THE EXPERT'S VOICE® IN C++

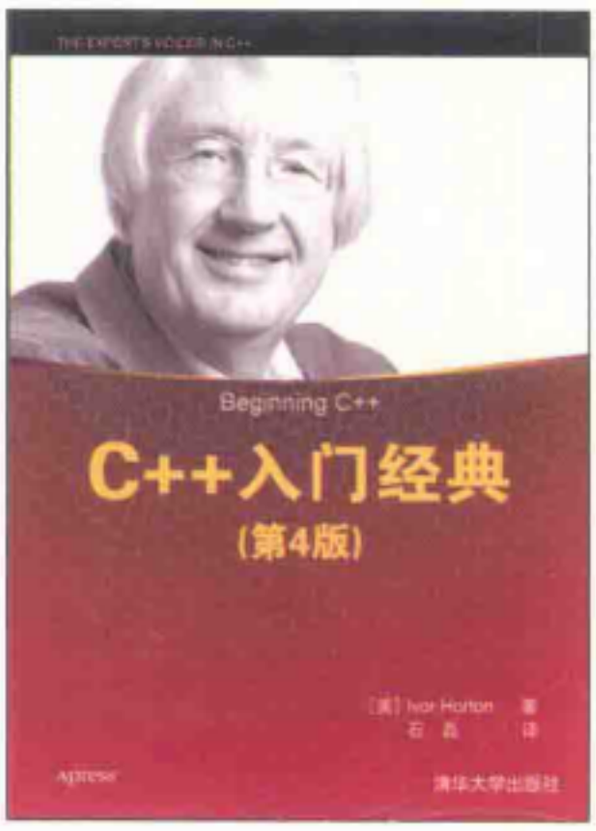
Using the C++ Standard Template Libraries

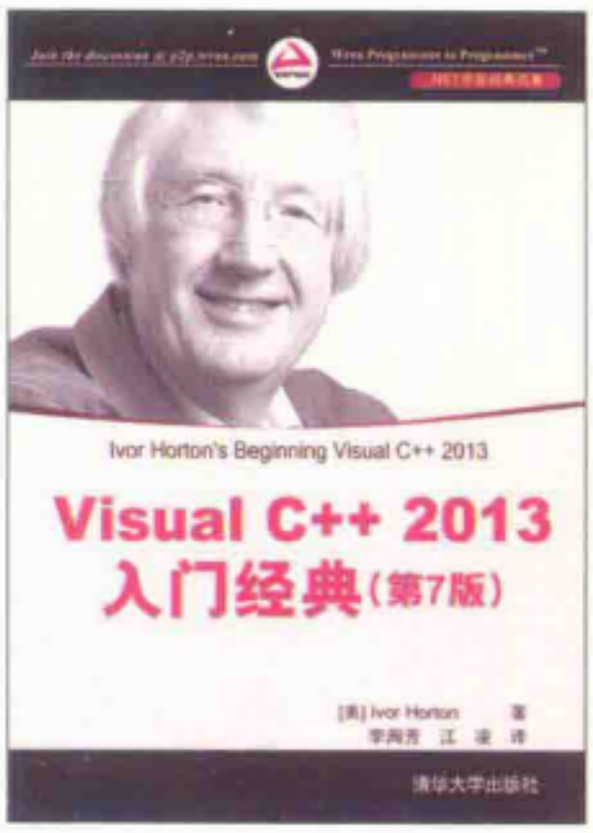
C++标准模板库编程实战

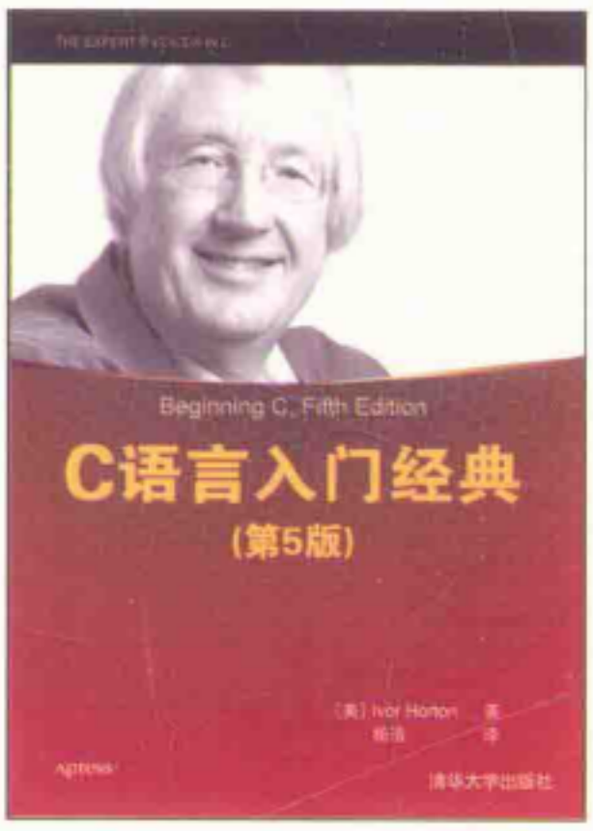
[美] Ivor Horton 著

郭小虎 程聪 译

清华大学出版社







《C++标准模板库编程实战》介绍最新的C++14标准的API、库和扩展，以及如何将它们运用到C++14程序中。在书中，作者Ivor Horton 则阐述了什么是STL，以及如何将它们应用到程序中。我们将学习如何使用容器、迭代器，以及如何定义、创建和应用算法。此外，还将学习函数对象和适配器，以及它们的用法。

阅读完本书之后，你将能够了解如何扩展STL，如何定义自定义类型的C++组件，你还将能够定义既满足C++ STL要求又遵从最常见的设计模式和最佳实践的自定义类型。

标准库是C++标准的一个基本部分，它为C++程序员提供了一套全面而又高效的工具，还提供了一些适用于多种类型程序的可重用组件。

主要内容

- 如何在C++程序中使用STL
- 如何使用容器
- 如何使用迭代器
- 如何定义、生成和运用算法
- 如何使用函数对象
- 如何扩展STL，以及如何定义自定义类型的组件
- 如何使用适配器
- 如何定义自定义类型，使之既满足C++ STL的要求又遵从通用的设计模式和最佳实践

作者简介

Ivor Horton是世界著名计算机图书作家，独立顾问，帮助无数程序员步入编程殿堂。他曾在IBM工作多年，以优异成绩拥有数学学士学位。他的资历包括：使用大多数语言(如在多种机器上使用汇编语言和高级语言)进行编程，实时编程，设计和实现实时闭环工业控制系统。Horton拥有丰富的面向工程师和科学家的编程教学经验(教学内容包括C、C++、Fortran、PL/1、APL等)。同时，他还是机械、加工和电子CAD系统、机械CAM系统和DNC/CNC系统方面的专家。

清华大学出版社数字出版网站
WQBook 文泉书局
www.wqbook.com

Apress®
www.apress.com

源代码下载

ISBN 978-7-302-45580-6

定价：69.80元

C++标准模板库编程实战

[美] Ivor Horton 著
郭小虎 程 聪 译

清华大学出版社
北 京

Ivor Horton
Using the C++ Standard Template Libraries
EISBN: 978-1-4842-0005-6

北京市版权局著作权合同登记号　图字：01-2016-8565

图书在版编目(CIP)数据

C++标准模板库编程实战 / (美)爱弗·霍顿(Ivor Horton) 著；郭小虎，程聪　译. — 北京：清华大学出版社，2017

书名原文：Using the C++ Standard Template Libraries
ISBN 978-7-302-45580-6

Ⅰ. ①C…　Ⅱ. ①爱… ②郭… ③程…　Ⅲ. ①C 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2016)第 281893 号

责任编辑：王　军　李维杰
封面设计：牛艳敏
版式设计：思创景点
责任校对：牛艳敏
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课　件　下　载：http://www.tup.com.cn，010-62794504
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：30　　字　　数：711 千字
版　　次：2017 年 1 月第 1 版　　印　　次：2017 年 1 月第 1 次印刷
印　　数：1～3000
定　　价：69.80 元

产品编号：069421-01

译者序

C++语言是C语言的继承者，它的经典性和地位是其他高级语言不可比拟的。虽然现在随着互联网时代的到来，计算机技术的发展日新月异，各种新语言层出不穷，令人眼花缭乱，但是强大、灵活且高效的C++语言一直在TIBOE榜单位居前列，这说明了它强大的生命力。C++语言可以进行底层驱动的开发，可以开发操作系统、传统桌面程序、大型游戏(3A大作基本都是在C++开发的游戏引擎的基础上开发的)，等等。甚至在新兴的移动开发领域也有其一席之地，它的身影遍及各个领域。C++语言是一门优秀的面向对象入门语言，对于初学者，在学习完C语言之后，就可以通过学习C++来学习面向对象思想。现在主流的高级语言基本都是面向对象语言，在掌握C++之后，学习其他高级语言，譬如C#、Java、Python和Go，会有事半功倍的效果，笔者结合自己的学习经历，深有感触。

学习C++就跳不过C++的STL，STL可以说是C++的精华所在。STL提供的算法和容器等工具大大方便了我们的日常开发，我们不用再重复造轮子，减少了出错的可能性，减少了代码量，增强了代码的复用性。STL也是C++的一大难点，市场上关于STL的书籍浩如烟海，但罕有像本书这样从浅入深、循序渐进介绍STL的书，这一点本书的作者功不可没。本书作者Ivor Horton是世界著名的计算机图书大师，他在IBM公司工作多年，有着丰富的实践经验，其著作帮助很多程序员走进编程殿堂。本书是他的新作，无疑又是一本学习C++语言的经典之作。

学习语言是极其枯燥的，但是学成之后，用它写出一个程序的感觉却是令人无比喜悦，很有成就感。就笔者个人的经历而言，对语言的学习，注重实践，所以对于每章后面的练习最好都先自己独立完成一遍，这样能够帮助你回顾在这一章学到的知识，中间遇到问题，也能引发思考，从而学到知识。

就本书的结构而言，是按照由浅入深、从易到难的顺序讲解，所以建议顺序阅读。一开始介绍序列容器，而后是容器适配器，再是map容器，到set容器等，这种结构很合理。此外，本书还引入了很多C++新标准的知识，比如lambda表达式、右值参数、移动语义、完美转发、auto等，让我们能够学习C++最新的技术标准，在帮我们夯实基础的同时，开拓了我们的眼界。在后面的章节中，也介绍了STL提供的很多高级算法，它们可以按照概

率论中的各种分布生成随机数，能处理复数以及与时间相关的一些操作。当然，STL 提供的库不止于此，但这已经能够展现 STL 强大丰富的一面。

最后，由衷地感激清华大学出版社的编辑们，本书的顺利出版凝聚了他们很多的心血和汗水。

对于这本书，译者本人在翻译的过程中力求忠于原文，尽量让译文再现原书风貌，尽量简洁明了到所翻译图书的原意。但由于译者本人水平有限，难免会出现错误和失误，如果发现本书的任何错误或有任何意见和建议，请不吝指正。本书全部章节由郭小虎、程聪翻译，参与翻译的还有徐浩、黄飞、胡泽靖、徐汉，在此一并表示感谢！

最后，希望读者能够通过本书学会 STL，领略 STL 的优雅与内涵。

译　者

作者简介

Ivor Horton 毕业于数学系，却因向往 IT 工作轻松而收入丰厚，因而涉足 IT 领域。尽管现实情况常常是工作辛苦而收入却相对不高，但他仍坚持从事计算机工作至今。在不同的时期，他从事过的工作包括程序设计、系统设计、顾问以及管理和实现相当复杂的项目。

Ivor 在计算机系统的设计和实现方面，拥有多年的工作经验，这些系统应用于多种行业的工程设计和制造运营。他不仅能运用多种编程语言开发特殊用途的应用程序，而且还为科研人员和工程人员提供教学，以帮助他们完成这类工作，在这些方面他都拥有相当丰富的经验。多年来他一直从事程序设计方面书籍的撰写工作，目前出版的著作有 C、C++和 Java 等方面的教程。目前，他不忙于写书或给他人提供咨询服务时，会钓鱼、旅游和尽情地享受生活。

技术编辑简介

Marc Gregoire 是来自于比利时的一位软件工程师，他以“在计算机科学中的土木工程师”(相当于在工程中的计算机科学理学硕士)学位毕业于比利时的天主教鲁汶大学。次年，他在母校获得了优等学位的人工智能专业硕士。在学习生涯结束后，Marc 为 Ordina Belgium 软件咨询公司工作。作为一名顾问，他服务于西门子和诺基亚西门子网络关键的 2G 和 3G 软件，这些软件运行在电信运营商的 Solaris 系统上。这要求工作在从南美和美国，跨越到欧洲、中东、非洲和亚洲的国际团队中。现在，Marc 为 Nikon Metrology 研发 3D 激光扫描软件。

Marc 主要擅长 C/C++，尤其是微软的 VC++和 MFC 框架。除了 C/C++，Marc 也喜欢 C#，用 PHP 生成网页程序。他的主要兴趣在于 Windows 开发，此外，他还有在 Linux 平台(比如 EIB 的家用自动化软件)上开发全天候运行的 C++程序的经验。

从 2007 年 4 月开始，他每年都因为 Visual C++专长被授予微软 MVP 称号。

Marc 是比利时 C++用户组的创始人，同时也是 CodeGuru 论坛(类似于 Marc G)的活跃成员。他创建了很多免费和共享软件，这些软件通过他的 www.nuonsoft.com 网站发布，并且他在 www.nuonsoft.com/blog/上维护了一个博客。

致　　谢

感谢 Mark Powers 和 Apress 出版社的其他编辑以及产品团队，感谢他们自始至终提供的帮助和支持。特别感谢马克所做的出色的技术审查工作，他的很多评论和建议毫无疑问使这本书变得更加优秀。

前　言

欢迎学习《C++标准模板库编程实战》一书。本教程介绍了由 C++标准库组成的头文件子集中所包含的一些类和函数的模板。这些模板是功能强大、易于使用的泛型编程工具，并使很多不容易实现的任务变得易于实现。它们生成的代码通常比我们自己编写的更加高效和可靠。

通常，笔者不喜欢只解释它们做了些什么，而不详细论述这么做的原因。从前者是很难猜出后者的。因此笔者的目标不仅仅是解释类和函数模板的功能，还会尽可能地展示如何在实际场景中应用它们。这会导致在某些知识点的介绍中包含相当大的代码块，但相信你会觉得它们是值得的。

之前提到的作为本书主题的来自于 C++标准库的头文件的集合，被称作 C++标准库或 STL。在本书中，会用 STL 作为一种方便的缩写来表示包含本书所讨论模板的头文件的集合。当然，并没有 STL 这种东西——C++语言标准并没有提到它，因此正规而言，它并不存在。尽管它并没有被定义，但很多 C++程序员都大致知道 STL 是什么意思。这种叫法由来已久。

贯穿 STL 的泛型编程思想早在 1979 年起源于 Alexander Stepanov——很久之前并没有 C++语言标准。C++的 STL 的第一个实现起源于 Stepanov 和其他在 1989 年前后工作于惠普公司的职员，而且在那时，STL 的实现和 C++编译器所提供的库是互补的。在 20 世纪 90 年代，STL 提供的功能开始被考虑纳入第一个 C++语言标准的提议中，而且 STL 的精髓使它成为公布于 1998 年的第一个 C++语言标准。从那时起，STL 所代表的泛型编程开始被改进和扩展，并且很多不属于 STL 的头文件中开始出现了模板。本书中的所有材料都和编写本书时最新通过的语言标准相关，也就是 C++14。

STL 不是一个准确的概念，并且本书中并没有包含 C++标准库的全部模板。本书只描述和展示了笔者认为 C++程序员应该首先选择理解的标准库中的模板，尤其是那些初次接触 C++的开发者。书中将被深度讨论的主要标准库头文件包括：

用于数据容器：<array>、<vector>、<deque>、<stack>、<queue>、<list>、<forward_list>、<set>、<unordered_set>、<map>、<unordered_map>

用于迭代器：<iterator>
用于算法：<algorithm>
用于随机数和统计：<random>
用于数值处理：<valarray>、<numeric>
用于时间和定时：<ratio>、<chrono>
用于复数：<complex>

来自于其他头文件的模板，比如<pair>、<tuple>、<functional>和<memory>也被加入到本书的不同章节中。数据容器的模板是基础，在很多程序中都会用到它们。迭代器是使用容器时的基本工具，因此它们也被包含了进来。算法是操作保存在容器中的数据的函数模板，也可以将这些强大的工具应用到数组上，在示例中会对此进行描述和展示。书中有一章将解释随机数生成和统计相关的模板，但是它们中有一些是相当专业的。在模拟、建模和游戏程序中，很多都得到了广泛应用。本书还讨论了计算扩展数值数据的模板，以及和时间、定时相关的模板。最后，简短介绍了一个关于用于处理复数的类模板。

使用本书的先决条件

为了理解本书的内容，需要具备一些 C++语言的基本知识。本书是对《C++入门经典(第4版)》一书的补充，所以如果成功读完了那本书，就可以开始阅读这本了。需要知道的基本知识包括：类和函数模板是什么，它们工作的本质是怎样的。笔者在第 1 章包含了这些基本知识的概述，如果之前不习惯使用模板，它们的语法会让人觉得它们比它们本身要复杂得多。一旦开始习惯这些记号，就会发现它们的用法相对容易。STL 中也频繁使用了 lambda 表达式，所以也必须习惯使用它。

我们需要一个兼容 C++14 的编译器。当然，为了编写程序代码，也需要一个合适的文本编辑器。在最近的几年中，C++编译器发展得相当好，尽管它是最近才通过的标准，但已经有几个很不错的编译器在很大程度上遵从了 C++14。至少可以选择 3 个可用的免费编译器：

- GCC 是支持 C++、C、Fortan 和其他语言的 GNU 编译器集合，它支持在本书中用到的所有 C++14 特性，可以从 gcc.gnu.org 下载 GCC。GCC 编译器集合适用于 GNU 和 Linux，但也可以从 www.mingw.org 下载 Microsoft Windows 版。
- ideaone 在线编译器支持 C++14，可以通过 ideaone.com 访问。在编写代码时，对于 C++14，它使用的是 GCC 5.1。ideaone.com 也支持很多其他语言，包括 C、Fortran 和 Java。
- Microsoft Visual Studio 2015 Community Edition 在 Microsoft Windows 操作系统下运行，它支持 C++，也支持几种其他的语言，并配置了一个完整的开发环境。

如何使用本书

对于大部分内容，为了可以顺序阅读，已经对本书的材料进行了组织，所以使用本书的最佳方式是从头阅读到尾。一般情况下，在没解释功能之前，不会使用它们。一旦解释了，无论在什么时候，只要有意义，都会把它插到后续的材料中，这也是为什么推荐按顺序阅读章节的原因。很少有需要理解背后的数学知识的主题，并且在这些实例中，也会介绍数学知识。如果不熟悉数学，可以跳过这些，因为这不会限制对后面内容的理解。

没有人可以只通过看书就学会编程。只有通过写代码才能学会如何使用 STL。强烈建议自己敲一遍所有的代码——而不仅仅只是从下载的文件中复制代码——并编译和执行你所敲入的代码。这有时可能变得很乏味，但令人惊讶的是，只需要敲一些程序语句就可以帮助我们理解代码，尤其是在我们有些琢磨不定时。它也能帮助我们记住东西。如果例子无法运行，请抵制直接回到书中查找原因的诱惑，尝试从自己的代码中查找错误。

在本书的所有章节中，对于很多部分来说，如果包含了适当的头文件，代码段都是可以执行的。如果把它们放到 main()函数中，一般都可以执行它们，并得到输出结果。因此建议为此创建一个程序项目。可以将代码复制到定义为空的 main()中，并为需要的头文件加上#include 指令。为了防止名称冲突，大多数时候需要删除之前的代码。

要让自己的错误成为学习过程的一部分，并且书中的练习为我们提供了这样的机会。我们犯的错误越多，找到和解决的问题就越多，就越能更好地了解使用模板时会犯哪些错误。确保自己可以完成所有能完成的练习，并且不去查看答案，除非确信自己不能独立地解决。许多练习都只涉及所涵盖章节的直接应用——换句话说，它们只是练习——但有些还需要有点想法，甚至需要一些灵感。

希望每个人都能学会 STL。总之，尽情享受吧！

要获取本书的源代码，读者可以访问网址 http://www.apress.com 或 http://www.tupwk.com.cn/downpage/，搜索本书的英文或中文书名，找到相应的链接下载即可，读者也可用手机扫描本书封底的二维码，直接下载。

—Ivor Horton

目　录

第 1 章

STL 介绍

本章将会说明标准模板库(Standard Template Library，STL)背后的基本思想，让你对 STL 的各种对象如何相互支持有一个全局的把握。你将看到关于本章内容的更深入示例和讨论。本章将介绍以下内容：

- 什么是 STL
- 如何定义和使用模板
- 什么是容器
- 什么是迭代器以及如何使用它们
- 智能指针的重要性以及它们在容器中的用法
- 什么是算法，如何运用它们
- 数值库提供了哪些支持
- 什么是函数对象
- 如何定义和使用 lambda 表达式

除了介绍 STL 背后的一些基本思想，本章将会对一些你需要熟悉的 C++语言特性做一个简短的回顾，因为在后续章节中会频繁地使用它们。如果已经熟悉某些章节的主题，可以略过它们。

1.1 基本思想

STL 是一个功能强大且可扩展的工具集，用来组织和处理数据。为了最大限度地满足各种类型数据的需求，这个工具集全部由模板定义。虽然断定你肯定比较熟悉如何定义和使用类模板、函数模板，但本书还是会在下一节为你介绍一些它们的要领。STL 可以被划分为 4 个概念库：

- **容器库(Containers Library)**定义了管理和存储数据的容器。这个库的模板被定义在以下几个头文件中：array、vector、stack、queue、deque、list、forward_list、set、unordered_set、map 以及 unorderd_map。
- **迭代器库(Iterators Library)**定义了迭代器，迭代器是类似于指针的对象，通常被用于引用容器类的对象序列。这个库被定义在单个的头文件 iterator 中。
- **算法库(Algorithms Library)**定义了一些使用比较广泛的通用算法，可以运用到容器中的一组元素上。这个库的模板被定义在 algorithm 头文件中。
- **数值库(Numerics Library)**定义了一些通用的数学函数和一些对容器元素进行数值处理的操作。这个库也包含一些用于随机数生成的高级函数。这个库的模板被定义在 complex、cmath、valarray、numeric、random、ratio 以及 cfenv 这些头文件中。其中 cmath 已经有一些年头了，由于它包含了很多的数学函数，因此在 C++ 11 中得到了扩展。

只需要几行简洁明了的 STL 代码，很多复杂困难的任务都可以轻松地完成。例如，以下代码可以从标准输入流中读取任意数目的浮点数，然后计算并输出它们的平均值：

```
std::vector<double> values;
std::cout << "Enter values separated by one or more spaces. Enter Ctrl+Z
   to end:\n ";
values.insert(std::begin(values), std::istream_iterator<double>(std::cin),
   std::istream_iterator<double>());
std::cout << "The average is "
          << (std::accumulate(std::begin(values), std::end(values),
              0.0)/values.size())
          << std::endl;
```

只有 4 行代码，虽然每行都很长，但是我们没有使用循环去输入，因为这些工作 STL 已帮我们完成。可以很容易地将这段代码修改为从文件中获取数据，然后实现上面同样的任务。STL 强大、广泛的适用性，使它已经成为任何 C++程序员必备的工具箱。所有的 STL 名称都被定义在 std 命名空间中，所以不会在代码中显式地使用 std 来限定 STL 名称。当然，本书会在任何必要的地方使用 std 来限定。

1.2 模板

模板是一组函数或类的参数实现。编译器能够在需要使用函数或类模板时，用模板生

成一个具体的函数或类的定义，也可以定义参数化类型的模板。因此模板并不是可执行代码，而是用于生成代码的蓝图或配方。在程序中，如果一个模板从来没有被使用过，那么它将会被编译器忽略，不会生成可执行代码。一个没有被使用的模板可能会包含一些编程错误，并且包含这个模板的程序仍然可以编译和运行。当模板用于生成代码时，也就是当它被编译时，模板中的错误才会被编译器标识出来。

从模板生成的函数或类的定义是模板的实例或实例化。模板的参数值通常是数据类型，所以一个函数或类的定义可以通过 int 类型的参数值生成，也可以通过 string 类型的参数值生成。形参的参数值不一定是数据类型(非类型模板参数)；一个具体的形参可能是 int 类型，那么它就需要一个 int 类型的实参。下面是一个非常简单的函数模板示例：

```
template <typename T> T& larger(T& a, T& b)
{
    return a > b ? a : b;
}
```

这是一个可以比较两个参数值大小且返回较大参数值的函数的模板，使用这个模板的唯一限制是参数值的类型必须是可以用>比较的。类型参数 T 决定了具体创建什么样的模板实例。当使用 larger()时，编译器可以从提供的参数中推断出具体的参数类型，即使是隐式提供。例如：

```
std::string first {"To be or not to be"};
std::string second {"That is the question."};
std::cout << larger(first, second) << std::endl;
```

上述代码需要包含 string 头文件。编译器可以从输入参数推断出 T 为 string 类型。如果想指定 T 为 string 类型，可以编写语句 larger<std:string>(first,second)。当函数的两个输入参数类型不同时，需要明确指出模板类型参数。例如，如果编写语句 larger(2,3.5)，编译器在推断 T 的类型时，就会产生多义性——T 可以是 int，也可以是 double，这样使用就会产生错误消息。如果将 larger(2,3.5)写为 larger<double>(2,3.5)，就可以消除这个问题。

以下是一个模板类示例：

```
template <typename T> class Array
{
private:
  T* elements;                                  // Array of type T
  size_t count;                                 // Number of array elements

public:
  explicit Array(size_t arraySize);             // Constructor
  Array(const Array& other);                    // Copy Constructor
  Array(Array&& other);                         // Move Constructor
  virtual ~Array();                             // Destructor
  T& operator[](size_t index);                  // Subscript operator
  const T& operator[](size_t index) const;      // Subscript operator-const
                                                // arrays
  Array& operator=(const Array& rhs);           // Assignment operator
  Array& operator=(Array&& rhs);                // Move assignment operator
```

```
    size_t size() { return count; }              // Accessor for count
};
```

size_t 类型别名定义在 cstddef 头文件中，表示无符号整型。上面的代码是元素类型为 T 的数组模板。如果愿意，可以这样隐式地定义数组 Array<T>。在模板定义的外面——外部成员函数的定义中，必须这样写 Array<T>。赋值操作符允许将一个 Array<T>对象赋值给另一个对象，这个操作在普通数组中是无法完成的。如果想禁用这个功能，仍然需要声明 operator=()为模板的成员函数。如果不这样做，编译器会自动为模板实例生成一个默认的 public 类型的赋值操作函数。为了阻止这种情况发生，应该按照如下方式进行定义：

```
Array& operator=(const Array& rhs)=delete; // No assignment operator
```

一般来说，如果想定义任何拷贝构造函数或移动构造函数、拷贝赋值运算符或移动赋值运算符以及析构函数，都需要定义这 5 个类成员，或者用 deleted 指定你不想要的成员。

■ **注意：** 实现了移动构造函数和移动赋值运算符的类都有移动语义。

---

size()成员函数在类模板中默认以内联的方式实现，所以不再需要外部定义。类模板成员函数的外部定义是放在头文件中的模板，通常和类模板在一个头文件中。一个成员函数甚至可以不依赖类型参数 T，如果 size()在类模板中没有定义，那么 size()将需要一个模板定义。定义了成员函数的模板的类型参数列表必须和类模板的类型参数列表相同，下面介绍如何定义构造函数：

```
template <typename T>                    // This is a function template with
                                         // parameter T
Array<T>::Array(size_t arraySize) try : elements {new T[arraySize]}, count
    {arraySize}
{}
catch(const std::exception& e)
{
  st::cerr << "Memory allocation failure in Array constructor." << std::endl;
  rethrow e;
}
```

为元素分配内存的操作可能会抛出一个异常，所以构造函数是一个 try 代码块，这能捕获到异常，处理必须被重新抛出的异常——如果不重新抛出那些 catch 语句块中的异常，那么它会被重新抛出。在构造函数名称的限定中，模板类型参数 T 是必要的，因为它将函数模板的定义和类模板关联起来了。需要注意的是，对于成员函数，不必使用 typename 这个关键字，它仅用于模板的参数列表中。

当然，可以以内联的方式为类模板的成员函数指定一个外部模板。例如，下面的代码将展示 Array 模板的拷贝构造函数可能的定义方式：

```
template <typename T>
inline Array<T>::Array(const Array& other)
try : elements {new T[other.count]}, count {other.count}
{
```

```
    for (size_t i {}; i < count; ++i)
      elements[i] = other.elements[i];
  }
  catch (std::bad_alloc&)
  {
    std::cerr << "memory allocation failed for Array object copy." << std:: endl;
  }
```

这就假定 T 类型适用于赋值运算符，在使用它之前，你不会看到模板的代码，也不会意识到赋值运算符上的依赖。这说明除了先前提到的那些动态分配内存的类的其他成员，定义一个赋值运算符是多么重要。

注意：当指定一个模板的参数时，class 和 tyname 关键字是可以互换使用的。因此，当定义一个模板时，可以写作 template<typename T>或 template<class T>。因为 T 不必是类类型，所以本书更喜欢选择使用 typename，因为当 T 参数既可以是基本类型，也可以是类类型时，这种表达看起来更直观。

编译器因为一个特定类型对象的定义，实例化了一个类模板。这里有一个示例：

```
Array<int> data {40};
```

每个类模板类型参数都需要一个实参，除非有默认实参。当这条语句被编译时，发生了三件事：因为 Array<int>类的定义被创建，所以确定了参数类型；因为必须调用构造函数去生成一个对象，所以生成了构造函数的定义；析构函数也被生成用来销毁对象。这就是编译器创建和释放一个数据对象所做的全部事情，也是这个时候从模板生成的唯一代码。通过用 int 替换模板中的 T，生成了类的定义，这非常巧妙。编译器只编译程序使用的成员函数，所以不需要得到模板参数被替换的完整类。基于数据对象被定义的规则，可以这样定义类：

```
class Array<int>
{
private:
  int* elements;
  size_t count;

public:
  explicit Array(size_t arraySize);
  virtual ~Array();
};
```

可以看到，只有构造函数和析构函数这两个成员函数。编译器不会创建任何和生成对象实例无关的东西，也不会包含程序中没有用到的模板代码。

可以为模板定义别名，当使用 STL 时，这一点非常有用。这里有一个为模板定义别名的示例：

```
template<typename T> using ptr = std::shared_ptr<T>;
```

这个模板定义 ptr<T>作为智能指针模板类型 std::shared_ptr<T>的别名，这样的话就可

以在代码中用 ptr<std::string>代替 std::shared_ptr<std::string>，这显然更简洁，更容易阅读。下面使用 using，代码会更加简化：

```
using std::string;
```

现在就可以在代码中用 ptr<string>代替 std::shared_ptr<std::string>，模板的类型别名能使代码更容易阅读，也更方便编写。

1.3 容器

容器是 STL 各种功能的基础，因为 STL 的其他部分和容器都有关联。容器是一个以特定方式存储和组织其他对象的对象。当使用容器时，不可避免地会使用 iterators 去访问其数据，所以你需要对 iterators 有一个较好的理解。STL 提供了以下几种类型的容器：

- **序列容器(Sequence Containers)**以线性组织的方式存储对象，和数组类似，但是不需要连续的存储空间。可以通过调用成员函数或者通过一个迭代器以线性方式访问对象。在一些地方，可以通过索引下标([])的方式来访问。
- **关联容器(Associative Containers)**存储了一些和键关联的对象。可以通过一个相关联的键从关联容器中获取对应的值。也可以通过迭代器从关联容器中得到对象。
- **容器适配器(Container Adapters)**是提供了替换机制的适配类模板，可以用来访问基础的序列容器或关联容器。

有一点需要特别注意，除非对象是右值的——临时对象——一种有移动语义的类型，否则所有的 STL 容器存储的都是对象的副本。STL 要求移动构造函数和赋值运算符必须被指定为 noexcept，这就意味着它们不会抛出异常。如果在容器中添加了一个不具有移动特性的对象，然后修改了原始对象，那么原始对象和容器中的对象将会不同。然而，当获取一个对象时，得到的是一个容器中对象的引用，所以可以修改这个对象的值。对象的副本由拷贝构造函数获得。对于一些对象来说，拷贝可能会产生大量的开销，这样的话，最好在容器中存储这个对象的指针。如果这个对象的移动语义被实现，也可以存储移动对象。

警告：不要在以基类为元素的容器中存放派生类，这将会导致派生类对象的切片。如果想在容器中访问派生类，则需要使用多态。也就是说，在存储基类指针或基类智能指针的容器中存储派生类指针。

容器在堆上存放对象，并且自动管理它们所占用的内存。存储类型 T 对象的容器的空间分配由 allocator 管理，allocator 的类型由模板的参数指定。默认的参数类型是 std::allocator<T>，一个这种类型的 allocator 为对应类型的对象分配堆空间，这使你有了自己提供 allocator 的可能性。因为性能，你可能选择自己做这些事，但是这种情况很少。大多数时候，默认的 allocator 性能是不错的。如何定义一个 allocator 是一个高级话题，在本书中并不想做深入讨论，所以当最后一个模板的参数表示 allocator 时，省略了它，使用默认值。std::vector<typename T, typename Allocator>模板有一个默认的 allocator——std::allocator<T>，所以可以简写为 std::vector<typename T>。指出这一点，主要是想让你知

道可以自己提供一个 allocator。

一个 T 类型的对象如果要存放在容器中，需要满足一些特定的要求，这些要求最终取决于你对这些元素执行的操作。一个容器通常会复制、移动、交换其元素。这有一个基本满足在容器中存放 T 类型对象的示例：

```
class T
{
  public:
    T();                              // default constructor
    T(const T& t);                    // Copy constructor
    ~T();                             // Destructor
    T& operator=(const T& t);         // Assignment operator
};
```

考虑到在大多数情况下，编译器已经默认实现了上述几种成员函数，这也是大多数类需要实现的成员函数。注意到我们并没有为 T 定义 operator<()函数，但是如果在关联容器(例如 map、set)中使用对象的话，对象需要定义 operator<()。如果容器的元素没有提供小于运算符的话，像 sort()和 merge()这类算法就不能运用到这些元素上。

■ 注意：如果对象类型不符合所用容器的一些要求，或者你误用了容器模板，就可能会得到一些编译器对于 STL 内部头文件的编译错误。当这种事情发生时，不要急于提交 STL 中的错误，而是小心检查使用了 STL 的代码。

1.4　迭代器

迭代器是一个行为类似于指针的模板类对象。只要迭代器 iter 指向一个有效对象，就可以通过使用*iter 解引用的方式来获取一个对象的引用。如果 iter 指向一个可以访问成员的类对象，类成员就可以通过 iter->member 来使用。所以，使用迭代器就像使用指针。

当要以某种方式处理容器的元素时，可以通过使用迭代器来访问它们，尤其是要使用 STL 算法时，迭代器将算法和不同类型容器的元素联系了起来。迭代器将算法和数据源分离出来，算法不需要知道数据来源于容器。迭代器是一个定义在 iterator 头文件中的模板类型实例，这个头文件被所有定义了容器的头文件所包含。

通常会使用一对迭代器来定义一段元素，这些元素可能是容器中的对象，或是标准数组中的元素，或是字符串中的字符，总之可以是任意支持迭代器的对象的元素。一段元素是一个通过起始迭代器指向第一个元素，通过结束迭代器指向最后一个元素的元素序列。甚至当这个元素序列是这个容器中的元素的子集时，第二个迭代器仍然指向这个序列的最后一个元素而不是这段元素的最后一个元素。结束迭代器表示一个容器中的所有元素，不会指向具体某个元素，所以不能解引用。STL 中的迭代器提供一个标准的机制用于识别一段元素。一段元素的规格是独立于元素来源的，因此一个给定的算法可以运用到来自于任何容器的一段元素上，只要迭代器能够满足这个算法的一些要求。稍后会介绍更多不同迭代器的一些特性。

一旦明白迭代器是如何工作的，就很容易定义自己的模板函数来处理那些数据序列，它们由迭代器作为参数指定，然后函数模板实例就可以被运用到来自任何数据源的一段元素上。这段代码做的工作其实就像从数组中读取数据。你将在后面看到一个这样操作的示例。

1.4.1 获取迭代器

可以通过调用容器对象的 begin()和end()函数来获取容器的迭代器；返回的这两个迭代器分别指向第一个和最后一个元素。end()返回的迭代器并没有指向一个有效的元素，所以既不能解引用，也不能递增它。字符串类就有这样的函数，所以也可以这样去获取它们的迭代器。还可以将容器作为参数，通过调用全局函数 std:begin()和 std:end()来获取迭代器，这和使用容器的 begin()和 end()来获取迭代器是一样的，这两个全局函数被定义在 iterator 头文件的模板中。全局函数begin()和end()也可以用普通的数组或字符串对象作为参数，因此这是一种通用的获取迭代器的方式。

迭代器允许以一种逐步自增步进的方式从一个元素移到另一个元素，就像图 1-1 演示的那样。图 1-1 中的容器表明，它可能是一个字符串对象或数组，也可能是一个 STL 容器。通过将自增的起始迭代器和结束迭代器进行比较来判断是否访问到最后一个元素。你还能对迭代器进行一些其他操作，不过这些操作取决于迭代器的类型，反过来也取决于使用的容器的类型。有两个全局函数cbeign()和cend()可以返回数组、容器或字符串对象的常量迭代器。记住，常量迭代器指向的是一个常量，但是仍然可以修改这个迭代器本身的值。本章稍后的部分会介绍一些返回其他类型迭代器的全局函数。

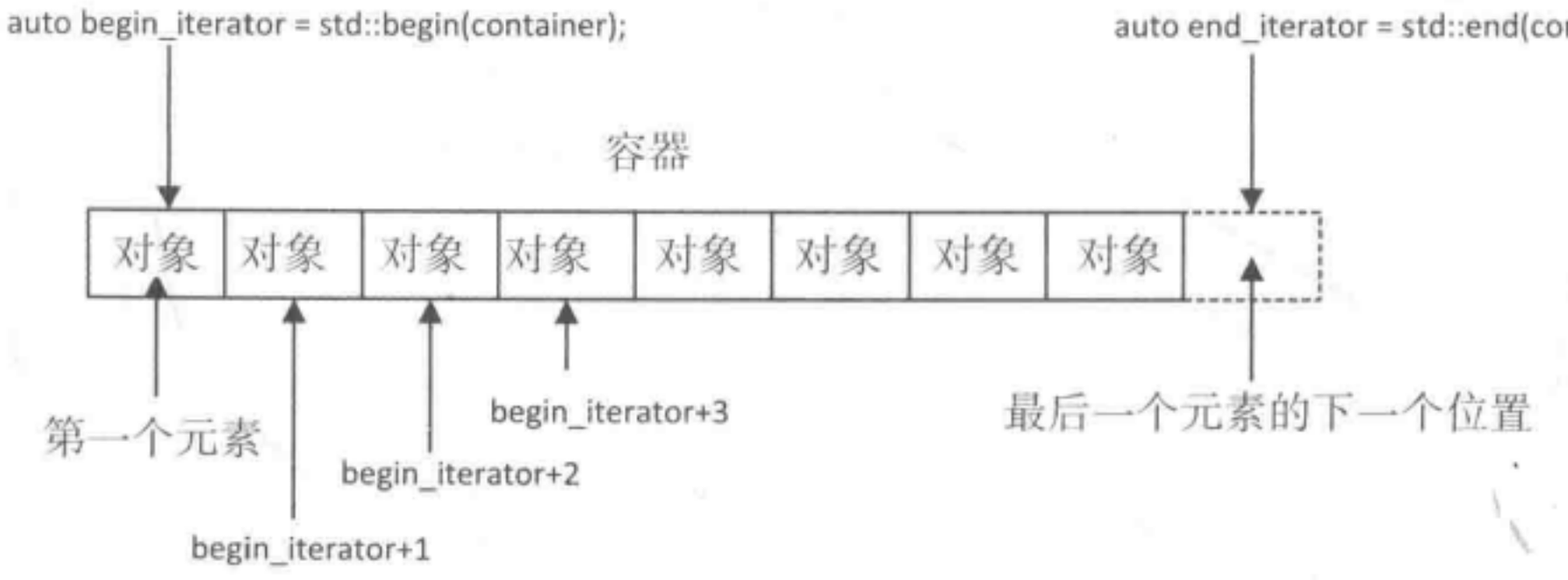


图 1-1　迭代器的操作

1.4.2 迭代器的类别

所有类型的迭代器都必须有一个拷贝构造函数、一个析构函数以及一个拷贝赋值运算符。迭代器指向的对象必须是可交换的(swappable)；下一节再进一步解释这意味着什么。5种类型的迭代器分别反映了它们不同层次的功能。不同的算法要求具有不同功能的迭代器，以此来确认算法能做哪些操作。不同类别的迭代器并不是一种新类型的迭代器模板。通过确定迭代器模板的参数，可以确认一个迭代器类型支持的类别。稍后会在本章进行更深入的解释。

从一个容器能够获取什么类型的迭代器，取决于这个容器的类型。算法可以通过传入

的迭代器参数的类别，来判断迭代器具有什么样的功能。一个算法可以两种方式使用迭代器参数的类别：第一种，为了满足操作，它会确立需要满足的最小功能要求；第二种，如果超出了迭代器的最小要求，算法使用扩展的功能可以更高效地执行运算。当然，算法只能运用到那些能提供具有要求功能的迭代器的容器中的元素上。

下面我们从简单到复杂，依次列出不同类别的迭代器：

1) **输入迭代器(input iterators)**提供对对象的只读访问。如果 itex 是一个输入迭代器，它必须支持表达式*iter 以引用它所指向的值。输入迭代器只能单个使用，这也就意味着迭代器自增后，只能通过使用新的迭代器来访问上一个它所指向的元素。每次想访问一个序列时，就需要创建一个新的迭代器。可以对输入迭代器做这些操作：++iter 或 iter++，iter1==iter2 和 iter1!=iter2。需要注意的是，*iter 并没有减量运算，可以在输入迭代器上使用 iter->member 表达式。

2) **输出迭代器(output iterators)**提供对对象的只写访问。如果 iter 是一个输出迭代器，那么它允许我们赋一个新的值，因此支持表达式*iter=new_value。输出迭代器只能单个使用。每次想写一个序列时，需要创建一个新的迭代器。可以对输出迭代器做这些操作：++iter 或 iter++，以及*iter。需要注意的是，*iter 并没有减量操作，输出迭代器是只写的，所以不能使用 iter->member 这样的表达式。

3) **正向迭代器(forward iterators)**结合了输入和输出迭代器的功能，并且可以使用多次，因此可以随意多次用它进行读或写操作。进行什么样的操作取决于何时需求正向迭代器。replace()算法搜索一个序列，替换所有需要正向迭代能力的元素。

4) **双向迭代器(bidirectional iterators)**具有和正向迭代器同样的功能，但允许进行前向和后向遍历。因此除了自增来移到另一个元素，也可以使用前缀或后缀的自减运算(--iter 和 iter--)来移到前一个元素。

5) **随机访问迭代器(random access iterators)**提供了和双向迭代器同样的功能，但是能够支持对元素的随机访问。除了支持双向迭代器的一些操作之外，也支持一些其他操作：

- 通过一个整数进行自增或自减：iter+n、iter−n、iter+=n、iter−=n。
- 通过一个整数进行索引访问：iter[n]，等同于*(iter+n)。
- 两个迭代器之间的差异：iter1−iter2，得到两个迭代器之间元素的个数。
- 迭代器的比较：iter1<iter2、iter1>iter2、iter1≤iter2、iter1≥iter2。

对一个序列进行排序，需要用随机访问迭代器指定这个序列。可以对随机访问迭代器使用下标运算。例如一个随机访问迭代器 first，表达式 first[3]就等同于*(first+3)，所以可以访问第 4 个元素。一般来说，使用 iter[n]这种表达式，n 是一个相对于 iter 的偏移量，我们能够通过它得到一个距离 iter n 个距离的元素的引用。需要注意的是，在下标中使用的索引值并没有被检查，所以无法阻止索引越界。

每种类别的迭代器都由一个称作迭代器标签类(iterator tag class)的空类来指定，iterator tag class 通常作为迭代器模板的参数使用。 iterator tag class 的唯一目的是指定一个特定类型的迭代器能够做什么，所以它们可以作为迭代器模板的参数使用。标准的 iterator tag class 如下所示：

```
input_iterator_tag
output_iterator_tag
```

```
forward_iterator_tag 派生于 input_iterator_tag
bidirectional_iterator_tag 派生于 forward_iterator_tag
random_access_iterator_tag 派生于 bidirectional_iterator_tag
```

以上这些类的继承结构反映了迭代器类别的累加性质。当一个迭代器模板的实例被创建时，第一个模板类型的参数是一个 iterator tag class，这将决定迭代器具有什么样的功能。在第 2 章将演示如何定义自己的迭代器，之后将介绍如何指定它的类别。

如果一个算法要求特定类别的迭代器，你就不能使用一个低级别的迭代器，而是需要使用一个高级别的迭起器。正向迭代器双向迭代器，随机访问迭代器可以是可变的或不可变的，这取决于解引用一个迭代器是否能够获取到一个引用或 const 引用。显然，不能对一个解引用的迭代器使用左赋值。

迭代器的特性取决于它所来自容器的类型。例如，vector 和 deque 提供了随机访问迭代器，也就表明这些容器中的元素都可以随机访问。输入迭代器、输出迭代器、正向迭代器通常用于指定算法的参数，表明它们满足算法所要求的最小条件。本书稍后将在关于对容器中元素使用算法的示例的基础上，更深入地解释迭代器。在一个实际使用场景下，会更容易理解。同时，下面这个简单的示例演示了迭代器对一个数组进行的操作：

```
// Ex1_01.cpp
// Using iterators
#include <numeric>          // For accumulate() - sums a range of elements
#include <iostream>         // For standard streams
#include <iterator>         // For iterators and begin() and end()

int main()
{
  double data[] {2.5, 4.5, 6.5, 5.5, 8.5};
  std::cout << "The array contains:\n";
  for (auto iter = std::begin(data); iter != std::end(data); ++iter)
    std::cout << *iter << " ";
  auto total = std::accumulate(std::begin(data), std::end(data), 0.0);
  std::cout << "\nThe sum of the array elements is " << total << std::endl;
}
```

可以看到，我们以数组作为全局函数 begin()和 end()的参数，得到了数组元素的迭代器。我们用这个迭代器循环列出了元素的值，用表达式*iter 来解引用一个迭代器，以获取引用对象的值。当然，可以在循环中自增 iter，如下所示：

```
for (auto iter = std::begin(data); iter != std::end(data);)
  std::cout << *iter++ << " ";
```

直接包含 iterator 头文件是多余的，因为在容器的头文件中都包含了 iterator。我们包含了头文件 numeric，这个头文件定义了 accumulate()函数的模板。通过两个参数指定元素的范围——一个指向第一个元素，另一个指向最后一个元素，这样 accumulate()就可以计算出这些元素的和，第三个元素是元素和的初始值。accumulate()函数适用于任何支持加法运算的序列，同样也适用于任何定义了 operator+()函数的类对象。

注意：当对容器使用 accumulate()时，你会发现另一个版本的函数模板，它允许你指定一个双目运算来替代默认的+。

1.4.3　流迭代器

可以使用流迭代器，在流和可以通过迭代器访问的数据源之间传输文本模式的数据。因为 STL 算法可以通过指定一对迭代器来输入序列，所以对于任何可以通过输入流访问的数据源，都可以将算法运用到它们的元素上。这就意味着，例如算法，可以像运用到容器中的对象上一样，运用到流中的对象上。算法可以运用到任何地方，只要能提供一个合适的迭代器。稍后，将解释如何使一个迭代器变得适用。同样地，也可以使用一个输出流迭代器，将一系列的元素传送到一个输出流。标准流迭代器有一个作为基类的迭代器模板。

可以创建流迭代器对象，用来处理指定类型的流对象数据。数据的类型就是迭代器模板的参数类型，流对象是构造函数的参数。一个 istream_iterator<T>就是一个输入流迭代器，它可以从一个 istream 读取 T 类型的对象。istream 可以是一个文件流，也可以是一个标准的输入流 cin。我们通过使用>>运算符来读取对象。所以读取的对象要支持这个运算符。

istream_iterator<T>无参构造函数可以创建一个结束迭代器来判断流是否结束。显然，流迭代器并不能用来传输混合类型的数据。istream_iterator 对象默认会忽略空格，可以通过使用 std::noskipws 操作底层的输入流来覆盖这个特性。一个 istream_iterator 只能使用一次，如果想再次从流中读取对象，需要使用一个新的 istream_iterator 对象。

输出流迭代器(ostream_iterator)弥补了输入流迭代器(istream_iterator)的不足，它为对象提供了一次向输出流输出的功能。使用<<运算符创建对象。当创建一个 ostream_iterator 对象时，可以选择字符串分隔符，用来分隔每个输出对象。

这里有一个使用输入流迭代器的示例：

```
// Ex1_02.cpp
// Using stream iterators
#include <numeric>              // For accumulate() - sums a range of elements
#include <iostream>             // For standard streams
#include <iterator>             // For istream_iterator

int main()
{
  std::cout << "Enter numeric values separated by spaces and enter Ctrl+Z
     to end:" << std::endl;

  std::cout << "\nThe sum of the values you entered is "
            << std::accumulate(std::istream_iterator<double>(std::cin),
               std::istream_iterator<double>(), 0.0)
            << std::endl;
}
```

这段代码通过 accumulate()函数，计算出 cin 的输入流迭代器输入的一些数据的和。我们可以输入任意个数的数据。第二个参数是一个流结束迭代器，当出现流结束的条件时(就像文件流 EOF 标识)，用来和第一个参数指定的迭代器进行匹配；当从键盘输入 Ctrl+Z 时，

会导致这样的事情发生。

1.4.4 迭代器适配器

迭代器适配器是一个类模板，它为标准迭代器提供了一些特殊的行为，使它们能够从迭代器模板得到派生。适配器类模板定义了三种不同的迭代器：反向迭代器(reverse iterators)、插入迭代器(insert iterators)和移动迭代器(move iterators)。它们是通过下面几个模板类定义的：reverse_iterator、insert_iterator 和 move_iterator。

1. 反向迭代器

反向迭代器的工作机制和标准迭代器的相反，可以创建双向或随机迭代器版本的反向迭代器。容器的成员函数 rbegin()和 rend()，分别返回一个指向最后一个元素的反向迭代器，以及一个指向最开始元素的前一个位置的反向迭代器。同样名称的全局函数，作用和成员函数是一样的，如图 1-2 所示。

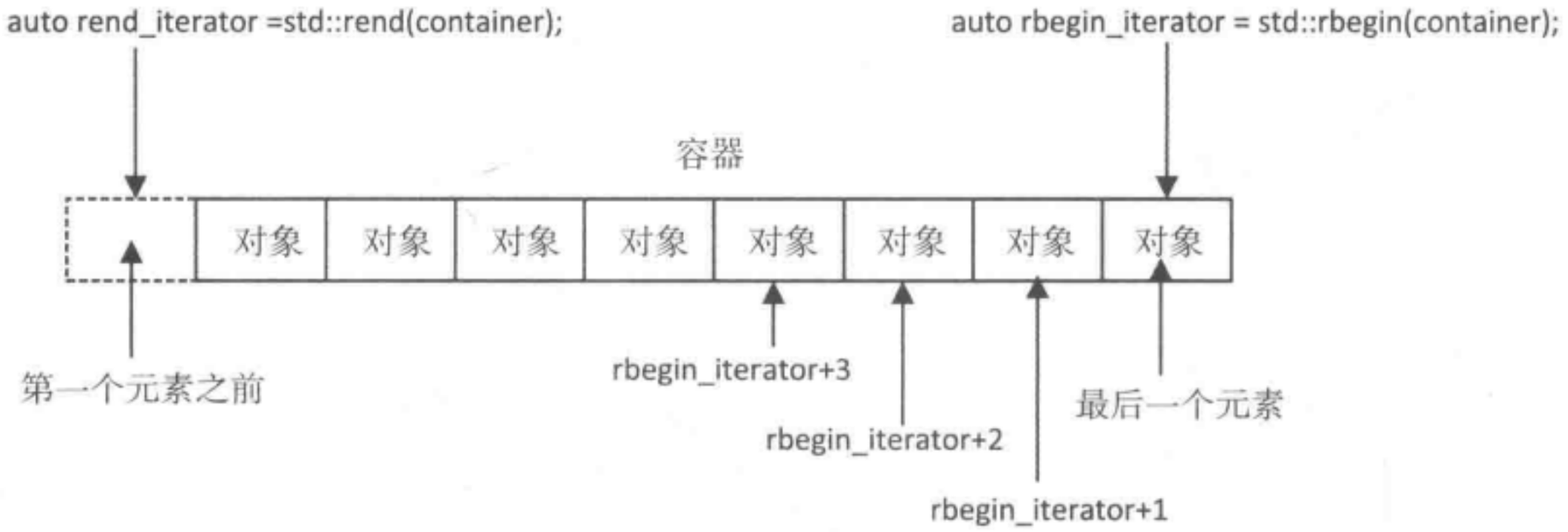


图 1-2 反向迭代器的操作

可以像使用标准随机访问迭代器一样，使用反向随机访问迭代器进行下标操作，效果是相反的。对于标准迭代器 iter 来说，表达式 iter[n]代表在 iter 后面，和 iter 指向的元素有 n 个元素间隔的元素，所以等同于*(iter+n)。对于反向迭代器 riter 来说，表达式 riter[n]等同于*(riter+n)，所以 riter[n]代表在 riter 前面，和 riter 指向的元素有 n 个元素间隔的元素。

图 1-3 显示了容器的反向迭代器和标准迭代器的关系。可以看到，容器元素的反向迭代器相对于正常的迭代器来说，向左偏移了一些。每一个反向迭代器都包含了一个标准迭代器，里面的迭代器也有相似的移位，所以没有指向相同的元素。每一个反向迭代器对象都有一个成员函数base()，可以返回一个基础的迭代器，因为它也是一个标准的迭代器，所以和反向迭代器的作用是相反的。反向迭代器 riter 的基础迭代器指向 riter 的下一个位置，如图 1-3 所示。有些容器的成员函数并不接受反向迭代器，当准备使用一个算法时，正好出现这种情况，已经使用了反向迭代器，可以调用base()去获取一个和反向迭代器对应的标准迭代器。显然，需要注意的事实是，基础迭代器总是指向反向迭代器指向的下一个位置，你会在下一节更多地了解这些内容。

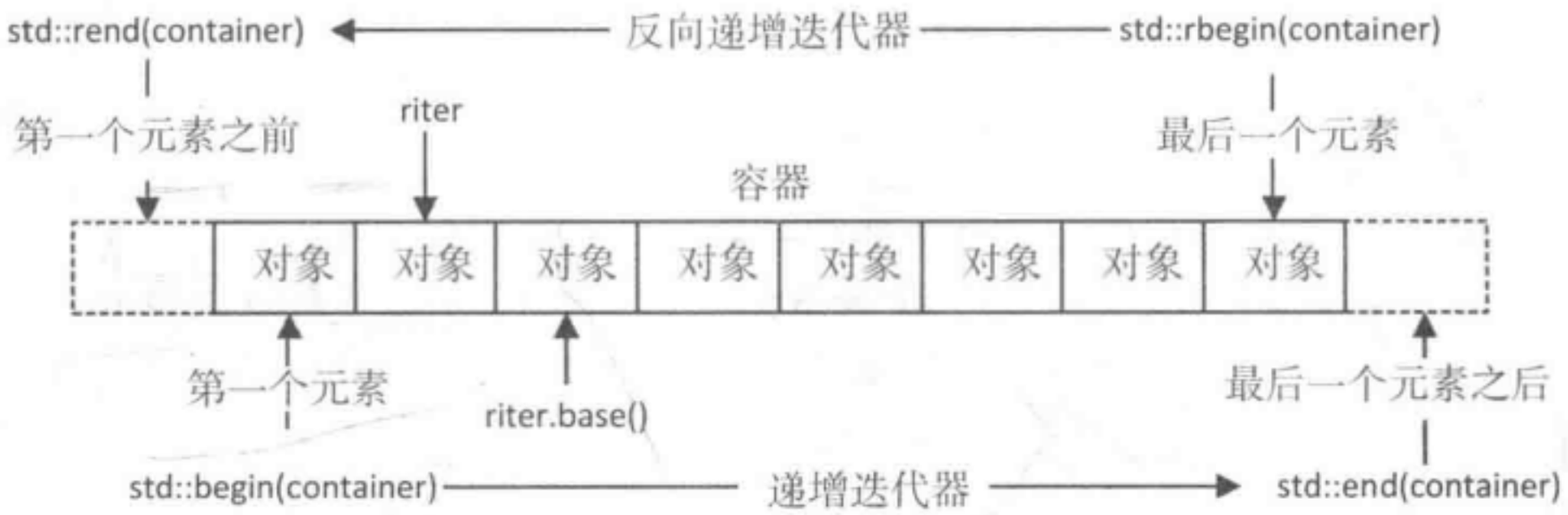


图 1-3　标准迭代器和反向迭代器是如何和容器关联的

2. 插入迭代器

尽管插入迭代器基于标准迭代器，但是它们的功能却有很大不同。一般的迭代器只能访问或改变序列中存在的元素。插入迭代器通常用于在容器的任何位置添加新的元素。插入迭代器不能被运用到标准数组和 array<T,N>这样的容器上，因为它们的元素个数是固定的。

有三种插入迭代器：

- 后向插入迭代器(back_insert_iterator)通过调用成员函数 push_pack()将一个新的元素添加到容器的尾部。vector、list、deque 容器都有一个 push_back()函数。如果容器没有定义 push_back()函数，那么后向插入迭代器无法使用。以容器作为全局函数 back_inserter()的参数传入可以得到一个后向插入迭代器对象。
- 前向插入迭代器(front_insert_iterator)通过调用成员函数 push_front()将一个新的元素添加到容器的头部。list、forward_list、deque 容器都有一个 push_front()函数。如果容器没有定义 push_front()函数，那么前向插入迭代器无法使用。以容器作为全局函数 front_inserter()的参数传入可以得到一个前向插入迭代器对象，显然这些容器必须是 list、forward_list 或 deque 容器类型。
- 可以使用插入迭代器(insert_iterator)向任何有 insert()函数的容器中插入一个新的元素。因为在 string 类的头文件中定义了一个 insert()函数，所以插入迭代器可以正常使用。以容器作为全局函数 inserter()的第一个参数可以得到一个插入迭代器对象，第二个参数是一个指向容器中插入位置的迭代器。

插入迭代器一般被用做拷贝或生成一段元素的算法的参数。在后面的章节你会看到它们的具体应用。

3. 移动迭代器

移动迭代器是从普通的迭代器中创建的，这个迭代器指向一定范围内的元素。可以用移动迭代器将某个范围的类对象移动到目标范围，而不需要通过拷贝去移动。可以将移动迭代器作为输入迭代器，将所指向的对象转换为右值引用，这样就能移动对象而不是去拷贝。移动迭代器会使指向的一段数据元素处于一种不确定的状态，这样就不能再次使用它们了。例如，make_move_iterator()被定义在 iterator 头文件中，因为可以将从 begin()和 end()得到的迭代器作为 make_move_iterator()的参数来得到一个移动迭代器，所以可以通过将从 begin()和 end()获取的迭代器作为 make_move_iterator()的参数来移动容器中的一段元素。在本书稍后部分，你将会看到使用移动迭代器的示例。

1.5 迭代器上的运算

迭代器的头文件定义了 4 个实现了迭代器运算的函数模板：

- advance()需要两个参数，第一个参数是迭代器 iter，第二个参数是整数值 n，将 iter 移动了 n 个位置。迭代器可以是任何具有输入迭代器功能的迭代器；如果迭代器是一个双向迭代器或随机访问迭代器，第二个参数也可以是负数。这个函数没有返回值。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::begin(data);
std::advance(iter, 3);
std::cout << "Fourth element is " << *iter << std::endl;
```

- distance()需要两个迭代器作为参数，可以返回两个迭代器之间的元素个数。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
std::cout << "The number of elements in data is "
          << std::distance(std::begin(data), std::end(data))
          << std::endl;
```

- next()需要两个参数，第一个参数是迭代器 iter，第二个参数是整数值 n，n 的默认值是 1。通过这两个参数，next()可以得到 iter 正向偏移 n 之后所指向位置的一个迭代器。迭代器 iter 必须是具有正向迭代器能力的迭代器。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::begin(data);
auto fourth = std::next(iter, 3);
std::cout << "1st element is " << *iter << " and the 4th is " << *fourth <<
std::endl;
```

- prev()需要两个参数，第一个参数是 iter，第二个参数是整数值 n，n 的默认值为 1。通过这两个参数，prev()返回 iter 反向偏移 n 之后所指向位置的一个迭代器。迭代器器 iter 必须是具有双向迭代器功能的迭代器。

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::end(data);
std::cout << "Fourth element is " << *std::prev(iter, 3) << std::endl;
```

显然这些函数都能通过使用随机访问迭代器来获取算术运算后的结果，但是其他类型的低级别迭代器就不能。这些函数能够简化代码。在其他不具有这种能力的迭代器上，如果想实现和 advance()相同的功能，需要写一个循环。

1.6 智能指针

指针是 C++语言的一部分，通常叫作原生指针，因为这种类型的变量都指向一个地址。原生指针可以指向自动变量、静态变量或堆上生成的变量。智能指针是一个可以模仿原生

指针的模板类。在很多情况下，它们没有任何差别。但是，有些情况下，它们还是有两点差别：

- 智能指针只能用来保存堆上分配的内存的地址。
- 不能像对原生指针那样，对智能指针进行一些自增或自减这样的算术运算。

对于在自由存储区创建的对象，通常使用智能指针而不是原生指针。智能指针的巨大优势是，不用担心不小心释放了堆内存，因为当不需要时，一个对象的内存会被分配给一个智能指针。智能指针自动释放当对象不再需要时，这意味着消除了内存泄漏的可能性。

可以在容器中使用智能指针，当使用一个类对象时，这一点尤其有用。如果使用一个类的基类作为智能指针的类型参数，可以用它指向一个派生类对象，这保存了一个对象的指针而不是对象，因而能够保持这个对象的多态性。智能指针的模板定义在 memory 头文件中，必须在代码中包含这个头文件来使用它们。在 std 命名空间中定义了三种不同类型的智能指针模板：

- unique_ptr<T>对象就像一个指向类型 T 的指针，而且 unique_ptr<T>是排他的，这意味着不可能有其他的 unique_pt<T>指向同一个地址。一个 unique_ptr<T>对象完全拥有它所指向的内容。不能指定或复制一个 unique_ptr<T>对象。可以通过使用一个定义在 utility 头文件中的 std::move()函数来移出一个 unique_ptr<T>对象存储的地址。在进行这个操作之后，之前的 unique_ptr<T>变为无效。当需要独占一个对象的所有权时，可以使用 nique_ptr<T>。
- shared_ptr<T>对象就像一个指向类型 T 的指针，和 unique_ptr<T>不同的是，多个 shared_ptr<T>可以指向同一个地址，因此 shared_ptr<T>允许共享一个对象的所有权。引用计数保存了指向给定地址的 shared_ptr<T>的数量，每当一个新的 shared_ptr<T>指针指向一个特定堆地址时，引用计数就会加 1。当一个 shared_ptr<T>被释放或者指向了新的地址时，引用计数就会减 1。当没有 shared_ptr<T>指向这个地址时，引用计数将会变为 0，在堆上为这个对象分配的内存就会自动释放。所有指向同一个地址的 shared_ptr<T>都能得到引用计数的值。
- weak_ptr<T>可以从一个 shared_ptr<T>对象创建，它们指向相同的地址。创建一个 weak_ptr<T>不会增加 shared_ptr<T>对象的引用计数，所以它会阻止指向的对象销毁。当最后一个 shared_ptr<T>引用被释放或重新指向一个不同的地址时，它们所指向对象的内存将被释放，即使相关的 weak_ptr<T>可能仍然存在。

使用 weak_ptr<T>对象的主要原因是，我们可能在不经意间创建一个循环引用。循环引用从概念上说就是一个 shared_ptr<T>对象 pA，指向了另一个 shared_ptr<T>对象 pB，在这种情况下，两个对象都不能被释放。在实际中，循环引用的发生要比这个复杂。设计 weak_ptr<T>就是为了避免循环引用的问题。通过使用 weak_ptr<T>指向一个 shared_ptr<T>所指向的对象，就可以避免循环引用，稍后会对此做一些解释。当最后一个 shared_ptr<T>对象析构时，它所指向的对象也析构了，这个时候任何和 shared_ptr<T>关联的 weak_ptr<T>对象都指向一个无效的对象。

1.6.1 使用 unique_ptr<T>指针

unique_ptr<T>对象唯一指向了一个对象，所以它独享了这个对象的所有权。当一个unique_ptr<T>对象析构时，它所指向的对象也析构了。当不需要使用多个智能指针，想独享对象的所有权时，可以选择使用这种类型的智能指针。当一个对象被一个 unique_ptr<T>指向时，也可以通过生成一个原生指针来访问对象。这里有一个使用构造函数生成一个unique_ptr<T>对象的示例：

```
std::unique_ptr<std::string> pname {new std::string {"Algernon"}};
```

在堆上生成的字符串对象被传入 unique_ptr<string>构造函数。默认的构造函数会生成一个内部原生指针为空指针的 unique_ptr<T>对象。

一种更好地生成 unique_ptr<T> 对象的方式是使用 make_unique<T>()函数，这是一个定义在 memory 头文件中的函数模板：

```
auto pname = std::make_unique<std::string>("Algernon");
```

该函数通过将参数传递给类的构造函数，来生成一个堆上的字符串对象，并且返回一个指向这个对象的指针。可以按照 T 构造函数的要求，尽可能多地为 make_unique<T>()函数提供参数。下面是一个示例：

```
auto pstr = std::make_unique<std::string>(6, '*');
```

这两个参数会被传入 string 构造函数中，生成一个包含“******”的对象。

可以通过解引用的方式来使用一个对象，就像使用原生指针一样：

```
std::cout << *pname << std::endl; // Outputs Algernon
```

可以生成一个指向数组的 unique_ptr<T>对象。例如：

```
size_t len{10};
std::unique_ptr<int[]> pnumbers {new int[len]};
```

这会生成一个 unique_ptr 对象，用来指向一个在空闲空间生成的具有 len 个元素的数组，同样也可以调用 make_unique<T>()函数得到相同的结果：

```
auto pnumbers = std::make_unique<int[]>(len);
```

同样，也可以生成一个指针，用来指向一个在堆上生成的具有 len 个元素的数组。可以对 unique_ptr 变量使用索引来访问数组元素。这里展示了如何改变数组的值：

```
for(size_t i{} ; i < len ; ++i)
  pnumbers[i] = i*i;
```

这使数组元素的值等于它们所对应索引的平方值。当然，可以使用取下标的方式来输出数组的值：

```
for(size_t i{} ; i < len ; ++i)
std::cout << pnumbers[i] << std::endl;
```

不能以传值的方式将一个 unique_ptr<T>对象传入函数中，因为它们不支持拷贝，必须使用引用的方式。unique_ptr<T>可以作为一个函数的返回值，因为它们不会被拷贝，但是它们必须以隐式移动运算的方式返回。

因为 unique_ptr<T>不能被拷贝，所以只能通过移动或生成它们的方式，在容器中存放 unique_ptr<T>对象。绝不会有两个 unique_ptr<T>对象指向同一个对象，shared_ptr<T>对象就没有这种特性，所以当需要多个指针指向一个对象时，或者当需要拷贝一个存放了智能指针的容器时，可以使用 shared_ptr<T>；否则，就要使用 unique_ptr<T>。在一个存放 unique_ptr<T>类型元素的容器中，可能会需要一个指向可用对象的原生指针。下面的代码演示了如何从 unique_ptr<T>获取一个原生指针：

```
auto unique_p = std::make_unique<std::string>(6, '*');
std::string pstr {unique_p.get()};
```

类的 get()成员函数可以返回一个 unique_ptr<T>所包含的原生指针。当需要访问一个对象时，指向它的智能指针被封装在一个类对象中。这时，就需要像上面那样做，这是一个典型的使用场景。因为 unique_ptr<T>不能被拷贝，所以不能返回 unique_ptr<T>。

1. 重置 unique_ptr<T>对象

当智能指针析构时，unique_ptr<T>对象所指向的对象也会被析构。调用一个无参 unique_ptr<T>对象的 reset()函数可以析构它所指向的对象，unique_ptr<T>对象中的原生指针将会被替换为空指针，这使你能够在任何时候析构智能指针所指向的对象。例如：

```
auto pname = std::make_unique<std::string>("Algernon");
...
pname.reset();                          // Release memory for string object
```

也可以将一个新生成的 T 对象的地址值传给 reset()函数。智能指针之前所指向的对象会被析构。然后它的地址值被替换为新对象的地址值：

```
pname.reset(new std::string{"Fred"});
```

pname 之前所指向的字符串对象的内存将会被释放，在内存中生成一个新的字符串对象——"Fred"，然后其地址被 pname 保存。

■ 警告：不要将其他 unique_ptr<T>所指向的一个对象的地址值传给 reset()，或者去生成一个新的 unique_ptr<T>对象，这种代码也许能通过编译，但是肯定会让程序崩溃。第一个 unique_ptr<T>的析构会释放它所指向的对象的内存；第二个智能指针析构时，将会试图再次释放已经释放的内存。

可以调用智能指针的 release()函数去释放一个 unique_ptr<T>所指向的对象，这样就能够在不释放对象内存的情况下，将指向它的 unique_ptr<T>内部的原生指针设为空指针。例如：

```
auto up_name = std::make_unique<std::string>("Algernon");
std::unique_ptr<std::string> up_new_name{up_name.release()};
```

up_name的成员函数 release()可以得到一个包含"Algernon"的字符串对象的原生指针，所以在执行完第二行代码时，up_new_name 会指向原始的那个字符串对象——"Algernon"。从unique 指针转移到另一个指针时，也转移了所指向对象的所有关系。

可以通过交换两个 unique_ptr<T>指针的方式来交换两个对象：

```
auto pn1 = std::make_unique<std::string>("Jack");
auto pn2 = std::make_unique<std::string>("Jill");
pn1.swap(pn2);
```

在执行完第二行代码后，pn1 会指向字符串"Jill"，pn2 指向字符串"Jack"。

3. 比较和检查 unique_ptr<T>对象

STL中非成员函数的模板函数定义了全套的比较运符，用来比较两个 unique_ptr<T>对象或者比较一个 unique_ptr<T>对象和一个空指针(nullptr)。比较两个 unique_ptr<T>对象也就是比较它们两个的成员函数 get()返回的地址值，比较一个 unique_ptr<T>对象和空指针也就是将智能指针的成员函数 get()返回的地址和空指针作比较。

unique_ptr<T>可以隐式地转换为布尔值。如果一个对象包含一个空指针，将会被转换为 false；否则转换为 true。这就意味着可以使用 if 语句来检查一个非空的 unique_ptr<T>对象：

```
auto up_name = std::make_unique<std::string>("Algernon");
std::unique_ptr<std::string> up_new{up_name.release()};
if(up_new)                                  // true if not nullptr
  std::cout << "The name is " << *up_new << std::endl;
if(!up_name)                                // true if nullptr
  std::cout << "The unique pointer is nullptr" << std::endl;
```

当对一个 unique_ptr()指针对象调用 reset()或 release()时，需要先做这种检查，因为在解引用一个指针时，需要保证它是一个非空的 unique_ptr<T>指针。

1.6.2 使用 shared_ptr<T>指针

可以按如下方式定义一个 shared_ptr<T>对象：

```
std::shared_ptr<double> pdata {new double{999.0}};
```

可以通过解引用一个 shared_ptr 指针，使用它所指向的对象或者修改对象的值：

```
*pdata = 8888.0;
std::cout << *pdata << std::endl;       // Outputs 8888.0
*pdata = 8889.0;
std::cout << *pdata << std::endl;       // Outputs 8889.0
```

pdata 的定义过程，包含了一个 double 变量的堆内存的分配，以及另一个和智能指针相关的控制块的内存分配，这个控制块记录了智能指针的副本计数。分配堆内存是相当耗时的，可以通过使用一个定义在 memory 头文件中的 make_shared<T>()函数来得到一个

shared_ptr<>对象，这个过程要有效率得多：

```
auto pdata = std::make_shared<double>(999.0); // Points to a double variable
```

尖括号中的数据类型指定了生成的变量的类型，函数名后圆括号中间的参数用来初始化生成的变量。

一般来说，make_shared<T>()函数的参数可以是任意数字，实际生成的数字取决于要生成的对象的类型。当需要使用 make_shared<T>()函数在堆上生成一个对象时，如果构造函数有要求的话，可以提供两个或更多个参数，这些参数用逗号隔开。auto 关键字可以使 pdata 的数据类型自动地从 make_shared<T>()返回的对象推导出来，所以事实上它是 shared_ptr<double>类型。尽管如此，当使用 auto 时，不要使用初始化列表，因为参数的类型会被推导为 std::initializer_list。

可以使用一个定义好的指针来初始化另一个 shared_ptr<T>指针：

```
std::shared_ptr<double> pdata2 {pdata};
```

pdata2 和 pdata 指向同一个变量，这会导致引用计数的增加。也可以将一个 shared_ptr<T>指针赋值给另一个 shared_ptr<T>指针：

```
std::shared_ptr<double> pdata{ new double{ 999.0 } };
std::shared_ptr<double> pdata2;          // Pointer contains nullptr
pdata2 = pdata;                          // Copy pointer - both point to the
                                         // same variable
std::cout << *pdata << std::endl;        // Outputs 999.0
```

当然，复制 pdata 会导致引用计数的增加。如果 double 变量使用的内存被释放，那么所有指向它的指针都需要被重置或析构。不能用 shared_ptr<T>指针来保存一个默认生成的数组，但是可以用来保存那些由你自己生成的 array<T>或 vector<T>容器对象。

■ 注意：你可能会创建一个指向数组的 shared_ptr<T>指针，这需要为智能指针提供一个 deleter 函数去释放数组的堆内存，具体怎么做超出了本书的讨论范围。

---

shared_ptr<T>和 unique_ptr<T>的使用方式相似，通过使用 shared_ptr<T>的成员函数 get()，可以从 shared_ptr<T>得到一个原生指针。pdata 可以像先前的章节那样去定义。可以这样写：

```
auto pvalue = pdata.get(); // pvalue is type double* and points to 999.0
```

当需要使用一个原生指针时，可以像上面那样去使用。

■ 警告：只能通过拷贝构造函数或赋值运算符去复制一个 shared_ptr<T>对象。通过一个由其他指针的 get()函数返回的原生指针，来生成一个 shared_ptr<T>指针。这可能会导致一些意想不到的问题，大多数情况也就意味着程序的崩溃。

---

1. 重置 shared_ptr<T>对象

如果将一个空指针赋给一个 shared_ptr<T>对象，那么它的地址值将会变为空，同样也

会使指针所指向对象的引用计数减 1。例如：

```
auto pname = std::make_shared<std::string>("Charles Dickens");
                                        // Points to a string object
// ... lots of other stuff happening...
pname = nullptr;                        // Reset pname to nullptr
```

以"Charles Dickens"为初始值，生成一个在堆上存放的字符串对象，然后生成一个共享指针来保存这个地址值。最后，用空指针来替换 pname 所保存的值。当然，这时候任何其他保存了字符串对象地址的 shared_ptr<T>对象仍然存在，因为引用计数还不等于 0。

通过调用 shared_ptr<T>对象的无参 reset()函数，可以得到同样的效果：

```
pname.reset();                          // Reset to nullptr
```

也可以通过为 reset()函数传入一个原生指针来改变共享指针指向的对象。例如：

```
pname.reset(new std::string{"Jane Austen"}); // pname points to new string
```

通过 reset()的参数传入的地址对象，类型必须和之前存放在智能指针中的对象类型一致，或者必须可以隐式转换成之前的类型。

2. 比较和检查 shared_ptr<T>对象

可以使用任意比较运算符来比较两个 shared_ptr<T> 保存的地址，或者将一个 shared_ptr<T>保存的地址和空指针比较。最有用的比较运算符是=和!=，这能够让我们知道两个指针是否指向同一个对象。给定两个 shared_ptr<T>对象 pA 和 pB，它们指向同一类型，你能够像下面这样比较它们：

```
if((pA == pB) && (pA != nullptr))
   std::cout << " Both pointers point to the same object.\n";
```

这两个指针可能相等或者都是空指针，所以这样一种简单的比较并不能充分说明它们都指向同一个元素。像 unique_ptr<T>指针对象和 shared_ptr<T>指针对象，都可能被隐式转换成布尔值，所以不能像下面这样使用：

```
if(pA && (pA == pB))
   std::cout << " Both pointers point to the same object.\n";
```

也可以检查 shared_ptr<T>对象是否有任何副本：

```
auto pname = std::make_shared<std::string>("Charles Dickens");
if(pname.unique())
   std::cout << there is only one..." << std::endl;
else
   std::cout << there is more than one..." << std::endl;
```

如果对象的实例数是 1，unique()成员函数返回 true，否则返回 false。也可以知道当前有多个实例：

```
if(pname.unique())
```

```
    std::cout << there is only one..." << std::endl;
  else
    std::cout << there are " << pname.use_count() << " instances." << std::endl;
```

成员函数 use_count()返回当前被调用对象的实例个数。如果 shared_ptr<T>对象包含的为空指针，那么返回 0。

1.6.3　weak_ptr<T>指针

weak_ptr<T>对象只能从 shared_ptr<T>对象创建。weak_ptr<T>指针同样被当作类的成员变量去存储同一个类的其他实例对象。当这个类的对象是在堆上创建时，用它的 shared_ptr<T>成员变量去指向另一个同类型的对象。在这种情况下，可能会生成一个循环引用，这使另一个同类型的类对象不能自动释放空间，这种情况可能并不常见，但却是有可能的，如图 1-4 所示。

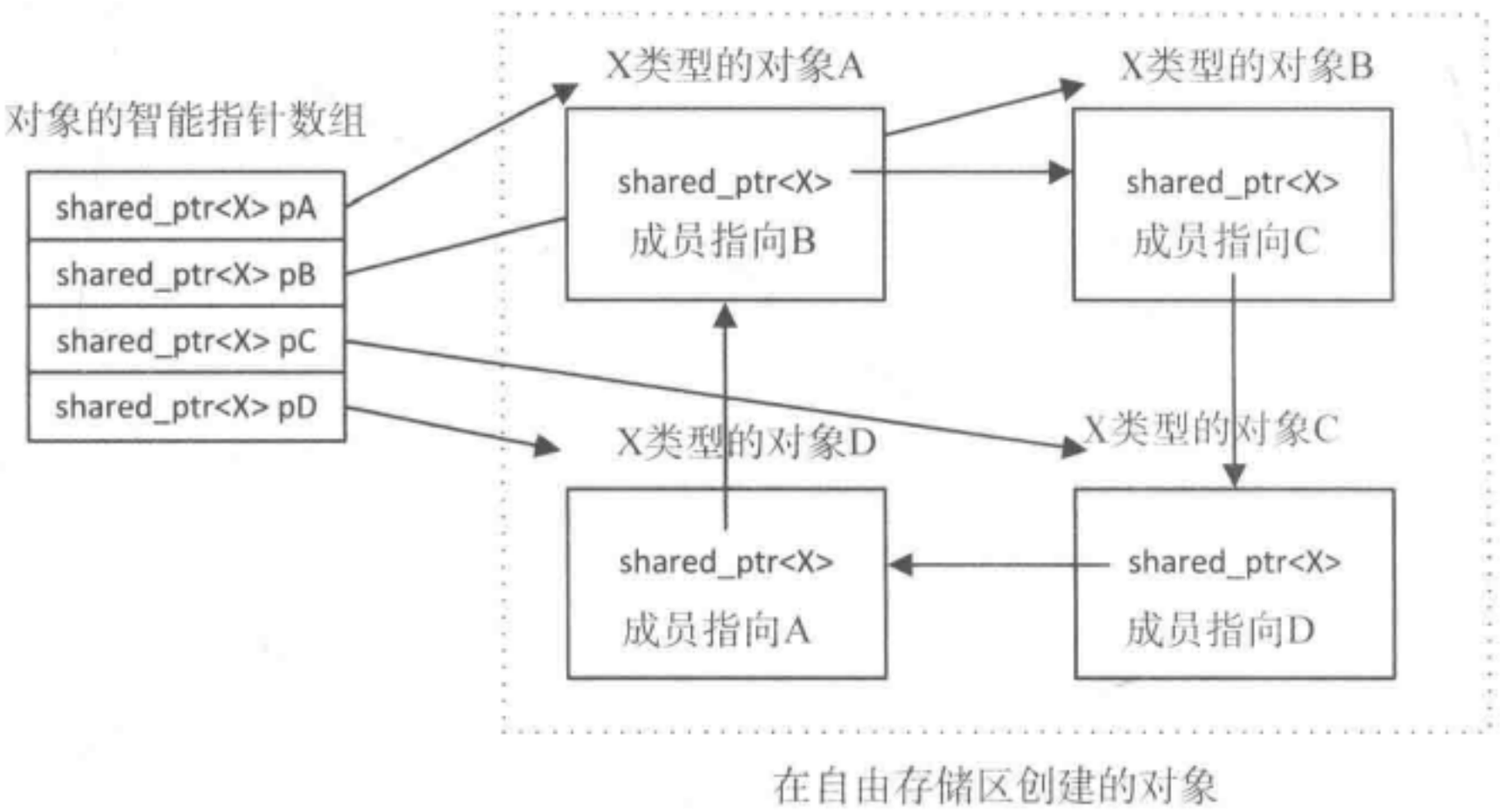


图 1-4　循环引用是如何阻止对象被删除的

删除图 1-4 所示数组中所有的智能指针或者把它们重置为空指针并不能释放它们所指向对象的内存，这是因为每个对象仍有一个 shared_ptr<X>指针指向它们，而且也没有外部的指针能够访问到它们，所以它们无法被销毁。如果对象使用 weak_ptr<X>作为成员变量来指向其他的对象，就可以解决这个问题。当外部数组中的指针被销毁或重置时，weak_ptr<X>就不会阻止对象的销毁。

可以像下面这样去创建一个 weak_ptr<T>对象：

```
auto pData = std::make_shared<X>();
                // Create a shared pointer to an object of type X
std::weak_ptr<X> pwData {pData};
                // Create a weak pointer from shared pointer
std::weak_ptr<X> pwData2 {pwData};
                // Create a weak pointer from another
```

因此，可以从一个 shared_ptr<T>或一个已有的 weak_ptr<T>去创建一个 weak_ptr<T>。weak_ptr<T>能做的事情有限，不能以解引用的方式去访问它所指向的对象。例如，可使用

weak_ptr<T>对象去做下面两件事：

- 可以判断它所指向的对象是否仍然存在，这也就意味着仍然有 shared_ptr<T>对象指向它。
- 可以从一个 weak_ptr<T>对象创建一个 shared_ptr<T>对象。

这里演示了如何使用 weak_ptr<T>指针来判断它所指向的对象是否还存在：

```
if(pwData.expired())
   std::cout << "Object no longer exists.\n";
```

如果对象不存在的话，pwData 对象的 expired()函数会返回 true。可以像下面这样从一个 weak_ptr<T>对象得到一个 shared_ptr<T>对象：

```
std::shared_ptr<X> pNew {pwData.lock()};
```

如果 pwData 指向的对象仍然存在，lock()函数会通过返回一个新的 shared_ptr<X>来初始化 pNew，锁住对象。如果对象不存在，lock()函数会返回一个具有空指针的 shared_ptr<X>对象。可以在 if 语句中检查这个结果：

```
if(pNew)
   std::cout << "Shared pointer to object created.\n";
else
   std::cout << "Object no longer exists.\n";
```

如何使用 weak_ptr<T>并不在本书的讨论范围中，所以不打算继续深入讨论 weak_ptr<T>。第 3 章会探讨存储智能指针的含义和优点。

1.7 算法

算法提供一些计算和分析的函数，主要运用于一些由一对迭代器指定的对象。起始迭代器指向第一个元素，结束迭代器指向最后一个元素的下一个位置。因为它们可以通过迭代器访问数据元素，所以算法并不关心数据在哪里。可以将算法运用到任何序列上，只要它们能提供特定的迭代器去访问，所以也能够将算法运用到一些容器的元素上，像字符串对象中的字符、标准数组元素、流、存放在类型容器中的序列，甚至任何元素，只要类支持迭代器。

算法是 STL 中最大的工具集。很多程序中都用到了它们，尽管它们的用途看起来可能比较专业。可以将它们分成三大类：

1) 非变化序列运算不会改变它们适用于任何方式的顺序。一个以指定值来匹配元素的算法不会改变元素的原始数据。数值运算(例如 inner_product()和 accumulate())处理元素时不会改变它们，就属于这一类运算。这种类型的算法包括 find()、count()、mismatch()、search()、equal()。

2) 可变序列运算会改变序列中元素的值。这种类型的算法包括 swap()、copy()、transform()、replace()、remove()、reverse()、rotate()、fill()、shuffle()。堆运算也属于这一类。

3) 很多实例中的排序、合并、关联运算，在使用时会改变序列的顺序。这种类型的算

法包括 sort()、stable_sort()、binary_search()、merge()、min()、max()。

当然，这里列出的每个种类的算法，是没有详尽的可用清单的；你在随后的章节中会了解到更多，以及如何运用它们。一些算法，例如 transform()，需要一个函数作为实参，运用到一段元素上。其他对元素重新排序的函数，经常提供一个判断去比较元素。下面让我们看一下把一个函数传给另一个函数作为参数的可能性。

1.8　将函数作为实参传入

函数的签名能不能作为另一个函数的实参使用是由这个函数的形参规格决定的。形参的规格取决于函数实参的类型。可以通过三种方式把一个函数作为实参传给另一个函数：

- 可以使用一个函数指针，这里就可以使用函数名作为参数值。这一点不会详述，因为你肯定已经熟悉了函数指针。下面是另外两种更好的方式：
- 可以传入一个函数对象作为实参。
- 可以使用一个 lambda 表达式作为实参。

在随后的章节中，你会看到很多充分运用了最后两种方式的示例，所以本书会提醒你一些这方面的细节，确保你不会对它们感到生疏。

1.8.1　函数对象

函数对象也被称作仿函数，这是一种重载了函数调用运算符 operator()()的类对象。它们提供了一种比使用原生指针更加高效的、将函数作为实参传入另一个函数的方式。让我们来看一些简单的示例，假设像下面这样定义了 Volume 类。

```
class Volume
{
public:
   double operator()(double x, double y, double z) {return x*y*z; }
};
```

可以创建一个能像函数一样使用的 Volume 对象来计算容量：

```
Volume volume;                              // Create a functor
double room { volume(16, 12, 8.5) }; // Room volume in cubic feet
```

room 的初始化列表中的值，是通过调用 volume 对象的 operator()()来获取的，所以这个表达式等同于 volume.operator()(16,12,8.5)。当一个函数接收一个函数对象作为实参时，它就可以像函数一样使用。当然，可以在类中定义不止一个版本的 operator()()，这允许一个对象以各种方式应用。假设我们定义了一个 Box 类，定义了一些成员变量来定义对象的长、宽、高，以及一些可以获取到这些值的成员函数。我们可以像下面这样扩展 Volume 类，使它容纳 Box 对象：

```
class Volume
{
```

```
public:
    double operator()(double x, double y, double z) {return x*y*z; }

    double operator()(const Box& box)
    { return box.getLength()*box.getWidth()*box.getHeight(); }
};
```

现在 Volume 对象能够计算 Box 对象的体积了：

```
Box box{1.0, 2.0, 3.0};
std::cout << "The volume of the box is " << volume(box) << std::endl;
```

为了允许将 Volume 对象作为实参传给函数，可以将函数的形参类型指定为 Volume&。STL 算法通常对参数使用一个更广义的规范，要求形参能够通过一个定义了这种类型的函数模板列表来表示函数。

1.8.2 lambda 表达式

一个 lambda 表达式定义了一个匿名函数。lambda 表达式定义的函数不同于一般函数的地方是，lambda 可以捕获它们作用域内的变量，然后使用它们。lambda 表达式通常用于 STL 算法。让我们看一个 lambda 表达式示例。假设你想赋予一个函数计算 double 类型值三次幂的功能。下面介绍如何使用 lambda 来实现：

```
[] (double value) { return value*value*value; }
```

这一对方括号称为 lambda 引入符。它们标识了 lambda 表达式的开始，不仅仅只有这里介绍的这种 lambda 引入符——方括号并不总是为空。lambda 引入符后面圆括号之间的是 lambda 的参数列表，这和一般函数的参数列表相似。在本例中，只有一个参数 value，当然也可以有更多的参数，用逗号隔开它们。可以为 lambda 表达式中的参数指定默认值。

lambda 表达式的主体部分在参数列表后的花括号中，这更像普通函数。lambda 的主体包含一条 return 语句，用来返回计算值。一般来说，lambda 主体可以包含任意数量的语句，需要注意的是，在上面的示例中并没有具体的返回类型，返回类型默认为返回值的类型。如果没有返回值，返回类型就是 void。可以使用尾返回类型语法(trailing return type syntax)来指定返回类型。上面的 lambda 表达式可以这样写：

```
[] (double value) -> double { return value*value*value; }
```

1. 对 lambda 表达式命名

尽管 lambda 表达式是匿名对象，但仍然可以用一个变量来保存它的地址。不需要知道它的类型，但编译器需要知道：

```
auto cube = [] (double value) { return value*value*value; };
```

auto 关键字告诉编译器从赋值运算符的右边来确定变量的类型，所以能够知道确定的类型来存放 lambda 表达式的地址。如果 lambda 表达式的引入符方括号中什么都没有，就可以这样做，有时候方括号中的内容会使你不能够以这种方式使用 auto。可以把 cube 当作

函数指针来使用，例如：

```
double x{2.5};
std::cout << x << " cubed is " << cube(x) << std::endl;
```

输入语句会输出 2.5 的立方值。

2. 将 lambda 表达式传给函数

通常，你并不知道一个 lambda 表达式的类型，因为没有通用的表达式类型。前面说过，可以把一个 lambda 表达式作为函数的实参，但这马上会产生一个疑问：如果以一个 lambda 表达式作为实参，那么如何确定形参的类型。这有很多的可能，简单的回答是定义一个函数的模板，它的类型参数是 lambda 表达式的类型。

编译器总是知道 lambda 表达式的类型，所以它们能实例化一个接受 lambda 表达式作为参数的函数模板。通过一个示例，我们很容易知道它们的工作原理。假设有一些 double 类型的值存放在容器中，想以任意方式转换它们：有时候想以它们的平方值代替它们，或者以它们的平方根，或者执行更复杂的转换——这取决于它们的值是否在某个特定的范围内。可以定义一个模板来转换一些由 lambda 表达式指定的元素。

下面看看这个模板的定义：

```
template <typename ForwardIter, typename F>
void change(ForwardIter first, ForwardIter last, F fun)
{
   for(auto iter = first; iter != last; ++iter)
                                    // For each element in the range...
     *iter = fun(*iter);            // ...apply the function to the object
}
```

形参 fun 接受任何适合的 lambda 表达式，也接受函数对象或普通的函数指针。你可能会好奇编译器是如何处理模板的，因为没有任何关于 fun 实现的信息，事实上是因为编译器根本没有处理它。编译器知道需要实例化一个模板时才会处理它。在上面模板的示例中，当使用它时，编译器知道关于这个 lambda 表达式的所有信息。这里有一个使用这个模板的示例：

```
int data[] {1, 2, 3, 4};
change(std::begin(data), std::end(data), [] (int value){ return
value*value; });
```

第二条语句会将 data 数组的元素全部替换为它们的平方值。

标准库的 functional 头文件定义了一个模板类型 std::function<>，这是对任意类型函数指针的封装，有给定的返回类型和形参类型。当然，这也包括 lambda 表达式。std::function 模板的类型实参是 Return_Type(Param_Types)的形式。Return_Type 是 lambda 表达式或被指向函数的返回值类型。Param_Types 是 lambda 表达式或被指向函数的参数类型的列表，它们都用逗号隔开。在前面章节中，表示 lambda 表达式的变量可以按如下方式定义：

```
    std::function<double(double)> op { [] (double value) { return
value*value*value; } };
```

op 变量可以作为实参，传给任意接受函数实参且具有相同签名的函数。当然，也可以重新定义 op，让它们实现一些其他功能，只要它们具有相同的返回类型、形参个数和形参类型即可：

```
    op = [] (double value) { return value*value; };
```

现在，op 表示一个能够返回实参的平方值的函数。可以用 std::function 类型模板来指定任何可以调用的类型，包括任何 lambda 表达式或函数对象。

3. 捕获子句

lambda 引入符[]并不一定为空。它可以包含捕获子句，指定一些封闭范围内的变量，它们可以在 lambda 主体中使用。如果一个 lambda 表达式的方括号中没有任何参数，lambda 的主体部分就只能使用 lambda 内的一些局部变量。没有捕获子句的 lambda 表达式被称为无状态的 lambda 表达式，因为它们不能访问封闭范围内的任何东西。图 1-5 显示了 lambda 的语法结构。

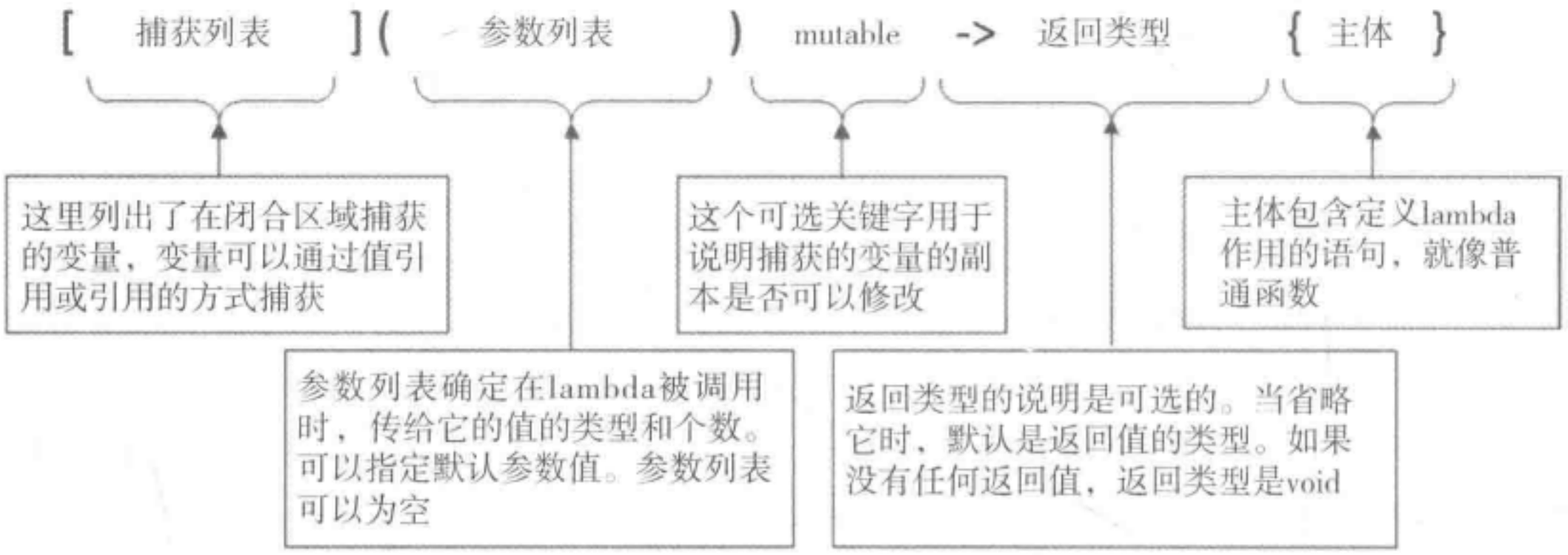


图 1-5　lambda 表达式的组成

默认的捕获子句捕获和 lambda 表达式定义同样范围内的所有变量。如果在方括号中放一个=，lambda 表达式的主体就能以值的形式访问所有封闭范围内的自动变量。也就是说，在 lambda 表达式里可以使用所有变量，但是不能改变它们的值。如果在包围了参数列表的括号后面添加了 mutable 关键字，就可以在 lambda 表达式中改变变量副本的值。使用 lambda 表达式时，仍然会保存着上一次执行时，从外部以值的方式捕获的变量的副本，所以副本是有效的静态值。

如果在方括号中添加了一个&，所有封闭范围内的变量都以引用的方式使用，所以它们的值能在 lambda 体中改变，这时候就不需要 mutable 关键字了。为了能够在 lambda 中使用外部变量，外部变量必须定义在 lambda 表达式之前。

不能用 auto 指定一个变量的类型，然后保存一个访问这个变量本身的 lambda 地址，这意味着正在用一个使用这个变量的 lambda 表达式去初始化这个变量。不能将 auto 用于任何使用了正在定义的变量的 lambda 表达式——auto 不能用于自引用。

可以捕获闭合范围内指定的变量。为了捕获那些想用值访问的指定变量，只需要在捕获子句中列出它们的名称。如果想以引用的方式捕获，需要在每一个名字的前面加上一个&前缀。捕获子句中两个以上的变量需要使用逗号隔开。可以在一个通过引用来捕获特定变量的捕获子句中，同时使用=。捕获子句[=,&factor]，允许我们以引用的方式使用 factor，以值引用的方式使用其他任何封闭范围内的变量。捕获子句[&,factor]，允许我们以值引用的方式捕获 factor，以引用的方式来捕获任何其他封闭范围内的变量。如果要修改 factor 副本的值，需要为它指定一个 mutable 关键字。

■ 警告：以值引用的方式捕获封闭范围内的所有变量会增加很多开销。因为不管是否使用了它们，都为它们中的每一个创建了副本，只捕获那些需要使用的变量才是明智的。

transform()算法将你提供的作为参数的函数运用到一定范围的元素上。transform()的前两个参数是用来指定函数作用域的迭代器；第 3 个参数是一个指定结果存放起始位置的迭代器；第 4 个参数是运用到输入序列的函数。下面这个示例演示了如何和 transform()算法一起使用函数、lambda 表达式和 std::function 模板类型：

```
// Ex1_03.cpp
// Passing functions to an algorithm
#include <iostream>                          // For standard streams
#include <algorithm>                         // For transform()
#include <iterator>                          // For iterators
#include <functional>                        // For function

class Root
{
public:
  double operator()(double x) { return std::sqrt(x); };
};
int main()
{
  double data[] { 1.5, 2.5, 3.5, 4.5, 5.5};

  // Passing a function object
  Root root; // Function object
  std::cout << "Square roots are:" << std::endl;
  std::transform(std::begin(data), std::end(data),
                 std::ostream_iterator<double>(std::cout, " "), root);

  // Using an lambda expression as an argument
  std::cout << "\n\nCubes are:" << std::endl;
  std::transform(std::begin(data), std::end(data),
                 std::ostream_iterator<double>(std::cout, " "),
                 [](double x){return x*x*x; });

  // Using a variable of type std::function<> as argument
  std::function<double(double)> op {[](double x){ return x*x; }};
  std::cout << "\n\nSquares are:" << std::endl;
  std::transform(std::begin(data), std::end(data),
                 std::ostream_iterator<double>(std::cout, " "), op);
```

```
    // Using a lambda expression that calls another lambda expression as
       argument
    std::cout << "\n\n4th powers are:" << std::endl;
    std::transform(std::begin(data), std::end(data),
                   std::ostream_iterator<double>(std::cout, " "),
                   [&op](double x){return op(x)*op(x); });

    std::cout << std::endl;
}
```

输出结果如下：

```
Square roots are:
1.22474 1.58114 1.87083 2.12132 2.34521

Cubes are:
3.375 15.625 42.875 91.125 166.375

Squares are:
3.375 15.625 42.875 91.125 166.375

4th powers are:
11.3906 244.141 1838.27 8303.77 27680.6
```

如果理解了先前的章节，对于本例应该不会觉得很难。transform()算法处理的输入数据来自一个 data 数组。在每一个transform()调用中，前两个参数都是数组的开始和结束迭代器。输出目的地是由一个输出流迭代器指定的，它将数据写入标准输出流。ostream_iterator 构造函数的第二个参数是一个用来分隔每个写入值的分隔符。

第 1 个 transform()调用传入一个 Root 对象作为最后一个参数。Root 类定义 operator()() 成员函数来返回 root 参数的开方。第 2 个 transform()调用表明可以写一个用来计算立方值的 lambda 表达式作为参数。第 3 个 transform()调用表明 std::function 类型的模板也可以在这里使用。最后一个调用表明一个 lambda 表达式可以调用另一个 lambda 表达式。所以，当需要把函数作为参数传给算法时，可以使用这些技巧中的任意一个。

1.9 小结

这一章介绍了 STL 背后的一些基本思想。在本书后面的章节中，对于在本章介绍的 STL 的所有方面，都会做更深入的示范和解释。本章还概述了一些的需要理解的 C++重要特性，因为它们是应用 STL 的基础，在随后的章节中会广泛地使用它们。本章所涉及的要点如下：

- STL 定义类模板，作为其他对象的容器。
- STL 定义迭代器——一种行为像指针的对象。通常用一对迭代器定义一段连续的元素。起始迭代器用于指向第一个元素，结束迭代器用于指向最后一个元素的下一个位置。

- 反向起始迭代器指向序列的最后一个元素，反向结束迭代器指向第一个元素的前一个位置。反向迭代器的工作意义和普通迭代器相反。
- iterator 头文件定义了一些全局函数，可以从容器、数组、任何支持迭代器的对象得到一些迭代器。全局函数 begin()、cbegin()、end()、cend()都能够返回普通迭代器。函数 rbegin()、crbegin()、rend()｜crend()能够返回反向迭代器。
- 可以使用流迭代器转换流对象中给定类型的数据。
- STL 定义了一些实现了算法的函数模板，可以运用到由迭代器指定的一段元素上。
- 智能指针是一种表现有些像指针的对象。对象通常是在堆上分配地址。对象由智能指针管理，当没有智能指针指向对象时，对象会被自动销毁。智能指针不能自增或自减。
- 一个 lambda 表达式定义了一个匿名函数。lambda 表达式通常用来把函数作为参数传给 STL 算法。
- 能够用定义在 functional 头文件中的 std::function<>模板类型，去指定任意种类的、有给定函数签名的可调用实体。

练习

这里有一些检验你是否记住了本章所讨论主题的练习。如果遇到困难，可以回顾前面的章节以寻找帮助。在这之后，如果还是无法解决的话，可以从 Apress 出版社的网站上下载解答(http://www.apress.com/9781484200056)，但这应该是最后的选择。

1. 编写一个程序，它定义了一个标准的数组：用你选择的一些词去初始化字符串对象，然后用迭代器一个一行地列出数组的内容。

2. 写一个程序，运用 transform()算法，将前一个练习中数组的所有小写元音字母以'*'替换，并将结果一个一行地写入标准输出流。以 lambda 表达式的方式定义一个使用迭代器的函数来替换字符串中的元音字母。

3. 写一个程序，运用 transform()算法，将第一个练习中的数组的字符串转换为大写，然后输出结果。用来转换字符串的函数，应该以 lambda 表达式的形式传给 transform()，这个 lambda 表达式会调用 transform()，来把 std::toupper()库函数运用到字符串中的每一个字符上。

第2章

使用序列容器

本章将介绍一类你会经常用到的容器——序列容器。本章将介绍以下内容：

- 序列容器的特性
- 如何获取和使用包含序列容器的迭代器
- 如何使用数组容器
- vector 容器具备哪些功能
- 双向队列容器的特性和功能，以及它们和 vector 容器的差别
- list 容器如何组织它所存储的数据元素，以及它的优缺点
- forward_list 容器和 list 容器有什么不同，何时该使用它
- 如何自定义迭代器

2.1 序列容器

序列容器以线性序列的方式存储元素。它没有对元素进行排序，元素的顺序和存储它们的顺序相同。以下有 5 种标准的序列容器，每种容器都具有不同的特性：

- array<T,N>(数组容器)是一个长度固定的序列，有 N 个 T 类型的对象，不能增加或删除元素。
- vector<T>(向量容器)是一个长度可变的序列，用来存放 T 类型的对象。必要时，可以自动增加容量，但只能在序列的末尾高效地增加或删除元素。
- deque<T>(双向队列容器)是一个长度可变的、可以自动增长的序列，在序列的两端都不能高效地增加或删除元素。
- list<T>(链表容器)是一个长度可变的、由 T 类型对象组成的序列，它以双向链表的形式组织元素，在这个序列的任何地方都可以高效地增加或删除元素。访问容器中任意元素的速度要比前三种容器慢，这是因为 list<T>必须从第一个元素或最后一个元素开始访问，需要沿着链表移动，直到到达想要的元素。

- forward_list<T>(正向链表容器)是一个长度可变的、由 T 类型对象组成的序列，它以单链表的形式组织元素，是一类比链表容器快、更节省内存的容器，但是它内部的元素只能从第一个元素开始访问。

图 2-1 说明了可供使用的序列容器以及它们之间的区别。

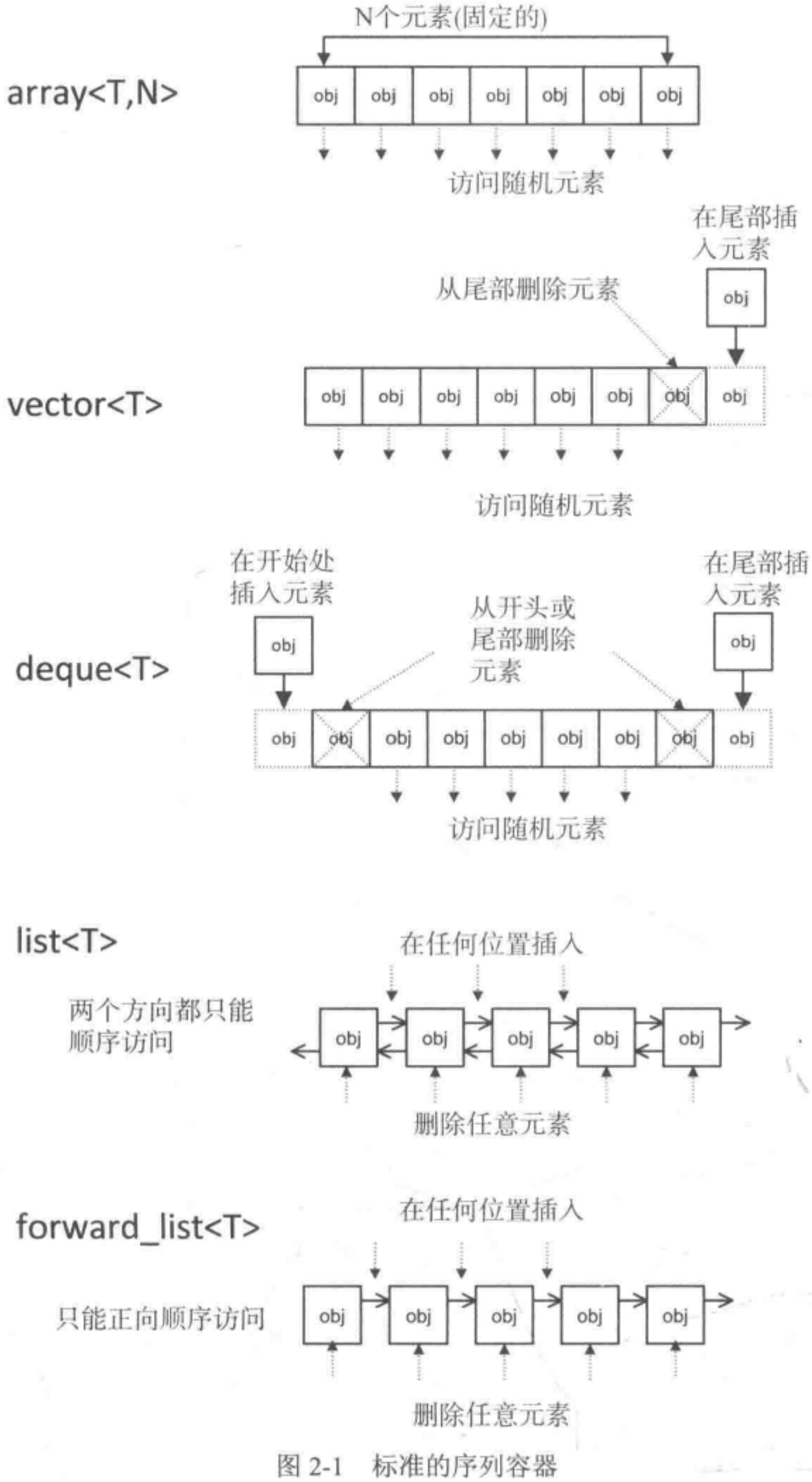


图 2-1　标准的序列容器

图 2-1 中每种类型容器的操作都可以高效执行，但其他示例中的操作可能会慢一些。

容器中常见的函数成员

在本章的剩余部分，会详细介绍每一类序列容器的用法。序列容器包含一些相同的成员函数，它们的功能也相同。本书会在某个容器的上下文中详细介绍下面的每个函数，但对于每种类型的容器不会重复介绍它们的细节。表 2-1 展示了 array、vector 和 deque 容器的函数成员，它们中至少有两个容器实现了同样的函数成员。

表 2-1　array、vector 和 deque 容器的函数成员

函数成员	array<T,N>	vector<T>	deque<T>
begin()-返回开始迭代器	是	是	是
end()-返回结束迭代器	是	是	是
rbegin()-返回反向开始迭代器	是	是	是
rend()-返回反向结束迭代器	是	是	是
cbegin()-返回 const 开始迭代器	是	是	是
cend()-返回 const 结束迭代器	是	是	是
crbegin()-返回 const 反向开始迭代器	是	是	是
crend()-返回 const 反向结束迭代器	是	是	是
assign()-用新元素替换原有内容	-	是	是
operator=()-复制同类型容器的元素，或者用初始化列表替换现有内容	是	是	是
size()-返回实际元素个数	是	是	是
max_size()-返回元素个数的最大值	是	是	是
capacity()-返回当前容量	-	是	-
empty()-返回 true，如果容器中没有元素的话	是	是	是
resize()-改变实际元素的个数	-	是	是
shrink_to_fit()-将内存减少到等于当前元素实际所使用的大小	-	是	是
front()-返回第一个元素的引用	是	是	是
back()-返回最后一个元素的引用	是	是	是
operator[]()-使用索引访问元素	是	是	是
at()-使用经过边界检查的索引访问元素	是	是	是
push_back()-在序列的尾部添加一个元素	-	是	是
insert()-在指定的位置插入一个或多个元素	-	是	是
emplace()-在指定的位置直接生成一个元素	-	是	是
emplace_back()-在序列尾部生成一个元素	-	是	是
pop_back()-移出序列尾部的元素	-	是	是
erase()-移出一个元素或一段元素	-	是	是

(续表)

函数成员	array<T,N>	vector<T>	deque<T>
clear()-移出所有的元素，容器大小变为 0	-	是	是
swap()-交换两个容器的所有元素	是	是	是
data()-返回包含元素的内部数组的指针	是	是	-

缺少“是”的列表明，对应的容器并没有定义这个函数。没有必要记住这张表，该表仅供参考。在深入了解到容器是如何组织元素以后，你会本能地知道哪个容器的哪些成员函数能使用。

在表 2-1 中，对于以链表来组织元素的容器，它们的内部组织方式有很大差别。list 和 forward_list 容器彼此非常相似。forward_list 中包含了 list 的大部分成员函数，forward_list 中未包含那些需要反向遍历的函数，所以示例中没有反向迭代器。作为参考，表 2-2 展示了 list 和 forward_list 的函数成员。

表 2-2　list 和 forward_list 的函数成员

函数成员	list<T >	forward_list<T >
begin()-返回开始迭代器	是	是
end()-返回结束迭代器	是	是
rbegin()-返回反向开始迭代器	是	-
rend()-返回反向结束迭代器	是	-
cbegin()-返回 const 开始结束迭代器	是	是
before_begin()-返回一个指向第一个元素前一个位置的迭代器	-	是
cbefore_begin()-返回一个指向第一个元素前一个位置的 const 迭代器	-	是
cend()-返回 const 结束迭代器	是	是
crbegin()-返回 const 反向开始迭代器	是	-
crend()-返回 const 反向结束迭代器	是	-
assign()-用新元素替换原有内容	是	是
operator=()-复制同类型容器的元素，或者用初始化列表替换现有内容	是	是
size()-返回实际元素个数	是	-
max_size()-返回元素个数的最大值	是	是
resize()-改变实际元素的个数	是	是
empty()-返回 true，如果容器中没有元素的话	是	是
front()-返回第一个元素的引用	是	是
back()-返回最后一个元素的引用	是	-
push_back()-在序列的尾部添加一个元素	是	-

(续表)

函数成员	list<T >	forward_list<T >
push_front()-在序列的起始位置添加一个元素	是	是
emplace()-在指定位置直接生成一个元素	是	-
emplace_after()-在指定位置的后面直接生成一个元素	-	是
emplace_back()-在序列尾部生成一个元素	是	-
emplace_front()-在序列的起始位置生成一个元素	是	是
insert()-在指定的位置插入一个或多个元素	是	-
insert_after()-在指定位置的后面插入一个或多个元素	-	是
pop_back()-移除序列尾部的元素	是	-
pop_front()-移除序列头部的元素	是	是
reverse()-反向元素的顺序	是	是
erase()-移除指定位置的一个元素或一段元素	是	-
erase_after()-移除指定位置后面的一个元素或一段元素	-	是
remove()-移除所有和参数匹配的元素	是	是
remove_if()-移除满足一元函数条件的所有元素	是	是
unique()-移除所有连续重复的元素	是	是
clear()-移除所有的元素，容器大小变为 0	是	是
swap()-交换两个容器的所有元素	是	是
sort()-对元素进行排序	是	是
merge()-合并两个有序容器	是	是
splice()-移动指定位置前面的所有元素到另一个同类型的 list 中	是	-
splice_after()-移动指定位置后面的所有元素到另一个同类型的 list 中	-	是

所有序列容器的函数成员 max_size()都会返回它能存储的元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$，所以我们很少会用到这个函数。

2.2　使用 array<T,N>容器

array<T,N>模板定义了一种相当于标准数组的容器类型。它是一个有 N 个 T 类型元素的固定序列。除了需要指定元素的类型和个数之外，它和常规数组没有太大的差别。显然，不能增加或删除元素。模板实例的元素被内部存储在标准数组中。和标准数组相比，array 容器的额外开销很小，但提供了两个优点：如果使用at()，当用一个非法的索引访问数组元素时，能够被检测到，因为容器知道它有多少个元素，这也就意味着数组容器可以作为参数传给函数，而不再需要单独去指定数组元素的个数。使用 array 容器类型时，需要在源文

件中包含头文件 array。数组容器非常好用，这里有一个示例，展示了如何创建具有 100 个 double 型元素的 array<>：

```
std::array<double, 100> data;
```

如果定义了一个 array 容器，却没有为元素指定初始值，那么元素就不会被初始化；但是可以如下所示，将它们初始化为 0 或者和默认元素类型等效的值：

```
std::array<double, 100> data {};
```

使用该语句初始化后，容器中所有的元素都会变为 0.0。形参 N 必须是一个常量表达式(constant expression)并且容器中元素的个数不能变。当然，当创建 array 容器的实例时，要像创建常规数组那样，对元素进行初始化：

```
std::array<double, 10> values {0.5, 1.0, 1.5, 2.0};
// 5th and subsequent elements are 0.0
```

初始化器列表中的 4 个值用于初始化前 4 个元素，其余的元素都将为 0。图 2-2 说明了这一点。

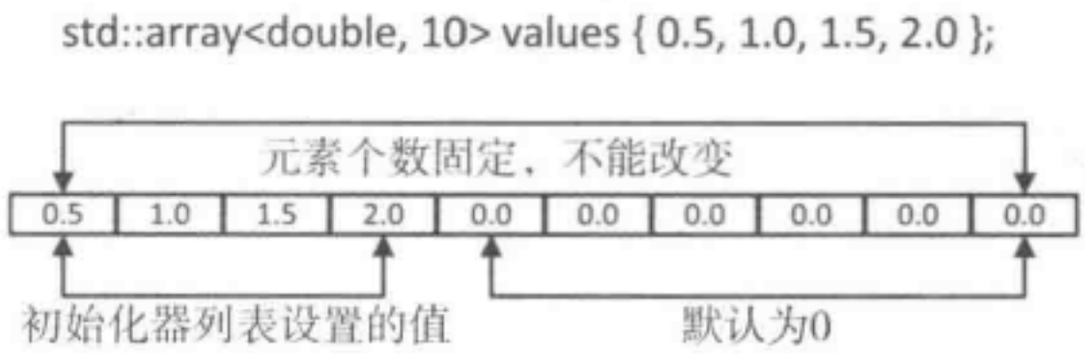


图 2-2　生成一个 array<T, N>容器

通过调用数组对象的成员函数 fill()，可以将所有元素设成给定值。例如：

```
values.fill(3.1415926);     // Set all elements to pi
```

fill()函数将所有元素都设为传入的实参值。

2.2.1　访问元素

可以通过在方括号中使用索引表达式访问和使用数组容器的元素，这和标准数组的访问方式相同，例如：

```
values[4] = values[3] + 2.0*values[1];
```

第 5 个元素的值被赋值为右边表达式的值。像这样使用索引时，因为没有做任何边界检查，所以，如果使用越界的索引值去访问或存储元素，就不会被检测到。为了能够检查越界索引值，应该使用成员函数 at()：

```
values.at(4) = values.at(3) + 2.0*values.at(1);
```

这和前一条语句的功能相同，除了当传给 at()的索引是一个越界值时，这时会抛出 std::out_of_rang 异常。应该总是使用 at()，除非确定索引没有越界。这也产生了一个疑问，

为什么 operator[]()的实现没有进行边界检查，答案是因为性能。如果每次访问元素，都去检查索引值，无疑会产生很多开销。当不存在越界访问的可能时，就能避免这种开销。

数组对象的 size()函数能够返回 size_t 类型的元素个数值，所以能够像下面这样去计算数组所有元素的和：

```
double total {};
for(size_t i {} ; i < values.size() ; ++i)
{
   total += values[i];
}
```

size()函数的存在，为数组容器提供了标准数组所没有的优势，数组元素能够知道它包含多少元素。接受数组容器作为参数的函数，只需要通过调用容器的成员函数 size()，就能得到元素的个数。不需要去调用 size()函数来判断一个数组容器是否为空。如果容器中没有元素的话，成员函数 empty()会返回 true：

```
if(values.empty())
   std::cout << "The container has no elements.\n";
else
   std::cout << "The container has " << values.size() << " elements.\n";
```

然而，我们很难想象数组容器没有元素的情形，因为当生成一个数组容器时，它的元素个数就固定了，而且无法改变。生成空数组容器的唯一方法是，将模板的第二个参数指定为 0，这种情况基本不可能发生。然而，对于其他元素可变或者元素可删除的容器来说，它们使用 empty()时的机制是一样的，因此为它们提供了一个一致性的操作。

对于任何可以使用迭代器的容器，都可以使用基于范围的循环，因此能够更加简便地计算容器中所有元素的和：

```
double total {};
for(auto&& value : values)
  total += value;
```

当然，通过把容器作为参数传给函数，也能达到同样的效果。数组容器的成员函数 front()和 back()分别返回第一个元素和最后一个元素的引用。成员函数 data()同样能够返回&front()，它是容器底层用来存储元素的标准数组的地址，一般不会用到。

模板函数 get<n>()是一个辅助函数，它能够获取到容器的第 n 个元素。模板参数的实参必须是一个在编译时可以确定的常量表达式，所以它不能是一个循环变量。它只能访问模板参数指定的元素，编译时会对它进行检查。get<n>()模板提供了一种不需要在运行时检查，但能用安全的索引值访问元素的方式。下面展示如何使用它：

```
std::array<std::string, 5> words {"one", "two", "three", "four", "five"};
std::cout << std::get<3>(words) << std::endl; // Output words[3]
std::cout << std::get<6>(words) << std::endl; // Compiler error message!
```

下面是一个示例，展示了关于数组容器你到目前为止所学到的知识：

```
// Ex2_01.cpp
/*
  Using array<T,N> to create a Body Mass Index (BMI) table
  BMI = weight/(height*height)
  weight is in kilograms, height is in meters
*/

#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
#include <array>                       // For array<T,N>

int main()
{
  const unsigned int min_wt {100U};   // Minimum weight in table in lbs
  const unsigned int max_wt {250U};   // Maximum weight in table in lbs
  const unsigned int wt_step {10U};   // Weight increment
  const size_t wt_count {1 + (max_wt - min_wt) / wt_step};

  const unsigned int min_ht {48U};    // Minimum height in table in inches
  const unsigned int max_ht {84U};    // Maximum height in table in inches
  const unsigned int ht_step {2U};    // Height increment
  const size_t ht_count { 1 + (max_ht - min_ht) / ht_step };

  const double lbs_per_kg {2.20462}; // Conversion factor lbs to kg
  const double ins_per_m {39.3701};  // Conversion factor ins to m

  std::array<unsigned int, wt_count> weight_lbs;
  std::array<unsigned int, ht_count> height_ins;

  // Create weights from 100lbs in steps of 10lbs
  for (size_t i{}, w{ min_wt } ; i < wt_count ; w += wt_step, ++i)
  {
    weight_lbs.at(i) = w;
  }
  // Create heights from 48 inches in steps of 2 inches
  unsigned int h{ min_ht };
  for(auto& height : height_ins)
  {
    height = h;
    h += ht_step;
  }

  // Output table headings
  std::cout << std::setw(7) << " |";
  for (const auto& w : weight_lbs)
    std::cout << std::setw(5) << w << " |";
  std::cout << std::endl;

  // Output line below headings
  for (size_t i{1} ; i < wt_count ; ++i)
    std::cout << "----------";
  std::cout << std::endl;
```

```
    double bmi {};                          // Stores BMI
    unsigned int feet {};                   // Whole feet for output
    unsigned int inches {};                 // Whole inches for output
    const unsigned int inches_per_foot {12U};
    for (const auto& h : height_ins)
    {
      feet = h / inches_per_foot;
      inches = h % inches_per_foot;
      std::cout << std::setw(2) << feet << "'" << std::setw(2) << inches <<
"\"" << "|";
      std::cout << std::fixed << std::setprecision(1);
      for (const auto& w : weight_lbs)
      {
        bmi = h / ins_per_m;
        bmi = (w / lbs_per_kg) / (bmi*bmi);
        std::cout << std::setw(2) << " " << bmi << " |";
      }
      std::cout << std::endl;
    }
    // Output line below table
    for (size_t i {1} ; i < wt_count ; ++i)
      std::cout << "----------";
    std::cout << "\nBMI from 18.5 to 24.9 is normal" << std::endl;
}
```

在本书中，并没有展示这个示例的输出结果，因为可能会占据很多篇幅。这里有两套参数，每套定义了4个有关身高、体重范围的常量，它们被包含在BMI表中。因为身高、体重都是整数、非负数，所以存放在数组容器中的身高和体重都是unsigned int类型的元素。在循环中，可以用适当的值初始化容器。第一个循环展示了at()函数的使用，这里也可以放心地使用weight_lbs[i]。接下来的两个循环分别输出了表的列头，以及一条用来分隔表头和表的横线。数据表是以循环嵌套的方式生成的。外循环遍历身高并输出英尺和英寸的最左一列的身高。内循环遍历体重，输出当前身高每行的BMI值。

2.2.2 使用数组容器的迭代器

数组模板定义了成员函数begin()和end()，分别返回指向第一个元素和最后一个元素的下一个位置的随机访问迭代器。如第1章所述，随机访问迭代器具有最多的功能，能使用它进行全部的操作。可以在循环中显式地使用迭代器来设置height_ins容器的值：

```
unsigned int h {min_ht};
auto first = height_ins.begin();                    // Iterator pointing to
                                                    // 1st element
auto last = height_ins.end();                       // Iterator pointing to 1
                                                    // past last element
while (first != last)
{
```

```
    *first++ = h;                                    // Store h in current element
    and increment iterator
    h += ht_step;
  }
```

迭代器对象是由 array 对象的成员函数 begin()和end()返回的。使用 auto 时不需要担心迭代的实际类型，除非你要自己考虑，在本例中它们是 std::array<unsigned int,19>::iterator 类型，这意味着 iterator 类型被定义在 array<T,N>类型中。可以看出，可以像使用普通指针那样去使用迭代器对象。在保存了元素值后，使用后缀++运算符对 first 进行自增。当 first 等于 end 时，所有的元素都被设完值，循环结束。

如第 1 章所述，最好用全局的begin()和 end()函数来从容器中获取迭代器，因为它们是通用的，first 和 last 可以像下面这样定义：

```
auto first = std::begin(height_ins); // Iterator pointing to 1st element
auto last = std::end(height_ins);    // Iterator pointing to 1 past last element
```

记住，当迭代器指向容器中的一个特定元素时，它们没有保留任何关于容器本身的信息，所以我们无法从迭代器中判断，它是指向 array 容器还是指向 vector 容器。容器中的一段元素可以由迭代器指定，这让我们有了对它们使用算法的可能，那么我们有适用于 Ex2_01.cpp 的任何算法么？定义在 algorithm 头文件中的 generate()函数模板，提供了用函数对象计算值、初始化一段元素的可能。我们可以像这样重写之前用来初始化 height_ins 容器的代码段：

```
unsigned int height {}; // Stores the current height initializing value
std::generate(std::begin(height_ins), std::end(height_ins),
              [height, &min_ht, &ht_step]()mutable
                  { return height += height == 0 ? min_ht : ht_step; });
```

为容器元素设置值的语句现在减少到两条，也不再需要显式的循环语句。第一条语句定义了一个变量用来保存元素的初始值。generate()的前两个参数分别是开始迭代器和结束迭代器，用来指定需要设置值的元素的范围。第三个参数是一个 lambda 表达式。lambda 表达式以引用的方式捕获 min_hi、ht_step。mutable 关键字使 lambda 表达式能够更新 height 局部副本的值，它是以值引用的方式捕获的。在 return 语句中，lambda 第一次执行后，height 的局部副本的值被设为 min_ht。然后，随着 lambda 的每次调用，height 都会增加 ht_step。在 lambda 表达式中，以值引用的方式捕获的变量局部副本的值会被一直保存，这一机制正好满足了我们的要求。

假定要用连续的递增值初始化一个数组容器，这里有一个函数模板 iota()可以做到，它定义在头文件 numeric 中。这里有一个它的用法示例：

```
std::array<double, 10> values;
std::iota(std::begin(values), std::end(values), 10.0);
   // Set values
   elements to 10.0 to 19.0
```

前两个参数是迭代器，用来定义需要设置值的元素的范围。第三个参数是第一个元素要设置的值，通过递增运算生成了随后每一个元素的值。iota()函数的使用并不仅限于数值。元素可以设为任意类型的值，只要这些类型支持 operator++()。

■ **注意**：不要忘记算法是独立于容器类型的，对于任何具有指定类型迭代器的容器来说，算法都可以应用到它们的元素上。generate()和 iota()函数模板只需要正向迭代器，所以用来指定任何容器的元素范围的迭代器都能发挥作用。

容器定义了成员函数 cbegin()和 cend()，它们可以返回 const 迭代器。当只想访问元素时，应该使用 const 迭代器。对于 non-const 迭代器，最好使用全局的 cbegin 和 cend()来获取。全局函数或成员函数 rbegin()和 rend()可以分别得到指向最后一个元素和第一个元素前一个位置的反向迭代器。函数 crbegin()和 crend()可以返回 const 反向迭代器。我们可以用反向迭代器以逆序方式处理元素。例如：

```
std::array<double, 5> these {1.0, 2.0, 3.0, 4.0, 5.0};
double sum {};
auto start = std::rbegin(these);
auto finish = std::rend(these);
while(start != finish)
   sum += *(start++);
std::cout << "The sum of elements in reverse order is " << sum << std::endl;
```

在循环中，从最后一个元素开始，我们计算出了所有元素的和。结束迭代器指向第一个元素之前的位置，所以 sum 加上第一个元素后，循环就结束了。在反向迭代器上使用递增运算符，会让迭代器用一种和普通正向迭代器移动方向相反的方式移动。我们也可以使用 for 循环：

```
for(auto iter = std::rbegin(these); iter != std::rend(these); ++iter)
  sum += *iter;
```

因为数组容器实例的元素个数固定，所以无法使用插入迭代器。插入迭代器通常用来向容器中添加元素。

2.2.3　比较数组容器

可以用任何比较运算符比较两个数组容器，只要它们有相同的大小，保存的是相同类型的元素，而且这种类型的元素还要支持比较运算符。示例如下：

```
std::array<double,4> these {1.0, 2.0, 3.0, 4.0};
std::array<double,4> those {1.0, 2.0, 3.0, 4.0};
std::array<double,4> them {1.0, 3.0, 3.0, 2.0};

if (these == those) std::cout << "these and those are equal." << std::endl;
if (those != them) std::cout << "those and them are not equal." << std::endl;
if (those < them) std::cout << "those are less than them." << std::endl;
if (them > those) std::cout << "them are greater than those." << std::endl;
```

容器被逐元素地比较。对于==，如果两个数组对应的元素都相等，会返回 true。对于!=，两个数组中只要有一个元素不相等，就会返回 true。这也是字典中单词排序的根本方式，两个单词中相关联字母的不同决定了单词的顺序。这个代码片段中所有的比较运算都是 true。所以当它们执行时，会输出 4 条消息。

不像标准数组，只要它们存放的是相同类型、相同个数的元素，就可以将一个数组容器赋给另一个。例如：

```
them = those;        // Copy all elements of those to them
```

赋值运算符左边的数组容器的元素，将会被赋值运算符右边的数组容器的元素覆盖。

2.3 使用 vector<T>容器

vector<T>容器是包含 T 类型元素的序列容器，和 array<T,N>容器相似，但不同的是 vector<T>容器的大小可以自动增长，从而可以包含任意数量的元素；因此类型参数 T 不再需要模板参数 N。只要元素个数超出 vector 当前容量，就会自动分配更多的空间。只能在容器尾部高效地删除或添加元素。Vector<T>容器可以方便、灵活地代替数组。在大多数时候，都可以用 vector<T>代替数组存放元素。只要能够意识到，vector<T>在扩展容量，以及在序列内部删除或添加元素时会产生一些开销；但大多数情况下，代码不会明显变慢。为了使用 vector<T>容器模板，需要在代码中包含头文件 vector。

2.3.1 创建 vector<T>容器

下面是一个生成存放 double 型元素的 vector<T>容器示例：

```
std::vector<double> values;
```

因为容器中没有元素，所以没有分配空间，当添加第一个数据项时，会自动分配内存。可以像下面这样通过调用 reserve()来增加容器的容量：

```
values.reserve(20);   // Memory for up to 20 elements
```

这样就设置了容器的内存分配，至少可以容纳 20 个元素。如果当前的容量已经大于或等于 20 个元素，那么这条语句什么也不做。注意，调用 reserve()并不会生成任何元素。values 容器这时仍然没有任何元素，直到添加了 20 个元素后，才会分配更多的内存。调用 reserve()并不会影响现有的元素。当然，如果通过调用 reserve()来增加内存，任何现有的迭代器，例如开始迭代器和结束迭代器，都会失效，所以需要重新生成它们。这是因为，为了增加容器的容量，vector<T>容器的元素可能已经被复制或移到了新的内存地址。

创建 vector 容器的另一种方式是使用初始化列表来指定初始值以及元素个数：

```
std::vector<unsigned int> primes {2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u};
```

以初始化列表中的值作为元素初始值，生成有 8 个素数的 vector 容器。

分配内存是比较花费时间的，所以最好只在必要时分配。vector 使用算法来增加容量，这个算法依赖一个经常使用的常对数来实现，这在早些时候会导致分配一些非常小的内存，但是随着 vector 容量的增大，内存增长数也会变大。可以如下所示，使用初始元素个数来生成 vector 容器：

```
std::vector<double> values(20);
// Capacity is 20 double values and there are 20 elements
```

这个容器开始时有 20 个元素，它们的默认初始值都为 0。生成容器时，同时指定元素个数，就能够减少空间额外分配的次数，这是一个很好的习惯。

■ **警告**：圆括号中的 20 表示元素的个数，是上述语句的核心。这里不能使用{}。如果如下所示定义 vector 容器，会产生不同的结果：

```
std::vector<double> values {20}; // There is one element initialized to 20
```

vector 并没有 20 个元素。它只有一个元素，并以 20 作为初始值。添加元素会导致分配额外的内存。

如果不想用 0 作为默认值，可以指定一个其他值：

```
std::vector<long> numbers(20, 99L);
// Size is 20 long values - all initialized with 99
```

第二个参数指定了所有元素的初始值，因此这 20 个元素的值都是 99L。第一个元素指定了 vector 中的元素个数，它不需要是一个常量表达式。它可以是一个表达式执行后的结果，也可以是从键盘输入的数。

可以用元素类型相同的容器来初始化 vector<T>容器。用一对迭代器来指定初始值的范围。下面是一个示例：

```
std::array<std::string, 5> words {"one", "two", "three", "four", "five"};
std::vector<std::string> words_copy {std::begin(words), std::end(words)};
```

words_copy 被 words 数组容器中的元素初始化。如果使用移动迭代器指定 words_copy 的初始化范围，words 中的元素将会从 words 移到 words_copy。这里有一个示例：

```
std::vector<std::string> words_copy {std::make_move_iterator(std::
                                     begin(words)),
                                     std::make_move_iterator(std::
                                      end(wo rds))};
```

words_copy 会像前面那样被初始化。但元素是移动过来的而不是复制过来的，所以 words 数组中的字符串对象现在都是空字符串。

2.3.2 vector 的容量和大小

vector 的容量大小，是指在不分配更多内存的情况下可以保存的最多元素个数，这时可能有 20 个元素，也可能没有。vector 的大小是它实际所包含的元素个数，也就是有值的元素的个数。图 2-3 对此做了说明：

```
std::vector<size_t> values {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47};
values.reserve(20);          // Increase capacity to 20
```

图 2-3　vector 的容量和大小

显然 vector 的大小不能超出它的容量。当大小等于容量时，增加一个元素就会导致更多内存的分配。对于一个 vector 对象来说，可以通过调用 size()和 capacity()函数来得到它的大小和容量。它们返回的是我们自己定义的无符号整型值。例如：

```
    std::vector<size_t> primes { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41 ,43 ,47 };
    std::cout << "The size is " << primes.size() << std::endl;
    std::cout << "The capacity is " << primes.capacity() << std::endl;
```

输出语句输出的容器大小和容量都为 15，这是由初始化列表决定的。然而，如果用 push_back()函数添加一个元素，然后再输出容器的大小和容量，这时大小变为 16，容量变为 30。当容器的大小等于容量时，容器每次增加多少容量，取决于算法的实现。一些实现可能会双倍地增加容量。

我们可能想把容器的大小和容量保存在变量中。vector<T>对象的大小和容量类型是 vector<T>::size_type，这表明 size_type 定义在一个由编译器从类模板中生成的vector<T>类中。因此，primes 的大小值是 vector<size_t>::size_type 类型。当定义一个变量去保存这些值时，通过使用 auto 关键字，可以让我们在大多数时候不需要去考虑这种细节。例如：

```
    auto nElements = primes.size();  // Store the number of elements
```

记住，对于 auto，需要使用=，而不能使用初始化列表；否则，就不能确定 size_type 的类型。通常保存容器的大小是为了可以用索引遍历 vector 的元素。我们也可以使用循环来遍历 vector：

```
    for(auto& prime : primes)
      prime *= 2;                          // Double each element value
```

在前面，我们知道可以通过调用 reserve()来增加容器的容量；这时元素的个数并没有改变。通过调用成员函数 resize()可以改变容器大小，这也可能导致容量的增加。下面是 resize()的几种用法：

```
std::vector<int> values {1,2,3}; // 1 2 3 : size is 3
values.resize(5);                // 1 2 3 0 0 : size is 5
values.resize(7, 99);            // 1 2 3 0 0 99 99 : size is 7
values.resize(6);                // 1 2 3 0 0 99 : size is 6
```

第一个 resize()调用会把元素的个数变为参数指定的值，所以会增加两个用默认值初始化的元素。如果添加了一个元素，导致超过当前容器的容量，容量会自动增加。第二个 resize()调用将元素增加到第一个参数指定的个数，并用第二个参数初始化增加的新元素。第三个 resize()调用将容器大小设为 6，小于当前元素的个数。当需要减小容器的大小时，会移除多余的元素，这就好像重复调用了几次 pop_back()函数。在本章的后面，会对此做一些解释。减少容器的大小不会影响容器的容量。

2.3.3　访问元素

可以在方括号中使用索引，为现有元素设定值，或者只是通过表达式使用它的值。例如：

```
std::vector<double> values(20);      // Container with 20 elements created
values[0] = 3.14159;                 // Pi
values[1] = 5.0;                     // Radius of a circle
values[2] = 2.0*values[0]*values[1]; // Circumference of a circle
```

vector 的索引从 0 开始，这和标准数组一样。通过使用索引，总是可以访问到现有的元素，但是不能这样生成新元素——需要使用 push_back()、insert()、emplace()或 emplace_back()。当像这样索引一个 vector 时，和数组容器一样，并没有检查索引值，所以当索引可能越界时，应该通过 at()函数去使用这个元素。

vector 的成员函数 front()和 back()分别返回序列中第一个和最后一个元素的引用，例如：

```
std::cout << values.front() << std::endl;    // Outputs 3.14159
```

因为成员函数 front()和 back()返回的是引用，所以它们可以出现在赋值运算符的左边。

```
values.front() = 2.71828;                    // 1st element changed to 2.71828
```

成员函数 data()返回一个指向数组的指针，它在内部被用来存储元素。例如：

```
auto pData = values.data();
```

pData 是 double*类型，一般来说，data()返回 vector<T>容器的 T*类型的值。在必要时，可以使用这个函数。

2.3.4 使用 vector 容器的迭代器

正如期望的那样，vector 容器实现了所有可以返回迭代器的成员函数，包括 const 迭代器和 non-const 迭代器，以及反向迭代器。vector 容器的迭代器是随机访问迭代器。当然，也可以通过全局函数获取它们。vector 有成员函数 push_back()，所以能够通过使用 back_insert_iterator 来添加新的值。你从第 1 章了解到，可以通过调用全局的 back_inserter()函数来获取一个后向插入迭代器。无法在 vector 容器中使用 front_insert_iterator，这需要 vector 有成员函数 push_front()，但是 vector 容器并没有定义这个函数。

可以通过演示如何用 copy()算法来添加元素，向你展示怎样在 vector 中使用后向插入迭代器。copy()的头两个参数是两个迭代器，指定了复制元素的范围，第三个参数指定了这些元素存放的位置。头两个参数要求是输入迭代器，所以可以接受任何其他类别的迭代器；显然第三个参数必须是一个输出迭代器。这里有一个示例：

```
std::vector<double> data {32.5, 30.1, 36.3, 40.0, 39.2};
std::cout << "Enter additional data values separated by spaces or Ctrl+Z
             to end:"
          << std::endl;
std::copy(std::istream_iterator<double>(std::cin), std::istream_iterator
                                                   <double>(),
                                                   std::back_inserter(data));
std::copy(std::begin(data), std::end(data), std::ostream_iterator<double>
   (std::cout, " "));
```

用初始化列表生成 data 容器。第一次调用 copy()时，使用一个 istream_iterator 对象作为第一个参数，它能够从标准输入流中读取 double 类型的值。第二个参数是一个流的结束迭代器，当识别到流结束时，istream_iterator 会变为结束迭代器；当从键盘输入 Ctrl+Z 时，这也会发生在 cin 中。

copy()的第三个参数是读入值的存放位置，是 data 容器的一个 back_insert_iterator，它是由 back_inserter()函数返回的，因此从 cin 读出的值都被作为新元素添加到 data 容器的后面。最后一次调用 copy()，会将 data 容器的所有元素复制到 cout；这是通过将一个 ostream_iterator 对象作为目的地址来实现的。让我们使用 vector 容器的迭代器来尝试一个完整的示例：

```
// Ex2_02.cpp
// Sorting strings in a vector container
#include <iostream>          // For standard streams
#include <string>            // For string types
#include <algorithm>         // For swap() and copy() functions
#include <vector>            // For vector (and iterators)
using std::string;
using std::vector;

int main()
{
```

```
vector<string> words;       // Stores words to be sorted
words.reserve(10);          // Allocate some space for elements
std::cout << "Enter words separated by spaces. Enter Ctrl+Z on a separate
          line to end:"
          << std::endl;
std::copy(std::istream_iterator<string> {std::cin}, std::istream_iterator
                                        <string> {},
                                        std::back_inserter(words));
std::cout << "Starting sort." << std::endl;
bool out_of_order {false}; // true when values are not in order
auto last = std::end(words);
while (true)
{
  for (auto first = std::begin(words) + 1; first != last; ++first)
  {
    if (*(first - 1) > *first)
    { // Out of order so swap them
      std::swap(*first, *(first - 1));
      out_of_order = true;
    }
  }
  if (!out_of_order) // If they are in order (no swaps necessary)...
    break; // ...we are done...
  out_of_order = false; // ...otherwise, go round again.
}

// Output the sorted vector
std::cout << "your words in ascending sequence:" << std::endl;
std::copy(std::begin(words), std::end(words),
          std::ostream_iterator<string> {std::cout, " "});
std::cout << std::endl;

// Create a new vector by moving elements from words vector
vector<string> words_copy {std::make_move_iterator(std::begin(words)),
                           std::make_move_iterator(std::end(words)) };
std::cout << "\nAfter moving elements from words, words_copy contains:"
          << std::endl;
std::copy(std::begin(words_copy), std::end(words_copy),
          std::ostream_iterator<string> {std::cout, " "});
std::cout << std::endl;

// See what's happened to elements in words vector...
std::cout << "\nwords vector has " << words.size() << " elements\n";
if (words.front().empty())
  std::cout << "First element is empty string object." << std::endl;

std::cout << "First element is \"" << words.front() << "\"" << std::endl;
}
```

示例输出如下：

```
Enter words separated by spaces. Enter Ctrl+Z on a separate line to end:
one two three four five six seven eight
^Z

Starting sort.
your words in ascending sequence:
eight five four one seven six three two

After moving elements from words, words_copy contains:
eight five four one seven six three two
words vector has 8 elements
First element is empty string object.
First element is ""
```

该程序使用流迭代器从标准输入流读取单词，然后将其作为字符串对象写入一个vector 容器。可以输入任意个数的单词。容器会在必要时自动增长。这里调用容器的 reserve()函数来为 10 个元素分配内存。有一个好主意是，每次只分配大致需要的内存，这会减少小幅度分配内存所带来的开销。back_inserter()生成了一个 back_insert_iterator，它能够调用容器的成员函数 push_back()，来将每一个字符串对象作为新元素添加到容器中。

copy()算法的头两个参数是输入流迭代器，其中的第二个参数是结束流迭代器。当从键盘输入 Ctrl+Z 时，流迭代器就会匹配到它，这相当于文件流的 EOF。

这里对 vector 元素进行排序的代码展示了迭代器的使用。稍后你会看到，sort()算法可以只用一条语句就完成相同的工作。这里的排序算法是十分简单的冒泡排序，通过遍历元素来反复排序。在每一趟排序中，如果临近的元素无序，就会互相交换。swap()函数定义在 algorithm 头文件中，可以高效地交换任何类型的元素。如果在一趟排序中，所有元素都没有交换，那么所有元素已经是升序序列了。最外层的循环是一个由迭代器控制的 for 循环。first 的初始值是 begin(words)+1，它指向 vector 的第二个元素。从第二个元素开始，是为了确保能够使用 first-1，这样可以保证两个连续元素的比较总是合法。当 first 自增到 end(words)时，一趟排序就会结束。

对 words vector 中的内容排序后的结果，可以通过使用copy()算法将全部元素转移到输出流迭代器来显示。转移元素的范围是由begin()和end()返回的迭代器指定的，所以会输出全部元素。ostream_iterator 构造函数的参数是数据流向的目的地址，分隔字符串会分隔每一个输出的值。

main()的最后一部分代码展示了如何使用移动迭代器，这里移动了所有元素。在这个操作之后，从程序输出可以发现，words 中包含的字符串都变成了空字符串。移动一个元素会留下一个由无参字符串构造函数创建的对象。一般来说，移动一个是类对象的元素，会导致这个元素处于一种不确定的状态，因此我们不应该再去使用这个对象。

main()中的排序代码其实并不依赖存放元素的容器。它只要求迭代器指定的元素能够支持排序算法的运算。STL 有一个 sort()函数模板，它远比我们能想出的任何方法都好。有时候，我们也可以定义自己的函数模板，去对能够排序的任意类型元素进行排序：

```
template<typename RandomIter>
```

```
void bubble_sort(RandomIter start, RandomIter last)
{
  std::cout << "Starting sort." << std::endl;
  bool out_of_order {false}; // true when values are not in order
  while (true)
  {
    for (auto first = start + 1; first != last; ++first)
    {
      if (*(first - 1) > *first)
      { // Out of order so swap them
        std::swap(*first, *(first - 1));
        out_of_order = true;
      }
    }
    if (!out_of_order)          // If they are in order (no swaps necessary)...
      break;                    // ...we are done...
    out_of_order = false;       // ...otherwise, go round again.
  }
}
```

模板类型参数是迭代器类型。因为 for 循环中迭代器算术操作的原因，bubble_sort()算法需要使用随机访问迭代器。只要容器可以提供随机访问迭代器，算法就可以对这个容器的内容进行排序；这也包括标准数组和字符串对象。如果在前面的 main()中使用此代码，就可以使用下面的语句替换掉 main()中对 words 进行排序的部分：

```
bubble_sort(std::begin(words), std::end(words)); // Sort the words array
```

定义一个只用迭代器实现操作的函数模板，会使这个函数的用法变得更灵活。任何处理一段元素的算法都可以用这种方式生成。完整使用 bubble_sort()模板的代码可以从 Ex2_02A 获取。

2.3.5　向 vector 容器中添加元素

记住，向容器中添加元素的唯一方式是使用它的成员函数。如果不调用成员函数，非成员函数既不能添加也不能删除元素。这意味着容器对象必须通过它所允许的函数去访问，迭代器显然不行。

1. 增加元素

可以通过使用容器对象的 push_back()函数，在序列的末尾添加一个元素。例如：

```
std::vector<double> values;
values.push_back(3.1415926); // Add an element to the end of the vector
```

在这个示例中，push_back()函数以传入的参数-3.1415926 作为新元素的值，然后把它添加到现有元素的后面。因为这里并没有现有的元素，所以这个元素就是第一个元素。如果没有调用 reserve()，容器就会为这个新元素分配内存。这里，第二个版本的 push_back()

使用了右值引用参数，这样就可以通过移动运算来添加元素。例如：

```
std::vector<std::string> words;
words.push_back(string("adiabatic"));
// Move string("adiabatic") into the vector
```

这里 push_back()的参数是一个临时对象，因此这会调用右值引用版的函数。当然，也可以这样写：

```
words.push_back("adiabatic"); // Move string("adiabatic") into the vector
```

编译器会生成一个以"adiabatic"为初值的 string 对象，然后这个对象会像前面那样移动到 vector 中。

还有一个更好的方法来添加元素。emplace_back()比 push_back()更有效率。下面这个代码片段说明了为什么：

```
std::vector<std::string> words;
words.push_back(std::string("facetious")); // Calls string constructor &
  // moves the string object
words.emplace_back("abstemious");          // Calls string constructor to
  // create element in place
```

emplace_back()的参数正是添加到容器中的对象的构造函数所需要的参数。emplace_back()用它的参数作为构造函数的参数，在容器中生成对象。如果不想使用移动运算，这个示例中就要使用 push_back()。可以在 emplace_back()函数中使用尽可能多的参数，只要它们满足对象构造函数的要求。这里有一个使用多参数的 emplace_back()的示例：

```
std::string str {"alleged"};
words.emplace_back(str, 2, 3);
// Create string object corresponding to "leg" in place
```

emplace_back()函数会调用接收三个参数的 string 构造函数，生成 string 对象，然后把它添加到 words 序列中。构造函数会生成一个从索引 2 开始、包含 str 中三个字符的子串。

2. 插入元素

通过使用成员函数 emplace()，可以在 vector 序列中插入新的元素。对象会在容器中直接生成，而不是先单独生成对象，然后再把它作为参数传入。emplace()的第一个参数是一个迭代器，它确定了对象生成的位置。对象会被插入到迭代器所指定元素的后面。第一个参数后的参数，都作为插入元素的构造函数的参数传入。例如：

```
std::vector<std::string> words {"first", "second"};
// Inserts string(5, 'A') as 2nd element
auto iter = words.emplace(++std::begin(words), 5, 'A');
// Inserts string("$$$$") as 3rd element
words.emplace(++iter, "$$$$");
```

这段代码执行后，vector 中的字符串对象如下：

```
"first" "AAAAA" "$$$$" "second"
```

在 emplace()的第一个参数的后面，可以使用尽可能多的参数，只要它们是被插入对象的构造函数所需要的。在上面的代码片段中，第一次调用 emplace()会得到一个由构造函数 string(5, 'A')生成的字符串对象。emplace()会返回一个指向插入元素的迭代器，被用来在插入元素的后面，插入一个新的元素。

成员函数 insert()可以在 vector 中插入一个或多个元素。第一个参数总是一个指向插入点的 const 或 non-const 迭代器。元素会被迅速插入到第一个参数所指向元素的前面，如果第一个参数是一个反向迭代器，元素会被插入到迭代器所指向元素的后面。如果选择使用 insert()来插入元素，稍后会分别阐述每一种可能的情况。会先定义一个 vector，然后列出一个相继调用 insert()的列表：

```
std::vector<std::string> words {"one", "three", "eight"};
// Vector with 3 elements
```

下面介绍一些使用 insert()插入单词的方式：

1) 插入第二个参数指定的单个元素：

```
auto iter = words.insert(++std::begin(words), "two");
```

在这个示例中，插入点是由 begin()返回的迭代器递增后得到的。它对应第二个元素，所以新元素会作为新的第二个元素插入，之前的第二个元素以及后面的元素，为了给新的第二个元素留出空间，都会向后移动一个位置。这里有两个 insert()重载版本，它们都可以插入单个对象，其中一个的参数是 const T&类型，另一个是 T&&类型——右值引用。因为上面的第二个参数是一个临时对象，所以会调用第二个函数重载版本，临时对象会被移动插入而不是被复制插入容器。

执行完这条语句后，words rector 容器包含的字符串元素为：

```
"one" "two" "three" "eight"
```

返回的迭代器指向被插入的元素 string("two")。需要注意的是，在使用同样参数的情况下，调用 insert()没有调用 emplace()高效。在 insert()调用中，构造函数调用 string("two")生成了一个对象，作为传入的第二个参数。在 emplace()调用中，构造函数用第二个参数直接在容器中生成了字符串对象。

2) 插入一个由第二个和第三个参数指定的元素序列：

```
std::string more[] {"five", "six", "seven"}; // Array elements to be inserted
iter = words.insert(--std::end(words), std::begin(more), std::end(more));
```

第二条语句中的插入点是一个迭代器，它是由 end()返回的迭代器递减后得到的。对应最后一个元素，因此新元素会被插入到它的前面。执行这条语句后，words vector 容器中的字符串对象为：

```
"one" "two" "three" "five" "six" "seven" "eight"
```

返回的迭代器指向插入的第一个元素"five"。

3) 在 vector 的末尾插入一个元素：

```
iter = words.insert(std::end(words), "ten");
```

插入点是最后一个元素之后的位置，因此新元素会被添加到最后一个元素之后。执行完这条语句后，words vector 容器中的字符串对象如下：

```
"one" "two" "three" "five" "six" "seven" "eight" "ten"
```

返回的迭代器指向插入的元素"ten"。这和上面的情况 1)相似；这表明，当第一个参数不指向元素而是指向最后一个元素之后的位置时，它才发挥作用。

4) 在插入点插入多个单个元素。第二个参数是第三个参数所指定对象的插入次数：

```
iter = words.insert(std::cend(words)-1, 2, "nine");
```

插入点是最后一个元素，因此新元素 string("nine")的两个副本会被插入到最后一个元素的前面。

执行完这条语句后，words vector 容器中的字符串对象如下：

```
"one" "two" "three" "five" "six" "seven" "eight" "nine" "nine" "ten"
```

返回的迭代器指向插入的第一个元素"nine"。注意，示例中的第一个参数是一个 const 迭代器，这也表明可以使用 const 迭代器。

5) 在插入点，插入初始化列表指定的元素。第二个参数就是被插入元素的初始化列表：

```
iter = words.insert(std::end(words), { std::string {"twelve"},
                                       std::string {"thirteen"}});
```

插入点越过了最后一个元素，因此初始化列表中的元素被添加到容器的尾部。执行完这条语句后，words vector 容器中的字符串对象如下：

```
"one" "two" "three" "five" "six" "seven" "eight" "nine" "nine" "ten" "twelve"
"thirteen"
```

返回的迭代器指向插入的第一个元素" twelve "。初始化列表中的值必须和容器的元素类型相匹配。T 类型值的初始化列表是 std::initializer_list<T>，所以这里的 list 类型为 std::initializer_list<std::string>。前面的 insert()调用中以单词作为参数的地方，参数类型是 std::string，所以单词作为字符串对象的初始值被传入到函数中。

记住，所有不在 vector 尾部的插入点都会有开销，需要移动插入点后的所有元素，从而为新元素空出位置。当然，如果插入点后的元素个数超出了容量，容器会分配更多的内存，这会增加更多额外开销。

vector 的成员函数 insert()，需要一个标准的迭代器来指定插入点；它不接受一个反向迭代器——这无法通过编译。如果需要查找给定对象的最后一个元素，或者在它的后面插入一个新的元素，就需要用到反向迭代器。这里有一个示例：

```
std::vector<std::string> str {"one", "two", "one", "three"};
auto riter = std::find(std::rbegin(str), std::rend(str), "one");
str.insert(riter.base(), "five");
```

find()算法会在头两个参数所指定的一段元素中，搜索第三个参数指定的元素，返回第一个找到的元素，因此会找到 string("one")。它会返回一个迭代器，这个迭代器和用来指定搜索范围的迭代器有相同的类型，是一个指向匹配元素的反向迭代器。如果没有找到匹配的元素，那么它就是指向第一个元素之前位置的迭代器 rend(str)。使用反向迭代器意味着 find()会找到最后匹配的元素；使用标准迭代器会找到第一个匹配的元素，如果没有匹配的元素，会返回 end(str)。调用 riter 的成员函数 base()可以得到一个标准迭代器，从序列反方向来看，它指向 riter 前的一个位置，也是朝向序列结束的方向。因为 riter 指向第三个元素，也就是“one”，所以 riter.base()指向第 4 个元素“three”。如果使用 riter.base()作为 insert()的第一个参数，“five”将被插入到这个位置之前，也就是 riter 所指向元素的后面。执行完这些语句后，str 容器会包含下面 5 个字符串元素：

```
"one", "two", "one", "five", "three"
```

如果想把插入点变成 find()返回位置的前一个位置，需要将 insert()的第一个参数变为 iter.base()−1。

2.3.6　删除元素

正像所说的那样，只能通过容器的成员函数来删除元素。可以通过使用 vector 的成员函数 clear()来删除所有的元素。例如：

```
std::vector<int> data(100, 99);// Contains 100 elements initialized to 99
data.clear();                  // Remove all elements
```

第一条语句创建了一个有 100 个 int 型元素的 vector 对象，它的大小和容量都是 100；所有元素的初始值都是 99。第二条语句移除了所有的元素，因此大小变为 0，因为这个操作并没有改变容器的容量，所以容量还是 100。

可以使用 vector 的成员函数 pop_back()来删除容器尾部的元素。例如：

```
std::vector<int> data(100, 99); // Contains 100 elements initialized to 99
data.pop_back(); // Remove the last element
```

第二条语句移除了最后一个元素，因此 data 的大小变为 99，容量还是 100。只要不在意元素的顺序，就可以通过删除最后一个元素的方式来删除容器的任何元素，这不需要移动大量元素。假设要删除 data 中的第二个元素，可以像这样操作：

```
std::swap(std::begin(data)+1, std::end(data)-1);
  // Interchange 2nd element with the last
data.pop_back(); // Remove the last element
```

第一条语句调用了模板函数 swap()，它在头文件 algorithm 和 utility 中都有定义。这个

函数将第二个元素和最后一个元素互相交换。然后调用 pop_back()移除最后一个元素，这样就从容器中移除了第二个元素。

注意：vector 也有成员函数 swap()，这个函数用来交换两个 vector 容器中的元素。显然，这两个容器的元素类型必须相同。全局的 swap()函数只要将两个容器作为参数，也可以交换它们的元素。

如果要去掉容器中多余的容量，例如不再向容器中添加新元素，那么可以通过使用成员函数 shrink_to_fit()来实现：

```
data.shrink_to_fit(); // Reduce the capacity to that needed for elements
```

不管这个操作是否依赖 STL 的实现，如果它生效了，那么这个容器现有的迭代器都失效，所以在执行完这个操作后，最好重新获取迭代器。

可以使用成员函数 erase()来删除容器中的一个或多个元素。如果只删除单个元素，那么只需要提供一个参数，例如：

```
auto iter = data.erase(std::begin(data)+1); // Delete the second element
```

删除一个元素后，vector 的大小减 1；但容量不变。会返回一个迭代器，它指向被删除元素后的一个元素。这里的返回值和表达式 std::begin(data)+1 相关；如果移除了最后一个元素，会返回 std::end(data)。

如果要移除一个元素序列，只需要传入两个迭代器，用来指定移除元素的范围。例如：

```
// Delete the 2nd and 3rd elements
auto iter = data.erase(std::begin(data)+1, std::begin(data)+3);
```

不要忘记，第二个迭代器指向这段元素末尾的下一个位置。上面的语句删除了位于 std::begin(data)+1 和 std::begin(data)+2 的元素。返回的迭代器指向被删除元素后的位置，它是 std::begin(data)+1；如果删除了最后一个元素，它就是 std::end(data)。

remove()算法由定义在 algorithm 头文件中的模板生成，它可以删除匹配特定值的一段元素。例如：

```
std::vector<std::string> words { "one", "none", "some", "all", "none",
"most", "many"};
auto iter = std::remove(std::begin(words), std::end(words), "none");
```

第二条语句在头两个参数指定的元素范围内，移除了所有匹配 remove()的第三个参数 string("none")的元素。移除元素这个表述有一点误导，remove()是一个全局函数，所以它不能删除容器中的元素。remove()移除元素的方式和从字符串中移除空格的方式相似，都是通过用匹配元素右边的元素来覆盖匹配元素的方式移除元素。图 2-4 展示了这个过程：

```
std::vector<std::string> words { "one", "none", "some", "all", "none", "most", "many"};
auto iter = std::remove(std::begin(words), std::end(words), "none");
```

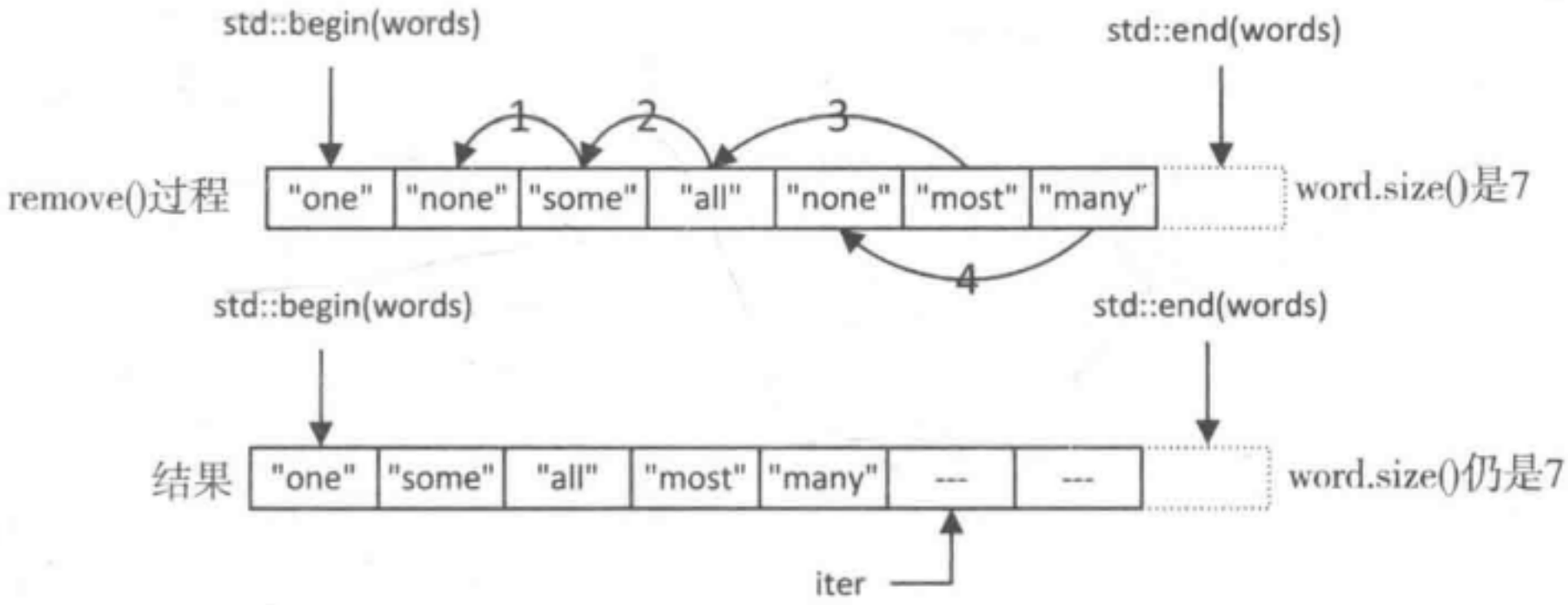


图 2-4　remove()算法的工作原理

如果在 remove()操作后输出 words 中的元素，只会输出前 5 个元素。尽管 size()返回的值仍然是 7，而且最后两个元素仍然存在，但是它们被替换成了空字符串对象。为了摆脱这些多余的元素，可以使用成员函数 erase()。remove()返回的迭代器可以这样使用：

```
words.erase(iter, std::end(words)); // Remove surplus elements
```

这被叫作 erase-remove，执行删除操作后，iter 指向最后一个元素之后的位置，所以它标识了被删除序列的第一个元素，被删除序列的结束位置由 std::end(words)指定。当然，在一条语句中，也能先移除元素，然后再删除末尾不想要的元素：

```
    words.erase(std::remove(std::begin(words), std::end(words), "none"),
std::end(words));
```

remove()算法返回的迭代器作为 erase()的第一个参数，erase()的第二个参数是所指向容器中最后一个元素后一个位置的迭代器。

了解如何为 vector 容器分配额外容量，可以让你明白会产生多少额外开销，以及可分配的内存量。下面是一个示例，可以让你了解到这一点：

```
// Ex2_03.cpp
// Understanding how capacity is increased in a vector container
#include <iostream>                     // For standard streams
#include <vector>                       // For vector container
int main()
{
  std::vector <size_t> sizes;           // Record numbers of elements
  std::vector <size_t> capacities;      // and corresponding capacities
  size_t el_incr {10}; // Increment to initial element count
  size_t incr_count {4 * el_incr};      // Number of increments to element
                                        // count
  for (size_t n_elements {}; n_elements < incr_count; n_elements += el_incr)
  {
    std::vector<int> values(n_elements);
```

```
        std::cout << "\nAppending to a vector with " << n_elements << " initial
elements:\n";
        sizes.push_back(values.size());
        size_t space {values.capacity()};
        capacities.push_back(space);

        // Append elements to obtain capacity increases
        size_t count {};                          // Counts capacity increases
        size_t n_increases {10};
        while (count < n_increases)
        {
          values.push_back(22);                   // Append a new element
          if (space < values.capacity())          // Capacity increased...
          { // ...so record size and capacity
            space = values.capacity();
            capacities.push_back(space);
            sizes.push_back(values.size());
            ++count;
          }
      }
      // Show sizes & capacities when increments occur
      std::cout << "Size/Capacity: ";
      for (size_t i {}; i < sizes.size(); ++i)
        std::cout << sizes.at(i) << "/" << capacities.at(i) << " ";
      std::cout << std::endl;
        sizes.clear();                            // Remove all elements
        capacities.clear();                       // Remove all elements
        }
    }
```

这个示例中的操作很简单。向容器中添加元素，直到不得不增加容器容量，这时候容器的大小和容量会被保存在sizes和capacities容器中。对不同初始元素个数的容器重复几次这种操作。编译器得到的输出结果如下：

```
    Appending to a vector with 0 initial elements:
    Size/Capacity: 0/0 1/1 2/2 3/3 4/4 5/6 7/9 10/13 14/19 20/28 29/42

    Appending to a vector with 10 initial elements:
    Size/Capacity: 10/10 11/15 16/22 23/33 34/49 50/73 74/109 110/163 164/244
245/366 367/549

    Appending to a vector with 20 initial elements:
    Size/Capacity: 20/20 21/30 31/45 46/67 68/100 101/150 151/225 226/337 338/505
    506/757 758/1135

    Appending to a vector with 30 initial elements:
    Size/Capacity: 30/30 31/45 46/67 68/100 101/150 151/225 226/337 338/505
506/757 758/1135 1136/1702
```

不同编译器的输出结果可能不同，这取决于用来增加 vector 容量的算法。从第一组的

输出可以看出，当开始使用一个空的vector 时，需要频繁地分配更多的内存，因为容量增量很小——内存从一个元素开始增加。其他组的输出表明，容量增量和容器的大小相关。每次分配，会额外分配相当于当前元素数目 50%的内存。这意味着在能够选择容器初始大小时，需要注意一些事情。

假设生成了一个初始容量为 1000 个元素的 vector，但实际上存储了 1001 个元素。这样就会有用于 499 个元素的多余容量。如果元素是数组或其他不会占用太多空间的对象，这不会有任何问题。但是如果对象非常大，例如每个 10KB，那么程序需要分配几乎 5MB 的多余内存。所以，最好可以稍微高估 vector 的初始大小，而不能低估。

当然，也能自己管理内存的分配。可以比较容器的大小和容量，当需要内存时，就可以通过容器的 reserve()函数来增加容器的容量。例如：

```
std::vector <size_t> junk {1, 2, 3};
for(size_t i {} ; i<1000 ; ++i)
{
  if(junk.size() == junk.capacity()) // When the size has reached the
                                     //capacity...
    junk.reserve(junk.size()*13/10); // ...increase the capacity
  junk.push_back(i);
}
```

这里容量增量为最大值的 30%而不是默认的 50%。容量增量不需要一定是当前大小的百分比。可以将 junk 指定为 reserve()的参数。例如 capacity()+10，无论当前大小为多少，都会将当前的容量增加 10 个元素大小。不要忘了当使用 reserve()为容器增加容量后，现有的容器迭代器都会失效。

2.3.7　vector<bool>容器

vector<bool>是 vector <T>模板的特例化，为 bool 类型的元素提供了更有效的内存使用方式。如何做到这一点是由实现定义的，通常布尔元素存储为 1 位。如果没有特例化，vector 容器中的每个布尔元素通常会占用 1 字节，甚至可能更多——这取决于实现方式。布尔值的序列通常不需要存储在连续的内存中，因此 vector<bool>并没有成员函数 data()。特例化的 vector<bool>的一些成员函数操作和一般模板实例有些不同。布尔值直接寻址时，它们被包装成一个字，因此 front()和 back()的返回值不是 bool&引用，而是一个中间对象的引用，它代表序列中第一个值和最后一个值。

bitset<N>类模板定义在 bitset 头文件中，当要用布尔值而且知道需要使用多少时，相对于使用 vector<bool>，bitset 是更好的选择。模板参数是位的个数。bitset 并不是一个容器——例如它没有迭代器，但是 bitset 实例提供了一些 vector<bool>不能提供的操作。因为 bitset 和 vector<bool>的应用往往都很专业，所以此处不会深入讨论它们。

2.4 使用 deque<T>容器

deque<T>是一个定义在 deque 头文件中的容器模板，可以生成包含 T 类型元素的容器，它以双端队列的形式组织元素。可以在容器的头部和尾部高效地添加或删除对象，这是它相对于 vector 容器的优势。当需要这种功能时，可以选择这种类型的容器。无论何时，当应用包含先入先出的事务处理时，都应该使用 deque 容器。处理数据库事务或模拟一家超市的结账队列，像这两种应用都可以充分利用 deque 容器。

2.4.1 生成 deque 容器

如果用默认的构造函数生成 deque 容器，容器中没有任何元素，因此添加第一个元素，就会导致内存的分配：

```
std::deque<int> a_deque;          // A deque container with no elements
```

可以生成给定元素个数的 deque 容器，这一点和 vector 容器在本质上相同：

```
std::deque<int> my_deque(10);   // A deque container with 10 elements
```

如图 2-5 所示，有一个存储 int 型元素、名为 my_deque 的 deque 容器。在这个容器中，保存了一些奇数元素：

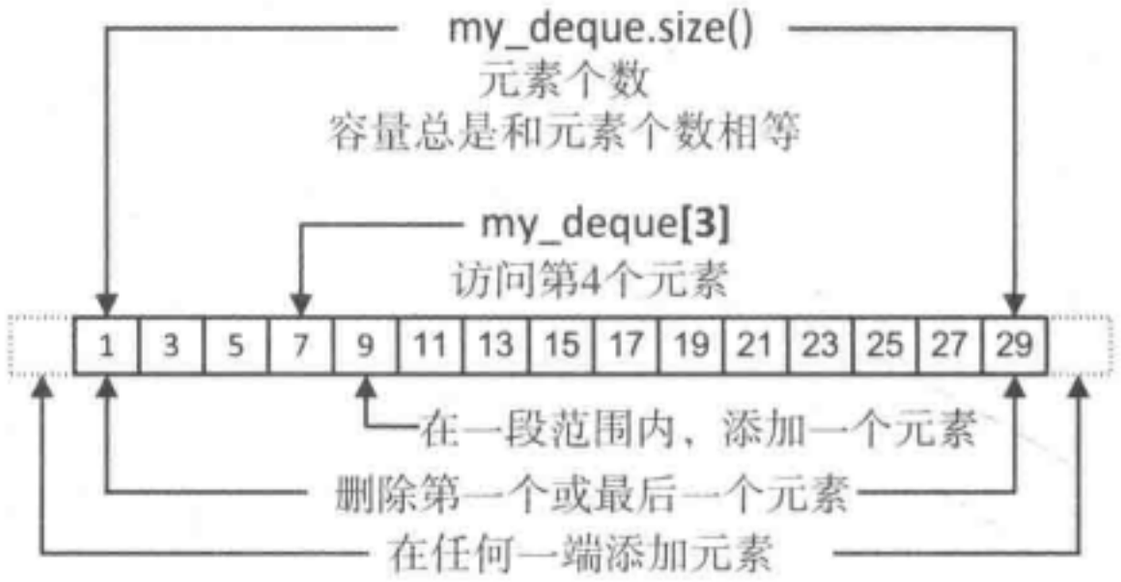


图 2-5 一个 deque 容器示例

当生成特定元素个数的 deque 时，每个元素保存的都是这种元素类型的默认值，因此前面定义的 my_deque 的所有元素的初始值都是 0。如果生成一个具有指定元素个数的 deque，每一个元素都会被构造函数 string()初始化。

也可以用初始化列表来生成 deque 容器：

```
    std::deque<std::string> words { "one", "none", "some", "all", "none", "most",
"many" };
```

这个容器将会有 7 个字符串元素，它们都是用初始化列表中的字符生成的。当然，也可以将初始化列表中的对象依次指定为 string("one")、string("none")等。

deque 容器也有拷贝构造函数，可以生成现有容器的副本：

```
std::deque<std::string> words_copy { words };
// Makes a copy of the words container
```

当生成一个 deque 容器时，也可以用由两个迭代器标识的一段元素来初始化它：

```
std::deque<std::string> words_part { std::begin(words), std::begin(words)
+ 5 };
```

这个容器有 5 个元素，它们和 words 容器的前 5 个元素相等。当然，作为初始值的一段元素，可以来自任何种类的容器，不需要一定是 deque。deque 提供了随机访问迭代器，所以能够以和 vector 相同的方式，从 deque 容器中获取 const 迭代器、non-const 迭代器、反向迭代器。

2.4.2　访问元素

可以使用下标运算符来访问 deque 容器中的元素。这个操作和 vector 容器中的类似，所以下标也没有做边界检查。deque 容器中的元素是序列，但是内部的存储方式和 vector 不同。它组织元素的方式导致容器的大小总是和容量相等。因为这个，所以没有定义成员函数 capacity()——deque 只有成员函数 size()，它以成员类型 size_type 的无符号整型来返回当前元素个数。同样因为 deque 内部组织元素的方式不同，deque 的操作和 vector 相比要慢。

可以用下标运算符来访问元素，但是索引并没有进行边界检查。为了用进行边界检查的索引来访问元素，可以选择使用成员函数 at()，这和 vector 相同：

```
std::cout << words.at(2) << std::endl; // Output the third element in words
```

参数必须是一个 size_t 类型的值，因此不能是负数。如果 at()的参数不在范围内，例如大于 words.size()-1，就会抛出一个 std::out_of_range 异常。

deque 成员函数 front()和 back()的用法也和 vector 相同，然而，deque 却没有成员函数 data(),因为元素并没有被存放在数组中。deque 容器和 vector 一样，有三种不同重载版本的 resize()函数，它们的操作基本相同。

2.4.3　添加和移除元素

deque 和 vector 都有成员函数 push_back()和 pop_back()，它们在序列尾部添加或删除元素的方式相同。deque 也有成员函数 push_front()和 pop_front()，可以在序列头部执行相似的操作。例如：

```
std::deque<int> numbers {2, 3, 4};
numbers.push_front(11);                 // numbers contains 11 2 3 4
numbers.push_back(12);                  // numbers contains 11 2 3 4 12
numbers.pop_front();                    // numbers contains 2 3 4 12
```

除了和 vector 一样都有 emplace_back()函数外，deque 还有成员函数 emplace_front()，可以在序列的开始位置生成新的元素。和 vector 一样，也可以使用 emplace()或 insert()在

deque 内部添加或移除元素。这个过程相对要慢一些，因为这些操作需要移动现有的元素。

本书所描述的关于 vector 容器的所有 insert()函数也同样适用于 deque 容器。在 deque 的任意位置插入一个元素会让现有的迭代器全部失效，因此需要重新生成它们。deque的成员函数 erase()也和 vector 的相同，它的成员函数 clear()可以移除一些元素。

2.4.4 替换 deque 容器中的内容

deque 的成员函数 assign()可以替换现有的所有元素。它有三个重版版本；替换的新内容可以是由初始化列表指定的元素，也可以是由迭代器指定的一段元素，或是一个特定对象的多个副本。这里展示了如何使用初始化列表来替换 deque 容器中的内容：

```
std::deque<std::string> words {"one", "two", "three", "four"};
auto init_list = {std::string{"seven"}, std::string{"eight"},
                  std::string{"nine"}};
words.assign(init_list);
```

最后一条语句用 init_list 中的 string 对象替换掉了 words 中的元素。注意，这里不能直接把字符放入初始化列表。如果这么做，init_list 的类型会被推导为 initializer_list<const char*>, 然而 assign()需要的是一个 initializer_list<string>类型的实参，这样就无法通过编译。当然，也可以不单独定义 init_list，可以在 assign()的实参中定义初始化列表，例如：

```
words.assign({"seven", "eight", "nine"});
```

因为 words 的成员函数 assign()需要一个 initializer_list<string>类型的实参，编译器会用字符串列表生成一个这种类型的初始化列表。为了给 deque 容器赋值，需要提供两个迭代器作为参数：

```
std::vector<std::string> wordset {"this", "that", "these", "those"};
words.assign(std::begin(wordset)+1, --std::end(wordset));
// Assigns "that" and "these"
```

assign()函数只需要输入迭代器，因此可以使用任何类别的迭代器。最后一种可能是，用重复的对象来替换容器中的内容：

```
words.assign(8, "eight"); // Assign eight instances of string("eight")
```

第一个参数指定了替换当前容器内容的第二个参数的个数。

vector 也提供了一套同样的 assign()函数，所以可以更换一套新的 vector 容器元素。

也可以使用赋值操作符来替换赋值运算符左边的 deque 容器的内容。赋值运算的右操作数必须和左操作数是相同类型，也可以是一个初始化列表。vector 容器同样也支持这些操作。下面是一个为 deque 替换一套新元素的示例：

```
std::deque<std::string> words {"one", "two", "three", "four"};
std::deque<std::string> other_words;
other_words = words; // other_words same contents as words
words = {"seven", "eight", "nine"}; // words contents replaced
```

执行完这些语句后，other_words 会包含和 words 相同的元素，words 则包含初始化列表中的那些元素。赋完值后，容器的大小会反映赋值元素的个数。为一个 vector 容器替换一套新的元素(来自于同类型的 vector 容器或初始化列表)，它的大小和这套新元素相同。

下面是一个使用 deque 容器的完整示例：

```
// Ex2_04.cpp
// Using a deque container
#include <iostream> // For standard streams
#include <algorithm> // For copy()
#include <deque> // For deque container
#include <string> // For string classes
#include <iterator> // For front_insert_iterator & stream
iterators

using std::string;

int main()
{
  std::deque<string> names;
  std::cout << "Enter first names separated by spaces. Enter Ctrl+Z on a new
line to end:\n";
  std::copy(std::istream_iterator<string> {std::cin}, std::istream_
    iterator <string> {},std::front_inserter(names));
  std::cout << "\nIn reverse order, the names you entered are:\n";
  std::copy(std::begin(names), std::end(names), std::ostream_iterator
    <string>{std::cout, " "});

  std::cout << std::endl;
}
```

下面是示例输出：

```
Enter first names separated by spaces. Enter Ctrl+Z on a new line to end:
Fred Jack Jim George Mary Zoe Rosie

^Z

In reverse order, the names you entered are:
Rosie Zoe Mary George Jim Jack Fred
```

该程序读入几个任意长度的字符串，然后把它们存储在 names 容器中。copy()算法将从 istream_iterator<string>获取到的序列，输入到 names 容器的 front_insert_iterator 中，这个迭代器是由函数 front_inserter()返回的。copy()的第一个参数是用来输入的开始迭代器，第二个参数是对应的结束迭代器。当使用键盘输入 Ctrl+Z 时，输入迭代器会对应为结束迭代器；如果是从文件流中读取数据，当读到 EOF 时，也会产生一个结束迭代器。因为 deque 容器有成员函数 push_front()，可以用来在序列的头部添加元素，所以我们这里可以使用 front_insert_iterator；front_insert_iterator 通过调用 push_front()在容器中添加元素，因此它适用于有成员函数 push_front()的任何容器。

输出也是由 copy()算法生成的。前两个参数是用来指定元素范围的迭代器，这些元素被复制到第三个参数所指定的位置。因为前两个参数正好是 deque 容器的开始和结束迭代器，因此会复制 deque 容器的全部元素。目的地是一个接收字符串对象的 ostream_iterator，它会将这些字符串对象写入标准输出流。

2.5 使用 list<T>容器

list<T>容器模板定义在 list 头文件中，是 T 类型对象的双向链表。list 容器具有一些 vector 和 deque 容器所不具备的优势，它可以在常规时间内，在序列已知的任何位置插入或删除元素。这是我们使用 list，而不使用 vector 或 deque 容器的主要原因。它的缺点是，无法通过位置来直接访问序列中的元素，也就是说，不能索引元素。为了访问 list 内部的一个元素，必须一个一个地遍历元素，通常从第一个元素或最后一个元素开始遍历。图 2-6 展示了 list 容器中的元素在概念上是如何组织的。

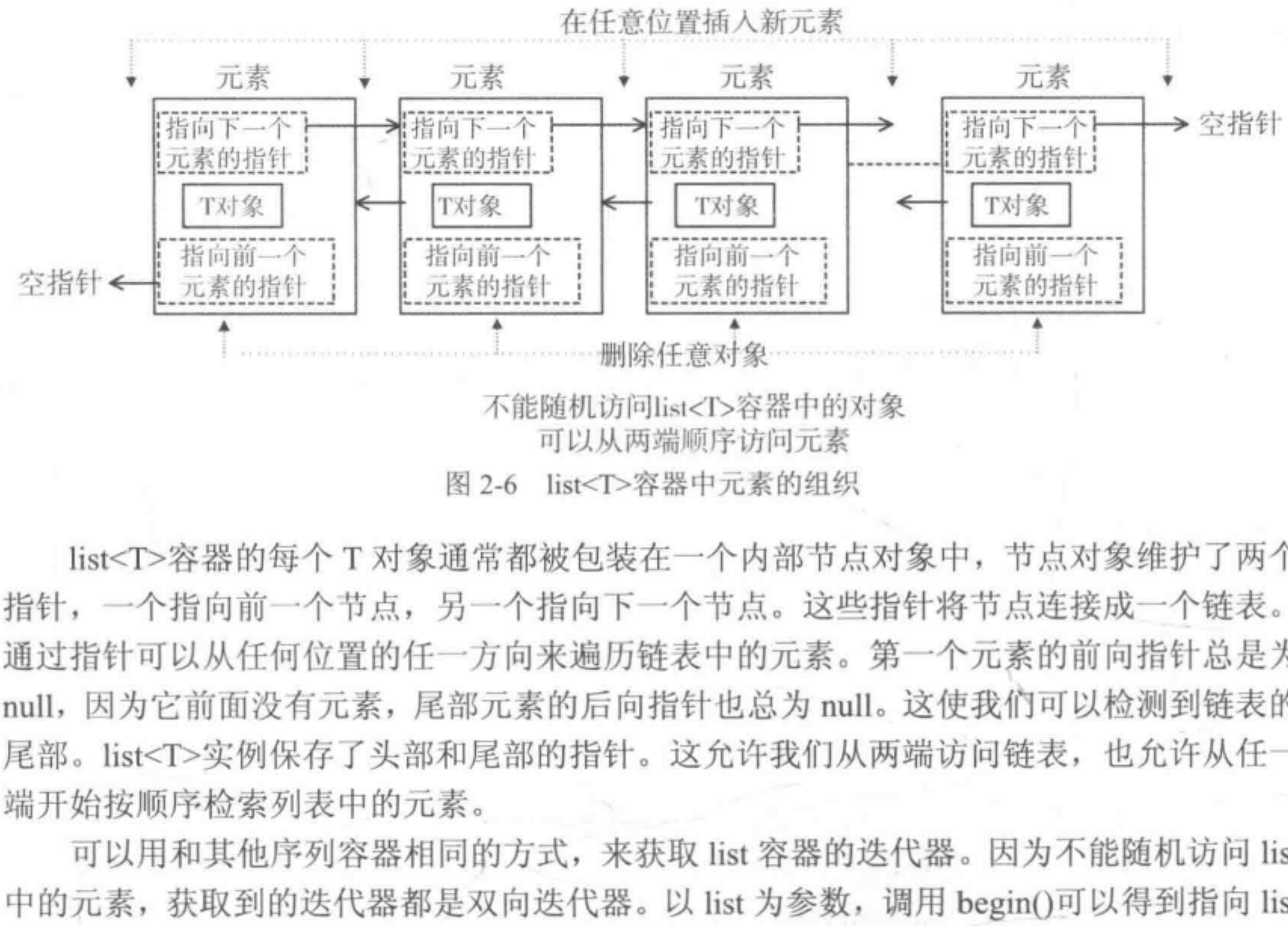


图 2-6 list<T>容器中元素的组织

list<T>容器的每个 T 对象通常都被包装在一个内部节点对象中，节点对象维护了两个指针，一个指向前一个节点，另一个指向下一个节点。这些指针将节点连接成一个链表。通过指针可以从任何位置的任一方向来遍历链表中的元素。第一个元素的前向指针总是为 null，因为它前面没有元素，尾部元素的后向指针也总为 null。这使我们可以检测到链表的尾部。list<T>实例保存了头部和尾部的指针。这允许我们从两端访问链表，也允许从任一端开始按顺序检索列表中的元素。

可以用和其他序列容器相同的方式，来获取 list 容器的迭代器。因为不能随机访问 list 中的元素，获取到的迭代器都是双向迭代器。以 list 为参数，调用 begin()可以得到指向 list 中第一个元素的迭代器。通过调用 end()，可以得到一个指向最后一个元素下一个位置的迭代器，因此像其他序列容器一样，可以用它们来指定整个范围的元素。也可以像其他容器一样，使用 rbegin()、rend()、crbegin()、crend()、cbegin()、cend()来获取反向迭代器和 const 迭代器。

2.5.1　生成 list 容器

list 容器的构造函数的用法类似于 vector 或 deque 容器。下面这条语句生成了一个空的 list 容器：

```
std::list<std::string> words;
```

可以创建一个带有给定数量的默认元素的列表：

```
std::list<std::string> sayings {20}; // A list of 20 empty strings
```

元素的个数由构造函数的参数指定，每一个元素都由默认的构造函数生成，因此这里调用 string()来生成元素。

下面展示如何生成一个包含给定数量的相同元素的列表：

```
std::list<double> values(50, 3.14149265);
```

这里生成了一个具有 50 个 double 型值的列表，并且每一个值都等于 π。注意在圆括号中，不能使用初始化列表——如果使用{50,3.14159265}，列表将仅仅包含两个元素。

list 容器有一个拷贝构造函数，因此可以生成一个现有 list 容器的副本：

```
std::list<double> save_values {values}; // Duplicate of values
```

可以用另一个序列的开始和结束迭代器所指定的一段元素，来构造 list 容器的初始化列表：

```
std::list<double> samples {++cbegin(values), --cend(values)};
```

除了 value 中的第一个和最后一个元素，其他元素都被用来生成列表。因为 list 容器的 begin()和 end()函数返回的都是双向迭代器，所以不能用它们加减整数。修改双向迭代器的唯一方式是使用自增或自减运算符。当然，在上面的语句中，初始化列表中的迭代器可以代表任意容器的一段元素，而不仅仅只是 list 容器。

可以通过调用 list 容器的成员函数 size()来获取它的元素个数。也可以使用它的 resize()函数来改变元素个数。如果 resize()的参数小于当前元素个数，会从尾部开始删除多余的元素。如果参数比当前元素个数大，会使用所保存元素类型的默认构造函数来添加元素。

2.5.2　添加元素

可以使用 list 容器的成员函数 push_front()在它的头部添加一个元素。调用 push_back()可以在 list 容器的末尾添加一个元素。在下面这两个示例中，参数作为对象被添加：

```
std::list<std::string> names {"Jane", "Jim", "Jules", "Janet"};
names.push_front("Ian");    // Add string("Ian") to the front of the list
names.push_back("Kitty");   // Append string("Kitty") to the end of the list
```

这两个函数都有右值引用参数的版本，这种版本的函数会移动参数而不是从参数复制

新的元素。它们显然要比其他使用左值引用参数的版本高效。然而，成员函数 emplace_front()和 emplace_back()可以做得更好：

```
names.emplace_front("Ian");
// Create string("Ian") in place at the front of the list
names.emplace_back("Kitty");
// Create string("Kitty") in place at the end of the list
```

这些成员函数的参数是调用以被生成元素的构造函数的参数。它们消除了 push_front()和 push_back()不得不执行的右值移动运算。

可以使用成员函数 insert()在 list 容器内部添加元素。像其他序列容器一样，它有三个版本。第一个版本可以在迭代器指定的位置插入一个新的元素：

```
std::list<int> data(10, 55); // List of 10 elements with value 55
data.insert(++begin(data), 66); // Insert 66 as the second element
```

insert()的第一个参数是一个指定插入点的迭代器，第二个参数是被插入的元素。begin()返回的双向迭代器自增后，指向第二个元素。上面的语句执行完毕后，list 容器的内容如下：

```
55 66 55 55 55 55 55 55 55 55 55
```

list 容器现在包含 11 个元素。插入元素不必移动现有的元素。生成新元素后，这个过程需要将 4 个指针重新设为适当的值。第一个元素的 next 指针指向新的元素，原来的第二个元素的 pre 指针也指向新的元素。新元素的 pre 指针指向第一个元素，next 指针指向序列之前的第二个元素。和 vector、deque 容器的插入过程相比，这个要快很多，并且无论在哪里插入元素，花费的时间都相同。

可以在给定位置插入几个相同元素的副本：

```
auto iter = begin(data);
std::advance(iter, 9);    // Increase iter by 9
data.insert(iter, 3, 88); // Insert 3 copies of 88 starting at the 10th
```

iter 是 list<int>::iterator 类型。insert()函数的第一个参数是用来指定插入位置的迭代器，第二个参数是被插入元素的个数，第三个参数是被重复插入的元素。为了得到第 10 个元素，可以使用定义在 iterator 头文件中的全局函数 advance()，将迭代器增加 9。只能增加或减小双向迭代器。因为迭代器不能直接加 9，所以 advance()会在循环中自增迭代器。执行完上面的代码后，list 容器的内容如下：

```
55 66 55 55 55 55 55 55 55 88 88 88 55 55
```

现在 list 容器包含 14 个元素。下面展示如何将一段元素插入到 data 列表中：

```
std::vector<int> numbers(10, 5); // Vector of 10 elements with value 5
data.insert(--(--end(data)), cbegin(numbers), cend(numbers));
```

insert()的第一个参数是一个迭代器，它指向 data 的倒数第二个元素。第二和第三个参数指定了 number 中被插入元素的范围，因此从 data 中倒数第二个元素开始，依次插入 vector

的全部元素。代码执行后，data 中的内容如下：

```
55 66 55 55 55 55 55 55 55 88 88 88 5 5 5 5 5 5 5 5 5 5 55 55
```

list容器现在有 24 个元素。从 numbers 中倒数第二个元素的位置开始插入元素，list 最右边两个元素的位置被替换了。尽管如此，指向最后两个元素的任何迭代器或结束迭代器都没有失效。list 元素的迭代器只会在它所指向的元素被删除时才会失效。

有三个函数可以在 list 容器中直接构造元素：emplace()在迭代器指定的位置构造一个元素；emplace_front()在 list 的第一个元素之前构造元素；emplace_back()在 list 的尾部元素之后构造元素。下面是一些用法示例：

```
std::list<std::string> names {"Jane", "Jim", "Jules", "Janet"};
names.emplace_back("Ann");
std::string name("Alan");
names.emplace_back(std::move(name));
names.emplace_front("Hugo");
names.emplace(++begin(names), "Hannah");
```

第 4 行代码用 std::move()函数将 name 的右值引用传入 emplace_back()函数。这个操作执行后，names 变为空，因为它的内容已经被移到 list 中。执行完这些语句后，names 中的内容如下：

```
"Hugo" "Hannah" "Jane" "Jim" "Jules" "Janet" "Ann" "Alan"
```

2.5.3　移除元素

对于 list 的成员函数 clear()和 erase()，它们的工作方式及效果，和前面的序列容器相同。list 容器的成员函数 remove()则移除和参数匹配的元素。例如：

```
std::list<int> numbers { 2, 5, 2, 3, 6, 7, 8, 2, 9};
numbers.remove(2);                  // List is now 5 3 6 7 8 9
```

第二条语句移除了 numbers 中出现的所有值等于 2 的元素。

成员函数 remove_if()期望传入一个一元断言作为参数。一元断言接受一个和元素同类型的参数或引用，返回一个布尔值。断言返回 true 的所有元素都会被移除。例如：

```
numbers.remove_if([](int n){return n%2 == 0;});
// Remove even numbers. Result 5 3 7 9
```

这里的参数是一个 lambda 表达式，但也可以是一个函数对象。

成员函数 unique()非常有意思，它可以移除连续的重复元素，只留下其中的第一个。例如：

```
    std::list<std::string> words {"one", "two", "two", "two", "three", "four",
"four"};
    words.unique(); // Now contains "one" "two" "three" "four"
```

这个版本的 unique()函数使用==运算符比较连续元素。可以在对元素进行排序后，再使用 unique()，这样可以保证移除序列中全部的重复元素。

重载的 unique()版本接受一个二元断言作为参数。使断言返回 true 的元素被认为是相等的。这提供了一个非常灵活的相等概念。然后就可以将两个长度或初始字符相同的字符串看作相等。这个断言可以有不同类型的参数，只要 list 迭代器解引用后的结果可以隐式地相互进行类型转换。

2.5.4 排序和合并元素

sort()函数模板定义在头文件 algorithm 中，要求使用随机访问迭代器。但 list 容器并不提供随机访问迭代器，只提供双向迭代器，因此不能对 list 中的元素使用 sort()算法。但是，还是可以进行元素排序，因为 list 模板定义了自己的 sort()函数。sort()有两个版本：无参 sort()函数将所有元素升序排列。第二个版本的 sort()接受一个函数对象或 lambda 表达式作为参数，这两种参数都定义一个断言用来比较两个元素。

下面是一个以断言作为参数调用 list 成员函数 sort()的示例：

```
names.sort(std::greater<std::string>()); // Names in descending sequence
```

这里使用了定义在头文件 functional 中的模板 greater<T>。这个模板定义了一个用来比较 T 类型对象的函数对象。如果第一个参数大于第二个参数，那么成员函数 operator()()返回 true。头文件 functional 中定义了一些定义了断言的模板，在后面，我们会看到更多的断言。排序执行完毕后，list 中的元素如下：

```
"Jules" "Jim" "Janet" "Jane" "Hugo" "Hannah" "Ann" "Alan"
```

因此，现在 names 中的元素是降序排列的。有一个简洁版的 greater<T>断言，可以如下所示使用：

```
names.sort(std::greater<>()); // Function object uses perfect forwarding
```

简洁版的函数对象可以接受任何类型的参数，使用完美转发(perfect forwarding)可以避免不必要的参数拷贝。因此，完美转发总是会快很多，因为被比较的参数会被移动而不是复制到函数中。

当然，在必要时可以将自定义的函数对象传给断言来对 list 排序。尽管对一般对象来说，并不需要这样。如果为自己的类定义了 operator>()，然后就可以继续使用 std::greater<>。当我们需要比较非默认类型时，就需要一个函数对象。例如，假设我们想对 names 中的元素进行排序，但是不想使用字符串对象标准的 std::greater<>来比较，而是想将相同初始字符的字符串按长度排序。可以如下所示定义一个类：

```
// Order strings by length when the initial letters are the same
class my_greater
{
public:
```

```
  bool operator()(const std::string& s1, const std::string& s2)
  {
    if (s1[0] == s2[0])
      return s1.length() > s2.length();
    else
      return s1 > s2;
  }
};
```

可以用这个来对 names 中的元素排序：

```
names.sort(my_greater()); // Sort using my_greater
```

代码执行完毕后，list 中包含的元素如下：

```
"Jules" "Janet" "Jane" "Jim" "Hannah" "Hugo" "Alan" "Ann"
```

这个结果和前面使用字符串对象标准比较方式的结果明显不同。names 中初始字符相同的元素按长度降序排列。当然，如果不需要重用 my_greater()断言，这里也可以使用 lambda 表达式。下面是一个用 lambda 表达式实现的示例：

```
names.sort([](const std::string& s1, const std::string& s2)
                { if (s1[0] == s2[0])
                    return s1.length() > s2.length();
                  else
                    return s1 > s2;
                });
```

这和前面代码的作用相同。

list 的成员函数 merge()以另一个具有相同类型元素的 list 容器作为参数。两个容器中的元素都必须是升序。参数 list 容器中的元素会被合并到当前的 list 容器中。例如：

```
std::list<int> my_values {2, 4, 6, 14};
std::list<int> your_values{ -2, 1, 7, 10};
my_values.merge(your_values);     // my_values contains: -2 1 2 4 6 7 10 14
your_values.empty();              // Returns true
```

元素从 your_values 转移到 my_values，因此，在执行完这个操作后，your_values 中就没有元素了。改变每个 list 节点的指针，在适当的位置将它们和当前容器的元素链接起来，这样就实现了元素的转移。list 节点在内存中的位置不会改变；只有链接它们的指针变了。在合并的过程中，两个容器中的元素使用 operator<()进行比较。在另一个版本的 merge()函数中，可以提供一个比较函数作为该函数的第二个参数，用来在合并过程中比较元素。例如：

```
    std::list<std::string> my_words {"three", "six", "eight"};
    std::list<std::string> your_words {"seven", "four", "nine"};
    auto comp_str = [](const std::string& s1, const std::string& s2){ return
s1[0]<s2[0]; };
```

```
my_words.sort(comp_str);                    // "eight" "six" "three"
your_words.sort(comp_str);                  // "four" "nine" "seven"
my_words.merge(your_words, comp_str);       // "eight" "four" "nine" "six"
                                            // "seven" "three"
```

这里的字符串对象比较函数是由 lambda 表达式定义的，这个表达式只比较第一个字符。比较的效果是，在合并的 list 容器中，"six"在"seven"之前。在上面的代码中，也可以无参调用 merge()，这样"seven"会在"six"之前，这是一般的排序。

list 容器的成员函数 splice()有几个重载版本。这个函数将参数 list 容器中的元素移动到当前容器中指定位置的前面。可以移动单个元素、一段元素或源容器的全部元素。下面是一个剪切单个元素的示例：

```
std::list<std::string> my_words {"three", "six", "eight"};
std::list<std::string> your_words {"seven", "four", "nine"};
my_words.splice(++std::begin(my_words), your_words,
                ++std::begin(your_words));
```

splice()的第一个参数是指向目的容器的迭代器。第二个参数是元素的来源。第三个参数是一个指向源 list 容器中被粘接元素的迭代器，它会被插入到第一个参数所指向位置之前。代码执行完中后，容器中的内容如下：

```
your_words: "seven," "nine"
my_words: "three," "four," "six," "eight"
```

当要粘接源 list 容器中的一段元素时，第 3 和第 4 个参数可以定义这段元素的范围。例如：

```
your_words.splice(++std::begin(your_words), my_words, ++
  std::begin(my_words),
  std::end(my_words));
```

上面的代码会将 my_words 从第二个元素直到末尾的元素，粘接到 your_words 的第二个元素之前。上面两个 list 容器的内容如下：

```
your_words: "seven", "four", "six", "eight", "nine"
my_words: "three"
```

下面的语句可以将 your_words 的全部元素粘接到 my_words 中：

```
my_words.splice(std::begin(my_words), your_words);
```

your_words 的所有元素被移到了 my_words 的第一个元素"three"之前。然后，your_words 会变为空。即使 your_words 为空，也仍然可以向它粘接元素：

```
your_words.splice(std::end(your_words), my_words);
```

现在，my_words 变为空，your_words 拥有全部元素。第一个参数也可以是 std::begin (your_words)，因为当容器为空时，它也会返回一个结束迭代器。

2.5.5　访问元素

list 的成员函数 front()和 back()，可以各自返回第一个和最后一个元素的引用。在空 list 中调用它们中的任意一个，结果是未知的，因此不要这样使用。可以通过迭代器的自增或自减来访问 list 的内部元素。正如我们所了解的那样，begin()和 end()分别返回的是指向第一个和最后一个元素下一个位置的双向迭代器。rbegin()和 rend()函数返回的双向迭代器，可以让我们逆序遍历元素。因为可以对 list 使用基于范围的循环，所以当我们想要处理所有元素时，可以不使用迭代器：

```
std::list<std::string> names {"Jane", "Jim", "Jules", "Janet"};
names.emplace_back("Ann");
std::string name("Alan");
names.emplace_back(std::move(name));
names.emplace_front("Hugo");
names.emplace(++begin(names), "Hannah");
for(const auto& name : names)
  std::cout << name << std::endl;
```

循环变量 name 是依次指向每个 list 元素的引用，使得循环能够逐行输出各个字符串。下面通过一个练习检验一下前面学到的知识。这个练习读取通过键盘输入的谚语并将它们存储到一个 list 容器中：

```
// Ex2_05.cpp
// Working with a list
#include <iostream>
#include <list>
#include <string>
#include <functional>

using std::list;
using std::string;

// List a range of elements
template<typename Iter>
void list_elements(Iter begin, Iter end)
{
  while (begin != end)
    std::cout << *begin++ << std::endl;
}

int main()
{
  std::list<string> proverbs;
  // Read the proverbs
  std::cout << "Enter a few proverbs and enter an empty line to end:"
            << std::endl;
  string proverb;
  while (getline(std::cin, proverb, '\n'), !proverb.empty())
    proverbs.push_front(proverb);
```

```
    std::cout << "Go on, just one more:" << std::endl;
    getline(std::cin, proverb, '\n');
    proverbs.emplace_back(proverb);
    std::cout << "The elements in the list in reverse order are:" << std::endl;
    list_elements(std::rbegin(proverbs), std::rend(proverbs));

    proverbs.sort(); // Sort the proverbs in ascending sequence
    std::cout << "\nYour proverbs in ascending sequence are:" << std::endl;
    list_elements(std::begin(proverbs), std::end(proverbs));

    proverbs.sort(std::greater<>()); // Sort the proverbs in descending sequence
    std::cout << "\nYour proverbs in descending sequence:" << std::endl;
    list_elements(std::begin(proverbs), std::end(proverbs));
}
```

下面是示例输出：

```
Enter a few proverbs and enter an empty line to end:
A nod is a good as a wink to a blind horse.
Many a mickle makes a muckle.
A wise man stands on the hole in his carpet.
Least said, soonest mended.

Go on, just one more:
A rolling stone gathers no moss.
The elements in the list in reverse order are:
A rolling stone gathers no moss.
A nod is a good as a wink to a blind horse.
Many a mickle makes a muckle.
A wise man stands on the hole in his carpet.
Least said, soonest mended.

Your proverbs in ascending sequence are:
A nod is a good as a wink to a blind horse.
A rolling stone gathers no moss.
A wise man stands on the hole in his carpet.
Least said, soonest mended.
Many a mickle makes a muckle.

Your proverbs in descending sequence:
Many a mickle makes a muckle.
Least said, soonest mended.
A wise man stands on the hole in his carpet.
A rolling stone gathers no moss.
A nod is a good as a wink to a blind horse.
```

因为输入的一系列谚语中包含空格，所以可以使用 getline()函数来输入。每个谚语都是单行读入，然后通过调用 push_front()函数，将它们作为新的 list 元素添加到 proverbs 容器中。这里，为了练习使用 emplace_back()，额外添加了一个谚语。由定义在 main()前面的函数模板 list_elements()产生输出结果。这个函数模板可以输出任意类型的支持流插入操

作的元素，只要元素的容器提供输出迭代器。上面的代码演示了如何将这个函数模板与前向迭代器和反向迭代器一起使用。

第一次调用proverbs的成员函数sort()时，没有提供参数，因此元素被默认排成升序。第二次调用时，提供了一个greater断言作为参数；这个模板和其他几个会在后面遇到的标准断言模板都定义在头文件functional中。表达式greater<>()定义了一个函数对象，这个函数对象可以用operator>()来比较对象，推导模板参数类型。结果，list中的元素变成了降序排列。还有其他一些对象，它们也定义了一些sort()会经常用到的断言，包括greater_equal<>()、less<>()和 less_equal<>()。从名称就可以看出它们是如何进行比较的。从这个示例的输入来看，一切都很符合我们的预期。

2.6　使用 forward_list<T>容器

forward_list 容器以单链表的形式存储元素。forward_list 的模板定义在头文件 forward_list 中。forward_list 和 list 最主要的区别是：它不能反向遍历元素；只能从头到尾遍历。forward_list 的单向链接性也意味着它会有一些其他的特性。首先，无法使用反向迭代器。只能从它得到 const 或 non-const 前向迭代器，这些迭代器都不能解引用，只能自增。其次，没有可以返回最后一个元素引用的成员函数 back()；只有成员函数 front()。最后，因为只能通过自增前面元素的迭代器来到达序列的终点，所以 push_back()、pop_back()、emplace_back()也无法使用。forward_list 的操作比 list 容器还要快，而且占用的内存更少，尽管它在使用上有很多限制，但仅这一点也足以让我们满意了。

forward_list 容器的构造函数的使用方式和 list 容器相同。forward_list 的迭代器都是前向迭代器。它没有成员函数 size()，因此不能用一个前向迭代器减去另一个前向迭代器，但是可以通过使用定义在头文件 iterator 中的 distance()函数来得到元素的个数。例如：

```
std::forward_list<std::string> my_words {"three", "six", "eight"};
auto count = std::distance(std::begin(my_words), std::end(my_words));
// Result is 3
```

distance()的第一个参数是一个开始迭代器，第二个参数是一个结束迭代器，它们指定了元素范围。当需要将前向迭代器移动多个位置时，advance()就派上了用场。例如：

```
std::forward_list<int> data {10, 21, 43, 87, 175, 351};
auto iter = std::begin(data);
size_t n {3};
std::advance(iter, n);
std::cout << "The " << n+1 << "th element is " << *iter << std::endl;
// Outputs 87
```

这并不神奇。advance()函数会将前向迭代器自增需要的次数。这使我们不必去循环自增迭代器。需要记住的是这个函数自增的是作为第一个参数的迭代器，但是并不会返回它——advance()的返回类型为 void。

因为 forward_list 正向链接元素，所以只能在元素的后面插入或粘接另一个容器的元素，这一点和 list 容器的操作不同，list 可以在元素前进行操作。因为这个，forward_list 包含成员函数 splice_after()和 insert_after()，用来代替 list 容器的 splice()和 insert()；顾名思义，元素会被粘接或插入到 list 中的一个特定位置。当需要在 forward_list 的开始处粘接或插入元素时，这些操作仍然会有问题。除了第一个元素，不能将它们粘接或插入到任何其他元素之前。这个问题可以通过使用成员函数 cbefore_begin()和 before_begin()来解决。它们分别可以返回指向第一个元素之前位置的 const 和 non-const 迭代器。所以可以使用它们在开始位置插入或粘接元素。例如：

```
std::forward_list<std::string> my_words {"three", "six", "eight"};
std::forward_list<std::string> your_words {"seven", "four", "nine"};
my_words.splice_after(my_words.before_begin(), your_words, ++std::begin
(your_words));
```

这个操作的效果是将 your_words 的最后一个元素粘接到 my_words 的开始位置，因此 my_words 会包含这些字符串对象——"nine"、"three"、"six"、"eight"。这时 your_words 就只剩下两个字符串元素："seven"和"four"。

还有另一个版本的 splice_after()，它可以将 forward_list<T>的一段元素粘接到另一个容器中：

```
my_words.splice_after(my_words.before_begin(), your_words,
    ++std::begin(your_words), std::end(your_words));
```

最后两个迭代器参数，指定了第三个参数所指定的 forward_list<T>容器的元素范围。在这个范围内的元素，除了第一个，其他的都被移到第一个参数所指定容器的特定位置。因此，如果在容器初始化后执行这条语句，my_words 会包含"four"、"nine"、"three"、"six"、"eight"，your_words 仅仅包含 "seven"。

另一个版本的 splice_after()会将一个 forward_list<T>容器的全部元素粘接到另一个容器中：

```
my_words.splice_after(my_words.before_begin(), your_words);
```

上面的代码会将 your_words 中的全部元素拼接到第一个元素指定的位置。

forward_list 和 list 一样都有成员函数 sort()和merge()，它们也都有 remove()、remove_if()和 unique()，所有这些函数的用法都和 list 相同。我们可以尝试在一个示例中演示 forward_list 的操作。容器会包含一些代表矩形盒子的 Box 类对象。下面是 Box 类的头文件中的内容：

```
// Box.h
// Defines the Box class for Ex2_06
#ifndef BOX_H
#define BOX_H
#include <iostream>                 // For standard streams
#include <utility>                  // For comparison operator templates
```

```
using namespace std::rel_ops;  // Comparison operator template namespace

class Box
{
private:
  size_t length {};
  size_t width {};
  size_t height {};
public:
  explicit Box(size_t l=1, size_t w=1, size_t h=1) : length {l}, width {w},
     height {h} {}
  double volume() const { return length*width*height; }
  bool operator<(const Box& box) { return volume() < box.volume(); }
  bool operator==(const Box& box) { return length == box.length && width ==
     box.width && height == box.height; }

  friend std::istream& operator>>(std::istream& in, Box& box);
  friend std::ostream& operator<<(std::ostream& out, const Box& box);
};

inline std::istream& operator>>(std::istream& in, Box& box)
{
  std::cout << "Enter box length, width, & height separated by spaces - Ctrl+Z
     to end: ";
  size_t value;
  in >> value;
  if (in.eof()) return in;

  box.length = value;
  in >> value;
  box.width = value;
  in >> value;
  box.height = value;
  return in;
}

inline std::ostream& operator<<(std::ostream& out, const Box& box)
{
  out << "Box(" << box.length << "," << box.width << "," << box.height << ") ";
  return out;
}
#endif
```

utility 头文件中的命名空间 std::relops 包含一些比较运算符的模板。如果一个类已经定义了 operator<()和 operator==()，那么在需要时，这个模板会生成剩下的比较运算符函数。Box 类有三个私有成员，它们定义了 Box 对象的整体尺寸。带默认值的构造函数提供了一个无参构造函数，当在容器中保存 Box 对象时，需要使用它；没有初始化的元素由这种元素默认的构造函数生成。内联的友元函数重载了流的提取和插入运算，这显然包含标准输入输出流。每次以三个输入值为一组，在读入第一个输入值后，operator>>()函数会通过调

用流对象的 eof()函数来检查是否读到 EOF。当输入 Ctrl+Z 或从文件输入流中读到文件结束符时，标准输入流会被设置为 EOF。当这些发生时会结束输入，然后返回一个流对象，EOF 状态会继续保持，因此调用程序可以检测到这个状态。

下面是一个在 forward_list 容器中保存 Box 对象的示例：

```
// Ex2_06.cpp
// Working with a forward list
#include <algorithm>                    // For copy()
#include <iostream>                     // For standard streams
#include <forward_list>                 // For forward_list container
#include <iterator>                     // For stream iterators
#include "Box.h"

// List a range of elements
template<typename Iter>
void list_elements(Iter begin, Iter end)
{
  size_t perline {6};                   // Maximum items per line
  size_t count {};                      // Item count
  while (begin != end)
  {
    std::cout << *begin++;
    if (++count % perline == 0)
    {
      std::cout << "\n";
    }
  }
  std::cout << std::endl;
}

int main()
{
  std::forward_list<Box> boxes;
  std::copy(std::istream_iterator<Box>(std::cin), std::istream_
    iterator<Box>(),std::front_inserter(boxes));

  boxes.sort(); // Sort the boxes
  std::cout << "\nAfter sorting the sequence is:\n";
  // Just to show that we can with Box objects - use an ostream iterator
  std::copy(std::begin(boxes), std::end(boxes), std::ostream_iterator
    <Box>(std::cout, " "));
  std::cout << std::endl;

  // Insert more boxes
  std::forward_list<Box> more_boxes {Box {3, 3, 3}, Box {5, 5, 5}, Box {4,
    4, 4}, Box {2, 2, 2}};
  boxes.insert_after(boxes.before_begin(), std::begin(more_boxes), std::
    end(more_boxes));
  std::cout << "After inserting more boxes the sequence is:\n";
  list_elements(std::begin(boxes), std::end(boxes));
```

```
    boxes.sort(); // Sort the boxes
    std::cout << std::endl;
    std::cout << "The sorted sequence is now:\n";
    list_elements(std::begin(boxes), std::end(boxes));

    more_boxes.sort();
    boxes.merge(more_boxes); // Merge more_boxes
    std::cout << "After merging more_boxes the sequence is:\n";
    list_elements(std::begin(boxes), std::end(boxes));

    boxes.unique();
    std::cout << "After removing successive duplicates the sequence is:\n";
    list_elements(std::begin(boxes), std::end(boxes));

    // Eliminate the small ones
    const double max_v {30.0};
    boxes.remove_if([max_v](const Box& box){ return box.volume() < max_v; });
    std::cout << "After removing those with volume less than 30 the sorted
      sequence is:\n";
    list_elements(std::begin(boxes), std::end(boxes));
}
```

示例输出如下：

```
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 4 4 5
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 6 5 7
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 2 2 3
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 1 2 3
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 3 3 4
Enter box length, width, & height separated by spaces - Ctrl+Z to end: 3 3 3
Enter box length, width, & height separated by spaces - Ctrl+Z to end: ^Z
After sorting the sequence is:
Box(1,2,3) Box(2,2,3) Box(3,3,3) Box(3,3,4) Box(4,4,5) Box(6,5,7)
After inserting more boxes the sequence is:
Box(3,3,3) Box(5,5,5) Box(4,4,4) Box(2,2,2) Box(1,2,3) Box(2,2,3)
Box(3,3,3) Box(3,3,4) Box(4,4,5) Box(6,5,7)

The sorted sequence is now:
Box(1,2,3) Box(2,2,2) Box(2,2,3) Box(3,3,3) Box(3,3,3) Box(3,3,4)
Box(4,4,4) Box(4,4,5) Box(5,5,5) Box(6,5,7)
After merging more_boxes the sequence is:
Box(1,2,3) Box(2,2,2) Box(2,2,2) Box(2,2,3) Box(3,3,3) Box(3,3,3)
Box(3,3,3) Box(3,3,4) Box(4,4,4) Box(4,4,4) Box(4,4,5) Box(5,5,5)
Box(5,5,5) Box(6,5,7)
After removing successive duplicates the sequence is:
Box(1,2,3) Box(2,2,2) Box(2,2,3) Box(3,3,3) Box(3,3,4) Box(4,4,4)
Box(4,4,5) Box(5,5,5) Box(6,5,7)
After removing those with volume less than 30 the sorted sequence is:
Box(3,3,4) Box(4,4,4) Box(4,4,5) Box(5,5,5) Box(6,5,7)
```

函数模板 list_elements()用来输出由开始和结束迭代器指定的元素，每 6 个元素一行。

这里用来输出 main()中的 forward_list 的内容。在 main()中，首先使用 copy()算法，以 istream_iterator<Box>对象作为数据源，以 front_inserter 作为数据存放地，从 cin 中读入一些 Box 对象的尺寸。istream_iterator 会调用定义在 Box.h 中的函数 operator>>()来读取 Box 对象。

front_inserter 会调用容器的成员函数 push_front()，这是为 forward_list 所做的工作。对 boxes 容器中的元素进行排序后，我们会通过 copy()算法将元素转移到 ostream_iterator<Box>对象中，然后输出它们。这个迭代器会调用定义在 Box.h 中的函数 operator<<()。这里有一个限制，我们无法控制每行元素输出的个数。剩下的代码是用模板 list_elements()输出的实例。

另一个 forward_list 容器 more_boxes 的内容被插入到 boxes 容器的头部。这是通过以函数 before_begin()返回的迭代器作为插入位置，然后调用函数 insert_after()来实现的。

下一步操作是对 boxes 中的元素进行排序，然后将 more_boxes 的内容合并到 boxes 中。这两个容器在调用 merge()前都必须先排序，因为只有在两个容器都是升序时，这个操作才能正常进行。因为插入新元素，显然会使 boxes 中出现一些重复的元素。调用 boxes 的函数 unique()就可以消除连续重复的元素。最后一个操作是调用 boxes()的函数 remove_if()来删除容器中的一些元素。由作为参数传入的一元断言来决定删除哪些元素。这里使用的是一个 lambda 表达式，如果元素体积小于 max_v，就返回 true。max_v 是从外部区域以值的方式捕获的，因此外部区域和表达式内部的值可能不同。由输出可以看出，所有的操作都符合预期。

2.7 自定义迭代器

如果觉得本章内容很难，就不要让自己陷入困境，因为对于本书后面的内容来说，不需要明白本节的内容。可以直接跳过它，学习第 3 章。但是，本章可以提供一个洞察 STL 迭代器结构以及感受模板强大功能的机会。迭代器对于任何自定义的类序列都是一个强大的附加工具。它允许我们将算法运用到有自定义类元素的容器上。可能会出现一种情形——没有可以满足我们需要的标准 STL 容器，这时候就需要定义一个自己的容器。我们的容器类可能需要迭代器。通过深入理解什么样的类(定义了迭代器)才能被 STL 所接受，可以让我们了解到 STL 内部发生了些什么。

2.7.1 STL 迭代器的要求

STL 对定义了迭代器的类类型有一些特定的要求。这是为了保证所有接受这种迭代器的算法都可以正常工作。算法不需要知道，也不在乎它所处理的元素来自何种容器，但是它们在意传给它们作为参数的迭代器的特性。不同的算法要求不同功能的迭代器。我们在第 1 章看到过这几类迭代器：输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机访问迭代器。我们总是可以在需要低级别迭代器的地方使用高级别迭代器。

定义算法的模板需要决定可传入迭代器的类别，用来验证所传入的迭代器的功能是否足够。知道迭代器参数的类别能为算法的应用提供潜在的优势，可以充分利用任何最少的

额外功能让算法更加高效。一般来说，这样做可能的，必须用标准的方式确认迭代器的功能。不同类别的迭代器意味着需要为迭代器类定义不同的成员函数集。我们已经知道，迭代器类别具有功能叠加性，这当然也会反映到每种类别的成员函数集上。在探讨这些之前，先介绍一下函数模板如何使用迭代器。

使用 STL 迭代器存在的问题

定义一个参数中有迭代器的函数模板会产生一个问题，我们并不总是知道在函数模板中要用到哪些类型的迭代器。思考下面用迭代器作为参数的交换函数：模板类型参数指定了迭代器类型：

```
template <typename Iter> void my_swap(Iter a, Iter b)
{
  tmp = *a;                    // error -- variable tmp undeclared
  *a = *b;
  *b = tmp;
}
```

函数模板的实例用来交换迭代器参数所指向的两个对象：a 和 b。tmp 应该是什么类型？我们没法知道——但知道迭代器指向对象的类型，但是我们却无计可施，因为直到类模板生成实例时，才能确定对象的类型。在不知道对象的类型时，如何定义变量？当然，这里可以使用 auto。在一些情况下，我们也想知道迭代器类型的值和类型差别。

有些其他的机制可以确定一个迭代器参数所指向值的类型。一种是，坚持要求每个使用 my_swap()的迭代器都应该包含一个公共定义的类型别名，因为这样就可以确定迭代器所指向对象的类型。既然这样，就可以在迭代器类中使用 value_type 的别名来指定函数模板 my_swap()中 tmp 的类型，如下所示：

```
template <typename Iter> void my_swap(Iter a, Iter b)
{
  typename Iter::value_type tmp = *a; // Better - but has limitations...
  *a = *b;
  *b = tmp;
}
```

因为 value_type 的别名定义在 Iter 类中，所以可以通过用类名限定 value_type 的方式引用它。这样定义了 value_type 别名的迭代器类就能在函数中正常使用。然而，算法既使用指针，也使用迭代器；如果 Iter 是普通类型的指针，例如 int*，甚至是 Box*，而 Box 是类型——这样可能就无法使用了。因为指针不是类，不能包含定义的别名，所以不能写成 int*::value_type 或 Box*::value_type。STL 用模板优雅地解决了这个问题和其他一些相关问题！

2.7.2　走进 STL

模板类型 iterator_traits 定义在头文件 iterator 中。这个模板为迭代器的类型特性定义了

一套标准的类型别名。关键是要解决前一节的难题，而且要让算法既可以用迭代器，也可以用一般的指针。iterator_traits 模板的定义如下所示：

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

相信你肯定记得，结构体和类在本质上是相同的，除了结构体的成员默认是 public。这个结构体模板中没有成员变量和成员函数。iterator_traits 模板的主体只包含类型别名的定义。这些别名以 Iterator 作为类型参数的模板。它在模板的类型别名——difference_type、value_type 等，以及用来生成迭代器模板实例的类型，与对应 Iterator 的类型参数之间定义了映射。因此对于一个实体类 Boggle，iterator_traits<Boggle>实例定义 difference_type 作为 Boggle::difference_type 的别名，定义 value_type 作为 Boggle::value_type 的别名，等等。

这帮我们有效地解决了不知道模板定义中类型是什么的问题。首先，假设定义了一个迭代器类型 MyIterator，它包含具有下列类型别名的定义：

- difference_type——两个 MyIterator 类型的迭代器之间差别值的类型。
- value_type——MyIterator 类型的迭代器所指向值的类型。
- pointer——MyIterator 类型的迭代器所表示的指针类型。
- reference——来自于*MyIterator 的引用类型。
- iterator_category——第 1 章所介绍的迭代器类别的标签类类型：它们是 input_iterator_tag、output_iterator_tag、forward_iterator_tag、bidirectional_iterator_tag、random_access_iterator_tag。

一个满足 STL 要求的迭代器类必须全部定义这些别名。但是对于输出迭代器，除了 iterator_category，所有的别名都可以定义为 void。这是因为输出迭代器指向对象的目的地址而不是对象。这套迭代器提供了我们所想知道的关于迭代器的一切。

当以迭代器为参数定义函数模板时，可以在模板中使用 iterator_traits 模板定义的标准类型别名。因此类型 MyIterator 的迭代器代表的指针类型总是可以作为 std::iterator_traits<MyIterator>::pointer 引用，因为它等同于 MyIterator::pointer。当需要指定一个 MyIterator 迭代器所指向值的类型时，可以写作 std::iterator_traits<MyIterator>:: value_type，这将会被映射为 MyIterator::value_type。我们用 my_swap()模板中的 iterator_traits 模板类型别名来指定 tmp 的类型，例如：

```
template <typename Iter>
void my_swap(Iter a, Iter b)
{
  typename std::iterator_traits<Iter>::value_type tmp = *a;
  *a = *b;
  *b = tmp;
```

```
}
```

上述代码将 tmp 的类型指定为 iterator_traits 模板中的 value_type 别名。当用 Iter 模板参数实例化 my_swap()模板时，tmp 的类型变为迭代器所指向的类型 Iter::value_type。

为了说清楚发生了什么，以及是如何解决这个问题的，让我们考虑一个 my_swap()模板实例的具体情况。假设一个程序包含下面的代码：

```
std::vector<std::string> words {"one", "two", "three"};
my_swap(std::begin(words), std::begin(words)+1); // Swap first two elements
```

当编译器遇到 my_swap()调用时，它会生成一个基于调用参数的函数模板实例。模板类型的迭代器是 iterator<std::string>。在 my_swap()模板的主体中，编译器不得不处理 tmp 的定义，编译器知道模板的类型参数是 iterator<std::string>，因此在向模板添加了这个类型后，tmp 的定义变为：

```
typename std::iterator_traits< iterator<std::string> >::value_type tmp = *a;
```

tmp 的类型现在是一个 iterator_traits 模板实例的成员变量。为了弄清楚这意味着什么，编译器需要使用 my_swap()函数中用来指定 tmp 类型的类型参数来实例化 iterator_traits 模板。下面是一个编译器将会生成的 iterator_traits 模板实例：

```
struct iterator_traits
{
typedef typename iterator<std::string>::difference_type difference_type;
typedef typename iterator<std::string>::value_type value_type;
typedef typename iterator<std::string>::pointer pointer;
typedef typename iterator<std::string>::reference reference;
typedef typename iterator<std::string>::iterator_category iterator_category;
};
```

从这里编译器可以确定 tmp 的类型为 iterator_traits<iterator<std::string>>::value_type，然而它也是 iterator<std::string>::value_type 的别名。就像所有的 STL 迭代器类型，iterator<std::string>类型的定义是从 iterator 模板中生成的，并且会包含 value_type 的定义，看起来像下面这样：

```
typedef std::string value_type;
```

现在编译器从 iterator_traits 实例中知道 iterator_traits<iterator<std::string>>::value_type 是 iterator<std::string>::value_type 的别名，并且从 iterator<std::string>类定义中知道 iterator<std::string>::value_type 是 std::string 的别名。通过将别名转换为真实类型，编译器推断出 my_swap()函数中 tmp 的定义是：

```
std::string tmp = *a;
```

有必要提醒自己模板不是代码——它是编译器用来生成代码的配方。iterator_traits 模板只包含类型别名，因此不会产生可执行代码。编译器在生成 C++代码的过程中，会用到它。

被编译的代码中将不会有 iterator_traits 模板的踪迹；它的唯一用武之地是在生成 C++代码的过程中。

这里仍然遗留了一些有关指针的问题。iterator_traits 如何让算法像接受迭代器一样接受指针。iterator_traits 模板特化了类型 T*和 const T*的定义。例如，当模板类型参数是指针类型 T*时，特化被定义为：

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

当模板类型参数是指针类型时，这定义了对应于别名的类型。T*类型的指针 value_type 的别名总是为 T；如果将 Box*类型的指针作为 my_swap()的参数，那么 value_type 的别名是 Box，因此 tmp 也为这种类型。随机访问迭代器类别所要求的全部操作都可以运用到指针上。因此对于指针，iterator_categor 的别名总是等同于 std::random_access_ iterator_tag 类型。因而 iterators_traits 能否正常工作取决于模板类型参数是指针还是迭代器类类型。当模板类型参数是指针时，会选择使用 iterators_traits 针对指针的特例化模板；否则选择标准的模板定义。

1. 使用迭代器模板

STL 定义了迭代器模板，用来帮助我们在自己的迭代器类中包含要求的类型别名。iterator 是一个结构体模板，它定义了 5 个来自于 iterator_traits 模板的类型别名：

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

这个模板定义了 STL 对迭代器所要求的全部类型。例如，如果有一个未知类型的模板参数 Iter，当需要声明一个指针时，它指向一个迭代器解引用时提供的类型，这时可以写作 Iter::pointer。iterator_category 的值必定是在第 1 章介绍的类别标签类中的一个。当定义一个表示迭代器的类时，可以使用以迭代器模板为基类生成的实例，这样会添加类需要的类型别名。例如：

```
    class My_Iterator : public std::iterator<std::random_access_iterator_tag,
int>
    {
      // Members of the iterator class...
    };
```

还需要注意，需要为迭代器定义 STL 要求的全部类型。模板的第 1 个参数指定了作为完全随机访问迭代器的迭代器类型。第 2 个参数是迭代器所指向对象的类型。最后的 3 个参数是默认值，因此第 3 个参数和这两个迭代器的类型不同，是 ptrdiff_t。第 4 个参数是一个指向对象的指针类型，因此是 int*。最后一个模板参数指定了引用的类型，是 int&。当然，迭代器类型不做任何事，仍然需要定义类的全部成员。

2. STL 迭代器成员函数的要求

STL 定义了一套需要迭代器类型支持并且依赖迭代器类别的成员函数。如果把它们编成组，显然很有用。第一组需要全部迭代器和一些所有迭代器类都需要的重要函数：默认构造函数、拷贝构造函数以及赋值运算符。根据经验，如果为迭代器类定义了其中任意一个函数，就需要定义一个显式的析构函数。该组中类型迭代器的全套函数是：

```
Iterator();                                  // Default constructor
Iterator(const Iterator& y);                 // Copy constructor
~Iterator();                                 // Destructor
Iterator& operator=(const Iterator& y);      // Assignment operator
```

对于随机访问迭代器类，STL 需要一整套的关系运算符。事实上，可以通过使用 utility 标准库头文件中的函数模板来完成这些定义：

```
bool operator==(const Iterator& y) const;
bool operator<(const Iterator& y) const;
```

下面假定已经直接#include 头文件 utility，并且直接使用命名空间 std::relops：

```
#include <utility>
using namespace std::rel_ops;
```

如果为类定义了 operator==()和 operator<()，然后 std 中声明的命名空间 rel_ops 在必要时，就可以包含我们为!=、>、>=和<=生成的运算符函数的函数模板。因此直接用 using 激活 std::rel_ops，就可以显式保存这 4 个运算符函数。如果定义一个运算符函数，但是它已经被命名空间 std::rel_ops 模板生成，那么我们的实现的优先级高于模板生成的实现。函数 operator()比较特别，叫作顺序关系，它对搜索和比较算法很重要。

函数 operator==()可以检验两个容器对象的内容是否相同。这是一个有趣的方面，对于任意一对操作数 x 和 y，表达式(x<y ||y<x || x==y)总为真，因为这个表达式的三部分中总有一部分为真。事实上，不需要那样做，如果 x==y 为真，那么 x<y 和 y<x 都不可能为真。唯一可以确定的是两个相等元素没有什么不同。然而，如果 x!=y，就无法断定 x<y 和 y<x 中的哪个为真。当表达式(!(x<y))&&(!(y<x))为真时，就可以说 x 和 y 不相等，也就意味着

在排序时没有偏向。这种情况的一个常见示例是对字符串进行排序时，但是要忽略大小写。如果以大小写敏感为基准，字符串"A123"和"a123"是不等价的(第一个字母不相同)，它们不相同，也不相等。

迭代器的其他操作由它的类别决定。可以在第 1 章看到每种迭代器的特定操作，而且因为迭代器的累加特性，随机访问迭代器可以支持全部操作。

让我们在示例中看一个简单迭代器类型的定义。我们定义一个类模板，用来表示一段数值类型值，也可以生成指定范围的开始和结束迭代器。这个迭代器也是模板类型，两个模板都定义在同一个头文件 Numeric_Range.h 中。下面是 Numeric_Range<T>模板的定义：

```
template <typename T> class Numeric_Iterator; // Template type declaration

// Defines a numeric range
template<typename T>
class Numeric_Range
{
  static_assert(std::is_integral<T>::value || std::is_floating_point
    <T>::value,"Numeric_Range type argument must be numeric.");

  friend class Numeric_Iterator <T>;

private:
  T start;                          // First value in the range
  T step;                           // Increment between successive values
size_t count;                       // Number of values in the range

public:
  explicit Numeric_Range(T first=0, T incr=1, size_t n=2) :
                                  start {first}, step {incr}, count {n}{}

  // Return the begin iterator for the range
  Numeric_Iterator<T> begin(){ return Numeric_Iterator<T>(*this); }

  // Return the end iterator for the range
  Numeric_Iterator<T> end()
  {
    Numeric_Iterator<T> end_iter(*this);
    end_iter.value = start + count*step; // End iterator value is one step
                                         // over the last
    return end_iter;
  }
};
```

类型参数 T 是序列的值类型，因此它必定是数值类型。对于模板主体中的函数 static_assert()，当 T 不是整型也不是浮点型时，它的第一个参数会为 false，这时会生成一条包含第二个字符串参数的编译时错误消息。这里使用的断言模板定义在头文件 type_traits 中，模板中还有一些其他的编译时模板类型参数检查断言。这个构造函数的三个参数都有

默认值，因此它也可以作为无参构造函数。这三个参数分别用来初始化值、指定一个值到另一个值的增量，以及指定值的个数。因此默认定义了又有两个值的元素段：0 和 1。编译器会在需要时，提供适当的拷贝构造函数。

另有两个成员函数生成，然后返回元素段的开始和结束迭代器。结束迭代器的成员变量 value 的值为最后一个 value+1。结束迭代器是通过修改开始迭代器的 value 生成的。Numeric_Iterator<T>模板类型的声明在其定义之前是必要的，因为还没有定义迭代器类型模板。Numeric_Iterator<T>模板被指定为这个模板的友元类，这样Numeric_Iterator<T>的实例就可以访问 Numeric_Range<T>的私有成员。Numeric_Range<T>模板也需要成为Numeric_Iterator<T>的友元类，因为Numeric_Range<T>的成员函数end()需要访问Numeric_Iterator<T>的一个私有成员。

这个迭代器的模板类型定义如下：

```
// Iterator class template - it's a forward iterator
template<typename T>
class Numeric_Iterator : public std::iterator <std::forward_iterator_tag, T>
{
  friend class Numeric_Range <T>;
private:
  Numeric_Range<T>& range;     // Reference to the range for this iterator
  T value;                     // Value pointed to
public:
  explicit Numeric_Iterator(Numeric_Range<T>& a_range) :
                              range {a_range}, value {a_range.start} {}
// Assignment operator
Numeric_Iterator& operator=(const Numeric_Iterator& src)
{
  range = src.range;
  value = src.value;
}

// Dereference an iterator
T& operator*()
{
  // When the value is one step more than the last, it's an end iterator
  if (value == static_cast<T>(range.start + range.count*range.step))
  {
    throw std::logic_error("Cannot dereference an end iterator.");
  }
  return value;
}

// Prefix increment operator
Numeric_Iterator& operator++()
{
   // When the value is one step more than the last, it's an end iterator
  if (value == static_cast<T>(range.start + range.count*range.step))
  {
```

```
        throw std::logic_error("Cannot increment an end iterator.");
      }
      value += range.step;          // Increment the value by the range step
      return *this;
    }
    // Postfix increment operator
    Numeric_Iterator operator++(int)
    {
      // When the value is one step more than the last, it's an end iterator
      if (value == static_cast<T>(range.start + range.count*range.step))
      {
        throw std::logic_error("Cannot increment an end iterator.");
      }
      auto temp = *this;
      value += range.step; // Increment the value by the range step
      return temp; // The iterator before it's incremented
    }

    // Comparisons
    bool operator<(const Numeric_Iterator& iter) const { return value < iter.value; }
    bool operator==(const Numeric_Iterator& iter) const { return value == iter.value; }
    bool operator!=(const Numeric_Iterator& iter) const { return value != iter.value; }
    bool operator>(const Numeric_Iterator& iter) const { return value > iter.value; }
    bool operator<=(const Numeric_Iterator& iter) const { *this < iter || *this ==
      iter; }
      bool operator>=(const Numeric_Iterator& iter) const { *this > iter || *this ==
        iter; }
    };
```

代码看起来虽多，却很简单直白。这个迭代器有一个成员变量，它保存了一个和它相关联的 Numeric_Range 对象的引用，另外还保存了它所指向元素的值。迭代器的构造函数的参数是一个 Numeric_Range 对象的引用。构造函数用参数初始化成员变量 range，并将成员变量 value 的值设为 Numeric_Range 的 start。还定义了一些解引用运算符、前缀或后缀自增运算符以及一套比较运算符。对元素段的结束迭代器的解引用或自增都是非法的，因此如果操作数是结束迭代器，那么自增运算符函数和解引用运算符函数都会抛出异常；这表明成员变量 value 超出了元素段中的最后一个值。为了简单，选择抛出一个标准异常。

头文件 Numeric_Range.h 的完整内容如下：

```
// Numeric_Range.h for Ex2_07
// Defines class templates for a range and iterators for the range
#ifndef NUMERIC_RANGE_H
#define NUMERIC_RANGE_H
#include <exception>                    // For standard exception types
#include <iterator>                     // For iterator type
#include <type_traits>                  // For compile-time type checking
template <typename T> class Numeric_Iterator; // Template type declaration
// Template to define a numeric range, as above...
```

```
// Template to define a numeric range iterator, as above...
#endif
// Ex2_07.cpp
// Exercising the Numeric_Range template
#include <algorithm>                  // For copy()
#include <numeric>                    // For accumulate()
#include <iostream>                   // For standard streams
#include <vector>                     // For vector container
#include "Numeric_Range.h"            // For Numeric_Range<T> & Numeric_Iterator<T>

int main()
{
  Numeric_Range<double> range {1.5, 0.5, 5};
  auto first = range.begin();
  auto last = range.end();
  std::copy(first, last, std::ostream_iterator<double>(std::cout, " "));
  std::cout << "\nSum = " << std::accumulate(std::begin(range), std::end(range),
    0.0) << std::endl;
  // Initializing a container from a Numeric_Range
  Numeric_Range<long> numbers {15L, 4L, 10};
  std::vector<long> data {std::begin(numbers), std::end(numbers)};
  std::cout << "\nValues in vector are:\n";
  std::copy(std::begin(data), std::end(data), std::ostream_iterator<long>
    (std::cout, " "));
  std::cout << std::endl;
  // List the values in a range
  std::cout << "\nThe values in the numbers range are:\n";
  for (auto n : numbers)
  std::cout << n << " ";
  std::cout << std::endl;
}
```

示例输出如下：

```
1.5 2 2.5 3 3.5
Sum = 12.5

Values in vector are:
15 19 23 27 31 35 39 43 47 51

The values in the numbers range are:
15 19 23 27 31 35 39 43 47 51
```

生成的第一个 Numeric_Range 实例有 5 个 double 型元素，它们从 1.5 开始，每次增加 0.5。Numeric_Range 的迭代器用来在 copy()算法中将值复制到 ostream_iterator。这表明算法可以接受这个迭代器。第二个 Numeric_Range 实例有 10 个 long 型元素。在 vector 容器的初始化列表中，使用开始和结束迭代器，然后用 copy()算法输出 vector 中的元素。最后，为了演示它的工作原理，以 for 循环的方式输出它的值。输出表明 Numeric_Range 模板成功创建了整型和浮点型的元素段，我们成功定义了一个可以使用 STL 的迭代器类型。

2.8 本章小结

本章是关于序列容器的，它们非常灵活，我们可能会经常用到它们。它们虽然没有对它们所包含的元素进行基本的排序，但是允许我们以想用的任何方式排序。本章需要了解的重要知识点如下：

- array<T,N>容器可以存放 N 个类型为 T 的元素。可以像常规数组那样使用它，但它提供了比常规数组更多的优点，array 容器知道自己的大小，因此它可以作为参数传给函数，而不需要传入 array 元素的个数。它也能够使用成员函数 at()来检查访问元素的索引。相对于常规数组，使用数组容器只是增加了一点开销。
- vector<T>容器可以存储任意个数的 T 类型元素。vector 容器会自动增长来容纳元素。
- 可以在 vector 的末尾高效地添加或删除元素；但在序列内部添加或删除元素会变慢，因为需要移动元素。
- 像数组那样，也可以使用索引来访问vector中的元素，或者调用会检查索引的成员函数 at()。尽管和常规数组相比，vector 会带来一些小的开销，但是在大多数情况下，我们都不需要注意这一点。
- deque<T>是一个双端队列，可以存储任意个数的 T 类型元素。可以用和 vector 同样的方式访问 deque 容器中的元素。
- 可以在 deque 容器的头部和尾部高效地添加或删除元素；但是添加或删除序列内部的元素会变慢。
- Array、vector、deque 容器提供了 const 和 non-const 随机访问迭代器和反向迭代器。
- list<T>是一个存储 T 类型元素的双向链表。可以在容器的任何位置高效地添加或删除元素。
- 只能以从序列头部或尾部遍历元素的方式访问 list 容器中的元素。
- list 容器提供双向迭代器。
- forward_list<T>容器以单链表的形式存储 T 类型的元素。它只能从第一个元素开始遍历。forward_list 容器比 list 容器更快、更简单。
- forward_list 容器提供正向迭代器。
- 头文件 algorithm 中定义的 copy()算法可以将一段元素复制到另一个迭代器指定的位置。
- 可以将 copy()算法和流迭代器一起使用，用来从输入流读取数据，然后把它们复制到容器中，或者在从容器读取数据后将它们输出到流中。
- 函数模板 sort()定义在头文件 algorithm 中，可以对随机访问迭代器指定的元素进行排序。元素默认会被排为升序，也可以用自定义的二元断言作为 sort()参数去决定元素的排列顺序。
- list 和 forward_list 容器都有成员函数 sort()，可以对元素进行排序。

练习

这里的练习用来检验你是否记住了本章的主题。如果遇到困难，可以回顾前面的章节，寻找帮助。在这之后，如果还是无法解决的话，可以从 Apress 出版社的网站下载解答(http://www.apress.com/9781484200056)，但这应该是最后的选择。

1. Fibonacci 数列是一个整数序列 0、1、1、2、3、5、8、13、21……，头两个数后面的每个整数都是它前面两个整数的和。写一个程序，用 lambda 表达式生成 50 个 Fibonacci 数来初始化 array<T,N>容器。在程序中用全局函数将容器元素每 8 个一行地输出。

2. 写一个程序，可以从键盘读取任意个数的城市名，然后以 std::string 对象的形式把它们存放到 vector<T>容器中。以升序的形式对城市名排序，并且每行几个地列出它们，每一个字段的长度固定，可以适应最长的城市名。按它们的开头字母分组输出，每组之间用一个空行隔开。

3. 用 list<T>容器重复前一个练习。想出一个通过输入流迭代器读入城市名的方法，甚至在它们由两个以上的名称构成时(例如"New York")，也能作为一个名称来保存。

4. 扩展前一个示例，使用前向插入迭代器将 list 容器的内容转移到 deque<T>容器中。然后对 deque 容器中的内容进行排序，并使用输出流迭代器输出城市名。

第 3 章

容器适配器

本章会说明关于前一章所学容器的一些变化。它们定义了一些在特定环境下使用的序列容器接口。我们也会了解如何在容器中保存指针。本章将介绍以下内容：

- 什么是容器适配器(Container Adapter)
- 如何定义堆栈(Stack)，何时使用以及如何使用
- 如何定义和使用队列(queue)
- 如何生成和使用优先级队列(priority queue)，及其与一般队列的区别
- 什么是堆(heap)，如何生成和使用堆，如何将堆和优先级队列关联起来
- 在容器中存放指针(尤其是智能指针)的好处

3.1 什么是容器适配器

容器适配器是一个封装了序列容器的类模板，它在一般序列容器的基础上提供了一些不同的功能。之所以称作适配器类，是因为它可以通过适配容器现有的接口来提供不同的功能。

这里有 3 种容器适配器：

- stack<T>是一个封装了 deque<T>容器的适配器类模板，默认实现的是一个后入先出(Last-In-First-Out，LIFO)的压入栈。stack<T>模板定义在头文件 stack 中。
- queue<T>是一个封装了 deque<T>容器的适配器类模板，默认实现的是一个先入先出(First-In-First-Out，LIFO)的队列。可以为它指定一个符合确定条件的基础容器。queue<T>模板定义在头文件 queue 中。
- priority_queue<T>是一个封装了 vector<T>容器的适配器类模板，默认实现的是一个会对元素排序，从而保证最大元素总在队列最前面的队列。priority_queue<T>模板定义在头文件 queue 中。

适配器类在基础序列容器的基础上实现了一些自己的操作，显然也可以添加一些自己的操作。它们提供的优势是简化了公共接口，而且提高了代码的可读性。后面我们会详细地探讨这些适配器的应用。

3.2 创建和使用 stack<T>容器适配器

stack<T>容器适配器中的数据是以 LIFO 的方式组织的，这和自助餐馆中堆叠的盘子、箱子中的一堆书类似。图 3-1 展示了一个理论上的 stack 容器及其一些基本操作。只能访问 stack 顶部的元素；只有在移除 stack 顶部的元素后，才能访问下方的元素。

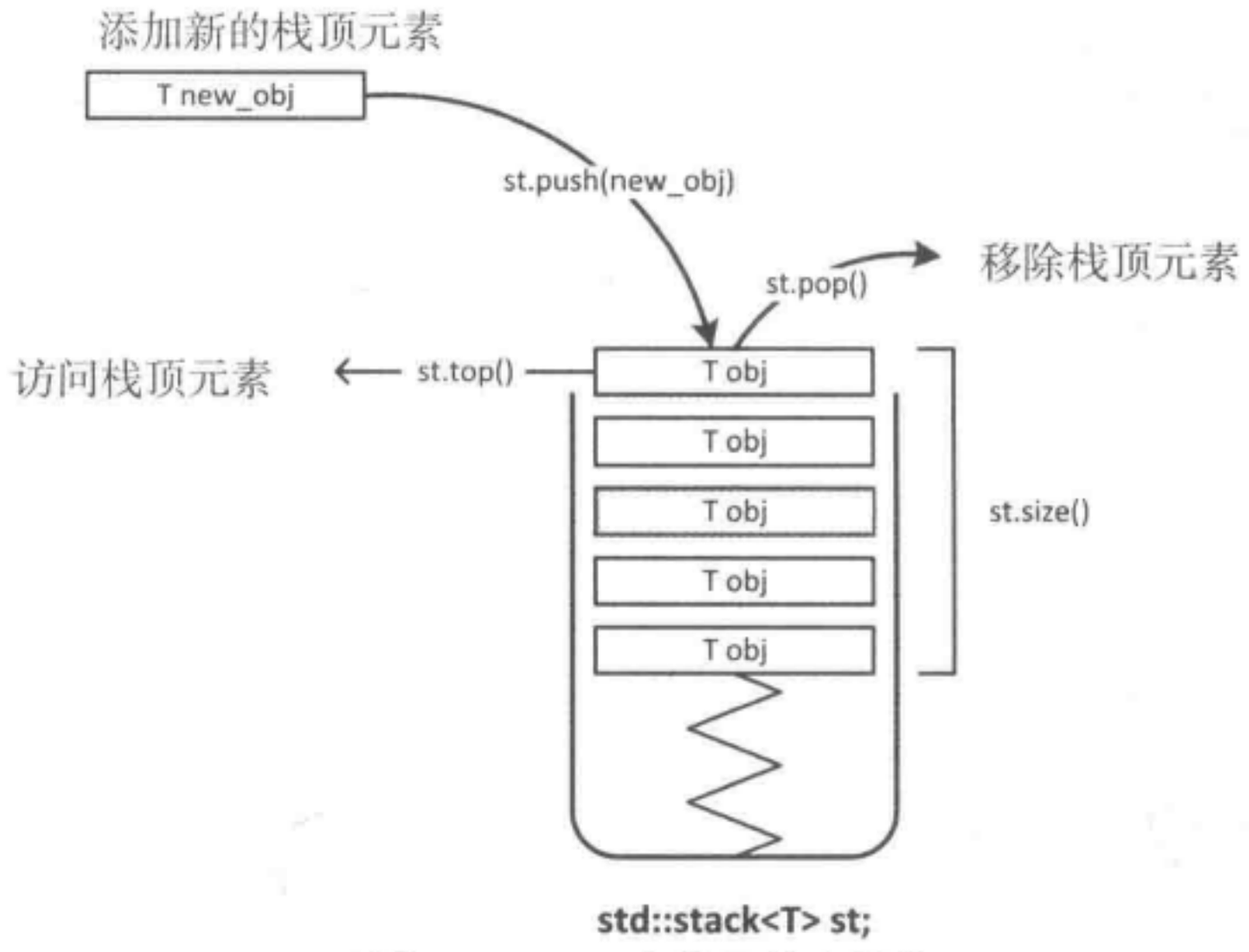


图 3-1 stack 容器的基本操作

stack 容器有广泛的应用。例如，编辑器中的 undo(撤销)机制就是用堆栈来记录连续的变化。撤销操作可以取消最后一个操作，这也是发生在堆栈顶部的操作。编译器使用堆栈来解析算术表达式，当然也可以用堆栈来记录 C++代码的函数调用。下面展示了如何定义一个用来存放字符串对象的 stack 容器：

```
std::stack<std::string> words;
```

stack 容器适配器的模板有两个参数。第一个参数是存储对象的类型，第二个参数是底层容器的类型。stack<T>的底层容器默认是 deque<T>容器，因此模板类型其实是 stack<typename T, typename Container=deque<T>>。通过指定第二个模板类型参数，可以使用任意类型的底层容器，只要它们支持 back()、 push_back()、pop_back()、empty()、size()这些操作。下面展示了如何定义一个使用 list<T>的堆栈：

```
std::stack<std::string, std::list<std::string>> fruit;
```

创建堆栈时，不能在初始化列表中用对象来初始化，但是可以用另一个容器来初始化，只要堆栈的底层容器类型和这个容器的类型相同。例如：

```
std::list<double> values {1. 414, 3. 14159265, 2. 71828};
std::stack<double, std::list<double>> my_stack (values);
```

第二条语句生成了一个包含 value 元素副本的 my_stack。这里不能在 stack 构造函数中

使用初始化列表；必须使用圆括号。如果没有在第二个 stack 模板类型参数中将底层容器指定为 list，那么底层容器可能是 deque，这样就不能用 list 的内容来初始化 stack；只能接受 deque。

stack<T>模板定义了拷贝构造函数，因而可以复制现有的 stack 容器：

```
std::stack<double, std::list<double>> copy_stack {my_stack};
```

copy_stack 是 my_stack 的副本。如你所见，在使用拷贝构造函数时，既可以用初始化列表，也可以用圆括号。

堆栈操作

和其他序列容器相比，stack 是一类存储机制简单、所提供操作较少的容器。下面是 stack 容器可以提供的一套完整操作：

- top()返回一个栈顶元素的引用，类型为 T&。如果栈为空，返回值未定义。
- push(const T& obj)可以将对象副本压入栈顶。这是通过调用底层容器的 push_back()函数完成的。
- push(T&& obj)以移动对象的方式将对象压入栈顶。这是通过调用底层容器的有右值引用参数的 push_back()函数完成的。
- pop()弹出栈顶元素。
- size()返回栈中元素的个数。
- empty()在栈中没有元素的情况下返回 true。
- emplace()用传入的参数调用构造函数，在栈顶生成对象。
- swap(stack<T> & other_stack)将当前栈中的元素和参数中的元素交换。参数所包含元素的类型必须和当前栈的相同。对于 stack 对象有一个特例化的全局函数 swap()可以使用。

stack<T>模板也定义了复制和移动版的 operator=()函数，因此可以将一个 stack 对象赋值给另一个 stack 对象。stack 对象有一整套比较运算符。比较运算通过字典的方式来比较底层容器中相应的元素。字典比较是一种用来对字典中的单词进行排序的方式。依次比较对应元素的值，直到遇到两个不相等的元素。第一个不匹配的元素会作为字典比较的结果。如果一个 stack 的元素比另一个 stack 的多，但是所匹配的元素都相等，那么元素多的那个 stack 容器大于元素少的 stack 容器。

我们可以用 stack 容器来实现一个简单的计算器程序。这个程序支持一些基本的加、减、乘、除、幂操作。它们分别对应运算符+、-、*、/、^。幂操作由定义在头文件 cmath 中的 pow()函数提供。表达式以单行字符串的形式读入，可以包含空格。在解析字符串之前可以使用 remove()算法来移除输入表达式中的空格，然后再执行这个表达式所包含的运算。下面我们会定义一个函数来获取运算符的优先级：

```
inline size_t precedence(const char op)
{
  if (op == '+' || op == '-')
```

```
    return 1;
  if (op == '*' || op == '/')
    return 2;
  if (op == '^')
    return 3;
  throw std::runtime_error {string{"invalid operator: "} + op};
}
```

+和-的优先级最低，其次是*和/，最后是^。运算符的优先级决定了它们在表达式中的执行顺序。如果参数是一个不支持的运算符，那么会抛出一个 runtime_error 异常对象。异常对象构造函数中的字符串参数，可以在 catch 代码块中通过调用对象的 what()函数获得。

这个程序通过从左到右扫描的方式来分析输入表达式，并且会将运算符保存到 stack 容器 operators 中。操作数存放在 stack 容器 operands 中。所有的运算符都需要两个操作数，所以每执行一次运算，都需要获取一个 operators 栈顶的运算符，以及两个 operands 栈顶的操作数。运算由下面的函数执行：

```
double execute(std::stack<char>& ops, std::stack<double>& operands)
{
  double result {};
  double rhs {operands. top()};          // Get rhs. . .
  operands. pop();                       // . . . and delete from stack
  double lhs {operands. top()};          // Get lhs. . .
  operands. pop();                       // . . . and delete from stack
  switch (ops. top())                    // Execute current op
  {
  case '+':
    result = lhs + rhs;
    break;
  case '-':
    result = lhs - rhs;
    break;
  case '*':
    result = lhs * rhs;
    break;
  case '/':
    result = lhs / rhs;
    break;
  case '^':
    result = std::pow(lhs, rhs);
    break;
  default:
    throw std::runtime_error {string{"invalid operator: "} + ops. top()};
  }
  ops. pop(); // Delete op just executed
  operands. push(result);
  return result;
}
```

函数的参数是两个 stack 容器的引用。可以用 operands 容器的 top()函数获取操作数。top()函数只能得到栈顶元素；为了访问下一个元素，必须通过调用 pop()来移除当前栈顶元素。注意栈中操作数的顺序是相反的，因此得到的第一个操作数是运算的右操作数。operators 容器顶部的元素用来在 switch 中选择运算。在它不匹配任何一个 case 分支时，会抛出一个异常来表明这个运算符是无效的。

下面是这个程序的完整代码：

```
// Ex3_01. cpp
// A simple calculator using stack containers

#include <cmath>                        // For pow() function
#include <iostream>                     // For standard streams
#include <stack>                        // For stack<T> container
#include <algorithm>                    // For remove()
#include <stdexcept>                    // For runtime_error exception
#include <string>                       // For string class
using std::string;

// Code for the precedence() function goes here. . .

// Code for the execute() function goes here. . .

int main()
{
  std::stack<double> operands;    // Push-down stack of operands
  std::stack<char> operators;     // Push-down stack of operators
  string exp;                     // Expression to be evaluated
  std::cout << " An arithmetic expression can include the operators +, -,
               *, /,"
            << " and ^ for exponentiation. "
            << std::endl;
  try
  {
    while (true)
    {
      std::cout << "Enter an arithmetic expression and press Enter"
                << " - enter an empty line to end:"
                << std::endl;
      std::getline(std::cin, exp, '\n');
      if (exp. empty()) break;

      // Remove spaces
      exp. erase(std::remove(std::begin(exp), std::end(exp), ' '),
        std::end(exp));
      size_t index {};                // Index to expression string

      // Every expression must start with a numerical operand
      operands. push(std::stod(exp, &index)); // Push the first (lhs) operand
                                              // on the stack
      while (true)
```

```
        {
          operators. push(exp[index++]); // Push the operator on to the stack

          // Get rhs operand
            size_t i {}; // Index to substring
          operands. push(std::stod(exp. substr(index), &i)); // Push rhs operand
          index += i; // Increment expression index

          if (index == exp. length()) // If we are at end of exp. . .
          {
            while (!operators. empty())  // . . . execute outstanding ops
              execute(operators, operands);
            break;
          }
          // If we reach here, there's another op. . .
          // If there's a previous op of equal or higher precedence execute it
          while (!operators. empty() && precedence(exp[index]) <= precedence
(operators. top()))
            execute(operators, operands);  // Execute previous op.
        }
        std::cout << "result = " << operands. top() << std::endl;
      }
    }
    catch (const std::exception& e)
    {
      std::cerr << e. what() << std::endl;
    }
    std::cout << "Calculator ending. . . " << std::endl;
  }
```

while 循环包含在一个 try 代码块中，这样就可以捕获抛出的任何异常。在 catch 代码块中，调用异常对象的成员函数 what()会将错误信息输出到标准错误流中。在一个死循环中执行输入操作，当输入一个空字符串时，循环结束。可以使用 remove()算法消除非空字符串中的空格。remove()不能移除元素，而只能通过移动元素的方式来覆盖要移除的元素。为了移除 exp 字符串中剩下的多余元素，可以用两个迭代器作为参数调用 erase()。其中第一个迭代器由 remove()返回，指向字符串的最后一个有效元素的后面位置。第二个迭代器是字符串原始状态的结束迭代器。这两个迭代器指定范围的元素会被删除。

每个浮点操作数的值都是用定义在头文件 string 中的 stod()函数获取的。这会将第一个字符串参数中的字符序列转换为 double 值。函数会从第一个表示有效浮点数的字符串的第一个字符开始，获取最长字符序列。第二个参数是一个整型指针，保存的是字符串中非数字部分第一个字符的索引。string 头文件中定义了 stod()函数，它可以返回一个 float 值。Stod()会返回一个 long double 值。

因为所有的运算符都需要两个操作数，所以有效的输入字符串格式总是为 *operand op operand op operand*，等等。序列的第一个和最后一个元素都是操作数，每对操作数之间有一个运算符。因为有效表达式总是以操作数开头，所以第一个操作数在分析表达式的嵌套

循环之前被提取出来。在循环中，输入字符串的运算符会被压入 operators 栈。在确认没有到达字符串末尾后，再从 exp 提取第二个操作数。这时，stod()的第一个参数是从 index 开始的 exp 字符串，它是被压入 operators 栈的运算符后的字符。非数字字符串的第一个索引保存在 i 中。因为 i 是相对于 index 的，所以我们会将 index 加上 i 的值，使它指向操作数后的一个运算符(如果是 exp 中的最后一个操作数，它会指向字符串末尾的下一个位置)。

当 index 的值超过 exp 的最后一个字符时，会执行 operators 容器中剩下的运算符。如果没有到达字符串末尾，operators 容器也不为空，我们会比较 operators 栈顶运算符和 exp 中下一个运算符的优先级。如果栈顶运算符的优先级高于下一个运算符，就先执行栈顶的运算符。否则，就不执行栈顶运算符，在下一次循环开始时，将下一个运算符压入 operators 栈。通过这种方式，就可以正确计算出带优先级的表达式的值。

下面是一些示例输出：

```
An arithmetic expression can include the operators +, -, *, /, and ^ for
    exponentiation.
Enter an arithmetic expression and press Enter - enter an empty line to end:
    2^0. 5
    result = 1. 41421
Enter an arithmetic expression and press Enter - enter an empty line to end:
    2. 5e2 + 1. 5e1*4 - 1000
    result = -690
Enter an arithmetic expression and press Enter - enter an empty line to end:
    3*4*5 + 4*5*6 + 5*6*7
    result = 390
Enter an arithmetic expression and press Enter - enter an empty line to end:
    1/2 + 1/3 +1/4
    result = 1. 08333
Enter an arithmetic expression and press Enter - enter an empty line to end:

Calculator ending. . .
```

程序输出表明计算器工作正常，也说明 stod()函数可以将不同符号的字符串转换为数字。当然，如果表达式能够支持括号，是不是会更好？本书准备把这个作为练习留给你们完成。

3.3　创建和使用 queue<T>容器适配器

只能访问 queue<T>容器适配器的第一个和最后一个元素。只能在容器的末尾添加新元素，只能从头部移除元素。许多程序都使用了 queue 容器。queue 容器可以用来表示超市的结账队列或服务器上等待执行的数据库事务队列。对于任何需要用 FIFO 准则处理的序列来说，使用 queue 容器适配器都是好的选择。

图 3-2 展示了一个 queue 容器及其一些基本操作：

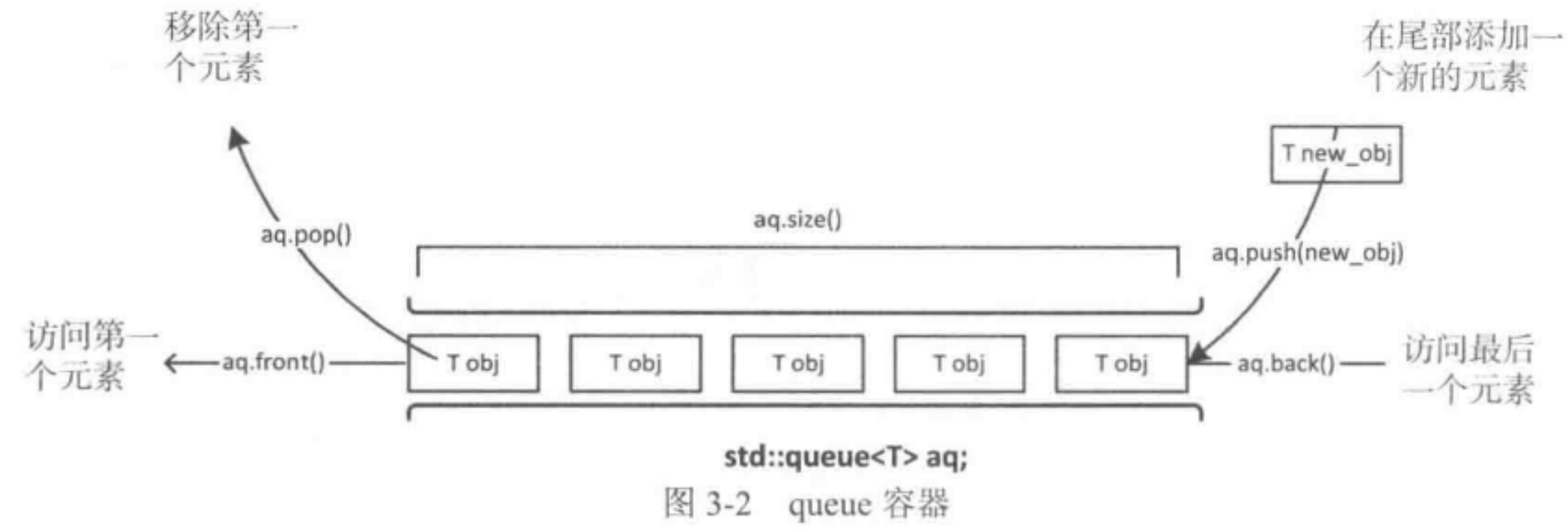


图 3-2　queue 容器

queue 的生成方式和 stack 相同，下面展示如何创建一个保存字符串对象的 queue：

```
std::queue<std::string> words;
```

也可以使用拷贝构造函数:

```
std::queue<std::string> copy_words {words}; // A duplicate of words
```

stack<T>、queue<T>这类适配器类都默认封装了一个 deque<T>容器，也可以通过指定第二个模板类型参数来使用其他类型的容器：

```
std::queue<std::string, std::list<std::string>> words;
```

底层容器必须提供这些操作：front()、back()、push_back()、pop_front()、empty()和 size()。

3.3.1　queue 操作

queue 和 stack 有一些成员函数相似，但在一些情况下，工作方式有些不同：

- front()返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- back()返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- push(const T& obj)在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push_back()来完成的。
- push(T&& obj)以移动的方式在 queue 的尾部添加元素。这是通过调用底层容器的具有右值引用参数的成员函数 push_back()来完成的。
- pop()删除 queue 中的第一个元素。
- size()返回 queue 中元素的个数。
- empty()返回 true，如果 queue 中没有元素的话。
- emplace()用传给 emplace()的参数调用 T 的构造函数，在 queue 的尾部生成对象。
- swap(queue<T> &other_q)将当前 queue 中的元素和参数 queue 中的元素交换。它们需要包含相同类型的元素。也可以调用全局函数模板 swap()来完成同样的操作。

queue<T>模板定义了拷贝和移动版的 operator=()，对于所保存元素类型相同的 queue

对象，它们有一整套的比较运算符，这些运算符的工作方式和 stack 容器相同。

和 stack 一样，queue 也没有迭代器。访问元素的唯一方式是遍历容器内容，并移除访问过的每一个元素。例如：

```
std::deque<double> values {1. 5, 2. 5, 3. 5, 4. 5};
std::queue<double> numbers(values);
while (!numbers. empty())                    // List the queue contents. . .
{
  std ::cout << numbers. front() << " "; // Output the 1st element
  numbers. pop();                          // Delete the 1st element
}
std::cout << std::endl;
```

用循环列出 numbers 的内容，循环由 empty()返回的值控制。调用 empty()可以保证我们能够调用一个空队列的 front()函数。如代码所示，为了访问 queue 中的全部元素，必须删除它们。如果不想删除容器中的元素，必须将它们复制到另一个容器中。如果一定要这么操作，我们可能需要换一个容器。

3.3.2　queue 容器的实际使用

这里汇集了一些使用 queue 容器的示例。这是一个用 queue 模拟超市运转的程序。结账队列的长度是超市运转的关键因素。它会影响超市可容纳的顾客数——因为太长的队伍会使顾客感到气馁，从而放弃排队。在很多情形中——医院可用病床数会严重影响应急处理设施的运转，也会产生同样的队列问题。我们的超市模拟是一个简单模型，灵活性有限。

可以在头文件 Customer. h 中定义一个类来模拟顾客：

```
// Defines a customer by their time to checkout
#ifndef CUSTOMER_H
#define CUSTOMER_H
class Customer
{
private:
  size_t service_t {};                  // Time to checkout
public:
  explicit Customer(size_t st = 10) :service_t {st}{}

  // Decrement time remaining to checkout
  Customer& time_decrement()
  {
    if(service_t > 0)
      --service_t;
    return *this;
  }
  bool done() const { return service_t == 0; }
};
#endif
```

这里只有一个成员变量 service_t，用来记录顾客结账需要的时间。每个顾客的结账时间都不同。每过一分钟，会调用一次 time_decrement()函数，这个函数会减少 service_t 的值，它可以反映顾客结账所花费的时间。当 service_t 的值为 0 时，成员函数 done()返回 true。

超市的每个结账柜台都有一队排队等待的顾客。Checkout.h 中定义的 Checkout 类如下：

```
// Supermarket checkout - maintains and processes customers in a queue
#ifndef CHECKOUT_H
#define CHECKOUT_H
#include <queue>                              // For queue container
#include "Customer. h"

class Checkout
{
private:
  std::queue<Customer> customers;             // The queue waiting to checkout
public:
  void add(const Customer& customer) { customers. push(customer); }
  size_t qlength() const { return customers. size(); }

  // Increment the time by one minute
  void time_increment()
  { // There are customers waiting. . .
    if (!customers. empty())
    { // There are customers waiting. . .
      if (customers. front(). time_decrement(). done())  // If the customer is
                                                         // done. . .
        customers. pop(); // . . . remove from the queue
    }
  };
  bool operator<(const Checkout& other) const { return qlength() < other.
qlength(); }
  bool operator>(const Checkout& other) const { return qlength() > other.
qlength(); }
};
#endif
```

这相当于自我解释。queue 容器是 Checkout 唯一的成员变量，用来保存等待结账的 Customer 对象。成员函数 add()可以向队列中添加新顾客。只能处理队列中的第一个元素。每过一分钟，调用一次 Checkout 对象的成员函数 time_increment()，它会调用第一个 Customer 对象的成员函数 time_decrement()来减少剩余的服务时间，然后再调用成员函数 done()。如果 done()返回 true，表明顾客结账完成，因此把他从队列中移除。Checkout 对象的比较运算符可以比较队列的长度。

为了模拟超市结账，我们需要有随机数生成的功能。因此打算使用 random 头文件中的一个非常简单的工具，但不打算深入解释它。我们会在本书后面的章节深入探讨 random 头文件中的内容。程序使用了一个 uniform_int_distribution<>类型的实例。顾名思义，它定

义的整数值在最大值和最小值之间均匀分布。在均匀分布中，所有这个范围内的值都可能相等。可以在 10 和 100 之间定义如下分布：

```
std::uniform_int_distribution<> d {10, 100};
```

这里只定义了分布对象 d，它指定了整数值分布的范围。为了获取这个范围内的随机数，我们需要使用一个随机数生成器，然后把它作为参数传给 d 的调用运算符，从而返回一个随机整数。random 头文件中定义了几种随机数生成器。这里我们使用最简单的一个，可以按如下方式定义：

```
std::random_device random_number_engine;
```

为了在 d 分布范围内生成随机数，我们可以这样写：

```
auto value = d(random_number_engine); // Calls operator()() for d
```

value 的值在 d 分布范围内。
完整模拟器的源文件如下：

```
// Ex3_02. cpp
// Simulating a supermarket with multiple checkouts
#include <iostream>                 // For standard streams
#include <iomanip>                  // For stream manipulators
#include <vector>                   // For vector container
#include <string>                   // For string class
#include <numeric>                  // For accumulate()
#include <algorithm>                // For min_element & max_element
#include <random>                   // For random number generation

#include "Customer. h"
#include "Checkout. h"

using std::string;
using distribution = std::uniform_int_distribution<>;

// Output histogram of service times
void histogram(const std::vector<int>& v, int min)
{
  string bar (60, '*');             // Row of asterisks for bar
  for (size_t i {}; i < v. size(); ++i)
  {
    std::cout << std::setw(3) << i+min << " " // Service time is index + min
      << std::setw(4) << v[i] << " " // Output no.  of occurrences
      << bar. substr(0, v[i])       // . . . and that no.  of asterisks
      << (v[i] > static_cast<int>(bar. size()) ? ". . . " : "")
      << std::endl;
  }
}
int main()
{
```

```
  std::random_device random_n;

// Setup minimum & maximum checkout periods - times in minutes
int service_t_min {2}, service_t_max {15};
distribution service_t_d {service_t_min, service_t_max};

// Setup minimum & maximum number of customers at store opening
int min_customers {15}, max_customers {20};
distribution n_1st_customers_d {min_customers, max_customers};

// Setup minimum & maximum intervals between customer arrivals
int min_arr_interval {1}, max_arr_interval {5};
distribution arrival_interval_d {min_arr_interval, max_arr_interval};

size_t n_checkouts {};
std::cout << "Enter the number of checkouts in the supermarket: ";
std::cin >> n_checkouts;
if(!n_checkouts)
{
  std::cout << "Number of checkouts must be greater than 0.  Setting to 1. "
            << std::endl;
  n_checkouts = 1;
}

std::vector<Checkout> checkouts {n_checkouts};
std::vector<int> service_times(service_t_max-service_t_min+1);

// Add customers waiting when store opens
int count {n_1st_customers_d(random_n)};
std::cout << "Customers waiting at store opening: " << count << std::endl;
  int added {};
  int service_t {};
  while (added++ < count)
  {
    service_t = service_t_d(random_n);
    std::min_element(std::begin(checkouts),std::end(checkouts))->
      add(Customer(service_t));
    ++service_times[service_t - service_t_min];
  }

  size_t time {}; // Stores time elapsed
  const size_t total_time {600}; // Duration of simulation - minutes
  size_t longest_q {}; // Stores longest checkout queue length

// Period until next customer arrives
int new_cust_interval {arrival_interval_d(random_n)};

// Run store simulation for period of total_time minutes
while (time < total_time) // Simulation loops over time
{
  ++time; // Increment by 1 minute

  // New customer arrives when arrival interval is zero
```

```
if (--new_cust_interval == 0)
{
  service_t = service_t_d(random_n); // Random customer service time
     std::min_element(std::begin(checkouts),
     std::end(checkouts))->add (Customer(service_t));
  ++service_times[service_t - service_t_min]; // Record service time

  // Update record of the longest queue occurring
  for (auto & checkout : checkouts)
    longest_q = std::max(longest_q, checkout. qlength());
     new_cust_interval = arrival_interval_d(random_n);
  }

  // Update the time in the checkouts - serving the 1st customer in each
     queue
  for (auto & checkout : checkouts)
    checkout. time_increment();
}

std::cout << "Maximum queue length = " << longest_q << std::endl;
std::cout << "\nHistogram of service times:\n";
histogram(service_times, service_t_min);

std::cout << "\nTotal number of customers today: "
          << std::accumulate(std::begin(service_times), std::end
             (service times), 0)
          << std::endl;
}
```

直接使用 using 指令可以减少代码输入，简化代码。顾客服务次数记录在 vector 容器中。服务时间减去 service_times 的最小值可以用来索引需要自增的 vector 元素，这导致 vector 的第一个元素会记录下最少服务时间的发生次数。histogram()函数会以水平条形图的形式生成每个服务时间出现次数的柱状图。

输入的唯一数字是 checkouts。此处选择将模拟持续时间设置为 600 分钟，也可以用参数输入这个时间。main()函数生成了顾客服务时间，超市开门时等在门外的顾客数，以及顾客到达时间间隔的分布对象。它表明顾客在同一时间到达。我们可以轻松地将这个程序扩展为每次到达的顾客数是一个处于一定范围内的随机数。

顾客总是可以被分配到最短的结账队列。通过调用 min_element()算法可以找到最短的 Checkout 对象队列。这会使用<运算符比较元素，但是这个算法的另一个版本有第三个参数可以指定比较函数。在这次模拟开始前，当超市开门营业时，在门外等待的顾客的初始序列被添加到 Checkout 对象中，然后服务时间记录被更新。

模拟在 while 循环中进行。在每次循环中，time 都会增加 1 分钟。在下一个顾客到达期间，new_cust_interval 会在每次循环中减小，直到等于 0。用新的随机服务时间生成新的顾客，然后把它加到最短的 Checkout 对象队列中。这个时候也会更新变量 longest_q，因为在添加新顾客后，可能出现新的最长队列。

然后调用每个 Checkout 对象的 time_increment()函数来处理队列中的第一个顾客。下面是一些示例输出：

```
Enter the number of checkouts in the supermarket: 3
Customers waiting at store opening: 18
Maximum queue length = 7

Histogram of service times:
  2 16 ****************
  3 20 ********************
  4 13 *************
  5 16 ****************
  6 16 ****************
  7 12 ************
  8 11 ***********
  9 14 **************
 10 10 **********
 11 20 ********************
 12 15 ***************
 13 15 ***************
 14 14 **************
 15 14 **************

Total number of customers today: 206
```

这里有 3 个结账柜台。将它们设为 2 个时，最长队列的长度达到 42——已经长到会让顾客放弃付款。还可以做更多改进，让模拟更加真实。均匀分配并不符合实际，例如，顾客通常成群结队到来。可以增加一些其他的因素，比如收银员休息时间、某个收银员生病工作状态不佳，这些都会导致顾客不选择这个柜台结账。

3.4 使用 priority_queue<T>容器适配器

不出所料，priority_queue 容器适配器定义了一个元素有序排列的队列。默认队列头部的元素优先级最高。因为它是一个队列，所以只能访问第一个元素，这也意味着优先级最高的元素总是第一个被处理。但是如何定义“优先级”完全取决于我们自己。如果一个优先级队列记录的是医院里等待接受急救的病人，那么病人病情的严重性就是优先级。如果队列元素是银行的借贷业务，那么借记可能会优先于信贷。

priority_queue 模板有 3 个参数，其中两个有默认的参数；第一个参数是存储对象的类型，第二个参数是存储元素的底层容器，第 3 个参数是函数对象，它定义了一个用来决定元素顺序的断言。因此模板类型是：

```
template
  <typename T, typename Container=std::vector<T>, typename Compare=std::
   less<T>>
  class priority_queue
```

如你所见，priority_queue 实例默认有一个 vector 容器。函数对象类型 less<T>是一个默认的排序断言，定义在头文件 function 中，决定了容器中最大的元素会排在队列前面。function 中也定义了 greater<T>，用来作为模板的最后一个参数对元素排序，最小元素会排在队列前面。当然，如果指定模板的最后一个参数，就必须提供另外的两个模板类型参数。

图 3-3 中显示元素的方式反映了它们被检索的顺序。在 vector 中它们也可以不像这样排序。在讨论堆时，会解释原因。

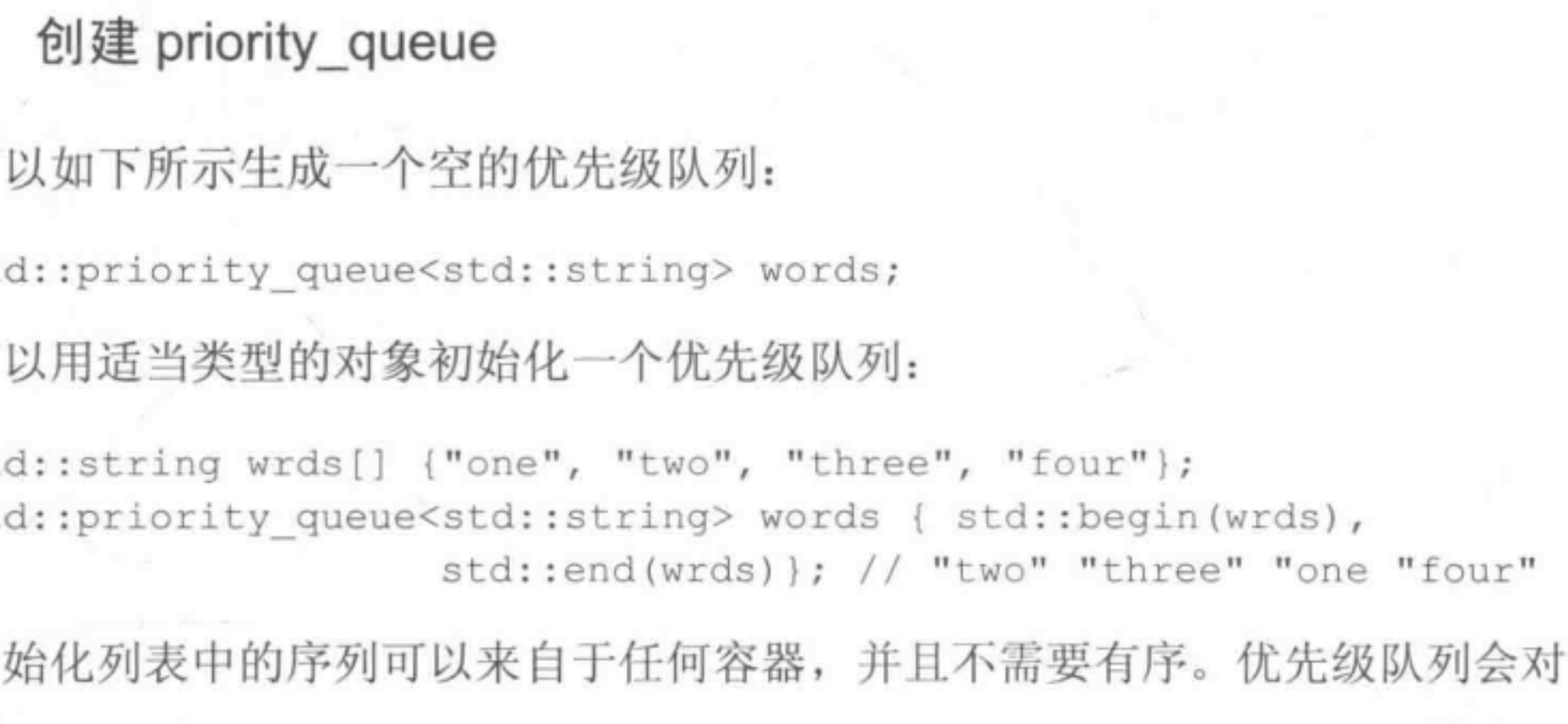


图 3-3　priority_queue 容器

3.4.1　创建 priority_queue

可以如下所示生成一个空的优先级队列：

```
std::priority_queue<std::string> words;
```

可以用适当类型的对象初始化一个优先级队列：

```
std::string wrds[] {"one", "two", "three", "four"};
std::priority_queue<std::string> words { std::begin(wrds),
                    std::end(wrds)}; // "two" "three" "one "four"
```

初始化列表中的序列可以来自于任何容器，并且不需要有序。优先级队列会对它们进行排序。

拷贝构造函数会生成一个和现有对象同类型的 priority_queue 对象，它是现有对象的一个副本。例如：

```
std::priority_queue<std::string> copy_words {words}; // copy of words
```

也有带右值引用参数的拷贝构造函数，它可以移动一个实参对象。

当对容器内容反向排序时，最小的元素会排在队列前面，这时候需要指定 3 个模板类型参数：

```
std::string wrds[] {"one", "two", "three", "four"};
std::priority_queue<std::string, std::vector<std::string>,
                    std::greater<std::string>>
```

```
words1 {std::begin(wrds), std::end(wrds)}; // "four" "one" "three "two"
```

这会通过使用 operator>()函数对字符串对象进行比较，进而生成一个优先级队列，因此这会和它们在队列中的顺序相反。

优先级队列可以使用任何容器来保存元素，只要容器有成员函数 front()、push_back()、pop_back()、size()、empty()。这显然包含了 deque 容器，因此这里也可以用 deque 来代替：

```
std::string wrds[] {"one", "two", "three", "four"};
std::priority_queue<std::string, std::deque<std::string>>
words {std::begin(wrds), std::end(wrds)};
```

这个 words 优先级队列在 deque 容器中保存了一些 wrds 数组中的字符串，这里使用默认的比较断言，因此队列中的元素会和上面 word1 中元素的顺序相同。priority_queue 构造函数会生成一个和第二个类型参数同类型的容器来保存元素，这也是 priority_queue 对象的底层容器。

可以生成 vector 或 deque 容器，然后用它们来初始化 priority_queue。下面展示了如何以 vector 的元素作为初始值来生成 priority_queue 对象：

```
std::vector<int> values{21, 22, 12, 3, 24, 54, 56};
std::priority_queue<int> numbers {std::less<int>(), values};
```

priority_queue 构造函数的第一个参数是一个用来对元素排序的函数对象，第二个参数是一个提供初始元素的容器。在队列中用函数对象对 vector 元素的副本排序。values 中元素的顺序没有变，但是优先级队列中的元素顺序变为：56 54 24 22 21 12 3。优先级队列中用来保存元素的容器是私有的，因此只能通过调用 priority_queue 对象的成员函数来对容器进行操作。构造函数的第一个参数是函数对象类型，它必须和指定的比较模板类型参数相同，函数对象类型默认是 less<T>。如果想使用不同类型的函数，需要指定全部的模板类型参数。例如：

```
std::priority_queue<int, std::vector<int>, std::greater<int>>
                    numbers1 {std::greater<int>(), values};
```

第三个类型参数是一个比较对象类型。如果要指定这个参数，必须指定前两个参数——元素类型和底层容器类型。

3.4.2 priority_queue 操作

对 priority_queue 进行操作有一些限制：

- push(const T& obj)将 obj 的副本放到容器的适当位置，这通常会包含一个排序操作。
- push(T&& obj)将 obj 放到容器的适当位置，这通常会包含一个排序操作。
- emplace(T constructor a rgs…)通过调用传入参数的构造函数，在序列的适当位置构造一个 T 对象。为了维持优先顺序，通常需要一个排序操作。
- top()返回优先级队列中第一个元素的引用。

- pop()移除第一个元素。
- size()返回队列中元素的个数。
- empty()返回 true，如果队列为空的话。
- swap(priority_queue<T>& other)和参数的元素进行交换，所包含对象的类型必须相同。

priority_queue 也实现了赋值运算，可以将右操作数的元素赋给左操作数；同时也定义了拷贝和移动版的赋值运算符。需要注意的是，priority_queue 容器并没有定义比较运算符。因为需要保持元素的顺序，所以添加元素通常会很慢。稍后会在堆(heaps)这一节讨论 priority_queue 的内部操作。

以下展示了如何将键盘输入的数据记录到 priority_queue 中：

```
std::priority_queue<std::string> words;
std::string word;
std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
             line to end:\n";
while (true)
{
  if ((std::cin >> word). eof())
    break;
  words. push(word);
}
```

按下 Ctrl+Z 组合键会在输入流中设置文件结束状态，因此可以用来结束循环输入。istream 对象的成员函数 operator>>()返回一个输入流对象，因此我们可以用 if 条件表达式来调用 eof()以检查 cin 的状态。这里会对输入单词进行排序，所以最大的单词总在 words 队列的前面——自动对输入单词排序。

priority_queue 没有迭代器。如果想要访问全部的元素——例如，列出或复制它们，会将队列清空；priority_queue 和 queue 有相同的限制。如果想在进行这样的操作后，还能保存它的元素，需要先把它复制一份，这里可以使用一个不同类型的容器。下面展示了如何列出优先级队列 words 的内容：

```
std::priority_queue<std::string> words_copy {words}; // A copy for output
while (!words_copy. empty())
{
  std::cout << words_copy. top() << " ";
  words_copy. pop();
}
std::cout << std::endl;
```

这里首先生成了一个 words 的副本，因为输出 words 会移除它的内容。输出 top()返回的元素后，我们需要使用 pop()来使下一个元素可访问。移除全部元素后，在循环条件中调用 empty()以结束循环。也可以使用表达式 words_copy. size()来控制循环，因为返回值会被隐式转换为布尔值，这样在 size()返回 0 时，表达式的结果为 false。

如果为 words 输入：

```
one two three four five six seven
^Z
```

那么输出为：

```
two three six seven one four five
```

当然，如果需要多次输出 priority_queue 的内容，最好定义一个函数。这个函数应该是通用的，如下所示：

```
template<typename T>
void list_pq(std::priority_queue<T> pq, size_t count = 5)
{
  size_t n{count};
  while (!pq. empty())
  {
    std::cout << pq. top() << " ";
    pq. pop();
    if (--n) continue;
    std::cout << std::endl;
    n = count;
  }
  std::cout << std::endl;
}
```

参数是以传值方式传入的，因此这里会处理一个优先级队列的副本。它是一个适用于任何类型容器的函数模板，只要容器实现了用于向 ostream 输出的 operator<<()函数。如果没有设置第二个参数，默认每 5 个输出值一行。当然也可以定义一个适用于 queue 容器适配对象的函数模板。

可以如下所示使用 priority_queue 的成员函数 emplace()：

```
words. emplace("nine");
```

以字符串为参数调用 string 类的构造函数会在容器的适当位置生成一个对象。这比下面的语句更有效率：

```
words. push("nine");
```

这里编译器会在字符文字处插入一个 string 构造函数来生成 push()的参数，然后以这个临时 string 对象作为参数调用 push()。push()函数然后会调用 string 类的拷贝构造函数来将生成对象添加到容器中。

我们把这些代码段组织成一个完整的程序：

```
// Ex3_03. cpp
// Exercising a priority queue container adapter
#include <iostream> // For standard streams
#include <queue> // For priority_queue<T>
#include <string> // For string class
```

```
using std::string;

// List contents of a priority queue
template<typename T>
void list_pq(std::priority_queue<T> pq, size_t count = 5)
{
  size_t n {count};
  while (!pq. empty())
  {
    std::cout << pq. top() << " ";
    pq. pop();
    if (--n) continue;
    std::cout << std::endl;
    n = count;
  }
  std::cout << std::endl;
}
int main()
{
  std::priority_queue<std::string> words;
  std::string word;
  std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
               line to end:\n";
  while (true)
  {
    if ((std::cin >> word). eof())
      break;
    words. push(word);
  }
  std::cout << "You entered " << words. size() << " words. " << std::endl;
  list_pq(words);
}
```

一些示例输出如下：

```
Enter words separated by spaces, enter Ctrl+Z on a separate line to end:
one two three four five six seven eight nine ten eleven twelve
^Z
You entered 12 words:
two twelve three ten six
seven one nine four five
eleven eight
```

list_pq<T>()函数模板实例的输出表明优先级队列对输入进行了排序。

3.5　堆

堆(heaps)不是容器，而是一种特别的数据组织方式。堆一般用来保存序列容器。堆很

重要，很多不同的计算机进程中都使用了它们。为了弄明白堆是什么，首先需要明白树是什么，因此首先说明树这种数据结构是什么。

树是分层排列的元素或节点。每个节点有一个键，它是节点中所保存的对象——就像链表中的节点。父节点是有一个或两个子节点的节点。一般父节点可以有任意个数的子节点，树中的父节点不需要有相同个数的子节点。没有子节点的节点叫作叶节点。一般父节点的键与其子节点有一些关系。树都有一个根节点，它是树的基础，从根节点可以到达所有的子节点。

图 3-4 展示了一棵树，它表示 2014 年世界杯最后一组比赛的结果。德国全部赢了，所以它是根节点；它在最后一场比赛中打败了巴西队，所以它和巴西队是它自己的子节点。每个父节点最多有两个子节点的树叫作二叉树。图 3-4 中的树是一个完全二叉树，因为每个父节点都有两个子节点。任何树的父节点都有指向子节点的指针。完全二叉树可以用数组的方式保存，也可以用其他顺序表的方式保存，例如 vector，这样就不需要保存子节点的指针，因为知道每一层节点的编号。如果将每一层树的层数记作 n，从根节点开始作为第 0 层，每一层包含 2^n 个节点。图 3-4 展示了世界杯比赛树的节点如何存储在数组中。每个节点上的整数值是索引值。根节点存放在数组的第一个元素中，后面是它的两个子节点。这对子节点的孩子节点出现在序列的下个位置，以此类推直到叶节点。子节点的索引值为 n，那么它的父节点的索引值就为$(n-1)/2$。如果数组元素从 1 开始索引，那么父节点的索引表达式更加简单，它为 n/2。

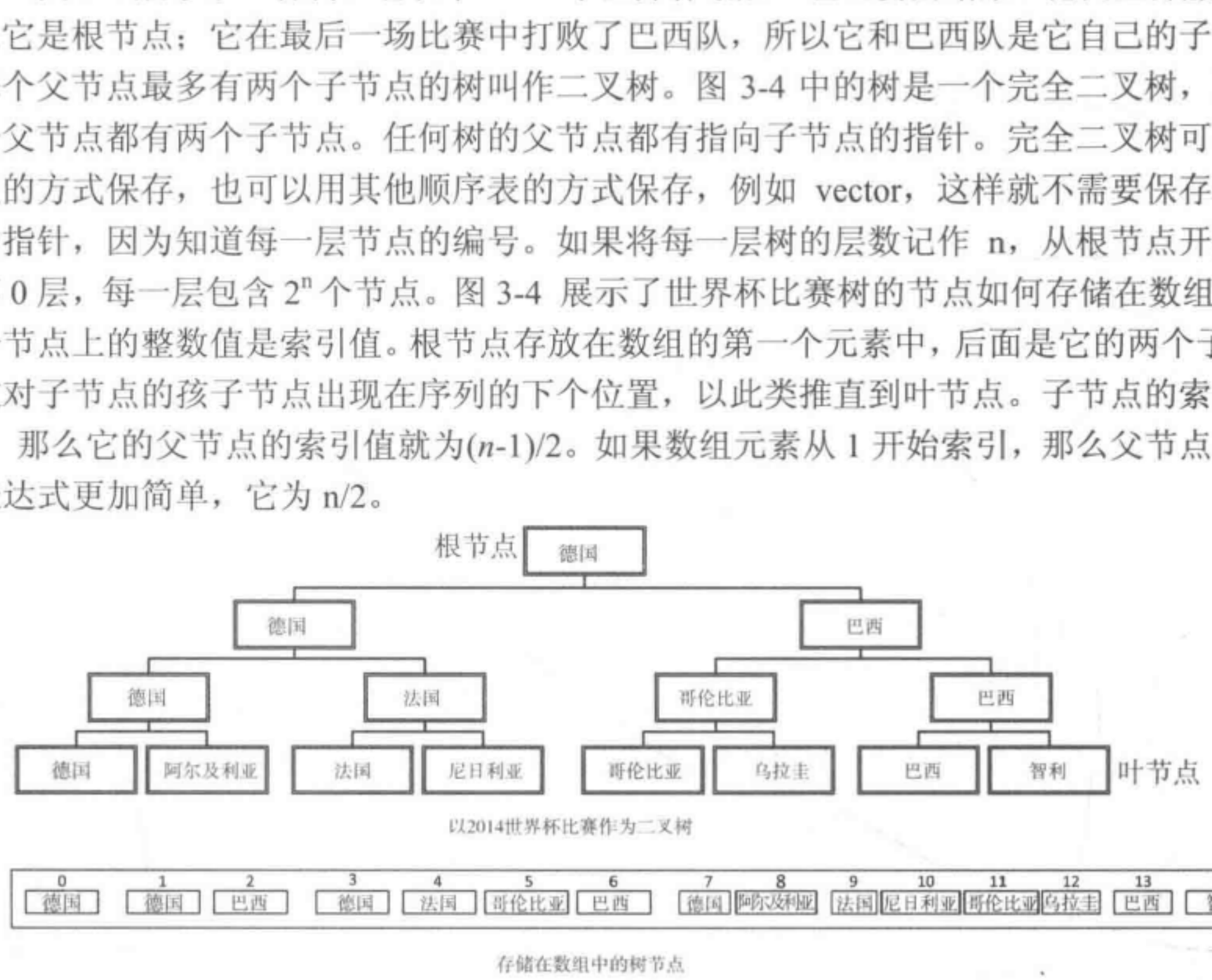


图 3-4　二叉树示例

现在可以定义一个堆：这个堆是一个完全二叉树，每个节点与其子节点位置相对。父节点总是大于或等于子节点，这种情况下被叫作大顶堆，或者父节点总是小于或等于子节点，这种情况下叫作小顶堆。注意给定父节点的子节点不一定按顺序排列。

3.5.1　创建堆

用来创建堆的函数定义在头文件 algorithm 中。max_heap()对随机访问迭代器指定的一段元素重新排列，生成一个堆。默认使用的是<运算符，可以生成一个大顶堆。例如：

```
std::vector<double> numbers { 2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
```

```
6. 0 };
    std::make_heap(std::begin(numbers), std::end(numbers));
    // Result: 12 10 3. 5 6. 5 8 2. 5 1. 5 6
```

调用 make_heap()后，vector 中的元素如注释所示，这也说明了图 3-5 所展示的结构。

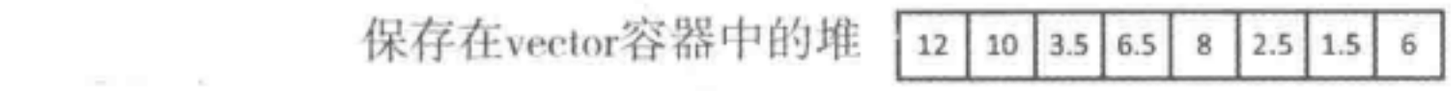


图 3-5　堆所表示的树

根节点是 12，10 和 3. 5 是它的子节点。10 的子节点是 6. 5 和 8，3. 5 的子节点是 2. 5 和 1. 5。6. 5 只有一个叶节点 6。

priority_queue 是一个堆。在底层，一个 priority_queue 实例创建了一个堆。这也是为什么图 3-3 没有展示底层容器的元素是如何排列的原因。在堆中，所有成对的连续元素不需要有相同的比较关系。图 3-5 所示堆中的前 3 个元素是顺序递减的，但第 4 个元素却大于第 3 个元素。既然如此，为什么 STL 有 priority_queue(它是一个堆)，却还需要创建堆，特别是还需要将堆作为优先级队列？

这是因为 priority_queue 可以提供堆没有的优势，它可以自动保持元素的顺序；但我们不能打乱 priority_queue 的有序状态，因为除了第一个元素，我们无法直接访问它的其他元素。如果需要的是一个优先级队列，这一点非常有用。

从另一方面来说，使用 make_heap()创建的堆可以提供一些 priority_queue 没有的优势：

- 可以访问堆中的任意元素，而不限于最大的元素，因为元素被存储在一个容器中，就像是我们自己的 vector。这也提供了偶然破坏元素顺序的可能，但是总可以调用 make_heap()来还原堆。
- 可以在任何提供随机访问迭代器的序列容器中创建堆。这些序列容器包括普通数组、string 对象、自定义容器。这意味着无论什么时候需要，都可以用这些序列容器的元素创建堆，必要时，可以反复创建。甚至还可以为元素的子集创建堆。

如果使用保持堆顺序的函数，那么可以将堆当作优先级队列使用。

这里有另一个版本的 make_heap()，它有第 3 个参数，可以用来指定一个比较函数用于堆的排序。通过定义一个大于运算符函数，可以生成一个小顶堆。这里可以使用 functional 中的断言。例如：

```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
```

```
    std::make_heap(std::begin(numbers), std::end(numbers),
                   std::greater<>()); // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5 10
```

可以将模板类型参数指定为 greater。这里的这个尖括号为空的版本推断并返回了类型参数。已经有一个用 make_heap()函数在容器中生成的堆。可以在它上面进行很多操作，下面我们来深入了解这些操作。

3.5.2 堆操作

堆不是容器，而是组织容器元素的一种特别方式。只能确定堆的范围——开始和结束迭代器指定的范围。这意味着可以用容器中的元素子序列创建堆。可以在已生成的堆中添加元素。乍一看，algorithm 中的函数模板 push_heap()创建堆的方式可能会觉得有些奇怪。为了向堆中添加元素，首先可以用任何方法将元素附加到序列中。然后调用 push_heap()来插入最后一个元素，为了保持堆的结构，这个元素会被重新排列到一个适当的位置。

```
    std::vector<double> numbers { 2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers));
    // Result: 12 10 3. 5 6. 5 8 2. 5 1. 5 6
    numbers. push_back(11); // Result: 12 10 3. 5 6. 5 8 2. 5 1. 5 6 11
    std::push_heap(std::begin(numbers), std::end(numbers));
    // Result: 12 11 3. 5 10 8 2. 5 1. 5 6 6. 5
```

注释显示了每个操作执行后的效果。必须以这种方式向堆中添加元素。只能通过调用成员函数向 queue 中添加新元素，而且这个成员函数只接受迭代器作为参数，不能直接以元素作为参数。push_back()会在序列末尾添加元素，然后使用 push_heap()恢复堆的排序。通过调用 push_heap()，释放了一个信号，指出我们向堆中添加了一个元素，这可能会导致堆排序的混乱。push_heap()会因此认为最后一个元素是新元素，为了保持堆结构，会重新排列序列。

从上面这个示例可以看出，重新排列是有必要的。我们注意到，尽管这个序列是一个堆，但是它的元素并不完全是按降序排列。这清楚地表明，尽管优先级队列是一个堆，但堆元素的顺序并不一定要和优先级队列相同。

当然，也可以用自己的比较函数来创建堆，但是必须和 push_heap()使用相同的比较函数：

```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers),
                   std::greater<>()); // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5 10
    numbers. push_back(1. 2);          // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5
10 1. 2
    std::push_heap(std::begin(numbers), std::end(numbers),
            std::greater<>()); // Result: 1. 2 1. 5 2. 5 6 8 12 3. 5 10 6. 5
```

如果push_heap()和 make_heap()的第 3 个参数不同，代码就无法正常执行。注释显示的

结果中，最后的 6. 5 似乎有些奇怪，图 3-6 展示的堆树能说明这个问题。

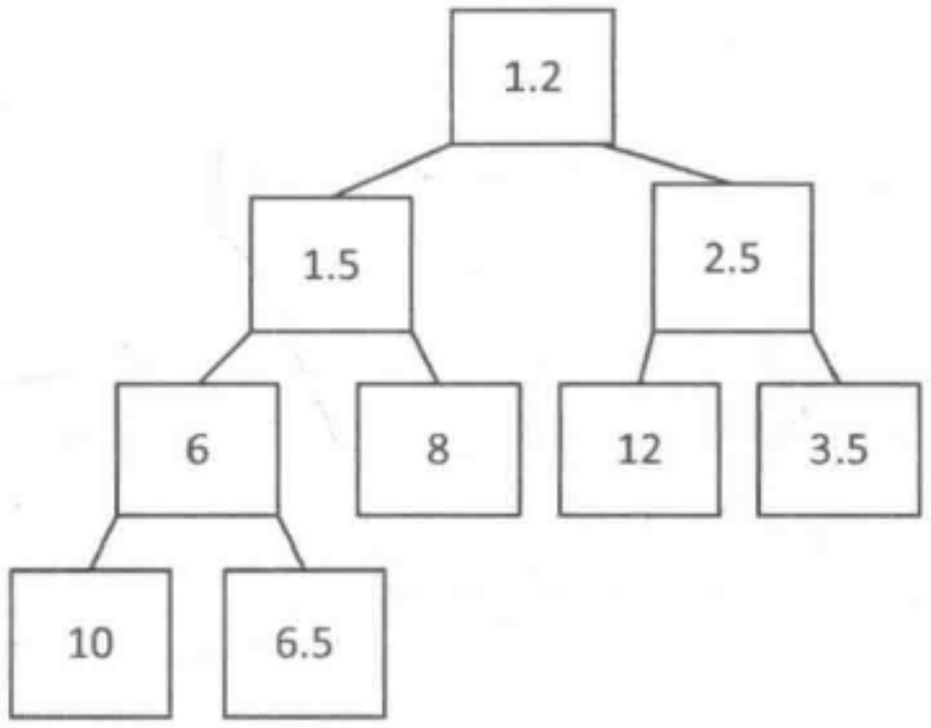


图 3-6　浮点值数堆

从树来看，显然 6. 5 是 6(而不是 10)的子节点，所以这个堆结构是正确的。

删除最大元素和添加元素到堆的过程有些相似，但所做的事是相反的。首先调用 pop_heap()，然后从容器中移除最大的元素，例如：

```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers));
    // Result: 12 10 3. 5 6. 5 8 2. 5 1. 5 6
    std::pop_heap(std::begin(numbers), std::end(numbers));
    // Result: 10 8 3. 5 6. 5 6 2. 5 1. 5 12
    numbers. pop_back(); // Result: 10 8 3. 5 6. 5 6 2. 5 1. 5
```

pop_heap()函数将第一个元素移到最后，并保证剩下的元素仍然是一个堆。然后就可以使用 vector 的成员函数 pop_back()移除最后一个元素。

如果 make_heap()中用的是自己的比较函数，那么 pop_heap()的第 3 个参数也需要是这个函数：

```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers),
                   std::greater<>()); // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5 10
    std::pop_heap(std::begin(numbers), std::end(numbers),
    std::greater<>());                // Result: 2. 5 6 3. 5 6. 5 8 12 10 1. 5
    numbers. pop_back();              // Result: 2. 5 6 3. 5 6. 5 8 12 10
```

从注释显示的操作结果来看，显然需要为 pop_heap()提供一个比较运算符函数。pop_heap()函数不会交换第一个元素和最后一个元素，它会对从 begin(numbers)到 end(numbers)-1 这个范围内的元素重新排序，从而保持堆的顺序。为了能够正确执行这个操作，pop_heap()必须和 make_heap()使用相同的比较函数。

因为可能会打乱容器中的堆，所以 STL 提供了一个检查序列是否仍然是堆的方法：

```
if (std::is_heap(std::begin(numbers), std::end(numbers)))
```

```
    std::cout << "Great! We still have a heap. \n";
else
    std::cout << "Oh bother! We messed up the heap. \n";
```

如果元素段是堆，那么 is_heap()会返回 true。这里是用默认的比较断言less<>来检查元素顺序。如果这里使用的是用 greater<>创建的堆，就会产生错误的结果。为了得到正确的结果，表达式需要写为:std::is_heap(std::begin(numbers),std::end(numbers),std::greater<>())。

甚至可以更深入地检查元素中是否有部分元素为堆。例如：

```
std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers),
                   std::greater<>()); // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5 10
    std::pop_heap(std::begin(numbers), std::end(numbers),
                   std::greater<>()); // Result: 2. 5 6 3. 5 6. 5 8 12 10 1. 5
    auto iter = std::is_heap_until(std::begin(numbers), std::end(numbers),
                  std::greater<>());
    if(iter != std::end(numbers))
      std::cout << "numbers is a heap up to " << *iter << std::endl;
```

is_heap_until()函数返回一个迭代器，指向第一个不在堆内的元素。这个代码段会输出最后一个元素的值 1.5，因为在调用 pop_heap()后，这个元素就不在堆内了。如果整段元素都是堆，函数会返回一个结束迭代器，因此 if 语句可以确保我们不会解引用一个结束迭代器。如果这段元素少于两个，也会返回一个结束迭代器。这里还有另一个版本的 is_heap_until()，它有两个参数，以 less<>作为默认断言。

STL 提供的最后一个操作是 sort_heap()，它会将元素段作为堆来排序。如果元素段不是堆，程序会在运行时崩溃。这个函数有以两个迭代器为参数的版本，迭代器指向一个假定的大顶堆(用 less<>排列)，然后将堆中的元素排成降序。结果当然不再是大顶堆。下面是一个使用它的示例：

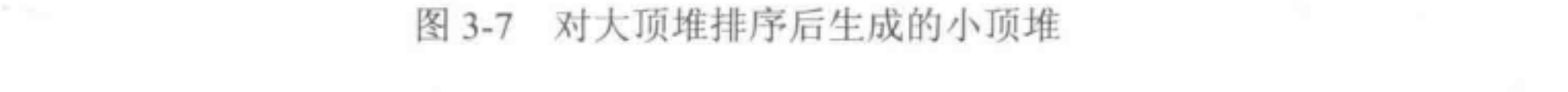
```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers));
    // Result: 12 10 3. 5 6. 5 8 2. 5 1. 5 6
    std::sort_heap(std::begin(numbers), std::end(numbers));
    // Result: 1. 5 2. 5 3. 5 6 6. 5 8 10 12
```

排序操作的结果不是一个大顶堆，而是一个小顶堆。如图 3-7 所示，尽管堆并不是全部有序的，但任何全部有序的序列都是堆。

第 2 个版本的 sort_heap()有第 3 个参数，可以指定一个用来创建堆的断言。如果用断言 greater<>来创建堆，会生成一个小顶堆，对它进行排序会生成一个降序序列。排序后的序列不是小顶堆。下面的代码对此做了展示：

```
    std::vector<double> numbers {2. 5, 10. 0, 3. 5, 6. 5, 8. 0, 12. 0, 1. 5,
6. 0};
    std::make_heap(std::begin(numbers), std::end(numbers),
```

```
                std::greater<>()); // Result: 1. 5 6 2. 5 6. 5 8 12 3. 5 10
std::sort_heap(std::begin(numbers), std::end(numbers),
                std::greater<>()); // Result: 12 10 8 6. 5 6 3. 5 2. 5 1. 5
```



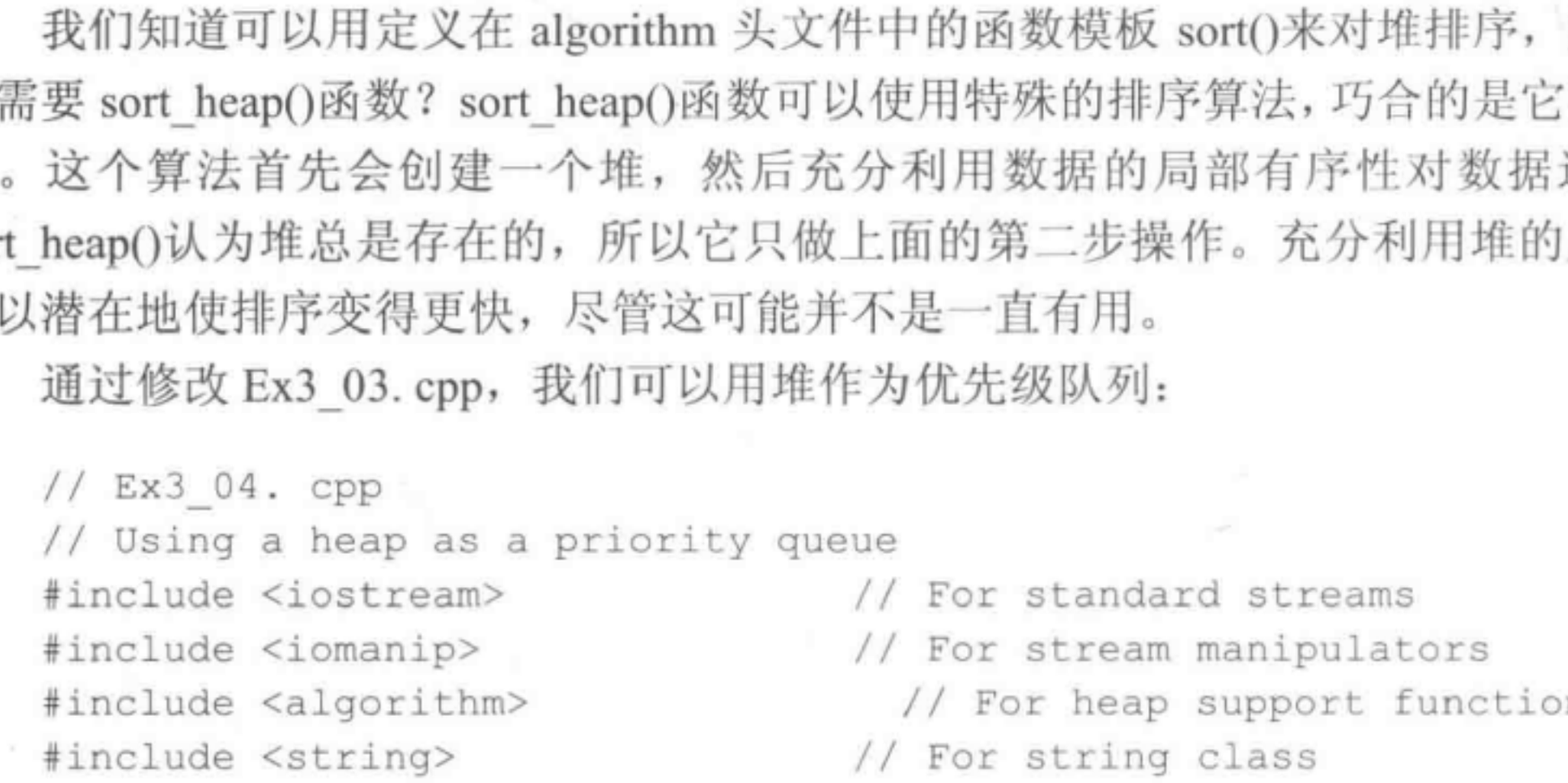


图 3-7　对大顶堆排序后生成的小顶堆

如最后一行注释中显示的那样，对小顶堆执行 sort_heap()后，会变成一个大顶堆。

我们知道可以用定义在 algorithm 头文件中的函数模板 sort()来对堆排序，那么为什么还需要 sort_heap()函数？sort_heap()函数可以使用特殊的排序算法，巧合的是它被叫作堆排序。这个算法首先会创建一个堆，然后充分利用数据的局部有序性对数据进行排序。sort_heap()认为堆总是存在的，所以它只做上面的第二步操作。充分利用堆的局部有序性可以潜在地使排序变得更快，尽管这可能并不是一直有用。

通过修改 Ex3_03. cpp，我们可以用堆作为优先级队列：

```
// Ex3_04. cpp
// Using a heap as a priority queue
#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
#include <algorithm>                     // For heap support functions
#include <string>                      // For string class
#include <deque>                       // For deque container
using std::string;

// List a deque of words
void show(const std::deque<string>& words, size_t count = 5)
{
  if(words. empty()) return;           // Ensure deque has elements
     // Find length of longest string
     auto max_len = std::max_element(std::begin(words), std::end(words),
                                     [](const string& s1, const string& s2)
                                     {return s1. size() < s2.
size(); })->size();
     // Output the words
     size_t n {count};
     for(const auto& word : words)
     {
```

```
        std::cout << std::setw(max_len + 1) << word << " ";
        if(--n) continue;
        std::cout << std::endl;
        n = count;
      }
      std::cout << std::endl;
    }

    int main()
    {
    std::deque<string> words;
    std::string word;
    std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
                 line to end:\n";
    while (true)
    {
      if ((std::cin >> word). eof())
      {
        std::cin. clear();
        break;
      }
      words. push_back(word);
    }
    std::cout << "The words in the list are:" << std::endl;
    show(words);

    std::make_heap(std::begin(words), std::end(words));
    std::cout << "\nAfter making a heap, the words in the list are:"
                 << std::endl;
    show(words);
    std::cout << "\nYou entered " << words. size() << " words.  Enter some more:"
                 << std::endl;
    while (true)
    {
      if ((std::cin >> word). eof())
      {
        std::cin. clear();
        break;
      }
      words. push_back(word);
      std::push_heap(std::begin(words), std::end(words));
      }
      std::cout << "\nThe words in the list are now:" << std::endl;
      show(words);
    }
```

下面是一些示例输出：

```
Enter words separated by spaces, enter Ctrl+Z on a separate line to end:
one two three four five six seven
```

```
^Z
The words in the list are:
one two three four five
six seven
After making a heap, the words in the list are:
two one three four five
six seven
You entered 7 words.  Enter some more:
eight nine ten twelve fifteen ninety forty fifty-three
^Z
The words in the list are now:
two twelve three nine ten
six seven eight four five
one fifteen ninety forty fifty-three
```

这个示例在一个 deque 容器中创建了一个堆，这和之前的示例不同：这里也可以使用 vector 容器。show()函数可以列出 deque<string>容器中的所有单词。为了能够整齐地输出，单词都以比最大单词长度长 1 的固定宽度输出。可以使用定义在 algorithm 头文件中的 max_element()函数来计算单词最大长度。通过使用提供的比较函数，max_element()会返回一个指向最大元素的迭代器。前两个参数是指定序列范围的迭代器。第 3 个参数是一个用于比较运算的 lambda 表达式。

注意 max_element()函数需要定义小于而不是大于运算，用来查找最大元素。比较函数的形式如下：

```
bool comp(const T1& a, const T2& b);
```

大多数情况下，第一个参数和第二个参数的类型相同，但有时类型也可以不同。唯一的要求是，这个范围内的元素需要可以隐式转换为 T1、T2 类型。参数不需要指定为 const，但最好这样做。在任何情况下，比较函数都不能改变传给它的参数值。

lambda 表达式可以返回字符串的 size()值的比较结果。max_element()返回的迭代器指向最长的字符串，因此可以调用它的成员函数 size()来将它的长度记录到 max_len 中。

用我们之前见过的方式从 cin 中读取单词。这里调用 cin 的成员函数 clear()来清除 EOF 状态，这个状态是在输入 Ctrl+Z 时设置的。如果不调用 clear()，EOF 状态会继续保留，这样后面就无法再从标准输入流获取输入了。

读入一些单词序列后，通过调用 make_heap()函数将 deque 容器中的内容排成堆。然后读取一些单词，在将每个单词添加到容器时，需要调用 push_heap()来保持堆序。push_heap()希望新元素被添加在容器的尾部；如果使用 push_front()，程序会因此崩溃，因为这时候堆是无效的。输出表明所有代码按预期工作。

当然，如果每次输入单词后，都使用 push_heap()，就不需要调用make_heap()。该例展示了如何使用我们控制的底层容器来访问全部元素，并且保留它们，而不需要像使用优先级队列那样在使用前不得不先备份它。

3.6 在容器中保存指针

通常用容器保存指针比保存对象更好，而且大多数时候，保存智能指针比原生指针好。下面是一些原因：

- 在容器中保存指针需要复制指针而不是它所指向的对象。复制指针通常比复制对象快。
- 在容器中保存指针可以得到多态性。存放元素基类指针的容器也可以保存其派生类型的指针。当要处理有共同基类的任意对象序列时，这种功能是非常有用的。应用这一特性的一个常见示例是展示一个含有直线、曲线和几何形状的对象序列。
- 对指针容器的内容进行排序的速度要比对对象排序快；因为只需要移动指针，不需要移动对象。
- 保存智能指针要比保存原生指针安全，因为在对象不再被引用时，自由存储区的对象会被自动删除。这样就不会产生内存泄漏。不指向任何对象的指针默认为 nullptr。

如你所知，主要有两种类型的智能指针：unique_ptr<T> 和 shared_ptr<T>，其中 unique_ptr<T>独占它所指向对象的所有权，而 shared_ptr<T>允许多个指针指向同一个对象。还有 weak_ptr<T>类型，它是一类从 shared_ptr<T>生成的智能指针，可以避免使用 shared_ptrs<T>带来的循环引用问题。unique_ptr<T>类型的指针可以通过移动的方式保存到容器中。例如，下面的代码可以通过编译：

```
std::vector<std::unique_ptr<std::string>> words;
words. push_back(std::make_unique<std::string>("one"));
words. push_back(std::make_unique<std::string>("two"));
```

vector 保存了 unique_ptr<string>类型的智能指针。make_unique<T>()函数可以生成对象和智能指针，并且返回后者。因为返回结果是一个临时 unique_ptr<string>对象，这里调用一个有右值引用参数的 push_back()函数，因此不需要拷贝对象。另一种添加 unique_ptr 对象的方法是，先创建一个局部变量 unique_ptr，然后使用 std::move()将它移到容器中。然而，后面任何关于拷贝容器元素的操作都会失败，因为只能有一个 unique_ptr 对象。如果想能够复制元素，需要使用 shared_ptr 对象；否则就使用 unique_ptr 对象。

3.6.1 在序列容器中保存指针

下面首先解释一些在容器中使用原生指针会碰到的问题，然后再使用智能指针——这是推荐的使用方式。下面是一段代码，用来从标准输入流读取单词，然后将指向自由存储区的字符串对象的指针保存到 vector 容器中：

```
    std::vector<std::string*> words;
    std::string word;
    std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
line to end:\n";
    while (true)
```

```
{
  if ((std::cin >> word). eof())
  {
    std::cin. clear();
    break;
  }
  words. push_back(new std::string {word}); // Create object and store its
                                            // address
}
```

push_back()的参数表达式在自由存储区生成了一个字符串对象，因此 push_back()的参数是一个对象的地址。可以按如下方式输出 words 中的内容：

```
for (auto& w : words)
  std::cout << w << " ";
std::cout << std::endl;
```

如果想使用迭代器来访问容器中的元素，输出字符串的代码可以这样写：

```
for (auto iter = std::begin(words); iter != std::end(words); ++iter)
  std::cout << **iter << " ";
std::cout << std::endl;
```

iter是一个迭代器，必须通过解引用来访问它所指向的元素。这里，容器的元素也是指针，因此必须解引用来获取 string 对象。因此表达式为：**iter。

注意在删除元素时，需要先释放它所指向的内存。如果不这样做，在删除指针后，就无法释放它所指向的内存，除非保存了指针的副本。这是容器中的原生指针常见的内存泄漏来源。下面演示它如何在 words 中发生：

```
for (auto iter = std::begin(words); iter != std::end(words) ; )
{
  if (**iter == "one")
    words. erase(iter);      // Memory leak!
  else
    ++iter;
}
```

这里删除了一个指针，但它所指向的内存仍然存在。无论什么时候删除一个是原生指针的元素，都需要首先释放它所指向的内存：

```
for (auto iter = std::begin(words); iter != std::end(words) ; )
{
  if (**iter == "one")
  {
    delete *iter;            // Release the memory. . .
    words. erase(iter);      // . . . then delete the pointer
  }
  else
    ++iter;
}
```

在离开 vector 的使用范围之前，记住要删除自由存储区的 string 对象。可以按如下方式来实现：

```
for (auto& w : words)
  delete w;                    // Delete the string pointed to
words. clear();                // Delete all the elements from the vector
```

用索引来访问指针，这样就可以使用 delete 运算符删除 string 对象。当循环结束时，vector 中的所有指针元素都会失效，因此不要让 vector 处于这种状态。调用 clear()移除所有元素，这样 size()会返回 0。当然，也可以像下面这样使用迭代器：

```
for (auto iter = std::begin(words); iter != std::end(words); ++iter)
  delete *iter;
```

如果保存了智能指针，就不用担心要去释放自由存储区的内存。智能指针会做这些事情。下面是一个读入字符串，然后把 shared_ptr<string>保存到 vector 中的代码片段：

```
std::vector<std::shared_ptr<std::string>> words;
std::string word;
std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
line to end:\n";
while (true)
{
if ((std::cin >> word). eof())
{
std::cin. clear();
break;
}
words. push_back(std::make_shared<string>(word)); // Create smart pointer
to string & store it
}
```

这和使用原生指针的版本没有什么不同。vector 模板现在的类型参数是 std::shared_ptr<std::string>，push_back()的参数会调用 make_shared()，在自由存储区生成 string 对象和一个指向它的智能指针。因为智能指针由参数表达式生成，这里会调用一个右值引用参数版的 push_back()来将指针移到容器中。

模板类型参数可能有些冗长，但是可以使用 using 来简化代码。例如：

```
using PString = std::shared_ptr<std::string>;
```

使用 using 后，可以这样定义：

```
std::vector<PString> words;
```

可以通过智能指针元素来访问字符串，这和使用原生指针相同。前面那些输出 words 内容的代码片段都可以使用智能指针。当然，不需要删除自由存储区的 string 对象；因为智能指针会做这些事情。执行 words. clear()会移除全部的元素，因此会调用智能指针的析构函数；这也会导致智能指针释放它们所指向对象的内存。

为了阻止 vector 太频繁地分配额外内存，可以先创建 vector，然后调用 reserve()来分配一定数量的初始内存。例如：

```
std::vector<std::shared_ptr<std::string>> words;
words. reserve(100);                    // Space for 100 smart pointers
```

这样生成 vector 比指定元素个数来生成要好，因为每一个元素都是通过调用 shared_ptr<string>构造函数生成的。不这样做也不是什么大问题，但会产生一些不必要的额外开销，即使开销很小。通常，每个智能指针所需要的空间远小于它们所指向对象需要的空间，因此可以大方地使用 reserve()来分配空间。

可以在外面使用保存的 shared_ptr<T>对象的副本。如果不需要这种功能，应该使用 unique_ptr<T>对象。下面展示如何在 words 中这样使用：

```
    std::vector<std::unique_ptr<std::string>> words;
    std::string word;
    std::cout << "Enter words separated by spaces, enter Ctrl+Z on a separate
line to end:\n";
    while (true)
    {
      if ((std::cin >> word). eof())
      {
        std::cin. clear();
        break;
      }
      words. push_back(std::make_unique<string>(word));
      // Create smart pointer to string & store it
    }
```

在上面的代码中，用'unique'代替'shared'是没有差别的。

我们看一下，如何使用智能指针来实现先前章节中 Ex3_02 的超市结账模拟程序。Customer 类的定义和之前的版本相同，但是 Checkout 类的定义中使用了智能指针，因而产生了一些变化，我们也可以在 main()中使用智能指针。在整个程序中，我们都不需要使用智能指针的副本，因此我们选择使用 unique_ptr<T>。下面是 Checkout. h 头文件中的新内容：

```
// Supermarket checkout - using smart pointers to customers in a queue
#ifndef CHECKOUT_H
#define CHECKOUT_H
#include <queue>                        // For queue container
#include <memory>                       // For smart pointers
#include "Customer. h"
using PCustomer = std::unique_ptr<Customer>;

class Checkout
{
private:
  std::queue<PCustomer> customers; // The queue waiting to checkout

public:
```

```
    void add(PCustomer&& customer) { customers. push(std::move(customer)); }
    size_t qlength() const { return customers. size(); }
    // Increment the time by one minute
    void time_increment()
    {
      if (customers. front()->time_decrement(). done())
      // If the customer is done. . .
        customers. pop(); // . . . remove from the queue
    };
```

```
    bool operator<(const Checkout& other) const { return qlength() < other.
qlength(); }
    bool operator>(const Checkout& other) const { return qlength() < other.
qlength(); }
  };
  #endif
```

我们需要直接包含 memory 头文件，这样就可以使用智能指针类型的模板。queue 容器保存 PCustomer 元素，用来记录排队结账的顾客。使用 using 为 std::unique_ptr<Customer> 定义了一个别名 PCustomer，这可以节省大量的输入。PCustomer 对象不能被复制，因而当调用 add()函数时，它的参数是右值引用，参数会被移到容器中。以 unique 指针作为元素时，也会以同样的方式被移到容器中；当然，参数不能是 const。做了这些修改后，就可以使用 unique_ptr 了，不再需要修改其他的内容。

```
// Ex3_05. cpp
// Using smart pointer to simulate supermarket checkouts
#include <iostream>                          // For standard streams
#include <iomanip>                           // For stream manipulators
#include <vector>                            // For vector container
#include <string>                            // For string class
#include <numeric>                           // For accumulate()
#include <algorithm>                         // For min_element & max_element
#include <random>                            // For random number generation
#include <memory>                            // For smart pointers
#include "Checkout. h"
#include "Customer. h"

using std::string;
using distribution = std::uniform_int_distribution<>;
using PCheckout = std::unique_ptr<Checkout>;

// Output histogram of service times
void histogram(const std::vector<int>& v, int min)
{
  string bar (60, '*'); // Row of asterisks for bar
  for (size_t i {}; i < v. size(); ++i)
  {
    std::cout << std::setw(3) << i+min << " " // Service time is index + min
      << std::setw(4) << v[i] << " " // Output no.  of occurrences
```

```
        << bar. substr(0, v[i]) // . . . and that no.  of asterisks
        << (v[i] > static_cast<int>(bar. size()) ? ". . . " : "")
        << std::endl;
  }
}
int main()
{
  std::random_device random_n;

  // Setup minimum & maximum checkout periods - times in minutes
      int service_t_min {2}, service_t_max {15};
      std::uniform_int_distribution<> service_t_d {service_t_min,
service_t_max};

      // Setup minimum & maximum number of customers at store opening
      int min_customers {15}, max_customers {20};
      distribution n_1st_customers_d {min_customers, max_customers};

      // Setup minimum & maximum intervals between customer arrivals
      int min_arr_interval {1}, max_arr_interval {5};
      distribution arrival_interval_d {min_arr_interval, max_arr_interval};
      size_t n_checkouts {};
      std::cout << "Enter the number of checkouts in the supermarket: ";
      std::cin >> n_checkouts;
      if(!n_checkouts)
      {
        std::cout << "Number of checkouts must be greater than 0.  Setting to
                1. "
                << std::endl;
        n_checkouts = 1;
      }

      std::vector<PCheckout> checkouts;
      checkouts. reserve(n_checkouts); // Reserve memory for pointers

      // Create the checkouts
      for (size_t i {}; i < n_checkouts; ++i)
        checkouts. push_back(std::make_unique<Checkout>());
      std::vector<int> service_times(service_t_max-service_t_min+1);

      // Add customers waiting when store opens
      int count {n_1st_customers_d(random_n)};
      std::cout << "Customers waiting at store opening: " << count << std::endl;
      int added {};
      int service_t {};

      // Define comparison lambda for pointers to checkouts
      auto comp = [](const PCheckout& pc1, const PCheckout& pc2){ return *pc1
< *pc2; };
      while (added++ < count)
      {
        service_t = service_t_d(random_n);
```

```
    auto iter = std::min_element(std::begin(checkouts),
                                 std::end(checkouts), comp);
    (*iter)->add(std::make_unique<Customer>(service_t));
    ++service_times[service_t - service_t_min];
  }

  size_t time {}; // Stores time elapsed
  const size_t total_time {600}; // Duration of simulation - minutes
  size_t longest_q {}; // Stores longest checkout queue length

  // Period until next customer arrives
  int new_cust_interval {arrival_interval_d(random_n)};

  // Run store simulation for period of total_time minutes
  while (time < total_time) // Simulation loops over time
  {
    ++time; // Increment by 1 minute

    // New customer arrives when arrival interval is zero
    if (--new_cust_interval == 0)
    {
      service_t = service_t_d(random_n); // Random customer service time
      (*std::min_element(std::begin(checkouts),
          std::end(checkouts), comp))->add(std::make_unique<Customer>
            (service_t));
      ++service_times[service_t - service_t_min]; // Record service time
      // Update record of the longest queue length
      for (auto& pcheckout : checkouts)
        longest_q = std::max(longest_q, pcheckout->qlength());
      new_cust_interval = arrival_interval_d(random_n);
    }
    // Update the time in the checkouts - serving the 1st customer in each queue
    for (auto& pcheckout : checkouts)
      pcheckout->time_increment();

  }

  std::cout << "Maximum queue length = " << longest_q << std::endl;
  std::cout << "\nHistogram of service times:\n";
  histogram(service_times, service_t_min);

  std::cout << "\nTotal number of customers today: "
            << std::accumulate(std::begin(service_times), std::end(service_times), 0)
            << std::endl;
}
```

vector 容器现在保存的是指向 Checkout 对象的 unique 指针。vector 的迭代器指向 Checkout 对象——unique_ptr<Checkout>对象的指针，因而可以通过迭代器来调用 Checkout 的成员函数。首先必须解引用迭代器，然后用间接成员选择运算符来调用函数。可以看到，我们已经修改了 main()中的相关代码。min_element()默认使用<运算符来从迭代器指向的元

素中获取结果。默认会比较智能指针，但是并不能得到正确的结果。我们需要为 min_element()提供第 3 个参数作为它所使用的比较函数。这个函数是由名为 comp 的 lambda 表达式定义的。因为我们想在后面继续使用这个表达式，所以对它做命名。为了访问 Checkout 对象，这个 lambda 表达式解引用了智能指针参数，然后使用 Checkout 类的成员函数 operator<()来比较它们。所有的 Checkout 和 Customer 对象都是在自由储存区生成的。智能指针会维护它们所使用的内存。这个版本的模拟程序的输出和之前版本的相同。在这个示例中也可以使用 shared_ptr<T>，但是它们会执行得慢一些。就执行时间和内存使用而言，unique_ptr<T>对象相对于原生指针的开销最小。

3.6.2　在优先级队列中存储指针

现在主要讲解智能指针的使用。这和原生指针在本质上是相同的，除非想要自己负责删除它们所指向的对象。当生成优先级队列或堆时，需要一个顺序关系来确定元素的顺序。当它们保存的是原生指针或智能指针时，总是需要为它们提供一个比较函数；如果不提供，就会对它们所保存的指针而不是指针所指向的对象进行比较，这肯定不是我们所希望的。让我们考虑一下，如何定义一个保存指针的 priority_queue，指针则指向自由存储区的 string 对象。为了缩短代码中语句的长度，假定下面的 using 声明对本节随后的内容都生效：

```
using std::string;
using std::shared_ptr;
using std::unique_ptr;
```

我们需要定义一个用来比较对象的函数对象，被比较的对象由 shared_ptr<string>类型的指针指向。按如下所示定义比较函数：

```
auto comp = [] (const shared_ptr<string>& wp1, const shared_ptr<string>& wp2)
                { return *wp1 < *wp2; };
```

这里定义了一个 lambda 表达式 comp，可以用来比较两个智能指针所指向的对象。对 lambda 表达式命名的原因是需要将它作为 priority_queue 模板的类型参数。下面是优先级队列的定义：

```
    std::priority_queue<shared_ptr<string>, std::vector<shared_ptr<string>>,
decltype(comp)>words1 {comp};
```

第一个模板类型参数是所保存元素的类型，第二个用来保存元素的容器的类型，第三个是用来比较元素的函数对象的类型。我们必须指定第三个模板类型参数，因为 lambda 表达式的类型和默认比较类型 std::less<T>不同。

仍然可以指定一个外部容器来初始化存放指针的 priority_queue：

```
std::vector<shared_ptr<string>> init { std::make_shared<string>("one"),
                        std::make_shared<string>("two"),
                        std::make_shared<string>("three"),
                        std::make_shared<string>("four")};
```

```
    std::priority_queue<shared_ptr<string>, std::vector<shared_ptr<string>>,
decltype(comp)>words(comp, init);
```

vector 容器 init 是用 make_shared<string>()生成的初始值。优先级队列构造函数的参数是一个定义了元素比较方式的对象以及一个提供初始化元素的容器。先复制 vector 中的智能指针，然后用它们来初始化优先级队列 words。当然，也可以用另一个容器中的元素来初始化优先级队列，这意味不能使用 unique_ptr<string>元素——它们必须是 shared_ptr<string>。

如果初始元素不需要保留，可以用 priority_queue 对象的成员函数 emplace()直接在容器中生成它们：

```
    std::priority_queue<shared_ptr<string>, std::vector<shared_ptr<string>>,
decltype(comp)>words1 {comp};
    words1. emplace(new string {"one"});
    words1. emplace(new string {"two"});
    words1. emplace(new string {"three"});
    words1. emplace(new string {"five"});
```

words1 的成员函数 emplace()会调用它们所保存对象的类型的构造函数，也就是 shared_ptr<string>的构造函数。这个构造函数的参数是自由存储区的一个string 对象的地址，string 对象是由 emplace()的参数表达式生成的。这段代码会在优先级队列中保存 4 个 string 对象的指针，它们分别指向"two"、"three"、"one"、"five"这 4 个对象。元素在优先级队列中的顺序取决于这一节前面定义的 comp。

当然，如果不需要保留初始元素，可以用优先级队列保存 unique_ptr<string>元素。例如：

```
    auto ucomp = [](const std::unique_ptr<string>& wp1, const
std::unique_ptr<string>& wp2){ return *wp1 < *wp2; };
    std::priority_queue<std::unique_ptr<string>,
std::vector<std::unique_ptr<string>>,decltype(ucomp)> words2 {ucomp};
```

这个定义比较运算的 lambda 表达式可以接受 unique_ptr<string>对象的引用。我们需要为这个优先级队列指定 3 个模板类型参数，因为我们需要指定比较函数的类型。第二个模板类型参数可以是 deque<string>，它是默认使用的容器类型。也可以用 emplace()向优先级队列中添加元素：

```
    words2. emplace(new string{"one"});
    words2. emplace(new string {"two"});
    words2. emplace(new string {"three"});
    words2. emplace(new string {"five"});
```

或者，也可以使用 push()：

```
    words2. push(std::make_unique<string>("one"));
    words2. push(std::make_unique<string>("two"));
    words2. push(std::make_unique<string>("three"));
    words2. push(std::make_unique<string>("five"));
```

make_unique<string>()返回的对象会被移到容器中，这是由右值引用参数版的 push()自动选择的。

3.6.3　指针的堆

当创建一个指针的堆时，需要提供一个用来比较对象的函数指针，下面是一个示例：

```
std::vector<shared_ptr<string>> words
          { std::make_shared<string>("one"), std::make_shared<string>("two"),
            std::make_shared<string>("three"), std::make_shared
              <string>("four") };
std::make_heap(std::begin(words), std::end(words),
          [](const shared_ptr<string>& wp1, const shared_ptr<string>&
          wp2){ return *wp1 < *wp2; });
```

make_heap()的第三个参数是一个定义了比较函数的 lambda 表达式，它通过解引用智能指针来比较两个 string 对象。make_heap()函数的模板有一个函数对象类型的参数。不像优先级队列适配器的类模板，这个模板参数没有默认的参数值，编译器会从这个函数调用的第三个参数来推断函数对象的类型。如果这个函数的第三个类型参数有默认的具体类型，就需要指定类型参数，优先级队列中也需要这样使用。

需要为下面任何函数调用——push_heap()、pop_heap()、 is_heap()、is_heap_until()提供和 make_heap()的最后一个参数相同的比较函数。当然，这里可以使用命名的 lambda 表达式。如果调用 sort_heap()，也需要提供一个比较函数。

3.6.4　基类指针的容器

可以在任何以基类类型元素定义的容器或容器适配器中保存派生类的指针。这使我们可以使用容器元素指向的对象的多态行为。我们需要一个基类和一个派生类来探讨这种可能性，因此让我们重新使用前面章节中 Ex2_06 的 Box 类，需要对它做一点修改。修改后的类定义如下：

```
class Box
{
protected:
  size_t length {};
  size_t width {};
  size_t height {};

public:
  explicit Box(size_t l=1, size_t w=1, size_t h=1) : length {l}, width {w},
height {h} {}
  virtual ~Box() = default;

  virtual double volume() const;    // Volume of a box

  // Comparison operators for Box object
```

```
    bool operator<(const Box& box) const;
    bool operator==(const Box& box) const;

    // Stream input and output
    virtual std::istream& read(std::istream& in);
    virtual std::ostream& write(std::ostream& out) const;
};
```

我们会从 Box 派生 Carton 类，因而成员变量是受保护的，并且将析构函数指定为虚函数。这里并不是必须这样做，但将基类的析构函数声明为虚函数是一个好习惯。这会产生一些额外开销，但可以避免派生类调用错误的析构函数。成员函数 volume()和两个 I/O 操作的成员函数 read()、write()都是虚函数，所以在需要时，可以在派生类中重写它们。

Box. h 中的 volume()函数应该按如下所示定义为内联函数：

```
inline double Box::volume() const { return length*width*height; }
```

比较函数也应该是内联函数：

```
// Less-than operator
inline bool Box::operator<(const Box& box) const
{ return volume() < box. volume(); }

//Equality comparion operator
inline bool Box::operator==(const Box& box) const
{
  return length == box. length && width == box. width && height == box. height;
}
```

小于运算符函数可以比较操作数的体积。恒等运算符可以比较数据成员的值。我们还需要为定义在 Box. h 中的 Box 对象重载流的输入和输出运算符：

```
// Stream extraction operator
inline std::istream& operator>>(std::istream& in, Box& box)
{
  return box. read(in);
}

// Stream insertion operator
inline std::ostream& operator<<(std::ostream& out, Box& box)
{
  return box. write(out);
}
```

这些运算符函数会调用适当的成员函数，read()或 write()来执行操作。这两个成员函数是虚函数，如果它们在派生类中被重新定义，这些运算符函数会调用第二个参数对象类型的 read()或 write()。

Box. h 的完整内容如下：

```
// Box. h
```

```
// Defines the Box class that will be a base for the Carton class
#ifndef BOX_H
#define BOX_H
#include <iostream>           // For standard streams
#include <istream>            // For stream classes
#include <utility>            // For comparison operator templates
using namespace std::rel_ops;  // Comparison operator template namespace

// Definition for the Box class as above. . .

// Definitions for inline functions as above. . .
#endif
```

成员函数 read()和 write()定义在一个单独的源文件 Box. cpp 中，该源文件的内容如下：

```
// Box. cpp
// Function members of the Box class
#include <iostream>
#include "Box. h"

// Read a Box object from a stream
std::istream& Box::read(std::istream& in)
{
  size_t value {};
  if ((in >> value). eof())
    return in;
  length = value;
  in >> width >> height;
  return in;
}
// Write a Box object to a stream
std::ostream& Box::write(std::ostream& out) const
{
  out << typeid(*this). name() << "(" << length << "," << width << ","
      << height << ")";
  return out;
}
```

它们都是直接定义的。注意当读到文件末尾时，read()会在 istream 对象中设置 EOF 指示符；如你所知，当从键盘读到 EOF(或从文件中读到 EOF)时，就会产生这种结果，因此可以用此种方法来检测 Box 对象的尺寸序列是否输入完毕。write()函数可以在类型名后的圆括号中输出对象的尺寸。当前对象的类型名是通过调用 type_info 对象的成员函数 name()获取到的，type_info 对象是使用 typeid 运算符生成的。这样每个对象输出的特定类型都可以标识它们的输出。

派生类 Carton 定义在头文件 Carton. h 中，它的内容如下：

```
// Carton. h
#ifndef CARTON_H
#define CARTON_H
```

```
#include "Box. h"

class Carton :public Box
{
public:
  explicit Carton(size_t l = 1, size_t w = 1, size_t h = 1) : Box {l, w, h}{}
  double volume() const override {return 0. 85*Box::volume(); }
};
#endif
```

这里的代码并不多。只重写了成员函数volume(),它可以返回普通Box对象体积的85%。显然 Carton 会有更多的厚边或内部包装，但最重要的是它和相同尺寸的 Box 体积不同，因此我们知道什么时候需要调用重写版本的 volume()。

下面是一个用指针在容器中存储不同类型对象的程序：

```
// Ex3_06. cpp
// Storing derived class objects in a container of base pointers
#include <iostream>              // For standard streams
#include <memory>                // For smart pointers
#include <vector>                // For the vector container
#include "Box. h"
#include "Carton. h"
using std::unique_ptr;
using std::make_unique;
int main()
{
  std::vector<unique_ptr<Box>> boxes;
  boxes. push_back(make_unique<Box>(1, 2, 3));
  boxes. push_back(make_unique<Carton>(1, 2, 3));
  boxes. push_back(make_unique<Carton>(4, 5, 6));
  boxes. push_back(make_unique<Box>(4, 5, 6));
  for(auto&& ptr : boxes)
    std::cout << *ptr << " volume is " << ptr->volume() << std::endl;
}
```

不需要对代码做太多解释。vector 保存了 std::unique_ptr<Box>类型的元素，因而可以用来保存指向 Box 对象的智能指针或指向任意以 Box 作为基类或间接基类的对象类型的智能指针。boxes 容器中填充了一些指向 Box 或 Carton 对象的智能指针，这些对象的指针是通过调用右值引用版的 push_back()函数插入的。使用 using 可以增强代码的可读性。用 for 循环输出 vector 中元素所指向对象的一些细节。for 循环中 ptr 的推导类型有一个右值。

下面是输出：

```
class Box(1,2,3) volume is 6
class Carton(1,2,3) volume is 5. 1
class Carton(4,5,6) volume is 102
class Box(4,5,6) volume is 120
```

输出表明我们总是可以通过调用成员函数 volume()来获取多态性。重载了函数

operator<<()来展示正确的类类型，因此 write()函数调用也是多态的。很明显，当我们需要使用具有层次关系的不同类型的元素时，智能指针可以提供很多优点。通过在容器中保存基类的智能指针，可以自动获得多态行为，也可以自动释放自由存储区的内存。这可以应用到任何类型的容器或容器适配器上。

3.6.5　对指针序列应用算法

算法可以处理指定的一段数据。当这段数据和保存指针的容器相关时，数据段的迭代器指向指针。因此，需要定义一个算法考虑到这一点的函数对象。尽管到目前为止，我们只看到了几个算法，但可以先通过几个示例看看效果。

定义在 numeric 头文件中的 accumulate()算法可以通过使用默认的 operator+()来计算一个数据段的和。第二个版本的 accumulate()有 4 个参数——前两个参数是序列的开始和结束迭代器，后两个参数分别是这个运算的初始值和一个用来进行二元运算的函数对象。可以使用此版本来替换默认操作，而且如果这段元素包含指针，为了获取有效的结果，必须使用这个版本的函数。下面展示如何用 accumulate()来连接 vector 中的 shared_ptr<string>对象所指向的 string 对象：

```
using word_ptr = std::shared_ptr<std::string>;
std::vector<word_ptr> words {std::make_shared<string>("one"),
   std::make_shared<string>("two"),std::make_shared<string>("three"),
   std::make_shared<string>("four")};
auto str = std::accumulate(std::begin(words), std::end(words), string {""},
                           [] (const string& s, const word_ptr& pw)->string
                              {return s + *pw + " "; });
```

accumulate()的第 4 个参数是一个二元操作，并且也是一个 lambda 表达式。accumulate()将它接收的第 3 个参数作为第一个参数传给二元函数，将迭代器解引用的结果作为第二个参数传给 lambda 函数。因此 lambda 的第一个参数必须和 accumulate()的第三个参数的类型相同，第二个参数必须和迭代器指向的元素类型相同。str 保存的结果是"one two three four"。

对于元素是指针的容器，需要考虑在它的成员函数中使用断言的情况。list 和 forward_list 容器有成员函数 remove_if()，下面是一个示例：

```
std::list<word_ptr> wrds {std::make_shared<string>("one"),
   std::make_shared<string>("two"),std::make_shared<string>("three"),
   std::make_shared<string>("four")};
wrds. remove_if([](const word_ptr& pw){ return (*pw)[0] == 't'; });
```

当字符串中的第一个字符是't'时，传入 remove_if()函数的 lambda 表达式会返回 true。这会从链表移除"two"和"three"。lambda 表达式的参数是一个指向 string 对象的智能指针，因此需要对它解引用来访问字符串。

3.7 本章小结

容器适配器可以为标准序列容器提供专门的接口；只能通过类适配器提供的接口来访问容器中的元素。当需要使用适配器提供的功能时，可以选择一个合适的适配器，它们的接口都很简单。可以用 algorithm 头文件中的函数模板在序列容器中创建堆，它提供了和优先级队列适配器相同的功能，但可以更灵活地访问容器中的元素。本章的要点如下：

- stack<T>容器适配器模板实现了一个压入栈，它提供先入后出(LIFO)的检索机制。可以用 vector<T>、deque<T>、list<T>作为 stack<T>内部存放元素的容器。
- queue<T>容器适配器模板实现了一个队列，它提供先入先出(FIFO)的检索机制。可以将 deque<T>、list<T>用作 queue<T>内部存放元素的容器。
- priority_queue<T>容器适配器模板实现了一个优先级队列，每次从它获取到的都是优先级最高的元素。vector<T>和 deque<T>可以作为 priority_queue<T>内部存放元素的容器。
- 堆是一个二叉树，它的部分节点和全部节点有相同的顺序。对于大顶堆，父节点会大于或等于它的子节点；对于小顶堆，父节点会小于或等于它的子节点。
- make_heap()函数模板可以在随机访问迭代器指定的范围内创建一个元素堆。默认创建的是大顶堆，但是也可以指定一个决定堆排序的函数对象。在添加元素或移除第一个元素后，可以使用函数 push_heap()和 pop_heap()来保持堆序。
- 可以在容器中保存指针。使用智能指针可以保证在不需要自由存储区的对象时，可以自动删除它们。
- 对于保存指针的容器来说，我们需要为它们提供一些算法需要的函数对象，这些函数通常用来进行比较或进行其他的一些运算。

练习

1. 写一个程序，它可以从键盘读入任意个数的单词，然后把它们保存到 deque<string>容器中。然后再把容器中的单词复制到 list<string>容器中，并将列表中的内容排成升序，最后输出排序后的结果。

2. 写一个程序，它用一个 stack<T>容器适配器实例将从键盘输入的一行文字反向。程序应该输出反向后的结果，并判断和原始字符串是否回文(回文字符串是一个正向和反向都相同的字符串——如果忽略空格和标点的话，例如 Are we not drawn onward to a new era?")。

3. 写一个程序，它可以用一个 priority_queue 容器适配器实例将从键盘输入的任何个数的单词降序排列。

4. 重复练习 1，但是单词要在自由存储区生成，然后在容器中保存它们的智能指针。

5. 重复练习 3，但是要在 priority_queue 容器适配器中保存单词的智能指针。

6. 写一个程序，它以字母降序的方式输出从键盘输入的任意个数的单词。需要在 vector 容器中保存单词的智能指针。

第 4 章

map 容器

可以通过元素在序列容器中的位置来检索元素。例如，可以通过位置访问 deque 容器的第一个或最后一个元素，也可以用元素在 vector 容器中的位置索引来访问元素；但 map 容器的工作原理和它们完全不同，这点稍后会在本章进行解释。本章将介绍以下内容:

- 关联容器的概念
- map 容器的概念及它的组织方式
- map 容器的类型及其功能
- map 容器提供的函数
- pair 的概念及其用法
- tuple 的概念及其使用方法

4.1 map 容器介绍

序列容器是管理数据的宝贵工具，但对大多数应用程序而言，序列容器不提供方便的数据访问机制。举个简单的示例，当我们用它处理姓名和地址时，在这种场景下，序列容器可能并不能如我们所愿。一种典型的方法是通过名称来寻找地址。如果记录保存在序列容器中，就只能通过搜索得到这些数据。相比而言，map 容器提供了一种更有效的存储和访问数据的方法。

map 容器是关联容器的一种。在关联容器中，对象的位置取决于和它关联的键的值。键可以是基本类型，也可以是类类型。字符串经常被用来作为键，如果想要保存姓名和地址的记录，就可以这么使用。名称通常可能是一个或多个字符串。关联容器中的对象位置的确定取决于容器中的键的类型，而且对于特定容器类型的内部组织方式，不同的 STL 有不同的实现。

map 容器有 4 种，每一种都是由类模板定义的。所有类型的 map 容器保存的都是键值对类型的元素。map 容器的元素是 pair<const K,T>类型的对象，这种对象封装了一个 T 类型的对象和一个与其关联的 K 类型的键。pair 元素中的键是 const，因为修改键会扰乱容器

中元素的顺序。每种 map 容器的模板都有不同的特性：

- map<K,T>容器保存的是 pair<const K,T>类型的元素。pair<const K,T>封装了一对键/对象，键的类型是 K，对象的类型是 T。每个键都是唯一的，所以不允许有重复的键；但可以保存重复的对象，只要它们的键不同。map 容器中的元素都是有序的，元素在容器内的顺序是通过比较键确定的。默认使用 less<K>对象比较。
- multimap<K,T>容器和 map<K,T>容器类似，也会对元素排序。它的键必须是可比较的，元素的顺序是通过比较键确定的。和 map<K,T>不同的是，multimap<K,T>允许使用重复的键。因此，一个 multimap 容器可以保存多个具有相同键值的<const K,T>元素。
- unordered_map<K,T>中 pair< const K，T>元素的顺序并不是直接由键值确定的，而是由键值的哈希值决定的。哈希值是由一个叫作哈希的过程生成的整数，本章后面会解释这一点。unordered_map<K,T>不允许有重复的键。
- unordered_multimap<K,T>也可以通过键值生成的哈希值来确定对象的位置，但它允许有重复的键。

map 和 mutilmap 容器的模板定义在 map 头文件中，unordered_map 和 unordered_multimap 容器的模板定义在 unordered_map 头文件中。可以通过 map 模板类型名的前缀来识别容器的特性。

- multi 前缀表明键不必唯一，但如果没有这个前缀，键必须唯一。
- unordered_ prefix 前缀表明容器中元素的位置是通过其键值所产生的哈希值来决定的，而不是通过比较键值决定的。如果没有该前缀，那么元素的位置就由比较键值决定。

接下来让我们先学习 map 容器。

4.2 map 容器的用法

map<K,T>类模板定义在 map 文件头中，它定义了一个保存 T 类型对象的 map，每个 T 类型的对象都有一个关联的 K 类型的键。容器内对象的位置是通过比较键决定的。可以用适当的键值从 map 容器中检索对象。图 4-1 展示了一个用名称作为键的 map<K,T>容器，对象是整数值，用来表示年龄。

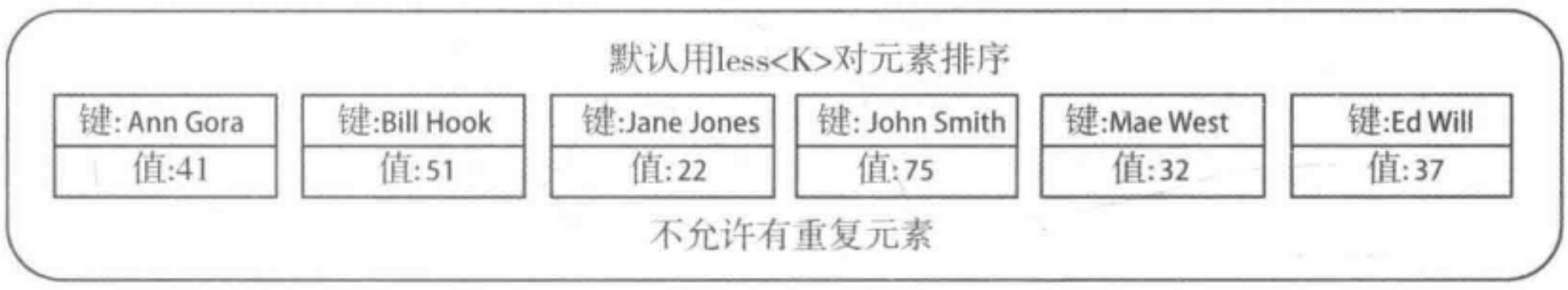


图 4-1　map<K,T>容器的概念展示图

图 4-1 表示的是 map<Name,size_t>类型的容器，其中的 Name 类可以这样定义：

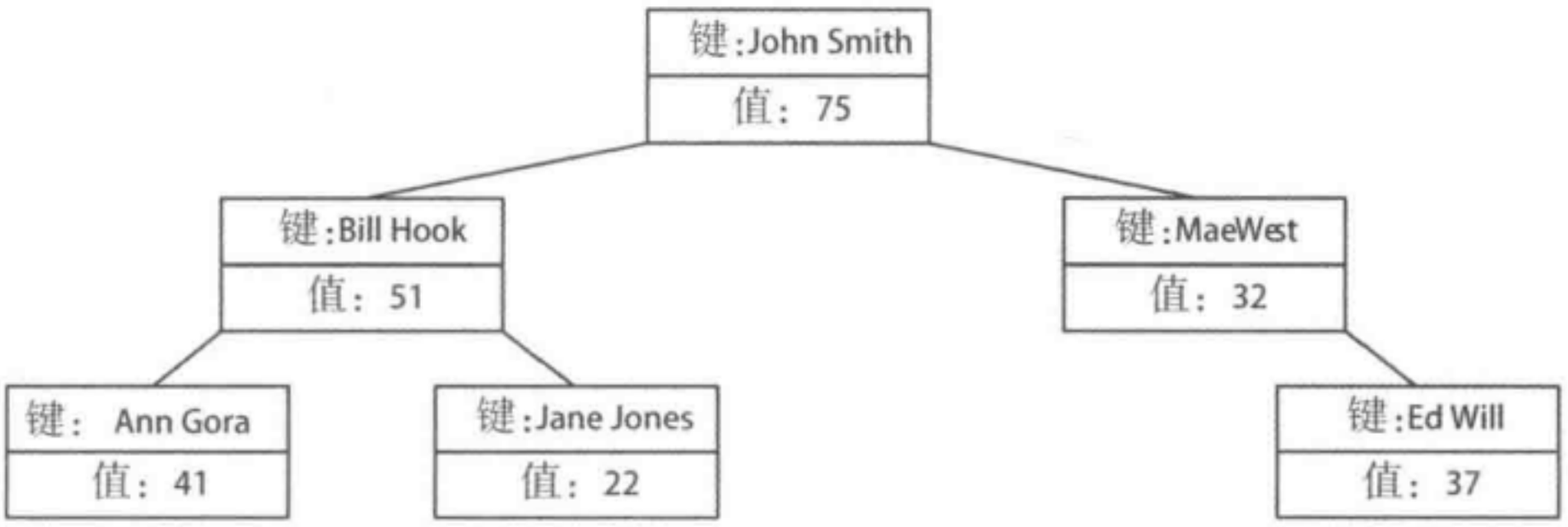
```
class Name
{
private:
  std::string firstname{};
  std::string secondname{};
public:
  Name(std::string first, std::string second) : firstname{first},
    secondname{second}{};
  Name()=default;

  bool operator<(const Name& name)
  { return secondname < name.secondname ||
    ((secondname == name.secondname) && (firstname < name.firstname)); }
};
```

为了可以在必要时生成默认的元素，容器保存的对象通常需要定义一个默认的构造函数。当两个 Name 对象的 secondname 不同时，成员函数 operator<()通过比较 secondname 来确定对象的顺序。如果 secondname 相等，比较结果就由 firstname 决定。string 类定义了 operator<()，因而可以用默认的 less<string>来比较。

不要因为 map 使用 less<K>对元素排序就被误导，这些元素并没有被组织成一个简单的有序序列，STL map 容器对元素的组织方式并没有具体要求，但元素一般都会保存在一个平衡二叉树中。容器中的元素被组织成一个平衡二叉树，因而树的高度——根节点和叶节点之间的高度是最低的。如果每个节点的左子树和右子树的高度差不超过 1，那么可以说这棵二叉树就是平衡的。图 4-2 展示了图 4-1 所表示的 map 容器可能的平衡二叉树。

图 4-2 所示的树有 3 层，所以从根节点开始，找到任意的元素最多需要 3 步。这里选择的根节点可以使树的高度最小，而且对于每个父节点来说，它的键值大于它的左子节点，但小于它的右子节点。为了保持二叉树的平衡，当添加一个新的元素时，可能会导致根节点发生改变。所以显然，在添加新元素时，为了保持树的平衡，会产生一些额外的开销。作为回报，容器中的元素越多，相对于线性排列和非平衡树，平衡树组织元素的效率也越高。从包含 n 个元素的平衡二叉树中检索一个随机元素所需的时间为 $O(\log_2 n)$，从序列中检索元素所需的时间为 O(n)。



map容器的平衡二叉树

图 4-2　map 容器的内部组织图

■ 注意：O(n)计算时间随着参数的增加而增加。O 被认为是有序的，O(n)表明线性执行时间在以 n 增加。$O(\log_2 n)$计算时间远没有 n 增加得快，因为它是以 $\log_2 n$ 计算的。

4.2.1 创建 map 容器

map 类模板有 4 个类型参数,但一般只需要指定前两个模板参数的值。第 1 个是键的类型，第 2 个是所保存对象的类型，第 3 个和第 4 个模板参数分别定义了用来比较键的函数对象的类型以及为 map 分配内存的对象的类型。最后两个参数有默认值。在本节稍后部分会展示如何定义不同类型的比较键的函数对象，但不会定义可替代的分配器类型。

map<>容器类的默认构造函数会创建一个空的 map 容器。例如，可以创建一个这样的容器，size_t 类型的值表示年龄，作为它保存的值，string 类型的值表示名称，作为它的键：

```
std::map<std::string, size_t> people;
```

第 1 个模板类型参数指定键的类型是字符串，第 2 个模板类型参数指定值的类型为 size_t。当然，这里的模板类型参数可以是任何类型，唯一的要求是键必须可以用 less<K>比较，或用自己指定的另一个函数对象来替代。

map<K,T>中的每个元素都是同时封装了对象及其键的 pair<const K,T>类型对象，这里不能修改 const K。pair<T1,T2>类的模板定义在 utility 头文件中，它被包含在 map 头文件中。因此 people 容器中的元素是 pair<const string, size_t>类型的。pair<T1,T2>这种模板类型并不是专门在这种情况下使用的。必要时可以用它将两个不同类型的对象组装成一个对象。本章稍后将讲解更多这方面的内容。

我们可以用初始化列表来指定 map 的初始值，但因为 map 中包含的是 pair< const K,T >类型的元素，所以初始化列表中的值也必须是这种类型。下面展示了如何为 people 容器设置初始值：

```
std::map<std::string, size_t> people{ {"Ann", 25}, {"Bill", 46}, {"Jack",
32}, {"Jill", 32} };
```

初始化列表中的值是通过将每个嵌套花括号中的两个值传递给构造函数产生的，因此列表会包含 4 个 pair<const string,size_t >对象。

utility 头文件中定义了 make_pair < T1,T2 >()函数模板，它提供了一种组合 T1 和 T2 类型对象的简单方法。因此，可以按如下方式创建 pair 对象来初始化 map：

```
std::map<std::string, size_t> people{ std::make_pair("Ann", 25),
   std::make_pair("Bill", 46),std::make_pair("Jack", 32),
   std::make_pair("Jill", 32) };
```

make_pair<T1,T2>()函数模板从函数参数中推断出类型参数值，因而由参数列表中调用 make_pair<>()返回的是<char const*,int>类型的对象。因为这些对象都是 map 容器 people 的初始值，所以这些 pair 对象会被转换成 map 中元素的类型，即 pair<const string, size_t>。

pair<T1，T2>的公共成员变量 first 和 second 分别保存了存储 T1 和 T2 类型的对象。只要原始 pair 对象的成员变量 first 和 second 可以隐式转换为与目标 pair 对象成员变量相同类型的变量，pair<T1,T2>模板的构造函数就可以提供这种类型的隐式转换。

map< K,T >模板定义了移动和复制构造函数，所以可以复制现有的容器。例如：

```
std::map<std::string, size_t> personnel {people}; // Duplicate people map
```

map 容器 personnel 包含 people 元素的副本。

可以用另一个容器的一段元素来创建一个 map，用开始和结束迭代器以通常的方式指定元素。显然，迭代器指向的 pair 元素的类型必须和容器兼容。这里有一个示例：

```
std::map<std::string, size_t> personnel {std::begin(people),
   std::end(people)};
```

这样就生成了personnel，并且用 people 容器的迭代器指定的元素对它进行了初始化。map 容器提供了双向迭代器，这样就可以通过自增或自减访问元素。map 容器还提供了反向迭代器，所以可以从最后一个元素遍历到第一个元素。personnel 容器包含的元素和 people 完全相同。当然，也可以用另一个容器的元素子集来创建容器：

```
std::map<std::string, size_t> personnel {++std::begin(people),
    std::end(people)};
```

4.2.2　map 元素的插入

map<K,T>容器的成员函数 insert()有多个版本，它们可以在 map 中插入一个或多个 pair<const K,T>对象。只能插入 map 中不存在的元素。下面这个代码片段展示了如何插入单个元素：

```
std::map<std::string, size_t> people {std::make_pair("Ann", 25),
   std::make_pair("Bill", 46),std::make_pair("Jack", 32),
   std::make_pair("Jill", 32)};
auto pr = std::make_pair("Fred", 22);       // Create a pair element...
auto ret_pr = people.insert(pr);            // ..and insert it
std::cout << ret_pr.first->first << " " << ret_pr.first->second
     << " " << std::boolalpha << ret_pr.second << " \n"; // Fred 22 true
```

第一条语句生成了一个 map 容器，并用初始化列表中的 4 个值对它进行了初始化；在这种情况下，这些值会被隐式转换为要求的类型。第二条语句生成了另一个被插入的 pair 对象 pr。pr 对象的类型是 pair<const char*,int>，因为 make_pair<>()函数模板的类型参数是从参数类型推断出来的；但是在 insert()操作中，这个对象会被隐式转换为容器元素类型。当然，如果不想依靠隐式转换，可以生成所要求类型的 pair 对象：

```
auto pr = std::make_pair<std::string, size_t>(std::string {"Fred"}, 22);
```

make_pair< >()模板的显式类型参数决定了返回的 pair 对象的类型。可以把文字字符串作为第一个参数，然后通过隐式转换创建键需要的字符串对象。可以省略 make_pair< >()

的模板类型参数，让编译器去推断它们。假设像下面这样声明：

```
auto pr = std::make_pair("Fred", 22);      // pair<const char*, int>
```

这里会返回和所要求类型不同的 pair 对象。当允许编译器推断模板参数类型时，make_pair()的参数可以准确地确定模板参数的类型。第一个参数是一个 const char*类型的字符串，第二个参数是 int 类型。尽管已经说明了元素的类型，但在这种情况下，并没有多大的用处，因为在插入一个新元素时，pair 对象可以被隐式转换为容器所需类型。当 make_pair()的参数的类型不能隐式转换成容器的键和对象的类型时，就需要注意了。

成员函数 insert()会返回一个 pair<iterator,bool>对象。对象的成员 first 是一个迭代器，它要么指向插入元素，要么指向阻止插入的元素。如果 map 中已经保存了一个和这个键相同的对象，就会出现后面这种情况。这个对象的成员变量 second(布尔型)是返回对象，如果插入成功，返回值为 true，否则为 false。输出语句像我们看到的那样，访问插入 pair 的成员变量 first 的表达式是 ret_pr.first->first。ret_pr 的成员变量 first 是一个指向 pair 对象的迭代器，所以可以使用->操作符来访问它的成员变量 first。输出展示了插入的元素。可以通过下面这个循环进行验证:

```
for(const auto& p : people)
std::cout << std::setw(10) << std::left << p.first << " " << p.second << " \n";
```

循环变量 p 通过引用依次访问 map 容器 people 中的每个元素。输出如下：

```
Ann 25
Bill 46
Fred 22
Jack 32
Jill 32
```

元素是以键的升序排列的，因为 map 中默认使用 less<string>函数对象对它们进行排序。通过执行下面这两条语句，可以看出元素插入后的效果：

```
ret_pr = people.insert(std::make_pair("Bill", 48));
std::cout << ret_pr.first->first << " " << ret_pr.first->second
          << " " << std::boolalpha << ret_pr.second << "\n"; // Bill 46 false
```

程序会输出如注释所示的内容。insert()返回了一个 pair 对象 ret_pr，它的成员变量 first 指向 map 中已有的和键匹配的元素，成员变量 second 为 false，表明元素没有插入成功。

当元素已经存在时，如果想将键值“Bill”对应的年龄值改为 48，可以像下面这样使用 insert()返回的 pair 对象来做到这一点：

```
if(!ret_pr.second) // If the element is there...
ret_pr.first->second = 48; // ... change the age
```

当键已经存在于 map 容器中时，ret_pr 的成员变量 second 为 false，所以这段代码会将 map 中这个元素的成员变量 second 的值设为 48。

可以用 pair 构造函数生成的对象作为 insert()的参数：

```
ret_pr = people.insert(std::pair<const std::string, size_t> {"Bill", 48});
```

这里会调用一个具有右值引用参数的 insert()版本，所以假如元素不在容器中，那么它会被移到容器中。

也可以提供一个提示符来指出元素插入的位置。提示符是迭代器的形式，它指向容器中的一个现有元素，通常从提示符指示的位置开始查找新元素的插入位置。好的提示符可以提高插入操作的速度，反之亦然。例如：

```
auto ret_pr = people.insert(std::make_pair("Jim", 48));
people.insert(ret_pr.first, std::make_pair("Ian", 38));
```

第一条语句插入了一个元素，并像前面那样返回了一个对象。pair 对象的成员变量 first 是一个指向被插入的元素或容器中与插入元素有相同键的元素的迭代器。下一个 insert()函数的第一个参数和上面的提示符有关，所以这里就是插入元素的地方。insert()的第二个参数指定的新元素会被插入到提示符的前面，并尽可能地靠近它。如果提示符不能以这种方式使用，那么将忽略它。同样地，如果被插入的元素已经在 map 中，会导致元素插入失败。带提示符的insert()调用会返回一个指向被插入元素或容器中阻止此插入操作的元素的迭代器，因此可以使用返回值来确定插入是否成功。当确定元素不存在时，可以只提供一个插入符，这是一个好的想法。如果不那么确定，而且仍然想使用插入符，map 中的 count()函数对我们有一些帮助。它会返回 map 中指定键对应元素的数目，这个数目可能是 0 或 1。因此代码可以这样写：

```
if(!people.count("Ian"))
  people.insert(ret_pr.first, std::make_pair("Ian", 38));
```

只有当 count()函数返回 0 时，insert()才会被调用，这说明“Iran”键不在 map 中。当然，在不用提示插入元素时，需要做一次这样的检查，但 insert()的返回值不管怎样都能告诉我们插入结果。

也可以将外部源中的一段元素插入 map 中，这些元素不必来自另一个 map 容器，但必须和被插入容器中的元素是同类型。这里有一些示例：

```
std::map<std::string, size_t> crowd {{"May", 55}, {"Pat", 66}, {"Al", 22},
{"Ben", 44}};
auto iter = std::begin(people);
std::advance(iter, 4);                          // begin iterator+ 4
crowd.insert(++std::begin(people), iter); // Insert 2nd, 3rd, and 4th
                                                // elements from people
```

这里生成了一个新的 map 容器 crowd，它有 4 个初始元素。iter 被初始化为 people 的开始迭代器。map 容器的迭代器是双向的，这样就可以对它们进行自增或自减，但是不能加上或减去一个值。这里使用了一个在第 1 章用过的 advance()函数模板的实例来对 iter 加 4，所以它将会指向第 5 个元素，它在下一行被当作 crowd 的成员函数 insert()的参数，用来作为指定元素段的结束迭代器。map 容器 people 的开始迭代器加 1，然后用它作为插入元素段的开始迭代器，所以会从 crowd 的第 2 个元素开始插入 3 个元素。

下面是一个接受初始化列表作为参数的 insert()版本：

```
crowd.insert({{"Bert", 44}, {"Ellen", 99}});
```

这里会将初始化列表中的两个元素插入到 map 容器 crowd 中。参数表达式生成的 initializer_list< >对象是 initializer_list<const string,size_t>类型，因为编译器知道这时 insert()函数的参数是这种类型。当然，也可以单独创建一个初始化列表，然后将它作为参数传递给 insert()函数：

```
std::initializer_list<std::pair<const std::string, size_t>> init {{"Bert",
44}, {"Ellen", 99}};
crowd.insert(init);
```

initializer-list 模板的第一个类型参数必须为 const 类型。initializer_list <string,size_t>无法隐式转换为 initializer_list<const string,size_t>，所以前者类型的对象不能作为 insert()函数的参数。

下面你会看到这些操作的一个完整示例。此处会定义一个有点不同的对象。Name 类型代表人名，这个类定义的头文件的内容如下：

```
// Name.h for Ex4_01
// Defines a person's name
#ifndef NAME_H
#define NAME_H
#include <string>                          // For string class
#include <ostream>                         // For output streams
#include <istream>                         // For input streams

class Name
{
private:
 std::string first {};
 std::string second {};

public:
 Name(const std::string& name1, const std::string& name2) :
                                          first (name1), second (name2) {}
 Name() = default;
 // Less-than operator
 bool operator<(const Name& name) const
 {
  return second < name.second || (second == name.second && first < name.first);
 }
 friend std::istream& operator>>(std::istream& in, Name& name);
 friend std::ostream& operator<<(std::ostream& out, const Name& box);
};

// Extraction operator overload
inline std::istream& operator>>(std::istream& in, Name& name)
{
```

```
 in >> name.first >> name.second;
 return in;
}

// Insertion operator overload
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{
 out << name.first + " " + name.second;
 return out;
}
#endif
```

这个类非常简单，只有两个string类型的私有成员变量first和second。这个构造函数可以接受string类型的参数或字符串常量参数。为了可以用这种对象作为map容器的键，必须为这个类定义operator<()。为了便于对象的输入输出，也需要为流定义插入和提取操作。

map容器中的元素是std::pair<const Name, size_t>类型，我们可以用下面定义的别名来简化代码：

```
using Entry = std::pair<const Name, size_t>;
```

当容器是map<Name,size_t>类型时，我们只能用Entry作为容器元素的类型。为了便于map元素的输出，我们可以把别名放到一个函数的定义中：

```
Entry get_entry()
{
  std::cout << "Enter first and second names followed by the age: ";
  Name name {};
  size_t age {};
  std::cin >> name >> age;
  return make_pair(name, age);
}
```

从cin先后读入了一个Name对象和一个年龄值，并用它们生成了一个pair对象。从输入中读取name激发了定义在Name.h中的istream对象的重载函数operator>>()，同样也支持流对象对Name对象的读入。

可以输出容器元素的辅助函数也是很有用的：

```
void list_entries(const map<Name, size_t>& people)
{
  for(auto& entry : people)
  {
   std::cout << std::left << std::setw(30) << entry.first
             << std::right << std::setw(4) << entry.second << std::endl;
  }
}
```

这里只用了基于范围的for循环来对元素进行遍历。循环变量entry依次引用map的每

个元素。每一个 map 元素都是一个 pair 对象，它的成员变量 first 是 Name 类型的对象，成员变量 second 是 size_t 类型的值。

包含 main()函数的源文件中的内容如下：

```
// Ex4_01.cpp
// Storing names and ages
#include <iostream>                          // For standard streams
#include <iomanip>                           // For stream manipulators
#include <string>                            // For string class
#include <map>                               // For map container class
#include <utility>                           // For pair<> & make_pair<>()
#include <cctype>                            // For toupper()
#include "Name.h"

using std::string;
using Entry = std::pair<const Name, size_t>;
using std::make_pair;
using std::map;

// Definition of get_entry() here...

// Definition of list_entries() here...

int main()
{
  map<Name, size_t> people { {{"Ann", "Dante"}, 25}, {{"Bill", "Hook"}, 46},
                             {{"Jim", "Jams"}, 32}, {{"Mark", "Time"}, 32} };
  std::cout << "\nThe initial contents of the map is:\n";
  list_entries(people);

  char answer {'Y'};
  std::cout << "\nEnter a Name and age entry.\n";
  while(std::toupper(answer) == 'Y')
  {
    Entry entry {get_entry()};
    auto pr = people.insert(entry);
    if(!pr.second)
    { // It's there already - check whether we should update
      std::cout << "Key \"" << pr.first->first
                << "\" already present. Do you want to update the age (Y or N)? ";
      std::cin >> answer;
      if(std::toupper(answer) == 'Y')
        pr.first->second = entry.second;
    }
    // Check whether there are more to be entered
    std::cout << "Do you want to enter another entry(Y or N)? ";
    std::cin >> answer;
  }

  std::cout << "\nThe map now contains the following entries:\n";
  list_entries(people);
}
```

定义一些额外的别名可以减少代码冗余。可以用 using std::namespace 来完全消除对 std 名称限定的需要，但不赞成这么做，因为 std 中的所有命名都被有效导入到当前作用域内，这违背了定义命名空间的初衷。

map 容器的定义中有一些初始值，它们是初始化列表中的元素。这里只是为了说明如何在这种情况下使用嵌套花括号。在定义元素的每个初始化列表中，花括号中的 Name 对象都是初始化列表，每个初始化元素的花括号中都是一个 Name 对象和一个年龄值。最外面的一对花括号包括了所有元素的初始值。

辅助函数 list_entries()用来显示容器的初始状态。从 for 循环中读入了更多的 entry 对象。循环由 answer 的值控制，如果一开始它的值就是'Y'，那么最少执行一次循环，最少从键盘输入一个元素。entry 对象的类型是 Entry，它也是容器元素的类型。辅助函数 get_entry()的返回对象被作为 entry 的初始值。然后将它作为 insert()的参数插入到容器中。这个函数返回的 pair 对象有一个成员变量 first，它指向容器中和 entry 的键匹配的元素。如果在插入之前，容器中就存在这个元素，那么它指向的是原始容器中的元素。如果键已经存在于容器中，插入不会成功，并且 pr 的成员变量 second 为 false。pr.first 是容器中元素的一个迭代器，因此 pr.first->second 可以访问与键关联的对象，如果用户确认需要更新，pr.first->second 的值会变为 entry.second。循环中的最后一个操作可以决定是否输入更多的 entry。在不需要输入更多的 entry 时，循环结束，用 list_entries()函数输出这个容器最终的内容。

下面是这个示例的一些输出结果：

```
The initial contents of the map is:
Ann Dante                25
Bill Hook                46
Jim Jams                 32
Mark Time                32
Enter a Name and age entry.
Enter first and second names followed by the age: Emma Nate 42
Do you want to enter another entry(Y or N)? y
Enter first and second names followed by the age: Emma Nate 43
Key "Emma Nate " already present. Do you want to update the age (Y or N)? Y
Do you want to enter another entry(Y or N)? y
Enter first and second names followed by the age: Eamonn Target 56
Do you want to enter another entry(Y or N)? N
The map now contains the following entries:
Ann Dante                25
Bill Hook                46
Jim Jams                 32
Emma Nate                43
Eamonn Target            56
Mark Time                32
```

这些元素是以键的升序排列的，因为容器中的元素是使用 less<Name>进行排序的。Name::operator<()先比较姓，在姓相同时才比较名。这导致返回的姓名是正常的顺序。

4.2.3 在 map 中构造元素

map 容器的成员函数 emplace()可以在适当的位置直接构造新元素，从而避免复制和移动操作。它的参数通常是构造元素，也就是 pair<const K,T>对象所需要的。只有当容器中现有元素的键与这个元素的键不同时，才会构造这个元素。下面是一个示例：

```
std::map<Name, size_t> people;
auto pr = people.emplace(Name{"Dan", "Druff"}, 77);
```

这个 map 包含的是 Name 类型的键，它是一个定义在 Ex4_01 中的类型。对象的类型是 size_t，所以 map 包含的是 pair<const Name,size_t>类型的元素。emplace()的第一个参数是一个 Name 对象，它同时也是元素的键，第二个参数是 size_t 类型的值，函数会用这些参数调用 pair<const Name,size_t>的构造函数以在适当的位置生成元素。如果用 emplace()的参数构造 pair 对象，那么会调用 pair<const Name,size_t>对象的移动构造函数。

成员函数emplace()和 insert()返回的 pair 对象提供的指示相同。pair 的成员变量 first 是一个指向插入元素或阻止插入的元素的迭代器；成员变量 second 是个布尔值，如果元素插入成功，second 就为 true。

map 的成员函数 emplace_hint()和emplace()生成元素的方式在本质上是一样的，除了必须为前者提供一个指示元素生成位置的迭代器，作为 emplace_hint()的第一个参数。例如：

```
std::map<Name, size_t> people;
auto pr = people.emplace(Name{"Dan", "Druff"}, 77);
auto iter = people.emplace_hint(pr.first, Name{"Cal", "Cutta"}, 62);
```

这里，emplace_hint()调用使用一个迭代器作为指示符，指向先前emplace()调用返回的 pair 对象。如果容器使用这个提示符，那么新元素会在这个指示符表示的位置之前生成，并尽可能靠近这个位置。提示符后面的参数用来构造新元素。需要注意的是，它和 emplace()的返回值是不一样的。emplace_hint()的返回值不是一个 pair 对象，如果新元素被插入，它返回的是指向新元素的迭代器；如果没有插入，返回的是和这个键匹配的现有元素的迭代器，拥有相同的 key 值，如果不是现有元素的话。没有提示可以直接判断是否生成了新元素。然而，并不是一点办法都没有——唯一的方法是，用 size()成员函数来获取 map 中对应元素的数量来检查 map 元素增加的数量。例如：

```
auto pr = people.emplace(Name{"Dan", "Druff"}, 77);
auto count = people.size();
auto iter = people.emplace_hint(pr.first, Name{"Cal", "Cutta"}, 62);
if(count < people.size()) std::cout << "Success!\n";
```

只有当调用 emplace_hint()后，people 中的元素增加时，才会显示这些信息。

4.2.4 访问 map 中的元素

我们已经知道，可以获取 map 容器的开始和结束迭代器以及反向迭代器，它们都可以

访问容器中的所有元素。map的成员函数 at()返回的是参数键对应的对象。如果这个键不存在，就会抛出 out_of_range 异常。下面展示如何使用这个函数：

```
Name key;
try
{
  key = Name {"Dan", "Druff"};
  auto value = people.at(key);
  std::cout << key << " is aged " << value << std::endl;
  key = Name {"Don", "Druff"};
  value = people.at(key);
  std::cout << key << " is aged " << value << std::endl;
}
catch(const std::out_of_range& e)
{
  std::cerr << e.what() << '\n'
            << key << " was not found." << std::endl;
}
```

需要在try代码块中调用map的成员函数at()——抛出的任何未捕获的异常都会导致程序的终止。这段代码获取了 people 容器中的两个对象，它们分别与两个 Name 键关联。如果 map 容器中的内容由执行的前一节中的代码段决定，输出效果如下：

```
Dan Druff is aged 77
invalid map<K, T> key
Don Druff was not found.
```

Try 代码块中第一次调用 at()函数成功，结果会在首行输出。第二次调用失败，抛出了一个可捕获的 out_of_range 异常，捕获结果在后面两行输出。异常对象的成员函数 what()是一个返回了描述异常产生原因的字符串。当 catch 代码块中的代码执行后，try 代码块中的所有变量会被销毁，因此不再可以访问。变量 key 是在 try 代码块之前定义的，因此仍然可以在 catch 代码块中访问。

map 容器提供了以键为参数的下标运算符，它可以返回一个和键所关联对象的引用。下面是一个示例：

```
auto age = people[Name {"Dan", "Druff"}];
```

这里获取到一个和 Name 键关联的 size_t 类型的值。注意，下标运算的使用并不是简单的检索机制。如果键不存在，元素默认的构造函数会用键和键所关联的对象生成一个新元素，如果键关联的对象是基本数据类型，它的值为 0。例如：

```
auto value = people[Name {"Ned", "Kelly"}]; // Creates a new element if the
                                            key is not there
```

因为容器中不存在这个键，所以用它生成了新元素。关联对象的值是 0，并会返回这个值。可以用下标运算符来更新 map 中的元素，如果元素不在 map 中，也可以用它插入元

素。下标运算主要用在左赋值上，用来修改已存在的元素：

```
people[Name {"Ned", "Kelly"}] = 39; // Sets the value associated with the
                                    // key to 39
```

让我们在新的示例中，用一种不同以往的方式使用 map，并且充分利用下标运算符。可以用 map 容器来确定每个字符在文本中出现的频率。确定词频是非常有用的——例如，可以用它对文档进行分类。下面展示了如何在任意文本序列中统计每个单词的出现次数：

```
// Ex4_02.cpp
// Determining word frequency
#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
#include <string>                      // For string class
#include <sstream>                     // For istringstream
#include <algorithm>                   // For replace_if() & for_each()
#include <map>                         // For map container
#include <cctype>                      // For isalpha()

using std::string;

int main()
{
  std::cout << "Enter some text and enter * to end:\n";
  string text_in {};
  std::getline(std::cin, text_in, '*');

  // Replace non-alphabetic characters by a space
  std::replace_if(std::begin(text_in), std::end(text_in),
                  [](const char& ch){ return !isalpha(ch); }, ' ');

  std::istringstream text(text_in); // Text input string as a stream
  std::istream_iterator<string> begin(text); // Stream iterator
  std::istream_iterator<string> end; // End stream iterator

  std::map<string, size_t> words; // Map to store words & word counts
  size_t max_len {}; // Maximum word length

  // Get the words, store in the map, and find maximum length
  std::for_each(begin, end, [&max_len, &words](const string& word)
                            { words[word]++;
                              max_len = std::max(max_len, word.length());
                            });
  // Ouput the words and their counts
  size_t per_line {4}, count {};
  for(const auto& w : words)
  {

    std::cout << std::left << std::setw(max_len + 1) << w.first
      << std::setw(3) << std::right << w.second << " ";

    if(++count % per_line == 0) std::cout << std::endl;
```

```
    }
    std::cout << std::endl;
}
```

从标准输入流读取到 text_in 中的文本是通过函数 getline()得到的字符串。replace_if()算法用空格替换了输入中的所有非字母字符。replace_if()函数的前两个参数是定义元素范围的迭代器，这里的元素范围就是输入字符串的字符。下一个参数是一个函数对象，当元素需要被替换时，它返回 true；这里是一个 lambda 表达式。最后一个参数是用来替换的元素——在这个示例中这个元素是空格。这个函数会替换掉所有的标点，所以最后每个元素都是用空格分隔的。

我们用 text_in 生成一个 istringstream 对象 text。istringstream 对象允许对它封装的字符串进行流输入操作，因此可以把它当作一个流。这也包括从 text 获得流迭代器的能力，然后可以在 for_each()中用它们提取单个单词。输入流的迭代器会陆续指向每个输入的字符串。这里输入的单词是连续的，因此开始和结束迭代器指定的范围是 text 中的所有单词。for_each()会将第 3 个参数指定的函数对象运用到前两个参数所指定范围内的元素上。函数对象必须以迭代器指向对象类型的引用作为参数，所以这里参数是 const string&。lambda 以引用的方式捕获变量 max_len 和 words，所以它们都可以修改。lambda 通过将每个单词作为下标来将它们以键的方式保存在容器中，并增加单词关联的值。如果单词不在容器中，会以这个单词为键(值为 1)来生成一个新的元素。如果单词先前就被添加到容器中，就自动增加值。因此与每个单词的关联值就是它在文本中累计出现的次数。为了保存最长字符串的长度，lambda 表达式也会更新 max_len。后面的输出中会用到这个值。

因而调用 for_each()会将输入的所有单词都插入到这个 map 容器中——不管有多少单词——并且累加计算出每个单词的出现次数，计算出最大单词的长度——很不错，一条语句就实现了上面这些功能。

下面是程序输出的结果：

```
Enter some text and enter * to end:
How much wood would a wood chuck chuck,
If a woodchuck could chuck wood?
A woodchuck would chuck as much wood as a woodchuck could chuck
if a woodchuck could chuck wood.
*
A       1 How       1 If          1 a         4
as      2 chuck     6 could       3 if        1
much    2 wood      5 woodchuck   4 would     2
```

在这个示例中，map 容器中保存的是整型对象，所以可以对容器的下标运算符返回的值运用自增运算符。当 map 的下标运算符返回的值是类类型的对象时，也可以对它们使用运算符，只要这个类实现了对应的运算符。为了说明我们所讨论的这种情况，下面创建另一个示例。

假设我们要通过人名来保存并检索名人名言。显然，一个名人会有很多名言，因此我

们需要通过单个键来保存多个名言。我们不能在 map 容器中保存重复的键，但是我们可以将键关联到封装了多个名言的对象上。我们可以用 Ex4_01 中的 Name 类作为键，然后定义 Quotations 类用来保存指定名人的所有名言。

我们知道，可以用键的下标运算符来访问和键关联的对象，因此我们可以通过扩展 Quotations 类的成员函数 operator[]()来实现这个功能。为了方便向 Quotation 类中添加名言，我们还在类中实现了operator<<()。我们可以方便地将名言保存在 vector 容器中。下面就是定义了这个类的 Quotations.h 头文件的内容：

```
#ifndef QUOTATIONS_H
#define QUOTATIONS_H
#include <vector>                    // For vector container
#include <string>                    // For string class
#include <exception>                 // For out_of_range exception

class Quotations
{
private:
  std::vector<std::string> quotes;   // Container for the quotations

public:
  // Stores a new quotation that is created from a string literal
  Quotations& operator<<(const char* quote)
  {
    quotes.emplace_back(quote);
    return *this;
  }

  // Copies a new quotation in the vector from a string object
  Quotations& operator<<(const std::string& quote)
  {
    quotes.push_back(quote);
    return *this;
  }

  // Moves a quotation into the vector
  Quotations& operator<<(std::string&& quote)
  {
    quotes.push_back(std::move(quote));
    return *this;
  }

  // Returns a quotation for an index
  std::string& operator[](size_t index)
  {
    if(index < quotes.size())
      return quotes[index];
    else
      throw std::out_of_range {"Invalid index to quotations."};
  }
```

```
size_t size() const { return quotes.size(); }
// Returns the number of quotations

// Returns the begin iterator for the quotations
std::vector<std::string>::iterator begin()
{
  return std::begin(quotes);
}

// Returns the const begin iterator for the quotations
std::vector<std::string>::const_iterator begin() const
{
  return std::begin(quotes);
}

// Returns the end iterator for the quotations
std::vector<std::string>::iterator end()
{
  return std::end(quotes);
}

// Returns the const end iterator for the quotations
std::vector<std::string>::const_iterator end() const
{
  return std::end(quotes);
}
};
#endif
```

这里用<<运算符来添加名言是合理的，它可以在其他一些场景下使用，例如输入流。这里也可以用+=运算符来代替。这个类定义了 3 个版本的operator<<()，提供了不同的方式去添加名言。第一个版本接收一个字符串常量参数，然后把它传给 vector 的成员函数 emplace_back()，emplace_back()会调用 string 的构造函数以在适当的位置生成元素。第二个版本只有一个参数，它是 string 对象的引用，这个参数会被传给 vector 的成员函数 push_back()。第三个版本有一个右值引用参数。当在函数体中通过名称使用右值引用时，它会变成左值，因此必须使用 move()函数将它变为右值，然后把它传给 vector 的成员函数 push_back()。这会保证对象总是移动传值，而不是复制传值。

类的成员函数[]()可以通过索引来访问成员元素。当索引不在范围内时，这个函数将抛出一个异常，这种情况不应该发生；如果真的发生，这会是程序中的一个 bug。

在 vector 容器中，begin()和 end()返回指向名言的迭代器。需要注意的是，返回类型是指定的。提供迭代器的容器通常会定义一个迭代器成员变量，作为它们支持的迭代器类型的别名，所以不需要知道类型的具体细节。类对象定义的迭代器可以结合 for 循环使用，但要求迭代器至少是正向迭代器。

在 Quotations 类中也定义了 const 版本的 begin()和 end()，它们的返回值都是 const 类型的迭代器。这个返回类型有一个别名，定义在 vector 模板中。如果没有定义 const 版的 begin()和 end()函数，就不能在 for 循环中使用 const 类型的循环变量，例如：

```
for(const auto& pr : quotations)          // Requires const iterators
  ...
```

可以在 main()中定义两个内联辅助函数。第一个用来从 cin 读入 name：

```
inline Name get_name()
{
  Name name {};
  std::cout << "Enter first name and second name: ";
  std::cin >> std::ws >> name;
  return name;
}
```

这里读取的 name 用来作为名和姓。控制符 ws 用来消除空格，因此会跳过 cin 中剩下的字符。

第二个辅助函数用来读取名言：

```
inline string get_quote(const Name& name)
{
  std::cout << "Enter the quotation for " << name
            << ". Enter * to end:\n";
  string quote;
  std::getline(std::cin >> std::ws, quote, '*');
  return quote;
}
```

可以输入多行文本，然后用*号终止输入。Ex4_03.cpp 支持保存名言：

```
// Ex4_03.cpp
// Stores one or more quotations for a name in a map
#include <iostream>                       // For standard streams
#include <cctype>                         // For toupper()
#include <map>                            // For map containers
#include <string>                         // For string class
#include "Quotations.h"
#include "Name.h"
using std::string;

// get_name() definition goes here...

// get_quote() definition goes here...

int main()
{
  std::map<Name, Quotations> quotations; // Container for name/quotes pairs

  std::cout << "Enter 'A' to add a quote."
               "\nEnter 'L' to list all quotes."
               "\nEnter 'G' to get a quote."
               "\nEnter 'Q' to end.\n";
Name name {};                             // Stores a name
```

```
string quote {};                          // Stores a quotation
char command {};                          // Stores a command

while(command != 'Q')
{
  std::cout << "\nEnter command: ";
  std::cin >> command;
  command = static_cast<char>(std::toupper(command));
  switch(command)
  {
  case 'Q':
    break;                                // Quit operations

  case 'A':
    name = get_name();
    quote = get_quote(name);
    quotations[name] << quote;
    break;

  case 'G':
  {
    name = get_name();
    const auto& quotes = quotations[name];
    size_t count = quotes.size();
    if(!count)
    {
     std::cout << "There are no quotes recorded for "
               << name << std::endl;
     continue;
    }
    size_t index {};
    if(count > 1)
    {
     std::cout << "There are " << count << " quotes for " << name << ".\n"
               << "Enter an index from 0 to " << count - 1 << ": ";
     std::cin >> index;
    }
    std::cout << quotations[name][index] << std::endl;
  }
  break;
  case 'L':
  if(quotations.empty()) // Test for no pairs
  {
    std::cout << "\nNo quotations recorded for anyone." << std::endl;
  }
  // List all quotations
  for(const auto& pr : quotations) // Iterate over pairs
  {
    std::cout << '\n' << pr.first << std::endl;
    for(const auto& quote : pr.second) // Iterate over quotations
```

```
        {
          std::cout << " " << quote << std::endl;
        }
      }
      break;

    default:
      std::cout << " Command must be 'A', 'G', 'L', or 'Q'. Try again.\n";
      continue;
      break;
    }
  }
}
```

quotations 容器保存的是 pair<const Name, Quotations>对象类型的元素。像 quotations[name]这种表达式可以引用 Name 对象 name 关联的对象。如果在 map 中不存在和键值 name 关联的 pair 对象，就用默认关联的 Quotations 对象生成一个 pair 对象，默认的 Quotations 对象为空。下面的语句会为 name 保存一条新的名言 quote：

```
quotations[name] << quote;
```

<<左边的操作数等同于 quotations.operator[](name)，它返回一个和 name 关联的 Quotations 对象，因此这条语句等价于。

```
quotations.operator[](name).operator<<(quote);
```

在 main()函数中可以看到，我们利用表达式 quotations[name][index]来得到一条名言，它等价于 quotations.operator[](name).operator[](index)，你应该知道 main()剩下的代码是如何工作的，下面就是一些示例输出：

```
Enter 'A' to add a quote.
Enter 'L' to list all quotes.
Enter 'G' to get a quote.
Enter 'Q' to end.

Enter command: a
Enter first name and second name: Winston Churchill
Enter the quotation for Winston Churchill . Enter * to end:
There are a terrible lot of lies going around the world, and the worst of
it is half of them are true.*

Enter command: a
Enter first name and second name: Dorothy Parker
Enter the quotation for Dorothy Parker . Enter * to end:
Beauty is only skin deep, but ugly goes clean to the bone.*

Enter command: a
Enter first name and second name: Winston Churchill
Enter the quotation for Winston Churchill . Enter * to end:
Never in the field of human conflict was so much owed by so many to so few.*
```

```
Enter command: a
Enter first name and second name: Winston Churchill
Enter the quotation for Winston Churchill . Enter * to end:
Courage is what it takes to stand up and speak, Courage is also what it takes
to sit down and listen.*
Enter command: a
Enter first name and second name: Dorothy Parker
Enter the quotation for Dorothy Parker . Enter * to end:
Money cannot buy health, but I'd settle for a diamond-studded wheelchair.*

Enter command: g
Enter first name and second name: Winston Churchill
There are 3 quotes for Winston Churchill .
Enter an index from 0 to 2: 1
Never in the field of human conflict was so much owed by so many to so few.

Enter command: L

Winston Churchill
  There are a terrible lot of lies going around the world, and the worst of
it is half of them are true.
  Never in the field of human conflict was so much owed by so many to so few.
  Courage is what it takes to stand up and speak, Courage is also what it
takes to sit down and listen.

Dorothy Parker
  Beauty is only skin deep, but ugly goes clean to the bone.
  Money cannot buy health, but I'd settle for a diamond-studded wheelchair.

Enter command: q
```

显然，这个程序可以有更好的容错能力，也可以支持忽略大小的键值比较，这取决于你的想法。

map容器的成员函数find()可以返回一个元素的迭代器，这个元素的键值和参数匹配。例如：

```
std::map<std::string, size_t> people {{"Fred", 45}, {"Joan", 33}, {"Jill",
22}};
std::string name{"Joan"};
auto iter = people.find(name);
if(iter == std::end(people))
  std::cout <<"Not found.\n";
else
  std:: cout << name << " is " << iter->second << std::endl;
```

如果没有和参数匹配的元素，find()函数会返回容器的结束迭代器，因此在使用这个迭代器之前，必须先对它进行检查。

为了兼容multimap，map容器包含了成员函数equal_range()、upper_bound()和lower_bound()，因为这些函数会用来查找具有相同键的多个元素。稍后在multimap容器这一节

会对它们进行深入探讨。

4.2.5 删除元素

map 的成员函数 erase()可以移除键和参数匹配的元素，然后返回所移除元素的个数，例如：

```
std::map<std::string, size_t> people {{"Fred", 45}, {"Joan", 33}, {"Jill",
22}};
std::string name{"Joan"};
if(people.erase(name))
  std::cout << name << " was removed." << std::endl;
else
  std::cout << name << " was not found." << std::endl;
```

显然，map 容器的返回值只可能是 0 或 1，0 表明元素不在容器中。也可以用指向删除元素的迭代器作为 erase()的参数。这种情况下，返回的迭代器指向被删除元素的下一个位置。这个参数必须是容器中的有效迭代器，不能是结束迭代器。如果迭代器参数指向的是容器的最后一个元素，那么会返回结束迭代器。例如：

```
auto iter = people.erase(std::begin(people));
if(iter == std::end(people))
  std::cout << "The last element was removed." << std::endl;
else
  std::cout << "The element preceding " << iter->first << " was removed."
            << std::endl;
```

当最后一个元素被移除时，这段代码会输出一条消息或被移除元素后面元素的键。

也有更高级版本的 erase()，它可以移除两个迭代器参数所定义范围内的元素，例如：

```
auto iter = people.erase(++std::begin(people), --std::end(people));
// Erase all except 1st & last
```

返回的迭代器指向这段元素中最后一个被删除的元素。如果想删除容器中的所有元素，可以调用成员函数 clear()。

4.3 pair<>和 tuple<>的用法

我们已经知道 pair<const K,T>对象是如何封装键及其关联的对象，也了解了 pair<const K,T>对象是如何表示 map 容器中的元素的。一般来说，pair 对象可以封装任意类型的对象，可以生成任何想生成的pair<T1,T2>对象——可以是数组对象或者包含pair<T1,T2>的vector容器。例如，pair 可以封装两个序列容器或两个序列容器的指针。pair<T1,T2>模板定义在 utility 头文件中，如果不想使用 map 而只想使用 pair 对象，可以包含这个头文件。

tuple<>模板是 pair 模板的泛化，但允许定义 tuple 模板的实例，可以封装不同类型的

任意数量的对象，因此 tuple 实例可以有任意数量的模板类型参数。tuple 模板定义在 tuple 头文件中。tuple 这个术语也适用于很多其他的场景，例如数据库，这里一个 tuple 就是由一些类型的不同数据项组成的，这和 tuple 的概念相似。我们发现 tuple 对象有很多用途。当需要将多个对象当作一个对象传给函数时，tuple 类型是很有用的。显然这里也可以定义一个由几个对象组成的容器元素。下面先详细讲述 pair<T1,T2>模板的使用，然后再探讨 tuple 对象的生成和使用。

4.3.1　pair 的操作

考虑到 pair 是一个比较简单的模板类型，它只有两个 public 数据成员 first 和 second。令人惊讶的是，它却可以构造各种不同的 pair<T1, T2>。我们已经知道如何使用 first 和 second 来创建对象。和右值引用参数一样，也有很多版本的引用参数，而且有一些版本的右值引用参数允许参数隐式转换为所需的类型。例如，下面有 4 种不同的方式来创建一个 pair 对象：

```
std::string s1 {"test"}, s2{"that"};
std::pair<std::string, std::string> my_pair{s1, s2};
std::pair<std::string, std::string> your_pair{std::string {"test"},
std::string {"that"}};
std::pair<std::string, std::string> his_pair{"test", std::string {"that"}};
std::pair<std::string, std::string> her_pair{"test", "that"};
```

第一个 pair 构造函数复制了所有参数的值，第二个移动参数值，第三个为了隐式转换而将第一个参数传给 string 的构造函数，最后一个构造函数将两个参数隐式转换为 string 对象而且它们会被移到 pair 的成员变量 first 和 second 中。由于这个构造函数有右值引用参数版本，因此任意一个或两个模板类型参数可以是 unique_ptr<T>。

make_pair<T1,T2>函数模板是一个辅助函数，可以生成并返回一个 pair<T1,T2>对象。可以如下所示生成先前代码块中的 pair 对象：

```
auto my_pair = std::make_pair(s1, s2);
auto your_pair = std::make_pair(std::string {"test"}, std::string {"that"});
auto his_pair = std::make_pair<std::string, std::string>("test",
                std::string {"that"});
auto her_pair = std::make_pair<std::string, std::string>("test", "that");
```

前两条语句中的函数模板的类型参数由编译器推断。在最后两条语句中，类型是明确的。如果在最后两条语句中忽略模板类型参数，那么对象类型将是 pair<const char*, string>和 pair<const char*, const char*>。

pair 对象也可以复制或移动构造它的成员变量。例如：

```
std::pair<std::string, std::string> new_pair{my_pair}; // Copy constructor
std::pair<std::string, std::string>
   old_pair{std::make_pair(std::string{"his"}, std::string{"hers"})};
```

old_pair 是由 pair<string,string>类的移动构造函数生成的。

另一个 pair 构造函数使用了 C++11 引入的这种机制，它允许通过在适当的位置生成 first 和 second 对象来构建 pair<T1, T2>。T1 和 T2 的构造函数的参数作为 tuple 参数传给 pair 的构造函数。下一节会介绍如何使用 tuple 对象。下面是一个使用 pair 对象的示例：

```
std::pair<Name, Name> couple{std::piecewise_construct,
std::forward_as_tuple("Jack", "Jones"), std::forward_as_tuple("Jill",
   "Smith")};
```

这里，pair 构造函数的第一个参数是一个定义在 utility 头文件中的 piecewise_construct 类型的实例，这是一个用来作为标签或标记的空类型。这个piecewise_construct 参数唯一的作用是区分这个构造函数的调用和有两个 tuple 参数的构造函数调用之间的区别，后者的两个参数通常用来作为 pair 成员变量 first 和 second 的值。这里，构造函数的第二和第三个参数指定了构造 first 和 second 对象的参数集，forward_as_tuple()是一个定义在 tuple 头文件中的函数模板。这里用它的转发参数生成了一个 tuple 引用。不会经常用到这种 pair 的构造函数，但它为不支持拷贝或移动运算的 T1 和 T2 类型提供了在适当位置生成 pair<T1, T2>对象的独特能力。

注意，如果参数是一个临时对象，forward_as_tuple()函数会生成一个右值引用的 tuple。例如：

```
int a {1}, b {2};
const auto& c = std::forward_as_tuple(a,b);
```

这里 c 的类型是 tuple<int &,int&>，因此成员变量是引用。但是假设这样写声明的话：

```
const auto& c = std::forward_as_tuple(1,2);
```

这里 c 的类型是 tuple<int &,int&>，成员变量作为值引用。

如果成员变量可以被复制和移动，pair 对象就支持复制和移动赋值。例如：

```
std::pair<std::string, std::string> old_pair;    // Default constructor
std::pair<std::string, std::string> new_pair{std::string{"his"},
std::string{"hers"}};
old_pair = new_pair;                             // Copy assignment
new_pair = pair<std::string, std::string>
{std::string{"these"}, std::string{"those"}};    // Move assignment
```

默认的 pair 构造函数会用它的成员变量——空的 string 对象来生成 old_pair，这是一个空的字符串对象。第 3 条语句一个成员一个成员地将 new_pair 复制到 old_pair 中。第 4 条语句将作为赋值运算符的右操作数的 pair 对象的成员变量移到 new_pair 中。

当 pair 对象包含不同类型的成员变量时，也可以将一个 pair 对象赋值给另一个 pair 对象，只要作为右操作数的 pair 对象的成员变量可以隐式转换为左操作数的 pair 对象的成员变量的类型。例如：

```
auto pr1 = std::make_pair("these", "those"); // Type pair<const char*, const char*>
```

```
std::pair<std::string, std::string> pr2;    // Type pair<string, string>
pr2 = pr1;                                  // OK in this case
```

pr1 成员变量 first 和 second 的类型是 const char*。这个类型可以隐式转换为 string，即 pr2 成员变量的类型，因此可以成功赋值。如果这些类型不能隐式转换，这条赋值语句就无法通过编译。

pair 对象有全套的运算符==、!= 、<、<=、>、>=。这些运算符都可以正常使用，作为操作数的 pair 对象的类型必须是相同的，它们的成员变量的比较方式也必须相同。相等运算符返回 true，如果左右操作数的成员变量相等的话：

```
std::pair<std::string, std::string> new_pair;
new_pair.first = "his";
new_pair.second = "hers";
if(new_pair == std::pair<std::string, std::string> {"his", "hers"})
   std::cout << "Equality!\n";
```

new_pair 的成员变量 first 和 second 被赋值为右操作数所包含的字符串。如果 pair 对象是相等的，if 语句会输出一些消息。当两个 pair 对象中的任何一个或两个成员不相等时，!= 比较会返回 true。

对于小于或大于比较，pair 对象的成员变量是按字典顺序比较的。如果 new_pair.first 小于 old_pair.first 的话，表达式 new_pair<old_pair 会返回 true。如果它们的成员变量 first 相等，但 new_pair.second 小于 old_pair.second，new_pair < old_pair 也为 true。下面是一个示例：

```
std::pair<int, int> p1 {10, 9};
std::pair<int, int> p2 {10, 11};
std::pair<int, int> p3 {11, 9};
std::cout << std::boolalpha << (p1 < p2) << " "  // Outputs "true"
          << (p1 > p3) << " "                    // Outputs "false"
          << (p3 > p2) << std::endl;             // Outputs "true"
```

第一个比较的结果为 true，因为 p1 和 p2 的成员变量 first 相等，p1 的成员变量 second 小于 p2 的成员变量 second。第二个比较的结果为 false，因为 p1 的 first 小于 p3 的 first。第三个比较的结果则为 true，因为 p3 的 first 大于 p2 的 first。

pair 的成员函数 swap()可以和作为参数传入的另一个 pair 对象交换其成员变量 first 和 second。显然，参数必须是相同类型。下面有一个示例：

```
std::pair<int, int> p1 {10, 11};
std::pair<int, int> p2 {11, 9};
p1.swap(p2);                                  // p1={11,9} p2={10,11}
```

如果执行两次 swap()，对象恢复成原来的值。

4.3.2 tuple 的操作

生成 tuple 对象的最简单方式是使用定义在 tuple 头文件中的辅助函数 make_tuple()。这个函数可以接受不同类型的任意个数的参数，返回的 tuple 的类型由参数的类型决定。例如：

```
auto my_tuple = std::make_tuple(Name{"Peter", "Piper"}, 42,
std::string{"914 626 7890"});
```

my_tuple 对象是 tuple<Name,int,string>类型，因为模板类型参数是由 make_tuple()函数的参数推导出来的。如果提供给 make_tuple()的第三个参数是一个字符串常量，my_tuple 的类型将是 tuple<Name,int,const*>，这和之前的不同。

tuple 对象的构造函数提供了可能会用到的每一种选择。例如：

```
std::tuple<std::string, size_t> my_t1;          // Default initialization
std::tuple<Name, std::string> my_t2{Name{"Andy", "Capp"},
   std::string{"Programmer"}};
std::tuple<Name, std::string> copy_my_t2{my_t2}; // Copy constructor
std::tuple<std::string, std::string> my_t3{"this", "that"};
// Implicit conversion
```

tuple 中的对象由默认构造函数用默认值初始化。为 my_t2 调用的构造函数将参数移到 tuple 的元素中。下一条语句会调用拷贝构造函数来生成 tuple，在最后一个构造函数调用中，将参数隐式转换为 string 类型并生成了一个 tuple 元素。

也可以用 pair 对象构造 tuple 对象，pair 可以是左值，也可以是右值。显然，tuple 只能有两个元素。下面有两个示例：

```
auto the_pair = std::make_pair("these", "those");
std::tuple<std::string, std::string> my_t4 {the_pair};
std::tuple<std::string, std::string> my_t5 {std::pair <std::string,
   std::string > {"this", "that"}};
```

第二条语句从 the_pair 生成了一个 tuple，它是一个左值。the_pair 的成员变量 first 和 second 可以隐式转换为这个 tuple 中的元素的类型。最后一条语句从右值 pair 对象生成了一个 tuple。

可以用任何比较运算符来比较相同类型的 tuple 对象。tuple 对象中的元素是按照字典顺序比较的。例如：

```
std:: cout << std::boolalpha << (my_t4 < my_t5) << std::endl; // true
```

tuple 对象中的元素是依次比较的，第一个不同的元素决定了比较结果。my_t4 的第一个元素小于 my_t5 的第一个元素，因此比较结果为 true。如果是相等比较，任何一对不相等的对应元素都会使比较结果为 false。

tuple 对象的成员函数 swap()可以将它的元素和参数交换。参数的类型必须和 tuple 对象的类型一致。例如：

```
my_t4.swap(my_t5);
```

通过调用成员函数 swap()来交换 my_t4 和 my_t5 对应的元素。显然，tuple 中所有元素的类型都必须是可交换的，tuple 头文件中定义了一个全局的 swap()函数，它能够以相同的方式交换两个 tuple 对象的元素。

因为 tuple 是 pair 的泛化，所以它的工作方式不同。pair 的对象个数是固定的，因此它们有成员名。tuple 中的对象数目是不固定的，所以访问它们的机制必须能够满足这种情况。函数模板 get<>()可以返回 tuple 中的一个元素。第一个模板类型参数是 size_t 类型值，它是 tuple 中元素的索引，因此 0 会选择 tuple 中的第一个元素，1 会选择第二个元素，以此类推。get<>()模板剩下的类型参数是和 tuple 的参数同样推导的。下面是一个使用 get<>()和索引来获取元素的示例：

```
auto my_tuple = std::make_tuple(Name{"Peter", "Piper"}, 42, std::string
  {"914 626·7890"});
std::cout << std::get<0>(my_tuple)
          << " age = " << std::get<1>(my_tuple)
          << " tel: " << std::get<2>(my_tuple) << std::endl;
```

在输出语句中第一次调用 get<>()时返回了 my_tuple 中第一个元素的引用，它是一个 Name 对象，第二次调用 get<>()时返回了下一个元素的引用，它是一个整数；第三次调用 get<>()时返回了第三个元素的引用，它是一个 string 对象。因此输出结果是：

```
Peter Piper age = 42 tel: 914 626 7890
```

也可以用基于类型的 get<>()从 tuple 获取元素，但要求 tuple 中只有一个这种类型的元素。例如：

```
auto my_tuple = std::make_tuple(Name{"Peter", "Piper"}, 42, std::string
  {"914 626 7890"});
std::cout << std::get<Name>(my_tuple)
          << " age = " << std::get<int>(my_tuple)
          << " tel: " << std::get<std::string>(my_tuple) << std::endl;
```

如果 tuple 中包含的 get<>()类型参数值的元素不止一个，代码就无法编译通过。这里 tuple 的全部 3 个成员为不同类型，所以可以正常使用。

全局的 tie<>()函数模板定义在 tuple 头文件中，它提供了另一种访问 tuple 元素的方式。这个函数可以把 tuple 中的元素值转换为可以绑定到 tie<>()的左值集合。tie<>()的模板类型参数是从函数参数中推导的。例如：

```
auto my_tuple = std::make_tuple(Name{"Peter", "Piper"}, 42,
   std::string{"914 626 7890"});
Name name{};
size_t age{};
std::string phone{};
std::tie(name, age, phone) = my_tuple;
```

在最后一条语句中，赋值运算符的左操作数表达式会返回一个参数的 tuple 引用。因此，赋值运算符左右的操作数都是 tuple 对象，并且用 my_tuple 中的元素值来对 tie()参数中的变量赋值。我们可能并不想存储每一个元素的值。下面展示了如何只保存 my_tuple 中 name 和 phone 的值：

```
std::tie(name, std::ignore, phone) = my_tuple;
```

ignore 定义在 tuple 中，它被用来标记 tie()函数中要被忽略的值。tuple 中被忽略的元素的值将不会被记录下来。在这个示例中只复制了第一个和第三个元素。

也可以用 tie()函数来实现对类的数据成员的字典比较。例如，可以在 Ex4_01 的 Name 类中实现 operator<()函数：

```
bool Name::operator<(const Name& name) const
{
  return std::tie(second, first) < std::tie(name.second, name.first);
}
```

在这个函数体中，调用 tie()得到的 tuple 对象的元素是按顺序比较的。用<运算符来比较连续的元素对，出现的第一对不同值会决定比较的结果；这个表达式的比较结果就是不同元素的比较结果。如果全部元素都相等或等价，那么结果为 false。

4.3.3 tuples 和 pairs 实战

让我们在一个示例中练习 tuple 和 pairs 的使用。这个示例不必是反映它们使用的最佳方式，而是为了尝试对 tuple 和 pair 的使用。这个示例会充分利用一个 map 容器，这个 map 容器以 pair 对象作为键，以 tuple 作为和 pair 对象关联的对象。每一个 map 元素会记录一个人的一些数据。以人的名字作为键，和键关联的 tuple 对象以生日、身高和职业这些信息为元素。生日也是一个 tuple，因此我们将生成一个以 tuple 为元素的 tuple。这个示例中会使用一些类型别名来使代码更简洁：

```
using std::string;
using Name = std::pair<string, string>;                  // Defines a name
using DOB = std::tuple<size_t, size_t, size_t>;    // Month, day, year
using Details = std::tuple< DOB, size_t, string> ;
// DOB, height(inches), occupation
using Element_type = std::map<Name, Details>::value_type; // Type of map
                                                          // element
```

Name 是 pair 类型的别名，它封装了两个 string 对象。DOB 是 tuple 类型的别名，它有三个 size_t 元素，分别是月、日、年。Details 是和键关联的对象的别名，它是一个有三个元素的 tuple，分别是 DOB、表示年龄的 size_t 类型值以及表示职业的 string。map 容器的元素类型是 pair<const K,T>对象，在这个示例中此类型是很混乱的。通过指定 map 容器的成员 value_type 的类型，我们可以很容易地为它定义 Element_type 别名。如果做了替换，可以显式看到 map 元素全部的名称类型。

```
std::pair<std::pair<std::string, std::string>,
          std::tuple<std::tuple<size_t, size_t, size_t>, size_t,
          std:: string>>
```

上面展示了类型别名是多么有用。

我们可以像下面这样用别名来定义容器：

```
std::map<Name, Details> people; // Records of the people
```

键是 Name 类型，关联的对象是 Details 类型——别名定义真的很简单。更进一步，我们可以为容器类型定义别名：

```
using People = std::map<Name, Details>;
```

现在我们定义的容器是这个样子：

```
People people; // Records of the people
```

我们可以将 map 元素的输入过程封装到一个函数中：

```
void get_people(Peoples& people)
{
  string first {}, second {}; // Stores name inputs
  size_t month {}, day {}, year {}; // Stores DOB input
  size_t height {}; // Stores height input
  string occupation {}; // Stores occupation input
  char answer {'Y'};

  while(std::toupper(answer) == 'Y')
  {

    std::cout << "Enter a first name and a second name: ";
    std::cin >> std::ws >> first >> second;

    std::cout << "Enter date of birth as month day year (integers): ";
    std::cin >> month >> day >> year;
    DOB dob {month, day, year}; // Create DOB tuple

    std::cout << "Enter height in inches: ";
    std::cin >> height;

    std::cout << "Enter occupation: ";
    std::getline(std::cin >> std::ws, occupation, '\n');
    // Create the map element in place- a pair containing a Name pair and
       //a tuple object
    people.emplace(std::make_pair(Name {first, second},
       std::make_tuple(dob, height, occupation)));

    std::cout << "Do you want to enter another(Y or N): ";
    std::cin >> answer;
  }
}
```

大部分代码都用于流的输入。用 getline()读入职业时可以输入多行文字描述。getline()函数的第一个参数会消除上一次输入操作留在输入缓冲区的空格，这里就是这种情况。如果缓冲区中有新行，getline()会读入一个空行。tuple 中保存的出生日期是从输入中读取的，使用 DOB 作为 tuple 类型的别名。通过调用 map 的成员函数 emplace()在合适的位置生成 pair 元素。pair 元素的成员 first 是 pair<string,string>对象，它是用 Name 别名生成的，因此会调用它的构造函数。pair 的成员 second 是 tuple 对象，它包含了 DOB 这个 tuple、身高、职业。它是通过调用辅助函数 make_tuple()生成的。

为 map 读取输入数据后，这个程序会按职业顺序列出人名。另一个函数会实现这些：

```
void list_DOB_Job(const People& people)
{
  DOB dob;
  string occupation {};
  std::cout << '\n';
  for(auto iter = std::begin(people); iter != std::end(people); ++iter)
  {
    std::tie(dob, std::ignore, occupation) = iter->second;
    std::cout << std::setw(20) << std::left << (iter->first.first + " " +
                 iter->first.second)
              << "DOB: " << std::right
              << std::setw(2) << std::get<0>(dob) << "-"
              << std::setw(2) << std::setfill('0') << std::get<1>(dob) << "-"
              << std::setw(4) << std::get<2>(dob) << std::setfill(' ')
              << " Occupation: " << occupation << std::endl;
  }
}
```

在 for 循环中使用迭代器进行输出——只是为了展示我们可以这样使用。它比使用基于范围的循环要简单，但在下一个函数中会用到它。循环中的第一条语句是一条赋值语句。

左操作数是 tie()函数的调用，它可以将函数参数作为左值生成一个 tuple。赋值运算符的右操作数是 iter 指向的 pair 的成员变量 second，它是一个 Details 类型的 tuple。赋值运算将作为右操作数的 tuple 的成员复制到左操作数的 tuple 成员中。因为 tie()的第二个参数是 ignore，所以只会保存赋值运算符右边 tuple 的第一个和第三个成员，它们分别被保存到 tuple 类型的变量 dob 和 occupation 中。

循环体中的第二条语句会输出人名、生日、职业。人的名和姓分别被保存在 pair 元素的成员变量 first 和 second 中：也就是说，key.iter 会指向 pair 元素，所以 iter->first 是键对象的引用；因此 iter->first.first 可以访问作为键的 pair 的成员变量 first，iter->first.second 可以访问成员变量 second。可以用 get<>()函数模板来访问 DOB 这个 tuple 的成员。通过 get<>()模板的参数来选择 tuple 的成员。

我们也可以包含一个用来输出每个人的详细信息的函数。如果可以选择记录的任何字段来对输出排序，将是很好的。实现这种功能的一种方式是允许将一个函数对象作为参数，这个函数对象用来比较和键关联的 Details 对象的成员。下面是实现代码：

```
template<typename Compare>
void list_sorted_people(const People& people, Compare comp)
{
  std::vector< Element_type*> folks;
  for(const auto& pr : people)
    folks.push_back(&pr);

  // Lambda to compare elements via pointers
  auto ptr_comp =
       [&comp](const Element_type* pr1, const Element_type* pr2)->bool
            { return comp(*pr1, *pr2); };

std::sort(std::begin(folks), std::end(folks), ptr_comp);
// Sort the pointers to elements

// Output the sorted elements
DOB dob {};
size_t height {};
string occupation {};
std::cout << '\n';

for(const auto& p : folks)
{
  std::tie(dob, height, occupation) = p->second;
  std::cout << std::setw(20) << std::left << (p->first.first + " " +
    p->first.second)
            << "DOB: " << std::right << std::setw(2) << std::get<0>(dob) << "-"
            << std::setw(2) << std::setfill('0') << std::get<1>(dob) << "-"
            << std::setw(4) << std::get<2>(dob) << std::setfill(' ')
            << " Height: " << height
            << " Occupation: " << occupation << std::endl;
  }
}
```

在函数调用中，这个函数模板可以推导出决定输出顺序的函数对象的类型。map 元素的顺序由键的顺序决定，因此只能在 map 容器外对元素重新排序。我们可以将这些元素复制到另一个容器中，但更好、更有效率的方式是在另一个容器中保存这些元素的指针，然后用第二个参数传入的函数对指针进行排序。

vector 容器中保存的指针是 const Element_type*类型的原生指针。这里使用 unique_ptr 是不明智的，因为 unique_ptr<T>拥有它所指向的 T 类型对象。如果 vector 包含 unique_ptr<T> 类型的元素，副本会由 map 中的元素组成，也就违背了使用指针的初衷。这里使用原生指针没有什么缺点，因为 vector 及其元素都是函数中的局部变量，只用来作为 map 元素的观察者。

vector 中的元素保存的是 map 中 pair 对象的地址，它们是在 map 的循环遍历中生成的。调用者不需要知道用指针进行的排序是否完成，所以 list_sorted_people()函数模板假设传给它的函数对象已经实现了两个 map 元素的比较。lambda 表达式 ptr_comp 定义在函数体中，它将传入的 map 元素的指针参数解引用，然后把解引用后的结果传给 comp 来进行比较。

因此，在sort()函数使用 ptr_comp 函数来对 vector 中的指针进行排序。tie()函数会提取 Details 的全部元素到局部变量中，然后输出名称以及与之关联的 Details 中的元素。

包含 main()的源文件中的内容如下：

```
// Ex4_04.cpp
// Using tuples and pairs
#include <iostream>                          // For standard streams
#include <iomanip>                           // For stream manipulators
#include <string>                            // For string class
#include <cctype>                            // For toupper()
#include <map>                               // For map container
#include <vector>                            // For vector container
#include <tuple>                             // For tuple template
#include <algorithm>                         // For sort() template

using std::string;
using Name = std::pair <string, string>; // Defines a name pair
using DOB = std::tuple <size_t, size_t, size_t>; // Month, day, year tuple
using Details = std::tuple < DOB, size_t, string > ; // DOB, height(inches),
                                                     // occupation
using Element_type = std::map<Name, Details>::value_type; // Type of map element
using People = std::map<Name, Details>; // Type of people container

// Code for get_people() function goes here...

// Code for list_DOB_Job() function goes here...

// Code for list_sorted_people() function template goes here...

int main()
{
  std::map<Name, Details> people;                // Records of the people
  get_people(people);                            // Read all the people

  std::cout << "\nThe DOB & jobs are: \n";
  list_DOB_Job(people);                          // List names, DOB & job

  // Define height comparison for people
  auto comp = [](const Element_type& pr1, const Element_type& pr2)
  {
    return std::get<1>(pr1.second) < std::get<1>(pr2.second);
  };
  std::cout << "\nThe people in height order are : \n";
  list_sorted_people(people, comp);
}
```

这一长串的#include令人印象深刻，它们的后面定义了一些先前你所见到的类型别名。因为有 3 个函数调用，所以 main()中的代码相对简洁了很多。map 元素的比较器定义在 lambda 表达式中，它也是一个函数对象。在这个示例中，它会比较 Details 对象的第二个元素，也就是身高。像我们之前看到的那样，可以用 get<>()来提取身高值。list_sorted_people()

可以对任何属性进行排序，只要定义了map中的这个元素属性的比较器。下面是示例输出：

```
Enter a first name and a second name: Dan Druff
Enter date of birth as month day year (integers): 2 3 1978
Enter height in inches: 74
Enter occupation: Trichologist
Do you want to enter another(Y or N): y
Enter a first name and a second name: Jane Brudit
Enter date of birth as month day year (integers): 13 11 1990
Enter height in inches: 63
Enter occupation: Barista
Do you want to enter another(Y or N): y
Enter a first name and a second name: Will Derness
Enter date of birth as month day year (integers): 5 5 1981
Enter height in inches: 76
Enter occupation: Explorer
Do you want to enter another(Y or N): N
The DOB & jobs are:

Dan Druff       DOB: 2-03-1978 Occupation: Trichologist
Jane Brudit     DOB: 13-11-1990 Occupation: Barista
Will Derness    DOB: 5-05-1981 Occupation: Explorer

The people in height order are :
Jane Brudit     DOB: 13-11-1990 Height: 63 Occupation: Barista
Dan Druff       DOB: 2-03-1978 Height: 74 Occupation: Trichologist
Will Derness    DOB: 5-05-1981 Height: 76 Occupation: Explorer
```

4.4 multimap容器的用法

multimap容器保存的是有序的键/值对，但它可以保存重复的元素。multimap中会出现具有相同键的元素序列，它们会被添加到容器中。multimap和map有相同范围的构造函数，默认的比较键的函数是less<K>()。multimap大部分成员函数的使用方式和map相同。因为重复键的原因，multimap有一些函数的使用方式和map有一些区别。接下来介绍multimap的那些用法和map容器不同的成员函数。

multimap容器的成员函数insert()可以插入一个或多个元素，而且插入总是成功。这个函数有很多的版本都可以插入单个元素，它们都会返回一个指向插入元素的迭代器。下面有一个示例，假设我们已经使用了声明using std::string：

```
std::multimap<string, string> pets; // Element is pair{pet_type, pet_name}
auto iter = pets.insert(std::pair<string, string>{string{"dog"},
   string{"Fang"}});
iter = pets.insert(iter, std::make_pair("dog", "Spot")); // Insert Spot
                                                         // before Fang
pets.insert(std::make_pair("dog", "Rover")); // Inserts Rover after Fang
pets.insert(std::make_pair("cat", "Korky")); // Inserts Korky before all dogs
```

```
pets.insert({{"rat", "Roland"}, {"pig", "Pinky"}, {"pig", "Perky"}});
// Inserts list elements
```

第三条语句的第一个参数是一个作为提示符的迭代器，它说明了元素应该被插入的位置。元素会被立即插入到 iter 所指向元素的前面，因此，这使我们可以覆盖默认的插入位置。对于默认的插入位置来说，元素会被插入到先前插入的键为"dog"的元素的后面。元素默认是按照键的升序插入的。如果没有用提示符改变插入位置，有相同键的元素的位置和插入位置相同。最后一条语句插入了一些初始化列表中的元素。有高级版本的 insert()，它可以接收两个迭代器参数，用来指定插入元素的范围。

和 map 一样，multimap 的成员函数 emplace()可以在容器的适当位置构造元素。在插入具有相同键的元素时，可以使用 multimap 的成员函数 emplace_hint()，可以通过为这个函数提供一个迭代器形式的提示符来控制元素的生成位置：

```
auto iter = pets.emplace("rabbit", "Flopsy");
iter = pets.emplace_hint(iter, "rabbit", "Mopsy");
// Create preceding Flopsy
```

这两个函数都返回一个指向插入元素的迭代器。emplace_hint()函数尽可能近地在第一个参数所指向位置的前面生成一个新元素。如果只使用 emplace()来插入"Mopsy"，它可能会被插入到当前所有键为"rabbit"的元素的后面。

multimap 不支持下标运算符，因为键并不能确定一个唯一元素。和 map 相似，multimap 也不能使用 at()函数。multimap 的成员函数 find()可以返回一个键和参数匹配的元素的迭代器。例如：

```
std::multimap<std::string, size_t> people{ {"Ann", 25}, {"Bill", 46},
   {"Jack", 77},{"Jack", 32}, {"Jill", 32}, {"Ann", 35} };
std::string name {"Bill"};
auto iter = people.find(name);
if(iter != std::end(people)) std::cout << name << " is " << iter->second
    << std::endl;
iter = people.find("Ann");
if(iter != std::end(people)) std::cout << iter->first << " is " <<
    iter->second << std::endl;
```

如果没有找到键，会返回一个结束迭代器，所以我们应该总是对返回值进行检查。第一个find()调用的参数是一个键对象，因为这个键是存在的，所以输出语句可以执行。第二个find()调用的参数是一个字符串常量，它说明参数不需要和键是相同的类型。对容器来说，可以用任何值或对象作为参数，只要可以用函数对象将它们和键进行比较。最后一条输出语句也可以执行，因为有等于"Ann"的键。事实上，这里有两个等于"Ann"的键，在笔者的系统上，输出了 25 岁的 Ann。你可能也会得到相同的结果——第一个 Ann——但这无法保证。

如果使用 multimap 容器，几乎可以肯定它会包含键重复的元素；否则，就应该使用 map。一般来说，我们想访问给定键对应的所有元素。成员函数 equal_range()就可以做到这一点。它会返回一个封装了两个迭代器的 pair 对象，这两个迭代器所确定范围内的元素

的键和参数值相等。例如：

```
auto pr = people.equal_range("Ann");
if(pr.first != std::end(people))
{
    for(auto iter = pr.first ; iter != pr.second; ++iter)
      std:cout << iter->first << " is " << iter->second << std::endl;
}
```

equal_range()的参数可以是和键同类型的对象，或是不同类型的但可以和键比较的对象。返回的 pair 对象的成员变量 first 是一个迭代器，它指向第一个大于等于参数的元素；如果键和参数相等的元素存在的话，它是第一个键和参数相同的元素。如果键不存在，pair 的成员变量 first 就是容器的结束迭代器，所以应该总是对它们进行检查。pair 的成员变量 second 也是一个迭代器，它指向键值大于参数的第一个参数；如果没有这样的元素，它会是一个结束迭代器。这段代码会输出容器中键值为"Ann"的元素的一些信息。

multimap 的成员函数 lower_bound()会返回一个迭代器，它指向键值和参数相等或大于参数的第一个元素，或者指向结束迭代器。upper_bound()也返回一个迭代器，它指向键值大于函数参数的第一个元素，如果这样的元素不出现的话，它就是一个结束迭代器。所以，当存在一个或多个相等键时，这些函数会返回一个开始迭代器和一个结束迭代器，它们指定了和参数匹配的元素的范围，这和 equal_range()返回的迭代器是相同的。因而前面的代码段可以这样重写：

```
auto iter1 = people.lower_bound("Ann");
auto iter2 = people.lower_bound("Ann");
if(iter1 != std::end(people))
{
  for(auto iter = iter1 ; iter != iter2; ++iter)
    std::cout << iter->first << " is " << iter->second << std::endl;
}
```

它和前一个代码段的输出结果是相同的。通过调用 multimap 的成员函数 count()可以知道有多少个元素的键和给定的键相同。

```
auto n = people.count("Jack"); // Returns 2
```

可以用不同的方式使用这些函数。可以选择 find()或 equal_range()来访问元素。如果以班级为键，在 mutilmap 中保存学生信息，可以用成员函数 count()来获取班级的大小。当然，通过将在第 1 章介绍的 distance()函数模板运用到成员函数 equal_range()返回的迭代器或者 lower_bound()和 upper_bound()返回的迭代器上，也可以获取键和给定键相等的元素的个数：

```
std::string key{"Jack"};
auto n = std::distance( people.lower_bound(key),
people.upper_bound(key));              // No. of elements matching key
```

■ **注意**：全局的 equal_range()、lower_bound()、upper_bound()函数模板的使用方式和关联容器中同名成员函数的使用方式略有不同。在本书后面的部分你会了解到这些。

multimap 的成员函数 erase()有三个版本。其中一个版本以待删除元素的迭代器作为参数；这个函数没有返回值。第二个版本以一个键作为参数，它会删除容器中所有含这个键的元素；它返回容器中被移除元素的个数。第三个版本接受两个迭代器参数，它们指定了容器中的一段元素，这个范围内的所有元素都会被删除，这个函数返回的迭代器指向最后一个被删除元素的后一个位置。

下面在示例中尝试一些 multimap 操作：

```
// Ex4_05.cpp
// Using a multimap
#include <iostream>                          // For standard streams
#include <string>                            // For string class
#include <map>                               // For multimap container
#include <cctype>                            // For toupper()

using std::string;
using Pet_type = string;
using Pet_name = string;

int main()
{
  std::multimap<Pet_type, Pet_name> pets;
  Pet_type type {};
  Pet_name name {};
  char more {'Y'};
  while(std::toupper(more) == 'Y')
  {
   std::cout << "Enter the type of your pet and its name: ";
   std::cin >> std::ws >> type >> name;
  // Add element - duplicates will be LIFO
  auto iter = pets.lower_bound(type);
  if(iter != std::end(pets))
   pets.emplace_hint(iter, type, name);
  else
   pets.emplace(type, name);

  std::cout << "Do you want to enter another(Y or N)? ";
  std::cin >> more;
  }
  // Output all the pets
  std::cout << "\nPet list by type:\n";
  auto iter = std::begin(pets);
  while(iter != std::end(pets))
  {
   auto pr = pets.equal_range(iter->first);
   std::cout << "\nPets of type " << iter->first << " are:\n";
```

```
        for(auto p = pr.first; start != pr.second; ++p)
        std::cout << " " << p->second;
        std::cout << std::endl;
        iter = pr.second;
    }
}
```

我们在代码中使用一些类型别名将类型及其表示的事物关联了起来。pets 容器保存的是 pair<string,string>类型的对象，这个 pair 对象以 pet 类型作为键，以 pet 的名称为对象。代码中的第一个循环，将给定键的第二个以及随后的元素插入到这个键序列的前面。这里使用 emplace_hint()来插入元素。如果它是给定类型的第一个元素，就调用 emplace()在适当的位置创建元素。在第二个 while 循环中，按照 pet 的类型分组输出元素。首先找到 iter 指向的 pet 的第一个类型，然后用 equal_range()返回的迭代器列出这种 pet 类型的全部序列。最后将 iter 设为这个序列的结束迭代器，它也是一个指向下一个 pet 类型的迭代器，或是容器的结束迭代器。后者会结束循环。下面是一些示例输出：

```
Enter the type of your pet and their name: rabbit Flopsy
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: rabbit Mopsy
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: rabbit Cottontail
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: dog Rover
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: dog Spot
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: snake Slither
Do you want to enter another(Y or N)? y
Enter the type of your pet and their name: snake Sammy
Do you want to enter another(Y or N)? y

Enter the type of your pet and their name: cat Max
Do you want to enter another(Y or N)? n

Pet list by type:

Pets of type cat are:
  Max

Pets of type dog are:
  Spot Rover

Pets of type rabbit are:
  Cottontail Mopsy Flopsy

Pets of type snake are:
  Sammy Slither
```

输出表明元素是按键的升序排列的，键相同的元素的顺序和它们输入的顺序相反。

4.5 改变比较函数

对于为什么要改变 map 或 multimap 的比较函数，我们有一些理由：可能想用降序排列的元素来代替默认升序排列的元素；或者键需要使用的比较函数和直接的小于或大于运算符不同。例如，如果键是指针的话，就需要使用这种函数。在看一个演示如何替代比较函数的示例之前，先强调比较键的函数对象所需要的一个非常重要的条件。

警告：map 容器的比较函数在相等时不能返回 true。

也就是说，不能使用<=或>=来比较。这是为什么？map 或 multimap 容器用等价来判断键是否相等。如果表达式 key1<key2 和 key2<key1 的结果都是 false，那么 key1 和 key2 是等价的，所以它们被认为是相等的。换一种方式，等价意味着!(key1<key2)&&!(key2<key1)的运算值为 true。如果我们的函数对象实现了<=，思考一下会发生什么。当 key1 和 key2 相等时，key1<=key2 和 key2<=key1 都为 true，因而表达式!(key1<=key2)&&!(key2<=key1)为 false；这意味着来自于容器的这两个键竟然不相等。事实上，不会出现用相等来判断键的情况。这意味着这个容器将无法进行正常操作。让我们来看一下如何替换比较函数，从而使容器可以正常操作。

4.5.1 greater<T>对象的用法

假设我们已经为先前章节使用的 Name 类实现了 operator>()。在这个类的定义中，成员函数 operator>()的定义如下：

```
bool operator>(const Name& name) const
{
  return second > name.second || (second == name.second && first >
     name.first);
}
```

当然，可以将成员函数的定义放在类的外面：

```
inline Name::bool operator>(const Name& name) const
{
  return second > name.second || (second == name.second && first >
    name.first);
}
```

现在可以用 Name 对象作为 map 的键，将容器中的 pair 对象按降序排列：

```
std::map<Name, size_t, std::greater<Name>> people
         { {Name{"Al", "Bedo"}, 53}, {Name{"Woody", "Leave"}, 33},
           {Name{"Noah", "Lot"}, 43} };
```

这里第三个模板类型参数指定了用来比较键的函数对象的类型。greater<Name>对象使

用>运算符来比较 Name 对象，因为 Name 类实现了 operator>()，所以可以这样做。这三个元素会按照降序排列。如果列出这些元素，它们的排列方式是很明显的，可以像下面这样列出元素：

```
for( const auto& p : people)
  std::cout << p.first << " " << p.second << " \n";
```

基于范围的 for 循环遍历了 people 容器中的所有元素，然后将它们输出：

```
Noah Lot 43
Woody Leave 33
Al Bedo 53
```

4.5.2　用自定义的函数对象来比较元素

如果 map 或 multimap 中的键是指针的话，那么需要定义一个函数来比较它们所指向的对象，否则会比较指针所表示的地址，这并不是我们想要的。如果键是不支持直接进行<或>比较的类型，为了可以在 map 或 multimap 中使用它们，必须为它们定义一个适当的函数对象。处理这两种情况的方式在本质上相同。

假设我们想用堆上生成的对象的指针作为 map 容器的键。下面用指向 string 对象的智能指针来说明这种情况。这个容器的键可以是 unique_ptr<string>类型，在这个示例中我们需要一个含两个 unique_ptr<string>参数的比较函数，这个函数可以比较它的参数所指向的 string 对象。可以用伪函数——函数对象来定义它，这里假设使用了 using std::string：

```
// Compares keys that are unique_ptr<string> objects
class Key_compare
{
public:
  bool operator()(const std::unique_ptr<string>& p1, const std::unique_ptr
    <string>& p2) const
  {
    return *p1 < *p2;
  }
};
```

可以用 Key_compare 类型作为 map 容器来比较键的函数对象的类型：

```
std::map<std::unique_ptr<string>, std::string, Key_compare> phonebook;
```

第三个 map 模板参数指定了用来比较元素的函数对象的对象，因为这个类型参数指定了默认的值 less<T>，所以我们需要指定我们自己定义的函数对象。map 中的元素是 pair 对象，它封装了一个指向 string 对象的智能指针，string 对象用来保存人名和电话。对于这个 map，我们不能使用初始化列表，因为初始化列表包含了副本，而 unique_ptr 对象是不能被复制的。我们至少有两种向容器中添加元素的方式：

```
phonebook.emplace(std::make_unique<string>("Fred"), "914 626 7897");
```

```
phonebook.insert(std::make_pair(std::make_unique<string>("Lily"), "212
    896 4337"));
```

第一条语句在容器的适当位置直接生成了一个 pair 对象。这个 pair 对象的构造函数可以移动这里指定的参数，所以可以无条件地复制它们。第二条语句调用了容器的成员函数 insert()，它会将参数元素移到容器中。

可以按如下所示列出 phonebook 容器中的元素：

```
for(const auto& p: phonebook)
  std::cout << *p.first << " " << p.second << std::endl;
```

基于范围的 for 循环遍历了 map 中的所有元素，map 中的元素都是 pair 对象。每一个 pair 对象的成员变量 first 都是 unique 指针，所以不得不通过解引用来访问它们所指向的 string 对象。如果使用迭代器来访问元素，语法略有不同：

```
for(auto iter = std::begin(phonebook); iter != std::end(phonebook); ++iter)
  std::cout << *iter->first << " " << iter->second << std::endl;
```

它和之前的循环有相同的输出，但使用的是迭代器。可以用->运算符来访问 pair 对象的成员。因为这样定义了伪函数 Key_compare，所以容器中的元素会被升序排列。

4.6 哈希

如果在容器中保存对象及其关联的键，并且不用键来决定键/对象对的顺序，那就必须对键值采用其他方式来确定元素在内存中的位置。如果使用像 string 这样的对象作为键，就会遇到一些问题，可能的变量的数目是巨大的。具有 10 个字符的字母字符串可能的个数是 26^{10}，换句话说就是 2.6×10^{10} 个字符串。这个索引范围没有多大用处。我们需要一种机制来将它变为可接受的范围；而且理想情况下，这个机制可以为每个键生成唯一的值。这也是哈希需要做的事情之一。

哈希是用给定范围的基本类型的数据项，或者用像 string 这样的对象，生成整数值的过程。哈希产生的值叫作哈希值或哈希码，它们通常被用在容器中，用来确定表中对象的位置。像前面所说的那样，理想情况下，每个对象应该产生唯一的哈希值，但这一般是不可能的。当不同键值的个数大于可能的哈希值个数时，显然就会出现上面所说的这种情况，我们早晚会得到重复的哈希值。重复的哈希值也叫作碰撞。用哈希值来确定元素位置的容器必须处理不同键产生重复哈希值的情况，后面在无序 map 容器的场景下，会解释它们是如何处理这种情况的。

哈希不仅可以在容器中保存对象，它也被应用到很多其他地方，例如密码和加密数据的安全系统中，密码识别有时也包含哈希。在系统中保存明文密码是有很大风险的。保存密码的哈希值要比保存明文密码更安全，更能防范黑客。得到哈希值的黑客需要将哈希值转换为对他们有用的原始密码——这是一个不可能完成的任务。因此 STL 提供的对不同类型数据哈希的能力不仅可以用在关联容器上，也可以被用在更加广阔的场景中。

虽然理解容器的哈希机制没在必要，但是这能让我们对它们能做些什么有一个基本的了解。哈希算法有很多，但却没有可以通用的。为某个场景确定合适的哈希算法并不总是简单。通常都需要对数据分割后再计算。这可能是最简单的处理键的算法了，不管什么类型的键，都会作为数值处理。所以哈希值可能是表达式 k%m 产生的。显然，这种方法最多允许有 m 个不同的哈希值，值的范围为 0 到 m-1。可以很容易地看到哪里会产生重复的哈希值。值为 k+m、k+2*m 的键会有重复的哈希值，m 值的选择对于减少重复哈希值的出现至关重要，而且可以保证值是均匀分布的。如果 m 是 2 的幂，也就是 2^n，哈希值的最小位为 k 的 n 位。这显然不是一个好的结果，因为 k 的大多数位都没有影响到哈希值；理想情况下，键的所有数位应该都可以影响哈希结果。m 通常是一个质数，因为它可以使哈希值更加均匀地分布在这个范围内。

另一种更好的计算哈希值的方式是选择一个常量 a，将它和 k 相乘，用 a*k 除以整数 m 来计算它的余数，然后从(a*k)%m 的结果中选择一个长度值作为哈希值。显然 a 和 m 的选择是非常重要的。对于 32 位的计算机来说，m 通常选为 2^{32}。乘数 a 是和 m 相近的质数，这就意味着 a 和 m 除了 1 之外没有其他的公共因子。此外，a 的二进制表示中头部和尾部不能为 0，否则会和其他头部有 0 或尾部有 0 的键值产生碰撞。基于这些原因，这个算法也被叫作乘法哈希。

也有几个专门哈希字符串的算法。其中一个将字符串看作一定个数的单词，使用像乘法算法这样的方法来计算第一个单词的哈希值，然后加上下一个单词，再计算它的哈希值，重复这个过程，直到计算出所有单词最后的哈希值。幸运的是，STL 已经为哈希提供了相当多的帮助，这也是下一节的主题。

生成哈希值的函数

functional 头文件中定义了无序关联容器使用的特例化 hash<K>模板。hash<K>模板定义了可以从 K 类型的对象生成哈希值的函数对象的类型。hask<K>实例的成员函数 operator()()接受 K 类型的单个参数，然后返回 size_t 类型的哈希值。对于基本类型和指针类型，也定义了特例化的 hash<K>模板。

hash<K>模板专用的算法取决于实现，但是如果它们遵循 C++14 标准的话，需要满足一些具体的要求。这些要求如下：

- 它们不能抛出异常
- 它们对于相等的键必须产生相等的哈希值
- 对于不相等的键产生碰撞的可能性必须最小接近 size_t 最大值的倒数

注意，相等键生成相等的哈希值只适用于单次执行。这也就意味着，在不同的场合允许给定的键可以生成不同的哈希值。这就使我们可以在哈希算法中使用随机数，当对密码进行哈希时，这是我们所希望使用的。注意，C++14 为了保持一致性并没有排除给定类型的键的哈希值等同于键的可能。在无序关联容器中，用哈希函数哈希整数值可能就是这种情况。

下面是一个用 hash<K>生成整数的哈希值的示例：

```
std::hash<int> hash_int;                          // Function object to hash int
std::vector<int> n {-5, -2, 2, 5, 10};
std::transform(std::begin(n), std::end(n), std::ostream_iterator<size_t>
  (std::cout, " "), hash_int);
```

这里使用 transform()算法来哈希 vector 中的元素。transform()参数中的前两个迭代器指定了被操作元素的范围，第三个参数是一个指定输出地址的迭代器，这里是一个 ostream 迭代器，最后一个参数是应用到范围元素上的函数对象 hash<int>。在一笔者的系统上，输出结果为：

```
554121069 2388331168 3958272823 3132668352 1833987007
```

在你的 C++编译器和库中，可能会产生不同的哈希值，所有的哈希值都是这样。下面是一个哈希浮点数值的示例：

```
std::hash<double> hash_double;
std::vector<double> x {3.14, -2.71828, 99.0, 1.61803399, 6.62606957E-34};
std::transform(std::begin(x), std::end(x),
               std::ostream_iterator<size_t>(std::cout, " "), hash_double);
```

在笔者系统上的输出结果为：

```
4023697370 332724328 2014146765 3488612130 3968187275
```

指针也很容易哈希：

```
std::hash<Box*> hash_box;                 // Box class as in Chapter 2
Box box{1, 2, 3};
std::cout << "Hash value = " << hash_box(&box)
          << std::endl;                   // Hash value = 2916986638 for me
```

可以用相同的函数对象来哈希智能指针：

```
std::hash<Box*> hash_box; // Box class as in Chapter 2
auto upbox = std::make_unique<Box>(1, 2, 3);
std::cout << "Hash value = " << hash_box(upbox.get())
          << std::endl;                   // Hash value = 1143026886 for me
```

这里调用 unique_ptr<Box>对象的成员函数 get()来获取保存自由存储区地址的原生指针，然后将它传给哈希函数。这里使用的 hash<K>模板也是 unique_ptr<T>和 shared_ptr<T>对象的特例化模板。例如，可以对 unique_ptr<Box>对象而不是对它所包含的原生指针进行哈希：

```
std::hash<std::unique_ptr<Box>> hash_box;  // Box class as in Chapter 2
auto upbox = std::make_unique<Box>(1, 2, 3);
std::cout << "Hash value = " << hash_box(upbox)
          << std::endl;          // Hash value = 4291053140 for me
```

原生指针和 unique_ptr 的哈希值是相同的。不要被这个误导，考虑到当一个类型的键没有具体的哈希函数时，这种对指针哈希的能力是很有用的。可以对地址进行哈希，而不

是对对象自己。这和指针指向的对象无关。如果想在无序容器中以指向键的指针为键，而不是以键为键，保存一些对象，思考一下会发生什么。指向键的指针的哈希值和原始键的哈希值有很大的不同，因为它们的地址不同，因而无法用它来检索对象。需要一种可以为使用的任何类型的键生成哈希值的方式。如果键的类型是我们所定义的，我们有一个选择，可以用 STL 提供的哈希函数来为我们定义的类的数据成员生成哈希值。

string 头文件中定义了一些特例化的 hash<K>模板，它们会生成一些函数对象，这些函数对象生成表示字符串的对象的哈希值。有 4 个特例化的模板，它们分别对应于字符串类型——string、wstring、u16string 和 u32string。wstring 类型的字符串包含的是 wchar_t 类型的字符；u16string 类型包含的是 char16_t 类型的字符，它是用 UTF-16 编码的 Unicode 字符；u32string 类型包含的是 char32_t 类型的字符，它是用 UTF-32 编码的 Unicode 字符。当然，字符类型——char、wchar_t、char16_t 和 char32_t 都是 C++14 中的基本类型。下面是一个对字符串对象进行哈希的示例：

```
std::hash<std::string> hash_str;
std::string food {"corned beef"};
std::cout << "corned beef hash is " << hash_str(food) << std::endl;
```

这里生成了一个函数对象，它用和前面章节中示例相同的方式来哈希 string 对象。在笔者的系统上，这段代码的输出结果如下：

```
corned beef hash is 3803755380
```

这里对 C 风格字符串的哈希没有具体的规定。使用 const char* 类型的 hash<T>模板会为指针进行特例化。如果想将 C 风格的字符串当作字符序列来哈希生成哈希值，可以先用它生成一个 string 对象，然后使用函数对象 hash<string>。

笔者所展示的代码段生成的哈希值都是非常大的数，这看起来对于确定对象在无序容器中的位置没有什么帮助。有几种方式可以用哈希值确定对象在容器中的位置。一个常见的用法是用哈希值的比特序列作为对象在表或树中的索引。

4.7　unordered_map 容器的用法

unordered_map 包含的是有唯一键的键/值对元素。容器中的元素不是有序的。元素的位置由键的哈希值确定，因而必须有一个适用于键类型的哈希函数。如果用类对象作为键，需要为它定义一个实现了哈希函数的函数对象。如果键是 STL 提供的类型，通过特例化 hash<T>，容器可以生成这种键对应的哈希函数。因为键可以不通过搜索就访问无序 map 中的对象，所以可以很快检索出无序 map 中的元素。迭代遍历无序 map 中的元素序列的速度一般没有有序 map 快，因此在某个应用中选择何种容器取决于想如何访问容器中的元素。

unordered_map 容器中元素的组织方式和 map 有很大的不同，元素的内部组织方式取决于 C++实现。一般情况下，元素被保存在哈希表中，这个表中的条目被称为格子，每个格子可以包含几个元素。一个给定的哈希值会选择特定的格子，因为哈希值可能的个数几

乎可以肯定会大于格子的个数，两个不同的哈希值可能会映射到同一个格子上。因此，因为不同键会产生相同的哈希值，所以会产生碰撞，而且两个不同的哈希值选择相同的格子也会导致碰撞的产生。

下面的一些参数可以影响元素存储的管理：

- 容器中格子的个数有一个默认值，但也可以定指定初始个数。
- 载入因子是每个格子平均保存的元素的个数。这个值等于容器中元素个数除以格子的个数。

最大载入因子，默认是 1.0，但也可以修改。这是载入因子的上限。当容器达到最大载入因子时，容器会为格子分配更多的空间，这通常也会对容器中的元素重新进行哈希。

任何时候都不要将单个格子中的最大元素个数和最大载入因子混淆。假设有一个容器，它有 8 个格子，前两个格子中各有 3 个元素，剩下的格子都为空。那么这时候的最大载入因子为 6/8，也就是 0.75，小于默认的最大载入因子 1.0，所以这没有什么问题。

图 4-3 展示了 unordered_map 的基本结构图。

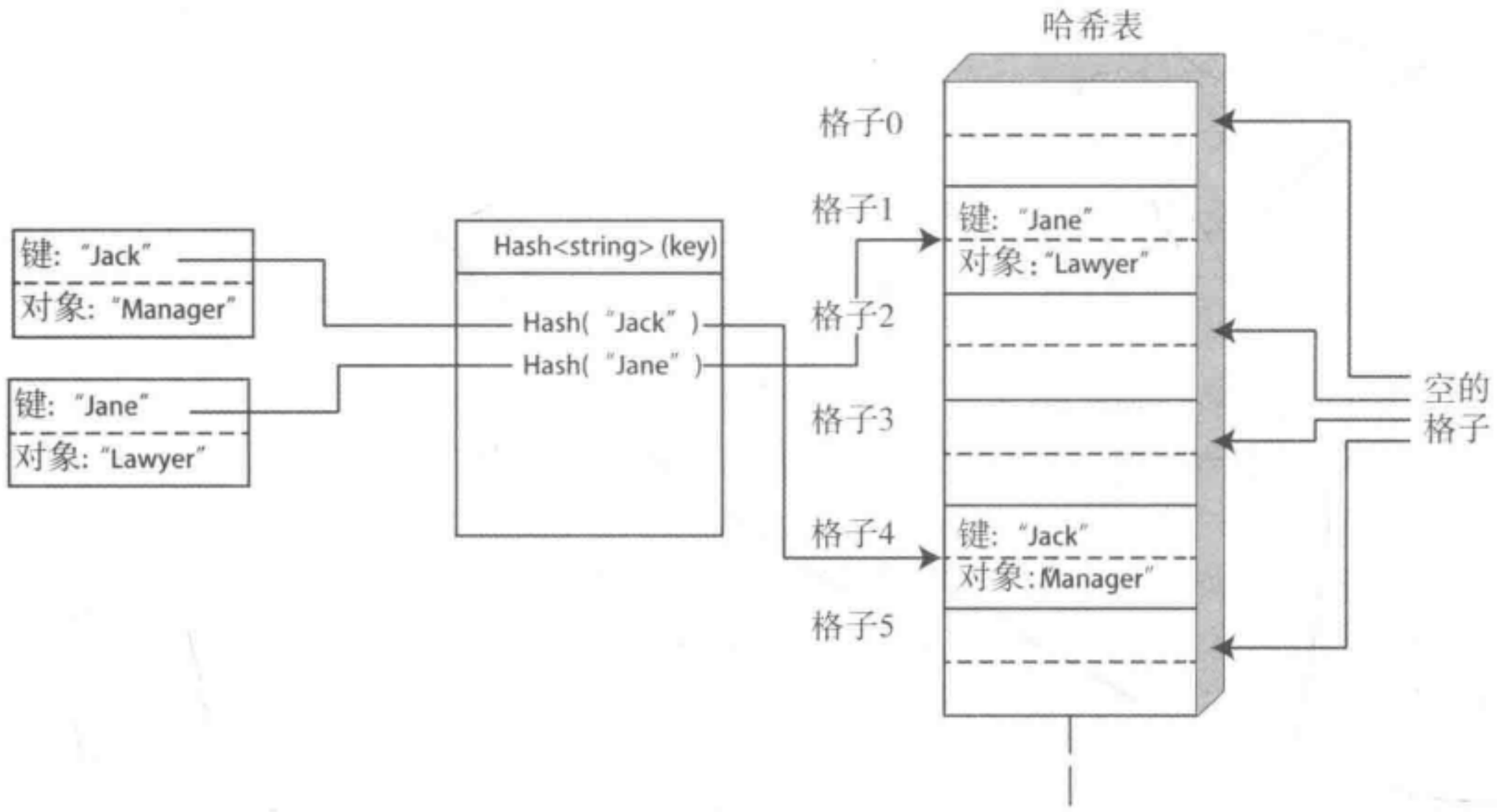


图 4-3　unordered_map 中的数据

为了简单起见，图 4-3 中的每个格子只有一个元素。可以使用从 0 开始的索引来访问格子。

组织格子的方式有很多。一种方式是将格子看作类似 vector 的序列，然后在哈希表中保存序列的地址。另一种方式是将格子定义为链表，在哈希表中保存根节点。具体使用哪种方法取决于我们的实现。

unordered_map 必须能够比较元素是否相等。当容器中有相同的键时，对从包含多个元素的格子中检索到的元素进行确认和选择很有必要。容器默认会使用定义在 functional 头文件中的 equal_to<K>模板。它会用==运算符来比较元素，所以当键相等时，容器会认为它们是相同的，这一点和 map 容器不同，map 容器使用的是等价。如果使用的键是没有实现 operator==()的类类型，那就必须提供一个函数对象来比较键。

4.7.1　生成和管理 unordered_map 容器

生成 unordered_map 容器和生成 map 一样简单，只要可以用 hash<K>的实例哈希 k 类型的键，而且必须能够用==运算符来比较键。下面展示了如何定义和初始化 unordered_map：

```
    std::unordered_map<std::string, size_t> people {{"Jan", 44}, {"Jim", 33},
{"Joe", 99}}; // Name,age
```

这样就生成了一个包含 pair<string, size_t>元素的容器，并用初始化列表中的元素对它进行了初始化。容器中格子的个数是默认的，它使用 equal_to<string>()对象来判断键是否相等。它会用定义在 string 头文件中的 hash<string>来对 string 进行哈希。如果没有提供初始值，默认的构造函数会生成一个空容器，它有默认个数的格子。

当我们知道要在容器中保存多少个元素时，可以在构造函数中指定应该分配的格子的个数：

```
    std::unordered_map<std::string, size_t> people {{{"Jan", 44}, {"Jim", 33},
{"Joe", 99}}, 10};
```

这个构造函数有两个参数：初始化列表和需要分配的格子数。

也可以用迭代器定义的一段 pair 对象来生成容器。显然，只要这个范围内的 pair 对象都是要求的类型，那么任何对象源都可以接受。例如：

```
    std::vector<std::pair<string, size_t>> folks {{"Jan", 44}, {"Jim", 33},
{"Joe", 99},{"Dan", 22}, {"Ann", 55}, {"Don", 77}};
    std::unordered_map<string, size_t> neighbors {std::begin(folks),
std::end(folks), 500};
```

folks 是一个包含 pair<string,size_t>类型元素的 vector 容器，然后用它的元素来填充 neighbors 容器。这里为 neighbors 分配了 500 个格子，但也可以省略这个参数，使用默认的格子个数。可以为前面的两个构造函数指定定义哈希函数的函数对象。这个函数对象会分别作为第 1 个构造函数的第 3 个参数，以及第 2 个构造函数的第 4 个参数，所以这时需要为第 2 个构造函数指定格子个数。接下来会展示如何在接收初始化列表的构造函数中指定这个参数。

假如我们想要用定义在 Ex4_01 中的 Name 对象作为键，那就必须为它定义一个哈希函数和一个恒等运算符，扩展后的类的定义如下：

```
class Name
{
  // Private and public members and friends as in Ex4_01...
public:
  size_t hash() const { return std::hash<std::string>()(first+second); }

  bool operator==(const Name& name) const { return first == name.first &&
                  second== name.second; }
};
```

在这个示例中，编译器提供的默认的 operator==()成员函数能够满足我们的要求，但还是想自己定义。成员函数 hash()用函数对象 hash<string>()来哈希 Name 对象的成员 first 和 second 所拼接的字符串。

unordered_map 容器的哈希函数只能接受和键同类型的单个参数，它会返回一个 size_t 类型的哈希值。我们可以定义一个满足这些条件的函数对象的类型，这个类型的函数对象会调用 Name 对象的成员函数 hash()：

```
class Hash_Name
{
public:
  size_t operator()(const Name& name) const { return name.hash(); }
};
```

当生成 unordered_map 容器时，可以用 Hash_Name 对象作为它的比较函数：

```
std::unordered_map<Name, size_t, Hash_Name> people
            {{{{"Ann", "Ounce"}, 25}, {{"Bill", "Bao"}, 46}, {{"Jack",
              "Sprat"}, 77}},
            500,                                  // Bucket count
            Hash_Name()};                         // Hash function for keys
```

这个容器中的元素是 pair<Name, size_t>类型对象。它的构造函数的第一个参数是一个初始化列表，里面定义了三个这种类型的对象。注意括号是如何嵌套的。最内层的括号中包含 Name 构造函数的参数。它上面的一层包含的是 pair<Name,size_t>构造函数的参数。unordered_map 构造函数的第 2 个参数是格子的个数——我们必须指定它，因为我们想使用第 3 个参数，第 3 个参数是用来哈希键的函数对象。Hash_Name 类型的函数对象会作为这个容器的第 3 个模板类型参数。这么做是必要的，因为模板类型参数有一个不同于我们函数对象的类型的默认值。unordered_map 有以元素段为参数的构造函数，它的前两个参数是迭代器，第 3 个参数是格子个数，第 4 个参数是哈希函数。

当需要指定用来比较两个键对象是否相等的函数对象时，必须指定格子个数，函数对象会用键值来生成哈希值。如果我们忽略了 Name 类的成员函数 operator==()，并且假设定义了一个定义了函数对象的类类型 Name_Equal，可以按如下方式在构造函数中指定它：

```
std::unordered_map<Name, size_t, Hash_Name, Name_Equal> people
            {{{{"Ann", "Ounce"}, 25}, {{"Bill", "Bao"}, 46}, {{"Jack",
              "Sprat"}, 77}},
            500,                                  // Bucket count
            Hash_Name(),                          // Hash function for keys
            Name_Equal()};                        // Equality comparison for keys
```

这里有一个额外的模板类型参数和一个额外的构造函数参数，因为参数有默认值，所以这个模板类型参数是必要的。模板参数列表中用来比较键的函数对象同样会用在以初始化列表为参数的构造函数中。

unordered_map 也有移动和拷贝构造函数。显然，可以用它们生成容器的副本，副本

容器的格子个数、哈希函数都和参数容器相同。

4.7.2　调整格子个数

在维持当前载入因子的前提下，如果插入元素数超过了格子可以满足的个数，容器将不得不增加格子的个数。那么为了将元素重新分配到新的格子中，元素会被再次哈希。这时，这个容器当前存在的任何迭代器都会失效。在任何时候都可以调用成员函数 rehash()来改变格子的个数：

```
people.rehash(15); // Make bucket count 15
```

rehash()的参数可以比当前格子数多或少。这条语句会将格子的个数变为 15，只要它不导致当前因子超过最大载入因子。容器中的所有元素都会被重新哈希分配到新的格子中，而且当前所有的迭代器都会失效。如果指定的格子个数导致载入因子超过最大载入因子，那么格子会自动增加来避免超出最大值。

如果确定会增加格子的个数，可以使用 bucket_count()返回的值：

```
people.rehash((5*people.bucket_count())/4); // Increase bucket count by 25%
```

另一种方式是增加最大载入因子，也就是增加每个格子所包含的元素个数：

```
people.max_load_factor(1.2*people.max_load_factor()); // Increase max load
                                                      // factor by 20%
```

为了改变最大载入因子，可以以新的最大载入因子为参数调用容器的 max_load_factor()；如果无参数地调用这个函数，它会返回当前的最大载入因子，可以用它来设置新的值。

可以发现，调用 unordered_map 的成员函数 load_factor()时返回的当前载入因子是一个浮点值：

```
float lf = people.load_factor();
```

也可以选择设置格子的个数，使它们在容纳给定个数的元素的同时将负载因子维持在最大数之内：

```
size_t max_element_count {100};
people.reserve(max_element_count);
```

这里设置了格子的个数，使它可以容纳 100 个元素而不超过最大载入因子的限制。这会导致容器中的内容被重新哈希，从而使所有的当前迭代器失效。当然，也可以不考虑载入因子和格子个数来生成和使用 unordered_map 容器。容器自己会处理这些事情。对于现实世界中的应用来说，性能是很重要的，而容器就是影响性能的一个重要因素。当每个格子中的元素不超过一个时，访问速度是最快的，但实际上这是不现实的，因为这需要很多内存而且会有很多的空格子。增大最大载入因子可以使每个格子容纳更多的元素，从而使格子的总数越少。所以从内存使用上来说，这是最有效率的。然而，每个格子中的元素越多，会导致访问元素的速度越慢。这需要根据每个程序的具体情况来选择。或许最重要的

事是要避免反复哈希容器的内容。如果可以预估出要保存的元素的个数，就可以设定格子的个数或者设定合适的载入因子，从而减少再次哈希的可能。

4.7.3 插入元素

unordered_map 容器的成员函数 insert()提供的能力和 map 容器的这个函数相同。可以通过复制或移动来插入一个元素，可以使用也可以不使用提示符来指明插入的位置。可以插入初始化列表中指定的元素或由两个迭代器指定范围内的元素。下面有一些示例——先看第一种情况：

```
std::unordered_map<std::string, size_t> people { {"Jim", 33}, {"Joe", 99}};
// Name,age
std::cout << "people container has " << people.bucket_count()
          << " buckets.\n"; // 8 buckets for me
auto pr = people.insert(std::pair<string,size_t> {"Jan", 44});
// Move insert
std::cout << "Element " << (pr.second ? "was" : "was not") << " inserted."
          << std::endl;
```

第一条语句用两个初始元素生成了一个容器，它的格子个数是默认的。第二条语句调用 people 的成员函数 bucket_count()来获取格子个数；注释中展示的是在笔者系统上执行代码后返回的值，但在你的系统上可能会有些不同。这个 insert()调用是一个有右值引用参数的版本，所以 pair 对象会被移到容器中。这个函数返回了一个 pai 对象，它的成员变量 first 是一个迭代器，它指向插入的新元素；如果元素没有被插入，它指向的是阻止插入的元素。Pair 的成员变量 second 是一个布尔值，如果对象插入成功，它的值为 true。

看下面这些语句：

```
std::pair<std::string, size_t> Jim {"Jim", 47};
pr = people.insert(Jim);
std::cout << "\nElement " << (pr.second ? "was" : "was not") << " inserted."
          << std::endl;
std::cout << pr.first->first << " is " << pr.first->second << std::endl;
// 33
```

因为参数是左值，所以会调用有 const 引用参数的 insert()版本，如果插入成功，它会将元素复制到容器中。这个插入操作不会成功，因为容器中已经有键值为 string("Jim")的元素，所以最后一条语句输出的年龄为 33。

```
auto count = people.size();
std::pair<std::string, size_t> person {"Joan", 33};
auto iter = people.insert(pr.first, person);
std::cout << "\nElement " << (people.size() > count ? "was" : "was not")
          << " inserted." << std::endl;
```

这里，insert()的第一个参数是一个迭代器，它是上一个 insert()调用返回的 pair 对象的成员变量 first，用来作为标识元素插入位置的指示符；容器不一定会使用这个指示符。insert()

的第二个参数是待插入的元素。这个版本的 insert()函数不会返回一个 pair 对象，但会返回一个指向插入元素或阻止插入操作的元素的迭代器。这段代码中使用容器的成员函数 size()返回的元素个数来判断元素是否插入成功。

也可以插入初始化列表中的内容：

```
people.insert({{"Bill", 21}, {"Ben", 22}});
// Inserts the two elements in the list
```

这个版本的 insert()不会有返回值，可以插入元素段的 insert()版本也没有返回值：

```
std::unordered_map<std::string, size_t> folks; // Empty container
folks.insert(std::begin(people), std::end(people));
// Insert copies of all people elements
```

迭代器所定义的来自于 people 容器的元素和 folks 中的元素是同种类型，people 可以是任意类型的容器，只要它的元素类型符合 folks 的要求。

可以调用 unordered_map 容器的成员函数 emplace()或 emplace_hint()在容器的适当位置生成元素。例如：

```
auto pr = people.emplace("Sue", 64); // returns pair<iterator, bool>
auto iter = people.emplace_hint(pr.first, "Sid", 67); // Returns iterator
people.emplace_hint(iter, std::make_pair("Sam", 59));
// Uses converting pair<string, size_t>
```

emplace()可以用参数在容器的适当位置生成对象，它会返回一个包含迭代器和布尔值的 pair 对象，这和 insert()返回的 pair 对象有同样的意义。emplace_hint()的第一个参数是一个作为提示符的迭代器，后面的参数被用来生成元素，它返回的迭代器指向被插入元素或阻止插入的元素。

为了可以用 unordered_map 对象的内容替换这个容器的内容，unordered_map 容器实现了赋值运算符：

```
folks = people; // Replace folks elements by people elements
```

显然，参数所包含的元素的类型必须和当前容器相同。

4.7.4　访问元素

对于 unordered_map，可以在下标运算符中使用键来获取它所对应对象的引用。例如：

```
people["Jim"] = 22;                 // Set Jim's age to 22;
people["May"] = people["Jim"];      // Set May's age to Jim's
++people["Joe"];                    // Increment Joe's age
people["Kit"] = people["Joe"];      // Set Kit's age to Joe's
```

这和 map 容器的操作是一样的。在下标中使用不存在的键时，会以这个键为新键生成一个新的元素，新元素中对象的值是默认的。如果容器中不存在"Kit"，上面最后一条语句

会生成一个以"Kit"为键、年龄值为 0 的元素；最后"Joe"所关联的对象会被复制到"Kit"。

成员函数 at()会返回参数所关联对象的引用，如果键不存在，会抛出一个 out_of_range 异常。所以当我们不想生成有默认对象的元素时，应该选择使用at()而不是下标运算符。你可能已经发现成员函数 find()和 equal_range()的工作方式和之前描述的 map 是一样的。

unordered_map 的迭代器是可以使用的，因此可以用基于范围的 for 循环来访问它的元素，例如：

```
for(const auto& person : people)
  std::cout << person.first << " is " << person.second << std::endl;
```

这样就可以列出 people 容器中的全部元素。

4.7.5 移除元素

可以调用 unordered_map 的成员函数 erase()来移除元素。参数可以是标识元素的一个键或是指向它的一个迭代器。当参数是键时，erase()会返回一个整数，它是移除元素的个数，所以 0 表示没有找到匹配的元素。当参数是迭代器时，返回的迭代器指向被移除元素后的元素。下面是一些示例：

```
auto n = people.erase("Jim");       // Returns 0 if key not found
auto iter = people.find("May");     // Returns end iterator if key not found
if(iter != people.end())
  iter = people.erase(iter);        // Returns iterator for element after "May"
```

也可以移除指定的一个元素序列。例如：

```
// Remove all except 1st and last
auto iter = people.erase(++std::begin(people), --std::end(people));
```

返回的迭代器指向被移除的最后一个元素的下一个位置。

成员函数 clear()会移除所有的元素。当容器中没有元素时，成员函数empty()返回 true。

4.7.6 访问格子

可以访问 unordered_map 的个别格子及其包含的元素。可以用这个容器的成员函数 begin()和 end()的重载版本来做到这一点，它们可以返回容器元素的迭代器。格子的索引从 0 开始，可以通过将索引值传给容器的成员函数 begin()来获取给定位置的格子中第一个元素的迭代器。例如：

```
auto iter = people.begin(1); // Returns an iterator for the 2nd bucket
```

将索引值传给容器的成员函数 cbegin()会返回一个 const 迭代器，它指向位于索引位置的格子中的第一个元素。这个容器的成员函数 end()和 cend()也有这样的版本，它们接受一个索引值，分别返回一个迭代器和一个 const 迭代器，它们指向位于指定位置的格子中的最后一个元素的下一个位置。可以输出特定格子中的元素——一个格子列表，也就是说，

需要使用循环：

```
size_t index{1};
std::cout << "The elements in bucket[" << index << "] are:\n";
for(auto iter = people.begin(index); iter != people.end(index); ++iter)
  std::cout << iter->first << " is " << iter->second << std::endl;
```

我们已经看到 unordered_map 的成员函数 bucket_count()返回的格子个数。bucket_size()可以返回参数指定的格子中的元素个数。bucket()返回的是格子的索引值，包含和传入的参数键匹配的元素。可以用不同的方式来组合使用它们。例如：

```
string key {"May"};
if(people.find(key) != std::end(people))
  std:: cout << "The number of elements in the bucket containing " << key
<< " is "<< people.bucket_size(people.bucket(key)) << std::endl;
```

bucket_size()的参数是 bucket()返回的索引值。当键在容器中时，这段代码才会执行输出语句。输出记录了包含键的格子中的元素个数。

下面有一个示例，可以让我们深入了解在添加元素时，我们系统中的 unordered_map 是如何做的：

```
// Ex4_06.cpp
// Analyzing how and when the number of buckets in an unordered_map container
//increases
#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
#include <string>                      // For string class
#include <unordered_map>               // For unordered_map container
#include <vector>                      // For vector container
#include <algorithm>                   // For max_element() algorithm

using std::string;
using std::unordered_map;

// Outputs number of elements in each bucket
void list_bucket_counts(const std::vector<size_t>& counts)
{
  for(size_t i {}; i < counts.size(); ++i)
  {
    std::cout << "bucket[" << std::setw(2) << i << "] = " << counts[i] << " ";
    if((i + 1) % 6 == 0) std::cout << '\n';
  }
  std::cout << std::endl;
}
int main()
{
  unordered_map<string, size_t> people;
  float mlf {people.max_load_factor()};     // Current maximum load \factor
  size_t n_buckets {people.bucket_count()};   // Number of buckets in container
```

```
std::vector<size_t> bucket_counts (n_buckets); // Records number of elements
                                               //per bucket
string name {"Name"};                          // Key - with value appended
size_t value {};                               // Element value
size_t max_count {8192};                 // Maximum number of elements to insert
auto lf = people.load_factor();                // Current load factor
bool rehash {false};                           // Records when rehash occurs
while(mlf <= 1.5f)                       // Loop until max load factor is 1.5
{
  std::cout << "\n\n***************New Container***************"
            << "\nNumber of buckets: " << n_buckets
            << " Maximum load factor: " << mlf << std::endl;
// Insert max elements in container
for(size_t n_elements {}; n_elements < max_count; ++n_elements)
{
  lf = people.load_factor();     // Record load factor before insert
  people.emplace("name" + std::to_string(++value), value);
  auto new_count = people.bucket_count(); // Current bucket count
  if(new_count > n_buckets)      // If bucket count increases...
  {                                                // Output info
    std::cout << "\nBucket count increased to " << new_count
              << ". Load factor was " << lf << " and is now "
              << people.load_factor()
              << "\nMaximum elements in a bucket was "
              << *std::max_element(std::begin(bucket_counts),
                 std::end (bucket_counts))
            << std::endl;
  if(n_buckets <= 64)
  {
    std::cout << "Bucket counts before increase were: " << std::endl;
    list_bucket_counts(bucket_counts);
  }

  n_buckets = new_count;                       // Update bucket count
  bucket_counts = std::vector<size_t>(n_buckets); // New vector for counts
  rehash = true;                               // Record rehash occurred
}
// Record current bucket counts
for(size_t i {}; i < n_buckets; ++i)
  bucket_counts[i] = people.bucket_size(i);

if(rehash)                                     // If the container was rehashed...
{                                              // ...output info
  rehash = false;                              // Reset rehash indicator

  std::cout << "\nRehashed container. Bucket count is " << n_buckets
            << ". Element count is " << people.size()
            << "\nMaximum element count in a bucket is now "
            << *std::max_element(std::begin(bucket_counts), std::end
              (bucket_counts))
```

```
                << std::endl;
      if(n_buckets <= 64)                    // If no more than 64 buckets...
      {
        std::cout << "\nBucket counts after rehash are:\n";
        list_bucket_counts(bucket_counts);
      }
    }
  }
  std::cout << "Final state for this container is:\n"
            << "Bucket count: " << people.bucket_count()
            << " Element count: " << people.size()
            << " Maximum element count in a bucket: "
            << *std::max_element(std::begin(bucket_counts),
               std::end(bucket_counts))
            << std::endl;
    value = 1; // Reset key suffix
    people = unordered_map<string, size_t>(); // New empty container
    n_buckets = people.bucket_count();
    bucket_counts = std::vector < size_t >(n_buckets); // New vector for
                                                       // bucket counts
    mlf += 0.25f; // Increase max load factor...
    people.max_load_factor(mlf); // ...and set for container
  }
}
```

上述程序会记录元素个数增加时的载入因子和格子个数。通过这种方式，可以了解容器在什么条件下是以何种方式增加格子的。因为这段代码的执行会花费相当长的时间，所以我们需要耐心等待一下。如果觉得花费的时间太长了，可以减小变量 max 的值。

这个示例以一个空的 unordered_map 容器开始，然后插入 max 所指定的有限个新元素。然后通过在参数"name"的后面追加 value 递增后 to_string()返回的字符串来构造唯一的键。to_string()函数定义在 string 头文件中，可以将任意数值类型转换为 string 对象。

每个格子中的元素个数记录在一个 vector 容器中。外层的 while 循环会一直进行下去，只要最大载入因子小于等于 1.5。嵌套的 for 循环会向 unordered_map 容器中插入 max_count 个元素。无论什么时候格子个数改变，都会调用辅助函数 list_bucket_counts()来输出每个格子中的元素个数。为了避免大量输出导致无法管理，只输出 64 个或更少的格子。当插入 max_count 个元素后，会使用更大的载入因子来生成 unordered_map，对这个新容器继续重复内会循环。这样就展示了最大载入因子如何影响格子个数的增加程度，从而导致元素被重新哈希。

笔者不想在自己的系统上复现这个输出，因为会占用太多的空间，但可以简单概括下发生了什么。默认的格子数是 8。在添加了 8 个元素后，格子数从 8 增加到了 64，这是一个非常大的变化。当格子的最大元素个数是 2，但只有一个格子包含两个元素时；元素的总个数是 9。输出显示当载入因子接近 1.0 时，会触发格子个数的增加。在笔者的系统上，格子个数下一次会乘以因子 8，从 64 到 512。格子个数的因子会缓慢增加到 2，所以格子个数的序列是 8、64、512、1024、2048、4096、8192。随着格子个数的增加，观察有多少

空格子是很有意思的。在笔者的系统上，所有格子的最大元素数目是 8，一点儿也不令人惊讶的是它也是最大的载入因子。在载入因子最大为 1.5 时，从一个格子中得到了 7 个元素。每次格子个数的增加都会导致容器中的所有元素被重新哈希分配到新的位置。可以很容易地对程序做一下调整，使它能够输出在格子增加前后的元素，这样就能够知道元素是如何被移动的。这显然会涉及相当大的开销，所以在保存的元素增加时，开始得到正确的格子数就变得更重要。

从系统上的输出可以了解到容器是如何将原始哈希值映射到格子的索引上的。格子数总是为 2 的幂。这样就允许格子的索引是来自于原始哈希值的固定位序列——8 个格子需要 3 位，64 个需要 6 位，512 个需要 9 位，以此类推。这就使获取格子的索引变得简单和快速。也就解释了为什么在格子数增加后，需要重新对元素进行哈希。从给定的哈希值中取 6 位要比取 3 位更能表示不同的索引值，所以一个给定的原始哈希值可能会被映射到不同的格子中。

4.8 unordered_multimap 容器的用法

unordered_multimap 是一个允许有重复键的无序 map。因此，它支持的操作实际上和 unordered_map 容器是相同的，为了处理多个重复键所做的添加和更改除外。后面会对这些差别做些讨论。生成 unordered_multimap 的方式和 unordered_map 相同。例如：

```
std::unordered_multimap<std::string, size_t> people {{"Jim", 33}, {"Joe",
99}};
```

可以使用 insert()、emplace()、emplace_hint()来添加新元素，这和 unordered_multimap 相同，只要参数和容器中的元素类型一致。这些成员函数都会返回一个指向容器中新元素的迭代器；在使用 insert()和 emplace()的情况下，unordered_multimap 和 unordered_map 有些不同，unordered_multimap 的这两个函数会返回一个 pair 对象，它用来说明插入是否成功。如果不成功，也是一个迭代器。例如：

```
auto iter = people.emplace("Jan", 45);
people.insert({"Jan", 44});
people.emplace_hint(iter, "Jan", 46);
```

第 3 条语句使用了第 1 条语句返回的迭代器作为插入元素的提示符。这个提示符有时会被容器或我们的实现所忽略。

unordered_map 支持的成员函数 at()和 operator[]()对于 unordered_multimap 来说并不可用，因为潜在的重复键。唯一的选择是使用 find()和 equal_range()来访问元素。find()总会返回它所找到的第一个元素的迭代器，如果找不到这个键，会返回一个结束迭代器。可以以键为参数调用 count()来发现容器中给定键的元素个数。下面展示实际用法：

```
std::string key{"Jan"};
auto n = people.count(key);          // Number of elements stored with key
  if(n == 1)
```

```
    std::cout << key << " is " << people.find(key)->second << std::endl;
  else if(n > 1)
  {
   auto pr = people.equal_range(key);   // pair of begin & end iterators returned
   while(pr.first != pr.second)
   {
    std::cout << key << " is " << pr.first->second << std::endl;
    ++pr.first;                          // Increment begin iterator
   }
}
```

当容器中只有一个 key 时，可以用 find()来访问这个元素。如果超过一个，可以用 equal_range()来访问这段元素。当然，在这两种情况下都可以使用 equal_range()。

让我们来看一个使用 unordered_multimap 的示例。在这个示例中也会展示一些定义函数模板来使用容器的方式。这个程序会实现一个电话簿，可以用姓名或名称来查找电话号码。此处用一个 pair 对象来封装名和姓，用一个元组来记录区号、交换码、电话号码，它们都是以 string 对象的形式保存的。使用下面的 using 简化代码：

```
using std::string;
using std::unordered_multimap;
using Name = std::pair<string, string>;
using Phone = std::tuple<string, string, string>;
```

电话号码可以用三个整数来表示，但这样组合起来更像编码而不是号码。电话号码中的每个元素都有固定个数的数字，有一些位数的组合是不允许的。如果想增加对号码检查的功能，显然用 string 对象可以使号码位数或区号的检查更简单。这里并没有包含这个功能，因为这会使本书的编码量大大增加。

为了从 istream 对象读取电话号码，对>>运算符进行了重载。函数如下：

```
inline std::istream& operator>>(std::istream& in, Phone& phone)
{
  string area_code {}, exchange {}, number {};
  in >> std::ws >> area_code >> exchange >> number;
  phone = std::make_tuple(area_code, exchange, number);
  return in;
}
```

phone 是元组模板类型。这里 make_tuple()使用从 in 读出的局部变量的值生成了一个 phone 对象。

我们可以为 Name 对象做相同的事：

```
inline std::istream& operator>>(std::istream& in, Name& name)
{
  in >> std::ws >> name.first >> name.second;
  return in;
}
```

这里会丢弃从 in 读入的任何前置空格，然后读出两个 name 字符串作为 pair 对象的成员。当然，我们也需要输出功能。这里定义了 operator<<()来为 phone 对象提供输出：

```
inline std::ostream& operator<<(std::ostream& out, const Phone& phone)
{
  std::string area_code {}, exchange {}, number {};
  std::tie(area_code, exchange, number) = phone;
  out << area_code << " " << exchange << " " << number;
  return out;
}
```

这里使用函数模板 tie<>()生了一个包含三个局部变量引用的元组。然后为了将 phone 的成员变量的值保存到局部变量中，将 phone 赋值给 tie<>()生成的元组，后面会输出这些局部变量。或者，使用函数模板 get<>()来访问 phone 成员变量的值。这种方法显然更好，因为能够避免上面出现的 string 对象的副本，但这里是为了展示 tie<>()函数的使用。

为 Name 对象重载<<是很简单的：

```
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{
  out << name.first << " " << name.second;
  return out;
}
```

所有的这些 I/O 函数都是内联的，所以把它们放在叫作 Record_IO.h 的头文件中。文件的开头是#include 和 using：

```
#include <string>                    // For string class
#include <istream>                   // For istream class
#include <ostream>                   // For ostream class
#include <utility>                   // For pair type
#include <tuple>                     // For tuple type

using Name = std::pair <std::string, std::string>;
using Phone = std::tuple <std::string, std::string, std::string>;
```

这个程序会使用两个关联容器——其中一个以名称作为键，另一个以电话号码为键，所以它们都会包含相同的基本信息，但它们的访问方式不同。当然有更有效率的方式来做这些，但这里我们是为了尝试 unordered_multimap 容器的使用。main()中定义的容器如下：

```
unordered_multimap<Name, Phone, NameHash> by_name {8, NameHash()};
unordered_multimap<Phone, Name, PhoneHash> by_number {8, PhoneHash()};
```

这里并没有为 pair 或 tuple 对象提供默认的哈希能力，所以我们不得不自己定义它们。这里它们分别是 NameHash 和 PhoneHash 类型的函数对象。构造函数中哈希函数对象之前的参数是格子的个数，所以这里的格子数是指定的。它们等于系统的默认值。

把这两个哈希函数类型的定义放在同一个头文件中，名为 Hash_Function_Objects.h，文件的开头是下面这些代码：

```
#include <string> // For string class
#include <utility> // For pair type
#include <tuple> // For tuple type

using Name = std::pair<std::string, std::string>;
using Phone = std::tuple < std::string, std::string, std::string>;
```

按如下方式定义 PhoneHash 类型：

```
class PhoneHash
{
public:
  size_t operator()(const Phone& phone) const
  {
    return std::hash<std::string>()(std::get<0>(phone)+std::get<1>
      (phone)+std::get<2>(phone));
  }
};
```

通过用定义在 string 头文件中的 hash<string>()模板的特例化来处理电话号码中三个元素串联后的结果，对应地生成哈希值。下面以相似的方式定义了 HashName 类型：

```
class NameHash
{
public:
  size_t operator()(const Name& name) const
  {
    return std::hash<std::string>()(name.first + name.second);
  }
};
```

将输出操作打包放在单独的函数中：

```
void show_operations()
{
  std::cout << "Operations:\n"
            << "A: Add an element.\n"
            << "D: Delete elements.\n"
            << "F: Find elements.\n"
            << "L: List all elements.\n"
            << "Q: Quit the progr.\n\n";
}
```

通过名称或号码列出所有的元素。我们定义了一个函数模板来处理这两种可能：

```
template<typename Container>
void list_elements(const Container& container)
{
  for(const auto& element : container)
   std::cout << element.first << " " << element.second << std::endl;
}
```

当函数被调用时，这个模板会从使用的参数中推断出容器的类型。两个容器包含的元素是 pair 对象。by_name 容器包含的是 pair<Name, Phone>对象类型的元素，by_number 容器包含的是 pair<Phone, Name>对象类型的元素。因为我们已经为 Name 和 Phone 类型重载了 operator<<()，循环体会为 pair 元素的成员变量类型自动选择合适的函数输出。把这个函数模板和 My_Templates.h 头文件后面的模板放在完整的示例中。

通过名称或号码来查找元素的过程本质上是相同的，所以我们也可以为它们定义一个函数模板：

```
template<typename Container>
auto find_elements(const Container& container) ->
          std::pair<typename Container::const_iterator, typename
            Container::const_iterator>
{
  typename Container::key_type key {};
  std::cin >> key;
  auto pr = container.equal_range(key);
  return pr;
}
```

这段代码对应于 C++11 标准。返回类型是依赖容器类型的 pair<>模板类型。这是因为返回的 pair 封装的迭代器类型是特定的容器类型。这就意味着编译器不能处理函数名之前的返回类型，因为容器的类型取决于函数参数的类型，这个类型稍后才用得到。为了能够使 C++11 编译器可以确定返回类型，必须使用尾部返回类型语法。这会允许编译器在处理完函数参数后再处理返回类型。注意，typename 关键字对模板类型的规范是至关重要的，它对于局部变量 key 的规范也至关重要。容器中 key 的类型是由容器类的成员变量 key_type 指定的，所以 key 的类型规范会自动从容器中选择正确的类型。如果需要和键关联的对象的类型，它是由成员 Container::mapped_type 指定的，Container::value_type 指定了容器元素的类型。

C++14 标准为编译器引入了推断函数返回类型的能力，因而函数模板可以这样写：

```
template<typename Container>
auto find_elements(const Container& container)
{
  typename Container::key_type key {};
  std::cin >> key;
  auto pr = container.equal_range(key);
  return pr;
}
```

这里不需要尾部返回类型，因为编译器可以从返回值 pr 中推断出返回类型。

这个操作允许通过搜索名称或号码来查找元素，每种情况都会返回一个 pair 对象，它包含的迭代器定义了一种或另一种类型元素的范围。我们可以另外定义一个函数模板来输出这段元素：

```
template<typename T>
```

```
void list_range(const T& pr)
{
  if(pr.first != pr.second)
  {
    for(auto iter = pr.first; iter != pr.second; ++iter)
      std::cout << " " << iter->first << " " << iter->second << std::endl;
  }
  else
    std::cout << "No records found.\n";
}
```

如果作为参数的 pair 对象的成员变量是相同的，那么范围为空，这种情况下我们只会输出一条消息。为 Name 和 Phone 类型实现的<<运算符函数使这个模板可以正常使用。pair 成员变量的实际类型会自动选择合适的 operator<<()函数。需要注意的是，在已编译的代码中，这些模板不会减少代码量。它们只是提供了一种生成所使用函数的转换机制，提供了如何使用模板的简单演示。

可以从下载的 Ex4_07.cpp 中获取 main()函数，它会包含下面这些代码：

```
// Ex4_07.cpp
#include <iostream>                 // For standard streams
#include <cctype>                   // For toupper()
#include <string>                   // For string class
#include <unordered_map>            // For unordered_map container

#include "Record_IO.h"
#include "My_Templates.h"
#include "Hash_Function_Objects.h"

using std::string;
using std::unordered_multimap;
using Name = std::pair<string, string>;
using Phone = std::tuple<string, string, string>;

// show_operations() definition goes here...

int main()
{
  unordered_multimap<Name, Phone, NameHash> by_name {8, NameHash()};
  unordered_multimap<Phone, Name, PhoneHash> by_number {8, PhoneHash()};

  show_operations();
  char choice {};                   // Operation selection
  Phone number {};                  // Records a number
  Name name {};                     // Records a name

  while(std::toupper(choice) != 'Q')    // Go until you quit...
  {
    std::cout << "Enter a command: ";
    std::cin >> choice;
```

```
switch(std::toupper(choice))
{
case 'A': // Add a record
  std::cout << "Enter first & second names, area code, exchange, "
            << "and number separated by spaces:\n";
  std::cin >> name >> number;
  by_name.emplace(name, number);       // Create in place...
  by_number.emplace(number, name);     // ...in both containers
  break;
case 'D': // Delete records
{
  std::cout << "Enter a name: ";          // Only find by name
  auto pr = find_elements(by_name);
  auto count = std::distance(pr.first, pr.second); // Number of elements
  if(count == 1)
  { // If there's just the one...
   by_number.erase(pr.first->second);  // ...delete from numbers container
   by_name.erase(pr.first);            // ...delete from names container
  }
  else if(count > 1)
  { // There's more than one
   std::cout << "There are " << count << " records for "
             << pr.first->first << ". Delete all(Y or N)? ";
   std::cin >> choice;

   if(std::toupper(choice) == 'Y')
   {
     // Erase records from by_number container first
     for(auto iter = pr.first; iter != pr.second; ++iter)
     {
       by_number.erase(iter->second);
     }
     by_name.erase(pr.first, pr.second);  // Now delete from by_name
   }
  }
}
break;

case 'F':                                      // Find a record
  std::cout << "Find by name(Y or N)? ";
  std::cin >> choice;
  if(std::toupper(choice) == 'Y')
  {
   std::cout << "Enter first name and second name: ";
    list_range(find_elements(by_name));
  }
  else
  {
    std::cout << "Enter area code, exchange, and number separated by spaces: ";
    list_range(find_elements(by_number));
```

```
    }
    break;

  case 'L': // List all records
    std::cout << "List by name(Y or N)? ";
    std::cin >> choice;
    if(std::toupper(choice) == 'Y')
      list_elements(by_name);
    else
      list_elements(by_number);
    break;
  case 'Q':
    break;

  default:
    std::cout << "Invalid command - try again.\n";
  }
 }
}
```

这段代码很好理解。在输入选项后，程序就会执行相应的操作，直到输入'q' 或 'Q'。循环体是一条大的 switch 语句，用来选择合适的操作。

添加元素只涉及每个容器元素的生成，by_name 使用的键/对象值和 by_number 使用的是相反的。

可以用 find_elements()来删除 by_name 容器的元素，为了保证两个容器中内容的同步，需要删除 by_number 容器的对应元素。为了能够从 by_name 容器移除多个元素，需要用定义元素范围的迭代器作为 erase()的参数。如果所有元素的键相同，就可以以这个范围内的第一个元素的键为参数来删除它们，例如：

```
by_name.erase(pr.first->first); // Delete elements with the specified key
```

为了查找操作，find_elements()模板实例返回的 pair 必须直接传给 list_range()模板的实例。编译器会自动保证生成合适的调用。最后，为了可以列出元素，必须用指定的键为参数来调用一个 list_elements()模板的实例，从而输出元素。

下面是输出示例：

```
Operations:
A: Add an element.
D: Delete elements.
F: Find elements.
L: List all elements.
Q: Quit the program.
Enter a command: a

Enter first & second names, area code, exchange, and number separated by spaces:
Bill Bloggs 112 234 4545
Enter a command: a
```

```
Enter first & second names, area code, exchange, and number separated by spaces:
Nell Bloggs
112 234 4545
Enter a command: a
Enter first & second names, area code, exchange, and number separated by spaces:
Bill Bloggs 914 626 7890
Enter a command: a
Enter first & second names, area code, exchange, and number separated by spaces:
Al Capone 312 334 4566
Enter a command: l
List by name(Y or N)? y
Nell Bloggs 112 234 4545
Bill Bloggs 112 234 4545
Bill Bloggs 914 626 7890
Al Capone 312 334 4566
Enter a command: l
List by name(Y or N)? n
112 234 4545 Bill Bloggs
112 234 4545 Nell Bloggs
914 626 7890 Bill Bloggs
312 334 4566 Al Capone
Enter a command: f
Find by name(Y or N)? y
Enter first name and second name: Bill Blogs
No records found.
Enter a command: f
Find by name(Y or N)? y
Enter first name and second name: Bill Bloggs
  Bill Bloggs 112 234 4545
  Bill Bloggs 914 626 7890
Enter a command: f
Find by name(Y or N)? n
Enter area code, exchange, and number separated by spaces: 112 234 4545
  112 234 4545 Bill Bloggs
  112 234 4545 Nell Bloggs
Enter a command: q
```

4.9 本章小结

对于使用键而不是索引值来访问的数据来说，本章所介绍的关联容器是一个强大的工具。它们特别适用于涉及关联数据项——名称和电话号码的应用，例如人名和地址、组件和子组件，或者子组件和零件。通常无序 map 容器提供的访问对象的速度要比有序 map 容器快，但这取决于键的哈希函数在大部分时生成的唯一哈希值。不好的哈希函数会减慢元素的检索速度，因为匹配键需要多次搜索格子。如果对自己的键的哈希函数没有信心，那么最好使用有序 map 容器。

虽然 map 容器为编程提供了很大便利，但总是值得考虑——用保存 pair 对象的序列容

器来代替它是否合理，尤其在我们能够用键对容器进行排序时。这样偶尔也能提供一个比关联容器好的解决方案。

本章重点如下：

- pair<T1,T2>对象封装了两个任意类型的对象。
- tuple<>模板类型的实例可以封装任意个数的不同类型的对象。
- map 容器是以 pair<const K,T>对象的形式来保存键/对象对元素的。
- map<K,T>容器保存的元素有唯一的键值，默认使用<运算符来对键进行排序，所以键类型必须支持<运算符，除非自己提供一个比较函数。
- multimap<K,T>包含的元素和 map 一样，都是有序的，但允许有重复的键。
- 有序关联容器用等价来决定两个键什么时候是相同的。通常只用<或>来比较键，生成比较结果。当键相等时，比较函数会返回 true，从而阻止容器正常工作。
- 哈希是从对象生成名为哈希值的相对唯一整数的过程。通常用哈希来决定元素在无序容器中的存放位置。哈希对于密码学也很重要。
- 使用键生成的哈希值将元素保存在 unordered_map<K,T>容器中。哈希值会选择特定的格子，而且每个格子可以包含几个元素。unordered_map<K,T>中的键必须是唯一的。
- unordered_multimap 容器和 unordered_map 相似，但它允许有重复的键。
- 无序 map 容器默认使用 equal_to<K>来比较键是否相等，所以键类型必须支持==比较和键对象的哈希。

练习

1. 实现一个程序，用来保存每门课程中任意名学生的姓名。这个程序应该支持课程的添加和删除，并可以列出所有的课程。需要可以通过课程名(例如"Biology")来检索一门课程的学生列表。

2. 实现一个程序，用 multimap<K,T>容器来保存每名学生参加的课程，这里的学生可能有重名。元素应该是降序排列的，支持添加和删除学生，列出所有的学生和他们所选的课程，可以通过给定的学生名来检索并展示他所选择的课程。

3. 写一个程序来模拟一个有若干收银台的超市，收银台的个数从键盘输入。用 map 中的一个元素表示一个收银台，键是收银台的 ID，收银台旁边的队列作为关联的对象。顾客应该以随机间隔到达收银台，并且他们所占用的服务时间也是随机的。这个程序应该在一段时间后报告平均和最长队列的长度，这段时间也从键盘输入的。

4. 为 Person 对象定义一个类来保存姓名、地址、电话号码。以姓名为键在 unordered_multimap 容器中保存 Person 对象。提供用姓名来检索地址和电话号码，同时支持以姓名的降序来列出所有的 Person 对象。定义一个 main()程序来展示提供的所有函数。

第 5 章

set 的使用

本章将介绍 set(集合)的使用。集合是一个简单直观的数学概念——具有共同特征的事物的集合。集合在 STL 中有两个概念，它们都涉及一系列的数学思想。集合可以是由两个迭代器定义的范围内的一系列对象，也可以是一种有特殊特征的容器类型。set 容器是关联容器，其中的对象是对象它们自己的键。本章将介绍以下内容：

- STL 提供了哪些 set 容器
- 不同类型的 set 容器的功能
- 可以对 set 容器使用的操作
- 如何创建和使用 set 容器
- 对象集合的运算

5.1 理解 set 容器

除了没有单独的键，set 容器和 map 容器很相似。定义 set 的模板有 4 种，其中两种默认使用 less<T>来对元素排序，另外两种使用哈希值来保存元素。有序 set 的模板定义在 set 头文件中。无序 set 的模板定义在 unordered_set 头文件中。因此有序 set 包含的元素必须支持比较运算，无序 set 中的元素必须支持哈希运算。

定义 set 容器的模板如下：

- set<T>容器保存 T 类型的对象，而且保存的对象是唯一的。其中保存的元素是有序的，默认用 less<T>对象比较。可以用相等、不相等来判断对象是否相同。
- multiset<T>容器和 set<T>容器保存 T 类型对象的方式相同，但它可以保存重复的对象。
- unorderd_set<T>容器保存 T 类型的对象，而且对象是唯一的。元素在容器中的位置由元素的哈希值决定。默认用 equal_to<T>对象来判断元素是否相等。
- unordered_multiset<T>容器保存 T 类型对象的方式和 unorderd_set<T>相同，但它可以保存重复的对象。

从有序和无序关联容器获取的各种迭代器之间有一些区别。我们可以从有序容器得到正向和反向迭代器，但是只能从无序容器得到正向迭代器。

如果之前没有用过 set 容器，你可能不知道如何检索这种容器中的对象，这需要提供一个相同的对象。如果我们已经有这个对象，那为什么还需要再去获取它？对此你也许会感到惊讶，set 容器有很多用途。

和一组事物相关的程序可以用 set 来保存候选数据，它可以确定程序需要的数据集。大学里的班级就是一个示例。每个班都可以用一个包含学生的 set 容器来表示。这里使用 set 容器比较合适，因为同一个班级中不可能有重复的学生；显然，两个学生可以同名，但是它们所表示的人是不同的。可以很容易查看某个学生是否申请了某门课程，也可以查看某个学生申请了哪些课程。

一般来说，当 set 中有大量元素时，在无序 set 上执行的随机插入和检索操作要比有序 set 快。在有 n 个元素的有序 set 中检索元素的时间复杂度是 log n。在无序 set 中检索元素的平均时间复杂度是常量，这和元素的个数无关，尽管实际性能会受元素哈希操作和内部组织效率的影响。

对象在容器中的存放位置取决于有序 set 的比较函数和无序 set 的哈希函数，对于保存同一种对象的不同 set，我们可以使用不同的比较函数或哈希函数。这里有一个简单的示例，用 Person 类来表示公司的员工，这个类会封装一些个人信息。类中应该包括个人 ID、部门、姓名、年龄、地址、性别、电话号码、薪酬等级等信息。然后可以用各种方式对员工进行分类。可以在一个 set 中用工作部门的哈希比较，在另一个 set 中用薪酬等级的哈希比较。这样我们就可以得到特定薪酬等级或特定部门的员工。这些 set 中不会保存重复的 Person 对象。可以在自由存储区创建 Person 对象，然后将它们的智能指针保存在容器中。在本章的后面你会看到一些相关示例。为了简化代码，假定在本章使用了 std::string。

5.2 使用 set<T>容器

通常，set<T>容器内部元素的组织方式和 map<K,T>相同——都是平衡二叉树。请考虑下面这个 set 容器的定义，可以用初始化列表来初始化 set 容器：

```
std::set<int> numbers {8, 7, 6, 5, 4, 3, 2, 1};
```

默认的比较函数是 less<int>，因此容器中的元素会升序排列。内部的二叉树和图 5-1 中所示的类似。执行下面的语句后，容器中的元素变成升序：

```
std::copy( std::begin(numbers), std::end(numbers),
  std::ostream_iterator<int>{std::cout, " "});
```

copy()算法会将前两个参数指定的一段元素复制到第三个参数指定的位置，这里第三个参数是一个输出流迭代器。这条语句会输出一个从 1 至 8 的整数递增序列。

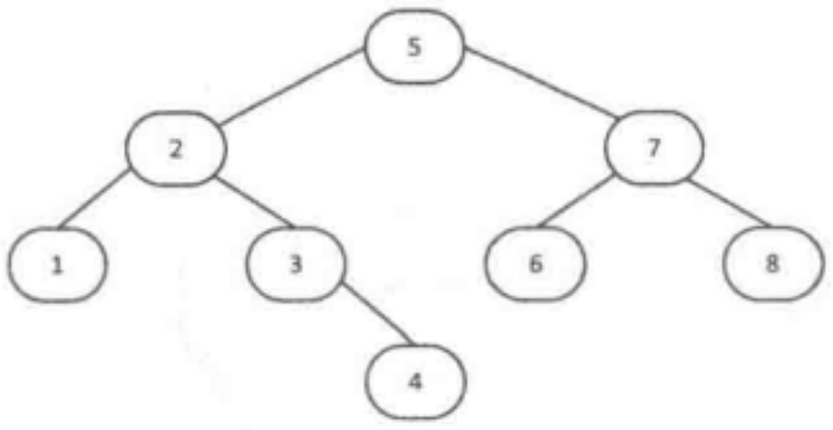


图 5-1　用 less<int>排序的整数平衡二叉树

当然，也可以为元素提供不同的比较函数：

```
std::set<std::string, std::greater<string>> words
{"one", "two", "three", "four", "five", "six", "seven", "eight"};
```

这个容器中的元素会降序排列，因此容器的树和图 5-2 类似。

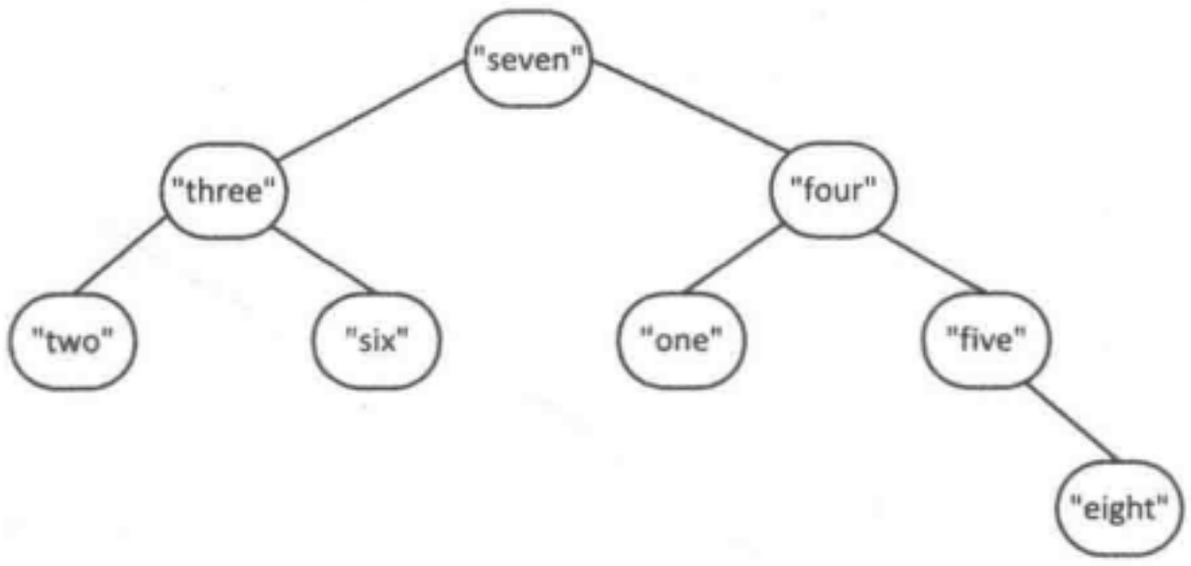


图 5-2　用 greater<string>排序的字符串的平衡二叉树

可以用元素段来创建 set 容器，并且可以指定它的比较函数：

```
std::set<string> words2 {std::begin(words), std::end(words)};
std::set<string, std::greater<string>> words3 {++std::begin(words2),
  std::end(words2)};
```

第一条语句定义了 words2，它包含了 words 中元素的副本，words 用默认的比较函数排序——less<string>实例。第二条语句定义了 words3，它包含 words2 中除第一个元素外的所有元素的副本。这个容器使用 less<string>实例排序。

set<T>模板也定义了拷贝和移动构造函数。移动构造函数比较重要，因为它可以不通过拷贝就返回函数局部定义的 set 容器。在函数结束时会返回一个局部 set 容器，编译器会识别这个动作，从而选择使用移动构造函数来返回 set 容器。你在本章的后面会看到一个这种用法的示例。

5.2.1　添加和移除元素

set 中没有实现成员函数 at()，也没有实现 operator[]()。除了这些操作外，set 容器提供 map 容器所提供的大部分操作。可以使用 insert()、emplace()、emplace_hint()成员函数来向 set 中添加元素。下面是一个使用 insert()的示例：

```
std::set<string, std::greater<string>> words {"one", "two", "three"};
```

```
auto pr1 = words.insert("four"); // pr1.first points to new element.
                                 // pr1.second is true
auto pr2 = words.insert("two");  // Element is NOT inserted - pr2.first
                                 // points to the
                                 // existing element and pr.second is false
auto iter3 = words.insert(pr.first, "seven");
// iter3 points to new element just before "four"
words.insert({"five", "six"}); // Insert list of elements - no return value
string wrds[] {"eight", "nine", "ten"};
words.insert(std::begin(wrds), std::end(wrds));  // Inserts range - no
                                                 // return value
```

插入单个元素会返回一个 pair<iterator,bool>对象。插入单个元素和一个标识，会返回一个迭代器。插入一段元素或一个初始化列表就不会有返回值。当 insert()的参数是初始化列表时，会用列表中的字符串创建 string 对象。

下面是两个在 set 容器中创建元素的示例：

```
std::set<std::pair<string,string>> names;
auto pr = names.emplace("Lisa", "Carr");
// pr.first points to new element. pr.second is true
auto iter = names.emplace_hint(pr.first, "Joe", "King");
```

这和 map 一样。成员函数 emplace()会返回一个 pair<iterator,bool>对象，而 emplace_hint()只返回一个迭代器。前者的参数被直接传入元素的构造函数，用来创建元素。emplace_hint()的第一个参数是一个迭代器，它指出了元素可能的插入位置，随后的参数会被传入元素的构造函数。

成员函数 clear()会删除 set 的所有元素。成员函数 erase()会删除迭代器指定位置的元素或与对象匹配的元素。例如：

```
std::set<int> numbers {2, 4, 6, 8, 10, 12, 14};
auto iter = numbers.erase(++std::begin(numbers));
                          // Removes the 2nd element - 4. iter points to 6
auto n = numbers.erase(12);     // Returns no. of elements removed - 1
n = numbers.erase(13);          // Returns no. of elements removed - 0
numbers.clear();                // Removes all elements
```

成员函数 erase()可以删除一段元素：

```
std::set<int> numbers {2, 4, 6, 8, 10, 12, 14};
auto iter1 = std::begin(numbers);  // iter1 points to 1st element
advance(iter1, 5);                 // Points to 6th element - 12
auto iter = numbers.erase(++std::begin(numbers), iter1);
                         // Remove 2nd to 5th inclusive. iter points to 12
```

如果 set 没有元素，成员函数 empty()返回 true，成员函数 size()返回它所包含的元素个数。如果担心无法在 set 中存储尽可能多的元素，可以调用成员函数 max_size()来得到可存储的最大元素个数，这显然会是一个很大的值。

5.2.2　访问元素

set 的成员函数 find()会返回一个和参数匹配的元素的迭代器。如果对象不在 set 中，会返回一个结束迭代器。例如：

```
std::set<string> words {"one", "two", "three", "four", "five"};
auto iter = words.find("one");                  // iter points to "one"
iter = words.find(string{"two"});               // iter points to "two"
iter = words.find("six");                       // iter is std::end(words)
```

调用成员函数 count()可以返回指定键所对应的元素个数，返回值通常是 0 或 1，因为 set 容器中的元素是唯一的。set 容器模板定义了成员函数 equal_range()、lower_bound()、upper_bound()，这和 multiset 容器在很大程度上是一致的。

5.2.3　使用 set

是时候了解一下 set 容器的用法了。我们把 vector、set 和 map 容器组合在一起来创建一个示例，并且会介绍一种新的有用算法。在这个示例中，你会将学习不同学科的学生分配到一组。每个学生都必须学习指定的最小数目的学科。每个学习特定学科的学生都被保存到 set 容器中，因为一个学生只能在一门特定课程中出现一次。这个示例不会特别有效率。在本例中会大量地拷贝学生对象，这里可能无关紧要，但是如果用来表示学生的对象很大，这就很重要了，因为这会产生很多开销。本章后面会介绍如何消除对象的副本。本例的基本工作流程如图 5-3 所示。

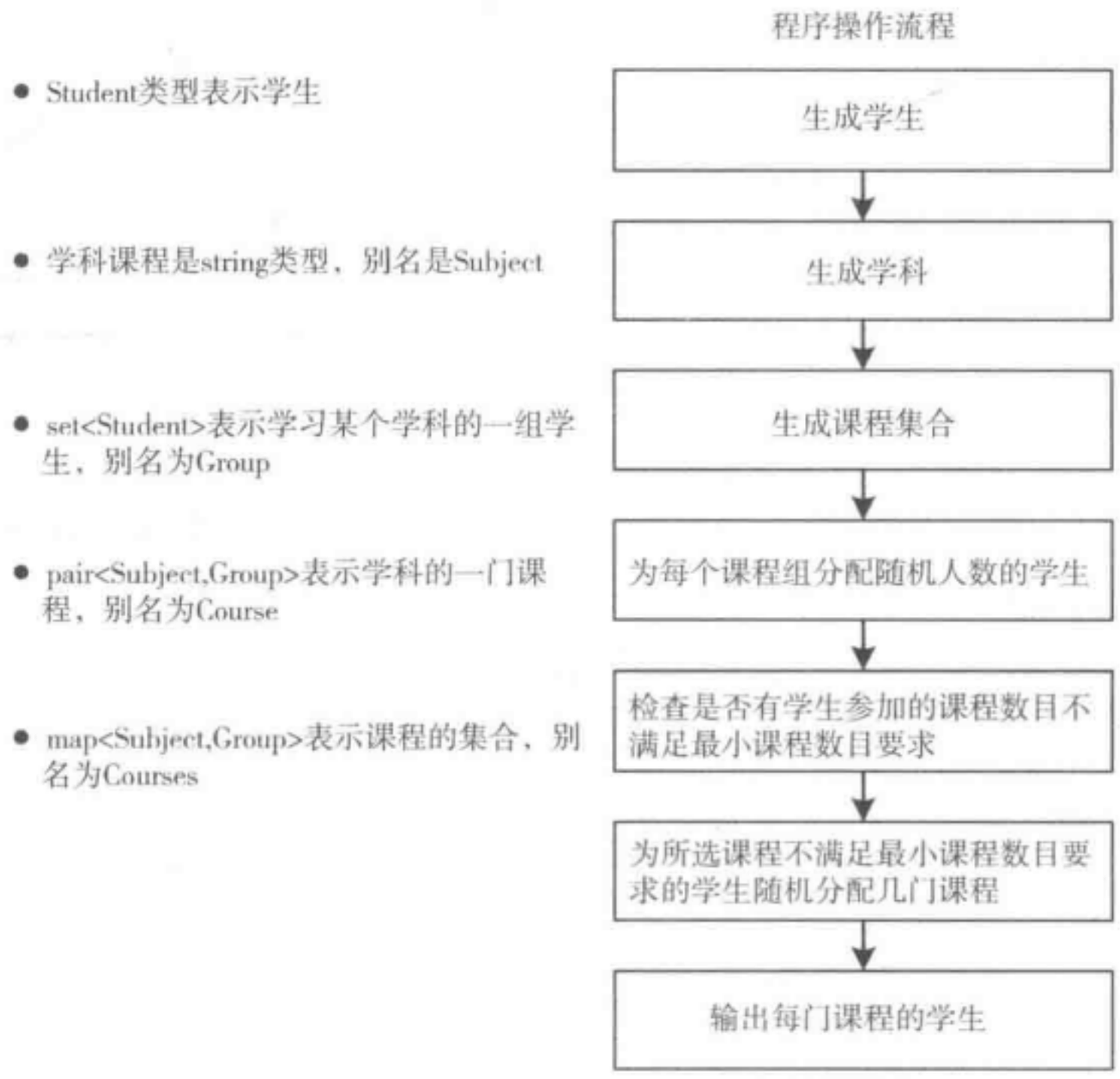


图 5-3　用 set 容器表示课程组

下面的 using 结构定义了一些本例中使用的别名：

```
using std::string;
using Distribution = std::uniform_int_distribution<size_t>;
using Subject = string;                          // A course subject
using Subjects = std::vector<Subject>;           // A vector of subjects
using Group = std::set<Student> ;                // A student group for a subject
using Students = std::vector<Student>;           // All the students
using Course = std::pair<Subject, Group>;        // A pair representing a course
using Courses = std::map<Subject, Group>;        // The container for courses
```

这些别名不是必需的，但它们可以使代码更加简洁。我们在第 4 章中使用的 std::uniform_int_distribution<size_t>中，distribution 就是它的别名，它定义了一个正态的统计分布，可以用来创建随机数。

首先，我们需要定义一个表示学生的类。Student 类的定义很简单，我们可以简单地在 Student.h 中定义这个类，例如：

```
#ifndef STUDENT_H
#define STUDENT_H
#include <string>                                // For string class
#include <ostream>                               // For output streams

class Student
{
private:
  std::string first {};
  std::string second {};

public:
  Student(const std::string& name1, const std::string& name2) : first
    (name1), second (name2){}
  // Move constructor
  Student(Student&& student) : first(std::move(student.first)),
                               second(std::move(student.second)){}

  Student(const Student& student) :
        first(student.first), second(student.second){} // Copy constructor
  Student() {}                                         // Default constructor

  // Less-than operator
  bool operator<(const Student& student) const
  {
    return second < student.second || (second == student.second && first <
      student.first);
  }

  friend std::ostream& operator<<(std::ostream& out, const Student & student);
};

// Insertion operator overload
```

```
inline std::ostream& operator<<(std::ostream& out, const Student& student)
{
  out << student.first + " " + student.second;
  return out;
}
#endif
```

Student 只有两个数据成员，用来保存学生的名和姓。Student 对象最初保存在 vector 容器中，因此需要定义默认的构造函数。这里有拷贝构造函数和移动构造函数，后者可以在适当时避免对象的副本。因为 Student 会被保存在 set 容器中，用来表示不同学科的课程，所以需要定义小于运算符。这里用友元函数重载了一个流插入运算符来辅助输出。

1. 创建 Student 对象

程序需要一定数量的 Student 对象，为了避免费力地从键盘输入数据，我们可以通过组合姓名的方式来创建 Student 对象的 vector 容器：

```
Students create_students()
{
  Students students;
  string first_names[] {"Ann", "Jim", "Eve", "Dan", "Ted"};
  string second_names[] {"Smith", "Jones", "Howe", "Watt", "Beck"};

  for(const auto& first : first_names)
  {
   for(const auto& second : second_names)
   {
    students.emplace_back(first, second);
   }
  }
  return students;
}
```

这里用 using 语句为 Student 对象的 vector 容器定义了一个别名 Students。函数用两个数组中的元素组合出了所有可能的姓名，在局部 students 容器中创建 Student 对象。外循环对名进行迭代，内循环在给定的名后添加姓。因此我们会有 25 名学生。我们可以按如下方式调用函数来创建 students：

```
Students students = create_students();
```

vector 容器有移动构造函数，因此编译器会移动而不是拷贝返回的局部 students。上面的语句会调用 vector<Student>的移动赋值运算符来移动 create_students()的返回值，因此不会拷贝这个 vector 及其元素。这里 students 的类型是显式的，也可以使用 auto，因为编译器可以从 create_students()的返回类型中推导出 students 的类型。

2. 创建某门学科的学生集合

在这个示例中需要随机选择一些学生和课程，因此需要一个随机数创建器。为了在整

个程序中都可以使用这个创建器，我们需要把它定义为全局变量：

```
static std::default_random_engine gen_value;
```

default_random_engine 和 uniform_int_distribution 类型都定义在 random 头文件中，分布对象是一个函数对象，它可以创建分布内的随机数，可以将随机数创建器对象作为参数传给分布对象的成员函数 operator()()。这里我们定义了一个函数，用来为指定学科随机创建一组学生：

```
Group make_group(const Students& students, size_t group_size,
                 const Distribution& choose_student)
{
 Group group;                    // The group of students for a subject

 // Select students for the subject group

 // Insert a random student into the group until there are group_size
   //students in it
 while(group.size() < group_size)
 {
  group.insert(students[choose_student(gen_value)]);
 }
 return group;
}
```

函数的第一个参数是一个包含 Student 对象的 vector，第二个参数是组中要求的学生个数，最后一个参数是分布的索引值，用来选择随机创建的学生。group 是一个包含 Student 对象的 set 容器。从作为函数的第一个参数传入的 vector 中随机选择一些学生，然后通过调用 group 的成员函数 insert()把它们插入到局部 set 容器 group 中。这里不需要检查插入是否成功。如果插入的元素已经在容器中，成员函数 insert()不会插入新的对象。从 students 容器中随机选择的学生可能已经在 group 容器中，但是循环会继续进行，尝试别的随机选择直到添加了 group_size 个学生。当函数结束时，会通过移动而不是复制的方式返回局部的 group 对象，这和前面的函数是一样的。

3. 创建学科和课程

学科的课程定义如下：

```
Subjects subjects {"Biology", "Physics", "Chemistry", "Mathematics",
"Astronomy","Drama", "Politics", "Philosophy", "Economics"};
```

Subjects 是 vector<Subject>的别名，Subject 是 string 的别名，因而 subjects 是一个包含 string 对象的 vector 容器。我们会在 map<Subject,Group>中保存一些课程，每门学科的键都是 subjects 容器中唯一的 Subject 对象。map<Subject,Group>的别名为 Courses，因此我们可以像下面这样定义一个包含全部课程的容器：

```
Courses courses; // All the courses with subject keys
```

这里定义了每个学生需要学习的最少学科数目。在创建某个学科的学生集合时，我们也对每个课程组的初始大小进行了限制：

```
size_t min_subjects {4};            // Minimum number of Subjects per student
size_t min_group {min_subjects}; // Minimum no. of students per course
size_t max_group {(students.size()*min_subjects)/subjects.size()};
                                    // Max initial students per course
```

这里随便选了 4 作为需要学习的最少学科数，而且将每组的最少学生数都设为相同的值。如果所有学生都学习最少的学科，而且被平均分配到每组，那么每个学科组的最大学生数就是一个平均值。可以尝试修改这些参数，看看它们对于学生分配有什么影响。

为了用随机值作为学科组分配的学生个数，也为了随机选择学生，我们需要定义一个用来在均匀分布中选择随机值的函数对象：

```
Distribution group_size {min_group, max_group}; // Distribution for
                                                // students per course
Distribution choose_student {0, students.size() - 1}; // Random student
                                                      // selector
```

group_size 可以创建 min_group～max_group 范围内的随机数。类似的，choose_student 分布可以创建 students 中的有效索引值。

我们也想为学生随机选择一些课程，所以这里也需要定义一个 Distribution 对象：

```
Distribution choose_course {0, subjects.size() - 1};
// Random course selector
```

上面的代码会创建 subjects 容器的一个有效索引值，可以用它来选择课程。

4. 为课程分配学生

main()中的 for 循环只用了一条语句就为 course 容器填充了一些表示课程信息的元素：

```
for(const auto& subject : subjects)
    courses.emplace(subject, make_group(students, group_size(gen_value),
            choose_student));
```

这个循环体中的语句看起来确实很长。调用 course 容器的 emplace()可以在适当的位置创建元素。每个元素都是一个 pair<Subject, Group>对象，因此 emplace()的参数必须是一个 Subject 对象和一个 Group 对象，emplace()函数会把它们传入 pair 的构造函数。Emplace()第一个参数是循环变量 subject，因为循环迭代器需要遍历 subjects。第二个参数是由 make_group()返回的。make_group()的第一个参数是一个 vector 容器 students，第二个参数通过将随机数创建器传入函数对象 group_size()来创建随机数，用它作为组的大小。第三个参数是一个用于选择学生的分布对象。

5. 检查学生选择的课程

在使用上面的循环创建了所有课程信息后，我们必须检查每个学生是否都选了足够的

课程。下面的循环会做这些事情，如果没有选择足够的课程，为他们申请其他的课程：

```
for(const auto& student : students)
{ // Verify the minimum number of Subjects has been met

  // Count how many Subjects the student is on
  size_t course_count = std::count_if(std::begin(courses),
    std::end(courses),[&student](const Course& course) { return course.
            second.count(student); });
  if(course_count >= min_subjects) continue; // On to the next student
// Minimum no. of Subjects not signed up for
size_t additional {min_subjects - course_count}; // Additional no. of
                                                 // Subjects needed
if(!course_count) // If none have been chosen...
  std::cout << student << " is work-shy, having signed up for NO
    Subjects!\n";
else // Some have - but E for effort
  std::cout << student << " is only signed up for " << course_count <<
   " Subjects!\n";

std::cout << "Registering " << student << " for " << additional
          << " more course" << (additional > 1 ? "s" : "") << ".\n\n";
// Register for additional Subjects up to the minimum
while(course_count < min_subjects)
  if((courses.find(subjects[choose_course(gen_value)])->second.
    insert(student)).second)
      ++course_count;
}
```

外层循环遍历 vector 中的 Student 对象。count_if()算法用来计算每个学生已报名参加的课程数。在前两个参数指定的范围内，算法计算了可以使第三个函数参数返回 true 的元素个数。前两个参数指定的是 courses 容器中的一段元素，因此迭代器指向的是 pair<Subject, Group>对象。count_if()的第三个参数必须是一个二元函数，它的返回值必须是布尔值或其他可以隐式转换为布尔型的值。类型参数来自于迭代器的解引用。这里的 lambda 表达式返回了 course.second.count(student)的值。course 是 pair<Subject,Group>类型的对象，因此这里表达式首先选择的是 course 对象的第二个成员。第二个成员是 Group 对象，它是 set<Student>类型的对象，所以这个表达式会调用 set 容器的成员函数 count()。因为 set 容器中不允许有重复元素，所以如果 student 在容器中，成员函数 count()只会返回 1；如果不在容器中，返回 0。幸运的是，这两个值都可以分别相应地转换为布尔值 true 和 false，因此，如果 student 在当前容器中，count_if()会增加计数。

我们会对任何没有申请足够数量课程的学生输出一些适当的信息，然后在嵌套的 while 循环中为这些学生申请新的课程，直到他们申请的课程数目满足要求。这是另一条做了很多工作的单条循环语句。从本质上来说，这是一条 if 语句，每当学生成功申请一门课程时，就会增加 course_count 的值。通过调用 courses 容器的成员函数 find()来选择课程，它会返回一个迭代器，它指向参数键所对应的元素；如果没有找到对应的键，会返回一个结束迭

代器。我们用所有可能的 Subject 作为键创建了 courses，所以后面的这种情况不会发生；如果真的发生了，我们会知道的，因为这会导致程序崩溃。用 choose_course 分布产生的随机索引值作为 subjects 的键来选择一门新的课程。find()返回的 pair 元素的第二个成员是选择这门课程的学生组，因此可以通过用 student 作为参数调用它的成员函数 insert()，来向组中添加一些之前不在这个组中的学生。总有可能出现学生已经选了这门课程的情况，这时 insert()返回的 pair 对象的第二个成员是 false，course_count 不会递增，循环会为当前学生继续随机选择其他的课程。在循环结束时，每个学生选择的课程数目都满足最小值 min_subjects。

6. 输出课程信息

为了可以用另一个 STL 算法输出课程信息，我们需要在 List_Course.h 头文件中定义一个下面这种类型的函数对象：

```
// List_Course.h
// Function object to output the students in a group for Ex5_01
#ifndef LIST_COURSE_H
#define LIST_COURSE_H
#include <iostream>                          // For standard streams
#include <string>                            // For string class
#include <set>                               // For set container
#include <algorithm>                         // For copy()
#include <iterator>                          // For ostream_iterator
#include "Student.h"

using Subject = std::string;                 // A course subject
using Group = std::set<Student>;             // A student group for a subject
using Course = std::pair<Subject, Group>; // A pair representing a course

class List_Course
{
public:
  void operator()(const Course& course)
  {
    std::cout << "\n\n" << course.first << " " << course.second.size()
              << " students:\n ";
    std::copy( std::begin(course.second), std::end(course.second),
               std::ostream_iterator<Student>(std::cout, " "));
  }
};
#endif
```

List_Course 类的成员函数 operator()()的参数是一个 Course 对象的引用，这个对象的类型是 pair<string,set<Student>>。这个函数会输出课程的一些信息，这些信息包括 course 的第一个成员，以及通过 pair 的第二个成员的成员函数 size()得到的学习这门课程的学生数。前两个参数分别是 set<Student>容器的开始迭代器和结束迭代器，这两个迭代器都是从

course 的第二个成员得到的。Student 对象的复制目的地由 copy()的第三个参数指定，它是一个 ostream_iterator<Student>对象。这个对象会通过调用每个 Student 对象的成员函数 operator<<()来将学生的信息输出到 cout，输出的信息用两个空格隔开。

通过 main()中的一个 List_Course 实例，我们可以用一行语句输出所有的课程信息：

```
std::for_each(std::begin(courses), std::end(courses), List_Course());
```

for_each 算法会将第三个参数指定的函数对象应用到前两个参数所指定范围内的所有元素上。前两个参数定义的范围相当于所有的课程，因此会以课程为参数连续调用 List_Courses()()。结果会将每门课程的全部学生输出到 cout。

当然，也可以不用 List_Course。我们可以定义一个 lambda 表达式来代替它作为 for_each()的第三个参数：

```
std::for_each(std::begin(courses), std::end(courses),
[](const Course& course){
std::cout << "\n\n" << course.first << " " << course.second.size()
          << " students:\n ";
std::copy(std::begin(course.second), std::end(course.second),
                          std::ostream_iterator<Student>(std::cout, " "));
});
```

我们在这个示例中定义 List_Course 只是为了演示如何定义它，但在这个程序中需要多次使用这个函数对象，使用函数就比使用 lambda 表达式简单一些。

7. 完整的程序

包含 main()的源文件的内容如下：

```
// Ex5_01.cpp
// Registering students on Subjects
#include <iostream>                          // For standard streams
#include <string>                            // For string class
#include <map>                               // For map container
#include <set>                               // For set container
#include <vector>                            // For vector container
#include <random>                            // For random number generation
#include <algorithm>                         // For for_each(), count_if()
#include "Student.h"
#include "List_Course.h"

using std::string;
using Distribution = std::uniform_int_distribution<size_t>;
using Subject = string;                      // A course subject
using Subjects = std::vector<Subject>;       // A vector of subjects
using Group = std::set<Student>;             // A student group for a subject
using Students = std::vector<Student>;       // All the students
using Course = std::pair<Subject, Group>; // A pair representing a course
using Courses = std::map<Subject, Group>; // The container for courses
```

```
static std::default_random_engine gen_value;

// create_students() helper function definition goes here...

// make_group () helper function definition goes here...

int main()
{
  Students students = create_students();
  Subjects subjects {"Biology", "Physics", "Chemistry", "Mathematics",
    "Astronomy","Drama", "Politics", "Philosophy", "Economics"};
  Courses courses; // All the courses with subject keys
  size_t min_subjects {4}; // Minimum number of Subjects per student
  size_t min_group {min_subjects}; // Minimum no. of students per course

  // Maximum initial students per course
  size_t max_group {(students.size()*min_subjects)/subjects.size()};

  // Create groups of students for each subject
  Distribution group_size {min_group, max_group};
  // Distribution for students per course
  Distribution choose_student {0, students.size() - 1};
  // Random student selector
  for(const auto& subject : subjects)
    courses.emplace(subject, make_group(students, group_size(gen_value),
      choose_student));

  Distribution choose_course {0, subjects.size() - 1};
  // Random course selector

  // Every student must attend a minimum number of Subjects...
  // ...but students being students we must check...
  for(const auto& student : students)
  { // Verify the minimum number of Subjects has been met

    // Count how many Subjects the student is on
    size_t course_count = std::count_if(std::begin(courses),
      std::end (courses),[&student](const Course& course){ return course.
          second.count(student); });
    if(course_count >= min_subjects) continue; // On to the next student

    // Minimum no. of Subjects not signed up for
    size_t additional {min_subjects - course_count};
    // Additional no. of Subjects needed
    if(!course_count) // If none have been chosen...
      std::cout << student << " is work-shy, having signed up for NO
        Subjects!\n";
    else // Some have - but E for effort
      std::cout << student << " is only signed up for " << course_count
                << " Subjects!\n";

    std::cout << "Registering " << student << " for " << additional
              << " more course" << (additional > 1 ? "s" : "") << ".\n\n";
```

```
    // Register for additional Subjects up to the minimum
    while(course_count < min_subjects)
      if((courses.find(subjects[choose_course(gen_value)])->second.
        insert(student)).second)
      ++course_count;
  }

  // Output the students attending each course
  std::for_each(std::begin(courses), std::end(courses), List_Course());
  std::cout << std::endl;
}
```

这里不会列出所有的输出，因为它相当冗长，但这里会展示得到的一些输出片段：

```
Ann Smith is only signed up for 1 Subjects!
Registering Ann Smith for 3 more courses.

Ann Watt is only signed up for 2 Subjects!
Registering Ann Watt for 2 more courses.

Ann Beck is only signed up for 3 Subjects!
Registering Ann Beck for 1 more course.

Jim Smith is work-shy, having signed up for NO Subjects!
Registering Jim Smith for 4 more courses.
...
Ted Beck is work-shy, having signed up for NO Subjects!
Registering Ted Beck for 4 more courses.

Astronomy 9 students:
  Dan Beck Ted Beck Eve Howe Ann Jones Dan Jones Eve Jones Ted Smith Ann Watt
    Dan Watt

Biology 14 students:
  Ann Beck Dan Beck Jim Beck Ann Howe Dan Howe Jim Howe Dan Jones Ted Jones Dan
Smith Eve Smith
  Ann Watt Eve Watt Jim Watt Ted Watt

Chemistry 10 students:
  Eve Beck Dan Howe Ann Jones Eve Jones Ted Jones Dan Smith Jim Smith Ann Watt Dan
Watt Jim Watt
...
Physics 15 students:
  Ann Beck Dan Beck Eve Beck Jim Beck Eve Howe Ted Howe Ann Jones Jim Jones Ann
Smith Eve Smith
  Ted Smith Dan Watt Eve Watt Jim Watt Ted Watt

Politics 12 students:
  Eve Beck Jim Howe Ted Howe Dan Jones Eve Jones Jim Jones Ann Smith Dan Smith Eve
Smith Jim Smith
  Dan Watt Ted Watt
```

5.2.4　set 迭代器

set<T>容器的成员返回的迭代器都是双向迭代器。这些迭代器的类型的别名定义在set<T>模板中，可以从 set 中得到类型别名有 iterator、reverse_iterator、const_iterator、const_reverse_iterator，从它们的名称就可以看出它们的类型。例如，成员函数 begin()和 end()会返回 iterator 类型的迭代器，成员函数 rbegin()和 rend()会返回 reverse_iterator 类型的迭代器，成员函数 cbegin()和 cend()会返回 const_iterator 类型的迭代器。最后，成员函数 crbegin()和 crend()可以返回 const_reverse_iterator 类型的迭代器。

然而，set 容器的迭代器类型的别名有时会让人产生一些误解。所有 set<T>容器的成员函数返回的迭代器都指向 const T 类型的元素。因此，iterator 迭代器会指向 const 元素，reverse_iterator 和其他类型的迭代器也是如此。这意味着我们不能修改元素。如果想要修改 set 容器中的元素，必须先删除它，然后再插入修改后的版本。

仔细思考一下，其实这是不合理的。set 中的对象以它们自己作为键，对象在容器中的位置是通过比较对象决定的。如果可以修改元素，元素的顺序就失效了，也会扰乱后面的访问操作。当必须修改元素而且仍然需要将它们组合到一个或多个 set 容器中时，还有一个方法可以做到这一点。可以在 set 容器中保存指针——最好选择智能指针。当使用 set 容器时，通常会在它们中保存 shared_ptr<T>或 weak_ptr<T>对象。在 set 容器中保存 unique_ptr<T>对象没有多少意义。因为容器中不存在和 unique_ptr<T>对象匹配的独立键，所以我们从来不会直接检索元素。

5.2.5　在 set 容器中保存指针

如果改变对象，可能会改变 set 中对象指针的顺序，所以指针的比较函数不能和对象有关。大多数时候，我们并不在意元素在 set 中的顺序，而是在意容器中是否有这个元素。在这种情况下，就可以使用一个适用于指针但和它们所指向的元素无关的比较函数对象，推荐使用定义在 memory 头文件中的 owner_less<T>函数对象类型的实例来比较容器中的智能指针。

owner_less<T>模板为 shared_ptr 和 weak_ptr 对象定义了用于小于比较的函数对象类型。换句话说，允许 weak_ptr 对象和 shared_ptr 对象比较，反过来也可以，也允许和 weak_ptr 或 shared_ptr 对象比较。通过调用智能指针的成员函数 owner_before()实现了一个 owner_less<T>实例，它提供了一个小于运算符，可以和另一个智能指针进行比较。shared_ptr<T> 和 weak_ptr<T>模板都定义了这个成员函数。当这个智能指针比参数传入的智能指针小时，owner_before<T>()实例会返回 true，否则返回 false。比较基于智能指针所拥有的对象地址，当两个指针指向同一个对象时，说明这两个指针等价。

在 shared_ptr<T>类模板中，定义 owner_before()实例的函数模板的原型看起来如下所示：

- template<typename X> bool owner_before(const std::shared_ptr<X>& other) const;
- template<typename X> bool owner_before(const std::weak_ptr<X>& other) const

weak_ptr<T>类模板定义了类似的成员。注意，这里模板类型参数和类模板的类型参数

不同。这意味着可以比较指向相同类型对象的指针，也可以比较指向不同类型对象的指针。也就是说，shared_ptr<T1>对象可以和 shared_ptr<T2>对象或 weak_ptr<T2> 对象比较。这意味着指针所指向的对象可以和它所拥有的对象不同。

所有权对于 shared_ptr<T>对象很重要。对于本书来说，这个话题可能有点深奥，但还是要说明一下。shared_ptr<T>可以共享一个不属于它的对象的所有权。换句话说，shared 指针所包含的地址并不是属于它的对象的地址。这种 shared_ptr 的一种用途是指向一个它所拥有对象的成员，如图 5-4 所示。

```
class Person
{
public:
 std::string name;

 Person (const string& nm):name(nm){}
 ...
};

auto pperson = std::make_shared<Person>("Fred");
std::shared_ptr<std::string> pname {pperson, &pperson->name};
```

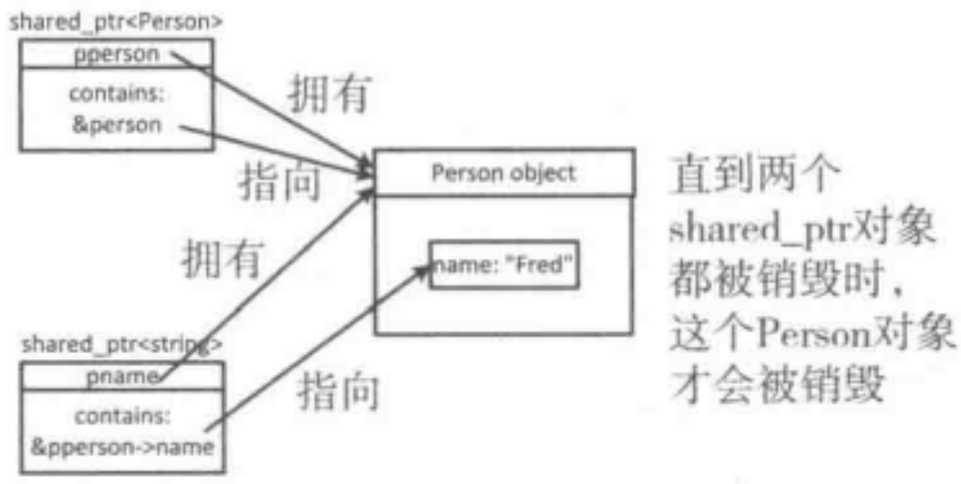


图 5-4　两个 shared 指针指向同一个对象的不同对象

在图 5-4 中用来创建 pname 的构造函数叫作别名构造函数。它的第一个参数是另一个 shared_ptr，它拥有的对象 pname 也拥有。第二个参数是一个原生指针，它保存在 pname 中。第二个参数指向的对象不由 pname 管理。然而，pname 可以用来访问 Person 对象的数据成员。在这个示例中，pname 拥有的对象是 Person，它指向对象的成员。

销毁图 5-4 中的 pperson 指针不会导致它所拥有的对象被销毁，因为 pname 指针仍然拥有这个对象。创建 pname 指针是用来访问 Person 对象的成员变量 name。*pperson 是 Person 对象的引用，因为它包含了已共享所有权的对象的地址。*pname 是 Person 对象的成员 name 的引用，因为 pname 包含了这个成员的地址，而且它也拥有 Person 对象的所有权。这时我们在没有包含 Person 地址的 shared_ptr<Person>指针时，仍然可以继续使用 pname。当所有拥有对象所有权的 shared_ptr 对象都被销毁时，才能销毁它们所指向的 Person 对象。如果没有别名构造函数提供的这种能力，就无法保证 pname 所保存指针的有效性。

1. 一个在 set 容器中保存指针的示例

为了展示如何在 set 容器中保存智能指针，将 Ex5_01 重新构造为 Ex5_02。vector 容器 students 中的元素和 set 容器中的元素都是智能指针，它们用来表示学习不同学科的学生群体。可以在 set 容器中使用 shared_ptr<Student>对象，但只在 vector 中使用 shared_ptr<Student>元素，所以这里选择在 set 容器中使用 weak_ptr<Student>对象。注意，当在容器中保存 weak_ptr<T>对象时，需要确保在使用 weak_ptr<T>时，它所依赖的 shared_ptr<T>对象仍然存在。当然，我们总是能够通过调用 weak_ptr<T>的成员函数 expired()来检查和它关联的 shared_ptr<T>是否存在；当关联的 shared_ptr<T>对象被删除时，它会返回 true。

我们所需要做的第一件事是重新定义 Ex5_02.cpp 中使用的类型别名，从而使它们可以适用于智能指针：

```
using std::string;
using Distribution = std::uniform_int_distribution<size_t>;
using Subject = string;                              // A course subject
using Subjects = std::vector<Subject>;               // A vector of subjects
using Group = std::set<std::weak_ptr<Student>,   // Group for a subject
   std::owner_less<std::weak_ptr<Student>>>;
using Students = std::vector<std::shared_ptr<Student>>; // All the students
using Course = std::pair<Subject, Group>;        // Represents a course
using Courses = std::map<Subject, Group>;        // The courses
```

这里只需要修改 Group 和 Students 的别名。Group 现在是一个包含 weak_ptr<Student>对象的 set 容器，它的元素可以用 owner_less<weak_ptr<Student>>实例来比较。这允许 set 的元素和 vector<shared_ptr<Student>>类型的容器 Students 的元素进行比较。如果两个指针指向同一个对象，那么 set 中的这个元素和 vector 的这个元素匹配。

2. 创建一个以 shared_ptr 为元素的 vector

Ex5_01 中的 Student 类可以保留下来，但是需要修改 create_students()函数中的一条语句：

```
Students create_students()
{
  Students students;
  string first_names[] {"Ann", "Jim", "Eve", "Dan", "Ted"};
  string second_names[] {"Smith", "Jones", "Howe", "Watt", "Beck"};

  for(const auto& first : first_names)
    for(const auto& second : second_names)
    {
      students.emplace_back(std::make_shared<Student>(first, second));
    }
  return students;
}
```

现在，emplace_back()的参数是一个由 make_shared<Student>()返回的 shared_ptr<Student>对象。因为它是一个临时对象，所以这个指针会被 emplace_back()转发到 shared_ptr<Student>的移动构造函数中来创建 vector 的元素。

令人惊讶的是，我们一点儿也不需要修改 make_group()。别名考虑到了所有这些修改。

3. 输出 weak_ptr 引用的对象

需要修改函数对象类型 List_Course：

```
using Subject = std::string;                    // A course subject
using Group =                                   // A group for a subject
  std::set<std::weak_ptr<Student>, std::owner_less<std::weak_ptr<Student>>>;
using Course = std::pair<Subject, Group>; // A pair representing a course

class List_Course
{
```

```
public:
  void operator()(const Course& course)
  {
    std::cout << "\n\n" << course.first << " " << course.second.size()
              << " students:\n ";
    std::copy(std::begin(course.second), std::end(course.second),
      std::ostream_iterator<std::weak_ptr<Student>>(std::cout, " "));
  }
};
inline std::ostream& operator<<(std::ostream& out, const std::weak_ptr
<Student>& wss)
{
  out << *wss.lock();
  return out;
}
```

已经在头文件中复制了在 Ex5_02.cpp Group 别名定义中所做的修改。首先需要修改的是函数调用运算符函数的定义。ostream_iterator 模板类型参数需要改为 weak_ptr<Student>。这需要对 weak_ptr<Student>对象的插入运算进行重载。为了将 Student 对象写入流中，需要解引用指针。不能直接解引用 weak_ptr<T>；必须首先获取一个和 weak_ptr<T>拥有相同对象的 shared_ptr<T>，然后再解引用这个 shared_ptr<T>。调用 weak_ptr<Student>对象的 lock()会返回一个 shared_ptr<Student>对象，它拥有这个 weak_ptr 所指向的对象，并且可以解引用。在这个示例中，我们有理由相信 Student 对象总是存在，但在一般情况下不可能这样。如果拥有的全部 shared_ptr 对象被销毁，weak_ptr 指向的对象会被销毁，调用 weak_ptr 的 lock()会返回一个包含 nullptr 的 shared_ptr。在可能出现这种情况的地方，为了避免程序崩溃，必须检查返回值是否为 nullptr。

类型别名已经照顾到了 main()中需要做的修改，但是仍然需要解引用 student，从而循环输出 students 容器中的对象，因此：

```
if(!course_count)                                  // If none have been chosen...
  std::cout << *student << " is work-shy, having signed up for NO Subjects!\n";
else                                               // Some have - but E for effort
  std::cout << *student << " is only signed up for " << course_count
            << " Subjects!\n";

std::cout << "Registering " << *student << " for " << additional
          << " more course" << (additional > 1 ? "s" : "") << ".\n\n";
```

完整的程序代码可以从第 5 章的 Ex5_02 中下载。如果运行这个程序，你会得到和 Ex5_01 相似的输出。

4. 以智能指针作为 map 容器中的键

可以从另一个方面来修改Ex5_02——map中所有的键是来自于当前subjects中的string对象的副本。如果用智能指针作为 map 的键，我们应该如何修改代码？

学科的别名需要修改，为了方便，我们增加了对 std::make_shared 的 using 声明，Courses

的别名定义也有些不同：

```
using std::make_shared;
using Subject = std::shared_ptr<string>;                    // A course subject
using Courses = std::map<Subject, Group, std::owner_less<Subject>>;
 // The container for courses
```

现在以 shared_ptr<string>指针为键，所以比较键的函数对象类型就是 owner_less<shared_ptr<string>>。显然，也需要修改 main()中 subjects 的定义：

```
Subjects subjects { make_shared<string>("Biology"), make_shared<string>
   ("Physics"),
make_shared<string>("Chemistry"), make_shared<string>("Mathematics"),
make_shared<string>("Astronomy"), make_shared<string>("Drama"),
make_shared<string>("Politics"), make_shared<string>("Philosophy"),
make_shared<string>("Economics") };
```

现在，vector 中的元素是智能指针。

List_Course 的成员函数 operator()()必须考虑到现在 map 中 course 的键是指针：

```
void operator()(const Course& course)
{
  std::cout << "\n\n" << *course.first << " " << course.second.size()
            << " students:\n ";
  std::copy(std::begin(course.second), std::end(course.second),
            std::ostream_iterator<std::weak_ptr<Student>>(std::cout, " "));
}
```

当然唯一的变化是需要用*来解引用 course 的第一个成员，它是一个键。main()中不需要做其他改变了——它会和前面的版本那样工作，产生相似的输出，但在一些地方会有些许不同。稍后会在功能扩展时阐明这一点。

5. 比较智能指针遇到的麻烦

假设不在 main()的末尾输出所有课程的信息，我们提供一种从键盘输入学科的能力，然后显示所有正在学习这个学科的学生。首先，为了确保输入正确的课程信息，我们需要为学科的输入提供提示：

```
std::cout << "Course subjects are:\n ";
for(const auto& p : subjects)
  std::cout << *p << " ";
std::cout << "\n\n";
```

现在可以定义一个循环来从键盘读入学科，然后列出选择这门课程的学生：

```
// Code that doesn't work!
char answer {'Y'};
string subject {};
while(std::toupper(answer) == 'Y')
```

```
{
  std::cout << "Enter a course subject to get the list of students: ";
  std::cin >> subject;
  auto iter = courses.find(make_shared<string>(subject));
  if(iter == std::end(courses))
    std::cout << subject << " not found!\n";
  else
  {
    List_Course()(*iter);
    std::cout << std::endl;
  }
  std::cout << "Do you want to see another subject(Y or N)? ";
  std::cin >> answer;
}
```

这看起来很简单。我们用输入流创建了一个 shared_ptr<string>对象，然后用它来查找键。然而，这段代码却没有找到一个学科。这个程序可以列出所有的课程，却不能找到任何和输入的学科匹配的课程。为什么？

答案就在于比较函数对象 owner_less<T>可以用来比较 map 中的键。当两个指针拥有相同的对象时，它们才匹配。指向相同对象的指针并不是同一个指针，而且它们也绝不相等。基于 subject 获取课程的唯一方法是访问 subject 的原始智能指针或其副本。下面是修改后的代码：

```
char answer {'Y'};
string subject {};
while(std::toupper(answer) == 'Y')
{
  std::cout << "Enter a course subject to get the list of students: ";
  std::cin >> subject;
  auto iter = std::find_if( std::begin(subjects),
            std::end(subjects), // Find the pointer in subjects
            [&subject](const Subject& psubj){ return subject == *psubj; });
  if(iter == std::end(subjects))
    std::cout << subject << " not found!\n";
  else
  {
    List_Course()(*courses.find(*iter));
    std::cout << std::endl;
  }
  std::cout << "Do you want to see another subject(Y or N)? ";
  std::cin >> answer;
}
```

find_if()算法可以返回一个迭代器，它指向在前两个参数所指定范围内的、可以使作为第三个参数的函数对象返回 true 的元素。如果这个元素不存在，返回指定范围内的最后一个迭代器。这里指定的范围包含 subjects 的所有元素，最后一个参数是一个 lambda 表达式，当这个范围内有元素解引用后和 subject 匹配时，表达式会返回 true。如果返回的迭代器不

是 subjects 的结束迭代器，那么它指向 vector 中的一个 shared_ptr<Subject>。通过解引用迭代器来访问 subjects 中的元素，可以作为参数传给 courses 容器的成员函数 find()。它会返回一个迭代器，指向和键对应的 Course 对象，可以通过解引用迭代器来得到一个 Course 对象，用它的成员函数 find()的返回值作为一个 List_Course 函数调用实例的参数来输出选择了课程的学生。这个版本的程序代码可以从下载的 Ex5_03 中获取。

5.3　使用 multiset<T>容器

multiset<T>容器就像 set<T>容器，但它可以保存重复的元素。这意味我们总可以插入元素——当然必须是可接受的元素类型。默认用 less<T>来比较元素，但也可以指定不同的比较函数。在元素等价时，它必须返回 false。例如：

```
std::multiset<string, std::greater<string>> words{ {"dog", "cat", "mouse"},
    std::greater<string>() };
```

这条语句定义了一个以 string 为元素的 multiset，它以 greater<string>作为构造函数的第二个参数。构造函数的第一个参数是一个初始化列表，它为这个容器指定了三个初始元素。和 set 一样，如果它的两个元素相等，那么它们就是匹配的。在一个有比较运算符 comp 的 multiset 中，如果表达式!(a comp b)&&!(b comp a)为 true，那么元素 a 和 b 就是相等的。multiset 容器和 set 容器有相同的成员函数，但是因为 multiset 可以保存重复元素，有些函数的表现会有些不同。和 set 容器中的成员函数表现不同的是：

- insert()总是可以成功执行。当插入单个元素时，返回的迭代器指向插入的元素。当插入一段元素时，返回的迭代器指向插入的最后一个元素。
- emplace()和 emplace_hint()总是成功。它们都指向创建的新元素。
- find()会返回和参数匹配的第一个元素的迭代器，如果都不匹配，则返回容器的结束迭代器。
- equal_range()返回一个包含迭代器的 pair 对象，它定义了一个和参数匹配的元素段。如果没有元素匹配的话，pair 的第一个成员是容器的结束迭代器；在这种情况下，第二个成员是比参数大的第一个元素，如果都没有的话，它也是容器的结束迭代器。
- lower_bound()返回和参数匹配的第一个元素的迭代器，如果没有匹配的元素，会返回容器的结束迭代器。返回的迭代器和 range()返回的 pair 的第一个成员相同。
- upper_bound()返回的迭代器和 equal_range()返回的 pair 的第二个成员相同。
- count()返回和参数匹配的元素的个数。

用 multiset 容器代替 map，实现例 Ex4_02 中分析单词出现次数的那个程序：

```
// Ex5_04.cpp
// Determining word frequency
#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
#include <string>                      // For string class
#include <sstream>                     // For istringstream
```

```
#include <algorithm>                    // For replace_if() & for_each()
#include <set>                          // For set container
#include <iterator>                     // For advance()
#include <cctype>                       // For isalpha()
using std::string;

int main()
{
  std::cout << "Enter some text and enter * to end:\n";
  string text_in {};
  std::getline(std::cin, text_in, '*');

  // Replace non-alphabetic characters by a space
  std::replace_if(std::begin(text_in), std::end(text_in),
                  [](const char& ch){ return !isalpha(ch); }, ' ');

  std::istringstream text(text_in);          // Text input string as a stream
  std::istream_iterator<string> begin(text); // Stream iterator
  std::istream_iterator<string> end;         // End stream iterator

  std::multiset<string> words;               // Container to store words
  size_t max_len {};                         // Maximum word length

  // Get the words, store in the container, and find maximum length
  std::for_each(begin, end, [&max_len, &words](const string& word)
                            { words.emplace(word);
                              max_len = std::max(max_len, word.length());
                            });
  size_t per_line {4},                       // Outputs per line
         count {};                           // No. of words output

  for(auto iter = std::begin(words); iter != std::end(words);
  iter = words.upper_bound(*iter))
  {
    std::cout << std::left << std::setw(max_len + 1) << *iter
              << std::setw(3) << std::right << words.count(*iter) << " ";
    if(++count % per_line == 0) std::cout << std::endl;
  }
  std::cout << std::endl;
}
```

在输入过程中，和 Ex4_02 一样都移除了输入中的非字母字符。单词是由 foreach 从 istringstream 对象的文本中提取的，然后把它们传给了一个 lambda 表达式，这个表达式是 for_each()的最后一个参数，用来创建 multiset 容器中的元素。从 text 中获取的每个单词都被单独保存，因为一般来说，容器中会出现重复的元素。for 循环遍历 multiset 容器 words 中的迭代器，从指向第一个元素的开始迭代器开始。容器中的元素是有序的，因而相等的元素位置是连续的。通过调用容器的成员函数 count()，可以获取和它的参数 iter 所指向元素相等的元素的个数。每次循环迭代结束后，iter 被设为 upper_bound()返回的值，它指向一个不同于当前元素的元素。如果不存在这样的元素，upper_bound()会返回容器的结束迭

代器，循环就此结束。

因为 multiset 中的元素是有序的，所以可以在循环中用相等的单词数量来增加迭代器，例如：

```
size_t word_count {};                    // Number of identical words
for(auto iter = std::begin(words); iter != std::end(words);)
{
  word_count = words.count(*iter);
  std::cout << std::left << std::setw(max_len + 1) << *iter
            << std::setw(3) << std::right << word_count << " ";
  if(++count % per_line == 0) std::cout << std::endl;
  std::advance(iter, word_count);
}
```

这种方式比之前的循环更好。但这个版本的循环结束不是那么明显。笔者觉得 Ex4_02 中的方法比这个 multiset 版本的优雅。下面是示例输出：

```
Enter some text and enter * to end:
He was saying godnight to his horse.
He was saying goodnight to his horse,
And as he was saying goodnight to his horse, he was saying goodnight to his
horse.
"Goodnight horse, goodnight horse", he was saying goodnight to his horse.*
And        1 Goodnight     1 He          2 as          1
godnight   1 goodnight     5 he          3 his         5
horse      7 saying        5 to          5 was         5
```

5.3.1　保存派生类对象的指针

我们可能想在 set 或 multiset 容器中保存派生类对象的指针，可以通过将元素类型指定为基类对象类型的指针来做到这一点。这里主要担心的是比较函数，它必须可以比较指向不同类型的派生类对象的基类指针。通常我们会自己定义函数来做这件事，这没有任何难度。但如何去比较，取决于我们是否对元素顺序有任何要求。如果不在乎元素的排序方式，可以使用 owner_less<T>实例。但需要记住，检索元素要用指向相同对象的指针，而不是使用相等的对象。让我们思考一个示例。使用一个 multiset 容器，即使我们没有重复元素要保存，但会有一些不同类型的元素。

假设我们想在容器中保存每个人所拥有的宠物，这里宠物的类型是由一个派生于基类 Pet 的类定义的。这个类被定义在头文件 Pet_Classes.h 中，代码如下：

```
ussing std::string;

class Pet
{
protected:
  string name {};
```

```
public:
  virtual ~Pet(){}                          // Virtual destructor for base class
  const string& get_name() const { return name; }
  {
    auto result = std::strcmp(typeid(*this).name(), typeid(pet).name());
    return (result < 0) || ((result == 0) && (name < pet.name));
  }
  friend std::ostream& operator<<(std::ostream& out, const Pet& pet);
};
```

需要注意 Pet 类的成员函数 operator<()的定义。为了获得派生类的多态行为，它被指定为虚函数。这里使用了运算符 typeid，它会创建一个 type_info 对象，这个对象封装了操作数的类型。使用 typeid 需要包含 typeinfo 头文件。调用 type_info 对象的成员函数 name()会返回一个 C 风格的字符串，这是类型名实现定义的一种表示。在笔者的系统上，类型名加上了前缀 "class"，因此对于 My_Type 类型的对象，name()返回的是"class My_Type"，在你的系统上可能会有所不同。

用定义在 cstring 头文件中的 strcmp()来比较类型名字符串。如果第一个参数小于第二个参数，这个函数会返回一个负数；如果两个参数相等，返回 0，否则返回一个正数。operator<()函数会返回两个表达式或运算后的结果。如果第一个表达式为 true，这个函数总是返回 true。当前对象的类型名小于参数对象的类型名时，会出现这种情况，因而对象主要依靠类型来排序。当第一个表达式是 false 时，这个表达式的结果取决于第二个表达式的结果。当类型名字符串相等，且比较运算的左操作数 name 小于右操作数的成员 name 时，第二个表达式才为 true。

返回表达式中对相同类型名的比较非常重要。set 或 map 容器指定的比较运算必须是严格弱序的。在其他的条件中，这要求如果 a<b 为 true，那么 b<a 必须为 false。不比较两个类型名是否相等。这个返回值的表达式无法满足这个条件。当对保存派生类对象的容器进行排序时，这就可能会导致程序崩溃。你很容易明白这是如何产生的。假设将一个 name 为"Tiddles"的 Cat 对象 cat 和一个 name 为"Rover"的 Dog 对象 dog 比较，因为类型名，表达式 cat < dog 为 true；而表达式 dog < cat 也为 true，因为宠物名。这两个对象同时小于彼此可能会产生一些问题。

当然可以不使用 strcmp()，可以将 type_info 的成员函数 name()返回的 null 结尾的字符串转换为 string 文件类型，然后就可以用<运算符比较它们。

Pet_Classes.h 头文件中为输出流定义的插入运算如下：

```
inline std::ostream& operator<<(std::ostream& out, const Pet& pet)
{
  return out << "A " <<
    string {typeid(pet).name()}.erase(0,6) << " called " << pet.name;
}
```

这段代码输出了类型名和宠物名。类型名字符串的表达式首先将 C 风格的字符串转换为 string 类型，然后从 string 移除前 6 个字符"class"。如果你的系统使用的是不同的类型名，

需要对代码做一些修改。

为简单起见，定义三个派生于 Pet 的类：Cat、Dog 和 Mouse。除了类型不同外，它们的定义在本质上是相同的。Dog 类的定义如下：

```
class Dog : public Pet
{
public:
  Dog() = default;
  Dog(const string& dog_name)
  {
    name = dog_name;
  }
};
```

在这个构造函数中，初始化了继承的成员变量。所有派生类的定义都在 Pet 头文件中。

5.3.2　定义容器

用 multiset 容器保存 shared_ptr<Pet>对象，可以用两个 using 指令来为它们指定类型别名：

```
using Pet_ptr = std::shared_ptr<Pet>;  // A smart pointer to a pet
using Pets = std::multiset<Pet_ptr>;   // A set of smart pointers to pets
```

Pet_ptr 别名简化了 multiset 容器的定义，Pets 别名简化了 map 容器的定义。这里会用 map 容器来保存 multiset 容器，并以人名作为 map 的键。Pets 容器可以保存 Pet 对象的指针，也能保存 Cat、Dog、Mouse 对象的指针。

需要为 Pet_ptr 对象的 multiset 容器定义一个小于运算符：

```
inline bool operator<(const Pet_ptr& p1, const Pet_ptr& p2)
{
  return *p1 < *p2;
}
```

这里解引用传入的指针参数，然后将解引用得到的对象传入 Pet 派生类的虚函数 operator<()。在这个示例中，multiset 容器默认的函数对象 less<Pet_ptr>会调用上面这个函数。

这里有两个更有用的 using 用法：

```
using std::string;
using Name = string;
```

Name 的别名可以使 map 容器的键类型更加清楚明白。在 main()中按如下方式定义 map：

```
std::map<Name, Pets> peoples_pets;
```

容器中的元素是 pair<Name, Pets>对象，它的完整形式是 pair<string, multiset<shared_ptr<Pet>>，后面这种表示形式不是那么直观。

5.3.3 定义示例的 main()函数

用辅助函数从标准输入流读取人名和他们的宠物：

```
Pets get_pets(const Name& person)
{
  Pets pets;
  std::cout << "Enter " << person << "'s pets:\n";
  char ch {};
Name name {};
while(true)
{
  std::cin >> ch;
  if(toupper(ch) == 'Q') break;
  std::cin >> name;
  switch(std::toupper(ch))
  {
  case 'C':
    pets.insert(std::make_shared<Cat>(name));
    break;
  case 'D':
    pets.insert(std::make_shared<Dog>(name));
    break;
  case 'M':
    pets.insert(std::make_shared<Mouse>(name));
    break;
   default:
    std::cout << "Invalid pet ID - try again.\n";
   }
 }
  return pets;
}
```

代码虽然看起来很多，但都很简单。首先会创建一个 Pets 类型的局部 multiset 容器。将人名作为参数传入函数，作为输入提示，然后在一个死循环中读入他的宠物。用首字母来定义宠物的类型——'C'是猫，'D'是狗，以此类推。在 main()中会为这种表示方式创建一个提示。类型字符后面是宠物的名字，输入'Q'会结束当前输入。swich 语句会创建适当类型的 shared_ptr<T>对象，然后把它保存到 pets 容器中。当输入结束时，会以移动运算的方式返回局部对象 pets。

程序会输出 Pets 容器中的宠物，因此这里需要实现一个流插入运算符：

```
inline std::ostream& operator<<(std::ostream& out, const Pet_ptr& pet_ptr)
{
  return out << " " << *pet_ptr;
}
```

解引用智能指针，然后用插入运算符将对象写入流 out。因此这里会调用 Pet 类的友元

函数 operator<<()。为了输出 map 容器中的元素，在另一个函数的定义中使用它：

```
void list_pets(const std::pair<Name, Pets>& pr)
{
  std::cout << "\n" << pr.first << ":\n";
  std::copy(std::begin(pr.second), std::end(pr.second),
                                   std::ostream_iterator<Pet_ptr> (std::
                                      cout, "\n"));
}
```

元素是 pair 对象，它的第一个成员是人名，第二个成员是一个 multiset 容器，里面包含了指向宠物的指针。在将 pair 的第一个成员输出到标准输出流后，用 copy()算法输出第二个成员的元素。copy()的前两个参数都是迭代器，它们定义了拷贝元素的范围。第三个参数指定了拷贝操作的目的地，是一个 ostream_iterator<Pet_ptr>对象。然后调用 operator<<()函数，将 Pet_ptr 作为它的第二个参数，这个函数然后会调用 Pet 类的友元函数 operator<<()。

main()函数的代码可以从下载的 Ex5_05.cpp 中获取。下面是这个文件的内容：

```
// Ex5_05.cpp
// Storing pointers to derived class objects in a multiset container
#include <iostream>                    // For standard streams
#include <string>                      // For string class
#include <algorithm>                   // For copy() algorithm
#include <iterator>                    // For ostream_iterator
#include <map>                         // For map container
#include <set>                         // For multiset container
#include <memory>                      // For smart pointers
#include <cctype>                      // For toupper()
#include "Pet_Classes.h"

using std::string;
using Name = string;
using Pet_ptr = std::shared_ptr<Pet>; // A smart pointer to a pet
using Pets = std::multiset <Pet_ptr>; // A set of smart pointers to pets

// operator<() function to compare shared pointers to pets goes here...

// Stream insertion operator for pointers to pets goes here...

// get_pets() function to read in all the pets for a person goes here...

// list_pets() function to list the pets in a Pets container goes here...

int main()
{
  std::map<Name, Pets> peoples_pets; // The people and their pets
  char answer {'Y'};
  string name {};
  std::cout << "You'll enter a person's name followed by their pets.\n"
            << "Pets can be identified by C for cat, D for dog, or M for
               mouse.\n"
```

```
                << "Enter the character to identify each pet type followed by
                  the pet's name.\n"
                << "Enter Q to end pet input for a person.\n";
    while(std::toupper(answer) == 'Y')
    {
      std::cout << "Enter a name: ";
      std::cin >> name;
      peoples_pets.emplace(name, get_pets(name));
      std::cout << "Another person(Y or N)? ";
      std::cin >> answer;
    }
    // Output the pets for everyone
    std::cout << "\nThe people and their pets are:\n";
    for(const auto& pr : peoples_pets)
      list_pets(pr);
  }
```

在 main()中定义了人名和宠物的 map 容器后，会有一个提示来说明输入过程。while 循环控制所有输入。通过调用 people_pets 容器的成员函数 emplace()来向它添加元素，这个函数会在合适的位置创建元素。name 是它的第一个参数，第二个参数是 get_pets()返回的 multiset 容器。当输入结束时，通过迭代访问 map 元素的方式，输出人名及其宠物。用 map 容器的当前 pair 元素作为辅助函数 list_pets()的参数，输出每个人的相关信息。

下面是一些输出示例：

```
You'll enter a person's name followed by their pets.
Pets can be identified by C for cat, D for dog, or M for mouse.
Enter the character to identify each pet type followed by the pet's name.
Enter Q to end pet input for a person.
Enter a name: Jack
Enter Jack's pets:
d Rover c Tom d Fang m Minnie m Jerry c Tiddles q
Another person(Y or N)? y
Enter a name: Jill
Enter Jill's pets:
m Mickey d Lassie c Korky d Gnasher q
Another person(Y or N)? n

The people and their pets are:

Jack:
  A Cat called Tiddles
  A Cat called Tom
  A Dog called Fang
  A Dog called Rover
  A Mouse called Jerry
  A Mouse called Minnie

Jill:
  A Cat called Korky
```

```
A Dog called Gnasher
A Dog called Lassie
A Mouse called Mickey
```

宠物是以宠物类型名的字母升序输出的，这和我们期望的一样。输出表明我们成功在容器中保存了指向派生类的基类类型的智能指针。

5.4　unordered_set<T>容器

unordered_set<T>容器类型的模板定义在 unordered_set 头文件中。unordered_set<T>容器提供了和 unordered_map<T>相似的能力，但 unordered_set<T>可以用保存的元素作为它们自己的键。T 类型的对象在容器中的位置由它们的哈希值决定，因而需要定义一个 Hash<T>()函数。这种容器不能存放重复的元素。元素类型必须可以比较是否相等，因为这可以确定元素什么时候相等。就像 unordered_map，元素被存放在哈希表内部的格子中。每个格子保存哪个元素，是由元素的哈希值决定的。unordered_set 容器组织方式的概念图如图 5-5 所示：

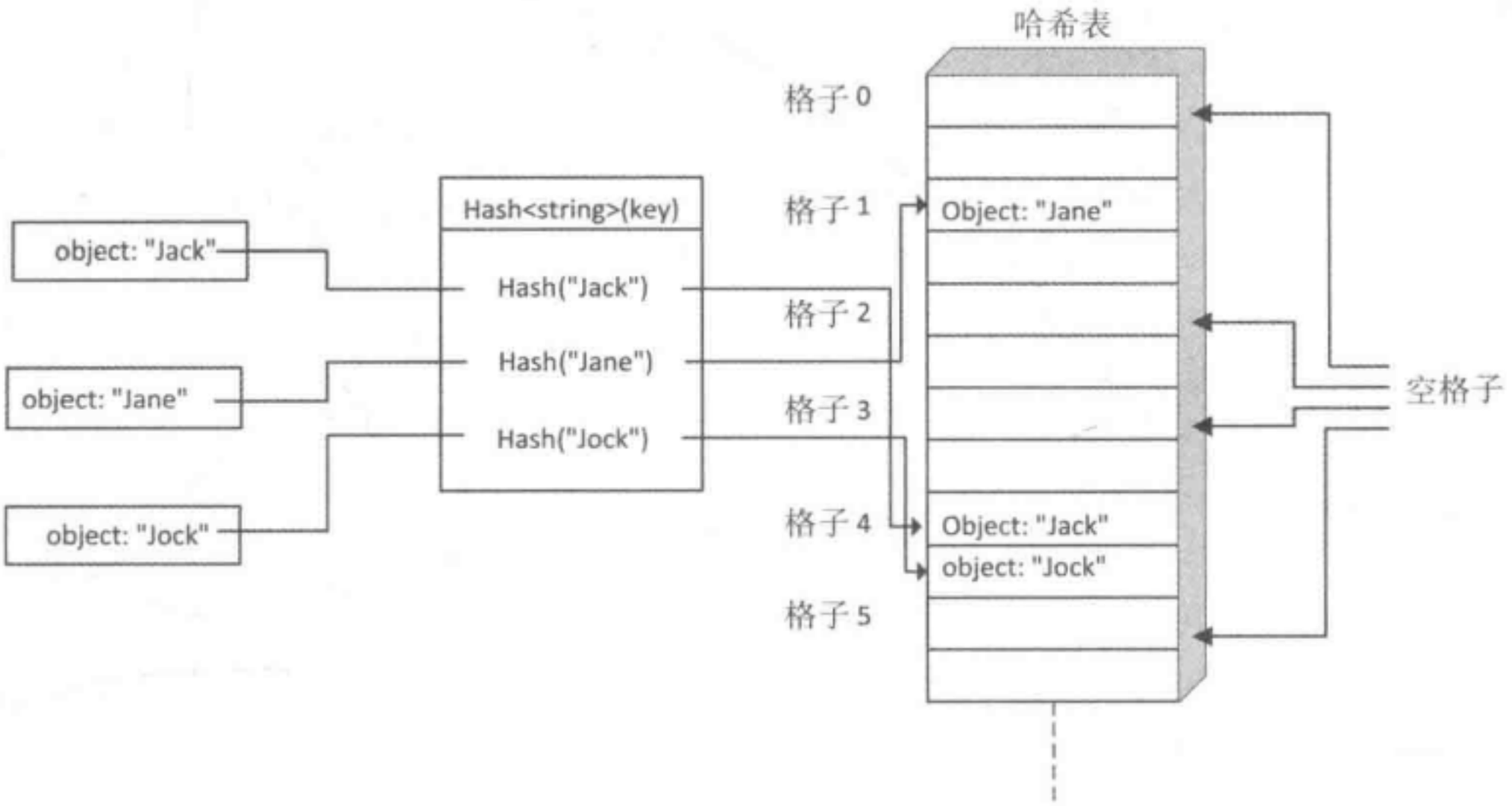


图 5-5　unordered_set 容器组织数据的方式

图 5-5 展示了一种情况，对于两个不同的对象"Jack"和"Jock"，它们的哈希值可能会使它们选择同一个格子。这里的格子个数是默认的，在创建容器时，可以修改。需要记住的是，正如我们在 unordered_map 中看到的那样，格子的个数通常是 2 的幂，这使我们可以更容易地从哈希值的比特数中选择格子。创建 unordered_set 的方式和 unordered_map 相似。下面是一些示例：

```
std::unordered_set<string> things {16};          // 16 buckets
std::unordered_set<string> words {"one", "two", "three", "four"};
                                                 // Initializer list
```

```
std::unordered_set<string> some_words {++std::begin(words), std::end
   (words)};                                          // Range
std::unordered_set<string> copy_wrds {words};  // Copy constructor
```

模板参数是默认的参数类型，指出了哈希函数的类型。当需要保存对象时，必须为它提供哈希函数，也需要为构造函数指出模板类型参数和函数参数。为了保存在 Ex4_01 中介绍的 Name 类型，需要按如下方式定义 unordered_set<Name>容器：

```
std::unordered_set<Name, Hash_Name> names {8, Hash_Name()};
// 8 buckets & hash function
```

上面第二个模板类型参数是用来哈希 Name 对象的函数对象类型。构造函数的第二个参数是这个函数对象的一个实例，当指定哈希函数时，需要同时指定格子的个数，因为它是构造函数的第一个参数。如果省略构造函数的第二个参数，容器会默认使用第二个模板类型参数的实例。如果 Hash_Name 是函数对象类型，就不再需要指定构造函数的第二个参数。

可以通过调用当前容器的成员函数 reserve()来增加它的格子个数。这可能会花费一些时间，因为这会对当前的元素重新哈希，然后将它们分配到新的格子中。

最大载入因子是每个格子所能容纳的最大元素个数。默认是 1.0，就像 unordered_map，也像 map，可以通过为 max_load_factor()传入一个新的载入因子来改变它。例如：

```
names.max_load_factor(8.0); // Max average no. of elements per bucket
```

通过增加最大载入因子，可以减少格子的使用个数，但会对元素访问的时间产生不利影响，因为增加访问元素会涉及格子的搜索可能性。

unordered_set 容器没有成员函数 at()，并且也没有定义下标运算，它和 unordered_map 有相同类型的成员函数。

5.4.1 添加元素

成员函数 insert()可以插入作为参数传入的单个元素。在这种情况下，它会返回一个 pair 对象，这个 pair 对象包含一个迭代器，以及一个附加的布尔值用来说明插入是否成功。如果元素被插入，返回的迭代器会指向新元素；如果没有被插入，迭代器指向阻止插入的元素。可以用一个迭代器作为 insert()的第一个参数，它指定了元素被插入的位置，如果忽略插入位置，在这种情况下，只会返回一个迭代器。另一个版本的 insert()函数可以插入初始化表中的元素，在这种情况下，什么都没有返回。

下面是一些说明性语句：

```
auto pr = words.insert("ninety");          // Returns a pair - an
                                           // iterator & a bool value
auto iter = words.insert(pr.first, "nine"); // 1st arg is a hint. Returns
                                           // an iterator
words.insert({"ten", "seven", "six"});     // Inserting an initializer list
```

当调用 insert()插入一段元素时，什么都不返回：

```
std::vector<string> more {"twenty", "thirty", "forty"};
words.insert(std::begin(more), std::end(more));
// Insert elements from the vector
```

unordered_set 容器的成员函数 emplace()和 emplace_hint()可以在容器的适当位置创建元素。正如我们之前所见的 set 容器，传入 emplace()的参数会被传入元素的构造函数，用来创建元素。emplace_hint()的迭代器参数可以指定元素的插入位置，后面是构造元素需要的参数。例如：

```
std::unordered_set<std::pair<string, string>, Hash_pair> names;
auto pr = names.emplace("Jack", "Jones"); // Returns pair<iterator, bool>
auto iter = names.emplace_hint(pr.first, "John", "Smith");
// Returns an iterator
```

容器的元素是用来表示名称的 pair 对象，这里的每个名称由两个 string 对象组成，它们分别表示一个人的姓和名。unordered_set<T>元素默认的哈希函数是一个 hash<T>类模板的实例。这个模板对基本类型、指针、string 对象有一些特例化的定义。

因为没有 hash<pair<string,string>>模板的特性化定义，所以需要定义一个哈希函数来哈希元素。这里将它的类型指定为-Hash_pair，它也是模板的第二个类型参数。emplace()的姓名参数会被传入 pair 的构造函数中，emplace_hint()使用了一个指向先前插入元素的迭代器，这一点可能会被忽略。后面的参数是 pair 构造函数的参数。函数对象类型 Hash_pair 可以用来对 names 容器中的元素进行哈希，可以按如下方式定义它：

```
class Hash_pair
{
public:
  size_t operator()(const std::pair<string, string>& pr)
  {
    return std::hash<string>()(pr.first + pr.second);
  }
};
```

这里使用了一个定义在 string 头文件中的 hash<string>函数对象的实例。它会哈希一个由 pair 对象的第一个成员和第二个成员串联的字符串，然后将结果作为这个 pair 元素的哈希值返回。

5.4.2　检索元素

调用 unordered_set 的 find()会返回一个迭代器。这个迭代器指向和参数哈希值匹配的元素，如果没有匹配的元素，会返回这个容器的结束迭代器。例如：

```
std::pair<string, string> person {"John", "Smith"};
if(names.find(person) != std::end(names))
  std::cout << "We found " << person.first << " " << person.second
            << std::endl;
else
```

```
std::cout << "There's no " << person.first << " " << person.second
          << std::endl;
```

这个容器是从上一节获取的，这段代码会报告 John Smith 是存在的。如果不存在，find()会返回这个容器的结束迭代器，并且会执行第二条输出语句。

unordered_set 容器中的元素是无序的，因此也不需要成员函数 upper_bound()和 lower_bound()。成员函数 equal_range()会返回一个以迭代器为成员的 pair，它指定了和参数匹配的一段元素。unordered_set 容器中只可能有一个匹配元素。如果没有，两个迭代器都是容器的结束迭代器。调用成员函数 count()会返回容器中参数的出现次数。对于 unordered_set 只可能是 0 或 1。当想要知道容器中总共有多少元素时，可以调用成员函数 size()。如果容器中没有元素，成员函数 empty()会返回 true。

5.4.3 删除元素

调用 unordered_set 容器的成员函数 clear()可以删除它的全部元素。成员函数 erase()可以删除容器中和传入参数的哈希值相同的元素。另一个版本的 erase()函数可以删除迭代器参数指向的元素。例如，这里不需要大费周章地来删除容器中的元素：

```
std::pair<string, string> person {"John", "Smith"};
auto iter = names.find(person);
if(iter != std::end(names))
  names.erase(iter);
```

erase()迭代器的参数必须是一个指向容器中元素的、有效的、可解引用的迭代器，因此需要确保它不是容器的结束迭代器。这个版本的 erase()函数会返回一个指向被删除元素的下一个位置的迭代器，如果删除的是最后一个元素，那么它就是结束迭代器。

如果 person 对象存在，可以像下面这样很容易地删除它：

```
auto n = names.erase(person);
```

这个版本的 erase()会返回一个 size_t 类型的数作为被删除元素的个数。在这种情况下，这个值只能是 0 或 1，但对于 unordered_multiset 容器来说，这个值可能会大于 1。显然，如果返回值是 0，那么容器中肯定没有这个元素。

尽管最初的示例可能没什么用，但当需要删除一些有特定字符的元素时，可以调用 erase()来删除迭代器所指向的元素，这是很有用的。假设需要移除 names 容器中名称以字符'S'开始的所有元素，那么下面这个循环就可以实现：

```
while(true)
{
  auto iter = std::find_if(std::begin(names), std::end(names),
[](const std::pair<string, string>& pr ){ return pr.second[0] == 'S';});
  if(iter == std::end(names))
    break;
  names.erase(iter);
}
```

find_if()算法的前两个参数定义了一个元素段的范围，它会找到这段元素中第一个可以使第三个参数返回 true 的元素，然后返回这个元素的迭代器。断言的参数必须是从解引用元素段得到的类型对象。这里的元素段是 names 容器中的全部元素，它们都是 pair<string, string>对象，断言是一个 lambda 表达式，当 pair 的第二个成员的首字母是'S'时。当没有元素使 lambda 返回 true 时，算法会返回这段元素的结束迭代器。

这里有另一个版本的 erase()，它可以移除一段元素。下面的语句可以移除除了第一个和最后一个元素之外的所有元素：

```
auto iter = names.erase(++std::begin(names), --std::end(names));
```

这个函数的参数是两个迭代器，它们指定了所删除元素的范围。这个函数会返回一个迭代器，它指向最后一个被删除元素后面的元素。

5.4.4　创建格子列表

可以用在 unordered_map 中看到的函数访问 unordered_set 容器中的格子。通过迭代器，我们可以访问存放在特定格子中的元素。可以通过将格子的索引传入容器的成员函数 begin()和 end()来选择特定的格子，它们会返回这个格子所包含元素段的开始和结束迭代器。当需要 const 迭代器时，可以将格子的索引传给容器的成员函数 cbegin()和 cend()。成员函数 bucket_count()返回格子的个数，所以可以用它来控制对容器格子的循环遍历。下面展示如何列出 names 容器中每个格子中的元素：

```
for(size_t bucket_index {}; bucket_index < names.bucket_count();
  ++bucket_index)
{
  std::cout << "Bucket " << bucket_index << ":\n";
  for(auto iter = names.begin(bucket_index); iter != names.end(bucket_index);
      ++iter)
  {
    std::cout << " " << iter->first << " " << iter->second;
  }
  std::cout << std::endl;
}
```

外循环遍历格子的索引。内循环遍历当前格子中的元素，然后将 pair 对象的第一个成员和第二个成员写入输出流。

成员函数 bucket_size()会返回索引指定的格子中的元素个数。可以通过将格子作为容器的成员函数 bucket()的参数，来获取这个格子在容器中的索引值。假定传入的对象不在容器中，如果将对象插入，函数会返回它所对应的格子的索引值，因此这里不能通过 bucket()来判断对象是否真的在容器中。下面展示如何列出 names 容器中的元素，以及元素所包含的成员：

```
for(const auto& pr : names)
std::cout <<pr.first << " " <<pr.second << " is in bucket " << names.bucket(pr)
          << std::endl;
```

■ 注意：如果不传入参数，就调用 begin()、cbegin()、end()、cend()，它们会返回对应的元素的开始和结束迭代器。

5.5 使用 unordered_multiset<T>容器

除了不能在 unordered_multiset<T>中保存重复的元素，它在本质上和 unordered_set<T>相似。前面讲过的所有关于 unordered_set 的成员函数都同样适用于 unordered_multiset，除了那些会被重复元素所影响的。成员函数 count()的返回值大于 1，而且调用成员函数 erase()会删除所有和参数具有相同哈希值的元素，而不是删除单个元素。让我们通过实际操作来看一个 unordered_multiset 容器的示例。

Ex5_06 这个示例会保存一些 Name 类变量的实例，Name 类是定义在 Ex4_01 中的 unordered_multiset 容器。这个容器用来记录所有的朋友，因此会知道节假日该送给谁贺卡。因为那时候邮费很贵，所以列表出于经济考虑应该越短越好。本例中 Name.h 头文件中的内容如下：

```
// Name.h for Ex5_06
// Defines a person's name
#ifndef NAME_H
#define NAME_H
#include <string>                          // For string class
#include <ostream>                         // For output streams
#include <istream>                         // For input streams
using std::string;

class Name
{
private:
  string first {};
  string second {};

public:
  Name(const string& name1, const string& name2) : first (name1), second
    (name2) {}
  Name() = default;

  const string& get_first() const { return first; }
  const string& get_second() const { return second; }

  size_t get_length() const { return first.length() + second.length() + 1; }

  // Less-than operator
  bool operator<(const Name& name) const
  {
    return second < name.second || (second == name.second && first < name.
      first);
  }

  // Equality operator
```

```
  bool operator==(const Name& name) const
  {
    return (second == name.second) && (first == name.first);
  }

  size_t hash() const { return std::hash<std::string>()(first+second); }

  friend std::istream& operator>>(std::istream& in, Name& name);
  friend std::ostream& operator<<(std::ostream& out, const Name& name);
};
// Extraction operator overload
inline std::istream& operator>>(std::istream& in, Name& name)
{
  in >> name.first >> name.second;
  return in;
}
// Insertion operator overload
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{
  out << name.first + " " + name.second;
  return out;
}
#endif
```

上述代码在Ex4_01版本的Name类的基础上添加了一些用来访问类的数据成员的成员函数，它们分别是：可以获取姓名长度的成员函数 get_length()，用来比较对象是否相等的成员函数 operator==()，以及为两个成员变量串接的字符哈希创建哈希值的 hash()。等价比较是这种容器类型所需要的。get_length()可以使输出更整齐。成员函数 length()返回的长度是这两个名称的总长度加 1，这里加 1 是为了在输出之间加 1 个空格。在 operator<<()的定义中，将姓名串联输出是为了可以通过设置输出宽度来使名称对齐。使用多个<<运算符可以提高输出的效率，但不能保证输出的名称是对齐的。容器类型需要将用来对对象哈希的函数对象类型作为模板参数。这个函数对象类型定义在 Hash_Name.h 头文件中，内容如下：

```
// Hash_Name.h
// Function object type to hash Name objects for Ex5_06
#ifndef HASH_NAME_H
#define HASH_NAME_H
#include "Name.h"

class Hash_Name
{
public:
  size_t operator()(const Name& name) { return name.hash(); }
};
#endif
```

这个函数叫作运算符函数，它会调用传给它的 Name 对象的成员函数 hash()。

用 vector 中的元素来合成容器中的 Name 对象。假设下面的类型别名有效，就可以很

方便用另一个版本的辅助函数来填充 unordered_multiset 容器。

```
using Names = std::unordered_multiset<Name, Hash_Name>;
```

unordered_multiset 的第一个模板参数类型是元素的类型，第二个是用来哈希元素的函数对象类型。下面是用于在容器中创建元素的辅助函数：

```
void make_friends(Names& names)
{
  // Names are duplicated to get duplicate elements
  std::vector<string> first_names {"John", "John", "John", "Joan", "Joan",
   "Jim", "Jim", "Jean"};
  std::vector<string> second_names {"Smith", "Jones", "Jones", "Hackenbush",
   "Szczygiel"};
  for(const auto& name1 : first_names)
    for(const auto& name2:second_names)
      names.emplace(name1,name2);
}
```

嵌套循环会在 names 容器中创建元素。这里会用所有可能的姓名组合来创建对象。有一些姓和名是重复的，可以保证我们能够创建相同的元素。

这个程序会通过列出容器中格子的内容来展示哪些朋友是放在同一个格子中的，另一个版本的辅助函数可以做到这些：

```
void list_buckets(const Names& names)
{
  for(size_t n_bucket {} ; n_bucket < names.bucket_count(); ++n_bucket)
  {
    std::cout << "Bucket " << n_bucket << ":\n";
    std::copy(names.begin(n_bucket), names.end(n_bucket), std::ostream_
              iterator<Name>(std::cout, " "));

    std::cout << std::endl;
  }
}
```

for 循环遍历格子的索引值。在每一次循环迭代中，头一行用来指出标识的格子，然后 copy()算法输出这个格子中的所有元素。copy()算法将每个迭代器所指向的 Name 元素复制到 ostream_iterator，它会将元素写到 cout 中。operator<<()函数的重载版本定义在 Name.h 头文件中，可以将 Name 对象写到 ostream 对象中。

包含 main()函数的源文件的代码如下：

```
// Ex5_06.cpp
// Using an unordered_multiset container
#include <iostream>           // For standard streams
#include <iomanip>            // For stream manipulators
#include <string>             // For string class
#include <unordered_set>      // For unordered_multiset containers
```

```
#include <algorithm>          // For copy(), max(), find_if(), for_each()
#include "Name.h"
#include "Hash_Name.h"

using std::string;
using Names = std::unordered_multiset<Name, Hash_Name>;

// Code for make_friends(Names& names) goes here...

// Code for list_buckets() goes here...

int main()
{
  Names pals {8};                  // 8 buckets
  pals.max_load_factor(8.0);       // Average no. of elements per bucket max
  make_friends(pals);              // Load up the container with Name objects
  list_buckets(pals);              // List the contents by bucket
  // Report the number of John Smith's that are pals
  Name js {"John", "Smith"};
  std::cout << "\nThere are " << pals.count(js) << " " << js << "'s.\n"
            << std::endl;

  // Remove all the John Jones's - we just don't get on...
  pals.erase(Name {"John", "Jones"});

  // Get rid of the Hackenbushes - they never invite us...
  while(true)
  {
    auto iter = std::find_if(std::begin(pals), std::end(pals),
      [](const Name& name){ return name.get_second() == "Hackenbush"; });
    if(iter == std::end(pals))
      break;
    pals.erase(iter);
  }

  // List the friends we still have...
  size_t max_length {};            // Stores the maximum name length
  std::for_each(std::begin(pals), std::end(pals), // Find the maximum name
                                                  // length...
    [&max_length](const Name name){ max_length = std::max(max_length, name.
      get_length()); });

  size_t count {};                 // No. of names written out
  size_t perline {6};              // No. of names per line
  for(const auto& pal : pals)
  {
    std::cout << std::setw(max_length+2) << std::left << pal;
    if((++count % perline) == 0) std::cout << "\n";
  }
  std::cout << std::endl;
}
```

这个容器初始时有 8 个格子。如果超出了最大转载因子，容器的格子个数会自动增长，因而调用 max_load_factor()函数将装载因子设为 8.0 可以大大减少出现这种情况的可能性。除了这里展示的这种作用，还可以减少列出格子时的输出行数。默认的最大装载因子是 1.0，在笔者的系统里，格子的个数达到 64 个，这会产生很多输出。实际上，在这里增加最大装载因子会明显减慢操作，因为会增加搜索必要格子的次数，这会抵消 unordered_multiset 相对 multiset 的一些优势。

调用 make_friends()可以用全部的姓名组合在容器中创建 Name 元素，调用 list_buckets()会输出每个格子的元素。这个程序然后会通过调用容器的函数 count()来输出朋友中所有名叫"John Smith"的人数。

通过将对应的对象传给容器的成员函数 erase()，删除了所有名叫"John Jones"的朋友。记得名叫 Hackenbushe 的没有一个人请我喝过咖啡，所以我决定把他也删掉。这需要一点技巧，因为我们需要找到容器元素的第二个成员。find_if()算法可以实现这一点，它会返回一个使第三个参数(即 lambda 表达式)返回 true 的元素的迭代器。find_if()返回的迭代器保存在 iter 中，它是在循环中用 auto 定义的，因此类型可以推导。如果在循环外使用这个 iter 的引用，需要将它定义为 Name::iterator。iterator 是 unordered_multiset<Name>容器中元素的迭代器的别名，它被定义在模板中。循环继续进行，直到 find_if()返回容器的结束迭代器，这表明容器中没有叫 Hackenbushe 的元素。最后，我们用 for 循环来遍历输出剩下的全部朋友。每一行的字段宽度值由 for_each()算法决定。算法会解引用前两个参数指定范围内的元素，然后将结果传给 lambda 表达式。这个 lambda 表达式最后会在 max_length 中保存姓名的最大长度，max_length 是通过引用捕获的。

可以用 copy()算法输出元素:

```
std::copy(std::begin(pals), std::end(pals), std::ostream_iterator<Name>
  {std::cout, "\n"});
```

它会在每一行输出一个元素，这会输出很多行。可以将它们全部输出到一行，但它们到 cout 的输出结果可能不会太令人满意。用 copy()更倾向于输出到文件中。

下面是笔者系统上的输出:

```
Bucket 0:
Joan Jones Joan Jones Joan Jones Joan Jones
Bucket 1:
Joan Szczygiel Joan Szczygiel
Bucket 2:
Jean Jones Jean Jones Jean Smith
Bucket 3:
Jim Szczygiel Jim Szczygiel Jim Hackenbush Jim Hackenbush John Jones John
Jones John Jones John Jones John Jones John Jones John Jones
Bucket 4:
Joan Smith Joan Smith John Hackenbush John Hackenbush John Hackenbush
Bucket 5:
Joan Hackenbush Joan Hackenbush
Bucket 6:
```

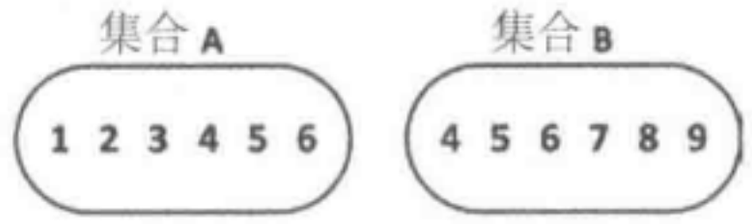
```
    Jim Jones Jim Jones Jim Jones Jim Jones Jim Smith Jim Smith John Szczygiel
John Szczygiel John Szczygiel John Szczygiel
    Bucket 7:
    Jean Szczygiel Jean Hackenbush John Smith John Smith John Smith
    There are 3 John Smith's.
    Jean Szczygiel John Smith John Smith     John Smith     Jim Szczygiel      Jim Szczygiel
    Joan Smith     Joan Smith Jim Jones      Jim Jones      Jim Jones          Jim Jones
    Jim Smith      Jim Smith  John Szczygiel John Szczygiel John SzczygielJoan Jones
    Joan Jones     Joan Jones Joan Jones     Joan Szczygiel Joan Szczygiel     JeanJones
    Jean Jones     Jean Smith
```

在你的系统上，姓名在格子中的分布可能会有些不同。这取决于所选格子的哈希值有多少位。每个格子会包含一些元素，Bucket 3 包含 10 个元素。直到每个格子保存的元素平均值超过 8，格子的个数才会增加。当然，以相同名字结尾的朋友都在同一个格子中，因为它们有相同的哈希值。

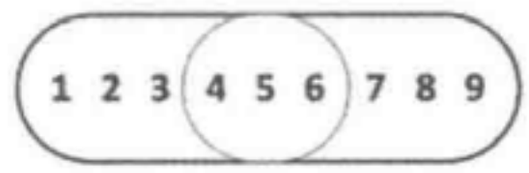
5.6　集合运算

集合的数学概念和 set 容器很像，它们都是在某些方面相似的事物的集合。集合定义的二元操作能够以不同的方式组合两个集合中的内容，然后创建一个新的集合。图 5-6 展示了这些操作，展示的操作如下：

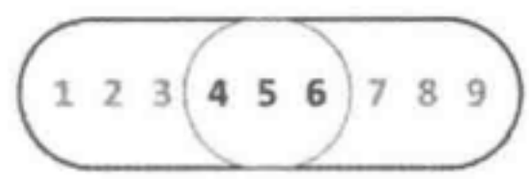
- 集合 A 和 B 包含一些整数
- 集合 A 和 B 的并集包含这两个集合的元素
- 集合 A 和 B 的交集包含的元素是这两个集合共有的
- 集合 A 和 B 的差集是从集合 A 中移除共有元素后的集合
- 集合 A 和 B 的对称差集是不包含任何共同元素的集合



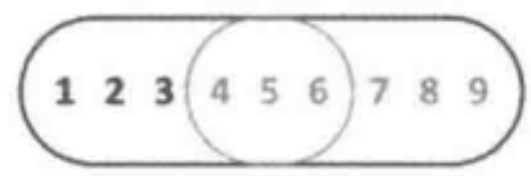
- A和B的并集：A∪B



- A和B的交集：A∩B



- A和B的差集：A\B



- A和B的对称差集：A△B

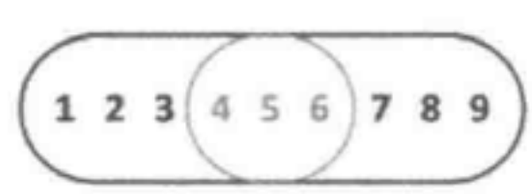


图 5-6　集合运算

图 5-6 中运算结果后的元素是用粗体表示的。如果以前没有接触过这些运算，它们看起来可能有些抽象，但却是很有用的。在 Ex5_01 中，我们需要为课程分配一些学生，每门课程的学生都被保存在 set 中。大学老师可能会对那些选择了物理却没有申请数学课程的学生感兴趣。对这两门课程的学生集合做差集就可以立即得到答案，在后面的一个示例中你会看到如何操作。

STL 提供了几个实现了集合对象运算的算法，包括图 5-6 中展示的 4 个二元运算，它们都定义在 algorithm 头文件中。这些函数不需要涉及 set 容器，尽管它们可以。两个迭代器指定的对象集合作为元素段传给这些算法。代表集合的这段元素必须是有序的。默认是升序，但必要时也可以改变。这是因为当对象有序时，这些运算的执行性能是线性的。这些运算中并没有包含排序，这是因为在很多情况下，包含排序是没有必要的。显然，一些类型的容器，例如 set 和 map，它们的元素都是有序的，就符合这种情况。set 运算的全部算法需要集合有相同的顺序，升序或降序。从任何 STL 算法创建的对象集合都是原集合对象的副本。

集合算法不能用在无序关联容器的元素上，因为它的元素是无序的。排序需要元素的随机访问迭代器，而无序关联容器没有这种迭代器。然而，我们可以将它的元素复制到另一种类型的容器里，例如 vector，然后对 vector 中的元素排序，最后对它使用集合运算。稍后会解释集合运算的每一种 STL 算法，然后在示例中展示它们。

5.6.1 set_union()算法

第一个版本的 set_union()函数模板实现了集合的并集运算，它需要 5 个参数：两个迭代器用来指定左操作数的集合范围，另两个迭代器用来作为右操作数的集合范围，还有一个迭代器用来指向结果集合的存放位置。例如：

```
std::vector<int> set1 {1, 2, 3, 4, 5, 6};
std::vector<int> set2 {4, 5, 6, 7, 8, 9};
std::vector<int> result;
std::set_union(std::begin(set1), std::end(set1), // Range for set that is
                                                 // left operand
               std::begin(set2), std::end(set2), // Range for set that is
               // right operand
               std::back_inserter(result));      // Destination for the
               // result:1 2 3 4 5 6 7 8 9
```

set1 和 set2 中的初始值都是升序。如果它们都不是，那么在使用 set_union()算法前，需要对 vector 容器排序。在第 1 章介绍的 back_inserter()函数模板定义在 iterator 头文件中，它会调用传入参数的函数 push_back()来返回一个 back_inserter_iterator 对象。所以，set1 和 set2 并集中的元素会被保存在 result 中。从并集运算得到的元素集合是容器元素的副本，因此这个运算并不会影响容器的原始内容。

当然，如果不保存运算结果；可以用一个流迭代器输出这些元素：

```
std::set_union(std::begin(set1), std::end(set1), std::begin(set2),
  std::end(set2),std::ostream_iterator<int> {std::cout, " "});
```

这里的目的地址是一个 ostream_iterator，它可以将结果输出到标准输出流中。

第二个版本的 set_union()函数模板接收的第 6 个参数是一个用来比较集合元素的函数对象。下面是它可能的一些用法：

```
std::set<int, std::greater<int>> set1 {1, 2, 3, 4, 5, 6}; // Contains 6 5
                                                          // 4 3 2 1
std::set<int, std::greater<int>> set2 {4, 5, 6, 7, 8, 9}; // Contains 9 8
                                                          // 7 6 5 4
std::set<int, std::greater<int>> result; // Elements in descending sequence
std::set_union(std::begin(set1), std::end(set1),std::begin(set2),
 std::end(set2),
std::inserter(result, std::begin(result)),  // Result destination: 9 8 7 6
                                            // 5 4 3 2 1
                      std::greater<int>()); // Function object for
                                            // comparing elements
```

这一次的集合是 set 容器中的元素。这些元素是用函数对象 greater<int>排序过的，因此它们都是升序。set_union()的最后一个参数是一个用来比较集合元素的 greater<int>类型的实例。结果的存放位置是 result 容器的 inserter_iterator，容器会调用成员函数 insert()来添加元素。不能对 set 容器使用 back_insert_iterator，因为它没有成员函数 push_back()。并集运算的结果是从两个集合得到的元素的副本的降序集合。

这两个版本的 set_union()函数都会返回一个指向被复制元素段末尾后一个位置的迭代器。如果目的容器包含操作前的元素，这是很有用的。例如，如果目的容器是 vector 容器，set_union()用 front_insert_iterator，如果用 back_inserter_iterator，可以用这个容器的结束迭代器，插入这个新元素，set_union()返回的迭代器会指向第一个原始元素。

5.6.2　set_intersection()算法

除了会创建两个集合的交集而不是并集之外，set_intersection()算法的用法和 set_union()相同。有两个版本的 set_intersection()，它们和 set_union()拥有相同的参数集。下面的一些语句可以说明它的用法：

```
std::set<string> words1 {"one", "two", "three", "four", "five", "six"};
std::set<string> words2 {"four", "five", "six", "seven", "eight", "nine"};
std::set<string> result;
std::set_intersection(std::begin(words1), std::end(words1),
                      std::begin(words2), std::end(words2),
                      std::inserter(result, std::begin(result)));
                                        // Result: "five" "four" "six"
```

这个 set 容器保存 string 对象，默认使用 less<string>的实例对元素排序。两个容器中元素的交集是它们共有的元素，它们被保存在 result 容器中。当然，这些元素是升序字符串序列。set_intersection()算法会返回一个迭代器，它指向目的容器中插入的最后一个元素的下一个位置。

5.6.3 set_difference()算法

set_difference()算法可以创建两个集合的差集，它也有两个版本的函数并且参数集和set_union()相同。下面是一个对降序 set 容器使用这个算法的示例：

```
    std::set<string, std::greater<string>> words1 {"one", "two", "three",
"four", "five", "six"};
    std::set<string, std::greater<string>> words2 {"four", "five", "six",
"seven", "eight", "nine"};
    std::set<string, std::greater<string>> result;
    std::set_difference(std::begin(words1), std::end(words1),
                        std::begin(words2), std::end(words2),
                        std::inserter(result, std::begin(result)),
                        // Result:"two" "three" "one"
                        std::greater<string>());
                        // Function object to compare elements
```

这里调用的这个版本的函数的第 6 个参数是一个用来比较元素的函数对象，因为来自 set 容器的这段元素也用这个函数排序。通过从 words 集合中移除 word1 和 word2 共有的元素来获取差集，差集由来自 word1 的元素组成。结果得到 words1 的前三个元素的降序序列。这个算法也会返回一个迭代器，它指向目的容器被插入的最后一个元素的下一个位置。

5.6.4 set_symmetric_difference()算法

set_symmetric_difference()算法和先前的集合算法遵循同一种模式。下面的几条语句展示了它的用法：

```
std::set<string> words1 {"one", "two", "three", "four", "five", "six"};
std::set<string> words2 {"four", "five", "six", "seven", "eight", "nine"};
std::set_symmetric_difference(std::begin(words1), std::end(words1),
                              std::begin(words2), std::end(words2),
                              std::ostream_iterator<string>{std::cout, " "});
```

这个范围内的元素默认是升序排列的。集合的对称差集中的元素是两个集合中不包括它们共有元素的元素。最后一个函数的参数定义了结果集的存放位置，它是一个 ostream_iterator，因此这些元素会被写入 cout，输出内容如下：

```
eight nine one seven three two
```

自然地，这些序列中的 string 元素都是对它们使用<运算符后得到的结果，因为默认的比较函数对象类型是 less<string>。

5.6.5 includes()算法

include()算法可以比较两个元素的集合，如果第一个集合中的全部元素都来自第二个集合，它会返回 true。如果第二个集合是空的集合，它也返回 true。下面是一些示例：

```
std::set<string> words1 {"one", "two", "three", "four", "five", "six"};
std::set<string> words2 {"four", "two", "seven"};
std::multiset<string> words3;
std::cout << std::boolalpha
          << std::includes(std::begin(words1), std::end(words1),
                           std::begin(words2), std::end(words2))
          << std::endl;                          // Output: false
std::cout << std::boolalpha
          << std::includes(std::begin(words1), std::end(words1),
             std::begin(words2), std::begin(words2))
          << std::endl;                          // Output: true

std::set_union(std::begin(words1), std::end(words1), std::begin(words2),
 std::end(words2),std::inserter(words3, std::begin(words3)));
std::cout << std::boolalpha
          << std::includes(std::begin(words3), std::end(words3),
                           std::begin(words2), std::end(words2))
          << std::endl;                          // Output: true
```

这里有两个 string 元素的 set 容器—— words1 和 words2，它们都是用初始化列表初始化的。第一条输出语句显示 false，因为 word1 不包含 word2 中的 string("seven")元素。第二条输出语句显示 true，因为第二个操作数指定的集合是空的—— 它的开始迭代器和结束迭代器相同。set_union()函数会通过 inserter_iterator 将 words1 和 words2 中的并集复制 word3 中。结果 word3 会包含 words2 中的全部元素，因而第三条输出语句会显示 true。

当容器是 multiset 时，会很容易对它们并集运算后的结果感到困惑。尽管 words3 是一个允许包含重复元素的 multiset，但 words1 和 words2 共有的元素并没有重复出现在 words3 中。下面的语句会输出 words3 中的元素：

```
std::copy(std::begin(words3), std::end(words3),
                              std::ostream_iterator<string> {std::cout, " "});
```

输出结果如下：

```
five four one seven seven six three two
```

这是因为并集运算只包含每个重复元素的一个副本。当然，如果 words1 和 words2 是包含重复单词的 multiset 容器，那么结果可能会包含一些重复元素：

```
std::multiset<string> words1 {"one", "two", "nine", "nine", "one", "three",
"four", "five", "six"};
std::multiset<string> words2 {"four", "two", "seven", "seven", "nine",
"nine"};
std::multiset<string> words3;
```

"one"是 words1 中的重复元素，"seven"是 words2 中的重复元素。"nine"在这两个容器中都是重复的。现在可以指向相同的 set_union()：

```
std::set_union(std::begin(words1), std::end(words1),
               std::begin(words2), std::end(words2),
               std::inserter(words3, std::begin(words3)));
```

输出的 words3 的内容如下：

```
five four nine nine one one seven seven six three two
```

并集结果中会有一个或多个重复元素，对于在两个集合中都单独出现的元素，并集运算中不会有重复的元素。当然，如果元素在两个集合中都重复，那么它们在结果中也是重复的。

5.6.6 集合运算的运用

可以看到集合运算在 Ex5_01 中的扩展，代码可以从下载的 Ex5_07 中获取。新的代码会在 Ex5_01 的 main()的末尾添加，因此这里只展示附加的代码。下面的代码可以发现哪些学生选择了物理课却没有选择数学课：

```
auto physics = courses.find("Physics");
auto maths = courses.find("Mathematics");
if(physics == std::end(courses) || maths == std::end(courses))
  throw std::invalid_argument {"Invalid course name."};
std::cout << "\nStudents studying physics but not maths are:\n";
std::set_difference(std::begin(physics->second), std::end(physics-> second),
                    std::begin(maths->second), std::end(maths->second),
std::ostream_iterator < Student > {std::cout, " "});
std::cout << std::endl;
```

调用 courses 容器的成员函数 find()会返回一个迭代器，它指向一个键和参数匹配的元素。如果没有匹配的键，函数会返回一个结束迭代器。因此，检查它的返回值是一个好习惯。这里知道如果键值参数拼写错了，会导致代码失败。如果发生了这种情况，会抛出一个标准异常来结束程序。set_difference()算法会产生我们期望的结果，在这个示例中，方式是通过 ostream_iterator 对象将 Student 对象写入 cout。当然，也可用另一个容器来保存结果集，后面的代码会展示具体操作。

下面的代码可以标识出那些既学数学又学物理的学生：

```
std::vector<Student> phys_and_math;
std::cout << "\nStudents studying physics and maths are:\n";
std::set_intersection(std::begin(physics->second),
                      std::end(physics->second),
                      std::begin(maths->second), std::end(maths->second),
                      std::back_inserter(phys_and_math));
std::copy(std::begin(phys_and_math), std::end(phys_and_math),
                      std::ostream_iterator <Student> {std::cout, " "});
 std::cout << std::endl;
```

set_intersection()算法会返回两个集合的交集，这是我们想要的。通过使用 back_inserter()返回的 phys_and_math 的 insert_iterator 可以将得到的元素插入 vector 容器。也可以使用 vector 的 front_insert_iterator，然后用 copy 算法输出 vector 的内容。

只要保存了结果集，就可以对结果集做进一步的集合运算。下面展示了如何筛选出同时申请了物理、数学和天文课的学生：

```
auto astronomy = courses.find("Astronomy");
if(astronomy == std::end(courses)) throw std::invalid_argument{"Invalid
  course name."};
std::cout << "\nStudents studying physics, maths, and astronomy are:\n";
std::set_intersection(std::begin(astronomy->second),
                      std::end(astronomy- >second),
                      std::begin(phys_and_math), std::end(phys_and_math),
                      std::ostream_iterator<Student>{std::cout, " "});
  std::cout << std::endl;
```

对上一个运算得到的集合和选择了天文课的学生的集合求交集。在笔者的系统上，交集中只有一个学生。

下面可以得到选择了戏剧或哲学课的学生，但是一个人都没有：

```
auto drama = courses.find("Drama");
auto philosophy = courses.find("Philosophy");
if(drama == std::end(courses) || philosophy == std::end(courses))
   throw std::invalid_argument{"Invalid course name."};

Group act_or_think;                          // set container for result
std::cout << "\nStudents studying either drama or philosophy are:\n";
std::set_symmetric_difference(std::begin(drama->second),
                              std::end(drama->second),
                              std::begin(philosophy->second),
                              std::end(philosophy->second),
                              std::inserter(act_or_think,
                              std::begin (act_or_think)));
std::copy(std::begin(act_or_think), std::end(act_or_think),
                          std::ostream_iterator<Student>{std::cout, " "});
std::cout << std::endl;
```

结果集是用 Group 别名定义的，它对应于 set<Student>类型。集合运算的结果可以保存在任何地方，只要可以通过迭代器插入元素。这里使用 set_symmetric_difference()算法非常合适。因为结果会被存放在 set 容器中，所以不能使用 back_insert_iterator 或 front_insert_iterator。这里我们需要使用的是 insert_iterator，它是通过 inserter()创建的。inserter()的参数是一个容器对象和一个指向元素插入位置的迭代器。

在 Ex5_01 中添加的最后一段代码用来输出选择了戏剧或哲学课，或者两门课都选了的学生：

```
act_or_think.clear();                // Empty the container to reuse it
std::cout << "\nStudents studying drama and/or philosophy are:\n";
std::set_union(std::begin(drama->second), std::end(drama->second),
               std::begin(philosophy->second), std::end(philosophy->second),
               std::inserter(act_or_think, std::begin (act_or_think)));
std::copy(std::begin(act_or_think), std::end(act_or_think),
```

```
            std::ostream_iterator<Student>{std::cout, " "});
    std::cout << std::endl;
```

为了保存这次运算的结果，我们会重用 act_or_think 容器，因此调用了它的 clear()函数来移除之前的元素。set_union()创建了一个并集，然后用 copy()算法输出并集中的元素。

5.7 本章小结

set 容器和相应的 map 容器有相似的操作，但通常它们的使用完全不同。可以用 map 容器来保存与检索与键关联的元素。寻找指定名的地址或电话号码就是一个典型的示例。如果对象集合的成员关系比较重要，那么应该使用 set 容器。如果我们想知道某个人是否选择了一门给定的课程，或者某人是否参加篮球队的同时也参加了足球队，或者某人是否是圣昆廷监狱的犯人，就应该使用 set 容器。

本章的重点包括：

- set 容器保存的对象用它们自己作为键。
- set 容器存储的 T 类型对象是有序的，默认使用 less<T>对对象排序。
- multiset<T>容器保存对象的方式和 set 相同，但对象并不是唯一的。
- 如果两个对象是相等的，那么它们在 set 或 multiset 中会被看成同一个元素。如果 a<b 为 false，b<a 为 false，那么对象 a 和 b 是相等的。
- unordered_set<T>容器保存 T 类型的对象，并且它们是唯一的，它们的位置由对象的哈希值确定。
- unordered_multiset<T>容器中的对象位置也是由对象的哈希值确定的，但对象不需要是唯一的。
- 无序 set 容器用== operator 来决定两个对象是否相等，因而 T 类型必须支持这个运算符。
- 无序 set 容器中的对象通常保存在哈希表的格子中，用对象的哈希值可以确定对象在容器中的位置。
- 无序 set 容器的装载因子是每个格子平均的元素个数。
- 无序 set 容器在初始时会分配一些格子。当超过最大装载因子时，格子的个数会自动增加。
- STL 定义了一些集合运算的算法。二元集合运算是交、并、差、对称、包含。这些运算适用于一段元素。

练习

1. 定义一个 Card 类，用来表示标准扑克中的牌。创建一个 Card 对象的 vector，用来代表完整的 52 张牌。将这些牌随机分配到 4 个 set 容器中，表示游戏里每个人手中有 13 张牌。将这 4 个 set 容器中的牌放到东、南、西、北(这些通常是桥牌中的外延)。应该按花色

和值的大小输出—— 相同花色的比较大小。

2. 用 unordered_multiset 容器保存从键盘输入的任意段数的文本，然后以每行 6 个的方式输出单词及其出现次数。

3. 模拟掷一对骰子(每个骰子的值都是从 1 到 6)，然后将它们掷 1000 次，在 multiset 容器中记录每次两个骰子投掷的和。显然和的范围是 2 到 12。输出每个结果出现的次数。

4. 选择 10 本书的书名，用它们初始化一个 vector 容器。创建一个以姓名作为键，以书名为元素的 set 容器作为对象的 multimap 容器。为 multimap 中的每个人随机选择 4 到 6 本书。列出人名和他们所拥有的书。判断并记录两个人是否有两本以上的相同的书，最多可以拥有 6 本相同的书，这不太常见。然后输出它们，按照共有书的数目的升序输出。

第 6 章

排序、合并、搜索和分区

本章将介绍一些同排序和合并松散关联的算法。其中有两组算法专门提供排序和合并功能，一组为给定值范围内的元素提供分区机制，另一组提供了在范围内查找一个或多个元素的方法。本章将介绍以下内容：

- 如何将随机访问迭代器定义的元素段按升序或降序排列
- 如何在排序操作中阻止相等的元素重新排序
- 如何合并有序元素段
- 如何在无序元素段中查找一个或多个元素
- 如何使用二分查找算法

6.1 序列排序

在很多应用中，排序都是至关重要的，而且很多 STL 算法也只适用于有序对象序列。定义在 algorithm 头文件中的函数模板 sort<Iter>()默认会将元素段排成升序，这也就意味着排序的对象的类型需要支持<运算符。对象也必须是可交换的，这说明可以用定义在 utility 头文件中的函数模板 swap()来对两个对象进行交换。这进一步表明这种对象的类型需要实现移动构造函数和移动赋值运算符。函数模板 sort<Iter>()的类型参数 Iter 是元素段元素对应的迭代器类型，而且它们必须支持随机访问迭代器。这表明 sort()算法只能对提供随机访问迭代器的容器中的元素进行排序，也说明 sort()只能接受 array、vector、deque 或标准数组中的元素。可以回顾第 2 章，list 和 forward_list 容器都有成员函数 sort()；这些用来排序的特殊成员函数是必要的，因为 list 只提供双向迭代器，且 forward_list 只提供正向迭代器。

可以从函数调用的参数中推导出 sort()的模板类型参数，它们是定义被排序对象范围的迭代器。当然，迭代器类型隐式定义了这个元素段中对象的类型。下面是一个使用 sort()算法的示例：

```
std::vector<int> numbers {99, 77, 33, 66, 22, 11, 44, 88};
std::sort(std::begin(numbers), std::end(numbers));
std::copy(std::begin(numbers), std::end(numbers),
          std::ostream_iterator<int> {std::cout, " "}); // Output: 11 22
                                                        // 33 44 66 77 88 99
```

sort()调用将 number 容器的全部元素排成升序，然后用 copy()算法输出结果。可以不必对容器的全部内容进行排序。下面这条语句对 numbers 中除了第一个和最后一个元素之外的元素进行了排序：

```
std::sort(++std::begin(numbers), --std::end(numbers));
```

为了将元素排成降序，需要提供一个用来比较元素的函数对象，作为 sort()的第三个参数：

```
std::sort(std::begin(numbers), std::end(numbers), std::greater<>());
```

这个比较函数必须返回布尔值，而且有两个参数，它们要么是迭代器解引用后得到的类型，要么是迭代器解引用后可以隐式转换成的类型。参数可以是不同类型的。只要比较函数满足这些条件，它就可以是你喜欢的任何样子，也包括 lambda 表达式。例如：

```
std::deque<string> words {"one", "two", "nine", "nine", "one", "three",
  "four", "five", "six"};
std::sort(std::begin(words), std::end(words),
  [](const string& s1, const string& s2){ return s1.back()> s2.back(); });
std::copy(std::begin(words), std::end(words),
  std::ostream_iterator<string> {std::cout, " "}); // six four two one nine
                                                   // nine one three five
```

这段代码对 deque 容器 words 中的 string 元素进行了排序，并且输出了排序后的结果。这里的比较函数是一个 lambda 表达式，它们用每个单词的最后一个字母来比较排序的顺序。结果元素以它们最后一个字母的降序来排序。

下面在一个简单的示例中介绍 sort()的用法。这里会先从键盘读取 Name 对象，然后将它们按升序排列，再输出结果。Name 类定义在 Name.h 头文件中，它包含下面这些代码：

```
#ifndef NAME_H
#define NAME_H
#include <string>                              // For string class
class Name
{
private:
  std::string first{};
  std::string second{};
public:
  Name(const std::string& name1, const std::string& name2) : first(name1),
    second(name2){}
  Name()=default;
```

```
  std::string get_first() const {return first;}
  std::string get_second() const { return second; }

  friend std::istream& operator>>(std::istream& in, Name& name);
  friend std::ostream& operator<<(std::ostream& out, const Name& name);
};

// Stream input for Name objects
inline std::istream& operator>>(std::istream& in, Name& name)
{
  return in >> name.first >> name.second;
}

// Stream output for Name objects
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{
  return out << name.first << " " << name.second;
}
#endif
```

这个流插入和提取运算符被定义为 Name 对象的友元函数。可以将 operator<()定义为类的成员函数，但为了展示如何为 sort()算法指定比较函数参数，这里没有定义它。下面是程序代码：

```
// Ex6_01.cpp
// Sorting class objects
#include <iostream>                    // For standard streams
#include <string>                      // For string class
#include <vector>                      // For vector container
#include <iterator>                    // For stream and back insert iterators
#include <algorithm>                   // For sort() algorithm
#include "Name.h"
int main()
{
  std::vector<Name> names;
  std::cout << "Enter names as first name followed by second name. Enter Ctrl+Z
    to end:";
  std::copy(std::istream_iterator<Name>(std::cin), std::istream_iterator
    <Name>(),std::back_insert_iterator<std::vector<Name>>(names));

  std::cout << names.size() << " names read. Sorting in ascending sequence... \n";
  std::sort(std::begin(names), std::end(names), [](const Name& name1, const
    Name& name2){return name1.get_second() < name2.get_second(); });

  std::cout << "\nThe names in ascending sequence are:\n";
  std::copy(std::begin(names), std::end(names),
    std::ostream_iterator <Name> (std::cout, "\n"));
}
```

main()中的一切几乎都是使用 STL 模板完成的。names 容器用来保存从 cin 读入的姓名。

输入由 copy()算法执行，它使用一个 istream_iterator<Name>实例读入 Name 对象。istream_iterator<Name>默认的构造函数会创建流的结束迭代器。copy()函数用 back_insert_iterator<Name>()创建的 back_inserter<Name>迭代器将输入的每个对象复制到 names 中。为 Name 类重载的流运算符允许使用流迭代器来输入和输出 Name 对象。

Name 对象的比较函数是用 lambda 表达式定义的，它是 sort()算法的第三个参数。如果想将 operator<()定义为 Name 类的成员函数，可以省略这个参数。然后用 copy()算法将排序后的 names 写入标准输出流，它会将前两个参数指定范围内的元素复制到作为第三个参数的 ostream_iterator<Name>对象中。

下面是示例输出：

```
Enter names as first name followed by second name. Enter Ctrl+Z to end:
Jim Jones
Bill Jones
Jane Smith
John Doe
Janet Jones
Willy Schaferknaker
^Z
6 names read. Sorting in ascending sequence...

The names in ascending sequence are:
John Doe
Jim Jones
Bill Jones
Janet Jones
Willy Schaferknaker
Jane Smith
```

对 names 的排序只考虑了姓。当姓相同时，可以将 lambda 表达式扩展为可以比较名。

你可能会好奇在这个示例中为什么不用 pair<string,string>来表示姓名，这比定义一个新的类要简单多了。显然我们可以这么做，但却没有定义类这么清楚明了。

6.1.1 排序以及相等元素的顺序

sort()算法可能会改变相等元素的顺序，有时候这不是我们想要的。假设有一个保存某种事务的容器，或许是银行账户。进一步假设，在处理它们之前，为了能够有序更新这些账户，需要按照账号对这些事务排序。如果出现相等事务的顺序反映的是它们添加到容器的时间顺序，就需要维持这个顺序不变。如果允许对给定账户的事务进行重新排列，可能会出现透支的情况。

这种情况下，stable_sort()算法可以满足我们的要求，它会对一段元素进行排序并保证维持相等元素的原始顺序。这里有两个版本：其中一个接受两个用来指定排序元素范围的迭代器，另一个接受用于比较的额外参数。为了演示 stable_sort()的使用，可以对这个用来对 Ex6_01.cpp 中的 names 容器进行排序的语句进行修改：

```
std::stable_sort(std::begin(names), std::end(names),
  [](const Name& name1, const Name& name2) { return name1.get_second() <
    name2.get_second(); });
```

当然，这里所展示的 Ex6_01 使用 sort()的输出并没有打乱相等元素的顺序，所以使用 stable_sort()并不会改变相同输入的输出。它们的不同之处在于，stable_sort()保证不会改变相等元素的顺序，这是 sort()算法所不能保证的。当需要使相等元素的顺序保持不变时，应该使用 stable_sort()。

6.1.2　部分排序

通过示例很容易理解什么是部分排序。假设有一个容器，它保存了 100 万个数值，但我们只对其中最小的 100 个感兴趣。可以对容器的全部内容排序，然后选择前 100 个元素，但这可能有点消耗时间。这时候需要使用部分排序，只需要这些数中的前 100 个是有序放置的。对于部分排序，有一个特殊的算法 partial_sort()，它需要 3 个随机访问迭代器作为参数。如果这个函数的参数是 first、second 和 last，那么这个算法会被应用到[first,last)这个范围内的元素上。执行这个算法后，[first,second)会包含降序序列[first,last)中最小的 second-first 个元素。

■ **注意**：在这个示例中，有一种之前没遇到过的表示方式[first,last)，用它来表示一个元素段，这是一个来自于数学领域的用来定义数字范围的概念——区间。这两个值叫作结束点，在这种表示法中，方括号表示包含相邻的结束点，圆括号表示相邻的结束点不包括在内。例如，如果(2,5)是一个整数区间，2 和 5 都被排除在外，所以它只表示整数 3 和 4；这也被叫作开区间，因为两个结束点都不包含。区间[2,5)包含 2 但不包含 5，所以它表示 2、3 和 4。(2,5]表示 3、4 和 5。[2,5]表示 2、3、4、5，并且它被叫作闭区间，因为包含了两个结束点。当然，first 和 last 都是迭代器，并且[first,last)表示包含 first 指向的元素而不包含 last 指向的元素——所以可以用它准确表示 C++中的范围。

下面这段代码演示了 partial_sort()算法的工作方式：

```
size_t count {5}; // Number of elements to be sorted
std::vector<int> numbers {22, 7, 93, 45, 19, 56, 88, 12, 8, 7, 15, 10};
std::partial_sort(std::begin(numbers), std::begin(numbers) + count,
  std::end(numbers));
```

执行 partial_sort()后的效果如图 6-1 所示。

最小的 count 个元素是有序的。在[first,second)范围内，second 指向的元素没有被包含在内，因为 second 是一个开结束点。图 6-1 展示的是在笔者系统上这段代码执行后的结果。在你的系统上结果可能会不同。需要注意的是，不能保持未排序元素的原始顺序。在执行 partial_sort()后这些元素的顺序是不确定的，这取决于我们的实现。

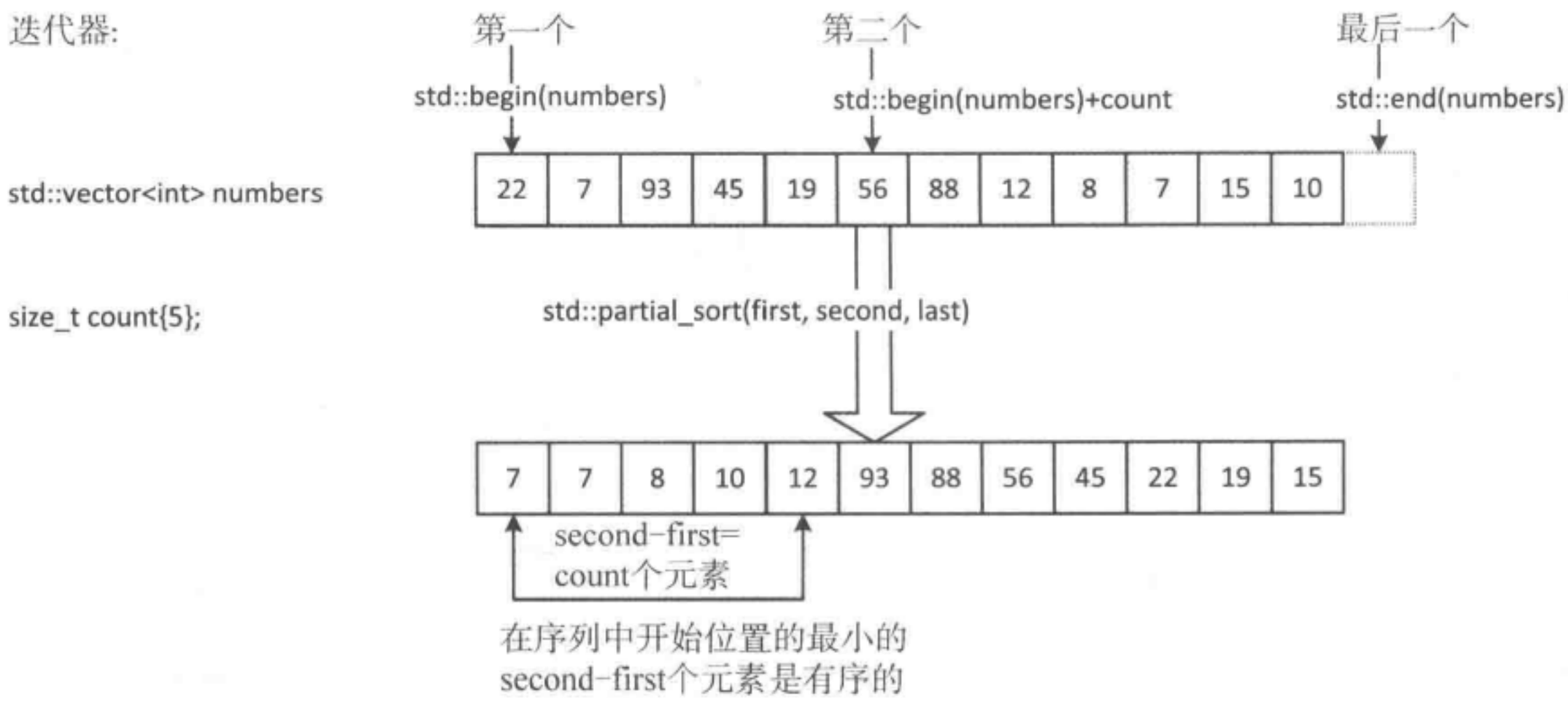


图 6-1　partial_sort()算法的操作

如果想让 partial_sort()算法使用和<运算符不同的比较函数，可以提供一个函数对象作为它的额外参数。例如：

```
std::partial_sort(std::begin(numbers), std::begin(numbers) + count,
  std::end(numbers),std::greater<>());
```

现在，number 中最大的 count 个元素会是容器开始处的一个降序序列。在笔者系统上这条语句的输出结果为：

```
93 88 56 45 22 7 19 12 8 7 15 10
```

同样，没有保持 numbers 中未排序元素的原始顺序。

partial_sort_copy()在本质上和 partial_sort()是相同的，除了前者会将排序元素复制到一个不同的元素段——另一个容器中。它的前两个参数是指定部分排序应用范围的迭代器；第 3 个和第 4 个参数是标识结果存放位置的迭代器。目的位置的元素个数决定了输入元素段中被排序元素的个数。例如：

```
std::vector<int> numbers {22, 7, 93, 45, 19, 56, 88, 12, 8, 7, 15, 10};
size_t count {5};                    // Number of elements to be sorted
std::vector<int> result(count); // Destination for the results - count elements
std::partial_sort_copy(std::begin(numbers), std::end(numbers),
  std::begin (result),std::end(result));
std::copy(std::begin(numbers), std::end(numbers), std::ostream_iterator<int>
  {std::cout, " "});
std::cout << std::endl;
std::copy(std::begin(result), std::end(result), std::ostream_iterator
  <int> {std::cout, " "});
std::cout << std::endl;
```

这些语句实现了对 numbers 容器的部分排序。这个想法是先对 numbers 中最小的 count 个元素排序，然后把它们保存在结果容器中。我们指定的用于存放元素的目的位置必须存在，也就是说，目的容器 result 必须至少有 count 个元素，在这个示例中我们需要分配准确数目的内存。执行这段代码后的输出如下：

```
22 7 93 45 19 56 88 12 8 7 15 10
7 7 8 10 12
```

可以看到，numbers 中元素的顺序并没有被打乱，result 中包含了 number 中按升序排列的最小的 count 个元素。

当然，也可以用额外的参数来指定不同的比较函数：

```
std::partial_sort_copy(std::begin(numbers), std::end(numbers),
std::begin
  (result), std::end(result),std::greater<>());
```

指定一个 greater<>的实例作为函数对象，将最大的 count 个元素以降序的形式复制到 result 中。如果这条语句后跟的是前一个代码段中的输出语句，会输出如下内容：

```
22 7 93 45 19 56 88 12 8 7 15 10
93 88 56 45 22
```

同前面一样，原始容器中元素的顺序没有被打乱。

nth_element()算法和 partial_sort()不同。应用的范围由它的第一个和第三个参数指定。第二个参数是一个指向第 n 个元素的迭代器。如果这个范围内的元素是完全有序的，nth_element()的执行会导致第 n 个元素被放置在适当的位置。这个范围内，在第 n 个元素之前的元素都小于第 n 个元素，而且它后面的每个元素都会比它大。算法默认用<运算符来生成这个结果。下面是一个使用 nth_elemen()的示例：

```
std::vector<int> numbers {22, 7, 93, 45, 19, 56, 88, 12, 8, 7, 15, 10};
size_t count {5};                                    // Index of nth element
std::nth_element(std::begin(numbers), std::begin(numbers) + count,
std::end(numbers));
```

这里的第 n 个元素是 numbers 容器的第 16 个元素，对应于 numbers[5]，图 6-2 展示了它的工作方式。

第 n 个元素之前的元素都小于它，但不必是有序的。同样，第 n 个元素后的元素都大于它，但也不必是有序的。如果第二个参数和第三个参数相同——元素段的末尾，这时这个算法是无效的。

正如本章前面的算法一样，可以自己定义比较函数作为函数的第 4 个参数：

```
std::nth_element(std::begin(numbers), std::begin(numbers) + count,
  std::end(numbers),std::greater<>());
```

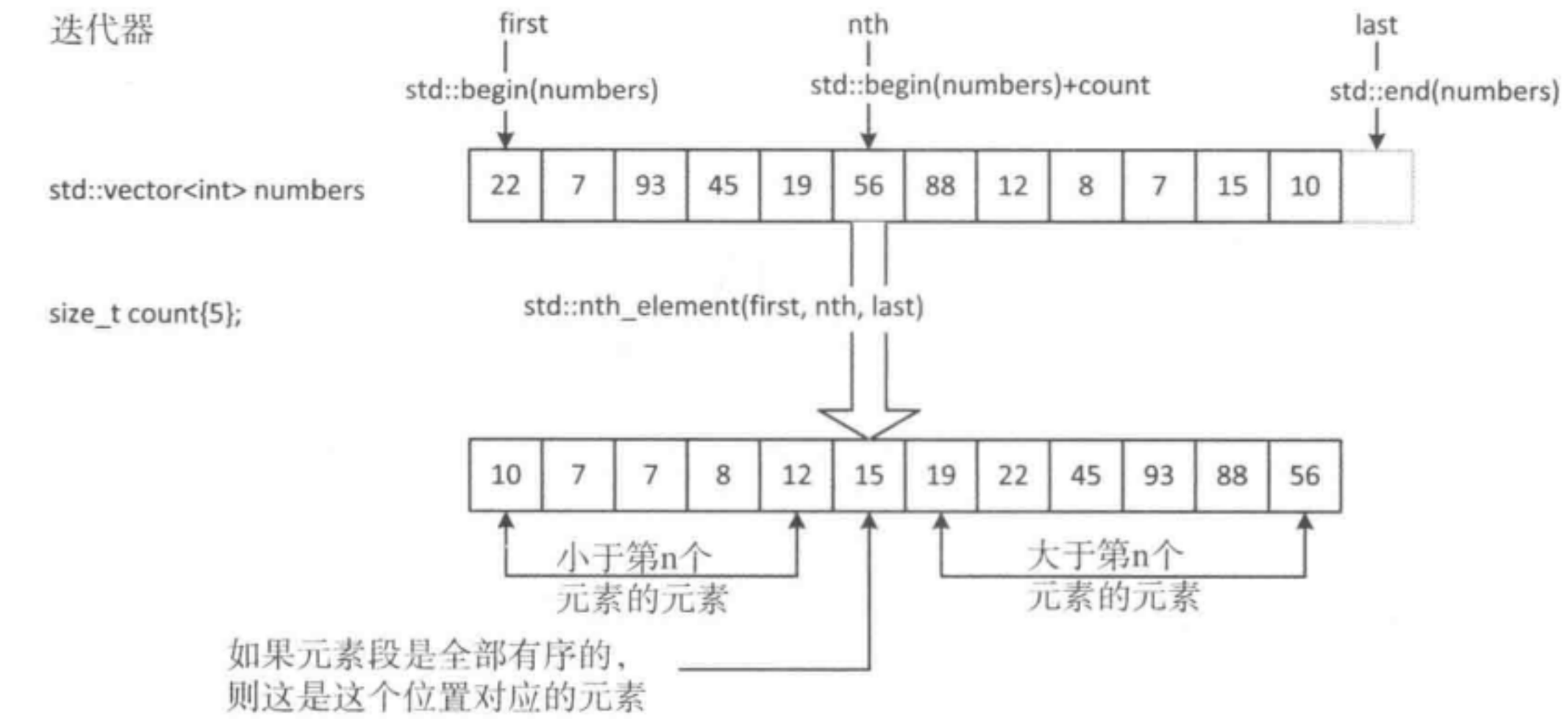


图 6-2　nth_element()算法的操作

这里使用>运算符来比较函数，所以第 n 个元素将是元素按降序排列后的第 n 个元素。第 n 个元素之前的元素都大于它，之后的元素都小于它。如果 number 容器中的初始值和之前的一样，那么结果为：

```
45 56 93 88 22 19 10 12 15 7 8 7
```

在你的系统上，第 n 个元素两边元素的顺序可能会不同，但它左边的元素都应该比它大，而右边的元素都应该比它小。

6.1.3　测试排序序列

排序是要耗费时间的，尤其是在有大量元素时。测试元素段是否已经排好序可以避免不必要的排序操作。如果两个迭代器参数所指定范围内的元素是升序排列的，函数模板 is_sorted()就会返回 true。为了能够顺序处理元素，迭代器至少需要是正向迭代器。提醒一下——正向迭代器支持前缀和后缀形式的自增运算。下面是一个使用 is_sorted()的示例：

```
std::vector<int> numbers {22, 7, 93, 45, 19};
std::vector<double> data {1.5, 2.5, 3.5, 4.5};
std::cout << "numbers is "
          << (std::is_sorted(std::begin(numbers), std::end(numbers)) ? "": "not ")
          << "in ascending sequence.\n";
std::cout << "data is "
          << (std::is_sorted(std::begin(data), std::end(data)) ? "": "not ")
          << "in ascending sequence." << std::endl;
```

默认使用的比较函数是<运算符。鉴于“data is”的输出，表明 numbers 不是一个升序序列。还有一个版本也可以指定用于比较元素的函数对象：

```
std::cout << "data reversed is "
```

```
          << (std::is_sorted(std::rbegin(data), std::rend(data),
              std::greater<>()) ? "": "not ")
          << "in descending sequence." << std::endl;
```

这条语句的输出说明 data 中元素的逆序是降序。

也可用函数模板 is_sorted_until()来判断一个元素段是否有序。它的参数用来定义要测试的迭代器。这个函数会返回一个指向这段元素中升序序列上边界元素的迭代器。例如：

```
std::vector<string> pets {"cat", "chicken", "dog", "pig", "llama", "coati",
  "goat"};
std::cout << "The pets in ascending sequence are:\n";
std::copy(std::begin(pets), std::is_sorted_until(std::begin(pets),
  std::end(pets)),std::ostream_iterator<string>{std::cout, " "});
```

copy()算法的前两个参数分别是 pets 容器的开始迭代器以及当 is_sorted_until()应用到 pets 中全部元素上时返回的迭代器。is_sorted_until()算法会返回指向 pets 中升序序列元素的上边界，它是指向小于其前面元素的第一个元素的迭代器。如果序列是有序的，则返回一个结束迭代器。这段代码的输出如下：

```
The pets in ascending sequence are:
cat chicken dog pig
```

"llama"是小于其前者的第一个元素，所以"pig"就是升序序列的第一个元素。

可以选择提供一个用于比较元素的函数对象：

```
std::vector<string> pets {"dog", "coati", "cat", "chicken", "pig", "llama",
  "goat"};
std::cout << "The pets in descending sequence are:\n";
std::copy(std::begin(pets),std::is_sorted_until(std::begin(pets), std::
end(pets), std:: greater<>()),
          std::ostream_iterator<string>{std::cout, " "});
```

这一次我们会查找降序元素，因为使用的是 string 类的成员函数 operator>()来比较元素。输出为：

```
The pets in descending sequence are:
dog coati cat
```

"chicken"是大于其前者的第一个元素，所以 is_sorted_until()返回的迭代器就指向这个元素，因而"cat"是降序序列的最后一个元素。

6.2　合并序列

合并操作会合并两个有相同顺序的序列中的元素，可以是两个升序序列，也可以是两个降序序列。结果会产生一个包含来自这两个输入序列的元素副本的序列，并且排序方式

和原始序列相同。图 6-3 说明了其工作方式。

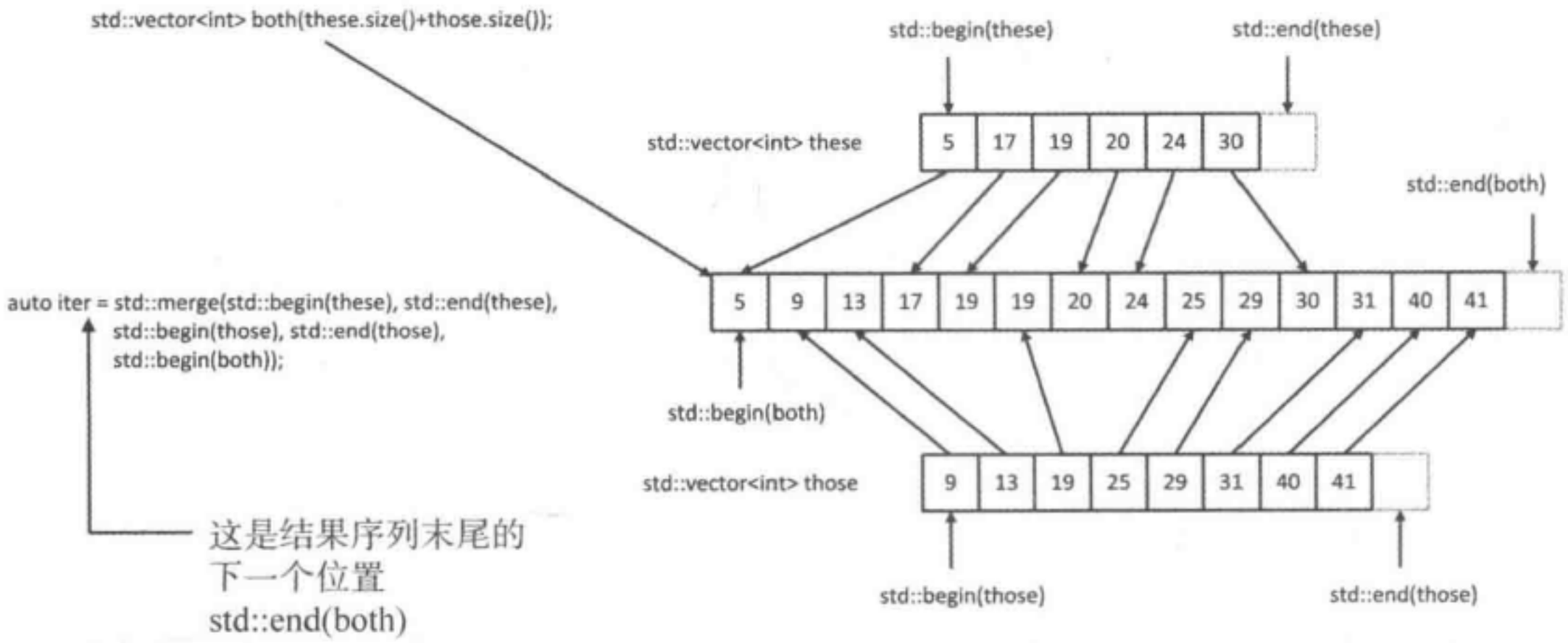


图 6-3　合并两个 vector 容器中的元素

merge()算法会合并两个序列并将结果保存到第三个序列中，它使用<运算符来比较元素。图 6-3 表明合并操作被运用到 these 和 those 容器的内容上，结果序列保存在 both 容器中。merge()算法需要 5 个参数。其中前两个指定第一个输入序列的迭代器——在这个示例中是 these，后面两个迭代器指定第二个输入序列——在这个示例中是 those，最后一个参数是一个指定合并元素存放位置的迭代器——both 容器。用来指定输入序列的迭代器只需要是最低层次的迭代器，用来保存合并结果的迭代器需要是一个输出迭代器。

merge()算法并没有关于被合并序列容器的信息，所以它们不能创建元素，只能用提供的作为第 5 个参数的迭代器来保存元素。因而在这个示例中，目的序列中的元素必须是已经存在的。在图 6-3 中，通过以两个输入容器元素个数之和为指定的元素个数创建一个 both 容器来保证此要求。创建的结果序列可以放在任何位置，甚至可以放在一个源序列容器中，但源序列和目的序列不能重叠；如果它们重叠了，结果是未定义的，但可以肯定的是效果肯定不好。当然，可以用一个插入迭代器来指定目的位置，元素会被自动创建。

merge()算法返回的迭代器指向合并序列末尾的后一个位置，所以可以通过这个函数调用使用的第 5 个参数加上这个函数返回的迭代器来确定合并序列的范围。

当需要使用不同于<运算符的其他比较运算时，可以提供一个函数对象用来作为第 6 个参数。例如：

```
std::vector<int> these {2, 15, 4, 11, 6, 7};
                                                    // 1st input to merge
std::vector<int> those {5, 2, 3, 2, 14, 11, 6}; // 2nd input to merge
std::stable_sort(std::begin(these), std::end(these),
                                                    // Sort 1st range in...
std::greater<>());
                                                    // ...descending sequence
std::stable_sort(std::begin(those), std::end(those),   // Sort 2nd range
std::greater<>());
std::vector<int> result(these.size() + those.size() + 10);
```

```
                                                    // Plenty of room for results
auto end_iter = std::merge(std::begin(these), std::end(these),
                                                     // Merge 1st range...
                           std::begin(those), std::end(those),
                                                    // ...and 2nd range...
                           std::begin(result), std::greater<>());
                                                    // ...into result
std::copy(std::begin(result), end_iter, std::ostream_iterator<int>{std::cout, " "});
```

这段代码首先用 stable_sort()将两个 vector 容器的内容排成降序，stable_sort()可以保证维持相等元素的原始顺序。合并操作会将两个容器的内容合并到第三个容器 result 中，创建的元素比需要的还多 10 个——仅仅是为了说明 merge()返回迭代器的用法。copy()算法会将由 result 的开始迭代器和 merge()返回的 end_iter 迭代器指定范围内的元素复制到输出迭代器。输出如下：

```
    15 14 11 11 7 6 6 5 4 3 2 2 2
```

inplace_merge()算法可以合并同一个序列中两个连续有序的元素序列。它有三个参数：first、second、last 和 last 是一个双向迭代器。这个序列中的第一个输入序列是[first,second)，第二个输入序列是[second,last)，因而 second 指向的元素在第二个输入序列中。结果为[first, last)。图 6-4 展示了这个操作。

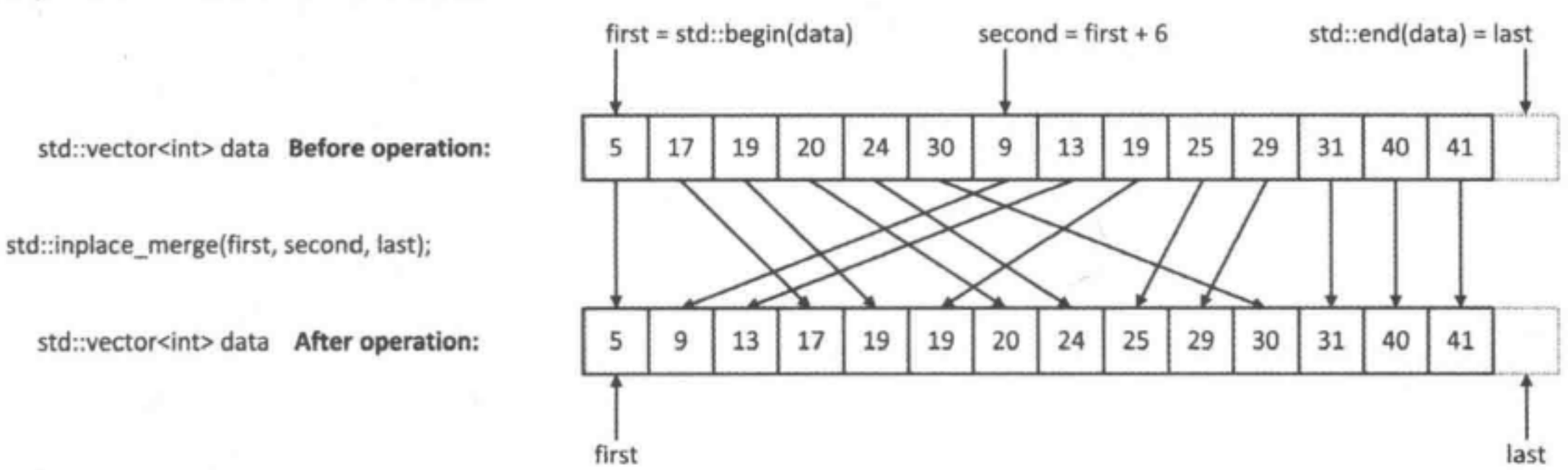


图 6-4　inplace_merge()操作

图 6-4 中的 data 容器有两个序列，并且都是升序序列。inplace_merge()操作将它们合并为同一容器中的升序序列。

我们将在一个示例中介绍本章已讲解的几种合并算法。这个示例用来处理从键盘输入的借记和贷记事务，并将它们应用到一组按需创建的账户上。我们创建的账户的余额总是为 0。一笔事务是一个包含账号、数额、数额是借记还是信用的标识的对象。处理不存在的账号的事务会导致这个账号被创建。一个账号对象会包含一些标识唯一账号、拥有者姓名以及当前余额的成员变量。账号拥有者的名字是一个包含名和姓的 pair 对象。账号是一个无符号整数。信用是一个布尔值，余额和借记或贷记的数额是一个 double 类型的数。

Transaction 类型在如下所示的 Transaction.h 头文件中定义：

```
#ifndef TRANSACTION_H
#define TRANSACTION_H
```

```
#include <iostream>                          // For stream class
#include <iomanip>                           // For stream manipulators
#include "Account.h"

class Transaction
{
private:
  size_t account_number {};                  // The account number
  double amount {};                          // The amount
  bool credit {true};                        // credit = true debit=false
public:
  Transaction()=default;
  Transaction(size_t number, double amnt, bool cr) : account_number {number},
    amount {amnt},credit {cr}{}
  size_t get_acc_number() const { return account_number; }

  // Less-than operator - compares account numbers
  bool operator<(const Transaction& transaction) const { return account_number <
                                                        transaction.account_number; }

  // Greater-than operator - compares account numbers
  bool operator>(const Transaction& transaction) const { return account_number >
                                                        transaction.account_number; }

  friend std::ostream& operator<<(std::ostream& out, const Transaction& transaction);
  friend std::istream& operator>>(std::istream& in, Transaction& transaction);

  // Making the Account class a friend allows Account objects
  // to access private members of Transaction objects
  friend class Account;

};
// Stream insertion operator for Transaction objects
std::ostream& operator<<(std::ostream& out, const Transaction& transaction)
{
  return out << std::right << std::setfill('0') << std::setw(5)
             << transaction.account_number
             << std::setfill(' ') << std::setw(8) << std::fixed << std::setprecision(2)
             << transaction.amount
             << (transaction.credit ? " CR" : " DR");
}
// Stream extraction operator for Transaction objects
std::istream& operator>>(std::istream& in, Transaction& tr)
{
  if((in >> std::skipws >> tr.account_number).eof())
    return in;
  return in >> tr.amount >> std::bcolalpha >> tr.credit;
}
#endif
```

默认的构造函数通常可以在容器中创建默认元素。这个类包含的<和>运算符允许以升

序或降序的方式对 Transaction 进行排列，尽管在这个示例中并没有充分利用这两个选择。下面将定义的 Account 类是 Transaction 类的友元类，所以 Account 的成员函数可以访问传入的 Transaction 对象的私有成员变量。因为重载了流的输入输出运算符，所以可以将 copy() 算法和 STL 提供的流迭代器结合起来，从而读或写 Transaction 对象。

Account 类定义在 Account.h 头文件中：

```
#ifndef ACCCOUNT_H
#define ACCCOUNT_H
#include <iostream>                          // For stream class
#include <iomanip>                           // For stream manipulators
#include <string>                            // For string class
#include <utility>                           // For pair template type
#include "Transaction.h"

using first_name = std::string;
using second_name = std::string;
using Name = std::pair<first_name, second_name>;

class Account
{
private:
  size_t account_number {};                  // 5-digit account number
  Name name {"", ""};                        // A pair containing 1st & 2nd names
  double balance {};                         // The account balance - negative when
overdrawn

public:
  Account()=default;
  Account(size_t number, const Name& nm) : account_number {number}, name {nm}{}

  double get_balance() const { return balance; }
  void set_balance(double bal) { balance = bal; }

  size_t get_acc_number() const {return account_number;}
  const Name& get_name() const { return name; }

  // Apply a transaction to the account
  bool apply_transaction(const Transaction& transaction)
  {
    if(transaction.credit)                   // For a credit...
      balance += transaction.amount;         // ...add the mount
    else                                     // For a debit...
      balance -= transaction.amount;         // ...subtract the amount
    return balance < 0.0;                    // Return true when overdrawn
  }

// Less-than operator - compares by account number
bool operator<(const Account& acc) const { return account_number < acc.account_
  number; }
friend std::ostream& operator<<(std::ostream& out, const Account& account);
```

```
};

// Stream insertion operator for Account objects
std::ostream& operator<<(std::ostream& out, const Account& acc)
{
  return out << std::left << std::setw(20) << acc.name.first + " " + acc.name.second
    << std::right << std::setfill('0') << std::setw(5) << acc.account_number
    << std::setfill(' ') << std::setw(8) << std::fixed << std::setprecision(2)
    << acc.balance;
}
#endif
```

除了账号，这个类还有一个 Name 成员，可以用来修改账号的拥有者。Name 只是pair<string,string>类型的别名，别名 first_name 和 second_name 只用来标识每个 pair 成员的意义。类型别名通常可以用来说明应用中一般类型的含义。

Account 对象重载了流插入运算符，这允许我们用<<将对象写到输出流。成员函数operator<()的定义允许 Account 对象按账号排序，仅当它们保存在有序容器或在有序容器中排序时。如果想用不同的方式来对 Account 排序——例如按照姓名，就需要定义一个函数对象来提供比较函数。在这个示例中会按姓名来对 Account 排序，相应的函数对象定义在 Compare_Names.h 文件中，内容如下：

```
#ifndef COMPARE_NAMES_H
#define COMPARE_NAMES_H
#include "Account.h"

// Order Account objects in ascending sequence by Name
class Compare_Names
{
public:
  bool operator()(const Account& acc1, const Account& acc2)
  {
    const auto& name1 = acc1.get_name();
    const auto& name2 = acc2.get_name();
    return (name1.second < name2.second) ||
              ((name1.second == name2.second) && (name1.first < name2.first));
  }
};
#endif
```

上面的代码理解起来并不难，所定义的函数调用操作符对两个 Account 对象的 Name 个数进行比较，比较顺序是先按姓氏，再按名字。

main()程序使用了 Ex6_02.cpp 中定义的类：

```
// Ex6_02.cpp
// Sorting and inplace merging
#include <iostream>                    // For standard streams
#include <string>                      // For string class
#include <algorithm>                   // For sort(), inplace_merge()
```

```
#include <functional>                    // For greater<T>
#include <vector>                        // For vector container
#include <utility>                       // For pair template type
#include <map>                           // For map container
#include <iterator>                      // For stream and back insert iterators
#include "Account.h"
#include "Transaction.h"
#include "Compare_Names.h"

using std::string;
using first_name = string;
using second_name = string;
using Name = std::pair<first_name, second_name>;
using Account_Number = size_t;

// Read the name of an account holder
Name get_holder_name(Account_Number number)
{
  std::cout << "Enter the holder's first and second names for account number " <<
    number << ": ";
  string first {};
  string second {};
  std::cin >> first >> second;
  return std::make_pair(first, second);
}

int main()
{
  std::vector<Transaction> transactions;
  std::cout << "Enter each transaction as:\n"
            << " 5 digit account number amount credit(true or false).\n"
            << "Enter Ctrl+Z to end.\n";

  // Read 1st set of transactions
  std::copy(std::istream_iterator<Transaction> {std::cin}, std::istream_iterator
   <Transaction> {},std::back_inserter(transactions));
  std::cin.clear();                      // Clear the EOF flag for the stream

  // Sort 1st set in descending account sequence
  std::stable_sort(std::begin(transactions), std::end(transactions), std::
    greater<>());

  // List the transactions
  std::cout << "First set of transactions after sorting...\n";
  std::copy(std::begin(transactions), std::end(transactions),
                        std::ostream_iterator<Transaction>{std::cout, "\n"});

  // Read 2nd set of transactions
  std::cout << "\nEnter more transactions:\n";
  std::copy(std::istream_iterator<Transaction> {std::cin}, std::istream_iterator
  <Transaction> {},std::back_inserter(transactions));
```

```
std::cin.clear();                          // Clear the EOF flag for the stream

// List the transactions
std::cout << "\nSorted first set of transactions with second set appended...\n";
std::copy(std::begin(transactions), std::end(transactions),
                  std::ostream_iterator<Transaction>{std::cout, "\n"});

// Sort second set into descending account sequence
auto iter = std::is_sorted_until(std::begin(transactions), std::end(transactions),
                                                          std::greater<>());
std::stable_sort(iter, std::end(transactions), std::greater<>());

// List the transactions
std::cout << "\nSorted first set of transactions with sorted second set
 appended...\n";
std::copy(std::begin(transactions), std::end(transactions),
                        std::ostream_iterator<Transaction>{std::cout, "\n"});

// Merge transactions in place
std::inplace_merge(std::begin(transactions), iter, std::end(transactions),
 std::greater<>());

// List the transactions
std::cout << "\nMerged sets of transactions...\n";
std::copy(std::begin(transactions), std::end(transactions),
                  std::ostream_iterator<Transaction>{std::cout, "\n"});

// Process transactions creating Account objects when necessary
std::map<Account_Number, Account> accounts;
for(const auto& tr : transactions)
{
 Account_Number number = tr.get_acc_number();
 auto iter = accounts.find(number);
 if(iter == std::end(accounts))
  iter = accounts.emplace(number, Account {number, get_holder_name(number)}).
     first;

 if(iter->second.apply_transaction(tr))
 {
  auto name = iter->second.get_name();
  std::cout << "\nAccount number " << number
         << " for " << name.first << " " <<name.second << " is overdrawn!\n"
         << "The concept is that you bank with us - not the other way round,
           so fix it!\n"
         << std::endl;
 }
}

// Copy accounts to a vector container
std::vector<Account> accs;
for(const auto& pr :accounts)
 accs.push_back(pr.second);
```

```
    // List accounts after sorting in name sequence
    std::stable_sort(std::begin(accs), std::end(accs), Compare_Names());
    std::copy(std::begin(accs), std::end(accs),
      std::ostream_iterator < Account > {std::cout, "\n"});
}
```

get_holder_name()是一个用来从 cin 读取给定账号的辅助函数。当处理给定账号的事务，但这个 Account 对象并不存在时，需要使用这个函数。返回的 Name 对象会用在这个 Account 对象的创建中。

事务以Transaction对象的形式被读入并保存在vector<Transaction>容器transactions中，这段代码读入一个 Transaction 序列，然后用 stable_sort()将它降序排列。然后第二个序列被读入相同的容器，而且以相同的方式对它们排序。

先创建一个包含两个有序事务序列的 vector，然后就可以利用 inplace_merge()来创建这两个序列的有序组合。

下面是由 5 个账户对应的 7 笔事务所产生的一个输出示例。我们选择这个事务数来说明排序和合并操作。

```
Enter each transaction as:
  5 digit account number amount credit(true or false).
Enter Ctrl+Z to end.
12345 40 true
12344 50 true
12346 75.5 true
^Z
First set of transactions after sorting...
12346 75.50 CR
12345 40.00 CR
12344 50.00 CR

Enter more transactions:
12344 25.25 true
12345 75 false
12345 100 true
12346 100 true
^Z

Sorted first set of transactions with second set appended...
12346 75.50 CR
12345 40.00 CR
12344 50.00 CR
12344 25.25 CR
12345 75.00 DR
12345 100.00 CR
12346 100.00 CR

Sorted first set of transactions with sorted second set appended...
12346 75.50 CR
12345 40.00 CR
```

```
12344 50.00 CR
12344 25.25 CR
12346 100.00 CR
12345 75.00 DR
12345 100.00 CR

Merged sets of transactions...
12346 75.50 CR
12346 100.00 CR
12345 40.00 CR
12345 75.00 DR
12345 100.00 CR
12344 50.00 CR
12344 25.25 CR
Enter the holder's first and second names for account number 12346: Stan Dupp
Enter the holder's first and second names for account number 12345: Ann Ounce

Account number 12345 for Ann Ounce is overdrawn!
The concept is that you bank with us - not the other way round, so fix it!

Enter the holder's first and second names for account number 12344: Dan Druff
Dan Druff       12344    75.25
Stan Dupp       12346   175.50
Ann Ounce       12345    65.00
```

在每个阶段都会列出 Transaction 对象的序列，所以可以看到，stable_sort()和inplace_merge()算法如预期那样工作。特别是，它们会维持等值事务的顺序，所以借方和贷方都在它们的原始位置。最后，为了显示被正确处理的事务，会以姓名的顺序列出账户。这是通过将 map 容器中的 Account 对象复制到 vector<Account>中来完成的，然后将stable_sort()算法应用到这个 vector 中的元素上，这个 vector 由 Compare_Names 函数对象提供比较。可以将 Account 对象复制到 set<Account, Compare_Names>中来代替 vector<Account>，set<Account, Compare_Names>可以自动对对象排序，但这时就没有使用 stable_sort()的机会了。

6.3 搜索序列

STL 提供了多种用来搜索对象序列的方法。它们大多用于无序序列，但有一些也要求序列是有序的，稍后会进行讲解。

6.3.1 在序列中查找元素

这里有三种可以在输入迭代器所定义的范围内查找单个对象的算法：

- find()算法会在前两个参数指定的范围内查找和第三个参数相等的第一个对象。
- find_if()算法会在前两个参数指定的范围内查找可以使第三个参数指定的谓词返回 true 的第一个对象。谓词不能修改传给它的对象。

- find_if_not()算法会在前两个参数指定的范围内查找可以使第三个参数指定的谓词返回 false 的第一个对象。谓词不能修改传给它的对象。

每一种算法都会返回一个指向被找到对象的迭代器，如果没有找到对象，会返回这个序列的结束迭代器。下面展示了如何使用 find()：

```
std::vector<int> numbers {5, 46, -5, -6, 23, 17, 5, 9, 6, 5};
int value {23};
auto iter = std::find(std::begin(numbers), std::end(numbers), value);
if(iter != std::end(numbers)) std::cout << value << " was found.\n";
```

这段代码会输出一条在 numbers 中找到 23 的消息，当然，可以反复调用 find()来找出这个序列中所有给定元素的匹配项：

```
size_t count {};
int five {5};
auto start_iter = std::begin(numbers);
auto end_iter = std::end(numbers);
while((start_iter = std::find(start_iter, end_iter, five)) != end_iter)
{
  ++count;
  ++start_iter;
}
std::cout << five << " was found " << count << " times." << std::endl; // 3 times
```

在 while 循环中，count 变量会通过自增来记录 five 在 vector 容器 numbers 中的发现次数。循环表达式调用 find()，在 start_iter 和 end_iter 定义的范围内查找 five。find()返回的迭代器被保存在 start_iter 中，它会覆盖这个变量先前的值。最初，find()会搜索 numbers 中的所有元素，因此 find()会返回一个指向 five 的第一个匹配项的迭代器。每次找到 five，循环体中的 start_iter 都会自增，因此它会指向被找到元素的后一个元素。所以，下一次遍历搜索的范围是从这个位置到序列末尾。当不再能够找到 five 时，find()会返回 end_iter，循环结束。

可以按如下方式使用 find_if()来查找 numbers 中第一个大于 value 的元素：

```
int value {5};
auto iter1 = std::find_if(std::begin(numbers), std::end(numbers),
                                        [value](int n) { return n > value; });
if(iter1 != std::end(numbers))
  std::cout << *iter1 << " was found greater than " << value << ".\n";
```

find_if()的第三个参数是一个 lambda 表达式的谓词。这个 lambda 表达式以值的方式捕获 value，并在 lambda 参数大于 value 时返回 true。这段代码会找到一个值为 46 的元素。可以用和前一个代码段相同的方式，在循环中使用 find_if()来查找 numbers 中所有大于 value 的元素。

可以按如下方式用 find_if_not()算法来查找使谓词为 false 的元素：

```
size_t count {};
int five {5};
```

```
auto start_iter = std::begin(numbers);
auto end_iter = std::end(numbers);
while((start_iter = std::find_if_not(start_iter, end_iter,
                          [five](int n) {return n > five; })) != end_iter)
{
  ++count;
  ++start_iter;
}
std::cout << count << " elements were found that are not greater than "
  << five << std::endl;
```

find_if_not()的第三个参数是一个谓词，它和先前在 find_if()算法中使用的 lambda 表达式相似。当元素大于 five 时，这个函数才会返回 true。当谓词返回 false 时，就找到了一个元素，所以这个操作可以用来查找小于或等于 five 的元素。这段代码会找到 5 个元素，它们分别是 5、-5、-6、5、5。

6.3.2 在序列中查找任意范围的元素

find_first_of()算法可以在第一个序列中搜索第二个序列在第一个序列中第一次出现的任何元素。序列被搜索的范围由输入迭代器指定，但用于确定搜索序列的迭代器至少是正向迭代器。用==运算符来比较这两个序列中的元素，所以如果序列中是类类型的对象，这个类必须实现 operator==()。下面是一个使用 find_first_of()的示例：

```
string text {"The world of searching"};
string vowels {"aeiou"};
auto iter = std::find_first_of(std::begin(text), std::end(text), std::begin(vowels),
                                                                 std::end(vowels));
if(iter != std::end(text))
std::cout << "We found '" << *iter << "'." << std::endl; // We found 'e'.
```

这段代码会在 text 中查找第一次出现的 vowels 中的任意字符。在这个示例中，返回的迭代器指向“The”的第三个字母。可以用循环来查找 text 中所有匹配 vowels 中字符的字符：

```
string found {};                        // Records characters that are found
for(auto iter = std::begin(text);
    (iter = std::find_first_of(iter, std::end(text),
          std::begin(vowels), std::end(vowels))) != std::end(text); )
  found += *(iter++);
std::cout << "The characters \"" << found << "\" were found in text."
  << std::endl;
```

为了展示我们可以这样做，这里使用了 for 循环。第一个循环控制表达式以 text 的开始迭代器为初值定义了 iter。第二个循环控制表达式调用 find_first_of()，在[iter, std::end(text))这段范围内搜索第一次出现的 vowel 中的字符。find_first_of()返回的迭代器保存在 iter 中，然后它被用来和 text 的结束迭代器进行比较。如果 iter 现在是 text 的结束迭代器，循环结

束。如果 iter 不是 text 的结束迭代器，循环体会将这个字符附加到 iter 指向的 found 字符串上，然后自增 iter，使它指向下一个字符。这个字符会作为下一次搜索的开始位置。这段代码产生的输出为：

```
The characters "eooeai" were found in text.
```

另一个版本的 find_first_of()可以让我们在第二个序列中搜索指定范围内的，可以使第 5 个参数指定的二元谓词返回 true 的元素。这个序列中的元素不必是同种类型。当这个==运算符不支持所有这些元素的比较时，就可以用这个版本的算法来定义相等比较，但也可以用其他的方式来定义。例如：

```
std::vector<long> numbers {64L, 46L, -65L, -128L, 121L, 17L, 35L, 9L, 91L, 5L};
int factors[] {7, 11, 13};
auto iter = std::find_first_of(std::begin(numbers),
                               std::end(numbers),// The range to be searched
                      std::begin(factors), std::end(factors),
                                                 // Elements sought
                      [](long v, long d) { return v % d == 0;});
                                                 // Predicate - true for a match
 if(iter != std::end(numbers)) std::cout << *iter << " was found." << std::endl;
```

这个谓词是一个 lambda 表达式，当第一个参数可以被第二个参数整除时，它返回 true。所以这段代码会找到-65，因为这是 numbers 中第一个可以被 factors 数组中的元素 13 整除的元素。断言中的参数类型可以和序列元素的类型不同，只要每个序列中的元素可以隐式转换为参数所对应的类型。在这里，factors 中的元素可以隐式转换为 long 类型。

当然，可以用循环来查找所有使谓词返回 true 的元素：

```
std::vector<long> numbers {64L, 46L, -65L, -128L, 121L, 17L, 35L, 9L, 91L, 5L};
int factors[] {7, 11, 13};
std::vector<long> results;                          // Stores elements found
 auto iter = std::begin(numbers);
while((iter = std::find_first_of(iter, std::end(numbers),  // Range searched
                                 std::begin(factors), std::end(factors),
                                                 // Elements sought
                                [](long v, long d) { return v % d == 0; }))
                                                 // Predicate
                                != std::end(numbers))
 results.push_back(*iter++);
std::cout << results.size() << " values were found:\n";
std::copy(std::begin(results), std::end(results),
  std::ostream_iterator < long > {std::cout, " " });
std::cout << std::endl;
```

这段代码可以找出 numbers 中所有以 factors 中的一个元素为因子的元素。while 循环会持续进行，只要 find_first_of()返回的迭代器不是 numbers 的结束迭代器。iter 变量开始时指向 numbers 的第一个元素，然后会用它来保存被找到元素的迭代器，先前的值被覆盖。

在循环体中，iter 指向的元素会被保存到 result 容器中，然后 iter 自增指向下一个元素。当循环结束时，results 包含的是找到的所有元素，然后用 copy()算法输出它们。

6.3.3 在序列中查找多个元素

adjacent_find()算法可以用来搜索序列中两个连续相等的元素。用==运算符来比较连续的一对元素，返回的迭代器指向前两个相等元素中的第一个。如果没有一对相等的元素，这个算法返回这个序列的结束迭代器。例如：

```
string saying {"Children should be seen and not heard."};
auto iter = std::adjacent_find(std::begin(saying), std::end(saying));
if(iter != std::end(saying))
   std::cout << "In the following text:\n\"" << saying << "\"\n'"
             << *iter << "' is repeated starting at index position "
             << std::distance(std::begin(saying), iter) << std::endl;
```

这里会搜索 saying 字符串的前两个相等字符，所以这段代码的输出如下：

```
In the following text:
"Children should be seen and not heard."
'e' is repeated starting at index position 20
```

adjacent_find()算法的第二个版本，允许我们提供一个应用于连续元素的谓词。下面展示了如何用这个函数来查找这个序列中第一对都为奇数的连续整数：

```
std::vector<long> numbers {64L, 46L, -65L, -128L, 121L, 17L, 35L, 9L, 91L, 5L};
auto iter = std::adjacent_find(std::begin(numbers), std::end(numbers),
  [](long n1, long n2){ return n1 % 2 && n2 % 2; });

if(iter != std::end(numbers))
  std::cout << "The first pair of odd numbers is "
            << *iter << " and " << *(iter+1) << std::endl;
```

当两个参数都不能被 2 整除时，这个 lambda 表达式就返回 true，所以这段代码会找到 121 和 17。

1. find_end()算法

find_end()会在一个序列中查找最后一个和另一个元素段匹配的匹配项，也可以看作在一个元素序列中查找子序列的最后一个匹配项。这个算法会返回一个指向子序列的最后一个匹配项的第一个元素的迭代器，或是一个指向这个序列的结束迭代器。下面是一个示例：

```
string text {"Smith, where Jones had had \"had\", had had \"had had\"."
             " \"Had had\" had had the examiners\' approval."};
std::cout << text << std::endl;
string phrase {"had had"};
auto iter = std::find_end(std::begin(text), std::end(text),
std::begin(phrase), std::end(phrase));
```

```
if(iter != std::end(text))
std::cout << "The last \"" << phrase
          << "\" was found at index " << std::distance(std::begin(text),
             iter) << std::endl;
```

这段代码会从 text 中搜索“had had”的最后一个匹配项，并输出如下内容：

```
    Smith, where Jones had had "had", had had "had had". "Had had" had had the
examiners' approval.
    The last "had had" was found at index 63
```

可以在 text 中搜索 phrase 的所有匹配项。在这个示例中只会记录匹配项的个数：

```
size_t count {};
auto iter = std::end(text);
auto end_iter = iter;
while((iter = std::find_end(std::begin(text), end_iter, std::begin(phrase),
       std::end(phrase))) != end_iter)
{
  ++count;
  end_iter = iter;
}
std::cout << "\n\""<< phrase << "\" was found " << count << " times." << std::endl;
```

这个 while 循环表达式会进行这项搜索工作。循环表达式会在[std::begin(text), end_iter)这个范围内搜索 phrase。最开始的搜索范围包含 text 中的全部元素。为了说明这里发生了什么，下面用图 6-5 来演示这个过程。

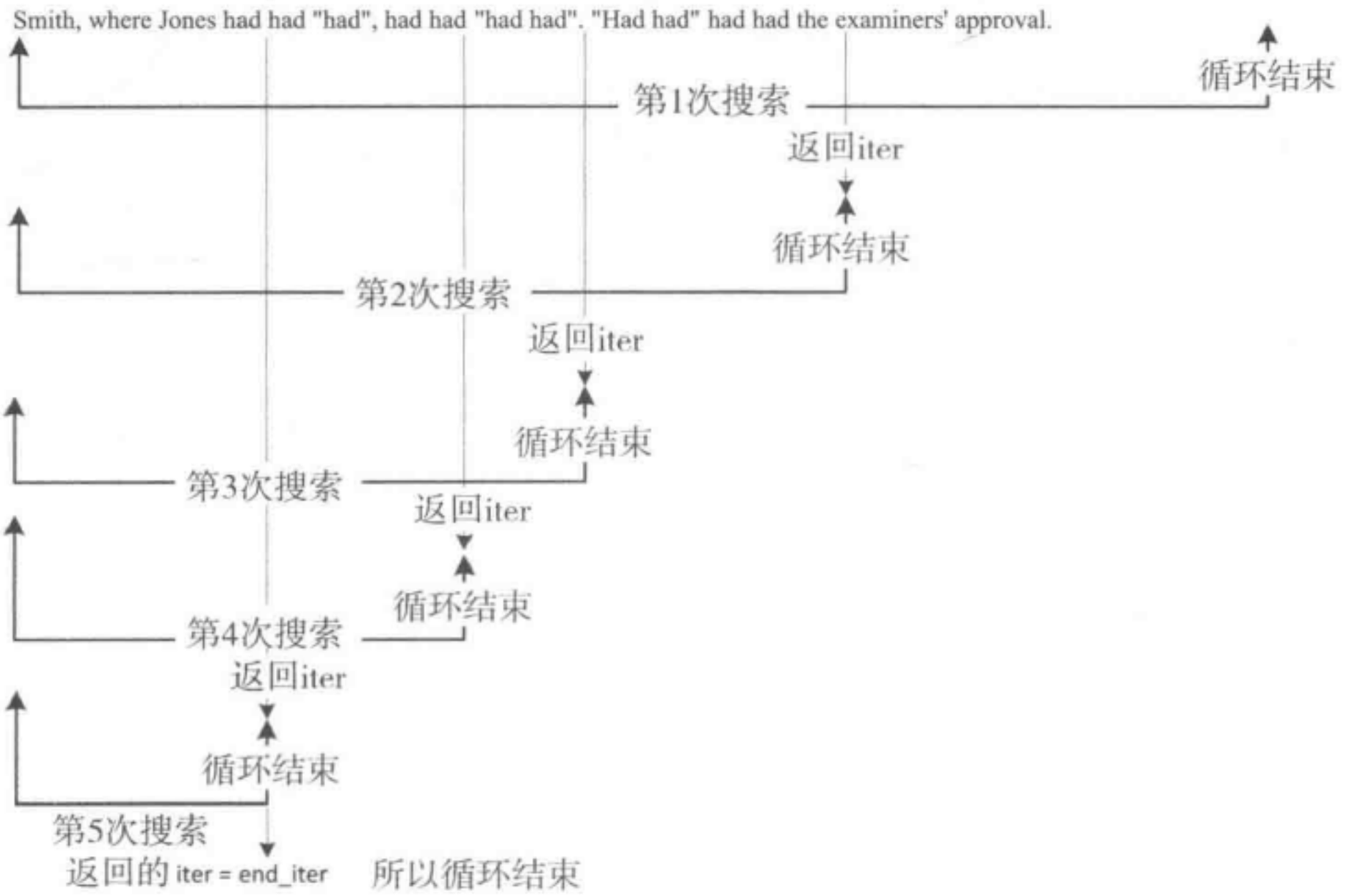


图 6-5　用 find_end()反复搜索

find_end()返回的迭代器被保存在 iter 中，当它等于 end_iter——iter 先前的值时，循环结束。因为 find_end()会查找子序列的最后一个匹配项，下一个搜索范围的结束迭代器 end_iter 会变为这个算法所返回的迭代器。这个迭代器会指向被找到序列的第一个字符，所以下一次搜索会从 text 的这个点开始，忽略已找到的元素。循环体中 count 自增后，end_iter 被设为 iter。这么做是必要的，因为如果下一次搜索找到 phrase，就会返回这个迭代器。

另一个版本的 find_end()接受一个用来比较元素的二元谓词作为第 5 个参数，可以用它来重复前面的搜索并忽略大小写：

```
size_t count {};
auto iter = std::end(text);
auto end_iter = iter;
while((iter = std::find_end(std::begin(text), end_iter, std::begin(phrase),
  std::end(phrase),[](char ch1, char ch2){ return std::toupper(ch1) == std::
       toupper(ch2); })) != end_iter)
{
  ++count;
  end_iter = iter;
}
```

在将字符转换为大写后，会按对比较这两个序列中的元素。在 text 中会找到 phrase 的 5 个实例，因为找到的"Had had"也和 phrase 相等。

2. search()算法

在查找序列的子序列方面，search()算法和 find_end()算法相似，但它所查找的是第一个匹配项而不是最后一个。和 find_end()算法一样，它也有两个版本——第二个版本接受用来比较元素的谓词作为第 5 个参数。可以用 search()来验证前面使用 find_end()搜索的结果。如何改变每次遍历搜索的具体范围是它们的主要不同之处。下面是一个示例：

```
string text {"Smith, where Jones had had \"had\", had had \"had had\"."
             " \"Had had\" had had the examiners\' approval."};
std::cout << text << std::endl;
string phrase {"had had"};
size_t count {};
auto iter = std::begin(text);
auto end_iter = end(text);
while((iter = std::search(iter, end_iter, std::begin(phrase),
std::end(phrase),[](char ch1, char ch2){ return std::toupper(ch1) == std::toupper
       (ch2); })) !=
       end_iter)
{
 ++count;
 std::advance(iter, phrase.size()); // Move to beyond end of subsequence found
}
 std::cout << "\n\""<< phrase << "\" was found " << count << " times." << std::endl;
```

这段代码执行后会输出下面的内容：

```
    Smith, where Jones had had "had", had had "had had". "Had had" had had the
examiners' approval.
    "had had" was found 5 times.
```

我们仍然忽略大小写来搜索“had had”，但会正向查找第一个匹配项。search()算法返回的迭代器指向找到的子序列的第一个元素，因此为了搜索 phrase 的第二个实例，iter 必须增加 phrase 中元素的个数，使它指向下一个找到的序列的第一个元素。

3. search_n()算法

search_n()算法会搜索给定元素的匹配项，它在序列中连续出现了给定的次数。它的前两个参数是定义搜索范围的正向迭代器，第 3 个参数是想要查找的第 4 个元素的连续匹配次数。例如：

```
std::vector<double> values {2.7, 2.7, 2.7, 3.14, 3.14, 3.14, 2.7, 2.7};
double value {3.14};
int times {3};
auto iter = std::search_n(std::begin(values), std::end(values), times, value);
if(iter != std::end(values))
  std::cout << times << " successive instances of " << value
            << " found starting index " << std::distance(std::begin(values), iter)
            << std::endl;
```

这段代码会在 values 容器中查找第一个有 3 个 value 实例的匹配项。它找到的序列的索引为 3。注意用来指定个数的第三个参数必须是无符号整型；如果不是，编译这段代码会产生警告。

这里用==来比较元素，但也可以提供额外的断言参数来代替它。当然，这不需要定义一个判断是否相等的比较。下面是一个完整的示例：

```
// Ex6_03.cpp
// Searching using search_n() to find freezing months
#include <iostream>                    // For standard streams
#include <vector>                      // For vector container
#include <algorithm>                   // For search_n()
#include <string>                      // For string class
using std::string;
int main()
{
std::vector<int> temperatures {65, 75, 56, 48, 31, 28, 32, 29, 40, 41, 44, 50};
int max_temp {32};
int times {3};
auto iter = std::search_n(std::begin(temperatures), std::end(temperatures),
  times, max_temp,[](double v, double max){return v <= max; } );
std::vector<string> months {"January", "February", "March", "April", "May", "June",
      "July", "August", "September", "October", "November", "December"};
```

```
if(iter != std::end(temperatures))
  std::cout << "It was " << max_temp << " degrees or below for " << times
            << " months starting in " << months[std::distance(std::begin
              (temperatures), iter)]
            << std::endl;
}
```

temperatures 容器中保存了一年中每个月份的平均温度。search_n()的最后一个参数是一个 lambda 表达式的谓词，当元素小于等于 max_temp 时，它会返回 true。month 容器中保存月份的名称。表达式 std::distance(std::begin(temperatures), iter)会返回 temperatures 这个序列的第一个元素的索引，这个序列可以使谓词返回 true。这个值用来索引选择 months 中月份的名称，所以这段代码会输出如下内容：

```
It was 32 degrees or below for 3 months starting in May
```

6.4 分区序列

在序列中分区元素会重新对元素进行排列，所有使给定谓词返回 true 的元素会被放在所有使谓词返回 false 的元素的前面。这就是 partition()算法所做的事。它的前两个参数是定义被分区序列范围的正向迭代器，第三个参数是一个谓词。下面展示，如何使用 partition()来重新排列序列中的值，所有小于平均值的元素会被放在所有大于平均值的元素的后面：

```
std::vector<double> temperatures {65, 75, 56, 48, 31, 28, 32, 29, 40, 41,
  44, 50};
std::copy(std::begin(temperatures), std::end(temperatures), // List the values
          std::ostream_iterator<double>{std::cout, " "});

std::cout << std::endl;

auto average = std::accumulate(std::begin(temperatures), // Compute the
                                                         // average value
               std::end(temperatures), 0.0)/ temperatures.size();
std::cout << "Average temperature: " << average << std::endl;

std::partition(std::begin(temperatures), std::end(temperatures),
  //Partition the values
  [average](double t) { return t < average; });
std::copy(std::begin(temperatures), std::end(temperatures), // List the values
                                                // after partitioning
                                    std::ostream_iterator<double>{std::cout, " "});
std::cout << std::endl;
```

这段代码会输出下面这些内容：

```
65 75 56 48 31 28 32 29 40 41 44 50
Average temperature: 44.9167
44 41 40 29 31 28 32 48 56 75 65 50
```

通过 accumulate()算法创建的元素之和除以元素个数，计算出 temperatures 容器中元素的平均值。在前面，我们已经介绍了 accumulate()算法，所以，我们记得它的第三个参数就是和的初始值。执行 partition()算法后，可以看到所有小于平均值的温度值都在大于平均值的温度值之前。

这个谓词可以不必是用来处理顺序关系的——它可以是我们喜欢的任何样子。例如，可以对表示个体的 Person 对象进行分区，将所有女性放在男性的前面，或者将有大学学历的放在没有大学学历的前面。下面是一个对 tuple 对象的序列进行分区的示例，这个元组对象用来表示人和标识他们的性别：

```
using gender = char;
using first = string;
using second= string;
using Name = std::tuple<first, second, gender>;
std::vector<Name> names {std::make_tuple("Dan", "Old", 'm'),
                         std::make_ tuple("Ann", "Old", 'f'),
                         std::make_tuple("Ed", "Old", 'm'),
                         std::make_tuple("Jan", "Old", 'f'),
                         std::make_tuple("Edna", "Old", 'f')};
std::partition(std::begin(names), std::end(names), // Partition the names
               [](const Name& name) { return std::get<2>(name) == 'f'; });
for(const auto& name : names)
  std::cout << std::get<0>(name) << " " << std::get<1>(name) << std::endl;
```

这里使用 using 声明来解释 tuple 对象成员变量的意义。当 tuple 对象的最后一个成员变量是“f”时，这个谓词会返回 true，所以输出中会出现 Edna、Ann 以及处在 Ed 和 Dan 之前的 Jan。在这个谓词中，可以用表达式 std::get<gender>(name)来引用 tuple 的第三个成员变量。这样做是可行的，因为第三个成员是唯一的，这就允许用它的类型来识别这个成员。

partition()算法并不保证维持这个序列原始元素的相对顺序。在上面的示例中，对于原始序列，元素 44 和 41 在 40 的后面。但在进行这项操作之后，它们就不是那样了。为了维持元素的相对顺序，可以使用 stable_partition()算法。它的参数和 partition()一样，可以用下面这些语句来代替前一段代码中的 partition()调用：

```
std::stable_partition(std::begin(temperatures), std::end(temperatures),
                      [average](double t) { return t < average; });
```

做出这些修改后，对应的输出如下：

```
65 75 56 48 31 28 32 29 40 41 44 50
Average temperature: 44.9167
31 28 32 29 40 41 44 65 75 56 48 50
```

可以看到，重排序时并不一定要对序列进行分区，元素的相对顺序被保留了。所有小于平均值的元素的相对顺序都没有被改变，所有大于平均值的元素也是如此。

6.4.1 partition_copy()算法

partition_copy()算法以和 stable_partition()相同的方式对序列进行分区，但那些使谓词返回 true 的元素会被复制到一个单独的序列中，使谓词返回 false 的那些元素会被复制到第三个序列中。这个操作不会改变原始序列。原始序列由前两个参数指定，它们必须是输入迭代器。第 3 个参数用来确定目的序列的开始位置，它会保存那些使谓词返回 true 的元素。第 4 个参数用来确定另一个目的序列的开始位置，它会保存那些使谓词返回 false 的元素。第 5 个参数是用来分区元素的谓词。下面是一个展示 partition_copy()用法的完整程序：

```
// Ex6_04.cpp
// Using partition_copy() to find values above average and below average
#include <iostream>                    // For standard streams
#include <vector>                      // For vector container
#include <algorithm>                   // For partition_copy(), copy()
#include <numeric>                     // For accumulate()
#include <iterator>                    // For back_inserter, ostream_iterator

int main()
{
  std::vector<double> temperatures {65, 75, 56, 48, 31, 28, 32, 29, 40, 41, 44, 50};
  std::vector<double> low_t;           // Stores below average temperatures
  std::vector<double> high_t;          // Stores average or above temperatures

  auto average = std::accumulate(std::begin(temperatures),
    std::end(temperatures), 0.0) /temperatures.size();

  std::partition_copy(std::begin(temperatures), std::end(temperatures),
                      std::back_inserter(low_t), std::back_inserter(high_t),
                           [average](double t) { return t < average; });

  // Output below average temperatures
  std::copy(std::begin(low_t), std::end(low_t), std::ostream_iterator<double>
     {std::cout, " "});
  std::cout << std::endl;

  // Output average or above temperatures
  std::copy(std::begin(high_t), std::end(high_t), std::ostream_iterator<double>
    {std::cout, " "});
  std::cout << std::endl;
}
```

这段代码所做的事情和先前介绍的 stable_partition()相同，但小于平均值的元素会被复制到 low_t 容器中，大于等于平均值的元素会被复制到 high_t 容器中。输出语句可以对此进行验证，它们产生的输出如下：

```
31 28 32 29 40 41 44
65 75 56 48 50
```

注意，main()中的这段代码使用辅助函数 back_inserter()创建的 back_insert_iterator 对象

作为partition_copy()调用中两个目的容器的迭代器。back_insert_iterator通过调用push_back()向容器中添加新元素，使用这种方式可以不需要提前知道容器中保存了多少元素。如果对目的序列使用开始迭代器，在执行这个操作前，为了可以复制尽可能多的元素，目的序列中必须有足够的元素。注意，如果输入序列和输出序列重叠，这个算法将无法正常工作。

6.4.2　partition_point()算法

可以用 partition_point()算法来获取分区序列中第一个分区的结束迭代器，它的前两个参数定义检查范围的正向迭代器，最后一个参数是用来对序列进行分区的谓词。我们通常不知道每个分区中元素的个数，这个算法使我们能够访问和提取这些分区中的元素。例如：

```
std::vector<double> temperatures {65, 75, 56, 48, 31, 28, 32, 29, 40, 41, 44, 50};
auto average = std::accumulate(std::begin(temperatures), // Compute the
                                                         // average value
                  std::end(temperatures), 0.0)/ temperatures.size();
auto predicate = [average](double t) { return t < average; };
std::stable_partition(std::begin(temperatures),
  std::end(temperatures), predicate);
auto iter = std::partition_point(std::begin(temperatures),
  std::end(temperatures), predicate);

std::cout << "Elements in the first partition: ";
std::copy(std::begin(temperatures), iter,
  std::ostream_iterator<double>{std:: cout, " "});
std::cout << "\nElements in the second partition: ";
std::copy(iter, std::end(temperatures),
  std::ostream_iterator<double>{std::cout, " "});
std::cout << std::endl;
```

这段代码会相对于平均温度对 temperatures 中的元素进行分区，并通过调用 partition_point()找到这个序列的分区点。这个分区点是第一个分区的结束迭代器，它被保存在 iter 中。所以[std::begin(temperatures),iter)对应的就是第一个分区中的元素，[iter, std::end(temperatures))包含的是第二个分区中的元素。这里使用两次 copy()算法来输出分区，输出内容如下：

```
Elements in the first partition: 31 28 32 29 40 41 44
Elements in the second partition: 65 75 56 48 50
```

在使用 partition_point()之前，需要确定序列是否已经被分区。如果对此不是很确定，在这种情况下可以使用 is_partitioned()来判断。它的参数是用来指定序列的输入迭代器和用来对序列进行分区的谓词。如果这个序列已经被分区，这个算法就返回 true，否则返回 false。在对 temperatures 使用 partition_point()算法之前，可以先用它来验证这个序列：

```
if(std::is_partitioned(std::begin(temperatures), std::end(temperatures),
                       [average](double t) { return t < average; }))
{
  auto iter = std::partition_point(std::begin(temperatures),
```

```
        std::end(temperatures),[average](double t) { return t < average; });
    std::cout << "Elements in the first partition: ";
    std::copy(std::begin(temperatures), iter, std::ostream_iterator<double>
     {std::cout, " "});
    std::cout << "\nElements in the second partition: ";
    std::copy(iter, std::end(temperatures), std::ostream_iterator<double>
     {std::cout, " "});
    std::cout << std::endl;
  }
  else
    std::cout << "Range is not partitioned." << std::endl;
```

只有在 is_partitioned()返回 true 时，这段代码才会执行。如果 if 语句为 true，iter 变量会指向分区点。如果想在后面继续使用 iter，可以按如下方式在 if 语句之前定义它：

```
std::vector<double>::iterator iter;
```

在所有容器类型的模板中都定义了这样的迭代器类型别名，它们使迭代器变得可以使用。它对应于这个容器类型的成员函数 begin()和 end()所返回的迭代器类型。

6.5 二分查找算法

目前你在本章中见到的搜索算法都是对序列进行顺序搜索，而且没有事先对元素进行排序的要求。二分查找一般比顺序搜索要快，但要求序列中的元素是有序的。这主要是因为二分查找的搜索机制，图 6-6 说明了这种机制。

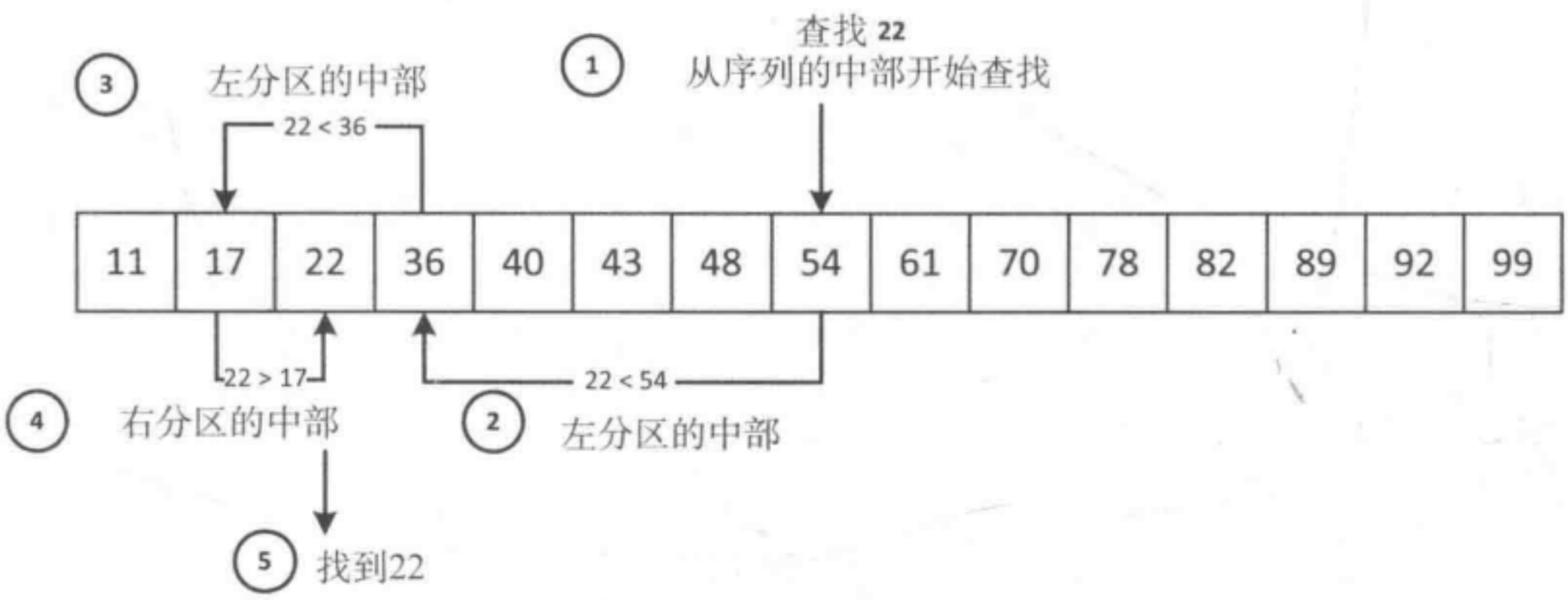


图 6-6　二分查找

图 6-6 展示了在一个升序序列中二分查找 22 的过程。因为元素是升序排列的，所以查找机制使用小于运算符来查找元素。搜索降序序列需要使用小于运算符来比较元素。二分查找总是选择从序列中部的元素开始，并将它和搜索的值作比较。如果元素和被查找的元素相等，就认为是匹配的，所以当!(x < n) && !(n < x)时，n 的值也就匹配 x 的值。如果检查的元素不匹配，比如 x<n，会继续从左分区的中间元素开始查找，否则继续从右分区的

中间元素开始查找。当找到相等的元素或所检查的分区只有一个元素时，查找结束。此时如果不匹配，就说明元素不在这个序列中。

6.5.1　binary_search()算法

正如你们所想的那样，binary_search()实现了一个二分查找算法。它会在前两个参数指定范围内搜索等同于第三个参数的元素。指定范围的迭代器必须是正向迭代器而且元素必须可以使用<运算符来比较。这个序列中的元素必须被排成升序序列或者至少相对于所查找元素是有序的。如果找到第三个参数，这个算法会返回布尔值 true，否则返回 false。所以它能告诉我们元素是否在这个序列中，但当它在序列中时，却不能告诉我们它的位置。当然，如果必须知道它的位置，可以使用前面介绍的查找算法或 lower_bound()、upper_bound()或 equal_range()。下面是一个使用 binary_search()的示例：

```
std::list<int> values {17, 11, 40, 36, 22, 54, 48, 70, 61, 82, 78, 89, 99,
  92, 43};
values.sort();                              // Sort into ascending sequence
int wanted {22};                            // What we are looking for
if(std::binary_search(std::begin(values), std::end(values), wanted))
  std::cout << wanted << " is definitely in there - somewhere..." << std::endl;
else
  std::cout << wanted << " cannot be found - maybe you got it wrong..."
            << std::endl;
```

这里用 list 来保存一个任意顺序的任意值的集合——这只是为了提醒你记住这个容器的用法。这段代码使用 binary_search()算法来查找期望的值。因为 binary_search()只能使用有序序列，所以首先我们确保 list 中的元素是有序的。不能对 list 容器中的元素应用 sort()算法，因为它需要的是随机访问迭代器，而 list 容器只提供了双向迭代器。因为这个 list 定义了一个成员函数 sort()，可以将全部的元素排成升序，所以可以用这个函数来对容器中的元素进行排序。当这段代码执行时，会输出是否找到希望值的消息。

另一个版本的 binary_search()接受一个额外的参数，它是一个用于查找元素的函数对象；显然，它必须和用于对被查找序列进行排序的比较操作有相同的效果。下面是一个演示如何将值排成降序，然后查找期望值的示例：

```
std::list<int> values {17, 11, 40, 36, 22, 54, 48, 70, 61, 82, 78, 89, 99,
  92, 43};
auto predicate = [](int a, int b){ return a > b; };
values.sort(predicate); // Sort into descending sequence
int wanted {22};
if(std::binary_search(std::begin(values), std::end(values), wanted, predicate))
  std::cout << wanted << " is definitely in there - somewhere..." << std::endl;
else
std::cout << wanted << " cannot be found - maybe you got it wrong..."
          << std::endl;
```

这里使用的 list 容器的成员函数 sort()接受一个定义比较的函数对象作为参数，这里它

是由一个 lambda 表达式定义的。同一个 lambda 表达式也被作为 binary_search()的第 4 个参数。当然，结果和前一段代码相同。

6.5.2 lower_bound()算法

lower_bound()算法可以在前两个参数指定的范围内查找不小于第三个参数的第一个元素——也就是说，大于等于第三个参数的第一个元素。前两个参数必须是正向迭代器。upper_bound()算法会在前两个参数定义的范围内查找大于第三个参数的第一个元素。对于这两个算法，它们所查找的序列都必须是有序的，而且它们被假定是使用<运算符来排序的。例如：

```
    std::list<int> values {17, 11, 40, 36, 22, 54, 48, 70, 61, 82, 78, 89, 99,
92, 43};
    values.sort();                          // Sort into ascending sequence
    int wanted {22};                        // What we are looking for
    std::cout << "The lower bound for " << wanted
              << " is " << *std::lower_bound(std::begin(values),
                 std::end(values), wanted)
              << std::endl;
    std::cout << "The upper bound for " << wanted
              << " is " << *std::upper_bound(std::begin(values), std::end(values),
                 wanted)
              << std::endl;
```

```
The lower bound for 22 is 22
The upper bound for 22 is 36
```

从 list 容器的整数中可以看出算法正像我们所描述的那样工作。这两个算法都有额外的版本，它们接受一个函数对象作为第三个参数，用于指定序列排序所使用的比较。

6.5.3 equal_range()算法

equal_range()可以找出有序序列中所有和给定元素相等的元素。它的前两个参数是指定序列的两个正向迭代器，第三个参数是要查找的元素。这个算法会返回一个 pair 对象，它有两个正向迭代器成员，其中的 first 指向的是不小于第三个参数的一个元素，second 指向大于第三个参数的一个元素，所以我们也可以通过在单个调用中调用 lower_bound()和 upper_bound()来得到同样的结果。可以用下面这些语句来替换前一个代码段中的两条输出语句：

```
auto pr = std::equal_range(std::begin(values), std::end(values), wanted);
std::cout << "the lower bound for " << wanted << " is " << *pr.first
          << std::endl;
std::cout << "the upper bound for " << wanted << " is " << *pr.second
          << std::endl;
```

它和前一段代码的输出完全相同。和前面的二分查找算法一样，equal_range()也有一

个有额外参数的版本，这个参数可以为有序序列提供一些不同于<运算符的比较。

已经说过，这一节的算法要求它们所处理的序列的元素是有序的，但这并不是全部。所有的二分查找算法都可以用于以特殊方式分区的序列。对于一个给定的希望值，序列中的元素必须按照(element < wanted)和!(wanted < element)来分区。可以用 equal_range()二分查找算法来做这些工作，在对 values 容器中的元素执行 equal_range()之前，可以按如下方式对它进行分区：

```
std::list<int> values {17, 11, 40, 36, 22, 54, 48, 70, 61, 82, 78, 89, 99, 92, 43};
// Output the elements in original order
std::copy(std::begin(values), std::end(values), std::ostream_iterator<int> {std::
  cout, " "});
std::cout << std::endl;

int wanted {22};                              // What we are looking for

std::partition(std::begin(values), std::end(values),
// Partition the values wrt value < wanted
              [wanted](double value) { return value < wanted; });
std::partition(std::begin(values), std::end(values),
// Partition the values wrt !(wanted < value)
              [wanted](double value) { return !(wanted < value); });

// Output the elements after partitioning
std::copy(std::begin(values), std::end(values), std::ostream_iterator<int>
  {std::cout, " "});
std::cout << std::endl;
```

这段代码的输出如下：

```
17 11 40 36 22 54 48 70 61 82 78 89 99 92 43
17 11 22 36 40 54 48 70 61 82 78 89 99 92 43
```

第一行显示的是元素原始的顺序，第二行显示的是分区之后的顺序。两次分区操作改变了元素的顺序，但改变的并不多，现在我们可以将 equal_range()应用到 values 容器的元素上，期望值为 wanted：

```
auto pr = std::equal_range(std::begin(values), std::end(values), wanted);
std::cout << "the lower bound for " << wanted << " is " << *pr.first
          << std::endl;
std::cout << "the upper bound for " << wanted << " is " << *pr.second
          << std::endl;
```

这段代码和前一段代码的输出是相同的，在前一段代码中，用容器对象的成员函数 sort()对元素进行了完全排序。本节的所有算法都可以用于以这种方式分区的序列上。显然，如果分区使用的是>，那么在查找算法中使用的函数对象也必须和它保持一致。

在前一段代码中，将 equal_range()应用到了一个只包含单个期望值的序列中。如果这个序列包含多个实例，pr.first 会指向 wanted 的第一个匹配项，所以[pr.first, pr.second)这个

范围包含的是所有的匹配项。下面是一个示例：

```
// Ex 6_05.cpp
// Using partition() and equal_range() to find duplicates of a value in a
   // range
#include <iostream>              // For standard streams
#include <list>                  // For list container
#include <algorithm>             // For copy(), partition()
#include <iterator>              // For ostream_iterator

int main()
{
  std::list<int> values {17, 11, 40, 13, 22, 54, 48, 70, 22, 61, 82, 78, 22,
   89, 99, 92, 43};

  // Output the elements in their original order
  std::cout << "The elements in the original sequence are:\n";
  std::copy(std::begin(values), std::end(values), std::ostream_iterator
   <int> {std::cout, " "});
  std::cout << std::endl;

  int wanted {22};                          // What we are looking for

  std::partition(std::begin(values), std::end(values), // Partition the values with
                                                       // (value < wanted)
                 [wanted](double value) { return value < wanted; });
  std::partition(std::begin(values), std::end(values), // Partition the values
                                                       // with !(wanted < value)
                 [wanted](double value) { return !(wanted < value); });

  // Output the elements after partitioning
  std::cout << "The elements after partitioning are:\n";
  std::copy(std::begin(values), std::end(values), std::ostream_iterator<int>
    {std::cout, " "});
  std::cout << std::endl;

  auto pr = std::equal_range(std::begin(values), std::end(values), wanted);
  std::cout << "The lower bound for " << wanted << " is " << *pr.first
            << std::endl;
  std::cout << "The upper bound for " << wanted << " is " << *pr.second
            << std::endl;

  std::cout << "\nThe elements found by equal_range() are:\n";
  std::copy(pr.first, pr.second, std::ostream_iterator<int> {std::cout, " "});
  std::cout << std::endl;
}
```

输出如下所示：

```
The elements in the original sequence are:
17 11 40 13 22 54 48 70 22 61 82 78 22 89 99 92 43
The elements after partitioning are:
```

```
17 11 13 22 22 22 48 70 54 61 82 78 40 89 99 92 43
The lower bound for 22 is 22
The upper bound for 22 is 48
The elements found by equal_range() are:
22 22 22
```

values 容器中有一些值为 22 的元素，22 也是 wanted 的值。equal_range()返回了 wanted 在这个序列中的三个实例。这个序列只是被分区了，并不是完全有序的，当序列完全有序时，显然也同样适应。

所以当序列像 Ex6_05 那样分区并且不是完全有序时，为什么 equal_range()会返回 wanted 的所有匹配项？为了弄明白这一点，需要理解两个 partition()调用的作用：

- 第一个分区操作保证严格小于 wanted 的所有元素都在左分区中，这些元素不需要是有序的。这个操作也保证所有大于等于 wanted 的元素都在右分区中，所以它们都在 wanted 的后面，而且也不需要是有序的。wanted 的所有匹配项都在右分区中，但是和大于 wanted 的元素混合在了一起。前一段代码中，在第一次调用 partition()后，values 中的元素为：

```
17 11 13 40 22 54 48 70 22 61 82 78 22 89 99 92 43
```

17、11、13 是仅有的小于 wanted 的几个值，它们显然在左分区中。分区并不能以任何特别的方式来确定和 wanted 所对应值的位置。22 的全部实例可能出现在右分区的任何位置。

- 第二个分区操作会被应用到第一个操作的结果上。表达式!(wanted <value)等同于(value <=wanted)。因此作为结果，小于等于 wanted 的所有元素会出现在左分区中，而且所有严格大于 wanted 的元素会在右分区中。这个操作的效果是将所有 wanted 的实例移到左分区中，所以它们被一起作为左分区的最后一个元素。在第二次调用 partition()后，values 包含的元素如下：

```
17 11 13 22 22 22 48 70 54 61 82 78 40 89 99 92 43
```

equal_range()找到的下边界指向 22 的第一个匹配项，上边界指向 22 的最后一个匹配项的后面——值为 48 的元素。

6.6　本章小结

如果使用 STL 容器，可以轻松地将它们和本章介绍的算法结合在一起。排序是一个很常见的要求，尤其是在那些处理事务的程序中。当需要对数据进行排序时，通常也需要合并数据。保证事务和所记录的顺序一样可以使更新过程更快。当然，有序的 set 和 map 容器，以及 priority_queue 容器适配器提供的元素默认都是有序的，它们都不需要显式地提供排序操作。然而，当需要在不同时刻的不同序列中使用相同的数据时，需要使用排序操作按要求对对象进行重新排列。这时 STL sort()算法就表现出它们灵活的一面——可以对我们

相比较的任何事物进行比较，另一方面，它们是非常有效率的——它们的实现几乎可以肯定比你自己实现的排序算法都要好。这并不是说 STL 算法总是最好的选择。有许多特例化的数据排序算法 C++标准库并没有实现，但当使用 STL 容器管理数据时，只是为了使用它的通用排序功能，STL 提供了方便的解决方案。

STL 查找算法的应用范围比排序和合并算法更加广泛。它们对所搜索的序列没有要求，只需要元素是可比较的。同样，对于定义序列范围的迭代器所要求的功能也是最小的。当序列有序时，二分查找算法是对查找算法的补充。最后，分区算法提供了无序和有序序列的转换。如你所见，可以将二分查找算法应用到分区序列中，在序列不是完全有序时，使我们仍然可以找到相等元素的序列。

练习

1. 定义一个 Card 类来表示扑克牌。创建一个 Card 对象的 vector，表示完整的 56 张牌。将这些牌随机分配到 4 个 vector 容器中，用来表示游戏中每手 13 张牌。在对每一手牌使用 sort()算法排序后，将 4 个 vector 容器的牌分别放到北、南、西、东 4 个标题的下面。这些牌应该按照花色和大小排序——所以牌的大小是从 2 到 10、J、Q、K、A。花色是按照梅花、方块、红心和黑桃的顺序。

2. 在练习 1 的基础上添加代码，合并 4 手牌，并输出合并结果。

3. 定义一个 Person 类，它至少可以通过人的姓名和头发的颜色来识别一个人。这个类应该为流实现 operator<<()，而且需要实现一个比较头发颜色的成员函数。创建一个包含 Person 对象的 vector，它所包含的 Person 对象并没有特别的顺序，这些 Person 对象的头发颜色分为金色、灰色、棕色和黑色。用 partition()算法按照颜色的顺序对这些容器中的对象进行重新排列，黑色在前，其次为灰色，然后是棕色，金色在最后。用 copy()算法按照颜色的分组输出这些 Person 对象。

4. 为练习 3 中的 Person 类添加一个用来比较名字的成员函数，然后对练习 3 中的解决方案进行扩展。在输出它们之前，在姓名按升序排列的序列中用 sort()算法按给定颜色对它们进行排序。

在公职候选人竞选时，为了防止对候选人有偏见，候选人的名字通常是随机排列的。例如，如果按照字母顺序，Joe Yodel 和 Bob Zippo 都会出现在后面。定义一个 Name 类，它按照下面的字母顺序对姓名进行排序：

R W Q O J M V A H B S G Z X N T C I E K U P D Y F L

所以，以 R 开头的姓名会排在前面，以 L 开头的排在最后。在 vector 容器中创建各种 Name 的对象，然后用 stable_sort()算法根据上面字母的顺序将它们降序排列。

第 7 章

更多的算法

本章会介绍 STL 提供的更多算法。这些算法通常可以分为两类：会改变它们所应用序列的算法以及不改变它们所应用序列的算法。本章会按照它们的用法分类，而不会按照是否改变序列来分类。如果知道算法做了些什么，显然就会知道它是否改变它所应用的数据。本章将介绍以下内容：

- 检查序列中元素的属性的算法
- 计算序列中有指定属性的元素个数的算法
- 比较两个序列中元素的算法
- 用来复制或移动序列的算法
- 设置或改变序列中元素的算法

7.1 检查元素的属性

algorithm 头文件中定义了 3 种算法，用来检查在算法应用到序列中的元素上时，什么时候使谓词返回 true。这些算法的前两个参数是定义谓词应用范围的输入迭代器；第三个参数指定了谓词。检查元素是否能让谓词返回 true 似乎很简单，但它却是十分有用的。例如，可以检查所有学生是否通过了考试，或者检查所有学生是否都参加了课程，或者检查有没有眼睛发绿的 Person 对象，甚至可以检查每个 Dog 对象是否度过了它自己的一天。谓词可以简单，也可以复杂——这取决于你。检查元素属性的三种算法是：

- all_of()算法会返回 true，前提是序列中的所有元素都可以使谓词返回 true。
- any_of()算法会返回 true，前提是序列中的任意一个元素都可以使谓词返回 true。
- none_of ()算法会返回 true，前提是序列中没有元素可以使谓词返回 true。

想象它们是如何工作的并不难。下面的一些代码用来说明如何使用 none_of()算法：

```
std::vector<int> ages {22, 19, 46, 75, 54, 19, 27, 66, 61, 33, 22, 19};
int min_age{18};
std::cout << "There are "
```

```
        << (std::none_of(std::begin(ages), std::end(ages),
                [min_age](int age) { return age < min_age; }) ? "no": "some")
        << " people under " << min_age << std::endl;
```

这个谓词是一个 lambda 表达式，用来将传入的 ages 容器中的元素和 min_age 的值作比较。用 none_of()返回的布尔值来选择包含在输出信息中的是“no”还是“some”。当 ages 中没有元素小于 min_age 时，none_of()算法会返回 true。在这种情况下，会选择“no”。当然，用 any_of()也能产生同样的结果：

```
std::cout << "There are "
          << (std::any_of(std::begin(ages), std::end(ages),
                  [min_age](int age) { return age < min_age; }) ? "some": "no")
          << " people under " << min_age << std::endl;
```

只有在有一个或多个元素小于 min_age 时，any_of()算法才会返回 true。
这里没有元素小于 min_age，所以也会选择“no”。
下面是一段代码，用来展示用 all_of()检查 ages 容器中的元素：

```
int good_age{100};
std::cout << (std::all_of(std::begin(ages), std::end(ages),
        [good_age](int age) { return age < good_age; }) ? "None": "Some")
          << " of the people are centenarians." << std::endl;
```

这个 lambda 表达式会将 ages 中的元素和 good_age 的值作比较，good_age 的值为 100。所有的元素都小于 100，所以 all_of()会返回 true，而且输出消息会正确报告没有记录的百岁老人。

count 和 count_if 可以告诉我们，在前两个参数指定的范围内，有多少满足指定的第三个参数条件的元素。count()会返回等同于第三个参数的元素的个数。count_if()会返回可以使作为第三个参数的谓词返回 true 的元素个数。下面是一些将这些算法应用到 ages 容器的示例：

```
std::vector<int> ages {22, 19, 46, 75, 54, 19, 27, 66, 61, 33, 22, 19};
int the_age{19};
std::cout << "There are "
          << std::count(std::begin(ages), std::end(ages), the_age)
          << " people aged " << the_age << std::endl;
int max_age{60};
std::cout << "There are "
          << std::count_if(std::begin(ages), std::end(ages),
                  [max_age](int age) { return age > max_age; })
          << " people aged over " << max_age << std::endl;
```

在第一条输出语句中使用 count()算法来确定 ages 中等于 the_age 的元素个数，第二条输出语句使用 count_if()来报告大于 max_age 的元素个数。

当我们想知道序列元素是否有某种特性或有多少满足标准时，本节中的所有算法都可以用来了解关于序列元素的基本特性的信息。如果想要知道具体的——序列中哪个元素匹配——可以使用第 6 章介绍的 find()算法。

7.2　序列的比较

可以用和比较字符串类似的方式来比较序列。如果两个序列的长度相同，并且对应元素都相等，equal()算法会返回 true。有 4 个版本的 equal()算法，其中两个用==运算符来比较元素，另外两个用我们提供的作为参数的函数对象来比较元素，所有指定序列的迭代器都必须至少是输入迭代器。

用==运算符来比较两个序列的第一个版本期望 3 个输入迭代器参数，前两个参数是第一个序列的开始和结束迭代器，第三个参数是第二个序列的开始迭代器。如果第二个序列中包含的元素少于第一个序列，结果是未定义的。用==运算符的第二个版本期望 4 个参数：第一个序列的开始和结束迭代器，第二个序列的开始和结束迭代器，如果两个序列的长度不同，那么结果总是为 false。本章会演示这两个版本，但推荐使用接受 4 个参数的版本，因为它不会产生未定义的行为。下面是一个演示如何应用它们的示例：

```
// Ex7_01.cpp
// Using the equal() algorithm
#include <iostream>                          // For standard streams
#include <vector>                            // For vector container
#include <algorithm>                         // For equal() algorithm
#include <iterator>                          // For stream iterators
#include <string>                            // For string class
using std::string;

int main()
{
   std::vector<string> words1 {"one", "two", "three", "four", "five", "six",
    "seven", "eight", "nine"};
   std::vector<string> words2 {"two", "three", "four", "five", "six",
    "seven", "eight", "nine", "ten"};
   auto iter1 = std::begin(words1);
   auto end_iter1 = std::end(words1);
   auto iter2 = std::begin(words2);
   auto end_iter2 = std::end(words2);

   std::cout << "Container - words1: ";
   std::copy(iter1, end_iter1, std::ostream_iterator<string>{std::cout, " "});
   std::cout << "\nContainer - words2: ";
   std::copy(iter2, end_iter2, std::ostream_iterator<string>{std::cout, " "});
   std::cout << std::endl;

   std::cout << "\n1. Compare from words1[1] to end with words2: ";
   std::cout << std::boolalpha << std::equal(iter1 + 1, end_iter1, iter2)
             << std::endl;

   std::cout << "2. Compare from words2[0] to second-to-last with words1: ";
   std::cout << std::boolalpha << std::equal(iter2, end_iter2 - 1, iter1)
             << std::endl;
```

```
    std::cout << "3. Compare from words1[1] to words1[5] with words2: ";
    std::cout << std::boolalpha << std::equal(iter1 + 1, iter1 + 6, iter2)
              << std::endl;

    std::cout << "4. Compare first 6 from words1 with first 6 in words2: ";
    std::cout << std::boolalpha << std::equal(iter1, iter1 + 6, iter2, iter2
              + 6) << std::endl;

    std::cout << "5. Compare all words1 with words2: ";
    std::cout << std::boolalpha << std::equal(iter1, end_iter1, iter2)
              << std::endl;

    std::cout << "6. Compare all of words1 with all of words2: ";
    std::cout << std::boolalpha << std::equal(iter1, end_iter1, iter2,
              end_iter2) << std::endl;

    std::cout << "7. Compare from words1[1] to end with words2 from first to
     second-to-last: ";
    std::cout << std::boolalpha
              << std::equal(iter1 + 1, end_iter1, iter2, end_iter2 - 1) << std::endl;
}
```

输出为：

```
Container - words1: one two three four five six seven eight nine
Container - words2: two three four five six seven eight nine ten
1. Compare from words1[1] to end with words2: true
2. Compare from words2[0] to second-to-last with words1: false
3. Compare from words1[1] to words1[5] with words2: true
4. Compare first 6 from words1 with first 6 in words2: false
5. Compare all words1 with words2: false
6. Compare all of words1 with all of words2: false
7. Compare from words1[1] to end with words2 from first to second-to-last: true
```

在这个示例中，对来自于 words1 和 words2 容器的元素的不同序列进行了比较。equal() 调用产生这些输出的原因如下：

- 第 1 条语句的输出为 true，因为 words1 的第二个元素到最后一个元素都从 words2 的第一个元素开始匹配。第二个序列的元素个数比第一个序列的元素个数多 1，但第一个序列的元素个数决定了比较多少个对应的元素。
- 第 2 条语句的输出为 false，因为有直接的不匹配；words2 和 words1 的第一个元素不同。
- 第 3 条语句的输出为 true，因为 word1 中从第二个元素开始的 5 个元素和 words2 的前五个元素相等。
- 在第 4 条语句中，words2 的元素序列是由开始和结束迭代器指定的。序列长度相同，但它们的第一个元素不同，所以结果为 false。
- 在第 5 条语句中，两个序列的第一个元素直接就不匹配，所以结果为 false。

- 第 6 条语句的输出为 false，因为序列是不同的。这条语句不同于前面的 equal()调用，因为指定了第二个序列的结束迭代器。
- 第 7 条语句会从 words1 的第二个元素开始，与 word2 从第一个元素开始比较相同个数的元素，所以输出为 true。

当用 equal()从开始迭代器开始比较两个序列时，第二个序列用来和第一个序列比较的元素个数由第一个序列的长度决定。就算第二个序列比第一个序列的元素多，equal()仍然会返回 true。如果为两个序列提供了开始和结束迭代器，为了使结果为 true，序列必须是相同的长度。

尽管可以用 equal()来比较两个同种类型的容器的全部内容，但最好还是使用容器的成员函数 operator==()来做这些事。示例中的第 6 条输出语句可以这样写：

```
std::cout << std::boolalpha << (words1 == words2) << " "; // false
```

这两个版本的 equal()接受一个谓词作为额外的参数。这个谓词定义了元素之间的等价比较。下面是一个说明它们用法的代码段：

```
std::vector<string> r1 {"three", "two", "ten"};
std::vector<string> r2 {"twelve", "ten", "twenty"};
std::cout << std::boolalpha
          << std::equal(std::begin(r1), std::end(r1), std::begin(r2),
          [](const string& s1, const string& s2) { return s1[0] == s2[0]; })
          << std::endl; // true
std::cout << std::boolalpha
          << std::equal(std::begin(r1), std::end(r1), std::begin(r2),
            std::end(r2),
            [](const string& s1, const string& s2) { return s1[0] == s2[0]; })
          << std::endl; // true
```

在 equal()的第一次使用中，第二个序列是由开始迭代器指定的。谓词是一个在字符串参数的第一个字符相等时返回 true 的 lambda 表达式。最后一条语句表明，equal()算法可以使用两个全范围的序列，并使用相同的谓词。

不应该用 equal()来比较来自于无序 map 或 set 容器中的元素序列。在无序容器中，一组给定元素的顺序可能和保存在另一个无序容器中的一组相等元素不同，因为不同容器的元素很可能会被分配到不同的格子中。

7.2.1　查找序列的不同之处

equal()算法可以告诉我们两个序列是否匹配。mismatch()算法也可以告诉我们两个序列是否匹配，而且如果不匹配，它还能告诉我们不匹配的位置。mismatch()的 4 个版本和 equal()一样有相同的参数——第二个序列有或没有结束迭代器，有或没有定义比较的额外的函数对象参数。mismatch()返回的 pair 对象包含两个迭代器。它的 first 成员是一个来自前两个参数所指定序列的迭代器，second 是来自于第二个序列的迭代器。当序列不匹配时，pair 包含的迭代器指向第一对不匹配的元素；因此这个 pair 对象为 pair<iter1 + n, iter2 + n>，这

两个序列中索引为 n 的元素是第一个不匹配的元素。

当序列匹配时，pair 的成员取决于使用的 mismatch()的版本和具体情况。iter1 和 end_iter1 表示定义第一个序列的迭代器，iter2 和 end_iter2 表示第二个序列的开始和结束迭代器。返回的匹配序列的 pair 的内容如下：

对于 mismatch(iter1, end_iter1, iter2)。

- 返回 pair<end_iter1, (iter2 + (end_iter1 - iter1))>，pair 的成员 second 等于 iter2 加上第一个序列的长度。如果第二个序列比第一个序列短，结果是未定义的。

对于 mismatch(iter1, end_iter1, iter2, end_iter2)。

- 当第一个序列比第二个序列长时，返回 pair<end_iter1, (iter2 + (end_iter1 - iter1))>，所以成员 second 为 iter2 加上第一个序列的长度。
- 当第二个序列比第一个序列长时，返回 pair<(iter1 + (end_iter2 - iter2)), end_iter2>，所以成员 first 等于 iter1 加上第二个序列的长度。
- 当序列的长度相等时，返回 pair<end_iter1, end_iter2>。

不管是否添加一个用于比较的函数对象作为参数，上面的情况都同样适用。

下面是一个使用带有默认相等比较的 mismatch()的示例：

```
// Ex7_02.cpp
// Using the mismatch() algorithm
#include <iostream>                    // For standard streams
#include <vector>                      // For vector container
#include <algorithm>                   // For equal() algorithm
#include <string>                      // For string class
#include <iterator>                    // For stream iterators

using std::string;
using word_iter = std::vector<string>::iterator;

int main()
{
   std::vector<string> words1 { "one", "two", "three", "four",
                                "five", "six", "seven", "eight", "nine"};
   std::vector<string> words2 { "two", "three", "four", "five",
                                "six", "eleven", "eight", "nine", "ten"};
   auto iter1 = std::begin(words1);
   auto end_iter1 = std::end(words1);
   auto iter2 = std::begin(words2);
   auto end_iter2 = std::end(words2);

  // Lambda expression to output mismatch() result
 auto print_match = [](const std::pair<word_iter, word_iter>& pr, const
  word_iter& end_iter)
                     {
                       if(pr.first != end_iter)
                        std::cout << "\nFirst pair of words that differ are "
                             << *pr.first << " and " << *pr.second << std::endl;
                       else
```

```
                    std::cout << "\nRanges are identical." << std::endl;
                  };

std::cout << "Container - words1: ";
 std::copy(iter1, end_iter1, std::ostream_iterator<string>{std::cout, " "});
std::cout << "\nContainer - words2: ";
 std::copy(iter2, end_iter2, std::ostream_iterator<string>{std::cout, " "});
std::cout << std::endl;

std::cout << "\nCompare from words1[1] to end with words2:";
print_match(std::mismatch(iter1 + 1, end_iter1, iter2), end_iter1);

std::cout << "\nCompare from words2[0] to second-to-last with words1:";
print_match(std::mismatch(iter2, end_iter2 - 1, iter1), end_iter2 - 1);

std::cout << "\nCompare from words1[1] to words1[5] with words2:";
print_match(std::mismatch(iter1 + 1, iter1 + 6, iter2), iter1 + 6);

std::cout << "\nCompare first 6 from words1 with first 6 in words2:";
print_match(std::mismatch(iter1, iter1 + 6, iter2, iter2 + 6), iter1 + 6);

std::cout << "\nCompare all words1 with words2:";
print_match(std::mismatch(iter1, end_iter1, iter2), end_iter1);

std::cout << "\nCompare all of words2 with all of words1:";
print_match(std::mismatch(iter2, end_iter2, iter1, end_iter1), end_iter2);

std::cout << "\nCompare from words1[1] to end with words2[0] to
  second-to-last:";
 print_match(std::mismatch(iter1 + 1, end_iter1, iter2, end_iter2 - 1),
  end_iter1);
}
```

注意 words2 中的内容和前面示例中的有些不同。每一次应用 mismatch()的结果都是由定义为 print_match 的 lambda 表达式生成的。它的参数是一个 pair 对象和一个 vector<string>容器的迭代器。使用 using 指令生成 word_iter 别名可以使 lambda 表达式的定义更简单。

在 main()的代码中使用了不同版本的 mismatch()，它们都没有包含比较函数对象的参数。如果第二个序列只用开始迭代器指定，为了和第一个序列匹配，它只需要有和第一个序列相等长度的元素，但也可以更长。如果第二个序列是完全指定的，会由最短的序列来确定比较多少个元素。

输出如下：

```
Container - words1: one two three four five six seven eight nine
Container - words2: two three four five six eleven eight nine ten

Compare from words1[1] to end with words2:
First pair of words that differ are seven and eleven

Compare from words2[0] to second-to-last with words1:
First pair of words that differ are two and one
```

```
Compare from words1[1] to words1[5] with words2:
Ranges are identical.

Compare first 6 from words1 with first 6 in words2:
First pair of words that differ are one and two

Compare all words1 with words2:
First pair of words that differ are one and two

Compare all of words2 with all of words1:
First pair of words that differ are two and one

Compare from words1[1] to end with words2[0] to second-to-last:
First pair of words that differ are seven and eleven
```

输出显示了每个 mismatch()的运用结果。

在我们提供自己的函数对象时，就可以完全灵活地定义相等比较。例如：

```
std::vector<string> range1 {"one", "three", "five", "ten"};
std::vector<string> range2 {"nine", "five", "eighteen", "seven"};
auto pr = std::mismatch( std::begin(range1), std::end(range1),
                         std::begin(range2), std::end(range2),
                                             [](const string& s1, const string& s2)
                                             { return s1.back() == s2.back(); });
if(pr.first == std::end(range1) || pr.second == std::end(range2))
  std::cout << "The ranges are identical." << std::endl;
else
  std::cout << *pr.first << " is not equal to " << *pr.second << std::endl;
```

当两个字符串的最后一个字符相等时，这个比较会返回 true，所以这段代码的输出为：

```
five is not equal to eighteen
```

当然，这是正确的——而且根据比较函数，“one”等于“nine”，“three”等于“five”。

7.2.2 按字典序比较序列

两个字符串的字母排序是通过从第一个字符开始比较对应字符得到的。第一对不同的对应字符决定了哪个字符串排在首位。字符串的顺序就是不同字符的顺序。如果字符串的长度相同，而且所有的字符都相等，那么这些字符串就相等。如果字符串的长度不同，短字符串的字符序列和长字符串的初始序列是相同的，那么短字符串小于长字符串。因此“age”在“beauty”之前，“a lull”在“a storm”之前。显然，“the chicken”而不是“the egg”会排在首位。

对于任何类型的对象序列来说，字典序都是字母排序思想的泛化。从两个序列的第一个元素开始依次比较对应的元素，前两个对象的不同会决定序列的顺序。显然，序列中的对象必须是可比较的。lexicographical_compare()算法可以比较由开始和结束迭代器定义的两个序列。它的前两个参数定义了第一个序列，第 3 和第 4 个参数分别是第二个序列的开

始和结束迭代器。默认用<运算符来比较元素，但在需要时，也可以提供一个实现小于比较的函数对象作为可选的第 5 个参数。如果第一个序列的字典序小于第二个，这个算法会返回 true，否则返回 false。所以，返回 false 表明第一个序列大于或等于第二个序列。序列是逐个元素比较的。第一对不同的对应元素决定了序列的顺序。如果序列的长度不同，而且短序列和长序列的初始元素序列匹配，那么短序列小于长序列。长度相同而且对应元素都相等的两个序列是相等的。空序列总是小于非空序列。下面是一个使用 lexicographical_compare()的示例：

```
std::vector<string> phrase1 {"the", "tigers", "of", "wrath"};
std::vector<string> phrase2 {"the", "horses", "of", "instruction"};
auto less = std::lexicographical_compare(std::begin(phrase1), std::end(phrase1),
                                         std::begin(phrase2), std::end(phrase2));
std::copy(std::begin(phrase1), std::end(phrase1), std::ostream_iterator<string>
  {std::cout, " "});
std::cout << (less ? "are" : "are not") << " less than ";
std::copy(std::begin(phrase2), std::end(phrase2), std::ostream_iterator
  <string>{std::cout, " "});
std::cout << std::endl;
```

因为这些序列的第二个元素不同，而且“tigers”大于“horses”，这段代码会生成如下输出：

```
the tigers of wrath are not less than the horses of instruction
```

可以在 lexicographical_compare()调用中添加一个参数，得到相反的结果：

```
auto less = std::lexicographical_compare(std::begin(phrase1), std::end(phrase1),
                                         std::begin(phrase2), std::end(phrase2),
    [](const string& s1, const string& s2){ return s1.length() < s2.length(); });
```

这个算法会使用作为第 3 个参数的 lambda 表达式来比较元素。这里会比较序列中字符串的长度，因为 phrase1 中第 4 个元素的长度小于 phrase2 中对应的元素，所以 phrase1 小于 phrase2。

7.2.3　序列的排列

排列就是一次对对象序列或值序列的重新排列。例如，“ABC”中字符可能的排列是：

```
"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"
```

三个不同的字符有 6 种排列，这个数字是从 3*2*1 得到的。一般来说，n 个不同的字符有 n！种排列，n！是 n×(n−1)×(n−2)...×2×1。很容易明白为什么要这样算。有 n 个对象时，在序列的第一个位置就有 n 种可能的选择。对于第一个对象的每一种选择，序列的第二个位置还剩下 n−1 种选择，因此前两个有 n×((n−1)种可能选择。在选择了前两个之后，第三个位置还剩下 n−2 种选择，因此前三个有 n × (n−1) × (n−2)种可能选择，以此类推。序

列的末尾是 Hobson 选择，因为只剩下一种选择。

对于包含相同元素的序列来说，如果一个序列中的元素顺序不同，就是一种排列。next_permutation()会生成一个序列的重排列，它是所有可能的字典序中的下一个排列。它默认使用<运算符来做这些事情。它的参数为定义序列的迭代器和一个返回布尔值的函数，这个函数在下一个排列大于上一个排列时返回 true，如果上一个排列是序列中最大的，它返回 false，所以会生成字典序最小的排列。

下面展示了如何生成一个包含 4 个整数的 vector 的排列：

```
std::vector<int> range {1,2,3,4};
do
{
 std::copy(std::begin(range), std::end(range), std::ostream_iterator<int>
   {std::cout, " "});
 std::cout << std::endl;
} while(std::next_permutation(std::begin(range), std::end(range)));
```

当 next_permutation()返回 false 时，循环结束，表明到达最小排列。这样恰好可以生成序列的全部排列，这只是因为序列的初始排列为 1、2、3、4，这是排列集合中的第一个排列。有一种方法可以得到序列的全排列，就是使用 next_permutation()得到的最小排列：

```
std::vector<string> words {"one","two", "three", "four", "five", "six",
 "seven", "eight"};
while(std::next_permutation(std::begin(words), std::end(words)))
//Change to minimum
;
do
{
 std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
   {std::cout, " "});
 std::cout << std::endl;
} while(std::next_permutation(std::begin(words), std::end(words)));
```

words 中的初始序列不是最小的排列序列，循环会继续进行，直到 words 包含最小排列。do-while 循环会输出全部的排列。如果想执行这段代码，需要记住它会生成 8!种排列，从而输出 40320 行，因此首先可能会减少 words 中元素的个数。

当排列中的每个元素都小于或等于它后面的元素时，它就是元素序列的最小排列，所以可以用 min_element()来返回一个指向序列中最小元素的迭代器，然后用 iter_swap()算法交换两个迭代器指向的元素，从而生成最小的排列，例如：

```
std::vector<string> words {"one","two", "three", "four", "five", "six",
 "seven", "eight"};
for (auto iter = std::begin(words); iter != std::end(words)-1 ;++iter)
 std::iter_swap(iter, std::min_element(iter, std::end(words)));
```

for 循环从序列的第一个迭代器开始遍历，直到倒数第二个迭代器。for 循环体中的语句会交换 iter 指向的元素和 min_element()返回的迭代器所指向的元素。这样最终会生成一

个最小排列，然后可以用它作为 next_permutation()的起始点来生成全排列。

在开始生成全排列之前，可以先生成一个原始容器的副本，然后在循环中改变它，从而避免到达最小排列的全部开销。

```
std::vector<string> words {"one","two", "three", "four", "five", "six",
  "seven", "eight"};
auto words_copy = words; // Copy the original
do
{
  std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
    {std::cout, " "});
  std::cout << std::endl;
  std::next_permutation(std::begin(words), std::end(words));
} while(words != words_copy); // Continue until back to the original
```

循环现在会继续生成新的排列，直到到达原始排列。

下面是一个找出单词中字母的全部排列的示例：

```
// Ex7_03.cpp
// Finding rearrangements of the letters in a word
#include <iostream>                    // For standard streams
#include <iterator>                    // For iterators and begin() and end()
#include <string>                      // For string class
#include <vector>                      // For vector container
#include <algorithm>                   // For next_permutation()
using std::string;

int main()
{
  std::vector<string> words;
  string word;
  while(true)
  {
    std::cout << "\nEnter a word, or Ctrl+z to end: ";
    if((std::cin >> word).eof()) break;
    string word_copy {word};
    do
    {
      words.push_back(word);
      std::next_permutation(std::begin(word), std::end(word));
    } while(word != word_copy);
    size_t count{}, max{8};
    for(const auto& wrd : words)
      std::cout << wrd << ((++count % max == 0) ? '\n' : ' ');
    std::cout << std::endl;
    words.clear();                     // Remove previous permutations
  }
}
```

这段代码会从标准输入流读取一个单词到 word 中，然后在 word_copy 中生成一个副本，将 word 中字符的全排列保存到 words 容器中。这个程序会继续处理单词直到按下 Ctrl+Z 组合键。用 word 的副本来判断是否已经保存了全排列。然后所有的排列会被写入输出流，8 个一行。像之前说的那样，随着被排列元素个数的增加，排列的个数增加也很快，所以这里不要尝试使用太长的单词。这个示例并不是很有用，但在第 9 章中会重新审视这个程序，那时候会介绍更多关于 STL 中文件的细节。那里它可能会读取一个包含很多英语单词的文件，并通过搜索这些单词来确定哪些排列是无效的单词。因此，这个程序可以找到字谜的源单词，并输出它们。

可以为 next_permutation()提供一个函数对象作为第三个参数，从而用这个函数对象定义的比较函数来代替默认的比较函数。下面展示如何使用这个版本的函数，通过比较最后一个字母的方式来生成 words 序列的排列：

```
std::vector<string> words {"one", "two", "four", "eight"};
do
{
 std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
  {std::cout, " "});
 std::cout << std::endl;
} while(std::next_permutation(std::begin(words), std::end(words),
  [](const string& s1, const string& s2){return s1.back() < s2.back(); }));
```

通过传入一个 lambda 表达式作为 next_permutation()的最后一个参数，这段代码会生成 words 中元素的全部 24 种排列。

next_permutation()是按照字典降序的方式生成的排列。当我们想以降序的方式生成排列时，可以使用 prev_permutation()。它和 next_permutation()一样有两个版本，默认使用<来比较元素。因为排列是以降序的方式生成的，所以算法大多数时候会返回 true。当生成最大排列时，返回 false。例如：

```
std::vector<double> data {44.5, 22.0, 15.6, 1.5};
do
{
 std::copy(std::begin(data), std::end(data), std::ostream_iterator<double>
  {std::cout, " "});
 std::cout << std::endl;
} while(std::prev_permutation(std::begin(data), std::end(data)));
```

这段代码会输出 data 中 4 个 double 值的全部 24 种排列，因为初始序列是最大排列，所以 prev_permutation()会在输入最小排列时，才返回 false。

可以用 is_permutation()算法来检查一个序列是不是另一个序列的排列，如果是，会返回 true。下面是在这个算法中使用 lambda 表达式的示例：

```
std::vector<double> data1 {44.5, 22.0, 15.6, 1.5};
std::vector<double> data2 {22.5, 44.5, 1.5, 15.6};
std::vector<double> data3 {1.5, 44.5, 15.6, 22.0};
```

```
auto test = [](const auto& d1, const auto& d2)
  {
    std::copy(std::begin(d1), std::end(d1), std::ostream_iterator<double>
      {std::cout, " "});
    std::cout << (is_permutation(std::begin(d1), std::end(d1), std::begin(d2),
                  std::end(d2)) ?"is": "is not")
              << " a permutation of ";
   std::copy(std::begin(d2), std::end(d2), std::ostream_iterator<double>
     {std::cout, " "});
   std::cout << std::endl;
  };

  test(data1, data2);
  test(data1, data3);
  test(data3, data2);
```

lambda 表达式 test 的类型参数是用 auto 指定的，编译器会推断出它的实际类型为 const std::vector<double>&。使用 auto 来指定类型参数的 lambda 表达式叫作泛型 lambda。lambda 表达式 test 用 is_permutation()来评估参数是否是另一种排列。算法的参数是一对用来定义被比较范围的迭代器。返回的布尔值会用来选择输出两个字符串中的哪一个。输出如下：

```
44.5 22 15.6 1.5 is not a permutation of 22.5 44.5 1.5 15.6
44.5 22 15.6 1.5 is a permutation of 1.5 44.5 15.6 22
1.5 44.5 15.6 22 is not a permutation of 22.5 44.5 1.5 15.6
```

另一个版本的 is_permutation()允许只用开始迭代器指定第二个序列。在这种情况下，第二个序列可以包含比第一个序列还要多的元素，但是只会被认为拥有第一个序列中的元素个数。然而，并不推荐使用它，因为如果第二个序列包含的元素少于第一个序列，会产生未定义的错误。接下来会展示一些使用这个函数的代码。我们可以在 data3 中添加一些元素，但它的初始序列仍然会是 data1 的一个排列。例如：

```
std::vector<double> data1 {44.5, 22.0, 15.6, 1.5};
std::vector<double> data3 {1.5, 44.5, 15.6, 22.0, 88.0, 999.0};
std::copy(std::begin(data1), std::end(data1), std::ostream_iterator
  <double> {std::cout, " "});
std::cout << (is_permutation(std::begin(data1), std::end(data1), std
              ::begin(data3)) ?"is": "is not")
          << " a permutation of ";
std::copy(std::begin(data3), std::end(data3), std::ostream_iterator
  <double> {std::cout, " "});
std::cout << std::endl;
```

这里会确认 data1 是 data3 的一个排列，因为只考虑 data3 的前 4 个元素。每一个版本的 is_permutation()都可以添加一个额外的参数来指定所使用的比较。

可以用 shuffle()算法来生成序列的随机排列，但会在第 8 章详细地讨论 STL 所提供的随机数生成能力时，再讨论这些。

7.3 复制序列

这一节会讨论用来复制序列的算法；但不要忘了，在需要将一个容器中的全部内容移到另一个容器中时，我们还有其他的选择。容器定义的赋值运算符，可以将一个容器的全部内容复制到另一个相同类型的容器中。容器的构造函数也接受序列作为初始内容的来源。大多数时候，本节的算法可以用来复制容器元素的子集。

先前已经介绍过很多 copy()算法的运用，所以知道它们是如何工作的。它会将前两个输入迭代器定义的源序列复制到第三个参数指定的目的序列的开始位置，第三个参数必须是一个输出迭代器。有 3 个更加高级的算法，它们不仅可以提供简单的复制过程，还能提供其他功能。

7.3.1 复制一定数目的元素

copy_n()算法可以从源容器复制指定个数的元素到目的容器中。第一个参数是指向第一个源元素的输入迭代器，第二个参数是需要复制的元素的个数，第三个参数是指向目的容器的第一个位置的迭代器。这个算法会返回一个指向最后一个被复制元素的后一个位置的迭代器，或者只是第三个参数——输出迭代器——如果第二个参数为 0。下面是一个使用它的示例：

```
std::vector<string> names {"Al", "Beth", "Carol", "Dan", "Eve",
                           "Fred", "George", "Harry", "Iain", "Joe"};
std::unordered_set<string> more_names {"Janet", "John"};
std::copy_n(std::begin(names) + 1, 3, std::inserter(more_names, std::
  begin(more_names)));
```

这个 copy_n()操作会从 names 的第二个元素开始复制 3 个元素到关联容器 more_names 中。目的容器是由一个 unordered_set 容器的 insert_iterator 对象指定的，它是由 inserter()函数模板生成的。insert_iterator 对象会调用容器的成员函数 insert()来向容器中添加元素。

当然，copy_n()的目的地址也可是以流迭代器：

```
std::copy_n(std::begin(more_names), more_names.size()-1,
                        std::ostream_iterator<string> {std::cout, " "});
```

这样会输出 more_names 中除了最后一个元素之外的全部元素。注意，如果被复制元素的个数超过了实际元素的个数，程序会因此崩溃。如果元素的个数为 0 或负数，copy_n()算法什么也不做。

7.3.2 条件复制

copy_if()算法可以从源序列复制使谓词返回 true 的元素，所以可以把它看作一个过滤器。前两个参数定义源序列的输入迭代器，第三个参数是指向目的序列的第一个位置的输

出迭代器，第 4 个参数是一个谓词。会返回一个输出迭代器，它指向最后一个被复制元素的下一个位置。下面是一个使用 copy_if()的示例：

```
std::vector<string> names {"Al", "Beth", "Carol", "Dan", "Eve",
                           "Fred", "George", "Harry", "Iain", "Joe"};
std::unordered_set<string> more_names {"Jean", "John"};
size_t max_length{4};
std::copy_if(std::begin(names), std::end(names), std::inserter(more_names, std::
 begin(more_names)),[max_length](const string& s){ return s.length() <= max_length; });
```

因为作为第 4 个参数的 lambda 表达式所添加的条件，这里的 copy_if()操作只会复制 names 中的 4 个字符串或更少。目的容器是一个 unordered_set 容器 more_names，它已经包含两个含有 4 个字符的名称。和前面的章节一样，insert_iterator 会将元素添加到限定的关联容器中。如果想要展示它是如何工作的，可以用 copy()算法列出 more_names 的内容：

```
std::copy(std::begin(more_names), std::end(more_names), std:: ostream iterator
          <string>
{std::cout, " "});
std::cout << std::endl;
```

当然，copy_if()的目的容器也可以是一个流迭代器：

```
std::vector<string> names {"Al", "Beth", "Carol", "Dan", "Eve",
                           "Fred", "George", "Harry", "Iain", "Joe"};
size_t max_length{4};
std::copy_if(std::begin(names), std::end(names), std::ostream iterator< string>
  {std::cout, " "},[max_length](const string& s) { return  s.length() >
                     max_length; });
std::cout << std::endl;
```

这里会将 names 容器中包含的含有 4 个以上字符的名称写到标准输出流中。这段代码会输出如下内容：

```
Carol George Harry
```

输入流迭代器可以作为 copy_if()算法的源，也可以将它用在其他需要输入迭代器的算法上。例如：

```
std::unordered_set<string> names;
size_t max_length {4};
std::cout << "Enter names of less than 5 letters. Enter Ctrl+Z on a separate
  line to end:\n";
std::copy_if(std::istream_iterator<string>{std::cin},
             std::istream iterator<string>{},std::inserter(names,std::begin(names)),
  [max_length](const string& s) { return s.length() <= max_length; });
std::copy(std::begin(names), std::end(names),
          std::ostream_iterator <string>
  {std::cout, " "});
std::cout << std::endl;
```

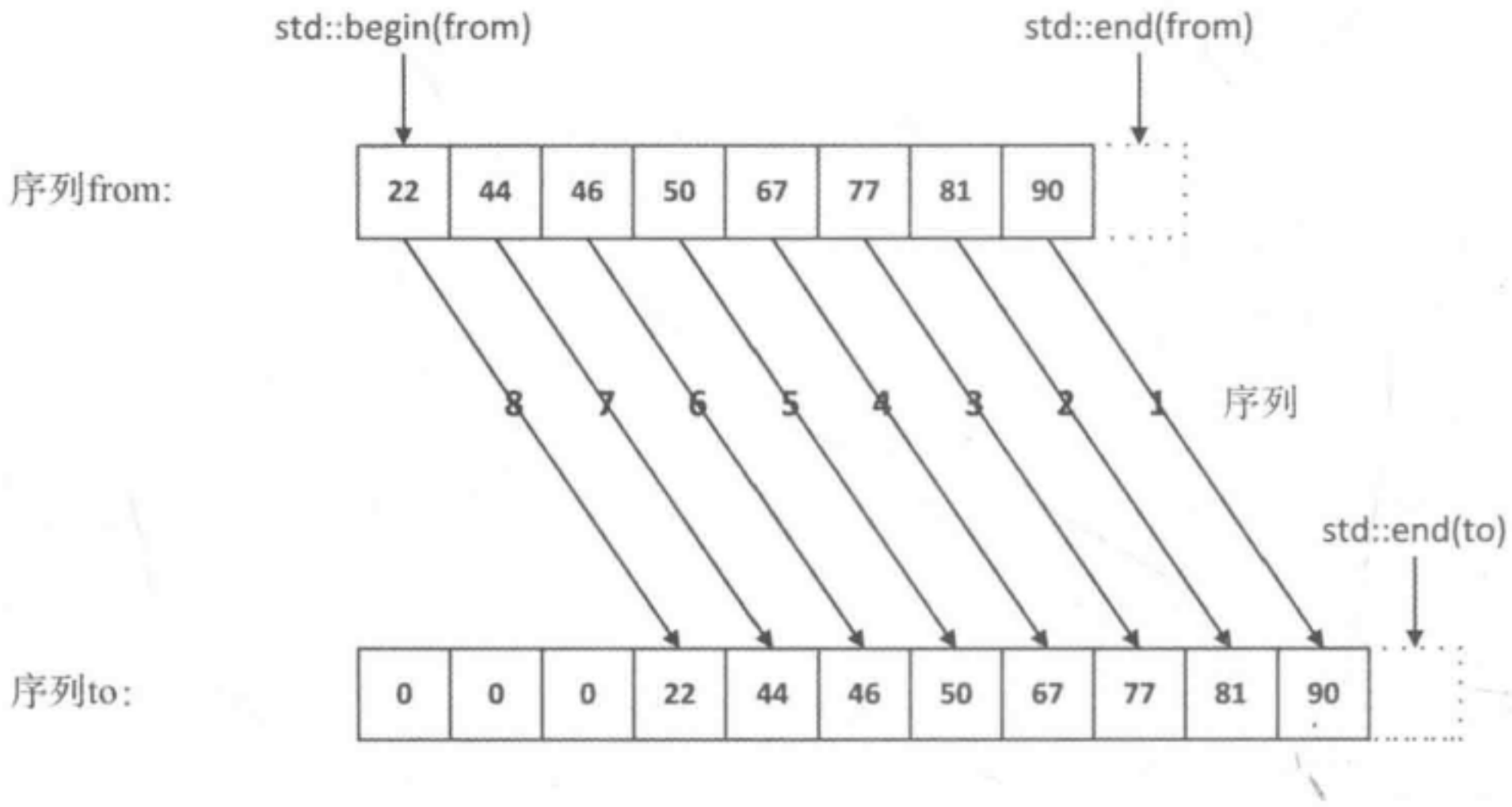
容器 names 最初是一个空的 unordered_set。只有当从标准输入流读取的姓名的长度小于或等于 4 个字符时，copy_if()算法才会复制它们。执行这段代码可能会产生如下输出：

```
Enter names of less than 5 letters. Enter Ctrl+Z on a separate line to end:
Jim Bethany Jean Al Algernon Bill Adwina Ella Frederick Don
^Z
Ella Jim Jean Al Bill Don
```

超过 5 个字母的姓名可以从 cin 读入，但是被忽略掉，因为在这种情况下第 4 个参数的判定会返回 false。因此，输入的 10 个姓名里面只有 6 个会被存储在容器中。

反向复制

不要被 copy_backward()算法的名称所误导，它不会逆转元素的顺序。它只会像 copy()那样复制元素，但是从最后一个元素开始直到第一个元素。copy_backward()会复制前两个迭代器参数指定的序列。第三个参数是目的序列的结束迭代器，通过将源序列中的最后一个元素复制到目的序列的结束迭代器之前，源序列会被复制到目的序列中，如图 7-1 所示。copy_backward()的 3 个参数都必须是可以自增或自减的双向迭代器，这意味着这个算法只能应用到序列容器的序列上。



copy_backward(std::begin(from), std::end(from), std:end(to));

图 7-1　copy_backward()的工作方式

图 7-1 说明了源序列 from 的最后一个元素是如何先被复制到目的序列 to 的最后一个元素的。从源序列的反向，将每一个元素依次复制到目的序列的前一个元素之前的位置。在进行这个操作之前，目的序列中的元素必须存在，因此目的序列至少要有和源序列一样多的元素，但也可以有更多。copy_backward()算法会返回一个指向最后一个被复制元素的迭代器，在目的序列的新位置，它是一个开始迭代器。

我们可能会好奇，相对于普通的从第一个元素开始复制的 copy()算法，copy_backward()提供了哪些优势。一个回答是，在序列重叠时，可以用 copy()将元素复制到重叠的目的序

列剩下的位置——也就是目的序列第一个元素之前的位置。如果想尝试用 copy()算法将元素复制到同一个序列的右边，这个操作不会成功，因为被复制的元素在复制之前会被重写。如果想将它们复制到右边，可以使用 copy_backward()，只要目的序列的结束迭代器在源序列的结束迭代器的右边。图 7-2 说明了在将元素复制到重叠的序列的右边时，这两个算法的不同。

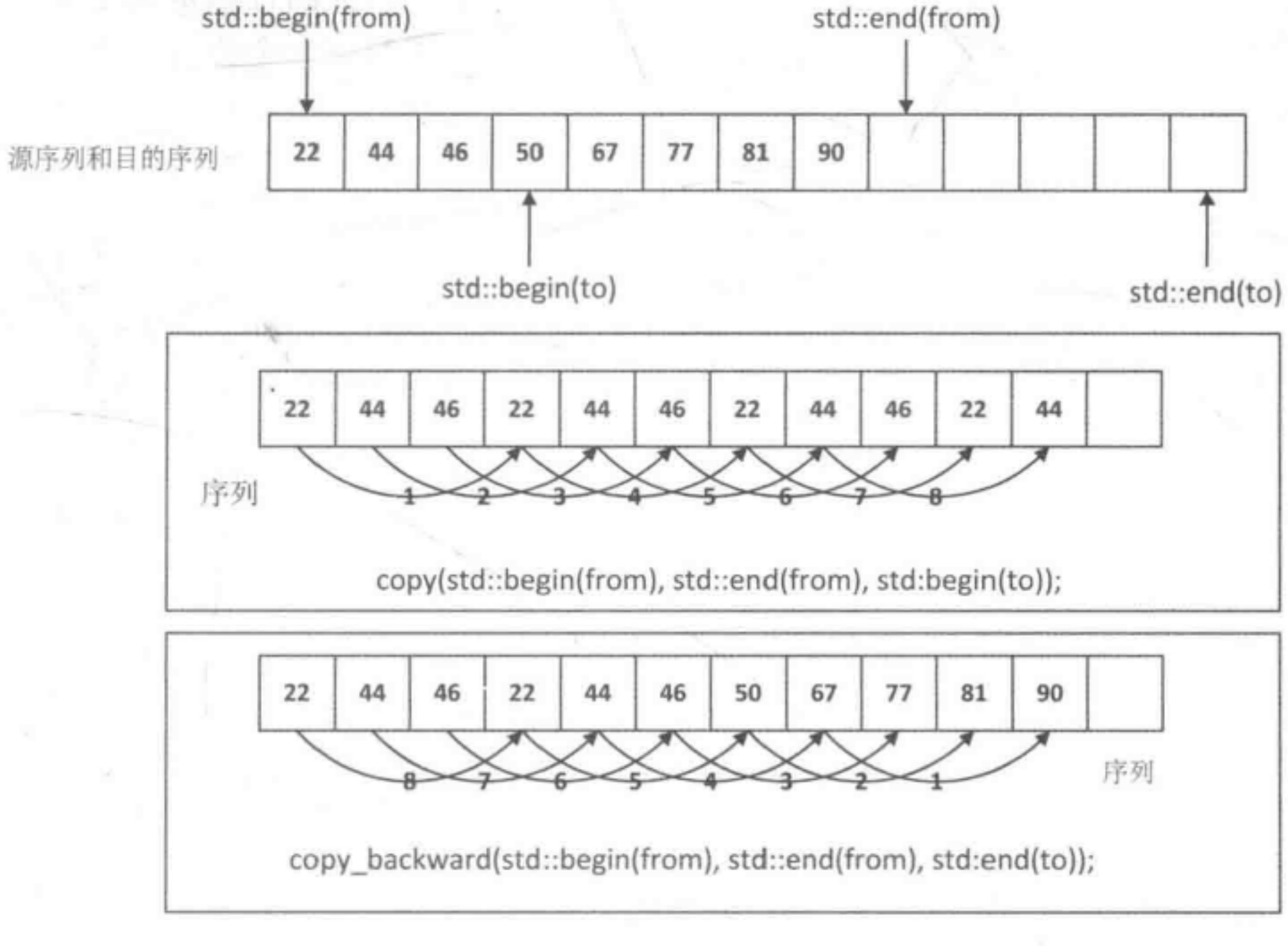


图 7-2　从右复制重叠序列

图 7-2 展示了在序列右边的前三个位置运用 copy()和 copy_backward()算法的结果。在想将元素复制到右边时，copy()算法显然不能如我们所愿，因为一些元素在复制之前会被重写。在这种情况下，copy_backward()可以做到我们想做的事。相反在需要将元素复制到序列的左边时，copy()可以做到，但 copy_backward()做不到。

下面是一个说明 copy_backward()用法的示例：

```
std::deque<string> song{"jingle", "bells", "jingle", "all", "the", "way"};
song.resize(song.size()+2); // Add 2 elements
std::copy_backward(std::begin(song), std::begin(song)+6, std::end(song));
std::copy(std::begin(song), std::end(song), std::ostream iterator <string>
  {std::cout, " "});
std::cout << std::endl;
```

为了能够在右边进行序列的反向复制操作，需要添加一些额外的元素，可以通过使用 deque 的成员函数 resize()来增加 deque 容器的元素个数。copy_backward()算法会将原有的元素复制到向右的两个位置，保持前两个元素不变，所以这段代码的输出如下：

```
jingle bells jingle bells jingle all the way
```

7.4 复制和反向元素顺序

reverse_copy()算法可以将源序列复制到目的序列中，目的序列中的元素是逆序的。定义源序列的前两个迭代器参数必须是双向迭代器。目的序列由第三个参数指定，它是目的序列的开始迭代器，也是一个输出迭代器。如果序列是重叠的，函数的行为是未定义的。这个算法会返回一个输出迭代器，它指向目的序列最后一个元素的下一个位置。下面是一个使用 reverse_copy()和 copy_if()的示例：

```
// Ex7_04.cpp
// Testing for palindromes using reverse_copy()
#include <iostream>       // For standard streams
#include <iterator>       // For stream iterators and begin() and end()
#include <algorithm>      // For reverse_copy() and copy_if()
#include <cctype>       // For toupper() and isalpha()
#include <string>
using std::string;

int main()
{
  while(true)
  {
    string sentence;
    std::cout << "Enter a sentence or Ctrl+Z to end: ";
    std::getline(std::cin, sentence);
    if(std::cin.eof()) break;

    // Copy as long as the characters are alphabetic & convert to upper case
    string only_letters;
    std::copy_if(std::begin(sentence), std::end(sentence), std::back inserter
      (only_letters),[](char ch) { return std::isalpha(ch); });
    std::for_each(std::begin(only_letters), std::end(only_letters), [](char& ch)
    { ch = toupper(ch); });
    // Make a reversed copy
    string reversed;
    std::reverse_copy(std::begin(only_letters), std::end(only_letters),
    std::back_inserter(reversed));
    std::cout << '"' << sentence << '"'
     << (only_letters == reversed ? " is" : " is not") << " a palindrome."
        << std::endl;
  }
}
```

这个程序会检查一条语句(也可以是很多条语句)是否是回文的。回文语句是指正着读或反着读都相同的句子，前提是忽略一些像空格或标点这样的细节。while 使我们可以检查尽可能多的句子。用 getline()读一条句子到 sentence 中。如果读到 Ctrl+Z，输入流中会设置 1 个 EOF 标志，它会结束循环。用 copy_if()将 sentence 中的字母复制到 only_letters。

这个 lambda 表达式只在参数是字母时返回 true，所以其他的任何字符都会被忽略。然后用 back_inserter()生成的 back_insert_iterator 对象将这些字符追加到 only_letter。for_each()算法会将第三个参数指定的函数对象应用到前两个参数定义的序列的元素上，因此这里会将 only_letters 中的字符全部转换为大写。然后用 reverse_copy()算法生成和 only_letters 的内容相反的内容。比较 only_letters 和 reversed 来判断输入的语句是否为回文。

下面是示例输出：

```
Enter a sentence or Ctrl+Z to end: Lid off a daffodil.
"Lid off a daffodil." is a palindrome.
Enter a sentence or Ctrl+Z to end: Engage le jeu que je le gagne.
"Engage le jeu que je le gagne." is a palindrome.
Enter a sentence or Ctrl+Z to end: Sit on a potato pan Otis!
"Sit on a potato pan Otis!" is a palindrome.
Enter a sentence or Ctrl+Z to end: Madam, I am Adam.
"Madam, I am Adam." is not a palindrome.
Enter a sentence or Ctrl+Z to end: Madam, I'm Adam.
"Madam, I'm Adam." is a palindrome.
Enter a sentence or Ctrl+Z to end: ^Z
```

回文很难生成，但法国人乔治·佩雷克设法构造了一个包含 1000 多个单词的回文。

reverse()算法可以在原地逆序它的两个双向迭代器参数所指定序列的元素。可以如下所示用它来代替 Ex7_04.cpp 中的 reverse_copy()：

```
string reversed {only_letters};
std::reverse(std::begin(reversed), std::end(reversed));
```

这两条语句会替换 Ex7_04.cpp 中 reversed 的定义和 reverse_copy()调用。它们生成一个 only_letters 的副本 reversed，然后调用 reverse()原地逆序 reversed 中的字符序列。

7.5　复制一个删除相邻重复元素的序列

unique_copy()会将一个序列复制另一个序列中，同时会移除连续的重复元素。默认使用==运算符来决定元素何时相等。它的前两个参数是指定源序列的迭代器，第 3 个参数是指向目的序列第一个元素的迭代器，可选的第 4 个参数可以接受一个函数对象，它可以替代==运算符。这个算法会返回一个输出迭代器，它指向目的序列最后一个元素的下一个位置。

复制一个像 1、1、2、2、3 这样的序列，会导致目的序列包含 1、2、3。因为只会移除连续的重复元素，像 1、2、1、2、3 这种序列，所有的元素都会被复制。当然，如果源序列已经被排序，那么所有的重复元素都会被移除，因此目的序列就只包含唯一的元素。

下面是一个将 unique_copy()应用到字符串中字符上的示例：

```
string text {"Have you seen how green the trees seem?"};
string result{};
```

```
std::unique_copy(std::begin(text), std::end(text), std::back_inserter(result));
std::cout << result << std::endl;
```

复制操作的源序列是整个字符串 text，目的序列是 result 的一个 back_insert_iterator，因此每个字符会被追加到 result。这种方式输出的句子几乎没什么用：

```
Have you sen how gren the trees sem?
```

由这个输出可以确定 unique_copy()移除了相邻的重复元素。

当我们提供了比较函数对象时，就不再局限于简单的相等比较——可以用自己喜欢的任何方式来定义比较函数。这也使我们有了可以选择不被复制的重复元素的可能。下面是一个展示如何从一个字符串中移除重复空格的示例：

```
string text {"There's no air in spaaaaaace!"};
string result {};
std::unique_copy(std::begin(text), std::end(text), std::back_inserter(result),
     [](char ch1, char ch2) { return ch1 == ' '  && ch1 == ch2; });
std::cout << result << std::endl;
```

unique_copy()的第 4 个参数是一个 lambda 表达式，只会在两个参数都是空格时，它才会返回 true。执行这段代码会输出如下内容：

```
There's no air in spaaaaaace!
```

这个输出说明空格已经被移除，但是没有移除 spaaaaaace 中的'a'。

7.6 从序列中移除相邻的重复元素

unique()算法可以在序列中原地移除重复的元素，这就要求被处理的序列必须是正向迭代器所指定的。在移除重复元素后，它会返回一个正向迭代器作为新序列的结束迭代器。可以提供一个函数对象作为可选的第三个参数，这个参数会定义一个用来代替==比较元素的方法。例如：

```
std::vector<string> words {"one", "two", "two", "three", "two", "two", "two"};
auto end_iter = std::unique(std::begin(words), std::end(words));
std::copy(std::begin(words), end_iter, std::ostream_iterator<string>{std::cout, " "});
std::cout << std::endl;
```

这样会通过覆盖来消除 words 中的连续元素。输出为：

```
one two three two
```

当然，没有元素会从输入序列中移除；算法并没有方法去移除元素，因为它并不知道它们的具体上下文。整个序列仍然存在。但是，无法保证新末尾之后的元素的状态；如果在上面的代码中用 std::end(words)代替 end_iter 来输出结果，在笔者系统上得到的输出如下：

```
one two three two two two
```

相同个数的元素仍然存在，但新的结束迭代器指向的元素为空字符串；最后两个元素还和之前一样。在你的系统上，可能会有不同的结果。因为这个，在执行 unique()后，最好按如下方式截断序列：

```
auto end_iter = std::unique(std::begin(words), std::end(words));
words.erase(end_iter, std::end(words));
std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
  {std::cout, " "});
std::cout << std::endl;
```

容器的成员函数 erase()会移除新的结束迭代器之后的所有元素，因此 end(words)会返回 end_iter。

当然，可以将 unique()运用到字符串中的字符上：

```
string text {"There's no air in spaaaaaace!"};
text.erase(std::unique(std::begin(text), std::end(text),
          [](char ch1, char ch2) { return ch1 == ' ' && ch1 == ch2; }),std::end(text));
std::cout << text << std::endl; // Outputs: There's no air in spaaaaaace!
```

这里使用 unique()会移除字符串 text 中的连续重复的空格。这段代码会用 unique()返回的迭代器作为 text 成员函数 erase()的第一个参数，而且它会指向被移除的第一个字符。erase()的第二个参数是 text 的结束迭代器，因此在没有重复元素的新字符串之后的所有字符都会被移除。

7.7　旋转序列

rotate()算法会从左边选择序列的元素。它的工作机制如图 7-3 所示。为了理解如何旋转序列，可以将序列中的元素想象成手镯上的珠子。rotate()操作会导致一个新元素成为开始迭代器所指向的第一个元素。在旋转之后，最后一个元素会在新的第一个元素之前。

rotate()的第一个参数是这个序列的开始迭代器；第二个参数是指向新的第一个元素的迭代器，它必定在序列之内。第三个参数是这个序列的结束迭代器。图 7-3 中的示例说明在容器 ns 上的旋转操作使值为 4 的元素成为新的第一个元素，最后一个元素的值为 3。元素的圆形序列会被维持，因此可以有效地旋转元素环，直到新的第一个元素成为序列的开始。这个算法会返回一个迭代器，它指向原始的第一个元素所在的新位置。例如：

```
std::vector<string> words {"one", "two", "three", "four", "five", "six",
  "seven", "eight"};
auto iter = std::rotate(std::begin(words), std::begin(words)+3, std::end(words));
std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
  {std::cout, " "});
std::cout << std::endl << "First element before rotation: " << *iter << std::endl;
```

这段代码对 words 中的所有元素进行了旋转。执行这段代码会生成如下内容：

```
four five six seven eight one two three
First element before rotation: one
```

输出说明"four"成为新的第一个元素，而且 rotate()返回的迭代器指向之前的第一个元素"one"。

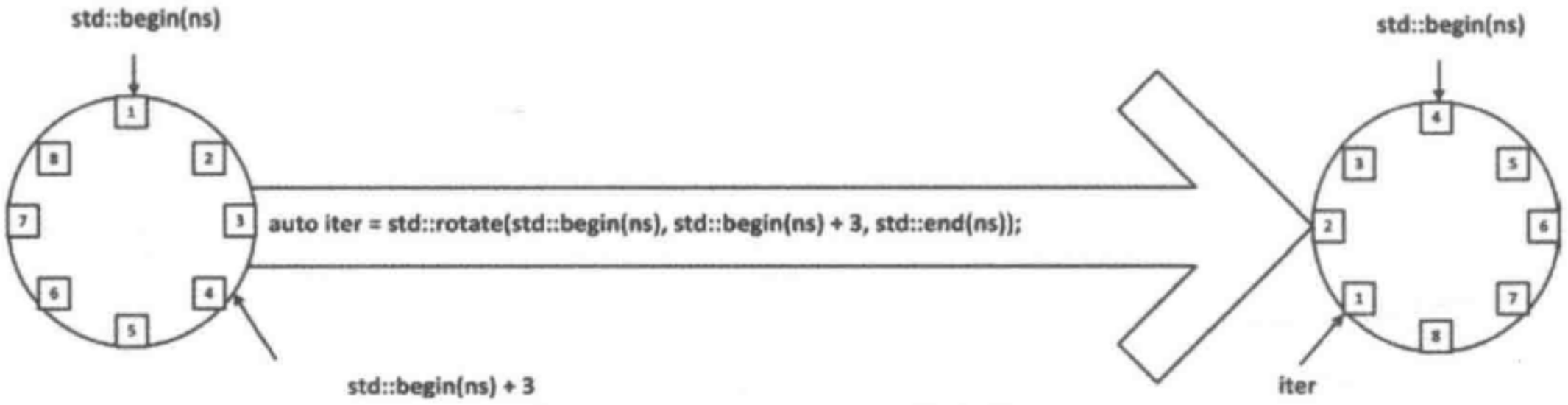


图 7-3　rotate()算法的工作方式

当然，不需要对容器中的全部元素进行旋转。例如：

```
std::vector<string> words {"one", "two", "three", "four", "five",
                           "six", "seven", "eight", "nine", "ten"};
auto start = std::find(std::begin(words), std::end(words), "two");
auto end_iter = std::find(std::begin(words), std::end(words), "eight");
auto iter = std::rotate(start, std::find(std::begin(words), std::end (words),
  "five"), end_iter);
std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
  {std::cout, " "});
std::cout << std::endl << "First element before rotation: " << *iter
          << std::endl;
```

这里用 find()算法分别获取了和"two"、"eight"匹配的元素的迭代器。它们定义了被旋转的序列，这个序列是容器元素的子集。这个序列会被旋转为使"five"成为第一个元素，输出说明它是按预期工作的：

```
one five six seven two three four eight nine ten
First element before rotation: two
```

rotate_copy()算法会在新序列中生成一个序列的旋转副本，并保持原序列不变。rotate_copy()的前 3 个参数和 copy()是相同的；第 4 个参数是一个输出迭代器，它指向目的序列的第一个元素。这个算法会返回一个目的序列的输出迭代器，它指向最后一个被复制元素的下一个位置。例如：

```
std::vector<string> words {"one", "two", "three", "four", "five",
                           "six", "seven", "eight", "nine", "ten"};
auto start = std::find(std::begin(words), std::end(words), "two");
```

```
auto end_iter = std::find(std::begin(words), std::end(words), "eight");
std::vector<string> words_copy;
std::rotate_copy(start, std::find(std::begin(words), std::end(words),
  "five"), end_iter,std::back_inserter(words_copy));
std::copy(std::begin(words_copy), std::end(words_copy),
                                  std::ostream_iterator<string>{std::cout, " "});
std::cout << std::endl;
```

这段代码会对 word 中从"two"到"seven"的元素生成一个旋转副本。通过使用 back_insert_iterator 将复制的元素追加到 words_copy 容器中，back_insert_iterator 会调用 words_copy 容器的成员函数 push_back()来插入每个元素。这段代码产生的输出如下：

```
five six seven two three four
```

这里 rotate_copy()返回的迭代器是 words_copy 中元素的结束迭代器。在这段代码中，并没有保存和使用它，但它却很有用。例如：

```
std::vector<string> words {"one", "two", "three", "four", "five",
                           "six", "seven", "eight", "nine", "ten"};
auto start = std::find(std::begin(words), std::end(words), "two");
auto end_iter = std::find(std::begin(words), std::end(words), "eight");
std::vector<string> words_copy {20}; // vector with 20 default elements
auto end_copy_iter = std::rotate_copy(start,std::find(std::begin(words),
  std::end(words), "five"), end_iter, std::begin(words_copy));
std::copy (std::begin (words_copy), end_copy_iter, std::ostream_iterator<string>
  {std::cout, " "});
  std::cout << std::endl;
```

生成的 words_copy 容器默认有 20 个元素。rotate_copy()算法现在会将现有元素的旋转序列保存到 words_copy 中。在输出时，这个算法返回的迭代器可以用来确定 words_copy 的尾部边界；如果没有它，就必须通过源序列的元素个数来计算出尾部边界。

7.8 移动序列

move()算法会将它的前两个输入迭代器参数指定的序列移到第三个参数定义的目的序列的开始位置，第三个参数必须是输出迭代器。这个算法返回的迭代器指向最后一个被移动到目的序列的元素的下一个位置。这是一个移动操作，因此无法保证在进行这个操作之后，输入序列仍然保持不变；源元素仍然会存在，但它们的值可能不再相同了，因此在移动之后，就不应该再使用它们。如果源序列可以被替换或破坏，就可以选择使用 move()算法。如果不想扰乱源序列，可以使用 copy()算法。下面是一个展示如何使用它的示例：

```
std::vector<int> srce {1, 2, 3, 4};
std::deque<int> dest {5, 6, 7, 8};
std::move(std::begin(srce), std::end(srce), std::back_inserter(dest));
```

这里会将 data 的最后 6 个元素移到容器的开头。它能够正常工作是因为目的地址在源序列之外。在移动之后，无法保证最后两个元素的值。这里它们虽然被移除了，但同样可以将它们重置为已知的值——例如 0。最后一行中的注释展示了输出结果。当然也可以用 rotate()算法来代替 move()移动元素，在这种情况下，我们肯定知道最后两个元素的值。

如果一个移动操作的目的地址位于源序列之内，move()就无法正常工作，这意味着移动需要从序列的右边开始。原因是一些元素在移动之前会被重写，但 move_backward()算法可以正常工作。它的前两个参数指定了被移动的序列，第三个参数是目的地址的结束迭代器。例如：

```
std::vector<int> data {1, 2, 3, 4, 5, 6, 7, 8};
std::move(std::begin(data) + 2, std::end(data), std::begin(data));
data.erase(std::end(data) - 2, std::end(data));  // Erase moved elements
std::copy(std::begin(data), std::end(data), std::ostream_iterator<int> {std::cout, " "});
std::cout << std::endl;                                  // 3, 4, 5, 6, 7, 8
```

这里使用 deque 容器只是为了换个容器使用。将前 6 个元素向右移动两个位置。在移动操作后，值无法得到保证的元素会被重置为 0。最后一行展示了这个操作的结果。

可以用 swap_ranges()算法来交换两个序列。这个算法需要 3 个正向迭代器作为参数。前两个参数分别是第一个序列的开始和结束迭代器，第三个参数是第二个序列的开始迭代器。显然，这两个序列的长度必须相同。这个算法会返回一个迭代器，它指向第二个序列的最后一个被交换元素的下一个位置。例如：

```
using Name = std::pair<string, string>;    // First and second name
std::vector<Name> people {Name{"Al", "Bedo"}, Name{"Ann", "Ounce"},
  Name {"Jo", "King"}};
std::list<Name> folks {Name{"Stan", "Down"}, Name{"Dan", "Druff"},
  Name {"Bea", "Gone"}};
std::swap_ranges(std::begin(people), std::begin(people) + 2, ++std::begin(folks));
std::for_each(std::begin(people), std::end(people),[](const Name& name)
                    {std::cout << '"' << name.first << " " << name.second << "\" ";});
std::cout << std::endl;                  // "Dan Druff" "Bea Gone" "Jo King"
std::for_each(std::begin(folks), std::end(folks),[](const Name& name)
                    {std::cout << '"' << name.first << " " << name.second << "\" "; });
std::cout << std::endl;                  // "Stan Down" "Al Bedo" "Ann Ounce"
```

这里使用 vector 和 list 容器来保存 pair<string,string>类型的元素，pair<string,string>用来表示名称。swap_ranges()算法被用来交换 people 的前两个元素和 folks 的后两个元素。这里并没有为了将 pair 对象写入流而重载 operator<<()函数，因此 copy()无法用输出流迭代器来列出容器的内容。为了生成输出，选择使用 for_each()算法将 lambda 表达式运用到容器的每个元素上。这个 lambda 表达式只会将传给它的 Name 元素的成员变量写入标准输出流。注释展示了执行这段代码后输出的结果：

定义在 utility 头文件中的 swap()算法的重载函数的模板原型为：

```
template<typename T1, typename T2> void swap(std::pair<T1,T2> left,
  std::pair<T1,T2> right);
```

这段代码会对 pair<T1,T2>对象进行交换，在前面的代码段中也可以用 swap_ranges()来交换元素。

用来交换两个 T 类型对象的 swap()模板也被定义在 utility 头文件中。除了 pair 对象的重载之外，utility 文件头中也有可以交换任何类型的容器对象的模板的重载。也就是说，可以交换两个 list<T>容器或者两个 set<T>容器但不能是一个 list<T>和 vector<T>，也不能是一个 list<T1>和一个 list<T2>。另一个 swap()模板的重载可以交换两个相同类型的数组。也有其他几个 swap()的重载，它们可以用来交换其他类型的对象，包含元组和智能指针类型，正如本章前面所述。iter_swap()算法有一些不同，它会交换两个正向迭代器所指向的元素。

7.9　从序列中移除元素

如果不知道具体的场景——元素保存在什么样的容器中，是不能从序列中移除元素的。因此，“移除元素的”算法也无法做到这一点，它们只会重写被选择的元素或者忽略复制的元素。移除操作不会改变被“移除”元素的序列的元素个数。有 4 种移除算法：

- remove()可以从它的前两个正向迭代器参数指定的序列中移除和第三个参数相等的对象。基本上每个元素都是通过用它后面的元素覆盖它来实现移除的。它会返回一个指向新的最后一个元素之后的位置的迭代器。
- remove_copy()可以将前两个正向迭代器参数指定的序列中的元素复制到第三个参数指定的目的序列中，并忽略和第 4 个参数相等的元素。它返回一个指向最后一个被复制到目的序列的元素的后一个位置的迭代器。序列不能是重叠的。
- remove_if()可以从前两个正向迭代器指定的序列中移除能够使作为第三个参数的谓词返回 true 的元素。
- remove_copy_if()可以将前两个正向迭代器参数指定的序列中，能够使作为第 4 个参数的谓词返回 true 的元素，复制到第三个参数指定的目的序列中。它返回一个指向最后一个被复制到目的序列的元素的后一个位置的迭代器。序列不能是重叠的。

可以按如下方式使用 remove()：

```
std::deque<double> samples {1.5, 2.6, 0.0, 3.1, 0.0, 0.0, 4.1, 0.0, 6.7, 0.0};
samples.erase(std::remove(std::begin(samples), std::end(samples), 0.0),
  std::end(samples));
std::copy(std::begin(samples), std::end(samples),
  std::ostream_iterator <double> {std::cout, " "});
  std::cout << std::endl; // 1.5 2.6 3.1 4.1 6.7
```

sample 中不应包含为 0 的物理测量值。remove()算法会通过左移其他元素来覆盖它们，通过这种方式就可以消除杂乱分布的 0。remove()返回的迭代器指向通过这个操作得到的新序列的尾部，所以可以用它作为被删除序列的开始迭代器来调用 samples 的成员函数 erase()。注释说明容器中的元素没有被改变。

如果想保留原始序列，并生成一个移除选定元素之后的副本，可以使用 remove_copy()。例如：

```
std::deque<double> samples {1.5, 2.6, 0.0, 3.1, 0.0, 0.0, 4.1, 0.0, 6.7, 0.0};
std::vector<double> edited_samples;
std::remove_copy(std::begin(samples), std::end(samples),
  std::back_ inserter (edited_samples), 0.0);
```

samples 容器中的非零元素会被复制到 edited_samples 容器中，edited_samples 正好是一个不同的容器——它是一个 vector 容器。通过 back_insert_iterator 对象将这些元素添加到 edited_samples，因此这个容器只包含从 sample 中复制的元素。

remove_if()提供了更强大的能力，它能够从序列中移除和给定值匹配的元素。谓词会决定一个元素是否被移除；它接受序列中的一个元素为参数，并返回一个布尔值。例如：

```
using Name = std::pair<string, string>; // First and second name
std::set<Name> blacklist {Name {"Al", "Bedo"}, Name {"Ann", "Ounce"}, Name
  {"Jo", "King"}};
std::deque<Name> candidates {Name {"Stan", "Down"}, Name {"Al", "Bedo"}, Name {"Dan",
  "Druff"},Name {"Di", "Gress"}, Name {"Ann", "Ounce"}, Name {"Bea", "Gone"}};
candidates.erase(std::remove_if(std::begin(candidates), std::end(candidates),
                                [&blacklist](const Name& name) { return
                                  blacklist.count(name); }),std::end(candidates));
std::for_each(std::begin(candidates), std::end(candidates), [](const Name& name)
                              {std::cout << '"' << name.first << " " << name. second
                                << "\" "; });
std::cout << std::endl;          // "Stan Down" "Dan Druff" "Di Gress" "Bea Gone"
```

这段代码用来模拟候选人申请成为俱乐部会员。那些众所周知的不安分人士的姓名被保存在 blacklist 中，它是一个集合。当前申请成为会员的候选人被保存在 candidates 容器中，它是一个 deque 容器。用 remove_if()算法来保证不会有 blacklist 中的姓名通过甄选过程。这里的谓词是一个以引用的方式捕获 blacklist 容器的 lambda 表达式。当参数在容器中存在时，set 容器的成员函数 count()会返回 1。谓词返回的值会被隐式转换为布尔值，因此对于每一个出现在 blacklist 中的候选人，谓词都会返回 true，然后会将它们从 candidates 中移除。注释中显示了通过甄选的候选人。

remove_copy_if()之于 remove_copy()，就像 remove_if()之于 remove()。下面展示它是如何工作的：

```
std::set<Name> blacklist {Name {"Al", "Bedo"}, Name {"Ann", "Ounce"}, Name
  {"Jo", "King"}};
std::deque<Name> candidates {Name {"Stan", "Down"}, Name {"Al", "Bedo"},
  Name {"Dan", "Druff"},Name {"Di", "Gress"}, Name {"Ann", "Ounce"},
                               Name {"Bea", "Gone"}};
std::deque<Name> validated;
std::remove_copy_if(std::begin(candidates), std::end(candidates), std:: back
  inserter(validated),[&blacklist](const Name& name) { return blacklist.
                          count(name); });
```

这段代码实现了和前一段代码同样的功能，除了结果被保存在 validated 容器中和没有修改 candidates 容器之外。

7.10　设置和修改序列中的元素

fill()和 fill_n()算法提供了一种为元素序列填入给定值的简单方式，fill()会填充整个序列；fill_n()则以给定的迭代器为起始位置，为指定个数的元素设置值。下面展示了 fill()的用法：

```
std::vector<string> data {12}; // Container has 12 elements
std::fill(std::begin(data), std::end(data), "none"); // Set all elements to "none"
```

fill()的前两个参数是定义序列的正向迭代器，第三个参数是赋给每个元素的值。当然这个序列并不一定要代表容器的全部元素。例如：

```
std::deque<int> values(13);                    // Container has 13 elements
int n{2};                                      // Initial element value
const int step {7};                            // Element value increment
const size_t count{3};                         // Number of elements with given value
auto iter = std::begin(values);
while(true)
{
  auto to_end = std::distance(iter, std::end(values)); // Number of elements
                                                       // remaining
  if(to_end < count)          // In case no. of elements not a multiple of count
  {
  std::fill(iter, iter + to_end, n);     // Just fill remaining elements...
  break;                                 // ...and end the loop
  }
  else
  {
    std::fill(iter, std::end(values), n);// Fill next count elements
  }
    iter = std::next(iter, count);    // Increment iter
    n += step;
}
```

上面创建了具有 13 个元素的 value 容器。在这种情况下，必须用圆括号将值传给构造函数；使用花括号会生成一个有单个元素的容器，单个元素的值为 13。在循环中，fill()算法会将 values 赋值给 count 个元素。以 iter 作为容器的开始迭代器，如果还有足够的元素剩下，每次遍历中，它会被加上 count，因此它会指向下个序列的第一个元素。执行这段代码会将 values 中的元素设置为：

```
2 2 2 9 9 9 16 16 16 23 23 23 30
```

fill_n()的参数分别是指向被修改序列的第一个元素的正向迭代器、被修改元素的个数以及要被设置的值。distance()和 next()函数定义在 iterator 头文件中。前者必须使用输入迭代器，而后者需要使用正向迭代器。

7.10.1 用函数生成元素的值

你已经知道可以用 for_each()算法将一个函数对象应用到序列中的每一个元素上。函数对象的参数是 for_each()的前两个参数所指定序列中元素的引用，因此它可以直接修改被保存的值。generate()算法和它有些不同，它的前两个参数是指定范围的正向迭代器，第三个参数是用来定义下面这种形式的函数的函数对象：

```
T fun(); // T is a type that can be assigned to an element in the range
```

无法在函数内访问序列元素的值。generate()算法只会保存函数为序列中每个元素所返回的值，而且 generate()没有任何返回值。为了使这个算法更有用，可以将生成的不同的值赋给无参数函数中的不同元素。也可以用一个可以捕获一个或多个外部变量的函数对象作为 generate()的第三个参数。例如：

```
string chars (30, ' ');                  // 30 space characters
char ch {'a'};
int incr {};
std::generate(std::begin(chars), std::end(chars), [ch, &incr]
                                                  {
                                                    incr += 3;
                                                    return ch + (incr % 26);
                                                  });
std::cout << chars << std::endl;    // chars is: dgjmpsvybehknqtwzcfiloruxadgjm
```

变量 chars 被初始化为一个有 30 个空格的字符串。作为 generate()的第三个参数的 lambda 表达式的返回值会被保存到 chars 的连续字符中。lambda 表达式以值的方式捕获 ch，以引用的方式捕获 incr，因此会在 lambda 的主体中对后者进行修改。lambda 表达式会返回 ch 加上 incr 后得到的字符，增加的值是 26 的模，因此返回的值总是在'a'到'z'之间，给定的起始值为' a'。这个操作的结果会在注释中展示出来。可以对 lambda 表达式做一些修改，使它可以用于任何大写或小写字母，但只生成保存在 ch 中的这种类型的字母。我们把它作为练习留给你们。

generate_n()和 generate()的工作方式是相似的。不同之处是，它的第一个参数仍然是序列的开始迭代器，第二个参数是由第三个参数设置的元素的个数。为了避免程序崩溃，这个序列必须至少有第二个参数定义的元素个数。例如：

```
string chars (30, ' ');                  // 30 space characters
char ch {'a'};
int incr {};
std::generate_n(std::begin(chars), chars.size()/2,[ch, &incr]
                                                  {
                                                    incr += 3;
                                                    return ch + (incr % 26);
                                                  });
```

这里，chars 中只有一半的元素会被算法设为新的值，剩下的一半仍然为空格。

7.10.2　转换序列

transform()可以将函数应用到序列的元素上，并将这个函数返回的值保存到另一个序列中，它返回的迭代器指向输出序列所保存的最后一个元素的下一个位置。这个算法有一个版本和 for_each()相似，可以将一个一元函数应用到元素序列上来改变它们的值，但这里有很大的区别。for_each()中使用的函数的返回类型必须为 void，而且可以通过这个函数的引用参数来修改输入序列中的值；而 transform()的二元函数必须返回一个值，并且也能够将应用函数后得到的结果保存到另一个序列中。输出序列中的元素类型可以和输入序列中的元素类型不同。这里也有一些区别。对于 for_each()，函数总是会被应用序列的元素上，但对于 transform()，这一点无法保证。

第二个版本的 transform()允许将二元函数应用到两个序列相应的元素上，但先来看一下如何将一元函数应用到序列上。在这个算法的这个版本中，它的前两个参数是定义输入序列的输入迭代器，第 3 个参数是目的位置的第一个元素的输出迭代器，第 4 个参数是一个二元函数。这个函数必须接受来自输入序列的一个元素为参数，并且必须返回一个可以保存在输出序列中的值。例如：

```
std::vector<double> deg_C {21.0, 30.5, 0.0, 3.2, 100.0};
std::vector<double> deg_F(deg_C.size());
std::transform(std::begin(deg_C), std::end(deg_C), std::begin(deg_F),
               [](double temp){ return 32.0 + 9.0* temp/5.0; });
               // Result 69.8 86.9 32 37.76 212
```

这个 transform()算法会将 deg_C 容器中的摄氏温度转换为华氏温度，并将这个结果保存到 deg_F 容器中。为了保存全部结果，生成的 deg_F 需要一定个数的元素。因此第三个参数是 deg_F 的开始迭代器。通过用 back_insert_iterator 作为 transform()的第三个参数，可以将结果保存到空的容器中：

```
std::vector<double> deg_F; // Empty container
std::transform(std::begin(deg_C), std::end(deg_C), std::back_inserter(deg_F),
               [](double temp){ return 32.0 + 9.0* temp/5.0; });
               // Result 69.8 86.9 32 37.76 212
```

用 back_insert_iterator 在 deg_F 中生成保存了操作结果的元素；结果是相同的。

第三个参数可以是指向输入容器的元素的迭代器。例如：

```
std::vector<double> temps {21.0, 30.5, 0.0, 3.2, 100.0}; // In Centigrade
std::transform(std::begin(temps), std::end(temps), std::begin(temps),
               [](double temp)
                { return 32.0 + 9.0* temp / 5.0; }); // Result 69.8 86.9 32 37.76 212
```

这里将 temp 容器中的值从摄氏温度转换成了华氏温度。第三个参数是输入序列的开始迭代器，应用第 4 个参数指定的函数的结果会被存回它所运用的元素上。

下面的代码展示了目的序列和输入序列是不同类型的情况：

```
std::vector<string> words {"one", "two", "three", "four", "five"};
std::vector<size_t> hash_values;
std::transform(std::begin(words), std::end(words), std::back_inserter(hash_values),
                                  std::hash<string>()); // string hashing function
std::copy(std::begin(hash_values), std::end(hash_values),
                                  std::ostream_iterator<size_t> {std::cout, " "});
std::cout << std::endl;
```

输入序列包含 string 对象，并且应用到元素的函数是一个定义在 string 头文件中的标准的哈希函数对象。这个哈希函数会返回 size_t 类型的哈希值，并且会用定义在 iterator 头文件中的辅助函数 back_inserter()返回的 back_insert_iterator 将这些值保存到 hash_values 容器中。在笔者的系统上，这段代码产生的输出如下：

```
3123124719 3190065193 2290484163 795473317 2931049365
```

你的系统可能会产生不同的输出。注意，因为目的序列是由 back_insert_iterator 对象指定的，这里 transform()算法会返回一个 back_insert_iterator<vector<size_T>>类型的迭代器，因此不能在 copy()算法中用它作为输入序列的结束迭代器。为了充分利用 transform()返回的迭代器，这段代码可以这样写：

```
std::vector<string> words {"one", "two", "three", "four", "five"};
std::vector<size_t> hash_values(words.size());
auto end_iter = std::transform(std::begin(words),
                  std::end(words), std::begin(hash_values),
                    std::hash<string>()); // string hashing function
std::copy(std::begin(hash_values), end_iter, std::ostream_iterator<size_t>
  {std::cout, " "});
std::cout << std::endl;
```

现在，transform()返回的是 hash_values 容器中元素序列的结束迭代器。

可以在 transform()所运用的函数中为元素序列调用一个算法。下面举例说明：

```
std::deque<string> names {"Stan Laurel", "Oliver Hardy", "Harold Lloyd"};
std::transform(std::begin(names), std::end(names), std::begin(names),
    [](string& s) { std::transform(std::begin(s), std::end(s),
                    std::begin(s), ::toupper);
                    return s;
                  });
std::copy(std::begin(names), std::end(names), std::ostream_iterator<string>
  {std::cout, " "});
std::cout << std::endl;
```

transform()算法会将 lambda 定义的函数应用到 names 容器中的元素上。这个 lambda 表达式会调用 transform()，将定义在 cctype 头文件中的 toupper()函数应用到传给它的字符串的每个字符上。它会将 names 中的每个元素都转换为大写，因此输出为：

```
STAN LAUREL OLIVER HARDY HAROLD LLOYD
```

当然，也有其他更简单的方式可以得到相同的结果。

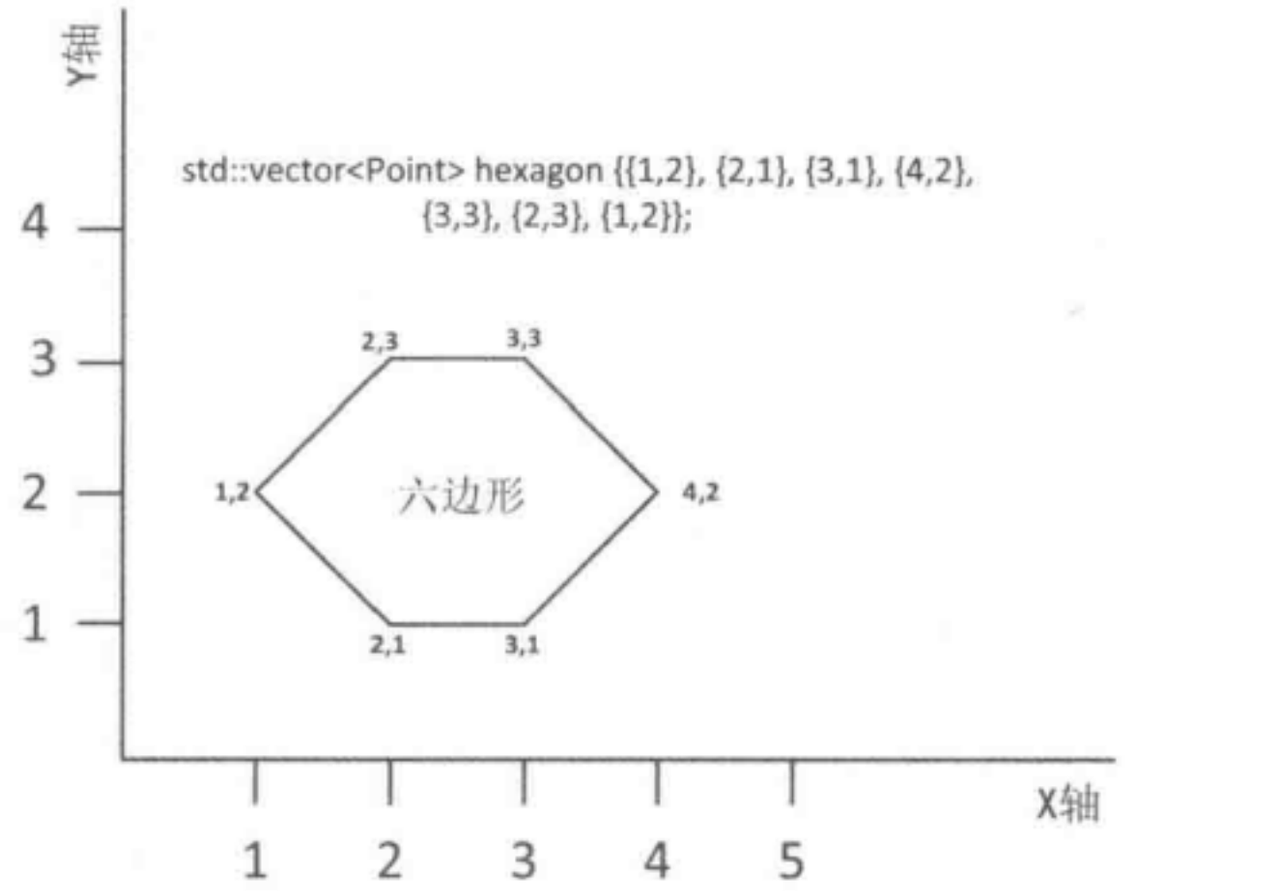
应用二元函数的这个版本的 transform()期望 5 个参数：

- 前两个参数是第一个输入序列的输入迭代器。
- 第 3 个参数是第二个输入序列的开始迭代器，显然，这个序列必须至少包含和第一个输入序列同样多的元素。
- 第 4 个参数是一个序列的输出迭代器，它所指向的是用来保存应用函数后得到的结果的序列的开始迭代器。
- 第 5 个参数是一个函数对象，它定义了一个接受两个参数的函数，这个函数接受来自两个输入序列中的元素作为参数，返回一个可以保存在输出序列中的值。

让我们来思考一个关于几何计算的简单示例。一条折线是由点之间连续的线组成的。折线可以表示为一个 Point 对象的 vector，折线线段是加入连续点的线。如果最后一个点和前一个点相同，折线就是闭合的——一个多边形。Point 被定义为一个类型别名，图 7-4 展示了一个示例：

```
using Point = std::pair<double, double>; // pair<x,y> defines a point
```

这里有 7 个点，因此图 7-4 中的六边形对象有 6 个折线段。因为第一个点和最后一个点是相同的，这 6 条线段实际上组成了一个多边形——六边形。可以用 transform()算法来计算这些线段的长度：



在点(x_1，y_1)和(x_2，y_2)之间的折线线段长度=$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$

图 7-4　一条表示六边形的折线

```
std::vector<Point> hexagon {{1,2}, {2,1}, {3,1}, {4,2}, {3,3}, {2,3}, {1,2}};
std::vector<double> segments; // Stores lengths of segments
std::transform( std::begin(hexagon), std::end(hexagon) - 1, std::begin(hexagon)+ 1,
                std::back_inserter(segments),
                [](const Point& p1, const Point& p2)
                {return st d::sqrt(
               (p1.first-p2.first)*(p1.first - p2.first) +
               (p1.second - p2.second)*(p1.second - p2.second)); });
```

transform()的第一个输入序列包含六边形中从第一个到倒数第二个 Point 对象。第二个输入序列是从第二个 Point 对象开始的，因此这个二元函数调用的连续参数为点 1 和 2、点 2 和 3、点 3 和 4，依此类推，直到输入序列的最后两个点 6 和 7。图 7-4 展示了计算(x_1,y_1)和(x_2,y_2)两点之前距离的公式，作为 transform()最后一个参数的 lambda 表达式实现的就是这个公式。线段的长度是由 lambda 表达式计算的，它们会被保存在 segments 容器中。我们可以用两种以上的算法来输出线段的长度和这个六边形的周长。例如：

```
std::cout << "Segment lengths: ";
std::copy(std::begin(segments), std::end(segments),
          std::ostream_iterator<double> {std::cout, " "});
std::cout << std::endl;
std::cout << "Hexagon perimeter: "
          << std::accumulate(std::begin(segments), std::end(segments),
            0.0) << std::endl;
```

这里使用 copy()算法来输出线段的长度。accumulate()函数可以求出 segments 中元素值之和，从而得到周长。

7.10.3 替换序列中的元素

replace()算法会用新的值来替换和给定值相匹配的元素。它的前两个参数是被处理序列的正向迭代器，第 3 个参数是被替换的值，第 4 个参数是新的值。下面展示了它的用法：

```
std::deque<int> data {10, -5, 12, -6, 10, 8, -7, 10, 11};
std::replace(std::begin(data), std::end(data), 10, 99);
// Result: 99 -5 12 -6 99 8 -7 99 11
```

这里，data 容器中和 10 匹配的全部元素都会被 99 替代。

replace_if()会将使谓词返回 true 的元素替换为新的值。它的第 3 个参数是一个谓词，第 4 个参数是新的值。参数的类型一般是元素类型的 const 引用；const 不是强制性的，但谓词不应该改变元素。下面是一个使用 replace_if()的示例：

```
string password {"This is a good choice!"};
std::replace_if(std::begin(password), std::end(password),
                [](char ch){return std::isspace(ch);}, '_');
                // Result:This_is_a_good_choice!
```

这个谓词会为任何是空格字符的元素返回 true，因此这里的空格都会被下划线代替。

replace_copy()算法和 replace()做的事是一样的，但它的结果会被保存到另一个序列中，而不会改变原始序列。它的前两个参数是输入序列的正向迭代器，第 3 个参数是输入序列的开始迭代器，最后两个参数分别是要被替换的值和替换值。例如：

```
std::vector<string> words {"one", "none", "two", "three", "none", "four"};
std::vector<string> new_words;
std::replace_copy(std::begin(words), std::end(words), std::back_inserter(new_words),
             string{"none"}, string{"0"}); // Result: "one", "0", "two",
```

```
                                                    // "three", "0", "four"
```

在执行这段代码后，new_words 会包含注释中的 string 元素。

可以在序列中有选择地替换元素的最后一个算法是 replace_copy_if()，它和 replace_if()算法是相同的，但它的结果会被保存到另一个序列中。它的前两个参数是输入序列的迭代器，第 3 个参数是输出序列的开始迭代器，最后两个参数分别是谓词和替换值。例如：

```
std::deque<int> data {10, -5, 12, -6, 10, 8, -7, 10, 11};
std::vector<int> data_copy;
std::replace_copy_if(std::begin(data), std::end(data),
                     std::back_inserter(data_copy),
                     [](int value) {return value == 10;}, 99);
                     // Result:99 -5 12 -6 99 8 -7 99 11
```

data_copy 是一个 vector 容器，这里使用它只是为了说明输出容器可以和输入容器不同。这段代码执行后，它会包含注释中所示的元素。

7.11　算法的应用

在本章的最后一个示例中，会运用一些算法来将曲线绘制到标准输出流中，这更贴近实际。用一个 pair<double,double>对象的序列来定义曲线，pair<double,double>代表点(x,y)。首先我们可以定义一个用来在标准输出流中绘制曲线的 plot()函数模板。模板的类型参数是定义序列的迭代器的类型，因此点可以来自于任何序列容器或数组。每个点都会被绘制为一个星号，x 轴是跨页的，y 轴在页的下面。因为输出是字符流，所以字符的长宽比会影响绘图的长宽比。理想字体的宽度和高度是相同的，在笔者系统上选择 8×8 的字体。

plot()函数的参数分别是定义曲线上点的序列的迭代器、一个指定输出曲线的名称的字符串，以及一定数量的字符串的宽度。最后两个参数有默认值，所以可以省略它们。点中 x 值的序列需要用一定数目的字符来满足指定的绘图宽度。这会决定一个字符和另一个字符之间的 x 的跨度。为了维持绘图的长宽比，行之间的跨度和 x 值之间的跨度是相同的。下面是 plot()函数模板的代码：

```
template<typename Iterator>
void plot(Iterator begin_iter, Iterator end_iter, string name = "Curve",
  size_t n_x = 100)
{ // n_x is plot width in characters, so it's the number of characters along
  // the x axis
  // Comparison functions for x and for y
  auto x_comp = [](const Point& p1, const Point& p2) {return p1.first < p2.first; };
  auto y_comp = [](const Point& p1, const Point& p2) {return p1.second < p2.second; };

  // Minimum and maximum x values
```

```
    auto min_x = std::min_element(begin_iter, end_iter, x_comp)->first;
    auto max_x = std::max_element(begin_iter, end_iter, x_comp)->first;

    // Step length for output - same step applies to x and y
    double step {(max_x - min_x) / (n_x + 1)};

    // Minimum and maximum y values
    auto min_y = std::min_element(begin_iter, end_iter, y_comp)->second;
    auto max_y = std::max_element(begin_iter, end_iter, y_comp)->second;
    size_t nrows {1 + static_cast<size_t>(1 + (max_y - min_y)/step)};
    std::vector<string> rows(nrows, string(n_x + 1, ' '));

    // Create x-axis at y=0 if this is within range of points
    if(max_y > 0.0 && min_y <= 0.0)
      rows[static_cast<size_t>(max_y/step)] = string(n_x + 1, '-');

    // Create y-axis at x=0 if this is within range of points
    if(max_x > 0.0 && min_x <= 0.0)
    {
      size_t x_axis {static_cast<size_t>(-min_x/step)};
      std::for_each(std::begin(rows), std::end(rows),
        [x_axis](string& row) { row[x_axis] = row[x_axis] == '-' ? '+' : '|'; });
    }
    std::cout << "\n\n " << name << ":\n\n";
    // Generate the rows for output
    auto y {max_y}; // Upper y for current output row
    for(auto& row : rows)
    {
      // Find points to be included in an output row
      std::vector<Point> row_pts; // Stores points for this row
      std::copy_if(begin_iter, end_iter, std::back_inserter(row_pts),
        [&y, &step](const Point& p) { return p.second < y + step && p.second >= y; });

      std::for_each(std::begin(row_pts), std::end(row_pts), // Set * for pts in the row
        [&row, min_x, step] (const Point& p)
              {row[static_cast<size_t>((p.first - min_x) / step)] = '*'; });
      y -= step;
    }
    // Output the plot - which is all the rows.
    std::copy(std::begin(rows), std::end(rows), std::ostream_iterator<string>
       {std::cout, "\n"});
    std::cout << std::endl;
    }
```

为了比较 x 和 y 的值，定义了两个 lambda 表达式 x_comp 和 y_comp。在这些表达式

中使用 max_element()和 min_element()算法来找出 x 值和 y 值的最大和最小边界。x 的边界可以确定输出行中字符之间的水平跨度，以及一个输出行和另一个输出行之间的垂直跨度。输出行的数目由 y 值序列和跨度长度决定。输出的每一行都是一个 string 对象，因此会在 rows 容器中生成完整的绘图，它是一个 string 对象的 vector 容器。

为了生成 rows 中的绘图，需要找出每一行点中的 y 值。这些点的 y 值在当前 y 值和 y+step 之间。copy_if()算法会将每一行中满足这些条件的输入序列中的点复制到 row_pts 容器中。row_pts 中点的 x 值会被用到传给 for_each()的函数中。对于每一个点，这个函数会确定在当前行中和这个点的 x 值对应的字符序列，并将它设置为星号。

这个示例中包含两个可以为指定类型的曲线生成点的函数。一个会生成正弦曲线上的点，和正弦相比相对简单；另一个会生成心形线上的点，虽有一点复杂，但却很有趣。正弦曲线是有趣的，因为它们会出现在很多场景中。例如，声波信号就可以被看作不同频率和振幅的正弦波的混合。这个示例中的函数只会计算由等式 $y = \sin(x)$定义的曲线，但我们可以很容易地将它扩展为可以使用不同的频率和振幅。下面是这个函数的代码：

```
// Generate x,y points on curve y = sin(x) for x values 0 to 4p
std::vector<Point> sine_curve(size_t n_pts = 100)
{ // n_pts is number of data points for the curve
  std::vector<double> x_values(n_pts);
  double value {};
  double step {4 * pi / (n_pts - 1)};
  std::generate(std::begin(x_values), std::end(x_values),
    [&value, &step]() { double v {value};
                        value += step;
                        return v; });
  std::vector<Point> curve_pts;
  std::transform(std::begin(x_values), std::end(x_values), std::back inserter
    (curve_pts),[](double x) { return Point {x, sin(x)}; });
  return curve_pts;
}
```

返回的点被作为 vector 容器的 Point 元素，Point 是类型 pair<double,double>的一个别名。这段代码用 generate()算法生成从 0 到 4π 之间的 x 值。然后 transform()算法在 curve_pts 容器中生成了 point 对象，curve_pts 是这个函数的返回值。

心形笛卡尔方程是基于半径 r 为$(x^2 + y^2 - r^2)^2 = 4r^2((x - r^2)^2 + y^2)$的圆，对于确定点是否在曲线上，它不是十分有用。在一种更有效的参数表现形式中，x 和 y 都是相对于独立参数 t 定义的：

$$x = r(\cos(t) - \cos(2t))\ y = r(\sin(t) - \sin(2t))$$

通过将 t 从 0 变为 2π，我们可以获取到心形线上对应于另一个具有相同半径的圆来说滚动圆半径为 r 的点。为了用这些等式生成点，我们定义了一个函数：

```
std::vector<Point> cardioid_curve(double r = 1.0, size_t n_pts = 100)
{ // n_pts is number of data points
  double step = 2 * pi / (n_pts - 1);    // Step length for x and y
```

```
    double t_value {};                          // Curve parameter

    // Create parameter values that define the curve
    std::vector<double> t_values(n_pts);
    std::generate(std::begin(t_values), std::end(t_values),
                                  [&t_value, step]() { auto value = t_value;
                                  t_value += step;
                                  return value; });
    // Function to define an x,y point on the cardioid for a given t
    auto cardioid = [r](double t)
      { return Point {r*(2*cos(t) + cos(2*t)), r*(2*sin(t) + sin(2*t))}; };
    // Create the points for the cardioid
    std::vector<Point> curve_pts;
    std::transform(std::begin(t_values), std::end(t_values), std::back_inserter
      (curve_pts),cardioid);
    return curve_pts;
}
```

这和正弦曲线的逻辑在本质上是相同的。generate()算法生成了一个自变量的值序列，在这个示例中，它是等式的参数 t。lambda 表达式 cardioid 是单独定义的，只是因为很容易看到这种方式。它可以在参数等式中为给定的 t 生成一个 Point 对象。transform()算法会将 cardioid 应用到 t_values 上，它是用来生成曲线上 Point 对象的 vector 容器的输入序列。

现在用来绘制正弦曲线和心形线的 main()程序非常简单：

```
// Ex7_05.cpp
// Apply some algorithms to plotting curves
// To get the output with the correct aspect ratio, set the characters
// in the standard output stream destination to a square font, such as 8
// x 8 pixels
#include <iostream>                 // For standard streams
#include <iterator>                 // For iterators and begin() and end()
#include <string>                   // For string class
#include <vector>                   // For vector container
#include <algorithm>                // For algorithms
#include <cmath>                    // For sin(), cos()
using std::string;
using Point = std::pair<double, double>;

static const double pi {3.1415926};

// Definition of plot() function template here...

// Definition of sine_curve() function here...

// Definition of cardioid_curve() function here...

int main()
{
  auto curve1 = sine_curve(50);
  plot(std::begin(curve1), std::end(curve1), "Sine curve", 90);
```

```
    auto curve2 = cardioid_curve(1.5, 120);
    plot(std::begin(curve2), std::end(curve2), "Cardioid", 60);
}
```

为了使这个程序中的所有代码都能访问静态变量 pi，它被定义在全局作用域内。两个用来生成曲线的函数都使用了它。曲线点的个数的定义、x 值的范围以及绘图的宽度，它们相互之间是有影响的。特征图中隐含的离散化意味着一个曲线部分可能看起来有点扁平或凹凸不平。得到的输出如图 7-5 所示。

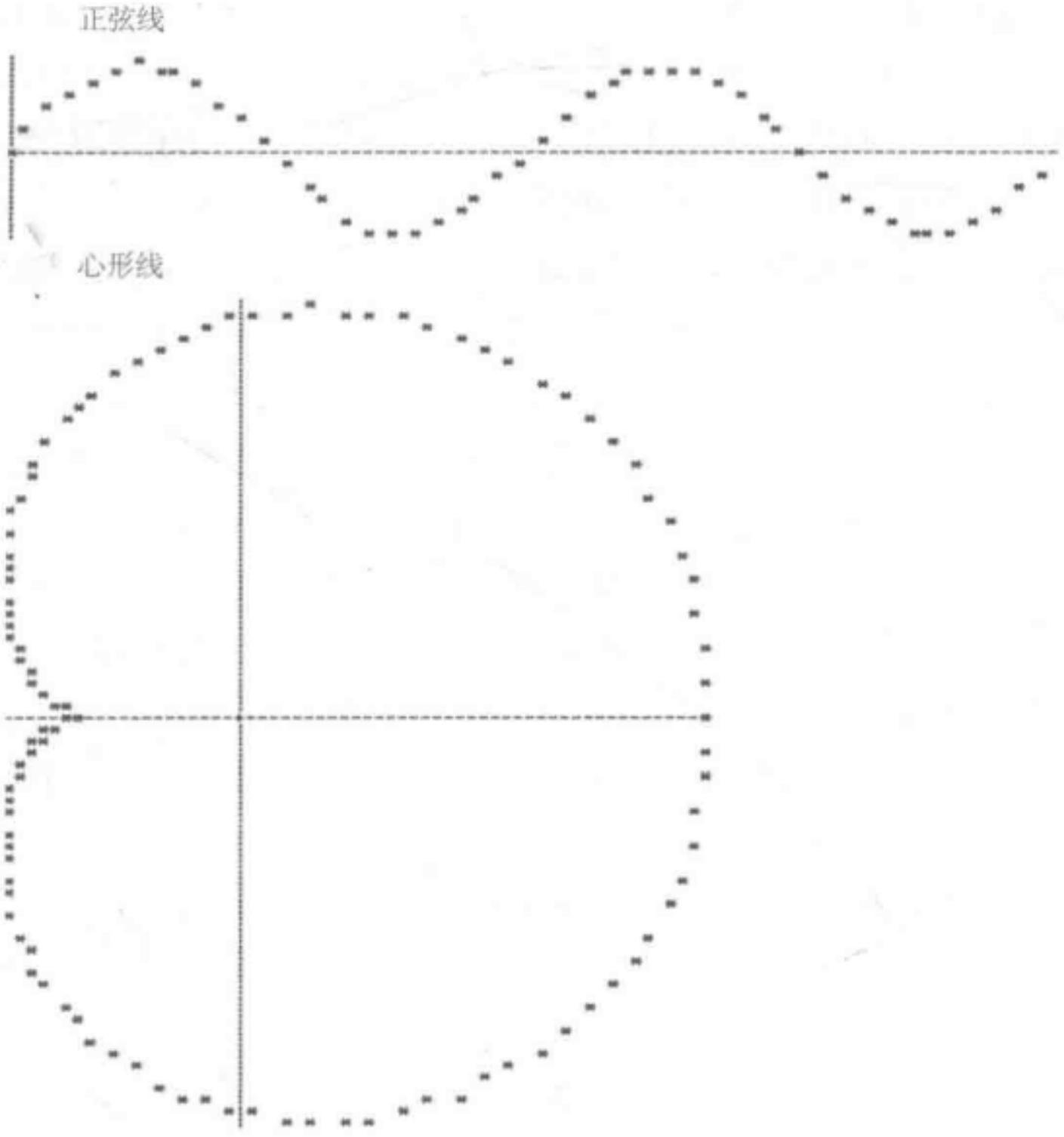


图 7-5　Ex 7_05.cpp 输出的曲线

7.12　本章小结

本章说明 STL 提供了很多种类的算法。为了执行给定的操作，通常对于一个算法有多种选择，除了写自己的循环之外。在任何情况下，最终的选择经常取决于个人的偏好。一般来说，算法通常会比编程中的显式循环要快，但有时使用循环的代码更易于理解。但是，使用自己的循环更容易出错，因此使用算法是首选。

下面是对本章所介绍的算法的总结，以供参考：

在序列中找出具有指定属性的元素的个数：

- all_of(Input_Iter beg, Input_Iter end, Unary_Predicate p)返回 true，如果 p 对[beg,end)中的全部元素都返回 true。

- any_of(Input_Iter beg, Input_Iter end, Unary_Predicate p)返回 true，如果 p 对[beg,end)中的任意一个元素返回 true。
- none_of(Input_Iter beg, Input_Iter end, Unary_Predicate p)返回 true，如果 p 对[beg,end)中的所有元素都返回 false。
- count(Input_Iter beg, Input_Iter end, const T& obj)返回[beg,end)中等于 obj 的元素的个数。
- count_if(Input_Iter beg, Input_Iter end, Unary_Predicate p)返回[beg,end)中使 p 返回 true 的元素的个数。

比较序列：

- equal(Input_Iter1 beg1, Input_Iter1 end1, Input_Iter2 beg2)返回 true，如果序列[beg1,end1)同 beg2 开始处对应的元素相等。
- equal(Input_Iter1 beg1, Input_Iter1 end1, Input_Iter2 beg2, Input_Iter2 end2)返回 true，如果序列[beg1,end1)中的元素和序列[beg2,end2)中对应的元素相等。
- equal(Input_Iter1 beg1, Input_Iter1 end1, Input_Iter2 beg2, Binary_Predicate p)返回 true，如果 p 为序列[beg1,end1)和序列 beg2 开始处对应的元素返回 true。
- equal(Input_Iter1 beg1, Input_Iter1 end1,Input_Iter2 beg2, Input_Iter2 end2, Binary_Predicate p)返回 true，如果 p 为[beg1,end1)和[beg2,end2)中对应的元素返回 true。
- mismatch(Input_Iter1 beg1, Input_Iter1 end1, Input_Iter2 beg2)返回一个 pair<Input_Iter1, Input_Iter2>对象，它包含第一对不相等元素的迭代器。
- mismatch(Input_Iter1 beg1,Input_Iter1 end1,Input_Iter2 beg2,Input_Iter2 end2)的返回和值上一个版本相同。
- mismatch(Input_Iter1 beg1, Input_Iter1 end1,Input_Iter2 beg2, Binary_Predicate p)返回一个 pair<Input_Iter1,Input_Iter2>对象，它包含第一对使 p 返回 false 的元素的迭代器。
- mismatch(Input_Iter1 beg1, Input_Iter1 end1, Input_Iter2 beg2, Input_Iter2 end2, Binary_Predicate p)的返回值和上一个版本相同。
- lexicographical_compare(Input_Iter1 beg1, Input_Iter1 end1,Input_Iter2 beg2, Input_Iter2 end2)返回 true，如果这两个序列包含的元素个数相同，并且对应元素相等；否则返回 false。
- lexicographical_compare(Input_Iter1 beg1, Input_Iter1 end1,Input_Iter2 beg2, Input_Iter2 end2,Binary_Predicate p)返回 true，如果这两个序列包含的元素个数相同，并且为所有对应的元素对 p 都返回 true；否则返回 false。

排列元素序列：

- next_permutation(Bi_Iter beg, Bi_Iter end)会按字典序的升序来生成元素的下一个排列，并返回 true，如果存在下一个排列的话。否则，元素会被排为序列的第一个排列，算法会返回 false。
- next_permutation(Bi_Iter beg, Bi_Iter end, Compare compare)会基于元素的比较函数 compare，按照字典序生成元素的下一个排列，并返回 true。如果不存在下一个排

列，元素会被排为基于 compare 的第一个序列，算法返回 false。

- prev_permutation(Bi_Iter beg, Bi_Iter end)会按字典序的升序生成元素的前一个排列，并返回 true，如果存在前一个排列的话。否则，元素会被排为序列中的最后一个排列，算法返回 false。
- prev_permutation(Bi_Iter beg, Bi_Iter end, Compare compare)会基于元素的比较函数 compare，按照字典序生成元素的前一个排列，并返回 true。如果不存在前一个排列，元素会被排为基于 compare 的序列的最后一个排列，算法返回 false。
- is_permutation(Fwd_Iter1 beg1, Fwd_Iter1 end1, Fwd_Iter2 beg2)返回 true，如果从 beg2 开始的元素序列(end1,beg1)是序列[beg1,end1)的一个排列，否则返回 false。它用==来比较元素。
- is_permutation(Fwd_Iter1 beg1, Fwd_Iter1 end1,Fwd_Iter2 beg2, Binary_Predicate p)和前一个版本相同，除了它是用 p 来比较元素是否相等之外。
- is_permutation(Fwd_Iter1 beg1, Fwd_Iter1 end1,Fwd_Iter2 beg2, Fwd_Iter2 end2)返回 true，如果[beg2,end2)是序列[beg1,end1)的一个排列，否则返回 false。它用==来比较元素。
- is_permutation(Fwd_Iter1 beg1, Fwd_Iter1 end1,Fwd_Iter2 beg2, Fwd_Iter2 end2, Binary_Predicate p)和前一个版本相同，除了它是用 p 来比较元素是否相等之外。

从序列中复制元素：

- copy(Input_Iter beg1, Input_Iter end1, Output_Iter beg2)会将序列[beg1, end1)复制到序列 beg2 的开始处。它返回的迭代器指向最后一个被复制到目的位置的元素的下一个位置。
- copy_n(Input_Iter beg1, Int_Type n, Output_Iter beg2)会从序列 beg1 的开始位置复制 n 个元素到 beg2 的开始位置。它返回的迭代器指向最后一个被复制到目的位置的元素的下一个位置。
- copy_if(Input_Iter beg1, Input_Iter end1,Output_Iter beg2, Unary_Predicate p)会从序列 beg1 的开始位置复制使 p 返回 true 的元素到 beg2 的开始位置。它返回的迭代器指向最后一个被复制到目的位置的元素的下一个位置。
- copy_backward(Bi_Iter1 beg1, Bi_Iter1 end1, Bi_Iter2 end2)会将序列[beg1, end1)复制到序列 end2 的末尾。这个操作会从 end1-1 指向的元素开始反向复制元素。算法会返回一个迭代器 iter，它指向被复制到目的位置的最后一个元素，因此在这个操作之后，目的序列为[iter, end2)。
- reverse_copy(Bi_Iter beg1, Bi_Iter end1, Output_Iter beg2)会反向地将[beg1, end1)复制到目的序列 beg2 的开始位置，并返回一个迭代器 iter，它指向被复制到目的位置的最后一个元素的下一个位置。因此，[beg2, iter)会以逆序包含[beg1, end1)中的元素。
- reverse(Bi_Iter beg, Bi_Iter end)会反转序列[beg, end)中元素的顺序。
- unique_copy(Input_Iter beg1, Input_Iter end1, Output_Iter beg2)会忽略连续的重复元素，将序列[beg1, end1)复制到序列 beg2 的开始位置。它是用==来比较元素的，它

返回一个指向被复制到目的位置的最后一个元素的下一个位置的迭代器。

- unique_copy(Input_Iter beg1, Input_Iter end1, Output_Iter beg2,Binary_Predicate p)所做的事和上一个版本相同，除了它是用 p 来比较元素之外。
- unique(Fwd_Iter beg, Fwd_Iter end)是通过将元素左移来覆盖前一个元素，从而移除连续的重复元素的。它用==来比较元素，并且这个算法会返回操作后得到的序列的结束迭代器。
- unique(Fwd_Iter beg, Fwd_Iter end, Binary_Predicate p)和前一个算法做相同的事情，除了用 p 来比较元素之外。

移动序列：

- move(Input_Iter beg1, Input_Iter end1, Output_Iter beg2)会将序列[beg1, end1)移到序列 beg2 的开始位置。它会返回一个指向最后一个被移到目的位置的元素的下一个位置的迭代器。beg2 不能在[beg1, end1)之内。
- move_backward(Bi_Iter1 beg1, Bi_Iter1 end1, Bi_Iter2 end2)会将序列[beg1, end1)移到序列 end2 的末尾，并且元素是以逆序移动的。算法会返回一个指向最后一个被移到目的位置的元素的迭代器。end2 不能在[beg1, end1)之内。

旋转元素的序列：

- rotate(Fwd_Iter beg, Fwd_Iter new_beg, Fwd_Iter end)会按逆时针旋转[beg, end)中的元素，从而使 new_beg 成为序列的第一个元素。算法会返回一个指向序列原始的第一个元素的迭代器。
- rotate_copy(Fwd_Iter beg1, Fwd_Iter new_beg1, Fwd_Iter end1, Output_Iter beg2)会将[beg1, end1)中的所有元素都复制到序列 beg2 的开始位置，从而使 new_beg1 指向的元素成为目的序列的第一个元素。算法会返回一个指向目的序列中最后一个元素的下一个位置的迭代器。

从序列中移除元素：

- remove(Fwd_Iter beg, Fwd_Iter end, const T& obj)可以从[beg, end)中移除等于 obj 的元素，并返回一个迭代器，它指向结果序列中最后一个元素的下一个位置。
- remove_if(Fwd_Iter beg, Fwd_Iter end, Unary_Predicate p)可以从[beg, end)中移除使 p 为 true 的元素，并返回一个迭代器，它指向结果序列中最后一个元素的下一个位置。
- remove_copy(Input_Iter beg1, Input_Iter end1, Output_Iter beg2, const T& obj)会将[beg1, end1)中的元素复制到 beg2 的开始位置，它会跳过等于 obj 的元素，并返回一个迭代器，它指向目的位置最后一个元素的下一个位置。
- remove_copy_if(Input_Iter beg1, Input_Iter end1,Output_Iter beg2, Unary_Predicate p)会将[beg1, end1)中的元素复制到 beg2 的开始位置，它会跳过使 p 返回 true 的元素，并返回一个迭代器，它指向目的位置最后一个元素的下一个位置。

替换序列中的元素：

- replace(Fwd_Iter beg, Fwd_Iter end, const T& obj, const T& new_obj)会用 new_obj 替换[beg, end)中等于 obj 的元素。

- replace_if(Fwd_Iter beg, Fwd_Iter end,Unary_Predicate p, const T& new_obj)会用 new_obj 替换[beg, end)中使 p 返回 true 的元素。
- replace_copy(Input_Iter beg1, Input_Iter end1,Output_Iter beg2, const T& obj, const T& new_obj)会将[beg1, end1)中的元素复制到序列 beg2 的开始位置，并用 new_obj 替换等于 obj 的元素。该算法返回一个迭代器，它指向目的位置最后一个元素的下一个位置。序列不能是重叠的。
- replace_copy_if(Input_Iter beg1, Input_Iter end1,Output_Iter beg2, Unary_Predicate p, const T& new_obj)会将[beg1, end1)中的元素复制到序列 beg2 的开始位置，并用 new_obj 替换使 p 返回 true 的元素。该算法返回一个迭代器，它指向目的位置最后一个元素的下一个位置。序列不能是重叠的。

修改序列中的元素：

- fill(Fwd_Iter beg, Fwd_Iter end, const T& obj)会将 obj 保存到序列[beg, end)的每一个元素中。
- fill_n(Output_Iter beg, Int_Type n, const T& obj)会将 obj 保存到序列 beg 开始的前 n 个元素中。
- generate(Fwd_Iter beg, Fwd_Iter end, Fun_Object gen_fun)会将 gen_fun 返回的值保存到序列[beg, end)的每一个元素中。gen_fun 必须是无参数的，并且必须能够返回一个可以保存到序列中的值。
- generate_n(Output_Iter beg, Int_Type n, Fun_Object gen_fun)会将 gen_fun 返回的值保存到序列 beg 开始的前 n 个元素中。算法会返回一个指向最后一个被保存元素的下一个位置的迭代器。
- transform(Input_Iter beg1, Input_Iter end1,Output_Iter beg2, Unary_Op op)会将 op 应用到序列[beg1, end1)的每一个元素上，并将这些元素返回的对应值保存到从 beg2 开始的序列中。
- transform(Input_Iter1 beg1, Input_Iter1 end1,Input_Iter2 beg2, Output_Iter beg3, Binary_Op op)会将 op 应用到序列[beg1, end1)中与从 beg2 开始的序列所对应的元素对上，并将这些元素返回的对应值保存到从 beg3 开始的序列中。

交换算法：

- swap(T& obj1, T& obj2)会交换 obj1 和 obj2 的值。第二个版本的 swap 可以交换两个同类型的数组，它们的长度必须相同。
- iter_swap(Fwd_Iter iter1, Fwd_Iter iter2)交换 iter1 和 iter2 指向的值。
- swap_ranges(Fwd_Iter1 beg1, Fwd_Iter1 end1, Fwd_Iter2 beg2)会交换序列[beg1, end1)和从 beg2 开始的序列的对应元素。该算法返回的迭代器指向从 beg2 开始的序列的最后一个元素。

恐怕我们不会那么轻松了，即使觉得目前看到的算法都很简单。在第 8 章中，你会遇到更多的算法。尤其是在第 10 章，会介绍专门用于数值计算的算法。

练习

尽可能用算法完成下面这些练习：

1. 写一个程序，用来从标准输入流读取像“month(string)day(integer)year(4-digit integer)”这种格式的数据，然后将数据保存到 deque 容器的 tuple 对象里。找出容器中不同月份的个数并输出。找到不同月份的名称并输出。将每个不同月份的数据复制到单独的容器中。按照一年中日的降序输出每个月份的数据。

2. 从标准输入流读取任意个数的像“feet (integer)inches(floating point)”这种格式的身高，并将每个身高作为一个 pair 对象保存到容器中。用 transform()算法将每个身高的度量值作为 tuple 对象(米、分米、毫米都是整数)保存到另一个容器中。transform()算法会在第三个容器中生成混合了两种身高表示的 pair 对象。用一个算法输出这个容器中的元素，从而展示身高的英尺、英寸及公制表示。

3. 写一个程序：

- 从标准输入流读入一个带标点的段落，并以单词为元素将它保存到序列容器中。
- 判断输入的重复单词的个数。
- 汇集所有少于 5 个字符的单词到一个容器中并输出它们。

4. 写一个程序，用来读取有任意个单词的短语，并用算法将它们的不同排列放到一个容器中，输出它们。用包含 5 个单词以上的短语测试这个程序(可以使用 5 个以上的单词，但是需要记住，n 个单词有 n！种排列)。

5. 从标准输入流读取任意个由名和姓组成的姓名。然后将这些姓名以 pair<string,string>元素的形式保存到容器中。将姓中有共同首字母的姓名提取到一个单独的容器中，并输出它们的内容。将原始的姓名集合复制到另一个以 pair<char, string>对象为元素的容器中，pair 的 first 成员是首字母，second 成员是姓。按照姓的长度输出新形式的姓名。

第 8 章

生成随机数

很多时候都需要生成随机数。大多数游戏程序以及模拟真实世界的程序，几乎都需要有生成随机数的能力。测试一个复杂的程序通常需要在一些点随机输入来验证程序在不同条件下能否工作，并且这种程序式的输入通常是很方便的。当然，也可以用随机数来生成随机选择的对象，因此可以生成随机选择的任何东西。

除非特别说明，否则在本章讨论的所有 STL 模板都定义在 random 头文件中。random 头文件中有很多内容，其中一些是非常专业的，肯定比这里讨论的详细。本章的目标是让你能够通过解释和演示，初步掌握 STL 提供的最有用的随机数生成能力。本章将介绍以下内容：

- 什么是随机数
- 什么是随机数生成引擎，STL 提供了哪些引擎
- 什么是随机数生成环境下的熵
- 什么是随机数生成器么，如何将它和引擎关联
- 什么是随机数生成引擎适配器
- 如何生成不确定的随机序列
- 什么是分布，STL 提供了什么样的分布
- 如何生成随机元素序列的随机排列

在本章中会有很多数学公式，它们可以清楚地解释算法是如何工作的。如果不擅长数学的话，可以忽略它们，这并不会对理解如何使用这种能力有所限制。

8.1 什么是随机数

随机性意味着无法预测。真正的随机性只存在于自然世界中。什么地方有闪电是随机的；如果有个地方正在酝酿风暴，可以相当确定那里会有闪电，但无法精确预测具体位置——所以不要站在树下。天气是随机的，每一天都会有明天的天气预报，通常是用昂贵的大型计算机来预测天气，尽管功能无比强大，但还是经常不准确。

为了描述一个数作为随机数所需要的环境——一个先前的数字序列。2 本身并不是随机数，它就是 2。然而，如果 2 是序列 46、1011、874、34、998871 中的下一个数，那么它就可能是随机数序列中的一个值。一个数是随机的，假定它出现在一个序列中，并且不能通过序列前面的元素来推测。当然，这并不意味着连续序列必须是不同的。序列 4、4、4、4 可能或不可能是一个随机序列的一部分。连续扔 4 次骰子可能会出现这种情况。这样就很难判断一个给定的序列是否是随机的，对于随机性有特别的数学测试，但在本书中不打算讨论。

数字计算机是确定性的，这意味着它对于给定的一系列输入值总会生成完全相同的结果。因此，除非硬件出现了严重的错误，否则任何程序代码生成的结果都不可能是随机的。因此计算机不能计算出随机数序列，下一个数的值总是由生成它的算法决定。然而，数字计算机可以产生数字的伪随机序列。它们是序列，因为伪随机属性只对序列这种环境有意义；被称作伪随机数是因为它们是由计算机生成的，因此并不是真正意义上的随机。话虽如此，仍会去掉“伪”——在本章剩下的部分把它们当作随机的。

8.2 概率、分布以及熵

生成随机数隐隐包含着统计学的一些概念，下面会对它们进行概述。本节只是为了让你熟悉这些概念，并且应该足以使你明白本章的其他内容，即使你可能从来没接触过这些概念。但要想完全理解这些概念，可以咨询一下统计学老师。

8.2.1 什么是概率

概率的值在 0 和 1 之间，它可以衡量事件发生的可能性。0 说明事件永远不会发生，1 说明事件肯定会发生。投一次骰子得到 6 的概率——或者实际上得到任何可能值的概率——都是 1/6。得到任意数字的概率是 1。

通常，事件发生的概率是它发生的次数除以它可能发生的次数。如果事件发生的概率是 p，那么它不发生的概率为 1–p。这样就为生活提供了宝贵指导。例如，在英国买彩票中奖的概率，从 49 个数中选取 6 个数，大约是 1/14 000 000；这意味着不中的概率大约为 13 999 999/14 000 000，基本是不可能中奖的。从这个角度来说，被闪电击中的概率是中彩票的 10 倍。

8.2.2 什么是分布

分布描述的是假设变量为序列中某个特定值的可能性。分布可以是离散或连续的：

- 离散分布描述的是假设变量是一组固定值中任意一个值的概率。根据定义，整数值的分布就是离散分布。用一个变量来表示投一次骰子的结果就是一个典型的离散分布的示例，它只能是从 1 到 6 的整数值。在离散分布中，所有可能值的概率之和为 1。
- 连续分布表示的是假设连续变量是序列中某个特定值的概率。连续变量可以是序

列中的任意一个值，一天中某个给定时间的温度就是一个示例。

用来表示一个连续随机变量在某个范围内的值的概率的曲线被叫作概率密度函数(Probability Density Function，PDF)。变量为给定值时，概率是点在 PDF 上对应的值。假设变量为 a 到 b 之间的任意值，变量的概率就是 a 到 b 之间的 PDF 曲线下方的面积。这意味着在 a 到 b 之间的 PDF 曲线下方的面积必须是 1，因为变量总为这个范围内的一个可能值。

离散变量的 PDF 被叫作离散概率函数。离散变量的不同值的概率通常使用图形的一套点或竖条来表示。像之前所说的那样，概率之和加起来必须为 1。

在现实世界中，有很多用来模型化事件是如何发生或如何测量的分布。它们通常被描述为数学公式，当以图形的形式展示时，就可以很容易地理解它们。图 8-1 展示了 4 个分布的示例。

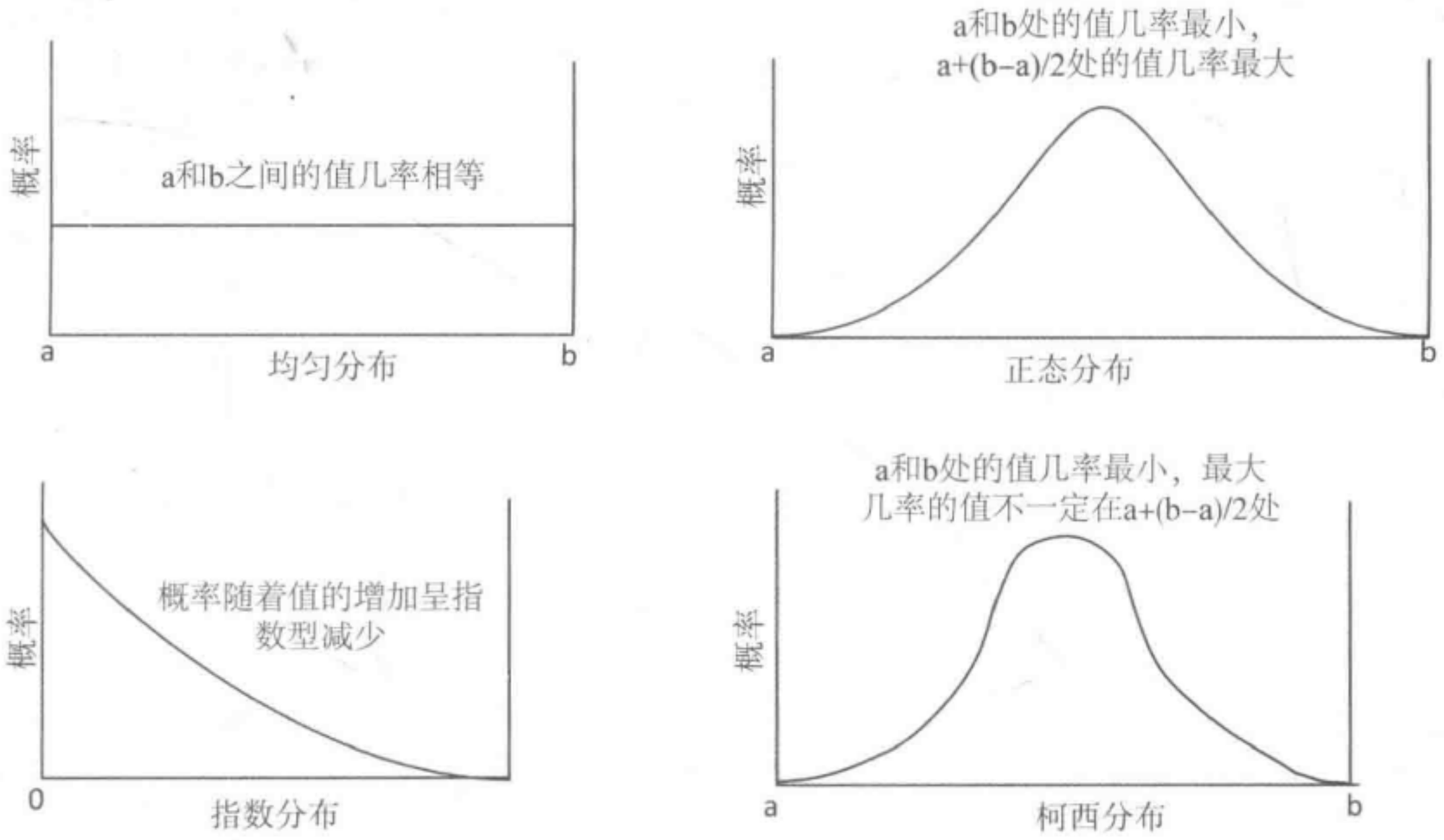


图 8-1　分布示例

在图 8-1 中，每个图表表示的是范围内可能值的出现概率。横轴记录的是变量的值；竖轴是概率。现实世界中不同种类变量的分布有很大的不同。在图 8-1 展示的均匀分布中，所有可能值的概率是相等的；均匀分布可以用来表示投掷一次骰子的结果。正态分布表示的是平均值每一侧的不同值。灯泡的寿命可能就是一个正态分布，因为灯泡的工作时间周期是分布在平均值两侧的——一般老式灯泡的灯丝可以使用 2000 小时，最新的 LED 灯泡被认为可以使用超过 15 000 小时，尽管后者在实践中并没有被确认。指数分布一般和随着时间推移而发生的事件相关：例如，放射性材料射出粒子的时间区间。图 8-1 中的第 4 个例子是柯西分布。在图示中，它看起来和正态分布隐约相似，但它们并不相同——曲线形状的差别可以很大。柯西分布在现实生活中出现的频率少于图 8-1 中所示的其他分布；它适用的一个场景是，量子力学的非稳态能量分布。有时候，会用不同的名称来引用一个分布；例如，正态分布也被叫作高斯分布。我们将会看到，STL 甚至支持更多其他种类的分布。

8.2.3 什么是熵

熵是对混乱程度的度量。当宇宙达到熵的最大值时，宇宙会发生热寂，根据热力学第二定律，这是不可避免的。但还不会发生——我们仍然有时间完成这本书。对于数据而言，熵的含义是由美国数学家克劳德·香农提出的。熵可以度量表示信息的效率；也可以度量数据的混乱程度。压缩文件通常用的无损算法，如果用它来生成一个 ZIP 文件，会增加熵。如果将文件压缩成 ZIP 文档，可以发现文件的大小并没有大大减小，这是因为原始文件有很高的熵——换句话说，它是非常随机的——因此不能被更有效地表示。英文文本的熵相当低，因为它不是特别随机。包含本章内容的英文文件如果被压缩到一个 ZIP 文件中，它的大小会大大减小，因为它不是随机数据。文件中的随机内容越少，数据的熵越低，压缩的潜力就越大。

在随机数生成的场景中，熵可以度量比特序列的随机性。最大的熵意味着全部随机序列的每一个比特都可能是 1 或 0；这种序列的信息量是最大的，因为信息不能被表示为更短的序列。最小的熵意味着序列是完全可预测的。转换其中的 1 和 0，或者诸如 1010 1010 1010 ... 或 1100 1101 1100 1101...这样的序列，它们有很低的熵。这种序列的信息量是很低的，因为它们是可预测的，并且可以很容易地用更短的序列来表示这些信息。当生成随机数时，希望可以使序列的熵最大，尽管这和生成这些值所需要计算的开销是不平衡的。

8.3 用 STL 生成随机数

在随机数生成方面 STL 有 4 个术语：

- 随机数生成引擎是一个定义了生成随机位序列的无符号整数序列机制的类模板。STL 定义了 3 个代表随机数引擎的类模板。本章的后面会对它们进行简短的介绍，但除非你对它们所使用的算法已经有深入的了解，否则不要直接使用它们，而应该使用随机数生成器。
- 随机数生成器是随机数引擎类模板的一个预定义的实例。每一个生成器都会将一套具体的模板参数应用到随机数引擎的类模板上——因此它是一个类型别名。STL 提供了几个预定义的随机数生成器，为了生成随机数，它们实现了一些著名的算法。
- 随机数引擎适配器是一个类模板，它通过修改另一个随机数引擎生成的序列来生成随机数序列。
- 分布表示的是随机序列中的数出现在序列中的概率。STL 定义了为各种不同的分布定义函数对象的类模板，包括图 8-1 所示的那些分布。

对于随机数生成，有多个分布类模板的原因是：我们在给定场景下生成的序列需要依靠数据的特性。医院前来就诊的病人的模式可能和到商店的顾客的模式有很大不同，因此需要运用不同的分布。而且，商店顾客的模式可能会有所不同，取决于商店的种类及其位置，因此对于不同商店的顾客的到达模型，可能也需要运用不同的分布。

之所以有多种随机数引擎和生成器，是因为没有一种算法可以生成适合所有情况的随机数。相比其他算法，有一些算法可以生成更长的无重复序列；一些算法要求的计算开销更

小。当知道想模型化的数据的特性时，就能够决定使用哪种分布和哪种随机序列生成能力。

8.3.1　生成随机数的种子

随机数的生成算法总是从单个或多个种子开始的，它们代表着产生随机数的计算的初始输入。种子决定了随机数生成器的初始状态和全部序列。随机数生成器的状态由用来计算序列的下一个值的所有数据组成。算法是递归的，因此种子会创建一个初始状态，它被用来生成序列的第一个值；生成的值会改变状态，然后用它来生成下一个值，以此类推。对于给定的单个或多个种子，随机数生成器总会生成相同的序列。显然在测试时，这很有帮助；可以确定程序是否正常工作，输入数据从任意一个运行到下一个至少也不是那么容易。当然，一旦程序被测试，我们希望程序每次运行都使用随机数生成器生成的不同序列。总是做同样事情的游戏程序也不会有趣。为了每次能够产生不同的序列，必须提供不同的种子——理想的随机值。它们被叫作不确定的值，因为它们是不可预测的。

STL 中的所有随机数生成算法都可以用单个种子来初始化。如果定义的初始状态需要更多的种子，它们可以自动生成。显然，随机数序列的熵取决于种子。种子的比特数是至关重要的，对于 1 字节的种子，只有 255 个可能的值，所以最多可生成 255 个不同的序列。为了使随机序列的熵达到最大，需要做两件事：需要一个真随机——不是伪随机的种子，以及种子的最大可能范围。

8.3.2　获取随机种子

random_device 类定义的函数对象可以生成用来作为种子的随机的无符号整数值。这个类应该使用非确定性数据源，它通常是由操作系统提供的。C++14 标准允许在非确定数据源不可用时，使用随机数引擎，但在大多数实现中，这没有必要。非确定性数据源可以是连续敲打键盘的时间区间，或者鼠标点击的区间，或者当前的时钟时间，或者一些物理特性。

可以按如下方式创建 random_device 对象：

```
std::random_device rd;                    // Source of seeds!
```

构造函数有一个 string&类型的参数，它有定义的默认值。在我们像这样省略它时，会得到我们环境中默认的 random_device 对象。理论上，参数允许我们提供一个字符串来标识使用的非确定性数据源，但需要检查我们的文档，看看我们的 C++库的这个选项是否可用。下面演示如何用 random_device 对象生成一个种子值：

```
auto my_1st_seed = rd();
```

这里用从函数对象 rd 得到的初始值生成了 my_1st_seed。下面是一个成功生成连续种子的程序：

```
// Ex8_01.cpp
// Generating a succession of 8 seeds
```

```
#include <random>                    // For random_device class
#include <iostream>                  // For standard streams

int main()
{
  std::random_device rd; // A function object for generating seeds
  for(size_t n {}; n < 8; ++n)
  std::cout << rd() << " ";
  std::cout << std::endl;
}
```

这段代码调用 8 次 rd 所表示的函数并输出它所返回的值。运行上述代码两次，得到了下面两行输出：

```
3638171046 3646201712 2185708196 587272045 1669986587 2512598564 1822700048 3845359386
360481879 3886461477 1775290740 2501731026 161066623 1931751494 751233357 3821236773
```

可以注意到，两次运行得到的输出是完全不同的。除了 operator()()，random_device 类只定义了其他的 3 个成员函数。成员函数 min()和 max()分别返回的是输出的最小和最大可能值。如果是用随机数引擎而不是非确定性数据源实现的，成员函数 entropy()返回的是将数据源看作 double 或 0 后的熵的估计值。

8.3.3 种子序列

seed_seq 类是用来帮助设置随机数生成器的初始状态的。正如我们看到的那样，可以生成一个随机数生成器，然后通过传入单个种子值到它的构造函数来设置它的初始状态。构造函数的参数也可以是 seed_seq 对象，它可以生成几个 32 位的无符号值，这些值为生成器提供的熵比单个整数多。也可以用 seed_seq 对象生成的值作为几个随机数生成器的种子。

seed_seq 类不仅仅是包含一系列值的简单容器。seed_seq 对象可以根据传入构造函数的一系列整数来生成任意个数的无符号整数值。这些生成的值是通过运用预定义的算法产生的。可以在 seed_seq 构造函数中指定一个或多个整数作为一个序列，或者作为一个初始化列表。不管输入值是如何分布的或者它们有多少个，这些生成的值都会分布到 32 位的无符号整数的全部范围内。对于相同的 seed_seq 构造函数的参数，总是可以得到相同的生成值序列。下面几个句子展示了生成一个 seed_seq 的几种方式：

```
std::seed_seq seeds1;                        // Default object
std::seed_seq seeds2 {2, 3, 4, 5};           // Create from simple integers
std::vector<unsigned int> data {25, 36, 47, 58}; // A vector of integers
std::seed_seq seeds3 {std::begin(data), std::end(data)};
 // Create from a range of integers
```

当然，也可以用 random_device 对象返回的值作为 seed_seq 构造函数的参数：

```
std::random_device rd {};
std::seed_seq seeds4 {rd(), rd()};   // Create from non-deterministic integers
```

每次运行这段代码，seed4 对象都会生成不同的值。

通过将两个指定范围的迭代器传给 seed_seq 对象的成员函数 generate()，可以将从 seed_seq 对象得到的给定个数的值保存到容器中。例如：

```
std::vector<unsigned int> numbers (10);      // Stores 10 integers
seeds4.generate(std::begin(numbers), std::end(numbers));
```

调用 seeds4 的成员函数 generate()可以保存 numbers 数组中被生成的值。通过一个示例，我们可以看到 seed_seq 对象在不同条件下生成的值的种类：

```
// Ex8_02
// Values generated by seed_seq objects
#include <random>                       // For seed_seq, random_device
#include <iostream>                     // For standard streams
#include <iterator>                     // For iterators
#include <string>                       // For string class
using std::string;

// Generates and list integers from a seed_seq object
void gen_and_list(const std::seed_seq& ss, const string title = "Values:",
  size_t count = 8)
{
  std::vector<unsigned int> values(count);
  ss.generate(std::begin(values), std::end(values));
  std::cout << title << std::endl;
  std::copy(std::begin(values), std::end(values),
            std::ostream_iterator<unsigned int>{std::cout, " "});
  std::cout << std::endl;
}
int main()
{
  std::random_device rd {};          // Non-deterministic source - we hope!
  std::seed_seq seeds1;              // Default constructor
  std::seed_seq seeds2 {3, 4, 5};    // From consecutive integers
  std::seed_seq seeds3 {rd(), rd()};

  std::vector<unsigned int> data {25, 36, 47, 58};
  std::seed_seq seeds4(std::begin(data), std::end(data)); // From a range
  gen_and_list(seeds1, "seeds1");
  gen_and_list(seeds1, "seeds1 again");
  gen_and_list(seeds1, "seeds1 again", 12);
  gen_and_list(seeds2, "seeds2");
  gen_and_list(seeds3, "seeds3");
  gen_and_list(seeds3, "seeds3 again");
  gen_and_list(seeds4, "seeds4");
  gen_and_list(seeds4, "seeds4 again");
  gen_and_list(seeds4, "seeds4 yet again", 12);
  gen_and_list(seeds4, "seeds4 for the last time", 6);
}
```

gen_and_list()是一个用来从 seed_seq 对象生成给定个数的值的辅助函数，并且按照标识标题输出它们。在 main()中展示了以不同的方式生成的 seed_seq 对象所生成的值。笔者得到了如下输出，但你们的肯定会在某些方面有所不同：

```
seeds1
3071959997 669715714 1197567577 671623915 1173633267 2920800313 1209690436 2235109613
seeds1 again
3071959997 669715714 1197567577 671623915 1173633267 2920800313 1209690436 2235109613
seeds1 again
3527767669 372316564 1386412362 441784 2145070594 2276674640 2205342996 1276311706
1119507491 75413245 2656280031 1908737279
seeds2
3388710944 2239790942 3836628790 2213304795 3411013659 2658117409 3275085354 3542843550
seeds3
3899021117 3310665364 4171568438 3922561248 250650243 1402466647 3483637440 3437969619
seeds3 again
3899021117 3310665364 4171568438 3922561248 250650243 1402466647 3483637440 3437969619
seeds4
2664408363 1749470183 3260020574 1632320446 534203587 2689329558 3154702548 1526239767
seeds4 again
2664408363 1749470183 3260020574 1632320446 534203587 2689329558 3154702548 1526239767
seeds4 yet again
2165145204 3274376652 3408995137 1909945219 3899048536 1143678586 807504975 3977354488
3428929103 552995692 24106733 509227013
seeds4 for the last time
1443036549 3195987434 1624705198 3337303804 479673966 3579734797
```

输出显示了关于 seed_seq 对象生成的值的如下事项：

- 无论如何生成 seed_seq 对象，都会得到范围广泛的 32 位的整数，甚至由默认构造函数构造的对象生成的值都会超出这个范围。
- 成员函数 generate()会生成尽可能多的不同值来填充指定的序列。
- 可以调用 generate()任意次数。

成员函数 generate()在序列中生成的值取决于序列的长度。给定长度的序列将会是完全相同的。不同长度的序列将会包含不同的值。

如果执行这个程序两次，可以看到对于 seed_seq 构造函数，相同的参数会生成相同的值。如果为构造函数提供不同的参数，不同的运行得到的序列是不同的，正如用 rd 函数对象返回的值。

seed_seq 类有两个其他的成员函数。成员函数 size()会返回用来生成对象的种子值的个数。成员函数 param()会保存原始种子的值；它期待用一个指定值的目的位置的输出迭代器作为参数，并且没有返回值。param()会将我们提供的原始种子值保存到迭代器参数所指定的开始位置。下面是一个展示这两个函数如何工作的代码段：

```
std::seed_seq seeds {3, 4, 5};
std::vector<unsigned int> data(seeds.size()); // Element for each seed value
seeds.param(std::begin(data)); // Stores 3 4 5 in data
```

这里会用 seeds 对象的成员函数 size()返回的值来确定生成的 vector 中元素的个数。seeds 的成员函数 param()会将传给构造函数的 3 个值保存到 data 中。也可以按如下方式将这些值添加到容器中：

```
seeds.param(std::back_inserter(data));     // Appends 3 4 5 to data
```

当然，不需要保存这些值——可以传入一个输出流迭代器作为 param()的参数：

```
seeds.param(std::ostream_iterator<unsigned int>{std::cout, " "}); // 3 4 5 to cout
```

8.4　分布类

STL 中的分布是一个函数对象：一个表示特定分布的类的实例。为了用给定的分布产生随机数，可以将分布应用到随机数生成器所产生的数中。这是通过将一个随机数生成器对象作为分布的函数对象的参数来做到的；这个函数对象会返回一个遵循这个分布的随机值。在详细讨论随机数生成器之前介绍分布似乎有点奇怪。这里有两个原因：

- 应该总是使用分布对象来获取随机数。分布对象将随机数生成器所产生的序列限制并塑造成了我们的应用场景所要求的形式。选择和使用合适的分布对象可以保证随机值在我们想使用的分布之内。直接将我们自己的算法应用到从随机数生成器得到的值上，不太可能产生具有合适特性的分布。
- 要充分理解随机数生成器之前的区别，需要具备它们实现的算法的数学知识。很多程序员不需要或不想了解这些。STL 提供的默认随机数生成器对大多数程序员来说都足够了，如果不够，需要将它和一个适当的分布对象结合。

在这一节会介绍默认随机数生成器，因此我们可以通过它来了解分布对象，然后讨论 STL 所提供的分布类型。在讨论 STL 所提供的分布类型之后，会解释其他的随机数生成器。

对于分布有 21 个类模板，其中的许多是相当专业的。此处会详细讨论哪些在笔者看来你是最感兴趣的，并且会向你展示该如何使用它们，剩下的只会讲解一个大概，留给你自己去更深入地学习，如果需要的话。如果只想生成随机序列而不想了解所有的可能，那么只需要掌握本章的这一节，可能就能满足你的全部需要了。但首先，要有效地使用分布，我们需要一个随机数生成器，这就是接下来的主题。

8.4.1　默认随机数生成器

默认随机数生成器是 std::default_random_engine 类型别名定义的随机无符号整数的通用源。这个别名表示实现是被定义的；应该看看我们的库文档为它提供的细节。它一般是 3 个随机数引擎模板中的一个实例，本章的后面会介绍它们。选择的模板类型参数需要能够为用户提供他们满意的序列。下面是一种生成 default_random_engine 类型的迭代器的简单方式：

```
std::default_random_engine rng1; // Create random number generator with
                                 // default seed
```

这里调用默认的构造函数，因此会用默认种子来设置初始状态。在不同的时刻执行从rng1 产生的随机数字序列总是相同的，因为种子保持不变。当然，也能自己提供种子：

```
std::default_random_engine rng2 {10}; // Create rng with 10 as seed
```

这里以 10 为种子生成了 rng2，因此从这个生成器中得到的随机序列和从 rng1 得到的是不同的，但仍然是固定的。如果想在每次执行代码时都得到不同的序列，需要提供非确定性的种子：

```
std::random_device rd; // Non-determinstic seed source
std::default_random_engine rng3 {rd()}; // Create random number generator
```

种子值是通过 random_device 类型的函数对象 rd 获得的。每一个 rd()调用都会返回不同的值，而且如果我们实现的 random_devic 是非确定性的，程序每次执行连续调用 rd()都会产生不同的序列。

另一个选择是，提供一个 seed_seq 对象作为 default_random_engine 构造函数的参数：

```
std::seed_seq sd_seq {2, 4, 6, 8};
std::default_random_engine rng4 {sd_seq}; // Same random number sequence each time
std::random_device rd;                    // Non-determinstic seed source
std::default_random_engine rng5 {std::seed_seq{rd(), rd(), rd()}};
```

这里第一次用从固定的初始种子生成的 seed_seq 对象生成 rng4 生成器；每次执行这段代码，从 rng4 得到的序列都是相同的。rng5 是从 seed_seq 构造的，seed_seq 拥有 random_device 函数对象产生的值，因此我们永远不知道会产生什么序列——每次都是一个惊喜。我们现在正处于一个创造和使用分布对象的位置，并开始一些严肃的随机活动。

8.4.2 创建分布对象

正如之前所说的，为了产生分布内的值，表示分布的函数对象需要一个随机数生成器对象作为参数。每次执行分布对象，都会返回一个在它所表示的分布之内的随机数，这个随机数是从随机数生成器所获得的值生成的。在分布返回的第一个值后面的连续值都依赖前面的值。创建一个分布对象需要一系列依赖分布类型的参数值。例如，均匀分布需要生成值的上下边界，而正态分布需要的是平均值和标准差。尽管不同类型的分布之间有本质上的不同，但它们还是有很多共同之处。所有的分布对象都有下面这些公共成员：

- result_type 是在类中为生成值的类型定义的类型别名。
- min()是一个返回分布对象可生成的最小值的成员函数。
- max()是一个返回分布对象可生成的最大值的成员函数。
- reset()是一个可以将分布对象重置为它的原始状态的成员函数，这样下次返回的值就不会依赖前一个值。这是否会发生取决于我们的实现。分布返回的值是独立的，reset()不会做任何事情。
- param_type 是定义在类中的一个结构体的类型别名。不同的分布需要不同的参数值，可能还需要是不同的类型，这些值会被保存到一个用来指定分布的 param_type

类型结构体中。

- param()是一个接受一个 param_type 类型的参数的成员函数，它会将分布对象的参数重置为新的值。
- param()的无参重载版本，它会返回分布对象所包含的 param_type 对象。
- 默认构造函数对于定义分布的参数有默认值。
- 一个构造函数，接受 param_type 类型的参数来定义分布。

后面会展示这些成员函数如何用于某些类型的分布。为了可以将分布对象的内部状态传送到流，或为了从流中读回状态，对流的插入和提取运算符、<和>>，为分布类型进行了重载。这样就提供了恢复程序的上一次执行中分布的状态的能力。

8.4.3　均匀分布

在均匀分布中，范围中内所有值都是等可能性的。均匀分布可以离散或连续的，如图 8-2 所示。

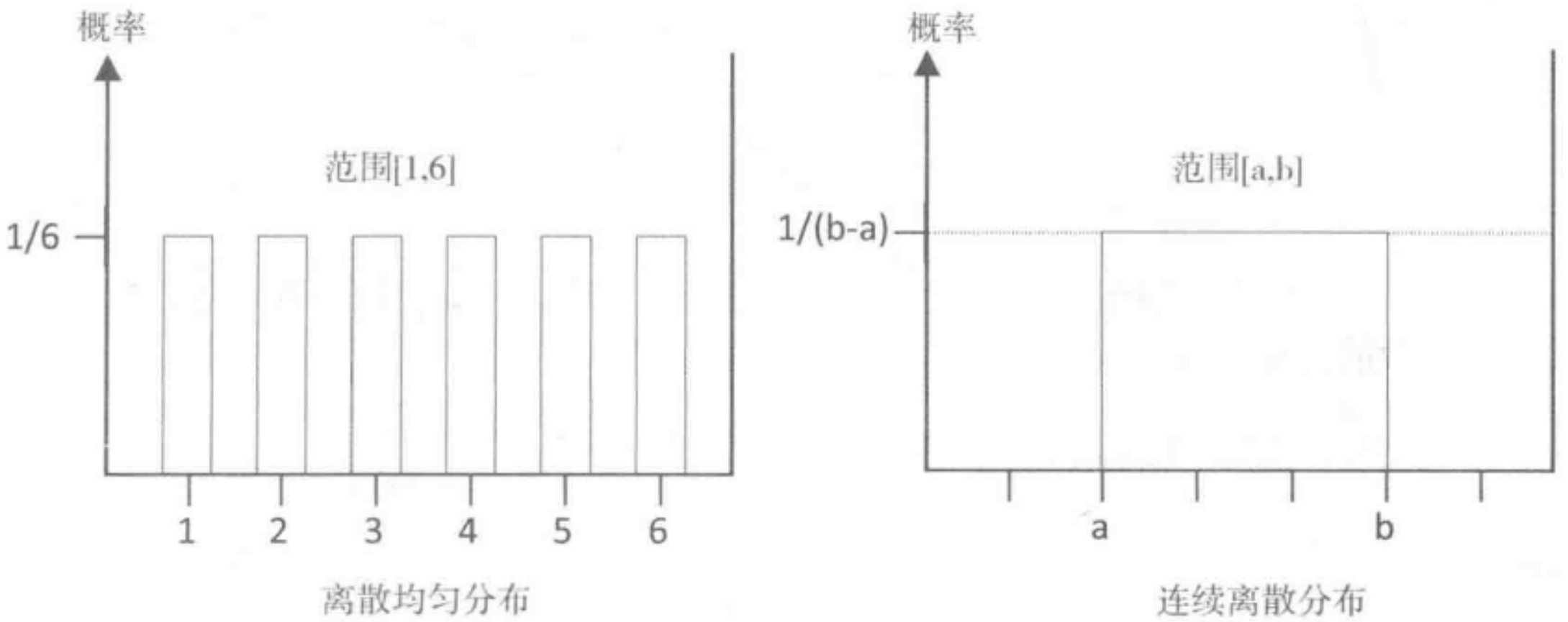


图 8-2　均匀分布

注意图 8-2 中的范围规范。离散均匀分布包含上边界和下边界；连续均匀分布不包含上边界，所以变量永远不会为上边界的值。

1. 离散均匀分布

uniform_int_distribution 类模板定义了分布对象，它返回的是均匀分布在闭合范围[a,b]内的随机整数。模板参数的类型是生成的整数的类型，默认类型是 int：定义在类中的类型别名 result_type 和分布所生成的值的类型相对应。模板类型参数只接受整数类型。一个构造函数有两个用来定义范围的上边界和下边界；下边界的默认值为 0，上边界的默认值是这种类型所生成的最大值。例如：

```
std::uniform_int_distribution<> d; // Distribution over 0 to max for type int, inclusive
std::cout << "Range from 0 to "
          << std::numeric_limits<std::uniform_int_distribution<> :: result-
            type>::max()
```

```
<< std::endl; // Range from 0 to 2147483647
```

第一条语句调用默认的构造函数来生成分布对象 d。所有的参数都是默认的，因此生成值的类型是 int，范围从 0 到 int 类型的最大值。最后的注释显示了得到的范围的界限；上边界是 int 类型的最大值，由定义在 limits 头文件中的函数模板 numeric_limits()生成。获取范围的界限有很多更加简单的方式。可以调用所有分布对象都有的成员函数 min()和 max()：

```
std::cout << "Range from " << d.min() << " to " << d.max() << std::endl;
```

在这种情况下，还有另外一种可能。uniform_int_distribution 类模板定义了成员函数 a()和 b()，它们可以分别返回范围的下边界和上边界。因此可以将前面的语句这样写：

```
std::cout << "Range from "<< d.a() << " to " << d.b() << std::endl;
```

成员函数 a()和 b()的名称比 min()和 max()更能说明它们所返回的值是如何和均匀分布关联的。

如果只想分布的整数范围大于或等于给定值，只需要提供构造函数的第一个参数：

```
std::uniform_int_distribution<> d {500};
std::cout << "Range from "<< d.a() << " to " << d.b()
<< std::endl;                                  // Range from 500 to 2147483647
```

当然，构造函数的参数也可以是负数。一般来说，需要指定范围的两个边界，这里有几个这样做的实际示例：

```
std::uniform_int_distribution<long> dist {-5L, 5L};
std::random_device rd;                         // Non-deterministic seed source
std::default_random_engine rng {rd()}; // Create random number generator
for(size_t i {}; i < 8; ++i)
std::cout << std::setw(2) << dist(rng) << " "; // -3 0 5 1 -2 -4 0 4
```

第一条语句定义了一个用来生成 long 类型随机整数的分布对象。范围从-5 到+5，所以这个分布对象能够返回 11 个可能的值。因此，每个可能值的出现概率为 1/11。通常，对于范围在[a,b]内的整数的均匀分布，返回的任何特定值的概率都是 1/(1+b-a)。在笔者系统上，所得到的输出如最后一句代码的注释所示，它肯定和在你系统上得到的不一样。

可以调用均匀分布的成员函数 param()来改变它所产生的值的范围。这里会向成员函数 param()传入一个均匀分布：指定新范围的 uniform_int_distribution 类——它的类型是由被分布类所定义的 param_type 别名指定的。也可以无参数地调用 param()来获取封装了这个分布的当前参数设置的对象。下面的代码说明了这两种可能：

```
std::uniform_int_distribution<> dist {0, 6};
std::random_device rd;                         // Non-determinstic seed source
std::default_random_engine rng {rd()};   // Create random number generator
for(size_t i {}; i < 8; ++i)
std::cout << std::setw(3) << dist(rng) << " ";     // first output line
std::cout << std::endl;
```

```
// Save old range and set new range
auto old_range = dist.param();                          // Get current params
dist.param(std::uniform_int_distribution<>::param_type {-10,20});
for(size_t i {}; i < 8; ++i)
  std::cout << std::setw(3) << dist(rng) << " ";   // Second output line
std::cout << std::endl;

// Restore old range...
dist.param(old_range);
for(size_t i {}; i < 8; ++i)
  std::cout << std::setw(3) << dist(rng) << " ";   // Third output line
std::cout << std::endl;
```

在笔者系统上，这段代码的输出如下：

```
6   1   5   6   1   3   6   2
19  16  15  5   0   7   6   -8
0   0   0   3   2   6   6   5
```

可以通过定义别名来简化 para()的参数类型：

```
using Range = std::uniform_int_distribution<>::param_type;
```

现在可以按如下方式用 param()调用来设置新的范围：

```
dist.param(Range {-10,20});
```

有很多场景能用到修改范围的界限的能力。一个显然的应用是，在一个程序中，当我们需要几个给定类型值的均匀分布时，每一个都需要一套不同的参数。这时候就可以只使用一个分布对象，然后根据需要在程序的任何地方设置参数。

应用均匀整数分布

在 Ex8_03.cpp 这个示例中会用 uniform_int_distribution 对象来处理牌。它的代码相当多，因此会一点一点介绍。为了可以将单个的分布对象用到代码的多个位置，我们可以像下面这样将它定义为一个函数中的静态对象：

```
std::uniform_int_distribution<size_t>& dist()
{
  static std::uniform_int_distribution<size_t> d;
  return d;
}
```

这个分布会返回 size_t 类型的值，它是一个无符号整数。最初会应用默认界限，但也可以调用 param()来为对象设置界限。为了使用分布对象，只需要调用 dist()来获取它的引用。我们可以用同样的方式将单个随机数生成器包装起来：

```
std::default_random_engine& rng()
{
  static std::default_random_engine engine {std::random_device()()};
  return engine;
}
```

这里用 random_device 对象返回的值来初始化静态引擎对象，因此每一次它被生成，都会产生一个不同的序列。表达式 dist(rng())会返回一个分布内的随机数。

用一个封装了花色和面值的 pair 对象来表示一张牌，用枚举类型来定义可能的花色和面值：

```
enum class Suit : size_t { Clubs, Diamonds, Hearts, Spades };
enum class Face : size_t { Two, Three, Four, Five, Six, Seven,
                           Eight, Nine, Ten, Jack, Queen, King, Ace };
```

枚举数的默认值是从 0 开始的，这一点非常重要，因为我们会用枚举数来索引容器，从而得到表示花色或面值名称的字符串。

可以为这种类型的牌、一手牌、桌面上的牌、手牌的集合定义别名：

```
using Card = std::pair<Suit,Face>;    // The type of a card
using Hand = std::list<Card>;         // Type for a hand of cards
using Deck = std::list<Card>;         // Type for a deck of cards
using Hands = std::vector<Hand>;      // Type for a container of hands
using Range = std::uniform_int_distribution<size_t>::param_type;
```

将桌面上的牌和手牌定义成 list 容器可以允许快速移除随机的牌。也可以用不同的序列容器，例如 vector。

我们会输出 Card 对象，因此实现 operator<<()来将 Card 对象写到流中是很有用的。下面是为 Card 对象重载<<的函数的定义：

```
std::ostream& operator<<(std::ostream& out, const Card& card)
{
  static std::array<string, 4> suits {"C", "D", "H", "S"}; // Suit names
  static std::array<string, 13> values {"2", "3", "4", "5", "6", "7",
                                        // Face value names
                                        "8","9","10","J","Q","K","A"};
  string suit {suits[static_cast<size_t>(card.first)]};
  string value {values[static_cast<size_t>(card.second)]};
  return out << std::setw(2) << value << suit;
}
```

下面是一个函数的定义，它可以将 Deck 容器初始为含有标准的 52 张牌：

```
Deck& init_deck(Deck& deck)
{
static std::array<Suit,4> suits{Suit::Clubs, Suit::Diamonds, Suit::Hearts,
  Suit::Spades};
static std::array<Face, 13> values {Face::Two, Face::Three, Face::Four,
  Face::Five, Face::Six, Face::Seven, Face::Eight, Face::Nine,Face::Ten,
  Face::Jack, Face::Queen, Face::King, Face::Ace};
deck.clear();
for(const auto& suit : suits)
  for(const auto& value : values)
  deck.emplace_back(Card {suit, value});
```

```
  return deck;
}
```

所有代表不同花色的对象都被保存在 suits 容器中。代表牌的可能面值的对象是 values 容器中的元素。这两个容器都是数组类型，几乎和使用标准的 C++数组一样高效。主要优势是，数组容器知道它自己的大小；在我们使用成员函数 at()时，它也能够检查越界的索引值。嵌套循环在 deck 中原地生成了表示每个花色全部值的 Card 对象。

初始的 Deck 对象中的牌是有序的。我们需要将牌分到每个人的手上，但我们想让每个人收到随机的牌。在分牌之前，我们需要先洗桌子上的牌，但为了练习使用不同界限的分布，我们会通过从桌面上随机选择来分牌。下面是一个这么做的函数：

```
void deal(Hands& hands, Deck& deck)
{
  auto d = dist();
  while(!deck.empty())
  {
    for(auto&& hand : hands)
    {
      size_t max_index = deck.size() - 1;
      d.param(Range{0, max_index});
      auto iter = std::begin(deck);
      std::advance(iter, d(rng()));
      hand.push_back(*iter);
      deck.erase(iter);
    }
  }
}
```

这里会处理作为第二个参数传入的 Deck 对象中的所有牌。不管作为第一个参数的 Hands 容器中有多少个 Hand 对象，牌都会被分配给 Hand 对象。外循环会继续循环，直到 deck 中的牌耗尽，当它的成员函数 empty()返回 true 时，就表示 deck 中的牌耗尽了。内循环会将牌一张一张地分配到 hands 中的每一个 Hand 中，因此，最终 deck 中的所有 Card 对象都会被分配到 hands 中的 Hand 对象中。

在这个循环中，max_index 被用来初始化 deck 中元素的合法的最大索引值。dist()返回的分布对象所产生的界限值会被设置为 0 到 max_index，这导致在每一次循环遍历中，分布对象生成的值都来自不同的范围。iter 会被初始化为 deck 中元素序列的开始迭代器，然后加上表达式 d(rng())的值，它将是一个从 0 到 max_index 的随机增量。在将*iter 指定的元素添加到当前的 Hand 后，就从 deck 容器移除这个元素。

手牌的顺序是随机的，因为它们是随机选择的，但一旦分配完毕，如果牌是升序的，手牌就会很容易处理。这个函数会对 Hands 容器中的每手牌进行排序：

```
void sort_hands(Hands& hands)
{
  for(auto&& hand : hands)
    hand.sort([](const auto& crd1, const auto& crd2) { return crd1.first <
```

```
            crd2.first ||(crd1.first == crd2.first && crd1.second <
            crd2.second); });
    }
```

循环遍历每个 hands 中的 Hand 对象。通过调用这个容器对象的成员函数 sort()来对容器中的每个元素进行排序。比较函数的参数是由一个通用的 lambda 表达式定义的，它会按花色和面值对 Card 对象排序。

当牌被分完时，我们肯定想输出 hands。下面的函数会处理这些：

```
void show_hands(const Hands& hands)
{
  for(auto&& hand : hands)
  {
    std::copy(std::begin(hand), std::end(hand), std::ostream_iterator<Card>
      {std::cout, " "});
    std::cout << std::endl;
  }
}
```

这里使用 copy()算法来将 hands 容器中的每手牌复制到用来写到 cout 的输出流迭代器。通过我们之前定义的 operator<<()函数，流迭代器会将 Hand 中的每个 Card 对象都写到输出流中。无论在什么时候我们想显示 hands，就可以只调用 show_hands()。

现在可以将这些函数放到一个完整的示例中，这个示例会生成一副标准的牌，分为 4 手牌，然后输出分配的每一手牌：

```
// Ex8_03.cpp
// Dealing cards at random with a distribution
#include <iostream>          // For standard streams
#include <ostream>           // For ostream stream
#include <iomanip>           // For stream manipulators
#include <iterator>          // For iterators and begin() and end()
#include <random>            // For random number generators & distributions
#include <utility>           // For pair<T1,T2> template
#include <vector>            // For vector<T> container
#include <list>              // For list<T> container
#include <array>             // For array<T,N> container
#include <string>            // For string class
#include <type_traits>       // For is_same predicate

using std::string;
enum class Suit : size_t {Clubs, Diamonds, Hearts, Spades};
enum class Face : size_t {Two, Three, Four, Five, Six, Seven,
                          Eight, Nine, Ten, Jack, Queen, King, Ace};
using Card = std::pair<Suit,Face>;       // The type of a card
using Hand = std::list<Card>;            // Type for a hand of cards
using Deck = std::list<Card>;            // Type for a deck of cards
using Hands = std::vector<Hand>;         // Type for a container of hands
using Range = std::uniform_int_distribution<size_t>::param_type;
```

```
// Definition of operator<<() for a Card object goes here...

// Definition of rng() to return a reference to a static default_random_engine
// object goes here...

// Definition of dist() for a reference to a static uniform_int_distribution<size_
// t> object here...

// Definition of init_deck() to initialize a deck to a full set of 52 cards here...

// Definition of deal() that deals a complete deck here...

// Definition of sort_hands() to sort cards in hands here...

// Definition of show_hands to output all hands here...

int main()
{
  // Create the deck
  Deck deck;
  init_deck(deck);

  // Create and deal the hands
  Hands hands(4);
  deal(hands, deck);

  // Sort and show the hands
  sort_hands(hands);
  show_hands(hands);
}
```

得到的 4 手牌如下所示：

```
3C  9C  10C QC  AC  2D  3D  9D  QD  2H  6H  JS  QS
2C  4C  6D  8D  JD  KD  3H  8H  9H  KH  9S  10S KS
5C  6C  8C  5D  AD  5H  10H JH  QH  3S  4S  7S  AS
7C  JC  KC  4D  7D  10D 4H  7H  AH  2S  5S  6S  8S
```

当然，在这个示例中，为了进一步加强对分布的练习，可以对它进行扩展。现在参与者开始打牌，每一轮，每个人会打出一张随机的牌。本书不会告诉你应该如何扩展，它是在本章末尾需要完成的练习 2。

2. 连续均匀分布

uniform_real_distribution 类模板定义了一个默认返回 double 型浮点值的连续分布。可以按如下方式生成一个返回值在范围[0,10)内的分布对象：

```
std::uniform_real_distribution<> values {0.0, 10.0};
std::random_device rd;                          // Non-deterministic seed source
std::default_random_engine rng {rd()}; // Create random number generator
for(size_t i {}; i < 8; ++i)
  std::cout << std::fixed << std::setprecision(2)
```

```
        << values(rng) << " ";      // 8.37 6.72 6.41 6.08 6.89 6.10 9.75 4.07
```

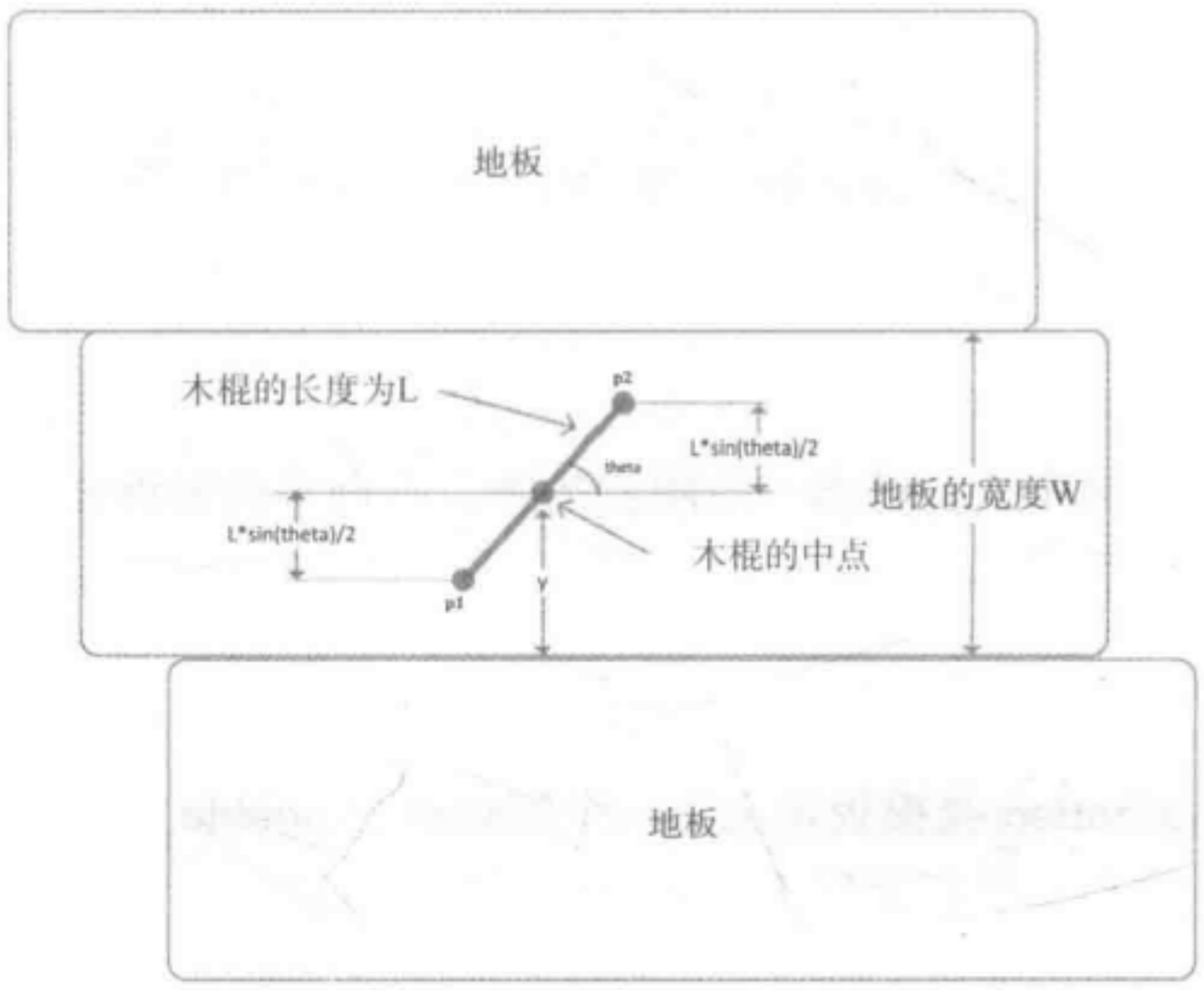
生成和使用 uniform_real_distribution 函数对象的方式和 uniform_int_distribution 有密切的相似之处。可以将一个随机数生成器对象作为参数值给分布函数对象来获取一个随机值。可以通过调用对象的成员函数 param()来获取和设置范围的界限。除了返回分布界限的成员函数 min()和 max()，uniform_real_distribution 对象也有成员函数 a()和 b()。注意，连续分布的范围是半开放的，分布对象返回值的范围不包括上边界。

现实世界中，能够应用均匀连续分布的变量是很少的。例如，和天气相关的参数的测量中，范围值并不是等可能性的。当我们看手表时，秒针的位置可能是一个均匀分布，但这没有什么用，这可能是真正的示例。均匀连续分布被用在应用于金融业的蒙特卡罗方法中，也被用到了工程和科学中。将它们放到另一个示例中——一个使用连续均分分布或 π 的值程序。

使用连续均匀分布

你知道可以用一根木棍来确定 π 的值吗？当然，并不是让你用木棍来威胁一个数学家，从而让他告诉你 π 的值——可以扔任意一根直的物品——铅笔甚至法拉克福香肠都可以。必须是一块光秃秃的带有地板的地面，并且扔的物品也必须是直的，它的长度必须小于地板的长度。过程很简单——只需要数扔木棍的次数，以及木棍落地时，它穿越地板边缘的次数。

为了得到一个不错的结果，需要将棍子抛掷一定的次数。当然，这会花费一些时间，并且还有点乏味，并不是说有多累。在几组均匀分布的帮助下，我们可以通过电脑来模拟抛掷——并计算出 π 的值。图 8-3 展示了木棍在地面上的随机位置以及它和地板的关系。为了解释发生了什么，需要用到一点数学知识，但并不难。



点p1在y−L*sin(theta)/2处
点p2在y+L*sin(theta)/2处

当y+L*sin(theta)/2≥W或y-L*sin(theta)/2≤0 时，木棍将穿过地板的边缘

图 8-3　地板上的木棍

图 8-3 显示了木棍在任意位置和地板底部边缘的距离以及相对地板的角度。木棍总会落到一个地板或另一个地板上，因此我们只需要考虑一个地板。图 8-3 显示了木棍和地板边缘相交的条件。在这种情况下，由 p1 和 p2 表示的木棍的任意一端在地板的上面或它的边缘。因为木棍只会穿过一个地板的边缘，从木棍的中心到闭合边缘之间的距离必须小于 L*sin(theta)/2。

我们会将木棍抛掷很多次，然后用 throws 来记录抛掷的总次数，木棍和地板的边缘重叠的次数记为 hits。因此现在我们有两个计数，我们如何从它们得到 π 值？图 8-4 会有些帮助。

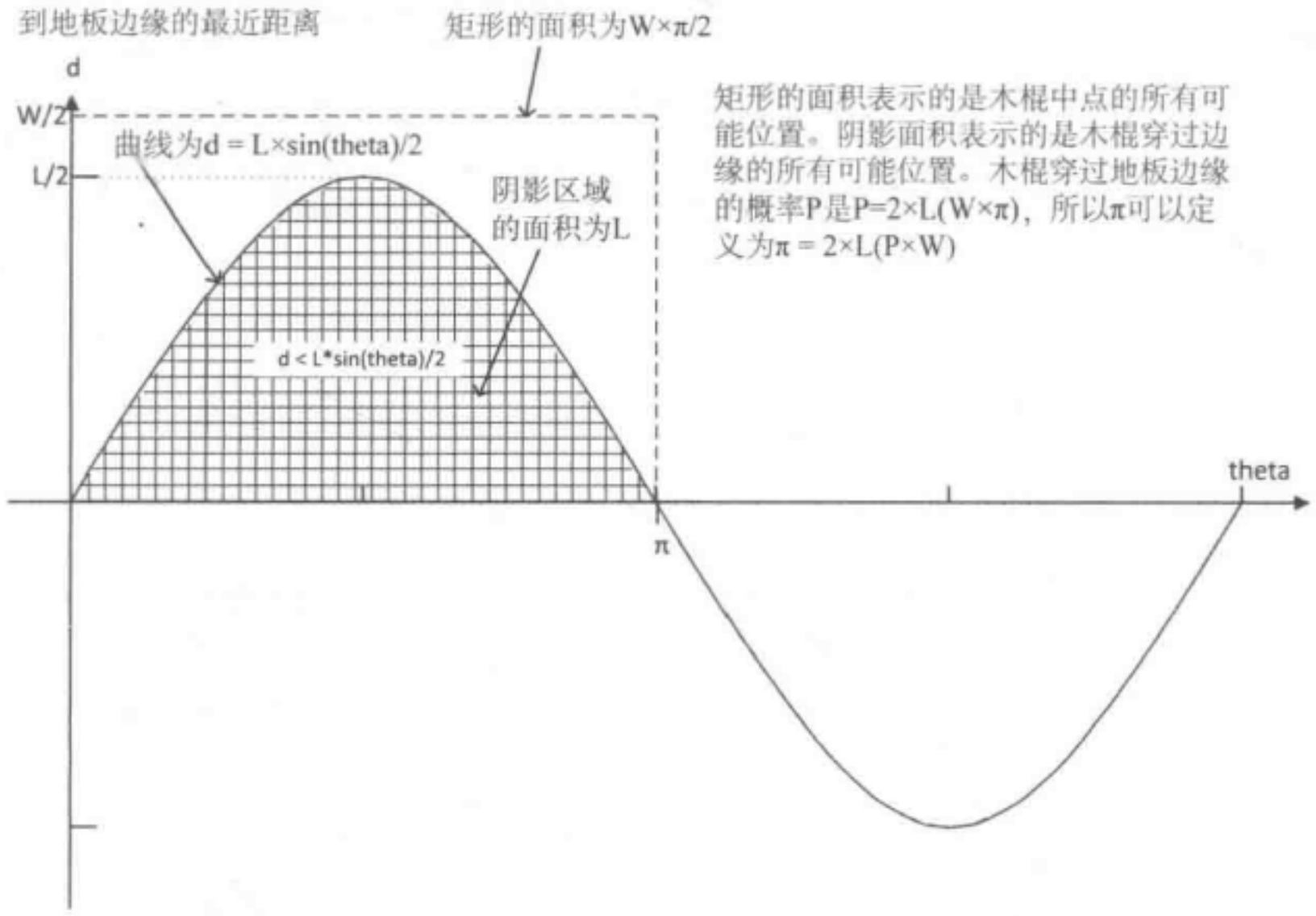


图 8-4　通过从木棍穿过地板边缘的概率来确定 π 值

木棍穿过地板边缘的概率 P 为 hit 和 throws 的比率，因此我们能用 P 以及图 8-4 中最后的等式来得到 π 值。π 值是下面这个表达式的结果：

```
2*stick_length*throws/(board_width*hits)
```

现在我们知道它是如何工作的了。下面是它的代码实现：

```
// Ex8_04.cpp
// Finding pi by throwing a stick
#include <iostream>              // For standard streams
#include <random>                // For distributions, random number gen
#include <cmath>                 // For sin() function

int main()
{
  const double pi = 3.1415962;
  double stick_length{};         // Stick length
  double board_width {};         // Board width
  std::cout << "Enter the width of a floorboard: ";
```

```
    std::cin >> board_width;
    std::cout << "Enter the length of the stick (must be less than "
              << board_width << "): ";
    std::cin >> stick_length;
    if(board_width < stick_length) stick_length = 0.9*board_width;

    std::uniform_real_distribution<> angle {0.0, pi}; // Distribution for
                                                      // angle of stick

    // Distribution for stick center position, relative to board edge
    std::uniform_real_distribution<> position {0.0, board_width};

    std::random_device rd; // Non-deterministic seed source
    std::default_random_engine rng {rd()}; // Create random number generator
    const size_t throws{5'000'000}; // Number of random throws
    size_t hits {}; // Count of stick intersecting the board

    // Throw the stick down throws times
    for(size_t i {}; i < throws; ++i)
    {
      double y {position(rng)};
      double theta {angle(rng)};
      // Check if the stick crosses the edge of a board
      if(((y + stick_length*sin(theta)/2) >= board_width) ||
                            ((y - stick_length*sin(theta) / 2) <= 0))
        ++hits; // It does, so increment count
      }
      std::cout << "Probability of the stick crossing the edge of a board is: "
                << (static_cast<double>(hits)/ throws) << std::endl;
      std::cout << "Pi is: " << (2* stick_length*throws)/(board_width*hits) << std::endl;
    }
```

你可能已经看到，确定 π 值的这个程序中有一点小小的瑕疵，在求 π 值之前需要知道它的值。然而，它仅仅是一个模拟——以及一个使用均匀分布的理由。唯一的取代方式是真正地扔 5 000 000 次木棍，如果身体不错并且有一根耐用的木棍的话，这可能也是一个选择。如果每 3 秒扔一次，需要大约 8 个月——而且还不能睡觉或吃饭……

相对于地板边缘的木棍中心的随机位置是由 position 分布对象生成的，木棍在每个位置的角度是由 angle 分布生成的。循环体中实现了图 8-3 中的计算，循环后的代码使用了图 8-4 中的公式。在你的系统上可以将扔木棍的次数改为合理的值。笔者得到的输出如下：

```
Enter the width of a floorboard: 12
Enter the length of the stick (must be less than 12): 5
Probability of the stick crossing the edge of a board is: 0.265281
Pi is: 3.14132
```

当然，运行之间的输出会有所不同，因为程序每次执行时会产生不同的随机序列。木棍相对于地板的长度也会受此影响，throws 的大小也一样。

■ 注意：图 8-4 中曲线下的阴影面积就是木棍的长度，这需要一点微积分知识——面积为：

$$\text{area}a=\frac{\mathrm{L}}{2}\int_0^n \sin\theta$$

$\sin\theta$ 的积分为$(-\cos\theta)$，因此面积为 $\frac{L}{2}((-\cos\pi)-(-\cos\theta))$，它等于 $\frac{L}{2}(1+1)$，也就是 L!

3. 创建标准均匀分布

标准均匀分布是一个在范围[0,1)内的连续分布。generate_canonical()函数模板会提供一个浮点值范围在[0,1)内，且有给定的随机比特数的标准均匀分布。它有 3 个模板参数：浮点类型、尾数的随机比特的个数，以及使用的随机数生成器的类型。函数的参数是一个随机数生成器，因此最后一个模板参数可以推导出来。下面是它可能的用法：

```
std::vector<double> data(8);                  // Container with 8 elements
std::random_device rd;                        // Non-deterministic seed source
std::default_random_engine rng {rd()}; // Create random number generator

std::generate(std::begin(data), std::end(data),
              [&rng] { return std::generate_canonical<double, 12>(rng); });

std::copy(std::begin(data), std::end(data), std::ostream_iterator<double>
  {std::cout, " "});
```

在 lambda 表达式中，被调用的 generate_canonical()函数被用来作为 generate()算法的第三个参数。lambda 表达式会返回一个有 12 个随机比特的 double 类型的随机值，因此 generate()会用这种数据来填充 data 中的元素。在笔者系统上，执行这些语句会产生如下输出：

```
0.766197 0.298056 0.409951 0.955263 0.419199 0.737496 0.547764 0.91622
```

上面的输出说明位数可能我们想要的要多，记住我们只指定了 12 个随机比特。可以按如下方式限制输出：

```
std::copy(std::begin(data), std::end(data),
std::ostream_iterator<double>{std::cout << std::fixed << std::setprecision
  (4) , " "});
```

流操作符被应用到每个输出值，因此现在的输出可能如下所示：

```
0.8514 0.5707 0.8322 0.6626 0.7026 0.8854 0.5427 0.8886
```

如果真的想得到这些位，可以用 hexfloat 以十六进制的格式输出这些值。

显然，随机位数越少，可能的随机值的范围越有限。可以通过将位数指定为这个类型的最大值来使范围达到最大。下面是展示如何这么做的代码：

```
std::vector<long double> data;                // Empty container
std::random_device rd;                        // Non-deterministic seed source
```

```
std::default_random_engine rng {rd()}; // Create random number generator
std::generate_n(std::back_inserter(data), 10, [&rng]
{return std::generate_canonical<long double, std::numeric_limits<long
    double>::digits>(rng); });

std::copy(std::begin(data), std::end(data), std::ostream_iterator<long
  double>{std::cout, " "});
std::cout << std::endl;
```

注意，这和前面的代码有些区别。这次，generate_n()的第一个参数是 data 容器的 back_insert_iterator，因此可以通过调用它的成员函数 push_back()来添加元素。generate_canonical()的第二个模板参数是 numeric_limits 对象的 long double 类型的成员变量的 digits 值。这是这个类型的位数的比特数，因此可以指定这个类型可能的随机比特位的最大个数(在笔者系统上只有 53 个)。笔者得到了这样的输出，但你的可能是不同的：

```
0.426365 0.0635646 0.208444 0.198286 0.338378 0.490884 0.841733 0.975676 0.193322 0.346017
```

8.4.4 正态分布

图 8-5 显示的是正态(或高斯)分布。它是一条连续的贝尔曲线，期望两边的值是相等的——期望就是平均值。它是一个概率分布，因此曲线下方的面积是 1。正态分布是由两个参数完全定义的：期望和标准差——这是衡量期望两边的值如何分布的一种方式。

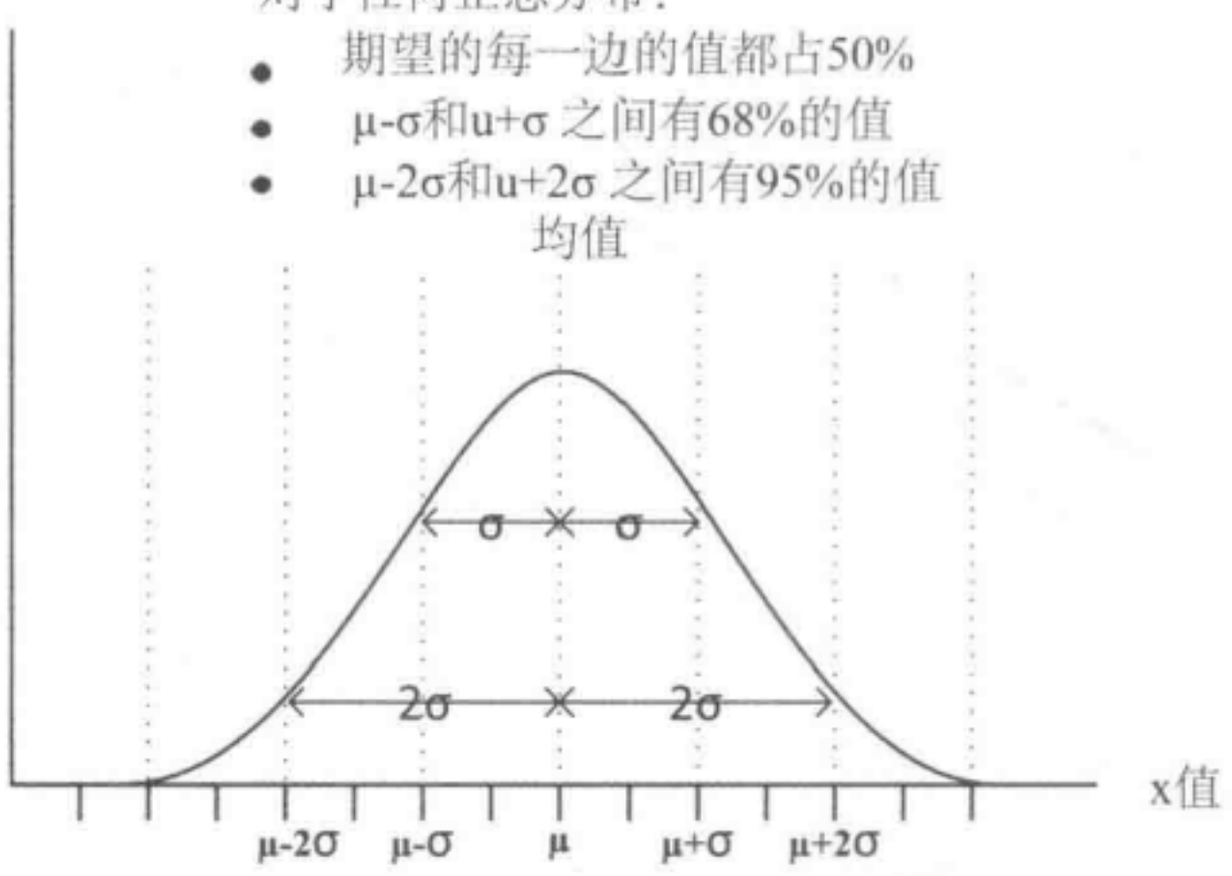


图 8-5 正态分布

期望和标准差分别是用希腊字母 μ 和 σ 来表示的，变量 x 有 n 个样本，这些是由下面的公式定义的：

$$\mu = \frac{\sum_{0}^{n} x_i}{n} \qquad \sigma = \sqrt{\frac{\sum_{0}^{n} (x_i - \mu)^2}{n-1}}$$

因此，期望就是值的和除以值的个数——换句话说，也就是平均值。可以通过值和期望的差值的平方子和除以 n-1，然后对结果开方来得到标准差。对于不同的期望和标准差

的值，正态分布的相对宽度和高度分布曲线的变化是相当大的。但是，分布值总是如图 8-5 所示。这意味着，如果知道一个符合正态分布的变量的期望和标准差，例如在大量人口中个体的身高，就可以知道 95%的人身高不超过期望的 2σ。标准正态分布的期望为 0，标准差为 1。

uniform_distribution 模板定义了可以产生随机浮点值的分布对象类型，默认是 double 类型。默认构造函数创建的是标准正态分布——因此期望是 0，方差是 1.0：

```
std::normal_distribution<> dist; // mu: 0 sigma: 1
```

下面展示了如何创建一个有特定值和标准差的正态分布：

```
double mu {50.0}, sigma {10.0};
std::normal_distribution<> norm {mu, sigma};
```

这里定义了一个生成 double 值的分布对象，期望为 50.0，标准差是 10.0。为了生成值，可以将一个随机数生成器传给 norm 函数对象。例如：

```
std::random_device rd;
std::default_random_engine rng {rd()};
std::cout << "Normally distributed values: "
          << norm(rng) << " " << norm(rng) << std::endl; // 39.6153 45.5608
```

可以通过调用对象的成员函数 mean()和 stddev()来获取它的期望值和标准差：

```
std::cout << "mu: " << norm.mean() << " sigma: " << norm.stddev()
          << std::endl; // mu: 50 sigma: 10
```

通过调用无参数的成员函数 param()，可以得到一个封装了这两个值的 param_type 对象。为了设置期望或标准差，需要将一个 param_type 对象传给成员函数 param()。分布类有用来获取期望和标准差的成员，param_type 对象拥有和它们的名字相同的成员函数。下面是一个示例：

```
using Params = std::normal_distribution<>::param_type; // Type alias for
                                                       // readability
double mu {50.0}, sigma {10.0};
std::normal_distribution<> norm {mu, sigma};        // Create distribution
auto params = norm.param(); // Get mean and standard deviation
norm.param(Params {params.mean(), params.stddev() + 5.0}); // Modify params
std::cout << "mu: " << norm.mean() << " sigma: " << norm.stddev()
          << std::endl; // mu: 50 sigma: 15
```

这里调用无参数的 param()来获取包含期望和方差的 param_type 对象。在第二个 param()调用中，通过传入一个 Paras 对象将标准差增加了 5.0。

可以通过传入一个 param_type 对象作为一个分布对象调用的第二个参数来临时设置期望和标准差：

```
using Params = std::normal_distribution<>::param_type; // Type alias for
                                                       // readability
```

```
std::random_device rd;
std::default_random_engine rng {rd()};
std::normal_distribution<> norm {50.0, 10.0};     // Create distribution
Params new_p {100.0, 30.0};                       // mu=100 sigma=30
std::cout << norm(rng, new_p) << std::endl; // Generate value with new_p: 100.925
std::cout << norm.mean() << " " << norm.stddev()
          << std::endl;                           // 50 10
```

new_p 定义的期望和标准差只会被应用到它被作为第二个参数传入的 norm 的执行中。原始的期望和标准差会被应用到随后的没有第二个参数的 norm 调用中。

成员函数 min()和 max()返回的是分布可以产生的最小值和最大值。对于分布来说，这并不是特别有用。因为返回值的类型可以这样表示最大值和最小值：

```
std::cout << "min: " << norm.min() << " max: " << norm.max()
          << std::endl; // min: 4.94066e-324 max: 1.79769e+308
```

使用正态分布

在这个示例中，会用从键盘输入的期望和标准差来生成一个正态分布对象。程序会用这个分布对象生成大量的随机值，然后将它们绘制成直方图来展示曲线的形状。概率会跨页，样本值在页的下面。下面是绘制值的范围的函数模板的代码：

```
template<typename Iter>
void dist_plot(Iter& beg_iter, Iter& end_iter, size_t width=90)
{
// Create data for distribution plot
std::map<int, size_t> plot_data; // Elements are pair<value, frequency>

// Make sure all values are present in the plot
auto pr = std::minmax_element(beg_iter, end_iter, [](const double v1,
                                                     const double v2)
                                                    {return v1 < v2; });
for(int n {static_cast<int>(*pr.first)}; n < static_cast<int>(*pr.second); ++n)
  plot_data.emplace(n,0);
// Create the plot data
std::for_each(beg_iter, end_iter, [&plot_data](double value)
  { ++plot_data[static_cast<int> (std::round (value))]; });
// Find maximum frequency to be plotted - must fit within page width
size_t max_f {std::max_element(std::begin(plot_data), std::end(plot_data),
  [](const std::pair<int,int>& v1, const std::pair<int,int>& v2)
                              { return v1.second < v2.second; })->second};

  // Draw distribution as histogram
  std::for_each(std::begin(plot_data), std::end(plot_data),
    [max_f, width](const std::pair<int, int>& v)
  {std::cout << std::setw(3) << v.first << " -| "
                             << string((width*v.second) / max_f, '*')
                             << std::endl; });
}
```

首先通过创建一个包含 pair 元素的 map 容器来生成绘图，每一个 pair 元素中包含了一个整数值及其出现频率。以低值来绘制，某个范围内没有值生成是可能的，但这些仍然应该在直方图中表示出来。为了确保范围内的所有值都可以绘制出来，我们要用 minmax_element()算法找出最小值和最大值，然后将这个范围内的所有元素添加到 map 中，并以 0 为初始计数。minmax_element()算法会返回一个指向最小和最大元素的迭代器，因此必须对它们解引用才能够得到值。因为输入序列的值可以是浮点数，在被保存到 map 之前，可以将它们转换为整数。在将值转换为 size_t 类型之前，lambda 表达式会用定义在 cmath 头文件中的 round()来将每个值取整为最接近的整数，然后通过 for_each()算法将这个 lambda 表达式运用到每个元素上。结果被保存在 map 中。不可能事件的整数值不在 map 中，它会被作为新元素添加到 second 成员中——它的频率——加 1；如果已经在 map 中，pair 的 second 成员会增加。round()函数取整的值在整数之间，从而避免偏向于 0。

用 max_element()算法来获得出现频率最大的值。在这个示例中，是用比较元素的 second 成员的 lambda 表达式来比较元素的。max_f 变量是由 max_element()返回的迭代器所指向的 pair 的 second 成员来初始化的。直方图是用 for_each()算法绘制的一系列 string 对象来表示的。缩放后，为了使最大值能够匹配一行字符的宽度，每个字符串都会包含一定数量的星号，它们和频率次数对应。

下面是充分使用 dist_plot()的这个程序：

```
// Ex8_05.cpp
// Checking out normal distributions
#include <random>                       // For distributions and random
number generators
#include <algorithm>                    // For generate(), for_each(),
// max_element(), transform()
#include <numeric>                      // For accumulate()
#include <vector>                       // For vector container
#include <map>                          // For map container
#include <cmath>                        // For pow(), round() functions
#include <iostream>                     // For standard streams
#include <iomanip>                      // For stream manipulators
#include <string>                       // For string class
using std::string;
using Params = std::normal_distribution<>::param_type;

// Template for dist_plot() function goes here...

int main()
{
  std::random_device rd;
  std::default_random_engine rng {rd()};
  std::normal_distribution<> norm;
  double mu {}, sigma {};
  const size_t sample_count {20000};
  std::vector<double> values(sample_count);
  while(true)
```

```
    {
      std::cout << "\nEnter values for the mean and standard deviation, or
                   Ctrl+Z to end: ";
      if((std::cin >> mu).eof()) break;
      std::cin >> sigma;
      norm.param(Params{mu, sigma});
      std::generate(std::begin(values), std::end(values), [&norm, &rng] { return
        norm(rng); });

      // Create data to plot histogram and plot it
      dist_plot(std::begin(values), std::end(values));

      // Get the mean and standard deviation for the generated random values
      double mean {std::accumulate(std::begin(values), std::end(values),
        0.0)/ values.size()};

      std::transform(std::begin(values), std::end(values), std::begin(values),
                     [&mean](double value) { return std::pow(value-mean,2); });
      double s_dev
      {std::sqrt(std::accumulate(std::begin(values), std::end(values), 0.0)/
        (values.size() - 1))};
      std::cout << "For generated values, mean = " << mean
                << " standard deviation = " << s_dev << std::endl;
    }
}
```

在每次循环中，都会提示为一个正态分布输入期望和标准差。循环会一直进行，直到输入 Ctrl+Z 而不是期望的值。输入的值会被用来生成 param_type 对象，它会被传给分布对象 norm 的成员函数 param()，norm 会为分布设置期望和标注差。包含 sample_count 个 double 类型元素的 vector 会被生成，并且通过 generate()算法，每一个元素会被设置为分布对象 norm 返回的随机值。然后调用 dist_plot()生成和这些值对应的分布。为了观察生成值所对应的期望和标准差与指定与原始规格之间的密切程度，会分别调用 accumulate()和 transform()算法来计算生成值所对应的期望和标准差。

对于显示的输入，得到了如下输出：

```
Enter values for the mean and standard deviation, or Ctrl+Z to end: 8 3
  -3 -|
  -2 -|
  -1 -| *
   0 -| **
   1 -| ******
   2 -| ************
   3 -| ************************
   4 -| ************************************
   5 -| ************************************************
   6 -| ************************************************************
   7 -| ********************************************************************************
   8 -| ********************************************************************************
```

```
9 -| **************************************************************************************
10 -| ************************************************************************
11 -| ********************************************************
12 -| ****************************************
13 -| ************************
14 -| ***********
15 -| *****
16 -| **
17 -| *
18 -|
19 -|
20 -|
21 -|
For generated values, mean = 8.02975 standard deviation = 2.99916

Enter values for the mean and standard deviation, or Ctrl+Z to end: ^Z
```

输出表明生成值的分布的形状是正确的，从随机值得到的期望和标准差与为分布对象指定的值非常接近。可以尝试使用不同的期望和标准差来了解形状是如何变化的。

8.4.5　对数分布

对数分布和表示随机变量的正态分布有关，这些值的对数分布是一个正态分布。对数分布是由期望和标准差定义的，但这些参数和变量无关，它们和变量的对数相关。具体来说，一个期望为 μ、标准差为 σ 的随机变量 x 的对数分布，说明 log x 是一个期望为 μ、标准差为 σ 的正态分布。图 8-6 展示了一个对数分布的曲线，以及改变期望和标准差时的效果。

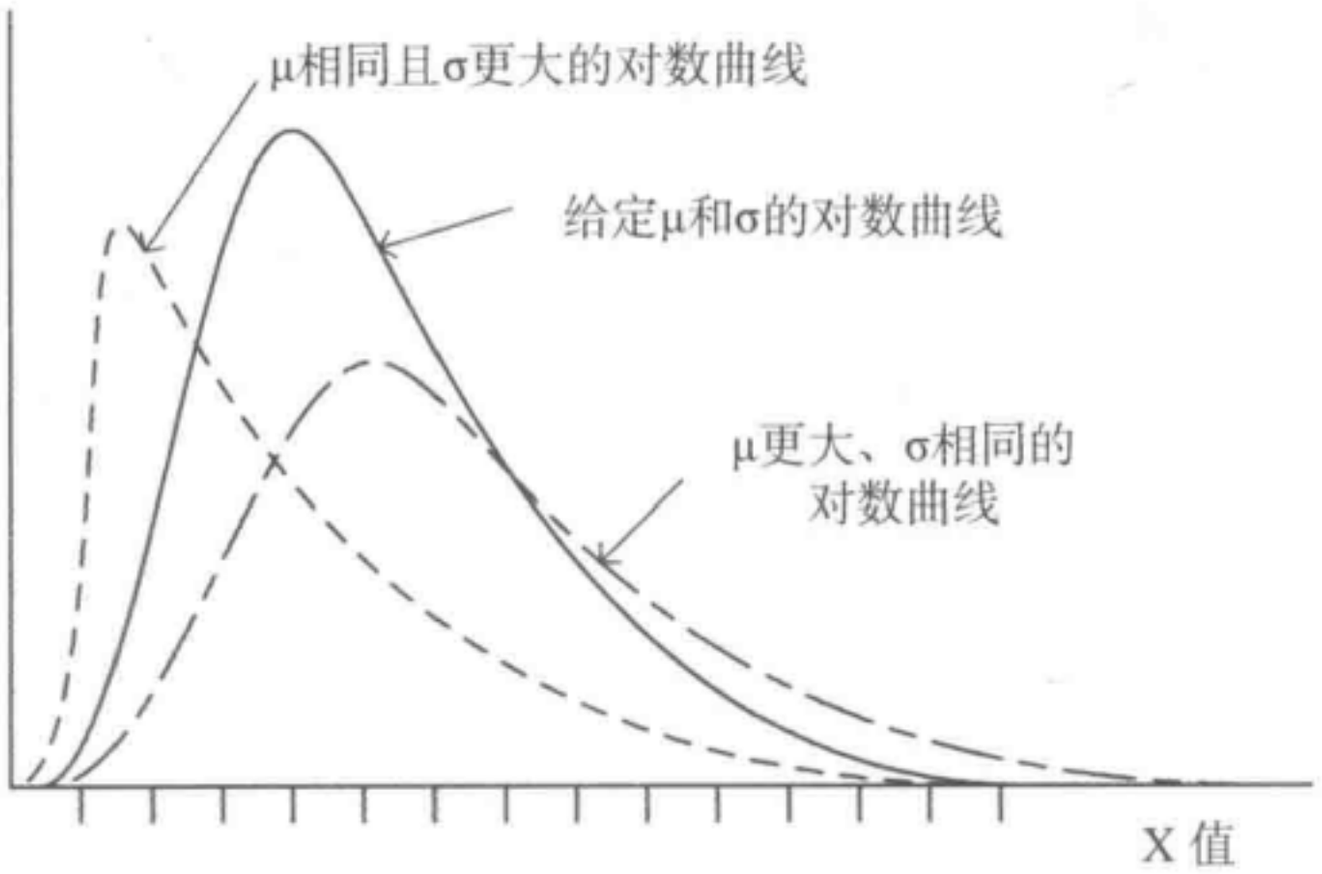


图 8-6　对数分布

对于大自然中的很多随机变量来说，对数分布比正态分布更接近概率的表示。病人的感染率就是一个对数分布模式。

lognormal_distribution 模板的一个实例定义了一个默认返回浮点类型值的对数分布对象。下面是一个期望为 5.0、标准差为 0.5 的对数分布对象的定义：

```
double mu {5.0}, sigma {0.5};
std::lognormal_distribution<> norm {mu, sigma};
```

构造函数的参数以 0 和 1 为默认值，因此省略了定义标准对数分布的参数。还有另一个构造函数，它接受封装了期望和标准差的 param_type 对象作为参数。

lognormal_distribution 对象也有所有分布类型都有的成员函数，如成员函数 m()和()，它们分别返回期望和标准差。可以像看到的其他分布那样去使用这个对象，因此让我们在示例中试试它的用法。

使用对数分布

为了不绘制不包含星号的行，这个示例会对 Ex8_05 中的函数模板 dist_plot()做一点修改。这是因为对数分布曲线会有很长的尾巴，不需要去欣赏这个分布的形状。plot_data()的最后一条语句为：

```
std::for_each(std::begin(plot_data), std::end(plot_data),
    [max_f, width](const std::pair<int, int>& v)
    { if((width*v.second)/max_f > 0)
      std::cout << std::setw(3) << v.first << " -| "
                << string((width*v.second)/max_f, '*') << std::endl;
    });
```

下面是程序代码：

```
// Ex8_06.cpp
// Checking out lognormal distributions
#include <random> // For distributions and random number generators
#include <algorithm> // For generate(), for_each(), max_element(), transform()
#include <numeric> // For accumulate()
#include <iterator> // For back_inserter()
#include <vector> // For vector container
#include <map> // For map container
#include <cmath> // For pow(), round(), log() functions
#include <iostream> // For standard streams
#include <iomanip> // For stream manipulators
#include <string>
using std::string;
using Params = std::lognormal_distribution<>::param_type;

// Modified plot_data template goes here...

int main()
{
  std::random_device rd;
  std::default_random_engine rng {rd()};
  std::lognormal_distribution<> log_norm;
  double mu {}, sigma {};
  const size_t sample_count {20000};
  std::vector<double> values(sample_count);
```

```
std::vector<double> log_values;
while(true)
{
  std::cout << "\nEnter values for the mean and standard deviation, or
               Ctrl+Z to end: ";
  if((std::cin >> mu).eof()) break;
  std::cin >> sigma;
  log_norm.param(Params {mu, sigma});
  std::generate(std::begin(values), std::end(values), [&log_norm, &rng]
    { return log_norm(rng); });

  // Create data to plot lognormal curve
  dist_plot(std::begin(values), std::end(values));

  // Create logarithms of values
  std::vector<double> log_values;
  std::transform(std::begin(values), std::end(values),
    std::back inserter (log_values),
    [] (double v){ return log(v); });

  // Create data to plot curve for logarithms of values
  std::cout << "\nThe distribution for logarithms of the values:\n";
  dist_plot(std::begin(log_values), std::end(log_values));
  // Get the mean and standard deviation - for the logarithms of the values
  double mean {std::accumulate(std::begin(log_values), std::end(log_
    values), 0.0)/log_values.size()};
  std::transform(std::begin(log_values), std::end(log_values), std::begin
      (log_values),
                 [&mean](double value) { return std::pow(value - mean, 2); });

  double s_dev {std::sqrt(std::accumulate(std::begin(log_values),
                 std::end(log_values), 0.0)/(log values.size() - 1))};
  std::cout << "For generated values, mean = " << mean
            << " standard deviation = " << s_dev << std::endl;
  }
}
```

这段代码的工作原理本质上和 Ex8_05 是一样的，while 死循环允许我们尝试各种参数。明显的区别是：它用的是 lognormal_distribution 对象。每次循环遍历中都有一个额外的图来展示生成值的对数分布。期望和标准差也和值的对数有关。

下面是一些示例输出：

```
Enter values for the mean and standard deviation, or Ctrl+Z to end: 3 .3
8 -|  *
9 -|  ******
10 -|  ***********
11 -|  ********************
12 -|  ***********************************
13 -|  ********************************************
14 -|  ************************************************************
```

```
15 -| **************************************************************
16 -| ******************************************************************************
17 -| ******************************************************************************
18 -| ******************************************************************************
19 -| ******************************************************************************
20 -| ******************************************************************************
21 -| ******************************************************************************
22 -| **************************************************************
23 -| **********************************************************
24 -| ******************************************************
25 -| ****************************************************
26 -| ********************************************
27 -| ****************************************
28 -| ************************************
29 -| ****************************
30 -| ********************
31 -| ******************
32 -| **************
33 -| ************
34 -| **********
35 -| ********
36 -| ******
37 -| ******
38 -| *****
39 -| ****
40 -| **
41 -| **
42 -| **
43 -| *
45 -| *
The distribution for logarithms of the values:
2 -| ****
3 -| ******************************************************************************
4 -| ****
For generated values, mean = 2.99837 standard deviation = 0.298659
Enter values for the mean and standard deviation, or Ctrl+Z to end: ^Z
```

可以看到对数的值是正态分布的，它是一幅标准差很小的狭窄图形。在此选择很小的标准差是为了避免对数的正态分布图的尾部过长，因为这样会占据本书太多篇幅，但建议你尝试不同的值来观察形状是如何变化的。当 S 增大时，第一幅图形会变得很长，第二幅图形会变得更像是典型的正态分布。值为 3 和 2.1 的效果应该不错。

8.4.6 其他和正态分布相关的分布

STL 定义进一步为正态分布类别的分布定义了 4 个模板，而且它们全部默认生成 double 类型的随机值。这里只对它们进行概述——它们是用和你已经看到的分布相似的方式生成和使用的。如果需要使用它们，我们已经了解它们很多了：

- chi_squared_distribution 模板为自由度的浮点参数定义的对象定义分布类型；自由度默认是 1.0。对象有一个成员函数 n()，可以返回自由度的数量。这种分布通常被广泛用来进行假设验证——检查现实世界是否和设想的表现的理论匹配。在一些学科中，现实世界和理论不匹配是如此令人失望，以至于为了和理论相符，调整测量真实世界的方式被认为是必要的。
- cauchy_distribution 模板定义了和正态分布相似的分布类型，但中位数两边的尾巴更加沉重。一个实例是由两个参数值——定义中点的中位数 a 以及确定曲线范围的分布 b 来定义的。这些可以通过调用对象的成员函数 a()和 b()来得到。柯西分布没有期望或标准差，尽管中位数标识了中点的值。
- fisher_f_distribution 模板定义了用来确定两个方差什么时候相等的分布类型，它是双卡方分布的比率。它的一个实例由两个参数值来定义的——分子分布的自由度和分母的分布的自由度。默认值都是 1.0。
- student_t_distribution 定义了用于少量样本或者当标准差未知时的分布类型的模板，只需用于指定自由度的单个参数值就可以生成实例；默认的自由度值是 1.0。

8.4.7　抽样分布

当需要定义一系列基于样本值的分布时，抽样分布是非常有用的。这些分布并没有固定的概率曲线——可能需要定义某个范围的值或一组值的范围。STL 支持 3 个抽样分布——离散分布、分段参数分布和分段线性分布。

1. 离散分布

discrete_distribution 模板定义了返回随机整数的范围在[0, n)内的分布，基于每个从 0 到 n-1 的可能值的概率权重。权重可以使我们能够决定为返回值使用何种分布。这种分布通常用返回值来选择随机对象，或从可以用索引访问的序列得到的值。序列可以包含任何类型的对象，包括函数对象，因此提供了极大的灵活性。如果想实现一个水果机模拟器，这种分布会有帮助。

必须为生成的值提供一些权重；权重的数量会决定生成的可能值的数目，而且权重的值也被用来决定概率。下面是一个示例，演示了该如何模拟投掷一个面值从 1 到 6 的骰子：

```
std::discrete_distribution<size_t> d{1, 1, 1, 1, 1, 3}; // Six possible values
std::random_device rd;
std::default_random_engine rng {rd()};
std::map<size_t, size_t> results;       // Store the results of throws
for(size_t go {}; go < 5000; ++go)      // 5000 throws of the die
  ++results[d(rng)];
for(const auto& pr : results)
  std::cout << "A " << (pr.first+1) << " was thrown " << pr.second << " times\n";
```

构造函数的初始化列表包含 6 个权值，因此分布只会生成[0,6)这个范围内的值——这意味着只包含从 1 到 5 的值。最后一个权值是其他权值的 3 倍，因此它出现的可能性是其

他权值的 3 倍。执行这段代码会生成如下内容：

```
A 1 was thrown 607 times
A 2 was thrown 645 times
A 3 was thrown 637 times
A 4 was thrown 635 times
A 5 was thrown 617 times
A 6 was thrown 1859 times
```

6 出现的可能性更大。权重是用来表示生成的整数的相对概率的浮点值。每个值的概率是它的权重除以所有权重之和，所以前 5 个值中的每一个的概率都是 1/8，最后一个值的概率是 3/8。下面这条语句也会产生同样的分布：

```
std::discrete_distribution<size_t> d{20, 20, 20, 20, 20, 60};
```

从 0 到 4，每个值的概率为 20/160，也就是 1/8，最后一个值的概率是 60/160 或 3/8。也可以用序列指定权重。下面是使用相同的随机数生成器的第一个代码段的变化版：

```
std::array<double,6> wts {10.0, 10.0, 10.0, 10.0, 10.0, 30.0};
std::discrete_distribution<size_t> d{std::begin(wts), std::end(wts)};
std::array<string, 6> die_value {"one", "two", "three", "four", "five", "six"};
std::map<size_t, size_t> results;       // Store the results of throws
for(size_t go {}; go < 5000; ++go)      // 5000 throws of the die
  ++results[d(rng)];
for(const auto& pr : results)
std::cout << "A " << die_value[pr.first] << " was thrown " << pr.second
          << " times\n";
```

这里的权值是从数组容器中得到的。分布对象生成的值被用来对数组进行索引输出。下面是得到的输出：

```
A one was thrown 653 times
A two was thrown 601 times
A three was thrown 611 times
A four was thrown 670 times
A five was thrown 600 times
A six was thrown 1865 times
```

更进一步，可以选择为 discrete distribution 对象定义权重来提供一个具有一元函数的构造函数，此构造函数可以从两个参数值中生成给定个数的权重。这种工作方式有一些复杂，因此我们会一步一步地检查它。

这个构造函数有 4 个参数：

- 权重 n 的个数
- 两个 double 类型的值——xmin 和 xmax，通常被用来计算概率
- 一元运算符 op

xmax 必须大于 xmin，如果 n 是 0，只有值为 1 的概率才会生成。因此在这种情况下，

分布总会产生相同的值—— 0。

增量会被定义为(xmax－xmin)/n，又称步进。可以通过执行表达式 op(xmin + (2*k+1)* step/2)来计算 k 从 0 到 n−1 的概率。

因此权重为：

```
op(xmin + step/2), op(xmin + 3*step/2), op(xmin + 5*step/2), ... op(xmin
  + (2*n-1)*step/2)
```

数值的示例可以帮助说明发生了什么。假设 n 是 6、xmin 是 0、xmax 是 12，因此步进值为 2。如果我们假设定义了使参数翻倍的 op，权重为 2、6、10、14、18、22，概率因此为 1/36、1/12、5/36、7/36、1/4、11/36。下面是这个分布对象的定义：

```
std::discrete_distribution<size_t> dist {6, 0, 12, [](double v) { return
  2*v; }};
```

一元运算符是由 lambda 表达式定义的，它会返回参数值的两倍。可以通过调用 discrete_distribution 对象的成员函数 probabilities()来获取概率。对于 dist 对象可以按如下方式获取概率：

```
auto probs = dist.probabilities(); // Returns type vector<double>
std::copy(std::begin(probs), std::end(probs),
      std::ostream_iterator<double> { std::cout << std::fixed << std::
        setprecision(2), " " });
std::cout << std::endl;              // Output: 0.03 0.08 0.14 0.19 0.25 0.31
```

通常，概率的个数是任意的，它和指定的权重的个数对应，因此这里返回了一个 vector<double>容器。注释中显示的输出对应于先前展示的分数值。

可以调用成员函数 param()，为有不同权重值的 discrete_distribution 对象设置新的概率；权重的个数也可以是不同的：

```
dist.param({2, 2, 2, 3, 3});      // New set of weights
auto parm = dist.param().probabilities();
std::copy(std::begin(parm), std::end(parm),
      std::ostream_iterator<double> {std::cout << std::fixed
      << std:: setprecision
       (2), " "});
std::cout << std::endl;              // Output: 0.17 0.17 0.17 0.25 0.25
```

第一次调用 param()成员函数时，它的参数是一个权重列表，这个列表中的值和原始值不同，值的个数超过之前的。调用无参数的 param()版本会返回一个 param_type 对象，但是并不能准确地知道别名代表的类型是什么。然而，我们知道它提供了和原始的分布对象相同的成员函数来访问参数。在这个示例中，意味着可以通过调用 param_type 对象的成员函数 probabilities()来得到 param_type 对象中的值。这会返回一个 vector<double>容器，然后就可以访问它。注释显示了它所包含的概率，并且可以看出它们是和新的权值对应的。

使用离散分布

可以运用 discrete_distribution 对象实现一个和电脑对战的使用 4 个骰子的游戏。骰子

的面不同于标准的骰子，并且每个筛子都不相同。骰子的面如下：

```
die 1: 3 3 3 3 3 3
die 2: 0 0 4 4 4 4
die 3: 1 1 1 5 5 5
die 4: 2 2 2 2 6 6
```

为了玩这个游戏，首先需要从 4 个骰子中任选一个，然后电脑会从剩下的 3 个中选一个。两个骰子扔 15 次，每次抛出的值最大的获胜，扔 15 次，赢的次数最多的就是获胜者。因为没有两个骰子的面值是相同的，扔一次骰子总有一方会赢，并且如果扔的次数是奇数，游戏就无法进行了。

实现这个游戏的一种简单方式是定义 Die 类来表示骰子，每个 Die 对象会保存它的 6 个面的值。这个类可以定义一个成员函数来模拟扔骰子，成员函数会使用按如下方式定义 discrete_distribution 的对象：

```
std::discrete_distribution<size_t> throw_die{1, 1, 1, 1, 1, 1};
```

然而，所有的值都是相同概率的对象并不能充分展示离散分布，因此做一些改变，让它变得更复杂一些。可以按如下方式在 Die.h 头文件中定义代表骰子的类：

```
#ifndef DIE_H
#define DIE_H
#include <random> // For discrete_distribution and random number generator
#include <vector>          // For vector container
#include <algorithm>       // For remove()
#include <iterator>        // For iterators and begin() and end()

// Alias for param_type for discrete distribution
using Params = std::discrete_distribution<size_t>::param_type;

std::default_random_engine& rng();

// Class to represent a die with six faces with arbitrary values
// Face values do not need to be unique.
class Die
{
public:
  Die() { values.push_back(0); };

  // Constructor
  Die(std::initializer_list<size_t> init)
  {
    std::vector<size_t> faces {init}; // Stores die face values
    auto iter = std::begin(faces);
    auto end_iter = std::end(faces);
    std::vector<size_t> wts; // Stores weights for face values
    while(iter != end_iter)
    {
      values.push_back(*iter);
```

```
      wts.push_back(std::count(iter, end_iter, *iter));
      end_iter = std::remove(iter, end_iter, *iter++);
    }
    dist.param(Params {std::begin(wts), std::end(wts)});
  }
size_t throw_it() { return values[dist(rng())]; }
private:
  std::discrete_distribution<size_t> dist; // Probability distributtion
                                           // for values
  std::vector<size_t> values; // Face values
};
#endif
```

这个类有两个私有成员，用来为骰子生成随机面值的 discrete_distribution<size_t>对象 dist，以及保存唯一面值的 vector 容器 values。默认的构造函数会用默认的分布对象来生成对象，默认的分布对象总是返回 0，并有一个包含单个 0 值的成员变量 values。它的第二个构造函数需要一个指定骰子面值的初始化列表。初始化列表会被用来初始化带有面值的局部容器 faces。面值可以是重复的，每个面值的权重——决定它们的概率——是面值出现的次数。构造函数的 while 循环会遍历 faces 当前范围内的元素，并计算出第一个元素的面值出现的次数。次数会被保存到 wts 容器中，并且它所对应的面值会被添加到成员变量 values 中。然后会从 faces 中移除当前面值的所有匹配项，这个删除操作所产生的结束迭代器会被保存到 end_iter 中。当循环结束时，values 会包含骰子的全部唯一面值，wts 会包含对应的权重。调用 dist 的成员函数 param()，将分布的参数设为 wts 容器中的权重。

用 Die 类实现的这个游戏的程序代码如下所示：

```
// Ex8_07.cpp
// Implementing a dice throwing game using discrete distributions
#include <random>       // For discrete_distribution and random number generator
#include <array>        // For array container
#include <utility>      // For pair type
#include <algorithm> // For count(), remove()
#include <iostream>   // For standard streams
#include <iomanip>    // For stream manipulators
#include "Die.h"      // Class to define a die

// Random number generator available throughout the program code
std::default_random_engine& rng()
{
  static std::default_random_engine engine {std::random_device()()};
  return engine;
}

int main()
{
  size_t n_games {};                  // Number of games played
  const size_t n_dice {4};            // Number of dice
  std::array<Die, n_dice> dice        // The dice
```

```
    {
      Die {3, 3, 3, 3, 3, 3},
      Die {0, 0, 4, 4, 4, 4},
      Die {1, 1, 1, 5, 5, 5},
      Die {2, 2, 2, 2, 6, 6}
    };
  std::cout <<"For each game, select a die from the following by entering
              1, 2, 3, or  4 (or Ctrl+Z to end):\n"
            << "die 1: 3 3 3 3 3 3\n"
            << "die 2: 0 0 4 4 4 4\n"
            << "die 3: 1 1 1 5 5 5\n"
            << "die 4: 2 2 2 2 6 6\n";

  size_t you {}, me {};              // Stores index of my dice and your dice

  while(true)
  {
    std:: cout << "\nChoose a die: ";
    if((std::cin >> you).eof()) break; // For EOF - it's all over

    if(you == 0 || you > n_dice) // Only 1 to 4 allowed
    {
      std::cout << "Selection must be from 1 to 4, try again.\n";
      continue;
    }

    // Choose my die as next in sequence
    me = you-- % n_dice; // you from 0 to 3, and me you+1 mod 4
    std::cout << "I'll choose: " << (me+1) << std::endl;

    // Throw the dice
    const size_t n_throws {15};
    std::array<std::pair<size_t, size_t>, n_throws> goes; // Results of goes -
                                                          // pair<me_value, you_value>
    std::generate(std::begin(goes), std::end(goes),    // Make the throws
      [&dice, me, you] { return std::make_pair(dice[me].throw_it(), dice[you].
        throw_it()); });

    // Output result of game
    std::cout << "\nGame " << ++n_games << ":\n";

    // Output results of my throws...
    std::cout << "Me : ";
    std::for_each(std::begin(goes), std::end(goes),
      [](const std::pair<size_t, size_t>& pr)
      {std::cout << std::setw(3) << pr.first; });
      auto my_wins = std::count_if(std::begin(goes), std::end(goes),
                                   [](const std::pair<size_t, size_ t>& pr)
                                   {return pr.first > pr.second;});
    std::cout << " My wins: " << std::setw(2) << std::right << my_wins
              << " I " << ((my_wins > n_throws / 2) ? "win!!" : "lose {:-(")
              << std::endl;
```

```
    // Output results of your corresponding throws - aligned below mine...
    std::cout << "You: ";
    std::for_each(std::begin(goes), std::end(goes), [](const std::pair<size_t,
        size_t>& pr){ std::cout << std::setw(3) <<  pr.second; });
    std::cout << " Your wins: " << std::setw(2) << std:: right << n_throws - my_wins
              << " You " << ((my_wins <= n_throws / 2) ? "win!!" : "lose!!!")
              << std::endl;
  }
}
```

dice 数组容器中保存了 4 个不同的骰子。在提示输入后，游戏就在 while 循环中开始了，每次迭代都是一场完整的游戏。玩家选择的骰子会被保存在 you 中，电脑选择的骰子保存在 me 中。可以从 1 到 4 选择一个数字，因此它的减量可以用来作为 dice 数组的索引。变量 me 会被任意设为序列的下一个骰子，取 4 的模。因此，如果 you 选择的是索引为 3 的最后一个骰子，me 会选择索引为 0 的第一个骰子。

变量 n_throws 指定了游戏中扔骰子的次数，在这个示例中是 15 次；游戏进行奇数次可以保证总有获胜者。generate()算法可以在游戏中将两个骰子扔 15 次，并将每次扔骰子的结果保存到 pair 对象中，pair 对象的成员变量 first 保存的是电脑的骰子值，second 成员变量保存的是玩家的骰子值。扔两个骰子的操作是由作为 generate()的第三个参数的 lambda 表达式执行的。扔骰子的结果是通过调用它的成员函数 throw_it()得到的。

计算机赢的次数是由 count_if()算法计算出来的，并被保存到 my_wins 中。如果 goes 容器中每个 pair 对象的成员 first 大于 second，计数会增加。玩家赢的次数保存在 n_throws-my_wins 中。

笔者得到的输出如下：

```
For each game, select a die from the following by entering 1, 2, 3, or 4 (or Ctrl+Z to end):
die 1: 3 3 3 3 3 3
die 2: 0 0 4 4 4 4
die 3: 1 1 1 5 5 5
die 4: 2 2 2 2 6 6

Choose a die: 2
I'll choose: 3

Game 1:
Me : 5 1 5 1 5 5 1 1 5 5 5 5 1 5 5 My wins: 11 I win!!
You: 4 0 4 4 4 4 4 4 4 4 4 0 4 4 4 Your wins: 4 You lose!!!

Choose a die: 4
I'll choose: 1

Game 2:
Me : 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 My wins: 9 I win!!
You: 6 6 2 2 2 2 2 2 6 6 2 2 2 6 6 Your wins: 6 You lose!!!

Choose a die: 1
I'll choose: 2
```

```
Game 3:
Me : 0 0 0 0 4 4 4 4 4 0 4 4 4 4 4 My wins: 10 I win!!
You: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 Your wins: 5 You lose!!!

Choose a die: 3
I'll choose: 4

Game 4:
Me : 6 2 2 2 2 6 2 2 2 2 6 2 2 2 2 My wins: 9 I win!!
You: 5 5 5 5 1 5 5 5 1 1 5 1 1 1 5 Your wins: 6 You lose!!!

Choose a die: 3
I'll choose: 4

Game 5:
Me : 2 2 6 2 2 2 6 2 2 2 2 6 2 6 6 My wins: 12 I win!!
You: 5 5 5 1 1 5 5 1 1 1 1 1 1 1 5 Your wins: 3 You lose!!!

Choose a die: 3
I'll choose: 4

Game 6:
Me : 6 2 2 2 2 2 6 2 2 2 2 6 6 2 2 My wins: 8 I win!!
You: 5 5 5 5 5 5 1 1 1 1 5 5 1 5 1 Your wins: 7 You lose!!!

Choose a die: 2
I'll choose: 3

Game 7:
Me : 5 5 1 1 5 5 5 5 5 5 5 1 1 1 1 My wins: 10 I win!!
You: 4 4 4 4 0 4 4 4 0 4 4 4 4 0 4 Your wins: 5 You lose!!!

Choose a die: ^Z
```

电脑相当成功——它赢了每次比赛。它的成功是因为玩家先选。这 4 个非传递性的骰子示例是由美国的一个叫作布拉德利·埃夫隆的统计学家发明的。一般来说，数值关系是可以传递的，这意味着如果 a>b、b>c，那么可以肯定地说 a>c。但在这些骰子中，并不是这样。对于 dice 数组中的 Die 对象，由 Ex8_07 实现的这个游戏会有下面这些情况：

- dice[3]赢了 dice[2]，dice[2]赢了 dice[1]，dice[1]赢了 dice[0]，dice[0]赢了 dice[3]！这是因为面值的概率来自于骰子的一次抛掷
- dice[3]赢了 dice[2]，因为 dice[3]中的 6(概率为 1/3)总会赢，dice[3]中的 2(概率为 2/3)在 dice[2]是 1(概率为 1/2)时会赢，因此 dice[3]赢的总概率为 1/3 + 2/3×1/2，是 2/3。
- dice[2 赢了 dice[1]，因为 dice[2]中的 5(概率为 1/2)总可以赢，dice[2]的 1(概率为 1/2)在 dice[1]是 0 (概率为 1/3)时会赢，因此 dice[2]赢的总概率为 1/2 + 1/2×1/3，是 2/3。
- dice[1]赢了 dice[0]，因为 dice[1]中 4(概率为 2/3)总会赢。
- dice[0]赢了 dice[3]，因为 dice[0]中 3(概率为 1)在 dice[3]是 2(概率为 2/3)时会赢，

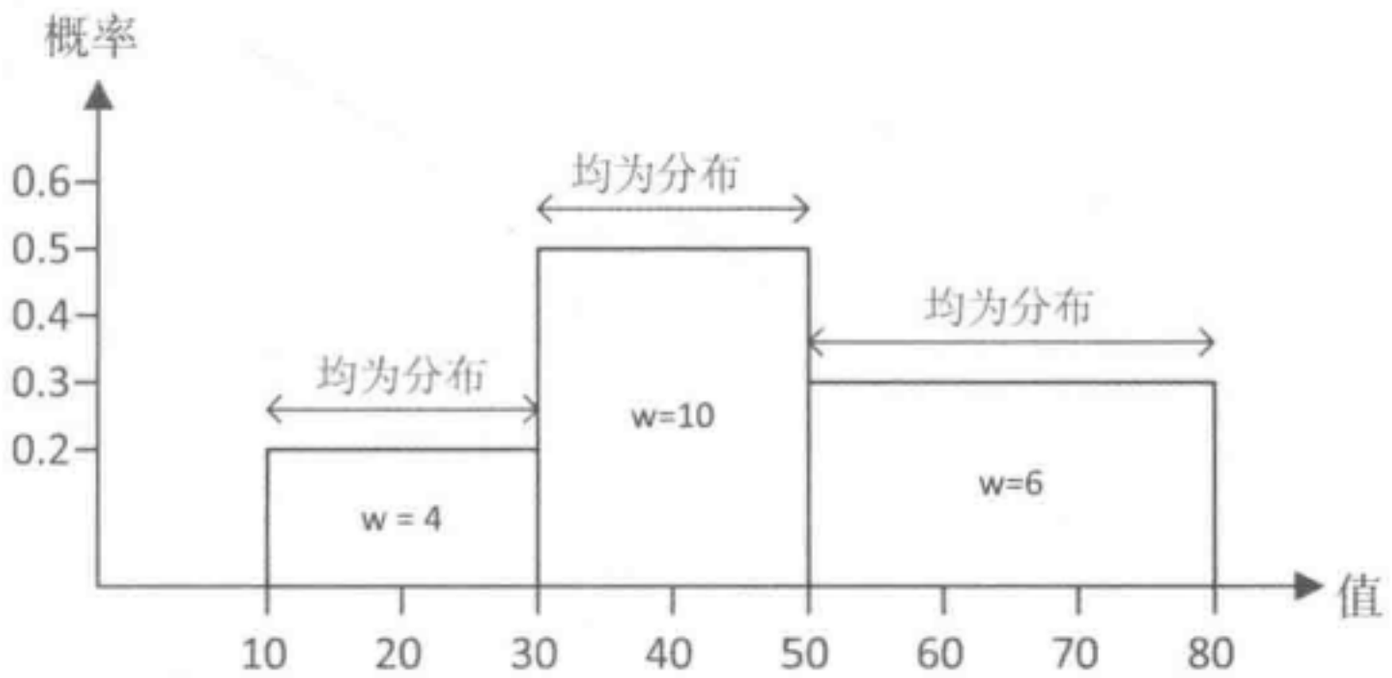
因此 dice[0]赢的总概率为 1×2/3，是 2/3。

这意味着无论我们选什么，电脑都可以从剩下的能赢我们的概率为 2/3 的 3 个骰子中选一个。

2. 分段常数分布

piecewise_constant_distribution 模板定义了一个在一组分段子区间生成浮点值的分布。给定子区间内的值是均匀分布的，每个子区间都有自己的权重。n 个区间边界可以定义 n−1 个子区间和 n−1 个可以运用到子区间的权重，由这 n 个区间边界定义一个对象，图 8-7 说明了这一点。

图 8-7 中的分布定义了 3 个区间，每个都有自己的权重。这 3 个区间是由定义在容器 b 中的 4 个边界值定义的。每个区间都有一个由容器 w 中的元素定义的权重。它的前两个参数是指定边界序列的迭代器，第三个参数是指向权重序列的第一个元素的迭代器。每个区间内的值都是均匀分布的，特定区间内的随机值的概率是由这个区间的权重决定的。




```
std::vector<double> b {10, 30, 50, 80};  // Boundaries
std::vector<double> w {4, 10, 6};        // Weights
std::piecewise_constant_distribution<> d {std::begin(b), std::end(b), std::begin(w)};
```

图 8-7　分段常数分布

除了所有分布都实现的成员函数之外，piecewise_constant_distribution 还有成员函数 intervals()和 densities()，它们分别返回区间的边界和区间内值的概率密度；这两个函数返回的值都是 vector 容器。通过尝试和图 8-7 所示的相似的分布，我们可以运用这些成员，并深入了解这个分布的效果。但区间很狭窄，因此输出需要的空间很少：

```
// Ex8_08.cpp
// Demonstrating the piecewise constant distribution
#include <random>       // For distributions and random number generator
#include <vector>       // For vector container
#include <map>          // For map container
#include <utility>      // For pair type
#include <algorithm>    // For copy(), count(), remove()
#include <iostream>     // For standard streams
#include <iterator>     // For stream iterators
#include <iomanip>      // For stream manipulators
```

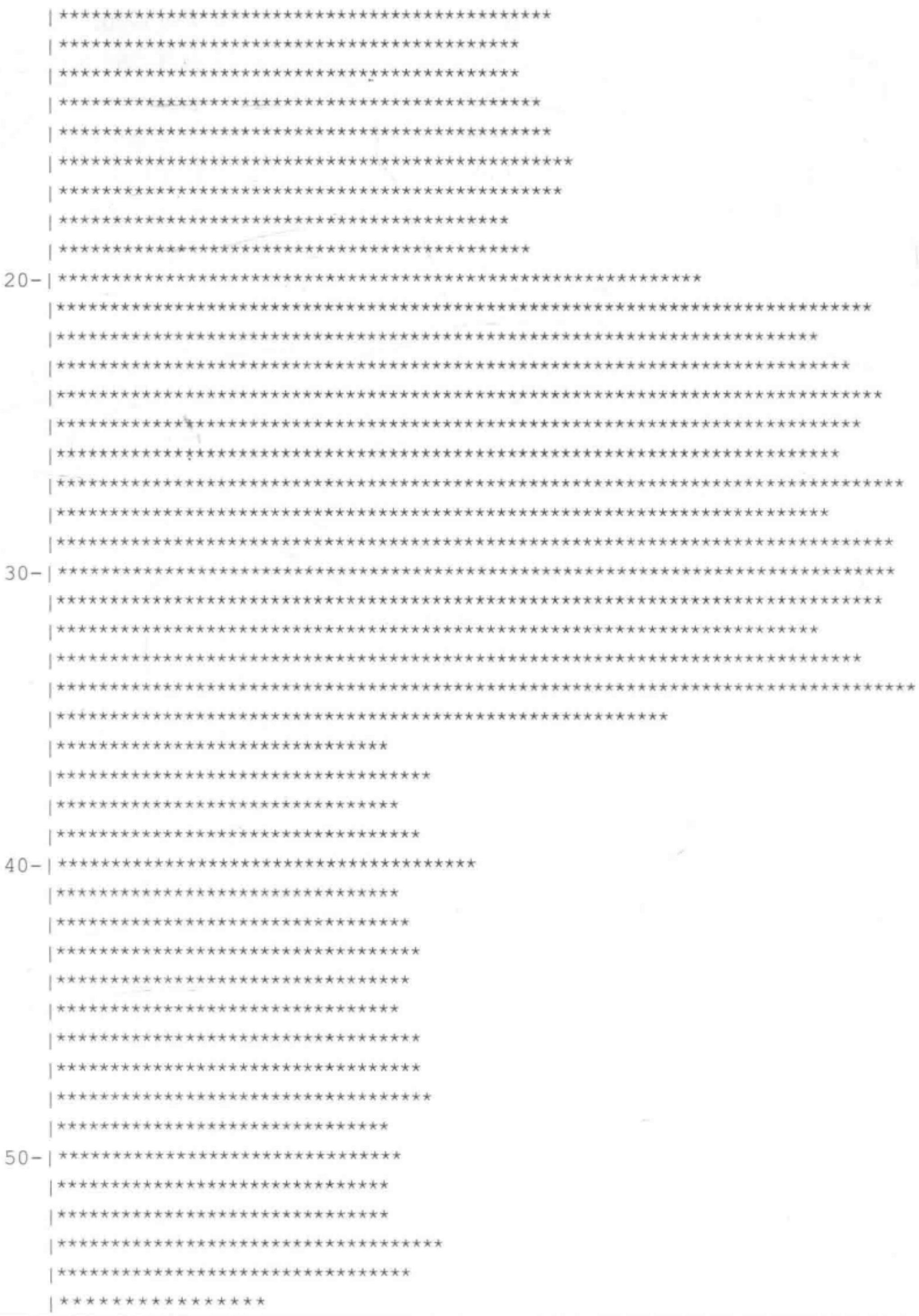
```
#include <string>       // For string class
using std::string;

int main()
{
  std::vector<double> b {10, 20, 35, 55}; // Intervals: 10-20, 20-35, 35-55
  std::vector<double> w {4, 10, 6}; // Weights for the intervals
  std::piecewise_constant_distribution<> d {std::begin(b), std::end(b),
    std::begin(w)};
  // Output the interval boundaries and the interval probabilities
  auto intvls = d.intervals();
  std::cout << "intervals: ";
  std::copy(std::begin(intvls), std::end(intvls), std::ostream iterator
    <double>{std::cout, " "});
  std::cout << " probabilities: ";
  auto probs = d.densities();
  std::copy(std::begin(probs), std::end(probs), std::ostream_iterator <double>
    {std::cout, " "});
  std::cout << '\n' << std::endl;

  std::random_device rd;
  std::default_random_engine rng {rd()};
  std::map<int, size_t> results; //Stores and counts random values as integers
  // Generate a lot of random values...
  for(size_t i {}; i < 20000; ++i)
    ++results[static_cast<int>(std::round(d(rng)))];
  // Plot the integer values
  auto max_count = std::max_element(std::begin(results), std::end(results),
                   [](const std::pair<int, size_t>& pr1, const std::pair<int,
                   size_t>& pr2){ return pr1.second < pr2.second; })->second;
std::for_each(std::begin(results), std::end(results),
                                   [max_count](const std::pair<int, size_t>& pr)
                                   { if(!(pr.first % 10)) // Display value if
                                                          // multiple of 10
                                     std::cout << std::setw(3) << pr.first
                                                               << "-|";
                                     else
                                       std::cout << " |";
                                     std::cout << std::string(pr.second * 80
                                                              / max_count, '*')
                                               << '\n'; });
}
```

这样就生成了一个我们之前看到的区间和权重的分布，并用这个分布生成了大量的值，然后在将它们转换为整数后，将这些值的出现频率绘制成直方图。值会在页的下面运行，条形图从左到右地表示相对频率。笔者得到的输出如下：

```
intervals: 10 20 35 55 probability densities: 0.02 0.0333333 0.015
10-|**************************
```



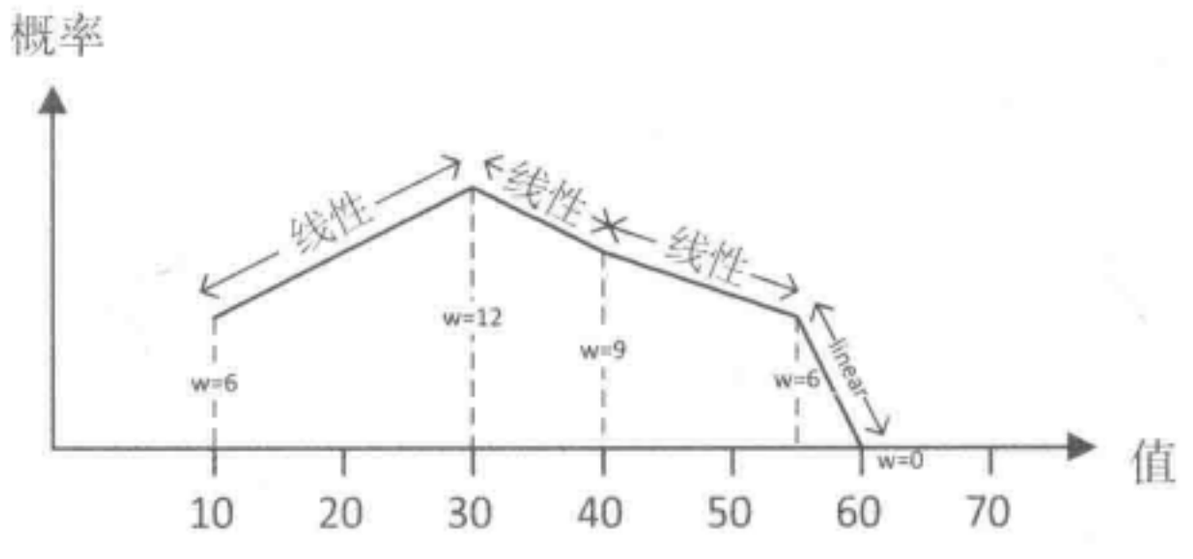
输出中有趣的地方是概率密度的值，以及第一个和最后一个区间内条形图的相对长度。这两个区间的权重分别为 4 和 6，因此值在第一个区间的概率是 4/20，也就是 0.2；值

在第二个区间的概率为 10/20，也就是 0.5；值在最后一个区间的概率是 6/20，也就是 0.3。然而，最后一个区间输出的条形图低于第一个区间，这似乎和概率有些矛盾。无论如何，输出中的概率密度都是不同的，为什么会这样？

原因在于它们是不同的。概率密度是区间内给定值出现的概率，而不是随机值出现在区间内的概率，一个值的概率密度与区间值出现概率除以区间的值的范围是对应的。因此，这个 3 个区间内值的概率密度分别为 0.2/10、0.5/15、0.3/20，幸运的是，这和输出是相同的。最后一个区间得到的值恰好是第一个区间的两倍，但它所跨越的范围更大，因此条形图更短。因此，条形图的长度反映了概率密度。

3. 分段线性分布

piecewise_linear_distribution 模板定义了浮点值的连续分布，它的概率密度函数是从一系列的样本值所定义的点得到的。每个样本值的权重都决定了它的概率密度值。图 8-8 展示了一个示例。




```
std::vector<double> v {10, 30, 40, 55, 60};  // Values
std::vector<double> w {6, 12, 9, 6, 0};      // Weights for values
std::piecewise_linear_distribution<> d {std::begin(v), std::end(v), std::begin(w)};
```

图 8-8　分段线性分布

图 8-8 展示了由 5 个定义在容器 v 中的样本值确定的分布。每个值都有权重，权重是由 w 容器中对应的元素定义的，每个权重确定了对应值的概率密度。

一个样本和另一个样本之间的概率密度值是线性的，在这两个样本的概率密度之间。构造函数的前两个参数是指定值序列的迭代器，第三个参数指向 weights 序列中第一个元素的迭代器。这个以分段线性曲线表示概率密度的分布会生成从 10 到 60 的随机值。分布中的样本值可以通过调用它的成员函数 intervals()来获取。可以通过调用这个分布对象的 densities()来得到一个包含这些概率密度的 vector 容器。

确定整个序列中值的概率密度有些复杂。整个概率曲线下的面积表示的是整个序列中出现的任何值的概率，因此必须是 1。为了适应这种情况，区间内值的概率可以按如下方式来计算：

首先计算出定义区间的权重值的平均数乘以区间的长度之和。因此，s 可以由下面这个等式定义：

$$s=\sum_{0}^{n-1}(v_{i+1}-v_i)\frac{(w_{i+1}+w_i)}{2}$$

v_i 是样本值，w_i 是它们对应的权重。

两个样本值之间的区间内任意值 x 的概率是 p，$[v_i, v_{i+1})$是由样本值概率的线性组合决定的，每个区间末尾所共享的概率和 x 到样本值的距离成正比。下面用数学术语来表示 x 的概率：

$$p = \frac{w_i(v_{i+1} - x) + w_{i+1}(x - v_i)}{s(v_{i+1} - v_i)}$$

在图 8-8 所示的示例中，s 是：

(30−10)×(12+6)/2+(40−30)×(9+12)/2+(55−40)×(6+9)/2+(60−55)×(0+6)/2

等于 412.5。

第 i 个样本值的概率为 w_i/s，因此图 8-8 中的概率值为 6/412.5、12/412.5、9/412.5、6/412.5 和 0/412.5。计算器指出它们分别对应于 0.0145、0.029、0.0218、0.0145 和 0。这里有一个和 Ex8_08 相似的示例，可以展示这是否正确，而且可以展示分段线性分布的全部特性：

```
// Ex8_09.cpp
// Demonstrating the piecewise linear distribution
#include <random>       // For distributions and random number generator
#include <vector>       // For vector container
#include <map>          // For map container
#include <utility>      // For pair type
#include <algorithm>    // For copy(), count(), remove()
#include <iostream>     // For standard streams
#include <iterator>     // For stream iterators
#include <iomanip>      // For stream manipulators
#include <string>       // For string class
using std::string;

int main()
{
std::vector<double> v {10, 30, 40, 55, 60};  // Sample values
std::vector<double> w {6, 12, 9, 6, 0};      // Weights for the samples
std::piecewise_linear_distribution<> d {std::begin(v), std::end(v),
   std::begin(w)};

// Output the interval boundaries and the interval probabilities
auto values = d.intervals();
std::cout << "Sample values: ";
std::copy(std::begin(values), std::end(values),
   std::ostream_iterator<double>{std::cout, " "});
std::cout << " probability densities: ";
auto probs = d.densities();
std::copy(std::begin(probs), std::end(probs),
   std::ostream_iterator<double>{std::cout, " "});
std::cout << '\n' << std::endl;

std::random_device rd;
std::default_random_engine rng {rd()};
std::map<int, size_t> results; // Stores and counts random values as integers
```

```
    // Generate a lot of random values...
    for(size_t i {}; i < 20000; ++i)
      ++results[static_cast<int>(std::round(d(rng)))];

    // Plot the integer values
    auto max_count = std::max_element(std::begin(results), std::end(results),
      [](const std::pair<int, size_t>& pr1, const std::pair<int, size_t>& pr2)
      { return pr1.second < pr2.second; })->second;
    std::for_each(std::begin(results), std::end(results),
                  [max_count](const std::pair<int, size_t>& pr)
                  {
                    if(!(pr.first % 10)) // Display value if multiple of 10
                      std::cout << std::setw(3) << pr.first << "-|";
                    else
                      std::cout << "  |";
                    std::cout << std::string(pr.second * 80 / max_count, '*')
                              << '\n';
                  });
  }
```

它和 Ex8_08 唯一不同的是分布对象的定义。对于分段线性分布，必须有和样本值同样数量的权重。下面是从这个程度得到的输出：

```
Sample values: 10 30 40 55 60 probability densities: 0.0145455 0.0290909 0.0218182
0.0145455 0
10-|********************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
20-|****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
  |****************************************
30-|****************************************
  |****************************************
  |****************************************
  |****************************************
```

这和用计算器计算出的概率密度非常相似，这使笔者很欣慰。这个分布为定义任何形状的概率密度函数提供了一个强有力的工具。

8.4.8　其他分布

STL 还为另外 9 种类型的分布定义了模板。在此只对它们进行概述以供参考，并展示其中的绝大数分布所生成值的绘图示例。需要记住的是，分布曲线的形状差别很大，都有不同的参数集，并且显示的插图为了适应页面的宽度都进行了缩放。

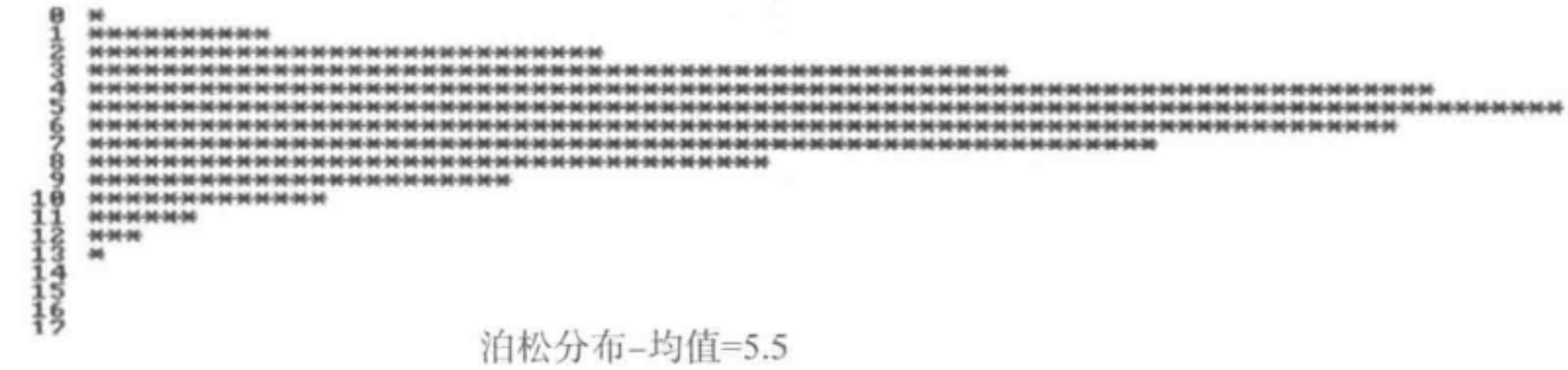
1. 泊松分布

泊松分布定义了表示一组独立事件发生频率的离散 PDF，每一个都有两种可能的结果，而且不同事件的成功概率可以是不同的。事件可以按事件或地点分配。这个分布起源于著名的法国数学家西缅·泊松。它的首次运用是对普鲁士军队士兵被马踢死的概率进行建模。它也可以用来对随着时间的推移，设备发生故障的可能性或交通事故的发生率进行建模。

poisson_distribution 模板定义了可以生成随机非负数的函数对象类型，随机非负数模式是 int 类型。可以用浮点值的平均值来生成这种分布的对象，例如：

```
double mean {5.5};
std::poisson_distribution<> poisson_d {mean};
```

默认构造函数分配的默认平均值为 1.0，对象有一个可以返回平均值的成员函数 mean()。对于上面定义的 poisson_d 生成的随机值，出现次数的绘图效果如图 8-9 所示。



泊松分布–均值=5.5

几何分布–p=0.4

指数分布–lambda=0.75

图 8-9　泊松分布、几何分布、指数分布示例

2. 几何分布

几何分布是一个离散分布，它被用来模拟取得某种有两个可能结果的事件所需要的实验次数。需要随机选择多少人来问，才能发现一个买了本书的人，这就是一个运用这种分布建模的示例。geometric_distribution 模板定义了返回非负整数值的离散分布类型，返回的非负整数值默认是 int 类型。构造函数的参数是一个指定一次实现取得成功的概率的 double 值。例如：

```
double p_success {0.4};
std::geometric_distribution<> geometric_d {p_success};
```

这个对象有一个成员函数 p()，它会返回成功的概率。geometric_d distribution 的绘图效果如 8-9 所示。

3. 指数分布

指数分布可以模拟事件发生之间的时间。可以把它想象成一个连续等值的几何分布。

exponential_distribution 模板定义了返回浮点值的分布类型，浮点值默认是 double 类型。构造函数的参数是一个表示到达率的 double 值，通常被标识为 lambda。例如：

```
double lambda {0.75};
std::exponential_distribution<> exp_d {lambda};
```

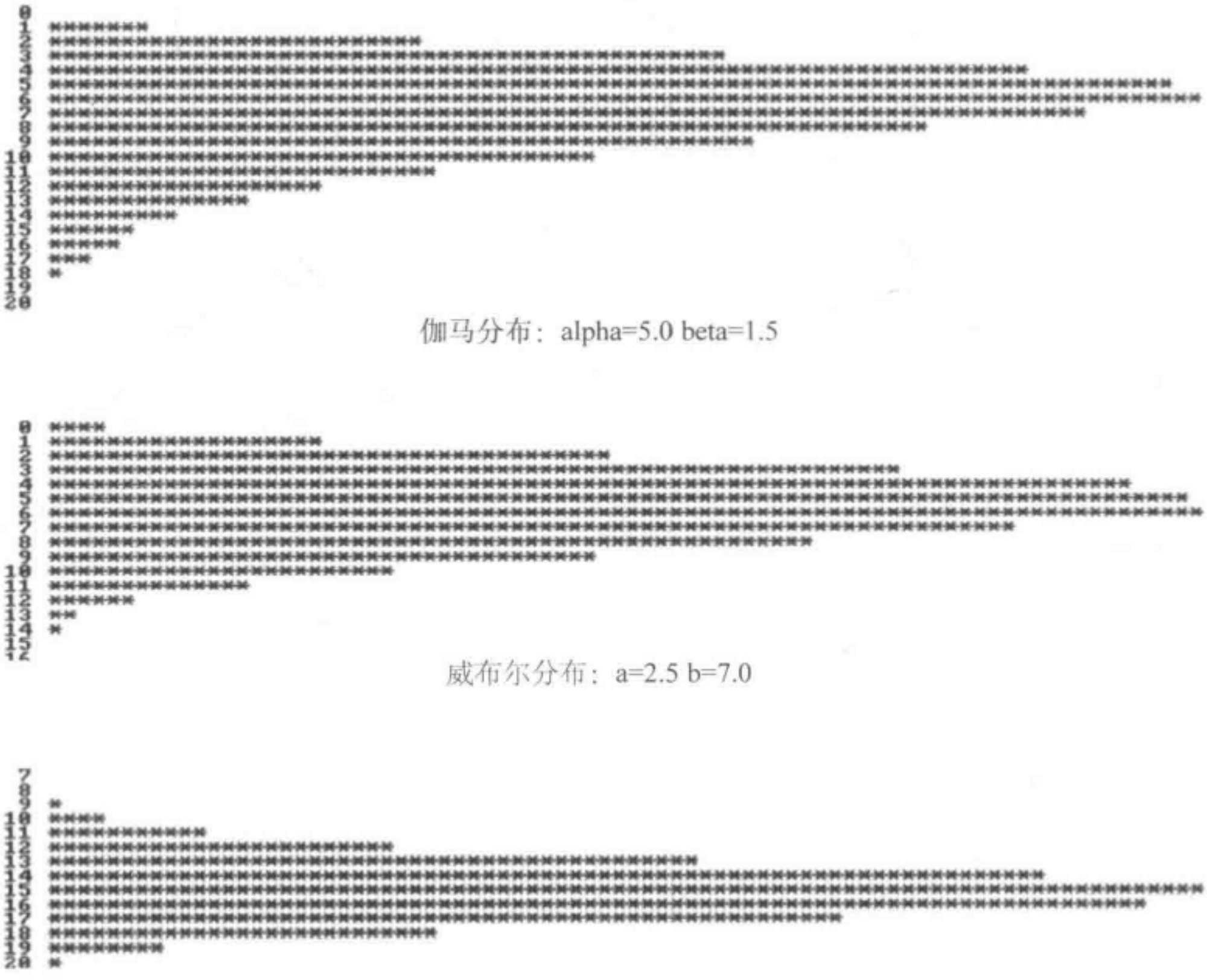
这个对象的成员函数 lambda()返回的是到达率的值，这个分布的绘图效果如图 8-9 所示。

4. 伽马分布

伽马分布是连续分布，通常用来模拟等待一个事件的时间，但比指数分布更常见。这个分布由两个参数定义：形状值 alpha 和速率值 beta。gamma_distribution 模板定义了返回浮点值的分布对象，浮点值默认是 double 类型。例如：

```
double alpha {5.0}, beta {1.5};
std::gamma_distribution<> gamma_d {alpha, beta};
```

成员函数 alpha()和 beta()会返回分布对象的参数。gamma_d 生成值的绘图效果如图 8-10 所示：



伽马分布：alpha=5.0 beta=1.5

威布尔分布：a=2.5 b=7.0

二项式分布：t=20 p=0.75

图 8-10　伽马分布、威布尔分布、二项式分布

5. 威布尔分布

威布尔分布定义了一个连续的 PDF，可以用时间函数来模拟失败率——通常是材料。它由两个参数定义，形状 a 和规模 b。下面是一个用 weibull_distribution 模板创建实例的示例：

```
double a {2.5};          // Shape
double b {7.0};          // Scale
std::weibull_distribution<> weibull_d {a, b};
```

weibull_d 对象默认会返回 double 类型的随机值。可以通过调用它的成员函数 a()和 b()来获取分布参数。weibull_d 生成值的绘图效果如图 8-10 所示。

6. 二项式分布

二项式分布是一个离散分布，它可以模型化一系列独立的二元事件。每一个事件只有两种结果——成功或失败，所有事件的成功率相同。它是由两个参数定义的——t 和 p，t 是实验的次数，p 是一次实现中成功的概率。下面展示了如何用 binomial_distribution 模板生成一个对象：

```
int t {20};                  // Number of trials
double p {0.75};             // Probability of success
std::binomial_distribution<> binomial_d {t, p};
```

这个对象的成员函数 t()和 p()会返回这些参数值。binomial_d 生成的值的图形如图 8-10 所示。

伯努利分布是一个 t 参数为 1 的二项式分布。STL 提供的 bernoulli_distribution 类定义了这种分布。因为 t 被固定为 1，那么就只需要像构造函数一样提供一个 p 值，并且这个对象会返回一个随机的布尔值。成员函数 p()可以返回成功的概率。下面是一个展示如何生成和使用这种对象的代码段：

```
std::random_device rd;
std::default_random_engine rng {rd()};
double p {0.75};                        // Probability of success
std::bernoulli_distribution bernoulli_d {p};
std::cout << std::boolalpha;            // Output bool as true or false
for(size_t i {}; i < 15; ++i)
  std::cout << bernoulli_d(rng) << ' ';
std::cout << std::endl;
```

执行这段代码后，得到的输出如下：

```
true true false true true true true true false true false true true false true
```

7. 负二项式分布

负二项式分布是一个离散分布，它可以模型化试验序列中指定的成功次数之前的失败

次数。试验只有两个可能的结果，它们彼此是相互独立的。如果成功的次数是 1，这个分布就成了几何分布。也可以将这个分布看作给定成功次数之前的失败次数的模型。negative_binomial_distribution 模板定义了一个默认返回 int 型整数的对象类型。negative_binomial_distribution 模板的构造函数需要两个参数：失败次数 k 和成功的概率 p。下面是生成一个对象的示例：

```
int k {5};                  // Number of successes
double p {0.4};             // Probability of success
std::negative_binomial_distribution<> neg_bi_d {k, p};
```

neg_bi_d 的成员函数 k()和 p()可以返回参数的值，neg_bi_d 生成的值的图形如图 8-11 所示。

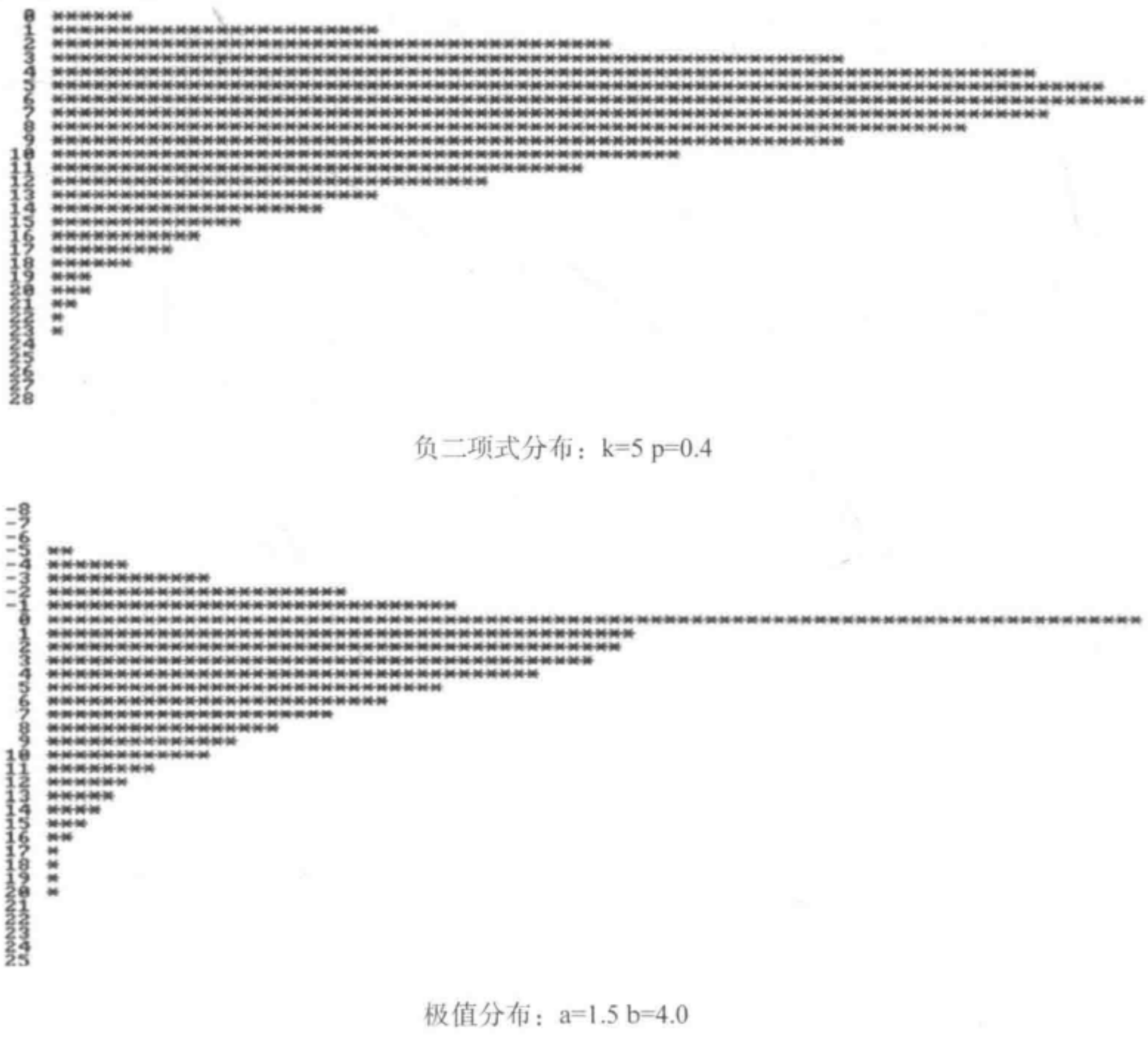


图 8-11　负二项式分布和极值分布

8. 极值分布

极值分布是一个连续分布，它被用来模型化以相同方式分配的独立变量序列的最大值或最小值的分布。它的应用之一是用来模拟自然界的极端现象，例如降雨或地震。extreme_value_distribution 模板定义了返回浮点值的对象类型，浮点值默认是 double 类型。构造函

数需要两个参数：参数的位置 a 和参数的规模 b，两个参数都是浮点值。例如：

```
double a {1.5};          // location
double b {4.0};          // Scale
std::extreme_value_distribution<> extreme_value_d {a, b};
```

参数值可以通过调用对象的成员函数 a()和 b()来获取。extreme_value_d 对象生成的值的图形如图 8-11 所示。

8.5 随机数生成引擎和生成器

STL 中有 3 个随机数引擎的类模板。它们中的每一个都实现了用来生成随机数序列的著名且高效的算法，但它们都有不同的优缺点。这里的 3 个模板是 STL 提供的所有 10 个标准的随机数生成器类类型的基础。除了默认的 default_random_engine 生成器类型(这是实现的定义)之外，还有 9 个更进一步的生成器类类型，可以自定义引擎来实现生成随机序列的已知的可靠算法。有 3 个随机数引擎适配器可以自定义从一个引擎得到的序列。它们中的每一个都有一个指定运用哪个引擎的模板参数。这 3 个引擎适配器模板是：

- independent_bits_engine 适配器模板会将引擎生成的值修改为指定的比特个数。
- discard_block_engine 适配器模板会将引擎生成的值修改为丢弃给定长度的值序列中的一些元素。
- shuffle_order_engine 适配器模板会将引擎生成的值返回到不同的序列中。通过保存从引擎得到的给定长度的值序列来做到这些，然后在随机序列中返回它们。

生成器类既可以直接用一套指定的模板参数值自定义一个引擎模板，也可以用一个随机数引擎适配器来自定义另一个生成器。引擎产生生成器的方式如图 8-12 所示。

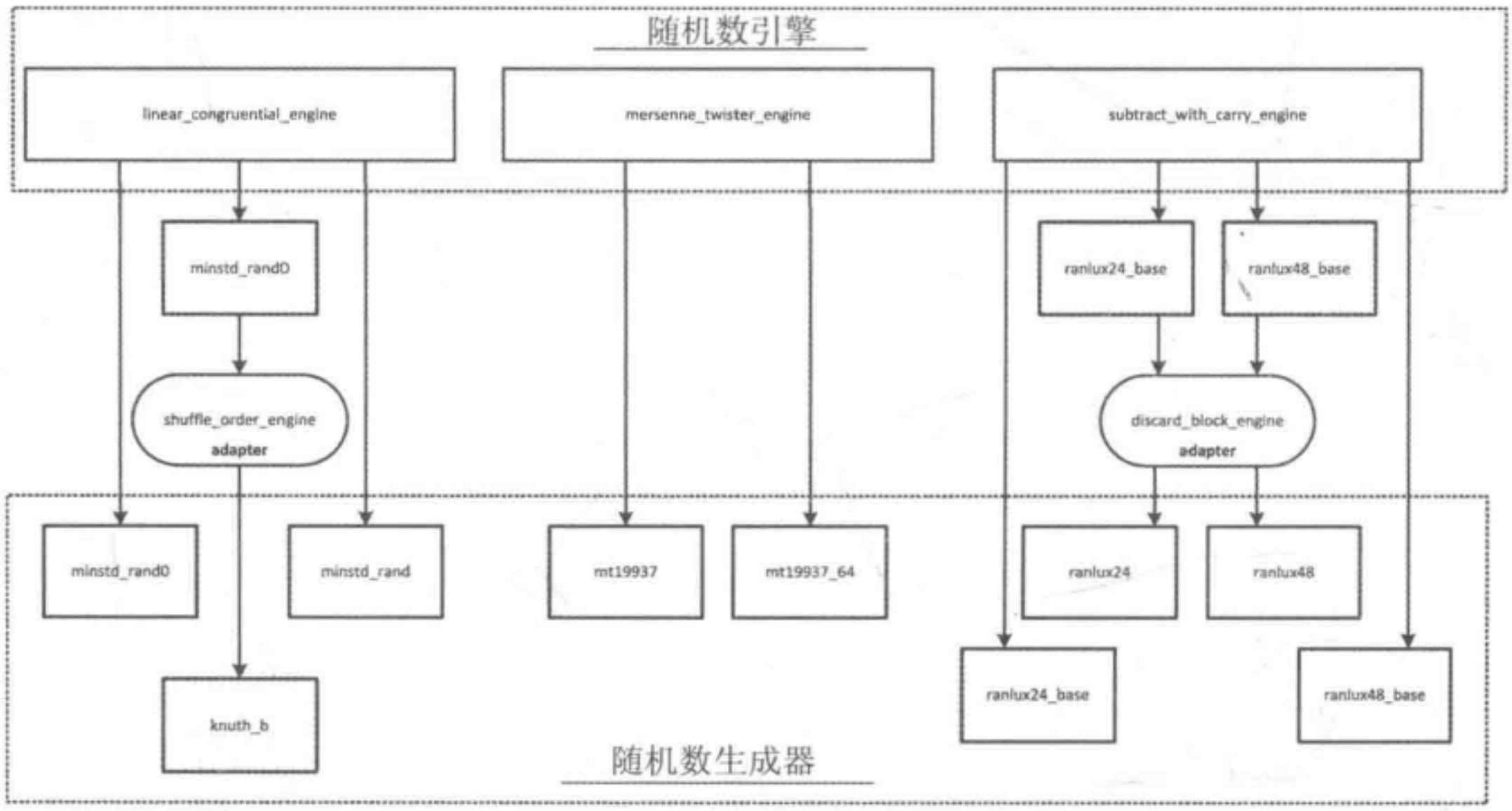


图 8-12 随机数生成器和随机数引擎的关系

每个生成器类类型都是通过将一套模板参数值应用到一个引擎模板生成的。为了让你明白它们做了些什么，在此会对随机数引擎进行一些概述，但强烈推荐用一种可以自定义引擎的随机数生成器类类型，而不是尝试自定义引擎模板。下面介绍在更小的细节上检查引擎以及从它们中定义的生成器类型。

8.5.1　线性同余引擎

linear_congruential_engine 类模板实现了一个最老且最简单的生成整数随机序列的算法，它被叫作线性同余法。这个算法包含 3 个参数：乘数 a、增量 c 和模 m。这些值的选择对于生成合理质量的随机序列至关重要。这个过程需要单个的整数种子和第一个随机值 x，x 理论上可以像这样计算：

```
unsigned int x = (a*seed + c) % m;
```

每个随机数 x_n 都可以用下面这个等式生成下一个 x_n+1：

$$x_{n+1}=(ax_n+c) \bmod_m$$

显然，因为随机值是余数，可以生成的不同值的最大个数是 m，并且 a 和 c 的选择不多，它所生成的值的个数会比 m 更少。然而这个算法是简单快速的，在高质量的随机序列对程序很重要时，最好选择其他引擎实例的生成器，例如 mersenne_twister_engine。

基于线性同余的生成器

有两个被定义为 linear_congruential_engine 模板实例别名的随机数生成器类型：minstd_rand0 和生成 32 位无符号整数的 minstd_rand。名字来自于“minimum standard random number generator”minstd_rand0 是 1998 年由 Stephen K. Park 和 Keith W. Miller 为生成随机数而提出的最低标准，因为那时候的生成器很少。a 被定义为 16 807，c 是 0，m 是 2 147 483 647。m 的值是小于 232 的最大梅森素数。minstd_rand 生成器是 a 为 48271 的 minstd_rand0 的一个改进版本。

归因于 Donald Knuth，knuth_b 随机数生成器实现了一个可以将 shuffle_order_engine 适配器应用到 minst_rand0 生成器所产生的值上的算法。这被描述在他的经典著作<<计算机程序设计的艺术：卷 2>>中，并伴随着大量的随机数生成方法和随机性测试。通过移除连续值之间的依赖来运用适配器增加序列的“随机性”。

这些生成器——事实上所有的生成器——的使用方式都和前面看到的 default_random_engine 相同，例如：

```
std::random_device rd;
std::minstd_rand rng {rd()};
std::uniform_int_distribution<long> dist {-5L, 5L};
for(size_t i {}; i < 8; ++i)
std::cout << std::setw(2) << dist(rng) << " "; // 3 -5 -2 4 -5 4 1 0
```

8.5.2 马特赛特旋转演算法引擎

mersenne_twister_engine 类模板实现了马特赛特旋转演算法，它被这样叫是因为周期长度是一个梅森素数。梅森素数是 2n-1 形式的素数，因此 7 和 127 是梅森素数；当然，用在这个算法中的梅森素数更大。这个引擎的应用非常广泛，因为它可以生成非常高质量的序列，但存在速度相对较慢的缺点。这个算法很复杂并且包含很多的参数，因此在此不作解释。

马特赛特旋转演算法引擎的实例

对于定义具体生成器的 mersenne_twister_engine 的实例有两个类型别名。mt19937 生成随机的无符号 32 位整数，mt19937_64 生成无符号的 64 位整数。mt19937 随机数生成器的周期长度为 219937-1，它因此得名。

可以用和其他生成相同的方式使用它们：

```
std::random_device rd;
std::mt19937_64 rng {rd()};        // Generates random 64-bit integers
std::uniform_real_distribution<long double> dist {-5.0L, 5.0L};
for(size_t i {}; i < 8; ++i)
  std::cout << std::setw(5)
            << dist(rng)
            << " ";                // -2.57481 3.0546 -1.6438 2.14798
                                   // -3.84095 0.973843 -2.98971 -2.1067
```

8.5.3 带进位减法引擎

subtract_with_carry_engine 模板定义了实现带进位减法的随机数引擎，这是对线性同余算法的改进。像线性同余算法一样，带进位减法算法采用递归关系定义序列中的连续值，但是每个值 x_i 都是从序列的前两个值 x_{i-r} 和 x_{i-s} 计算出的，并不是只由前一个值计算得出。r 和 s 分别被称作长脚和短脚，并且都必须是正数，r 必须大于 s。生成这个序列的等式如下：

$\Delta_i = (x_{i-r} - x_{i-s} - c_{i-1}) \bmod m$，m 是 2n，n 是一个单词的比特个数。

$x_i = \Delta_i$ 和 $c_n = 0$，假设 Δ_i 0

$x_i = m + \Delta_i$ 和 $c_n = 1$，假设 $\Delta_i < 0$

c 是一个可能是 0 或 1 的“进位”，这取决于先前的状态。这个算法要求种子值 r 和进位 c 的初始值。和线性同余算法一样，带进位减法算法也对参数值的选择非常敏感。

带进位减法引擎的实例

r 为 24、s 是 10 的 ranlux24_base 生成器类会生成 24 位整数的随机序列。r 是 12、s 是 5 的 ranlux48_base 类产生成 48 位整数的序列。如图 8-12 所示，使用带进位减法引擎的生成器有两个更加高级的类——ranlux24 和 ranlux48，它们是通过将 ranlux24_base 和 ranlux48_base 传入一个 discard_block_engine 适配器的实例生成。一个 ranlux24 的实例会丢

弃每个含有 233 个值的块中的 200 个值，块中的值是由 ranlux24_bas 生成的；一个 ranlux48 的实例会丢弃每个含有 389 个值的块中的 378 个值，块中的值是由 ranlux48_base 生成的，因此它们的值都是从它们接受的基础数据源中精挑细选的。

ranlux24 和 ranlux48 都广泛应用于蒙特卡罗模拟。ranlux 这个名字来源于第一个用 Fortran 实现这个算法的弗雷德詹姆斯。这个名字来自于随机和奢侈地丢掉基础序列中的许多值。下面是一个使用 ranlux24 的示例：

```
std::random_device rd;
std::ranlux24 rng {rd()};
std::uniform_real_distribution<long double> d {-5.0L, 5.0L};
for(size_t i {}; i < 8; ++i)
  std::cout << std::setw(5) << d(rng)
            << " "; // 2.02142 -0.920689 -0.277198 -1.33417 4.70217 -3.31706
             -3.32692 4.36376
```

这段代码本质上和其他的生成器是相同的。这个生成器对象生成的值需要基于构造函数的参数的初始状态。

8.6　重组元素序列

shuffle()会将序列中的元素重新排列成随机排列。shuffle()的函数模板定义在 algorithm 头文件中，但在此并没有包含它，因为它需要一个随机数生成器。元素的全部可能排列是等可能的。shuffle()的前两个参数定义序列的随机访问迭代器，第三个参数是一个用来生成随机序列的均匀随机数生成器的函数对象。下面这段代码可以演示它是如何工作的：

```
std::random_device rd;
std::mt19937 rng {rd()};
std::vector<string> words {"one", "two", "three", "four", "five", "six", "seven", "eight"};
for(size_t i {}; i < 4 ; ++i)
{
  std::shuffle(std::begin(words), std::end(words), rng);
  std::for_each(std::begin(words), std::end(words),
                [](const string& word) {std::cout << std::setw(8) <<
                std::left << word; });
  std::cout << std::endl;
}
```

在 for 循环中，马特赛特旋转演算法 rng 被作为最后一个传入 shuffle()算法的参数。每次遍历 words 容器都会对元素进行重新排列。得到的输出如下：

```
two      seven three five  six   eight one four
eight    five  seven six   three four  one two
seven    one   five  six   eight four  two three
three    four  six   five  seven one   two eight
```

8.7 本章小结

使用 STL 工具总是包含的 3 个组件来生成随机数序列：

- 随机数引擎可以生成随机比特序列。有如下 3 个定义了随机数引擎的类模板：
  - mersenne_twister_engine——能够生成最高质量的随机序列，但它是这 3 个类模板中最慢的。
  - linear_congruential_engine——最简单且最快，但序列的质量比其他两个引擎差。
  - subtract_with_carry_engine——可以比 linear_congruential_engine 生成更好质量的序列，但它的状态会占用太多的内存，而且还有一点慢。
- 自定义引擎模板的随机数生成器实现了一个可以生成一个非负整数的均匀随机分布的特定算法。除了 default_random_engine 是由定义实现的之外，还有 9 个定义了不同生成器的类：
  - mt19937 和 mt19937_64 是从 mersenne_twister_engine 模板得到的。
  - minstd_rand0、minstd_rand、knuth_b 是从 linear_congruential_engine 模板得到的。
  - ranlux24_base、ranlux48_base、ranlux24、ranlux48 是从 subtract_with_carry_engine 模板得到的。
- 使用随机数生成器生成的序列的分布函数对象会用给定的概率分布生成整数或浮点值的序列，有 21 个定义了分布的模板——但只有一个是类模板。
  - 均匀分布：uniform_int_distribution，uniform_real_distribution 和 generate_canonical() 函数模板。
  - 正态分布：normal_distribution、lognormal_distribution、chi_squared_distribution、cauchy_distribution、fisher_f_distribution 和 student_t_distribution。
  - 抽样分布：discrete_distribution、piecewise_constant_distribution、piecewise_linear_distribution。
  - 伯努利分布：bernoulli_distribution、geometric_distribution、binomial_distribution、negative_binomial_distribution。
  - 泊松分布：poisson_distribution、gamma_distribution、weibull_distribution、extreme_value_distribution 和 exponential_distribution。

一些类型的随机数生成器会用随机数引擎适配器来修改从引擎得到的序列。引擎适配器有 3 个类模板：

- 用来定义 knuth_b 的 shuffle_order_engine。
- 用来定义 ranlux24 和 ranlux48 的 discard_block_engine。
- 未在 STL 中应用的 independent_bits_engine。

随机数生成器需要一个或多个种子值来初始它的状态。random_device 类定义了可以返回非负整数的均匀分布序列的函数对象，非法整数的均匀分布的大多数实现都是非确定性的。为了能够确定是否得到高质量的序列，不能直接使用随机数引擎——应该总是使用随

机数生成器。

练习

1. 将 Ex8_03 修改为在发牌之前使用 shuffle()算法洗牌。

2. 将前一个练习的解决方案扩展为在随机发完 4 手牌之后，再开始游戏。每个玩家都会依次从他们的手牌中随机地打一张牌，输出每一轮打出的牌，并且确认有赢牌的玩家，赢牌是排序序列中最大的牌。

3. 用单个离散分布模拟扔两个标准的骰子，因此分布对象生成的值是两个骰子之和。抛 5000 次骰子，并画出它们的直方图。

4. 写一个程序，估计在同时通过抛掷 6 个标准骰子来模拟大量骰子被抛出时，所有可能出现的概率。

第9章

流 操 作

本章会重新回顾第 1 章介绍的流迭代器，并更加详细地讨论它们的功能，还会介绍如何在 STL 的其他功能中将流和流缓冲区迭代器结合起来。本章将介绍以下内容：

- 流迭代器类提供了哪些成员函数
- 如何用流迭代器读写单独的数据项
- 什么是流缓冲区迭代器流及其与迭代器的差别
- 如何使用流缓冲区迭代器读写文件
- 如何用流迭代器读写文件
- 什么是字符串流以及 STL 定义的不同类型的字符串流
- 如何对字符串流使用流迭代器和流缓冲区迭代器

9.1 流迭代器

众所周知，流迭代器是从流中读取的单通迭代器，它是一个输入流迭代器，或写入流(如果它是一个输出流迭代器的话)。流迭代器只能传送给定类型的数据到流中或者从流中读取给定类型的数据。如果想用流迭代器来传送一系列不同类型的数据项，就必须将数据项打包到一个单一类型的对象中，并保证这种类型存在流插入和/或流提取运算符。和其他迭代器相比，流迭代器有一点奇怪。例如，递增一个输入流迭代器并不会将迭代器转移指向下一个数据项——它会从流中读取一个值。让我们开始探讨这些细节吧！

9.1.1 输入流迭代器

输入流迭代器是一个可以在文本模式下从流中提取数据的输入迭代器，这意味着不能用它处理二进制流。一般用两个流迭代器来从流中读取全部的值：指向要读入的第一个值的开始迭代器，指向流的末尾的结束迭代器。在输入流的文件结束状态(End-Of-File, EOF)被识别时，就可以确定结束迭代器。定义在 iterator 头文件中的 istream_iterator 模板会用提

取运算符>>从流中读入 T 类型的值。对于这些工作，必须有一个从 istream 对象读取 T 类型值的 operator>>()函数的重载版本。因为 istream_iterator 是一个输入迭代器，所以它的实例是一个单向迭代器；它只能被使用一次。默认情况下，我们认为这种流包含的是 char 类型的字符。

可以通过传入一个输入流对象到构造函数来生成一个 istream_iterator 对象。istream_iterator 对象有拷贝构造函数。下面是一个生成输入流迭代器的示例：

```
std::istream_iterator<string> in {std::cin};   // Reads strings from cin
std::istream_iterator<string> end_in;          // End-of-stream iterator
```

默认构造函数会生成一个代表流结束的对象——也就是当 EOF 被识别时的那种对象。

虽然默认情况下，这种流被认为包含的是 char 类型的字符，但能够定义输入流迭代器来读取包含其他类型字符的流。例如，下面展示了如何定义输入流迭代器来读取一个包含 wchar_t 字符的流：

```
std::basic_ifstream<wchar_t> file_in {"no_such_file.txt"}; // File stream
                                                           // of wchar_t
std::istream_iterator<std::wstring, wchar_t> in {file_in}; // Reads strings
                                                          // of wchar_t
std::istream_iterator<std::wstring, wchar_t> end_in; // End-of-stream iterator
```

第一条语句定义了一个 wchar_t 字符的输入文件流。在下一节会提醒你注意文件流的一些关键细节。为了读这个文件，第二条语句定义了一个流迭代器。流中的字符类型是由 istream_iteratot 的第二个模板类型参数指定的，在这个实例中是 wchar_t。当然，指定从流中读入的对象的类型的第一个模板类型参数必须是 wstring，它是 wchar_t 字符的字符串类型。

istream_iterator 对象有下面这些成员函数：

- operator*()会返回一个流中当前对象的引用，可以运用这个运算符多次以重读相同的值。
- operator->()会返回流中当前对象的地址。
- operator++()会从底层输入流中读取一个值，并将它保存到一个迭代器对象中，返回一个迭代器对象的引用。因此，表达式*++in 的值为最新的被保存的值。这不是一般的用法，因为它可能会跳过流中的第一个值。
- operator++(int)会从底层输入流读取一个值，并将它保存到一个迭代器对象中，为使用 operator*()或 operator->()访问做准备。在流中的新值被保存之前，这个函数会返回迭代器对象的一个代理。这意味着在读和保存底层输入流中的最新值之前，表达式*in++的值是保存在迭代器中的对象。

也有非成员函数，operator==()和 operator!=()可以比较相同类型的对象。两个输入迭代器是相等的，前提是它们都是同一个流的迭代器或者都是流的结束迭代器；否则，它们就不相等。

1. 迭代器和流迭代器

输入流迭代器和常规迭代器在数据项序列的关联方式上是不同的，明白这一点很重

要。常规迭代器指向的是数组或容器中的元素。递增一个常规迭代器会改变它所指向的元素；这对指向同一个序列的元素的其他迭代器没有影响。这一点和流迭代器不同。

当用流迭代器来从标准输入流读取数据时，显然要考虑会发生什么；当流迭代器指向文件时，就可能不那么明显，但仍然可以应用。如果生成了两个和同一个流相关的输入流迭代器，初始时它们都指向第一个数据项。如果用一个迭代器来读取流，另一个不会再引用第一个数据值。当从标准输入流读取时，值会被第一个迭代器消耗。这是因为这个迭代器在读取一个值时，会修改流对象。输入流迭代器不仅会改变它所指向的元素——在引用时得到的结果——也会改变底层流中确定下一次读操作从哪里开始的位置。因此，给定流的两个或两个以上的输入流迭代器会指向从这个流中得到的下一个数据项。这意味着两个输入流迭代器所确定的序列只能由开始迭代器和流的结束迭代器组成。我们无法生成两个指向同一个流中两个不同值的流迭代器，这并不是说不能用输入流迭代器来访问数据项。在适当时，我们会看到可以这么做。

2. 用输入流的成员函数读取数据

下面是一些说明如何用成员函数读取字符串的代码：

```
std::cout << "Enter one or more words. Enter ! to end:\n";
std::istream_iterator<string> in {std::cin};   // Reads strings from cin
std::vector<string> words;
while(true)
{
  string word = *in;
  if(word == "!") break;
  words.push_back(word);
  ++in;
}
std::cout << "You entered " << words.size() << " words." << std::endl;
```

循环从标准输入流中读取单词，并把它们添加到 vector 容器中，直到按下回车键。表达式*in 的值是从底层流读到的当前 string 对象。++in 会导致从流中读取下一个 string 对象，并保存到这个迭代器中。下面是执行这段代码的示例输出：

```
Enter one or more words. Enter ! to end:
Yes No Maybe !
You entered 3 words.
```

下面是一个演示如何用成员函数来读取数值数据的示例，但不一定要这样使用：

```
// Ex9_01.cpp
// Calling istream_iterator function members
#include <iostream>                 // For standard streams
#include <iterator>                 // For stream iterators

int main()
{
```

```
std::cout << "Enter some integers - enter Ctrl+Z to end.\n";
std::istream_iterator<int> iter {std::cin};     // Create begin input
                                                // stream iterator...
std::istream_iterator<int> copy_iter {iter};    // ...and a copy
std::istream_iterator<int> end_iter;            // Create end input
                                                // stream iterator

// Read some integers to sum
int sum {};
while(iter != end_iter)                         //Continue until Ctrl+Z read
{
  sum += *iter++;
}
std::cout << "Total is " << sum << std::endl;

std::cin.clear();                               // Clear EOF state
std::cin.ignore();                              // Skip characters

// Read integers using the copy of the iterator
std::cout << "Enter some more integers - enter Ctrl+Z to end.\n";
int product {1};
while(true)
{
  if(copy_iter == end_iter) break;              // Break if Ctrl+Z was read
  product *= *copy_iter++;
}
std::cout << "product is " << product << std::endl;
}
```

在显示输入提示后，我们可以用一个输入流迭代器从 cin 中读取 int 类型的值；然后会复制迭代器对象的一个副本。在原始对象 iter 被使用后，我们可以用副本 y_iter 从 cin 读取输入，我们只需要一个结束迭代器对象，因为它从没有改变。第一个循环会求出所有用输入流迭代器读入的值的和，直到识别出 EOF 状态，它是通过从流中读取 Ctrl+Z 标志设置的。解引用 iter 可以使它所指向的值变得可用，之后，后递增运算会移动 iter，使它指向下一个输入。如果输入的是 Ctrl+Z，循环结束。

在从 cin 读取更多数据之前，必须调用流对象的 clear()来重置 EOF 标志；也需要跳过输入缓冲区中留下的'\n'字符，这是通过调用流对象的 ignore()做到的。第二个循环会用 copy_iter 来读取值并计算它们的 product。第一个循环的主要差别是它是：通过比较 copy_iter 和 end_iter 的相等性来终止循环的。

下面是这个示例的输出：

```
Enter some integers - enter Ctrl+Z to end.
1 2 3 4^Z
Total is 10
Enter some more integers - enter Ctrl+Z to end.
3 3 2 5 4^Z
product is 360
```

在大多数时候，我们并不会像这样使用输入流迭代器。一般来说，只需要用一个流的开始和结束迭代器作为一个函数的参数。我们可能意识到，能够只用一条语句来替代Ex9_01中的第一个循环及其后面的输出语句：

```
std::cout << "Total is " << std::accumulate(iter, end_iter, 0) << std::endl;
```

下面是通过用一个输入流迭代器，将从cin得到的浮点值插入到一个容器中的代码：

```
std::vector<double> data;
std::cout << "Enter some numerical values - enter Ctrl+Z to end.\n";
std::copy(std::istream_iterator<double>{std::cin},
          std::istream iterator
  <double>{},
std::back_inserter(data));
```

copy()算法会将任意个数的值附加到vector容器中，直到读到Ctrl+Z。vector容器有一个接受序列来初始化元素的构造函数，因此可以用生成容器的语句中的输入流迭代器来读取值：

```
std::cout << "Enter some numerical values - enter Ctrl+Z to end.\n";
std::vector<double> data {std::istream_iterator<double>{std::cin},
                          std::istream_iterator<double>{}};
```

这会从标准输入流中读取浮点值，并用它们作为容器中元素的初始值。

9.1.2 输出流迭代器

输出流迭代器是由ostream_iterator模板定义的，这个模板的第一个模板参数是被写值的类型，第二个模板参数是流中字符的类型；第二个模板参数默认是 char 类型的值。ostream_iterator是一个能够将任意T类型对象写到文本模式的输出流中的输出迭代器；只要T类型的对象实现了将T类型对象写到流中的operator<<()。因为它是一个输出迭代器，所以它支持前向和后向递增操作，并且是一个单向传入的迭代器。输出流迭代器定义了它的拷贝赋值运算符，因此可以用插入运算符将T对象写到流中。默认的输出流迭代器会按照char字符的序列来写入值。通过指定第二个模板类型参数，可以写包含不同类型字符的流。ostream_iterator类型定义了下面这些成员函数：

- 构造函数：第一个构造函数会用作为第一个参数的ostream对象的输出流生成一个开始迭代器，第二个参数是分隔符字符串。输出流对象会在每个被它写入到流中的对象的后面写分隔符字符串。第二个构造函数可以省略第二个差数，它会生成一个写对象时后面不跟分隔符的迭代器。
- operator=(const T& obj)会将obj写到流中，然后写分隔符字符串，前提是在构造函数中指定了一个。这个函数返回一个迭代器的引用。
- operator*()不做任何事，除了返回迭代器对象。为了将迭代器限定为输出迭代器，这个操作必须被定义。

- operator++()和 operator++(int)都被定义了但不做任何事，除了返回迭代器对象。为了将迭代器限定为输出迭代器，前向或后向自增操作必须被支持。

不做任何事的运算符函数是必要的，因为它们是说明输出迭代器可以做些什么的规范的一部分。如果想写数据到一个文本模式的流中，并且随后打算以文本方式读取，流的值之间就需要分隔符。出于这个原因，尽管可以显式地写分隔符，但通常会使用有两个参数的构造函数。

用输出流迭代器的成员函数写

下面是一个示例，展示了成员函数可以被使用的各种方式：

```
// Ex9_02.cpp
// Using output stream iterator function members
#include <iostream>                 // For standard streams
#include <iterator>                 // For iterators and begin() and end()
#include <vector>                   // For vector container
#include <algorithm>                // For copy() algorithm
#include <string>
using std::string;

int main()
{
  std::vector<string> words {"The", "quick", "brown", "fox", "jumped",
"over", "the", "lazy", "dog"};

// Write the words container using conventional iterator notation
std::ostream_iterator<string> out_iter1 {std::cout}; // Iterator with no
                                                     // delimiter output
for(const auto& word : words)
{
  *out_iter1++ = word;              // Write a word
  *out_iter1++ = " ";               // Write a delimiter
}
*out_iter1++ = "\n";                // Write newline

// Write the words container again using the iterator
for(const auto& word : words)
{
  (out_iter1 = word) = " ";         // Write the word and delimiter
}
out_iter1 = "\n";                   // Write newline

// Write the words container using copy()
std::ostream_iterator<string> out_iter2 {std::cout, " "};
std::copy(std::begin(words), std::end(words), out_iter2);
out_iter2 = "\n";
}
```

这段代码以 3 种方式将 words 容器中的元素写到标准输出流中。out_iter1 流迭代器通过调用只以输出流为参数的构造函数而生成。第一个循环以常规方式输出迭代器符号，在

解引用后递增它，并复制 word 的当前值到 out_iter1 的解引用的结果中。循环后面的语句会在流中写一个新行。注意，不能这样写：

```
out_iter1 = '\n';                    // Won't compile!
```

定义的迭代器会将 string 对象写入流中，因此它不能写任何其他类型的数据。成员函数 operator=()只接受一个 string 参数，因此这条语句无法通过编译。

像前面描述的那样，operator*()和前自增及后自增运算符除了返回迭代器的引用外，不会做任何事。因此可以去除这些操作，像第二个循环中的语句那样，不使用它们也可以产生同样的输出。为了保证应用分隔符的第二条赋值语句有输出迭代器作为它的左操作数，语句中的括号是必要的。

第三行的输出是通过以你在前面的章节中看到过的方式调用 copy()算法生成的。元素的值会被复制到 out_iter2 中，定义它的构造函数的第二个参数指定了每个输出值后面的分隔符字符串。

9.2 重载插入和提取运算符

必须为任何想和输出流迭代器一起使用的类类型重载插入和提取运算符。对于自己的类，这是很容易的事。必要的话，可以提供 get 和 set 函数来访问任何 private 或 public 数据成员，或者将运算符函数指定为友元函数。下面是一个简单的示例，有一个表示姓名的类可以说明：

```
class Name
{
private:
  std::string first_name{};
  std::string second_name{};
public:
  Name() = default;
  Name(const std::string& first, const std::string& second) :
                    first_name{first}, second name {second} {}
  friend std::istream& operator>>(std::istream& in, Name& name);
  friend std::ostream& operator<<(std::ostream& out, const Name& name);
};

// Extraction operator for Name objects
inline std::istream& operator>>(std::istream& in, Name& name)
{ return in >> name.first_name >> name.second_name; }

// Insertion operator for Name objects
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{ return out << name.first_name << ' ' << name.second_name; }
```

这里定义了运算符的重载，可以像这个示例一样，用流迭代器来读和写 name 对象：

```
std::cout << "Enter names as first-name second-name. Enter Ctrl+Z on a
             separate line to end:\n";
std::vector<Name> names {std::istream_iterator<Name> {std::cin},
                         std::istream_iterator<Name>{}};
std::copy(std::begin(names), std::end(names), std::ostream_ iterator< Name>
  {std::cout, " "});
```

names 容器会被输入的 Name 对象初始化，直到输入了 Ctrl+Z 来结束输入。copy()算法会将 Name 对象复制到输出流迭代器标识的目的位置，这个迭代器会将对象写到标准输出流中。我们这里的名和姓会阻止姓名按指定的宽度列对齐。例如，下面代码的效果就不会很好：

```
for(const auto& name: names)
std::cout << std::setw(20) << name << std::endl;
```

有个主意，就是在单一的列中输出姓名。这个效果也不理想，因为指定的宽度只会被应用到成员 first_name 上。可以通过改变 operator<<()来做到这一点，因此在输出姓名之前，会先将它们连接起来：

```
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{ return out << name.first_name + ' ' + name.second_name; }
```

这没有原始的<<高效，因为这种方式会生成临时的 string 对象，但它能够使前面的循环按要求工作。

有时候，可能会想以不同的方式来输出，这取决于目的位置是不是一个文件。例如，我们可能会想在标准输出流的输出中包含一些在将它们写入文件时不想包含的额外信息。这可以通过检查 ostream 对象的实际类型来做到：

```
inline std::ostream& operator<<(std::ostream& out, const Name& name)
{
  if(typeid(out) != typeid(std::ostream))
    return out << name.first_name << " " << name.second_name;
  else
    return out << "Name: " << name.first_name << ' ' << name.second_name;
}
```

现在，当它被写到 ostream 对象的流时，name 会被带上前缀"Name: "。在输出到文件流或从 ostream 派生的其他流类型中，前缀会被省略。

9.3 对文件使用流迭代器

流迭代器并不知道底层流的特性。当然，它们只适用于文本模式，否则它们不会关心数据是什么。流迭代器可以以文本模式来读写任何类型的流。这意味着除了其他的一些流之外，我们可以用迭代器以文本模式来读和写文件。在深入讲解如何对文件使用流迭代器之前，需要提醒你文件流的一些本质特征以及如何生成一个封装了文件的流对象。

9.3.1 文件流

文件流封装了一个实际的文件。文件流有长度，也就是这个流中字符的个数，因此对于新的输出文件，长度就是 0；文件流有起始位置，起始位置是流中索引为 0 的第一个字符的索引；文件流也有结束位置，结束位置是文件流中最后一个字符的下一个位置。文件流还有当前位置，是下一个写或读操作的开始位置的索引。可以以文本模式或二进制模式将数据转移到文件中或从文件中读出来。

在文本模式下，数据是字符的序列。可以用提取和插入运算符来读写数据，至少对于输入来说，数据项必须由一个或多个空格隔开。数据经常被写成以'\n'终止的连续行。一些系统，例如微软的 Windows 系统，在读写时会转换换行符。在读到回车和换行符时，它们会被映射到单个字符'\n'。在另一些系统中，换行符被当作单个字符读写。因此，文件输入流的长度依赖于它们所来自的系统环境。

在二进制模式下，内存和流之间是以字节的形式传送数据的，不需要转换。流迭代器只能工作在文本模式下，因此不能用流迭代器来读写二进制文件。本章后面要介绍的流缓冲迭代器，可以读写二进制文件。

尽管在二进制模式下，从内存中读取和写入的字节从来不会改变，但当谈到处理写到不同系统中的二进制文件时，仍然会有很多陷阱。一个考虑是，写文件的系统的字节顺序和读文件的系统的字节顺序是相反的。字节顺序决定了内存中字的写入顺序。在小端字节序的处理器中，例如英特尔的 X86 处理器，最低位的字节在最低的地址，所以字节的写入顺序是从最底字节到最高字节。在大端字节序的处理器中，例如 IBM 大型机，比特的顺序是相反的，最高字节在最低位置，因此它们的文件会出现和小端字节序的处理器相反的顺序。因此，当在小端字节序的系统中读来自于大端字节序的系统中的二进制文件时，需要考虑字节序的差别。

■ 注意：大端字节序也被称为网络字节序，因为数据一般是以大端序在互联网上传输的。

9.3.2 文件流类的模板

这里有 3 个表示文件流的类模板：ifstream 表示文件的输出流，ofstream 是为输出定义的文件流，fstream 定义了可以读和写的文件流。这些类的继承关系如图 9-1 所示。

文件流模板从 istream 和/或 ostream 继承，因此在文本模式中，它们的工作方式和标准流相同。能够对一个文件流做什么是由它的打开状态决定的，可以用下面这些定义在 ios_base 类中的常量的组合来指定它们的打开状态：

- binary：会将文件设置成二进制模式。如果没有设置二进制模式，默认的模式是文本模式。
- app：在每个写操作(append operation)之前会移到文件的末尾。
- ate：会在打开文件之后(at the end)，移到文件的末尾。
- in：打开文件来读。对于 ifstream 和 fstream 来说，这是默认的。
- out：打开文件来写。对于 ostream 和 fstream 来说，这是默认的。

- trunc：将当前存在的文件长度截断为 0。

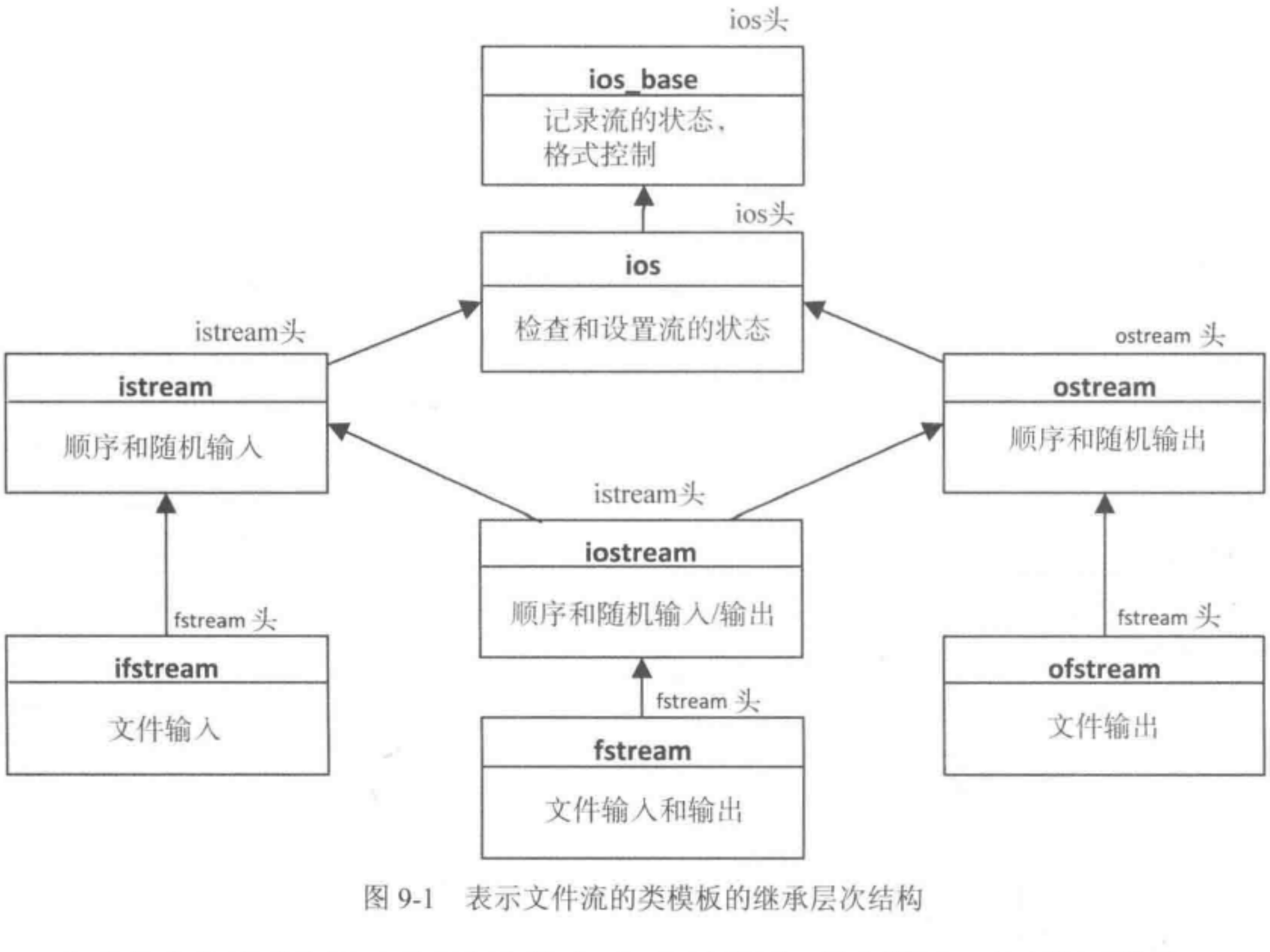


图 9-1　表示文件流的类模板的继承层次结构

生成的文件流对象默认是在文本模式下；为了得到二进制模式，必须指定常量 binary。文本模式的操作会用>>和<<运算符来读写数据，数值数据在被写到流之前会被转换为它们的字符表示。在二进制模式下，没有数据转换；内存中的字节会被直接写到文件中。当指定一个并不存在的文件的名称作为 ofstream 的构造函数参数时，这个文件会被生成。如果在生成或打开一个文件输出流对象时不指定 app 或 ate，任何存在的文件的内容都会被覆盖。

在本章的几个示例中都会读 dictionary.txt，它被包含在下载的代码中。在微软的 Windows 环境下，会用文本模式来写文件；但如果在不同的环境中执行这个程序，这个示例也应该读它。这个示例使用了微软 Windows 的路径 G:。这样做是为了使需要改变这些来适应系统环境变得更加可能，同时会使我们有责任确认不会覆盖掉重要的文件。

9.3.3　用流迭代器进行文件输入

一旦创建用于读文件的文件流对象，用流迭代器来访问数据和从标准输入流读数据就基本相同。我们可以写一个查找字谜的程序，通过下载代码中的字典文件来查找它们。在这种情况下，我们需要用流迭代器来将字典文件中的所有单词读取到容器中。下面是代码：

```
// Ex9_03.cpp
// Finding anagrams of a word
#include <iostream>                          // For standard streams
```

```
#include <fstream>                    // For file streams
#include <iterator>                   // For iterators and begin() and end()
#include <string>                     // For string class
#include <set>                        // For set container
#include <vector>                     // For vector container
#include <algorithm>                  // For next_permutation()
using std::string;

int main()
{
  // Read words from the file into a set container
  string file_in {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream in {file_in};
  if(!in)
  {
    std::cerr << file_in << " not open." << std::endl;
    exit(1);
  }
  std::set<string> dictionary { std::istream_iterator<string>(in),
                                std::istream_iterator<string>()};
  std::cout << dictionary.size() << " words in dictionary." << std::endl;

  std::vector<string> words;
  string word;
  while(true)
  {
    std::cout << "\nEnter a word, or Ctrl+z to end: ";
    if((std::cin >> word).eof()) break;
    string word_copy {word};
    do
    {
      if(dictionary.count(word))
        words.push_back(word);
      std::next_permutation(std::begin(word), std::end(word));
    } while(word != word_copy);

    std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
      {std::cout, " "});
    std::cout << std::endl;
    words.clear();                    // Remove previous permutations
  }
  in.close();                         // Close the file
}
```

字典文件中包含超过 100 000 个单词，因此读取会花费一些时间。ifstream 对象是用文件 dictionary.txt 的全路径来生成。这个文件包含了相当数量的不同单词，可以用来搜查检查字谜。整个文件的内容被用来作为 set 容器的初始值。正如我们知道的那样，set 容器会以升序保存单词，并且这个容器中的每个单词都有自己的 key。words 容器中包含从 cin 中输入的单词的字谜。每一个单词都在 while 循环的第一个 if 表达式中读取。这里调用流对

象的 eof()，当输入 Ctrl+Z 时，eof()会返回 true。输入的单词的字符重排列是通过在 do-while 循环中调用 next_permutation()算法生成的。对于每次排列，包括第一次，都调用 count()来确定这个单词是否在 dictionary 容器中。如果在，这个单词会被附加到 words 容器中。当排列返回原始单词时，do-while 循环结束。当找到一个单词的所有字谜时，会用以输出流迭代器作为目的地址的 copy()算法将 words 写到 cout 中。如果期望每次出现 8 个字谜，需要使用循环来生成多行输出：

```
size_t count {}, max {8};
for(const auto& wrd : words)
  std::cout << wrd << ((++count % max == 0) ? '\n' : ' ');
```

下面是一些示例输出：

```
109582 words in dictionary.

Enter a word, or Ctrl+z to end: realist
realist retails saltier slatier tailers

Enter a word, or Ctrl+z to end: painter
painter pertain repaint

Enter a word, or Ctrl+z to end: dog
dog god

Enter a word, or Ctrl+z to end: ^Z
```

9.3.4 用流迭代器来反复读文件

当然，如果字典文件非常大，可能不想把它全部读到内存中。在这种情况下，可以在每次想要查找字谜时，用流迭代器来重读文件。下面的版本可以实现这一点——尽管它的表现并不怎么令人印象深刻：

```
// Ex9_04.cpp
// Finding anagrams of a word by re-reading the dictionary file

// include directives & using directive as Ex9_03.cpp...

int main()
{
  string file_in {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream in {file_in};
  if(!in)
  {
    std::cerr << file_in << " not open." << std::endl;
    exit(1);
  }
  auto end_iter = std::istream_iterator<string> {};

  std::vector<string> words;
  string word;
```

```
while(true)
{
  std::cout << "\nEnter a word, or Ctrl+z to end: ";
  if((std::cin >> word).eof()) break;
  string word_copy {word};
  do
  {
    in.seekg(0);                              // File position at beginning

    // Use find() algorithm to read the file to check for an anagram
    if(std::find(std::istream_iterator<string>(in), end_iter, word) !=
      end_iter)
      words.push_back(word);
    else
      in.clear();                             // Reset EOF

    std::next_permutation(std::begin(word), std::end(word));
  } while(word != word_copy);

  std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
    {std::cout, " "});
  std::cout << std::endl;
  words.clear();                              // Remove previous permutations
}
in.close();                                   // Close the file
}
```

结束流迭代器不会改变，因此为了可以多次使用它，会将它定义为 end_iter。循环基本相同，除了用来发现给定的排列是否在文件中，如果在文件中，就是一个字谜。文件的位置需要是第一个字符的位置，在这个示例中会调用文件对象的 seekg()来保证这一点。find()的前两个参数是定义从当前位置得到的序列的 istream_iterator<string>对象，它们被设置为文件的开始位置和结束位置。find()算法会返回一个迭代器，它指向的元素和第三个参数匹配，或者返回结束迭代器，如果它并不存在的话。因此，如果 find()返回的是结束流迭代器，就没有找到 word；返回其他任何迭代器都意味着找到了 word。当 w 没有找到 word 时，为文件流对象调用 clear()来清除 EOF 标志是有必要的。如果不这么做，随后访问文件会失败，因为 EOF 标志已被设置。

下面是一些说明它工作的示例输出：

```
Enter a word, or Ctrl+z to end: rate
rate tare tear erat

Enter a word, or Ctrl+z to end: rat
rat tar art

Enter a word, or Ctrl+z to end: god
god dog

Enter a word, or Ctrl+z to end: ^Z
```

选择输入短单词，因为检查字谜的过程缓慢而痛苦。n 个字符的单词有 n！种排列。为检查一个排列是否在文件中，需要比较 100 000 次读操作，取决于它是否在文件中。因此，检查一个单词，例如“retain”需要 7 百万次读操作，这也是为什么这个过程十分缓慢的原因。istream_iterator<T>对象会从流中一次读取一个 T 对象，因此如果有大量的对象，这个过程也总是很慢。一旦文件被读到 set 容器中，Ex9_03.cpp 肯定比 Ex9_04.cpp 快，因为随后的所有操作都是在内存的字典 words 中进行的。Ex9_03.cpp 更快的第二个原因是，对 set 容器的访问包含时间复杂度为 O(log n)的二分查找。串行访问文件包含每次从起始位置开始读单词直到找到匹配，时间复杂度为 O(n)。如果文件中的数据是有序的(dictionary.txt 中的单词是有序的)，就能用二分查找技术来找到数据项。在这种情况下，流迭代器是多余的，因为总会读单个单词，对流对象使用>>运算符可以更容易地做到这一点。然而，这并不容易实现，因为单词的长度不同。

9.3.5 用流迭代器输出文件

写文件和写标准输出流没有什么不同。例如，可以按如下方式用流迭代器复制 dictionary.txt 文件的内容：

```
// Ex9_05.cpp
// Copying file contents using stream iterators
#include <iostream>                    // For standard streams
#include <fstream>                     // For file streams
#include <iterator>                    // For iterators and begin() and end()
#include <string>                      // For string class

int main()
{
  string file_in {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream in {file_in};
  if(!in)
  {
    std::cerr << file_in << " not open." << std::endl;
    exit(1);
  }
  string file_out {"G:/Beginning_STL/dictionary_copy.txt"};
  std::ofstream out {file_out, std::ios_base::out | std::ios_base::trunc };
  std::copy(std::istream_iterator<string> {in}, std::istream_iterator<string> {},
                              std::ostream_iterator<string>{out, " "});
  in.clear();                                        // Clear EOF
  std::cout << "Original file length: " << in.tellg() << std::endl;
  std::cout << "File copy length: " << out.tellp() << std::endl;
  in.close();
  out.close();
}
```

这个程序会从输入文件复制单词到输出文件，以空格隔开输出的单词。这个程序总会覆盖输出文件的内容。下面是得到的输出：

```
Original file length: 1154336
File copy length: 1154336
```

输出文件流的打开模式除了有 ios_base::out 标志之外，还有 ios_base::trunc 标志，因此如果文件已经存在，它会被截断。这阻止生成不断增长的文件，如果多次运行这个示例的话。如果用编辑器检查 dictionary.txt 的内容，会看到单词是被单个空格隔开的。我们写的文件副本的每个单词之间都有一个空格，因此文件的长度相同。然而，如果原始文件中的单词是用两个或两个以上空格隔开的，文件的副本会更短。为了确保能够用流迭代器准确地复制原文件，必须一个字符一个字符地读文件，并阻止>>运算符忽略空格。下面展示了如何这么做：

```
std::copy(std::istream_iterator<char>{in >> std::noskipws},
          std:: istream iterator<char>{},
          std::ostream_iter ator<char>{out});
```

这段代码会按字符复制 in 流，包含空格，这样复制文件比流迭代器更快，在本章的后面会进行解释。

9.4 流迭代器和算法

我们已经看到，可以对流迭代器使用像 find()和 copy()这种算法。可以用流迭代器来为接受输入迭代器指定数据源的任何算法指定数据源。如果算法需要正向、双向或随机访问迭代器来定义输入，就不能用流迭代器。下面是一个示例，它会对流迭代器使用 count_if()算法来确定 dictionary.txt 中出现和单词中有相同初始字母的频率：

```
// Ex9_06.cpp
// Using count_if() with stream iterators to count word frequencies
#include <iostream>                    // For standard streams
#include <iterator>                    // For iterators and begin() and end()
#include <iomanip>                     // For stream manipulators
#include <fstream>                     // For ifstream
#include <algorithm>                   // For count_if()
#include <string>
using std::string;

int main()
{
  string file_in {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream in {file_in};
  if(!in)
  {
    std::cerr << file_in << " not open." << std::endl;
    exit(1);
  }
  string letters {"abcdefghijklmnopqrstuvwxyz"};
```

```
const size_t perline {9};
for(auto ch : letters)
{
  std::cout << ch << ": "
    << std::setw(5)
    << std::count_if(std::istream_iterator<string>{in}, std::istream_
                                                        iterator<string>{},
                                                        [&ch](const string& s)
                                                        { return s[0] == ch; })
    << (((ch - 'a' + 1) % perline) ? " " : "\n");
    in.clear();                          // Clear EOF...
    in.seekg(0);                         // ... and back to the beginning
  }
  std::cout << std::endl;
}
```

得到的输出如下：

```
a:  6541 b: 6280 c: 10324 d: 6694 e: 4494 f: 4701 g: 3594 h: 3920i: 4382
j:  1046 k:  964 l:  3363 m: 5806 n: 2475 o: 2966 p: 8448 q: 577 r: 6804
s: 12108 t: 5530 u:  3312 v: 1825 w: 2714 x:   79 y:  370 z: 265
```

这个程序展示了 count_if()算法对流迭代器的使用，但不是特别有效率。for 循环遍历 letter 中的字符，并在每次遍历中调用 count_if()来计算以当前字母开头的单词的个数，这会遍历文件中的所有单词。因为输入文件是有序的，不需要每次读入全部的文件。能够用 for_each()算法更快地得到相同的结果：

```
std::map <char, size_t> word_counts;   // Stores word count for each initial letter
size_t perline {9};                    // Outputs per line

// Get the words counts for each initial letter
std::for_each(std::istream_iterator<string>{in}, std::istream_iterator<string>{},
                          [&word_counts](const string& s) {word_counts[s[0]]++;});

std::for_each(std::begin(word_counts), std::end(word_counts), // Write out the counts
                         [perline](const std::pair<char, size_t>& pr)
                         {std::cout << pr.first << ": "
                                    << std::setw(5) << pr.second
                                    << (((pr.first - 'a' + 1) % perline) ?
                                       " " : "\n");
                         });
std::cout << std::endl;
```

for_each()算法的第一次调用遍历文件中的单词，并第一次保存一个新的 pair 到 words_counts 容器中，单词的给定的初始字母会被传到 lambda 表达式中。当单词遇到前面发现的初始字母时，pair 的值会递增。for_each()的第二次调用会输出 map 中的元素。文件只会被处理一次，因此比前面的版本快约 26 倍。

generate_n()算法适用于流迭代器。下面展示如何将一个流迭代器传入到算法中来生成

一个包含一些 Fibonacci 数列的文件，然后读这个文件来验证它是否有效：

```
// Ex9_07.cpp
// Using stream iterators to write Fibonacci numbers to a file
#include <iostream>                    // For standard streams
#include <iterator>                    // For iterators and begin() and end()
#include <iomanip>                     // For stream manipulators
#include <fstream>                     // For fstream
#include <algorithm>                   // For generate_n() and for_each()
#include <string>
using std::string;

int main()
{
  string file_name {"G:/Beginning_STL/fibonacci.txt"};
  std::fstream fibonacci {file_name, std::ios_base::in | std::ios_base::out |
       std::ios_base::trunc};
  if(!fibonacci)
  {
    std::cerr << file_name << " not open." << std::endl;
    exit(1);
  }
  unsigned long long first {0ULL}, second {1ULL};
  auto iter = std::ostream_iterator<unsigned long long> {fibonacci, " "};
  (iter = first) = second; // Write the first two values
  const size_t n {50};
  std::generate_n(iter, n, [&first, &second]
  {auto result = first + second;
  first = second;
  second = result;
  return result; });
  fibonacci.seekg(0);                  // Back to file beginning
  std::for_each(std::istream_iterator<unsigned long long> {fibonacci},
    std::istream_iterator<unsigned long long> {},
    [](unsigned long long k)
  {const size_t perline {6};
    static size_t count {};
    std::cout << std::setw(12) << k << ((++count % perline) ? " " : "\n");
  });
  std::cout << std::endl;
  fibonacci.close();                   // Close the file
}
```

这里用了一个 fstream 对象来封装文件，并且这个文件最初不存在。fstream 对象能够写和读文件，默认它只会打开存在的文件。指定的 ios_base::trunc 可以看作打开模式的标志，如果文件不存在，会导致生成文件；如果存在，会导致文件的内容被截断。Fibonacci 数增长很快，因此用 unsigned long long 作为值的类型，并限制个数为 50，不包含前两个。前两个数被定义为 first 和 second，并且会用 iter 将它们写到文件中，iter 是一个输出流迭

代器。这使文件的位置变为文件的第二个值的后面，因此 generate_n()算法写的 50 个值会跟在它后面。在值被写入之后，会调用 seekg()用于设置文件读指针的位置——将文件设置回开始位置，并且准备读取文件。应该使用 seekp()用于设置文件写指针的位置——来重置文件写入的位置。

用 for_each()算法将这个文件的内容写到标准输出流中。lambda 表达式会在 1 行中写 6 个值。可以用 generate_n()来写任意类型的值序列到可以用函数对象创建的文件中。假设需要一个包含正态分布的温度值的文件作为测试数据源。下面展示了如何用流迭代器和 generate_n()做到这些：

```
// Ex9_08.cpp
// Using stream iterators to create a file of random temperatures
#include <iostream>       // For standard streams
#include <iterator>       // For iterators and begin() and end()
#include <iomanip>        // For stream manipulators
#include <fstream>        // For file streams
#include <algorithm>      // For generate_n() and for_each()
#include <random>         // For distributions and random number generator
#include <string>         // For string class
using std::string;

int main()
{
  string file_name {"G:/Beginning_STL/temperatures.txt"};
std::ofstream temps_out {file_name, std::ios_base::out | std::ios_base::trunc};
const size_t n {50};                    // Number of temperatures required

std::random_device rd;                  // Non-determistic source
std::mt19937 rng {rd()};                // Mersenne twister generator
double mu {50.0}, sigma {15.0};         // Mean: 50 degrees SD: 15
std::normal_distribution<> normal {mu, sigma}; // Create distribution

// Write random temperatures to the file
std::generate_n(std::ostream_iterator<double> { temps_out, " "}, n,
  [&rng, &normal]
  {return normal(rng); });
temps_out.close();                      // Close the output file

// List the contents of the file
std::ifstream temps_in {file_name};  // Open the file to read it
for_each(std::istream_iterator<double>{temps_in}, std::istream_iterator
  <double> {},
  [](double t)
    {const size_t perline {10};
     static size_t count {};
     std::cout << std::fixed << std::setprecision(2) << std::setw(5) << t
               << ((++count % perline) ? " " : "\n");
    });
std::cout << std::endl;
```

```
  temps_in.close();                          // Close the input file
}
```

得到的输出如下：

```
59.61 53.71 42.76 61.45 48.43 43.48 59.09 36.76 62.12 35.13
55.85 58.72 35.34 39.95 49.31 33.42 41.88 46.63 57.89 32.39
52.36 49.56 68.11 44.49 49.72 48.30 33.48 77.92 58.02 19.17
47.75 31.14 24.13 37.18 44.04 30.64 65.47 55.15 68.73 54.17
62.88 35.45 70.11 9.67  25.89 39.71 72.83 90.08 57.25 51.40
```

这种工作方式和前面的示例 Ex9_07 相似，除了文件是用 ofstream 对象生成的，然后用 ifstream 对象来读。作为 generate_n()的最后一个参数，lambda 表达式会生成写到文件的值；它会返回一个随机浮点温度值，该值位于期望为 50、标准差为 15 的正态分布中。normal 对象定义了分布，rng 对象是随机数生成器。尽管可以将流迭代器用于 generate_n()，但不能用于 generate()算法，因为它要求的是正向迭代器。

9.5 流缓冲区迭代器

流缓冲区迭代器不同于流迭代器，流迭代器只会传送字符到流缓冲区或从流缓冲区读出字符。它们可以直接访问流的缓冲区，因此不包含插入和提取运算符。也没有数据的转换，数据之间也不需要分隔符，即使有分隔符，也可以自己处理它们。因为它们读写数据时没有数据转换，流缓冲区迭代器适用于二进制文件。流缓冲区迭代器读写字符的速度比流迭代器快。istreambuf_iterator 模板定义了输入迭代器，ostreambuf_iterator 模板定义了输出迭代器。可以构造读写任意 char、wchar_t、char16_t、char32 类型的字符的流缓冲区迭代器。

9.5.1 输入流缓冲区迭代器

为了生成可以从流中读取任意给定类型的字符输入流迭代器，需要将一个流对象传给构造函数：

```
std::istreambuf_iterator<char> in {std::cin};
```

这个对象是一个可以从标准输入流读取任意 char 类型字符的输入流缓冲区迭代器。默认构造函数生成的对象表示的是流结束迭代器：

```
std::istreambuf_iterator<char> end_in;
```

可以用这两个迭代器从 cin 中读取字符序列到字符串中，直到在单行中输入的 Ctrl+Z 被作为流结束的信号——例如：

```
std::cout << "Enter something: ";
string rubbish {in, end_in};
std::cout << rubbish << std::endl; // Whatever you enter will be output
```

用从键盘输入的全部字符来初始化 string 对象 rubbish，直到遇到流的结束符。

输入流缓冲区迭代器有下面这些成员函数：

- operator*()返回的是流中当前字符的副本。流的位置不会被提前，因此可以反复地获取当前字符。
- operator->()可以访问当前字符的成员——如果它有成员的话。
- operator++()和 operator++(int)都会将流的位置移到下一个字符。operator++()在移动位置后才返回流迭代器，operator++(int)在被移动位置之前返回流迭代器的一个代理。前缀++运算符很少使用。
- equal()接受另一个输入流缓冲区迭代器为参数，并返回 true，前提是当前迭代器和参数都不是流结束迭代器，或者都是流结束迭代器。如果它们之中只有一个流结束迭代器，会返回 false。

也有非成员函数，operator==()和 operator!=()，可以用来比较两个迭代器。我们不得不依靠流的结束来终止输入。可以用递增和解引用运算符来从流中读取字符，直到找到特定的字符。例如：

```
std::istreambuf_iterator<char> in {std::cin};
std::istreambuf_iterator<char> end_in;
char end_ch {'*'};
string rubbish;
while(in != end_in && *in != end_ch) rubbish += *in++;
std::cout << rubbish << std::endl; // Whatever you entered up to '*' or EOF
```

while 循环会从 cin 中读取字符，直到识别到流的结束符或者在回车后输入一个星号。循环体中会运用解引用运算符来返回流中的当前字符，然后，后缀自增运算符会移动迭代器，使它指向下一个字符。注意循环表达式中的解引用说明它没有改变迭代器；只要不是'*'，在迭代器递增之前，循环体中的相同字符会被再次读取。

9.5.2 输出流缓冲区迭代器

通过传给构造函数一个流对象，可以生成一个 ostreambuf_iterator 对象来将给定类型的字符写到流中：

```
string file_name {"G:/Beginning_STL/junk.txt"};
std::ofstream junk_out {file_name};
std::ostreambuf_iterator<char> out {junk_out};
```

输出对象可以将 char 类型的字符写到文件输出流 junk_out 中，junk_out 封装了名为 junk.txt 的文件。为了读取不同类型的字符，例如 char32_t，需要指定模板类型参数作为字符类型。当然，必须为字符类型生成流，因此不能用 ofstream，因为 ofstream 是 basic_ofstream<char>类型的别名。下面是一个示例：

```
string file_name {"G:/Beginning_STL/words.txt"};
std::basic_ofstream<char32_t> words_out {file_name};
```

```
std::ostreambuf_iterator<char32_t> out {words_out};
```

这个流缓冲区迭代器可以将 Unicode 字符写到流缓冲区。有 wchar_t 类型的字符的文件流被定义为别名 wofstream。

也可以通过将一个流缓冲区的地址传给构造函数来生成输出流缓冲区对象。可以这样来生成上面的 out 对象：

```
std::ostreambuf_iterator<char> out {junk_out.rdbuf()};
```

ofstream 对象的成员函数 rdbuf()会返回流内部的缓冲区的地址。成员函数 rdbuf()继承自 ios_base，它是所有流对象的基类。

ostreambuf_iterator 对象有下面这些成员函数：

- operator=()会将参数字符写到流缓冲区中。如果识别到 EOF，就说明流缓冲区是满的，这个写操作失败。
- 当上一次写缓冲区失败时，failed()返回 true。当识别到 EOF 时，会发生这种情况，因为输出流缓冲区是满的。
- operator*()不做任何事。之所以定义它，是因为它需要的 ostreambuf_iterator 对象是一个输出迭代器。
- operator++()和 operator++(int)不做任何事。之所以定义它们，是因为它们需要的 ostreambuf_iterator 对象是一个输出迭代器。

通常只关心成员函数赋值运算符。下面是使用它的一种方式：

```
string ad {"Now is the discount of our winter tents!\n"};
std::ostreambuf_iterator<char> iter {std::cout}; // Iterator for output to cout
for(auto ch: ad)
iter = ch;                                       // Write the character to the stream
```

执行这段代码会将字符串逐字节写到标准输出流中。当然，通过调用 copy()算法也可以得到同样的结果：

```
std::copy(std::begin(ad), std::end(ad), std::ostreambuf_iterator<char>
  {std::cout});
```

这两个示例中的编码方式有些滑稽可笑，因为它们等同于下面这行代码：

```
std::cout << ad;
```

尽管它并没有告诉我们关于输出流缓冲区迭代器的更多信息。

9.5.3 对文件流使用输出流缓冲区迭代器

可以用流缓冲区迭代器逐个字符地复制文件，并且没有格式化读写开销。下面是一个复制 dictionary.txt 的程序：

```
// Ex9_09.cpp
// Copying a file using stream buffer iterators
```

```
#include <iostream>                          // For standard streams
#include <iterator>                          // For iterators and begin() and end()
#include <fstream>                           // For file streams
#include <string>                            // For string class
using std::string;

int main()
{
  string file_name {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream file_in {file_name};
if(!file_in)
{
  std::cerr << file_name << " not open." << std::endl;
  exit(1);
}
string file_copy {"G:/Beginning_STL/dictionary_copy.txt"};
std::ofstream file_out {file_copy, std::ios_base::out | std::ios_base::trunc};

std::istreambuf_iterator<char> in {file_in};    // Input stream buffer iterator
std::istreambuf_iterator<char> end_in;          // End of stream buffer iterator
std::ostreambuf_iterator<char> out {file_out}; // Output stream buffer iterator
while(in != end_in)
  out = *in++;                                  // Copy character from in to out

std::cout << "File copy completed." << std::endl;

  file_in.close();                              // Close the file
  file_out.close();                             // Close the file
}
```

这会将 ifstream 对象封装的文件 file_in 复制到 ofstream 对象封装的文件 file_out 中，这是通过逐字符地复制输入文件流缓冲区到输出文件流缓冲区来复制输入文件的。while 循环是用流缓冲区对象 in 和 out 来实现复制的。解引用 in 会返回输入缓冲区中的当前字符，后缀++运算符会增加迭代器，使它指向输入缓冲区的下一个字符。输出流缓冲区对象的赋值运算符会保存输出流缓冲区中的右操作数字符，使迭代器前进，从而指向输出缓冲区的下一个位置。

这展示了流缓冲区对象的成员函数的直接用法，但也能用 copy()算法。可以用一条语句来代替 while 循环以及定义 in、end_in、out 的语句：

```
std::copy(std::istreambuf_iterator<char> {file_in},
          std::istreambuf_ iterator <char> {},
          std::ostreambuf_iterator <char>{file_out});
```

这会将前两个迭代器指定的序列复制到第三个参数指定的目的位置。文件缓冲区代表 Windows 上的整个文件流，在必要时，需要对它们做一些调整。因此，当输入缓冲区已经被读取时，会从流中补充它，并且当输出缓冲区满时，它会被写到输出流。

流缓冲区迭代器并不在乎原文件是如何被写的。可以将文件流定义为 wchar_t 字符流，它是两个字节的字符，例如：

```
std::wifstream file_in {file_name};
std::wofstream file_out {file_copy, std::ios_base::out | std::ios_base::trunc};
```

然后可以将原文件作为 wchar_t 字符来复制：

```
std::copy(std::istreambuf_iterator<wchar_t>{file_in},
          std::istreambuf_iterator <wchar_t>{},
          std::ostreambuf_itera tor<wchar_t> {file_out});
```

只在必要时改变流缓冲区迭代器的模板类型参数。

9.6 string 流、流，以及流缓冲区迭代器

可以用流迭代器和流缓冲区迭代器来传送数据和读取 string 流。string 流是表示内存中字符缓冲区中的 I/O 对象,是定义在 sstream 头文件中的 3 个模板中的一个模板的实例：

- basic_istringstream 支持从内存中的字符缓冲区读取数据。
- basic_ostringstream 支持写数据到内存中的字符缓冲区。
- basic_stringstream 支持字符缓冲区上的输入和输出操作。

字符数据类型是一个模板参数，对于 char 类型的 string 流有如下类型别名：istringstream、ostringstream 和 stringstream。这些对象的继承层次如图 9-2 所示。

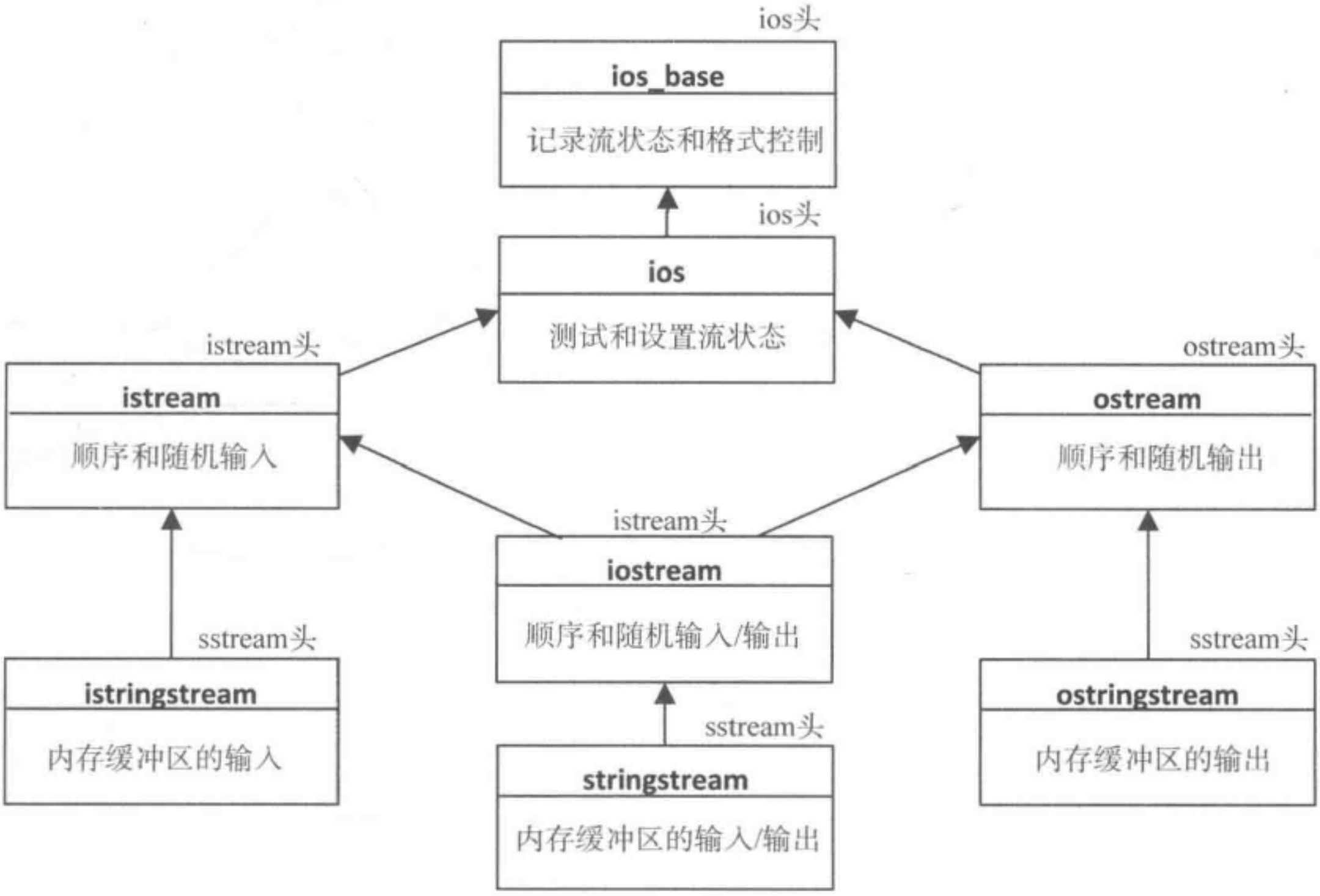


图 9-2 string 流类型的继承层次

你会注意到这里的直接和非直接基类与文件流的类型是相同的，这意味着对文件流做

的几乎任何事，同样也可以对 string 流做。可以用插入和提取运算符来格式化 string 流的 I/O；这意味着可以用流迭代器来读写它们。它们也支持文件流支持的无格式的 I/O 操作，因此可以用流缓冲区迭代器来读写它们。

为了保存 wchar_t 类型的字符，string 流也有一些别名，这些名称是 char 别名的名称加上前缀'w'。只使用 char 类型的 string 流，因为它们正是通常所需要的。

要能够为内存缓冲区上的 I/O 操作提供极大的灵活性。当需要读很多次数据时，从内存缓冲区读取数据的速度要比从外部设备读取快。一个出现这种情况的场景是，当输入流的内容是变动的时。需要多次读取来确定数据是什么。可以用一个有 string 流和 string 流迭代器的新版 ex9_03.cpp 来示范：

```
// Ex9_10.cpp
// Using a string stream as the dictionary source to anagrams of a word
#include <iostream>                     // For standard streams
#include <fstream>                      // For file streams
#include <iterator>                     // For iterators and begin() and end()
#include <string>                       // For string class
#include <set>                          // For set container
#include <vector>                       // For vector container
#include <algorithm>                    // For next_permutation()
#include <sstream>                      // For string streams
using std::string;

int main()
{
  string file_in {"G:/Beginning_STL/dictionary.txt"};
  std::ifstream in {file_in};
  if(!in)
  {
    std::cerr << file_in << " not open." << std::endl;
    exit(1);
  }
  std::stringstream instr;              // String stream for file contents
  std::copy(std::istreambuf_iterator<char>{in},
            std::istreambuf_iterator<char>(),
            std::ostreambuf_iterator<char>{instr});
  in.close();                           // Close the file

  std::vector<string> words;
  string word;
  auto end_iter = std::istream_iterator<string> {}; // End-of-stream iterator
  while(true)
  {
    std::cout << "\nEnter a word, or Ctrl+z to end: ";
    if((std::cin >> word).eof()) break;

  string word_copy {word};
  do
  {
```

```
        instr.clear();                          // Reset string stream EOF
        instr.seekg(0);                         // String stream position at beginning
                                                // Use find() to search instr for word
        if(std::find(std::istream_iterator<string>(instr), end_iter, word) !=
          end_iter)
        words.push_back(word);                  // Store the word found

        std::next_permutation(std::begin(word), std::end(word));
      } while(word != word_copy);

      std::copy(std::begin(words), std::end(words), std::ostream_iterator<string>
                {std::cout, " "});
        std::cout << std::endl;
        words.clear();                          // Remove previous anagrams
      }
    }
```

copy()算法会将 dictionary.txt 的全部内容复制到一个 stringstream 对象中。复制的过程使用了流缓冲区迭代器，因此没有包含数据转换——文件的字节被复制到 instr 对象中。当然，可以对流迭代器使用格式化的 I/O 操作，在这种情况下，复制操作将变为：

```
std::copy(std::istream_iterator<string>{in}, std::istream_iterator<string>(),
          std::ostream_iterator<string>{instr, " "});
```

这清楚地表明流迭代器适用于 string 流对象，但比前面的版本要慢。如下是一个将文件内容复制到 stringstream 对象的更快方法：

```
instr << in.rdbuf();
```

ifstream 对象的成员函数 rdbuf()会返回一个封装了文件内容的 basic_filebuf 对象的地址。basic_filebuf 以 basic_streambuf 为基类，并重载了插入运算符，用来将右操作数指向的 basic_streambuf 对象复制到左操作数 basic_ostream 对象中。这个操作很快，因为没有包含格式化或数据转换。

在 instr 中查找字谜和在文件流中查找是一样的，因为它是一个流——正好在内存中。读取 string 流会移动当前位置，所以在再次读取内容时，必须调用它的成员函数 seekg()来将位置重置为开始位置。类似地，在读到 instr 中数据的末尾时会设置 EOF 标志，必须调用成员函数 clear()来重置这个标志。如果不这么做，随后的读操作会失败。

下面是 Ex9_10.cpp 的一些示例输出：

```
Enter a word, or Ctrl+z to end: part
part prat rapt tarp trap

Enter a word, or Ctrl+z to end: painter
painter pertain repaint

Enter a word, or Ctrl+z to end: ^Z
```

在笔者系统上它要比 Ex9_04.cpp 快，但仍然不那么可观。分析 4 个字母的单词相当快，但分析 7 个字母的单词就会花费很长时间——并且比将文件内容读到 set 容器的 Ex9_03.cpp

版本要慢。除了分析 7 个字母的单词所花费的时间约是分析 4 个字母的单词的 210 倍这一事实之外，这也部分说明了用带有格式化输入的提取运算符会产生多少开销。它如此慢的另一个原因是：访问 set 容器查找单词使用的是二分查找，但在此处是从 string 流的开始位置顺序查找单词的。

9.7 本章小结

本章解释了 STL 辅助我们使用流的各种方式。流迭代器可以读写格式化的字符流，流缓冲区迭代器可以无转换地在内存和流之间传送数据。流迭代器是由类模板定义的。为了读取流，istream_iterator 定义了一个单向的输入迭代器。为了写入流，ostream_iterator 定义了一个单向的输出迭代器。读写的数据类型是由第一个模板类型参数定义的。第二个模板类型参数指定了流的字符类型，默认值为 char 类型。istreambuf_iterator 类模板定义了用来读取流的流缓冲区迭代器，ostreambuf_iterator 模板定义了用来写入流的迭代器。流中字符的类型是由第一个模板类型参数定义的，默认值为 char 类型。

可以用流的成员函数和流缓冲区迭代器来读写流，如示例中所示，但这很少需要，也不是我们想要的。直接对流使用流的提取和插入运算符更加简单和高效。这些迭代器主要用于算法。能够用输入流迭代器将一个文件的内容传送到算法中，并且可以用输出流迭代器将结果写到另一个文件中，这是一个非常强大的机制。流迭代器和流缓冲区迭代器能够极大的简化读写文件的代码，但和流类提供的 I/O 功能相比，需要付出增加执行时间的代价。在数据容量不是很大的地方，为了简化代码所付出的开销是有价值的。但是，在反复读写大容量的数据时，开销可能是无法接受的。

练习

1. 写一个以 std::pair<string, size_t>类型的对象保存一个人的名和年龄的程序。该程序应该能够读取任意个数的名/年龄对，并将它们写到一个输出文件中。然后关闭这个文件，将它作为输入文件打开，从文件中读取 pair 对象，将它们写到标准输出流中。所有的输入和输出都应该用流迭代器来执行。

2. 写一个程序，读取练习 1 生成的文件，并写一个以逆序包含 pair 对象的新文件。所有的输入和输出都应该使用流迭代器。

3. 写一个程序，用流缓冲区迭代器读取练习 1 生成的文件到一个 stringstream 对象。用输入流迭代器访问 stringstream 中的 string 和 size_t 的值，并以 pair 对象的形式将它们写到容器中；选择以姓名升序排列的 pair 对象的容器。为了说明一切都是正常运转的，用流迭代器输出容器的内容到标准输出流。

4. 用流迭代器将均匀分布在 0 到 10 亿之间的 100 个随机整数写到文件中。用算法和流迭代器来确定最小值和最大值，并计算出平均值。输出计算值，然后是文件中的值，每 8 个一行。用迭代器进行全部的输入和输出。

第 10 章

使用数值、时间和复数

本章介绍的 STL 支持的 3 类数据比 STL 支持的其他类数据更加专业。numeric 头文件定义了使数值数据处理更简单或高效的 STL 功能。chrono 头文件提供了处理时间的能力，包括时钟时间和时间间隔。最后，complex 头文件定义了支持复数处理的类模板。

本章将介绍以下内容：

- 如何生成保存数值数据的 valarray 对象
- 什么是 slice 对象，如何生成和使用它们
- 什么是 gslice 对象，如何使用它们
- ratio 类模板的作用以及如何使用
- 如何访问和使用硬件时钟。
- 如何生成封装了复数且将之应用到它们上的操作的对象

10.1 数值计算

数值计算的效率对于很多工程、科学、数学场景都具有重要意义。虽然这些场景都是特定的，但也有很多比较常见的涉及密集数值计算的应用场景。音频处理，例如语音识别或数码录音，都涉及数字滤波—— 一个处理器密集处理过程。多媒体设备也广泛应用了数字图像的处理，例如 CT 和 MRI 扫描器。但在其他常见应用中—— 如果我们曾经用编辑套件修改过照片，就能体会到这些操作要花多长时间，数值计算的高效执行对于大多数程序来说，都是至关重要的。下一节是关于数值计算的 STL 算法的，其中的一些我们已经介绍过。在那之后，会介绍一个被设计用来尽可能高效地对数组进行数值计算的类模板。

10.2 数值算法

在本章偶尔会用到矩阵(复数矩阵)和 vector。在数学和科学中，矩阵是数的二维数组。

vector 是一维数组——数的线性序列。一般情况下，当在本文中使用术语 vector 时，指的是数的一维数组而不是 vector 容器，但也能够是 vector 容器。矩阵一般用来保存 valarray 对象，在解释这个算法之后，就会用到它。你可能已经见过一些数值数据的 STL 算法，但在本章还会介绍它们和一些新的算法。所有这些算法都可以用于输入迭代器指定的序列。

10.2.1 保存序列中的增量值

定义在 numeric 头文件中的 itoa()函数模板会用连续的 T 类型值填充序列。前两个参数是定义序列的正向迭代器，第三个参数是初始的 T 值。第三个指定的值会被保存到序列的第一个元素中。保存在第一个元素后的值是通过对前面的值运用自增运算符得到的。当然，这意味着 T 类型必须支持 operator++()。下面展示了如何生成一个有连续的浮点值元素的 vector 容器：

```
std::vector<double> data(9);
double initial {-4};
std::iota(std::begin(data), std::end(data), initial);
std::copy(std::begin(data), std::end(data),
std::ostream_iterator<double>{std::cout
  << std::fixed << std::setprecision (1), " "});
std::cout << std::endl;          // -4.0 -3.0 -2.0 -1.0 0.0 1.0 2.0 3.0 4.0
```

以 4 为初始值调用 iota()会将 data 中元素的值设为从-4 到+4 的连续值。

当然，初始值并不需要一定是整数：

```
std::iota(std::begin(data), std::end(data), -2.5);
                        // Values are -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5 4.5 5.5
```

增量是 1，因此 data 中的值和注释显示的一样。可以将 iota()算法应用到任意类型的序列上，只要它有自增运算符。下面是另一个示例：

```
string text {"This is text"};
std::iota(std::begin(text), std::end(text), 'K');
std::cout << text << std::endl;              // Outputs: KLMNOPQRSTUV
```

很容易看到输出如注释所示——字符串中的字符被设为以 K 开头的字符序列。这个示例发生了什么并不是那么明显：

```
std::vector<string> words (8);
std::iota(std::begin(words), std::end(words), "mysterious");
std::copy(std::begin(words), std::end(words),std::ostream_ iterator<string>
  {std::cout, " "});
std::cout << std::endl; // mysterious ysterious sterious terious erious rious
                        // ious ous
```

输出如注释所示。这是该算法的一个有趣应用，但没有什么用处。这只适用于第三个参数是一个字符串常量的情形。如果参数是 string{"mysterious"}，将无法通过编译，因为

没有为 string 类定义 operator++()。字符串常量对应的值是一个 const char*类型的指针，可以将++运算符应用到它上面。因此对于 words 中第一个元素后的每个元素，指针的递增会导致字符串常量前面的字母被丢弃。将++应用到指针的结果是生成一个 string 对象，然后它会被保存到当前的元素序列中。只要++可以应用到序列中的元素类型上，就能将 iota()算法应用到序列上。

■ **注意**：*很有趣的是，iota()算法来源于 IBM 的编程语言 APL 中的 iota 运算符 ι。在 APL 中，表达式 ι_{10} 会生成从 1 到 10 的整数的 vector。APL 是肯・艾弗森在 20 世纪 60 年代发明的。它是一门相当简洁的语言，能够隐式处理 vector 和数组。APL 的一个完整程序会从键盘读取任意个值，计算出它们的平均值，然后输出被表示为 10 个字符结果。*

10.2.2　求序列的和

我们已经介绍过 accumulate()算法的基本版本，可以用+运算符求出元素序列的和。前两个参数是定义序列的输入迭代器，第三个参数是和的初值；第三个参数的类型决定了返回值的类型。第二个版本的第 4 个参数是定义应用到总数和元素之间的二元函数对象。这时，我们在必要时可以定义自己的加法运算。例如：

```
std::vector<int> values {2, 0, 12, 3, 5, 0, 2, 7, 0, 8};
int min {3};
auto sum = std::accumulate(std::begin(values), std::end(values), 0, [min](int
  sum, int v)
                                                    {
                                                    if(v < min)
                                                      return sum;
                                                    return sum + v;
                                                    });
std::cout << "The sum of the elements greater than " << min-1
          <<" is " << sum << std::endl;                          // 35
```

这里忽略了值小于 3 的元素。这个条件可以尽可能复杂，因此，我们能够求出指定范围内的元素之和。这个运算并不一定要是加法，可以是任何不修改操作数或不使定义范围的迭代器无效的运算。例如，为数值元素定义的乘法运算函数会生成元素的乘积，只要初值为 1。实现浮点元素除法的函数会生成元素乘积的倒数，只要初值为 1。下面展示了如何生成元素的乘积：

```
std::vector<int> values {2, 3, 5, 7, 11, 13};
auto product = std::accumulate(std::begin(values), std::end(values), 1,
                  std::multiplies<int>()); // 30030
```

这里用来自于 functional 头文件的函数作为第 4 个参数。如果有值为 0 的元素，可以像上一个代码段中的 lambda 表达式那样忽略它们。

string 类支持加法，因此可以将 accumulate()应用到 string 对象的序列上：

```
std::vector<string> numbers {"one", "two", "three", "four", "five",
                             "six", "seven", "eight", "nine", "ten"};
auto s = std::accumulate(std::begin(numbers), std::end(numbers), string{},
                         [](string& str, string& element)
                           {
                             if(element[0] == 't') return str + ' ' + element;
                             return str;
                             });          // Result: " two three ten"
```

这段代码连接了以't'开头的 string 对象，并用空格将它们隔开。cumulate()算法得到的结果可能和它所应用的序列中的元素类型不同：

```
std::vector<int> numbers {1, 2, 3, 10, 11, 12};
auto s = std::accumulate(std::begin(numbers), std::end(numbers), string
  {"The numbers are"},[](string& str, int n)
                                  { return str + ": " + std::to_string(n); });
std::cout << s << std::endl;   // Output: The numbers are: 1: 2: 3: 10: 11: 12
```

lambda 表达式使用的 to_string()函数会返回一个数值参数的 string 形式，所以应用 accumulate()到这里的整数序列会返回注释中显示的 string。

10.2.3 内积

两个 vector 的内积是对应元素的乘积之和。为了能够得到内积，vector 的长度必须相同。内积是矩阵算术的基本运算。两个矩阵的乘积是一个矩阵，它是由第一个矩阵的每一行乘以第二个矩阵的每一列得到的，如图 10-1 所示。

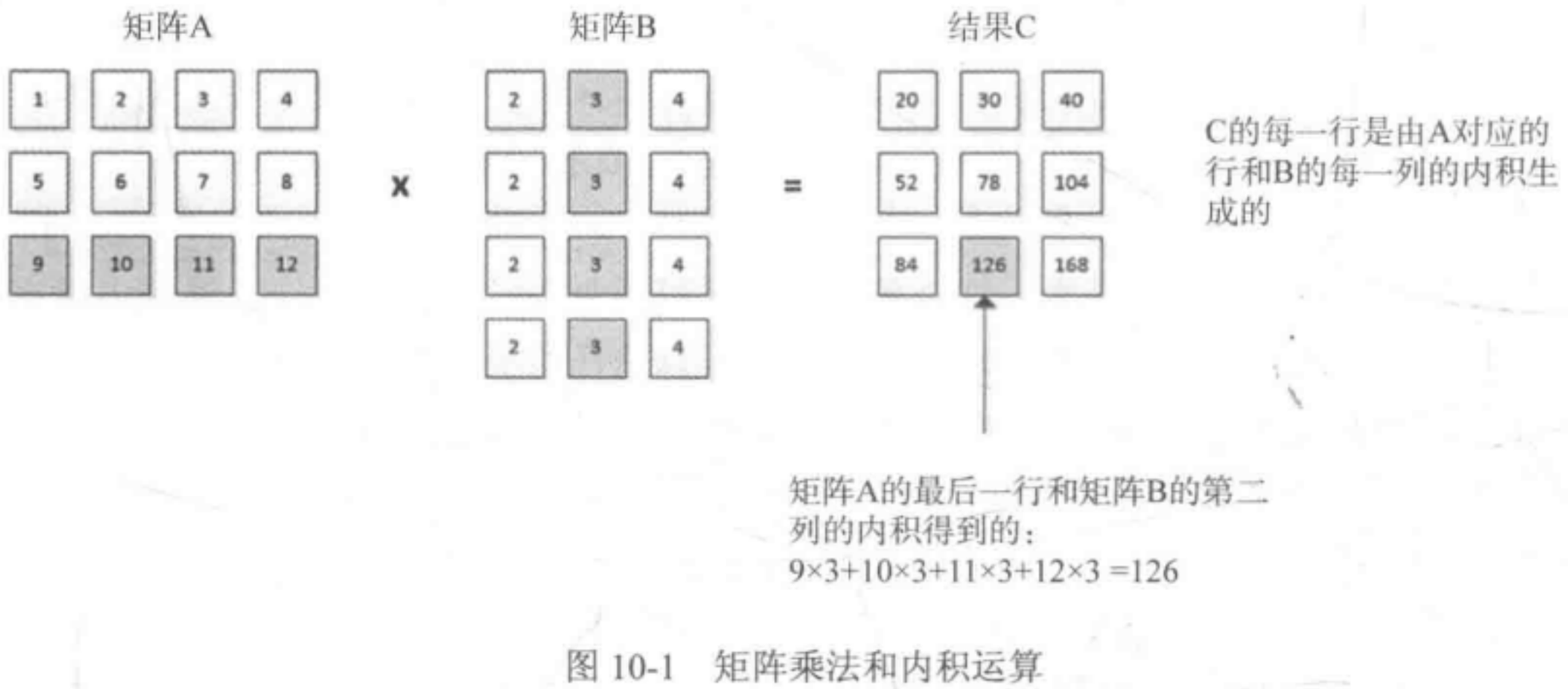


图 10-1　矩阵乘法和内积运算

为了使矩阵内积成为可能，左操作数(矩阵 A)的列数必须和右操作数(矩阵 B)的行数相同。如果左操作数有 m 行和 n 列(m×n 的矩阵)，右操作数有 n 行和 k 列(n×k 的矩阵)，结果是一个 m 行和 k 列的矩阵(m×k 的矩阵)。

定义在 numeric 头文件中的 inner_product()算法可以计算两个 vector 的内积。这个函数模板有 4 个参数：前两个参数是定义第 1 个 vector 的输入迭代器，第 3 个参数是确定第 2

个 vector 的开始输入迭代器，第 4 个参数是和的初值。算法会返回 vector 的内积。例如：

```
std::vector<int> v1(10);
std::vector<int> v2(10);
std::iota(std::begin(v1), std::end(v1), 2);    // 2 3 4 5 6 7 8 9 10 11
std::iota(std::begin(v2), std::end(v2), 3);    // 3 4 5 6 7 8 9 10 11 12
std::cout << std::inner_product(std::begin(v1), std::end(v1),
                std::begin(v2), 0)
          << std::endl;                                  // Output: 570
```

因为两个 vector 的内积的标准定义，内积的初值为 0，但可以选择为对应元素的乘积之和指定不同的初值。在使用 inner_product()时，使用正确的类型很重要，如下所示：

```
std::vector<double> data {0.5, 0.75, 0.85};
auto result1 = std::inner_product(std::begin(data), std::end(data),
               std:: begin(data), 0);
double result2 = std::inner_product(std::begin(data), std::end(data),
                 std:: begin(data), 0);
auto result3 = std::inner_product(std::begin(data), std::end(data),
                std::begin(data), 0.0);
std::cout << result1 << " " << result2
          << " " << result3 << std::endl;              // Output: 0 0 1.535
```

第二条和第三条语句显然做的是同样的事，但返回的类型是由第 4 个参数决定的。即使迭代器指向的是浮点参数，当内积的初值是整数类型时，对应元素相乘的结果的组合运算适用的是整数运算。这同样适用于 accumulate()算法，因此需要保证内积的初值是合适的类型。幸运的是，当初值的类型和这个运算所涉及的元素不同时，大多数编译器会发出警告。我们可以在一个示例中尝试使用 inner_product()算法和其他的一些算法。

1. 应用内积

最小二乘线性回归是求系数的一种方法，对于行 a、b、$y=ax+b$，这最适合通过一组点 (x,y)，这些点通常是现实世界中的某种数据样本。这个方法来自高斯，找到 a 和 b 的系数，这样样本点到行的垂直距离平方和是最小的。下面介绍一个可以做到这些的等式，不需要知道这些等式是如何得到的，但是如果不想和任何数学打交道，可以跳过它直接看代码。

给定 n 个点(x_i,y_i)，这个方法涉及求解下面的等式：

$$nb + a\sum x_i = \sum y_i$$

$$b\sum x_i + a\sum x_i^2 = \sum x_i y_i$$

解出等式的系数 a 和 b：

$$a = \frac{\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - \left(\sum x_i\right)^2}$$

$$b = \mu_y - a\mu_x$$

如果能随着 x 和 y 的平均值计算出各种和，就能将它代入这些方程中，得到回归直线的系数。在第 8 章看到，变量 x 的 n 个值的平均值 u 的等式是：

$$\mu_x = \frac{\sum x_i}{n}$$

显然，accumulate()和 inner_product()算法对这些是非常有帮助的。

这个示例会生成与从文件中得到的一组数据点相拟合的直线。文件在下载的代码中，记录了每千瓦时的耗电成本和欧洲几个国家平均每人的可再生能源发电装机瓦数。程序的输出应该显示配置的可再生能源的容量和成本是否是线性关系。下面是代码：

```
// Ex10_01.cpp
// Least squares regression
#include <numeric>                  // For accumulate(), inner_product()
#include <vector>                   // For vector container
#include <iostream>                 // For standard streams
#include <iomanip>                  // For stream manipulators
#include <fstream>                  // For file streams
#include <iterator>                 // For iterators and begin() and end()
#include <string>                   // For string class
using std::string;

int main()
{
  // File contains country_name renewables_per_person kwh_cost
string file_in {"G:/Beginning_STL/renewables_vs_kwh_cost.txt"};
std::ifstream in {file_in};

if(!in)                             // Verify we have a file
{
  std::cerr << file_in << " not open." << std::endl;
  exit(1);
}

std::vector<double> x;              // Renewables per head
std::vector<double> y;              // Corresponding cost for.a kilowatt hour
// Read the file and show the data
std::cout << " Country " << " Watts per Head " << " kwh cost(cents) "
          << std::endl;
while(true)
{
string country;
double renewables {};
double kwh_cost {};

if((in >> country).eof()) break;      // EOF read - we are done
in >> renewables >> kwh_cost;
x.push_back(renewables);
y.push_back(kwh_cost);
std::cout << std::left << std::setw(12) << country // Output the record
```

```
                << std::right
                <<std::fixed<<std::setprecision(2)<<std::setw(12)<<renewables
                << std::setw(16) << kwh_cost << std::endl;
}
auto n = x.size();                                  // Number of points
auto sx = std::accumulate(std::begin(x), std::end(x), 0.0); // Sum of x values
auto sy = std::accumulate(std::begin(y), std::end(y), 0.0); // Sum of y values
auto mean_x = sx/n;                                 // Mean of x values
auto mean_y = sy/n;                                 // Mean of y values

// Sum of x*y values and sum of x-squared
auto sxy = std::inner_product(std::begin(x), std::end(x), std::begin(y), 0.0);
auto sx_2 = std::inner_product(std::begin(x), std::end(x), std::begin(x), 0.0);

double a {}, b {};                                  // Line coefficients
auto num = n*sxy - sx*sy;                           // Numerator for a
auto denom = n*sx_2 - sx*sx;                        // Denominator for a
a = num / denom;
b = mean_y - a*mean_x;
std::cout << std:: fixed << std::setprecision(3) << "\ny = " // Output equation
          << a << "*x + " << b << std::endl; // for regression line
}
```

在 while 循环中读取了文件。只保存数值，每一个都完整记录了国家名称、人均新再生能源装机量的瓦特数，每千瓦时花费的成本则以美分的形式被写到标准输出流中。对于保存在 vector 容器中的两个值：x 记录的是每个国家人均的可再生容量，y 记录的是对应的千瓦时的成本。

x 和 y 的平均值是由 accumulate()算法通过算出每个容器中的元素之和计算得出的，然后用结果除以元素的个数。x 的平方之和与 xy 的内积之和是通过 inner_product()算法计算得出的。通过使用前面展示的等式，可以用这些结果算出直线的系数 a 和 b。

注意，可以简化系数 a 的等式。如果将分母和分子除以 n^2，等式就可以写为：

$$a=\frac{\sum x_i y_i / n-\mu_x \mu_y}{\sum x_i^2 / n-\mu_x^2}$$

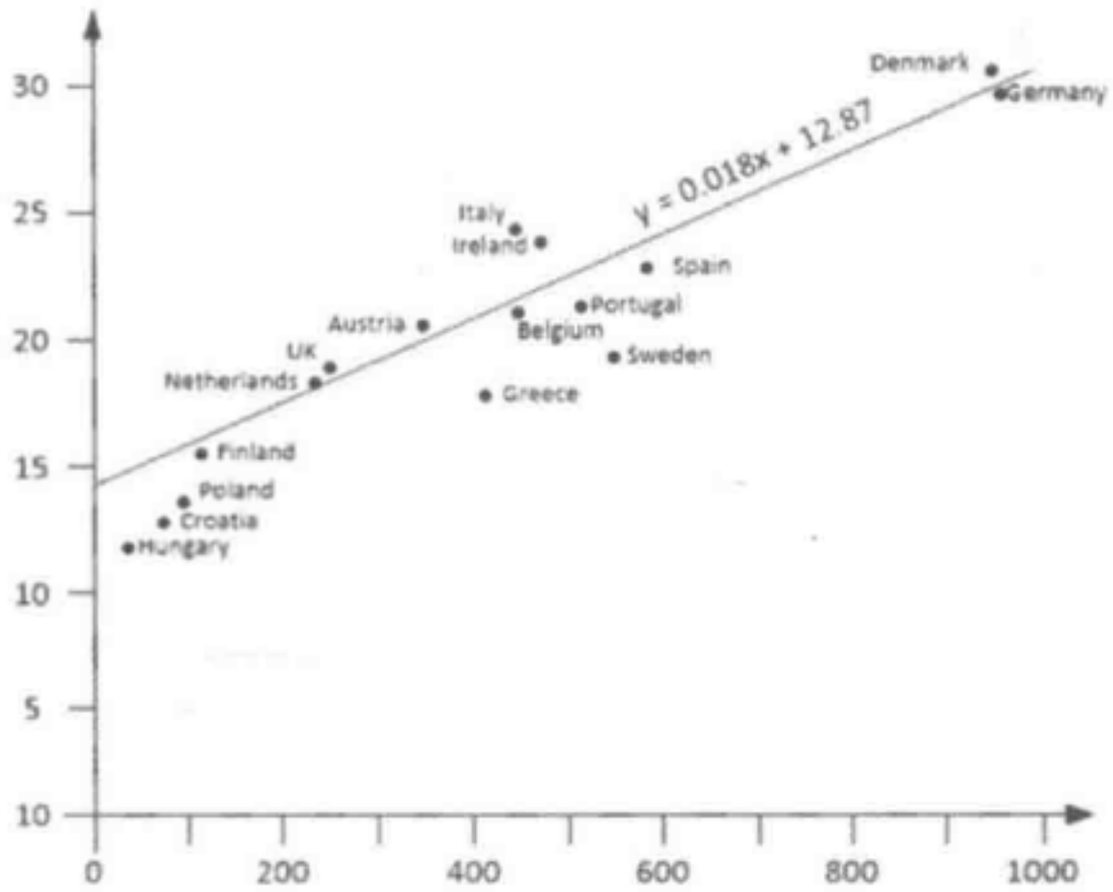
现在 x 的值和 y 的值不再显式需要了。计算系数的代码可以写为：

```
auto n = x.size(); // Number of points
// Calculate mean values for x, y, xy, and x-squared
auto mean_x = std::accumulate(std::begin(x), std::end(x), 0.0)/n;
auto mean_y = std::accumulate(std::begin(y), std::end(y), 0.0)/n;
auto mean_xy = std::inner_product(std::begin(x), std::end(x),
               std::begin (y), 0.0)/n;
auto mean_x2 = std::inner_product(std::begin(x), std::end(x),
               std::begin (x), 0.0)/n;

// Calculate coefficients
auto a = (mean_xy - mean_x*mean_y)/(mean_x2 - mean_x*mean_x);
```

```
auto b = mean_y - a*mean_x;
```

这里用更少的代码实现了相同的结果。图 10-2 的右边显示了程序的输出，左边是回归直线和原始数据点的图形。



Country	Watts per Head	kwh cost(cents)
Austria	353.00	20.21
Belgium	452.00	20.97
Denmark	956.00	30.42
Croatia	89.00	13.12
Finland	114.00	15.63
Germany	969.00	29.81
Greece	424.00	17.67
Hungary	37.00	12.02
Ireland	483.00	24.07
Italy	453.00	24.46
Netherland	234.00	18.21
Poland	100.00	14.21
Portugal	517.00	21.75
Spain	596.00	22.52
Sweden	561.00	19.67
UK	257.00	19.18

y = 0.018*x + 12.870

图 10-2　最小二乘线性回归的结果

绘制的图形相当有说服力——原始点很接近它。看起来好像人均每增加 100 瓦的可再生能源发电量，每千瓦时需要付出几乎两美分的成本。

2. 定义内积运算

之前见到的 inner_product()版本将两个输入序列的对应元素相乘，然后算出总数。第二个版本有两个以上的定义函数对象的参数。第二个函数对象定义了运用到两个序列中对应元素的二元运算，以及第一个用来代替加法合并结果的二元运算。我们提供的作为参数的函数对象不能使任何迭代器无效，也不能修改输入序列的元素。下面展示了如何生成和的积而不是积的和：

```
std::vector<int> v1(5);
std::vector<int> v2(5);
std::iota(std::begin(v1), std::end(v1), 2);          // 2 3 4 5 6
std::iota(std::begin(v2), std::end(v2), 3);          // 3 4 5 6 7
std::cout << std::inner_product(std::begin(v1), std::end(v1), std::begin(v2), 1,
             std::multiplies<>(), std::plus <>())
          << std::endl;                              // Output: 45045
```

inner_product()调用中，作为参数的函数对象被定义在 functional 头文件中。plus<T>对象会计算出两个 T 类型值的和，这里定义运算的模板实例会被应用到来自于输入序列的 int 类型的对应元素上。作为 inner_product()的第 5 个参数的 multiples 实例，它会将乘法结果累计起来，注意因为结果是乘积，如果不想结果总是为 0，初值必须不为 0。Functional 头文件中也定义了可以用于 inner_product()的其他二元算术运算——减、除、求余。也可以用定义了位运算的函数对象的模板；它们是 bit_and、bit_or、bit_eor。

10.2.4　相邻差

numeric 头文件中的 adjacent_difference()算法可以算出输入序列中相邻元素对的差，并将它们保存到另一个序列中。第一个元素会被原封不动地复制到新的序列中，然后用第二个元素减去第一个元素的结果作为新序列的第二个元素，再用第三个元素减去第二个元素的结果作为新序列的第三个元素，以此类推。例如：

```
std::vector<int> data {2, 3, 5, 7, 11, 13, 17, 19};
std::cout << "Differences: ";
std::adjacent_difference(std::begin(data), std::end(data),
                         std::ostream_iterator<int>{std::cout, " "});
std::cout << std::endl;                          // Differences: 2 1 2 2 4 2 4 2
```

data 容器中元素之间的差值被 adjacent_difference()算法直接输出，因为输出序列的迭代器是一个写到 cout 的输出流迭代器。这里产生的输出如注释所示。

这个算法的第二个版本允许指定一个运算符来代替应用到元素对上的减法运算符。下面是一个示例：

```
std::vector<int> data {2, 3, 5, 7, 11, 13, 17, 19};
std::cout << "Products: ";
std::adjacent_difference(std::begin(data), std::end(data),
                         std::ostream_iterator<int>{std::cout, " "},
                         std::multiplies<>());
std::cout << std::endl;                          // Products: 2 6 15 35 77 143 221 323
```

第 4 个参数是一个指定元素之间运算的函数对象——在这个示例中是一个来自于 functional 头文件的 multiplies 实例。可以看到，这样会得到 data 中连续元素的乘积。这个函数可以接受任何二元运算，只要不改变输入序列或者使迭代器无效。下面是一个将 plus<T>函数对象用作元素对之间的运算符来计算 Fibonacci 数列的示例：

```
std::vector<size_t> fib(15, 1); // 15 elements initialized with 1
std::adjacent_difference(std::begin(fib), std::end(fib)-1, std::begin(fib)+1,
                         std::plus<size_t>());
std::copy(std::begin(fib), std::end(fib),
          std::ostream_iterator<size_t>{std::cout, " "});
std::cout << std::endl; // Output: 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

这里 adjacent_difference()算法会对 fib 容器中的元素对进行相加，并将结果写回从第二个元素开始的 fib 容器。fib 的最后一个元素没有包含输入序列，输入序列中最后两个元素的和会覆盖最后一个元素的值。在这个运算之后，fib 会包含一个从 1 开始的 Fibonacci 数列。注释显示了 copy()算法生成的输出。

10.2.5　部分和

定义在 header 头文件中的 partial_sum()可以计算输入序列中元素的部分和，并将结果

保存到一个输出序列中。它会计算出输入序列中长度从 1 开始不断增加的序列的和，所以第一个输出值就是第一个元素，下一个值是前两个元素的和，再下一个值就是前三个元素的和，以此类推。这和 adjacent_difference()算法是相反的，因此 partial_sum()不会做 adjacent_difference()做的事。下面是一个示例：

```
std::vector<int> data {2, 3, 5, 7, 11, 13, 17, 19};
std::cout << "Partial sums: ";
std::partial_sum(std::begin(data), std::end(data),
                 std::ostream_ iterator<int>{std::cout, " "});
std::cout << std::endl; // Partial sums: 2 5 10 17 28 41 58 77
```

可以看到，输出是由长度稳定增加的序列的和组成的。通过执行下面的代码，可以很容易展示出这些结果：

```
std::vector<int> data {2, 3, 5, 7, 11, 13, 17, 19};
std::cout << "Original data: ";
std::copy(std::begin(data), std::end(data), std::ostream_ iterator<int>
  {std::cout, " "});
std::adjacent_difference(std::begin(data), std::end(data), std::begin (data));
std::cout << "\nDifferences: ";
std::copy(std::begin(data), std::end(data), std::ostream_iterator<int>
  {std::cout, " "});
std::cout << "\nPartial sums: ";
std::partial_sum(std::begin(data), std::end(data), std::ostream_iterator
  <int>{std::cout, " "});
std::cout << std::endl;
```

注意，这里的输出迭代器和输入序列的开始迭代器是相同的。这是合法的，可以认为 data 是可覆盖的，但算法被定义为阻止这么做。执行这段代码后得到的输出如下：

```
Original data: 2 3 5 7 11 13 17 19
Differences: 2 1 2 2 4 2 4 2
Partial sums: 2 3 5 7 11 13 17 19
```

输出显示了用原始值的差值计算出的部分和，这一点不令人吃惊。

像 adjacent_difference()算法一样，可以提供一个函数对象作为 partial_sum()的额外参数，额外参数被用来定义一个代替加法的运算符。下面是它可能的运用方式：

```
std::vector<int> data {2, 3, 5, 7, 11, 13, 17, 19};
std::cout << "Partial sums: ";
std::partial_sum(std::begin(data), std::end(data),
    std::ostream_iterator<int>{std::cout, " "}, std::minus<int>());
std::cout << std::endl;       // Partial sums: 2 -1 -6 -13 -24 -37 -54 -73
```

这里使用了减法运算符，所以值是 2、2-3、2-3-5、2-3-5-7 的结果，以此类推。

10.2.6　极大值和极小值

前面介绍了一些用来确定极小值和极大值的算法，但本章无论如何都会包含它们。algorithm 头文件中定义了 3 个可以运用到序列的算法：min_element()会返回一个指向输入序列的最小元素的迭代器，max_element()会返回指向最大元素的迭代器，minmax_element()会以 pair 对象的形式返回这两个迭代器。序列必须由正向迭代器指定，仅仅有输入迭代器是不够的。对于这 3 个算法，除了序列的开始和结束迭代器，可以选择提供第三个参数来定义比较函数。下面是将这 3 个算法应用到整数的一些代码：

```
std::vector<int> data {2, 12, 3, 5, 17, -11, 113, 117, 19};

std::cout << "From values ";
std::copy(std::begin(data), std::end(data), std::ostream_iterator<int>{std::
  cout, " "});
std::cout << "\n Min = " << *std::min_element(std::begin(data), std::end
  (data))
          << " Max = " << *std::max_element(std::begin(data),
                                            std::end(data))
          << std::endl;

auto start_iter = std::begin(data) + 2;
auto end_iter = std::end(data) - 2;
auto pr = std::minmax_element(start_iter, end_iter); // Get min and max

std::cout << "From values ";
std::copy(start_iter, end_iter, std::ostream_iterator<int>{std::cout, " "});
std::cout << "\n Min = " << *pr.first << " Max = " << *pr.second << std::endl;
```

min_element()和 max_element()被用来从 data 中查找最小值和最大值。minmax_element()会被应用到同一个序列上，但会略过前两个和后两个元素。执行这段代码会输出下面的内容：

```
From values 2 12 3 5 17 -11 113 117 19
      Min = -11 Max = 117
From values 3 5 17 -11 113
      Min = -11 Max = 113
```

algorithm 头文件中也定义了 min()、max()、minmax()的函数模板，它们分别返回最小值、最大值或者两个对象或一个对象的初始化列表的最小值和最大值。我们已经看到它们可以用来比较两个参数。下面是一个将它们用于初始化列表的示例：

```
auto words = {string {"one"}, string {"two"}, string {"three"}, string {"four"},
  string {"five"}, string {"six"}, string {"seven"}, string {"eight"}};
std::cout << "Min = " << std::min(words)
          << std::endl;                          // Min = eight

auto pr = std::minmax(words, [] (const string& s1, const string& s2)
                                {return s1.back() < s2.back();});
std::cout << "Min = " << pr.first << " Max = " << pr.second
          << std::endl;                          // Min = one Max = six
```

words 是 string 对象的一个初始化列表。元素都是 string 对象很重要。如果使用简单的 char*，那么算法将无法正常工作，因为比较的是地址而不是 string 的内容。min()算法会用 string 对象默认的 operator<()来确定 words 中的最小对象，然后用 minmax()找出列表中最小和最大对象，minmax()使用一个自定义的比较字符串的最后一个字符的比较函数。结果显示在注释中。也有接受函数对象作为定义比较的最后一个参数的 min()和 max()版本。

10.3 保存和处理数值

定义在 valarray 头文件中的 valarray 类模板定义了保存和操作数值序列的对象的类型，主要用来处理整数和浮点数，但也能够用来保存类类型的对象，只要类满足一些条件：

- 类不能是抽象的。
- public 构造函数必须包含默认的构造函数和拷贝构造函数。
- 析构函数必须是 public。
- 类必须定义赋值运算符，而且必须是 public。
- 类不能重载 operator&()。
- 成员函数不能抛出异常。

不能保存引用或 valarray 中用 const、volatile 修饰的对象。如果类满足所有这些约束，就可以使用它了。

valarray 模板为数值数据处理提供的功能比任何序列容器(例如 vector)都多。首先，最重要的是，它被设计为允许编译器以一种不应用到序列容器的方式来优化它的操作性能。但是，编译器是否优化并不依赖 valarray 操作的实现。其次，有相当数量的一元和二元运算都是应用到 valarray 对象的内置类型上的。然后，有相当数量的内置一元函数可以将定义在 cmath 头文件中的运算应用到每个元素上。最后，valarra 类型内置提供了将数据作为多维数组使用的能力。

生成一个 valarray 对象很容易。下面是一些示例：

```
std::valarray<int> numbers(15); // 15 elements with default initial values 0
std::valarray<size_t> sizes {1, 2, 3};     // 3 elements with values 1 2 and 3
std::valarray<size_t> copy_sizes {sizes};  // 3 elements with values 1 2 and 3
std::valarray<double> values;              // Empty array
std::valarray<double> data(3.14, 10);      // 10 elements with values 3.14
```

每个构造函数都生成了有给定元素数目的对象。在最后一条语句中，使用圆括号来定义 data 是必要的；如果使用花括号，data 会包含 3.14 和 10 两个元素。也可以用从普通数组得到的一定个数的值来初始化 valarray 对象。例如：

```
int vals[] {2, 4, 6, 8, 10, 12, 14};
std::valarray<int> vals1 {vals, 5};        // 5 elements from vals: 2 4 6 8 10
std::valarray<int> vals2 {vals + 1, 4};  // 4 elements from vals: 4 6 8 10
```

后面会介绍其他的构造函数，因为它们有一些必须解释的参数类型。

10.3.1　valarray 对象的基本操作

valarray 对象和 array 容器一样，不能添加或删除元素。但是，能够改变 valarray 容器中的元素个数，为它们赋新值。例如：

```
data.resize(50, 1.5); // 50 elements with value 1.5
```

在这个操作之前，如果 data 中保存有元素，就会丢失它们的值。当需要得到元素的个数时，可以调用成员函数 size()。

成员函数 swap()可以交换当前对象和作为参数传入的另一个 valarray 对象的元素。例如：

```
std::valarray<size_t> sizes_3 {1, 2, 3};
std::valarray<size_t> sizes_4 {2, 3, 4, 5};
sizes_3.swap(sizes_4);    // sizes_3 now has 4 elements and sizes_4 has 3
```

valarray 对象中包含的元素个数可以不同，但显然两个对象中的元素必须是相同类型的。成员函数 swap()没有返回值。非成员函数 swap()函数模板做的事是一样的，因此最后一条语句可以替换为：

```
std::swap(sizes_3, sizes_4); // Calls sizes_3.swap(sizes_4)
```

可以调用 valarray 的成员函数 min()和 max()来查找元素的最小值和最大值。例如：

```
std::cout << "The elements are from " << sizes_4.min() << " to " << sizes_4.max()
          << '\n';
```

为了能够正常工作，元素必须是支持 operator<()的类型。

成员函数 sum()会返回元素的和，它是使用+=运算符计算出来的。因此，可以在 valarray 中按如下方式计算元素的平均值：

```
std::cout << "The average of the elements " << sizes_4.sum()/sizes_4.size()
          << '\n';
```

这比使用 accumulate()算法要简单得多。

valarray 没有返回元素迭代器的成员函数，但有专门的非成员函数版本的 begin()和 end()可以返回随机访问迭代器。这使我们能够使用基于范围的 for 循环来访问 valarray 中的元素，并且可以将算法应用到元素上；后面你会看到一些示例。不能对 valarray 使用插入迭代器，因为没有做这些事的成员函数，这也是它的大小不变的原因。

有两个成员函数可以对元素进行移位——是对序列进行移位而不是移位单个元素值中的比特。首先，成员函数 shift()会将全部的元素序列移动由参数指定的位数。函数会返回一个新的 valarray 对象作为结果，保持原序列不变。如果参数是正数，元素会被左移；如果是负数，会右移元素。这看起来很像位移。元素被移位之后，序列的左边或右边会为 0 或其他等同的类型。当然，如果不将移位操作之后的结果存回原容器，原对象是不会改变的。下面是一些展示它如何工作的代码：

```
std::valarray<int> d1 {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
auto d2 = d1.shift(2);                    // Shift left 2 positions
for(int n : d2) std::cout << n << ' ';
std::cout << '\n';                        // Result: 3 4 5 6 7 8 9 0 0
auto d3 = d1.shift(-3);                   // Shift right 3 positions
std::copy(std::begin(d3), std::end(d3),
          std::ostream_iterator<int>{std::cout, " "});
std::cout << std::endl;                   // Result: 0 0 0 1 2 3 4 5 6
```

注释解释发生了什么。为了展示可以使用的输出方式，在两个示例中用了不同的方式来输出结果。valarray 模板为对象定义了赋值运算符，所以如果想替换原始序列，可以这样写：

```
d1 = d1.shift(2); // Shift d1 left 2 positions
```

元素移位的第二种方式是使用成员函数 cshift()，它会将元素序列循环移动由参数指定的数目的位置。序列会从左边或右边循环移动，这取决于参数是正数还是负数。这个成员函数也会返回一个新的对象。下面是一个示例：

```
std::valarray<int> d1 {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto d2 = d1.cshift(2);          // Result d2 contains: 3 4 5 6 7 8 9 1 2
auto d3 = d1.cshift(-3);         // Result d3 contains: 7 8 9 1 2 3 4 5 6
```

apply()函数是 valarray 的一个非常强大的成员函数，它可以将一个函数应用到每个元素上，并返回一个新的 valarray 对象为结果。valarray 类模板中定义了两个 apply()函数模板：

```
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
```

有三件事需要注意。第一，所有版本都是 const，所以函数不能修改原始元素。第二，参数是一个有特定形式的函数，这个函数以 T 类型或 T 的 const 引用为参数，并返回 T 类型的值；如果 apply()使用的参数不符合这些条件，将无法通过编译。第三，返回值是 valarray<T>类型，因此返回值总是一个和原序列有相同类型和大小的数组。

下面是一个使用成员函数 apply()的示例：

```
std::valarray<double> time {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  9.0}; // Seconds
auto distances = time.apply([](double t)
                 {
                  const static double g {32.0}; // Acceleration due to gravity ft/sec/sec
                  return 0.5*g*t*t;
                 }); // Result: 0 16 64 144 256 400 576 784 1024 1296
```

如果从高层建筑向下丢砖块，distances 对象可以计算出，过了相应的秒数后，砖块下落了多少距离；为了使最后结果有效，建筑的高度必须超过 1296 英尺。注意，不能使用从闭合区域捕获的值作为参数的 lambda 表达式，因为这和函数模板中参数的规格不匹配。例如，下面的代码就无法通过编译：

```
const double g {32.0};
auto distances = times.apply([g](double t) { return 0.5*g*t*t; }); // Won't compile!
```

在 lambda 表达式中以值捕获 g 会改变它的类型，以至于它不符合应用模板的规范。对于可以作为 apply()参数的 lambda 表达式，捕获语句必须为空。必须有一个和数组类型相同的参数，并返回一个这种类型的值。

valarray 头文件定义了许多来自于 cmath 头文件的函数的重载版本，因此它们能够应用到 valarray 对象的所有元素上。接受一个 valarray 对象为参数的函数有：

```
abs(), pow() , sqrt(), exp(), log(), log10()sin(), cos(), tan(), asin(),
acos(), atan(), atan2()sinh(), cosh(), tanh()
```

下面是一个将这一节的代码段组合到一起的示例，它提供了一次将 cmath 函数用到 valarray 对象上的机会：

```
// Ex10_02.cpp
// Dropping bricks safely from a tall building using valarray objects
#include <numeric>                  // For iota()
#include <iostream>                 // For standard streams
#include <iomanip>                  // For stream manipulators
#include <algorithm>                // For for_each()
#include <valarray>                 // For valarray
const static double g {32.0};       // Acceleration due to gravity ft/sec/sec

int main()
{
  double height {}; // Building height
std::cout << "Enter the approximate height of the building in feet: ";
std::cin >> height;
// Calculate brick flight time in seconds
double end_time {std::sqrt(2 * height / g)};
size_t max_time {1 + static_cast<size_t>(end_time + 0.5)};

std::valarray<double> times(max_time+1);          // Array to accommodate times
std::iota(std::begin(times), std::end(times), 0); // Initialize: 0 to max_time
*(std::end(times) - 1) = end_time;                // Set the last time value

// Calculate distances each second
auto distances = times.apply([](double t) { return 0.5*g*t*t; });

// Calculate speed each second
auto v_fps = sqrt(distances.apply([](double d) { return 2 * g*d;}));

// Lambda expression to output results
auto print = [](double v) { std::cout << std::setw(6) << static_cast<int>(std::
  round(v)); };

// Output the times - the last is a special case...
std::cout << "Time (seconds): ";
std::for_each(std::begin(times), std::end(times)-1, print);
std::cout << std::setw(6) << std::fixed << std::setprecision(2) << *(std::
  end(times)-1);
std::cout << "\nDistances(feet):";
```

```
    std::for_each(std::begin(distances), std::end(distances), print);

    std::cout << "\nVelocity(fps): ";
    std::for_each(std::begin(v_fps), std::end(v_fps), print);

    // Get velocities in mph and output them
    auto v_mph = v_fps.apply([](double v) { return v*60/88; });
    std::cout << "\nVelocity(mph): ";
    std::for_each(std::begin(v_mph), std::end(v_mph), print);
    std::cout << std::endl;
}
```

这样就能够确定从高层建筑丢下砖块时发生了什么。下面是一些从迪拜塔得到的示例输出，为了避免砖块撞到墙壁，假设是从一根足够长的杆子上扔下砖块的：

```
Enter the approximate height of the building in feet: 2722
Time (seconds):  0  1  2   3   4   5   6   7   8    9   10   11   12   13 13.04
Distances(feet): 0 16 64 144 256 400 576 784 1024 1296 1600 1936 2304 2704 2722
Velocity(fps):   0 32 64  96 128 160 192 224  256  288  320  352  384  416  417
Velocity(mph):   0 22 44  65  87 109 131 153  175  196  218  240  262  284  285
```

首先，如果我们真的这么做了，肯定会坐牢——甚至更糟。其实，计算时忽略了阻力，但这本书是关于 STL 的，不考虑物理因素。最后，可以通过花费的时间乘以加速度得到速度，但那样就不能将 sqrt()应用到 valarray 上了，不是吗？

所有的代码都很简单。常量 g 被定义在全局作用域内，因为这样方便在代码的不同位置使用，包括 lambda 表达式。times 数组中保存的时间是以秒为单位的，用 iota()算法从 0 开始填充元素。最后的时间值，对应于砖块撞到地面时，几乎可以肯定不是整数，所以保存的值是具体的。用 for_each()来产生输出，因为它比 copy()更能控制输出值和输出流迭代器。最后的时间值不可能是整数秒，因此它被当作输出中的特殊情况。lambda 表达式 print 是显式定义的，因此它能被重用，从而输出每组值。

为了得到或设置 valarray 中给定索引的元素，可以使用下标运算符[]，但下标运算符可以做的事不止于此，在本章的后面你就会看到。

10.3.2 一元运算符

这里有 4 个可以应用到 valarray 对象的一元运算符：+、-、~和!。效果是将运算符应用到数组的每个元素上，并返回一个新的 valarray 对象作为结果，不改变原对象。只能将它们应用到元素类型支持这些运算符的 valarray 对象上应用！运算符得到的新元素总为布尔类型，所以这个运算的结果是一个 valarray<bool>类型的对象。本章的后面会讨论它的使用场景。其他运算符生成的结果必须和这个运算的合法原始元素是相同的类型。例如，一元减法运算符只能反转有符号数的元素的符号，因此它无法用于无符号类型。下面的代码展示了!运算符的效果：

```
std::valarray<int> data {2, 0, -2, 4, -4};
auto result = !data;                    // result is of type valarray bool
```

```
std::copy(std::begin(result), std::end(result),
          std::ostream_iterator<bool>{std::cout <<std::boolalpha, " "});
std::cout << std::endl;                    // Output: false true false false false
```

当!被应用到 data 中的值时，值首先会被隐式转换为布尔，然后运算符会被应用到结果上。如果想用 copy()算法以布尔值的形式输出值，结果肯定是 true false true true true，这解释了为什么上述代码的输出是注释显示的那样。

~运算符是位的 NOT(位的取反)或 1 的补码。下面是一个示例：

```
std::valarray<int> data {2, 0, -2, 4, -4};   // 0x00000002 0 0xfffffffe
                                             // 0x00000004 0xfffffffc
auto result = ~data;
std::copy(std::begin(result), std::end(result), std::ostream_iterator<int>
  {std::cout, " "});
std::cout << std::endl;                      // Output: -3 -1 1 -5 3
```

为了生成结果中的元素，可以通过对原始整数值取反的方式来得到。例如，data 中的第二个元素的比特位全为 0，因此应用～产生的值的比特位全为 1，对应于十进制的-1。

+运算符对数值没有效果；-运算符会改变符号。例如：

```
std::valarray<int> data {2, 0, -2, 4, -4};
auto result = -data;
std::copy(std::begin(result), std::end(result), std::ostream_iterator<int>
  {std::cout, " "});
std::cout << std::endl;                      // Output: -2 0 2 -4 4
```

当然，也可以覆盖原始对象。为了使代码不那么杂乱，从现在起，假设为 std::valarray 使用的 using 指令生效，然后去掉代码中用于对 valarray 类型进行限定的 std 命名空间。

10.3.3　用于 valarray 对象的复合赋值运算符

所有的复合赋值运算符的左操作数都是 valarray 对象，右操作数是一个和所保存的元素同类型的值。在这种情况下，值会被合并到每个元素的值上，合并方式由运算符决定。右操作数也可以是和左操作数有相同元素个数和元素类型的 valarray 对象。在这种情况下，左操作数会因为合并了右操作数对应的元素而被修改。这一类的运算符有：

- 复合算术赋值运算符+=、-=、*=、/=、%=。例如：

```
valarray<int> v1 {1, 2, 3, 4};
valarray<int> v2 {3, 4, 3, 4};
v1+= 3;                              // v1 is: 4 5 6 7
v1 -= v2;                            // v1 is: 1 1 3 3
```

- 复合位操作赋值运算符&=、|=、^=。例如：

```
valarray<int> v1 {1, 2, 4, 8};
valarray<int> v2 {4, 8, 16, 32};
v1 |= 4;                            // v1 is: 5 6 4 12
```

```
v1 &= v2;                          // v1 is: 4 0 0 0
v1 ^= v2;                          // v1 is: 0 8 16 32
```

- 复合移位赋值运算符>>=、<<=。例如：

```
valarray<int> v1 {1, 2, 3, 4};
valarray<int> v2 {4, 8, 16, 32};
v2 <<= v1;                        // v2 is: 8 32 128 512
v2 >>= 2;                         // v1 is: 2 8 32 128
```

复合位操作和复合移位运算符一般用于整数类型。

10.3.4 valarray 对象的二元运算

可以将能够应用到基本类型值的任何二元运算符应用到 valarray 对象上，也能够合并两个 valarray 对象的相应元素，或者将 valarray 中的元素和一个同类型的值合并。在 valarray 头文件中定义了下面这些二元运算符的非成员操作符函数：

- 算术运算符+、-、*、/、%
- 位操作运算符&、|、^
- 位移运算符>>、<<
- 逻辑运算符&&、||

这些运算符有不同的版本，可以应用到一个 valarray<T>对象和一个 T 类型对象上、一个 T 类型对象和一个 valarray 对象上，或者应用到两个 valarray 对象上。两个 valarray 对象上的运算需要它们有相同个数的相同类型的元素。逻辑运算符会返回一个和 valarray 操作数有相同个数元素的 valarray<bool>对象。其他的运算符会返回一个和 valarray 操作数有相同类型且相同个数元素的 valarray 对象。

将 valarray 对象的内容输出到 cout 来说明发生了什么是很有用处的：

```
// perline is the number output per line, width is the field width
template<typename T>
void print(const std::valarray<T> values, size_t perline = 8, size_t width = 8)
{
  size_t n {};
  for(const auto& value : values)
  {
    std::cout << std::setw(width) << value << " ";
    if(++n % perline == 0) std::cout << std::endl;
  }
  if(n % perline != 0) std::cout << std::endl;
  std::cout << std::endl;
}
```

这个函数适用于包含任何 T 类型元素的 valarray 对象，只要 T 类型元素支持输出流的 operator<<()。

这里不会为所有的二元运算符都举一个示例——只做少量说明。下面是一个将二元算术运算符用于 valarray 对象的示例：

```
valarray<int> even {2, 4, 6, 8};
valarray<int> odd {3, 5, 7, 9};
auto r1 = even + 2;
print(r1, 4, 3);                    // r1 contains: 4 6 8 10
auto r2 = 2*r1 + odd;
print(r2, 4, 3);                    // r2 contains: 11 17 23 29
r1 += 2*odd - 4*(r2 - even);
print(r1, 4, 3);                    // r1 contains: -26 -36 -46 -56
```

最后一条语句使用复合赋值运算符的成员函数来加上右操作数(一个表达式)的结果。这说明可以像数值那样来合并包含 valarray 对象的运算，包括括号的使用。下面是一个使用位移运算的示例：

```
print(odd << 3, 4, 4); // Output is: 24 40 56 72
```

print()的第一个参数是将 odd 中的元素向左移动 3 个比特位之后得到的 valarray 对象。每行输出 4 个值，每个值的宽度是 4。

valarray 头文件中也定义了一些非成员函数，可以用来比较两个 valarray<T>对象，或者将 valarray<T>对象的每一个元素和 T 类型的值做比较。比较的结果保存在 valarray<bool>对象中，它和 valarray 包含相同个数的元素。支持的运算是==、!=、<、<=、>和>=。下面是一些使用这些运算符的示例：

```
valarray<int> even {2, 4, 6, 8};
valarray<int> odd {3, 5, 7, 9};
std::cout << std::boolalpha;
print(even + 1 == odd, 4, 6);       // Output is: true true true true
auto result = (odd < 5) && (even + 3 != odd);
print(result);                      // Output is: true false false false
```

倒数第二条语句用二元运算符&&来合并比较结果。结果说明，当 odd 元素小于 5 时，even 中相应的元素加上 3 后不等于 odd 中的元素；只有在比较 even 和 odd 的第一个元素时表达式才为 true，因为 odd 中只有第一个元素能使 odd<5，能使 even + 3 != odd 总为 true。

有一些定义了适用于 valarray 的元素子集的辅助类。主要的辅助类是 std::slice 和 std::gslice。为这些代码去掉 std 命名空间限定符。在更深入了解能用 valarray 做什么之前，让我们先探索如何将辅助类用于 valarray。

10.3.5　访问 valarray 对象中的元素

valarray 对象以序列的方式保存其中的元素。像之前说的那样，通过使用下标运算符来使用索引可以得到任何元素的引用，并能够获取或设置值。下面是一些示例：

```
std::valarray<int> data {1,2,3,4,5,6,7,8,9};
data[1] = data[2] + data[3];      // Data[1] is 7
data[3] *= 2;                     // Data[3] is 8
data[4] = ++data[5] - data[2];    // data[4] is 4, data[5] is 7
```

这就像访问一般数组的元素一样。但是，valarray 对象的下标运算符有更多用处。可以用有下标运算符的辅助类的实例来代替索引。这使我们能够指定和访问元素的子集。辅助类定义的元素选择机制使它们可以用于元素，只要这些元素所在的数组是二维或更多维数的数组。理解这是如何工作的很重要，因为这是 valarray 超越序列容器的主要优势。

还有很多细节，因此让我们先看看大致的流程。我们首先会探讨元素选择机制一般是如何工作的，然后如何从二维数组中选择特定的行或列。会解释如何将辅助类用于 valarray 对象以不同方式选择的元素子集上，以及如何表示子集。在说明生成子集的各种可能方式之后，会讨论能对它们做些什么。在这之后，会展示如何将这些技术运用到应用的场景中。

1. 创建切片

std::slice 类定义在 valarray 头文件中。由传给 valarray 对象的下标运算符的 slice 对象定义的 slice 就像索引。可以用 slice 对象作为 valarray 对象的下标来选择两个或更多个元素。被选择的元素不需要是数组中的连续元素。slice 选择的数组元素可以作为引用，因此可以访问或改变这些元素的值。

从根本上说，slice 对象为从 valarray 选择的元素封装了一系列索引。可以将 3 个 size_t 类型的值传给 slice 构造函数来定义 slice 对象：

- valarray 对象中的 start index(起始索引)指定了子集的第一个元素
- size 是子集中的元素个数
- stride(步进)是为从子集中得到下一个元素的 valarray 索引的增量

构造函数的参数采用描述的顺序，因此可以按如下方式定义 slice 对象：

```
slice my_slice {3, 4, 2}; // start index = 3, size = 4, stride = 2
```

这个对象指定了从索引 3 开始的 4 个元素，后续索引的增量为 2。也有构造函数，因此可以复制进行 slice 对象。默认的构造函数会将 start index、size 和 stride 全部设为 0，它的唯一目的是进行切片对象数组的创建。

可以调用 slice 对象的成员函数 start()来得到它的 start index。slice 对象也有成员函数 size()和 stride()，它们各自返回 size 和 stride。返回的 3 个值都是 size_t 类型。

通常，在使用 slice{start, size, stride}对象作为 valarray 对象的下标时，可以选择索引值位置的元素：

```
start, start + stride, start + 2*stride,... start +(size - 1)*stride
```

图 10-3 用一个值从 1 到 15 的 valarray 对象进行了展示。

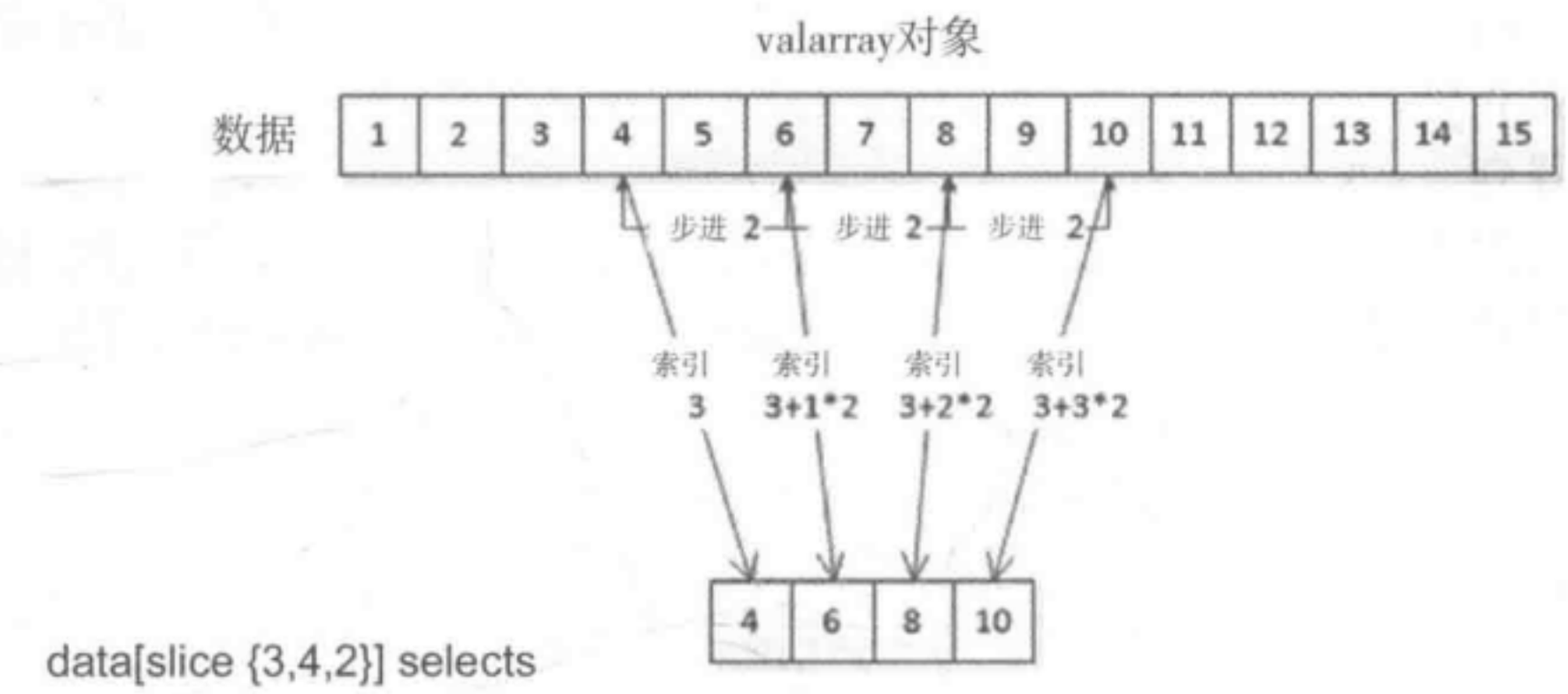


图 10-3　slice 对象选择的 valarray 中的元素子集

图 10-3 中以 slice 对象为 data 的下标运算符的参数会选择索引位置在 3、5、7、9 的元素，它们是数组中的第 4 个、第 6 个、第 8 个和第 10 个元素。slice 构造函数的第一个参数是第一个元素的索引，第二个参数是索引值的个数，第三个参数是索引值从一个到另一个的增量。用 slice 对象作为 valarray<T>对象的索引的结果是得到另一个对象——它还能是什么？它是一个 slice_array<T>类型的对象，封装了 slice 在 valarray<T>中选择的元素的引用。在解释更多关于如何使用分片之后，会回来说明能够用 slice_array 对象做些什么。

选择行

假设图 10-3 中 data 对象的值表示的是有 3 行 5 个元素的二维数组——按行的顺序，图 10-4 说明了如何用 slice 对象选择第二行。

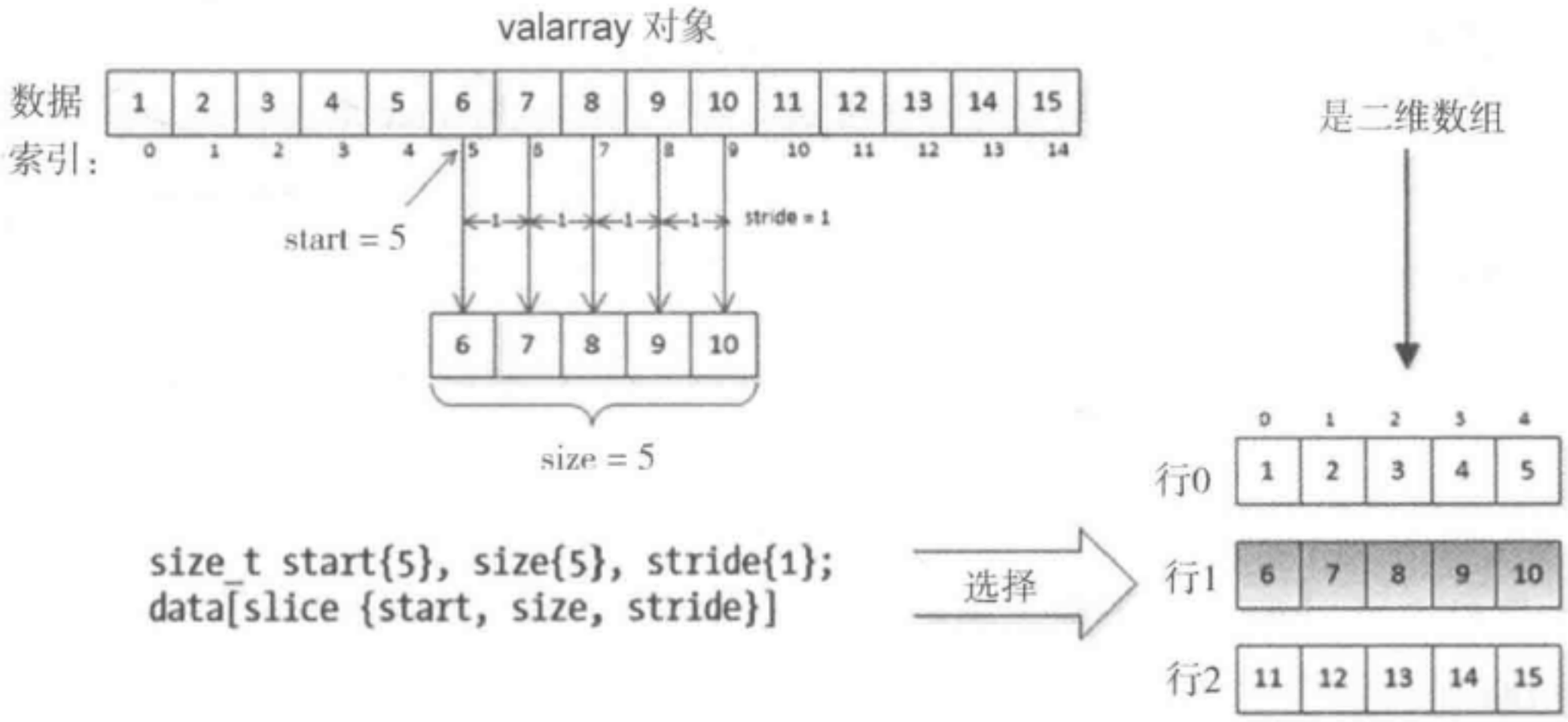


图 10-4　选择二维数组的单行

start index 是第二行的第一个元素的索引，值为 15。stride 是 1，每一行内的元素都是连续保存的。size 是 5，因为每行有 5 个元素。调用表示一行的 slice 对象的成员函数会返

回这一行的第一个元素，在我们使用多行时，这是很有用的。当然，a_slice 定义的 valarray 对象的第 n 个元素(索引从 0 开始)的索引是 a_slice.start()+n。

选择列

假设想从二维数组中选择一列。数组中列的元素不是连续的，这可能吗？“这是肯定的，斯坦利”，奥利会说。图 10-5 说明了如何定义一个 slice 对象来选择图 10-4 中同一个数组的第三列。

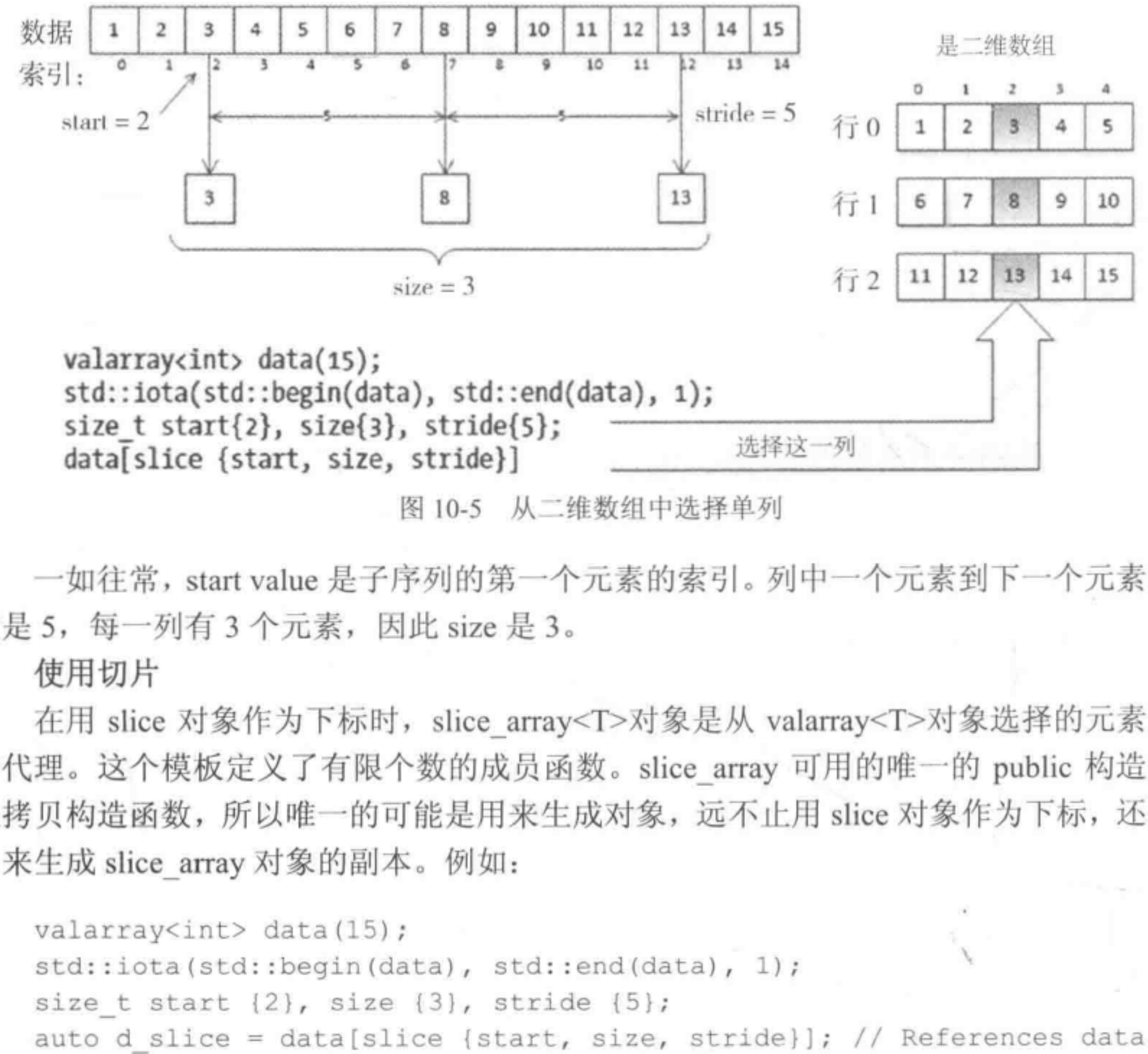


图 10-5　从二维数组中选择单列

一如往常，start value 是子序列的第一个元素的索引。列中一个元素到下一个元素的增量是 5，每一列有 3 个元素，因此 size 是 3。

使用切片

在用 slice 对象作为下标时，slice_array<T>对象是从 valarray<T>对象选择的元素子集的代理。这个模板定义了有限个数的成员函数。slice_array 可用的唯一的 public 构造函数是拷贝构造函数，所以唯一的可能是用来生成对象，远不止用 slice 对象作为下标，还可以用来生成 slice_array 对象的副本。例如：

```
valarray<int> data(15);
std::iota(std::begin(data), std::end(data), 1);
size_t start {2}, size {3}, stride {5};
auto d_slice = data[slice {start, size, stride}]; // References data[2],
                                                  // data[7], data[12]
slice_array<int> copy_slice {d_slice}; // Duplicate of d_slice
```

这里没有默认的构造函数，因此无法生成 slice_array 对象的数组。可以应用到 slice_array 对象的唯一操作是赋值和复合赋值。赋值运算符会将 slice_array 对象引用的全部元素设为给定值。也能够用它将被引用的元素设为另一个 valarray 对应元素的值，只要这个 valarray 与 slice_array 对象关联的 valarray 有相同的元素个数和类型。例如：

```
valarray<int> data {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
valarray<int> more {2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6};
```

```
data[slice{0, 5, 1}] = 99;    // Set elements in 1st row to 99
data[slice{10, 5, 1}] = more; // Set elements in last row to values from more
std::cout << "data:\n";
print(data, 5, 4);
```

可以看到，能够很容易用 data[slice{0, 5, 1}]这种表达式在赋值运算的左边生成 slice_array。这里调用了 slice_array 的成员函数 operator=()。右操作数可以是一个单值元素，或是一个包含相同类型元素的 valarray，或是另一个同类型的 slice_array。将单值元素赋给 slice_array 会将 valarray 中的元素设置为它所引用的值。当右操作数是 valarray 或 slice_array 时，必须保证它至少包含和左操作数一样多的元素；如果元素没有那么多，得到的结果将不是我们想要的。执行上面的代码，输出是：

```
data:
  99   99   99   99   99
   6    7    8    9   10
   2    2    3    3    3
```

可以看到 data 的第一行和第三行已经被改变了。

下面的任何一个复合赋值运算符(op=)都可以用于 slice_array 对象：

- 算术运算符+=、-=、*=、/-、%=
- 位操作运算符&=、|=、^=
- 移位运算符>>=、<<=

在每种情况下，左操作数都必须是一个 slice_array 对象，右操作数都必须是一个包含和 slice_array 同类型元素的 valarray 对象。op=运算符会将 op 运用到 slice_array 的每一个元素引用和作为右操作数的 valarray 的相应元素之间。注意不支持单值右操作数；总是需要用一个 valarray 对象作为右操作数，即使右边的所有对应元素的值都相同。

右操作数 valarray 一般会包含和左操作相同个数的元素，但不是绝对必要的。它包含的元素不能比另一个操作数少，但可以多。如果 slice_array 是有 n 个元素的左操作数，运算会使用右操作数的前 n 个元素。下面是一个用+=来修改 valarray 对象的分片的示例：

```
valarray<int> data {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
auto d_slice = data[slice {2, 3, 5}]; // References data[2], data[7], data[12]
d_slice += valarray<int>{10, 20, 30}; // Combines the slice with the new valarray
std::cout << "data:\n";
print(data, 5, 4);
```

对于 data 中被 d_slice 引用的元素，有更多的对象会被加到相应索引位置的元素值上。输出如下：

```
data:
   1    2   13    4    5
   6    7   28    9   10
  11   12   43   14   15
```

在这个运算之后，slice 所选的 data 数组中列元素的值是从 3+10、8+20 和 13+30 得到

的。在分片中和元素相乘也很简单：

```
valarray<int> factors {22, 17, 10};
data[slice{0, 3, 5}] *= factors; // Values of the 1st column: 22 102 110
```

slice 对象会选择 data 中的第一列，这一列的每一个元素都会和 factor 对象中的相应元素相乘。如果只是想用分片乘以一个给定的值，可以只生成一个适当的 valarray 对象：

```
valarray<int> data {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
slice row3 {10, 5, 1};
data[row3] *= valarray<int>(3, row3.size()); // Multiply 3rd row of data by 3
```

data 中的后 5 个元素会乘以 3。在最后一条语句中，通过调用 slice 对象的成员函数 size() 设置了右操作数 valarray 中的元素个数。这保证了元素个数和 data 中被选择的元素个数是相同的。

假设想将 data 中的一列元素加到另一列上，我们不能将一个 slice_array 加到另一个 slice_array 上，但仍然可以做我们想做的事。一种方式是使用一个接受 slice_array 为参数的 valarray 构造函数。通过这个构造函数，slice_array 对象引用的值会被用来初始化被生成的 valarray 对象中的元素。然后可以用这个对象作为 slice_array 的复合赋值中的右操作数。下面展示了如何将 data 中的第 5 列元素加到第 2 和第 4 列上：

```
valarray<int> data {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
valarray<int> col5 {data[slice{4, 3, 5}]}; // Same as 5th column in data
data[slice{1, 3, 5}] += col5;              // Add to 2nd column
data[slice{3, 3, 5}] += col5;              // Add to 4th column
print(data, 5, 4);
```

以 slice 对象作为索引的 data 被作为参数对象传给 valarray 的构造函数，从而可以生成一个 slice_array 对象，它可以用来生成对象 col5。valarray 构造函数并不是被显式定义的，因此它可以将 slice_array 类型隐式转换为 valarray 类型。可以按如下方式定义 col5 对象：

```
valarray<int> col5 = data[slice{4, 3, 5}]; // Convert slice_array to valarray
```

这里调用了 col5 对象的成员函数 operator=()，它需要一个 valarray 对象作为右操作数。编译器会为接受一个 slice_array 对象作为参数的 valarray 构造函数插入一个调用来将 slice_array 转换为 valarray。注意这和下面的语句并不相同：

```
auto col = data[slice{4, 3, 5}]; // col will be type slice_array
```

这里并没有发生转换。只是将 col 定义为以 slice 对象为索引从 data 中得到的 slice_array 对象。

调用 print()的上一段代码产生的输出为：

```
    1    7    3    9    5
    6   17    8   19   10
   11   27   13   29   15
```

当然，也能用一个简单的老式循环来做同样的事：

```
size_t row_len {5}, n_rows {3};          // Row length, number of rows
for(size_t i {}; i < n_rows*row_len; i += row_len)
{
  data[i+1] += data[i+4];                // Increment 2nd column
  data[i+3] += data[i+4];                // Increment 4th column
}
```

循环索引 i 选择 data 中的第一列元素的索引值为步进。在循环体中使用 i+n 形式的表达式作为 data 的下标来从第 n 列选择元素。让我们看一下 slice 对象在更真实应用中的使用。

2. 应用切片来解等式

我们可以写一个用 slice 对象和 valarray 对象解一组线性等式的程序。下面是一组普通的线性等式：

$$2x_1 - 2x_2 - 3x_3 + x_4 = 23$$

$$5x_1 + 3x_2 + x_3 + 2x_4 = 77$$

$$x_1 + x_2 - 2x_3 - x_4 = 14$$

$$3x_1 + 4x_2 + 5x_3 + 6x_4 = 23$$

4 个变量有 4 个线性等式，因此可以用它们找出满足这些等式的 x_1、x_2、x_3、x_4 的值，只要每个等式相对其他的等式是独立的。我们的程序会使用著名的高斯消元法来计算出 n 个线性等式中的 n 个未知变量，我们会用 valarray 对象来保存这些等式，进而实现它。valarray 对象会保存这些变量的系数和每个等式右边的值。例如，我们用下面这个 valarray 来保存上面的等式：

```
valarray<double> equations {2, -2, -3,  1, 23,
                            5,  3,  1,  2, 77,
                            1,  1, -2, -1, 14,
                            3,  4,  5,  6, 23 };
```

注意，equation 对象中的数据是一个有 4 行 5 列的二维数组。通常，n 个变量、n 个等式会用 n 行、n+1 列来表示。在写任何代码之前，需要先理解方法。

高斯消元法

高斯消元法包含两个基本的步骤。第一步是将原始的等式组转换为能够确定变量的值的不同形式，第二步是确定变量的值。图 10-6 说明了这个过程。

图 10-6 显示了 4 个方程、4 个未知数一般的表示，a 是系数，x 是变量，c 是等式右边的值。第一步是将等式的左边变换成右边的形式，这称作行阶梯形。图 10-6 描述了第二步，也就是从行阶梯形矩阵得到所有变量的值。这个过程称作回代。那么我们如何将这组等式转换为左边的行阶梯形矩阵？

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = c_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = c_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = c_3$$
$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = c_4$$

转换为

$$b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 = d_1$$
$$b_{22}x_2 + b_{23}x_3 + b_{24}x_4 = d_2$$
$$b_{33}x_3 + b_{34}x_4 = d_3$$
$$b_{44}x_4 = d_4$$

a_{ij}是等式i中变量x_j的系数。
c_i是等式i中右边的值。

从最后一个等式可以得出 $x_4=d_4/b_{44}$
知道x_4，可以确定x_3，$x_3=(d_3-b_{34}\times x_4)/b_{33}$
通过这种方式，继续求出另外两个变量的值

图 10-6　高斯消元法的作用

消元过程

笔者确定，你知道可以在等式的两边加上或减去相同的值，这样得到的等式仍然是有效的。这意味着可以加上或减去另一个等式的倍数，并且仍然可以得到有效的等式。图 10-7 展示了如何应用此思想来将 4 个线性等式表示为行阶梯形矩阵。

图 10-7　将线性等式变换为行阶梯形矩阵

图 10-7 显示了如何通过三个步骤将矩阵中连续行的元素设为 0。第一步是从第一行后的行开始反复减去专门选择的第一行的倍数。这个过程是反复的——减去第二行和第三行的倍数，这样就可以生成一个行阶梯形矩阵。从第一行开始做这些比较方便，并且依次进行下来，但也可以以任何顺序消除元素。

矩阵中被选择用来消除其他行中相应元素的元素被称作 pivot。每一个从一行中减去另一行的倍数的操作都会改变相应元素的系数，使 pivot 为 0，因此可以消除它。当然，这个操作也会改变其他系数的值，图 10-7 中字母的变化反映了这些。只要等式可解，消除过程

就能进行。如果任何等式都能通过其他的一个或多个等式的组合得到，这种情况就不适用。

找到最佳的 pivot

当然，一些系数可能是 0，因此不能任意使用消除过程。如果 a_{11} 是 0，继续遵循第二行减第一行的模式，就肯定会出现灾难性的后果。我们需要确定 pivot 元素不为 0。如果它的绝对值是列中最大的，就是数值有利的。矩阵的行所表示的等式顺序是任意的，因此行的顺序可以改变任意次，不会影响我们得到系数的解。所以，如果给定的 pivot 不是最大值，可以通过将当前行和有最大绝对值的行交换来将它换一个最大值。图 10-8 说明了这个过程。

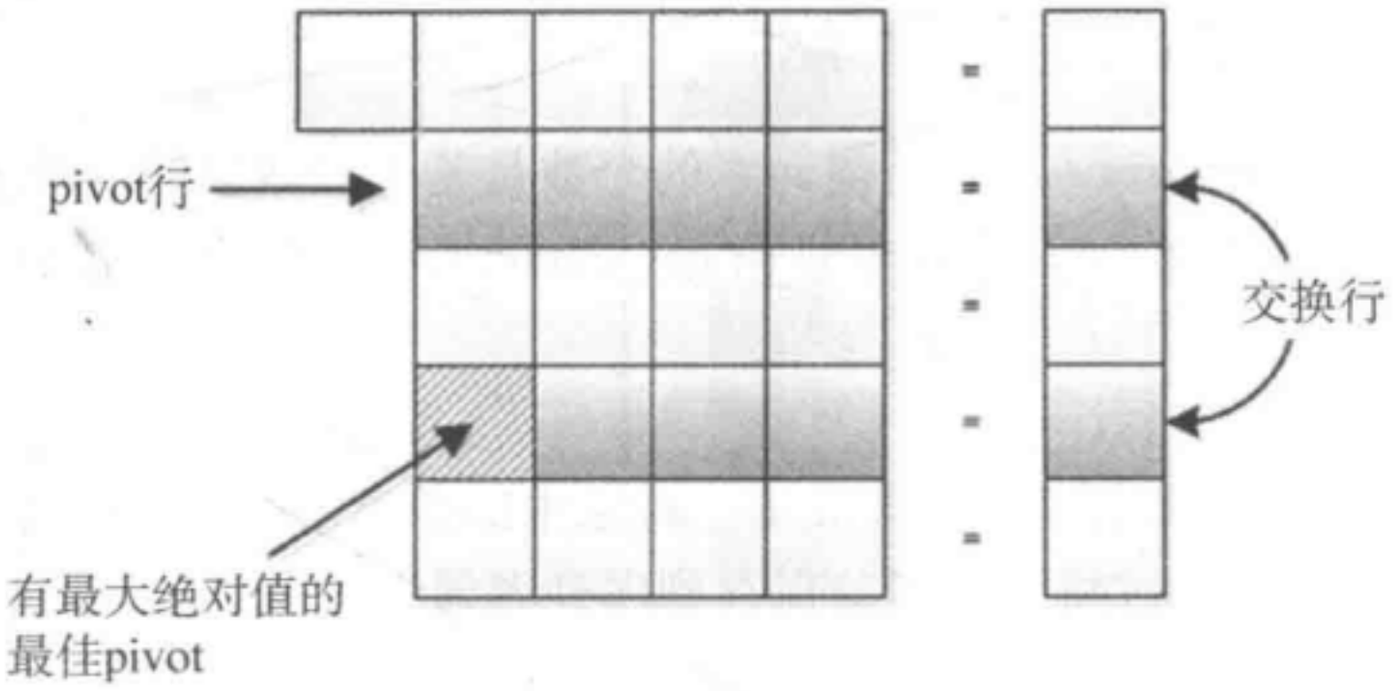


图 10-8　选择最佳 pivot

图 10-8 说明的是 5 个等式在第一行已经被完全消除后的情况。pivot 的最佳值在倒数第二行，因此下一步的消除开始之前，这一行和第二行交换。在有很多变量时，交换一行的所有元素可能会花费很长时间，所以最好避免这种操作。通过 slice 对象来确定行，可以不需要移动 valarray 的任何元素就能交换行，valarray 包含的是等式的矩阵。

我们已经足够了解高斯消元法是如何工作的，可以开始写代码了。我们会将等式中的所有数据保存到一个 valarray<double>对象中。程序中还有几个函数，把除了 main()之外的其他所有函数都放到一个单独的源文件 gaussian.cpp 中。下面从一个从标准输入流读取等式的函数开始。

得到输入数据

n 个变量的每一个等式都是下面这种形式：

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$$

总是会有 n 个等式和 n 个系数 a_i，每一个等式和右边的值都会被作为连续元素保存到 valarray 对象中。因此输入函数必须读 n*(n+1)个值，并将它们保存到 valarray 中。下面是做这些事情的代码：

```
// Read the data for n equations in n unknowns
valarray<double> get_data(size_t n)
{
  valarray<double> equations(n*(n + 1));     // n rows of n+1 elements
  std::cout << "Enter " << n + 1
            << " values for each of "<< n << " equations.\n"
```

```
                << "(i.e. including coefficients that are zero and the rhs):\n";
  for(auto& coeff: equations) std::cin >> coeff;
  return equations;
}
```

这个函数需要得到等式的个数，这和变量的个数相同。为了能够作为参数使用，调用程序，也就是 main()，必须提供这个参数。下面的代码会做这些事情：

```
size_t n_rows {};
std::cout << "Enter the number of variables: ";
std::cin >> n_rows;
auto equations = get_data(n_rows);
```

get_data()中生成的 valarray 对象所需元素的个数是基于参数值的，基于范围的 for 循环会为每个元素从 cin 读取值。这个对象是由 get_data()局部生成的，返回时会被移到调用位置。

将行作为 slice 对象

当选择 pivot 时，我们想避免移动 equations 对象中的数据。我们可以通过生成一个 slice 对象来定义每一行，并且通过将这些对象保存到序列容器中来做到。在 main()中可以按如下方式定义 slice 对象：

```
std::vector<slice> row_slices;          // Objects define rows in sequence
std::generate_n(std::back_inserter(row_slices), n_rows,
                                                [n_rows]()
{static size_t index {};
 return slice {(n_rows+1)*index++,
   n_rows+1, 1};
});
```

generate_n()算法在 row_slices 容器中保存了 n_rows 个 slice 对象，这些对象是通过 lambda 表达式生成的。lambda 表达式以值的方式捕获 n_rows。这些 slice 对象的唯一不同之处在于它们的起始索引值，索引值从 0 开始，以 n_rows+1 为步进，n_rows+1 是行的长度。每个 slice 的表示都是索引值为 n_rows+1、步进为 1。为了交换两行，我们只需要交换 row_slices 容器中这些行的 slice 对象。valarray 中的元素可以保持不变。

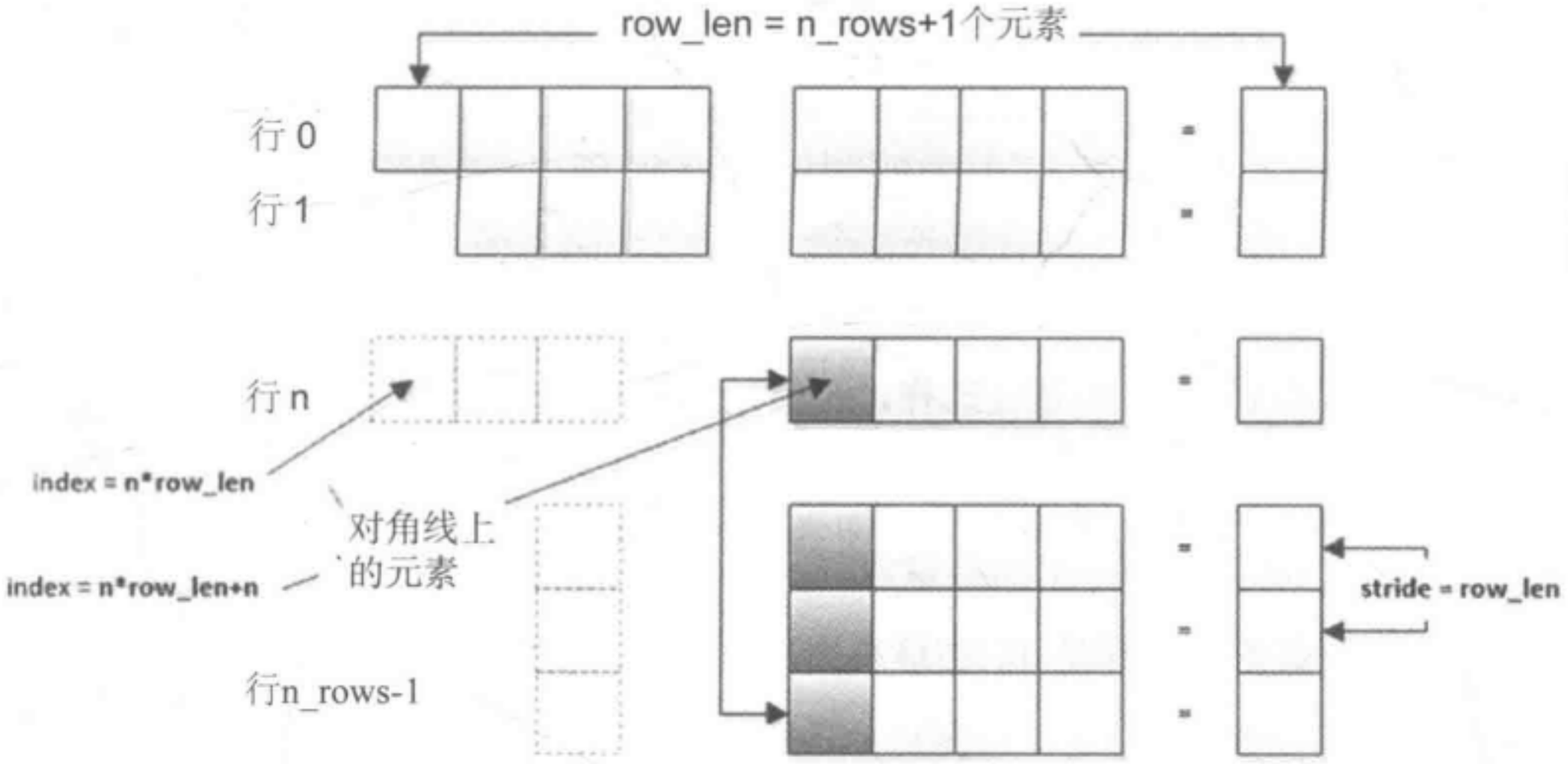
我们能够在容器中保存 slice 对象的指针，但因为 slice 对象是非常小的，在笔者系统上只有 12 字节，似乎并不值得这么麻烦。为了解出这些等式，我们只需要访问包含了它们的数据的 valarray 对象以及在矩阵中定义了行的 row_slices 容器。row_slices 对象的大小是行数，所以当我们访问 row_slices 容器时，我们需要知道行数和长度。

找到最佳 pivot

可以回到图 10-7 来看看如何通过减去后面的行中每一行的倍数来生成阶梯形等式，每一步会消除对角左边的一列变量，所以最后的结果是最后一行只有一个变量。先前的每步消除，都需要从消除的这行中找到最佳 pivot，并且驱动消除过程的整个循环会从第一行遍历到倒数第二行。查找 pivot 总是会涉及在列中搜索元素，搜索从等式矩阵的对角线开始，一直到最后一行。通过用两个可以计算的索引值来访问等式中的元素，可以确定和找到列

中最大的元素。下面用切片对象来做实践。

我们需要能够定义一个 slice 对象，它会选择从对角线到任意一行的列元素。图 10-9 显示了这是如何确定的。



- 索引值会被应用到valarray包含的元素上
- valarray包含了n_rows*row_len个元素
- 选择的阴影列的slice对象是：

slice{n*row_len+n,n_rows –n ,row_len}

图 10-9　确定要寻找 pivot 的列的 slice 对象

图 10-9 表示等式的 n_rows 行中的每行有 n_rows+1 个元素。第一行是行 0。任何元素到它下面的元素的距离总为行的长度。第 n 行中的第一个元素的索引是行长度的 n 倍。第 n 行中对角线元素的索引是第 n 行的第一个元素的索引加 n。

下面是一个为任意行设置最佳 pivot 的函数的代码：

```
// Selects the best pivot in row n (rows indexed from 0)
void set_pivot(const valarray<double>& equations, std::vector<slice>& row_
  slices, size_t n)
{
  size_t n_rows {row_slices.size()}; // Number of rows
  size_t row_len {n_rows + 1}; // Row length = number of columns

// Create an object containing the elements in column n, starting row n
valarray<double> column {equations[slice {n*row_len + n, n_rows - n, row_ len}]};
column = std::abs(column); // Absolute values

size_t max_index {}; // Index to best pivot in column
for(size_t i {1}; i < column.size(); ++i) // Find index for max value
{
  if(column[max_index] < column[i]) max_index = i;
```

```
    }
    if(max_index > 0)
    { // Pivot is not the 1st in column - so swap rows to make it so
      std::swap(row_slices[n], row_slices[n + max_index]);
    }
    else if(!column[0]) // Check for zero pivot
    { // When pivot is 0, matrix is singular
      std::cerr << "No solution. Ending program." << std::endl;
      std::exit(1);
    }
```

这会在第 n 行找到 pivot 的最佳选择，也就是第 n 列。这个过程已在图 10-9 中做了说明。Column 对象包含的是你感兴趣的列中的值，第一个元素会在第 n 行。最佳的 pivot 假设初始时在列的第一个元素，在第 n 行。如果 pivot 是在下面的行 n 中发现的，pivot 不能是 0，因为根据定义它大于行 n 中的元素。如果发现新的 pivot，可能在行 n 中的 pivot 为 0。这意味着这一列的其他元素都是 0，在这种情况下，等式无解。

形成行阶梯形式

reduce_matrix()函数会通过使用 set_pivot()函数减少列中最佳 pivot 之前的元素，来将等式中的值矩阵转换为行阶梯形式：

```
// Reduce the equations matrix to row echelon form
void reduce_matrix(valarray<double>& equations, std::vector<slice>& row_slices)
{
  size_t n_rows {row_slices.size()}; // Number of rows
  size_t row_len {n_rows + 1}; // Row length
  for(size_t row {}; row < n_rows - 1; ++row) // From 1st row to second-to-last
  {
    set_pivot(equations, row_slices, row); // Find best pivot

    // Create an object containing element values for pivot row
    valarray<double> pivot_row {equations[row_slices[row]]};
    auto pivot = pivot_row[row]; // The pivot element .
    pivot_row /= pivot; // Divide pivot row by pivot

    // For each of the rows following the current row,
    // subtract the pivot row multiplied by the row element in the pivot column
    for(size_t next_row {row + 1}; next_row < n_rows; ++next_row)
    {
      equations[row_slices[next_row]] -=
                equations[row_slices[next_row].start()+ row] * pivot_row;
    }
  }
}
```

这个函数会从等式的第一行一直遍历到倒数第二行。对于每一行，都调用 set_pivot()来得到最佳 pivot。生成的 valarray 对象会包含从当前行——pivot 行得到的元素的副本。

valarray 对象的成员函数 operator/=()会用每个元素作为左操作数除以右操作数的值，并且这个函数会被应用到 pivot_row 上，这时以 pivot 元素作为右操作数，从而使 pivot 的系数为 1。对于后面的每一行，pivot 行会乘以 pivot 列中元素的值，并且由此产生的 valarray 对象会从这行中去除。这会将 pivot 列中元素的值设为 0。

回代

在等式矩阵的行阶梯形式下，我们可以通过回代(back substitution)得到变量的值。矩阵最后一行定义的等式的所有系数除了最后一个之外都为 0。因此，最后一个变量的值就是右边的值除以系数的值。如果最后一行除以的系数是 1，变量的值就是这行中最后一个元素的值。然后可以将这个值代入前面的每一行，并依次在每行中减去它。这就在所有的行中消除了最后的变量，现在倒数第二个变量只有一个非零系数。我们重复这个过程，这看起来像一个循环。图 10-10 显示了 4 个等式的这个过程。

这个过程的结果是除了对角线之外所有的系数都是 0，对角线上的系数是 1。所以最后一列的值表示的就是等式的解。下面是一个实现如图 10-10 所示过程的函数的实现代码：

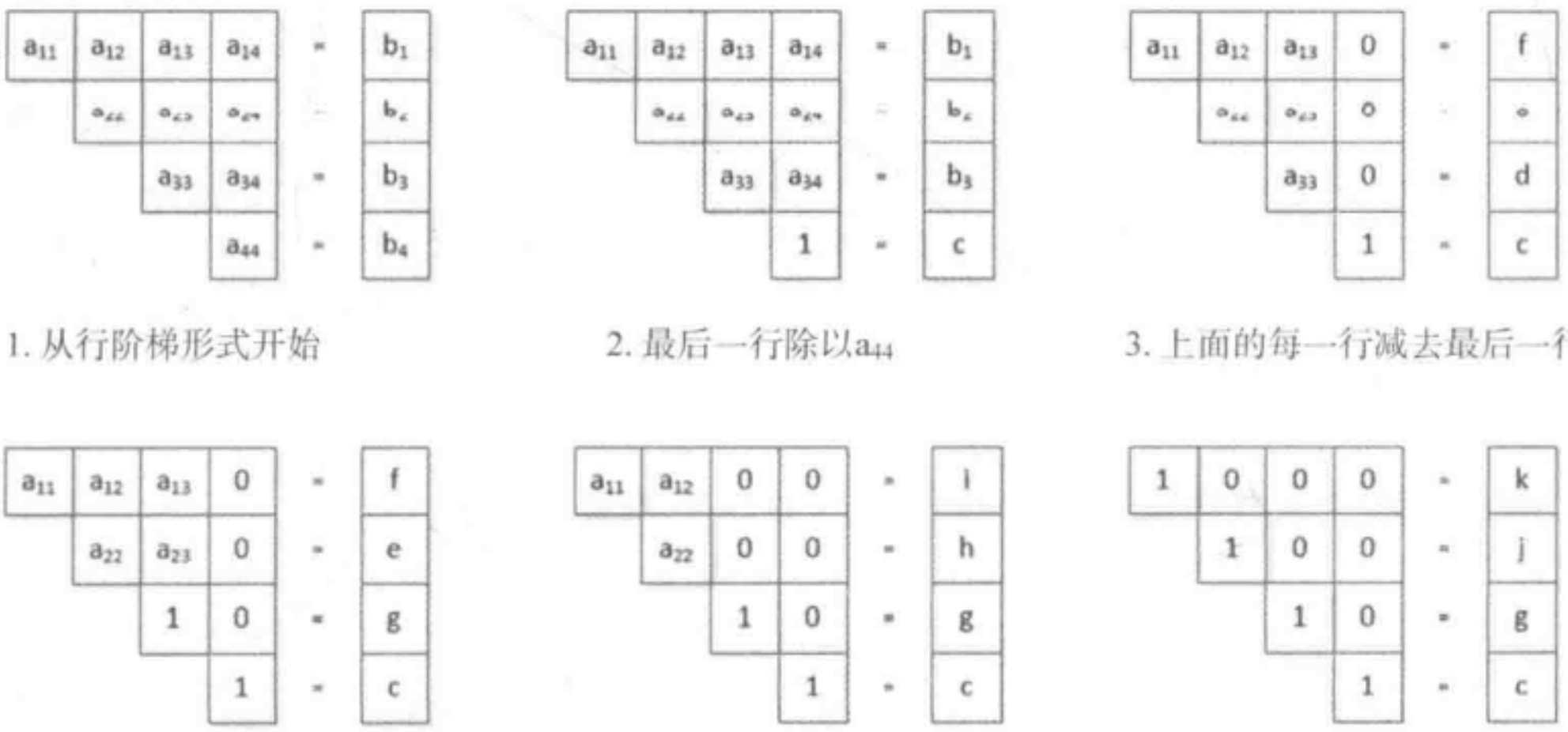


图 10-10　回代

```
// Perform back substitution and return the solution
valarray<double> back_substitution(valarray<double>& equations,
                                   const std::vector<slice>&row_slices)
{
  size_t n_rows{row_slices.size()};
  size_t row_len {n_rows + 1};

  // Divide the last row by the second to last element
  // Multiply the last row by the second to last element in each row and subtract
  // it from the row.
  // Repeat for all the other rows
  valarray<double> results(n_rows); // Stores the solution
```

```
    for(int row {static_cast<int>(n_rows - 1)}; row >= 0; --row)
    {
      equations[row_slices[row]] /=
           valarray<double>(equations[row_slices[row].start() + row], row_len);
      valarray<double> last_row {equations[row_slices[row]]};
      results[row] = last_row[n_rows]; // Store value for x[row]
      for(int i {}; i < row; ++i)
      {
        equations[row_slices[i]] -= equations[row_slices[i].start() + row] *
            last_row;
      }
    }
    return results;
  }
```

最重要的是，要记住是 row_slices 定义了序列的等式。查找最佳 pivot 的过程可以确定总是会改变等式的顺序，这是通过交换 row_slices 中的元素做到的，而不是通过移动 equations 数组中的元素。回代过程因此不得不使用由 row_slice 确定的行的顺序，而不是它们出现在等式中的行的顺序。因为这个，需要定义 results 对象来保存这些等式的解的值。外循环以相反的顺序遍历行。在外循环的每次遍历中，当前的行都会除以对角线上的系数。然后会生成一个包含当前行的副本的 valarray 对象 last_row。然后内循环会从前面的每个行中减去 last_row 的倍数，其中，倍数是行中对角线元素的值。当外循环结束时，results 对象中包含的解会被返回。

完整的程序

这个程序由两个文件组成。gaussian.cpp 文件中的内容如下：

```
// Gaussian.cpp
// Functions to implement Gaussian elimination
#include <valarray>                    // For valarray, slice, abs()
#include <vector>                      // For vector container
#include <iterator>                    // For ostream iterator
#include <algorithm>                   // For copy_n()
#include <utility>                     // For swap()
#include <iostream>                    // For standard streams
#include <iomanip>                     // For stream manipulators
using std::valarray;
using std::slice;

// Definition for get_data() ...

// Definition for set_pivot() ...

// Definition for reduce_matrix() ...

// Definition for back_substitution() ...
```

main()程序读取数据并以正确顺序调用 gaussian.cpp 中的函数，然后输出结果。Ex10_03.cpp 的内容如下：

```
// Ex10_03.cpp
// Using the Gaussian Elimination method to solve a set of linear equations
#include <valarray> // For valarray, slice, abs()
#include <vector> // For vector container
#include <iterator> // For ostream iterator
#include <algorithm> // For generate_n()
#include <utility> // For swap()
#include <iostream> // For standard streams
#include <iomanip> // For stream manipulators
#include <string> // For string type
using std::string;
using std::valarray;
using std::slice;
// Function prototypes
valarray<double> get_data(size_t n);
void reduce_matrix(valarray<double>& equations, std::vector<slice>&
row_slices);
valarray<double> back_substitution(valarray<double>& equations,
const std::vector<slice>& row_slices);
int main()
{
size_t n_rows {};
std::cout << "Enter the number of variables: ";
std::cin >> n_rows;
auto equations = get_data(n_rows);
// Generate slice objects for rows in row order
std::vector<slice> row_slices; // Objects define rows in sequence
size_t row_len {n_rows + 1};
std::generate_n(std::back_inserter(row_slices), n_rows,
                              [row_len]()
                              { static size_t index {};
                                return slice {row_len*index++, row_len, 1};
                              });
reduce_matrix(equations, row_slices);    // Reduce to row echelon form
auto solution = back_substitution(equations, row_slices);
// Output the solution
size_t count {}, perline {8};
std::cout << "\nSolution:\n";
string x{"x"};
for(const auto& v : solution)
{
  std::cout << std::setw(3) << x + std::to_string(count+1) << " = "
            << std::fixed << std::setprecision(2) << std::setw(10)
            << v;
  if(++count % perline) std::cout << '\n';
```

```
    }
    std::cout << std::endl;
  }
```

main()中的第一个操作是读取输入的问题的变量个数。然后，通过调用 get_data()读出 equations 的数据，得到的 valarray 对象会被移到 equations 中。生成一个选择 equations 中连续行的 slice 对象的 vector。起始索引从 0 开始，以行的长度递增。所有 slice 对象的大小和步进分别为行的长度和 1。在调用 reduce_matrix()之后，back_substitution()会返回解，除非没有解。在最后一个循环中会输出解的值。可以用基于范围的 for 循环来遍历 valarray 对象中的元素，因为迭代器可用。

下面是在解 6 个等式时的示例输出：

```
Enter the number of variables: 6
Enter 7 values for each of 6 equations.
(i.e. including coefficients that are zero and the rhs):
 1  1   1   1  1  1    8
 2  3  -5  -1  1  1  -18
-1  5   2   7  2  3   40
 3  1  10   2  1 11  -15
 3 17   5   1  3  2   41
 5  7   3  -4  2 -1    9

Solution:
x1 = -2.00
x2 = 1.00
x3 = 3.00
x4 = 4.00
x5 = 7.00
x6 = -5.00
```

通常可以通过在 reduce_matrix()的适当位置添加输出来跟踪矩阵的缩小进度。可以用一种类似的方式来跟踪回代机制。这可以让我们在实际中更好地了解高斯消元法。可以使用你在本章前面看到的 print()函数模板来做这件事。使用 slice 对象是很简单的事情，是时候来介绍一点更有挑战的内容了。

10.3.6 多个切片

valarray 头文件中定义了 gslice 类，这是切片思想的泛化。gslice 对象从起始索引生成索引值，就像一个切片，但它能生成两个或两个以上的切片，具体实现方式有一点复杂。通常，gslice 会假定从表示多维数组的 valarray 对象中选择的元素会被用来作为元素的线性序列。gslice 由 3 个参数值定义。第一个构造函数参数是起始索引，它是指定第一个分片的第一个元素的 size_t 类型的值，这和 slice 一样。第二个参数是一个 valarray<size_t>对象，它的每个元素都指定了 size。对于第二个参数指定的每一个 size，都有一个相应的步进；

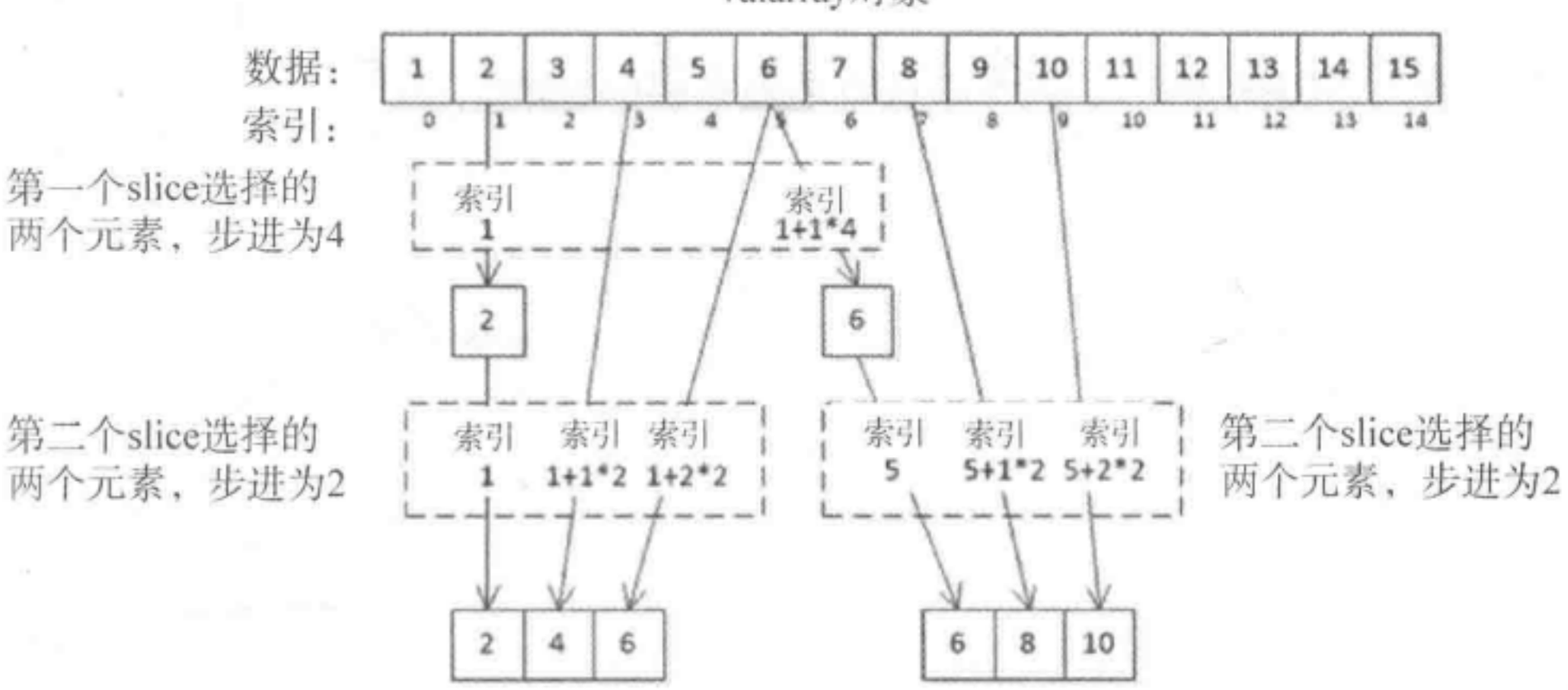
这些步进由第三个参数定义，它是一个和第二个参数有相同元素个数的 vlarray<size_t>对象。第二个参数中，每个 size 的步进都在第三个参数中有相应的元素。

当然，gslice 表示的每个 slice 都有一个起始索引、大小和步进。第一个 slice 的起始索引是 gslice 构造函数的第一个参数。第一个 slice 的 size 是 valarray(由 size 组成)的第一个元素，步进是另一个的 valarray(由 stride 组成)的第一个元素。这应该可以很容易发现，但现在它变得有些复杂。

第一个 slice 生成的索引值是第二个 slice 的起始索引。也就是说，第二个 slice 从来自于第一个 slice 的每个索引中定义了一套索引值。整个过程以此类推。

第一个 slice 之后的每一个 slice 的索引都是由前一个 slice 生成的，它们会产生一组索引值。例如，如果 glice 的第一个 slice 的大小是 3，它定义了 3 个索引值；如果第二个 slice 的大小是 2，它生成了 2 个索引值。第二个 slice 的大小和步进会被使用 3 次——每一次都会用第一个 slice 的索引值作为起始索引。因此可以从这两个 slice 中得到 6 个索引值，它会从 valarray 中选择 6 个元素。

用 gslice 作为 valarray 的下标的结果是会包含一个给定元素的多个引用。当 gslice 的最后一个 slice 所生成的索引序列重叠时，会出现这种情况。图 10-11 展示了一个 gslice 从 valarray 中选择元素的简单示例。




```
valarray<size_t> sizes   {2, 3};
valarray<size_t> strides {4, 2};

data[gslice {1,sizes,strides}]
```

选择2、4、6、8、10

gslice指定
- 起始索引为1
- 从索引1开始的步进为4、大小为2的第一个slice
- 从第一个slice的每一个索引得到的第二个slice，它的大小为3、步进为2。

图 10-11　gslice 对象如何从 valarray 中选择元素

图 10-11 中的 gslice 定义了两个 slice。图 10-11 显示了第一个 slice 如何生成应用于第二个 slice 的起始索引的索引值，第二个索引的大小为 3。因为最后两个索引序列是重叠的，

结果中索引 5 处的元素 6 是重复的。通常，gslice 对象选择的元素的个数是由 valarray 对象的大小值定义的，它是构造函数的第二个参数——在这个示例中是 2×3。

和 slice 对象一样，用 gslice 去索引 valarray<T>会得到一个封装了 valarray 中元素的引用对象，但这个对象是不同的类型——是 gslice_array<T>类型。后面会介绍如何使用它。首先将介绍 gslice 对象的一些作用。

10.3.7 选择多行或多列

可以通过用 gslice 对象作为下标运算符的参数来从 valarray 对象选择多行或多列，被选择的行或列必须是被均匀隔开的，这意味着连续行或列的第一个元素之间的增量是相同的。选择两行或两行以上相对来说要简单一些。gslice 的起始索引是被选择的第一行的第一个元素的索引值。它的第一个大小是行数，相应的步进是行之间的增量，是行的长度的倍数。第二个大小和步进的值会选择每行的元素，所以第二个大小是行的长度，第二个步进是 1。

假设按如下方式用 sizes 和 strides 定义 valarray 对象：

```
valarray<size_t> sizes {2, 5};      // Two size values
valarray<size_t> strides {10, 1};   // Stride values corresponding to the
                                       size values
```

选择图 10-11 中数组的第一行和第三行的表达式是：

```
data[gslice{0, sizes, strides}]
```

这两行分别从索引值 0 和 10 开始；这些索引值是从作为 gslice 的构造函数的第一个参数的起始索引 0 定义的第一个 slice 得到的，并且第一个 slice 的大小及对应的步进值在作为 gslice 构造函数的第二和第三个参数的 valarray 对象中。每行有 5 个连续的元素，第二个 slice 的所选行的大小是 5，对应的步进值是 1。注意，不必显式地定义 sizes 和 strides。

可以将选择这两行的表达式写为：

```
data[gslice{0, {2, 5}, {10, 1}}]
```

现在，让我们思考一下更困难的任务——选择两列或更多列。作为一个示例，让我们看看如何从图 10-11 所示的数组中选择第一列、第三列和最后一列。图 10-12 用在元素的二维表示中选择的置灰列来加以说明。

第一个大小和步进确定了每列被选择的第一个元素的索引值。第二个大小和步进选择的是每列的元素；一列中元素之间的增量是行的长度。因为第一个步进是固定的，所以只能用这种方式选择有相同间隔的两列或更多列；例如不能选择第 1，第 2，第 5 列。

定义了三列或更多列的 gslice 对象，能够以同样的方式应用到三维或更高维的数组中，但它变得更复杂了。需要注意 gslice 不会生成单独的索引值；如果生成了，结果是未定义的，而且效果肯定不好。大多时候，slice 和 gslice 对象会被应用到一维或二维数组上，所以重点会放在它们身上。

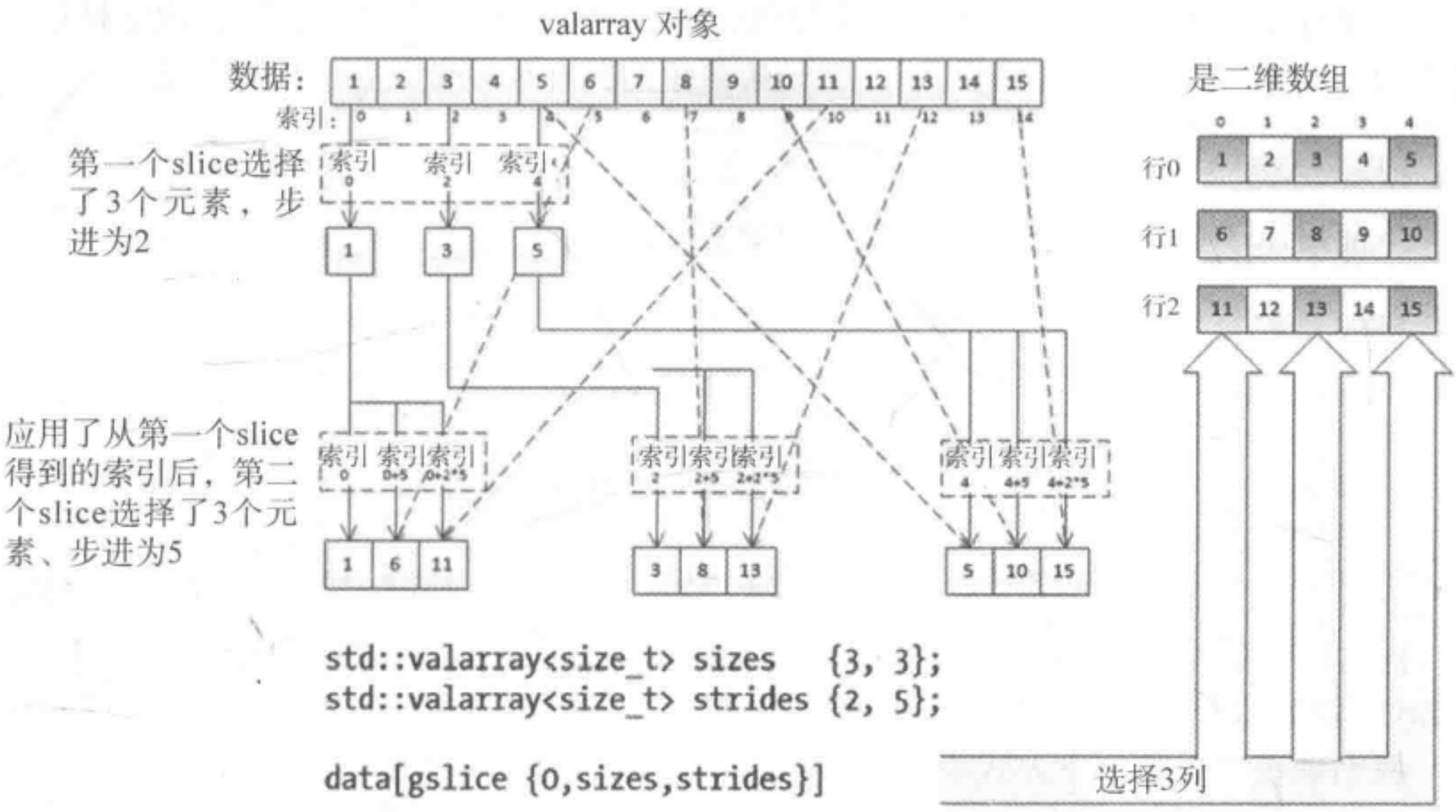


图 10-12　从二维数组中选择多列

10.3.8　使用 gslice 对象

当用 gslice 对象索引 valarray<T>对象时，得到的 gslice_array<T>对象和 slice_array 有很多相同之处。它和 slice_array 有相同作用域的成员函数，因此同一作用域的运算符也可以被应用到它上面。有赋值运算符和相同作用域的 op=运算符，也有接受 gslice_array 对象作为参数的 valarray 构造函数，它可以用来将 gslice_array<T>类型隐式转换为 valarray<T>。

让我们思考一下我们能用 gslice 做什么。假设我们定义了下面的 valarray 对象：

```
valarray<int> data {2,   4,   6,  8,       // 4 x 4 matrix
                    10, 12, 14, 16,
                    18, 20, 22, 24,
                    26, 28, 30, 32};
```

这里有 4 行，每行有 4 个元素。我们可以用前面看到的 print()函数模板来输出第二行和第三行：

```
valarray<size_t> r23_sizes {2,4};     // 2 and 4 elements
valarray<size_t> r23_strides {4,1};   // strides: 4 between rows, 1 between
                                      // elements in a row
gslice row23 {4, r23_sizes, r23_strides}; // Selects 2nd + 3rd rows - 2 rows of 4

print(valarray<int>(data[row23]),4); // Outputs 10 12 14 16/18 20 22 24
```

row23 对象定义的行索引序列从 4 到 7、从 8 到 11、步进为 1，这里会选择 data 的中间两行。当然，可以用一条语句来输出这两行：

```
print(valarray<int>(data[gslice{4, valarray<size_t>{2,4}, valarray<size_t>
  {4,1}}]), 4);
```

在执行这条语句之后，gslice 对象与包含大小和步进的对象会被舍弃。像这样很难看到从 data 中选择了什么，但它可以。另一条更短的同样可以做这件事的语句是：

```
print(valarray<int>(data[gslice{ 4, {2,4}, {4,1} }]), 4);
```

下面展示了如何输出 data 中的第二和第三列：

```
std::valarray<size_t> sizes2 {2,4};     // 2 and 4 elements
// strides: 1 between columns, 4 between elements in a column
std::valarray<size_t> strides2 {1,4};
gslice col23 {1, sizes2, strides2};     // Selects 2nd and 3rd columns - 2
  columns of 4
print(valarray<int>(data[col23]), 4);  // Outputs 4 12 20 28/6 14 22 30
```

gslice 的起始索引是第二个元素，它也是第二列的第一个元素。现在应该可以清楚这是如何指定 data 中的两列的。

我们现在可以按如下方式添加第二和第三行的值到第二和第三列了：

```
data[col23] += data[row23];
print(data, 4);
```

执行这些语句之后会产生下面的输出：

```
2  14  24  8
10 24 34 16
18 34 44 24
26 44 54 32
```

如果用这些和用来初始化 data 的原始数据比较，你会发现已经得到了想要的结果。第二列是 4+10、12+12、20+14、28+16；第三列是 6+18、14+20、22+22、30+24。

10.3.9 选择元素的任意子集

如果想要比 slice 和 gslic 提供的 valarray 的常规索引更灵活地访问 valarray 元素，可能需要一些时间。在这种情况下，可以用包含任意组索引值的 valarray<size_t>对象作为 valarray<T>对象的下标。结果会得到一个 indirect_array<T>类型的对象，其中封装了在索引值位置的元素的引用。注意索引值必须是 size_t 类型；valarray<int>将无法工作。

indirect_array 对象和 slice_array 对象有相同的成员函数，因此可以用它做同样的事情。valarray 的一个构造函数可以将 indirect_array<T>类型隐式转换为 valarray<T>类型。

用 valarray<size_t>对象可以选择数组中元素的任何组合，但不应该复制索引值。如果 indirect_array 对象复制的是引用，对于操作结果是未定义的；有时候可能也能工作，但不是所有时候都能。下面是一个从 valarray 中选择任意组元素的示例：

```
valarray<double> data {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32};
std::valarray<size_t> choices {3, 5, 7, 11}; // Indexes to select arbitrary elements
print(valarray<double>(data[choices]));  // Values selected: 8 12 16 24
```

```
data[choices] *= data[choices];              // Square the values
print(valarray<double>(data[choices]));  // Result: 64 144 256 576
```

choices 对象包含从 data 对象中选择的 4 个浮点值的索引值。倒数第二条语句对选择的值求平方，执行这些语句的结果显示在注释中。因为 choices 是一个 valarray，可以通过对一组索引值的算术运算来生成新的组。例如：

```
size_t incr{3};
data[choices+incr] += valarray<double>(incr+1, choices.size());
// Add 4 to selected elements
print(valarray<double>(data[choices+incr]));                    // 18 22 26 34
```

表达式 choices+incr 会生成新的 valarray<size_T>对象，对象包含的索引值来自于 choice 中的增量 3，因此它包含的是 6、8、10、14。用新的 valarray 对象作为 data 的下标运算符的参数会返回一个 indirect_array<double>对象，它包含的是 data 中值为 14、18、22、30 的元素的引用。+=运算会将作为右操作数的 valarray<double>中的对应元素的值加到这些值上，它们都会加上 4。

10.3.10　有条件地选择元素

在前面已经看到，可以用比较运算符来将 valarray<T>对象中的元素和另一个 valarray<T>对象中的对应元素作比较，或者和一个 T 类型的值作比较。在每种情况下，结果都会得到是一个元素值是元素比较结果的 valarray<bool>对象。可以将一个 valarray<bool>对象传给 valarray 的下标运算符来选择对应为 true 的元素。因此我们有了一种在任何情况下都能选择元素的方法。使用 valarray<bool>对象作为下标的结果是得到一个 mask_array<T>对象，数组中正在访问的就是 T 类型的值。mask_array 对象和 slice_array 有相同的功能。下面的示例有些牵强：

```
std::uniform_int_distribution<int> dist {0, 25};
std::random_device rd;                          // Non-deterministic seed source
std::default_random_engine rng {rd()};   // Create random number generator
std::valarray<char> letters (52);

for(auto& ch: letters)
  ch = 'a' + dist(rng); // Random letter 'a' to 'z'
print(letters,26, 2);

auto vowels = letters == 'a'||letters =='e'|| letters == 'i' ||
              letters == 'o' || letters == 'u';
valarray<char> chosen {letters[vowels]};   // Contains copies of the vowels
                                           // in letters
size_t count {chosen.size()};              // The number of vowels
std::cout << count << " vowels were generated:\n";
print(chosen, 26, 2);

letters[vowels] -= valarray<char>('a'-'A', count);
print(letters, 26, 2);
```

这段代码演示了如何有条件地选择元素——这肯定不是最好的实现方式。回顾第 8 章，正好做了几件事。letters 对象保存了 char 类型的元素，并且用均匀离散分布在基于序列的 for 循环中用随机的小写字母填充它。分布 dist 生成的值的范围为从 0 到 25，因此在循环中可以得到从'a'到'z'的字母。vowels 对象是通过将 letters 中的元素和元音字母相或得到的。每次比较生成的 valarray<bool>对象和 letters 有相同的元素个数。当对应元素是元音时，对象中会有值为 true 的元素。或的组合(使用非成员函数 operator||())可以得到一个 valarray<bool>对象，当 letters 中相应的元素是任何小写字母时，它会包含值为 true 的元素。用 valarray<bool>对象 vowels 作为 letters 的下标可以得到引用了 letters 中元音字母的 mask_array<char>对象。最后，会调用 mask_array<char>的成员函数 operator-=()来从 letters 的元素中减去'a'和'A'，因此它们会被转换为大写字母。

得到的输出如下：

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

13 vowels were generated.
a i o i o i i i e o e a o
d A v I d h O t x v I v d O p I I n d q p g r q f s


g I c E w O b r E t A b w l l q j h x f j h n p O y
```

这个输出使人备受鼓舞。前 8 个随机字母可以说明，如果数组足够大，执行代码的时间足够长，就能得到一本完整的莎士比亚作品。

10.3.11 有理数算法

ratio 头文件定义了 ratio 模板类型，这是一种在很多方面都很奇怪的类型，特别是因为它做的每件事都是在编译期而不是运行期做的。不需要生成对象——只需要一个定义类型的 ration 类模板的实例。如果想使用后面会讨论的时钟和定时器的话，理解 ratio 类模板是必需的。

有理数只是一小部分——两个整数的比率。我们知道，许多小数不能精确地表示成二进制数或十进制数。例如，2/3 就既不能用二进制，也不能用十进制来表示；两种表示都需要无限个数才能表示精确，所以很多有理数的浮点表示总会和它们的准确值有偏差。这种误差很小，当然，24 位尾数的误差不会大于 2^{-24}。这是可以忽略不计的——除非想用此类值做计算。假设一个有理数有准确的值 V，但在浮点数中它的值是 V-e，这个误差非常小。让我们看一个浮点数乘以自身的示例。结果为(V-e)*(V-e)，展开后是 $V^2-2Ve+e^2$。我们想要的准确结果是 V^2，因此剩下的都是正确结果的偏差。e^2 部分是 2^{-48}，我们可以忽略，但是剩下的，2Ve 就不能忽略了。这个计算结果的误差是原值中误差的 2V 倍，而且这个误差会随着后面的计算增加。ratio 模板和 ratio 头文件中定义的其他模板类型提供了一种方式来解决这个问题——至少是在编译期。

ratio 模板定义了一些表示有理数的类型，这些有理数是由两个 intmax_t 类型的整数值的分子和分母定义的。intmax_t 定义在 cstdint 头文件中，是我们可得到的最大范围的整数类型。注意它是表示有理数而不是对象的类型。因此可以用下面的类型表示 2/3：

```
using v2_3 = std::ratio<2, 3>; // A type to represent two thirds
```

v2_3 类型的模板参数是这个有理数的分子和分母的值。这些值被各自保存在类类型的静态成员 num 和 den 中，它们是 const 型静态成员，因此不能在定义类型后修改。参数 den 的默认值是 1，因此可以通过指定第一个类型参数来定义表示整数的类型。例如，ratio<99>是表示值 99 的类型。假定后面的代码都为 std::ratio 使用了 using 指令，这样就可以去掉 std 命名空间限定符了。

ratio 类型总是会用它的最简形式来表示数。例如，如果定义 ratio<4,8>类型，num 和 den 分别为 1 和 2。可以用下面的语句输出 v2_3 在运行时表示的数:

```
std::cout << "The v2_3 type represents " << v2_3::num << "/" << v2_3::den
          << std::endl;
```

ratio 类型之间的算术运算是由更高级的模板类型定义的，所以编译器可以做这个工作。下面展示了如何将 2/3 和 3/7 相加：

```
using v2_3 = ratio<2, 3>;                  // A type to represent two thirds
using v3_7 = ratio<3, 7>;                  // A type to represent three sevenths
using sum = std::ratio_add<v2_3, v3_7>; // A type representing the sum of 2/3
                                           // and 3/7
std::cout << sum::num << "/" << sum::den << std::endl; // Output: 23/21
```

ratio_add<T1,T2>的实例是 ratio 模板 ratio<T3,T4>的特化。T3 和 T4 对应于分子和分母相加后的结果。因为是 ratio 类型的特化，sum 类型有静态成员 num 和 den，它们会为从这个运算得到的无理数表示分子和分母，所以我们能够输出它们。

不需要为每 ratio 类型都定义别名，编译器会推导类型。可以将加法定义为：

```
using sum = ratio_add< ratio<2, 3>, ratio<3, 7>>; // A type for the sum of
                                                  // 2/3 and 3/7
```

这和前面 sum 别名的定义所做的事相同。这里有一些表示有理数之间算术运算的其他模板类型：

- ratio_subtract<T1, T2>类型的实例是 ratio 类型，表示的是 T2 类型所表示的值减去 T1 类型所表示的值的结果。
- ratio_multiply<T1, T2>类型的实例是 ratio 类型，表示的是 T2 类型和 T1 类型所表示的值的乘积。
- ratio_divide<T1, T2>类型的实例是 ratio 类型，表示的是 T1 类型所表示的值除以 T2 类型所表示的值的结果。

所有这些工作都是在编译时完成的，会生成 ration 模板的一个特化，并且可以将它用到任何组合中。result 是 ratio 类型，因此 result 总是显示它的最简形式。这样可以减小多

个算术运算后，分子或分母超过整数类型的最大值的风险。它也会检查分母是否为 0，下面是一个使用 ratio 类型实例的示例。

```
using result = std::ratio_multiply<std::ratio_add<ratio<2, 3>, ratio<3, 7>>,
  ratio<15>>;
std::cout << result::num << "/" << result::den << std::endl; // Output: 115/7
```

result 的定义会从(2/3+3/7)*15 中生成一个 ratio 实例，它所表示的值显示在注释中。

有一些模板可以表示两个 ratio 类型所表示值的比较结果，这些比较操作显然和模板类型的名称是一样的。

```
ratio_equal<RT1, RT2>  ratio_less<RT1, RT2>  ratio_less_equal<RT1, RT2>
ratio_not_equal<RT1, RT2> ratio_greater<RT1, RT2> ratio_greater_equal<RT1, RT2>
```

模板参数是表示有理数的 ratio 模板的实例。每个比较模板类型都有一个静态布尔成员 value，如果 ratio 类型的数字的比较表示真正的结果，可以在运行时用它来检查 ratio 实例之间的关系：

```
using div1 = std::ratio_divide<ratio<7, 10>, ratio<11, 7>>;
using div2 = std::ratio_divide<ratio<9, 5>, ratio<3, 7>>;
std::cout << "(7/10)/(11/7) "
          << (std::ratio_greater<div1, div2>::value ? "is" : "is_not")
          << " greater than (9/5)/(3/7)" << std::endl;
```

要比较 ratio 类型的所有类型都定义了 operator bool()函数，这个函数会调用 operator()()。前者允许将比较类型的对象隐式转换为布尔类型。所以可以这样写：

```
std::ratio_greater<div1, div2> cmp;
std::cout << "(7/10)/(11/7) " << (cmp ? "is" : "is_not")
          << " greater than(9/5)/(3/7)"
          << std::endl;
```

第一条语句生成了一个表示 ratio 类型的比较结果的对象。在输出语句中，通过调用 cmp 的成员函数 operator bool()将 cmp 隐式转换为布尔类型：

也可以这样写输出语句：

```
std::cout << "(7/10)/(11/7) " << (cmp() ? "is" : "is_not")
          << " greater than (9/5)/(3/7)"
          << std::endl;
```

这会调用 cmp 对象的 operator()()，它会返回值成员，因此结果是相同的。当然，比较的结果是 false。

ratio 头文件也为表示 SI 比率的 ratio 类型的实例定义了下面这些别名：

SI 前缀	值	类型别名	值
deca	10	deci	10^{-1}
hecto	10^2	centi	10^{-2}

kilo	10^3	milli	10^{-3}
mega	10^6	micro	10^{-6}
giga	10^9	nano	10^{-9}
tera	10^{12}	pico	10^{-12}
peta	10^{15}	femto	10^{-15}
exa	10^{18}	atto	10^{-18}
zetta	10^{21}	zepto	10^{-21}
yotta	10^{24}	yocto	10^{-24}

这些表示整数常量的类型都有第二个模板参数，因此成员变量 den 的默认值是 1；对于其他一些成员，它是第一个模板参数，因此成员 num 为 1。如果在你的系统上，intmax_t 类型表示的最大值不够大，yocto、zepto、etta、yotta 这些类型不会被定义。这些常量对于减小出错的概率是非常有用的，尤其在想使用非常大或非常小的 SI 比率时。很容易犯多几个或少几个 0 的错误。

所展示的语句说明了 ratio 模板类型是如何工作的，但它是用于做什么的？不能用它在编译期做大量的计算。它的目的是允许在编译期，通过指定模板参数值来轻松定义有理数。在编译期，执行任何必要的算术运算都能够帮助避免出现溢出。在下一节你会看到一个要求我们提供 ratio 模板类型作为模板类型参数值的 STL 模板。

10.4　时序模板

在程序中经常会用到时间间隔。游戏程序显然就是一个可能会用到它的场景，在很多程序中也需要测量执行的性能。当然，测量时间并不仅仅只涉及软件。底层的硬件会提供时钟和间隔定时的能力，你的实现所提供的 STL 功能是硬件通过操作系统提供的接口。STL 提供的所有时间功能最终都会通过操作系统提供的接口和硬件的定时器联系到一起。

chrono 头文件定义了和时间间隔或持续时间、实时、时钟关联的类和类模板。后面你会看到，可能会想要使用 ctime 头文件提供的功能和时钟得到的时间。chrono 头文件中的所有名称都被定义在 std::chrono 命名空间中。时间间隔、实时和时钟都是相关的，它们之间的关系如下：

- duration(持续时间)是定义为时间刻度数的时间间隔，可以指定一个时间刻度是多少秒。因此，时间刻度是衡量时间长短的基础。duration 模板的实例类型的对象定义了 duration。时间刻度所表示的默认时间间隔是 1 秒，但可以将它定义为更多秒或秒几分之一。例如，如果定义时间刻度为 3600 秒，但意味着 10 个 duration 就是 10 个小时；也能够定义时间刻度为一秒的十分之一，在这种情况下，10 个 duration 表示 1 秒。
- clock(时钟)记录的是从给定的时间开始的时间流逝——被称作时期。时期是固定的时间点。有 3 个封装了硬件时钟的类，后面会介绍它们。时间是按刻度衡量的，所以时期会定义给定的时钟确定刻度段的持续时间。

- 时间的实例称作时间点，由一个 time_point 类模板类型的对象表示。时间点是相对于开始时间的时间长度，这里时间的开始点是时钟定义的时期；因此给定的时间点是由提供时期的时钟以及定义相对于这个时期和时刻段的个数的持续时间定义的。

让我们看看如何定义持续时间，能用它做什么。

10.4.1 定义 duration

chrono 头文件中的 std::chrono::duration<T,P>模板类型表示 duration。模板参数 T 是值的类型，一般是基本的算术类型，并且参数 P 对应的值是时间刻度所表示的秒数，它所对应的值为 1 秒。必须用 ratio 类型指定 P 的值，默认为 ratio<1>。下面是 duration 的一些示例：

```
std::chrono::duration<int,
std::milli> IBM650_divide {15}; // A tick is 1 millisecond so 15 milliseconds
std::chrono::duration<int> minute {60}; // A tick is 1 second by default
                                        // so 60 seconds
std::chrono::duration<double, ratio<60>> hour {60}; // A tick is 60 seconds
                                        // so 60 minutes
// A tick is a microsecond so 1 millisecond
std::chrono::duration<long, std::micro> millisec {1000L};
// A tick is fifth of a second so 1.1 seconds
std::chrono::duration<double, ratio<1,5>> tiny {5.5};
```

第 1 条语句使用从 ratio 头文件中获取的对应于 ratio<1,1000>的别名 milli。第 2 条语句省略了第二个模板参数的值，因此是 ratio<1>，这意味着持续时间以 1 秒为单位。在第 3 条语句中，ratio<60>模板参数值指定了时间刻度为 60 秒，因此 hour 对象的值是按分钟测量的，它的初值表示 1 小时。第 4 条语句使用了 ratio 头定义的 micro 类型来定义 ratio<1, 1000000>，因此 tick 是毫秒，millisec 变量有一个表示毫秒的初值。最后一条语句定义了一个时间刻度为 1/5 秒的对象，初值是 5.5 的五分之一，它是 1.1 秒。

chrono 头文件为有整数类型值的常用持续时间类型在 std::chrono 命名空间中定义了别名。标准别名是：

```
nanoseconds<integer_type, std::nano> microseconds<integer_type, std:: micro>
milliseconds<integer_type, std::milli>  seconds<integer_type>
minutes<integer_type, std::ratio<60>> hours<integer_type, std::ratio <3600>>
```

这些别名中，每个整数类型的持续时间值都取决于实现，但 C++14 标准要求 duration 至少为 292 年，可以是正数或负数。在笔者系统上小时和分钟类型将 duration 保存为 int 类型，将其他的保存为 long long 类型。因此前面的代码段可以这样定义 millisec 变量：

```
std::chrono::microseconds millisec {1000}; // Duration is type long long
                                           // on my system
```

当然，这个定义和原始定义不同。变量的初值表示的是相同的时间间隔——1 毫秒——但这里持续时间的单位是 1 毫秒，而在先前的代码中是 1 微秒。前面的 millisec 的定义允

许更精确地表示持续时间。

所有可以应用到 duration 对象的算术运算符，都可以被用到左操作数是 duration 对象的复合赋值中，+=和-+这些运算的右操作数必须是 duration 对象。*=和/=的右操作数必须和作为左操作数的时间刻度数有相同类型的数值，或者可以隐式转换为数值。%=的右操作数可以是 duration 对象或数值。它们中的每一个都能产生我们期望的结果。例如下面使用+=的代码：

```
std::chrono::milliseconds millisec {1}; // Duration is also type long long
                                        // on my system
```

第一个+=运算的操作数为相同类型，作为右操作的对象所保存的值会被加到左操作数上。第二个+=运算的操作数为不同类型，但右操作数会被隐式转换为左操作的类型。这是可能的，因为转换是向有较短时间刻度的 duration 类型转换的。如果向相反的方向转换就不行——所以不能在+=运算中用 ong_time 作为左操作数、用 short_time 作为右操作数。

1. duration 之间的算术运算

可以将前缀或后缀自增和自减运算符应用到 duration 对象上，并且可以通过调用成员函数 count()来得到时间刻度数。下面是示例代码：

```
std::chrono::duration<double, ratio<1,5>> tiny {5.5}; // Measured in 1/5 second
std::chrono::microseconds very_tiny {100}; // Measured in microseconds
++tiny;
very_tiny--;
std::cout << "tiny = " << tiny.count()
          << " very_tiny = " << very_tiny.count()
          << std::endl;                          // tiny = 6.5 very_tiny = 99
```

可以将任何二元算术运算符+、-、*、/、%应用到 duration 对象上，会得到一个 duration 对象作为结果。这些都是作为非成员运算符函数实现的。下面是一个示例：

```
std::chrono::duration<double, ratio<1,5>> tiny {5.5};
std::chrono::duration<double, ratio<1,5>> small {7.5};
auto total = tiny + small;
std::cout << "total = " << total.count() << std::endl; // total = 13
```

算术运算符也可以用不同类型的 std::chrono::duration<T,P>模板实例作为操作数，模板的两个参数都可以不同。这是通过使用 common_type<class... T>模板的特化将两个操作数转换为它们的共有类型来实现的，common_type<class... T>定义在 type_traits 头文件中。对于 duration<T1, P1>和 duration<T2, P2>类型的参数，返回值为 duration 类型 duration<T3, P3>。T3 是 T1 和 T2 的共有类型。这些类型是通过将算术运算应用到这些类型的值来得到的。

P3 是 P1 和 P2 的最大公因数。有一个示例会更清楚一些：

```
std::chrono::milliseconds ten_minutes {600000}; // A tick is 1 millisecond
                                                // so 10 minutes
```

```
std::chrono::minutes half_hour {30};   // A tick is 1 minute so 30 minutes
auto total = ten_minutes + half_hour;  // 40 minutes in common tick period
std::cout << "total = " << total.count()
          << std::endl;                // total = 2400000
```

加法的结果是 40 分钟，因此可以推断出 total 是毫秒类型的对象。下面是另一个示例：

```
std::chrono::minutes ten_minutes {10};                  // 10 minutes
std::chrono::duration<double, std::ratio<1,5>> interval {4500.0}; // 15 minutes
auto total_minutes = ten_minutes + interval;
std::cout << "total minutes = " << total_minutes.count()
          << std::endl;                          // total minutes = 7500
```

total_minutes 的值是 double 类型。我们知道结果必定是 25 分钟，也就是 1500 秒；结果为 7500，因此时间刻度的长度为 ratio<1,5>——1/5 秒。最好尽可能地避免对混合的 duration 类型进行算术运算，因为很容易不知道时间刻度是什么。

所有可以应用到 duration 对象的算术运算都可以用到左操作数是 duration 的复合赋值中。+=和-=的右操作数必须是 duration 对象。*=和/=的右操作数必须和左操作数的时钟刻度类型相同，或者可以隐式转换为那种类型。%=的右操作数可以是 duration 对象或数值。它们中的每一个都能产生我们想要的结果。例如，下面是使用+=的示例：

```
std::chrono::minutes short_time {20};
std::chrono::minutes shorter_time {10};
short_time += shorter_time;          // 30 minutes
std::chrono::hours long_time {3};  // 3hrs = 180 minutes
short_time += long_time;
std::cout << "short_time = " << short_time.count() << std::endl;
// short_time = 210
```

第一个+=运算的操作数都为相同类型，因此右操作对象保存的值会被加到左操作数上。第二个+=运算的操作数为不同类型，但是右操作数可以隐式转换为左操作数的类型。这是可能的，因为转换的是短时钟周期的 duration 类型。相反，就不能转换，因此不能在用+=时以 long_time 作为左操作数，以 short_time 作为右操作数。

2. duration 类型之间的转换

通常，一个 duration 类型总是可以被隐式转换为另一个 duration 类型，如果它们都是浮点型 duration 值的话。当源类型的时钟周期是目的类型的时钟周期的整数倍时，只能对整数值进行隐式转换。下面是一些示例：

```
std::chrono::duration<int, std::ratio<1, 5>> d1 {50};  // 10 seconds
std::chrono::duration<int, std::ratio<1, 10>> d2 {50}; // 5 seconds
std::chrono::duration<int, std::ratio<1, 3>> d3 {45};  // 15 seconds
std::chrono::duration<int, std::ratio<1, 6>> d4 {60};  // 10 seconds
d2 += d1;   // OK - implicit conversion of d1
d1 += d2;   // Won't compile 1/10 not a multiple of 1/5
d1 += d3;   // Won't compile 1/3 not a multiple of 1/5
```

```
d4 += d3;   // OK - implicit conversion of d3
```

可以显式地使用 duration_cast 模板进行强制转换。下面是一个示例，假设 d1 和 d2 都有上述代码定义的初值：

```
d1 += std::chrono::duration_cast<std::chrono::duration<int, std::ratio<1,
  5>>>(d2);
std::cout << d1.count() << std::endl; // 75 - i.e. 15 seconds
```

第一条语句会用 duration_cast 来使 d2 加到 d1 的运算能够进行。在这个示例中，结果是准确的，但并不总是如此。例如：

```
std::chrono::duration<int, std::ratio<1, 5>> d1 {50};    // 10 seconds
std::chrono::duration<int, std::ratio<1, 10>> d2 {53};   // 5.3 seconds
d1 += std::chrono::duration_cast<std::chrono::duration<int, std::ratio<1,
  5>>>(d2);
std::cout << d1.count() << std::endl; // 76 - i.e. 15.2 seconds
```

不能将 duration 值 d1 和 d2 的和表示成 0.2 秒的整数倍，因此结果值稍微超出了。如果 d2 的值是 54，得到的正确结果应该是 77。

duration 类型支持赋值，因此可以将一个 duration 对象的值赋给另一个 duration 对象。如果与这一节开始时描述的条件相符，就可以使用隐式转换；否则需要对右操作数显式转换。

例如，可以这样写：

```
std::chrono::duration<int, std::ratio<1, 5>> d1 {50};    // 10 seconds
std::chrono::duration<int, std::ratio<1, 10>> d2 {53};   // 5.3 seconds
d2 = d1;                                                 // d2 is 100 = 10 seconds
```

3. 比较 duration

有一整套完整的运算符可以用来比较两个 duration 对象。它们都是由非成员函数实现的，允许比较不同类型的 duration 对象。这个过程可以确定操作数共同的时钟周期，当表示成相同的时钟周期时，就可以比较 duration 的值。例如：

```
std::chrono::duration<int, std::ratio<1, 5>> d1 {50};    // 10 seconds
std::chrono::duration<int, std::ratio<1, 10>> d2 {50};   // 5 seconds
std::chrono::duration<int, std::ratio<1, 3>> d3 {45};    // 15 seconds
if((d1 - d2) == (d3 - d1))
std::cout << "both durations are "
          << std::chrono::duration_cast<std::chrono::seconds>(d1 - d2).count()
          << " seconds" << std::endl;
```

这里显示了从算术运算得到的 duration 对象的比较方法。它们是相等的，当然也会产生这样的输出。seconds 类型的强制转换允许秒数表示成整数，而不用管结果为 duration 类型。如果想使用非整数的秒数值，可以强制转换为 duration<double>类型。

4. duration 常量

chrono 头文件中定义了可以让我们指定 duration 对象的常量的运算符。这些运算符被

定义在命名空间 std::literals::chrono_literals 中，命名空间 literals 和 chrono_literals 都是内联的。有了下面的声明之后，就可以对 duration 常量使用常量运算符：

```
using namespace std::literals::chrono_literals;
```

但是，如果指定了下面的声明，上面的声明会被自动包含：

```
using namespace std::chrono;
```

可以将 duration 常量指定为整数或浮点值，后缀指定了时钟周期。这里有 6 个可以使用的后缀：

- h 是小时，例如 3h 或 3.5h。
- min 是分钟，例如 20min 或 3.5min。
- s 是秒，例如 10s 或 1.5s。
- ms 是毫秒，例如 500ms 或 1.5ms。
- us 是微秒，例如 500us 或 0.5us。
- ns 是纳秒，例如 2ns 或 3.5ns。

如果一个 duration 常量有整数值，它会从这些被定义在 chrono 头文件中的常量得到一个合适的别名。因此 24h 是一个 std::chrono::hours 类型的常量，25ms 的类型是 std::chrono::milliseconds。如果常量的值不是整数，这个常量会是浮点值类型的 duration 类型。浮点值的周期取决于后缀；对于后缀 h、min、s、ms、us 和 n，时钟周期分别为 ratio<3600>、ratio<60>、ratio<1>、milli、micro 和 nano。

下面举例说明使用它们的一些方式：

```
using namespace std::literals::chrono_literals;
std::chrono::seconds elapsed {10};    // 10 seconds
elapsed += 2min; // Adding type minutes to type seconds: 130 seconds
elapsed -= 15s;                       // 115 seconds
```

如示例所示，当需要改变间隔的长短时，duration 常量是非常有用的。需要记住的是，为了能够进行算术运算，作为右操作数的时钟周期必须是作为左操作数的时钟周期的整数倍。例如：

```
elapsed += 100ns; // Won't compile!
```

变量 lapsed 的时钟周期为 1，不能向它加一个周期小于 1 的 duration。

可以用常量将同类型的变量定义为常量。例如：

```
auto some_time = 10s;         // Variable of type seconds, value 10
elapsed = 3min - some_time;// Set to difference between literal and
                           // variable: result 170
some_time *= 2;            // Doubles the value - now 20s
const auto FIVE_SEC = 5s;  // Cannot be changed
elapsed = 2s + (elapsed - FIVE_SEC)/5; // Result 35
```

这里，some_time 是一个 seconds 类型的变量，它的类型为 duration<long long, ratio<1>>,

值为 10。第 3 条语句说明可以改变 some_time 的值。FIVE_SEC 是 const seconds 类型的值，因此不能改变它的值。最后一条语句展示了一个包含一个 duration 常量、一个 duration 变量、一个常量类型的 seconds 对象，以及一个整数常量的算术表达式。

10.4.2　时钟和时间点

STL 定义的时钟类型提供了硬件时钟或操作系统时钟的接口。时钟有时钟周期，时间是通过时钟周期数来衡量的。命名空间 std::chrono 中定义了 3 种时钟：

- system_clock 类封装了当前真实时钟时间。尽管它间一般都是在增加，但在有些时候也会减少。当然，挂钟是可以减少的，在冬季和夏季会按照季节调整。如果移到不同的时区，system_clock 也会改变。
- steady_clock 类封装了一个适合记录时间间隔的时钟。这个时钟总是在增加——不能减小。
- high_resolution_clock 的实例是一个具有当前系统所能使用的最小时钟周期的时钟。在一些实现中，它可能是 system_clock 或 steady_clock 的别名，在这种情况下没有提供额外的 high_resolution_clock。

每个时钟类型都定义了自己的时期和持续时间。持续时间为时钟确定了时钟周期和记录时间间隔相对于时期数的类型。如果时钟记录的时间总是增加，并且是以同样的长度增加——也就是说，时钟周期是一个常量，那么这个时钟就是稳定的。所有的时钟都不是稳定的。system_clock 通常不是稳定的时钟，因为它无法保证时间总是增加，而且系统活动会影响时间刻度之间的时间。这也是为什么选择 steady_clock 类型来测量时间间隔的原因。

每个封装了物理硬件时钟的时钟类型——都是处理器或其他芯片的一部分——但有 3 个时钟类类型并不就意味着必须有 3 个不同的时钟。不需要也没有意义去生成时钟类型。时钟类型通过静态成员提供硬件时钟的接口。

3 个时钟类型都有分有且静态的数据成员，is_steady，它的值为布尔类型。这个成员的值可以说明这个时钟是否是稳定的时钟。steady_clock 的 is_steady 总是为 true，对于其他的时钟类型，可以是 true 或 false，这取决于实现。已经说过，system_clock 的 is_steady 通常为 false。如果代码依赖一个稳定的时钟，就应该总是检查成员 is_steady 的状态——或者使用 steady_clock。检查是否是稳定的时钟很容易：

```
std::cout << std::boolalpha << std::chrono::system_clock::is_steady
          << std::endl;
```

这条语句在笔者系统上输出了 false，可能在你的系统上也是。当然，如果 high_resolution_clock 的类型是 system_clock 的别名，就只有一个稳定时钟。每个时钟类都定义了如下类型的别名作为成员：

- rep 是用来记录时间刻度的算术类型的别名。
- period 是以秒为单位的 ratio 模板类型的别名。
- duration 是 std::chrono::duration<rep, period>的别名。
- time_point 是时钟表示瞬时时间的时间点类型的别名。

因此，当需要知道 **system_clock** 类型的时钟间隔时，可以使用表达式 **system_**clock::period。这是一个 ratio 类型，因此数值为秒数的时间刻度表示的是 system_ clock::period::num 除以 system_clock::period::den。

1. 生成时间点

time_point 对象表示的是相对于某一时刻的瞬时时间，某一时刻是由时钟定义的。所以，time_point 总是基于时刻以及相对于时刻的持续时间定义的。当向时钟请求时间时，会得到一个 time_point 对象。std::chrono::time_point 类模板定义了 time_point 类型。这个模板有两个类型参数：提供某一时刻的时钟类型 Clock 以及 Clock 类型默认定义的 duration 类型。因此当定义 time_point 对象时，第一个模板类型参数值是 std::chrono::system_clock，第二个模板类型参数的默认值是 std::chrono::system_clock::duration。

time_point 对象总是和具体的时钟类型相关联，所以在生成对象时，使用特定时钟类型的成员的 time_point 类型别名是很方便的。但如果愿意，也可以将时钟类型指定为模板参数值。例如：

```
std::chrono::system_clock::time_point tp_sys1; // Default object - the epoch
std::chrono::time_point<std::chrono::system_clock> tp_sys2;
// Default object - the epoch
```

这两条语句都调用了默认的 time_point<system_clock>构造函数。默认构造函数会生成一个为指定的时钟类型表示时刻的对象，因此 duration 是 0。第一条语句更简洁，因此优先选择它。可以通过为时钟和时间点类型定义别名来使代码更简洁，在后面的代码中，就会这样做。

可以用 duration 作为构造函数的参数来生成一个表示瞬时相对于某一时刻的 time_point 对象。duration 对象定义的时间可以加到时刻上：

```
using Clock = std::chrono::steady_clock;
using TimePoint = std::chrono::time_point<Clock>;
TimePoint tp1 {std::chrono::duration<int> (20)}; // Epoch + 20 seconds
TimePoint tp2 {3min};                            // Epoch + 180 seconds
TimePoint tp3 {2h};                              // Epoch + 720 seconds
TimePoint tp4 {5500us};                          // Epoch + 0.0055 seconds
```

这些语句说明传给 time_point 构造函数的 duration 对象可以是任何时钟周期。TimePoint 的别名也可以用下面的指令定义：

```
using TimePoint = Clock::time_point;
```

可以用不同于定义时刻的时钟类型的时间段来定义 time_point 对象。例如：

```
std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>
  tp {2h};
```

这不是我们应该做的事，除非有好的理由。这里用 system_clock 类型定义的时刻定义了一个 time_point 对象 tp。以分钟为单位的初值被作为一个表示两小时的 duration 常量。

2. 时间点的持续时间

可以调用 time_point 对象的成员函数 time_since_epoch()来得到表示从某一时刻到现在已流逝的时间的 duration 对象：

```
using Clock = std::chrono::steady_clock;
using TimePoint = Clock::time_point;
TimePoint tp1 {std::chrono::duration<int> (20)}; // Epoch + 20 seconds
auto elapsed = tp1.time_since_epoch(); // Duration for the time interval
```

现在有了 duration 对象，也就有了它所表示的时间。我们不知道 elapsed 对象的类型，但知道它是 duration 类型，因此可以将它强制转换为已知的 duration 类型。例如，可以这样获取 elapsed 表示的纳秒数：

```
auto ticks_ns = std::chrono::duration_cast<std::chrono::nanoseconds>
  (elapsed).count();
```

ticks_ns 的值是 elapsed 表示的时间间隔中的纳秒数。当然，如果不需要精度为纳秒的时间，可以将它强制转换为其他 duration 类型别名，例如 milliseconds、seconds、hours。可以用 time_since_epoch()函数定义一个函数模板，它能以秒为单位显示任何 time_point 表示的时间间隔：

```
// Outputs the exact interval in seconds for a time_point<>
template<typename TimePoint>
void print_timepoint(const TimePoint& tp, size_t places = 0)
{
  auto elapsed = tp.time_since_epoch(); // duration object for the interval
  auto seconds = std::chrono::duration_cast<std::chrono::duration<double>>
    (elapsed).count();
  std::cout << std::fixed << std::setprecision(places) << seconds
            << " seconds\n";
}
```

用 duration_cast<double>强制转换 elapsed 对象，会生成一个包含 double 值的时间刻度数并且秒为时钟周期的 duration 对象。对这个对象调用 count()会返回一个 double 类型的值，它以秒为单位。这个值在输出时，小数点后需要的位置个数是由第二个参数决定的。在下一节我们会在一个示例中使用 print_timepoint()函数模板。

3. 时间点的算术运算

time_point 对象有一个拷贝赋值运算符，可以用来在 time_point 上加减一个 duration 对象。加减的结果是一个新的 time_point 对象，原有 time_point 的间隔会被 duration 对象调整。加法和减法是由非成员运算符函数实现的。在右操作数是 duration 对象时，也能用+=和-=运算符来加减一个 time_point 对象。这些运算符都是作为左操作的 time_point 对象的成员函数。下面是一个演示这些运算的完整程序：

```
// Ex10_04.cpp
// Arithmetic with time-point objects
#include <iostream>              // For standard streams
#include <iomanip>               // For stream manipulators
#include <chrono>                // For duration, time_point templates
#include <ratio>                 // For ratio templates
using namespace std::chrono;

// Function template for print_timepoint() goes here...

int main()
{
  using TimePoint = time_point<steady_clock>;
  time_point<steady_clock> tp1 {duration<int>(20)};
  time_point<system_clock> tp2 {3min};
  time_point<high_resolution_clock> tp3 {2h};
  std::cout << "tp1 is ";
  print_timepoint(tp1);
  std::cout << "tp2 is ";
  print_timepoint(tp2);

  std::cout << "tp3 is ";
  print_timepoint(tp3);

  auto tp4 = tp2 + tp3.time_since_epoch();
  std::cout << "tp4 is tp2 with tp3 added: ";
  print_timepoint(tp4);

  std::cout << "tp1 + tp2 is ";
  print_timepoint(tp1 + tp2.time_since_epoch());

  tp2 += duration<time_point<system_clock>::rep, std::milli> {20'000};
  std::cout << "tp2 incremented by 20,000 milliseconds is ";
  print_timepoint(tp2);
}
```

为命名空间 std::chrono 使用 using 指令可以无限定符地使用类型名称，并且可以隐式包含含有 duration 常量的运算符函数的命名空间。这个示例说明，time_point 对象的各种算术运算会涉及不同类型的时钟。这些算术运算总是会涉及将 duration 加到 time_point 上。从 time_point 对象得到的 duration 并不知道时间间隔所来自的时钟的具体信息，因此可以将它加到一个基于不同类型的时钟的 time_point 上。输出能够更清楚加以说明：

```
tp1 is 20 seconds
tp2 is 180 seconds
tp3 is 7200 seconds
tp4 is tp2 with tp3 added: 7380 seconds
tp1 + tp2 is 200 seconds
tp2 incremented by 20,000 milliseconds is 200 seconds
```

可以将 time_point 对象强制转换为具有不同 duration 的 time_point 类型的对象。新对

象和原对象有来自同一时钟的时刻。如果目的 duration 的精度比源 duration 低，在这个过程中会有数据丢失。命名空间 std::chrono 中的 time_point_cast 模板可以做这种转换；模板类型参数值是一个新的 duration 类型。例如：

```
using TimePoint = std::chrono::time_point<std::chrono::system_clock,
  std::chrono::seconds>;
TimePoint tp_sec {75s};          // 75 seconds
auto tp_min = std::chrono::time_point_cast<std::chrono::minutes>(tp_sec);
print_timepoint(tp_min);         // 60 seconds
```

因为强制转换到分钟，丢失了源 duration 中的 15s。

4. 比较时间点

可以用运算符==、!=、<、<=、>=和>中的任何一个来比较给定时钟的两个 time_point 对象。比较的结果是通过比较为操作数调用 time_since_epoch()后得到的结果产生的。虽然 time_point 对象所关联的时钟必须相同，但它们可以有不同的时钟周期，并且比较也需要考虑到这一点。下面是展示它们用法的示例：

```
using TimePoint1 = std::chrono::time_point<std::chrono::system_clock>;
using TimePoint2 = std::chrono::time_point<std::chrono::system_clock,
                   std::chrono::minutes>;
TimePoint1 tp1 {120s};
TimePoint2 tp2 {2min};
std::cout << "tp1 ticks: " << tp1.time_since_epoch().count()
          << " tp2 ticks: " << tp2.time_since_epoch().count() << std::endl;
std::cout << "tp1 is " << ((tp1 == tp2) ? "equal":"not equal") << " to tp2"
          << std::endl;
```

上述代码的输出为：

```
tp1 ticks: 1200000000 tp2 ticks: 2
tp1 is equal to tp2
```

从输出可以看出，tp1 和 tp2 的时间刻度数并不完全相同，因为 ticks 表示的是不同的时间量，但从 system_clock 的时刻开始计量，tp1 和 tp2 表示的是同一瞬间，所以在比较是否相等时，返回了 true。所有比较 time_point 对象的比较运算符都是根据 time_point 所表示的时间来比较的，而不是根据它们的时间刻度数。

5. 时钟的操作

除了每个时钟类型默认的构造函数之外，它们所有的成员函数都是静态的。所有时钟都包含一个含固定时间点、持续时间和 now 函数的时刻。静态成员函数 now()会返回一个表示当前时间的 time_point 对象。所有的时钟类都将下面的别名定义为成员：

- rep 是一个用来记录时钟数的类型。
- period 是一个定义了时钟周期的类型，并且它是 ratio 模板的一个实例。ratio 的静

态成员 num 以及这种类型的成员 den 定义了以秒为单位的时钟周期。

- duration 是记录从某一时刻开始的时钟数的 duration 类型，并且和 std::chrono::duration <rep, period>类型相对应。
- time_point 是时钟的成员函数 now()所返回的时间点的值类型，这里是模板类型 std::chrono::time_point<clock_type>。

除了 3 个时钟类型都实现的 now()函数之外，system_clock 还定义了两个静态成员函数。它们提供了 time_point 类型 std::chrono::time_point<std::chrono::system_clock>对象和定义在 ctime 头文件中的用来表示时间间隔的 std::time_t 类型的对象之间的转换。system_clock 的成员函数 to_time_t()接受一个 time_point 参数，并以 time_t 类型返回它，成员函数 from_time_t()则恰好相反。to_time_t()函数特别有用，因为它能够用 ctime 头文件中提供的功能来将 system_clock 的成员函数返回的 time_point 对象转换为表示日历时间——时间、天、日期的字符串。ctime 头文件是 C 语言的 time.h 头的 C++版本。ctime 头文件中有一些函数不推荐使用，因为它们不安全，但是其中的一些函数在目前的标准库中并没有替代方案。可以用定义在 ctime 文件中的函数以各种不同的方式来获取包含时间、天、日期的字符串。但只会展示如何输出这种信息，剩下的留给你进一步探讨 ctime 头文件：

```
using Clock = std::chrono::system_clock;
auto instant = Clock::now(); // Returns type std::chrono::time_point<Clock>
std::time_t the_time = Clock::to_time_t(instant);
std::cout << std::put_time(std::localtime(&the_time),
                           "The time now is: %R.%nToday is %A %e %B %Y. The time
                           zone is %Z.%n");
```

注意：使用 localtime()可能会导致编译器错误，因为这个函数是不安全的。备选方案并不是标准的 C++——Microsoft Visual Studio 2015 提供的 ocaltime_s()或 Linux 编译器提供的 localtime_r()，因此在这段代码中使用 localtime()。当在你的环境中编译这段代码时，选择其中一个合适的来使用。

此时，在笔者系统上执行这段代码后的输出是：

```
The time now is: 13:27.
Today is Thursday 3 September 2015. The time zone is GMT Summer Time.
```

ctime 头文件中的 localtime 函数接受一个 time_t 对象的指针，并返回一个 tm 类型的内部静态对象的指针。指针会被传给定义在 iomanip 头文件中的 put_time()，它会返回一个对象，这个对象实际上对于它的第一个参数指向的 tm 对象是一个高效的格式化输出函数。第二个参数是一个用来确定如何呈现 tm 对象中所保存的数据的格式化字符串。这里有很多的转换说明符，每一个的前面都是%，它们指定了 tm 对象中的各项数据如何显示，以什么样的顺序显示。

tm 对象是一个包含下面这些 int 类型成员的结构体：

- tm_sec(0-60)、tm_min(0-59)、tm_hour (0-23)分别用来指定时间的秒、分、小时。
- tm_mday(1-31)、tm_mon(0-11)、tm_year 分别指定日期的日、月、年。

- tm_wday(0-6)和 tm_yday(0-365)分别指定一周或一年中的日期。
- tm_isdst 是正值，如果夏令时有效的话；如果夏令时无效，它会是 0；如果信息不可用，它是一个负值。

可以通过 local_time()函数返回的指针来任意访问这些值——例如，像下面这样：

```
std::time_t t = Clock::to_time_t(Clock::now());
auto p_tm = std::localtime(&t);
std::cout << "Time: " << p_tm->tm_hour << ':'
                      << std::setfill('0') << std::setw(2) << p_tm->tm_min
                      << std::endl;                    // Time: 15:06
```

在 put_time()的格式字符串中有一系列用来说明 tm 结构体的成员的格式说明符，并且它们的每个前面都有%字符。例如%H 会将 tm_hour 写成 24 小时的时钟值，%I 会将它写成 12 小时的时钟值，%A 会将 tm_wkday 写成全称，%B 会将 tm_mon 写成月份的全称。tm 对象的成员的格式说明符能够以任意顺序出现，并且在必要时也可以在格式字符串中包含其他的文本，包含%n 可以得到一个新行，包含%t 可以得到一个制表符。tm 的数据成员还有很多其他的格式说明符，它们可以从 C++标准库的 put_time()文档中找到。

6. 定时执行

在程序中能够测量执行所花费的时间是很有用的，这用时钟可以很容易做到。为了说明这一点，我们可以在 Ex10_03 的 main()中添加一些代码来确定解出方程花费了多长时间，并输出花费的时间。下面是 Ex10_05.cpp 的代码：

```
// Ex10_05.cpp
// Determining the time to solve a set of linear equations
#include <valarray>                 // For valarray, slice, abs()
#include <vector>                   // For vector container
#include <iterator>                 // For ostream iterator
#include <algorithm>                // For generate_n()
#include <utility>                  // For swap()
#include <iostream>                 // For standard streams
#include <iomanip>                  // For stream manipulators
#include <string>                   // For string type
#include <chrono>                   // For clocks, duration, and time_point
using std::string;
using std::valarray;
using std::slice;
using namespace std::chrono;

// Function prototypes
valarray<double> get_data(size_t n);
void reduce_matrix(valarray<double>& equations,
    std::vector<slice>& row_slices);
valarray<double> back_substitution(valarray<double>& equations,
    const std::vector<slice>& row slices);
```

```
// Code for print_timepoint() template goes here...

int main()
{
// Code to read the data for the equations as in Ex10_03.cpp...
  auto start_time = steady_clock::now(); // time_point object
  // Code to generate slice objects for rows as in Ex10_03.cpp...
  // Code to solve equations as in Ex10_03.cpp...
  auto end_time = steady_clock::now(); // time_point object
  auto elapsed = end_time - start_time.time_since_epoch();
  std::cout << "Time to solve " << n_rows << " equations is ";
  print_timepoint(elapsed);

  // Code to output the solution as in Ex10_03.cpp...

}
```

我们充分使用了你在本章前面看到的 print_timepoint()函数模板，当然，也需要 Ex10_03 中的 gaussian.cpp。完整的程序可以从下载的代码中得到(Ex10_05)。计时过程只需要在 main()中的解决代码之前加 1 行，在解决代码之后加 4 行。类似的代码可以用于对任何应用的运算计时。在笔者系统上，解 6 个方程式，得到的输出如下：

```
Enter the number of variables: 6
Enter 7 values for each of 6 equations.
(i.e. including coefficients that are zero and the rhs):
 1  1  1  1  1  1   8
 2  3 -5 -1  1  1 -18
-1  5  2  7  2  3  40
 3  1 10  2  1 11 -15
 3 17  5  1  3  2  41
 5  7  3 -4  2 -1   9
Time to solve 6 equations is 0.000219379 seconds

Solution:
x1 =      -2.00
x2 =       1.00
x3 =       3.00
x4 =       4.00
x5 =       7.00
x6 =      -5.00
```

在笔者系统上，解这 6 个方程式，大约花费了 220 毫秒，这效果一点都不差。

10.5 复数

复数是 a+bi 形式的数，其中 a 和 b 是真数——在 C++代码中是浮点值——i 是 $\sqrt{-1}$ 。a 被称作复数的实数部分，b 乘以 i 被称作虚数部分。使用复数的程序一般都很专业，例如，复数可以用于电气和电磁理论、数字信号处理，当然也可以用于数学。复数可以用来生成

非常复杂的 Mandelbrot 集合和 Julia 集合的分形图。因为 STL 为复数提供的功能相对其他功能来说都要少，所以只以相当简单的形式介绍基本知识。如果一点都不了解复数，可以跳过这一节。

complex 头文件定义了用于处理复数的功能。complex<T>模板类型的实例表示的是复数，这里定义了 3 个特化类型：complex<float>、complex<double>、complex<long double>。在这一节中，全部使用 complex<double>，但其他特化类型的操作是基本相同的。

10.5.1　生成表示复数的对象

complex<double>类型的构造函数接受两个参数——第一个参数是实部的值，第二个部分是虚部的值。例如：

```
std::complex<double> z1 {2, 5}; // 2 + 5i
std::complex<double> z; // Default parameter values are 0 so 0 + 0i
```

它也有拷贝构造函数，因此可以按如下方式复制 z1：

```
std::complex<double> z2 {z1}; // 2 + 5i
```

显然，我们需要复数常量以及复数对象，命名空间 std::literals::complex_literals 中定义了 3 个运算符函数，在这个命名空间中，命名空间 literals 和 complex_literals 都是内联定义的。在对 std::literals::complex_literals、std::literals 或 std::complex_literals 使用 using 指令之后，就可以访问用于复数常量的运算符函数。假设使用了一个或多个这种指令，并且 using std::complex 对这一节余下的代码都有效。

运算符""i()函数定义了实部为 0 的 complex<double>类型的常量。因此，3i 是一个等同于 complex<double>{0, 3}的常量。当然，可以用实部和虚部表示复数——例如：

```
z = 5.0 + 3i; // z is now complex<double>{5, 3}
```

这展示了如何定义两部分都是非零值的复数，并顺便说明已经为复数对象实现了赋值运算符。可以对 complex<float>常量使用后缀 if，对 complex<long double>常量使用后缀 il，例如 22if 或 3.5il。这些后缀是由函数 operator" "if()和 operator" "il()定义的。注意，不能写成 1.0+i 或 2.0+il，因为这里的 i 和 il 会被解释为变量名，必须写成 1.0 +1i 和 2.0+1.0il。

所有的复数类型都定义了成员函数 real()和 imag()，它们可以用来访问对象的实部或虚部，或者用提供的参数设置这些部分。例如：

```
complex<double> z{1.5, -2.5};          // z: 1.5 - 2.5i
z.imag(99);                            // z: 1.5 + 99.0i
z.real(-4.5);                          // z: -4.5 + 99.0i
std::cout << "Real part: " << z.real()
          << " Imaginary part: " << z.imag()
          << std:: endl;               // Real part: -4.5 Imaginary part: 99
```

real()和 imag()接受参数的版本什么都不会返回。

有为复数对象实现流的插入和提取的非成员函数模板。当从流中读取一个复数时，它

可能只有实部，例如 55，或者括号中只有实部，例如(2.6)，或者实部和虚部在由一个逗号隔开的括号中，例如(3,-2)。如果只提供了实部，虚部会为 0。下面是一个示例：

```
complex<double> z1, z2, z3;          // 3 default objects 0+0i
std::cout << "Enter 3 complex numbers: ";
std::cin >> z1 >> z2 >> z3;          // Read 3 complex numbers
std::cout << "z1 = " << z1 << " z2 = " << z2 << " z3 = " << z3 << std::endl;
```

下面是示例的输入和输出结果：

```
Enter 3 complex numbers: -4 (6) (-3, 7)
z1 = (-4,0) z2 = (6,0) z3 = (-3,7)
```

如果输入的一个复数没有括号，就不会有虚部。但是，在括号中可以省略虚部。复数的输出周围总是有括号，虚部即使为 0 也会被输出。

10.5.2 复数的运算

complex 类模板为有复数操作数的二元运算符+、-、*、/及一元+和-运算符定义了非成员函数。成员函数定义了+=、-=、*=和/=。下面是使用它们的一些示例：

```
complex<double> z {1,2};             // 1+2i
auto z1 = z + 3.0;                   // 4+2i
auto z2 = z*z + (2.0 + 4i);          // -1+8i
auto z3 = z1 - z2;                   // 5-6i
z3 /= z2;                            // -.815385-0.523077i
```

注意，复数对象和数值常量之间的运算需要数值常量是正确的类型。不能将整数常量(例如 2)加到 complex<double>对象上；为了能够进行这个运算，必须写成 2.0。

10.5.3 复数上的比较和其他运算

一些非成员函数模板可以用来比较两个复数对象相等或不相等。也有==和！=运算可以用来比较复数对象和数值，这里数值会被看作虚部为 0 的复数。为了相等，所有的部分都必须相等，如果操作数的实部或虚部不同，它们就不相等。例如：

```
complex<double> z1 {3,4};            // 3+4i
complex<double> z2 {4,-3};           // 4-3i
std::cout << std::boolalpha
<< (z1 == z2) << " "                 // false
<< (z1 != (3.0 + 4i)) << " "         // false
<< (z2 == 4.0 - 3i) << '\n';         // true
```

注释中的结果很清楚。注意在最后一个比较中，编译器会将 4.0 - 3i 看作复数。

另一种比较复数的方法是比较它们的量。各部分值和复数的实部及虚部都相同的向量的量和复数相同，是两部分平分和的平方根。非成员函数模板 abs()接受 complex<T>类型的

参数，并返回一个 T 类型的量。下面是一个将 abs()函数应用到前面的代码段中定义的 z1 和 z2 上的示例：

```
std::cout << std::boolalpha
<< (std::abs(z1) == std::abs(z2))      // true
<< " " << std::abs(z2 + 4.0 + 9i);     // 10
```

最后的输出值是 10，因为作为 abs()的参数的表达式的计算结果是(8.0+6i)；82 和 62 是 100，平方根是 10。

- norm()函数模板会返回复数的量的平方。
- arg()模板会返回以弧度为单位的相角，是复数 z 对应的 std::atan(z.imag()/z.real())。
- conj()函数模板会返回共轭复数，是 a+bi 和 a-bi。
- polar()函数模板接受量和相角作为参数，并返回和它们对应的复数对象。
- proj()函数模板返回的复数是复数参数在黎曼球上的投影。

一些非成员函数模板提供了一整套的三角函数，并为复数参数提供了双曲函数。也有用于复数参数的 cmath 版本的函数 exp()、pow()、log()、log10()和 sqrt()。下面是一个有趣的示例：

```
complex<double> zc {0.0, std::acos(-1)};
std::cout << (std::exp(zc) + 1.0) << '\n'; // (0, 1.22465e-16) or zero near enough
```

acos(-1)是 π，所以这揭示了欧拉方程令人震惊的真相，π 和欧拉数 e 是有关联的：

$$e^{i\pi}+1=0$$

10.5.4　一个使用复数的简单示例

这个示例会应用复数从可能的无限个数中生成 Julia 集合的分形图。这个示例不会展示出 Julia 集合的美好形象，因为它是基于字符表示的，但能让我们知道它看起来是什么样的。这些分形图通常会被绘制为彩色的图像，但这需要调用系统函数。可以通过点 z 的递归方程在复平面上生成一个二阶 Julia 集合：

$Z_{n+1}=z_n^2+c$，c 是复数常量，c 的值决定了 Julia 集合的形状。

每个新的 z 都是复平面上不同的点。Julia 集合是由复平面上的点组成的，在这个平面上，方程应用无数次，z 的量也不会趋于无穷。当然，需要一个策略来确定 z 是否趋于无穷。在这个程序中，这个方程会被应用到表示每个像素极大次数的复数 z 上，并且如果 z 的量一直小于 2，这个点就在 Julia 集合中。如果大于 2，最有可能会趋向于无穷，因此就不在 Julia 集合中。程序会用 chrono 头文件中的功能来确定生成这个图像需要多长时间。如果字体是方的话，输出的显示效果最好——笔者用的是 8×8 的点阵字体。下面是程序代码：

```
// Ex10_06.cpp
// Using complex objects to generate a fractal image of a Julia set
#include <iostream>          // For standard streams
#include <iomanip>           // For stream manipulators
#include <complex>           // For complex types
#include <chrono>            // For clocks, duration, and time_point
```

```
using std::complex;
using namespace std::chrono;
using namespace std::literals;

// Function template definition for print_timepoint() goes here...

int main()
{
  const int width {100}, height {100};   // Image width and height
  size_t count {100};                    // Iterate count for recursion
  char image[width][height];
auto start_time = steady_clock::now(); // time_point object for start
complex<double> c {-0.7, 0.27015};     // Constant in z = z*z + c

for(int i {}; i < width; ++i)          // Iterate over pixels in the width

{
  for(int j {}; j < height; ++j) // Iterate over pixels in the height
  {
    // Scale real and imaginary parts to be between -1 and +1
    auto re = 1.5*(i - width/2) / (0.5*width);
    auto im = (j - height/2) / (0.5*height);
    complex<double> z {re,im};           // Point in the complex plane
    image[i][j] = ' ';                   // Point not in the Julia set
    // Iterate z=z*z+c count times
    for(size_t k {}; k < count; ++k)
    {
      z = z*z + c;
    }
    if(std::abs(z) < 2.0)                // If point not escaping...
      image[i][j] = '*';                 // ...it's in the Julia set
  }
}
auto end_time = std::chrono::steady_clock::now();
// time_point object for end
auto elapsed = end_time - start_time.time_since_epoch();
std::cout << "Time to generate a Julia set with " << width << "x" << height
          << " pixels is ";
print_timepoint(elapsed, 9);
std::cout << "The Julia set looks like this:\n";
for(size_t i {}; i < width; ++i)
{
for(size_t j {}; j < height; ++j)
std::cout << image[i][j];
std::cout << '\n';
}
}
```

这个程序使用了你在前面见到的 print_timepoint()模板来输出花费的时间。完整的程序代码可以从下载代码中的 Ex10_06.cpp 获取。得到下面的输出——Julia 集合的图形如图 10-13 所示：

```
Time to generate a Julia set with 100x100 pixels is 0.286463017 seconds
The Julia set looks like this:
```

图 10-13　Ex10-06 生成的 Julia 集合

这里涉及很多计算。在笔者系统上大约花了 1/3 秒来计算集合中的点。

10.6　本章小结

定义在 valarray 头文件中的 valarray 类模板旨在使编译器的数值计算比其他有潜力进行并行运算的数组或容器更有效率。valarray 对象提供了在 C++中进行大规模密集计算的基础。slice 类型的对象表示的是一维元素序列，它们以指定间隔分布在 valarrary 所保存的数据中。slice 对象使我们能够在数组的整行或整列中表示操作。gslice 对象是 slice 的泛化，

表示均匀分隔的 slice 对象。gslice 使我们能够将操作表示到它所定义的所有行或列上。

ratio 头文件定义了 ratio 类模板，并且每个 ratio 类型都定义了有理数，因此不再需要，也没有意义去定义 ratio 对象。为了将二元加、减、乘和除运算应用到两个 ratio 类型表示的有理数上，ratio 头文件也定义了下面的类模板：

```
ratio_add<typename R1, typename R2>  ratio_subtract<typename R1, typename R2>
ratio_multiply<typename R1, typename R2>  ratio_divide<typename R1, typename R2>
```

这些模板中的每一个都定义了表示运算结果的新的 ration 类型。也有模板可以通过比较 ratio 类型后得到的结果来生成布尔值。

chrono 头文件定义了作为硬件时钟的接口的类。为时钟定义了如下 3 个类：

- system_clock 表示的是挂钟时间，可以用来确定时间和日期。
- steady_clock 是无变化的时钟，一般用来测量时间间隔。
- high_resolution_clock 是为时间测量提供最高精度的时钟。这个类型的时钟可能是另外两个类型时钟中的某个时钟的别名。

时钟类型只有静态成员，因此不需要定义时钟对象。时钟测量的是相对于某一时刻的时间，称作 epoch(时刻)。time_point 类型实例的成员函数 now()会返回瞬时时间。time_point 类型包含定义了(时刻)和相对于(时刻)的时间间隔的时钟的引用。时间间隔由 duration<typename Rep, typename Period=ratio<1>>模板的实例表示。时间间隔是表示为 Rep 类型值的时钟数。时钟数是作为第二个 duration 模板类型参数的 ratio 类型定义的；默认值 ratio<1>指定 1 个时钟是 1 秒。

complex<T>类模板定义在 complex 头文件中。这个模板的实例是表示复数的实部和虚部都为 T 类型值的类型。complex 模板对类型 float、double、long double 都有特化，定义了一系列全面支持复数运算的函数。

练习

1. 写一个程序，生成均匀分布在1到100之间的100 000个浮点值，并将它们保存到valarray中。假设数组有 100 行，每行有 1000 个元素。用 slice 对象计算并输出每行的平均值。

2. 修改练习 1 中的程序，使它除了能够计算平均值之外，还可以计算每一行的值的标准差，输出完成计算的时间，然后输出平均值和每行的标准差(标准差的公式见第 8 章)。这个程序还应该可以输出执行的日期。

3. 修改练习 2 的解决方案，以纳秒为单位输出用来计算每行的值的标准差所花费的时间。

4. 修改本章的 EX10_06，用 valarray 来保存图像。

5. 如果你是一个代数和复数的爱好者，这个练习就是为你而留。写一个程序，读取任意二次方程的系数：$ax^2 + bx + c = 0$

使用标准公式，用复数对象来确定和输出等式的根：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{4ac}$$