.sort(pr)。

merge

为了将由cont2所控制的有序序列合并到由cont所控制的有序序列中, 请调用cont.merge(cont2)。合并操作中用到了接入操作, 于是cont2在合并后就为空了(除非它是和cont相同的对象)。参与合并的两个序列都必须是按operator<排序的序列, 以使得合并正常进行。(如果不是的话, 按照上面所讲, 对它们进行排序, 然后再做合并。)为了用pr替换operator<来作为排序函数, 请调用cont.merge(cont2, pr)。

reverse

最后, 我们还可以通过调用reverse()来反转整个被控序列。该操作中也用到了接入操作。

实现 <list>

list

程序清单11-1列出了文件list。它定义了模板类list, 以及一些以列表为操作数的模板函数。

```
程序清单11-1: // list standard header
list
#ifndef LIST_
#define LIST_
#include <functional>
#include <memory>
#include <stdexcept>
namespace std {
```

```
// TEMPLATE CLASS List_nod
template<class Ty, class A>
    class List_nod {
protected:
    typedef typename A::template
        rebind<void>::other::pointer Genptr;
    struct Node;
    friend struct Node;
    struct Node {
        Genptr Next, Prev;
        Ty Value;
    };
    List_nod(A A1)
        : Alnod(A1) {}
    typename A::template rebind<Node>::other Alnod;
};

// TEMPLATE CLASS List_ptr
template<class Ty, class A>
    class List_ptr : public List_nod<Ty, A> {
protected:
    typedef typename List_nod<Ty, A>::Node Node;
    typedef typename A::template
        rebind<Node>::other::pointer Nodeptr;
    List_ptr(A A1)
        : List_nod<Ty, A>(A1), Alptr(A1) {}
    typename A::template rebind<Nodeptr>::other Alptr;
};

// TEMPLATE CLASS List_val
template<class Ty, class A>
    class List_val : public List_ptr<Ty, A> {
protected:
    List_val(A A1 = A())
        : List_ptr<Ty, A>(A1), Alval(A1) {}
    typedef typename A::template
        rebind<Ty>::other Alty;
    Alty Alval;
};

// TEMPLATE CLASS list
template<class Ty, class Ax = allocator<Ty> >
    class list : public List_val<Ty, Ax> {
public:
    typedef list<Ty, Ax> Myt;
    typedef List_val<Ty, Ax> Mybase;
```

```
    typedef typename Mybase::Alty A;
protected:
    typedef typename List_nod<Ty, A>::Genptr Genptr;
    typedef typename List_nod<Ty, A>::Node Node;
    typedef typename A::template
        rebind::<Node>::other::pointer Nodeptr;
    struct Acc;
    friend struct Acc;
    struct Acc {
        typedef typename A::template
            rebind::<Nodeptr>::other::reference Nodepref;
        typedef typename A::reference Vref;
        static Nodepref Next(Nodeptr P)
            {return ((Nodepref)(*P).Next); }
        static Nodepref Prev(Nodeptr P)
            {return ((Nodepref)(*P).Prev); }
        static Vref Value(Nodeptr P)
            {return ((Vref)(*P).Value); }
    };
public:
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type Dift;
    typedef Dift difference_type;
    typedef typename A::pointer Tptr;
    typedef typename A::const_pointer Ctptr;
    typedef Tptr pointer;
    typedef Ctptr const_pointer;
    typedef typename A::reference Reft;
    typedef Reft reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;

    // CLASS iterator
    class iterator;
    friend class iterator;
    class iterator : public Bidit<Ty, Dift, Tptr, Reft> {
public:
    typedef Bidit<Ty, Dift, Tptr, Reft> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    iterator()
```

```
    : Ptr(0) {}
iterator(Nodeptr P)
    : Ptr(P) {}
reference operator*() const
    {return (Acc::Value(Ptr)); }
Tptr operator->() const
    {return (&**this); }
iterator& operator++()
    {Ptr = Acc::Next(Ptr);
     return (*this); }
iterator operator++(int)
    {iterator Tmp = *this;
     ++*this;
     return (Tmp); }
iterator& operator--()
    {Ptr = Acc::Prev(Ptr);
     return (*this); }
iterator operator--(int)
    {iterator Tmp = *this;
     --*this;
     return (Tmp); }
bool operator==(const iterator& X) const
    {return (Ptr == X.Ptr); }
bool operator!=(const iterator& X) const
    {return (!(*this == X)); }
Nodeptr Mynode() const
    {return (Ptr); }

protected:
    Nodeptr Ptr;
};

// CLASS const_iterator
class const_iterator;
friend class const_iterator;
class const_iterator
    : public Bidit<Ty, Dift, Ctptr, const_reference> {
public:
    typedef Bidit<Ty, Dift, Ctptr, const_reference>
        Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
```

```
typedef typename Mybase::reference reference;
const_iterator()
    : Ptr(0) {}
const_iterator(Nodeptr P)
    : Ptr(P) {}
const_iterator(const typename list<Ty,Ax>::iterator& X)
    : Ptr(X.Mynode()) {}
const_reference operator*() const
    {return (Acc::Value(Ptr)); }
Ctptr operator->() const
    {return (&**this); }
const_iterator& operator++()
    {Ptr = Acc::Next(Ptr);
     return (*this); }
const_iterator operator++(int)
    {const_iterator Tmp = *this;
     ++*this;
     return (Tmp); }
const_iterator& operator--()
    {Ptr = Acc::Prev(Ptr);
     return (*this); }
const_iterator operator--(int)
    {const_iterator Tmp = *this;
     --*this;
     return (Tmp); }
bool operator==(const const_iterator& X) const
    {return (Ptr == X.Ptr); }
bool operator!=(const const_iterator& X) const
    {return (!(*this == X)); }
Nodeptr Mynode() const
    {return (Ptr); }
protected:
    Nodeptr Ptr;
};

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

list()
    : Mybase(), Head(Buynode()), Size(0)
    {}
explicit list(const A& A1)
    : Mybase(A1), Head(Buynode()), Size(0)
```

```

    {}
explicit list(size_type N)
    : Mybase(), Head(Buynode()), Size(0)
    {insert(begin(), N, Ty()); }
list(size_type N, const Ty& V)
    : Mybase(), Head(Buynode()), Size(0)
    {insert(begin(), N, V); }
list(size_type N, const Ty& V, const A& A1)
    : Mybase(A1), Head(Buynode()), Size(0)
    {insert(begin(), N, V); }
list(const Myt& X)
    : Mybase(X.Alval),
      Head(Buynode()), Size(0)
    {insert(begin(), X.begin(), X.end()); }
template<class It>
    list(It F, It L)
        : Mybase(), Head(Buynode()), Size(0)
        {Construct(F, L, Iter_cat(F)); }
template<class It>
    list(It F, It L, const A& A1)
        : Mybase(A1), Head(Buynode()), Size(0)
        {Construct(F, L, Iter_cat(F)); }
template<class It>
    void Construct(It F, It L, Tnt_iterator_tag)
        {insert(begin(), (size_type)F, (Ty)L); }
template<class It>
    void Construct(It F, It L, input_iterator_tag)
        {insert(begin(), F, L); }
~list()
    {erase(begin(), end());
     Freenode(Head);
     Head = 0, Size = 0; }
Myt& operator=(const Myt& X)
    {if (this != &X)
       assign(X.begin(), X.end());
     return (*this); }
iterator begin()
    {return (iterator(Head == 0 ? 0
                      : Acc::Next(Head))); }
const_iterator begin() const
    {return (const_iterator(Head == 0 ? 0
                                : Acc::Next(Head))); }
iterator end()
    {return (iterator(Head)); }
const_iterator end() const

```

```
        {return (const_iterator(Head)); }
reverse_iterator rbegin()
        {return (reverse_iterator(end())); }
const_reverse_iterator rbegin() const
        {return (const_reverse_iterator(end())); }
reverse_iterator rend()
        {return (reverse_iterator(begin())); }
const_reverse_iterator rend() const
        {return (const_reverse_iterator(begin())); }
void resize(size_type N)
        {resize(N, Ty()); }
void resize(size_type N, Ty X)
        {if (size() < N)
            insert(end(), N - size(), X);
        else
            while (N < size())
                pop_back(); }
size_type size() const
        {return (Size); }
size_type max_size() const
        {return (Mybase::Alval.max_size()); }
bool empty() const
        {return (size() == 0); }
allocator_type get_allocator() const
        {return (Mybase::Alval); }
reference front()
        {return (*begin()); }
const_reference front() const
        {return (*begin()); }
reference back()
        {return (*(--end())); }
const_reference back() const
        {return (*(--end())); }
void push_front(const Ty& X)
        {Insert(begin(), X); }
void pop_front()
        {erase(begin()); }
void push_back(const Ty& X)
        {Insert(end(), X); }
void pop_back()
        {erase(--end()); }
template<class It>
void assign(It F, It L)
        {Assign(F, L, Iter_cat(F)); }
template<class It>
```

```
    void Assign(It F, It L, Int_iterator_tag)
        {assign((size_type)F, (Ty)L); }
template<class It>
    void Assign(It F, It L, input_iterator_tag)
        {erase(begin(), end());
         insert(begin(), F, L); }
void assign(size_type N, const Ty& X)
    {Ty Tx = X;
     erase(begin(), end());
     insert(begin(), N, Tx); }
iterator insert(iterator P, const Ty& X)
    {Insert(P, X);
     return (--P); }
void Insert(iterator P, const Ty& X)
    {Nodeptr S = P.Mynode();
     Nodeptr Snew = Buynode(S, Acc::Prev(S));
     Incsize(1);
     try {
         Mybase::Alval.construct(&Acc::Value(Snew), X);
     } catch (...) {
         --Size;
         Freenode(Snew);
         throw;
     }
     Acc::Prev(S) = Snew;
     Acc::Next(Acc::Prev(Snew)) = Snew; }
void insert(iterator P, size_type M, const Ty& X)
    {size_type N = M;
     try {
         for (; 0 < M; --M)
             Insert(P, X);
     } catch (...) {
         for (; M < N; ++M)
             {iterator Pm = P;
              erase(--Pm); }
         throw;
     }}
template<class It>
    void insert(iterator P, It F, It L)
        {Insert(P, F, L, Iter_cat(F)); }
template<class It>
    void Insert(iterator P, It F, It L,
                Int_iterator_tag)
        {insert(P, (size_type)F, (Ty)L); }
template<class It>
```

```
void insert(iterator P, It F, It L,
           input_iterator_tag)
{size_type N = 0;
try {
    for (; F != L; ++F, ++N)
        Insert(P, *F);
} catch (...) {
    for (; 0 < N; --N)
        {iterator Pm = F;
         erase(--Pm); }
    throw;
}}
template<class It>
void Insert(iterator P, It F, It L,
            forward_iterator_tag)
{It Fs = F;
try {
    for (; F != L; ++F)
        Insert(P, *F);
} catch (...) {
    for (; Fs != F; ++Fs)
        {iterator Pm = P;
         erase(--Pm); }
    throw;
}}
iterator erase(iterator P)
{Nodeptr S = (P++) .Mynode();
Acc::Next(Acc::Prev(S)) = Acc::Next(S);
Acc::Prev(Acc::Next(S)) = Acc::Prev(S);
Mybase::Alval.destroy(&Acc::Value(S));
Freenode(S);
--Size;
return (P); }
iterator erase(iterator F, iterator L)
{while (F != L)
    erase(F++);
return (F); }
void clear()
{erase(begin(), end()); }
void swap(Mylist& X)
{if (Mybase::Alval == X.Alval)
    {std::swap(Head, X.Head);
     std::swap(Size, X.Size); }
else
    {iterator P = begin();
```

```

        splice(P, X);
        X.splice(X.begin(), *this, P, end()); }
void splice(iterator P, Myt& X)
    {if (this != &X && !X.empty())
     {Splice(P, X, X.begin(), X.end(), X.Size); }}
void splice(iterator P, Myt& X, iterator F)
    {iterator L = F;
     if (F != X.end() && P != F && P != ++L)
         {Splice(P, X, F, L, 1); }}
void splice(iterator P, Myt& X, iterator F, iterator L)
    {if (F != L && P != L)
     {size_type N = 0;
      for (iterator Fs = F; Fs != L; ++Fs, ++N)
          if (Fs == P)
              return; // else granny knot
      Splice(P, X, F, L, N); }}
void remove(const Ty& V)
    {iterator L = end();
     for (iterator F = begin(); F != L; )
         if (*F == V)
             erase(F++);
         else
             ++F; }
template<class Pr1>
void remove_if(Pr1 Pr)
    {iterator L = end();
     for (iterator F = begin(); F != L; )
         if (Pr(*F))
             erase(F++);
         else
             ++F; }
void unique()
    {iterator F = begin(), L = end();
     if (F != L)
         for (iterator M = F; ++M != L; M = F)
             if (*F == *M)
                 erase(M);
             else
                 F = M; }
template<class Pr2>
void unique(Pr2 Pr)
    {iterator F = begin(), L = end();
     if (F != L)
         for (iterator M = F; ++M != L; M = F)
             if (Pr(*F, *M))

```

```
        erase(M);
    else
        F = M; }
void merge(Myt& X)
{if (&X != this)
    {iterator F1 = begin(), L1 = end();
 iterator F2 = X.begin(), L2 = X.end();
 while (F1 != L1 && F2 != L2)
     if (*F2 < *F1)
         {iterator Mid2 = F2;
          Splice(F1, X, F2, ++Mid2, 1);
          F2 = Mid2; }
     else
         ++F1;
     if (F2 != L2)
 Splice(L1, X, F2, L2, X.Size); }}
template<class Pr3>
void merge(Myt& X, Pr3 Pr)
{if (&X != this)
    {iterator F1 = begin(), L1 = end();
 iterator F2 = X.begin(), L2 = X.end();
 while (F1 != L1 && F2 != L2)
     if (Pr(*F2, *F1))
         {iterator Mid2 = F2;
          Splice(F1, X, F2, ++Mid2, 1);
          F2 = Mid2; }
     else
         ++F1;
     if (F2 != L2)
 Splice(L1, X, F2, L2, X.Size); }}
void sort()
{if (2 <= size())
    {const size_t MAXN = 25;
 Myt X(Mybase::Alval), Arr[MAXN + 1];
 size_t N = 0;
 while (!empty())
     {X.splice(X.begin(), *this, begin());
      size_t I;
      for (I = 0; I < N && !Arr[I].empty(); ++I)
          {Arr[I].merge(X);
           Arr[I].swap(X); }
      if (I == MAXN)
          Arr[I - 1].merge(X);
      else
          {Arr[I].swap(X); }}
```

```

        if (I == N)
            ++N; })
    for (size_t I = 1; I < N; ++I)
        Arr[I].merge(Arr[I - 1]);
    swap(Arr[N - 1]); })
template<class Pr3>
void sort(Pr3 Pr)
{if (2 <= size())
{const size_t MAXN = 25;
Myt X(Mybase::Alval), Arr[MAXN + 1];
size_t N = 0;
while (!empty())
    {X.splice(X.begin(), *this, begin());
     size_t I;
     for (I = 0; I < N && !Arr[I].empty(); ++I)
         {Arr[I].merge(X, Pr);
          Arr[I].swap(X); }
     if (I == MAXN)
         Arr[I - 1].merge(X, Pr);
     else
         {Arr[I].swap(X);
          if (I == N)
              ++N; })
     for (size_t I = 1; I < N; ++I)
         Arr[i].merge(Arr[I - 1], Pr);
     swap(Arr[N - 1]); })
void reverse()
{if (2 <= size())
{iterator L = end();
 for (iterator F = ++begin(); F != L; )
     {iterator M = F;
      Splice(begin(), *this, M, ++F, 1); }})
protected:
Nodeptr Buynode(Nodeptr Narg = 0, Nodeptr Parg = 0)
{Nodeptr S = Alnod.allocate(1, (void *)0);
Alptr.construct(&Acc::Next(S),
Narg != 0 ? Narg : S);
Alptr.construct(&Acc::Prev(S),
Parg != 0 ? Parg : S);
return (S); }
void Freenode(Nodeptr S)
{Alptr.destroy(&Acc::Next(S));
Alptr.destroy(&Acc::Prev(S));
Alnod.deallocate(S, 1); }
void Splice(iterator P, Myt& X, iterator F,iterator L,

```

```
size_type N)
{if (Mybase::Alval == X.Alval)
{if (this != &X)
{Incsize(N);
X.Size -= N; }
Acc::Next(Acc::Prev(F.Mynode())) =
L.Mynode();
Acc::Next(Acc::Prev(L.Mynode())) =
P.Mynode();
Acc::Next(Acc::Prev(P.Mynode())) =
vF.Mynode();
Nodeptr S = Acc::Prev(P.Mynode());
Acc::Prev(P.Mynode()) =
Acc::Prev(L.Mynode());
Acc::Prev(L.Mynode()) =
Acc::Prev(F.Mynode());
Acc::Prev(F.Mynode()) = S; }
else
{insert(P, F, L);
X.erase(F, L); }
void Incsize(size_type N)
{if (max_size() - size() < N)
throw length_error("list<T> too long");
Size += N; }
Nodeptr Head;
size_type Size;
};

// list TEMPLATE OPERATORS
template<class Ty, class A> inline
void swap(list<Ty, A>& X, list<Ty, A>& Y)
(X.swap(Y); }

template<class Ty, class A> inline
bool operator==(const list<Ty, A>& X,
const list<Ty, A>& Y)
{return (X.size() == Y.size()
&& equal(X.begin(), X.end(), Y.begin())); }
template<class Ty, class A> inline
bool operator!=(const list<Ty, A>& X,
const list<Ty, A>& Y)
{return (!(X == Y)); }
template<class Ty, class A> inline
bool operator<(const list<Ty, A>& X,
const list<Ty, A>& Y)
```

```

    {return (lexicographical_compare(X.begin(), X.end(),
        Y.begin(), Y.end())); }
template<class Ty, class A> inline
    bool operator>(const list<Ty, A>& X,
        const list<Ty, A>& Y)
    {return (Y < X); }
template<class Ty, class A> inline
    bool operator<=(const list<Ty, A>& X,
        const list<Ty, A>& Y)
    {return (!(Y < X)); }
template<class Ty, class A> inline
    bool operator>=(const list<Ty, A>& X,
        const list<Ty, A>& Y)
    {return (!(X < Y)); }
} /* namespace std */
#endif /* LIST_ */

```

一个list对象中存储一个指针和一个计数器来表示它所控制的序列。如下面所描述的，除了分配器对象外，list中还存储了两个对象，它们分别为：

Head

- Head 是一个指针，指向一个虚构的“头”节点。向前移动就指向被控序列的开始处，向后则指向其末端。

Size

- Size 记录了列表中元素的个数。

虚构的头节点极大地简化了许多有关列表的操作。它去除了列表中对于第一个和最后一个元素的特殊处理。我们为头节点付出的代价只是列表中一个没有被使用的元素空间而已。这种开销通常都很小，但对于一个元素非常大的列表来说，它也可能变得很可观。

对于那些曾经自己管理过列表的人来说，这很熟悉，也很简单。链表对象也存储了三个不同的分配器对象。下面就是我们所要讲述的真正技巧。

在STL容器中，所有被控制的存储空间名义上都是由我们在构造容器对象时所指定的分配器对象管理的。我们在前面的章节中也提到过，在和vector对象协同工作时，分配器对象被发明出来了，它最初用来分配及释放某些类型为T的元素所组成的数组。从那以后，它们开始变得越来越值得炫耀，并且也越来越复杂。模板类vector可以只使用其中最简单的用法，但对于模板类list来说这样做就不行了，下面就是一些微妙的原因。

Node

在一开始处，考虑一下我们通常是如何管理一个双向链表的。我们需要定义一个类Node，它存储了一个列表节点所需的所有数据。为了存储一个类型为T的元素，我们可以这么写：

```
class Node {
```

```
    Node *Next, *Prev;  
    T Value;  
};
```

这个小技巧使我们又想起最初的那段使用C的时光（C++也是基于C的基础发展而来的）。前向指针 `Next` 和后向指针 `Prev` 都是自引用（self-referential）指针——它们所指向的和它们所处的对象类型一致的对象。不用怕，我们可以在结构化类型中定义不完整类型的指针，即使这个类型是我们当前正准备完成的。

但是，分配器出现了问题。第一个问题就是构造类型为 `list<T, allocator<T>>` 的链表对象时使用的分配器对象不能完成整个工作。我们所希望的不是分配类型为 `T` 的对象（这个 `allocator<T>` 对象早就知道该如何分配，但是由于它知道如何构造和销毁一个这样的对象，所以我们还是需要它）。实际上，我们需要分配类型为 `Node` 的对象。这意味着我们需要一个类型为 `allocator<Node>` 的分配器对象，并且我们还期望以一种明显的方式把它和构造 `list` 对象时使用的 `allocator<T>` 对象联系起来。例如，该分配器可能会从一个我们所希望使用的私有存储池中分配对象。

rebind

所有的分配器类型都提供了两个技巧，给了我们想要的能力。第一个是成员模板类 `rebind`——下面这个奇怪的公式 `A::rebind<Node>::other` 是命名类型 `allocator<Node>` 的一种方式（此处 `A` 是 `allocator<T>` 的同义词）。即使在得到了我们所希望得到的分配器对象的名称后，我们仍然需要从原有的分配器对象中创建出该类型的对象。为此，所有的分配器类型都提供了一个模板构造函数。对于默认的模板类 `allocator` 来说，该构造函数看起来如下所示：

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

然后，我们就可以由一个 `allocator<T>` 对象构造出一个 `allocator<Node>` 对象。对于那些比模板类 `allocator` 更复杂的分配器来说，该构造函数必须足够智能，能够从任意的指针中复制对象到私有存储空间或是我们所指定的空间中去。

智能指针

分配器还会导致另一个问题。它们保留着将自己所分配的对象存储到不适当位置的权利。更确切地说，假设一个分配器类型 `A` 定义了类型 `A::pointer`，我们就不得不使用它来描述指向被分配对象的任意“指针”。我们在此处使用引号是因为在从 C 继承而来的老式 C++ 中，该类型可以不是一个真正的指针。（参见第 4 章对智能指针的讨论。）如果 `p` 的类型为 `A::pointer`，它保证 `*p` 是一个指向被分配对象的左值。（我们可以使用 `*p` 对该对象进行存取或赋值。）除此之外，再没有别的了。

这个无力的保证导致了在声明一个类Node时会出现实际问题。我们希望这么写：

```
class Node {
    A::rebind<Node>::pointer Next, Prev;
    T Value;
};
```

但是这样做却是行不通的。分配器模板只可以为一个完整类型所特化，而类型Node在作为类型定义结束的大括号出现前却是不完整的。因此我们在它被完整定义前需要描述它存储的指针。那么，我们该如何做呢？

void指针

当出现了这样的循环依赖时，在C中通常都是引入泛型的、或者“void类型的”指针来解决这个问题，如：

```
class Node {
    void *Next, *Prev;
    T Value;
};
```

一个void指针用来存储我们可以在C中声明的任意类型的对象指针。使用这种方法，我们丧失了一部分的类型检测能力，并且还时不时需要在使用这种指针前进行一些类型转换，但这种方法确实解决了问题。

然而，对于由分配器提供的指针类型，C++标准就讲得没那么确切了。它只是建议A::pointer可以是任意的模板类，只要它符合我们前面给出的限制条件就可以了。但它并没有强制要求这样的模板类类型定义必须和一个void指针相等。这由实现去完成。

本实现假设任意A::pointer都可以和任意A::rebind<void>::pointer相互转化。从另一方面讲，类型A::rebind<void>::pointer为所有的A<T>::pointer提供了一个泛型指针类型（前提是所有的A<T>::pointer都有着同样的表达方式）。不管是谁写的分配器模板，都必须无条件地为类型void提供一个显式特化版本。对于“保证该显式特化版本会提供一个有足够的弹性的指针”则没有太多的要求。

空指针

本实现也假设整数0可以作为一个空指针(null pointer)，不管该指针由分配器定义得多么奇怪。值得一提的是，我们可以把0赋值给一个泛型指针的对象，得到的结果和指向任意被分配对象的泛型指针相比不会相等。我们也可以使用operator==来判断一个泛型指针对象是否等于0：仅在该泛型指针对象所存储的值是一个空指针时，结果才会为true。（在前面的章节中我们使用过这样的假设，如关于成员对象First的使用中就用过这种假设。）

智能指针引入了最后一个技巧。与从C中继承得到的标量指针(scalar pointer)不同的是，本实现假设它们应该有一个非平凡构造函数和析构函数。为了严格地按照规则行事，我们需要一个分配器对象来协助我们完成这些任务（虽然我们很难想像出会需要什么样的技巧）。于是，一个list对象使用如下三种分配器对象：

Alnod

- Alnod 是为节点类型所服务的，它可以分配和释放节点。

Alptr

- Alptr 是为节点指针类型所服务的，用来创建和销毁存储于节点中的链接。

Alty

- Alty 是为元素类型所服务的，用于创建和销毁存储于节点中的元素。

原则上说，我们只需要存储这三个分配器对象中的一个。其他两个可以根据需要实时创建，如：

```
A::rebind<Node>::other(Alty).destroy(p);
```

会创建一个类似于Alnod的临时分配器，它可以存活足够长以销毁节点的指针p。编译器可能会把实际产生一个临时对象的操作给优化掉，最起码对于默认的分配对象是如此。从另一方面来说，我们知道，一个典型的分配器的存储空间是可以被优化掉的。（参见第10章对于大小为零的分配器对象的讨论。）在本实现中，我们采取了比较安全的那种假设，即往一个list对象中存储三个分配器对象并没有造成空间和时间上的开销。

在给出漫长的导言后，我们现在可以更加聪明地学习这些代码了。文件list揭示了特化版本list<T, T>实际上是从下面三个基类中的一个派生而来的：

List_nod

- List_nod<T, A>定义泛型指针类型 Genptr 和节点类型 Node。它也存储分配器对象 Alnod。

List_ptr

- List_ptr<T, A>定义节点指针类型 Nodeptr。它也存储分配器对象 Alptr。

List_val

- List_val<T, A>存储分配器对象 Alnod。

一个足够智能的编译器会知道：对这三个基类来说，不要在list<T, A>对象中为它们分配任何存储空间。

在成员结构体Acc中还封装着更多的复杂性。它提供了一些很方便的函数，用以存取保存在列表节点中的对象。也就是说，表达式Acc::Next(Ptr)允许我们存取由Ptr所指定的节点中的后向指针Next。为了使这个表达式为左值，函数Acc::Next必须返回一个指向被存储对象的引用。对于分配器在定义引用类型中有着多大的自由度，仍然有很多不同的意见。有些人认为Alloc<T>::reference始终应该是T&的同义词。但为防万一有人提供一种更聪明的分配器，这些函数都统一使用由分配器定义的引用类型。

`list`中定义了非平凡成员类`iterator`和`const_iterator`。它们甚至比模板类`Ptrit`还要简单（`vector`用`Ptrit`来定义它的迭代器）——`list`只提供双向迭代器，而不是像`vector`那样提供随机存取迭代器。这两个容器的迭代器都只是存储了一个简单的指针。

`Buynode`
`Freenode`
`Incsize`

有几个保护型的成员函数实现了一些基本的操作。调用`Buynode(next, prev)`会分配一个节点并适当地初始化它的两个指针成员对象。调用`Freenode(p)`会释放一个节点。这两个函数都假设存在另外一个代理，由它来负责在必要时构造和销毁存储的元素。调用`Incsize(n)`会增加被控序列的长度。它会检测那些使得链表增长太大的不可靠行为。

`Splice`

调用`splice(p, cont2, first, last, n)`会把列表对象`cont2`中由`[first, last)`所指定的`n`个节点接入到`p`前面。它假设调用方已经就可能造成问题的重叠现象做过检测。但是它并没有检测在两个拥有不兼容分配器对象的容器之间进行的接合，在这种情况下，它会复制（并且删除）那些被接入的节点。在此给出的连接顺序的改变方法非常精巧，对于不同的情况它就会有所变化，以此达到最好的效率。

其他容器和`list`都有的成员函数没有引入什么有新意的地方。我们在讨论模板类`vector`时已经描述过大部分的机制。在此可以找到的最大发现就是当程序员提供的代码抛出异常时容器所拥有的恢复能力。让我再次重申，所有容器中只有`list`保证在异常出现时回退到当前操作前的状态。

`splice`

`splice`的三个版本都会在适当的检测后，把实际工作交给`Splice`和`Incsize`去做。将一系列节点接入到该序列中某个位置的尝试是值得特殊考虑的。C++标准只是简单地说：这样的操作是未定义的。而且，它还要求不管子区间中有多少元素，在同一容器中对该子区间的接入要在常数时间内完成。这个要求没有给我们留下任何空间来对避免在列表中打结的要求做检测。

本实现在检测这方面背离了C++标准。如果接合导致了打结，被控序列将不发生改变。

`sort`

`sort`的两种形式以同样的方式工作。它们对一个由临时`list`对象形成的数组执行合并。该数组中的每个元素存储一个列表，它可以在它被合并到下一个大一点的列表中前，增长为它前面的那个列表的两倍。最后那个元素比较特别——它存储了一个任意长度的列表。但是数组的大小被（任意地）设置为有25个元素，其中还包括最后的那个溢出列表。这样，在它偏离简单的倍增算法前，`sort`最多可以对32 000 000个元素进行排序。最后一步就是把所有的临时列表都合并回原来的容器中（此时容器已经空了）。

测试 <list>

tlist.c

程序清单11-2列出了文件tlist.c。它是三个看起来很相像的测试程序中的一个，另外两个是文件tvector.c和文件tdeque.c。为了简单地比较这三个测试程序，我们只是简单地注释掉那些和给定容器不一致的测试段，而不是删除这些无用的代码。

程序清单 11-2:

```
// test <list>
#include <assert.h>
#include <iostream>
#include <functional>
#include <list>
using namespace std;

// TEST <list>
int main()
{typedef allocator<char> Myal;

    // TEST list
    typedef list<char, Myal> Mycont;
    char ch, carr[] = "abc";
    Mycont::allocator_type *p_alloc = (Myal *)0;
    Mycont::pointer p_ptr = (char *)0;
    Mycont::const_pointer p_cptr = (const char *)0;
    Mycont::reference p_ref = ch;
    Mycont::const_reference p_cref = (const char&)ch;
    Mycont::size_type *p_size = (size_t *)0;
    Mycont::difference_type *p_diff = (ptrdiff_t *)0;
    Mycont::value_type *p_val = (char *)0;

    Mycont v0;
    Myal al = v0.get_allocator();
    Mycont v0a(al);
    assert(v0.empty() && v0.size() == 0);
    assert(v0a.size() == 0 && v0a.get_allocator() == al);
    Mycont v1(5), v1a(6, 'x'), v1b(7, 'y', al);
    assert(v1.size() == 5 && v1.back() == '\0');
    assert(v1a.size() == 6 && v1a.back() == 'x');
    assert(v1b.size() == 7 && v1b.back() == 'y');
    Mycont v2(v1a);
    assert(v2.size() == 6 && v2.front() == 'x');
    Mycont v3(v1a.begin(), v1a.end());
    assert(v3.size() == 6 && v3.front() == 'x');
    const Mycont v4(v1a.begin(), v1a.end(), al);
```

```
assert(v4.size() == 6 && v4.front() == 'x');
v0 = v4;
assert(v0.size() == 6 && v0.front() == 'x');
// assert(v0[0] == 'x' && v0.at(5) == 'x');

// v0.reserve(12);
// assert(12 <= v0.capacity());
v0.resize(8);
assert(v0.size() == 8 && v0.back() == '\0');
v0.resize(10, 'z');
assert(v0.size() == 10 && v0.back() == 'z');
assert(v0.size() <= v0.max_size());

Mycont::iterator p_it(v0.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v0.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert(*p_it == 'x' && *(p_it = v0.end()) == 'z');
assert(*p_cit == 'x' && *(p_cit = v4.end()) == 'x');
assert(*p_rit == 'z'
       && *(p_rit = v0.rend()) == 'x');
assert(*p_crit == 'x'
       && *(p_crit = v4.rend()) == 'x');

assert(v0.front() == 'x' && v4.front() == 'x');
v0.push_back('a');
assert(v0.back() == 'a');
v0.pop_back();
assert(v0.back() == 'z' && v4.back() == 'x');

v0.push_front('b');
assert(v0.front() == 'b');
v0.pop_front();
assert(v0.front() == 'x');

v0.assign(v4.begin(), v4.end());
assert(v0.size() == v4.size()
      && v0.front() == v4.front());
v0.assign(4, 'w');
assert(v0.size() == 4 && v0.front() == 'w');
assert(*v0.insert(v0.begin(), 'a') == 'a');
assert(v0.front() == 'a'
      && *++v0.begin() == 'w');
v0.insert(v0.begin(), 2, 'b');
assert(v0.front() == 'b'
      && *++v0.begin() == 'b'
      && *++v0.begin() == 'a');
```

```
v0.insert(v0.end(), v4.begin(), v4.end());
assert(v0.back() == v4.back());
v0.insert(v0.end(), carr, carr + 3);
assert(v0.back() == 'c');
v0.erase(v0.begin());
assert(v0.front() == 'b' && *++v0.begin() == 'a');
v0.erase(v0.begin(), ++v0.begin());
assert(v0.front() == 'a');

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && !v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

v0.insert(v0.begin(), carr, carr + 3);
v1.splice(v1.begin(), v0);
assert(v0.empty() && v1.front() == 'a');
v0.splice(v0.end(), v1, v1.begin());
assert(v0.size() == 1 && v0.front() == 'a');
v0.splice(v0.begin(), v1, v1.begin(), v1.end());
assert(v0.front() == 'b' && v1.empty());
v0.remove('b');
assert(v0.front() == 'c');
v0.remove_if(binder2nd<not_equal_to<char> >(
    not_equal_to<char>(), 'c'));
assert(v0.front() == 'c' && v0.size() == 1);

v0.merge(v1, greater<char>());
assert(v0.front() == 'c' && v0.size() == 1);
v0.insert(v0.begin(), carr, carr + 3);
v0.unique();
assert(v0.back() == 'c' && v0.size() == 3);
v0.unique(not_equal_to<char>());
assert(v0.front() == 'a' && v0.size() == 1);
v1.insert(v1.begin(), carr, carr + 3);
v0.merge(v1);
assert(v0.back() == 'c' && v0.size() == 4);
v0.sort(greater<char>());
assert(v0.back() == 'a' && v0.size() == 4);
v0.sort();
assert(v0.back() == 'c' && v0.size() == 4);
v0.reverse();
```

```
assert(v0.back() == 'a' && v0.size() == 4);

cout << "SUCCESS testing <list>" << endl;
return (0); }
```

该测试程序只是对模板类list的一个特化版本进行了简单的测试，按照其意图测试了它所给出的每个成员函数和成员类型。如果测试一切顺利的话，测试程序将打印出：

```
SUCCESS testing <list>
```

然后正常退出。

习题

习题11-1

模板类list的这个实现从不会在节点之间复制元素。元素只会被构造和销毁，但从不会被赋值。在什么情况下我们会需要这样的行为？

习题11-2

为什么allocator只能为一个完整的类型来特化？

习题11-3

重写模板类list，去除头节点的使用。在什么情况下，我们所重写的版本会是一个好的设计？

习题11-4

写出模板类forward_list，它仅在每个节点中存储一个后向指针。与模板类list相比，什么样的操作此时会变得更难（拥有更差的时间复杂度）？对这两个容器来说，相关的节点大小是什么？

习题11-5

改变列表迭代器的定义，使得它可以很容易地判断两个迭代器是否指向不同列表中的元素。

习题11-6

有一种实现散列表(hash table)的方法就是使用一个由列表组成的向量。一个散列函数可以把一个键值映射到向量中的一个索引上面。给定列表中的所有元素分享同一个散列值（即使它们的键并不相同）。（如果散列函数很好的话，典型散列表中的列表一律很短，于是查找给定键的时间也就成了一个常数）。写出实现一个简单散列表的模板类hash_list。

习题11-7

将上面hash_list的实现改为使用一个单独的列表和一个由列表迭代器组成的向量。这两个版本的优缺点各是什么？

习题11-8

[较难] 如何实现每个节点只存储一个指针对象的双向链表？

习题11-9

[特难] 在定义列表节点时，如何去除对泛型指针的需求？

第12章 <deque>

背景知识

<deque>
deque

头文件<deque>中只定义了模板类deque，它是一个容器，它所控制的长度为N的序列是以统一的长度为B的块所构成的。我们可以通过块指针组成的映射数组Map来存取这些块。原则上说，我们可以通过下面的表达式来存取第N个元素：

Map[N / B]{N % B}

也就是说，双队列和向量一样支持在常数时间内存取任意元素。

在第9章的表9-1中列出了模板类deque的附加复杂度所带来的回报。它是除了vector之外惟一一个支持随机存取迭代器的容器。（我们可以从那个表中得到这个信息，如它可以用表达式X[N]来在常数时间内存取一个任意元素。）但在一个很重要的方面它做的比vector好——我们可以在被控序列的开始处以常数时间来插入或删除元素。当然，我们也可以在序列的末端做到这些，但这样它就和vector没有什么区别了。如我们在前面所讨论的，在一个vector的末端添加元素可能为常数时间操作。（参见第10章。）

在此还有个小插曲，对于“deque”是如何发音的有着两种不同的说法。一种把它作为“deck”的同音词来发音。确实，双队列的行为和一副牌差不多——我们可以以同样任意的方式在它的两端增加或删除元素。但另一种则把它发音为“DEE-queue”，它意味着双端队列（Double Ended QUEUE），这种说法有着令人信服的理由。不管怎么说，如何发音随你的便。

为了能够在两端都可以高效地增长，双队列在某个偏移量Off处开始其有效内容。它把所有有效元素的个数（即序列的长度）存储在另一个整数Size中。在将上面表达式中的N替换成N + Off后，我们可得到一个更现实的查找第N个元素的方法。

然而，实际中的双队列还是没那么简单。STL双队列还有着另外的承诺。假如我们没有在被控序列的中间插入或删除元素，那么即使对它的任意一端进行了推入和弹出，原有的迭代器还能够继续指向它原来所指向的元素。这个要求限制了当被控序列增长或缩短时，映射是如何改变的。我们将在本章的后面部分对此进行讨论。

于是，双队列就可以像列表一样很好地用来实现先进先出（first-in first-out, FIFO）队列或后进先出（last-in first-out, LIFO）队列（即我们都知道的栈）。当我们希望频繁地在任意位置插入新元素时，它就没有列表那么好了——双队列所需的时间为线性的，而列表只需要常数时间。然而，如果我们需要对被控序列中的元素进行随机存取，那么双队列就要远胜过列表了。

vector

表9-1还引发了一个有趣的问题，在所有的操作中，双队列的效率比向量的非平即胜，那么为什么我们仍然还在使用向量呢？对于这个问题，有两个答案。一个就是，双队列比向量需要更多存储空间的开销，通常是每个元素多一个指针。这可能很重要，也可能不重要，一切都取决于在每个元素中存储的有用数据的多少和一个给定的应用程序同时使用的元素个数。

还有一个更重要的问题就是每次操作用的开销。在双队列中存取任意一个元素的时间平均来说可能是一个常数，但它相对于在向量中进行随机存取时所需的时间仍然显得太多了。一个给定的应用程序可能不会从双队列所提供的弹性中充分获益，但它仍然需要接受双队列带来的代码空间和执行时间上的额外开销，而这一切都不会发生在能够提供类似操作的向量中。

概括来说，我们可以把双队列看做是一个在向量和列表之间的折衷。如果对于给定应用程序中的大部分常用操作，其时间复杂度在能够接受的范围内，还是请选择那两个简单容器中的一个。如果需要在效率和由此效率引入的复杂度之间维持平衡，那就选择双队列。

功能描述

```
namespace std {
    template<class T, class A>
        class deque;

        // TEMPLATE FUNCTIONS
    template<class T, class A>
        bool operator==(const deque<T, A>& lhs,
                           const deque<T, A>& rhs);
    template<class T, class A>
        bool operator!=(const deque<T, A>& lhs,
                           const deque<T, A>& rhs);
    template<class T, class A>
        bool operator<(const deque<T, A>& lhs,
```

```

        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<=((
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    void swap(
        deque<T, A>& lhs,
        deque<T, A>& rhs);
};

```

包含STL标准头文件<deque>可以到了模板类deque以及几个支持模板的定义。

deque

```

template<class T, class A = allocator<T> >
    class deque {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    deque();
    explicit deque(const A& al);
    explicit deque(size_type n);
    deque(size_type n, const T& v);
    deque(size_type n, const T& v,

```

```
    const A& a1);
    deque(const deque& x);
    template<class InIt>
        deque(InIt first, InIt last);
    template<class InIt>
        deque(InIt first, InIt last, const A& a1);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n);
    void resize(size_type n, T x);
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    reference at(size_type pos);
    const_reference at(size_type pos) const;
    reference operator[](size_type pos);
    const_reference operator[](size_type pos);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
        void assign(size_type n, const T& x);
        iterator insert(iterator it, const T& x);
        void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        void clear();
        void swap(deque& x);
};
```

该模板类描述的对象控制一个元素类型为T的可变长度序列。该序列以允许在任意一端插入和删除元素并且操作只会引起单个元素的复制（常数时间）的方式表示。在序列中间进行这样的操作使得元素复制及赋值操作的次数和该序列中元素的个数成比例（线性时间）。

双队列对象通过存储于其内部的类A的分配器对象来进行存储空间的分配及释放。该分配器对象必须拥有和模板类allocator一样的外部接口。注意，在对容器进行赋值时，被存储的分配器对象并没有被复制。

只有在成员函数必须对被控序列的元素进行插入和删除，双队列对象才会重新分配空间：

- 如果元素被插入到一个空的序列中，或者元素被删除后序列为
空，那么先前由begin()和end()所返回的迭代器将变得无效。
- 如果元素被插入到first()处，那么先前所有指向已有元素的迭代器（注意不是引用）将变得无效。
- 如果元素被插入到end()处，那么end()以及先前所有指向已有元素的迭代器（注意不是引用）将会变得无效。
- 如果位于first()处的元素被删除了，只有先前指向被删除元素的迭代器和引用才会变得无效。
- 如果位于last() - 1处的元素被删除了，只有last()以及先前指向被删除元素的迭代器和引用才会变得无效。
- 除去上面所有情况以外的插入和删除元素的操作将导致所有的迭代器和引用变为无效。

口 deque::allocator_type
 typedef A allocator_type;

该类型是模板参数A的同义词。

口 deque::assign
 template<class InIt>
 void assign(InIt first, InIt last);
 void assign(size_type n, const T& x);

如果 InIt 是一个整数类型，第一个成员函数就相当于 assign((size_type)first, (T)last)。否则，第一个成员函数会把*this所控制的序列替换成序列[first, last)，其中[first, last)不能和*this最初所控制的序列重叠。第二个成员函数将*this所控制的序列替换成一个由n个值为x的元素组成的新序列。

口 deque::at
 const_reference at(size_type pos) const;
 reference at(size_type pos);

该成员函数返回一个引用，它指向被控序列中位置为pos的元素。如果给定的位置是无效的，该函数就向外抛出一个类out_of_range的异常。

■ deque::back

```
reference back();
const_reference back() const;
```

该成员函数返回一个引用，它指向被控序列中最后一个元素（被控序列不能为空）。

■ deque::begin

```
const_iterator begin() const;
iterator begin();
```

该成员函数返回一个随机存取迭代器，它指向被控序列的第一个元素（如果被控序列为空，则指向紧接着序列末端的下一个位置）。

■ deque::clear

```
void clear();
```

该成员函数调用erase(begin(), end())。

■ deque::const_iterator

```
typedef T1 const_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量随机存取迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。

■ deque::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。

■ deque::const_reference

```
typedef typename A::const_reference const_reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。

■ deque::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>
const_reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量反转型随机存取迭代器来使用。

■ deque::deque

```
deque();
explicit deque(const A& a1);
explicit deque(size_type n);
deque(size_type n, const T& v);
deque(size_type n, const T& v,
       const A& a1);
deque(const deque& x);
template<class InIt>
```

```

    deque(Init first, Init last);
    template<class Init>
        deque(Init first, Init last, const A& al);

```

所有的构造函数都会向对象中存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，它就是A()。

开始的两个构造函数指定一个空的初始被控序列。第三个构造函数指定有n个值为T()的元素序列。第四个和第五个构造函数指定n个值为x的元素序列。第六个构造函数指定一个由x控制的序列的拷贝。如果Init是一个整数类型的话，最后两个构造函数指定有(size_type)first个值为(T)last的元素序列；否则，这两个构造函数指定序列[first, last]。

口 deque::difference_type

```
typedef T3 difference_type;
```

该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。

口 deque::empty

```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。

口 deque::end

```
const_iterator end() const;
iterator end();
```

该成员函数返回一个随机存取迭代器，它指向紧接着序列末端的下一个位置。

口 deque::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除被控序列中由[first, last)所指定的区间内的所有元素。这两个函数都返回一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

删除N个元素将导致调用N次析构函数以及从删除点至序列较近的末端^①的每个元素都被赋值一次。在序列的两端删除一个元素仅使得原先指向被删除元素的迭代器和引用变得无效。否则，删除一个元素将导致所有的迭代器和引用都变得无效。

^① 因为双队列所控制的长度为N的序列是以统一的长度为B的块所构成的，所以才有着较近的末端一说。——译者注

口 deque::front

```
reference front();
const_reference front() const;
```

该成员函数返回指向被控序列中第一个元素的引用（被控序列不能为空）。

口 deque::get_allocator

```
A get_allocator() const;
```

该成员函数返回存储的分配器对象。

口 deque::insert

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
void insert(iterator it, InIt first, InIt last);
```

所有这些成员函数都是在被控序列中由it所指定的元素前插入由其他操作数指定的新序列）。第一个成员函数插入一个值为x的单个元素，然后返回一个指向刚被插入的这个元素的迭代器。第二个成员函数插入连续n个值为x的元素。

如果InIt是一个整数类型，那么最后一个成员函数就相当于insert(it, (size_type)first, (T)last)。否则，最后的那个成员函数将向被控序列中插入序列[first, last]（它不能和最初的被控序列有重叠）。

在插入一个单个元素时，被复制的元素个数与插入点到序列较近的末端之间的元素个数成线性比例。在向序列的任意一端插入一个单个元素时，被分摊处理复制的元素个数为一个常数。当插入N个元素时，被复制的元素个数与N加上插入点到序列较近的末端之间元素的个数成线性比例——除非此模板成员函数是对于InIt为输入或前向迭代器的一个特化版本，此时它相当于进行N次单独的插入。在任意一端插入一个元素都会导致所有指向已有元素的迭代器（但不包括引用）变得无效。否则，插入一个元素就会导致所有的迭代器和引用变得无效。

如果在插入单个元素时有异常抛出，容器将保持不变并将异常继续向外抛出；如果在插入多个元素时有异常抛出，并且异常不是在复制元素时抛出的，那么容器将保持不变并将异常继续向外抛出。

口 deque::iterator

```
typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的随机存取迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

口 deque::max_size

```
size_type max_size() const;
```

该成员函数返回对象所能控制的最长序列的长度。

口 deque::operator[]
 const_reference operator[](size_type pos) const;
 reference operator[](size_type pos);

该成员函数返回一个指向被控序列中位置为pos的元素的引用。如果给定的位置无效，那么该函数的行为将是未定义的。

口 deque::pointer
 typedef typename A::pointer pointer;

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 deque::pop_back
 void pop_back();

该成员函数删除被控序列的最后一个元素，在执行pop_back前序列不能为空。删除元素仅会导致原先指向被删除元素的迭代器和引用变得无效。

该成员函数不会抛出任何异常。

口 deque::pop_front
 void pop_front();

该成员函数删除被控序列的第一个元素，在执行pop_front前序列不能为空。删除元素仅会导致原先指向被删除元素的迭代器和引用变得无效。

该成员函数不会抛出任何异常。

口 deque::push_back
 void push_back(const T& x);

该成员函数会把一个值为x的元素插入到被控序列的末端。插入元素会导致所有指向已有元素的迭代器（但不包括引用）变为无效。

如果在调用过程中有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。

口 deque::push_front
 void push_front(const T& x);

该成员函数会把一个值为x的元素插入到被控序列的开始处。插入元素会导致所有指向已有元素的迭代器（但不包括引用）变为无效。

如果在调用过程中有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。

口 deque::rbegin
 const_reverse_iterator rbegin() const;
 reverse_iterator rbegin();

该成员函数返回一个反转型迭代器，它指向紧接着被控序列的末端

的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 `deque::reference`

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 `deque::rend`

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

该成员函数返回一个反转型迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 `deque::resize`

```
void resize(size_type n);
void resize(size_type n, T x);
```

这两个成员函数都确保调用`size()`从而返回n。如果在调用过程中必须要扩展被控序列的长度，那么第一个成员函数将会向被控序列末端添加值为T()的元素，而第二个元素添加值为x的元素。如果想缩短被控序列的长度，这两个成员函数都会调用`erase(begin() + n, end())`。

口 `deque::reverse_iterator`

```
typedef reverse_iterator<iterator>
reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型随机存取迭代器来使用。

口 `deque::size`

```
size_type size() const;
```

该成员函数返回被控序列的长度。

口 `deque::size_type`

```
typedef T2 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

口 `deque::swap`

```
void swap(vector& x);
```

该成员函数会在`*this`和`x`之间相互交换被控序列。如果`get_allocator() == x.get_allocator()`，它将可以在常数时间为完成交换并且不会抛出任何异常，另外，它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则，它将以与这两个被控序列的元素个数成比例的次数调用元素的构造函数以及为元素赋值。

口 deque::value_type

```
typedef typename A::value_type value_type;
```

该类型是模板参数T的同义词。

口 operator!=

```
template<class T, class A>
```

```
bool operator!=(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数返回!(lhs == rhs)。

口 operator==

```
template<class T, class A>
```

```
bool operator==(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数重载operator==来比较两个模板类deque的对象。该函数返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

口 operator<

```
template<class T, class A>
```

```
bool operator<(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数重载operator<来比较两个模板类deque的对象。该函数返回lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

口 operator<=

```
template<class T, class A>
```

```
bool operator<=(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数返回!(rhs < lhs)。

口 operator>

```
template<class T, class A>
```

```
bool operator>(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数返回rhs < lhs。

口 operator>=

```
template<class T, class A>
```

```
bool operator>=(
```

```
const deque <T, A>& lhs,
```

```
const deque <T, A>& rhs);
```

该模板函数返回!(lhs < rhs)。

¶ swap

```
template<class T, class A>
void swap(
    deque <T, A>& lhs,
    deque <T, A>& rhs);
```

该模板函数执行lhs.swap(rhs)。

使用 <deque>

deque

如果想在程序中使用模板类 deque, 请把头文件<deque>包含到程序中。我们也可以指定 deque 来存储类型为 T 的元素, 只需这样写一条类型定义语句:

```
typedef deque<T, allocator<T> > Mycont;
```

然而, 通过使用默认的模板参数, 我们可以省略第二个模板参数。

模板类deque支持所有我们在第9章中所给出的、对于容器来说是常见的一些操作。(参见从第9章的“使用容器”一节开始的讨论。) 我们在此仅概括出模板类deque所特有的属性。

构造函数

为了构造一个类deque<T, A>的对象, 可以这么写:

- deque(), 声明一个空的双队列;
- deque(al), 也是声明一个空的双队列, 但它还存储一个分配器对象al;
- deque(n), 声明一个有n个元素的双队列, 每个元素都是由其默认构造函数 T()所构造出来的;
- deque(n, val), 声明一个有n个元素的双队列, 每个元素的值是由其复制构造函数 T(val)得来的;
- deque(n, val, al), 声明一个和上面一样的双队列, 但它也存储分配器对象al;
- deque(first, last), 声明一个双队列, 其元素的初始值来源于由区间 [first, last)所指定的序列中的元素;
- deque(first, last, al), 声明一个和上面一样的双队列, 但它也存储分配器对象al。

如果我们已经为类型为allocator<T>的分配器特化了模板类(这是我们所常做的也是STL提供的默认值), 那么再在构造函数中显式地指定分配器参数al并不能给我们带来任何好处。仅对程序中显式定义的某些分配器, 这样的参数才会起作用。(参见第4章中对分配器的讨论。)

下面所有的描述中都假定 cont 是类 deque<T, A> 的一个对象。

resize

为了将被控序列的长度改为只容纳 n 个元素, 请调用 cont.resize(n)。超出的元素将会被删除。如果序列需要扩展, 那么值为 T() 的元素会插入到序列末端, 直到填满整个序列。我们也可以调用 cont.resize(n, val) 来插入值为 val 的元素。

clear

为了删除所有的元素, 请调用 cont.clear()。然而请注意, clear 并不保证一定会减少保留的存储空间的大小。(必要时我们会分配和释放元素块, 但映射只会一直增大而不会缩小。) 在调用完 cont.clear() 或把它赋值为一个大小为 0 的双队列后, 本实现确实会释放保留的存储空间, 但这种行为并不是 C++ 标准所要求的。有一种特殊的能确保释放掉所有保留存储空间的办法为:

```
cont.swap(deque<T, A>());
```

该语句先是构造一个临时的空双队列, 然后再把它和 cont 相交换。该语句结束时, 这个临时对象会被销毁, 先前由 cont 所分配的存储空间也随之一同释放。

**front
back**

为了存取被控序列的第一个元素, 请调用 cont.front()。为了存取最后一个元素, 请调用 cont.back()。如果 cont 不是一个常量对象, 调用这两个成员函数得到的表达式将会是一个左值 (lvalue), 于是我们就可以改变原来存储于该元素内的值, 我们只需这么写: cont.front() = T()。然而, 如果序列为空, 上述表达式的行为将是未定义的。

**operator[]
at**

为了存取元素 i (以 0 为基值), 可以这么写: cont[i]。如果 cont 不是一个常量对象, 该表达式产生的将会是一个左值。如果 i 不在半开区间 [0, cont.size()) 中, 那么 cont[i] 将有未定义的行为。我们也可以用 cont.at(i) 来替换 cont[i]。在这种调用方式下, 如果 i 不是一个有效的元素, 表达式就会向外抛出一个类 out_of_range 的异常。

**push_back
push_front**

为了向对象末端插入值为 x 的元素, 请调用 cont.push_back(x)。为了向对象开始处插入值为 x 的元素, 请调用 cont.push_front(x)。

**pop_back
pop_front**

为了删除最后那个元素, 请调用 cont.pop_back()。为了删除第一个元素, 请调用 cont.pop_front()。当然, 对于这两个弹出操作来说, 被控序列都不能为空, 否则操作将有未定义的行为。

assign

为了将被控序列替换成由 [first, last) 所指定的序列, 请调用 cont.assign(first, last)。[first, last) 不能为最初被控序列中的一部分。为了将被控序列替换成 n 个值为 x 的元素, 请调用 cont.assign(n, x)。

insert

为了在由迭代器 it 指定的元素前插入一个值为 x 的元素, 请调用 cont.insert(it, x)。它的返回值是一个迭代器, 指向刚插入的那个元素。为了在由迭代器 it 所指定的元素前插入一个由 [first, last) 指定的序列, 请调用 cont.insert(it, first, last)。序列 [first, last) 不能为最初被控序列中的一部分。为了插入 n 个值为 x 的元素, 请调用 cont.insert(it, n, x)。

erase

为了删除由 it 所指定的元素, 请调用 cont.erase(it)。它的返回值为一个迭代器, 指向紧接着被删除元素的下一个元素。为了删除由区间 [first, last) 指定的所有元素, 请调用 cont.erase(first, last)。

实现 <deque>

程序清单 12-1 列出了文件 deque。在它里面定义了模板类 deque，以及一些以双队列作为操作数的模板函数。

```
程序清单 12-1: // deque standard header
deque
#ifndef DEQUE_
#define DEQUE_
#include <memory>
#include <stdexcept>
namespace std {

    // TEMPLATE CLASS Deque_map
    template<class Ty, class A>
    class Deque_map {
protected:
    Deque_map(A Al)
        : Almap(Al) {}
    typedef typename A::template
        rebind<Ty>::other::pointer Tptr;
    typename A::template rebind<Tptr>::other Almap;
};

    // TEMPLATE CLASS Deque_val
    template<class Ty, class A>
    class Deque_val : public Deque_map<Ty, A> {
protected:
    Deque_val(A Al = A())
        : Deque_map<Ty, A>(Al), Alval(Al) {}
    typedef typename A::template
        rebind<Ty>::other Alty;
    Alty Alval;
};

    // TEMPLATE CLASS deque
    template<class Ty, class Ax = allocator<Ty> >
    class deque
        : public Deque_val<Ty, Ax> {
public:
    enum {DEQUEMAPSIZ = 8}; /* at least 1 */
    enum {DEQUESIZ = sizeof (Ty) <= 1 ? 16
          : sizeof (Ty) <= 2 ? 8
```

```
: sizeof (Ty) <= 4 ? 4
: sizeof (Ty) <= 8 ? 2 : 1};

typedef deque<Ty, Ax> Myt;
typedef Deque_val<Ty, Ax> Mybase;
typedef typename Mybase::Alty A;
typedef A allocator_type;
typedef typename A::size_type size_type;
typedef typename A::difference_type Dift;
typedef Dift difference_type;
typedef typename A::pointer Tptr;
typedef typename A::const_pointer Ctptr;
typedef Tptr pointer;
typedef Ctptr const_pointer;
typedef typename A::template
    rebind::<Tptr>::other::pointer Mapptr;
typedef typename A::reference Reft;
typedef Tptr pointer;
typedef Ctptr const_pointer;
typedef Reft reference;
typedef typename A::const_reference const_reference;
typedef typename A::value_type value_type;

// CLASS iterator
class iterator;
friend class iterator;
class iterator : public Ranit<Ty, Dift, Tptr, Reft> {
public:
    typedef Ranit<Ty, Dift, Tptr, Reft> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    iterator()
        : Idx(0), Deque(0)
    {}
    iterator(difference_type I, const deque<Ty, A> *P)
        : Idx(I), Deque(P)
    {}
    reference operator*() const
    {size_type Block = Idx / DEQUESIZ;
     size_type Off = Idx - Block * DEQUESIZ;
```

```
    if (Deque->Mapsize <= Block)
        Block -= Deque->Mapsize;
    return ((Deque->Map)[Block][Off]); }

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

iterator& operator++()
{ ++Idx;
    return (*this); }

iterator operator++(int)
{ iterator Tmp = *this;
    ++*this;
    return (Tmp); }

iterator& operator--()
{ --Idx;
    return (*this); }

iterator operator--(int)
{ iterator Tmp = *this;
    --*this;
    return (Tmp); }

iterator& operator+=(difference_type N)
{ Idx += N;
    return (*this); }

iterator& operator-=(difference_type N)
{ return (*this += -N); }

iterator operator+(difference_type N) const
{ iterator Tmp = *this;
    return (Tmp += N); }

iterator operator-(difference_type N) const
{ iterator Tmp = *this;
    return (Tmp -= N); }

difference_type operator-(const iterator& X) const
{ return (Idx - X.Idx); }

reference operator[](difference_type N) const
{ return (*(*this + N)); }

bool operator==(const iterator& X) const
{ return (Deque == X.Deque && Idx == X.Idx); }

bool operator!=(const iterator& X) const
{ return (!(*this == X)); }

bool operator<(const iterator& X) const
{ return (Idx < X.Idx); }

bool operator<=(const iterator& X) const
{ return (!(X < *this)); }

bool operator>(const iterator& X) const
{ return (X < *this); }

bool operator>=(const iterator& X) const
```

```
        {return (!(*this < X)); }

protected:
    difference_type Idx;
    const deque<Ty, A> *Deque;
};

// CLASS const_iterator
class const_iterator;
friend class const_iterator;
class const_iterator
    : public Ranit<Ty, Dift, Ctptr, const_reference> {
public:
    typedef Ranit<Ty, Dift, Ctptr, const_reference>
        Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    const_iterator()
        : Idx(0), Deque(0)
    {}
    const_iterator(difference_type I,
        const deque<Ty, A> *P)
        : Idx(I), Deque(P)
    {}
    const_iterator(
        const typename deque<Ty, A>::iterator& X)
        : Idx(X.Idx), Deque(X.Deque)
    {}
    const_reference operator*() const
        {size_type Block = Idx / DEQUESIZ;
         size_type Off = Idx - Block * DEQUESIZ;
         if (Deque->Mapsize <= Block)
             Block -= Deque->Mapsize;
         return ((Deque->Map)[Block][Off]); }
    Ctptr operator->() const
        {return (&**this); }
    const_iterator& operator++()
        {++Idx;
         return (*this); }
    const_iterator operator++(int)
        {const_iterator Tmp = *this;
```

```
    ++*this;
    return (Tmp); }
const_iterator& operator--()
{--Idx;
 return (*this); }
const_iterator operator--(int)
{const_iterator Tmp = *this;
 --*this;
 return (Tmp); }
const_iterator& operator+=(difference_type N)
{Idx += N;
 return (*this); }
const_iterator& operator-=(difference_type N)
{return (*this += -N); }
const_iterator operator+(difference_type N) const
{const_iterator Tmp = *this;
 return (Tmp += N); }
const_iterator operator-(difference_type N) const
{const_iterator Tmp = *this;
 return (Tmp -= N); }
difference_type operator-
{const const_iterator& X) const
{return (Idx - X.Idx); }
const_reference operator[](difference_type N) const
{return (*(*this + N)); }
bool operator==(const const_iterator& X) const
{return (Deque == X.Deque && Idx == X.Idx); }
bool operator!=(const const_iterator& X) const
{return (!(*this == X)); }
bool operator<(const const_iterator& X) const
{return (Idx < X.Idx); }
bool operator<=(const const_iterator& X) const
{return (!(X < *this)); }
bool operator>(const const_iterator& X) const
{return (X < *this); }
bool operator>=(const const_iterator& X) const
{return (!(*this < X)); }
protected:
difference_type Idx;
const deque<Ty, A> *Deque;
};

typedef std::reverse_iterator<iterator>
reverse_iterator;
typedef std::reverse_iterator<const_iterator>
```

```
    const_reverse_iterator;

    deque()
        : Mybase(), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {}
    explicit deque(const A& A1)
        : Mybase(A1), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {}
    explicit deque(size_type N)
        : Mybase(), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {insert(begin(), N, Ty()); }
    deque(size_type N, const Ty& V)
        : Mybase(), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {insert(begin(), N, V); }
    deque(size_type N, const Ty& V, const A& A1)
        : Mybase(A1), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {insert(begin(), N, V); }
    deque(const Myt& X)
        : Mybase(X.Alval), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {insert(begin(), X.begin(), X.end()); }

    template<class It>
    deque(It F, It L)
        : Mybase(), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {Construct(F, L, Iter_cat(F)); }

    template<class It>
    deque(It F, It L, const A& A1)
        : Mybase(A1), Map(0),
          Mapsize(0), Offset(0), Size(0)
    {Construct(F, L, Iter_cat(F)); }

    template<class It>
    void Construct(It F, It L, Int_iterator_tag)
    {insert(begin(), (size_type)F, (Ty)L); }

    template<class It>
    void Construct(It F, It L, input_iterator_tag)
    {insert(begin(), F, L); }

    ~deque()
    {clear(); }
```

```
Myt& operator=(const Myt& X)
    {if (this == &X)
     ;
     else if (X.size() == 0)
         clear();
     else if (X.size() <= size())
         (iterator S = copy(X.begin(), X.end(), begin()));
         erase(S, end()); }
     else
         {const_iterator Sx = X.begin() + size();
          copy(X.begin(), Sx, begin());
          insert(end(), Sx, X.end()); }
     return (*this); }
iterator begin()
    {return (iterator(Offset, this)); }
const_iterator begin() const
    {return (const_iterator(Offset, this)); }
iterator end()
    {return (iterator(Offset + Size, this)); }
const_iterator end() const
    {return (const_iterator(Offset + Size, this)); }
reverse_iterator rbegin()
    {return (reverse_iterator(end())); }
const_reverse_iterator rbegin() const
    {return (const_reverse_iterator(end())); }
reverse_iterator rend()
    {return (reverse_iterator(begin())); }
const_reverse_iterator rend() const
    {return (const_reverse_iterator(begin())); }
void resize(size_type N)
    {resize(N, Ty()); }
void resize(size_type N, Ty X)
    {if (size() < N)
        insert(end(), N - size(), X);
     else if (N < size())
        erase(begin() + N, end()); }
size_type size() const
    {return (Size); }
size_type max_size() const
    {return (Alval.max_size()); }
bool empty() const
    {return (size() == 0); }
allocator_type get_allocator() const
    {return (Alval); }
const_reference at(size_type P) const
```

```
    {if (size() <= P)
        Xran();
     return (*begin() + P)); }
reference at(size_type P)
{if (size() <= P)
    Xran();
 return (*begin() + P)); }
const_reference operator[](size_type P) const
{return (*begin() + P)); }
reference operator[](size_type P)
{return (*begin() + P)); }
reference front()
{return (*begin()); }
const_reference front() const
{return (*begin()); }
reference back()
{return (*(end() - 1)); }
const_reference back() const
{return (*(end() - 1)); }
void push_front(const Ty& X)
{if (Offset % DEQUESIZ == 0
    && Mapsize <= (Size + DEQUESIZ) / DEQUESIZ)
    Growmap(1);
size_type Newoff = Offset != 0 ? Offset
    : Mapsize * DEQUESIZ;
size_type Block = --Newoff / DEQUESIZ;
if (Map[Block] == 0)
    Map[Block] = Alval.allocate(DEQUESIZ, (void *)0);
try {
Offset = Newoff;
++Size;
Alval.construct(Map[Block] + Newoff % DEQUESIZ, X);
} catch (...) {
pop_front();
throw;
}}
void pop_front()
{if (!empty())
    {size_type Block = Offset / DEQUESIZ;
     Alval.destroy(Map[Block] + Offset % DEQUESIZ);
     if (Mapsize * DEQUESIZ <= ++Offset)
         Offset = 0;
     if (--Size == 0)
         Offset = 0; }}
void push_back(const Ty& X)
```

```
    {if ((Offset + Size) % DEQUESIZ == 0
        && Mapsize <= (Size + DEQUESIZ) / DEQUESIZ)
        Growmap(1);
    size_type Newoff = Offset + Size;
    size_type Block = Newoff / DEQUESIZ;
    if (Mapsize <= Block)
        Block -= Mapsize;
    if (Map[Block] == 0)
        Map[Block] = Alval.allocate(DEQUESIZ, (void *)0);
    try {
        ++Size;
        Alval.construct(Map[Block] + Newoff % DEQUESIZ, X);
    } catch (...) {
        pop_back();
        throw;
    }
    void pop_back()
    {if (!empty())
        {size_type Newoff = Size + Offset - 1;
        size_type Block = Newoff / DEQUESIZ;
        if (Mapsize <= Block)
            Block -= Mapsize;
        Alval.destroy(Map[Block] + Newoff % DEQUESIZ);
        if (--Size == 0)
            Offset = 0; }}
template<class It>
    void assign(It F, It L)
    {Assign(F, L, Iter_cat(F)); }
template<class It>
    void Assign(It F, It L, Int_iterator_tag)
    {assign((size_type)F, (Ty)L); }
template<class It>
    void Assign(It F, It L, input_iterator_tag)
    {erase(begin(), end());
     insert(begin(), F, L); }
void assign(size_type N, const Ty& X)
    {Ty Tx = X;
erase(begin(), end());
insert(begin(), N, Tx); }
iterator insert(iterator P, const Ty& X)
    {if (P == begin())
        {push_front(X);
        return (begin()); }
    else if (P == end())
        {push_back(X); }
```

```
        return (end() - 1); }
else
    {iterator S;
size_type Off = P - begin();
Ty Tx = X;
if (Off < size() / 2)
    {push_front(front());
S = begin() + Off;
copy(begin() + 2, S + 1, begin() + 1); }
else
    {push_back(back());
S = begin() + Off;
copy_backward(S, end() - 2, end() - 1); }
*S = Tx;
return (S); }
void insert(iterator P, size_type M, const Ty& X)
{iterator S;
size_type I;
size_type Off = P - begin();
size_type Rem = Size - Off;
if (Off < Rem)
if (Off < M)
    {for (I = M - Off; 0 < I; --I)
push_front(X);
for (I = Off; 0 < I; --I)
push_front(begin()[M - 1]);
S = begin() + M;
fill(S, S + Off, X); }
else
    {for (I = M; 0 < I; --I)
push_front(begin()[M - 1]);
S = begin() + M;
Ty Tx = X;
copy(S + M, S + Off, S);
fill(begin() + Off, S + Off, Tx); }
else
if (Rem < M)
    {for (I = M - Rem; 0 < I; --I)
push_back(X);
for (I = 0; I < Rem; ++I)
push_back(begin()[Off + I]);
S = begin() + Off;
fill(S, S + Rem, X); }
else
    {for (I = 0; I < M; ++I)
```

```
        push_back(begin()[Off + Rem - M + I]);
        S = begin() + Off;
        Ty Tx = X;
        copy_backward(S, S + Rem - M, S + Rem);
        fill(S, S + M, Tx); })
template<class It>
void insert(iterator P, It F, It L)
(Insert(P, F, L, Iter_cat(F)); )
template<class It>
void Insert(iterator P, It F, It L,
Int_iterator_tag)
(insert(P, (size_type)F, (Ty)L); )
template<class It>
void Insert(iterator P, It F, It L,
input_iterator_tag)
(size_type Off = P - begin();
for (; F != L; ++F, ++Off)
    insert(begin() + Off, *F); )
template<class It>
void Insert(iterator P, It F, It L,
bidirectional_iterator_tag)
(size_type M = 0;
Distance(F, L, M);
size_type I;
size_type Off = P - begin();
size_type Rem = Size - Off;
if (Off < Rem)
    if (Off < M)
        {It Qx = F;
        advance(Qx, M - Off);
        for (It Q = Qx; F != Q; )
            push_front(*--Q);
        for (I = Off; 0 < I; --I)
            push_front(begin()[M - 1]);
        copy(Qx, L, begin() + M); }
    else
        {for (I = M; 0 < I; --I)
            push_front(begin()[M - 1]);
        iterator S = begin() + M;
        copy(S + M, S + Off, S);
        copy(F, L, begin() + Off); }
else
    if (Rem < M)
        {It Qx = F;
        advance(Qx, Rem); }
```

```
        for (It Q = Qx; Q != L; ++Q)
            push_back(*Q);
        for (I = 0; I < Rem; ++I)
            push_back(begin() [Off + I]);
        copy(F, Qx, begin() + Off); }
    else
        {for (I = 0; I < M; ++I)
            push_back(begin() [Off + Rem - M + I]);
        iterator S = begin() + Off;
        copy_backward(S, S + Rem - M, S + Rem);
        copy(F, L, S); }
    iterator erase(iterator P)
        {return (erase(P, P + 1)); }
    iterator erase(iterator F, iterator L)
        {size_type N = L - F;
        size_type M = F - begin();
        if (M < (size_type)(end() - L))
            {copy_backward(begin(), F, L);
        for (; 0 < N; --N)
            pop_front(); }
        else
            {copy(L, end(), F);
        for (; 0 < N; --N)
            pop_back(); }
        return (M == 0 ? begin() : begin() + M); }
void clear()
    {while (!empty())
        pop_back();
    Freemap(); }
void swap(Myt& X)
    {if (Alval == X.Alval)
        {std::swap(Map, X.Map);
        std::swap(Mapsize, X.Mapsize);
        std::swap(Offset, X.Offset);
        std::swap(Size, X.Size); }
    else
        {Myt Ts = *this; *this = X, X = Ts; }}
protected:
    void Xlen() const
        {throw length_error("deque<T> too long"); }
    void Xran() const
        {throw out_of_range("deque<T> subscript"); }
    void Freemap()
```

```

        (for (size_type M = Mapsize; 0 < M; )
            {Alval.deallocate(*(Map + --M), DEQUESIZ);
             Almap.destroy(Map + M); }
            Almap.deallocate(Map, Mapsize);
            Mapsize = 0;
            Map = 0; }

void Growmap(size_type N)
    {if (max_size() / DEQUESIZ - Mapsize < N)
        Xlen();
     size_type I = Mapsize / 2; // try to grow by 50%
     if (I < DEQUEMAPSIZ)
        I = DEQUEMAPSTZ;
     if (N < I && Mapsize <= max_size() / DEQUESIZ - I)
        N = I;
     size_type Ib = Offset / DEQUESIZ;
     Mapptr M = Almap.allocate(Mapsize + N, (void *)0);
     Mapptr Mn = M + Ib;
     for (I = Ib; I < Mapsize; ++I, ++Mn)
        {Almap.construct(Mn, Map[I]);
         Almap.destroy(Map + I); }
     if (Ib <= N)
        {for (I = 0; I < Ib; ++I, ++Mn)
            {Almap.construct(Mn, Map[I]);
             Almap.destroy(Map + I);
             Almap.construct(M + I, 0); }
         for (; I < N; ++I, ++Mn)
             Almap.construct(Mn, 0); }
     else
        {for (I = 0; I < N; ++I, ++Mn)
            {Almap.construct(Mn, Map[I]);
             Almap.destroy(Map + I); }
         for (Mn = M; I < Ib; ++I, ++Mn)
             {Almap.construct(Mn, Map[I]);
              Almap.destroy(Map + I); }
         for (I = 0; I < N; ++I, ++Mn)
             Almap.construct(Mn, 0); }
     Map = M;
     Mapsize += N; }

Mapptr Map;
size_type Mapsize, Offset, Size;
};

// deque TEMPLATE OPERATORS
template<class Ty, class A> inline

```

```

        void swap(deque<Ty, A>& X, deque<Ty, A>& Y)
        {X.swap(Y); }

template<class Ty, class A> inline
    bool operator==(const deque<Ty, A>& X,
                      const deque<Ty, A>& Y)
    {return (X.size() == Y.size())
     && equal(X.begin(), X.end(), Y.begin())); }
template<class Ty, class A> inline
    bool operator!=(const deque<Ty, A>& X,
                      const deque<Ty, A>& Y)
    {return !(X == Y); }
template<class Ty, class A> inline
    bool operator<(const deque<Ty, A>& X,
                     const deque<Ty, A>& Y)
    {return (Lexicographical_compare(X.begin(), X.end(),
                                    Y.begin(), Y.end())); }
template<class Ty, class A> inline
    bool operator<=(const deque<Ty, A>& X,
                     const deque<Ty, A>& Y)
    {return !(Y < X); }
template<class Ty, class A> inline
    bool operator>(const deque<Ty, A>& X,
                     const deque<Ty, A>& Y)
    {return (Y < X); }
template<class Ty, class A> inline
    bool operator>=(const deque<Ty, A>& X,
                     const deque<Ty, A>& Y)
    {return !(X < Y); }
} /* namespace std */
#endif /* DEQUE_ */

```

模板类deque以一个两层结构的方式存储了一个长度为N的被控序列。
有两个参数可以检测该结构的形状：

DEQUESIZ

- 双队列元素存储在固定大小的块中，每一个块都是一个连续的拥有DEQUESIZ个元素的数组。（DEQUESIZ对应于本章前面所描述过的参数B。）设计出该定义是为了保证在试图将多个小元素打包存到一个合理大小的块中时，每一块都至少有一个元素。

DEQUEMAPSIZ

- 上述的每一个块都由一个映射数组中的一个元素指定。该映射数组包括一个（逻辑上）连续的指针序列，每个指针都指向存储在一个大的包容数组中的存储块。这样的包容数组一旦被分配，就至少有DEQUEMAPSIZ个元素。DEQUEMAPSIZ大小的选择是为了使得在双队列第一次增长时所需的对映射的重新分配最小化，以及调整在堆中所分配的

对象通常所拥有的开销。

附带说一句，DEQUESIZ 和 DEQUEMAPSIZ 都是需要权衡的参数。它们越小，增长一个短的被控序列所浪费的开销也就越少。它们越大，浪费在增长大的受控序列上的时间就越少。本实现试图分配的存储块最小为 16 个字节，这样可以调整在堆中分配存储块时所造成的开销。它同样也试图让映射尽可能小，但并没有小到离谱的状态。一般在一个映射中最少会有 8 个指针。

模板类 deque 定义分配器对象的方式和模板类 list 中的差不多一样，并且理由也一样。（参见第 11 章的“实现<list>一节”开始的讨论。）它存储两种分配器对象：

Almap

- Almap 是为元素指针类型服务的，它可以分配和释放指针数组（映射），并且创建和销毁在这些数组中的指针（如果指针类型不是标量指针的话）。

Altv

- Altv 是为元素类型所服务的，它可以分配和释放元素数组（块），并且创建和销毁这些数组中的元素。

Deque_map

与文件 list 相同的还有，文件 deque 也定义两个模板基类以存储上述分配器（我们希望在双队列对象中使用它们以得到更好的存储效率）。（参见第 10 章对大小为零的分配器的讨论。）这两个基类中的第一个就是模板类 Deque_map。Deque_map<T, A> 的特化版本可以作为 deque<T, A> 的最底层基类。它唯一的目的就是存储分配器对象 Almap。

Deque_val

模板类 Deque_val 提供了第二个分配器对象。Deque_val<T, A> 的特化版本以公共方式派生自 Deque_map<T, A>，并可以作为 deque<T, A> 的公共基类。它唯一的目的就是存储分配器对象 Alval。

除了分配器对象外，deque 对象中还存储了几个用以表示被控序列的对象。更确切地说，deque 中存储了一个映射指针和三个计数器：

Map

• Map 是一个指针，指向一个指针映射数组。对于一个刚构造好的空双队列来说，它最初是一个空指针。没有被使用的映射数组元素也存储了空指针。

Mapsize

- Mapsize 计算在整个映射数组中指针元素的个数。

Offset

- Offset 计算双队列中第一个有效元素前空闲元素的个数。（第一个有效元素位于 Map[Offset / DEQUEMAPSIZ][Offset % DEQUEMAPSIZ]。）

Size

- Size 计算双队列中有效元素的个数。

双队列中，最前面和最后面的存储块可以是部分填满的。后面的存储块是从前往后填，而前面的则是从后往前填。如果在一个存储块中已经没有空间可以容纳新增的元素，双队列对象就会分配另一个存储块并扩展当前的映射，把它指向新的存储块。

对映射 进行增长

一个双队列对象会试图保留出空间使得映射可以向两端扩展，这样就会使由前新增（prepend）和由后新增（append）一样高效。一旦映射中没有多余的空间来满足在包容它的数组中增长的需要，双队列对象就

必须分配一个新的包容数组并把整个映射复制过去。我们肯定不希望在不必要的的情况下经常发生这样的行为。否则，新增一个元素的时间复杂度就将变为线性的（和被控序列的长度一同增长），而不是一个常数（和被控序列的长度无关）。

这就导致了一个有趣的问题。进一步说，双队列只是简单地用指向元素（块）的指针数组来替换元素数组。当它增长时，它将不得不重复地重新分配和复制这些指针，就像是向量不得不重复地重新分配和复制元素一样。于是，看起来双队列增长的时间复杂度和向量应该一样。如果指针的大小小于它所指向的元素块的大小，我们所节省下来的只是那些复制时间。如果指针拥有平凡构造函数和析构函数（如标量指针），我们同样也可以节省下部分时间，而对于元素来说这就不可能了。但不管怎么说，在增长双队列中的映射时，我们必须像对待增长向量中的元素数组一样小心。

我们在第10章对vector::insert的描述中提过一种基本的技巧。它可以降低增长一个向量或一个双队列映射所需分摊的时间复杂度。在这个简单的策略中，每个新分配出来的映射数组都是原来大小的两倍。当被控序列越变越长时，重新分配的速度则越来越小。这样就限定了被控序列增长时的平均成本。本实现试图每次把映射增大50%。这和vector::insert中的一样，得到的时间复杂度也一样。但代码为了达到更高效的存储效率，将会进行大量的重新分配。

本实现还利用了另一种重要的技术。它允许当前映射在映射数组的任意一端增长时被包装（wrap）起来。这种方法就是：仅在被控序列开始处的元素面临着和末端元素共享同一存储块时——换一种说法就是此时映射已经满了，当前映射才会需要进行重新分配。例如，成员函数push_front以下面的测试开始：

```
if (Offset % DEQUESIZ == 0  
    && Mapsize <= (Size + DEQUESIZ) / DEQUESIZ)  
    Growmap(1);
```

我们将在push_back的开始处发现一个类似的测试。相反，一个通常的模板类deque的实现并不允许把映射包装起来。如果当前映射到达了映射数组中的一端，就会发生以下两件事情之一。要么是当前映射移出了末端（或起始端），要么就是它被复制到一个新分配的映射中并获得了继续增长的空间。

每一种方法都有正反两方面的影响。如果没有包装映射：

- 对于双队列迭代器调用operator*()就变得没有价值。

- operator++()和其他算术运算将会更华丽^②。
- 即使映射数组并没有完全填满时，溢出情况也可能发生在映射的任意一端。

对映射进行增长会变得更复杂。

如果包装了映射：

- 对于双队列迭代器调用operator*()将会更华丽。
- operator++()和其他算术运算将会变得没有价值。
- 只有在映射被完全填满时才可能发生溢出。
- 对映射进行增长会变得很简单。

也就是说，本实现假设，在保存到其他一些地方后，通过迭代器存取元素的额外开销比通过偏移量所需的要多。

iterator

const_iterator

通过学习嵌套类**iterator**和**const_iterator**，我们可以知道在一个双队列中存取元素时所需的所有知识。它们实现了遍历被控序列、包装映射以及在存储块之间跳转等技巧。双队列迭代器和模板类Ptrit（参见第3章的“实现<iterator>”一节）非常相似，而Ptrit又是向量迭代器的基础。它们都实现了随机存取迭代器。

Deque

Idx

每个双队列迭代器都存储了两个对象：

- Deque 是一个指向双队列对象的指针
- Idx 是被指定元素的偏移量

于是，被指定的元素就是：

```
Deque->Map[(Idx / DEQUESIZ) % Deque->Mapsiz]
[Idx * DEQUESIZ]
```

（参见每个嵌套的迭代器类中operator*()的定义。）当映射增长时，它的有效内容就被复制，此时对于被控序列中的所有有效元素来说，该表达式仍然保持有效。

嵌套类**const_iterator**和嵌套类**iterator**有着明显的区别。它还增加了一个构造函数，用以将一个**iterator**对象隐式地转化为一个**const_iterator**对象^③。我们可以在所有STL容器迭代器定义中找到相似的机制。

push_back
push_front
Growmap

通过学习成员函数**push_back**和**push_front**，我们就可以了解所有有关双队列增长的知识。这两个函数都会调用保护型的成员函数**Growmap**，用它在映射中没有空间容纳新增的块指针时来增长映射。**Growmap**必须在复制映射时很小心地把它们“去除包装（unwrap）”，正如前面所描述的一样，这样做的目的是为了保持已有的迭代器有效。

pop_back
pop_front

通过学习成员函数**pop_back**和**pop_front**，我们就可以了解所有有关双队列缩小的知识。注意，这两个成员函数并不会在块为空时释放它们。

^② 意指它的开销会比较大。——译者注

^③ 一般来说，不推荐使用隐式转化。——译者注

它们被保留下来以备被控序列又增长时能够快速回收使用。一个更关注存储空间的有效使用的实现可能就会选择更主动地释放这些块。

clear
Freemap
operator=

成员函数**clear**会释放所有的块以及映射。它通过把所有的元素都弹出的方式清空整个容器，然后调用保护型的成员函数**Freemap**来释放存储空间以及销毁映射指针。注意，仅在销毁双队列或者是使用赋值运算符(**operator=**)来赋值一个空双队列时，**clear**才会被调用。C++标准并没有指明STL容器中隐藏的存储空间是如何分配和释放的，这些主要取决于实现的策略。再次重申一遍，一个更关注存储空间的有效使用的实现可能会选择更主动地去释放映射存储空间。

Vector

模板类**deque**的其余部分看起来和模板类**vector**的内部结构很相似，当然，这是有足够理由的。它们都以统一的外部接口实现随机存取容器。它们主要的区别在于：**deque**通过使用一连串的推入操作，减少了增长被控序列所需的花费；通过使用一连串的弹出操作，减少了缩短被控序列所需的花费。但是还有，我们已经连同模板类**vector**(参见第10章)和模板类**list**(参见第11章)，讨论了我们在此没有讨论的其他特性。

程序清单 12-2: // test <deque>
tdeque.c

#include <assert.h>
#include <iostream>
#include <deque>
using namespace std;

// TEST <deque>
int main()
{
 (typedef allocator<char> Myal;
 // TEST deque
 typedef deque<char, Myal> Mycont;
 char ch, carr[] = "abc";
 Mycont::allocator_type *p_alloc = (Myal *)0;
 Mycont::pointer p_ptr = (char *)0;
 Mycont::const_pointer p_cptr = (const char *)0;
 Mycont::reference p_ref = ch;
 Mycont::const_reference p_cref = (const char&)ch;
 Mycont::size_type *p_size = (size_t *)0;
 Mycont::difference_type *p_diff = (ptrdiff_t *)0;
 Mycont::value_type *p_val = (char *)0;

 Mycont v0;
 Myal al = v0.get_allocator();
 Mycont v0a(al);
 assert(v0.empty() && v0.size() == 0);
}

```
assert(v0a.size() == 0 && v0a.get_allocator() == al);
Mycont v1(5), v1a(6, 'x'), v1b(7, 'y', al);
assert(v1.size() == 5 && v1.back() == '\0');
assert(v1a.size() == 6 && v1a.back() == 'x');
assert(v1b.size() == 7 && v1b.back() == 'y');
Mycont v2(v1a);
assert(v2.size() == 6 && v2.front() == 'x');
Mycont v3(v1a.begin(), v1a.end());
assert(v3.size() == 6 && v3.front() == 'x');
const Mycont v4(v1a.begin(), v1a.end(), al);
assert(v4.size() == 6 && v4.front() == 'x');
v0 = v4;
assert(v0.size() == 6 && v0.front() == 'x');
assert(v0[0] == 'x' && v0.at(5) == 'x');

// v0.reserve(12);
// assert(12 <= v0.capacity());
v0.resize(8);
assert(v0.size() == 8 && v0.back() == '\0');
v0.resize(10, 'z');
assert(v0.size() == 10 && v0.back() == 'z');
assert(v0.size() <= v0.max_size());

Mycont::iterator p_it(v0.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v0.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert(*p_it == 'x' && *--(p_it = v0.end()) == 'z');
assert(*p_cit == 'x' && *--(p_cit = v4.end()) == 'x');
assert(*p_rit == 'z'
&& *--(p_rit = v0.rend()) == 'x');
assert(*p_crit == 'x'
&& *--(p_crit = v4.rend()) == 'x');

assert(v0.front() == 'x' && v4.front() == 'x');
v0.push_front('a');
assert(v0.front() == 'a');
v0.pop_front();
assert(v0.front() == 'x' && v4.front() == 'x');

v0.push_back('a');
assert(v0.back() == 'a');
v0.pop_back();
assert(v0.back() == 'z' && v4.back() == 'x');
```

```
v0.assign(v4.begin(), v4.end());
assert(v0.size() == v4.size())
    && v0.front() == v4.front());
v0.assign(4, 'w');
assert(v0.size() == 4 && v0.front() == 'w');
assert(*v0.insert(v0.begin(), 'a') == 'a');
assert(v0.front() == 'a'
    && *++v0.begin() == 'w');
v0.insert(v0.begin(), 2, 'b');
assert(v0.front() == 'b'
    && *++v0.begin() == 'b'
    && *++v0.begin() == 'a');
v0.insert(v0.end(), v4.begin(), v4.end());
assert(v0.back() == v4.back());
v0.insert(v0.end(), carr, carr + 3);
assert(v0.back() == 'c');
v0.erase(v0.begin());
assert(v0.front() == 'b' && *++v0.begin() == 'a');
v0.erase(v0.begin(), ++v0.begin());
assert(v0.front() == 'a');

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

cout << "SUCCESS testing <deque>" << endl;
return (0); }
```

测试 <deque>

tdeque.c

程序清单12-2列出了文件tdeque.c。它是三个看起来很相似的测试程序中的一个，另外两个是文件tvector.c和文件tlist.c。为了简单地比较这三个测试程序，我们只是简单地注释掉那些和给定容器不一致的测试段，而不是删除这些没有使用的代码。

该测试程序只是对模板类deque的一个特化版本进行了简单的测试，

按照其意图测试了它所给出的每个成员函数和成员类型。如果测试一切顺利的话，测试程序将打印出：

```
SUCCESS testing <deque>
```

然后正常退出。

习题

- 习题12-1 什么时候我们会选择双队列，而不是字符串？
- 习题12-2 在双队列中允许块大小可变会导致什么结果？
- 习题12-3 找出一个比较好的办法，缩短模板类deque用以存储映射的数组。
- 习题12-4 在模板类deque中添加一个参数，使得我们可以指定存储块的大小。我们为什么希望这么做呢？
- 习题12-5 [较难] 改变模板类deque的定义，使之在每个块中只存储一个元素并且从不对元素赋值。我们为什么希望这么做呢？
- 习题12-6 [特难] 改变模板类deque的定义，使得调用iterator::operator*和iterator::operator++的花费都很小。

第13章 <set>

背景知识

<set>

set

multiset

头文件<set>中定义了模板类 set 和模板类 multiset。它们都是容器，它们各自控制的长度为 N 的序列是以一个有 N 个节点的有序二叉树的方式存储的。每个节点存储一个常量元素。该序列用以作为排序依据的函数对象的类型为该类的一个模板参数。多重集合允许两个（相邻的）元素有相等的次序，而集合则不允许出现这种情况。更通俗点讲，集合中的所有元素都是独一无二的，而多重集合则允许有副本。

<map>

map

multimap

在下一章中，我们将介绍<set>的伙伴<map>。它定义了模板类 map 和模板类 multimap。它们与 set 以及 multiset 有一个基本的区别——它们中存储的元素类型为 pair<const Key, T>，但仅有 const Key 会参与次序比较。而对集合和多重集合来说，整个元素的值都会参与次序比较。通俗点讲，映射或者多重映射中相应于每个键（key），存储了一个“映射值（mapped value）”。正如你所猜测的那样，映射中的所有键都是独一无二的，而多重映射则允许有重复。

关联容器

模板类set、multiset、map以及multimap被称为关联容器（associative container）。它们都是把一个键值与一个元素联系起来，并使用该键来加速诸如查找、插入以及删除元素等操作。第9章的表9-1中列出了在关联容器中增加的复杂度所带来的回报。（见该表中的“set/map”一栏。）毫无疑问，在需要使用键值来查找元素时，它们要比其他所有的STL容器表现得更好。虽说一个有序向量或双队列也能达到同样的效果，但它们在插入和删除时都无法做到比线性时间更好的效果。它们底层的树表达形式可以使这些操作在对数时间内完成。

如果我们需要在结合键值的同时完成按索引进行查找，向量和双队列的效果可能会更好，不过这种情况很罕见。如果我们舍不得每个元素的额外指针所占的存储空间，向量和双队列也是不错的选择。然而，再次重申一次，只有在很少的情况下它们才会被主要考虑。如果我们需要维护一个有序序列以加速关联查找、插入或删除等操作，最好的选择就是使用关联容器中的一个。但如果不需要这种额外的性能，我们或许可以避免那些导致较好时间效率的可观的复杂性。

这四个容器中的所有成员通常都使用排序以及树结构以达到更好的效率。首先考虑通过排序来加速元素的查找。假设有一个值val，我们想

知道一个容器对象中是否存储了一个值等于它的元素。如果容器是无序的，我们就不得不对容器中的每个对象都检测一遍以找到所有匹配该值的元素。仅仅为了找到一个匹配的元素，平均下来我们都必须检查容器中的一半元素。在这两种情况中，对于一个有N个元素的容器来说，操作所需的时间复杂度都是线性的，或者表达为 $O(N)$ 。

有序容器

如果在要比较的值上面加以某种次序，并且可以从该种次序中获益（如跳过某些比较），就可以加速查找。例如，假设在某个常见类型的对象X和Y上定义operator<，我们就可以使用 $X < Y$ 来对容器中的元素进行比较。如果希望在被控序列的越前面出现的值越小，并且不允许有重复的键，那么我们就可以认定：如果X在Y前面，则 $X < Y$ 就一定为true；如果允许有重复，那么如果X在Y前面，则 $!(Y < X)$ 就一定为true。（在没有定义operator \leq 的情况下，这种方法对于允许出现相等次序元素的情况有着周到的考虑，它把相等次序的元素放置在相邻的位置上。）

binary_search

然而，一个排序良好的双向链表并不能够给我们带来任何好处。我们仍然不得不在链表中从一个元素移到下一个元素上，以遍历链表中所有的元素。正式点讲，模板类list仅支持通过双向迭代器来存取它里面的元素。我们最好还是提供一个向量或者是双队列。它们支持通过随机存取迭代器来存取元素，这样我们就可以在一个有序序列中实现一个以对半检索来查找一个特定的值（或者是找不到）的方法。模板函数binary_search（在头文件<algorithm>中定义）就是用这种方法来查找元素的。（参见第6章。）

对于二分法来说，查找所需的时间和N的对数成正比。相对于线性查找来说，这是一个极大的飞跃。让我们还是用实例来说话，考虑对一个有1000个元素的序列进行查找时的开销：平均来说，线性查找必须进行500次比较，而二分查找只需进行10次。当N增加时，这种差异就会变得更显著。对于1000000个元素来说，相应的比较次数就变成了500000比20。这一点毫无疑问。

priority_queue

这样看起来我们就需要定义模板类ordered_vector以及ordered_deque。而我们实际上所做的却有点不一样（参见第16章中的模板容器适配器priority_queue）。每个有序容器实际上都可以根据已有的模板容器实现。我们可能希望增加一个模板参数Pred来指定排序规则。这样，一个类Pred的对象pred就可以提供谓词pred(X, Y)作为排序规则 $X < Y$ 的一般化版本。

这种新增机制所带来的回报就是可以快速地通过键值来查找。例如，我们可以增加一个成员函数find(key)，给它一个键值作为参数，然后它就可以在很短的时间内找到相匹配的元素。当然，我们对于“匹配”这个词的含义有着更精确的要求。假设给定一个排序规则如 $X < Y$ ，如果 $!(X <$

$Y) \&& !(Y < X)$ (即没有一个操作数会小于另外一个)，我们就说两个操作数是相匹配的。对这个匹配的定义也就是我们一直说的相等次序。而基于operator<或者相等谓词的排序规则，我们就称之为“严格弱序 (strict weak ordering) ”。

但对一个有序序列的管理来说，除了通过键值查找元素之外，通常我们还有更多的事情要做。我们可能要随着时间的变化不断地重新建立这个序列。我们也可能不断地从序列中删除元素。如果插入和删除的次数比得上进行查找的次数，并且所有这些操作都是交替进行的，那么对于这种情况来说，有序的随机存取容器并不是最佳的选择。于是，我们就引入了树的概念。

二叉树

为什么要使用树这种数据结构？基本的理由已经足够明显了。它就是按照老式的“分而治之”的方式来工作的。假设我们所希望存储的每个元素都包含了一个类型为Key的键，并且Key还定义了operator<(const Key&)，我们就可以定义一个类型为Node的树的节点，它看起来就像是这样：

```
class Node {
    class Node *parent, *left, *right;
    Key key;
    ...};
```

类型为Node的对象x是被称为“二叉树 (binary tree)”的简单树的一个基本组成部分。每个节点都拥有一个父节点和最多两个子节点，如果在被控序列中每个键都是惟一的，那么这棵树就按以下两种规则来排序：

- 如果left是x的左子树中的任意节点，那么就有left->key < x.key。
- 如果right是x的右子树中的任意节点，那么就有x.key < right->key。

基于树的set容器同样也遵循这些规则。(对于map容器来说也是这样。它所不同的就在于每个节点中所附加存储的数据。我们上面所给出的Node声明中用省略号表示这种可能性。)

如果在被控序列中不是每个键都是惟一的，那么树就按另外两个有点不同于上面的规则来排序：

- 如果left是x的左子树中的任意节点，那么就有!(x->key < left.key)。
- 如果right是x的右子树中的任意节点，那么就有!(right.key < x->key)。

基于树的multiset容器同样也遵循这些规则。(multimap容器也是如此。只不过它还像map一样，带有一些其他的内容而已。)

我们可以以递增的方式来存取一个有序树中的元素：从树的根节点开始，递归遍历左子树中的每个节点，然后是节点本身，再然后是它的右子树。这也就是我们在递增迭代器时对二叉树进行走访的方式。(注意，我们所使用的是一个双向迭代器。)更重要的是，我们可以通过在树

中从根节点开始不断下降并拿存储在当前节点中的键值和搜索查找键进行比较，从而找到给定的查找键——或它应该位于树的什么地方。每次比较都会排除掉整个子树。

如果树很长但宽度又很小的话，我们通常都不会取得好的效率。例如，每个右子树都只包含一个单独的节点，而左子树则总是包含所有其他的元素。于是，我们就会发现，这实际上是一种更昂贵的实现链表的方法。查找需要的时间复杂度是线性的——查找一个元素所需的平均时间直接和被控序列中元素的个数N成比例。更确切点说，对于一个键的随机分布来说，它就是 $N/2$ 。最差的情况下查找时间将会是N。这很不好。

平衡二叉树

但先还是让我们认为可以设计及保持一棵短而且茂密的树。在理想情况下它就是一棵平衡树，从根节点到任意叶子节点间的路径长度之间不会相差超过1个链接。（如果树中有 2^N-1 个节点，那么所有的路径都必须一样长。）对于这样的树来说，最长的路径长度就为 $\log_2 N$ 。查找操作的时间复杂度也就为对数时间。这意味着我们可以用大概20次的比较在100000个查找键中找到我们想要的键。还有，如果平均比较次数是19的话，那么最差的情况也只不过为20。这太棒了。

这样，诀窍就在于当树增长或收缩时仍然能够保持树的平衡状态。否则，我们就将不得不付出更多的时间复杂度。每次我们在树的底端加入一个新的节点时，我们都必须重新平衡该树。这意味着我们要兢兢业业地从树的底部开始，重新考虑到达根节点前的每个节点。如果有两个子树的最长长度相差超过了1，那么我们还不得不重新排列一些邻近的节点以恢复原有的平衡状态。

幸运的是，有许多方法可以表示任何给定的序列。我们也有一些有用的自由度来重写树。练习实现一个平衡二叉树并不是毫无意义的，但却是可以控制的。最有意思的部分就是向树中插入节点及从树中删除节点这两个函数。在这两个操作中，它们的时间复杂度和查找操作一样，都是对数时间。惟一可惜的就是在每次插入和删除时为了保持树的平衡，我们不得不一直向上爬到树的顶端。

近似平衡树

如果我们并没有把一切都做得十分完美，但也能将一棵树保持为近似平衡（mostly balanced）的状态，那不也很好吗？如果能够给自己增加一些其他措施，我们也就可能同时加速插入和删除这两个操作。在某些允许的地方加上适当的限制，我们仍可以避免牺牲那个不错的对数级时间复杂度。事实证明我们可以从几个方面来考虑：如在操作的平均开销、代码复杂度以及最坏情况下的行为之间权衡。实际上，我们有许多种选择。关于有序树的文献多得不可思议，并且讨论深度也是让人觉得不可思议。我们在此只讲述一些基本内容。

首先，我们可以什么也不用做。无论我们如何忽视它们，在树中仍

然有大部分的地方处于平衡状态下。我们可以去除所有为维持平衡树所造成的额外开销，幸运的话，我们甚至可以一直这么做。但我们在前面就讨论了可能碰到的最坏情况。非平衡树可能对于客户代码来说很不错，但它们和STL希望提供的可重用工具的目标还是有很大的差距。

AVL树

我们同样也可以允许在每个子树的基础上有一点不平衡出现。例如，AVL树就是一个平衡二叉树，它允许在每个子树中，其左子树和右子树之间的差别为-1到+1个链接。这样累加后的开销平均下来可能会让树变得有点长，但并不是所有的都会这样。平均来说，对于这样的树重建平衡会非常快。我们通常都可以吸收由删除节点但又没有一直爬到树的顶端所造成的不平衡。

不幸的是，我们需要了解路径的长度以知道如何对每个节点进行重新平衡。至少，我们不得不往每个节点中存储一个三态对象用以记录它的两个子树深度之间的差距。对于一个AVL子树来说，它的合法值为-1、0以及+1。虽说一个占两位的对象就可以完成这件事情，但实际上，在给定现代计算机的字节填充的限制条件后，这两位将会让我们在每个节点中都消耗掉1至4个字节的存储空间。

红 - 黑树

惠普所提供的STL代码中使用了一种更复杂的技术：红 - 黑（red-black）树。实际上包括本书中的实现在内，所有现有的STL实现都是这么做的。C++标准中并没有强制要求一定要使用红 - 黑树，或其他类型的树，但这样做的确能够满足我们对于时间复杂度的要求。

红 - 黑树是一种有着三种不同类型节点的平衡树。最简单的节点类型我们已经非常熟悉了，那就是二叉树节点，它存储一个元素并最多可以指定两个子节点。但这种树还可以拥有存储两个元素并指定三个子节点的节点。如果存在这样的节点，那么中间子树中所包含的元素在排序时正好处于该节点左子树中元素以及右子树中元素的中间。最后，这种树还可以拥有有着三个元素并指定四个子节点的节点。

红 - 黑树的特点就在于它的那些大节点。如果被插入元素相邻的节点还可以变得更大，那么插入元素就会非常容易。我们只需把该节点改成为一个相应的大节点就可以了。只有在插入点已经满了的时候，我们的工作才会变得困难。我们得构造两个节点，每个都有两个元素，然后对树进行重新平衡。重新平衡一棵树是从树的底端向顶端边爬边对节点重新排列的过程，直到我们碰到的节点可以吸收掉树的改变为止。一般我们很少需要对整个树遍历一遍，以此来重新平衡它。类似地，只有在有着一个元素的节点被删除时，删除元素才会导致对树的重新平衡。在删除元素后进行重新平衡也很有可能只对树的局部产生改变。

然而，要想将上面的描述直接用C++实现出来可能会显得很粗陋。我们将不得不声明三个不同种类的节点类，或者试图仅用一种节点类来蒙

混过关而为没有用到的元素浪费大量的存储空间。相比而言，二叉树就优雅多了。当然，我们也可以将这三种不同的节点按照各自的情况表示为一个小的二叉树片段，如：有着0个、1个以及2个子节点的二叉树。用这些小树来替换原来树中的那些大节点，我们就可以把整个树再次看成是一个简单的二叉树了。当然，我们也可以把该树假设为传统的二叉树来在它里面进行查找。只有在需要插入或删除元素时，我们才有必要把节点内的链接和节点间的链接区分开来。

这时，我们就引入了红和黑两种齐名的颜色。我们将所有在大的节点内的链接描成红色，而所有在节点间的链接描成黑色。事实上，我们就画出了二叉树的节点。每个链接只指向一个节点，而所有的节点就包含树的所有存储空间。和AVL树中需要采用的两位平衡计数器不同的是，我们只需要使用一位颜色指示符。但那已经是我们所需要存储的所有额外信息了，用它来标明红-树所不能提供的信息。

还有另一种描述红-黑树的方式。可以说黑色的链接表示的是真正的树，而红色的链接所表示的则是一种引申后的树。我们想要容许在子树中的不平衡最大为2，但不能再多了。为了保证这个条件，在树产生链接时有两个限制规则：

- 增加一个红色的链接并不会改变子树的正式高度。
- 在同一行中不能同时出现两个红色的链接。

正是第二个规则使得我们可以保证不会让树太远地偏离平衡状态。它等同于我们在上面对红-黑树的最初描述中讲的对于节点大小的限制。

这种描述的好处在于：它可以使我们忘记那些实际上并不存在的可怕的、有着多个元素的大节点。我们只是在插入和删除动作中做着那些保持子树高度不变的事情，这就和在AVL树中一样。它们两者之间的不同仅在于遵循的规则有了一些改变，这种改变会降低插入和删除的成本。注意，对于所有的操作来说，时间复杂度还是维持在对数级别上。我们

在完成那些复杂的重新平衡操作时所付出的开销通常都很小，所得到的回报就是节省了执行所需的时间。并且我们仍然可以控制对于树的不同

这两个模板类都存储了值类型为Key的元素；它们也都是按类型为Pred的函数对象排序的，这一点同我们在前面对于假想的有序向量和有序双队列类的描述一样；它们还都有一个普遍出现在STL容器中的分配器参数。如同我们在前面章节中详细描述的一样，容器使用类A的对象来分配和释放存储空间。我们在本章的前面早就强调过，这两个模板类的本质区别在于：多重集合允许其中的元素有相等次序，但集合则不允许这种现象出现。

map
multimap

另外两种我们所知道的关联容器就是map和multimap。我们将在下一章对它们进行详细讨论。

关联容器中定义了可以利用它们施加到被控序列上的次序的成员函数。它们还省略了那些对于一个有序容器来说没有意义（或意义很小）的成员函数。我们将不会发现成员函数push_front、pop_back，甚至front。我们也不会再发现指定某些值的副本的构造函数和成员函数，如：assign(3, 'x')。我们所能找到的成员函数如下：

insert
erase
find
lower_bound
upper_bound
equal_range
count

- insert(const Key&)将一个给定键值的元素插入到容器中。
- erase(const Key&)删除一个与给定键值相匹配的元素。
- find(const Key&)在容器中查找一个和给定键值相匹配的元素。
- lower_bound(const Key&)在容器中查找第一个等于（或大于）给定键值的元素。
- upper_bound(const Key&)在容器中查找最后一个等于（或小于）给定键值的元素。
- equal_range(const Key&)在容器中查找一个所有元素都与给定键值相等的子区间。
- count(const Key&)在容器中计算出所有与给定键值相等的元素的个数。

在本章的“使用<set>”小节中，我们会详细讨论这些成员函数。

功能描述

```
namespace std {
    template<class Key, class Pred, class A>
        class set;
    template<class Key, class Pred, class A>
        class multiset;

        // TEMPLATE FUNCTIONS
    template<class Key, class Pred, class A>
        bool operator==(const set<Key, Pred, A>& lhs,
                           const set<Key, Pred, A>& rhs);
```

```
template<class Key, class Pred, class A>
    bool operator==(const multiset<Key, Pred, A>& lhs,
                      const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(const set<Key, Pred, A>& lhs,
                      const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=!(const multiset<Key, Pred, A>& lhs,
                      const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(const set<Key, Pred, A>& lhs,
                      const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<!(const multiset<Key, Pred, A>& lhs,
                      const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(const set<Key, Pred, A>& lhs,
                      const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>!(const multiset<Key, Pred, A>& lhs,
                      const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(const set<Key, Pred, A>& lhs,
                      const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(const set<Key, Pred, A>& lhs,
                      const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(const multiset<Key, Pred, A>& lhs,
                      const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap()
```

```
        set<Key, Pred, A>& lhs,
        set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
void swap(
    multiset<Key, Pred, A>& lhs,
    multiset<Key, Pred, A>& rhs);
};
```

包含STL的标准头文件<set>可以得到模板类set和multiset，以及几个支持模板的定义。

¶ multiset

```
template<class Key, class Pred = less<Key>,
         class A = allocator<Key> >
class multiset {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;

    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
multiset();
explicit multiset(const Pred& comp);
multiset(const Pred& comp, const A& al);
multiset(const multiset& x);
template<class InIt>
    multiset(InIt first, InIt last);
template<class InIt>
    multiset(InIt first, InIt last,
             const Pred& comp);
template<class InIt>
    multiset(InIt first, InIt last,
             const Pred& comp, const A& al);
```

```

        const_iterator begin() const;
        const_iterator end() const;
        const_reverse_iterator rbegin() const;
        const_reverse_iterator rend() const;
        size_type size() const;
        size_type max_size() const;
        bool empty() const;
        A get_allocator() const;
        iterator insert(const value_type& x);
        iterator insert(iterator it, const value_type& x);
        template<class InIt>
            void insert(InIt first, InIt last);
        iterator erase(iterator it);
        iterator erase(iterator first, iterator last);
        size_type erase(const Key& key);
        void clear();
        void swap(multiset& x);
        key_compare key_comp() const;
        value_compare value_comp() const;
        const_iterator find(const Key& key) const;
        size_type count(const Key& key) const;
        const_iterator lower_bound(const Key& key) const;
        const_iterator upper_bound(const Key& key) const;
        pair<const_iterator, const_iterator>
            equal_range(const Key& key) const;
    };

```

该模板类描述的对象控制一个元素类型为const Key的可变长度序列。该序列以谓词Pred为排序依据。每个元素既是排序键，又是存储的值。该序列的表示方式允许以与序列中元素个数的对数成比例的时间对它的任意元素进行查找、插入以及删除等操作。而且，插入元素并不会导致任何迭代器无效，而删除元素也只会让原来指向被删除元素的迭代器无效。

对象通过调用它存储的类型为Pred的函数对象来对它控制的序列进行排序。我们可以通过成员函数key_comp()来存取这个对象。该函数对象必须在类型为Key的排序键上施加严格弱序。对于序列中任意排在y前面的元素x，key_comp()(y, x)都为false。（对于默认的函数对象less<Key>来说，排序键从不以降序的方式存在。）与模板类set不同的是，模板类multiset的对象中并不保证key_comp()(x, y)一定为true。（因为键不需要一定是惟一的。）

multiset对象是通过存储于其内部的一个类型为A的分配器对象来进行存储空间的分配及释放的。该分配器对象必须拥有和模板类allocator

一样的外部接口。注意，当对容器赋值时，存储的分配器对象并没有被复制。

口 multiset::allocator_type

typedef A allocator_type;

该类型是模板参数A的同义词。

口 multiset::begin

const_iterator begin() const;

该成员函数返回一个双向迭代器，指向被控序列的第一个元素（如果被控序列为空，则指向紧接着该序列末端的下一个位置）。

口 multiset::clear

void clear();

该成员函数调用erase(begin(), end())。

口 multiset::const_iterator

typedef T1 const_iterator;

该类型描述的对象可以作为一个指向被控序列的常量双向迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。

口 multiset::const_pointer

typedef typename A::const_pointer const_pointer;

该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。

口 multiset::const_reference

typedef typename A::const_reference const_reference;

该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。

口 multiset::const_reverse_iterator

typedef reverse_iterator<const_iterator>

const_reverse_iterator;

该类型描述的对象可以作为一个指向被控序列的常量反型双向迭代器来使用。

口 multiset::count

size_type count(const Key& key) const;

该成员函数返回区间[lower_bound(key), upper_bound(key))中元素x的个数。

口 multiset::difference_type

typedef T3 difference_type;

该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。

- ❑ multiset::empty


```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。
- ❑ multiset::end


```
const_iterator end() const;
```

该成员函数返回一个双向迭代器，指向紧接着被控序列末端的下一个位置。
- ❑ multiset::equal_range


```
pair<const_iterator, const_iterator>
equal_range(const Key& key) const;
```

该成员函数返回一个迭代器对x，其中x.first == lower_bound(key),
x.second == upper_bound(key)。
- ❑ multiset::erase


```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

第一个成员函数删除被控序列中由it所指定的元素。第二个成员函数删除区间[first, last)中的所有元素。这两个函数都返回一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

第三个成员函数删除区间[lower_bound(key), upper_bound(key))中具有给定排序键的所有元素。它返回所删除元素的个数。

这些成员函数不会抛出任何异常。
- ❑ multiset::find


```
const_iterator find(const Key& key) const;
```

该成员函数返回一个迭代器，它指向被控序列中排序键与key次序相等的第一个元素。如果不存在这样的元素，函数将返回end()。
- ❑ multiset::get_allocator


```
A get_allocator() const;
```

该成员函数返回存储的分配器对象。
- ❑ multiset::insert


```
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
void insert(InIt first, InIt last);
```

第一个成员函数向被控序列中插入一个元素x，然后返回一个指向被插入元素的迭代器。第二个成员函数返回insert(x)，用it作为被控序列中的起始位置开始查找插入点。（如果插入点紧接着it的话，插入操作执行

的时间将是被分摊付出的常数时间，而不是对数时间。) 第三个成员函数通过对区间[first, last)中的每个it调用insert(*it)，将整个区间中的元素都插入到被控序列中。

如果在插入单个元素时有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。如果在插入多个元素时有异常抛出，容器将保持为一个稳定但未指定的状态，并且将异常继续向外抛出。

口 multiset::iterator

```
typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的双向迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

口 multiset::key_comp

```
key_compare key_comp() const;
```

该成员函数返回存储的函数对象，用来对被控序列中的元素排序。该函数对象定义了成员函数：

```
bool operator(const Key& x, const Key& y);
```

如果x按照严格的排序规则出现在y之前，它就会返回true。

口 multiset::key_compare

```
typedef Pred key_compare;
```

该类型描述的函数对象可以用来比较两个排序键，以检测被控序列中两个元素之间的相对次序。

口 multiset::key_type

```
typedef Key key_type;
```

该类型描述了被控序列的每个元素中存储的排序键对象。

口 multiset::lower_bound

```
const_iterator lower_bound(const Key& key) const;
```

该成员函数返回一个迭代器，指向被控序列中第一个让key_comp()(x, key)为false的元素x。如果不存在这样的元素，函数将返回end()。

口 multiset::multiset

```
multiset();
explicit multiset(const Pred& comp);
multiset(const Pred& comp, const A& al);
multiset(const multiset& x);
template<class InIt>
multiset(InIt first, InIt last);
template<class InIt>
multiset(InIt first, InIt last,
         const Pred& comp);
template<class InIt>
multiset(InIt first, InIt last,
```

```
const Pred& comp, const A& al);
```

所有的构造函数都会向对象中存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，分配器对象将会为A()。

上述所有构造函数都还存储一个函数对象，我们可以随后通过调用key_comp()来得到这个函数对象。如果有的话，该函数对象就是参数comp。对于复制构造函数来说，它将是x.key_comp()。否则，该函数对象就是Pred()。

开始的三个构造函数会指定空的被控序列。第四个构造函数指定由x控制的序列的拷贝。最后三个构造函数指定[first, last)中的元素序列。

口 multiset::max_size

```
size_type max_size() const;
```

该成员函数返回容器对象所能控制的最长序列的长度。

口 multiset::ppointer

```
typedef typename A::pointer pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 multiset::rbegin

```
const_reverse_iterator rbegin() const;
```

该成员函数返回一个反转型双向迭代器，它指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 multiset::reference

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 multiset::rend

```
const_reverse_iterator rend() const;
```

该成员函数返回一个反转型双向迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 multiset::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型双向迭代器来使用。

口 multiset::size

```
size_type size() const;
```

该成员函数返回被控序列的长度。

口 multiset::size_type

```
typedef T2 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

口 multiset::swap

```
void swap(multiset& x);
```

该成员函数在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator(), 它将可以在常数时间内完成交换。只有在复制被存储的类型为Pred的函数对象时, 它才有可能抛出异常。另外, 它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则, 它调用元素的构造函数以及为元素赋值的次数与这两个被控序列的元素个数成比例。

口 multiset::upper_bound

```
const_iterator upper_bound(const Key& key) const;
```

该成员函数返回一个迭代器, 指向被控序列中第一个使key_comp()(key, x) 为true的元素x。如果不存在这样的元素, 它将返回end()。

口 multiset::value_comp

```
value_compare value_comp() const;
```

该成员函数返回一个用以检测被控序列中元素之间次序的函数对象。

口 multiset::value_compare

```
typedef Pred value_compare;
```

该类型描述是一个函数对象, 它可以把被控序列中的两个元素作为排序键来比较, 并检测它们之间的相对次序。

口 multiset::value_type

```
typedef Key value_type;
```

该类型描述被控序列中的一个元素。

口 operator!=

```
template<class Key, class Pred, class A>
bool operator!=(
    const set <Key, Pred, A>& lhs,
    const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
bool operator!=(
    const multiset <Key, Pred, A>& lhs,
    const multiset <Key, Pred, A>& rhs);
```

该模板函数返回!(lhs == rhs)。

```
# operator==

template<class Key, class Pred, class A>
bool operator==(const set<Key, Pred, A>& lhs,
                  const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
bool operator==(const multiset<Key, Pred, A>& lhs,
                  const multiset<Key, Pred, A>& rhs);
```

第一个模板函数重载operator==来比较模板类set的两个对象。第二个模板函数重载operator==以比较模板类multiset的两个对象。这两个函数都返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())。

```
# operator<

template<class Key, class Pred, class A>
bool operator<(const set<Key, Pred, A>& lhs,
                  const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
bool operator<(const multiset<Key, Pred, A>& lhs,
                  const multiset<Key, Pred, A>& rhs);
```

第一个模板函数重载operator<来比较模板类set的两个对象。第二个模板函数重载operator<以比较模板类multiset的两个对象。这两个函数都返回lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

```
# operator<=

template<class Key, class Pred, class A>
bool operator<=(const set<Key, Pred, A>& lhs,
                  const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
bool operator<=(const multiset<Key, Pred, A>& lhs,
                  const multiset<Key, Pred, A>& rhs);
```

该模板函数返回!(rhs < lhs)。

```
# operator>

template<class Key, class Pred, class A>
bool operator>(const set<Key, Pred, A>& lhs,
                  const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
bool operator>(const multiset<Key, Pred, A>& lhs,
```

```
    const multiset <Key, Pred, A>& rhs);
```

该模板函数返回rhs < lhs。

口 operator>=

```
template<class Key, class Pred, class A>
    bool operator>=(  
        const set <Key, Pred, A>& lhs,  
        const set <Key, Pred, A>& rhs);  
template<class Key, class Pred, class A>
    bool operator>=(  
        const multiset <Key, Pred, A>& lhs,  
        const multiset <Key, Pred, A>& rhs);
```

该模板函数返回!(lhs < rhs)。

口 set

```
template<class Key, class Pred = less<Key>,
         class A = allocator<Key> >
    class set {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    set();
    explicit set(const Pred& comp);
    set(const Pred& comp, const A& al);
    set(const set& x);
    template<class InIt>
        set(InIt first, InIt last);
    template<class InIt>
        set(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
```

```

        set(Init first, Init last,
            const Pred& comp, const A& al);
    const_iterator begin() const;
    const_iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class Init>
        void insert(Init first, Init last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(set& x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    const_iterator lower_bound(const Key& key) const;
    const_iterator upper_bound(const Key& key) const;
    pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};


```

该模板类描述的对象控制一个元素类型为const Key的可变长度序列。该序列按谓词Pred排序。每个元素既是排序键，又是存储的值。该序列的表示方式允许我们以与序列中元素个数的对数成比例的时间对它里面的任意元素进行查找、插入以及删除等操作。而且，插入元素并不会导致任何迭代器无效，而删除元素也只会让原来指向被删除元素的迭代器无效。

对象通过调用它存储的类型为Pred的函数对象来对它控制的序列进行排序。我们可以通过成员函数key_comp()来存取这个对象。该函数对象必须在类型为Key的排序键上施加严格弱序。对于序列中任意排在y前面的元素x，key_comp()(y, x)都为false。（对于默认的函数对象less<Key>来说，排序键从不以降序的方式存在。）与模板类multiset不同的是，模板类set的对象保证key_comp()(x, y)一定为true。（因为键是惟一的。）

set对象是通过存储的一个类型为A的分配器对象来进行存储空间的

分配及释放的。该分配器对象必须拥有和模板类allocator一样的外部接口。注意，当对容器进行赋值时，存储的分配器对象不会被复制。

- `set::allocator_type`
`typedef A allocator_type;`
该类型是模板参数A的同义词。
- `set::begin`
`const_iterator begin() const;`
该成员函数返回一个双向迭代器，它指向被控序列的第一个元素（如果被控序列为空，则指向紧接着该序列末端的下一个位置）。
- `set::clear`
`void clear();`
该成员函数调用`erase(begin(), end())`。
- `set::const_iterator`
`typedef T1 const_iterator;`
该类型描述的对象可以作为一个指向被控序列的常量双向迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。
- `set::const_pointer`
`typedef typename A::const_pointer const_pointer;`
该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。
- `set::const_reference`
`typedef typename A::const_reference const_reference;`
该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。
- `set::const_reverse_iterator`
`typedef reverse_iterator<const_iterator>`
`const_reverse_iterator;`
该类型描述的对象可以作为一个指向被控序列的常量反转型双向迭代器来使用。
- `set::count`
`size_type count(const Key& key) const;`
该成员函数返回区间`[lower_bound(key), upper_bound(key)]`中元素x的个数。
- `set::difference_type`
`typedef T3 difference_type;`
该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。

- `set::empty`
`bool empty() const;`
当被控序列为空时，该成员函数返回true。
- `set::end`
`const_iterator end() const;`
该成员函数返回一个双向迭代器，指向紧接着被控序列末端的下一个位置。
- `set::equal_range`
`pair<const_iterator, const_iterator>`
`equal_range(const Key& key) const;`
该成员函数返回一个迭代器对x，其中`x.first == lower_bound(key)`,
`x.second == upper_bound(key)`。
- `set::erase`
`iterator erase(iterator it);`
`iterator erase(iterator first, iterator last);`
`size_type erase(const Key& key);`
第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除区间[first, last)中的所有元素。这两个函数都返回一个迭代器，指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。
第三个成员函数删除区间[lower_bound(key), upper_bound(key))中具有给定排序键的所有元素。它返回所删除的元素个数。
这些成员函数不会抛出任何异常。
- `set::find`
`const_iterator find(const Key& key) const;`
该成员函数返回一个迭代器，指向被控序列中排序键与key相等的元素。如果不存在这样的元素，函数将会返回end()。
- `set::get_allocator`
`A get_allocator() const;`
该成员函数返回存储的分配器对象。
- `set::insert`
`pair<iterator, bool> insert(const value_type& x);`
`iterator insert(iterator it, const value_type& x);`
`template<class InIt>`
`void insert(InIt first, InIt last);`
第一个成员函数先检测被控序列中是否存在其键与x有相等次序的元素y。如果没有的话，它就会创建一个这样的元素y，并把x作为它的初始值。函数然后就会让迭代器 it指向y。如果有插入发生，函数将返回

`pair(it, true)`。否则，它返回`pair(it, false)`。

第二个成员函数返回`insert(x)`，用`it`作为被控序列中的起始位置开始查找插入点。（如果插入点紧接着`it`的话，插入操作执行的时间将是被分摊付出的常数时间，而不是对数时间。）第三个成员函数通过对区间`[first, last)`中的每个`it`调用`insert(*it)`，将整个区间中的元素都插入到被控序列中。

如果在插入单个元素时有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。如果在插入多个元素时有异常抛出，容器将保持为一个稳定但未指定的状态，并且将异常继续向外抛出。

口 `set::iterator`

```
typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的双向迭代器来使用。在此处它被描述为由实现定义的类型`T0`的同义词。

口 `set::key_comp`

```
key_compare key_comp() const;
```

该成员函数返回存储的函数对象，用来对被控序列中的元素排序。该函数对象定义了成员函数：

```
bool operator(const Key& x, const Key& y);
```

如果`x`按照严格的排序规则出现在`y`之前，它就会返回`true`。

口 `set::key_compare`

```
typedef Pred key_compare;
```

该类型描述的函数对象可以用来比较两个排序键，以检测被控序列中两个元素之间的相对次序。

口 `set::key_type`

```
typedef Key key_type;
```

该类型描述了构成被控序列中的每个元素的排序键对象。

口 `set::lower_bound`

```
const_iterator lower_bound(const Key& key) const;
```

该成员函数返回一个迭代器，指向被控序列中第一个让`key_comp()(x, key)`为`false`的元素`x`。如果不存在这样的元素，函数将返回`end()`。

口 `set::max_size`

```
size_type max_size() const;
```

该成员函数返回容器对象所能控制的最长序列的长度。

口 `set::pointer`

```
typedef typename A::pointer pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 `set::rbegin`

```
const_reverse_iterator rbegin() const;
```

该成员函数返回一个反转型双向迭代器，指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 `set::reference`

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 `set::rend`

```
const_reverse_iterator rend() const;
```

该成员函数返回一个反转型双向迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 `set::reverse_iterator`

```
typedef reverse_iterator<iterator> reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型双向迭代器来使用。

口 `set::set`

```
set();
explicit set(const Pred& comp);
set(const Pred& comp, const A& al);
set(const set& x);
template<class InIt>
set(InIt first, InIt last);
template<class InIt>
set(InIt first, InIt last,
    const Pred& comp);
template<class InIt>
set(InIt first, InIt last,
    const Pred& comp, const A& al);
```

所有的构造函数都会存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，分配器对象将会为A()。

上述所有构造函数还会存储一个函数对象，我们可以随后通过调用key_comp()来得到这个函数对象。如果有的话，该函数对象就是参数comp。对于复制构造函数来说，它将是x.key_comp()。如果构造函数中没有提供参数comp的话，那么该函数对象就是Pred()。

开始的三个构造函数会指定空的被控序列。第四个构造函数则指定

由x控制的序列的拷贝。最后三个构造函数指定(first, last)中的元素序列。

口 set::size

size_type size() const;

该成员函数返回被控序列的长度。

口 set::size_type

typedef T2 size_type;

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

口 set::swap

void swap(set& x);

该成员函数在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator(), 它将可以在常数时间内完成交换。只有在复制被存储的类型为Pred的函数对象时，它才有可能抛出异常。另外，它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则，它将以与这两个被控序列的元素个数成比例的次数调用元素的构造函数以及为元素赋值。

口 set::upper_bound

const_iterator upper_bound(const Key& key) const;

该成员函数返回一个迭代器，指向被控序列中第一个使key_comp()(key, x)为true的元素x。如果不存在这样的元素，它将返回end()。

口 set::value_comp

value_compare value_comp() const;

该成员函数返回一个用以检测被控序列中元素之间次序的函数对象。

口 set::value_compare

typedef Fred value_compare;

该类型描述的是一个函数对象，它可以把被控序列中的两个元素作为排序键来比较，并检测它们之间的相对次序。

口 set::value_type

typedef Key value_type;

该类型描述被控序列中的一个元素。

口 swap

template<class Key, class Pred, class A>

void swap(

multiset <Key, Pred, A>& lhs,

multiset <Key, Pred, A>& rhs);

template<class Key, class Pred, class A>

```
void swap(
    set <Key, Pred, A>& lhs,
    set <Key, Pred, A>& rhs);
```

该模板函数执行lhs.swap(rhs)。

使用 `<set>`

`set` 如果想在程序中使用模板类`set`或模板类`multiset`，请把头文件`<set>`包含到程序中。只有在允许出现多个元素有着相同的键值时——或者更确切地说，成对的元素有相等次序时，我们才使用后一个模板类。由于这两个容器很相似，所以我们开始会把`set`作为一个例子来描述。在本节的后面部分，我们将着重描述它们之间的不同之处。

我们也可以指定`set`存储类型为`const Key`的元素以及类型为`less<Key>`的排序规则，我们只需这样写一条类型定义语句：

```
typedef set<Key, less<Key>, allocator<Key> > Mycont;
```

通过使用默认的模板参数，可以省略第二个和第三个模板参数。

`less` 如果我们省略了第二个参数，默认的函数对象类型将是`less<Key>`。容器中所存储的被控序列也就是以这种规则来排序的。也就是说，至少对于数字类型来说，序列中最后的那个元素将会有最大的值。

模板类`set`支持所有我们在第9章中所给出的、对于容器来说是常见的一些操作。（具体可以参见第9章的“使用容器”一节开始的讨论。）我们在此仅概括出模板类`set`所特有的属性。

构造函数 为了构造一个类`set<Key, Pred, A>`的对象，可以这么写：

- `set()`，声明一个空的按`Pred()`排序的集合；

- `set(pr)`，声明一个空的按函数对象`pr`排序的集合；

- `set(pr, al)`，声明一个和上面一样的集合，不过它还存储一个分配器对象`al`；

- `set(first, last)`，声明一个按`Pred()`排序的集合，它的初始内容是由`[first, last]`指定的序列（当然，这是一个有序序列）中复制过来的；

- `set(first, last, pr)`，声明一个按`pr`排序的集合，它的初始内容是由`[first, last]`指定的序列（和上面一样，该序列是一个有序序列）中复制过来的；

- `set(first, last, pr, al)`，声明一个和上面一样的集合，不过它还存储一个分配器对象`al`。

如果我们已经将该模板类的分配器类型指定为`allocator<Key>`（这是我们所常做的并且也是STL提供的默认值），那么再在构造函数中显式地

指定分配器参数 `ai` 并不能给我们带来任何好处。仅对于程序中显式定义的某些分配器，这样的参数才会起作用。（参见第4章中对于分配器的讨论。）

下面所有的描述中都假定 `cont` 是类 `set<Key, Pred, A>` 的一个对象。

`clear`

为了删除所有的元素，请调用 `cont.clear()`。

`erase`

为了删除由 `it` 所指定的元素，请调用 `cont.erase(it)`。它返回一个迭代器，指向被删除元素的下一个元素。为了删除由区间 `[first, last)` 指定的所有元素，请调用 `cont.erase(first, last)`。

我们也可以执行一些能够从 `set` 的独一无二的表达方式中获益的操作。尤其是，我们还可以通过调用 `cont.erase(key)` 来删除键（值）与 `key` 有相等次序的元素。其他和 `set`（以及其他关联容器）相关的特殊操作有：

`insert`

为了插入一个键（值）为 `key` 的元素，请调用 `cont.insert(key)`。它的返回值是类 `pair<iterator, bool>` 的一个对象 `ans`。只有当被控序列中没有与 `key` 次序相等的元素时，成员对象 `ans.second` 才会为 `true`。在这种情况下，`ans.first` 就指向被插入的元素。否则，插入将会失败并且 `ans.first` 仍然指向它原来指向的元素。

为了插入区间 `[first, last)` 中所有的元素，请调用 `cont.insert(first, last)`。被插入的序列不能为最初被控序列中的一部分。

为了向 `it` 所指定的元素后面插入一个键（值）为 `key` 的元素，请调用 `cont.insert(it, key)`。如果“提示”被证明为正确的话，插入将会耗费常数时间而不是对数时间。不管在何种情况下，这样的调用和调用 `insert(key)` 都是一样的，并且它们都返回相同的迭代器值。

`find`

为了在被控序列中查找一个键（值）与 `key` 次序相等的元素，请调用 `cont.find(key)`。

`lower_bound`

为了在被控序列中查找到第一个键（值）未排在 `key` 之前的元素，请调用 `cont.lower_bound(key)`。注意，它也就是所有与 `key` 有相等次序的元素形成的子区间的起始处。

`upper_bound`

为了在被控序列中查找到第一个键（值）排在 `key` 之后的元素，请调用 `cont.upper_bound(key)`。注意，它也就紧接着所有与 `key` 有相等次序的元素形成的子区间的末端。

`equal_range`

为了检测出被控序列中所有元素都与 `key` 次序相等的子区间，请调用 `cont.equal_range(key)`。注意，它返回一个这样的迭代器对：`pair<iterator, iterator>(cont.lower_bound(key), cont.upper_bound(key))`。对于一个 `set<key>` 对象来说，该子区间的长度要么为 0，要么就为 1。

`count`

为了检测出调用 `cont.equal_range(key)` 所得到的子区间的长度，请调用 `cont.count(key)`。对于一个 `set<key>` 对象来说，它的返回值要么为 0，要么就为 1。

`key_comp`

为了获得以和 `cont` 中同样的规则对键排序的对象，请调用

value_comp
multiset

insert
find
equal_range
count

Tree

`cont.key_comp()`。需要指出的是，如果在被控序列中，`key1`排在`key2`的前面，那么`cont.key_comp()(key1, key2)`就为`true`。

为了获得一个可以对`cont`元素存储的值进行排序的函数对象，请调用`cont.value_comp()`。对于类型为`set<key>`的对象`cont`来说，它的返回值和`cont.key_comp()`的一样。

模板类`set`和模板类`multiset`之间的不同之处在于它们最基本的区别——`multiset`可以存储次序相等的元素，而`set`则不行。它们所有的构造函数以及成员函数的函数签名都一样。但对于`multiset`对象来说：

调用`cont.insert(key)`总是会插入一个新元素，所以函数返回的就是指向这个新元素的迭代器，而不是类`pair<iterator, bool>`的对象。

如果有许多元素与`key`次序相等，调用`cont.find(key)`返回的迭代器将指向被控序列中第一个与`key`次序相等的元素。

调用`cont.equal_range(key)`所得到的子序列可以有区间`[0, cont.size()]`中的任意长度。

同样，调用`cont.count(key)`所得到的子序列可以有区间`[0, cont.size()]`中的任意值。

实现`<set>`

`set`、`multiset`、`map`以及`multimap`之间的相似之处要多于它们之间的不同之处。它们都是以红-黑树来存储所控制序列的。它们最主要的区别来自于下面的两个选择：

- 被存储的元素只是键（这时选择`set`和`multiset`），还是一个{键，值}对（这时选择`map`和`multimap`）。
- 它们中两个被存储的元素可以有相等次序的键（这时选择`multiset`和`multimap`），还是不允许出现这种情况（这时选择`set`和`map`）。

于是，很自然的事情就是写出一个常用的基础实现，使得它可以接受这两个选择的任何组合。

本实现提供了一个模板类`Tree`，它将容器实现为一个红-黑树。它也被设计成为可以作为模板类`set`、`multiset`、`map`以及`multimap`的基类。`Tree`只需很少的参数。为了让模板参数列表更容易管理，我们遵循了通常大家都使用的方法：引入一个“traits”类，使得我们可以用一个句柄传递不同类型的参数。`traits`对象也可以带有一个或多个可能会在执行期使用的对象，如：用于比较的函数对象。

也就是说，模板类`Tree`被简单地声明为：

```
template<class Tr>
class Tree;
```

此时我们用Tr来传递一大堆的参数。模板类Tree期望参数Tr有许多属性。下面就是一个具有代表性的为Tree服务的traits类，我们把它叫做tree_traits：

```
tree_traits {
    struct tree_traits {
        typedef T1 key_type;
        typedef T2 value_type;
        typedef T3 allocator_type;
        enum {Multi = <allow equivalent keys>};
        typedef T4 key_compare;
        typedef T5 value_compare;
        struct Kfn {
            const key_type& operator()(const value_type&) const;
        };
        key_compare comp;
        tree_traits(key_compare);
    };
}
```

更确切地说：

- key_type是每个元素所存储的值中作为键的那部分的类型。
- value_type是每个元素所存储的整个值的类型。
- allocator_type是用来为被控序列分配和释放存储空间的分配器的类型。
 - 如果两个被存储的元素可以有相等次序的键，Multi就是一个非零常数值。
 - key_compare是函数对象prk的类型，该函数对象可以比较两个类型为key_type的键，如prk(key1, key2)。
 - value_compare是函数对象prv的类型，该函数对象可以比较两个类型为value_type的值，如prv(val1, val2)。
 - Kfn是函数对象prx的类型，该函数对象可以从类型为value_type的值中提取类型为key_type的键，如prx(val)。
 - comp是存储的类型为key_compare的函数对象，该类型的函数可以用来进行真正的键比较。
 - tree_traits(key_compare)是一个构造函数，该构造函数可以用参数来初始化被存储的函数对象comp。

在本章的后面，我们将会看到这个traits类的不同版本，它使Tree的行为和set或multiset类似。在下一章，我们将会看到更多的traits版本用以使Tree的行为和map或multimap类似。

list中的机制。（参见第11章。）

```
程序清单 13-1: // tree internal header
xtree
#ifndef XTREE_
#define XTREE_
#include <functional>
#include <memory>
#include <stdexcept>
namespace std {

    // TEMPLATE CLASS Tree_nod
    template<class Tr>
        class Tree_nod : public Tr {
protected:
    typedef typename Tr::allocator_type allocator_type;
    typedef typename Tr::key_compare key_compare;
    typedef typename Tr::value_type value_type;
    typedef typename allocator_type::template
        rebind<void>::other::pointer Genptr;
    struct Node;
    friend struct Node;
    struct Node {
        Genptr Left, Parent, Right;
        value_type Value;
        char Color, Isnil;
    };
    Tree_nod(const key_compare& Parg,
              allocator_type Al)
        : Tr(Parg), Alnod(Al) {}
    typename allocator_type::template
        rebind<Node>::other
        Alnod;
};

    // TEMPLATE CLASS Tree_ptr
    template<class Tr>
        class Tree_ptr : public Tree_nod<Tr> {
protected:
    typedef typename Tree_nod<Tr>::Node Node;
    typedef typename Tr::allocator_type allocator_type;
    typedef typename Tr::key_compare key_compare;
    typedef typename allocator_type::template
        rebind<Node>::other::pointer Nodeptr;
    Tree_ptr(const key_compare& Parg,
```

```
allocator_type Al)
: Tree_nod<Tr>(Parg, Al), Alptr(Al) {}
typename allocator_type::template
    rebind<Nodeptr>::other
        Alptr;
};

// TEMPLATE CLASS Tree_val
template<class Tr>
class Tree_val : public Tree_ptr<Tr> {
protected:
    typedef typename Tr::allocator_type allocator_type;
    typedef typename Tr::key_compare key_compare;
    Tree_val(const key_compare& Parg,
              allocator_type Al)
        : Tree_ptr<Tr>(Parg, Al), Alval(Al) {}
    allocator_type Alval;
};

// TEMPLATE CLASS Tree
template<class Tr>
class Tree
    : public Tree_val<Tr> {
public:
    typedef Tree<Tr> Myt;
    typedef Tree_val<Tr> Mybase;
    typedef typename Tr::key_type key_type;
    typedef typename Tr::key_compare key_compare;
    typedef typename Tr::value_compare value_compare;
    typedef typename Tr::value_type value_type;
    typedef typename Tr::allocator_type allocator_type;
protected:
    typedef typename Tree_nod<Tr>::Genptr Genptr;
    typedef typename Tree_nod<Tr>::Node Node;
    enum Redbl {Red, Black};
    typedef typename allocator_type::template
        rebind::<Node>::other::pointer Nodeptr;
    typedef typename allocator_type::template
        rebind::<Nodeptr>::other::reference Nodepref;
    typedef typename allocator_type::template
        rebind::<key_type>::other::const_reference Keyref;
    typedef typename allocator_type::template
        rebind::<char>::other::reference Charref;
    typedef typename allocator_type::template
        rebind::<value_type>::other::reference Vref;
```

```
static Charref Color(Nodeptr P)
    {return ((Charref)(*P).Color); }
static Charref Isnil(Nodeptr P)
    {return ((Charref)(*P).Isnil); }
static Keyref Key(Nodeptr P)
    {return (Kfn()(Value(P))); }
static Nodepref Left(Nodeptr P)
    {return ((Nodepref)(*P).Left); }
static Nodepref Parent(Nodeptr P)
    {return ((Nodepref)(*P).Parent); }
static Nodepref Right(Nodeptr P)
    {return ((Nodepref)(*P).Right); }
static Vref Value(Nodeptr P)
    {return ((Vref)(*P).Value); }

public:
    typedef typename allocator_type::size_type size_type;
    typedef typename allocator_type::difference_type Dift;
    typedef Dift difference_type;
    typedef typename allocator_type::template
        rebind::<value_type>::other::pointer Tptr;
    typedef typename allocator_type::template
        rebind::<value_type>::other::const_pointer Ctptr;
    typedef typename allocator_type::template
        rebind::<value_type>::other::reference Reft;
    typedef Tptr pointer;
    typedef Ctptr const_pointer;
    typedef Reft reference;
    typedef typename allocator_type::template
        rebind::<value_type>::other::const_reference
        const_reference;

    // CLASS iterator
    class iterator;
    friend class iterator;
    class iterator : public Bidit<value_type, Dift,
        Tptr, Reft> {
public:
    typedef Bidit<value_type, Dift,
        Tptr, Reft> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
//    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
```

```
typedef typename Mybase::pointer pointer;
typedef typename Mybase::reference reference;
iterator()
    : Ptr(0) {}
iterator(Nodeptr P)
    : Ptr(P) {}
reference operator*() const
    {return (Value(Ptr)); }
Tptr operator->() const
    {return (&**this); }
iterator& operator++()
    {Inc();
     return (*this); }
iterator operator++(int)
    {iterator Tmp = *this;
     ++*this;
     return (Tmp); }
iterator& operator--()
    {Dec();
     return (*this); }
iterator operator--(int)
    {iterator Tmp = *this;
     --*this;
     return (Tmp); }
bool operator==(const iterator& X) const
    {return (Ptr == X.Ptr); }
bool operator!=(const iterator& X) const
    {return (!(*this == X)); }
void Dec()
    {if (Isnil(Ptr))
        Ptr = Right(Ptr);
     else if (!Isnil(Left(Ptr)))
        Ptr = Max(Left(Ptr));
     else
        {Nodeptr P;
         while (!Isnil(P = Parent(Ptr)))
             && Ptr == Left(P))
            Ptr = P;
         if (!Isnil(P))
            Ptr = P; }
    }
void Inc()
    {if (Isnil(Ptr))
        ;
     else if (!Isnil(Right(Ptr)))
        Ptr = Min(Right(Ptr)); }
```

```
        else
            {Nodeptr P;
             while (!Isnil(P = Parent(Ptr))
                   && Ptr == Right(P))
                 Ptr = P;
             Ptr = P; } }
        Nodeptr Mynode() const
        {return (Ptr); }
protected:
    Nodeptr Ptr;
};

// CLASS const_iterator
class const_iterator;
friend class const_iterator;
class const_iterator : public Bidit<value_type, Dift,
    Ctptr, const_reference> {
public:
    typedef Bidit<value_type, Dift,
        Ctptr, const_reference> Mybase;
    typedef typename Mybase::iterator_category
        iterator_category;
//    typedef typename Mybase::value_type value_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::reference reference;
    const_iterator()
        : Ptr(0) {}
    const_iterator(Nodeptr P)
        : Ptr(P) {}
    const_iterator(const typename
        Tree<Tr>::iterator& X)
        : Ptr(X.Mynode()) {}
        const_reference operator*() const
        {return (Value(Ptr)); }
        Ctptr operator->() const
        {return (&**this); }
    const_iterator& operator++()
        {Inc();
         return (*this); }
    const_iterator operator++(int)
        {const_iterator Tmp = *this;
         ++*this;
         return (Tmp); }
```

```
const_iterator& operator--()
    {Dec();
     return (*this); }
const_iterator operator--(int)
    {const_iterator Tmp = *this;
     --*this;
     return (Tmp); }
bool operator==(const const_iterator& X) const
    {return (Ptr == X.Ptr); }
bool operator!=(const const_iterator& X) const
    {return (!(*this == X)); }
void Dec()
    {if (Isnil(Ptr))
        Ptr = Right(Ptr);
     else if (!Isnil(Left(Ptr)))
        Ptr = Max(Left(Ptr));
     else
        {Nodeptr P;
         while (!Isnil(P = Parent(Ptr))
                && Ptr == Left(P))
            Ptr = P;
         if (!Isnil(P))
            Ptr = P; })
    void Inc()
    {if (Isnil(Ptr))
        ;
     else if (!Isnil(Right(Ptr)))
        Ptr = Min(Right(Ptr));
     else
        {Nodeptr P;
         while (!Isnil(P = Parent(Ptr))
                && Ptr == Right(P))
            Ptr = P;
         Ptr = P; })
    Nodeptr Mynode() const
        {return (Ptr); }
protected:
    Nodeptr Ptr;
};

typedef std::reverse_iterator<iterator>
    reverse_iterator;
typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef pair<iterator, bool> Pairib;
```

```
typedef pair<iterator, iterator> Pairii;
typedef pair<const_iterator, const_iterator> Paircc;

explicit Tree(const key_compare& Parg,
    const allocator_type& Al)
: Mybase(Parg, Al)
{Init(); }

Tree(const value_type *F, const value_type *L,
    const key_compare& Parg, const allocator_type& Al)
: Mybase(Parg, Al)
{Init();
 insert(F, L); }

Tree(const Myt& X)
: Mybase(X.key_comp(), X.get_allocator())
{Init();
 Copy(X); }

~Tree()
{erase(begin(), end());
 Freenode(Head);
 Head = 0, Size = 0; }

Myt& operator=(const Myt& X)
{if (this != &X)
 {erase(begin(), end());
 comp = X.comp;
 Copy(X); }
 return (*this); }

iterator begin()
{return (iterator(Lmost())); }

const_iterator begin() const
{return (const_iterator(Lmost())); }

iterator end()
{return (iterator(Head)); }

const_iterator end() const
{return (const_iterator(Head)); }

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

const_reverse_iterator rbegin() const
{return (const_reverse_iterator(end())); }

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

const_reverse_iterator rend() const
{return (const_reverse_iterator(begin())); }

size_type size() const
{return (Size); }

size_type max_size() const
```

```

    {return (Alval.max_size()); }

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

allocator_type get_allocator() const
    {return (Alval); }

key_compare key_comp() const
    {return (comp); }

value_compare value_comp() const
    {return (value_compare(key_comp())); }

Pairib insert(const value_type& V)
    {Nodeptr X = Root();
     Nodeptr Y = Head;
     bool Addleft = true;
     while (!Isnil(X))
        {Y = X;
         Addleft = comp(Kfn()(V), Key(X));
         X = Addleft ? Left(X) : Right(X); }
     if (Multi)
        return (Pairib(Insert(Addleft, Y, V), true));
     else
        {iterator P = iterator(Y);
         if (!Addleft)
             ;
         else if (P == begin())
             return (Pairib(Insert(true, Y, V), true));
         else
             --P;
         if (comp(Key(P.Mynode()), Kfn()(V)))
             return (Pairib(Insert(Addleft, Y, V), true));
         else
             return (Pairib(P, false)); }
     iterator insert(iterator P, const value_type& V)
     {if (size() == 0)
      return (Insert(true, Head, V));
     else if (P == begin())
      {if (comp(Kfn()(V), Key(P.Mynode())))
       return (Insert(true, P.Mynode(), V)); }
     else if (P == end())
      {if (comp(Key(Rmost()), Kfn()(V)))
       return (Insert(false, Rmost(), V)); }
     else
      {iterator Pb = P;
       if (comp(Key((--Pb).Mynode()), Kfn()(V))
           && comp(Kfn()(V), Key(P.Mynode())))
           if (Isnil(Right(Pb.Mynode())))

```

```
        return(Insert(false, Pb.Mynode(), V));
    else
        return(Insert(true, P.Mynode(), V)); }
    return (insert(V).first); }

template<class It>
void insert(It F, It L)
{for (; F != L; ++F)
    insert(*F); }

iterator erase(iterator P)
{if (Isnil(P.Mynode()))
    throw out_of_range("map/set<T> iterator");
Nodeptr X, Xpar;
Nodeptr Y = (P++) .Mynode();
Nodeptr Z = Y;
if (Isnil(Left(Y)))
    X = Right(Y);
else if (Isnil(Right(Y)))
    X = Left(Y);
else
    Y = Min(Right(Y)), X = Right(Y);
if (Y == Z)
    (Xpar = Parent(Z));
if (!Isnil(X))
    Parent(X) = Xpar;
if (Root() == Z)
    Root() = X;
else if (Left(Xpar) == Z)
    Left(Xpar) = X;
else
    Right(Xpar) = X;
if (Lmost() != Z)
    ;
else if (Isnil(Right(Z)))
    Lmost() = Xpar;
else
    Lmost() = Min(X);
if (Rmost() != Z)
    ;
else if (Isnil(Left(Z)))
    Rmost() = Xpar;
else
    vRmost() = Max(X); }

else
    {Parent(Left(Z)) = Y;
    Left(Y) = Left(Z); }
```

```
        if (Y == Right(Z))
            Xpar = Y;
        else
            (Xpar = Parent(Y));
            if (!Isnil(X))
                Parent(X) = Xpar;
            Left(Xpar) = X;
            Right(Y) = Right(Z);
            Parent(Right(Z)) = Y; }
        if (Root() == Z)
            Root() = Y;
        else if (Left(Parent(Z)) == Z)
            Left(Parent(Z)) = Y;
        else
            Right(Parent(Z)) = Y;
        Parent(Y) = Parent(Z);
        std::swap(Color(Y), Color(Z)); }

    if (Color(Z) == Black)
        {for ( ; X != Root() && Color(X) == Black;
            Xpar = Parent(X))
        if (X == Left(Xpar))
            {Nodeptr W = Right(Xpar);
            if (Color(W) == Red)
                {Color(W) = Black;
                Color(Xpar) = Red;
                Lrotate(Xpar);
                W = Right(Xpar); }
            if (Isnil(W))
                X = Xpar; // shouldn't happen
            else if (Color(Left(W)) == Black
                    && Color(Right(W)) == Black)
                {Color(W) = Red;
                X = Xpar; }
            else
                {if (Color(Right(W)) == Black)
                    {Color(Left(W)) = Black;
                    Color(W) = Red;
                    Rrotate(W);
                    W = Right(Xpar); }
                    Color(W) = Color(Xpar);
                    Color(Xpar) = Black;
                    Color(Right(W)) = Black;
                    Lrotate(Xpar);
                    break; })
        else
```

```

        {Nodeptr W = Left(Xpar);
        if (Color(W) == Red)
            {Color(W) = Black;
             Color(Xpar) = Red;
             Rrotate(Xpar);
             W = Left(Xpar); }
        if (Isnul(W))
            X = Xpar; // shouldn't happen
        else if (Color(Right(W)) == Black
                  && Color(Left(W)) == Black)
            {Color(W) = Red;
             X = Xpar; }
        else
            {if (Color(Left(W)) == Black)
               {Color(Right(W)) = Black;
                Color(W) = Red;
                Lrotate(W);
                W = Left(Xpar); }
               Color(W) = Color(Xpar);
               Color(Xpar) = Black;
               Color(Left(W)) = Black;
               Rrotate(Xpar);
               break; }
        Color(X) = Black; }
    Destval(&Value(Z));
    Freenode(Z);
    if (0 < Size)
        --Size;
    return (P);
iterator erase(iterator F, iterator L)
{if (size() == 0 || F != begin() || L != end())
   {while (F != L)
      erase(F++);
   return (F); }
else
{Erase(Root());
 Root() = Head, Size = 0;
 Lmost() = Head, Rmost() = Head;
 return (begin()); }
size_type erase(const key_type& X)
{Pairii P = equal_range(X);
 size_type N = 0;
 Distance(P.first, P.second, N);
 erasa(P.first, P.second);
 return (N); }

```

```
void erase(const key_type *F, const key_type *L)
    {while (F != L)
     erase(*F++); }
void clear()
    {erase(begin(), end()); }
iterator find(const key_type& Kv)
    {iterator P = lower_bound(Kv);
     return (P == end()
            || comp(Kv, Key(P.Mynode())))
            ? end() : P; }
const_iterator find(const key_type& Kv) const
    {const_iterator P = lower_bound(Kv);
     return (P == end()
            || comp(Kv, Key(P.Mynode())))
            ? end() : P; }
size_type count(const key_type& Kv) const
    {Paircc Ans = equal_range(Kv);
     size_type N = 0;
     Distance(Ans.first, Ans.second, N);
     return (N); }
iterator lower_bound(const key_type& Kv)
    {return (iterator(Lbound(Kv))); }
const_iterator lower_bound(const key_type& Kv) const
    {return (const_iterator(Lbound(Kv))); }
iterator upper_bound(const key_type& Kv)
    {return (iterator(Ubound(Kv))); }
const_iterator upper_bound(const key_type& Kv) const
    {return (iterator(Ubound(Kv))); }
Pairii equal_range(const key_type& Kv)
    {return (Pairii(lower_bound(Kv), upper_bound(Kv))); }
Paircc equal_range(const key_type& Kv) const
    {return (Paircc(lower_bound(Kv), upper_bound(Kv))); }
void swap(Myrt& X)
    {if (get_allocator() == X.get_allocator())
     (std::swap(comp, X.comp);
      std::swap(Head, X.Head);
      std::swap(Size, X.Size); )
    else
     {Myrt Ts = *this; *this = X, X = Ts; }}
protected:
void Copy(const Myrt& X)
    {Root() = Copy(X.Root(), Head);
     Size = X.size();
     if (!Isnil(Root()))
      {Lmost() = Min(Root());}
```

```

        Rmost() = Max(Root()); }
else
    Lmost() = Head, Rmost() = Head; }
Nodeptr Copy(Nodeptr X, Nodeptr P)
{Nodeptr R = Head;
if (!Isnil(X))
    {Nodeptr Y = Buynode(P, Color(X));
try {
    Consval(&Value(Y), Value(X));
} catch (...) {
    Freenode(Y);
    Erase(R);
    throw;
}
Left(Y) = Head, Right(Y) = Head;
if (Isnil(R))
    R = Y;
try {
    Left(Y) = Copy(Left(X), Y);
    Right(Y) = Copy(Right(X), Y);
} catch (...) {
    Erase(R);
    throw;
}}
return (R); }
void Erase(Nodeptr X)
{for (Nodeptr Y = X; !Isnil(Y); X = Y)
    {Erase(Right(Y));
Y = Left(Y);
Destval(&Value(X));
Freenode(X); }}
void Init()
{Head = Buynode(0, Black);
Isnil(Head) = true;
Root() = Head;
Lmost() = Head, Rmost() = Head;
Size = 0; }
iterator Insert(bool Addleft, Nodeptr Y,
const value_type& V)
{if (max_size() - 1 <= Size)
    throw length_error("map/set<T> too long");
Nodeptr Z = Buynode(Y, Red);
Left(Z) = Head, Right(Z) = Head;
try {
    Consval(&Value(Z), V);
}

```

```
        } catch (...) {
        Freenode(Z);
        throw;
    }
    ++Size;
    if (Y == Head)
        {Root() = Z;
        Lmost() = Z, Rmost() = Z; }
    else if (Addleft)
        {Left(Y) = Z;
        if (Y == Lmost())
            Lmost() = Z; }
    else
        {Right(Y) = Z;
        if (Y == Rmost())
            Rmost() = Z; }
    for (Nodeptr X = Z; Color(Parent(X)) == Red; )
        if (Parent(X) == Left(Parent(Parent(X))))
            {Y = Right(Parent(Parent(X)));
            if (Color(Y) == Red)
                {Color(Parent(X)) = Black;
                Color(Y) = Black;
                Color(Parent(Parent(X))) = Red;
                X = Parent(Parent(X)); }
            else
                {if (X == Right(Parent(X)))
                    {X = Parent(X);
                    Lrotate(X); }
                Color(Parent(X)) = Black;
                Color(Parent(Parent(X))) = Red;
                Rrotate(Parent(Parent(X))); })
        else
            {Y = Left(Parent(Parent(X)));
            if (Color(Y) == Red)
                {Color(Parent(X)) = Black;
                Color(Y) = Black;
                Color(Parent(Parent(X))) = Red;
                X = Parent(Parent(X)); }
            else
                {if (X == Left(Parent(X)))
                    {X = Parent(X);
                    Rrotate(X); }
                Color(Parent(X)) = Black;
                Color(Parent(Parent(X))) = Red;
                Lrotate(Parent(Parent(X))); })}
```

```

        Color(Root()) = Black;
        return (iterator(Z));
    Nodeptr Lbound(const key_type& Kv) const
    {
        (Nodeptr X = Root());
        Nodeptr Y = Head;
        while (!Isnil(X))
            if (comp(Key(X), Kv))
                X = Right(X);
        else
            Y = X, X = Left(X);
        return (Y);
    }
    Nodeptr& Lmost()
    {
        (return (Left(Head)));
    }
    Nodeptr& Lmost() const
    {
        (return (Left(Head)));
    }
    void Lrotate(Nodeptr X)
    {
        Nodeptr Y = Right(X);
        Right(X) = Left(Y);
        if (!Isnil(Left(Y)))
            Parent(Left(Y)) = X;
        Parent(Y) = Parent(X);
        if (X == Root())
            Root() = Y;
        else if (X == Left(Parent(X)))
            Left(Parent(X)) = Y;
        else
            Right(Parent(X)) = Y;
        Left(Y) = X;
        Parent(X) = Y;
    }
    static Nodeptr Max(Nodeptr P)
    {
        while (!Isnil(Right(P)))
            P = Right(P);
        return (P);
    }
    static Nodeptr Min(Nodeptr P)
    {
        while (!Isnil(Left(P)))
            P = Left(P);
        return (P);
    }
    Nodeptr& Rmost()
    {
        (return (Right(Head)));
    }
    Nodeptr& Rmost() const
    {
        (return (Right(Head)));
    }
    Nodeptr& Root()
    {
        (return (Parent(Head)));
    }
    Nodeptr& Root() const
    {
        (return (Parent(Head)));
    }

```

```
void Rrotate(Nodeptr X)
    {Nodeptr Y = Left(X);
     Left(X) = Right(Y);
     if (!Isnil(Right(Y)))
         Parent(Right(Y)) = X;
     Parent(Y) = Parent(X);
     if (X == Root())
         Root() = Y;
     else if (X == Right(Parent(X)))
         Right(Parent(X)) = Y;
     else
         Left(Parent(X)) = Y;
     Right(Y) = X;
     Parent(X) = Y; }

Nodeptr Ubound(const key_type& Kv) const
{Nodeptr X = Root();
 Nodeptr Y = Head;
 while (!Isnil(X))
     if (comp(Kv, Key(X)))
         Y = X, X = Left(X);
     else
         X = Right(X);
 return (Y); }

Nodeptr Buynode(Nodeptr Parg, char Carg)
{Nodeptr S = Alnod.allocate(1, (void *)0);
 Alptr.construct(&Left(S), 0);
 Alptr.construct(&Right(S), 0);
 Alptr.construct(&Parent(S), Parg);
 Color(S) = Carg;
 Isnil(S) = false;
 return (S); }

void Consval(Tptr P, const value_type& V)
{Alval.construct(P, V); }

void Destval(Tptr P)
{Alval.destroy(P); }

void Freenode(Nodeptr S)
{Alptr.destroy(&Parent(S));
 Alptr.destroy(&Right(S));
 Alptr.destroy(&Left(S));
 Alnod.deallocate(S, 1); }

Nodeptr Head;
size_type Size;
};

// Tree TEMPLATE OPERATORS
```

```

template<class Tr> inline
    void swap(Tree<Tr>& X, Tree<Tr>& Y)
        {X.swap(Y); }
template<class Tr> inline
    bool operator==(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (X.size() == Y.size())
            && equal(X.begin(), X.end(), Y.begin())); }
template<class Tr> inline
    bool operator!=(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (!(X == Y)); }
template<class Tr> inline
    bool operator<(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (lexicographical_compare(X.begin(), X.end(),
            Y.begin(), Y.end(), X.value_comp())); }
template<class Tr> inline
    bool operator>(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (Y < X); }
template<class Tr> inline
    bool operator<=(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (!(Y < X)); }
template<class Tr> inline
    bool operator>=(const Tree<Tr>& X, const Tree<Tr>& Y)
        {return (!(X < Y)); }
} /* namespace std */
#endif /* XTREE_ */

```

这两个容器（Tree和list）都把元素存储在单个的节点中，并且节点之间彼此链接。它们惟一的不同就是：链表节点只有两个链接（一个指向链表中的上一个节点，一个指向下一个节点）；而树的节点则有三个链接（指向左右子树和父节点）。这两个容器也都使用了虚构的“头”节点来定位整个被控序列。但在这两个容器中，申请和释放节点、存取节点中的元素等行为看起来都非常相似。

和list一样的是，Tree对象中也只存储一个指针和一个用以表示被控序列中元素个数的计数器。除了下面要讨论的分配器对象外，Tree中只存储两个对象：

Head

- Head就是指向虚构的头节点的指针，它的左指针指向被控序列中的第一个元素，它的右指针指向被控序列的最后一个元素，而它的父指针则指向树的根节点。对于一个空树来说，所有这些指针都指回头节点本身。

Size

- Size计算树中元素的个数。

那个虚构的头节点提供一个附加的用途。以往为了表达一个子树被去除，我们不得不采用某种办法。最容易想到的就是用一个空指针，但这样做却有它的缺陷。模板类Tree中的大量代码都把时间花在对树的爬升上。

爬下上面。夸张一点讲，这样的代码可能会变得非常棘手，尤其是当它们不得不经常检测一个指针是否为空以判定子树已经被删除时更是如此。然而即使这样，至少还有一种常用的实现使用的空节点指针来指定被删除的子树。

空节点

一个常用的替代法就是用“空(nil)”节点来有效地把整个树的末梢节点给包围起来。粗略地看，空节点和其他节点没有什么两样。只有当代码在空节点上面有着疑惑并且做出一些不凡的事情时，我们才需要给予它特殊的关照。某些实现使用普通静态节点来存储空节点。在多线程环境中，这会导致问题的产生，因为有些代码会临时改变存储在空节点中的颜色。树间所共享的静态空节点可能导致很多令人惊奇的线程间交互。

本实现使用了空节点，但用了一种特殊的手法。它把头节点作为每个列表中惟一的空节点重新利用了。在同一个对象中，原来表示节点颜色的那一位被设置为标注该节点是否为空节点。（这种情况只可能出现在头节点中。）这样我们得到的代码就和那些使用明显的空节点的代码同样简单和高效了，并且可读性也不会降低。

一个Tree<Tr>的特化版本实际上派生自一连串的基类。最终的基类就是traits类Tr。剩下的三个基类则用来存储分配器对象，这一点和list很像：

Tree_nod

- Tree_nod<T, A>定义泛型指针类型 Genptr 和节点类型 Node。它也存储分配器对象 Alnod。

Tree_ptr

- Tree_ptr<T, A>定义节点指针类型 Nodeptr。它也存储分配器对象 Alptr。

Tree_val

- Tree_val<T, A>存储分配器对象 Alnod。

一个足够智能的编译器会知道不为这些基类对象在Tree<Tr>对象中分配存储空间。

Tree 定义了那些常见的类型定义。某些这样的类型其实有可能是从 traits 基类中得来的。其他的则也是按照常见的方式，从分配器类引入，而分配器又是由 traits 类指定的。Tree 也定义了一些静态的存取器函数(如 Key 和 left)用以操作存储于节点中的对象。

iterator const_iterator

成员类iterator和const_iterator只支持双向迭代器(这一点和list一样)，而不是像vector一样支持随机存取迭代器。每个迭代器中只存储一个指向节点的指针。在这两个成员类中的成员函数Inc和Dec实现了按照适当的顺序遍历整个序列的操作。例如，在此给出了递增一个迭代器的代码：

```
void Inc()
{
    if (Isnil(Ptr))
        ;
    else if (!Isnil(Right(Ptr)))
        Right(Ptr) = Inc();
}
```

```

        Ptr = Min(Right(Ptr));
    else
        {Nodeptr P;
        while (!Isnil(P = Parent(Ptr))
            && Ptr == Right(P))
            Ptr = P;
        Ptr = P; }
    
```

一个指向空（头）节点的迭代器实际上就是被控序列的end-of-sequence值。对这样的迭代器值进行递增是无效的。与草率的代码相比，这段代码什么也没有做。一个指向有着右子树的节点的迭代器会挪到指向该子树中“最小的”（或最左边的）那个元素。这也就是表达式Min(Right(Ptr))所干的事。否则，这段代码将会向树的顶端爬直到发现一个在树中处于它右边的父节点为止。（即该子节点不是该父节点的右子树的根节点。）如果一直到树的顶端还没有发现这样的节点，那么最初的值将会被改为end-of-sequence值。（即该迭代器指向被控序列中“最大的”，或者说最右边的元素。）

我们可以在这两个迭代器类的Dec函数中发现相似的逻辑。如果能够理解Inc和Dec，我们就可以理解树结构是如何加强排序需求的。

Buynode
Freenode

有几个保护型成员函数实现了一些基本的操作。例如，调用Buynode(parent, color)会分配一个节点，并把它的父指针和表示颜色的成员对象初始化为适当的值。调用Freenode(p)会释放一个节点。这两个函数都假设存在着另外一个代理，由它来负责在必要时对节点中存储的元素以及左右指针进行构造和销毁。

注意，没有一个构造函数或模板参数会对这些行为有着默认情况。模板类Tree主要是在其他STL容器的内部使用。因而它也就不需要所有那些用户级模板类缩写了。

insert

在两个相邻元素间插入一个元素是很困难的，成员函数insert和Insert封装了这个操作。基本上来说，insert会向树的下部爬，直到找到可以把新节点作为其叶子节点的合适节点Y为止。如果Addleft的返回值为true，该叶子节点就是Y的左孩子，否则它就是Y的右孩子。在此，Pairib就是pair<iterator, bool>的类型定义。

Insert

Insert执行实际的插入。它的代码会乐观地把新的节点加入到一个红色的链接中。如果它上面的链接是黑色的话，插入就算是完成了。否则，它就会逐步向树的顶端爬，直到发现一个它可以改正引入的额外的黑色链接为止。相同代码的两个版本都在镜像中出现。理解了其中的一个就可以知道另外一个是怎么样存在和工作的。（不过它们不是那种很容易被理解的代码。）

所有的重排都是通过调用保护型成员函数Lrotate(X)和Rrotate(X)来实

现的。前者实际上是获得X的右子树，把它放在X所处的位置，然后让X作为左子树挂在它的下面。（注意：在进行这样的转化后，该树的次序仍然没有改变。）后者是它的镜像，它把X的左子树提升到X的位置。

erase

删除一个元素的困难工作也封装在第一个成员函数erase中。如果想了解它是如何工作的，我们建议你花些时间画一张草图。你将不得不画出大量的子树以使自己确信所有可能的情况都被完全处理了。如果它很不幸地被要求删除一个叶子节点，它就会用序列中下一个元素（它也必须为一个叶子节点）来替代被删除的元素，然后再删除该元素。如果被删除的叶子节点有一个红色的链接，事情就结束了。否则，它就必须为删除一个黑色的链接做修正。在此重申一遍，相同代码的两个版本都出现在镜像中。其中没有一个是容易理解的。

最本质的策略就是寻找一个机会，把红色的链接转化为黑色的链接。直到可以这么做之前，代码都是在树上爬。在每一次爬的过程中，它不得不对它所经过的其他子树进行处理。例如，如何在节点上引入一个黑色的链接来安排该子树，而不把其他子树搞乱。我们不得不实现的操作实际上并不会比玩魔方难多少。不过要想得到最终代码还真不是一件容易的事情。

模板类Tree中还有其他许多神秘之处。它也是一个很难设计、撰写和理解的类。但大部分其余的复杂性都可以从在此给出的其他STL容器类中得到。我们在此不想重复早期的说教，虽然它们可能会帮助我们更好地解释这个复杂的模板类。然而，如果希望学习到更好的算法，建议仔细研究模板类Tree。

set

程序清单13-2列出了文件set。它定义了模板类set和multiset。一旦给定了模板类Tree，实现这两个容器的其余工作就变得轻松起来。

```
程序清单 13-2: // set standard header
set           #ifndef SET_
               #define SET_
               #include <xtree>
               namespace std {
                   // TEMPLATE CLASS Tset_traits
                   template<class K, class Pr, class Ax, bool Mfl>
                   class Tset_traits {
               public:
                   typedef K key_type;
                   typedef K value_type;
                   typedef Pr key_compare;
                   typedef typename Ax::template rebind<value_type>::other
                           allocator_type;
                   enum {Multi = Mfl};
```

```
Tset_traits()
    : comp()
    {}
Tset_traits(Pr Parg)
    : comp(Parg)
    {}
typedef key_compare value_compare;
struct Kfn {
    const K& operator()(const value_type& X) const
        (return (X); )
    };
Pr comp;
};

// TEMPLATE CLASS set
template<class K,
          class Pr = less<K>,
          class A = allocator<K> >
class set
    : public Tree<Tset_traits<K,Pr, A, false> > {
public:
    typedef set<K, Pr, A> Myt;
    typedef Tree<Tset_traits<K, Pr, A, false> >
        Mybase;
    typedef K key_type;
    typedef Pr key_compare;
    typedef typename Mybase::value_compare value_compare;
    typedef typename Mybase::allocator_type allocator_type;
    typedef typename Mybase::size_type size_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::const_pointer const_pointer;
    typedef typename Mybase::reference reference;
    typedef typename Mybase::const_reference
        const_reference;
    typedef typename Mybase::iterator iterator;
    typedef typename Mybase::const_iterator const_iterator;
    typedef typename Mybase::reverse_iterator
        reverse_iterator;
    typedef typename Mybase::const_reverse_iterator
        const_reverse_iterator;
    typedef typename Mybase::value_type value_type;
    set()
        : Mybase(key_compare(), allocator_type()) {}
```

```
    explicit set(const key_compare& Pred)
        : Mybase(Pred, allocator_type()) {}
    set(const key_compare& Pred, const allocator_type& A1)
        : Mybase(Pred, A1) {}
    template<class It>
        set(It F, It L)
            : Mybase(key_compare(), allocator_type())
            {for (; F != L; ++F)
                insert(*F); }
    template<class It>
        set(It F, It L, const key_compare& Pred)
            : Mybase(Pred, allocator_type())
            {for (; F != L; ++F)
                insert(*F); }
    template<class It>
        set(It F, It L, const key_compare& Pred,
            const allocator_type& A1)
            : Mybase(Pred, A1)
            {for (; F != L; ++F)
                insert(*F); }
    };

    // TEMPLATE CLASS multiset
template<class K,
         class Pr = less<K>,
         class A = allocator<K> >
class multiset
    : public Trcc<Tset_traits<K, Pr, A, true> > {
public:
    typedef multiset<K, Pr, A> Myt;
    typedef Tree<Tset_traits<K, Pr, A, true> >
        Mybase;
    typedef K key_type;
    typedef Pr key_compare;
    typedef typename Mybase::value_compare value_compare;
    typedef typename Mybase::allocator_type allocator_type;
    typedef typename Mybase::size_type size_type;
    typedef typename Mybase::difference_type difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::const_pointer const_pointer;
    typedef typename Mybase::reference reference;
    typedef typename Mybase::const_reference const_reference;
    typedef typename Mybase::iterator iterator;
    typedef typename Mybase::const_iterator const_iterator;
    typedef typename Mybase::reverse_iterator
```

```

        reverse_iterator;
typedef typename Mybase::const_reverse_iterator
    const_reverse_iterator;
typedef typename Mybase::value_type value_type;
multiset()
    : Mybase(key_compare(), allocator_type()) {}
explicit multiset(const key_compare& Pred)
    : Mybase(Pred, allocator_type()) {}
multiset(const key_compare& Pred, const allocator_type& Al)
    : Mybase(Pred, Al) {}
template<class It>
    multiset(It F, It L)
        : Mybase(key_compare(), allocator_type())
        {for (; F != L; ++F)
            insert(*F); }
template<class It>
    multiset(It F, It L, const key_compare& Pred)
        : Mybase(Pred, allocator_type())
        {for (; F != L; ++F)
            insert(*F); }
template<class It>
    multiset(It F, It L, const key_compare& Pred,
        const allocator_type& Al)
        : Mybase(Pred, Al)
        {for (; F != L; ++F)
            insert(*F); }
iterator insert(const value_type& X)
    {return (Mybase::insert(X).first); }
iterator insert(iterator P, const value_type& X)
    {return (Mybase::insert(P, X)); }
template<class It>
    void insert(It F, It L)
    {for (; F != L; ++F)
        insert(*F); }
};

} /* namespace std */
#endif /* SET_ */

```

Tset_traits

模板类 `Tset_traits` 是 `Tree traits` 类的一个版本，它适合用来定义 `set`。如同我们可以看到的一样，它并没有做很多事情，但它却可以使得我们在特化一个 `Tree` 时不再需要写原来那么长的一串参数列表。注意，对于 `set` 来说，`key_type` 是 `value_type` 的同义词。也就是说，通过 `Kfn` 从值中提取出键实际上返回整个值。

multiset 除了回送一些类型定义和提供适当的构造函数之外，模板类 `set` 并没有其他的事情要做。模板类 `multiset` 和 `set` 也差不多，不过它稍微多了些东西。它包装了对 `Tree::insert` 的调用以简化成员函数 `insert(const value_type&)` 的返回值，其他的包装则不是严格必要的，但它们还是可以帮助某些编译器避免陷入混乱状态。

测试<set>

程序清单 13-3 列出了文件 `tset.c`。它提供了两个相似的模板类 `set` 和 `multiset` 的测试。它实际上也是两个很相似的测试程序中的一个。另外一个是第 14 章的文件 `tmap.c`。为了简化对这两个文件中四个测试的比较，我们只是简单地注释出对于给定容器的特定测试。

```
程序清单 13-3: // test <set>
tset.c      #include <assert.h>
            #include <iostream>
            #include <functional>
            #include <set>
            using namespace std;

            // TEST set
void test_set()
{
    {typedef allocator<char> Myal;
     typedef less<char> Mypred;
     typedef set<char, Mypred, Myal> Mycont;
     char ch, carr[] = "abc", carr2[] = "def";

     Mycont::key_type *p_key = (char *)0;
     Mycont::key_compare *p_kcomp = (Mypred *)0;
     Mycont::value_type *p_val = (char *)0;
     Mycont::value_compare *p_vcomp = (Mypred *)0;
     Mycont::allocator_type *p_alloc = (Myal *)0;
     Mycont::pointer p_ptr = (char *)0;
     Mycont::const_pointer p_cptr = (const char *)0;
     Mycont::reference p_ref = ch;
     Mycont::const_reference p_cret = (const char&)ch;
     Mycont::size_type *p_size = (size_t *)0;
     Mycont::difference_type *p_diff = (ptrdiff_t *)0;

     Mycont v0;
     Myal al = v0.get_allocator();
     Mypred pred;
     Mycont v0a(pred), v0b(pred, al);
```

```
assert(v0.empty() && v0.size() == 0);
assert(v0a.size()==0 && v0a.get_allocator()==al);
assert(v0b.size()==0 && v0b.get_allocator()==al);
Mycont v1(carr, carr + 3);
assert(v1.size() == 3 && *v1.begin() == 'a');
Mycont v2(carr, carr + 3, pred);
assert(v2.size() == 3 && *v2.begin() == 'a');
Mycont v3(carr, carr + 3, pred, al);
assert(v3.size() == 3 && *v3.begin() == 'a');
const Mycont v4(carr, carr + 3);
v0 = v4;
assert(v0.size() == 3 && *v0.begin() == 'a');

Mycont::iterator p_it(v1.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v1.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert(*p_it=='a' && *(p_it = v1.end()) == 'c');
assert(*p_cit=='a' && *(p_cit = v4.end())=='c');
assert(*p_rit=='c' && *(p_rit=v1.rend())=='a');
assert(*p_crit=='c'&&*(p_crit=v4.rend())=='a');

v0.clear(); // DIFFERS FROM multiset
pair<Mycont::iterator, bool> pib = v0.insert('d');
assert(*pib.first == 'd' && pib.second);
assert(*--v0.end() == 'd');
pib = v0.insert('d');
assert(*pib.first == 'd' && !pib.second);
assert(*v0.insert(v0.begin(), 'e') == 'e');
v0.insert(carr, carr + 3);
assert(v0.size() == 5 && *v0.begin() == 'a');
v0.insert(carr2, carr2 + 3);
assert(v0.size() == 6 && *--v0.end() == 'f');
assert(*v0.erase(v0.begin())=='b'&& v0.size()==5);
assert(*v0.erase(v0.begin(), ++v0.begin()) == 'c'
      && v0.size() == 4);
assert(v0.erase('x') == 0 && v0.erase('e') == 1);

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
```

```
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

assert(v0.key_comp()('a', 'c')
    && !v0.key_comp()('a', 'a'));
assert(v0.value_comp()('a', 'c')
    && !v0.value_comp()('a', 'a'));
assert(*v4.find('b') == 'b');
assert(v4.count('x') == 0 && v4.count('b') == 1);
assert(*v4.lower_bound('a') == 'a');
assert(*v4.upper_bound('a') == 'b');
pair<Mycont::const_iterator, Mycont::const_iterator> pcc =
    v4.equal_range('a');
assert(*pcc.first == 'a' && *pcc.second == 'b'); }

// TEST multiset
void test_multiset()
{typedef allocator<char> Myal;
typedef less<char> Mypred;
typedef multiset<char, Mypred, Myal> Mycont;
char ch, carr[] = "abc", carr2[] = "def";

Mycont::key_type *p_key = (char *)0;
Mycont::key_compare *p_kcomp = (Mypred *)0;
Mycont::value_type *p_val = (char *)0;
Mycont::value_compare *p_vcomp = (Mypred *)0;
Mycont::allocator_type *p_alloc = (Myal *)0;
Mycont::pointer p_ptr = (char *)0;
Mycont::const_pointer p_cptr = (const char *)0;
Mycont::reference p_ref = ch;
Mycont::const_reference p_cref = (const char&)ch;
Mycont::size_type *p_size = (size_t *)0;
Mycont::difference_type *p_diff = (ptrdiff_t *)0;

Mycont v0;
Myal al = v0.get_allocator();
Mypred pred;
Mycont v0a(pred), v0b(pred, al);
assert(v0.empty() && v0.size() == 0);
assert(v0a.size()==0 && v0a.get_allocator()==al);
assert(v0b.size()==0 && v0b.get_allocator()==al);
Mycont v1(carr, carr + 3);
assert(v1.size() == 3 && *v1.begin() == 'a');
Mycont v2(carr, carr + 3, pred);
assert(v2.size() == 3 && *v2.begin() == 'a');
```

```
Mycont v3(carr, carr + 3, pred, al);
assert(v3.size() == 3 && *v3.begin() == 'a');
const Mycont v4(carr, carr + 3);
v0 = v4;
assert(v0.size() == 3 && *v0.begin() == 'a');

Mycont::iterator p_it(v1.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v1.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert(*p_it=='a' && *(p_it = v1.end()) == 'c');
assert(*p_cit=='a' && *(p_cit = v4.end())=='c');
assert(*p_rit=='c' && *(p_rit=v1.rend())=='a');
assert(*p_crit=='c'& *(p_crit=v4.rend())=='a');

v0.clear(); // DIFFERS FROM set
assert(*v0.insert('d') == 'd');
assert(*--v0.end() == 'd');
assert(*v0.insert('d') == 'd');
assert(v0.size() == 2);
assert(*v0.insert(v0.begin(), 'e') == 'e');
v0.insert(carr, carr + 3);
assert(v0.size() == 6 && *v0.begin() == 'a');
v0.insert(carr2, carr2 + 3);
assert(v0.size() == 9 && *v0.end() == 'f');
assert(*v0.erase(v0.begin())=='b'& v0.size()==8);
assert(*v0.erase(v0.begin(), ++v0.begin()) == 'c'
&& v0.size() == 7);
assert(v0.erase('x') == 0 && v0.erase('e') == 2);

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

assert(v0.key_comp()('a', 'c')
&& !v0.key_comp()('a', 'a'));
assert(v0.value_comp()('a', 'c')
&& !v0.value_comp()('a', 'a'));
assert(*v4.find('b') == 'b');
```

```
assert(v4.count('x') == 0 && v4.count('b') == 1);
assert(*v4.lower_bound('a') == 'a');
assert(*v4.upper_bound('a') == 'b');
pair<Mycont::const_iterator,
Mycont::const_iterator> pcc =
    v4.equal_range('a');
assert(*pcc.first == 'a' && *pcc.second == 'b'); }

// TEST <set>
int main()
{test_set();
test_multiset();
cout << "SUCCESS testing <set>" << endl;
return (0); }
```

该测试程序只是对模板类set的一个特化版本进行了简单的测试，按照其意图测试了它所给出的每个成员函数和成员类型。如果测试一切顺利的话，测试程序将打印出：

```
SUCCESS testing <set>
```

然后正常退出。

习题

习题13-1

模板类Tree的这个实现从不在节点间复制元素。元素只能被创建和销毁，但从不被赋值。在什么样的情况下，我们会需要这样的行为呢？

习题13-2

重写从模板类Tree，消除掉对于头节点的使用。在什么样的情况下，这样的重写会是一个比较好的设计呢？

习题13-3

改变Tree迭代器的定义，使得我们更容易检测两个迭代器是否指向不同的被控序列中的元素。

习题13-4

有些人相信，C++标准允许存储在set或multiset元素中的值改变。如果允许出现这种情况，它的优缺点各在哪呢？

习题13-5

[较难] 改变模板类Tree，去除掉树平衡逻辑，然后在一些应用程序中测试平衡树和非平衡树的相对性能。在什么情况下，非平衡树要胜过平衡树？

习题13-6

[特难] 改变模板类set和multiset，使得它们在被存储的值有改动时，根据需要自动重排元素。

第 14 章 <map>

背景知识

<map>

map

multimap

头文件<map>中定义了模板类 `map` 和模板类 `multimap`。它们都是容器，通常它们所控制的长度为 N 的序列都是以有 N 个节点的有序二叉树的方式存储的。每个节点中存储一个类型为 `pair<const Key, T>` 的单个元素。该序列是以类型为该类的一个模板参数的函数对象进行排序的。只有 `const Key` 这个部分才参与次序比较。`multimap` 允许两个（相邻的）元素有次序相等的键，而 `map` 则不允许出现这种情况。更通俗点讲，`map` 和 `multimap` 连同每个键一起存储了一个“映射值”。正如你所猜测的那样：在 `map` 中，所有的键都必须是唯一的；而 `multimap` 则允许有重复。

<set>

set

multiset

在前一章中，我们介绍了和它类似的头文件<set>。它定义了模板类 `set` 和模板类 `multiset`。它们与 `map` 和 `multimap` 有一个基本的区别——它们中存储的元素类型为 `const Key`，但只有 `Const key` 会参与次序比较。对于 `set` 和 `multiset` 来说，整个元素的值都会参与次序比较。通俗点讲，`set` 或者 `multiset` 中对于每个元素只存储了一个不变的键。正如你所猜测的那样，`set` 中的所有键都是独一无二的，而在 `multiset` 中则有可能重复。

关联容器

如同我们在前面那章所详细描述的一样，模板类 `set`、`multiset`、`map` 和 `multimap` 被称为“关联容器”。它们都是把一个键值与一个元素联系起来，并使用该键来加速诸如查找、插入以及删除元素等操作。在第 9 章的表 9-1 中，我们已经列出了对应于在关联容器中所增加的复杂度带来的回报。（见该表中的“`set/map`”一栏。）毫无疑问，在我们需要使用键值来查找元素时，它们要胜过其他所有的 STL 容器。虽说有序向量或双队列也能达到同样的效果，但它们在插入和删除时都无法达到比线性时间更好的效果。它们的基础树的表示形式可以在对数时间内完成这些操作。要想了解更多关联容器和有序二叉树的优点的细节，参见第 13 章。

集合是以它存储的值来进行排序的。那是一种表示许多集合（例如，数学意义上的集合）的简便方法。但有的时候，它受到的限制太多了。有时，我们可能只想针对存储在每个元素中的部分信息对序列排序，其他的信息则不考虑。例如，一个符号表。我们希望它以符号的名字来排序，以实现快速查找；但我们并不希望排序对于符号的值以及任何属性标志有什么意义。

这没问题。我们可以简单地定义我们认为合适的谓词类。它可以比

较存储值中的名字部分并忽略其他内容。虽然这并不是一个大问题，但它还是会变得很讨厌。我们不希望对于每个既存储了键又存储了一个独立的值的容器都写出一个特殊的谓词类来。于是，STL为这两个模板类的实现也采用了二叉树的方式。

```
template<class Key, class T, class Pred = less<Key>,
         class A = allocator<pair<const Key, T>>>
class map;
template<class Key, class T, class Pred = less<Key>,
         class A = allocator<pair<const Key, T>>>
class multimap;
```

这两个模板类存储的元素都是一个值对。值对中的一个元素就是排序键，它的类型为`const Key`；另一个元素的类型为`T`。其他参数的描述都和`set`以及`multiset`的一样。实际上，这四个关联容器之间相似之处要多于它们的不同之处。

operator[]

然而，模板类`map`还有一个独一无二的特性。它定义了成员操作符`T& operator[const Key& key]`，它把键值`key`和它所映射的值联系起来了。更确切地说，它会尽量去尝试查找一个其键与`key`次序相等的元素。如果查找失败的话，它就插入一个键值为`key`、映射值为默认值的元素。在这两种情况下，该成员操作符都返回一个引用，用于指向仅有的那个匹配`key`参数的映射值。

这是一个很有用的表达方式，可以用来查找给定键值惟一确定的元素，或者是在不存在这样的元素时添加一个。当然，对于其他的关联容器来说，定义这么一个操作符的意义很小。对于`set`来说，映射是毫无意义的——`insert(const Key&)`已经完成了这件事情。对于`multiset`和`multimap`来说，映射将会导致歧义。所以，只有`map`才定义了这个成员操作符。

功能描述

```
namespace std {
template<class Key, class T, class Pred, class A>
class map;
template<class Key, class T, class Pred, class A>
class multimap;

// TEMPLATE FUNCTIONS
template<class Key, class T, class Pred, class A>
bool operator==(  

    const map<Key, T, Pred, A>& lhs,  

    const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator==(
```

```
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        map<Key, T, Pred, A>& lhs,
        map<Key, T, Pred, A>& rhs);
```

```
template<class Key, class T, class Pred, class A>
void swap(
    multimap<Key, T, Pred, A>& lhs,
    multimap<Key, T, Pred, A>& rhs);
};
```

包含STL的标准头文件<map>可以得到模板类map和multimap，以及几个支持模板的定义。

map

```
template<class Key, class T, class Pred = less<Key>,
         class A = allocator<pair<const Key, T> > >
class map {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    map();
    explicit map(const Pred& comp);
    map(const Pred& comp, const A& al);
    map(const map& x);
    template<class InIt>
        map(InIt first, InIt last);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp, const A& al);
    iterator begin();
    const_iterator begin() const;
```

```

iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
mapped_type operator[](const Key& key);
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(map& x);
key_compare key_comp() const;
value_compare value_comp() const;
iterator find(const Key& key);
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
};

```

该模板类描述的对象控制一个元素类型为pair<const Key, T>的可变长度序列。该序列按谓词Pred排序。每个值对中的第一个元素是排序键，第二个是关联值。该序列的表示方式允许我们以与序列中元素个数的对数成比例的时间对它里面的任意元素进行查找、插入以及删除等操作。而且，插入元素并不会导致任何迭代器无效，而删除元素也只会让原来指向被删除元素的迭代器无效。

对象调用它存储的类型为Pred的函数对象来对它控制的序列进行排序。我们可以通过成员函数key_comp()来存取这个对象。该函数对象必须在类型为Key的排序键上施加严格弱序。对于序列中任意排在y前面的元

素x, key_comp(y.first, x.first)都为false。(对于默认的函数对象less<Key>来说,排序键从不以降序的方式存在。)与模板类multimap不同的是,模板类map的对象中保证key_comp(x.first, y.first)一定为true。(因为键是惟一的。)

map对象是通过存储于内部的一个类型为A的分配器对象来进行存储空间的分配及释放的。该分配器对象必须拥有和模板类allocator一样的外部接口。注意,当对容器进行赋值时,存储的分配器对象并没有被复制。

口 map::allocator_type

```
typedef A allocator_type;
```

该类型是模板参数A的同义词。

口 map::begin

```
const_iterator begin() const;  
iterator begin();
```

该成员函数返回一个双向迭代器,它指向被控序列的第一个元素(如果被控序列为空,则指向紧接着该序列末端的下一个位置)。

口 map::clear

```
void clear();
```

该成员函数调用erase(begin(), end())。

口 map::const_iterator

```
typedef T1 const_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量双向迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。

口 map::const_pointer

```
typedef typename A::const_pointer const_pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。

口 map::const_reference

```
typedef typename A::const_reference const_reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。

口 map::const_reverse_iterator

```
typedef reverse_iterator<const_iterator>  
const_reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的常量反转型双向迭代器来使用。

- map::count


```
size_type count(const Key& key) const;
```

该成员函数返回区间[lower_bound(key), upper_bound(key))中元素x的个数。
- map::difference_type


```
typedef T3 difference_type;
```

该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。
- map::empty


```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。
- map::end


```
const_iterator end() const;
iterator end();
```

该成员函数返回一个双向迭代器，它指向紧接着被控序列末端的下一个位置。
- map::equal_range


```
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

该成员函数返回一个迭代器对x，其中x.first == lower_bound(key), x.second == upper_bound(key)。
- map::erase


```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除区间[first, last)中的所有元素。这两个函数都返回一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

第三个成员函数删除区间[lower_bound(key), upper_bound(key))中具有给定排序键的所有元素。它返回所删除元素的个数。

这些成员函数不会抛出任何异常。
- map::find


```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

该成员函数返回一个迭代器，它指向被控序列中排序键与key次序相等的元素。如果不存在这样的元素，函数将返回end()。

- 口 map::get_allocator

```
A get_allocator() const;
```

该成员函数返回存储的分配器对象。

- 口 map::insert

```
pair<iterator, bool> insert (const value_type& x);  
iterator insert (iterator it, const value_type& x);  
template<class InIt>  
void insert (InIt first, InIt last);
```

第一个成员函数会检测其键与x有相等次序的元素y是否存在于被控序列中。如果没有这样的元素，它就会创建一个这样的元素y并用x来初始化它。然后函数会确定指向y的迭代器it。如果有插入发生，函数就返回pair(it, true)；否则，函数返回pair(it, false)。

第二个成员函数返回insert(x)，用it作为被控序列中的起始位置开始查找插入点。(如果插入点紧接着it后面的话，插入操作执行的时间将是被分摊付出的常数时间，而不是对数时间。)第三个成员函数通过对区间[first, last)中的每个it调用insert(*it)，将整个区间中的元素都插入到被控序列中。

如果在插入单个元素时有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。如果在插入多个元素时有异常抛出，容器将保持为一个稳定但未指定的状态，并且将异常继续向外抛出。

- 口 map::iterator

```
typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的双向迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

- 口 map::key_comp

```
key_compare key_comp() const;
```

该成员函数返回存储的函数对象，它用来对被控序列中的元素排序。

该函数对象定义了成员函数：

```
bool operator(const Key& x, const Key& y);
```

如果x按照严格的排序规则出现在y之前，它就会返回true。

- 口 map::key_compare

```
typedef Pred key_compare;
```

该类型描述的函数对象可以用来比较两个排序键，以检测被控序列中两个元素之间的相对次序。

- 口 map::key_type

```
typedef Key key_type;
```

该类型描述了被控序列的每个元素中存储的排序键对象。

口 map::lower_bound

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

该成员函数返回一个迭代器，指向被控序列中第一个让key_comp()(x, key)为false的元素x。如果不存在这样的元素，函数将返回end()。

口 map::map

```
map();
explicit map(const Pred& comp);
map(const Pred& comp, const A& al);
map(const map& x);
template<class InIt>
map(InIt first, InIt last);
template<class InIt>
map(InIt first, InIt last,
      const Pred& comp);
template<class InIt>
map(InIt first, InIt last,
      const Pred& comp, const A& al);
```

所有的构造函数都会存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，分配器对象将会为A()。

上述所有构造函数还会存储一个函数对象，我们可以随后通过调用key_comp()来得到这个函数对象。如果有的话，该函数对象就是参数comp。对于复制构造函数来说，它将是x.key_comp()。如果构造函数中没有提供参数comp的话，那么该函数对象就是Pred()。

开始的三个构造函数会指定空的被控序列。第四个构造函数则指定由x控制的序列的拷贝。最后三个构造函数指定[first, last)中的元素序列。

口 map::mapped_type

```
typedef T mapped_type;
```

该类型是模板参数T的同义词。

口 map::max_size

```
size_type max_size() const;
```

该成员函数返回容器对象所能控制的最长序列的长度。

口 map::operator[]

```
T& operator[](const Key& key);
```

该成员函数会检测作为insert(value_type(key, T()))返回值的迭代器it。(如果没有找到这样的元素，它就会插入一个具有指定键的元素。)然后函数返回一个指向(*it).second的引用。

- 口 map::pointer

```
typedef typename A::pointer pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。
- 口 map::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

该成员函数返回一个反转型双向迭代器，它指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。
- 口 map::reference

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。
- 口 map::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

该成员函数返回一个反转型双向迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。
- 口 map::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型双向迭代器来使用。
- 口 map::size

```
size_type size() const;
```

该成员函数返回被控序列的长度。
- 口 map::size_type

```
typedef T2 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。
- 口 map::swap

```
void swap(map& x);
```

该成员函数在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator()，它将可以在常数时间内完成交换。只有在复制被存储的类型为Pred的函数对象时，它才有可能抛出异常。另外，它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则，它将以与这两个被控序列的元素个数成比例的次数调用元素的构造

函数以及为元素赋值。

口 map::upper_bound

```
iterator upper_bound(const Key& key)
const_iterator upper_bound(const Key& key) const;
```

该成员函数返回一个迭代器，指向被控序列中第一个使 `key_comp(key, x.first)` 为 true 的元素 x。如果不存在这样的元素，它将返回 `end()`。

口 map::value_comp

```
value_compare value_comp() const;
```

该成员函数返回一个用以检测被控序列中元素之间次序的函数对象。

口 map::value_compare

```
class value_compare
: public binary_function<value_type, value_type,
    bool> {
public:
    bool operator()(const value_type& x,
                     const value_type& y) const
    {return (comp(x.first, y.first)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
};
```

该类型描述的是一个函数对象，它可以比较被控序列的两个元素中的排序键，并检测它们在被控序列中的相对次序。该函数对象存储一个类型为 `key_compare` 的对象 `comp`。成员函数 `operator()` 使用这个对象来比较两个元素中排序键的部分。

口 map::value_type

```
typedef pair<const Key, T> value_type;
```

该类型描述了被控序列中的一个元素。

口 multimap

```
template<class Key, class T, class Pred = less<Key>,
         class A = allocator<pair<const Key, T>>>
class multimap {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
```

```
typedef pair<const Key, T> value_type;
class value_compare;
typedef A::reference reference;
typedef A::const_reference const_reference;
typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;
multimap();
explicit multimap(const Pred& comp);
multimap(const Pred& comp, const A& al);
multimap(const multimap& x);
template<class InIt>
multimap(InIt first, InIt last);
template<class InIt>
multimap(InIt first, InIt last,
    const Pred& comp);
template<class InIt>
multimap(InIt first, InIt last,
    const Pred& comp, const A& al);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(multimap& x);
key_compare key_comp() const;
```

```

        value_compare value_comp() const;
        iterator find(const Key& key);
        const_iterator find(const Key& key) const;
        size_type count(const Key& key) const;
        iterator lower_bound(const Key& key);
        const_iterator lower_bound(const Key& key) const;
        iterator upper_bound(const Key& key);
        const_iterator upper_bound(const Key& key) const;
        pair<iterator, iterator> equal_range(const Key& key);
        pair<const_iterator, const_iterator>
            equal_range(const Key& key) const;
    };
}

```

该模板类描述的对象控制一个元素类型为pair<const Key, T>的可变长度序列。该序列按谓词Pred排序。每个值对中的第一个元素是排序键，第二个是关联值。该序列的表示方式允许我们以与序列中元素个数的对数成比例的时间对它里面的任意元素进行查找、插入以及删除等操作。而且，插入元素不会导致任何迭代器无效，而删除元素也只会让原来指向被删除元素的迭代器无效。

对象通过调用存储的类型为Pred的函数对象来对它控制的序列进行排序。我们可以通过定义成员函数key_comp()来存取这个对象。该函数对象必须在类型为Key的排序键上施加一个严格弱序。对于序列中任意排在y前面的元素x, key_comp()(y.first, x.first)都为false。(对于默认的函数对象less<Key>来说，排序键从不以降序的方式存在。)与模板类map不同的是，模板类multimap的对象中并不保证key_comp()(x.first, y.first)一定为true。(因为键不需要一定是惟一的。)

map对象是通过存储的一个类型为A的分配器对象来进行存储空间的分配及释放的。该分配器对象必须拥有和模板类allocator一样的外部接口。注意：当对容器进行赋值时，被存储的分配器对象并没有被复制。

¶ multimap::allocator_type
`typedef A allocator_type;`

该类型是模板参数A的同义词。

¶ multimap::begin
`const_iterator begin() const;`
`iterator begin();`

该成员函数返回一个双向迭代器，它指向被控序列的第一个元素(如果被控序列为空，则指向紧接着该序列末端的下一个位置)。

¶ multimap::clear
`void clear();`

该成员函数调用erase(begin(), end())。

口 multimap::const_iterator

typedef T1 const_iterator;

该类型描述的对象可以作为一个指向被控序列的常量双向迭代器来使用。在此处它被描述为由实现定义的类型T1的同义词。

口 multimap::const_pointer

typedef typename A::const_pointer const_pointer;

该类型描述的对象可以作为一个指向被控序列中元素的常量指针来使用。

口 multimap::const_reference

typedef typename A::const_reference const_reference;

该类型描述的对象可以作为一个指向被控序列中元素的常量引用来使用。

口 multimap::const_reverse_iterator

typedef reverse_iterator<const_iterator>
const_reverse_iterator;

该类型描述的对象可以作为一个指向被控序列的常量反转型双向迭代器来使用。

口 multimap::count

size_type count(const Key& key) const;

该成员函数返回区间[lower_bound(key), upper_bound(key))中元素x的个数。

口 multimap::difference_type

typedef T3 difference_type;

该有符号整数类型描述的对象可以表示被控序列中任意两个元素地址之间的差距。在此处它被描述为由实现定义的类型T3的同义词。

口 multimap::empty

bool empty() const;

当被控序列为空时，该成员函数返回true。

口 multimap::end

const_iterator end() const;
iterator end();

该成员函数返回一个双向迭代器，它指向紧接着被控序列末端的下一个位置。

口 multimap::equal_range

pair<iterator, iterator> equal_range(const Key& key);

```
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

该成员函数返回一个迭代器对x，其中x.first == lower_bound(key),
x.second == upper_bound(key)。

口 multimap::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

第一个成员函数从被控序列中删除由it所指定的元素。第二个成员函数删除区间[first, last)中的所有元素。这两个函数都返回一个迭代器，指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向end()。

第三个成员函数删除区间[lower_bound(key), upper_bound(key))中具有给定排序键的所有元素。它返回所删除元素的个数。

这些成员函数不会抛出任何异常。

口 multimap::find

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

该成员函数返回一个迭代器，它指向被控序列中排序键与key次序相等的第一个元素。如果不存在这样的元素，函数将返回end()。

口 multimap::get_allocator

```
A get_allocator() const;
```

该成员函数返回存储的分配器对象。

口 multimap::insert

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
void insert(InIt first, InIt last);
```

第一个成员函数向被控序列中插入一个元素x，然后返回一个指向被插入元素的迭代器。第二个成员函数返回的是insert(x)，it被用做一个起始位置，标注着从被控序列的哪个位置开始查找插入点。（如果插入点紧接着it的话，插入操作执行的时间将是被分摊付出的常数时间，而不是对数时间。）第三个成员函数通过对区间[first, last)中的每个it调用insert(*it)，将整个区间中的元素都插入到被控序列中。

如果在插入单个元素时有异常抛出，那么容器将保持不变并且继续将该异常向外抛出。如果在插入多个元素时有异常抛出，容器将保持为一个稳定但未指定的状态，并且将异常继续向外抛出。

口 multimap::iterator

```
typedef T0 iterator;
```

该类型描述的对象可以作为一个指向被控序列的双向迭代器来使用。在此处它被描述为由实现定义的类型T0的同义词。

口 multimap::key_comp

```
key_compare key_comp() const;
```

该成员函数返回存储的函数对象，用来对被控序列中的元素排序。

该函数对象定义了成员函数：

```
bool operator(const Key& x, const Key& y);
```

如果x按照严格的排序规则出现在y之前，它就会返回true。

口 multimap::key_compare

```
typedef Pred key_compare;
```

该类型描述的函数对象可以用来比较两个排序键，以检测被控序列中两个元素之间的相对次序。

口 multimap::key_type

```
typedef Key key_type;
```

该类型描述了被控序列的每个元素中存储的排序键。

口 multimap::lower_bound

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

该成员函数将返回一个迭代器，指向被控序列中第一个让key_comp()(x.first, key)为false的元素x。如果不存在这样的元素，函数将返回end()。

口 multimap::mapped_type

```
typedef T mapped_type;
```

该类型是模板参数T的同义词。

口 map::max_size

```
size_type max_size() const;
```

该成员函数返回容器对象所能控制的最长序列的长度。

口 multimap::multimap

```
multimap();
explicit multimap(const Pred& comp);
multimap(const Pred& comp, const A& al);
multimap(const multimap& x);
template<class InIt>
    multimap(InIt first, InIt last);
template<class InIt>
    multimap(InIt first, InIt last,
            const Pred& comp);
template<class InIt>
    multimap(InIt first, InIt last,
```

```
const Pred& comp, const A& al);
```

所有的构造函数都会存储一个分配器对象并且初始化被控序列。如果有的话，分配器对象就是参数al。对于复制构造函数来说，它将是x.get_allocator()。否则，分配器对象将会为A()。

上述所有构造函数还都存储一个函数对象，我们可以随后通过调用key_comp()来得到这个函数对象。如果有的话，该函数对象就是参数comp。对于复制构造函数来说，它将是x.key_comp()。如果构造函数中没有提供参数comp的话，那么该函数对象就是Pred()。

开始的三个构造函数会指定空的初始被控序列。第四个构造函数指定由x控制的序列的拷贝。最后三个构造函数指定[first, last)中的元素序列。

口 multimap::pointer

```
typedef typename A::pointer pointer;
```

该类型描述的对象可以作为一个指向被控序列中元素的指针来使用。

口 multimap::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

该成员函数返回一个反转型双向迭代器，它指向紧接着被控序列末端的下一个位置。因此，它也就指向该序列的逆序序列的开始处。

口 multimap::reference

```
typedef typename A::reference reference;
```

该类型描述的对象可以作为一个指向被控序列中元素的引用来使用。

口 multimap::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

该成员函数返回一个反转型双向迭代器，它指向被控序列中的第一个元素（如果序列是一个空序列的话，则指向紧接着该序列末端的下一个位置）。因此，它也就指向该序列的逆序序列的末端。

口 multimap::reverse_iterator

```
typedef reverse_iterator<iterator> reverse_iterator;
```

该类型描述的对象可以作为一个指向被控序列的反转型双向迭代器来使用。

口 multimap::size

```
size_type size() const;
```

该成员函数返回被控序列的长度。

口 multimap::size_type

```
typedef T2 size_type;
```

该无符号整数类型描述的对象可以表示任意被控序列的长度。在此处它被描述为由实现定义的类型T2的同义词。

口 multimap::swap

```
void swap(multimap& x);
```

该成员函数在*this和x之间相互交换被控序列。如果get_allocator() == x.get_allocator(), 它将可以在常数时间内完成交换。只有在复制被存储的类型为Pred的函数对象时, 它才有可能抛出异常。另外, 它还不会导致任何指向这两个被控序列中元素的引用、指针以及迭代器无效。否则, 它将以与这两个被控序列的元素个数成比例的次数来调用元素的构造函数以及为元素赋值。

口 multimap::upper_bound

```
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

该成员函数返回一个迭代器, 指向被控序列中第一个使key_comp()(key, x.first)为true的元素x。如果不存在这样的元素, 它将返回end()。

口 multimap::value_comp

```
value_compare value_comp() const;
```

该成员函数返回一个用以检测被控序列中元素次序的函数对象。

口 multimap::value_compare

```
class value_compare
    : public binary_function<value_type, value_type,
        bool> {
public:
    bool operator()(const value_type& x,
                     const value_type& y) const
        {return (comp(x.first, y.first)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
};
```

该类型描述的是一个函数对象, 它可以比较被控序列的两个元素中的排序键, 并检测它们在被控序列中的相对次序。该函数对象存储一个类型为key_compare的对象comp。成员函数operator()使用这个对象来比较两个元素中的排序键部分。

```

口 multimap::value_type
    typedef pair<const Key, T> value_type;
    该类型描述了被控序列中的一个元素。

口 operator!=
    template<class Key, class T, class Pred, class A>
        bool operator!=(
            const map <Key, T, Pred, A>& lhs,
            const map <Key, T, Pred, A>& rhs);
    template<class Key, class T, class Pred, class A>
        bool operator!=(
            const multimap <Key, T, Pred, A>& lhs,
            const multimap <Key, T, Pred, A>& rhs);
    该模板函数返回!(lhs == rhs)。

口 operator==
    template<class Key, class Pred, class A>
        bool operator==(
            const map <Key, T, Pred, A>& lhs,
            const map <Key, T, Pred, A>& rhs);
    template<class Key, class Pred, class A>
        bool operator==(
            const multimap <Key, T, Pred, A>& lhs,
            const multimap <Key, T, Pred, A>& rhs);
    第一个模板函数重载operator==来比较模板类map的两个对象。第二个
    模板函数重载operator==以比较模板类multimap的两个对象。这两个函数都
    返回lhs.size() == rhs.size() && equal(lhs.begin(), lhs.end(), rhs.begin())。

口 operator<
    template<class Key, class T, class Pred, class A>
        bool operator<(
            const map <Key, T, Pred, A>& lhs,
            const map <Key, T, Pred, A>& rhs);
    template<class Key, class T, class Pred, class A>
        bool operator<(
            const multimap <Key, T, Pred, A>& lhs,
            const multimap <Key, T, Pred, A>& rhs);
    第一个模板函数重载operator<来比较模板类map的两个对象。第二个
    模板函数重载operator<以比较模板类multimap的两个对象。这两个函数都
    返回lexicographical_compare( lhs.begin(), lhs.end(), rhs.begin(), rhs.end())。

口 operator<=
    template<class Key, class T, class Pred, class A>
        bool operator<=(
            const map <Key, T, Pred, A>& lhs,
```

```

        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator<=
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

该模板函数返回!(rhs < lhs)。

口 operator>

```

template<class Key, class T, class Pred, class A>
bool operator>(
    const map <Key, T, Pred, A>& lhs,
    const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator>(
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

该模板函数返回rhs < lhs。

口 operator>=

```

template<class Key, class T, class Pred, class A>
bool operator>=(
    const map <Key, T, Pred, A>& lhs,
    const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
bool operator>=(
    const multimap <Key, T, Pred, A>& lhs,
    const multimap <Key, T, Pred, A>& rhs);

```

该模板函数返回!(lhs < rhs)。

口 swap

```

template<class Key, class T, class Pred, class A>
void swap(
    map <Key, T, Pred, A>& lhs,
    map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
void swap(
    multimap <Key, T, Pred, A>& lhs,
    multimap <Key, T, Pred, A>& rhs);

```

该模板函数执行lhs.swap(rhs)。

使用 <map>

map
multimap

如果想在程序中使用模板类map或模板类multimap，请把头文件<map>包含到程序中。只有在允许出现多个元素有着相同的键值时——或

者更确切地说，成对的元素有相等次序时，我们才使用后一个模板类。由于这两个容器很相似，所以我们先把map作为一个例子来描述。在本节的后面部分，我们将着重描述它们之间的不同之处。

可以指定map所存储的元素类型pair<const Key, T>以及类型为less<Key>的排序规则，我们只需这样写一条类型定义语句：

```
typedef map<Key, T, less<Key>,
allocator<pair<const Key, T>> Mycont;
```

通过使用默认的模板参数，可以省略第三个和第四个模板参数。

less

如果省略了第三个参数，默认值将由类型为less<Key>的函数对象给出。容器中所存储的被控序列就是以这种规则来排序的。也就是说，至少对于数字类型来说，序列中最后那个元素将会有最大的值。

模板类map支持所有我们在第9章中所给出的、对于容器来说是常见的一些操作。（参见从第9章的“使用容器”一节开始的讨论。）我们在此仅概括模板类map所特有的属性。

构造函数

为了构造一个类map<Key, T, Pred, A>的对象，可以这么写：

- map(), 声明一个按Pred()排序的空映射；
- map(pr), 声明一个按函数对象pr排序的空映射；
- map(pr, al), 声明一个和上面一样的映射，不过它还存储一个分配器对象al；
- map(first, last), 声明一个按Pred()排序的映射，它的初始内容是从由[first, last)指定的序列（当然，这是一个有序序列）中复制过来的；
- map(first, last, pr), 声明一个按pr排序的映射，它的初始内容是从由[first, last)指定的序列（和上面一样，该序列是一个有序序列）中复制过来的；
- map(first, last, pr, al), 声明一个和上面一样的映射，不过它还存储一个分配器对象al。

如果我们已经为类型为allocator<const Key, T>的分配器特化了模板类（这是我们所常做的并且也是STL提供的默认值），那么再在构造函数中显式地指定分配器参数al并不能带来任何好处。仅对于在程序中显式定义的某些分配器，这样的参数才会起作用。（参见第4章中对分配器的讨论。）

下面所有的描述中都假定cont是类map<Key, T, Pred, A>的一个对象。

clear erase

为了删除容器中的所有元素，请调用cont.clear()。

为了删除由it所指定的元素，请调用cont.erase(it)。它的返回值为一个迭代器，指向紧接着被删除元素的下一个元素。为了删除由区间[first, last)所指定的所有元素，请调用cont.erase(first, last)。

我们也可以执行一些能够从map对象的独一无二的表达方式中获益的操作。尤其是，我们还可以通过调用cont.erase(key)来删除其键值与key

次序相等的元素。其他map(以及其他关联容器)所特有的操作有：

insert

为了插入一个具有{键，值}对的元素val，请调用cont.insert(val)。它的返回值是一个类pair<iterator, bool>的对象ans。只有当被控序列中没有与key次序相等的元素时，成员对象ans.second才会为true。在这种情况下，ans.first就指向被插入的元素。否则，插入将会失败并且ans.first指向与该元素次序相等的已有元素。

为了插入区间[first, last)中所有的{键，值}对元素，请调用cont.insert(first, last)。被插入的序列不能为最初被控序列中的一部分。

为了向it所指定的元素后面插入一个{键，值}对元素val，请调用cont.insert(it, key)。如果“提示”被证明为正确的话，插入将会耗费常数时间而不是对数时间。不管在何种情况下，这样的调用和调用insert(val)的效果都是一样的，并且它们返回同样的迭代器值。

find

为了在被控序列中查找一个键(值)与key次序相等的元素，请调用cont.find(key)。

lower_bound

为了在被控序列中查找到第一个其键(值)不排在key之前的元素，请调用cont.lower_bound(key)。注意，它也就是所有与key次序相等的元素形成的子区间的起始处。

upper_bound

为了在被控序列中查找到第一个其键排在key之后元素，请调用cont.upper_bound(key)。注意，它紧接着所有与key次序相等的元素形成的子区间的末端。

equal_range

为了检测出被控序列中所有元素都与key次序相等的子区间，请调用cont.equal_range(key)。注意，它返回一个这样的迭代器对：pair<iterator, iterator>(cont.lower_bound(key), cont.upper_bound(key))。对于一个map<key, T>对象来说，该子区间的长度要么为0，要么为1。

count

为了检测出调用cont.equal_range(key)所得到的子区间的长度，请调用cont.count(key)。对于一个map<key, T>对象来说，它的返回值要么为0，要么为1。

key_comp

为了获得以和cont同样的规则对键排序的对象，请调用cont.key_comp()。需要指出的是，如果在被控序列中，key1排在key2的前面，那么cont.key_comp()(key1, key2)就为true。

value_comp

为了获得一个可以对cont元素中存储的{键，值}对进行排序的对象，请调用cont.value_comp()。对于map<key, T>对象cont来说，它的返回值和cont.key_comp()相同。

multimap

模板类map和模板类multimap之间的不同之处产生于它们最基本的区别——multimap可以存储有相等次序的元素，而map则不行。值得注意的一个例外情况是：它们所有的构造函数以及成员函数的函数签名都一样。

operator[]

map中还有一个multimap中没有的操作。为了定位键值与key次序相

等的元素，我们可以使用表达式`cont[key]`。如果没有找到这样的元素，该操作符就会向容器中添加元素`{key, mapped_type()}`。

除此之外，对于`multimap`对象来说还有：

`insert`

调用`cont.insert(key)`总是会插入一个新元素，所以函数返回的就是指向这个新元素的迭代器，而不是类`pair<iterator, bool>`的对象。

`find`

如果有多个元素与`key`次序相等，那么没有指定调用`cont.find(key)`返回的迭代器具体指向哪个元素。

`equal_range`

- 调用`cont.equal_range(key)`所得到的子区间的长度可以是区间`[0, cont.size()]`中的任意值。

`count`

- 同样，调用`cont.count(key)`所得到的子区间长度也可以为`[0, cont.size()]`中的任意值。

实现 `<map>`

`map`

程序清单 14-1 列出了文件`map`。它定义了模板类`map`和`multimap`。我们将会发现，它和第 13 章中的文件`set`非常相似。它们都大量依赖于模板类`Tree`（在第 13 章的文件`xtree`中定义）。

程序清单 14-1:

```
// map standard header
#ifndef MAP_
#define MAP_
#include <xtree>
namespace std {
    // TEMPLATE CLASS Tmap_traits
    template<class K, class T, class Pr, class Ax, bool Mfl>
        class Tmap_traits {
    public:
        typedef K key_type;
        typedef pair<const K, T> value_type;
        typedef Pr key_compare;
        typedef typename Ax::template
            rebind<value_type>::other
            allocator_type;
        enum {Multi = Mfl};
        Tmap_traits()
            : comp()
        {}
        Tmap_traits(Pr Parg)
            : comp(Parg)
        {}
        class value_compare
```

```
        : public binary_function<value_type, value_type,
bool> {
    friend class Tmap_traits<K, T, Pr, Ax, Mfl>;
public:
    bool operator()(const value_type& X,
                     const value_type& Y) const
        {return (comp(X.first, Y.first)); }
    value_compare(key_compare Pred)
        : comp(Pred) {}
protected:
    key_compare comp;
};
struct Kfn {
    const K& operator()(const value_type& X) const
        {return (X.first); }
};
Pr comp;
};

// TEMPLATE CLASS map
template<class K, class T,
          class Pr = less<K>,
          class A = allocator<pair<const K, T> >>
class map
    : public Tree<Tmap_traits<K, T, Pr, A, false> > {
public:
    typedef map<K, T, Pr, A> Myt;
    typedef Tree<Tmap_traits<K, T, Pr, A, false> >
        Mybase;
    typedef K key_type;
    typedef T mapped_type;
    typedef T referent_type;
    typedef Pr key_compare;
    typedef typename Mybase::value_compare value_compare;
    typedef typename Mybase::allocator_type
        allocator_type;
    typedef typename Mybase::size_type size_type;
    typedef typename Mybase::difference_type
        difference_type;
    typedef typename Mybase::pointer pointer;
    typedef typename Mybase::const_pointer const_pointer;
    typedef typename Mybase::reference reference;
    typedef typename Mybase::const_reference
        const_reference;
    typedef typename Mybase::iterator iterator;
```

```
        typedef typename Mybase::const_iterator
        const_iterator;
        typedef typename Mybase::reverse_iterator
        reverse_iterator;
        typedef typename Mybase::const_reverse_iterator
        const_reverse_iterator;
        typedef typename Mybase::value_type value_type;
map()
    : Mybase(key_compare(), allocator_type()) {}
explicit map(const key_compare& Pred)
    : Mybase(Pred, allocator_type()) {}
map(const key_compare& Pred, const allocator_type& Al)
    : Mybase(Pred, Al) {}
template<class It>
map(It F, It L)
    : Mybase(key_compare(), allocator_type())
{for (; F != L; ++F)
    insert(*F); }
template<class It>
map(It F, It L, const key_compare& Pred)
    : Mybase(Pred, allocator_type())
{for (; F != L; ++F)
    insert(*F); }
template<class It>
map(It F, It L, const key_compare& Pred,
const allocator_type& Al)
    : Mybase(Pred, Al)
{for (; F != L; ++F)
    insert(*F); }
mapped_type& operator[](const key_type& Kv)
{iterator P =
    insert(value_type(Kv, mapped_type())).first;
return (*P).second; }
};

// TEMPLATE CLASS multimap
template<class K, class T,
class Pr = less<K>,
class A = allocator<pair<const K, T>>>
class multimap
    : public Tree<Tmap_traits<K, T, Pr, A, true>> {
public:
    typedef multimap<K, T, Pr, A> Myt;
    typedef Tree<Tmap_traits<K, T, Pr, A, true>>
    Mybase;
```

```
typedef K key_type;
typedef T mapped_type;
typedef T referent_type; //old name, magically gone
typedef Pr key_compare;
typedef typename Mybase::value_compare value_compare;
typedef typename Mybase::allocator_type
allocator_type;
typedef typename Mybase::size_type size_type;
typedef typename Mybase::difference_type
difference_type;
typedef typename Mybase::pointer pointer;
typedef typename Mybase::const_pointer const_pointer;
typedef typename Mybase::reference reference;
typedef typename Mybase::const_reference
const_reference;
typedef typename Mybase::iterator iterator;
typedef typename Mybase::const_iterator
const_iterator;
typedef typename Mybase::reverse_iterator
reverse_iterator;
typedef typename Mybase::const_reverse_iterator
const_reverse_iterator;
typedef typename Mybase::value_type value_type;
multimap()
    : Mybase(key_compare(), allocator_type()) {}
explicit multimap(const key_compare& Pred)
    : Mybase(Pred, allocator_type()) {}
multimap(const key_compare& Pred, const allocator_type&
A1)
    : Mybase(Pred, A1) {}
template<class It>
multimap(It F, It L)
    : Mybase(key_compare(), allocator_type())
    {for (; F != L; ++F)
        insert(*F); }
template<class It>
multimap(It F, It L, const key_compare& Pred)
    : Mybase(Pred, allocator_type())
    {for (; F != L; ++F)
        insert(*F); }
template<class It>
multimap(It F, It L, const key_compare& Pred,
const allocator_type& A1)
    : Mybase(Pred, A1)
    {for (; F != L; ++F)
```

```

        insert(*F); }
iterator insert(const value_type& X)
    {return (Mybase::insert(X).first); }
iterator insert(iterator P, const value_type& X)
    {return (Mybase::insert(P, X)); }
template<class It>
void insert(It F, It L)
{for (; F != L; ++F)
    insert(*F); }
} /* namespace std */
#endif /* MAP_ */

```

Tmap_traits

模板类 `Tmap_traits` 是上一章所讲述的 `Tree traits` 类的一个版本，它适合用来定义映射。如同我们可以看到的一样，它并没有做很多的事情，但它却可以使我们在特化一个 `Tree` 时不再需要写原来那么长的一串参数列表。注意，和集合不同的是，映射的 `key_type` 并不是 `value_type` 的同义词。也就是说，我们仍然需要做一些事情才能用 `Kfn` 提取键。

multimap

除了回送一些类型定义，提供适当的构造函数，以及定义 `operator[]` 之外，模板类 `map` 并没其他事情要做。模板类 `multimap` 和 `map` 差不多，它们之间只有一些小小的区别。当然，`multimap` 省略了 `operator[]`。它也包装了对 `Tree::insert` 的调用，这最主要是为了简化成员函数 `insert(const value_type&)` 的返回值。和模板类 `multiset` 中的包装一样，其他这些包装不是严格必要的，但它们还是可以帮助某些编译器避免陷入混乱状态。

测试 `<map>`

程序清单 14-2 列出了文件 `tmap.c`。它提供了对于两个相似的模板类 `map` 和 `multimap` 的测试。它实际上也是两个很相似的测试程序中的一个。另外一个是第 13 章中的文件 `tset.c`。为了简化对这两个文件中四个测试的比较，我们只是简单地注释出对于给定容器的特定测试。

程序清单 14-2:

```

tmap.c // test <map>
#include <assert.h>
#include <iostream>
#include <functional>
#include <map>
using namespace std;

// TEST map
void test_map()
{typedef allocator<int> Myal;

```

```
typedef less<char> Mypred;
typedef pair<const char, int> Myval;
typedef map<char, int, Mypred, Myval> Mycont;
Myval x, xarr[3], xarr2[3];
for (int i = 0; i < 3; ++i)
    {new (&xarr[i]) Myval('a' + i, 1 + i);
     new (&xarr2[i]) Myval('d' + i, 4 + i); }

Mycont::key_type *p_key = (char *)0;
Mycont::mapped_type *p_mapped = (int *)0;
Mycont::key_compare *p_kcomp = (Mypred *)0;
Mycont::allocator_type *p_alloc = (Myal *)0;
Mycont::value_type *p_val = (Myval *)0;
Mycont::value_compare *p_vcomp = 0;
Mycont::pointer p_ptr = (Myval *)0;
Mycont::const_pointer p_cptr = (const Myval *)0;
Mycont::reference p_ref = x;
Mycont::const_reference p_cref=(const Myval&)x;
Mycont::size_type *p_size = (size_t *)0;
Mycont::difference_type *p_diff=(ptrdiff_t *)0;

Mycont v0;
Myal al = v0.get_allocator();
Mypred pred;
Mycont v0a(pred), v0b(pred, al);
assert(v0.empty() && v0.size() == 0);
assert(v0a.size()==0 && v0a.get_allocator() == al);
assert(v0b.size()==0 && v0b.get_allocator() == al);
Mycont v1(xarr, xarr + 3);
assert(v1.size()==3 && (*v1.begin()).first == 'a');
Mycont v2(xarr, xarr + 3, pred);
assert(v2.size()==3 && (*v2.begin()).first == 'a');
Mycont v3(xarr, xarr + 3, pred, al);
assert(v3.size()==3 && (*v3.begin()).first == 'a');
const Mycont v4(xarr, xarr + 3);
assert(v4.size()==3 && (*v4.begin()).first == 'a');
v0 = v4;
assert(v0.size()==3 && (*v0.begin()).first == 'a');

assert(v0.size() <= v0.max_size());

Mycont::iterator p_it(v1.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v1.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
```

```
assert((*p_it).first == 'a' && (*p_it).second == 1
      && (*--(p_it = v1.end())).first == 'c');
assert((*p_cit).first == 'a'
      && (*--(p_cit = v4.end())).first == 'c');
assert((*p_rit).first == 'c' && (*p_rit).second==3
      && (*--(p_rit = v1.rend())).first == 'a');
assert((*p_crit).first == 'c'
      && (*--(p_crit = v4.rend())).first == 'a');

v0.clear(); // DIFFERS FROM multimap
pair<Mycont::iterator, bool>pib=v0.insert(Myval('d', 4));
assert((*pib.first).first == 'd' && pib.second);
assert((*--v0.end()).first == 'd');
pib = v0.insert(Myval('d', 5));
assert((*pib.first).first == 'd'
      && (*pib.first).second == 4 && !pib.second);
assert((*v0.insert(v0.begin(), Myval('e', 5))).first == 'e');
v0.insert(xarr, xarr + 3);
assert(v0.size() == 5 && (*v0.begin()).first=='a');
v0.insert(xarr2, xarr2 + 3);
assert(v0.size() == 6 && (*--v0.end()).first=='f');
assert(v0['c'] == 3);
assert((*v0.erase(v0.begin())).first == 'b'
      && v0.size() == 5);
assert((*v0.erase(v0.begin(),++v0.begin())).first == 'c'
      && v0.size() == 4);
assert(v0.erase('x') == 0 && v0.erase('e') == 1);

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

assert(v0.key_comp()('a', 'c')
      && !v0.key_comp()('a', 'a'));
assert(v0.value_comp()(Myval('a', 0), Myval('c', 0))
      && !v0.value_comp()(Myval('a', 0),Myval('a',1)));
assert((*v4.find('b')).first == 'b');
assert(v4.count('x') == 0 && v4.count('b') == 1);
assert((*v4.lower_bound('a')).first == 'a');
```

```
assert((*v4.upper_bound('a')).first == 'b');
pair<Mycont::const_iterator, Mycont::const_iterator> pcc
= v4.equal_range('a');
assert((*pcc.first).first == 'a'
&& (*pcc.second).first == 'b'); }

// TEST multimap
void test_multimap()
{typedef allocator<int> Myal;
typedef less<char> Mypred;
typedef pair<const char, int> Myval;
typedef multimap<char, int, Mypred, Myal> Mycont;
Myval x, xarr[3], xarr2[3];
for (int i = 0; i < 3; ++i)
    {new (&xarr[i]) Myval('a' + i, 1 + i);
     new (&xarr2[i]) Myval('d' + i, 4 + i); }

Mycont::key_type *p_key = (char *)0;
Mycont::mapped_type *p_mapped = (int *)0;
Mycont::key_compare *p_kcomp = (Mypred *)0;
Mycont::allocator_type *p_alloc = (Myal *)0;
Mycont::value_type *p_val = (Myval *)0;
Mycont::value_compare *p_vcomp = 0;
Mycont::pointer p_ptr = (Myval *)0;
Mycont::const_pointer p_cptr = (const Myval *)0;
Mycont::reference p_ref = x;
Mycont::const_reference p_cref = (const Myval&)x;
Mycont::size_type *p_size = (size_t *)0;
Mycont::difference_type *p_diff = (ptrdiff_t *)0;

Mycont v0;
Myal al = v0.get_allocator();
Mypred pred;
Mycont v0a(pred), v0b(pred, al);
assert(v0.empty() && v0.size() == 0);
assert(v0a.size() == 0 && v0a.get_allocator() == al);
assert(v0b.size() == 0 && v0b.get_allocator() == al);
Mycont v1(xarr, xarr + 3);
assert(v1.size() == 3 && (*v1.begin()).first == 'a');
Mycont v2(xarr, xarr + 3, pred);
assert(v2.size() == 3 && (*v2.begin()).first == 'a');
Mycont v3(xarr, xarr + 3, pred, al);
assert(v3.size() == 3 && (*v3.begin()).first == 'a');
const Mycont v4(xarr, xarr + 3);
assert(v4.size() == 3 && (*v4.begin()).first == 'a');
```

```
v0 = v4;
assert(v0.size() == 3 && (*v0.begin()).first == 'a');

assert(v0.size() <= v0.max_size());

Mycont::iterator p_it(v1.begin());
Mycont::const_iterator p_cit(v4.begin());
Mycont::reverse_iterator p_rit(v1.rbegin());
Mycont::const_reverse_iterator p_crit(v4.rbegin());
assert((*p_it).first == 'a' && (*p_it).second == 1
    && (*--(p_it = v1.end())).first == 'c');
assert((*p_cit).first == 'a'
    && (*--(p_cit = v4.end())).first == 'c');
assert((*p_rit).first == 'c' && (*p_rit).second == 3
    && (*--(p_rit = v1.rend())).first == 'a');
assert((*p_crit).first == 'c'
    && (*--(p_crit = v4.rend())).first == 'a');

v0.clear(); // DIFFERS FROM map
assert((*v0.insert(Myval('d', 4))).first == 'd');
assert((*--v0.end()).first == 'd');
assert((*v0.insert(Myval('d', 5))).first == 'd');
assert(v0.size() == 2);
assert((*v0.insert(v0.begin(), Myval('e', 5))).first == 'e');
v0.insert(xarr, xarr + 3);
assert(v0.size() == 6 && (*v0.begin()).first == 'a');
v0.insert(xarr2, xarr2 + 3);
assert(v0.size() == 9 && (*--v0.end()).first == 'f');
assert((*v0.erase(v0.begin()))).first == 'b'
    && v0.size() == 8);
assert((*v0.erase(v0.begin(), ++v0.begin()))).first == 'c'
    && v0.size() == 7);
assert(v0.erase('x') == 0 && v0.erase('e') == 2);

v0.clear();
assert(v0.empty());
v0.swap(v1);
assert(!v0.empty() && v1.empty());
swap(v0, v1);
assert(v0.empty() && !v1.empty());
assert(v1 == v1 && v0 < v1);
assert(v0 != v1 && v1 > v0);
assert(v0 <= v1 && v1 >= v0);

assert(v0.key_comp()('a', 'c')
```

```
    && !v0.key_comp()('a', 'a'));
assert(v0.value_comp()(Myval('a', 0), Myval('c', 0))
    && !v0.value_comp()(Myval('a', 0), Myval('a', 1)));
assert((*v4.find('b')).first == 'b');
assert(v4.count('x') == 0 && v4.count('b') == 1);
assert((*v4.lower_bound('a')).first == 'a');
assert((*v4.upper_bound('a')).first == 'b');
pair<Mycont::const_iterator, Mycont::const_iterator> pcc
= v4.equal_range('a');
assert((*pcc.first).first == 'a'
    && (*pcc.second).first == 'b'); }

// TEST <map>
int main()
{test_map();
 test_multimap();
 cout << "SUCCESS testing <map>" << endl;
 return (0); }
```

该测试程序只是对模板类map的一个特化版本和模板类multimap的一个特化版本进行了简单的测试，按照其各自的意图测试了所给出的每个成员函数和成员类型。如果测试一切顺利的话，测试程序将打印出：

SUCCESS testing <map>

然后正常退出。

习题

习题14-1

将一个映射实现为一个集合是否可行？请解释为什么？

习题14-2

将一个集合实现为一个映射是否可行？请解释为什么？

习题14-3

为模板类multimap定义operator[]。如果这个函数可以返回多个值，请提供至少三个可供选择的有意义的版本。

习题14-4

为模板类multimap实现你认为最好的operator[]版本。

习题14-5

[较难] 改变模板类map的定义，使之在构造好后也允许改变排序规则。

习题14-6

[特难] 改变模板类map的定义，使得它对于两个或多个排序规则能做到对数时间内的查找。

第15章 <stack>

背景知识

<stack>

stack

头文件<stack>中只定义了模板类 stack。它是一个容器适配器 (container adapter) —— 即自己不直接维护被控序列的模板类。确切地说，是它存储的容器对象来为它实现所有的功能。模板类 stack 是 STL 中最简单的容器适配器（其他的还有模板类 queue 和 priority_queue，我们将在下一章描述它们）。

容器适配器

那么什么是适配器呢？唔，一个典型的STL模板容器类中有许多的成员函数。提供所有对被控序列可能有意义的操作这个主意很不错。记住，这些容器都是被设计成能够大量重用的。因此，提供一个可能没有被给定的应用程序使用的成员函数要比要求应用程序对容器进行扩展简单得多。但是，存取的弹性并不总是一个优点。

例如，假设我们需要的是一个栈，或称为后进先出 (LIFO) 队列，它是一种维护着严格存取原则的栈。我们所能看到被控序列中的部分只是栈的顶端元素——即我们最后插入（或压入）栈中的那个元素。我们所惟一能够从栈中删除（或弹出）的也只是它的顶端元素。栈中其他所有元素都被隐藏起来，就像在餐厅中一摞盘子的中间那些盘子一样。

vector

我们可能会选择模板类vector（参见第10章）来实现一个栈。如果这么做了，我们就可以比在任意的栈中更容易且更快速地存取那些被隐藏的元素。但这并不代表我们一定要这么做。面向对象的设计 (object-oriented design) 中很突出信息隐藏的重要性。例如，我们将一个类的成员对象声明为私有型以防止程序员写出直接依赖于类的特定实现的代码来。这样我们就有了更多的弹性来改变（并且可能提升）以前的实现，并且不用担心破坏了以前使用这个类的代码或是破坏了代码中对于类成员的直接存取。

同样，我们也可能希望隐藏模板类vector中其他额外的功能。和它的私有成员对象一样，向量的成员函数在栈的实现中暴露得太多。由于这个原因，我们希望避免在由模板类派生栈的时候带来的“是一个 (is-a)”的关系，如：

```
template<class T>
class stack : public vector<T>; // "is-a" vector
```

push
pop
top

这样的派生使得我们可以添加一些成员函数，如：push、pop和top等；但它对隐藏我们所希望隐藏的成员函数并没有任何帮助。不错，我们可以使基类为保护型或私有型，但在很多方面栈仍然“是一个”向量，这样的关系是不正确的。

stack

于是，STL提供了一个模板类stack，它里面有一个（has-a）容器作为它的成员对象。实现栈的被控序列实际上是存储于这个容器中的。然而，不管该容器支持什么样的操作，栈只是让我们能够压入和弹出元素。于是，模板类被声明为：

```
template<class T, class C = deque<T>>
class stack;
```

模板参数T指定栈中所存储的元素的类型。模板参数C指定用来控制元素序列的容器类型。

deque

通过使用默认的模板参数，我们就可以这么写：stack<int>，翻译器会自动提供容器类型deque<int>。双队列很适合用来作为栈。（参见第12章。）否则，我们必须自己指定容器的类型。于是，一切就依赖于我们自己了，必须确保我们的容器C可以正确地存储类型为T的元素。我们将在下面详细地描述对于这样的容器的要求。现在，我们只需知道有三个STL的容器（vector、deque和list）能够满足模板类stack的所有要求就够了。

功能描述

```
namespace std {
    template<class T, class Cont>
        class stack;

        // TEMPLATE FUNCTIONS
    template<class T, class Cont>
        bool operator==(const stack<T, Cont>& lhs,
                        const stack<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator!=(const stack<T, Cont>& lhs,
                        const stack<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator<(const stack<T, Cont>& lhs,
                        const stack<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator>(const stack<T, Cont>& lhs,
                        const stack<T, Cont>& rhs);
    template<class T, class Cont>
        bool operator<=(const stack<T, Cont>& lhs,
                        const stack<T, Cont>& rhs);
}
```

```

        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const stack<T, Cont>& lhs,
        const stack<T, Cont>& rhs);
};

```

包含STL的标准头文件<stack>可以得到模板类stack，以及两个支持模板的定义。

■ operator!=

```

template<class T, class Cont>
    bool operator!=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数返回!(lhs == rhs)。

■ operator==

```

template<class T, class Cont>
    bool operator==(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数重载operator==来比较模板类stack的两个对象。函数返回lhs.c == rhs.c。

■ operator<

```

template<class T, class Cont>
    bool operator<(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数重载operator<来比较模板类stack的两个对象。函数返回lhs.c < rhs.c。

■ operator<=

```

template<class T, class Cont>
    bool operator<=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数返回!(rhs < lhs)。

■ operator>

```

template<class T, class Cont>
    bool operator>(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数返回rhs < lhs。

■ operator>=

```

template<class T, class Cont>
    bool operator>=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);

```

该模板函数返回!(lhs < rhs)。

```

# stack

    template<class T,
              class Cont = deque<T> >
    class stack {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    stack();
    explicit stack(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    };

```

该模板类描述的对象控制一个可变长度的元素序列。该对象通过类Cont的一个保护型对象c来为它所控制的序列分配和释放存储空间。被控序列中元素的类型T必须匹配value_type。

类Cont的对象必须提供几个和deque、list和vector中一样的公共成员（它们都是类Cont的合适候选）。所需要的成员包括：

```

typedef T value_type;
typedef T0 size_type;
Cont();
bool empty() const;
size_type size() const;
value_type& back();
const value_type& back() const;
void push_back(const value_type& x);
void pop_back();
bool operator==(const Cont& X) const;
bool operator!=(const Cont& X) const;
bool operator<(const Cont& X) const;
bool operator>(const Cont& X) const;
bool operator<=(const Cont& X) const;
bool operator>=(const Cont& X) const;

```

在此，T0是满足上述需求的一个未指明的类型。

stack::container_type

```
typedef Cont container_type;
```

该类型是模板参数Cont的同义词。

```

■ stack::empty
    bool empty() const;
    当被控序列为为空时，该成员函数返回true。

■ stack::pop
    void pop();
    该成员函数删除被控序列的最后一个元素，在执行pop前序列不能为空。

```

```

■ stack::push
    void push(const T& x);
    该成员函数向被控序列末端插入一个值为x的元素。

```

```

■ stack::size
    size_type size() const;
    该成员函数返回被控序列的长度。

```

```

■ stack::size_type
    typedef typename Cont::size_type size_type;
    该类型是Cont::size_type的同义词。

```

```

■ stack::stack
    stack();
    explicit stack(const container_type& cont);
    第一个构造函数将存储的对象初始化为c(), 以此来指定一个空的初始被控序列。第二个构造函数将存储的对象初始化为c(cont)，把被控序列初始化为cont所控制序列的一个拷贝。

```

```

■ stack::top
    value_type& top();
    const value_type& top() const;
    该成员函数返回一个引用，指向被控序列中最后那个元素，此时被控序列不能为空。

```

```

■ stack::value_type
    typedef typename Cont::value_type value_type;
    该类型是Cont::value_type的同义词。

```

使用 <stack>

stack

如果想在程序中使用模板类stack，请把头文件<stack>包含到程序中。我们可以特化模板类stack以在类型为deque<T>的对象中存储类型为T的元素，只需这样写一条类型定义语句：

```
typedef stack<T, deque<T, allocator<T> > > Mycont;
```

通过使用默认的模板参数，可以省略第二个模板参数。

作为一种可能的方案，可以用 `vector<T>` 或 `list<T>` 来替换第二个参数。当不需要频繁地进行存储空间的重新分配时，向量可以作为小且高效的栈。记住，我们可以通过调用 `vec.reserve(n)` 为一个向量对象 `vec` 预先分配一部分的空间（最少可以容纳 `n` 个元素）以满足以后可能的增长，这样我们就不需要经常性地进行重新分配了。

对容器的要求

`value_type`

`size_type`

`empty`

`size`

`back`

`push_back`

`pop_back`

构造函数

`push`

`pop`

`top`

`empty`

`size`

比较操作

我们也可以指定第二个参数为自己设计的容器类型。确切地说，容器 `Cont` 至少应该提供如下的外部接口：

- 它必须定义成员类型 `value_type`（如 `T`）和 `size_type`（通常是 `size_t`）。

• 它必须定义常量成员函数 `empty` 和 `size`，并且它们具有通常的意义。

• 它必须以常量和非常量两种形式定义成员函数 `back`，并且它们具有通常的意义。

• 它必须定义成员函数 `push_back` 和 `pop_back`，并且它们具有通常的意义。

• 它必须以对类型为 `Cont` 的对象施加完全排序的方式，定义那六种比较操作符（如 `operator==(const Cont&, const Cont&)`）。

模板类 `stack` 并不支持所有我们在第 9 章中所给出的、对于容器来说是常见的一些操作。（参见从第 9 章的“使用容器”一节开始的讨论。）它只是有意提供一系列受限制的操作。下面的描述都假设 `cont` 是类 `stack<T, C>` 的一个对象。

为了构造类 `stack<T, C>` 的一个对象，我们可以这么写：

- `stack()`，声明一个空栈。
- `stack(cont)`，声明一个初始内容复制自 `cont` 的栈。

为了将一个值为 `v` 的新元素压入到栈中，请调用 `cont.push(v)`。

为了弹出栈中的元素，请调用 `cont.pop()`。如果栈为空，结果就是未定义的。注意，`cont.pop()` 是一个 `void` 型函数——它并不返回被弹出元素的值。

为了存取栈的顶端元素，请调用 `cont.top()`。如果栈为空，它的结果就是未定义的。注意，`cont.top()` 返回的是一个引用——如果栈不是一个常量对象，我们就可以改变存储在顶端元素中的值，只需这么写一条赋值表达式：`cont.top() = 0`。

为了测试栈是否为空，请调用 `cont.empty()`。为了获得栈中元素的个数，请调用 `cont.size()`。

通过常见的那六个操作符，可以把栈 `cont` 和另一个栈 `cont2` 做全面的比较，如：`cont == cont2` 或者 `cont >= cont2`。它们的返回值与比较存储的容器对象得到的返回值一样。

c 最后，如果定义了一个以stack<T, C>为公共基类的类，那么我们就可以在存取存储的容器对象。这个对象就是保护型成员c。

实现 <stack>

stack

程序清单 15-1 列出了文件 stack。如同我们所能看见的一样，它的所有操作都是通过调用存储的容器对象中的相应成员函数来实现的。它是一个比较简单的模板类。

```
程序清单 15-1: // stack standard header
stack
#ifndef STACK_
#define STACK_
#include <deque>
namespace std {
    // TEMPLATE CLASS stack
    template<class T, class C = deque<T> >
    class stack {
public:
    typedef C container_type;
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    explicit stack(const C& Cont)
        : c(Cont) {}
    stack()
        : c() {}
    bool empty() const
        {return (c.empty()); }
    size_type size() const
        {return (c.size()); }
    value_type& top()
        {return (c.back()); }
    const value_type& top() const
        {return (c.back()); }
    void push(const value_type& X)
        {c.push_back(X); }
    void pop()
        {c.pop_back(); }
    bool Eq(const stack<T, C>& X) const
        {return (c == X.c); }
    bool Lt(const stack<T, C>& X) const
        {return (c < X.c); }
protected:
    C c;
```

```

};

// stack TEMPLATE FUNCTIONS
template<class T, class C> inline
bool operator==(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (X.Eq(Y)); }
template<class T, class C> inline
bool operator!=(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (!(X == Y)); }
template<class T, class C> inline
bool operator<(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (X.Lt(Y)); }
template<class T, class C> inline
bool operator>(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (Y < X); }
template<class T, class C> inline
bool operator<=(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (!(Y < X)); }
template<class T, class C> inline
bool operator>=(const stack<T, C>& X,
                  const stack<T, C>& Y)
{return (!(X < Y)); }
} /* namespace std */
#endif /* STACK_ */

```

Eq
Lt

在实现它时惟一特殊的地方就是使用了两个成员函数Eq和Lt，它们实现了紧接着模板类定义后面的所有模板操作符中的比较功能。这些“秘密名字”为那六个比较操作符提供了对于保护型成员c的最小化存取方式。

测试 <stack>

tstack.c

程序清单15-2列出了文件tstack.c。由于头文件本身很小且要测试的内容不多，所以它非常简单。它只是检测了明显存在的定义。它用了几个STL的容器来特化模板类stack，然后以明显的方式来检测它们的成员函数。如果测试一切顺利的话，测试程序将打印出：

```
SUCCESS testing <stack>
```

然后程序正常退出。

程序清单 15-2:

```
tstack.c // test <stack>
#include <assert.h>
#include <iostream>
#include <deque>
#include <list>
#include <stack>
#include <vector>
using namespace std;

// TEST <stack>
int main()
{
    {typedef allocator<char> Myal;
    typedef deque<char, Myal> Myimpl;
    typedef stack<char, Myimpl> Mycont;
    typedef list<char, Myal> Myimpl2;
    typedef stack<char, Myimpl2> Mycont2;
    typedef vector<char, Myal> Myimpl3;
    typedef stack<char, Myimpl3> Mycont3;
    Mycont::container_type *p_cont = (Myimpl *)0;
    Mycont::value_type *p_val = (char *)0;
    Mycont::size_type *p_size = (size_t *)0;

    Mycont v0(Myimpl(3, 'x')), v0a;
    Mycont2 v1;
    Mycont3 v2;
    assert(v0.size() == 3 && v0.top() == 'x');
    assert(v0a.empty());
    v0 = v0a;
    v0.push('a');
    assert(v0.size() == 1 && v0.top() == 'a');
    v0.push('b');
    assert(v0.size() == 2 && v0.top() == 'b');
    v0.push('c');
    assert(v0.size() == 3 && v0.top() == 'c');
    assert(v0 == v0 && v0a < v0);
    assert(v0 != v0a && v0 > v0a);
    assert(v0a <= v0 && v0 >= v0a);
    v0.pop();
    assert(v0.top() == 'b');
    v0.pop();
    assert(v0.top() == 'a');
```

```
v0.pop();
assert(v0.empty());

cout << "SUCCESS testing <stack>" << endl;
return (0); }
```

习题

- 习题15-1 对于作为栈的基础表示的容器来说，对它们的绝对最小需求有哪些？
- 习题15-2 为什么那六个模板比较操作符要作为一个有两个参数的函数定义在模板类stack外面，而不是作为有一个参数的成员函数定义在它里面？
- 习题15-3 为什么模板类stack要同时定义Eq和Lt？可不可以去掉它们中的一个并仍能实现那六个模板比较操作符？
- 习题15-4 在什么情况下，我们可能希望使用list<T>来作为栈的基础容器？
- 习题15-5 在什么情况下，我们可能会从stack<T, C>派生出一个类？为什么我们希望能存取被存储的容器对象c？
- 习题15-6 修改模板类stack，使得当程序对一个空栈进行top或pop操作时能够抛出异常。
- 习题15-7 [较难] 修改模板类stack，使得它维护一个自身的被控序列，而不是依赖于其他的容器类型来完成这件事情。你认为哪一种数据表示最适合作为一个通用的栈？
- 习题15-8 [特难] 实现一个栈，它可以动态适应压入和弹出的模式，从而在压入和弹出的时间为可分摊的常数时间时，使得没有被使用的存储空间最小。

第 16 章 <queue>

背景知识

<queue> 头文件<queue>中定义了两个模板类：queue 和 priority_queue。它们都是容器适配器——即自己并没有直接维护被控序列的模板类。确切地说，是它里面存储的容器对象来为它实现所有的功能。在前一章，我们已经讨论过了最简单的容器适配器：模板类 stack。

queue 模板类 queue 实现的是先进先出（FIFO）的策略。而栈实现的则是后进先出（LIFO）的策略。在程序开发中，有很多的机会会用到队列，它们至少和栈一样普遍。这两者在结构上来说也非常相似。它们也和栈一样有着重用性的问题——对于实现一个通用的队列来说，没有任何单个数据结构会优于其他所有的数据结构。于是在此我们就利用了将栈和队列打包成一个容器适配器的好处。可以为一个给定的应用程序选择我们认为最好的基础表示，然后再特化该容器适配器以产生我们所需的那种栈或者队列。

priority_queue 两个容器适配器中的后者也是最复杂的那个。和简单的 queue 以及 stack 一样，模板类 priority_queue 也限制了对被控序列的存取，但它还有着一些额外的要求。它实质上也是一个队列，不过这个队列以一个谓词来检测它里面哪个元素拥有最高的优先级。该模板类可以确保每次通过 pop 从它里面所取得的元素都是剩下元素中优先级最高的那个。为了做到这一点，在每次通过 push 向它里面加入元素时，它所控制的整个序列都会在必要时被重排。更确切地讲，它把被控序列当作一个堆来维护，使用了一些我们在第 6 章中描述过的算法。

从很多方面看来，模板类priority_queue都是我们所看到的STL机制中最尖端的一部分。它使用了已有的容器来管理控制一个序列的细节，用到了很多对性能及复杂度的折衷。它使用堆算法来保持序列有序化。它使得我们可以指定不同的函数对象来检测一个给定的优先队列（priority queue）的“最高优先级”的意义。

我们用队列和优先队列来作为讲解STL的收尾是很合适的。因为它们大量地向我们展示了STL的威力、弹性以及可扩展性。

功能描述

```
namespace std {
    template<class T, class Cont>
        class queue;
```

```

template<class T, class Cont, class Pred>
    class priority_queue;

        // TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
template<class T, class Cont>
    bool operator!=(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
template<class T, class Cont>
    bool operator<(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
template<class T, class Cont>
    bool operator>(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
template<class T, class Cont>
    bool operator<=(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
template<class T, class Cont>
    bool operator>=(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);
};

```

包含STL的标准头文件<queue>可以得到模板类priority_queue、queue，以及几个支持模板的定义。

□ operator!=

```

template<class T, class Cont>
    bool operator!=(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);

```

该模板函数返回!(lhs == rhs)。

□ operator==

```

template<class T, class Cont>
    bool operator==(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);

```

该模板函数重载operator==来比较模板类queue的两个对象。函数返回lhs.c == rhs.c。

□ operator<

```

template<class T, class Cont>
    bool operator<(const queue<T, Cont>& lhs,
                      const queue<T, Cont>& rhs);

```

该模板函数重载operator<来比较模板类queue的两个对象。函数返回lhs.c < rhs.c。

```

□ operator<=
    template<class T, class Cont>
        bool operator<=(const queue <T, Cont>& lhs,
                      const queue <T, Cont>& rhs);
    该模板函数返回!(rhs < lhs)。

□ operator>=
    template<class T, class Cont>
        bool operator>(const queue <T, Cont>& lhs,
                      const queue <T, Cont>& rhs);
    该模板函数返回rhs < lhs.

□ operator>=
    template<class T, class Cont>
        bool operator>=(const queue <T, Cont>& lhs,
                      const queue <T, Cont>& rhs);
    该模板函数返回!(lhs < rhs)。

□ priority_queue
    template<class T,
              class Cont = vector<T>,
              class Pred = less<typename Cont::value_type> >
        class priority_queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    priority_queue();
    explicit priority_queue(const Pred& pr);
    priority_queue(const Pred& pr,
                  const container_type& cont);
    priority_queue(const priority_queue& x);
    template<class InIt>
        priority_queue(InIt first, InIt last);
    template<class InIt>
        priority_queue(InIt first, InIt last,
                      const Pred& pr);
    template<class InIt>
        priority_queue(InIt first, InIt last,
                      const Pred& pr, const container_type& cont);
    bool empty() const;
    size_type size() const;
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:

```

```

Cont _c;
Pred comp;
};

```

该模板类描述的对象控制一个可变长度的元素序列。该对象通过一个类型为Cont的保护型对象来为它所控制的序列分配和释放存储空间。被控序列中元素的类型T必须匹配value_type。

该序列通过一个保护型对象comp来进行排序。在每次对最顶端的元素（在位置0处的元素）进行插入或删除后，假设迭代器P0和Pi指向位置0和i处的元素，那么comp(*P0, *Pi)就为false。[对默认的模板参数less<typename Cont::value_type>来说，序列中最顶端的元素有着最大（或最高）的优先级。]

类Cont的对象必须能够提供随机存取迭代器以及几个和deque或vector中一样的公共成员（deque和vector都是类Cont的合适候选）。所需要的成员包括：

```

typedef T value_type;
typedef T0 size_type;
typedef T1 iterator;
Cont();
template<class InIt>
    Cont(InIt first, InIt last);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator begin();
iterator end();
bool empty() const;
size_type size() const;
const value_type& front() const;
void push_back(const value_type& x);
void pop_back();

```

在此，T0和T1都是满足上述需求的未指明的类型。

- priority_queue::container_type
typedef typename Cont::container_type container_type;

该类型是模板参数Cont的同义词。

- priority_queue::empty
bool empty() const;

当被控序列为空时，该成员函数返回true。

- priority_queue::pop
void pop();

该成员函数将被控序列的最后一个元素从序列中删除（在执行pop前

序列不能为空），然后再重排整个序列。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>

namespace std {
    class priority_queue {
        public:
            priority_queue();
            explicit priority_queue(const Pred& pr);
            priority_queue(const Pred& pr,
                           const container_type& cont);
            priority_queue(const priority_queue& x);
            template<class InIt>
            priority_queue(InIt first, InIt last);
            template<class InIt>
            priority_queue(InIt first, InIt last,
                           const Pred& pr);
            template<class InIt>
            priority_queue(InIt first, InIt last,
                           const Pred& pr, const container_type& cont);
    };
}
```

所有带有参数cont的构造函数都会将存储的对象初始化为c(cont)，剩下的构造函数则把被存储对象初始化为c，以指定一个空的初始被控序列。然后，后三个构造函数就会调用c.insert(c.end(), first, last)。

所有的构造函数还会往comp中存储一个函数对象。如果有的话，该函数对象就是参数pr。对于复制构造函数来说，它就是x.comp。否则，该函数对象就是Pred()。

通过调用make_heap(c.begin(), c.end(), comp)，我们就把一个初始的非空被控序列给排好序了。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>
```

该成员函数往被控序列末端插入一个值为x的元素，然后重排该序列。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>
```

该成员函数返回被控序列的长度。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>
```

该类型是Cont::size_type的同义词。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>
```

该成员函数返回一个引用，指向被控序列中第一个（优先级最高的）元素，此时被控序列不能为空。

```
#include <utility>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include <limits>
#include <new>
```

该类型是Cont::value_type的同义词。

口 queue

```
template<class T,
         class Cont = deque<T> >
class queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    queue();
    explicit queue(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& back();
    const value_type& back() const;
    value_type& front();
    const value_type& front() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont *c;
};
```

该模板类描述的对象控制一个可变长度的元素序列。该对象通过一个类型为Cont的保护型对象c来为它所控制的序列分配和释放存储空间。被控序列中元素的类型T必须匹配value_type。

类Cont的对象必须提供几个和deque或list中一样的公共成员（deque和list都是类Cont的合适候选）。所需要的成员包括：

```
typedef T value_type;
typedef T0 size_type;
Cont();
bool empty() const;
size_type size() const;
value_type& front();
const value_type& front() const;
value_type& back();
const value_type& back() const;
void push_back(const value_type& x);
void pop_front();
bool operator==(const Cont& X) const;
bool operator!=(const Cont& X) const;
bool operator<(const Cont& X) const;
bool operator>(const Cont& X) const;
bool operator<=(const Cont& X) const;
bool operator>=(const Cont& X) const;
```

在此，T0是满足上述需求的一个未指明的类型。

口 queue::back

```
value_type& back();
const value_type& back() const;
```

该成员函数返回一个引用，指向被控序列中最后那个元素，此时被控序列不能为空。

口 queue::container_type

```
typedef Cont container_type;
```

该类型是模板参数Cont的同义词。

口 queue::empty

```
bool empty() const;
```

当被控序列为空时，该成员函数返回true。

口 queue::front

```
value_type& front();
const value_type& front() const;
```

该成员函数返回一个引用，指向被控序列中的第一个元素，此时被控序列不能为空。

口 queue::pop

```
void pop();
```

该成员函数删除被控序列的第一个元素，在执行pop前序列不能为空。

口 queue::push

```
void push(const T& x);
```

该成员函数往被控序列末端插入一个值为x的元素。

口 queue::queue

```
queue();
explicit queue(const container_type& cont);
```

第一个构造函数将被存储的对象初始化为c()，以此来指定一个空的初始被控序列。第二个构造函数将存储的对象初始化为c(cont)，并把被控序列初始化为cont所控制序列的一个拷贝。

口 queue::size

```
size_type size() const;
```

该成员函数返回被控序列的长度。

口 queue::size_type

```
typedef typename Cont::size_type size_type;
```

该类型是Cont::size_type的同义词。

▪ queue::value_type
 typedef typename Cont::value_type value_type;
 该类型是Cont::value_type的同义词。

使用 <queue>

<queue> 如果想在程序中使用模板类 queue 或者 priority_queue, 请把头文件<queue>包含到程序中。

queue 我们也可以特化模板类 queue 以在类型为 deque<T>的对象中存储类型为 T 的元素, 只需这样写一条类型定义语句:

```
typedef queue<T, deque<T, allocator<T> > > Mycont;
```

通过使用默认的模板参数, 可以省略第二个模板参数。

对容器的要求 作为一种可能的方案, 可以用 list<T>来替换第二个参数。(然而, 不能用 vector<T>来替换它。) 也可以指定第二个参数为我们自己设计的容器。确切地说, 容器 Cont 至少应该提供如下的外部接口:

- value_type
- size_type
- empty
- size

• 它必须定义常量的成员函数empty以及size, 并且它们具有通常的意义。

• back

front 它必须以常量和非常量两种形式定义成员函数back和front, 并且它们具有通常的意义。

push_back

pop_front 它必须定义成员函数push_back和pop_front, 并且它们具有通常的意义。

• pop

push 它必须以在类型为Cont的对象上施加完全排序的方式定义那六种比较操作符(如operator==(const Cont&, const Cont&))。

模板类 queue 并不支持所有我们在第 9 章中所给出的、对于容器来说是常见的一些操作。(参见从第 9 章的“使用容器”一节开始的讨论。) 它只是有意提供一系列受限制的操作。下面的描述中都假设 cont 是类 queue<T, C>的一个对象。

构造函数 为了构造类 queue<T, C>的一个对象, 我们可以这么写:

- queue(), 声明一个空队列。
- queue(cont), 声明一个队列, 它的初始内容复制自cont。

push 为了将一个值为 v 的新元素接到队列的末端, 请调用 cont.push(v)。

pop 为了弹出队列中的第一个元素, 请调用 cont.pop()。如果队列为空, 结果就是未定义的。注意, cont.pop()是一个 void 函数——它并不会返回被弹出元素的值。

front 为了存取队列中的第一个(最早被压入队列的)元素, 请调用 cont.front()。为了存取队列中末端(最后被压入队列)那个元素, 请调用

	cont.back()。不管是压入还是弹出，如果队列为空，结果都是未定义的。注意，这两个函数返回的都是引用——如果队列不是一个常量对象，我们就可以改变存储在队列前端或后端的元素的值，如写一个这样的赋值表达式：cont.front() = 0。
empty size	为了测试队列是否为空，请调用 cont.empty()。为了获得队列中元素的个数，请调用 cont.size()。
比较操作	通过常见的那六个操作符，可以把队列 cont 和另一个队列 cont2 做全面的比较，如：cont == cont2 或者 cont >= cont2。它们的返回值与比较它们内部存储的容器对象得到的返回值一样。
c	最后，如果定义了一个以 queue<T, C> 为公共基类的类，那么就可以存取它里面存储的容器对象。这个对象就是保护型的成员 c。
priority_ queue	我们也可以特化模板类 priority_queue 以在类型为 vector<T> 的对象中存储类型为 T 的元素以及类型为 less<T> 的比较规则，只需这样写一条类型定义语句：
	<pre>typedef priority_queue<T, vector<T>, allocator<T>, less<T> > > Mycont;</pre>
less	通过使用默认的模板参数，可以省略第二个和第三个模板参数。
对容器的 要求	如果省略了第三个参数，那么它的默认值就是一个给定的函数对象 less<T>。在存储的容器中的被控序列就由它来排序。也就是说，至少对于数字类型来说，拥有“最高优先级”的元素也就是拥有最大值的那个元素。
value_type size_type iterator	同样可以用 deque<T> 来替换第二个参数（然而，如果换成 list<T> 就不行）。或者，可以为第二个模板参数指定一个我们自己设计的容器类型。确切地说，容器 Cont 至少应该提供如下的外部接口：
empty size begin end front insert push_back pop_back	<ul style="list-style-type: none"> • 它必须定义成员类型 value_type（如 T）和 size_type（通常是 size_t）。 • 它必须定义成员类型 iterator（它是一个随机存取迭代器）。 • 它必须定义默认构造函数 Cont() 以及模板构造函数 Cont(first, last) —— 我们用 (first, last) 所指定的序列来初始化它的被控序列。 • 它必须定义常量成员函数 empty 和 size，并且它们具有通常的意义。 • 它必须定义成员函数 begin 以及 end，并且它们具有通常的意义。 • 它必须定义成员函数 front，并且它具有通常的意义。 • 它必须定义模板成员函数 insert(it, first, last) 用以将 (first, last) 指定的序列插入到 it 前面。 • 它必须定义成员函数 push_back 和 pop_back，并且它们具有通常的意义。

上面给出的接口非常复杂。在C++标准化进程中，在priority_queue加入标准时导入了大量的额外要求（而它们并没有出现在queue和stack中），这些新加的要求并没有考虑到它们对基础容器要求的影响。

模板类priority_queue并不支持所有我们在第9章中所给出的、对于容器来说是常见的一些操作。（参见从第9章的“使用容器”一节开始的讨论。）它只是有意提供一系列受限制的操作。下面的描述都假设cont是类priority_queue<T, C, Pred>的一个对象。

构造函数

为了构造类priority_queue<T, C, Pred>的一个对象，我们可以这么写：

- priority_queue(), 声明一个按Pred()排序的空优先队列；
- priority_queue(pr), 声明一个按函数对象pr排序的空优先队列；
- priority_queue(pr, cont), 声明一个按pr排序的优先队列，它的初始内容是从cont中复制过来并排序后的内容；
- priority_queue(first, last), 声明一个按Pred()排序的优先队列，它的初始内容是从[first, last)所指定的序列中复制过来并排序后的内容；
- priority_queue(first, last, pr), 声明一个按pr排序的优先队列，它的初始内容是从[first, last)所指定的序列中复制过来并排序后的内容；
- priority_queue(first, last, cont, pr), 声明一个按pr排序的优先队列，它的初始内容复制自cont加上由[first, last)指定的序列，并且也是排序后的内容。

push

为了向优先队列中压入一个值为v的新元素，请调用cont.push(v)。

pop

为了从优先队列中弹出它顶端（优先级最高）的元素，请调用cont.pop()。如果该优先队列为空，则它的结果就是未定义的。注意，cont.pop()是一个void函数——它并不会返回被弹出元素的值。

top

为了存取优先队列中顶端（它的优先级最高，并且也是下一个被弹出）的元素，请调用cont.top()。如果该优先队列为空，则它的结果就是未定义的。注意，该成员函数返回的是一个常量的引用——我们不能改变被存储的值。

empty

size

为了测试优先队列是否为空，请调用cont.empty()。为了获得优先队列中元素的个数，请调用cont.size()。

比较操作

虽然队列和栈可以进行比较，但我们不可能比较两个优先队列。

c

comp

最后，如果定义了一个以priority_queue<T, C, Pred>为公共基类的类，那么我们就可以存取它里面存储的容器对象以及函数对象。容器对象就是保护型的成员c，而用来给优先队列排序的函数对象是comp。

实现 <queue>

程序清单16-1列出了文件queue。正如我们可以看到的一样，queue和priority_queue的所有操作都是通过调用存储的容器对象中的相应成员函数来实现的。确实，在这个方面queue和前一章描述的stack非常相似。

```
程序清单 16-1: // queue standard header
queue
#ifndef QUEUE_
#define QUEUE_
#include <algorithm>
#include <deque>
#include <functional>
#include <vector>
namespace std {
    // TEMPLATE CLASS queue
    template<class T, class C = deque<T> >
        class queue {
public:
    typedef C container_type;
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    queue()
        : c() {}
    explicit queue(const C& Cont)
        : c(Cont) {}
    bool empty() const
        {return (c.empty()); }
    size_type size() const
        {return (c.size()); }
    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()); }
    void push(const value_type& X)
        {c.push_back(X); }
    void pop()
        {c.pop_front(); }
    bool Eq(const queue<T, C>& X) const
        {return (c == X.c); }
```

```
        bool Lt(const queue<T, C>& X) const
            {return (c < X.c); }
protected:
    C c;
};

// queue TEMPLATE FUNCTIONS
template<class T, class C> inline
    bool operator==(const queue<T, C>& X,
                      const queue<T, C>& Y)
        {return (X.Eq(Y)); }
template<class T, class C> inline
    bool operator!=(const queue<T, C>& X,
                      const queue<T, C>& Y)
        {return (!(X == Y)); }
template<class T, class C> inline
    bool operator<(const queue<T, C>& X,
                     const queue<T, C>& Y)
        {return (X.Lt(Y)); }
template<class T, class C> inline
    bool operator>(const queue<T, C>& X,
                     const queue<T, C>& Y)
        {return (Y < X); }
template<class T, class C> inline
    bool operator<=(const queue<T, C>& X,
                     const queue<T, C>& Y)
        {return (!(Y < X)); }
template<class T, class C> inline
    bool operator>=(const queue<T, C>& X,
                     const queue<T, C>& Y)
        {return (!(X < Y)); }

// TEMPLATE CLASS priority_queue
template<class T, class C = vector<T>,
         class Pr = less<typename C::value_type> >
class priority_queue {
public:
    typedef C container_type;
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    priority_queue()
        : c(), comp() {}
    explicit priority_queue(const Pr& X)
        : c(), comp(X) {}
    priority_queue(const Pr& X, const C& Cont)
        : c(Cont), comp(X)
```

```

        {make_heap(c.begin(), c.end(), comp); }
template<class It>
priority_queue(It F, It L)
: c(F, L), comp()
{make_heap(c.begin(), c.end(), comp); }
template<class It>
priority_queue(It F, It L, const Pr& X)
: c(F, L), comp(X)
{make_heap(c.begin(), c.end(), comp); }
template<class It>
priority_queue(It F, It L, const Pr& X,
              const C& Cont)
: c(Cont), comp(X)
{c.insert(c.end(), F, L);
 make_heap(c.begin(), c.end(), comp); }
bool empty() const
{return (c.empty()); }
size_type size() const
{return (c.size()); }
const value_type& top() const
{return (c.front()); }
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(); }
protected:
C c;
Pr comp;
};
} /* namespace std */
#endif /* QUEUE_ */

```

测试 <queue>

tqueue.c

程序清单16-2列出了文件tqueue.c。它很简单，因为头文件本身就很小并且还没有很多的内容需要被测试。实际上，它测试模板类queue以及priority_queue的方式和前一章中tstack.c测试stack的方法一样。如果测试一切顺利的话，测试程序将打印出：

SUCCESS testing <queue>

然后正常退出。

```
程序清单 16-2: // test <queue>
tqueue.c      #include <assert.h>
               #include <iostream>
               #include <deque>
               #include <functional>
               #include <list>
               #include <queue>
               #include <vector>
using namespace std;

               // TEST queue
void test_queue()
{
    {typedef allocator<char> Myal;
     typedef deque<char, Myal> Myimpl;
     typedef queue<char, Myimpl> Mycont;
     typedef list<char, Myal> Myimpl2;
     typedef queue<char, Myimpl2> Mycont2;
     Mycont::container_type *p_ccnt = (Myimpl *)0;
     Mycont::value_type *p_val = (char *)0;
     Mycont::size_type *p_size = (size_t *)0;

     Mycont v0(Myimpl(3, 'x')), v0a;
     Mycont2 v1;
     assert(v0.size() == 3 && v0.front() == 'x');
     assert(v0a.empty());
     v0 = v0a;
     v0.push('a');
     assert(v0.size() == 1 && v0.front() == 'a'
           && v0.back() == 'a');
     v0.push('c');
     assert(v0.size() == 2 && v0.front() == 'a'
           && v0.back() == 'c');
     v0.push('b');
     assert(v0.size() == 3 && v0.front() == 'a'
           && v0.back() == 'b');
     assert(v0 == v0 && v0a < v0);
     assert(v0 != v0a && v0 > v0a);
     assert(v0a <= v0 && v0 >= v0a);
     v0.pop();
     assert(v0.front() == 'c');
     v0.pop();
     assert(v0.front() == 'b');
     v0.pop();
     assert(v0.empty()); }
```

```
// TEST priority_queue
void test_priority_queue()
{
    typedef allocator<char> Myal;
    typedef less<char> Mypred;
    typedef vector<char, Myal> Myimpl;
    typedef priority_queue<char, Myimpl, Mypred> Mycont;
    typedef deque<char, Myal> Myimpl2;
    typedef priority_queue<char, Myimpl2, Mypred>
        Mycont2;
    Mycont::container_type *p_cont = (Myimpl *)0;
    Mycont::value_type *p_val = (char *)0;
    Mycont::size_type *p_size = (size_t *)0;

    Mypred pr;
    char carr[] = "acb";
    Mycont v0(pr, Myimpl(3, 'x')), v0a(pr), v0b;
    Mycont2 v1;
    const Mycont v2(carr, carr+3), v2a(carr,carr+3,pr),
        v2b(carr, carr + 3, pr, Myimpl(3, 'x'));
    assert(v0.size() == 3 && v0.top() == 'x');
    assert(v0a.empty());
    v0 = v0a;
    assert(v2.size() == 3 && v2.top() == 'c');
    assert(v2a.size() == 3 && v2a.top() == 'c');
    assert(v2b.size() == 6 && v2b.top() == 'x');
    v0.push('a');
    assert(v0.size() == 1 && v0.top() == 'a');
    v0.push('c');
    assert(v0.size() == 2 && v0.top() == 'c');
    v0.push('b');
    assert(v0.size() == 3 && v0.top() == 'c');
    v0.pop();
    assert(v0.top() == 'b');
    v0.pop();
    assert(v0.top() == 'a');
    v0.pop();
    assert(v0.empty()); }

// TEST <queue>
int main()
{
    test_queue();
    test_priority_queue();
    cout << "SUCCESS testing <queue>" << endl;
    return (0); }
```

习题

- 习题16-1 为什么不能在模板类queue中使用模板类vector？为什么不能在模板类priority_queue中使用模板类list？
- 习题16-2 为什么不能在模板类queue中使用模板类set？
- 习题16-3 要想成为队列的基础表示，容器所需要满足的绝对最小要求是什么？如果换成是优先级队列呢？
- 习题16-4 [较难] 改变模板类queue的实现，使得它也可以和set或multiset这样的容器搭配使用。此时是否还需要使用priority_queue？
- 习题16-5 [特难] 改变模板类priority_queue的实现，使得它既可以作为队列使用，又可以作为优先队列使用。特别是，增加成员函数back和pop_back，使得它们在与push搭配使用时可以提供FIFO这样的行为，并且仍然保留top和pop用来存取和删除优先级最高的元素。

附录 A 接口

STL的头文件非常独立。由于所有的模板定义都限制在头文件中，所以它们不需要从任何C++源文件获得支持。它们也可以用来写出高移植性的标准C++程序。实际上，STL的头文件只直接依赖于标准C++库中的其他4个头文件：

<cstdint>

- <cstdint>中定义了类型ptrdiff_t和size_t。
- <iostream>中定义了几个模板的前向引用（forward reference）。
- <new>中声明了operator delete和operator new的几个版本。
- <stdexcept>中定义了两个异常类。

当然，这些头文件也会依赖于标准C++库中的其他头文件，但对STL的直接接口已经足够窄了。

cstdint

程序清单A-1列出了文件cstdint中的相关部分。C++标准引入了这个头文件，使得原来在标准C头文件<stddef.h>中的定义现在包含在名字空间std中。STL中只使用了<cstdint>中的类型定义ptrdiff_t和size_t。

程序清单A-1:

```
// cstdint standard header (partial)
namespace std {
    // TYPE DEFINITIONS
    typedef int ptrdiff_t; // or another signed integer type
    typedef unsigned int size_t; // or another unsigned integer type
} /* namespace std */
```

iosfwd

程序清单A-2列出了文件iosfwd中的相关部分。C++标准导入这个头文件是为了给其他头文件中所定义的模板和类提供前向引用（即不完全类型的声明）。STL中只对如下4个模板迭代器指向的模板声明感兴趣：istream_iterator、ostream_iterator、istreambuf_iterator和ostreambuf_iterator。

程序清单A-2:

```
// iosfwd standard header (partial)
namespace std {
    // FORWARD REFERENCES
    template<class E>
        class char_traits;
    template<class E, class Tr>
        class basic_istream;
    template<class E, class Tr>
        class basic_ostream;
    template<class E, class Tr>
```

- ```
 class basic_streambuf;
 } /* namespace std */

char_traits • char_traits 描述了在输入流（或输出流）中元素（或通常意义的字符）的属性。
basic_istream • basic_istream 描述的对象控制从输入流中提取元素的操作。一个常见的例子就是标准的输入流 cin。
basic_ostream • basic_ostream 描述的对象控制向输出流中插入元素的操作。一个常见的例子就是标准的输出流 cout。
basic_streambuf • basic_streambuf 描述的对象实现了对输入流（或输出流）的实际缓冲。
new 程序清单 A-3 列出了文件 new 中的相关部分。C++ 标准引入该头文件作为传统的头文件 <new.h> 的后续。为了显式调用 operator new 或 operator delete，必须在程序中包含 <new>。（相反地，如果只是简单地写一些 new 和 delete 的表达式，就不需要包含这个头文件。）模板类 allocator 以及模板函数 get_temporary_buffer 会直接调用这些操作符，以使得存储空间的分配及释放从和对象的构造及销毁的耦合中解脱出来。
```

**程序清单 A-3:**

```
// new standard header (partial)
namespace std {
 // STRUCT nothrow_t
 struct nothrow_t {};
 extern const nothrow_t nothrow; // = nothrow_t()
} /* namespace std */
// OPERATOR new
void *operator new(size_t);
void *operator new(size_t, const std::nothrow_t&) throw();
inline void *operator new(size_t, void *P); // just return P
// OPERATOR delete
void operator delete(void *);
```

**nothrow\_t** 结构体 **nothrow\_t** 是标准化委员会的一个发明。它只用于辨别 **operator new** 的一个版本，在该版本中，如果不能从系统中分配到足够的存储空间，它将返回一个空指针。相反，传统的 **operator new** 版本如今已不再返回空指针了：如果不能分配到足够的存储空间，它就会向外抛出一个异常。至于常数 **nothrow** 只是为了方便而产生的，我们可以用 **new(nothrow) X** 来产生一个类型为 **X** 的对象，而不是用 **new(nothrow\_t()) X**。相对来说，后者比较啰嗦且容易产生不必要的附加代码。

**new** STL 使用了三种形式的 **operator new**：
 • 如果请求的存储分配必须成功或是抛出异常（不成功），使用 **operator new(N)**。
 • 如果请求的存储分配可以失败并且返回一个空指针，使用 **operator**

`new(N, nothrow)`。

- 为了支持定位放置 `new` 的语法，使用 `operator new(N, P)`。

**定位放置**

**new**

上面给出的最后那种 `new` 表达式使得我们可以在已经分配的空间上面构造出另外一个对象，如：

`new(P) T()`

它将会在先前所分配好的未构造的存储空间上<sup>①</sup>（以 `P` 指明该空间的起始位置）调用类 `T` 的默认构造函数，以构造出一个类型为 `T` 的对象。

**delete**

通过调用 `operator delete(P)` 来释放先前由 `operator new` 所分配的存储空间。如果在该空间上已经有对象被构造出来，请先显式地将它销毁。

**stdexcept**

程序清单 A-4 列出了文件 `stdexcept` 中的相关部分。C++ 标准导入这个头文件并在它里面定义了几个异常类。STL 类只显式地抛出以下两个异常类的对象：

**程序清单 A-4:**

```
// stdexcept standard header (partial)
#include <exception>
namespace std {
 // CLASS logic_error
 class logic_error : public exception {
public:
 explicit logic_error(const string& S);
};

 // CLASS length_error
 class length_error : public logic_error {
public:
 explicit length_error(const string& S);
};

 // CLASS out_of_range
 class out_of_range : public logic_error {
public:
 explicit out_of_range(const string& S);
};
} /* namespace std */
```

**length\_error**

- 为了向外报告试图把容器中的被控序列变得太长的企图，STL 会抛出类型为 `length_error` 的异常。

**out\_of\_range**

- 为了向外报告试图在随机存取容器中存取一个有效区间外的元素的企图，STL 会抛出类型为 `out_of_range` 的异常。

<sup>①</sup> 实际上该空间也可以是已经构造过的空间，但此时调用定位放置 `new` 要小心从事，否则容易产生内存泄露。——译者注

**logic\_error** 上面这两个类都是派生自**logic\_error**。当出现的错误可能是由不正确的程序逻辑引起时，就会向外抛出类型为该基类的异常。

**exception** 实际上，**logic\_error**也是由类**exception**所派生而来的，**exception**同时也是标准C++库所抛出的一切异常的基类。很显然，**exception**应该在头文件**<exception>**中定义。所有的这些异常类都定义了一个带有错误消息的构造函数，该错误消息是一个类型为**string**的对象。一旦包含头文件**<string>**，也就获得了模板类**basic\_string**和**char\_traits**的定义。

在这个头文件中，它还同样定义了**string**，它是**basic\_string<char, char\_traits<char>>**的同义词。

如此等等。如果试图把STL头文件的所有依赖关系都列出来的话，我们可能最终会把整个标准C++库的大部分都描述一遍。此处的展示出于有限的目的。它只是展示了STL所依靠金字塔部分的一个小小的顶端。

# 附录 B 术语

本附录列出了在本书中所出现过的有着特殊意义的术语。如果怀疑某个术语的实际含意与最初所想的不一样的话，你可以在此查阅它们的实际意义。

## A

### **access (存取)**

to obtain the value stored in an object or to store a new value in the object

获得存储在对象中的值，或者向对象中存储新的值。

### **allocated storage (已分配存储空间)**

objects whose storage is obtained during program execution

在程序执行期间获得存储空间的对象。

### **allocator (分配器)**

an object that encapsulates strategies for allocating and freeing storage for objects of some type **T**

封装了为类型为T的对象分配及释放存储空间的策略的对象。

### **Amendment 1 (修正版 1)**

the first amendment to the ISO C Standard

ISO C标准的第一次修正版。

### **amortized (分摊)**

spread over a large number of operations, as when an operation takes "amortized constant time" (any occasions that take extratime are sufficiently rare that the overall time complexity remains constant)

将成本扩散到大量的操作中去，如我们说一次操作占用了“分摊的常数时间”(占用额外时间的情况非常罕见，因此总的时间复杂度能够保持为常数)。

### **ANSI (美国国家标准化委员会)**

American National Standards Institute, the organization authorized to formulate computer-related standards in the U.S.

美国国家标准化委员会，它是制订美国计算机相关标准的权威组织。

### **argument (参数; 实参数; 实参)**

an expression that provides the initial value for one of the parameters in a function call

在函数调用时向函数的形式参数(或形参)提供初始值的表达式。

### **arithmetic type (算术类型)**

an integer or floating-point type

整型或浮点型。

**array type (数组类型)**

an object type consisting of a prespecified repetition of an object element

一种对象类型，由一系列重复的预先指定的对象元素组成。

**assign (赋值)**

to store a value in an object

向对象中存储一个值。

**assignment operator (赋值操作符)**

an operator that stores a value in an object, such as **operator**

向对象存储值的操作符，如**operator=**。

**B****base class (基类)**

a class from which another class is derived by inheriting the properties of the base class

派生类从中继承其属性的类。

**beginning (起始)**

the first element of a sequence, or the end of an empty sequence

序列中的第一个元素，或者是空序列的末端。

**benign redefinition (良性重定义)**

a macro definition that defines an existing macro to have the same sequence of tokens spelled the same way and with white space between the same pairs of tokens, or a type definition that defines an existing type to be a synonym for its earlier definition

宏的良性重定义是：新定义的宏和已有的宏具有相同的记号序列（它们的拼写也相同）以及相同的记号对之间的空白；类型的良性重定义是：它定义的类型是先前所定义的类型的同义词。

**balanced binary tree (平衡二叉树)**

a binary tree all of whose terminal nodes (leaves) have essentially the same minimum path length to the root node

一个二叉树，它所有的终端节点（即叶子节点）到达根节点的最短路径从本质上来说都是相同的。

**bidirectional iterator (双向迭代器)**

a category of iterators that can be incremented or decremented

既可以递增，也可以递减的迭代器种类。

**binary tree (二叉树)**

a sequence whose elements are stored as nodes linked to a parent and two child nodes (hence a sequence that also supports bidirectional access to neighboring elements in constant time)

所有元素都以链接到一个父节点和两个子节点的节点的方式存储的序列（因此，该序列还支持在常数时间内对其邻居元素进行双向存取）。

**block (语句块)**

a group of statements in a C++ function enclosed in braces  
在C++函数中由{}括起来的一组语句。

**boolean[布尔(型)：布尔值]**

an expression (of type **bool** in C++) with two meaningful values, true and false (represented by the keywords **true** and **false** in C++)  
只能有两个有意义的值的表达式（在C++中它的类型为bool），这两个值分别为“真”和“假”（在C++中用关键字true和false表示它们）。

**buffer (缓冲区)**

an array object used as a convenient work area or for temporary storage, as between a program and a file  
一个数组对象，它主要用来在程序和文件之间作为方便的工作区域或临时存储空间。

**C****C header (C头文件)**

one of the headers in the Standard C++ library inherited from the Standard C library, having a name ending in .h  
标准C++库中的那些继承自标准C库的头文件，其文件名带有后缀.h。

**C Standard (C标准)**

a description of the C programming language adopted by ANSI and ISO to minimize variations in C implementations and programs  
ANSI和ISO所采纳的C编程语言的描述，它可以使C实现和C程序的不同最小化。

**C++ header (C++头文件)**

one of the headers unique to the Standard C++ library, having a name not ending in .h  
标准C++库中所独有的头文件，其文件名中不包含后缀.h。

**C++ Standard (C++标准)**

a description of the C++ programming language adopted by ANSI and ISO to minimize variations in C++ implementations and programs  
ANSI和ISO所采纳的C++编程语言的描述，它可以使C++实现和C++程序的不同最小化。

**catch clause (catch子句)**

a sequence of tokens that specifies how to handle an exception thrown within an associated try block  
指定如何处理相应的try语句块所抛出的异常的记号序列。

**character (字符型)**

an object type in C++ that occupies one byte of storage and that can represent all the codes in the basic C++ character set  
在C++中的一种对象类型，占用一个字节的存储空间并且可以表示基本C++字符集中的所有编码。

**character constant (字符常量)**

a token in a C++ program, such as 'a', whose integer value is the code for a character in the execution character set

C++程序中的记号，如'a'（它的整型值是执行字符集中该字符的编码）。

**class (类)**

an object type consisting of a sequence of function, object, and type members of different types

一种由一系列不同类型的函数、对象以及类型成员组成的对象类型。

**code (代码)**

colloquial term for programming language text or the executable binary produced from that text

代指编程语言文本或者从该文本产生的可执行二进制代码的术语。

**compiler (编译器)**

a translator that produces an executable file

用于产生可执行文件的翻译器。

**computer architecture (计算机体系结构)**

a class of computers that can all execute a common executable-file format

可以执行通用可执行文件格式的一类计算机。

**constant type (常数类型)**

the type of an object that you cannot store into (it is read-only) once it is initialized because it has the **const** type qualifier

一种对象的类型，它具有**const**类型限定符，因此一旦对该类型的对象进行了初始化，就无法向该对象中存储值（即该对象是只读的）。

**constructor (构造函数)**

a member function that constructs an object, typically by constructing its subobjects and storing initial values in its scalar member objects

一个成员函数，它通常通过构造该对象的子对象并向它的标量成员对象存储初始值来构造一个对象。

**container (容器)**

a class that supplies an organizing structure for a sequence of elements of some contained class

一个类，它为被包含的类的元素序列提供组织结构。

**conversion, type (类型转换)**

altering the representation of a value of one type (as necessary) to make it a valid representation of a value of another type

（在必要的条件下）改变某种类型的值的表示，使它成为另一种类型的的有效表示。

**copy constructor (复制构造函数)**

a constructor that can be called with a single argument that is a **const** reference to another object of the same class,

whose stored value provides the initial value of the object being constructed

一个构造函数，它可以用一个参数调用且该参数为指向同一个类的另一个对象的常量引用，这个对象存储的值为构造中的对象提供初始值。

## D

### **declaration (声明)**

a sequence of tokens in a C++ program that gives meaning to a name, allocates storage for an object, defines the initial content of an object or the behavior of a function, and/or specifies a type

C++程序中的记号序列，它可以为名字提供意义、为对象分配存储空间、定义对象的初始内容或函数的行为，以及指定一个类型。

### **default (默认的)**

the choice made when a choice is required and none is specified

当选择是必需的且没有指定的选择时做出的选择。

### **definition (定义)**

a declaration that, among other things, allocates storage for an object, associates a value with a const object, specifies the behavior of a function, or gives a name to a type; also the define directive for a macro

包括下列行为的声明：为对象分配存储空间、将值和const对象关联起来、指定函数行为，或者是为一个类型给出名字；另外，宏的define指令也是定义的一种。

### **delete expression (delete 表达式)**

an expression involving operator delete, which destroys the object, then calls the operator function to free storage for the object

调用operator delete的表达式，它销毁对象，然后调用操作符函数来释放该对象的存储空间。

### **deque (双队列)**

a queue that also supports random access to arbitrary elements in constant time

一个队列，它还支持在常数时间内对其任意元素进行随机存取。

### **derived class (派生类)**

a class that inherits properties from one or more base classes

从一个或多个基类中继承属性的类。

### **destructor (析构函数)**

a member function that destroys an object, typically by freeing any storage or other resources associated with the constructed object

一个成员函数，它通常通过释放已构造对象以及与之相关的资源所占用的存储空间来销毁一个对象。

### **destroy (销毁)**

to call the destructor for an object

对一个对象调用析构函数。

**diagnostic (诊断消息)**

a message emitted by a C++ translator reporting an invalid program

由C++翻译器报告的表示程序有错的消息。

**dynamic storage (动态存储空间)**

objects whose storage is allocated on entry to a block (or function) and freed when the activation of that block terminates, such as function parameters, **auto** declarations, and **register** declarations

存储空间是在进入一个语句块（或者函数）时分配，在该语句块中的活动终止时释放的对象，如函数参数、auto声明以及register声明。

## E

**element (元素)**

one of the repeated components of an array object, or one of the components of a sequence designated by a pair of iterators, or one of the components of a sequence controlled by a container object

数组对象中的一个重复组成部分，或者是由一对迭代器指定的序列中的一个组成部分，或者是由容器对象所控制的序列的一个组成部分。

**encapsulation (封装)**

localizing a design decision so that a change of design results in changes within a small and easily identified region of source text

将设计决定区域化，以使得设计改变时，只需更改源文本中一个小的、容易标识的部分。

**end (结束)**

the position just beyond the last element of a sequence (same as the beginning of an empty sequence)

序列中紧接着其最后一个元素的位置（当序列为空时，它和序列的开始处相同）。

**end-of-file (文件末端)**

the file position just after the last byte in a file  
文件中位于文件的最后一个字节后面的位置。

**end-of-sequence (序列末端)**

the iterator value that signals the end of a sequence, such as end-of-file or just past the end of an array

用于标识序列结束的迭代器值，如文件末端或数组末端（数组中紧接着最后一个元素的位置）。

**equivalence relationship (相等关系)**

a transitive and commutative ordering that guarantees that a value is always "ordered before" itself (**operator==** on integers imposes an equivalence relationship)

一种具有传递性和交换性的次序，它保证一个值总是“排在自身的前面”（操作整型值的**operator==**施加相等关系）。

**equivalent ordering (相等次序)**

a commutative ordering between two values where neither value is ordered before the other

两个值之间存在着一种具有交换性的次序，此时没有一个值会排在另外一个值的前面。

**erase (删除)**

to remove an element from a sequence

从序列中移除元素。

**exception (异常)**

an object specified by a *throw* expression within a *try* block and caught by a *catch* clause that specifies the object type, used to report an abnormal condition

由try语句块中的*throw*表达式指定的对象，它能够被指定对象类型的*catch*子句所捕获，用于报告异常情况。

**exception handler (异常处理器)**

the part of a *catch* clause that responds to a thrown exception

*catch*子句中对抛出的异常进行响应的部分。

**exception specification (异常规范)**

a qualifier appended to a function declaration that specifies the types of exceptions that can propagate out of the function when it executes

函数声明后紧接着的一个限定符，它指定函数在执行时可能抛出的异常的类型。

**executable file (可执行文件)**

a file that the operating system can execute without further translation or interpretation

操作系统可以不借助翻译程序或解释程序就能够执行的文件。

**explicitly specialize (显式特化)**

to write a template class or template function definition by hand that replaces a particular specialization of the template

手工编写模板类或模板函数的定义来替代模板的一个特定特化版本。

**expression (表达式)**

a sequence of tokens in a C++ program that specifies how to compute a value and generate side effects

C++程序中的记号序列，它指定如何计算一个值以及产生副作用的方式。

**extract (提取)**

to obtain the value of the next character in a sequence controlled by a stream, and to point past that character position in the stream

获得由流控制的序列中的下一个字符的值，并指向流中该字符的下一个字符所处的位置。

**extractor (提取器)**

a function that extracts one or more characters from a stream, converts them to the value of some object type, and stores the value in the object

用来从流中提取一个或多个字符的函数，该函数能够将这些字符转换为某种对象类型的值并将它们存储到某个对象中。

**F****field (字段)**

a contiguous group of characters that matches a pattern specified by a scan format conversion specification, and hence by an extractor

一个连续的字符组，它能够匹配由扫描格式转换规范（或提取器）所指定的模式。

**file (文件)**

a contiguous sequence of bytes that has a name, maintained by the environment

拥有一个名字并由环境维护的由字节组成的连续序列。

**floating-point type (浮点类型)**

any of the types float, double, or long double  
类型float、double或long double。

**forward iterator (前向迭代器)**

a category of iterators that can be incremented but not decremented

迭代器的一种，它只可以递增，不可以递减。

**free (释放)**

to release storage allocated for an object during earlier program execution

释放在程序前期运行期间为一个对象分配的存储空间。

**friend (友元)**

a class or function declaration within a class that makes the name space of the class available within the declared class or function

在一个类中声明的类或函数，它使得该类中的名字空间对于这种类或函数来说是可用的。

**function (函数)**

a contiguous group of executable statements that accepts argument values corresponding to its parameters when called from within an expression and (possibly) returns a value for use in that expression

由可执行语句组成的连续的语句组，它可以被调用时从调用它的表达式得到对应于其形参的实参值，并且（可能的话）向调用它的表达式返回一个值。

**function signature (函数签名)**

the name of a function, along with the type information for its parameters, used in overload resolution to determine

**which function to call in an expression**

函数名字连同其形参的类型信息，用于在重载解析时检测哪个函数在表达式中调用。

#### **function type (函数类型)**

a type that describes a function, as opposed to an object type

用于描述一个函数的类型，对应于对象类型。

## H

#### **has-a relationship (has-a关系)**

describes the use of an existing class to declare a member object (which the newly declared class "has" as a member), as opposed to the use of an existing class as a base (which the newly declared class "is" by derivation)

描述用已有的类来声明成员对象的用法（此时，新声明的类就“has”了已有的类），它和使用一个已有的类作为基类的用法（此时，新声明的类通过派生“is”已有的类）是相对的。

#### **header file (头文件)**

a text file that is made part of a translation unit by being named in an #include directive in a C++ source file

一个文本文件，可以在C++源文件中使用#include指令将它包含到程序中，使它成为翻译单元的一部分。

#### **heap (堆)**

a portion of memory that an executable program uses to store objects allocated and freed in arbitrary order, also a weakly ordered sequence whose first element is not ordered before any of the others (the first element is the largest)

可执行程序用来存储以任意规则分配及释放的对象的内存空间；第一个元素不排在其他任意元素之前的弱序序列（即第一个元素是最大的元素）。

## I

#### **identifier (标识符)**

a name

一个名字。

#### **implementation (实现)**

a working version of a specification, such as a programming language

一个规范（如编程语言）的工作版本。

#### **include file (包含文件)**

a text file made part of a translation unit by being named in an #include directive in a C++ source file or another include file

一个文本文件，通过在C++源文件（或其他包含文件）中使用#include指令，它可以成为翻译单元的一部分。

#### **incomplete type (不完整类型)**

a type that describes an object, but supplies incomplete

information for determining its content (such as **void**, an array of unknown size, or a struct or union of unknown content)

一个用于描述对象的类型，不过它并没有提供用于决定对象内容的完整信息（如：void，大小未知的数组，或者拥有未知内容的结构体或联合）。

**inheritance (继承)**

obtaining the member definitions and type equivalence from a base class

从基类中获得成员以及类型的定义。

**input iterator (输入迭代器)**

a category of iterators used only to read elements from a sequence until the input iterator compares equal to an end-of-sequence value (such as `end-of-file`)

迭代器的一个种类，它只可以用从序列中读取元素，直到迭代器等于`end-of-sequence`值（如`end-of-file`）为止。

**input stream (输入流)**

a stream that can be read from

可以从中读取的流。

**insert (插入)**

to add an element to a sequence before a designated point  
新增一个元素到序列中的指定位置前。

**inserter (插入器)**

a function that converts a value of some object type to a sequence of one or more characters and inserts those characters into a stream

将对象的值转化为一系列的字符（一个或多个）并将这些字符插入到流中的函数。

**instantiate (实例化)**

when applied to templates, often used as a synonym for "specialize"

应用于模板时通常是`specialize`（特化）的同义词。

**integer (整数)**

a whole number, possibly negative or zero

一个数字，可以为负数或零。

**integer type (整型)**

an object type that can represent some contiguous range of integers including zero

一种对象类型，它可以表示某些由整数组成的连续区间（可以包括零）。

**invalid (无效的)**

not conforming to the C++ Standard

不遵循C++标准的。

**I/O**

input and output

输入和输出。

**ISO (国际标准化组织)**

International Organization for Standardization [sic],  
the organization charged with developing international  
computer-related standards

国际标准化组织，它是负责制订国际计算机相关标准的组织。

**is-a relationship (is-a 关系)**

describes the use of an existing class as a base (which  
the newly declared class "is" by derivation), as opposed  
to the use of an existing class to declare a member object  
(which the newly declared class "has" as a member)

描述使用一个已有的类作为基类的用法（此时，新声明的类通过派生“is”已有的类），它和以一个已有的类来声明成员对象的用法（此时，新声明的类“has”已有的类）是相对的。

**iterator (迭代器)**

an object used to access elements, of some type T, from  
an ordered collection, behaving much like a pointer to T

用于从一个有序集合中存取类型为T的元素的对象，它的行为和指向T的指针极为相似。

# J

**J16**

the ANSI-authorized committee responsible for C++  
Standard, formerly X3J16

负责C++标准的ANSI授权委员会，最初叫做X3J16。

# L

**length error (长度出错)**

specifying a sequence length that is too large  
为一个序列指定一个超出其控制范围的长度。

**librarian (库管理者)**

a program that maintains libraries of object modules  
用于维护对象模块的程序。

**library (库)**

a collection of object modules that a linker can  
selectively incorporate into an executable program to  
provide definitions for names with external linkage

由对象模块组成的集合，链接器可以从中选取一部分为可执行程序  
提供具有外部链接属性的名字的定义。

**linker (链接器)**

a program that combines object modules to form an  
executable file

用于将不同的对象模块联合起来构成可执行文件的程序。

**list (列表)**

a sequence whose elements are stored as nodes linked  
in both the forward and backward direction (hence a sequence  
that also supports bidirectional access to neighboring  
elements in constant time)

一个序列，其元素以相互链接的节点形式存储，并且链接是双向的

(因此，可以在常数时间内双向存取该系列中的邻近元素)。

**lvalue (左值)**

an expression that designates an object  
指明一个对象的表达式。

## M

**machine (机器)**

colloquial term for a distinct computer architecture  
明确的计算机体系结构的通俗术语。

**macro (宏)**

a name defined by the **#define** directive that specifies  
replacement text for subsequent invocations of the macro  
in the translation unit

由#define指令定义的名字，它为该宏在翻译单元的后续调用指定替换文本。

**macro definition (宏定义)**

the replacement text associated with a macro name  
与宏名相关联的替换文本。

**macro guard (守卫宏)**

a macro name used to ensure that a text sequence is  
incorporated in a translation unit at most once

用来保证在一个翻译单元中一个文本序列最多只出现一次的宏名。

**macro, masking (宏屏蔽)**

a macro definition that masks a declaration of the same  
name earlier in the translation unit

一个宏定义，它屏蔽同一翻译单元中早期出现的同一宏名的声明。

**member (成员)**

an object declaration that specifies one of the  
components of a class, struct, or union declaration

一个对象声明，它指定类、结构体或联合声明中的一个组件。

**modifiable lvalue (可修改的左值)**

an expression that designates an object that you can  
store a new value into (having neither a constant nor an  
array type)

一个表达式，它指定可以存储新值的对象（也就是说该对象既不是常数类型，也不是数组类型）。

**multithread (多线程)**

supporting more than one program execution in a given  
time interval, possibly allowing interactions between the  
separate program executions

在给定的时间间隔中支持多个程序的执行，或者允许在不同的程序执行间进行交互。

## N

**name (名字)**

a token from a large set used to designate a distinct

**entity** – such as a function, macro, type, or member – in a translation unit

从大的集合中挑出来的用来指定翻译单元中明确实体的记号，该实体可以是函数、宏、类型或成员。

#### **name space**

a set of names distinguishable by context within a C++ program

程序中可以由环境区分的名字的集合。

#### **namespace (名字空间)**

a region of source code qualified by a namespace name  
由名字空间名字所限定的源代码区域。

#### **new expression (new 表达式)**

an expression involving **operator new**, which calls the operator function to allocate storage for an object, then constructs the object

包括 **operator new** 的表达式，它调用该操作符函数为对象分配存储空间，然后建造该对象。

#### **null pointer (空指针)**

the value of a pointer object that compares equal to zero, and hence designates no function or object

一个其值等于 0 的指针对象，因此它没有指明任何函数或对象。

## O

#### **object (对象)**

a group of contiguous bytes in memory that can store a value of a given type

内存中的一组连续字节，可以存储给定类型的值。

#### **object module (对象模块)**

the translated form of a translation unit, suitable for linking as part of an executable program

翻译单元被翻译后的形式，适合链接为可执行程序的一部分。

#### **object type (对象类型)**

a type that describes an object, as opposed to a function type

描述对象的类型，与函数类型相对应。

#### **offset (偏移量)**

the relative address of an element within a sequence  
序列中元素的相对地址。

#### **operand (操作数)**

a subexpression in a C++ expression acted on by an operator  
C++ 表达式中由操作符所操作的子表达式。

#### **operating system (操作系统)**

a program that runs other programs, usually masking many variations among computers that share a common architecture

运行其他程序的程序，通常屏蔽共享同一体系结构的计算机之间的不同之处。

**operator (操作符)**

a token in a C++ expression that yields a value of a given type, and possibly produces side effects, given one to three subexpressions as operands

C++表达式中的记号，给定1至3个子表达式作为操作数，它可以产生给定类型的值，还可能导致副作用。

**operator function (操作符函数)**

a function called implicitly by the evaluation of an operator in an expression, such as **operator=**

在计算表达式中的操作符时隐式调用的函数，如**operator=**

**ordering (次序)**

satisfying a predicate between two values **x** and **y**, such as **x < y**

在两个值X和Y之间满足某一谓词，如x<y。

**ordered binary tree (有序二叉树)**

a binary tree ordered by a predicate such that elements in nodes of each left subtree are ordered before the element in the node, and the element in the node is ordered before the elements in the nodes of each right subtree

按某个谓词排序的二叉树，使得节点任一左子树中的节点中的元素都排在该节点中的元素前面，而该节点则排在任一右子树的节点中的元素前。

**ordered sequence (有序序列)**

a sequence ordered by a predicate such that elements earlier in the sequence are ordered before each element

按某个谓词排序的序列，使得：在序列前面出现的元素按谓词排序在其后面元素之前。

**output iterator (输出迭代器)**

a category of iterators used only to append elements to a sequence of unspecified length

迭代器的一种，它只可以用来向一个未指定长度的序列后端添加元素。

**output stream (输出流)**

a stream that can be written to  
可以往其中写入的流。

**overload (重载)**

to provide more than one definition for a name in a given scope

在给定范围内为一个名字提供超过一个的定义。

**overload resolution (重载解析)**

to determine which of two or more overloaded functions to call within an expression

决定在一个表达式中应该调用重载函数（超过一个）中的哪一个。

**override (重写)**

to provide a definition in a derived class that masks a definition in its base class

在派生类中提供一个定义，以屏蔽在其基类中提供的定义。

**P****parameter (参数; 形参)**

an object declared in a function that stores the value of its corresponding argument on a function call

在函数（原型）中声明的对象，它存储的值就是在实际函数调用中外界传递给该函数的相应实际参数。

**parse (解析)**

to determine the syntactic structure of a sequence of tokens

确定一系列记号的语法结构。

**partially specialize (部分特化)**

to write a template class or template function definition that determines some, but not all, of the parameters for another template

编写一个模板类（或模板函数）的定义，它确定另一个模板的一些（不是全部）形式参数。

**pointer (指针)**

an object that can designate a function or object when used as the (lone) operand of **operator\***

作为operator\*的（惟一）操作数使用时可以指明函数或对象的对象。

**pointer type (指针类型)**

an object type that describes a pointer

用描述指针的对象类型。

**PODS**

an object type that describes a "plain old data structure," essentially a class type whose non-static member objects could equally well be declared as a C struct

一个描述“纯粹的、老式的数据结构”的对象类型，本质上是一个类型，它的非静态成员对象可以作为C结构体来声明。

**portability (移植性)**

cheaper to move to another environment than to rewrite for that environment

“移动到另一环境中的代价要小于为该环境重新写一次的代价”这种特性。

**predicate (谓词)**

an expression that yields a Boolean result

产生一个布尔结果的表达式。

**preprocessor (预处理器)**

that portion of a C++ translator that processes text-oriented directives and macro invocations

C++翻译器的一部分，它只处理面向文本的指令以及宏调用。

**priority queue (优先队列)**

a queue that is ordered after each push back, so that each pop back erases the element that orders highest

一个队列，在每次进行push back后都是有序的，因此每次对它进行pop back都可以移除排序最高的元素。

**program (程序)**

a collection of functions and objects that a computer can execute to carry out the semantic intent of a corresponding set of C++ source files

函数及对象的集合，它可以在计算机上执行并完成相应C++源文件集中给定的语义意图。

**program startup (程序起始期)**

the period in the execution of a program just before **main** is called

程序运行时，调用main函数之前的时期。

**program termination (程序终止期)**

the period in the execution of a program just after **main** returns or **exit** is called

程序运行时，main函数返回或是调用exit后的时期。

**pop back**

to erase an element at the end of a sequence

移除序列中的最后一个元素。

**pop front**

to erase an element at the beginning of a sequence

移除序列中的第一个元素。

**push back**

to append an element to a sequence

向序列的末端添加一个元素。

**push front**

to prepend an element to a sequence

向序列的开始处添加一个元素。

**Q****queue (队列)**

a sequence that supports the operations pop back, pop front, push back, and push front in constant time

一个序列，它能在常数时间内完成pop back、pop front、push back以及push front等操作。

**R****raise**

to throw an exception

抛出一个异常。

**random-access iterator (随机存取迭代器)**

a category of iterators that can be incremented, decremented, or altered by an integer offset in constant

time

迭代器的一种，可以在常数时间内递增、递减，以及由整型偏移量对它进行改变。

**read-only (只读)**

containing a stored value that cannot be altered  
包含不可改变的存储值。

**reachable iterator (可到达的迭代器)**

an iterator that serves as end values for a finite sequence (the beginning iterator, incremented a finite number of times, will eventually compare equal to a reachable end iterator)

在一个有限序列中可以作为最终值的迭代器（即一开始的迭代器在经过有限次的递增后，可以等于最终可到达的迭代器）。

**recursion (递归)**

calling a function while an invocation of that function is active

调用一个正在被调用的函数。

**red-black tree (红—黑树)**

an ordered binary tree that permits a bounded degree of imbalance (hence an ordered sequence that supports searches in logarithmic time and also supports bidirectional access to neighboring elements in constant time)

一个有序二叉树，它允许有一定程度的不平衡（因此，可以在对数时间之内完成对它的搜索，也可以在常数时间内双向存取其邻居元素）。

**reference (引用)**

an object that designates a function or another object  
指明函数或其他对象的对象。

**reference type (引用类型)**

an object type that describes a reference  
描述引用的对象类型。

**representation (表示)**

the number of bits used to represent an object type,  
along with the meanings ascribed to various bit patterns

用于表示一个对象类型的位数，连同因不同的位模式而有所不同的意义。

**reserved name (保留名字)**

names available for use only for a restricted purpose  
只能用于有限用途的名字。

**reverse iterator (反转型迭代器)**

iterators that, when incremented, walk backwards through a sequence

在递增时逆向遍历系列的迭代器。

**rvalue (右值)**

an expression that designates a value of some type  
(without necessarily designating an object)

指明某种类型值的表达式（不需要指明某个对象）。

## S

### **secret name (秘密名字)**

a name from the space of names reserved to the implementor,  
such as `_Abc`

在所有的名字中选出来为实现保留的名字，如`_Abc`。

### **semantics (语义)**

the meaning ascribed to valid sequences of tokens in  
a language

一种编程语言的有效记号序列的意义。

### **side effect (副作用)**

a change in the value stored in an object or in the state  
of a file when an expression executes

执行表达式对象存储的值或文件状态的改变。

### **signed integer (有符号整数)**

an integer type that can represent negative as well as  
positive values

一个整数类型，它可以表示负数和正数。

### **source file (源文件)**

a text file that a C++ translator can translate to an  
object module

一个文本文件，C++翻译器可以将它翻译为对象模块。

### **space (空格)**

a character that occupies one print position but displays  
no graphic

一个字符，它占用了一个打印位置但不显示任何图形。

### **specialize (特化)**

to write a template class or template function and supply  
any parameter values to specify a particular instance of  
the template, as in `vector<float>` (often used improperly  
to mean "explicitly specialize")

编写一个模板类（或模板函数），提供所需的所有参数值以指定该  
模板的一个特定实例，如`vector<float>`（经常不适当当地意指“显式特化”）。

### **standard C (标准 C)**

that dialect of the C programming language defined by  
the ANSI/ISO C Standard

由ANSI/ISO C 标准定义的编程语言方言。

### **standard C library (标准 C 库)**

the set of functions, objects, and headers defined by  
the C Standard, usable by any hosted C or C++ program

由C标准定义的函数、对象以及头文件的集合，可以由任意的宿主C  
或C++程序使用。

### **standard C++ (标准 C++)**

that dialect of the C++ programming language defined  
by the ANSI/ISO C++ Standard

由ANSI/ISO C++标准定义的C++编程语言方言。

**standard C++ library (标准 C++ 库)**

the set of functions, objects, templates, and headers defined by the C++ Standard, usable by any hosted C++ program

由C++标准定义的函数、对象、模板以及头文件的集合，可以由任意的宿主C++程序使用。

**standard header (标准头文件)**

one of many headers defined by the C++ Standard  
由C++标准定义的头文件。

**statement (语句)**

an executable component of a function that specifies an action, such as evaluating an expression or altering flow of control

函数中的可执行组件，它指定一个动作，如：计算一个表达式，或是改变控制流程。

**static storage (静态存储)**

objects whose lifetime extends from program startup to program termination, initialized prior to program startup

生存周期延展到程序起始段至程序终止段之间的对象，其初始化要早于程序起始段。

**store (存储)**

to retain a value, or to replace the value stored in an object with a new value

保留对象的原始值，或是用新值替换对象存储的值。

**stream (流)**

an object that maintains the state of a sequence of reads, writes, and file-positioning requests for an open file

为打开的文件维护一系列读、写以及文件定位请求状态的对象。

**strict weak ordering (严格弱序)**

a transitive ordering that guarantees that a value is never ordered before itself and that also defines an equivalent ordering between pairs of values (`operator<` on integers imposes a strict weak ordering, but `operator<=` does not)

具有传递性的排序方式，它保证一个值不可能排在其自身前面，同时还定义了值对之间的相等关系（对整数来说，`operator<`施加严格弱序，但`operator<=`则没有）。

**string (字符串)**

an object of class `basic_string<E>` that controls a sequence of characters (of type `E`)

类`basic_string<E>`的对象，它控制一个字符（类型为E）序列。

**structure type (结构类型)**

an object type consisting of a sequence of function, object, and type members of different types (usually called a "class" in C++)

一个由一系列不同类型的函数、对象以及类型成员组成的对象类型（在C++中通常叫做“类”）。

**synonym (同义词)**

an alternate way of designating a type that is otherwise equivalent to the original type

指明一种类型的可替换方法，以该方法得到的类型以不同的方式等价于原有的类型。

**syntax (语法)**

the grammatical constraints imposed on valid sequences of tokens in a language

在一门语言中，强加于有效记号序列的文法限制。

**T****template (模板)**

a class or function declaration written in terms of one or more type or value parameters for later specialization with specific parameter values

一个类或函数的声明，在编写时，它的一个或多个类型或值参数都被留到后面的特化阶段给出。

**text (文本)**

a sequence of characters nominally suitable for writing to a display device (to be read by people)

名义上适合向显示设备输出（让人们阅读）的字符序列。

**thread of control (控制线程)**

the execution of a program by a single agent  
由一个代理负责的程序执行。

**throw expression (throw 表达式)**

an expression of the form **throw ex** that throws the object **ex** as an exception

具有**throw ex**形式的表达式，它将对象**ex**作为异常抛出。

**token (记号)**

a sequence of characters treated as a single element in a higher-level grammar

在高层文法中视为单一元素的字符序列。

**translation unit (翻译单元)**

a C++ source file plus all the files included by **#include** directives, excluding any source lines skipped by conditional directives

一个C++源文件，加上由**#include**指令包含进来的文件，但不包括由条件指令跳过的源行。

**translator (翻译器)**

a program, such as a compiler, that converts a translation unit to executable form

用来将编译单元转化为可执行格式的程序，如编译器。

**try block (try 语句块)**

a block followed by one or more catch clauses prepared to handle exceptions thrown within the block

后面带有一个或多个**catch**子句的语句块，那些子句准备处理该语句

块中抛出的异常。

**type (类型)**

the attribute of a value that determines its representation and what operations can be performed on it, or the attribute of a function that determines what arguments it expects and what it returns

值的一种属性，它确定值的表示以及在其上可以进行哪些操作；函数的一种属性，它确定函数能够接受哪些实际参数以及返回哪些值。

**type definition (类型定义)**

a declaration that gives a name to a type

为类型给出一个名字的声明。

# U

**union type (联合类型)**

an object type consisting of an alternation of object members, only one of which can be represented at a time

由不同的对象成员组成的对象类型，在一个确定的时间内只能有一种成员对象被表示。

**unsigned integer (无符号整数)**

an integer type that can represent values between zero and some positive upper limit

一个整数类型，它可以表示从0到某个正数上限之间的值。

# V

**variable (变量)**

older term for an object

用来表示对象的早期术语。

**vector (向量)**

a sequence implemented as an array whose length can vary (hence a sequence that also supports random access to arbitrary elements in constant time)

一个实现为数组但长度可变的系列（因此支持在常数时间内随机存取任意元素）。

**virtual member (虚成员)**

a member function whose overriding definition is called even when accessed as an object of its base class

一个成员函数，即使是在作为基类的对象存取时，也可以调用它的重写定义。

**void type (void类型)**

a type that has no representation and no values

一个没有表示以及值的类型。

**volatile type (易变类型)**

a qualified type for objects that may be accessed by more than one thread of control

对象的限定类型，它的对象可以被超过一个的控制线程存取。

## W

### WG21

the ISO-authorized committee responsible for C++ standardization

负责C++标准化进程的ISO授权委员会。

### white space (空白字符)

a sequence of one or more space characters, possibly mixed with other spacing characters such as horizontal tab

一系列空格字符（一个或多个）可能与其他间隔字符（如水平制表符）混合出现。

### wide character (宽字符)

a code value of type `wchar_t` used to represent a very large character set

类型为`wchar_t`的编码值，它用来表示一个非常大的字符集。

### width (宽度)

part of a conversion specification in a format that partially controls the number of characters to be transmitted

有关格式的转化规范中的一部分，它部分地控制将要转换的字符的数量。

### writable (可写的)

can have its value altered, opposite of read-only

其值可以改变的，与只读相对应。

## X

### X3J16

the older name for J16, The ANSI-authorized committee responsible for C++ Standard

J16的旧名字，负责C++标准的ANSI授权委员会。

# 参 考 文 献

下面所列出来的一些书籍和资料要么直接和本书中的内容相关，要么可以作为相关的指导材料。该列表并不是一个和本书主题相关的所有资料的清单。

□ ANS89

- ANSI标准X3.159-1989（纽约：美国国家标准协会，1989）。最初的C标准，由ANSI授权委员会X3J11开发。伴随C标准的基本原理解释了许多加入到其中的决定。

□ Aus99

- Matthew H. Austern，《Generic Programming and the STL》（Reading MA: Addison-Wesley，1999）。由积极增强和扩展STL的人士之一最近撰写的一本书。

□ ISO90

- ISO/IEC 标准 9899:1990（日内瓦：国际准组织，1990）。世界范围的官方C标准。除了格式细节和各部分编号之外，ISO C 标准和ANSI C 标准相同。

□ ISO94

- ISO/IEC标准9899:1990的修正版1（日内瓦：国际准组织，1994）。C标准的第一次（也是唯一的）修正版。它提供了对操作大字符集的潜在支持。

□ ISO98

- ISO/IEC标准14882:1998（日内瓦：国际标准组织，1998）。世界范围的官方C++标准。ISO C++ 标准和ANSI C++ 标准相同。

□ Jos99

- Nicolai M. Josuttis，《The C++ Standard Library, A Tutorial and Reference》（Reading MA: Addison-Wesley，1999）。标准C++库（包括STL）的概述。

□ M&S87

- David R. Musser和Alexander Stepanov，“Ada语言中的泛型算法库”，《1987年 SIGAda国际会议记录》（纽约：计算机协会，1987）。

□ M&S89

- David R. Musser和Alexander Stepanov，《Ada Generic Library》（柏林：Springer Verlag，1989）。

# C++ STL

## 中文版

ANSI/ISO C++ STL (Standard Template Library, 标准模板库) 是一个功能强大的工具集，使用它可最大限度地提高生产效率、软件质量和性能。STL 提供了丰富的容器类和基础算法，它们构成了用于 C++ 开发的一个大型、系统、有效的架构。现在，在这本众人期待已久的书中，创造者们对 C++ STL 进行了权威而深入的解析。本书的每一章都讲述了 STL 的一个组成部分，包括：

- 各组成部分及其用途的详细背景知识
- 相应的 ANSI/ISO C++ 标准的全面回顾
- 使用和实现各部分的实际方法
- 详尽的代码实例
- 实用习题

本书附录列出了接口及术语，最后列出了相关的参考文献。

本书全面阐释了最终版本的 ANSI/ISO STL 标准，确实是富有经验的 C++ 程序员必不可少的宝典。

**P.J. PLAUGER** 是 Dinkumware 有限公司的总裁，该公司负责为符合标准 C 和 C++ 的库及文档提供授权。他身兼《The C/C++ Users Journal》主编编辑和《Embedded Systems Programming》的特邀编辑多年。长期以来 Plauger 在 C 和 C++ 国际化标准进程中一直非常活跃。

**ALEXANDER A. STEPANOV** 现供职于 AT&T，以前是加利福尼亚 Palo Alto 惠普研究实验室的巨型编程项目的负责人。

**MENG LEE** 目前是惠普研究实验室的一名技术人员，在这里她和 Stepanov 开发了最初的 STL。STL 被提交给 ANSI/ISO C++ 标准化委员会，并最终得到认可。

**DAVID R. MUSSER** 是 Rensselaer 工学院计算机科学系的教授，专门研究泛型编程。他与 Stepanov 一起为 STL 打下了基础。

责任编辑：关敬 宋宏

ISBN 7-5083-1058-6



9 787508 310589

ISBN 7-5083-1058-6 / TP · 321

定价：69.00 元