深入C++系列

《C++沉思录》作者作品
美国斯坦福大学C++教材

Accelerated C++ 中文版

Practical Programming by Example

[美] Andrew Koenig　Barbara E. Moo 著
覃剑锋 柯晓江 蓝图 等 译
王昕 校

Addison Wesley

中国
www.in

深入 C++ 系列

Accelerated C++ 中文版

Practical Programming by Example

[美] Andrew Koenig　Barbara E. Moo 著
覃剑锋 柯晓江 蓝图 等 译
王昕 校

Addison Wesley

中国电力出版社

Accelerated C++ (ISBN 0-201-70353-X)
Andrew Koenig Barbara E. Moo
Authorized translation from the English language edition, entitled Accelerated C++, published by Addison Wesley, Copyright©2000
All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2003

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2001-2336 号

图书在版编目（CIP）数据
Accelerated C++中文版 /（美）克尼格著；覃剑锋等译.
北京：中国电力出版社，2003
ISBN 7-5083-1819-6
Ⅰ.A... Ⅱ.①克...②覃... Ⅲ.C 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 100502 号

丛 书 名：深入C++系列
书　　名：Accelerated C++中文版
编　　著：（美）Andrew Koenig Barbara E. Moo
翻　　译：覃剑锋 柯晓江 蓝图 等
审　　校：王昕
责任编辑：陈维宁
出版发行：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092 1/16　印　张：22　字　数：482 千字
书　　号：ISBN 7-5083-1819-6
版　　次：2003 年 12 月北京第 1 版　2003 年 12 月第 1 次印刷
定　　价：39.50 元
版权所有 翻印必究

前　言

教授 C++编程的一种新方法

我们假设，读者们希望迅速地掌握编写实用 C++程序的方法。因此，我们会首先解释 C++的最有用的部分。如果我们这样去理解的话，那这个策略看上去似乎是显而易见的，不过，它却有着根本的含意——那就是，即使 C++是以 C 为基础的，但是我们也并不会从 C 的教学开始。相反，我们从一开始就使用了高级数据结构，而且只会在以后才去解释这些数据结构所依赖的基础。这个方法可以让你马上开始编写地道的 C++程序。

从另一个角度来看，我们的方法也是很独特的：我们集中注意力来解决问题，而不是专门去探究语言和库的特征。当然，我们也会解释这些特征，但这样做的目的是为程序提供支持，而不是用程序来作为演示特征的工具。

因为本书是教授 C++程序设计而不单单是讲解语言特征的，所以对那些已经具备一定 C++基础并想以更自然、更高效的风格使用这门语言的读者来说，它尤为有用。C++的初学者会很注重语言技巧的学习，但是他们却常常不懂得如何运用它们来解决日常的问题。

我们的方法对初学者及熟练的程序员同样适用

在过去的每个暑假中我们都会在斯坦福大学开设为期一周的 C++强化教程。一开始，我们在教学中采用了传统的方法：假定学生已经掌握 C 语言，于是我们的教学从类的定义方法开始，然后会系统地过渡到语言的其他方面。我们发现，学生们会在开始的两三天内感到困惑并且会出现挫败感——直到他们掌握的知识足以让他们编写出实用的程序了，这种现象才会消失。而一旦到达了这种程度，他们就可以很轻松地继续学下去。

当我们接触到一种对崭新的标准库提供足够支持的 C++系统环境的时候，我们就对课程进行了彻底的更新。在新的课程中，我们从一开始就使用了标准库，同时将注意力集中到用来编写实用的程序上。而且，只是在学生掌握了足以让他们高效地使用各种语言细节的知识以后，我们才对语言的细节进行深入的探讨。

结果是很戏剧化的：一天之后，我们的学生就能在课堂上编写出在旧教程中要花费他们大半周时间的程序。而且，他们的挫败感也消失得无影无踪。

抽象

我们的方法之所以适用，是因为 C++（以及我们对它的了解）已经渐趋成熟了。这

熟可以让我们忽略许多为早期的C++程序和程序员所依赖的低层次的概念。

这种允许忽略细节的能力是成熟技术的一个特征。例如，早期的汽车是经常会出故障的，因此每一个司机都迫于无奈地变成了业余的技工。那时候，在不懂得如何解决驾驶中的突发问题的情况下，人们是不敢贸然出去驾驶的。今天的司机无需掌握详细的工程知识就可以使用汽车来进行运输了。当然，他们可能会出于其他目的而去学习工程学的细节，但那又是另外一回事了。

我们把抽象定义为可供选择的忽略（把注意力集中于与手头上的任务相关的概念而忽略了其他所有的枝节）我们认为这是现代程序设计中最重要的方法。编写一个成功的程序的关键在于是否清楚问题的哪些部分应给予考虑，哪些部分应该被忽略。每一种程序设计语言都提供了工具来让我们创造有用的抽象，而每个成功的程序员都应该懂得如何使用这些工具。

我们认为抽象是非常实用的，因此在本书中到处都可以见到抽象。当然，我们通常并不直接称之为抽象，因为它们是以各种各样的形式出现的。相反，我们提到了函数、数据结构、类以及继承——所有这些都是抽象。我们不但是提及它们，而且在书中我们还会常常使用它们。

如果我们能够很好地去设计和选择抽象，那么我们就有理由相信，即使在不了解其所有细节的情况下，我们也可以正确地使用它们。我们无需成为机械工程师就可以驾驶汽车，同样的道理，我们在使用C++之前也无需了解关于C++运作的所有细节。

覆盖范围

如果你是以严谨的态度去看待C++程序设计的话，那你就有必要去掌握我们在本书中所介绍的所有知识点——尽管这本书并没有向你介绍你需要知道的一切知识。

这句话听起来有些绕口，但却并不矛盾。没有哪一本类似厚度的书能够覆盖你需要了解的关于C++的一切知识，因为不同的程序员和应用需要不同的知识。因此，任何一本覆盖了C++的所有知识的书籍——例如Stroustrup所著的《The C++ Programming Language》（Addison-Wesley，2000）——都会无可避免地向你介绍许多你无需了解的东西。这是因为，即使你不需要它们，也自有其他人会有这个需要。

另一方面，C++的许多部分是十分重要的。因此，如果想取得高效率的话，我们就必须切实地掌握它们。我们会把注意力集中在这些重要的部分。而仅仅应用本书提供的信息来编写各种各样的实用程序是完全有可能的。事实上，本书的一个复审者（他是一个用C++编写的大型商业系统的主程序员）告诉我们，这本书基本上涵盖了他在工作中所要用到所有的工具。

拥有了这些工具，我们就可以编写真正的C++程序了——而不仅仅是编写具有C语言或其他任何语言风格的程序。掌握了这本书所介绍的知识之后，我们就能很轻松地继续往下学我们所希望学到的知识，而且我们也因此而获得了一个科学的学习方法。在业余的望远镜制造者群体中流传着一个这样的说法，就是先制造一个3英寸的镜片然后造出6英寸的镜片比从头开始制造一个6英寸的镜片更容易——对C++来说也是同样的道理。

在这本书中，我们仅仅覆盖了标准C++而忽略了其他专门的延伸。这种做法有

优势，也就是说，我们教你编写的程序可以在任何环境下运行。不过，这也表明了，我们不会探讨如何去编写运行在窗口环境下的程序，因为这样的程序将会无可避免地与特定的环境或与特定的厂商密切联系在一起。如果你想编写仅仅在特定的环境下运行的程序，那你就必须通过其他途径去学习具体的编写方法——但千万不要就这样把这本书合上了！这是因为，我们在这本书中介绍的方法是普遍适用的，以后你可以在任何的环境中使用在这里学到的任何知识。当然，如果你希望了解 GUI 的应用原理的话，那么你应该继续阅读一些关于这一方面的书籍——但是，请首先阅读这一本。

给熟练的 C 和 C++程序员的一点提醒

在学习一门新的程序设计语言的时候，我们可能会不自觉地用类似于我们已经了解的语言的风格去编写程序。我们的教学方法从一开始就使用了来自 C++标准库的高级抽象，这样就可以避免这种不自觉了。如果你已经是一个熟练的 C 或 C++程序员，那么你会从这个方法中得到一些好消息和一些坏消息——而实际上，好消息、坏消息都只不过是同样的消息罢了。

这些消息就是，在我们介绍 C++的时候，你可能会感觉到很惊奇——因为你所掌握的知识对你的学习来说好像用处并不大。一开始，你需要学习的知识将会比你意料中的多得多（这是坏消息），但你的学习效率将会比你预期的要高很多（这是好消息）。特别地，如果你已经对 C++有所了解，那么，之前你首先学习的很可能会是 C 编程，这就意味着你的 C++程序风格是建立在 C 的基础上的。这种做法本身并没有错，但是，我们的方法是如此的与众不同，以至于我们认为，我们会让你看到你从未见过的 C++的另一面。

当然，许多语法细节都是相似的，不过它们仅仅是细节。我们处理重要概念时所采用的顺序很可能会跟你之前接触过的完全不同。例如，直到第 10 章我们才会提到指针和数组，另外，我们甚至根本不会对你们可能非常熟悉的 printf 和 malloc 进行讨论。另一方面，我们会在第 1 章就开始讨论标准库的 string 类。我们所说的要采用一种全新的方法是名副其实的！

本书的结构

你有可能会发现，把本书分成两部分来考虑会更方便一点。第一部分是从开始到第 7 章，在这一部分，我们将把注意力集中于使用标准库抽象的程序；第二部分从第 8 章开始，我们会在这部分定义属于我们自己的抽象。

对库进行介绍首先就是一种不寻常的想法，但我们认为这种想法是正确的。C++语言的许多部分（尤其是那些较难的部分）在很大的程度上是为了库作者的利益而存在的，库的使用者根本无需了解语言的这些部分。因此，我们一直到本书的第二部分才提及这些部分，这样的话，很快我们就可以编写实用的 C++程序了。而如果使用的是一种更为传统的方法，那我们可能还要过很久才能开始编写程序。

一旦掌握了库的使用方法，你就可以开始学习那些低级工具并且可以尝试使用这些工具来编写你自己的库了，顺便提一下，库就是以这些工具为基础而建立起来的。另外，对于

一个库变得更加有用以及何时应该避免全部重写新的代码这两点，你也将会有一个感性的认识。

虽然这本书的厚度比许多 C++书籍要薄，但是，在书中我们尽量将每一个重要的概念使用至少两次以上，而关键概念的使用次数则更多。因此，我们在本书的许多部分中会提到其他的部分。在我们提及其他部分的时候，我们采用诸如§39.4.3 这样的形式，§39.4.3 指的是第 39.4.3 小节——或者说，如果这本书有这么多的小节的话，它至少会有此意义。在第一次对概念进行解释时，我们都会用粗体来标明这个概念，因为这样做可以方便读者的查找，而且这样还可以吸引读者的注意力，让读者把它当作一个重点来看待。

本书的每一章（除了最后一章）都会有一部分叫做“小结”的内容。这一部分内容是为两个目的服务的：它们可以让你加深对本章所介绍的概念的记忆；同时，它们也涵盖了额外的一些相关的知识——我们认为你迟早会需要了解这些知识。建议读者们在初次阅读时跳过这些内容，在日后有需要时再来参考它们。

本书的两个附录在细节的层次上概述并解释了语言和库的重要组成部分，我们希望它们会对你编写程序有所帮助。

让本书物尽其用

每一本关于程序设计的书都会包含有程序，这一本也不例外。为了理解程序是如何工作的，就必须在计算机上运行它们。这样的计算机到处都有，而且新的计算机也会不断出现——这句话的意思是说，到你读到这句话为止，我们提到的任何关于它们的信息都可能会是不准确的。因此，如果你还不知道怎样编译和运行 C++程序的话，那么请访问站点 http://www.acceleratedcpp.com 并参阅我们在那里发布的信息。我们会不断更新此站点的内容，为它添加关于 C++程序运行技巧的信息和建议。我们在这个网站中也提供了一些机器可读的示例程序版本和一些或许会让你感兴趣的其他信息。

致谢

我们谨对以下的人们表示我们的谢意，因为没有他们就不可能会有这本书的诞生。本书的成功在很大程度上要归功于我们的复审人员：Robert Berger, Dag Brück, Adam Buchsbaum,Stenphen Clamage, John Kalb, Jeffrey Oldham, David Slayton, Bjarne Stroustrup, Albert Tenbusch, Bruce Tetelman 和 Clovis Tondo。许多来自 Addison-Wesley 的工作人员参与了本书的出版工作；我们所知道的有：Tyrrell Albaugh, Bunny Ames, Mike Hendrickson, Deborah Lafferty, Cathy Ohala 以及 Simone Payment 等等。Alexander Tsiris 核对了§13.2.2 中的希腊词源。最后要说的是，开始高级编程的想法已经在我们脑海中萦绕了好几年了，这是受到数以百计听过我们课程的学生和成千上万参与我们讨论的人们的激励而产生的。

Andrew Koenig　　Gillette，New Jersey
Barbara E.Moo

目 录

前 言

第 0 章 开始学习 C++ ……1

0.1 注释 ……1

0.2 #include 指令 ……2

0.3 主函数 main ……2

0.4 花括号 ……2

0.5 使用标准库进行输出 ……3

0.6 返回语句 ……3

0.7 一些较为深入的观察 ……4

0.8 小结 ……5

第 1 章 使用字符串 ……8

1.1 输入 ……8

1.2 为姓名装框 ……10

1.3 小结 ……14

第 2 章 循环和计数 ……17

2.1 问题 ……17

2.2 程序的整体结构 ……18

2.3 输出数目未知的行 ……18

2.4 输出一行 ……22

2.5 完整的框架程序 ……27

2.6 计数 ……31

2.7 小结 ……32

第 3 章 使用批量数据 ……36

3.1 计算学生成绩 ……36

3.2 用中值代替平均值 ……42

3.3 小结 ……50

第 4 章 组织程序和数据 ……52

4.1 组织计算 ……

4.2 组织数据 ……63
4.3 把各部分代码连接到一起 ……68
4.4 把计算成绩的程序分块 ……71
4.5 修正后的计算成绩的程序 ……73
4.6 小结 ……75

第 5 章 使用顺序容器并分析字符串 ……78

5.1 按类别来区分学生 ……78
5.2 迭代器 ……82
5.3 用迭代器来代替索引 ……86
5.4 重新思考数据结构以实现更好的性能 ……87
5.5 list 类型 ……88
5.6 分割字符串 ……91
5.7 测试 split 函数 ……94
5.8 连接字符串 ……95
5.9 小结 ……100

第 6 章 使用库算法 ……105

6.1 分析字符串 ……105
6.2 对计算成绩的方案进行比较 ……114
6.3 对学生进行分类并回顾一下我们的问题 ……122
6.4 算法、容器以及迭代器 ……125
6.5 小结 ……126

第 7 章 使用关联容器 ……128

7.1 支持高效查找的容器 ……128
7.2 计算单词数 ……129
7.3 产生一个交叉引用表 ……131
7.4 生成句子 ……135
7.5 关于性能的一点说明 ……142
7.6 小结 ……143

第 8 章 编写泛型函数 ……146

8.1 泛型函数是什么？ ……146
8.2 数据结构独立性 ……150
8.3 输入输出迭代器 ……158
8.4 用迭代器来提高适应性 ……159
8.5 小结 ……

第 9 章　定义新类型 ……163
9.1　回顾一下 Student_info ……163
9.2　自定义类型 ……164
9.3　保护 ……167
9.4　Student_info 类 ……171
9.5　构造函数 ……172
9.6　使用 Student_info 类 ……175
9.7　小结 ……176

第 10 章　管理内存和低级数据结构 ……178
10.1　指针与数组 ……178
10.2　再看字符串常量 ……185
10.3　初始化字符串指针数组 ……186
10.4　main 函数的参数 ……188
10.5　文件读写 ……189
10.6　三种内存分配方法 ……192
10.7　小结 ……195

第 11 章　定义抽象数据类型 ……197
11.1　Vec 类 ……197
11.2　实现 Vec 类 ……198
11.3　复制控制 ……205
11.4　动态的 Vec 类型对象 ……213
11.5　灵活的内存管理 ……214
11.6　小结 ……220

第 12 章　使类对象像一个数值一样工作 ……222
12.1　一个简单的 string 类 ……222
12.2　自动转换 ……224
12.3　Str 操作 ……225
12.4　有些转换是危险的 ……232
12.5　类型转换操作函数 ……233
12.6　类型转换与内存管理 ……235
12.7　小结 ……237

第 13 章　使用继承与动态绑定 ……239
13.1　一个简单的 string 类 ……239
13.2　多态和虚拟函数 ……

13.3 用继承来解决我们的问题……249
13.4 一个简单的句柄类……255
13.5 使用句柄类……260
13.6 微妙之处……262
13.7 小结……263

第 14 章 近乎自动地管理内存……267

14.1 用来复制对象的句柄……268
14.2 引用计数句柄……274
14.3 可以让你决定什么时候共享数据的句柄……277
14.4 可控句柄的一个改进……279
14.5 小结……283

第 15 章 再探字符图形……284

15.1 设计……284
15.2 实现……293
15.3 小结……303

第 16 章 今后如何学习 C++……306

16.1 好好地利用你已经掌握的知识……306
16.2 学习更多的东西……308

附录 A C++语法细节……310

A.1 声明……310
A.2 类型……315
A.3 表达式……321
A.4 语句……324

附录 B 标准库一览……327

B.1 输入-输出……327
B.2 容器和迭代器……329
B.3 算法……337

第 0 章

开始学习 C++

让我们从一个小的 C++程序开始我们的学习：

```
//一个小的C++程序
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

程序员经常把这样的一个小程序称为 Hello, world!程序。尽管这个程序很小，但是，在往下阅读之前你还是应该抽点时间出来在你的计算机上编译和运行一下它。这个程序会在标准输出上显示 Hello, world!。一个典型的标准输出是显示在屏幕上的一个窗口。如果碰到麻烦的话，可以找那些已经了解 C++的人来寻求帮助，也可以咨询我们的站点 http://www.accelerated cpp.com。

由于它的简短，这个程序还是很有用的。如果你对如此简单的一个程序都有问题的话，那么最可能的原因就是本书中存在着明显的印刷错误或者你还没有掌握编译器实现的使用方法。此外，透彻地了解哪怕是一个如此简单的程序也会教给你许多出乎你意料之外的 C++基础原理。为了让我们的理解更加深入，让我们逐行地来分析这个程序吧。

0.1 注释

程序的第一行是：

```
//一个小的C++程序
```

字符//表示一段**注释**的开始，以这种方式开始的注释将会一直延伸至该行结束。编译器在编译时会忽略掉注释：它们的用途是为阅读该程序的人解释程序。

0.2 #include 指令

C++中的许多基本工具，例如输入/输出，都不属于**语言核心**，而是**标准库**的一部分。这个差别是很重要的——因为语言核心对所有的 C++程序来说都是可用的，但是，在使用标准库时，我们必须明确指定我们所希望使用的那部分标准库。

#include 指令就是这样的一种工具。它一般出现在程序的开头。在我们的程序中，所用到的标准库部分仅仅是输入/输出。我们通过下面的语句来请求使用它：

```
#include <iostream>
```

名称 iostream 表示对顺序或流输入/输出的支持，不过它不支持随机存储或图形输入/输出。由于 iostream 出现在#include 指令中，而且被尖括号（<和>）括起，所以它就代表 C++库的**标准头文件**。

C++标准文档并没有告诉我们**标准头文件**是什么，但是它明确地定义了每一个头文件的名称和行为。在包含了**标准头文件**之后，程序就可以使用相关的库所提供的功能了——我们并不需要去关心它的实现，这是编译器实现所需考虑的事情。

0.3 主函数 main

函数（function）是一段具有名称的程序，程序的其他部分可以**调用**函数或使函数运行。每个 C++程序都必须包含一个名为 main 的函数。当我们请求 C++实现运行程序时，它就会调用这个函数来响应我们的请求。

main 函数要返回一个整数类型的值作为其结果，这样它就可以告知编译器是否运行成功。零值表示成功；任何其他的值都意味着程序运行有问题。因此，我们用

```
int main()
```

来表示我们定义了一个名为 main、返回值类型为 int 的函数。在这里，int 是核心语言用来描述整数的名称。main 后面的括号括住了函数从编译器中接收到的参数。在这个特殊的例子中，main 函数并没有参数，因此括号内没有任何东西。在§10.4 中我们将会看到 main 函数参数的使用方法。

0.4 花括号

紧接着上面所说的括号，让我们继续分析 main 函数的定义。在 main 函数中，括号的后面是一连串用花括号括住的语句：

```
int main()
{                    //左花括号
                     //语句放此处
```

```
}                    //右花括号
```

在 C++中，花括号告诉编译器把出现在它们之间的所有内容当作一个单元来处理。在本例中，左花括号标明了我们的主函数语句的开始，右花括号则标明了它的结束。换句话说，花括号表明，在它们之间的所有语句都隶属于同一个函数。

如我们在这个函数中见到的那样，如果在花括号中有两条或更多的语句，那么编译器就会按照这些语句出现的先后顺序来执行它们。

0.5 使用标准库进行输出

花括号内的第一条语句执行了程序的实际工作。

```
std::cout << "Hello, World!" << std::endl;
```

这条语句首先使用了标准库的输出运算符<<来把 Hello, world!写到标准输出中，然后它写入了 std:endl 的值。

名称之前的 std::表明了这个名称是一个名为 std 的名字空间的一部分。名字空间是一个相关名称的集合；标准库使用 std 来包含所有由它定义的名称。例如，标准头文件 iostream 定义了名称 cout 和 endl，这些名称实际上也就是我们所使用的 std::cout 和 std::endl。

名称 std::cout 指**标准输出流**，标准输出流是所有 C++实现用来进行普通的程序输出的工具。在窗口操作系统环境下的一个典型 C++实现中，在程序运行的时候，系统环境会把程序和一个窗口联系起来，而 std::cout 就指示了这个窗口。在这样的系统环境下，写到 std::cout 中的输出将会出现在相应的窗口中。

我们用值 std::endl 来作为当前输出语句行的结束，如果程序需要产生更多的输出，接下来的输出就会在新的一行中出现。

0.6 返回语句

返回（return）语句，例如

```
return 0;
```

会在其出现的位置终止函数的执行，并把一个值返回给调用这个函数的程序，返回值出现在 return 和分号之间（本例中为 0）。返回值的类型必须和函数声明的返回类型一致。就 main 函数而言，返回类型是 int，接收 main 的返回值的程序是 C++实现本身。因此，main 函数中的 return 语句必须包含一个整数值的表达式，这个表达式会把返回值传递给实现。

当然，我们有可能会在多个位置上合理地终止程序，这样的程序可能会有多条的 return 语句。如果函数定义保证了函数会返回一个特定类型的值，那么函数中的每条 return 语句都必须返回一个适当类型的值。

0.7 一些较为深入的观察

这个程序使用了两个在C++中到处可见的概念：表达式和作用域。随着本书内容的进展，我们会对这些概念进行更为深入的讨论，但是，在这里我们有必要对某些基础知识进行初步的了解。

表达式会让编译器对某些事物进行运算。运算会产生一个**结果**，同时还有可能会具有**副作用**——也就是说，副作用不是结果的直接部分，但它会影响程序或系统环境的状态。例如，3+4是一个表达式，它产生的结果是7，这个运算并没有副作用。而

```
std:cout<<"Hello, world!"<<std:endl
```

就是一个具有副作用的表达式，它会把Hello,world!写进标准输出流而且会结束当前行。

表达式包含有操作数和运算符，操作数和运算符都能以多种形式出现。在我们的Hello,world!表达式中，两个<<符号是运算符，std::cout、"Hello,world!"和std::endl则是操作数。

每一个操作数都具有一个**类型**。以后我们会进一步讨论类型，但是，从本质上来看，类型表示的是一种数据结构和对此数据结构的合理操作。运算符的效果取决于它的操作数的类型。

类型通常会有名称。例如，核心语言定义了int来作为一种类型的名称，这种类型代表的是整数。库定义了std::ostream来作为提供了基于流的输出的类型。在我们的程序中，std::cout的类型就是std::ostream。

运算符<<具有两个操作数，然而我们却使用了两个<<运算符和三个操作数。为什么会这样呢？答案是：<<是**左结合**的。不严格地说就是，如果<<在同一个表达式中出现了至少两次的话，那么每一个<<都可以为其左操作数使用尽可能多的表达式，而对于它的右操作数则要使用尽可能少的表达式。在我们的例子中，第一个<<运算符的右操作数是"Hello,world"，左操作数是std::cout；第二个<<运算符的右操作数是std::ednl，左操作数是std::cout<<"Hello,world!"。如果用括号来表明运算符和操作数之间的关系，我们就会看到，该输出表达式等价于

```
(std::cout << "Hello,world!") << std::endl
```

每个<<的行为都取决于它的操作数类型。第一个<<的左操作数是std::cout，而std::cout的类型是std::ostream。它的右操作数是一个字符串直接量，我们将在§10.2中开始讨论字符串直接量的类型。根据这些操作数类型，<<会把它的右操作数的字符写到左操作数所指示的流中，它的结果就是它的左操作数。

因此，第二个<<的左操作数就是一个表达式，它会产生类型为std::ostream的std::cout；它的右操作数则是std::endl，std::endl是一个**控制器**（manipulator）。控制器的关键性质是：如果把一个控制器写到流中，那么我们就可以控制这个流了。而为此单单将字符写到流中是不够的，我们还要通过其他的途径来实现对流的控制。如果<<的左操作数的类型是std::ostream，而右操作数是一个控制器，那么<<就会对特定的流执行控制器所指定的动作，同时它会返回流作为其结果。就std::endl而言，我们所做的动作是结束当前的输出行。

因此，整个表达式所产生的值是 std::cout，此外，作为其副作用，它还会把 Hello,world! 写到标准输出流并结束输出行。当我们在表达式后面紧接了一个分号时，就表明我们让系统环境丢掉了这个返回值——这是合理的，因为我们仅仅对副作用感兴趣。

名字的**作用域**是程序的一部分，只有在这一部分中这个名字是有意义的。C++有几种不同的作用域，在这个程序中，我们已经见到了其中的两种。

我们用到的第一种作用域是名字空间，正如我们刚才看到的那样，名字空间是一个相关名称的集合。标准库在一个名为 std 的名字空间中定义了它所提供的所有名称，这样它就可以避免跟我们自己定义的名称发生冲突了——前提是我们还没有愚蠢到想要定义 std 这个名称。在我们使用一个标准库所提供的名称时，我们必须指明所需要的那个名称是来自库的；例如，std::cout 表示 cout 是在名为 std 的名字空间中定义的。

名称 std::cout 是一个**限定名称**，它使用了::运算符。我们把::运算符称为**作用域运算符**。::的左边是一个（很可能是限定的）作用域的名称，就 std::cout 而言，这个名称是一个名为 std 的名字空间。::的右边也是一个名称，这个名称是在左边命名的作用域中定义的。因此，std::cout 表示“在（名字空间）作用域 std 中的名称 cout”。

花括号是另一种作用域。main 的函数体（以及每个函数的函数体）本身就是一个作用域。在这样的一个简单的程序中，我们可能不会对这个事实有太大的兴趣，但它几乎会跟每一个我们所编写的函数都有关联。

0.8 小结

尽管我们编写的程序很简单，但是在这一章中，我们还是涉及了很多的问题。在这里，我们想进一步讨论一下我们所介绍的知识，因此，在继续往下阅读之前，请你先透彻地理解本章的内容。

为了帮你达到这个目的，在这一章（以及在除了第 16 章以外的每一章中）我们都会以一个名为“小结”的章节和一系列的习题来结束我们的讨论。小结概述并偶尔扩展了正文中的信息。读者们有必要去看一下各章的小结，因为它们可以让你们加深对我们在各章中所介绍的概念的理解。

程序结构：C++程序通常具有**自由风格**，也就是说，只是在防止相邻的符号混在一起的时候，我们才必须使用空白符。另外，新行（也就是系统环境从程序的一行转换到下一行时所用的方式）也是另一种形式的空白，此外它并没有其他的特别含义。程序中空白符出现位置的不同可能会提高（或降低）程序的可读性。一般来说，我们应该提高程序的可读性。

以下的三个实体不能具有自由风格：

字符串直接量	用双引号括住的字符；不可以跨行
#include 名称	必须在单独的一行中出现（注释除外）
//注释	//后面可以跟着任何东西；结束于当前行的行尾

以/*开始的注释具有自由风格：它结束于第一个相邻的*/并且可以跨越多行。

类型定义了数据结构以及对这些数据结构的操作。C++有两种形式的类型：核心语言提供的内建类型，如 int；定义在核心语言之外的类型，如 std::cout。

名字空间是一种把相关名称聚集在一起的技术。来自标准库的名称是在名为 std 的名字空间中定义的。

字符串直接量从双引号（"）开始而且结束于双引号；每个字符串直接量都必须全部出现在程序中的一行中。如果字符串直接量中的字符是跟在反斜杠（\）后面的，那么它们会具有特殊的含义：

\n	换行符
\t	水平制表符
\b	回退符
\"	我们把这个符号当作字符串的一部分而不是把它当作字符串的结束符
\'	与字符串直接量中的'具有一样的意义，用来保持字符直接量的一致性（§1.2）
\\	若字符串中包括一个\，那么就可以把接下来的字符当作普通字符来看待

在§10.2和§A.2.1.3中我们会对字符串直接量做进一步的讨论。

定义和头文件：C++程序使用的每一个名称都必须具有一个相应的定义。标准库在头文件中定义它的名称，程序可以通过#include 指令来访问这些名称。名称必须在使用前定义；因此，我们必须在使用头文件中的任何名称之前编写#include 指令。<iostream>头定义了库中的输入/输出工具。

主函数（main）：所有的 C++程序都必须定义且只能定义一个名为 main 的函数，这个函数返回一个 int 类型的值。系统环境通过调用 main 函数来运行程序。main 函数返回值为 0 意味着运行成功；返回值不是 0 则意味着失败。一般来说，函数必须包含至少一条的 return 语句，而且在函数的最后一定要有 return 语句。但 main 比较特殊：它可以没有返回语句；如果是这样的话，编译器就会假设它返回 0。尽管如此，我们还是应该养成在 main 函数中明确声明一条返回语句的好习惯。

花括号和分号：在 C++程序中，这些并不引人注目的符号是很重要的。我们经常会因为它们的微不足道而忽略了它们。它们之所以如此重要是因为，假如我们遗忘了它们中的某一个，那么编译器就很有可能会发出难于理解的诊断信息。

出现在花括号中的零条或多条语句也是一条语句，它要求编译器实现按它们出现的先后次序来顺序执行这些语句。就算函数的函数体只含有一条语句，我们也必须用花括号来括住它。在一对匹配的花括号之间的的语句构成了一个作用域。

如果一个表达式的后面跟着一个分号，那这个表达式也是一条语句。它的执行只是为了获得副作用，实际产生的结果将会被忽略。表达式是可选的：忽略了表达式就会产生一条空语句，这条语句是没有任何作用的。

输出：对 std::cout<<e 进行计算会把 e 的值写到标准输出流中，同时这个计算会产

为 ostream 的 std::cout 来作为它的结果值，这样我们就可以进行链式的输出操作了。

习题

0-0 编译并运行 Hello, world!程序。

0-1 下面的表达式是做什么的？

```
3 + 4;
```

0-2 编写一个程序，使它在运行时输出：

```
This (") is a quote , and this (\) is a backlash.
```

0-3 字符串直接量"\t"代表一个水平制表符；不同的 C++实现以不同的形式显示制表符。在你的实现中试验一下，看它是怎样处理制表符的。

0-4 编写一个程序，运行时以 Hello,world!程序作为这个程序的输出。

0-5 下面的程序是一个有效的程序吗？说出理由。

```
#include <iostream>
int main()  std::cout << "Hello, world!" << std::endl;
```

没有花括号没有返回值

0-6 下面的程序是一个有效的程序吗？说出理由。

```
#include <iostream>
int main()  {{{{{{ std::cout << "Hello, world!" << std::endl; }}}}}}
```

没返回值
花括号成这样也OK其实

0-7 那下面的这个程序呢？

```
#include <iostream>
int main()
{
    /*这是一个注释，因为我们使用了/*和*/来作为它的定界符，
     所以它占据了几行的范围*/
    std::cout << "Does this work?" << std::endl;
    return 0;
}
```

注释的符号识别不出最后一个*/

0-8 ……这个呢？

```
#include <iostream>
int main()
{
    //这是一个注释，它占据了几行的范围
    //在这里，我们使用了//而不是/*
    //和*/来为注释定界
    std::cout << "Does this work?" << std::endl;
    return 0;
}
```

木有错误，但会有warning

0-9 最短的有效程序是什么？ 上来就return 0?

0-10 重写 Hello,world!程序，让程序中每一个允许出现空白符的地方都换行。

狂用std::endl吧

第 1 章

使用字符串

在第 0 章中我们深入地剖析了一个很小的程序，通过这个程序，我们介绍了 C++的许多基本概念：注释、标准头文件、作用域、名字空间、表达式、语句、字符串直接量以及输出。在这一章中，我们会继续对基本概念的概述，将使用字符串来编写类似的简单程序。在此过程中，我们将会学习声明、变量和初始化，而且我们还会了解到关于输入和 C++的 string 库的知识。本章中的程序是非常简单的，它们甚至不需要用到我们将在第 2 章提及的任何控制结构。

1.1 输入

一旦我们可以输出文字，那么逻辑上的下一步就是读入。例如，我们可以修改 Hello,world! 程序来对某一个特定的人说“hello”：

```
//请求某人输入其姓名，然后向这个人发出问候
#include <iostream>
#include <string>

int main()
{
//请某人输入其姓名
    std::cout << "Please enter your first name: ";

//读姓名
    std::string name;      // 定义变量 name
    std::cin >> name;      // 把姓名读进 name

//输出对这个人的问候语
    std::cout << "Hello, " << name  << "!" << std::endl;
    return 0;
}
```

在我们运行程序的时候，它会在标准输出上显示：

```
Please enter your first neme
```

如果我们作出响应，例如，我们输入了

```
Valadimir
```

程序就会输出：

```
Hello, Valadimir!
```

让我们看看发生了什么吧。为了读入输入，我们必须要找个地方来存放它。而这样的一个地方就叫作**变量**。变量是一个具有名称的**对象**，对象则是计算机中的一段具有类型的内存空间。对象和变量两者之间的区别是很重要的，因为，正如我们将在§3.2.2和§10.6.1中看到的那样，有些对象可能会没有名称。

如果我们想要使用一个变量，我们就要告诉系统环境应该给它一个什么样的名称，以及我们希望它具有什么样的类型。同时提供名称和类型的请求会让系统环境更加方便地为我们的程序产生高效的机器代码。这个请求同时也会让编译器检测到误拼的变量名称——除非误拼的变量名称恰好与某一个我们想要在程序中使用的名称相匹配。

在这个例子中，我们的变量被命名为 name，它的类型是 std::string。正如我们在§0.5和§0.7中看到的那样，我们使用 std::来表明，紧跟着它的名称 string 是标准库的一部分，而不是核心语言或非标准库的一部分。跟标准库的每一部分一样，std::sting 有一个相关的名叫<string>的标准头文件，因此在程序中我们增加了一条相应的#include 指令。

到现在为止，第一条语句

```
std::cout <<"Please enter your first name: ";
```

对我们来说应该是很熟悉的了：它输出一个提示来请求用户输入姓名。在这条语句中，很重要的一点是：它并没有那个叫做 std::endl 的控制器。因为我们没有使用 std::endl，所以程序在输出完提示信息之后并不会开始新的一行。相反，一旦它输出了提示后，计算机就会等待在同一行中的输入。

下一条语句

```
std::string name;        //定义变量 name
```

是一个**定义**，它定义了我们的名为 name 的变量，它的类型是 std::string。因为这个定义出现在一个函数体内，所以变量 name 是一个**局部变量**，也就是说，它只存活在花括号括起的程序运行期内。一旦程序运行到达了"}"，变量 name 就会被销毁，同时这个变量所占用的全部内存也会被系统回收用于其他用途。局部变量有限的生存期是区分变量和对象的一个重要依据。

隐含在对象类型中的还有其**接口**——对于某种类型的对象来说，接口就是可实现的操作的集合。如果我们定义了一个名为 name 的 string 变量（具名对象），那就表示，我们能够对变量 name 做库允许我们对 string 所做的所有操作。

这些操作中的一个是**初始化** string。我们在定义一个 string 变量的时候就已经对它进行了隐式的初始化，这是因为，标准库要求每一个 string 对象都要有一个初始化值。我们即将

在创建一个字符串的时候，我们可以给它提供一个明确的值。如果我们没有这样做的话，那么，在一开始的时候字符串根本不会包含任何字符。我们把这样的字符串称为空字符串。

定义了 name 变量之后，我们就可以执行以下的语句了：

```
std::cin >> name;      //把姓名读进 name
```

这条语句把 std::cin 中输入的值读入到变量 name 中。跟输出语句中的<<运算符和 std::cout 的使用相似，库使用了>>运算符和 std::cin 来进行输入。在这个例子中，>>运算符从标准输入读进一个字符串，然后把读到的值存储在名为 name 的对象中。在我们请求库来读一个字符串的时候，它会首先略去输入开始时碰到的**空白字符**（空白、制表键、回退键或换行符），然后它连续地把字符读到 name 变量中，直到它遇到了另一个空白字符或文件结束标记为止。因此，执行 std::cin>>name 表达式的结果是从标准输入读进一个词，然后把构成这个词的字符存储在 name 变量中。

输入操作有另一个副作用：它产生一个出现在计算机输出装置上的提示，这个提示会请求用户输入其姓名。一般来说，输入/输出库会把它的输出保存在一个叫做**缓冲区（buffer）**的内部数据结构中，缓冲区是用来优化输出操作的。大多数的系统都花费了大量的时间来把字符写到输出装置中，而不去关心待写的字符的个数。为了减少对应于每一个输出请求的写操作的系统开销，库使用了缓冲区来累积待写的字符，而且只是在有必要的时候，它才会把缓冲区的内容写到输出装置中从而**刷新**缓冲区。这样，它就能把几个输出操作合并到一个单独的写操作中了。

有三种事件会导致系统刷新缓冲区。第一种是，缓冲区已经满了，这样的话，库会自动刷新它。第二种是，请求库从标准输入流中读数据。这样，库不用等到缓冲区满就可以马上刷新输出缓冲区了。第三种情况是，如果我们明确要求刷新，那么系统就会刷新缓冲区。

如果我们的程序把它的提示写到 cout 中，那么输出就会进入到与标准输出流相关联的缓冲区中。接下来，我们会尝试从 cin 中读数据。这个读操作会刷新 cout 缓冲区，因此我们就能确保我们的用户看到那个提示。

接下来的语句产生了输出，这条语句明确地命令库刷新缓冲区。跟输出提示的那一条语句相比，这一条只是稍为复杂了一点。在这里，我们输出了字符串直接量"Hello, "，在字符串直接量后接着输出了字符串变量 name 的值。这条语句的最后是 std::endl，对 std::endl 的值的写操作会结束输出行，然后它会刷新缓冲区，这使得系统会立刻向输出流写入数据。

在我们编写那些可能会花费大量时间来运行的程序的时候，我们应该养成在适当的时刻刷新缓冲区的好习惯。否则，有些程序的输出可能会长时间滞留在缓冲区中，也就是说，我们有可能要等待很久才可以看到我们的程序所写的输出。

1.2 为姓名装框

到目前为止，我们的程序的灵活性仍受其问候语句的限制。我们不妨用更生动的

修改它，这样的话，输入和输出的形式如下：

```
Please enter your first name: Estragon
********************
*                  *
* Hello, Estragon! *
*                  *
********************
```

我们的程序会产生五行的输出。第一行是框架的开始，它由一系列的*字符组成，其长度为姓名、问候语("Hello, ")以及两端的空格和*所占字符的总长。接下来的一行由相应数目的空格以及开头和结尾的两个*字符组成。第三行依次有一个*字符、一个空格、信息、一个空格和最后的一个*字符。最后的两行则分别与第二行和第一行完全相同。

对此的一个明智的实现策略是每次一行地逐步建立输出。首先，我们要读入姓名，然后会用读到的姓名来构建问候语句，接下来利用问候语句来建立起输出的每一行。下面的程序使用了这个策略来解决我们的问题：

```
//请求用户输入姓名，然后产生一个带框架的问候语句
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;

    //构造我们将要输出的信息
    const std::string greeting = "Hello, " + name + "!";

    //构建输出的第二和第四行
    const std::string spaces(greeting.size(), ' ');
    const std::string second = "* " + spaces + " *";

    //构建输出的第一和第五行
    const std::string first(second.size(), '*');

    //输出所有的内容
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;
```

```
    return 0;
}
```

首先，我们的程序会请求用户输入其姓名，并把姓名读到名为 name 的变量中。然后，它会定义一个名为 greeting 的变量，这个变量包含有它要输出的信息。接下来，它定义了一个名为 space 的变量，这个变量包含有跟 greeting 变量中的字符个数一样多的空格。程序使用 space 变量来定义另一个变量 second，second 包含了第二行输出的内容。然后程序创建了一个 first 变量，这个变量包含了跟 second 变量中的字符数目一样多的*字符。最后，程序每次一行地进行输出。

我们应该不会对#include 指令和程序中的头三条语句感到陌生。另一方面，在 gteeting 变量的定义中包含有三个新的概念。

第一个概念是，在我们定义一个变量的时候，可以赋一个值给它。为此，我们要在变量名称和跟在名称后面的分号之间放置一个符号“=”，其后是我们希望变量所具有的值。如果变量和值具有不同的类型——就像我们将在 § 10.2 中看到的那样，字符串和字符串直接量具有不同的类型——系统环境就会把初始值的类型**转换**成变量的类型。

第二个概念是，我们可以用+来把一个字符串和一个字符串直接量**连接**起来——或者，就此而言，我们可以连接两个字符串（但不能连接两个字符串直接量）。在第 0 章中我们注意到，3+4 等于 7。但在本例中的+号则具有完全不同的含义。在不同的情况下，可以通过检查运算符+的操作数类型来判定它是做什么的。如果一个运算符对于不同类型的操作数来说具有不同的含义，那么我们就说此运算符被**重载**了。

最后的一个概念是，const（用来表示**常量**的关键字）可以作为变量定义的一部分。这样做可以保证，在变量的生存期内我们不会改变它的值。严格来说，在使用 const 的时候，程序不会从中获得任何东西。然而，如果我们指出了哪些变量是不会改变的，那我们就能让程序更容易理解。

值得注意的是，如果我们说一个变量是常量，那我们必须在定义它的时候对它进行初始化，否则，以后我们就再也不会有这个机会了。我们还要注意，用来对常量变量进行初始化的数值本身不必也是一个常量。在这个例子中，只有我们把一个值读进 name 变量中了，我们才可以知道 greeting 变量的值是什么，很明显，这个读操作在我们运行程序之前是不会发生的。我们会把数据读到 name 中从而改变它的值，因此我们不能说 name 是常量。

运算符的一个永远不会改变的性质是它的结合律。我们在第 0 章了解到，<<是左结合的，因此，std::cout<<s<<t 和(std::cout<<s)<<t 是等价的。同样地，运算符+（以及运算符>>）也是左结合的。这样的话，把"Hello,"和 name 连接起来，然后再把所得的结果和"!"连接起来，最后得到的结果就是"Hello, "+name+"!"的值。例如，如果变量 name 的值是 Estragon 的话，那么"Hello, "+name+"!"的值就是 Hell,Estragon!。

到这里为止，我们已经构造出我们的问候语了，而且我们在 greeting 变量中存放了这个信息。我们的下一步的工作是构建一个框架，这个框架会把我们的问候语括起来。为此，

下面的这条语句中使用了另外的三个概念：

```
std::string spaces(greeting.size(),' ');
```

在我们定义 greeting 的时候，我们使用了一个符号“=”来对它进行初始化。而此处 spaces 的后面有两个表达式，我们用逗号把这两个表达式分隔开来并用括号括起它们。如果我们使用了=，我们就可以明确地给这个变量指定一个值。正如我们对 greeting 所做的一样，如果在定义中使用了括号，那我们就可以让系统环境**构造**一个来自表达式的变量——在这个例子中，这个变量是 spaces，而变量的构造方式则取决于变量的类型。换句话来说，为了更好地理解这个定义，我们必须先理解，根据两个表达式来构造一个字符串的操作究竟有什么意义。

一个变量的构造方式完全取决于它的类型。在这个特殊的例子中，我们构造了一个字符串类型的变量，此变量来自于——对了，是来自什么呢？两个表达式的形式都是我们之前没有见过的。它们到底是什么含义呢？

第一个表达式 greeting.size()是一个调用**成员函数**的例子。事实上，名为 greeting 的对象有一个被称为 size 的成员。这个成员实际上是一个函数，因此我们可以调用它来获得一个值。变量 greeting 的类型是 std::string，定义了这种类型之后，对 greeting.size()的求值就会产生一个整数，这个整数代表了 greeting 所包含的字符个数。

第二个表达式' '是一个**字符直接量**。字符直接量和字符串直接量完全不同。字符直接量总是用单引号括起的；而字符串直接量却总是用双引号括起的。字符直接量的类型是内建类型 char；字符串直接量的类型则复杂得多，我们将在§10.2 中对它进行解释。一个字符直接量代表着一个字符。在字符串直接量中具有特殊意义的字符在字符直接量中也会具有同样特殊的意义。因此，如果我们想用'或\符号，就要在它之前加上\。就此而言，'\n'、'\t'、'\''以及其他相关形式的性质是跟我们在第 0 章看到的它们用于字符串时表现出来的性质是类似的。

为了透彻地理解 spaces，我们需要知道：如果我们根据一个整数值和一个字符值来构造一个字符串，那么在所得到的结果中就会有这个字符值的多份复制，而复制的份数跟整数的值相等。例如，如果我们定义了：

```
std::string stars(10 , '*');
```

那么，stars.size()的结果是 10，而 stars 本身就会包括 10 个 * 字符，即**********。

因此，spaces 的字符个数跟 greeting 的相同，不过，所有的这些字符都是空白。

我们并不需要新的知识就可以理解 second 的定义：我们把" * "、空白字符串和" * "连接到一起就得到了我们的框架信息的第二行。理解变量 first 的定义时我们也不需要新的知识：它会赋给 first 一个包含*字符的值，而且，*字符的个数跟 second 变量的相等。

对于程序的其余部分，我们应该不会再感到陌生了：它所做的只不过是（就像我们在§1.1 中所做的那样）输出字符串。

1.3 小结

类型：

char	内建类型，用来保存由系统环境定义的普通字符。
wchar_t	内建类型，用来保存“宽字符”。宽字符是一种足够大的数据类型，它可以为诸如日语这样的语言保存字符。

string 类型定义于标准头文件<string>中。一个 string 类型的对象包含了一连串的零个或多个字符。假如 n 是一个整数，c 的类型是 char，is 是一个输入流，而 os 是一个输出流，那么对 string 类型的数据的操作包括：

std::tring s;	把 s 定义为类型为 std::string、初始为空的变量。
std::string t=s;	把 t 定义为类型为 std::string 的变量，它的初始值包含 s 的字符的一个复制，在这里，s 可以是一个字符串或者是一个字符串直接量。
std::string z(n, c);	定义类型为 std::string 的变量 z，且把 z 初始化为包含 n 个字符 c 的字符串。这里的 c 必须是一个字符，它既不能是字符串也不能是字符串直接量。
os<<s	不改变格式而把 s 所包含的字符写到由 os 指示的输出流中。这个表达式的结果是 os。
is>>s	从 is 所指示的流中读字符，把出现在第一个非空白字符之前的所有空白字符都丢掉。然后，连续地从 is 把字符读到 s 中，用读到的值把任何可能已经存在于 s 中的值都覆盖掉，直到读进的下一个字符是空白字符此过程才结束。这个表达式的结果是 is。
s+t	这个表达式的结果是一个 std::string 类型的值，它包含 s 中全部的字符以及后面紧接着的 t 中的全部字符。其中，s 或者 t 中的任一个都可以是（但不能两个都是）字符串直接量或 char 类型的值。
s.size()	结果表示 s 中的字符个数。

变量的定义形式可以是以下三种形式中的一种：

```
std::string hello="Hello";      //用明确的初始化值来定义变量
std::string stars(100, '*')     //根据类型和给定的表达式来构造一个变量
std::string name;               //定义一个变量，定义时不明确指定它的初始化值，
                                //因此，这个变量的初始化值取决于它的类型
```

在一对花括号内定义的变量是局部变量，它仅仅是在花括号内的那部分程序被执行的时候才存在。当系统环境执行到}的时候，它就会销毁这个变量，然后系统会回收变量所占用的所有内存。

把一个变量定义成常量可以确保这个变量的值在它的生存期内不被改变。我们必须

的时候对这样的一个变量进行初始化，否则在以后就没有机会这样做了。

输入：执行 std::cin>>v 表达式会忽略任何在标准输入流中的空白字符，然后它从标准输入把数据读到变量 v 中。这个表达式会返回类型为 istream 的 std::cin，这样，我们就可以进行链式的输入操作了。

习题

1-0 编译、运行并测试本章中的程序。

1-1 以下的定义有效吗？理由是什么？

```
const std::string hello = "Hello";
const std::string message = hello + ", world" + "!";
```

无效，两个常量字符串无法相加

1-2 以下的定义有效吗？理由是什么？

```
const std::string exclam = "!";
const std::string message = "Hello" + ", world" + exclam;
```

不行，主要那两个明明可以写在一个分号里

1-3 以下的程序有效吗？如果有效的话，它是做什么的？如果无效，为什么呢？

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

int main()
{
    { const std::string s = "a string";
      std::cout << s << std::endl; }

    { const std::string s = "another string";
      std::cout << s << std::endl; }
    return 0;
}
```

有效，就是在两个花括号定义了同一个名字的s的字符串

1-4 下面的这个程序又怎样呢？如果我们把倒数第三行的}}改成};}的话，会出现什么情况呢？

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

int main()
{
    { const std::string s = "a string";
      std::cout << s << std::endl;
    { const std::string s = "another string";
```

```
    //  std::cout << s << std::endl; }}
      std::cout << s << std::endl; };}
    return 0;
}
```

1-5 下面这个程序呢？如果有效，它是做什么的？如果无效，说出理由，然后把它改写成有效的程序。

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

int main()
{
    { std::string s = "a string";
    { std::string x = s + ", really";
      std::cout << s << std::endl; }
      std::cout << x << std::endl;  // x is out of its scope here
    }
    return 0;
}
```

1-6 在下面的程序向你发出输入请求的时候，如果你输入了两个名字（例如，Samuel Beckett），它会怎么样处理呢？在运行程序之前先预测一下结果，然后上机试一下。

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

int main()
{
    std::cout << "What is your name? ";
    std::string name;
    std::cin >> name;
    std::cout << "Hello, " << name
              << std::endl << "And what is yours? ";
    std::cin >> name;
    std::cout << "Hello, " << name
              << "; nice to meet you too!" << std::endl;
    return 0;
}
```

输入两个名字不会有错误
它会直接分开识别

第 2 章

循环和计数

在 § 1.2 中，我们改进了那个输出问候语的程序，改写后的程序输出了一个格式化的框架，这个框架则框住了原程序中的问候语。在本章中，我们将进一步增加这个程序的灵活性，以使我们无需重新编写程序就能直接改变框架的长度。

沿着这个思路，我们将开始学习 C++中的运算，并了解一下 C++是如何支持循环和条件分支的，然后，我们会讨论一下循环不变式的相关概念。

2.1 问题

§1.2 中的程序输出了一条周围有框架框住的问候语。例如，如果我们的用户输入一个姓名 Estragon，那我们的程序就会输出：

```
********************
*                  *
* Hello, Estragon! *
*                  *
********************
```

这个程序每次一行地构造输出。它定义了变量 first 和 second，这两个变量分别包含了输出的第一和第二行；此外，它还输出了第三行的问候语句并用一些字符把这个问候语围了起来。没有必要为输出的第四行和第五行定义不同的变量，因为它们分别和第二行以及第一行相同。

这个做法有一个主要的缺陷：每一行输出都需要和程序的某一部分（并且还有一个变量）相对应。因此，即使是对输出格式的一个简单的修改（例如，除去在问候语和框架之间的空白）都会要求我们重新编写程序。我们将会构造一个更为灵活的输出形式，这样我们就不用把每一行都存储在一个局部变量中了。

让我们仔细地观察一下这个问题。我们可以分别产生输出的每一个字符，不过问候语本身是个例外，因为我们已经把它定义成一个 string 类型的变量了。我们发现，并没有必要把输出字符存储在变量中，因为一旦输出了某个字符，就再也不会需要它了。

2.2 程序的整体结构

让我们先来回顾一下不需要重写的那一部分程序：

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

int main()
{
    //请求用户输入姓名
    std::cout << "Please enter your first name: ";

    //读入姓名
    std::string name;
    std::cin >> name;

    //构造我们将要输出的信息
    const std::string greeting = "Hello, " + name + "!";

    //我们要重写这一部分代码……

    return 0;
}
```

在我们开始重写 “我们要重写这一部分代码……”所表示的这一部分程序前，我们已经定义好了 name、greeting，并导入了标准库中的相关名称。我们会一部分一部分地来逐步建立这个程序的新版本，然后，在§2.5.4 中会把所有的这些程序段连接到一起。

2.3 输出数目未知的行

我们可以把我们的输出看作是一个矩形的字符数组，而且我们要每次输出数组中的一行。虽然我们不知道它的行数是多少，但是我们知道如何去算出这个行数。

问候语占了一行，框架的顶部和底部也是各占一行。到目前为止，我们已经数出了三行。如果我们知道应该在问候语句和框架之间留多少个空白行的话，那我们就能够把这个行数乘以 2 然后再加 3，从而就获得了所有要输出的行数：

```
//围住问候语句的空白行数
const int pad = 1;
//所有要输出的行数
const int rows = pad * 2 + 3;
```

我们希望让定义了空白行数的那一部分程序更易于查找，因此我们给了这个行数一

称。这个名为 pad 的变量代表问候语和框架之间的填充行数。定义了 pad 以后，我们就可以用它来计算 rows 了，rows 就是我们将要控制输出的行数。

内建类型 int 是用于整数的最常见的类型，因此我们为 pad 和 rows 变量选择了这种类型。我们也可以说这两个变量都是常量，从§1.2 中我们知道，常量确保了 pad 或 rows 的值将不会被改变。

更进一步，我们想在左边、右边、顶部和底部都使用同样数目的空白。因此，一个变量将同时为四边服务。如果我们能够谨慎地用这个变量来表示空白的数目，那么，只要我们修改程序，给这个变量一个不同的值，就可以改变输出框架的大小。

我们已经算出了需要输出的行数；那么，下一步工作就是输出数据：

```
//把输出和输入分隔开
std::cout << std::endl;

//输出 rows 行
int r = 0;

//不变式：到目前为止，我们已经输出了 r 行
while (r != rows) {
    //输出一行（像我们将在§2.4 中描述的一样）
    std::cout << std::endl;
    ++r;
}
```

正如我们在§1.2 中所做的那样，我们首先输出了一个空白行，因此在输出和输入之间会有一些空白。在这个程序段的其余部分中包含有很多新的概念，对这些概念我们需要进行深入的探讨。一旦我们理解了这个程序段是如何工作的，就可以开始考虑怎样独立地输出每一行了。

2.3.1 while 语句

我们用一条 **while 语句**来控制程序的输出行数，只要某个特定的条件为真，while 语句就会重复地执行一条特定的语句。while 语句的形式如下：

```
while (condition)
    statement
```

其中，statement（语句）通常叫做 **while 循环体**。

while 语句首先检测条件（condition）的值。如果条件为假（false），那么它就根本不会执行循环体。否则，它会执行循环体一次，然后它再次检测条件，以此类推……while 在检测条件和执行循环体这两个操作之间轮换，一直到条件为假为止，只要条件为假，程序就马上会从位于整条 while 语句的结尾之后的那条语句开始继续往下执行。

简单来说，在我们的例子中，我们可以把 while 语句说成是："只要 r 的值不等于 rows 就在{}内执行语句。"

习惯上，我们会把 while 语句的循环体放在单独的一行中并适当缩进，这样做是为了提高程序的可读性。但下面的写法也没有错：

```
while (condition) statement
```

不过，如果我们这样写的话，就应该考虑一下，这样会不会给其他可能会阅读我们的程序的人造成麻烦。

值得注意的是，在这个描述中，我们并没有在 statement 之后使用分号。实际上，statement 既可以是单独的一条语句，也可以是一个**块语句**。块语句是一系列的零条或多条用{}括起的语句。如果 statement 只是一条普通语句，那么在这条语句的末尾会有它自己的分号，这样我们就没有必要再用另一个了；如果是块语句的话，块的}符号标志着语句的结束，那么我们也没有必要使用一个分号。由于一个块语句就是一系列用括号括起的语句，所以从§0.7 我们可以知道，块语句本身就构成了一个作用域。

while 语句在开始执行的时候首先测试它的**条件**（condition），条件是一个出现在需要真值的上下文中的表达式。表达式 r != rows 就是条件的一个例子。这个例子中使用了**不等式运算符** !=来比较 r 和 rows。这样的表达式是布尔（bool）类型的，布尔类型是一种用来表示真值的内建类型。它的两个可能的值分别是：true（真）和 false（假），这两个值所代表的意义就如其字面所示的那样。

在这个程序中，我们在 while 循环体的最后一条语句中使用了一种新的语言特性，这条语句是：

```
++r;
```

++是一个**增量运算符**，它的作用是递增——也就是，对变量 r 加 1。我们也可以用如下的等价形式表示：

```
r = r+1;
```

但是，对一个对象加 1 是一种很常见的操作，因此用一个特殊符号来表示这种操作是一种很实用的方法。此外，正如我们在§5.1.2 中将看到的那样，把一个值变成它的直接后继数值的概念和计算任意值的概念形成了鲜明的对比，这一点对于抽象数据结构是十分重要的，因此我们有必要使用一个特殊的记号。

2.3.2 设计 while 语句

有时候，我们很难准确地判断出应该在 while 语句中编写什么样的条件。同样地，想要精确地理解一条特定的 while 语句所做的动作可能也会是比较困难的。不过，我们不用花太大的力气就可以看出，§2.3 中的 while 语句是用来进行输出的，而且输出的行数取决于 rows 的值。但是，我们怎么样才可以准确地知道程序将会输出多少行呢？例如，我们怎样才能知道，这个输出行数究竟是 rows、rows-1、rows+1 或者是其他完全不同的值呢？我们可以人工跟踪

语句，在程序运行中留意每一条语句的执行效果，但是我们又怎么能知道用这个方法不会犯错呢？

我们可以使用一种有用的技术来编写以及理解 while 语句，这种技术依赖于两个关键的概念—— 一个是关于 while 语句的定义；另一个则通常与程序的行为有关。

第一个概念就是，当 while 语句结束时，它的条件必须为假——否则，while 将不会结束。例如，当§2.3 中的 while 语句终止的时候，我们知道表达式 r!=rows 的结果为假，也就是说 r 等于 rows。

第二个概念是**循环不变式**（Loop invariant）。循环不变式是 while 语句的一个特性，每当 while 语句要测试它的条件的时候，我们都能断言这个特性为真。我们选择一个不变式来确保程序的行为会跟我们的意图相符合，而且我们编写的程序要让不变式有适当的为真的次数。虽然不变式并不是程序正文的一部分，但是，在程序设计中它是一个很有价值的工具。我们所能想像到的每一条有用的 while 语句都会有一个不变式与之关联。在注释中描述不变式能让 while 语句更容易理解。

为了让这个讨论更加具体，让我们再来看看§2.3 中的 while 语句吧。紧接在 while 语句之前的注释指出了不变式是什么：到目前为止，我们已经输出了 r 行。

为了判断这个程序段中的不变式是否正确，我们必须在每次 while 语句要测试它的条件的时候都确保这个不变式为真。为此我们需要在程序的两个特定时刻来验证它是否为真。

第一个时刻是在 while 语句第一次测试它的条件之前。在我们的例子中，在这一时刻，我们很容易就可以证实这个不变式：因为到目前我们写的输出行数还是为零，所以很明显，只要把 r 设为 0 就可以让不变式为真。

第二个时刻是在我们到达了 while 循环体的结尾并即将开始下一次循环之前。如果不变式在这个时刻为真，那么在下一次 while 测试条件时，它也会为真。这样的话，不变式每次都将为真。

如果我们编写的程序符合了这两个要求（让不变式在 while 语句开始之前以及在到达 while 循环体结尾的时刻都为真），那么，我们就确保了不但在 while 语句每一次测试条件时不变式为真，而且在 while 语句终止时它也为真。否则，不变式就会在 while 循环体的某一次循环过程的开始时刻为真并在此之后为假——我们已经消除了这种情况发生的可能性。

接下来，让我们浏览一下我们的程序片段：

```
//不变式：到目前为止，我们已经输出了 r 行
    int r = 0;
//把 r 设为 0，让不变式为真
while (r != rows) {
    // 在这里，我们可以假定不变式为真
    //输出一行，令不变式为假
    std::cout << std::endl;
```

```
    //对 r 加 1，使不变式再次为真
    ++r;
}
//我们可以断定，在这里不变式为真
```

我们的 while 语句的不变式是：到目前为止，我们已经输出了 r 行。在我们定义 r 的时候，给它指定了一个初始值 0。到这一时刻为止，我们还没有输出过任何东西。很明显，把 r 设为 0 可以使不变式为真，我们就已经符合了第一个要求。

为了符合第二个要求，我们必须证实，只要不变式在 while 语句测试条件的时候为真，那么，在一个循环过程经过了条件和循环体而达到了循环体的结尾的时候，不变式还是为真的。

输出一行之后，不变式就会变为假，因为 r 不再是我们已经输出的行数了。不过，在 r 增加了 1 从而算出了输出的行数之后，不变式就将会再次为真。这样的话，我们就会在循环体的结尾让不变式为真，从而符合第二个要求。

因为程序已经符合了这两个要求，所以我们知道，在 while 语句结束之后，我们已经输出了 r 行。此外，我们已经见到了 r==rows。与此同时，这两个事实表明了，rows 就是我们已经输出的全部行的数目。

我们迟早都会在不同情况中用到这个理解循环的策略。通常我们会找出一个不变式来说明包含在循环中的变量的相关特性（我们已经输出了 r 行），同时我们还要使用条件来保证，在循环终止时这些变量都会具有有用的值（如 r==rows）。循环体的工作则是要处理相关的变量从而最终把条件安排为真，同时它还要保持不变式的真实性。

2.4 输出一行

既然我们已经知道了怎样去输出特定数目的行，那我们就可以转移我们的注意力来输出单独的一行了。换句话说，我们可以开始填充§2.3 中的注释“输出一行”所代表的那一部分程序了。

让我们先来观察一下，实际上，所有输出行的长度都是一样的。如果我们把输出看作是一个矩形数组，那么这个长度就是在数组中的列数。我们可以算出这个数目，为此，把填充值乘以 2 然后与 greeting 的长度相加，最后再加上 2 以表示两端的星号：

```
const std::string::size_type cols = greeting.size( ) + pad*2 + 2;
```

在阅读这个定义的时候，我们很容易就可以看出，cols 是一个常量（const）。这样我们就确保了，在定义了 cols 之后，它的值将不会被修改。在定义中比较难理解是，我们使用了一个不熟悉的类型来定义 cols，这个类型就是 std::string::size_type。我们知道，第一个::是作用域运算符，限定名 std::string 则意味着名称 string 是来自于名字空间 std 的。同样地，第二个::表示，我们希望使用来自 string 类的名称 size_type。与名字空间和块语句一样，类定义了它们自己的作用域。std::string 把 size_type 定义为一个适当的类型名称，我们使用这种类型来保存一个字符串中的字符个数。无论何时，只要我们需要一个局部变量来保存一个字符串的长

们就应该使用 std::string::size_type 来作为这个变量的类型。

我们之所以要把 cols 的类型定为 std::string::size_type，是因为我们要确保 cols 足以容纳 greeting 的字符个数——不管这个数目可能有多大。我们可能会把 cols 的类型定为 int，而且，实际上这样做也有可能是行得通的。然而，cols 的值取决于我们程序的输入长度，而且我们无法控制程序的可能的输入长度。因此，我们可以想像得到，说不定有人会为程序输入一个太长的字符串，以致于 int 类型的数据不足以容纳它的长度。

int 类型对于 rows 来说是足够大的，因为行数仅仅依赖于 pad 的值，而这个值是在我们的控制范围之内的。所有的 C++系统环境都允许 int 类型的变量所取的最大值至少达到 32767，而这个值已经是很大的了。不过，无论何时，只要我们想要定义一个变量来保存特定数据结构的大小，就应该养成使用库为特殊用途而定义的类型的好习惯。

包含在字符串中的字符个数不可能是负数。因此，std::string::size_type 是一个无符号类型——这种类型的对象不可以包含负数值。这个性质并不会影响本章中的程序，但是在§8.1.3 中，我们将会看到它是非常重要的。

算出了待输出的字符个数之后，我们就可以使用另一条 while 语句来输出这些字符：

```
std::string::size_type c = 0;

//不变式：到目前为止，在当前行中我们已经输出了 c 个字符
while (c != cols) {
    //输出一个（或多个）字符
    //修改 c 的值，保持不变式的真实性
}
```

这条 while 语句所做的操作跟§2.3 中的那条很相似。不过，在循环体中有一个不同的地方：这一次我们是说“输出一个（或多个）字符”，而不是像在§2.3 中所做的那样输出一整行。我们没有理由在一次循环过程中仅仅输出一个字符，但只要输出了至少一个字符，就可以保证循环的继续了。我们所要做的只不过是要确保在这一行所输出全部字符的个数恰好等于 cols。

2.4.1 输出边界字符

接下来，我们要做的就是判断待输出的是什么字符。如果我们能注意到以下的一点，那么我们的问题就很容易被解决：如果是位于第一行或最后一行或者是位于第一列或最后一列的话，那么我们就应该输出一个星号。此外，我们可以使用循环不变式的知识来判断是否到了要输出一个星号的时刻。

例如，如果 r 等于零，那么根据不变式可以知道，我们还没有输出过任何一行，这就意味着我们将要输出第一行的字符。同样地，如果 r 等于 rows-1，那么我们知道已经输出了 rows-1 行，因此，现在我们必须输出最后一行的字符。根据同样的推理我们可以判断，如果 c 为零的话，那么我们将要输出第一列字符；如果 c 等于 cols - 1，那么我们要输出的则是最后一列的字符。应用这个知识，我们就能为我们的程序编写更多的代码了：

```
//不变式：到目前为止，在当前行中我们已经输出了 c 个字符
while (c != cols) {
    if (r == 0 || r == rows - 1 || c == 0 || c == cols -1) {
        std::cout << "*";
        ++c;
    } else {
        //输出一个以上的非边界字符
        //修改 c 的值，保持不变式的真实性
    }
}
```

这条语句中引入了很多的新概念，因此我们要对它进行详细的解释。

2.4.1.1 if语句

while 循环体是由一个块语句（§2.3.1）组成的。这个块语句包含有一条 **if 语句**，我们使用这条 if 语句来判断是否到了要输出一个星号的时刻。if 语句有两种形式：

```
if(condition)
    statement
```

或者

```
if(condition)
    statement1
else
    statement2
```

与 while 语句一样，if 语句的条件（condition）也是一个表达式，这个表达式会产生一个真值。如果条件为真(true)，那么程序将执行紧跟在 if 后面的语句。在 if 语句的第二种形式中，如果条件为假(false)，那么程序将执行紧跟在 else 后面的语句。

值得注意的是，跟我们所描述的 while 语句的形式一样，我们仅仅是使用了传统的格式来说明 if 语句。不过，读者很快就会发现，如果我们在程序代码中遵循了本书例子中使用的格式习惯，那么程序就会更加简明易懂。

2.4.1.2 逻辑运算符

下面的这个条件如何呢？

```
r==0||r==row-1||c==0||c==cols-1
```

如果 r 等于 0 或等于 rows-1，又或者是 c 等于 0 或等于 cols-1，那么这个条件为真。我们在这个条件中使用了两个新的运算符==和||。C++程序使用==符号来检测**相等**，以区别于表示赋值的符号=。因此，r==0 会产生一个布尔(bool)值来表示 r 的值是否为 0。**逻辑或**运算符的写法为||，如果它的两个操作数中至少有一个为真(true)，那么它的结果就为真。

关系运算符的优先级比算术运算符的低。在包含不止一个操作数的表达式中，优先级定义了运算符的组合规则。例如：

```
r==rows-1
```

等价于

```
r==(rows-1)
```

而不是

```
(r==rows)-1
```

这是因为算术运算符-的优先级比关系运算符==的高。换句话说，在这个程序中，我们所期望的操作是，把 rows 减 1，然后把减法的结果与 r 进行比较。如果我们希望把一个子表达式当作一个单独的操作数来使用，那我们可以用括号来括住此子表达式——这样就可以忽略优先级了。例如，如果我们确实希望执行(r==rows)-1，那么，只要用括号来括起 r==rows 就达到目的了。这个表达式将会比较 r 和 rows，然后再从比较所得的结果中减去 1，这样就产生了等于 0 或-1 的结果，但结果究竟是哪个值则取决于 r 和 rows 是否相等。

逻辑或运算符检测它的任一个操作数是否为真。它的形式如下：

```
condition1||condition2
```

像平常一样，这个表达式的 condition1 和 condition2 都是条件——也就是产生真值的表达式。||表达式会产生一个布尔值，如果任何一个条件为真的话，那么这个布尔值也就为真。

||运算符的优先级比关系运算符的低，而且，像大多数的 C++二元运算符一样，||运算符是左结合的。此外，它具有一个大多数 C++运算符所没有的特性：如果程序发现||的左操作数的值为真，它就根本不会再去计算右操作数。这个属性经常被称作短路求值（short-circuit evaluation），而且，正如我们将在§5.6 中看到的那样，它对我们的程序设计有着非常重要的作用。

因为||是左结合的，同时由于||、==和-之间的优先关系，所以

```
r == 0 || r == rows - 1 || c = 0 || c == cols - 1
```

的意义与我们把它的所有子表达式括在括号中所得到的形式等价：

```
((r == 0 || r == (rows - 1)) || c = 0 ) || c == (cols - 1)
```

为了使用短路求值策略来计算后面的表达式，程序会首先计算最外面的||的左操作数，也就是：

```
(r == 0 || r == (rows - 1)) || c = 0
```

而为此，它必须先来计算里面的||的左操作数，即：

```
r == 0 || r == (rows - 1)
```

这就意味着要计算：

```
r == 0
```

如果 r 等于 0，那么下面的每一个表达式

```
r == 0 || r == (rows - 1)
```

```
(r == 0 || r == (rows - 1)) || c == 0
((r == 0 || r == (rows - 1)) || c == 0) || c == (cols -1 )
```

都一定为真。如果r是一个非零值，那么下一步的工作是比较r和rows-1是否相等。如果不等，程序就会把c与0比较，如果结果还是不等，那么它将比较c和cols-1以确定最后的结果。

换句话说，如果我们编写了一系列用||运算符分隔的条件，就意味着程序会依次检测每一个这样的条件。如果有任何一个内部条件为真，那么整个条件表达式为真；否则，整个条件表达式的结果就为假。一旦||运算符判断出了它自己的结果，那么检测操作就会马上停止，因此，如果有任何一个内部条件为真，那么程序就无须测试随后的条件了。如果回顾一下上面的细节，我们就能看出，那四个相等检测所做的只不过是检查我们是否位于第一行、最后一行、第一列或最后一列。因此，如果我们位于最顶部或最底部的一行，又或者是位于第一列或最后一列，那么if语句就会输出一个星号。否则，它会执行其他的操作，而这正是我们即将要定义的。

2.4.2 输出非边界字符

现在，让我们来编写对应于§2.4.1中的程序片段的注释的语句：

```
//输出一个以上的非边界字符
//修改c的值，保持不变式的真实性
```

这些语句必须处理那些不属于边界部分的字符。很明显，每一个这样的字符要么是空白，要么就是问候语的一部分。我们惟一的问题是要弄清楚，它究竟是哪一个字符以及我们应该如何处理它。

我们首先检查即将要输出的是不是问候语中的第一个字符，为此我们要判断，我们是不是位于适当的行中以及是不是在这一行的适当的列上。我们要查找的适当的行就是位于由星号组成的第一行和附加的个数为pad的填充行之后的那一行；而那个适当的列则紧跟在开头的星号和在星号后面的个数为pad的空白之后。根据不变式的知识我们知道，如果r等于pad+1，那么我们位于适当的行中；如果c等于pad+1，就表示我们在正确的列上。

换句话说，为了判断即将要输出的是不是问候语的第一个字符，我们必须检查r和c是否都等于pad+1。如果我们到达了正确的位置，我们就会输出问候语；否则，我们将输出一个空白。在这两种情况下，我们都要相应地更新c的值：

```
if (r == pad + 1 && c == pad +1) {
    std::cout << greeting;
    c += greeting.size();
} else {
    std::cout << " ";
    ++c;
}
```

我们在if语句的条件中使用了**逻辑与**运算符。跟||运算符一样，&&运算符会检

件并产生一个真值。&&运算符是左结合的，而且它使用短路求值策略。与||运算符不同，&&运算符仅仅在两个条件都为真的时候才会产生真值。如果有任何一个条件为假，那么&&的结果就会为假。而且，只有在第一个条件为真的情况下第二个条件才会被检测。

如果检测成功，那么我们就应该输出问候语。在我们输出了问候之后，不变式会变为假，这是因为 c 不再等于我们在当前行中已经输出的字符数了。如果我们调整 c 的值，让它等于我们已经输出的字符数，就会使不变式再次为真。在那个对 c 的值进行更新的表达式中，我们使用了另一个新的运算符——这个运算符就是**复合赋值**运算符，在输出姓名的时候，我们用它来调整 c 的值，让 c 等于我们已经输出的字符数。这样的复合赋值运算是一种简写形式，它把左操作数与右操作数相加并把结果存储在左操作数中。换句话说，我们所编写的 c+=greeting.size()语句与 c=c+greeting.size()是等效的。

最后要考虑的是，我们有可能不是位于边界上，这样就不用输出问候语。此时，我们应该输出一个空格并对 c 加 1 以使不变式再次为真。我们是在 if 语句的 else 分支中完成这个动作的。

2.5 完整的框架程序

到目前为止，我们已经修正了整个程序，不过程序的代码太散乱了，对它们进行查找比较困难。因此，我们有必要再次出示完整的程序代码。不过，在这样做之前，我们想从三个方面来缩短我们的程序。

第一个缩写是一种声明，这种声明可以让我们仅此一次地说明一个特定的名称是来自标准库的。这样的话，我们就可以略去很多个重复的 std::了。第二个缩写则是一个简写形式，我们用这个简写形式来编写一种很常用的 while 语句。最后，我们可以在一个而不是在两个地方对 c 加 1，这样也可以稍微缩短我们的程序。

2.5.1 略去重复使用的 std::

现在，你可能会对在每一个标准库的名称之前都看到（或者编写）std::而感到厌烦。明确地使用 std::是一个很好的做法，因为这样可以提示读者哪些名称是来自标准库的，不过，到现在为止，读者们可能已经对这些名称相当了解了。

C++提供了一个方法来让我们声明，一个特定的名称总是应该被看作是来自特定的名字空间的。例如，在编写了下面的语句之后：

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

我们就可以说，我们想用名称 cout 来专门指代 std::cout，而且我们不会定义自己的 cout。一旦编写了这种形式的语句，我们就可以用 cout 来代替 std::cout 了。

习惯上，我们把这样的一个声明称为 **using 声明**。出现在 using 声明中的名称所具有的特性与其他名称相似。例如，如果一个 using 声明出现在花括号中，那么它定义的名称的作用域会从定义的地方开始，到对应的结束花括号处结束。

从现在开始，我们将使用 using 声明来缩短我们的程序。

2.5.2 使用 for 语句来缩短程序

让我们回顾一下我们在§2.3 的程序中所使用的控制结构。如果我们只是看最外层结构的话，那么会看到：

```
int r = 0;

while (r != rows) {
    //下面的代码不会改变 r 的值
    ++r;
}
```

我们经常会用到这种形式的 while 语句。在开始编写 while 语句之前，我们定义且初始化了一个局部变量，在条件中我们会对这个变量进行测试。while 的循环体改变了这个变量的值，因此，在循环体的最后条件将不成立。因为这种控制结构很常见，所以 C++也为我们提供了一种简写形式：

```
for ( int r = 0; r != rows; ++r) {
    //下面的代码不会改变 r 的值
}
```

这两个例子中的任意一个都会让 r 取一系列的值，而这些值中的第一个是 0，最后一个是 rows-1。我们可以把 0 看作是一个区间的开始而把 rows 看作是区间的**越界**值（**off-the-end value**）。这样的一个值域被称为**半开区间**，它的写法通常为[开始值，越界值)。这对不平衡的括号[)提醒读者，这个区间是不对称的。例如，区间[1,4)包含了 1，2 和 3，但并没有包含 4。同样地，我们说，r 在[0,rows)中取值。

for 语句的形式如下：

```
for(init-statement condition; expression)
    statement
```

我们经常把第一行称为 **for 语句头**，它会对后面的语句（statement）进行控制，而跟在 for 语句头后面的语句通常被称为 **for 循环体**。值得注意的是，在初始化语句（init-statement）和条件（condition）之间并没有分号，这是因为在初始化语句的末尾通常都会有一个属于它自己的分号。

for 语句首先执行 for 语句头中的初始化语句，而且这个操作只在 for 语句开始执行的时候进行一次。典型的做法是，我们用初始化语句来定义并初始化循环控制变量——作为条件的一部分，这个变量将会被检测。如果一个变量是在初始化语句中定义的，那么在程序退出 for 语句的时候，这个变量将会被销毁。因此，位于 for 语句之后的代码不可以访问这个变量。

在每一次的循环过程（包括第一次）中，程序都会对条件进行计算。如果条件产

那么程序就会执行 for 循环体。完成了这个动作之后，它会接着执行表达式（expression）。然后它再次检测条件，如果检测成功的话就继续执行 for 循环体，接下来它再次执行 for 语句头的表达式……这个过程会一直持续到检测条件失败为止。

一般而言，for 语句的意义相当于：

```
{
    init-statement
    while (condition) {
        statement
        expression;
    }
}
```

在上面的语句中，我们用一个额外的花括号把初始化语句和 while 语句括了起来，这样的话，我们就限定了在初始化语句中声明的所有变量的生存期。我们应该特别注意对分号的使用。因为 init-statement 和 statement 本身就是语句，而且在必要时它们会带有自己的分号，所以不用在之后加上分号。如果我们想把表达式变成语句，就必须在表达式后包括一个分号。

2.5.3 压缩检测

我们可以把在§2.4 中与注释“**输出一个以上的非边界字符**”相对应的代码分成三种情况来考虑：我们将输出一个星号、一个空格或整条问候语。正如我们的程序所示，在输出一个星号之后，我们将会调整 c 的值以保持不变式的真实性；而且，在输出了一个空格之后，我们还得再次对它进行调整。这个做法本身并没有错，但是，我们经常都可以在程序中改变检测的顺序，从而把两条以上的等价的语句合并成一条语句。

上面所说的那三种情况是互相排斥的，我们可以按任何顺序来检测它们。如果我们首先检测是否要输出问候语，那么我们知道，在另外的两种情况中，我们只要对 c 加 1 就足以保持不变式的真实性了，因此，可以把两个加 1 操作压缩成一个：

```
if (we are about to write the greeting) {
    cout << greeting;
    c += greeting.size();
} else {
    if (we are in the border)
        cout << "*";
    else
        cout << " ";
    ++c;
}
```

在压缩了加 1 操作之后，我们就会发现，两个块语句都是由单独的一条语句构成的，因此我们能够去掉两对花括号。值得注意的是，不管我们是不是位于边界上，那条缩进量不

他语句的++c;语句都会被执行。

2.5.4 完整的框架程序

如果把所有的程序片段都连接到一起并使用这三种缩写技术，那么我们就会得到下面的程序：

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

//声明我们使用的标准库名称
using std::cin;     using std::endl;
using std::cout;    using std::string;

int main()
{
    //请求用户输入姓名
    cout << "Please enter your first name: ";

    //读入用户输入的姓名
    string name;
    cin >> name;

    //构造我们将要输出的信息
    const string greeting = "Hello, " + name + "!";

    //围住问候语的空白个数
    const int pad = 1;

    //待输出的行数与列数
    const int rows = pad * 2 + 3;
    const string::size_type cols = greeting.size() + pad * 2 + 2;

    //输出一个空白行，把输出和输入分隔开
    cout << endl;

    //输出 rows 行
    //不变式：到目前为止，我们已经输出了 r 行
    for (int r = 0; r != rows; ++r) {
        string::size_type c = 0;

        //不变式：到目前为止，在当前行中我们已经输出了 c 个字符
        while (c != cols) {

            //应该输出问候语了吗？
            if (r == pad + 1 && c == pad + 1) {
```

```
                cout << greeting;
                c += greeting.size();
            } else {

                //我们是位于边界上吗?
                if (r == 0 || r == rows -1 ||
                    c == 0 || c = cols - 1)
                    cout << "*";
                else
                    cout << " ";
                ++c;
            }
        }

        cout << endl;
    }

    return 0;
}
```

2.6 计数

大多数熟练的 C++程序员都会有一个乍看上去很奇怪的习惯：他们的程序永远都是从 0 而不是从 1 开始计数的。例如，如果我们把上面程序的外层 for 循环的实际计数减 1，我们就得到：

```
for (string::size_type r = 0; r != rows; ++r) {
    //输出一行
}
```

我们也可以把这个循环写成：

```
for (string::size_type r = 1; r <= rows; ++r) {
    //输出一行
}
```

第一种形式从 0 开始计数并使用!=来进行比较；另一种则从 1 开始计数并使用<=来进行比较。实际上，在每一种情况中的循环次数都是一样的。那么，我们为什么选择要第一种而不用另一种呢？

之所以要从 0 开始计数的一个原因是，这样做可以让我们使用不对称的区间来表示间隔。例如，跟使用区间[1,row]来描述第二条 for 语句一样，使用区间[0,row)来描述第一条 for 语句也是显得同样的自然。

跟对称区间相比，对不对称区间的使用通常会更为简便，之所以会这样是由于一个[illegible]性质：在一个形式为[m,n)的区间包含有 n - m 个元素，而在形式为[m,n]的区间中则有

个元素。例如，我们很容易就可以看出在区间[0,rows)中的元素个数（也就是，rows-0 或 rows），然而，区间[1，rows]中的元素个数却没有这么明显。

就空区间而言，在不对称区间和对称区间之间的差别是特别明显的：如果使用不对称区间，那我们就能用[n,n)来表示一个空区间；而对于对称区间我们却要使用[n,n-1]。如果一个区间的末尾比开头小，那么，在设计程序的时候我们就有可能会因此而碰到无穷无尽的麻烦。

另一个从 0 开始计数的原因是，这样做可以令循环不变式更易于表达。在我们的例子中，从 0 开始计数会让不变式更为简明易懂：到目前为止，我们已经输出了 r 行。如果我们从 1 开始计数的话，那又会怎样呢？

对此，可能会有人说，不变式是“我们即将要输出第 r 行”，但是这种表达方式对于不变式来说是不适合的。这是因为，在 while 语句最后一次检验其条件的时候，r 等于 rows+1，而我们只需要输出 rows 行。因此，我们并不是即将要输出第 r 行，这样的话，不变式就不会为真！

我们可以把不变式说成是“到目前为止，我们已经输出了 r-1 行”。然而，既然要这样表达的话，那么我们为什么不说得简单点，让 r 从 0 开始呢？

还有一个原因可以说明我们为什么要从 0 开始计数，那就是，这样做可以让我们选择!=而不是用<=来进行比较。这个差别看起来可能是无关紧要的，但是，在循环结束时，它会影响我们对程序状态的了解。例如，如果条件是 r!=rows，那么，在循环结束时，我们可以知道 r==rows。因为不变式表明了，我们已经输出了 r 行，所以在循环结束的时候，我们知道刚好是输出了 rows 行。另一方面，如果条件是 r<=rows，那么所有能够证明的只不过是我们至少输出了 rows 行。但是，很明显，这意味着我们有可能已经输出了更多行。

如果我们从 0 开始计数，就可以用 r!=rows 来作为条件，这样能确保循环次数刚好是等于 rows；又或者，如果我们只是想确保循环次数大于等于 rows，那么就可以使用 r<rows。如果我们是从 1 开始计数的，那么可以用 r<=rows 来确保循环次数至少为 rows——但是，如果我们想让 rows 等于一个精确的数值的话，又该如何呢？这样的话，我们就必须检测一个更加复杂的条件，例如 r==rows+1。可是，这种额外的复杂度并不能为我们提供任何补偿性的优势。

2.7 小结

表达式：C++从 C 中继承了一套数目众多的运算符，而我们已经使用了这些运算符中的几个。另外，跟我们所看到的输入和输出运算符一样，C++程序可以对核心语言进行扩展，使那些作用于内部类型的运算符也可以作用于自定义类型的对象之上。如果我们想编写出高效的 C++程序，就必须正确地理解复杂的表达式。在我们理解这样的表达式的时候，首先要理解以下的概念：

- 操作数的组合方式，这是由在表达式中使用的运算符的优先级和结合性控制的

- 操作数是怎样被转换成其他类型的——如果可以转换的话。
- 操作数的运算次序。

不同的运算符具有不同的优先级。赋值运算符和只有一个操作数的运算符是右结合的，而除此之外的大多数运算符都是左结合的。在这里，我们把大多数常用的运算符都列了出来——不管我们有没有在本章中用到它们。我们按优先级从高到低的顺序来排列它们，对同一优先级的运算符我们用双行线来分组。

x.y	对象 x 的成员 y
x[y]	在对象 x 中索引为 y 的元素
x++	对 x 加 1，返回 x 的原始值
x- -	把 x 减 1，返回 x 的原始值
++x	对 x 加 1，返回相加后的值
- -x	把 x 减 1，返回相减后的值
! x	逻辑非，如果 x 的值为 true（真），那么! x 的值就是 false（假）
x*y	x 乘以 y 的积
x/y	x 除以 y 的商。如果两个操作数都是整数，系统环境会选择是否向 0 或-∞ 进行舍入
x%y	x 除以 y 的余数，与 x – ((x – y)*y)等价
x + y	x 与 y 的和
x – y	x 减去 y 的结果
x>>y	对于整数 x 和 y，x 右移 y 位；y 必须是非负数。如果 x 是一个 istream 类型的对象，那么从 x 把数据读到 y
x<<y	对于整数 x 和 y，x 左移 y 位；y 必须是非负数。如果 x 是一个 ostream 类型的对象，那么把 y 写到 x 中
x *relop* y	是一个关系运算符，它产生一个布尔值来指示关系的真实性。运算符（<,>,<=和>=）的意义如字面所示
x==y	产生一个布尔值来指示 x 和 y 是否相等
x!=y	产生一个布尔值来指示 x 和 y 是否不相等
x&&y	产生一个布尔值来指示 x 和 y 是否两个都为真。只有在 x 为真时才会计算 y
x ‖ y	产生一个布尔值来指示 x 和 y 中的任一个是否为真。只有在 x 为假时才会计算 y
x = y	把 y 的值赋给 x，此运算的结果为 x
x *op*= x	复合赋值运算符：等价于 x =x *op* y。在这里，*op* 是一个算术运算符或移位符
x ? y1 : y2	若 x 为真，则产生 y1，否则产生 y2。只计算 y1 和 y2 中的一个

一般来说，表达式中的运算符并不一定是按以上的次序来进行计算。因为计算次序并不是固定的，所以我们应该避免编写如下的表达式：这些表达式中的某一个操作数会依赖于另一个操作数的值。在§4.1.5 中，我们将会看到一个这样的例子。

操作数会根据需要而被转换成适当的类型——如果可以转换的话。在表达式或关系式中的数值操作数会按照**普通算术转换规则**而被转换成其他类型，我们将会在§A.2.4.4 中对此

细的描述。基本上，普通算术转换规则会尽量保持精度。较小的类型会被转换成较大的类型，而有符号类型则会被转换成无符号类型。算术值可以会被转换成布尔值：0 值可以被看成是 false（假）；其他的所有值则被看成是 true（真）。自定义类型的操作数被转换成类型指定的类型。在第 12 章中将会看到，我们是怎样控制这样的转换的。

类型：

bool	代表真值的内部类型；其值可以是 true 或 false
unsigned	只包含非负数的整数类型
short	必须保存至少 16 位的整数类型
long	必须保存至少 32 位的整数类型
size_t	无符号整数类型（定义在<cstddef>中），可以保存任何对象的长度
string::size_type	无符号整数类型，可以保存任何字符串的长度

半开区间包含且只包含了它们的端点数值中的一个。例如，[1,3)包括了 1 和 2，但不包含 3。

条件：产生一个真值的表达式。条件中的算术值会被转换成布尔值：非零值被转换成 true；零值则被转换成 false。

语句：

using namespace-name::name;

把 name 定义为 namespace-name::name 的替代名。

type-name name;

定义了类型为 type-name 的变量 name。

type-name name=value;

定义了类型为 type-name 的变量 name 并用 value 的一个拷贝来对 name 进行初始化。

type-name name(args);

定义类型为 type_name 的变量 name 并根据 args 中的变量来构造 name 的值。

expression;

执行 expression（表达式）并产生它的副作用。

{ statement(s)}

这种形式的语句被称为块语句。块语句按顺序执行零条或多条语句（statement）。我们可以在任何需要一条语句的地方使用块语句。在花括号内定义的变量的作用域被限定在块语句内——也就是，作用域从变量被定义的地方开始，到标志该块语句结束的右花括号处结束。

while (condition) statement

如果条件（condition）为假(false)，就不执行任何操作；否则，执行 statement 然后重复整个 while 语句。

for(init-statement condition; expression) statement

与{ init-statement while (condition) {statement expression;} }等价。

if (condition) statement

如果条件（condition）为真，则执行 statement。

if (condition) statement1; else statement2

如果条件（condition）为真则执行 statement1；否则执行 statement2。每一个 else 都与最邻近的能与之匹配的 if 相关联。

return val;

退出函数且返回 val 给函数的调用程序。

习题

2-0 编译并运行我们在本章中介绍的程序。

2-1 改写框架程序，输出跟框架没有间隔的问候语。

2-2 在我们的框架程序中，我们使用了一定数目的空格来把问候语和顶部以及底部边界分隔开来。现在，重新编写这个程序，在重写的程序中使用数量跟原程序不同的空格来把各边界和问候语分隔开。

2-3 重写框架程序，让用户自己提供在框架和问候语之间的空格个数。

2-4 在框架程序中的空白行是用来把边界和问候语分隔开的，程序每次一个字符地输出了大部分的空白行。改写这个程序，让它在单独的一条输出表达式中输出所有的空白行。

2-5 编写一个程序，让它输出一系列的“*”字符，程序输出的这些字符将构成一个正方形，一个长方形和一个三角形。

2-6 下面的代码是做什么的？

```
int i = 0;
while (i < 10) {
    i +=1;
    std::cout << i <<std::endl;
}
```

2-7 编写一个程序来依次输出从 10~ - 5 的整数。

2-8 编写一个程序来计算区间[1, 10)中的所有数值的乘积。

2-9 编写一个程序，让用户输入两个数值并告知用户在这两个数值中哪一个较大。

2-10 在下面的程序中，对 std::的每一次使用进行解释。

```
int main()
{
    int k = 0;
    while (k != n) {            //不变式：到目前为止，我们已经输出了 k 个星号
        using std::cout;
        cout << "*";
    }
    std::cout << std::endl;     //在这里必须使用 std::
}
```

第 3 章

使用批量数据

我们在第 1 章和第 2 章中讨论的那些程序所做的工作只不过是读一个字符串并把它输出，有时装饰一下输出的格式。然而，与这些简单程序所能够解决的问题相比，大多数的其他问题都会更加复杂。我们经常要在程序中处理多个相似数据段——这也是程序复杂度的一个很常见的来源。

如果说字符串包含有多个字符，那么在这种说法之下，我们的程序已经是这样做了。实际上，正是因为我们有了把未知数目的字符放进一个对象（字符串）中的能力，我们才能以这么简单的方式去编写我们的程序。

在本章中，我们将会讨论到更多的处理批量数据的方法。为了展开这个讨论，我们将会编写一个程序来读学生的考试和家庭作业成绩并计算出总成绩。沿着这个思路，我们将学会存储所有成绩的方法，即使事先并不知道成绩的个数。

3.1 计算学生成绩

假定有一门课程，在这门课程中，每个学生的总成绩都由期末考成绩（占总成绩的 40%）、期中考成绩（占 20%）和家庭作业的平均成绩（占 40%）构成。作为我们的第一次尝试，我们编写了下面的程序来为学生计算总成绩：

```
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>

using std::cin;          using std::setprecision;
using std::cout;         using std::string;
using std::endl;         using std::streamsize;
using std::precision;

int main()
{
```

```
    //请求输入并读入学生的姓名
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    //请求输入并读入期中和期末成绩
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    //请求输入家庭作业成绩
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    //到目前为止，读到的家庭作业成绩的个数及总和
    int count = 0;
    double sum = 0;

    //把家庭作业成绩读到变量 x 中
    double x;

    //不变式：
    //到目前为止，我们已经读到了 count 个家庭作业成绩，
    //而且 sum 等于头 count 个成绩的总和
    while (cin >> x) {
        ++count;
        sum += x;
    }

    //输出结果
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
         << setprecision(prec) << endl;

    return 0;
}
```

跟平常一样，我们首先为程序要用到的库工具编写了#include 指令和 using 声明。这些工具包括我们之前没有见过的<iomanip>和<ios>。头文件<ios>定义了一个类型 streamsize，输入/输出库就是用这个类型来表示长度的。头文件<iomanip>则定义了控制器 setprecision，这个控制器可以让我们指明输出所包含的有效位数。

在我们使用另一个控制器 endl 的时候，并没有必要把头文件<iomanip>也包括进去。

我们在设计程序的时候要经常使用到控制器 endl，所以，为了方便使用，C++把这个控制器的定义放在了<iostream>而不是在<iomanip>中。

程序首先请求输入并读入学生的姓名、期中和期末考试成绩。接下来，它请求输入学生的家庭作业成绩。程序会连续地读入家庭作业成绩，直到它遇到了一个文件结束标志为止。不同的C++系统环境为用户提供了不同的方法来让用户向程序发出这样的一个标志，而最常见的方法是开始新的一行输入，即先按住 control 键，然后按下 z 键（对于在 Microsoft Windows 系统下运行的计算机）或 d 键（对于在 Unix 或 Linux 系统下运行的计算机）。

在读家庭作业成绩的时候，程序使用了 count 来跟踪输入的成绩的个数，同时它把一个动态的成绩总和存储在 sum 中。一旦程序读进了所有的成绩，它就会输出一条问候信息，然后报告学生的总成绩。在这样做的时候，它使用了 count 和 sum 来计算家庭作业的平均成绩。

对我们来说，这个程序中的许多部分都应该是相当熟悉的，不过，在程序中也出现了几个新的概念，我们将会对这些概念进行解释。

第一个新概念出现在读学生考试成绩的那一部分代码中：

```
cout << "Please enter your midterm and final exam grades: ";
double midterm, final;
cin >> midterm >> final;
```

上面的第一条语句是很常见的：它输出了一个信息，在这个例子中，这个信息告诉了学生下一步应该做什么。接下来的语句定义了类型为 double 的变量 midterm 和 final。double 是用来存储双精度浮点数的内部类型；在 C++中还有一个单精度浮点类型，它叫做 float。尽管从表面上看来，float 可能会是适当的类型，不过使用 double 来进行浮点数的计算几乎总会是正确的选择。

在过去，内存的价格要比今天的昂贵得多，而这些类型名称就是在这种背景下出现的。那个较短的浮点类型 float 最多只能提供大约 6 位的十进制有效位，这样的一个数值甚至不足以用来表示一栋住宅的价格。double 类型最少可以提供 10 个有效位，而据我们所知，所有的系统环境都会提供至少 15 位的有效位。在现代的计算机中，double 通常会比 float 更精确，同时跟 float 类型相比，它也不会慢多少。甚至，有时候 double 类型会比 float 更加快。

定义了变量 midterm 和 final 之后，我们就读数值到这两个变量中。跟输出运算符一样(§0.7)，输入运算符返回它的左操作数作为结果。因此，像我们在输出操作中所做的那样，我们可以把输入操作连接起来。所以，

```
cin >> midterm >> final;
```

跟下面的形式等价：

```
cin >> midterm;
cin >> final;
```

这两种形式所做的都是先从标准输入读一个数值到 midterm 中，然后把下一个

final。

下一条语句请学生输入家庭作业成绩：

```
cout << "Enter all your homework grades, "
        "followed by end-of-file: ";
```

如果认真地阅读一下这条语句，我们就会发现，这条语句只有一个<<——尽管它好像是输出了两个字符串直接量。我们之所以能够这样做是因为，在程序中，如果两个以上的字符串直接量仅仅是被空白符分隔开的话，那么这些字符串直接量就会被自动连接到一起。因此，实际上这条语句跟下面的这一条是等价的：

```
cout << "Enter all your homework grades, followed by end-of-file: ";
```

如果程序某些行太长的话，那么对它们的阅读将会是很不方便的。把字符串直接量分成两部分之后，我们就避免了这种情况的出现。

接下来的一段代码定义了两个变量，我们用这两个变量保存我们所读到的信息。当然，在这一段代码中，最让人感兴趣的部分是：

```
int cout = 0;
double sum = 0;
```

值得注意的是，我们给 sum 和 count 指定的初始值都是 0。数值 0 的类型是 int，这就意味着，为了用它来对 sum 进行初始化，系统环境就必须把它转换成 double 类型。如果在初始化 sum 的时候，我们使用 0.0 来代替 0，那么我们就可以避免这个转换操作。但是在这里，这个做法本身并没有什么不同之处：任何一个功能完善的系统环境都会在编译期间进行这个转换操作，因此使用 0 的结果还是一样的，这样做并不会带来运行时间的额外开销。

在本例中，比转换更为重要的是，我们应该给这些变量指定一个初始值。如果不这样做的话，那么我们就要隐含地依赖**缺省初始化**了。缺省初始化操作取决于变量的类型。对于自定义类型的对象，如果我们没有给它们指定一个初始化程序的话，那么类就会自己指定一个。例如，在§1.1 中我们注意到，如果我们不明确初始化一个字符串，那么这个字符串就会被隐含地初始化为空。对于内部类型的局部变量则没有这样的隐含初始化形式。

如果我们不明确初始化内部类型的局部变量，那这些变量就是未定义的，这意味着，在创建变量的时候，系统会给这些变量分配适当的内存单元，而变量的值就是由这些单元中的随机信息组成的。我们可以用有效值来覆盖未定义变量，而除此之外的所有对此变量的操作都是不合法的。许多系统环境都不会对违反这条规则的操作进行检测，而且它们还允许对未定义变量的访问。这样的话，出现冲突和错误结果的机率就会非常高，这是因为，存储在内存中的随机信息几乎从来都不会是一个正确的值——它经常会是一个对类型无效的值。

如果不给 sum 或 count 指定初始值的话，那么我们的程序很可能会以失败告终。这是因为，程序对这些变量所做的第一件事就是使用它们的值：程序读 count 的值并对它加 1；程序

的值并把这个值和我们刚刚读到的输入值相加。同样的道理，我们用不着给 x 指定一个初始值，因为我们对它做的第一件事就是读数值给它，这样的话，我们就会把它可能已经具有的任何值都删除掉。

在程序的 while 语句中，我们惟一感到陌生的是它的条件的形式：

```
//不变式：
//到目前为止，我们已经读到了 count 个家庭作业成绩，
//而且 sum 等于头 count 个成绩的总和
while (cin >> x) {
    ++count;
    sum += x;
}
```

我们知道，只要条件 cin >> x 成功，那么 while 语句的循环就会一直执行下去。在§3.1.1 中我们将讨论把 cin>>x 当作一个条件来使用究竟意味着什么，不过，现在对我们来说最重要的是要知道，如果最近的一个输入请求（也就是 cin >> x）成功的话，那么条件也会成功。

在 while 语句的内部，我们使用了增量运算符和复合赋值运算符，这两个运算符已经在第 2 章出现过了。根据第 2 章的讨论我们知道，++count 对 count 加 1，而 sum+=x 则把 x 加到 sum 中。

最后，我们将解释程序是如何处理输出的：

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

我们的目的是输出具有三位有效位的总成绩，为此我们使用了 setprecision。像 endl 一样，setprecision 也是一个控制器。这个控制器为流的后继输出设置了一个特定的有效位数，这样它就可以对流进行控制了。编写了 setprecision(3)后，我们就可以让系统环境输出具有三位有效位的成绩了，输出的形式通常是（十进制）小数点前有两位有效数，小数点后则有一位。

通过使用 setprecision，我们就改变了出现在 cout 中的所有后继输出的精度。因为这条语句是位于程序末尾的，所以我们知道这样的后继输出并不存在。然而，我们有理由相信，把输出精度重置为修改前的精度是一个明智的做法。为此，我们调用了 cout 中的一个名为 precision 的成员函数(§1.2)。使用了这个函数之后我们就可以知道流在进行浮点数输出时所使用的精度。我们用 setprecision 来把精度设置为 3，跟着输出了总成绩，然后我们再把精度重新设置成从 precision 那里得到的值。在这个计算成绩的表达式中，我们使用了几个算术运算符：*是相乘，/是相除，+则是相加，每一个运算符的意义都是很明显的。

我们可以使用 precision 成员函数来设置精度，为此我们要编写如下的语句：

```
//把精度设为 3，返回原先的精度
streamsize prec = cout.precision(3);
```

```
cout << "Your final grade is "
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count << endl;

//把精度重新设置成它的原始值
cout.precision(prec);
```

不过，我们还是偏向于使用 setprecision 控制器，因为这样的话，我们可以把设置精度的那一部分程序代码最小化。

3.1.1 检测输入

从概念上来说，在这个程序中，我们惟一感到陌生的一部分是 while 语句的条件。这个条件不明确地使用了 istream 来作为 while 条件的主体：

```
while (cin >> x) { /* …*/ }
```

我们使用这条语句来尝试从 cin 中读数据。如果读操作成功，x 将保存我们刚刚读到的值，同时 while 语句的测试也会成功。如果读取失败（不管是因为输入已读完还是因为遇到了对 x 的类型无效的输入），那么 while 的测试就会失败，而且我们将不能使用 x 的值。

代码的工作原理对我们来说可能会是比较难理解的。我们可以先回忆一下，>>运算符会返回它的左操作数，因此，请求 cin>>x 的值等价于先执行 cin>>x 然后请求 cin 的值。例如，我们可以把一个数据读到 x 中，然后执行下面的语句来测试此动作是否成功：

```
if (cin >> x) { /* … */ }
```

这条语句与下面的语句等价：

```
cin>>x;
if (cin) { /* … */ }
```

如果我们使用 cin >> x 来作为条件，那我们就不仅仅是检验条件；作为副作用，我们还会把一个值读到 x 中。现在，我们所要做的只不过是要理解，在 while 语句中使用 cin 作为条件究竟有何意义。

因为 cin 的类型是 istream，而 istream 是标准库的一部分，所以，为了理解 if（cin）或 while（cin）的意义，我们就必须先理解 istream 的定义。事实证明，这个定义是非常复杂的，因此，我们要到§12.5 才对它进行详细的讨论。不过，就算没有这些细节，我们还是可以从这些语句中了解到很多有用的信息。

我们在第 2 章使用的条件全部都包含有关系运算符，这些条件直接产生了 bool 类型的值。此外，我们还可以使用产生算术类型的值的表达式来作为条件。如果我们在条件中使用了这种表达式，那么算术值就会被转换成一个 bool 类型的值：非零值被转换成 true；零值被转换成 false。就目前来说，我们需要了解的也就是类似的内容，就是类 istream 提供了一个转换

cin 转换成一个可以在条件中使用的值。我们还不知道这个值的类型，但是我们知道这个值可以被转换成布尔值，这个转换所产生的值取决于 istream 对象的内部状态，这个状态会记住最近一次读数据的尝试是否成功。因此，用 cin 来作为条件等价于检测最近一次从 cin 读数据的尝试是否成功。

在以下的几种情况中，我们从一个流读数据的尝试可能会以失败告终：

- 我们可能已经到达了输入文件的结尾。
- 我们碰到的输入有可能会跟我们试图读取的变量的类型不一致，例如，有可能会出现这样的情况：我们想要读的是一个整数，但实际读到的却不是整数。
- 系统可能会在输入装置中检测到一个硬件问题。

对于上面的任意一种情况，结果都是一样的：如果我们使用这个输入流来作为条件，那么条件将会为假。此外，一旦我们不能成功地从流读到数据，那接下来的所有从流读数据的尝试都会以失败告终，除非是我们重新设置了流，在§4.1.3 中我们会了解到重置流的方法。

3.1.2 循环不变式

在理解循环不变式（§2.3.2）的时候我们要特别小心，因为 while 的条件具有副作用。这些副作用会影响到不变式的真实性：如果我们成功地执行了 cin >> x，那么不变式的第一部分（也就是说明了我们已经读到 count 个成绩的那一部分）就会为假。因此，我们必须对我们的分析作一些改动，考虑条件对不变式可能产生的影响。

我们知道，在判定条件之前不变式为真，因此我们可以知道，我们已经读进了 count 个成绩。如果 cin>>x 成功，那么现在我们已经读进了 count+1 个成绩。只要我们对 count 加 1，就可以令不变式的这一部分再次为真。然而，这样做会使不变式的第二部分为假（也就是说明 sum 是头 count 个成绩的总和的那一部分），因为在我们对 count 加了 1 以后，sum 已经是头 count-1 个而不是 count 个成绩的总和了。幸运的是，我们可以执行 sum+=x 从而使不变式的第二部分为真；这样的话，在随后的 while 的循环过程中整个不变式都将为真。

如果条件为假，那就意味着我们的输入尝试失败了，因此我们并没有读到更多的数据，这样的话，这个不变式就仍然为真。结果，在 while 语句结束以后，我们就没有必要再去考虑条件的副作用了。

3.2 用中值代替平均值

到目前为止，我们所编写的程序有一个设计上的缺陷：一旦程序读进了一个家庭作业成绩，它就会马上把这个成绩丢掉。对于平均值的计算来说，这种处理方式是适合的，但是，如果我们想用家庭作业成绩的中值来代替平均值的话，那我们又该如何呢？

查找数值集合的中值的最简单方法是，把数值按递增（或递减）顺序排列起来，

中间的一个——或者，如果数值的个数为偶数，那我们就取最靠近中间的那两个值的平均值。中值经常比平均值更加有用，因为它们不但具有跟平均值一样的作用，而且它们还不会因为一些糟糕的分数而令总成绩下降。

如果稍微考虑一下，我们就会相信，为了计算中值必须从根本上改写我们的程序。为了查找未知数目的数值的中值，我们必须存储每一个数值，直到把它们都读进来为止。为了查找平均值，我们只需要存储数值的个数和使用读到的数据项的总和。平均值就是总和除以个数而得到的结果。

3.2.1 把数据集合存储在向量中

为了计算中值，我们需要读取并存储所有的家庭作业成绩，然后对它们进行排序，最后我们还得取出中间的一个数（或两个）。为了更方便、更高效地完成这个计算过程，我们需要一种方法来做以下工作：

- 存储大量的数值，我们将每次读取一个数值，而且在读数值的时候我们并不会预先知道数值的个数。
- 在读取了所有的数值之后对它们进行排序。
- 高效地获取中间的一个或两个数值。

标准库提供了一种名叫 vector（向量）的类型，我们可以使用这种类型来很轻松地解决所有这些问题。vector 保存了一系列具有特定类型的数值，它的大小可以根据需要增长以容纳添加的数值，还可以让我们高效地访问每一个独立的数值。

现在，让我们开始重新编写计算成绩的程序，为此，我们将把家庭作业成绩存进一个向量中，而不是计算其总和并把这些成绩丢掉。原来的代码版本的形式如下：

```
//原来的程序（摘录）
int count = 0;
double sum = 0;
double x;

//不变式：
//到目前为止，我们已经读到了 count 个家庭作业成绩，
//而且 sum 等于头 count 个成绩的总和
while (cin >> x) {
    ++count;
    sum += x;
}
```

这个循环跟踪了读到的成绩的个数，同时它会不断地计算这些成绩的总和。在读数值的时候，我们要让这两个变量跟数值保持合拍，这样的话，不变式就会变得相当复杂。与之相反，在读数值的时候，如果我们使用向量来存储这些数值的话，不变式就会简单很多：

```
//修正版本（摘录）：
```

```
double x;
vector<double> homework;

//不变式：到目前为止，homework 包含了所有读到的家庭作业成绩
while (cin >> x)
    homework.push_back(x);
```

在这里，我们并没有改变代码的基本结构：它仍然是每次一个地把数值读到 x 中，直到它遇到了文件结束标志或无效输入为止。不过，在处理这些数值的时候，我们采用了与前面不同的方式。

让我们先从 homework 开始，在定义 homework 的时候我们所使用的类型是 vector<double>。向量是一个存储数值集合的容器。在一个向量中的所有数值都具有相同的类型，但是不同的向量可以保存不同类型的对象。无论何时，只要我们定义了一个向量，我们就必须指定向量所保存的数值的类型。根据 homework 的定义我们可以知道，homework 是一个向量，而且这个向量将保存 double 类型的值。

在定义向量类型的时候，我们使用了一种名为**模板类**的语言特征。在第 11 章中，我们将会看到模板类的定义方法。不过，现在对我们来说最重要的是要认识到，我们可以把一个向量和这个向量所保存的特定类型的对象分隔开来。我们在尖括号内指定了对象的类型。例如，vector<double>类型的对象是向量，这些向量保存了 double 类型的对象；vector<string>类型的对象则保存了字符串，等等。

while 循环从标准输入读数值并把这些数值存储在向量中。和原先一样，我们把数据读到 x 中——直到我们碰到文件结束标志或遇到了非 double 类型的输入。我们在 while 语句中用到的新概念是：

```
homework.push_back(x);
```

像 § 1.2 中的 greeting.size()一样，我们可以把 push_back 看作一个成员函数，这个函数被定义成 vector 类型的一部分，而且我们用这个函数来对 homework 对象进行操作。我们调用了这个函数并把 x 传递给它。push_back 函数所做的是添加一个新的元素到向量的末尾。它会给新元素一个值，这个值就是我们传递给它的那个参数。因此，push_back 会把它的参数压进向量的尾部。作为副作用，它会把向量的长度加 1。

因为 push_back 函数所做的操作恰好就是我们想做的，所以我们很容易就可以看出，在调用它的时候，我们的循环不变式将会保持为真。因此很明显，当我们从 while 语句中退出时，将读取了全部的家庭作业成绩并已经把它们存储在 homework 中了——这正是我们想要的。

接下来，我们应该考虑输出。

3.2.2 产生输出

在§3.1 中的原始程序版本中，我们是在输出表达式中计算学生成绩的：

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     <<setprecision(prec) << endl;
```

在这里，final 和 midterm 保存了考试成绩，sum 和 count 则包含了所有家庭作业成绩的总和以及所输入的成绩的个数。

如我们在§3.2.1 中讨论的那样，计算中值的最简单方法是对数据进行排序，然后找出中间的数值或者是两个中间数值的平均值——如果我们的元素为偶数的话。如果我们把计算中值的操作从输出代码中分离出来，就可以让运算变得更加容易理解。

为了找出中值，我们要先留意一下，我们至少要在两个地方获得向量 homework 的长度：一是检查长度是否为 0；二是计算中间元素的位置。如果我们把长度存储在一个局部变量中，就不必两次计算长度了：

```
typedef vector<double>::size_type vec_sz;
vec_sz size = homework.size();
```

vector 类型定义了一个名为 vector<double>::size_type 的类型和一个名为 size 的函数。这些成员的操作方式与 string 类型中的相似：类型 size_type 是一个 unsigned 类型，它必须大到可以用来保存向量可能容纳的最大长度；size()则返回了一个 size_type 类型的值，我们用这个值来表示向量中的元素个数。

因为需要在两个地方获知向量的长度，所以我们会用一个局部变量来记住这个值。不同的系统环境使用了不同的类型来表示长度，因此，如果我们希望保持系统环境的独立性，就不能直接编写恰当的类型。由于这个原因，我们应该养成使用库定义的 size_type 来表示容器的长度的良好的编程习惯——在命名 size 的类型的时候，我们就是这样做的。

在这个例子中，对这种类型的编写和阅读都非常不方便。为了简化程序，我们使用了一种这之前没有见过的语言工具，这种工具叫做 **typedef**。如果我们的定义包含有单词 typedef，那就表示，我们所定义的名称是特定类型的一个替代名，而不是这种类型的一个变量。这样的话，因为我们的定义包括了 typedef，所以它就把名称 vec_sz 定义成了 vector<double>::size_type 的一个替代名。通过 typedef 定义的名称会具有跟其他所有名称一样的作用域。也就是说，我们可以把 vec_sz 作为 size_type 的替代名——一直到我们到达了当前作用域的末尾。

如果我们知道了如何给 homework.size()的返回值的类型命名，那我们就可以把这个值存储在一个同一类型的、名为 size 的局部变量中。值得注意的是，即使我们是出于两个不同的目的而使用 size 这个名称，也不会因此而导致冲突或意义上的含混。计算一个向量大小的惟一办法是调用 size 函数，在我们调用的时候，size 函数名出现在圆点“.”的右边，而该向量名就在圆点的左边。也就是说，被定义为局部变量的 size 跟那个对向量进行操作的 size 是处于不同的作用域内的。由于这两个名称所处的作用域不同，所以编译器（和程序员）能够判别出哪个 size 才是他们所需要的。

对我们来说，查找一个空数据集的中值是毫无意义的，因此我们下一步的工作是验证我们确实拥有了一些数据：

```
if (size == 0) {
    cout << endl << "You must enter your grades. "
                 "Please try again." << endl;
    return 1;
}
```

通过检查 size 是否为 0，我们就能够检测到上面的这种情形。如果 size 为 0，那么，最明智的做法是发出错误提示并终止程序。为此，我们返回 1 来表示失败。正如我们在第 0 章所讨论的那样，系统假定，如果 main 函数返回 0，则程序运行成功。返回其他的任何值都会表示一个系统环境自定义的意义，不过，大部分的系统环境都把所有的非 0 值看作是失败。

如果我们已经验证了我们的确有数据，那么，现在我们就可以开始计算中值了。第一步要做的是对数据进行排序，为此，我们调用了一个库函数：

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

sort 函数定义在头文件<algorithm>中，它把容器中的数据重新排序成非递减序列。我们之所以用非递减而不用递增是因为，容器中的某些数据元素可能会与其他元素相等。

sort 函数的参数指定了被排序的元素的范围。vector 类为这个用法提供了两个成员函数 begin 和 end。在§5.2.2 中我们会对 begin 和 end 函数进行更多的讨论，不过现在最重要的是要知道 homework.begin()指示了名为 homework 的向量的第一个元素；同理，homework.end()则指向紧跟在 homework 的最后一个元素之后的位置。

sort 函数巧妙地完成了它的任务：它仅仅是调换了原容器中的元素值的相对顺序，而不是创建一个新的容器来存储排序后的结果。

一旦我们完成了对 homework 的排序，我们就要找出它的一个或两个中间元素：

```
vec_sz mid = size/2;
double median;
median = size % 2 == 0 ? (homework[mid] + homework[mid-1]) / 2
                       : homework[mid];
```

在程序段的开始我们把 size 除以 2，这样我们就确定了向量的中间位置。如果元素个数为偶数，则除法的结果是精确的；如果元素个数为奇数，那么结果应该是小于商的所有整数中最大的那个。

如何计算中值依赖于元素个数的奇偶性。如果是偶数，则中值是最靠近中间位置的两个元素的平均值。否则，在中间会有一个元素，它的值就是中值。

我们在那个对 median 赋值的表达式中使用了两个新的运算符：**取模**运算符%和一个**条件**运算符，而这个**条件**运算符经常被称作？：运算符。

取模运算符%返回一个由它的左操作数除以右操作数而得到的余数。如果元素个数

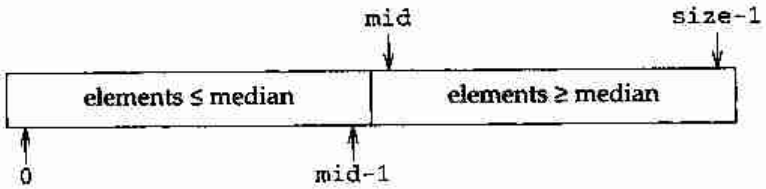
所得到的余数为 0，那么，程序已经读进了偶数个元素。

条件运算符则是简单的 if-then-else 表达式的简写。首先，它计算在运算符？部分之前的条件表达式并获得一个布尔值。如果条件产生 true 值，那么结果就是紧跟在后面的在“?”和“:”之间的表达式的值；否则，结果就是“:”之后表达式的值。因此，如果我们读进的元素个数为偶数，我们就把中值设为中间两个元素的平均值。如果个数为奇数，那么我们把中值设为 homework[mid]。与&&和||运算符类似，？：运算符首先计算它的最左操作数。以这个计算所得到的结果值为基础，接下来它会计算且只计算其他操作数中的一个。

对 homework[mid]和 homework[mid – 1]的引用为我们展示了访问向量元素的一种方法。每个向量中的任意一个元素都有一个被称为**索引**的整数与之相联。例如，homework[mid]是 homework 中索引为 mid 的元素。跟读者在§2.6 中所看到的一样，homework 向量的第一个元素是 homework[0]，最后一个元素则是 homework[size – 1]。

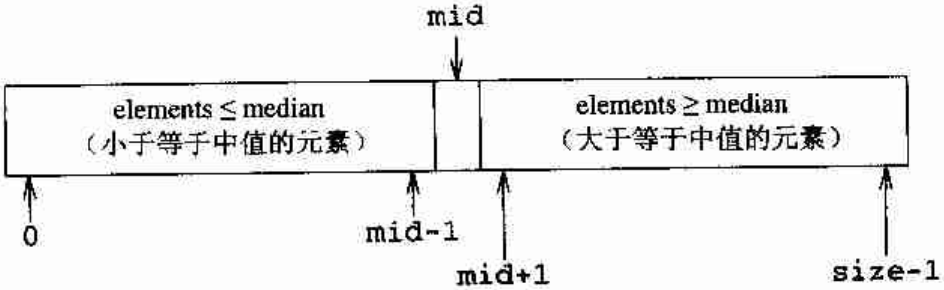
每一个元素本身都是一个（未命名）的对象，这个对象的类型跟存储在容器中的类型相同。因此，homework[mid]是一个 double 类型的对象，我们可以对这个对象做 double 类型所支持的任何操作。特别地，我们可以把两个元素相加，同时还可以把这个加法的结果除以 2 以获得两个对象的平均值。

一旦懂得了如何去访问 homework 中的元素，我们就可以理解中值运算是怎样进行的了。我们首先假定 size 为偶数，这样的话，mid 的值就等于 size/2。那么，在中间位置的左右两边都一定会有且只有 mid 个元素。



我们知道，homework 的每一半都刚好有 mid 个元素，这样我们就能很容易地看出，最靠近中间位置的那两个元素的索引是 mid – 1 和 mid，而中值就是这两个元素的平均值。

如果元素的个数是奇数，那么截取除法的结果之后所得到的 mid 实际就是(size – 1)/2。这样的话，我们就可以把 homework 向量看作是各有 mid 个元素的两个片段。这两个片段是被中间的一个元素分割开的，这个元素就是中值：



在上面的每一种情况中，中值的计算都取决于在只知道元素索引的情况下我们对这个元素进行访问的能力。

如果我们算出了中值，那么其余所要做的只不过是计算并输出最后的成绩：

```
streamsize  prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * median
     << setprecision(prec) << endl;
```

虽然最后的这段程序做的工作要比§3.1 中的程序段所做的多得多，但是，这段程序的复杂度并没有增加多少。特别地，即使我们的 homework 向量会根据需要而扩充容量——这样它就可以容纳得下数量在学生的可接受范围之内的家庭作业的成绩，我们的程序也不需要考虑如何获得内存来存储所有的这些成绩，因为标准库已经自动为我们做了所有的这些工作了。

下面是完整的程序。程序中我们惟一没有提及过的就是那些#include 指令、相应的 using 声明以及一些注释：

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
#include <vector>

using std::cin;          using std::sort;
using std::cout;         using std::streamsize;
using std::endl;         using std::string;
using std::precision;    using std::vector;
using std::setprecision;

int main()
{
    //请求输入并读入学生的姓名
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    //请求输入并读入期中和期末成绩
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    //请求输入家庭作业成绩
    cout << "Enter all your homework grades, "
```

```
            "followed by end-of-file: ";

    vector<double> homework;
    double x;

    //不变式: homework 包含了所有的家庭作业成绩
    while (cin >> x)
        homework.push_back(x);

    //检查 homework 是否为空
    typedef vector<double>::size_type vec_sz;
    vec_sz size = homework.size();
    if (size == 0) {
        cout << endl << "You must enter your grades. "
                     << "Please try again." << endl;
        return 1;
    }

    //对成绩进行排序
    sort(homework.begin(), homework.end());

    //计算家庭作业成绩的中值
    vec_sz mid = size/2;
    double median;
    median = (size % 2 == 0) ? (homework[mid] + homework[mid-1]) / 2
             : homework[mid];

    //计算并输出总成绩
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * median
         << setprecision(prec) << endl;

    return 0;
}
```

3.2.3 一些更为深入的观察

在上面的代码中，有几个地方是值得我们特别留意的。首先，我们需要进一步理解，在 homework 为空的情况下我们为什么要退出程序。从逻辑上来说，取一个空的数值集合中值的这个操作是未经定义的——我们根本不知道它到底有什么意义。再者，退出程序通常都是一个正确的选择：如果我们不知道要做什么，那么我们还是退出比较好。但是，我们还是有必要去了解一下，如果我们继续往下执行的话那将会发生什么。如果输入为空而且我们忘记了检查是否读进了至少一个数据，那么计算中值的那一部分代码将会以失败告终。为什么呢？

如果我们并没有读到元素，那么 homework.size()（也就是 size）将会等于 0。同样地，mid 也会等于 0。当我们执行到 homework[mid]的时候，我们将会查看 homework 的第一个元素（也就是索引为 0 的那个）。但问题是，homework 中并没有元素存在！如果我们执行 homework[0]，那我们就根本不用猜测我们得到了什么——因为这样做会是徒劳无功的。向量不会检查索引是不是在其值域中取值，这样的检查是由用户来完成的。

我们观察到的另外一点很重要的信息是，像所有的标准库长度类型一样，vector<double>::size_type 是**无符号整数类型**。这样的类型是根本不能用来存储负数值的；相反，它们所存储的值以 2^n 为模（n 的大小取决于不同的系统环境）。例如，在程序中我们永远不会检查 homework.size() < 0 是否成立，因为这个不等式总是产生 false 值。

此外，每当普通整数和无符号整数在表达式中结合在一起时，普通整数就要被转换成无符号整数。因此，诸如 homework.size() - 100 这样的表达式将会产生无符号的结果，这也意味着结果不能小于 0——即使是 homework.size() <100。

最后，值得注意的是，事实上我们程序的运行性能是相当好的——尽管 vector<double>类型的对象是根据需要而增长以容纳其输入，而不是从一开始就分配了正确的长度的。

我们之所以对程序的性能充满信心，是因为 C++标准对库的执行性能的要求很高。库不仅仅要符合行为方面的规定，它还要达到规范化的性能目标。所有符合标准的 C++系统环境都必须：

- 往 vector 后面添加大量元素时，其性能应该不会随着元素个数的增加而成比例地恶化
- 以不低于 nlog(n)的速度（即，运行时间要小于等于 nlog(n)）实现排序算法，在这里，n 是待排序的元素的个数

这样的话，我们就确保了整个程序能够在任何符合规定的系统环境上成功运行，而且运行时间会少于等于 nlog(n)。实际上，在设计标准库的时候人们就已经把大量的注意力放在了性能方面。C++是被设计来为特别注重性能的应用服务的，同时在库中也处处体现了对速度的强调。

3.3 小结

如果在定义的时候我们不给局部变量指定明确的初始值，那么它就被缺省初始化。内部类型的缺省初始化意味着值是未经定义的。未定义的数值只能用作赋值运算的左侧操作数。

类型定义：

typedef type name;　　把 name 定义为 type 的替代名。

vector 类型是在<vector>中定义的，它是库中的一种容器形式类型，我们用它来保存一系列特定类型的值。向量可以动态地增长。对向量的一些重要操作如下：

vector<T>::size_type　一种类型，它确保了能够保存可能存在的最大向量中的所有元素

v.begin( )　　返回一个数值，这个数值指示了 v 的第一个元素。

v.end( )	返回一个数值，这个数值指示了紧跟在 v 的最后一个元素之后的位置。
vector<T> v;	创建了一个空的向量，这个向量可以保存 T 类型的元素。
v.push_back(e)	给向量添加一个元素，这个元素的初始值为 e。
v[i]	返回存储在位置 i 中的值。
v.size()	返回 v 的元素个数。
其他库工具：	
sort(b, e)	把在区间[b,e)中定义的元素重新排列成非递减序列，这个函数是在<algorithm>中定义的。
max(e1,e2)	返回表达式 e1 和 e2 中的较大者；e1 和 e2 必须具有完全相同的类型。这个函数是在<algorethm>中定义的。
while(cin>>x)	把一个适当类型的值读到 x 中并检查流的状态。如果流处于错误状态，那么检查失败；否则，检查成功，while 的循环体被执行。
s.precision(n)	为了后继输出而把流 s 的精度设置为 n（如果省略 n，那么 s 的精度不变）。返回原先的精度。
setprecision(n)	返回一个值，如果是对输出流 s 编写这个函数，那么这个操作具有调用 s.precision(n)的效果。定义在<iomanip>中。
streamsize	是 setprecision 期望并且由 precision 返回的值的类型。定义在<ios>中。

习题

3-0 编译、运行并测试本章中的程序。

3-1 假设我们希望找出一个数值集合的中值；同时假定到目前为止，我们已经读进一些数值了，而且不清楚还要再读进多少个值。证明：我们不能丢掉已经读到的任何值。提示：一个可行的证明策略是，先假定我们可以丢掉一个值，然后找出我们的集合中未读的（也就是未知的）那部分数值，要求这些数值将会使中值恰好就是我们丢掉的那个值。

3-2 把一个整数集合分为个数相等的四部分，而且第一部分含有的整数值比其他各部分的都大，第二部分的值比第一部分的小比其他两部分的大，剩下的两部分则以此类推。按照上面的要求，编写一个程序来计算并且打印这四部分。

3-3 编写一个程序来计算在它的输入中每个不同的单词所出现的次数。

3-4 编写一个程序来报告它的输入中最长以及最短的字符串的长度。

3-5 编写一个程序来同时跟踪 n 个学生的成绩。要求程序能够保持两个向量的同步：第一个应保存学生的姓名；第二个保存总成绩，而这个总成绩能根据读到的输入来计算。读者应假定家庭作业成绩的个数是固定的。我们将在§4.1.3 中看到如何处理与学生姓名混合的数量可变的成绩。

第 4 章

组织程序和数据

虽然在§3.2.2 中的程序比我们预想的要大，但是如果没有向量、字符串和排序算法的话那它将会更加庞大。这些库工具跟我们使用过的其他的工具一样，都有着几个同样的特性：它们中的每一个

- 都能解决某些特定类型的问题
- 与其他的大多数工具都相互独立
- 都具有一个名称

我们自己的程序具备了这些特性中的第一个，但是它们并不具备另外的两个。这对于小程序来说是件好事，不过当我们开始动手解决更为困难的问题时，就会发现，除非把我们的解决方案分割成相互独立的具有名称的各个部分，否则它们将会变得无法控制。

跟大多数程序设计语言一样，C++提供了两种基本的方法来让我们组织大型的程序：函数（有时候被称为子程序）和数据结构。另外，C++还可以让程序员把函数和数据结构结合在一个叫做类的概念中——我们将会在第 9 章中就这一点展开讨论。

就算我们懂得了如何使用函数和数据结构来组织运算，还是需要找到一个方法来把程序划分为我们可以独立编译且在编译后能组合在一起的文件。在本章的最后一部分我们将说明 C++是如何支持独立编译的。

4.1 组织计算

让我们先编写一个函数来计算一个学生的总成绩，这个总成绩是根据期中、期末考试成绩和家庭作业总体成绩计算出来的。我们将假定，我们已经从个别的家庭作业成绩中算出了家庭作业的总体成绩，而且我们计算得到的总体成绩是平均值或中值。除了这个假定之外，这个函数还将使用我们在前面一直沿用的那个策略：家庭作业和期末考试成绩各占总成绩的 40%，而期中考试则占剩下来的 20%。

每当我们要（或者可能要）在几个地方进行同一个计算的时候，我们都应考虑把这个计算放在同一个函数里面。显而易见，之所以要这样做是因为：我们可以用函数来明确地代替重复的计算。使用函数不但可以减轻我们程序的工作量，而且这样做还可以让我们在有必要时更为方便地改变运算的过程。例如，假定我们想改变那个计算成绩的策略。如果我们为了查

成绩的那一部分代码而不得不从头到尾的搜索我们之前编写的每一个程序的话，那我们可能很快就会泄气了。

对这样的计算使用函数还会有一个更为微妙的好处：函数都具有名称。如果我们对计算命名，那我们就能以更为抽象的方式去思考它——我们能更为细致地去考虑它是做什么的，而对它是如何去做的则考虑得较少。如果我们能够识别出我们问题中的重要部分并创建与这些部分相对应的具名程序段，那我们的程序就会变得更容易理解，而我们的问题的求解难度也会相应降低。

根据我们的策略，我们可以编写如下的程序来计算成绩：

```
//根据学生的期中考试、期末考试以及家庭作业成绩来计算总成绩
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

到目前为止，我们定义过的所有函数都被命名为 main。我们可以类似地定义其他的绝大多数的函数。在定义时，我们先要指定它的返回值类型，类型后面紧跟着函数名，函数名的后面是用（）括起的**参数列表**，函数最后的一部分则是用{}括起的函数体。对于返回值代表其他函数这种情况而言，这些规则是更加复杂的。详情请参阅§A.1.2。

在这个例子中，函数的参数是 midterm、final 以及 homework，每一个参数的类型都是 double。这些参数是作为函数中的局部变量而被使用的。这就意味着，在调用函数时它们会被创建，并且在函数返回时会被销毁。

跟其他的任何变量一样，在使用参数之前我们必须先定义它们。与其他变量不同，对它们进行定义并不表示会马上创建它们，它们只是在函数被调用时才会创建。因此，无论何时，只要我们调用了函数，我们就必须提供相应的参数。这些参数是被用来在函数开始执行时对参数进行初始化的。例如，在§3.1 中我们编写了下面的语句来计算一个成绩：

```
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

利用 grade 函数，我们就可以把上面的语句改写成：

```
cout << "Your final grade is " << setprecision(3)
     << grade(midterm, final, sum / count)
     << setprecision(prec) << endl;
```

我们提供的参数不但要对应于我们调用的函数所具有的参数，而且这些参数的顺序还必须跟参数的相应顺序相同。因此，如果我们要调用函数 grade，那么第一个参数必须是期中考试成绩，第二个必须是期末成绩，而第三个则必须是家庭作业成绩。

参数不仅仅是变量，它们还可以是诸如 sum/count 这样的表达式。一般而言，每个

是用来对相应的参数进行初始化的，参数在初始化之后就可以像普通的局部变量一样的在函数中使用了。例如，如果我们调用 grade(midterm, final, sum/count)，那么 grade 函数的参数并不是直接指向参数本身，实际上，这些参数的初始值是相应参数值的复制。这个行为经常被称为**按值调用（call by value）**，这是因为参数获得的只是参数值的一个复制。

4.1.1 查找中值

在§3.2.2 中我们还解决了中值的查找问题。不难想像，在其他情况中，我们还可能要解决向量中值的查找问题。在§8.1.1 中我们将会看到我们是如何定义一个通用函数来容纳处理任意类型值的向量。而现在，我们将首先把注意力集中到 vector<double>类型上。

为了编写我们的函数，让我们先来看一下在§3.2.2 中那一部分计算中值的程序，在此我们做了一些改动：

```
//计算一个 vector<double>类型的变量的中值
//值得注意的是，调用函数时整个 vector 参数都会被复制
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;

    vec_sz size = vec.size();
    if (size == 0)
        throw domain_error("median of an empty vector");

    sort(vec.begin(), vec.end());

    vec_sz mid = size/2;

    return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}
```

其中的一个改动是，我们将向量命名为 vec 而不再是把它称作 homework。毕竟，我们的函数可以计算任何事物（而不仅仅是家庭作业）的中值。同时我们还去掉了变量 median，这是因为我们已经不再需要它了：我们一算出了中值就马上可以返回它。不过我们仍然使用了变量 size 和 mid，只是现在它们变成了 median 函数的局部变量。这样的话，在其他地方这两个变量就是不可访问（并且是不相关）的。调用 median 函数时这些变量将会被创建，从函数返回时它们则会被销毁。我们把 vec_sz 定义成一个局部的类型名称，因为我们不希望和任何出于其他目的而使用这个名称的人发生冲突。

函数中最为显著的改动是，我们对空向量采取了不同的处理办法。从§3.2.2 中我们可以知道，我们将必须向任何正在使用我们的程序的人发出错误提示，同时我们还可以从中得知是谁在使用我们的程序以及什么样的错误提示才是有意义的。在这个经修正的版本中，我们并不知道是谁在使用它或者是出于何种目的而使用它，因此，我们需要找到一种办法来发出更

的错误提示。而这种办法就是，如果向量为空，那我们就抛出一个**异常**。

如果一个程序抛出了一个异常，那么这个程序就会在抛出异常的地方终止执行并转移到程序的另一部分，并向这部分提供一个**异常对象**，异常对象中含有调用程序可以用来处理异常的信息。

通常，在待传递的信息中最重要的一部分只不过是这样的一个事实：一个异常被抛出了。一般来说，这个事实和异常对象的类型结合起来就足以让调用程序知道应该采取什么措施了。在这个特殊的例子中，我们抛出的异常是 domain_error。这是在头文件<stdexcept>中定义的一种类型，它会向我们报告，函数参数的取值是函数所不能接受的。在我们创建一个待抛出的 domain_error 对象时，我们可以给它一个字符串来描述错误信息。正如我们将在§4.2.3 中看到的那样，捕获异常的程序可以在一个诊断信息中使用这个字符串。

还有一个关于函数行为的细节对于我们的理解是很重要的。在我们调用一个函数的时候，我们可以把参数看作是初始值等于参数的局部变量。这样的话，我们就能看到，在调用一个函数的时候，参数同时也会被复制到参数中。特别地，如果我们调用 median 函数，那么我们用做参数的那个向量就会被复制到 vec 中。

就 median 而言，把参数复制到参数中是非常有用的，但这样做会花费较多的时间——因为 median 函数会通过调用 sort 从而改变它的参数值。如果函数使用复制参数的方式，那么 sort 所作的改变就不会反馈到调用程序中了。这个做法是很有意义的，因为我们获取向量的中值时不应该连带改变向量的本身。

4.1.2 重新制定计算成绩的策略

§4.1 中的 grade 函数假定我们已经知道了学生的家庭作业总体成绩，而不仅仅是知道个别的家庭作业成绩。如何获取这个成绩是我们策略的一部分：在§3.1 中我们使用平均值，而在§3.2.2 中则使用了中值。因此，像我们在§4.1 中所做的那样，我们可能会希望以函数的形式来表示我们这一部分的计算成绩的策略：

```
//根据期中、期末考试成绩和保存家庭作业的向量来计算学生的总成绩
//这个函数不用复制它的参数，因为 median 已经为我们完成了这个工作
double grade(double midterm, double final, const vector<double>& hw)
{
    if (hw.size() == 0)
        throw domain_error("student has done no homework");
    return grade(midterm, final, median(hw));
}
```

在这个函数中，有三个地方特别让人感兴趣。

第一个是，我们为第三个参数指定的类型是 const vector<double>&。这个类型经常被称为“对参数类型为 double 的向量常量的引用”，或者通俗点，我们把它称为“双精度向量常量引用”。我们说名称是一个引用是指，这个名称是一个特定对象的另一个名称。例如，如

编写了以下的语句：

```
vector<double> homework;
vector<double>& hw = homework;        //hw是homework的一个替代名
```

我们说，hw 是 homework 的另一个名称。从现在起，我们对 hw 所做的任何动作都等价于对 homework 做同样的动作，反过来也是一样。在下面的语句中再增加一个 const：

```
//chw是homework的一个只读的替代名
const vector<double>& chw = homework;
```

这条语句仍然是表示 chw 是 homework 的另一个替代名，但是 const 确保了我们将不会对 chw 做任何可能改变它的值的动作。

因为一个引用是原先的变量的另一个名称，所以不存在诸如一个引用的引用这样的说法。定义一个引用的引用跟定义原来对象的引用的效果是一样的。例如，如果我们编写了：

```
//hw1和chw1是homework的替代名，chw1是只读的
vector<double>& hw1 = hw;
const vector<double>& chw1 = chw;
```

那么 hw1 跟 hw 一样都是 homework 的另一个名称；而 chw1 则跟 chw 一样是 homework 的不允许写访问的另一个名称。

如果我们定义一个非常量引用（也就是一个允许写访问的引用）我们就不能让它指向一个常量对象或常量引用，因为这样做表示了，我们将要做 const 所不允许做的动作。因此，我们不能编写：

```
vector<double>& hw2 = chw;            //错误：请求了对chw的写访问
```

这是因为，我们之前已经保证了不会去改变 chw 的值。

同样地，如果我们说一个参数的类型是 const vector<double>&，那么，实际上我们就是在请求系统环境给予我们对关联变量的直接访问权而不用我们去复制它，同时这样做还可以确保我们将不会改变参数的值（否则参数的值也就跟着改变了）。因为这个参数是对常量的一个引用，所以我们既可以为常量也可以为非常量向量调用这个函数。又因为参数是一个引用，所以我们就可以避免复制参数的额外开销。

grade 函数中特别让人感兴趣的第二个地方是，跟§4.1 中的那个函数一样，这个函数的函数名是 grade——尽管它会调用另一个 grade 函数。我们能够让几个函数具有同样的函数名的这个概念叫做**重载**，它在许多的 C++程序中都会经常出现。就算我们有两个函数的函数名是相同的，这也不会导致意义上的含糊，因为不管我们在什么时候调用 grade，我们都会提供一个参数列表，系统环境能够根据第二个参数的类型来辨别我们所指的是哪个函数。

第三点是，我们会检查 homework.size()是否为 0——尽管我们知道 median 会为我们完成这个任务。这样做的原因是，如果 median 发现我们是在请求一个空向量的中值话，它[illegible]出一个包含了信息“median of an empty vector”的异常。对那些正在计算学生成绩的

这条信息并不是直接有用的。因此，我们抛出了自己的异常，我们希望它能给用户更多的提示，让用户更好地了解错误的所在。

4.1.3 读家庭作业成绩

另一个我们必须在几种情况中解决的问题是：我们应该如何把家庭作业成绩读进一个向量中呢？

在设计这样一个函数的行为时，我们会遇到一个问题：它要求一次返回两个值。一个值当然就是它读到的家庭作业成绩；而另一个则指示输入尝试是否成功。

我们找不到一个直接的方法来从函数中返回多于一个的值。一个间接的处理办法是，给函数一个参数，这个参数是对一个对象的引用，而函数的其中一个结果值会放置在这个参数里面。这个策略对于读输入的函数来说是很常见的，因此我们不妨使用它。这样的话，我们的函数如下所示：

```
//读一个输入流，把家庭作业成绩读进一个 vector<double>类型的向量中
istream& read_hw(istream& in, vector<double>& hw) {
    //这一部分代码有待补充
    return in;
}
```

在§4.1.2 中，我们看到的那个程序具有一个类型为 const vector<double>&的参数；而现在我们则去掉了 const。一个不含 const 的引用参数通常表示我们可以修改作为函数参数的对象的值。例如，如果我们执行

```
vector<double> homework;
read_hw(cin, homework);
```

那么实际上，read_hw 的第二个参数是一个引用。根据这个事实我们就可以预料到，对 read_hw 的调用将会改变 homework 的值。

因为我们希望函数能够修改它的参数的值，所以在调用这个函数的时候，我们不能在参数列表中使用任何的表达式。相反，我们必须传递一个**左值**参数给引用参数。左值是一个用来指示非临时对象的值。例如，如果一个变量是左值，那么同时它也可以是一个引用或者说可以是返回一个引用的函数的调用结果。一个产生算术值的表达式，例如 sum/count，并不是左值。

read_hw 函数的两个参数都是引用，这是因为，我们希望用这个函数来改变这两个参数的状态。我们并不了解 cin 的细节，但可以假定，库是把它定义成一个数据结构的。在这种数据结构中存储了库需要用来了解我们的输入文件状态的所有东西。从标准输入文件读输入会改变文件的状态，因此，从逻辑上说，它同时也会改变 cin 的值。

值得注意的是，read_hw 会返回 in。此外，它是以引用的形式来完成这个动作的。实际上，这是表示我们指定了一个对象，而我们将不会在函数调用时对此对象进行复制；同时我们还会在函数返回时返回同一对象——即返回时也不复制对象。由于函数返回的是流，所以我

使用以下的语句：

```
if (read_hw(cin, homework)) { /* … */ }
```

来作为下面的语句的简写：

```
read_hw(cin, homework);
if (cin) { /* … */ }
```

现在，我们可以着手考虑如何去读家庭作业成绩了。很明显，我们希望把现有的所有成绩都读进来。这样的话，看起来我们惟一能做的就是编写下面的语句了：

```
//第一个尝试——并不完全正确的一个尝试
double x;
while (in >> x)
    hw.push_back(x);
```

这个策略并不是十分的有效，我们有两个原因可以解释这一点。

第一个原因是，hw 并不是由我们定义的——我们的调用程序已经为我们定义好了。因为我们没有对它进行定义，所以我们就不知道里面是不是已经有了某些数据。我们所知道的只不过是，我们的调用程序可能正在使用我们的函数来处理许多学生的家庭作业成绩。在这种情况下，hw 可能会含有先前的学生成绩。为了解决这个问题，我们可以在开始我们的工作之前先调用 hw.clear()。

我们的策略会失败的第二个原因是，我们并不十分明确应在何时停止。我们可以维持读成绩的动作直到不能继续这样做为止，但是在这个时刻我们又会有一个问题。有两个原因可以说明我们不能继续读进一个成绩：我们可能已经到达了文件结尾；或者我们所碰到的不是成绩。

在第一种情况中，我们的调用程序将会设想我们到达了文件结尾。这个设想是成立的，但它会引起误解，这是因为文件结尾标记只是在我们成功地读入了所有的数据后才有可能出现。一般来说，一个文件结尾标记表示的是输入尝试失败了。

第二种情况，如果我们碰到的并不是成绩，那么库就会把输入流标记成**失败状态**。这就意味着，后面的输入请求将会以失败告终，就好像我们到达了文件结尾一样。因此，我们的调用程序将认为输入数据出现了问题，而惟一的问题只可能是，跟在最后的一个家庭作业成绩后面的数据并不是家庭作业成绩。

对于上面的任一种情况，我们都可以“假装”我们从未看到过跟在最后一个家庭作业成绩后面的输入数据。这样的“假装”是很容易实现的：如果我们到达了文件结尾，那就不用再读入额外的输入了；如果我们碰到的不是成绩，库就会略过它不读而直接转入下一次的输入尝试。因此，所有我们要做的只不过是让库忽略任何可能导致输入尝试失败的情况——如果这些情况是到达了文件结尾或碰到了无效输入的话。为此我们要调用 in.clear()来清除 in 内部的错误状态，这样就会让库忽略失败情况而继续输入了。

我们还需要考虑另外的一个细节：有时甚至是在试图读入第一个家庭作业成绩之前

就有可能已经读尽了所有的输入或碰到了一个出错的情况。这样的话，我们就必须让输入流不受干涉。否则，我们就有可能在未来的某个时刻尝试读那些不存在的输入。

下面是完整的 read_hw 函数：

```
//从输入流中将家庭作业的成绩读入到一个 vector<double>中
istream read_hw(istream& in, vector<double>& hw)
{
    if (in) {
        //清除原先的内容
        hw.clear();

        //读家庭作业成绩
        double x;
        while (in >> x)
            hw.push_back(x);

        //清除流以使输入动作对下一个学生有效
        in.clear();
    }
    return in;
}
```

值得注意的是，clear 成员在为 istream 服务时所表现出来的行为特性跟它在为向量对象服务时的是完全不同的。对于 istream 对象，它清除了所有的错误标记以使输入动作可以继续；而对于向量对象，它删除了向量中可能已经含有的全部内容，这样就会让我们再次拥有一个空的向量。

4.1.4 三种函数参数

在这里，我们不妨暂时停止我们当前的讨论而转去观察一下下面的这个事实：我们已经定义了三个对家庭作业向量进行操作的函数，这三个函数分别是 median、grade 以及 read_hw。这些函数中的每一个都会出于某种目的而去处理对应的参数，同时这三个函数所采取的处理方法都是互相不同的。

median 函数(§4.1.1)有一个 vector<double>类型的参数。因此，就算对应的参数可能是一个巨型的向量，对这个函数的调用还是会导致此参数被复制。尽管效率不高，但是对于 median 来说，vector<double>依然是一个正确的参数类型，这是因为这个类型确保了在不修改向量的情况下我们可以获得向量的中值。median 会对它的参数进行排序。如果它不复制其参数的话，那么我们调用 median(homework)的时候就会改变 homework 的值。

grade 函数具有一个家庭作业向量(§4.1.2)，这个函数有一个 const vector<double>&类型的参数。在这个类型中，&让系统环境不用复制对应的参数，同时 const 确保了程序将不会改变参数的值。使用这样的参数是提高程序效率的一种重要的手段。无论何时，函数都不应

参数的值，而且对属于诸如 vector 或 string 这样类型的参数值的复制可能会耗费比较多的时间。一般而言，我们并没有必要为了诸如 int 或 double 这样简单的内部类型的参数而去使用 const 引用。对这类小对象的复制通常都是非常快的，因此在按值调用它们的时候，复制动作就算会有额外的开销，也只是很小的一部分。

read_hw 函数有一个类型为 vector<double>&的参数——在这里并没有 const。同样地，&让系统环境直接地把参数和参数连接起来。这样我们就可以避免对参数的复制了。不过在这里，我们之所以要避免复制操作是因为函数要改变参数的值。

与非常量引用参数对应的参数必须是左值——也就是说，它们必须是非临时对象。按值传递或与一个常量引用连接在一起的参数可以取任何值。例如，假定我们有一个返回空向量的函数：

```
vector<double> emptyvec()
{
    vector<double> v;            //没有元素
    return v;
}
```

我们可以调用这个函数并且使用其结果来作为在§4.1.2 中的第二个函数的一个参数：

```
grade(midterm, final, emptyvec());
```

在运行的时候，grade 函数会马上抛出一个异常，这是因为它的参数为空。然而，从语法上看，这样的调用完全是合法的。

在我们调用 read_hw 的时候，它的两个参数都必须是左值，这是因为它的两个参数都是非常量引用。如果我们给 read_hw 一个并不是左值的向量

```
read_hw(cin, emptyvec());        //错误：emptyvec()不是左值
```

编译器就会提示出错，因为我们在调用 emptyvec 时所创建的那个未命名的向量将会在 read_hw 返回时立即消失。如果我们被允许进行这样的调用，那结果就是，我们把输入存进了一个我们无法访问的对象中。

4.1.5 使用函数来计算学生的成绩

我们编写所有的这些函数的惟一目的就是用它们来解决问题。例如，我们可以用它们来重写在 3.2.2 中的那个计算成绩的程序：

```
// include 指令和对库工具的 using 声明
// §4.1.1 中的 median 函数代码
// §4.1 中的 grade(double, double, double)函数代码
// §4.1.2 中的 grade(double, double, const vector<double>&)函数代码
// §4.1.3 中的 read_hw(istream&, vector<double>&)函数代码
```

```
int main()
{
    // 请求并读入学生姓名
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    // 请求并读入期中和期末考试的成绩
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // 请求用户输入家庭作业成绩
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    vector<double> homework;

    // 读入家庭作业成绩
    read_hw(cin, homework);

    // 如果可以的话，计算生成总成绩
    try {
        double final_grade = grade(midterm, final, homework);
        streamsize prec = cout.setprecision();
        cout << "Your final grade is " << setprecision(3)
             << final_grade << setprecision(prec) << endl;
    } catch (domain_error) {
        cout << endl << "You must enter your grades. "
             << "Please try again." << endl;
        return 1;
    }

    return 0;
}
```

与较早前的版本相比，我们在这里采用了不同的方法来读取家庭作业成绩并计算和输出结果。

在请求用户输入家庭作业成绩之后，我们就调用 read_hw 函数来读数据。read_hw 内部的 while 语句会重复地读家庭作业成绩——直到我们到达了文件结尾或者是碰到了一个对 double 类型来说无效的数值。

在这个例子中，最重要的新概念是那条 try 语句。它尝试执行在{}中紧跟在 **try** 关键字之后的语句。如果在这些语句中的任何地方发生了一个 domain_error 异常，那么它就会使

这些语句，然后转去执行另外的一系列用{}括起的语句。这些语句是 **catch（捕获）子句**的一部分，catch 子句从关键字 **catch** 开始，它指示了所捕获的异常的类型。

如果在 try 和 catch 之间的语句没有引发异常而正常结束的话，那么程序就会跳过全部的 catch 子句从而继续执行下一条语句——在这个例子中，这条语句是 return 0;。

每当编写一条 try 语句的时候，我们都必须认真地考虑副作用及副作用发生的时刻。我们必须假定在 try 和 catch 之间的任何东西都有可能会引发异常。如果是这样的话，在异常发生之后，所有本来应该执行的运算都会被跳过。我们有必要意识到下面的这个事实：一个在时间上本来是跟在异常之后执行的运算，在程序正文中是没有必要位于异常发生的地点之后的。

例如，假设我们以更为简洁的形式把输出块语句改写成：

```
//这个例子不能正常工作
try {
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
        << grade(midterm, final, homework) << setprecision(prec);
} ...
```

在这个重写的代码中的问题是，虽然系统环境需要从左到右地执行<<运算符，但是它并不需要按照任何特定的顺序来对操作数进行计算。特别地，它可能在输出了“Your final grade is”之后就马上调用 grade 函数。假如 grade 抛出了一个异常，那么输出就有可能包含了上面的那个无用的短语。此外，对 setprecision 的第一次调用会把输出流的精度设为 3，然而第二次调用却有可能根本不能获得一个机会来把精度重新设置成它原先的值。另外的一种情况，系统环境有可能会在写任何的输出之前就调用了 grade 语句；而是否会这样做则要视不同的系统环境而定。

上面的分析说明了为什么我们要把输出块语句分割成两部分：第一条语句确保了，在产生任何的输出之前程序就调用了 grade。

我们有一条相当好的经验规则——也就是，我们要保证在一条语句中的副作用个数不会超过一个。抛出一个异常是一个副作用，因此在一条可能会引发异常的语句中不应该再出现任何其他的副作用，尤其是那些包含有输入和输出的语句。

当然，我们编写的这个 main 函数还不能正常运行。我们还需要添加 include 指令，同时我们还要为程序使用的库工具添加 using 声明。我们还要使用 read_hw 和 grade 函数，这两个函数的第三个参数的类型都是 const vector<double>&。这些函数的定义依次使用了 median 函数和 grade 函数，median 和 grade 都具有三个 double 类型的参数。

为了执行这个程序，我们要保证，这些函数是在 main 函数之前（以恰当的顺序）定义的。如果是这样的话，我们就肯定会碰到一个让人困惑的难题。在这里我们并不直接的指出这个问题，但是在§4.3 中将会看到，我们是怎样把这样的程序分割成更简明的文件的。在这样做之前，让我们先寻找一个更好的办法来构造我们的数据。

4.2 组织数据

计算一个学生的成绩对这个学生来说是很有用的，不过这个计算十分简单，我们用一台袖珍计算器就能取得几乎跟我们的程序一样的效果。另一方面，如果我们是在教授一门课程的话，那我们就要算出一整班学生的成绩。让我们修正我们的程序，让它对教师也同样适用吧。

我们不希望交互地报告一个学生的成绩，而是假定有一个包含有许多学生的姓名以及成绩的文件。每一个姓名后面都紧跟着一个期中成绩和一个期末考试成绩，然后，接下来有一个或多个的家庭作业成绩。这样的一个文件的形式如下：

```
Smith       93  91  47  90  92  73  100  97
Carpenter   75  90  87  92  93  60  0    98
...
```

我们的程序将用中值计算出每一个学生的总成绩：家庭作业的中值占40%；期末成绩40%；期中成绩则占20%。

对于这个输入，相应输出应为：

```
Carpenter   86.8
Smith       90.4
...
```

在这个输出中，我们希望按学生姓名的字母顺序来组织输出报表，而且我们希望纵向排列总成绩以便于阅读。这些需求表明了，我们需要有一个地方来存储所有学生的记录，这样我们就可以按字母顺序来排列它们了。为了确定在每一个姓名和对应的成绩之间要放置多少个空格，我们还要找出最长的姓名的长度。

假定有一个地方可供我们存储一个学生的数据，那我们就可以用一个向量来保存所有学生的数据了。一旦向量包含了所有学生的数据，我们就可以对这个向量进行排序，然后我们就计算并输出每一个学生的成绩。首先，我们会建立一个数据结构来保存学生的数据，然后我们会编写一些辅助函数来读取并处理这些数据。在详述了这些抽象之后，我们就使用它们来解决所有的问题。

4.2.1 把一个学生的所有数据放置在一起

我们知道，我们需要读所有学生的数据然后按字母顺序来对这些学生进行排序。在这样处理的时候，我们希望把学生的姓名与成绩放置在一起。因此，我们需要找到一个方法来把属于一个学生的所有信息通通都存储在一个地方。这个所谓的地方应该是一个数据结构，它保存了学生的姓名、期中和期末考试成绩以及所有的家庭作业成绩。

在C++中，我们定义了如下的一种数据结构：

```
struct Strudent_info {
```

```
    string name;
    double midterm, final;
    vector<double> homework;
};  //注意这里的分号——它是不可缺少的
```

这个 struct（结构）的定义表示，Student_info 是一种具有四个数据成员的类型。因为 Student_info 是一种类型，所以我们就可以定义这种类型的对象，而每一个对象都会包含这四个数据成员的一个实例。

第一个叫做 name 的成员是 string 类型的；第二个和第三个都是 double 类型的，它们的名称分别是 midterm 和 final；而最后一个成员则是一个 double 类型的向量，名字叫做 homework。

每一个 Student_info 类型的对象都保存了一个学生的信息。因为 Student_info 是一种类型，所以我们用一个 vector<Student_info>类型的对象来保存任意数目的学生的信息——就好像我们之前用一个 vector<double>类型的对象来保存任意数目的家庭作业成绩一样。

4.2.2 处理学生记录

如果把我们的问题分成几个部分来考虑，我们将看到有三个独立的步骤，而且我们可以用不同的函数表示它们：我们要把数据读到一个 Student_info 类型的对象中；要为一个 Student_info 类型的对象生成总成绩；还要找到一种方法来对一个 Student_info 类型的向量进行排序。

这个读一条记录的函数跟我们在§4.1.3 中编写的 read_hw 函数十分相似。实际上，我们可以使用这个函数来读家庭作业成绩。另外，我们还需要读学生的姓名以及考试成绩：

```
istream& read(istream& is, Student_info& s)
{
    //读入并存储学生的姓名以及期中、期末考试成绩
    is >> s.name >> s.midterm >> s.final;

    read_hw(is, s.homework);        //读入并存储学生的所有家庭作业成绩
    return is;
}
```

把这个函数命名为 read 并不会导致意义上的含混，这是因为它的第二个参数的类型会让我们知道我们正在读什么。重载将会把它跟其他任何可能读数据到任意类型的结构中的名为 read 的函数区分开来。跟 read_hw 一样，这个函数具有两个引用：一个是 istream，我们就是从 istream 中读数据的；另一个引用则是一个对象，它存储了函数所读到的数据。在函数内部使用参数 s 的时候，我们将会影响传递给我们的那个参数的状态。

这个函数首先把数值读到对象 s 的成员 name、midterm 和 final 中，然后它调用 read_hw 来读家庭作业成绩。在这个过程中的任一时刻，我们都有可能到达文件结尾或遇到输入失败的情况。如果是这样的话，那么随后的输入尝试将会什么都做不了，从而在我们返回的

将处于适当的错误状态中。值得注意的是，这个行为之所以会发生是有一个前提的：如果在我们调用 read_hw 函数（§4.1.3）的时候输入流已经处于错误状态了，那么 read_hw 就会让输入流继续保持这个状态。

另外一个我们所需要的函数是被用来为一个 Student_info 类型的对象计算总成绩的。在§4.1.2 中，在我们定义 grade 函数的时候，我们就已经解决了这个问题的绝大部分。我们会通过重载 grade 函数来让我们的工作开展得更深入一点。重载后的版本算出了一个 Student_info 类型的对象的总成绩：

```
double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

这个函数对一个 Student_info 类型的对象进行处理，然后它返回一个表示总成绩的 double 类型的值。注意到参数的类型是 const Student_info&而不仅仅是简单的 Student_info，这样的话，在我们调用它的时候就不会有复制整个 Student_info 对象而带来的额外开销了。

我们还注意到，我们在这个函数中调用的 grade 函数有可能会抛出一个异常，而我们在函数中并没有就这种可能性做任何相应的防护工作。之所以这样做是因为，我们的 grade 函数不能丢下它所调用的函数已经进行的操作不管而去对异常做任何的处理动作。因为我们的 grade 函数不会捕获异常，所以出现的任何异常都会往后传递给我们的调用程序，而在调用程序中决定应该对那些没有做过家庭作业的学生采取什么样的处理动作是较为合适的做法。

在编写整个程序之前，我们的最后一个任务是要决定怎样对我们的 Student_info 对象的向量进行排序。在那个中值函数（§4.1.1）中，我们在对一个名为 vec 的 vector<double>类型的参数进行排序时所使用的工具是库的 sort 函数：

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

然而，如果我们假定我们的数据是存储在一个名为 students 的向量中的，那我们就不能用以下的形式了：

```
sort(student.begin(), student.end());        //不太正确
```

为什么会这样呢？在第 6 章中我们会对 sort 和其他库算法展开更多的讨论，不过在这里我们有必要以较为抽象的方式去思考一下 sort 的工作原理。特别地，sort 是如何得知向量中值的排序规则的呢？

sort 函数必须比较向量中的元素以便把它们按顺序存放。对任何待排序的向量元素类型，它都会使用<运算符来做这个动作。我们可以对 vector<double>类型调用 sort，因为<运算符会比较两个 double 类型的数，然后得出一个适当的结果。如果 sort 要去比较类型为 Student_info 类型的值的话那又会怎样呢？在使用于 Student_info 对象的时候<运算符并没有一个明显的意义。实际上，如果 sort 试图去比较两个这样的对象的话，那编译器就会提示出错。

值得庆幸的是，sort 函数还有一个可选的第三个参数，这个参数是一个谓词。谓词是一个函数，它会产生一个真值，而且，这个真值的典型类型是 bool（布尔）。如果第三个参数存在，那 sort 函数就会使用它而不是使用<运算符来比较元素。因此，我们要再定义一个函数，这个函数有两个类型为 Student_info 的参数。我们用这个函数来判别第一个参数是否小于第二个。因为我们希望按姓名字母顺序来排列学生，所以我们所编写的比较函数将仅仅对姓名进行比较：

```
bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}
```

这个函数所做的只不过是把比较 Student_info 对象的工作交给类 stirng 去完成。string 类提供了一个<运算符来比较字符串，这个运算符按照普通的字典排序法比较字符串。这就是说，如果左操作数在字母表中位于右操作数之前，那么它就认为左操作数比右操作数小。这样的行为刚好是我们想要的。

定义了 compare 之后，我们就可以把 compare 函数作为第三个参数传递给库函数 sort 从而对向量进行排序了：

```
sort(students.begin(), students.end(), compare);
```

sort 比较元素的时候就会调用我们的 compare 函数。

4.2.3 生成报表

既然已经有了处理学生记录的函数，那我们就可以生成我们的报表了：

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;

    //读并存储所有的记录，然后找出最长的姓名的长度
    while (read(cin, record)) {
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    //按字母顺序排列记录
    sort(students.begin(), students.end(), compare);

    for (vector<Student_info>::size_type i = 0;
         i != students.size(); ++i) {
```

```
        //输出姓名，填充姓名以达到maxlen + 1的长度
        cout << setw(maxlen+1) << students[i].name;

        //计算并输出成绩
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                << setprecision(prec);
        } catch (domain_error e) {
            cout << e.what();
        }

        cout << endl;
    }

    return 0;
}
```

我们已经见过了这个代码的绝大部分，但是其中有两个地方对我们来说是陌生的。第一个是对库函数 max 的调用，这个函数是在头文件<algorithm>中定义的。表面上，max 的行为是很明显的。然而，它的行为的其中一个方面是不明显的：由于某些复杂的原因，它的两个参数必须具有同样的类型——我们将在§8.1.3 中对此进行讨论。根据这个要求，仅仅把 maxlen 定义成 int 类型是不适合的，我们必须把 maxlen 定义成一个 string::size_type 类型的变量。

另外一个陌生的地方是那些我们用来安排格式的函数：别忘了，我们之前已经决定把成绩对齐在一直列中了。幸运的是，库为我们提供了 setw 和 setprecision 控制器，它们可以帮助我们解决这一部分问题。

在§3.1 我们原先的那个计算成绩的程序中我们使用了 setprecision。setw 的工作方式与之相似，但它与其他库工具的交互方式则明显跟 setprecision 不同。跟 setprecision 一样，我们用 setw 来控制输出流的格式。就 setw 而言，我们使用它来让库把下一个输出项填充成有特定数目的字符集（通过给输出项添加空格来完成）。我们可以用 setw 来确保输出中的每个姓名都会消耗跟输入中的最长的姓名一样多的字符，同时它还会用一个额外的空格来作为姓名与成绩之间的空白。

跟修改流精度的操作不同，对宽度的修改是短暂的。也就是说，每个标准输出运算符在完成了它的输出动作之后就马上会重置流的宽度。因此，如果我们编写下面的语句，

```
setw(maxlen + 1) << students[i].name
```

那么这条语句的作用就是为 name 填充字符，同时输出 name 的运算符将把流的宽度重置为 0。因此，我们并没有必要像在使用 setprecision 时所做的那样记住并重置宽度。

我们使用索引 i 来逐个处理 students 的元素。为了获取当前的 Student_info 元素，我

索引写到 students 中，这样我们就取得了待输出的姓名。然后我们输出对象的 name 成员并使用 setw 来填充输出。

接下来我们输出每一个学生的总成绩。如果学生并未做过任何的家庭作业，那么对成绩的计算就会引发一个异常。这样的话，我们就要捕获这个异常。在这样的情况下我们不会输出一个数字成绩，而是会输出那个被引发的作为异常的一部分的信息。每一个标准库异常（例如 domain_error）都会记住一个（可选的）参数，这个参数被用来描述那个导致抛出异常的问题。每一种类型的异常都会把这个参数的内容复制下来，而且在一个名为 what 的成员函数中会用到这个复制。在这个程序中的 catch 语句会给那个来自 grade 函数的异常指定一个名称，这样它就可以输出 what( )中的信息了。这个信息会告知用户“学生并没有做过家庭作业”。如果没有异常的话，我们就用 setprecision 控制器来指明我们将输出三个有效位，然后我们输出 grade 函数的结果。

4.3 把各部分代码连接到一起

到现在为止，我们已经定义了很多的抽象（函数和数据结构），这些抽象对于解决各种成绩计算的问题是很有用的。我们之前所见过的使用这些抽象的惟一办法是把它们的所有定义都放在同一个文件中并编译这个文件。很明显，这个方法很快就会变得越来越复杂。跟许多语言一样，C++为了降低复杂度而为分块编译的概念提供了支持，这个概念允许我们把程序放进不同的文件中并独立地编译这些文件中的每一个。

让我们先了解一下，为了方便他人的使用，我们应如何组装 median 函数。我们首先把 median 函数的定义放进一个单独的文件中，这样就可以独立地编译它了。这个文件必须包含有 median 函数使用的所有名称的声明。median 使用了库的 vector 类型、sort 函数以及 domain_error 异常。因此我们必须把对应于这些工具的头文件包含进去：

```
//median 函数的源文件
#include <algorithm>          //获取 sort 的声明
#include <stdexcept>          //获取 domain_error 的声明
#include <vector>             //获取 vector 的声明

using std::domain_error;   using std::sort;   using std::vector

//计算一个 vector<double>类型的对象的中值
double median(vector<double> vec)
{
    //在§4.1.1 中定义的函数体
}
```

跟所有的文件一样，我们必须给我们的源文件指定一个名称。C++标准并未告诉我们如何给源文件命名，但是一般而言，一个源文件的名称应该向人提示它的内容。此外，大

统环境都会对源文件名称进行约束，而一般的做法是要求名称的最后几个字符具有特定的形式。系统环境用这些文件后缀来判别某个文件是否是一个 C++源文件。大多数系统环境都要求 C++源文件名称的后缀是.cpp、.C 或.c，因此我们可以把 median 函数放进一个名为 median.cpp、median.C 或 median.c 的文件中，但究竟用哪个名称则要视系统环境而定。

接下来我们要让 median 函数对其他用户也可用。标准库把其定义的名称放进头文件中，与之类似，我们可以编写自己的头文件，这样用户就可以使用我们所定义的名称了。例如，我们可以在一个名为 median.h 的文件中标明 median 函数的存在。这样的话，用户只要编写了以下的语句就可以使用它了：

```
//使用median函数的一种较好的方法
#include "median.h"
#include <vector>

int main { /* ... */ }
```

如果我们在#include 指令中使用了一个双引号而不是尖括号来把头文件名称括起，那么就表示，为了替代#include 指令，我们要求编译器把与此名称对应的头文件中的所有内容都复制到我们的程序中。每一个系统环境都会判断头文件的位置，同时它还会判别在引号和文件名称之间的字符串所表示的关系。“系统环境所判定的与名称 median.h 相对应的那一个文件”的一个简称就是“头文件 median.h”。

值得注意的是，我们把自己的头文件称作头文件，而对于系统环境提供的头文件，我们却把它们称作标准头（standard header）而不是标准头文件（standard header file）。这样做的原因是，在所有的 C++系统环境中，头都是真正的文件，然而我们却没有必要让系统头作为文件。尽管我们能用#include 指令访问头文件和系统头，但我们还是没有必要以同样的方式来实现它们。

既然我们必须提供一个头文件，那么我们就会很自然地提出一个这样的问题：在这个头文件里面应该有些什么呢？对此问题的一个简单的回答是，我们必须为 median 编写一个声明。为此我们要用一个分号来代替这个函数的函数体，同时我们还可以去掉参数的名称，因为在没有函数体的情况下它们是不相干的：

```
double median(vector<double>);
```

我们的 median.h 头不能单单包含这个声明；我们还需要把声明本身所用到的所有名称都包含进去。这个声明使用了 vector，因此我们必须确保在编译器看到我们的声明之前这个名称就已经是可用的：

```
// median.h
#include <vector>
double median(std::vector<double>);
```

在声明 median 的参数的时候，为了使用名称 std::vector，我们就必须把 vector 头包括在内

我们为什么要明确地提到 std::vector 而不是编写一个 using 声明呢？有一个微妙的原因。

一般而言，头文件应该只声明必要的名称。限定了包含在头文件中的名称之后，我们就可以为我们的用户保留最大限度的灵活性了。例如，我们之所以对 std::vector 使用限定名是因为，我们无法得知 median 函数的用户希望以何种方式引用 std::vector。我们的代码的用户可能不想使用一个 using 声明。因此，如果我们在头中编写了一个的话，那么所有包含了我们的头的程序就会获得一个 using std::vector 声明，而不管它们是否需要它。因此，头文件应该使用完整的限定名而不是使用 using 声明。

让我们讨论一下最后的一个细节：作为对程序的编译的一部分，如果我们要把头文件不止一次地包含在内的话，那么头文件就必须保证这个多次包含的动作是安全的。这样的话，我们的头文件就已经是安全的，因为它仅仅是包含了声明。然而，我们不但要在需要多次包含的头文件中满足这个要求，还应该让每一个头文件都满足多次包含的要求。为此，我们可以给文件添加一些关键的预处理程序：

```
#ifndef GUARD_median_h
#define GUARD_median_h

//median.h 的最终版本
#include <vector>
double median(vector<double>);

#endif
```

#ifndef 指令检查 GUARD_median_h 是否被定义。GUARD_median_h 是一个预处理程序变量的名称。在 C++中有许多控制程序编译的方法，而使用预处理程序变量就是其中的一种。对预处理程序的详细讨论已经超出了本书的范围。

在这里的例子中，如果这个特定的名称还未定义的话，那么#ifndef 指令就会请求预处理程序对在它和下一个匹配的#endif 之间的所有内容做出适当的处理动作。我们必须在整个程序中选择一个惟一的名称，因此根据我们的文件名称和一个诸如 GUARD_这样的字符串构造了一个，我们希望这个名称不会跟其他地方的任何名称相同。

在 median.h 第一次被包含在一个程序中的时候，GUARD_median_h 将会是未经定义的，因此预处理程序会查看这个文件的其余部分。它所做的第一件事是定义 GUARD_median_h，这样的话，如果随后我们还尝试把 median.h 包含进去，那这些尝试将会无效。

最后，我们应该让#ifndef 刚好位于文件的第一行——就算是注释也不能跑到它的前头：

```
#ifndef  variable
-
#endif
```

之所以要这样做是因为，有些 C++系统环境会对这种形式的文件进行检测，而且，如果 variable 已经定义了的话，那它们就根本不会第二次读这个文件。

4.4 把计算成绩的程序分块

既然我们已经知道了怎样去安排 median 函数的独立编译，那么下一步的工作就是把我们的 Student_info 结构和相关的函数组装起来。

```
#ifndef GUARD_Student_info
#define GUARD_Student_info

//头文件 Student_info.h
#include <iostream>
#include <string>
#include <vector>

struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};

bool compare(const Student_info&, const Stdudent_info&);
std::istream& read(std::istream&, Student_info&);
std::istream& read_hw(std::istream&, std::vector<double>&);
#endif
```

值得注意的是，我们在限定标准库的名称的时候所使用的是 std::而不是 using 声明。另外，我们在 Student_info.h 中声明了与 Student_info 结构密切相关的 compare、read 以及 read_hw 函数。我们只是在使用这个结构的时候才会使用这些函数，因此把这些函数与结构的定义组装在一起的做法是合理的。

这些函数应该在如下的源文件中定义：

```
//与 Student_info 相关的函数的源文件
#include "Student_info.h"

using std::istream;    using std::vector;

bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

istream& read(istream& is, Student_info& s)
{
    //我们把这部分代码定义在§4.2.2 中
}
```

```
istream& read_hw(istream& is, vector<double>& hw)
{
    //我们把这部分代码定义在§4.1.3中
}
```

值得注意的是，因为我们包含了 Student_info.h 文件，所以这个文件把我们的函数的定义和声明都包含了进去。这种冗余是无害的，不单如此，它还是一种很好的做法。它为编译器提供了一个机会来让它检查声明与定义是否一致。在大多数的系统环境中这些检查都是不彻底的，因为完全的检查需要查看整个程序。不过这些检查还是很有用的，所以我们有必要在源文件中包含相应的头文件。

这种检查及其不完全性的起源是很简单的：C++语言要求函数的声明和定义在返回值类型、参数的个数以及类型这几个方面严格匹配。这条规则解释了系统环境的检验能力——但是为什么会有不完全性呢？原因是，如果一个声明和定义的差别太大的话，那么系统环境就只能假定它们是描述一个重载函数的两个不同的版本的，同时系统环境还会假定那个丢失的定义会在其他地方出现。例如，假设我们定义了§4.1.1 中的那个 median 函数，同时我们错误地把它声明为：

```
int median(std::vector<double>);    //返回类型应该是double
```

如果编译器在编译定义的时候看到这个声明，它就会提示出错，因为它知道 median(vector<double>)的返回类型不能同时为 double 和 int。再如，假设函数的声明是：

```
double median(double);              //参数类型应该是vector<double>
```

现在编译器就无法编译了，这是因为，median(double)可能是在其他地方定义的。如果我们调用这个函数，那么最终系统环境还是要查找它的定义。如果找不到定义的话，它就会在这一点上提示出错。

我们还应该留意到，源文件中的 using 声明并没有问题。不像头文件，源文件对使用这些函数的程序并没有影响。因此在源文件中使用 using 声明只不过是一个局部的决策罢了，它并不会影响到全局。

剩下来我们要做的是编写一个头文件来声明不同的 grade 重载函数：

```
#ifndef GUARD_grade_h
#define GUARD_grade_h

//grade.h
#include <vector>
#include "Student_info.h"

double grade(double, double, double);
double grade(double, double, const std::vector<double>&);
```

```
double grade(const Student_info&)
```

应该留意一下，把这些重载函数的声明放在一起会让这些可供选择的函数更易查找。因为这三个函数是密切相关的，所以我们将在同一个文件中定义它们。此外，这个文件的名称将取决于系统环境，不过，它可能是 grade.cpp、grade.C 或 grade.c：

```
#include <stdexcept>
#include <vector>
#include "grade.h"
#include "median.h"
#include "Student_info.h"

using std::domain_error;   using std::vector;

//§4.1、§4.1.2 以及§4.2.2 中的 grade 函数的定义
```

4.5 修正后的计算成绩的程序

最后，我们就可以编写出完整的程序了：

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include "grade.h"
#include "Student_info.h"

using std::cin;                 using std::setprecision;
using std::cout;                using std::sort;
using std::domain_error;        using std::streamsize;
using std::endl;                using std::string;
using std::max;                 using std::vector;
using std::setprecision;

int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;      // 最长的姓名的长度
```

```
    // 读入并存储所有学生的数据
    // 不变式:
    //      students 包含了所有的学生记录
    //      max 包含了 students 中最长的姓名
    while (read(cin, record)) {
        // 找出最长的姓名的长度
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    // 按字母顺序排列学生记录
    sort(students.begin(), students.end(), compare);

    // 输出姓名和成绩
    for (std::vector<Student_info>::size_type i = 0;
        i != students.size(); ++i) {

        // 输出姓名，把姓名填充至 maxlen + 1 个字符的长度
        cout << setw(maxlen+1) << students[i].name;

        // 计算并输出成绩
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                << setprecision(prec);
        } catch (domain_error e) {
            cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

这个程序应该是相当简明易懂的。跟往常一样，我们从必要的 include 指令和 using 声明开始。当然，我们有必要提一下在这个源文件中使用到的头文件和声明。在这个程序中，我们不仅仅包含了自己的头文件，同时还包含了库提供的头文件。使用了这些头文件之后，这个程序就可以使用 Student_info 类型的定义和那些我们用来处理 Student_info 对象并生成成绩的函数的声明了。而这里的 main 函数跟我们在§4.2.3 中介绍的那一个是完全一样的。

4.6 小结

程序结构：

#include <系统头文件>

尖括号括住的是系统头。系统头可能是也可能不是以文件的形式实现的。

#include "用户定义的头文件名称"

包含在程序中的用户定义的头文件名称是用双引号括起的。通常，用户定义的头有一个后缀.h。

我们应该防止对头文件的重复包含，为此我们可以用一条#ifndef 指令来把这个文件围起。在头文件中我们应该避免声明它们无需用到的名称。特别地，它们不应该包含有 using 声明，相反，我们应该给标准库的名称明确地加一个前缀 std::。

类型：

T&　　表示对类型 T 的一个引用，它通常被用来传递一个可以由函数修改的参数。与这种参数对应的参数必须是左值。

const T&　　表示对类型 T 的一个引用，而且我们不能对这个引用的值进行修改。我们可以用它来避免复制函数参数所产生的开销。

结构：结构是一种包含有 0 个或多个成员的类型。每一个结构类型的对象都包含它自己的所有成员的实例。

每一个结构都应该有相应的定义：

```
struct 类型名{
    类型说明符  成员名;
    ...
};        //要注意这里的分号
```

跟所有的定义一样，结构的定义在每一个源文件中只可以出现一次，因此它应该出现在一个有适当防护措施的头文件中。

函数：一个函数在每一个使用它的源文件中被声明，函数仅仅被定义一次。声明和定义的形式是类似的：

```
ret-type function-name (parm-decls);                //函数声明
[inline] ret-type function-name (parm-decls) {      //函数定义
    //函数体
}
```

在这里，ret-type 是函数的返回类型，parm-decls 是一个有逗号分隔的函数参数的类型列表。在调用函数之前我们必须先声明它们。每一个参数的类型都必须与相应的参数一致。如果我们想要声明或定义一个具有十分复杂的返回类型的函数，那我们就需要一种不同的语法。详情请参阅§A.1.2。

函数名可以重载：同一个 function-name（函数名）可以定义多个函数——只要这些函数在参数的个数或类型上有差异就行了。系统环境能区分对同一个类型的引用和常量引用。

inline 是可选的，我们能够用它来限定一个函数定义。inline 会在适当的条件下请求编译器把调用扩展成内联子过程——也就是说，为了避免函数调用的额外开销，编译器会用函数体的一个复制来替换对函数的每一个调用并根据需要进行修正。为此，编译器必须要看到函数的定义，因此内联子过程通常是在头文件而不是在源文件中定义的。

异常处理：

`try {` //代码 启动一个可能会引发异常的块语句。

`} catch (t) { /* 代码 */ }`

终止 try 块语句并处理与类型 t 匹配的异常。跟在 catch 之后的代码执行了适当的操作来处理 t 报告的异常。

`throw e;` 终止当前的函数并把值 e 传回给调用程序。

异常类：库定义了几个异常类，异常类的名称表明了它们可以报告的问题的种类：

logic_error	domain_error	invalid_argument
length_error	out_of_range	runtime_error
range_error	overflow_error	underflow_error

`e.what( )` 返回一个值，这个值报告了问题的所在。

库工具：

`s1 < s2` 应用字典排序法来比较字符串 s1 和 s2。

`s.width(n)` 为了下一次的输出操作而把流 s 的宽度设置为 n（如果省略了 n，那么精度保持不变）。输出的右边会被填充至给定的长度。返回原先的长度。值得注意的是，标准输出运算符使用已有的宽度值，然后它们会调用 width(0) 来重置宽度。

`setw(n)` 返回一个类型为 streamsize 的值（§3.1）。如果把这个函数用于输出流 s，那么它的作用跟调用 s.width(n)的一样。

习题

4-0 编译、运行并测试本章中的程序。

4-1 我们注意到，在§4.2.3 中，在调用 max 的时候，必须让参数的类型严格匹配。下面的代码正确吗？如果有问题的话，那你将怎样改正它呢？

```
int maxlen;
Student_info s;
max(s.name.size() , maxlen);
```

4-2 编写一个程序来计算从 1~100 的整数（int）值的平方。程序的输出分为两列，第一列是整数值，第二列是整数值的平方。使用控制器来控制输出，让数值按列排列起来

4-3 如果我们重写了上题中的程序，让它计算从 1 到 999 的整数的平方。但是，我们忘记了更改 setw 的参数的值。这样做会有什么问题呢？重写这个程序，让它具有更好的适应性。重写后的程序应实现这样的目标：当 i 增长时我们不需要修正 setw 的参数。

4-4 现在，再次修改你的求平方程序，用它来求 double 类型而不是 int 类型的值的平方。使用控制器来控制输出，让数值按列排列起来。

4-5 编写一个函数来从输入流读单词，把读到的单词存储在一个向量中。利用这个函数编写一个程序来计算输入的单词的数目以及每一个单词所出现的次数。

4-6 重写 Student_info 结构并使用重写后的结构来直接计算成绩，要求在程序中仅仅存储总成绩。

4-7 编写一个程序来计算存储在一个 vector<double>类型的向量中的数据的平均值。

4-8 如果下面的代码是合法的，那么对于 f 的返回类型我们能做出什么推断呢？

```
double d = f( ) [n];
```

4-9 在C++编程中有一个相当常见的程序错误：如果我们像在§4.2.3 中所做的那样给 setw 一个字符串类型的参数，那么 setw 函数可能会无法正常工作。重写§4.2.3 中的程序，让它不用再依赖 setw 函数。提示(§1.2)：string spaces(n, ' ')构造了一个含有 n 个空格的字符串。

第 5 章

使用顺序容器并分析字符串

通过前面的学习，我们已经具备了相当的核心 C++语言基础，同时我们也掌握了一定的字符串类和向量类的知识。有了这些工具，我们就可以解决许多问题了。

在这一章中，我们会把注意力发散到这些工具之外的地方，我们将开始更深入地了解库的使用方法。正如我们即将看到的那样，库提供的工具能够帮我们解决比到目前为止所遇到的更为杂乱的问题。

标准库不仅提供了有用的数据结构和函数，而且它还反映了一个具有一致性的体系结构：一旦我们了解了一种容器的行为特性，那我们就可以很轻松地掌握所有库容器的使用方法了。

例如，像我们将在本章的后半部分所见到的那样，我们经常可以把一个字符串“当作”向量来使用。逻辑上，对一种库类型的许多有用的操作和对另外一种类型的相应操作是等价的。库的构造为这些等价操作提供了方便——对于不同的类型，这样的等价操作会以同样的方式去工作。

5.1 按类别来区分学生

让我们回顾一下在§4.2 中那个计算学生成绩的问题。现在，假定我们不仅希望算出学生的成绩，同时我们还想知道哪些学生不能通过这一门课程。解决这个问题的思想是，如果我们有一个 Student_info 记录的向量，我们就抽取出课程不能过关的那些学生的记录，然后把这些记录存储在另一个向量中。同时我们还希望在原先的向量中删去这些不能过关的学生的数据，这样的话，原先的向量就仅仅是包含了通过这门课程的学生的记录。

我们先编写一个简单的函数来判断一个学生是否不及格：

```
//判断学生是否不及格
bool fgrade(const Student_info& s)
{
    return grade(s) < 60;
}
```

我们使用了§4.2.2 中的 grade 函数来计算成绩，同时我们把低于 60 分的成绩定为不及格成绩。

我们可以用一个最简单的方法来解决全部的问题：逐个检查所有学生的记录，然后在两个向量中选择一个来存放它，这两个向量中的一个是用来存储成绩及格的学生记录的，另一个则为成绩不及格的学生记录而设。

```
//第一次尝试：把及格和不及格的学生记录分离开来
vector<Stufdent_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> pass, fail;

    for (vector<Student_info>::size_type i = 0;
         i != students.size(); ++i)
        if (fgrade(student[i])
            fail.push_back(student[i]);
        else
            pass.push_back(student[i]);

    students = pass;
    return fail;
}
```

当然，我们的这个代码还不能通过编译，为此，我们还需要为我们用到的名称加上#include指令和 using 声明。一般来说，在我们所介绍的代码中将不会再出示这些语句。不过，如果我们使用了一个新的头文件，那当然要提到它。

跟第 4 章的 read_hw 和 read 函数一样，这个函数有两个有效输出：一个是我们返回的vector<Student_info>类型的向量，它包含了成绩不及格的学生记录；另一个则是在调用extract_fails 时所产生的副作用。这个函数的参数是一个引用，因此对参数的修改会反映到参数中。在我们的函数结束执行的时候，那个作为参数传递过来的向量将会仅仅包含成绩及格的学生的记录。

这个函数会创建两个向量，这两个向量分别保存了成绩及格和成绩不及格的学生数据。函数会查看 students 中的每一个记录，如果所查看的记录的成绩及格的话，那函数就给 pass 添加这个记录的一个副本，否则，它会把这个副本添加到 fail 中。

在 for 语句结束之后，我们就把成绩及格的记录重新复制到 students 中，然后函数返回成绩不及格的记录。

5.1.1 就地删除元素

exract_fails 函数为我们完成了我们所期望的工作，并且这个函数是相当有效的。不过它还是有一个很小的缺陷：它要求有足够的内存来保存每一个学生记录的两个副本。之所以会这样是因为，在它不断增加 pass 和 fail 的长度的同时，原先的记录却依然存在。在函数处理了 for 语句并准备复制结果以及返回的时候，每一个学生的记录就都会有两个副本。

除非是有特别的需要，否则我们都不希望让数据的多个副本同时存在。为此，我们

整个 pass 都去掉。我们并没有必要创建两个向量；相反，我们将创建一个名叫 fail 的局部向量来保存我们想要返回的值。我们会为 students 的每一条记录计算成绩。如果成绩及格，我们就不去管这条记录；如果不及格，那我们就把它的一个副本添加给 fail 并从 students 中删除它。

为了使用这个策略，我们需要找到一种方法来从向量中删除一个元素。对此的好消息是，这样的工具确实存在；坏消息是，从向量中删除元素将会耗费大量的时间，因此对于大规模的输入我们不应该采用这样的做法。如果我们处理的数据实在太大的话，那性能就会降至一个令人惊讶的程度。

例如，如果我们的全部学生都不及格，那我们就会看到，函数的运行时间将会与学生数目的平方成比例地增加。这意味着，对于一个有 100 个学生的班级，程序的运行时间将会是在只有一个学生的情况下的 10,000 倍。之所以会出现这个问题是因为，我们的输入记录是存储在一个向量中的，而向量是为了快速随机存取而被优化的。这种优化的一个代价是，在除了向量结尾的其他地方插入或删除元素的开销可能会很大。

我们将会看到两个解决性能问题的方法：我们可以使用一个对我们的算法更加合适的数据结构；或者可以使用一个更加聪明的算法来避免在我们原先的设计中的额外开销。从这里开始一直到§5.5.2，我们将会详细讨论一个使用了更加合适的数据结构的解决方案。而在§6.3 中我们将出示一个算法上的解决方案。

现在我们还不能理解这些解决方案为何是一种进步，但不要紧，让我们先在某些方面取得进展吧。因此，让我们先来观察一下那个效率不高但很直接的解决办法：

```
// 第二次尝试：函数正确，不过可能会相当慢
vector<Student_info> extract_fails(vector<Student_info& students)
{
    vector<Student_info> fail;
    vector<Student_info>::size_type i = 0;

    // 不变式：students 的索引在[0,i)中的元素所代表的成绩是及格的
    while (i != students.size()) {
        if (fgrade(student[i]) {
            fail.push_back(students[i]);
            students.erase(students.begin() + i);
        } else
            ++i;
    }
    return fail;
}
```

在这个版本的开头，我们首先创建了 fail。fail 是一个向量，我们把成绩不及格的学生的记录复制给它。接下来我们定义了 i，用 i 来作为 students 的索引。我们将逐个处理每一个记录，对 students 进行循环访问，直到处理完了 students 的所有项目为止。

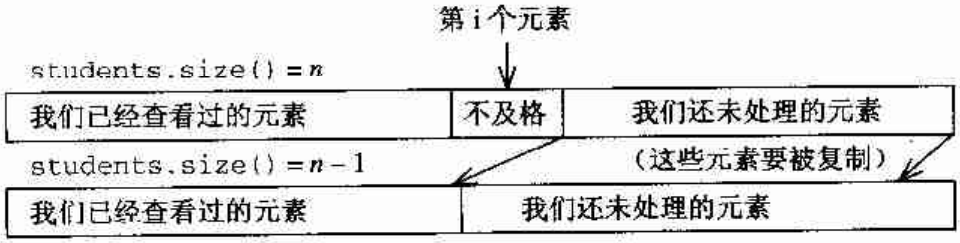
对 students 的每一条记录，我们都要判断它是否代表了一个不及格的成绩。如果是

话，那我们就要把这条记录复制到 fail 中并且把它从 students 中删除掉。这个 push_back 调用对我们来说并不陌生——我们用它来把 students[i]的一个副本添加给 fail。陌生的是那个我们用来删除 students 的元素的方法。

```
students.erase(students.begin( ) + i);
```

vector 类型含有一个名为 erase 的成员，这个成员的作用是从向量中删除一个元素。erase 的参数指示了待删除的元素。有时候 erase 函数并不能对索引进行操作，这是因为，正如我们将在§5.5 中看到的那样，并不是所有的容器都支持索引的，对于库来说，更为实用的是提供某一种形式的 erase 函数，让这个函数对于所有的容器都能以同样的方式工作。erase 函数可以使用另一种参数类型，而在§5.2.1 中我们将会对此进行讨论。现在，对我们来说重要的是要理解，我们可以把索引和 student.begin()的返回值相加从而指示待删除的元素。student.begin()的返回值指示了向量的第一个元素——也就是索引为 0 的那个。如果我们让这个返回值和一个整数（例如 i）相加，那结果就表示了索引为 i 的元素。现在我们可以看出，对 erase 的调用会把 students 中的第 i 个元素删除掉。

一旦我们从向量中删除了一个元素，那么与删除前相比，现在的向量就会少了一个元素：



除了改变向量的长度之外，erase 还会删除索引为 i 的元素,这样它就会让 i 指示序列中的下一个元素。每一个在位置 i 之后的元素都会被复制到前一个位置中。因此，尽管 i 并未改变，但 erase 已经有了调整索引的效果。调整后的索引会指示向量中的下一个元素，这就意味着我们不能为下一次的循环而把它加 1。

如果我们当前正在查看的记录的成绩及格的话，那么我们就把它保留在 students 中。这样的话，我们就必须对 i 加 1，这样在 while 的下一次循环中 i 就会指向下一条记录。

我们把 i 和 students.size()比较，这样我们就可以判断出我们是否已经查看了 students 的所有记录。如果我们删除了向量中的一个元素，那向量的元素个数就会比之前少一个。因此，我们有必要在每一次经过条件时都调用 students.size()。不然的话，如果我们预先计算出 size 并把它的结果存储起来：

```
// 错误的优化将会使这个代码以失败告终
vector<Student_info>::size_type size = students.size();
while (i != size) {
    if (fgrade(students[i])) {
        fail.push_back(students[i]);
```

```
        students.erase(student.begin() + i);
    } else
        ++i;
}
```

我们的程序将会以失败告终，这是因为调用 erase 会改变 students 中的元素个数。如果我们预先计算了 size 的值并且随后删除了所有不及格的学生记录，那么我们就会对 students 进行过多的并且是不必要的访问——而且，students[i]将会指向一个并不存在的元素！幸运的是，对 size 的调用通常都是很快速的，因此我们可以忽略每一次调用 size 时所导致的系统开销。

5.1.2 顺序存取与随机存取

我们的两个版本的 extract_fails 函数跟许多使用容器的程序一样都有着一个相同的特征，在代码中这个特征并不是一眼就能看出来的：每一个这样的函数都只是顺序访问容器中的元素。也就是说，这个函数的每一个版本都是依次查看每一个学生的记录的，它们先对当前的记录作出处理，然后就继续查看下一条记录。

因为这个函数使用了一个整数 i 来访问 students 的每一个元素，所以在代码中我们并不容易察觉到这个特征。我们能以任意的方式来计算一个整数的值，这就意味着，为了让我们更方便地去判断我们是不是在顺序的访问容器，我们就必须查看每一个可能会影响 i 的值的操作，并确定这个操作所带来的影响。我们也可以从另一个角度去观察这个问题，如果我们用 students[i]来访问 students 的一个元素，那么就意味着我们可能会按任何的顺序来访问 students 的元素，而不单单是顺序访问。

我们之所以会关心访问容器时所采取的次序，是因为不同类型的容器会有不同的性能特性，并且会支持不同的操作。如果我们知道，我们的程序仅仅是使用了一个特定类型的容器所能够有效支持的操作，那么我们就能使用这种容器来提高程序效率。

换句话说，因为我们的函数仅仅需要顺序访问，所以我们并没有必要使用那些提供了随机访问任何元素的能力的索引。相反，我们希望重新编写这个函数，让重写后的函数使用那些只支持顺序访问的操作来访问容器的元素。到后来，C++库提供了系列的名为迭代器（iterator）的类型，这些类型允许我们以库能够控制的方式来访问数据结构。这个控制让库确保了高效的实现。

5.2 迭代器

为了使我们的讨论更加具体，让我们先来看一下 extract_fail 实际使用的容器操作。

第一个这样的操作是使用索引 i 来从 Student_info 结构中取值。例如，fgrade(students[i])获取了 students 向量的第 i 个元素并把这个元素传递给 fgrade 函数。我们知道，我们是顺序访问 students 的元素的，这是因为，我们只是以 i 作为索引从而对这些元素进行访问，而

i 执行的惟一操作是读它并让它和向量的长度比较，然后对它加 1：

```
while ( i !=students.size() ) {
    // 在这里进行的操作并不会改变 i 的值
    ++i;
}
```

从上面我们可以清楚地看到，我们仅仅是将之用作顺序访问。

令人遗憾的是，即使我们知道这个事实，但是库却没有办法获知。如果我们用迭代器来代替索引，那么就能让库应用这些知识了。

一个迭代器（iterator）是一个值，它能够

- 识别一个容器以及容器中的一个元素
- 让我们检查存储在这个元素中的值
- 提供操作来移动在容器中的元素
- 采用对应于容器所能够有效处理的方式来对可用的操作进行约束

因为迭代器的行为跟索引相似，所以我们经常可以重写那些使用索引的程序，让它们用迭代器来代替索引。例如，假定 students 是一个 vector<Student_info>类型的向量，它包含了一些学生的记录。让我们看看我们是如何把这些学生的姓名写到 cout 中的。一个方法是使用索引来进行重复操作：

```
for <vector<Student_info>::size_type i = 0; i != students.end(); ++i)
    cout << students[i].name << endl;
```

另一个方法是使用迭代器：

```
for (vector<Student_info>::const_iterator iter = students.begin();
    iter != students.end(); ++iter) {
    cout << (*iter).name << endl;
}
```

为了完成这个重写，我们还需要做相当多的工作。那么就让我们一步步来吧。

5.2.1 迭代器类型

每一个标准容器，例如向量，都定义了两种相关的迭代器类型：

```
container-type::const_iterator
container-type::iterator
```

在这里，container-type 是诸如 vector<Student_info>这样的容器的类型，它包括了容器元素的类型。如果我们想用一个迭代器来修改存储在容器中的值，使用 iterator 类型；如果我们仅仅需要读操作，那么使用 const_iterator 类型。

抽象就是有选择的忽略。我们可能并不了解一个迭代器会有什么样的特殊类型——因为关于这种类型的细节是很复杂的。不过话说回来，我们也没有必要去详细了解它们，我们

了解引用迭代器类型的方法以及迭代器所允许的操作就已经足够了。为了创建迭代器类型的变量，我们有必要了解一下这种类型，但是我们并没有必要去探究关于这种类型的实现的所有细节。例如，我们对 iter 的定义

```
vector<Student_info>::const_iterator iter = students.begin();
```

表明了，iter 的类型是 vector<Student_info>::const_iterator。我们并不知道 iter 的“真实”类型是什么（这是向量的一个实现细节），而且我们也没有必要知道这一点。所有我们要知道的只不过是，vector<Student_info>有一个名为 const_iterator 的成员，这个成员定义了一种类型，我们能使用这种类型来获得对向量元素的只读访问的权限。

我们需要了解的另一点是，从类型 iterator 到类型 const_iterator 有一个自动的转换。正如我们即将看到的那样，students.begin()的返回值类型是 iterator，不过我们说，iter 的类型是 const_iterator。为了用 students.begin()的值对 iter 进行初始化，系统环境会把 iterator 类型转换成相应的 const_iterator 类型。这个转换意味着，我们能把一个 iteratro 类型的对象转换为 const_iterator 类型的，但反过来则不行。

5.2.2 迭代器操作

定义了 iter 之后，我们就把它的值设为 students.begin()。之前我们已经使用过 begin 和 end 函数，因此我们应该知道这些函数是做什么的：它们返回一个值，而这个值指示了一个容器的开头或指示了紧接在容器最后一个元素后面的那个位置。在§8.2.7 中，我们将对我们重复强调的“紧接在容器最后一个元素后面的位置”进行解释。而现在，我们所要知道的是，begin 和 end 函数为容器返回了一个迭代器类型的值。因此，begni 返回了一个类型为 vector<Student_info>::iterator 的迭代器，这个迭代器指向容器的第一个元素。这样的话，iter 一开始会指向 students 的第一个元素。

for 语句中的条件

```
iter != students.end()
```

检查我们是否到达了容器的结尾。end 的返回值指示了在容器最后一个元素后面的第一个位置。跟 begin 一样，这个值的类型是 vector<Student_info>::iterator。我们可以比较两个迭代器（不管其类型有 const 或没有 const）是否相等。如果 iter 等于 stduents.end()的返回值，那么我们就结束循环。

for 语句头的最后一个表达式是++iter，这个表达式对 iter 加 1，让它在 for 的下一次循环过程中指向 students 中的下一个元素。表达式++iter 使用了增量运算符，这个运算符是为迭代器类型重载的。增量运算符的效果是把迭代器推进到容器的下一个元素。我们并不了解而且也不关心增量运算符是怎样工作的，所有我们要知道的是，加 1 后的迭代器指示了容器的下一个元素。

在 for 的循环体内，iter 指向 students 的一个元素，而我们则要对这个元素进行输[illegible]我们通过调用间接引用运算符*来访问这个元素。当*被用于一个迭代器的时候，它会返回

值（§4.1.3），而这个左值是迭代器所指向的元素。因此，输出操作

```
cout << (*iter).name
```

的作用是把当前元素的 name 成员写到标准输出中。

为了正确地执行这个表达式，我们要用括号来覆盖普通运算符的优先级。表达式*iter 返回迭代器所指示的值。“.”的优先级比“*”的高，这就意味着，如果我们只是想把*运算符应用到.的左操作数中，那我们就必须用括号把*iter 括起从而得到(*iter)这样的形式。如果我们编写了*iter.name，那么编译器就会把它当成*(iter.name)来处理，*(iter.name)是一个请求，它请求获取对象 iter 的 name 成员，并用间接引用运算符对这个对象进行操作。因为 iter 并没有一个名为 name 的成员，所以编译器会提示出错。而(*iter).name 则表示，我们希望引用*iter 对象的 name 成员。

5.2.3 一点语法知识

在我们刚才看到的代码中，我们间接引用了一个迭代器，然后根据引用操作所返回的值获取了一个元素。这种操作的结合是十分常见的，我们可以用一种简写形式来等价地表示它：作为

```
(*iter).name
```

的代替形式，我们可以编写

```
iter->name
```

我们可以应用这个语法知识来重写§5.2 中的最后一个例子：

```
for (vector<Student_info>::const_iterator iter = students.begin();
     iter != students.end(); ++iter) {
    cout << iter->name << endl;
}
```

5.2.4 students.erase(students.begin() + i)的含意

既然我们已经对迭代器有了进一步的了解，那我们就可以看出§5.1.1 中的那个程序的关键所在了：

```
students.erase(students.begin() + i);
```

我们已经看到了，students.begin()是一个指向 students 的第一个元素的迭代器，表达式 students.begin() + i 则指向 students 的第 i 个元素。我们有必要了解一下，这里的+运算符是对 students.begin()和 i 的类型定义的，后面的那个表达式的含意来自于+的定义。换句话说，迭代器和索引的类型决定了这个表达式中的+运算符的意义。

如果 students 是一个不支持随机访问索引操作的容器，那么 stduents.begin()就很可能会属于一种无法定义+的类型——这样的话，表达式 students.begin + i 将不能通过编译。实际上，这样的一个容器将会禁止对它的元素的随机访问，不过它仍会允许迭代器的顺序访问。

5.3 用迭代器来代替索引

根据我们所掌握的迭代器的知识和一个新的处理办法，我们在实现 extract_fails 函数的时候就根本无需用到索引：

```
// 版本 3：用迭代器来代替索引；效率可能还是不高
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> fail;
    vector<Student_info>::iterator iter = students.begin();

    while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter;
    }
    return fail;
}
```

像我们之前所做的一样，我们首先定义了 fail。接下来，我们定义了名为 iter 的迭代器，我们会用这个迭代器（代替索引）来查看 studens 中的元素。值得注意的是，我们给它指定的类型是 iterator 而不是 const_iterator：

```
vector<Student_info>::iterator iter = students.begin();
```

我们之所以要使用 iterator 是因为：在我们调用 erase 的时候，我们想用它来对 students 进行修改。我们对 iter 进行初始化，让它指示 students 的第一个元素。

接着我们编写了一条 while 语句来查看 sutdents 的每一个元素。别忘了，迭代器 iter 指示的是容器中的一个元素，因此*iter 就是这个元素的值。为了判断一个学生是不是及格，我们把这个值传递给 fgrade。同样地，对那部分把不及格的记录复制到 fail 中的代码，我们要作一些改动。为此，我们编写了

```
fail.push_back(*iter);          //间接引用迭代器来获取元素
```

来代替

```
fail.push_back(student[i]);     //利用索引来获取向量的元素
```

现在，我们可以直接传递一个迭代器。因此，erase 函数就变得更加简单了：

```
iter = students.erase(iter);
```

当我们计算一个迭代器的时候，我们就不再需要把索引 i 与 sudent.begin()相加了。

我们在这里所用到的新的处理办法很容易被忽略，不过它却是非常重要的：现在，我们把 erase 的返回值赋给 iter。为什么要这样呢？

稍微思考一下我们会知道，这个删除 iter 所指示的元素的动作一定会使迭代器失效。在我们调用了 students.erase(iter)之后，我们知道，iter 已经不能再指向同一个元素了——因为这个元素已经不存在。实际上，如果我们调用 erase 来删除向量中的一个元素，那么所有指向位于被删除元素后面的元素的迭代器都会失效。让我们回顾一下§5.1.1 中的那幅图：很明显，在我们删除了那个有“fail”记号的元素之后，这个元素就消失了，同时在它后面的所有元素都已经移动了。如果这些元素已经移动了位置，那任何指向它们的迭代器都一定是没有意义的。

值得庆幸的是，erase 返回了一个迭代器，这个迭代器指向紧跟在我们刚刚删除掉的元素后面的那个元素。因此，执行

```
iter = students.erase(iter)
```

会让 iter 指向被删除元素后面的那个元素——这个结果正是我们所期望的。

如果我们所处理的是成绩及格的元素，那我们仍然要对 iter 加 1。这样的话，在下一次循环中我们就会位于下一个元素上。我们是在 else 分支中把 iter 加 1 的。

顺便提一下，正如我们在§5.1.1 中看到的那样，我们可能会不自觉地去优化我们的循环，为了不用在 while 的每一次循环中计算 students.end()，我们有可能会把它的值保存起来。换句话说，我们可能不自觉地把

```
while (iter != students.end() )
```

改写成

```
// 由于错误的优化，这个代码将以失败告终
vector<Student_info>::iterator iter = students.begin(),
                               end_iter = students.end();
while (iter != end_iter) {
    //...
}
```

这个代码十有八九会以失败告终。为什么呢？

原因是，如果我们执行了 students.erase，那么所有在被删除位置之后的迭代器都会失效（也包括 end_iter）。因此，我们有必要在每一次的循环过程中调用 students.end()——跟我们在§5.1.1 中所做的一样，在那里我们要在每一次循环中调用 students.size。

5.4 重新思考数据结构以实现更好的性能

对于小规模的输入，我们的程序会有相当好的表现。然而，正如我们在§5.1.1 中所提到的那样，当我们的输入增长的时候，程序的性能表现就会急剧下降。为什么会这样呢？

让我们再来衡量一下使用 erase 函数来删除向量中的元素的利弊。库优化了向量数

以使我们可以快速地访问任意的元素。此外，在§3.2.3中我们看到了，如果是每次增加一个元素而且元素是添加到向量的末尾的话，那向量的性能是相当好的。

在向量的内部插入或删除元素已经是另一个话题了。如果要这样做的话，那为了保持快速随机存取的特性，我们就必须移动位于被插入或删除的元素后面的所有元素。移动元素意味着，我们的新代码所耗费的运行时间会跟向量元素个数的二次方成比例。对于小规模输入，我们可能不会在意这一点时间；但是，如果每一次输入的长度都加倍的话，那运行时间就有可能每次都变成原先的四倍。如果我们要让程序处理一所学校的全部学生而不仅仅是一个班级的学生的数据，那就算使用一台很快速的计算机，我们的程序也将耗费太多的时间。

如果我们想做得更好，那就需要使用另一个数据结构，这个数据结构可以让我们在容器的任何位置插入和删除元素。这样的一个数据结构可能不支持使用索引的随机访问。不过，就算它能提供这样的支持，整数类型的索引也将不会有太多用处，这是因为插入和删除元素将肯定会改变其他元素的索引。既然我们知道了如何去使用迭代器，那我们就可以找到一个方法来处理这样的一个不提供索引操作的数据结构。

5.5 list 类型

在使用迭代器来重写了代码之后，我们就已经消除了对索引的依赖。现在，我们要使用一个数据结构来重新实现我们的程序，这个数据结构能让我们高效地从容器内删除元素。

我们常常需要在一个数据结构的内部插入和删除元素。毫不奇怪，库提供了一种名为 list（list）的类型，这种类型是在<list>头文件中定义的。这种类型是为这种访问方式而被优化的。

vector（向量）是为快速随机访问而被优化的，同样地，优化了 list 类型之后，我们就可以在容器中的任何位置快速的插入和删除元素了。因为 list 必须使用一个更为复杂的结构，所以，如果我们只是顺序地访问容器的话，那 list 的速度就会比 vector 慢。也就是说，如果容器只是（或主要是）从尾部增长和缩小的话，那么 vector 的性能就会比 list 好。不过，如果程序要从容器中删除很多元素——正如我们的程序所做的那样——那么对于大规模的输入，list 的速度就会更快，而且当输入增加时它的速度优势就会更加明显。

跟 vector 一样，list 是一个容器，它能保存绝大多数类型的对象。正如我们即将看到的那样，list 和 vector 有许多相同的操作。因此，我们经常能把对 vector 的操作转化成对 list 的，反过来也一样。一般情况下，我们要更改的只不过是变量类型罢了。

vector 支持而 list 不支持的一个非常重要的操作是索引。和我们刚刚看到的一样，我们能编写一个使用了 vector 的 extract_fails 函数版本来取出成绩不及格的学生的记录，这个版本使用了迭代器来代替索引。事实证明，我们可以改写这个版本的 extract_fails 函数，改写后的函数使用表来代替 vector。为此，我们只需要修改相应的类型：

```
//版本 4：用 list 来代替 vector
list<Student_info> extract_fails(list<Student_info>& students)
```

```
{
    list<Student_info> fail;
    list<Student_info>::iterator iter = students.begin();

    while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter;
    }
    return fail;
}
```

如果把这个代码和§5.3 中的那个版本比较一下，我们就会发现惟一的变化是，在前四行中我们用表代替了 vector。因此，这个函数的返回类型和参数现在变成了 list<Student_info>，同时那个用来保存不及格成绩的局部变量 fail 的类型也有了相同的变化。同样地，迭代器的类型是由 list 类定义的。因此，我们定义时给它指定的类型是 iterator，这种类型是 list<Student_info>的一个成员。list 类型是一个模板，因此我们必须命名尖括号中的类型以说明表保存的对象的类别，这跟我们在定义一个 vector 时所做的是一样的。

我们并没有改变这个程序的逻辑结构。当然，现在我们的调用程序必须给我们提供一个 list，而我们的函数也会返回一个 list 给调用程序。此外，库实现这些操作时所采用的方式与之前的相比已经有了很大的变化，这是因为这个版本是对 list 操作而其他的版本是对 vector 操作的。如果我们执行了++iter，那实际上我们所做的就是把迭代器推进到 list 的下一个元素。同样地，下面的语句

```
iter = students.erase(iter);
```

调用了 erase 的使用 list 的版本并把从 erase 返回的 list 迭代器赋给 iter。当然，对加 1 和 erase 操作的实现肯定会和对应的使用 vector 的那部分有所不同。

5.5.1 一些重要的差别

与那些对 vector 的操作相比，对 list 的操作在某些方面会有所不同，而最大的差别是：对迭代器进行的某些操作所产生的影响是彼此不同的。例如，如果我们从 vector 中删除一个元素，那所有指向被删除的元素或随后的元素的迭代器都会失效。使用 push_back 来给 vector 添加一个元素会使所有指向这个 vector 的迭代器失效。之所以会出现这样的情况是因为：如果我们删除了一个元素，那随后的元素也会相应地移动位置；如果我们给 vector 添加元素，那么给新元素分配空间的动作可能会导致整个 vector 的重新分配。如果我们要在循环中使用这些操作，那就一定要特别小心，我们必须确保，它们不会保存任何可能是无效的迭代器的复制。在前面我们已经看到了，如果我们不适当地保存了 students.end()的值，那么程序出现错误的机率

常大的。

另一方面，对于 list 来说，erase 和 push_back 操作并不会使指向其他元素的迭代器失效。只有指向已被删除的元素的迭代器才会失效，这是因为这个元素已经不再存在了。

我们已经提到过，list 类的迭代器并不支持完全随机的访问。在§8.2.1 中我们会对迭代器的性质进行更为深入的探讨。而目前对我们来说重要的是要知道，因为缺少了这种支持，所以我们就不能使用标准库的 sort 函数来为存储在 list 中的值排序。对此，list 类提供了自己的 sort 成员函数，这个函数使用了一个优化的算法来为存储在 list 中的数值排序。因此，为了排列 list 中的元素，我们就必须调用这个 sort 成员

```
list<Student_info> students
students.sort(compare);
```

而不是像我们对 vector 所做的那样，调用全局的 sort 函数

```
vector<Student_info> students;
sort(students.begin(),students.end(),compare);
```

值得注意的是，compare 函数是对 Student_info 对象操作的，在前面我们使用这个 compare 来对 vector 进行排序，因此，在这里我们也可以使用同样的 compare 来为 list 的 Students_info 记录排序。

5.5.2 一个恼人的话题

这个用来抽取不及格的学生记录的代码是一个很好的例子，从这个例子可以看出我们选择的数据结构对性能的影响。这部分代码对元素进行顺序访问，这就意味着，vector 通常是我们最好的选择。另一方面，我们也要从容器内删除元素，这样的话，我们就应该选择 list。

跟所有和性能有关的问题一样，“最好”的数据结构取决于性能是否足够。性能是一个棘手的话题，一般来说，它超出了本书的范围。不过，值得注意的是，我们选择的数据结构可能会对程序的性能产生深远的影响。对于小规模的输入，list 的效率要比 vector 的低；而对于大规模的输入，一个以不适当的方式使用 vector 的程序运行起来可能会比使用 list 的那些慢很多。特别是在输入增加时，性能可能会以一个令人惊讶的速度下降。

为了测试我们的程序的性能，我们使用了三个学生记录文件。第一个文件有 735 条记录；第二个文件的记录数是第一个的十倍；而第三个的则再大十倍，也就是说，第三个文件有 73500 条记录。下面的表格记录了对每一个文件执行程序所分别耗费的时间（以秒计算）：

文件大小	list	vector
735	0.1	0.1
7,350	0.8	6.7
73,500	8.8	597.1

对于那个有 73,500 条记录的文件，那个使用 list 的程序版本所耗费的运行时间还不到 9 秒，然而，使用 vector 的那个却花了几乎十分钟。如果有更多的学生不及格的话，这个差别甚至还会更大。

5.6 分割字符串

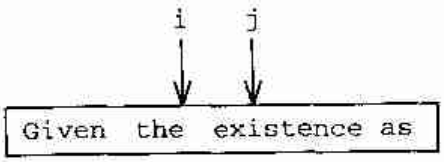
既然我们已经看到了一些对容器的操作，那么，接下来就让我们把注意力转回到字符串上。到目前为止，我们只是对字符串做了一些工作：我们创建它们、读它们、把它们连接起来、输出它们以及查看它们的长度。在所有的这些应用中，我们都把字符串当作单独的实体来处理。在很多情况下，这种抽象的用法正是我们所期望的：我们希望可以忽略字符串的详细内容。当然，有时候我们还是需要查看字符串的特定字符。

事实证明，我们可以把字符串看成是一种特殊的容器：它仅仅包含字符，而且它支持了某些（但不是全部的）容器操作。索引就是它支持的操作之一。string 类型还提供了一种迭代器，这种迭代器与 vector 迭代器类似。因此，我们能把许多技术同时应用于 vector 和字符串。

例如，我们可能会希望把一个输出分割成单词，并且单词间用空白分隔开（空格、制表键或行结尾）。如果我们能直接地读输入，那我们也就可以直接地从输入获取这些单词。毕竟，这恰好正是字符串输入运算符的执行方式：它连续地读字符，直到出现了空白为止。然而，我们经常会希望一次过读一整行输入并检查在这一行中的每一个单词。在§7.3 和§7.4.2 中我们将会看到具体的例子。

这样的一个操作可能具有很广泛的用途，因此我们将编写一个函数来完成它。这个函数有一个字符串参数，它的返回值类型是 vector<string>，字符串的每一个以空白分隔的单词都是函数所返回的 vector 的一个项目。为了更好地理解这个函数，我们有必要了解一下，字符串也支持索引操作，而且它采用的方式跟 vector 几乎是一样的。因此，如果 s 是一个字符串并且它至少有一个字符，那么 s 的第一个字符就是 s[0]，它的最后一个字符则是 s[s.size() - 1]。

我们会在函数中定义两个索引，这两个索引分别是 i 和 j，它们依次为每一个单词定界。我们计算 i 和 j 的值从而为一个单词定位，这样的话，被定位的单词将会由索引域[i,j)所确定的字符组成。例如，



一旦我们有了这些索引，我们就使用它们界定的字符来创建一个新字符串，然后把这个字符串复制到 vector 中。完成了这些操作之后，就把 vector 返回给调用程序：

```
vector<string> split(const string& s)
```

```
{
    vector<string> ret;
    tyepdef string::size_type string_size;
    string_size i = 0;

    // 不变式：我们已经处理了在索引域[先前的 i，i)中的字符
    while (i != s.size()) {
        // 忽略前端的空白
        // 不变式：索引域[先前的 i,当前的 i)中的所有字符都是空格
        while (i != s.size() && isspace(s[i]))
            ++i;

        // 找出下一个单词的终结点
        string_size j = i;
        // 不变式：索引域[先前的 j,当前的 j)中的任一个字符都不是空格
        while (j != s.size() && !isspace(s[i]))
            ++j;

        // 如果找到了一些非空白字符
        if (i != j) {
            // 从 i 开始复制 s 的 j - i 个字符
            ret.push_back(s.substr(i, j - i));
            i = j;
        }
    }
    return ret;
}
```

除了那些我们已经见过的系统头文件以外，这个代码还需要<cctype>头文件，这个头文件定义了 isspace。说得更通俗点，这个头文件中定义了有用的函数来让我们处理独立的字符。在 cctype 开头的那个 c 提醒我们，ctype 工具是 C++从 C 继承过来的。

split 函数有惟一的一个参数，这个参数是对一个名为 s 的常量字符串的引用。因为我们是从 s 中复制单词的，所以 split 并不需要修改这个字符串。正如我们在§4.1.2 中看到的那样，我们可以传递一个常量引用从而避免复制字符串所产生的耗费，同时，这样做还可以确保 split 将不会修改它的参数。

我们首先定义了 ret，函数用它来保存来自输入字符串的单词。接下来的两条语句定义并初始化了第一个索引 i。正如我们在§2.4 中看到的那样，对一个字符串的索引来说，string::size_type 是它的一个合适的类型名。因为我们会多次用到这个类型，所以，像我们在§3.2.2 中所做的那样，我们先要为这个类型定义一个短一点的替代名，这样我们就可以简化后面的声明了。我们用索引 i 来找出每一个单词的开头，为此，我们要在输入字符串中推进 i——而且是每次推进一个单词。

在最外层的 while 语句中进行的检测确保了，一旦我们处理完了输入的最后一个单

们就会马上停止。

在这条 while 语句的内部，我们先为两个索引定位。首先，我们找出 s 中的第一个非空白字符，这个字符处于当前的 i 所指示的位置之上或之后。因为在输入中可能有多个空白字符，所以我们对 i 加 1，直到它所指示的字符不是空白为止。

下面的语句包含了相当多的知识点：

```
while (i != s.size() && isspace(s[i]))
    ++i;
```

isspace 函数是一个谓词，它有一个 char 类型的参数并且它会返回一个值，这个返回值指示了参数所代表的字符是否为空白。&&运算符检查它的两个操作数，如果这两个操作数中的任一个为假（false），则检查失败。在这个表达式中，如果 i 不等于 s 的长度（这意味着我们还未到达字符串的尾部）并且 s[i]是一个空白字符的话，那这个操作将会成功。这样的话，我们将对 i 加 1，然后继续检查。

正如我们在§2.4.2.2 中描述的那样，逻辑&&运算符使用了短路策略来计算它的操作数。跟我们较早的例子不同，这一个依赖于&&的短路性质：二元逻辑操作（运算符&&和||）在执行时会首先检测它的左操作数。如果单凭这个检测就可以确定整个结果，那么它们就不再对右操作数进行运算。就&&而言，只有在第一个条件为真的情况下，第二个条件才会被计算。因此，while 语句中的条件执行时首先检测 i != s.size()是否为真，只有这个检测成功了，它才会用 i 来查看 s 中的字符。当然，如果 i 与 s.size()相等的话，那么就表示没有要检查的字符了，这样我们就会退出这个循环。

一旦结束了这条 while 语句，我们就会知道，到底是 i 指示了一个非空白字符还是在找到一个这样的字符之前输入就已经被耗尽了。

假定 i 仍然是一个有效的索引，那么下一条 while 语句将查找一个空格，这个空格结束了当前的单词。我们首先创建另一个索引 j 并用 i 的值来对它进行初始化。这条 while 语句

```
whlie (j != s.size() && !isspace(s[j]))
```

的行为跟前面的那条类似，只是这一条会在碰到了一个空白字符的时候停止。跟之前一样，我们首先要确保 j 的值仍然是在值域中。如果是这样的话，我们就调用 isspace 来判断索引为 j 的字符。这一次我们用**逻辑非**运算符！来对 isspace 的返回值进行求反运算。换句话说，我们希望，如果 isspace(s[j])的结果为假，那这个条件就为真。

在完成了这两个内部的 while 循环之后，我们就知道，我们到底是找到了另一个单词还是在查找单词的时候读尽了输入。如果我们耗尽了输入，那么 i 和 j 两者都会等于 s.size()。否则，我们就已经找到了一个单词，而且我们必须把这个单词放进 ret 中：

```
// 如果找到了一些非空白字符
if (i != j) {
    // 从 i 开始复制 s 的 j - i 个字符
    ret.push_back(s.substr(i, j - i));
```

```
    i = j;
}
```

在调用 push_back 的时候我们使用了 string 类的一个名为 substr 的成员，在此之前我们还没有见过这个成员。它的参数是一个索引和一个长度，它的作用是创建一个新字符串，而这个新字符串包含了来自原始字符串的一个副本。substr 从第一个参数给定的索引开始复制字符，复制的字符个数由第二个参数指定。我们取出的子字符串从索引 i 指示的字符开始，i 表示的字符是我们刚刚找到的单词的第一个字符。我们从索引 i 所指示的字符开始对 s 中的字符进行连续的复制，直到我们复制完了（半开）区间[i,j)所表示的字符为止。别忘了，从§2.6 中我们可以知道，一个半开域中的元素个数就是上下限之间的差。在这里我们看到了，我们将会刚好复制 j－i 个字符。

5.7 测试 split 函数

在编写了函数之后，我们将对它进行测试。对此的最简单的办法是遍写一个程序来读一行输入，然后把这行输入传递给 split 函数。这样的话，我们就可以输出由 split 返回的 vector 的内容了。用这样的一个函数来检查输出是很容易的，同时我们还可以根据它验证 split 函数是否生成了我们所期望的单词。

更为有用的是，这个测试函数产生的结果跟另一个程序所产生的一样，而另外的这个程序会从标准输入读单词并一行一个地输出这些单词。我们也可以编写后面的这个程序，然后对同样的输入文件分别运行它和我们的测试程序，这样我们就可以验证这两个程序是否会产生同样的输出。如果确实是这样的话，我们就能对我们的 split 函数相当放心了。

让我们先为 split 编写测试程序：

```
int main()
{
    string s;
    // 读并分割每一行输入
    while (getline(cin, s)) {
        vector<string> v = split(s);

        //输出 v 中的每一个单词
        for (vector<string>::size_type i = 0; i != v.size(); ++i)
            cout << v[i] << endl;
    }
    return 0;
}
```

这个程序需要一次读入一整行输入。值得庆幸的是，string 库提供了函数 getline 给我们使用，我们用 getline 来读输入直到行尾。它有两个参数，第一个参数是一个输入流（istream），它会从这个流读数据；第二个参数是一个字符串引用，我们把读到的数据存储在这个字

用中。跟平常一样，getline 返回一个引用给那个我们从中读数据的流，这样我们就可以在一个条件中测试这个流了。如果我们到达了文件结尾或遇到了无效输入，那 getline 的返回值就会指示失败，从而会跳出 while 语句。

只要我们能够读一行输入，就把这一行数据存储在 s 中，然后把它传递给 split，而 split 的返回值则会存储在 v 中。接下来，我们循环访问 v，把向量 v 中的每一个字符串都输出到独立的一行中。

假定我们已经添加了适当的#include 指令，在这些指令中，有一个包含了我们自己的头文件，这个头文件含有 split 的一个声明。这样的话，我们就能运行这个函数并亲眼验证它和 split 了，我们会看到它和 split 做的工作是否符合我们的要求。甚至，我们还可以做得更好，我们可以将这个输出跟另一个让库来做这些工作的程序的输出比较一下：

```
int main()
{
    string s;
    while (cin >> s)
        cout << s << endl;
    return 0;
}
```

这个程序和之前的那个应该会产生同样的输出。在这里，我们用 string 输入运算符来把输入流分割成一系列的单词，在程序中我们每行输出一个这样的单词。对同样复杂的输入分别运行这两个程序，我们就会很清楚地知道，split 函数是确实行得通的。

5.8 连接字符串

在§1.2 和§2.5.4 中，我们编写了一个程序来输出了位于一星号图中央的姓名。然而，我们还从未真正地创建过一个字符串来保存我们程序的输出。相反，我们要分别输出图形各部分——而且是一次输出一部分，然后我们还要用输出文件来把这些片段组合成一幅图案。

现在，为了建立一个数据结构来表示整个被框起的字符串，我们将回顾一下这个问题。这个程序是我们所钟爱的一个实例的简化版本，这个实例叫做是“字符图案”。一幅字符图案是一个矩形的可显示的字符阵列。它是发生在现实应用中的事物的一个简化——就此而言，这里所说的应用是以位图为基础的。简化的方法是：用字符串来代替位，并写输出到普通文件中，而不是把它在图形硬件上显示出来。这个问题首先在 Stroustrup 所著的<<C++ programming Language>>（Addison-Wesley,1986，第一版）的一条习题中出现，而我们则在<<Ruminations on C++>> (Addison-wesley,1997)一书中对它进行了较为深入的探讨。

5.8.1 为图案装框

在这一节中，我们将探讨一下字符图案问题的一个特殊的变化：我们想每行一个地

储在一个 vector<string>类型的向量中的所有单词，并用一个边框来围住这些字符串。我们将沿着左边框把这些字符串排列起来并在星号图和我们输出的单词之间留一个空格。

假定 vector<string>类型的向量 p 包含了字符串“this is an”、“example”、“to”、“illustrate”以及“framing”。同时我们需要用一个名为 frame 的函数，对这个函数的调用 frame(p)会产生一个 vector<string>类型的值，在输出的时候，这个值包含的元素是：

```
**************
* this is an *
* example    *
* to         *
* illustrate *
* framing    *
**************
```

注意到边框是一个对齐的矩形——尽管字符串本身可能有不同的长度。这个事实表明，我们需要一个函数来找出向量中最长的字符串的长度。就让我们先完成这个工作吧：

```
string::size_type width(const vector<string>& v)
{
    string::size_type maxlen = 0;
    for (vector<string>::size_type i = 0; i != v.size(); ++i)
        maxlen = max(maxlen, v[i].size());
    return maxlen;
}
```

这个函数将对向量进行循环访问，它把 maxlen 设置成目前我们已经见到的最大的长度。当我们退出循环的时候，maxlen 将保存了 v 的最长的字符串的长度。

在 frame 函数中惟一一个复杂的地方是它的接口。我们知道，它将对一个 vector<string>类型的向量进行操作，但我们应该采用什么样的返回类型呢？对这个问题有一个简便的解决方法，我们可以让函数创建一个新图案而不是修改原来的图案：

```
vector<string> frame(const vector<string>& v)
{
    vector<string> ret;
    string::size_type maxlen = width(v);
    string border(maxlen + 4, '*');

    // 输出顶部的边框
    ret.push_back(border);

    // 输出内部的行，每一行都用一个星号和一个空格来框起
    for (vector<string>::size_type i = 0; i != v.size(); ++i) {
        ret.push_back("* " + v[i] +
                      string(maxlen - v[i].size(), ' ') + " *");
    }
```

```
    // 输出底部的边框
    ret.push_back(border);
    return ret;
}
```

我们说，函数并不会修改传递过来的图案，因此我们可以把参数声明为对常量的一个引用。函数将返回一个 vector<string>类型的向量，这个向量是在 ret 中建立的。我们首先算出每一个输出字符串的长度，然后我们创建了跟这个长度一样长的、全部由星号组成的字符串，我们用这个字符串来创建顶部和底部的边框。

这些边框比最长的字符串多了四个字符：右边框和左边框各占一个，另外的两个则是用来隔开边框和字符串的空白。从§1.2 的 spaces 的定义中（我们在§1.2 中解释了这个定义）我们可以得到一个语法上的提示。根据这个提示，我们把 border 定义成一个含有 maxlen + 4 个星号的字符串，然后我们调用 push_back 来把 border 的一个复件添加给 ret，这样就形成了顶部边框。

接下来，我们复制了那幅要被框起的图案。我们定义了索引 i，我们用 i 来循环地访问 v，直到我们复制了每一个元素为止。在调用 push_back 的时候，我们使用了 string 类中的+运算符，正如我们在§1.2 中所了解到的一样，这个运算符会把函数的参数连接起来。

为了构造输出行，我们把左边界和右边界以及我们想要显示的存储在 v[i]中的字符串连接了起来。在我们的连接操作中，第三个字符串 string(maxlen – v[i].size(), ' ')构造了一个未命名的临时字符串来保存适当数目的空格。在构造这个字符串的时候，我们采用的方式跟我们初始化 border 时所用的方式是一样的。我们把 maxlen 与当前字符串的长度相减从而获得了空格的数目。

有了这些知识，我们就能看出，push_back 的参数是一个新的字符串，它开头是一个星号，紧跟在星号之后的是一个空格，空格后是当前的字符串，再后面是足够的空格，这些空格使得当前的字符串跟最长的字符串具有同样的长度，最后是另外的一个空格以及另一个星号。

我们剩下来的所有工作是添加底部边框并返回。

5.8.2 纵向连接

我们之所以会觉得字符图案是一个有趣的例子，是因为一旦我们有了这些图案，我们就能对它们作出处理。我们刚才看到了一个操作——为一幅图案装框。另一个操作是连接，我们既可以做纵向的连接，也可以做横向的连接。在这里，让我们先看一下纵向连接，横向连接则是下一小节我们所要讲述的内容。

我们可以按行把图案自然地组织起来，也就是说，我们可以用一个 vector<string>类型的向量来表示一幅图案，这个向量中的每一个元素都是图案的一行。因此，把两幅图案连接起来的工作看起来是很简单的：我们只需要把表示它们的两个向量连接起来就行了。这样做会让两幅图案沿着它们的左边界排列起来。这是一个定义纵向连接的合理方法。

惟一的问题是，我们虽然有字符串的连接操作，但是却找不到一个向量的连接操作

我们必须自己来完成这个工作：

```
vector<string> vcat(const vector<string>& top,
                    const vector<string>& bottom)
{
    // 复制顶部图案
    vector<string> ret = top;

    // 复制整个底部图案
    for (vector<string>::const_iterator it = bottom.begin();
         it != bottom.end(); ++it)
        ret.push_back(*it);

    return ret;
}
```

这个函数所使用的工具都是我们已经见过的：我们定义 ret 来作为 top 的一个复制；把 bottom 的每一个元素都添加给 ret；然后，返回 ret 以作为函数的结果。

在这个函数中的循环执行了一个常见的操作，也就是，我们把一个容器的元素的复制插入到另一个容器中。在这个特殊的例子中，实际上是在添加元素，也可以把这个动作看成是在容器末尾插入它们。

因为这个操作是十分常见，所以库提供了一种无需用到循环的实现方法。我们无需编写

```
for (vector<string>::const_iterator it = bottom.begin();
     it != bottom.end(); ++it)
    ret.push_back(*it);
```

而可以用下面的语句来获得同样的效果：

```
ret.insert(ret.end() , bottom.begin() , bottom.end());
```

5.8.3 横向连接

通过横向连接方式，我们可以把两幅图案拼接成一幅新的图案。在这幅新图案中，两幅输入图案中的一幅构成了它的左半部分，而另一幅则构成了右半部分。在开始这个工作之前，我们需要先考虑一下，如果待连接的两幅图案的尺寸不同，那我们应该如何处理。对此，我们可以作出一个简单的决定，我们将沿着顶部边界把它们对齐。因此，输出图案中的每一行都是连接两幅输入图案的对应行的结果。我们必须填充左侧的图案以使它的行在输出图案中占用适当数量的空格。

除了填充左侧的图案之外，我们还要思考一下，如果那两幅待连接的图案的行数不同，那应该如何处理呢？例如，如果 p 保存了我们原始的图案，那我们可能会希望把 p 的原始值和带边框的 p 的结果值连接起来。也就是说，我们可能想用 hcat(p,frame(p))来构造：

```
this is an **************
example    * this is an *
to         * example    *
illustrate * to         *
framing    * illustrate *
           * framing    *
           **************
```

值得注意的是，左侧图案的行数比右侧的少。这个事实说明，我们将不得不填充输出的左侧以占据这些空行。如果是左侧的图案较长，那就只需要把它的字符串复制到新图案中；并没有必要填充（空的）右侧。

有了这些分析之后，就可以编写函数了：

```
vector<string> hcat(const vector<string>& left,
                    const vector<string>& right)
{
    vector<string> ret;

    // 对width(left)加1，在两幅图案之间留一个空格
    string::size_type width1 = width(left) + 1;

    // 用来遍历left和right的索引
    vector<string>::size_type i = 0, j = 0;

    // 循环操作直到我们查看完了两幅图案的所有行
    while (i != left.size() || j != right.size()) {
        // 构造新字符串来保存字符，这些字符来自两幅输入图案
        string s;

        // 如果右侧图案还有待复制的行，那就复制一行
        if (i != left.size())
            s = left[i++];

        // 填充至适当的长度
        s += string(width1 - s.size(), ' ');

        // 如果左侧图案还有待复制的行，那就复制一行
        if (j != right.size())
            s += right[j++];

        // 把s添加到我们正在创建的图案中
        ret.push_back(s);
    }

    return ret;
}
```

正如对 frame 和 vcat 所做的那样，我们首先定义我们将返回的图案。下一步工作是计算一个宽度——我们必须填充左侧的图案，让左侧图案的宽度跟这个宽度相同。这个宽度将比左侧图案本身的宽度大一，这样，在连接的时候我们就可以在两侧的图案之间留一个空格了。接下来，我们对两幅图案进行反复操作，我们从第一幅复制一个元素，并且在必要时对它进行填充，这个元素后面紧跟着第二幅图案的一个元素。

在函数中我们要注意的惟一复杂的一部分是，如果我们在耗尽一幅图案的元素之前就已耗尽了另一幅的，那应该采取什么对策。我们的迭代器会循环工作直到我们已经复制完了每一个输入向量中的所有元素。因此，while 循环会持续至两个索引都到达了它们所对应的图案的结尾为止。

如果我们还没有耗尽 left，那我们就把它当前的元素复制到 s 中。不管我们有没有从 left 复制到元素，我们接下来都会调用字符串复合赋值运算符+=来把这个输出填充至适当的宽度。string 库定义的复合赋值运算符会以你所期望的方式去操作：它把右操作数添加给它的左操作数并把结果存储在左操作数中。当然，这里的“添加”意味着字符串的连接。

我们把 s.size()减去 width1 从而判定待填充的字符数。我们知道，s.size()要么是我们从 left 中复制的字符串的长度要么是零——因为根本没有项目可复制。对于第一种情况，s.size()将大于零而小于 width1，这是因为要把最长的字符串的长度加 1 以占去两幅图案之间的一个空格。这样的话，我们将给 s 添加一个或多个的空白。如果 s.size()等于零，那我们将填充整个输出行。

复制和填充完左侧图案的字符串之后，剩下来我们所要做的就是用右侧图案的元素来填充这个字符串——如果 right 中仍然有一个元素可以复制的话。不管我们有没有从 right 增加了一个值，我们都将把 s 推进输出向量中，然后，继续操作到已经处理完两个输入向量为止——别忘了我们要向调用程序返回我们所创建的图案。

非常值得注意的是，我们程序的正确的性能表现依赖于 s 是 while 循环中的局部变量这个事实。因为 s 是在 while 的内部声明的，所以它在每次创建时都是一个空值，并且在每执行了循环一次后它都会被销毁。

5.9 小结

容器与迭代器：标准库的设计是很科学合理的，对不同容器的相似操作有着同样的接口和语义。到目前为止，我们所用到的容器都是顺序容器。在第 7 章中我们将会看到，库也提供了关联容器。所有的这些顺序容器和 string 类型都支持如下的操作：

container<T>::iterator

container<T>::const_iterator

表示这个容器的迭代器的类型名。

container<T>::size_type

类型名称，用来保存这个容器可能存在的最大实例的长度。

c.begin()
c.end()　　指向容器第一个元素和紧跟在最后一个元素之后的那个位置的迭代器。

c.rbegin()
c.rend()　　对于允许以逆序访问其元素的容器，表示的是指向容器最后一个元素和位于第一个元素之前的那个位置的迭代器。

container<T>　c;
container<T>　c(c2);
　　定义一个容器 c。如果给定 c2，那么 c 是 c2 的一个复件；否则 c 为空。

contanier<T> c(n);
　　定义一个有 n 个元素的容器 c，c 根据 T 的类型而被数值初始化（§7.2）。如果 T 是一个类类型，那么这个类型将控制元素的初始化方式。如果 T 是一个内部算术类型，那么元素将被初始化成 0。

contanier<T> c(n,t);
　　定义一个有 n 个元素的容器 c，c 的元素是 t 的复件。

contanier<T> c(b,e);
　　创建一个容器，这个容器保存了位于区间[b,e)中的迭代器所指示元素的一个复件。

c=c2;　　用容器 c2 的一个复件来替换容器 c 的内容。
c.size()　　返回 c 的元素个数，返回值的类型是 size_type。
c.empty()　　一个判定，被用来指示 c 是否没有元素。

c.insert(d,b,e)
　　复制由位于区间[b,e)中的迭代器所指示的元素，并且把它们插入到 c 中位于 d 之前的位置中。

c.erase(it)
c.erase(b,e)　　从容器 c 中删除由 it 指示或由[b,e)指示的元素。对于表，这个操作是快速的；但对于向量和字符串则可能比较慢，这是因为，对于这些类型，它包括了复制位于被删除位置之后的所有元素的动作。对于表来说，指向被删除元素的迭代器会失效。而对于向量和字符串，所有指向位于被删除元素之后的元素的迭代器都会失效。

c.push_back(t)　　在 c 的末尾添加一个元素，这个元素的值是 t。

支持随机访问的容器和 string 类型同时也提供了如下的操作：

c[n]　　从容器 c 中取出位于位置 n 的字符。

迭代器操作：

*it　　间接引用迭代器 it 来获取存储在容器中位于 it 所指示的位置的值。这个操作经常与“.”结合起来以获取类对象的一个成员。例如(*it).x

迭代器 it 所指示的对象的一个成员。* 的优先级比 . 的低而跟++和--的相同。

It->x　与(*it).x 等价，它返回通过间接调用迭代器 it 而获得的对象所指示的成员 x。其优先级与 . 运算符的相同。

++it

it++　对向量加 1，让它指示容器中的下一个元素。

b==e

b!=e　为了判断相等或不相等而比较两个向量。

string（字符串）类型提供了迭代器来支持跟向量的迭代器所支持的一样的操作。特别地，string 支持完全随机访问，在第 8 章我们会学到更多的关于这方面的知识。除了容器所能提供的操作之外，string 还提供了：

s.substr(i,j)　创建一个新的字符串来保存 s 的在区间[i,i+j)中的索引所指示的字符的一个复件。

getline(is,s)　从 is 读一行输入并把它存储在 s 中。

s+=s2　用 s+s2 来替代 s 的值。

vector(向量)类型提供了在所有的库容器中功能最为强大的迭代器，我们把它称作随机访问迭代器。在第 8 章我们将会对此有更深入的了解。

尽管我们已经编写的所有函数都依赖于对我们的向量元素的动态分配，但是，还是存在着一种预分配元素的技术；同时我们还可以找到一种技术来控制向量的分配，这种技术不需要额外的内存，这就避免了重复的内存分配所带来的系统开销。

v.reverse(n)　保留空间以保存 n 个元素，但不对这些元素进行初始化。这个操作不会改变容器的大小。它仅仅会影响向量为了响应对 insert 或 push_back 的重复调用而分配内存的频率。

v.resize(n)　给 v 一个新长度，这个长度等于 n。如果 n 比 v 的当前长度小，那么，在这个向量中位于位置 n 之后的元素会被删除掉。如果 n 比当前长度大，那新的元素会被添加到向量中，而且这些元素会被初始化成与 v 的元素类型相应的值。

list 类型是为了高效地在容器中任何位置插入和删除元素而被优化的。对 list 和 list 迭代器的操作包含了我们在§5.9 中描述的那些。此外，

l.sort()

l.sort(cmp)　使用适用于 list 元素类型的<运算符来排列 l 的元素；或者使用判定 cmp 来排列元素。

<cctype>头文件为处理字符数据而提供了有用的函数：

isspace(c)　如果 c 是一个空白字符则结果为 true(真)。

isalpha (c)　如果 c 是一个字母字符则结果为 true(真)。

isdigit (c)　如果 c 是一个数字字符则结果为 true(真)。

isalnum(c)	如果 c 是一个字母或数字则结果为 true(真)。
ispunct(c)	如果 c 是一个标点字符则结果为 true(真)。
isupper(c)	如果 c 是一个大写字母则结果为 true(真)。
islower(c)	如果 c 是一个小写字母则结果为 true(真)。
toupper(c)	产生一个等于 c 的大写字母。
tolower(c)	产生一个等于 c 的小写字母。

习题

5-0 编译、运行并测试本章中的程序。

5-1 设计和实现一个程序来产生一个置换索引。在一个置换索引中，每一个短语都是以这个短语的每一个单词作为索引的。因此，假如有如下的输入：

```
The quick brown fox
jumped over the fence
```

那么，输出将会是：

```
       The quick      brown fox
  jumped over the     fence
   The quick brown    fox
          jumped      over the fence
          jumped      over the fence
             The      quick brown fox
     jumped over      the fence
                      The quick brown fox
```

在 Aho、Kernighan 和 Weinberger 所著的《The AWK Programming Language》（Addison-Wesley,1988）一书中有一个很好的算法。它分三个步骤来处理这个问题：

1. 读入输入的每一行并对每一行输入产生一个轮转的集合。每一个轮转都把输入的下一个单词放到第一个位置上，并把原先的第一个单词旋转到短语的末尾。因此，输入的第一行所表示的短语的输出将会是：

```
The quick brown fox
quick brown fox The
brown fox The quick
fox The quick brown
```

当然，重要的是要知道最初的短语是在哪里结束，而轮转的开头又是从哪里开始的。

2. 对这些轮转集合排序。

3. 反向轮转并输出置换索引，其中包含了查找分隔符号、把短语重新连接到一起以及以正确的格式输出短语等操作。

5-2 编写一个新的程序来计算学生的成绩，要求使用向量来抽取不及格学生的记录

后再编写这个程序的另外一个版本，要求在这个版本中使用 list。对于 10 行、100 行和 10000 行的输入文件分别比较一下两个程序在性能上的差异。

5-3 使用一个 typedef，我们就能编写出上面的程序的一个既可以基于向量解决方案又可以基于 list 解决方案的版本。编写并测试程序的这个版本。

5-4 回顾一下在前面的练习中你所编写的驱动程序（注：并不是我们所认为通常意义下的设备驱动程序）。值得注意的是，我们完全可以编写出一个这样的驱动程序：对于 list 和向量，这个驱动程序仅仅在用来保存输入文件的数据结构的类型声明上有所不同。如果你的 list 驱动程序和 vector 驱动程序在其他的方面还有差别的话，那重写它们，让它们仅仅在这个声明上有所不同。

5-5 编写一个名为 center(const vector<string>&)的函数。这个函数返回一幅图案，在这幅图案中，原始图案的所有行都会被填充满（也就是，每一行都跟原图案中最长的行一样长）。要求在图案的左侧和右侧尽可能均匀地分布填充料。思考一下，图案要具有什么样的性质，这个函数才会是有用的？我们怎样能分辨出一幅图案是否具有这样的性质呢？

5-6 在§5.1.1 中的 extract_fails 函数删除了输入向量 v 的所有不及格的学生记录，重写这个函数，让它不用删除 v 的不及格记录而直接地把及格的学生记录复制到 v 的开头，然后使用 resize 函数来从 v 的尾部删除多余的元素。和§5.1.1 的中那个函数相比，这个版本的性能是怎样的呢？

5-7 假定我们有§5.8.1 中的 frame 函数，而且我们编写了下面的代码片段

```
vector<string> v;
frame(v);
```

描述一下在这个调用中出现的情况。特别地，跟踪一下 width 和 frame 函数的操作。现在，运行这个代码。如果结果跟你预料的有所不同，那就先思考一下，为什么你的预料会和程序有出入，然后改变你的思考方式或者改写程序，让你的预期能够跟程序吻合。

5-8 在§5.8.3 的 hcat 函数中，如果我们在 while 的作用域之外定义 s 的话，那会怎样呢？重写并运行这个程序以证实你的推测。

5-9 编写一个程序来输出在输入中的单词，输出的格式是：先输出全部的小写单词，然后再输出大写单词。

5-10 回文是指一种顺读和倒读都一样的单词。编写一个程序，让它找出一个单词集中的所有回文并且找出最长的回文。

5-11 在文本处理工作中，有时候，了解一下一个单词中是否有上行字母或下行字母是很有必要的。上行字母是向上超出了文本行的小写字母；在英文字母表中，字母 b、d、f、h、k、l 以及 t 都是上行字母。同样地，下行字母是向下超出了文本行的小写字母；在英文字母表中，字母 g、j、p、q 和 y 都是下行字母。编写一个程序来判断在一个单词中是否包含有上行字母或下行字母。扩充这个程序，用它来找出既没有上行字母也没有下行字母的单词集中的最长的单词。

第 6 章

使用库算法

正如我们在第 5 章所看到的那样，许多的容器操作都可以应用到不止一种类型的容器中。例如，向量、字符串和表允许我们通过调用 insert 来插入元素以及调用 erase 来删除元素。对于每一种支持它们的类型，这些操作都有着同样的接口。就此而言，许多容器操作也可以应用到 string 类中。

所有的容器（包括 string 类）都提供了配套的迭代器类型，这些类型可以让我们通过容器并同时检查它的元素。此外，库确保了所有提供同一个操作的迭代器都通过同样的接口来实现其功能。例如，我们能使用++运算符来让任何类型的迭代器从一个元素推进到下一个元素中；我们也能用*运算符来访问与任何类型的迭代器相关联的元素；等等。

在本章中，我们将看到库是如何利用这些公用接口来提供一个标准算法集合的。通过使用这些算法，我们就能够避免一次又一次地编写（特别是重写）同样的代码。更为重要的是，我们能编写出比我们用其他方式编写的更短小、更简单的程序——有时这个优势甚至是令人惊讶的。

跟容器和迭代器一样，算法也使用了一致的接口协定。这个一致性可以让我们首先掌握一些算法然后让我们在有必要的时候把这些知识应用到其他算法中。在本章中，我们将使用几个库算法来解决与处理字符串和学生成绩有关的问题。沿着这个思路，我们将覆盖算法库中大部分的核心概念。

除非我们另外说明，否则<algorithm>头文件定义了我们在本章中介绍的所有算法。

6.1 分析字符串

在§5.8.2 中，我们使用了一个循环来连接两幅字符图案：

```
for (vector<string>::const_iterator it = bottom.begin();
      it != bottom.end(); ++it)
    ret.push_back(*it);
```

我们注意到这个循环等价于在 ret 的末尾插入 bottom 的元素的一个复制，向量直接提供了下面的一个操作：

```
ret.insert(ret.end(), bottom.begin(), bottom.end() );
```

这个问题还有一个更为一般的解决办法：我们可以把复制元素的概念和在一个容器的末尾插入元素的概念分离开来，形式如下所示：

```
copy(bottom.begin() , bottom.end() , back_inserter(ret));
```

在这里，copy 是一个泛型（generic）算法的例子，而 back_inserter 则是一个迭代器适配器的例子。

泛型算法是一个不属于任何特定类别容器的算法，相反，它会从它的参数类型获得关于如何访问它所使用数据的提示。标准库的泛型算法通常在它们的参数间采用迭代器来处理基本容器的元素。例如，copy 采用了三个迭代器，我们分别把它们称作 begin、end 和 out。我们用它们把区间[begin,end)中的所有元素复制到一连串从 out 开始的元素中，并在必要时对 out 所属的容器进行扩展。换句话来说，

```
copy(begin, end, out);
```

的作用与下面的语句一样：

```
while (begin != end)
    *out++ = *begin++;
```

它们之间的不同之处在于：while 循环体修改了迭代器的值，而 copy 则不会进行修改。

在我们开始描述迭代器适配器之前，我们首先应该注意到这个循环依赖于增量运算符的**后缀**版本的应用。这些运算符跟我们到目前为止所使用过的前缀版本不同，begin++返回 begin 的原始值的一个复件，其副作用是 begin 中存储的值增加 1。换句话说，

```
it = begin++;
```

等价于

```
it = begin;
++begin;
```

增量运算符的优先级与*的相同，并且它们都是右结合的，这就意味着*out++的意义与*(out++)的一样。因此，

```
*out++ = *begin++;
```

与下面更详细的形式等价：

```
{ *out = *begin; ++out; ++begin; }
```

让我们回到对**迭代器适配器**的讨论，迭代器是一种函数，它会产生有着与其参数相关的属性的迭代器以做他用。迭代器适配器是在<itertor>中定义的。最常用的迭代器适配器是 back_inserter，它用一个容器作为它的参数并产生一个迭代器，在生成的迭代器被用作一个目的地的时候，它会向容器末端添加数值。例如，在 back_inserter(ret)被用作目的地的时

会为 ret 添加一个元素。因此，

```
copy(bottom.begin() , bottom.end() , back_inserter(ret) );
```

就复制了 bottom 中的所有元素并且把它们添加到 ret 的末尾。在这个函数结束之后，ret 的长度将增加 bottom.size()。

值得注意的是我们不可以这样调用

```
// 错误——ret 不是一个迭代器
copy(bottom.begin(), bottom.end(), ret);
```

这是因为 copy 的第三个参数应该是一个迭代器而不应是一个容器。我们也不可以这样调用

```
// 错误——在 ren.end()没有元素
copy (bottom.begin() , bottom.end() , ret.end() );
```

后面的这个错误隐藏得很深，因为程序将会通过编译。当你尝试运行它的时候它会做些什么那又完全是另外一回事了。copy 要做的第一件事是赋一个值给 ret.end()中的元素。由于在此位置并不存在一个元素，系统环境将做些什么就全凭我们的猜测了。

为什么 copy 是按这种方式设计的呢？这是因为，如果把复制元素与扩展一个容器的概念分离开来，那么程序员就可以选择使用哪些操作了。例如，我们可能会希望不修改容器的大小而直接在已有元素之上复制元素。在§6.2.2 中我们将看到另外的一个例子，我们可能希望用 back_inserter 来给一个容器添加元素，而不仅仅是复制另一个容器的元素。

6.1.1 另一个实现 split 的方法

另外一个我们能更直接地使用标准库算法的函数是 split，我们已经在§5.6 中见过这个函数了。编写这个函数时我们需要着重考虑的是索引的处理办法，而这些索引是用来为输入行中的每一个单词定界的。我们可以用迭代器来替代索引，也可以用标准库算法来为我们完成大部分的工作：

```
// 如果参数是空白区域则为 true(真)，否则为 false
bool space(char c)
{
    return isspace(c);
}

// 如果参数是空白区域则为 false,否则为 true
bool not_space(char c)
{
    return !isspace(c);
}
```

```
vector<string> split(const string& str)
{
    typedef string::const_iterator iter;
    vector<string> ret;

    iter i = str.begin();
    while (i != str.end()) {
        // 忽略前端的空白
        i = find_if(i, str.end(), not_space);

        // 找出下一个单词的结尾
        iter j = find_if(i, str.end(), space);

        // 复制在[i,j)中的字符
        if (i != str.end())
            ret.push_back(string(i, j));
        i = j;
    }
    return ret;
}
```

我们在这个代码中使用了许多新的功能，因此，我们需要对这个代码进行一些解释。我们应牢记的概念是，它实现了跟原先一样的算法，它也是使用 i 和 j 来为 str 中的每个单词定界的。一旦找到了一个单词，我们就从 str 复制它，然后把这个复件添加到 ret 的末尾。

这一次 i 和 j 是迭代器而不是索引。我们用 typedef 来简化迭代器类型，这样我们就能使用 iter 来代替那个较长的 string::const_iterator 了。虽然字符串类型不支持全部的容器操作，但是它确实支持迭代器。因此，我们能用这些标准库算法来处理一个字符串的字符，这就跟我们能用它来处理向量的元素一样。

我们在这个例子中使用的算法是 find_if。这个算法的头两个参数是指示一个序列的迭代器；第三个参数则是一个谓词，它会检测自己的参数然后返回 true 或 false。find_if 函数会对这个序列中的每个元素都调用这个谓词，在找到一个让谓词产生 true 值的元素的时候就会停止调用。

标准库提供了一个 isspace 函数来让我们检查一个字符是否是空格。这个函数是一个重载的函数，这样它就可以处理使用其他字符类型（比如 wchar_t）的语言，例如日语。把一个重载函数作为一个参数而直接传递给一个模板函数并不是件容易的事情。其中的困难是，编译器并不知道我们所指的是哪一个版本的重载函数，而之所以会这样是因为我们没有提供让编译器用来选择一个版本的任何参数。从而，我们需要编写我们自己名为 space 和 not_space 的谓词来解释我们所指的是哪个版本的 isspace。

对 find_if 的第一次调用是为了查找第一个非空白字符，这个字符是一个单词的开头。别忘了，一个或多个的空格可能是一行的开始，或者可能是用来把输入中相邻的单词分隔开的，我们并不希望在输出中包含这些空格。

在第一次调用了 find_if 之后，i 将指示第一个非空白字符，这个字符是一个单词的开始——如果在 str 中有一个单词的话。我们在对 find_if 的下一次调用中用 i 来查找在[i,str.end())中的第一个空格。如果 find_if 不能找到一个满足这个谓词的值，那么它将返回它的第二个参数——在这个例子中，返回的就是 str.end()。因此，j 将被初始化以指示那个把 str 中的下一个单词和本行其余部分分隔开的空白；或者，如果我们到达了最后一个单词的话，j 就将等于 str.end()。

到此，i 和 j 界定了 str 中的一个单词。剩下来我们要做的是使用这些迭代器来把 str 中的数据复制到 ret 中。在 split 的较早前的版本中，我们使用 string::substr 来创建这些复制。然而，split 的那个版本是对索引而不是对迭代器进行操作的，同时我们也找不到对向量进行操作的 substr 版本。相反，我们用我们所拥有的迭代器直接构造了一个新的字符串。为此我们使用了一个表达式 string(i, j)，这个表达式跟我们在§1.2 中所解释的 spaces 定义有些相似。我们现在的这个例子构造了一个字符串，它是在区间[i,j)中的字符的一个复件。我们把这个新的字符串推入 ret 的末端。

我们有必要指出，我们在这个程序版本中省略了对索引 i 和 str.size()的相互大小的测试。而且我们没有显式检测迭代器和 str.end()是否相等。这样做是因为，这些库算法是被用来适度处理那些传递一个空区间的调用的。例如，在某个时刻对 find_if 的第一次调用将把 i 设置成 str.end()的返回值，但我们并没有必要在把 i 传递给 find_if 的第二次调用之前而对它进行检测。这是因为，find_if 会查看区间[i,str.end())而且它将返回 str.end()以指明并未发现匹配。

6.1.2 回文

我们能用库来简便地解决另一个关于字符处理的问题，而这个问题就是判断一个单词是不是一个回文。回文是顺读和逆读都一样的单词。例如，“civic”、“eye”、“level”、“madam”以及“rotor”都是回文。

下面是一个使用库的小型的解决方案：

```
bool is_palindrome(const string& s)
{
    return equal(s.begin(), s.end(), s.rbegin());
}
```

我们在这个函数体的 return 语句中调用了 equal 和 rbegin 成员函数，这两个函数都是我们以前没有见过的。

跟 begin 一样，rbegin 返回一个迭代器，但是这一次，这个迭代器会从容器的最后一个元素开始，并且从后向前地逆向访问容器。

equal 函数比较了两个序列以判断它们是否包含有相等的值。跟平常一样，传递给 equal 的头两个迭代器指定了第一个序列。第三个参数则是第二个序列的起点。equal 函数假定第二个序列的长度与第一个相同，因此它并不需要一个结尾迭代器。我们传递 s.rbegin()以

二个序列的起点，这个调用的作用是从 s 的结尾向前逆向地比较数值。equal 函数将把 s 中的第一个字符和最后一个比较。然后它会比较第二个和倒数第二个，等等…… 这个行为正是我们所期望的。

6.1.3 查找 URL

作为字符处理的最后一个实例，让我们编写一个函数来查找被称为“统一资源地址(URL)”的万维网地址，这些地址被嵌入在一个字符串中。在使用这样的一个函数的时候，我们可以创建单独的字符串来保存一个文档的全部内容。这个函数将扫描这个文档并查找它所包含的所有的 URL。

一个 URL 是一个具有如下形式的字符序列：

```
protocol://resource-name
```

这里的 protocol（协议名称）仅仅包含字母，而 resource（资源名称）可以由字母、数字和某些标点字符组成。我们的函数将有一个字符串参数，函数会在这个字符串中查找://的实例。每当找到一个这样的实例时，我们就查找在它之前的 protocol-name 和跟在它后面的 resource-name。

我们希望我们的函数能够查找出在它的输入中的所有 URL，因此我们让它返回一个 vector<string>类型的值，每一个 URL 都是返回值中的一个元素。函数执行时会在这个字符串内移动迭代器 b 并查找字符://，而且查找到的://有可能会是一个 URL 的一部分。如果找到了这些字符，那们函数就往回查找 protocol-name 并向前查找 resource-name：

```
vector<string> find_urls(const string& s)
{
    vector<string> ret;
    typedef string::const_iterator iter;
    iter b = s.begin(), e = s.end();

    //检查整个输入
    while (b != e) {
        //查找一个或多个紧跟着://的字母
        b = url_beg(b, e);
        //如果查找成功
        if (b != e) {
            //获取此 URL 的其余部分
            iter after = url_end(b, e);

            //记住这个 URL
            ret.push_back(string(b, after));

            //将 b 向前推进并查找位于本行中的其他 URL
```

```
            b = after;
        }
    }
    return ret;
}
```

我们首先声明了向量 ret，我们用这个向量来保存我们所查找到的 URL，这些 URL 是通过用来定界的迭代器所取得的。我们必须编写 url_beg 和 url_end 函数，这两个函数将为我们找出输入中的任何 URL 的开头和结尾。这个 url_beg 函数负责识别是否有一个有效的 URL。如果有的话，它就返回一个指向相应协议名称的第一个字符的迭代器；如果它识别不到输入中的 URL，那么它将返回它的第二个变量（对这个例子而言是 e）来指示失败。

如果 url_beg 找到了一个 URL，那么我们的下一个任务是调用 url_end 来找出这个 URL 的结尾。这个函数将从给定的位置开始搜索，直到它到达了输入的结尾或者是到达了一个不可能作为 URL 的一部分的字符为止，然后它会返回一个指向位于这个 URL 的最后一个字符之后的那个位置的迭代器。

因此，在调用了 url_beg 和 url_end 之后，迭代器 b 就指示一个 URL 的开头，而迭代器 e 则指示了此 URL 的最后一个字符之后的那个位置：

在这个区间中，我们根据这些字符而构造一个新的字符串，然后把这个字符串推入 ret 的末尾。

剩下来我们所要做的是增加 b 值以便查找下一个 URL。因为 URL 不能相互交迭，所以我们把 b 设置为紧位于我们刚刚找到的 URL 的结尾之后的一个位置，然后我们继续 while 的循环直到查找完所有的输入。一旦循环终止，我们就向调用程序返回那个包含了 URL 的向量。

现在，让我们来考虑一下 url_beg 和 url_end。url_end 函数比较简单，因此我们先来介绍它：

```
string::const_iterator
url_end(string::const_iterator b, string::const_iterator e)
{
    return find_if(b, e, not_url_char);
}
```

这个函数所做的只不过是把它的工作交给库的 find_if 函数去完成，我们已经在§6.1.1 中使用过 find_if 了。我们向 find_if 传递了一个名为 not_url_char 的谓词，下面将编写这个谓词。如果传递过来的字符是不可能出现在一个 URL 中的，那么这个谓词返回 true：

```
bool not_url_char(char c)
{
    //除了字母数字以外，其他有可能出现在一个 URL 中的字符
    static const string url_ch = "~;/?:@=&$-_.+!*'(),";

    //看 c 是否能出现在一个 URL 中并返回求反的结果
    return !(isalnum(c) ||
             find(url_ch.begin(), url_ch.end(), c) != url_end());
}
```

尽管这个函数很小，不过我们在其中还是使用了不少新概念。首先是 static（静态）**存储类别说明符**。被声明为 static 的局部变量具有全局寿命，即其生存期贯穿了整个函数调用过程。因此，我们将仅仅在第一次调用 not_url_char 的时候构造并初始化字符串 url_ch，而随后的调用将使用第一次调用所构造的对象。因为 url_ch 是一个常量字符串，所以，一旦我们对它进行了初始化，那么它的值就不会被修改。

not_url_char 还使用了由<cctype>头文件定义的 isalnum 函数。这个函数检验它的参数是否是一个字母数字字符（一个字母或一个数字）。

最后，find 是另一个我们还没有使用过的算法。它与 find_if 相似，不过它没有调用谓词，相反，它会查找由它的第三个参数所给定的特定值。跟 find_if 一样，如果我们查找的值存在的话，那么函数就返回一个迭代器，这个迭代器指示了在给定的序列中第一次出现这个值的位置。如果找不到这个值，那么 find 就返回它的第二个参数。

有了手头上的这些信息，我们就能够理解 not_url_char 函数了。因为在返回之前我们要对整个表达式的值取反，所以，如果 c 是一个字母、一个数字或者是 url 中的任何字符的话，那么 not_url_char 将产生 false 值。如果 c 是任何其他值，那函数返回 true。

现在问题来了：我们应该如何实现 url_beg 呢？这个函数是比较杂乱的，因为它必须对下面这种可能会发生的情况进行处理：输入所包含的://有可能会位于一个根本不可能是有效 URL 的上下文中。实际上，我们可能已经有了一个列表，这个列表包含了可接受的 protocol-name，而且我们只查找列表中的 protocol-name。为了简单起见，我们增加了一个约束，我们必须确保在://分隔符之前有一个或多个的字母，而且在它后面至少有一个字符。

```
strig::const_iterator
url_beg(string::const_iterator b, string::const_iterator e)
{
    static cosnt string sep = "://";

    typedef string::const_iterator iter;

    // i 标记了查找到的分隔符的位置
    iter i = b;

    while ((i = search(i, e, sep.begin(), sep.end())) != e) {
```

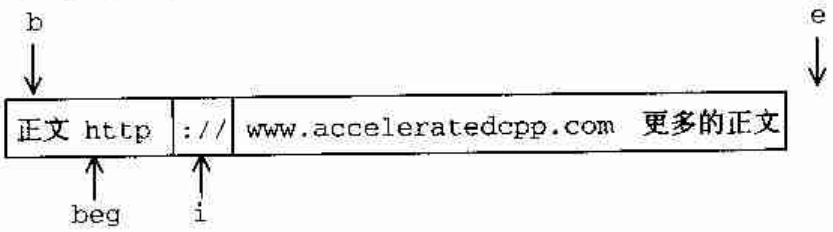
```
        // 确保分隔符不是本行中惟一的一个符号
        if (i != b && i + sep.size() != e) {
            // beg 标记协议名称的开头
            iter beg = i;
            while (beg != b && isalpha(beg[-1]))
                --beg;

            // 在分隔符前面及后面至少有一个字符吗?
            if (beg != i && i + sep.size() != e
                && !not_url_char(i[sep.size()]))
                return beg;
        }

        // 我们找到的分隔符不是一个 URL 的一部分
        if (i != e)
            i += sep.size();
    }
    return e;
}
```

我们很容易就可以编写出这个函数头。我们知道，我们将传递两个迭代器来指示所查看的区间，并且将返回一个迭代器来指示在这个区间中的第一个 URL 的开头——如果存在一个 URL 的话。我们同时还声明并初始化了一个局部变量，这个变量保存分隔符的组成字符，而我们将用这个分隔符来识别一个潜在的 URL。跟 not_url_char 函数中的 url_ch 一样，这个字符串是静态的(static)，而且还是一个常量(const)，因此，我们将不能修改这个字符串，并且它的值将仅仅是在第一次调用 url_beg 的时候被创建。

这个函数执行时把两个迭代器放置在由 b 和 e 界定的这个字符串中：



如果存在着一个 URL 的话，那迭代器 i 将指示这个 URL 分隔符的开头；如果存在着 protocol-name 的话，那 beg 将指示这个 protocol-name 的开头。

这个函数首先调用了 search 来查找分隔符，search 是一个我们之前没有使用过的库函数。这个函数有两对迭代器参数：第一对指示了我们要查找的序列，而第二对则指示了一个序列——我们希望为这个序列定位。跟其他的库函数一样，如果 search 失败，那么它将返回第二个迭代器。因此，在调用了 search 之后，i 会指示紧位于输入字符串末尾之后的那个位置或者指示了一个紧位于//之前的:。

如果我们找到了一个分隔符，那么下一步的工作就是获取构成了 protocol-name 的

母——如果存在 protocol-name 的话。我们所做的第一个检查是查看这个分隔符是否填满了整个输入行。如果是这样的话，那我们就知道我们无法找到一个 protocol-name。否则，我们应该尝试为 beg 定位。内部的 while 循环在输入中自后而前地逆向移动 beg，直到它碰到了一个非字母字母或者字符串的开头为止。我们在循环中使用了两个新概念：第一个概念是，如果一个容器支持索引，那么它的迭代器也会支持。换句话说，bed[-1]就是位置紧位于由 beg 所指示的字符之前的那一个字符。我们可以把 beg[-1]看成是*(beg - 1)的简写。在§8.2.6 中，我们将会对这样的迭代器进行更深入的理解。第二个新概念是那个在<cctype>中定义的 isalpha 函数，它检验了它的参数是不是一个字母。

如果我们能把这个迭代器推进至少一个字符的位置，那么我们就假定，我们找到了一个 protocol-name。在返回 beg 之前，我们还要检查是否有至少一个的有效字符跟在分隔符的后面。这个检查是更加复杂的。首先，我们要检查在输入中是不是有至少一个其他的字符，为此我们要比较 i+sep.size()和 e 的值。如果这两个值不相等，那就说明有这样的一个字符。然后，我们用 i[sep.size()]来访问这个字符，i[sep.size()]是*(i + sep.size())的一个简写。我们把这个字符传递给 not_url_char，从而检验它是否可以出现在一个 URL 中。如果这个字符无效，那么这个函数返回 true，因此，我们把返回值取反从而检查这个字符是否有效。

如果这个分隔符不属于一个 URL 的一部分，那么函数会推进 i，让它越过这个分隔符并继续向前查找。

我们在代码中使用了**减量运算符**。在§2.7 中的运算符表中我们提到过这个运算符，不过，在此之前我们并没有使用过它。它的工作方式与增量运算符类似但作用相反——它对它的操作数减 1。跟增量运算符一样，减量运算符也有前缀版本和后缀版本。我们在这里用到的前缀版本会对它的操作数减 1，然后返回新的值。

6.2 对计算成绩的方案进行比较

在§4.2 中，我们介绍过一个计算成绩的方案，我们在这个方案中所计算的学生总成绩在一定的程度上以他们的家庭作业成绩为基础。爱耍小聪明的学生可能会利用这个方案中的缺陷，他们可能不会上交所有的家庭作业——反正成绩较低的那一半家庭作业成绩对他们的总成绩是没有影响的。如果他们已经做了足够的家庭作业以确保可以有一个好成绩的话，那么，他们又何必把家庭作业都做完呢？

根据我们以往的经验，大多数的学生都不会利用这个特殊的漏洞。然而，在我们执教过的班级中确实有一个班的学生愉快且公开地做了这样的事情。我们想知道，从平均上来说，那些漏交家庭作业的学生和那些做了所有家庭作业的学生的总成绩上是否会有所不同。在我们考虑如何回答这个问题的时候，我们觉得，如果我们使用了两个可供选择的计算成绩的方案中的一个，那么这个问题的答案可能会是很有趣的：

- 使用平均值代替中值，把学生没有上交的作业成绩设定为零。

● 仅仅使用学生实际提交的作业成绩的中值。

对于这些计算成绩的方案中的每一个，我们都希望把提交了所有家庭作业的学生的中值成绩和那些漏交了一次以上作业的学生的中值成绩比较。我们要编写程序来处理下面两个不同的子问题：

1. 读所有的学生记录，把做了全部家庭作业的学生与其他的学生分隔开。
2. 对每一组中的所有学生分别使用每一个的计算成绩的方案，报告每一组的中值成绩。

6.2.1 处理学生记录

我们的第一个子问题是读入学生的记录并对这些记录进行分类。值得庆幸的是，我们已经拥有了一些代码，这些代码可以帮我们解决这一部分问题：我们能使用§4.2.1 中的 Student_info 类型和§4.2.2 中的相关的 read 函数来读学生数据记录。不过，我们还需要一个函数，这个函数会帮我们检查一个学生是否做了所有的家庭作业。编写这样的一个函数是一件很容易的事情：

```
bool did_all_hw(const Studend_info& s)
{
    return ((find(s.homework.begin(), s.homework.end(), 0))
            == s.homework.end());
}
```

这个函数访问了 s.homework 并查看存储在里面的所有值中是否有为 0 的。因为所有被提交的作业都至少会具有一个安慰成绩，所以一个为 0 的成绩意味着这个作业并没有提交。我们把 find 的返回值和 homework.end()相比较。跟往常一样，如果找不到所要查找的值，那么 find 将返回它的第二个参数。

有了这些函数，我们就可以很轻松地编写代码来读入学生记录并对这些记录进行分类了。我们将读每一个学生的记录，检查这个学生是否做了所有的家庭作业，最后我们还要把这个记录添加给两个向量中的一个。为了让我们的程序更容易理解，我们将把这两个向量命名为 did 和 didnt。

在处理向量的时候，我们要检查它们，看它们是否为空。这样的话，我们就知道我们的分析将会告诉我们一些有用的信息：

```
vector<Student_info> did, didnt;
Student_info student;

//读所有的记录，根据是否做了所有的家庭作业而对它们进行分类
while (read(cin, student)) {
    if (did_all_hw(student))
        did.push_back(student);
    else
        didnt.push_back(student);
}
```

```
//检查两个向量是否有包含数据
if (did.empty()) {
    cout << "No student did all the homework!" << endl;
    return 1;
}
if (didnt.empty()) {
    cout << "Every student did all the homework!" << endl;
    return 1;
}
```

在这里，我们惟一感到陌生是那个 empty 成员函数。如果容器为空，那么这个函数会产生 true；否则，它会产生 false。我们应该用这个函数来检查一个容器是否为空，而不应把容器长度和 0 比较，之所以要这样是因为，对于某些种类的容器，检查容器是否有元素比精确地算出元素个数具有更高的效率。

6.2.2 分析成绩

现在，我们知道了如何读入学生记录并把它们归类到 did 和 didnt 向量中。接下来我们要分析它们，这意味着我们要思考一下如何去组织我们的分析。

我们要完成三个分析，每一个分析都有两个部分，这两部分分别分析了做了所有家庭作业的学生和未全部完成作业的学生。因为我们的每一个分析都是对两个数据集合进行的，所以我们自然会希望让每一个分析都具有它自己的函数。然而，有一些操作——例如用一个公用的格式来报告输出——是我们打算对一对分析而不是对独立的分析进行的。很明显，我们也希望在一个函数中同时输出每一对分析的结果。

麻烦的是，我们希望调用那个输出分析结果的函数三次，也就是每一个分析一次。我们还想让这个函数调用适当的分析函数两次，一次为 did 对象而另一次则是为 dinnt 对象。然而，我们希望这个报告输出的函数在每次我们调用它的时候都调用一个不同的分析函数！我们应如何来安排呢？

最简单的解决方法是定义三个分析函数，并把每一个分析函数当作一个参数传递给这个报告函数。别忘了，我们已经用过这样的参数了。例如，在§4.2.2 中我们就把 compare 函数传递给库的 sort 例程。这样的话，我们希望输出例程具有五个参数：

- 接收输出的流
- 一个表示分析名称的字符串
- 用于分析的函数
- 两个参数，每一个参数都是一个向量，我们希望对这两个向量进行分析。

例如，让我们假定，查找中值的第一个分析是由一个名为 median_analysis 的函数完成的。那么，我们希望执行下面的函数来为每一组学生报告结果：

```
write_analysis(cout, "median", median_analysis, did, didn't);
```

在定义 write_analysis 之前，让我们先来定义 median_analysis。我们将给这个函数一个用来保存学生记录的向量，同时我们还将根据那个普通的计算成绩的方案来计算学生的成绩并返回这些成绩的中值。我们把这个函数定义如下：

```
// 这个函数并不十分有效
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(),
            back_inserter(grades), grade);
    return median(grades);
}
```

虽然这个函数乍看上去可能是难于理解的，但是我们只不过是在函数中使用了一个新的名为 transform 的函数概念。这个函数的参数是三个迭代器和一个函数。开头的两个迭代器指定了待转换元素的区间；第三个迭代器是一个目的地，它将保存函数的运行结果。

在我们调用 transform 的时候，我们要确保这个目的地有足够的空间以保存来自输入序列的值。在本例中，这不成问题，因为我们是通过调用 back_inserter (§6.1)从而获得这个目的地的，transform 的结果会被添加到 grades 中，而 grades 将在必要时自动增长以容纳这个结果。

transform 的第四个参数是一个函数，transform 将这个函数应用于输入序列的每个元素中以获得输出序列中的对应元素。因此，当我们在这个例子中调用 transform 的时候，这个调用的作用就是把 grade 函数应用于 students 的每一个元素，并且把每一个成绩添加到那个名为 grades 的向量中。如果我们拥有了所有这些学生的成绩，我们就调用在§4.1.1 中定义的 median 来计算它们的中值。

这个例子惟一的问题是：正如那个注释所示，这个函数并不十分有效。

函数不太有效的一个原因是，grade 函数有几个重载版本。编译器并不知道应该调用哪一个版本，因为我们没有给 grade 任何参数。我们知道，我们想调用的是§4.2.2 中的那一个版本，但是我们要找到一个办法来告知编译器这样做。

另外的一个原因是，如果有学生根本没做过家庭作业的话，那么 grade 函数将引发一个异常，而且 transform 函数不会对异常做任何的处理动作。如果发生了一个异常，那 transform 函数将在发生异常的地方终止执行，同时控制权将返回给 median_analysis。因为 median_analysis 也不会对异常进行处理，所以这个异常将继续向外传播。结果是，这个函数也将过早地退出、把控制权传递给它的调用程序……等等——直到控制权传给了一个适当的 catch。如果并不存在着这样的 catch，就好像在这个例子中一样，程序本身就会终止，而且那条被引发的消息会被打印（或者不会打印，这取决于具体的系统环境）。

我们可以编写一个辅助函数来解决这两个问题，这个辅助函数将在 grade 内执行

并且处理异常。因为我们是明确地调用 grade 函数而不是把它作为一个参数传递的，所以编译器能够理解我们所指的是哪一个版本：

```
double grade_aux(const Student_info& s)
{
    try {
        return grade(s);
    } catch (domain_error) {
        return grade(s.midterm, s.final, 0);
    }
}
```

这个函数将调用§4.2.2 中的那个 grade 版本。如果发生异常，我们将捕获（catch）它并调用§4.1 中的那个版本的 grade，这个版本有三个 double 类型参数，这三个参数表示的是考试成绩和总体家庭作业成绩。因此，我们将假定那些根本没做家庭作业的学生的家庭作业成绩是 0，但是他们的考试成绩依然有效。

现在，我们就可以重写那个分析函数并使用 grade_aux 了：

```
// 这个版本性能良好
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(),
            back_inserter(grades), grade_aux);
    return median(grades);
}
```

既然我们已经知道了一个分析例程是什么回事，那么，现在就让我们来定义 write_analysis 吧。write_analysis 使用一个分析例程来比较两个学生数据集合：

```
void write_analysis(ostream& out, const string& name,
                    double analysis(const vector<Student_info>&),
                    const vector<Student_info>& did,
                    const vector,Student_info>& didnt)
{
    out << name << ": median(did) = " << analysis(did) <<
                    ", median(didnt) = " << analysis(didnt) << endl;
}
```

这个函数也是非常小的——尽管我们的确在其中使用了两个新的概念。第一个概念告诉我们应该怎样定义一个代表了一个函数的参数。analysis 的这个参数定义看起来和我们在§4.2 中编写的那个函数声明很相似。（实际上，跟我们将会在§10.1.2 中看到的那样，我们还会对此进行更加深入的讨论。不过我们另外讨论的那个细节并不会直接影响到我们当前的讨论。）

另一个新的概念是那个返回类型 **void**。我们只能以某些受约束的方式来使用内部类型 void，例如，我们可以用它来为一个返回类型命名。当我们说一个函数“返回”空（void）值的时候，实际上我们是说它并没有返回值。我们可以执行一条并没有值的 return 语句从而退出这样的一个函数，例如：

```
return;
```

或者，正如我们在这里所做的一样，我们可以省去函数的结尾。通常我们不能直接省去一个函数的结尾，但是 C++语言允许函数返回空值从而达到这个目的：

至此，我们就能够编写其余部分的程序了：

```
int main()
{
    // 做以及没做全部家庭作业的学生
    vector<Student_info> did, didnt;

    // 读入学生记录并划分它们
    Student_info student;
    while (read(cin, student)) {
        if (did_all_hw(student))
            did.push_back(student);
        else
            didnt.push_back(student);
    }

    // 证实这些分析将向我们出示某些结果
    if (did.empty()) {
        cout << "No student did all the homework!" << endl;
        return 1;
    }
    if (didnt.empty()) {
        cout << "Every student did all the homework!" << endl;
        return 1;
    }

    // 进行分析
    write_analysis(cout, "median", median_analysis, did, didnt);
    write_analysis(cout, "average", average_analysis, did, didnt);
    write_analysis(cout, "median of homework turned in",
                   optimistic_median_analysis, did, didnt);

    return 0;
}
```

下面只要编写 average_analysis 和 optimistic_median_analysis 就可以了。

6.2.3 计算基于家庭作业平均成绩的总成绩

我们希望让 average_analysis 函数使用家庭作业的平均成绩而不是中值成绩来计算学生的总成绩。因此，对我们来说逻辑上的第一步是编写一个函数来计算一个向量的平均值，这样我们就可以用它来代替 median 从而计算出成绩：

```
double average(const vector<double>& v)
{
    return accumulate(v.begin(), v.end(), 0.0) / v.size();
}
```

这个函数使用了 accumulate 函数，跟我们已经用过的其他库算法不同，accumulate 是在<numeric>中声明的。正如这个头文件的名称所示，它为数值运算提供了工具。accumulate 函数的头两个参数指示了一个区间，而函数所做的工作是对区间中的值求和，函数的第三个参数所给定的值则是求和结果的开始。

和的类型就是第三个参数的类型，因此，正如我们在这里所做的一样，我们使用了 0.0 而不是 0——这一点非常关键，否则，求和的结果将是一个 int 类型的值，而且所有的小数部分都将会丢失。

用 accumulate 算出了区间中有元素的和之后，我们就用 v.size()来除这个总和，v.size()是这个区间中的元素个数。当然，相除后的结果就是我们要返回给调用程序的平均值。

一旦我们有了这个 average 函数，我们就能使用它来实现 average_grade 函数以反映这个计算成绩的方案了：

```
double average_analysis(const Student_info& s)
{
    return grade(s.midterm, s.final, average(s.homework));
}
```

这个函数使用了 average 函数来计算家庭作业的总体成绩，然后它把这个成绩交给§4.1 中的 grade 函数，让 grade 用它来计算总成绩。

有了这个基础结构之后，average_analysis 函数本身就是相当简单的：

```
double average_analysis(const vector<Student_info>& students)
{
    vector<double> grades;

    transform(students.begin(), students.end(),
            back_inserter(grades), average_grade);
    return median(grades);
}
```

这个函数和 median_analysis 函数之间惟一不同的是它的名称以及它用 average_grade 来代替了 grade_aux。

6.2.4 上交的家庭作业的中值

最后一种分析方案 optimistic_median_analysis 的名称来源于一个 optimistic(乐观的)假设，我们假设学生上交的家庭作业的数量对他们的家庭作业成绩并没有影响。根据这个假设，我们将计算每个学生所实际提交的家庭作业的中值成绩。我们把这个计算叫做 optimistic 中值，我们将首先编写一个函数来计算这个中值。当然，对那些有可能根本不做家庭作业的学生，我们不得不加以防范，如果是这样的话，我们将用 0 来作为家庭作业的总体成绩：

```
// s.homework 的非零元素的中值；如果没有这样的元素存在的话，则为 0
double optimistic_median(const Student_info& s)
{
    vector<double> nonzero;
    remove_copy(s.homework.begin(), s.homework.end(),
                back_inserter(nonzero), 0);

    if (nonzero.empty())
        return grade(s.midterm, s.final, 0);
    else
        return grade(s.midterm, s.final, median(nonzero));
}
```

这个函数从向量 homework 中取出非零元素，并把它们放进一个新的向量中，这个新向量叫做 nonzero。一旦我们有了这些非零的家庭作业成绩，我们就调用在 § 4.1 中定义的 grade 函数版本来计算总成绩，这个总成绩是基于被实际提交的家庭作业的中值的。

在把数值存进 nonzero 的时候，我们采用了一种新的方法，而这也是我们在这个函数中用到的惟一的新概念。为了把数值存进 nonzero，我们调用了 remove_copy 算法。为了理解对 remove_copy 的调用，我们有必要先了解一下：库会提供许多算法的“复制”版本。例如，remove_cope 做了 remove 所做的工作，但它会把结果复制到一个指定的目的地。

remove 函数查找与一个特定值匹配的所有值并把这些值从容器中“删除”掉。在输入系列中所有不被“删除”的值将被复制到目的地。在此，我们将简单讨论一下“删除”意味着什么。

remove_copy 函数有三个迭代器和一个数值。跟大多数的算法一样，remove_copy 算法假定在目的地有足够的空间来保存所有被复制的元素。在有必要的时候，我们会调用 back_inserter 来为 nonzero 增加容量。

现在我们应该能看到了，调用 remove_copy 的作用是把 s.homework 中所有非零元素都复制 nonzero 中。然后，我们查看 nonzero 是否为空，如果不是的话，我们就进行普通的 grade 运算，这个运算是基于非零家庭作业成绩的中值的。如果 nonzero 为空，那我们用 0 来作为家庭作业成绩。

当然，为了完成我们的分析，需要编写一个分析函数来调用我们的 optimistic_media 函数。我们把它作为一个练习题。

6.3 对学生进行分类并回顾一下我们的问题

在第 5 章我们看到了一个问题，它要求把成绩不及格的记录复制到一个单独的向量中并从现有的向量中删除掉这些记录。事实证明，在输入长度增加的时候，对这个问题的明显而简单的求解方法的性能十分难以断定。接下来，我们用一个链表来代替向量从而解决这个性能问题，我们希望回顾一下这个问题并出示一个算法解决方案，这个方案的性能会跟那个经修正的数据结构的性能相似。

我们能使用算法库来演示另外的两个解决方案。第一个方案的速度有点欠缺，因为它使用了一对库算法并访问了每个元素两次。不仅如此，我们还可以做得更好，为此我们使用了一个更为专用的库算法，这个算法让我们只需一次传递就能解决问题。

6.3.1 一种两次传递的解决方案

我们的第一个方法将使用一个跟我们在§6.2.4 中所使用的相类似的策略，在这两个方法中我们都只想要非零的家庭作业成绩。这样的话，我们就不想去修改 homework 本身，因此我们使用 remove_copy 来把非零的家庭作业的复件存进一个单独的向量中。就我们当前的问题来说，我们同时需要复制和删除那些非零元素：

```
vector<Student_info> extract_fails(vector<Student_inf>& students)
{
    vector<Student_info> fail;
    remove_copy_if(students.begin(), students.end(),
                   back_inserter(fail), pgrade);
    students.earse(remove_if(students.begin(), students.end(),
                             gfrade), student.end());
    return fail;
}
```

这个程序和§5.3 中的那个程序具有相同的接口，在§5.3 的程序中我们介绍了一个基于向量的解决方案，而且我们在那个方案中使用了迭代器来代替索引。跟我们在那个方案中所做的一样，在这里我们将使用传递给我们的那个向量来为及格的学生保存成绩，同时我们定义了 fail 来保存不及格的成绩。

在我们原先的程序中，我们使用了一个名为 iter 的迭代器来向前访问容器，同时把成绩不及格的记录复制进 fail 中，并使用 erase 成员函数来从 students 中清除这些记录。这一次，我们使用 remove_copy_if 函数来把不及格的成绩复制进 fail 中。这个函数的操作跟我们在§6.2.4 中所用的 remove_copy 相似，不同之处是，它使用了一个谓词而不是一个值来作为它的检测。我们给它一个谓词来把 fgrade(§5.1)的调用结果取反：

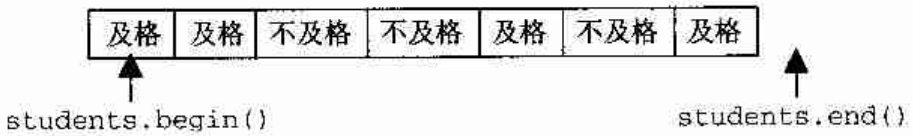
```
bool pgrade(const Student_info& s)
{
```

```
    return !fgrade(s);
}
```

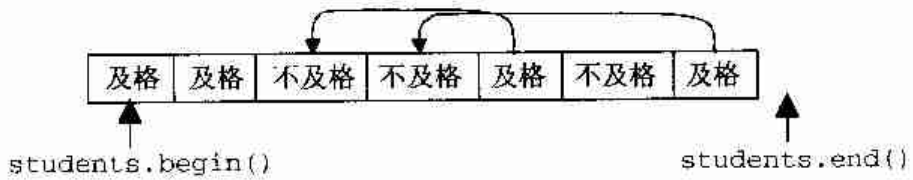
如果我们把一个谓词传递给 remove_copy_if，那就表示我们在请求它“删除”每一个满足谓词的元素。就此而言，“删除”一个元素意味着不复制它，因此我们仅仅复制那些不满足谓词的元素。这样的话，把 pgrade 传递给 remove_copy_if 意味着函数仅仅复制成绩不及格的学生记录。

接下来的一条语句有些复杂。首先，我们调用 remove_if 来“删除”与不及格成绩相对应的元素。其次，对“删除”加引号是因为实际上并没有元素被删除。相反，remove_if 复制所有不满足谓词的元素——这样的话，所有的学生记录都将具有及格的成绩。

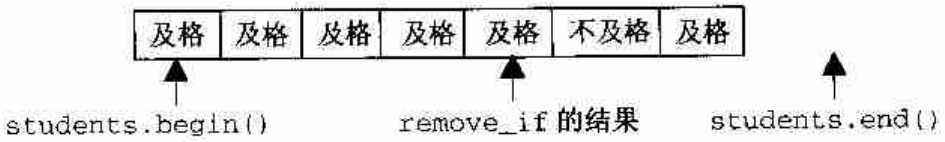
读者们可能会觉得这个调用难以理解，因为 remove_if 使用了同一个的序列来作为它的源点和目的地。而实际上它所做的是把不满足谓词的元素都复制到序列的开头。例如，假定我们首先从成绩如下的七个学生开始：



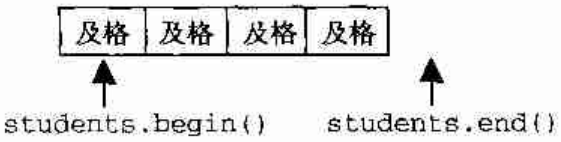
那么对 remove_if 的调用将会让开头的两条记录维持不变，这是因为它们已经处于正确的位置了。它将“删除”接下来的两条记录，对此，它把这两条记录当作空闲空间来处理，并且用后面应被保留的记录来重写这些空间。因此，当它查看到第五条代表着一个成绩及格的学生的记录的时候，它将把这条记录复制到当前的空闲位置中，而这个位置本来是被用来保存第一条被“删除”的成绩不及格的记录的。以此类推：



这样的话，结果将会是把那四条及格的记录复制到序列的开头，而让剩下来的三条记录维持不变。从而我们就能够知道序列中有多少个元素仍然是有关联的，remove_if 返回一个迭代器，这个迭代器指向位于最后一个不被删除的元素后面的那个位置：



接下来，我们要从 students 中清除那些我们不需要的记录。我们之前并没有用过这个版本的 erase 函数。它有两个迭代器参数，同时它清除了由这些迭代器所界定的区间中的所有元素。如果我们清除了在调用 remove_if 所返回的迭代器与 students.end()之间的元素，所剩下来的就仅仅是那些及格的记录了：



6.3.2 一种一次传递的解决方案

我们的第一个算法解决方案的性能是相当好的，但是我们还可以稍加改进。这是因为，§6.3.1 中的解决方案为 students 中的每一个元素都计算了两次成绩：一次在 remove_copy_if 中计算，而另一次则是在 remove_if 中。

虽然并没有库算法可以精确地按我们的期望去做，但是我们有一个从不同的角度去处理我们的问题的算法：它以一个序列作为参数并重新排列序列的元素，以使满足谓词的元素排在那些不满足谓词的元素之前。

实际上，这个算法有两个版本，这两个版本分别叫做 partition 和 stable_partition。两个版本的不同之处是，partition 可能会在每一种类内部重新排列元素，而 stable_partition 除了划分区域以外还会让各区域内的元素的相互顺序保持不变。例如，如果学生的姓名已经是按字母顺序排列好而且我们希望每一种类之内的姓名的相互顺序都不变的话，那我们就需要使用 stable_partition 而不是 partition。

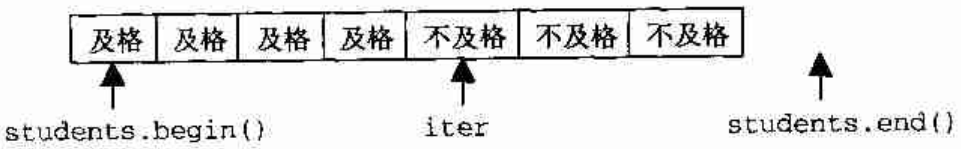
这两个算法中的每一个都返回一个迭代器，这个迭代器表示了第二个区域的第一个元素。这样的话，我们就可以取出不及格的成绩了：

```
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info>::iterator iter =
        stable_partition(students.begin(), students.end(), pgrade);
    vector<Student_info> fail(iter, students.end());
    students.erase(iter, students.end());

    return fail;
}
```

为了理解这个函数是做什么的，让我们再次从我们假设的输入数据开始吧：

在调用了 stable_partition 之后，我们就得到：



我们用不及格记录的复制来构造 fail，这些记录是来自区间[iter , students.end ())的

我们在 students 中删除了这些元素。

在我们运行这些基于算法的解决方案的时候，它们大体上会有跟基于链表的解决方案一样的整体性能。正如我们所料，一旦输入足够大，那么，跟使用了 erase 的向量解决方案相比，这个算法和基于表的解决方案会具有明显的优势。这两个算法解决方案是非常好的，如果我们对多达 75000 条记录的输入文件来计算运行时间，那么在运行时间中中占支配地位的是输入库消耗的时间。为了在 extract_fails 中比较这两个策略的效果，我们独立地分析了这一部分程序的性能。我们的计时证实了那个一次传递的算法的运行速度是两次传递的两倍。

6.4 算法、容器以及迭代器

在理解算法、迭代器、容器的用法的时候，下面的一个事实对我们的理解是很关键的：

算法作用于容器的元素——它们并不是作用于容器。

sort、remove_if 以及 partition 函数都会把基本容器中的元素移动到新的位置，但是它们并没有改变容器本身的特性。例如，remove_if 不会改变它所操作的容器的长度；它仅仅是在容器内部的不同位置复制元素。

在我们理解算法是如何跟它们用于输出的容器相互作用的时候，这个差别是特别重要的。让我们更详细地了解一下在§6.3.1 中对 remove_if 的使用。我们已经看到了，调用

```
remove_if ( students.begin() , students.end() , fgrade )
```

并不会改变 students 的长度。相反，它把每一个令谓词为 false 的元素都复制到 students 的头部并让其余的元素保持不变。如果我们需要缩短这个向量以忽略这些元素，那我们必须自己去完成这个动作。

在我们的例子中，我们用了：

```
students.erase(remove_if(students.begin(),students.end(),fgrade),
                students.end());
```

在此，erase 删除了由它的参数所指示的序列从而改变了向量。这个对 erase 的调用缩短了 students，被缩短后的 students 仅仅包含了我们想要的元素。值得注意的是，erase 必须是向量的一个成员，因为它会直接作用于容器，而不仅是作用于容器的元素。

同样地，认识到迭代器和算法以及迭代器和容器操作之间的相互作用也是很重要的。在§5.3 和§5.5.1 中我们已经看到了，诸如 erase 和 insert 这样的容器操作会使与被清除元素相对应的迭代器失效。更为重要的是，就向量和字符串而言，对于那些指示了位于被清除或被插入的元素之后的元素的迭代器，诸如 erase 和 insert 这样的操作会使所有的这些迭代器失效。因为这些操作可能会使迭代器无效，所以，在使用这些操作的时候我们必须小心地保存迭代器的值。

同样地，诸如 partition 或 remove_if 这样的函数（它们能在容器内不同位置移动元素）也将会改变特定的迭代器所指示的元素。在运行了这样的一个函数之后，我们就不能继续

迭代器来指示一个特定的元素了。

6.5 小结

类型修饰符：

static type variable;

用于局部声明，声明了具有 static 存储类型的 variable（变量）。此变量的值会在这个作用域的执行过程中维持不变，而且我们还要保证它的初始化工作在变量被第一次使用之前完成。如果程序从这个作用域退出，那么这个变量会保留它的值直到程序下一次进入这个作用域。在§13.4 中我们将看到在不同的情况中 static 所包含的具体意义。

类型：内部类型 void 的用法相当有限，其中的一种用法是：我们用它来表示一个函数并没有产生返回值。这样的函数可以通过一条“return;”语句退出，这条语句并没有值或者说它省略了函数的结尾。

迭代器适配器是产生迭代器的函数。最常见的是那些产生迭代器 insert_iterators 的适配器，这样的迭代器会让关联的容器动态地增长。这样的迭代器能被安全地用作一个复制算法的目的地。它们是在头文件<iterator>中定义的：

back_inserter(c)

对容器 c 产生一个迭代器，这个迭代器会给 c 添加元素。这个容器必须支持链表、向量以及字符串类型都会支持的 push_back 操作。

front_inserter(c)

作用与 back_inserter 一样，但它是在容器的头部插入元素。这个容器必须支持 push_front 操作——链表会支持这个操作，然而字符串和向量类型则不会。

inserter(c,it)

作用与 back_inserter 一样，但它是在迭代器 it 之前插入元素。

算法：除非是特别说明，否则以下的算法都是在<algorithm>中定义的：

accumulate(b,e,t)

把区间[b,e)中的元素的总和加上 t 之后存储在 t 中。是在<numeric>中定义的。

find(b,e,t)

find_if(b,e,p)

search(b,e,b2,e2)

在序列[b,e)中查找一个特定值的算法。算法 find 查找值 t；find_if 算法根据谓词 p 来检查每一个元素；算法 search 查找由[b2,e2)所指示的序列。

copy(b,e,d)

remove_copy(b,e,d,t)

remove_copy_if(b,e,d,p)

这些算法把[b,e)所指示的序列复制到由 d 指示的目的地中。算法 copy 复制了整个序列；remove_copy 复制了所有不等于 t 的元素；remove_copy_if 则复制了所有使谓词 p 为假的元素。

remove_if(b,e, p)

排列容器以使在区间[b,e)中使谓词 p 为假的元素位于这个域的头部。返回一个迭代器，这个迭代器指示了位于那些不被“删除”的元素之后的那个位置。

remove(b,e,t)

作用与 remove_if 一样，但是检测了哪些元素不等于值 t。

transform(b,e,d,f)

根据域[b,e)中的元素运行函数 f，把 f 的结果存储在 d 中。

partitoin(b,e,p)

stable_partition(b,e,p)

以谓词 p 为基础从而划分在域[b,e)中的元素以使那些使谓词为真(true)的元素处于容器的头部。返回一个迭代器，这个迭代器指示了第一个令谓词为假（false）的元素；或者，如果对所有的元素谓词都是 true，那就返回 e。stable_partition 会让在每一个区域内的元素的输入顺序保持不变。

习题

6-0 编译、运行并测试本章中的程序。

6-1 使用迭代器来重新实现§5.8.1 和§5.8.3 中的 frame 以及 hcat 操作。

6-2 编写一个程序来测试 find_urls 函数。

6-3 下面的这个程序片段是做什么的？

```
vector<int> u(10, 100);
vector<int> v;
copy(u.begin(), u.end(), v.begin());
```

编写一个包含这个片段的程序并编译以及运行这个程序。

6-4 改进你在上面的那个练习所编写的从 u 复制到 v 的程序。至少存在着改进这个程序的两种可能的方法。实现这两种方法并描述这两种方法之间的优缺点。

6-5 编写一个分析函数来调用 optimistic_median。

6-6 注意到在前面的练习中的函数和§6.2.2 以及 § 6.2.3 中的那两个函数完成了同样的功能。把这三个分析函数合并成一个单独的函数。

6-7 在§6.2.2 中，那个计算成绩的分析程序的一部分功能是读入学生记录并对其进行分类，这一部分程序依赖于学生是否做了（或没有做）全部的家庭作业。这个问题跟我们在 extract_fails 中所解决的那个类似。写一个函数来处理这个子问题。

6-8 编写一个函数，用这个函数来按照你自己的选择准则来对学生进行分类。使用它来代替 extract_fails 程序从而对它进行测试，并在程序中用它来分析学生的成绩。

6-9 使用一个库算法来连接一个 vector<string>对象中的所有元素。

第 7 章

使用关联容器

到目前为止我们使用过的所有容器都是顺序容器，我们会为这些容器的元素选择适当的顺序，而容器就会按照我们所选择的顺序把元素保存起来。当我们使用 push_back 或 insert 来给一个顺序容器添加元素的时候，每一个元素都将停留在我们原先为它们选定的位置上不动，直到我们使用了某个操作来重新排列这些元素为止。

如果我们只限于使用顺序容器的话，那么对于某些种类的程序，我们可能找不到一个方法来高效地编写它们。例如，如果我们有一个整数类型的容器，而且我们希望编写一个程序来判断在容器中是否有任何元素的值等于 42，那么我们有两个似是而非的策略——但是没有一个是理想的。其中的一个策略是逐个检查容器中的元素，直到我们找到了 42 或者检查完了所有的元素为止。这个方法是直截了当的，但是它可能会比较慢——尤其是在容器有很多元素的时候。我们的另一个策略是把容器排列成适当的顺序并设计一个高效的算法来查找我们想要的元素。这个方法可能会产生快速的搜索，但是，这样的算法并不是那么容易设计的。换句话说，我们必须忍受慢吞吞的程序，要不我们就得拿出自己的高级算法来。值得庆幸的是，正如我们将在本章中看到的那样，库向我们提供了第三个可供选择的方案。

7.1 支持高效查找的容器

我们能用一个关联容器来代替顺序容器从而进行数据的存储。这样的容器会自动地把它们的元素安排在一个序列中——这个序列并不是我们插入元素的那一个，相反，它依赖于元素本身的值。此外，与顺序容器相比，关联容器利用这种排序方法来让我们可以更为快速地为特定的元素定位——这并不需要我们自己来保持容器的顺序。

关联容器提供了高效的方法来让我们查找一个包含有特定值而且有可能同时包含了附加信息的元素。我们可以用容器的一部分来进行这些高效的查找，而容器的这一部分就叫做键值。例如，如果我们是在跟踪学生的信息，那么我们可以用学生的姓名作为键值，这样我们就可以根据姓名而高效地查找学生了。

在顺序容器中，我们已经看到的最接近于键值的一部分是整数索引，每一个整数索引都是与向量的每一个元素相伴的。然而，甚至这样的索引也不是真正的键值，这是因为在我们

插入或删除一个向量元素的时候，我们都会隐含地改变位于我们所触及的元素后面的所有元素的索引。

最常见的一种关联数据结构存储了“键-值”对。这种结构把每个键和一个值联系起来，并且让我们根据键值而快速地插入和检索元素。如果我们把一个特定的“键-值”对放进了这种数据结构中，那么这个键将一直与对应的值相关联，直到我们删除了这个数对为止。这样的一个数据结构被称作**关联数组**。许多语言，例如 AWK、Perl 以及 Snobol 都有内部的关联数组。在 C++中，关联数组是库的一部分。在 C++中最常用的一种关联数组是 map（映射表）。跟其他容器类似，它是在<map>头中定义的。

在许多方面，映射表的行为特性跟向量很相似。但是在它们之间有一个基本的区别，就是映射表的索引不一定是整数；它可以是字符串，或者是任何其他的类型——但要求每一个这样的类型的值都是可以比较的，这样我们才可以为这些值排序。

在关联容器和顺序容器之间的另一个重要的差别是，因为关联容器是自动排序的，所以我们的程序不可以做任何动作来改参数素的顺序。这样的话，对容器内容进行修改的算法经常会对关联容器失效。作为对这个限制的补偿，关联容器提供了许多有用的操作，而顺序容器是无法高效地实现这些操作的。

7.2 计算单词数

作为一个简单的例子，让我们考虑一下，我们如何才能够算出在我们的输入中每一个不同的单词所出现的次数呢？如果我们使用关联容器，那么对这个问题的解决方案就几乎是微不足道的：

```
int main()
{
    string s;
    map<string, int> counters; // 存储每一个单词和一个关联的计数器

    // 读输入，跟踪每一个单词出现的次数
    while (cin >> s)
        ++counters[s];

    // 输出单词以及相关的数目
    for (map<string, int>::const_iterator it = counters.begin();
         it != counters.end(); ++it) {
        cout << it->first << "\t" << it->second << endl;
    }
    return 0;
}
```

跟其他的容器一样，我们必须指定映射表所保存的对象类型。因为映射表保存了键

数对，所以我们不但需要提及值的类型，而且我们还要指定键值的类型。因此，

```
map<string, int> counters;
```

把 counters 定义为一个映射表，它保存了类型为 int 并与 string 类型的键相关联的值。我们经常把这样的一个容器说成是“一个从字符串到整数的映射表”，在使用这样的一个映射表的时候，我们可以给它一个字符串以作为键并获取关联的整数。

我们的 counters 定义可以让我们达到目的，它把我们读到的每一个单词和一个整数类型的计数器联系起来——这个计数器记录了相关的单词在输入中出现的次数。输入循环一次一个单词地把标准输入的数据读到 s 中。在这个循环中令人感兴趣的是：

```
++counters[s];
```

在这里，我们所做的工作是，以我们刚刚读到的单词作为键从而访问 counters。counters[s]的结果是一个整数，而这个整数是和存储在 s 中的字符串相关联的。然后我们使用++来对这个整数加 1，这就表明了，这个单词在输入中出现的次数增加了一次。

当我们第一次遇到一个单词时会发生什么呢？这样的话，counters 将还没有包含一个具有这个键的元素。如果我们用一个未曾出现过的键来作为映射表的索引，那么这个映射表会自动创建一个具有这个键的新元素。这个元素具有初始化的值，也就是说，对于诸如 int 这样的简单类型，这个初始化形式等价于把值设置为零。因此，当我们第一次读入一个新单词并对这个新单词执行++counters[s]的时候，我们就确保了，在我们对 counters[s]加 1 之前它的值都将为零。从而，对 counters[s]加 1 将正确地表示到目前为止我们已经看到过这个单词一次了。

一旦我们读入了整个输入，我们就要输出计数器的值和关联的单词。我们为此而使用的方法跟我们在输出一个表或一个向量的内容时用的很相似：我们在一个 for 循环中对容器进行重复访问，为此我们使用了一个 map 类定义的迭代器类型的变量。这两种方法的惟一差别是，在 for 语句的循环体中我们使用了不同的方法来输出数据：

```
cout<< it->first << "\t" << it->second <<endl ;
```

让我们回忆一下，一个关联数组存储了一个“键-值”对的集合。在我们使用[ ]来访问一个映射表元素的时候这个事实被隐瞒起来了，这是因为，我们是把键放进[ ]中从而获取关联值的。例如，counters[s]是一个整数类型的值。然而，在我们重复访问一个映射表的时候，我们必须找到一种方法来同时接触到键和关联的值。对此，映射表容器为我们提供了一个伴随的库类型，这个库类型就叫做 pair。

一个**数对**（pair）是一个简单的数据结构，它保存了两个分别叫做 first 和 second 的元素。实际上，映射表的每一个元素都是一个数对，这个数对的 first 成员包含了键，second 成员则包含了关联的值。如果我们间接引用一个映射表迭代器，那我们获得的就是和这个映射表关联的一个 pair 类型的值。

pair 类能保存不同类型的值，因此，如果我们创建一个数对，那我们就得同时指明

和 second 成员的类型。对一个键类型为 K 而值类型为 V 的映射表而言，关联的 pair 类型是 pair<const K,V>。

值得注意的是，和一个映射表关联的数对有一个常量(const)键类型。因为 pair 中的键是常量，所以我们将被禁止修改一个元素的键值。如果这个键不是常量，那我们有可能会不知不觉地改动了这个元素在映射表中的位置。从而，这个键必须是常量，这样的话，如果我们间接引用一个 map<string, int>迭代器，那我们就获得了一个 pair<const string , int>类型的对象。因此，it->first 是当前元素的键，而 it->second 则是其关联值。因为 it 是一个迭代器，所以*it 是一个左值（§4.1.3），而且 it->first 和 it->second 也是左值。不过，it->first 的类型包含了 const，这会禁止我们对它的修改。

有了这些知识，我们就能看出，这条输出语句输出了所有的键（也就是输入中每一个不同的单词），在输出中每一个键后面都跟着一个制表符和相应的次数。

7.3 产生一个交叉引用表

一旦我们知道了如何计算在输入中单词的出现次数，那么，逻辑上的下一步就是编写一个程序来产生一个交叉引用表从而指示每一个单词是在输入中哪个地方出现的。这个扩展需要对我们的基本程序作几个改动。

首先，我们需要每次读入一行而不是一个单词，这样的话，我们就能把行编号与单词联系起来了。如果我们是逐行地读输入而不是逐单词地读，那么我们就要用一种方法来把组成每一行的单词分离开。值得庆幸的是，我们已经编写了一个这样的函数，这就是在§6.1.1 中的 split 函数。我们可以使用这个函数来把每一个输入行转换成一个 vector<string>类型的向量，而且我们能从这个变量中取出每一个单词。

我们并不打算直接地使用 split，相反，我们打算把它用作这个交叉引用函数的一个参数。这样的话，我们就对外公开了一种这样的可能性：外界可以修改我们在一行中查找单词的时候所采用的方法。例如，我们能传递§6.1.3 中的 find_urls 函数，并使用这个交叉函数来查看 URL 是在输入行中的哪个位置出现的。

和之前一样，我们将使用一个映射表，这个表的键是在输入中的不同的单词。不过，这一次我们必须让每一个键与一个更为复杂的值相关联。我们将不再跟踪一个单词的出现次数，而是想知道有这个单词出现的所有行的行编号。因为任何的特定单词都可以出现在多个行中，所以我们要用一个容器把这些行编号存储起来。

如果我们获得了一个新的行编号，那么，所有我们要做的就是把这个编号添加到我们已经拥有的属于这个单词的行编号集中。为此，顺序访问这个容器的元素就已经是足够的了，因此我们能用一个向量来记录行编号。这样的话，我们将需要一个从 string 类型到 vector<int>类型的映射表。

有了这些并不显眼的预备知识之后，接下来就让我们看看代码吧：

```
// 查找指向输入中每一个单词的所有行
map<string, vector<int> > xref(istream& in,
        vector<string> find_words(const string&) = split)
{
    string line;
    int line_number = 0;
    map<string, vector<int> > ret;

    // 读下一行
    while (getline(in, line)) {
        ++line_number;

        // 把输入行分割成单词
        vector<string> words = find_words(line);

        // 记住出现在当前行的每一个单词
        for (vector<string>::const_iterator it = words.begin();
             it != words.end(); ++it)
            ret[*it].push_back(line_number);
    }
    return ret;
}
```

我们应该注意一下这个函数的返回类型和参数列表。如果读者们留意一下返回类型的声明以及那个局部变量 ret，那么你们将会看到，我们编写的是> >而不是>>。编译器需要这个空格，这是因为，如果它看到的是没有插入空格的>>，那它就会假定它是在查看一个>>运算符而不是两个独立的>符号。

在参数列表中我们应该注意到，find_words 定义了一个函数参数，这样的话，我们就可以把这个函数传递给 xref，并让 xref 用它把输入分解成单词。另一个让人感兴趣的地方是，我们在 find_words 的定义之后使用了=split，这就表明了，这个参数有一个**缺省**参数。如果我们给参数一个缺省参数，那就表示调用程序可以在有必要的时候省略这个参数。如果它们提供了一个参数，那么函数就将使用所提供的参数；如果它们省略了这个参数，那编译器将会代入缺省值。因此，用户能以下面的任一种形式来调用这个函数：

```
xref(cin);                  //在输入流中使用 split 来查找单词
xref(cin, find_urls);       //使用名为 find_urls 的函数来查找单词
```

我们在函数体中首先定义了一个叫做 line 的字符串变量，它将在我们读输入的时候保存输入的每一行；同时我们还定义了一个叫做 line_number 的 int 类型的变量来保存我们当前正在处理的行的行编号。输入循环调用了 getline(§5.7)来每次一行地把输入读到 line 中。只要还有输入，我们就对行计数器加 1 然后处理本行中的每一个单词。

我们在这个处理过程的开头首先定义了一个名叫 words 的局部变量，它将保存在 l

所有单词，而且我们调用 find_words 来对它进行初始化。进行初始化的函数也可以是我们的 split 函数（§6.1.1）——这个函数会把 line 分割成独立的单词——或者是另一个带有一个字符串参数并返回一个 vector<string>类型的结果的函数。接下来我们使用一条 for 语句来访问 words 中的每一个单词，并在每一次经过 words 的时候更新映射表。

for 语句头对我们来说应该是很熟悉的：它定义了一个迭代器并在 words 中顺序地向前推进这个迭代器。那条构成 for 循环体的语句在初次阅读时可能是难于理解的：

```
ret[*it].push_back(line_number);
```

因此，我们将把它分成几部分来讨论。迭代器 it 指示了 words 的一个元素，这样的话，*it 就是输入行中的一个单词。我们用这个单词来作为映射表的索引。表达式 ret[*it]返回存储了存储在映射表中的一个值，而这个值位于*it 所指示的位置上，它的类型是 vector<int>，它保存了到目前为止有这个单词出现的所有行的行编号。我们调用向量的 push_back 成员来把当前的行编号添加到向量中。

跟我们在§7.2 中看到的一样，如果我们是第一次看到这个单词，那么那个相关联的 vector<int>类型的向量将会被数值初始化。正如我们将会在§9.5 中看到的那样，类类型（自定义类型）的数值初始化有点复杂；我们所要了解的是，向量的数值初始化方式跟在我们不明确地指定一个初始化值的情况下 vector 类型的变量的创建方式是一样的。在这两种情况中，向量被创建的时候都不会有任何元素。因此，当我们把一个新的字符串键插入到映射表中的时候，这个键将与一个空的 vector<int>类型的向量相关联。调用 push_back 将会把当前的行编号添加到这个初始为空的向量中。

编写了 xref 函数之后，我们就能用它来产生一个交叉引用表了：

```
int main()
{
    // 缺省使用 split 来调用 xref
    map<string, vector<int> > ret = xref(cin);

    // 输出结果
    for (map<string, vector<int> >::const_iterator it = ret.begin();
         it != ret.end(); ++it) {
        // 输出单词
        cout << it->first << " occurs on line(s): ";

        // 后面跟着一个或多个的行编号
        vector<int>::const_iterator line_it = it->second.begin();
        cout << *line_it;            // 输出第一个行编号

        ++line_it;
        // 如果有的话输出其余的行编号
        while (line_it != it->second.end()) {
```

```
            cout << ", " << *line_it;
            ++line_it;
        }
        // 换一个新行以便把每一个单词与下一个分隔开来
        cout << endl;
    }
    return 0;
}
```

我们希望这个代码看起来会跟那个更新映射表的代码不同。不过，我们在这个代码中只是使用了我们已经见过的操作。

我们首先调用了 xref 来建立一个保存行编号的数据结构，而这些行是指每一个单词在其中出现的所有行。我们使用了函数参数的缺省值，因此对 xref 的这个调用将会使用 split 来把输入分割成单词。这个程序的其余部分输出了由 split 返回的数据结构的内容。

这个程序的很大的一部分是那条 for 语句，这条语句的形式和§7.2 中的那条相似。它从 ret 的第一个元素开始查看序列中的所有元素。

在阅读这条 for 语句的循环体的时候，别忘了间接引用一个映射表迭代器会产生一个 pair 类型的值。这个值的 first 元素保存了那个（常量）键，second 元素则是与这个键相关联的值。

在开始 for 循环的时候，我们首先输出了我们正在处理的单词并且输出了一个信息：

```
cout << it->first << "occur on line(s):";
```

这个单词就是在映射表中位于与迭代器 it 相关联的位置上的键。我们通过间接引用这个迭代器来访问这个键并获取数对的 first 元素。

事实证明，在这里输出这个信息是适当的，这是因为，一个元素只有在代表着一个具有一次或多次引用的单词的情况下，它才会被存进 ret 中。这样的话，我们就可以肯定至少会有一个行编号跟在这个信息后面。我们并不知道是否会有多于一个的编号，因此我们使用了不明确的复数形式。

正如 it->first 是键一样，it->second 就是那个关联的值，就此而言，这个关联值的类型是 vector<int>，它保存了当前单词的行编号。我们把 line_it 定义为一个迭代器，我们将使用这个迭代器来访问 it->second 的元素。

我们希望用逗号把那些编号分隔开，不过我们并不希望在末尾有一个多余的逗号。因此，我们必须对第一个或最后一个元素进行特殊的处理。我们的选择是特殊处理第一个元素，为此我们明确地输出了这个元素。这个做法是很安全的，因为 ret 的每一个元素都代表着一个单词并且至少会有一个对这个单词的引用。输出了一个元素之后，为了表示我们已经这样做了，所以我们要对迭代器加 1。然后 while 循环对 vector<int>类型的向量所剩下来的元素进行重复操作——如果还有剩余的元素的话。对于每一个元素它都会输出一个逗号，逗号后面紧跟着这个元素的值。

7.4 生成句子

在这一章中，我们还将讨论一个稍为复杂的例子：我们能用一个映射表来编写一个程序，这个程序描述了一种句子结构（或者说一种文法）并且能生成符合这个描述的随机的句子。例如，我们能把一个英语句子描述成一个名词跟一个动词或者是一个名词或动词跟一个对象的组合，等等……

如果我们能够处理复杂的规则的话，那我们构造的句子就将会更加有趣。例如，我们不但可以说一个句子是一个名词后面跟着一个动词，而且我们还允许有名词词组，名词词组既可以是一个单独的名词也可以是一个形容词后面跟着一个名词词组。作为一个简单的例子，我们假定有下面的输入：

种　类	规　则
<名词>	cat
<名词>	dog
<名词>	table
<名词词组>	<名词>
<名词词组>	<形容词> <名词词组>
<形容词>	large
<形容词>	brown
<形容词>	absurd
<动词>	jumps
<动词>	sits
<位置>	on the stairs
<位置>	under the sky
<位置>	where it wants
<句子>	the <名词词组> <动词> <位置>

我们的程序可能生成

```
the table jumps wherever it wants
```

这个程序总是首先找出一条规则来生成一个句子。在这个输入中只有一条这样的规则——也就是我们表格的最后一行：

```
<句子>          the  <名词词组>  <动词>  <位置>
```

这条规则表示：如果要生成一个句子，那我们应该编写单词 the、一个名词词组、一个动词以及最后的一个表示位置的状语。程序首先随机选择一条与<名词词组>相匹配的规则。很明显，程序选择的规则是：

```
<名词词组>  <名词>
```

然后它用了下面的规则类（来）解析上面的那个名词：

```
<名词>      table
```

这个程序还必须解析动词和位置状语，很明显它选用了规则

```
<动词>      jumps
```

来解析动词，并使用了

```
<位置>      wherever it wants
```

来解析位置状语。值得注意的是，最后的这个规则把一个种类映射成几个单词，在生成的句子中，这几个单词是结合在一起的。

7.4.1 表示规则

我们的表格包含了两类表项：用尖括号括住的种类和普通单词。每一个种类都有一条或多条的规则；每一个普通单词都代表了它本身的含义。如果程序看到一个有尖括号括住的字符串，我们就知道这个字符串代表着一个种类，因此我们必须为程序找到一条与这个种类相匹配的规则并展开规则的右侧部分。如果程序查看到没有尖括号修饰的单词，那么我们知道，它将会直接把这些单词放进生成的句子中。

思考一下我们的程序是如何操作的，看来这个程序要读一个描述，这个描述是关于如何创建句子的，然后程序会随机生成一个句子。因此，我们的第一个问题是：我们应该怎样存储这个描述呢？在我们生成句子的时候，我们要让每一种类和一条将会展开此种类的规则相匹配。例如，我们首先需要找到一个规则来解析如何创建“<句子>”；从这个规则出发，我们还需要找到对于<名词词组>、<动词>、<位置>等等的规则。很明显，我们应该用一个映射表来把种类映射到相应的规则中去。

但问题是我们应该使用哪种类型的映射表呢？对于种类来说这个问题是很简单的：我们可以把它们存储在字符串中，因此我们的映射表的键类型将会是 string。

映射表的值类型则较为复杂。如果我们再观察一下那个表格，我们就能看出，任何特定的规则都可以是一个字符串的集合。例如，种类“句子”与一条有四个组成部分的规则相关联：单词 the 和三个其他的字符串，而这三个字符串本身也是种类。我们已经懂得了如何去表示这种类型的值：我们能使用一个 vector<string>类型的变量来保存每一条规则。但麻烦的是，每一个种类在输入中出现的次数都有可能不止一次。例如，在那个简单的输入描述中，种类<名词>出现了三次，同时种类<形容词>以及<位置>也都出现了同样的次数。因为这些种类出现了三次，所以它们中的每一个就都会有三条匹配的规则。

处理同一个键的多个实例的最简单的方法是，把每一个规则集合都存储在它自己的向量中。因此，我们将把这个文法存储在一个从字符串到向量的映射表中，其中的向量

vector<string>类型的对象。

这个类型名是相当冗长的。如果我们为我们的中间类型引入替代名的话，那我们的程序将会清晰很多。我们说每一条规则都是一个 vector<string>类型的变量，同时每一个种类都会映射到一个保存了这些规则的向量中。我们的分析表明，实际上我们想要定义的是三个类型——一个是规则的类型，一个是规则集合的类型，还有一个则是映射表的类型。

```
typedef vector<string> Rule;
typedef vector<Rule> Rule_collection;
typedef map<string, Rule_collection> Grammar;
```

7.4.2 读入文法

解决了文法的表示问题之后，就让我们编写一个函数来读入它吧：

```
// 从一个特定的输入流读入一个文法
Grammer read_grammer(istream& in)
{
    Grammer ret;
    string line;

    // 读输入
    while (getline(in, line)) {
        // 把输入分割成单词
        vector<string> entry = split(line);

        if (!entry.empty())
            // 用种类来存储相关联的规则
            ret[entry[0]].push_back(
                Rule(entry.begin() + 1, entry.end()));
    }
    return ret;
}
```

这个函数将从输入流读数据并生成一个文法作为输出。这个 while 循环看起来跟我们之前见过的许多 while 循环形式很相似：它每次一行地从 in 读入数据并把它所读到的数据存储在 line 中。在我们读完了输入或遇到了无效数据的时候 while 语句就会终止。

while 语句的循环体是相当简洁的。我们使用了§6.1.1 中的 split 函数来把输入分割成单词并把结果向量存储在一个名为 entry 的变量中。如果 entry 为空，那就表示我们遇到的是一个空白的输入行，因此我们会略过它。否则，我们知道，entry 的第一个元素将会是那个我们正在定义的种类。

我们把这个元素当作索引从而将它用于 ret。表达式 ret[entry[0]]生成了 rule_collection 类型的对象，这个对象是和 entry[0]中的种类相关联的。记住，一个 rule_collection 类型的对象是一个向量，它的每一个元素都保存了一个类型为 rule 的对象（或者，换句话说，是一个

vector<string>的对象）。因此，ret[entry[0]]是一个向量，我们把我们刚刚读到的规则压进这个向量的尾部。这个规则是在 entry 中的，它从第二个元素开始；entry 中的第一个元素则是种类。我们通过复制 entry 中的元素（第一个元素除外）从而构造了一个新的、未命名的 rule 类型的变量，我们把这个新创建的变量推入 rule_collection 类型的由 ret[entry[0]]指示的对象的尾部。

7.4.3 生成句子

读进了所有的输入之后，接下来我们必须生成一个随机的句子。我们知道，我们的输入是一个文法并且我们希望生成一个句子。我们的输出将是一个表示输出句子的 vector<string>类型的向量。

这一部分程序是很简单的。我们更加感兴趣的问题是函数到底是如何工作的。我们知道，最初我们需要找到一条对应于<sentence>的规则。此外，我们打算分段地构造我们的输出，我们将用不同的规则和规则的某些部分来组成这个输出。

原则上，我们可以把这些段连接起来形成我们的结果。然而，由于没有内建的对 vector 的连接操作，所以我们将从一个空的向量开始并对它重复调用 push_back。

这两个限制（从< sentence >开始并对一个初始为空的向量重复调用 push_back）表明了我们希望用一个辅助函数来定义我们的句子生成程序，对辅助函数的调用如下所示：

```
vector<string> gen_sentence(const Grammar& g)
{
    vector<string> ret;
    gen_aux(g, "<sentence>", ret);
    return ret;
}
```

实际上，对 gen_aux 的调用是一个请求，它请求使用文法 g 并依照<sentence>规则生成一个句子。

剩下来我们所要做的工作是定义 gen_aux。在这样做之前，我们应该先注意一下，gen_aux 必须判断一个单词是不是代表了一个种类，为此它会检查这个单词是否括起。因此，我们需要定义一个判定来完成这个功能：

```
bool bracketed(const string& s)
{
    return s.size() > 1 && s[0] == '<' && s[s.size() - 1] == '>';
}
```

gen_aux 的工作是展开那个作为它的第二个参数而被传递过来的输入字符串，为此它要在那个作为它的第一个参数的文法中查找这个字符串并把它的输出放在第三个参数中。这里的“展开”意味着我们将使用在§7.4 中描述的那个过程。如果我们的字符串是有括号括起的，那么我们必须找到一条相应的规则，我们将会展开这条规则从而代替这个被括起的种类。如果输入字符串没有括号括起，那么输入本身就是我们的输出的一部分，而且我们不需要做这

处理就可以把这个输入推入到那个输出向量中：

```
void gen_aux(const Grammar& g, const string& word, vector<string>& ret)
{
    if (!bracketed(word)) {
        ret.push_back(word);
    } else {
        // 为对应于word的规则定位
        Grammar::const_iterator it = g.find(word);
        if (it == g.end())
            throw logic_error("empty rule");

        // 获取可能的规则集合
        const Rule_collection& c = it->second;

        // 从规则集合中随机选择一条规则
        const Rule& r = c[nrand(c.size())];

        // 递归展开所选定的规则
        for (Rule::const_iterator i = r.begin(); i != r.end(); ++i)
            gen_aux(g, *i, ret);
    }
}
```

我们的第一个工作是微不足道的：如果单词没有括号括起，那它就代表了它本身的含义，因此我们把它添加到 ret 中——至此，我们完成了任务。现在让我们转入那个令人感兴趣的部分：在 g 中查找和我们的单词相对应的规则。读者们可能会觉得，我们可以简单地引用 g[word]，但是，这样做将会导致错误的结果。回忆一下§7.2 中的内容，当我们尝试用一个不存在的键来作为一个映射表的索引的时候，这个映射表会自动创建一个带有这个键的元素。但是，在这个例子中却永远不会出现这种情况，这是因为我们不希望让伪规则搞乱我们的文法。此外，g 是一个常量映射表，这样的话，就算我们想创建新的项目也是无法做到的。实际上，在一个常量映射表中甚至连[ ]的定义都没有。

很明显，我们必须使用一个不同的工具：map 类的 find 成员，它根据特定的关键字（键值）来查找元素——如果有这样的元素存在的话；如果能找到一个元素，那它会返回一个指向这个元素的迭代器。如果在 g 中没有这样的元素存在，那 find 算法返回 g.end()。因此，如果我们比较了 it 和 g.end()，那我们就可以确保有规则存在。如果规则不存在，那就意味着这个输入是不一致的（它使用了一个没有相应规则的被括起的单词）这样的话，我们就抛出一个异常。

在这里，it 是一个迭代器，这个迭代器指向映射表 g 的一个元素，间接引用这个迭代器会产生一个数对（pair），这个数对的第二个成员是映射表元素的值。因此，it->second 指示了一个与这个种类相应的规则集合。为了方便起见，我们定义了一个名为 c 的引用来作为这

的一个替代名。

我们下一步的工作是从这个集合中随机选择一个元素，我们是在 r 的初始化语句中完成这个动作的。这行代码

```
const Rule& r = c[nrand(c.size())];
```

对我们来说并不熟悉，因此,我们有必要仔细地观察一下它。首先，让我们回忆一下，我们把 c 定义成一个 Rule_collection 类型的变量，这个变量是一种向量。我们调用了一个名为 nrand 的函数（我们将会在§7.4.4 中定义这个函数）来随机选择这个向量的一个元素。如果我们给 nrand 一个参数 n，那么它就会返回在域[0,n)中的一个随机的整数。最后，我们把 r 定义为这个元素的一个替代名。

在 gen_aux 函数中我们最后的任务是检查 r 的每一个元素。如果某一个元素有括号括起，那我们就必须把它展开为一系列的单词；否则，我们把它添加到 ret 中。在初次阅读的时候，读者可能会感到很惊讶：这个处理恰好就是我们正在对 gen_aux 所做的——因此，我们可以调用 gen_aux 来完成它！

这样的一个调用叫做**递归**，它是那些看起来不可能有效的技术中的一种——直到已经试验过几次以后，我们才会消除这种疑惑。为了使自己确信这个函数能起作用，读者们不妨先留意一下，如果 word 不带括号的话，那么这个函数显然会起作用。

接下来，假定 word 有括号括起，但是它的规则的右边并没有被括起的单词。就此而言，我们还是很容易就可以看出这个程序将起作用，这是因为，在它每次进行递归调用的时候，它所调用的 gen_aux 都将马上看出这个单词是没有括号的。因此，它将把这个单词添加给 ret 并返回。

接下来，让我们假定 word 指向一条略为复杂的规则——也就是一条在其右边使用了带括号的单词的规则，但这些单词所指向的规则都是本身不含有带括号的单词的。当我们遇到一个对 gen_aux 的递归调用的时候，我们不用试图去理解它做了什么。相反，我们要记住，我们已经可以确定在这种情况中它确实是起作用的——因为我们都知道，就算是在最坏的情况下，它的参数都会是一个这样的种类：这个种类不会导致出现任何更多的被括起的单词。最后，我们将会看到，在所有的情况中这个函数都会起作用，因为每一个递归调用都会简化其参数。

我们无法找到一个可靠的方法来解释递归。我们的经验是，人们很久之前就开始使用递归程序了，而且在这样做的时候他们并不理解它们是如何工作的。然后，有一天，他们突然就明白了其中的奥秘——不过他们还是不能了解，为什么之前他们会觉得它们是难于理解的呢？很明显，理解递归程序的关键在于先理解递归。这样的话，其他的就很容易理解了。

编写了 gen_sentence、read_grammar 和相关的辅助函数之后，我们希望使用它们：

```
int main()
{
    // 生成句子
    vector<string> sentence = gen_sentence(read_grammar(cin));
```

```
    // 输出第一个单词，如果存在的话
    vector<string>::const_iterator it = sentence.begin();
    if (!sentence.empty()) {
        cout << *it;
        ++it;
    }

    // 输出其余的单词，每一个单词之前都有一个空格
    while (it != sentence.end()) {
        cout << " " << *it;
        ++it;
    }

    cout << endl;
    return 0;
}
```

我们首先读入文法，根据文法生成一个句子；然后我们一次一个单词地输出这个句子。在这个程序中惟一一个稍为复杂的地方是，我们在句子的第二个以及随后的单词的前面放了一个空格。

7.4.4 选择一个随机元素

现在是时候让我们来编写 nrand 了。我们先留意一下，标准库包含了一个名为 rand 的函数（在<cstdlib>中定义）。这个函数不含参数，它返回在域（区间）[0,RAND_MAX)中的一个随机整数，这里的 RAND_MAX 是一个大整数，它也是在<cstdlib>中定义的。我们的任务是把包含 0 和 RAND_MAX 的域（区间）[0,RAND_MAX]缩小为包含 0 但不包含 n 的域[0,n)，对此我们首先要认识到 n<RAND_MAX。

读者们可能会觉得计算 rand()%n 就已经足够了，这里的 rand()%n 是把那个随机整数除以 n 所得到的余数。实际上，有两个原因会使得这个技术将以失败告终。

最重要的一个原因是实际意义上的：实际上，rand()返回的只是伪随机数。当商是小整数的时候，许多 C++系统环境的伪随机数生成器所产生的余数并不是绝对随机的。例如，rand()的逐次结果既可能是偶数也可能奇数——这种情况并不少见。这样的话，如果 n 等于 2，那 rand()%n 的逐次结果就将在 0 和 1 之间选择。

还有另外一个更为微妙的原因使我们避免使用 rand()%n：如果 n 的值很大，那么 RAND_MAX 就不会均匀地被 n 除尽，一些余数出现的频率将会比其他的更加大。例如，假定 RAND_MAX 是 32767（对于任何的系统环境，RAND_MAX 最小的允许值）且 n 是 20000。这样的话，rand()将会有两个不同的值能令 rand()%n 等于 10000（即 10000 和 30000），但是 rand()仅仅有一个值能让 rand()%n 等于 15000（也就是 15000）。因此，nrand 的简单实现产生 10000（作为 nrand(20000)的一个值）的概率将会是它产生 15000 的概率的两倍。

为了避免这些缺陷，我们将使用一种不同的策略，我们把这些可利用的随机数分成长度精确相等的存储桶。我们能计算一个随机数并返回相应的存储桶的编号。因为存储桶的长度相同，所以某些随机数根本没可能进入到任何的存储桶中。这样的话，我们会反复请求随机数，直到我们获得了一个合适的为止。

这个函数编写起来比描述的更为简单：

```
// 返回[0,n)中的一个随机整数
int nrand(int n)
{
    if (n <= 0 || n > RAND_MAX)
        throw domain_error("Argument to nrand is out of range");

    const int bucket_size = RAND_MAX / n;
    int r;

    do r = rand() / bucket_size;
    while (r >= n);

    return r;
}
```

bucket_size 的定义依赖于整数除法会对其结果进行截尾这个事实。这个性质表明了，在那些小于等于精确的商的整数中，RAND_MAX/n 是最大的一个。结果，在具有性质 n*bucket_size <=RAND_MAX 的所有整数中，bucket_size 是最大的。

接下来的语句是一条 do while 语句。do wile 语句与 while 语句类似，只是它总会执行循环体至少一次，并且它是在语句的最后检验条件的。如果条件产生真值，那循环就会重复执行 push_back 直到 while 语句的条件为假。在这个例子中，循环体把 r 设为一个存储桶的编号。存储桶 0 将对应于区间[0,bucket_size)中的值；存储桶 1 将与区间[bucket_size,bucket_size*2)的值对应；等等……如果 rand()的值足够大从而使 r>=n 成立，那么程序将继续试验随机数直到找到一个合适的为止，接着它就会返回相应的 r 值。

例如，让我们假定 RAND_MAX 等于 32767 而 n 等于 20000。那么 bucket_size 将等于 1，并且 nrand 运行时将会丢弃随机数，直到找到一个小于 20000 的为止。另一个例子，假定 n 等于 3。那么 bucket_size 将会是 10922。这样的话，在区间[0,10922)中 rand()的值将产生 0，区间[10922，21844)中的值将产生 1，区间[21844,32766)中的值将产生 2，而 32766 或 32767 的值将会被丢弃。

7.5 关于性能的一点说明

如果你已经使用过其他语言的关联数组，那么你就会知道，这些数组很可能是根据

叫**散列表**的数据结构而实现的。散列表可以很快速，不过它们同时还有着随之而来的缺点：

- 对每一种键类型，用户都必须提供一个散列函数，这个散列函数会从键的值计算出一个适当的整数。
- 一个散列表的性能对散列函数的细节极度敏感。
- 通常找不到一个简单的方法来按一个有用的顺序检索散列表的元素。

按照散列函数来实现 C++关联容器是很困难的：

- 键类型仅仅需要<运算符或者等价的比较函数。
- 访问一个关联容器中有特定键的元素所耗费的时间是容器中元素总数目的对数，而不管键的值是什么。
- 关联容器的元素总是根据键来排序的。

换句话说，尽管 C++关联容器比最好的散列表数据结构稍慢，但是它们的性能却比低级数据结构好得多。它们的性能并不依赖于用户设计出良好的散列函数，而且由于它们的自动排序，它们比散列表更加方便。一般来说，如果你对关联数据结构熟悉的话，那么你很可能会希望了解这一点：C++库很典型地使用了一个平衡自调节树（自调节平衡树）结构来实现关联容器。

如果你真的想用散列表，那么它们作为许多 C++系统环境的一部分也是可利用的。然而，由于它们不是标准 C++的一部分，所以它们超出了本书的讨论范围。虽然没有一个对所有用途都是理想的标准，但是标准关联容器对大多数的应用来说还是绰绰有余的。

7.6 小结

do while 语句与 while 语句(§2.3.1)相似，只是它对条件的检测是在语句的结尾。这种语句的一般形式是

```
do statement
while (condition);
```

do 后面的 statement（语句）是首先被执行的，之后才交替执行 condition 和 statement，直到条件为假(false)。

数值初始化：对映射表中一个还不存在的元素的访问会创建一个元素，此元素的值是 v()，在这里，v 是存储在映射表中的值的类型。这样的一个表达式被称为数值初始化。我们在§9.5 中解释了数值初始化的细节：最重要的一方面则是内部类型会被初始化为 0。

rand() 是一个函数，它产生一个在区间[0,RAND_MAX]中的随机整数。rand 和 RAND_MAX 都是在<cstdlib>中定义的。

pair<k,v>是一种简单的类型，它的对象保存了一对数值。对这些数值的访问要通过它们的名称，这两个名称分别是 first 和 second。

map<k,v>是一个关联数组，它的键类型是 k，而值类型是 v。映射表(map)的元素

值”数对，这些数对是按键顺序排列的，这样我们就可以通过键从而高效地访问元素。映射表的迭代器是双向的(§8.2.5)。间接引用一个映射表迭代器会产生一个类型为 pair<const k,v>的值。映射表操作包括：

map<K, V> m;	创建了一个新的空映射表，它的键类型是 const K，值类型是 V。
map<K, V> m(cmp);	创建了一个新的空映射表，它的键类型是 const K，值类型是 V。使用判定 cmp 来确定元素的顺序。
m[k]	使用 K 类型的键 k 来索引访问映射表并返回一个类型为 V 的左值。如果对于这个特定的键没有一个合适的项目，那就根据这个键创建一个新的被数值初始化的元素，并把这个元素插入映射表中。因为使用[ ]来访问一个映射表可能会创建一个新元素，所以不允许对常量(const)映射表使用[ ]。
m.begin() m.end()	返回迭代器，这些迭代器能被用来访问一个映射表的元素。值得注意的是，间接引用一个这样的迭代器会产生一个“键-值”对而不仅是一个值。
m.find(k)	返回一个迭代器，这个迭代器指向键为 k 的元素；或者，如果没有这样的元素存在的话那就返回 m.end()。

对于一个 map<K , V>和一个相联迭代器 p，下面的操作是可用的：

p->first	产生一个类型为 const K 的左值，它是 p 所指示的元素的键。
p->second	产生一个类型为 V 的左值，它是 p 所指示的元素的数值部分。

习题

7-0 编译、运行并测试本章中的程序。

7-1 扩充§7.2 中的程序，按出现次数来构造它的输出。也就是说，程序在输出的时候应该对所有的单词分组，首先是输出出现了一次的，然后是出现了两次的，以此类推。

7-2 扩充§4.2.3 中的程序，按分数范围评出等级：

A 90—100

B 80—89.99…

C 70—79.99…

D 60—69.99…

F <60

在输出中列出每一种类有多少个学生。

7-3 在§7.3 中的交叉引用程序是可以改进的：正如它所示，如果一个单词在同一输入行中多次出现的话，那么程序将会多次报告这一行。修改代码以使它能够检测到同一个行编号的多次重复出现并且仅仅插入这个行编号一次。

7-4 如果输入文件太大的话，那么交叉引用程序所产生的输出的格式将会不够美观。重新编写这个程序以使它在行太长的时候换行输出。

7-5 用表（链表）来作为数据结构而重新实现那个文法程序，让这个程序构造一个句子。

7-6 使用两个向量来重新实现 gen_sentence 程序：一个向量保存程序生成的那个完全展开的句子；另一个则被用作堆栈，程序将用它来保存规则。在程序中不允许使用任何递归调用。

7-7 修改那个交叉引用程序的驱动程序，使它在只有一行时输出 line，否则就输出 lines。

7-8 修改那个交叉引用程序以查找一个文件中的所有 URL，并且输出每一个不同的 URL 在其中出现的所有行。

7-9 （较难）在§7.4.4 中，nrand 的实现在参数大于 RAND_MAX 的时候将会失效。通常，这个限制是不会有问题的，因为在一般情况下，无论如何 RAND_MAX 都能等于可能存在的最大的整数。然而，在某些系统环境中 RAND_MAX 会比可能存在的最大的整数小得多。例如，RAND_MAX 等于 32767（$2^{15}-1$）而可能存在的最大的整数等于 2147483647（$2^{31}-1$）的例子并不少见。重新实现 nrand，使它对所有的 n 都能表现良好。

第 8 章

编写泛型函数

本书的第一部分把注意力集中于编写程序来解决具体的问题，这些程序使用了 C++语言的基本要素以及标准库提供的抽象。从本章开始，我们将把我们的注意力转入到另一个方面的学习中去，我们将学习如何编写我们自己的抽象。

这些抽象具有几种形式。在这一章我们先讨论泛型函数，泛型函数的参数类型是我们事先不知道的——直到我们调用了这个函数我们才会得知。从第 9 章一直到第 12 章我们演示了如何实现抽象的数据类型。最后，从第 13 章开始，我们将学习面向对象程序设计方法（OOP）。

8.1 泛型函数是什么？

到目前为止，无论我们是在什么时候编写一个函数，我们都是知道这个函数的参数和返回值的类型的。乍看上去，这个知识好像是任何函数描述的不可或缺的一部分。但是，认真观察一下我们就会知道，我们已经用过了（不过没有编写过）那些在我们使用之前并不知道其参数以及返回类型的函数。

例如，在§6.1.3 中，我们使用了一个名为 find 的函数，它的参数是两个迭代器和一个数值。我们可以使用这个 find 函数在任何种类的容器中查找任何类型适当的值。这个用法表明了，在我们使用 find 之前我们并不知道它的参数或返回类型是什么。这样的一个函数被称为**泛型函数**，使用和创建泛型函数的能力是 C++语言的一个重要特征。

语言支持泛型函数是不难理解的。难于理解的是，当我们说 find 能接受“任何适当类型”的参数的时候，我们表示的到底是什么意思。例如，如果我们要让希望使用 find 函数的人知道它对特定类型的参数是否有效，那我们应该怎样描述这个函数的行为呢？这个问题的一部分答案位于 C++语言内部，而另一部分则位于语言之外。

位于语言内部的那一部分答案是一个概念：函数对一个未知类型的参数的使用方式约束了这个参数的类型。例如，如果一个函数有两个参数 x 和 y，并且函数计算了 x + y，那么，为了使运算结果存在，x 和 y 就必须具有这样的类型；对于这些类型，x + y 是可定义的。无论何时，只要你调用了一个这样的函数，系统环境就会检查你的参数类型是否符合约束条件，而这些约束条件是由这个函数对其参数的使用所暗示的。

位于C++语言之外的那部分答案就是标准库对其函数参数的约束条件所采用的组织方式。我们已经向读者们出示了这种组织方式的一个例子——也就是迭代器的概念。有些类型是迭代器；有些类型则不是。find 函数有三个参数，这三个参数的头两个必须是迭代器。

当我们说一个特定的类型是一个迭代器的时候，实际上，我们是说到了关于这种类型所支持操作的性质：当且仅当一种类型以某种特定的方式支持一个特定的操作集时，这个类型才会是一个迭代器。如果我们打算编写自己的 find 函数，那么从某种程度上说，我们将仅仅依赖于所有的迭代器都必须支持的操作。如果我们想要编写自己的容器——就像我们将在第 11 章中所做的那样——那我们将必须提供支持了所有适当的操作的迭代器。

迭代器的概念不是 C++语言特有的一部分。然而，它是标准库组织的一个基本的组成部分，而且就是这一部分使得泛型函数跟这些概念一样有用。在这一章中我们将出示一些例子来让读者们看看库是如何实现泛型函数的。沿着这个思路，我们解释了一个迭代器到底是什么——或者，更准确地讲，迭代器是什么，这样说是因为迭代器可以分为五个种类。

这一章将会比我们以前的所有章节都更为抽象，这是因为，本章中的泛型函数恰好具有抽象的性质。如果我们所编写的函数是用来解决特定问题的，那么这些函数将不会是泛型函数。不过，读者们很快就会发现，我们所描述的大多数函数都是很熟悉的，因为我们已经在前面的例子中使用过它们了。此外，也不难想像如何使用那些即使并不熟悉的函数。

8.1.1 未知类型的中值

实现了泛型函数的语言特征被称作**模板函数**。模板可以让我们为一个行为特性相似的函数族（或者类型族）编写一个单独的定义，我们把族中各个函数（或类型）之间的差别归因于它们的**模板参数**的类型不同。在本章中我们将讨论模板函数，在第 11 章中我们将探讨模板类。

位于模板之后的关键概念是，不同类型的对象仍然可以享有共同的行为特性。模板参数让我们可以按照共同的行为特性来编写程序——即使在定义模板的时候我们并不知道与模板参数相对应的特定的类型。在我们使用一个模板的时候我们确实是知道这些类型的，而在我们编译以及连接程序的时候，这个认识是有用的。对于泛型参数，系统环境无需考虑对那些在运行期间类型会起变化的对象做什么样的处理动作——它只需要在编译期间考虑这个问题。

虽然模板是标准库的一个基础，不过我们还是可以把它们用于我们自己的程序。例如，在§4.1.1 中，我们编写了一个函数来计算一个 vector<double>类型的向量的中值。这个函数依赖于对向量进行排序的能力，然后它就可以用一个指定的索引来获取特定的元素。这样的话，如果我们想用这个函数来处理任意序列的数值，可能要花很大的工夫才行。不过，即使是这样，我们也没有理由把这个函数限制为 vector<double>类型；我们也可以取其他类型的向量的中值。模板函数为我们做到了这一点：

```
template <class T>
T median(vector<T> v)
{
```

```
    typedef typename vector<T>::size_type vec_sz;

    vec_sz size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");

    sort(v.begin(), v.end());

    vec_sz mid = size/2;

    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2: v[mid];
}
```

在这段代码中，我们首先就注意到了那个奇怪的模板头：

```
template<class T>
```

和在参数列表和返回类型中的 T 的使用。这个模板头告知系统环境，我们定义的是一个模板函数，而且这个函数将会有一个**类型参数**。类型参数的操作方式和函数参数很相似：它们定义了可以在函数作用域内使用的名称。不过，类型参数是表示类型而不是表示变量的。这样的话，不管 T 出现在函数的哪一个地方，系统环境都会认为 T 命名了一个类型。在这个 median 函数中，我们明确地使用这个类型参数来表示那个名为 v 的向量所保存的对象的类型，同时我们还用 T 来指定了函数的返回类型。

当我们调用这个 median 函数的时候，系统环境会在编译期间把它判定的一个类型赋给 T。例如，我们可以调用 median(vi)从而给 median 函数一个 vector<int>类型的名为 vi 的对象。根据这个调用，系统环境就能推断出 T 是 int。这样的话，无论我们是在这个函数的哪一个地方使用 T，系统环境都会产生效果跟 int 一样的代码。实际上，系统环境会对我们的代码进行实例化，就好像我们编写了一个参数类型为 vector<int>、返回类型为 int 的特殊版本的 median 函数一样。我们即将会对实例化进行更多的讨论。

在 vec_sz 的定义中，**typename** 的使用是另一个我们不熟悉的概念。它会告知系统环境，vector<T>::size_type 是一个类型名——即使系统环境还不知道类型 T 具体是代表什么的。无论何时，只要我们有一个诸如 vector<T>这样的依赖于一个模板参数的类型，并且我们希望使用这个类型的一个诸如 size_type，这样的成员——这个成员本身也是一个类型，我们都必须在整个名称的前面加上 typename，从而让系统环境知道应该把这个名称当作一个类型来处理。虽然标准库会确保，对于任何的 T，vector<T>::size_type 都是一个类型的名称，但是对标准库类型，系统环境却没有特殊的了解，所以它没有办法来获知这个事实。

在阅读一个模板的时候，我们通常都会看到，类型参数会遍及它的整个定义——即使许多类型依赖都是不明确的。在我们的 median 函数中，我们只是在函数的返回类型和参数列表以及在 vec_sz 的定义中明确地使用了类型参数。然而，因为 v 的类型是 vector<T>，所以任何包含 v 的操作都隐含地涉及了这个类型。例如，在表达式

```
(v[mid] + v[mid - 1]) / 2
```

中，只有在知道了 v 的元素类型的情况下，我们才会知道 v[mid]和 v[mid - 1]的类型。这些类型也同时决定了+和/运算符的类型。如果我们使用 vector<int>类型来调用 median，那么我们就会看到，+和/的操作数类型都是 int，而且这些运算会返回 int 类型的结果。如果我们对一个 vector<double>类型的参数调用了 median，那么 median 就会对 double 类型的数值进行运算。我们不能对 vector<string>类型的参数调用 median，这是因为，median 使用了除法，而 string 类型没有除法运算符。这个行为正是我们所希望的，毕竟，有谁知道查找一个 vector<string>类型的对象的中值到底有何意义呢？

8.1.2 模板实例化

如果我们对 vector<int>类型调用 median，系统环境就将高效地创建并编译这个函数的一个实例，它会用 int 来代替所有对 T 的使用。如果我们对类型 vector<double>调用 median 的话，那么系统环境将会再次根据调用推断出正确的类型。这样的话，T 被赋予的值将会是 double，同时系统环境将会产生 median 的另一个版本，这个版本使用了 double 来代替 T。

C++标准并没有说明系统环境是如何处理模板的实例化的，因此，所有的系统环境都会以它自己的特定的方式去处理实例化。当我们不能确切地说明编译器将如何处理实例化的时候，我们应该记住如下的两点：第一，对于那些沿用了传统的编辑-编译-连接模式的系统环境来说，实例化动作经常不是在编译期间而是在连接期间发生的。只有模板被实例化了，系统环境才能证实，模板代码能被用于指定的类型。因此，在连接期间，我们就可以发现那些看起来在编译期间可能发生的错误。

只有在我们编写自己的模板的时候，才用考虑第二点：为了对一个模板进行实例化，当前的大多数系统环境都要求这个模板的定义（而不仅仅是声明）必须是系统环境可以访问的。一般来说，这个要求意味着，定义了模板的源文件和头文件都必须是可访问的。源文件的定位方式是由具体的系统环境决定的，不同的系统环境会有不同的处理办法。许多系统环境都要求模板的头文件直接地或者是通过#include 指令而把源文件包含进去。如果你们想了解你们的系统环境的要求，那最可靠的办法还是查阅它的文献。

8.1.3 泛型函数和类型

在§8.1 中我们提到，在设计以及使用模板的时候最困难的部分是精确地理解在一个模板和能被用于此模板的“适当的类型”之间的交互作用。在我们对 median 的模板版本的定义中，我们看到了一个很明显的类型依赖：存储在向量中的类型会被传递给 median，这些类型必须支持加法和除法操作，而且这些操作最好是表示标准的算术意义。值得庆幸的是，大多数定义了除法的类型都是算术类型，这样的话，在实践中这些依赖是不会产生问题的。

模板和类型转换之间的相互作用还会导致更为细微的问题。例如，当我们调用 fin[illegible]查学生是否做了他们的全部作业的时候，我们编写了下面的语句：

```
find(s.homework.begin() , s.homework.end() , 0);
```

这里的 homework 是一个 vector<double>类型的向量，但是我们却要用 find 来查找一个 int 类型的数值。这个特殊的类型失配是不会有问题的：我们能把一个 int 类型的值与一个 double 类型的值相比较，这样并不会造成意义的丢失。

然而，当我们以下面的形式来调用 accumulate 的时候，

```
accumulate(v.begin() , v.end() , 0.0)
```

我们注意到，我们的程序的正确性依赖于我们对 double 形式而不是 int 形式的零值的使用。原因在于，accumulate 函数使用了它的第三个参数的类型来作为它的累加器的类型。如果这个类型是 int（即使我们是在对一系列的 double 类型的数值进行累加）那这个加法就会被截尾，即它的结果只能有整数部分。在本例中，系统环境将会让我们传递一个 int 类型的参数，但是我们获得的总和将会缺乏精度。

最后，当我们调用 max 时，

```
string::size_type  maxlen - 0;
maxlen = max(maxlen , name.size());
```

我们注意到，maxlen 的类型和 name.size()的返回类型必须精确地匹配。如果这两个类型不匹配，那么这个调用将不能通过编译。既然我们知道了，模板的参数类型是根据参数类型而被推断出来的，那我们就能理解为什么会有这些情况存在。思考一下 max 函数的下面这个似是而非的实现：

```
template <class T>
T max(const T& left, const T& right)
{
    return left > right? left : right;
}
```

如果我们传递两个分别为 int 类型和 double 类型的参数给这个函数，那么系统环境就无法推断出应该把哪一个参数转换成另一个参数的类型。它是应该把这个调用解析成比较 int 类型的值，然后返回一个 int 类型的值，还是应该把两个参数都当作 double 类型的值来处理，然后返回一个 double 值呢？系统环境无法作出决定，因此在编译期间这个调用将不能通过。

8.2 数据结构独立性

我们刚刚实现的 median 函数使用了模板，它能适用于所有可以被向量容纳的类型上。我们可以调用这个函数来查找一个保存有任意算术类型值的向量的中值。

更为常见的是，我们希望能够编写出一个简单的函数，这个函数可以处理被存储在任何种类的数据结构中的数值——例如链表、向量或者是字符串。就此而言，我们希望它能够

容器的一部分而不是被迫使用整个容器。

例如，标准库使用迭代器来让我们对任何容器的任何连续的部分调用 find。如果 c 是一个容器并且 val 是一个值——这个值的类型跟存在在容器中的类型相同，那么，为了使用 find，我们可以编写以下的表达式：

```
find(c.begin() , c.end() , val)
```

我们为什么要两次提到 c 呢？为什么库不让我们使用类似于 c.size()的

```
c.fnd(val)
```

或者，干脆让我们使用

```
find(c , val)
```

来把这个容器当作一个参数而直接地传递给 find 呢？事实证明，对这两个问题我们有同样的答案：如果库使用了迭代器从而要求我们两次提到 c，那么我们就有可能只需编写一个 find 函数就可以在任何容器的任何连续的一部分中查找到一个值。而其他的两个方法都不能让我们做到这一点。

让我们先来看一看 c.find(val)。如果库让我们使用 c.find(val)，那么，无论 c 具有什么样的类型，我们都将会把 find 当作 c 的一个成员来调用——也就是说，所有定义了 c 的类型的人都必须把 find 定义为一个成员。同样地，如果库对算法使用了 c.find(val)形式，那么我们就不能对内部数组使用这些函数。我们将会在第 10 章学习这一点。

为什么库要求我们把 c.begin()和 c.end()用作 find 的参数而不让我们直接传递 c 呢？之所以要传递两个值是因为，这样做会界定一个区间，这样的话，我们就可以只搜索容器的一部分而不需要彻底地搜索整个容器了。例如，读者们可以思考一下，在§6.1.1 中，如果我们给 find_if 一个限制从而让它只能搜索整个容器的话，那我们应该如何编写那个 split 函数呢？

有一个更加细微的原因可以说明泛型函数为什么要用迭代器参数而是不直接地使用容器参数：如果我们使用迭代器的话，那我们就可以访问到具有特殊意义的、而不存在于容器中的元素。例如，在§6.1.2 中我们使用了 rbegin 函数，它会产生一个迭代器，而这个迭代器能按逆向顺序来访问它的容器的元素。如果我们把这样的一个迭代器作为参数传递给 find 或 find_if，那我们就能按反向顺序来搜索容器的元素——假如这些函数直接用一个容器来作为参数的话，那我们是不可能做到这一点的。

当然，重载库函数是可能的，这样的话，在我们调用它们的时候，我们就可以用一个容器或者用一对迭代器来作为一个参数了。然而，我们还完全不清楚，为了这一点额外的方便我们是否值得给库增加不必要的复杂度。

8.2.1 算法与迭代器

如果我们想理解模板是如何让我们编写数据结构独立的程序的，那么，最简单的方[illegible]解一下某些更为常用的标准库函数的实现。这些函数都在它们的参数中包含有迭代器，

代器则标识了容器的元素，函数将会作用于这些元素。所有标准库容器以及其他的一些类型（例如 string）都提供了迭代器以允许这些函数作用于这些容器中的元素。

有一些迭代器会支持其他迭代器所不支持的操作。这个支持（或者，没有这个支持）会转化成某些迭代器支持而其他的迭代器不支持的操作。例如，在向量中用一个特定的索引来直接访问一个元素是可能的，不过在链表中则不行。因此，如果我们有一个迭代器是指向向量的一个元素的，那么，该迭代器的设计就允许我们可以获得一个指向同一向量中另一个元素的迭代器——只要我们把这两个元素的索引的距离差加到原来的迭代器中就行了。指向链表元素的迭代器并没有提供类似的工具。

因为不同种类的迭代器提供不同种类的操作，所以，重要的是要理解不同的算法对它们所使用的迭代器的具体需求以及不同种类的迭代器所分别支持的操作。无论何时，只要两个迭代器支持同样的操作，那么它们就会给这个操作同样的名称。例如，所有的迭代器都使用++来让一个迭代器指向它的容器的下一个元素。

并不是所有的算法都需要全部的迭代器操作。某些算法（例如 find）只是使用了很少的一部分迭代器操作。我们能使用我们所见过的任何的容器迭代器来查找数值。其他的算法（例如 sort）则对迭代器使用了大部分的功能强大的操作，其中包括了算术运算。在我们所见过的库类型中，只有向量和字符串类型是和 sort 兼容的。（如果我们为一个字符串排序，那么这个字符串的各个字符就会按非递减次序排列起来。）

库定义了五种**迭代器（iterator categories）**，其中的每一种都对应于一个特定的迭代器操作集合。这些迭代器种类划分了每一个库容器提供的迭代器的类别。每一个标准库算法都表示了它所期望的是哪些种类的迭代器参数。因此，迭代器种类给我们提供了一种方法来理解哪些容器可以使用哪一些的算法。

每一个迭代器种类都对应于一个访问容器元素的策略。因为迭代器种类是与访问策略相对应的，所以它们也对应于特定种类的算法。例如，有些算法只需要对它们的输入进行一次传递，因此它们并不需要能够进行多次传递的迭代器。而有些算法则要求在仅仅是给定了元素索引的情况下就能够对任意的元素进行有效的访问，因此，它们也要求能够把索引和整数相加。

现在，我们将分别描述每一个访问策略，对每一个策略，我们都会出示一个使用了它的算法并描述相应的迭代器种类。

8.2.2 顺序只读访问

访问序列的一个简单快捷的方法是顺序地读它的元素。find 就是一个采用了这种处理方法的库函数，我们用以下的代码来实现了这个函数：

```
template <class In, class X> In find(In begin, In end, const X& x)
{
    while (begin != end && *begin != x)
        ++begin;
    return begin;
```

```
}
```

如果我们调用 find(begin , end , x)，那结果要么是在区间[begin,end)中的第一个满足*iter == x 的迭代器 iter，要么就是 end——如果没有这样的迭代器存在的话。

我们知道，这个函数顺序地访问了在区间[begin,end)中的元素，这是因为，它用来修改 begin 值的惟一的操作是++。除了使用++来修改 begin 的值之外，它还分别使用了!=和*，!=是用来比较 begin 和 end，而*则是用来访问 begin 所指向的容器元素的。如果我们希望顺序地读一个由一对迭代器所指示的区间所容纳的元素，那么使用这些操作就已经是足够的了。

虽然对我们来说这些操作已经足够，但是，除了这些操作之外，我们可能还希望使用其他的操作。例如，我们可能会以如下的形式实现 find：

```
template <class In, class X> In find(In begin, In end, const X& x)
{
    if (begin == end || *begin != x)
        return begin;
    begin++;
    return find(begin, end, x);
}
```

大多数的 C++程序员都会发现这种递归程序设计风格是比较少见的，不过熟悉诸如 Lisp 或 ML 这样的语言的程序员却会觉得很有亲切感。这个版本的 find 使用了 begin++来代替++begin 并且使用了==来代替!=。根据这两个例子，我们可以得出结论，对一个序列的元素提供了顺序只读访问操作的迭代器应该支持++（前缀或后缀形式的）、==、!=以及一元*运算符。

这样的一个迭代器还应该支持另外的一个运算符。在 iter->member 和(*iter).member 之间的运算符是等价的。在§7.2 中我们已经使用过这样的例子了，在那里我们把 it->first 用作(*it).first 的一个简写——我们当然很乐意使用这样的简写。

如果一个类型提供了所有的这些操作，我们就把它称作**输入迭代器**。我们见过的每一个容器迭代器都支持所有的这些操作，因此它们全部都是输入迭代器。当然，它们也支持其他的操作，但这个额外的支持并不影响它们是输入迭代器这个事实。

当我们说 find 要求用输入迭代器来作为它的第一和第二个参数的时候，我们实际上是说，我们可以把符合输入迭代器要求的任何类型的参数传递给 find——在这里我们所说的输入迭代器当然也包括支持额外操作的那些迭代器。

8.2.3 顺序只写访问

输入迭代器只能被用来读一个序列的元素。很明显，在一些情况下我们希望能够使用迭代器来写一个序列的元素。例如，考虑一下这个 copy 函数：

```
template <class In, class Out>
Out copy(In begin, In end, Out, dest)
{
```

```
    while (begin != out)
        *dest++ = *begin++;
    return dest;
}
```

这个函数带有三个迭代器：开头的两个指示了一个序列，函数会从这个序列进行复制，第三个则指示了目的序列的开头。我们在这里所看到的循环跟§6.1 中的那个是一样的：函数在容器中向前推进 begin 并在此过程中把每一个元素复制到 dest，这个过程会一直持续到它到达了 end 为止。

正如名称 In 的字面意思所示，begin 和 end 是输入迭代器。我们只用它们来读元素——这跟我们在 find 中所做的是一样的。不过，参数 dest 的类型 Out 又是怎样的呢？让我们看一下对 dest 所进行的操作。我们可以看到，在这个函数中，我们只需要计算*dest = value 以及 dest++。跟 find 一样，逻辑完备性表明了我们也能够计算++dest。

另外，还有一个不太明显的要求。假定 it 是一个我们只用来输出的迭代器，而且我们执行了：

```
*it = x;
++it;
++it;
*it = y;
```

我们在对*it 的两次赋值运算之间对 it 进行了两次加 1 操作，执行了这些语句之后，我们已经在我们的输出序列中留了一个间隔。因此，如果我们希望对输出专门使用一个迭代器的话，那么就会有一个隐含的要求：我们不能在对*it 的两个赋值运算之间执行超过一次的++it 操作；同时，我们也不能在没有对 it 进行递增的情况下对 it 进行多次赋值。

如果一个函数以某种方式使用了一种符合这些要求的类型，那我们就把这个类型称为**输出迭代器**。所有的标准容器都提供了符合这些要求的迭代器——正如 back_inserter 所做的那样。值得注意的是，这个“一次写入”的性质是对使用迭代器的程序，而不是对迭代器本身的一个要求。也就是说，只满足输出迭代器要求的迭代器只能支持那些具备了这个性质的程序。back_inserter 所产生的迭代器是一个输出迭代器，因此，所有使用它的程序都必须符合这个“一次写入”的要求。所有的容器迭代器都提供了额外的操作，这样的话，使用它们的程序就不用受这个约束了。

8.2.4 顺序读-写访问

假定我们希望能够读和写一个序列的元素，但只是顺序地读以及写——也就是说，一旦我们处理了一个元素，我们就绝不会再次访问它[1]。作为一个库函数的例子，replace 就是这样做

[1] 实际上，forward iterator 可以对同一个元素进行多次访问，它只是要求我们不能将 iterator 移动到前置而已。

的。replace 是定义在<algorithm>头文件中的：

```
template <class For, class X>
void replace(For beg, For end, const X& x, const X& y)
{
    while (beg != end) {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

这个函数检查了在区间[beg,end)中的元素并用 y 来替换所有等于 x 的元素。很明显，类型 For 必须支持一个输入迭代器所支持的所有操作，同时它还要支持一个输出迭代器所支持的所有操作。此外，它不必满足输出迭代器的一次赋值要求，这是因为，现在它的作用是在对一个元素赋值之后再读这个元素的值，同时它也有可能会改参数素的值。这样的一个类型是一个**正向迭代器**，它必须支持：

```
*it（对于读和写）
++it 和 it++（但不用支持--it 或 it--）
it == j 和 it != j（在这里，j 的类型和 it 一样）
it->member（作为(*it).member 的一个替代名）
```

所有的标准库容器都满足正向迭代器的要求。

8.2.5 可逆访问

有些函数需要按逆向顺序访问一个容器的元素。作为一个最简单的例子，reverse 函数就是这样做的。标准库在<algorithm>头中定义了这个函数。

```
template <class Bi> void reverse(Bi begin, Bi end)
{
    while (begin != end) {
        -- end;
        if (begin != end)
            swap(*begin++, *end);
    }
}
```

在这个算法中，我们从向量的尾部自后而前地逆向推进 end 并从向量的开头正向推进 begin，在这个过程中，我们交换了它们所指向的元素。

这个函数以类似于正向迭代器的方式使用了迭代器 begin 和 end。除此之外，函数还使用了--——很明显，这是逆向遍历一个序列的关键所在。如果一个类型满足了一个正向迭代器的所有要求并且还支持--（前缀和后缀），那我们就把它称作**双向迭代器**。

所有的标准库容器类都支持双向迭代器。

8.2.6 随机访问

有些函数需要在容器内部的各处跳转。对此有一个很好的函数例子，这个例子就是那个经典的折半查找算法。标准库用几种形式实现了这个算法，而最为简单的那种被称为（正如你所料）binary_search。实际上，标准库的实现使用了一些聪明的技术（对此的讨论已经超出了本书的范围），这些技术让它可以对正向迭代器定义的序列做折半查找。一个更加简单的版本（要求有随机访问迭代器）如下所示：

```
template <class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x)
{
    while (begin < end) {
        // 查找区间的中点
        Ran mid = begin + (end - begin) / 2;

        // 看区间的哪一部分含有 x；只往下查找这一部分
        if (x < *mid)
            end = mid;
        else if (*mid < x)
            begin = mid + 1;
        // 如果我们在这里得到了待查找的值，那么*mid ==x，完成查找
        else return true;
    }
    return false;
}
```

除了要依赖普通的迭代器性质之外，这个函数还依赖于对迭代器进行算术运算的能力。例如，它把一个迭代器和另一个迭代器相减从而获得一个整数，同时它还把一个迭代器和一个整数相加从而获得另一个迭代器。同样地，逻辑完备性的概念也对随机访问迭代器增加了要求。如果 p 和 q 是这样的迭代器，同时 n 是一个整数，那么除了那些对双向迭代器的要求之外，其他的所有额外要求如下所示：

```
p + n，p - n和n + p
p - q
p[n]（与*(p + n)等价）
p < q，p > q和p >= q
```

把两个迭代器相减会产生迭代器之间的距离，这个距离是整数类型的——我们将会在§10.1.4 中讨论这一点。我们之所以不在这些要求中包含==和!=是因为，随机访问迭代器同时也支持了对双向迭代器的要求。

我们已经使用过一个要求有随机访问迭代器的算法，这个算法就是 sort 函数。向量和字符串迭代器都是随机访问迭代器。而链表迭代器则不是这样的迭代器：它只支持双向迭代器。为什么呢？

根本的原因是，链表是为了快速的插入和删除而被优化的。因此，我们无法快速地定位到表中的任意元素。定位链表中元素的惟一办法是按顺序查看每一个元素。

8.2.7 迭代器区间和越界值

像我们已经见到的那样，在库中有一个几乎是通用的约定：算法要用两个参数来指定区间。第一个参数指向区间的第一个元素；第二个参数则指向了紧位于区间最后的元素后面的那个位置。为什么我们要指定紧位于区间最后的元素之后的一个位置呢？在什么时候这样做才会是有效的呢？

在§2.6 中，我们看到了对区间使用上界（这个上界比区间的最后一个值大一个单位）的一个原因——也就是说，如果我们用一个值来指定了区间的末端，而这个值等于最后的那个元素，那么我们就不明确地表示了，最后的那个元素是相对特殊的。如果最后的那个值被特殊处理的话，那么在编写程序的时候我们就很可能会犯错，我们可能会在到达区间终点之前就停止了迭代器。就迭代器来说，至少有三个另外的原因可以解释为什么我们要用指示了紧位于区间最后一个元素后面的那个位置的迭代器，而不是用一个直接指向最后的元素的迭代器来标记区间终点。

第一个原因是，如果区间根本没有元素，那我们就找不到一个最后的元素来标记终点。这样的话，我们就会处于一个很古怪的位置上，我们将不得不用一个迭代器来指明一个空的区间，而这个迭代器将会指向紧位于区间开头之前的那个位置。如果我们采取这个策略，那我们就必须把空区间当作是和其他所有的区间都不同的特例来处理，这样会使我们的程序难以理解并且会降低程序的可靠性。在§6.1.1 中我们看到，以同样的方式来处理空区间和其他的区间一样会简化我们的程序。

第二个原因是，如果我们用一个迭代器（这个迭代器指向位于最后的元素之后的位置）来标记区间终点，那我们就可以仅仅为了判定相等或不相等而去比较迭代器，而且，使用了这种标记方法之后，我们就无须再定义“一个迭代器小于另一个迭代器”的具体意义了。最重要的是，在比较了两个迭代器之后我们就能马上判断出区间是否为空；只有在两个迭代器相等的情况下，区间才会为空。如果它们不相等，那么我们知道，那个开始迭代器指向了一个元素，因此，我们可以先做一些处理动作，然后我们能对这个迭代器加 1 从而缩短区间的长度。换句话说，如果我们用开头和紧位于末尾元素之后的位置来标记区间，那我们就可以使用以下形式的循环：

```
// 不变式：我们还需要处理在区间[begin,end)中的元素
while (begin != end) {
    // 处理begin所指向的元素
    ++begin;
}
```

我们只需要设法比较迭代器是否相等。

第三个原因是，如果我们用开头和紧位于末尾元素之后的位置来定义一个区间，那我们就能以一种自然的方式来表示“区间之外”。许多标准库算法（以及我们自己编写的算法）都利用了这个“区间之外”的值，为此它们返回一个区间的第二个迭代器来指示失败。例如，在§6.1.3中，我们的 url_beg 函数使用了这个约定来发出无法找到一个 URL 的信号。如果算法没有利用这个值，那么它们就必须创造一个，但是这样做又会增加算法以及使用这些算法的程序的复杂度。

总而言之，虽然使用指向区间末尾之后的一个元素的迭代器来指示一个区间的终点看起来可能有点古怪，但是，和其他的处理方式相比，这样做会使大多数的程序更加简单、更加可靠。到最后，每一种容器类型都要为了它的迭代器而支持一个越界值。每一个容器的 end 成员都会返回一个这样的值，并且这个值也可能是其他容器操作所产生的结果。例如，如果 c 是一个容器，那么复制 c.begin()并把复件递增 c.size()次将会产生一个等于 c.end()的迭代器。间接引用一个越界迭代器的作用是未定义的；同样，计算一个以下的迭代器值的作用也是未定义的：这个迭代器或者是在容器的开头元素之前，或者是位于容器末尾元素之后但和末尾元素的距离超过了一个元素单位。

8.3 输入输出迭代器

如果所有的标准容器都不要求输入输出迭代器和正向迭代器之间要有差别，那么，为什么还要有这两种种类呢？一个原因是，并不是所有的迭代器都是与容器相关联的。例如，如果 c 是一个支持 push_back 的容器，那么 back_inserter(c) 就是一个输出迭代器，这个迭代器并不满足任何其他的迭代器要求。

作为另一个例子，标准库提供了能被连接到输入和输出流的迭代器。毫不奇怪，用于 istream 的迭代器满足了输入迭代器的要求，而用于 ostream 的则满足了输出迭代器的要求。有了适当的迭代器，我们就能使用普通的迭代器操作来处理一个 istream 或一个 ostream 了。例如，++将把迭代器推进到流的下一个值。对于输入流，*将产生在输入当前位置的值；而对于输出流，*将让我们写数据到相应的 ostream 中。流迭代器是在<iterator>头中定义的。

输入流迭代器是一种名为 istream_iterator 的输入迭代器类型：

```
vector<int> v;

// 从标准输入中读整数值并把它们添加到 v 中
copy(istream_iterator<int>(cin), istream_iterator<int>(),
     back_inserter(v));
```

跟之前一样，copy 的开头两个参数指定了一个区间，函数从这个区间复制数据。第一个参数构造了一个新的 istream_iterator 类型的迭代器，这个迭代器被连接到 cin，它要求读进 int 类型的值。别忘了，C++输入和输出是带类型的操作：在我们从一个流读数据的时候，

要表明我们所期望读到的值的类型——尽管这些类型经常是隐含在读操作中的。例如，

```
getline( cin , s);   //读数据到一个字符串
cin >> s.name >> s.midterm >>s.final   //读一个字符串和两个double类型的值
```

同样地，在我们定义一个流迭代器的时候，我们必须设法告诉它应该从流读进什么类型的数据或应该写什么类型的数据到流。因此，流迭代器是模板。

copy 的第二个参数创建了一个缺省（为空）的 istream_iterator<int>类型的迭代器，这个迭代器不会与任何的文件连接在一起。istream_iterator 类型有一个缺省值，而这个缺省值有一个性质，就是 istream_iterator 类型的迭代器一旦到达了文件末尾或者是处于错误状态中，那它就会和这个缺省值相等。因此，我们能够用这个缺省值来为 copy 表示那个“超过末尾元素一个单位位置”的协定。

我们不能用一个 istream_iterator 类型的迭代器来进行写操作。如果我们希望写数据，那我们需要一个 ostream_iterator 类型的迭代器，我们用这种迭代器类型的对象来进行输出：

```
// 输出 v 的元素，元素之间用一个空格隔开
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

在这里，我们把整个向量复制到标准输出。第三个参数构造了一个新的迭代器，此迭代器被连接到 cout，它要求写 int 类型的值。

用来构造 ostream_iterator<int>类型的对象的第二个参数指定了一个值，这个值会被写到每一个元素之后。一般来说，这个值是一个字符串文字。如果我们不提供一个这样的值，那么这个 ostream_iterator 类型的迭代器在写数值的时候就不会带任何的分隔。因此，如果我们省略了分隔符，调用 copy 的时候所有的数值就会跑到一起，这些数值是乱七八糟而且是根本不可读的：

```
// 在两个元素之间并没有分隔！
copy(v.begin(), v.end(), ostream_iterator<int>(cout));
```

8.4 用迭代器来提高适应性

我们可以稍稍改进一下我们在§6.1.1 中介绍的那个 split 函数。正如代码所示，split 返回一个 vector<string>类型的向量，这种做法有其局限性：我们的用户可能不想要一个向量，相反，他们有可能希望得到一个 list<string>类型的链表或者是其他种类的容器。而且 split 算法根本没有要求我们非要产生一个向量不可。

我们可以重写 split 以获得更好的适应性，重写后的函数有一个输出迭代器而不是返回一个值。在这个版本的函数中，我们将使用这个迭代器来输出我们查找到的单词。我们的调用程序在调用前要把这个迭代器和放置数值的输出位置连接起来：

```
template <class Out>                    // 有改动
```

```
void split(const string& str, Out os) {    // 有改动
    typedef string::const_iterator iter;

    iter i = str.begin();
    while (i != str.end()) {
        // 忽略前端的空白
        i = find_if(i, str.end(), not_space);

        // 找出下一个单词的结尾
        iter j = find_if(i, str.end(), space);

        // 复制在[i,j)中的字符
        if (i != str.end())
            *os++ = string(i, j);          // 有改动

        i = j;
    }
}
```

像我们在§6.2.2 所编写的 write_analysis 函数一样，这个新版本的 split 没有返回语句，因此我们说它的返回类型是 void。现在我们已经把 split 变成一个模板函数了，它有一个类型参数 Out，这个参数的名称表明了它是一个输出迭代器。要记住，正向、双向和随机迭代器都满足所有的输出迭代器要求，这样的话，我们就能把我们的 split 函数用于任何种类的迭代器——但是纯粹的诸如 istream_iterator 这样的输入迭代器除外。

参数 os 的类型是 Out。我们将用它来输出我们所查找到的单词的值。我们在接近函数末尾的地方完成这个动作：

```
*os++ = string(i, j);          // 有改动
```

这条语句输出了我们刚刚找到的单词。子表达式*os 指示了 os 被连接到的容器的当前位置，因此，我们把值 string(i ,j)赋给位于这个位置的元素。完成了赋值操作之后，我们对 os 加 1，通过这样就满足了输出迭代器的要求，而且在下一个循环过程中函数会赋一个值给容器的下一个元素。

如果程序员希望使用这个经修正的 split 函数，那他们就必须修改他们的程序，不过，现在我们可以把单词写到几乎所有的容器中了。例如，如果 s 是一个字符串，而且我们希望把它们的单词添加到一个名为 word_list 的链表中，那么，我们就能以下面的形式来调用 split：

```
split( s , back_inserter(word_list) );
```

同样地，我们可以编写一个小程序来测试我们的 split 函数：

```
int main()
{
    string s;
```

```
    while (getline(cin, s))
        split(s, ostream_iterator<string>(cout, "\n"));
    return 0;
}
```

跟我们在§5.7 中编写的那个驱动函数一样，这个函数调用 split 来把输入行分割成独立的单词并把这些单词写到标准输出。我们通过传递一个 ostream_iterator 类型的迭代器给 split 从而把单词写到 cout——这里的迭代器会被连接到 cout。当 split 赋值给*os 的时候，实际上它是在写数据到 cout 中。

8.5 小结

返回简单类型的**模板函数**的形式如下：

```
template<class type-parameter, [, class type-parameter]...>
ret-type function-name(parameter-list)
```

每一个 type-parameter（类型参数）都是一个名称，我们可以在函数定义内的任何一个需要类型的位置上使用这个名称。每一个这样的名称都应该出现在函数的 parameter-name（参数列表）中，我们用它们来为一个或多个参数的类型命名。

如果这些类型不全部出现在参数列表中，那么调用程序就必须用具体的类型（我们无法推断到底是什么类型）来限定 function-name（函数名）。例如，

```
template<class T> T zero() {return 0;}
```

把 zero 定义成一个模板函数，它有一个类型参数，这个类型参数是用来为返回类型命名的。在调用这个函数的时候，我们必须明确地提供返回类型：

```
double x = zero<double> ();
```

如果我们在声明中使用了由模板类型参数定义的类型，那么就必须用关键字 typename 来限定这个声明。例如，

```
typename T::size_type name;
```

声明了 name 的类型是 size_type，这个类型必须被定义为一个在 T 内部的类型。

对于每一个在函数调用中使用的类型集合，系统环境会自动创建一个独立的模板函数实例。

迭代器：C++标准库的一个主要的贡献是，它确立了一种算法设计思想，也就是说，算法能够用迭代器来作为算法与容器之间的“粘合剂”从而获得数据结构的独立性。此外，算法所用到的迭代器都要求有某些操作，我们能以这些操作为基础而分解算法，这就意味着我们可以把一个容器和能够使用这个容器的算法匹配起来——这是很容易做到的。

迭代器可以分为五个种类。一般来说，后面的种类都包含了前面的种类所含有的操

输入迭代器：	按一个方向顺序地访问，只能输入
输出迭代器：	按一个方向顺序地访问，只能输出
正向迭代器：	按一个方向顺序地访问，既能输入也能输出
双向迭代器：	按两个方向顺序地访问，既能输入也能输出
随机访问迭代器：	能有效地访问任何元素，既能输入也能输出

习题

8-0 编译、运行并测试本章中的程序。

8-1 注意一下，我们在§6.2 中编写的各个 analysis 函数都具有同样的行为特性；这些函数的差异在于：它们调用了不同的函数来计算总成绩。编写一个模板函数，根据计算成绩函数的类型而对这个函数进行参数化，并且用这个函数来评估那些计算成绩方案。

8-2 实现下面的库算法——我们在第 6 章使用过它们，在§6.5 中描述了它们。指定它们所要求的迭代器种类。尽量把把每一个函数所需要的不同的迭代器操作的个数降到最少。在实现了这些算法之后，对照一下§B.3，看看你做得好不好。

```
equal(b, e, d)                search(b, e, b2, e2)
find(b, e, t)                 find_if(b, e, p)
copy(b, e, d)                 remove_copy(b, e, d, t)
remove_copy_if(b, e, d, p)    remove(b, e, t)
transform(b, e, d, f)         partition(b, e, p)
accumulate(b, e, t)
```

8-3 正如我们在§4.1.4 中所了解到的那样，按值返回（或传递）一个容器的代价可能是很大的。然而，我们在§8.1.1 中编写的 median 函数还是按值传递了向量。我们可以重写这个 median 函数，让它对迭代器操作而不是传递向量吗？如果我们这样做，那么，你觉得对性能的影响将会是怎样的呢？

8-4 实现我们在§8.2.5 中使用的 swap 函数。为什么我们要调用 swap 而不是直接交换*beg 和*end 的值呢？提示：实践一下，看看会有什么样的结果吧。

8-5 重新实现第 7 章的 gen_sentence 和 xref 函数，使用输出迭代器而不是把它们的输出直接写到一个 vector<sting>类型的向量。测试这些新版本，编写程序来把输出迭代器与标准输出直接连接起来，把结果分别存储在 list<string>类型和 vector<string>类型的变量中。

8-6 假定 m 的类型是 map<int , string>，而且我们遇到了一个调用 copy(m.begin() , m.end() , back_inserter(x))，那我们应该怎样看待 x 的类型呢？如果这个调用的形式是 copy(x.begin() , x.end() , back_inserter(m))的话那又怎样呢？

8-7 为什么 max 函数不使用两个模板参数（各用于一个参数类型）呢？

8-8 在§8.2.6 的 binary_search 函数中，我们为什么不编写(begin + end)/2 来代替那个更为复杂的 begin + (end – begin)/2 呢？

第 9 章

定义新类型

C++的类型可以分为两个种类：内部类型和自定义类型（class type）。之所以叫内部类型是因为，它们是被定义成语言核心的一部分的。内部类型包括了 char、int 以及 double。我们已经使用过的库的类型——例如 string、vector 和 stream 都是自定义类型。除了在输入-输出库中的某些低级的、系统专用的例程之外，库中的类所依赖的语言工具跟任何程序员能用来定义应用专用的类型的语言工具是一样的。

许多 C++设计都依赖于这样的一种设计思想：我们应该让程序员创建跟内部类型一样易于使用的类型。正如我们即将看到的一样，创建具有简明直观的接口的类型除了要求有实质性的语言支持之外还要求我们具有在类的设计过程中的体验和判断。我们将首先应用第 4 章的那个计算成绩的问题来对那些最为基本的类定义工具展开讨论。从第 11 章开始，我们会通过逐步建立和库中一样完整的类型来建立起这些基本概念。

9.1 回顾一下 Student_info

在§4.2.1 中，我们编写了一个简单的名为 Student_info 的数据结构以及少数的函数，在我们编写程序来处理学生的课程成绩的时候，这些函数为我们提供了方便。然而，我们编写的这些数据结构和函数对于其他的程序员来说不是很合适，他们并不能方便地使用它们。

不管我们是否认识到了这一点，希望使用我们的函数的程序员都还是必须遵从某些约定。例如，我们假定，任何使用了一个新创建的 Student_info 的人都将首先读数据给它。如果这个动作失败，那就产生具有一个空的 homework 向量的对象，并且 midterm 和 final 的值将会是未定义的。任何对这些值的使用都将产生不可预测的行为——或者是出现不正确的结果，或者是彻底的崩溃。此外，如果用户想要检查一个 Student_info 是否包含了有效的数据，那么惟一的办法是查找真实的数据成员，而这又要求用户对 Student_info 的实现有详细的了解。

一个相关的问题是，使用我们的程序的人很可能会假定，一旦某个学生的记录已经从一个文件中读进来了，那以后这个学生的数据就不会改变。不幸的是，我们的代码没有为这个假定提供基础。

第三个问题是，我们原来的 Student_info 结构的“接口”是很分散的。通过约定，我们都

能把那些会改变一个 Student_info 对象的状态的函数（例如 read）放进一个单独的头文件中。这样做将会对那些随后使用我们的代码的用户有所帮助——如果我们决定这样做的话——不过，我们没有必要进行这样的分类。

正如我们将在本章中所看到的一样，我们能扩充 Student_info，这样的话，我们就可以解决所有的这些问题了。

9.2 自定义类型

从表面上来看，自定义类型是一种用来把相关的数据值组合在一个数据结构中的技术，有了这种技术，我们就能把这个数据结构当作一个单独的实体来处理。例如，在§4.2.1 中我们建立的 Student_info 结构

```
struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

使我们可以定义和操作 Student_info 类型的对象。这种类型的每一个对象都有四个数据元素：一个是 name，类型为 std::string；一个是 homework，类型为 std::vector<double>；还有两个是 midterm 和 final，类型都为 double。

使用 Student_info 类型的程序员可以（而且必须）直接地操作这些数据元素。程序员之所以可以直接地操作数据，是因为 Student_info 的定义并没有限制对数据元素的访问；之所以必须这样做，是因为对于 Student_info 并没有其他可用的操作。

我们不打算让用户直接访问数据，相反，我们希望把 Student_info 对象的存储方式的实现细节隐藏起来。特别地，我们要求这个类型的用户仅仅通过函数来访问对象。为此，我们首先要向用户提供方便的对 Student_info 对象的操作。这些操作将构成我们的类接口。

在了解这些函数之前，我们有必要回顾一下，为什么我们要使用完整的限定名 std::string 和 std::vector，而不是假定我们已经有了一个允许我们直接访问名字的 using 声明呢？使用了 Student_info 结构的代码都必须访问类的定义，因此，我们将把这个定义放在一个头文件中。正如我们在§4.3 中所指出的那样，供他人使用的代码应该包含最少数量的必要声明。很明显，我们必须定义名称 Student_info，因为这个名称是供用户使用的。在 Student_info 中使用了 string 和 vector 这个事实是由人为因素造成的，我们没有理由仅仅因为我们在实现的对候使用这些类型而把 using 声明强加给 Student_info 的用户。

在程序设计实例中，作为一个良好的习惯，我们在头文件的代码中使用限定名，不过我们将继续假定，对应的源文件包含了适当的 using 声明。因此，如果我们打算让程序正文出现在头文件之外，那么在编写的时候我们通常不会使用完整的限定名。

9.2.1 成员函数

为了控制对 Student_info 对象的访问，我们需要定义一个可供程序员使用的接口。让我们先定义一些操作来读一条记录并计算总成绩：

```
struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;

    std::istream read(std::istream&);        // 新增的
    double grad() const;                     // 新增的
};
```

在这里，每一个 Student_info 对象还是有四个数据元素，不过我们也给了 Student_info 两个成员函数。它们让我们从一个输入流读一条记录，并为任何的 Student_info 对象计算总成绩。在 grade 的声明中的 const 确保了对 grade 函数的调用将不会修改 Student_info 对象的任何数据成员。

在§1.2 中，当我们讨论对类 string 的 size 成员的使用的时候，我们第一次探讨了成员函数。从本质上说，一个成员函数是类对象的一个成员。为了调用一个成员函数，我们的用户必须指明被调用的函数是对象的一个成员。因此，类似于对一个名为 greeting 的 string 对象调用 greeting.size()，我们的用户会对一个名为 s 的 Student_info 对象调用 s.read(in)或 s.grade()。s.read(in) 调用从标准输入读数值并适当地设置 s 的状态。s.grade()调用则会为 s 计算并返回总成绩。

我们的第一个成员函数的定义看起来和§4.2.2 中原先的那个版本很相似：

```
istream& Student_info::read(istream& in)
{
    in >> name >> midterm >> final;
    read_hw(in, homework);
    return in;
}
```

跟我们原先所做的一样，我们将把这些函数放置在一个名为 Student_info.cpp、Student_info.c 或 Student_info.C 的源文件中。重要的是，这些函数的声明现在是我们的 Student_info 结构的一部分，因此它们对 Student_info 类的所有用户都必须是可用的。

我们可以从下面的三个地方来比较这个代码和原先的那个：

1. 函数名是 Student_info::read 而不是 read。

2. 因为这个函数是 Student_infl 对象的一个成员，所以我们不需要把一个 Student_info 对象作为参数来传递，而且我们根本不需要定义一个 Student_info 对象。

3. 我们直接地访问对象的数据元素。例如，在§4.2.2 中，我们引用了 s.midterm；

我们只是引用了 midterm。

我们将分别解释这三点差别。

在这个函数名中的::是一个作用域运算符，早在§0.7 中我们就已经使用过这个运算符了，在那里，我们用它来访问标准库定义的名称。例如，我们可以编写 string::size_type 从而获得类 string 的一个成员名称 size_type。同样地，通过编写 Student_info::read，我们就定义了名为 read 的函数，这个函数是 Student_info 类型的一个成员。

因为 Student_info&参数将隐含在任何的调用中，所以这个成员函数只要求有一个 istream&类型的参数。别忘了，在我们调用向量或字符串对象的一个成员函数的时候，我们必须指明我们想要的是哪一个向量或字符串。例如，如果 s 是一个字符串，那么我们可以编写 s.size()来调用对象 s 的 size 成员。如果我们不指明一个字符串对象，那我们就无法从 string 类调用这个 size 函数。同样地，在我们调用 read 函数的时候，必须明确地指明我们是在把数据读到哪一个 Student_info 对象中。这个对象会隐含地在 read 函数中被使用。

在 read 内部引用成员是无需使用限定形式的，这是因为，我们引用的是那个正在由我们操作的对象的成员。换句话说，如果我们对一个名叫 s 的 Student_info 对象调用 s.read(in)，那么我们就是在操作对象 s。当我们在 read 使用 mditerm、final 以及 homework 的时候，实际上我们就是在分别使用 s.midterm、s.final 以及 s.homework。

现在，让我们来看看 grade 成员：

```
double Student_info::grade() const
{
    return ::grade(midterm, final, homework);
}
```

这个版本类似于§4.2.2 中的那个，跟我们刚才提到的 read 版本的差别相似，这两个 grade 版本的差别是：在这里，我们把 grade 定义成 Student_info 的一个成员，这个函数隐含地（而不是明确地）引用了一个 Student_info 对象，而且它不需要使用限定形式来访问这个对象的成员。

这个代码还包含了另外两个重要的差别。首先，让我们留意一下对::grade 的调用。如果我们把::放在一个名称之前，那就表明了我们要使用这个名称的某一个版本，而所使用的这个版本不能是任何事物的成员。这样的话，如果我们希望在这个调用中使用在§4.1.2 中定义的那个版本的 grade 函数——这个版本有两个 double 类型的参数和另一个 vector<double>类型的参数，那我们就要使用::。否则，编译程序将会认为我们所指的是 Student_info::grade，而且它会提示出错——因为我们在调用时使用了过多的参数。

另一个的差别是，在这个版本中，我们在 grade 的参数列表后面使用了 const。把新的函数声明和原先的那个比较一下，我们就能理解这个用法：

```
double Student_info::grade() const {…} // 成员函数版本
double grade(const Student_info&) {…}  // 在§4.2.2 中的原先的版本
```

在原先的函数中，在传递 Student_info 的时候我们把它当作一个 const（常量）引

样我们就确保了我们能够请求带有 const Student_info 对象的 grade 函数，而且，如果这个 grade 函数试图修改它的参数，编译程序就会提示出错。

在我们调用一个成员函数的时候，如果这个函数是某个对象的成员，那么这个对象就不能是一个参数。因此，我们无法在参数列表中指明这个对象是 const。相反，我们对函数本身作了限制，这样我们就能让它成为一个**常量(const)成员函数**。常量成员函数不可以改变正在由它们执行的对象的内部状态：如果我们对一个名叫 s 的 Student_info 对象调用了 s.grade()，那我们就要确保，这样做将不会修改 s 的数据成员。

因为这个函数保证了它不会修改对象的值，所以我们可以用常量对象调用它。出于同样的原因，我们不能对常量对象调用非常量函数。例如，我们不能对一个常量 Student_info 对象调用那个 read 成员。毕竟，一个诸如 read 这样的函数表明了它能够修改对象的状态。对一个常量对象调用这样的一个函数可能会破坏我们所承诺的常量性。

另外值得我们注意的是，即使一个程序从未直接创建过任何的常量对象，它还是有可能在函数调用的过程中创建许多对常量对象的引用。如果我们把一个非常量对象传递给一个具有常量引用参数的函数，那么这个函数就会把这个对象当作常量来处理，同时编译程序将允许它只调用这种对象的常量成员。

注意，我们在类定义内部的函数声明和函数的定义中都包括有限定词 const。而且，在函数声明和函数定义中的参数类型必须总是相同的(§4.4)。

9.2.2 非成员函数

我们在新的设计中把 read 和 grade 变成了成员函数。对于 compare 我们又应该如何处理呢？它也应该作为类的一个成员吗？

跟我们即将在§9.5、§11.2.4、§11.3.2、§12.5 以及§13.2.1 中看到的那样，C++语言要求某些种类的函数要被定义为成员。事实证明，compare 并不是一个这样的成员，因此我们可以选择用其他形式来实现它。有一条通用规则可以帮助我们判断对这样的情况应采取什么样的措施：如果函数会改变一个对象的状态，那它就应该作为这个对象的成员。不幸的是，即使是这条规则也没有说明我们应该怎样处理那些不会改变对象状态的函数，因此我们仍要决定应该如何处理 compare。

为此，我们不妨先考虑一下这个函数是做什么的以及用户可能会希望以怎样的方式来调用它。compare 函数判断了在它的 Student_info 参数中“比较小”的是哪一个，这个判断基于对参数的 name 成员的检查。在§12.2 中我们将会看到，有时候在类实体之外定义诸如 compare 这样的操作会有一个好处。因此，我们将让 compare 作为一个全局函数，并且我们即将会实现它。

9.3 保护

把 grade 和 read 函数定义为成员之后，我们就已经修正了问题的一半了：Student_

型的用户不再需要直接操控对象的内部状态了。当然，他们仍然可以这样做。我们希望把数据隐藏起来并允许用户仅仅通过我们的成员函数访问这些数据。

C++支持数据隐藏机制，它允许类型的作者指明类型的哪些成员是**公有的（public）**以及哪些成员是**私有的（private）**。类型的所有用户都可以访问公有成员；私有成员对类型的用户来说则是不可访问的：

```
class Student_info {
public:
    // 类型提供的接口
    double grade() const;
    std::istream& read(std::istream&);

private:
    // 类型的实现
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

我们对 Student_info 作了两个改动：我们使用了 class 来代替 struct，同时我们增加了两个**保护标识符**。每一个保护标识符都定义了跟在此标识符后面的所有成员的可访问性。标识符能在类的内部以任何顺序出现，同时它们也可以出现多次。

如果我们把 name、homework、midterm 以及 final 放在一个 private 标识符后面，那么对 Student_info 类型的用户来说，这些数据元素就是不可访问的。现在，在非成员函数中引用这些成员是非法的，而且编译程序将会生成一条诊断信息来说明成员是私有的或不可访问的。在 public 程序段中的全部成员都是可访问的；任何用户都可以调用 read 或 grade。

用 class 来代替 struct 又是怎么回事呢？我们可以使用任意一个这样的关键字来定义一个新的类型。使用 struct 和 class 的惟一差别是：如果在第一个保护标识符之前有成员，那么对于 struct 和 class 来说，应用于这些成员的缺省保护方式是彼此不同的。如果我们使用 class Student_info，那么，在第一个{和第一个保护标识符之间的所有成员都是私有的。相反，如果我们编写了 struct Student_info，那么在{和第一个保护标识符之间声明的所有成员都是公有的。例如，

```
class Student_info {
public:
    double grade( ) const;
    // 等等
};
```

等价于

```
struct Student_info {
```

```
    double grade( ) const;      // 缺省为公有
    // 等等
};
```

同时，

```
class Student_info {
    std::string name;           // 缺省为私有
    // 其他私有成员
public:
    double grade( ) const;
    // 其他公有成员
};
```

等价于

```
struct Student_info {
private:
    std::string name;
    // 其他私有成员
public:
    double grade( ) const;
    // 其他公有成员
};
```

在所有的这些定义中，我们都允许用户访问 Student_info 对象的成员函数，但是我们不允许他们访问数据成员。

我们能对一个结构(struct)或一个类(class)做相同的处理动作。实际上，除非是阅读了代码，否则我们的用户都无法辨别，在定义自定义类型的时候我们是使用了 struct 还是使用了 class。我们对结构或类的选择可以对我们的程序设计起到很好的指示作用。一般而言，我们的程序设计风格是保留结构以指示简单的类型，并且我们希望公开这些类型的数据结构。出于这个原因，在第 4 章我们使用了结构来定义我们原先的 Student_info 数据类型。既然我们想构造一个类型并控制对其成员的访问，那我们就应该使用类来定义我们的 Student_info 类型。

9.3.1 存取器函数

目前为止，我们已经隐藏了我们的数据成员，这样用户就不能再修改 Student_info 对象的数据了。相反，他们必须使用 read 操作来设置数据成员，而且他们还要使用 grade 来为一个特定的 Student_info 对象找出其总成绩。我们还必须提供另外的一个操作：我们必须设法让用户可以访问到学生的姓名。例如，考虑一下§4.5 中的程序，在那里我们编写了一个格式化的学生成绩报表。为了生成报表，那个程序需要访问学生的姓名。我们希望允许读访问，不过我们不愿意允许写访问。下面的做法比较直截了当：

```
class Student_info {
```

```
public:
    double grade() const;
    std::istream& read(std::istream&);      // 必须修改定义
    std::string name() const { return n; } // 新增的
private:
    std::string n;                          // 有改动
    double midterm, final;
    std::vector<double> homework;
};
```

为了不让用户访问数据成员 name，我们增加了一个也是叫做 name 的成员函数，这个函数让用户可以（只读地）访问相应的数值。当然，为了避免数据成员的名称和函数的名称发生混淆，我们必须修改相应的数据成员的名称。

name 函数是一个不带参数的常量成员函数，它返回一个字符串，这个字符串是 n 的一个复件。如果我们复制 n 而不是返回它的一个引用，我们就确保了，用户能读但不能修改 n 的值。因为我们只需要对数据的读访问，所以我们把这个成员函数声明为一个常量函数。

我们在类定义以外的地方完成了对 grade 和 read 的定义。跟我们在这里对 name 函数所做的一样，如果我们定义了一个成员函数以作为类定义的一部分，那么，实际上我们是在提示编译程序，它应该避免函数调用的系统开销并且在可能的情况下把调用扩展为函数内联子过程（§4.6）。

诸如 name 这样的函数经常被称为**存取器函数**。这个名称有可能会引起误解，因为它意味着我们容许对我们的一部分数据结构的访问。历史上，我们经常利用这样的函数来允许对被隐藏起来的数据的简单访问，而这样做会破坏我们正在追求的封装性。就 Student_info 而言，我们的抽象就是对一个学生以及一个对应的总成绩的抽象。因此，一个学生姓名的概念就是我们的抽象的一部分，这样的话，提供 name 函数会是一个适当的做法。另一方面，我们并没有对其他成绩提供访问器——midterm、final 或 homework。这些成绩是我们的实现的一个基本的组成部分，但它们不是我们的接口的一部分。

增加了 name 成员函数之后，我们就能编写 compare 函数了：

```
bool compare(const Stduent_info& x, const Student_info& y)
{
    return x.name() < y.name();
}
```

这个函数看起来和§4.2.2 中的那个版本很相似。惟一的差别是，我们现在用了不同的方法来访问学生的姓名。在原先的版本中，我们可以直接地访问这个数据成员；而在这里，我们必须调用 name 函数，这个函数会返回学生的姓名。因为 compare 函数是我们的接口的一部分，所以我们应该在那个定义了 Student_info 的头中把这个函数的声明包含进去，同时这个定义还应该包含在相关的包括了成员函数定义的源文件中。

9.3.2 检查对象是否为空

在隐藏了我们的成员并提供了适当的访问器函数之后，我们还要解决剩下来的一个问题：用户仍然可以找到一个理由来要求直接地查看一个对象的数据。例如，思考一下，如果我们在还没有对一个对象调用 read 的情况下就调用了 grade，那么这样做有什么后果呢？

```
Student_info s;
cout << s.grade( ) << endl;  // 异常：s 没有数据
```

因为我们还没有调用 read 来为 s 赋值，所以 s 的 homework 成员将会为空，同时对 grade 的调用会引发一个异常。虽然我们的用户能捕获这个异常，但是他们还是无法预先检测到这个问题，这样的话他们就有可能根本不敢进行这个调用。

用户可以用第 4 章中原先的那个 Student_info 结构来检查 homework 成员，这个检查可以判断对 grade 的调用是否会成功。如果 homework 真的为空，那么他们就知道不应该调用 grade。这个办法确实有效，但这样做是有代价的——为了完成这个检查，用户必须要了解对象的结构。如果我们以更为抽象的形式提供这个同样的检测，就可以获得更为理想的效果：

```
class Student_info {
public:
    bool valid() const { return !homework.empty(); }
    // 跟之前一样
}
```

这个 valid 函数将告诉用户对象是否包含了有效的数据。如果它的返回值为 true，那么就表示这个学生做了至少一次的家庭作业，这样的话用户就可以计算这个学生的成绩了。我们的用户可以调用 valid 来判断随后的操作是否会成功。例如，在调用 grade 之前，用户可以检查对象是否为空从而避免了一个潜在的异常。

9.4 Student_info 类

到现在为止，我们已经消除了在原先的 Student_info 结构中的大多数缺陷，因此应该回顾一下，看我们已经做了些什么：

```
class Student_info {
public:
    std::string name() const { return n; }
    bool valid() const { return !homework.empty(); }
    // 其定义可以从 §9.2.1/157 中得到，不过在此是读到 n 中而不是读到 name 中
    std::istream& read(std::istream&);
    double grade() const;        // 其定义位于 §9.2.1/158 中
private:
    std::string n;
```

```
    double midterm, final;
    std::vector<double> homework;
};
bool compare(const Student_info&, const Student_info&);
```

用户只能通过调用 read 成员函数而改变 Student_info 对象的状态。他们不能到达对象的内部从而直接修改任何的数据成员。用户不再需要了解我们的实现细节[1]；相反，我们提供的操作会把我们的实现隐藏起来。最后，对 Student_info 对象的所有操作从逻辑上来说都是聚集在一起的。

9.5 构造函数

虽然我们的类已经相当完整而且便于使用，不过还是要考虑一下另外的一个问题：我们还没有讨论过，在对象被创建的时候将会出现什么样的情况。

我们都知道，在我们定义库的一个类对象时，库会保证让这个对象有一个适当的初始值。例如，如果我们在定义字符串或向量的时候不指定一个初始值，那我们就会得到一个空字符串或向量。string 和 vector 类型都允许我们给一个新对象指定一个初始值，例如我们可以指定一个长度或指定一个数量以及一个填充字符。

构造函数是一个特殊的成员函数，它定义了对象的初始化方式。一个构造函数不能被显式地调用。相反，在创建一个自定义类型的对象的时候，作为其副作用，一个适当的构造函数会被自动调用。

如果我们没有定义任何的构造函数，那么编译程序就会为我们合成一个。在§11.3.5 中我们将会对合成操作进行更深入的讨论。现在我们需要了解的是，如果我们不定义任何构造函数的话那将会发生什么。在我们的例子中，我们的用户能够定义 Student_info 对象，但他们不能明确地初始化这些对象——除非是作为其他 Student_info 对象的复制。

合成的构造函数将初始化数据成员，成员的初始值取决于对象的创建方式。如果对象是一个局部变量，那么数据成员将会被缺省初始化(§3.1)。如果是下面的三种情况中之一，那么对象的成员会被数值初始化（§7.2）：第一种情况是，对象被用来初始化一个容器元素；第二种是，为映射表添加一个新元素，而对象是这个添加动作的副作用；第三种则是，定义一个有特定长度的容器，对象是这个容器的元素。这些规则有些复杂，但它们的实质不外是：

- 如果对象属于一种自定义类型，而这种自定义类型定义了一个或多个构造函数，那么合适的构造函数就完全控制了对类的对象的初始化。
- 如果对象属于内部类型，那么数值初始化方式会把它设为零，而缺省初始化方式会给它一个未定义的值。

[1] 这种说法不够精确：如果只是编写使用 Student_info 的代码，用户可以不需要知道它的实现细节；为程序准备一些输入文件的话，那最好还是了解一些实现细节为好。

- 否则，对象就只能是属于未定义任何构造函数的自定义类型。这样的话，对这个对象的数值或缺省初始化操作就会对它的每一个数据成员进行相应的数值或缺省初始化。如果有任一个数据成员属于一种本身具有构造函数的自定义类型的话，那这个初始化过程都将会是递归的。

从上面可以看出，我们的 Student_info 类属于第三种情况：它是一个自定义类型，但是我们并没有明确地说明 Student_info 对象的构造方式。因此，如果我们定义了一个局部的 Student_info 变量，那么 n 和 homework 就会分别被自动地初始化成空的字符串和向量，这是因为它们是具有构造函数的类对象。与之相反，对 midterm 和 final 的缺省初始化将会给它们未定义的值。这就意味着它们将保存存储在它们被创建时所获得的内存区域中的任何无用信息。

就我们那些简单的操作来说，这个行为看起来是没有什么害处的：我们用 read 来为这些成员赋值，而在调用 read 来为对象进行第一次初始化之前，我们的任何一个操作都不会用到 midterm 和 final 的值。不过，我们通常应该养成一个良好的习惯——也就是说，我们应该确保所有的数据成员在任何时候都会具有有意义的值。例如，以后我们（或者是我们代码的维护人员）有可能要增加一些操作来检查这些数据成员。如果我们不是在构造函数中对它们进行初始化的，那么这些新的操作在以后就有可能会导致失败。此外，正如我们将会在§11.3.5 中所看到的那样，即使我们不会明确地使用 midterm 和 final，但可能还是会有某些合成操作对这个类型做这样的动作。除了写数据到未定义的值之外，其他的任何用法都是不合法的（§3.1）。因此，严格来说，我们必须初始化这些值。

实际上，我们希望定义两个构造函数：第一个不带参数，它创建了一个空的 Student_info 对象；第二个函数具有一个对输入流的引用，它从这个流中读进一条学生记录从而初始化了这个对象。

```
Student_info s;              // 一个空的 Student_info 对象
Student_info s2(cin);        // 从 cin 读数据，初始化 s2
```

构造函数跟其他成员函数有两点不同：它们的名称和类本身的名称相同；它们没有返回类型。我们能定义多个版本的构造函数，这些版本在参数个数或参数类型这两方面会有所不同——构造函数的这个性质跟其他函数是类似的。拥有了这些知识，现在我们就能更新我们的类了，我们增加了两个构造函数：

```
class Student_info {
public:
    Student_info( );                  // 构造一个空的 Student_info 对象
    Student_info(std::istream&);      // 读一个流从而构造一个对象
    // 跟之前一样
};
```

9.5.1 缺省构造函数

那个不带参数的构造函数被看作是缺省构造函数。它的工作通常是确保其对象的数据成员被正确地初始化。就 Student_info 对象而言，我们希望初始化数据以表示我们还没有读到记录：我们希望让 homework 成员成为一个空的向量，n 成员成为一个空字符串，而 midterm 和 final 成员则被初始化为零：

```
Student_info::Student_info( ): midterm(0) , final(0 ) { }
```

在这个构造函数的定义中我们使用了一些新的语法。在:和{之间有一系列的**构造函数初始化程序**，它们命令编译程序初始化给定的成员，并且在初始化的时候使用出现在相应的括号之间的值。因此，这个特殊的缺省构造函数显式地把 midterm 和 final（分别是期中、期末考试成绩）设为 0。除了这个动作之外，这个构造函数并没有做其他的工作：函数体是空的。正如我们即将看到的那样，n 和 homework 是被隐式初始化的。

要理解对象的创建以及初始化方式，关键是要先理解构造函数初始化程序。在我们创建一个新的类对象的时候会有以下几个连续的步骤：

1. 实现分配内存以保存这个对象。
2. 按照构造函数初始化程序列表而对对象进行初始化。
3. 执行构造函数的函数体。

实现会初始化每一个对象的所有数据成员——不管这些成员有没有在构造函数初始化程序列表中出现。构造函数的函数体有可能会随后改变这些初始值，不过这个初始化动作是在构造函数的函数体开始执行之前发生的。一般来说，在构造函数的函数体中为一个成员赋值并不是太理想的做法，更好的做法是明确地为成员指定一个初始化值，这样我们就避免了做两次同样的工作。

我们说，如果构造函数要退出，那它必须先确保对象已经被创建并且它们的数据成员都已经处于有意义的状态。一般来说，这个设计目标意味着，每一个构造函数都应该初始化所有的数据成员。而对于内部类型的成员来说，构造函数就更应该为每一个这样的成员指定一个值。如果构造函数没有初始化这样的成员，那么在局部作用域内声明的对象的初始化值将会是内存中的无用信息，而且这些信息十有八九都是不正确的。

现在我们明白了，为什么我们要说 Student_info 的缺省构造函数并没有做其他“公开的”工作。我们只是显式初始化了 midterm 和 final，其他的数据成员的初始化工作则是隐式进行的。尤其值得我们注意的是 n 和 homework，对这两个成员的初始化工作分别是由 string 缺省构造函数和 vector 缺省构造函数完成的。

9.5.2 带参数的构造函数

我们的第二个 Student_info 构造函数则更为简单：

```
Student_info::Student_info(istream& is) { read(is); }
```

这个构造函数把实际的工作交给 read 函数去完成。在函数中并没有显式的初始化程序，因此 homework 和 n 成员的初始化工作将会分别由 vector 和 string 的缺省构造函数来完成。只有在对象被数值初始化的情况下，midterm 和 final 成员才会有显式的初始化值。不过，就算没有这个初始化工作也是不要紧的，因为 read 会马上给这些变量指定新的值。

9.6 使用 Student_info 类

现在，和第 4 章中原先的那个 Student_info 结构相比，我们的新的 Student_info 类已经发生了很大的改变。毫无疑问，使用类和使用原先的结构是完全不同的两回事。毕竟，我们的目标是不让用户修改我们的数据值——为此我们把这些数据设为私有的。相反，我们打算让用户按照我们的类所提供的接口来编写他们的程序。例如，我们可以重新编写在§4.5 中原先的那个 main 函数，它使用了这个版本的类并在一个格式化的报表中输出了学生的总成绩：

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;

    // 读并存储数据
    while (record.read(cin)) {                          // 有改动
        maxlen = max(maxlen, record.name().size();      // 有改动
        students.push_back(record);
    }

    // 按字母顺序排列学生记录
    sort(students.begin(), students.end(), compare);

    // 输出姓名和成绩
    for (vector<Student_info>::size_type i = 0;
         i != students.size(); ++i) {
        cout << setw(maxlen + 1) << students[i].name;  // 有改动
        try {
            double final_grade = students[i].grade();  // 有改动
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec) << endl;
        } catch (domain_error e) {
            cout << e.what() << endl;
        }
    }
```

```
    return 0;
}
```

这里的变化是，对 name、read 以及 grade 的调用已经跟原先不同了。例如，现在的第一个 while 循环是

```
while (record.read (cin) ) {
```

而不是

```
while (read (cin , record) ) {
```

修改后的版本调用了对象 record 的成员 read。早期的版本则调用了全局的 read 函数，并且把 record 当作一个显式的参数而传递给这个函数。这两个调用都有同样的效果：从 cin 中读到的值将会被赋给对象 record。

9.7 小结

用户自定义类型可以被定义成结构(struct)或者类(class)。两者之间的惟一差别是：如果在第一个保护标识符之前有成员，那么对于 struct 和 class 来说，应用于这些成员的缺省保护方式是彼此不同的。在 struct 之后定义的成员是公有的；在 class 之后定义的那些则是私有的。

保护标识符控制了对一个类类型的成员的访问方式：公有(public)成员通常是可访问的；私有(private)成员仅仅对类的成员才是可访问的。在一个类中，保护标识符能以任何的顺序出现，同时它们也可以出现多次。

成员函数：跟数据一样，类型也可以定义成员函数。对一个特定对象的成员函数调用实际上隐含着对这个对象的引用。在一个成员函数内部引用数据成员或函数成员的时候，这些引用会隐含地被绑定到调用该函数的对象上。

成员函数既可以在类定义的内部定义，也可以在类定义以外的地方定义。如果我们在类的内部定义了一个成员函数，那实际上我们就是请求了实现把对这个函数的调用扩展成内联子过程，这样就避免了函数调用的额外开销。如果成员函数是在类的外部定义的，那么我们必须在这个函数的函数名中指明它是来自类作用域的："class-name：： member-name"指的是类 class-name 中的成员 member-name。

成员函数可以被定义成 const(常量)，为此我们要在函数的参数列表之后插入关键字 const。成员函数的调用都是对于某一个对象进行的，而这样的成员函数不可以改变相应的对象的状态。对于常量对象，我们只能调用常量函数。

构造函数是特殊的成员函数，它定义了类型的对象是如何被初始化的。构造函数的函数名跟类的名称一样，而且它没有返回值。一个类能定义多个构造函数——只要这些构造函数的参数类型或参数个数有所不同。我们应该养成一个这样的好习惯——就是说，每一个构造函数在退出前都应该先保证所有的数据成员都具有了有意义的值。

构造函数初始化程序列表：构造函数初始化程序是一个用逗号分隔的“member-name（value）”的列表。每一个 member-name（成员名）所表示的成员都会被初始化成相关联的 value（值）。如果数据成员不是被显式初始化的，那它们就会被隐式地初始化。

成员初始化的先后顺序是由类声明的顺序决定的，因此，当我们使用一个类成员来初始化另一个成员的时候，我们必须加以小心。较为安全的做法是避免这样的相互依赖，为此，我们应该在构造函数的函数体内给这些成员赋值，而不应该在构造函数初始化程序中初始化它们。

习题

9-0 编译、运行并测试本章中的程序。

9-1 重新实现 Student_info 类，让它在读学生记录的时候计算总成绩，把成绩存储在对象中。重新实现 grade 函数，让它使用这个预计算的值。

9-2 如果我们把 name 函数定义成一个简单的、非常量的成员函数，那么在我们的系统中，其他的函数必须做什么改动以及为什么要做这些改动呢？

9-3 如果在还没有读入一个 Student_info 对象的值的情况下，一个用户就试图对这个对象计算成绩，那么我们的 grade 函数就会引发一个异常。这就要求用户自己去捕获这个异常。编写一个程序来触发这个异常但不捕获它。然后，编写另一个程序来捕获这个异常。

9-4 重新编写在前面的练习中的程序，使用 valid 函数来避免异常的发生。

9-5 编写一个类以及相关的函数来为学生产生成绩，用 pass（及格）/ fail（不及格）来表示成绩。假定只根据期中和期末成绩来计算，而且，如果一个学生的平均考试分数大于等于 60 的话，那这个学生及格。输出时按字母顺序列出学生姓名并列出相应的成绩，成绩用 P 或 F 来表示。

9-6 为 pass（及格）/ fail（不及格）的学生重新编写那个计算成绩的程序，输出的时候首先出示所有及格的学生，跟着就输出不及格的学生。

9-7 在§4.1.3 中的 read_hw 函数解决了一个很一般的问题（把一系列的数据读到一个向量中）——尽管它的名称表明了，它应该是 Studeng_info 实现的一部分。当然，我们能改变它的名称——但是，让我们假设，尽管从表面上看它是很一般的，但是，为了表明我们不打算让它成为公有访问的，我们还是希望把它和其余的 Student_info 代码结合在一起，那么我们应该如何做到这一点呢？

第 10 章

管理内存和低级数据结构

到目前为止，我们尝试过将数据储存在变量中，或者储存在标准库中提供的容器中，如向量类。我们之所以使用标准库，是因为标准库的工具通常比核心语言提供的工具更灵活，也更易于使用。

如果你懂得如何使用标准库，下一步你将要试着去理解它是如何工作的。要理解标准库，关键是使用核心语言编程工具与技巧，这些方法在后面的章节中还要用到。我们用术语“低级”（low level）来表示这种思想，因为它们比标准库更加底层，而且更接近一般计算机硬件的工作方式。正因如此，这些方法比标准库的方法更难使用，也更危险，不过如果你能充分理解这种方法，那么它们也将是更有效的。因为使用标准库不可能解决所有问题，所以很多的 C++程序经常使用“低级”技巧。

以往我们会在解决一个问题之前提出该问题，本章将与此惯例不同，因为我们即将要使用的工具将会在一个很低层的水平上工作，因此不可能只使用某一个工具本身来解决一个实际的问题。我们打算先提出两个相关的概念：数组和指针。我们将展示如何结合这些方法，使用 new 表达式与 delete 表达式来动态地分配内存，通过该方法程序员们能更直接地控制内存的分配，对比之下，如果使用向量和链表等库函数，程序员将无权控制内存的分配。

一旦理解了数组和指针的工作原理，我们将在第 11 章里揭示，标准库是如何使用这些工具来实现容器的。

10.1 指针与数组

数组是容器的一种，类似于向量但没有向量强大。指针是一种随机存取的迭代器，可以用来存取数组中的元素，此外还有一些其他的用途。指针与数组都是 C 与 C++语言中最原始的数据结构之一。它们事实上是彼此不可分离的，从这个意义上来说，我们如果只使用数组而不使用指针将不可能解决任何有意义的问题，而指针的强大将在使用数组的时候得以体现。

因为这两个概念彼此联系密不可分，我们将在试图分别用它们来解决有意义的问题之前介绍一下这两个概念。我们现在先来讨论一下指针的使用，因为即使没有数组的知识我们也能比较容易地理解指针，而反之则难得多。

10.1.1 指针

一个**指针**是一个存放对象地址的值。每一个单独的对象都有一个惟一的地址，该地址指向计算机内存中存放该对象的位置。如果能访问一个对象，就一定能获得它的地址，反之亦然。举个例子，假设 x 是一个对象，那么&x 就是该对象的地址，而如果 p 是一个对象的地址，那么*p 就是该对象本身。在表达式&x 中，&是一个**求址算符**，注意，&在这里的用法与前文（§4.1.2）中提到的按引用调用类型的用法是不同的。在表达式*p 中*是一个**间接引用算符**，它的工作方式类似于在前文中（§5.2.2）提到的将*用到其他迭代器前的用法。如果指针变量 p 中保存了对象 x 的地址，我们也可以说 p 是一个**指向** x 的**指针**。一般将两者的关系图示如下：

如同其他自带的变量类型一样，一个指针类型的局域变量在被赋值之前没有任何有意义的值。程序员们通常用 0 来初始化指针变量，因为将 0 转化成指针值，可以确保产生一个与指向具体对象的指针不同的值。另外，常量 0 也是惟一可以被用来转化成指针类型的整型值。我们称 0 转化成的指针类型值为空指针，空指针在编程中具有很重要的作用。

像其他的 C++值一样，指针也有不同的类型。一个 T 类型的对象的地址具有“指向 T 的指针”类型，在定义或者类似的代码中记为 T*。

假设 x 是一个整型（int）的变量，定义成

```
int x;
```

下面我们想定义变量 p，使之可以存储整型变量 x 的地址，或者说要将 p 定义成“指向 int 类型的指针”。我们可以定义*p 为 int 类型

```
int *p;          // *p 具有 int 类型
```

这里*p 是一个声明符，是一个简单变量的定义的一部分。尽管*和 p 都是这个简单的声明符的一部分，但是大多数的 C++程序员都习惯将上面的定义写成

```
int* p;          // p 具有 int*类型
```

这样可以起到强调 p 是一个特殊的变量类型（例如，int*）的作用。这两种用法是等价的，因为*边上的空格在编译中会被忽略。但是，后一种用法隐藏着一个极大的缺陷，我们应该对其引起足够的注意：

```
int* p, q;       // 这个定义是什么意思呢?
```

上面的语句将 p 定义成“指向 int 型的指针”，而将 q 定义成一个整型变量。这个例子如果写成下面的形式将会更容易理解：

```
int *p, q;       // *p 和 q 都具有 int 类型
```

或者，我们也可以写成：

```
int (*p), q;     // (*p)和q都具有int类型
```

又或者，我们可以写成下面的形式，这样就绝对不会产生歧义了。

```
int* p;          // *p具有int类型
int q;           // q具有int类型
```

下面，我们可以使用指针写出一个简单的程序：

```
int main()
{
    int x = 5;

    // p指向x
    int* p = &x;
    cout << "x=" << x << endl;

    // 通过p改变x的值
    *p = 6;
    cout << "x=" << x << endl;
    return 0;
}
```

程序的输出结果为

```
x = 5
x = 6
```

一旦我们定义了变量 p 以后，变量 p 和 x 的关系将如下图所示

当程序执行第一个输出语句时，x 的值是 5。接下来的语句通过执行*p = 6 将变量 x 的值改变成 6。请记住，一旦 p 储存了 x 的地址，*p 和 x 将是指向同一对象的两种完全等效的方法，所以当第二个输出语句执行时，x 的值变成了 6。

我们可以将一个对象理解成一个只包含该对象一个元素的“容器”，而将指向该对象的指针理解成为一个指向一个“容器”中惟一的元素的迭代器，这对于后面的理解很重要。

10.1.2 指向函数的指针

在§6.2.2 中，我们研究了一个程序，该程序将一个函数作为另一个函数的参数，在那里曾经说过，该程序执行的背后还有一些与表面上看上去不同的地方。事实上函数不是对象，我们无法对它们进行复制或赋值，也无法将它们直接作为参数。特别是，在程序中无法创

修改一个函数——只有编译器可以这样做。一个程序对一个函数进行的全部操作只有调用该函数，或者得到它的地址。

然而，在§6.2.2中我们将函数median_analysis作为参数传递给函数write_analysis，事实上调用了一个以另一函数为参数的函数。在这一表象的背后，编译器悄悄地将这一调用进行了转化，实际上并没有直接调用函数本身，而是使用了指向函数的指针来进行操作。指向函数的指针与其他类型的指针使用起来也是差不多的。一旦你间接引用一个指针，你对产生结果的函数所要做的就只能是调用它——或者说调用函数的地址。

指向函数的指针的声明符与其他类型的指针声明符是一样的。举个例子来说，我们写出下面的语句：

```
int *p;
```

意思是说*p具有int类型，因而表明p是一个指针，如果写出下面的语句：

```
int (*fp)(int);
```

那么我们间接引用了fp，调用它时以一个int类型的变量作为参数，返回结果也具有int类型。也就是说，fp是指向具有一个int类型参数并返回int类型结果的函数的指针。

因为你对一个函数所能做的所有操作只能是获得它的地址或者调用它，所以在任何地方如果出现一个函数名而且不是在调用该函数时，即使没有显式地使用&声明，编译器都会将它解释成为该函数的地址。类似地，你也可以不用显式地间接引用一个指向函数的指针而直接调用它。举个例子，如果我们有一个与fp类型匹配的函数如下：

```
int next(int n)
{
    return n + 1;
}
```

那么我们可以通过下面两种方法中的任意一种使fp指向next函数：

```
// 下面的两个语句是等价的
fp = &next;
fp = next;
```

类似地，如果有一个int类型的变量i，我们要通过fp来调用next函数以使i加一，可以用下面的两种方法中的任意一种实现：

```
// 下面的两个语句是等价的
i = (*fp)(i);
i = fp(i);
```

最后要提到的是，如果我们写了一个程序，看上去这个程序以另一个程序为参数，编译器会在背后悄悄地将该参数转化成一个指向函数的指针。所以，在§6.2.2的write_analys[illegible]中，我们将参数写成：

```
double analysis(const vector<Student_info>&)
```

也可以等价地写成

```
double (*analysis)(const vector<Student_info>&)
```

但是，这一转变对于函数的返回值却不会自动执行。如果我们想写一个返回类型为指向函数的指针的函数，其返回类型要求与 write_analysis 的类型一样，那么必须显式地声明该函数返回一个指针。一种实现的方法是先使用关键字 typedef 定义 analysis_fp 为一个合适的指针类型的类型名：

```
typedef double (*analysis_fp)(const vector<Student_info>&);
```

然后可以用该类型来声明我们的新函数：

```
// get_analysis_ptr 函数的返回值为一个指向 analysis 函数的指针
analysis_fp get_analysis_ptr();
```

如果用等价的另一种声明

```
double (*get_analysis_ptr())(const vector<Student_info>& );
```

就显得太冗长了。结果是，如果调用 get_analysis_ptr()函数，并且间接引用结果，那么你得到的将是一个返回值为 double 类型，以一个 const vector<Student_info>&类型变量为参数的函数。幸运的是，返回一个指向函数的指针的函数在实际编程中很少见！在本书的其余部分我们都不再使用到该语法，不过在 § A.1 中对该语法有进一步的具体解释。

指向函数的指针经常被用作另一函数的参数。作为一个例子，下面给出在函数库中 find_if 函数的实现代码：

```
template<class In, class Pred>
In find_if(In begin, In end, Pred f)
{
    while (begin != end && !f(*begin))
        ++begin;
    return begin;
}
```

在该例中，在 f(*begin)具有一个有意义的值时，Pred 可以是任何类型。假设现有一个判断函数，定义如下：

```
bool is_negative(int n)
{
    return n < 0;
}
```

而且我们用 find_if 来定位名为 v 的 vector<int>类型容器中第一个负值的元素：

```
vector<int>::iterator i = find_if(v.begin(), v.end(), is_negative);
```

我们可以将&is_negative 写成 is_negative，因为编译器会自动将函数名转化成指向函数的指针。类似地，在 find_if 函数的实现代码中也可以将(*f)(*beg)写成 f(*beg)，因为编译器将调对函数指针的调用自动地解释成调用该指针指向的函数。

10.1.3 数组

数组是容器的一种，但它是核心语言的一部分而不是标准库的内容。每个数组都包含一个或几个同类型的对象，每一个对象称之为一个元素。C++的语法要求数组元素的个数必需在编译的时候确定，这一要求表明，数组不能像标准库中的其他容器一样动态地增加或减小尺寸。

因为数组不是类，所以没有成员函数和成员变量。特别要注意的是，数组没有 size_type 这个成员变量，所以无法利用成员变量命名一个合适的类型来处理数组尺寸的问题。取而代之的是，它在<cstddef>头文件中定义了一个更普遍的类型 size_t。size_t 被定义成无符号类型，它的大小足以装下任何对象。因此，我们可以，当然也必须用 size_t 来表示一个数组的大小，就像在容器中我们使用 size_type 来表示容器的大小一样。

例如，一个三维几何点在程序中可以定义如下：

```
double coords[3];
```

如果你知道物理空间里的维数是不会随时间而变化的，那么就可以对三维几何点写出一种更好的定义：

```
const size_t NDim = 3;
double coords[NDim];
```

在计算机的管理中，NDim 的值是在编译的时候就知道的（因为 const size_t 将 NDim 初始化为一个常量。所以我们使用常量 NDim 来和同样可以表示为三角形边数的常量 3 相区别。

无论我们如何定义一个数组，在数组与指针之间都有一个基本的关系：只要我们将数组名做为一个值使用，那么数组名将表示指向数组首元素的指针，换言之，数组名保存了该数组的首地址。我们已经定义 coords 为一个数组，所以 coords 就给出了数组首元素的地址。像其他类型的指针一样，我们可以用*运算符对 coords 间接引用，以访问该指针指向的对象，执行下面的语句

```
*coords = 1.5;
```

把值 1.5 赋给 coords 数组的首元素。

10.1.4 指针算法

现在我们懂得了如何定义数组，也懂得如何获得数组首元素的地址。那么数组中的其他元素呢？我们该如何访问它们并对它们进行操作？回忆一下在 § 10.1 中我们曾经说过指针

是一种迭代器，或者确切地说，指针是一个随机存储的迭代器。由这一事实我们可以知道数组的第二个基本性质：如果 p 指向数组中的第 m 个元素，那么（p+n）指向第（m+n）个元素，（p-n）指向第（m-n）个元素，当然，前提是必须存在第（m+n）和第（m-n）个元素。

继续我们上面的讨论，按照惯例，coords 数组的首元素其标识数字为 0，或者说首元素是数组的第 0 号元素，那么 coord+1 表示 coords 数组的第 1 号元素（也就是首元素后面的那个元素）的地址，coords+2 表示第 2 号元素（也是 corrds 数组中的最后一个元素）的地址。

那么 coords+3 表示什么呢？这个值表示 coords 数组中的第“3”号元素的地址，可是这所谓的第“3”号元素并不存在。

不过无论如何，coords+3 都是一个有效的指针，尽管它并不指向 coords 数组中的任何一个元素。类似于 vector 和 string 这两个容器，对一个含有 n 个元素的数组的首元素地址加 n 得到一个新地址，该地址不指向数组中的任何对象，但是该地址是有效的。对于与 p 相关联的三个表达式 p、p+n 和 p-n，即使它们中有些可能超出数组的地址范围，但它们都是有效的，只是不可预测而已。

举个例子，为了把 coords 中的内容复制到一个向量中，我们可以编写如下代码：

```
vector<double> v;
copy(coords, coords + NDim, back_inserter(v));
```

在这里，NDim 像前面一样，只是常量 3 的另一种写法而已。在这一例程中，coords + NDim 不指向数组中的任何元素，它不代表容器的最后一个元素，但它是有效的，将该地址作为第二个参数传递给 copy 函数不会产生任何问题。

作为另一个例子，我们将用两个迭代器构造出一个向量类型对象，将 coords 元素的复制构造成一个新的变量 v，程序语句如下：

```
vector<double> v(coords, coords + NDim);
```

换句话说，假设 a 是一个具有 n 个元素的数组，v 是一个向量，而且我们想把标准库的算法应用到 a 数组的元素上。那么，在任何我们使用 v.begin()和 v.end()以使得标准库算法可以访问到 v 中元素的地方，我们都要用 a 和 a+n 来作参数，以便将这些算法应用到 a 的元素上。

如果 p 和 q 都是指向同一数组中的元素的指针，那么 p - q 是一个整数，它表示 p 和 q 所指向的元素在数组中的间距。或者更确切地说，p – q 的定义确保了(p - q) + q 等于 p。因为 p-q 可能是负值，所以它是一个带符号的整数类型。该类型到底是整型(int)值还是长整型(long)取决于系统环境，所以在标准库中提供了一个同义词 ptrdiff_t 来表示相应的类型。像 size_t 一样，ptrdiff_t 类型在<cstddef>头文件中定义。

我们在 § 8.27 中看到，对指向一个容器前面的地址的迭代器进行计算是不允许的。类似地，对数组前面的地址进行计算被视为是非法的。换句话说，如果 a 是一个 n 元素的数组，那么 a + i 只有在 $0 \le i \le n$ 时才是有效的，而且只有在 $0 \le i < n$（不包括 i 等于 n 的情况）时 a + i 才指向数组中的某一个元素。

10.1.5 索引

在§10.1中，我们提到指针是数组的随机访问迭代器。像所有的随机访问迭代器一样，它们（指针）都支持索引的功能。特别值得一提的是，如果指针p指向一个数组中的第m个元素，那么p[n]就代表数组中的第m+n个元素本身——注意不是代表该元素的地址，而是该元素本身。

回忆一下，在§10.13中我们说数组的名字代表该数组首元素的地址。这一事实和上一段中说的关于p[n]的事实说明，如果a是一个数组，那么a[n]就是数组的第n个元素。或者说，如果p是一个指针而n是一个整数，那么p[n]将与*(p+n)等价。

在大多数的程序语言中，索引都是必不可少而且是个很明显的特性。而在C++中，索引并不是数组的一个直接的特性。更确切地说，索引其实包含了数组名与指针这两个特性，以及指针支持为随机访问迭代器定义的操作这个事实。

10.1.6 数组初始化

数组有一个很重要的的特性，这在标准库提供的容器中是没有的，那就是，我们可以很简单地对数组中的每个元素进行初始化。此外，使用这种初始化的语法还可以避免显式地定义该数组的大小。

例如，如果我们要写一段程序对日期进行操作，我们可能希望知道每个月有多少天。一种可行的办法如下：

```
const int month_lengths[] = {
    31,28,31,30,31,30,    // 在程序的其他地方我们相应地处理闰年
    31,31,30,31,30,31
}
```

在这里，我们直接把每个月的天数赋给数组作为初始值，该数组将1月作为第0号月份，12月做为第11号月份。由此我们可以通过month-lengths[i]来获得第i号月份的天数。

注意，在定义数组 month_length 时我们并没有显式地说明该数组有多少个元素。因为我们已经显式地对它进行了初始化，编译器将自动计算出该数组的元素个数——显然计算机比我们更加适合做这项工作。

10.2 再看字符串常量

我们已经掌握了足够的知识，现在我们可以理解字符串常量的本质了：事实上，字符串常量是一个字符常量数组，该数组的大小是字符串的长度加一。这多出来的一个字符是编译器自动在其他的字符后面加上的一个空字符（也就是'\0'）。换句话说，如果我们定义

```
const char hello[]={'H','e','l','l','o','\0'};
```

那么 hello 数组将与字符串常量“Hello”等价。当然，除非这个字符数组与字符串常量是两个不同的对象，因而具有不同的地址（而这是显然的）。

编译器向字符串常量的末尾加入一个空字符，是作为该字符串的终止符，要知道，一个字符串变量只给出该变量首字符的地址，程序在一个字符串中遇到第一个空字符时认为字符串结束。在<cstring>库函数中有一个函数叫作 strlen，可以用来求一个字符串变量或者一个以空字符结束的数组的大小，但要注意，strlen 函数返回的大小没有算上空字符这个终止符。strlen 函数的一种实现方式如下：

```
一个标准库函数实现的例子
size_t strlen(const char* p)
{
    size_t size = 0;
    while ( *p++ != '\0' )
        ++size;
}
```

回忆一下，在§10.1.3 中我们提到 size_t 是一个无符号整型的变量，它足够大，可以容下任何数组。strlen 函数从指针 p 指向的位置开始向后数，直到遇着第一个空字符为止，但不包括这个空字符，由此得到字符串的长度。

因为 hello 变量与字符串常量“Hello”等价，

```
string s(hello);
```

上面的语句就定义了一个 s 变量，该变量储存了字符串“Hello”的一个复件。另外，我们可以将两个迭代器连接成一个字符串，所以可以写出如下的语句

```
string s(hello, hello + strlen(hello));
```

在这里，我们用数组 hello 的名字来作为指向 hello 数组首元素地址的指针，而 hello+strlen(hello)是指向空字符'\0'的指针，这个空字符位于 hello 字符串的'o'字符之后，也是字符串中的一个字符。因为指针是迭代器，所以我们可以用两个指针来构造一个新的字符串，就象我们在§6.1.1 中所做的那样，在那里我们将两个迭代器连接成一个新的字符串。在这两个例子中，第一个迭代器指向我们想要构造的新字符串内容的第一个字符，而第二个迭代器指向最后一个字符内容后面的字符。

10.3 初始化字符串指针数组

在§10.2 我们提到一个字符串常量其实只是一个地址，它方便地表示了一个以空字符为结束标志的字符串的首字符的地址。在§10.1.6 中我们把相应的一序列字符用大括号括起来赋给一个数组，用这种方法来对数组元素进行初始化。从上面提到的两点可以想到，我们可

序列的字符串常量来初始化一个字符指针数组。

上面这个设想我们可以将它转化为一个实例，假设我们要把分数成绩按照下面的规律转化成字母成绩：

如果分数不低于	97	94	90	87	84	80	77	74	70	60	0
那么字母成绩为	A+	A	A-	B+	B	B-	C+	C	C-	D	F

下面是一个转化程序：

```
string letter_grade(double grade)
{
    // 分数成绩的界限
    static const double numbers[] = {
        97,94,90,87,84,80,77,74,70,60,0
    };

    // 字母成绩表示
    static const char* const letters[]= {
        "A+","A","A-","B+","B","B-","C+","C","C ","D","F"
    };

    // 根据数组的大小计算成绩的个数
    // 和单个元素的大小
    static const size_t ngrades = sizeof(numbers) / sizeof(*numbers);

    // 根据分数成绩得到相应的字母成绩
    for (size_t i = 0; i < ngrades; ++i) {
        if (grade >= numbers[i])
            return letters[i];
    }

    return "?\?\?";
}
```

在 numbers 的定义中用到了关键字 static，在前面 § 6.1.3 中我们对 static 的使用进行过介绍。在上面的程序中，static 关键字的使用告诉编译器在使用 letters 与 numbers 数组之前只要进行一次初始化。如果不使用 static 关键字，编译器将在每次调用这两个数组之前都进行一次初始化，显然这会减慢程序的执行速度。另外我们将数组元素定义为常量，因为这些数值不需要在运行期间进行改变，也正因如此，我们才可以只对数组进行一次初始化。

letters 是一个这样的数组，它的每个元素都是指向一个常量字符的指针。在这里，每个指针元素指向与分数成绩相对应的字母分数字符串的首字符。

在 ngrades 的定义中我们引入了一个新的关键字，sizeof，它被用来获得 numbers 数组的元素个数，这样我们就不必自己动手编程去数了。如果 e 是一个表达式，那么 sizeof（e）返回一个 size_t 值，告诉我们 e 类型的对象占用多少内存。这一过程无需对表达式进行过多的

因为不需要进行处理以得到它的数据类型，而且每个给定类型的对象都占用同样大小的内存。

sizeof 运算符返回的数值以字节（bytes）为单位，这是实际的存储单位，随具体编程工具不同而有所差异。关于字节惟一可以肯定的是一个字节包含八个位（bit），每个对象至少占用一个字节，一个字符（char）变量正好占用一个字节的空间。

当然，我们只是想知道在 numbers 数组中有多少个元素，而并不关心它占用了多少字节的空间。为此，我们把整个数组的大小除以每个元素占用的空间大小。回忆一下，在 § 10.1.3 中我们曾经提到，numbers 是一个数组，*numbers 是数组的一个元素。而这个元素正好就是数组的首元素，当然这一点在这里并没有什么重要的意义。但是因为所有元素都是一样大小，所以很重要的一点是 sizeof(*numbers)就是 numbers 数组中任一个元素的大小，因此 sizeof(numbers) / sizeof(*numbers)就是该数组的元素个数。

一旦建立了分数成绩与字母成绩的对应数据表格，接下来判断其字母分数的工作将是十分简单的。我们顺序在 numbers 的元素中查找，直到发现 grade 数据大于或等于其中的一个元素。当我们在 numbers 中找到了相应的元素，我们返回其在 letters 中的相应元素。这个元素只是一个指针，但是正如我们在 § 10.2 中所说的，由该指针我们可以得到其指向的字符串。

如果我们没有找到一个合适的字母成绩，这意味着用户提供给我们的是一个负数表示的分数成绩，这时候程序返回一个无意义的字母成绩。在程序代码中使用反斜杠“\”，这是因为 C++程序中不允许存在连续的两个或多个问号（关于这一点在附录 § A.2.1.4 中将会详细讲解）。所以在这里，我们必须用“?\?\?”来表示“???”。

10.4 main 函数的参数

现在我们已经理解了指针与字符数组的工作原理，下面来讲讲如何向 main 主函数传递参数。如果需要的话，大多数的操作系统都提供一种途径来向 main 函数传递一个或者几个字符串的参数。如果 main 函数需要使用参数的话，那么它就具有两个参数，一个整型（int）参数与一个指向字符指针的指针参数。像其他的参数一样，main 函数的参数可以随便以任何名字命名，不过程序员一般习惯把这两个参数叫做 argc 和 argv。argv 是指向一个指针数组首元素地址的指针，数组中的每个元素都指各一个字符串参数。argc 的值是 argv 指向的数组中的指针个数。argv 数组的首元素总是 main 函数编译后的程序名，所以 argc 的值至少是 1。如果有参数的话，这些参数总是在数组中作为连续的几个元素出现。

下面的例子程序向外输出它所接受到的参数，如果必要的话，在各个参数间会插入空格以分隔开它们：

```
int main(int argc, char** argv)
{
    // 如果有参数，那么将它们显示出来
    if (argc > 1) {
```

```
        int i;     // 在 for 循环外面声明变量 i，因为我们在循环结束之后还要使用它
        for (i = 1; i < argc-1; ++i)    // 输出除了最后一个以外的所有参数
                                        // 参数间用空格隔开
            cout << argv[i] << " ";     // argv[i]是一个字符指针 char*

        cout << argv[i] << endl;    // 输出最后一个参数，参数后面没有空格
    }
    return 0;
}
```

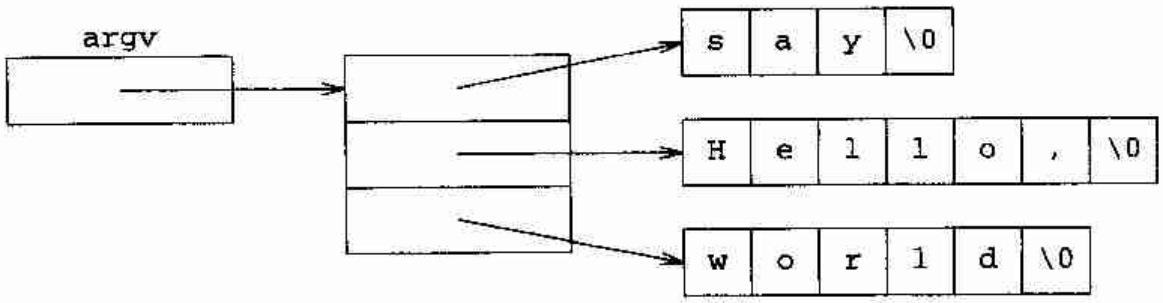
对该程序进行编译，得到一个可执行文件，假设这个文件的文件名是 say，那么如果在操作系统中执行以下命令

```
say Hello, world
```

我们将得到下面的输出结果：

```
Hello, world
```

在这一实例中，argc 等于 3，而 argv 中的三个指针元素将分别指向"say"、"Hello,"与"world"三个字符中的首字符。关于 argv 参数的关系图如下：



10.5 文件读写

本书中的程序实例都是使用 cin 和 cout 来进行输入与输出操作。更多的应用程序要求访问多个文件，进行文件输入或者文件输出的操作。在 C++中提供了多种方式进行文件操作，在这里我们只简单地介绍一下这方面的知识。

10.5.1 标准错误流

在一个程序中，当出现非正常情况时，往往希望能有一种方法将异常输出。这一输出将告诉用户程序出现了错误，或者出现异常后能建立一个事件日志，这一点对于一个好的程序是很重要的。

为了使这种异常输出区别于其他的普通输出，在 C++库里除了标准输入输出流以外，还定义了**标准错误流**。这种错误流常常与标准输出流合为一体，但是在很多的系统中都提供了专门的方法来区分它们。

在 C++程序中，可以用 cerr 或者 clog 来进行标准错误流输出。这两个输出流的最终目的都一样，不过它们在处理缓冲时略有差异（详见 § 1.1）。

正如它的名字一样，clog 流倾向于生成日志。因此，clog 流与 cout 有着一样的缓冲特性：平时储存着错误信息，在系统认为适当的时候将它们输出。而 cerr 流则是即时输出错误信息，这可以保证只要一发生异常，就及时显示错误信息。因此，如果你要及时反映错误的话，请选择使用 cerr；如果只是为了生成一个异常信息日志，就应该选择使用 clog。

10.5.2 处理多个输入输出文件

标准输入输出流与错误流有可能与文件关联，也可能不与文件关联。例如，一个视窗系统在运行一个 C++程序时可能会把与该程序相连的流和该程序关联起来，可能使用另外一种截然不同的方式来这样做，例如直接访问磁盘文件。

因此，C++标准库里用于文件输入输出的对象和用于流的输入输出对象具有不一样的类型。如果你想处理一个输入输出文件，那就必须分别创建一个 ifstream 和 ofstream 类型的对象。这一要求看上去造成了毫无必要的麻烦。毕竟，我们看到在库函数里输入输出都是按照 istream 与 ostream 的类型定义的。那么在标准库里还有没有对 ifstream 与 ofstream 进行其他的定义呢？

幸运的是答案是没有。在第 13 章里我们将会看到，有可能存在一个类型，它与另一个类型十分相似，甚至可以取代它。在标准库里，ifstream 被准确地定义为 istream 的一种，而 ofstream 是 ostream 的一种。因此，我们可以在任何需要 istream 的地方使用 ifstream，在任何需要 ostream 的地方使用 ofstream。这些类的定义可以在头文件<fstream>中找到。

在定义一个 ifstream 类型对象或者一个 ofstream 类型对象的时候，可能会要求提供一个 string 类型形式的文件名。事实上，C++要求提供的不是一个 string 类型对象，而是一个指向以空字符结尾的字符数组的首元素的指针。产生这一要求是有其充分的理由的。第一个理由是因为这种方式可以使程序用到输入——输出库的特性而不依赖于 string 类的特性。第二个理由是一个历史的原因：输入——输出库的出现比 string 类要早好几年。还有第三个理由是，以这种方式提供文件名的时候可以使程序更易于与操作系统的输入——输出函数之间建立接口，一般来说它们都是通过指针来通信的。不管是出于什么原因，总之程序在处理文件的时候一般都要求提供一个指向空字符结尾的字符数组的指针作为文件名参数。

下面这个例子将一个文件名为 in 的文件复制到一个文件名为 out 的文件中去：

```
int main()
{
    ifstream infile("in");
    ofstream outfile("out");
```

```
    string s;

    while (getline(infile, s))
        outfile << s << endl;
    return 0;
}
```

上面的程序正是基于字符串常量是一个指向空字符结尾的数组的首字符的指针这一事实。如果不想把文件名定义成一个字符串常量，那最好的代替办法是把文件名储存在 string 类型的变量里，然后使用 c_str 成员函数（该函数我们将在 § 12.6 讲到）。举个例子，如果 file 是一个 string 型变量，它包含了我们要读的文件的文件名，那么我们可以创建一个 ifstream 对象来读文件，该对象定义如下：

```
ifstream infile(file.c_str());
```

作为最后一个例子，这里给出一个程序，该程序可以复制一个或几个文件，要复制的文件的文件名在 main 的参数里给出。

```
int main(int argc,char **argv)
{
    int fail_count = 0;
    // 对于每个要输出的文件
    for (int i = 1; i < argc; ++i) {
        ifstream in(argv[i]);
        // 如果文件存在，则复制文件内容，否则生成一个错误信息
        if (in) {
            string s;
            while (getline(in, s))
                cout << s << endl;
        } else {
            cerr << "cannot open file" << argv[i] << endl;
            ++fail_count;
        }
    }
    return fail_count;
}
```

对每一个在 main 函数的参数（相关的内容见 § 10.4），程序生成一个 ifstream 对象来读该文件的内容。当对象值为 false 时（也就是说该文件不存在，或者是不能打开等其他原因），程序用 cerr 将报告错误，然后用一个变量记录总共出现几次错误。如果程序成功地创建了 ifstream 对象，那么它将读取该文件内容进入 s，每次只读一行内容，然后将内容逐行写到标准的输出设备中。

当程序将控制权交还给系统时，同时返回读操作失败的文件个数。一般来说，返回

表示成功返回，在这里表示我们可以成功读取所有在 main 函数参数中列出的文件。

10.6 三种内存分配方法

至今为止，我们已经见到过两种不同的内存分配方法，不过我们还没有单独地对它们进行仔细的讲解。第一种方法是自动分配内存，这种方法常与局域变量联系在一起：一个局域变量只在程序执行到该变量定义的时候才由系统自动分配内存给它，当包含该变量的定义的模块结束时，该变量占用的内存自动释放。

当一个变量所占有的空间被释放后，任何指向它的指针都将变得无效。程序员们应当注意避免使用这种无效的指针。例如：

```
// 该程序故意产生一个无效指针
// 这只是一个反面例子--您可千万别这么做哦!
int* invalid_pointer()
{
    int x;
    return &x;              // 紧急灾难!
}
```

该函数返回一个局域变量 x 的地址，但不幸的是，在函数返回的时候，定义了局部变量 x 的语句块也同时被终止，x 所占的内存也被释放。&x 创建的指针现在是无效的，但是函数仍执意要返回这个指针，那么该函数将返回一个不可预料的值。特别是，C++执行的时候不会发现这个错误——你将得到一个错误的返回值。

如果真要想返回变量 x 的地址，你可以试试使用另一种内存分配方法，把 x 声明为**静态分配**内存的变量：

```
// 该函数是合法的
int* pointer_to_static()
{
    static int x;
    return &x;
}
```

通过把 x 声明为静态(static)变量就可以解决刚才的问题了。我们只要在指向该静态变量的指针被使用之前声明变量 x，那么系统将对它进行一次而且只进行一次内存分配，之后直到程序结束之前该变量占用的内存都不会被释放。返回一个静态变量的地址就不会再有错误了；只要程序正在运行该指针就是有效的，当然如果程序结束后又另当别论了。

然而，静态分配内存存在潜在的危机，因为每次对一个指向一个静态变量的指针的调用都返回指向同一个对象的指针！假设我们要定义一个函数，在我们每次调用该函数时，都返回一个指向特定的新的对象的指针，该对象一直存在直到我们决定不要需要它。为了达到这个

我们可以使用**动态分配**内存的方法，在这个方法中我们要用到 new 和 delete 这两个关键字。

10.6.1 为一个对象分配/释放内存

如果 T 是一个对象的类型，那么 new T 表达式将为一个 T 类型的对象分配内存，该变量由构造函数对其初始化，并且产生一个指向该新分配内存的对象（该对象甚至没有命名）的指针。执行象 new T(args)这样的初始化语句可以给变量赋予一个特定的值。该对象一直存在直至程序结束或者执行了 delete p 语句，其中 p 是在 new 语句中返回的一个指针。为了用 delete 删除一个指针，这个指针必须是指向一个用 new 语句分配内存的对象，或者是一个零指针。删除一个零指针不进行任何操作。

下面是一个例子：

```
int* p = new int(42);
```

该语句为一个无名的 int 类型的对象分配内存，将该对象初始化为 42，并使 p 指针指向该对象。我们可以在下面的语句中改变该对象的值：

```
++ *p;        //p 现在等于 43
```

执行该语句后 p 将变成 43。在我们用完该对象后，我们可以执行

```
delete p;
```

该语句将*p 对象占用的内存空间释放出来，p 将是一个无效的指针，在我们将一个新的值赋给它之前 p 具有不可预知的值。

做为另一个例子，我们可以写一个函数，为一个 int 类型的对象动态分配内存，对它进行初始化并返回一个指向该对象的指针：

```
int* pointer_to_dynamic()
{
    return new int(0);
}
```

调用这个函数来声明一个对象，我们就可以在任何一个适当的时候释放该对象。

10.6.2 为一个数组分配/释放内存

如果 T 是一个类型名，而 n 是一个非负整数，那么 new T[n]语句将为一个 n 个 T 类型对象的数组分配内存，并且返回一个指向数组首元素的指针（该指针类型为 T*）。每个对象都将被默认初始化，也就是说如果 T 是内建类型而且数组又只是在局部作用域内分配内存，那么对象将不会被初始化。如果 T 是一个类，那么数组中的每个元素都会运行类的缺省构造函数进行初始化。

如果 T 是一个自定义类型，那么在初始化的过程中有两点要注意的：第一点是，如果该类不允许默认初始化，那么编译器将终止程序；第二点是，数组中 n 个元素的每一个元

被初始化，这将带来一定的运行时开销。在第 11 章中我们将会学习到另一种为数组动态分配内存的更加灵活的机制，该方法由标准库提供。这种新的方法在为数组动态分配内存的时候要比使用 new 更好。

一般来说，一个数组要求至少有一个元素，但是当使用 new T[n]来为一个数组分配内存时，如果 n 值为零，那么在数组中将不包含任何元素。发生这种情况时，new 函数无法返回一个指向首元素的指针——因为数组根本就没有元素。事实上 new 函数此时会返回一个有效但无意义的 off-the-end 指针，我们可以把它作为 delete[]的参数使用，我们还可以把它想像成为一个指向（如果存在的话）首元素的指针。

上面这个奇怪的解释实际上是为了允许类似下面这样的程序在 n 等于零时仍能执行。

```
T* p = new T[n];
vector<T> v(p, p + n);
delete[] p;
```

尽管当 n 等于零时 p 不指向任何元素，但这一点对我们来说没所谓，重要的是 p 和 p + n 像其他的指针一样仍是合法的指针。一般而言，向量将有 n 个元素。即使 n 等于零时这类程序仍然能正常运行，这对于编程者们来说是一个很大的方便。

请注意在这个例子中 delete[]的使用，中括号在这里是必不可少的，它告诉系统释放整个数组占用的内存，而不仅仅是释放其中一个元素的内存。一个数组一旦用 new[]分配了内存，那么该内存将一直被使用直到程序终止或者在程序中执行了 delete[] p 语句（其中 p 是 new[]语句返回的指针的一个复件）。在释放数组之前，系统根据相反的顺序逐个释放数组中的每个元素。

作为一个例子，这儿有一个函数，该函数使用一个指向一个空字符结尾的字符数组（例如是一个字符串文字）的指针，将数组中的每一个元素（包括结尾的空字符）复制到一个新分配的数组中，然后返回指向这个新数组首元素的指针。

```
char* duplicate_chars(const char* p)
{
    // 分配足够的内存空间，记住要为空字符预留空间
    size_t length = strlen(p) + 1;
    char* result = new char[length];
    // 复制进新分配的内存空间并返回指向首元素的指针
    copy(p, p + length, result);
    return result;
}
```

回忆一下，在§10.2 中我们说过，strlen 函数返回一个空字符结尾的数组中的字符个数，但不包括数组的最后的那个空字符元素。正因如此，我们将 strlen 返回的值加 1 作为新分配内存空间的大小，然后再动态分配内存。因为指针是迭代器，我们可以用 copy 来把 p 指向的数组中的字符复制到 result 指向的新分配的内存空间中去。因为 length 表示的数组长度并

组结尾的空字符，所以调用 copy 函数可以以预期的方式工作。

10.7 小结

指针是用来存贮对象地址的随机访问迭代器。例如

p = &s　　使 p 指向 s。

*p = s2　　对 p 间接引用并把一个新的值赋给 p 指向的对象。

vector<string> (*sp)(const string&) = split;

　　把 sp 定义成一个指向 split 函数的函数指针。

int nums[100];

　　把 nums 定义成一个具有 100 个整型元素的数组。

int* bn = nums;

　　把 bn 定义成一个指向 nums 数组首元素的指针。

int* en = nums + 100;

　　把 en 定义成指向 nums 紧接着数组最后一个元素后面的指针。

指针可以指向一个简单对象，也可以指向由对象作为元素的数组，或者指向一个函数。当一个指针指向一个函数时，它的值只能用来调用函数。

数组是 C++自带的具有固定大小的容器，它的迭代器是指针。C++把数组名的使用自动解释为指向数组首元素的指针。一个字符串常量是一个空字符结尾的字符数组。对数组索引的定义是根据指针的运算得来的：对于任意一个数组 a 和它的一个索引 n，都有 a[n]与*(a + n)等价。如果 a 是一个具有 n 个元素的数组，那么区间[a,a+n)将代表了 a 中的所有元素。

一个数组可以在被定义的时候进行初始化：

```
string days[]={'Mon", "Tues", "Wed", "Thu", "Fri", "Sat', "Sun"};
```

该语句执行时由编译器根据初始化的元素个数自动算出数组 days 的大小。

主函数（main 函数）可以带两个可选的参数。第一个参数是一个整型（int）数，该数值表明第二个参数中储存了多少个字符数组，第二个参数的类型是 char**，有时候也可定义成下面的形式：

```
char* argv[];
```

上式与 char**是等价的，不过这种语法只在参数列表中出现时才是合法的。

输入/输出：

cerr　　标准错误流。输出不进入缓冲。

clog　　标准错误流，主要用于保存作日志。输出进入缓冲。

ifstream(cp)　　输出流，文件名必须定义成 char* cp。支持 istream 操作。

ofstream(cp)　　输出流，文件名必须定义成 char* cp。支持 ostream 操作。

输入/输出文件流在头文件<ifstream>中被定义。

内存分配：

new T	为一T类型的对象分配内存，对其进行默认初始化，返回一个指向该新建对象的指针。
new T(args)	为一个T类型的对象分配内存，用args参数对其进行初始化，返回一个指向该新建对象的指针。
delete p	删除指针p指向的对象并释放该对象（*p）占用的内存空间。要求该指针指向一个动态分配内存的对象。
new T[n]	为一个具有n个元素的数组分配内存空间并对其进行默认初始化，该数组的元素具有T类型。返回一个指向数组首元素的指针。
delete[] p	删除指针p指向的数组中的对象并释放数组占用的内存空间。指针p必须指向一个动态分配内存的数组的首元素。

习题

10-0 编译、运行并测试本章的例程。

10-1 重写§9.6的学生成绩程序，要求生成字母成绩。

10-2 重写§8.1.1的median函数，使之可以通过向量或者C++自带的数组来调用该函数。该函数要求可以调用容纳有任何算术类型的容器。

10-3 写一个测试程序验证刚才写的median函数。确保调用median函数时不会改变容器内的元素的先后顺序。

10-4 写一个类使之成为一个可以存储strings的链表。

10-5 为上面的String_list类写一个双向迭代器。

10-6 为了检验上面的类，重写split函数把结果输入到String_list类中。

第 11 章

定义抽象数据类型

在第 9 章里，我们学习了定义新类型所要掌握的基本语法。但是在那里写的 Student_info 类型尚有很多不足之处，因为我们无法知道在对 Student_info 对象进行复制、赋值以及销毁该对象时应该做什么。在本章中我们可以看到，（除去第 9 章讲过的成员函数的功能之外）类的作者同样也可以对对象的这些行为进行控制。本章中令我们感到惊讶的是，用于正确地构造一个类型的操作的本质，其实就来源于我们的直觉和使用习惯。

因为我们在前面经常用到向量类，所以在这里我们要创建一个类似于向量的类来加深我们对如何设计和实现一个类的理解。为了让读者只关注于它的功能以及效率，我们所提供的实现被尽量地简化了。因为这是仿照标准库中的向量类写的版本，为了避免与库函数发生冲突，我们把要写的类叫做 Vec。虽然我们想把主要的注意力放在如何对 Vec 类进行复制、赋值和删除的操作上，但是如果先写一些更简单的成员函数作为开始将有利于我们的理解。等到我们把这些简单的函数写出来以后，再回来看看如何控制进行复制、赋值和删除操作。

11.1 Vec 类

在开始设计一个类的时候，一般来说要先确定在类中要提供什么样的接口。正确地确定接口的一种途径是研究一下类的使用者们将会用我们写的类来编写什么样的程序。既然我们在 Vec 类中想要实现的是与标准库中的向量类相同的功能，下面就让我们从研究过去使用向量类的一个例子开始：

```
// 构造一个 vector
vector<Student_info> vs;          // 一个空的 vector
vector<double> v(100);            // 一个有 100 个元素的 vector

// 获得 vector 使用的类型的名字
vector<Student_info>::const_iterator b, e;
vector<Student_info>::size_type i = 0;

// 用 size 函数与索引值查看 vector 中的元素
```

```
for (i = 0; i != vs.size(); ++i)
    cout << vs[i].name();

// 返回指向第一个元素的迭代器与指向最后一个元素后面那个元素的迭代器
b = vs.begin(); e = vs.end();
```

当然，上面使用的操作只是利用到了标准类 vector 提供的一小部分功能而已，但是通过实现这一小部分的功能，我们可以理解用于实现向量（vector）的其他接口函数的语法技巧。

11.2 实现 Vec 类

一旦确定了要在 Vec 类中实现的操作后，我们下一步将考虑的就是如何写出 Vec 类来。

我们决定先写一个**模板**类，这是一个很好的决定。我们希望让用户用 Vec 类来贮存几种不同类型的数据成员。在 § 8.1.1 中讲到的用于函数的模板技巧同样适用于类，在那里我们写了一个模板函数，并用那个模板生成用在不同的类型上的几种版本。类似地，我们也可以定义一个模板类，然后根据提供的参数用这个模板类来生成一系列不同的类。我们以前曾经使用过这种类型，如 vector、list 和 map 等。

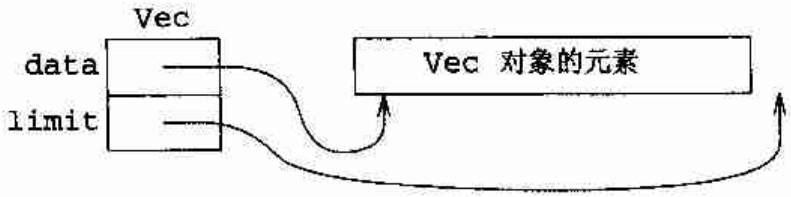
就像定义模板函数那样，我们来定义一个模板类，要注意，这个类是一个模板，它列出了在类定义中将要用到的所有类型参数：

```
template <class T> class Vec{
public:
    // 接口
private:
    // 实现
};
```

上面的代码声明 Vec 是一个具有一个类型参数 T 的模板类。像其他类的定义一样，模板类的定义可分成公共（public）与私有（private）两部分，用于分别代表它的接口和实现部分。

下面要考虑的问题是，我们要储存些什么数据？我们可能需要些空间来保存 Vec 对象中的元素，所以我们必须知道在 Vec 类型对象中包含了多少元素。显然，一个好的建议是用动态建立的数组来容纳 Vec 中的元素。

那么我们将在 Vec 中保存有关这个数组的哪些信息呢？因为在这个类中要实现 begin、end 和 size 等函数的功能，所以 Vec 中要保存首元素地址，末元素后面的地址以及元素的个数等。不过这三个数据不是必须同时保存的，因为我们可以从其中任意两个数据推算出第三个来。因此，我们决定只保存数组的首元素及末元素的地址指针，然后计算出数组的大小来。这个类的数据结构大致如下图所示：



至此，我们可以把 Vec 类进一步改写如下：

```
template <class T> class Vec{
public:
    // 接口
private:
    T* data;     // Vec 中的首元素
    T* limit;    // Vec 中的末元素
};
```

这个类定义声明 Vec 是一个模板类型，它只带一个类型参数。在类定义的主体中，我们调用了类型 T。在使用 T 类型的时候，编译器会用用户在生成一个 Vec 的时候提供的参数来代替 T。例如当我们写

```
Vec<int> v;
```

的时候，该定义会告诉编译器实例化一个 Vec 类的具体版本，在该版本里，所有的 T 都替换成 int。编译器生成的代码把包含 T 的表达式当成是使用 int 的语句。因此，既然在 data 与 limit 的声明中我们用了类型参数 T，它们的实际类型将决定于 Vec 实际封装的对象的类型。

该类型只有在一个 Vec 的定义实例化时才被确定下来。一旦我们决定我们想要一个 Vec<int>，那么 data 与 limit 的类型就定下来了：它们在 Vec 的这一实例中是 int*类型的。类似地，如果我们使用 Vec<string>，编译器将生成 Vec 的另一个不同的实例，这时候 T 将被替换成 string，相应地，data 与 limit 的数据类型变成 string*。

11.2.1 内存分配

因为我们要为数组动态分配内存（§10.6.2），所以我们可能希望用 new T[n]为 Vec 分配空间，其中 n 是我们要为其分配内存的对象数目。但是要记住，new T[n]不仅分配内存空间，还会运行 T 的构造函数来为元素进行默认初始化。如果我们打算使用 new T[n]，那么必须满足一个要求：只有在 T 具有默认的构造函数时才能创建一个 Vec<T>。而标准的向量类没有这样的限制。因为我们要写的类是在模仿标准库中的向量类，所以我们希望在 Vec 类中也没有这个限制。

库函数中提供了一个内存分配类，在内存分配的方面提供了更为详尽的操作。如果用这个类可以替代 new 和 delete，那它就可以充分满足我们的要求。通过这个类我们可以申请分配到未被初始化的内存空间（raw memory），然后（通过一个特殊的步骤）在那片内存中生成对象。

我们假定最后我们将不得不写一些实用函数来为我们管理内存，来避免直接研究那个类的细节。现在假定这些函数已经存在，我们将用它们来完成 Vec 类。通过使用这些函数，可以让我们对它们的功能有更好的了解，那么在实现它们的时候，我们就清楚自己要实现什么。

这些新的实用成员函数是我们类中 private 实现部分的一部分。它们专门负责在需要的时候进行内存分配或者释放内存，以及初始化和删除保存在 Vec 中的元素。因此，这些函数将管理指针 data 与 limit。只有这些内存管理函数可以为这些成员数据赋予新值。Vec 类中的 public 成员函数只用来读取 data 与 limit，但是不能改变它们的值。

当一个公有的成员函数需要做些什么时（如构造一个新的 Vec 对象——这将需要改变 data 和 limit 的值），Vec 内部的实现就会挑选一个合适的内存管理函数来做到这点。这种策略导致我们可以将工作分为如下两个部分：一部分成员将给用户提供接口，另一部分将处理实现细节。

在 11.5 节第 203 页我们会回过头来讨论如何具体实现这些工具函数（utility function）。

11.2.2 构造函数

我们知道，在 Vec 类中我们需要定义两个构造函数：

```
Vec<student_info> vs;   // 默认的构造函数
Vec<double> vs(100);    // 带一个大小参数的构造函数
```

标准的向量类还提供一个相关的第三种构造函数，除了一个大小参数，还带有一个初始值的参数，用来把向量中的元素全部初始化为该值的一个复件。这个构造函数与只带一个大小参数的构造函数十分类似，下面我们也会编写代码实现这第三个构造函数。

所有构造函数具有一个共同的目标，就是确保对象被正确地初始化。对于 Vec 中的对象，我们要对 data 与 limit 进行初始化。这包括给 Vec 中的元素分配内存空间并给它们置一个初始值。在默认构造函数中，我们想创建一个空的 Vec，所以不需要分配任何空间。对于带一个大小参数的构造函数，就要为该参数给定数目的对象分配内存空间。如果用户在给出大小参数的同时还提供了初始值，就可以用这个初始值对分配了空间的元素设置初始值。如果用户仅给出一个大小参数，那么我们将调用 T 的默认构造函数来设置元素的初始值。现在，我们要继续前进，把内存管理函数，也就是对 data 与 limit 进行初始化以及相关的内存分配、设置初值等操作函数完成。

```
template<class T> class Vec {
public:
    Vec() { create(); }
    cxplicit Vec(size_type n, const T& val=T()) { create(n,val);}
    //  其他保留接口
private:
    T* data;
    T* limit;
};
```

默认构造函数不带任何参数，它首先清空 Vec 类的对象（也就是说没有任何元素），这是通过调用 create 成员函数来实现的，create 函数也是我们必须要编写代码实现的函数之一。调用 create 函数返回后，data 与 limit 的值都被设置成零。

在第二个构造函数中我们用到一个新的关键字 explicit，下面对它进行一个简要的说明。首先，我们要知道这个构造函数要做些什么工作。注意到它使用一个默认的变量（§7.3）作为第二个参数。因此这个构造函数事实上定义了两个构造函数，一个使用 size_t 类型的变量作参数，另一个则用到两个参数——一个 size_t 类型与一个 const T&类型的变量。在两种情况里我们都调用带一个大小和一个值共两个参数的 create 函数。create 函数留在 § 11.5 再写，现在假定它可以为 n 个 T 类型的对象分配内存空间，并用 val 来设置它们的初始值。我们的设计是允许可以由用户显式地提供这个初始值，也可以用 T 的默认构造函数来生成这个初始值，相关的内容在 § 9.5 讨论值初始化时已经详细讲解过了。

现在让我们对 explicit 的使用作进一步了解。这个关键字只在定义带一个参数的构造函数的时候才有意义。如果声明一个构造函数是 explicit 的，那么编译器只有在用户显式地使用构造函数时才会调用它，否则就不调用。

```
Vec<int> vi(100);   // 正确，显式地调用 Vec 的构造函数，以一个 int 类型数据作参数
Vec<int> vi = 100;  // 错误：隐式地调用 Vec 的构造函数（§11.3.3）
                    // 并把它复制到 vi
```

在某些上下文中，构造函数可能会被隐式地调用，这时候 explicit 的使用显得至关重要，我们将在 § 12.2 进一步讨论这个关键字。虽然在本章下面的例子中并不要求如此，但是为了保证与标准的向量类一致，我们在这里还是声明这个构造函数是 explicit 的。

11.2.3 类型定义

按照标准类模板的惯例，我们要提供用户能使用的类型名称，这可以隐藏实现该类的具体细节。特别是，我们必须为常量（const）与非常量（nonconst）迭代器类型以及我们用来表示 Vec 的大小的类型提供类型定义（typedef）。

在标准的库容器里还定义了另一个名为 value_type 的类型，这是容器中存储的对象的类型的另一个同义词。让我们继续我们的工作，现在要往 Vec 类中加入 push_back 函数，利用该函数，用户可以动态地增加他们的 Vec 对象。如果我们也定义了 value_type，那么用户就可以用 back_inserter（这由 push_back 与 value_type 共同决定）来生成一个使 Vec 大小增加的输出迭代器了。

定义这些类型的惟一难点在于选择我们要定义些什么类型。我们知道，迭代器是这样一种对象，它用来定位容器中的不同对象，并提供一种检验对象值的途径。通常迭代器本身也是自定义的类型。例如，我们来看看一个生成一个连续列表的类。在逻辑上我们会把它理解成一系列的节点，每个节点包含该节点的值及一个指向列表中下一节点的指针。这种类的迭代器包含着指向其中一个节点的指针，并且可以用++运算符来获得列表中的下一个节点。这

代器必须是自定义的类型。

因为我们用一个数组来封装 Vec 的元素，所以可以使用普通指针作为 Vec 迭代器类型，所有的指针都指向内部的 data 数组。在 § 10.1 我们学过，指针支持所有的随机存取迭代器操作。通过使用一个指针作为内部的迭代器类型，就可以提供完全的随机存取操作，这与标准的向量类是一致的。

那么其他的类型怎么样呢？value_type 的类型显然必须是 T。那么表示 Vec 的大小的类型是什么呢？在前面我们说过，size_t 变量足够大，它可以装下任何一个数组能容纳的元素个数。既然用数组来存储 Vec 的元素，我们可以用 size_t 作为 Vec::size_type 的基础类型。现在，Vec 类可以写成如下形式：

```
template <class T> class Vec {
public:
    typedef T* iterator;                    // 新加部分
    typedef const T* const_iterator;        // 新加部分
    typedef size_t size_type;               // 新加部分
    typedef T value_type;                   // 新加部分
    Vec(){ create(); }
    explicit Vec(size_type n, const T& val=T()) { create(n,val); }
    // 其他保留接口
private:
    iterator data;                          // 更改部分
    iterator limit;                         // 更改部分
};
```

除了加入合适的 typedef，我们还在类中使用了新的类型。在类中使用同样的名字（在 typedef 声明中被定义）可以使程序代码更具可读性，另外，如果在今后改变了某个地方的类型，其他地方也会自动地发生同步变化。

11.2.4 索引与大小

用户可以用 size 函数获知一个 Vec 类型对象里有多少个元素，使用索引操作可以访问 Vec 中的任何一个元素。例如，

```
for (i = 0; i != vs.size(); ++i)
    cout << vs[i].name();
```

从上面的使用可以知道，size 函数必须是 Vec 类的一个成员函数，另外我们还要定义对 Vec 类使用“[]”运算符的含义。size 函数十分简单：它不带任何参数，返回一个 Vec::size_type 类型的数，表示 Vec 中的元素个数。在定义索引运算符之前，我们来大致了解一下重载运算符是如何工作的。

定义重载运算符函数与定义任何一个其他的函数基本上是一样的：必须要有一个[illegible]带几个参数，并指定其返回类型。

重载一个运算符时，要把运算符放在关键字 operator 后面。也就是说，在这里要写成 operator []的形式。

运算符的种类（是一个一元运算符还是一个二元运算符）一定程度上决定了该函数要带多少个参数。如果运算符是一个函数而不是一个成员，那么函数的参数个数与运算符的操作数一样多。第一个参数一定是左操作数；第二个参数一定是右操作数。如果运算符被定义成一个成员函数，那么它的左操作数必须是调用该运算符的对象。可见，成员运算符函数比简单的运算符函数要少带一个参数。

一般来说，运算符函数既可以是成员函数，也可以是非成员函数。但是索引运算符必须是成员函数。当用户写出 vs[i]这样的表达式时，将调用 vs 对象的[]运算符函数，并传递 i 作为参数。

我们知道，操作数是一个整型值，它必须足够大以用来表示任何大小的 Vec 类型对象的最后一个元素，这个类型就是 Vec::size_type。剩下来的就是决定索引运算符要返回的类型种类。

只要稍微动脑筋想想，就可以下结论，我们要返回一个指向 Vec 储存的元素的引用。因为这样可以使用户直接进行写操作，就象直接进行读操作一样方便。尽管在我们的例程中只需要用索引来从 vs 中读取数据，但是如果能给用户提供直写功能显然更好。现在，让我们把 Vec 类改写如下：

```
template <class T> class Vec {
public:
    typedef T* iterator;
    typedef const T* cosnt_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec() { create(); }
    explicit Vec(size_type n, const T& val=T()) {create(n,val);}
    // 新增操作：大小与索引
    size_type size() const {return limit - data;}
    T& operator[](size_type i) {return data[i];}
    const T& operator[](size_type i) const {return data[i];}
private:
    iterator data;
    iterator limit;
};
```

size 函数把数组的两个边界指针相减，得到 Vec 类型对象中的元素个数。在 § 10.1.4 曾经讲过两个指针相减生成一个 ptrdiff_t 类型的值，表示两个指针的距离（以元素为单位）。因为 Vec 类的 size 函数不能改变 Vec 类型对象本身，所以在此把 size 声明为 const 成员函数。

索引运算符在数组中定位到正确的元素位置并返回该元素的一个引用。通过这个返回的引用我们可以修改 Vec 类型对象中所储存的数据。我们可以直接对元素进行写操作，这意味着我们需要写出 Vec 类的两个版本：一个是用于常量（const）Vec 对象，另一个是用于

（nonconst）Vec 对象。注意，const 版本返回一个指向 const 的引用，这样做是为了确保用户只能用索引来读 Vec 的数据，而不能改变它的值。我们返回一个引用而不是返回一个值，如果这仅仅是为了与标准的向量类保持一致性的话那就没有任何意义了。这样做的真正原因其实是为了避免当容器中的对象很大时对它进行复制，那样做既浪费空间又影响运行速度。

我们还可以对索引运算符进行重载，这一点看起来令人惊讶，因为看上去它们的参数列表是完全一样的；它们的参数都是一个 size_type 类型的变量。但是，类中的每一个成员函数，包括这些运算符函数，都必须带一个隐式的参数做为作用对象。因为操作的对象可能是常量，也可能不是常量，所以我们可以对运算符进行重载。

11.2.5 返回迭代器的操作

下面来讨论一下返回迭代器类型的函数。我们要实现 begin 与 end 操作，分别返回一个定位在 Vec 首元素及末元素后一个元素的迭代器。

```
template <class T> class Vec{
public:
    typedef T* iterator;
    typedef const T* cosnt_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec() { create(); }
    explicit Vec(size_type n, const T& val=T()) {create(n,val);}
    T& operator[](size_type i) {return data[i];}
    const T& operator[](size_type i) const {return data[i];}
    size_type size() const {return limit - data;}
    // 新增返回迭代器的函数
    iterator begin() { return data; }                // 新增部分
    const_iterator begin() const { return data; }    // 新增部分
    iterator end() { return limit; }                 // 新增部分
    const_iterator end() const { return limit; }     // 新增部分
private:
    iterator data;
    iterator limit;
};
```

我们在此提供了 begin 与 end 函数的两个重载版本，根据 Vec 是否是常量分别调用不同的版本。在常量版本中返回一个 const_iterator 类型量，在该版本中用户只能通过返回的迭代器读取数据而不能改变它们。

至此，Vec 类的编写还只是处于初级阶段，不过基本要素已经齐全了。实际上，如果再往该类中加入 push_back 和 clear 等操作的话，我们就可以在本书的所有例程中用 Vec 类代替标准的向量类。但是不得不承认，我们的 Vec 类在一些关键的重要方法上与向量类还有一定差距。

11.3 复制控制

在本章的介绍中曾经说过，类的作者可以控制在对象被创建、复制以及被销毁时程序的行为。我们已经介绍了如何创建对象，但还没有谈到当对象被复制、赋值以及销毁时，会有什么样的情况发生。我们将会得知，如果我们忘记了定义这些操作，那么编译器将会在必要时为我们合成它们。有时候这些被合成出的操作正是我们所期望的。其他情况下，这些默认操作可能会导致莫名其妙的结果，甚至产生实时错误。

C++是惟一可以允许编程者对对象进行如此层次的控制并被广泛使用的语言。毫无疑问，正确使用这些操作对于构建有用的数据类型至关重要。

11.3.1 复制构造函数

下面这个例子把一个对象的值作为参数传递给函数，或者从函数中通过传值返回一个对象，这就是在对对象进行隐式的复制操作：

```
vector<int> vi;
double d;
d = median(vi);                          // 把 vi 作为参数传递给 median 函数

string line;
vector<string> words = split(line);      // 把 split 函数的返回值赋给 words
```

类似地，我们也可以通过使用一个对象来初始化另外一个对象，从而显式地复制对象：

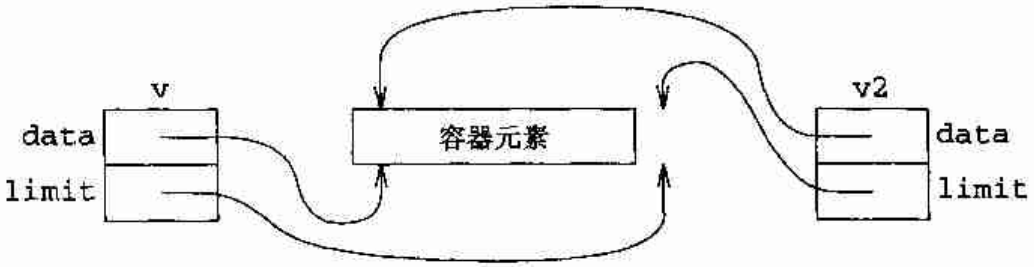
```
vector<Student_info> vs;
vector<Student_info> v2 = vs;        // 将 vs 复制到 v2
```

无论是显式的还是隐式的复制，都由一个名为 **copy constructor**（复制构造函数）的特殊的构造函数进行。像其他的构造函数一样，复制构造函数也是与类名同名的一个成员函数。因为它是用来复制一个已存在的同类型对象，以此来初始化一个新的对象，所以复制构造函数只带一个参数，该参数与类本身具有相同的类型。因为我们定义了复制所带来的效果，包括产生函数参数的复件，所以当参数是引用类型时就有了问题。此外，由于复制对象不会改变原对象的值，所以复制构造函数的参数使用了一个常量引用类型：

```
template <class T> class Vec{
public:
Vec(const Vec& v);  // 复制构造函数
    // 其他部分与前面相同
};
```

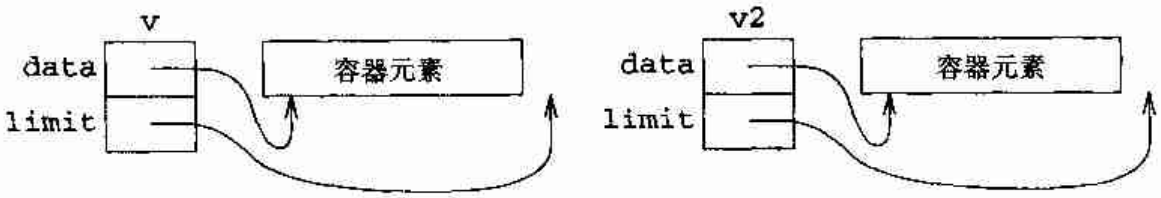
声明了复制构造函数以后，我们可以想像得到这个函数会进行些什么操作。一般来说，复制构造函数会从一个已存在的对象中“复制”每个数据元素到一个新的对象中。注意到

二字上加了双引号，因为在有些时候，复制操作并不仅仅复制数据元素的内容，它可能同时进行了一些其他的操作。下面进一步解释一下这句话。在 Vec 类中有两个指针类型的数据元素，如果我们复制这两个指针的值，那么被复制的对象与复制后的对象都指向内存中的同一个数据。例如假设 v 是一个 Vec 类型的对象，现在把 v 复制到 v2，如果只是复制指针，那么我们将得到下图所示结果：



显然，对其中任何一个对象的数据元素的改动都会影响到另一个对象的值。也就是说，如果对 v[0]赋一个新值，那么 v2[0]的值也被改变了。这难道是我们想要的结果吗？

像在定义其他的操作一样，我们先来看看在标准的向量类中是如何处理这个问题的。回忆一下在 § 4.1.1 中进行的讨论，在那里我们是通过值传递把向量类型对象作为参数传送给 median 函数的，这样做是为了把向量类型对象的元素复制过去而不仅仅是复制它的指针值。复制向量类型对象可以确保在 median 函数内部做的改动不会影响函数外部对象的值。这么看来，事实上运行结果也表明，当按值调用的时候，median 函数内部的向量类型对象与函数外部定义的向量类型对象并不是共享同一块内存。事实上，向量的每一份复件分别占用独立的内存空间，对其中任何一个复件所做的修改不会对其他复件有影响：



显然，在我们的 Vec 类中进行复制操作时，也应该开辟新的内存空间并把源对象的内存复制进新的内存块里。像前面一样，我们先假设已经写好了这样的函数，这样就可以就可以在复制构造函数中先用到这个函数：

```
template <class T> class Vec{
public:
Vec(const Vec& v) { create(v.begin(),v.end());}
// 其他部分与前面相同
};
```

在打算开始写这个函数时我们发现，它实际上像是 create 函数的另一个版本——只

个版本包含了两个迭代器（也就是指针），而且对这两个指针之间的元素进行了初始化。

11.3.2 赋值运算符

在一个类的定义中必须详细描述在类被复制时要进行些什么操作，类似的，在类定义中还必须描述赋值（assignment）的具体操作。一个类中可以定义几种不同的**赋值运算符**（习惯上通过不同的参数进行重载），其中以一个指向类自身的常量引用作为参数的版本比较特殊：它定义了在把一个自定义类型值（对象）赋给另一个自定义类型（对象）时的操作。尽管在类定义中可以定义几个不同版本的“operator=”函数，但是我们一般都是以这个特殊的版本作为代表，把它们叫做“赋值运算符”。像索引运算符一样，赋值运算符也必须是类的一个成员函数。像所有的运算符一样，赋值运算符必须有一个返回值，所以要对其定义一个返回类型。为了与C++自带的赋值算符一致，我们让它返回左操作数的引用：

```
template <class T> class Vec{
public:
Vec& operator=(const Vec&);
// 其他部分与前面相同
};
```

赋值操作与复制构造函数不同，赋值是总是把一个已经存在的值（运算符左侧的对象）擦去，然后代之以一个新的值（运算符右侧的对象）。而在进行复制时，我们先创建一个新的对象，所以不需要对一个已经存在的对象进行删除操作。不过赋值操作通常也要把运算符右操作数对象的每一个元素的值都复制过去，这一点与复制构造函数相同。指针数据成员在进行复制操作时也遵从同样的规律。在进行赋值操作时，要确保运算符右操作数对象里的每个元素都被复制。

在写出代码之前还有一个细节要加以考虑，那就是自我赋值的问题。有时候用户可能会把一个对象赋给它本身。下面我们将看到，正确处理自我赋值问题十分重要：

```
template <class T>
Vec<T>& Vec<T>::operator=(const Vec& rhs)
{
    // 判断是否进行自我赋值
    if (&rhs != this) {
        // 删除运算符左侧的数组
        uncreate();
        // 从右侧复制元素到左侧
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

在这个函数中涉及到两个新概念，下面我们要进一步解释之。

第一个新概念是在除了类的头文件以外的地方定义一个模板成员函数的语法。对于任何一个模板，我们都在一开始就告诉编译器我们要定义一个模板，并且马上给出模板的参数。注意到在这里我们定义返回类型时，用的是 Vec<T>&。对比在头文件中定义返回类型的语法，在头文件中定义时用的是 Vec&类型，不需要显式地声明返回类型名称。因为在模板文件的范围内，C++允许我们忽略其具体类型名称。在头文件里，因为模板参数是隐式的，所以不需要重复写<T>。而在头文件外面，我们必须声明返回类型，所以要在必要的地方显式地写出模板参数。类似地，函数名是 Vec<T>::operator=，而不能简写成 Vec::operator=。不过，一旦在前面指定我们的定义是一个 Vec<T>类型的成员函数，后面就不需要重复使用这个定语了。因此，在参数列表中我们把参数写成 const Vec&的形式，而不必写成 const Vec<T>&这样的复杂形式。

在本函数中还用到了另一个新的关键词：**this**。this 关键词只在成员函数内部才有效，代表指向函数操作的对象的指针。例如，在 Vec::operator=函数里，this 的类型为 Vec*，这是因为 operator=函数是类 Vec 的一个成员函数，所以 this 是指向一个 Vec 类型对象的指针。对于一个二元操作，例如赋值操作等，this 总是指向左操作数。一般来说，我们在需要指向对象本身的时候用 this 关键字，就像在这里的 if 判断表达式和 return 函数中的情况那样。

我们用 this 来决定在赋值操作的左边与右边的对象是否指向同一个对象，如果是指向同一对象，那么它们就共用同一个地址。在前面§10.1.1 中提到过，&rhs 是一个指向 rhs 的指针。this 指针指向左操作数，现在我们通过比较&rhs 与 this 指针来显式地判断是否自我赋值。如果赋值操作的左边与右边指向同一个对象，那么这个赋值操实际上没有执行任何操作，程序直接跳转到 return 语句。如果指向的是不同的对象，那么就需要先释放对象占用的旧的内存空间，然后把新值分别赋给对象中的各个元素，把右操作数的内容复制到新分配的对象空间里。显然，还需要写出另一个实用函数 uncreate，我们要用它来删除 Vec 中的元素，释放其占用的内存空间。一旦调用了 uncreate 函数，抹去了原来的值以后，就可以使用 create 函数来把右操作数的值赋给左操作数。

我们通过显式地检查左操作数与右操作数是否是同一个对象来处理自我赋值的情况，正确地处理这种情况对于赋值操作是至关重要的。为了说明其重要性，我们来考虑一下，如果从我们的程序中去掉这个判断将会出现什么情况。去掉判断后，程序总会调用 uncreate 函数，把左操作数对象的元素删除并释放其占用的内存。注意，如果左右操作数指向同一个对象，那么右操作数也同时被删除。这时候如果还用右操作数的元素来为左操作数生成新的数组元素，那将会带来一场灾难：在释放左操作数对象的内存空间的时候，我们已经把右操作数对象的空间也释放了。当 create 函数试图复制 rhs 的元素时，这些元素实际上已经被删除，其内存早已被释放回系统中。

虽然一般情况下我们都通过一个直接的判断来处理自我赋值的情况，就像在这里所做的一样，但这并不是惟一的方法，而且也不见得在任何情况下都是最好的方法。不过，使用哪一种方法并不重要，最重要的是要正确地处理好自我赋值的情况。

最后还有一个有趣之处是 return 语句，它对 this 间接引用，以此得到它指向的对象

返回一个该对象的引用。因为是返回一个对象的引用，在函数返回以后，引用指向的对象仍然存在。如果将一个引用返回给一个局域量对象将会引发灾难：被引用参照的对象在函数返回时会被删除，将导致一个错误的引用。在赋值操作的例程中，我们返回一个指向表达式左操作数对象的一个引用调用，那个对象的生存周期大于赋值操作，保证了在函数返回的时候不被删除。

11.3.3 赋值不是初始化

经验告诉我们，赋值操作与初始化操作的区别是 C++语言学习中的一个难点。在很多编程语言中，其中包括 C 语言，这二者没有显著的差别，所以程序员一般不必关心它们的差别。而等号“=”既可以用来初始化又可以用来赋值这一事实使我们更加难理解二者的区别。在使用“=”为一个变量赋一个初始值的时候，程序自动调用复制构造函数。而在赋值表达式中，程序调用 operator=赋值操作函数。设计类时类作者必须要注意它们的区别，以实现正确的语义操作。

赋值与初始化有两个主要的区别：赋值（operator=函数）总是删除一个旧的值；而初始化则没有这步操作。确切地说，初始化包括创造一个新的对象并同时给它一个初始的值。在下面的时候会发生初始化：

- 声明一个变量的时候
- 在一个函数的入口处用到函数参数的时候
- 函数返回中使用函数返回值的时候
- 在构造初始化的时候

赋值只在表达式中使用=运算符的时候会被调用。例如：

```
string url_ch = "~,/?:@=&$- .+!*'(),"; // 初始化
string spaces(url_ch.size(),' ');       // 初始化
string y;                               // 初始化
y = url_ch;                             // 赋值
```

在第一个变量声明中生成了一个新的对象，所以程序必须对该对象进行初始化，因此，我们在表达式中给出初始值。具体语法为

```
string url_ch = "~,/?:@=&$-_.+!*'(),";
```

该语句从代表字符串常量“~,/?:@=&$-_.+!*'(),”的 const char*中得到一个 string 对象。为此，编译器将调用 string 的带 const char*参数的构造函数。这个构造函数可以直接用一个字符串常量来构造 url_ch 变量，也可以先用一个字符串常量来构造一个没有名字的临时变量，然后再调用 copy 构造函数来把 url_ch 构造成刚才那个临时变量的一个复件。

第二个变量声明使用了初始化的另一种方式：表达式中直接给出构造函数的一个或几个参数。编译器根据参数的数目与类型来决定调用一个最合适的构造函数。第一个参数告诉函数 spaces 变量将会有几个字符；第二个参数告诉函数变量 spaces 的每一个字符是什么字符

把 spaces 变量定义成与 url_ch 常量有相同的字符个数，其每一个字符都是空格的 string 类型的变量。

第三个变量声明就比较容易了：通过调用默认构造函数来生成一个空的 string 型变量 y。最后那个语句根本就不是一个变量声明。实际上，它在表达式中使用了=运算符，所以这是一个赋值操作。这个赋值操作通过调用 string 类的赋值操作函数来完成赋值。

更复杂一点的例子包括函数参数以及返回值。例如，假设 words 变量是待输出的字符串，现在调用 § 6.1.1 中的 split 函数：

```
vector<string> split(const string&);    // 函数声明
vector<string> v;                       // 初始化
v = split(words);                       // 在 split 的函数入口处初始化 words
                                        // 在出口处则既对返回值进行初始化
                                        // 又将返回值赋值给变量 v
```

split 函数的声明很有趣，因为它使用了一个类来作为函数的返回类型。将一个函数返回的类型对象的返回值进行赋值分成两步操作：第一步，运行复制构造函数，并在这个地方生成一个临时变量，构造函数对该临时变量进行初始化；第二步，运行赋值运算符函数，把这个临时变量的值赋给左操作数。

初始化操作与赋值操作的区别是很重要的，因为其中一个操作的执行可能导致另一个操作的执行，二者是紧密相关的，但要注意：

- 构造函数始终只控制初始化操作。
- operator=成员函数只控制赋值操作。

11.3.4 析构函数

我们还必须提供另一种操作来定义在 Vec 对象被删除时编译器应该做些什么工作。一个在局域范围里被创建的对象在它的生存范围以外时就会被自动删除；而一个动态分配内存的对象则只有在我们使用 delete 来删除它时才会被删除。例如，我们来看看 § 6.1.1 的 split 函数：

```
vector<string> split(const string& str)
{
vector<string> ret;
// 将 str 分离成不同的单词，并将这些单词存储在 ret 变量中
return ret;
}
```

当我们从 split 函数返回时，局域变量 ret 超出了其生存范围，被自动删除。

就像复制与赋值函数一样，我们在类中定义当一个对象被删除的时候要进行一些什么操作。在构造函数中定义了如何创建一个对象，类似地，类中有一个特殊的成员函数叫做 destructor（析构函数），由它来定义如何删除该类的一个对象实例。析构函数的函数名是在类的名字前加一波浪线前缀（~）。析构函数不带任何参数，而且没有返回值。

析构函数的任务是做好一个对象被删除时的一切大扫除工作。这一大扫除工作一般指的是释放资源，例如释放在构造函数中为对象分配的内存资源等。

```
template <class T> class Vec{
public:
~Vec() { uncreate(); }
// 同上
};
```

对于 Vec 类型对象，我们在构造函数中为其分配了内存，所以必须在析构函数中释放该内存。这个工作类似于在赋值操作中删除原来左操作数的内容的工作。请不要觉得奇怪，我们也可以在析构函数中调用相同的函数来删除对象中的元素并释放其占用的内存空间，因为它们的目的都是一样的。

11.3.5 默认操作

有一些类，像在第 4 章与第 9 章中定义的 Student_info 类一样，既没有显式地定义一个 copy 构造函数，也没有显式地定义一个赋值运算符函数或者定义一个析构函数。那么在创建这些类的对象时，对它们进行赋值操作或者删除这类型对象时，逻辑上应该进行什么操作呢？答案是，在编写类时如果没有显式地定义这些操作，编译器将自动为类生成相应的默认版本的函数，进行一些默认的操作。

这些默认的函数被定义成一系列的递归的操作——对每个成员数据按照它们相应的类型规则进行复制、赋值或者是删除。如果成员变量是类的对象实例，那么对它们进行复制，赋值与删除时会调用相应的类的构造函数，赋值运算符函数或析构函数等。如果成员变量是 C++ 自带的变量类型，那么在对它们进行复制或者赋值时将对它们的值进行复制或者赋值，而这类变量在删除时不需要做任何额外的工作，即使其变量类型是指针也不例外。特别值得指出的是，在通过默认析构函数删除一个指针变量时不会释放该指针指向的对象占用的内存空间。

现在我们可以理解 Student_info 类的默认操作是如何执行的了。例如，复制构造函数对四个数值元素进行复制。为此，它要激活 string 与 vector 类的复制构造函数来分别复制 name 与 homework 成员，另外，它还直接复制类中另外两个 double 类型的成员变量，midterm 和 final。

最后还有一个问题，在 § 9.5 中提到过，默认构造函数有一个默认的操作。如果类中没有定义任何构造函数，那么编译器将自动生成一个默认的不带任何参数的构造函数。这一自动生成的构造函数通过一种对象自身在初始化时采取的方式，对其成员数据进行递归初始化。如果上下文要求进行默认初始化，那么它将对数据成员进行默认初始化；如果上下文要求进行值初始化，那么它就对数据成员进行值初始化。

值得指出的是，只要在类中显式地定义了任何一个构造函数（包括复制构造函数），编译器将不会为类自动生成默认的构造函数。默认的构造函数在有些情况下是必需的：其中一种情况就是生成默认构造函数本身。在为一个类生成默认构造函数时，为了对每一个数据成

默认初始化，要求成员数据的类型都有相应的默认构造函数。所以我们应该养成一个良好的编程风格，就是记住为每一个类定义一个默认构造函数，既可以显式地定义（就像在第 9 章中所做的那样），也可以隐式地定义（就象在第 4 章里所做的那样）。

11.3.6 “三位一体”规则（rule of three）

在写一个管理资源（如内存资源）的类时应该特别注意对复制函数的控制。一般来说，默认的操作对于这种类是不够用的。如果不能很好地控制每个复制操作将会让使用者感到迷惑，并最终导致运行时的错误。

以 Vec 类为例，假设在类中我们没有定义复制构造函数，赋值操作或者析构函数。那么就像在 § 11.3.1 中看到的那样，运气好的话我们只会令 Vec 类的使用者们大吃一惊。因为在把一个 Vec 类型对象复制给另一个 Vec 类的对象时，用户一般是希望这两个对象完全独立，对其中任何一个对象所进行的操作不应当对另一个对象有任何影响。

运气不好的话就糟了，如果我们没有定义析构函数，那么默认析构函数将被调用。这个默认析构函数会仅仅删除对象的指针，而删除一个指针不会释放指针指向对象占用的内存空间。最终结果导致内存泄漏：Vec 类型对象使用的内存空间再也不能被收回使用。

如果我们只是想提供一个析构函数来修正这个内存泄漏的错误，仍然不显式地写出复制构造函数与赋值运算符函数，那么情况将会变得更加一团糟。如果这样的话，有可能两个 Vec 类型对象共享同一块内存空间（就像我们在 § 11.3.1 中所描述的那样）。当其中一个对象被删除以后，析构函数将释放那片共享的内存空间。接下来对这片已经释放的内存的任何一个引用都将导致不可预见的后果。

在构造函数中进行了动态资源分配的类都要求该类的每个对象要正确地处理这些资源。这些类几乎都需要一个析构函数来释放资源。如果类需要一个析构函数，那么它就几乎一定需要一个复制构造函数和一个赋值运算符成员函数。对一个动态分配资源的类型对象进行复制或者赋值操作一般都要重新分配内存，就像创建一个对象时做的那样。为了控制 T 类型对象的每一个复件，你需要

`T::T()`	一个或几个构造函数，可能会带参数
`T::~T()`	析构函数
`T::T(const T&)`	复制构造函数
`T::operator=(const T&)`	赋值运算符函数

一旦我们定义了这些成员函数，编译器将会在创建一个类型对象的时候，复制一个类型对象的时候，为一个类型对象赋值的时候，或者删除一个类型对象的时候调用相应的函数。注意，类型对象有可能会被隐式地创建、复制或者删除。无论是隐式的还是显式的操作，编译器都会激活相应的函数。

复制构造函数、析构函数和赋值运算符函数相互之间关系十分密切，它们之间构成了一个“三位一体”规则（rule of three）：如果类需要一个析构函数，那么它同时可能也需要一

制构造函数与一个赋值运算符函数。

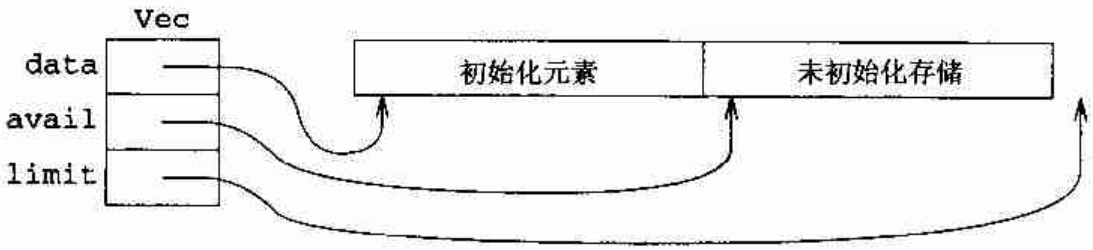
11.4 动态的 Vec 类型对象

在写内存管理函数之前，我们必须认识到现在 Vec 类型对象在某个重要的方面还比不上标准的向量类：因为我们没有在 Vec 类型对象中提供 push_back 操作，所以它的大小是固定的。还记得吗，push_back 函数将它的参数加在向量类型对象的后面作为一个新的元素，这么一来，向量类型对象就增加了一个元素的大小。

现在我们也要向 Vecs 类中加入一个 push_back 函数，它为 Vec 类的对象分配新的内存，它能够比当前对象所占的空间多容纳一个元素。在分配了新的内存空间以后，我们要向这块内存中复制当前对象所容纳的所有元素，并用 push_back 的参数来构造一个新的末端元素。显然，如果用户频繁地调用 push_back 函数的话，这种方法的开销将会是很昂贵的。

为了解决某些问题，有时候会采用一个经典的方法：为程序分配比实际需要更多的内存。只有在我们使用完了所有预分配的内存时，我们才可以申请分配更多的内存。简单地说，无论何时，只要 push_back 函数要求得到更多的内存空间，我们都会为程序分配当前使用的空间的两倍的内存空间。所以，如果我们创建了一个含有 100 个元素的 Vec 类的对象，然后第一次调用 push_bck 函数，这个函数将分配足够容纳 200 个元素的内存空间。它会把现存的 100 个元素复制进新分配内存的前一半空间，并为接下来的第一个元素空间进行初始化。

这种方法表明，我们需要改变一下获得数组中某个元素的方式了。像以前一样，我们仍然要获得数组第一个元素的地址，但与以往不同的是，现在需要两个“末”指针。一个指向我们的构造元素的末元素后的那个元素，也就是指向我们可以访问的第一个元素；另一个则指向新分配内存空间的末元素。所以，现在的 Vec 类型对象将会如下图所示：



幸运的是，只有 push_back 和其他类似的内存管理函数（这些函数现在尚未写出来）才需要知道这些新的元素。更为幸运的是，push_back 函数本身是十分简单的；它把复杂的、困难的工作推给了另外两个内存管理函数：grow 函数和 unchecked_append 函数，而这两个函数根本不需要我们亲自动手编写。

```
template <class T> class Vec{
public:
```

```
    void push_back(const T& val) {
        if (avail == limit)        //获得需要的空间
            grow();
        unchecked_append(val);  //将新增元素加入到对象末端
    }
private:
    iterator data;      //如前，指针指向 Vec 的第一个元素
    iterator avail;     //指针指向构造元素末元素后面的一个元素
    iterator limit;     //现在指向最后一个可获得元素后面的一个元素
    //类的其余接口与实现同前
};
```

11.5 灵活的内存管理

在开始写 Vec 类的时候曾经说过，我们不打算使用 C++内建的 new 运算符与 delete 运算符来管理内存。因为如果使用了 new 与 delete 运算符，我们写出来的 Vec 类使用起来就要受到诸多限制，就不如标准向量类的使用范围那么广了。而且 C++内建的 new 运算符要做许多工作：既要分配新的内存空间，又要对新内存进行初始化。在为一个类型为 T 的数组分配空间时，它需要去调用 T 的默认构造函数。而这有悖于我们想为用户提供尽可能大的弹性的初衷。

使用 new 运算符还可能会为程序带来过多的资源开销。如果我们使用 new，它会使用 T::T() 构造函数为一个类型为 T 的数组中的每一个元素都进行初始化。如果我们想用自己提供的数据来初始化 Vec 类型对象的元素的话，实现上要进行两次初始化——一次是 new 自动进行的，另一次是在把用户提供的数值赋给 Vec 类型对象的元素时进行的。更为糟糕的是，想想在我们使用 push_back 函数中所采用的方法吧，这种方法通过分配我们实际需要的内存的两倍的空间来获得更多的内存空间，我们没有理由要为这些额外的元素进行初始化。这些空间只会在把一个新的数据放进一个元素空间时才会被 push_back 函数使用。而如果用 new 来为数组分配内存空间的话，不管我们需不需要使用这些额外的空间，都无一例外地对它们进行初始化。

除了使用自带的 new 与 delete 运算符来管理内存以外，我们还可以使用 C++专门设计以支持灵活的内存管理的一些类来管理内存，这样我们可以做得更好些。语言核心本身没有提供直接的内存分配管理的语法，因为有关内存的性质实在是太复杂多变了，没有必要把相关的特性固定在语言中。

例如，现在的计算机中有很多种类的内存。一台计算机上可能有几种不同速度的内存在使用。计算机上的内存可能是具有特殊用途的，像图形缓冲内存或者共享内存等。这内存也可能是在断电以后仍然保持记忆的。因为用户有可能想分配其中任何一种类型的内存空间，所以最好是把如何分配和管理内存的工作留给函数库去做。标准函数库并不需要支持所有的内存。内存管理者们一般都不乐意亲自动手写内存管理函数，C++的标准库中提供了管理内存的功能.

但是只是为内存管理者们提供了一个统一的内存管理接口而已。和前面所说的把输入/输出作为标准库的一部分而不是语言的特性相同，内存管理功能也是库的一部分，这一特性为我们方便地管理各种不同种类的内存提供了很大的弹性。

在<memory>头文件中提供了一个名为 allocator<T>的类，它可以分配一块预备用来储存 T 类型对象但是尚未被初始化的内存块，并返回一个指向这块内存块的头元素的指针。这样的指针是很危险的，因为指针的类型表明它们指向类型对象，但实际上这些内存块却并没有储存实际的对象。标准库提供了一种途径来为这种内存块进行初始化，也提供了删除对象的方法——仅仅是删除对象，而没有释放内存空间。由程序员们来使用 allocator 类得到这些被指定用来存放类型对象但实际上没有被初始化的内存空间的地址。

在 allocator 类中，我们只对与我们的实现目的相关的部分感兴趣，这一部分包括四个成员函数和两个相关的非成员函数：

```
template<class T> class allocator {
public:
     T* allocate(size_t);
     void deallocate(T*,size_t);
     void construct(T*,T);
     void destroy(T*);
     //…
};
void uninitialized_fill(T*,T*,const T&);
T* uninitialized_copy(T*,T*,T*);
```

allocate 成员函数用来分配一块被指定了类型但却未被初始化的内存块，它足以储存相应类型对象的元素。被指定了类型的内存，意思是这块内存块将用来储存类型为 T 的值，我们可以通过使用一个 T*类型的指针来得到它的地址。未被初始化的内存，意思是这块内存是原始的，在这块内存块中没有储存任何实际的对象。deallocate 成员函数则是用来释放未被初始化的内存，它带有两个参数：一个是 allocate 函数返回的指针；另一个是该指针指向的内存块的大小。construct 成员函数是用来在 allocate 申请分配但尚未被初始化的内存区域上进行初始化，生成单个的对象，destroy 成员函数则用来删除这个对象。construct 构造函数带有两个参数：一个是 allocate 函数返回的指针；另一个是用来复制到指针指向的内存块的对象值。destroy 函数调用析构函数，删除它的参数所指对象的元素。

另外还有两个相关函数我们比较感兴趣，它们是 uninitialized_copy 和 uninitialized_fill 函数。这两个函数对 allocate 所分配的内存进行初始化。uninitialized_fill 函数向内存块中填充一个指定的值。在函数调用结束后，前两个参数指针指向的内存区间中的元素都被初始化成第三个参数所指对象的内容。uninitialized_copy 函数的工作机理类似于标准库中的 copy 函数，它用来把前两个参数指针所指向的内存区间中的值复制到第三个参数指针所指向的目标内存块中。像 uninitialized_fill 函数一样，它假定目标内存块尚未被初始化而不是已经储存着

际对象的值，它将在目标内存块中构造新的对象。

像所有的模板一样，与 T 相关的实际变量类型在编译时被实例化。这一实例化过程将为每一个使用到该类模板的类型生成一个合适的 allocator 类。为了在 Vec<T>类中包含正确的 allocator 类，我们向 Vec<T>中加入一个 allocator<T>成员函数，它用来正确地为 T 类的对象分配内存。通过加入这个成员函数，并使用它的相关库函数，我们可以提供一种和标准向量类一样有效、一样灵活的内存管理方法。

11.5.1 最后的 Vec 类

最后完成的 Vec 类中包含内存管理函数的声明，但是没有其定义，现在整个类如下：

```
template <class T> class Vec {
public:
     typedef T* iterator;
     typedef const T* const_iterator;
     typedef size_t size_type;
     typedef T value_type;

     Vec() { create(); }
     explicit Vec(size_type n, const T& t=T()) { create(n,t); }

     Vec(const Vec& v) { create(v.begin(),v.end() ) ;}
     Vec& operator=(const Vec&);   //如§11.3.2 中的定义
     ~Vec() { uncreate(); }
     T& operator[] (size_type i) { return data[i]; }
     const T& operator[](size_type i) const { return data[i]; }

     void push_back(const T& t) {
          if(avail == limit )
             grow();
          unchecked_append(t);
     }
     size_type size() const {return avail-data;}

     iterator begin() {return data;}
     const_iterator begin() const { return data; }

     iterator end() { return avail; }
     const_iterator end() const { return avail; }

private:
     iterator data;    //Vec 中的首元素
     iterator avail;   //Vec 中末元素后面一个元素
     iterator limit;   //新分配内存中末元素后面一个元素
```

```
    //内存分配工具
    allocator<T> alloc;   //控制内存分配的对象

    // 为底层的数组分配空间并对它进行初始化
    void create();
    void create(size_type,const T&);
    void create(const_iterator,const_iterator);

    //删除数组中的元素并释放其占用的内存
    void uncreate();

    // 支持push_back的函数
    void grow();
    void unchecked_append(const T&);
};
```

现在剩下的是如何实现用来进行内存分配的私有成员函数。在写这些函数的时候，牢记下面的几条准则对于编写程序十分有益，只要我们有一个有效的 Vec 类型对象，那它必须始终满足以下四个条件：

1. 如果对象中有数据元素的话，data 指向对象数组的首元素，否则为零。
2. data≤avail≤limit。
3. 在[data,avail)区间内的元素被初始化。
4. 在[avail,limit)区间内的元素不会被初始化。

这四个条件叫做**类不变式（class invariant）**。它与§2.3.2 中提到的循环不变式极其相似，一旦构造了一个类的对象，我们总是要建立起类不变式条件。如果满足了这四个条件，而且只要四个成员函数不改变这四个条件，那么这个规律就永远成立。

注意，类中的所有公有成员函数都不能打破这一不变式。因为打破它的办法是改变 data、avail 或者 limit 的值，而这些公有成员函数都不能做到这一点。

我们先来看看几个不同的 create 函数，create 函数用来分配内存空间，并对这片内存进行初始化，以及正确地设置指针。在所有情况下，我们都对新分配的内存空间进行初始化，所以在运行了 create 函数之后，limit 指针与 avail 指针始终是相等的，因为最后一个被初始化的元素也就是最后一个被分配内存的元素。运行下面列出的任何一个函数，你就可以验证类不变式：

```
template <class T> void Vec<T>::create()
{
   data = avail = limit = 0;
}

template <class T> void Vec<T>::create(size_type n,const T& val)
{
    data=alloc.allocate(n);
```

```
    limit=avail=data+n;
    uninitialized_fill(data,limit,val);
}

template<class T>
void Vec<T>::create(const_iterator i,const_iterator j)
{
    data=alloc.allocate(j-i);
    limit=avail=uninitialized_copy(i,j,data);
}
```

不带任何参数的那个版本的 create 函数仅仅生成一个空的 Vec 类型对象，它的工作只是使指针指向零地址。

带一个大小参数与一个值参数的版本的 create 函数利用大小参数申请分配一定大小的内存空间。通过调用 allocator<T>类的 allocate 成员函数分配足够大的内存空间以储存指定个数的 T 类型对象。因此，alloc.allocate(n)函数分配足以储存 n 个 T 类型对象的内存空间。allocate 函数返回首元素的地址，储存在 data 指针变量里。因为 allocate 函数分配的内存没有被初始化，所以我们必须调用 uninitialized_fill 函数来对它进行初始化，这个函数把第三个参数指向的对象内容复制到前两个参数指向的尚未被初始化的内存空间。uninitialized_fill 函数调用结束后，allocate 函数分配的内存中将被构造许多的新元素，并且这些元素都被初始化成 val 值。

create 函数的最后一个版本调用 uninitialized_copy 函数来初始化 allocate 函数分配的内存空间，其余操作与另两个版本相似。uninitialized_copy 函数把第一、第二个参数所指向的区间内的值复制到第三个参数指向的内存空间中，并返回一个指向被初始化的内存中的末元素后面一个元素的地址指针，这个地址正是 limit 与 avail 指针的内容。

uncreate 成员函数所做的工作与 create 函数正好相反：它运行析构函数，删除该对象，并释放其占用的内存：

```
template<class T> void Vec<T>::uncreate()
{
    if(data){
        //（以相反的顺序）删除构造函数生成的元素
        iterator it=avail;
        while(it!=data)
          alloc.destroy(--it);

        //返回占用的所有内存空间
        alloc.deallocate(data,limit-data);
    }
    //重置指针以表明 Vec 类型对象为空
    data=limit=avail=0;
}
```

如果 data 为零，那么将不执行任何操作。如果使用 delete 函数来释放内存，我们就不需要判断 data 是否为零，因为 delete 作用在一个零指针上是不会产生错误的。与 delete 不同的是，alloc.deallocate 函数需要一个非零指针作为参数（即便是它并不准备释放任何内存）。因此，我们必须检查 data 是否为零。

我们用 it 迭代器来遍历 Vec 类型对象中的每一个元素，调用 destroy 函数来删除这些元素，为了使之与 delete[]更加一致，我们让 it 迭代器从后向前以和初始化时相反的顺序遍历 Vec 对象中的元素，所以是以相反的顺序删除各元素。在删除了对象中的元素之后，调用 deallocate 函数释放元素占用的空间。这个函数带两个参数：一个是指向待释放的内存空间的首元素的指针；另一个参数是一个整型值，表示有多少个 T 类型对象的元素要被释放。为了释放所有被分配的内存，我们调用 deallocate 函数来释放 data 指针与 limit 指针所示地址之间的内存空间。

接下来要做的是实现使用 push_back 函数的成员函数：

```
template<class T> void Vec<T>::grow()
{
  //在扩展对象大小的时候，为对象分配实际使用的两倍大小的内存空间
  size_type new_size=max(2*(limit-data),ptrdiff_t(1));
  //分配新的内存空间并将已存在的对象元素内容复制到新内存中
  iterator new_data=alloc.allocate(new_size);
  iterator new_avail=uninitialized_copy(data,avail,new_data);
  //返回原来的内存空间
  uncreate();
  //重置指针，使其指向新分配的内存空间
  data=new_data;
  avail=new_avail;
  limit=data+new_size;
}
//假设 avail 指向一片新分配但尚未被初始化的内存空间
template<class T> void Vec<T>::unchecked_append(const T& val)
{
  alloc.construct(avail++,val);
}
```

grow 函数的任务是分配足够的内存空间，以使其能至少再多储存一倍的元素。它分配了比实际需要更多的内存，接下来调用 push_back 函数以使用多出来的内存，从而避免了频繁地进行内存分配。在 § 11.4 中，我们说到，我们的策略是为每一个请求分配双倍的内存空间。当然，Vec 对象当前可能是空的，这样我们就可以选择一个元素以及当前空间的两倍大小中的较大值进行内存分配。翻翻 § 8.1.3 我们可以看到，max 函数的两个参数必须是同一类型，我们用 ptrdiff_t 类型的值 1 来初始化一个对象，从 § 10.1.4 我们知道，它是一个 limit-data 类型。

现在回忆一下前面所做的工作，在 new_size 变量里储存了我们要为对象分配的内存空间的大小。我们要先分配一个合适大小的内存空间，然后调用 uninitialized_copy 函数把当前

的元素内容复制到新分配空间中。然后释放旧的内存，调用 uncreate 函数删除旧元素。最后重置指针使 data 指向新分配内存的数组的首元素，limit 指针指向新构造的 Vec 类型对象元素的末元素后面一个元素，而 avail 指针指向新分配了空间但尚未被初始化的末元素的后面一个元素。

注意，我们要保存好 allocate 函数与 uninitialized_copy 函数返回的值，这一点很重要，因为如果我们马上要使用这些数值来重置 data 与 limit，那么接下来对 uncreate 的调用将删除并释放刚才分配的内存空间，而不是删除并释放旧的内存空间。

unchecked_append 函数在紧跟着构造的元素后的位置新生成一个元素。它假定 avail 指向的内存空间是已分配但是尚未被初始化的，还没有被用来储存任何构造元素。不过因为我们一般只在前面调用了 grow 函数之后马上调用 unchecked_append 函数，所以这个函数使用起来还是很安全的。

11.6 小结

模板类可以使用在 § 8.1.1 中提到的模板工具来生成：

```
template <class type-parameter [, class type-parameter ]…>
class class-name { … };
```

上面的语句生成一个名为 class-name 的模板类，类名取决于给定的类型参数。这些类型参数名称可以在模板内任何一个需要的地方使用。在类的范围里任何地方，模板类都可以无需加任何限定词就直接使用；在类范围以外的地方使用时，必须使用类型参数来限定 class-name 前加上类参数名：

```
template<class T>
 Vec<T>& Vec<T>::operator=(const Vec&){…}
```

在创建模板类型的对象时，用户需要为模板指定实际的类型参数：Vec<int>就是一个和类型参数 int 相绑定的 Vec 版本的实例化。

复制控制：一般来说，由类来控制对象在被创建、复制、赋值以及删除时具体干些什么。在创建和复制对象时会调用构造函数；在包括赋值的表达式中会调用赋值运算符函数；而析构函数则是在对象被显式地删除，或者程序运行到对象的生存范围以外的地方的时候被调用。

在构造函数中分配资源的类几乎都要定义复制构造函数、复制运算符函数以及析构函数。在写一个赋值运算符函数的时候，一定要注意判断是否是自我赋值。为了与 C++自带的赋值运算符函数一致，最好在函数中返回一个对左操作数的引用调用。

自动生成的操作：如果一个类没有定义任何构造函数，编译器将自动生成默认的构造函数。如果在类中没有显式地定义复制构造函数、赋值运算符函数以及析构函数，编译器会自动生成相应的默认成员函数。这些自动生成的操作被定义为递归操作：它会递归地对类中的每

数据调用相应的操作。

重载运算符函数指的是对一个已经被定义过的运算符 op 进行重复定义，函数名为 operator op。该运算符中至少有一个操作数的类型和该类相同。如果某个运算符函数是类的一个成员函数，那它的左操作数（适用于二元运算符）或者它的惟一的操作数（适用于一元运算符）必须是调用它的对象。索引运算符与赋值运算符都是类的成员函数。

习题

11-0 编译、执行并测试本章讲到的程序。

11-1 在第 9 章中定义的 Student_info 结构中没有定义复制构造函数，也没有定义赋值操作或者析构函数，为什么？

11-2 在那个结构中也没有定义一个默认的构造函数，为什么？

11-3 Student_info 的对象在调用自动生成的赋值操作函数时具体有什么操作？

11-4 在 Student_info 中，自动生成的析构函数删除了多少个成员变量？

11-5 在 Student_info 类加入计数代码，计算一下对象被创建、复制、赋值或者删除了多少次。用这个可计数的类来运行第 6 章的学生成绩程序。使用这个具有计数功能的 Student_info 类可以算出在库中的算法进行了多少次复制操作。对比库中不同的类测到的复制次数，可以估计出这些不同的类对资源的消耗量。试一试进行计数并对结果进行分析。

11-6 在 Vec 类中增加一个删除其中一个元素的操作，再增加一个清空整个 Vec 类型对象的操作。它们的作用与向量类中的 erase 与 clear 成员函数相同。

11-7 在向 Vec 类中加入了 erase 与 clear 函数后，在本书前面举出的大部分例程中都可以用 Vec 类代替向量类。用 Vec 代替向量，重写第 9 章的 Student_info 程序和第 5 章中处理字符图形的程序。

11-8 为标准 list 类以及它的相关迭代器写一个简化版本。

11-9 §11.5.1 中的 grow 函数为程序分配了实际需要的两倍的内存空间。估算一下这种方法能对工作效率有多大的提高。如果你能估计出它的不同之处，请适当地改写 grow 函数，然后测量一下它们的差别。

第 12 章

使类对象像一个数值一样工作

C++内建类型的对象使用起来像一个数值一样：在对这种对象执行复制操作时，源对象与它的那个复件值内容一样，但它们是彼此独立的。改变其中任何一个对象都不会影响到另外一个对象。我们可以随意创建这种类型的对象，把它们作为参数传递给函数，作为函数的返回值，对它们进行复制，或者把它们赋值给其他的对象。

对于大多数的自带类型，C++语言定义了大量的操作，并且为逻辑上相似的类型之间提供了自动转换功能。例如，如果我们将一个 int 和一个 double 相加，编译器将自动把 int 转换为 double。

当我们定义了自己的类时，我们就可以控制对类进行的扩展，使其对象可以像数值一样工作。在第 9 章中我们看到，在编写一个类的时候必须亲自定义对象如何被创建、复制、赋值以及删除。适当地定义复制与赋值操作，可以使该类的对象工作起来就像是数值一样。也就是说，类的编写者们可以可以设法使各个对象之间彼此独立。我们写的 Vec 类与 Student_info 类就是这样的例子。

在本章，我们可以看到在 C++中，类的编写者们可以控制类型转换以及对类型对象的相关操作，从而写出与 C++自带类型的对象几乎一样的类。标准库中的 string 类就是这种类型的一个很好的例子，它有丰富的函数支持自动转换。因此，像在第 11 章中编写向量类的一个简化版本 Vec 一样，在本章我们将定义 string 类的一个简化版本，并且把该类的名字叫做 Str。我们将把注意力放在如何实现 strings 具备的操作与转换功能上。本章不会太关心程序执行效率的问题，在第 14 章我们会回过头来看 Str 类，并讨论如何更加有效地利用每个 Str 对象的内存的技术问题。

我们不用担心 Str 类的实现细节，因为在写 Vec 类时我们已经完成了其大部分的工作。相应地，本章的大部分讨论将围绕着如何为 Str 类设计一个友好的接口而展开。

12.1 一个简单的 string 类

现在让我们开始写一个简单的 Str 类，这个类可以像我们希望的那样生成对象：

```
class Str{
public:
    typedef Vec<char>::size_type size_type;
    //默认构造函数，创建一个空的 Str
    Str(){}
    //生成一个 Str 对象，包含 c 的 n 个复件
    Str(size_type n, char c):data(n,c){}
    //生成一个 Str 对象并用一个空字符结尾的字符数组来初始化
    Str(const char* cp){
        std::copy(cp,cp+std::strlen(cp),std::back_inserter(data));
    }
    //生成一个 Str 对象并用迭代器 b 和 e 之间的内容对它进行初始化
    template<class In> Str(In b, In e){
        std::copy(b,e,std::back_inserter(data));
    }
private:
    Vec<char> data;
};
```

Str 类中把管理数据的工作交给了我们在第 11 章中写的 Vec 类。Vec 类与我们要写的 Str 类功能十分相似；不过它还缺少一个 clear 函数，当时我们把这件工作作为第 11 章后面的一个练习题。

Str 类有四个构造函数，它们都用来生成成员数据 data，并适当地对这个 Vec 类型对象进行初始化。

Str 的默认构造函数隐式地调用 Vec 类的默认构造函数，生成一个空的 Str 类型对象。注意，因为 Str 类还有其他的构造函数，所以虽然默认构造函数与编译器自动生成的版本做的工作一样，我们仍需显式地定义默认构造函数。其他三个构造函数带有参数，用来对 data 进行初始化。

带有一个大小参数与一个字符参数的构造函数用 Vec 类中相应的构造函数来构造 data 数据，除此之外，它不进行其他的操作，所以函数调用完毕后对象仍为空。

另外两个构造函数彼此很相似。它们的构造预置状态为空，也就是说数据 data 被隐式地初始化为一个空的 Vec 类型对象。每次构造数据时都调用 copy 函数把参数提供的字符加到最初为空的 data 对象里。例如，带有一个 const char*的那个构造函数用 strlen 来得到字符串的长度，根据这个长度，它可以计算出表示这些加入的字符的两个迭代器，并调用 copy 与 back_inserter 函数来把这些字符加入 data 中。因此，这个构造函数的调用结果是生成一个 data 成员数据，在 data 中储存了由 cp 指向的数组中的字符内容的复件。

最有趣的算是最后那个构造函数了，它带有两个迭代器参数，用来新生成一个包含给定序列中所有字符的复制的 Str 类型对象。像前面几个构造函数一样，它通过 copy 与 back_inserter 函数把[b,e)区间内的内容加到 data 对象的后面。这个构造函数的有趣之处在于，它本身是一个模板函数。因为是一个模板，它事实上有效地定义了一组构造函数，随着不同类型的

实例化出不同的构造函数。例如，这个构造函数可以被用来从一个字符数组构造一个 Str 类型对象，也可以从一个 Vec<char>类型对象构造 Str 类型对象。

值得一提的是，在 Str 类中并没有定义复制构造函数，也没有定义赋值运算符函数或者析构函数，想想这是为什么？

答案是，因为有默认的操作。Str 类本身没有分配内存的能力。它把管理内存的细节留给编译器，让编译器自动生成相应的函数，而这些函数通过调用 Vec 中的相应函数来进行操作。为了看清楚这些默认的操作，我们注意到 Str 类不需要一个析构函数。实际上，如果真要为它写一个析构函数，这个析构函数也没有什么工作要做。一般来说，一个不需要析构函数的类也不需要显式地定义复制构造函数或赋值运算符函数（§11.3.6）。

12.2 自动转换

至此，我们已经定义了一组构造函数，并隐式地定义了复制、赋值和析构函数。这些操作的定义使得 Str 类型对象使用起来像一个值一样：在我们复制一个 Str 类型对象的时候，源对象与其复制出来的对象彼此独立。接下来要考虑的是转换问题。C++自带的类型变量之间可以自动地进行转换。例如，我们可以用一个 int 类型的值来初始化一个 double 类型变量，也可以把一个 int 类型的值直接赋给一个 double 类型变量：

```
double d=10;  //把 10 转换成 double 类型并用来初始化 double 变量 d
double d2;
d2=10;        //把 10 转换成 double 类型并将它赋给 double 变量 d2
```

在我们的 Str 类中，我们定义了如何用一个 const char*类型的参数来构造一个 Str 类型对象，所以可以写出下面的定义

```
Str s("hello");              //构造 s
```

这个定义显式地调用构造函数，用一个 const char*类型的变量来构造变量 s。我们也可以把它写成下面的形式

```
Str t="hello";              //初始化 t
s="hello";                  //把一个新的值赋给 s
```

回忆一下，在§11.3.3 中我们曾经提到，符号=在上一例子中有两层含义。第一个语句定义了 t。这种形式的初始化始终需要调用一个参数类型为 const Str&的复制构造函数。第二个语句不是声明，而是一个表达式语句，所以=是一个赋值运算符。与 Str 类型对象关联的惟一一个赋值运算符函数需要一个 const Str&类型的参数，是由编译器自动为我们定义的。换句话说，第二个例子中的每个语句都使用的是 const char*类型的字符串常量，而事实上在这儿要求用的是 const Str&类型的值。

因此，我们也许会考虑给 Str 类添加一个参数类型为 const char*的赋值运算符函数

决如何对复制构造函数进行重载的问题。幸运的是，实际上我们并不需要这么做，因为我们已经有了一个带 const char*参数的构造函数，而且那个构造函数就像一个**用户定义的转换**（user-defined conversion）一样工作。用户定义的转换定义了如何把一个其他的对象转换成该类的对象，以及反过来的操作。像 C++自带的转换一样，编译器在使用用户定义的转换时把一个值转换成程序需要的类型。

在类中定义类型转换包括两个方面的定义：把其他类型转换成该类类型，或者把该类类型转换成其他类型。第二种转换我们留在§12.5 再进行讨论。更常用的转换定义是定义如何把其他类型转换成本类类型。现在就要进行这种转换的定义，我们通过定义一个只带有一个参数的构造函数来定义类型转换。

在 Str 类中已经有这样的一个构造函数，它带有一个 const char*类型的参数。所以，编译器可以在需要一个 Str 类型的对象，但是程序提供的却是一个 const char*类型的对象时候调用这个构造函数。在把一个 const char*类型的对象赋给一个 Str 类型的变量的时候也会调用这个构造函数。在我们写出 s="hello"这样的表达式时，编译器实际上调用 Str(const char*)构造函数为这个字符串常量构造一个没有名字、局部的、临时的 Str 类型对象。然后再调用编译器自动生成的赋值运算符函数把这一临时值赋给 s。

12.3 Str 操作

回想一下我们写的用到 string 类的程序代码，可以看到这些代码用到了以下的几种操作：

```
cin >> s        //用输入运算符来读取一个字符串
cout << s       //用输出运算符来写出一个字符串
s[i]            //用索引运算符访问 s 中的一个字符
s1 + s2         //用加运算符连接两个字符串
```

上面的几个运算符都是二元运算符。所以如果我们把它们定义成为函数，每个函数都应该有两个参数。如果被定义的是一个成员函数，那么其中一个参数可以是隐式被提供的。在§11.2.4 中提到过，重载运算符的函数名是在单词 operator 后面加上运算符的符号。例如，重载输入运算符的函数名为 operator>>，而重载索引运算符的函数名为 operator[]，等等。

在§11.2.4 中，我们已经看到如何实现索引操作，而且知道它是类的一个成员函数，现在我们从讲解索引运算符函数开始。

```
class Str{
public:
    //构造函数同前
    char& operator[](size_type i) { return data[i]; }
    const char& operator[](size_type i) const { return data[i]; }
private:
    Vec<char> data;
```

```
};
```

索引运算符函数只是把它的工作交给 Vec 类中的相应函数。值得提出的是，在 Vec 类中我们定义了索引操作的两个版本——其中一个可以对常量对象进行操作，而另一个不行。只能处理非常量对象的那个索引操作函数版本返回指向字符的引用，所以可以对返回的字符进行写操作。而可以处理常量对象的那个索引操作版本返回一个指向一个 const char 变量的引用，这样用户就不能对之进行修改。实际上我们返回一个 const char&而不是一个简单的 char，这是为了与标准的 string 类保持一致。

至于其他的操作又如何呢？在定义这些函数的时候最有趣的问题是决定哪些函数作为 Str 类的成员函数。要解决这个问题，必须先知道这几个操作的不同之处。现在我们先来看看输入/输出运算符函数，在 § 12.3.3 中再对连接运算符进行讨论。

12.3.1 输入-输出运算符

在 § 9.2.2 中，我们必须判断 compare 函数是否是 Student_info 类的一个成员函数。当时建议的一种判断方法是看这个操作是否会改变对象的状态。输入运算符当然是会改变对象的状态的。毕竟，在使用输入操作的时候会将一个新值写入一个已经存在的对象中。所以我们考虑将输入运算符函数作为 Str 类的一个成员函数。但是实际上，这么做并不会像我们想像的那样工作。

为了解释为什么会这样，我们回忆一下在 § 11.2.4 中讲到的，表达式中的操作数与重载运算符函数的参数是如何紧密相关的。对于一个二元运算符函数，其左操作数必须作为函数的第一个参数，右操作数必须作为函数的第二个参数。如果该运算符函数是成员函数，那么第一个参数（也就是左操作数）总是默认地传递给该成员函数。因此，

```
cin >> s;
```

等价于

```
cin.operator>>(s);
```

它调用了对象 cin 的被重载了的>>运算符。这种行为暗示着>>运算符必须是类 istream 的一个成员。

当然，我们对于 istream 的定义没有权限修改，也就不能把这个操作添加到它里面。如果我们把 operator >>作为 Str 的一个成员，那么我们的用户将不得不用如下的方式来对 Str 的进行输入操作：

```
s.operator>>(cin);
```

或者，使用它的等价形式：

```
s >> cin;
```

这一点与整个库的语法规则有点不同。因此，我们可以下这么一个结论：输入/输出函数不能做为类的成员函数。

我们继续来完善 Str 类，现在要向 Str.h 头文件中加入输入/输出运算符函数的声明：

```
std::istream& operator>>(std::istream&,Str&);          //新增加
std::istream& operator<<(std::ostream&,const Str&)     //新增加
```

输出运算符函数的定义十分简单：它用一个迭代器遍历 Str 类中的每个元素，每次输出一个单个的字符：

```
ostream& operator<<(ostream& os, const Str& s)
{
    for(Str::size_type i = 0; i!=s.size(); ++i)
        os<<s[i];
    return os;
}
```

这种用法还要求我们再给 Str 类增加一个 size 函数：

```
class Str{
public:
     size_type size() const { return data.size(); }
     //其他同前
};
```

虽然输出运算符函数十分简单，但是我们还是要彻底地理解它的每一步操作。在循环中的每一步中，我们都调用 Str::operator[]函数来获得一个字符并输出它。而这个函数又会调用 Vec::operator[]函数来从 Vec 类型对象的元素中得到实际的值。类似的，在每一步循环中，我们调用 s.size()函数来获得 Str 类型对象的大小，而 s.size()函数又调用 Vec 类型对象的 size 成员函数来得到该对象的实际大小。

12.3.2 友员函数

输入运算符函数比输出运算符函数稍难一点，但也不会难到哪里去。它需要从输入流中读出字符并把它储存起来。每次使用输入运算符时，它都会读取字符，但它会忽略打头的空格字符，然后连续地读出其他字符并储存起来，直到遇到另一个空格字符或者遇到一个文件结尾标志。我们的输入运算符函数是一个简化的版本（它忽略了某些输入/输出库操作上的细节，因为那些知识超出了本书的范围）不过它却的确能完成我们想要做的工作：

```
//这些代码还不能被编译通过
istream& operator>>(istream& is, Str& s)
{
    //抹去存在的值（s）
    s.data.clear();
```

```
    //按序读字符并忽略前面的空格字符
    char c;
    while(is.get(c) && isspace(c))
        ;        //只判断循环条件，不进行其他工作
    //如果读到非空格字符，重复以上操作直到遇到一个空格字符
    if(is){
        do s.data.push_back(c);     //产生一个编译错误，因为 data 是私有成员数据
        while (is.get(c) && !isspace(c));
        //如果遇到一个空格字符，把它放在输入流的后面
    }
    return is;
}
```

我们先来解释一下这个函数，然后再来看看它为什么不能通过编译。

首先，我们忽略在 data 中储存的任何已存在的值，因为向一个 Str 类型对象中读入任何东西都会把 data 当前的值擦除。然后，我们要从输入流中读取字符，每次读一个字符，直到遇到一个非空格字符。因为要判断读出的字符是不是一个空格字符，所以我们对输入流调用 get 函数。在重载输入运算符>>的时候，我们忽略空格字符，在这里 get 函数读取在输入流中的下一个字符，包括空格字符在内。因此，在 while 循环中不断读取字符，直到它遇到一个非空格字符，或者读完了输入流。如果读出来的字符是空格字符，那么程序不需要做任何工作而是继续读下一个字符，所以 while 循环的循环体是空的。

if 语句判断是否读到一个非空格字符或者超出了输入流的范围，如果是的话就退出 while 循环。如果读到的是一个非空格字符，那么我们将继续读字符直到再次遇到一个空格字符，并把每次读出来的字符放入 data 变量中。这些操作是在接下来的 do while 循环中进行的（有关 do while 循环的介绍见 § 7.4.4），在 do while 循环体中，我们把在上一个 while 循环中读到的字符加在 data 原有的字符串后面，然后继续读字符，直到遇到一个空格字符或者超出输入流范围。每次读到一个非空格字符，程序都调用 push_back 函数把该字符加到 data 的字符串后面。

无论是在不需要再从 is 输入流中读取字符的时候，还是在遇到一个空格字符的时候，我们都要从 do while 循环中退出。对于前者，我们认为读到的这个空格字符是多余的，所以调用 is.unget()函数把这个空格字符放回输入流。unget 函数取消最近一次从输入流中读取一个字符的操作，在调用 unget 函数返回后，输入流看上去就像没有调用过 get 函数一样。

在注释中说到，这段代码不能通过编译，问题出在运算符函数 operator>>上，它不是 Str 类的成员函数，所以不能访问 s 的私有成员数据 data。在 § 9.3.1 中，compare 函数要访问 Student_info 对象的 name 成员数据的时候我们面临过类似的问题。在那里我们通过增加一个访问函数来解决问题。在这里，仅仅为 data 增加一个访问函数还不够：因为输入运算符函数不仅要读出 data 变量的值，还要向 data 变量中写入新的值。输入运算符函数是 Str 抽象概念的一部分，所以给该函数赋予对 data 的写权力是明智的。另一方面，因为我们不想让所有用户都对 data 数据拥有写的权力。所以也不能通过把 data 作为 Str 类的一个公有数据成员来

题。

除了定义一个公有的访问函数以外，我们还有一种更好的解决办法，就是把输入运算符函数声明为 Str 类的一个**友员**函数。友员函数拥有和成员函数一样的的访问权力。通过把输入运算符函数声明为 Str 类的友员函数，我们赋予了该函数与类的成员函数一样的对 Str 类私有成员数据的读写权力：

```
class Str{
    friend std::istream& operator>>(std::istream&,str&);
    //其余同前
};
```

我们已经向 Str 类中加入了一个友员函数声明。这一声明说明 operator>>函数的这个版本带有一个 istream&类型和一个 Str&类型的参数，它可以访问 Str 类的私有数据成员。在向 Str 类中加入了这样的声明后，前面的那段程序就可以通过编译了。

友员函数的声明可以加在类定义的任何一个地方：把它加在一个 private 标识后面与加在一个 public 标识后面没有任何区别。因为友员函数具有特殊的访问权力，所以它是类接口的一部分。因此，一般在类声明的前面，public 接口的附近，把所有友员函数的声明放在一起作为一个相对独立的组。

12.3.3 其他二元运算符

接下来的工作是实现加号（+）运算符函数。在继续编写程序之前，我们先考虑以下几个问题：这个运算符函数是否是一个成员函数？它的操作数是什么类型？该函数返回什么类型的结果？接下来我们会看到，这些问题有些微妙的含意。

现在，让我们来对答案做一些猜测。首先，我们希望可以连接两个 Str 类型的对象的值。其次，在进行两个对象的连接的时候不能改变原来两个操作数的值。这些要求表明，没有什么特殊的理由要求该运算符函数是一个成员函数。最后，我们还希望能够在一个简单的表达式中对几个 Str 类型的对象进行连接，如下所示：

```
s1+s2+s3
```

其中 s1，s2 和 s3 都是 Str 类型的对象。这种用法要求加号运算符函数的返回类型必须为 Str 类型。

这些决定意味着我们要把加号运算符函数实现成一个非成员函数：

```
Str operator+(const Str&, const Str&);
```

在开始具体的函数实现工作之前稍微考虑一下就会知道，如果我们提供了 operator+函数，那么最好同时提供 operator+=。也就是说，我们希望能够让用户用下面两种形式之一来连接 s 和 s1，然后将连接结果赋给变量 s：

```
s = s + s1;
s += s1;
```

事实证明，要想方便地实现 operator+函数，最好先写出 operator+=函数。与简单的连接操作不同，复合的版本改变了运算符的左操作数，所以我们把它写成 Str 类的一个成员函数。在往 Str 类中加入了新的连接运算符函数声明以后，现在 Str 类如下所示：

```
class Str{
    //输入运算符函数的实现见§12.3.2
    friend std::istream& operator>>(std::istream&,Str&);
public:
    Str& operator+=(const Str& s){
        std::copy(s.data.begin(),s.data.end(),
                std::back_insertr(data));
        return *this;
    }
    //同前
    typedef Vec<char>::size_type size_type;
    Str() { }
    Str(size_type n, char c):data(n,c) {}
    Str(const char* cp){
        std::copy(cp,cp+std::strlen(cp),std::back+inserter(data));
    }
    template<class In> Str(In i, In j){
       std::copy(i,j,std::back_inserter(data));
    }
    char& operator[](size_type i) { return data[i]; }
    const char& operator[](size_type i) const { return data[i]; }
    size_type size() const { return data.size(); }
private:
    Vec<char> data;
};
//输出运算符函数的实现见§12.3.2
std::ostream& operator<<(std::ostream&, const Str&);
Str operator+(const Str&,const Str&);
```

因为我们在底层的存储中用的是 Vec 类型对象，所以实现 operator+=的工作很琐碎：调用 copy 函数把右操作数的一个复件追加到左操作数的那个 Vec 类型对象上，然后像赋值操作里做的那样，返回一个指向左操作数对象的引用作为结果。

现在可以用 operator+=函数来实现 operator+函数了：

```
Str operator+(const Str& s,const Str& t)
{
    Str r = s;
    r += t;
```

```
    return r;
}
```

记住，这个连接函数是一个非成员函数，它会生成一个新的 Str 类型对象。在上面的程序中，我们首先定义一个局域变量 r，将它初始化成 s 的一个复件来生成一个新的 Str 类型对象。这个初始化过程用到了 Str 复制函数。然后调用+=运算符函数，将 r 与 t 连接作为 r 的新值。最后返回 r（再次隐式地调用复制构造函数）作为结果。

12.3.4 混合类型表达式

我们已经定义了连接运算符，它的操作数是 const Str&类型的对象。如果表达式中包含字符指针，那又如何呢？例如，我们如何用 Str 类来实现 § 1.2 中的程序？那个程序里有下面这行代码

```
const std::string greeting = "Hello, "+name+ "!";
```

其中 name 是一个 string，类似的，该式我们也可以这么写

```
const Str greeting = "Hello"+name+ "!";
```

其中的 name 现在是一个 Str 类型的对象。

我们知道+运算符是从左到右进行的，也就是说上面的表达式等价于先计算下式

```
"Hello, "+name
```

然后将其结果再与“!”相加，也就是说，等价于下式

```
("Hello, "+name) + "!"
```

通过把上式分解成几个不同的组元，我们看到事实上有两种不同形式的+。在其中一种情况下，我们把一个字符串常量传递给加号运算符函数作为第一个操作数，把一个 Str 类型对象传递给它作为第二个操作数。在另一种情况下，左操作数是连接之后生成的一个 Str 类型的对象，而右操作数是一个字符串常量。因此，在两种情况中，我们都是以一个 const char*类型和一个 Str 类型作为参数调用加号运算符函数，只不过两个参数的顺序不同而已。

在 § 12.3.3 中，我们定义+运算符函数的参数类型是 Str，而不是 const char*。不过，从 § 12.2 中可以知道，通过定义一个以 const char*类型为参数的构造函数，我们也就同时定义了一个从 const char*到 Str 的类型转换操作。显然，在 Str 类中已经对这些表达式进行了处理。在任何一种情况里，编译器都先把 const char*类型的参数转换成 Str 类型的参数，然后再调用 operator+函数。

理解这个转换操作的机制十分重要，例如，

```
Str greeting = "Hello, " + name + "!";
```

上式运行后 greeting 对象的结果与下面的代码的运行结果相同

```
Str temp1("Hello, ");          //Str::Str(const char*)
Str temp2 = temp+name;         //operator+(const Str&,const Str&)
Str temp3("!")                 //Str::Str(const char*)
Str s = tmep2 + temp3;         //operator+(const Str&,const Str&)
```

在上面的代码中用到了很多个临时变量，所以使用这种方法是很耗内存的。实际上，因为这样做要生成许多临时变量，对内存消耗极大，所以真正标准库中的 string 类不是依赖于自动转换来实现混合类型的操作数相加，而是用一种十分繁琐的方法，重载加号操作数函数，为每一种可能的操作数类型的连接定义一个版本。

12.3.5 设计二元运算符

在二元运算符的设计中参数转换的地位十分重要。如果一个类支持转换，那么把二元运算符定义成非成员函数是一个良好的习惯。这么做可以保证两个操作数的对称性。

如果一个运算符函数是类的成员函数，那么这个运算符的左操作数不能是自动转换得到的结果。之所以有这样的限制，是因为当一个程序员在写像 x+y 这样的表达式时，编译器不会去对整个程序进行检测，以发现把 x 转换成一个拥有成员 operator+的类型的可能性。因为这个限制，编译器（以及程序员）不得不亲自检查作为非成员函数的 operator+函数以及 x 类的 operator+成员函数。

一个非成员运算符函数的左操作数以及所有运算符函数的右成员函数，都遵循与一般函数的参数一样的规律：操作数可以是任何一种能被转换成参数指定类型的类型。如果定义该二元运算符函数是一个成员函数，那么也就同时引入了操作数的非对称性：右操作数可能是自动转换之后的结果，但左操作数不能。这种非对称性对于一个像+=这样的固有的非对称操作函数来说不是问题，可是在对称操作数的环境中，这种要求会使人迷惑并可能导致错误。一般来说都希望这样的两个操作数是完全对称的，这就要求我们把运算符函数定义成一个非成员函数。

在赋值这个二元运算符函数里，我们规定左操作数的类型必须是该类的类型。如果不这样又会发生什么事呢？如果我们允许左操作数被转换，那么就可能会把操作数转换成该类的对象，并把它放在一个临时变量中，最后把一个新的值赋给这个临时变量。既然这是一个临时变量，在赋值操作完成后，我们没有办法访问到刚才生成的这个对象！因此，就像赋值运算符函数一样，所有的复合赋值操作也必须是类的成员函数。

12.4 有些转换是危险的

回忆一下，在§11.2.2 中我们用关键字 explicit 修饰了一个带有一个大小参数的构造函数。现在我们知道在带一个参数的构造函数中定义了自动转换，也就能理解在对构造函数使用 explicit 关键字的时候实际上发生了什么：这么做可以告诉编译器该构造函数只能用来显式地构造对象。编译器不会用一个 explicit 类型的构造函数通过隐式地转换操作数类型来生成

为了理解把构造函数定义为 explicit 类型的好处，让我们来做一个假设，假设我们没有把向量的构造函数声明为 explicit。那么我们就可能会隐式地生成一个指定大小的向量类型对象。在调用一个函数（例如在§5.8.1 中的 frame 函数）的时候我们可以隐式地进行转换。回忆一下，那个函数带有一个 const vector<string>&类型的参数，它把一个框架加在输入的向量的周围，生成一个字符图形。如果构造函数不是 explicit 的，那么我们就可以调用

```
vector<stirng> p = frame(42);
```

结果怎样呢？又应该是怎样的？更重要的是，用户将怎样认为呢？

事实上在这种情况下，用户只得到一个围绕着一个空图形的框架，这个空图形包含 42 行的空（'\0'）字符。这是用户要的东西吗？用户原本不是希望程序给 42 这个值加上一个方框的吗？这种函数调用确实是一种错误，所以我们在向量类中将带有一个整形参数的构造函数声明为 explicit。

一般来说，我们总是把定义将要被构造对象的结构的构造函数声明为 explicit。有些构造函数的参数最后会变成对象的一部分，这些构造函数一般就不必要被声明为 explicit。

作为一个例子，string 类与 Str 类都有带一个 const char*类型参数的构造函数，它们就没有声明为 explicit。这两个构造函数都用它们的 const char*类型的参数来初始化对象的值。因为参数决定了生成的对象的值，所以允许在表达式和函数调用中对 const char*进行自动转换的操作是一件明智的事情。

另一方面，vector 类与 Vec 类的构造函数中带有一个 Vec::size_type 类型参数的构造函数被声明成 explicit。这两个构造函数使用它们的参数值来决定为对象分配多少个元素的内存。这些构造函数的参数只决定了对象的结构，而不是对象的内容。

12.5 类型转换操作函数

在§12.2 中，我们看到有些构造函数也定义了转换操作。类的编写者们可以定义显式的类**型转换操作**，该操作定义了如何把一个对象从原来的类型转换成一个希望得到的类型。转换操作必须定义成类的成员函数。转换操作函数的函数名为 operator 加上目标类型名。因此，如果一个类有一个名为 operator double 的成员函数，这个成员函数可以用来把一个该类类型的变量转换成 double 类型的变量。例如，

```
class Student_info{
public:
    operator double();1
    //...
};
```

上面的代码定义了如何把一个 Student_info 类的对象转换成一个 double 类型的变量

转换的具体意义依赖于操作函数的具体定义。在需要一个 double 类型的值的地方，程序提供的却是一个 Student_info 类型对象，这时候编译器自动调用这个转换操作函数。例如，假设 vs 是一个 vector<Student_info>的对象，那么为了计算所有学生的平均成绩，我们可以像下面这样编写程序：

```
vector<Studen_info> vs;
//填充 vs
double d=0;
for(int i=0;i!=vs.size();++i)
    d+=vs[i];    //vs[i]被自动转换成 double 类型的值
cout<<"Average grade:"<<d/vs.size()<<endl;
```

转换操作函数在把一个自定义类型转换成一个 C++内建的类型的时候经常被调用，有时候也可以用它来把一个类的类型转换成另一个我们没有代码的类的类型。在两种情况下（两种情况指 C++内建类型和没有代码的类型），我们都不能往目标类中加入构造函数，因此我们只能在我们拥有代码的类中把转换函数定义成类的一部分。

事实上，在我们写的每一个隐式地检测 istream 的值的循环中，都调用了这类转换操作函数。就像我们在 § 3.1.1 中所讨论的那样，我们可以在满足下面的条件的地方使用一个 istream 对象。

```
if( cin >> x ){ /* ... */ }
```

上式等效于下面的语句

```
cin >> x;
if(cin) { /* ... */ }
```

现在我们就能够理解在上面的表达式后面发生了些什么事了。

我们知道，if 语句对一个条件语句进行判断，该语句生成一个值。准确来说这个值是 bool 类型的值。在使用其他任何一个数学类型或者指针类型的值时都会先把它转换成 bool 类型的值，所以我们可以在一个判断表达式中使用这些类型的值。当然，iostream 既不是指针也不是数学类型，但是，在标准库中定义了一个从 iostream 类型到 **void***类型的转换，而 void*是一个指向 void 类型的指针。标准库中定义了 istream::operator void*，它通过检验不同的状态标志来判断 istream 是否有效，并返回 0 或者一个自定义的非零 void*值以表示流的状态。

在前面我们还没有使用过 void*类型。我们在 § 6.2.2 中讲到，void 类型有几种用途：最基本的用途是作为一个指针指向的类型。一个指向 void 的指针有时候又叫做通用指针。因为这是一个可以指向任何类型对象的指针。当然，你不能对这个指针间接引用，因为它指向的对象的类型还是未知的。不过我们可以把一个 void*类型转换成 bool 类型，bool 类型正是在上面的程序中用到的类型。

istream 类被定义成 operator void*而不是 operator bool，这是为了让编译器可以检测下面的错误用法：

```
int x;
cin << x;      //原本是应该写成 cin >> x 的;
```

如果 istream 类被定义成 operator bool，那么这个表达式将调用 istream::operator bool，把 cin 转换成 bool，然后把生成的 bool 值转换成 int 类型的值，把该值向左移动 x 位，然后舍弃这个结果！通过定义一个从其他类型向 void*类型（而不是一个简单的数学类型）的转换，标准库允许把一个 istream 类型用作一个判断条件表达式，但是不能把它作为一个算术值来使用。

12.6 类型转换与内存管理

很多系统都是用 C 语言或者汇编语言来写的，这些语言都用以空字符结尾的字符数组来装字符串数据，许多 C++程序都要和这些系统进行交互。就像我们在 § 10.5.2 中所说的那样，C++标准库本身在获得输入/输出文件名的时候就使用这个惯例。因为这个惯例，我们决定在 Str 类中提供一个从 Str 类型到空字符结尾的字符数组类型的转换。这样就可以让我们的用户把 Str 传递给一个以空字符结尾的字符数组类型为参数的函数。不幸的是，在下面我们将看到，这么做充满了内存管理上的缺陷。

假定我们想提供一个从 Str 到 char*类型的转换，那么我们可能会想同时提供这个转换的常量版本与非常量版本：

```
class STr{
public:
    //看似有理，但实际上问题多多的类型转换操作
    operator char*();                    //新加部分
    operator const char*() const;        //新加部分
    //其他部分同前
private:
    Vec<char> data;
};
```

像上面那样重写 Str 类之后，Str 的用户就可能写出下面的代码

```
Str s;
// ...
ifstream in(s);  //用户希望把 s 转换成 in 类型，然后打开这个名为 s 的流
```

上面的代码几乎不可能正确地完成转换。显然，我们需要一个字符数组类型的 data，而程序提供的是一个错误的 Vec<char>的类型，所以我们不可能仅仅返回 data。退一步说，即使类型匹配了，返回的 data 变量仍然会违反 Str 类的封装性：一个希望获得指向 data 的指针的用户有可能会使用该指针改变 string 的值。同样糟糕的事发生在 Str 类型对象被删除的时候。如果用户在 Str 对象被删除之后仍试图使用该指针，那么该指针将指向一片已经释放还给系统的内存，这是个无效的指针。

我们可以通过只提供一个向 const char*的转换来解决这个封装性的问题，但是这么做并不能防止用户在删除了 Str 类型对象之后继续使用指向 data 的指针。为了解决这后面一个问题，我们可以分配一块新的内存空间，然后将 data 中的字符复制到这块内存中，然后返回一个指向新分配内存空间的指针。用户今后就只能对这块新内存进行操作，在不需要的时候把内容删除并释放空间。

事实证明，这种方法也不能使程序正常运行。因为转换可能是隐式地发生的，这时候用户就没有指针可以删除了！再来看看下面的程序

```
Str s;
ifstream is(s);        //隐式的转换——用户没有办法释放数组!
```

如果 Str 类有相应的转换操作函数，那么在把 s 传递给 ifstream 的构造函数的时候，我们将隐式地把 Str 类型的变量转换成构造函数要求的 const char*类型参数。这一个转换分配新的内存空间来储存 s 变量的值的一个复件。但是这么做不会返回一个指向该新分配内存的一个显式的指针，那么用户就不能释放这片内存。显然，一个会导致内存发生泄漏的设计是不完善的。

在设计一个类的时候，我们不希望在用户写一些无害的代码的时候也会遇到麻烦。在 C++ 标准库写完之前，很多的库函数的卖主都提供几种不同的 strings 类。有些版本也会提供一个向字符数组类型的类型转换，但是编译器同时会在用户写出这些隐藏的错误的时候亮显出错误的代码行。

标准库的 string 类用的是另一种方法：该类允许用户获得一个储存在字符数组中的字符串的一个复件，但是只允许用户显式地这么做。标准库中的 string 类提供了三个成员函数来从一个 string 类型对象中获得一个字符数组。第一个函数是 c_str()，它把 string 类型对象的内容复制到一个空字符结尾的字符数组中。这个数组属 string 类的对象所有，用户不能删除指向它的指针。在这个数组中的数据只是临时的，在下一次调用一个可以改变 string 的成员函数的时候它就会失效。c_str()函数要求用户只能短暂地使用返回的指针或者把数据复制到一块指定的内存中去。第二个函数是 data()，除了返回一个不是以空字符结尾的字符数组以外，其他都与 c_str()函数一样。第三个函数是 copy 函数，它带有一个类型为 char*和一个类型为 int 的的参数，用来把 int 类型参数指定个数的字符复制到 char*类型参数的内存中，这片内存必须由用户来分配和释放空间。我们把这三个函数的实现留作习题给读者练习。

注意到 c_str 函数和 data 函数都具有隐式地向 const char*类型的转换的缺陷。不过在另一方面，因为它要求用户只能显式地调用这个类型转换，所以用户一定要理解他们正在调用的函数。他们知道在获得一个指针的一个复件的时候会带来些什么隐患。如果标准库允许用户进行隐式的类型转换，那么就可能轻易地使用户陷进这些问题中去。他们甚至可能都不知道自己在程序中隐式地调用了一个类型转换，当然就更不可能理解为什么这么做会导致程序出错。

12.7 小结

类型转换可以被定义成一个带单个参数的非 explicit 的构造函数，也可以被定义成 operator type_name()形式的转换运算符，其中 type_name 是该类型的对象要被转换成的目标类型的类型名。转换运算符必须是类的成员。如果两个类之间彼此定义了向另一个类的转换操作，会导致转换时的二义性的出现。

友员函数的声明可以在类定义的任何地方进行，C++语法规定一个类的友元函数可以访问类中的私有成员。

```
template<class T>
class Thing{
    friend std::istream& operator>>(std::istream&,Thing&);
    // ...
};
```

在 § 13.4.2 中我们将看到，类也可以被声明为友员类。

作为类成员的模板函数：一个类可以把模板函数作为成员函数。这种类本身可以是模板类，也可以不是模板类。一个拥有模板成员函数的类相当于拥有许多同名的成员函数。类的模板成员函数的声明方法与定义方法与其他的模板函数一样。

string 类中的操作

s.c_str()	生成一个指向一个空字符结尾的字符数组的 const char*类型的值。数组中的值只短暂地存在，在下一个可能改变 s 的值的操作发生的时候就会失效。用户不能删除该数组的指针，也不应该使用该指针的一个复件，因为该指针指向的内容只是暂时存在而已。
s.data()	类似于 s.c_str()函数，不同之处在于数组不以空字符结尾。
s.copy(p,n)	从 s 中复制 n 个字符到指针 p 指向的内存块中。用户必须保证 p 指针指向的内存足够大以装下 n 个字符。

习题

12-0 编译、运行并测试本章举的例程。

12-1 重写 Str 类，要求新的 Str 类可以自己管理内存。例如可以保存一个字符数组和一个长度。在编程的时候要注意这一改变导致的对复制控制的要求。还要考虑到在使用 Vec 时候的资源消耗情况（例如在内存资源上的花费）。

12-2 实现 c_str、data 和 copy 函数。

12-3 为 Str 类定义一个相关运算符函数。提示：在<cstring>头文件中定义了一个名为 strcmp 的函数，该函数对两个字符指针进行比较。如果第一个指针指向以空字符结尾的字符数组比第二个指针指向的数组小，函数返回一个负整数，如果两个数组一样大，函数返回

果第一个数组比第二个数组大，则返回一个正整数。

12-4 为 Str 类与一个等号运算符函数和一个不等号运算符函数。

12-5 写一个用于 Str 类型对象与字符串常量串连的函数，使得操作不再依赖于从 const char*类型到 Str 类型的转换函数。

12-6 为 Str 写一个操作函数，以使用户可以隐式地使用一个 Str 对象做为条件表达式。要求在 Str 为空的时候表达式值为假（FALSE），否则表达式值为真（TRUE）。

12-7 标准的 string 类提供一个随机访问迭代器来对字符串的字符进行操作。为 Str 类添加迭代器与迭代器操作 begin 与 end 函数。

12-8 为 Str 类添加 getline 函数。

12-9 使用 ostream_iterator 类来重写 Str 类的输出运算符函数。想想为什么不用 istream_iterator 类来重写该函数？

12-10 在 § 12.1 中我们学习了如何在 Str 中定义一个带有两个迭代器参数的构造函数，这种构造函数在 Vec 类中也很有用。试一试往 Vec 类中加入这种构造函数，然后不要调用 copy 函数，而是调用 Vec 的构造函数来重写 Str 类。

12-11 如果你往 Str 类中添加了本练习中提出的操作函数，你就可以在本书中的所有例程用这个 Str 类。用新的 Str 类重写第 5 章的字符图形操作函数，§ 5.6 和 § 6.1.1 的 split 函数。

12-12 为 Vec 类与 Str 类分别定义带两个迭代器参数的 insert 函数。

12-13 提供一个 assign 函数，用来把一个数组的值赋给一个 Vec 类型对象。

12-14 写一个程序用一个 string 变量初始化一个 Vec 类型对象。

12-15 § 4.1.3 中的 read_hw 函数从一个流中读取并检验每个字符，以判断函数是否遇到流的结尾，或者遇到一个无效的输入。我们的 Str 类中没有相应的操作，为什么？Str 类可能会使输入流处于无效状态吗？

第 13 章

使用继承与动态绑定

前面几章里我们已经探讨过如何建立自己的数据类型。这一技巧是面向对象编程（OOP）的基本技巧之一。本章将看看其他的 OOP 关键技术——继承和动态绑定。

在第 9 章里，我们编写了一个简单的类以封装为解决第 4 章出现的成绩问题而写的操作函数。本章将回过头来看看这些问题。这次我们假设在类的要求中有一个改动：有些学生可以选修本科学分课程，而另有一些学生可以选修研究生学分的课程。选取修研究生学分的课程要求学生做些额外的工作。假定研究生除了要完成家庭作业和参加考试以外，还要写一篇论文。我们在下面将看到，这一要求上的改变将导致一个面向对象的解决方案，通过这个方案我们来进一步探讨 C++提供的支持面向对象的特性。

我们的目的是写出一个新的类以达到这些要求。我们当然也希望在 § 9.6 中所写的那些函数继续工作。也就是说，我们希望这个新的类可以使用以前写的代码来读取一个记录分数的文件，写出一个一定格式的报告，为我们生成一个最终的成绩报表。

13.1 一个简单的 string 类

在我们的成绩问题中，一个研究生学分成绩的记录与一个本科生的学分成绩记录基本上是一样的，惟一不同之处在于研究生的学分成绩记录多了一个与论文相关的属性。这种关系最自然的解决办法正是使用**继承**（**inheritance**）。继承是 OOP 的基石之一。在一个类与另一个类比较起来，除了一些扩充以外其余部分完全相同的情况下，我们考虑使用继承特性。这里我们定义这样两个类：一个类集成了一些核心的公共的操作；另一个类增加了与研究生学分成绩相关的操作。

基本上我们已经知道如何实现第一个类了：它有点像以前写的 Student_info 类，在这里我们把它改名为 Core 类，这样改名的原因在 § 13.4 中自然就会明白了。现在，我们只需要知道 Core 类不再适用于所有的学生，它只适用于在课程选择上适用于核心要求的学生。我们希望保留 Student_info 来表示所有学生的信息。除了把类名改为 Core 以外，我们还向其中加入了一个私有类成员函数，用来读取学生记录中所有学生公共部分的数据：

```
class Core{
public:
    Core();
    Core(std::istream&);
    std::string name() const;
    std::istream& read(std::istream&);
    double grade() const;
private:
    std::istream& read_common(std::istream&);
    std::string n;
    double midterm,final;
    std::vector<double> homework;
};
```

Grad 类将满足获得研究生学位的一些额外的要求：

```
class Grad:public core{
public:
    Grad();
    Grad(std::istream&);
    double grade() const;
    std::istream& read(std::istream&);
private:
    double thesis;
};
```

上面的代码中定义了一个新的名为 Grad 的类型，它是从 Core 类中**派生**（或者说**继承**）出来的，或者说，Core 类是 Grad 类的一个**基类**。因为 Grad 类从 Core 类中派生出来，所以 Core 类中的每个成员（构造函数、赋值运算符函数与析构函数除外）也是 Grad 的成员。Grad 类可以加入自己的成员，在上面的代码中我们就向其中增加了新的 thesis 成员数据与新的构造函数。也可以在 Grad 类中重定义基类中的成员，在上面的代码中我们重定义了 grade 函数与 read 函数。但是在派生类中不能删除任何基类成员。

在上面的第一行代码中使用到了 public 关键字，它表示 Grad 类从 Core 类派生而来的是 Grad 接口的一部分，而不属于它的实现部分。也就是说，Grad 把 Core 的公有接口继承过来作为 Grad 公有接口的一部分。Core 类中的公有成员同时也是 Grad 类的公有成员。例如，如果我们有一个 Grad 类的对象，那么虽然在 Grad 类中我们没有定义 name 函数，但仍可以调用 name 成员函数来得到学生的名字。

Grad 类可以得到学生的论文成绩，并且使用一种新的算法来计算学生的最终成绩，这一点与 Core 类不同。因此 Grad 类型对象有五个成员数据。其中四个是从 Core 类中继承而来；第五个成员数据则是 double 类型的 thesis 变量。Grad 类中有两个构造函数和四个成员函数，其中两个成员函数（name 函数与 read_common 函数）由 Core 类继承而来，并在 Grad 类中重新进行了定义。

13.1.1 回顾保护类型

上面的代码表明，Core 类中的所有四个成员数据以及 read_common 成员函数都不能被 Grad 类中的成员函数访问。我们说 Core 类中的成员数据是私有的。只有这个类本身与它的友员可以访问该类的私有成员。不幸的是，要实现 Grad 类中的 grade 函数与 read 函数，我们就必须能够访问 Core 类中的这些私有成员。为了解决这个问题，我们使用 protecedt 关键字重写了上面的 Core 类：

```
class Core{
public:
    Core();
    Core(std::istream&);
    std::string name() const;
    double grade() const;
    std::istream& read(std::istream&);
protected:
    std::Istream& read_common(std::istream&);
    double midterm,final;
    std::vector<double> homework;
private:
    std::string n;
};
```

n 仍然是私有的，但是现在 read_common 函数与 midterm、final 和 homework 成员数据都变成了**保护**类型（**protected**）的数据。protected 关键字给 Grad 这样的派生类赋予了访问基类中的保护成员的权力，同时又保持使这些成员不能被类的其他使用者访问。

因为 n 是 Core 类的私有成员数据，只有 Core 类的成员函数与它的友员函数可以访问 n。而 Grad 类没有办法访问 n：它只能通过 Core 的公有成员函数来访问 n。

read、name 与 grade 函数都是 Core 类的公有成员函数，因此它们可以被 Core 类的所有使用者调用——当然也能被 Core 的派生类调用。

13.1.2 操作函数

为了完成我们的类，我们还需要写出四个构造函数，它们分别为 Core 类与 Grad 类的两个构造函数：默认构造函数与带一个 istream 类型参数的构造函数。我们还需要完成六个操作函数：Core 类中的 name 函数与 read_common 函数，还有两个类中都有的 read 函数与 grade 函数。在 § 13.1.3 中我们再来看如何实现构造函数。

在继续写代码之前先来想一下学生成绩的数据结构是怎样的。像前面一样，我们希望这种数据结构能储存不同数量的家庭作业成绩，所以家庭作业成绩必须来自每个记录的结尾处。现在，我们先来假定每条记录至少由一个学生的姓名、期中考试成绩和期末考试成绩组成。如果这是一个本科生的记录，那么后面马上再加上家庭作业的成绩。如果这是一个研究生

就先在期末考试成绩后面加上论文成绩，再在后面加上家庭作业成绩。

由上面的分析我们可以写出 Core 类中的操作函数如下：

```
string Core::name() const {return n; }
double Core::grade() const
{
    return ::grade(midterm,final,homework);
}
istream& Core::read_common(istream& in)
{
    //读出学生的姓名与考试成绩并储存起来
    in>>n>midterm>>final;
    return in;
}
istream& Core::read(istream& in)
{
    read_common(in);
    read_hw(in,homework);
    return in;
}
```

Grad::read 函数体与上面的 core::read 函数基本上相同，只是在调用 read_hw 函数之前先进行了读取 thesis 数据的操作：

```
istream& Grad::read(istream& in)
{
    read_common(in)
    in>>thesis;
    read_hw(in,homework);
    return in;
}
```

注意到在 Grad::read 函数的定义中，我们无需声明就可以调用基类中的成员函数与成员数据，因为这些 Core 类的成员也是 Grad 类的成员。要想显式地表明这些成员从 Core 类中继承而来，可以在代码中使用范围运算符：

```
istream& Grad::read(istream& in)
{
    Core::read_common(in)
    in>>thesis;
    read_hw(in,Core::homework);
    return in;
}
```

当然，thesis 变量有点不同，因为它是 Grad 的一部分，而不是 Core 的成员数据，所以我们可以写成 Grad::theis，但是却不能写成 Core::thesis。

grade 函数做了些改变，用来计算论文成绩（thesis）的影响。该函数比较论文成绩与考试和家庭作业计算得到的成绩，返回低一点的那个成绩：

```
double Grad::grade() const
{
    return min(Core::grade(),thesis);
}
```

这里我们调用的是基类中的 grade 函数，因为要计算的是不包括论文分数的成绩。在这种情况下，范围运算符就是必不可少的了，如果我们写的是

```
return min(grade(),thesis);
```

那么实际调用的将是 Grad 类的 grade 函数，那将导致灾难的发生。

在上面我们使用<algorithm>头文件中定义的 min 函数来决定返回哪一个成绩。min 的工作原理与 max 有点相似，只不过它返回的是两个参数中较小的那一个。和 max 函数一样，min 函数的两个参数也必须是同一类型的数据。

13.1.3 继承与构造函数

在写 Core 类与 Grad 类的构造函数之前，我们必须先理解编译器是如何生成一个派生类的对象的。对于所有的自定义类型，编译器都要先为对象分配内存空间。接下来调用一个合适的构造函数初始化对象。如果这个对象是派生类类型的，那么编译器将在构造对象的过程中另外增加一个步骤以构造对象的基类部分数据。派生类型对象在构造的时候经过以下步骤：

- 为整个对象分配内存空间（包括基类中与派生类中定义的数据）
- 调用基类的构造函数以初始化对象中的基类部分数据
- 用构造初始化器对对象的派生类部分数据进行初始化
- 如果有的话，执行派生类构造函数的函数体

惟一的一个新内容是如何选择调用基类中的哪一个构造函数。毫不奇怪，我们用构造初始化器来指定想要调用的基类构造函数。在派生类的构造初始化器中使用它的基类名，并在基类名后面附上一个参数列表（可以为空）。这些参数是用来构造对象中基类部分的初始值；同时编译器根据参数的个数与类型来选择调用基类中的哪一个构造函数。如果初始化的时候没有指定调用基类中的哪一个构造函数，编译器将调用基类默认构造函数来构造对象的基类部分。

```
class Core{
public:
    //Core 类的默认构造函数
    Core():midterm(0),final(0) { }
    //用一个 istream 类型变量来构造一个 Core 对象
    Core(std::istream& is) { read(is); }
    // ...
};
```

```
class Grad:public Core{
public:
    //两个构造函数都隐式地调用Core::Core()函数来初始化对象中的基类部分
    Grad():thesis(0) { }
    Grad(std::istream& is) { read(is);}
    // ...
};
```

Core 类的构造函数与前面§9.5.1 和§9.5.2 中的一样：它们用来直接生成一个 Core 类型对象或者用一个 istream 变量来生成一个 Core 类型对象。Grad 类的构造函数也同样是用这些值来生成 Grad 类型对象的，也就是说，直接生成对象或者用一个 istream&类型的变量生成 Grad 类型对象。值得注意的是，C++没有要求派生类构造函数一定要带与基类构造函数一样的参数类型。

Grad 类的默认构造函数是用来直接生成一个 Grad 类型对象的，它先构造 Core 基类的部分，并且把 thesis 设为 0。绝大部分的工作都是隐式地进行的：因为构造初始化代码是空的，所以编译器隐式地调用 Core 类的默认构造函数对 midterm、final、homework 和 name 成员数据进行初始化。通过同样的方式，Core 类的默认构造函数调用 name 与 homework 的默认构造函数对这两个成员数据进行初始化，只对 midterm 和 final 显式地初始化。惟一显式的操作是 Grad 的默认构造函数对 thesis 成员数据进行初始化。除了这些工作，Grad 的构造函数并没有其他工作要做，所以它的函数体是空的。

用一个 istream 类型的变量来生成一个 Grad 类型对象的过程与用一个 istream 类型变量生成一个 Core 类型对象的过程是一样的——都是通过调用 read 成员函数来实现。但是在调用 read 函数之前，我们要先（隐式地）调用基类的默认构造函数来初始化对象的基类部分。然后用 Grad::read（因为这个构造函数是 Grad 类的一个成员）来调用 read 函数。我们不必为初始化 thesis 变量而操心，因为 read 函数会从 is 变量中读进一个值赋给 thesis。

理解一个派生类型对象怎样被构造出来十分重要。请看下面的这个语句

```
Grad g;
```

执行这个语句会告诉系统为 Grad 类型对象 g 分配可以储存五个数据元素的内存空间，然后运行 Core 类的默认构造函数对 g 对象的 Core 类部分数据成员进行初始化，最后调用 Grad 的默认构造函数。类似的，在执行下面的语句时

```
Grad g(cin);
```

系统先为 Grad 类型对象 g 分配合适的内存空间，然后运行 Core 类的默认构造函数，接下来调用 Grad::Grad(istream&)构造函数为 name、midterm、final 和 homework 成员数据进行初始化。

13.2 多态和虚拟函数

到现在为止，我们并没有颠覆 Student_info 类的抽象模型。这个抽象模型中有一个

函数，它是该模型接口的一部分：它使用 compare 函数对两个学生记录进行比较。sort 函数用来把学生记录按照字母顺序进行排列，在 sort 函数中调用了 compare 函数，

新的比较函数与 § 9.3.1 中的 compare 函数基本上相同，只是在类型名上有一点改变：

```
bool compare(const Core& c1,const Core& c2)
{
    return c1.name()<c2.name();
}
```

我们通过对比学生记录的学生姓名来比较两条学生记录。现在不要去管 string 类中定义的 <运算符是怎样工作的。我们关心的是用来比较两个记录的那部分代码，它可以对两个 Core 类型记录或者两个 Grad 类型记录进行比较，甚至可以对一个 Core 类型记录与一个 Grad 类型记录进行比较：

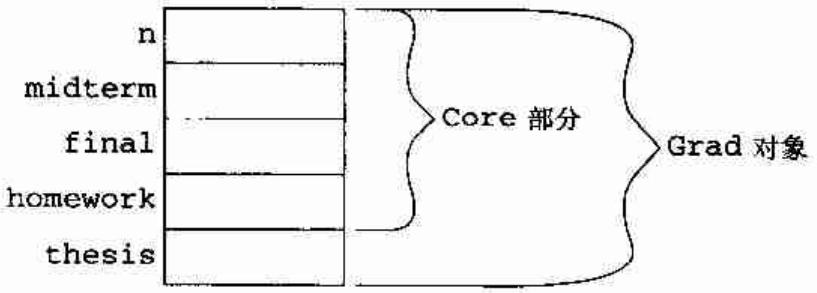
```
Grad g(cin);        //读一个 Grad 记录
Grad g2(cin);       //读一个 Grad 记录

Core c(cin);        //读一个 Core 记录
Core c2(cin);       //读一个 Core 记录

compare(g,g2);      //对两个 Grad 记录进行比较
compare(c,c2);      //对两个 Core 记录进行比较
compare(g,c);       //对一个 Grad 记录和一个 Core 记录进行比较
```

在上面对 compare 函数的调用中，程序会自动调用 Core 类的 name 函数来判断 compare 函数的返回值。显然，对 Core 类来说，这个调用是正确的，那么对于 Grad 类来说又如何呢？在定义 Grad 类的时候，我们说过它是从 Core 类派生而来的，而且在 Grad 中没有另外定义 name 函数。因此，在我们激活 g.name() 函数的时候，实现上调用的是从 Core 类中继承而来的 name 函数。该函数就像在 Core 类中那样被调用：它从对象的 Core 基类部分获得 n 的值。

我们可以向一个要求 Core&类型参数的函数传递一个 Grad 类的对象作为参数，这是因为 Grad 类是由 Core 类派生而来，每个 Grad 类型对象都有一个 Core 类的部分，如下图所示：



因为每个 Grad 类型对象都有一个 Core 类的部分，所以 compare 函数的引用参数可以是 Grad 类型对象中的 Core 类部分，也可以是一个平常的 Core 类型对象。类似地，我们

一个指向 Core 类型的指针或者一个 Core 类型对象本身（与 Core 类型对象的一个引用参照形成对比）作为 compare 函数的参数。无论是哪种情况，我们都可以以 Grad 类型对象的名义调用 compare 函数。如果函数以指针作为参数，可以将指向 Grad 的指针传给它，编译器会将 Grad* 转换成 Core*，并将指针绑定到 Grad 对象中的 Core 部分。如果该函数以一个 Core 类型对象为参数，那么实际传递过去的只是对象的 Core 类部分。在下面我们将看到，向函数传递一个对象本身作为参数，与传递对象的一个引用作为参数有着很大的差别。

13.2.1 在不知道对象类型的情况下获得对象的值

在我们用一个 Grad 类型对象作为参数调用 compare 函数的时候，compare 参数能正常工作，因为 name 函数是由 Grad 类与 Core 类共享的。如果我们在比较两个学生的时候不是比较他们的姓名，而是想比较它们的期末考试成绩的时候，情况又会怎样呢？例如，现在不是要生成一个按学生姓名排序的期末考试成绩的报表，而是想生成一个根据期末考试成绩进行排序的报表。

要解决该问题，我们先写出一个类似于 compare 的函数：

```
bool compare_grades(const Core& c1, const Core& c2)
{
    return c1.grade()<c2.grade();
}
```

该函数与 compare 函数的惟一不同之处在于，我们调用的是 grade 函数而不是 name 函数。这一区别的意义是十分重大的！

这其中的差别在于，Grad 类中重定义了 grade 函数，而且我们没有写任何代码来区分这两个版本的 grade 函数。在执行 compare_grades 函数的时候将调用 Core:: grade 成员函数，这就像在 compare 函数中调用 Core::name 函数一样。此时，如果比较的是 Grad 类型对象，调用 Core 版本的函数将得到错误的结果，因为 Core 类和 Grad 类中的 grade 函数是不同的。对于 Grad 类型对象，考虑到 thesis 的因素，我们只能调用 Grad::grade。

我们需要为 compare_grades 函数调用正确的 grade 函数，这要依赖于传递给函数的实际参数类型：如果 c1 和 c2 指向 Grad 类型对象，那么将调用 Grad 类中的 grade 函数；如果 c1 和 c2 指向 Core 类型对象，那么将调用 Core 类的 grade 函数。我们希望在运行时进行判断后再做出决定。也就是说，我们希望系统根据实际传递给函数的参数的类型来运行正确版本的 grade 函数，而参数的类型只有到运行的时候才是已知的。

为了支持这种运行时选择，C++提供了**虚拟（virtual）**函数：

```
class Core{
public:
    virtual double grade() const;     //增加虚拟函数
    //...
};
```

现在，grade 是一个虚拟函数。如果调用 compare_grades 函数，程序在执行时将由参数 c1 和 c2 的实际类型来决定调用哪个 grade 函数。如果参数是一个 Grad 类型对象，就调用 Grad::grade 函数，如果参数是一个 Core 类型对象，就调用 Core::grade 函数。

virtual 关键字只能在类的定义里被使用。如果函数体在声明之外单独定义，我们不能在定义的时候重复 virtual 关键字。因此 Core::grade()这个定义不需要改变。类似地，如果类中一个函数是虚拟的，那么在派生类中它的虚拟特性也会被继承，所以在 Grad 类中对 grade 函数的声明不需要再重复 virtual 关键字。现在我们必须把新的类定义代码进行编译，一旦我们这样做了，由于基类中的函数被 virtual 所修饰，我们就可以得到我们所期望的行为。

13.2.2 动态绑定

只有在以引用或者指针为参数调用虚拟函数的时候，它的运行时选择特性才会有意义。如果我们以对象（与通过引用或指针相对比）的名义调用一个虚拟函数，那我们就可以在编译的时候知道对象的类型，对象的类型一旦确定了，即使在运行的时候也不会改变。相反地，一个指向基类型对象的引用或指针可能是确实是指向一个基类型对象，也可能是指向该基类派生出来的类的对象，也就是说，引用或者指针指向的对象的实际类型在运行的时候可以是变化的。正是在这种情况下，虚拟函数的机制与非虚拟函数的机制之间产生了一些区别。

例如，假设我们重写 compare_grades 函数如下：

```
//程序执行出现错误!
bool compare_grades(Core c1, Core c2)
{
    return c1.grade()<c2.grade();
}
```

在这个版本的 compare_grades 函数中，参数的类型是对象，而不是对象的引用。在这种情况下，我们知道对象的类型由 c1 和 c2 决定：它们都是 Core 类型对象。我们还能以 Grad 类型对象的名字调用这个函数，不过参数类型是 Grad 类这一事实没有什么实质的意义。因为这时候真正传递给函数的只是对象的基类部分的数据。这个 Grad 类型对象将被删减得只剩下 Core 类部分，然后将这一个部分的一个复件传递给 compare_grades 函数。因为参数仍然是 Core 类型的对象，所以这种对 grade 类型对象的函数调用是**静态**绑定的——它们在编译的时候就已经被确定了——调用 Core::grade 函数。

理解**动态**绑定与静态绑定的区别是理解 C++对 OOP 的支持的基础之一。动态绑定（dynamic binding）顾名思义，就是指在运行的时候才决定调用什么函数，而不是在编译的时候就决定下来，那种情况属于静态绑定。如果以一个对象的名义调用一个虚拟函数，这个调用就是静态绑定的——也就是说，它将在编译的时候就被绑定好——因为这个对象不可能在运行的时候改变为与编译时不同的类型。相反地，如果通过一个指针或者一个引用来调用虚拟函数，那么函数将是动态绑定的——也就是说在运行的时候被绑定。在运行的时候，通过引用或者指

针实际指向的对象类型来决定调用虚拟函数的具体版本：

```
Core c;
Grad g;
Core* p;
Core& r = g;

c.grade();          //对 Core::grade()函数进行静态绑定
g.grade();         //对 Core::grade()函数进行静态绑定
p->grade();        //根据 p 所指对象的类型进行动态绑定
r.grade();          //根据 r 所引对象的类型进行动态绑定
```

前面的两个函数调用都是进行静态绑定的：我们知道 c 是一个 Core 类型对象，而在运行的时候，c 仍然是一个 Core 类型对象，所以编译器可以静态地解决这个函数调用。而在第三和第四个函数调用中，我们不知道 p 或 r 指向的对象的类型：它们可能是 Core 类型对象，也可能是 Grad 类型对象，这种情况下，只有在程序运行的时候才能根据 p 或 r 实际所指的对象类型来决定调用哪个函数。

在要求一个指向基类对象的指针或引用的地方，我们却可以用一个指向派生类的指针或引用来代替，这是 OOP 中的一个关键概念：**多态性（polymorphism）**。polymorphism 一词来自希腊词 polymorphos，意思是“具有多种形式”，最早在 19 世纪中叶开始出现在英语中。在程序设计中，它指的是用一个类型表示几种类型的能力。C++通过虚拟函数的动态绑定特性来支持多态性。在通过一个指针或者一个引用调用虚拟函数的时候，我们实际上就在进行一个多态的调用。引用（或者指针）参数的类型是固定的，但是参数所引用（或者所指）的对象的类型却可以是所引（或所指）对象的类型，或者是由该类派生出来的任何一个子类。因此，我们可以通过一种类型来调用许多函数中的一个。

关于虚拟函数还有一点值得注意：无论是否调用它们，在程序中都要对定义它们。对于非虚拟函数，如果程序中没有调用它，那么我们就可以在类中仅仅对它进行声明而不定义。但是如果在类中只对虚拟函数进行了声明而没有加以定义的话，很多编译器都会生成一些奇怪的错误信息。如果一个程序在编译的时候产生一个莫名奇妙的信息，而且那条信息说什么东西没有定义，那你最好检查一下是不是对所有的虚拟函数进行了定义。在大多数情况下，你会由此找到错误的根源。

13.2.3 简单回顾

在继续往下讲之前，有必要对前面的知识进行一个总结，并对前面所写的类做一个小小的改动：我们来把 read 函数也声明为一个虚拟函数。这是因为我们希望根据 read 函数的实际参数类型来决定调用哪一个 read 函数。经过这样的改动，现在的类如下所示：

```
class Core{
public:
    Core():midterm(0),final(0) { }
```

```
    Core(std::istream& is) { read(is); }

    std::string name() const;

    // 其余与§13.1.2中所定义的一样
    virtual std::istream& read(std::istream&);
    virtual double grade() const;
protected:
    //可以被所有派生类的成员访问
    std::istream& read_common(std::istream&);
    double midterm,final;
    std::vector<dobule> homework;
private:
    //只能被Core类的成员访问
    std::string n;
};
class Grad:public Core{
public:
    Grad():thesis(0) { }
    Grad(std::istream& is) { read(is);}
    //其余与§13.1.2中定义相同;
    //注意: grade函数与read函数也被继承为虚拟函数
    double grade() const;
    std::istream& read(std::istream&);
private:
    double thesis;
};
bool compare(const Core&, const Core&);
```

我们现在写了两个类来封装两种不同类型的学生。其中 Core 类适用于学习核心课程的学生，而另一个类 Grad 从 Core 类中派生出来，与 Core 类比较，它对学生增加了必须写一篇论文的要求。我们有两种方法可以生成 Core 类型对象或者 Grad 类型对象。默认构造函数用来生成一个被适当初始化的空对象；而另一个带 istream&类型参数的构造函数从一个指定的流中得到初始值。这种操作允许我们改变对象，重设对象的值，也可以让我们获得学生的姓名与最终的成绩。注意，在这个版本中，我们把 grade 与 read 函数都声明为虚拟的。最后，我们的接口部分包括一个全局的非成员函数 compare，它通过对比学生的姓名来比较两个对象。

13.3 用继承来解决我们的问题

现在的 Core 类与 Grade 类可以分别用来模拟不同的学生，我们打算用这些类来解决§9.6中的分数问题。在那个程序里，我们从一个储存有学生分数记录的文件中读取数据，然后为每个学生计算总成绩，最后按学生姓名进行排序把学生的成绩输出成一个报表。现在我们

决同样的问题，而且在读取的文件中包含两种不同的学生的成绩记录。

在解决整个问题之前，我们先来设法解决两个简单一点的问题：写两个程序分别读出完全由两种记录中的一种记录组成的文件。这两个程序除了声明类型以外与我们前面的程序完全相同：

```
int main()
{
    vector<Core> students;        //读取并处理文件中的 Core 记录
    Core record;
    string::size_type maxlen = 0;
    //读入并储存数据
    while (record.read(cin)) {
        maxlen = max(maxlen, record.name().size());
        students.push_back(record);
    }
    //对学生记录按字母排序
    sort(students.begin(),students.end(),compare);
    //输出学生姓名与成绩
    for(vector<Core>::size_type i=0; i!=students.size(); ++i) {
        cout<<setw(maxlen+1)<<students[i].name();
        try{
            double final_grade=students[i].grade();   //Core::grade
            streamsize prec=cout.precision();
            cout<<setprecision(3)<<final_grade
                <<setprecision(prec)<<endl;
        }catch(domain_error e) {
            cout<<e.what()<<endl;
        }
    }
    return 0;
}
```

我们可以仅仅改变定义的类型，然后写出上面类似的处理 Grad 类记录的程序：

```
int main()
{
    vector<Grad> students;          //与以前的 vector 类型不同
    Grad record;                    //与以前读的数据类型不同
    string::size_type maxlen=0;
    //读取并储存数据
    while(record.read(cin)) {       //从 Grad 类型对象而不是 Core 类型对象中读
        maxlen=max(maxlen,record.name().size());
        students.push_back(record);
    }
    //把学生记录按字母排序
    sort(students.begin(),students.end(),compare);
```

```
    //输出学生姓名与成绩
    for(vector<Grad>::size_type i=0; i!=students.size(); ++i) {
        cout<<setw(maxlen+1)<<students[i].name();
        try{
            double final_grade=students[i].grade(); //Grad::grade
            streamsize prec=cout.precision();
            cout<<setprecision(3)<<final_grade
                <<setprecision(prec)<<endl;
        }catch(domain_error e) {
            cout<<e.what()<<endl;
        }
    }
    return 0;
}
```

当然，运行中实际调用的函数依赖于 record 的实际类型和包含在向量中的对象的实际类型。例如，表达式

```
record.read(cin)
```

有可能调用 Core::read 函数或者 Grad::read 函数，这决定于 record 的具体类型。值得注意的是这个调用是静态绑定的：因为我们不需要在运行中才知道 record 的类型，我们是以对象的名义调用 read 函数的，而不是以指向一个对象的指针或者引用的名义调用它。因此，record 不是一个 Core 类型对象就是一个 Grad 类型对象，这由运行的程序版本决定。然而，一旦我们定义了 record，它的类型就固定下来了，所以对 record.read(cin)的调用在编译的时候就已经绑定好了。类似地，在生成输出报表的时候调用的 grade 函数

```
students[i].grade()
```

也将被静态绑定，在运行第一个程序的时候被指定调用 Core 类的 grade 函数，在运行第二个程序的时候被指定调用 Grad 类中的 grade 函数。

在这两个版本的程序中，name()函数的使用都指向 Core 类中定义的（非虚拟）版本。因为这个函数被 Grad 类继承，所以在用一个 Grad 类型对象调用 name 函数的时候，实际上运行的也是在 Core 类中定义的版本。我们传递给 sort 操作的 compare 函数是作用在 Core 类型对象上的。如果运行处理 Grad 类记录的程序，将只对 Grad 类型对象的 Core 类部分进行比较。

显然，分别写这么两个程序过于繁琐。实际上我们要写的是一个版本的程序，它既可以处理 Core 类的对象，又可以处理 Grad 类的对象。

为了写一个既可以读取 Core 类记录的文件又可以读取 Grad 类记录的文件，我们再来仔细地看看这些代码，然后找出在哪些地方需要特别留意记录的类型。为了写出程序的单一版本，我们必须把下面这些类型依赖性从程序中去掉：

- 对向量的定义（我们用向量来保存读出来的数据）

- 对局域的临时变量的定义（我们用这些临时变量暂存从记录中读出来的数据）
- read 函数
- grade 函数

这么一来剩下的代码都是与类型无关的（用向量来排序或者通过向量遍历每个记录的代码），或者对于两个版本都是一样的（例如 name 函数与 compare 函数）。如果我们把 grade 函数与 read 函数定义成虚拟的，就已经把后面两个问题解决掉了。

实际上，前面的两个子问题（临时变量用什么类型以及在容器中保存什么类型的数据）也可以用同样的办法解决。事实证明有两种方法可以用来解决这些问题。第一种方法简单明了，在下面的部分我们将讨论一下这种方法；另外一种方法中包含了 C++中的另一个重要的惯用法，我们留待 § 13.4 再作探讨。

13.3.1 容纳（实际上）未知类型的容器

我们要解决的这一问题是去掉下面这些定义的类型依赖性：

```
vector<Core> students;          //必须储存 Core 类型的对象，而不能储存多态的类型
Core record;                    //一个 Core 类型对象，而不能是 Core 类派生出来的类
```

这些代码很明显地依赖于类型的种类。在定义 record 的时候，我们明确地指定了 record 对象的类型：它必须是一个 Core 类的对象，这就是定义中包含的意义。类似地，在定义 students 的时候，我们也明确地指明，这是一个用来储存 Core 类型对象的容器。在 § 13.6.1 中我们还会谈到这种容器，不过在这里重要的是认识到，在定义一个 vector<Core>的时候，我们就声明了向量中的每个对象必须是一个 Core 类的对象——而不能是由 Core 类派生出来的子类的对象。

在我们找出程序中依赖于类型的代码的时候，注意到通过把 read 函数与 grade 函数声明为虚拟函数已经解决了一半的类型依赖性问题。如前所述，另一半的问题在于，我们的程序对这些虚拟函数实行静态的绑定。为了得到我们所要的动态绑定特性，我们通过一个指向 Core 的指针或引用来调用 read 函数和 grade 函数。这样，绑定的对象类型随着指针或引用的类型而变。经过前面的讨论，我们已经找到解决所有四个问题的办法：写一个控制指针而不是控制对象的程序。我们可以定义 vector<Core*>，然后把 record 也定义成一个指针。这样我们就在程序中实现了动态绑定，同时消除了在向量和局域临时变量的定义中存在的类型依赖性。不幸的是，在下面我们将看到，这种解决方案给用户带来了太多的麻烦。例如，下面这么一段想当然的改进代码根本不能正常运行：

```
int main()
{
    vector<Core*> studnets;

    Core* record;
```

```
    while (record->read(cin)) {     // 出错!
        // ...
    }
}
```

这段程序将产生可怕的错误，因为我们还没有让 record 指向任何一个实际的对象！

我们可以解决这个问题，但只有一种方法，那就是要求用户亲自管理好从文件中读出的数据所占用的内存。我们的用户还不得不检测程序正在读的记录的类型。我们假定每个记录中都包含一个标志，用这个标志可以区分记录中包含的是什么类型的数据：研究生的记录以 G 字母打头，而本科生的记录以 U 字母打头。

在重写这个程序让它使用指针之前，还有一个问题必须解决：我们如何对一个包含指针的容器进行排序？答案十分简单，我们只需要一个带有两个指向 Core 类型对象的指针的新的比较函数。不过要注意不能把这个函数叫做 compare。回忆一下，在 § 8.1.3 中我们讨论过为一个传递给模板函数作为参数的值获得正确类型的微妙之处。由于同样的原因，我们不能把一个重载函数命名为一个模板参数。如果一定要这么做的话，编译器将无法决定调用函数的哪一个版本。最后我们决定写一个新的比较函数，这个函数的函数名是 compare_core_ptrs：

```
bool compare_core_ptrs(const Core* cp1,const Core* cp2)
{
    return compare(*cp1,*cp2);
}
```

写完这个专用的比较函数以后，我们就可以重写主函数了：

```
int main()
{
    vector<Core*> students;          //用来保存指针而不是对象
    Core* record;                    //临时变量也必须是一个指针
    char ch;
    string::size_type maxlen=0;
    //读出并储存数据
    while(cin>>ch){
        if(ch=='U')
            record=new Core;         //为一个 Core 类型对象分配内存
        else
            record=new Grad;         //为一个 Grad 类型对象分配内存
        record->read(cin);           //虚拟调用
        maxlen=max(maxlen,record->name().size());  //间接引用
        students.push_back(record);
    }
    //把以指针为参数的比较函数做为参数传递给排序函数
    sort(students.begin(),student.end(),compare_Core_ptrs);
    //输出学生的姓名与成绩
    for(vector<Core*>::size_type i=0;
```

```
        i!=students.size(); ++i) {
        //students[i]是一个指针，用来解除对函数的调用
        cout<<setw(maxlen+1)<<studnets[i]->name();
        try{
            double final_grade=students[i]=>grade();
            streamsize prec=cout.precision();
            cout<<setprecision(3)<<final_grade
                <<setprecision(prec)<<endl;

        }catch(domain_error e){
            cout<<e.what()<<endl;
        }
        delete students[i];        //释放在读文件的时候生成的临时变量
    }
    return 0;
}
```

请注意，在代码的注释中该代码与以前的代码之间的不同之处。这些改变都源于程序是对指针进行操作而不是对对象进行操作。

我们在 while 循环中进行了一些修改以从输入流中读取首字符，接下来检测该首字符并由此得到我们将要读取的记录的类型。一旦我们知道要读取的对象的类型，就可以为恰当的类型对象分配内存，然后用该对象储存从输入流中读出来的数据。因为 read 函数被定义成虚拟函数，所以在运行中根据 record 指向的对象类型来决定调用哪一个版本的 read 函数。注意，我们必须记得间接引用 record 这个用来访问 read 的指针。用来计算最长姓名的长度的函数也变成通过这个指针的间接引用来调用，不过下面几行代码保持不变。

在进行输出的循环中，我们要牢牢记住，studnet[i]生成一个指针。一旦获得了 students[i]，我们也就得到了一个必须本身被间接引用以得到其指向的对象的指针。在对 read 函数的调用中，因为 grade 函数的调用是一个虚拟调用，所以编译器会自动激活正确的 grade 函数版本以计算成绩。最后一个改变是记得要释放对象占用的内存空间，在这里我们通过对指向 students[i]对象的指针调用 delete 函数完成该操作。

13.3.2 虚拟析构函数

现在我们的程序基本上已经能正常工作了。现在只剩下一个问题，这个问题会在我们的输出循环中删除对象的时候发生。在为这些对象分配内存的时候，我们同时为 Grad 对象和 Core 对象分配了内存。但是我们把所有指向对象的指针都保存为 Core*类型指针，而不是 Grad*类型指针，即使在指针的确指向一个 Grad 类型对象的时候也是如此。当我们删除这个指针时，我们是在删除一个指向 Core 的指针，而不是指向 Grad 的指针（即便该指针指向的对象的类型确实是 Grad）。不过幸运的是，这一问题其实很好解决。

在对一个指针调用 delete 函数的时候发生了两件事情：第一件事是调用了指针所指

析构函数，第二件事是对象所占用的内存空间被释放回系统。在程序删除 students[i]中的指针的时候，指针有可能指向一个 Core 类型对象，也可能指向一个 Grad 类型对象。无论是 Core 类还是 Grad 类中都没有显式地定义一个析构函数，这意味着在调用 delete 函数的时候，将调用编译器自动合成的析构函数并释放对象占用的内存空间。合成的析构函数会调用类中每个数据元素的各自的析构函数。但是，在运行 delete 函数的时候，系统将调用哪个类的合成析构函数呢？而析构函数删除的是一个 Core 类型对象的成员数据呢，还是一个 Grad 类型对象的数据？另外，在释放内存的时候返回多大的内存空间呢——是一个 Core 类型对象大小的内存空间，还是一个 Grad 类型对象大小的内存空间？

这些问题看起来应该可以通过虚拟的机制来解决——事实上也确实如此。为了得到一个虚拟的析构函数，在我们的类中必须显式地定义一个析构函数，然后再把它定义成一个虚拟函数：

```
class Core{
public:
    virtual ~Core() {}
//其他部分同前
};
```

现在，如果我们执行 delete students[i]，系统将根据 students[i]实际上指向的对象的类型来决定调用哪个析构函数。类似地，释放返回系统的内存空间的大小也由 students[i]实际上指向的对象的类型来决定。

注意到析构函数的函数体是空的。删除一个 Core 类型对象的惟一工作是删除对象的成员数据，系统会为我们自动完成这项工作。一个空的虚拟析构函数其实并不少见。在所有通过一个指向基类的指针来删除一个派生类型对象的时候，都要用到虚拟析构函数。而如果除此之外没有什么理由要定义这个析构函数的话，这个析构函数就没有什么事要做，那它的函数体当然就是空的了。

我们没有必要为 Grad 类也增加一个析构函数。因为在基类 Core 中这是一个虚拟的析构函数，所以在包括 Grad 类在内的所有派生类中都继承了这个虚拟的析构函数。因为 Grad 类在删除对象的时候没有什么额外的工作要做，所以没有必要为这个派生类重写这个析构函数。因为在派生类中继承了基类中的析构函数的虚拟特性，现在我们要做的只不过是重新编译这个新程序。

13.4 一个简单的句柄类

在前面的讨论中我们进展十分顺利，不过在程序中仍然存在问题：为了管理指针，程序中引用了太多的复杂性，同时也引入了一些可能导致错误的缺陷。我们的用户必须记得在读记录的时候为这些记录分配内存空间，还得记住在不用的时候释放数据占用的内存空间。在上面的代码中经常要对一个指针间接引用以得到指针指向的对象。不过无论如何，我们终于写

个程序，它可以处理混有两种不同记录的文件。

接下来我们要做的是找到一种新的办法，使写出来的程序既可以保持简洁性（就像在区别处理 Core 类型对象与 Grad 类型对象的两个简单程序中那样），又不会产生这样那样的问题，当然，我们还希望这个新程序能够同时处理两种不同类型的记录。事实上存在这样的方法，利用一个普通的编程技巧——**句柄类（handle class）**，我们就可以达到目的。

在意识到我们要处理的对象的类型只有在运行时才是已知的时候，我们的代码便会变得十分混乱。我们知道，我们要处理的每一个对象要不就是一个 Core 类型对象，要不就是 Core 的派生类的对象。在我们的解决方案中用到指针，因为我们可以为一个 Core 类型的指针分配内存，然后使让该指针指向一个 Core 类型对象或者一个 Grad 类型对象。现在的问题是这样做给用户留下了一个可能导致错误的隐患。我们无法消除这个隐患，不过却可以写一个新的类封装这个指向 Core 类型对象的指针，这样就把隐患在用户面前藏了起来。

```
class Student_info{
public:
    //构造函数与复制控制
    Student_info():cp(0){}
    Student_infor(std::istream& is):cp(0){ read(is);}
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    ~Student_info(){ delete cp; }

    //操作
    std::istream& read(std::istream&);

    std::string name() const{
        if(cp) return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const{
        if(cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }
    static bool compare(const Student_info& s1,
                        const Student_info& s2) {
        return s1.name() < s2.name();
    }

private:
    Core* cp;
};
```

这里的主要思想是定义一个 Student_info 类，该类的对象既可以表示一个 Core 类型对象又可以表示一个 Grade 类型对象。从这个意义上来看，它就像一个指针一样。不过，Studer

的用户不用担心为 Student_info 对应的对象进行内存分配。因为这个类本身就能够处理程序中这些繁琐而又充满着错误隐患的工作。

每个 Student_info 类的对象中都有一个名为 cp 的指针，cp 指向一个 Core 类的对象或者指向一个 Core 类派生出来的类的对象。在 § 13.4.1 中将会看到，我们在 read 函数中为 cp 所指对象分配内存。因此，两个构造函数都把 cp 指针初始化为 0，表明 Student_info 对象尚未指定对象。在带有一个 istream 类型参数的构造函数中调用了 Student_info::read 函数。这个构造函数为适当的类型对象分配内存空间，然后从指定的 istream 中读取数据，对对象进行初始化。

从 § 11.3.6 中提到的“三位一体”知道，我们需要一个复制构造函数、一个赋值运算符函数和一个析构函数来管理这个指针。析构函数要做的工作十分简单：它只需要把构造函数为对象分配的内存释放还给系统就行了。因为在 § 13.3.2 中我们声明 Core 类的析构函数是虚拟的，所以 Student_info 的析构函数无论在删除一个 Grad 类型对象的时候还是在删除一个 Core 类型对象的时候，都可以正常工作。在 § 13.4.2 中我们将定义复制构造函数与赋值运算符函数。

因为用户可能会在程序中用到 Student_info 类的对象，而不是用 Core 类或者 Grad 类型对象，所以在 Student_infor 类中必须提供和 Core 类相同的接口。对于 name 函数与 grade 函数，Student_info 都没有特别的事要做，在对 Student_info 类调用 name 函数或者 grade 函数的时候，实际上调用的是 cp 所指的 Core 类型对象或者 Grad 类型对象的相应函数。

但是，cp 可能会是 0。在用户调用默认构造函数生成一个 Student_info 类型对象的时候 cp 的值就是 0，这时候不会从输入流中把数据读入对象。如果 cp 的值是 0，我们就无法将这些函数调用传递给底层的对象去处理。此时，我们将显示一个运行时错误，表示程序中出现了错误。

记住，Core::grade 函数是个虚拟函数，这一点十分重要，它表示当我们通过 cp 指针调用 grade 函数的时候，系统根据 cp 实际所指的对象类型来判断实际调用的是哪一个类的 grade 函数。例如在 cp 指向一个 Grad 类型对象的时候，在运行时将调用 Grad::grade 函数。

接口中的另一个函数是 compare 操作，这一操作有两个有趣的性质。第一，对于 Core 类来说，compare 是一个全局的非成员函数，因此，我们把它实现成一个**静态成员函数**。静态成员函数与一般成员函数不同，静态成员函数不能对类的对象进行操作。和其他的成员函数不同的是，静态成员函数与类关联，而不是与一个特定的类型对象关联。因此，静态成员函数不能访问类型对象的非静态数据成员：因为没有与该函数关联的对象，所以它不能访问任何成员。

静态成员函数具有一个重要的好处：它们的函数名带有它们所属的类的限定词。所以当我们在声明 compare 为一个静态成员函数的时候，就是在声明一个叫做 Student_info::compare 的函数。因为该函数有一个带限定词的名字，所以它不会重载我们用来比较 Core 类型对象的 compare 函数。因此，用户可以把 Student_infor::compare 作为参数传递给 sort 函数，而编译器会知道应该调用哪个函数。

compare 函数的另一个有趣的性质是关于它的运行的。这个函数调用 Student_info::name 函数来获得保存在记录中的学生姓名。仔细想一想这里面发生的事情。如果 cp 不为 0，对 Student_info::name 的调用实际上就是在调用 Core::name。而如果 cp 等于 0，那么这一

用将产生一个异常，并把这个异常传播给 compare 函数的调用者。因为 compare 函数用的是 Student_info 类的公有接口，所以该函数不需要直接检测 cp 是否为 0。像其他的用户代码一样，compare 函数会把问题转给 Student_info 类去做。

13.4.1 读取句柄

read 函数有三个任务：首先它必须释放该句柄指向对象（如果句柄指针不为 0）占用的空间；其次，它必须判断将要读的对象是属于什么类型；最后，它还要为正确类型的对象分配合适大小的内存空间，然后从 read 函数参数提供的输入流中读取数据对对象进行初始化：

```
istream& Student_info::read(istream& is)
{
    delete cp;          //如果有的话，删除以前所指的对象

    char ch;
    is>>ch;             //得到记录的类型

    if(ch=='U'){
        cp=new Core(is);
    } else {
        cp=new Grad(is);
    }

    return is;
}
```

这个 read 函数先为现存的句柄对象指向的对象（如果有的话）释放内存。我们不需要在调用 delete 之前先检测 cp 是否为 0，因为在 C++语言中删除一个零指针是无害的。释放了旧对象的内存之后，开始准备读取新值。先读取输入流的首字母并对其进行判断。根据这个首字母，程序可以得到记录的类型，从而可以生成相应类型的对象，为其分配合适大小的内存并运行正确构造函数对其初始化。这些构造函数调用自身的 read 函数，从输入流中读取数据，放进新生成的对象中。在对象被初始化之后，我们把指向对象的指针保存在 cp 中。最后，程序返回参数给出的数据流对象。

13.4.2 复制句柄对象

为了管理指向 Core 对象的指针，复制构造函数与赋值运算符函数是必需的。构造函数在调用 read 函数的时候被自动调用，用来为指针分配内存。在复制一个 Student_info 类型对象的时候，可能想为一个新的对象分配内存并用正在复制的对象的值来初始化新的对象。但是，这么做存在一个问题：那就是我们正在复制的对象是什么类型？没有一种现成的方法可以知道我们正在复制的 Student_info 类型对象是指向一个 Core 类型对象还是指向一个 Core 类派生

对象。

这里提供的解决该问题的办法是在 Core 类与它的派生类中声明一个虚拟函数。这个虚拟函数生成一个新的对象，用来储存原来那个对象的一个复件：

```
class Core{
    friend class Student_info;
protected:
    virtual Core* clone() const { return new Core(*this);}
    //其他部分同前
};
```

clone 函数以一种令人惊讶的简洁的风格按照我们想像的方式正常地工作。我们为一个新的 Core 类型对象分配内存，并且调用 Core 构造函数为新的对象进行正确的初始化。我们在以前的 Core 类中没有显式地定义一个复制构造函数。尽管如此，从 § 11.3.5 的讨论中我们知道：编译器会为我们自动合成一个默认的复制构造函数，这个函数把现存的 Core 类型对象中的每一个元素都复制到新建对象中。

clone 函数是我们为了实现特殊目的而写的一个函数，所以没有把它放在 Core 类的公共接口中。事实上 clone 函数是保护类型的接口，而且把 Student_info 类声明为 Core 类的友元类，这样 Student_info 类就可以调用 clone 函数了。友员类与在 § 12.3.2 中提到的友员函数类似。在那里我们学习了友员函数，它可以访问类的私有成员和保护成员。类似地，把一个类声明为另一个类的友员类可以使该类中的所有成员都成为另一个类的友员。也就是说，如果在 Core 类的定义中加上下面的声明：

```
friend class student_info;
```

那么 Student_info 中的所有成员函数都被声明为 Core 类的友员函数，它们都可以访问 Core 类的私有成员和保护成员。

在向基类中加入 clone 这个虚拟函数以后，一定要在其派生类中重定义这个函数，这样在以派生类为名义调用一个 clone 函数的时候，我们就会正确地复制出一个新的 Grad 类的对象：

```
class Grad{
protected:
    Grad* clone() const { return new Grad(*this);}
    //其他部分同前
};
```

对于 Core::clone 函数，我们在调用它的时候生成一个新的对象，并用*this 的内容对其初始化，但是在这里我们返回一个 Grad*类型对象而不是一个 Core 类型对象。一般来说，在派生类中重定义基类的一个函数的时候，参数列表与返回类型都是一样的。但是，如果基类中的函数返回一个指向基类型对象的指针（或者引用），那么派生类中相应的函数将返回一个指向派生类型对象的指针（或者引用）。

我们不需要另外指定 Student_info 类为 Grad 的友员类，当然这并不是因为友员关系也会被继承，而是因为我们在 Student_info 类中不需要直接调用 Grad::clone 函数；它只需通过对 Core::clone 的虚拟调用来间接调用 Core::clone 函数，如果 Student_info 类型对象指向的是一个 Core 类型对象，由于 clone 函数的虚拟特性会自动调用 Core::clone 函数，当然，如果指向的是一个 Grad 函数，实际调用的当然就是 Grad::clone 函数了。

做出上面的改变之后，我们就可以写出复制构造函数和赋值运算符函数了：

```
Student_info::Student_info(const Student_info& s):cp(0)
{
    if(s.cp) cp=s.cp->clone();
}
Student_info& Student_info::operator=(const Student_info& s)
{
    if(&s!=this) {
        delete cp;
        if(s.cp)
            cp=s.cp->clone();
        else
            cp=0;
    }
    return *this;
}
```

在复制构造函数中，我们把 cp 指针初始化为 0，然后有条件地调用 clone 函数，程序中先对 s.cp 进行判断，如果 cp 指向一个实际的对象，就调用 clone 函数复制该对象，否则 cp 就仍然为 0，也就是说 cp 仍然没有指向任何对象。类似地，赋值运算符函数也是有条件地调用 clone 函数。不过在满足条件并调用 clone 函数之前还要做些预备工作。首先必须谨防发生自我赋值，这通过对比两个操作数以确保两个对象的地址不同来实现。其次，在为不同的对象进行赋值的时候，必须在使 cp 指向新建对象之前先释放 cp 当前指向的对象的内存空间。

如果 cp 为 0，无论是复制构造函数还是赋值运算符函数都不必做任何事，因为对一个空句柄进行复制或者赋值在语法上都是合法的。

13.5 使用句柄类

在写出句柄类之后，我们现在可以用它来使前面 §9.6 中的程序正常运行，现在只需对该程序的一个地方进行改动：

```
int main()
{
    vector<Student_info> students;
    Student_info record;
```

```
    string::size_type maxlen=0;
    //读出并储存数据
    while(record.read(cin)) {
        maxlen=max(maxlen,record.name().size());
        students.push_back(record);
    }
    //对学生记录按姓名字母排序
    Sort(students.begin(),students.end(),Student_info::compare);
    //输出学生姓名与成绩
    for(vector<Student_info>::size_type i=0;
        i!=students.size(); ++i) {
        cout<<setw(maxlen+1)<<students[i].name();
        try{
            double final_grade=students[i].grade();
            streamsize prec=cout.precision();
            cout<<setprecision(3)<<final_grade
                <<setprecision(prec)<<endl;
        }catch(domain_error e) {
            cout<<e.what()<<endl;
        }
    }
    return 0;
}
```

现在的程序在输出循环中读取并处理两种不同的记录：一种是本科生的记录；一种是研究生的记录。因为 Student_info 类的 read 可以读取任何一种记录。这个 read 函数先读出输入流的首字母，根据这个首字母决定将要读的记录的类型，然后为一个新建的相应类型的对象分配内存空间，用输入流读出的数据初始化这个新对象。构造函数用读取的数据来初始化相应的 Core 类型对象或者 Grad 类型对象，然后在 record 中保存一个指向该新建对象的指针。在将这个对象复制到向量中时，作为副作用，Student_info 的复制构造函数被调用。

接下来对数据进行排序。程序把 Student_info::compare 函数传递给 Sort 函数，并调用 sort 函数。它调用基类的 name 函数来对对象中的姓名进行比较。

输出循环部分的代码保持不变。在每一步循环中，students[i]都代表一个 Student_info 类型对象。该对象包含一个指向 Core 类型对象或 Grad 类型对象的指针。在调用 Student_info 类的 grade 函数时，该函数使用指针调用底层对象的虚拟函数 grade。在运行时根据句柄实际所指对象的类型来决定调用哪一个版本的 grade 函数。

最后当退出主函数的时候，在 read 函数里为 Student_info 类中的成员建立并分配内存空间的对象将会被自动释放。在退出主函数的时候，向量对象将被删除。向量类的析构函数将删除 students 中的每个元素，这又会调用 Student_info 类的析构函数。在这些析构函数运行之后，在 read 函数中建立并分配内存的对象都将被删除。

13.6 微妙之处

虽然继承与动态绑定的技巧十分强大有效，但是至少在它们刚出现的时候，还是蒙着一层神秘面纱的。我们已经研究过一个使用这类技巧的例子了，现在再来看看这些技巧的一些微妙之处，如果不加注意，它们可能会带来一些麻烦。

13.6.1 继承与容器

在§13.3.1 中，我们说过，如果要把 Core 类型对象保存在一个容器中，那么这个容器就只能存放 Core 类型对象，不能同时存放其他对象。这一点看起来让人迷惑：因为按常理应该是能在这个容器中存放 Core 类型对象以及 Core 类的派生类的对象的。但是，如果我们仔细回忆一下在第 11 章中是如何实现 Vec 类的，我们会发现，Vec 类不得不为容器中的对象分配内存空间。在分配内存的时候，必须告诉系统要为哪一种类型的对象分配内存。这里没有像虚拟函数那样的机制来自动判断要为哪一种类型的对象分配内存。

如果我们坚持 vector<Core>可以用来储存 Core 类型对象和 Grad 类型对象，那将会发生更加令人惊讶的事。事实上我们可以这么做，只不过得到的结果比较出人意料，例如

```
vector<Core> students;
Grad g(cin);                //读一个 Grad 对象
students.push_back(g);      //在 students 里储存 g 对象的 Core 类部分
```

我们可以把 Grad 类型对象储存在 students 对象中，因为我们可以在需要任何一个指向 Core 类型对象的引用的地方用一个 Grad 类型对象代替。因为 push_back 函数的参数类型是指向向量类型对象中的元素的引用，所以我们可以把 g 对象传递给 push_back 函数。但是要注意在把对象存放在 students 中的时候，只有 g 对象的 Core 类部分被复制进 students 中！在§13.2.2 中，这正是我们想要的结果，尽管这结果让人感到很意外——尤其是在第一次遇到这种情况的时候。在这里，push_back 函数会以为它得到的参数是一个 Core 类的对象，然后调用 Core 类的构造函数，只把 g 对象的 Core 类部分复制过来，而 g 中 Grad 类特有的成员数据将被丢弃。

13.6.2 需要哪一个函数？

有一点十分重要，那就是如果一个基类函数与一个派生类函数具有相同的函数名，但是两个函数的参数个数与参数类型都不相同，那么它们就像完全不相干的两个函数一样。例如，我们可能希望添加一个辅助函数用来改变学生的期末考试分数。对于 Core 类学生，这个函数只能用来改变期末考试成绩；而对于 Grad 类的学生，这个函数带有两个参数，第一个参数还是用来改变期末考试成绩，而第二个参数是用来改变论文分数的：

```
void Core::regrade(double d) { final=d; }
void Grad::regrade(double d, double d2 ) {final=d1; thesis=d2; }
```

如果 r 是一个 Core 类型对象的引用，那么下面的程序中

```
r.regrade(100);              //能够正常运行，调用 Core::regrade 函数
r.regrade(100,100);          //产生编译错误，因为 Core::regrade 函数只带一个参数
```

即使 r 指向一个 Grad 类的对象，第二个函数调用也会导致编译错误。因为 r 是指向 Core 类型对象的引用，而在 Core 类的 regrade 函数里只有一个 double 类型的参数。

如果 r 是一个指向 Grad 类型对象的引用，那么接下来发生的事情会更加让人莫名其妙：

```
r.regrade(100);              //产生编译错误，因为 Grad::regrade 带有两个参数
r.regrade(100,100);          //能够正常运行，调用 Grad::regrade 函数
```

现在 r 是一个 Grad 类型对象的引用，我们来看看实际调用的是哪一个函数。Grad 类的 regrade 函数带有两个参数。虽然我们有一个带一个参数的基类 Core 类的 regrade 函数版本，但是因为在派生类中也有 regrade 这个成员函数，所以这个版本的 regrade 函数被隐藏了起来。如果我们想要在一个派生类型对象中以一个基类型对象的名义调用这个版本的函数，就要对它进行显式地调用：

```
r.Core::regrade(100);        //能够正常运行，调用 Core::regrade
```

如果要把 regrade 函数声明为虚拟函数，那么我们必须把基类与派生类中该函数的接口声明成一样的，我们可以像下面这样在 Core 版本的 regrade 函数中另外增加一个具有默认值的不被使用的参数：

```
virtual void Core::regrade(double d, double=0 ) {final=d; }
```

13.7 小结

继承允许我们方便地生成一个与另一个类很相似，不过又有一点不同的类：

```
class base{
public:
    //普通接口
protected:
    //用以访问派生类的成员函数的接口
private:
    //只能用以访问基类的成员函数的接口
};
//基类的公有接口也是派生类接口的一部分
class derived:public base { ... };
```

在从基类中派生而来的类中可以重定义基类中的操作函数，也可以往派生类中加入它自己的成员。类可以被私有地继承：

```
class priv_derived:private base { ... };
```

上面的用法很少使用，一般只在为了实现方便的时候使用。

一般派生类是从一个基类中以公有方式继承而来，这些类的对象一般在需要一个对象，或者需要一个指向基类的引用或者指针的时候使用。

继承关系可以是嵌套的：

```
class derived2:public derived { ... };
```

derived2 类的对象有一个 derived 类的部分，而 derived 类的对象又有一个 base 类的部分。因此，在 derived2 类的对象中同时有 derived 类部分与 base 类部分。

派生类的对象在调用构造函数的时候要为整个对象分配内存空间，它先对对象中的基类部分进行构造，然后构造派生类的部分。一般来说，具体调用哪一个构造函数是在运行时决定的，由实际用来生成派生类型对象的参数来决定调用哪一个构造函数。通过使用构造初始化器列表，这个构造函数会用参数中的数据来初始化对象中的基类部分。如果在构造初始化器中没有显式地对基类部分进行初始化，程序将会自动调用基类的默认构造函数进行初始化。

动态绑定指的是一种在运行时决定具体调用哪一个函数的能力，它根据程序在运行时调用该函数的实际对象的类型来决定实际调用的函数体。动态绑定只有在通过指针或引用来调用虚拟函数的时候才有意义。如果一个函数在基类中被声明为虚拟的，那么这一虚拟特性可以被派生类继承，在派生类中不需要重复这一声明。

在派生类中不需要重定义虚拟函数。如果一个类不对一个继承而来的虚拟函数进行重定义，那么它可以继承其最近的基类中该虚拟函数的定义。但是，在该类中第一次被声明的虚拟函数，一定要在该类中对它进行定义。如果在一个类中声明了一个虚拟函数但却忘了为它写定义，编译器会给出一些莫名其妙的错误信息。

覆盖：如果一个派生类的成员函数与其基类中的一个同名成员函数有相同数目与相同类型的参数，而且两个函数都是（或者都不是）const 类型的函数，那么派生类的这个函数会覆盖基类的函数。在这种情况下，两者的返回类型必须要匹配，不过有一种特殊情况除外，那就是基类的函数返回一个指向基类型对象的指针（或引用），派生类的函数返回一个指向派生类型对象的指针（或引用）。如果参数列表不匹配，那么基类的这个同名函数与派生类中的这个函数之间就没有任何关系。

虚拟析构函数：如果我们想用一个指向基类型对象的指针来删除一个实际上可能是派生类的对象，那么基类中就需要声明一个虚拟的析构函数。如果该类的析构函数不需要做任何工作，我们可以只声明一个虚拟析构函数而不定义它的函数体：

```
class base{
public:
    virtual ~base() { }
};
```

像其他的函数一样，基类中析构函数的虚拟性质也可以被派生类继承，而且在派生

需要对这个析构函数进行重定义。

构造函数与虚拟函数：在一个对象被构造的时候，对象的类型是就是调用构造函数的类的类型——即使这个对象只是一个派生类型对象的基类部分。所以，在构造函数内部调用虚拟函数会被静态绑定成调用正在构造的对象的类中的函数。

静态成员是作为类的成员而存在，而不是存在于每一个对象实例中。所以，在一个静态成员函数中不能使用 this 关键字。这种函数只能访问静态成员数据。整个类的每个静态成员数据只有一个实例，一般来说，它必须在实现类的成员函数的源文件中被初始化。因为静态成员数据是在类定义的范围以外被初始化的，所以你必须在初始化它的时候在成员数据前面加上类名限制：

```
value-type class_name::static-member-name=value;
```

上面的语句对 class-name 类中的 static-member-name 这个静态成员数据进行初始化，对该成员赋以初始值 value。

习题

13-0 编译、运行并测试本章的例程。

13-1 在 Core 类与 Grad 类的构造函数中加入输出函数，使构造函数在执行的时候输出构造函数名与参数列表。例如：

```
cerr<<"Grad::Grad(istream&)"<<endl;
```

你可以向带有一个 istream&类型参数的 Grad 构造函数中加入上面这行语句。然后写一个小程序检验每个构造函数。预测一下会有什么输出结果。不断修改程序并做出预测，直到预测与实际输出的结果相符。

13-2 对于在本章已经定义的 Core 类与 Grad 类，指出下面的各条语句实际上会调用哪个函数：

```
Core* p1=new Core;
Core* p2=new Grad;
Core s1;
Grad s2;

p1->grade();
p1->name();

p2->grade();
p2->name();

s2->name();
s2->grade();
```

在 name 函数与 grade 函数中加入输出语句输出信息表明调用了哪个函数，看看你

是否正确。

13-3 在第 9 章写的类中包括一个成员数据 valid，它是用来让用户检测该对象是否储存着一个学生的记录的。在本章的类的继承树中加入同样的功能。

13-4 向这些类中添加一个函数，根据 § 10.3 中讲到的方法把一个分数成绩转换成相应的字母成绩。

13-5 写一个判断语句判断一个指定的学生是否达到了所有相应的要求，也就是说，检查一下这个学生有没有做完所有的家庭作业，是不是一个研究生，如果是的话有没有写出一篇论文来，等等。

13-6 为系统添加一个新的类，这个类用来代表选取修可过/可不过的学分的学生。假设这些学生没有被要求一定要完成家庭作业，但是他们也可能会自觉地做作业。如果他们做了作业，那么根据正常的计算公式，这些作业成绩会对总分成绩的计算起作用。如果他们没有做家庭作业，那么就只拿他们的期中考试成绩与期末考试成绩求平均来得到他的最后成绩。在最后成绩为 60 分或者 60 分以上的时候才算通过。

13-7 为系统添加一个类，该类用来表示查分的学生。

13-8 写一个程序生成一个成绩报表，它可以处理上面提到的所有四种学生类的对象。

13-9 想想看，在 § 13.4.2 中定义的赋值操作函数如果不能判断一个赋值操作是否是自我赋值，结果会发生什么？

第 14 章

近乎自动地管理内存

在前面第 13 章里我们写了一个 Student_info 句柄函数，在那里我们把两个独立的抽象概念综合在一个类中。Student_info 类不仅提供了对一个学生记录进行操作的接口，还可以用来操纵一个指向实际实现对象的指针。把两个独立的概念综合在一个类，这种做法通常不是很好的习惯。

我们想定义一个类似于 Student_info 的类，不过它从严格的意义上来说只是一个接口类。这种接口类在 C++中是十分常见的，特别是在继承树中尤为常见。我们要设法使我们的接口类把实现细节交给另一个类，这另一个类像一个指针一样工作，不过它是用来管理底层的内存的，我们把它叫作类指针类。一旦我们把 Student_info 类分成一个接口类和一个类指针类，我们就可以把一个类指针类与几个接口类放在一起使用。

下面我们将会看到，我们可以用这样的类来写出一个经常需要管理内存的程序。在合适的地方通过使用几个类指针的对象指向一个底层的对象，我们可以避免对某些对象进行不必要的复制操作。

本章的大部分篇幅用于寻求一个简单问题的解答，这个问题就是：在复制一个对象的时候程序实际上做了什么？乍一看，这个问题好像十分简单：复制出来的新对象是一个独立的对象，它与源对象有着相同的数据。但是，在复制的时候也可能会得到一个仅仅是源对象的一个引用的对象，这么一来问题就变得复杂起来：如果对象 x 是另一个对象 y 的引用，那么复制 x 的时候会不会使 y 也被复制？

有时候后一个问题的答案是显而易见的：如果 y 是 x 的一个成员，那么答案是肯定的，如果 x 只不过是一个指向 y 的指针，那么答案是否定的。本章我们将定义三个不同版本的类指针类，它们在对复制的定义上彼此不同。

这些关于复制的问题，还有类指针类的思想，都是 C++中的抽象概念。在本章我们将实现这些抽象概念，所以本章的内容是本书中最为抽象的部分。希望读者加倍认真地学习本章的内容。

14.1 用来复制对象的句柄

让我们再来看看第 13 章中提到的成绩问题。在解决那个问题的时候，我们需要储存并处理一个对象集合，这些对象代表不同类型的学生。这些对象属于两个类之一，这两个类之间是继承的关系，今后可能会再扩展出更多的类。在 § 13.3.1 的第一个解决方案中，我们用指针来储存混合类型对象的集合。每一个指针有可能指向一个 Core 类型对象，也有可能指向一个 Core 类的派生类的对象。用户必须在代码中负责为对象动态分配内存，还要记住为对象释放内存。这个程序因为要程序员亲自处理管理指针的细节而变得混乱不堪，充满着安全隐患。

之所以产生这样的问题，是因为指针是 C++中最原始、最底层的数据结构之一。对指针进行编程充满了可能引发错误的隐患。很多指针引发的问题都是因为不同指针指向的对象之间不是互相独立的，从而导致产生错误：

- 复制一个指针不会导致对指针所指对象的复制，当无意中使两个指针指向同一个对象时，常常会产生莫名其妙的结果。
- 删除一个指针不会释放指针所指对象所占用的内存，这常常导致内存泄漏。
- 删除一个对象但是没有删除指向该对象的指针会产生一个空悬指针（dangling pointer），在程序中使用这些指针的时候会导致未定义操作。
- 如果定义一个指针但是不对它进行初始化，会使该指针没有指向任何地方，这时候如果程序中用到该指针，会导致未定义操作。

在 § 13.5 中，我们再次解决了成绩问题，这一次我们在程序中用到了 Student_info 句柄类。因为这个类是操纵指针的，所以用户代码只需要处理 Student_info 类型对象，而不需要直接与指针打交道。但是，Student_info 类马上被 Core 类继承的问题束缚着：它包含着从 Core 类公共接口定义映射而来的操作。

现在我们要做的是把这些抽象的概念分开来放进不同的类中。我们还是用 Student_info 类来提供接口，但是想通过另一个类来控制“句柄”。这也就是说，这另一个类将用来管理指向实现对象的指针。这个新的类与类句柄匹配对象所属的类是彼此独立的。

14.1.1 一个通用句柄类

因为我们希望我们的类与它所操作的对象的类之间彼此独立，所以这个类必须是一个模板。因为我们希望用它来封装句柄行为，所以我们把这个类叫做 Handle。在这个类中将具有以下几点性质：

- 一个 Handle 类型对象是一个指向某对象的值。
- 我们可以对 Handle 类型对象进行复制。
- 我们可以通过检测一个 Handle 类型对象来判断它是否指向另一个对象。
- 如果一个 Handle 类型对象指向继承关系树中某个类的一个对象，我们可以用

类对象来触发多态行为。也就是说，如果通过 Handle 类来调用一个虚拟函数，我们希望程序在运行的时候能够动态地选择调用哪一个函数，就像是我们在通过一个真实的指针调用这个虚拟函数那样。

我们的 Handle 类将会有一个规定的接口：一旦你使一个 Handle 类型对象指向一个对象，这个 Handle 类型对象将负责为那个对象进行内存管理。对一个对象我们只能为其匹配一个 Handle 类型对象，在此之后就不应该在随后直接通过指针来访问该对象；所有的访问都要通过这个 Handle 类型对象来进行。这一限制使得 Handle 避免了使用 C++自带的指针时候固有的问题。在复制一个 Handle 类型对象的时候，我们为该对象生成一个新的复件，这样每个 Handle 类型对象都指向各自的复件。在删除一个 Handle 类型对象的时候，它会删除相应的对象，而这也是删除该对象的惟一途径。我们还允许用户生成一个没有指向任何对象的 Handle 类型对象，不过在用户试图通过这样一个对象来访问它所指向的对象时程序会抛出一个异常。用户可以通过检测 Handle 是否有效来避免抛出这个异常。

这些性质有点像我们在实现 Student_info 类的时候提出的要求。Student_info 类的复制构造函数和赋值运算符函数调用 clone 函数来复制相关的 Core 类型对象。它的析构函数也会删除 Core 类型对象。这个对底层对象进行的操作会在删除对象之前先检查一下 Student_info 类型对象是否指向一个实际的对象。现在我们需要一个类来封装类似的行为，不过我们是用它来管理所有类型的对象的：

```
template <class T> class Handle{
public:
    Handle():p(0) { }
    Handle(const Handle& s):p(0) { if (s.p) p=s.p->clone(); }
    Handle& operator=(const Handle&);
    ~Handle() { delete p; }
    Handle(T* t):p(t) { }
    operator bool() const  { return p; }
    T& operator*() const;
    T* operator->() const;
private:
    T* p;
};
```

Handle 类是一个模板类，所以我们可以用它来为所有类型生成相应的 Handle 类。每一个 Handle<T>类型对象都封装有一个指向 T 类型对象的指针；类中的其他操作都是用来管理这个指针的。除了函数名有些变化以外，Handle 类的前四个函数与 Student_info 类中的相应函数是一样的。默认构造函数把指针初始化为 0 以表明该指针是无效的。复制构造函数（根据条件判断）调用相关对象的 clone 函数来生成该对象的一个新的复件。Handle 析构函数删除该对象并释放对象占用的内存空间。赋值运算符函数和复制构造函数一样，也是（根据条件判断）调用 clone 函数来生成对象的一个新的复件：

```
template<class T>
Handle<T>& Handle<T>::operator=(const Handle& rhs)
{
    if(&rhs!=this){
        delete p;
        p=rhs.p?rhs.p->clone():0;
    }
    return *this;
}
```

像其他地方一样，Handle 模板类中的赋值运算符函数先检查是否是在进行自我赋值，如果是的话就不进行任何操作。如果不是，就先释放我们操作着的对象的内存。然后把右操作数对象的内容复制到左操作数对象里。进行复制的程序语句中使用条件运算符（见§3.2.2）来判断调用 clone 函数是否安全。如果 rhs.p 有效，则调用 rhs.p->clone 并使结果指针指向 p，否则就把 p 设为 0。

因为 Handle 类模拟指针的行为，所以我们需要找到一种办法使指针与一个实际的对象相关联，我们在带有一个 T*类型参数的构造函数中进行这一操作。这个构造函数从参数中得到一个指针 T，然后使 Handle 类与 T 所指的对象关联在一起。例如，如果我们定义

```
Handle<Core> student(new Grad);
```

也就构造了一个名为 student 的 Handle 类型对象，它封装了一个 Core*指针，我们对该指针初始化，使它指向我们刚才生成的一个 Grad 类型对象：

最后，我们定义了三个运算符函数。第一个是 operator bool()函数，它可以让用户在一个条件语句中检测一个 Handle 类型对象的值。在 Handle 类型对象与一个实际的对象关联的时候该函数返回真值，否则返回假值。另外两个算符函数是 operator*和 operator->函数，它们是用来访问与 Handle 相关联的对象的：

```
template<class T>
T& Handle<T>::operator*() const
{
    if(p)
        retrun *p;
    throw runtime_error("unbound Handle");
}
template<class T>
T* Handle<T>::operator->() const
{
    if(p)
```

```
        return p;
    throw runtime_error("unbound Handle");
}
```

把一个 C++内建的一元运算符*作用在一个指针上将得到该指针指向的对象。这里我们定义了我们自己的*运算符，对一个 Handle 类型对象作用*运算符，其结果相当于对 Handle 类型对象中的指针成员作用 C++内建的*运算符。如果有一个 student 类型对象，那么*student 将得到与*(studnet.p)相同的结果（假设我们有权访问 p 的数据成员）。换句话说，*student 的结果事实上是指向一个 Grad 类型对象的引用，这个对象是在对 student 进行初始化时生成的。

->运算符比前面的运算符稍微复杂一点。粗看上去，->运算符像是一个二元算符，但是事实上它与普通的二元算符不同。像一个范围运算符或者一个点运算符一样，->运算符用来访问左操作数对象中的一个成员，这个成员名称由->运算符的右操作数来表示。因为成员名不是表达式，所以我们不能通过一个成员名来直接访问用户指定的成员。C++语法要求我们定义一个->运算符来返回一个可以看作是指针的值。在定义 operator->函数的时候，如果 x 是定义 oeratror->函数的类的一个对象，那么

```
x->y
```

就等同于

```
(x.operator->())->y
```

在这里，operator->函数返回 x 对象封装的指针。因此，对于一个 student 对象，

```
student->y
```

就等同于

```
(student.operator->())->y
```

根据我们对 operator->函数的定义，上式又等同于

```
student.p->y
```

（忽略了一个事实，那就是自保护机制一般不允许我们直接访问 student.p）。因此，->运算符会把一个 Handle 类型对象对它的函数调用交给 Handle 类型对象的指针成员，由这一指针成员调用该运算符。

我们的目标之一是使 Handle 类具有与 C++自带指针相关的多态性。看看已经写好的 operator*与 operator->定义，我们知道目的已经达到了。这两个运算符函数生成一个引用或者一个指针，通过这个引用或者指针可以获得动态绑定特性。例如，如果我们运行 student->grade()，实际上是通过 student 对象里的 p 指针来调用 grade 函数。在运行的时候系统会根据 p 所指对象的实际类型决定实际调用的是哪一个版本的 grade 函数。假设 student 对象仍然指向初始化时的一个 Grad 类型对象，那么实际调用的就是 Grad::grade 函数。类似

为 operator*得到一个引用，所以如果运行(*student).grade()，那么我们是在通过一个引用调用 grade 函数，在运行的时候系统才会决定调用哪一个版本的 grade 函数。

14.1.2 使用一个通用句柄

我们可以用 Handle 类重写 § 13.3.1 中基于指针的成绩程序：

```
int main()
{
    vector<Handle<Core>> students;      //改变其类型
    Handle<Core> record;                //改变其类型
    char ch;
    string::size_type maxlen=0;

    //读取并储存数据
    while(cin>>ch) {
        if(ch=='U')
            record=new Core;        //为一个 Core 类型对象分配内存
        else
            record=new Grad;        //为一个 Grad 类型对象分配内存
        record->read(cin);    //先调用 Handle<T>::->，然后调用虚拟的 read 函数
        maxlen=max(maxlen,record->name().size()); //调用 Handle<T>:: >
        students.push_back(record);
    }
    //重写 compare 函数使之可以用 const Handle<Core>&类型作参数
    sort(students.begin(),students.end(),compare_Core_handles);
    //输出姓名与成绩
    for(vector<Handle<Core>>::size_type i=0;
         i!=students.size(); ++i) {
        //students[i]是一个 Handle 类型对象，对之间接引用以调用函数
        cout<<setw(maxlen+1)<<students[i]->name();
        try{
            double final_grade=students[i]->grade();
            streamsize prec cout.precision();
            cout<<setprecision(3)<<final_grade
                <<setprecision(prec)<<endl;
        } catch(domain_error e) {
            cout<<e.what()<<endl;
        }
        //没有 delete 语句
    }
    return 0;
}
```

这个程序保存的是 Handle<Core>类型对象而不是 Core*类型对象，因此，像我们在 [illegible] 中所做的那样，我们需要写一个非重载的比较操作函数以用来对传递给 sort 函数的

Handle<Core>&类参数进行比较。我们把这个函数的实现留在练习中给读者自己完成，现在我们假设已经写好这么个函数，它的函数名为 compare_Core_handles。

与以前的版本相比，这个版本还有另外一个不同之处，那就是输出循环部分。我们对 students 中得到的值进行间接引用，得到一个 Handle 类型对象，然后调用 Handle<T>::operator-> 运算符函数，通过底层的 Core*来访问 name 函数与 grade 函数。例如，在 students[i]->grade() 中调用重载运算符->，我们实际上运行的是 students[i].p->grade()。因为 grade 函数是虚拟的，所以会在运行的时候根据 students[i].p 所指对象的类型来决定并调用一个版本的 grade 函数。更进一步说，因为 Handle 类会替我们管理内存，所以我们不需要另外调用 delete 函数来删除 students 中的指针元素所指向的对象。

更为重要的是，我们可以重新编写 Student_info 类，现在我们可以让这个类成为一个纯接口类，让它从管理指针的工作中解脱出来：

```
class Student_info{
public:
    student_info() { }
    student_info(std::istream& is) { read(is); }
    //现在不再需要复制构造函数，也不再需要赋值操作函数与析构函数了
    std::istream& read(std:istream&);
    std::string name() const{
        if(cp) return cp->name();
        else throw runtime_error("uninitialized Student");
    }
    double grade() const {
        if(cp) return cp->grade();l
        else throw runtime_error(uninitialized Student");
    }
    static bool compare(const Student_info& s1,
                        const Student_info& s2) {
        return s1.name()<s2.name();
    }
private:
    Handle<Core> cp;
};
```

在这个版本的 Student_info 中，cp 的类型已经不再是 Core*，而是 Handle<Core>。由于 Handle 可以很好地管理它所操纵的底层对象，我们也就不需要再去实现那些用于控制复制的函数。另外那个构造函数则没有改变。函数 name 和 grade 看起来也没有改变，但它们的执行依赖于 cp 向 bool 转换得到的结果和 Handle 类中被重载的 operator->运算符（它被用来调用底层对象的函数的）。

为了完善我们这次的重新实现，我们还需要编写函数 read 如下：

```
istream& Student_info::read(istream& is)
```

```
{
    char char;
    is >> ch;          // 获取记录类型

    // 为适当的类型分配新的对象
    // 使用 Handle<T>(T*)来为指向那个对象的指针构造一个 Handle<Core>的对象
    // 调用 Handle<T>::opeartor 来将 Handle<Core>赋值给左边的对象
    if (ch == 'U')
        cp = new Core(is);
    else
        cp = new Grad(is);

    return is;
}
```

上面的代码看上去很像前面写的 Student_info::read 函数，不过运行起来的时候两者大不一样。最明显的区别是，这里去掉了 delete 语句，因为对 cp 进行赋值会释放对象占用的内存。为了理解这些代码，我们需要对它进行仔细的研究。例如，如果执行 new Core(is)，我们将得到一个 Core*类型的对象，然后隐式地调用 Handle(T*)构造函数把它转换成一个 Handle<Core>类型对象。然后调用 Handle 类的赋值运算符函数把这个 Handle 类的值赋给 cp，在调用赋值运算符函数的时候会自动删除 Handle 此前所指的对象。在这个删除操作过程中，我们构造和删除了一个我们生成的 Core 类型对象的一个复件。

14.2 引用计数句柄

现在，我们已经把管理指针的工作从接口类中分离出来，生成了一个新的类指针类。我们可以用 Handle 类来实现不同的接口类，在使用它们的时候不必考虑内存管理操作。但是，我们的 Handle 类在对底层数据进行复制或者赋值操作的时候仍然存在问题，即使在它们不需要这么做的时候也不例外。原因是 Handle 类总是会复制 Handle 类型对象指向的对象。

一般来说，我们希望亲自决定是否对对象进行复制。例如，我们可能希望一个对象是另一个对象的复件，但是它们的值共用同一块内存，这时候我们不需要它们像是一个值一样工作。有些时候一旦类的对象生成以后，其他类没有办法改变对象的状态。在这些时候，没有理由要对基层的对象进行复制，复制这些对象只会浪费 CPU 时间与内存空间。为了支持这些类型的类，我们希望有种 Handle 类，它在 Handle 类型对象本身复制的时候不对底层的对象进行实际的复制。当然，即使我们允许几个 Handle 类型对象指向同一个底层对象，也还是要在某个时候释放这个对象占用的内存。显然，释放对象内存的时机应该选在指向该对象的最后一个 Handle 类型对象被删除的时候。

最后，我们将用到**引用计数（reference count）**，这是一个对象，它用来记录有多个对象指向某个底层对象。每个目标对象都有一个与该对象关联的引用计数。在每次生成一

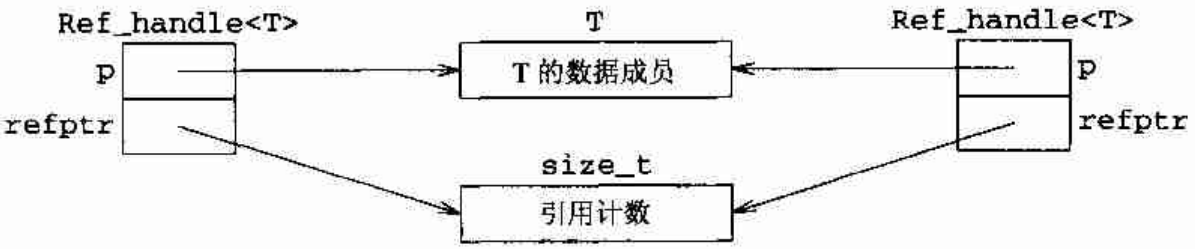
指向目标对象的句柄的时候，我们让引用计数加一，在有一个句柄对象被删除的时候让引用计数减一。在最后一个句柄对象被删除的时候，引用计数将变为零。在引用计数为零的时候，我们就可以安全地删除目标对象并释放其内存。

这一技术省去了大量不必要的内存管理与数据复制。我们首先建立一个名为 Ref_handle 的新类，这个类将定义如何往我们的 Handle 类型对象中加入引用计数。然后，在下面的两个部分里，我们将探讨一下，如何利用引用计数，使用同样的代码来实现一个像值一样工作的类。

为了增加对某个类型对象的计数功能，我们必须分配一个计数器，然后修改对象的创建、复制与删除操作，以使它们可以正确地更改计数器的值。每个与 Ref_handle 类型对象关联的对象都有一个与对象相关的引用计数。现在惟一的问题是把计数器存放在什么地方。一般来说，我们没法得到任何一种类的源代码，所以不能通过往类中加入一个计数器来实现计数。实际上，我们是在 Ref_handle 类中加入另外一个指针来跟踪计数。每一个与一个 Ref_handle 类型对象关联的对象都有一个相应的引用计数，这个引用计数用来表示我们生成了该对象的多少个复件：

```
template <class T> class Ref_handle {
public:
    //像管理指针一样管理引用计数
    Ref_handle():refptr(new size_t(1)),p(0) { }
    Ref_handle(T* t):refptr(new size_t(1)),p(t) { }
    Ref_handle(const Ref_handle& h):refptr(h.refptr),p(h.p)
    {
        ++*refptr;
    }
    Ref_handle& operator=(const Ref_handle&);
    ~Ref_handle();
    //同前
    operator bool() const { return p; }
    T& operator*() const
    {
        if(p)
            return *p;
        throw std::runtime_error("unbound Ref_handle");
    }
    T* operator->() const
    {
        if(p)
            return p;
        throw std::runtime_error("unbound Ref_handle");
    }
private:
    T* p;
    size_t* refptr;             //新加部分
};
```

我们在 Ref_handle 类中新增加了一个数据成员，并且对构造函数稍加改动了一下以对该数据成员进行初始化。默认构造函数与带有一个 T*参数的构造函数被用来生成一个新的 Ref_handle 类型对象，所以它们要为一个新的引用计数（size_t 类型）分配内存，然后把该计数器的值设为 1。复制构造函数不用生成一个新的对象，它对参数给出的 Ref_handle<T>类型对象进行指针复制，然后使计数器加 1，以表示指向这个 T 类型对象的指针又比以前多了一个。因此，新的 Ref_handle<T>类型对象指向同一个 T 对象与同一个引用计数。举个例子来说，如果 X 是一个 Ref_handle<T>类型对象，而且我们生成 X 的一个复件 Y，那么它们之间的关系如下图所示：



在赋值运算符函数中也使用了这个引用计数，而不是直接复制底层对象：

```
template<class T>
Ref_handle<T>& Ref_handle<T>::operator=(const Ref_handle& rhs)
{
    ++*rhs.refptr;
    //释放左操作数对象，如果必要的话删除指针
    if(--*refptr==0) {
        delete refptr;
        delete p;
    }
    //复制右操作数对象的值
    refptr=rhs.refptr;
    p=rhs.p;
    return *this;
}
```

像以前一样，我们要注意进行自我赋值的情况，我们通过在对左操作数对象的引用计数减一之前先对右操作数对象的引用计数加一来解决自我赋值的问题。如果两个操作数都指向同一个对象，那么最终的效果是该对象的引用计数保持不变，这可以确保引用计数不会被无意归零。

如果在为引用计数减一的时候使它达到零值，那么左操作数就是与底层对象关联的最后一个 Ref_handle 类型对象。因为我们要抹去左操作数对象的值，所以要调用 delete 函数释放对象的最后一个引用。因此，我们必须在重置 refptr 与 p 的值之前删除这个对象以及该对象的引用计数。我们必须同时对 p 与 refptr 调用 delete 函数，因为这两个都是动态分配内存的对象，为了避免内存泄漏，我们必须删除它们并为它们释放内存。

在删除指针之后，如果有必要，我们还要把右操作数对象的值复制到左操作数对象中，然后一般返回一个指向左操作数的引用。

析构函数和赋值算符函数一样，先检查要删除的 Ref_handle 对象是否是指向 T 对象的最后一个引用。如果是的话，就删除指针所指向的对象并释放其占用的内存：

```
template<class T> Ref_handle<T>::~Ref_handle()
{
    if(--*refptr==0) {
        delete refptr;
        delete p;
    }
}
```

对于可以在一个对象的不同复件之间共享同一块内存数据的类，这个版本的 Ref_handle 类可以正常工作，但是对于像 Studnet_info 这样的，想要使类型对象像一个值一样被操作的类又怎么样呢？例如，如果我们把 Ref_handle 类应用到 Student_info 类上，那么在运行下面的语句的时候：

```
Student_info s1(cin);          //用标准的输入来初始化 s1
Student_info s2=s1;            //把 s1 的值“复制”到 s2 中
```

尽管看上去 s2 是 s1 的一个复件，但实际上 s1 与 s2 这两个对象指向同一个底层对象。在我们的新的 Ref_handle 类中从来没有调用过 clone 函数，由此可以证实上面的论断。因为 Ref_handle 从未调用过 clone 函数，这个类型的句柄从来就没有真正复制过这些对象。另一方面，Ref_handle 类的这一版本也可以避免对数据进行没有必要的复制。问题是，无论需要与否，它都不会对数据进行复制。这下我们该怎么办呢？

14.3 可以让你决定什么时候共享数据的句柄

迄今为止，我们已经讲过这个通用句柄类的两个可能的定义。第一个版本总是对底层对象进行复制；第二个版本从来不会真正地复制底层对象。一种更好的办法是写一个类，这个类允许由程序来决定什么时候需要复制目标对象，而什么时候又不需要复制目标对象。这样一个句柄类更加实用，它保留了 Ref_handle 类的功能，而且允许类的作者使 Handle 类型对象像一个值一样被使用。这样一个句柄保留了 C++自带的指针的有用的特性，同时避免了许多的安全缺陷。下面我们将使用最后写出来的句柄类 Ptr，并由此证明它是 C++自带的指针的一个很好的替代品。一般来说，Ptr 类必须在我们打算改变一个对象的值的时候复制该对象，当然，这要求同时有另一个句柄指向同一个对象。幸运的是，引用计数可以告诉我们一个句柄是不是指向某对象的最后一个引用。

这里的 Ptr 类的基本原理与前面 § 14.2 中的 Ref_hadle 类相同。我们要做的只不过是新加

入一个成员函数：

```
template<class T> class Ptr {
public:
    //新加一个成员函数，用来在需要的时候有条件地复制对象
    void make_unique() {
        if(*refptr !=1) {
            --*refptr;
            refptr=new size_t(1);
            p=p?p->clone():0;
        }
    }
    //剩下的部分除了名字以外与 Ref_handle 类相同
    Ptr():refptr(new size_t(1)),p(0) { }
    Ptr(T* t):refptr(new size_t(1)),p(t) { }
    Ptr(const Ptr& h):refptr(h.refptr),p(h.p) { ++*refptr; }
    Ptr& operator=(const Ptr&);     //实现部分与 § 14.2 中相似
    ~Ptr();                         //实现部分与 § 14.2 中相似
    operator bool() const { return p; }
    T& operator*() const;           //实现部分与 § 14.2 中相似
    T* operator->() const;          //实现部分与 § 14.2 中相似
private:
    T* p;
    size_t* refptr;
};
```

这个新加入的 make_unique 函数完成了我们想做的工作：如果引用计数的值是 1，该函数不执行任何工作；否则，它就调用句柄所指对象的类的 clone 函数，生成该对象的一个复件，然后使 p 指向这个复件。如果引用计数不是 1，那么至少还有另一个 Ptr 类型对象指向初始的对象。因此，我们要把与初始对象关联的引用计数减 1（这么做可能使其减小为 1 但不会减为 0）。然后，我们为句柄，为其他所有可能在以后进行复制的对象生成一个新的引用计数。因为到现在为止只有一个 Ptr 类型对象与我们的这个复件关联，所以把计数器设为 1。在调用 clone 函数之前，先检查指向将要复制的对象的指针是不是指向一个真实的对象。如果是的话，就调用 clone 函数来复制该对象。完成以上操作之后，这个 Ptr 类型对象是惟一一个与 p 所指对象关联的对象。这个对象可能与初始的对象是同一对象（在初始对象的引用计数为 1 的时候），也可能只是初始对象的一个复件（在初始对象的引用计数大于 1 的时候）。

现在我们可以在 § 14.1.2 中写的 Student_info 类中使用这个最新版本的 Ptr 类。这么做完全不需要对 Student_info 类的应用程序做任何改变。因为 Student_info 类的任何一个操作都不会不覆盖而改变对象的值。惟一一个会改变对象的值的操作是 read 函数，但是这个函数总是把一个新生成的值赋给对象的 Ptr 成员数据。在这么做的时候，Ptr 类的赋值运算符函数可能删除旧值，也可能保留旧值，这要取决于是不是还有其他的对象指向旧值。在任何一种情况中

我们将要读取的对象都有一个新的 Ptr 类型对象，这也是该对象的惟一使用者。如果运行下面的程序：

```
Student_info s1;
read(cin,s1);              //为 s1 赋一个值
Student_info s2=s1;        //把 s1 的值复制到 s2 中
read(cin,s2);              //读取 s2：改变的是 s2 而不是 s1
```

那么 s2 的值将在 read 函数的调用中被重置，而 s1 的值保持不变。

另一方面，如果我们把在 § 13.6.2 中写的 virtual 版本的 regrade 函数加入 Core 类的继承树中，并且假设在 Student_info 类中已经给出一个相应的接口函数，那么为了调用 make_unique 函数，regrade 函数要做些改动：

```
void Student_info::regrade(double final,double thesis)
{
    //在改变对象之前先得到自己的复件
    cp.make_unique();
    if(cp)
        cp->regrade(final,thesis);
    else  throw run_time_error("regrade of unknown student");
}
```

14.4 可控句柄的一个改进

尽管我们的可控句柄十分有用，可是它并不能总是按照我们希望的方式工作。例如，现在假设我们想用它来实现第 12 章中所讲的 Str 类。在 § 12.3.4 中，我们通过连接两个已有的 Str 类型对象，隐式地复制一大串的字符，生成一个新的 Str 类型对象。通过对 Str 类型对象加入引用计数，我们希望至少能省去一些复制操作：

```
//这个版本能正常运行吗?
class Str{
    friend std::istream& operator>>(std::istream&,Str&);
public:
    Str& operator+=(const Str& s) {
        data.make_unique();
        std::copy(s.data->begin(),s.data->end(),
                std::back_inserter(*data));
        return *this;
    }
    //接口部分与前面相同
    typedef Vec<char>::size_type size_type;
    //重写生成 Ptr 类型对象的构造函数
    Str():data(new Vec<char>) { }
```

```
    Str(const char* cp):data(new Vec<char>) {
        std::copy(cp,cp+std::strlen(cp),
                std::back_inserter(*data));
    }
    Str(size_type n, char c):data(new Vec<char>(n,c)) { }
    template<class In> Str(In i,In j):data(new Vec<char>) {
        std::copy(i,j,std::back_inserter(*data));
    }
    //在必要的时候调用 make_unique 函数
    char& operator[](size_type i) {
        data.make_unique();
        return (*data)[i];
    }
    const char& operator[](size_type i) const { return (*data)[i]; }
    size_type size() const {return data->size(); }
private:
    //保存一个指向向量的 Ptr
    Ptr<Vec<char>> data;
};
//像§12.3.2 和 12.3.3 中那样实现
std::ostream& operator<<(std::ostream&,const Str&);
Str operator+(const Str&,const Str&);
```

我们保留了 Str 类的接口，但是从根本上改变了接口的实现。我们不是在每一个 Str 类型对象中直接储存一个向量类型对象，而是使用了一个指向向量的 Ptr 对象。这种办法可以允许多个 Str 类型对象共用同一底层字符数据。构造函数通过为一个新的向量类型对象分配内存，并用适当的值对向量类型对象初始化来初始化该 Ptr 类型对象。读取（而不是改写）data 数据的操作代码，与以前的版本相同。当然，这些操作现在是作用在一个 Ptr 类型对象上，也就是说这些操作是间接地通过 Ptr 类型对象中的指针来获得 Str 类型对象中的底层字符数据的。真正有趣的是那些用来改变 Str 类型对象的操作，譬如输入运算符，混合连接运算符和非常量版本的下标运算符等等。

例如，我们来看一看 Str::operator+=函数的实现代码。这个函数要把数据追加到底层的向量类型对象中，所以它调用 data.make_unique()函数。这么做之后，Str 类型对象就有了自己的一个底层数据的复件，它可以对这个复件自由地进行修改。

14.4.1 复制我们不能控制的类型

不幸的是，make_unique 函数的定义中存在一个严重的问题：

```
tmeplate<class T>
void Ptr<T>::make_unique()
{
    if(*refptr != 1) {
```

```
        --*refptr;
        refptr=new size_t(1);
        p=p?p->clone():0;        //这里存在问题
    }
}
```

我们来看一看对 p->clone 函数的调用，因为我们使用了 Ptr<vector<char> >，所以它将试图调用 vector<char>类中的 clone 函数。但是不幸的是，在该类中没有 clone 函数。

因为 clone 函数是一个虚拟函数，所以它必须是一个与 Ptr 类关联的类的成员函数。换句话说，为了使 Ptr 有可能适用于一个继承树上所有的类，把 clone 函数定义成为一个成员函数是至关重要的；当然，这么做是不可能的，因为我们不能改变 Vec 类的定义。这个类被设计来实现标准向量类的接口函数的一个子集接口。如果我们向其中加入一个 clone 成员函数，那么就是加入了一个向量类所没有的成员，这样得到的 Vec 类其接口函数就不是向量类接口函数的子集了。那我们该怎么做呢？

解决这一类难题的办法通常都要用到一个思想，这个思想我们常常戏称为软件工程中的基本定理：所有的问题都可以通过引入一个额外的间接层来解决。现在的问题是，我们试图调用一个实际上并不存在的函数，但是我们又没有办法令这个函数存在。那么解决的办法是，不要直接去调用这个成员函数，而是定义一个既可以直接调用又可以创建的中间的全局函数。这个函数我们仍然叫它 clone 函数。

```
template<class T> T* clone(const T* tp)
{
    return tp->clone();
}
```

适当修改 make_unique 成员函数以调用 clone 函数

```
teplate<class T>
void Ptr<T>::make_unique()
{
    if(*refptr!=1) {
        --*refptr;
        refptr=new size_t(1);
        p=p?clone(p):0;  //调用 clone 的全局函数版本（而不是成员函数版本）
    }
}
```

很显然，引入这样一个中间函数不会影响 make_unique 函数。它仍然调用 clone 函数，而这样调用的是实际上被复制的对象的 clone 成员函数。不过，现在 make_unique 是通过一个间接的途径工作的：它先调用非成员函数版本的 clone 函数，而由这一非成员版本的 clone 函数调用 p 所指向的对象的 clone 成员函数。对于像 Student_info 这样的定义了 clone 成员函数的类来说，这一间接的调用是多余的。但是对于像 Str 这种类，它储存着 Ptr 类型对象，而在 Ptr 类中没有定义 clone 函数，这种情况下这一间接调用就显得很重要了。对于后面这种类型

我们也可以定义另一个中间函数：

```
//这是使 Ptr<Vec<char>>类正常工作的关键
template<>
Vec<char>* clone(const Vec<char>* vp)
{
    return new Vec<char>(*vp);
}
```

在该函数的开头使用 template<>表明这个函数是一个模板特化。这种特化为特定的参数类型定义了一个特殊版本的模板函数。通过定义它为特殊化，clone 函数在带一个指向 Vec<char>类型对象的指针作为参数的时候，会与带其他类型的指针作为参数时具有不同的功能。在向 clone 函数传递一个 Vec<char>*类型的参数时，编译器调用一个特殊版本的 clone 函数。在向 clone 函数传递其他类型的指针参数时，编译器将对 clone 函数的通用模板进行实例化，这一实例化使得编译器实际调用的是指针所指对象的 clone 成员函数。clone 函数的特化版本调用 Vec<char>类的复制构造函数，用参数提供的指针所指对象来构造一个新的 Vec<char>类型对象。虽然这个特化的 clone 函数没有提供虚拟特性，但是因为 Vec 类没有派生类，我们也不需要这么做。

现在我们知道 clone 不一定非要作为一个类的成员函数而存在，这在一定程度上减少了我们对 clone 函数的依赖性。通过引入一个中间函数，我们可以特化一个 clone 模板，让这个特化的 clone 函数可以在复制一个特殊类（该类中没有定义 clone 成员函数）对象的时候被调用，从而调用一个 clone 成员函数，调用一个复制构造函数或者你想要调用的任何函数。在不调用这个特殊化的 clone 函数的时候，我们调用相应类的 clone 函数，不过这要求同时调用 make_unique 函数。换句话说

- 如果你使用了 Ptr<T>类但是没有调用 Ptr<T>::make_unique 函数，那么有否定义 T::clone 函数都没有关系。
- 如果你调用了 Ptr<T>::make_unique 函数，而且定义了 T::clone 函数，那么 make_unique 函数将调用 T::clone。
- 如果你调用了 Ptr<T>::make_unique 函数，而且不想调用 T::clone 函数（也许它根本就不存在），那么你可以通过特化一个 clone<T>模板来做你想做的工作。

这个额外的中间函数使得我们可以很细微地定义 Ptr 类的功能。现在只剩下一个问题了，这也是最难的部分——决定是否真的要进行复制操作。

14.4.2 复制在什么时候是必要的？

为了解决本例的最后一个问题，我们有必要对前面的内容详细地作一回顾。看一看我们定义的两个不同版本的 operator[]函数。其中一个调用了 data.make_unique 函数；而另一个版本没有。为什么会有此差别呢？

这一差别与函数是否是一个常量成员有关。operator[]函数的第二个版本是一个常量成员函数，这意味着该函数不会改变对象的内容，因为它返回一个 const chra&类型值给调用者。因此，即使让几个 Str 类型对象共享同一底层的 Vec<char>类型对象也没有什么坏处，毕竟，用户不能通过一个常量返回值来改变 Str 类型对象的值。

不同的是，operator[]函数的第一个版本返回一个 char&值，这意味着用户可以用这一返回值改变 Str 类型对象的值。如果用户果真这么做，我们希望把对 Str 类型对象值的改变只限制这个对象上，不让这一改变影响到共享同一底层 Vec 类型对象的其他对象。为了实现这一目的，我们在定义中规定，在返回一个指向 Vec 类型对象的字符的引用之前，不能通过调用 Ptr 类型对象的 make_unique 函数来改变其他 Str 类型对象的值。

14.5 小结

模板特化看起来像是被特化了的模板定义，但实际上它忽略了一个或多个类型参数，而以一个特定类型来代替。对特化模板的各种各样的使用已经超出了本书的讨论范围，不过你应该知道它们确实存在，它们对于在编译过程中需要根据参数类型来作出决定的时候十分重要。

习题

14-0 编译、运行并测试本章的例程。

14-1 实现 Ptr<Core>类的对比操作函数。

14-2 使用 Ptr<Core>对象实现学生成绩程序并对其进行检验。

14-3 使用 Ptr 类的最终版本来实现 Student_info 类，然后用这一版本的 Student_info 类来实现 § 13.5 的学生成绩程序。

14-4 使用 Ptr 类的最终版本重写 Str 类。

14-5 通过编译重写后的 Str 类并运行使用了 Str 的程序（例如 split 函数和使用了 Vec<Str>类的图形操作函数）检验这个类的正确性。

14-6 Ptr 类实际上解决了两个问题：保留了引用计数，为对象分配内存和释放内存。请定义一个只负责引用计数的类；然后用该类重新实现 Ptr 类。

第 15 章

再探字符图形

在编程处理大而复杂的系统时，使用继承是最为有用的，这在任何一本入门书籍里都达成了共识。一直以来，我们对§5.8 中介绍的字符图形的例子都感到十分满意，因为这是一个面向对象的解决方案，但是，我们还可以对它进行优化，仅用几百行的代码就可以将它实现。这个例子我们已经用了很多年，在使用中我们不断地对它进行提炼和简化。在本书中我们再来回顾一下这个例子，如果使用在第 14 章中学习的标准库和通用句柄类，这个例子的代码起码可以减少到原来的一半。

在本书§5.8 中，我们写了几个函数，把一个字符图形描述成一个 vector<string>，在这种方法中我们每生成一个图形都需要复制字符。复制所有这些字符既既耗费时间又耗费空间。举个例子来说，如果我们要连接一个图形的两个复件，那么我们将不得不保存每个字符的三个复件：一个是原始图形的复件，还有新生成的图形的两边各需要一个复件。

更糟糕的是，§5.8 提出的解决方案丢失了所有关于图形结构的信息。我们无从知道一个给出的图形是如何生成的。它可能是我们的客户给出的原始图形，也有可能是对某一原始图形进行一种或者几种特定操作而生成的图形。有些很有用的操作需要知道一个图形的结构信息，例如如果想修改一副图中的框架字符，那么我们必需知道图中哪些部分给加了框架，哪些部分没有加框架。仅仅看在当前给出的图形中哪些字符有框架是不够的，因为极有可能这些加了框架的字符是原始图形中的一部分，而不是后来进行了加框操作而生成的。

大家将看到，在本章我们通过使用继承和通用句柄类来保存图形的结构信息，同时又能够有效地节省系统开销。

15.1 设计

现在有两个不同的问题等待解决。一个是总体设计上的问题——我们想保存关于一个图形是如何生成的结构信息。另一个是实现细节中的问题——我们想减少要保存的数据的复件份数。我们决定要将一幅图形保存为一个 vector<string>，上面说的两个问题都源于这个目的，所以，让我们再来看一看这个决定。

对于实现细节中的问题，我们可以通过将数据定义成在 14 章中讨论过的 Ptr 类来解决。在 Ptr 类中，我们将实际的字符数据保存在一个简单的对象中，然后让几个图形共用同一个对象。例如，如果要给一个图形加一个框架，我们不再需要复制图形中要加框的字符。在 Ptr 类中有一个引用计数与这些数据关联，记录着有多少图形同时在使用这些数据。

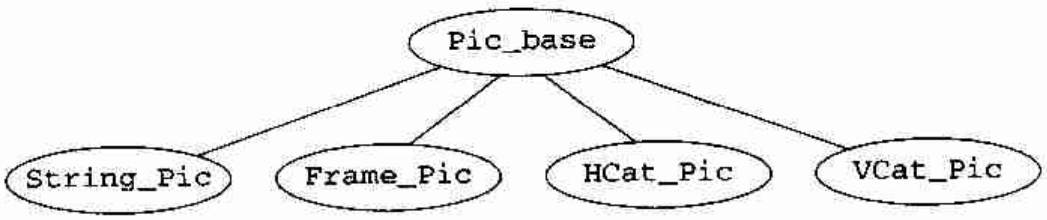
而总体设计上的问题则比较棘手。每一个图形都有自己的结构，我们想要保存这些结构信息。我们手中的图形可能是一张原始的图形，也可能是通过调用以下三种操作函数之一生成的图形：frame 函数，生成一幅框架图形；hcat 函数和 vcat 函数，分别生成一幅水平/垂直连接的图形。

换句话说，我们有四种类似的图形。且不说它们的相似点，我们知道它们有不同的生成途径，现在我们想知道是哪一种途径。

15.1.1 使用继承来模拟结构

现在来研究一下如何用继承完美地解决我们的问题：我们有几种不同的数据结构，它们彼此相似，但其差异不容我们忽视。这几种数据结构都是一类图形，所以继承是对这些数据结构的一种很好的描述方式。我们可以先定义一个基类，让它描述图形的一些基本的特征，然后从该基类中派生出不同的子类以描述一种特定特征的图形。

我们将构造四种派生类：String_Pic、Frame_Pic、Hcat_Pic 和 Vcat_Pic。客户直接给出的字符串的图形用 String_Pic 描述；通过给某幅图形加框架生成的图形用 Frame_Pic 描述；通过将两幅图形水平/垂直连接生成的图形用 HCat_Pic/Vcat_Pic 描述。为了使这些派生类通过继承联系起来，我们可以用虚拟函数来写那些不需要明确指定是通过哪种操作生成图形的代码。这样，我们的客户还可以像以前一样执行任何一个操作而不需要知道他们正在进行操作的图形是哪一种类型的图形。所有这四个派生类都将从一个更具有普遍特征的基类（Pic_base）派生出来，下图画出了基类与派生类的继承关系层次。



接下来的问题是决定要不要使这个继承关系对客户可见。经过分析我们发现我们没有任何理由使之对客户可见，因为我们的任何一个操作都不是对某个特定的图形进行的，相反，这些操作要处理的只是一幅图形的抽象概念。另外，我们还打算使用引用计数。在程序中隐藏继承关系并使用关联引用计数将会使得我们的客户在使用中感觉更加方便。

我们的客户不需要直接对 Pic_base 基类进行操作，也不需要与其派生类打交道，我们将定义一个特定图形的接口类。用户们可以不必关心实现细节也可以访问这个类。值得一

由于我们的类使用了 Ptr，使用接口类健会导致继承架构被隐藏。显然，我们需要定义六个类：接口类、基类与四个派生类。我们把接口类叫做 Picture，并且用 Ptr 来组织数据。

我们应该用什么类型的 Ptr，或者说这里的 Ptr 管理的是什么类型的对象呢？它管理的是我们的实现类：Pic_base。因此，我们将在类 Picture 中加入一个 Ptr<Pic_base>类型的数据成员。

在前面曾经说过，我们打算隐藏 Pic_base 和其关联继承的可见性，客户只能通过 Picture 类间接地操作对象而不能直接地访问任何一个其他类。能够实现这个目的的最直接途径依赖于正常的保护机制。我们将这些类的 public 接口设置为空，也就是说在 public 接口中没有定义任何的数据或者函数，通过这种方式可以使编译器保证与图形之间的任何作用都将通过 Picture 类发生。

现在让我们先写出下面的代码，它实现了到现在为止我们所讨论的所有功能：

```
//只在执行的时候用到的私有类
class Pic_base { };

class String_Pic:public Pic_base { };
class Frame_Pic:public Pic_base { };
class Vcat_Pic:public Pic_base { };
class Hcat_Pic:public Pic_base { };

//公有接口类与操作
class Picture {
public:
    Picture(const std::vector<std::string>&=
            std::vector<std::string>());
private:
    Ptr<Pic_base> p;
};
```

Picture 对象中有一个私有的 Ptr<Pic_base>类的对象。Pic_base 类是其他四个描述特定图形的派生类的公共基类。Ptr 类用来管理引用计数以实现对基础对象的共享。在 Picture 类中我们将每一个操作通过 Ptr 类型对象转给底层的派生类型对象来实现该操作。至今为止，我们还没有考虑过我们具体进行哪一个操作，Pic_base 类和其派生类的主体部分还仍然是空的。

现在，Picture 类已经十分简单：它只有一个操作，就是用来从一个装有 string 类型对象的容器获得数据，生成一个 Picture 类型对象。我们使用默认参数（见 § 7.3）来使向量参数成为可选的。如果一个用户在构造一个 Picture 类型对象的时候没有提供任何参数，那么编译器将自动用 vector<string>()提供一个参数，它生成一个没有任何元素的 vector<string>对象。因此，默认参数的结果是允许我们使用像下面这样的定义

```
Picture p;          //一个空的 Picture 类型对象
```

来生成一个没有任何元素的 Picture 类型对象。

下面，我们需要考虑一下如何实现 Picture 类的其他操作。我们下面的任务是要实现 frame、hcat 和 vcat 这三个操作函数。我们不但要设计好如何实现这三个函数，还要决定是否把它们定义成 Picture 类的成员函数。因为这些操作都不会改变正在作用的 Picture 类型对象的状态，所以没有必要把它们定义为成员函数。事实上，还有一个更好的理由让我们不这样做：在§12.3.5 里我们看到，如果把它们定义为非成员函数，可以允许我们使用自动类型转换。

例如，因为 Picture 类的构造函数没有被声明为 explicit（显式）的，所以用户可以写出下面的语句：

```
vector<string> vs;
Picture p=vs;
```

在执行的时候，系统会自动为我们把 vs 对象转换成一个 Pictuere 类型对象。如果我们希望发生这种转换（事实上我们确实希望如此）那么就应该允许用户写出像 frame(vs)这样的表达式。如果 frame 是一个成员函数，那么用户就不能把 frame(vs)写成 vs.frame()，尽管看上去二者是等价的。记住，这种转换不会应用到 . 运算符的左操作数上，因此 vs.frame()这一函数调用将被解释成调用 vs 的 frame 成员函数（事实上它并不存在）。

另外我们认为，允许通过写一个表达式来生成一个复杂的图形对于用户来说十分便利。也就是说如果一个语句写成

```
hcat(frame(p),p)
```

会比写成下面

```
p.frame().hcat(p)
```

的形式要更明了，因为第一个表达式反映了 hcat 函数参数的对称性，而从第二种形式中却看不出来这一对称性。

除了用来生成 Pictuer 类型对象的函数以外，我们还要定义一个输出运算符函数，用它来输出 Picture 类型对象的内容。由此我们进一步充实我们的接口：

```
Picture frame(const Picture&);
Picture hcat(const Picture&,const Picture&);
Picture vcat(const Picture&,const Picture&);
std::ostream& operator<<(std::ostream&,const Picture&);
```

15.1.2 Pic_base 类

下一步我们将计划往 Pic_base 类中加入详细的实现代码。如果回过头来看看最初的代码，你可以看到在前面我们用 vector<string>::size 函数来判断在一个给定的图形中有多少个 string 类型对象，另外我们还写了一个独立的 width 函数（§5.8），这个函数在输出的时候十分有用。在考虑如何显示一个图形的时候要明白，我们希望在派生类中能够实现从 Pic_base 类中继承而来的同样的操作。所以这些操作函数必须定义成虚拟的，这样我们就可以从任

Pic_base 类型对象中得到它有多少行，以及每一行有多宽。此外，因为用户可能使用输出运算符来输出一个 Pic_base 类型对象的内容，所以我们还需要另外一个虚拟函数，用来在一个输出流中显示指定的 Pic_base 类型对象。

在这些操作函数中，我们只需要特别留意 display 函数就行了。显然，display 函数必须有一个参数是用来输出结果的流。那么其他的参数又是什么呢？这需要我们仔细想想 display 函数是如何工作的。在输出一个 Picture 类型对象的时候，这个对象可能是由一个或者几个部分组成的，每一部分都是一个 Pic_base 类派生类的对象。如果我们想输出一个水平连接的图形，那么显然，单个图形的每一行输出都包括几个子图的输出。特别要注意的是，我们不能先输出完一个子图的所有内容，然后再输出另一个子图的所有内容。事实上我们只能先输出某个子图的某一行内容，然后交替地输出其他子图的相应行的内容。由此我们知道，在 display 的参数中必须有一个参数告诉函数需要输出多少行内容。

类似地，在我们显示一个水平连接图形的左子图的时候，我们要求相应的子图调用每一行的 width()函数，向每一行内容加入空格补齐。我们还需要告诉一个包含在 Frame_pic 类型对象中的图形把每一行补齐到最大宽度。另一方面，如果我们要显示一个只包含一个 String_Pic 对象的 Picture 类型对象，或者要显示一个由 String_Pic 类型对象组成的垂直连接的 Picture 类型对象时，那么只要在每一行输出中加入足够多的空格就行了。因此，为了优化函数，我们为 display 增加第三个参数，用来标志是否要在输出中填充空格。

从上面的讨论中我们作出决定，让 display 函数带三个参数：一个是用来生成输出的流；另一个是要输出的行数，还有一个是一个 bool 型变量，表示是否让图形补齐各行的宽度。在作出这一决定以后，我们来开始为 Pic_base 系列的类加入实现代码：

```
class Pic_base{
    //没有公有接口
    typedef std::vector<std::string>::size_type ht_sz;
    typedef std::string::size_type wd_sz;

    virtual wd_sz width() const =0;
    virtual ht_sz height() const=0;
    virtual void display(std::ostream&,ht_sz,bool) const=0;
};
```

我们先来为 size_type 定义一个简写名字，在下面的实现中我们要用到这一简写名。仔细思考一下，我们知道底层数据仍然是一个 vector <string>类型对象，所以可以用 vector<string>类的类型成员 size_type 来表示图形的高度，而 string 类中的类型成员 size_type 可以用来表示图形的宽度。我们把图形高度缩写为 ht_sz，而图形宽度则缩写为 wd_sz。

上面的代码中还为基类定义了虚拟函数，注意到在这里的定义中使用了一种新的形式：每个定义中函数体的部分都用=0 来代替。这一语法表示该虚拟函数没有其他的定义。为什么我们要这样定义这些虚拟函数呢？

为了回答这个问题，让我们先来想想，如果我们试图写出这些虚拟函数，那么这些定义将会是怎么样的。在我们前面的设计中，Pic_base 类只是作为我们各种不同的具体类的公共的、基础的类而存在的。我们可能会在 Picture 类中的某一个操作中生成一个具体图形类的对象，或者从用户给出的一个 vector<string>中生成一个 Picture 类型对象。这些操作不会直接生成一个 Pic_base 类的对象，也不会对这样的对象进行操作。如果根本就没有 Pic_base 类的对象的话，从一个 Pic_base 类型对象中（与一个 Pic_base 类的派生类型对象相对比）获得 height 与 width 成员数据又有什么意义呢？这些操作只有在 Pic_base 的派生类中才需要，这些派生类都是对应于一种具体的图形。对于 Pic_base 类来说，没有必要获得 height 与 widht 数据成员。

C++语言没有要求我们随意捏造出一个定义，而是允许我们声明一个虚拟函数可以是没有定义的。如果不用为一个虚拟函数写出其具体实现，同时也意味着这个类不可能有相应的对象。可能会有该类派生而来的类的对象，但是不会有这个类本身的对象。

为了声明我们不想定义一个虚拟函数的具体实现，我们可以在函数体的地方写=0，就像在上面我们定义 height、width 和 display 函数时做的那样。这种做法实际上是定义了一个**纯虚拟函数**。如果为一个类定义了即使是一个纯虚拟函数，我们也就隐式地声明了该类没有其相应的对象。我们把这种类叫做**抽象基类**。因为它们在一个继承树中只是用来提供抽象的接口的基类。它们是完全抽象的：没有自己的对象。一旦为一个类定义了一个纯虚拟函数，该类就是一个抽象基类，编译器将强行禁止我们为这个类生成相应的对象。

15.1.3 派生类

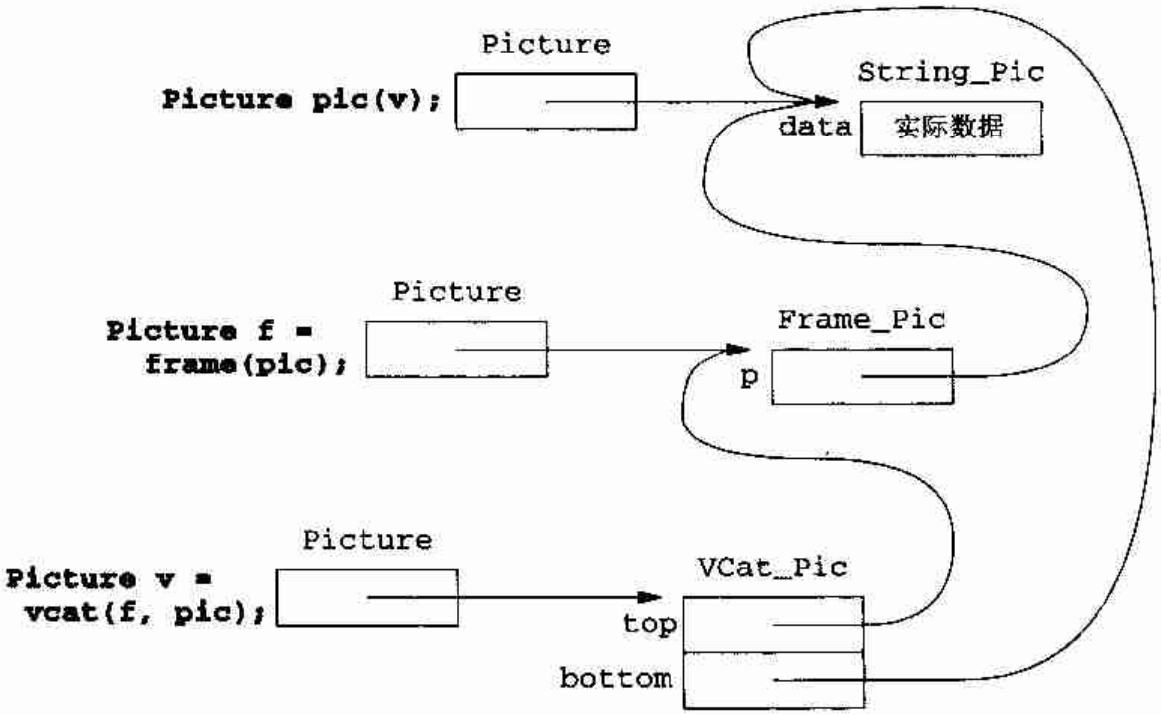
一个函数的纯虚拟特性也可以被继承。如果在一个派生类中定义了所有继承而来的纯虚拟函数，那它就是一个具体的类，我们可以为该类生成它的对象。但是，只要在派生类中有一个继承而来的纯虚拟函数没有进行定义，那么该派生类将继承基类的**抽象特性**，它也是一个抽象类。在这种情况下，派生类本身也是抽象的，我们不能生成该派生类的对象。因为事实上我们的每一个派生类都要用来模拟一种具体的图形，所以我们必须在每一个派生类中重定义所有的虚拟函数。

还有一件事情我们必须现在加以考虑，那就是在我们的派生类中应该包含哪些数据？还有一个相关的问题是我们将如何为不同的类构造对象。我们设计的这些类是用来模拟一个图形形成的结构。图形对象的类型可以告诉我们图形是如何生成的：例如，一个 String_Pic 类的对象是用用户为我们提供的字符数据生成的；一个 Frame_Pic 类的对象是在另一个 Picture 类型对象上加上框架得到的。除了要知道一个对象是如何生成的，我们还要把生成对象的源对象储存起来。对于一个 String_Pic 类，我们在 vector<string>类里把用户提供给我们的字符串保存下来。我们为另一个 Picture 类型对象加上边框，由此得到一个 Frame_Pic 类型对象，因此我们还要把这个 Picture 类型对象保存下来。类似的，我们通过把两个 Picture 类型对象水平（或者垂直）连接生成一个 Hcat_Pic（Vcat_Pic）类型对象，所以要把源对象都保存下来。所有这些类都要把用来生成新对象的 Picture 类源对象保存下来。

在开始设计如何在一个 Pic_base 类的派生类中保存 Picture 类型对象之前，我们先来更深入地思考一下。Picture 类是一个为用户设计的接口类，它只为我们的解决方案提供接口而不是实现。尤其值得一提的是，这个类中没有提供 height、width 和 display 操作函数。想一想这些函数是如何实现的，我们就会知道我们需要对每一个派生类型对象中储存的 Picture 类型对象调用这些相应的操作。例如，为了计算 VCat_Pic 类型对象的 height，我们需要把两个用来构成该对象的 Picture 类型对象的 height 相加。类似地，我们通过比较两个成员 Picture 对象的 width 得到其中较大的宽度，作为 Vcat_Pic 类型对象的 width。

我们要在每一个派生类中保存一个 Picture 类型对象，这要求在 Picture 类中增加用来复制 Pic_base 类操作的函数。不过这么做会使得我们最初的设计思想变得模糊，因为我们原本计划在 Picture 类中只定义接口而没有具体实现。认识到这一点以后，我们决定在类中不是储存一个 Picture 类型对象，而是储存一个 Ptr<Pic_base>类型对象。这种设计就可以使我们把接口与实现清楚地分开来，同时又能对我们的实现对象进行引用计数，以避免不必要的数据复制。

尽管我们的设计方案把接口与实现清楚地分开来了，但是仍然引入了太多的中间层次：



在这里我们假设要生成三个 Picture 类型对象。第一个 Picture 类型对象代表一个储存有从用户那获得的字符数据的 String_Pic 类型对象。第二个 Picture 类型对象代表一个 Frame_Pic 类型对象，我们通过对前面生成的 Picture 对象调用 frame 函数来构造这个对象。最后一个 Picture 类型对象用来表示对前面两个 Picture 类型对象运行 vcat 函数后得到的输出。每个 Picture 类型对象都有一个数据成员 Ptr<Pic_base>，这个 Ptr 类型对象指向某个相应的派生类的

在这些派生类型对象中可能包含着向量类型对象（储存着用户提供的数据的一个复件），也可能包含着一个或者两个 Ptr 类型对象（指向用来生成 Picture 类型对象的 Pic_base 类型对象）。在上图中没有没有标示出与 Ptr 类型对象相关联的引用计数，因为我们假定 Ptr 类已经能完成这项工作，所以忽略了这个工作的实现细节。

与第 5 章中例子的不同之处在于，这里只有 String_Pic 类可以包含字符。其他类中储存着一个或者两个 Ptr 类型对象。因此，我们在生成 f 或者 v 的时候不希望复制任何字符。相反的是，Ptr 可以是用来生成新的 Picture 对象的已有 Picture 对象中 Ptr 的另外一个引用。Ptr 类本身提供了管理引用计数的功能。所以，在调用 frame(pic)函数的时候，结果是生成一个新的 Frame_Pic 类型对象，并且使对象的 Ptr 指向保存在 pic 中的同一个 String_Pic 对象。类似地，Vcat_Pic 类型对象中包含两个 Ptr 类型对象，它们分别指向 Frame_Pic 类型对象和 String_Pic 类型对象。我们不需要亲自删除这些 Pic_base 类型对象，因为 Ptr 类会代我们完成这一任务。不过在最后一个指向某个 Pic_base 类型对象的 Ptr 类型对象被删除以后，我们要亲自为该 Pic_base 类型对象调用 delete 函数删除它。

现在，我们要用代码实现上面的设计思想。我们对于每个对象中应该储存什么数据已经心里有数，写出操作函数如下：

```
class Pic_base{
    //没有公有接口
    typedef std::vector<std::string>::size_type ht_sz;
    typedef std::string::size_type wd_sz;

    //这是个抽象基类
    virtual wd_sz width() const=0;
    virtual ht_sz height() const 0;
    virtual void display(std::ostream&,ht_sz,bool) const=0;
};
class Frame_Pic:public Pic_base {
    //没有公有接口
    Ptr<Pic_base> p;
    Frame_Pic(const Ptr<Pic_base>& pic):p(pic) { }
    wd_sz width() const;
    ht_sz height() const;
    void display(std::ostream&,ht_sz,bool) const;
};
```

这里，我们定义 Frame_Pic 类从 Pic_base 类派生而来，并且打算把从基类中继承而来的三个虚拟函数都定义成具体的函数。因此，Frame_Pic 类不是一个抽象类，我们可以为 Frame_Pic 类生成相应的对象。

值得注意的是，我们是在类的私有接口部分声明这些虚拟函数的。这么做可以让编译器强制实现我们的设计思想，只有 Picture 类与对 Picture 类型对象的操作可以访问 Pic_base

承树。当然，因为这些虚拟函数是私有的，所以我们必须重写 Picture 类的友员类，如果必要的话，还要重写相应的操作函数。

Frame_Pic 类的构造函数只需要把将要被加上框架的源对象复制到 Ptr 类型对象中，这一切是在构造初始化器中进行的。在这里构造函数的函数体是空的，因为我们没有其他的事情要做。

继续看看其他的派生类，连接类以类似于 Frame_Pic 类的方式工作：每个类都需要保存好它的两个源图形。在类中将隐式地定义这两个源图形是如何连接起来的（是垂直连接还是水平连接）：

```
class Vcat_pic:public Pic_base{
    Ptr<Pic_base> top,bottom;
    VCat_Pic(const Ptr<Pic_base>& t,const Ptr<Pic_base>& b):
          top(t),bottom(b) { }
    wd_sz width() const;
    ht_sz height() const;
    void display(std::ostream&,ht_sz,bool) const;
};
class Hcat_Pic:public Pic_base {
    Ptr<Pic_base> left, right;
    HCat_Pic(const Ptr<Pic_base>& l,const Ptr<Pic_base>& r):
          left(l),right(r) { }
    wd_sz width() const;
    ht_sz height() const;
    void display(std::ostream&,ht_sz,bool) const;
};
```

String_Pic 类与其他类有一点点不同：它保存了一个 vector<char>类型对象（这个对象中包含了图形的数据）。

```
class String_Pic:public Pic_base{
    std::vector<std::string> data;
    String_Pic(const std::vector<std::string>& v):data(v) { }

    wd_sz width() const;
    hz_sz height() const;
    void display(std::ostream&,ht_sz,bool) const;
};
```

我们从向量类参数 v 中把底层的字符串复制到 data 数据成员里。这是整个程序中惟一对字符进行复制的地方。在其他的地方，我们只是复制 Ptr<Pic_base>对象，它实际上只是复制了指针，使引用计数器增加一。

15.1.4 复制控制

在我们的设计中也许最有趣之处在于，程序中既没有定义复制构造函数，也没有定义赋值运算符函数和析构函数，为什么？

原来编译器会自动为我们合成相应的默认函数。用户为我们提供字符以生成一个新的 Picture 类型对象，向量类负责管理这些字符的第一次复制。如果我们对两个指向 String_Pic 类型对象的 Picture 类型对象进行复制或者赋值操作，那么 Ptr 的操作函数将会负责管理这些 Picture 类型对象，并负责在适当的时候删除底层 String_Pic 类型对象。更一般的是，Ptr 类负责对其他 Pic_base 类中的 Ptr 类成员对象进行复制、赋值和删除操作——对 Picture 类本身里的 Ptr 类成员对象也如此。

15.2 实现

现在我们对接口类和实现类都已经做了充分的设计。Picture 类和相应的对 Picture 类型对象的操作都是用来管理用户接口的。Picture 类的构造函数与其操作将生成一个 Pic_base 类派生类的对象。我们用 Ptr<Pic_base>来管理底层的空间，这样可以避免进行没有必要的数据复制。现在到了为接口类及其派生类的操作写出实现代码的时候了。

15.2.1 实现用户接口

我们先来实现接口类与它的操作函数。现在我们有

```
class Picture{
public:
    Picture(const std::vector<std::string>&=
        std::vector<std::string>());
    private:
        Ptr<Pic_base> p;
};
Picture frame(const Picture&);
Picture hcat(const Picture&,const Picture&);
Picture vcat(const Picture&,const Picture&);
std::ostream& operator<<(std::ostream&,const Picture&);
```

让我们先来考虑一下生成一个新的 Picture 类型对象的操作吧。这些操作用来为一个 Pic_base 类的派生类生成一个对象。这个对象将从一个 Picture 类型对象中复制 Ptr 对象，我们将把一个 Picture 类型对象绑定到这个新建的 Pic_base 类型对象上，并返回该 Picture 类型对象。例如，如果 p 是一个 Picture 类型对象，那么 frame(p)将生成一个新的 Frame_Pic 类型对象（该对象隶属于 p 中的 Pic_base 类型对象），然后生成一个新的 Picture 类型对象（该对象隶属于前面的这个 Frame_Pic 类型对象）。请看下面的程序代码：

```
Picture frame(const Picture& pic)
{
    Pic_base* ret=new Frame_Pic(pic.p);
    //我们将返回什么呢?
}
```

我们先定义一个 Pic_base 类型的局域指针，然后生成一个新的 Frame_Pic 类型对象来初始化它，该 Frame_Pic 复制了 pic 类型对象中的 Ptr 类型对象。现在出现两个问题。比较简单的那个问题是，Frame_Pic 类的构造函数是私有的。在 § 15.1.3 中我们看到，每一个 Pic_base 类都是一个隐藏类。我们不希望让用户知道这些类的存在，所以只定义私有的操作，让编译器来强制执行这一思想。通过定义 frame 操作为 Frame_Pic 类的一个友员函数可以解决这个简单的问题。

另外还有一个比较麻烦的问题：我们已经生成了一个 Frame_Pic 类的新对象，但是实际上需要的是一个 Picture 类型对象。可以想像，hcat 函数、vcat 函数以及下面将要写的几个函数都会生成 Pic_base 类派生类的对象，而且这些函数即使在我们需要生成 Picture 类型对象的时候仍然会这么做。幸运的是，在 § 12.2 中我们已经知道，如果提供一个合适的构造函数，我们可以把一种类型转换成另一种类型。在这里，这个合适的构造函数通过一个 Pic_base*指针来构造一个 Picture 类型对象：

```
class Picture{
    Ptr<Pic_base> p;
    Picture(Pic_base* ptr):p(ptr) { }
    //其他部分同前
};
```

这个构造函数用一个指向 Pic_base 类型对象的指针来初始化 p。记住，Ptr 类有一个带有一个 T*类型参数（在这里是一个 Pic_base*类型的参数）的构造函数。初始化器 p(ptr)激活 Ptr::Ptr(T*)构造函数，把 ptr 类型对象传递给它。有了这个 Picture 类的构造函数以后，我们就能把 frame 操作函数写出来：

```
Picture frame(const Picture& pic)
{
    return new Frame_Pic(pic.p);
}
```

我们把局域的 Pic_base 类型对象删除掉（因为以后再也用不着它了），然后生成一个新的 Frame_Pic 类型对象，这个对象的地址自动转换成一个 Picture 类型对象，并把这个 Frame_Pic 类型对象作为函数返回值。要想彻底理解这个小函数，你必须充分理解自动类型转换与复制构造函数的精妙之处。这个简单的语句与下面的语句作用是相同的：

```
//生成一个新的 Frame_Pic 类型对象
Pic_base* templ=new Frame_Pic(pic.p);
```

```
//用一个 Pic_base*指针构造一个 Picture 类型对象
Picture temp2(temp1);

//返回一个 Picture 类型对象，这将激活 Picture 类的复制构造函数
return temp2;
```

像 frame 函数一样，连接函数依赖于 Picture 类的一个新的构造函数：

```
Picture hcat(const Picture& l, const Picture& r)
{
    return new HCat_Pic(l.p, r.p);
}
Picture vcat(const Picture& t, const Picture& b)
{
    return new VCat_Pic(t.p, b.p);
}
```

在上面的每个函数中，我们都构造一个合适类型的对象，并把一个 Ptr<Pic_base>类型对象与之捆绑在一起，用这个 Ptr<Pic_base>类型对象构造一个 Picture 类型对象，然后返回该 Picture 类型对象的一个复件。当然，为了编译这些函数，我们还需要向 HCat_Pic 类与 VCat_Pic 类中加入合适的友员声明。

为了用一个 vector<string>类型对象来构造一个 Picture 类型对象，我们采用与其他类型的图形相同的办法：

```
Picture::Picture(const vector<string>& v):
    p(new String_Pic(v)) { }
```

我们再一次生成一个新的 String_Pic 类型对象，不过这一次我们直接用它来初始化 p，而不是返回该对象。当然，我们还要记住把 Picture 类定义成 String_Pic 类的一个友员类，这样就可以通过 Picture 类访问 String_Pic 类的构造函数了。

理解构造函数与 frame、hcat 和 vcat 函数为什么不同十分重要。这几个函数都被定义成返回一个 Picture 类型对象，而且在每一个函数中都使用了一个指向某 Pic_base 类派生类型对象的指针。因此，我们隐式地使用 Picture(Pic_base*)的构造函数来生成一个 Picture 类型对象并作为函数的返回值。在刚才写的 Picture 类的构造函数中，我们还生成一个 Pic_base 类派生类的指针（在这里是一个 String_Pic 类的指针），但是现在我们使用这个指针来初始化 p 成员，p 是 Ptr<Pic_base>类型的对象。这样调用了 Pic_base 类中的 Ptr(T*)构造函数，而不是 Picture(Pic_base*)构造函数，因为我们是在构造一个Ptr<Pic_base>类型对象，而不是一个Picture 类型对象。

为了实现接口函数，我们必须定义输出运算符函数。这个函数很简单：我们需要遍历底层的 Pic_base 类型对象，然后调用 display 函数来显示每一行输出：

```
ostream& operator<<(ostream& os,const Picture& picture)
{
    const Pic_base::ht_sz ht = picture.p->height();
    for(Pic_base::ht_sz  i=0; i!=ht; ++i) {
        picture.p->display(os, i, false);
        os<<endl;
    }
    return os;
};
```

我们为底层的 Pic_base 类型对象调用虚拟的 height 函数来初始化 ht，这样就不用在每一次循环里都重新计算图形高度。记住，p 实际上是一个 Ptr<Pic_base>类型对象，而在 Ptr 类中重载了->运算符以把一个 Ptr 的引用变成一个 Ptr 包含的指针的引用。循环 ht 次，每次调用 display 虚拟函数输出当前行。第三个参数（false）表明 display 函数不需要补齐每一行输出。如果指定要以补齐的方式来输出一个内部的 Picture 类型对象，那么 display 函数将会用空格把每一行输出补至相同宽度。目前我们还不知道要不要对每行输出用空格补齐。在每一行输出的后面都写一个 endl，最后返回 os。

除了已经实现的操作，我们还要记住为 Pic_base 加一个友员声明，这样就可以让 operator<< 函数访问类中的 display 函数与 height 函数。

15.2.2 String_Pic 类

在前面我们实现了接口类与其操作，现在来看一看它的派生类。我们先来看看 String_Pic 类：

```
class String_Pic:public Pic_base{
    friend class Picture;
    std::vector<std::string> data;
    string_Pic(const std::vector<std::string>& v):data(v) { }
    ht_sz height() const  {return data.size(); }
    wd_sz width() const;
    void display(std::ostream&,ht_sz,bool) const;
};
```

在上面的代码中，除了 height 函数，其他部分仍然与 § 15.1.3 中 String_Pic 类的版本一样。height 函数的函数体很简单，它把工作交给向量类的 size 函数来完成。

为了得到 String_Pic 类的 width 函数，我们先来看看 data 中的每一个元素，看看哪一个元素是最长的：

```
Pic_base::wd_sz String_Pic::width() const
{
    Pic_base::wd_sz n=0;
    for(Pic_base::ht_sz i=0; i!=data.size(); ++i)
```

```
        n=max(n,data[i].size());
    return n;
}
```

除了类型名不同以外，这个函数的其他部分与§5.8 的版本是一样的，这是因为在String_Pic 类型对象中储存了一个 vector<string>类型对象。

display 函数就稍微复杂一点。它遍历底层的容器，输出指定行的字符串。

现在再来看看关于补齐行的问题。注意到这个函数有可能在使用输出算符的时候被直接调用，这种情况发生在 Picture 类型对象指向一个 String_Pic 类型对象的时候：这个函数还可能在输出一个包括 String_Pic 类型对象的更大的 Picture 类型对象的时候被间接调用。在第二种情况下要求 display 函数为每行补齐空格以使输出的每一行具有相同的大小。补上的空格数随每一行字符串的大小而变，我们要为短的字符串补上足够数量的空格以使它的长度与最长的字符串相同。换句话说，我们要把每个字符串的长度补齐到 String_Pic.width 的大小。从长远来考虑，我们最好也为其他的图形补齐各行。现在假定我们已经有一个 pad 函数，它带有三个参数：第一个参数是输出流；第二个参数是要补上空格的首位置；第三个参数是要补上空格的末位置（后面的位置）。我们将简单地实现这个函数。

程序还有一个麻烦之处，传递给 display 函数的行数参数有可能会大于 String_Pic 类型对象图形的高度。如果 String_Pic 类型对象是一个水平连接起来的图形的一部分，这个图形的一边可能小于另一边，那么就可能会出现上面所说的麻烦的情况。我们的两个子图形是顶对齐的，但是它们的高度可能不同。所以，有必要检查一下要输出的行是不是超出范围了。经过这么一分析，我们可以写出 display 函数的代码来：

```
void String_Pic::display(ostream& os ht_sz row, bool do_pad) const
{
    wd_sz start=0;

    //如果 row 没有超出范围，就输出第 row 行
    if(row<height()) {
        os<<data(row);
        start=data[row].size();
    }

    //如果必要的话补齐输出各行
    if(do_pad)
        pad(os,start,width());
}
```

我们首先检查指定要输出内容的行号 row 是否超出范围——也就是说，row 是不是大于String_Pic.height()，如果没有超出范围，就输出该行内容并设置 start 使之指示要写多少个字符。无论要不要输出一行字符串内容，我们都要判断是否要补齐输出结果。如果要补齐，我
从 start 开始直到 String_Pic.width()的地方都填充上空格。如果行号超出了范围，start 的

0，这时候我们就输出一行空格。

15.2.3 补齐输出结果

现在来看看用来补齐输出的函数。因为希望可以从任何一个派生类中访问这个函数，所以我们把这个函数定义为 Pic_base 类的一个成员函数，并且把它定义成静态的（static）和保护类型的（protected）：

```
class Pic_base{
    //其他部分同前
protected:
    static void pad(std::ostream& os, wd_sz beg, wd_sz end) {
        while(beg!=end) {
            os<<" ";
            ++beg;
        }
    }
};
```

这个函数有一个 ostream 参数，函数用它来输出空格，还有两个其他参数，是用来控制要输出多少个空格的。如果一个显示函数要调用 pad 函数，它将把当前的列号和需要在当前的 display 函数中填充空格的最后一列后面一列的列号传递给 pad 函数。pad 函数将在这一范围内填满空格。

请注意在 pad 的声明中用到的 static 关键字。在 § 13.4 中我们已经介绍过该关键字的用法，static 的使用表示 pad 函数是一个静态的成员函数。这种函数与一个普通的成员函数不同，因为这种函数不隶属于类的某个对象。

令人惊讶的是，我们还可以为一个抽象基类定义一个成员函数。因为我们很难想像，如果一个基类不能有相应的对象，那它为什么还可以拥有成员函数呢？但是请记住，每个派生类的对象都包含有一个基类的部分。在每一个派生类中还继承了基类中定义的所有成员函数。因此，一个基类的函数将在一个派生类型对象的基类部分中被调用。在这种特殊的情况下，我们定义的函数是一个静态的函数，所以访问基类成员函数的问题是没有实际意义的。不过我们要认识到在抽象类中既可以定义静态函数，也可以定义成员数据和普通的成员函数，这一点很重要。这些函数将用来访问一个派生类型对象中的基类部分。

静态成员（包括静态函数与静态数据）十分好用，因为它们可以让我们尽量少定义全局的函数或者变量。这里的 pad 函数就是一个很好的例子。我们可以在很多情况下用到补齐输出的方法。本书中我们讲到在输出指定格式的学生成绩报表的时候用到补齐输出的思想，在输出 Picture 类型对象的时候也用到该思想。如果 Picture 类把 pad 函数定义成一个全局函数，那么我们就不能在 Student_info 类中定义 pad 函数，反之亦然。通过把 pad 定义成一个静态函数，我们还可以在其他的地方应用补齐输出的思想。如果每个类都只在类的内部定义 pad 函

数体，那么一个程序中就可以在几个互相独立的不同地方同时应用补齐的思想。

15.2.4 VCat_Pic 类

在实现连接图形的类并不难。下面是 VCat_Pic 类的代码：

```
class VCat_Pic:public Pic_base
{
    friend Picture vcat(const Picture&,const Picture&);
    Ptr<Pic_base> top, bottom;
    VCat_Pic(const Ptr<Pic_base>& t, const Ptr<Pic_base>& b):
        top(t),bottom(b) { }
    wd_sz width() const
        { return max(top->width(),bottom->width()); }
    ht_sz height() const
        { return top->height() + bottom->height(); }
    void display(std::ostream&,ht_sz,bool) const;
};
```

我们为 vcat 函数添加一个友员声明，并且以内联子过程的方式实现 height 函数与 width 函数。如果一个 Picture 类型对象代表的图形是两个源图形垂直连接而成的，那么它的高度将是子图形的高度之和，而其宽度是两个子图形宽度的较大值。

display 函数也并不难：

```
void VCat_Pic::display(ostream& os, ht_sz row, bool do_pad) const
{
    wd_sz w=0;
    if(row<top->height()) {
        //现在处于上面的子图形里
        top->display(os,row,do_pad);
        w=top->width();
    } else if(row<height()) {
        //现在处于下面的子图形里
        bottom->display(os,row-top->height(),do_pad);
        w=bottom->width();
    }
    if(do_pad)
        pad(os,w,width());
}
```

首先，我们定义一个 w 变量，这个变量在需要为某行补齐空格的时候用来储存当前行的行宽。然后通过比较行号 row 与上子图的高度，判断是不是处于上子图的位置。如果是在上子图的范围里，就向 display 函数传递一个 bool 型变量以指示是否补齐每行的输出，并激活 display 函数输出上子图的内容。记住，display 是一个虚拟函数，所以对它的调用实际上激活

的是上子图指向的具体某个 Pic_base 类型对象的 display 函数。在输出了第 row 行的内容之后，我们也同时保存了该行的行宽 w。

如果当前行号不是在上子图的范围，那就一定是在下子图中。在 else 的条件判断中，我们还要判断行号 row 是不是在整个图形的范围里，所以判断 row<height()是否成立。如果成立，说明现在处于下子图的范围内。与在上子图里一样，我们在下子图的范围里也调用 display 函数输出图形，不过在把行号作为参数传递给 display 函数的时候，要把 row 减去上子图的高度，得到下子图中的相对行号。在输出下子图中的某一行内容之后，保存该行的行宽 w。如果行号 row 超出了整个图形的范围，就令 w 为 0。

在输出第 row 行的内容以后，我们判断该行是否需要补齐。如果需要，就把只有 w 个字符宽的该行补上空格，使之达到整个图形的宽度。

15.2.5 HCat_Pic 类

看到下面的 HCat_Pic 类与前面的 VCat_Pic 类十分相似，大家应该不会觉得奇怪吧：

```
class HCat_Pic:public Pic_base{
    friend Picture hcat(const Picture&, const Picture&);
    Ptr<Pic_base> left, right;
    HCat_Pic(const Ptr<Pic_base>& l, const Ptr<Pic_base>& r):
         left(l), right(r) { }
    wd_sz width() const { return left->width() + right->width(); }
    ht_sz height() const
       { return max(left->height(), right->height()); }
    void display(std::ostream&, ht_sz, bool) const;
};
```

因为现在是把两个子图水平地连接在一起，所以整个图形的宽度 width 是两个子图形宽度之和，而图形高度 height 就是两个子图高度中的较高者。这里提供的 display 函数比 VCat_Pic 类中相应的函数要简单得多，因为在这里我们不用去判断行号 row 是在哪个子图的范围里：

```
void HCat_Pic::display(ostream& os, ht_sz row, bool do_pad) const
{
    left->display(os,row,do_pad||row<right->height());
    right->display(os,row,do_pad);
}
```

首先，我们对 left 对象调用 display 函数输出左子图的指定行。如果程序指定要补齐输出，那么就要把该行补上足够的空格；或者如果在输出左子图的某行之后紧跟着要输出右子图的内容，那也要进行补齐输出，因为只有这样才能保证右子图该行的输出是从正确的位置开始的。如果行号 row 超出了左子图的范围， 在 display 函数中会自己解决该问题。类似的，在输出右子图某行内容的时候通过调用 right 对象的 display 函数来实现。不过在这里直接把 do_pad 参数传递给函数就可以了，因为没有充分的理由要求在输出右子图时一定要进行补齐输出

15.2.6 Frame_Pic 类

现在还剩下一个派生类没有实现：

```
class Frame_Pic:public Pic_base {
    friend Picture frame(const Picture&);
    Ptr<Pic_base> p;
    Frame_Pic(const Ptr<Pic_base>& pic):p(pic) { }

    wd_sz width() const { return p->width()+4; }
    ht_sz height() const { return p->height()+4; }
    void display(std::ostream&, ht_sz, bool) const;
};
```

height 函数与 width 函数都把计算工作交给未加框的源图形 p 来做。在求得源图形的高度或者宽度以后再加上 4 作为已加框图形的高度或者宽度，之所以要加上 4，是因为要加上边界与空格的宽度。

display 函数很冗长乏味，但是一点也不难：

```
void Frame_Pic::display(ostream& os, ht_sz row, bool do_pad) const
{
    if(row>-height()) {
        //超出范围
        if(do_pad)
            pad(os,0,width());
    }else {
        if(row==0||row==-height()-1) {
            //在最顶行或者最底行
            os<<string(width(),'*');
        }else if(row==1||row==-height()-2) {
            //在第二行或者倒数第二行
            os<<"*";
            pad(os,1,width()-1);
            os<<"*";
        }else {
            //在内部图形中
            os<<"*";
            p->display(os,row-2,true);
            os<<"*";
        }

    }
}
```

在这个函数中，我们首先检查指定的行号是否在图形的范围内：如果不是而且又指定要补齐输出，那么就把整行输出为空格。如果行号在图形的范围内，程序可能面临三种情况

出的是顶行或者底行的边界；要输出的是边界与内部图形之间的空格行；要输出的是内部图形的某行。

我们知道如果行号 row 等于 0 或者等于 height()-1，那么它指向的是最顶行或者最底行。这时候，我们为整行输出星号字符（*）来生成边界。如果行号指示我们处于第二行或者倒数第二行，那么输出的内容将由一个星号字符加上一连串的空格字符，最后再加上一个星号字符组成。最后，在行号指示要输出内部的图形时，我们先输出图形的左边界（由一个星号字符与一个空格字符组成），然后输出内部图形中该行的内容，最后输出图形的右边界（由一个空格和一个星号字符组成）。输出内部图形内容是通过调用 display 函数来实现的，为了得到内部图形中的相对行号，我们把 row 减去 2，然后把这个相对行号传递给 display 函数。在调用 display 函数的时候，我们指示函数无论什么情况下都要对源图形的输出进行补齐，因为这样才能保证输出的右边界是一条直的竖线。

15.2.7 不要忘了友元类声明

现在还有一件事没有做，那就是向 Picture 类和 Pic_base 类中添加友员声明。我们以前说过要为 Picture 类的每一个操作函数添加一个友员声明。因为这些操作都要用到 Picture 类型对象里的 Ptr 类型对象，所以必须获得访问 Ptr 类成员的权力。但是向 Pic_base 类中加入一组友员声明就不是那么显而易见的了：

```
//对 Picture 类进行前置声明，关于前置声明的内容将在§15.3 介绍
class Picture;

class Pic_base{
    friend std::ostream& operator<<(std::ostream&, const Picture&);
    friend class Frame_Pic;
    friend class HCat_Pic;
    friend class VCat_Pic;
    friend class String_Pic;
    //没有公有接口
    typedef std::vector<std::string>::size_type ht_sz;
    typedef std::string::size_type wd_sz;
    //这是个抽象基类
    virtual wd_sz width() const =0;
    virtual ht_sz height() const=0;
    virtual void display(std::ostream&, ht_sz, bool) const=0;
protected:
    static void pad(std::ostream& os, wd_sz, wd_sz);
};
class Picture{
    friend std::ostream& operator<<(std::ostream&, const Picture&);
    friend Picture frame(const Picture&);
    friend Picture hcat(const Picture&, const Picture&);
```

```
    friend Picture vcat(const Picture&, const Picture&);
public:
    Picture(const std::vector<std::string>&=
        std::vector<std::string>());
private:
    Picture(Pic_base* ptr):p(ptr) { }
    Ptr<Pic_base> p;
};
//对 Picture 类型对象进行操作
Picture frame(const Picture&);
Picture hcat(const Picture&, const Picture&);
Picture vcat(const Picture&, cosnt Picture&);
std::ostream& operator<<(std::ostream&, const Picture&);
```

Pic_base 类中的第一个友员声明很容易理解。输出运算符会激活 height()函数与 display()函数，所以要求该运算符函数可以访问这些函数。比较费解的是对 Pic_base 类的派生类的友员声明。难道它们不能通过继承获得访问 Pic_base 基类成员的权力吗？是的，它们确实因为继承而能访问 Pic_base 类中的某些成员，但仅限于拥有访问 Pic_base 类的保护成员的权力。那么为什么我们不像定义pad函数那样把这些成员函数定义为Pic_base类的保护成员呢？答案是仅仅这么做并不能解决问题。

派生类只能访问类的对象自身拥有的保护成员，但是它们没有访问其他对象的保护成员的特权。我们设计的继承架构中的每个派生类只能访问自己本身所包含的 Pic_base 部分中的保护成员。例如：

```
ht_sz Frame_Pic::height() const { return p->height()+4; }
```

Frame_Pic 类的这个函数实际上调用了 p 对象的 height 成员函数。因此，即使 height 已经是一个保护成员，我们还是要通过友元声明才能允许进行这种访问。

这个规则也许会令你感到惊讶，但是从逻辑上来讲这是十分简单的：如果语法允许一个派生类访问任何一个对象的保护成员，那么将会破坏这种保护机制。如果在需要访问某个类中的保护成员的时候，我们可以写一个新的类作为该类的派生类，然后在这个新生成的派生类中定义一个成员函数来访问基类中的保护对象。这么做会破坏类的设计者们最初的保护机制。因此，对保护成员的访问权力仅限于该对象，而不允许其他的对象访问该对象的保护成员。

15.3 小结

抽象基类中有一个或者多个纯虚拟函数：

```
class Node{
    virtual Node* clone() const=0;
};
```

上面的代码声明 clone 函数是一个纯虚拟函数，这也就意味着 Node 类是一个抽象基类。我们不能生成一个抽象基类的对象。一个类可以通过继承而成为抽象类：如果在一个派生类中没有对所有继承而来的纯虚拟函数进行重定义，也就是说哪怕还有一个纯虚拟函数存在，这个派生类就是一个抽象类。

前置声明：有时候会有两个类在定义中互相用到了对方的定义，这时候应该把哪个定义放在前面呢？为了解决这个问题，你可以先写出下面的语句：

```
class class-name;
```

这个语句声明了一个 class-name 类，但是还没有具体实现它，不过你可以在接下来定义另一个类的时候直接使用 class-name 类。

在 § 15.2.7 中我们提到了一个前置声明的例子。Picture 类的定义中要用到 Ptr<Pic_base> 类的一个成员，而 Pic_base 类中对 operator<<函数的友员声明中用到了 const Picture&的类型。因此，这两个类互相引用，这种时候就适合使用前置声明的方法。

像这样两个类互相使用互相依赖有可能使程序无法正常工作。例如：

```
class Yang;   //预声明
calss Yin {
    Yang y;
};
class Yang{
    Yin y;
};
```

在这里，每个 Yin 类型对象都包含一个 Yang 类型对象，而在 Yang 类型对象中又包含一个 Yin 类型对象，而这个 Yin 类型对象中又包含着一个 Yang 类型对象.....这么一来，要实现这种类型就需要无穷大的内存。

在我们的图形类中存在这种相互依赖性，但是却不会导致上面所说的问题，因为 Picture 类中没有直接包含一个 Pic_base 类的对象作为数据成员。它包含的是一个 Ptr<Pic_base>类的对象成员，而这个成员只是包含了一个 Pic_base*类型的指针。对指针的这种使用避免了对象之间无穷的嵌套使用。

另外，对于一个指针（或者一个引用），编译器事先并不关心指针类型的实现细节，直到通过该指针（或者引用）来调用一个函数的时候才会关心指针类的具体内容。因为在 operator<<函数的声明中使用 const Picture&类型只是作为一个参数类型，所以编译器只需知道 Picture 是一个类型。而该类型的实现细节在定义 operator<<之前编译器并不关心。

习题

15-0 编译、运行并测试本章举的例程。

15-1 写一个测试程序，执行下面的语句：

```
picture p=//对源图形进行初始化
picture q=frame(p);
picture r=hcat(p,q);
picture s=vcat(q,r);
cout<<frame(hcat(s,vcat(r,q)))<<endl;
```

15-2 重写 Frame_Pic 类，在为源图形加上边框时使用三种字符：一种字符用于边框的四个角；另一种字符用于顶边与底边；还有一种字符用于两条垂直的边。

15-3 改写上面所说的类，让用户可以自己选择用什么字符来形成边框。

15-4 加入一个操作函数，用来为一个图形重加边框，它可以改变边线的字符。这个函数要求改变内部图形的所有边框。

15-5 重写 HCat_Pic 类，要求不同大小的子图形在连接的时候，小的那个子图形必须在大图形的中间位置与大图形连接。也就是说，如果我们水平地连接两个子图形，并且其中一个子图形有四行，而另一个子图形有两行，那么在输出这个小图形的时候，要在它的顶上与底下各多输出一行空格。请注意这时候某行内容在子图形中的相对行号与以前的版本不同。

15-6 我们在第 11 章与 12 章中写的 Vec 类与 Str 类也可以用来实现 Picture 类。在本章写的类中用 Vec<Str>类来代替 vector<string>类，并编程对你写的类进行测试。

第 16 章

今后如何学习 C++

现在本书要讲述的内容已经结束。然而我们所讲的东西还肤浅得很：在 C++语言中我们还有些重要的内容没有讲，还有 C++标准库的大部分内容我们都没有在本书中提及。但是，本书仍然到此打住，不打算再深入地讲下去了，其原因有二。

其中一个原因是现在我们已经掌握了足够多的知识与技巧，用它们已经可以解决在编程中遇到的大部分问题。在学习 C++的其他技巧之前建议读者先消化一下本书学到的知识，通过把它们用于实践中解决些实际的问题来牢牢地掌握这些知识。我们建议读者重新阅读一下本书，把在第一次阅读本书的时候没有完成的习题做完。

如果你想学习一下编程的良好风格与一些巧妙的编程技巧，建议你学习一下《Ruminations on C++》（Addison-Wesley 出版社，1997）一书，这本书讲述了一些 C++编程的风格问题并提供了一些编程的实例。

本书到此打住的另一个原因是，如果你能应用本书所讲的知识，在实践中写出自己的程序，那么你就可以不需要像本书这么详尽的讲解了，你已经从本书中汲取了足够的知识，获得了足够的编程技巧，可以看多些专注于细节内容，少点解释的书了。

16.1 好好地利用你已经掌握的知识

有一个古老的故事，说的是一个初到纽约市的访问者，这个访问者在纽约市里迷了路。他手拿一张钢琴演奏会的门票向一个过路人问路："请问，去卡耐基礼堂的路怎么走？"过路人回答说："练习！"

在学习 C++中的新知识之前，巩固并彻底理解如何使用在本书中学到的知识很重要。你可以把从标准库中学习到的知识以及其他的一些知识应用到自己的编程中去解决实际的问题，这样往往能写出简短而高效的程序。值得一提的是，如果在设计阶段能利用 C++的一些抽象概念设计好整个程序的框架，那么在实际动手解决问题的时候往往会得心应手。

现在举一个例子。在第 13 章中我们曾经写过一个用来储存学生成绩的类，在第 15 章中写过用来生成字符图形的类。这个字符图形的类已经被使用了很多年了，只不过有时候会适当地改变

下它的形式。相反，学生成绩的类只是本书中的一个特殊的例子。仔细想想本章说的东西，也许你能够把这两个类中毫不相干的抽象概念结合在一起，得到一个很好的解决问题的方案。

我们可以把字符图形的概念引入学生成绩的类中，把不同学生的成绩输出成柱状图。通过这样的图形输出，我们可以很直观地比较出不同成绩的差别，在数字表示的成绩表中就没有这种效果。基本的思想是把每个学生的期末考试成绩转换成一个等号字符串，该字符串的长度与成绩分数成正比。例如，对于某个适当的成绩表，我们可能会得到下面的图形输出：

```
*********************************************
*                                           *
* James      ====================           *
* Kevin      ===================            *
* Lynn       ==================             *
* MaryKate   ========================       *
* Pat        ==========                     *
* Paul       ==========================     *
* Rhia       ==================             *
* Sarah      ================               *
*                                           *
*********************************************
```

从上面的柱状图中，我们可以很直观地看到，Pat 这个学生需要加加倍努力了。

更让人叫绝的是这个例子十分短小简洁，并且很直接地解决了这个问题：

```
Picture histogram(const vector<Student_info>& students)
{
    Picture names;
    Picture grades;
    //对于每个学生
    for(vector<Student_info>::const_iterator it=students.begin();
        it!=students.end();++it) {
        //生成不同的姓名（或者成绩）垂直连接图形
        names=vcat(names,vector<string>(1, it->name()));
        grades=vcat(grades,
            vector<string>(1, " " + string(it->grade()/5, '=' )));
    }
    //水平连接姓名图形与成绩图形
    return hcat(names,grades);
}
```

上面的 histogram 函数带有一个引用参数，这个引用指向一个 Student_info 类型对象中的向量类型对象，向量类型对象中包含的每个对象都代表一个学生。从学生记录中我们可以生成两个 Picture 类型对象：一个是 names；另一个是 grades。names 对象中包含了所有学生的姓名；而 grades 对象中包含了所有学生成绩。然后我们把这两个图形水平地连接在一起，把每个学生的姓名和他的成绩排成直线一一对应。因为每个图形都是一个长方形，所以在把它们

连接起来的时候要自动地处理不同学生姓名的长度。

在主函数中定义了一个向量类型对象来读取一个保存着学生记录的文件并保存起来。读完以后，程序调用 histogram 函数来生成一个 Picture 类型对象，为图形加上边框，并调用输出算符函数输出结果：

```
int main()
{
    vector<Student_info> students;
    Student_info s;
    //读取姓名与成绩
    while(s.read(cin))
        students.push_back(s);
    //对学生姓名按字母排序
    sort(students.begin(),students.end(),Student_info::compare);
    //输出名字与柱状成绩图
    cout<<frame(histogram(students))<<endl;
    return 0;
}
```

本例中最重要的新思想是，它实际上没有用到任何新的知识！之所以能这么容易而且出色地解决这个问题，只不过是因为我们对已经学过的知识十分熟悉，并且可以出人意料地把这些知识结合在一起。而要想更加熟悉已经学过的知识，我们只有不断地练习，练习，再练习！

16.2 学习更多的东西

在前面，我们建议读者通过不断练习充分理解本书讲的要点。可是读者可能还希望知道关于 C++语言与其标准库的更多的细节，而很多知识只通过练习是很难学到的。还有的时候，你可能会面临一个难题，这个问题光是利用本书介绍的知识还不能解决，那么你就要看些其他的书来寻找答案了。

在这里笔者向读者推荐两本好书。第一本好书是 Bjarne Stroustrup 著的《The C++ Programming Language》第三版（Addison-Wesley，1998）。这本书基本上都是 C++的源代码，涵盖了整个 C++的语法和标准库，在书中你可以找到关于语法与库的所有问题的答案。第二本书好书是 Matthew Austern 著的《Generic Programming and the STL》(Addison-Wesley，1999)。这本书详细地讨论了标准库的使用法则以及数据结构，比本书和 Stroustrup 的书都要详细得多。另外，这本书是 C++语言方面的权威书籍，虽然 Austern 没有负责 C++标准库的撰写（标准库是由 Alex Stepanov 编写的），但是他在过去几年里曾经与 Stepanov 紧密地合作了好几年的时候，可以这么说，Austern 是标准库进一步发展的源动力之一。

本书、《Ruminations on C++》，还有 Stroustrup 和 Austern 的两本书，加起来已经有 2000 页之多。所以我们不想再向读者推荐其他的书了。不过如果你真的阅读欲望十分旺盛，

看些相关的好书的话，我们建议你访问网站 http://www.accu.org。在那里，你可以找到至少 2000 本书的评论，很多书都是 C++的。该网站也对《Ruminations on C++》一书和 Stroustrup 与 Austern 的书作了很高的评价。

在寻找这些书的时候请牢记一点，摆在书架上的书不会使你成为一个好的程序员。最后还要指出一点，提高你的编程技巧的惟一途径是编程。

祝你在 C++的学习过程中学得轻松，学得开心！

习题

16-0 编译、运行并测试本章的例程。

16-1 写一个自我复制的程序，这个程序没有输入，它在运行的时候复制一份自己的源代码，并把代码写入输入流里。

附录 A

C++语法细节

本附录包括两方面的内容：它增加了一些关于底层细节的知识，另外还对 C++语言的表达式与语句作了些总结，包括一些本书其他地方没有用到的知识。这些底层的细节主要是关于 C++复杂的声明语法，还有些是关于 C++自带的算术类型的详细知识。这两方面的语法都是从 C 语言中继承而来。它们对于理解本书的内容并不是必要的，要写出一个好的 C++程序不一定需要掌握这些细节。但是，有很多的程序用到这方面的知识，所以了解一下这些细节十分有用。

在本附录中我们遵循下面的约定来描述语法：常体字（正常字体的）表示它们本身，斜体字表示语法种类，... 表示在后面跟着零个或者一个或者多个重复的项，而斜体中方括号里的内容是可选的。另外，我们还使用斜体花括号来表示分组，用竖线 | 表示二者中选择其中一项。例如：

```
declaration-stmt : decl-specifiers [declarator [initializer]] [,declarator [initializer]]...;
```

表示一个声明语句 *declaration-stmt* 包含了一个 decl-specifiers，在它后面跟有 0 个或多个 declarator，这些 declarator 可以通过一个可选的 initializer 来初始化，最后，该语句结束于一个分号。

A.1 声明

声明有时候很难理解，尤其是在同时声明几个具有不同类型的变量，在声明返回值类型是指向函数的指针的函数时就更难理解了。例如，在 § 10.1.1 中，我们见过下面的声明：

```
int* p,q;
```

这个声明把 p 定义为一个整型指针，而把 q 定义为一个整型变量。在 § 10.1.2 中，我们见过下面的声明：

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

这个声明把 get_analysis_ptr 定义为一个不带参数的函数，该函数返回一个指向函数的指针，指针指向的这个函数带有一个 const vector<Student_info>&类型的参数，并返回一个 double

类型的值。

你也可以把上面这些声明分开几个语句来写，例如，上面的两个声明语句可以分别写成：

```
int* p;
int q;
```

和

```
//把analysis_fp定义成一个带一个const vector<Student_info>&类型
//参数的函数，返回一个double类型的值
typedef double(*analysis_fp)(const vector<Student_info>&);
analysis_fp get_analysis_ptr();
```

不过，你可以采用这种分开写的编程风格，却无法阻止其他程序员采用前面那种简洁但是比较费解的风格，所以两种风格都学习一下还是有好处的。

一般来说，声明采用下面的格式：

```
declaration-stmt:decl-specifiers[declarator[initializer]][,declarator[init
ializer]]...;
```

这个语句声明了一个或者几个声明符 declarator。这些声明符从被声明的时候开始生效，在声明所在的作用域结束时自动失效。有些声明同时也是定义。C++中允许对一个名字重复声明几次，但是只能对该名字定义一次。如果在声明语句中为名字分配了内存，或者定义了一个类，或者定义了一个函数体，那么这个声明同时也就是一个定义。

C++从 C 中继承了它的声明语法。理解声明的关键是要认识到每个声明都包括两个部分：一连串的 decl-specifier（声明指定说明）用来指定被声明的类型或者其他属性，后面跟着零个或者一个或者几个 declarator（声明符，每一个声明符后面可选地跟着相应的初始值 initializer）。每个声明符都有一个类型，这个类型由语句中的指定说明部分决定。

理解声明的第一步是要分清楚指定说明部分与声明符部分的边界。其实这是很简单的事：因为指定说明部分都是由类型名称或者类型关键字组成，所以这一部分是在遇到第一个非类型关键字的时候结束的。例如，在

```
const char * const * const * cp;
```

中，第一个既不是类型名称又不是类型关键字的符号是*，所以指定说明部分为 const char，而（惟一的）声明符是*const*const*cp。

再举一个例子，在§10.1.2 有这么一个奇怪的声明：

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

这个声明语句的边界也很容易确定：double 是一个类型，接下来的括号既不是类型关键字也不是类型名。所以，指定说明部分（也就是 decl-specifier 部分）只有 double，而声明符部分则是语句中除去 double 与分号的剩余部分。

A.1.1 指定说明

我们可以把声明语句中的 decl-secifier（声明指定说明）分为三个部分：第一部分是类型指定说明；第二部分是存储类型指定说明；第三部分是其他的指定说明：

```
decl-specifiers:[type-specifier|storage-class-specifier|other-decl-specifier]...
```

但是，这种人为的划分只是为了便于理解，实际上在语法上并没有相应的划分：不同的声明指定说明可以以任何顺序出现。

类型指定说明决定了任何一个声明符的类型。我们将在§A.2 讨论 C++自带的类型。

```
type-specifier: char | wchar_t | bool | short | int | long | signed
      unsigned | float | double | void | type-name | const | volatile
type-name:class-name | enum-name | typedif-name
```

加上 const 这个指定说明表示该类型的对象不会再改变。volatile 关键字告诉编译器，这个变量可以以语言定义以外的方式改变，编译器不能对该变量进行强制性的优化。

注意 const 既可以作为指定说明的一部分，用来指定该类型的对象不能改变，又可以作为声明符的一部分，用来指定一个常量的指针。这两者绝对不会引起混乱，因为作为声明符的一部分的 const 后面总是跟在一个星号字符后面。

存储类型指定说明决定了一个变量的储存位置及其生存周期：

```
storage-class-specifiers: register | static | extern | mutable
```

在声明语句中加上 register 指定说明可以告诉编译器，如果可能的话在运行的时候把该变量的值储存在一个寄存器里以优化程序。

一般来说，一个局部变量在被声明的模块结束的时候自动被删除，该变量占用的内存也被自动释放，还给系统；而一个被指定为 static 的变量的生存周期则为从一个模块的入口到该模块的出口。

extern 指定说明的意思是，当前的声明不是一个定义，该声明相应的定义在程序的其他地方可以找到。

mutable 这种存储器类只用于类里面的数据成员，一个 mutable 的数据成员即使被定义成常量成员也可以被修改。

其他的指定说明定义的是与类型无关的性质：

```
other-decl-specifier: friend | inline | virtual | typedef
```

其中 friend 指定说明（详细使用说明见§12.3.2 和§13.4.2）用于在遵循语言的保护机制的情况之下访问类的被保护成员。

inline 用于函数声明中，编译器在编译的时候将尽可能地把该函数定义的代码内插到程序中用到该函数的地方。在调用这类函数的时候要求函数的定义必须在同一作用域内，所以最好

养成一个良好的编程习惯，就是把内嵌函数的定义与该函数的声明语句放在同一个头文件中。

virtual 关键字（见§13.2.1）只能用在成员函数上，用来告诉编译器该函数在调用的时候是动态绑定的。

typedef 指定说明（见§3.2.2）用来为一个类型定义一个替代名。

A.1.2 声明符

声明语句为每一个声明符声明了一个实体，为该实体命名，并且由指定说明指定该实体的变量类型、存储类，以及其他的属性。指定说明与声明符一起，把一个名称定义为一个对象、数组、指针、引用或者函数。例如：

```
int *x, f();
```

上面这行语句把 x 声明为一个指向 int 类型的指针，把 f 声明为一个返回 int 类型结果的函数。声明符*x 和 f()决定了 x 与 f 的类型之间的差别。

```
declarator: [*[const] | &]...direct-declarator
direct-declarator: declartor-id | (declarator) |
    direct-declarator (parmeter-declaration-list) |
    direct-declarator[constant-expression]
```

declarator-id 是一个标识符，可以加上下面的限定：

```
declartor id:[nested-name-specifier]identifier
nested-name-specifier:{class-or-namespace-name::}...
```

如果一个声明符是一个只由 declarator-id 组成的 direct-declarator，那意味着这个标识符只具有 decl-specifier 指定的性质，不能再作其他的改变。例如，在下面的声明语句中：

```
int n;
```

声明符是 n，这是一个只由一个 declarator-id（声明符标识）组成的 direct-declarator（直接声明符），因此，n 是一个 int 类型的变量。

如果某个声明符具有其他的形式，那么你可以通过下面的途径来决定该标识符的类型：首先，忽略其他（如 friend 或者 static 等）非类型的性质，通过 decl-specifier 指定 T 的类型，然后声明 D 为这个声明符。然后不断重复下面的几个步骤直到 D 变为一个 declarator-id，这时候 T 就是你想要的类型了：

1. 如果 D 具有（D1）的形式，那么用 D 代替 D1。

2. 如果 D 具有*D1 或者*const D1 的形式，那么用“指向 T 的指针”或者“指向 T 的常量指针”（而不是“指向常量 T 的指针”）来代替 T，至于是用常量指针还是非常量指针，这要取决于 D 的形式中有没有 const。然后用 D 来代替 D1。

3. 如果 D 具有 D1（parameter-declaration-list）的形式，那么用“返回 T 类型的函

代替 T，该函数带有 parameter-declaration-list（参数声明列表）所定义的参数，然后用 D 代替 D1。

4．如果 D 具有 D1[constant-expression]的形式，那么用“T 类型的数组”来代替 T，该数组具有 constant-expression（常量表达式）个元素，然后用 D 代替 D1。

5．最后，如果声明符具有&D1 的形式，那么用“指向 T 的引用”来代替 T，然后用 D 代替 D1。

举个例子来说，在下面的声明语句中：

```
int *f();
```

T 和 D 开始分别是 int 类型和*f（）类型，所以 D 具有*D1 的形式，其中 D1 是 f（）。

你可能会认为 D 要么具有 D1（）的形式，要么具有*D1 的形式。不过，如果 D 具有 D1（）的形式，那么 D1 就是*f，而 D1 就必须是一个 direct-declarator（因为本节开始的语法说过，在括号之前必须是一个 direct-declarator）。通过查阅前面对于 direct-declarator 的定义，我们知道在 direct-declarator 的定义中不能包含星号字符*。因此，D 只能是*f（），它具有*D1 的形式，其中 D1 是 f（）。

现在我们已经知道 D1 是 f（），所以必须用“指向 T 类型的指针”（在这里是一个“指向 int 类型的指针”）来代替 T，然后用 f（）代替 D。

至此，我们还没有把 D 简化为一个 declarator-id，所以必须重复上面的过程。这一次，D1 只能是 f，所以我们用“返回 T 类型的函数”来代替 T，在这里这是个“不带参数的、返回一个指向 int 类型指针的函数”，然后用 f 来代替 D。

经过以上的努力，我们终于把 D 规约为一个 declarator-id。现在我们知道，下面这个声明语句

```
int *f();
```

把 f 声明为一个“不带任何参数的、返回一个指向 int 类型的指针的函数”。

再多举一个例子，先来看看下面的声明语句：

```
int* p,q;
```

这个声明中有两个声明符，*p 和 q。对这两个声明符，T 都是 int 类型的。对于第一个声明符，D 是*p，所以我们把 T 转换成“指向 int 类型的指针”，把 D 转换成 p。所以上面这个声明语句声明 p 为一个“指向 int 类型的指针”。

现在来单独分析一下第二个声明符，T 仍然是 int 类型的，而 D 是 q。所以显然在这个声明中把 q 声明为一个 int 类型的变量。

最后，让我们来分析一下§10.1.2 中提到的一个奇怪的例子：

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

对该声明的分析分五步进行：

1. T：一个 double 类型　　D：（*get_analysis_ptr()）(const vector<Student_info>&)
2. T：一个带 const vector<Student_info>&参数的、返回一个 double 类型的函数　　D：（*get_analysis_ptr()）
3. T：一个步骤 2 中所说的函数　　D：*get_analysis_ptr()
4. T：一个指向步骤 2 中所说的函数的指针　　D：get_analysis_ptr()
5. T：一个返回指向步骤 2 中所说的函数的指针　　D：get_analysis_ptr

也就是说，最后我们得出结论，get_analysis_ptr 是一个函数，它的返回值是一个指向函数的指针，这个函数带有一个类型为 const vector<Student_info>&的参数并且返回值的类型为 double。至于对 const vector<Student_info>&的分析则留下来作为一个练习请读者自己去思考一下。

幸运的是，一般的声明语句都不会这么复杂，大多数的语句都是采取下面这样的简单形式：

```
declarator:declarator-id (parameter-declaration-list)
```

迄今为止最难的情况不过是具有一个指向函数的指针的返回值的函数。

A.2 类型

关于类型的使用遍及所有的 C++程序。每一个对象、表达式或者函数都有一个类型，一个实体的类型决定了它的行为。除了一个简单的例外以外（在一个继承树中，通过一个指针或者引用访问的对象的类型），任何一个实体的类型都是在编译的时候就已经被确定下来。

在 C++中，类型既决定了构造内存与访问内存的方式，又决定了定义对该类型对象操作的方式。也就是说，类型同时决定了一个数据的数据特性和操作特性。

虽然本书把注意力集中放在使用和生成较高层的数据结构上，但是理解用来生成高层数据结构的底层的原始类型也是十分重要的。这些原始的类型代表着接近硬件的一般抽象概念，例如数字（整型数与浮点数）、字符（包括国际字符集在内的广义的字符）、真值以及机器地址（指针、引用和数组）等。还有一种叫常量的类型，可以是整型值，也可以是浮点值、布尔值、字符值或者字符串中的任何一种类型值。在本节我们来详细学习一下与 C++自带类型相关的知识。

A.2.1 整数类型

C++从 C 中继承了多种令人眼花缭乱的整数类型，包括整型、布尔型和字符类型等。因为 C++倾向于用来编写在大多数的硬件上正常运行的程序，所以 C++对很多基本类型都没有进行精确的定义，而是把基本类型的实现细节留给不同的编译器来完成。

A.2.1.1 整型

整型类型分为三种截然不同的带符号整型类型与三种不同的无符号整型类型：

```
short int          int          long int
```

```
unsigned short int        unsigned int        unsigned long int
```

short int 可以简写为 short，long int 可以简写为 long。如果在类型中有不止一个的关键字，那么关键字的顺序可以是任意的。

上面的这几个类型可以用来表示任何实用意义上的整数。上面列出的几个带符号的整型类型中，每一个类型都能储存一个与它前面的那个类型一样或者更大范围的整数。例如，short int 与 int 类型可以存储一个从-32767~+32767（$\pm(2^{15}-1)$）范围的整数，而 long int 类型至少可以存储一个从-2147483647~+2147483647（$\pm(2^{31}-1)$）的整数。

每一个带符号的整数类型都有一个对应的无符号整数类型。每一个无符号整数类型都可以储存 2^n 这么大的整数，n 取决于具体类型与系统环境。与带符号类型一样，每一个无符号整数类型对应的 n 都至少与前面那个无符号整数类型对应的 n 一样大。另外，每一个无符号类型都可以用来储存对应的带符号类型范围内的非负整数值，而且如果一个带符号类型与一个相应的无符号类型保存的值相同，那么它们的内部表示方式是一样的。由于有以上的这些要求，所以三种无符号类型都比相应的带符号类型要多出一位表示数值(这一位在带符号类型中是用来表示整数的符号的)，也就是说，无符号整型类型可以存储的最大的整数值 2^n 对应的 n 值分别至少是 8、16 和 32。

编译器在实现时被允许将程序中的有符号整数表示为一元补码或二元补码的形式。

标准库中定义了一个叫做 size_t 的类型，它是其中一个无符号整型的替代名。size_t 类型保证在该类型的变量中可以容纳最大的对象（包括数组）的尺寸。该类型定义于系统头文件<cstddef>中。

整型直接量：一个整型直接量由一组阿拉伯数字组成，前面可以可选地加上一个基数标志，后面可以可选地加上一个大小标志。严格来说，整型直接量是没有符号的，所以-3 是一个表达式，而不是一个数值。

如果一个数值以 0x 或者 0X 开头，那么这是一个十六进制的整数，十六进制整数的主体可以包含 AaBbCcDdEeFf 这些字符以及 10 个“阿拉伯数字”。如果这个数值以 0 开头而且 0 后面没有紧跟着 x 或者 X，那么这就是一个八进制的整数，它的主体只能由 01234567 八个阿拉伯数字组成。

大小标志可以是 u、l、ul 或者 lu，可以写成大写或者小写的形式。在数值后面加上 lu 或者 ul 标志表示这是个无符号的长整型（unsigned long）直接量。如果在后面加上 u，表示该数值是个无符号整型直接量或者无符号长整型直接量，这取决于具体数值的大小。如果在后面加上 l，那么该数值就是个长整型直接量或者无符号长整型直接量，这也取决于数值是否带符号和数值的大小。

如果一个整型直接量只有一个基数标志而没有大小标志，那么该整型直接量的类型根据其大小与是否带符号，看它有没有可能是一个 int 类型的数值，如果有可能，那该数值就是一个 int 类型的整型直接量，如果不可能，就视该值的大小和其是否带符号来看有没有可能

unsigned 类型，按照 int、unsigned、long 和 unsigned long 的顺序依此类推可以决定该数值的类型。如果既没有基数标志又没有大小标志，那么编译器优先把该数值看作一个 int 类型的数值，如果不可能是 int 类型，就把它当作一个 long 类型的数值。

这些规律意味着一个整型直接量的类型对于不同的实现环境有可能是不同的。不过幸运的是，在一个好的程序中整型直接量一般都很小，所以这些细节问题一般不会太伤脑筋。不过在这里我们还是对这个问题进行了详细的讲解，没准什么时候你会面对这个问题呢。

A.2.1.2 布尔类型

一个条件表达式一定是具有布尔类型的值。布尔值只可能是真或者假。也可以用数字或者指针来作为一个布尔值。如果用数字或者指针来表示布尔值，那么 0 表示布尔值假，其他值表示布尔值真。

如果把布尔值作为数字使用，那么编译器会把假和真分别作为 0 和 1 使用。

布尔型直接量：布尔型直接量只可能是 true 或者 false，它们都是布尔类型，意思也是显而易见的，在此就不多做解释了。

A.2.1.3 字符类型

在 C++中，字符类型的值只不过是一些很小的整数。另外，它们也可以在算术表达式中被当作整数使用。

像整数类型的值一样，字符类型也可以划分为有符号字符或无符号字符。在任何一个系统环境中，都要求带符号的字符类型变量可以表示计算机基本字符集中的所有字符，也就是说对这种类型的字符要求其表示范围至少为±127（±（2^7-1））。

另外，C++中还有一种无格式的字符类型，虽然这也是一个独立的类型，不过它要求与其他两种字符类型具有相同的表示方式。由系统环境来决定这种无格式的字符类型到底是哪一种实际的类型。一般来说，系统环境选择用哪一种字符类型表示这种无格式的字符类型，就意味着该系统觉得使用这种字符类型最自然。

还有一种叫做“宽字符”的类型，关键字是 wchar_t，它包含至少 16 位，主要用来表示像日语之类的语言中的字符，日语中有远多于拉丁字母的基本字符。wchar_t 类型与其他的整数类型具有相同的使用语法。此外，其他的特殊类型的具体实现由系统环境决定，一般都根据系统环境选择生成最有效的表述。

字符直接量：字符直接量（例如‘a’字符）通常是一个用单引号括起来的单个字符。字符常量具有 char 类型，从§A.2.1.3 的内容中我们知道，它是一种整数类型的常量。每个系统环境都定义了字符常量与整数之间的对应关系。大多数的程序都不能依赖于这个对应关系，因为程序员们有可能用一个字符常量如‘a’来表示“与字符 a 对应的整数”。因为字符与整数的对应关系随不同的系统环境而异，所以程序员们也不能使程序依赖于字符的算术特性。例如，‘a’+1 不一定等于‘b’，‘a’+1 的结果是随不同的系统环境而改变的。但是可以保证的是，阿拉伯数字字符所代表的整数值之间是连续的。所以‘3’+1 一定等于‘4’（不过可不一定等于 4）。

字符串常量：一个字符串常量，例如“Hello, world!”，通常是由一对双引号括起来的一串（零个或一个或几个）字符。字符串常量的类型是 const char*。编译器会自动为每个字符串常量的结尾加上一个空字符。

两个由空格分开的字符串常量会自动地连接在一起，生成一个更长的字符中常量。这一特性使得一个超过一行的字符串写起来特别方便。

A.2.1.4 字符表示法

在前面说过，一个字符常量一般是由一对单引号括起来的一个字符表示的，而一个字符串常量一般是由一对双引号括起来的一串字符表示。这里用到“一般”一词，因为除了这种常见的表示方法以外，还有一些特殊情况。下面列出的这些特殊的表示法同时适用于字符常量与字符串常量：

- 在字符常量（或者字符串常量）表达式中为了表示单引号（或者双引号），你必须在单引号（或双引号）前面加上一个反斜杠符号（例如'\'' 或者 "the \"quotes\" "）以表示这个引号不是作为字符（或者字符串）的结束标志。为了方便起见，你也可以在任何一种引号前面加上一个反斜杠（例如'\"'）而不用管这是在一个字符常量中还是在一个字符串常量中。
- 为了表示一个反斜杠符号，你可以在反斜杠符号前面再加一个反斜杠，如 '\\'，这样编译器就知道在反斜杠后面并没有跟着某个需要加一个反斜杠表示的特殊字符。
- 对于国际标准字符集的表示还有一些其他规则，这些规则超出了本书的讨论范围，不过因这些规则，在代码中存在连续两个问号的时候会影响程序的原意。为了避免代码中出现连续两个问号的情况，C++允许用\?来代替一个问号，这样你就可以写出"What?\?"来表示问号直接量，从而在代码中避免出现连续的两个问号。
- 另外还有一组用来控制输出格式的控制字符：另起一行（\n），水平制表（\t），垂直制表（\v），后退一个字符（\b），回车换行（\r），换页（\f）以及响铃（\a）。根据不同的系统环境，这些控制字符在对输出设备进行输出的时候有不同的控制效果。
- 如果真的需要用整数来表示一个字符，你可以用\x 跟着一个十六进制的数值来表示该特殊字符，也可以用\跟着一个最多三位数的八进制数来表示。例如，'\x20'和'\40'都表示十进制整数 32（用十六进制表示是 20，用八进制表示是 40）对应的字符。在基于 ASCII 字符集的系统环境中，它表示的就是空格字符' '。这种表示方法最常见的用途（常常也是某个程序中该表示方法的惟一用途）是用'\0'来表示空字符，它的整数值为 0。

A.2.2 浮点类型

C++中有三种浮点类型，分别是 float、double 和 long double 类型，这三种浮点类型的精度是递增（或者相等）的。在具体的系统环境中，float 类型与 double 类型可能具有相同的精度，而 double 类型与 long double 类型也可能具有相同的精度。不过所有的系统都要求
类型变量能存储至少六位数的值（十进制数），要求 double 类型变量和 long double 类型

少能存储十位数的值。大部分的系统都为 float 类型变量提供存储六位数的空间，为 double 类型变量提供存储十五位数的空间。

浮点常量是一个非空数位序列，可以在最后带有指数或中间带有小数点。和整型常量一样，不能以符号开头；-3.1 是一个表达式，而非常量。小数点可以在数位序列的开头、中间或结尾，如有指数还可以被省略。指数的表达形式是 e 或 E，后面是可选的符号，以及一位及以上数字。指数以十进制进行运算。

例如，312E5 和 31200000 表示同样的数字，但是 31200000 是一个整型常量，而不是一个浮点型的常量。再举一个例子，0.0012 可以表示成 1.2E-3，也可以表示成 0.000012e+2 的形式。

浮点型常量默认情况下是 double 类型的。如果你希望指定一个常量是 float 类型，就必须在常量后面加上 f 后缀或者 F 后缀；如果你想指定它为 long double 类型，则要在常量后面加上 l 后缀或者 L 后缀。

A.2.3 常量表达式

一个常量表达式的值是一个整型直接量，它的值在编译的时候就已经确定。在一个常量表达式中只能包含直接量、枚举类型量、常量变量或者是整型的静态数据成员，不过这个静态数据成员要求是用另一个常量表达式或者一个 sizeof 表达式赋值。在常量表达式中不能包含函数、类对象、指针或者引用，而且不允许使用赋值、递增、递减、函数调用或者逗点运算符。

一个常量表达式可以出现在任何常量能够出现的地方。例如，在数组声明中表示数组的维数（见§10.1.3），又例如在 switch 语句中作为标签（见§A.4），还有作为计数器的初始值（见§A.2.5）等。

A.2.4 类型转换

在必要时，在运算符需要所有操作数具有相同类型的情况下，会产生类型的转换。在可以选择的情况下，转换往往是倾向于保留原有信息，而不是丢失它。另外，向无符号类型进行的转换要优于向有符号类型进行的转换。在对短整型和字符型变量进行算术运算的时候会发生向整型（或者长整型）类型的转换；而在浮点数的算术运算中往往发生向双精度浮点（或者长双精度）类型的转换。

最简单的转换是升级转换。升级转换允许一个较小类型（例如，char 类型）的值升级成一个较大的相关类型（例如，int 类型）的值；在该转换中保留了初始值的符号。如果可以的话，整型值的升级把一个 char、signed char、unsigned char、short 或者 unsigned short 类型的值转换成一个 int 类型的值，如果不行就转换成一个 unsigned int 类型的值。宽字符类型与枚举类型（见§A.2.5）的值会升级为能够用来表示所有这些基类型的值的最小的 int 类型。在升级转换的时候，依 int、unsigned int、long 和 unsigned long 这个顺序先后进行升级转换。bool 类型升级转换为 int 类型，而 float 类型升级转换为 double 类型。

从一个整数类型向一个浮点类型的转换会尽可能多地保留原整型数的精度，这取决

的硬件环境。

从一个较大的带符号类型（例如，long 类型）值向一个较小的类型（例如，short 类型）值的由具体系统环境定义。从一个较大的无符号类型值向一个较小的无符号类型值的转换的结果是将该数除于 2^n 后得到的模，在这里 n 是较小类型的存储位数。从一个浮点类型值向一个整数类型值的转换会忽略小数部分并将这部分截除。如果被截除后的值依然不能被该类型表达的话，那么我们能够得到的行为是一个没有定义的行为。

指针类型、整数类型和浮点类型的值可以转换成布尔类型的值。如果它们的值是 0，那么返回的布尔值为假，否则为真。布尔类型的值也可以被转换成其他类型的值。真被转换成 1 而假被转换成 0。

一个结果为 0 的常量表达式（见 § A.2.3）可以转换成一个指针。

任何一个指针都可以转换成 void*类型。一个指向非常量类型的指针可以转换成一个指向常量的指针——类似于非常量类型的引用。一个指向某个派生类型对象的指针或者引用可以转换成一个指向基类型对象的指针或者引用，不过要求派生类必须是从基类中以公有的形式派生出来的。

算术类型转换：因为在算术运算中，操作数既可以是整型值，也可以是浮点值，既可以是带符号类型的，也可以是无符号类型的，所以在算术运算中要决定最后结果的类型比较困难。最后结果的类型通过下面的**常用算术类型转换**规律确定：

- 如果有一个操作数是浮点类型，那么结果也是浮点类型，结果的精度与操作数中具有最大精度的操作数具有相同精度。
- 否则，如果有一个操作数是 unsigned long 类型，那么结果也是 unsigned long 类型。
- 否则，如果有一个操作数是 long int 类型而另一个是 unsigned 类型（包括除了 unsigned long 类型以外的所有无符号类型，因为根据上一条规律，unsigned long 会使结果强制转换成 unsigned long 类型），那么结果类型决定于具体的系统环境：如果 long int 类型的范围包含了 unsigned int 类型的范围，那么结果是 long int 类型的；否则结果的类型就是 unsigned int。
- 否则，如果有一个操作数是 long int 类型，那么结果也是 long int 类型。
- 否则，两个操作数肯定都是带符号的整数类型 int 类型或者是更短的类型，结果都是 int 类型。

由上面的转换规律可以得到一条结论：任何一个算术运算都不会得到 short 类型或者 char 类型的结果。所有的算术运算都把操作数转换成 int 类型或者更长的类型进行计算。

A.2.5 枚举类型

枚举类型定义了一组特定的整型值。该类型的对象只能取类型定义中指定的值：

```
enum enum-name{
        enumerator[ , enumerator ] ...
```

```
};
```

enum-name 是类型名，可以在任何一个需要 type-name 类型的值的地方使用 enum-name 类型的值。

enum-name 类型的变量可以是在 enumerator 列表中列出的任何一个值：

```
enumerator:identifier[=consttant-expression]
```

除非特别指明，否则枚举类型的值对应着从零开始的连续整数。可以为枚举类型显式地指定对应的整数值。用来初始化的值必须是具有整数类型（见§A.2.1）的，可以在编译过程（见§A.2.3）中被编译器识别。如果在一个要求整数的地方提供一个枚举类型的值，则该值会被自动转换成相应的整数值。

A.2.6 重载

可以有不止一个函数体共用一个函数名(但是这几个函数体的参数个数或者参数类型必须至少有一个不同)，这种用法叫做函数重载。

如果程序中某个地方调用了一个重载函数，那么编译器会在编译的时候会检查参数个数和参数类型，由此决定具体调用哪一个函数。与实际参数最佳匹配的函数将被调用。所谓与实际参数最佳匹配，指的是该函数要求的参数与其他函数要求的参数比起来，至少要有一个参数更加匹配，而其他的任何一个参数都不比其他函数的参数差。

与某个参数最佳匹配被定义如下：

- 精确的匹配（参数类型完全相同）是最佳的匹配。
- 可以使用升级转换（见§A.2.4）得到所要求的参数类型的匹配，比使用 C++内建类型转换得到所要求的参数类型的匹配要好，而第一种匹配又比使用在类中定义的转换（见§12.2 和§12.5）得到所要求的参数类型的匹配要好。

如果有不止一个函数与实际调用中传递的参数匹配，那就是错误的。

A.3 表达式

C++保留了 C 中丰富的表达式语法特性，与 C 比起来，C++中增加了运算符重载（见§A.3.1）的机制。运算符重载机制可以允许程序员们重定义一个运算符的参数、返回类型以及运算符的含义，但是不允许改变该运算符的优先权、价（也就是操作数的个数）或者结合性，另外程序员还不能重载对 C++内建类型操作数进行运算的内建运算符。在本节我们将讲述一下作用于 C++内建类型的操作数上的运算符。

每一个运算符都生成一个值，结果的类型取决于操作数的类型。一般来说，如果操作数具有相同的类型，那么结果的类型与操作数类型相同。否则，在运算之前先进行标准的类型转换以得到一种共同的类型（见§A.2.4）。

左值（lvalue）指的是一个非临时的对象，所以它具有一个地址。某些操作只有作用在左值上才有效，另外某些操作只生成一个左值结果。每个表达式都生成一个值，有些表达式也生成一个左值。

只有三个运算符要求它们的操作数按顺序满足一定条件后才会进行计算：

&&　只有当左操作数为真时才会对右操作数进行计算。

||　只有当左操作数为假时才会对右操作数进行计算。

?:　跟在条件后面的两个表达式只对其中一个进行计算。当条件表达式为真时对？后面的表达式进行计算；否则对：后面的表达式进行计算。结果是被计算的表达式的值；如果两个表达式都是同一类型的左值，那么结果也是一个左值。

对于其他的运算符，对操作数进行计算的顺序没有任何要求（除了优先权规则以外），也就是说，编译器可以以任意的顺序对操作数进行计算。可以用圆括号来改变表达式默认的计算优先权，但是必须显式地使用临时变量来完全控制操作数计算的顺序。

每个运算符都规定了优先权与结合性。在下面列出的表格中，我们根据其优先权先后顺序对所有的运算符进行了归纳。当几个运算符被括号括成一组使用的时候，它们共用相同的优先权与结合性。每个组都引入新的优先权层次。下面这个表格是第 2 章出现的表格的一个扩展，包括了所有的运算符。

终止符	标识符或者直接量常量；标识符是左值，直接量不是
C::m	C 类中的 m 成员。
N::m	名字空间 N 中的 m 成员。
::m	全局作用域内的名字 m。
x[y]	x 对象中的第 y 个元素。生成一个左值。
x->y	x 指向的对象中的 y 成员。生成一个左值。
x.y	x 对象中的 y 成员。如果 x 是左值的话则该表达式也生成一个左值。
f（*args*）	调用函数 f，并传递 args 作为参数（列表）。
x++	对左值 x 加一。生成 x 的初始值。
x--	对左值 x 减一。生成 x 的初始值。
*x	对指针 x 间接引用。生成 x 指向的对象，这是一个左值。
&x	对象 x 的地址。生成一个指针。
-x	相反数。这是个一元算符，x 必须是数字类型的表达式。
!x	逻辑非。如果 x 为 0，则!x 的值为真，否则为假。
~x	x 的补码。x 必须是一个整数类型的值。
++x	递增左值 x。生成递增后的值作为左值。
--x	递减左值 x。生成递减后的值作为左值。
sizeof(e)	表达式 e 占用的字节数，是一个 size_t 类型的值。

sizeof(T)	T 类型的对象占用的字节数，是一个 size_t 类型的值。
T(*args*)	用 *args* 参数来构造一个 T 类型对象。
new T	为一个 T 类型对象分配内存并进行默认初始化。
new T (*args*)	为一个 T 类型对象分配内存并用 *args* 参数对它初始化。
new T[n]	为一个具有 n 个 T 类型元素的矩阵分配内存并对它进行默认初始化。
delete p	释放 p 指向的对象占用的内存空间。
delete [ ] p	释放 p 所指向的数组所占用的内存空间。
x*y	x 与 y 的乘积
x/y	x 与 y 的商。如果两个操作数都是整数，则由系统环境决定是对商四舍五入成 0 还是负无穷大。
x%y	x-((x-y)*y)。
x+y	如果两个操作数都是数值，那么表示 x 与 y 的和。如果 x 是一个指针，而 y 是一个整数，则生成一个指针，该指针指向 x 地址后面第 y 个元素。
x-y	如果两个操作数都是数值，那么表示 x 与 y 的差。如果 x 和 y 都是指针，则生成两个指针所指对象之间的距离，单位是一个元素。
x>>y	如果 x 和 y 都是整数类型，表示把 x 右移 y 位；y 必须是一个非负整数。如果 x 是一个流，那么从 x 读取数据放进 y 中并返回左值 x。
x<<y	如果 x 和 y 都是整数类型，表示把 x 左移 y 位；y 必须是一个非负整数。如果 x 是一个流，那么把 y 的值输出到 x 中并返回左值 x。
x *relop* y	生成一个布尔类型的值，表示该关系式是否为真。 关系运算符（<，>，<=，和>=）具有它们常用的意义。 如果 x 和 y 都是指针，那么它们必须指向同一个对象或数组。
x==y	生成一个布尔类型的值，表示 x 与 y 是否相等。
x!=y	生成一个布尔类型的值，表示 x 与 y 是否不相等。
x&y	按位“与”。x 和 y 必须是整数。
x^y	按位“异”。x 和 y 必须是整数。
x|y	按位“或”。x 和 y 必须是整数。
x&&y	生成一个布尔类型的值，表示是否 x 和 y 都为真。 只有在 x 为真的时候才会对 y 进行计算。
x||y	生成一个布尔类型的值，表示是否在 x 和 y 中至少有一个为真。 只有在 x 为假的时候才会对 y 进行计算。
x=y	把 y 的值赋给 x。生成 x 这个左值作为结果。
x *op*= x	复合赋值运算符。等价于 x = x *op* y，在这里，*op* 是一个数学运算符，按位逻辑运算符或者是移位运算符。

x?y1:y2	如果 x 为真，则生成 y1，否则生成 y2。 y1 和 y2 中只有一个会被计算。 y1 和 y2 必须是同一类型的表达式或者变量。 如果 y1 和 y2 都是左值，那么结果也是一个左值。 运算符是右结合的。
throw x	抛出 x 的值以显示一个错误。 x 的类型决定了由哪个句柄来捕捉该错误。
x , y	先对 x 进行计算，无论结果如何都接着对 y 进行计算，然后返回 y 的值作为结果。

A.3.1 运算符

C++内建的大部分运算符都可以被重载。除了 throw 运算符、范围运算符、点运算符，和有条件运算符（？：运算符）不能被重载，其他的运算符都可以被重载。在 § 11.2.4 中我们已经讲过如何定义一个重载运算符。

++/--后缀运算符在重载定义的时候与++/--前缀运算符不同，++/--后缀运算符在定义的时候带有一个没有用的伪参数。也就是说，在重载++/--后缀运算符的时候必须这样写：

```
class Number{
public:
    Number operator++(int) { /* 函数体 */ }
    Number operator--(int) { /* 函数体 */ }
};
```

最常用的重载运算符有赋值运算符和索引运算符等，移位运算符通常用在 ostream 类型对象和 istream 类型对象上用来进行输入-输出操作，在 § B.2.5 还会介绍用来实现迭代器的运算符。

A.4 语句

和大多数的编程语言一样，C++严格区分声明、表达式和语句。在一些特定的上下文里，声明和语句可以在其他的声明和语句里嵌套使用，但是它们都不能插在表达式里使用。每个语句最终都出现在函数定义里，用来描述函数调用时的部分行为。

除了一些例外，一般来说构成函数的语句都按照它们出现的次序顺序执行。这些例外包括使用循环、函数调用、goto 语句、break 语句和 continue 语句的执行，以及用来运行异常处理的 try 语句和 throw 语句等。

语句可以以很自由的形式书写。在一个语句的中间另起一行书写不会影响该语句的原意。大部分的语句都以分号结尾——块语句例外（它是以{开始，以}结尾的）。

; 空语句：在执行的时候没有任何作用。

e; 表达式语句：通过计算表达式对程序产生副作用。

{ } 语句块：按顺序执行块里的语句。

在某个块里进行的声明直到出现右花括号之前一直保持有效。

if（条件表达式）语句 1

先对条件表达式进行计算，如果为真就执行语句 1。

if（条件表达式）语句 1 else 语句 2

先对条件表达式进行计算，如果为真就执行语句 1；否则执行语句 2。每一个 else 关键字都与在它前面最近的 if 关键字匹配。

while（条件表达式）语句

先计算条件表达式，如果为真就执行语句；然后重复前面的过程，只要条件表达式为真，就再次执行语句并重复循环；直到条件表达式为假时跳出循环。

do 语句 while（条件表达式）

先执行语句再测试条件表达式。不断执行语句和测试条件表达式，直到条件表达式为假。

for（初始化语句条件表达式；表达式）语句

在循环入口处只执行一次初始化语句，然后测试条件表达式。如果条件为真，则先执行语句再计算表达式。重复测试条件表达式、执行语句和计算表达式，直到条件表达式的值为假。

switch（表达式）语句

在实际使用当中，switch 语句中的语句几乎总是一个具有标签指示的语句块，标签具有下面的形式

```
case value:
```

在这里 value 必须是一个整数类型的常量表达式（见 § A.2.3）。另外，下面这个标签也可能出现：

```
default:
```

但是在 switch 语句中它只能出现一次。

执行一个 switch 语句的时候，先计算表达式，然后跳转到与表达式的值匹配的标签处。如果没有匹配的标签，程序跳转到整个 default: 标签，如果没有 default: 标签，就跳到整个 switch 语句的下一个语句继续执行。

因为 case 标签仅仅是标签，所以如果程序员不显式地进行控制，程序会自然地顺序执行不同标签指向的语句，一般都用 break 语句指示程序在执行完一个标签指向的语句后跳出 switch 语句。

break; 跳出该语句所在的 while 循环、for 循环、do 循环或者 switch 语句。

continue; 跳入一个 for 循环语句、while 循环语句或者一个 do 循环语句的下一个循环（包括条件测试表达式）。

goto 标签; 该语句的作用与它在其他语言中的作用一样。goto 语句的目标是一个标签，是一个后跟冒号的标识。标签可以拥有与其他实体相同的名字而不引起混乱。标签的作用域是出现该标签的整个函数，这意味着可以使用该语句从一个块的外部跳到内部。但是，这种跳转不能绕开对一个变量的初始化。

```
try{语句}catch（参数-1） {语句-1}
   [catch（参数-2） {语句-2}] ...
```

先执行语句中的代码，这些语句可能会产生一个异常事件，该异常事件由一个或者多个 catch 子句来处理。如果在主语句中产生的异常值与某个 catch 子句的参数-n 类型匹配，那么程序将运行该 catch 子句中的语句-n 来处理这个异常。在这里，所谓匹配指的是异常值与参数具有相同的类型或者是参数类型的一个派生类型。

如果 catch 子句具有 catch（ ... ）的形式，那么该子句中的语句用来处理所有其他子句都不匹配的异常。

如果任何一个 catch 语句都与异常值不匹配，那么该异常将被传递给包含该 try 块的最近的另一个 try 块。如果没有其他 try 块包含该 try 块，程序将被终止。

throw 表达式; 中断程序或者把程序的控制权交给正在执行的 try 块中相匹配的 catch 子句。由表达式的类型来决定由哪一个 catch 子句来处理该异常。如果在当前 try 块里没有匹配的 catch 子句，程序将被终止。

异常通常是类的对象，一般在一个函数中产生，然后被另一个函数捕获并进行处理。

附录 B

标准库一览

标准库是 C++标准的一个主要部分。在本书前面各章中，我们使用标准库写出了一些合乎习惯的简洁的 C++程序。本章将回顾前面用到的标准库提供的功能，另外还讲了一些前面没用过的常用工具。下面每一节都讲了一个或者一组标准库中相关的类，对类提供的相关功能进行了详细的讲解。

一般来说，标准库在名字空间 std 里定义了所有的名字。所以在用到标准库的函数时，必须在函数名前加上 std::前缀或者用 using 声明来使该函数可以被使用。不过在本附录里，我们省略了这个前缀。例如，一个输出函数会被写成 cout 的形式而不是 std::cout 的形式。

本章的例子中用到的符号都满足下面的约定：

n	一个可以表示为任意整型值的变量或者表达式
t	一个 T 类型的值
s	一个字符串值
cp	一个指向某个以空字符结尾的字符数组的首元素的指针
c	一个字符值
p	一个判断表达式，通常是一个返回布尔类型的或者可以转换成布尔类型的值
os	一个输出流
is	一个输入流
strm	一个输入流或者一个输出流
b	一个指示起点的迭代器
e	一个指示终点（后面位置）的迭代器
d	一个指示目的的迭代器
it	一个指示一个元素的迭代器

B.1 输入-输出

istream 类、ostream 类、ifstream 类和 ofstream 类的对象都是用来表示连续的流，在任何时候都只能有一个输入-输出类型对象与一个流相关联。这些类型的对象不能被复制或者赋值

只能通过一个指针或者引用来把流传递给一个函数，也只能通过一个指针或者引用来从一个函数中返回流。

基础

#include<iostream> 声明了输入-输出类和相关的操作。

cout

cerr

clog 与标准输出流（cout）和错误流（cerr，clog）相关联的 ostream 类型对象。默认情况下，输出到 cout 和 clog 的流会被缓冲输出，而输出到 cerr 的流不会经过缓冲。

cin 一个与标准输入流关联的 istream 类型对象。

读与写

is>>t 忽略空白字符，从 is 中读取一个值到 t。输入流 is 必须提供一个可以转换成 t 变量的类型的形式，它是一个非常量的左值。不正确的输入形式导致请求失败，在调用 is.clear()函数之前 is 一直处于失败状态。在库中为 C++自带类型和字符串类型定义了>>输入运算符；在编写类的时候要求按照库中定义的格式使用该算符。

os<<t 以一种与 t 变量的类型匹配的格式，把 t 的值送至输出流 os 中。在库中为 C++自带的类型和字符串类型定义了<<运算符；在编写类的时候要求按照库中定义的格式使用该运算符。

is.get(c) 从 is 输入流中读取下一个字符（包括空白字符）放进 c 变量。

is.unget() 取消最近一次从一个输入流读出一个字符的操作。在我们想不断从输入流中读出数据，直到读到一个特殊的字符，然后希望把刚才读出的那个字符返回输入流中（因为在进行的操作中可能要求保留该字符），在这种时候需要调用该函数。unget 函数只能取消最近一次的读取操作。

迭代器（iterator）

#include<iterator> 声明了输入-输出流迭代器。

istream_iterator<T> in(is);

把 in 定义为一个输入迭代器，可以用它来从输入流 is 中读取 T 类型的值。

ostream_iterator<T> out(os,const char* sep=" ");

把 out 定义成一个输出迭代器，可以用它来从输出流 os 中输出 T 类型的值。在输出每个元素之后输出一个 sep 分隔符。默认情况下，分隔符 sep 是一个空字符串，不过它可以被指定为任意一个字符串常量（见§10.2）或者一个指向某个以空字符结尾的字符数组的指针。

文件流

#include<fstream> 为一个与文件关联的流定义输入-输出函数。

ifstream is(cp);	定义了 is，并把它连接到 cp 参数指定（对于不同的编译环境有不同的指定方式）的文件上。ifstream 类派生自 istream 类。
ofstream os(cp);	定义了 os，并把它连接到 cp 参数指定的文件上。ofstream 类派生自 ostream 类。
控制输出格式	
#include<ios>	定义了 streamsize 类型，这是一个带符号的整数类型，适于用来表示一个输入-输出缓冲区的大小。
os.width()	
os.width(n)	返回一个 streamsize 类型的值，表示与 os 相关联的输出流的宽度。如果函数给出 n 参数的话，就把输出流的宽度设置为 n。输出流的每一次输出都会反映到流的宽度上，输出完成后流的宽度被重置为 0。
os.precision()	
os.precision(n)	返回一个 streamsize 类型的值，表示与 os 相关联的输出流的精度。如果函数给出 n 参数，就把输出流的精度设置为 n 位精度。接下来输出的浮点值将被输出成指定精度位数的形式。
控制符	
#include<iomanip>	声明了像除 endl 以外的控制符，endl 控制符在<iostream>中声明。
os<<endl	结束当前行输出并刷新与 os 关联的流的缓冲区。
os<<flush	刷新与 os 关联的流的缓冲区。
os<<setprecision(n)	
os<<setw(n)	分别等价于 os.precision(n)和 os.width(n)。
错误与文件结尾	
strm.bad()	返回一个布尔类型的值，用来表示对 strm 的最近一次操作是否因为无效的数据而失败。
strm.clear()	在一个无效的操作之后重置 strm，使之可以再次使用。如果重置 strm 失败则抛出 ios::failure 异常。
strm.eof()	返回一个布尔类型的值，表示 strm 是否已经到了文件结尾处。
strm.fail()	返回一个布尔类型的值，用来表示对 strm 的最近一次操作是否因为硬件问题或者系统底层的问题而失败。
strm.good()	返回一个布尔类型的值，用来表示对 strm 的最近一次操作是否成功。

B.2 容器和迭代器

本书介绍了不同的容器和容器的性质，包括顺序容器 vector 和 list，关联容器 map stirng 类。所有容器在提供某些操作的时候都使用了相似的接口。本节先对各种容器的

作进行介绍，然后对个别容器提供的特殊的操作进行介绍。

如果没有特别的理由，程序员们应该在要用到顺序容器的时候使用 vector。有时候您可能想在一个容器的任意一个地方插入或者删除几个元素，而不仅仅是在容器的末尾进行这些操作，此时就应该考虑使用 list 容器，因为 list 类在进行这些操作的时候可以提供比 vector 类要方便而有效得多的函数。

B.2.1 共有的容器操作

所有容器与 string 类都提供下面的接口：

container<T>::iterator　一种与 *container*<T>关联的迭代器类型，该类型的对象可以用来改变存储在容器中的值。

container<T>::const_iterator　一种与 *container*<T>关联的迭代器类型，该类型的对象只能用来读取（而不能用来修改）存储在容器中的值。

container<T>::reverse_iterator
container<T>::const_reverse_iterator
一种用来沿相反的顺序访问容器中的元素的迭代器类型。

container<T>::size_type
一种无符号整数类型，它足够大，可以用来装下最大的容器。

container<T>::value_type　容器所储存的元素的类型。

c.begin()
c.end()
如果容器中装有元素的话，分别表示一个指向首元素的迭代器和一个指向末元素后面一个元素的迭代器。这两个函数的返回值可能是 c 容器的 const_iterator 类型，也可能是 c 容器的 iterator 类型，取决于 c 是否是一个常量。

c.rbegin()
c.rend()
用来沿相反的顺序访问 c 容器中的元素的迭代器，它具有 c 容器的 const_reverse_iterator 类型或者是 c 容器的 reverse_iterator 类型，具体类型由 c 是否是一个常量来决定。

container<T> c;　定义 c 为一个空的容器，这时候 c.size()的值为 0。

container<T> c2(c);　定义 c2 为一个容器，而且令 c2 容器的大小等于 c 容器的大小。c2 中的每一个元素都是 c 中相应元素的一个复件。

c=c2　复制c2容器中的元素到c容器中(覆盖c容器中相应的元素)，并返回 c 作为左值。

c.size()　c 容器中元素的个数。

c.empty()　如果 c 为空，则返回真值，否则返回假值。

c.clear()　清空 c 容器。与 c.erase(c.begin(),c.end())具有相同的作用。

执行完后，c 容器的大小变为 0。该函数返回 void 类型。

B.2.2 顺序容器的操作

除了提供上面列出的共有的容器操作以外，string 类与顺序容器（包括 vector 和 list 等容器）还支持下面的操作：

container<T> c(n,t);	定义 c，c 中包含 n 个元素，每个元素都是 t 的一个复件。
container<T> c(b,e);	定义 c 并对它进行初始化，初始化之后 c 容器的值为 b、e 两个迭代器所指序列内容的复件。
c.insert(it,t) c.insert(it,n,t) c.insert(it,b,e)	在 c 容器中 it 所指的位置前面插入一个新元素。如果 c 是一个容器或者 string 类型对象，那么在执行完该操作（或者所有会导致迭代器重定位的操作）以后，之前指向 c 容器（或者 string 类对象）的所有迭代器都会失效。注意，对于 vector 和 string，如果 it 迭代器指向的元素离末元素很远，这一操作可能会很慢。第一种形式的插入函数插入 t 的一个复件，返回一个指向新插入元素的迭代器。第二种形式的插入函数插入 t 的 n 个复件，然后返回 void 类型。第三种形式的插入函数向容器内插入 b 和 e 两个迭代器所指向的序列中的所有元素，然后返回 void 类型。其中 b 与 e 两个迭代器不能指向 c 中的元素。
c.errase(it) c.erase(b,e)	删除 c 容器中 it 指向的元素，或者删除 c 容器中[b,e)范围内的元素。该操作使所有指向被删除元素的迭代器无效。如果 c 是一个容器或者 string 类型对象，那么在完成删除操作后所有指向 c 中元素的迭代器都失效。函数返回一个指向被删除元素后面那个元素的迭代器。注意，如果被删除元素离 c 容器的末元素很远，该删除操作可能速度会很慢。
c.assign(b.,e)	把迭代器 b 与 e 指向的内容赋给 c 容器，c 容器原有的元素被覆盖。
c.front()	返回指向 c 容器首元素的一个引用。如果 c 是一个空容器则该操作的行为没有定义。
c.back()	返回指向 c 容器末元素的一个引用。如果 c 是一个空容器则该操作的行为没有定义。
c.push_back(t)	把 t 的一个复件加入 c 容器的末尾，该操作使 c 容器的大小增加 1。返回 void 类型。
c.pop_back()	从 c 容器中删除末元素。返回 void 类型。如果 c 是一个空

	则该操作的行为没有定义。
inserter(c,it)	返回一个输出迭代器，在 it 迭代器指向的元素前面向 c 容器插入新的元素。该函数在<iterator>中被定义。
back_inserter(c)	返回一个输出迭代器，可以通过调用 c.push_back 函数向 c 容器的末尾加入一个新的元素。该函数在<iterator>中被定义。

B.2.3 其他的顺序容器操作

下面的这些操作只在某些容器中提供，只有对这些容器才能进行下面的操作：

c[n]	一个指向 c 容器中第 n 个元素的引用，该容器的头元素是第 0 个元素。如果 c 是一个常量，那么该引用也是一个常量，否则就不是一个常量。如果 n 超出范围，那么该表达式将得到没有定义的行为。该表达式只对 vector 和 string 类对象有效。
c.push_front(t)	在 c 的起始处插入 t 的一个复件，c 容器的大小将增加 1。返回 void 类型。该表达式对 string 和 vector 类对象无效。
c.pop_front()	删除 c 容器的首元素。返回 void 类型。如果 c 是一个空的容器，则该函数的行为没有定义。该函数只对 list 类型对象有效。
front_inserter(c)	返回一个输出迭代器，可以通过调用 c.push_front 函数在 c 容器的起始位置插入新的元素。该函数在<iterator>中被定义。

B.2.4 关联容器的操作

关联容器被优化来快速访问容器中的元素，它是基于键来访问容器中的元素的。除了在 § B.2.1 列出的一般的容器操作以外，关联容器还提供下面的操作：

container<T>::key_type	容器键的类型。如果一个关联容器具有 K 类型键值和 V 类型元素，那么该联合容器的 value_type 类型不是 V，而是 pair<const K,V>。
container<T> c(cmp);	把 c 定义为一个空的关联容器，使用判断表达式 cmp 来为容器中的元素排序。
container c(b,e,cmp);	把 c 定义为一个关联容器，用迭代器 b 和 e 界定的元素序列对 c 容器进行初始化，并用判断表达式 cmp 来为容器中的元素排序。
c.insert(b,e)	向 c 容器中插入迭代器 b 和 e 界定的元素序列。对于 map 容器，该操作向容器插入那些 c 中不存在的键值的元素。
c.erase(it)	从 c 容器中删除 it 迭代器指向的元素。返回 void 类型。
c.erase(b,e)	从 c 容器中删除[b,e)范围内的元素。返回 void 类型。
c.erase(k)	从 c 容器中删除所有键值为 k 的元素。返回删除的元素个数。
c.find(k)	返回键值等于 k 的指向元素的迭代器。如果不存在这样的元素该函数返回 c.end()。

B.2.5 迭代器

标准库在很大程度上依赖于迭代器来实现其算法数据结构的独立性。迭代器是指针概念的抽象，类似于使用指针来访问数组中的元素，迭代器被用来访问容器中的元素。

在制定标准的算法的时候假设库被按照迭代器分成不同的类型。每一个使用特定种类迭代器的库算法只能用于提供该类型迭代器的标准库组件或者用户自定义的类。

- **输出迭代器：** 可以通过该迭代器来顺序地遍历整个容器，每次前进一个元素直到容器末尾，每次能而且也只能对一个元素进行一次输出操作。例如：ostream-iterator 类是一个输出迭代器，该类的 copy 函数要求用一个输出迭代器作为函数的第三个参数。
- **输入迭代器：** 可以用该迭代器顺序地遍历整个容器，每次前进一个元素，在前进到下一个元素之前可以按照需要对当前元素进行任意次读操作。例如：istream-iterator 类是一个输出迭代器，该类的 copy 函数要求用一个输出迭代器作为它的前两个参数。
- **正向迭代器：** 可以通过该迭代器来顺序地遍历整个容器，每次前进一个元素直到容器末尾，可以用该迭代器再次访问前一个迭代器指向的元素，而且可以按需要对该元素进行任意次读或写操作。例如：replace 就要求用前进迭代器作为参数。
- **双向迭代器：** 可以通过该迭代器双向地（既可以顺序地也可以逆序地）遍历整个容器，每次可以前进或后退一个元素。例如：list 和 map 类提供双向的迭代器，它们的 reverse 函数要求用一个双向迭代器作为参数。
- **随机访问迭代器：** 可以使用所有的指针支持的操作来访问容器中的某个元素。例如：vector，string 类和 C++自带的数组都支持随机访问迭代器。sort 函数要求以随机访问迭代器作为参数。

所有迭代器类型都支持判断是否相等的操作。随机访问迭代器还支持所有的关系操作。

一个迭代器可以具有几种迭代器类型，例如一个前进迭代器也同时是一个输出迭代器或者输入迭代器，每个双向迭代器同时是一个前进迭代器，而每一个随机访问迭代器同时是一个双向迭代器。因此，任意一个可以接收迭代器作为参数的算法都可以以随机访问迭代器作为参数。ostream_iterator 类提供输出迭代器，所以该类型的对象可以用在要求提供一个输出迭代器作参数的操作中。

所有的迭代器都支持下面的操作：

++p
p++　　让 p 指向容器中的下一个元素。然后把 p 作为一个左值返回；p++操作返回 p 的上一个值的一个复件。

*p　　p 所指向的元素。对于一个输出迭代器，*p 只能用作=运算符的左操作数，而且 p 的每一个不同的值只能以这种方式使用一次。对于输入迭代器，*p 可以用来读取元素的值；而增加 p 的操作会使组成 p 的旧值的所有复件都无效。对于所有其他类型的迭代器，*p 生成一个引用，该引用指各 p 指向的容

值，只要 p 指向的元素还存在，p 就一直有效。

p==p2　　如果 p 等于 p2 就生成一个真值；否则生成一个假值。

p!=p2　　如果 p 不等于 p2 就生成一个真值；否则生成一个假值。

除了输出迭代器以外所有的迭代器都支持下面这个操作

p->x　　等价于（*p）.x

双向迭代器与随机访问迭代器还支持自减一的操作：

--p

p--　　p 自减一，指向容器中当前所指元素的前一个元素。--p 在完成减一后返回新的 p 值作为左值；p--返回 p 的旧值的一个复件。

随机访问迭代器提供所有类似于指针的操作，列于下文：

p+n　　如果 n>=0，那么结果是一个指向 p 所指元素后面第 n 个元素的迭代器。如果在 p 所指元素后面只有少于 n-1 个元素，那么该操作将没有定义。如果 n<0，那么结果是一个指向 p 所指元素前面第 n 个元素的迭代器，除非 p+n 所指元素在容器的范围之内，否则该操作没有定义。

n+p　　等价于 p+n 操作。

p-n　　等价于 p+（-n）操作。

p2-p　　只有当 p2 与 p 指向同一个容器中的元素的时候才有定义。如果 p2>=p，那么返回[p,p2)范围内的元素个数。否则返回[p2,p)之间的元素个数的相反数。结果具有 ptrdiff_t 的类型（见§10.1.4）。

p[n]　　等价于*（p+n）。

p<p2　　如果 p 与 p2 都指向同一容器中的元素，并且 p 指向的元素在 p2 指向的元素之前，该表达式的值为真，否则为假。如果 p 和 p2 不是指向同一个容器中的元素，则该表达式没有定义。

p<=p2　　等价于（p<p2）||（p==p2）。

p>p2　　等价于 p2<p。

p>=p2　　等价于 p2<=p。

B.2.6 向量（vector）

向量（vector）类提供进行动态内存分配的，具有任意类型元素的数组，而且支持随机访问迭代器。除了前面所说的公共的容器操作（见§B.2.1）和连续容器的操作（见§B.2.2）以外，向量还提供下面的操作：

#include<vector>　　声明了 vector 类及其相关的操作。

v.reserve(n)　　为 v 重新分配内存，使 v 的空间增大，不需要再分配新的内存空间也能装下至少 n 个元素。

v.resize(n)　　为 v 重新分配内存，使之只可以装下 n 个元素。该操作使所有

中的元素的迭代器都无效。保存向量中的前 n 个元素。如果新的容器大小比原来的要小，那么多出来的元素将被删除。如果新的容器大小比原来的要大，那么新分配元素的内存将被值初始化（见§9.5）。

B.2.7 链表（list）

链表（list）类提供动态分配内存的，具有任意类型元素的，双重连接的列表，并且支持双向迭代器（与 vector 类不同，vector 类支持随机访问迭代器）。除了前面所说的公有的操作和顺序容器的操作（见§B.2.1 和§B.2.2）以外，list 类还支持下面的操作：

#include<list>	声明了 list 类及相关的操作。
l.splice(it,l2)	把 l2 的所有元素插入到 l 中 it 指向的元素前面，并且删除 l2 中的这些元素。该操作使 l 的所有迭代器以及 l2 的所有引用都失效。该操作结束后，l.size()等于原来 l 和 l2 的大小之和，而新的 l2 的大小变为 0。该函数返回 void 类型。
l.splice(it,l2,it2)	
l.splice(it,l2,b,e)	在 l 中 it 指向的元素的位置前面插入 l2 中 it2 指向的元素，或者[b,e)范围内的元素序列，然后从 l2 中把相应的元素删除。it2 所指向的元素，或者[b,e)范围内的元素都必须在 l2 中。该操作使所有指向接头元素的迭代器和引用都无效。返回 void 类型。
l.remove(t)	
l.remove_if(p)	从 l 中删除所有等于 t 的元素，或者删除所有令 p 判断为真的元素。返回 void 类型。
l.sort(cmp)	
l.sort()	对 l 中的元素进行排序。使用<，或者用参数中提供的判断 cmp 来对元素进行比较。

B.2.8 字符串（string）

string 类提供可变长度的字符串，并提供一个随机访问迭代器来访问字符串中的任何一个字符。虽然 string 类对象不是真正的容器，但是它们支持在前面列出的容器操作（见§B.2.1 和§B.2.2），并且可以用在后面说讲的算法（见§B.3）中。除此之外，string 类还支持下面的操作：

#include<string>	声明 string 类及相关的操作。
string s(cp);	把 s 定义成一个 string 类对象，并把它初始化成 cp 所指的字符串的一个复件。
os<<s	把 s 中的字符输出到 os 中，返回一个指向 os 的引用。
is>>s	从 is 中读取一个量到 s 中，覆盖 s 中原来的内容。返回一个指

	引用。is 中的词用空白字符（包括空格符、制表符和换行符）来隔开。
getline(is,s)	从输入流 is 中读取一行字符（包括换行符），并且把这行字符（包括换行符）保存到 s 中，覆盖 s 中原来的内容。返回一个指向 is 的引用。
s+=s2	把 s2 字符串加到 s 字符串的末尾，并返回一个指向 s 的引用。
s+s2	返回一个 s 字符串与 s2 字符串连接后的值。
s *relop* s2	返回一个布尔值，表示这个关系操作是否为真。string 库定义了所有的关系运算符和等式运算符：<、<=、>、>=、==和!=。如果两个字符串类型对象中的每个元素都相等，那么这两个字符串类型对象也相等。如果一个字符串类型对象是另一个字符串类型对象的前面一部分，那么这个短的字符串类型对象就小于长的那个字符串类型对象。否则，该关系式的值将由两个字符串中第一个不相等的字符进行比较得到。
s.substr(n,n2)	返回一个新的字符串类型对象，该对象中的字符串由 s 对象中第 n 个开始的 n2 个字符组成。如果 n>s.size()，那么该函数的行为为未定义。如果 n+n2 比 s 对象的 s.size()要大，那么就返回由 s 对象中从 n 个字符到最后一个字符组成的新的字符串类型对象。
s.c_str()	生成一个指向一个空字符结尾的字符数组的常量指针，该数组中包含 s 中的字符的一个复件。在对 s 进行下一个非常量类型的成员函数调用之前，该数组一直存在于内存中。
s.data()	与 c_str 函数类似，不过该函数中的数组不是以空字符结尾。
s.copy(cp,n)	从 s 中复制前 n 个字符（不包括空结束符），然后储存到 cp 指向的字符数组中。调用该函数的时候必须保证 cp 指向的字符数组必须有至少能储存 n 个字符的空间。

B.2.9 对（pair）

pair<K,V>类用来模拟由一个 K 类型的值与一个 V 类型的值组成的对。pair<K,V>类支持以下操作：

#include<utility>	声明 pair 类与相关的操作。
x.first	一个名为 x 的 pair 类对象的第一个元素。
x.second	x 对的第二个元素。
pair<K,V> x(k,v);	把 x 定义为一个新的对，这个对由一个 K 类型的值 k 和一个 V 类型的值 v 组成，所以 x.first 具有 K 类型，而 x.second 具有 V 类型。注意，为了以这种形式显式地声明一个对，你必须知道其元素的类型。
make_pair(k,v)	生成一个新的 pair<K,V>类对象，该对象具有 k 和 v 两个元素。在使用这个函数时不需要知道 k 和 v 的类型。

B.2.10 图（map）

map 类提供动态分配内存的，具有独立类型元素的关联数组。在该关联数组中用到 pair 类作为辅助类以保存（名，值）对，这个对也是图的元素。图中提供双向迭代器。每个 map 类型对象中都储存有与 const K 类型的关键字相关联的 V 类型的值。因此，图中某个元素保存的值可以改变，但是键不能被改变。除了前面讲过的公共的容器操作（见§B.2.1）和联合容器特有的操作（见§B.2.4）以外，图还支持下列操作：

#include<map>	声明 map 类及相关的操作。
map<K,V,P> m(cmp);	把 m 定义成一个新的空图，可以用这个图储存与 const K 类型的键相关联的 V 类型的值，在把这些值插入图中的时候用 P 类型的 cmp 判断来比较新元素。
m[k]	生成一个 m 中由 k 索引的元素的一个引用。如果在 m 中没有该元素，那么该操作将向 m 中插入一个值初始化的 V 类型的元素（见§9.5）。因为在运算 m[k]的时候可能会隐式地改变 m 的内容，所以 m 必须是一个常量。
m.insert(make_pair(k,v))	在 m 中键 k 指示的位置插入 v 值。如果在键 k 指示的位置已经有一个值存在，那么相应的值将不会被改变。返回一个 pair<map<K,V>::iterator,bool>类型的对，这个对的第一个元素是指向 k 键值指示的元素的引用，而这个对的第二个元素是一个布尔型值，用来表示是否有一个新的元素被插入图中。注意，make_pair 函数生成一个 pair<K,V>对，在调用 insert 函数的时候这个对被强制转换成 pair<const K,V>类型的对。
m.find(k)	返回一个指向键 k 指示的元素的迭代器。如果这个元素不存在，那么该函数返回 m.end()。
*it	在 it 迭代器指向的位置生成一个 pair<const K,V>对，这个对包括键与键关联的值。因此，it->first 代表键，具有 const K 类型，而 it->second 代表键对应的值，具有 V 类型。

B.3 算法

标准库中提供了很多常用的算法，这些算法通过迭代器来获得算法所处理的特定的数据结构的独立性以及在该数据结构中存储的类型。注意，在联合容器中具有指向象 pair<const K,V>这样的混合类型的迭代器，所以在对这些联合容器应用算法时要特别小心。

大部分对序列操作的算法都提供两个迭代器来访问数据，第一个迭代器通常指向序列的首元素，第二个元素指各序列的末元素后面的一个元素。除了特别指出的某些算法，其他

都在<algorithm>头文件中定义。

#include<algorithm> 包括常用算法的声明。

accumulate(b,e,t)

accumulate(b,e,t,f) 在<numeric>头文件中定义。生成一个临时对象 *obj*，这个临时对象与 t 具有相同的类型和值。对于[b,e)范围内的每个输入迭代器 it，计算 *obj*=*obj*+*it 或者 *obj*=f(*obj*, *it)，具体进行哪一个计算决定于 accumulate 函数以哪一种形式调用。结果是 *obj* 对象的一个复件。注意，因为+可能被重载，所以即使是 accumulate 函数的第一种形式也可能对非 C++自带的算术类型进行操作。例如，我们可以用 accumulate 函数来连接一个容器中的所有字符串。

binary_search(b,e,t) 返回一个布尔类型的值，表示 t 的值是否存在于由两个正向迭代器 b 和 e 所界定的序列中。

copy(b,e,d) 把由输出迭代器 b 和 e 界定的序列中的值复制到目标容器中，目标容器由输出迭代器 d 指定。该函数假定在目标容器中有足够的空间存储复制过来的值。返回一个值指向目标容器末元素后面一个元素。

equal(b,e,b2)

equal(b,e,b2,p) 返回一个布尔类型的值，表示两个序列中的元素是否相等，第一个序列由输入迭代器 b 和 e 界定；第二个序列是从输入迭器 b2 指向的元素开始的与第一个序列等长的一个序列。在进行判断的时候使用关系符 p，如果函数没有提供 p 参数，则用= =关系符。

fill(b,e,t) 把由输入迭代器 b 和 e 界定的序列的值设为 t。返回 void 类型。

find(b,e,t)

find_if(b,e,p) 返回一个迭代器，指向一个序列中首次出现 t 值的元素，或者指向一个序列中首次满足 p 表达式的元素（如果函数中提供了 p 作为参数的话），该序列由输入迭代器 b 和 e 界定。如果没有满足条件的元素存在则返回 e。

lexicographical_compare(b,e,b2,e2)

lexicographical_compare(b,e,b2,e2,p)

返回一个布尔类型的值，表示由[b,e)界定的序列是否小于[b2,e2）界定的序列，在进行元素比较的时候使用关系符 p，如果函数中没有提供 p 参数，则用<关系符进行比较。如果一个序列是另一个序列前面的一部分，那么认为这个短序列比长序列要小。否则，对两个序列中第一对不相等的元素进行比较以得出结果。b，e，b2 和 e2 必须是输入迭代器。

max(t1,t2)
min(t1,t2)

返回 t1 和 t2 中的较大（对于 max 函数）的那个或者是较小（对于 min 函数）的那个，t1 和 t2 必须是同样类型的值。

max_element(b,e)
min_element(b,e)

返回两个序列中的较大者（或者较小者），这两个序列分别由正向迭代器 b 和 e 所指向。

partition(b,e,p)
stable_partition(b,e,p)

把由双向迭代器 b 和 e 界定的序列分成两部分，把满足表达式 p 为真的元素放在容器的前面，不满足 p 为真的元素放在容器的后面。返回一个指向第一个不满足 p 为真的元素的迭代器，如果对于所有元素都满足 p 为真，则返回 e。stable_partition 函数使两部分中的元素都保持原来在容器中的先后顺序。

remove(b,e,t)
remove_if(b,e,p)

重新排列由正向迭代器 b 和 e 界定的序列中的元素，以使序列中与 t 不匹配的元素，或者使条件式 p 为假（如果函数提供 p 参数的话）的元素从相关的序列开始处排列。返回一个指向没有移动的最后一个元素后面一个元素的迭代器。

remove_copy(b,e,d,t)
remove_copy_if(b,e,d,p)

类似于 remove 函数的功能，不过该函数把与 t 不匹配的元素，或者使条件式 p 为假（如果函数中提供 p 参数的话）的元素，复制到由输出迭代器 d 指向的目标容器中。返回一个指向目标容器末元素后面一个元素的迭代器。该函数假定目标容器足够大以装下复制过来的元素。由迭代器 b 和 e 界定的序列中的元素不会被移动，因此，我们只需要 b 和 e 是输入迭代器就可以了。

replace(b,e,t1,t2)
replace_copy(b,e,d,t1,t2)

在由正向迭代器 b 和 e 界定的序列中，把所有等于 t1 的元素都替换成 t2 的值。返回 void 类型。第二种形式的函数在用 t2 值替换等于 t1 的元素后，把新序列的元素复制到由输出迭代器 d 指向的容器中，并返回一个指向目标容器末元素后面一个元素的迭代器。第二个函数只要求 b 和 e 是输入迭代器类型就可以了。

reverse(b,e)
reverse_copy(b,e,d)

第一个函数通过交换元素一个序列中的所有元素进行倒序排列，该序列由双向迭代器 b 和 e 界定，然后返回 void 类型。第二个函数把倒序后的序列保存到输出迭代器 d 指向的目标容器中，然后返回一个指向目标容器末元素后面一个元素的迭代器。该

定目标容器中具有足够大的空间，可以装下复制过来的元素。

search(b,e,b2,e2)
search(b,e,b2,e2,p)

在由正向迭代器 b 和 e 界定的序列中，查找由正向迭代器 b2 和 e2 界定的子序列，然后返回指向第一个匹配的子序列首元素的正向迭代器。如果函数提供 p 参数，那么在比较的时候使用 p 关系符，否则用==关系符进行比较。

transform(b,e,d,f)
transform(b,e,b2,d,f)

如果函数没有提供 b2 参数，那么 f 函数必须带有一个参数；在调用 transform 函数时先调用 f 函数，f 函数以由输入迭代器 b 和 e 界定的序列中的元素为参数。如果函数中提供 b2 参数，那么 f 函数必须带有两个参数，在调用 transform 函数的时候先调用 f 函数，f 函数以 b 和 e 界定的序列和以 b2 指向的等长序列为参数。任何一种情况下，transform 函数都把生成的结果序列储存到由输出迭代器 d 指向的目标容器中，并返回一个指向目标容器末元素后面一个元素的迭代器。该函数假定目标容器足够大，可以装下生成的序列中的元素。注意，在第二个函数中，d 可以等于 b 或者 b2，这时候得到的结果序列将覆盖原来的输入序列。

sort(b,e)
sort(b,e,p)
stable_sort(b,e)
stable_sort(b,e,p)

对由随机迭代器 b 和 e 界定的序列进行排序。在排序判断中使用 p 关系符，如果函数中没有提供 p 参数则用<做判断。stable_sort 函数在排序时，对于相等的各元素保持原来的顺序。

unique(b,e)
unique(b,e,p)

重新排列由正向迭代器 b 和 e 界定的序列，使连续相等元素组成的子序列中的第一个元素被移到容器的起始处。返回一个迭代器，该迭代器指向第一个不满足要求的元素（如果输入元素中的所有连接对都不相等，则返回 e）。如果函数中提供 p 参数，则使用 p 关系符进行判断，否则用==关系符进行判断。

unique_copy(b,e,d,p)

复制由输入迭代器 b 和 e 界定的序列，然后存放到由输出迭代器 d 指向的序列中，该过程会覆盖所有的相邻的副本。在使迭代器 d 增加与被复制的元素个数相同的整数后，函数返回 d。该函数假定 d 指向的容器足够大，可以装下复制过来的元素。如果函数提供 p 参数，则使用 p 关系符进行判断，否则使用==关系符进行判断。

Accelerated C++ 中文版

Practical Programming by Exmple

这是一本一流的C++入门书，它采用了一种和实践相结合的方式来解决具体的问题。相比我所见过的其他C++入门书来说，本书以令人惊奇的紧凑格式覆盖了更多的关于C++编程的领域。

——Dag Bruck，ANSI/ISO C++委员会成员

通过让学生尽快地编写具有实际意义的程序，作者在本书中向我们展示了一种清晰的、令人信服的C++教学方式。

——Stephen Clamage，Sun Microsystems, Inc.，ANSI C++委员会主席

所有读过该书并完成上面的例子和习题的人都将获得和大多数专业C++程序员一样的技能。

——Jeffrey D. Oldham，Stanford University

为什么《Accelerated C++》给人的印象会如此深刻呢？这是因为：

- 它一开始就向读者教导那些最有用的概念，而不是那些简单的注释
  读者很快就能够以此开始编程
- 它描述的是现实中的问题和解决方案，而不是单纯的语言特性
  读者不但学习到了这些特性，而且还知道如何把它们应用到程序中去
- 它涵盖的范围同时包括了语言本身和标准库
  读者可以从一开始就使用标准库来编写自己的程序

作者通过他们在美国斯坦福大学的教学经验证明了这种方法的有效性。在那里，学生们在他们的第一堂课中就学习到了如何编写真实的程序。

不管你是一个渴望开始学习C++编程的新手，还是一个已经使用C++多年并对它有了很深了解的老手，作者们独特的教学方法和经验都使得本书应该成为你书架中不可缺少的一个补充。

Andrew Koenig 是AT&T公司Shannon实验室大规模编程研究部门中的成员，同时他也是C++标准委员会的项目编辑。他拥有超过30年的编程经验，其中有15年的C++使用经验。他已经出版了超过150篇和C++有关的论文，并且在世界范围内就这个主题进行过多次演讲。他同时还是《C Traps and Pitfalls》一书的作者，并协同妻子Barbara E. Moo合作出版了另外一本书籍：《Ruminations on C++》(《C++沉思录》)。

Barbara E. Moo 是一个在软件领域中拥有超过20年经验的独立咨询顾问。在AT&T工作的近15年中，她参与了第一个使用C++编写的商业产品的开发，领导了公司中第一个C++编译器项目，管理了AT&T中广受赞誉的WorldNet Internet service business的开发。她同时也是《Ruminations on C++》的作者之一，并且和丈夫Andrew Koenig一同在世界范围内进行C++的教学活动。

ISBN 7-5083-1819-6

9 787508 318196 >

责任编辑/陈维宁
封面设计/王红柳

ISBN 7-5083-1819-6
定价：39.50